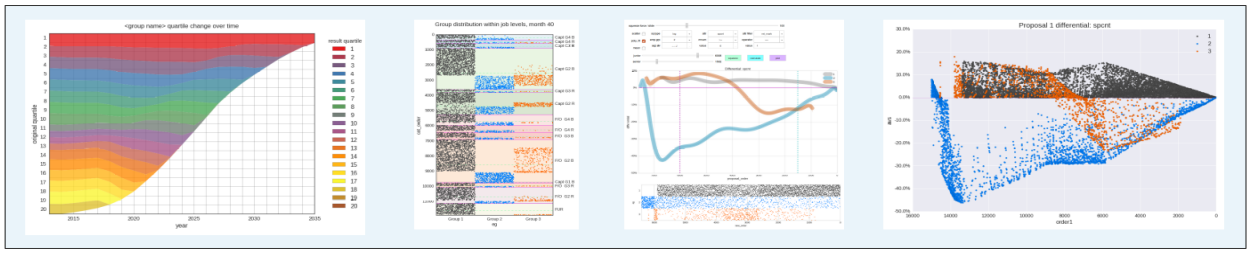
seniority_list Documentation

Release 0.65

Robert E. Davison

May 13, 2020

CONTENTS



Quick links:

- The introductory article recently published[1] through Cornell University
- YouTube[2] channel
- Contact[3]

Welcome to seniority_list!

seniority_list is an analytical tool used when seniority-based work groups merge. It brings modern data science to the area of labor integration, utilizing the powerful data analysis capabilities of Python scientific computing. While the software was developed with an initial focus on the airline industry, seniority_list is adaptable to any industry or group where workers operate under a seniority system.

seniority_list offers an unbiased, numerical method to measure and compare the outcome of proposed combined work group seniority lists. It is able to quantify how the careers of workers would be affected under various seniority list orderings and conditions in ways that have been difficult to measure previously.

seniority_list works by generating detailed data models (datasets) for various integration scenarios as described within a few basic Excel* spreadsheets prepared by the user. The datasets may then be thoroughly analyzed with many customizable, built-in visualization functions and statistical reports, or other user-defined methods. The program is also able to construct and modify lists in near real time, with full outcome results produced within seconds.

seniority_list does not attempt to predict the bidding preferences of individual employees. Instead, the program focus is on utilizing variables that are fixed or that can be modeled in a quantifiable state, such as birth dates, jobs available, proposed list orderings, furlough recall schedules, and special job assignment conditions or restrictions. The model is based on the assumption that all employees will bid for the highest paying or highest ranked jobs at all times. In reality, employees will make choices based on individual situations. However, the overall result of these individual choices is a group average, ultimately constrained by list positioning. seniority_list models the effect of list ordering combined with other customizable factors to provide useful, objective information for interested parties.

[1] http://scholarship.sha.cornell.edu/chrpubs/246/
[2] https://www.youtube.com/channel/UCO7bj5LkSGFXUoqfp2wRMgQ
[3] http://www.rubydatasystems.com/contact.html#contact

A complete example case study including sample input data and analysis examples is included with seniority_list.

Compared to tools which may have been used in the past, seniority_list offers:

- *speed* - easily modify parameters, rerun, and generate new comprehensive reports within a few minutes
- *flexibility* - wider range of data analysis through numerous function parameters and input file settings and options
- *conditional modeling* - accurately model "no bump, no flush" job bidding/assignment, job assignment conditions and restrictions, changes in number and category of jobs available for bid over time, and furlough and recall
- *additional job granularity* - part/full-time sections within common job compensation levels permits additional precision
- *financial studies* - the model incorporates compensation data allowing individual career and cumulative group analysis and comparison
- *extensive statistical evaluation* - the entire Python "scientific stack" may be utilized to evaluate list and outcome metrics
- *advanced visualization* - an extensive range of chart types and features is readily available through various Python and javascript libraries
- *accuracy* - designed with enterprise-level Python data science libraries and methods
- *interactive list editing* - the editor tool allows list adjustments to be made and the results viewed within seconds
- *easy adaptation* - the design of the program and the simple data input interface via spreadsheets makes it easy to use seniority_list with many different integration cases
- *open source* - all programming code is open and available for examination and usage

---

Note: seniority_list was developed independently. The program is not affiliated with any labor or industry organization and is licensed under the GNU General Public License v3.0. Please direct consulting inquiries to rubydatasystems@fastmail.net.

Images in the web version of this documentation may be clicked for a full-size view.

*"Excel" is defined to mean the Microsoft Excel® spreadsheet program or any other spreadsheet program which is compatible to .xlsx spreadsheet files, such as Calc[4].

[4] https://www.libreoffice.org/discover/calc/

Part I

seniority_list

CHAPTER

ONE

FEATURES

- Examine and compare pre- and post-integration lists and calculated outcomes with **statistics and charts** over a wide range of metrics, on an **individual or group** basis
- Analyze integrated list **outcome models** using any of the multiple attributes within the calculated datasets including time period selection, monthly and career compensation, job level granularity, and position percentage within job levels
- Slice and group datasets by any dimension for additional insight into the **real effects** of integration proposals
- Model and compare various pre- and post- job assignment **special conditions** within proposed integrated lists
- Model **job count changes** on a per job category, per month basis
- Model **furlough and recall**
- Model an increase(s) in **retirement age**
- Incorporate **delayed implementation** of list integration with smooth transition from separate to combined operation
- Switch easily between **basic and enhanced job level studies**
- Model "full flush" or "no bump no flush" **rules for job assignment** modeling
- Study financial **compensation metrics** with or without an assumed increase (or decrease) in pay rates following contract expiration
- Analyze combinations of **any number of employee groups**
- Produce **customized results** for any subset of employee groups
- Generate complete **summary reports** for all proposals in a matter of minutes
- **Share summary reports** by copying a single output folder
- Recalculate datasets in near **real-time** when inputs are modified
- Experiment easily with **"what-if" scenarios**

- Provide user data to seniority_list via a basic **Excel spreadsheet interface**
- **Identify differences** in data values which may exist between Excel spreadsheets submitted by the parties
- **Edit proposed lists intuitively using an interactive visual interface** and see the recalculated results almost immediately
- **Build "hybrid" lists** using a hierarchy of attribute priorities
- **Reinsert inactive employees** into the integrated list prior to producing a final list result in Excel format
- Create/save/share **publication-quality visualizations** utilizing a variety of chart types, format styles, and/or color mappings

seniority_list includes considerable analysis capability through a comprehensive set of built-in plotting functions designed to be applied to the calculated datasets. The user is free to explore the model datasets with custom functions as well.

CHAPTER

TWO

PROGRAM NOTES

- seniority_list is written in the Python 3[5] programming language
- The project was initiated in October of 2015 with the first version complete in April of 2016
- Software development is performed within the interactive Jupyter[6] notebook and the Sublime Text 3[7] editor.
- seniority_list primarily uses the pandas[8] and NumPy[9] libraries for computation
- The program uses the Python matplotlib[10], seaborn[11] and bokeh[12] libraries for data visualization
- Python pickling is utilized for fast dataset storage and retrieval
- This documentation website was produced with the Sphinx[13] documentation generator along with the Shutter screenshot tool and the yEd Graph Editor[14].

Basic knowledge of Python is required. The seniority_list program code is open-source and available here[15].

[5] https://www.python.org/
[6] http://jupyter.org/
[7] https://www.sublimetext.com/
[8] http://pandas.pydata.org/
[9] http://www.numpy.org/
[10] http://matplotlib.org/
[11] https://stanford.edu/~mwaskom/software/seaborn/
[12] https://bokeh.pydata.org/en/latest/
[13] http://www.sphinx-doc.org/en/stable/#
[14] https://www.yworks.com/products/yed
[15] https://github.com/rubydatasystems/seniority_list/

CHAPTER

THREE

INSTALLATION

The software and program files necessary to run the seniority_list program are free to download and use.

3.1 dependencies

Python 3*

Python libraries

- bokeh[16] *
- bottleneck[17] *
- Ipython[18] *
- matplotlib[19] *
- numba[20] *
- numexpr[21] *
- NumPy[22] *
- openpyxl[23] *
- pandas[24] *
- python dateutil[25] *
- SciPy[26] *

[16] https://bokeh.pydata.org/en/latest/
[17] https://pypi.python.org/pypi/Bottleneck
[18] https://ipython.org/
[19] http://matplotlib.org/
[20] http://numba.pydata.org/
[21] https://numexpr.readthedocs.io/en/latest/index.html
[22] http://www.numpy.org/
[23] https://openpyxl.readthedocs.io/en/stable/
[24] http://pandas.pydata.org/
[25] http://labix.org/python-dateutil
[26] https://scipy.org/scipylib/

seaborn[27] *
xlrd[28] *
xlsxwriter[29] *
xlwt[30] *

Jupyter notebook*

* included with anaconda[31]

seniority_list is designed to use the Jupyter Notebook[32] (notebook) as its user interface. The notebook is required to run the interactive *editor*.

The name "Jupyter" is a loose acronym referring to the Julia, Python, and R programming languages. The notebook supports these and many other languages.

Information concerning the Jupyter notebook may be found here[33]. There are numerous guides and tutorial videos online as well. Specific details pertaining to notebook usage with seniority_list are located in the "user guide" section.

spreadsheet program

A spreadsheet program compatible with .xlsx files is required to handle the input data and to read output from the program. Microsoft Excel® may be used with seniority_list, but is not required. LibreOffice Calc[34] is an open-source spreadsheet program which works well with seniority_list. LibreOffice is free and may be downloaded here[35].

3.2 installing Python and Python libraries

Your computer must have the Python program and the associated Python libraries (helper programs which perform specialized tasks) as listed above on your computer to be able to run seniority_list. Nearly all of these requirements are met with one download and installation of the Anaconda scientific platform. Navigate to this webpage[36] and select the Python 3.6 (or above) and 64-bit version appropriate for your operating system. Install, using the default prompts. Detailed installation instructions are found here[37], if needed.

[27] http://seaborn.pydata.org/
[28] https://xlrd.readthedocs.io/en/latest/
[29] https://xlsxwriter.readthedocs.io/
[30] https://xlwt.readthedocs.io/en/latest/
[31] https://www.anaconda.com/why-anaconda/
[32] http://jupyter.org/
[33] http://jupyter.org/
[34] https://www.libreoffice.org/discover/calc/
[35] https://www.libreoffice.org/download/download/
[36] https://www.anaconda.com/distribution/
[37] https://docs.anaconda.com/anaconda/install/

Note: The anaconda download is 350-600mb depending on your operating system and will require approximately 3gb of disk space when it is installed.

At this point, the Python program files necessary to run seniority_list have been installed. The next step is to download and install the actual seniority_list program.

3.3 installing seniority_list

The seniority_list program and sample data files are downloadable from a GitHub repository. GitHub is a widely used hosting service for open source software projects. A repository is a container which holds code and other files relating to a project. The repository address for seniority_list is:

https://github.com/rubydatasystems/seniority_list.git

The easiest way to obtain the seniority_list program files from GitHub is to install the git[38] program, which has a built-in method to grab files from GitHub repositories with one command line input.

git may be installed through the Anaconda platform. To install git, type or copy and paste the following command into a terminal:

```
conda install git
```

Once git is installed, open a terminal and type or copy and paste the following command:

```
git clone https://github.com/rubydatasystems/seniority_list.
↪git
```

The git program will retrieve and install all of the program and sample data files from the GitHub repository (< 15mb).

Note: When running seniority_list with a Windows operating system, references to the terminal apply to the Anaconda Prompt, opened from the Start Menu. Linux and MacOS users may use the standard terminal prompt.

---

[38] https://git-scm.com/

CHAPTER

FOUR

OPERATIONAL OVERVIEW

4.1 abstract

The program models employee future career progression, reflecting standalone and integrated list proposals, and stores this information in files known as datasets. The datasets are analyzed and compared across a broad range of attributes. This process provides objective, outcome-based analytics for integration decision-makers.

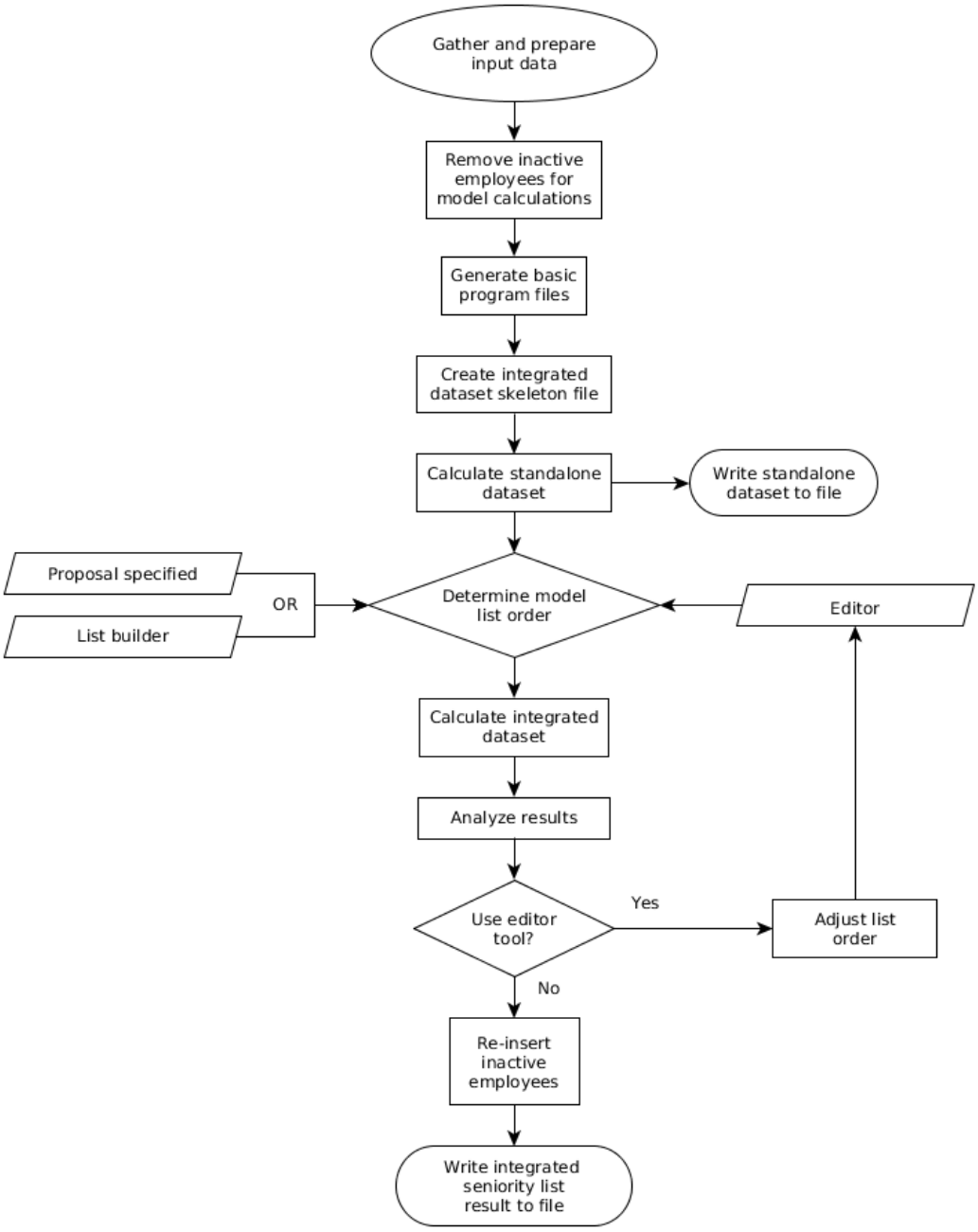


Fig. 1: high-level dataset production diagram (click to enlarge)

4.1.1 basics

The seniority_list program generates a data model built upon predictable variables while isolating inputs that cannot be directly quantified.

In other words, certain aspects or parameters are known and unlikely to change, while others are likely or sure to change. The factors that are known or predictable are incorporated within the model calculations. The unpredictable factors are handled equally (controlled) for each group so that their influence upon accuracy of the model is minimized or eliminated.

Examples of predictable variables include job counts, retirement counts, and pay scales. Unpredictable variables include individual bidding choices and future employee work leaves.

With the effect of the unpredictable variables removed, the results of the calculations will be directly related to the predictable variable inputs. List order is the primary predictable variable and has by far the most influence on the resultant datasets.

By default, seniority_list constructs the job level hierarchy in accordance with compensation scales and assumes that all employees will continuously bid for the highest paying job. Consequently, the resultant employee career metrics produced by the program reflect and focus primarily on the true effect of the ordering of proposed integrated lists. However, the data model job level hierarchy (used by the program job assignment routines) may be set by the user to match the requirements of specific case studies when necessary.

4.2 quick outline of seniority_list

The information below will cover the basics of seniority_list - an overview of how the program works and what it does.

The "user guide" section of the documentation will provide much more detailed discussion and instruction after basic program concepts are introduced here.

4.2.1 gather and prepare data

Before seniority_list can begin, it must be able to read specifically formatted input from within designated project folders. Therefore, the first step for the user is to aquire the required data and to format and store it properly.

seniority_list is designed to read all user input data directly from Excel spreadsheets. This information includes general employee lists, compensation data, proposed inte-

grated lists, job assignment schedules, and many other miscellaneous user-specified options.

During the initial phase of data preparation, any differences between input lists must be resolved, such as the number and status of employees or the number of jobs available in each category. The *list_builder* module contains functions useful for rapidly finding differences between spreadsheets.

Excel workbook data sources

seniority_list uses four Excel workbooks as source data when creating the foundational program files. Detailed guidance concerning the content and formatting requirements of the input workbooks is presented in the "excel input files" section of the documentation.

Each case study will require the following four workbooks to be placed within a case-specific folder located within the program's "excel" folder.

master.xlsx

- This is the workbook which contains general employee data. This single-worksheet workbook contains approximately 10 columns of data for every employee.

pay_tables.xlsx

- The compensation information will likely require the most formatting in terms of the worksheet layout and data preparation. Specific worksheet naming and formatting is required for two worksheets, one containing hourly pay rates for each job level/longevity combination, and another listing the total monthly pay hours for each job category.

proposals.xlsx

- List order proposals are stored in the third workbook with a separate worksheet for each ordering proposal. These worksheets contain only two columns: order number and employee number (empkey).

settings.xlsx

- This workbook stores data related to program options and schedules. It also contains some plotting function values concerning labels and colors for output charts.

other list order sources

List proposals submitted by parties are normally stored and read from the Excel proposals.xlsx input file.

There are two other ways to prepare and provide list order input to the program:

1. New list ordering may be generated by utilizing the functions within the **list_builder** script.
2. Modifications to any list ordering may be accomplished by utilizing the interactive list editor tool.

4.2.2 build the basic program files from the input data

seniority_list begins by creating certain files needed by the program for dataset generation and other operations.

As mentioned above, seniority_list is able to directly read and write Excel files. However, it is magnitudes faster to use a different format optimized for Python when retrieving and storing data for internal program operation. Therefore, each input Excel file is read once, converted to a pandas dataframe, and then stored as a serialized "pickle" file for further use within seniority_list.

seniority_list also modifies the structural format of the input files as necessary during the conversion process. The format modification allows for fast and efficient data indexing and access during program operation. For example, the compensation data will be converted from a wide-form, spreadsheet-style table to an indexed long-form format, while the master list file will be stored in a nearly identical row and column format as the original.

A few helper program files derived from the input files are calculated and stored during this process as well.

4.2.3 create the "skeleton"

Note: pandas (with a small "p") is a powerful Python library (add-on program) used for data analysis work. pandas, along with a number of other specialized Python libraries, is used extensively within the seniority_list code base. The primary data structure provided by the pandas library is the dataframe. The dataframe can be described as an in-memory tabular structure similar to a spreadsheet, but far more capable and powerful, especially when combined with other Python tools.

A dataset is the calculated data model resulting from a particular integrated list ordering proposal and its associated conditions. The skeleton provides the starting point or frame for the creation of a complete dataset. Each case has one unique skeleton, just as each case has its own set of employees and lists.

The skeleton is a "long-form" pandas dataframe containing calculated data derived from the basic "short-form" master list data.

- "Short-form" refers to a dataset containing static list data, without any future progression calculations. It's length is equal to the number of employees.
- "Long-form" refers to a dataset that contains information for every month for each employee remaining on the list (not retired) for that month.

The skeleton contains many columns of data, most of which is general employee data relating to specific employees and months. All of the pre-calculated information contained within the skeleton is independent of and unaffected by changes in list order. The data includes such things as hire date, month number, employee group number, age, and retirement date.

The skeleton forms the foundation or starting point for the production of all datasets pertaining to a particular case. The number of rows in the skeleton and in the final dataset is the same, but many additional columns of calculated data will be added to the skeleton as a dataset is formed. Because each particular proposal orders the integrated list differently, prior to each proposal dataset generation, the skeleton is first reordered to match the order of the appropriate proposal (model) list order before any calculations begin.

The case-specific skeleton provides a common source of pre-calculated, order-independent data which serves as a starting point for each large dataset generation process. The skeleton must only be sorted to match a specific proposal ordering each time a dataset is generated.

4.2.4 calculate standalone dataset

"Standalone" refers to an unmerged, or independent employee group, and normally relates to modeling each employee group separately as if no merger had occurred.

The skeleton file contains information for all of the employee groups.

Information pertaining to each separate group may be extracted from the skeleton file quite easily and processed independently. Because list order within each native group is static, it is a fairly straightforward task to compute the standalone datasets as compared to an integrated dataset computation.

Though the job assignment process is less demanding with separate groups, there are other conditions which complicate matters. If there are any pre-existing special rights to jobs within one or more of the employee groups, they must be honored and applied.

Additionally, the number of jobs within each job level will likely fluctuate over time. This directly affects job assignment. Finally, furloughs and recalls must be handled properly according to job count changes and recall schedules.

The standalone dataset is created as a pandas dataframe and is stored in a serialized pickle file format.

4.2.5 calculate integrated order-dependent dataset

The production of an integrated dataset is more complex than the standalone datasets.

The integrated datasets are list order dependent. As mentioned above, before any work can begin, an appropriate list order must be selected and the skeleton file sorted accordingly. A properly sorted skeleton file serves as the framework for an integrated dataset.

An integrated list typically introduces multiplex requirements into the dataset calculation process.

A standard provision when integrating a workforce is that an employee will be able to keep a job held prior to a merger, even if the integrated list places that employee in a position that would not permit it. This provision is known as "no bump, no flush".

Quite often, due to differences in demographics, hiring patterns, and job opportunities, "fences", or conditions and restrictions are applied prospectively to the operation of a combined seniority list. These fences may place a cap or floor on the number of jobs which may be held by employees from one or more of the original groups, provide some sort of ratio assignment process, or apply some other corrective action to ensure an equitable outcome.

It is also common to see a time span between the "official" merger date and the actual implementation of an integrated seniority list. This delayed implementation affects the future operation of the list.

seniority_list is able to incorporate all of these factors along with pre-existing job assignment conditions, job count changes, furlough and recall schedules, and compensation schedules when calculating integrated datasets.

As with the standalone dataset, the integrated dataset(s) will be pandas dataframes, written to disk as serialized pickle files.

The integrated datasets contain one row for each employee for every month within the model. This means that the datasets may be fairly large. While the exact size depends on the demographics of the employees, an initial list with 12,000 employees will typically produce a dataset with over 1.5 million rows containing 34 columns of data. For reference, as of version 0.62, the time required to produce one dataset of that size and write it to disc is under 3 seconds with a linux desktop computer equiped with a relatively fast processor (i7) and a solid state drive. Processing time will be more or

less depending on the computer hardware and operating system utilized when running the program.

4.2.6 analyze results

Once the datasets have been produced, the user is free to explore them using many of the built-in methods of Python and the "scientific stack" libraries including pandas, NumPy, SciPy, and others. The datasets are stored as files which may be converted to other types of files for analysis within other programs. Interactive exploration and visualization of the dataset is readily available through the use of the Jupyter notebook or an Ipython session. The Jupyter notebook is the recommended interface to seniority_list for all users due to its excellent interactive features.

seniority_list includes many built-in plotting functions making it relatively simple to visually explore and contrast multiple attributes of the datasets. Most of these functions accept a variety of inputs allowing a wide range of analysis. The included **STATIC_PLOTTING.ipynb** and **INTERACTIVE_PLOTTING.ipynb** notebooks demonstrate many of these functions.

The standard built-in charts are produced by a Python library called matplotlib[39], and another called seaborn[40], which is a charting library built on top of matplotlib with a focus on statistics. Interactive charts and the editor tool are powered by the bokeh[41] library which provides users with real-time selection, filtering, and animation of the dataset results.

seniority_list also includes a *reports* module with functions that produce summarized statistical data from all calculated datasets for the current case study. The generated data is presented in tabular form via excel spreadsheets and/or visually through numerous chart images. The summary reports offer a high-level view into integrated list outcomes across a limited set of attributes for quick familiarization and comparison of proposal outcomes. The **REPORTS.ipynb** notebook included with the program provides an example of the report generation process.

[39] http://matplotlib.org/
[40] https://stanford.edu/~mwaskom/software/seaborn/
[41] https://bokeh.pydata.org/en/latest/

4.2.7 modify list order with the editor tool (optional)

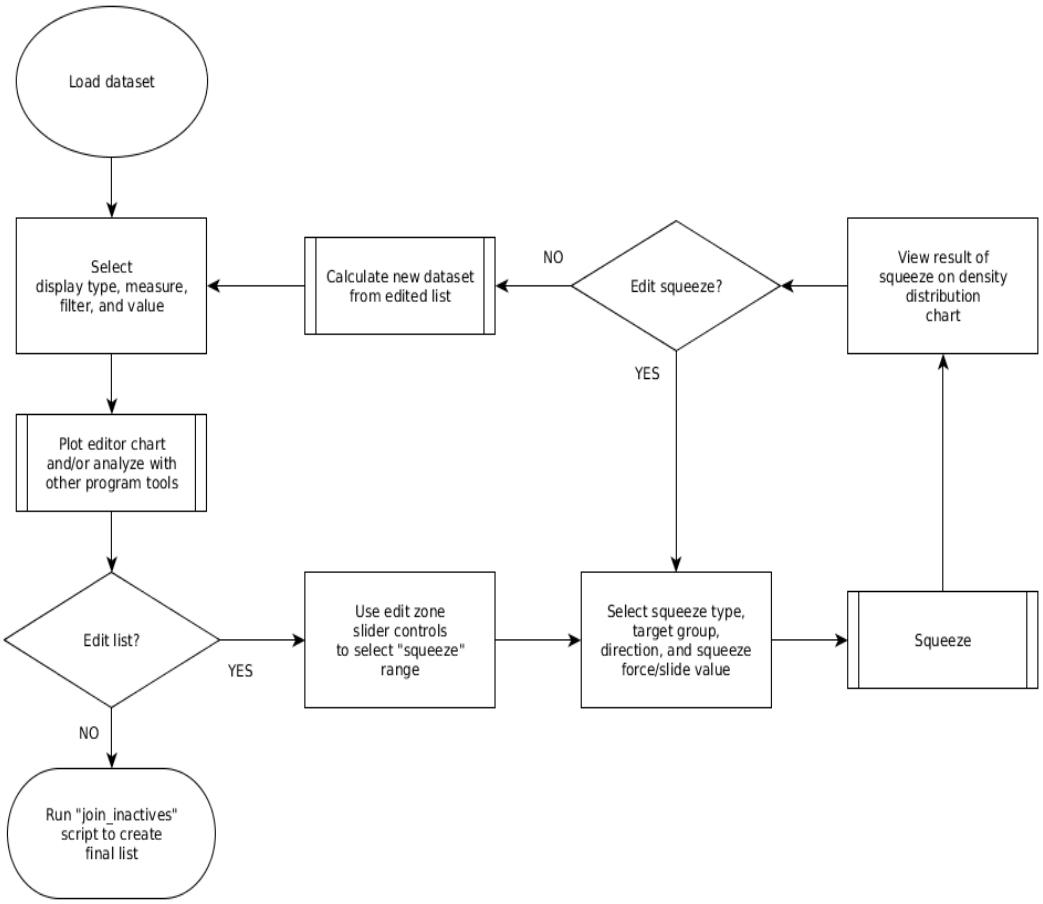


Fig. 2: list editing process (click to enlarge)

Initial dataset analysis will likely reveal certain issues of inequity related to a particular list order proposal. The editor tool was designed to allow adjustment of proposed list order through an interactive process.

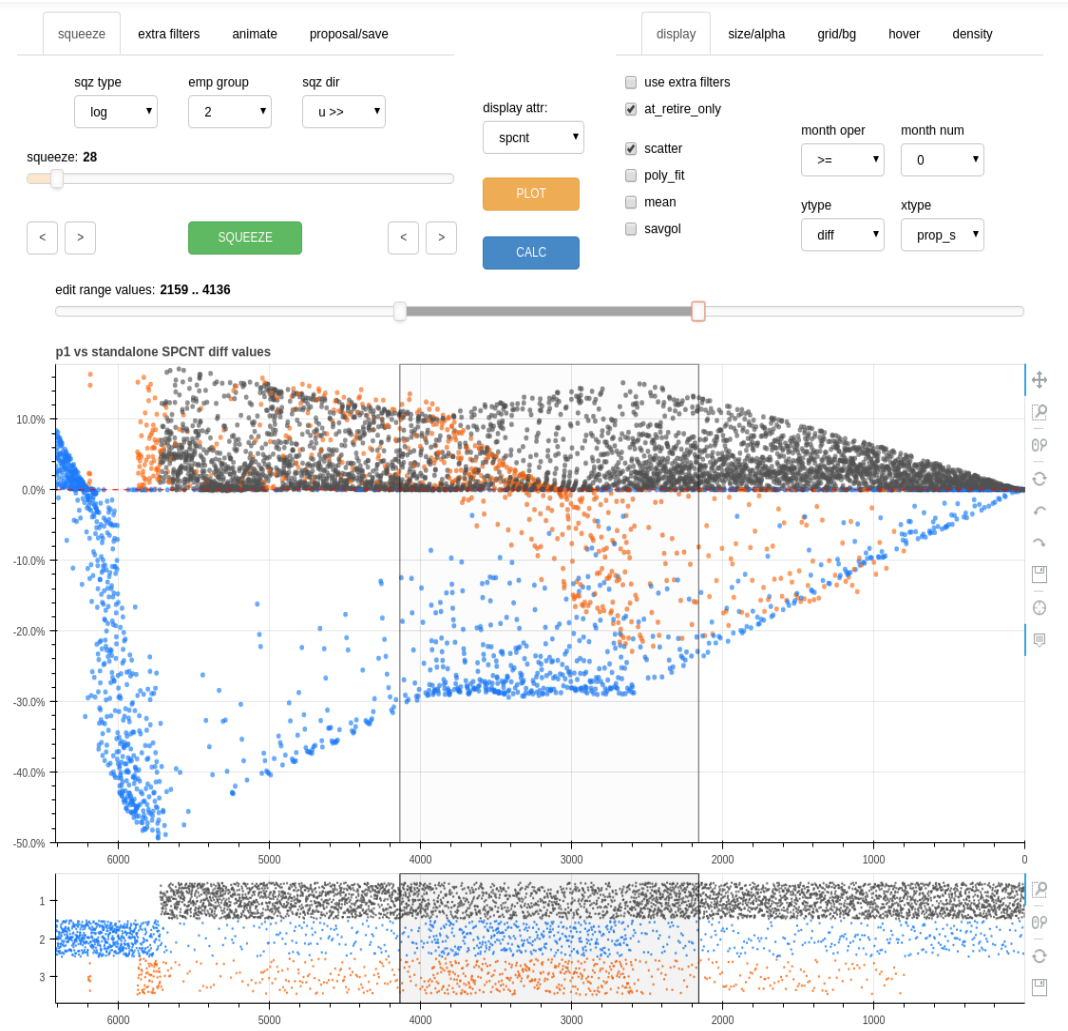


Fig. 3: the editor tool interface

An attribute differential comparison or actual values chart is used to quickly reveal equity distortions within an integrated dataset and to identify where modification of list order may be necessary to minimize excessive gains or losses for a specific employee group(s) or to more evenly distribute opportunities within the combined workforce.

A section of an integrated list may be edited by using slider controls within the editor tool to position vertical lines on either side of the section. An algorithm within the editor tool is then utilized to "slide" or "squeeze" the members from one of the original employee groups up or down the list, creating a new modified order, while maintaining proper relative ordering within each employee group. The movement of an employee group relative to another employee group(s) within an integrated seniority list not only changes the relative ranking of employee groups to one another but also effectively changes the distribution of jobs over the life of the data model, which in turn affects other outcome dataset metrics.

The relative positioning of each employee group may be precisely adjusted with the editor tool so that calculated attribute differentials (gains or losses) are minimized, or observed inequitable attribute outcomes are reduced or eliminated.

The edit process may be repeated and adjusted as necessary to selected sections of integrated list until the equity distortion(s) are reduced or eliminated.

4.2.8 create lists with list_builder (optional)

The *list_builder* module contains functions allowing custom list construction from the master file input. "Hybrid" lists may be built by ranking and sorting the master list according to a combined weighted attribute product. Any combination of attributes and weightings may be incorporated to contruct lists.

Note: "Hybrid" lists must only be considered as a starting point in nearly all cases. This is due to the simple fact that a consistent formula applied to combine lists with inconsistent attributes, such as demographics and hiring patterns, will invariably lead to inequitable outcome results. An unmodified hybrid style list solution would be acceptable only in the rare case when employee groups are nearly identical in terms of attribute distribution.

4.2.9 reinsert inactives

Inactive employees are defined as employees who are not occupying or bidding for a position which would otherwise affect the job opportunities for those employees below him/her on the seniority list. Examples of inactive employees may include those with a status of medical, military, or supervisory leave.

Because inactive employees do not bid for jobs and have no effect on the operation of a seniority list, they are removed from the list prior to the dataset calculation process. While many on inactive status will return to active status, the assumption is made that other employees will do the opposite and provide a counterbalance.

Once a final integrated list order has been determined, the inactive employees must be reinserted into the overall list.

The inactive employees are reinserted using the **join_inactives** script. The inactives may be inserted into the integrated list by locating them next to an employee from their native group who is either just senior or just junior to them. The final product of this process is converted to an Excel spreadsheet, placed within the **reports** folder.

4.3 interacting with seniority_list

seniority_list was designed to use the browser-based Jupyter notebook as its interface in all areas of functionality. The Jupyter notebook[42] has a relatively shallow learning curve while yielding vast returns.

seniority_list has been tested using FireFox and Chrome (or Chromium) browsers. Chrome offers better performance when using seniority_list and is recommended. FireFox will work with the program, but will lag somewhat when displaying complex visualizations. Other browsers have not been tested.

There are five notebook files included with seniority_list to help the user get started. Please refer to the user guide for more information.

[42] http://jupyter.org/

CHAPTER

FIVE

USER GUIDE

This user guide will begin with a general discussion of the foundational elements of the program followed by a detailed instruction manual.

Please read the "operational overview" section prior to tackling this user guide.

5.1 general

The programmatic goal of seniority_list is to create relatively large data models which can be analyzed and compared. The program orders an integrated list as directed and then uses multiple algorithms to calculate the resultant metrics.

The seniority_list program is written in a procedural style and is designed to employ the Jupyter notebook as the user interface. Therefore, to use seniority_list, first launch the Jupyter notebook from the terminal (PowerShell is recommended if running Windows):

```
jupyter notebook
```

A new browser window will open with a presentation of files and folders. Navigate to the **seniority_list** folder and then to the desired notebook file or initialize a new notebook as desired. The notebook interface provides a platform from which to run the program scripts and functions. Detailed operational instructions are located in the "notebook interface" section below.

An analysis of a particular integration will be referred to as a "case study" within the seniority_list documentation, and the particular files, data, etc. relating to that case study will be described as being "case-specific".

Case-specific Excel input files are selected by the program for processing as directed by a case_study input variable read initially as an argument to the **build_program_files.py** script. The Excel input files must be formatted and located in a user-created, case-study-named folder within the **excel** folder so that the program

can find and process them. In other words, the case_study input variable will be the same as the name of the folder containing the input Excel files and determines which input files will be used to create the foundational program files.

The input files consist of four Excel files. The actual names of the Excel input workbooks and the spreadsheets within them remain the same for all case studies. Only the name of the container folder changes and each case study has its own folder. This system allows many different case studies to exist within the program, and makes it a trivial excercise to switch between case studies, simply by changing the "case" argument to the **build_program_files.py** script when loading a new study into the program.

After the case-specific Excel files are in place, seniority_list is able to rapidly generate the datasets using a series of scripts as follows (simplified):

> First, foundational program files are generated (pandas dataframes) and are stored as serialized pickle files within the auto-generated **dill** folder with the **build_program_files.py** script.
>
> Next, a relatively long pandas dataframe is created from the freshly-created program files. This file is know as the "skeleton" file because it serves as the frame for all of the datasets which will be generated by the program. The skeleton file contains employee data which remains constant regardless of list order. The skeleton file is created by running the **make_skeleton.py** script.
>
> Finally, datasets are generated with two scripts, one for the standalone data and one for the integrated data, **standalone.py** and **compute_measures.py**. The integrated dataset creation process will be repeated for each list ordering proposal. Both the standalone and integrated datasets will incorporate specific options and scenarios set by the user.

The process to produce the datasets may be rapidly accomplished utilizing the **RUN_SCRIPTS** notebook included with the program, with modifications appropriate for a particular case study. When the program is initially downloaded, the notebook is set up properly for use with the sample case study included with the program. The **RUN_SCRIPTS** notebook serves as a template for use within actual user case studies, as do the other program notebooks.

seniority-list includes many visualization functions which have been designed specifically for seniority list analysis. These functions are located within the *matplotlib_charting* and *interactive_plotting* modules. Most of the plotting functions are demonstrated with the sample **STATIC_PLOTTING** notebook.

An array of summary statistical data formulated from all outcome datasets may be generated with functions within the *reports* module. The **REPORTS** notebook contains code cells to demonstrate the summary reports functionality of seniority_list.

The editor tool allows list order analysis, editing and feedback through an interactive

interface. The **EDITOR_TOOL** notebook is included with the program and will start the editor when it is run.

seniority_list is also able to rapidly produce "hybrid" lists with proportional weighting applied to any number of attributes through functions found within the *list_builder* module. A sample hybrid list is created when the **RUN_SCRIPTS** notebook is run.

5.1.1 program components and file structure

This section describes the files which make up the seniority_list program prior to and after running the program scripts.

The file components of seniority_list may be categorized as follows:

original files

- function modules:
  - *functions.py* - dataset generation, editor helper routines
  - *matplotlib_charting.py* - static charting functions
  - *interactive_plotting.py* - interactive charting functions
  - *converter.py* - convert basic job data to enhanced job data
  - *list_builder.py* - formulate list proposals
  - *reports.py* - generate basic summary reports
  - *editor_function.py* - editor tool
- scripts
  - *build_program_files.py* - create and format intial program dataframes from input data files
  - *make_skeleton.py* - generate framework for datasets
  - *standalone.py* - non-integrated dataset production
  - *compute_measures.py* - integrated dataset production
  - *join_inactives.py* - finalize list with inactives
- Jupyter notebooks
  - *RUN_SCRIPTS.ipynb* - generate program files and datasets
  - *STATIC_PLOTTING.ipynb* - create example data model visualizations
  - *INTERACTIVE_PLOTTING.ipynb* - example interactive charts
  - *REPORTS.ipynb* - generate high-level report, visual and tabular

  - *EDITOR_TOOL.ipynb* - run the interactive editor tool

- Excel input
  - *master.xlsx* - foundational employee data
  - *pay_tables.xlsx* - compensation data
  - *proposals.xlsx* - list order proposals
  - *settings.xlsx* - options and settings

generated files (all are dataframes except the .xlsx files and the dictionaries)

- datasets (pandas dataframes stored as serialized pickle files)
  - *ds_<proposal name>.pkl* - integrated dataset(s) generated from proposed list orders
  - *standalone.pkl* - dataset generated with non-integrated results
- reports
  - *pay_table_data.xlsx* - computed compensation data (Excel format)
  - *final.xlsx* - final list (Excel format)
  - *final.pkl* - final list (dataframe format)
  - *ret_stats.xlsx** - retirement statistics
  - *annual_stats.xlsx** - annual statistics
  - **ret_charts*** - retirement statistics chart images folder
  - **annual_charts*** - annual statistics chart images folder

    *generated with the *reports* module
- dictionaries
  - *dict_settings.pkl* - options and settings
  - *dict_colors.pkl* - colormaps for plotting
  - *dict_attr.pkl* - dataset attribute descriptions
  - *dict_job_tables.pkl* - monthly job counts
  - *editor_dict.pkl* - initial values for editor tool
- indexed pay tables
  - *pay_table_basic.pkl* - basic job levels monthly pay
  - *pay_table_enhanced.pkl* - enhanced job levels monthly pay
- proposals

  - *p_<proposal name>.pkl* - list order proposals
- misc.
  - *master.pkl* - employee data
  - *case_dill.pkl* - single-value dataframe with case study name
  - *last_month.pkl* - percentage of retirement month working
  - *proposal_names.pkl* - the integrated list order proposals

seniority_list includes a sample integration case study for simulating an integration involving three employee groups. The sample case_study is named "sample3". The **sample3** folder and its contents within the **excel** folder contain the sample case study data.

Note: The reference to the **seniority_list** folder throughout the documentation refers to the **seniority_list** folder within the parent **seniority_list** folder.

```
seniority_list/seniority_list/
```

The images below display the file structure or "tree" views of the files and folders within the **seniority_list** folder of the program. The **seniority_list** folder contains all of the code used to actually operate the program. There are other files and folders located within the **seniority_list** folder which are of an administrative nature and have been removed from the tree views for clarity (.ipynb_checkpoints and __pycache__ folders).

The next several images will highlight the new files created as the various scripts are run. The specifics concerning the purpose and product of the various scripts will be explained later.

The left image below shows the tree structure of the program as it exists when initially downloaded.

The file structure of seniority_list expands significantly when the **build_program_files** script is run. In the right image below, new files are shown within the red boxed areas, and new folders are shown within the green boxes. Most of the new files are pandas dataframes which have been converted to a "pickle" (.pkl extension) file format, a format which is optimized for fast storage and retrieval. All of the "pickle" files in the **dill** folder are cleared and replaced when a new case study is selected and calculated.

Notice a new folder, **reports**, has been created which in turn contains another new folder with the case study name ("sample3" in this case). This folder contains a new Excel file, *pay_table_data.xlsx*, pertaining to calculated compensation information.

The fact that there are three files in the **dill** folder beginning with "p_" indicates that three integrated list proposals were read from the Excel input file, *proposals.xlsx*.

initial program files

```
.
├── build_program_files.py
├── compute_measures.py
├── converter.py
├── dill
├── editor_function.py
├── EDITOR_TOOL.ipynb
├── excel
│   └── sample3
│       ├── master.xlsx
│       ├── pay_tables.xlsx
│       ├── proposals.xlsx
│       └── settings.xlsx
├── functions.py
├── INTERACTIVE_PLOTTING.ipynb
├── interactive_plotting.py
├── join_inactives.py
├── list_builder.py
├── make_skeleton.py
├── matplotlib_charting.py
├── REPORTS.ipynb
├── reports.py
├── RUN_SCRIPTS.ipynb
├── standalone.py
└── STATIC_PLOTTING.ipynb

3 directories, 21 files
```

after **build_program_files** script

```
.
├── build_program_files.py
├── compute_measures.py
├── converter.py
├── dill
│   ├── case_dill.pkl
│   ├── dict_attr.pkl
│   ├── dict_color.pkl
│   ├── dict_job_tables.pkl
│   ├── dict_settings.pkl
│   ├── editor_dict.pkl
│   ├── last_month.pkl
│   ├── master.pkl
│   ├── pay_table_basic.pkl
│   ├── pay_table_enhanced.pkl
│   ├── p_p1.pkl
│   ├── p_p2.pkl
│   ├── p_p3.pkl
│   └── proposal_names.pkl
├── editor_function.py
├── EDITOR_TOOL.ipynb
├── excel
│   └── sample3
│       ├── master.xlsx
│       ├── pay_tables.xlsx
│       ├── proposals.xlsx
│       └── settings.xlsx
├── functions.py
├── INTERACTIVE_PLOTTING.ipynb
├── interactive_plotting.py
├── join_inactives.py
├── list_builder.py
├── make_skeleton.py
├── matplotlib_charting.py
├── reports
│   └── sample3
│       └── pay_table_data.xlsx
├── REPORTS.ipynb
├── reports.py
├── RUN_SCRIPTS.ipynb
├── standalone.py
└── STATIC_PLOTTING.ipynb

5 directories, 36 files
```

The framework upon which datasets are built for a particular case study is the "skeleton" file. The skeleton file is created with the **make_skeleton.py** script. The output is stored as *skeleton.pkl* within the **dill** folder, indicated in the lower left image.

Next, a "standalone" or unmerged dataset is generated which contains information for each employee group as if a merger had not occurred. This data is all in one file, *standalone.pkl*, as indicated in the lower right image.

after **make_skeleton** script

```
.
├── build_program_files.py
├── compute_measures.py
├── converter.py
├── dill
│   ├── case_dill.pkl
│   ├── dict_attr.pkl
│   ├── dict_color.pkl
│   ├── dict_job_tables.pkl
│   ├── dict_settings.pkl
│   ├── editor_dict.pkl
│   ├── last_month.pkl
│   ├── master.pkl
│   ├── pay_table_basic.pkl
│   ├── pay_table_enhanced.pkl
│   ├── p_p1.pkl
│   ├── p_p2.pkl
│   ├── p_p3.pkl
│   ├── proposal_names.pkl
│   └── skeleton.pkl
├── editor_function.py
├── EDITOR_TOOL.ipynb
├── excel
│   └── sample3
│       ├── master.xlsx
│       ├── pay_tables.xlsx
│       ├── proposals.xlsx
│       └── settings.xlsx
├── functions.py
├── INTERACTIVE_PLOTTING.ipynb
├── interactive_plotting.py
├── join_inactives.py
├── list_builder.py
├── make_skeleton.py
├── matplotlib_charting.py
├── reports
│   └── sample3
│       └── pay_table_data.xlsx
├── REPORTS.ipynb
├── reports.py
├── RUN_SCRIPTS.ipynb
├── standalone.py
└── STATIC_PLOTTING.ipynb

5 directories, 37 files
```

after **standalone** script

```
.
├── build_program_files.py
├── compute_measures.py
├── converter.py
├── dill
│   ├── case_dill.pkl
│   ├── dict_attr.pkl
│   ├── dict_color.pkl
│   ├── dict_job_tables.pkl
│   ├── dict_settings.pkl
│   ├── editor_dict.pkl
│   ├── last_month.pkl
│   ├── master.pkl
│   ├── pay_table_basic.pkl
│   ├── pay_table_enhanced.pkl
│   ├── p_p1.pkl
│   ├── p_p2.pkl
│   ├── p_p3.pkl
│   ├── proposal_names.pkl
│   ├── skeleton.pkl
│   └── standalone.pkl
├── editor_function.py
├── EDITOR_TOOL.ipynb
├── excel
│   └── sample3
│       ├── master.xlsx
│       ├── pay_tables.xlsx
│       ├── proposals.xlsx
│       └── settings.xlsx
├── functions.py
├── INTERACTIVE_PLOTTING.ipynb
├── interactive_plotting.py
├── join_inactives.py
├── list_builder.py
├── make_skeleton.py
├── matplotlib_charting.py
├── reports
│   └── sample3
│       └── pay_table_data.xlsx
├── REPORTS.ipynb
├── reports.py
├── RUN_SCRIPTS.ipynb
├── standalone.py
└── STATIC_PLOTTING.ipynb

5 directories, 38 files
```

The integrated list datasets are produced with the **compute_measures.py** script, run separately for each integrated list proposal. The dataset file names begin with "ds_" (lower left image).

The *final.xlsx* and *final.pkl* files shown in the lower right image were generated by the **join_inactives.py** script. Normally these files would be created at the end of the entire analysis process when a final integrated list has been produced and the inactive employees are reinserted into the new combined list. These files contain the same information, only the file format is different - one is a pandas dataframe and the other is an Excel file generated for user convenience.

after **compute_measures** script

```
.
├── build_program_files.py
├── compute_measures.py
├── converter.py
├── dill
│   ├── case_dill.pkl
│   ├── dict_attr.pkl
│   ├── dict_color.pkl
│   ├── dict_job_tables.pkl
│   ├── dict_settings.pkl
│   ├── ds_p1.pkl
│   ├── ds_p2.pkl
│   ├── ds_p3.pkl
│   ├── editor_dict.pkl
│   ├── last_month.pkl
│   ├── master.pkl
│   ├── pay_table_basic.pkl
│   ├── pay_table_enhanced.pkl
│   ├── p_p1.pkl
│   ├── p_p2.pkl
│   ├── p_p3.pkl
│   ├── proposal_names.pkl
│   ├── skeleton.pkl
│   └── standalone.pkl
├── editor_function.py
├── EDITOR_TOOL.ipynb
├── excel
│   └── sample3
│       ├── master.xlsx
│       ├── pay_tables.xlsx
│       ├── proposals.xlsx
│       └── settings.xlsx
├── functions.py
├── INTERACTIVE_PLOTTING.ipynb
├── interactive_plotting.py
├── join_inactives.py
├── list_builder.py
├── make_skeleton.py
├── matplotlib_charting.py
├── reports
│   └── sample3
│       └── pay_table_data.xlsx
├── REPORTS.ipynb
├── reports.py
├── RUN_SCRIPTS.ipynb
├── standalone.py
└── STATIC_PLOTTING.ipynb

5 directories, 41 files
```

after **join_inactives** script

```
.
├── build_program_files.py
├── compute_measures.py
├── converter.py
├── dill
│   ├── case_dill.pkl
│   ├── dict_attr.pkl
│   ├── dict_color.pkl
│   ├── dict_job_tables.pkl
│   ├── dict_settings.pkl
│   ├── ds_p1.pkl
│   ├── ds_p2.pkl
│   ├── ds_p3.pkl
│   ├── editor_dict.pkl
│   ├── final.pkl
│   ├── last_month.pkl
│   ├── master.pkl
│   ├── pay_table_basic.pkl
│   ├── pay_table_enhanced.pkl
│   ├── p_p1.pkl
│   ├── p_p2.pkl
│   ├── p_p3.pkl
│   ├── proposal_names.pkl
│   ├── skeleton.pkl
│   └── standalone.pkl
├── editor_function.py
├── EDITOR_TOOL.ipynb
├── excel
│   └── sample3
│       ├── master.xlsx
│       ├── pay_tables.xlsx
│       ├── proposals.xlsx
│       └── settings.xlsx
├── functions.py
├── INTERACTIVE_PLOTTING.ipynb
├── interactive_plotting.py
├── join_inactives.py
├── list_builder.py
├── make_skeleton.py
├── matplotlib_charting.py
├── reports
│   └── sample3
│       ├── final.xlsx
│       └── pay_table_data.xlsx
├── REPORTS.ipynb
├── reports.py
├── RUN_SCRIPTS.ipynb
├── standalone.py
└── STATIC_PLOTTING.ipynb

5 directories, 43 files
```

The program includes four sample Jupyter notebook files which are indicated with the red boxes below and left.

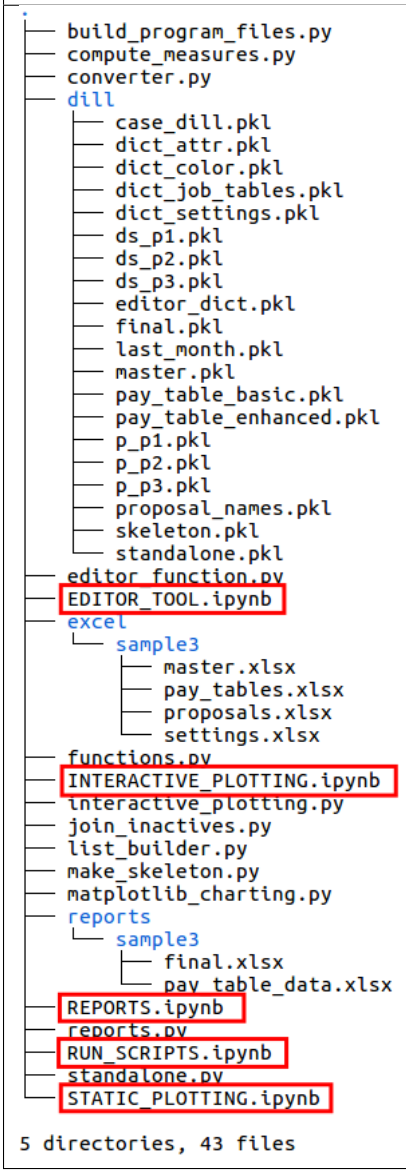
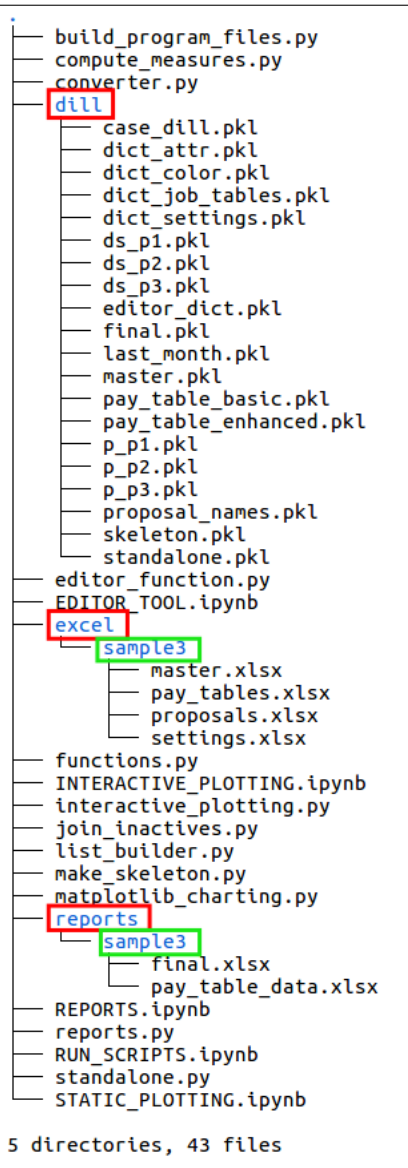
The folders within the **seniority_list** folder are indicated in the lower right image.

The **dill** folder contains the generated program files which are used by the various scripts to make the model datasets. The datasets are also stored within the **dill** folder once they are produced. The names and the quantity of files within the **dill** folder will be the same for all case studies, with the exception of case-specific proposal files (starting with "p_") and case-specific dataset files (starting with "ds_". The actual contents of the program files will be different for each case.

The **excel** folder contains a folder for each existing case study. In this view, there is

only one folder, (**sample3**), containing the four Excel input files. If other case studies existed, there would be additional folders within the **excel** folder, each containing four Excel input files with the same names, but with the contents of the Excel file worksheets modified as appropriate for each case.

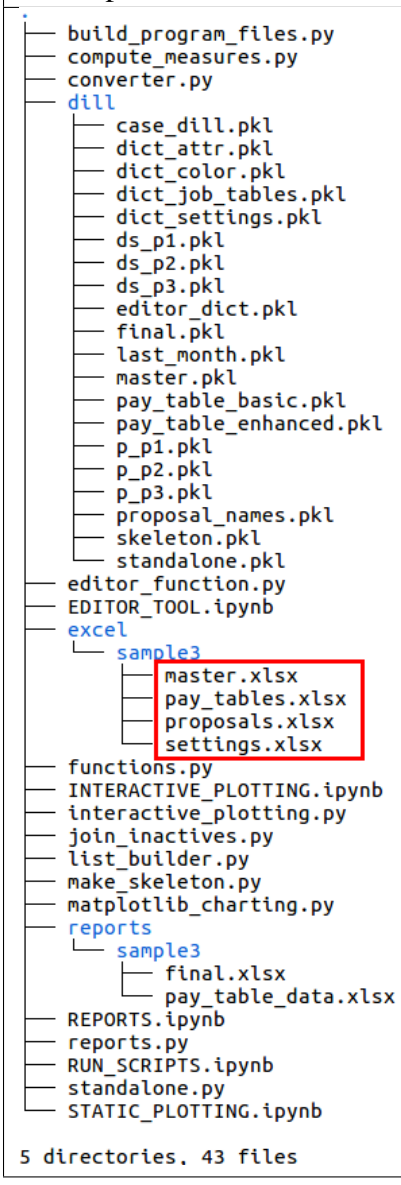
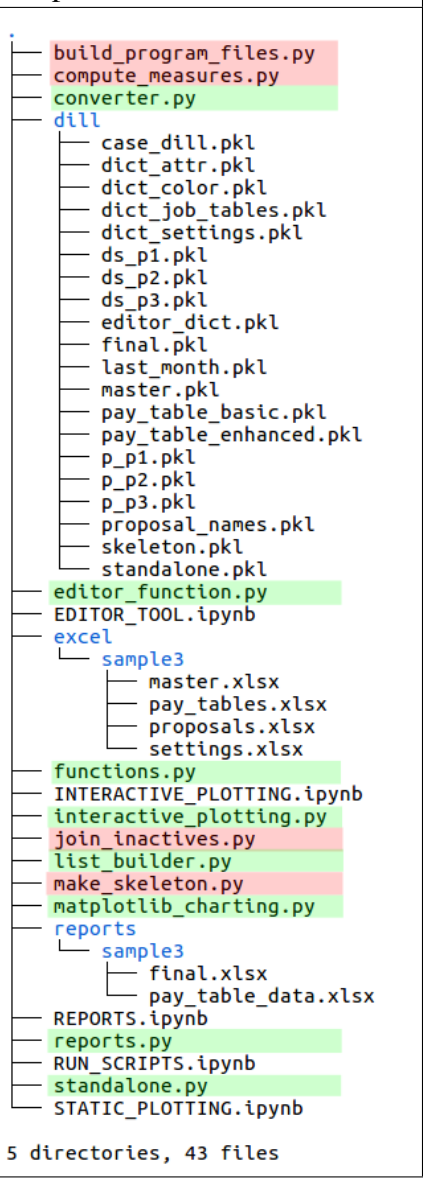
The **reports** folder will contain an auto-generated folder for each case study. The Excel files located within these folders are created by the program.

the 5 sample notebook files	program folders
	

The user input files are marked in the image below and left. The four case-specific Excel files are located within a folder named after the case study (sample3) within the **excel** folder.

The tree view below right highlights the program scripts in red and the function mod-

ules in green. The program scripts perform most of the work of seniority_list while the function module components are used within the scripts and the Jupyter notebooks to perform specific actions.

user input files	scripts and function modules
	

5.2 program flow

5.2.1 input data

seniority_list reads user-defined input data from four Excel workbooks. These input files must be formatted and located properly for the program to run.

Note: The following discussion provides information concerning how the input files fit in with program flow. Please see the "excel input files" page of this documentation for complete descriptions and formatting requirements of the Excel input files.

The four Excel input files:

- master.xlsx
  - basic employee data file
  - contains data for all employee groups within one worksheet
- proposals.xlsx
  - order and empkey (unique number derived from employee group number and employee number)
  - contains one worksheet for each proposed integrated list order
- pay_tables.xlsx
  - pay table for basic job levels
  - basic and enhanced monthly pay hours, descriptive job codes, full-time vs. part-time job level percentages
- settings.xlsx
  - scalar options (single value variables)
  - tabular data sources to be converted to various lists and dictionaries

setup workflow summary

The basic idea is to use existing Excel input files workbooks as an easy starting point or template for new case study inputs.

1. **Navigate to the excel folder within the seniority_list folder.**
2. **Copy the sample3 (or any other case study folder) and paste it right back into the same folder.**

3. **Rename the new folder as the new case study name.**
4. **Modify the content of the workbooks within the new case study folder to match the new case study parameters.**

Input file basics

The program requires input from four prepared Excel workbooks containing employee data, pay scales, job counts, proposed integrated list orderings, and other program data and options information.

Examples of input information:

- job counts per job category per employee group
- changes in job counts over time
- colors to be used when plotting data
- use a constant retirement age or calculate an increase at some point
- an option to use basic or enhanced job levels
- whether or not to assume a delayed implementation of the integrated list

Input file naming and location

Data for many merger studies may be stored within seniority_list at the same time. A naming convention applied to the folders containing the Excel input files ensures that the program uses the correct data for the selected integration study.

The user will choose a case study name when preparing to analyze an employee group merger with seniority_list. For purposes of discussion, we will assume there are two companies involved in a hypothetical merger, "Southern, Inc." and "Acme Co.", and the case name chosen is "southern_acme". This name will become the name of the folder which will contain the four case-specific Excel input data files.

The recommended way to create the input files for a new case study is to navigate to the **excel** folder, copy an existing case study input folder (the **sample3** folder if no other case studies exist), then paste it back into the **excel** folder and rename it with the desired case study name ("southern_acme" in this case.) The user will then modify the contents of the workbooks within the case study folder to match the actual parameters of the new case study as described within the "excel input files" section of the documentation.

The names of the four files located within a case study folder are the same for all case studies: "master.xlsx", "pay_tables.xlsx", "proposals.xlsx", and

"settings.xlsx". These file names should not be modified because the program will look for them specifically regardless of the case study name.

By far, the most of the effort involved when utilizing seniority_list will be directed toward preparing Excel input data for consumption by the program. However, once everything is set up, minimal effort is required to analyze multiple integration scenarios.

Selecting a case study

With the input files in place and loaded with proper information, the user selects an integration study for analysis by manually setting the "case" argument for the **build_program_files.py** script. The "southern_acme" case study has been selected in the example below (Jupyter notebook cell command):

```
%run build_program_files southern_acme
```

This one argument will set up the program to select the proper source files for all of the calculations used to produce multiple data models corresponding to designated integration proposals. The user may easily switch between completely different case studies simply by changing the single argument to the **build_program_files.py** script and then rerunning the program. If the user desired to run the sample case study after analyzing the "southern_acme" case, he/she would rerun the script as follows:

```
%run build_program_files sample3
```

After running the **build_program_files** script, the other scripts involved in building the datasets must be run as well, as described in the sections below. The included **RUN_SCRIPTS** notebook offers a template to make this process easy for any case study, with simple modification. This will be explained within the "operation" section below.

The input Excel files and the files generated by the **build_program_files** script relating to a specific case study provide the foundational information for the main dataset generation process.

5.2.2 build program files

Processing script: **build_program_files.py**

This script creates the necessary support files from the input Excel files required for program operation. The input files are read from the appropriate case study folder within the **excel** folder.

The **build_program_files.py** script requires one argument which designates the case

study to be analyzed. That argument directs the script to look for the input files within a folder with the same name as the argument, in the **excel** folder.

For example, to run the script from the Jupyter notebook using the sample case study, type the following into a notebook cell and run it:

```
%run build_program_files sample3
```

The files and folder created with **build_program_files.py** are as follows:

from the input Excel files:

- from proposals.xlsx:
  - proposal_names.pkl
  - p_<proposal name>.pkl for each proposal
- from master.xlsx:
  - master.pkl
  - last_month.pkl
- from pay_tables.xlsx:
  - pay_table_basic.pkl
  - pay_table_enhanced.pkl
  - pay_table_data.pkl
- from settings.xlsx
  - dict_settings.pkl
  - dict_attr.pkl

created with this script without reference to the input files:

- from code within script:
  - case_dill.pkl
  - editor_dict.pkl
  - dict_color.pkl
  - case-study-named folder in the **reports** folder (if it doesn't already exist)

descriptions of the created files:

All images may be clicked to enlarge.

The *case_dill.pkl* file is a tiny dataframe (only one value) containing the name of the current case study, as set by the "case" argument of the **build_program_files.py** script. It is referenced by the **join_inactives.py** script when writing the *final.xlsx* file within the appropriate case study folder, in the **reports** folder.

	case
value	sample3

Fig. 1: case_dill mini-dataframe

The *proposal_names.pkl* file is a very small dataframe which contains the names of the various list order proposals, obtained from the worksheet names within the *proposals.xlsx* input file. This file is referenced by many other functions when referencing list order proposals.

	proposals
0	p1
1	p2
2	p3

Fig. 2: proposal_names mini-dataframe

The *editor_dict.pkl* file is used to set the initial values in the editor tool interactive widgets (sliders, dropdown boxes, etc.) and is modified by the editor tool when in use.

```
{'base_ds_name': '',
 'box_fill_alpha': '.05',
 'box_fill_color': 'black',
 'box_line_alpha': '.8',
 'box_line_color': 'black',
 'box_line_width': '1.0',
 'case': 'sample3',
 'chk_color_apply': [0],
 'chk_display': [0],
 'chk_filter': [1],
 'chk_hover_on': [],
 'chk_hover_sel': [],
 'chk_mean': False,
 'chk_minor_grid': [],
 'chk_poly_fit': False,
 'chk_sagov': False,
 'chk_scatter': True,
 'chk_trails': [],
 'cht_title': 'spcnt',
 'cht_xflipped': False,
 'cht_xformat': '0',
 'cht_xsize': 1200,
 'cht_yflipped': False,
 'cht_yformat': '0.0%',
 'cht_ysize': 580,
 'edit_max': 6415,
 'ez_end': 6415,
 'ez_step': 5,
 'minor_grid_alpha': 0.0,
 'num_of_months': 453,
 'p2_marker_alpha': 0.8,
 'p2_marker_size': 2.2,
 'sel_base': 'standalone',
 'sel_bgc': 'White',
 'sel_bgc_alpha': '.10',
 'sel_cond': 'none',
 'sel_emp_grp': '1',
 'sel_filt1': '',
 'sel_filt2': '',
 'sel_filt3': '',
 'sel_gridc': 'Gray',
 'sel_gridc_alpha': '.20',
 'sel_measure': 'spcnt',
 'sel_mth_num': '0',
 'sel_mth_oper': '>=',
 'sel_oper1': '==',
 'sel_oper2': '==',
 'sel_oper3': '==',
 'sel_proposal': 'edit',
 'sel_sqz_dir': '<<  d',
 'sel_sqz_type': 'log',
 'sel_xtype': 'prop_s',
 'sel_ytype': 'diff',
 'slider_squeeze': 100,
 'total_count': 6415,
 'txt_input1': '',
 'txt_input2': '',
 'txt_input3': '',
 'x_high': 4169,
 'x_low': 2245}
```

Fig. 3: editor_dict dictionary for editor tool settings (sample values)

The *master.pkl* file is a pandas dataframe version of the *master.xlsx* input workbook employee list data. The dataframe structure is the same as the worksheet structure with the addition of a calculated "retdate" (retirement date) column.

	empkey	eg	lname	dob	doh	ldate	sg	fur	line	eg_order	retdate
empkey											
10011102	10011102	1	tooeyoo	1949-07-13	1973-02-26	1975-01-29	0	0	1	1	2014-07-13
10010475	10010475	1	rubelot	1949-02-05	1975-05-27	1975-05-27	0	0	1	2	2014-02-05
10013096	10013096	1	yeloxid	1949-01-08	1977-01-18	1977-01-18	0	0	1	3	2014-01-08
10012178	10012178	1	xayeaue	1951-06-07	1977-11-15	1977-11-15	0	0	1	4	2016-06-07
10014447	10014447	1	finuceu	1951-10-17	1977-12-09	1977-12-09	0	0	1	5	2016-10-17
...	...	..	...	...	...	...	..	..	..	..	...
30010130	30010130	3	eueaq	1974-09-24	2008-03-18	2010-05-02	0	1	0	800	2039-09-24
30010638	30010638	3	ueiap	1976-07-11	2008-03-22	2010-05-06	0	1	0	801	2041-07-11
30010292	30010292	3	zaber	1974-11-13	2008-04-19	2013-07-24	0	1	0	802	2039-11-13
30010633	30010633	3	iopip	1976-12-13	2008-04-15	2010-06-26	0	1	0	803	2041-12-13
30010470	30010470	3	nufaros	1983-03-01	2008-04-19	2010-07-28	0	1	0	804	2048-03-01

[7510 rows x 11 columns]

Fig. 4: master file excerpt

The *dict_attr.pkl* file is a dictionary containing dataset column names as keys and descriptions of those names as values, as delineated on the "attribute_dict" worksheet within the *settings.xlsx* workbook. The descriptions are used for chart labeling.

```
{'age': 'age',
 'cat_order': 'global job ranking',
 'cpay': 'cumulative career pay',
 'date': 'date',
 'doh': 'date of hire',
 'eg': 'employee group',
 'empkey': 'employee number',
 'fbff': 'full bump full flush',
 'fur': 'furlough',
 'idx': 'index',
 'jnum': 'job level',
 'job_count': 'job level count',
 'jobp': 'percentage within job level',
 'ldate': 'longevity date',
 'line': 'active',
 'lname': 'last name',
 'lnum': 'list number (includes furloughed employees)',
 'lspcnt': 'list percentage (includes furloughed employees)',
 'mlong': 'longevity (months)',
 'mnum': 'month number',
 'mpay': 'monthly pay',
 'mth_pcnt': 'month pay percentage',
 'new_order': 'editor order',
 'orig_job': 'original job',
 'pay_raise': 'pay rate multiplier',
 'rank_in_job': 'rank in job level',
 'ret_mark': 'retirement month',
 'retdate': 'retirement date',
 's_lmonths': 'starting longevity - months',
 'scale': 'longevity pay scale',
 'sg': 'special group',
 'snum': 'seniority number',
 'spcnt': 'seniority list percentage',
 'year': 'contract year',
 'ylong': 'longevity (years)'}
```

Fig. 5: attribute dictionary

The *dict_color.pkl* file is a relatively large dictionary containing matplotlib colormap names to color lists key-value pairs. The color lists are in [red, green, blue, alpha] format. The color dictionary is discussed in the "visualization" section below.

```
OrderedDict([('Accent',
              [(0.49803921568627452,
                0.78823529411764703,
                0.49803921568627452,
                1.0),
               (0.49803921568627452,
                0.78823529411764703,
                0.49803921568627452,
                1.0),
               (0.74509803921568629,
                0.68235294117647061,
                0.83137254901960789,
                1.0),
               (0.74509803921568629,
                0.68235294117647061,
                0.83137254901960789,
                1.0),
               (0.99215686274509807,
                0.75294117647058822,
                0.52549019607843139,
                1.0),
               (0.99215686274509807,
                0.75294117647058822,
                0.52549019607843139,
                1.0),
```

Fig. 6: color dictionary excerpt, rgba format

The *dict_settings.pkl* file is a dictionary containing program options and data necessary for seniority_list to operate. Nearly all of the data from the *settings.xlsx* input file ends up in this dictionary, either in native format or as a modified format as a calculated derivative or reshaped as elements within a Python data structure (or both).

```
                          93,
                          94},
'delayed_implementation': 1,
'discount_longev_for_fur': 1,
'dist_count': 'split',
'dist_ratio': 'split',
'dist_sg': 'part',
'eg_counts': [[197, 470, 1056, 412, 628, 1121, 0, 0],
              [80, 85, 443, 163, 96, 464, 54, 66],
              [0, 26, 319, 0, 37, 304, 0, 0]],
'enhanced_jobs': 1,
'enhanced_jobs_full_suffix': ' B',
'enhanced_jobs_part_suffix': ' R',
'future_raise': 0,
'imp_month': 34,
'implementation_date': datetime.datetime(2016, 10, 31, 0, 0),
'init_ret_age': 65.0,
'init_ret_age_months': 0,
'init_ret_age_years': 65,
'integrated_counts_preimp': 0,
'j_changes': [[1, [35, 64], 43, [40, 3, 0]],
              [4, [35, 64], 72, [66, 6, 0]],
              [2, [1, 52], -408, [-377, -23, -8]],
              [5, [1, 52], -510, [-474, -26, -10]],
              [3, [1, 61], 411, [376, 26, 9]],
              [6, [1, 61], 411, [376, 26, 9]]],
'jc_months': {1,
              2,
              3,
              4,
```

Fig. 7: settings dictionary excerpt

The *dict_job_tables.pkl* file is a dictionary containing data related to monthly job counts. The dictionary values are numpy arrays pertaining to both standalone and integrated employee groups, incorporating changes in the number of jobs over time as described with the *job_changes* worksheet within the *settings.xlsx* input file. These arrays are referenced during the job assignment and analysis routines.

```
# table
  (array([[ 166,  111,  363, 1181,  218,  637,  345,  230,  475, 1229,   35,  286,  660,   19,   43,   23],
       [ 166,  111,  358, 1185,  215,  639,  345,  230,  469, 1233,   35,  283,  662,   19,   43,   23],
       [ 166,  111,  353, 1189,  212,  641,  345,  230,  463, 1237,   35,  279,  664,   19,   43,   23],
       [ 166,  111,  348, 1194,  209,  644,  345,  230,  457, 1242,   35,  275,  667,   19,   43,   23],
       [ 166,  111,  343, 1198,  206,  646,  345,  230,  450, 1246,   35,  272,  669,   19,   43,   23],
       [ 166,  111,  338, 1203,  203,  649,  345,  230,  444, 1251,   35,  268,  672,   19,   43,   23],
       [ 166,  111,  333, 1207,  200,  651,  345,  230,  438, 1255,   35,  264,  674,   19,   43,   23],
       [ 166,  111,  328, 1212,  197,  653,  345,  230,  432, 1260,   35,  260,  676,   19,   43,   23],
       [ 166,  111,  323, 1216,  194,  656,  345,  230,  425, 1264,   35,  257,  679,   19,   43,   23],
       [ 166,  111,  318, 1221,  191,  658,  345,  230,  419, 1269,   35,  253,  681,   19,   43,   23],
       ...,
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23],
       [ 192,  128,  108, 1448,   65,  781,  388,  259,  156, 1496,   35,   95,  804,   19,   43,   23]]), a
```

Fig. 8: one of multiple arrays within the job table dictionary

A dataframe is created from each proposed integrated list order as indicated on the worksheets within the *proposals.xlsx* workbook input file. (*p_p1.pkl, p_p2.pkl, p_p3.pkl* with the sample case)

```
                idx
empkey
10011102  1
10010475  2
10013096  3
10012178  4
10014447  5
10014384  6
10012843  7
10014067  8
10010929  9
10014351  10
20010692  11
10011974  12
...       ..
20011068  7499
20010715  7500
20010378  7501
20011663  7502
20010104  7503
20011567  7504
20010176  7505
20010265  7506
20010549  7507
20010666  7508
20011230  7509
20010022  7510

[7510 rows x 1 columns]
```

Fig. 9: proposal file excerpt

The *pay_table_basic.pkl* and *pay_table_enhanced.pkl* files are calculated indexed compensation dataframes derived from the *pay_tables.xlsx* Excel input file. These files provide rapid data access during the dataset creation routine.

> "Indexed" means that the index of the dataframe(s) contains a unique value representing the year, longevity step, and job level. The only column ("monthly") contains the corresponding monthly compensation value.

The "ptindex" (pay table index) contains year, longevity, and job level information. The last two whole digits represent the job level. In this example case, there are 8 basic levels and 16 enhanced levels.

indexed basic pay table

ptindex	monthly
201300101.0	2050.92
201300102.0	2050.92
201300103.0	2050.92
201300104.0	2050.92
201300105.0	2050.92
201300106.0	2050.92
201300107.0	2050.92
201300108.0	2050.92
201300109.0	0.00
201300201.0	9739.44
201300202.0	8279.01
201300203.0	7686.09
...	...
201901107.0	8252.28
201901108.0	5658.66
201901109.0	0.00
201901201.0	15773.13
201901202.0	13429.80
201901203.0	12646.53
201901204.0	10802.97
201901205.0	9201.60
201901206.0	8667.81
201901207.0	8313.03
201901208.0	5706.45
201901209.0	0.00

[864 rows x 1 columns]

indexed enhanced pay table

ptindex	monthly
201300101.0	2152.20
201300102.0	1873.68
201300103.0	2152.20
201300104.0	2152.20
201300105.0	1873.68
201300106.0	1873.68
201300107.0	2152.20
201300108.0	1873.68
201300109.0	2152.20
201300110.0	2152.20
201300111.0	2152.20
201300112.0	1873.68
...	...
201901206.0	11553.62
201901207.0	11336.45
201901208.0	9869.38
201901209.0	9656.00
201901210.0	9095.85
201901211.0	8723.55
201901212.0	8406.40
201901213.0	7918.74
201901214.0	7594.62
201901215.0	5988.25
201901216.0	5213.30
201901217.0	0.00

[1632 rows x 1 columns]

A decimal representing the portion of an employee's final work month may be calculated using retirement date and the number of days in the retirement month. This decimal is calculated for all employee retirement dates and stored in *last_month.pkl* (the "last_pay" column below) to be used when calculating dataset career earnings attribute.

retdate	last_pay
2013-12-13	0.419355
2013-12-14	0.451613
2013-12-27	0.870968
2014-01-02	0.064516
2014-01-05	0.161290
2014-01-08	0.258065
2014-01-14	0.451613
2014-01-16	0.516129
2014-01-17	0.548387
2014-01-18	0.580645
2014-01-25	0.806452
2014-01-26	0.838710
...	...
2049-11-17	0.566667
2049-11-21	0.700000
2049-12-24	0.774194
2050-06-01	0.033333
2050-06-14	0.466667
2050-07-01	0.032258
2050-07-21	0.677419
2050-07-26	0.838710
2050-08-16	0.516129
2050-09-24	0.800000
2051-08-09	0.290323
2051-08-17	0.548387

[4940 rows x 1 columns]

Fig. 10: last_month file excert

The **join_inactives.py** script reinserts inactive employees into a combined seniority list order and creates two files containing the final integrated seniority list. Both files are the same - one is a pandas dataframe (*final.pkl*) and the other is written to disk in the **reports** folder as an Excel workbook (*final.xlsx*). See the "building_lists" section below for more information concerning the **join_inactives.py** script.

empkey	empkey	eg	lname	dob	doh	ldate	sg	fur	line	eg_order	retdate	snum
10011102	10011102	1	tooeyoo	1949-07-13	1973-02-26	1975-01-29	0	0	1	1	2014-07-13	1
10010475	10010475	1	rubelot	1949-02-05	1975-05-27	1975-05-27	0	0	1	2	2014-02-05	2
10013096	10013096	1	yeloxid	1949-01-08	1977-01-18	1977-01-18	0	0	1	3	2014-01-08	3
20010692	20010692	2	eafeuir	1951-03-04	1974-04-16	1975-04-25	0	0	1	1	2016-03-04	4
20011034	20011034	2	cibofen	1949-07-15	1977-03-12	1977-03-12	0	0	1	2	2014-07-15	5
...	...	..	...	...	...	...	..	..	..	..	...	..
10013323	10013323	1	aoouuae	1965-05-09	2013-12-05	2013-12-05	0	0	1	4919	2030-05-09	7506
10011198	10011198	1	taioe	1966-01-21	2013-12-07	2013-12-07	0	0	1	4920	2031-01-21	7507
10011913	10011913	1	guioj	1966-09-14	2013-12-04	2013-12-04	0	0	1	4921	2031-09-14	7508
10014005	10014005	1	uuuotau	1968-09-19	2013-12-06	2013-12-06	0	0	1	4922	2033-09-19	7509
10010515	10010515	1	oouai	1970-06-14	2013-12-08	2013-12-08	0	0	1	4923	2035-06-14	7510

[7510 rows x 12 columns]

Fig. 11: final file excerpt, dataframe version

pay_table_data.xlsx (program-generated workbook)

seniority_list calculates total monthly compensation tables which are the source for the *pay_table_enhanced.pkl* file and *pay_table_basic.pkl* files (above) used when generating compensation attributes within datasets. The monthly compensation data may be reviewed on one of the worksheets from the auto-generated *pay_table_data.xlsx* workbook within the **reports** folder. (Note that a furlough pay level has been added by the program for each year.)

	year	jnum	1	2	3	4	5	6	7	8	9	10	11	12	basic_hours	full_hours	part_hours	jobstr	order
0	2013	1	2050.92	9739.44	9818.01	9895.77	9974.34	10052.1	10130.67	10208.43	10287	10365.57	10443.33	10521.9	81	85	74	CA_4	1
1	2013	2	2050.92	8279.01	8344.62	8411.04	8477.46	8543.88	8610.3	8676.72	8743.14	8809.56	8875.98	8942.4	81	85	74	CA_3	2
2	2013	3	2050.92	7686.09	7747.65	7809.21	7870.77	7932.33	7993.89	8055.45	8117.01	8178.57	8240.13	8301.69	81	85	74	CA_2	3
3	2013	4	2050.92	4915.89	5927.58	6072.57	6219.18	6367.41	6517.26	6667.92	6872.04	7026.75	7131.24	7184.7	81	85	74	FO_4	4
4	2013	5	2050.92	4186.08	5043.87	5166.99	5291.73	5417.28	5544.45	5672.43	5845.77	5976.99	6065.28	6110.64	81	85	74	FO_3	5
5	2013	6	2050.92	3889.62	4685.85	4800.06	4915.08	5031.72	5149.98	5269.05	5429.43	5550.93	5633.55	5674.86	81	85	74	FO_2	6
6	2013	7	2050.92	5256.09	5298.21	5340.33	5381.64	5423.76	5465.88	5507.19	5549.31	5591.43	5632.74	5674.86	81	85	74	CA_1	7
7	2013	8	2050.92	2674.62	3216.51	3293.46	3372.03	3451.41	3531.6	3612.6	3721.14	3804.57	3860.46	3888.81	81	85	74	FO_1	8
8	2013	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FUR	9
9	2014	1	2050.92	10511.37	10595.61	10679.85	10764.9	10849.14	10933.38	11018.43	11102.67	11186.91	11271.15	11356.2	81	85	74	CA_4	1
10	2014	2	2050.92	8933.49	9004.77	9076.86	9148.14	9220.23	9291.51	9363.6	9434.88	9506.97	9578.25	9650.34	81	85	74	CA_3	2
11	2014	3	2050.92	8293.59	8360.01	8426.43	8492.85	8560.08	8626.5	8692.92	8759.34	8825.76	8892.18	8958.6	81	85	74	CA_2	3
12	2014	4	2050.92	5302.26	6394.95	6551.28	6709.23	6869.61	7030.8	7194.42	7413.93	7581.6	7694.19	7751.7	81	85	74	FO_4	4
13	2014	5	2050.92	4513.32	5439.96	5572.8	5707.26	5843.34	5980.23	6118.74	6305.85	6447.6	6543.18	6591.78	81	85	74	FO_3	5
14	2014	6	2050.92	4193.37	5053.59	5176.71	5301.45	5427	5554.17	5682.96	5856.3	5987.52	6076.62	6121.98	81	85	74	FO_2	6
15	2014	7	2050.92	5669.19	5714.55	5759.91	5805.27	5850.63	5895.18	5940.54	5985.9	6031.26	6076.62	6121.17	81	85	74	CA_1	7
16	2014	8	2050.92	2881.17	3465.99	3549.42	3634.47	3720.33	3806.19	3893.67	4011.93	4101.03	4161.78	4192.56	81	85	74	FO_1	8
17	2014	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FUR	9
18	2014.1	1	12503.16	12604.41	12706.47	12807.72	12908.97	13010.22	13112.28	13213.53	13314.78	13416.03	13517.28	13618.53	81	85	74	CA_4	1
19	2014.1	2	10552.68	10641.78	10727.64	10812.69	10904.22	10988.46	11069.46	11160.18	11238.75	11358.63	11479.32	11597.58	81	85	74	CA_3	2
20	2014.1	3	10000.26	10081.26	10162.26	10244.88	10328.31	10410.12	10491.93	10574.55	10655.55	10743.84	10833.75	10922.04	81	85	74	CA_2	3
21	2014.1	4	3659.58	6786.18	7926.66	8116.2	8308.17	8515.53	8750.43	8948.88	9044.46	9165.96	9248.58	9331.2	81	85	74	FO_4	4
22	2014.1	5	3659.58	5736.42	6697.08	6857.46	7022.7	7196.85	7392.06	7562.97	7639.92	7764.66	7858.62	7950.15	81	85	74	FO_3	5
23	2014.1	6	3659.58	5436.72	6346.35	6498.63	6654.15	6819.39	7008.12	7168.5	7244.64	7345.89	7417.98	7489.26	81	85	74	FO_2	6
24	2014.1	7	6601.5	6650.91	6705.18	6758.64	6808.86	6863.94	6915.78	6970.05	7022.7	7076.97	7131.24	7183.08	81	85	74	CA_1	7
25	2014.1	8	3659.58	3659.58	4199.04	4298.67	4398.3	4507.65	4630.77	4735.26	4784.67	4849.47	4894.02	4935.33	81	85	74	FO_1	8
26	2014.1	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FUR	9
27	2015	1	12875.76	12979.44	13084.74	13189.23	13293.72	13398.21	13502.7	13607.19	13711.68	13815.36	13919.85	14024.34	81	85	74	CA_4	1
28	2015	2	10866.96	10957.68	11046.78	11134.26	11229.03	11314.89	11399.13	11492.28	11573.28	11696.4	11821.14	11942.64	81	85	74	CA_3	2
29	2015	3	10297.53	10380.96	10464.39	10549.44	10635.3	10719.54	10803.78	10888.83	10973.07	11063.79	11155.32	11246.85	81	85	74	CA_2	3
30	2015	4	3766.5	6987.87	8161.56	8356.77	8554.41	8768.25	9009.63	9215.37	9312.57	9438.12	9523.17	9608.22	81	85	74	FO_4	4
31	2015	5	3766.5	5905.71	6894.72	7059.96	7230.87	7409.88	7611.57	7787.34	7865.91	7994.7	8091.09	8186.67	81	85	74	FO_3	5
32	2015	6	3766.5	5597.1	6533.46	6690.6	6850.98	7021.08	7215.48	7380.72	7459.29	7563.78	7637.49	7711.2	81	85	74	FO_2	6

Fig. 12: *pay_table_data.xlsx* example, "basic ordered" worksheet

The expanded monthly compensation table for enhanced job levels is generated by seniority_list automatically. The job level sort (ranking) will be consistent for all years and will be based on a monthly compensation sort for a year and longevity selected by the user.

"Enhanced" job levels delineate between full- and part-time positions within each basic job level. See the discussion within the "pay_tables.xlsx" section on the "excel input files" page of the documentation for further explanation.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1		year	jnum	1	2	3	4	5	6	7	8	9	10	11	12	basic_hours	full_hours	part_hours	jobstr	order
2	0	2013	1	2152.2	10220.4	10302.85	10384.45	10466.9	10548.5	10630.95	10712.55	10795	10877.45	10959.05	11041.5	81	85	74	CA_4B	1
3	1	2013	2	1873.68	8897.76	8969.54	9040.58	9112.36	9183.4	9255.18	9326.22	9398	9469.78	9540.82	9612.6	81	85	74	CA_4R	2
4	2	2013	3	2152.2	8687.85	8756.7	8826.4	8896.1	8965.8	9035.5	9105.2	9174.9	9244.6	9314.3	9384	81	85	74	CA_3B	3
5	3	2013	4	2152.2	8065.65	8130.25	8194.85	8259.45	8324.05	8388.65	8453.25	8517.85	8582.45	8647.05	8711.65	81	85	74	CA_2B	4
6	4	2013	5	1873.68	7563.54	7623.48	7684.16	7744.84	7805.52	7866.2	7926.88	7987.56	8048.24	8108.92	8169.6	81	85	74	CA_3R	5
7	5	2013	6	1873.68	7021.86	7078.1	7134.34	7190.58	7246.82	7303.06	7359.3	7415.54	7471.78	7528.02	7584.26	81	85	74	CA_2R	6
8	6	2013	7	2152.2	5158.65	6220.3	6372.45	6526.3	6681.85	6839.1	6997.2	7211.4	7373.75	7483.4	7539.5	81	85	74	FO_4B	7
9	7	2013	8	1873.68	4491.06	5415.32	5547.78	5681.72	5817.14	5954.04	6091.68	6278.16	6419.5	6514.96	6563.8	81	85	74	FO_4R	8
10	8	2013	9	2152.2	4392.8	5292.95	5422.15	5553.05	5684.8	5818.25	5952.55	6134.45	6272.15	6364.8	6412.4	81	85	74	FO_3B	9
11	9	2013	10	2152.2	4081.7	4917.25	5037.1	5157.8	5280.2	5404.3	5529.25	5697.55	5825.05	5911.75	5955.1	81	85	74	FO_2B	10
12	10	2013	11	2152.2	5515.65	5559.85	5604.05	5647.4	5691.6	5735.8	5779.15	5823.35	5867.55	5910.9	5955.1	81	85	74	CA_1B	11
13	11	2013	12	1873.68	3824.32	4607.98	4720.46	4834.42	4949.12	5065.3	5182.22	5340.58	5460.46	5541.12	5582.56	81	85	74	FO_3R	12
14	12	2013	13	1873.68	3553.48	4280.9	4385.24	4490.32	4596.88	4704.92	4813.7	4960.22	5071.22	5146.7	5184.44	81	85	74	FO_2R	13
15	13	2013	14	1873.68	4801.86	4840.34	4878.82	4916.56	4955.04	4993.52	5031.26	5069.74	5108.22	5145.96	5184.44	81	85	74	CA_1R	14
16	14	2013	15	2152.2	2806.7	3375.35	3456.1	3538.55	3621.85	3706	3791	3904.9	3992.45	4051.1	4080.85	81	85	74	FO_1B	15
17	15	2013	16	1873.68	2443.48	2938.54	3008.84	3080.62	3153.14	3226.4	3300.4	3399.56	3475.78	3526.84	3552.74	81	85	74	FO_1R	16
18	16	2013	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FUR	17
19	17	2014	1	2152.2	11030.45	11118.85	11207.25	11296.5	11384.9	11473.3	11562.55	11650.95	11739.35	11827.75	11917	81	85	74	CA_4B	1
20	18	2014	2	1873.68	9602.98	9679.94	9756.9	9834.6	9911.56	9988.52	10066.22	10143.18	10220.14	10297.1	10374.8	81	85	74	CA_4R	2
21	19	2014	3	2152.2	9374.65	9449.45	9525.1	9599.9	9675.55	9750.35	9826	9900.8	9976.45	10051.25	10126.9	81	85	74	CA_3B	3
22	20	2014	4	2152.2	8703.15	8772.85	8842.55	8912.25	8982.8	9052.5	9122.2	9191.9	9261.6	9331.3	9401	81	85	74	CA_2B	4
23	21	2014	5	1873.68	8161.46	8226.58	8292.44	8357.56	8423.42	8488.54	8554.4	8619.52	8685.38	8750.5	8816.36	81	85	74	CA_3R	5
24	22	2014	6	1873.68	7576.86	7637.54	7698.22	7758.9	7820.32	7881	7941.68	8002.36	8063.04	8123.72	8184.4	81	85	74	CA_2R	6
25	23	2014	7	2152.2	5564.1	6710.75	6874.8	7040.55	7208.85	7378	7549.7	7780.05	7956	8074.15	8134.5	81	85	74	FO_4B	7
26	24	2014	8	1873.68	4844.04	5842.3	5985.12	6129.42	6275.94	6423.2	6572.68	6773.22	6926.4	7029.26	7081.8	81	85	74	FO_4R	8
27	25	2014	9	2152.2	4736.2	5708.6	5848	5989.1	6131.9	6275.55	6420.9	6617.25	6766	6866.3	6917.3	81	85	74	FO_3B	9
28	26	2014	10	2152.2	4400.45	5303.15	5432.35	5563.25	5695	5828.45	5963.6	6145.5	6283.2	6376.7	6424.3	81	85	74	FO_2B	10
29	27	2014	11	2152.2	5949.15	5996.75	6044.35	6091.95	6139.55	6186.3	6233.9	6281.5	6329.1	6376.7	6423.45	81	85	74	CA_1B	11
30	28	2014	12	1873.68	4123.28	4969.84	5091.2	5214.04	5338.36	5463.42	5589.96	5760.9	5890.4	5977.72	6022.12	81	85	74	FO_3R	12
31	29	2014	13	1873.68	3830.98	4616.86	4729.34	4843.3	4958	5074.18	5191.84	5350.2	5470.08	5551.48	5592.92	81	85	74	FO_2R	13
32	30	2014	14	1873.68	5179.26	5220.7	5262.14	5303.58	5345.02	5385.72	5427.16	5468.6	5510.04	5551.48	5592.18	81	85	74	CA_1R	14
33	31	2014	15	2152.2	3023.45	3637.15	3724.7	3813.95	3904.05	3994.15	4085.95	4210.05	4303.55	4367.3	4399.6	81	85	74	FO_1B	15
34	32	2014	16	1873.68	2632.18	3166.46	3242.68	3320.38	3398.82	3477.26	3557.18	3665.22	3746.62	3802.12	3830.24	81	85	74	FO_1R	16
35	33	2014	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FUR	17
36	34	2014.1	1	13120.6	13226.85	13333.95	13440.2	13546.45	13652.7	13759.8	13866.05	13972.3	14078.55	14184.8	14291.05	81	85	74	CA_4B	1
37	35	2014.1	2	11422.64	11515.14	11608.38	11700.88	11793.38	11885.88	11979.12	12071.62	12164.12	12256.62	12349.12	12441.62	81	85	74	CA_4R	2
38	36	2014.1	3	11073.8	11167.3	11257.4	11346.65	11442.7	11531.1	11616.1	11711.3	11793.75	11919.55	12046.2	12170.3	81	85	74	CA_3B	3
39	37	2014.1	4	10494.1	10579.1	10664.1	10750.8	10838.35	10924.2	11010.05	11096.75	11181.75	11274.4	11368.75	11461.4	81	85	74	CA_2B	4
40	38	2014.1	5	9640.72	9722.12	9800.56	9878.26	9961.88	10038.84	10112.84	10195.72	10267.5	10377.02	10487.28	10595.32	81	85	74	CA_3R	5
41	39	2014.1	6	9136.04	9210.04	9284.04	9359.52	9435.74	9510.48	9585.22	9660.7	9734.7	9815.36	9897.5	9978.16	81	85	74	CA_2R	6

Fig. 13: *pay_table_data.xlsx* example, "enhanced ordered" worksheet

The "job_dict" worksheet information serves as the calculated source for the basic-to-enhanced job level conversion process when required.

