MILLIMAN BRIEFING NOTE

Automatic Report Generation

Investigating Automated Reporting Systems

Donal McGinley, FSAI
Joseph Sloan, FSAI

Actuarial reporting systems have changed significantly in recent decades, driven by factors such as improvements in computer software and hardware, and an increase in the onerousness of regulatory requirements. Insurers have embraced new technologies—cloud computing and data science tools and techniques, among others—which have enabled them to develop faster, more automated systems to analyse and report on their businesses.

In contrast, the tools and techniques used to create written actuarial reports have changed little in recent years. Reports are typically produced using word processing software such as Microsoft Word (MS Word). The standard practice is for the report writer to source the information from various locations, and type or paste the relevant data into the report. The approach is typically manual in nature, time-consuming, and difficult to automate.

MS Word is a good tool. It is very flexible, enabling report writers to write professional-looking reports that are in line with corporate branding guidelines. It contains most of the features that one would expect to find in a report, such as text, tables, graphs, lists, section headings, and other elements.

However, MS Word has some shortcomings, including:

1. Writing reports is quite time consuming. This is not a major problem for documents that are written on an infrequent basis, such as quarterly or annual reports. However, time costs can be significant if reports are produced regularly (daily, weekly, monthly), or if multiple reports need to be produced at once (such as a separate report for each individual product, policy, or fund).
2. The process of writing reports is largely manual. The report writer typically types or pastes the relevant data into the report rather than linking the report to the underlying data. This pattern introduces the risk of numbers being entered incorrectly or out of date.
3. Documents are more prone to errors. MS Word contains some dynamic features which enable limited automation, such as dynamic references which can be updated automatically whenever the document is refreshed. However, there is also the risk that the dynamic references will not get refreshed at times and thus display out-of-date data. Users may need to manually check to see if a document has been refreshed, which defeats the purpose of having automatic links to some extent.

In summary, MS Word is a good tool for producing infrequent, one-off reports, and we believe that actuaries will continue to use it for the foreseeable future. However, its manual, time-consuming nature means that it can be costly to produce multiple reports.

In this paper, we have investigated an alternative approach that addresses some of MS Word's shortcomings—specifically, whether the Jupyter Notebook application and the Python programming language can be used to produce professional-looking actuarial reports in an automated fashion. To demonstrate whether this approach would be feasible, we built a prototype system, as outlined below.

We believe that the Jupyter/Python approach is compelling for a number of different reasons, including the following:

1. Jupyter Notebook can be used to create professional-looking documents.
2. Python is one of the most popular, widely used programming languages in the world. It has strong actuarial modelling, data exploration, and data visualisation capabilities. Therefore, we believe it could be used as a powerful engine to pull together the data from different sources, perform calculations on the data if necessary, and generate the text, tables, and graphs to be used in the automated report. Python is increasingly being used by insurance companies to automate actuarial processes, and it would seem to be a natural step for these companies to use Jupyter/Python to embed the results into a report.

3. Jupyter interlinks seamlessly with Python and can be used to embed the Python-generated text, tables, and graphs into a professional-looking document.
4. It is relatively straightforward to draft reports in Jupyter Notebook. The draft document can be viewed and edited in real-time by the report writer. The person who drafts the initial report will have full access to the underlying data, as well as the text, tables, and graphs to be used. These elements can be amended as necessary while drafting the report.
5. Jupyter Notebook is language-agnostic, which means it can run code from many different languages, including R and SQL. The report writer can use Python or a different language such as R if they so desire.
6. Both Python and Jupyter are regarded as being relatively easy to learn and are open-source; therefore it is feasible that companies could adopt such a system without great difficulty.