	A	B	C	D	E	F
1	job	full	part	jobstr	full_pcnt	
2	1	1	2	Capt G4	0.6	
3	2	3	5	Capt G3	0.625	
4	3	4	6	Capt G2	0.65	
5	4	7	8	F/O G4	0.6	
6	5	9	12	F/O G3	0.625	
7	6	10	13	F/O G2	0.65	
8	7	11	14	Capt G1	0.65	
9	8	15	16	F/O G1	0.65	
10						

Fig. 14: *pay_table_data.xlsx* example, "job_dict" worksheet

Other worksheets are contained within the *pay_table_data.xlsx* workbook within the **reports** folder for user review.

5.2.3 creating the static 'skeleton' file

Processing script: **make_skeleton.py**

Columns created:

```
['mnum',
'idx',
'empkey',
```

(continues on next page)

(continued from previous page)

```
'mth_pcnt',
'date',
'year',
'pay_raise',
'fur',
'eg',
'retdate',
'doh',
'ldate',
'lname',
'line',
'sg',
'ret_mark',
'scale',
's_lmonths',
'age']
```

empkey	mnum	idx	empkey	mth_pcnt	date	year	pay_raise	fur	eg	retdate	doh	ldate	lname	line	sg	ret_mark	scale	s_lmonths	age
10011102	0	0	10011102	1.000000	2013-12-31	2013.0	1.0	0	1	2014-07-13	1973-02-26	1975-01-29	tooeyoo	1	0	0	12	467	64.465054
10010475	0	1	10010475	1.000000	2013-12-31	2013.0	1.0	0	1	2014-02-05	1975-05-27	1975-05-27	rubelot	1	0	0	12	463	64.903226
10013096	0	2	10013096	1.000000	2013-12-31	2013.0	1.0	0	1	2014-01-08	1977-01-18	1977-01-18	yeloxid	1	0	0	12	443	64.978495
10012178	0	3	10012178	1.000000	2013-12-31	2013.0	1.0	0	1	2016-06-07	1977-11-15	1977-11-15	xayeaue	1	0	0	12	433	62.564516
10014447	0	4	10014447	1.000000	2013-12-31	2013.0	1.0	0	1	2016-10-17	1977-12-09	1977-12-09	finuceu	1	0	0	12	432	62.204301
...	..	..	...	...	...	...	...	..	..	...	...	...	...	..	..	..	..	...	...
20010471	450	5606	20010471	1.000000	2051-06-30	2019.0	1.0	0	2	2051-08-17	2013-03-19	2013-03-19	reloxep	1	0	0	12	9	64.870968
20011588	451	5581	20011588	1.000000	2051-07-31	2019.0	1.0	0	2	2051-08-09	2013-01-16	2013-01-16	aezib	1	0	0	12	11	64.975806
20010471	451	5606	20010471	1.000000	2051-07-31	2019.0	1.0	0	2	2051-08-17	2013-03-19	2013-03-19	reloxep	1	0	0	12	9	64.954301
20011588	452	5581	20011588	0.290323	2051-08-31	2019.0	1.0	0	2	2051-08-09	2013-01-16	2013-01-16	aezib	1	0	1	12	11	65.000000
20010471	452	5606	20010471	0.548387	2051-08-31	2019.0	1.0	0	2	2051-08-17	2013-03-19	2013-03-19	reloxep	1	0	1	12	9	65.000000

[930850 rows x 19 columns]

Fig. 15: skeleton file excerpt

To run the script from the Jupyter notebook, type the following into a notebook cell and run it:

```
%run make_skeleton
```

The *skeleton.pkl* file is a dataframe containing employee data that is independent of list order, meaning that such information is a constant for each individual employee throughout any data model. An example of this would be employee age.

The skeleton file can initially be in any integrated order, but the members of each employee group must be in proper relative order to each other. In other words, the sort order of the members from any employee group must be maintained no matter how the employee groups are meshed together in an integrated list.

The skeleton file is a relatively "long" dataframe. With the sample case study of 7500 total employees, it is almost one million rows long. The skeleton file is organized by data model month ("mnum"), starting with the data for the first month and sequentially "stacking" sequential month data below. The size (number of rows) of the data for each

month is directly related to the number of employees who remain active (non-retired) in that month.

Much of the information in the skeleton file is constant from month-to-month, such as date of hire and last name. Other data does change, such as date and age.

The index of the skeleton is purposefully a duplicate index of the empkeys column(unique employee ID).

Because the skeleton file contains data which is unaffected by the order of an integrated list, it may be calculated once and simply retrieved and resorted to form the basis of subsequent integrated list datasets.

The skeleton file is utilized in the production of both standalone and integrated datasets.

5.2.4 creating datasets

Processing scripts: **standalone.py**, **compute_measures.py**

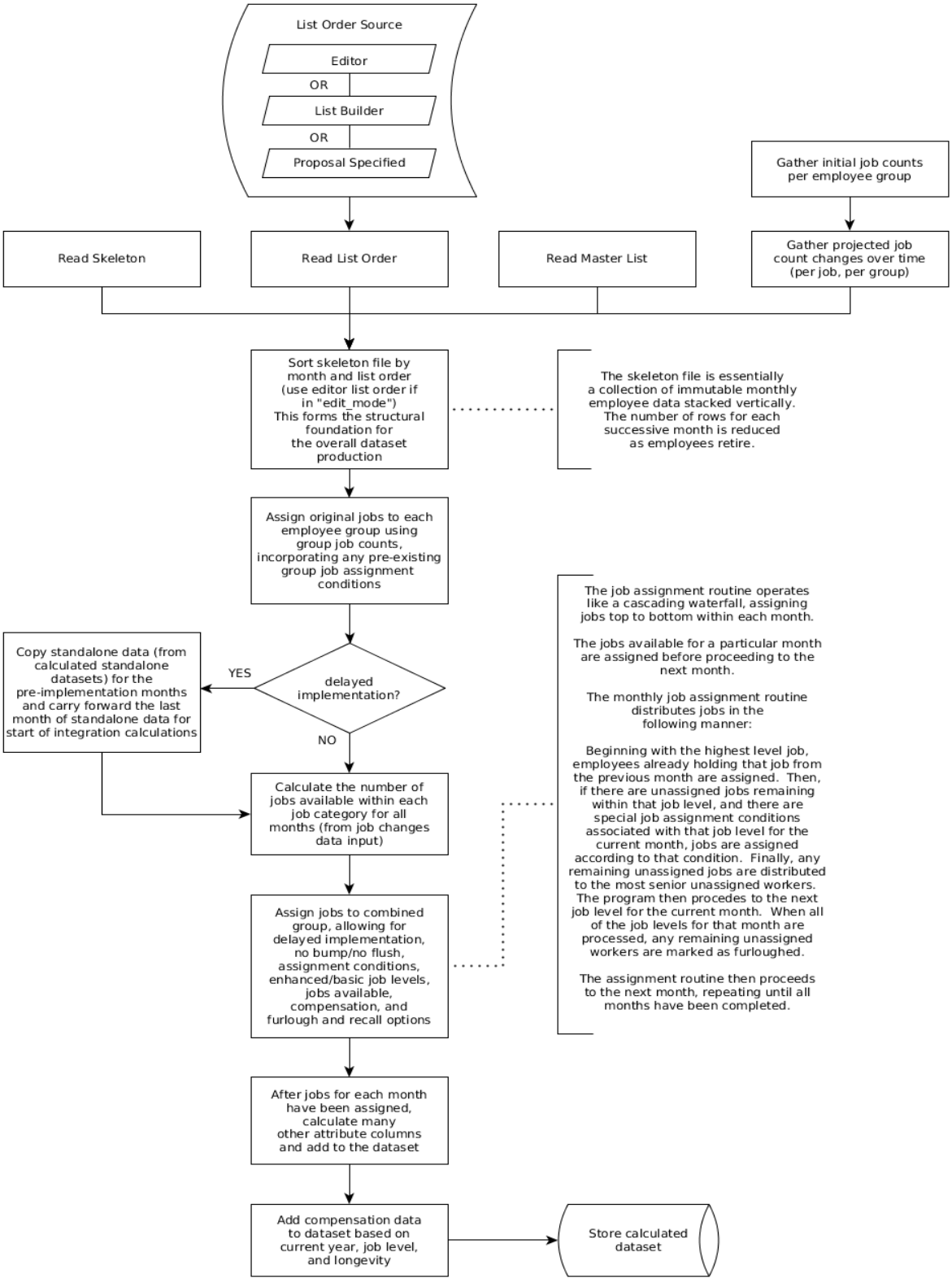


Fig. 16: high-level dataset creation flowchart

Dataset creation is the heart of seniority_list, producing a collection of metrics calculated from a particular integrated list ordering proposal, including any job assignment conditions associated with that proposal, or from standalone list data. The datasets become the source for the objective analysis of potential integrated lists and associated conditions. Integrated datasets are generated using the **compute_measures.py** script. Standalone datasets are generated using the **standalone.py** script.

```
          mnum   idx    empkey  mth_pcnt       date    year  pay_raise  fur  eg    retdate       doh      ldate    lname  line  sg  ret_mark  scale  s_lmonths        age  \
empkey
10011102  0     0     10011102  1.000000 2013-12-31  2013.0  1.0        0    1  2014-07-13 1973-02-26 1975-01-29  tooeyoo  1     0   0         12     467        64.465054
10010475  0     1     10010475  1.000000 2013-12-31  2013.0  1.0        0    1  2014-02-05 1975-05-27 1975-05-27  rubelot  1     0   0         12     463        64.903226
10013096  0     2     10013096  1.000000 2013-12-31  2013.0  1.0        0    1  2014-01-08 1977-01-18 1977-01-18  yeloxid  1     0   0         12     443        64.978495
10012178  0     3     10012178  1.000000 2013-12-31  2013.0  1.0        0    1  2016-06-07 1977-11-15 1977-11-15  xayeaue  1     0   0         12     433        62.564516
10014447  0     4     10014447  1.000000 2013-12-31  2013.0  1.0        0    1  2016-10-17 1977-12-09 1977-12-09  finuceu  1     0   0         12     432        62.204301
...       ..    ...        ...       ...        ...     ...  ...        ..   ..         ...        ...      ...      ...   ..  ..        ..     ...              ...
20010471  450   5606  20010471  1.000000 2051-06-30  2019.0  1.0        0    2  2051-08-17 2013-03-19 2013-03-19  reloxep  1     0   0         12     9          64.870968
20011588  451   5581  20011588  1.000000 2051-07-31  2019.0  1.0        0    2  2051-08-09 2013-01-16 2013-01-16  aezib    1     0   0         12     11         64.975806
20010471  451   5606  20010471  1.000000 2051-07-31  2019.0  1.0        0    2  2051-08-17 2013-03-19 2013-03-19  reloxep  1     0   0         12     9          64.954301
20011588  452   5581  20011588  0.290323 2051-08-31  2019.0  1.0        0    2  2051-08-09 2013-01-16 2013-01-16  aezib    1     0   1         12     11         65.000000
20010471  452   5606  20010471  0.548387 2051-08-31  2019.0  1.0        0    2  2051-08-17 2013-03-19 2013-03-19  reloxep  1     0   1         12     9          65.000000

          new_order  orig_job  jnum  fbff  snum     spcnt  lnum    lspcnt  rank_in_job  job_count      jobp  mlong      ylong       mpay         cpay  cat_order
empkey
10011102  1          0.0       1     1     1.0   0.000257  1     0.000235  1            118        1.008475  468    39.000000  11.041500  11.041500    1.406780
10010475  2          0.0       1     1     2.0   0.000515  2     0.000470  2            118        1.016949  464    38.666667  11.041500  11.041500    2.813559
10013096  3          0.0       1     1     3.0   0.000772  3     0.000705  3            118        1.025424  444    37.000000  11.041500  11.041500    4.220339
10012178  4          0.0       1     1     4.0   0.001030  4     0.000940  4            118        1.033898  434    36.166667  11.041500  11.041500    5.627119
10014447  5          0.0       1     1     5.0   0.001287  5     0.001175  5            118        1.042373  433    36.083333  11.041500  11.041500    7.033898
...       ..         ...       ..    ..    ...        ...  ..         ...  ..           ...             ...  ...         ...        ...         ...         ...
20010471  7412       11.0      1     1     2.0   0.000331  2     0.000331  2            192        1.010417  460    38.333333  16.552050  5723.444050  2.000000
20011588  7386       11.0      1     1     1.0   0.000166  1     0.000166  1            192        1.005208  463    38.583333  16.552050  5769.527070  1.000000
20010471  7412       11.0      1     1     2.0   0.000331  2     0.000331  2            192        1.010417  461    38.416667  16.552050  5739.996100  2.000000
20011588  7386       11.0      1     1     1.0   0.000166  1     0.000166  1            192        1.005208  464    38.666667  4.805434   5774.332504  1.000000
20010471  7412       11.0      1     1     2.0   0.000331  2     0.000331  2            192        1.010417  462    38.500000  9.076931   5749.073031  2.000000

[930850 rows x 35 columns]
```

Fig. 17: integrated dataset file excerpt

Note that seniority_list assigns list percentages and job ranking numbers near zero to the best, most "senior" positions, and higher percentages and numbers to less desirable, most "junior" positions.

Integrated datasets

Integrated datasets build upon a properly-sorted skeleton file. Integrated dataset construction is highly dependent on list order.

The program uses the proposed integrated list orderings from the *proposals.xlsx* workbook to sort the framework for a proposal dataset prior to calculating all of the various attributes which are utilized for analysis. seniority_list may also process list orderings from other sources (the editor tool and the **list_builder.py** script).

The technical process to resort the skeleton file is as follows:

> Use the short-form "idx" column from either a proposed list or the "new_order" column from an edited list to create a new column, "new_order", within the long-form skeleton dataframe.
>
> The ordering information column data from either the proposed or edited list is joined into the skeleton with the pandas data alignment feature using the common empkey indexes. The skeleton may then be sorted by the month ("mnum") and the "new_order" columns.

The generic command to create an integrated dataset is as follows:

```
%run compute_measures <proposal_name>
```

The *compute_measures.py* accepts up to three arguments specifying job assignment conditions from the following list:

```
['prex', 'ratio', 'count']
```

The arguments correspond to the 'prex', 'ratio_cond', and 'ratio_count_capped_cond' job assignment conditions described within the 'settings.xlsx' portion of the 'excel_input_files' section of the documentation.

The following command would run the script for proposal "p1" with both pre-existing and a ratio job assignment conditions as specified in the *settings.xlsx* input file:

```
%run compute_measures p1 prex ratio
```

Other options for integrated dataset construction are defined via the input files, such as job change schedules and recall schedules.

Standalone datasets

Standalone datasets for each separate employee group are also created by the program for comparative use. The creation process is very similar to the integrated process described above, with the exception of the integrated list sorting and job assignment by employee group. After the standalone datasets are created, they are combined into one dataset (retaining the standalone metrics), permitting simple comparison with any integrated dataset.

The following command would create a standalone dataset with a pre-existing job assignment condition. The condition "prex" argument is optional, and is the only conditional argument accepted by the *standalone.py* script.

```
%run standalone prex
```

dataset attributes (columns)

The program generates many attributes or measures associated with the data model(s). These calculated attributes become the source for data model analysis. The attributes marked with an asterisk in the list below are precalculated within the skeleton file. The remaining attributes below are calculated and added to a sorted skeleton file as columns when a dataset is created.

1. mnum* - data model month number
2. idx* - index number (associated with separate group lists)
3. empkey* - standardized employee number
4. mth_pcnt* - percent of month for pay purposes (always one except for pro-rated retirement month)

5. date* - monthly date, end of month
6. year* - contract year for pay purposes
7. pay_raise* - additional (or reduced) modeled annual pay percentage
8. fur* - furloughed employee, indicated with one or zero
9. eg* - employee group numerical code
10. retdate* - employee retirement date
11. doh* - date of hire
12. ldate* - longevity date
13. lname* - last name
14. line* - active employee, indicated with one or zero
15. sg* - special treatment group, indicated with one or zero
16. ret_mark* - employee retirement month, indicated with one or zero
17. scale* - employee longevity year for pay purposes
18. s_lmonths* - employee longevity in months at starting date
19. age* - age for each month
20. snum - seniority number for each month
21. mlong - employee longevity in months for each month
22. ylong - employee longevity in decimal years for each month
23. new_order - order of integrated list or edited integrated list
24. orig_job - employee job held at starting date (or at implementation date for the data model months after a delayed implementation)
25. jnum - job (level) number
26. spcnt - monthly seniority percentage of list (active only, most senior is .0, most junior is 1.0)
27. lnum - monthly employee list number, includes furloughed employees
28. lspcnt - monthly percentage of list, includes furloughed employees
29. job_count - monthly count of jobs corresponding to job held by employee
30. rank_in_job - monthly rank within job held by employee
31. jobp - monthly percentage within job held by employee
32. cat_order - monthly employee job ranking number (rank on list organized from best job to least job)

33. mpay - monthly employee compensation
34. cpay - career pay (cumulative monthly pay)

These attribute names and their definitions are stored within the *dict_attr.py* file, generated with the **build_program_files.py** script.

5.2.5 filtering and slicing datasets

Datasets are large pandas dataframes and may be sliced and filtered in many ways. The user may be interested in reviewing the pandas documentation[43] concerning indexing and selecting data from dataframes (and series) for more detailed information. One of the more common methods particularly helpful with the seniority_list datasets is boolean indexing. Boolean (True/False) vectors may be created by specifying attribute column value parameters within the bracket symbols. Only rows matching a True condition will be returned as part of the new, filtered dataset.

For example, to retrieve all of the data from a dataset named "ds" where employee age was greater than or equal to 45 years:

```
ds[ds.age >= 45]
```

If an additional filter is desired, it can be added by enclosing both filters with parentheses, joined with the "&" symbol. This filter slices for employees greater than or equal to 45 years of age and who belong to employee group ("eg") 1:

```
ds[(ds.age >= 45) & (ds.eg == 1)]
```

Filtered datasets may be assigned to a variable and then further ananlysis conducted on that particular subset of the original dataset. One common usage for this new filtered dataset variable would be as the dataframe input for a plotting function.

5.2.6 visualization

The seniority_list data models are full of calculated metrics ready to be analyzed. The pandas dataframe format was specifically designed for data analysis, and the user is encouraged to explore the datasets with the many methods available with the python scientific stack. In addition to these user-defined analysis techniques, seniority_list offers over 25 built-in visualization functions which may be used to produce highly customizable charts. One of the notebooks included with the program, **STATIC_PLOTTING.ipynb**, demonstrates some of the capability of these functions in an editable format. The **INTERACTIVE_PLOTTING.ipynb** notebook contains

[43] https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html

interactive charts. Please explore the docstrings for specific descriptions of the capabilities, inputs, and options available.

The built-in plotting fuctions follow a default layout convention when applicable, as indicated below:

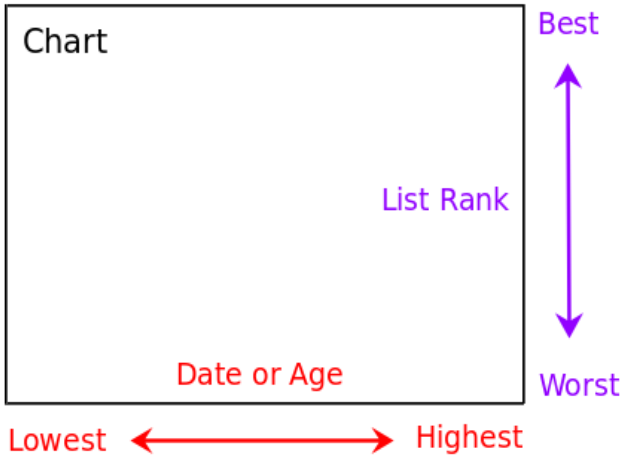


Fig. 18: default layout for built-in plotting functions

In the case where a chart x-axis represents list percentage or job ranking, ordering is presented worst to best, left to right.

As mentioned previously, the plotting functions may receive pre-filtered datasets if the user desires to study a specific subset. Additionaly, many of the functions contain built-in filtering arguments to make this option more convenient.

Example plotting function definition with attribute filtering arguments:

```
def stripplot_eg_density(df,
                         mnum,
                         eg_colors,
                         ds_dict=None,
                         attr1=None,
                         oper1='>=',
                         val1=0,
                         attr2=None,
                         oper2='>=',
                         val2=0,
                         attr3=None,
                         oper3='>=',
                         val3=0,
                         bg_color='white',
                         title_fontsize=12,
```

(continues on next page)

(continued from previous page)

```
                                suptitle_fontsize=14,
                                xsize=5,
                                ysize=10):
```

The "attrx", "operx", and "valx" (substitute x for a common number: 1, 2, or 3) inputs allow the user to specify a dataset filtering operation by specifying the attribute, operator, and value respectively.

For example, in the function argument excerpt below, the visualization would only include employees with a longevity date less than or equal to December 31, 1986.

```
attr1='ldate', oper1='<=', val1='1986-12-31',
```

The following code example demonstrates how the function above could be used within a Jupyter notebook cell to filter the input "p1" dataset to include only employees who are at least 62 years old:

```
import matplotlib_charting as mp

mp.stripplot_eg_density('p1',
                        40,
                        eg_colors,
                        attr1='age',
                        oper1='>=',
                        val1='62',
                        ds_dict=ds_dict,
                        xsize=4)
```

The *slice_ds_by_index_array* function permits another type of specific filtering relating to a certain condition existing within a particular month. The function will find the employee data which meets the selected criteria within the selected month, and then use the index of those results to load data from the entire dataset for the matching employees. For example, a study of the global metrics for only employees who were older than 55 years of age during month 24 could be easily performed. The output of this function is a new dataframe which becomes input for other analysis functions.

seniority_list offers a wide range of chart plotting color schemes. A color dictionary is created as part of the **build_program_files.py** script with matplotlib colormap names as keys and lists of colors as values. All matplotlib colormaps (87 as of September 2017) are now available for plotting. Each color list is automatically generated with a length equal to the number of job levels in the data model + 1. This supplies a color for each job level plus an additional color for a furlough level. Additional customization of the colormaps is available - please see the *matplotlib_charting.py* module plotting function *make_color_list* docstring[44] for full information. To use one of the generated

[44] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.make_color_list

colormaps, call cdict["<colormap name>"] where "cdict" is a variable pointing to the color dictionary.

The "example gallery[45]" section of the documentation showcases more of the visualization capabilities of seniority_list.

The visualization functions are located within the *matplotlib_charting.py* module.

5.3 editor tool

seniority_list excels at outcome analysis of integrated list proposals. A powerful additional feature of seniority_list is the ability to **easily modify list ordering and conditional inputs** in order to achieve equitable outcome results. This task is accomplished through the use of the **editor tool**. The editor tool allows the user to make precise adjustments to integrated list order segments through an intuitive, interactive, and iterative visual process. The integrated outcome result for each modification is presented to the user in near real time, for further analysis and editing.

After a change or edit has been made to an integrated list proposal, the editor tool creates a completely new outcome dataset based on that modification. The user then selects attributes from the new dataset to be viewed and measured and/or compared to another dataset. The tool will display the results for each employee group independently within the main chart area.

For example, the need for an adjustment to a proposed integrated list may be indicated when its differential outcome result reveals significant loss for one work group in terms of job opportunities, career compensation, or another job quality metric while showing significant gains in the same areas for another group. Outcome inequities will certainly exist when a strict formula(s) is applied when combining lists, unless each employee group list contains a relatively equivalent distribution of age, hiring patterns, and jobs. Inequities may also exist due to one or more of the parties attempting to gain advantage for members of their own group at the expense of the other group(s). Whatever the cause, it is relatively easy to alleviate or eliminate outcome equity distortions with the editor tool.

By utilizing the recursive editing feature of the editor tool, the user may **create entirely new integrated list proposals** with objective, quantifiable, and balanced outcomes. Outcome results are observable directly within the tool interface and may be easily validated with the other analysis capabilities of seniority_list.

The editor tool is used within the Jupyter notebook and is run as a bokeh server application usinge the *editor* function from the *editor_function* module. Optional styling arguments may be passed to the function but are not required for it to run.

[45] http://rubydatasystems.com/gallery.html

The bokeh *FunctionHandler* and *Application* class objects are used to run the editor within the notebook, along with the functools "partial" method which permits optional editor function arguments to be used.

```
import editor_function as ef
from functools import partial

from bokeh.application import Application
from bokeh.application.handlers import FunctionHandler

from bokeh.io import show, output_notebook

output_notebook()

handler = FunctionHandler(partial(ef.editor,
                          #insert editor function arguments␣
↪here as desired...
                         ))

app = Application(handler)
show(app)
```

There is no limit to the number of "edits" which may be accomplished - the user may recursively apply as many large or small adjustments as are needed to achieve the desired results. Additionally, the tool may be reset to an original unedited list proposal at any time, so that the user may freely explore the tool confidently.

The editor tool is able to incorporate job assignment conditions (conditions and restictions) within modified list outcomes. (See the "applying conditions" section below).

Outcome values may be displayed within the main chart area using either an **absolute** (actual) attribute outcome of a list proposal, or a comparative **differential** attribute result between two proposals. The user may quickly switch between the two views using a dropdown selection and button click.

The tool offers a filtering feature so that attribute cohorts from each employee group may be isolated and measured. For example, this capability permits comparison of employees from each group hired before a selected year, holding a specific job, or display of selected data relating to a particular data model month.

Animation of monthly data model results, hovering over data points for tooltip information, real-time adjustment of chart colors and element sizing, and other interactive exploration features are included with the tool.

Note: There is a notebook included with seniority_list, **EDITOR_TOOL.ipynb**, which makes it easy to open the tool.

The editor tool interface consists of input controls, the main chart, and a distribution density chart.

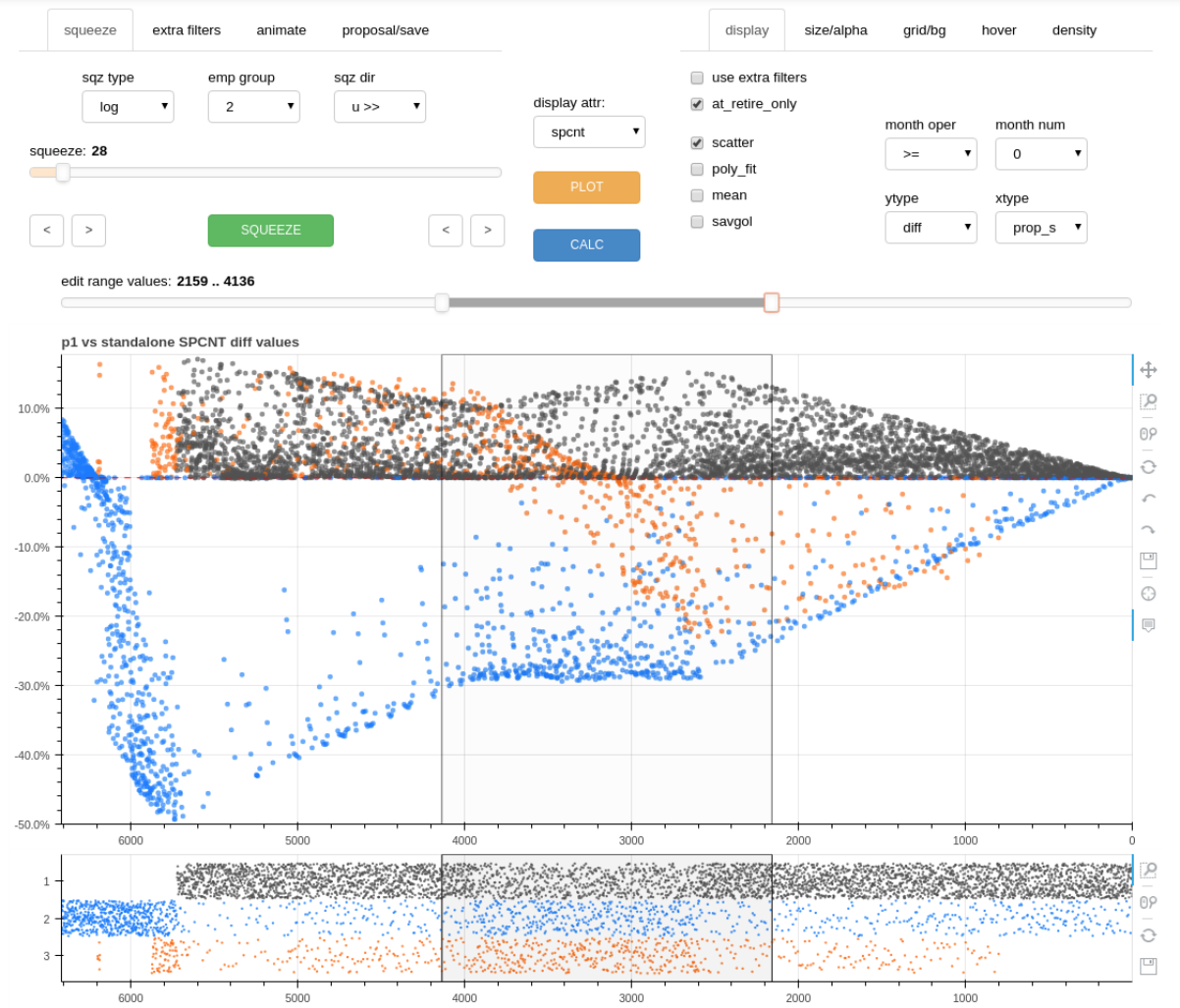


Fig. 19: the editor tool interface

The following flowchart presents the overall list editing process. The sections below will describe the process in detail.

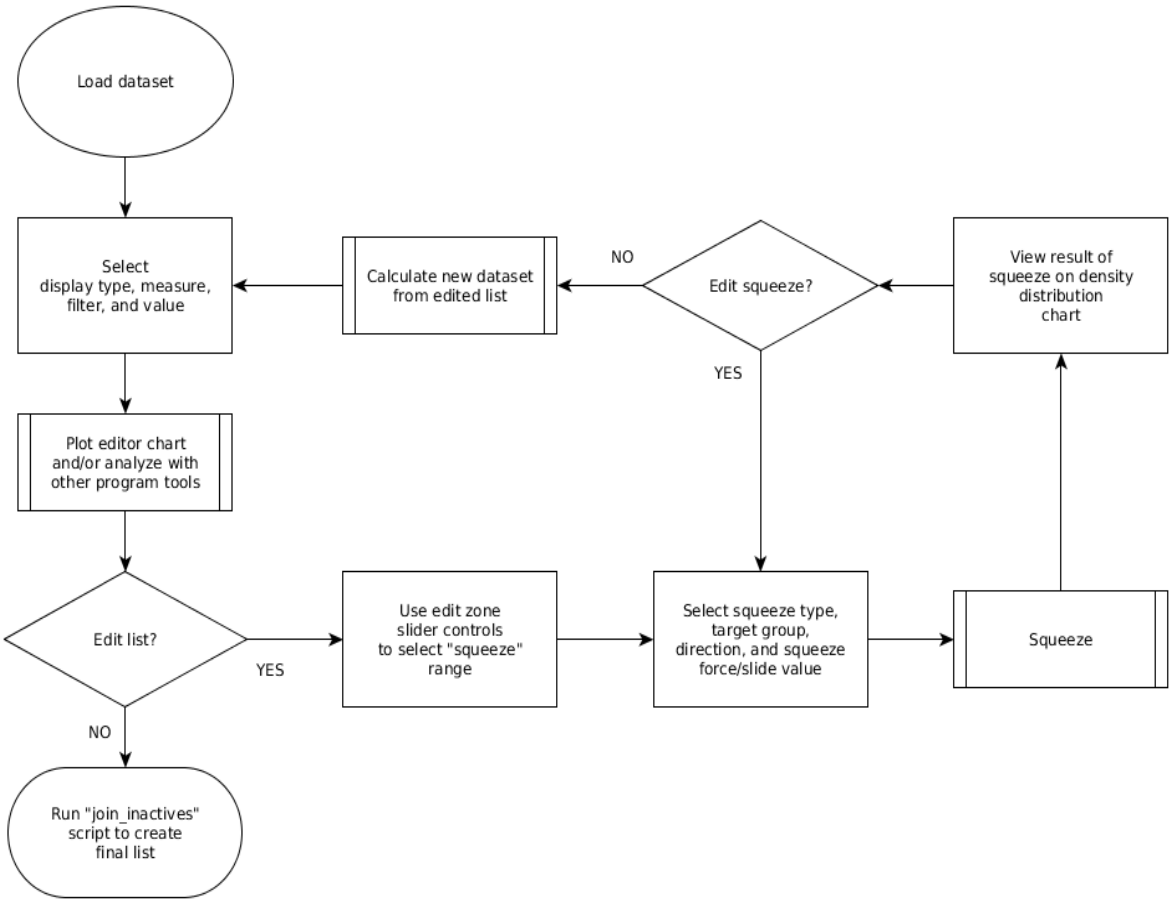


Fig. 20: list editing process

Files created/updated by the editor tool:

With “CALC” button:

- *p_edit.pkl*
- *editor_dict.pkl*

With “SAVE EDITED DATASET” button:

- *p_edit.pkl*
- *editor_dict.pkl*
- *ds_edit.pkl*

With “SAVE EDITED ORDER to proposals.xlsx” button:

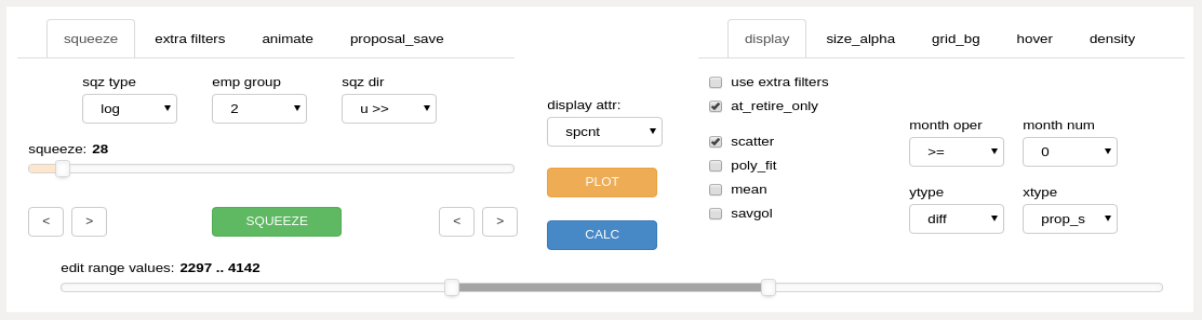
- *proposals.xlsx* (add or replace an “edit” worksheet)

Note: **Edited datasets are not automatically saved**. The user must click on the “SAVED EDITED DATASET” button (located on the proposal/save

tab) to preserve an edited dataset. Previously saved edited datasets will be overwritten unless the *ds_edit.pkl* file is first moved outside of the **dill** folder.

5.3.1 the editor tool controls

The *editor* function itself accepts some arguments, but most of the interaction with the editor tool will be through the editor tool controls, consisting of various sliders, dropdowns, checkboxes, and buttons.



The editor controls are grouped into several sections consisting of the upper left panels, the center section, the upper right panels, and the edit zone slider along the spanning the bottom of the control area.

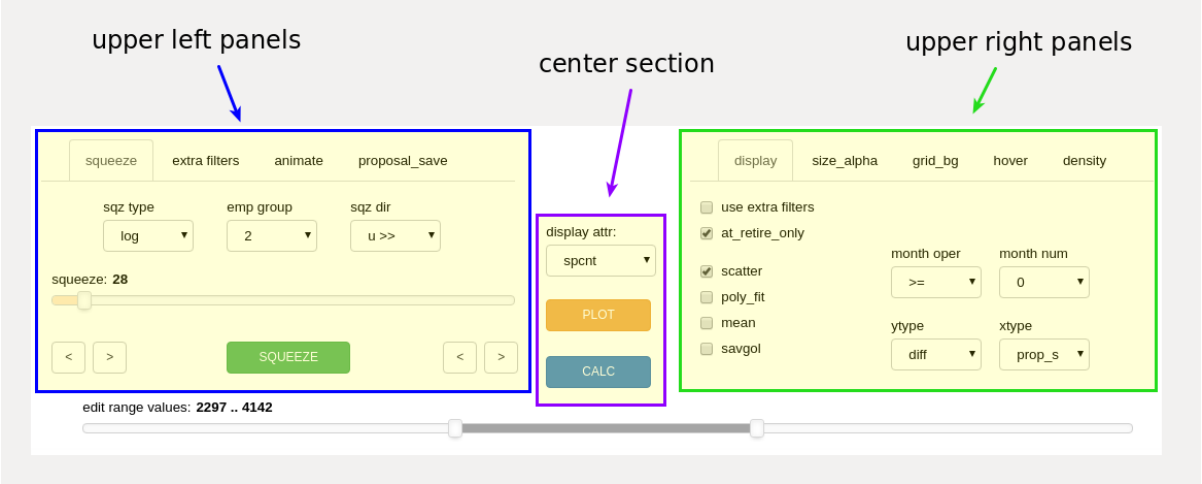


Fig. 21: editor control grouping with the edit zone slider at the bottom (unmarked)

Many editor tool controls are contained within subpanels, selectable by tabs at the top of the tool.

The controls will be introduced below, proceding left to right as they appear within the editor tool. Details on how to use the controls will be covered in the next section.

squeeze panel

This panel is used extensively during the editing process.

Fig. 22: squeeze panel (upper left panels)

- **sq type dropdown**
  - **log**
    - select a log or incremental packing squeeze operation
  - **slide**
    - select a defined positional movement squeeze operation
- **emp grp dropdown**
  - select the employee group (integer code) to move within the selected section of the integrated list
- **sqz dir dropdown**
  - select the direction of movement for the squeeze operation
- **squeeze slider**
  - adjust the single slider control to control:
    - squeeze force if the “sq type” dropdown is set to “log”
    - list position movement if the “sq type” dropdown is set to “slide”
- **edit range toggle buttons**
  - precisely adjust the edit zone cursor lines on the main chart
- **“SQUEEZE” button**
  - command the program to execute a squeeze (list order modification)

extra filters panel

The main chart display output may be further filtered by the inputs on this panel.

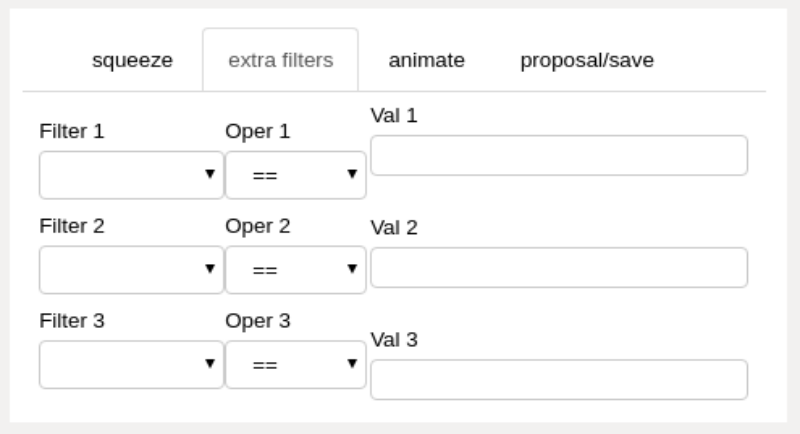


Fig. 23: extra filters panel (upper left panels)

- **attribute dropdowns**
  - select the dataset attribute to filter
- **operator dropdowns**
  - select the mathematical operator to use for the filter
- **value input boxes**
  - type in the value limit for the filter

The additional filters **will not** work unless the "use extra filters" checkbox is checked on the "display" panel.

animate panel

The editor tool is able to display results for any data model month. The animate feature brings this information to life. Monthly results may be quickly displayed sucessively by user controlled forward and backward buttons, automatically with a "PLAY" button, or through the use of a slider control. Outcome results over time may be quickly understood, providing rapid insight into equity distortions or validation of equitable solutions, and everything in between.

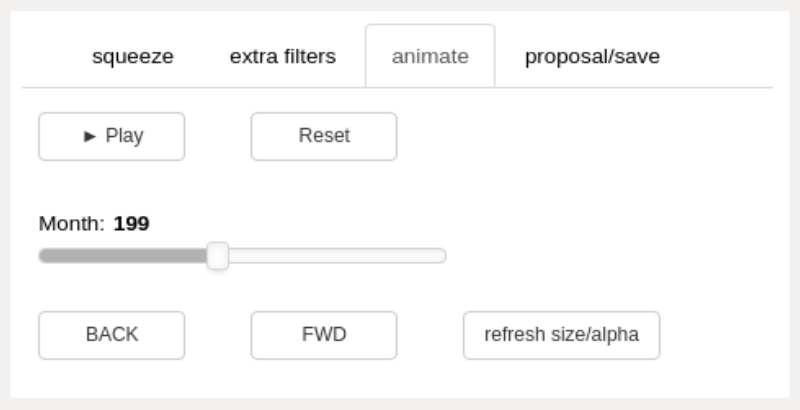


Fig. 24: animate panel (upper left panels)

- **Play button**
  - advance through the data model one month at a time
  - button text will display "Pause" while the animation is running
- **Reset button**
  - reset the data model month to the starting month, month zero.
- **animation slider**
  - use the slider to move forward and backward in time
- **BACK and FWD buttons**
  - move one month in time either direction
- **refresh size_alpha button**
  - if the size or transparency of the scatter markers has been changed using the sliders on the size_alpha tab, use this to apply the changes to the animation output for all months of the data model. Otherwise, only the current month displayed will use the size and alpha selected by the size_alpha sliders.

proposal_save panel

The proposal_save tab contains controls providing inputs related to list orderings and datasets as follows:

- selecting and creating the datasets used by the editor tool
- preserving the results of the editing process

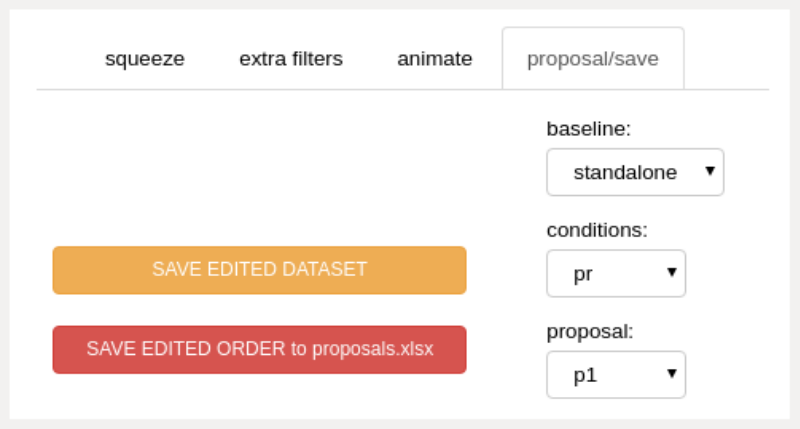


Fig. 25: proposal_save panel (upper left panels)

- **baseline dropdown**
  - select the dataset to use
- **conditions dropdown**
  - select conditions to apply to proposal dataset
    - ‘none’: no conditions
    - ‘prex’: prex
    - ‘count’: count
    - ‘ratio’: ratio
    - ‘pc’: prex, count
    - ‘pr’: prex, ratio
    - ‘cr’: count, ratio
    - ‘pcr’: prex, count, ratio
- **proposal dropdown**
  - edit
    - use the most recent edited dataset as the starting point for each successive list modification
    - the program automatically selects “edit” when a squeeze operation is performed
  - <other dataset names>
    - display comparative or absolute results for other precalculated datasets
- **SAVE EDITED DATASET button** Saves the following files to the **dill** folder:

  - *p_edit.pkl*
  - *editor_dict.pkl*
  - *ds_edit.pkl*
- **SAVE EDITED ORDER to proposals.xlsx button** Adds/updates an "edit" worksheet to:
  - *proposals.xlsx*

center section

Fig. 26: center section dropdown and buttons

- **display attr dropdown**
  - select the dataset attribute for display within the main chart
- **PLOT button**
  - show analysis results as determined by other control inputs
- **CALC button**
  - calculate a dataset after a change of list order, conditional job assignment, or proposal inputs

display panel

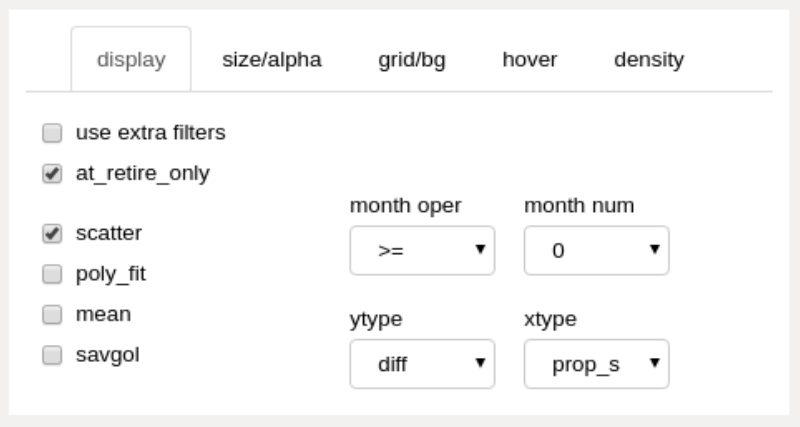


Fig. 27: display_panel (upper right panels)

The display panel contains checkboxes on the left and dropdowns on the right, further divided into upper and lower sections.

upper left

filter checkboxes

- **use extra filters** if checked, use additional filtering as selected on the "extra filters" panel
- **at_retire_only** if checked, only show results for employees in last month of employment before retirement

lower left

display type checkboxes

- **scatter** show results with scatter markers, one marker per employee, color coded by employee group
- **poly_fit** show a polynomial fit line for each group
- **mean** show an exponential moving average line for each group
- **savgol** show a smoothed Savitzky-Golay filter line for each group

upper right

month filter dropdowns

- **month oper** filter data model month using selected mathmatical operator
- **month num** select data model month for filtering

lower right

axis display type dropdowns

- **ytype**
  - **diff** select a differential or comparative result display relative to baseline dataset
  - **abs** select a view of results from proposal or edited dataset only (no comparison)
- **xtype**
  - **prop_s (proposed order, "static" or "starting")** x axis shows original (data model starting month) positioning for results
  - **prop_r (proposed order, "running")** x axis shows updated position for selected data model month
  - **pcnt_s (proposed order, "percentage")** same as "prop_s", but showing list percentage vs list position
  - **pcnt_r (proposed order, "running percentage")** same as "prop_r", but showing list percentage vs list position

size_alpha panel

The controls on this tab control the size and alpha (transparency) of the scatter markers within the main chart.

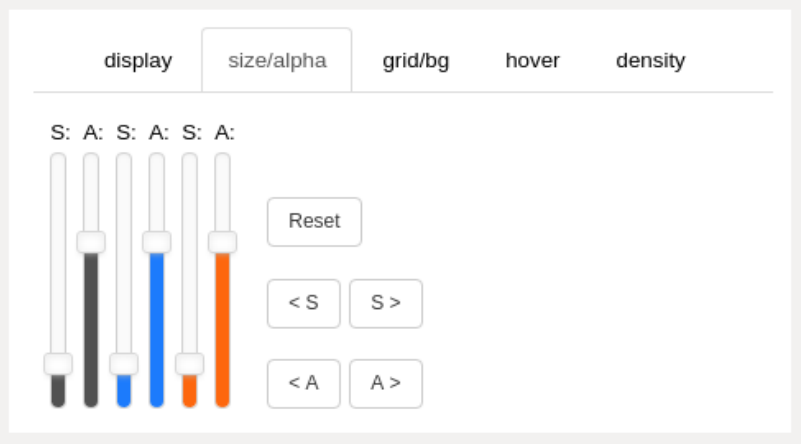


Fig. 28: size_alpha panel (upper right panels), number of sliders will vary with number of employee groups merging

- **sliders**

  - the vertical sliders are color-matched to each employee group color
  - each employee group within the data model will have a pair of sliders:
    - "S" will adjust the size of the scatter markers on the main plot
    - "A" will adjust transparency (alpha) of the scatter markers
  - the program will automatically create the proper number of sliders for each case study

- **Reset button**
  - set size and alpha sliders to default values
- **<S and S> buttons**
  - decrease or increase the size of all markers
- **<A and A> buttons**
  - decrease or increase the alpha value of all markers

Size/alpha adjustment occurs immediately on the main chart (no need to use plot button).

grid_bg panel

Fig. 29: grid_bg panel (upper right panels)

- **chart bg/grid and edit zone checkboxes**
  - apply the "chart / edit_fill" or "grid / edit_line" color and alpha value to the checked areas.
  - the updates only occur when a color value or alpha value changes

  - if "chart bg/grid" is checked, the top color dropdown controls the chart background color and the bottom color dropdown controls the chart grid color
  - if "edit zone" is checked, the top color dropdown controls the color of the fill between the edit zone cursors and the bottom color dropdown controls the color of the cursor lines

- **chart / edit_fill dropdown**
  - select the color of the corresponding areas
- **grid / edit_line dropdown**
  - select the color of the corresponding areas
- **alpha dropdowns**
  - select the alpha (transparency) of the corresponding areas
- **Reset button**
  - reset the colors, alphas, and edit_line_width to default values
- **minor grid lines checkbox**
  - show minor grid lines when checked
  - color and alpha is locked to main grid line color and alpha as they exist when checkbox is checked
- **edit_line_width dropdown**
  - select the width of the edit zone cursor lines

Grid/bg adjustment occurs immediately on the main chart (no need to use plot button).

hover panel

The hover feature will provide selected data as tooltips when the mouse cursor is positioned over a scatter marker.

Use the "PLOT" button to refresh/include selected hover attributes within calculated chart data. Ensure that the chart hover tool is active to display tooltips (click on hover tool icon to display vertical blue line next to the icon).

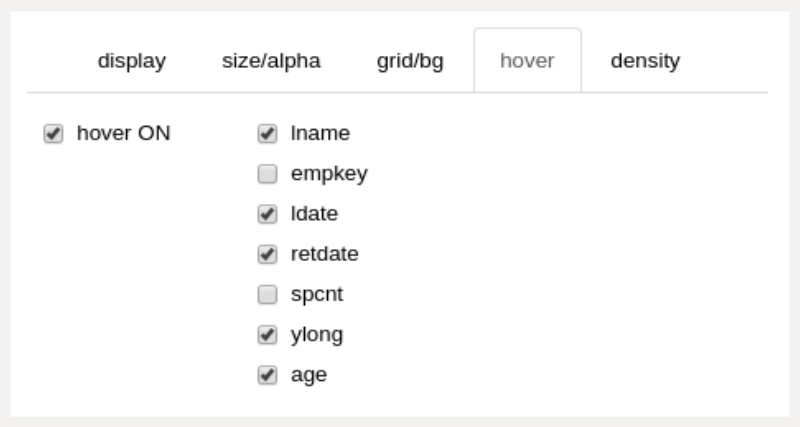


Fig. 30: hover panel (upper right panels)

- **hover ON checkbox**
  - turn the hover feature on and off
  - unchecking this feature when it is not needed will slightly improve the performance of the editor tool
- **hover attributes checkboxes**
  - select the attributes to display as tooltips
  - if the chart display attribute is the same as a selected hover attribute, the hover attribute will not display as a tooltip

density panel

Fig. 31: density panel (upper right panels)

- **"S" slider**
  - controls the stripplot (density) chart marker size

- **“A” slider**
  - controls the stripplot chart marker alpha (transparency)

Density adjustment occurs immediately on the density chart (no need to use plot button).

edit zone slider

Fig. 32: edit zone slider and chart cursor lines delineating the edit zone

The edit zone slider is used to select a section of an integrated list proposal. The selected section is used by the “squeeze” routine when editing list order.

Each end of the slider range may be adjusted independently with a mouse click and drag of an end handle or the entire range may be moved with a click and drag of the slider section between the end handles.

The slider movement is used to position vertical cursor lines within the main and density charts in real time. If data for a future month is displayed within the main

chart, future list positioning data is converted for correct display within the density chart which always displays data for the complete integrated list proposal. Therefore, the main chart cursor lines and the density chart cursor lines will often be misaligned vertically. This is normal due to different x axis scaling between the charts.

Precise adjustment of the cursor lines is available with the toggle buttons found on the "squeeze" panel.

5.3.2 using the editor tool

The editor tool is really an analysis tool and a corrective/creative tool in one. Datasets which have already been generated can be analyzed in many ways, both by themselves and compared to each other without any editing taking place. When equity outcome distortions are apparent, the tool may be used to adjust input list order to reduce the distortions. A new outcome dataset is created based on the modified input, which is in turn available for analysis and further modification. The end result may be an entirely new list proposal which has been created by the editor tool based upon outcome equity measurements.

Common actions when using the editor tool include:

- apply various filters to datasets and then click the "PLOT" button to see the results
- set tooltips to "hover to discover" further information associated with each employee scatter marker - use the "PLOT" button to load
- select an "edit zone" using the edit zone slider and modify list order input by using the "SQUEEZE" button, then calculate the outcome with the "CALC" button
- animate outcome datasets over time
- adjust colors and sizes of many of the chart elements in real time
- compare datasets or simply see the results for one dataset using the "ytype" selection
- control the x axis display to show original list position or an updated future list position

The normal workflow centers on editing the integrated list order using the controls on the "squeeze" panel and checking the results using the "display attr" dropdown with the "PLOT" button. With practice, the user will find that using the tool is relatively easy and visually intuitive.

Datasets (models) for use within the editor tool are specified with the "baseline" and "proposal" dropdown selections on the "proposal_save" panel.

If "edit" is selected from the "proposal" dropdown, and an edited dataset is not found by the program, the program will default to the first of the integrated datasets listed

within the *proposal_names.pkl* file as a starting point. After the first "calculate" button execution, the program will automatically use the newly created "ds_edit" dataset for each subsequent operation.

attribute selection

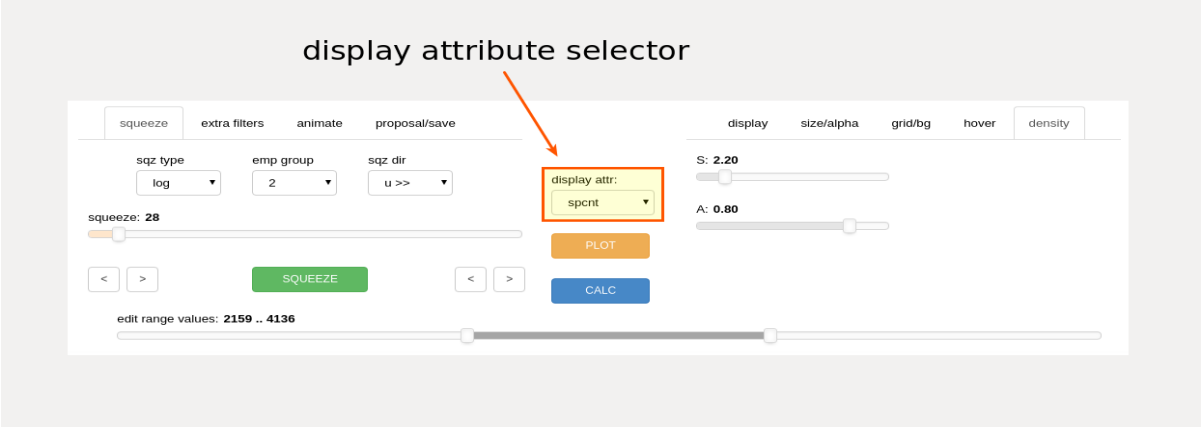


Fig. 33: the editor display attribute dropdown control

display attribute selector

- **display attr dropdown** -select the dataset attribute to display within the main chart

This selection controls the metric (attribute) values which will be displayed within the main chart. To display a different attribute, use the dropdown to pick another measurement and then click the "PLOT" button. Possible attribute selections include list percentage, career compensation, job levels, and others. Further filtering (see below) is available to limit displayed results to a particular month, group, or other targeted attribute(s). In the image below, note that to the right of the dropdown, a filter has been set to show only employees in their retirement month ("ret_only" checkbox). The differential chart is presenting information associated with the final month seniority list percentage ("spcnt") for all employees.

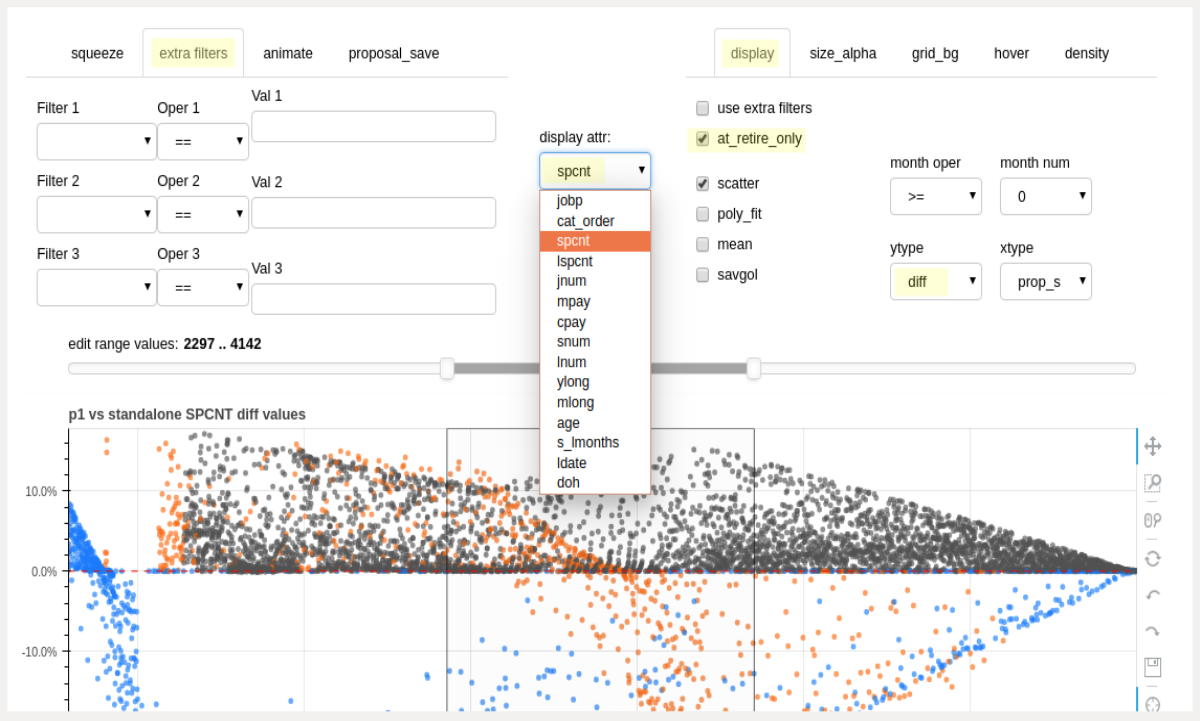


Fig. 34: attribute selection dropdown

basic filters

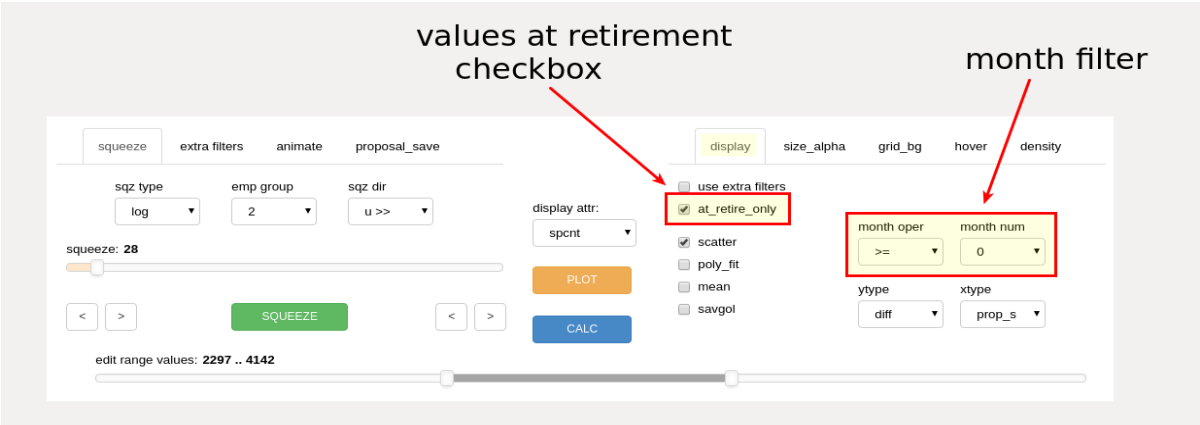


Fig. 35: retirement only and month filter controls (display tab)

values at retirement checkbox

- **ret only checkbox**
  - display only results for employees as measured in their final month of working just prior to retirement

month filter

- **month operator dropdown**
  - select operator (such as '>=' or '==') to be used with month number dropdown for display month filtering
- **month number dropdown**
  - select month number to be used for display filtering

The month filter is always active, except when the animation feature is in use. Results for all months are displayed by selecting month '0' combined with the '>=' operator.

Month '0' represents the start month for the data model.

A single month of data may be displayed by using the '==' operator combined with a selected month number.

The user may remove the display of pre-implementation information by setting the operator to '>=' combined with the implementation month value.

extra filters

The main chart display output may be further filtered if the user wishes to measure specific segments of the employee group(s). This filtering does not affect overall dataset calculations - only the chart display output is filtered. Up to three display filters may be used simultaneously. This extra filtering is in addition to any month or retirement filtering from the display panel.

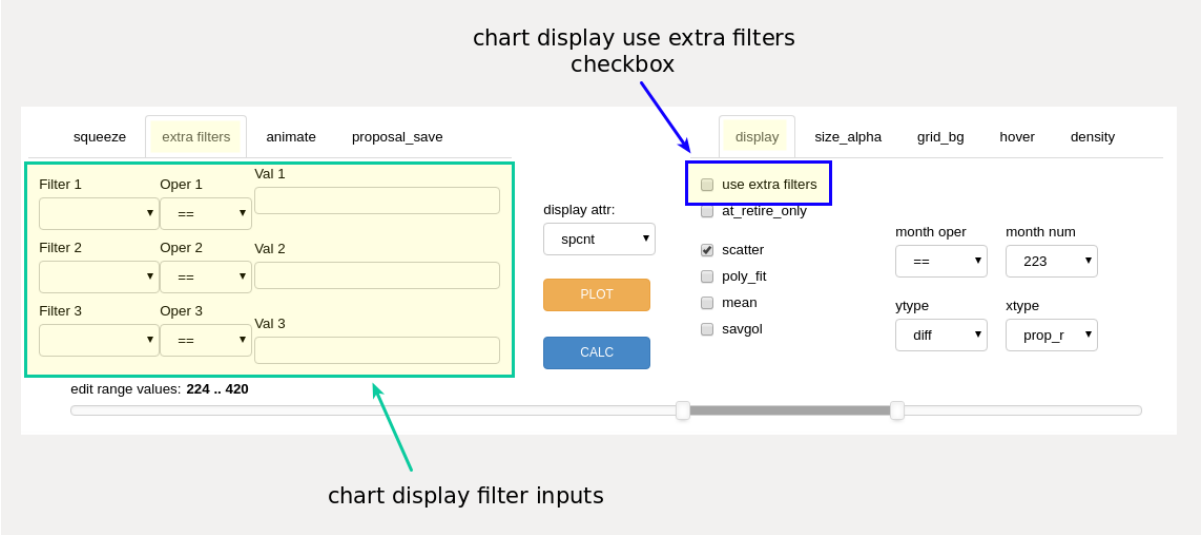


Fig. 36: the editor display filter controls

The additional filters **will not** work if the "filter" checkbox is not checked. Example filters are "ldate <= 1999-12-31", "jnum == 6", or "ylong > 30"

Fig. 37: example filtering: seniority list percentage for employees retiring with 35 or more years of longevity, absolute values

marker style and axis mode selection

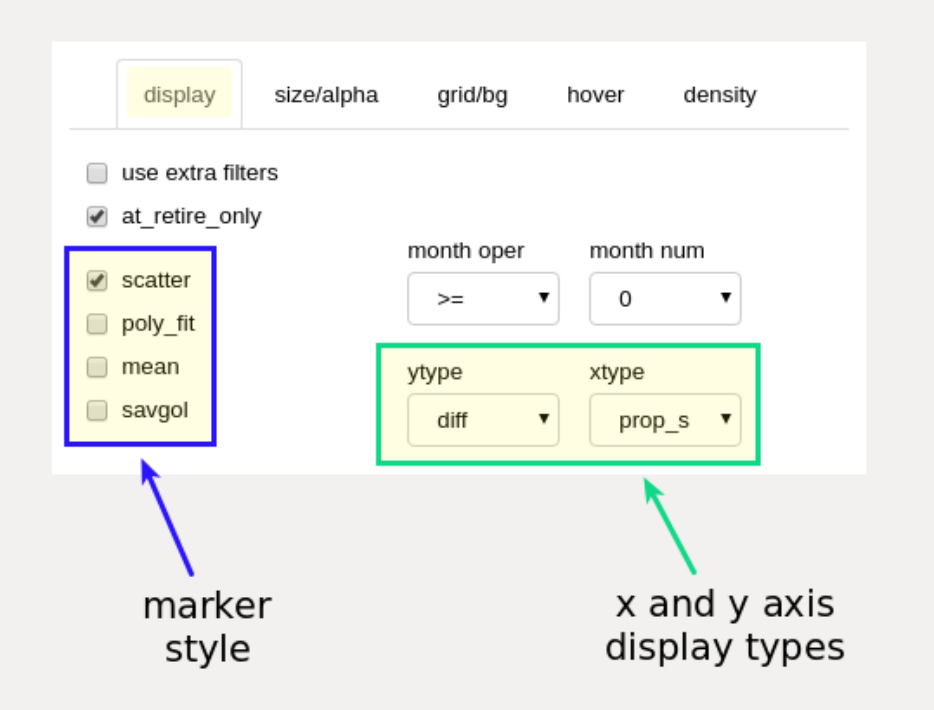


Fig. 38: marker style and axis type selection (display tab)

marker style selection

These checkboxes control the aggregate display type for the information presented within the main chart display. All styles differentiate between employee groups by color.

- **scatter checkbox**
  - show results as a scatter chart, one dot for each employee result
- **poly_fit checkbox**
  - show the results as a smooth polynomial fit line, one line per employee group
- **mean checkbox**
  - show the results as an average line, one line per employee group
- **savgol checkbox**
  - show the results as a smoothed line, calculated using a Savitzky-Golay filter, one line per employee group

Changes to marker style are reflected immediately on the main chart (no need to use plot button). More than one style may be selected at the same time.

axis mode section

- **ytype dropdown** Note: See further description and discussion in the "differential display mode" and "absolute display mode" sections below.

  abs (absolute values)

  – display the actual results for the input (proposal) dataset (non-comparative).

  diff (differential values)

  – display the difference between the same selected dataset attribute between two specified datasets, normally standalone data and another calculated dataset, which could be the results of a proposed integrated list model or an edited model produced from the editor tool itself. Displayed differences may be between any datasets - there is no requirement to compare only with standalone data.

- **xtype dropdown prop_s**

  – an abbreviation for "proposal order, static (or starting)" and is likely to be the most used setting for the display. The main chart displays results with the employee group(s) ordered along the x axis according to the full initial underlying integrated list, which may be a proposal submitted by one of the parties or an edited list. All squeeze operations are performed according to proposal order.

  prop_r

  – an abbreviation for "proposal order, running". The main chart displays results with the employee group(s) ordered along the x axis according to the underlying integrated list as it would exist at a particular designated month within the data model. With this display, employees advance position ranking as retirements or other factors allow, and those new list positions are used for the x axis positioning.

  pcnt_s

  – "percentage, static". Same as the "prop_s" display type, with list percentage displayed instead of static list order (seniority) number.

  pcnt_r

  – "percentage, running". Same as the "prop_r" display type, with list percentage displayed instead of running list order (seniority) number.

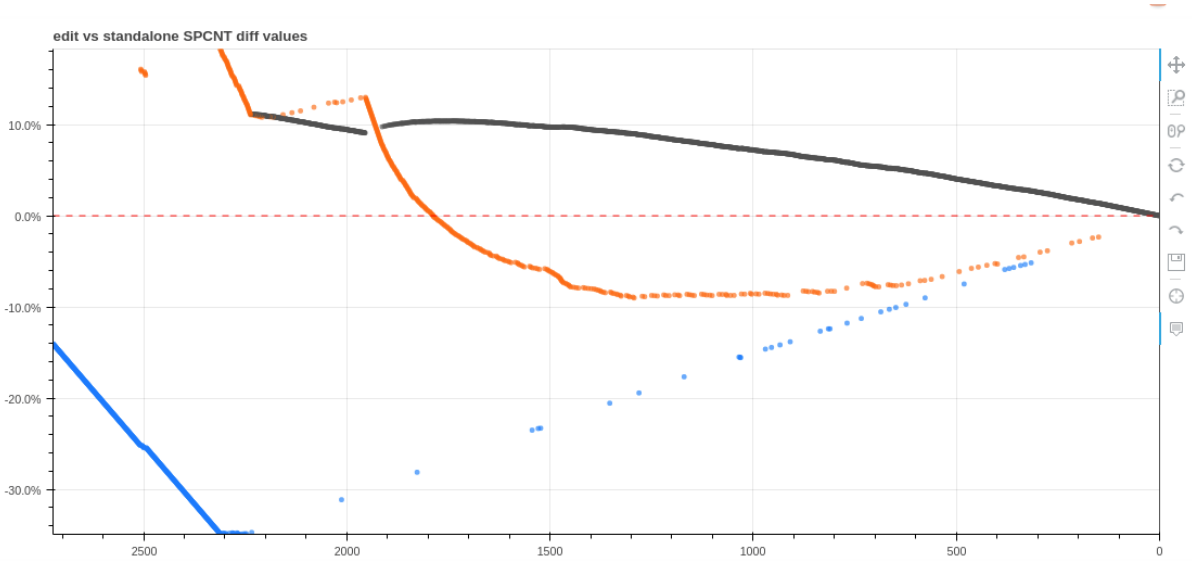


Fig. 39: editor chart ordered by static integrated list order for a future month

execution buttons

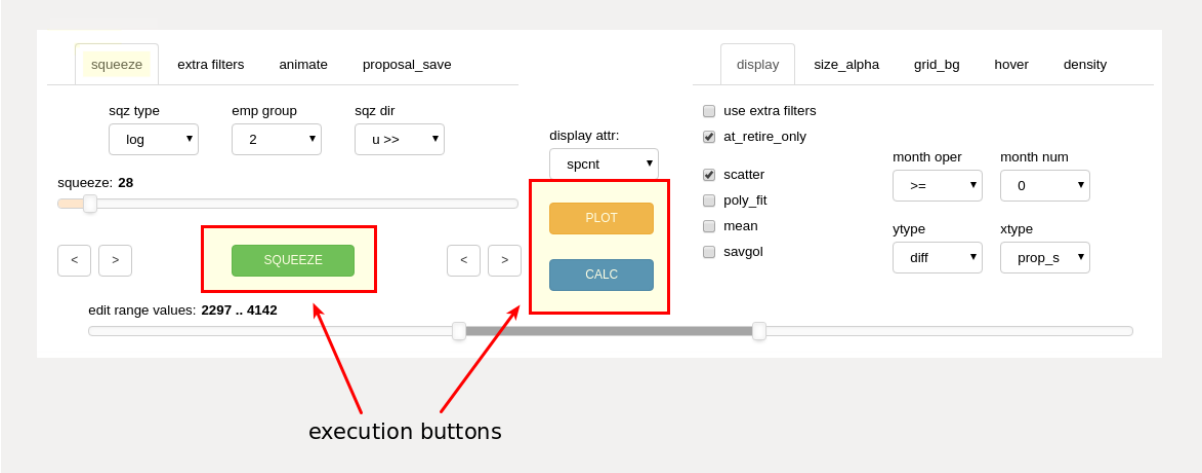


Fig. 40: the editor execution buttons (the "SQUEEZE" button is located on the squeeze panel)

execution buttons

- **SQUEEZE button**
  - executes a squeeze operation, using the input values from the squeeze slider, the edit zone range slider, and the squeeze selection dropdown boxes
- **PLOT button**

- uses the attribute selection, ytype and xtype selections, month and other applicable filters to display data on the main chart. Plot inputs may be changed between plot displays.
- all results correspond to the last *calculated* dataset, based on all non-squeeze editor tool inputs.
- refreshes or creates the data source for hover tooltips
- resets the main chart edit zone cursor lines and slider position (Note: use the bokeh "reset" tool button to reset chart scaling, further explained in the "using the bokeh chart tools" section below)

- **CALC button**
  - calculates a new dataset based on the most recent squeeze operation and displays the results on the main chart display.

It is not necessary to recalculate the dataset to view various attribute results associated with a resultant dataset. Simply select the desired attribute and filter(s) and click the "PLOT" button. Calculation using the "CALC" button is only required when actually modifiying integrated list order after a squeeze operation.

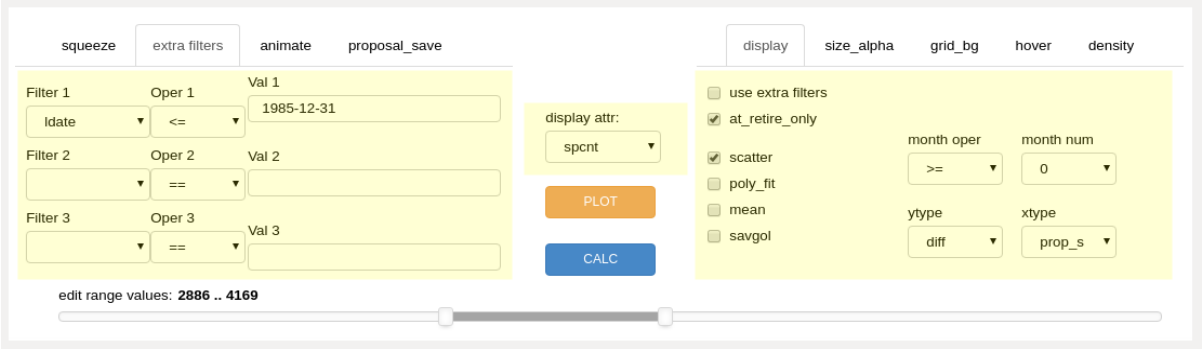


Fig. 41: items highlighted may be changed without recalculating - use the "PLOT" button

differential display mode

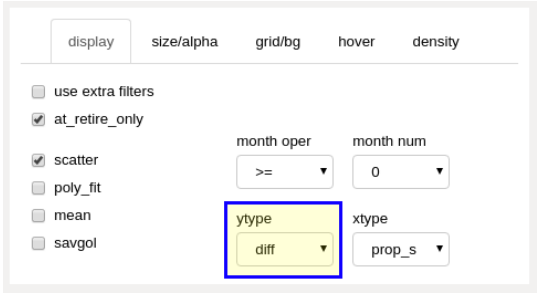


Fig. 42: differential ytype dropdown selection (display panel)

When a "diff" ytype (y axis) is selected, the **difference** between attribute calculations from the proposal and baseline datasets will be displayed within the main chart area. Distortions are generally identified by inequitable positive or negative deviation from the norm, as represented by the zero line on the differential chart. By default, the norm (baseline) is defined as the standalone outcome results, but any calculated dataset may be set as the baseline for comparison.

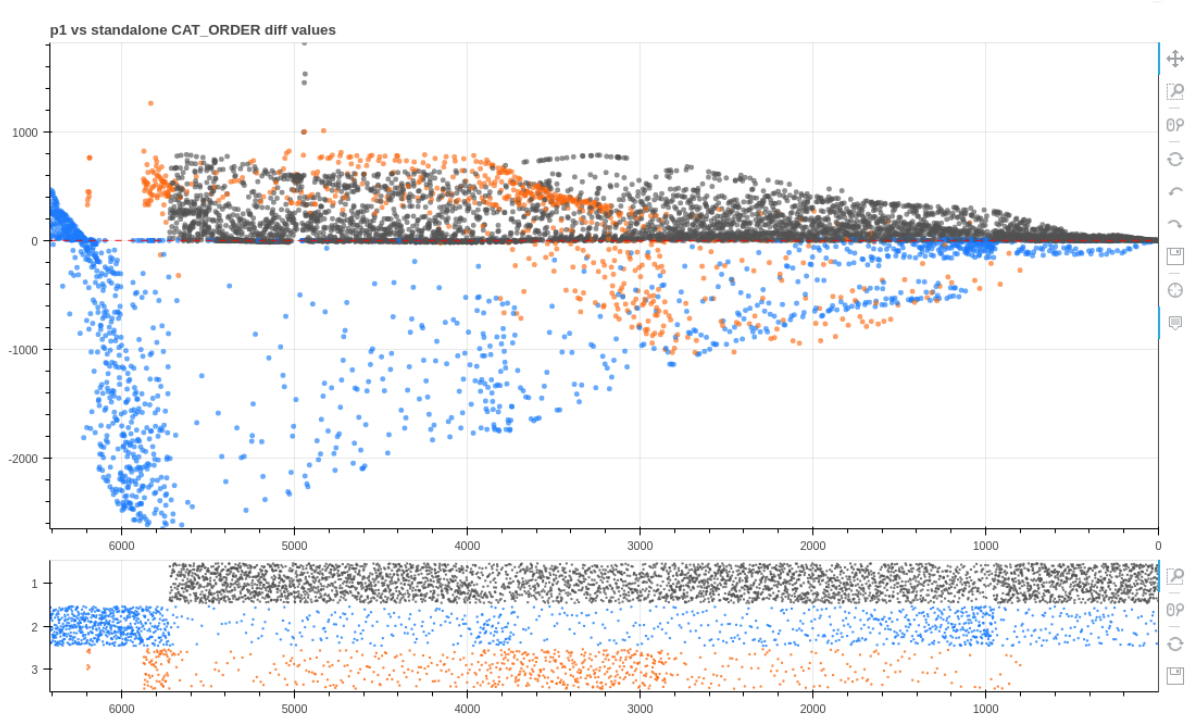


Fig. 43: cat_order (job value) differential, proposal p1 vs. standalone

The chart below displays the seniority percentage *difference* at retirement between sample proposal "p1" and standalone outcomes. In this case, only the average differential is displayed. This type of output is selected with the "mean" checkbox control. The results reveal group 1 employees retiring at a better (more senior) combined seniority list percentage than under projected standalone conditions, and group 2 experiencing a large negative result under the "p1" proposal, as compared to standalone projections.

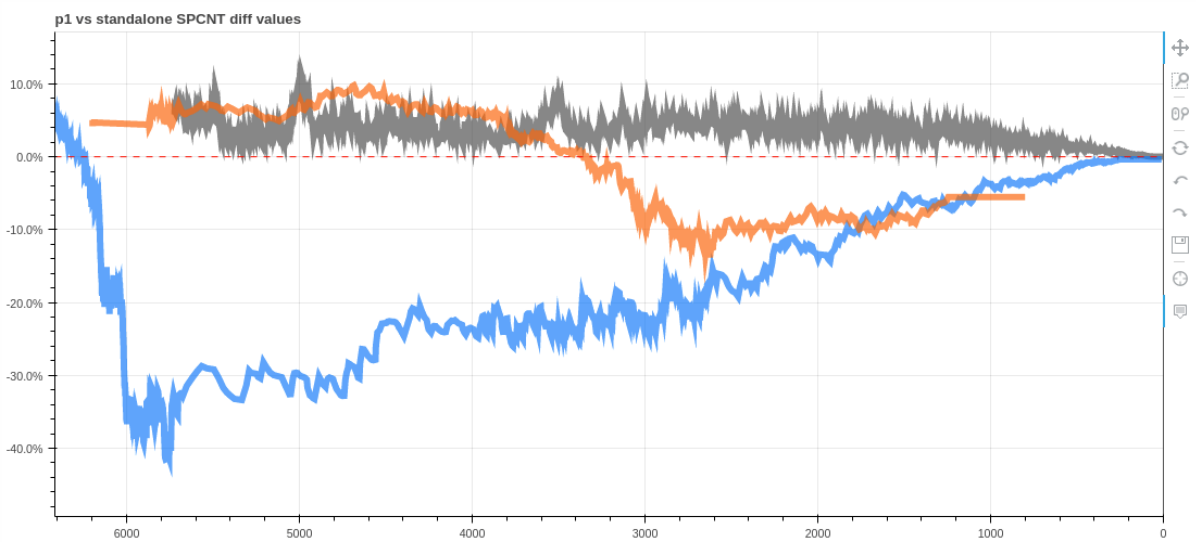


Fig. 44: average differential values proposal "p1" vs. standalone seniority percentage at retirement

absolute display mode

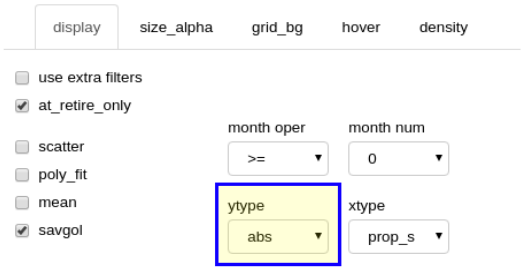


Fig. 45: absolute mode dropdown selection (display panel)

When an "abs" ytype (y axis) is selected, the **actual** attribute calculation values from the dataset defined by the "proposal" dropdown ("proposal_save" panel) will be displayed within the main chart area. Subsequent displayed values will be from the last calculated **edited** dataset.

The chart below displays the *actual* seniority list percentage for all 3 groups within the sample dataset at the time of retirement for each employee, smoothed with a Savitzky-Golay filter ("savgol" checkbox on the "display" panel). These results are derived from the same input as the average differential chart above. The results indicate group 2 employees on average will not advance up through the integrated seniority list as far as employees from the other groups prior to reaching retirement under proposal "p1".

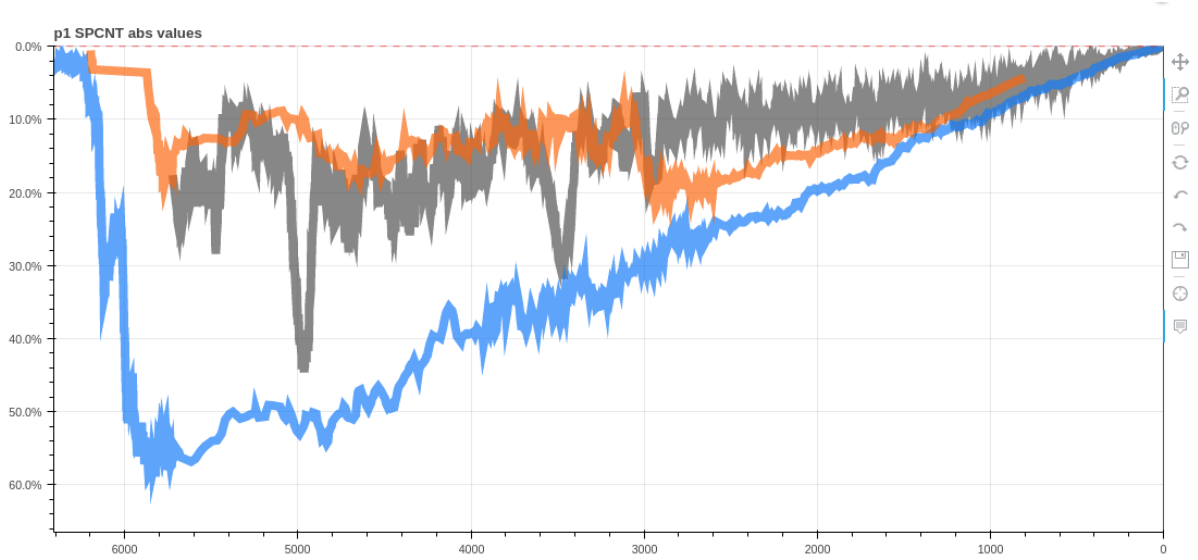


Fig. 46: smoothed absolute (actual) values proposal "p1" seniority percentage at retirement

applying conditions

Job assignment conditions may be applied to datasets created with the editor tool function by selecting a condition from the "conditions" dropdown selection on the "proposal_save" panel. Conditions associated with baseline datasets are incorporated and applied when the datasets are constructed with the **compute_measures.py** script. Conditions applied with the editor tool only apply to datasets created by the editor tool (designated with the "proposal" dropdown selection).

The editor tool will apply job assignment conditions during the construction of the proposed integrated dataset according to the cond_dict dictionary key, value pairs below, with the "condition" dropdown selection as the key and the corresponding value as the condition list to apply.

```
cond_dict = {'none': [],
             'prex': ['prex'],
             'count': ['count'],
             'ratio': ['ratio'],
             'pc': ['prex', 'count'],
             'pr': ['prex', 'ratio'],
             'cr': ['count', 'ratio'],
             'pcr': ['prex', 'count', 'ratio']}
```

The conditional job assignment parameters are set within the *settings.xlsx* spreadsheet input file, and are converted to a python dictionary file for use within the program during the execution of the **build_program_files.py** script. See the "excel input files" section for the definitions of the various job assignment conditions. Advanced users may wish to directly access and modify dictionaries associated with conditional job

assignments located within the *dict_settings.pkl* file when experimenting with "what if" scenarios.

Results for a precalculated dataset may be viewed by using the "proposal" dropdown selection and then clicking the "CALC" button.

Note: Job assignment conditions (as defined by the "conditions" dropdown selection) will be included within calculated datasets produced and viewed with the editor tool. When analyzing a previously calculated dataset, be sure that the selected conditions match the conditions applied to the original dataset if results containing equivalent conditions are desired. Conversely, the effect of particular job assignment conditions may be analyzed by comparing a dataset without conditions to a baseline dataset which included conditions.

squeezing

If one group is enjoying a windfall in all or only a section of the proposed list while another group(s) is suffering a loss, the corrective action is to reduce the list position of the windfall group relative to the other group(s), and then recalculate for further analysis and adjustment.

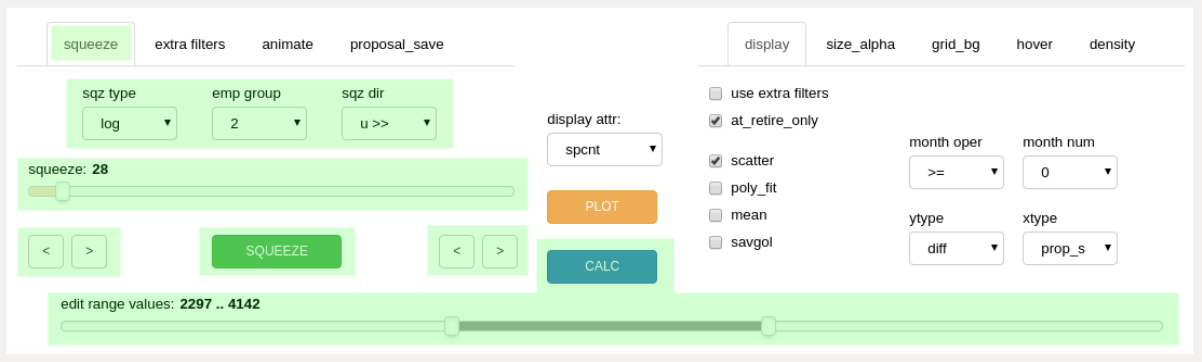


Fig. 47: controls used when editing an integrated list

A slice of the differential display may be selected by using an interactive slider control (labeled "edit zone" above) which positions two vertical lines on the chart. The area of the chart between the lines represents a section of the integrated seniority list (the "edit zone").

If a future month(s) filter has been applied to the data displayed, the selected section will be internally converted to include all employees who have already retired prior to the future month. This is done because all editing occurs to the initial integrated list ordering and so that section selections and calculated results sync with the cursor lines display. The lower density chart cursor lines are automatically corrected to show the equivalent beginning month section.

Once the section of the list has been selected, an algorithm within the editor tool is then utilized to "slide" or "squeeze" the members from one of the original employee groups up or down the list. This action creates a new modified order, while maintaining proper relative ordering within each employee group.

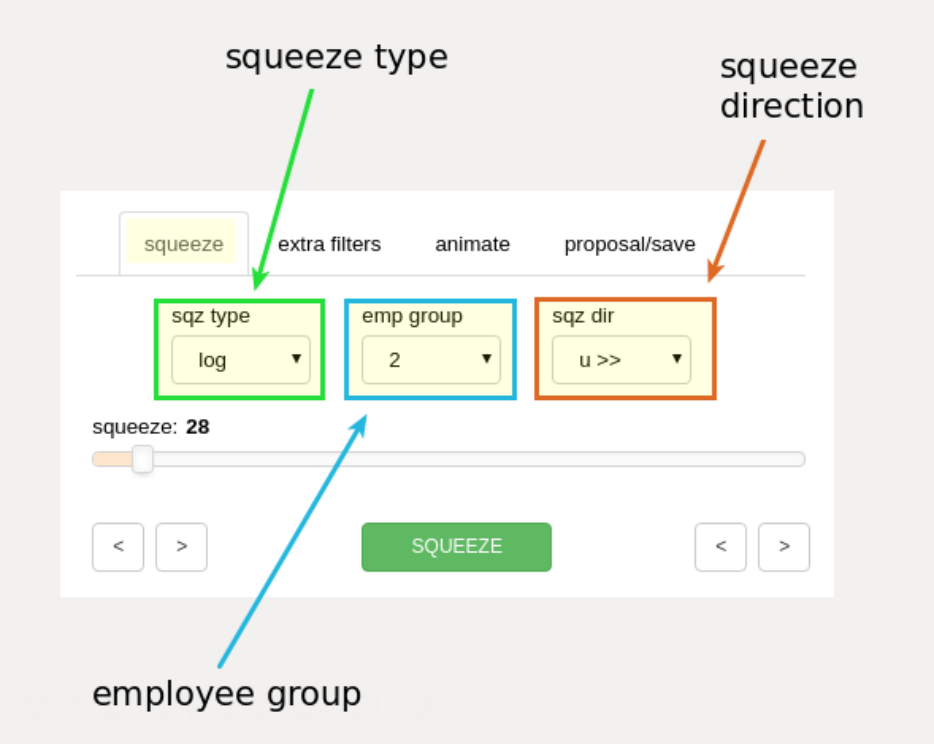


Fig. 48: squeeze selectors (squeeze tab)

The squeeze algorithm works by one of two basic ways. The first method is a logarithmic move, the second is a position slide move.

The **logarithmic** method first selects all of the employees from the defined list section, and sets aside the employees who are not part of the employee group to be moved. The "emp grp" dropdown box selection sets the group to be moved. The employees from the target group are then spread throughout the list section in a pattern determined by the "sqz dir" (squeeze direction) and the value set with the "squeeze" slider control. The logarithmic squeeze "packs" employees in one direction or the other so that there is an ever-increasing density of employees filling list slots. The severity of the density differential is set with the "squeeze" slider. If the squeeze is set to the lowest possible value of 1, all employees from the target group are spread evenly within the selected section.

The position **slide** method simply moves all employees from the selected employee group a certain number of positions one way or the other. The number of positions to move is set with the "squeeze" slider setting. If the requested move would cause target employees to be moved outside of the selected list section, the employees will "pile

up" or be compressed at the edge of the section as necessary to stay within the section boundary.

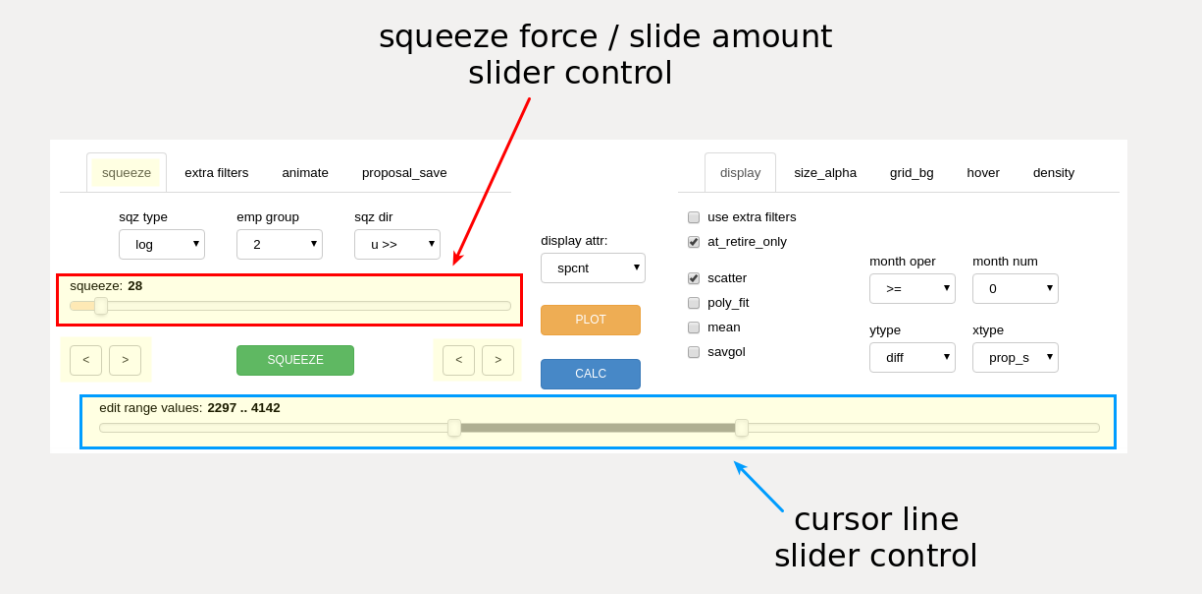


Fig. 49: the editor slider controls and value readouts (squeeze tab)

With either squeeze method, employees from the other groups are reinserted into the remaining slots within the selected integreated list section, in pre-squeeze order. The net effect is that the groups trade positions while maintaining order within each group.