Desirable features of an automated report

Before creating the prototype report generation system, we made a checklist of desirable features. Specifically, the system should be able to generate reports that have a professional style and incorporate company brand guidelines on typography, colours, charts, and other elements. Also, the system would ideally have all the main report writing features that are available in MS Word, for example:

- Formatted text (headings, paragraphs, bullet points, font, and colouring scheme)
- Tables and graphs
- Footnotes / endnotes / hyperlinks / watermarks / captions
- Ability to embed graphics, e.g., company logo
- Ability to convert into a PDF
- Ability to print as an A4 document

It is important that an automated report writing system is interoperable with other files and software tools to avoid the need for manual intervention in the process. In order to assess the system's automation capabilities, we considered the following:

- Is it possible to source data from multiple files (CSV, Excel, SQL, etc.)?
- Is it possible to populate text with values from the data (i.e., text interpolation)?
- Can documents be saved to the server automatically?
- Is it possible to include the automated report process in an automated workflow (i.e., multiple runs can be set off, one after the other)?

We also examined whether the automated report system would be sufficiently robust to perform this type of work. How would the accuracy of reports compare to the traditional approach of producing reports in MS Word? Is the automated system auditable or is it a black box that is hard to follow for those not familiar with the process? Is the system sufficiently easy to use? In particular, is it feasible that the people who are responsible for performing and checking actuarial calculations can use this system to report on the results that they have produced?

Choosing a system

There are numerous options that address some of the shortcomings of MS Word. For example, users could automate MS Word documents using applications such as PowerApps and PowerAutomate, or Python libraries such as docx. Alternatively, report writers could move away from MS Word and instead write reports using open-source software, such as RMarkdown and the R programming language, or Jupyter Notebook with Python. These approaches have different strengths and weaknesses. In this paper, we have investigated using Jupyter Notebook and Python, as we believe this combination offers a powerful, reliable system for producing reports.

Jupyter Notebook is an interactive open-source web application that allows you to integrate code and the output into a single document that combines calculation, narrative text, visualizations, and other elements. It has become very popular in recent years, particularly among data scientists. Jupyter Notebook contains many of the desirable features of an automated reporting system. It can be used to convert easy-to-write Markdown text into a richly formatted HTML document, which can be viewed in a web browser or converted to PDF format. Jupyter Notebook can be viewed in a web browser (such as Google Chrome or Microsoft Edge) and typically contains a single column of runnable cells. Each cell can contain either text or code.

Jupyter Notebook serves as a fully fledged user interface for the Python programming language, and can be used to run any arbitrary piece of Python code. Users can create code cells that import and process data, perform calculations on the data, save the calculation results to the server, and create summary graphs or tables of the calculation results. Users can also create Markdown cells, which contain headings or explanatory text. Users can also build automated reports by using the text cells to structure the document, and by using the code cells to source the data, perform the relevant calculations (if any) and then produce the summary graphs, tables, and explanatory text describing the results. The notebook can then be saved into a user-friendly format such as HTML (which can be viewed in any web browser) or a PDF document. The code cells can be hidden before converting the notebook to HTML or PDF to ensure the final report does not include any unnecessary code and contains only text, graphs, and tables.

Jupyter Notebook supports many different programming languages, including Python, R, SQL, and Matlab. We built the prototype using Python as it is a general purpose programming language and is regarded as one of the best languages for automating tasks. It is relatively easy to use (compared to other programming languages) and has a wide range of existing capabilities for automating tasks. Also, it has strong data science capabilities, including the ability to import data from a wide variety of sources (including Excel, CSVs, and SQL databases), and strong calculation and visualisation capabilities. These features are useful when building an automated reporting system.

Jupyter enables users to create HTML documents and to format those documents using a Cascading Style Sheets (CSS) template, a widely used method for formatting HTML documents and making them presentable to users. These capabilities are important for good reasons. When writing reports, actuaries may be required to use certain fonts and colours as indicated by corporate branding guidelines. CSS makes it possible to create a CSS template that describes elements such as font, colour, size, spacing between paragraphs, and other effects, and thereby ensures that final documents conform to relevant branding guidelines.

In summary, we developed our prototype for the automated report system using Jupyter Notebook, Python, and CSS. It was our view that this selection of tools is capable of implementing the desired features outlined previously.

Prototype capabilities

In this section, we discuss whether we were able to implement all of the desirable features listed previously and outline the strengths and weaknesses of the prototype compared to MS Word. An example of a report produced using Jupyter/Python/CSS is included in the Appendix.