Following the squeeze operation, the horizontal list density chart is updated. This represents the new population density within the edit zone as a result of the squeeze. The squeeze may be repeated differently if the density chart indicates an undesired population shift. The list density chart will be updated each time a squeeze operation is performed. The resultant squeeze chart is always presented in integrated list order perspective.

When satisfied with the squeeze, the squeeze-modified integrated list order is then sent back to the dataset creation routine by clicking the "calculate" button. A new dataset is generated using the modified list order (and any selected conditions from the "proposal_save" panel) and the results appear within the main chart display.

The edit process may be repeated many times, each time using the results of the previous operation. The interactive and iterative nature of the editor tool provides the user with a method to rapidly reduce or eliminate observed equity distortions while "drilling down" to possible list solutions.

using the bokeh chart tools

The editor tool was developed using the bokeh[46] plotting library. Many interactive features are "built-in" with bokeh, allowing the user to explore data much more fully than with static charts. The editor tool incorporates a number of these interactive chart tools to allow zooming, tooltip data display, and other features.

The chart tools are located on the right side of both chart displays.

A blue vertical line to the left of a tool icon means that the tool is active. A click on a tool icon will activate/deactivate it.

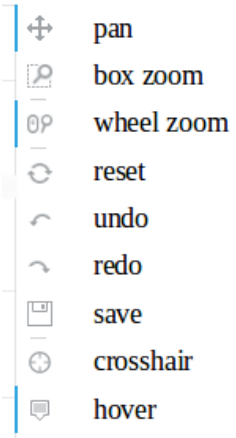


Fig. 50: chart tool icons

The tool definitions (paraphrased) below are from the bokeh user guide[47] :

- The pan tool allows the user to pan the plot by left-dragging a mouse across the plot region.
- The box zoom tool allows the user to define a rectangular region to zoom the plot bounds to, by left-dragging the mouse across the plot region.
- The wheel zoom tool will zoom the plot in and out, centered on the current mouse location.
- The reset tool will restore the plot ranges to their original values.
- The undo tool allows to restore previous state of the plot.
- The redo tool reverses the last action performed by undo tool.
- The save tool allows the user to save a PNG image of the plot.
- The crosshair tool draws a crosshair annotation over the plot, centered on the current mouse position.

[46] https://bokeh.pydata.org/en/latest/
[47] https://bokeh.pydata.org/en/latest/docs/user_guide/tools.html

- The hover tool is an inspector tool.

Controlling main chart auto-scaling

- using the box zoom or wheel zoom will "lock" chart scaling
  - this is helpful during animation, so that the chart is not continuously rescaling to accommodate changing data values (can cause slow or jerky animation)
  - hint: one small mouse wheel move forward and backward using the wheel zoom tool will easily lock scaling
- click the reset chart tool icon to reset auto chart scaling
  - a locked chart scale may mean that nothing is seen after changing a display attribute and plotting until auto-scaling is reset

editor function inputs

The inputs described here are related to the actual editor function arguments, not the inputs made through the various dropdown, check boxes, sliders, and buttons which are displayed above the main chart.

The editor function inputs (arguments) are described within the function docstring, accessed as described in the "notebook interface" section below. The inputs are related to items concerning smoothing values for the various display options, and other sizing and appearance options less commonly altered when running the editor tool.

To change one of the parameters, insert the key value definition within the partial method after the editor function.

Example:

```
handler = FunctionHandler(partial(ef.editor,
                          plot_width=800,  # change plot width
                          plot_height=450,  # change main␣
↪plot height
                          ema_len=30  # adjust exponential␣
↪moving average length
                         ))
```

The default settings for the optional inputs will be changed to the new values. The new values will be used when the editor tool is created by the program.

editor output

The editor produces a dataframe containing the edited list order and a calculated dataset based on that edited list order. A small python dictionary is also generated to allow the tool settings to persist between iterations. All of these files are stored as pickle files within the **dill** folder.

The list order dataframe is named *p_edit.pkl*. It is like other proposal files, containing an index consisting of employee numbers and one column representing list order number.

The dataset produced from the editor tool is stored as *ds_edit.pkl*. It may be fully examined and visualized in the same manner as other datasets. Edited datasets **are not automatically stored** when the "CALC" button is clicked. To store an edited dataset, the user must click the "SAVE EDITED DATASET" button on the "proposal_save" panel.

The small dictionary file is named *editor_dict.pkl*. This file does exist with default values prior to any editing and will be updated with actual editor tool settings as the program is used.

The edited dataset, *ds_edit.pkl*, is only created/updated when the "calculate" button is clicked. The other two files, *p_edit.pkl* and *editor_dict.pkl*, are created/updated every time the "SQUEEZE" button or the "CALC" button is clicked.

5.3.3 summary

The editor is a fast and powerful tool, extremely useful for detecting relative gains or losses or comparing actual outcome values for each employee group under various proposals. It is able to dynamically adjust input list data based on calculated output metrics. Resultant equity distortions may be identified, measured, and corrected interactively. The editor tool offers tremendous flexibility to compensate as necessary throughout the entire range of a combined employee population. This capability provides users with the opportunity to construct integrated seniority lists based on objective data while compensating for unique demographics existing within each employee group list. Results are measurable and transparent. This is a huge distinction and profound improvement from integrated list construction processes which employ uniform list combination formulas, or other non-outcome-based techniques.

The seniority_list program provides the user with insight into the most important aspect of seniority integration: the way a combined list will affect workers for the remainder of their careers. The editor tool feature allows the user to create a solution foundationally focused on fair, equitable, and quantifiable outcome for all workers.

Edited lists may be analyzed, confirmed, and adjusted with reference to the other tools available with seniority_list, so that results are further validated and cross-checked.

Each case study will likely require a blending of analysis techniques to reach an equitable solution.

5.4 building lists

Processing scripts: **list_builder.py**, **join_inactives.py**

seniority_list was designed primarily as an instrument to discover the practical short- and long-term results of various integrated seniority list proposals by calculating, measuring, and comparing multiple attribute metrics, and to correct deficiencies by using the editor tool feature. However, it may also be used as an initial integrated list construction tool by utilizing a weighted-ratio, or "hybrid", combination technique.

To build a hybrid list, import the *list_builder* module. Then run the *build_list* function using the *prepare_master_list* function as the first argument, and lists of attributes and corresponding weightings as the other arguments. The function produces a new list ordering which can then be used as input for the dataset creation routine. The new hybrid list order is stored in the dill folder as "hybrid.pkl". The *build_list* function is able to combine any number of employee groups simultaneously.

Example, with equal weighting applied to longevity and job percentage:

```
import list_builder as lb

lb.build_list(lb.prepare_master_list(),
              ['ldate', 'jobp'],
              [.5, .5])
```

After a integrated list solution has been determined, a final list must be built which reinserts inactive employees who were removed prior to the analysis process. This task is accomplished with the **join_inactives.py** script.

The arguments required by the **join_inactives.py** script are:

- the name of the proposal dataframe containing the final order solution, without "**p_**" prefix or file extension. The argument representing proposal dataframe *p_p3.pkl* would set as "p3".
- the "fill style" or cohort placement technique for the inactive employee insertion.

If the proposed list ordering does not contain the inactive employees, the "fill_style" argument determines where the inactive employees will be placed within the combined list relative to their same employee group active cohorts, just senior to the closest junior cohort or just junior to closest senior cohort.

"ffill" - inactives attached to just *senior* same-group cohort

"bfill" - inactives attached to just *junior* same-group cohort

The result list/dataframe is stored as a pickle file with the name *final.pkl* (within the **dill** folder) and as an Excel file with the name *final.xlsx* (within the case-specific folder located in the **reports** folder).

Example usage from within a notebook cell:

```
%run join_inactives p3 ffill
```

This integrated list result including active and inactive employees in concert with with any associated conditions represents the ultimate end goal of the seniority_list program.

5.5 notebook interface

This section will provide a basic primer to the Jupyter notebook, run through a short demo of the program using the notebook, and provide guidance in the event that the program needs to be reinstalled if it becomes inoperable for some reason.

From the Project Jupyter homepage[48]:

> "The Jupyter Notebook is a web application that allows you to
> create and share documents that contain live code, equations,
> visualizations and explanatory text."

From the predecessor IPython notebook original webpage (the Jupyter notebook was originally the IPython notebook):

> "It is an interactive computational environment,
> in which you can combine code execution, rich text,
> mathematics, plots and rich media."

The name "Jupyter" is a combination of three programming language names, Julia, Python, and R. It is actually capable of running code in over forty different languages, not just the three making up its name.

[48] http://jupyter.org//

seniority_list was designed to utilize the Jupyter notebook for running scripts, creating datasets, and exploring and visualizing data.

The Chrome web browser is recommended for use with seniority_list for best performance.

5.5.1 notebook basics

starting the notebook

To run a notebook, open a terminal window or powershell and then type or copy and paste:

```
jupyter notebook
```

A browser window will open containing a file listing. Navigate to one of the Jupyter notebook files (ending with a *.ipynb* file extension, above) and click on the file name. The notebook will open.

imports

Python files containing functions (modules) may be imported or loaded for use within a notebook by using the "import" statement. Once a module has been imported, the program "knows" about all the functions from that module. A module is imported and assigned an alias, or shortened name as follows:

```
import matplotlib_charting as mp
```

Now, all of the plotting functions may be run from the notebook, and tab completion is active. By typing the import name ("mp" in this case) followed by a dot, the user may then hit the TAB key for a list of all of the functions available for use from the imported module. The function may still be typed in manually or selected from the list presented with the tab completion feature. Select from the list using the up and down arrows and the ENTER key.

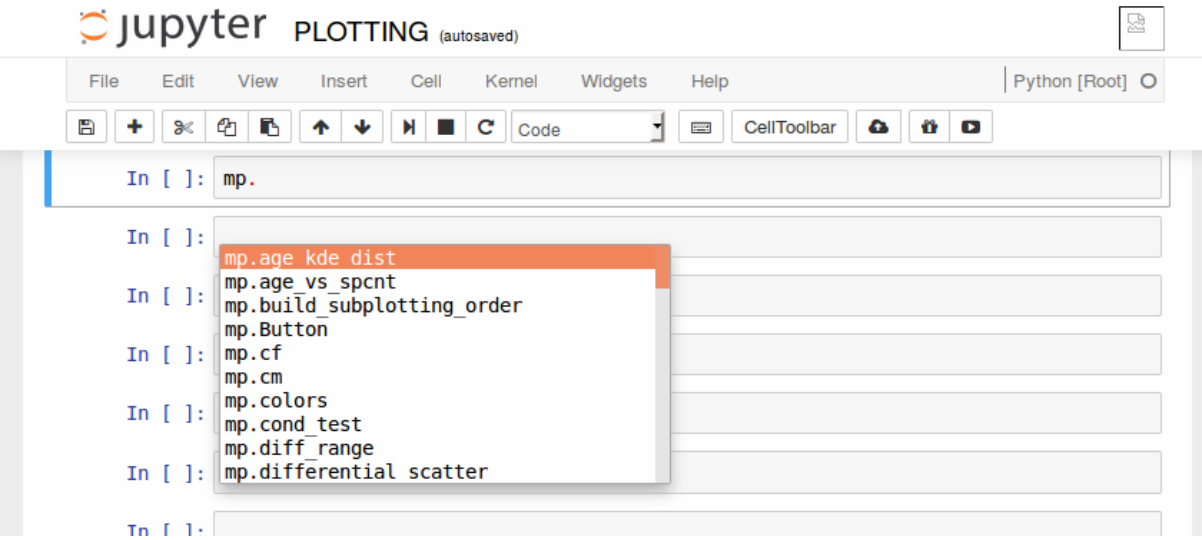


Fig. 51: type mp. then TAB to reveal a list of functions available from the *matplotlib_charting* module

running cells

The Jupyter notebook is a web interface which consists of a collection of 'cells' normally containing program code. The code may be executed with the results appearing beneath the cell, if there is actual output from the code. To execute a cell, it must be 'active' (click on it). Then hit "Shift + Enter" on the keyboard to run it. There is also a "run" button in the notebook header section that will work as well.

Sometimes the type of cell changes to something other than "code", which will prevent the execution of the cell. The type of cell may be changed back to "code" with a dropdown box in the notebook header section.

There are numerous free tutorials and other educational materials on the web pertaining to the use of the Jupyter notebook (formally known as the Ipython notebook). Here[49] is one video which may be of interest to new users.

running scripts within cells

Several of the program files are Python scripts used to perform major program tasks. The work performed by the scripts is described in the "program flow" section above. Below is a summary table containing required and optional arguments for each program script.

[49] https://youtu.be/HW29067qVWk

script	required arguments	optional arguments
build_program_files	**case name**	none
make_skeleton	none	none
standalone	none	**prex**
compute_measures	**proposal name**	**conditions**
join_inactives	**order, fill**	none

- **case name** - the case study name (the name of the folder containing the input files)
- **prex** - if a pre-existing job condition exists, use "prex" to direct the program to apply a special job assignment condition as defined in the settings.xlsx input file
- **proposal name** - name of a proposed list ordering, such as "p3", originally set from worksheet names in the proposals.xlsx input file. Names of the proposals may also be read from the *proposal_names.pkl* file within the **dill** folder.
- **conditions** - string name representing various conditional job assignment routines. Choices are:

```
['prex', 'ratio', 'count']
```

  A "ratio" or "count" condition may be combined with the "prex" condition.

  The "excel input files" section of the documentation contains descriptions of the condition options. The parameters for the conditions are set through the *settings.xlsx* input file.

- **order** - a dataframe formatted with an index of empkeys and a single column with an order number, named either "idx" in the case of an original proposal or "new_order" when using the output of the editor tool.
- **fill** - the fill style to use which determines how the inactive employees are reinserted into the integrated list. See the "building lists" section above.

A script may be run from a Jupyter notebook cell by inserting the special command, "%run", before a script name, and then hitting "Shift + Enter" on the keyboard to run the cell and execute the script. The following code will execute the **compute_measures.py** script for proposal "p2" with a ratio count-capped condition applied. The first line, "%%time", will provide the user with a print out of the amount of time it took to complete the script task:

```
%%time
%run compute_measures p2 count
```

function docstrings

Click on function name within a cell, then Shift+TAB keyboard combination to reveal the function docstring. Another way to do the same thing is to type the name of a function followed by a question mark and then run the cell:

```
mp.quantile_years_in_position?
```

The information displayed may be expanded by clicking on the ^ or + symbols in the upper righthand corner of the docstring window.

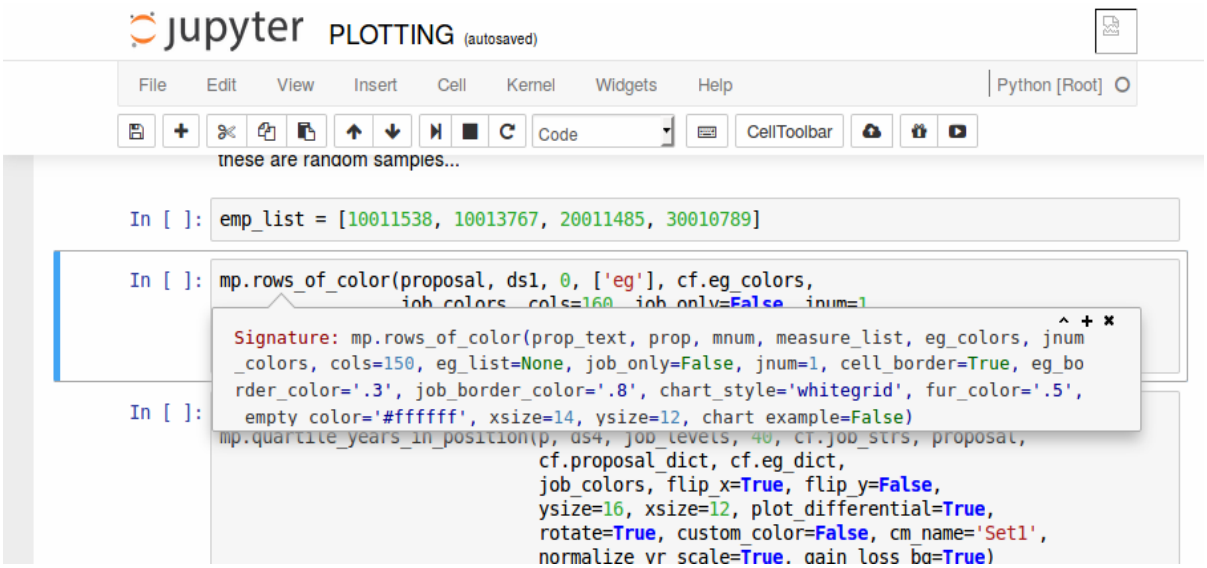


Fig. 52: click on function name, then Shift+TAB to reveal docstring, or type the name of the function with a question mark, then run cell

The docstrings contain descriptions, instruction, and input definitions for the many program functions of seniority_list. The docstring may in fact be the best source of information concerning the usage of a function.

The "program demonstration" section below has more information about function docstrings.

functions and variables

The function variables visible within the notebook cells are contained within parentheses following the function name. The order of the variables is important and must be maintained for the function to operate. Other variables may not be displayed within the notebook code cell and are defined with default values within the function definition itself. It is likely that they may changed to another value by the user.

To view the full function code, a text editor may be used to open the appropriate module and search for the function name. Be careful not to change anything within the module to ensure proper program function. When using the notebook, function code may also be viewed by using two question marks after a function name, as follows:

```
mp.quantile_groupby??
```

A window containing the function code will open in the lower section of the notebook.

If a change is made to any seniority_list program code, please submit a pull request or send an email with the change as required by the licensing terms of the program. Please see the "contact" section for the developer email address.

exiting the notebook

To discontinue use of the notebook, save all notebooks and close the notebook browser windows. Then use the keyboard combination CTRL+C within the terminal to shut down the notebook server.

5.5.2 sample notebooks

Four sample Jupyter notebooks are included with seniority_list.

- RUN_SCRIPTS.ipynb
- STATIC_PLOTTING.ipynb
- INTERACTIVE_PLOTTING.ipynb
- REPORTS.ipynb
- EDITOR_TOOL.ipynb

As mentioned on the installation page, the Jupyter notebook is included and installed with the Anaconda scientific platform.

The RUN_SCRIPTS notebook creates many files and the datasets from the sample files included with seniority_list, and will provide a feel for the capability and speed of the program.

The STATIC_PLOTTING notebook runs many of the built-in plotting functions using the datasets produced from the Run_Scripts notebook. This notebook provides a platform for practice exploring, plotting, and analyzing datasets. The "STATIC" part of the notebook title simply means that the chart output is not interactive.

The INTERACTIVE_PLOTTING notebook was added to the program in January of 2018. It offers chart output which can be modified within the notebook in real time using sliders and dropdown selections.

The REPORTS notebook demonstrates the generation of summary statistical reports for all program datasets, with output in spreadsheet and chart image formats. This feature is described in the "quick report" section of the documentation.

The EDITOR_TOOL notebook will load the interactive editor tool. Please review the "editing" section above for the powerful visualization and editing features available with this function. This function will only run within the Jupyter notebook interface.

> Note: The **RUN_SCRIPTS.ipynb** notebook must initially be run prior to the other sample notebooks included with the program. The other notebooks require the dataset files which are created by the **RUN_SCRIPTS.ipynb** notebook.

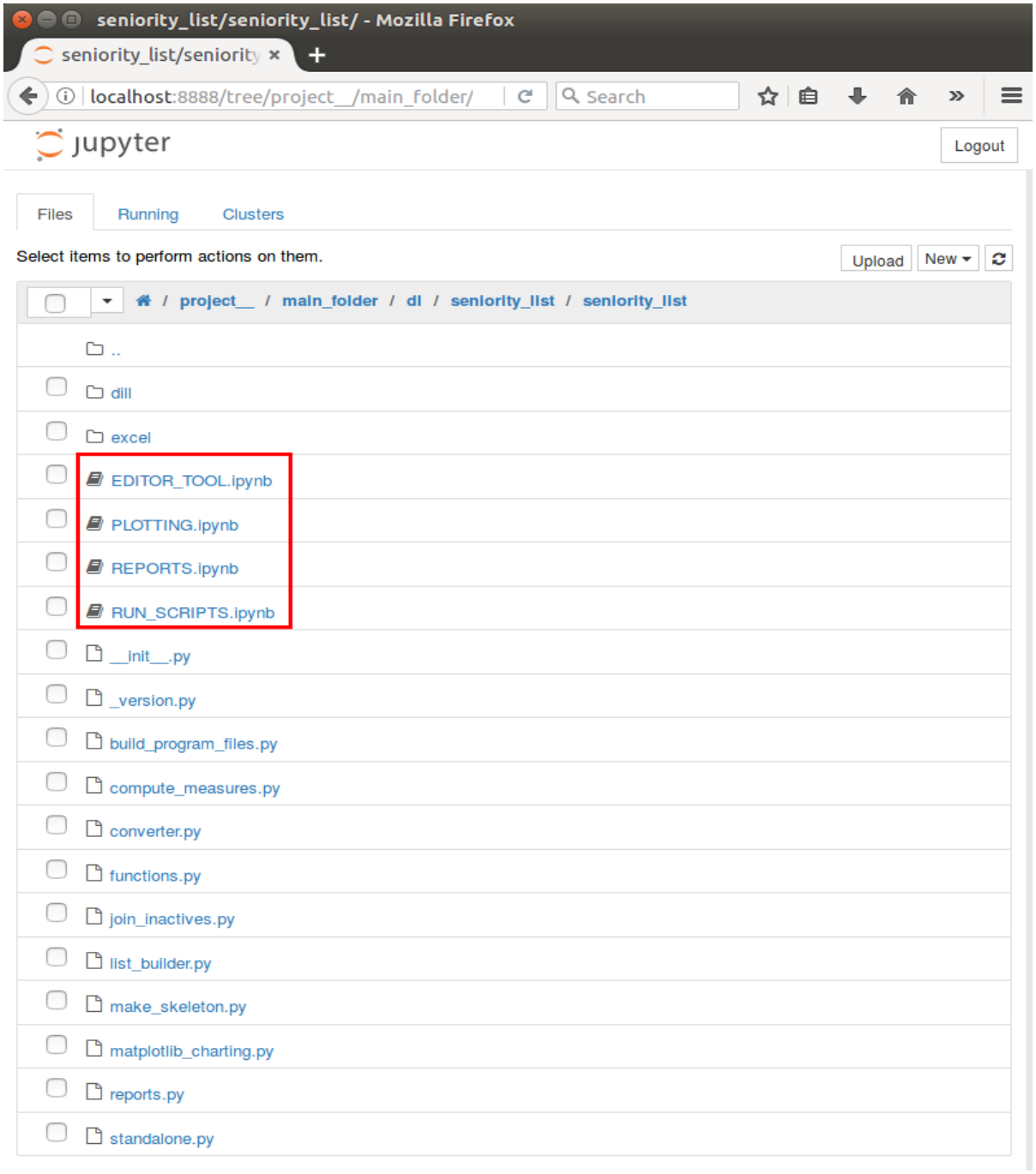


Fig. 53: click on a file with the *.ipynb* file extension

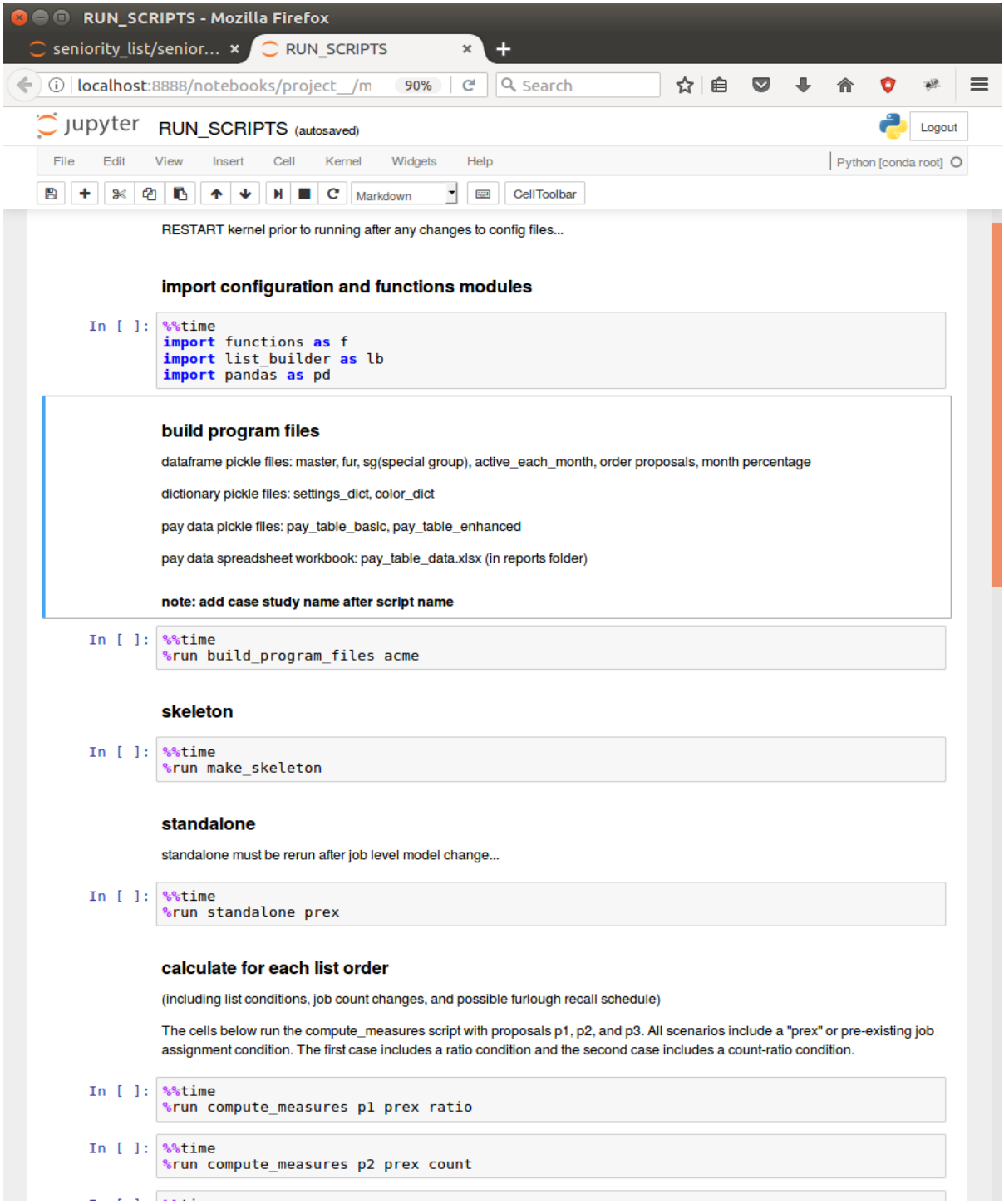


Fig. 54: the Jupyter notebook in the browser, with code cells waiting to be run

To run the notebook, click on “Cell” and then from the dropdown menu, select “Run All”. If all goes well, the notebook will load the required data, run each section of code (“cells”), and display results below each cell.

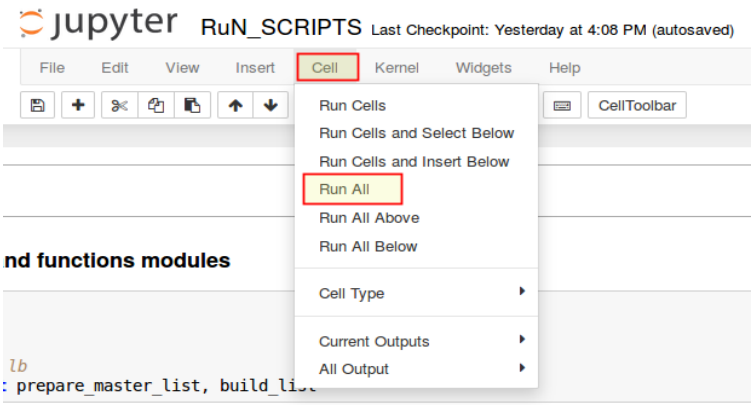
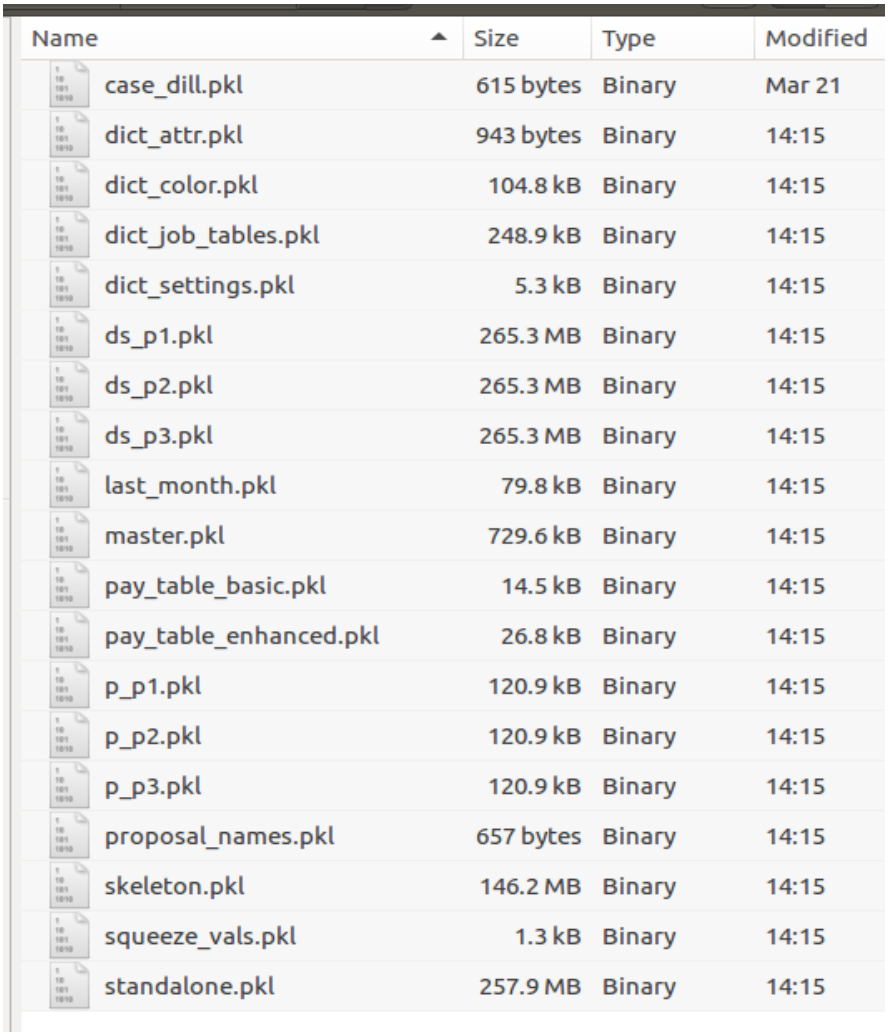


Fig. 55: running the sample notebooks

To view the files which were created during the Run_Scripts notebook execution, use a file explorer to view the contents of the program **dill** folder.



Name	Size	Type	Modified
case_dill.pkl	615 bytes	Binary	Mar 21
dict_attr.pkl	943 bytes	Binary	14:15
dict_color.pkl	104.8 kB	Binary	14:15
dict_job_tables.pkl	248.9 kB	Binary	14:15
dict_settings.pkl	5.3 kB	Binary	14:15
ds_p1.pkl	265.3 MB	Binary	14:15
ds_p2.pkl	265.3 MB	Binary	14:15
ds_p3.pkl	265.3 MB	Binary	14:15
last_month.pkl	79.8 kB	Binary	14:15
master.pkl	729.6 kB	Binary	14:15
pay_table_basic.pkl	14.5 kB	Binary	14:15
pay_table_enhanced.pkl	26.8 kB	Binary	14:15
p_p1.pkl	120.9 kB	Binary	14:15
p_p2.pkl	120.9 kB	Binary	14:15
p_p3.pkl	120.9 kB	Binary	14:15
proposal_names.pkl	657 bytes	Binary	14:15
skeleton.pkl	146.2 MB	Binary	14:15
squeeze_vals.pkl	1.3 kB	Binary	14:15
standalone.pkl	257.9 MB	Binary	14:15

Fig. 56: files created by Run_Scripts.ipynb

Note that the dataset files (starting with "ds") are large at 260mb+. and are generated from a sample list of approximately 7500 employees. The files depicted above were generated utilizing the sample case study, "Sample3", which includes approximately 7500 employees from 3 separate employee groups and 3 different integrated list proposals.

There is one other file created, *pay_table_data.xlsx*, an Excel file stored in the **reports** folder (not shown here).

The screenshot below is an example of matplotlib charts displayed within the sample STATIC_PLOTTING notebook. Notice that just above each chart area there is a cell which contains the plotting function which created the charts. The inputs to the functions may be modified directly within the notebook and re-executed, creating new chart results in seconds.

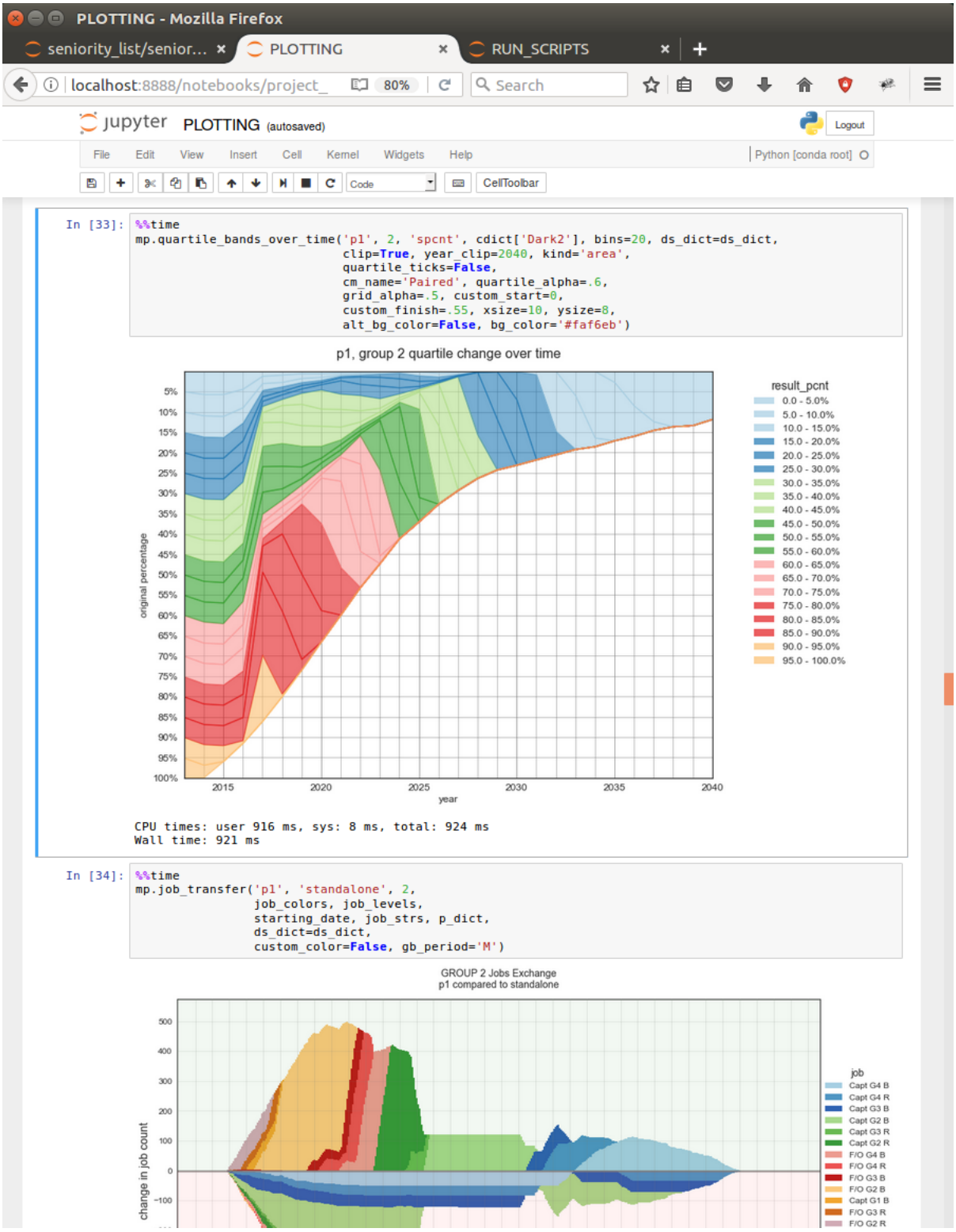


Fig. 57: changing function inputs produces new results in real time. . .

5.6 program demonstration

This demo will walk through the steps involved with setting up and analyzing a new case study. It is assumed that the program has been downloaded in accordance with the "installation" section of this documentation.

Note: To run the demo using the included sample dataset, "Sample3", no modification of files is necessary - simply run the included notebooks in the order within the description below.

The screenshots below were taken while using a linux operating system. The information may be presented differently with other operating systems, but the actions remain the same.

5.6.1 new case study

set up inputs

1. Navigate to the **seniority_list** folder within the main **seniority_list** folder with a file browser

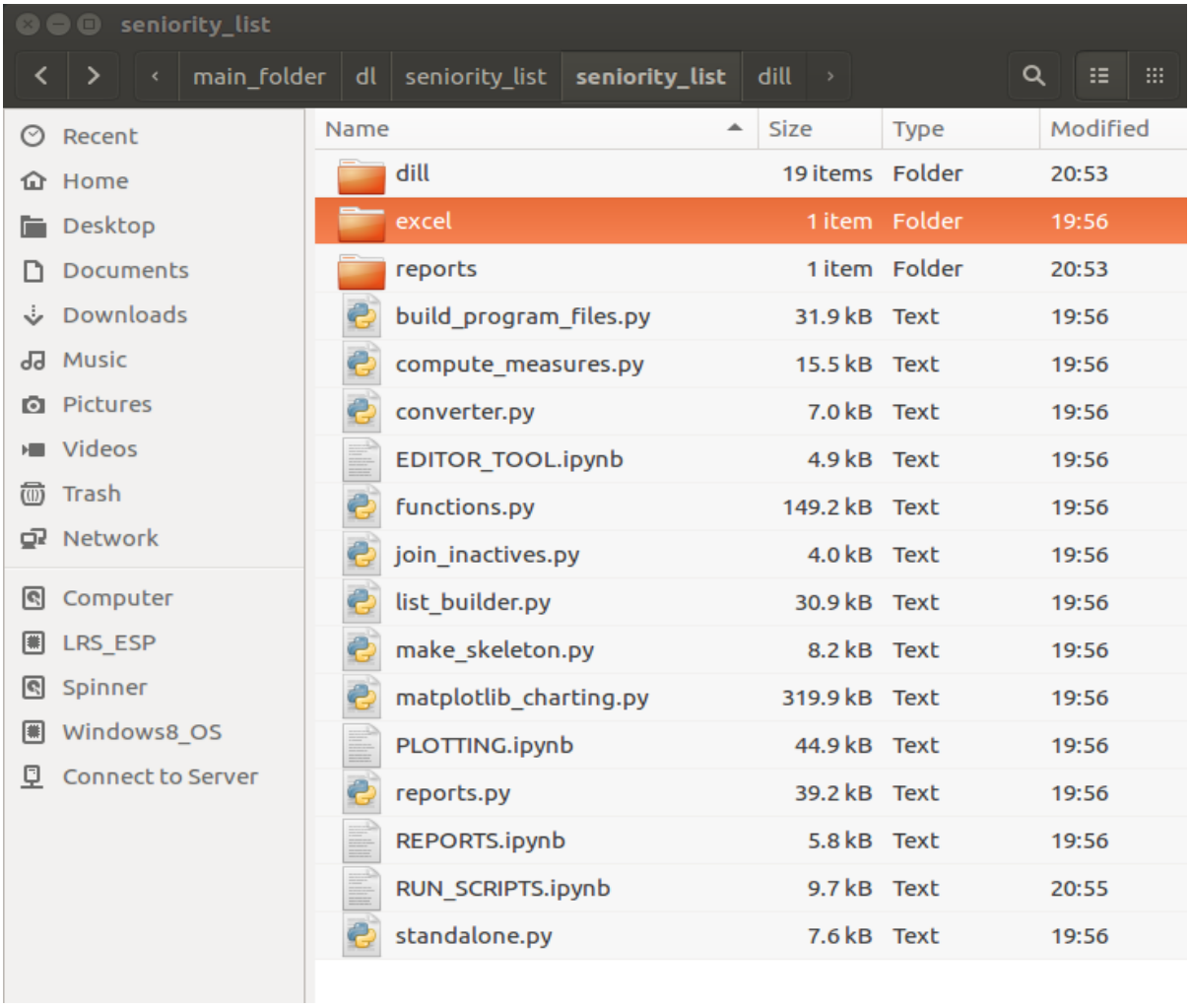


Fig. 58: the seniority_list folder within the main seniority_list folder. . .

2. Copy a case study folder within the **excel** folder (**sample3** is fine)

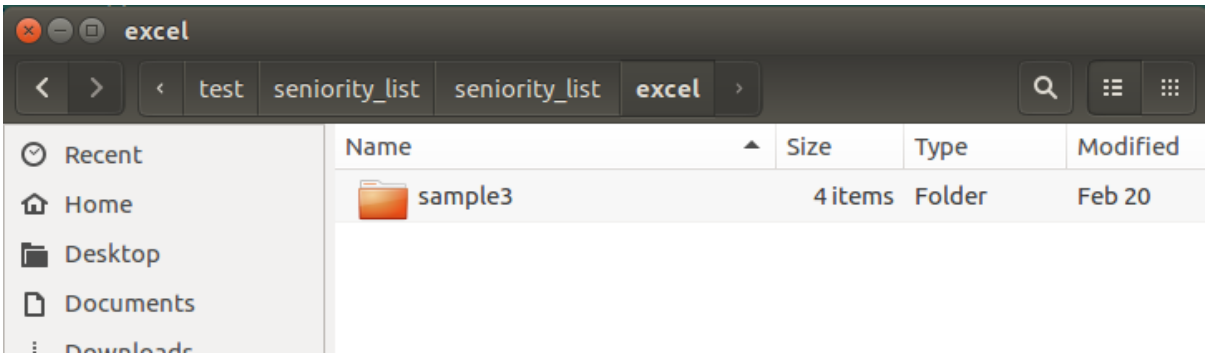


Fig. 59: inside the excel folder, home of the case study input folders. . .

3. Paste the folder back into the **excel** folder and rename it t match the desired case study name - this example will use “acme”

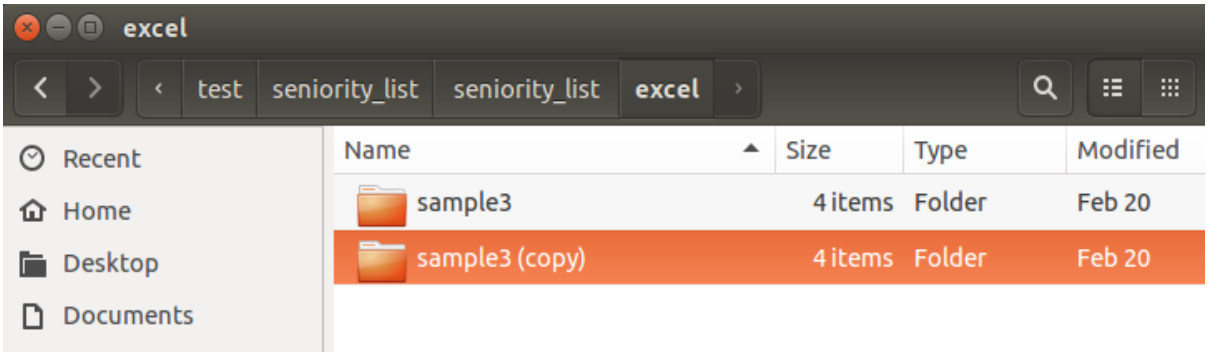


Fig. 60: the copied folder. . .

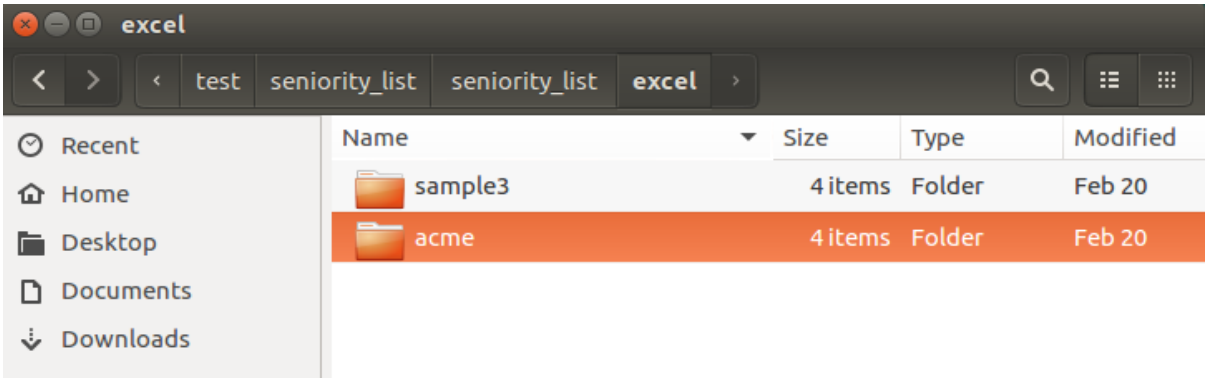


Fig. 61: the copied folder renamed for the case study. . .

4. Modify the contents of the Excel workbooks within the **acme** folder as appropriate, using the “excel input files” section of the documentation as a guide

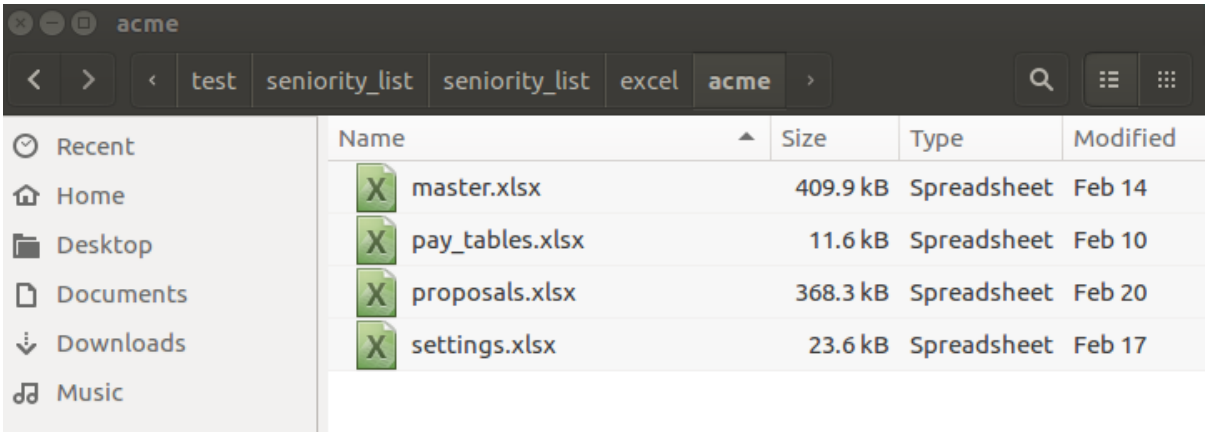


Fig. 62: the four .xlsx files ready for case study customization

set up jupyter notebook

5. Open a Jupyter notebook and navigate to the **seniority_list** folder containing the 5 sample notebooks (.ipynb files).

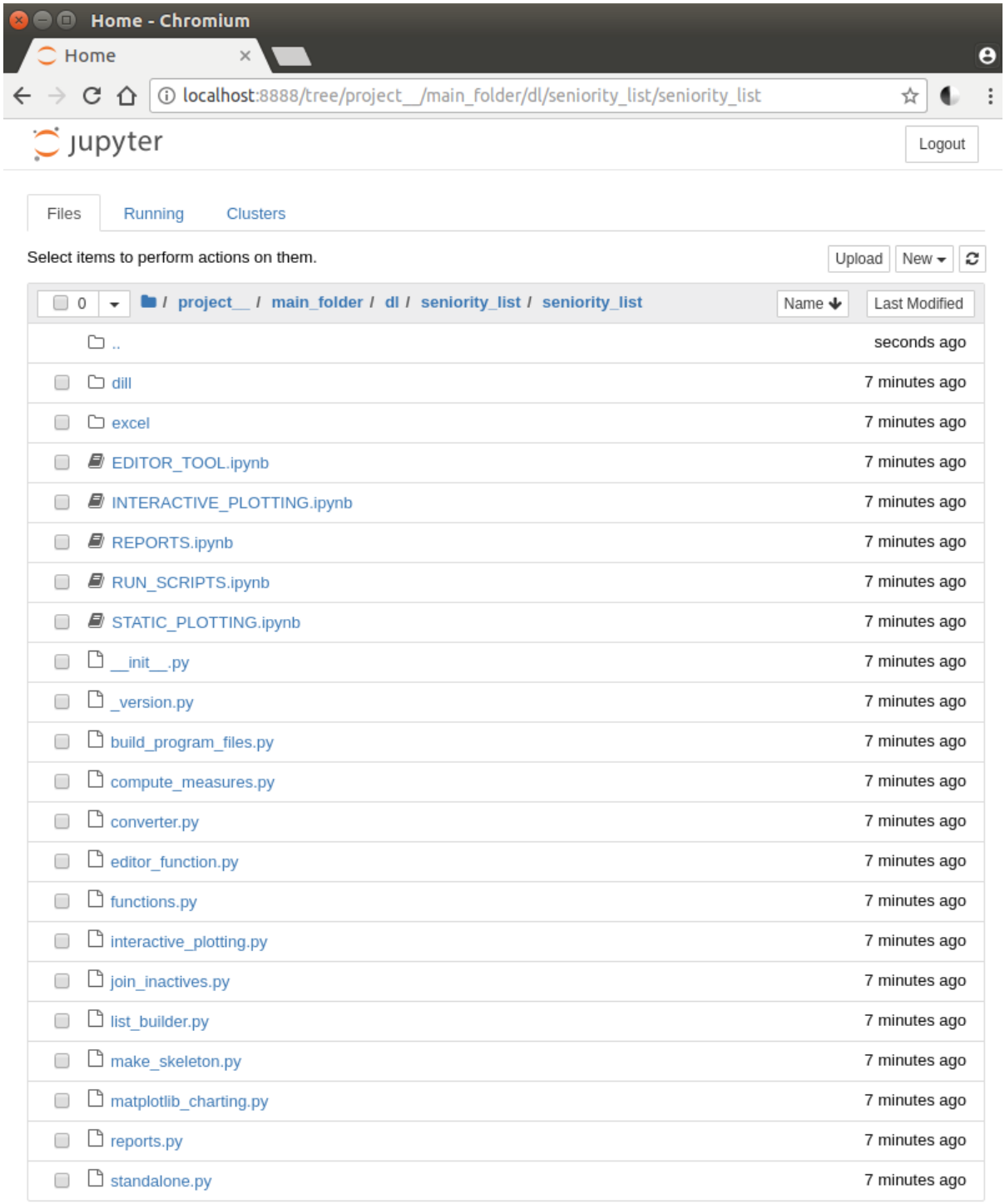


Fig. 63: the seniority_list folder in the jupyter notebook

6. Open the notebooks by clicking on the titles - a new browser tab will open for each notebook

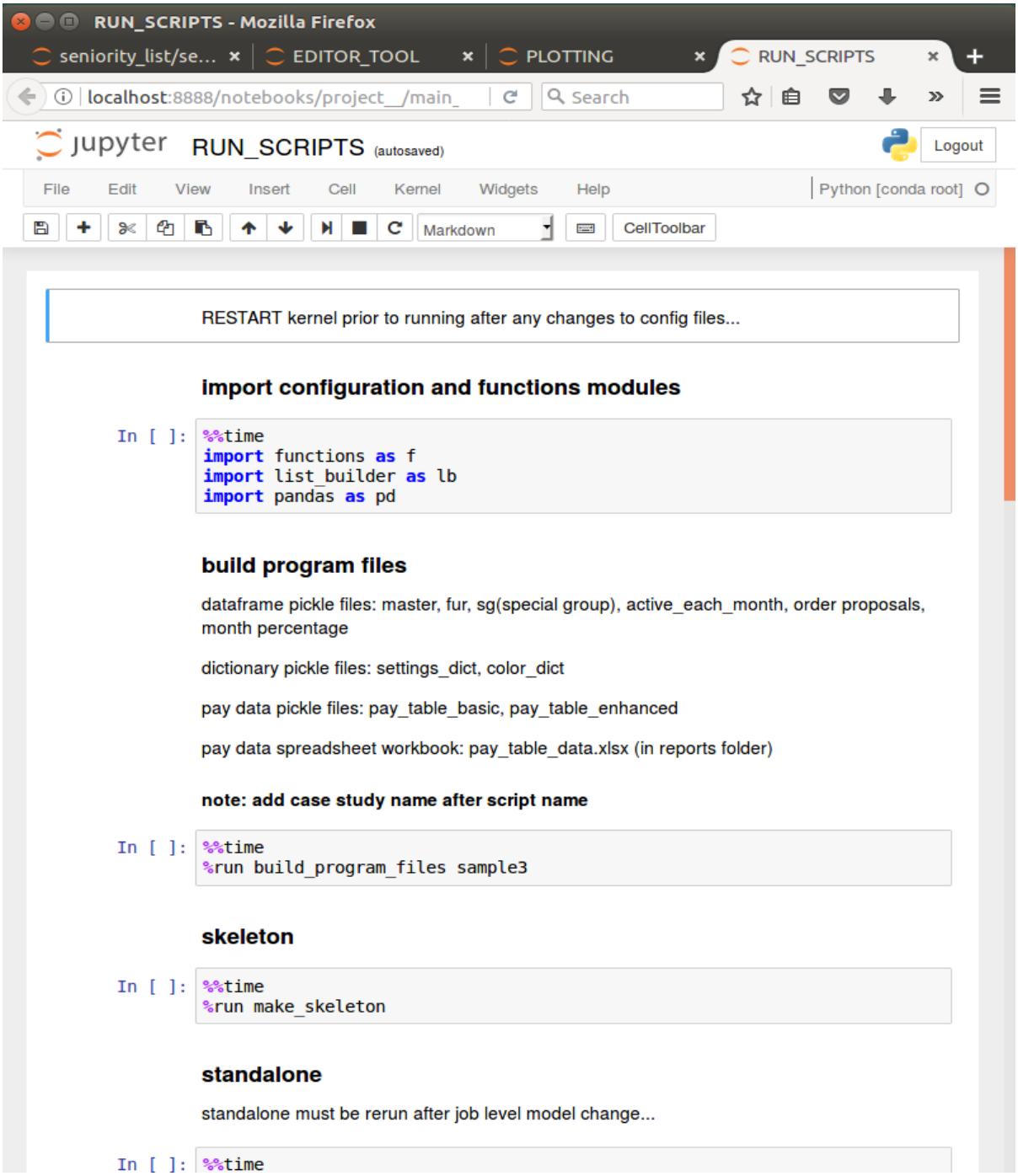


Fig. 64: a browser tab for each notebook, with the RUN_SCRIPTS notebook displayed

7. Modify the script arguments in the notebook cells to match the new arguments which pertain to the current case study.

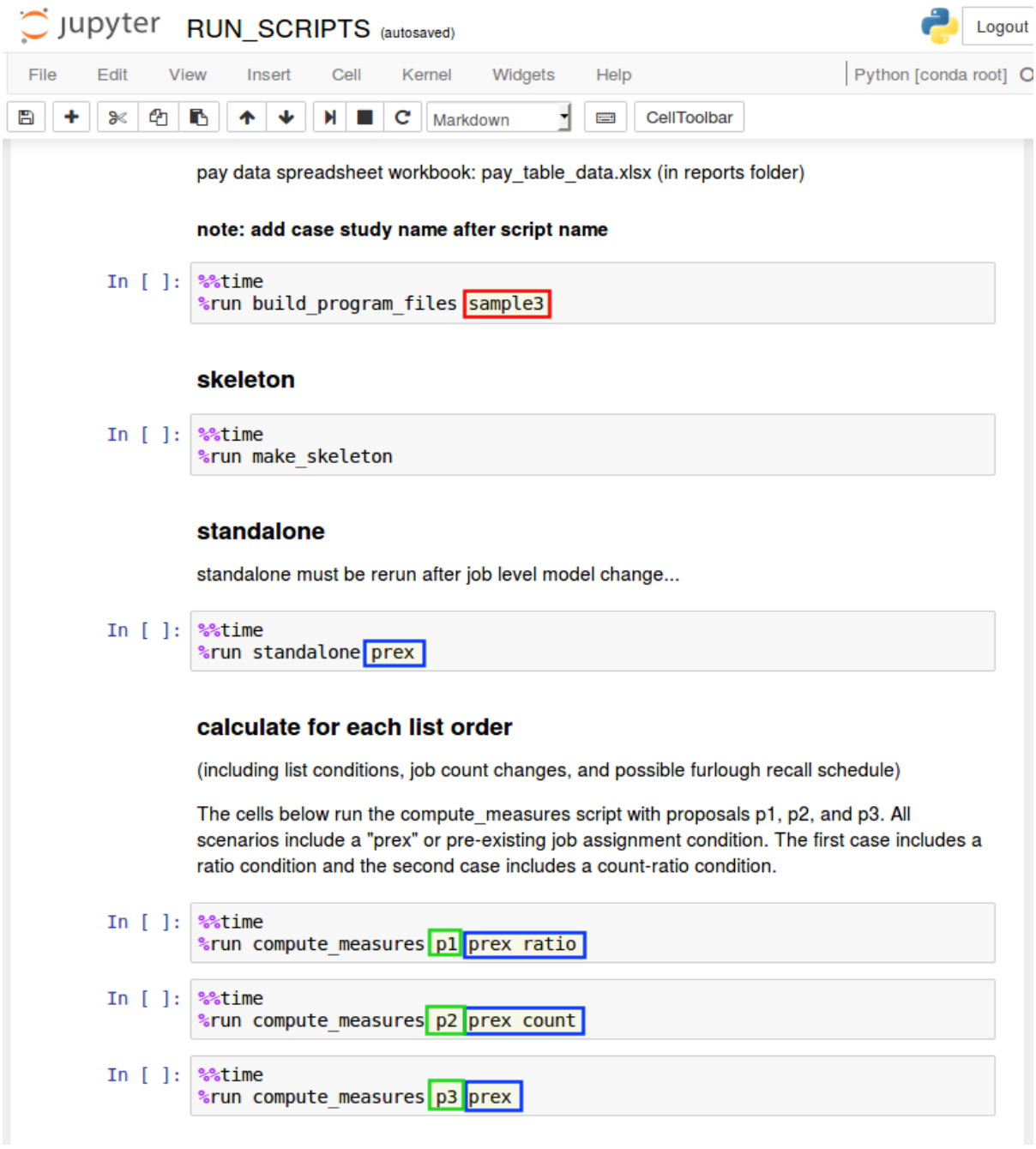


Fig. 65: case study marked red, conditions marked blue, proposal names marked green

- **RUN_SCRIPTS** notebook
  - case study name for **build_program_files** script
  - condition argument(s) for **standalone** and **compute_measures** scripts
  - proposal names for the **compute_measures** script cells (one for each proposal)

- **STATIC_PLOTTING** notebook
  - proposal string inputs to match proposal name(s)

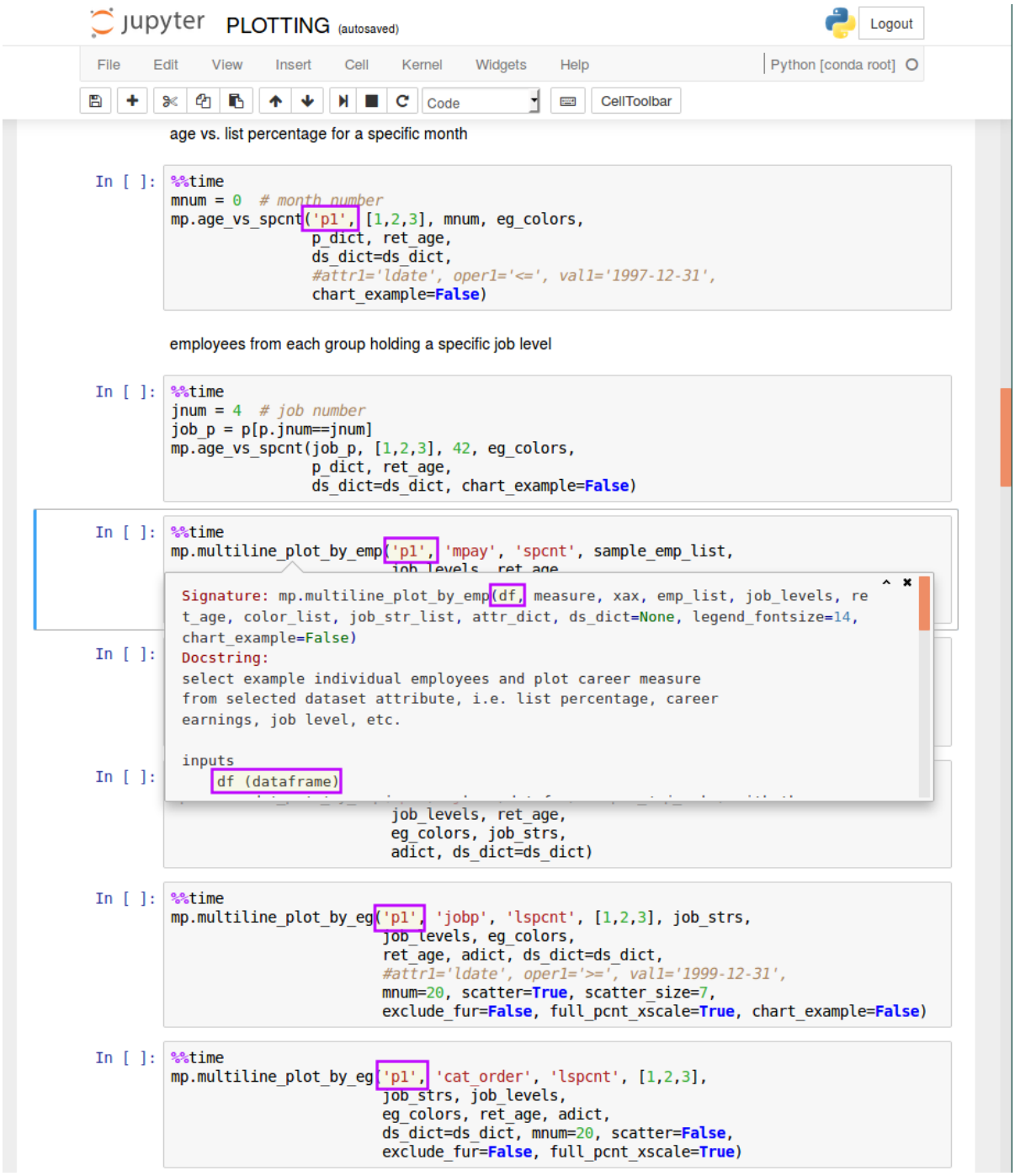


Fig. 66: plotting functions inputs matching a proposal name

The functions have been coded so that they can accept the name of an integrated list proposal as an input to represent a dataset calculated from that proposal. Con-

sequently, the inputs must match the proposal names which are part of the current case study. The source of these names are the worksheet names within the proposals.xlsx input file.

create program files and datasets

8. Run the **RUN_SCRIPTS** notebook

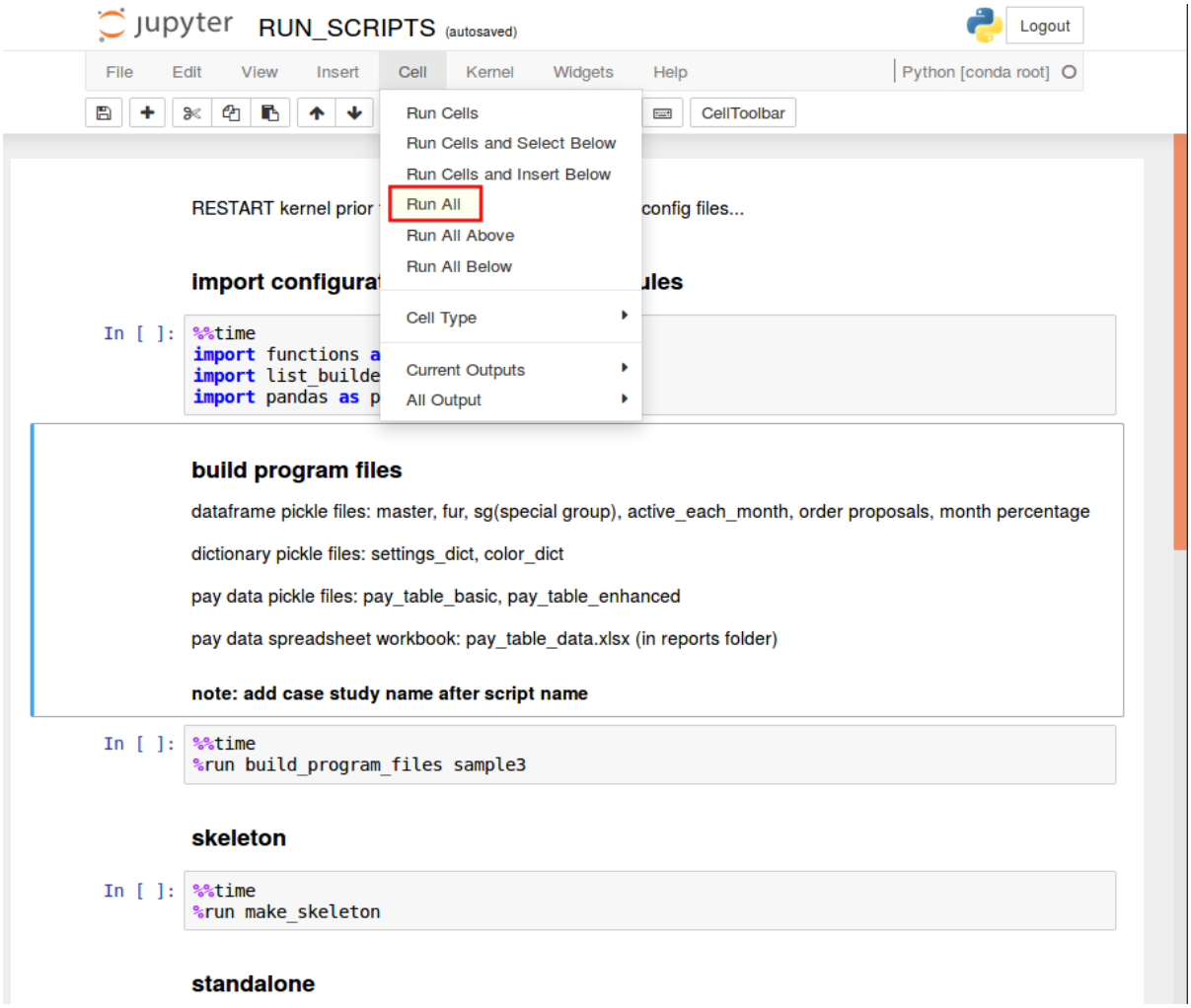


Fig. 67: select the "Cell" button from the menu bar, then click "Run All"

- the contents of the **dill** folder will be cleared and the folder will then be repopulated with program files and datasets pertaining to the "acme" case study.
- open the **dill** folder in a file browser to see the files populate the folder in real time. . .

analyze datasets

9. Run the **REPORTS** notebook

   A statistical summary of proposed integrated list outcomes will be generated, in the form of chart images and spreadsheets.

   - **ret_charts** and **annual_charts** folders will be created within the **reports** folder. Each of these folders will contain several other folders with many basic chart images.
   - *ret_stats.xlsx* and *annual_stats.xlsx* spreadsheet files will be created within the **reports** folder.

10. Run the **STATIC_PLOTTING** notebook (with the correct proposal name argument(s)

    - this will run the sample plotting functions with parameters set as they existed when the program was downloaded.

11. Modify plotting function arguments as necessary for analysis (see docstrings)

    - there are many options associated with most of the plotting functions
    - view the docstrings as described in the section above to learn about the analysis possibilities with each function

    The images below demonstrate a sample of possibilities with one plotting function, *quantile_groupby*.

    The function groups an initial list of employees into equally sized segments and tracks each segment over time according to a selected metric. The result for each segment is displayed as a line on the chart. This technique provides a quick view into how employees at various levels within a seniority list fare under standalone and integrated scenarios.

    The program generates a global job ranking metric, termed "cat_order" which stands for category order. Standalone dataset "cat_order" results are normalized with integrated results, allowing direct comparisons to be made between them.

    The job ranking "cat_order" value is closely related to the "jobp" metric, which reflects percentage of position within a job level, with the advantage of true scaling for chart presentation. In other words, job levels with many jobs occupy a larger part of the chart than job levels with few jobs.

    The first image below is an example of the *quantile_groupby* function output displaying the results for the "cat_order" measure for one employee group in a standalone scenario, as grouped into 40 quantiles.

```
%%time
# cat_order progression for the median of each 2.5% segment (40 quartiles)
# of employee group 2 (standalone dataset)
mp.quartile_groupby('standalone', [2], 'cat_order', 40,
                    eg_colors, job_colors, sdict, adict,
                    ds_dict=ds_dict, show_job_bands=False,
                    custom_color=False,
                    through_date='2037-12-31')
```

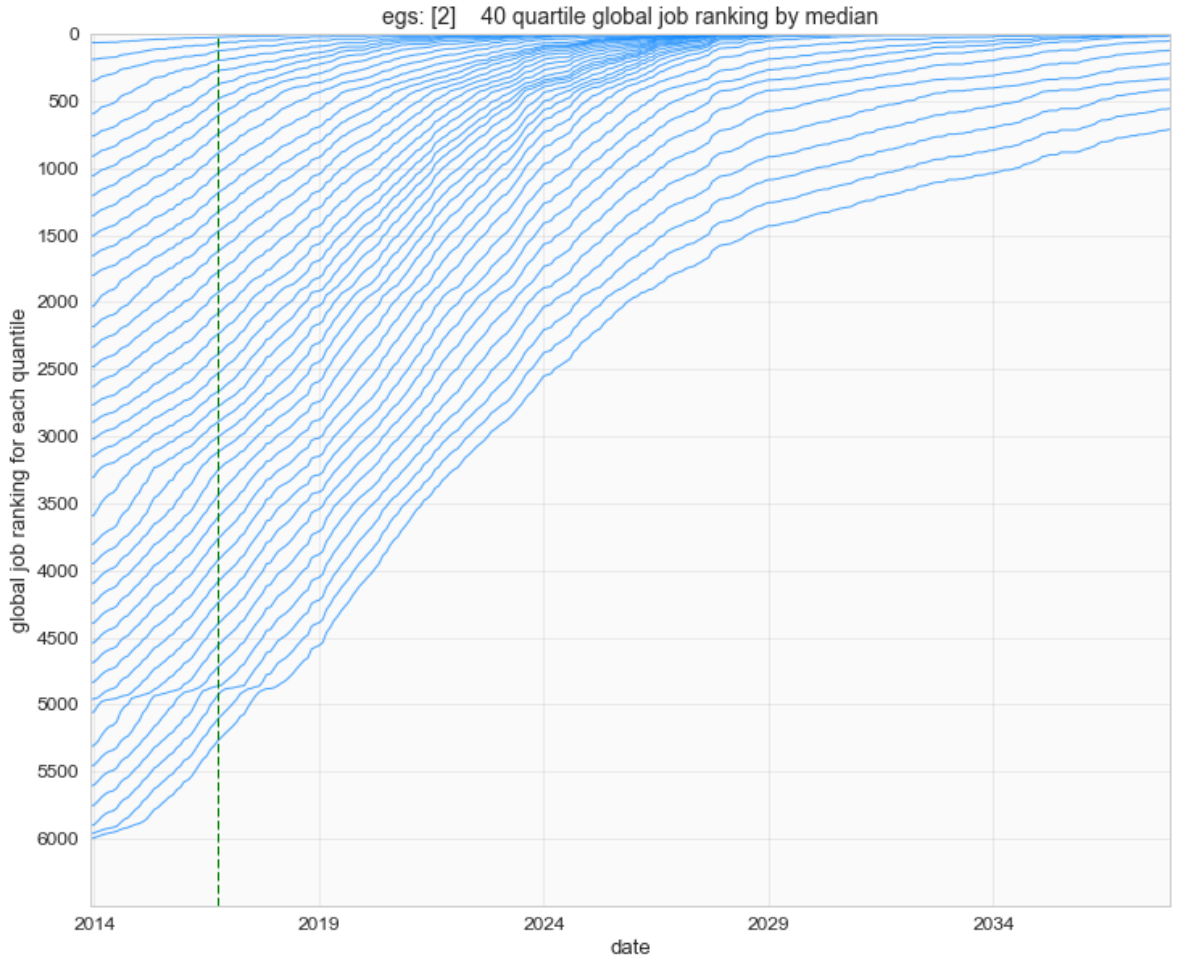


Fig. 68: standalone dataset, employee group 2, category order (job ranking), 40 quantiles

Here is the same group, with the first argument changed to an integrated proposal, "p1". The computed scenario included a delayed implementation date, indicated with the dashed vertical date line and the sudden change in the progression of the lines.

```
%%time
# same as above, but as affected by integration proposal p1,
# with a delayed implementation date
mp.quartile_groupby('p1', [2], 'cat_order', 40,
                    eg_colors, job_colors, sdict, adict,
                    ds_dict=ds_dict,
                    show_job_bands=False, custom_color=False,
                    through_date='2037-12-31')
```

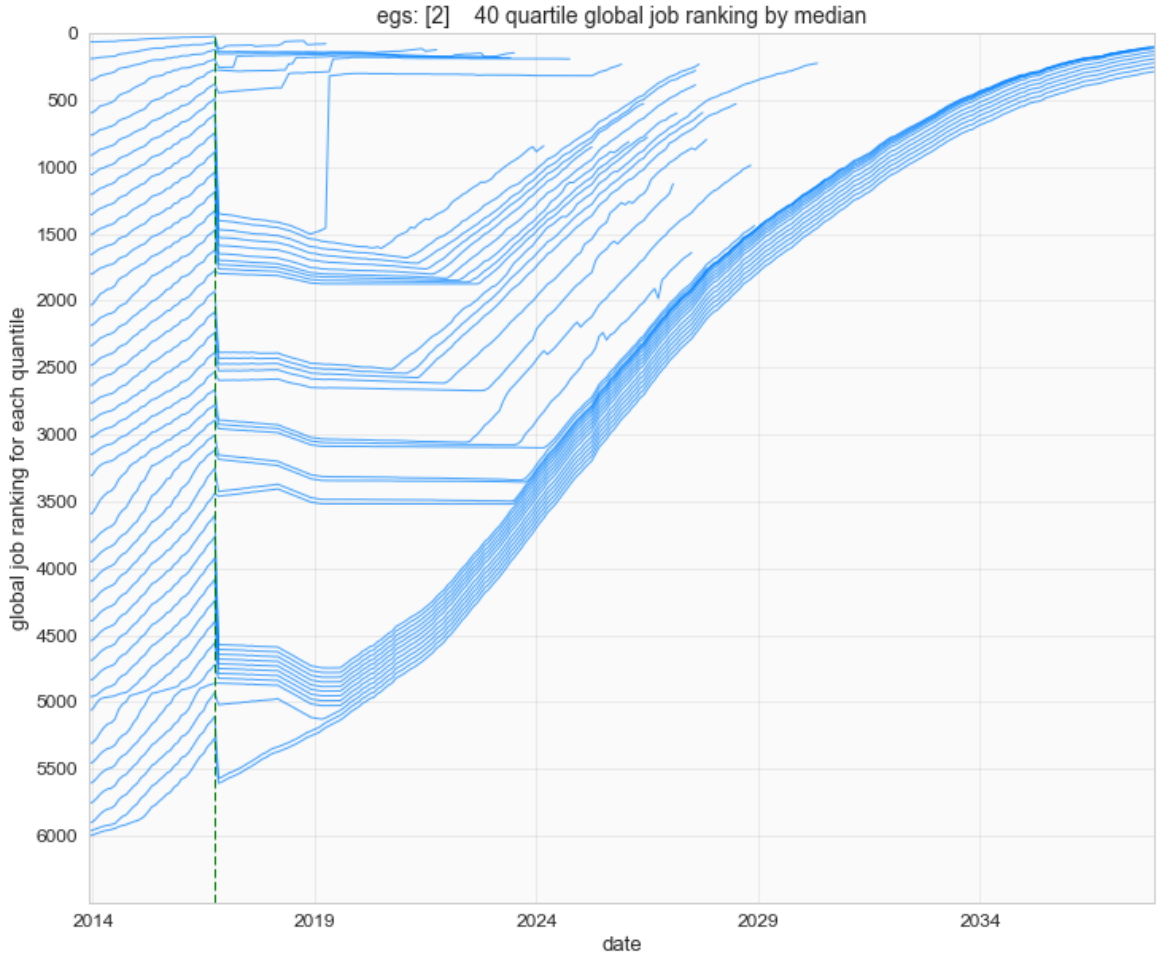


Fig. 69: p1 dataset, employee group 2, category order (job ranking), 40 quantiles

The "show_job_bands" option changed to "True". A background job level hierarchy is now shown, and the sudden changes in the progression of the chart lines begins to gain context.

```
%%time
# same as above, but with the addition of job bands
mp.quartile_groupby('p1', [2], 'cat_order', 40,
                    eg_colors, job_colors, sdict, adict,
                    ds_dict=ds_dict,
                    show_job_bands=True, custom_color=False,
                    through_date='2037-12-31')
```

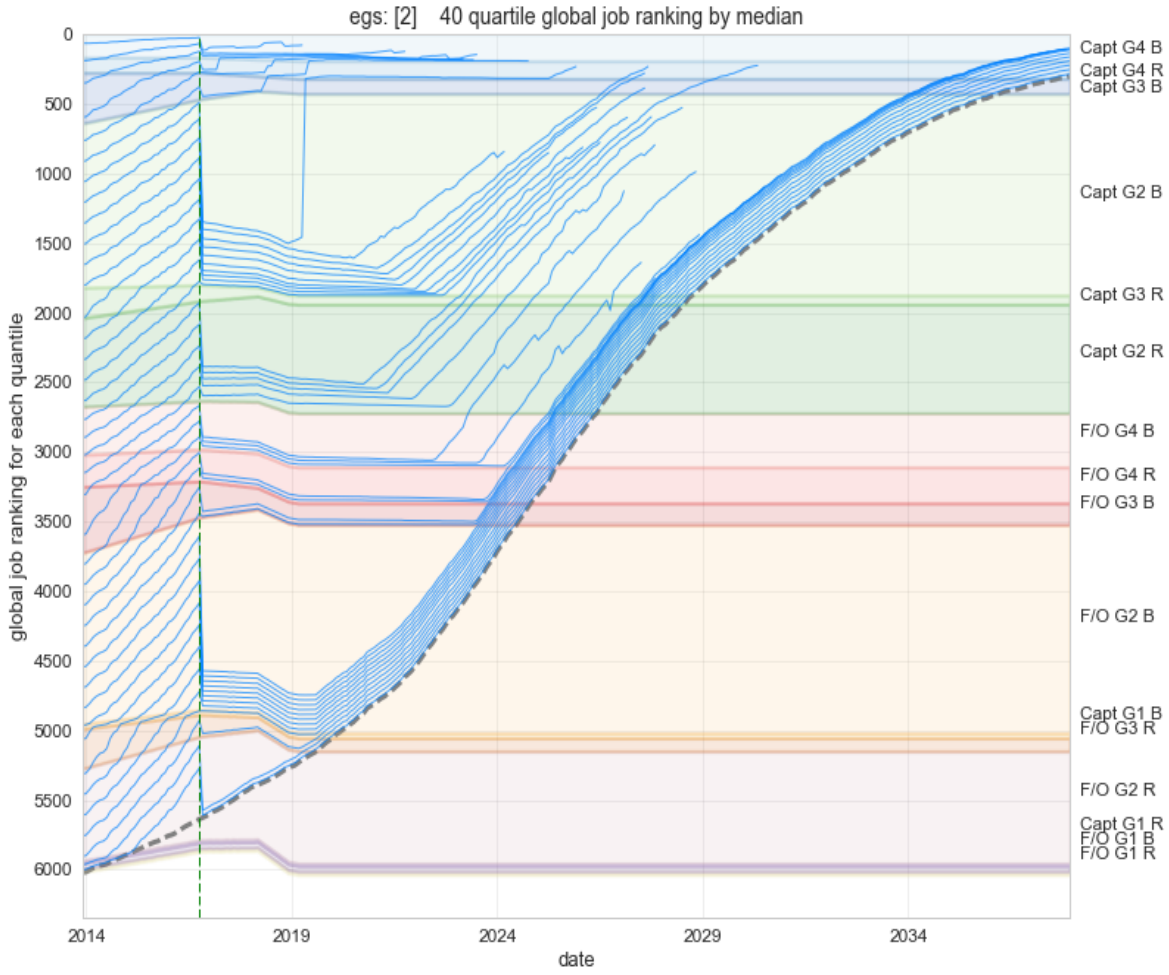


Fig. 70: p1 dataset, employee group 2, category order (job ranking), 40 quantiles, with job bands

Here, the number of quantiles input was changed to 250 to produce a much denser presentation and a custom color spectrum was introduced, allowing a much clearer visual presentation of the outcome.

```
%%time
# cat_order progression for the median of each .4% segment (250 quartiles)
# of employee group 2 (p1 dataset)
# custom color example
mp.quartile_groupby('p1', [2], 'cat_order', 250,
                    eg_colors, job_colors, sdict, adict,
                    groupby_method='median',
                    ds_dict=ds_dict, show_job_bands=True,
                    line_width=1, custom_color=True, chart_style='whitegrid',
                    show_grid=True, cm_name='Dark2',
                    through_date='2037-12-31')
```

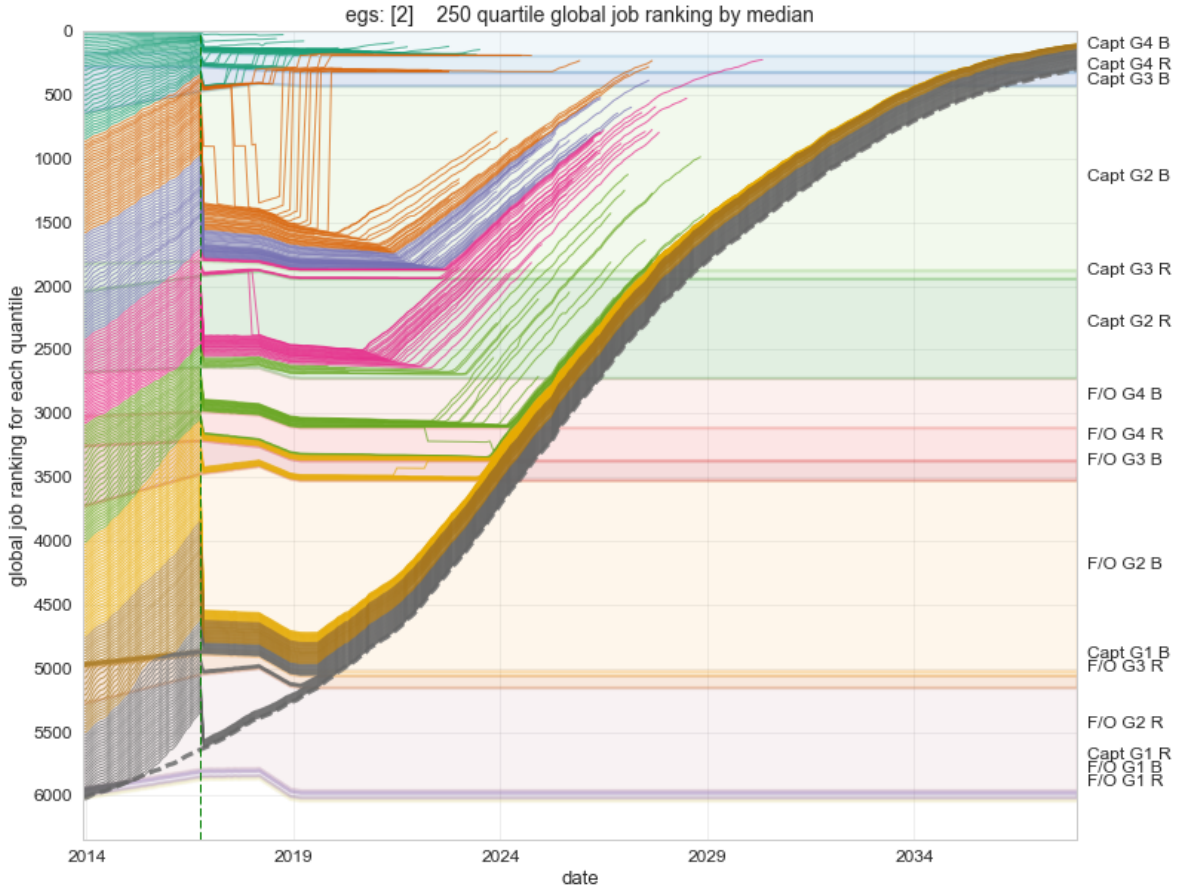


Fig. 71: p1 dataset, employee group 2, category order (job ranking), 250 quantiles, custom_color, with job bands

Every function has a list of all possible inputs and a description of those inputs contained within what is known as a docstring (documentation string (text)). The docstring may be viewed by clicking on the function name within a jupyter notebook cell, and then pushing the Shift + TAB keys at the same time.

The function docstrings also contain an overall description of the task performed by the function.

Docstrings may also be viewed within this website by clicking on the home button

and then scrolling to the bottom of the page and clicking the "Index" or "Module Index" text. Full function code may be accessed by subsequently clicking on the "[source]" text found after each function name. The top section of each function code section will contain the docstring.

Below, a docstring has been accessed within the jupyter notebook. The first four arguments in the *quantile_groupby* inputs correspond to the first four parameters listed in the docstring (or more technically here, the Signature).

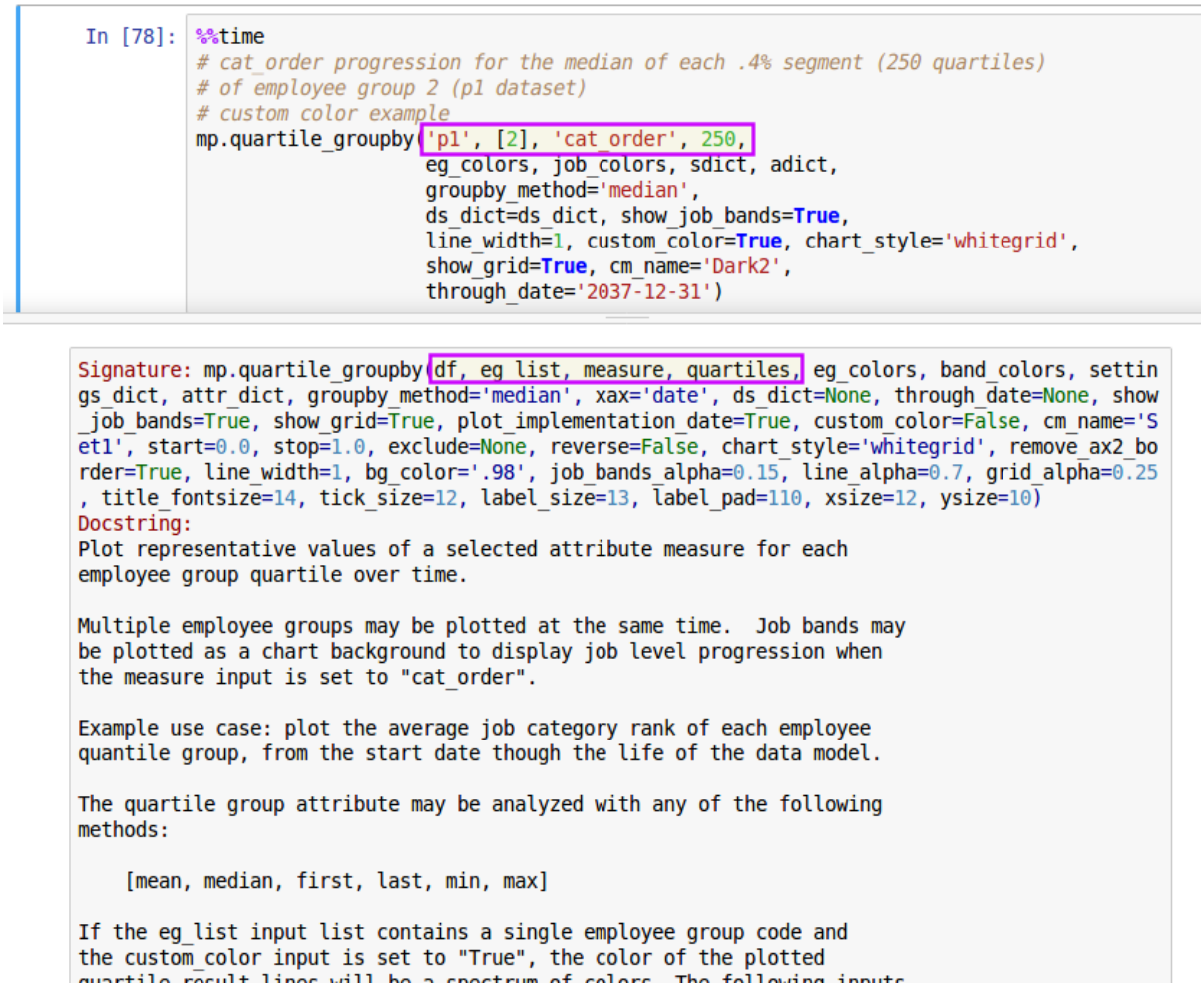


Fig. 72: function docstring accessed by clicking cursor on top of function name in the notebook cell, then pushing Shift + TAB keyboard buttons

Further down within the docstring associated with this function are the input descriptions. The descriptions provide information pertaining to the data type of each input and sometimes a short explanation of the purpose or effect of the input.

```
inputs
    df (dataframe)
        Any long-form dataframe which contains "date" (and "mnum" if xax
        input is set to "mnum") and "eg" columns and at least one
        attribute column for analysis.  The normal input is a calculated
        dataset with many attribute columns.
    eg_list (list)
        List of eg (employee group) codes for analysis.  The order of the
        employee codes will determine the z-order of the plotted lines,
        last employee group plotted on top of the others.
    measure (string)
        Attribute column name
    quartiles (integer)
        The number of quartiles to create and plot for each employee
        group in the eg_list input.
    eg_colors (list)
        list of color values for plotting the employee groups
    band_colors (list)
        list of color values for plotting the background job level
        color bands when the using a measure of 'cat_order' with the
        'show_job_bands' variable set to True
    settings_dict (dictionary)
        program settings dictionary generated by the build_program_files
        script
    attr_dict (dictionary)
        dataset column name description dictionary
    groupby_method (string)
        The method applied to the attribute data within each quartile.  The
        allowable methods are listed in the description above.  Default is
        'median'.
    xax (string)
        The first groupby level and x axis value for the analysis.  This
        value defaults to "date" which represents each month of the model.
        Alternatively, "mnum" may be used.
    job_levels (integer)
        The number of job levels (excluding the furlough level) in the data
        model.
    ds_dict (dictionary)
        A dictionary containing string to dataframes, used if df input
        is not a dataframe but a string key (examples: 'standalone', 'p1')
    through_date (date string)
        If set as a date string, such as '2020-12-31', only show results
        up to and including this date.
    show_job_bands
        If measure is set to "cat_order", plot properly scaled job level
        color bands on chart background
    show_grid (boolean)
        if True, plot a grid on the chart
    plot_implementation_date
        If True and the xax argument is set to "date", plot a dashed
        vertical line at the implementation date.
    custom_color (boolean)
        If set to True, will permit a custom color spectrum to be produced
        for plotting a single employee group "cat_order" result
    cm_name (string)
        The colormap name to be used for the custom color option
```

Fig. 73: beginning of input definitions for sample function

Note that the *quantile_groupby* function can present information relating to other metrics, not just the "cat_order" measure. It can also display more than one employee group at a time. These are more options which are easily selected by changing the function argument values.

Global settings which affect the way the datasets are calculated are controlled by values in the Excel input files. The function arguments only control how the previously calculated dataset information is displayed.

12. Run the **INTERACTIVE_PLOTTING** notebook
    - select attributes to compare with the dropdown selectors
    - use the slider and buttons to view data model results over time
    - change the "proposal" variable as needed to explore other datasets

create or edit lists

13. Run the **EDITOR_TOOL** notebook
    - modify list order to smooth distortions (see the "editing" discussion within the "program flow" section)

generate final list

14. Run the **join_inactives** script
    - reinsert inactive employees into the integrated list solution (see the "building lists" discussion within the "program flow" section)

5.6.2 changing program options or settings

Global settings include such things as basic vs. enhanced job hierarchies, delayed implementation, and changes in job counts over time. Datasets must be recalculated when any foundational parameter is modified.

To change a global program option, the input(s) within the Excel input file(s) must be modified and the program rerun as follows:

Changing main program input options or other parameters:

1. Open the appropriate Excel input file.
2. Change the value
3. Save the Excel file
4. Return to the Jupyter notebook

5. Restart the kernel:
    - click on the "Kernel" button in the menu bar and select "Restart"
    - then rerun notebook cells:
        - click on "Cells" button in menu bar, select appropriate item
6. Rerun program (generate program files)
    - recalculate the datasets, rerun the following scripts:
        - **build_program_files**
        - **make_skeleton**
        - **standalone**
        - **compute_measures** (for each proposal)
7. Restart other notebook kernels
    - rerun all analysis to reflect updated source data

Restarting the kernel flushes all previously loaded variable values. When notebook cells are rerun, the program will use any updated values derived from modified input files. If the notebook is not restarted after changing input file values or recalculating a dataset, it will not capture the updated values. The kernel must be restarted individually for all open notebooks - restarting the kernel for the **RUN_SCRIPTS** notebook will not restart the kernel for the **STATIC_PLOTTING** notebook, for example.

Plotting function arguments may be changed within a notebook cell and the cell rerun without any other action (a kernel restart or file saving beforehand is not required or desired).

5.6.3 saving/loading calculated case study data

The *save_and_load_dill_folder* function may be used to quickly switch between case studies by loading previously calculated and saved program-generated files (including calculated datasets).

saving

The "save" functionality will copy the current **dill** folder and save it in the **saved_dill_folders** folder, named as **<case study name>_dill_folder**. The **saved_dill_folders** folder will be created if it does not already exist. The function will perform the save action when it is executed without any arguments:

```
import functions as f

f.save_and_load_dill_folder()
```

The case study name will be automatically determined by reading the *dill/case_dill.pkl* file.

saving and loading

The "load" functionality will save the current **dill** folder, look for a saved dill folder corresponding to the string parameter provided to the "load_case" argument, and replace the current **dill** folder with the **dill** folder to load.

```
f.save_and_load_dill_folder(load_case='sample3')
```

If the specified load folder does not exist, the only action to occur will be saving the current **dill** folder. The function will alert the user that the load operation failed.

Here is an example of attempting to load a folder which does not exist:

```
f.save_and_load_dill_folder(load_case='bad_case_name')
```

. . . will give:

```
'"sample3" dill folder copied to:'

   'saved_dill_folders/sample3_dill_folder'


 '''Error >>>  problem finding a saved dill folder with a
↪"bad_case_name" prefix in the "saved_dill_folders" folder.''
↪'
```

The **dill** folder contents remain unchanged.

query for saved folders

The user may determine which case study dill folders are available to load by running the function with the "print_saved" argument set to True. All saved case study names will be printed and no other action will take place:

```
f.save_and_load_dill_folder(print_saved=True)
```

The print output will be in this format (the names of the case studies are examples only):

```
'The saved dill folders available to load are:'

  ['sample3', 'acme_southern']

'Nothing changed, set print_saved input to "False" if␣
↪you wish to save and/or load a folder'
```

5.6.4 anonymizing input data

The parties involved with an integration may consider certain input data attributes to be private and confidential, making it difficult or impossible to share the analysis results with others. seniority_list includes a set of specialized functions designed to address this issue.

Employee information - name, employee number, date of birth, date of hire, and longevity date - may be replaced with substitute values to de-identify personal information. Compensation tables may also be proportionally adjusted. This shielding of personal information offers a potential solution to privacy or proprietary concerns.

Anonymizing dates should be avoided if possible to avoid deviations from the original data model, due to the effect on retirements and other date-related measurements. The random date adjustments are small, but will invariably affect the results, even if slight.

The anonymizing functions are located within the *functions* module.

- *anon_master*
- *anon_pay_table*

> **Warning:** Even though the anonymizing functions are coded to create a copy of original data, it is recommended to copy and save the entire **excel** folder outside of the seniority_list file structure before applying any of the anonymizing methods.

Internally, the *anon_master* and *anon_pay_table* functions use the "sheet_name=None" option of the pandas read_excel method to return a dictionary of worksheets with worksheet name, dataframe as key, value pairs. The targeted worksheet (now represented as a dataframe) is selected and updated with anonymized values. Then the updated dataframe is written back to the appropriate worksheet within the excel file.

Simply rerun the program to produce datasets and visualizations incorporating the anonymized personal information.

anonymize *master.xlsx*

The aptly named *anon_master* function is used to anonymize the *master.xlsx* file. The user may select to anonymize any or all of the following attributes:

- last names (lname)
- employee numbers (empkey)
- birth date (dob)

- hire date (doh)
- longevity date (ldate)

A copy of the original *master.xlsx* file will be saved as *master_orig.xlsx*.

The function will generate new employee numbers (empkey) and last names (lname) by default.

All of the names in the "lname" column will be replaced with randomly generated substitute strings, and all empkeys will be replaced with substitute integer values. Empkeys will still begin will the appropriate employee group code number.

```
import functions as f

f.anon_master(<case name>)
```

Note: The <case name> placeholder in the code examples must be replaced with the string name for the case study, such as "sample3" or "acme".

To anonymize any or all of the date columns, set the "date" option to True.

```
f.anon_master(<case name>, date=True)
```

The default action is to adjust hire dates and longevity dates together from zero to five days forward, and separately (with different random sequence) adjust birth dates forward in the same fashion. These parameters are all adjustable with various inputs.

Another option related to randomizing date, "sampling", is also available to the user either through an option with the *anon_master* function, or by using the *sample_dataframe* function directly. Using the *anon_master* function, a sample, or subset, of a master list (by rows) may be randomly selected for testing or other purposes by setting the "sample" option to True. Sample size may be set with with a row count ("n") input or by a decimal fraction ("frac") input. Below, the "frac" input will direct the program to randomly select .2 or 20% of the rows in the master_df dataframe for the output.

```
f.anon_master(<case name>, sample=True, frac=.2)
```

All or none of anonymizing options discussed above may be applied to a master dataframe sample.

The modified excel file output from the *anon_master* function will be saved as *excel/<case name>/master.xlsx.*

anonymize *pay_tables.xlsx*

The underlying compensation information for the data model may be replaced with substitute data using the *anon_pay_table* function. The original hourly pay data may be reduced or increased, proportionally or disproportionally. A copy of the original pay date will be stored as *pay_tables_orig.xlsx*.

```
import functions as f

f.anon_pay_table(<case name>)
```

The "mult" input is a multiplier used to proportionally transform all of the pay rate data at once. The "mult=.5" input below would produce modified pay rates equal to 50% of the original rates.

```
f.anon_pay_table(<case name>, mult=.5)
```

The pay rates data may also be "randomized" in a disproportionate manner by setting the "proportional" input to False. The data will be altered with a fixed algorithm.

```
f.anon_pay_table(<case name>, proportional=False)
```

reversion to original data

The *copy_excel_file* function includes a "revert" option which will delete an anonymized file and replace it with the original data.

Restore the *master.xlsx* file:

```
f.copy_excel_file(<case name>,
                  'master',
                  revert=True)
```

Restore the *pay_tables.xlsx* file:

```
f.copy_excel_file(<case name>,
                  'pay_tables',
                  revert=True)
```

5.7 program restoration

If for some reason, a portion of the code base is accidentally deleted or corrupted, simply save all custom input files to a directory outside of the main **seniority_list** folder, then delete the main **seniority_list** folder and reinstall the program. After reinserting the input files in the proper locations, the program will be ready to operate again.

Specific files to preserve for reinsertion after reinstalling the program:

- entire case-specific folders within the **excel** folder which hold Excel input files
- any edited lists (*p_edit.pkl*) and/or datasets (*ds_edit.pkl*) from use of the editor tool

All other files are quickly reproduced when the program is run.

CHAPTER

SIX

EXCEL INPUT FILES

seniority_list is designed to produce comprehensive datasets reflecting the data model(s) described by the user. The information and the data model description necessary for this process is transmitted to seniority_list through simple spreadsheet workbooks.

While the Microsoft Excel program may be used to produce the workbooks, any spreadsheet program may be used to work with seniority_list as long as it can produce .xlsx files (such as LibreOffice Calc). The reference to "Excel" throughout this user guide refers to .xlsx files, not specifically the Excel program.

There are four Excel files required as inputs for each case study. They are located within an appropriately-named folder, created by the user. There may be many different case study folders within seniority_list ant any one time. The case study folders are located within the **excel** program folder.

```
excel/<name_of_case_study>/
```

With our hypothetical case, this would translate to a folder and file as such:

```
excel/southern_acme/
```

The sections below will walk through each of the four Excel input files and will provide detailed data and formatting requirements for each of them.

Note: The task of formatting input files as described below will be magnitudes simpler if another case study folder is copied into the **excel** folder, renamed to match the new case study name, and the workbooks therein modified as appropriate.

In the following image, the highlighted section within the **sample3** folder shows the excel files included with the program for the sample case study. The name of the sample case study included with the seniority_list program is "sample3". For our theoretical case ("southern_acme"), the user would copy the **sample3** folder, paste it

back into the **excel** folder, and rename it "southern_acme". The user would then open the Excel workbooks in the **southern_acme** folder and modify the contents of the worksheets as appropriate.

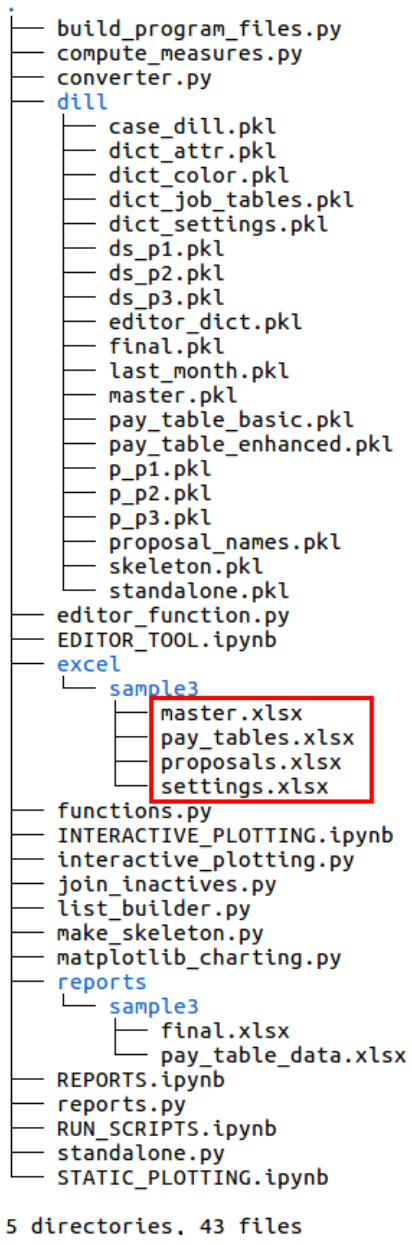


Fig. 1: tree view of seniority_list package, with the Excel input files highlighted

---

Note: All date inputs must be formatted as dates within each spreadsheet input file. Right-click on any cell or group of cells containing dates and select "format cells" or something similar and verify date format. If the date inputs are actually formatted as text even though they look like dates,

the program will not be able to generate the files needed to run the program.

6.1 master.xlsx

The *master.xlsx* workbook provides basic employee data to the program:

Employee data (for each employee)

- unique employee key (number)
- employee group membership
- last name
- date of birth
- date of hire
- longevity date
- special group membership
- furlough status
- work status (active or inactive)
- order within employee group

	A	B	C	D	E	F	G	H	I	J
1	**empkey**	**eg**	**lname**	**dob**	**doh**	**ldate**	**sg**	**fur**	**line**	**eg_order**
2	10011102	1	tooeyoo	1949-07-13	1973-02-26	1975-01-29	0	0	1	1
3	10010475	1	rubelot	1949-02-05	1975-05-27	1975-05-27	0	0	1	2
4	10013096	1	yeloxid	1949-01-08	1977-01-18	1977-01-18	0	0	1	3
5	10012178	1	xayeaue	1951-06-07	1977-11-15	1977-11-15	0	0	1	4
6	10014447	1	finuceu	1951-10-17	1977-12-09	1977-12-09	0	0	1	5
7	10014384	1	pevagai	1954-06-02	1978-01-12	1978-01-12	0	0	1	6
8	10012843	1	quniy	1953-05-14	1978-02-18	1978-02-18	0	0	1	7
9	10014067	1	qimed	1955-05-25	1978-03-01	1978-03-01	0	0	1	8
10	10010929	1	xovejiq	1955-06-20	1978-03-07	1978-03-07	0	0	1	9
11	10014351	1	yanag	1950-03-22	1978-03-12	1978-03-12	0	0	1	10
12	10011974	1	eipieuv	1955-02-14	1978-03-21	1978-03-21	0	0	1	11

Fig. 2: master.xlsx file format example

master.xlsx contains only one worksheet and the name of the worksheet is unimportant.

master.xlsx contains information for the employees of all groups. The physical order of the list is unimportant at this step, as long as the "eg_order" (employee group order, or order within each group) column data value is correct, even if the list is not sorted according to this value.

Note that since there is only one master list stored within the program, any employee data discrepancies which may exist between the information supplied by the various parties must be resolved as part of the data preparation phase.

6.1.1 master.xlsx format guide

Do not add any additional columns to the worksheet, such as a row count column. Only include the columns shown above and described below, with the exact column names in the first row, in lower case. Columns A, B and G through J should be formatted as numbers (integers), column C as strings (text or general), and columns D through F as dates (format YYYY-MM-DD).

The *master.xlsx* employee data worksheet must have one row for each employee and columns of attributes for each employee as follows:

1. **empkey (standardized employee number, integer)**

   Recommended format: employee number + (10,000,000 * eg (employee group) number)

   #23456 in eg 1 becomes #10023456

   #23456 in eg 2 becomes #20023456

2. **eg (employee group, integer)** Assign the same number (i.e. 1, 2, etc.) in this column to each member of the same group. Always begin with the number one and use sequential numbers for other groups. This format is important for proper operation of other functions within the program.

3. **lname (last name, string)** lowercase

4. **dob (date of birth, date format)** Date of birth is used to calculate retirement date using a retirement age input. The program will correctly compensate for leap years.

5. **doh (date of hire, date format)** Normalized initial class date

6. **ldate (longevity date, date format)** Date for pay longevity and/or non-furlough time calculations

7. **sg (special group, integer (1 or 0))** Employees with special job conditions are marked with a 1, others with a 0. If there are no employees with special job conditions, the values for the entire column should be zeros.

8. **fur (furlough, integer (0 or 1))** Furloughed employees are marked with a 1, all others with a 0. If there are no furloughed employees, the values for the entire column should be zeros.

9. **line (line, integer (0 or 1))** Line (active) employees. Active employees are marked with a 1, others (sick leave, supervisory, etc.) are marked with a

0. If there are no inactive employees (unlikely), the values for the entire column should be zeros.

10. **eg_order (employee group order, integer)** A number that represents the correct list order within each employee group, starting with 1 for each employee group. These numbers are independent of a combined list number.

The information within *master.xlsx* is read by seniority_list and is stored within the **dill** folder as a pandas dataframe, *master.pkl*. The dataframe structure matches the worksheet structure with the addition of a calculated retirement date column ("retdate").

6.2 proposals.xlsx

The *proposals.xlsx* workbook provides the following information to the program:

Integrated list data (for each proposal)

- order by unique employee number (empkey)
- proposal names (as set by worksheet names)

	A	B
1	**order**	**empkey**
2	1	10011102
3	2	10010475
4	3	10013096
5	4	10012178
6	5	10014447
7	6	10014384
8	7	10012843
9	8	10014067
10	9	10010929
11	10	10014351
12	11	20010692
13	12	10011974
14	13	10014916
15	14	10013233
16	15	10010598
17	16	10014294
18	17	20011034
19	18	10011916
20	19	10011396
21	20	10012952
22	21	10013060
23	22	10014377
24	23	10010054
25	24	10012113
26	25	10013413
27	26	10014156
28	27	10011656

Fig. 3: example list order, used to order skeleton file

The proposal orderings are derived from the proposed integrated lists supplied by the parties. There is no limit imposed by seniority_list to the number of proposals which

may be included on separate worksheets.

Note: seniority_list may process list orderings from other sources (the editor tool and the **list_builder.py** script). These features are discussed within the user guide.

6.2.1 proposal.xlsx format guide

The *proposals.xlsx* Excel file is a multi-sheet workbook, with each sheet containing a different list ordering proposal. The names of the worksheets are incorporated into the names of the resultant dataset names and will be the reference when working with the various outcomes for analysis and plotting. Therefore, the names of the worksheets should reflect the proposal therein. With our hypothetical integration study, the worksheets should be named "southern" and "acme" (or a shorter abreviation), reflecting the proposals from each group. The proposal names should be limited to a maximum of 10 characters. Short proposal names are preferred, because these names will be used as inputs to many of the plotting functions.

The worksheets must contain at least one column with the header "empkey" (employee number key, exact spelling, lower case) containing the unique empkeys in the proposed order. The "order" column is not technically required for program operation but may be included as a user convenience with no detriment to program operation. The "empkey" column should be formatted as a number (integer), not text.

Each proposal must contain the same list of empkeys (employee numbers), reflecting the active employees as determined by the *master.xlsx* file "line" column.

6.3 pay_tables.xlsx

pay_tables.xlsx provides the following information to the program:

Compensation

- pay rate tables for each basic job level, employee longevity, and contract year category
- pay rates for an interim period
- number of modeled pay hours per month for each job level, basic and enhanced

Jobs

- basic-to-enhanced job conversion data
- job level text descriptions

The program uses the data supplied from the *pay_tables.xlsx* workbook to create optimized compensation lookup files which are used when the datasets are generated. The data is also used to create a multi-sheet Excel workbook containing computed monthly pay tables and job level hierarchy tables. This workbook, *pay_table_data.xlsx*, is written to a case study folder within the **reports** folder. The workbook format permits the computed pay data to be reviewed by the user.

The pay-related files are created when the **build_program_files.py** script is run.

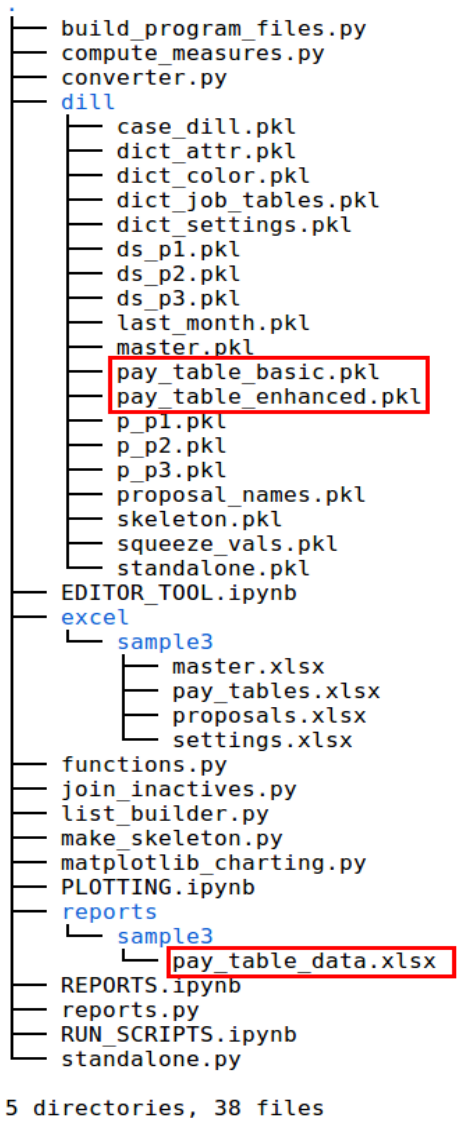


Fig. 4: files produced from *pay_tables.xlsx* data

basic vs. enhanced job levels

seniority_list is designed with the capacity to handle two different job hierarchy methodologies.

The first method is a basic mode which assumes a "stovepipe" or linear movement upwards through the distinct job levels, each of which have a defined compensation rate and a uniform number of monthly pay hours within each job level.

The second method is the "enhanced" mode which offers additional job level layers for the program data model when it is appropriate. This would occur when contractual or other provisions provide for some workers to receive less monthly pay hours than other workers at the same compensation level.

Note: It is not a requirement to incorporate "enhanced" job levels within the model when they do not exist for the industry case or are not desired. In that case, the "enhanced_jobs" value on the "scalars" worksheet within *settings.xlsx* should be set to "False".

For example, assume that an industry contract defines five separate job levels, ranging in hourly pay from $20/hour to $100/hour, and assumes each worker will be paid 160 hours/month. In this case, the "basic" mode of job hierarchy is appropriate and completely sufficient to model job and compensation projections.

However, if that industry contract further defines that some workers at each level will work and be paid 120 hours/month, this doubles the number of job levels to be considered in an integration analysis, because each job level contains two categories of monthly pay hours. It also complicates the career progression model, since employees will likely prefer positions based on total compensation amounts, not just hourly rate of pay.

The job hierarchy mode is selected via the "enhanced_jobs" value (True or False) within the *settings.xlsx* workbook ("scalars" worksheet).

6.3.1 pay_tables.xlsx format guide

The *pay_tables.xlsx* workbook contains compensation data on two worksheets with specific names, in lower case:

- **"rates"** formatted hourly pay rate table for basic job levels including contractual pay changes and longevity increments
- **"hours"** small table containing basic and enhanced job level hours per month and descriptive job codes

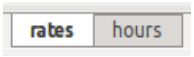


Fig. 5: worksheet tabs within the Excel *pay_tables.xlsx* workbook

rates

The pay table "rates" worksheet has a straightforward formatting layout. The user must create the the worksheet from contractual pay information. There is one pay data worksheet for each case study.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	**year**	**jnum**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
2	2013.0	1	25.32	120.24	121.21	122.17	123.14	124.10	125.07	126.03	127.00	127.97	128.93	129.90
3	2013.0	2	25.32	102.21	103.02	103.84	104.66	105.48	106.30	107.12	107.94	108.76	109.58	110.40
4	2013.0	3	25.32	94.89	95.65	96.41	97.17	97.93	98.69	99.45	100.21	100.97	101.73	102.49
5	2013.0	4	25.32	60.69	73.18	74.97	76.78	78.61	80.46	82.32	84.84	86.75	88.04	88.70
6	2013.0	5	25.32	51.68	62.27	63.79	65.33	66.88	68.45	70.03	72.17	73.79	74.88	75.44
7	2013.0	6	25.32	48.02	57.85	59.26	60.68	62.12	63.58	65.05	67.03	68.53	69.55	70.06
8	2013.0	7	25.32	64.89	65.41	65.93	66.44	66.96	67.48	67.99	68.51	69.03	69.54	70.06
9	2013.0	8	25.32	33.02	39.71	40.66	41.63	42.61	43.60	44.60	45.94	46.97	47.66	48.01
10	2014.0	1	25.32	129.77	130.81	131.85	132.90	133.94	134.98	136.03	137.07	138.11	139.15	140.20
11	2014.0	2	25.32	110.29	111.17	112.06	112.94	113.83	114.71	115.60	116.48	117.37	118.25	119.14
12	2014.0	3	25.32	102.39	103.21	104.03	104.85	105.68	106.50	107.32	108.14	108.96	109.78	110.60
13	2014.0	4	25.32	65.46	78.95	80.88	82.83	84.81	86.80	88.82	91.53	93.60	94.99	95.70
14	2014.0	5	25.32	55.72	67.16	68.80	70.46	72.14	73.83	75.54	77.85	79.60	80.78	81.38
15	2014.0	6	25.32	51.77	62.39	63.91	65.45	67.00	68.57	70.16	72.30	73.92	75.02	75.58
16	2014.0	7	25.32	69.99	70.55	71.11	71.67	72.23	72.78	73.34	73.90	74.46	75.02	75.57
17	2014.0	8	25.32	35.57	42.79	43.82	44.87	45.93	46.99	48.07	49.53	50.63	51.38	51.76
18	2014.1	1	154.36	155.61	156.87	158.12	159.37	160.62	161.88	163.13	164.38	165.63	166.88	168.13
19	2014.1	2	130.28	131.38	132.44	133.49	134.62	135.66	136.66	137.78	138.75	140.23	141.72	143.18
20	2014.1	3	123.46	124.46	125.46	126.48	127.51	128.52	129.53	130.55	131.55	132.64	133.75	134.84
21	2014.1	4	45.18	83.78	97.86	100.20	102.57	105.13	108.03	110.48	111.66	113.16	114.18	115.20
22	2014.1	5	45.18	70.82	82.68	84.66	86.70	88.85	91.26	93.37	94.32	95.86	97.02	98.15
23	2014.1	6	45.18	67.12	78.35	80.23	82.15	84.19	86.52	88.50	89.44	90.69	91.58	92.46
24	2014.1	7	81.50	82.11	82.78	83.44	84.06	84.74	85.38	86.05	86.70	87.37	88.04	88.68
25	2014.1	8	45.18	45.18	51.84	53.07	54.30	55.65	57.17	58.46	59.07	59.87	60.42	60.93
26	2015.0	1	158.96	160.24	161.54	162.83	164.12	165.41	166.70	167.99	169.28	170.56	171.85	173.14
27	2015.0	2	134.16	135.28	136.38	137.46	138.63	139.69	140.73	141.88	142.88	144.40	145.94	147.44
28	2015.0	3	127.13	128.16	129.19	130.24	131.30	132.34	133.38	134.43	135.47	136.59	137.72	138.85
29	2015.0	4	46.50	86.27	100.76	103.17	105.61	108.25	111.23	113.77	114.97	116.52	117.57	118.62
30	2015.0	5	46.50	72.91	85.12	87.16	89.27	91.48	93.97	96.14	97.11	98.70	99.89	101.07
31	2015.0	6	46.50	69.10	80.66	82.60	84.58	86.68	89.08	91.12	92.09	93.38	94.29	95.20

Fig. 6: *pay_tables.xlsx* format example, "rates" worksheet

The worksheet can be thought of as employee pay rate tables for multiple years, stacked together forming one table. Within each contractual year block, the pay rates for the various job levels are positioned vertically from highest to lowest and longevity pay increases are positioned horizontally, lowest to highest.

All columns are formatted as numbers. The header row contains a "year" column and a "jnum" column (both lower case), and other columns with integer headers representing the longevity year pay steps (1 through the top of scale year).

The data in the year column is a float type (decimal number) representation of the applicable contract pay year. The year 2018 would be represented as "2018.0".

The year column may include one or more exception values ("2014.1" in the image below) which allow for a temporary or interim pay scale(s) if they exists. An interim

pay scale might exist for a certain transitional time period such as a partial year at new contract pay rates. A pay exception year value and duration is set using the "pay_exceptions" worksheet in *settings.xlsx*. Simply adding .1 to the year in which a pay exception occurs will allow for the fastest follow-on indexing calculations which utilize this data.

The data within the "jnum" (job number) column represents the different job levels within the data model. A job level of 1 represents the highest paid position, with subsequent incremental job level numbers representing job levels with decreasing compensation rates. The program will automatically insert an additional job level after the lowest job level to represent employees who are furloughed (or who could become furloughed in various modeling scenarios) when the Excel file is read. A model with 8 active job levels will be modified to have job numbers 1-9 in the "jnum" column for each contract year in the "year" column, with job number 9 representing furlough with no pay. This can be reviewed by examining the *pay_table_data.xlsx* file within the **reports** folder after running the **build_program_files.py** script.

The year longevity columns (integers) hold the hourly compensation data for employees with various levels of service with the enterprises. Column 1 would be the rates for employees working in their 1st year of service, column 5 would contain the rates for employees working in their 5th year, etc. up to the maximum longevity scale. Employees with more years than the maximum longevity scale are capped at the maximum longevity rates.

Note that the year column has repeating row values for each of the job levels. Also, the pay exception built into this pay table is evident with all the rows with 2014.1 in the "year" column. The user may directly examine the Excel files included with the program for further clarity (*excel/sample3/pay_tables.xlsx*).

hours

This worksheet supplies the program data model with the number of monthly pay hours applicable to each job level. In other words, by inputing data into this table, the user sets the average number of pay hours to be allocated within each job level, for both the basic (no full- and part-time pay within job levels considered) and the enhanced (including full- and part-time jobs within job categories) job data model options.

	A	B	C	D	E	F	G
1	**jnum**	**basic_hours**	**full_hours**	**part_hours**	**full_pcnt**	**jobstr**	
2	1	81	85	74	0.600	Capt G4	
3	2	81	85	74	0.625	Capt G3	
4	3	81	85	74	0.650	Capt G2	
5	4	81	85	74	0.600	F/O G4	
6	5	81	85	74	0.625	F/O G3	
7	6	81	85	74	0.650	F/O G2	
8	7	81	85	74	0.650	Capt G1	
9	8	81	85	74	0.650	F/O G1	
10							
11							

Fig. 7: pay_tables.xlsx format example, "hours" worksheet

The "jobstr" column ("job string") is used by the program to provide a short job level description in text form for various chart legends and titles. If an enhanced model is selected, designated full- and part-time suffixes will be added to the enhanced job descriptions appropriately. The suffixes are specified by the user through the "enhanced_job_part_suffix" and "enhanced_jobs_full_suffix" values provided through the "scalars" worksheet within the *settings.xlsx* workbook. The enhanced job strings may be viewed on the "enhanced ordered" worksheet within the *pay_table_data.xlsx* file within the **reports/<case study>** folder after the program files have been constructed.

The "jnum" column (job number) contains the integer code value for each non-furlough basic job level within the data model, in sequential order. These job numbers correspond to the job numbers in the "jnum" column in the "rates" worksheet.

The "basic_hours" column contains a user-specified number of monthly pay hours for each job level. The number of pay hours may vary for different job levels. Calculated values derived from this column are utilized by the program when the user specifies a basic, non-enhanced data model, by setting the "enhanced_jobs" option to "FALSE" on the "scalars" worksheet within the *settings.xlsx* workbook.

The "full_hours", "part_hours", and "full_pcnt" columns contain data pertaining to enhanced job models. As in the "basic_hours" column, the values in each of these columns may vary from job to job, as required.

- full_hours - the number of monthly hours within a job level for "full-time" employees
- part_hours - the number of monthly hours within a job level for "part-time" employees
- full_pcnt - the percentage of all jobs within a job level to be allocated as "full-time". The remaining percentage of jobs will be allocated as "part-time".

Even if a basic data model is selected by the user, the above columns must remain in place to prevent a calculation error, though the enhanced job model inputs will not be used for further program analysis.

6.3.2 job level hierarchy

seniority_list uses a job-level hierarchy based on compensation. This hierarchy determines the order of job assignments and employee compensation throughout the entire data model.

A basic or non-enhanced data model assumes that the proper value order of job levels is as supplied by the user through the *pay_tables.xlsx* input workbook, with job level 1 the best-paying and most desirable, and therefore, most "senior" job level.

However, it is possible that some job level(s) may compensate workers proportionally more or less in certain pay-scale longevity years as compared to other job levels. This means that independent sorts of job level compensation for all contract years and/or longevity steps could show slightly different orderings when the underlying job level pay rates vary enough over contract years or, more likely, contract longevity steps.

When an enhanced model is used, additional discrepancies in compensation sorts may be introduced, when the issue above is combined with the more numerous enhanced job levels.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		year	jnum	1	2	3	4	5	6	7	8	9	10	11	12
102	100	2017	16	3645	3645	4184	4283	4383	4493	4615	4720	4769	4832	4877	4919
103	101	2017	17	0	0	0	0	0	0	0	0	0	0	0	0
104	102	2018	1	14755	14875	14996	15116	15235	15354	15474	15594	15714	15833	15953	16073
105	103	2018	2	12846	12950	13055	13159	13263	13367	13472	13576	13680	13784	13888	13993
106	104	2018	3	12452	12556	12657	12759	12867	12966	13062	13169	13262	13404	13546	13685
107	105	2018	4	11799	11895	11990	12087	12186	12283	12379	12477	12573	12678	12783	12888
108	106	2018	5	10840	10931	11019	11108	11202	11288	11372	11465	11545	11669	11793	11914
109	107	2018	6	10272	10356	10438	10523	10609	10694	10777	10862	10946	11037	11129	11220
110	108	2018	7	4310	8003	9349	9574	9800	10045	10322	10557	10670	10814	10911	11009
111	109	2018	8	3752	6967	8139	8335	8531	8745	8987	9191	9289	9414	9499	9584
112	110	2018	9	4310	6763	7897	8087	8282	8488	8719	8921	9011	9159	9269	9378
113	111	2018	10	4310	6409	7483	7663	7847	8043	8265	8454	8544	8664	8749	8833
114	112	2018	11	7785	7843	7907	7970	8030	8095	8156	8220	8282	8346	8411	8472
115	113	2018	12	3752	5887	6875	7040	7211	7390	7591	7766	7845	7974	8070	8164
116	114	2018	13	3752	5580	6515	6671	6832	7002	7196	7360	7438	7543	7617	7690
117	115	2018	14	6778	6828	6883	6938	6991	7047	7100	7157	7211	7266	7322	7376
118	116	2018	15	4310	4310	4947	5065	5183	5312	5457	5581	5639	5715	5768	5817
119	117	2018	16	3752	3752	4307	4410	4513	4624	4751	4859	4909	4975	5022	5064
120	118	2018	17	0	0	0	0	0	0	0	0	0	0	0	0
121	119	2019	1	15195	15318	15443	15566	15689	15813	15936	16059	16182	16305	16428	16552

Fig. 8: uneven enhanced job level sort (highlighted with color conditional formatting for clarity), primarily job levels 11 and 14 for longevity steps 1 through 6, within contract year 2018. In this case, the chosen sort index is contract year 2018, longevity step 7 (outlined).

Because of this possibility it is recommended that the user carefully examine the "basic_ordered" and "enhanced_ordered" worksheets within the *pay_table_data.xlsx* workbook, located within **reports** folder. This workbook is automatically generated by seniority_list when the **build_program_files** script is run. These worksheets display the job level hierarchy created by the program, according to a compensation sort for a particular contract year scale and contract longevity step. These values are set and adjusted by the user through the "pay_table_year_sort" and "pay_table_longevity_sort" value inputs in the "scalars" worksheet within the *settings.xlsx* input excel file. A

contract year and longevity step should be selected which provides the best overall compensation sorts for the life of the model as indicated by the information in the *pay_table_data.xlsx* each time new program files are generated.

The job level sorting algorithm will ensure that the order of jobs will be the same in all contract years and longevity steps throughout the data model. If the user wishes to adjust the pay level sort indexing, the *settings.xlsx* workbook must be saved after inserting the new values and the **build_program_files** script must be rerun. The results of this procedure will include replacing the *pay_table_data.xlsx* file with a new file containing the updated results.

Job compensation sort variation will not exist in all case studies, and in fact is likely to be an unusual situation. When it does exist, seniority_list provides a method to minimize its effect by allowing users to chose a pay sorting index point. This offers the best overall solution to construct a consistent job level hierarchy, while controlling (usually minor) uneven job level compensation rates uniformly for all of the employee groups.

6.4 settings.xlsx

settings.xlsx provides the following information to the program:

Jobs

- job counts
- job count changes schedule

Compensation

- annual compensation increase or reduction after contract expiration
- contract top of scale longevity (years)
- pay raise option on/off
- pay raise percentage
- pay scale exception year code(s) and duration(s)
- pay scale exception start/end dates
- compute pay measures on/off
- pay table year and longevity for job level hierarchy sort

Furlough

- recall schedule

Descriptions

- employee group number to text descriptions for stats/charting output
- jobs to text descriptions for stats/charting output
- some of the color lists for visualization
- dataset attribute descriptions

Job Assignment Special Conditions

- schedule
- jobs affected
- other supporting data

Dates

- starting date
- delayed implementation date

Special Conditions

- pre-existing job rights
- ratio job assignment
- ratio count-capped job assignment
- minimum count job assignment

Retirement Age

- retirement age increase on/off
- retirement age increase dates and age increase

Merger-specific Information

- abbreviations, proposal names, job descriptions, chart colors
- label dictionaries

Integration Delay

- delayed implementation (on/off)

Jobs

- compute results using job change information vs a static number of jobs (on/off)
- number of job levels (enhanced/basic)

Furlough

- compute incorporating recall (on/off)
- ignore time furloughed for longevity calculation (on/off)
- include furloughed employees when calculating certain list percentages (on/off)

Calculation Type

- no bump-no flush on/off
- actives_only on/off
- lspcnt calculation on/off
- columns to include in the dataset
- compute cat_order (global job category rank) on/off

File Storage

- save_to_pickle on/off

The *settings.xlsx* workbook contains many worksheets and is the source for the four dictionaries which seniority_list uses to produce datasets and build chart displays, with the exception of much of the color dictionary which is generated internally from the matplotlib colormap collection.

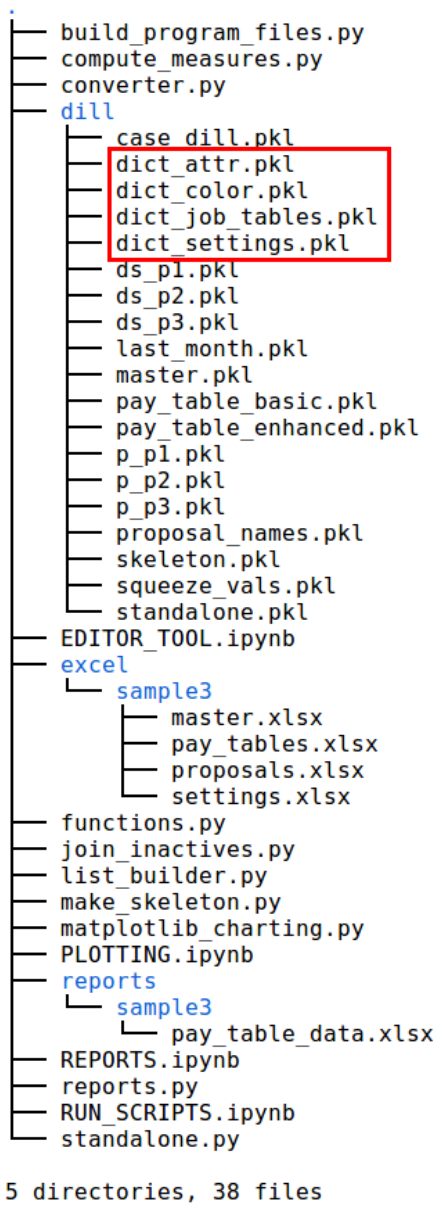


Fig. 9: The attribute, color, and settings dictionary files

6.4.1 settings.xlsx format guide

The *settings.xlsx* workbook contains multiple worksheets with various formatting requirements. This section describes those requirements and is applicable to any case study.

The *settings.xlsx* file is customized by manually updating worksheet cell values and possibly adding or deleting certain rows/columns as appropriate for each case study.

It is important to not change the structure of the worksheets or the "headers" (first row column names) in each worksheet except as described below. Many of the Python routines are looking for a specific layout to be able to gather and process the data correctly.

Except for the "scalars" and "attribute_dict" worksheets (the first two), the definitions below refer to columns in each sheet.

All dates used in seniority_list to designate a calendar month or starting and ending dates are end-of-month dates. "2015-12-31" is ok, "2015-01-01" will fail.

The screenshots below walk through each worksheet of the sample ("sample3") case-specific *settings.xlsx* file. The displayed values in the screenshots are from the sample case study which must be changed to the appropriate values for each actual case study, using the guidance below.

The 14 worksheets within the *settings.xlsx* workbook:

1. scalars
2. attribute_dict
3. ret_incr
4. pay_exceptions
5. job_counts
6. job_changes
7. recall
8. prex
9. ratio_cond
10. ratio_count_capped_cond
11. proposal_dictionary
12. eg_colors
13. basic_job_colors
14. enhanced_job_colors

scalars

The "scalars" worksheet contains a column of options and a column of corresponding values. The items in the value column are set by the user as appropriate/desired. The items in the option column should not be changed - the program looks for these specific phrases when it operates.

	A	B
1	**option**	**value**
2	enhanced_jobs	TRUE
3	job_levels_basic	8
4	job_levels_enhanced	16
5	enhanced_jobs_full_suffix	B
6	enhanced_jobs_part_suffix	R
7	compute_with_job_changes	TRUE
8	no_bump	TRUE
9	recall	TRUE
10	discount_longev_for_fur	TRUE
11	lspcnt_calc_on_remaining_population	FALSE
12	starting_date	2013-12-31
13	delayed_implementation	TRUE
14	implementation_date	2016-10-31
15	integrated_counts_preimp	FALSE
16	compute_pay_measures	TRUE
17	future_raise	FALSE
18	annual_pcnt_raise	0.02
19	top_of_scale	12
20	pay_table_year_sort	2018
21	pay_table_longevity_sort	7
22	init_ret_age_years	65
23	init_ret_age_months	0
24	ret_age_increase	FALSE
25	dist_sg	part
26	dist_ratio	split
27	dist_count	split
28	compute_job_category_order	TRUE
29	add_eg_col	TRUE
30	add_retdate_col	TRUE
31	add_doh_col	TRUE
32	add_ldate_col	TRUE
33	add_lname_col	TRUE
34	add_line_col	TRUE
35	add_sg_col	TRUE
36	add_ret_mark	TRUE
37	save_to_pickle	TRUE
38		

Fig. 10: Program options are set on this sheet along with many other single value variables.

GENERAL

- **enhanced_jobs** : boolean

  True - use enhanced job levels model
  False - use basic job levels model

- **job_levels_basic** : integer

  The number of job levels in the model without any enhancement for full- or part-time stratification. Include jobs that are found in any employee group even if those jobs are not found in all employee groups.

- **job_levels_enhanced** : integer

  Total count of part- and full-time job levels. Normally double the number of basic levels.

- **enhanced_jobs_full_suffix** : string

  The suffix to append to full-time job level descriptions, for print-out within the *pay_table_data.xlsx* report, stored within the case-specific folder in the **reports** folder.

- **enhanced_jobs_part_suffix** : string

  The suffix to append to part-time job level descriptions, for print-out within the *pay_table_data.xlsx* report, stored within the case-specific folder in the **reports** folder.

- **compute_with_job_changes** : boolean

  True - use job count changes in model
  False - assume static or constant job counts for model

- **no_bump** : boolean

  True - compute data model utilizing the no-bump, no-flush system
  False - allow full flush and bump for job assignment

- **recall** : boolean

  True - include recall of furloughed employees within data model
  False - inhibit recall

- **discount_longev_for_fur** : boolean

  True - do not apply furlough time towards longevity time
  False - allow furlough time to be included as longevity time

- **lspcnt_calc_on_remaining_population** : boolean

  True - include furloughed employees and employees remaining each month within denominator for list percentage (lspcnt) calculations

False - include furloughed employees and the greater of employees remaining or jobs available each month within denominator for list percentage (lspcnt) calculations

- **starting_date** : date string

  The effective data of the merger. This date is different than the implementation date. The effective date is the date when participating employees and the corresponding list data is frozen for modeling purposes.

- **delayed_implementation** : boolean

  True - permit independent operation of the employee group seniority lists until the implementation date to be included within the integrated dataset

  False - calculate dataset with an integrated seniority list commencing on the starting_date

- **implementation_date** : date string

  The anticipated date when the separate employee seniority lists will be integrated into one list. seniority_list will model each group separately until this date.

- **integrated_counts_preimp** : boolean

  True - Assign integrated job counts prior to implementation date when calculating integrated dataset.

  False - Use separate employee group job counts prior to the implementation date when calculating integrated dataset.

COMPENSATION DATA

- **compute_pay_measures** : boolean

  True - compute and include compensation-related metrics within the calculated dataset(s). The dataset will include the following columns:

```
['mlong', 'ylong', 'scale', 'mpay', 'cpay']
```

  False - do not compute compensation-related metrics. This will cut down on dataset computation time when pay-related columns are not needed.

- **future_raise** : boolean

Set to True if the model will incorporate an assumed raise at the end of the current contract.

- **annual_pcnt_raise** : float

  Assumed annual increase (or decrease) in pay rates if the future_raise input is True. A two percent annual increase would be set as .02

- **top_of_scale** : integer

  The number of longevity pay levels. The model assumes the same pay rates for all groups, before and after the implementation date.

INDEXED PAY TABLE

- **pay_table_year_sort** : float

  The calendar year within the pay table to use for the model job level hierarchy sort. The year 2018 would be represented as 2018.0

- **pay_table_longevity_sort** : integer

  The longevity year within the pay table to use for the model job level hierarchy sort (used in combination with the pay_table_year_sort input).

RETIREMENT

- **init_ret_age_years** : integer

  Initial retirement age in years. This is described as initial, because the retirement age may be increased in a future year(s) within the data model.

- **init_ret_age_months** : integer

  If the init_ret_age_years input above is not an even year value, use this input to add the number of months needed to represent the correct retirement age. A retirement age of 65.5 would mean that this input should be 6, to represent 6 months. Set at zero if the retirement age is an even year.

- **ret_age_increase** : boolean

  Set to True if the model will incorporate an increase in retirement age. If False, the ret_incr variable below will be ignored.

BASIC TO ENHANCED CONVERSION

- **dist_sg** : string

  If the enhanced_jobs option value is True, meaning a data model containing enhanced job levels has been selected, this input will determine how basic job level counts are converted and distributed to enhanced job levels.

For example, if enhanced job levels 3 and 5 are the full- and part-time jobs associated with basic level 2, the user may direct that all of the jobs from level 2 be assigned to job 3 (full-time), to job 5 (part-time), or divided between the two according to the percentages specified within the job dictionary (jd variable).

The possible value inputs are:

```
['full', 'part', 'split']
```

The dist_sg (distribution to special group) input controls the distribution of job counts from basic job levels affected by pre-existing job rights to the corresponding enhanced job level counts.

- **dist_ratio** : string

  Same as above, but controlling the ratio_cond enhanced job count conversion/distribution.

- **dist_count** : string

  Same as above, but controlling the ratio_count_capped_cond enhanced job count conversion/distribution.

OPTIONAL DATASET COLUMNS (default is True for all, some functions may not operate without some of these columns existing within the calculated dataset(s))

- **compute_job_category_order** : boolean

  True - generate a "cat_order" job rank metric
  False - omit the "cat_order" job rank metric

- **add_eg_col** : boolean

  True - Add an "eg" column to the dataset containing an employee group code for each employee for every month (integer)
  False - Do not include an "eg" column within the calculated dataset.

- **add_retdate_col** : boolean

  True - Add a "retdate" column to the dataset containing employee retirement dates
  False - Do not include a "retdate" column within the calculated dataset.

- **add_doh_col** : boolean

  True - Add a "doh" column to the dataset containing employee date of hire
  False - Do not include a "doh" column within the calculated dataset.

- **add_ldate_col** : boolean

True - Add an "ldate" column to the dataset containing employee longevity date

False - Do not include an "ldate" column within the calculated dataset.

- **add_lname_col** : boolean

  True - Add an "lname" column to the dataset containing employee last name

  False - Do not include an "lname" column within the calculated dataset.

- **add_line_col** : boolean

  True - Add a "line" column to the dataset indicating employee active status (1 is active, 0 is inactive)

  False - Do not include a "line" column within the calculated dataset.

- **add_sg_col** : boolean

  True - Add a "sg" column to the dataset indicating employees with special pre-existing job rights ("special group", marked with a 1 vs 0)

  False - Do not include a "sg" column within the calculated dataset.

- **add_ret_mark** : boolean

  True - Add a "ret_mark" column to the dataset and mark an employee's retirement month with a 1. This was developed to be used when the retirement age changes within the model, but may be used as a final month flag as a convenience.

  False - Skip the "ret_mark" column

- **save_to_pickle** : boolean

  True - save calculated program datasets to disk

  False - calculated datasets will not be written to disk

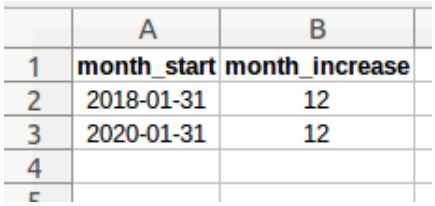
attribute_dict

	A	B
1	**col_name**	**col_description**
2	mnum	month number
3	idx	index
4	empkey	employee number
5	mth_pcnt	month pay percentage
6	date	date
7	year	contract year
8	pay_raise	pay rate multiplier
9	fur	furlough
10	eg	employee group
11	retdate	retirement date
12	doh	date of hire
13	ldate	longevity date
14	lname	last name
15	line	active
16	sg	special group
17	ret_mark	retirement month
18	scale	longevity pay scale
19	s_lmonths	starting longevity – months
20	age	age
21	new_order	editor order
22	orig_job	original job
23	jnum	job level
24	fbff	full bump full flush
25	snum	seniority number
26	spcnt	seniority list percentage
27	lnum	list number (includes fur)
28	lspcnt	list percent (includes fur)
29	rank_in_job	rank in job level
30	job_count	job level count
31	jobp	percentage within job level
32	cat_order	global job ranking
33	mlong	longevity (months)
34	ylong	longevity (years)
35	mpay	monthly pay
36	cpay	cumulative career pay
37		
38		
39		

Fig. 11: The "attribute_dict" sheet is the source for the *dict_attr.pkl* file, which is a column name to column description dictionary used for plotting labels.

The values on this worksheet are not normally changed unless the user desires to change the description associated with an attribute. Changes here will be reflected in certain chart titles and labels.

ret_incr



	A	B
1	month_start	month_increase
2	2018-01-31	12
3	2020-01-31	12
4		

Fig. 12: specify retirement age increase(s), by date and increase in months

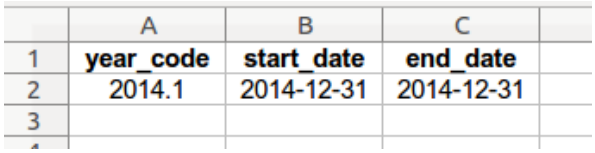
- **month_start**: month end date of month to begin retirement age increase (integer)
- **month_increase**: increase in retirement age in months (integer)

If the user elects to include an increase in the data model retirement age, this worksheet will provide the program with the necessary inputs. The user simply adds a row for each planned age increase with the month end date to begin the new retirement age, along with the number of months to increase the age.

The program transforms the worksheet data into a tuple of tuples for program consumption. The example data above would be stored within the settings dictionary as follows:

```
{'ret_incr': (('2018-01-31', 12), ('2020-01-31', 12))}
```

pay_exceptions



	A	B	C
1	year_code	start_date	end_date
2	2014.1	2014-12-31	2014-12-31
3			

Fig. 13: designate pay_exception periods with separate line entries

Pay rate change periods that do not occur on a calendar year basis are entered on this worksheet. There is no limit to the number of periods and the duration of any period may be set to any monthly time span. "year_code" entries in the first column must correspond to a rate schedule in the *pay_tables.xlsx* "rates" worksheet, under the "year" header. Additional pay exceptions may be designated simply by adding another row of information.

If the case study compensation data does not contain any pay exceptions, enter "no" in column A beneath the "year_code" header (retain the *pay_exceptions* worksheet for proper program operation).

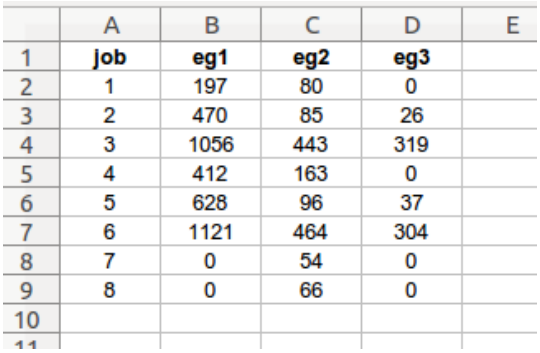
If the pay exception period is effective for only one month, enter that month in both the "start_date" and "end_date" columns, as shown in the example above.

The program will convert each row of worksheet information into a Python dictionary, using the "year_code" column as keys and a list of the "start_date" and "end_date" date values as values. The worksheet example above would be stored by the program as such:

```
{2014.1: [Timestamp('2014-12-31 00:00:00'),
          Timestamp('2014-12-31 00:00:00')]}
```

The pay exception information is used by seniority_list during the monthly compensation index construction process.

job_counts



	A	B	C	D	E
1	**job**	**eg1**	**eg2**	**eg3**	
2	1	197	80	0	
3	2	470	85	26	
4	3	1056	443	319	
5	4	412	163	0	
6	5	628	96	37	
7	6	1121	464	304	
8	7	0	54	0	
9	8	0	66	0	
10					

Fig. 14: define the starting number of jobs in each level, by employee group

- **job**: basic job level number codes (integer)
- **eg<n>**: job counts in each basic job level category for each employee group (integer)

The "job_counts" worksheet provides an accounting of the number of basic jobs available within each job level for each employee group at the starting date of the data model.

Basic counts will be converted to enhanced counts automatically if the "enhanced_jobs" input ("scalars" worksheet) is set to "True".

In the example above, three work group counts are indicated. The user should create one column of job counts for each employee group involved in the integration. Each column header containing counts must begin with the letters "eg" (employee group) and be in the order of the employee codes asssigned in *master.xlsx*, from left to right. The program looks specifically for job counts in columns which begin with "eg". If

this employee group does not have any jobs at a certain level, use a zero as a placeholder.

The job counts are stored in the settings dictionary as a list of lists:

```
{'eg_counts': [[197, 470, 1056, 412, 628, 1121, 0, 0],
               [80, 85, 443, 163, 96, 464, 54, 66],
               [0, 26, 319, 0, 37, 304, 0, 0]]}
```

This input is especially important to the central job assignment routine.

job_changes

	A	B	C	D	E	F	G	H
1	**job**	**month_start**	**month_end**	**total_change**	**eg1**	**eg2**	**eg3**	
2	1	35	64	43	40	3	0	
3	4	35	64	72	66	6	0	
4	2	1	52	-408	-377	-23	-8	
5	5	1	52	-510	-474	-26	-10	
6	3	1	61	411	376	26	9	
7	6	1	61	411	376	26	9	
8								
9								

Fig. 15: Increase or decrease the number of jobs at any job level, for any time period

- **job**: job level affected (integer)
- **start_month**: month in which job change begins (integer, counted from beginning of data model)
- **end_month**: month in which job change ends (integer, counted from beginning of data model)
- **total_change**: total change in job count (positive or negative integer)
- **eg<n>**: job change counts in each employee group, total for the employee groups must equal the total change count. Employee count column headers must begin with "eg", and be in ascending order from left to right. (integer)

This worksheet provides information to the program if the user elects to model a scenario where the number of jobs available in one or more job levels changes over time.

The job changes always refer to the basic job levels. If an enhanced model is selected, the job changes will be converted automatically to the proper number of enhanced jobs in each level.

Each job change event row will be programmatically converted to a list with the following format for use within seniority_list:

[job level affected, [start and end month], total change, [standalone allocation]]

Example (derived from first job change event row in screenshot):

```
[1, [35, 64], 43, [40, 3, 0]]
```

The list above has been set to indicate a change in the number of jobs available at job level 1, starting in month 35 and ending at month 64, increasing 43 jobs, with separate employee group allocation set as 40 to group 1, 3 to group 2, and none to group 3. The program will use an algorithm to apply an even, incremental increase in jobs at level 1 over the number of months specified and will use the allocation schedule to apply the increase to the separate groups until the job changes occur after an implementation date.

Job changes may be an increase or decrease (positive or negative integer) and different job changes may occur at the same time.

Each job change list becomes an element within a list of all the job change events when processed by the program.

The number of "eg" columns, or employee group allocation columns, must match the actual number of employee groups. For example, if the case study only includes 2 employee groups, there would be no "eg3" column in the worksheet. Each column header containing job change counts for a specific employee group must begin with the letters "eg" and be in the order of the employee codes asssigned in *master.xlsx*, from left to right. The program looks specifically for employee group job change counts in columns which begin with "eg".

recall

	A	B	C	D	E	F	G
1	**total_monthly**	**eg1**	**eg2**	**eg3**	**month_start**	**month_end**	
2	8	6	0	2	50	75	
3	10	10	0	0	75	150	
4							
5							

Fig. 16: Recall schedule information for one or more recall periods

- **total_monthly**: total monthly recall count (integer)
- **eg<n>**: monthly recall counts for each employee group. The total for the employee groups must equal the total monthly recall count. Employee count column headers must begin with "eg". (integer)
- **month_start**: month in which recall begins (integer, counted from beginning of data model)

- **month_end**: month in which recall ends (integer, counted from beginning of data model)

This worksheet provides information to the program if the user elects to model a scenario where furloughed employees are recalled over time.

Each recall event schedule (row) will be programmatically converted to a list with the following format for use within seniority_list:

[total monthly_recall_count, eg recall allocation, start_month, end_month]

Example (derived from first recall event row in screenshot):

```
[8, [6, 0, 2], 50, 75]
```

The list above has been set to indicate a recall of 8 employees per month starting in month 50 and ending in month 75. The employee group allocation is 6 per month for employee group 1, and 2 per month for employee group 3. The separate group recall will apply until an implementation date. After an implementation date, the monthly recall amount will be applied to all furloughed employees, according to a recall priority function input. The default is to recall employees by rank within the proposed integrated seniority list (most senior first), but may be set to another method within the *mark_for_recall* function (*functions* module).

Each recall schedule list becomes an element within a list of all the recall events when processed by the program.

Recall schedules are ignored once all furloughees have returned to work. There may be more than one recall schedule and recall schedules may overlap.

The number of "eg" columns, or employee group allocation columns, must match the actual number of employee groups. For example, if the case study only includes 2 employee groups, there would be no "eg3" column in the worksheet. Each column header containing recall counts for a specific employee group must begin with the letters "eg" and be in the order of the employee codes asssigned in *master.xlsx*, from left to right. The program looks specifically for employee group monthly recall counts in columns which begin with "eg".

prex

	A	B	C	D	E	F
1	**eg**	**job**	**count**	**month_start**	**month_end**	
2	1	2	43	0	67	
3	1	3	130	0	67	
4	1	5	43	0	67	
5	1	6	130	0	67	
6						
7						

Fig. 17: pre-existing job rights information, by employee group, basic job, allotment, and time frame

- **eg**: employee group code (integer)
- **job**: basic job level (integer)
- **count**: job allocation count (integer)
- **month_start**: month in which special job right begins (integer, counted from beginning of data model)
- **month_end**: month in which special job right ends (integer, counted from beginning of data model)

This worksheet provides information to the program when modeling a scenario which contains special job guarantees to a subset of employees within one or more of the merging employee groups. These pre-existing job rights will be incorporated within both standalone and integrated models. This type of job right would likely be part of a previous seniority integration award or settlement.

The terms "prex" (pre-existing condition) and "sg" (special group) are used interchangeably.

The job rights always refer to the basic job levels. If an enhanced model is selected, the job rights will be converted automatically to the proper number of enhanced jobs in each level. See the "sg_dist" definition in the "scalars" worksheet discussion for guidance on controlling how the job rights are distributed between basic and enhanced job levels.

Each pre-existing job rights schedule (row) will be programmatically converted to a list with the following format for use within seniority_list:

[eg, jnum, count, start_month, end_month]

Example:

```
[1, 5, 43, 0, 67]
```

The list above has been set to permit employees delineated as having special rights from employee group 1 to be assigned up to 43 positions in job level 5, starting with month 0 and continuing to month 67.

Employees with special job rights must be marked with a 1 in the input *master.xlsx* "sg" column. This marks employees within an employee group as those employees subject to special job assignment rights.

ratio_cond

	A	B	C	D	E	F	G	H	I
1	**basic_job**	**group1**	**group2**	**weight1**	**weight2**	**month_start**	**month_end**	**snapshot**	
2	**1**	1	2,3	2.48	1	34	72	FALSE	
3	4	1	2,3	2.46	1	34	72	FALSE	
4									

Fig. 18: designate ratio condition employee groups, basic job(s) affected, ratio weightings, effective months, and snapshot option

- **basic_job**: (integer) basic job level
- **group<n>** columns: (integer or comma separated integers) for each "group" column, designate a ratio group by employee group code(s). If more than one employee group will make up the ratio group, enter both employee group codes, separated by a comma, such as "2,3".
- **weight<n>** columns: (integer or float) for each "weight" column, designate a value to be used as a ratio weighting. Any number value is valid. Weights will correspond to group codes and must be in same order as the group codes column-wise, from left to right.
- **month_start**: (integer) model month in which to begin condition (from starting date, inclusive)
- **month_end**: (integer) model month in which to begin condition (from starting date, inclusive)
- **snapshot**: (boolean ["TRUE", "FALSE"]) capture the existing job count ratio which exists at the "month_start" data model month (ignores the weight column inputs).

This worksheet provides information to the program if the user elects to model a scenario which contains a prospective job assignment condition based maintaining a ratio of jobs in a specified job level(s) between one employee group and one or more other employee group(s).

The basic_job column always refers to the basic job levels. If an enhanced model is selected, the job levels will be converted automatically to include enhanced jobs

associated with the basic job levels or as otherwise directed by the *convert* function from the *converter* module. See the "ratio_dist" definition in the "scalars" worksheet discussion for guidance on controlling how the job levels are determined when using an enhanced job level model.

The user may add or delete "group" columns and "weight" columns as required for the case study, as long as this is done in corresponding pairs. The program will look for and match ratio groups ("group" columns) with ratio weightings ("weight" columns) by column order. Group and weight columns are identified by the program when column headers begin with "group" and "weight". It is acceptable to have entires of zero ("0") in a group column if necessary (with mergers involving more than two employee groups) to ignore that group column within a row entry, as in the following example:

	A	B	C	D	E	F	G	H	I	J
1	basic_job	group1	group2	group3	weight1	weight2	weight3	month_start	month_end	snapshot
2	1	1	2,3	0	2.48	1	0	34	72	FALSE
3	4	1	2	3	2.1	1	0.8	34	72	FALSE
4										

Fig. 19: group 3 is not included in the basic job level 1 ratio condition and any weighting in the "weight3" column is ignored

A zero entry in a weight column corresponding to a valid non-zero group column will be interpreted by the conditional job assignment routine to mean that no new job openings should be assigned to that group(s). No bump, no flush rules will protect employees from the affected group from being displaced from current job levels. Positions available each month will be assigned to the appropriate group(s) so as to get as close as possible to the desired job ratio(s) over time.

The ratio_cond worksheet data is programmatically combined to form a dictionary for program operation. The dictionary is used as an argument for the *assign_cond_ratio* function. Jobs are assigned to the ratio groups according to the corresponding weightings beginning with the month_start and continuing until the ending date.

The function may be used in conjunction with the *set_snapshot_weights* function to capture an existing ratio of jobs between ratio groups as they exist at the "month_start" month snapshot. The snapshot weightings will be used during the condition period only ("month_start" to "month_end"). The snapshot option is selected by a "TRUE" cell input within the "snapshot" column.

The function will adjust the job assignment quota counts if the number of jobs available within job levels changes from month to month.

The dictionary is formed with the following format:

job level: [(employee groups), (weightings), start_month, end_month}]

Example (from first row of top example above, converted to enhanced job levels):

```
{1: [([1], [2, 3]), (2.48, 1.0), 34, 72],
 2: [([1], [2, 3]), (2.48, 1.0), 34, 72]}
```

The dictionary input above has been set to distribute job assignments for job levels 1 and 2 between employee group 1 and employee groups 2 and 3 (combined) at a ratio of 2.48:1, in data model months 34 through 72.

ratio_count_capped_cond

	A	B	C	D	E	F	G	H	I	J	K	L
1	basic_job	group1	group2	group3	weight1	weight2	weight3	cap	month_start	month_end	snapshot	
2	1	1	2	0	2.48	1	0	318	34	94	FALSE	
3	4	1	2	0	2.46	1	0	580	34	94	FALSE	
4												
5												

Fig. 20: define count-ratio condition employee groups, basic job level(s), ratio weightings, cap(s), and effective months

- **basic_job**: (integer) basic job level
- **group<n>** columns: (integer or comma separated integers) for each "group" column, designate a ratio group by employee group code(s). If more than one employee group will make up the ratio group, enter both employee group codes, separated by a comma, such as "2,3".
- **weight<n>** columns: (integer or float) for each "weight" column, designate a value to be used as a ratio weighting. Any number value is valid. Weights will correspond to group codes and must be in same order as the group codes column-wise, from left to right.
- **cap**: (integer) the maximum total number of jobs to distribute among the applicable employee groups
- **month_start**: (integer) model month in which to begin condition (from starting date, inclusive)
- **month_end**: (integer) model month in which to begin condition (from starting date, inclusive)
- **snapshot**: (boolean ["TRUE", "FALSE"]) capture the existing job count ratio which exists at the "month_start" data model month (ignores the weight column inputs). This ratio will be used for job assignments (up to the applicable job count cap).

This worksheet provides information to the program if the user elects to model a scenario which contains a prospective job assignment condition based on maintaining a ratio of jobs in a specified job level(s) count between designated employee groups. The condition is not applied to job assignments above the job count cap.

The job column always refers to the basic job levels. If an enhanced model is selected, the job levels will be converted automatically to include enhanced jobs associated with the basic job levels or as otherwise directed by the *convert* function from the *converter* module. See the "quota_dist" definition in the "scalars" worksheet discussion for guidance on controlling how the job levels are determined when using an enhanced job level model.

This data is used with the *assign_cond_ratio_capped* function, which assigns a limited pool of jobs from a selected job level between one group and another group according to a set ratio. There must be the same number of "weight" columns as "group" columns. Entries of "0" in "group" and "weight" columns are acceptable and will be ignored during calculations. The user may add or delete "group" columns and "weight" columns as required for the case study, as long as this is done in corresponding pairs. The program will look for and match ratio groups ("group" columns) with ratio weightings ("weight" columns) by column order. Group and weight columns are identified by the program when column headers begin with "group" and "weight".

Alternatively, a minimum job count may be assigned to one employee group only, by listing the employee group in a "group" column, assigning a corresponding positive weighting (any other weightings should be zero), and assigning a "cap" as the minimum job allocation for the employee group.

As with the ratio condition above, the function may be used in conjunction with the *set_snapshot_weights* function to capture an existing ratio of jobs between ratio groups as they exist at the "month_start" month snapshot. The snapshot weightings will be used during the condition period only ("month_start" to "month_end"). The snapshot option is selected by a "TRUE" cell input within the "snapshot" column.

In the case where there are less jobs than the cap amount, the actual number of jobs available will be distributed according to the weightings.

The data is programmatically converted to a dictionary with the following format:

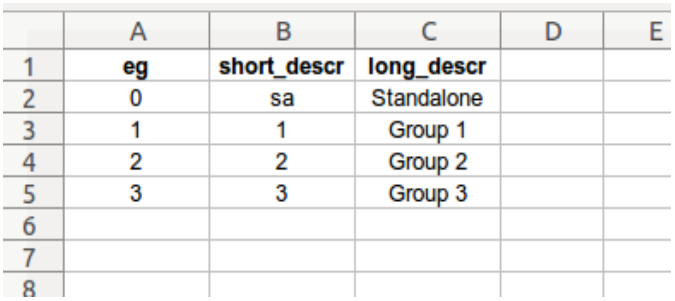
job level: [(employee groups), (weightings), cap, start_month, end_month]

Example (from first row of example above, converted to enhanced job levels):

```
{1: [([1], [2]), (2.48, 1.0), 191, 34, 94],
 2: [([1], [2]), (2.48, 1.0), 127, 34, 94]}
```

The dictionary input above has been set to distribute job assignments for job levels 1 and 2 between employee group 1 and employee groups 2 and 3 (combined) at a ratio of 2.48:1, in data model months 34 through 72. The conditional assignment will operate for up to the first 191 jobs in job level 1, and up to the first 127 jobs in job level 2.

proposal_dictionary



	A	B	C	D	E
1	**eg**	**short_descr**	**long_descr**		
2	0	sa	Standalone		
3	1	1	Group 1		
4	2	2	Group 2		
5	3	3	Group 3		
6					
7					
8					

Fig. 21: proposal number to description dictionary data

- **eg**: employee group code, insert a zero to represent standalone data for plotting (integer)
- **short_descr**: short descriptive labels for chart labels and titles
- **long_descr**: longer descriptive labels for chart labels and titles

This worksheet provides information to the program which is used for some charting labels and titles relating to employee groups. It would be more correct to think of this worksheet as an employee group dictionary. Future coding work will change the name of this worksheet.

The data contained on this worksheet is transformed into two dictionaries. The "eg" column will be the integer keys in both, and the other columns will each make up the value items in separate dictionaries.

The "eg" column should contain the employee group codes in low to high sequential order, with the addition of a zero at the beginning to represent standalone data plotting.

The brief descriptions in the "short_descr" column will be the values in the "p_dict" dictionary:

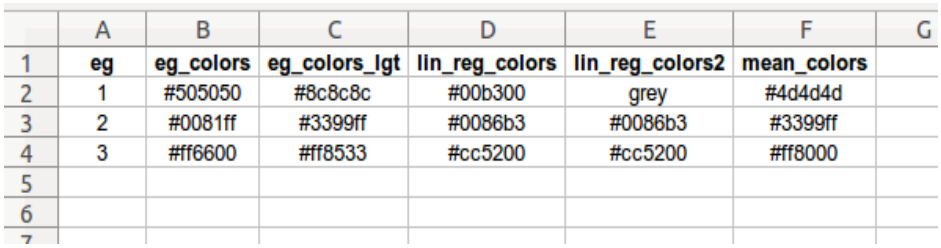
```
{0: 'sa',
 1: '1',
 2: '2',
 3: '3'}
```

The slightly longer descriptions in the "long_descr" column will be the values in the "p_dict_verbose" dictionary:

```
{0: 'Standalone',
 1: 'Group 1',
 2: 'Group 2',
 3: 'Group 3'}
```

Both dictionaries are stored in the settings dictionary.

eg_colors



	A	B	C	D	E	F	G
1	**eg**	**eg_colors**	**eg_colors_lgt**	**lin_reg_colors**	**lin_reg_colors2**	**mean_colors**	
2	1	#505050	#8c8c8c	#00b300	grey	#4d4d4d	
3	2	#0081ff	#3399ff	#0086b3	#0086b3	#3399ff	
4	3	#ff6600	#ff8533	#cc5200	#cc5200	#ff8000	
5							
6							

Fig. 22: user-defined charting colors corresponding to the employee groups

- **eg**: employee group codes, one row for each group (integer)
- **eg_colors**: chart colors to use when plotting values representing the employee groups (color values)
- **eg_colors_lgt**: alternate lighter chart colors to use when plotting values representing the employee groups (color values)
- **lin_reg_colors**: sample set of colors which may be used with the editor tool when plotting polynomial regression lines (color values)
- **lin_reg_colors2**: sample set of colors which may be used with the editor tool when plotting polynomial regression lines (color values)
- **mean_colors**: sample set of colors which may be used with the editor tool when plotting average value lines (color values)

This worksheet provides lists of colors which are used to represent the employee groups or values associated with the employee groups when creating charts.

The lists are arranged vertically on the worksheet. The rows represent the employee groups and must be in employee group code sequential order, lowest to highest. The program will arrange each worksheet column into a dictionary with the color name (worksheet column header) as the key, and the color values in each column as the value (as a list).

Example (for the 3 employee group example in the image above, "eg_colors" column):

```
{'eg_colors': ['#505050', '#0081ff', '#ff6600']}
```

The rest of the color lists would be treated similarly. The output dictionaries from this worksheet are added to the color dictionary file, *dict_color.pkl*.

basic_job_colors

	A	B	C	D	E	F
1	job	red	green	blue	alpha	
2	1	0.65	0.8	0.89	1	
3	2	0.14	0.48	0.7	1	
4	3	0.66	0.85	0.51	1	
5	4	0.28	0.62	0.21	1	
6	5	0.97	0.53	0.53	1	
7	6	0.9	0.21	0.16	1	
8	7	0.99	0.79	0.49	1	
9	8	0.94	0.54	0.2	1	
10	9	0.5	0.5	0.5	1	
11						
12						
13						

Fig. 23: user-defined job level charting colors

- **job**: basic job level (integer)
- **red**: float value from 0.0 to 1.0
- **green**: float value from 0.0 to 1.0
- **blue**: float value from 0.0 to 1.0
- **alpha**: float value from 0.0 to 1.0

This worksheet provides user-defined basic job colors in red, green, blue, alpha float format.

While the *make_color_list* plotting function provides many lists of colors for plotting, the user may wish to define specific color values to represent the various job levels within the data model.

List of color codes used by various plotting functions to represent job levels. The example color codes are in [red, green, blue, and alpha] float format, but color hex codes or color names may be used as well. The rgba color codes may be derived from the *make_color_list* plotting function, and then copied into the worksheet cells.

The length of these lists is: job level count + 1. The last color value will be used to represent furlough. The example below shows a color list for a case study with 8 basic job levels.

If the "scalars" worksheet "enhanced_jobs" input is False, the program will store the information on this worksheet in the color dictionary as a list of color lists:

```
{'job_colors': [[0.65, 0.8, 0.89, 1.0],
                [0.14, 0.48, 0.7, 1.0],
                [0.66, 0.85, 0.51, 1.0],
```

(continues on next page)

(continued from previous page)

```
                [0.28, 0.62, 0.21, 1.0],
                [0.97, 0.53, 0.53, 1.0],
                [0.9, 0.21, 0.16, 1.0],
                [0.99, 0.79, 0.49, 1.0],
                [0.94, 0.54, 0.2, 1.0],
                [0.5, 0.5, 0.5, 1.0]]}
```

enhanced_job_colors

	A	B	C	D	E	F
1	**job**	**red**	**green**	**blue**	**alpha**	
2	1	0.65	0.81	0.89	1	
3	2	0.31	0.59	0.77	1	
4	3	0.19	0.39	0.7	1	
5	4	0.66	0.85	0.55	1	
6	5	0.41	0.73	0.32	1	
7	6	0.22	0.6	0.23	1	
8	7	0.93	0.61	0.57	1	
9	8	0.93	0.32	0.32	1	
10	9	0.75	0.1	0.1	1	
11	10	0.99	0.79	0.49	1	
12	11	0.95	0.65	0.19	1	
13	12	0.82	0.42	0.12	1	
14	13	0.82	0.67	0.71	1	
15	14	0.6	0.47	0.72	1	
16	15	0.5	0.35	0.6	1	
17	16	0.9	0.87	0.6	1	
18	17	0.5	0.5	0.5	1	
19						
20						
21						

Fig. 24: user-defined job level charting colors

- **job**: enhanced job level (integer)
- **red**: float value from 0.0 to 1.0
- **green**: float value from 0.0 to 1.0
- **blue**: float value from 0.0 to 1.0
- **alpha**: float value from 0.0 to 1.0

The description for this worksheet is almost identical to the "basic_job_colors" guide above. However, there will normally be twice as many job colors (one for each enhanced job level) plus a furlough color value.

If the "scalars" worksheet "enhanced_jobs" input is True, the program will store the information on this worksheet in the color dictionary as a list of color lists:

```
{'job_colors': [[0.65, 0.81, 0.89, 1.0],
                [0.31, 0.59, 0.77, 1.0],
                [0.19, 0.39, 0.7, 1.0],
                [0.66, 0.85, 0.55, 1.0],
                [0.41, 0.73, 0.32, 1.0],
                [0.22, 0.6, 0.23, 1.0],
                [0.93, 0.61, 0.57, 1.0],
                [0.93, 0.32, 0.32, 1.0],
                [0.75, 0.1, 0.1, 1.0],
                [0.99, 0.79, 0.49, 1.0],
                [0.95, 0.65, 0.19, 1.0],
                [0.82, 0.42, 0.12, 1.0],
                [0.82, 0.67, 0.71, 1.0],
                [0.6, 0.47, 0.72, 1.0],
                [0.5, 0.35, 0.6, 1.0],
                [0.9, 0.87, 0.6, 1.0],
                [0.5, 0.5, 0.5, 1.0]]}
```

6.5 anonymizing input data

seniority_list includes several functions which are able to modify worksheet data within the excel input files, with focus on the *master.xlsx* and *pay_tables.xlsx* files. These operations are helpful when the user wishes to publicly share data or analysis which could otherwise be considered confidential. For example, these functions can quickly produce substitute names and employee numbers for all employees. Subsequent datasets and chart analysis will reference the modified input data.

Please see the "program demonstration" section within the user guide for more information.

CHAPTER

SEVEN

QUICK REPORT

7.1 general

seniority_list is able to rapidly generate statistical summary reports for all integrated list outcomes. This capability is provided through the functions within the *reports* module. Existing datasets are automatically recognized and loaded internally by the *reports* module through use of the *load_datasets* function from the *functions* module.

This program feature offers insight into significant outcome metrics prior to more detailed analysis using the built-in plotting functions or other techniques. The user may also find this functionality helpful for testing or validation following list modification with the editor tool.

Quick reporting trades the extensive customization and analytical resolution offered with the built-in plotting functions for a fast outline reporting of a limited set of predetermined attribute measurements.

Report output is stored within the **reports/<case name>** folder. The report information is presented as two excel spreadsheets and numerous chart images. Summary reports may be shared with others by simply copying and disributing the **reports/<case name>** folder.

A more recent addition to the reporting capability of senior_list is time-in-job comparisons, discussed in the section below. The file output from the *job_diff_to_excel* function is stored within the **reports/<case name>/by_employee** folder.

7.1.1 computed statistics

The *reports* module functions generate a collection of summary data consisting of average attribute values for each employee group over the life of the data model.

The statistics are computed in two ways:

- values for employees at retirement only
- annual values for all employees

The values are calculated from groupings, or bins, of certain categorical data:

- longevity or date of hire year
- starting job level
- population quantile membership (within each employee group), with two subsets:
  - initial list quantile
  - monthly running quantile

Within the categorical groupings, the routines measure a default set of attributes:

- seniority list percentage ("spcnt")
- seniority number ("snum")
- job value rank ("cat_order")
- percentage within job level ("jobp")
- career earnings ("cpay")

7.1.2 grouping method definitions

- **longevity year** or **date of hire year**

  Employees may be grouped and compared by the longevity year or date of hire year (selectable as a function input). Grouping in this fashion permits future year comparison of employees from each employee group from the same hire year or with the equivalent longevity year.

- **quantile**

  The default number of quantiles used for membership grouping is 10, meaning an employee at 5% on the list would be a member of quantile 1, 25% would be quantile 3, etc. The number of quantiles may be modified through a function input.

  - initial list quantile

Employees are assigned to a quantile group based on separate employee group seniority list percentage postition at the merger date. Initial list quantile members are tracked throughout the data model time period, for each employee group separately. This tracking provides a comparative attribute value analysis for cohort list percent employees from each group. Using the initial quantile membership will allow comparing employees from separate groups in future years who were initially members of the same relative quantile.

- monthly running quantile

  For each month of the data model, employees are assigned to a quantile group based on separate employee group seniority list percentage postition. Running quantile members are tracked throughout the data model time period, for each employee group separately. This tracking provides a comparative attribute value analysis, averaged on an annual basis, for cohort list percent employees within each group. This style of analysis will show, for example, the average job level held by the 3rd quantile of employees within each group for the year 2022.

- **starting job level**

  Employees are assigned to a initial job level group based on separate employee group job level postitions at the merger date. Initial list job level members are tracked throughout the data model time period, for each employee group separately. This tracking provides a comparative attribute value analysis for cohort initial job level employees from each group. Using the initial job level membership will allow comparing employees from separate groups in future years who were initially members of the same relative job group.

7.1.3 excel files

The *stats_to_excel* function stores the statistical data in two excel workbooks:

- *ret_stats.xlsx*
- *annual_stats.xlsx*

Each workbook contains many worksheets. Each worksheet contains results for a specific calculated dataset with a certain type of grouping applied.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1			spcnt			snum			cat_order			jobp			cpay			
2		eg	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	
3	ldate	retdate																
92		2027	0.0%	5.5%		2	335		2	335		1.01	3.16		2399	2070		
93		2028	0.0%			1			1			1.01			2556			
94	1986	2014	12.8%	39.9%	7.9%	495	578	55	667	2532	829	4.03	6.86	4.19	66	61	74	
95		2015	14.0%	38.2%		534	553		757	2421		4.16	6.67		178	131		
96		2016	12.5%	42.8%	6.4%	485	1644	44	698	2081	700	4.16	5.96	4.14	355	301	268	
97		2017	7.9%	56.0%	20.4%	465	3284	1195	548	2162	1203	4.08	6.13	4.56	519	430	558	
98		2018	6.9%	52.0%	20.5%	410	3087	1220	485	2199	715	4.04	6.04	3.59	652	580	655	
99		2019	5.9%	47.5%	17.7%	357	2871	1069	440	2275	237	3.83	6.12	2.36	806	731	840	
100		2020	4.8%	43.6%	15.1%	289	2631	911	346	2388	211	3.23	6.43	2.16	987	855	1052	
101		2021	3.6%	39.3%	14.0%	220	2372	847	244	2217	188	2.44	6.10	1.98	1145	994	1148	
102		2022	2.7%	35.4%		162	2136		169	2136		1.91	6.08		1324	1131		
103		2023	1.9%	29.0%	8.3%	116	1749	504	116	1749	228	1.61	5.04	2.29	1510	1307	1539	
104		2024	1.1%	24.3%	4.9%	66	1470	297	66	1470	190	1.34	4.72	1.99	1714	1442	1782	
105		2025	0.7%	19.4%	2.9%	44	1169	173	44	1169	173	1.23	4.51	1.90	1883	1616	2065	
106		2026	0.3%	15.2%		21	921		21	921		1.11	4.34		2106	1754		
107		2027	0.2%	11.2%		10	678		10	678		1.05	4.17		2294	1917		
108		2028	0.0%	8.7%		2	527		2	527		1.01	4.07		2498	2052		
109		2029	0.0%			1			1			1.01			2697			
110		2030		3.7%			224			224			2.25			2379		
111	1987	2014	19.1%	49.9%		735	723		1075	3072		4.39	8.30		74	63		
112		2015	19.4%	45.6%	10.0%	738	661	69	1101	2839	880	4.44	7.58	4.26	217	170	181	
113		2016	17.0%	44.6%	9.7%	752	645	66	998	2772	823	4.39	7.40	4.25	361	257	330	
114		2017	12.2%	73.4%	29.9%	717	4309	1758	778	2745	1383	4.25	7.36	4.69	508	405	467	
115		2018	10.8%	70.7%	24.5%	639	4196	1450	697	2789	1422	4.20	7.37	4.71	638	533	627	
116		2019	9.7%	63.8%	21.9%	588	3852	1322	658	2782	303	4.16	7.23	2.86	796	672	841	
117		2020	7.8%	61.8%	19.2%	472	3730	1159	535	2872	302	4.07	7.43	2.86	961	800	1018	
118		2021	6.2%	56.9%	16.9%	375	3440	1021	419	2882	307	3.81	7.45	2.90	1129	933	1176	
119		2022	4.7%	50.1%	14.5%	285	3024	873	309	2789	312	2.94	7.25	2.93	1283	1095	1320	
120		2023	3.3%	46.6%	10.6%	199	2813	641	207	2758	315	2.15	7.24	2.96	1464	1223	1507	
121		2024	2.2%	40.4%	6.9%	136	2439	416	136	2439	314	1.71	6.56	2.95	1637	1355	1632	
122		2025	1.5%	34.9%	6.1%	90	2108	369	90	2108	320	1.47	5.96	3.00	1844	1486	1829	
123		2026	0.8%	25.9%	2.4%	47	1562	144	47	1562	144	1.25	4.96	1.75	2065	1666	2067	
124		2027	0.4%	20.7%	1.3%	21	1252	81	21	1252	81	1.11	4.57	1.42	2248	1815	2281	
125		2028	0.1%	21.3%	1.2%	8	1287	72	8	1287	72	1.04	4.59	1.38	2435	1946	2320	
126	1988	2014	23.6%	59.1%	13.7%	906	857	94	1368	3777	1052	4.64	10.09	4.38	111	60	107	
127		2015	23.8%	53.8%	13.9%	906	780	95	1379	3306	1029	4.66	9.18	4.39	216	141	217	
128		2016	24.8%	51.7%		934	749		1435	3138		4.72	8.65		314	224		
129		2017	15.0%		32.3%	880		1896	928		1429	4.36		4.72	476		554	
130		2018	14.2%	81.2%		852	4796		892	2076		4.22	8.18		670	476		

Fig. 1: example worksheet from ret_stats workbook[50]

ret_stats.xlsx workbook

The image below is the same worksheet as the example above, with the addition of some formatting for descriptive clarity. The yellow header row contains the measured attributes, and the blue row just below contains employee group codes. The peach-colored column contains longevity year information, while the green column holds retirement year data.

This worksheet reveals average retirement attribute values for equivalent longevity year employees from each employee group. The red-boxed area shows average attribute values for employees with a 1986 longevity year retiring in year 2021. In this example, using the columns under the "spcnt" header, employees from group 2 with a longevity year of 1986 retiring in

[50] http://rubydatasystems.com/reports.html#reports.stats_to_excel

2021 will finish at an average of 39.3% on the integrated list. This compares to 3.6% and 14% for groups 1 and 3 respectively.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1			spcnt			snum			cat_order			jobp			cpay			
2		eg	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	
3	ldate	retdate																
93		2027	0.0%	5.5%		2	335		2	335		1.01	3.16		2399	2070		
94		2028	0.0%			1			1			1.01			2556			
95	1986	2014	12.8%	39.9%	7.9%	495	578	55	667	2532	829	4.03	6.86	4.19	66	61	74	
96		2015	14.0%	38.2%		534	553		757	2421		4.16	6.67		178	131		
97		2016	12.5%	42.8%	6.4%	485	1644	44	698	2081	700	4.16	5.96	4.14	355	301	268	
98		2017	7.9%	56.0%	20.4%	465	3284	1195	548	2162	1203	4.08	6.13	4.56	519	430	558	
99		2018	6.9%	52.0%	20.5%	410	3087	1220	485	2199	715	4.04	6.04	3.59	652	580	655	
100		2019	5.9%	47.5%	17.7%	357	2871	1069	440	2275	237	3.83	6.12	2.36	806	731	840	
101		2020	4.8%	43.6%	15.1%	289	2631	911	346	2388	211	3.23	6.43	2.16	987	855	1052	
102		2021	3.6%	39.3%	14.0%	220	2372	847	244	2217	188	2.44	6.10	1.98	1145	994	1148	
103		2022	2.7%	35.4%		162	2136		169	2136		1.91	6.08		1324	1131		
104		2023	1.9%	29.0%	8.3%	116	1749	504	116	1749	228	1.61	5.04	2.29	1510	1307	1539	
105		2024	1.1%	24.3%	4.9%	66	1470	297	66	1470	190	1.34	4.72	1.99	1714	1442	1782	
106		2025	0.7%	19.4%	2.9%	44	1169	173	44	1169	173	1.23	4.51	1.90	1883	1616	2065	
107		2026	0.3%	15.2%		21	921		21	921		1.11	4.34		2106	1754		
108		2027	0.2%	11.2%		10	678		10	678		1.05	4.17		2294	1917		
109		2028	0.0%	8.7%		2	527		2	527		1.01	4.07		2498	2052		
110		2029	0.0%			1			1			1.01			2697			
111		2030		3.7%			224			224			2.25			2379		
112	1987	2014	19.1%	49.9%		735	723		1075	3072		4.39	8.30		74	63		
113		2015	19.4%	45.6%	10.0%	738	661	69	1101	2839	880	4.44	7.58	4.26	217	170	181	
114		2016	17.0%	44.6%	9.7%	752	645	66	998	2772	823	4.39	7.40	4.25	361	257	330	
115		2017	12.2%	73.4%	29.9%	717	4309	1758	778	2745	1383	4.25	7.36	4.69	508	405	467	
116		2018	10.8%	70.7%	24.5%	639	4196	1450	697	2789	1422	4.20	7.37	4.71	638	533	627	
117		2019	9.7%	63.8%	21.9%	588	3852	1322	658	2782	303	4.16	7.23	2.86	796	672	841	
118		2020	7.8%	61.8%	19.2%	472	3730	1159	535	2872	302	4.07	7.43	2.86	961	800	1018	
119		2021	6.2%	56.9%	16.9%	375	3440	1021	419	2882	307	3.81	7.45	2.90	1129	933	1176	
120		2022	4.7%	50.1%	14.5%	285	3024	873	309	2789	312	2.94	7.25	2.93	1283	1095	1320	
121		2023	3.3%	46.6%	10.6%	199	2813	641	207	2758	315	2.15	7.24	2.96	1464	1223	1507	
122		2024	2.2%	40.4%	6.9%	136	2439	416	136	2439	314	1.71	6.56	2.95	1637	1355	1632	
123		2025	1.5%	34.9%	6.1%	90	2108	369	90	2108	320	1.47	5.96	3.00	1844	1486	1829	
124		2026	0.8%	25.9%	2.4%	47	1562	144	47	1562	144	1.25	4.96	1.75	2065	1666	2067	
125		2027	0.4%	20.7%	1.3%	21	1252	81	21	1252	81	1.11	4.57	1.42	2248	1815	2281	
126		2028	0.1%	21.3%	1.2%	8	1287	72	8	1287	72	1.04	4.59	1.38	2435	1946	2320	
127	1988	2014	23.6%	59.1%	13.7%	906	857	94	1368	3777	1052	4.64	10.09	4.38	111	60	107	
128		2015	23.8%	53.8%	13.9%	906	780	95	1379	3306	1029	4.66	9.18	4.39	216	141	217	
129		2016	24.8%	51.7%		934	749		1435	3138		4.72	8.65		314	224		
130		2017	15.0%		32.3%	880		1896	928		1429	4.36		4.72	476		554	
131		2018	14.3%	81.2%		853	4796		892	3076		4.33	8.18		670	476		

stats at retirement for each employee group tracked by longevity year membership

annual_stats.xlsx workbook

The example below has also been formatted as described above.

This worksheet reveals average annual attribute values for employees with the same intial job level at the start of the data model.

In other words, a snapshot of jobs held by all employees is taken at the very beginning of the data model. Employees within each beginning snapshot job level are then tracked throughout the entire data model time period, with average attribute measurements sampled on an annual basis. The measurements are taken for each employee group separately.

In the image below, the red-boxed area contains average attribute values for employees with an initial job level of 6, as measured in year 2022 of the data model. The boxed data under the "spcnt" header indicated that employees from group 2 will be positioned at an average of 56% on the integrated list. This compares to 26.1% and 33% for groups 1 and 3 respectively.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1			spcnt			snum			cat_order			jobp			cpay			ylong			
2		eg	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1.0	2.0	3.0	
3	eg_initQ	date																			
103		2035	0.0%		0.3%	1		15	1		15	1.01		1.08	3537		3402	43.7		37.9	
104	6	2013	60.3%	54.9%	56.9%	2341	797	390	3519	3470	3844	9.46	9.37	10.06	6	6	6	16.6	26.5	15.3	
105		2014	60.2%	53.2%	55.8%	2322	772	383	3506	3321	3728	9.49	9.04	9.93	51	52	49	17.2	27.0	15.8	
106		2015	60.0%	49.4%	53.7%	2285	715	368	3480	3039	3427	9.55	8.22	9.35	146	152	142	18.2	28.0	16.8	
107		2016	57.3%	51.1%	52.0%	2316	1310	837	3439	2824	3060	9.59	7.56	8.49	250	269	251	19.2	29.0	17.8	
108		2017	44.4%	77.9%	54.3%	2604	4575	3187	3262	2836	2886	9.23	7.57	8.02	357	396	369	20.2	30.0	18.8	
109		2018	41.5%	74.2%	50.9%	2462	4403	3018	3035	2880	2748	8.24	7.58	7.48	471	526	495	21.2	31.0	19.8	
110		2019	37.7%	69.6%	46.4%	2279	4203	2804	2674	2942	2730	7.05	7.59	7.22	599	660	628	22.2	32.0	20.8	
111		2020	34.0%	65.6%	42.2%	2054	3965	2546	2223	2958	2612	6.32	7.63	6.93	736	796	766	23.2	33.0	21.8	
112		2021	30.2%	61.1%	37.7%	1821	3692	2276	1873	2965	2283	5.54	7.65	6.44	877	932	905	24.2	34.0	22.8	
113		2022	26.1%	56.0%	33.0%	1578	3385	1991	1580	2978	1994	4.85	7.68	5.87	1030	1068	1044	25.2	34.9	23.8	
114		2023	21.9%	50.5%	28.1%	1322	3053	1695	1322	2899	1695	4.62	7.53	4.89	1187	1201	1195	26.2	35.9	24.8	
115		2024	17.4%	44.3%	22.9%	1054	2674	1386	1054	2668	1386	4.43	7.09	4.66	1343	1338	1354	27.1	36.9	25.8	
116		2025	13.3%	38.2%	18.1%	803	2307	1093	803	2307	1093	4.26	6.30	4.46	1499	1475	1515	28.1	37.8	26.8	
117		2026	9.4%	32.4%	13.5%	569	1955	818	569	1955	818	4.04	5.72	4.27	1655	1618	1671	29.0	38.8	27.8	
118		2027	6.2%	28.7%	9.6%	374	1735	577	374	1735	577	3.35	5.20	4.10	1813	1744	1829	29.8	39.6	28.8	
119		2028	4.0%	23.7%	6.7%	239	1429	406	239	1429	406	2.41	4.69	3.71	1983	1899	1988	30.6	40.6	29.8	
120		2029	2.6%		4.9%	157		295	157		295	1.85		2.82	2166		2155	31.5		30.8	
121		2030	1.7%		3.4%	103		208	103		208	1.53		2.14	2354		2323	32.4		31.7	
122		2031	1.1%		2.3%	63		138	63		138	1.33		1.72	2547		2514	33.3		32.7	
123		2032	0.6%		1.5%	39		91	39		91	1.20		1.47	2747		2708	34.2		33.7	
124		2033	0.4%		0.9%	24		57	24		57	1.13		1.30	2942		2903	35.1		34.7	
125		2034	0.3%		0.6%	16		37	16		37	1.08		1.19	3136		3081	36.0		35.6	
126		2035	0.1%		0.4%	9		22	9		22	1.05		1.11	3335		3245	37.0		36.4	
127		2036	0.1%		0.1%	4		8	4		8	1.02		1.04	3526		3464	37.9		37.5	
128		2037	0.0%		0.1%	3		5	3		5	1.01		1.02	3698		3662	38.8		38.5	
129		2038	0.0%		0.0%	2		3	2		3	1.01		1.02	3928		3802	39.9		39.2	
130		2039	0.0%			1			1			1.01			4120			40.9			
131		2040	0.0%			1			1			1.01			4206			41.3			
132	7	2013	71.2%	64.9%	67.2%	2767	942	461	4050	4178	4303	10.02	10.37	10.47	6	6	6	14.6	22.5	14.3	
133		2014	71.2%	63.2%	66.2%	2747	916	454	4042	4038	4224	10.04	10.29	10.44	49	48	48	15.2	23.0	14.8	
134		2015	71.1%	59.0%	64.0%	2709	855	439	4023	3702	4065	10.07	9.99	10.36	141	138	137	16.2	23.9	15.8	
135		2016	69.1%	60.2%	61.3%	2808	1504	942	3958	3366	3896	10.06	9.38	10.29	243	240	236	17.2	24.9	16.8	
136		2017	58.4%	85.5%	58.9%	3441	5019	3459	3696	3379	3852	9.89	9.24	10.29	347	349	337	18.2	25.9	17.8	

Fig. 2: annual stats tracked by separate employee group initial quantile membership (the red-boxed area shows average attribute values in 2022 for employees who initially belonged to quantile 6)

7.1.4 chart images

The *retirement_charts* and *annual_charts* functions within the *reports* module create many simple statistical charts which are stored as image files within auto-generated folders located within the **reports/<case name>** folder. The chart images are visual representations of the computed statistical data.

With the default function inputs, several directories will be created within the case-specific **reports** folder:

```
.
└── sample3
    ├── annual_charts
    │   ├── ann_charts_init_job
    │   ├── ann_charts_init_qntl
    │   ├── ann_charts_longevity date
    │   └── ann_charts_run_qntl
    └── ret_charts
        ├── ret_charts_init_job
        ├── ret_charts_init_qntl
        ├── ret_charts_longevity date
        └── ret_charts_run_qntl
```

Fig. 3: folders created with the chart creation functions

The total number of chart images stored within the **annual_charts** and **ret_charts** folders may be relatively large. With the "sample3" example case study, a total of over 2,000 chart images are produced!

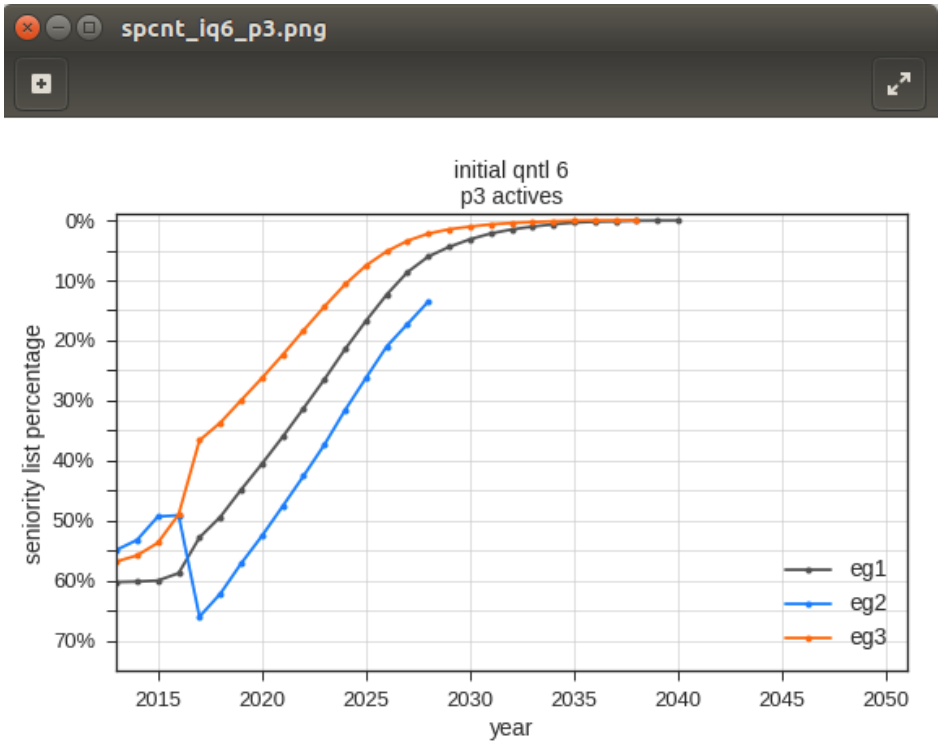


Fig. 4: the *reports* module functions produce numerous charts similar to the chart above

Despite the large quantity, it is does not take long to review the charts using a standard image viewer and left and right arrow keyboard buttons. The routines that produce the charts use the same chart background, scales, and labels for all charts within a category

- only the data lines and the titles change from chart to chart. This setup makes is very easy to see how measurements change between charts.

7.1.5 time-in-job and career pay differential report

The *job_diff_to_excel* report function will generate spreadsheet reports indicating differences in the number of months employees will spend working within the various job levels, and the corresponding difference in career compensation. The user may select any two outcome datasets for comparison.

By default, the generated spreadsheets will be formatted to display employee group color-coded rows and color-coded font to indicate gains or losses in the various job level categories. This formatting is very useful for visual interpretation, but does add time to the process (for reference, the "sample3" example case requires approximately 40 seconds with an i7 linux desktop computer). The formatting may be turned off to create the files more quickly.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	empkey	order	lname	ldate	retdate	eg	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	cpay_diff
3547	10011018	3546	xuxay	1998-12-18	2022-01-08	1						17	-1	-2	-3	-11								36,069
3548	10014704	3547	lasejob	1998-12-18	2023-08-17	1				11	2	7	-4	-2	-2	-12								56,989
3549	20011200	3548	kubie	1986-03-22	2019-02-14	2																		0
3550	30010422	3549	fooosaa	1998-07-28	2029-02-23	3		2	10	-20	-2	10												-4,876
3551	10012320	3550	cigaluo	1998-12-17	2027-11-28	1				17	1	2	-4	-2	-3	-11								66,809
3552	20011070	3551	neqapee	1986-03-22	2022-10-08	2			-10	10														-7,609
3553	10011868	3552	naiozii	1998-12-17	2032-06-12	1	1	1	3	12	1	2	-4	-2	-3	-11								73,695
3554	20010998	3553	juvulab	1986-03-19	2023-11-16	2	-1	-11	-11	-48	71													-93,270
3555	10012713	3554	vasejun	1998-12-19	2024-02-22	1				17	1	2	-3	-4	-2	-11								67,828
3556	30010008	3555	qolueuc	1998-07-31	2029-11-26	3		11	10	-29	-1	9												6,141
3557	10014438	3556	nufof	1998-12-15	2025-03-15	1				17	1	2	-3	-4	-2	-11								68,069
3558	30010335	3557	penetux	1998-08-28	2026-10-27	3				-7		7												-12,022
3559	20010491	3558	vanonol	1986-03-18	2026-04-16	2	-30	-10	-12	-20	72													-189,104
3560	10014781	3559	tenan	1998-12-18	2025-04-10	1				16	2	3	-4	-4	-2	-11								67,285
3561	10012705	3560	cauou	1998-12-19	2027-11-09	1				16	2	3	-4	-4	-2	-11								67,285
3562	20010026	3561	rexuoiw	1986-04-11	2017-06-27	2				-8	8													-7,415
3563	30010791	3562	uuuepeb	1998-08-27	2027-04-11	3				-7		7												-12,022
3564	10014683	3563	janoi	1998-12-18	2025-08-01	1				16	2	3	-4	-4	-2	-11								67,285
3565	20011012	3564	tapiqov	1986-04-12	2018-06-06	2				-20	20													-18,246
3566	10010578	3565	vakoted	1998-12-19	2026-05-28	1				16	2	3	-4	-4	-2	-11								67,285
3567	20010550	3566	ceeixen	1986-04-09	2019-09-08	2				-34	34													-32,230
3568	10012290	3567	redin	1998-12-16	2029-02-21	1		2	3	11	2	3	-4	-4	-1	-12								72,509
3569	30010386	3568	xakik	1998-08-26	2029-12-23	3		12	10	-29		7												9,571
3570	10013840	3569	oalov	1998-12-18	2029-10-05	1		2	3	11	2	3	-5	-3	-1	-12								71,122
3571	20010974	3570	kikok	1986-04-11	2020-07-18	2				-44	44													-42,564
3572	10014769	3571	giridiu	1998-12-18	2017-07-09	1																		0
3573	20011274	3572	bahovea	1986-04-08	2021-02-12	2				-51	51													-49,424
3574	20010968	3573	kakixee	1986-04-09	2021-07-20	2				-56	56													-54,650
3575	10014071	3574	mufoe	1998-12-17	2018-04-19	1																		0
3576	30010763	3575	zoreooj	1998-08-26	2033-05-20	3	35	18	10	-69	-1	7												131,371
3577	10013338	3576	xuzaiow	1998-12-18	2025-07-05	1				16	2	3	-5	-2	-2	-12								66,591
3578	10012548	3577	geuas	1998-12-17	2028-03-15	1			1	15	2	3	-5	-2	-2	-12								66,989
3579	30010524	3578	eetiq	1998-09-23	2019-02-18	3																		0
3580	20011685	3579	duuam	1986-04-08	2023-03-13	2		-3	-11	-57	71													-81,831
3581	10012503	3580	suoom	1998-12-15	2029-12-18	1		1	4	11	1	4	-5	-2	-2	-12								70,302
3582	10011834	3581	tujib	1998-12-18	2033-04-06	1		1	4	12		4	-5	-2	-2	-12								71,304
3583	20010275	3582	qiquxun	1986-04-11	2023-07-07	2		-6	-12	-53	71													-85,849
3584	30010475	3583	veaup	1998-09-22	2021-08-03	3																		0
3585	10010611	3584	yaaie	1998-12-19	2023-02-12	1				4	2	15	-5	-2	-2	-12								45,001
3586	20011469	3585	dauakah	1986-04-09	2024-01-26	2	-3	-9	-12	-46	70													-98,543
3587	10013327	3586	jokaxin	1998-12-16	2026-06-08	1				17		4	-5	-2	-2	-12								66,877
3588	10010336	3587	jayiruu	1998-12-18	2031-07-04	1		2	2	13		4	-4	-3	-2	-12								72,266
3589	20010616	3588	davuv	1986-04-10	2025-10-27	2	-23	-10	-11	-26	70													-164,586
3590	30010751	3589	juquyak	1998-09-22	2024-05-29	3				-5	-3	8												-10,733

Fig. 5: example *job_diff_to_excel* module function output - column width and "cpay_diff" number formatting must be done manually following the creation of the spreadsheet

REPORTS notebook

seniority_list includes an example notebook demonstrating the usage of the *reports* module functions. The datasets must be created first before attempting to generate reports.

CHAPTER

EIGHT

EXAMPLE GALLERY

The examples on this page are a subset of what is possible with seniority_list. The charts below were created using the **built-in plotting functions** included with the program. The plotting functions generally accept many different inputs and optional parameters, offering analysis over a range of attributes for single or mutliple employee groups. Again, **much more is possible than what is seen in the samples below**.

The datasets generated by the main program serve as the data source for the charts. Program inputs may be altered and new datasets quickly recalculated to reflect a different scenario, such as a change in job counts or a different recall schedule. All of the charts charts below could be redrawn to reflect those changes in a matter of minutes.

The visualization of the data is not limited to the built-in functions. Users with some coding experience may write customized functions to explore the data in other ways.

These charts are representative of a three-party integration. The program is able to handle an integration of any number of workgroups.

Note: Chart titles and other references in this gallery are generic. Actual charts include text linked to inputs. Job category descriptions in this gallery reflect airline pilot positions. The descriptions are easily customizable to match job descriptions for other industry case studies.

8.1 screenshots and notes

Most of the following plots have multiple inputs. These inputs allow various groupings to be studied. The more common groupings include values in a particular month or months, age or date ranges, employee group (eg) selection(s), quantiles, and job levels.

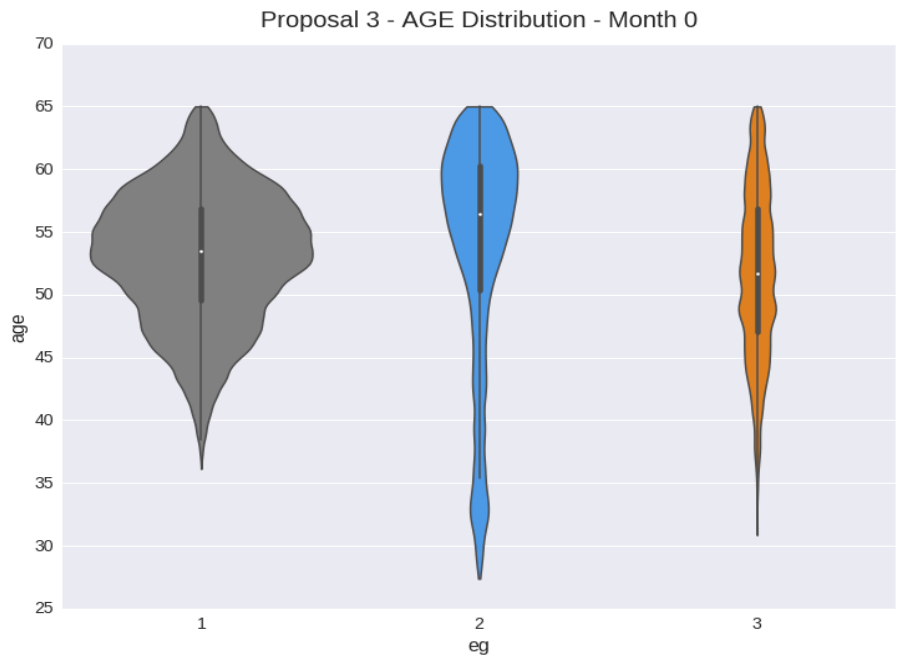


Fig. 1: age distribution by group - violin plot[51]

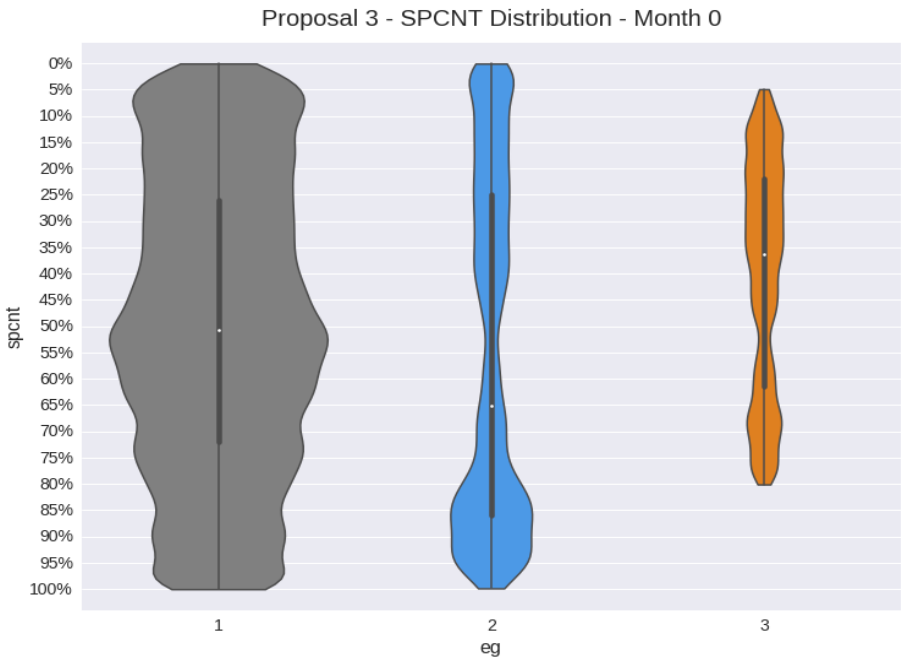


Fig. 2: percentage distribution by group[52]

[51] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.violinplot_by_eg
[52] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.violinplot_by_eg

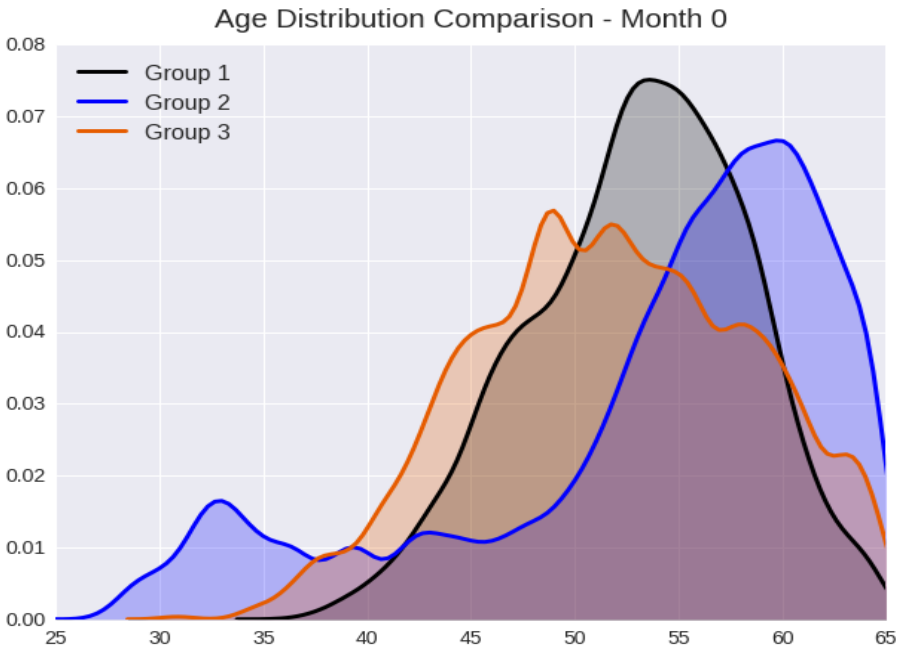


Fig. 3: kde distribution[53]

How are jobs distributed throughout the separate lists? This chart compares native job distribution to native list percentage. The shaded areas indicate that the job level distribution within one group is not monotonic (uniformly decreasing) due to a pre-existing special premium job assignment condition for certain members of one group and also furloughed employees mixed in with active employees.

[53] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.age_kde_dist

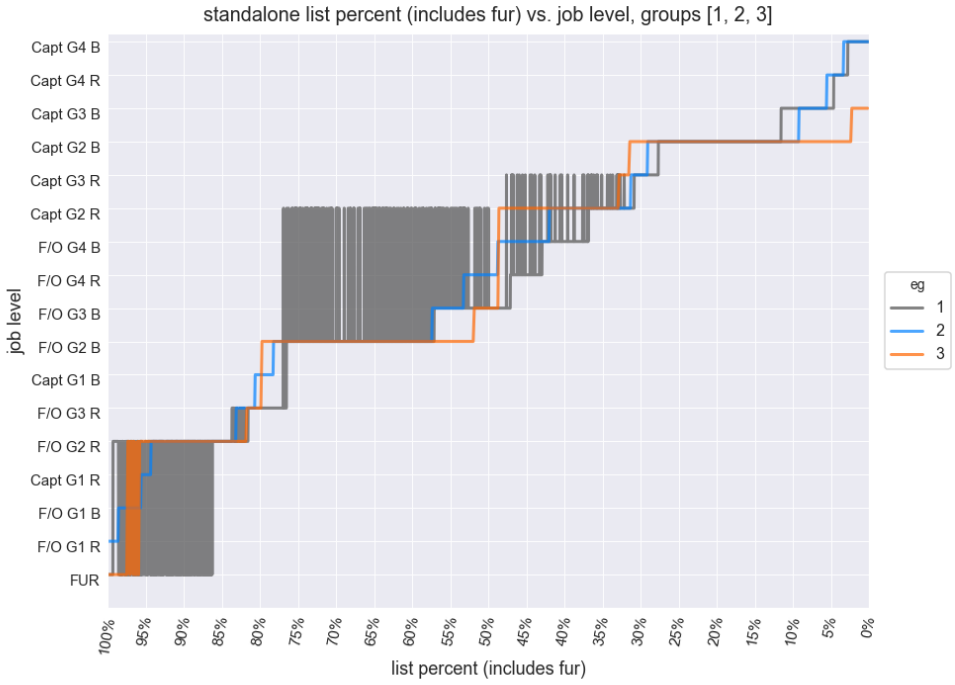


Fig. 4: standalone jobs by group and list percentage[54]

An age-percent chart presents a visualization of the distribution of employees by age and percentage within a proposed integrated list or standalone list(s). Users may plot data from a particular month, employee group(s), job level, age range, longevity range, etc.

[54] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

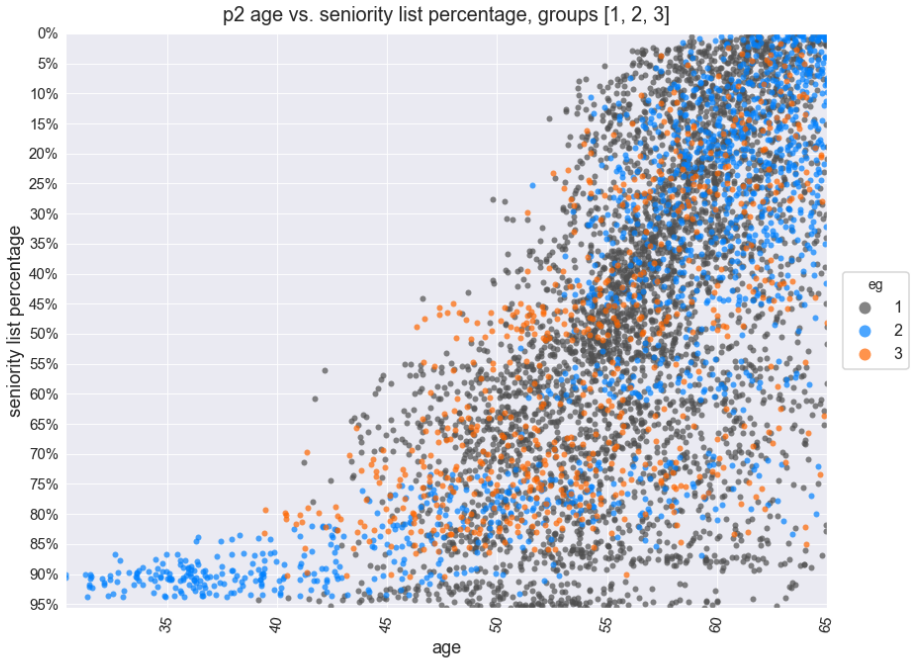


Fig. 5: age vs. list percentage by group[55]

Example view of a data for a selected single group displayed for a future month:

[55] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

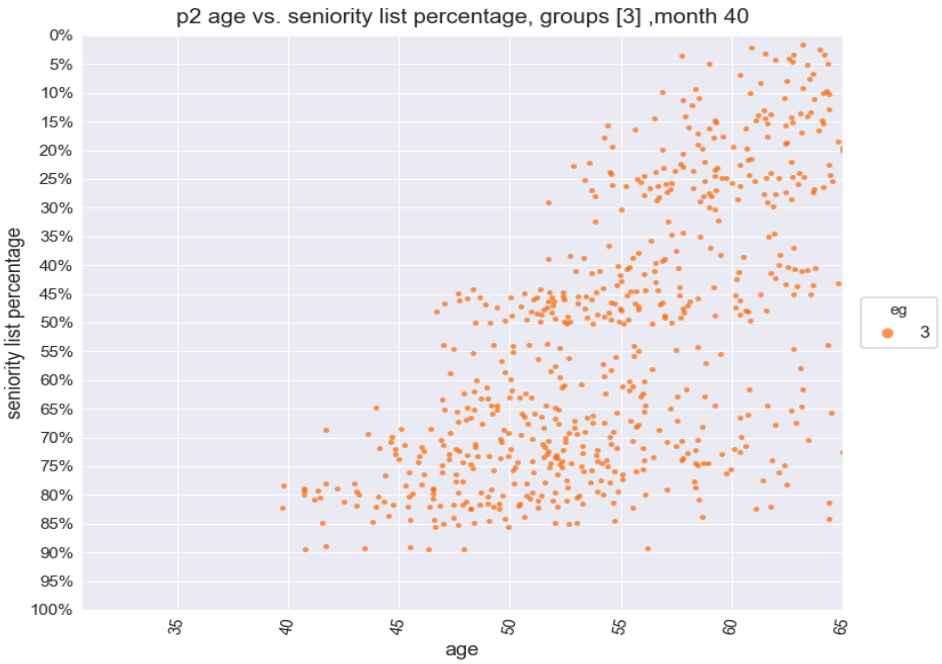


Fig. 6: age vs. list percentage, single group, future month[56]

Due to jobs held at implementation combined with the no bump, no flush provision, employees actually holding a job within a job level may be dispersed over a wide range of an integrated list rather than falling within a concise percentile range. Other special conditions may have a similar effect. An example of this distorted disbursement is illustrated below. The three groups are holding the same job within the integrated list yet are located on the list at different levels. After implementation and as the separate groups mesh, this stratification would lead to different job opportunities within the same bid category.

[56] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

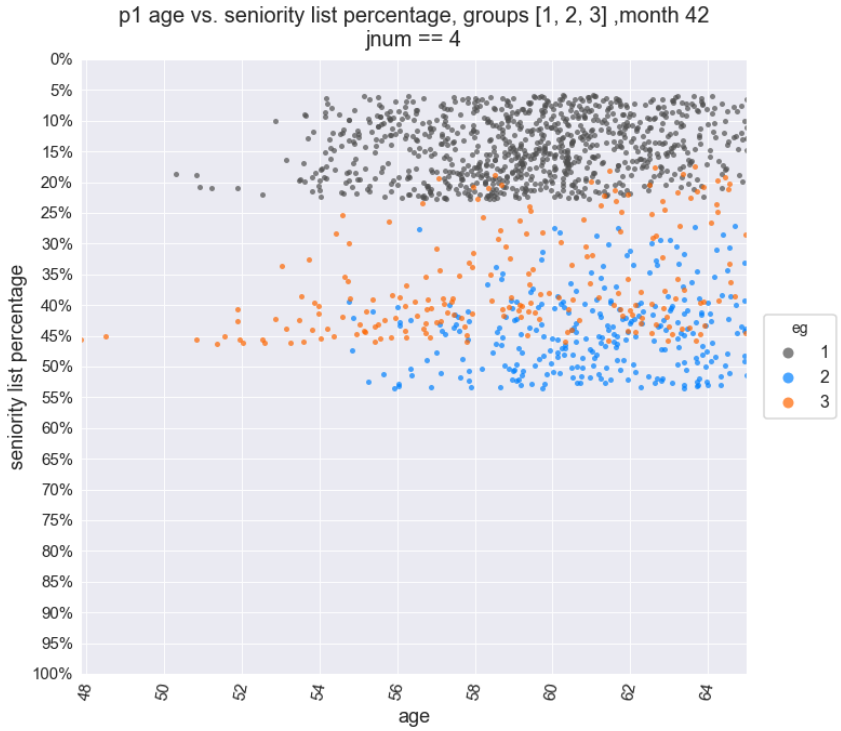


Fig. 7: age vs. list percentage, all groups, same job, future month[57]

Here is an example of an aggregate group measure in the form of average longevity vs. list percentage for three groups. YLONG stands for the decimal year longevity attribute. The y axis is years of longevity and the x axis is list percentile.

[57] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

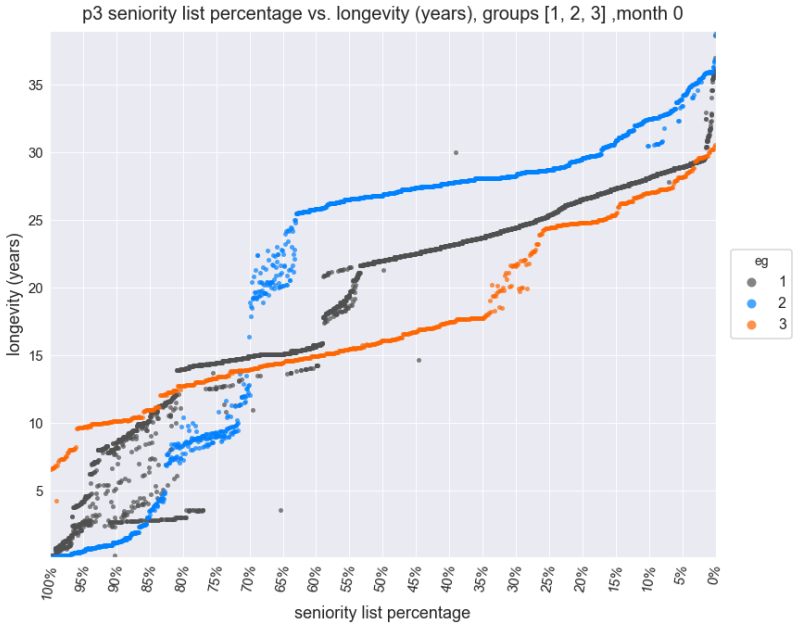


Fig. 8: longevity (y axis) vs. proposed list percentage (x axis), month 0[58]

The following chart illustrates the average JNUM (job number) indicated with respective job description labels (y axis) for a proposed list over time. The job description labels are fully customizable inputs.

[58] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

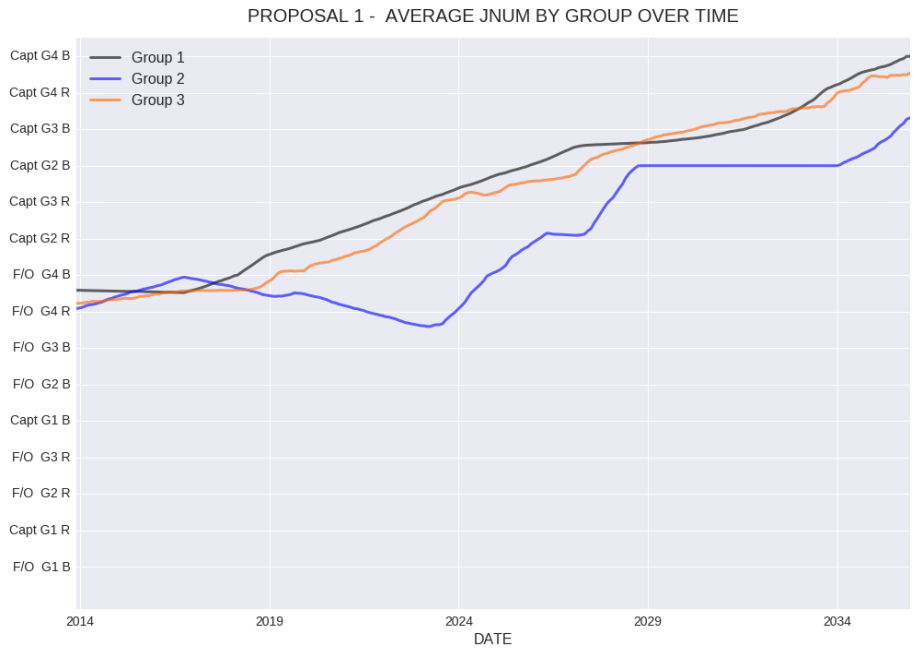


Fig. 9: average group job level[59]

This is the same chart with comparative standalone average job levels added with the dashed lines.

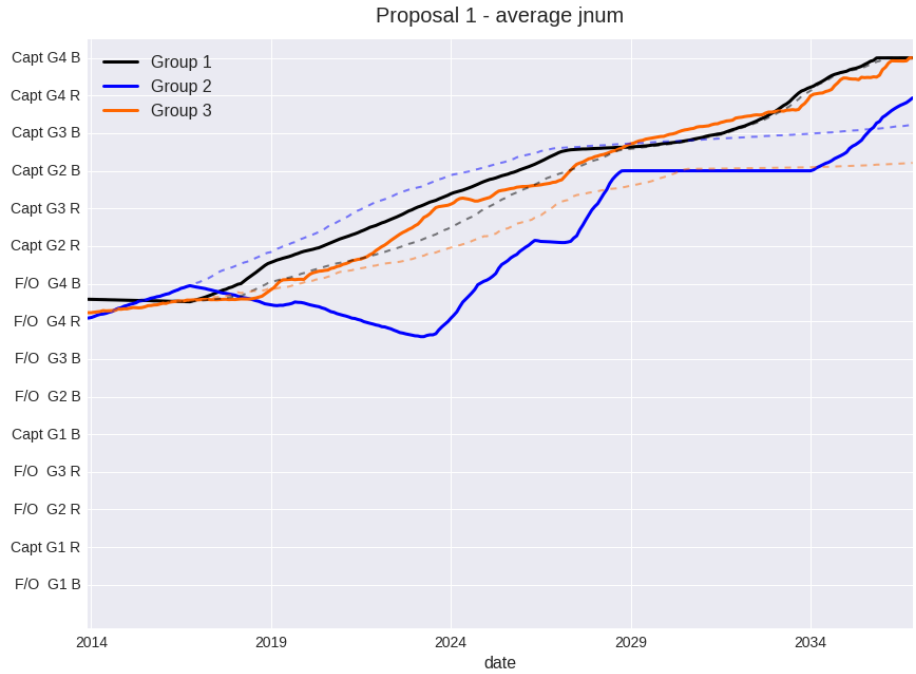


Fig. 10: average group job level with standalone comparison[60]

[59] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.group_average_and_median

Specific employee numbers may be selected for individual plots. The plots reflect an average for that percentage level on the list for the respective employee group due to the fact that the list is organized in a stovepipe fashion.

In the following chart, the display indicates the job level progression for three employees from three different groups. These employees are next to each other on the proposed list. The three employees initially hold three different jobs from their original lists. The employees which initially hold a higher level job are protected in that job until his or her new list cohorts "catch up" through the job vacancy process. At that point, all three track together until retirement.

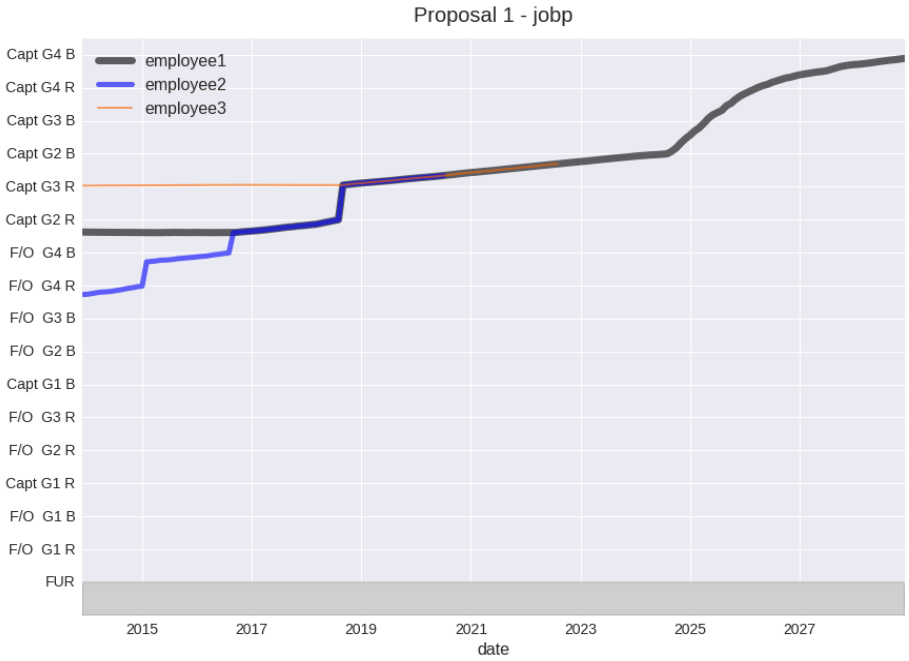


Fig. 11: percentage within job level indicating no bump no flush effect[61]

Because a compensation attribute is built into the model, it is possible to study the change in the flow of money among the workgroups. In the chart below, a normalized compensation comparison is made with standalone figures. The model assumes the same level of compensation pre- and post- integration. This result could also be thought of as a job quality change measurement.

[60] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.group_average_and_median
[61] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.multiline_plot_by_emp

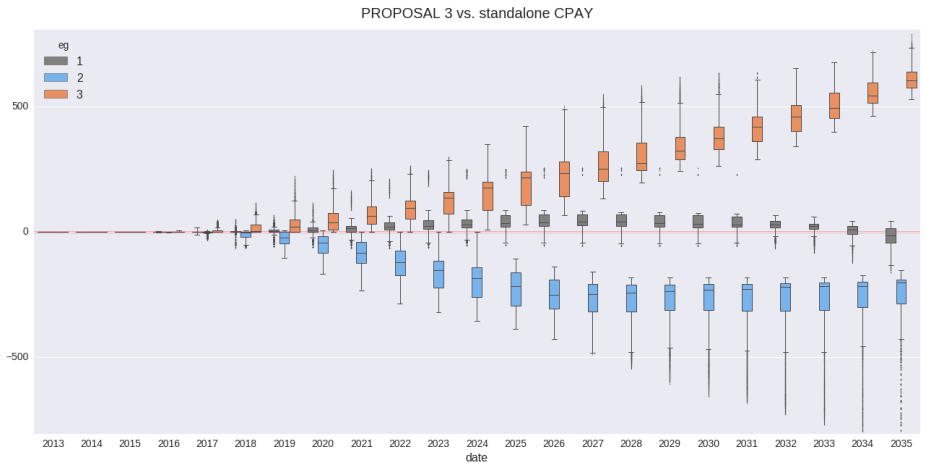


Fig. 12: group compensation within an integrated proposal vs. standalone, delayed implementation[62]

Several attributes may be loaded into the function which produced the chart above. Here is another example of the same charting function which compares list seniority percentage, native vs. proposed ordering.

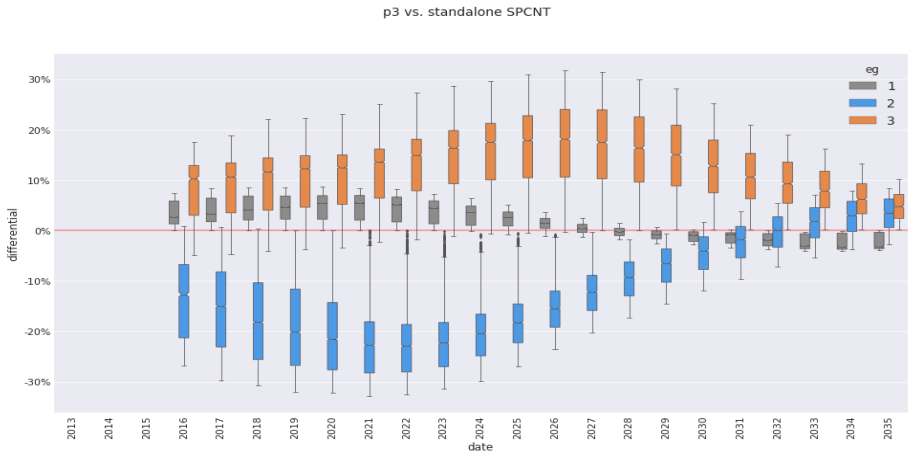


Fig. 13: employee group list percentage differential: integrated proposal vs. standalone[63]

Actual attribute value ranges (as opposed to differential values) for each employee group may be plotted as well. This chart compares the placement of employee groups within an integrated data model for each year through 2035 (x axis) in terms of list percentage (y axis, most senior at the top), with implementation of the integrated list in late 2016. Other dataset attributes may be easily displayed, such as job category ranking or career pay.

[62] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_diff_boxplot
[63] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_diff_boxplot

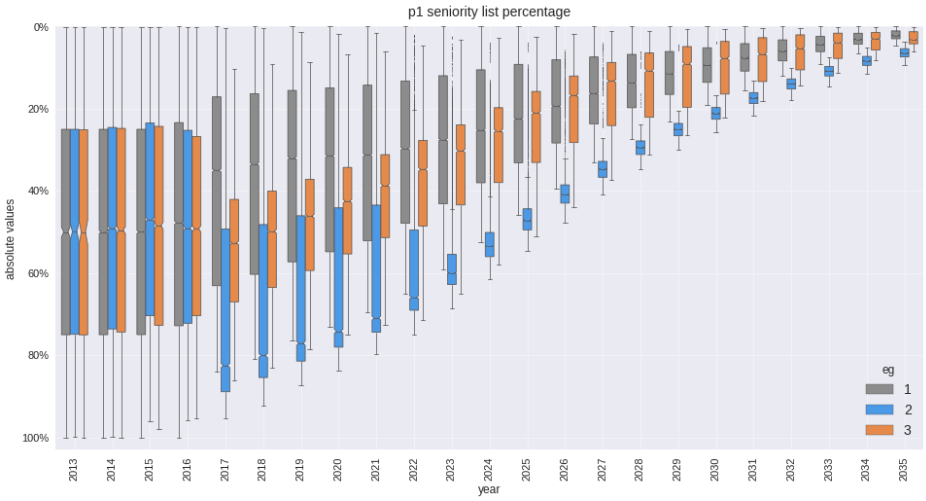


Fig. 14: absolute (actual) group list percentage within an integrated proposal, over time[64]

The vertical stripplot offers another way to visualize employee group distribution.

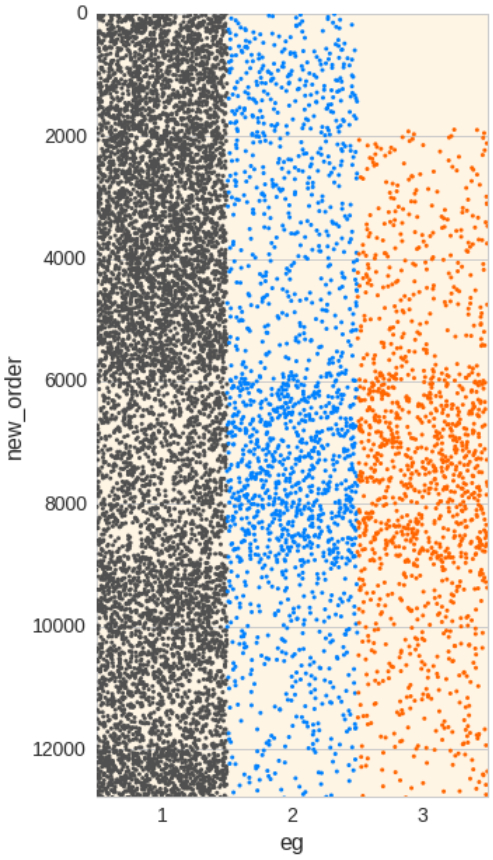


Fig. 15: group distribution - stripplot[65]

[64] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_boxplot

Here is a stripplot which reveals the job distribution within correctly sized job level bands, for a month in the future, for a particular proposal.

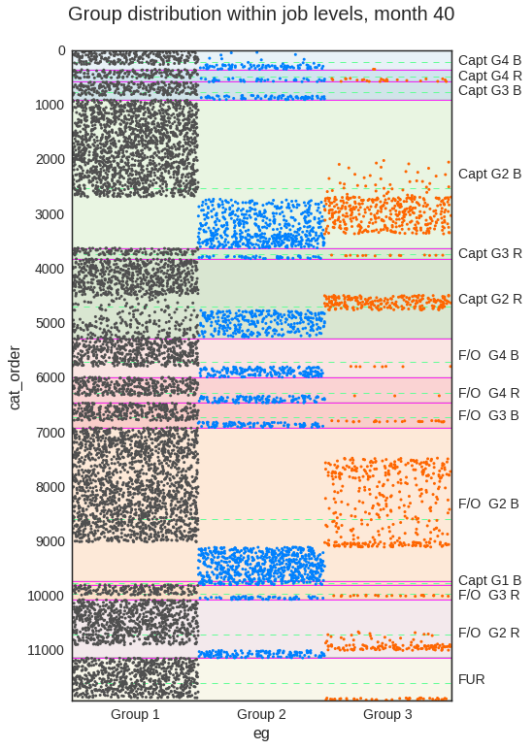


Fig. 16: group distribution within job level zones, future month, with job level color bands[66]

This chart shows the distribution of the employee groups within each job level as it relates to a proposed seniority ordering. The green markers represent a subgroup of one of the main groups. This subgroup has special job rights which existed prior to the integration. The program incorporates and accurately models special conditions.

[65] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.stripplot_eg_density
[66] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.stripplot_distribution_in_category

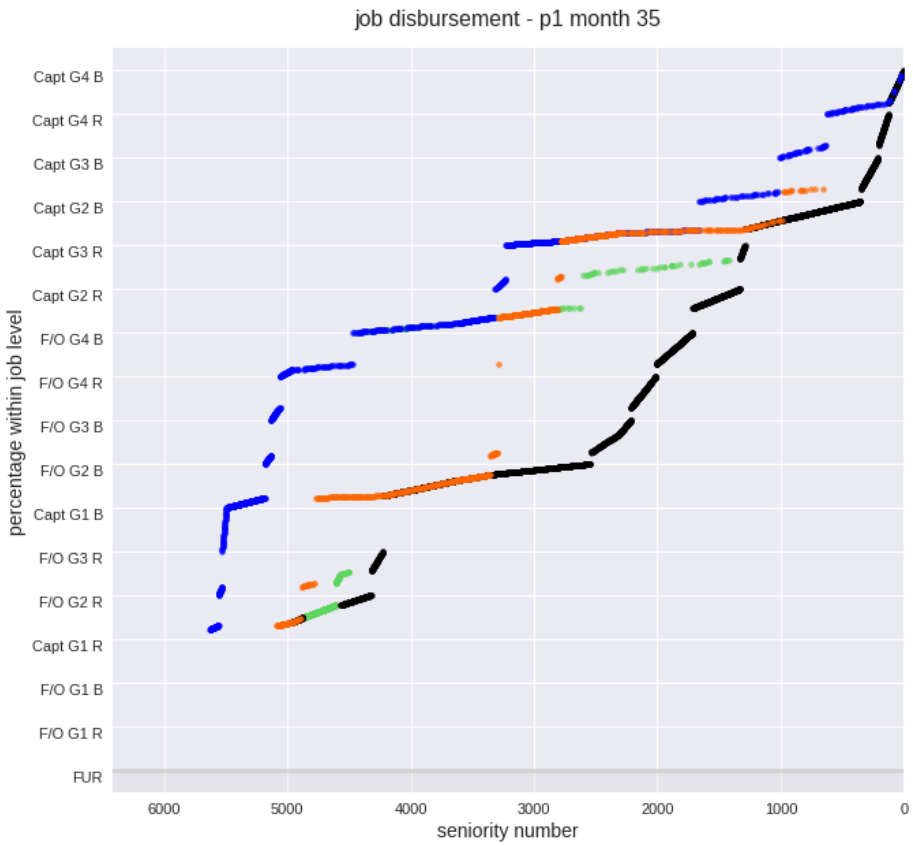


Fig. 17: job distribution vs. proposed seniority number[67]

This is a scatter plot displaying similar information as the chart above, with the added feature of appropriately colored and correctly scaled job level bands. The 35th month of the model has been selected for study, which in this case is the first month following the implementation of the integrated list. The displayed job bands account for job count changes over the life of the data model and are correctly sized for the selected month. Each employee group may be displayed separately and a line chart may be drawn instead of the scatter plot. There are other chart options as well. In this particular presentation, the green markers again identify a subset of one of the employee groups who possess special job assignment rights.

[67] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_multiplot_with_cat_order

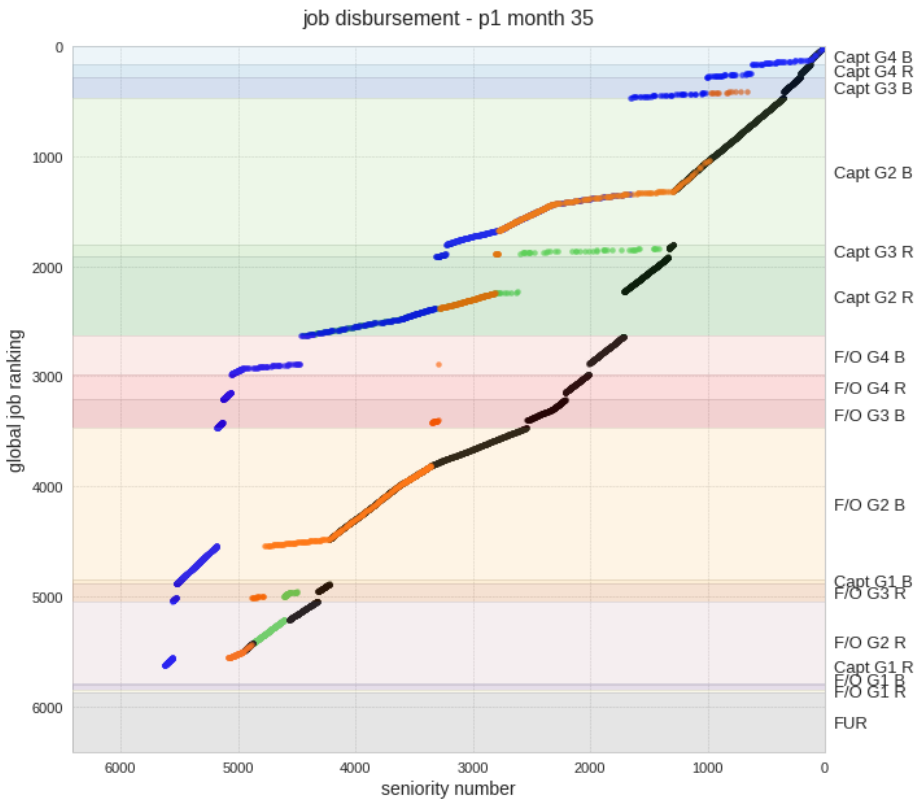


Fig. 18: group distribution within job level zones vs. list percentage, starting month[68]

Similar to the above chart, with the x axis changed to reflect age:

[68] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_multiplot_with_cat_order

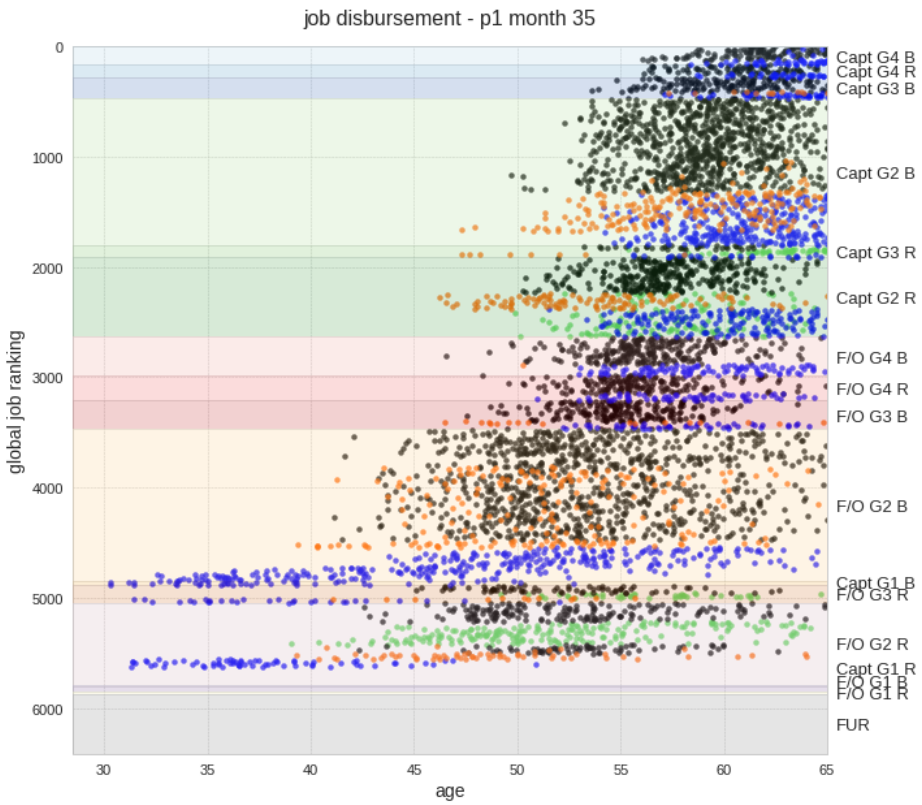


Fig. 19: group distribution within job level zones vs. age, future month[69]

. . . with the x axis changed to reflect years of longevity (in this case, stovepiped, or ordered, longevity for each group):

[69] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_multiplot_with_cat_order

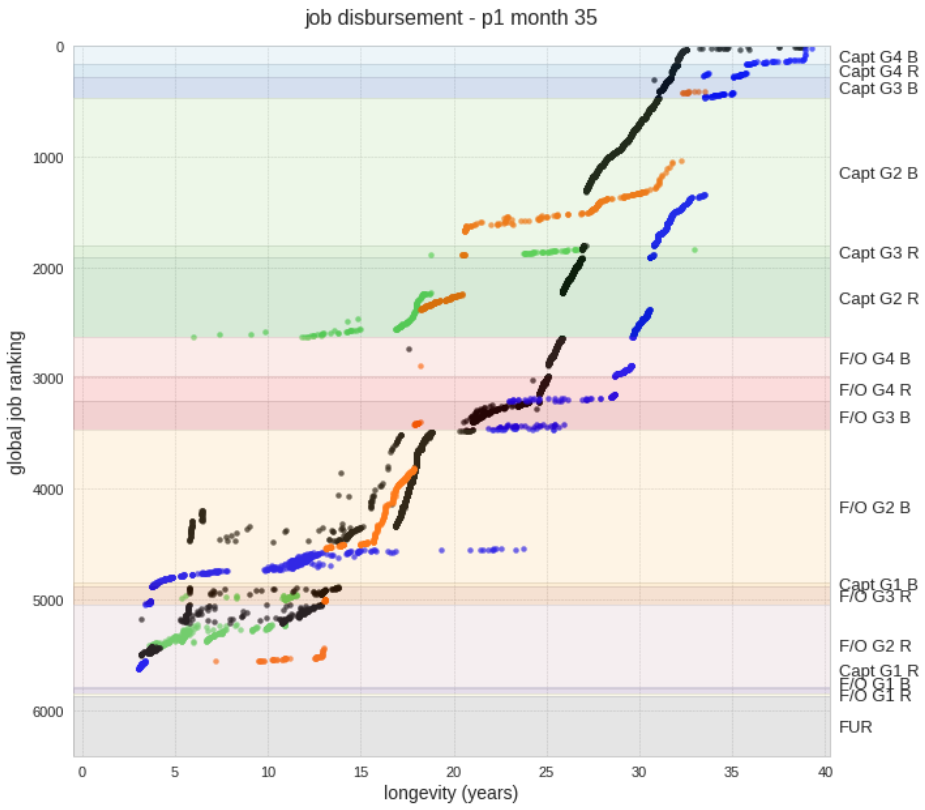


Fig. 20: group distribution within job level zones vs. longevity, starting month[70]

Differences between list locations for employees with equivalent attribute values but from different groups may be studied with the cohort_differential chart. In the following example, the longevity attribute for employee group 1 is being compared to employees from groups 2 and 3. The code finds the list locations of employees from groups 2 and 3 which match the longevity values from group 1, then displays the location differences from the group 1 locations.

[70] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_multiplot_with_cat_order

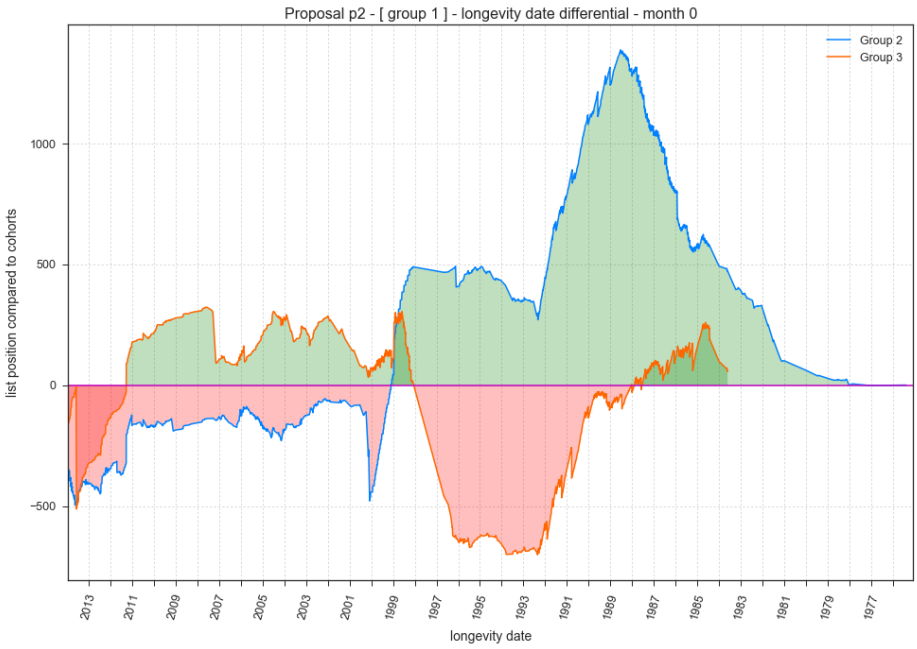


Fig. 21: longevity cohort list location differential (click to enlarge)[71]

Job levels obtained over time may be visually represented by a step-type chart and/or by a line representing the percentile within that job level.

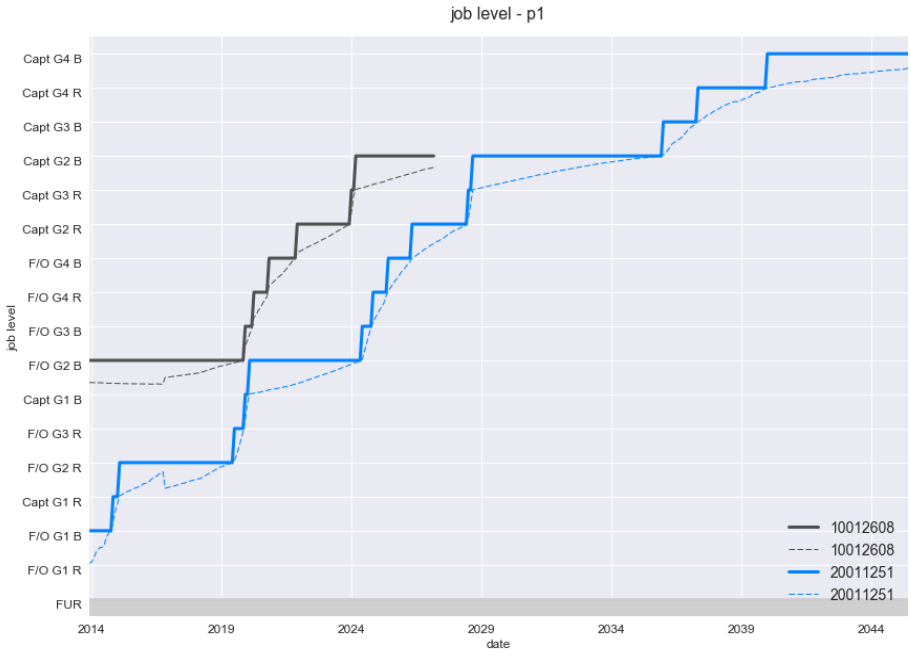


Fig. 22: job number and job percentage[72]

[71] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.cohort_differential

The datasets may be filtered and sliced in many ways to examine and compare target ranges. This is an example of time slicing, focusing on employees hired in 1989. Note that the underlying data model incorporated a delayed implementation date of late 2016. The employee groups operate independently until then.

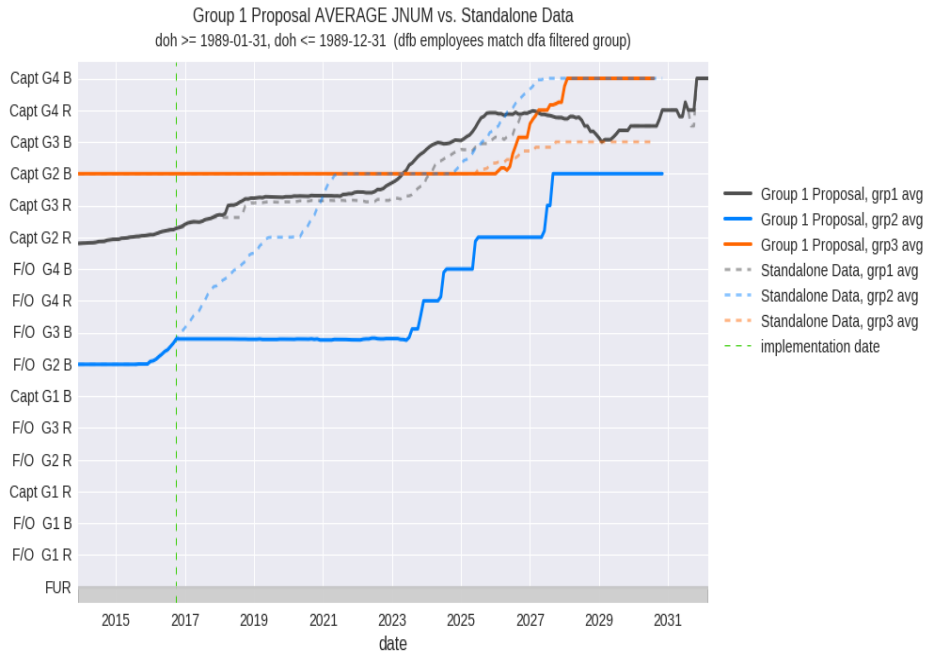


Fig. 23: time slice example, standalone data is indicated with the dashed lines[73]

In this example proposal, a significant loss is indicated for one group while another group gains. Standalone data is indicated with the dashed lines. Note that the groups began with nearly identical values at the implementation date. This chart is displaying a slice of a job rank attribute reflecting employees with a longevity date of 1995 or earlier.

[72] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.multiline_plot_by_emp
[73] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.group_average_and_median

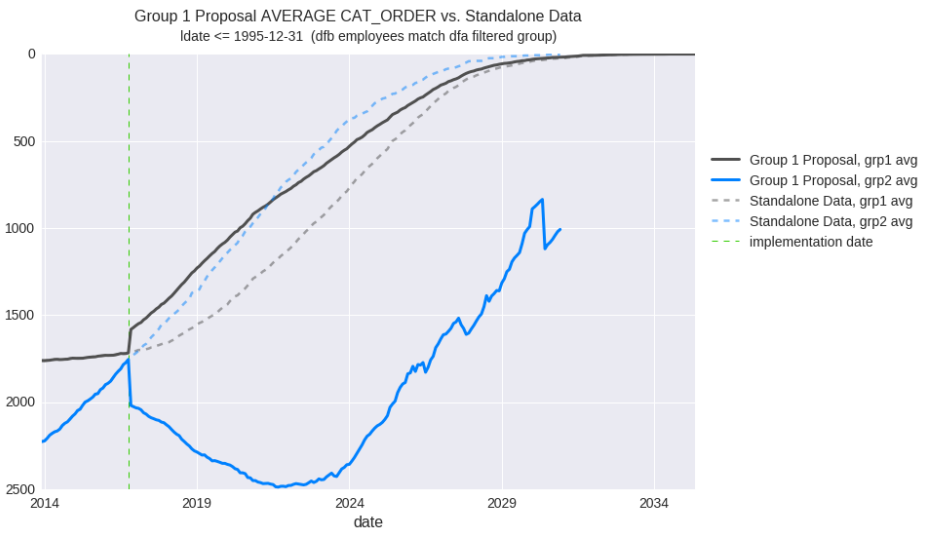


Fig. 24: longevity filter example, standalone data is indicated with the dashed lines[74]

Here is a chart utilizing the compensation section of the data model. The y axis represents monthly compensation in thousands of dollars while the x axis indicates percentage on the list. In this case, the plots represent results for several individual employees.

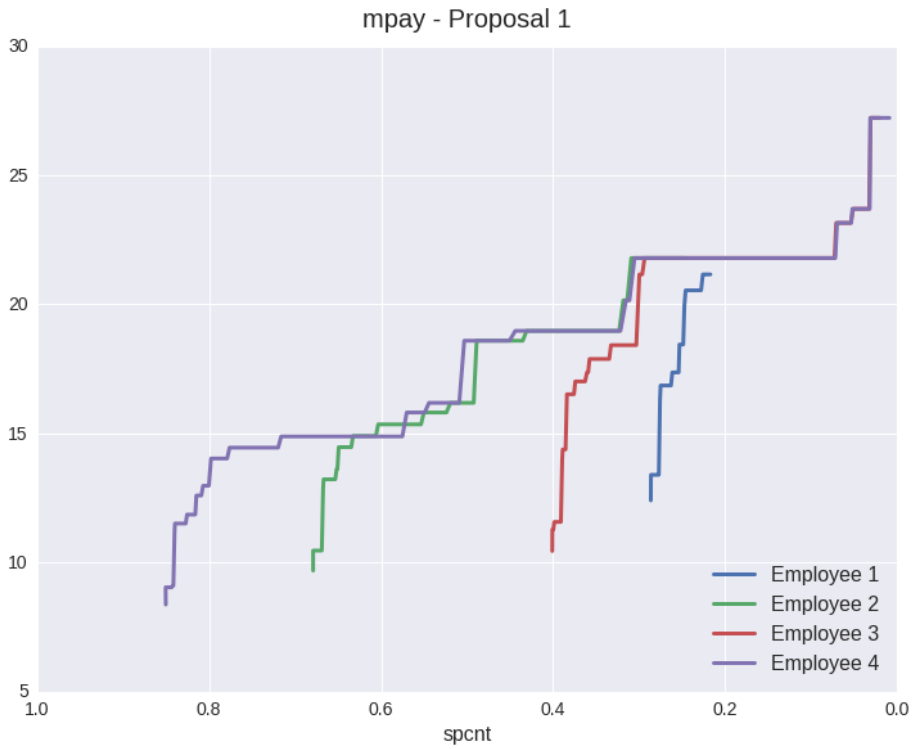


Fig. 25: monthly compensation (y axis) vs. list percentage, selected employees[75]

[74] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.group_average_and_median

Parallel coordinates type charts are very good at comparing positional differences, such as when comparing list percentage or numerical job levels.

For each of the subplots below, standalone list percentage is indicated on the left vertical line. The top row is for month zero, and the second and third rows are for future months. The other vertical lines represent the list percentage for those same employees within different proposals. Any number of groups and future months may be plotted and the left vertical line can be set to represent any of the proposals.

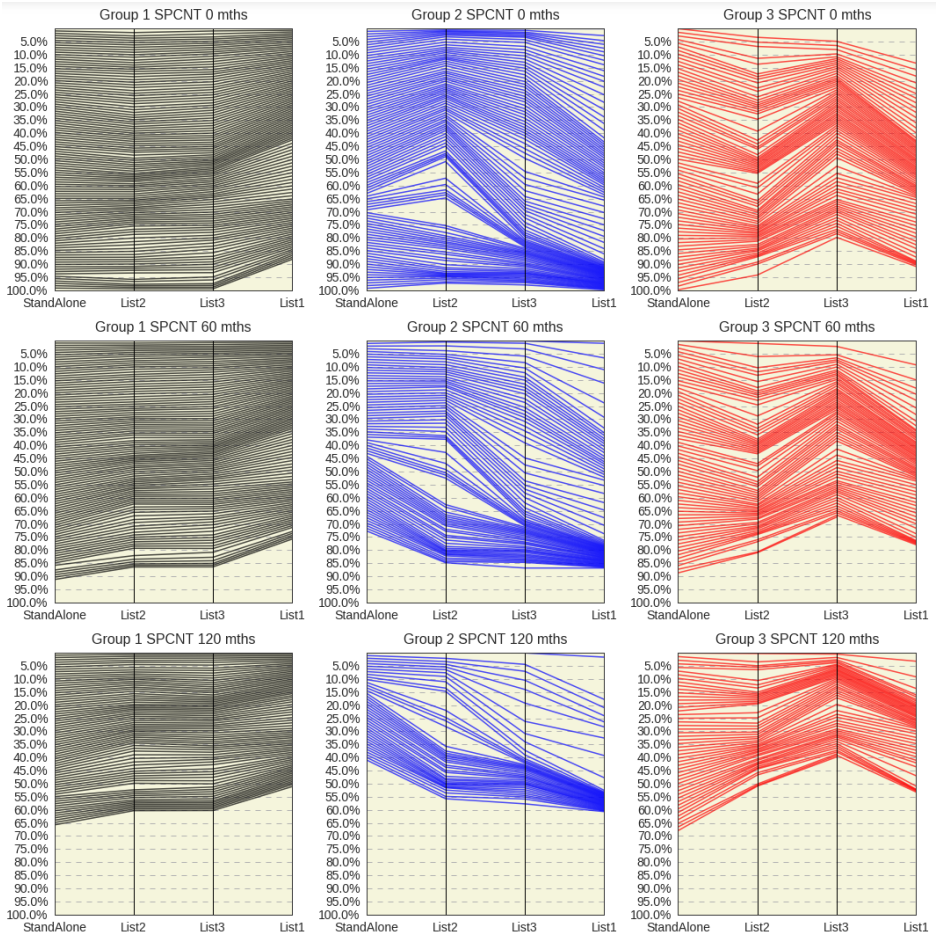


Fig. 26: parallel coordinates, group list percentage, 3 proposals vs. standalone, selectable time period[76]

Same as above, with percentage within job level attribute. (May not be the same proposal as used for chart above)

[75] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.multiline_plot_by_emp
[76] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.parallel

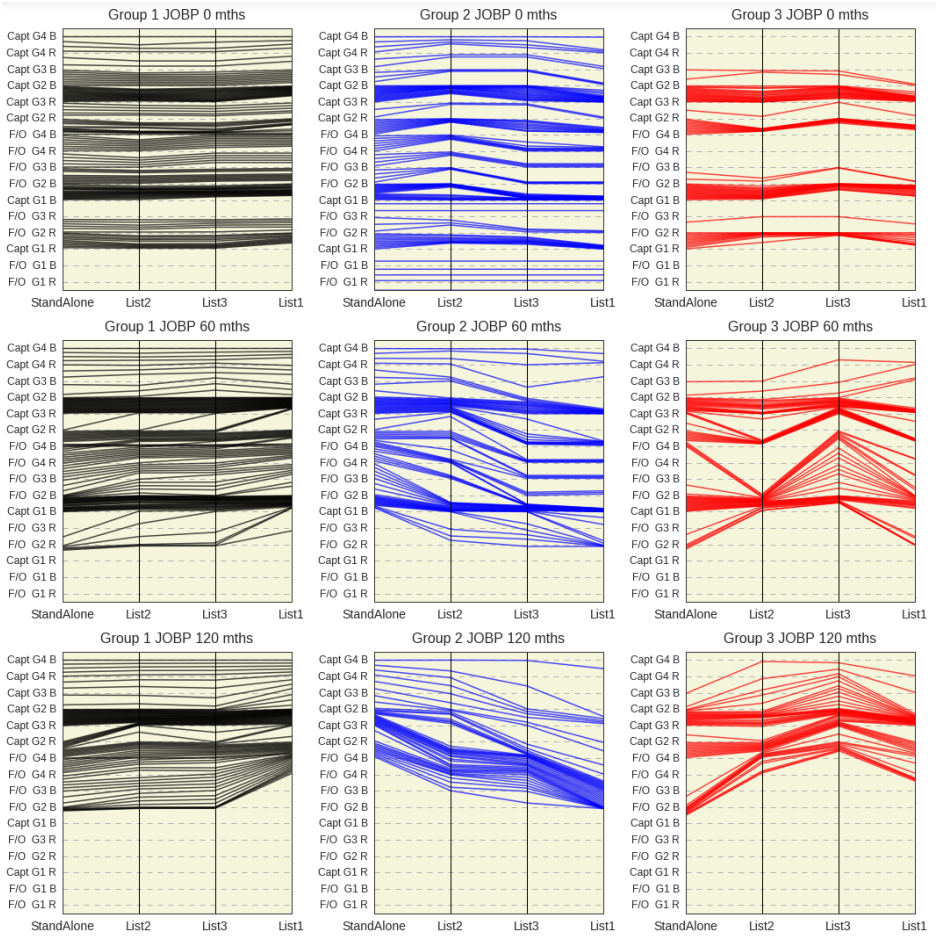


Fig. 27: parallel coordinates, group job levels, 3 proposals vs. standalone, selectable time period[77]

List percentage differential over time may be analyzed in another format utilizing differential binning, or counting the number of employees within various levels of percentage change. The following chart displays the annual differential between proposal "p1" and standalone data, as it applies to employee group 2.

[77] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.parallel

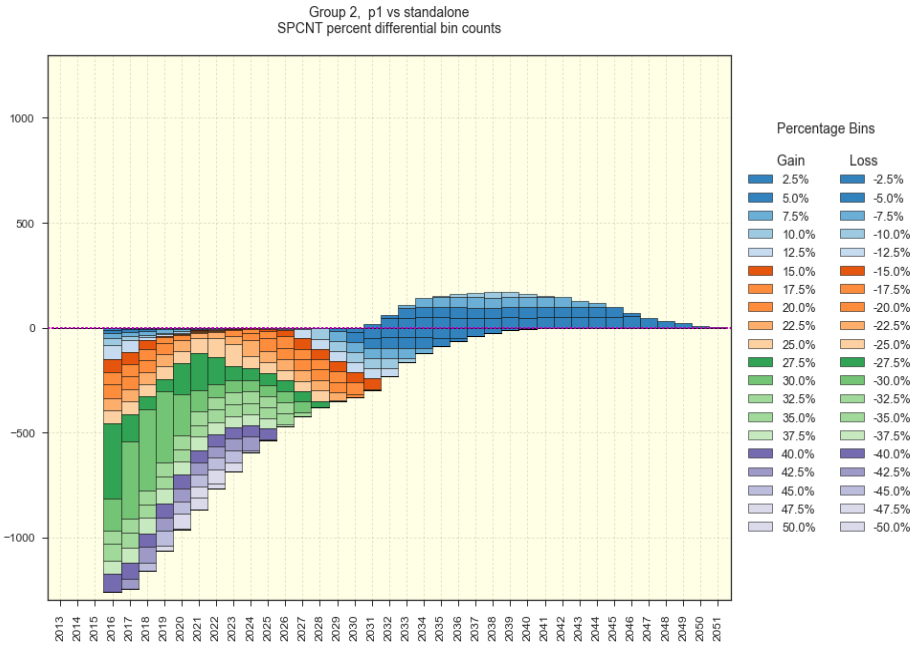


Fig. 28: annual percentage differential bin counts, revealing loss in list percentage up to 50% for many years[78]

This type of chart, along with many of the other built-in charts, may easily be set to display data which has filtered by up to three attributes. The chart below is showing differential in list percentage at retirement for employees belonging to employee group 2.

[78] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.percent_diff_bins

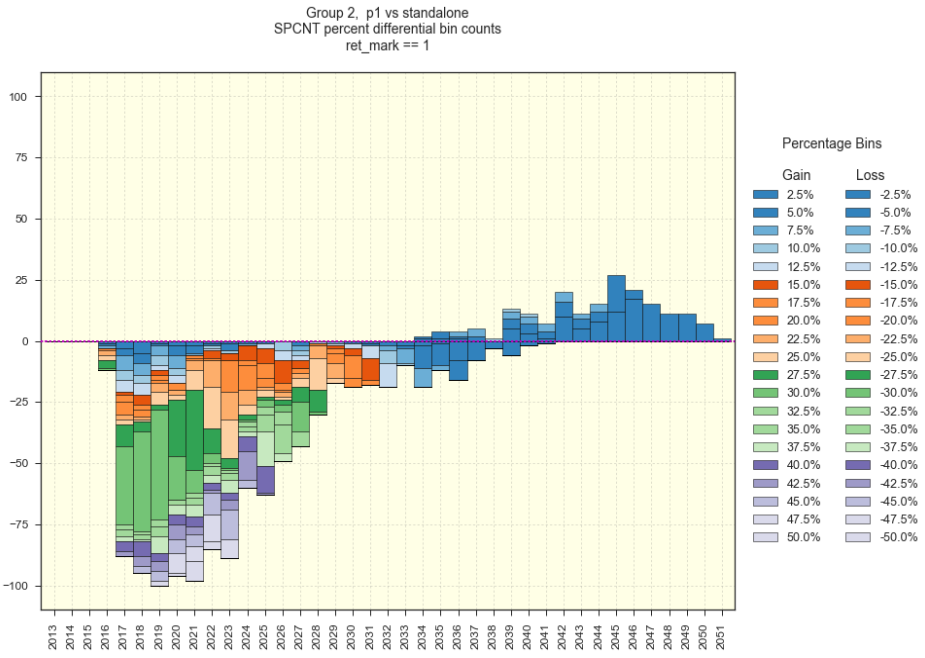


Fig. 29: annual percentage differential bin counts, measured at retirement[79]

This is one chart from a set of charts representing annual retirement data for the employee groups. The bars indicate percentage of original group count retiring each year and the job level held at retirement.

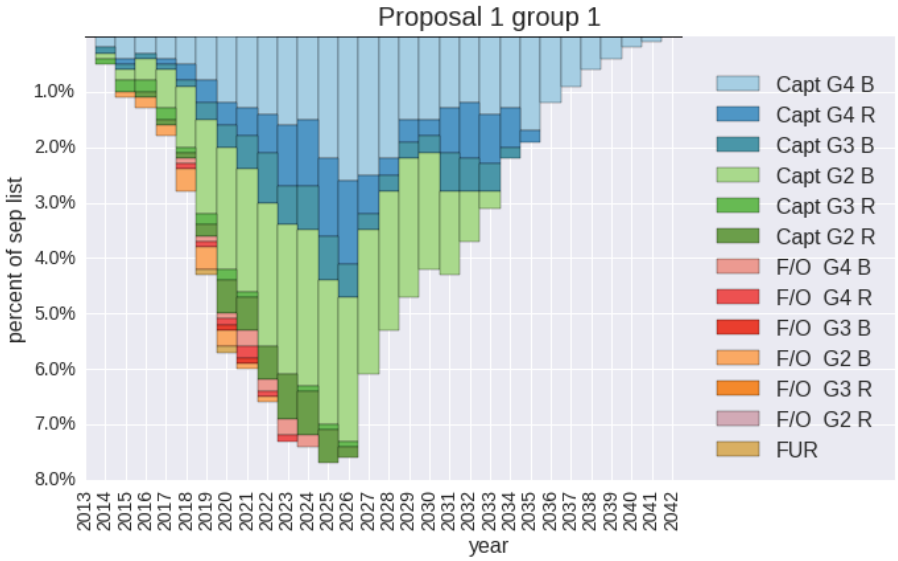


Fig. 30: retirement percentage per year including retirement job levels[80]

[79] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.percent_diff_bins
[80] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_grouping_over_time

Rows of colors charts may also be produced. In the following chart, each color represents an employee group or furloughees. As with most of the other program charts, this type of chart may be quickly customized to present various data for any month.

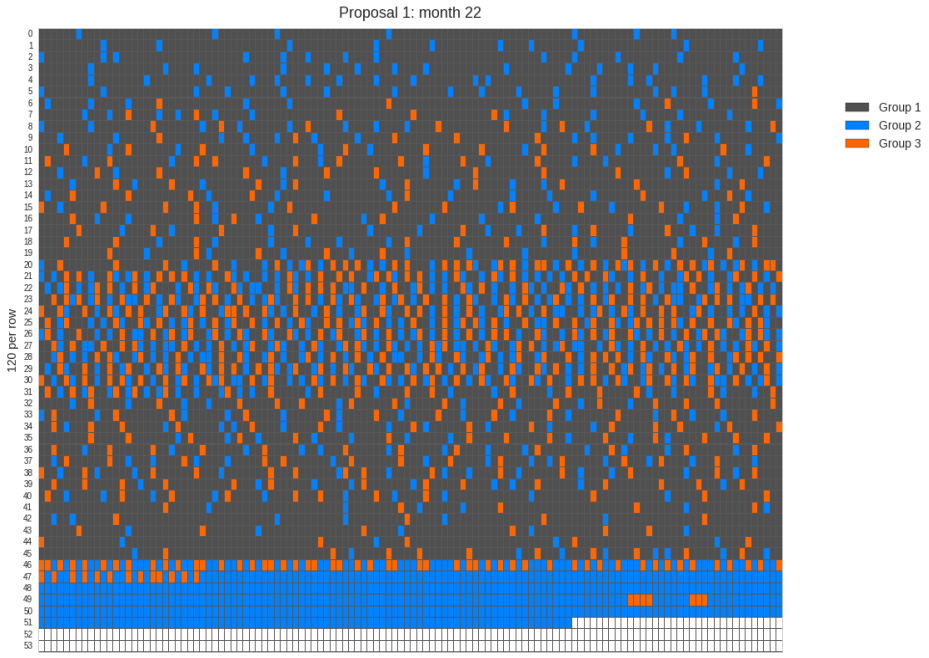


Fig. 31: group distribution, future month - color rows chart[81]

How are jobs within a job level distributed? This is a rows of colors chart very similar to the age-percent chart above.

[81] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.rows_of_color

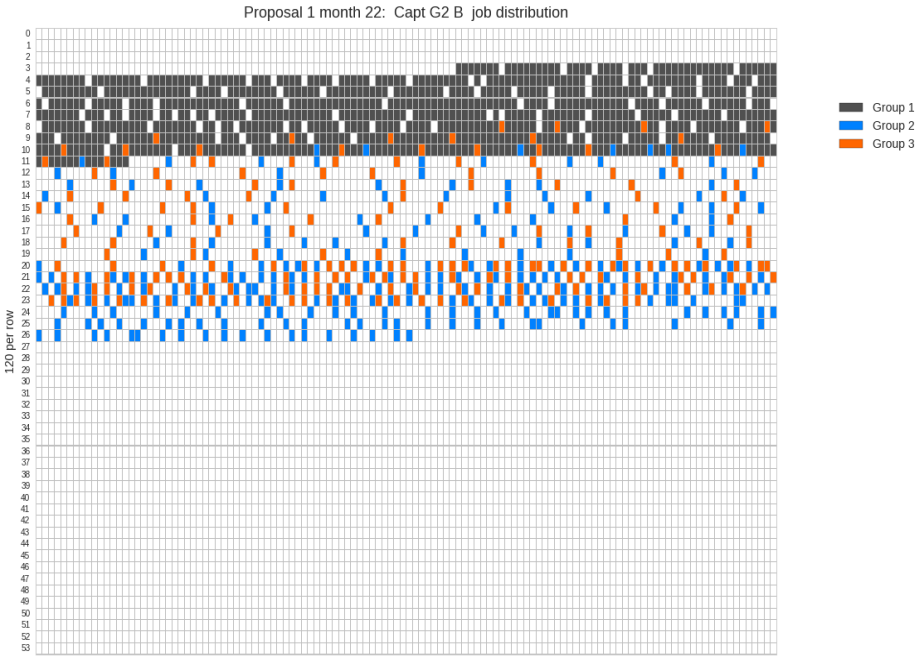


Fig. 32: single job level distribution by group, future month - color rows chart[82]

How are all the jobs distributed?

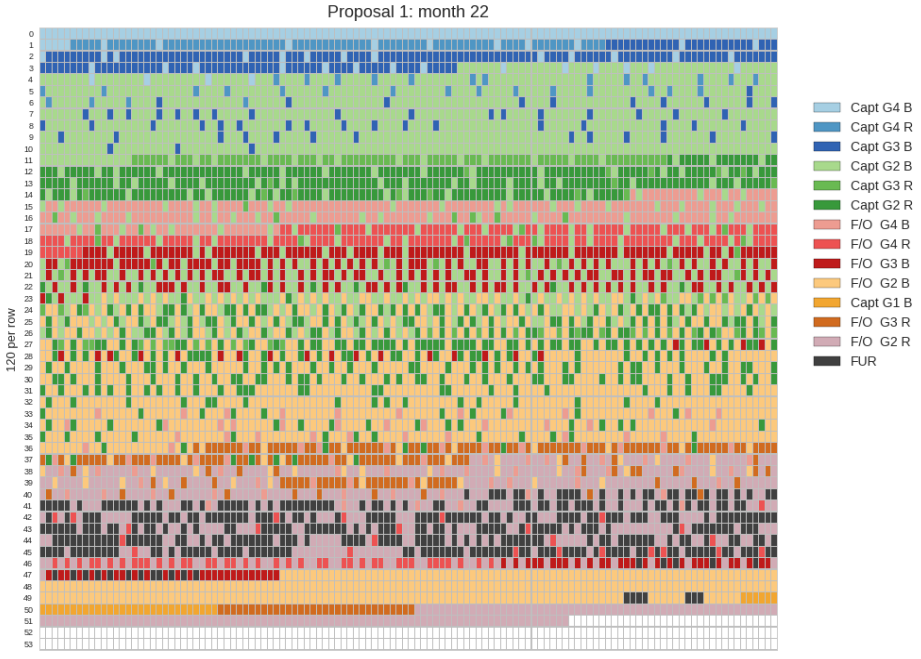


Fig. 33: all job distribution, future month - color rows chart[83]

[82] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.rows_of_color
[83] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.rows_of_color

Finally, show only the jobs assigned to one employee group.

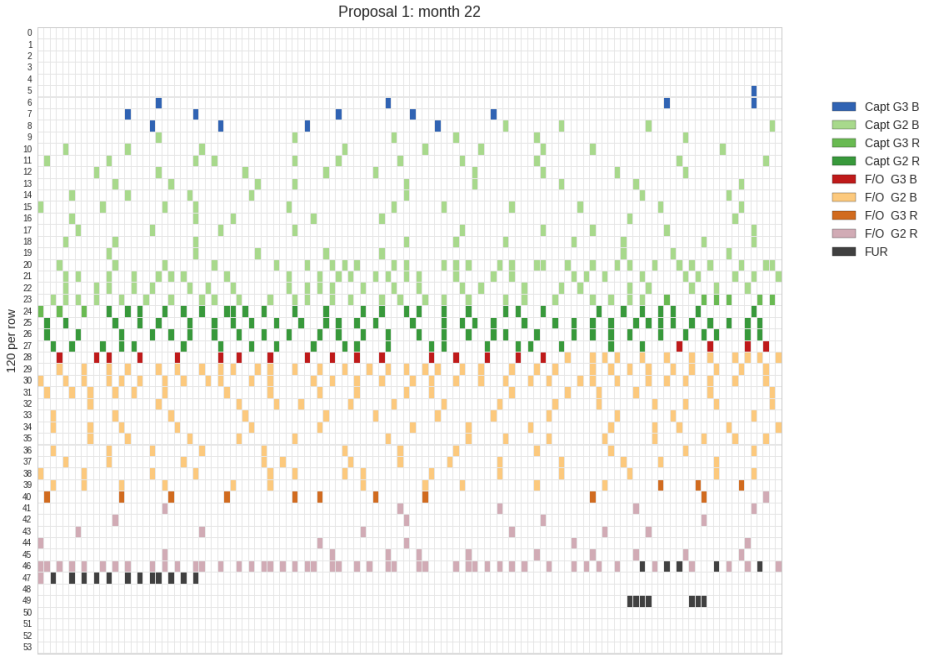


Fig. 34: job distribution, future month, single employee group - color rows chart[84]

What is the actual count of jobs held within each job level by each employee group over time? In the charts below (an excerpt of the output), modeled job counts for a proposed integration are represented by dashed lines against a baseline (normally standalone) shown with the solid lines. The total count of jobs within a given job level is represented with the green lines. This chart type is closely related to the job transfer type chart described below.

[84] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.rows_of_color

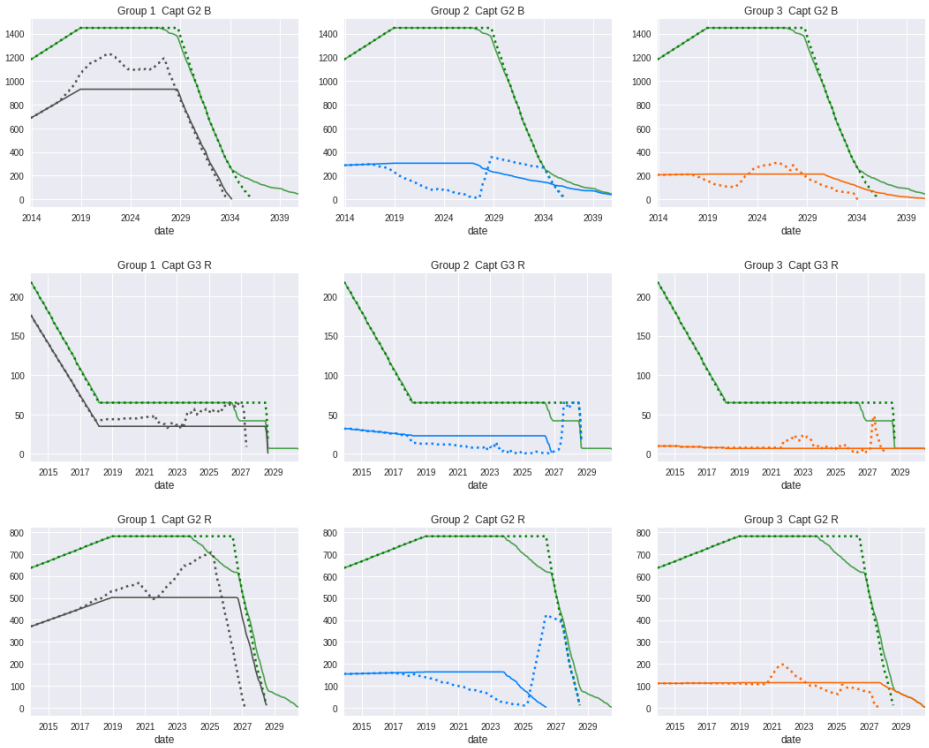


Fig. 35: job counts for three job levels (horizontal alignment) for three employee groups (vertical alignment) - dashed lines are the outcome counts for an integration proposal[85]

This chart displays the average years spent in the various job levels for an employee group within a proposal using the basic job level model. The results are grouped by quantiles. There are eight job levels within the model while using the basic job level mode in this example. The number of quantiles for the study may be changed easily.

[85] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_count_charts

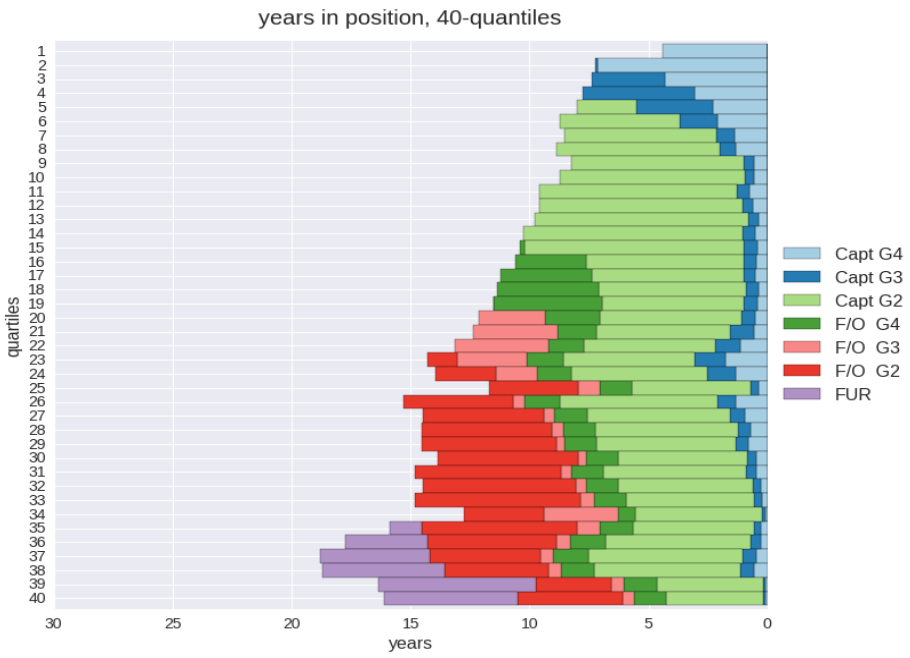


Fig. 36: single group years in position, 8 job levels[86]

Same as above, except using an enhanced job level model. The process to change between basic and enhanced job level mode is trivial. The enhanced job level mode will split the major job levels and allows more detailed job level ranking and ordering. The colors representing the job categories are fully customizable.

[86] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_years_in_position

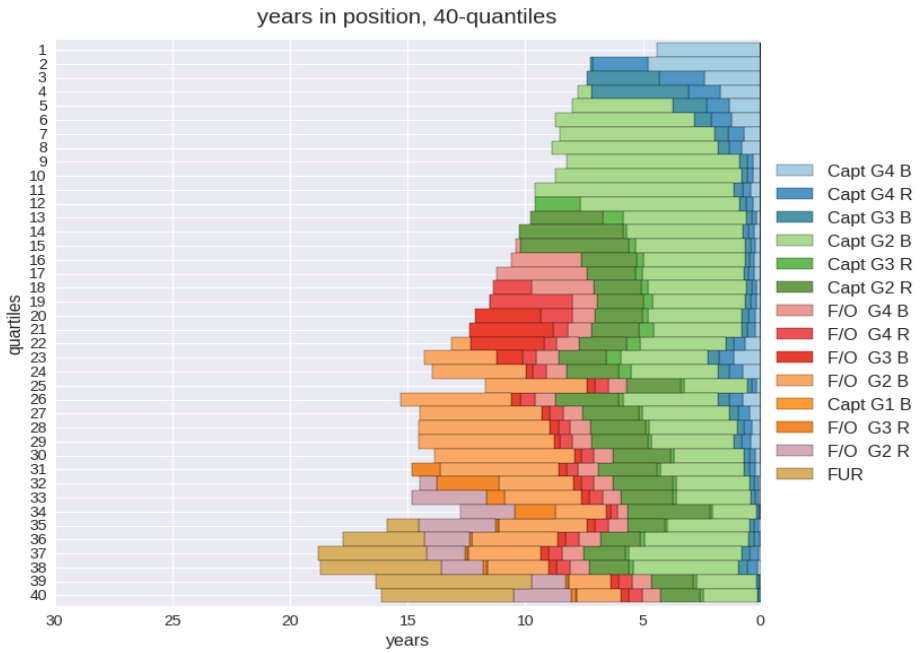


Fig. 37: single group years in position, 16 job levels[87]

The differences between quantile years in position may be studied as well.

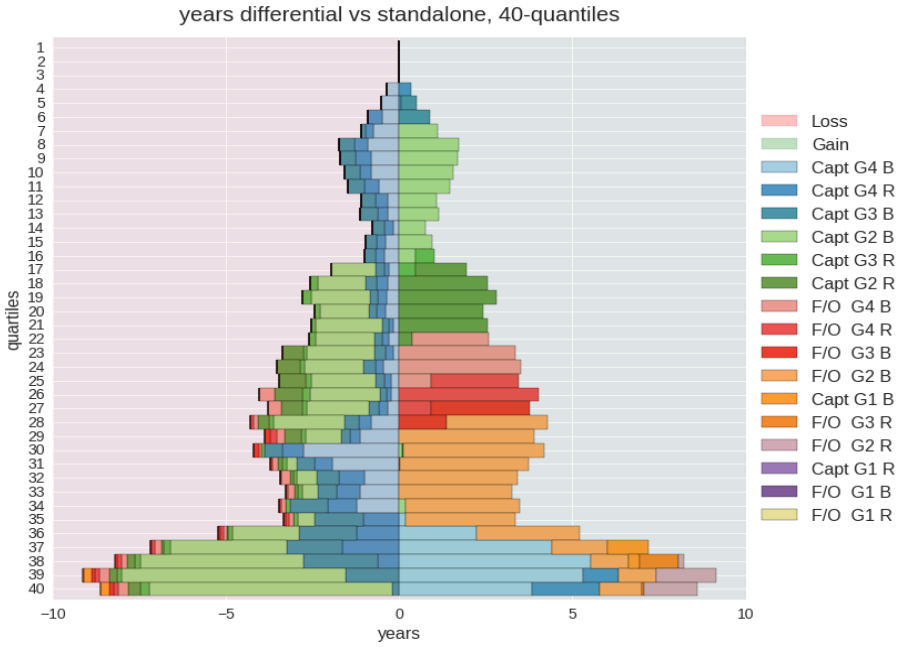


Fig. 38: years in position gain or loss vs. standalone, by quantile, 16 job levels[88]

[87] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_years_in_position
[88] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_years_in_position

The jobs available to each group will change when operating within an integrated list. The job transfer charts reveal how they will change over time.

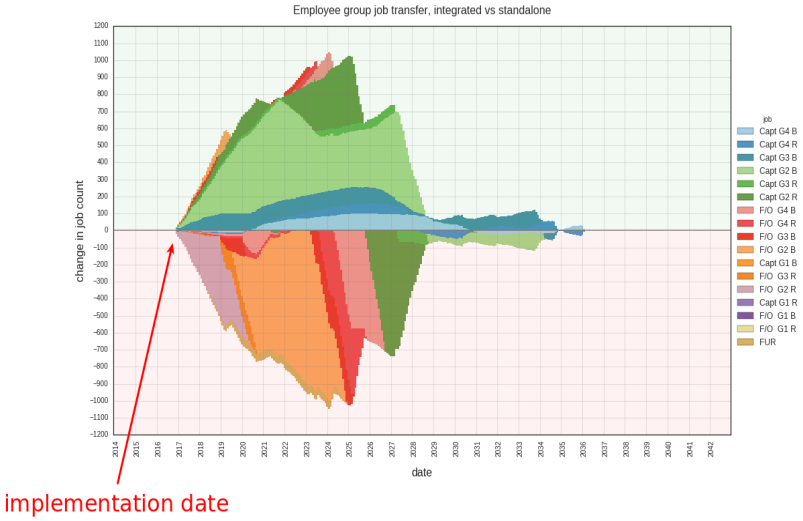


Fig. 39: separate group job count changes over time, integrated vs. standalone, delayed implementation date, 16 job levels[89]

[89] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_transfer

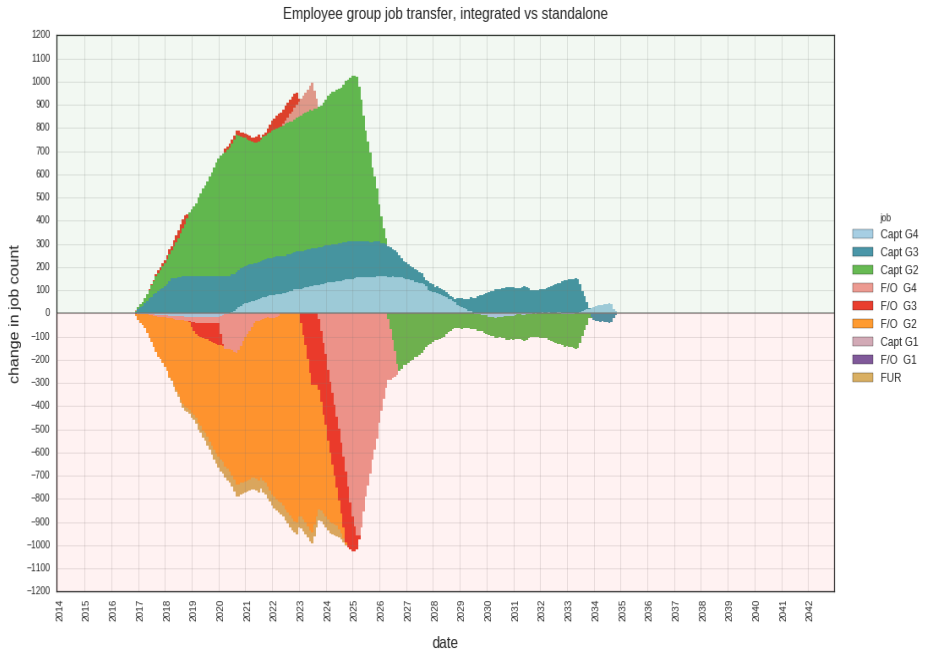


Fig. 40: same as above, with 8 job levels[90]

Closely related to the above information is an analysis of the change in *time* spent in each position per employee. This scatter chart displays months in position differential between models. Each dot represents a change in the number of months spent in a job level for those employees who do in fact experience a change. There may be multiple dots positioned vertically for the same employee, but the monthly gains and losses for the same employee will always total zero. The differential is indicated with the y axis, and the x axis represents employee percentile position within an integrated list, most senior to the right.

[90] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_transfer

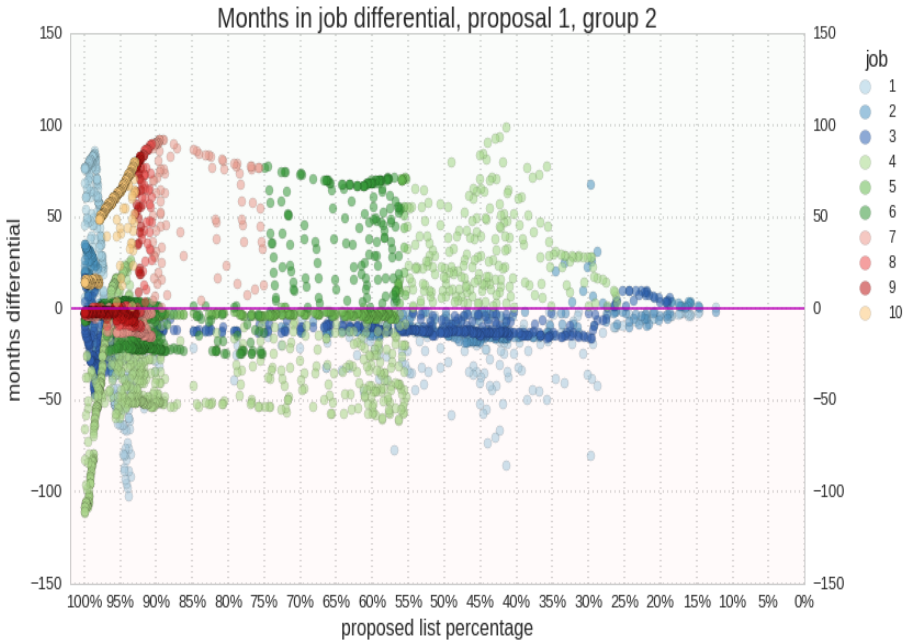


Fig. 41: time in job differential for job levels 1 through 10, indicating more time spent in lower level jobs overall (job colors above the line are generally a lower rank than below the line)[91]

Comparisons between proposals for individual employees are simple to perform. This is an example of the same employee under standalone and three other proposed integrated lists. The different paths are affected by no bump, no flush, a pre-existing special condition, and other prospective conditions.

[91] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_time_change

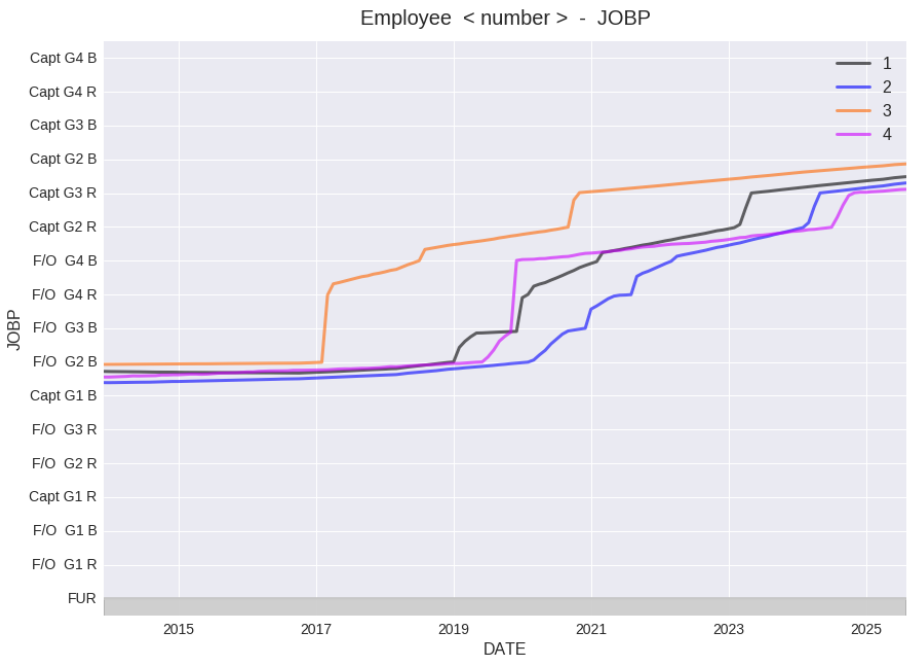


Fig. 42: career progression with various proposals, single employee, job levels[92]

Here is another example, this time measuring the different outcomes for another employee with the cat_order, or global job rank attribute. The outcomes diverge at the modeled implementation date in late 2016.

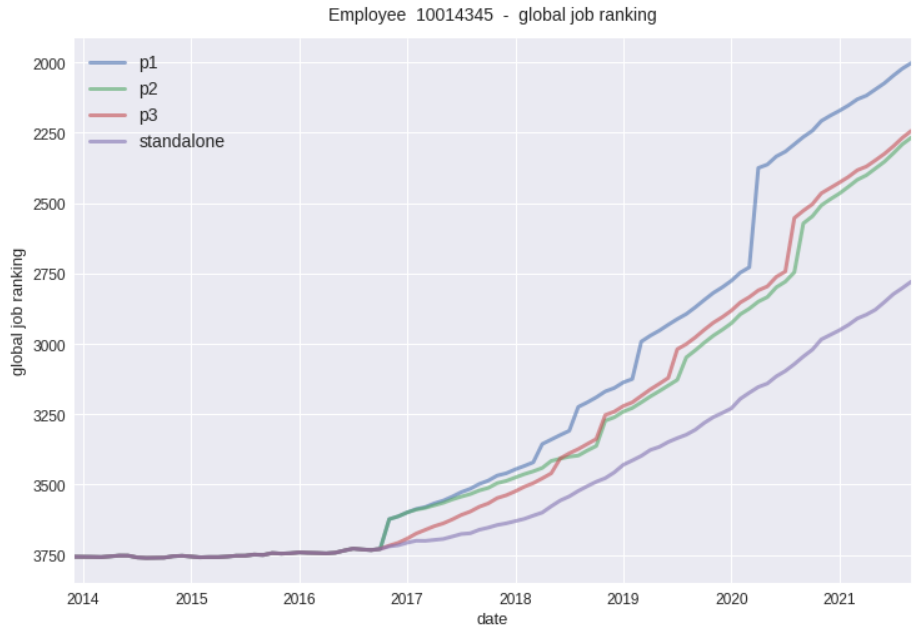


Fig. 43: career progression with various proposals, single employee, global job ranking[93]

[92] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.single_emp_compare

The next two charts indicate each employee's percentage on the list at retirement and the month in which it will occur, for a given proposal or standalone.

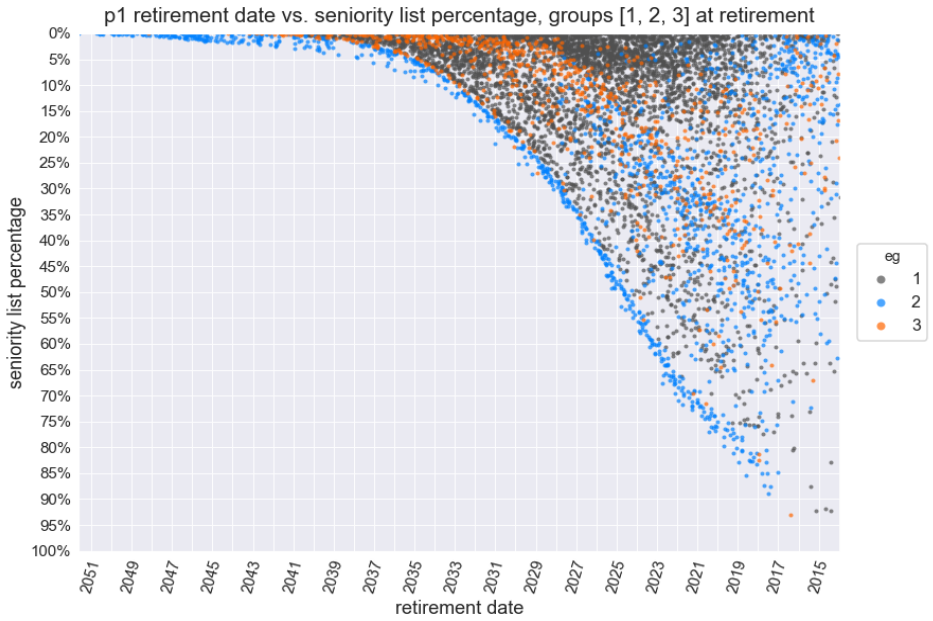


Fig. 44: list percentage at retirement, by group (x axis is month number)[94]

[93] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.single_emp_compare
[94] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

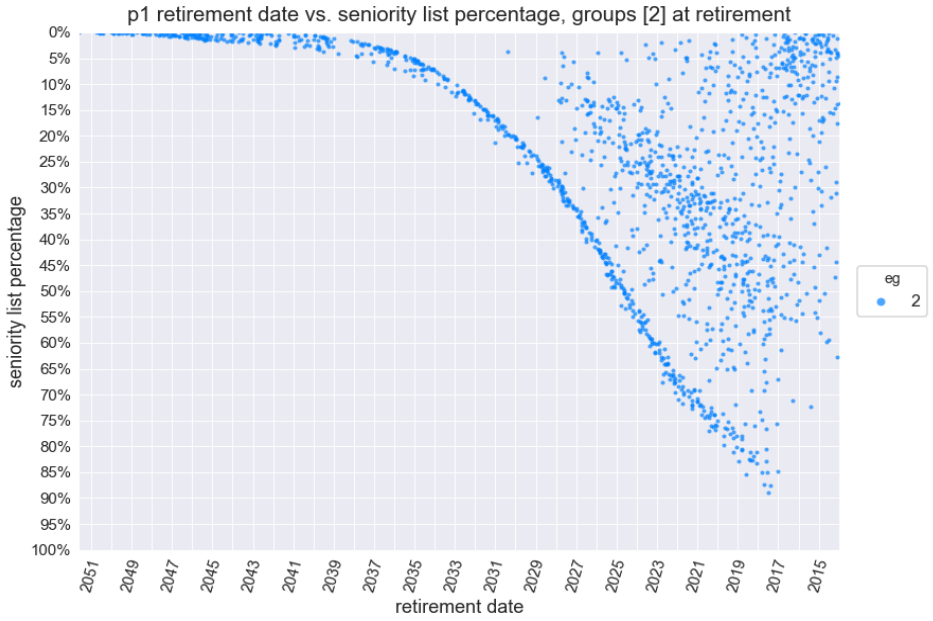


Fig. 45: list percentage at retirement, single group (2)[95]

Quantile membership lines and bands may be used to compare population percentage with attribute levels. The chart below displays the list percentage at retirement for members of group 2 who have a longevity date of 1999 or earlier with the group 1 proposal. The addition of quantile bands reveals that 50% of that group (right chart scale) will retire within the top 31% of the proposed integreated seniority list (left chart scale).

[95] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

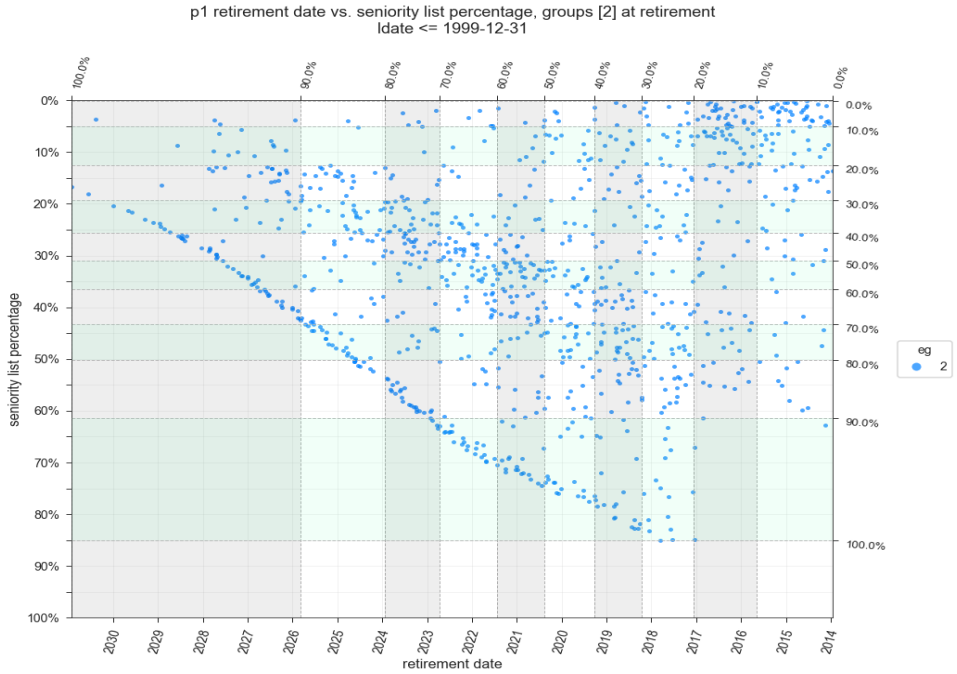


Fig. 46: list percentage at retirement with quantile bands, single group (2)[96]

Using the same conditions as above shows that group 1 fares much better, with 50% retiring within the top 5% of the proposed integrated seniority list. The quantile membership bands are available for any attribute comparison and may be set to correspond to the entire combined population or only to the displayed group(s).

[96] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

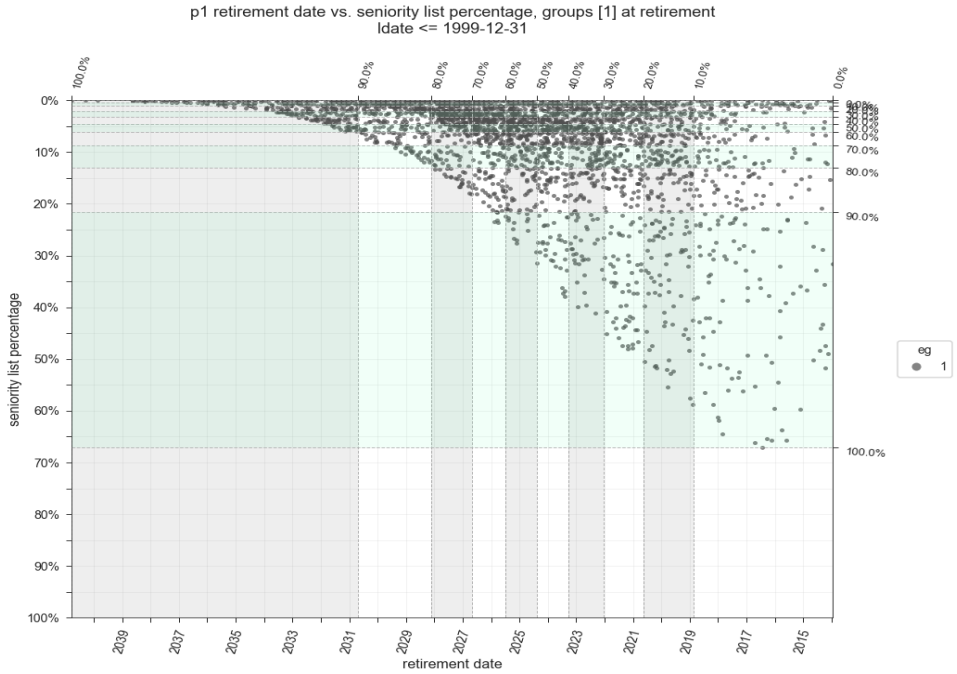


Fig. 47: list percentage at retirement with quantile bands, single group (1)[97]

This is a scatter differential chart which can compare several attributes. In this example, seniority percentage at retirement vs. proposed list order is represented for three separate groups.

[97] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.eg_attributes

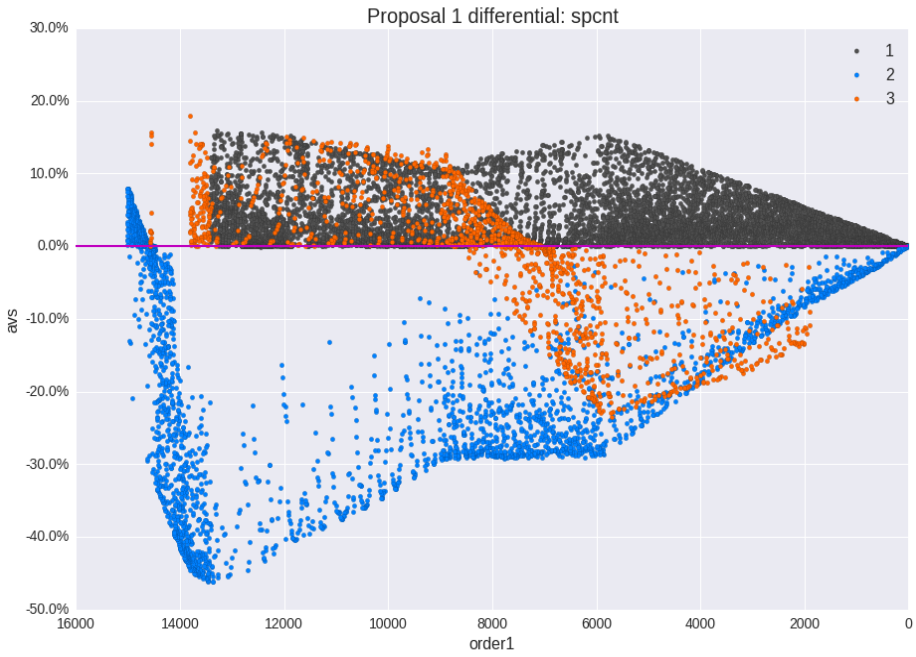


Fig. 48: group list percentage vs. standalone at retirement[98]

The same chart as above, with a polynomial fit applied. This helps to simplify the information and may be used with the editing tool.

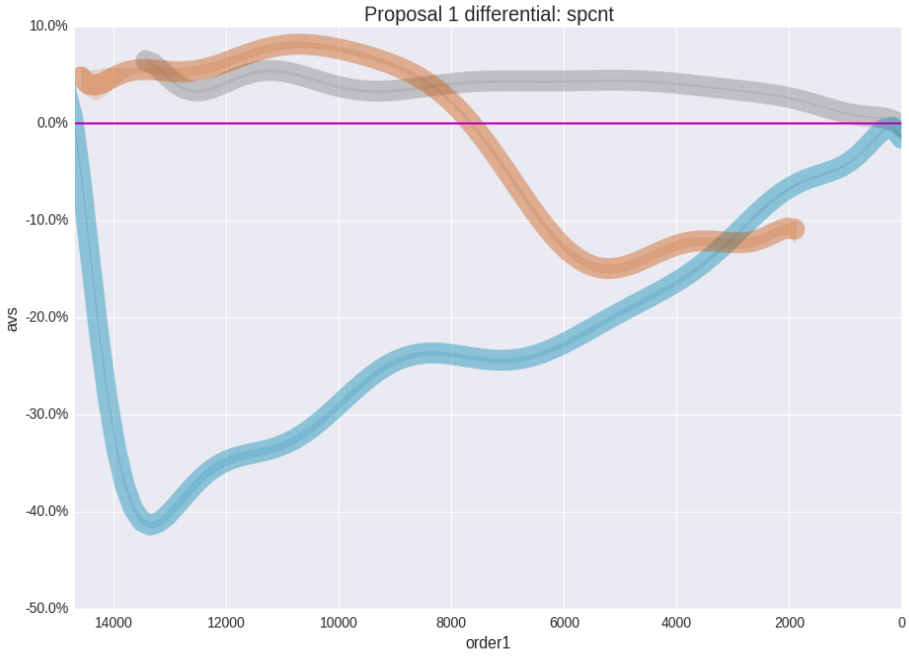


Fig. 49: polynomial regression applied. . . [99]

[98] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.differential_scatter

The two charts above had the x axis scaled to represent the proposal list order. The chart below organizes each group according to their native list percentage.

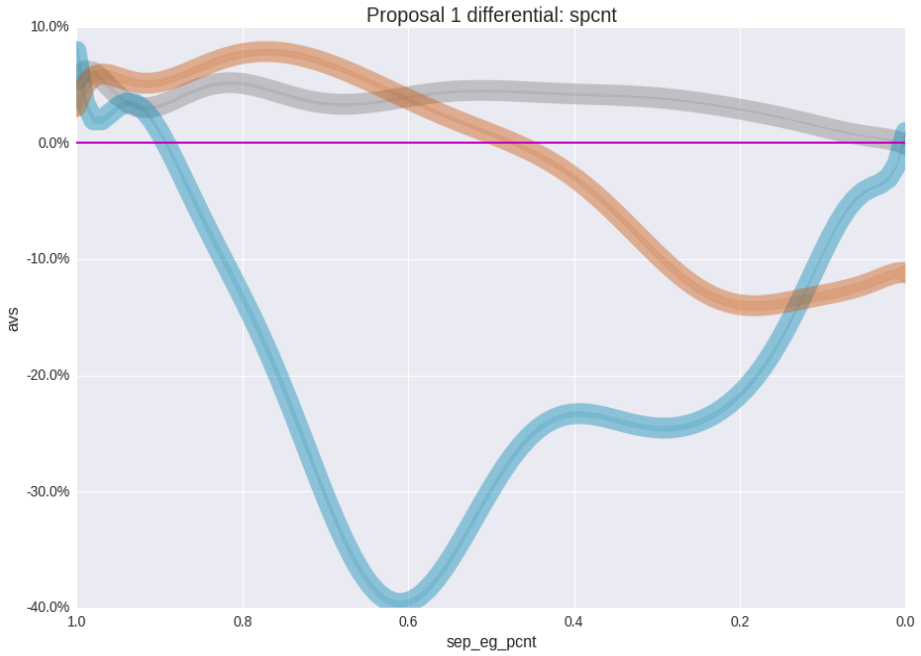


Fig. 50: same as above except using separate group native list percentage as x axis[100]

The next chart is related to the charts above. However, instead of showing results for each employee at retirement or monthly snapshop data, this chart indicates ranges of differential results over time. The bands of color within each plot represent the results for the same employee group under different list orderings and conditions. The chart below is showing data for group three under proposals one, two, and three.

[99] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.differential_scatter
[100] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.differential_scatter

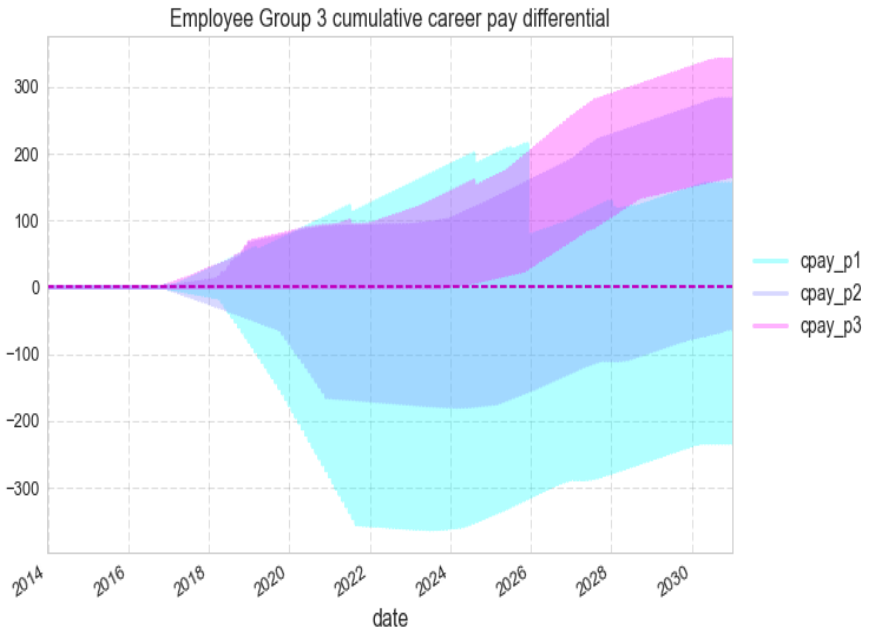


Fig. 51: the colored bands represent different proposals, not different groups (a negative number indicates a change toward the top or senior levels of the list)[101]

This type of study can look at other attributes and has an option to plot the mean of the data. Here is a job level differential chart. Note that as in the chart above, a negative number indicates an improvement to a higher level job (the best jobs have the lower job level numbers).

[101] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.diff_range

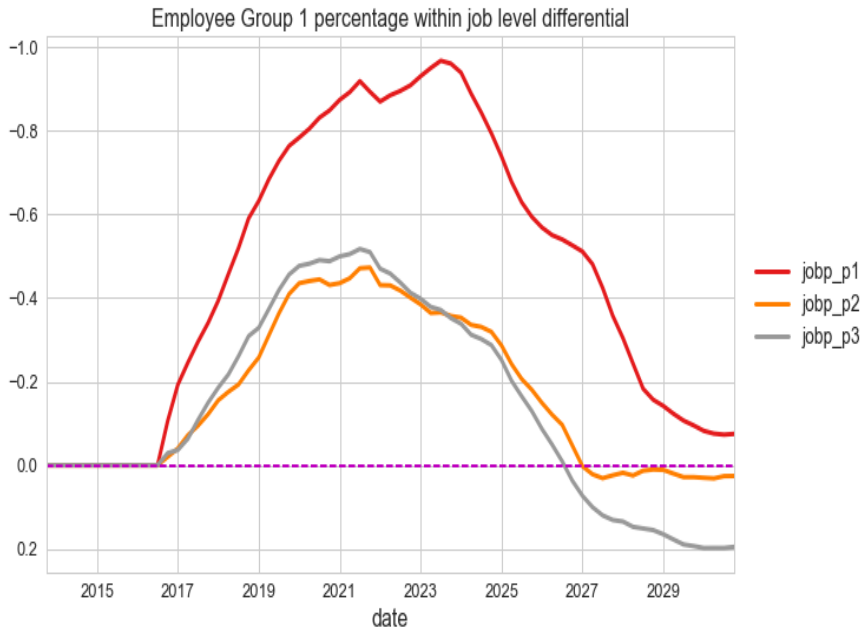


Fig. 52: average job transfer between the employee groups with various proposals[102]

This is a time-series chart with job level bands and a headcount line. The headcount line indicates the extent of the remaining affected employees (present at the time of the merger) each year. The job level bands are responsive to the model fleet change inputs within the configuration file.

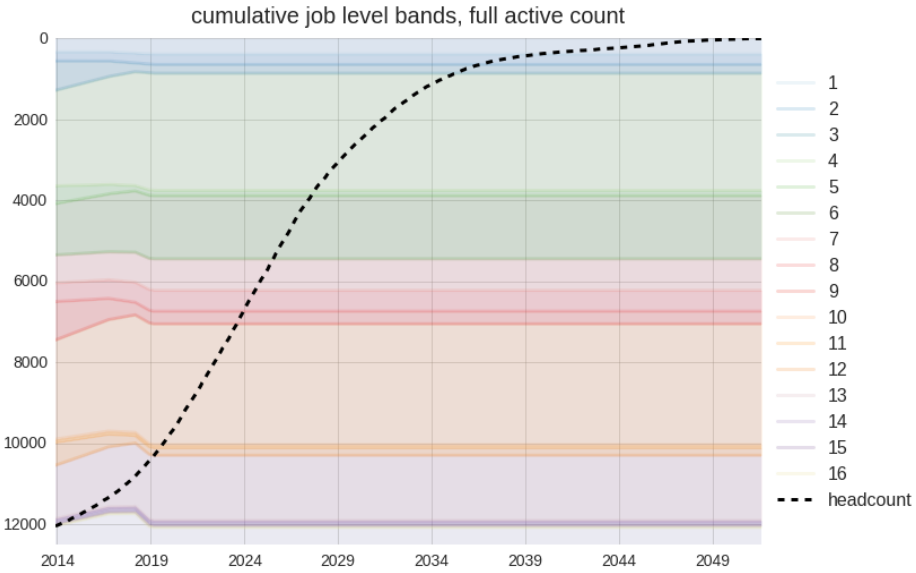


Fig. 53: job level bands over time (this example indicates fleet changes until 2019)[103]

[102] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.diff_range

The next two charts show how seniority list percentage does not always equate to the same job bidding capability due to no bump, no flush and other conditions. The chart below includes three employees placed next to each other on an integrated proposal with an implementation date in late 2016. After that point, the three lines representing career progression as list percentage are superimposed.

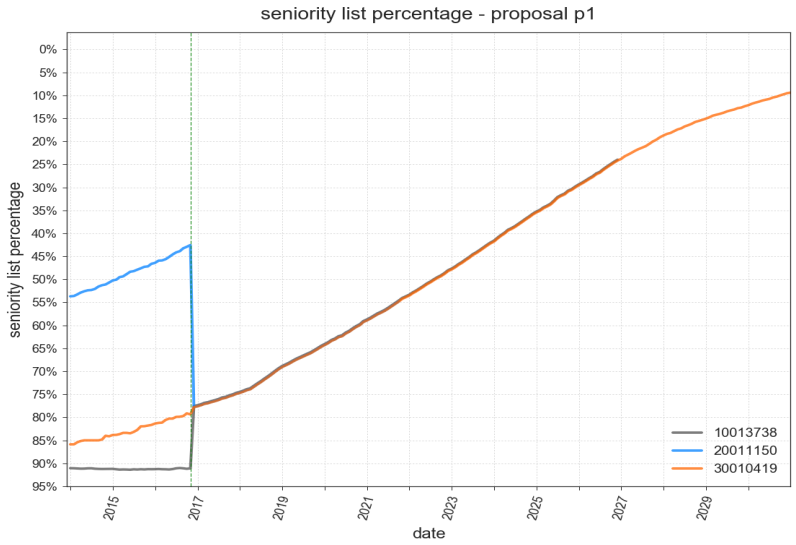


Fig. 54: example career percentage, 3 separate group employees (superimposed, nearly identical percentage over time)[104]

This chart reveals a more accurate model of what would occur in terms of jobs available to these three employees. The thin vertical dashed line represents a modeled implementation date. Each group operates independently until that time. The black line represents an employee with a pre-existing special condition, allowing the large jump in job levels. The blue line represents the employee who holds a higher ranked job at implementation which is protected until his retirement. Employees holding a job due to no bump, no flush protection or special job assignment rights remain in that job until his list partners from other group(s) "catch up". Once the employees have reached the point in time where the same bidding opportunities exist for all three, they then move together in terms of list percentage, if they have not already retired.

The program model accurately accounts for all of these conditions.

[103] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_count_bands
[104] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.multiline_plot_by_emp

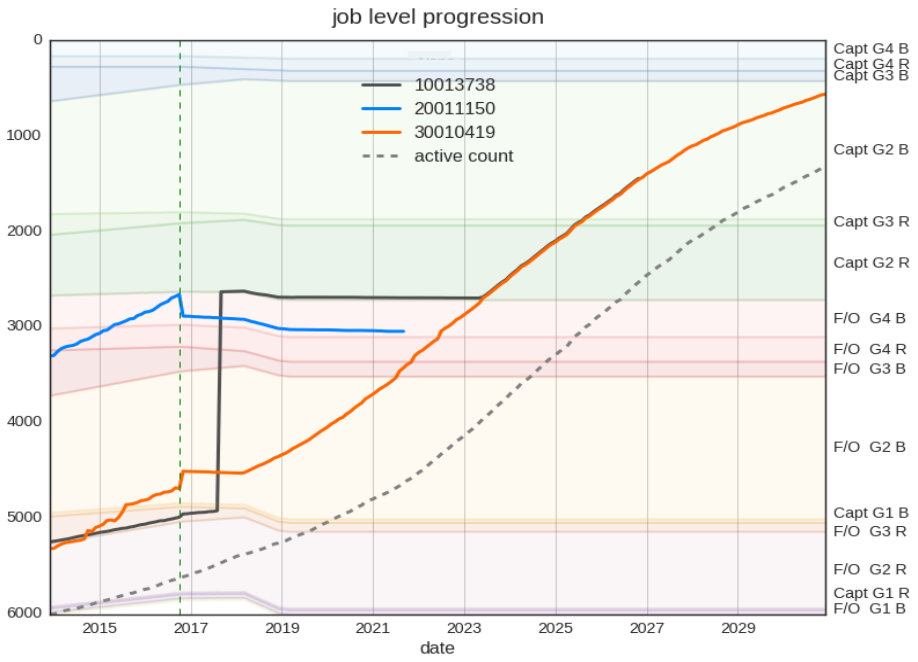


Fig. 55: same employees and list as above, career progression through job levels with effects of special conditions and no bump no flush. y axis is arranged by job order number. . . [105]

This type of chart may display selected groups of employees as well. This chart is showing projection for workers from one of the merging groups who have special job assignment quotas which pre-exist the merger. Each line represents the modeled career path for an individual worker.

[105] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_level_progression

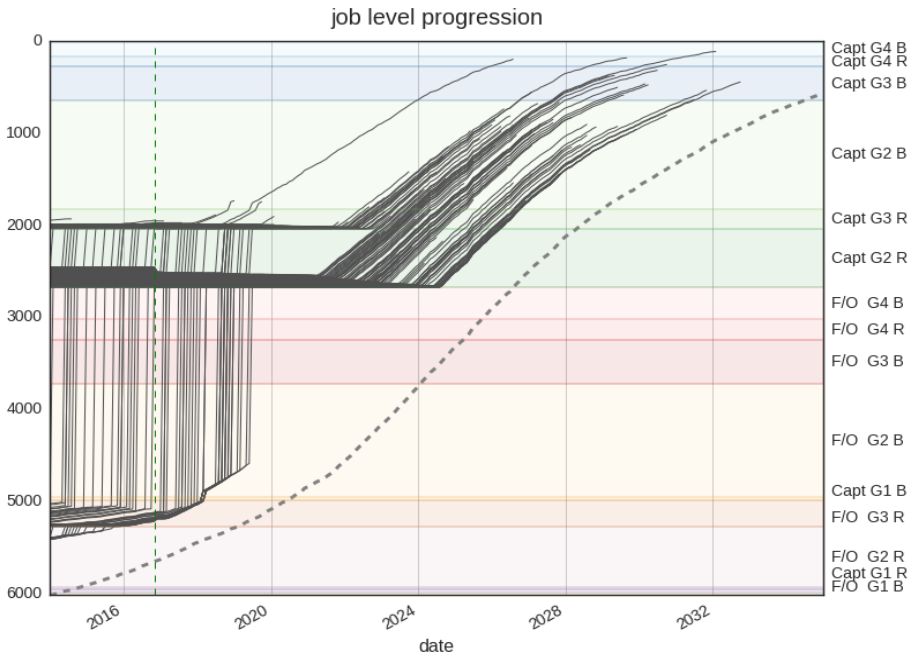


Fig. 56: a sampling of job progression plots belonging to a subset group with pre-existing job assignment quotas. Note the vertical lines representing a jump up to a level to satisfy a quota.[106]

Another built-in chart type available with the seniority_list program is the quantile-groupby chart. Initial lists from each employee group may be segmented into equal-sized segments (quantiles) and the metrics associated with the employees belonging to those segments may be analyzed in various ways over time. This method provides information concerning the career progression experience of stratified sections of each employee group over many different metrics. The chart below is displaying the job category ranking results for an employee group split into 40 quantiles, or 2.5 percent bands, for a standalone (unmerged) employee group. The data shown here represents the results for the last employee within each quantile. Other methods, such as quantile average or median are also available for display.

[106] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_level_progression

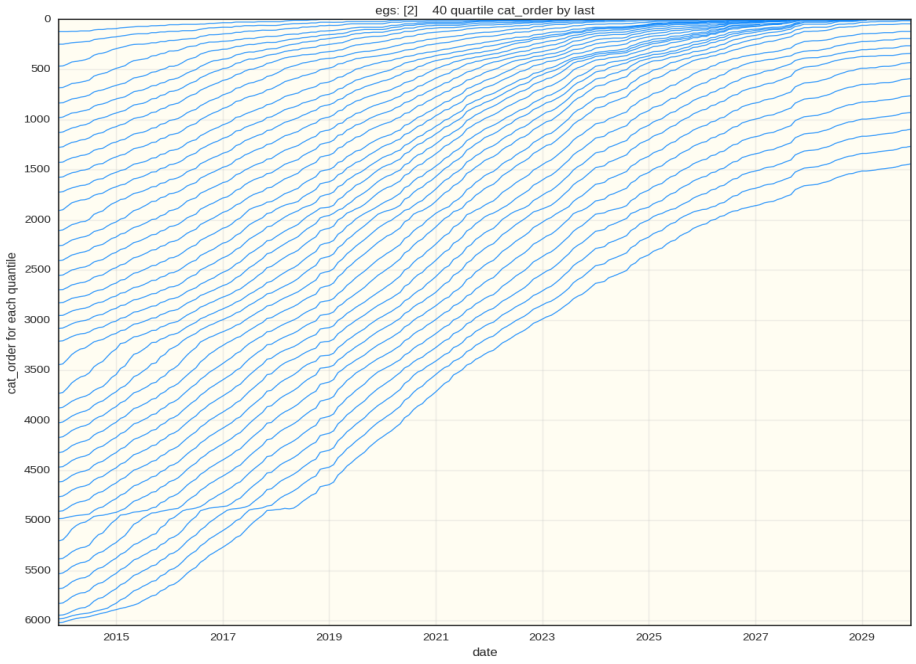


Fig. 57: data representing the job category ranking of the last employee within each of 40 quantiles from the initial employee group population, standalone scenario.[107]

Here are the results for the same employee group when combined with other employee groups using a proposed integration list and conditions. The results following the implementaion date in late 2016 indicate much lower job opportunities and long-term job level stagnation for this workgroup. This charting function is capable of measuring other data such as pay, list percentage, and jobs held over the life of the data model.

[107] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_groupby

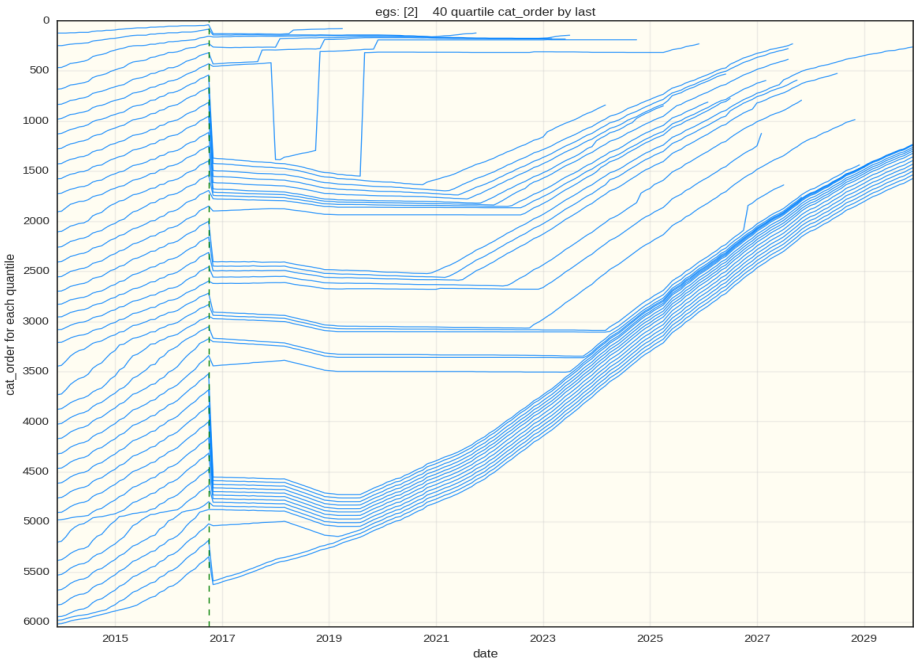


Fig. 58: data representing the job category ranking of the last employee within each of 40 quantiles from the initial employee group population, integrated scenario.[108]

When analyzing quantile groupings with the job category attribute, it is possible to show integrated job level zones in the chart background. Here is the same analysis as above with the addition of job bands.

[108] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_groupby

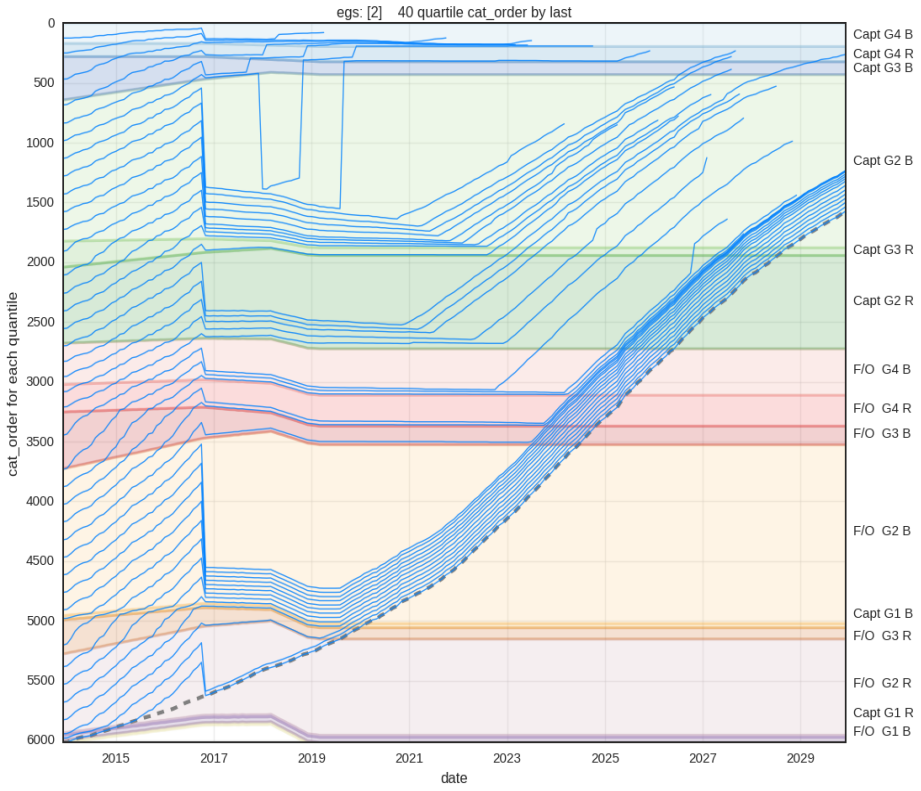


Fig. 59: accurately proportioned job levels assist in understanding employee group career progression.[109]

The plot line colors may be pulled from a customized matplotlib colormap when plotting a single employee group, which adds further qualitative insight to this type of analysis. This example clearly reveals an employee group overwhelmingly disadvantaged with a proposed integration. While the employees are protected with "no-bump, no_flush" provisions (cannot normally be displaced from a job held at time of integration), due to poor placement within an integrated list, this group is relegated to the bottom sections of each of the job levels for quite some time with greatly diminished career advancement opportunities.

[109] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_groupby

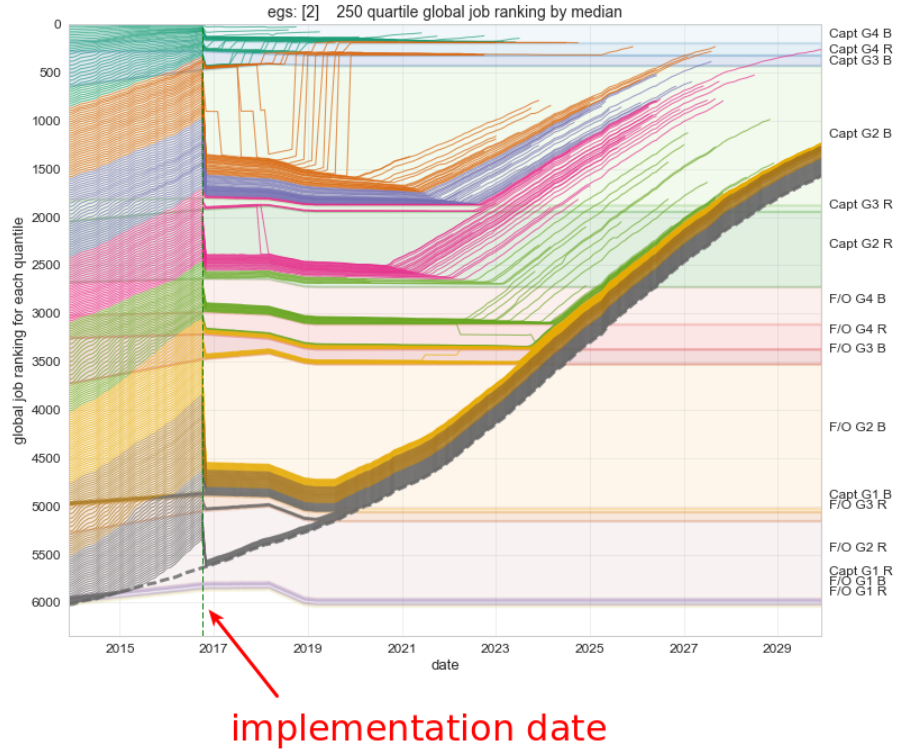


Fig. 60: color spectrum line plots offer additional insight concerning the short- and long-term effects of proposed integrated list scenarios.[110]

Quantile progression for the same employee group under different integrated list proposals may be directly compared by plotting the output from two different data models within the same chart. In the following example, the lines represent median job value ranking for employees grouped into 10 population quantiles. The progression lines diverge after the implementation date, with the solid lines representing standalone progression and the dashed lines representing the progression of the same quantile groups under a selected integration proposal. Clearly the selected proposal would cause major career stagnation and disruption for the employee group represented here.

[110] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_groupby

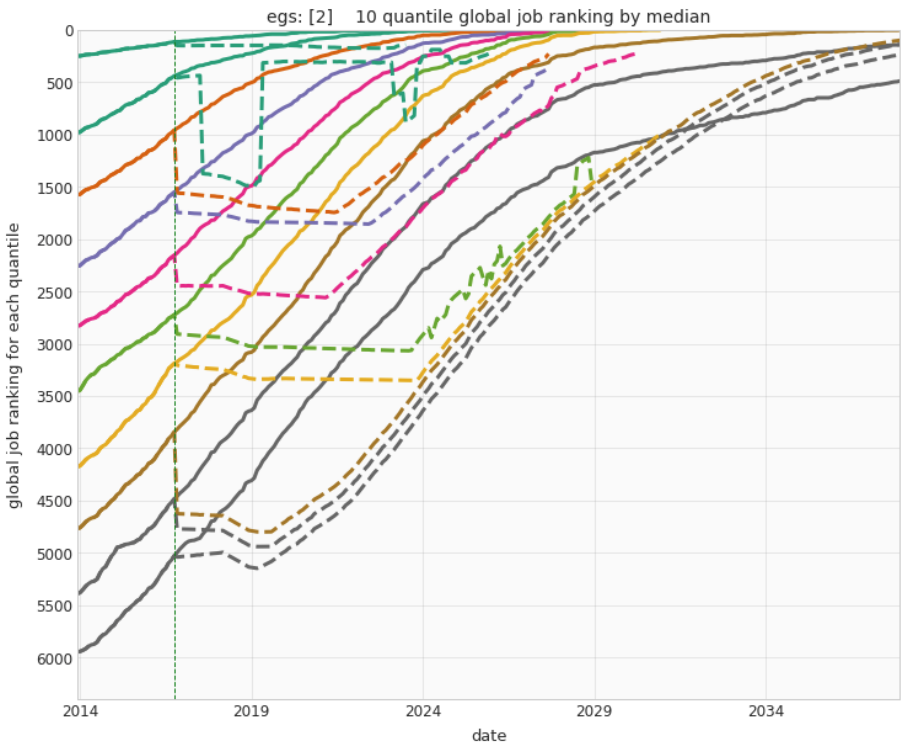


Fig. 61: dataset comparative line plots reveal differences in career progression for the same employee group under 2 different integrated list scenarios. The up and down trace for quantile group 2 is a result of applying a conditional job assignment rule with the integrated list proposal.[111]

> This chart represents separate group colored job (jnum) bands. The y axis represents the employee group count. The slight variations in job band thickness are due to modeled fleet changes. Employee career advancement tracks would all be contained within these bands, passing upward and to the right as employees age and become more senior.

[111] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_groupby

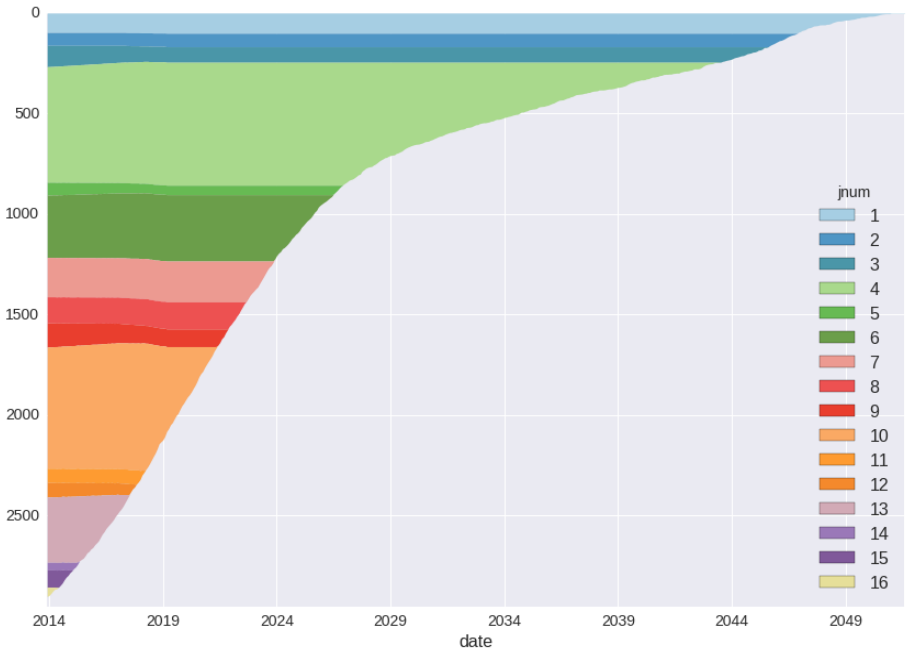


Fig. 62: separate group job bands pre-integration[112]

Here are the job bands for the same separate group as in the above chart, as they are affected when a combined list proprosal is applied. The horizontal section at the left of the chart reflects a delayed implementation date.

[112] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_count_bands

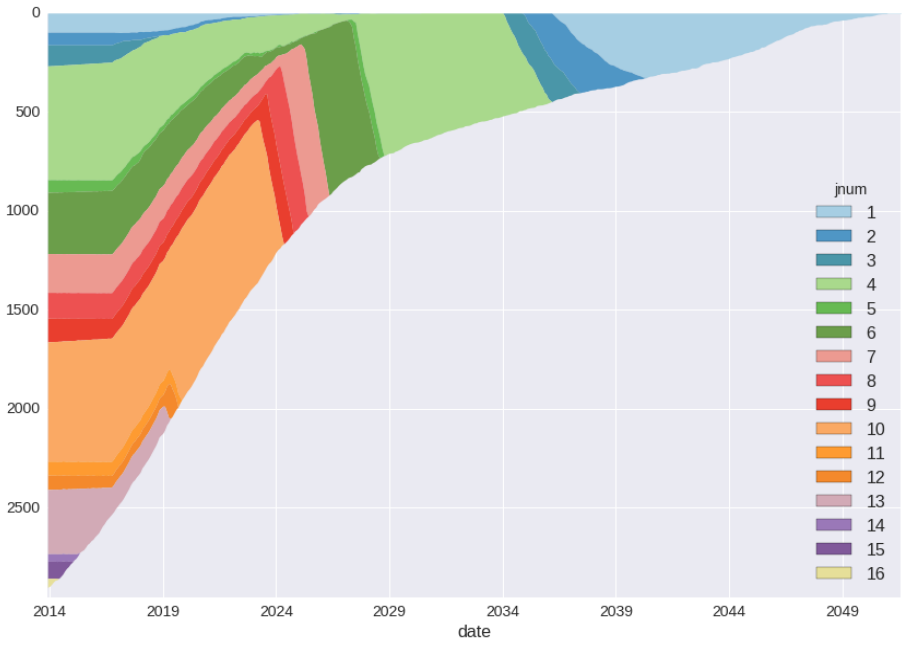


Fig. 63: separate group job bands post-integration (delayed implementation)[113]

This is a slightly different format of the same chart above with the addition of a sample career advancement track. In this scenario, the job bands "move up" almost at exactly the same pace as the example career track, meaning little to no job advancement for the sample employee for many years.

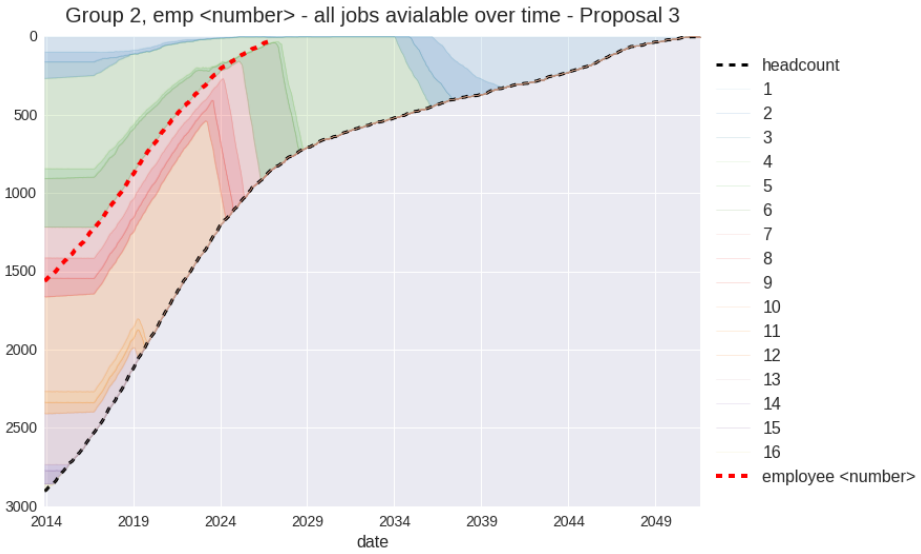


Fig. 64: example career advancement through post-integration job bands[114]

[113] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_count_bands
[114] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_count_bands

Here is an example of a separate group's job bands under a proposal which allows more rapid advancement. (Notice the higher band levels reaching lower with time.)

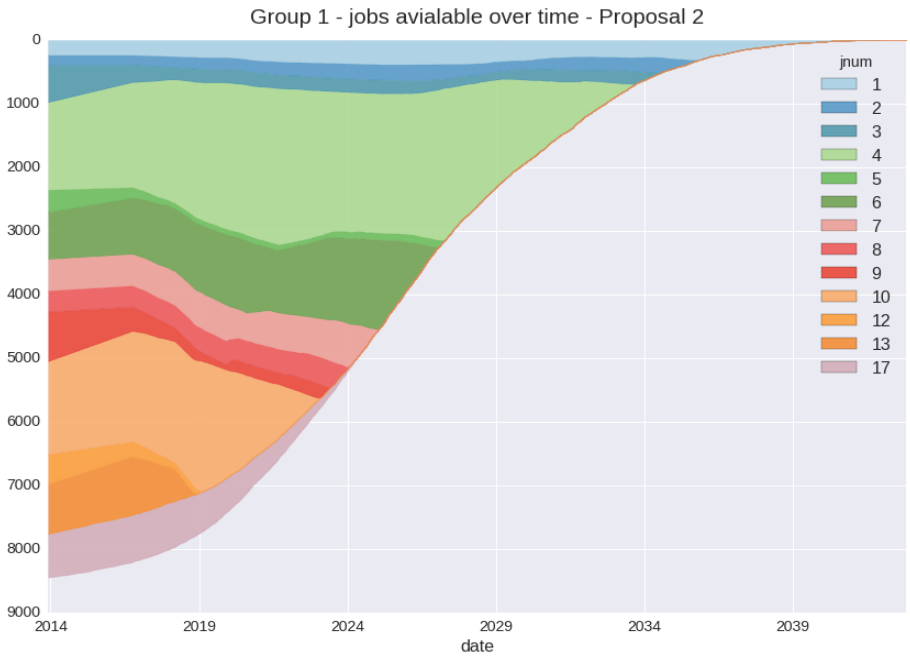


Fig. 65: separate group post-integration job bands indicating overall improvement with furlough recall[115]

The following quantile change charts indicate relative position change only and do not directly represent potential job positions due to the effect of no bump, no flush considerations and other special conditions. These conditions are reflected in the "job band" type charts above. The quantile charts may also be diplayed as percentage band charts.

To use: compare the underlying quantile and year (square grid) with the resultant overlying colored grid level as indicated by the legend.

[115] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.job_count_bands

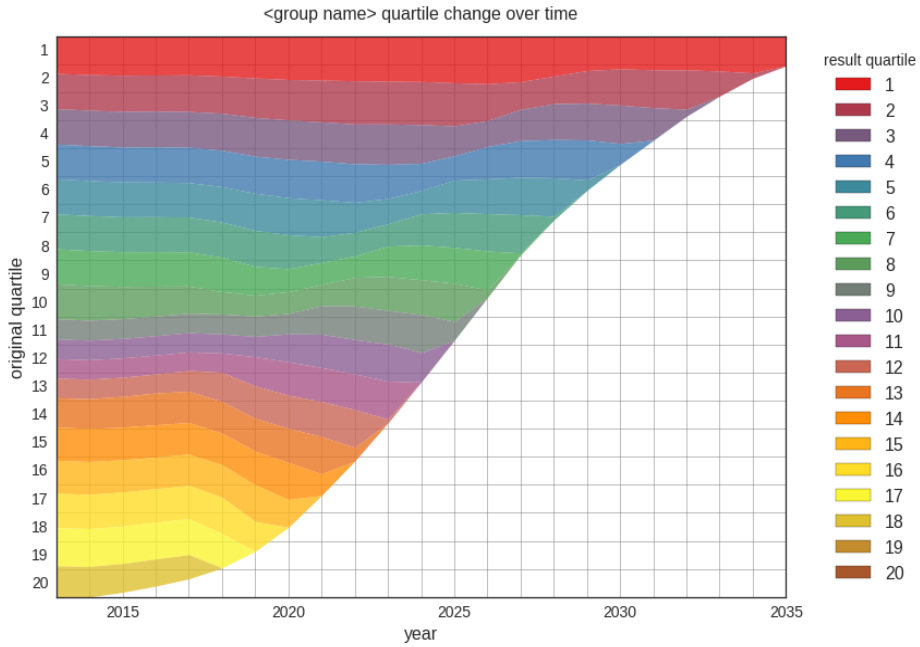


Fig. 66: separate group post-integration quantile change over time indicating improvement for group[116]

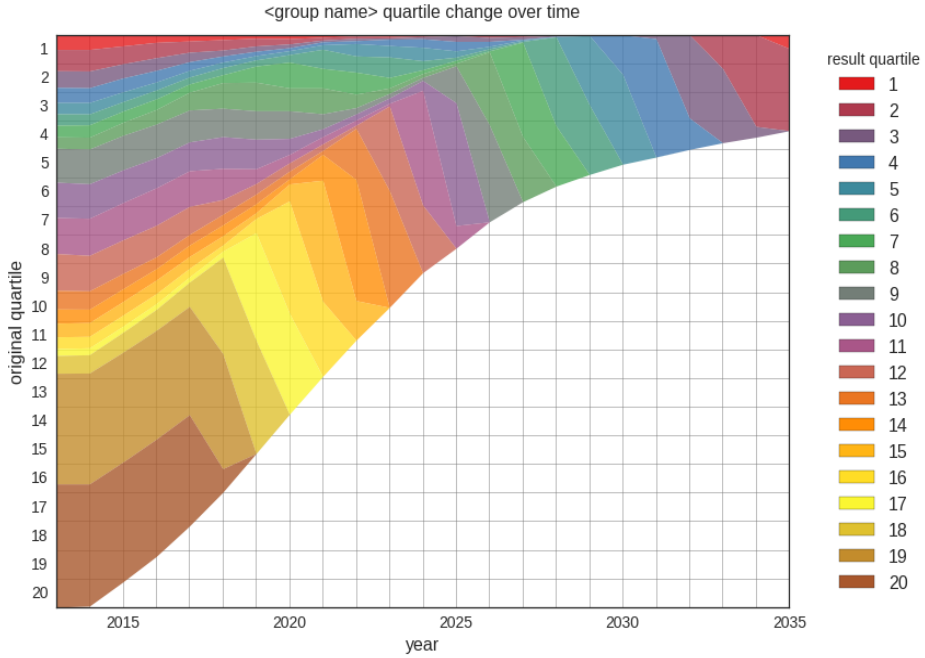


Fig. 67: separate group post-integration quantile change over time indicating large distortion and loss for group[117]

[116] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_bands_over_time
[117] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_bands_over_time

The slight change in the quantiles representation for a standalone group below is due to a modeled change in the number of jobs available to that standalone group over time.

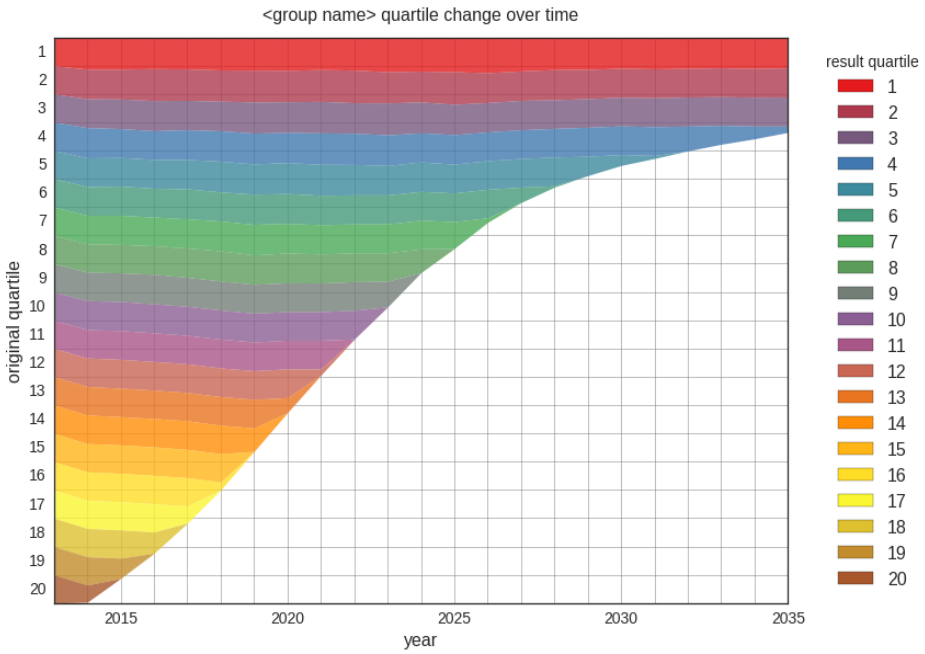


Fig. 68: separate group pre-integration quantile change over time - same group as previous chart. . . before operating within combined list[118]

8.2 editor tool

The editor is an interactive tool which allows list adjustments to be made and recalculated results to be viewed within seconds. The display includes a main display chart and a horizontal stripplot. The main chart may display a comparative attribute differential between two integration proposals or display absolute (actual) attribute values for a single integrated list proposal. Attributes such as list percentage, job levels, or career earnings values may be displayed for any or all employee groups upon reaching retirement or for any selected month. Comparisons may be made between proposals or with standalone data. The tool also includes a display filtering feature, allowing further analysis of targeted subgroups, such as employees with high longevity or within a specified job value range.

The editor tool was designed to easily identify outcome equity distortions associated with various proposals and to permit simple yet precise corrective editing. Distortions

[118] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.quantile_bands_over_time

are identified through intuitive data visualization and corrections are made using the interactive editor controls and special program algorithms.

To make a corrective edit, the user positions two vertical cursors (lines) on either side of the target section using the "edit zone" range slider. Then after selecting the appropriate employee group, style of adjustment, and direction, a "squeeze" is performed which has the effect of sliding the target group up or down the proposed list (within the "edit zone") while maintaining relative list order within each group. The results of the move are calculated and displayed for further analysis and adjustment if required. The stripplot offers a visualization of the density distribution of the groups within the proposed list order after the squeeze process but before the recalculation. All recalculations include all of the conditions in the overall model.

The following screenshot of the editor shows it in scatter mode and loaded with a list percentage at retirement differential chart described earlier.

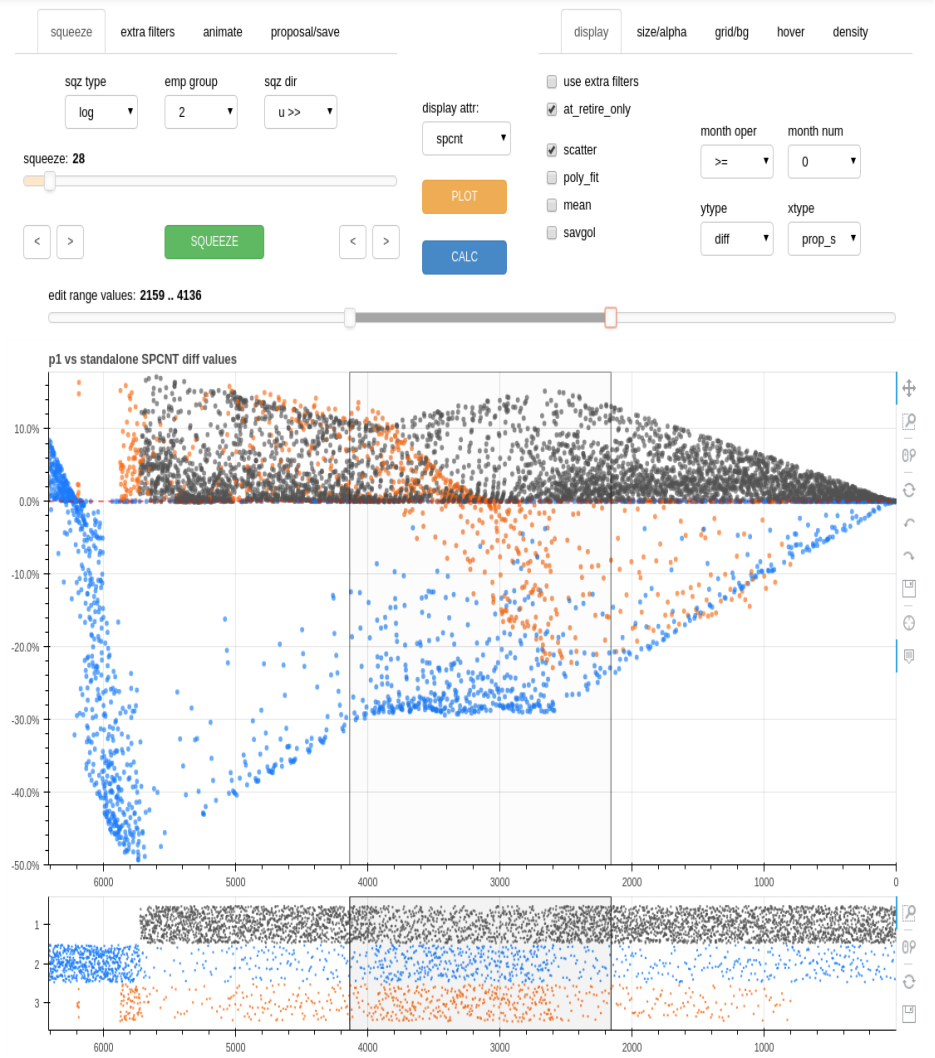


Fig. 69: Editor tool with interactive sliders and other selectors. . . [119]

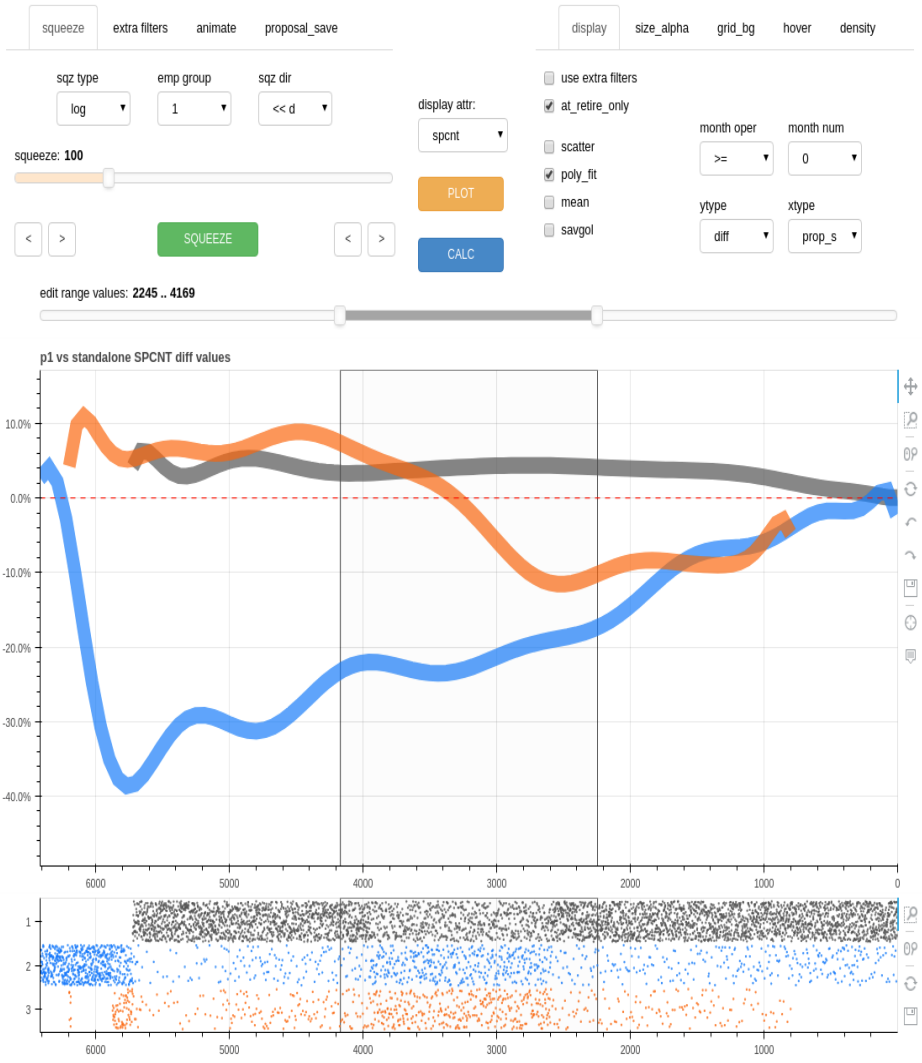


Fig. 70: Same as above, except utilizing the polynomial fit option.[120]

[119] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.editor
[120] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.editor

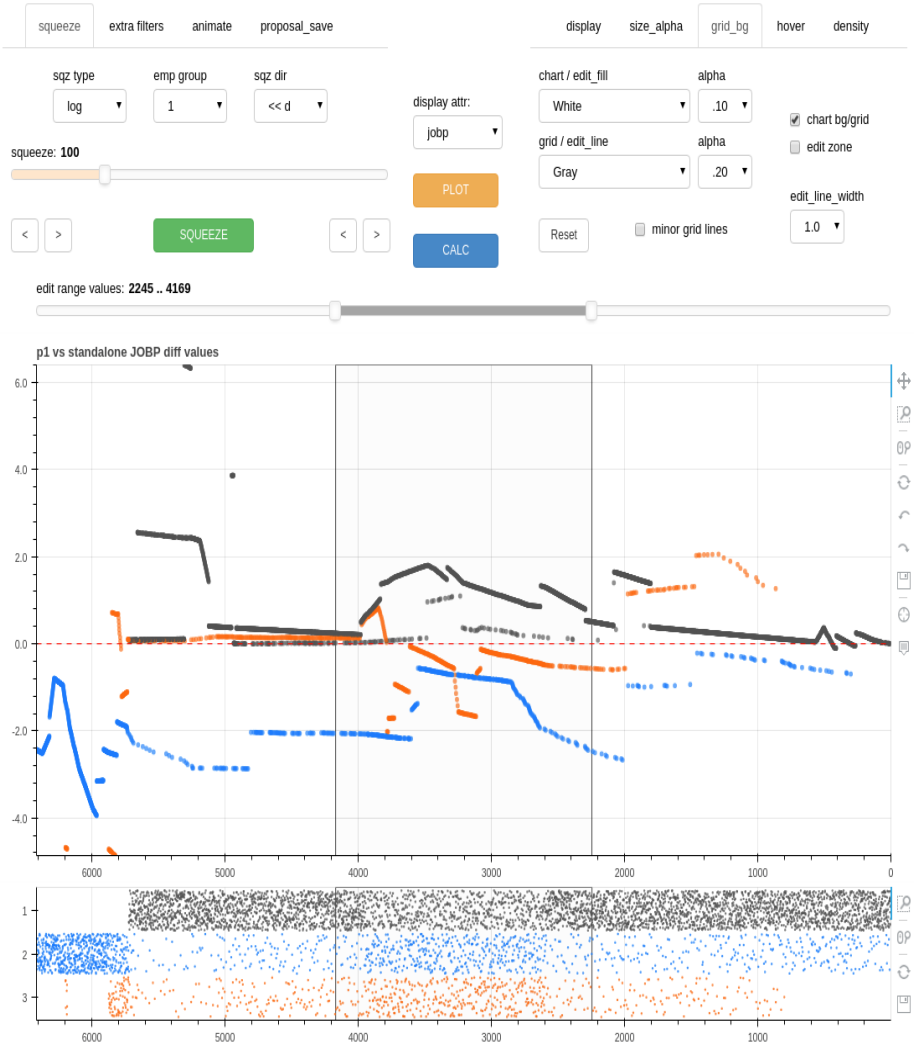


Fig. 71: Example of a job level comparison for a future month.[121]

[121] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.editor

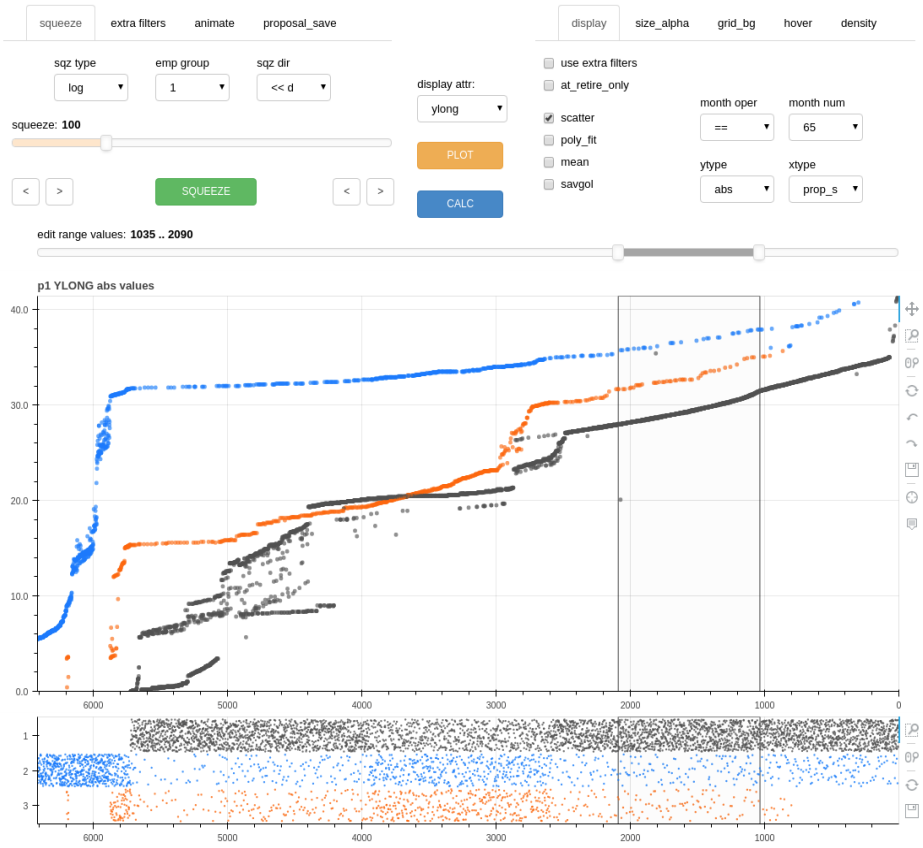


Fig. 72: Example of actual (not differential) year longevity attribute for a future month.[122]

[122] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.editor

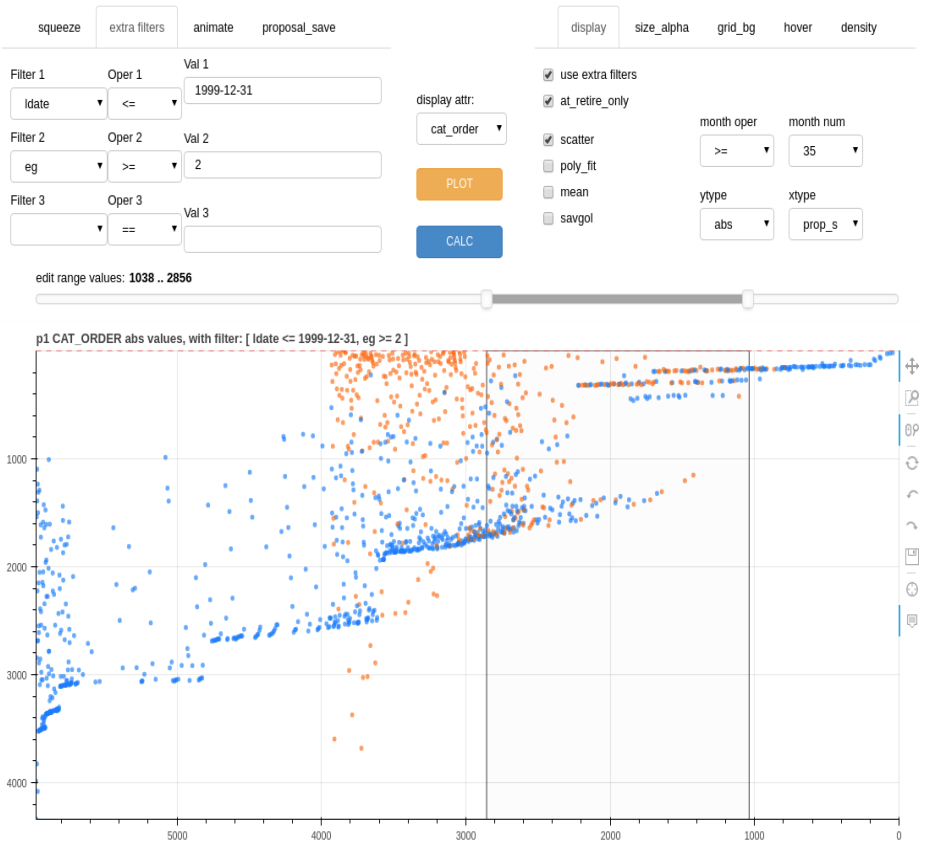


Fig. 73: Abosolute value filtered display, proposal "p1" cat_order (job value) at retirement for groups 2 and 3 with a longevity date of 1999 or earlier.[123]

[123] http://rubydatasystems.com/matplotlib_charting.html#matplotlib_charting.editor

CHAPTER

NINE

CONVERTER MODULE

converter.**convert**(*job_dict=None*, *sg_list=None*, *count_ratio_dict=None*, *ratio_dict=None*, *ratio_onoff_dict=None*, *count_onoff_dict=None*, *dist_sg=None*, *dist_ratio=None*, *dist_count_ratio=None*)

Convert data relating to job assignment conditions from basic job level inputs to enhanced job level inputs

Inputs are the basic job level values for the various conditions, the job dictionary, and the distribution methods used during conversion.

This function is called within the build_program_files script when the "enhanced_job" key value within the settings dictionary is set to "True".

inputs

job_dict (dictionary) case_specific jd variable. This input contains full-time job level conversion percentages

sg_list (list) case-specific sg_rights variable

ratio_list (list) case-specific ratio_cond variable

ratio_dict (dictionary) dictionary containing ratio condition data

count_ratio_dict (dictionary) dictionary containing all data related to a capped ratio or count ratio condition

dist_sg, dist_ratio, dist_count (string) options are: 'split', 'full', 'part'

determines how jobs are distributed to the enhanced job levels.

'split' - distribute basic job count to full- and part-time enhanced job levels according to the ratios set in the job dictionary (jd) variable

'full' - distribute basic job count to corresponding enhanced full-time job levels only

'part' - distribute basic job count to corresponding enhanced part-time job levels only. This option could be selected if the employees with special job rights are

placed in a relatively low position on the integrated list, eliminating the option of obtaining a full-time job position

The distribution type for each condition input is independent of the other condition distributions.

If these variables are not assigned, the program will default to “split”.

CHAPTER

TEN

EDITOR_FUNCTION MODULE

bokeh_editor.py

EDITOR TOOL

requires bokeh 0.12.13+ - uses bokeh server

class editor_function.**Data**(*data=None*)
Bases: object

update_data(*d*)

class editor_function.**Kwargs**(*kdict=None*)
Bases: object

add(*key*, *value*)

clear()

remove(*key*)

update(*other_dict*)

class editor_function.**PropOrder**(*list_order=None*, *name=None*)
Bases: object

update_name(*new_name*)

update_order(*new_order*)

editor_function.**alpha_list**()
provides a list of string decimals for editor grid_bg tab alpha selectors

editor_function.**color_list**()
provides a list of string color names for editor grid_bg tab color selectors

editor_function.**editor**(*doc*, *poly_dim=15*, *ema_len=25*, *savgol_window=35*, *savgol_fit=1*, *animate_speed=350*, *plot_width=1100*, *plot_height=500*, *strip_eg_height=50*, *start_dot_size=4.75*, *max_dot_size=25*, *start_marker_alpha=0.65*, *marker_edge_color=None*, *marker_edge_width=0.0*)

create the editor tool

use the following code to run within the notebook:

```
import editor_function as ef
from functools import partial

from bokeh.io import show, output_notebook
from bokeh.application.handlers import FunctionHandler
from bokeh.application import Application

output_notebook()

handler = FunctionHandler(partial(ef.editor,
                          # optional kwargs,
                          ))

app = Application(handler)
show(app)
```

inputs

doc (variable) a variable representing the bokeh document, do not modify

poly_dim (integer) the order of the polynomial fit line

ema_len (integer) the smoothing length to use when constructing the exponential moving average line

savgol_window (positive odd integer) Savitzky-Golay filter window length

savgol_fit (integer) The order of the polynomial used to fit the samples. This value must be less than the savgol_window value.

animate_speed (integer) Number of milliseconds between each animated month display

plot_width (integer) width of main and density charts in pixels

plot_height (integer) height of main chart in pixels

strip_eg_height (integer) height alloted for each employee group when constructing the density chart

start_dot_size (float) initial scatter marker size for main chart

max_dot_size (integer) maximum scatter marker size for the main chart display, set to size sliders

start_marker_alpha (float) initial scatter marker alpha (transparency) for main chart display

marker_edge_color (color value string or None) color of scatter marker edge color for main chart when marker edge width value is greater than zero

marker_edge_width (float) width of scatter marker edge width when marker_edge_color is not None

editor_function.**line_widths**()
provides a list of string decimals for editor grid_bg tab edit line width selector

editor_function.**make_dataset**(*proposal_name=''*, *df_order=None*, *conditions=[]*, *ds=None*, *ds_stand=None*)

editor_function.**use_first_proposal_found**(*proposal_name*)
find and return the first list order found in 'dill/proposal_names.pkl'. This function is used when another proposal name is designated by another section of the program but does not exist.

inputs

proposal_name (string) the name of the proposal which was not found

CHAPTER

ELEVEN

FUNCTIONS MODULE

The functions module contains core program routines related to building and working with the data model and associated files. General definitions: dataset "month_form" is length n months in model "short_form" data has a length equal to the number of employees "long_form" data is the length of the cumulative sum non-retired employees for all months in the data model (could be millions of rows, depending on workgroup size and age)

functions.**add_zero_col**(*arr*)

Add a column of zeros as the first column in a 2d array. Output will be a numpy array.

example:

input array:

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

output array:

```
array([[ 0,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [ 0, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [ 0, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [ 0, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [ 0, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

inputs

arr (array) 2-dimensional numpy array

functions.**age_correction**(*month_nums_array*, *ages_array*, *retage*)

Long_Form

Returns a long_form (all months) array of employee ages by incrementing starting ages according to month number.

Note: Retirement age increases are handled by the build_program_files script by incrementing retirement dates and by the clip_ret_ages function within the make_skeleton script.

inputs

month_nums_array (array) gen_month_skeleton function output (ndarray)

ages_array (array) starting_age function output aligned with long_form (ndarray) i.e. s_age is starting age (aligned to empkeys) repeated each month.

retage (integer or float) output clip upper limit for retirement age

Output is s_age incremented by a decimal month value according to month_num (this is candidate for np.put refactored function)

functions.**align_fill_down**(*l*, *u*, *long_indexed_df*, *long_array*)

Data align current values to all future months (short array segment aligned to long array) This function is used to set the values from the last standalone month as the initial data for integrated dataset computation when a delayed implementation exists.

uses pandas df auto align - relatively slow TODO (for developer) - consider an all numpy solution

inputs

l, u (integers) current month slice indexes (from long df)

long_indexed_df (dataframe) empty long dataframe with empkey indexes

long_array (array) long array of multiple month data (orig_job, fur_codes, etc)

declare long indexed df outside of function (input). grab current month slice for array insertion (copy). chop long df to begin with current month (copy). assign array to short df. data align short df to long df (chopped to current month and future). copy chopped df column as array to long_array return long_array

functions.**align_next**

"Carry forward" data from one month to the next. Compare indexes (empkeys) from one month to the next month. When matching index is found, assign corresponding index value to new result array. Effectively finds the remaining employees (not retired) in the next month and copies the target column data values for them from current month data into the next months data.

inputs

this_index_arr (array) current month index of unique employee keys

next_index_arr (array) next month index of unique employee keys (a subset of this_index_arr)

these_vals_arr (array) the data column segment (attribute) to carry forward

functions.**anon_dates**(*df*, *date_col_list*, *max_adj=5*, *positive_only=True*, *inplace=False*)

Add (or optionally, add or subtract) a random number of days to each element of a date attribute column.

inputs

df (dataframe) short-form (master list) pandas dataframe containing a date attribute column

date_col_list (list) name(s) of date attribute column(s) to be adjusted (as a list of strings)

Example:

```
['ldate', 'doh', 'dob']
```

max_adj (integer) the maximum number of days to add (or optionally subtract) from each element within the date column

positive_only (boolean) if True limit the range of adjustment days from zero to the max_adj value. If False, limit the range of adjustment from negative max_adj value to positive max_adj value.

inplace (boolean) if True, insert the results directly into the date column(s) of the input dataframe. Caution: make a copy first!

functions.**anon_empkeys**(*df*, *seq_start=10001*, *frame_num=10000000*, *inplace=False*)

Produce a list of unique, randomized employee numbers, catogorized by employee group number code. Output may be used to anonymize a dataset empkey column.

Dataframe input (df) must contain an employee group (eg) column.

inputs

df (dataframe) short-form (master list) pandas dataframe containing an employee group code column

seq_start (integer) this number will be added to each employee group cumulative count to "seed" the random employee numbers. These numbers will be shuffled within employee groups by the function for the output

frame_num (integer) This number will be multiplied by each employee code and added to the employee group cumulative counts (added to the seq_start number), and should be much larger than the data model population to provide a constant length employee number (empkey) for all employees.

inplace (boolean) if True, insert the results directly into the "empkey" column of the input dataframe. Caution: make a copy first!

functions.**anon_master**(*case, empkey=True, name=True, date=False, sample=False, seq_start=10001, frame_num=10000000, min_seg=3, max_seg=3, add_rev=False, date_col_list=['ldate', 'doh'], max_adj=5, positive_only=True, date_col_list_sec=['dob'], max_adj_sec=5, positive_only_sec=True, n=None, frac=None, reset_index=False*)

Specialized function to anonymize selected columns from a master.xlsx file and/or select a subset. All operations are inplace. The original master file is copied and saved as master_orig.xlsx.

The default parameters will replace last names and employee keys with substitute values. Date columns, (doh, ldate, dob) will also be adjusted if the date input is set True and the proper column names are set as column list inputs.

The function reads the original excel file, copies and saves it, modifies the original file as directed, and writes the results back to the original file. Subsequent dataset creation runs will use the modified data. The output master list will be sorted according to the original master list order.

inputs

case (string) the case study name

empkey (boolean) if True, anonymize the empkey column

name (boolean) if True, anonymize the lname column

date (boolean) if True, anonymize date columns as disignated with the date_col_list and the date_col_list_sec inputs

sample (boolean) if True, sample the dataframe if the n or frac inputs is/are not None

seq_start (integer) beginning anonymous employee number portion of empkey

frame_num (integer) large frame number which will contain all generated employee numbers. This number will be adjusted to begin with the appropriate employee group code

min_seg (integer) minimum number of 2-character segments to include in the generated substitute last names.

max_seg (integer) maximum number of 2-character segments to include in the generated substitute last names.

add_rev (boolean) if True, add reversed, non-duplicated 2-character segments to the pool of strings for name construction. This is normally not necessary and will construct output strings with multiple consecutive consonants/vowels.

date_col_list (list) list of date value columns to adjust. All columns in this list will be adjusted in a syncronized fashion, meaning a random day adjustment for each row will be applied to each row member of all columns.

max_adj (integer) maximum random adjustment deviation, in days, from the original date(s)

positive_only (boolean) if True, only adjust dates forward in time

date_col_list_sec (list) a secondary list of date column(s) which will be adjusted independently from the date columns in the date_col_list

max_adj_sec (integer) maximum random adjustment deviation, in days, from the original date(s) in the date_col_list_sec columns

positive_only_sec (boolean) if True, only adjust dates forward in time (for secondary cols)

n (integer or None) number of rows to sample if the sample input is True. This input will override the frac input

frac (float (0.0 - 1.0) or None) decimal fraction (0.0 to 1.0) of the master list to sample, if the sample input is True and the n input is None

reset_index (boolean) if True, reset the index of the output master list (zero-based integer index). Do not use this option normally because it will wipe out the empkey index of the master list.

`functions.`**`anon_names`**(*length=10*, *min_seg=3*, *max_seg=3*, *add_rev=False*, *df=None*, *inplace=False*)

Generate a list of random strings

Output may be used to anonymize a dataset name column

The length of the output strings will be determined by the min_seg and max_seg inputs. The segments (seg) are random 2-letter combinations of a consonant and a vowel. An additional random consonant or vowel will be added to the segment combinations, so the length of the output strings will always be an odd number. The min and max may be the same value to produce a list of strings of uniform length.

Example:

If the min_seg input is 1 and the max_seg input is 3, the output list will contain strings from 3 (2-letter seg + 1 random letter) to 7 characters.

inputs

length (integer) the length of the output list

min_seg (integer) the minimum number of 2 letter segments to include in the output list

max_seg (integer) the maximum number of 2 letter segments to include in the output list (must be => "min_seg" input)

add_rev (boolean) add vowel-consonant combinations to the consonant-vowel segments. (this is not normally needed to produce random and readable strings)

df (dataframe) optional short-form pandas dataframe input. If not None, use the length of the dataframe as the "length" input value

inplace (boolean) if the "df" input is not None, insert the results directly into the input "lname" column. Caution: make a copy first!

functions.**anon_pay**(*df*, *proportional=True*, *mult=1.0*, *inplace=False*)

Substitute pay table baseline rate information a proportional method or with a non-linear, non-proportional method.

inputs

df (dataframe) pandas dataframe containing pay rate date (dataframe representation of the "rates" worksheet from the pay_tables.xlsx workbook)

proportional (boolean) if True, use the mult input to increase or decrease all of the "rates" worksheet pay data proportionally. If False, use a fixed algorithm to disproportionally adjust the pay rates.

mult (integer or float) if the proportional input is True, multiply all pay rate values by this input value

inplace (boolean) if True, replace the values within the original dataframe with the "anonomized" values.

functions.**anon_pay_table**(*case*, *proportional=True*, *mult=1.0*)

Anonymize the "rates" worksheet of the "pay_tables.xlsx" input file. The rates may be proportionally adjusted (larger or smaller) or disproportionally adjusted with a fixed algorithm.

A copy of the original excel file is copied and saved as "pay_tables_orig.xlsx".

All modifications are inplace.

inputs

case (string) the case name

proportional (boolean) if True, use the mult input to increase or decrease all of the "rates" worksheet pay data proportionally. If False, use a fixed algorithm to disproportionally adjust the pay rates.

mult (integer or float) if the proportional input is True, multiply all pay rate values by this input value

functions.**assign_cond_ratio**(*job*, *this_job_count*, *ratio_dict*, *orig_range*, *assign_range*, *eg_range*, *fur_range*, *cap=None*)

Apply a job ratio assignment condition

The main job assignment function calls this function at the appropriate months.

This function applies a ratio for job assignment between ratio groups. Each ratio group may contain one or more employee groups. The number of jobs affected may be limited with the "cap" input.

The ratio for job assignment is set with the inputs on the "ratio_cond" worksheet of the *settings.xlsx* input spreadsheet, using the "group" columns and the corresponding "weights" columns.

Optionally, the ratio of jobs which exists during the "month_start" spreadsheet input may be captured and used for job assignment during the data model months when the ratio job assignment condition is applicable ("month_start" through "month_end"). The existing ratios are captured and used by setting the "snapshot" input cell to True for the appropriate basic job row. When using the snapshot option, any weightings designated within the "weights" columns will be ignored.

There may be a mix of snapshot ratios and ratios set by the "weight" columns for use within the program. There may also be count-capped ratio assignments and straight ratio assignments within the same data model as long as the effective months and jobs do not overlap, but there may only be one row of ratio data for a job level within the same input worksheet.

No bump, no flush rules apply when assigning jobs by ratio, meaning only job openings due to retirements, increases in job counts, or other openings will be assigned according to the ratio schedule. Employees previously holding a job affected by the ratio condition will not be displaced to allow an employee from a different ratio group to have that job when the ratio assignment period begins. Therefore, it may take some time for the desired ratio of job assignments to be achieved if it differs significantly from the actual ratio(s) when the time period of conditional job assignment begins.

inputs

job (integer or float) job level number

this_job_count (integer or float) number of jobs available

ratio_dict (dictionary) ratio condition dictionary, constructed with the build_program_files script and possibly modified by the set_snapshot_weights function if the "snapshot" option is set to True on the "ratio_cond" worksheet of the "settings.xlsx" input spreadsheet.

orig_range (1d array) original job range Month slice of the orig_job column array (normally pertaining a specific month).

assign_range (1d array) job assignment range Month slice of the assign_range column array

eg_range (1d array) employee group range Month slice of the eg_range column array

fur_range (1d array) furlough range Month slice of the fur_range column array

cap (integer (or whole float)) if a count ratio job assignment is being used, this number represents the number of jobs affected by the conditional assignment. Available jobs above this amount are not affected.

functions.**assign_job_counts**

assign job counts to job count array month slice

inputs

job_count_range (array) month slice of long job count array

assign_range (array) month slice of long job assignment array

job (integer) job level number

this_job_count (integer) job count alloted for job level

functions.**assign_jobs_full_flush_job_changes**(*nonret_counts*, *job_counts*, *num_job_levels*)

(Long_Form)

Using the nonret counts for each month:

a. determine the long form slice for assignment, and

b. slice the jobs list from the top to create job assignment column

c. create a corresponding furlough column

d. create a job count column

Uses the job_counts (job_gain_loss_table function)[0] to build stovepiped job lists allowing for job count changes each month and a furlough status column.

Unassigned employees (not enough jobs), are left at job number zero This is the full bump and full flush version

inputs

nonret_counts (numpy array) array containing the number of non-retired employees for each month

job_counts (numpy array) array containing the monthly counts of jobs for each job level

num_job_levels (integer) the number of job levels in the model (excluding furlough level)

functions.**assign_jobs_nbnf_job_changes**(*df*, *lower*, *upper*, *total_months*, *job_reduction_months*, *start_month*, *condition_list*, *sdict*, *tdict*, *fur_return=False*)

(Long_Form)

Uses the job_gain_or_loss_table job count array for job assignments. Jobs counts may change up or down in any category for any time period. Handles furlough and return of employees, prior rights/conditions and restrictions and recall of initially furloughed employees.

Inputs are precalculated outside of function to the extent possible. Returns tuple (long_assign_column, long_count_column, orig jobs, fur_data)

inputs

df (dataframe) long-form dataframe with ['eg', 'sg', 'fur', 'orig_job'] columns.

lower (array) ndarry from make_lower_slice_limits function (calculation derived from cumsum of count_per_month function)

upper (array) cumsum of count_per_month function

total_months (integer or float) sum of count_per_month function output

job_reduction_months (list) months in which the number of jobs is decreased from the get_job_reduction_months function

start_month (integer) integer representing the month number to begin calculations, likely month of integration when there exists a delayed integration (from settings dictionary)

condition_list (list) list of special job assignment conditions to apply, example: ['prex', 'count', 'ratio']

sdict (dictionary) the program settings dictionary (produced by the build_program_files script)

tdict (dictionary) job tables dictionary (produced by the build_program_files script)

fur_return (boolean) model employee recall from furlough if True using recall schedule from settings dictionary (allows call to mark_for_recall function)

Assigns jobs so that original standalone jobs are assigned each month (if available) unless a better job is available through attrition of employees.

Each month loop starts with the lowest job number.

For each month and for each job level:

1. assigns nbnf (orig) job if job array (long_assign_column) element is zero (unassigned) and orig job number is less than or equal to the job level in current loop, then
2. assigns job level in current loop to unassigned slots from top to bottom in the job array (up to the count of that job level remaining after step one above)

Each month range is determined by slicing using the lower and upper inputs. A comparison is made each month between the original job numbers and the current job loop number.

Job assignments are placed into the monthly segment (assign_range) of the long_assign_column.

The long_assign_column eventually becomes the job number (jnum) column in the dataset.

Original job numbers of 0 indicate no original job and are treated as furloughed employees. No jobs are assigned to furloughees unless furlough_return option is selected.

functions.**assign_standalone_job_changes**(*eg*, *df_align*, *lower*, *upper*, *total_months*, *job_counts_each_month*, *total_monthly_job_count*, *nonret_each_month*, *job_change_months*, *job_reduction_months*, *start_month*, *sdict*, *tdict*, *apply_sg_cond=True*)

(Long_Form)

Uses the job_gain_or_loss_table job count array for job assignments. Jobs counts may change up or down in any category for any time period. Handles furlough and return of employees, prior rights/conditions and restrictions and recall of initially furloughed employees.

Inputs are precalculated outside of function to the extent possible. Returns tuple (long_assign_column, long_count_column, held_jobs, fur_data, orig_jobs)

inputs

eg (integer) input from an incremental loop which is used to select the proper employee group recall scedule

df_align (dataframe) dataframe with ['sg', 'fur'] columns

num_of_job_levels (integer) number of job levels in the data model (excluding a furlough level)

lower (1d array) ndarry from make_lower_slice_limits function (calculation derived from cumsum of count_per_month function)

upper (1d array) cumsum of count_per_month function

total_months (integer or float) sum of count_per_month function output

job_counts_each_month (array) output of job_gain_loss_table function[0] (precalculated monthly count of jobs in each job category, size (months,jobs))

total_monthly_job_count (array) output of job_gain_loss_table function[1] (precalculated monthly total count of all job categories, size (months))

nonret_each_month (1d array) output of count_per_month function

job_change_months (list) the min start month and max ending month found within the array of job_counts_each_month inputs (find the range of months to apply consideration for any job changes - prevents unnecessary looping)

job_reduction_months (list) months in which the number of jobs is decreased (list). from the get_job_reduction_months function

start_month (integer) starting month for calculations, likely implementation month from settings dictionary

sdict (dictionary) the program settings dictionary (produced by the build_program_files script)

tdict (dictionary) job tables dictionary (produced by the build_program_files script)

apply_sg_cond (boolean) compute with pre-existing special job quotas for certain employees marked with a one in the sg column (special group) according to a schedule defined in the settings dictionary

Assigns jobs so that original standalone jobs are assigned each month (if available) unless a better job is available through attrition of employees.

Each month loop starts with the lowest job number.

For each month and for each job level: 1. assigns nbnf (orig) job if job array (long_assign_column) element is zero (unassigned) and orig job number is less than or equal to the job level in current loop, then 2. assigns job level in current loop to unassigned slots from top to bottom in the job array (up to the count of that job level remaining after step one above)

Each month range is determined by slicing using the lower and upper inputs.

A comparison is made each month between the original job numbers and the current job loop number.

Job assignments are placed into the monthly segment (assign_range) of the long_assign_column.

The long_assign_column eventually becomes the job number (jnum) column in the dataset.

Original job numbers of 0 indicate no original job and are treated as furloughed employees - no jobs are assigned to furloughees unless furlough_return option is selected.

functions.**career_months**(*ret_input*, *start_date*)

(Short_Form)

Determine how many months each employee will work including retirement partial month

"ret_input" (retirement dates) may be in the form of a pandas dataframe, pandas series, array, list, or string

Output is a numpy array of integers containing the number of months between the start_date and each date in the ret_input (months from start date to retirement for each employee)

inputs

ret_input (dataframe, series, array, list, or string) retirement dates input

start_date (string date) comparative date for the retirement dates input, normally the data model starting date

functions.**clear_dill_files**()
remove all files from 'dill' folder. used when changing case study, avoids possibility of file from previos calculations being used by new study

functions.**clip_ret_ages**(*ret_age_dict*, *init_ret_age*, *dates_long_arr*, *ages_long_arr*)
Clip employee ages in employee final month to proper retirement age if the model includes an increasing retirement age over time

inputs

ret_age_dict (dictionary) dictionary of retirement increase date to new retirement age as defined in settings dictionary

init_ret_age initial retirement age prior to any increase

dates_long_arr (numpy array) array of month dates (long form, same value during each month)

ages_long_arr (numpy array) array of employee ages (long form)

functions.**contract_year_and_raise**(*df*, *settings_dict*)
(Month_Form)

Generate the contract pay year for indexing into the pay table. Pay year is clipped to last year of contract.

Also create an annual assumed raise column applicable to the time period beyond the contract duration. This is a multiplier column with a compounded value each subsequent year. If no raise is elected (via the settings.xlsx input file, "scalars" worksheet), then this column will be all ones. The annual raise percentage is designated on the same worksheet. The input df must be a single column dataframe containing end-of-month dates, one for each month of the data model.

NOTE: (this function can accept any number of pay exception periods through the pay_exceptions dictionary, populated by the "pay_exceptions" worksheet values within the *settings.xlsx* input file, see the program documentation for more information)

inputs

df (dataframe) a single column dataframe containing end-of-month dates, one for each month of the data model

settings_dict (dictionary) dictionary of program settings generated by the *build_program_files* script

functions.**convert_to_datetime**(*date_data*, *attribute*)
Convert a dataframe column, series, list, or string input into an array of datetimes

inputs

data_data (dataframe, series, array, list, or string) pandas dataframe with a date column containing string dates or datetime objects, pandas series of dates (strings or datetime objects), a list/array of date strings or datetime objects, or a single comma-separated string containing date information.

attribute (string) if the date_data type is a dataframe, the name of the column containing the date information. Otherwise, this input is ignored.

functions.**convert_to_enhanced**(*eg_job_counts*, *j_changes*, *job_dict*)

Convert employee basic job counts to enhanced job counts (includes full-time and part-time job level counts) and convert basic job change schedules to enhanced job change schedules. Returns tuple (enhanced_job_counts, enhanced_job_changes)

inputs

eg_job_counts A list of lists of the basic level job counts for each employee group. Each nested list has a length equal to the number of basic job levels.

example:

```
[[197, 470, 1056, 412, 628, 1121, 0, 0],
[80, 85, 443, 163, 96, 464, 54, 66],
[0, 26, 319, 0, 37, 304, 0, 0]]
```

j_changes input from the settings dictionary describing change of job quantity over months of time (list)

example:

```
[1, [35, 64], 87, [80, 7, 0]]
```

[[job level, [start and end month], total job count change, [eg allotment of change for standalone calculations]]

job_dict conversion dictionary for an enhanced model. This is the "jd" key value from the settings dictionary. It uses the basic job levels as the keys, and lists as values which containin the new full- and part-time job level numbers and the percentage of basic job counts to be converted to full-time jobs.

example:

```
{1: [1, 2, 0.6],
2: [3, 5, 0.625],
3: [4, 6, 0.65],
4: [7, 8, 0.6],
5: [9, 12, 0.625],
6: [10, 13, 0.65],
7: [11, 14, 0.65],
8: [15, 16, 0.65]}
```

functions.**convert_to_hex**(*rgba_input*)

convert float rgba color values to string hex color values rgba = color values expressed as:

red, green, blue, and (optionally) alpha float values rgba_input may be:

1. a single rgba list or tuple
2. a list or tuple containing rgba lists or rgba tuples
3. a dictionary of key: rgba values

output is string hex color values in place of rgba values

Examples:

input single rgba value:

```
sample_value = (.5, .3, .2)
convert_to_hex(sample_value)
'#7f4c33'
```

input list:

```
sample_list = [[0.65, 0.81, 0.89, 1.0],
               [0.31, 0.59, 0.77, 1.0],
               [0.19, 0.39, 0.70, 1.0],
               [0.66, 0.85, 0.55, 1.0]]

convert_to_hex(sample_list)
['#a5cee2', '#4f96c4', '#3063b2', '#a8d88c']
```

input dict:

```
sample_dict = {1: (.65, .45, .45, 1.),
               2: [.60, .45, .45, 1.],
               3: (.55, .45, .45, 1.)}

convert_to_hex(sample_dict)
{1: '#a57272', 2: '#99593a', 3: '#8c7249'}
```

inputs

rgba_input (tuple, list, or dictionary) input may be a single list or tuple OR a list of float rgba values as lists or tuples OR a dictionary with values as lists or tuples. Valid string hex values may also be passed as inputs and will be returned unchanged.

functions.**copy_excel_file**(*case*, *file*, *return_path_and_df=False*, *revert=False*, *verbose=True*)

Copy an excel file and add '_orig' to the file name, or restore an excel file from the '_orig'

copy.

inputs

case (string) the data model case name

file (string) the excel file name without the .xlsx extension

return_path_and_df (boolean) if True, return a tuple containing the file path as a string and the worksheet designated by the "file" input as a dataframe

revert (boolean) if False, copy the excel file and add '_orig' to the file name. if True, restore the copied file and drop the '_orig' suffix

verbose (boolean) if True, print a brief summary of the operation result

functions.**count_avail_jobs**

use numba to loop through the job assignment range and count the number of jobs in a specified job level previously assigned from the previous month, then subtract result from the total job level positions count. This result identifies the number of openings available for the current month.

inputs

assign_range (array) monthly slice of job assignment array

job (integer) job level being tested

this_job_count (integer) total job positions count for the job being tested

functions.**count_per_month**(*career_months_array*)

Month_Form

Returns number of employees remaining for each month (not retired). Cumulative sum of career_months_array input (np array) that are greater or equal to each incremental loop month number.

Note: alternate method to this function is value count of mnums: df_actives_each_month = pd.DataFrame(df_idx.mnum.value_counts()) df_actives_each_month.columns = ['count']

input

career_months_array output of career_months function. This input is an array containing the number of months each employee will work until retirement.

functions.**create_snum_and_spcnt_arrays**(*jnums*, *job_level_count*, *monthly_population_counts*, *monthly_job_counts*, *lspcnt_remaining_only*)

Calculates: long_form seniority number ('snum', only active employees), seniority percentage ('spcnt', only active employees), list number ('lnum', includes furlougees), list percentage ('lspcnt', includes furloughees).

Iterate through monthly jobs count data, capturing monthly_job_counts to be used as the denominator for percentage calculations.

This function produces four ndarrays which will make up four columns in the long_form pandas dataset.

Returns tuple (long_snum_array, long_spcnt_array, long_list_array, long_lspcnt_array)

inputs

jnums the long_form jnums result

job_level_count number of job levels in model

monthly_population_counts count_per_month function output

monthly_job_counts total of all jobs each month derived from job_gain_loss_table function (table) >>> np.add.reduce(table, axis=1)

lspcnt_remaining_only calculate list percentage based on employees remaining in each month including furloughees, otherwise percentage calculation denominator is the greater of employees remaining (incl fur) or jobs available

functions.**cross_val**

functions.**distribute**(*available*, *weights*, *cap=None*)

proportionally distribute 'available' according to 'weights'

usage example:

```
distribute(334, [2.48, 1])
```

returns distribution as a list, rounded as integers:

```
[238, 96]
```

inputs

available (integer) the count (number) to divide

weights (list) relative weighting to be applied to available count for each section. numbers may be of any size, integers or floats. the number of resultant sections is the same as the number of weights in the list.

cap (integer) limit distribution total to this amount, if less than the "available" input.

functions.**eg_quotas**(*quota*, *actual*, *cap=None*, *this_job_count=None*)

determine the job counts to be assigned to each ratio group during a ratio condition job assignment routine

inputs

quota (list or list-like) the desired job counts for each employee group

actual (list or list-like) the actual job counts for each employee group

cap (integer (or whole float)) if a count ratio routine is being used, this is the total count of jobs to be affected by the ratio

this_job_count (integer (or whole float)) the monthly count of the applicable job

functions.**find_index_val**(*df1*, *df2*, *df2_vals*, *col1=None*, *col2=None*)

find a value in another dataframe with the same index of another given value in a dataframe.

df1 index, df2 index, and the value columns must contain unique values.

inputs

df1 (dataframe) the first dataframe containing values to index match in another dataframe

df2 (dataframe) the second dataframe with corresponding index values

df2_vals (list) values to match

functions.**find_nearest**

functions.**gen_month_skeleton**

Long_Form

Create an array of month numbers with each month number repeating n times for n non-retired employees in each month. i.e. the first month section of the array will be all zeros (month: 0) repeating for the number of non-retired employees. The next section of the array will be all ones (month: 1) repeating for the number of employees remaining in month 1. Output is a 1d ndarray.

This funtion creates the first column and the basic form of the skeleton dataframe which is the basis for the dataset dataframes.

inputs

month_count_array a numpy array containing the number of employees remaining or not retired for each month. This input is the result of the count_per_month function.

functions.**gen_skel_emp_idx**

Long_Form

For each employee who remains for each month, grab that employee index number.

This index will be the key to merging in other data using data alignment. Input is the result of the count_per_month function (np.array) and the result of the career_months function

inputs

monthly_count_array (numpy array) count of non-retired active employees for each month in the model, the ouput from the count_per_month function.

career_mths_array (numpy array) career length in months for each employee, output of career_months functions.

empkey_source_array (numpy array) empkey column data as array

Returns tuple (skel_idx_array, skel_empkey_array)

functions.**get_indexes**

functions.**get_job_change_months**(*job_changes*)

extract a sorted list of only the unique months containing a change in any job count as defined within the settings dictionary job change schedules

input

job_changes list of job change schedule lists, normally equal to the j_changes variable from the settings dictionary

functions.**get_job_reduction_months**(*job_changes*)

extract a sorted list of only the unique months containing a reduction in any job count as defined within the settings dictionary job change schedules

input

job_changes list of job change schedule lists, normally equal to the j_changes variable from the settings dictionary

functions.**get_month_slice**(*df*, *l*, *h*)

Convenience function to extract data for a particular month. Input is low and high indexes of target month data (within dataset containing many months)

The input may also be an array (not limited to a dataframe).

inputs

df dataframe (or array) to be sliced

l lower index of slice

h upper index of slice

functions.**get_recall_months**(*list_of_recall_schedules*)

extract a sorted list of only the unique months containing a recall as defined within the settings dictionary recall schedules.

input

list_of_recall_schedules list of recall schedule lists, normally equal to the recalls variable from the settings dictionary

functions.**hex_dict**()

returns a color name to hex code dictionary (no inputs)

functions.**job_gain_loss_table**(*months*, *job_levels*, *init_job_counts*, *job_changes*, *standalone=False*)

Make two arrays of job tally information.

The first array has a row for each month in the model, and a column for each job level (excluding furlough). This array provides a count for each job for each month of the model accounting for changes provided by the job change schedules defined by the settings dictionary. The second array is a one-dimensional array containing the sum for all jobs for each month of the model.

inputs

months (integer) number of months in model

job_levels (integer) number of job levels in model (excluding furlough level)

init_job_counts (tuple of two numpy arrays) initial job counts. Output from the make_jcnts function, essentially an array of the job count lists for each employee group and an array of the combined counts.

job_changes (list) The list of job changes from the settings dictionary.

standalone (boolean) if True, use the job count lists for the separate employee groups, otherwise use the combined job count

Returns tuple (job_table, monthly_job_totals)

functions.**load_datasets**(*other_datasets=['standalone', 'skeleton', 'edit', 'hybrid']*)

Create a dictionary of proposal names to corresponding datasets. The datasets are generated with the RUN_SCRIPTS notebook. This routine reads the names of the case study proposals from a pickled dataframe ('dill/proposal_names.pkl'), created by the build_program_files.py script. It then looks for the matching stored datasets within the dill folder. The datasets are loaded into a dictionary, using the proposal names as keys.

The dictionary allows easy reference to datasets from the Jupyter notebook and from within functions.

input

other_datasets (list) list of datasets to load in addition to those computed from the proposals (from the case-specific proposals.xlsx Excel file)

functions.**longevity_at_startdate**(*ldate_input*, *start_date*, *return_as_months=False*)

(Short_Form)

determine how much longevity (years) each employee has accrued as of the start date

float output is longevity in years (+1 added to reflect current 1-based pay year)

inputs

ldate_input (dataframe, series, list, or string) list of longevity dates in datetime format

start_date (string date) comparative date for longevity dates, normally the data model starting date

return_as_months (boolean) option to return result as month value instead of year value

functions.**make_cat_order**(*ds*, *table*)

make a long-form "cat_order" (global job ranking) column This function assigns a global job position value to each employee, considering the modeled job level hierarchy and the job count within each level. For example, if a case study contains 3 job levels with 100 jobs in each level, an employee holding a job in the middle of job level 2 would be assigned a cat_order value of 150.

Category order for standalone employee groups is "normalized" to an integrated scale by applying *standalone* job level percentage (relative position within a job level) to the *integrated* job level counts. This process allows "apples to apples" comparison between standalone and integrated job progression.

Standalone cat_order will only reflect job levels available within the standalone scenario. If the integrated model contains job levels which do not exist within a standalone employee group model, standalone cat_order results will exclude the respective job level rank segments and will rank the existing standalone data according to the integrated ranking scale.

The routine creates numpy array lookup tables from integrated job level count data for each month of the model. The tables are the source for count and additive information which is used to calculate a rank number within job level and cumulative job count additives.

Month number and job number arrays (from the input ds (dataset)) are used to index into the numpy lookup arrays, producing the count and additive arrays.

A simple formula is then applied to the percentage, count, and additive arrays to produce the cat_order array.

inputs

ds (dataframe) a dataset containing ['jobp', 'mnum', 'jnum'] columns

table (numpy array) the first output from the job_gain_loss_table function which is a numpy array with total job counts for each job level for each month of the data model

functions.**make_decile_bands**(*num_bands=40*, *num_returned_bands=10*)

creates an array of lower and upper percentile values surrounding a consistent schedule of percentile markers. If the user desires to sample data at every 10th percentile, this function provides selectiable bottom and top percentile limits surrounding each 10th percentile, or variable width sample ranges.

num_bands input must be multiple of 5 greater than or equal to 10 and less than 10000.

num_returned_bands input must be multiple of 5, equal to or less than the num_bands input, and num_bands/num_returned_bands must have no remainder.

Used for selecting sample employees surrounding deciles (0, 10, 20 etc. percent levels).

Top and bottom bands will be half of normal size.

inputs

num_bands Width of bands in percentage is determined by num_bands input. Input of 40 would mean bands 2.5% wide. (100/40) Top and bottom bands would be 1.25% wide. Ex. 0-1.25%, 8.75-11.25%, ... 98.75-100%

num_returned_bands number of returned delineated sections. Must be a multiple of 5 less than or equal to the num_bands value with no remainder when divided into the num_bands value. (note: an input of 10 would result in 11 actual segments, one-half band at the top and bottom of list (0% and 100%), and 9 full bands surrounding each decile, 10% to 90%)

functions.**make_delayed_job_counts**(*imp_month*, *delayed_jnums*, *lower*, *upper*)

Make an array of job counts to be inserted into the long_form job counts array of the job assignment function. The main assignment function calls this function prior to the implementation month. The array output of this function is inserted into what will become the job count column. These jobs are from the standalone job results. The job count column displays a total monthly count of the job in the corresponding jnum (job number) column.

inputs

imp_month (integer) implementation month, defined by settings dictionary

delayed_jnums (numpy array) array of job numbers, normally data from the start of the model through the implementation month

lower (numpy array) array of indexes marking the beginning of data for each month within a larger array of stacked, multi-month data

upper (numpy array) array of indexes marking the end of data for each month

functions.**make_dict_from_columns**(*df*, *key_col*, *value_col*)

Make a dictionary from two dataframe columns. One column will be the keys and the other the values.

Unique key column values will be assigned dictionary values. If the key_col input contains duplicates, only the last duplicate key-value pair will exist within the returned dictionary.

inputs

df (dataframe) pandas dataframe containing the columns

key_col (string (or possibly integer)) dataframe column which will become dictionary keys

value_col (string (or possibly integer)) dataframe column which will become dictionary values

functions.**make_eg_pcnt_column**(*df*, *recalc_each_month=False*, *mnum=0*, *inplace=True*, *trim_ones=True*, *fixed_col_name='eg_start_pcnt'*, *running_col_name='eg_pcnt'*)

make an array derived from the input df reflecting one of the following options:

Option A: The percentage of each employee within his/her original employee group for **a selected month**. The array values will be data-aligned with the df input index. This option is useful for tracking percentile cohorts throughout the model.

Option B: The percentage of each employee within his/her original employee group **recalculated each month**. This has the effect of adjusting each group relative percentage for population changes due to retirements, furlough, etc. This option is useful for monthly percentile cohort comparisons.

This function either adds a column to the input dataframe or returns an array of values, the same length as the input dataframe.

Note: This function calculations include any furloughed employees assign to long-form dataframe (with default month 0 values aligned):

```
make_eg_pcnt_column(df)
```

inputs

df (dataframe) pandas dataframe containing an employee group code column ('eg') and a month number column ('mnum'). The dataframe must be indexed with employee number code integers ('empkey')

recalc_each_month (boolean)

if True: recalculate separate employee group percentage each month of data model

if False: calculate values for one month only - align those values by employee number (empkey) to the entire data model

mnum (integer) if recalc_each_month is True, calculate values for this selected month number

inplace (boolean) if True, add a column to the input dataframe with the calculated values. If False, return a numpy array of the calculated values.

trim_ones (boolean) if True, replace 100% values (1.0) with a value slightly under 1.0 (.9999). This action assists construction of percentile quantiles for membership grouping purposes.

exclude_fur (boolean) if True, remove furloughed employees from percentage calculations

fixed_col_name (string) manually designated name for dataframe column when recalc_each_month input is False and inplace input is True.

running_col_name (string) manually designated name for dataframe column when recalc_each_month input is True and inplace input is True.

functions.**make_group_lists**(*df*, *column_name*)

this function is used with Excel input to convert string objects and integers into Python lists containing integers. This function is used with the count_ratio_dict dictionary construction. The function works with one column at a time.

Output is a list of lists which may be reinserted into a column of the dataframe.

example:

A	B	C	D
1	6	0	"2,3"
8	4	5	"5"

```
make_group_lists(df, ["D"])

[[2, 3], [5]]
```

This function allows the user to type the string 2,3 into an Excel worksheet cell and have it interpreted by seniority_list as [2, 3]

inputs

df (dataframe) dataframe containing Excel employee group codes

column_name dataframe column name to convert

functions.**make_intgrtd_from_sep_stove_lists**(*job_lists_arr*, *eg_arr*, *fur_arr*, *eg_total_jobs*, *num_levels*, *skip_fur=True*)

Month_Form

Compute an integrated job list built from multiple independent eg stovepiped job lists.

This function is for multiple egs (employee groups) - multiple lists in one job_lists_arr.

Creates an ndarray of job numbers.

Function takes independent job number lists and an array of eg codes which represent the eg ordering in the proposed list.

Job numbers from the separate lists are added to the result array according to the eg_arr order. Jobs on each list do not have to be in any sort of order. The routine simply adds items from the list(s) to the result array slots in list order.

inputs

job_lists_arr array of the input job number arrays. represents the jobs that would be assigned to each employee in a list form. each list within the array will be the length of the respective eg.

eg_arr short_form array of eg codes (proposal eg ordering)

fur_arr short_form array of fur codes from proposal

eg_total_jobs list length n egs sums of total jobs available for each eg, form: [n,n,n]

num_levels number of job levels in model (excluding furlough level)

skip_fur (boolean) ignore or skip furloughs when assigning stovepipe jobs. If True, employees who are originally marked as furloughed are assigned the furlough level number which is 1 greater than the number of job levels. If False, jobs are assigned within each employee group in a stovepipe fashion, including those employees who are marked as furloughed

functions.**make_jcnts**(*job_count_lists*)

Make two arrays: 1. array of n lists of job counts for n number of eg job count input lists 2. array of one summation list of first array (total count of all eg jobs) The arrays above will not contain a furlough count. Returns tuple (eg_job_counts, combined_job_count)

inputs

job_count_lists list of the employee job count list(s). If the program is using the enhanced jobs option, this input will be the output of the convert_jcnts_to_enhanced function. Otherwise, it will be the eg_counts variable from the settings dictionary.

Example return:

```
(array([
[ 237,  158,  587, 1373,  352,  739,  495,  330,  784,
 1457,    0,  471,  785,    0,    0,    0],
[  97,   64,  106,  575,   64,  310,  196,  130,  120,
 603,   71,   72,  325,   38,   86,   46],
[  0,    0,   33,  414,   20,  223,    0,    0,   46,
 395,    0,   28,  213,    0,    0,    0]]),
array(
[ 334,  222,  726, 2362,  436, 1272,  691,  460,  950,
 2455,   71,  571, 1323,   38,   86,   46]))
```

functions.**make_lists_from_columns**(*df*, *columns*, *remove_zero_values=False*, *try_integers=False*, *as_tuples=False*)

combine columns row-wise into separate lists, return a list of lists

example:

A	B	C	D
1	6	0	2
8	4	5	3

```
make_lists_from_columns(df, ["A", "B", "C"])

[[1, 6, 0], [8, 4, 5]]

make_lists_from_columns(df, ["A", "B", "C"],
                        remove_zero_values=True,
                        as_tuples=True)

[(1, 6), (8, 4, 5)]
```

inputs

df (dataframe) pandas dataframe containing columns to combine

columns (list) list of column names

try_integers (boolean) if True, if all column values are numerical, the output will be converted to integers

remove_zero_values (boolean) if True, remove zero values from list or tuple outputs. The routine checks for zeros as a zero value or a list with a single zero value

as_tuples (boolean) if True, output will be a list of tuples instead of a list of lists

functions.**make_lower_slice_limits**(*month_counts_cumsum*)

for use when working with unique month data within larger array (slice).

The top of slice is cumulative sum, bottom of each slice will be each value of this function output array. Output is used as input for nbnf functions.

inputs

month_counts_cumsum (numpy array) cumsum of count_per_month function output (employee count each month)

functions.**make_original_jobs_from_counts**(*jobs_arr_arr*, *eg_array*, *fur_array*, *num_levels*)

Short_Form

This function grabs jobs from standalone job count arrays (normally stovepiped) for each employee group and inserts those jobs into a proposed integrated list, or a standalone list. Each eg (employee group) is assigned jobs from their standalone list in order top to bottom.

Result is a combined list of jobs with each eg maintaining ordered independent stovepipe jobs within the combined list of jobs jobs_arr_arr is an array of arrays, likely output[0] from make_array_of_job_lists function.

Order of job count arrays within jobs_arr_arr input must match emp group codes order (1, 2, 3, etc.).

If total group counts of job(s) is less than slots available to that group, remaining slots will be assigned (remain) a zero job number (0).

eg_array is list (order sequence) of employee group codes from proposed list with length equal to length of proposed list.

Result of this function is ultimately merged into long form for no bump no flush routine.

employees who are originally marked as furloughed are assigned the furlough level number which is 1 greater than the number of job levels.

inputs

jobs_arr_arr (numpy array of arrays) lists of job counts for each job level within each employee group, each list in order starting with job level one.

eg_array (numpy array) employee group (eg) column data from master list source

fur_array furlough (fur) column data from master list source

num_levels number of job levels (without furlough level) in the model

functions.**make_preimp_array**(*ds_stand*, *ds_integrated*, *imp_high*, *compute_cat*, *compute_pay*)

Create an ordered numpy array of pre-implementation data gathered from the pre-calculated standalone dataset and a dictionary to keep track of the information. This data will be joined with post_implementation integrated data and then copied into the appropriate columns of the final integrated dataset.

inputs

ds_stand (dataframe) standalone dataset

ds_integrated (dataframe) dataset ordered for proposal

imp_high highest index (row number) from implementation month data (from long-form dataset)

compute_cat (boolean) if True, compute and append a job category order column

compute_pay (boolean) if True, compute and append a monthly pay column and a career pay column

functions.**make_starting_val_column**(*df*, *attr*, *inplace=True*)

make an array of values derived from the input dataframe which will reflect the starting value (month zero) of a selected attribute. Each employee will be assigned the zero-month attribute value specific to that employee, duplicated in each month of the data model.

This column allows future attribute analysis with a constant starting point for all employees. For example, retirement job position may be compared to initial list percentage.

assign to long-form dataframe:

```
df['start_attr'] = make_starting_val_column(df, attr)
```

input

df (dataframe) pandas dataframe containing the attr input column and a month number coulumn. The dataframe must be indexed with employee number code integers ('empkey')

attr (column name in df) selected zero-month attribute (column) from which to assign values to the remaining data model months

functions.**make_stovepipe_jobs_from_jobs_arr**(*jobs_arr*, *total_emp_count=0*)

Month_Form

Compute a stovepipe job list derived from the total count of jobs in each job level.

This function is for one eg (employee group) and one jobs_arr (list).

Creates an array of job numbers from a job count list (converted to np.array).

Result is an array with each job number repeated n times for n job count. - job count list like : job_counts = [334, 222, 701, 2364] - jobs_array = np.array(job_counts)

inputs

jobs_arr (numpy array) job counts starting with job level 1

total_emp_count if zero (normal input), sum of jobs_arr elements, otherwise user-defined size of result_jobs_arr

functions.**make_stovepipe_prex_shortform**(*job_list*, *sg_codes*, *sg_rights*, *fur_codes*)

Short_Form

Creates a 'stovepipe' job assignment within a single eg including a special job assignment condition for a subgroup. The subgroup is identified with a 1 in the sg_codes array input, originating with the sg column in the master list.

This function applies a pre-existing (prior to the merger) contractual job condition, which is likely the result of a previous seniority integration.

The subset group will have proirity assignment for the first n jobs in the affected job category, the remainding jobs are assigned in seniority order.

The subgroup jobs are assigned in subgroup stovepipe order. This function is applicable to a condition with known job counts. The result of this function is used with standalone

calculations or combined with other eg lists to form an integrated original job assignment list.

inputs

job_list list of job counts for eg, like [23,34,0,54,...]

sg_codes ndarray eg group members entitled to job condition (marked with 1, others marked 0) length of this eg population

sg_rights list of lists (from settings dictionary) including job numbers and job counts for condition. Columns 2 and 3 are extracted for use.

fur_codes array of ones and zeros, one indicates furlough status

functions.**make_tuples_from_columns**(*df*, *col_list*, *return_as_list=True*, *date_cols=[]*, *return_dates_as_strings=False*, *date_format='%Y-%m-%d'*)

Combine row values from selected columns to form tuples. Returns a list of tuples which may be assigned to a new column. The length of the list is equal to the length of the input dataframe. Date columns may be first converted to strings before adding to output tuples if desired.

inputs

df (dataframe) input dataframe

col_list (list) columns from which to create tuples

return_as_list (boolean) if True, return a list of tuples

date_cols (list) list of columns to treat as dates

return_dates_as_strings (boolean) if True, for columns within the data_cols input, convert date values to string format

date_format (string) string format of converted date columns

functions.**mark_for_furlough**(*orig_range*, *fur_range*, *month*, *jobs_avail*, *num_of_job_levels*)

Assign fur code to employees when count of jobs is less than count of active employees in inverse seniority order and assign furloughed job level number. note: normally only called during a job change month though it will do no harm if called in other months

inputs

orig_range current month slice of jobs held

fur_range current month slice of fur data

month current month (loop) number

jobs_avail total number of jobs for each month array, job_gain_loss_table function output [1]

num_of_job_levels from settings dictionary, used to mark fur job level as num_of_job_levels + 1

functions.**mark_for_recall**(*orig_range*, *num_of_job_levels*, *fur_range*, *month*, *recall_sched*, *jobs_avail*, *standalone=True*, *eg_index=0*, *method='sen_order'*, *stride=2*)

change fur code to non-fur code for returning employees according to selected method (seniority order, every nth furloughee, or random) note: function assumes it is only being called during a recall month

inputs

orig_range original job range

num_of_job_levels number of job levels in model, normally from settings dictionary

fur_range current month slice of fur data

month current month (loop) number

recall sched list(s) of recall schedule (recall amount/month, recall start month, recall end month)

jobs_avail total number of jobs for each month array, job_gain_loss_table function output [1]

standalone (boolean) This function may be used with both standalone and integrated dataset generation. Set this variable to True for use within standalone dataset calculation, False for integrated dataset calculation routine.

eg_index (integer) selects the proper recall schedule for standalone dataset generation, normally from a loop increment. The recall schedule is defined in the settings dictionary. set to zero for an integrated routine (integrated routine uses a global recall schedule)

method means of selecting employees to be recalled default is by seniority order, most senior recalled first other options are:

stride: i.e. every other nth employee. (note: could be multiple strides per month if multiple recall lists are designated).

random: use shuffled list of furloughees

stride set stride if stride option for recall selected. default is 2.

functions.**mark_fur_range**

apply fur code to current month fur_range based on job assignment status

inputs

assign_range current month assignment range (array of job numbers, 0 indicates no job)

fur_range current month fur status (1 means furloughed, 0 means not furloughed)

job_levels number of job levels in model (from settings dictionary)

functions.**max_of_nested_lists**(*nested_list*, *return_min=False*)

Find the maximum value within a list of lists (or tuples or arrays). The function may optionally return the minimum value within nested containers.

inputs

nested_list (list, tuple, or array) nested container input

return_min (boolean) if True, return minimum of nested_list input (vs. max)

functions.**monotonic**(*sequence*)

test for stricly increasing array-like input May be used to determine when need for no bump, no flush routine is no longer required. If test is true, and there are no job changes, special rights, or furlough recalls, then a straight stovepipe job assignment routine may be implemented (fast).

inputs

sequence array-like input (list or numpy array ok)

functions.**print_settings**()

grab settings dictionary data settings and put it in a dataframe and then print it for a quick summary of scalar settings dictionary inputs

functions.**remove_zero_groups**(*ratio_dict*)

remove data related to a "dummy" group represented by a zero

example:

```
{2: [([2], [0], [1]), [0, 2, 6], 34, 120]}
```

becomes:

```
{2: [([2], [1]), [0, 6], 34, 120]}
```

inputs

ratio_dict (dictionary) the ratio dictionary produced by the build_program_files script originating from the "ratio_cond" worksheet of the *settings.xlsx* input file

functions.**sample_dataframe**(*df*, *n=None*, *frac=None*, *reset_index=False*)

Get a random sample of a dataframe by rows, with the number of rows in the returned sample defined by a count or fraction input.

inputs

df (dataframe) pandas dataframe for sampling

n (integer) If not None, the count of the rows in the returned sample dataframe. The "n" input will override the "frac" input if both are not None. Will be clipped between zero and len(df) if input exceeds these boundries.

frac (float) If not None, the size of the returned sample dataframe relative to the input dataframe. Will be ignored if "n" input is not None. Will be clipped between 0.0 and 1.0 if input exceeds these boundries. An input of .3 would randomly select 30% of the rows from the input dataframe.

reset_index (boolean) If True, reset the output dataframe index

If both the "n" and "frac" inputs are None, a random single row will be returned.

The rows in the output dataframe will be sorted according to original order.

functions.**save_and_load_dill_folder**(*save_as=None*, *load_case=None*, *print_saved=False*)

Save the current "dill" folder to the "saved_dill_folders" folder, or load a saved dill folder as the "dill" folder if it exists.

This function allows calculated case study pickle files (including the calculated datasets) to be saved to or loaded loaded from a "saved_dill_folders" folder.

The "saved_dill_folders" folder is created if it does not already exist. The load_case input is a case study name. If the load_case input is set to None, the function will only save the current dill folder and do nothing else. If a load_case input is given, but is incorrect or no matching folder exists, the function will only save the current dill folder and do nothing else.

The user may print a list of available saved dill folders (for loading) by setting the print_saved input to True. No other action will take place when this option is set to True.

If an award has conditions which differ from proposed conditions, the settings dictionary must be modified and the dataset rebuilt.

This function allows previously calculated datasets to be quickly retrieved and eliminates continual adjustment of the settings spreadsheet if the user switches between case studies (assuming the award has been determined and no more input adjustment will be made).

inputs

save_as (string) A user-specified folder prefix. If None, the current "dill" folder will be saved using the current case study name as a prefix. If set to a string value, the current dill folder will be saved with the "save_as" string value prefix.

Example with the save_as variable set to "test1". The existing dill folder would be saved as:

```
saved_dill_folders/test1_dill_folder
```

load_case (string) The name of a case study. If None, the only action performed will be to save the current "dill" folder to the "saved_dill_folders" folder. If the load_case variable is a valid case study name and a saved dill folder for that case study exists, the saved dill folder will become the current dill folder (contents of the saved dill folder will be copied into the current dill folder). This action will occur after the contents of the current dill folder are copied into the "saved_dill_folders" folder.

print_saved (boolean) option to print the saved folder prefixes only. This provides a quick check of the folders available to be loaded. No other action will take place with this option set to True.

functions.**set_snapshot_weights**(*job*, *ratio_dict*, *orig_rng*, *eg_range*)

Determine the job distribution ratios to carry forward during the ratio condition application period using actual jobs held ratios. likely called at implementation month by main job assignment function Count the number of jobs held by each of the ratio groups for each of the affected job level numbers. Set the weightings in the distribute function accordingly.

inputs

ratio_dict (dictionary) dictionary containing job levels as keys and ratio groups, weightings, month_start and month end as values.

orig_rng (numpy array) month slice of original job array

eg_range (numpy array) month slice of employee group code array

functions.**squeeze_increment**(*data*, *eg*, *low_num*, *high_num*, *increment*)

Move members of a selected eg (employee group) within a list according to an increment input (positive or negative) while retaining relative ordering within all eg groups.

inputs

data (dataframe) dataframe with empkey as index which at minimum includes an order column and an eg column

eg (integer) employee group number code

low_num and high_num indexes for the beginning and end of the list zone to be re-ordered

increment (integer) the amount to add or subrtract from the appropriate eg order number increment can be positive (move down list) or negative (move up list - toward zero)

Selected eg order numbers within the selected zone (as a numpy array) are incremented - then the entire group order numbers are reset within the zone using scipy.stats.rankdata.

The array is then assigned to a dataframe with empkeys as index.

functions.**squeeze_logrithmic**(*data*, *eg*, *low_value*, *high_value*, *log_factor=1.5*, *put_segment=1*, *direction='d'*)

perform a log squeeze (logrithmic-based movement of one eg (employee group), determine the closest matching indexes within the rng to fit the squeeze, put the affected group in those indexes, then fill in the remaining slots with the other group(s), maintaining orig ordering within each group at all times

inputs

data (dataframe) a dataframe indexed by empkey with at least 2 columns: employee group (eg) and order (order)

eg (employee code integer) the employee group to move

low_val and high_val (integers) integers marking the boundries (rng) for the operation (H must be greater than L)

log_factor (float) determines the degree of 'logrithmic packing'

put_segment (float) allows compression of the squeeze result (values under 1)

direction (string) squeeze direction: "u" - move up the list (more senior) "d" - move down the list (more junior)

functions.**starting_age**(*dob_input*, *start_date*)

Short_Form

Returns decimal age at given date.

"dob_input" (birth dates) may be in the form of a pandas dataframe, pandas series, list, or string

inputs

dob_list (dataframe, series, list, or string) birth dates input

start_date comparative date for the birth dates, normally the data model starting date

functions.**update_excel**(*case*, *file*, *ws_dict={}*, *sheets_to_remove=None*)

Read an excel file, optionally remove worksheet(s), add worksheets or overwrite worksheets with a dictionary of ws_name, dataframe key, value pairs, and write the excel file back to disk

inputs

case (string) the data model case name

file (string) the excel file name without the .xlsx extension

ws_dict (dictionary) dictionary of worksheet names as keys and pandas dataframes as values. The items in this dictionary will be passed into the excel file as worksheets. The worksheet name keys may be the same as some or all of the worksheet names in the excel file. In the case of matching names, the data from the input dict will overwrite the existing data (worksheet) in the excel file. Non-overlapping worksheet names/dataframe values will be added as new worksheets.

sheets_to_remove (list) a list of worksheet names (strings) representing worksheets to remove from the excel workbook. It is not necessary to remove sheets which are being replaced by worksheet with the same name.

CHAPTER

TWELVE

INTERACTIVE_PLOTTING MODULE

interactive_plotting.**bk_basic_interactive**(*doc*, *df=None*, *plot_height=700*, *plot_width=900*, *dot_size=5*)

run a basic interactive chart as a server app - powered by the bokeh plotting library. Run the app in the jupyter notebook as follows:

```
from functools import partial
import pandas as pd

import interactive_plotting as ip

from bokeh.io import show, output_notebook

from bokeh.application.handlers import FunctionHandler
from bokeh.application import Application

output_notebook()

proposal = 'p1'
df = pd.read_pickle('dill/ds_' + proposal + '.pkl')

handler = FunctionHandler(partial(ip.bk_basic_interactive, df=df))

app = Application(handler)
show(app)
```

inputs

doc (required input) do not change this input

df (dataframe) calculated dataset input, this is a required input

plot_height (integer) height of plot in pixels

plot_width (integer) width of plot in pixels

Add plot_height and/or plot_width parameters as kwargs within the partial method:

```
handler = FunctionHandler(partial(ip.bk_basic_interactive,
                                  df=df,
                                  plot_height=450,
                                  plot_width=625))
```

Note: the "df" argument is not optional, a valid dataset variable must be assigned.

CHAPTER

THIRTEEN

LIST_BUILDER MODULE

list_builder.**build_list**(*df*, *measure_list*, *weight_list*, *show_weightings=False*, *hide_rank_cols=True*, *return_df=False*)

Construct a "hybrid" list ordering.

Note: first run the "prepare_master_list" function and use the output for the "df" input here.

Combine and sort various attributes according to variable multipliers to produce a list order. The list order output is based on a sliding scale of the priority assigned amoung the attributes.

The default output is a dataframe containing the new hybrid list order and employee numbers (empkey) only, and is written to disk as 'dill/p_hybrid.pkl'.

The entire hybrid-sorted dataframe may be returned by setting the "return_df" input to True. This does not affect the hybrid list order dataframe - it is produced and stored regardless of the "return_df" option.

inputs

df the prepared dataframe output of the prepare_master_list function

measure_list a list of attributes that form the basis of the final sorted list. The employee groups will be combined, sorted, and numbered according to these attributes one by one. Each time the current attribute numbered list is formed, a weighting is applied to that order column. The final result number will be the rank of the cummulative total of the weighted attribute columns.

weight_list a list of decimal weightings to apply to each corresponding measure within the measure_list. Normally the total of the weight_list should be 1, but any numbers may be used as weightings since the final result is a ranking of a cumulative total.

show_weightings add columns to display the product of the weight/column mutiplcation

return_df option to return the new sorted hybrid dataframe as output. Normally, the function produces a list ordering file which is written to disk and used as an input by the compute measures script.

hide_rank_cols remove the attrubute rank columns from the dataframe unless visual review is desired

list_builder.**compare_dataframes**(*base*, *compare*, *return_orphans=True*, *ignore_case=True*, *print_info=False*, *convert_np_timestamps=True*)

Compare all common index and common column DataFrame values and report if any value is not equal in a returned dataframe.

Values are compared only by index and column label, not order. Therefore, the only values compared are within common index rows and common columns. The routine will report the common columns and any unique index rows when the print_info option is selected (True).

Inputs are pandas dataframes and/or pandas series.

This function works well when comparing initial data lists, such as those which may be received from opposing parties.

If return_orphans, returns tuple (diffs, base_loners, compare_loners), else returns diffs. diffs is a differential dataframe.

inputs

base baseline dataframe or series

compare dataframe or series to compare against the baseline (base)

return_orphans separately calculate and return the rows which are unique to base and compare

ignore_case convert the column labels and column data to be compared to lowercase - this will avoid differences detected based on string case

print_info option to print out to console verbose statistical information and the dataframe(s) instead of returning dataframe(s)

convert_np_timestamps numpy returns datetime64 objects when the source is a datetime date-only object. this option will convert back to a date-only object for comparison.

list_builder.**find_index_locs**(*df*, *index_values*)

Find the pandas dataframe index location of an array-like input of index labels.

Returns a list containing the index location(s).

inputs

df dataframe - the index_values input is a subset of the dataframe index.

index_values array-like collection of values which are a subset of the dataframe index

list_builder.**find_row_orphans**(*base_df*, *compare_df*, *col*, *ignore_case=True*, *print_output=False*)

Given two columns (series) with the same column label in separate pandas dataframes, return

values which are unique to one or the other column, not common to both series. Will also work with dataframe indexes.

Returns tuple (base_loners, compare_loners) if not print_output. These are dataframes with the series orphans.

Note: If there are orphans found that have identical values, they will both be reported. However, currently the routine will only find the first corresponding index location found and report that location for both orphans.

inputs

base_df first dataframe to compare

compare_df second dataframe to compare

col column label of the series to compare. routine will compare the dataframe indexes with the input of 'index'.

ignore_case convert col to lowercase prior to comparison

print_output print results instead of returning results

`list_builder.`**`find_series_locs`**(*df*, *series_values*, *column_label*)

Find the pandas dataframe index location of an array-like input of series values.

Returns a list containing the index location(s).

inputs

df dataframe - the series_values input is a subset of one of the dataframe columns.

series_values array-like collection of values which are a subset of one of the dataframe columns (the column_lable input)

column_label the series within the pandas dataframe containing the series_values

`list_builder.`**`names_to_integers`**(*names*, *leading_precision=5*, *normalize_alpha=True*)

convert a list or series of string names (i.e. last names) into integers for numerical sorting

Returns tuple (int_names, int_range, name_percentages)

inputs

names List or pandas series containing strings for conversion to integers

leading_precision Number of characters to use with full numeric precision, remainder of characters will be assigned a rounded single digit between 0 and 9

normalize_alpha If True, insert 'aaaaaaaaaa' and 'zzzzzzzzzz' as bottom and top names. Otherwise, bottom and top names will be calculated from within the names input

output

1. an array of the name integers
2. the range of the name integers,
3. an array of corresponding percentages for each name integer relative to the range of name integers array

Note: This function demonstrates the possibility of constructing a list using any type or combination of attributes.

list_builder.**prepare_master_list**(*name_int_demo=False*, *pre_sort=True*)

Add attribute columns to a master list. One or more of these columns will be used by the build_list function to construct a "hybrid" list ordering.

Employee groups must be listed in seniority order in relation to employees from the same group. Order between groups is uninmportant at this step.

New columns added: ['age', 's_lmonths', 'jnum', 'job_count', 'rank_in_job', 'jobp', 'eg_number', 'eg_spcnt']

inputs

name_int_demo if True, lname strings are converted to an integer then a corresponding alpha-numeric percentage for constructing lists by last name. This is a demo only to show that any attribute may be used as a list weighting factor.

pre_sort sort the master data dataframe doh and ldate columns prior to beginning any calculations. This sort has no effect on the other columns. The s_lmonths coulumn will be calculated on the sorted ldate data.

Job-related attributes are referenced to job counts from the settings dictionary.

list_builder.**sort_and_rank**(*df*, *col*, *tiebreaker1=None*, *tiebreaker2=None*, *reverse=False*)

Sort a datframe by a specified attribute and insert a column indicating the resultant ranking. Tiebreaker inputs select columns to be used for secondary ordering in the event of value ties. Reverse ordering may be selected as an option.

inputs

df input dataframe

col (string) dataframe column to sort

tiebreaker1, tiebreaker2 (string(s)) second and third sort columns to break ties with primary col sort

reverse (boolean) If True, reverses sort (descending values)

list_builder.**sort_eg_attributes**(*df, attributes=['doh', 'ldate'], reverse_list=[0, 0], add_columns=False*)

Sort master list attribute columns by employee group in preparation for list construction. The overall master list structure and order is unaffected, only the selected attribute columns are sorted (normally date-related columns such as doh or ldate)

inputs

df The master data dataframe (does not need to be sorted)

attributes columns to sort by eg (inplace)

reverse_list If an attribute is to be sorted in reverse order (descending), use a '1' in the list position corresponding to the position of the attribute within the attributes input

add_columns If True, an additional column for each sorted attribute will be added to the resultant dataframe, with the suffix '_sort' added to it.

list_builder.**test_df_col_or_idx_equivalence**(*df1*, *df2*, *col=None*)

check whether two dataframes contain the same elements (but not necessarily in the same order) in either the indexes or a selected column

inputs

df1, df2 the dataframes to check

col if not None, test this dataframe column for equivalency, otherwise test the dataframe indexes

Returns True or False

CHAPTER

FOURTEEN

MATPLOTLIB_CHARTING MODULE

matplotlib_charting.**add_pad**(*list_in*, *pad=100*)

Separate all elements in a monotonic list by a minimum pad value.

Used by plotting functions to prevent overlapping tick labels.

inputs

list_in (list) a monotonic list of numbers

pad (integer) the minimum separation required between list elements

If the function is unable to produce a list with the pad between all elements (excluding the last list spacing), the original list is returned. The function will permit the final list padding (between the last two elements) to be less than the pad value.

matplotlib_charting.**age_kde_dist**(*df*, *color_list*, *p_dict*, *max_age*, *ds_dict=None*, *mnum=0*, *title_size=14*, *min_age=25*, *chart_style='darkgrid'*, *xsize=12*, *ysize=10*, *image_dir=None*, *image_format='png'*)

From the seaborn website: Fit and plot a univariate or bivariate kernel density estimate.

inputs

df (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

color_list (list) list of colors for employee group plots

p_dict (dictionary) eg to string dict for plot labels

max_age (float) maximum age to plot (x axis limit)

ds_dict (dictionary) output from load_datasets function

mnum (integer) month number to analyze

title_size (integer or float) text size of chart title

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**age_vs_spcnt**(*df*, *eg_list*, *mnum*, *color_list*, *p_dict*, *ret_age*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *chart_style='darkgrid'*, *size=20*, *alpha=0.8*, *suptitle_size=14*, *title_size=12*, *legend_size=12*, *xsize=10*, *ysize=8*, *image_dir=None*, *image_format='png'*)

scatter plot with age on x axis and list percentage on y axis. note: input df may be prefiltered to plot focus attributes, i.e. filter to include only employees at a certain job level, hired between certain dates, with a particular age range, etc.

inputs

df (string or dataframe) text name of input proposal dataset, also will accept any dataframe variable (if a sliced dataframe subset is desired, for example) Example: input can be 'proposal1' (if that proposal exists, of course, or could be df[df.age > 50])

eg_list (list) list of employee groups to include example: [1, 2]

mnum (int) month number to study from dataset

color_list (list) color codes for plotting each employee group

p_dict (dict) dictionary, numerical eg code to string description

ret_age (integer or float) chart xaxis limit for plotting

ds_dict (dict) variable assigned to the output of the load_datasets function, reqired when string dictionary key is used as df input

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

chart_style (string) any valid seaborn plotting style

size (integer) size of scatter points

alpha (float) scatter point alpha (0.0 to 1.0)

suptitle_size (integer or font) text size of chart super title

title_size (integer or float) text size of chart title

legend_size (integer or float) text size of chart legend

xsize, ysize (integer or float) plot size in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**build_subplotting_order**(*rows*, *cols*)

build a list of integers to permit passing through subplots by columns note: only used when looping completes one vertical column before continuing to next column

inputs

rows, cols (integer) number of rows and columns in multiple chart output

matplotlib_charting.**cohort_differential**(*ds*, *base*, *sdict*, *cdict*, *adict*, *measure='ldate'*, *compare_value='2010-12-31'*, *mnum=None*, *ds_dict=None*, *single_eg_compare=None*, *sort_xax_by_measure=False*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *pos_color='g'*, *neg_color='r'*, *pos_alpha=0.25*, *neg_alpha=0.25*, *bg_color=None*, *zero_line_color='m'*, *title_size=16*, *label_size=14*, *tick_size=12.5*, *legend_size=12.5*, *xsize=14*, *ysize=10*, *image_dir=None*, *image_format='png'*)

Compare proposed integrated list locations of employees from different groups who share a similar attribute value.

This function is best used with date-type attributes, such as longevity date or date of hire.

The comparative list locations are a continuous list of index locations determined by finding the last list position within an attribute column from another employee group which is less than or equal to a corresponding column from the base employee group. A variance or differential is calculated by comparing the base and comparative locations.

Attributes (measures) are sorted within each employee group prior to comparison. The x axis may be arranged to display proposed list ordering or the attribute value range (typically a date range).

Differences in list position are shown with a line above or below zero. One employee group (base) is compared to other group(s) in the proposed list within a selected month. When the line is above zero, it means that the base group cohort at a particular x axis position is on the list ahead of another group cohort by an amount equal to the y displacement of the line. The line colors correspond to the employee group color codes.

The default behavior is to compare the base group with all other groups at once, but single group comparison may be accomplished as well.

When the x axis is set to display list location (not attribute values), the user may designate a compare value. The list location of employees from each group who share the comparison attribute value will be marked on the chart with a color-coded vertical line.

inputs

ds (dataframe) dataset for analysis

base (integer) employee group number code

sdict (dictionary) program settings dictionary

cdict (dictionary) program color dictionary

adict (dictionary) program attribute dictionary

measure (string) attribute column for list location comparison, likely 'ldate' or 'doh'

compare_value (type to match measure input dtype) value to mark on chart if "sort_xax_by_measure" input is False. Likely a date string, such as "2001-01-31"

mnum (integer) data model month number to study

ds_dict (dictionary) dictionary of datasets, likely generated by the "load_datasets" function

single_eg_compare (integer) if not None, compare base employee group to this group only

sort_xax_by_measure (boolean) if True, use an x axis for the chart based on the selected measure. if False, use list location for the x axis

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

pos_color, neg_color (color value string) color used for the positive and negative area shading

pos_alpha, neg_alpha (integer or float) transparency value assigned to the positive and negative color shading areas (0.0 to 1.0)

bg_color (color value string) if not None, the color for the chart background

zero_line_color (color value string) color for the zero line

title_size (integer or float) text size for the chart title

label_size (integer or float) text size for the chart axis labels

tick_size (integer or float) text size for the chart tick labels

legend_size (integer or float) text size for the chart legend

xsize, ysize (integer or float) size of the chart in inches (width, height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**cond_test**(*df*, *grp_sel*, *enhanced_jobs*, *job_colors*, *job_dict*, *basic_jobs=None*, *ds_dict=None*, *plot_all_jobs=False*, *min_mnum=None*, *max_mnum=None*, *limit_to_jobs=None*, *use_and=False*, *print_count_months=None*, *print_all_counts=False*, *plot_job_bands_chart=True*, *only_target_bands=False*, *legend_size=14*, *title_size=16*, *xsize=8*, *ysize=8*, *image_dir=None*, *image_format='png'*)

visualize selected job counts over time applicable to computed condition with optional printing of certain data.

Primary usage is validation of job assignment conditions by charting the count(s) of job(s) assigned by the program to particular employee groups over time.

The function may also be used to evaluate distribution of jobs with various proposals. Career progression of employees who enjoy special job rights may be understood particularily well

by utilizing the print_all_counts option.

The output is 2 charts. The first chart is a line chart displaying selected job count information over time. The second is a stacked area chart displaying all job counts for the selected group(s) over time.

There are additional optional print outputs. The print_all_counts option will print a dataframe containing job count totals for each month. The print_count_months input is a list of months to print the only the plotted job counts, primarily for testing purposes.

inputs

df (dataframe) dataset(dataframe) to examine

grp_sel (list) integer input(s) representing the employee group code(s) to select for analysis. This argument also will accept the string 'sg' to select a special job rights group(s). Multiple inputs are normally handled as 'or' filters, meaning an input of [1, 'sg'] would mean employee group 1 **or** any special job rights group, but can be modified to mean only group 1 **and** special job rights employees with the 'use_and' input.

enhanced_jobs (boolean) if True, basic_jobs input job levels will be converted to enhanced job levels with reference to the job_dictionary input, otherwise basic_jobs input job levels will be used

job_colors (list) list of color values to use for job plots

job_dict (dictionary) dictionary containing basic to enhanced job level conversion data. This is likely the settings dictionary "jd" value.

basic_jobs (list) basic job levels to plot. This list will be converted to the corresponding enhanced job list if the enhanced_jobs input is set to True. Defaults to [1] if not assigned.

ds_dict (dictionary) dataset dictionary which allows df input to be a string description (proposal name)

plot_all_jobs (boolean) option to plot all of the job counts within the input dataset vs only those selected with the basic_jobs input (or as converted to enhanced jobs if enhanced_jobs input is True). The jobs plotted may be filtered by the limit_to_jobs input.

min_mnum (integer) integer input, only plot data including this month forward(mnum). Defaults to zero.

max_mnum (integer) integer input, only plot data through selected month (mnum). Defaults to maximum mnum for input data

limit_to_jobs (list) a list of jobs to plot, allowing focus on target jobs. Should be a subset of normal output, otherwise no filtering of normal output occurs

use_and (boolean) when the grp_sel input has more than one element, require filtered dataframe for analysis to be part of all grp_sel input sets.

print_count_months (list) list of month(s) for printing job counts

print_all_counts (boolean) if True, print the entire job count dataframe.

plot_job_bands_chart (boolean) if True, plot an area chart beneath the job count chart. The area chart will display all of the jobs available to the selected employee group(s) over time with job band areas

only_target_bands (boolean) if True, plot area chart of jobs from job count chart only, vs the default of all job levels

legend_size (integer or float) text size of legend labels

title_size (integer or float) text size of chart title

xsize, ysize (integer or float) size of chart display in inches (width and height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**determine_dataset**(*ds_def*, *ds_dict=None*, *return_label=False*)

this function permits either a dictionary key (string) or a dataframe variable to be used in functions as a dataframe object.

inputs

ds_def (dataframe or string) A pandas dataframe or a string representing a key for a dictionary which contains dataframe(s) as values

ds_dict (dictionary) A dictionary containing string to dataframes, used if ds_def input is not a dataframe

return_label (boolean) If True, return a descriptive dataframe label if the ds_dict was referenced, otherwise return a generic "Proposal" string

matplotlib_charting.**diff_range**(*df_list*, *dfb*, *measure*, *eg_list*, *attr_dict*, *ds_dict=None*, *cm_name='Set1'*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *year_clip=2042*, *show_range=False*, *range_alpha=0.25*, *show_mean=True*, *normalize_y=False*, *suptitle_size=16*, *title_size=16*, *tick_size=13*, *label_size=16*, *legend_size=14*, *chart_style='whitegrid'*, *ysize=6*, *xsize=11*, *image_dir=None*, *image_format='png'*)

Plot a range of differential attributes or a differential average over time. Individual employee groups and proposals may be selected. Each chart indicates the results for one group with color bands or average lines indicating the results for that group under different proposals. This is different than the usual method of different groups being plotted on the same chart.

inputs

df_list (list) list of datasets to compare, may be ds_dict (output of load_datasets function) string keys or dataframe variable(s) or mixture of each

dfb (dataframe, can be proposal string name) baseline dataset, accepts same input types as df_list above

measure (string) differential data to compare

eg_list (list) list of integers for employee groups to be included in analysis. example: [1, 2, 3] A chart will be produced for each employee group number.

eg_colors (list) list of colors to represent different proposal results

attr_dict (dictionary) dataset column name description dictionary

ds_dict (dictionary) output from load_datasets function

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

year_clip (integer) only plot data up to and including this year

show_range (boolean) show a transparent background on the chart representing the range of values for each measure for each proposal

range_alpha (float) transparancy level for range plotting (0.0 to 1.0)

show_mean (boolean) plot a line representing the average of the measure values for the group under each proposal

normalize_y (boolean) if measure is 'spcnt' or 'lspcnt', equalize the range of the y scale on all charts (-.5 to .5)

suptitle_size (integer or font) text size of chart super title

title_size (integer or font) text size of chart title

tick_size (integer or font) text size of chart tick labels

label_size (integer or font) text size of chart x and y axis labels

legend_size (integer or font) text size of the legend labels

chart_style (string) any valid seaborn plotting style (string)

xsize, ysize (integer or font) size of chart in inches (width and height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**differential_scatter**(*df_list*, *dfb*, *measure*, *eg_list*, *attr_dict*, *color_dict*, *p_dict*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *prop_order=True*, *show_scatter=True*, *show_lin_reg=True*, *show_mean=True*, *mean_len=50*, *dot_size=15*, *lin_reg_order=15*, *ylimit=False*, *ylim=5*, *suptitle_size=14*, *title_size=12*, *legend_size=14*, *tick_size=11*, *label_size=12*, *bright_bg=False*, *bright_bg_color='#faf6eb'*, *chart_style='whitegrid'*, *xsize=12*, *ysize=8*, *image_dir=None*, *image_format='png'*)

plot an attribute differential between datasets.

datasets may be filtered by other attributes if desired.

Example: plot the difference in cat_order (job rank number) between all integrated datasets vs. standalone for all employee groups, applicable to month 57. (optionally add a pre-filter(s), such as all employees hired prior to a certain date)

The chart may be set to use proposal order or native list percentage for the x axis.

The scatter markers are selectable on/off, as well as an average line and a linear regression line.

inputs

df_list (list) list of datasets to compare, may be ds_dict (output of load_datasets function) string keys or dataframe variable(s) or mixture of each

dfb (string or variable) baseline dataset, accepts same input types as df_list above

measure (string) attribute to analyze

eg_list (list) list of employee group codes

attr_dict (dictionary) dataset column name description dictionary

color_dict (dictionary) dictionary containing color list string titles to lists of color values generated by the build_program_files script

p_dict (dictionary) employee group code number to description dictionary

ds_dict (dictionary) output from load_datasets function

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

eg_list (list) a list of employee groups to analyze

prop_order (boolean) if True, organize x axis by proposal list order, otherwise use native list percent

show_scatter (boolean) if True, draw the scatter chart markers

show_lin_reg (boolean) if True, draw linear regression lines

show_mean (boolean) if True, draw average lines

mean_len (integer) moving average length for average lines

dot_size (integer or float) scatter marker size

lin_reg_order (integer) regression line is actually a polynomial regression lin_reg_order is the degree of the fitting polynomial

ylimit (boolean) if True, set chart y axis limit to ylim (below)

ylim (integer or float) y axis limit positive and negative if ylimit is True

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) text size of chart title

legend_size (integer or float) text size of chart legend labels

tick_size (integer or float) text size of x and y tick labels

label_size (integer or float) text size of x and y descriptive labels

bright_bg (boolean) use a custom color chart background

bright_bg_color (color value) chart background color if bright_bg input is set to True

chart_style (string) style for chart, valid inputs are any seaborn chart style

xsize, ysize (integer or float) size of chart (width, height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**display_proposals**()

print out a list of the proposal names which were generated and stored in the dill folder by the build_program_files script

no inputs

matplotlib_charting.**eg_attributes**(*ds*, *xmeasure*, *ymeasure*, *sdict*, *adict*, *cdict*, *eg_list=None*, *mnum=None*, *ret_only=False*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *q_eglist_only=True*, *xquant_lines=True*, *x_quantiles=10*, *xl_alpha=1*, *xl_ls='dashed'*, *xl_lw=1*, *xl_color='.7'*, *x_bands=True*, *xb_fc='.3'*, *xb_alpha=0.09*, *yquant_lines=True*, *y_quantiles=10*, *yl_alpha=1*, *yl_ls='dashed'*, *yl_lw=1*, *yl_color='.7'*, *y_bands=True*, *yb_fc='#66ffb3'*, *yb_alpha=0.09*, *linestyle=''*, *linewidth=0*, *markersize=5*, *marker_alpha=0.7*, *grid_alpha=0.25*, *chart_style='ticks'*, *full_xpcnt=True*, *full_ypcnt=True*, *xax_rotate=70*, *label_size=13*, *qtick_size=12*, *tick_size=12*, *border_size=0.5*, *legend_size=14*, *title_size=18*, *y_title_pos=1.12*, *box_height=0.95*, *xsize=15*, *ysize=11*, *image_dir=None*, *image_format='png'*)

Plot selected employee group(s) attribute data.

Chart x and y axes may be any dataset attributes, including date attributes.

Quantile membership for the x and/or y attribute may also be displayed. Membership may be relative to the entire integrated population or only to the employee group(s) selected for display (q_eglist_only input).

inputs

ds (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

xmeasure (string) attribute to plot on x axis

ymeasure (string) attribute to plot on y axis

sdict (dictionary) program settings dictionary

adict (dictionary) dataset column name description dictionary

cdict (dictionary) program colors dictionary

eg_list (list) list of employee groups to plot (integer codes)

mnum (integer) month number for analysis

ret_only (boolean) if True, mnum input is ignored and results are displayed for all employees at retirement

ds_dict (dictionary) output of the load_datasets function, dictionary. This keyword argument must be set if a string key is used as the df input.

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

q_eglist_only (boolean) if set to True:

if quantile bands are displayed, show membership based on selected employee groups (eg_list input).

if set to False:

if quantile bands are displayed, show membership based on the integrated group population (all groups).

xquant_lines (boolean) if True, show quantile membership for x axis attribute

x_quantiles (integer) number of quantiles to display if xquant_lines input is True

xl_alpha (float) transparency value of x axis quantile lines (0.0 to 1.0)

xl_ls (string) x axis quantile lines linestyle ('dashed', 'dotted', etc.)

xl_lw (integer or float) x axis quantile lines line width

xl_color (string color value) x axis quantile lines color

x_bands (boolean) if True, show a background color within every other x axis quantile membership area

xb_fc (string color value) x axis quantile bands background color

xb_alpha (float) x axis quantile bands color transparency value (0.0 to 1.0)

yquant_lines (boolean) if True, show quantile membership for y axis attribute

y_quantiles (integer) number of quantiles to display if yquant_lines input is True

yl_alpha (float) transparency value of y axis quantile lines (0.0 to 1.0)

yl_ls (string) y axis quantile lines linestyle ('dashed', 'dotted', etc.)

yl_lw (integer or float) y axis quantile lines line width

yl_color (string color value) y axis quantile lines color

y_bands (boolean) if True, show a background color within every other y axis quantile membership area

yb_fc (string color value) y axis quantile bands background color

yb_alpha (float) y axis quantile bands color transparency value (0.0 to 1.0)

markersize (integer or float) size of chart scatter points

marker_alpha (integer or float) transparency setting for plot lines or points (0.0 to 1.0)

grid_alpha (float) transparency value for the chart grid corresponding to the x and y attribute values (not the quantile membership lines)

chart_style (string) any valid seaborn chart style name

full_xpcnt (boolean) if True, show full range percentage (0 to 100 percent) when a percentage attribute is displayed on the x axis

full_ypcnt (boolean) if True, show full range percentage (0 to 100 percent) when a percentage attribute is displayed on the y axis

xax_rotate (integer) rotation value (in degrees) for the x axis tick labels

qtick_size (integer or float) text size of the quantile membership tick labels

tick_size (integer or float) text size of the x and y attribute tick labels

label_size (integer or float) text size of x and y axis labels

border_size (integer or float) width of the chart border line (chart spines)

legend_size (integer or float) text size of chart legend

title_size (integer or float) text size of chart title

y_title_pos (float) vertical position of the chart title when attribute filtering has been applied. (typical values are 1.1 to 1.2)

box_height (float) chart height multiplier which slightly shrinks vertical chart area for proper printing (saving) purposes. This input does not affect the displayed values.

xsize, ysize (integer or float) plot size in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**eg_boxplot**(*df_list*, *eg_list*, *eg_colors*, *job_clip*, *attr_dict*, *measure='spcnt'*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *year_clip=2035*, *exclude_fur=False*, *saturation=0.8*, *chart_style='dark'*, *width=0.7*, *notch=True*, *show_whiskers=True*, *show_xgrid=True*, *show_ygrid=True*, *grid_alpha=0.4*, *grid_linestyle='solid'*, *whisker=1.5*, *fliersize=1.0*, *linewidth=0.75*, *suptitle_size=14*, *title_size=12*, *tick_size=11*, *label_size=12*, *xsize=12*, *ysize=8*, *image_dir=None*, *image_format='png'*)

create a box plot chart displaying ACTUAL attribute values (vs. differential values) from a selected dataset(s) for selected employee group(s).

inputs

df_list (list) list of datasets to compare, may be ds_dict (output of load_datasets function) string keys or dataframe variable(s) or mixture of each

eg_list (list) list of integers for employee groups to be included in analysis example: [1, 2, 3]

measure (string) attribute for analysis

eg_colors (list) list of colors for plotting the employee groups

attr_dict (dictionary) dataset column name description dictionary

ds_dict (dictionary) output from load_datasets function

job_clip (float) if measure is jnum or jobp, limit max y axis range to this value

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

year_clip (integer) only present results through this year

exclude_fur (boolean) remove all employees from analysis who are furloughed within the data model at any time (boolean)

chart_style (string) chart styling (string), any valid seaborn chart style

width (float) plotting width of boxplot or grouped boxplots for each year. a width of 1 leaves no gap between groups

notch (boolean) If True, show boxplots with a notch at median point

show_xgrid (boolean) include vertical grid lines on chart

show_ygrid (boolean) include horizontal grid lines on chart

grid_alpha (float) opacity value for grid lines

grid_linestyle (string) examples: 'solid', 'dotted', 'dashed'

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) text size of chart title

tick_size (integer or float) text size of x and y tick labels

label_size (integer or float) text size of x and y descriptive labels

xsize, ysize (integer or float) width and hieght of plot in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**eg_diff_boxplot**(*df_list*, *dfb*, *eg_list*, *eg_colors*, *job_levels*, *job_diff_clip*, *attr_dict*, *measure='spcnt'*, *comparison='baseline'*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *suptitle_size=14*, *title_size=12*, *tick_size=11*, *label_size=12*, *year_clip=None*, *exclude_fur=False*, *width=0.8*, *chart_style='dark'*, *notch=True*, *linewidth=1.0*, *xsize=12*, *ysize=8*, *image_dir=None*, *image_format='png'*)

create a DIFFERENTIAL box plot chart comparing a selected measure from computed integrated dataset(s) vs. a baseline (likely standalone) dataset or with other integrated datasets.

inputs

df_list (list) list of datasets to compare, may be ds_dict (output of load_datasets function) string keys or dataframe variable(s) or mixture of each

dfb (string or variable) baseline dataset, accepts same input types as df_list above

eg_list (list) list of integers for employee groups to be included in analysis example: [1, 2, 3]

eg_colors (list) corresponding plot colors for eg_list input

job_levels (integer) number of job levels in the data model (excluding furlough)

job_diff_clip (integer) if measure is jnum or jobp, limit y axis range to +/- this value

attr_dict (dictionary) dataset column name description dictionary

measure (string) differential data to compare

comparison (string) if 'p2p' (proposal to proposal), will compare proposals within the df_list to each other, otherwise will compare proposals to the baseline dataset (dfb)

ds_dict (dictionary) output from load_datasets function

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) text size of chart title

tick_size (integer or float) text size of x and y tick labels

label_size (integer or float) text size of x and y descriptive labels

year_clip (integer) only present results through this year if not None

exclude_fur (boolean) remove all employees from analysis who are furloughed within the data model at any time

use_eg_colors (boolean) use case-specific employee group colors vs. default colors

width (float) plotting width of boxplot or grouped boxplots for each year. a width of 1 leaves no gap between groups

chart_style (string) chart styling (string), any valid seaborn chart style

notch (boolean) If True, show boxplots with a notch at median point vs. only a line

xsize, ysize (integer or float) plot size in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**eg_multiplot_with_cat_order**(*df*, *mnum*, *measure*, *xax*, *job_strs*, *job_level_colors*, *job_levels*, *settings_dict*, *attr_dict*, *color_dict*, *egs=[]*, *ds_dict=None*, *fur_color=None*, *exclude_fur=False*, *plot_scatter=True*, *s=20*, *a=0.7*, *lw=0*, *job_bands_alpha=0.3*, *title_size=14*, *tick_size=12*, *label_pad=110*, *chart_style='whitegrid'*, *remove_ax2_border=True*, *lgd_h_adj=None*, *xsize=13*, *ysize=10*, *image_dir=None*, *image_format='png'*)

plot any dataset attributes as x or y values for comparison

when "cat_order" is selected as measure, show job category bands

inputs

df (dataframe) pandas dataframe input

mnum (integer) month number for analysis

measure (string) dataframe column name (attribute for analysis)

xax (string) x axis attribute

job_strs (list) list of job descriptions for labels (normally sdict['job_strs'])

job_level_colors (list) list of colors for job level zones (normally cdict['job_colors'])

job_levels (integer) number of job levels in model (sdict['num_of_job_levels'])

settings_dict (dictionary) program job settings dictionary

attr_dict (dictionary) program attribute name to attribute description dictionary

color_dict (dictionary) color dictionary

egs (list) list of employee groups for plotting

ds_dict (dictionary) output from load_datasets function

fur_color (string color value) if not None, color for furlough span color

exclude_fur (boolean) if True, remove furloughed employees from input data

plot_scatter (boolean) if True (default), plot a scatter chart, otherwise plot a line chart

s (integer or float) size of scatter markers if a plot_scatter input is True

a (float) transparency value for both line plots and scatter plots (0.0 to 1.0)

lw (integer or float) width of maker edge lines with a scatter plot

job_bands_alpha (float) transparency value for job level color spans

title_size (integer or float) text size of chart title

tick_size (integer or float) text size of chart tick labels

label_pad (integer) minimum padding between job description labels that would otherwise overlap

chart_style (string) any seaborn plotting style name

remove_ax2_border (boolean) if True, remove axis 2 (ax2) chart spines

xsize, ysize (integer or float) width and height of chart

lgd_h_adj (float) set to a small float value (for example: .02, -.01) to adjust the horizontal position of the chart legend if required. Use negative values to move left, positive values to move right

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**emp_quick_glance**(*empkey*, *df*, *ds_dict=None*, *title_size=14*, *tick_size=13*, *lw=4*, *chart_style='dark'*, *xsize=8*, *ysize=48*, *image_dir=None*, *image_format='png'*)

view basic stats for selected employee and proposal

A separate chart is produced for each measure.

inputs

empkey (integer) employee number (in data model)

df (dataframe) dataset to study, will accept string proposal name

ds_dict (dictionary) variable assigned to load_datasets function output

title_size (integer or float) text size of chart title

tick_size (integer or font) text size of chart tick labels

lw (integer or float) line width of plot lines

chart_style (string) any valid seaborn charting style

xsize, ysize (integer or float) size of chart display

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**filter_ds**(*ds*, *attr1=None*, *oper1=None*, *val1=None*, *attr2=None*, *oper2=None*, *val2=None*, *attr3=None*, *oper3=None*, *val3=None*, *return_title_string=True*)

Filter a dataset (dataframe) by attribute(s).

Filter process is ignored if attr(n) input is None. All attr, oper, and val inputs must be strings. Up to 3 attribute filters may be combined.

Attr, oper, and val inputs are combined and then evaluated as expressions.

If return_title_string is set to True, returns tuple (ds, title_string), otherwise returns ds.

inputs

ds (dataframe) the dataframe to filter

attr(n) (string) an attribute (column) to filter. Example: 'ldate'

oper(n) (string) an operator to apply to the attr(n) input. Example: '<='

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

return_title_string (boolean) If True, returns a string which dexcribes the filter(s) applied to the dataframe (ds)

matplotlib_charting.**group_average_and_median**(*dfc*, *dfb*, *eg_list*, *eg_colors*, *measure*, *job_levels*, *settings_dict*, *attr_dict*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1='0'*, *attr2=None*, *oper2='>='*, *val2='0'*, *attr3=None*, *oper3='>='*, *val3='0'*, *plot_median=False*, *plot_average=True*, *compare_to_dfb=True*, *use_filtered_results=True*, *show_full_yscale=False*, *job_labels=True*, *max_date=None*, *chart_style='whitegrid'*, *xsize=14*, *ysize=8*, *image_dir=None*, *image_format='png'*)

Plot group average and/or median for a selected attribute over time for compare and/or base datasets. Standalone data may be used as compare or baseline data.

Results may be further filtered/sliced by up to 3 constraints, such as age, longevity, or job level.

This function can plot basic data such as average list percentage or could, for example, plot the average job category rank for employees hired prior to a certain date who are over or under a certain age, for a selected integrated dataset and/or standalone data (or for two integrated datasets).

inputs

dfc (string or dataframe variable) comparative dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

dfb (string or dataframe variable) baseline dataset to plot (likely use standalone dataset here for comparison, but may plot and compare any dataset), may be a dataframe variable or a string key from the ds_dict dictionary object

eg_list (list) list of integers representing the employee groups to analyze (i.e. [1, 2])

eg_colors (list) list of colors for plotting the employee groups

measure (string) attribute (column) to compare, such as 'spcnt' or 'jobp'

job_levels (integer) number of job levels in the data model

settings_dict (dictionary) program settings dictionary generated by the build_program_files script

attr_dict (dictionary) dataset column name description dictionary

ds_dict (dictionary) dataset dictionary (variable assigned to the output of load_datasets function)

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

plot_meadian (boolean) plot the median of the measure for each employee group

plot_average (boolean) plot the average(mean) of the measure for each employee group

compare_to_dfb (boolean) plot average dfb[measure] data as dashed line. (likely show standalone data with dfb, or reverse and show standalone as primary and integrated as dfb) (dfb refers to baseline dataframe or dataset)

use_filtered_results (boolean) if True, use the same employees from the filtered proposal list. For example, if the dfc list is filtered by age only, the dfb list could be filtered by the same age and return the same employees. However, if the dfc list is filtered by an attribute which diverges from the dfb measurements for the same attribute, a different set of employees could be returned. This option ensures that the same group of employees from both the dfc (filtered first) list and the dfb list are compared. (dfc refers to the comparison proposal, dfb refers to baseline)

show_full_yscale (boolean) if measure input is one of these: 'jnum', 'nbnf', 'jobp', 'fbff', if True, show all job levels on chart. Otherwise, allow chart to autoscale with plotted data

job_labels (boolean) if measure input is one of these: 'jnum', 'nbnf', 'jobp', 'fbff', use job text description labels vs. number labels on the y axis of the chart (boolean)

max_date (date string) maximum chart date. If set to 'None', the maximum chart date will be the maximum date within the list data.

chart_style (string) option to specify alternate seaborn chart style

xsize, ysize (integer or float) x and y size of chart in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**job_count_bands**(*df_list*, *eg_list*, *job_colors*, *settings_dict*, *ds_dict=None*, *emp_list=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *fur_color=None*, *show_grid=True*, *max_date=None*, *plot_alpha=0.75*, *legend_alpha=0.9*, *legend_xadj=1.3*, *legend_yadj=1.0*, *legend_size=11*, *title_size=14*, *tick_size=12*, *label_size=13*, *chart_style='darkgrid'*, *xsize=13*, *ysize=8*, *image_dir=None*, *image_format='png'*)

area chart representing count of jobs available over time

This chart displays the future job opportunities for each employee group with various list proposals.

This is not a comparative chart (for example, with standalone data), it is simply displaying job count outcome over time. However, the results for the employee groups may be compared and measured for equity.

Inputs

df_list (list) list of datasets to compare, may be ds_dict (output of load_datasets function) string keys or dataframe variable(s) or mixture of each

eg_list (list) list of integers for employee groups to be included in analysis example: [1, 2, 3]

job_colors (list) list of colors to represent job levels

settings_dict (dictionary) program settings dictionary generated by the build_program_files script

ds_dict (dictionary) output from load_datasets function

emp_list (list) optional list of employee number(s) to plot (empkey attribute)

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

fur_color (color code in rgba, hex, or string style) custom color to signify furloughed job level band (otherwise, last color in job_colors input will be used)

max_date (date string) only include data up to this date example input: '1997-12-31'

plot_alpha (float, 0.0 to 1.0) alpha value (opacity) for area plot (job level bands)

legend_alpha (float, 0.0 to 1.0) alpha value (opacity) for legend markers

legend_xadj, legend_yadj (floats) adjustment input for legend horizontal and vertical placement

legend_size (integer or float) text size of legend labels

title_size (integer or float) text size of chart title

tick_size (integer or float) text size of x and y tick labels

label_size (integer or float) text size of x and y descriptive labels

chart_style (string) chart styling (string), any valid seaborn chart style

xsize, ysize (integer or float) plot size in inches (width and height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**job_count_charts**(*dfc*, *dfb*, *settings_dict*, *eg_colors*, *eg_list=None*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *plot_egs_sep=False*, *plot_total=True*, *xax='date'*, *year_max=None*, *chart_style='darkgrid'*, *base_ls='-'*, *prop_ls=':'*, *base_lw=1.6*, *prop_lw=2.5*, *suptitle_size=14*, *title_size=12*, *total_color='g'*, *xsize=5*, *ysize=4*, *image_dir=None*, *image_format='png'*)

line-style charts displaying job category counts over time.

optionally display employee group results on separate charts or together

inputs

dfc (dataframe) proposal (comparison) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

dfb (dataframe) baseline dataset; proposal dataset is compared to this dataset, may be a dataframe variable or a string key from the ds_dict dictionary object

settings_dict (dictionary) program settings dictionary generated by the build_program_files script

eg_colors (list) list of color values for plotting the employee groups, length is equal to the number of employee groups in the data model

eg_list (list) list of employee group codes to plot Example: [1, 2]

ds_dict (dictionary) variable assigned to load_datasets function output

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

plot_egs_sep (boolean) if True, plot each employee group job level counts separately

plot_total (boolean) if True, include the combined job counts on chart(s)

xax (string) x axis groupby attribute, options are 'date' or 'mnum', default is 'date'

year_max (integer) maximum year to include on chart Example: if input is 2030, chart would display data from beginning of data model through 2030 (integer)

base_ls (string) line style for base job count line(s)

prop_ls (string) line style for comparison (proposal) job count line(s)

base_lw (float) line width for base job count line(s)

prop_lw (float) line width for comparison (proposal) job count lines

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) chart title(s) font size

total_color (color value) color for combined job level count from all employee groups

xsize, ysize (integer or float) size of chart display in inches (width and height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**job_grouping_over_time**(*df*, *eg_list*, *jobs*, *job_colors*, *p_dict*, *plt_kind='bar'*, *ds_dict=None*, *rets_only=True*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *time_group='A'*, *display_yrs=40*, *legend_loc=4*, *chart_style='darkgrid'*, *suptitle_size=14*, *title_size=12*, *legend_size=13*, *tick_size=11*, *label_size=13*, *xsize=12*, *ysize=10*, *image_dir=None*, *image_format='png'*)

Inverted bar chart display of job counts by group over time. Various filters may be applied to study slices of the datasets.

The 'rets_only' option will display the count of employees retiring from each year grouped by job level.

developer TODO: fix x axis scaling and labeling when quarterly ("Q") or monthly ("M") time group option selected.

inputs

df (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

eg_list (list) list of unique employee group numbers within the proposal Example: [1, 2]

jobs (list) list of job label strings (for plot legend)

job_colors (list) list of colors to be used for plotting

p_dict (dictionary) employee group to string description dictionary

plt_kind (string) 'bar' or 'area' (bar recommended)

ds_dict (dictionary) output from load_datasets function

rets_only (boolean) calculate for employees at retirement age only

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

time_group (string) group counts/percentages by year ('A'), quarter ('Q'), or month ('M')

display_years (integer) when using the bar chart type display, evenly scale the x axis to include the number of years selected for all group charts

legend_loc (integer) matplotlib legend location number code

2	9	1
6	10	7
3	8	4

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) text size of chart title

legend_size (integer or float) text size of chart legend labels

tick_size (integer or float) text size of x and y tick labels

label_size (integer or float) text size of x and y descriptive labels

xsize, ysize (integer or float) size of each chart in inches (width, height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**job_level_progression**(*df*, *emp_list*, *through_date*, *settings_dict*, *color_dict*, *eg_colors*, *band_colors*, *ds_dict=None*, *rank_metric='cat_order'*, *chart_style='white'*, *show_implementation_date=True*, *job_bands_alpha=0.1*, *max_plots_for_legend=5*, *xgrid_alpha=0.65*, *xgrid_linestyle='dotted'*, *ygrid_alpha=0.5*, *ygrid_linestyle='dotted'*, *tick_size=13*, *job_descr_size=12.5*, *job_descr_pad=115*, *label_size=15*, *title_size=18*, *xsize=12*, *ysize=10*, *image_dir=None*, *image_format='png'*)

show employee(s) career progression through job levels regardless of actual positioning within integrated seniority list.

This x axis of this chart represents rank within job category. There is an underlying stacked area chart representing job level bands, adjusted to reflect job count changes over time.

This chart reveals actual career path considering no bump no flush, special job assignment rights/restrictions, and furlough/recall events.

Actual jobs held may not be correlated to jobs normally associated with a certain list percentage for many years due to job assignment factors.

inputs

df (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

emp_list (list) list of empkeys to plot

through_date (date string) string representation of y axis date limit, ex. '2025-12-31'

settings_dict (dictionary) program settings dictionary generated by the build_program_files script

color_dict (dictionary) dictionary containing color list string titles to lists of color values generated by the build_program_files script

eg_colors (list) colors to be used for employee line plots corresponding to employee group membership

band_colors (list) list of colors to be used for stacked area chart which represent job level bands

ds_dict (dictionary) output from load_datasets function

rank_metric (string) column name for y axis chart ranking. Currently only 'cat_order' is valid.

chart_style (string) any valid seaborn plotting chart style name

show_implementation_date (boolean) plot a vertical dashed line at the implementation date

job_bands_alpha (float) opacity level of background job bands stacked area chart

max_plots_for_legend (integer) if number of plots more than this number, reduce plot linewidth and remove legend

xgrid_alpha, ygrid_alpha (float) transparency value for grid. x and y axis may be set independently

xgrid_linestyle, ygrid_linestyle (string) matplotlib line style for grid, such as "dotted" or "dashed". x and y axis may be set independently

job_descr_size (integer or float) font size of job description text labels on right side of chart

job_descr_pad (integer) padding to add between job description labels when they would otherwise overlap

tick_size (intger or float) font size of tick labels

job_descr_size (integer or float) font size of job description labels

label_size (integer or float) font size of axis labels

title_size (integer or label) font size of title

xsize, ysize (integer or float) plot size in inches (width, height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**job_time_change**(*ds_list*, *ds_base*, *eg_list*, *job_colors*, *job_strs_dict*, *job_levels*, *attr_dict*, *xax*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *marker='o'*, *edgecolor='k'*, *linewidth=0.05*, *size=25*, *alpha=0.95*, *bg_color='#ffffff'*, *x_max=1.02*, *limit_yax=False*, *ylimit=40*, *zeroline_color='m'*, *zeroline_width=1.5*, *pos_neg_face=True*, *pos_neg_face_alpha=0.03*, *legend_job_strings=True*, *legend_position=1.18*, *legend_marker_size=130*, *suptitle_size=16*, *title_size=14*, *tick_size=12*, *chart_style='whitegrid'*, *label_size=13*, *xsize=12*, *ysize=10*, *image_dir=None*, *image_format='png'*, *experimental=False*)

Plots a scatter plot displaying monthly time in job differential, by proposal and employee group. X axis percentage reflects first month within each comparative dataset, which will be the same as standalone for all groups unless the data model implementation date occurs at month zero.

inputs

ds_list (list) list of datasets to compare, may be ds_dict (output of load_datasets function) string keys or dataframe variable(s) or mixture of each

ds_base (string or variable) baseline dataset, accepts same input types as ds_list above

eg_list (list) list of integers for employee groups to be included in analysis example: [1, 2, 3]

job_levels (integer) number of job levels in the data model

job_colors (list) list of color values for job level plotting

job_strs_dict (dictionary) dictionary of job code (integer) to job description label

attr_dict (dictionary) dataset column name description dictionary

xax (string) list percentage attrubute, i.e. spcnt or lspcnt

ds_dict (dictionary) output from load_datasets function

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

job_colors (list) list of color values for the job level plotting

job_strs_dict (dictionary) job number to job label dictionary

marker (string) scatter chart matplotlib marker type

edgecolor (color value) matplotlib marker edge color

linewidth (integer or float) matplotlib marker edge line size

size (integer or float) size of markers

alpha (float) marker alpha (transparency) value

bg_color (color value) background color of chart if not None

x_max (integer or float) high limit of chart x axis

limit_yax (integer or float) if True, restrict plot y scale to this value may be used to prevent outliers from exagerating chart scaling

ylimit (integer or float) y axis limit if limit_yax is True

zeroline_color (color value) color for zeroline on chart

zeroline_width (integer or float) width of zeroline

pos_neg_face (boolean) if True, apply a light green tint to the chart area above the zero line, and a light red tint below the line

legend_job_strings (boolean) if True, use job description strings in legend vs. job numbers

legend_position (float) controls the horizontal position of the legend

legend_marker_size (integer or float) adjusts the size of the legend markers

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) text size of chart title

tick_size (integer or float) text size of chart tick labels

xsize, ysize (integer or float) x and y size of each plot in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

experimental (boolean) show additional output under development consisting of a table, heatmap, and bar chart

matplotlib_charting.**job_transfer**(*dfc*, *dfb*, *eg*, *job_colors*, *job_levels*, *job_strs*, *p_dict*, *ds_dict=None*, *gb_period='M'*, *min_date=None*, *max_date=None*, *tgt_jobs_list=None*, *job_alpha=0.85*, *chart_style='whitegrid'*, *fur_color=None*, *draw_face_color=False*, *draw_grid=True*, *grid_alpha=0.2*, *zero_line_color='m'*, *ytick_interval=None*, *y_limit=None*, *title_size=14*, *legend_size=12*, *xsize=14*, *ysize=9*, *image_dir=None*, *image_format='png'*)

plot a differential stacked area chart displaying color-coded job transfer counts over time.

Output chart is actually 2 area charts (one for positive values and one for negative values) displayed on a shared axis.

inputs

dfc (dataframe) proposal (comparison) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

dfb (dataframe) baseline dataset; proposal dataset is compared to this dataset, may be a dataframe variable or a string key from the ds_dict dictionary object

eg (integer) integer code for employee group

job_colors (list) list of colors for job levels, may be value from color dictionary

job_levels (integer) number of job levels in data model

job_strs (list) list of job descriptions (labels)

p_dict (dictionary) dictionary of employee number codes to verbose string description, (normally "p_dict_verbose" from the settings dictionary)

Example:

```
{0: 'Standalone', 1: 'Acme', 2: 'Southern'}
```

ds_dict (dictionary) output from load_datasets function

gb_period (string) group_by period. default is 'M' for monthly, other options are 'Q' for quarterly and 'A' for annual

min_date (string date format) if set, analyze job transfer data from this date forward

max_date (string date format) if set, analyze job transfer data up to this date

tgt_jobs_list (list) if not None, only plot job level(s) in this list

job_alpha (float) chart alpha level for job transfer plotting (0.0 - 1.0)

chart_style (string) seaborn plotting library style

fur_color (color code in rgba, hex, or string style) custom color to signify furloughed employees (otherwise, last color in job_colors input will be used)

draw_face_color (boolean) apply a transparent background to the chart, red below zero and green above zero

draw_grid (boolean) show major tick label grid lines

grid_alpha (float) opacity setting for grid lines (0.0 - 1.0)

zero_line_color (color value) color of the horizontal line a zero

ytick_interval (integer) optional manual ytick spacing setting (function has auto-spacing built in)

y_limit (integer) optional manual y axis chart limit (enter positive value only). This input may be used to "lock" vertical scaling (shut off auto_scaling) for comparing gains and losses between proposals and employee groups.

title_size (integer or float) chart title text size

legend_size (integer or float) chart legend text size

xsize (integer or float) horizontal size of chart

ysize (integer or float) vertical size of chart

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**make_color_list**(*num_of_colors=10, start=0.0, stop=1.0, exclude=None, reverse=False, cm_name_list=['Set1'], return_list=True, return_dict=False, print_all_names=False, palplot_cm_name=False, palplot_all=False*)

Utility function to generate list(s) of colors (rgba format), any length and any from any

section of any matplotlib colormap.

The function can return a list of colors, a dictionary of colormaps to color lists, plot result(s) as seaborn palplot(s), and print out the names of all of the colormaps available.

The end goal of this function is to provide customized color lists for plotting.

inputs

num_of_colors (integer) number of colors to produce for the output color list(s), used within the cm_subsection data calculation

start (float) the starting point within the selected colormap to begin the spectrum color selection (0.0 to 1.0), used within the cm_subsection data calculation

stop (float) the ending point within the selected colormap to end the spectrum color selection (0.0 to 1.0), used within the cm_subsection data calculation

exclude (list) list of 2 floats representing a section of the colormap(s) to remove before calculating the result list(s).

reverse (boolean) reverse the color list order which reverses the color spectrum

cm_name_list (list) any matplotlib colormap name(s)

return_list (boolean) if True, return a list of rgba color codes for the cm_name_list colormap input only, or (if the return_dict input is set to True) a dictionary of all colormap names to all of the resultant corresponding calculated color lists using the cm_subsection data

return_dict (boolean) if True (and return_list is True), return a dictionary of all colormap names to all of the resultant corresponding calculated color lists

print_all_names (boolean) if True (and return_list is False), print all the names of available matplotlib colormaps

palplot_cm_name (boolean) if True (and return_list is set to False), plot a seaborn palplot of the color list produced with the cm_name_list colormap input using the cm_subsection data

palplot_all (boolean) if True (and return_list and palplot_cm_name are False), plot a seaborn palplot for all of the color lists produced from all available matplotlib colormaps using the cm_subsection data

matplotlib_charting.**mark_quantiles**(*df*, *quantiles=10*)

add a column to the input dataframe identifying quantile membership as integers (the column is named "quantile"). The quantile membership (category) is calculated for each employee group separately, based on the employee population in month zero.

The output dataframe permits attributes for employees within month zero quantile categories to be be analyzed throughout all the months of the data model.

The number of quantiles to create within each employee group is selected by the "quantiles" input.

The function utilizes numpy arrays and functions to compute the quantile assignments, and pandas index data alignment feature to assign month zero quantile membership to the long-form, multi-month output dataframe.

This function is used within the quantile_groupby function.

inputs

df (dataframe) Any pandas dataframe containing an "eg" (employee group) column

quantiles (integer) The number of quantiles to create.

example:

If the input is 10, the output dataframe will be a column of integers 1 - 10. The count of each integer will be the same. The first quantile members will be marked with a 1, the second with 2, etc., through to the last quantile, 10.

matplotlib_charting.**multiline_plot_by_emp**(*df*, *measure*, *xax*, *emp_list*, *job_levels*, *ret_age*, *color_list*, *job_str_list*, *sdict*, *attr_dict*, *ds_dict=None*, *plot_jobp=False*, *show_implementation_date=True*, *through_date=None*, *pcnt_ylimit=1.0*, *chart_style='ticks'*, *linewidth=3*, *line_alpha=0.7*, *grid_linestyle='dotted'*, *grid_alpha=0.75*, *legend_size=14*, *label_size=13*, *tick_size=13*, *title_size=18*, *xsize=12*, *ysize=9*, *image_dir=None*, *image_format='png'*)

select example individual employees and plot career measure from selected dataset attribute, i.e. list percentage, career earnings, job level, etc.

inputs

df (dataframe) dataset to examine, may be a dataframe or a string key with the ds_dict dictionary object

measure (string) dataset attribute to plot. Usually only one attribute to plot, but may be more than one, such as 'jnum' and 'jobp'

xax (string) dataset attribute for x axis

emp_list (list) list of employee numbers or ids

job_levels (integer) number of job levels in model

ret_age (float) retirement age (example: 65.0)

color list (list) list of colors for plotting

job_str_list (list) list of string job descriptions corresponding to number of job levels

sdict (dictionary) program settings dictionary

attr_dict (dictionary) dataset column name description dictionary

ds_dict (dictionary) output of the load_datasets function, dictionary. This keyword argument must be set if a string key is used as the df input.

plot_jobp (boolean) if measure input is 'jnum', also plot 'jobp' if set to True

show_implementation_date (boolean) if True and "xax" input is "date", plot a vertical line at the implementation date

chart_style (string) any seaborn plotting style name

linewidth (integer or float) width of chart solid lines

line_alpha (float) transparency value of the plotted lines (0.0 to 1.0)

grid_linestyle (string) matplotlib line style for grid, such as "dotted" or "solid"

grid_alpha transparency value for grid (0.0 to 1.0)

legend_size (integer or float) text size of chart legend

label_size (integer or float) font size of x and y axis labels

tick_size (integer or float) font size of chart tick labels

title_size (integer or float) font size of chart title

xsize, ysize (integer or float) plot size in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**numeric_test**(*value*)

determine if a variable is numeric

returns a boolean value

input

value any variable

matplotlib_charting.**parallel**(*df_list*, *dfb*, *eg_list*, *measure*, *month_list*, *job_levels*, *eg_colors*, *dict_settings*, *attr_dict*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *left=0*, *stride_list=None*, *chart_style='whitegrid'*, *grid_color='.7'*, *suptitle_size=14*, *title_size=12*, *facecolor='w'*, *xsize=6*, *ysize=8*, *image_dir=None*, *image_format='png'*)

Compare positional or value differences for various proposals with a baseline position or value for selected months.

The vertical lines represent different proposed lists, in the order from the df_list list input.

inputs

df_list (list) list of datasets to compare, may be ds_dict (output of load datasets function) string keys or dataframe variable(s) or mixture of each

dfb (string or variable) baseline dataset, accepts same input types as df_list above. The order of the list is reflected in the chart x axis lables

eg_list (list) list of employee group integer codes to compare example: [1, 2]

measure (string) dataset attribute to compare

month_list (list) list of month numbers for analysis. the function will plot comparative data from each month listed

job_levels (integer) number of job levels in data model

eg_colors (list) list of colors to represent the employee groups

dict_settings (dictionary) program settings dictionary generated by the build_program_files script

attr_dict (dictionary) dataset column name description dictionary

ds_dict (dictionary) output from load_datasets function

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

left (integer) integer representing the list comparison to plot on left side of the chart(s). zero (0) represents the standalone results and is the default. 1, 2, or 3 etc. represent the first, second, third, etc. dataset results in df_list input order

stride_list (list) optional list of dataframe strides for plotting every other nth result (must be same length and correspond to eg_list)

grid_color (string) string name for horizontal grid color

facecolor (color value) chart background color

xsize, ysize (integer or float) size of individual subplots (width, height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**pct_format**()

Apply "to_percent" custom format for chart tick labels

matplotlib_charting.**percent_bins**(*eg*, *base*, *compare*, *measure='spcnt'*, *by_year=True*, *quantiles=20*, *time_col='date'*, *agg_method='median'*)

Return a tuple of two dataframes containing differential percentage bin counts, one containing positive counts and another containing negative counts.

This function first compares list percentage between two datasets on a grouped time period basis (annual or monthly), then counts the number of employees within specified percentage gain or loss quantiles.

The counts are returned in dataframes with indexes reflecting the quantiles and columns representing the grouped time period.

This function is used in the percent_diff_bins plotting function.

inputs

eg (integer) employee group code

base (dataframe) baseline dataframe (dataset) containing a list percentage column

compare (dataframe) comparison dataframe (dataset) containing a list percentage column

measure (string) dataset percentage attribute column ('spcnt' or 'lspcnt')

by_year (boolean) if True, group employee percentage differentials by year, otherwise by time_col input

quantiles (integer) number of quantiles to measure. An input of 20 would translate to quantiles of 5% each (100 / 20).

time_col (string) if by_year is False, group percentage differentials by this time unit. Inputs may be "mnum" or "date".

agg_method (string) quantile bin aggregation method. Inputs may be "mean" or "median"

matplotlib_charting.**percent_diff_bins**(*compare*, *base*, *eg*, *measure='spcnt'*, *kind='bar'*, *quantiles=40*, *num_display_colors=25*, *area_xax='date'*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *man_plotlim=None*, *invert_barh=False*, *chart_style='ticks'*, *cmap_pos='tab20c'*, *cmap_neg='tab20c'*, *zero_line_color='m'*, *bright_bg=False*, *bg_color='#ffffe6'*, *title_size=14*, *legend_size=12.5*, *xsize=16*, *ysize=10*, *image_dir=None*, *image_format='png'*)

Display employee group counts within differential list percentage bins over time.

Chart style options include bar, barh, and area.

Selectable inputs include the number of percentile bins, chart colors and the number of colors in the color cycle representing the bins.

The analysis groups may be targeted by up to three attribute value filters.

inputs

compare (dataframe) comparison dataframe (dateset)

base (dataframe) baseline dataframe (dataset)

eg (integer) employee group code

measure (string) list percentage attribute for comparison ('spcnt' or 'lspcnt')

kind (string) chart style ('bar', 'barh', or 'area')

quantiles (integer) the number of differential percentage bins. If the input is 40, each bin width will be 2.5% (100 / 40)

num_display_colors (integer) the number of distinct colors to create from the cmap inputs. If the input is less than the number of bins found for display, the colors display will cycle or repeat as necessary.

area_xax (string) attribute to use for the chart when the kind input is set to 'area'. Inputs may be 'mnum' or 'date'.

ds_dict (dictionary) variable assigned to the output of the load_datasets function. This keyword variable must be set if string dictionary keys are used as inputs for the dfc and/or dfb inputs.

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

man_plotlim (integer) if not None, restrict chart differential axis to this value. Otherwise, limit is set by an algorithm.

invert_barh (boolean) If 'kind' input is set to 'barh', if True, invert the chart y axis

chart_style (string) any valid seaborn plotting style name

cmap_pos (string) any matplotlib colormap name representing colors to be applied to positive chart values

cmap_neg (string) any matplotlib colormap name representing colors to be applied to negative chart values

zero_line_color (color value) color to be applied to the chart zero line

bright_bg (boolean) if True, color the chart background with the 'bg_color' color value

bg_color (color value) color to use for the chart background if 'bright_bg' is True

title_size (integer or float) text size for the chart title

legend_size (integer or float) text size for the chart legend

xsize, ysize (integers or floats) Width and height of chart in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

`matplotlib_charting.`**`pprint_dict`**(*dct*, *marker1='#'*, *marker2=''*, *skip_line=True*)

print the key-value pairs in a horizontal, organized fashion.

inputs

dct (dictionary) the dictionary to print

marker1, marker2 prefix and suffix for the dictionary key headers

matplotlib_charting.**quantile_bands_over_time**(*df*, *eg*, *measure*, *bins=20*, *ds_dict=None*, *year_clip=None*, *kind='area'*, *quantile_ticks=False*, *cm_name='tab20c'*, *chart_style='ticks'*, *quantile_alpha=0.75*, *grid_alpha=0.4*, *custom_start=0.0*, *custom_finish=1.0*, *alt_bg_color=False*, *bg_color='#faf6eb'*, *legend_size=13*, *label_size=13*, *xsize=14*, *ysize=8*, *image_dir=None*, *image_format='png'*)

Visualize quantile distribution for an employee group over time for a selected proposal.

This chart answers the question of where the different employee groups will be positioned within the seniority list for future months and years.

Note: this is not a comparative study. It is simply a presentation of resultant percentage positioning.

The chart contains a background grid for reference and may display quantiles as integers or percentages, using a bar or area type display, and includes several chart color options.

inputs

df (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

eg (integer) employee group number

measure (string) a list percentage input, either 'spcnt' or 'lspcnt'

bins (integer) number of quantiles to calculate and display

ds_dict (dictionary) output from load_datasets function

year_clip (integer) maximum year to display on chart (requires 'clip' input to be True)

kind (string) type of chart display, either 'area' or 'bar'

quantile_ticks (boolean) if True, display integers along y axis and in legend representing quantiles. Otherwise, present percentages.

cm_name (string) colormap name (string), example: 'Set1'

chart_style (string) style for chart output, any valid seaborn plotting style name

quantile_alpha (float) alpha (opacity setting) value for quantile plot

grid_alpha (float) opacity setting for background grid

custom_start (float) custom colormap start level (a section of a standard colormap may be used to create a custom color mapping)

custom_finish (float) custom colormap finish level

alt_bg_color (boolean) if True, set the background chart color to the bg_color input value

bg_color (color value) color for chart background if 'alt_bg_color' is True (string)

legend_size (integer or float) text size for chart legend

label_size (intger or float) text size for chart x and y axis labels

xsize, ysize (integer or float) chart size inputs in inches (width, height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**quantile_groupby**(*dataset_list*, *eg_list*, *measure*, *quantiles*, *eg_colors*, *band_colors*, *settings_dict*, *attr_dict*, *job_dict*, *groupby_method='median'*, *xax='date'*, *ds_dict=None*, *num_cat_order_yticks=10*, *through_date=None*, *verbose_title=True*, *plot_total=True*, *show_job_bands=True*, *show_grid=True*, *plot_implementation_date=True*, *draw_reserve_levels=False*, *custom_color=False*, *cm_name='Set1'*, *start=0.0*, *stop=1.0*, *exclude=None*, *reverse=False*, *chart_style='whitegrid'*, *remove_ax2_border=True*, *line_width=1*, *use_dashed_line_compare=True*, *bg_color='.98'*, *job_bands_alpha=0.15*, *line_alpha=0.7*, *grid_alpha=0.3*, *title_size=14*, *tick_size=12*, *label_size=13*, *label_pad=110*, *xsize=12*, *ysize=10*, *image_dir=None*, *image_format='png'*)

Plot representative values of a selected attribute measure for each employee group quantile over time.

Multiple employee groups may be plotted at the same time. Job bands may be plotted as a chart background to display job level progression when the measure input is set to "cat_order".

Two data models may be plotted and compared on the same chart. Only the first employee group found within the eg_list input will be compared when plotting more than one dataset.

Example use case: plot the average job category rank of each employee quantile group, from the start date though the life of the data model.

The quantile group attribute may be analyzed with any of the following methods:

[mean, median, first, last, min, max]

If the eg_list input list contains a single employee group code and the custom_color input is set to "True", the color of the plotted quantile result lines will be a spectrum of colors. The following inputs are related to the custom color generation:

[cm_name, start, stop, exclude, reverse]

The above inputs will be used by the make_color_list function located within this module to produce a list of colors with a length equal to the quantiles input. (Please see the docstring for the make_color_list function for further explaination). If the quantiles input is set to a relatively high value (100-200), the impact on the career profiles of the employee groups is easily discernible when using a qualitative color map.

inputs

dataset_list (dataframes) A list of long-form dataframes, each of which contains "date" (and "mnum" if xax input is set to "mnum") and "eg" columns and at least one attribute column for analysis. The normal input is a list of calculated datasets with many attribute columns. The list may only hold one or two datasets.

eg_list (list) List of eg (employee group) codes for analysis. The order of the employee codes will determine the z-order of the plotted lines, last employee group plotted on top of the others.

measure (string) Attribute column name

quantiles (integer) The number of quantiles to create and plot for each employee group in the eg_list input.

eg_colors (list) list of color values for plotting the employee groups

band_colors (list) list of color values for plotting the background job level color bands when the using a measure of 'cat_order' with the 'show_job_bands' variable set to True

settings_dict (dictionary) program settings dictionary generated by the build_program_files script

attr_dict (dictionary) dataset column name description dictionary

job_dict (dictionary) dictionary containing basic to enhanced job level conversion data. This is likely the settings dictionary "jd" value.

groupby_method (string) The method applied to the attribute data within each quantile. The allowable methods are listed in the description above. Default is 'median'.

xax (string) The first groupby level and x axis value for the analysis. This value defaults to "date" which represents each month of the model. Alternatively, "mnum" may be used.

ds_dict (dictionary) A dictionary containing string to dataframes, used if df input is not a dataframe but a string key (examples: 'standalone', 'p1')

num_cat_order_yticks (int) approximate number of y axis ticks to display on the lefthand side of the chart when "cat_order" is selected as the "measure" input. The actual number of ticks displayed will be adjusted to match an optimal numerical

interval between tick values. This input does not have a linear relationship with the output and may require a significant input change to affect the chart display.

through_date (date string) If set as a date string, such as '2020-12-31', only show results up to and including this date.

verbose_title (boolean) If True, chart title will use the long descriptions for each employee group from the settings.xlsx input file, proposal_dictionary worksheet. Otherwise, the eg number codes will be used in the title

plot_total (boolean) If True, plot a dotted gray line representing the total count of active pilots over time (only when "measure" input is set to "cat_order" and "show_job_bands" input is True)

show_job_bands If measure is set to "cat_order", plot properly scaled job level color bands on chart background

show_grid (boolean) If True, plot a grid on the chart

plot_implementation_date If True and the xax argument is set to "date", plot a dashed vertical line at the implementation date.

draw_reserve_levels (boolean) If True and basic job levels have been selected via the settings.xlsx "scalars" worksheet, "enhanced jobs" setting, draw a horizontal red dashed line within each basic job category level representing the upper limit of reserve status

custom_color (boolean) If set to True, will permit a custom color spectrum to be produced for plotting a single employee group "cat_order" result (color map is selected with the cm_name input)

cm_name (string) The colormap name to be used for the custom color option

start (float) The starting point of the colormap to begin a custom color list generation (0.0 to less than 1.0)

stop (float) The ending point of the colormap to finish a custom color list generation (greater than 0.0 to 1.0)

exclude (list) A list of 2 floats between 0.0 and 1.0 describing a section of the original colormap to exclude from a custom color list generation. (Example [.45, .55], the middle of the list excluded)

reverse (boolean) If True, reverse the sequence of the custom color list

chart_style (string) set the chart plot style for ax1 from the avialable seaborn plotting themes:

["darkgrid", "whitegrid", "dark", "white", and "ticks"]

The default is "whitegrid".

remove_ax2_border (boolean) if "cat_order" is set as the measure input and the show_job_bands input is set True, a second axis is generated to be the container for the job level labels. The chart style for ax2 is "white" which avoids unwanted grid lines but includes a black solid chart border by default. This ax2 border may be removed if this input is set to True. (The border may be displayed if the chart_style input (for ax1) is set to "white" or "ticks").

line_width (float) The width of the plotted lines. Default is .75

use_dashed_line_compare (boolean) If True, when comparing output from 2 datasets, plot the second dataset output with a dashed line, otherwise use a solid line

bg_color (color value) The background color for the chart. May be a color name, color abreviation, hex value, or decimal between 0 and 1 (shades of black)

job_bands_alpha (float) If show_job_bands input is set to True and measure is set to "cat_order", this input controls the alpha or transparency of the background job level bands. (0.0 to 1.0)

line_alpha (float) Transparency value of plotted lines (0.0 to 1.0)

grid_alpha (float) Transparency value of grid lines (0.0 to 1.0)

title_size (integer or float) Font size value for title

tick_size (integer or float) Font size value for chart tick (value) labels

label_size (integer or float) Font size value for x and y unit labels

xsize, ysize (integers or floats) Width and height of chart in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**quantile_years_in_position**(*dfc*, *dfb*, *job_levels*, *num_bins*, *job_str_list*, *p_dict*, *color_list*, *style='bar'*, *plot_differential=True*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *chart_style='darkgrid'*, *grid_alpha=None*, *custom_color=False*, *cm_name='Dark2'*, *start=0.0*, *stop=1.0*, *fur_color=None*, *flip_x=False*, *flip_y=False*, *rotate=False*, *gain_loss_bg=False*, *bg_alpha=0.05*, *normalize_yr_scale=False*, *year_clip=30*, *suptitle_size=14*, *title_size=12*, *xsize=12*, *ysize=12*, *image_dir=None*, *image_format='png'*)

stacked bar or area chart presenting the time spent in the various job levels for quantiles of a selected employee group.

inputs

dfc (string or dataframe variable) text name of proposal (comparison) dataset to explore (ds_dict key) or dataframe

dfb (string or dataframe variable) text name of baseline dataset to explore (ds_dict key) or dataframe

job_levels (integer) the number of job levels in the model

num_bins (integer) the total number of segments (divisions of the population) to calculate and display

job_str_list (list) a list of strings which correspond with the job levels, used for the chart legend example: jobs = ['Capt G4', 'Capt G3', 'Capt G2', ...]

p_dict (dictionary) dictionary used to convert employee group numbers to text, used with chart title text display

color_list (list) a list of color codes for the job level color display

style (string) option to select 'area' or 'bar' to determine the type of chart output. default is 'bar'.

plot_differential (boolean) if True, plot the difference between dfc and dfb values

ds_dict (dictionary) variable assigned to the output of the load_datasets function. This keyword variable must be set if string dictionary keys are used as inputs for the dfc and/or dfb inputs.

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

chart_style (string) any valid seaborn plotting style name

custom_color, cm_name, start, stop (boolean, string, float, float) if custom color is set to True, create a custom color map from the cm_name color map style. A portion of the color map may be selected for customization using the start and stop inputs.

fur_color (color code in rgba, hex, or string style) custom color to signify furloughed employees (otherwise, last color in color_list input will be used)

flip_x (boolean) 'flip' the chart horizontally if True

flip_y (boolean) 'flip' the chart vertically if True

rotate (boolean) transpose the chart output

gain_loss_bg (boolean) if True, apply a green and red background to the chart in the gain and loss areas

bg_alpha (float) the alpha of the gain_loss_bg (if selected)

normalize_yr_scale (boolean) set all output charts to have the same x axis range

yr_clip (integer) max x axis value (years) if normalize_yr_scale set True

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) text size of chart title

xsize, ysize (integer or float) size of chart display

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples: 'svg', 'png'

matplotlib_charting.**rows_of_color**(*df*, *mnum*, *measure_list*, *eg_colors*, *jnum_colors*, *dict_settings*, *ds_dict=None*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *cols=150*, *eg_list=None*, *job_only=False*, *jnum=1*, *shrink_to_fit=False*, *cell_border=True*, *eg_border_color='.2'*, *job_border_color='.2'*, *chart_style='whitegrid'*, *fur_color=None*, *empty_color='#737373'*, *suptitle_size=14*, *title_size=12*, *legend_size=14*, *xsize=15*, *ysize=9*, *image_dir=None*, *image_format='png'*)

plot a heatmap with the color of each rectangle representing an employee group, job level, or status.

This chart will show a position snapshot indicating the distribution of employees within the entire population, employees holding a certain job, or a combination of the two.

For example, all employees holding a certain job in month 36 may be plotted with original group delineated by color. Or, all employees from one group may be shown with the different jobs for that group displayed with different colors.

Also will display any other category such as a special group such as furloughed employees. Input dataframe must have a numerical representation of the selected measure, i.e. furloughed indicated by a 1, and others with a 0.

inputs

df (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

mnum (integer) month number of dataset to analyze

measure_list (list) list form input, 'categorical' only such as employee group number or job number, such as ['jnum'], or ['eg'] ['eg', 'fur'] is also valid when highlighting furloughees

eg_colors (list) colors to use for plotting the employee groups. the first color in the list is used for the plot 'background' and is not an employee group color

jnum_colors (list) job level plotting colors, list form

ds_dict (dictionary) output from load_datasets function

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

cols (integer) number of columns to construct for the heatmap plot

eg_list (list) employee group integer code list (if used), example: [1, 2]

job_only (boolean) if True, plot only employees holding the job level identified with the jnum input

jnum (integer) job level distribution to plot if job_only input is True

shrink_to_fit (boolean) if True, adjust the size of the heatmap to match the size of the filtered monthly data. If False, maintain the number of cells in the heatmap to be equal to the starting size of the employee population

cell_border (boolean) if True, show a border around the heatmap cells

eg_border_color (color value) color of cell border if measure_list includes 'eg' (employee group)

job_border_color (color value) color of cell border when plotting job information

chart_style (string) underlying chart style, any valid seaborn chart style (string)

fur_color (color code in rgba, hex, or string style) custom color to signify furloughed employees (otherwise, last color in jnum_colors input will be used)

empty_color (color value) cell color for cells with no data

suptitle_size (integer or float) text size of chart super title

title_size (integer or float) text size of chart title

legend_size (integer or float) text size of chart legend

xsize, ysize (integer or float) size of chart in inches (width, height)

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**single_emp_compare**(*emp*, *measure*, *df_list*, *xax*, *job_strs*, *eg_colors*, *p_dict*, *job_levels*, *attr_dict*, *ds_dict=None*, *chart_style='whitegrid'*, *standalone_color='#ff00ff'*, *title_size=14*, *tick_size=12*, *label_size=13*, *legend_size=14*, *xsize=12*, *ysize=8*, *image_dir=None*, *image_format='png'*)

Select a single employee and compare proposal outcome using various calculated measures.

inputs

emp (integer) empkey for selected employee

measure (string) calculated measure to compare examples: 'jobp' or 'cpay'

df_list (list) list of calculated datasets to compare

xax (string) dataset column to set as x axis

job_strs (list) string job description list

eg_colors (list) list of colors to be assigned to line plots

p_dict (dictionary) dictionary containing eg group integer to eg string descriptions

job_levels (integer) number of jobs in the model

attr_dict (dictionary) dataset column name description dictionary

ds_dict (dictionary) output from load_datasets function

chart_style (string) any valid seaborn plotting style

standalone_color (color value) color of standalone plot (This function assumes one proposal from each group, any additional proposal is assumed to be standalone)

title_size (integer or float) text size of chart title

tick_size (integer or float) text size of chart tick labels

label_size (integer or float) text size of x and y axis chart labels

legend_size (integer or float) text size of chart legend

xsize, ysize (integer or float) width and height of output chart in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**slice_ds_by_filtered_index**(*df*, *ds_dict=None*, *mnum=0*, *attr='age'*, *attr_oper='>='*, *attr_val=50*)

filter an entire dataframe by only selecting rows which match the filtered results from a target month. In other words, zero in on a slice of data from a particular month, such as employees holding a specific job in month 25. Then, using the index of those results, find only those employees within the entire dataset as an input for further analysis within the program.

The output may be used as an input to a plotting function or for other analysis. This function may also be used repeatedly with various filters, using output of one execution as input for another execution.

inputs

df (dataframe, can be proposal string name) the dataframe (dataset) to be filtered

ds_dict (dictionary) A dictionary containing string to dataframes, used if ds_def input is not a dataframe

mnum (integer) month number of the data to filter

attr (string) attribute (column) to use during filter

oper (string) operator to use, such as '<=' or '!='

attr_val (integer, float, date as string, string (as appropriate)) attr1 limiting value (combined with oper) as string

Example filter: jnum >= 7 (in mnum month)

matplotlib_charting.**stripplot_dist_in_category**(*df*, *job_levels*, *full_time_pcnt*, *eg_colors*, *band_colors*, *job_strs*, *attr_dict*, *p_dict*, *ds_dict=None*, *rank_metric='cat_order'*, *mnum=None*, *attr1=None*, *oper1='>='*, *val1='0'*, *attr2=None*, *oper2='>='*, *val2='0'*, *attr3=None*, *oper3='>='*, *val3='0'*, *bg_alpha=0.12*, *fur_color=None*, *show_part_time_lvl=True*, *size=3*, *alpha=1*, *title_size=14*, *label_pad=110*, *label_size=13*, *tick_size=12*, *xsize=4*, *ysize=12*, *image_dir=None*, *image_format='png'*)

visually display employee group distribution concentration within accurately sized job bands for a selected month.

This chart reveals how evenly or unevenly the employee groups share the jobs available within each job category.

inputs

df (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

job_levels (integer) number of job levels in the data model

full_time_pcnt (float) percentage of each job level which is full time

eg_colors (list) list of colors for eg plots

band_colors (list) list of colors for background job band colors

job_strs (list) list of job strings for job description labels

attr_dict (dictionary) dataset column name description dictionary

p_dict (dictionary) eg to group string label

ds_dict (dictionary) output from load_datasets function

rank_metric (string) rank attribute (currently only accepts 'cat_order')

mnum (integer) month number - if not None, analyze data from this month

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

bg_alpha (float) color alpha for background job level color

fur_color (color code in rgba, hex, or string style) custom color to signify furloughed job band area (otherwise, last color from band_colors list will be used)

show_part_time_lvl (boolean) if True, draw a line within each job band representing the boundry between full and part-time jobs when using a basic jobs only data model (set this input to False when using an enhanced job data model)

size (integer or float) size of density markers

alpha (float) alpha of density markers (0.0 to 1.0)

title_size (integer or float) text size of chart title

label_size (integer or float) text size of x and y descriptive labels

tick_size (integer or float) text size of x and y tick labels

xsize, ysize (integer or float) width and height of chart in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**stripplot_eg_density**(*df*, *mnum*, *eg_colors*, *ds_dict=None*, *mnum_order=True*, *attr1=None*, *oper1='>='*, *val1=0*, *attr2=None*, *oper2='>='*, *val2=0*, *attr3=None*, *oper3='>='*, *val3=0*, *dot_size=3*, *chart_style='whitegrid'*, *bg_color='white'*, *title_size=12*, *suptitle_size=14*, *xsize=5*, *ysize=10*, *image_dir=None*, *image_format='png'*)

plot a stripplot showing density distribution for non-retired employees for each employee group separately at the selected month. The stripplot displays remaining employees positioned according to the selected month or initial month integrated list order (controlled by the "mnum_order" input).

Note: To analyze job category distribution density, use the "stripplot_dist_in_category" plotting function.

The input dataframe (df) may be a dictionary key (string) or a pandas dataframe.

The input dataframe may be filtered by attributes using the attr(x), oper(x), and val(x) inputs.

inputs

df (string or dataframe) text name of input proposal dataset, also will accept any dataframe variable (if a sliced dataframe subset is desired, for example) Example: input can be 'proposal1' (if that proposal exists, of course, or could be df[df.age > 50])

mnum (integer) view data for employees remaining (not yet retired) within this data model month number

eg_colors (list) color codes for plotting each employee group

ds_dict (dictionary) output from load_datasets function

mnum_order (boolean) if True, plot list position in month selected with the "mnum" input, otherwise plot according to initial integrated list position

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (integer, float, date as string, string (as appropriate)) attr(n) limiting value (combined with oper(n)) as string

dot_size (integer or float) size of stripplot markers

bg_color (color value) chart background color

title_size (integer or float) chart title text size

suptitle_size (integer or float) chart text size of suptitle

xsize, ysize (integer or float) size of chart width and height in inches

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

matplotlib_charting.**to_percent**(*decimal*, *position*, *precision=0*)

Custom format for matplotlib axis as a percentage.

Ignores the passed in position variable. This has the effect of scaling the default tick locations.

inputs

decimal (axis values) no user input

position ignored

precision (integer) number of decimals in output percentage labels

matplotlib_charting.**violinplot_by_eg**(*df*, *measure*, *ret_age*, *cdict*, *attr_dict*, *ds_dict=None*, *mnum=0*, *linewidth=1.5*, *attr1=None*, *oper1='>='*, *val1='0'*, *attr2=None*, *oper2='>='*, *val2='0'*, *attr3=None*, *oper3='>='*, *val3='0'*, *scale='count'*, *saturation=1.0*, *title_size=12*, *chart_style='darkgrid'*, *xsize=12*, *ysize=10*, *image_dir=None*, *image_format='png'*)

From the seaborn website: Draw a combination of boxplot and kernel density estimate.

A violin plot plays a similar role as a box and whisker plot. It shows the distribution of quantitative data across several levels of one (or more) categorical variables such that those distributions can be compared. Unlike a box plot, in which all of the plot components correspond to actual datapoints, the violin plot features a kernel density estimation of the underlying distribution.

inputs

df (dataframe) dataset to examine, may be a dataframe variable or a string key from the ds_dict dictionary object

measure (string) attribute to plot

ret_age (float) retirement age (example: 65.0)

cdict (dictionary) color dictionary for plotting palatte

attr_dict (dictionary) dataset column name description dictionary

ds_dict (dictionary) output from load_datasets function

mnum (integer) month number to analyze

linewidth (integer or float) width of line surrounding each violin plot

attr(n) (string) filter attribute or dataset column as string

oper(n) (string) operator (i.e. <, >, ==, etc.) for attr(n) as string

val(n) (string, integer, float, date as string as appropriate) attr(n) limiting value (combined with oper(n)) as string

scale (string) From the seaborn website: The method used to scale the width of each violin. If 'area', each violin will have the same area. If 'count', the width of the violins will be scaled by the number of observations in that bin. If 'width', each violin will have the same width.

saturation (float) Proportion of the original color saturation. Large patches often look better with slightly desaturated colors, but set this to 1.0 if you want the plot colors to perfectly match the input color spec.

title_size (integer or float) text size of chart title

image_dir (string) if not None, name of a directory in which to save an image of the chart output. If the directory does not exist, it will be created.

image_format (string) file extension string for a saved chart image if the image_dir input is not None

Examples:

'svg', 'png'

CHAPTER

FIFTEEN

REPORTS MODULE

reports.**annual_charts**(*ds_dict*, *adict*, *cdict*, *plot_year_group=True*, *plot_job_group=True*, *quantiles=10*, *plot_init_quarter=True*, *plot_running_quarter=True*, *pcnt_ylim=0.75*, *cpay_stride=500*, *fixed_col_name='eg_initQ'*, *running_col_name='eg_runQ'*, *figsize=None*, *date_grouper='ldate'*, *chartstyle='ticks'*, *verbose_status=True*, *tick_size=13*, *legend_size=14*, *label_size=14*, *title_size=14*, *adjust_chart_top=0.85*)

Generates multiple charts representing general annual attribute statistics of all calculated datasets for all employee groups FOR ALL ACTIVE EMPLOYEES (annual results for all employees).

The user may select grouping analysis by any or all of the following:

1. longevity or date of hire year
2. job level
3. initial employee group list quantile membership
4. annual employee group list quantile membership

Stores the output as images in multiple folders within the **reports/<case_name>/ann_charts** folder.

inputs

ds_dict (dictionary) output of load_datasets function, a dictionary of datasets

adict (dictionary) dataset column name description dictionary

cdict (dictionary) program colors dictionary

plot_year_group (boolean) if True, create chart images grouped by the date_grouper input year

date_grouper (string) column name representing a column of dates within a dataframe. Year membership of this column will be used for grouping. Input is limited to 'ldate' or 'doh'.

plot_job_group (boolean) if True, create chart images grouped by job level held by employees

quantiles (integer) the number of binning quantiles to measure for the initial and running (annually updated) quantile membership analysis (default is 10)

plot_init_quarter (boolean) if True, produce output grouped by initial list quantile membership, for each employee group

plot_running_quarter (boolean) if True, produce output grouped by annual list quantile membership, for each employee group

pcnt_ylim (float) output chart maximum y axis value for percentage attribute charts as a float, example: .75 equals max displayed chart value of 75%

cpay_stride (integer) y axis chart tick interval (in thousands) for charts displaying cpay (career pay)

fixed_col_name (string) label to use for quantile number column when calculating using the initial quantile membership for all results

running_col_name (string) label to use for quantile number column when calculating using a continuously updated quantile membership for all results

figsize (tuple) optional size of all generated chart images. Default is None. This input will allow creation of larger chart images than the default small charts, at the price of an increase in the time required to run the function.

date_grouper (string) 'ldate' or 'doh' date column grouping attribute used when plot_year_group input is True

chartstyle (string) any valid seaborn charting style ('ticks', 'dark', 'white', 'darkgrid', 'whitegrid'), defalut is 'ticks'

verbose_status (boolean) if True, print status of calculations as function is running

tick_size (integer or float) text size of tick labels on the output chart images

legend_size (integer or float) text size of the legend on the output chart images

label_size (integer or float) text size of the x and y axis labels on the output chart images

title_size (integer or float) text size of the title on the output chart images

adjust_chart_top (float) input to permit adjustment of the top location of the generated charts - used to ensure full chart title is captured by the save chart figure code. Defalt top position is 1.0, default vaule for this input is .85 which "shrinks" the

charts slightly vertically so that the two-line chart titles are captured when saving the charts to file as images.

reports.**job_diff_to_excel**(*base_ds, compare_ds, ds_dict, add_cpay=True, diff_color=True, row_color=True, lighten_factor=0.65, neg_color='red', pos_color='blue', zero_color='white', id_cols=['lname', 'ldate', 'retdate']*)

Generates a spreadsheet which reports the differential number of months spent at each job level between two outcome datasets. Results are reported for every employee.

The order of the employees shown will be the order from the "compare" dataset input.

The user may choose to apply formatting to the output spreadsheet. The generation of the output with formatting is much slower than without, however.

Stores the output within the **reports/<case_name>/by_employee** folder.

inputs

base_ds (dataframe) baseline dataset

compare_ds (dataframe) comparison dataset

add_cpay (boolean) if True, add a "cpay_diff" column to show data model pay differential (compare vs. base)

diff_color (boolean) if True, use the neg_color, pos_color, and zero_color inputs to color the spreadsheet job differential output

row_color (boolean) color spreadsheet rows by employee group if True. Color will be a tint (lighter color version) of the colors used to represent the employee groups in chart output.

lighten_factor (float) when the "row_color" input is True, this input controls the tint of the normal employee group colors to use for the cell background row coloring. The input is limited from 0.0 to 1.0 and a higher value will make the coloring lighter.

neg_color, pos_color, zero_color (color values) this input will determine the font colors to use for negative, positive, and zero job differential values within the spreadsheet output. Inputs may by string hex values, or rgb values within tuples or lists

id_cols (list) list of columns to include within the spreadsheet output which are in addition to the job level columns. This list (with the addition of the "order" column) will also be colored according to employee group when the "row_color" input is set to True.

reports.**retirement_charts**(*ds_dict*, *adict*, *cdict*, *plot_year_group=True*, *date_grouper='ldate'*, *plot_job_group=True*, *plot_init_quarter=True*, *plot_running_quarter=True*, *quantiles=10*, *pcnt_ylim=0.75*, *cpay_stride=500*, *fixed_col_name='eg_initQ'*, *running_col_name='eg_runQ'*, *figsize=None*, *chartstyle='ticks'*, *verbose_status=True*, *tick_size=13*, *legend_size=14*, *label_size=14*, *title_size=14*, *adjust_chart_top=0.85*)

Generates multiple charts representing general attribute statistics of all calculated datasets for all employee groups AT RETIREMENT ONLY.

The user may select grouping analysis by any or all of the following:

1. longevity or date of hire year
2. job level
3. initial employee group list quantile membership
4. annual employee group list quantile membership

Stores the output as images in multiple folders within the **reports/<case_name>/ret_charts** folder.

inputs

ds_dict (dictionary) output of load_datasets function, a dictionary of datasets

adict (dictionary) dataset column name description dictionary

cdict (dictionary) program colors dictionary

plot_year_group (boolean) if True, create chart images grouped by the date_grouper input year

date_grouper (string) column name representing a column of dates within a dataframe. Year membership of this column will be used for grouping. Input is limited to 'ldate' or 'doh'.

plot_job_group (boolean) if True, create chart images grouped by job level held by employees

quantiles (integer) the number of binning quantiles to measure for the initial and running (annually updated) quantile membership analysis (default is 10)

plot_init_quarter (boolean) if True, produce output grouped by initial list quantile membership, for each employee group

plot_running_quarter (boolean) if True, produce output grouped by annual list quantile membership, for each employee group

pcnt_ylim (float) output chart maximum y axis value for percentage attribute charts as a float, example: .75 equals max displayed chart value of 75%

cpay_stride (integer) y axis chart tick interval (in thousands) for charts displaying cpay (career pay)

fixed_col_name (string) label to use for quantile number column when calculating using the initial quantile membership for all results

running_col_name (string) label to use for quantile number column when calculating using a continuously updated quantile membership for all results

figsize (tuple) optional size of all generated chart images. Default is None. This input will allow creation of larger chart images than the default small charts, at the price of an increase in the time required to run the function.

date_grouper (string) 'ldate' or 'doh' date column grouping attribute used when plot_year_group input is True

chartstyle (string) any valid seaborn charting style ('ticks', 'dark', 'white', 'darkgrid', 'whitegrid'), defalut is 'ticks'

verbose_status (boolean) if True, print status of calculations as function is running

tick_size (integer or float) text size of tick labels on the output chart images

legend_size (integer or float) text size of the legend on the output chart images

label_size (integer or float) text size of the x and y axis labels on the output chart images

title_size (integer or float) text size of the title on the output chart images

adjust_chart_top (float) input to permit adjustment of the top location of the generated charts - used to ensure full chart title is captured by the save chart figure code. Defalt top position is 1.0, default vaule for this input is .85 which "shrinks" the charts slightly vertically so that the two-line chart titles are captured when saving the charts to file as images.

reports.**stats_to_excel**(*ds_dict*, *quantiles=10*, *date_grouper='ldate'*, *fixed_col_name='eg_initQ'*, *running_col_name='eg_runQ'*)

Create a set of basic statistics for each calculated dataset and write the results as spreadsheets within the **reports** folder.

There are 2 spreadsheets produced, one related to retirement data and the other related to annual data.annual

The retirement information is grouped by employees retiring in future years, further grouped for longevity or initial job.

The annual information is grouped by the model year, and further grouped by 10% quantiles, either by initial quantile membership or by an annual quantile adjustment of remaining

employees.

inputs

ds_dict (dictionary) output of load_datasets function, a dictionary of datasets

quantiles (integer) the number of binning quantiles to measure for the initial and running (annually updated) quantile membership analysis (default is 10)

date_grouper (string) column name representing a column of dates within a dataframe. Year membership of this column will be used for grouping. Input is limited to 'ldate' or 'doh'.

fixed_col_name (string) label to use for quantile number column when calculating using the initial quantile membership for all results

running_col_name (string) label to use for quantile number column when calculating using a continuously updated quantile membership for all results

CHAPTER

SIXTEEN

CHANGE LOG

16.1 version history

16.1.1 0.65

May 12th, 2020

This version updates seniority_list to be compatible with changes in some of the supporting Python data science packages which have reached 1.0 release status since this program was developed.

- update rquirements.txt
- modify the *update_stripplot* function within **editior_function.py** to restore correct display of the scatter density plot within the **EDITOR_TOOL.ipynb** notebook
- modify the layout parameters for the editor tool due to changes within bokeh
- slight change to the *align_next* function for changes within the numba package and rewrite docstring
- update *job_count_charts* plotting function to be compatible with how matplotlib handles empty groupby groups
- update *eg_attributes* plotting function to allow for new matplotlib datetime axis handling
- update *bk_basic_interactive* interactive plotting function due to different way that widgets are defined with bokeh since 1.0

16.1.2 0.64

March 1st, 2020

Minor update to fix broken links in documentation.

16.1.3 0.63

October 8th, 2018

This version adds functionality within many of the scripts and plotting functions, updates the plotting functions for compatibility with matplotlib 3.0, adjusts the editor tool code for compatibility with the bokeh plotting library, and corrects a few bugs.

Script and non-plotting functions updates:

- modify **build_program_files.py** script to allow edited list order from *proposals.xlsx* to be constructed properly with a "new_order" column vs an "idx" column
- modify **compute_measures.py** script to accept edited proposal orderings from *proposals.xlsx*
- update **reports.py** script functions *retirement_charts* and *annual_charts* to be compatible with matplotlib 2.2 (this prevents the previous behavior of automatic plotting of the final calculated charts within jupyter notebook)
- corrected bug in **build_program_files.py** script when using basic jobs (non-enhanced)
- update comment cells in **RUN_SCRIPTS.ipynb** notebook
- update the *anon_master* and *anon_pay_table* functions (in the functions module) to use the "sheet_name" keyword parameter with pandas read_excel functions. this is due to a revision within pandas
- add docstring to *hex_dict* function
- remove ipywidgets from program requirements

Plotting function updates:

- improve *quantile_groupby* plotting function. Now two datasets may be compared for the same employee group. Update **STATIC_PLOTTING.ipynb** notebook with correct variable inputs and new plotting example. Add chart example to documentation gallery.
- update *stripplot_eg_density* plotting function (removed "attr_dict" input and improved chart title labels)

- update *quantile_groupby* plotting function (add "verbose_title option, add "plot_total" option, correct bug when "through_date" input was greater than maximum data model date)
- update *job_transfer* plotting function so that title shows verbose employee group name instead of an employee group code number
- update *stripplot_eg_density* plotting function to permit display of list order relative to selected month order or initial integrated list order and also improve the chart labels
- enhance title display in the *quantile_years_in_position* plotting function
- add code to handle situation when filtering results are an empty dataset in the *differential_scatter* plotting function
- update *quantile_groupby* plotting function to include auto-yscale tick spacing when "cat_order" is selected for the "measure" input. This prevents the plotting library from picking random tick spacing.

Editor tool updates:

- update animate function callbacks within the *editor_function* to align with change in bokeh api version 0.12.16+
- adjust height of bokeh textinput widget within the **editor_function.py** module to less than optimal height to maintain usability. The bokeh textinput widget is missing functionality for proper sizing. When the functionality is implemented, the txt_height variable will be readjusted.
- remove the global variable from callbacks within the *editor_function* and the *bk_basic_interactive* function and replace with a new class object
- update editor tool layout spacers
- refactor editor tool periodic_callback code for compatibiltiy with bokeh update

16.1.4 0.62

April 18th, 2018

This a minor update with changes for compatibility with matplotlib 2.2 and minor code tweaks to allow a wider range of user scenarios.

- change references to "Vega20c" matplotlib colormap to "tab20c"
- change matplotlib tick parameters from "on" and "off" to "True" and "False"
- add ax.margins(x=0) to plotting code where needed
- update **build_program_files.py** to allow cases without any furloughed employees

- update *contract_year_and_raise* function to allow compenstation data without any pay exceptions
- update *distribute* function
- update *group_average_and_median* plotting function to permit proper plotting when default job level scaling interval is less than one

16.1.5 0.61

February 26th, 2018

This update refactored the job assignment routine used when a ratio condition is applied, added a time in job differential study to the **reports** module, and applied miscellaneous code and docstring cleanup.

Users may elect to capture an existing job distribution ratio (between the employee groups) to be applied during the effective condition time period for both capped and unrestricted ratio job assignment. The input spreadsheet *settings.xlsx* "ratio cond" and "ratio_cond_capped_count" worksheets now contain an additional column ("snapshot") for selecting this option. The "excel input files" section of the documentation has been updated. Code changes related to the new ratio job assignment routine:

- update *set_snapshot_weights* function
- update *assign_cond_ratio* function
- update *distribute* function
- remove *assign_cond_ratio_capped* function
- add *eg_quotas* function
- update **build_program_files** script
- update **converter** script
- refactor *remove_zero_groups* function

This version adds to the built-in reporting capability of seniority_list with the new *job_diff_to_excel* function. The function will calculate the time difference (in months) each employee would spend in each job level between data models. The results are presented as a formatted spreadsheet stored within the **reports** folder. The *hex_dict* function was added to support the formatting requirements for the spreadsheet output.

The NumPy "in1d" function has been replaced with the NumPy "isin" function throughout as recommended by NumPy[124].

Hard code used during development was removed/updated within the *violinplot_by_eg* and the *eg_multiplot_with_cat_order* functions.

[124] https://docs.scipy.org/doc/numpy/reference/generated/numpy.in1d.html

Some formatting of function docstrings was updated to improve the output format of the web and pdf documentation.

16.1.6 0.60

January 12th, 2018

The documentation has now been updated for the new editor tool and the old version of the editor has been removed.

A new **interactive_plotting.py** module has been added to the program, along with a companion **INTERACTIVE_PLOTTING.ipynb** notebook file. Only one interactive chart is included at this point.

Revision highlights include:

- the editor zone delineation for each chart area has been changed from a bokeh rect glyph to a box annotation. The vertical spread of the zone will now alway extend to the limits of the chart areas
- a correction was made to the edit zone cursor line conversion calculation when using a "running" xtype x axis
- the "proposal" dropdown selection on the "proposal_save" panel will now automatically change to "edit" when a squeeze is performed
- added styling control for the edit zone
- added code to handle data model months with no data when extra filters have been applied
- renamed the **PLOTTING** notebook to **STATIC_PLOTTING** to accomodate the new **INTERACTIVE_PLOTTING** notebook

16.1.7 0.59

December 23rd, 2017

The editor tool has been completely rewritten and is now implemented as a local web server application within the notebook using the Bokeh plotting library. This first release version is now included with the program but is not yet supported with documentation. A revised user guide will be forthcoming soon. The documentation related to the editor tool will be incrementally revised over the next several weeks. Much of the current documention can be applied to the new tool.

Other improvements with this revision include:

- updated assign_standalone_job_changes function

- fixed old editor tool display functionality following ipywidgets update, though performance when using the cursor sliders is less than ideal
- changed all pandas "read_excel" parameters from "sheetname" to "sheet_name" for compatibility with future versions of pandas
- added editor_dict to the build_program_files.py script which provides initial values for the new editor tool display and will store editor tool values during and between sessions
- added convert_to_hex function which converts rgba values (such as those produced by the make_color_list function) to string hex color values
- added the find_nearest and cross_val functions for use with the editor tool p1 and p2 cursor equivalent position feature (p1 and p2 are the bokeh chart figures)

16.1.8 0.58

September 25th, 2017

This update includes coding updates which improve the computational efficiency of the program, resulting in a 10-15% reduction in the time required to compute a dataset.

- General changes were made through entire code base to increase computational speed wherever possible:
  - numpy.arange() to range()
  - numpy.sum(<condition>) to numpy.count_nonzero(<condition>)
  - numpy.array(dataframe_column) to dataframe_column.values
  - max(array) to array.max()
- Applied fast **numba** jit (just in time compiling) to the following refactored functions:
  - *align_next*
  - *mark_fur_range*
- Replaced standard numpy expressions used for job counting and job count column assignment with two new **numba-optimized** functions:
  - *count_avail_jobs*
  - *assign_job_counts**
- Improved the performance of the following functions through the use of line profiling and refactoring:
  - *career_months*

  - *convert_to_datetime*
  - *count_per_month*
  - *gen_skel_emp_idx*
  - *age_correction*

- Updated the **standalone.py** script to use the *create_snum_and_spcnt_arrays* function for faster generation of the snum, spcnt, lnum, and lscpnt columns.

Other improvements were made to the program which are not related to reducing computation time:

- Added the *find_squeeze_vals* function and incorporated it within the **editor tool**. The new function permits editor squeezing (a visual exercise based on displayed data) when future month data is displayed to the user. Future month cursor line postion is converted to the equivalent original list positions for use within the squeeze algorithm.
- Added an experimental section to the *job_time_change* plotting function. The **PLOTTING** notebook was updated accordingly.
- Removed the no longer used "orig" output from *assign_jobs_nbnf_job_changes* function.
- Changed code reference from "qtr" to the semantically correct "qntl" for use within the summary **reports** charts output.
- Restored "full_flush" job assignement functionality with updates to the *assign_jobs_full_flush_job_changes* function.
- Added a sort routine to the *eg_count* settings dictionary value creation routine within the **build_program_files** script to ensure continuity with other program calculations.
- Removed functions which have been superceded and are no longer used:
  - *snum_and_spcnt*
  - *create_snum_array*

16.1.9 0.57

August 23rd, 2017

This update includes a major editor tool upgrade.

- added editor tool absolute value display

  Previously, only a differential comparison of attribute values between a baseline and comparative dataset was possible. Now the actual values,

initially from the comparative and then the edited dataset (after the first edit), may be displayed. This option allows the user to directly analyze the distribution of equity and opportunity within the merged operation of integration proposals.

- added editor tool additional display filtering

  The user may now show only results for targeted subsets of the merged population, allowing rapid analysis of certain list attribute cohorts. For example, this feature permits additional outcome evaluation for employees who may have limited years remaining in their careers or employees belonging to a special job assignment category.

- extensive updates to the editor tool documentation and the editor tool function docstring
- updeated EDITOR_TOOL notebook to incorporate the new editor tool functionality
- added *find_index_val* function to **functions** module
- improved excel input file documentation
  - added sections on job level hierarchy and the "hours" worksheet preparation, both within the "pay_tables.xlsx format guide".

16.1.10 0.56

June 21st, 2017

- editor tool stylistic update
  - replaced the independent "junior" and "senior" slider controls with a single, easier to use range selector slider tool
  - increased the width of the sliders for easy value selection
  - applied a "flex" sizing method to the controls which allows the tool to auto-adjust the width of the controls to match the available screen size
  - various other styling added

16.1.11 0.55

May 23rd, 2017

- new dataset **reports** capability

  This update includes a new *reports* module. General statistics may be generated quickly for all calculated datasets, providing a broad overview of how each proposed integrated list will affect employees from each work group. This process provides useful absolute and comparative information for targeted attributes. The statistics are converted to excel spreadsheets and chart images, stored within the **reports** folder.

  Data is produced for the targeted metrics both at retirement and on an annual basis.

  The charts are smaller and of lower image quality than the charts produced by the dedicated plotting functions included with seniority_list. This is done to reduce the time required to generate the hundreds of charts in the output. If the user desires better quality charts for the general overview charts, a larger chart size may be designated through a function input.

- added a new "quick report" section to the documentation covering the new reporting capability
- added a new example **REPORTS** notebook to the program. This notebook provides code examples for the new reporting capability and will generate summary spreadsheets and chart images for the current case study when it is executed.
- updated the ds_dictionary creation routine - output is now dataset name/dataset key-value pair vs. the previous dataset name/(dataset, dataset name) tuple dictionary values.

16.1.12 0.54

May 13th, 2017

- combined *career_months_df_in* and *career_months_list_in* into one function, *career_months*
- add *convert_to_datetime* function
- add *pcnt_format* function and update plotting code to incorporate the change
- improve code relating to saving chart images
- consolidate "imp_date" and "implementation_date" references

- update the code that groups data according to "empkey" attribute due to a version change in the pandas library
- update pandas "parallel coordinates" import due to a version change in the pandas library
- add the *eg_attributes* plotting function. This function replaces the *multiline_plot_by_eg* plotting function. This new function is able to plot any attribute (including date attributes) on either the x or y axis and introduces quantile membership lines and bands.
- remove *multiline_plot_by_eg* plotting function and *eval_strings* function
- docstring updates throughout

16.1.13 0.53

April 30th, 2017

Improvements with charts, plotting:

- updated the *multiline_plot_by_eg*, *multiline_plot_by_emp*, *job_level_progression*, and *quantile_years_in_position* plotting functions
- numerous updates and improvements to chart styling control for many plotting functions

Expanded pay exception capability:

- refactor contract_year_and_raise fuction to permit any number of pay exception periods
- add new "pay_exceptions" worksheet in the *settings.xlsx* input file
- update **make_skeleton.py** script to use the new pay_exceptions method

New anonymizing functions:

- added capability to anonymize input data with the following new functions:
  - *anon_names*
  - *anon_empkeys*
  - *anon_dates*
  - *anon_pay*

Each of the above functions generates random substitute data for the related input data column. These "helper" functions were combined into the following functions which can anonymize the *master.xlsx* and *pay_tables.xlsx* files all at once, inplace.

- *anon_master*

- *anon_pay_table*

New sampling ability:

- added the *sample_dataframe* function, which returns a random sample of a dataframe (by rows), with the quantity of rows selected by the user

New excel-related functions:

- *update_excel*
- *copy_excel_file*

pdf documentation

- added downloadable pdf version of the program documentation
- formatted function definitions for proper presentation within the pdf document

Program coding improvement:

- added "if __name__ == "__main__":" execution protection to all scripts

There were some older developemental files and references to settings remaining within the code base that were not needed any longer.

- removed several developemental functions
- remove several items from the settings dictionary
- remove several rows from the "scalars" worksheet within the *settings.xlsx* file

16.1.14 0.52

April 19th, 2017

This update version focused on updating the visualization capabilities of seniority_list.

- refactor *job_transfer* plotting function for speed and added features
  - updated function is approximately 25 times faster
  - added ability to plot only targeted job level(s)
  - new y scale limit option
  - new min and max date options
- add new *percent_bins* function and corresponding *percent_diff_bins* plotting function
  - plots count of employees in list percentile change bins over time
- add new *cohort_differential* plotting function

  - analyze differences between list locations for employees with equivalent attribute values but from different groups

- add code to all notebooks for an automatic wide display
- update *multiline_plot_by_emp* plotting function to permit simultaneous display of "jnum" and "jobp" attributes
- update *multiline_plot_by_eg* plotting function to permit plotting of values at retirement for all employees
- add ability to plot individual employee progression lines with *job_count_bands* plotting function
- update all plotting function code to matplotlib object-oriented style
- update many plotting function chart legend generation routines
- add capability to save charts as images (including SVG format)
- update **PLOTTING** notebook to incorporate new plotting functions/features
- update documentation

16.1.15 0.51

April 1st, 2017

- remove "example_chart" option from plotting functions
- add exception types to most try/except blocks throughout program
- remove "master_name" argument from **join_inactives.py** script
- update **join_inactives.py** script to permit input from editor tool output list order
- update *assign_jobs_nbnf_job_changes* function:
  - reduce the number of arguments for the main integrated job assignment function
  - add job table dictionary to the function arguments
  - eliminate the "this_job_col" variable within monthly loop
- reduce and simplify the arguments for the *assign_standalone_job_changes* function, and use settings dictionary and job table dictionary as arguments
- add the *add_zero_col* function to the *functions* module. This function will add a column of zeros as the first column of a 2D numpy array
- move the code to generate the *dict_job_tables.pkl* dictionary file from the **make_skeleton.py** script to the **build_program_files.py** script for consistency with other generated files

- add a section within the **build_program_files.py** script to create a *loop_check* array. This boolean array will prevent unnecessary looping during the job assignment routine when all remaining employees have already been assigned. Reduces "Sample3" dataset generation times by approximately 5%.
- update **RUN_SCRIPTS** and **PLOTTING** notebooks
- update documentation

16.1.16 0.50

March 20th, 2017

This update improved the flexibility of the ratio-based conditional job assignment routines. Inputs for these routines are now designated on individual worksheets within the *settings.xlsx* input file. Conditions may include any combination of jobs, weightings, and employee groupings.

- refactor **build_program_files.py** script:
  - change ranges relating to month time spans to sets vs ranges
  - remove references to condition durations, month ranges as sets have replaced these inputs
  - add new dictionary generation routine used with input from the *ratio_cond* worksheet in *settings.xlsx*.
  - remove code related to *count_cond*, *ratio_cond*, and *quota_dict*.
- update **converter.py** to handle the basic to enhanced conversion of new ratio-condition related dictionaries and remove code no longer needed.
- eliminate many arguments for the *assign_jobs_nbnf_job_changes* function and replace with a settings dictionary argument.
- refactor variable preparation sections within the *assign_jobs_nbnf_job_changes* function for use with the new dictionaries and month sets loaded from the settings dictionary when ratio-based conditions are selected.
- refactor the *assign_cond_ratio_capped* and *assign_cond_ratio* job assignment functions. The new functions are simpler and more flexible in terms of inputs. Both functions accept a new dictionary argument, built from input worksheets which have been reformatted.
- refactor the *set_ratio_cond_dict* function and rename it as *set_snapshot_weights*. The function modifies the weightings within the *ratio_dict* dictionary for all jobs at once to match existing job counts for a target month.

- add a "cap" argument to the *distribute* function. The cap argument allows the function to be used within a ratio count-capped conditional job assignment routine.
- modify the *distribute_vacancies_by_weights* function for simplicity and precision. This function is no longer used and may be removed at a future date.
- the *quota_dict* and *count_ratio_condition* worksheets were removed from the *settings.xlsx* input file. These worksheets were replaced with the new *ratio_count_capped_cond* worksheet.
- the format of the *ratio_cond* worksheet in *settings.xlsx* was updated for use with the new *assign_cond_ratio* function.

The job table generation has now been centralized within the **make_skeleton.py** script. The job tables are now stored as a dictionary within the **dill** folder permitting one-time calculation and universal program access.

- add create job tables routine to **make_skeleton.py** and store tables as a dictionary, *dill/dict_job_tables.pkl*. Additionally, the *j_changes* and *jcnts_arr* variables are stored within the dictionary.
- remove job table generation routines from individual plotting functions within the **matplotlib_charting.py** script, the **standalone.py**script, and the **compute_measures.py** script. Replace all by reading the stored *dill/dict_job_tables.pkl* dictionary.

Finally, a new utiliy function was added which prints the contents of dictionaries in an organized, landscape fashion.

- add *pprint_dict* function to the *matplotlib_charting* module.

16.1.17 0.49

March 9th, 2017

- Change documentation references from configuration file to settings dictionary.
- Remove **make_pay_tables_from_excel.py** script. This script is now incorporated within the **build_program_files.py** script
- Change references throughout code from *eg_dict* to renamed *p_dict*.
- Create the **dill** folder with the **build_program_files.py** script if it does not exist. An empty **dill** folder is no longer part of the original program files.
- Modify *clear_dill_files* function to check for the existence of the **dill** folder before executing.
- Add proposal name argument test and exception messages to **compute_measures.py** and **join_inactives.py** scripts.

- Add *add_editor_list_to_excel* function to *matplotlib_charting* module. This function will add an edited proposal list order (output of editor tool) to the *proposals.xlsx* input file, as a new worksheet named *edit*. The edited proposal list order may be preserved in this fashion and permits an easy way to reproduce the corresponding dataset.
- Add code to remove stored pickle files prior to overwriting for a speed improvement.
- Add a return_min option to the *max_of_nested_lists* function.
- Extensive updates to the *matplotlib_charting* and the *function* modules doctrings defining input types and function descriptions.
- Refactored *cond_test* plotting function for improved capability and output.
- Add *count_ratio_dict* worksheet to *settings.xlsx* input file. This worksheet will eventually replace the *count_ratio_condition* and the *quota_dict* worksheets as the count ratio condition code is updated.
- Add code to the **build_program_files.py** script to read the new *count_ratio_dict* worksheet.
- Add code to the *convert* function within the *converter* module to convert the data from the *count_ratio_dict* for an enhanced job level model when appropriate.
- Delete function *make_intlists_from_columns*.
- Modify function *make_lists_from_columns* to handle deleted function above.
- Add *make_group_lists* function. This function is used with Excel input (specifically worksheet cells) to convert string objects (ex. "2,3") and integers into Python lists containing integers. This function is used with the count_ratio_dict dictionary construction.
- Add *make_eg_pcnt_column* function. Create an array of values which may be added to the input dataframe as a column reflecting the starting percentage of each employee within his/her original employee group at month zero.
- Add *make_starting_val_column* function. Create an array of values which may be added to the input dataframe as a column reflecting the starting value (month zero) of a selected attribute for each employee for every month (repeating values for successive months, indexed and unchanging for each employee).
- Add *save_and_load_dill_folder* function. Save the current **dill** folder to the **saved_dill_folders** folder (created if it does not already exist). Load a saved dill folder as the **dill** folder if it exists. This function allows previously calculated pickle files (including the datasets) to be loaded into the **dill** folder for quick review. All adds up to mean convenient switching between previously calculated case study files.

16.1.18 0.48

February 6th, 2017

This version is a **major** update. All inputs for the program are now read solely from spreadsheet workbooks - the configuration files have been completely eliminated. This change was made to make it easier for non-programmers to interact with seniority_list and to generally simplify the work flow when setting up the program for a particular case study and for further parameter modifications in the course of analysis. The new workbook containing the information previously held within the config files is named *settings.xlsx* and is located within the **excel** folder.

The data from the new *settings.xlsx* spreadsheet is stored in three dictionaries which serve as a fast data source for operations.

- Settings dictionary - essentially contains all of the information previously located in the configuration files.
- Color dictionary - a new source of color lists for plotting.
- Attribute dictionary - a collection of dataset column name descriptions used for plotting titles and labels.

The dictionary generation process has been incorporated within the **build_program_files** script, adding to the other generated data files and compensation table data. The dictionaries are stored in the **dill** folder as separate files.

When beginning a new case study, the user will now simply create a new case study folder within the **excel** folder and paste copies of the sample workbooks into it. The user will then go through each spreadsheet and modify the contents as appropriate to the new case study.

The old **case_files** folder and its contents are no longer used or needed. The old config.py file in the main **seniority_list** folder has been eliminated as well.

An added bonus of this update is the availability of a wide-range of chart plotting color schemes. The new color dictionary is created with multiple color lists as values and matplotlib colormap names as keys. All matplotlib colormaps are now available at all times. Each color list is automatically generated with a length equal to the number of job levels in the data model + 1. This supplies a color for each job level plus an additional color for a furlough level.

All scripts and functions were updated to utilize the new dictionaries with many functions receiving additional arguments and additional docstring descriptions for even more control and customization of analysis output.

Four new functions were developed to assist with the spreadsheet to python conversion.

- *make_tuples_from_columns*
- *make_dict_from_columns*
- *make_intlists_from_columns*
- *make_lists_from_columns*

These functions are essentially "helper" functions used within the **build_program_files** script and are contained within the *functions* module.

Two new plotting-related functions were built as well.

- *make_color_list*
- *add_pad*

The *make_color_list* function is able to perform multiple tasks, from producing a custom color list to plotting an example of every matplotlib colormap. It is used within the **build_program_files** script to produce the color dictionary.

The *add_pad* function automatically spaces chart labels when they would otherwise overlap one another. It has been incorporated within several plotting functions.

The new plotting functions are located within the *matplotlib_charting* module.

16.1.19 0.47

January 15th, 2016

- added a metric (attribute) description dictionary, "m_dict", to general configuration file. This dictionary will provide labels for many of the plotting functions.
- refactored the delayed implementation methodology to use standalone data stored within a numpy array, generated by a new function, *make_preimp_array*. The new method allows any pre-implementation attributes to be transferred to the integrated dataset and is simpler than earlier code.
- refactored the "cat_order" attribute generation by employing a new function, *make_cat_order*. The new function is faster than the old method and correctly restricts standalone results to available standalone job levels.
- removed enhanced job level conditional variable assignment from case-specific configuration files and replaced with the new *convert* function. The new function is contained within a new module, *converter.py*, which is imported by the case-specific file(s). Only basic job level conditional job assignment data will be entered into the case-specific configuration files now. The basic level data will be automatically converted to enhanced data as appropriate.

16.1.20 0.46

December 31st, 2016

- added *slice_ds_by_index_array* function to *matplotlib_charting* module and example to the PLOTTING notebook (subsequently renamed to *slice_ds_by_filtered_index*).
  - filter an entire dataframe by only selecting rows which match

  the filtered results from a target month. In other words, zero in on a slice of data from a particular month, such as employees holding a specific job in month 25. Then, using the index of those results, find only those employees within the entire dataset as an input for further analysis within the program.
  - The output may be used as an input to a plotting function or for other analysis. This function may also be used repeatedly with various filters, using output of one execution as input for another execution.
- improved the *make_decile_bands* function and docstring.
- updated **case_template.py** file variable names for simplicity.
- refactored some hard-coding found within the pre-existing condition section within the **compute_measures.py** script. This change will prepare any employee group(s) for special rights calculations.
- added numerous function docstring improvements, primarily input variable descriptions.
- refactored *gen_skel_emp_idx* function so that it now generates a long-form employee index array in addition to the idx_array. The **make_skeleton.py** script was updated to use this new output.
- refactored the *align_fill_down* function, removing one input.
- added numerous comments in many of the program files.
- combined the *convert_jcnts_to_enhanced* and *convert_job_changes_to_enhanced* functions into one new function, *convert_to_enhanced*. The **list_builder.py** script was updated to use the new function, along with some plotting functions.
- refactored *cond_test* plotting function, allowing much more flexible job assignment validation.
- added *mark_quantiles* plotting function. This function is used by the *quantile_groupby* function below.
- added *quantile_groupby* plotting function.

  - This function permits the user to group the members of a selected employees group(s) into equally-sized sections, or quantiles, and track the attributes of those groups over time using various groupby methods. The available methods are as follows (default is median):

    [mean, median, first, last, min, max]

  - For example, an input of 40 for the quantiles input would equate to 40 sections of the initial employee group population, each representing 2.5% of the group. The progression of these group segments will be calculated and plotted, maintaining the original members of each segment. quantile calculation from separate groups is independent of each other, but can be tracked through an integrated dataset for robust comparison of outcome.

  - If the user selects "cat_order" (job category numerical ranking), color bands representing the various job levels may be displayed as a chart background. This provides the user with a clear visualization of the way the employee group would progress through the various job levels over time under various list ordering proposals.

  - Examples of the *quantile_groupby* plotting function have been added to the PLOTTING notebook.

- extensive narrative, definitions, and examples have been added to the "user guide" section of the documentation.

16.1.21 0.45

November 27th, 2016

- upgraded the *editor* tool function.
  - The editor tool will now automatically use the edited dataset for the recursive editing routine. The initial "compare_ds_text" dataset reference will now only have effect when an edited dataset does not exist.
  - The process may be interrupted and reset with a new "reset" argument.
  - The function will default to the first dataset proposal if the "compare_ds_text" input is invalid.
  - The title of the differential chart will now reference the dataset being compared to the baseline.
- many minor code improvements.
- continual work on the program documentation, particularly the operational overview and the user guide.

16.1.22 0.44

October 20th, 2016

- Refactored *make_pay_tables_from_excel.py* script.
  - The requirements for the input Excel workbook related to compensation have been greatly simplified. Only two worksheets are necessary, one containing basic job level hourly rates and another with monthly pay hours per level and job description labels.
  - Enhanced job tables are now automatically prepared when appropriate. This is controlled by the *config.py* enhanced_jobs variable.
  - Furlough job levels are now added automatically as the bottom level within each annual grouping of pay data.
  - Total monthly compensation tables may be ordered by a select pay year and longevity level.
  - The script now creates a new Excel-format file with worksheets containing the calculated pay tables utilized for the case study, *pay_table_data.xlsx*. Sorted pay tables may be examined and the sort basis changed if desired. The workbook also contains other worksheets pertaining to the job level order used within the model. The file is stored in an auto-generated, case-study named folder within the "reports" folder.
  - Other features added as described within the user guide.
- The *join_inactives.py* script now stores its Excel file output within the "reports" folder, next to the pay data file mentioned above.

16.1.23 0.43

October 5th, 2016

- Added the *job_count_bands* plotting function to the library of built-in plotting functions included with seniority_list. This function returns a chart which displays progressive counts of job opportunities available to selected employee group(s) under selected list order proposal(s) as an area chart with bands of different colors representing job levels. The input data may be filtered by up to three attributes, so that analysis may target particular population segments, as described in the previous version summary.
- Continued work developing the "user guide" section of the documentation.

16.1.24 0.42

September 28th, 2016

- This version includes a major update to the plotting functions and changes the way datasets are loaded for analysis.
- Most built-in plotting functions now have a three-layer filtering capability. This permits simple drill-down into the dataset for further insight. (Note: This is an added user-friendly convenience feature only. The capability to pre-filter datasets existed prior to this update but required additional programming knowledge to use it.) Analysis of specific subsets of the datasets is now straight-forward and much more convenient. For example, a target filtered attribute dataset with employees above a certain age, with a minimum longevity value, who are holding a certain job would be trivial to select with this new capability. For most plotting functions, a filtered subset could be viewed for a particular model month as well. This added filtering capability is handled with the new *filter_ds* function. The *filter_ds* function checks for attribute filtering arguments and uses them to filter the datasets prior to analysis within the various plotting functions.
- The way that pickeled datasets are read for use in the program has been updated. The names of the case study proposal worksheets are read from the source Excel workbook (proposals.xlsx). The program then looks for the matching datasets within the dill folder and loads them into a dictionary, using the proposal names as keys. Labels associated with the datasets are generated at the same time. These labels are used in the plotting functions. This functionality is provided with the new *load_datasets* function.
- Another new function allows flexible dataset variable input for for nearly all of the plotting functions. The *determine_dataset* function allows inputs to be a string key referenced to the dictionary output of the *load_datasets* function, or any variable representing a pandas dataframe.
- All program files and notebooks were updated to handle the new methods described above and the plotting functions documentation was revised.

16.1.25 0.41

September 3rd, 2016

- Removed "fur.pkl", "sg.pkl", and "active_each_month.pkl" file generation from the **build_program_files.py** script. These files were no longer needed.
- Consolidated the two standalone dataset scripts into one. This eliminated the *standalone_with_job_changes.py* and *standalone_no_job_changes.py* scripts in favor of the new script, *standalone.py*.

- Refactored *config.py* to create a job change schedule reflecting no job changes when the compute_with_job_changes option is False. This allows the job changes routine to run with all dataset calculations, adding simplicity and eliminating unnecessary code.
- Updated the pay_tables.xlsx Excel file by removing worksheets which are no longer needed.
- Added a quota_dict section to the basic job level configuration section of *sample3.py* and the *case_template.py* files.
- Removed the "actives_only" option in the *config.py* file. All datasets will now include any furloughed employees and will not incorporate other inactive employees.
- Modifications made with other program files to accommodate the removal of the "actives_only" option.

16.1.26 0.40

August 31st, 2016

- Add *clear_dill_files* function, used by "auto-cleaning" below.
- Add "auto-cleaning" of dill folder when case_study config input is changed. This prevents residual files from a previous study coexisting with new case study files within the "dill" folder.
- Add auto-generated sample employee and employee list to PLOTTING notebook. This will pick median employees from any list(s) for use with sample plotting.
- Moved one-time editor tool ipywidget config command to last cell in EDITOR_TOOL notebook. A recent update to ipywidgets required this command to be run one time. The user will uncomment the code, run the cell, then recomment the code.
- Updated case_template.py file to match recent upgrades.
- Add documentation for website user guide relating to input file naming conventions and file locations.

16.1.27 0.39

August 30th, 2016

- Simplified structure of *config.py* module. Users will have a much clearer understanding of modifiable vs. imported variables from the case-specific config file. Sections designed for user-modifiable inputs are now clearly delineated.
- Added pay table related configuration file inputs which will be imported from the case-specific config file including option for a future raise and/or temporary pay scale exceptions.
- Modified *contract_pay_year_and_raise* function to accept the customized pay-related inputs from config file.
- Added the plotting function *eg_boxplot*. This function will plot actual attribute ranges for employee groups over time as boxplots.

16.1.28 0.38

August 26th, 2016

- This update is a collection of minor edits, docstring additions, notebook adjustments, and refactoring.
- EDITOR notebook... recent update to ipywidgets requires a one line configuration command for proper operation. Added a cell within the notebook to accomplish that requirement. (Note: the update broke the button colors)
- PLOTTING notebook... adjusted variable inputs within notebook cells to match minor configuration file color list changes
- Updated chart labeling for the *group_average_and_median* function
- Improved *rows_of_color* plotting function, users may now select any job level combined with any employee group(s) for any month, also can display other categories such as furlough or other special group
- Added documentation for several plotting functions
- Removed standalone or furlough colors from case-specific configuration color lists. Now these additional colors are added when needed from within a function.

16.1.29 0.37

August 12th, 2016

- Adjusted *editor* function widget positioning, and minor code adjustment to permit compatibility with anaconda ipywidgets version (which is lagging behind latest version significantly, though the editor retains full functionality).
- Added *group_average_and_median* plotting function. This function permits plotting of group average and/or median for a selected attribute over time for a main and secondary dataset. Standalone data may be used as main or secondary data. The attributes may be further filtered/sliced by up to 3 constraints, such as age, longevity, or job level. This function can plot basic data such as average list percentage or could, for example, plot the average job category rank for employees hired prior to a certain date who are over or under a certain age, for a selected integrated dataset and/or standalone data (or for two integrated datasets).

16.1.30 0.36

August 9th, 2016

- *job_time_change* function may now display job numbers or custom job labels (from case-specific config file).
- Added **EDITOR_TOOL.ipynb** notebook to repository.
- Eliminated need for "edit_mode" input within general configuration file. The program will now use edit mode whenever the editor tool is used.

16.1.31 0.35

August 4th, 2016

- Refactored *parallel* plotting function to handle any number of datasets and any number of employee groups.
- Added new plotting function *job_time_change*. This function compares the amount time in months spent in various jobs under different list proposals. The information is presented only for employees who experience a change. Any number of datasets, employee groups, and job levels may be selected for analysis.
- Added documentation for multiple plotting functions

16.1.32 0.34

July 31st, 2016

- Added cat_order attribute (job rank number) to standalone dataset. The cat_order for each independent group is normalized to be accurate for the integrated group. This allows direct comparison with integrated job levels.
- Refactored **compute_measures** script so that standalone cat_order data is merged with integrated cat_order data when a delayed implementation exists. This new capability can be visualized for individual employees with the *job_level_progression* plotting function.

16.1.33 0.33

July 26th, 2016

- Refactored *eg_diff_boxplot* function to allow any number of datasets to be compared with standalone data or with each other. Employee groups for analysis may now be selected and the plot colors will be correct for the group(s). Option added to exclude employees who will be furloughed at any point within the data model, reducing or eliminating outlier data for some attribute measures.

16.1.34 0.32

July 17th, 2016

- Added additional filtering capability to the editor tool. Filtering may now be accomplished with monthly data combined with with additional attribute selection.
- Added *reset_editor* function to restore the editor if invalid filter attributes are selected, leading to an exception. Exception handling will be added as my available developer time permits.

16.1.35 0.31

July 16th, 2016

- Added option to increase employee retirement age. The retirement age may be raised with specified increments at designated times in the future.
- New function *clip_ret_ages* sets proper retirement age for employees in their retirement month when the model includes a retirement age increase.
- Added a "ret_mark" column during the skeleton file creation routine which is passed to the calculated datasets. The ret_mark column will indicate "1" when

an employee is in their last working month. This is helpful for filtering or plotting retirement data when the datasets contain multiple retirement ages.

16.1.36 0.30

July 10th, 2016

- Altered standalone dataset generation scripts to accept any number of employee groups.
- Modified *differential scatter* plotting function to accept any number of proposals and employee groups
- Replaced numpy "unique" function with pandas "unique" function throughout the code for speed improvement

16.1.37 0.29

July 7th, 2016

- Changed "cat_order" attribute calculation method to a groupby operation vs. a sort and resort yielding **10-15** percent reduction in total dataset compilation time.

16.1.38 0.28

July 6th, 2016

- Added a **case_files** folder to the project. This will be the home for data specific files belonging to a particular integration case. The general config file will import the case-specific information and also allow other general options to be added and used by seniority_list. This arrangement will permit multiple cases to be available for analysis, easily selected with one input within the general config file.
- Moved the special condition job assignment data out of the general config file and into the case-specific file(s).
- Moved the notebooks from the notebook folder to the seniority_list folder and deleted the notebooks folder. This will allow the notebooks to run without import issues at this point.
- Added information to the "installation" and "user guide" sections of the documentation. Much more to come.

16.1.39 0.27

July 4th, 2016

- Added *cond_test* function. Used to visualize selected job counts applicable to computed job assignment condition. Primary usage is testing, though the function can chart any job level(s).
- Added *single_emp_compare* function. Select a single employee and compare proposal outcomes using various calculated measures.
- Add installation page to documentation.
- Add notebooks folder to project with "Plotting" and "Run_Scripts" jupyter notebook files.
- Minor code cleanup.

16.1.40 0.26

June 24th, 2016

- Added plotting functions *job_count_charts* and *emp_quick_glance*. Updated the *quantile_years_in_position* plot layout and added helper function *build_subplotting_order*.

16.1.41 0.25

June 18th, 2016

- Initial work for config, job assign, and data source refactor. Initial spreadsheet list data, compensation information, and order proposals will be contained within case-specific folders within the "excel" folder and will be selected with a config file variable. Basic program files are generated from these properly formatted source spreadsheets. Other case data such as job counts, job changes, conditions, and recall schedules will be contained within case-specific python modules.
- Function module docstring cleanup

16.1.42 0.24

June 12th, 2016

- Split *align* function into two functions: *align_next* and *align_fill_down*. Month-to-month data alignment is now accomplished with numpy index alignment vs. pandas dataframe alignment. The new *align_next* function replaces the old *align* function and is primarily used during the job assignment portion of the dataset generation scripts. Net result is an overall **40-50** percent reduction in the time required for dataset generation.
- Other minor code improvements throughout and additions to function documentation.

16.1.43 0.23

June 5th, 2016

- Added *find_row_orphans* and *compare_dataframes* functions to the **list_builder** script. These functions are used to compare dataframe columns and/or entire dataframes. They are able to pinpoint differences within large datasets very quickly, which is particularly helpful during the master list data construction phase.

16.1.44 0.22

June 3rd, 2016

- Added *sort_eg_attributes*, *build_list*, *sort_and_rank*, and *names_to_integers* functions to the **list_builder** script. List proposals may now be rapidly constructed from sample or case master lists. One or more attribute columns may be selected as list order inputs and a "hybrid" ordering achieved by applying variable weightings to those columns.

16.1.45 0.21

May 28th, 2016

- Added **list_builder** script and the *prepare_master_list* function. This is the first step toward manual list building using various attribute weighting, merging, and sorting. This feature is considered a convenience tool only. It may be used for initial list building and ordering prior to analysis and further editing.

16.1.46 0.20

May 27th, 2016

- Added a sample pay table file to the sample_data folder. Sample pay tables may now be generated from the sample file for use with the sample datasets. The sample file simulates a typical Excel input file with pay scale information.
- Replaced the previous pay table generation script with a modified version. The script converts the Excel workbook to Python pickle files for use within the program, either with real or sample data.

16.1.47 0.19

May 26th, 2016

- Added sample master list and sample proposals (both in Excel format) to the sample_data folder. These files can be the source for testing the operation of the program and creating sample datasets when the "sample_mode" option within the configuration file is set to "True". Sample pay-related files will be added soon. Updated several other scripts, including significant updates to the config file, so they will operate with the sample data.
- Added *print_config_selections* function. The function provides a quick report of configuration file selections in a dataframe format.

16.1.48 0.18

May 19th, 2016

- Added **build_files** script. Build supporting files from initial Excel file input such as master data list, proposal orderings, last month percent, etc.
- Added **standalone_no_job_changes** script. Used with most basic dataset creation. This file is rarely used but available if a dataset without any job changes over time is desired.
- Other coding changes to format the program to accept a wider range of list input

16.1.49 0.17

May 16th, 2016

- Added **join_inactives** script. Edited or active employee only lists may now be merged into the original master list which contains all employees including inactive employees (such as sick leave, supervisory, etc.) The inactives may be attached either to the "just senior" employee group active cohort or the "just junior" with an argument option. The resulting list will be sorted and numbered in the new list order.

16.1.50 0.16

May 14th, 2016

- Added *range_diff* plotting function which computes and displays aggregate differential data over time, comparing proposal results with standalone data.
- Modified compute_measures script. A master data file will now be reordered by a specific proposal list order or an order from the editor tool instead of storing separate data files for each proposal.

16.1.51 0.15

May 12th, 2016

- Added *eg_multiplot_with_cat_order* function. Adds flexible x y plotting for most attributes with special color bands and scaling when cat_order is the selected measure. The function is able to select certain employee groups for independent views.

16.1.52 0.14

May 2nd, 2016

- Added multiple controls to the editor interface making the tool easier to use. "One click" recalculation with chart updating is now enabled.

16.1.53 0.13

May 1st, 2016

- Added *editor* function. Editor is an interactive, visual list editing tool for use within the Jupyter notebook. This tool can be used to remove list distortions using comparative data.

16.1.54 0.12

April 26th, 2016

- Added *job_transfer* function.

16.1.55 0.11

April 22nd, 2016

- Added *edit mode* to config file in preparation for visual span selector editing tool.
- Minor documentation edits including adding proper table format for documentation format.
- New differential option added to *quantile_years_in_position* plotting function along with other plot output options.
- Added new *quantile_bands_in_position* plotting function.

16.1.56 0.10

April 15th, 2016

- Initial commit.

CHAPTER

SEVENTEEN

LICENSE

17.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must

make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would

make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing

this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or

for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Pro-

gram, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO

MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> {one line to give the program's name and a brief idea of what it does.}
> Copyright (C) {year} {name of author}
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

> {project} Copyright (C) {year} {fullname} This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

> You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.
>
> The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

CHAPTER

EIGHTEEN

CONTACT

The seniority_list program provides a pathway to fair, equitable, and transparent workforce integration outcome through modern data analysis technology - while significantly reducing the cost, time, and angst historically expended in past procedings.

Please note that while seniority_list was developed with the airline industry in mind, it may be adapted to any industry or group where workers operate under a seniority system.

I have recently co-authored an article introducing seniority_list which was published by Cornell University. Please click here[125] for a download link to the paper.

Questions, comments, suggestions, and consulting inquiries are welcome.

Bob Davison

rubydatasystems@fastmail.net

[125] http://scholarship.sha.cornell.edu/chrpubs/246/

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

J

R

S

T

U

V