FORMATTED TEXT (HEADINGS, PARAGRAPHS, BULLET POINTS, FONT, AND COLOURING SCHEME)

Jupyter Notebook has a cell-based structure, and typically the cells are arranged in a single column. There are two main types of cells—Markdown cells, containing text or headings, and code cells, which can run any arbitrary piece of Python code. We can write formatted text inside a Markdown cell.

Markdown is a lightweight language which allows the writer to draft a report in plain text, and then enhance formatting by adding some simple metadata to the text. For example, any line which starts with the hash symbol (#) will be shown as a heading, and any line which starts with an asterisk (*) will be shown as a bullet point. Jupyter automatically converts the plain Markdown text into richly formatted, coloured text.

There are three main steps for writing formatted text. Firstly, type the text in plain Markdown format in a Jupyter Notebook cell. Secondly, create a CSS template that can apply the desired formatting scheme to the plain Markdown text. Lastly, use Jupyter Notebook to convert the Markdown text into HTML and to apply the CSS styling scheme to the HTML text. The end result is text shown in richly formatted HTML rather than plain Markdown.

We give some examples of this below.

1. Plain Text in a Paragraph

It is straightforward to write a paragraph in Markdown. Simply draft the paragraph as plain text in a Markdown cell in Jupyter Notebook.

In the prototype, we typed the following input:

```
This is a paragraph of plain text
```

We then ran the cell, to convert it to formatted HTML output. In our CSS template, we specified that the normal text should use Times New Roman font. After running the cell, we got the following output:

This is a paragraph of plain text

2. Headings

If we need to change any text formats, we can add some indicators to the Markdown text. For example, for text to be shown as a first-level heading, we add a hash mark and a space (#) to the beginning of the line. This informs Jupyter Notebook that the line should be treated as a heading rather than plain text. If we want to add a second-level heading, we add two hash marks and a space (##) to the beginning of the line.

The following screenshots give an example of the headings in our prototype. The report writer types the following text:

```
# This is a First-Level Heading
```

```
## This is a Second-Level Heading
```

```
This is some normal text
```

The report writer then runs the cells, and Jupyter converts the Markdown into the following HTML:

This is a First-Level Heading

This is a Second-Level Heading

This is some normal text

In the prototype, for the purposes of demonstration, we specified in the CSS template that the first-level headings should be shown in blue and that second level headings should have a smaller font size and be shown in red.

3. Bullet Points and Ordered Lists

It is also possible to use bullet points in Markdown cells. These can be nested by indenting the line, similar to MS Word. Solid square and solid circle bullet points can be selected. Similarly, ordered lists can be created in a Markdown cell.

If you give the following text to a Markdown cell:

```
* A bulletpoint
* Another bulletpoint
    * Nested bulletpoint
* Another bulletpoint

1. Ordered List
    1. Ordered List
    1. Ordered List
1. Ordered List
```

It will display the following list:

- A bulletpoint
- Another bulletpoint
    - Nested bulletpoint
- Another bulletpoint

1. Ordered List
    A. Ordered List
    B. Ordered List
2. Ordered List

4. Font and Colouring Scheme

Emphasizing text with bold, italics, and underlining is easy to do using Markdown. It is also possible to change the font size, style, and colour within each Markdown cell. However, if the purpose of a Jupyter Notebook is to generate a report, it is likely easier to accomplish this task using CSS. This means the specifications can be set for all cells in the notebook, rather than repeating the font specifications and colours in every Markdown cell, to ensure the report is presented in a consistent way.

In summary, it is quite straightforward to write text and headings using Markdown text cells in Jupyter Notebook.

5. Tables and Graphs

We investigated whether the prototype was capable of producing tables and graphs that could be used in formal reports.

The table below was produced by our prototype. In this example, the table is based on a dummy dataset and is constructed using the Pandas Python library. CSS is used to format the table by applying, for example, the company colour scheme and by also selecting a centre alignment to present the table.

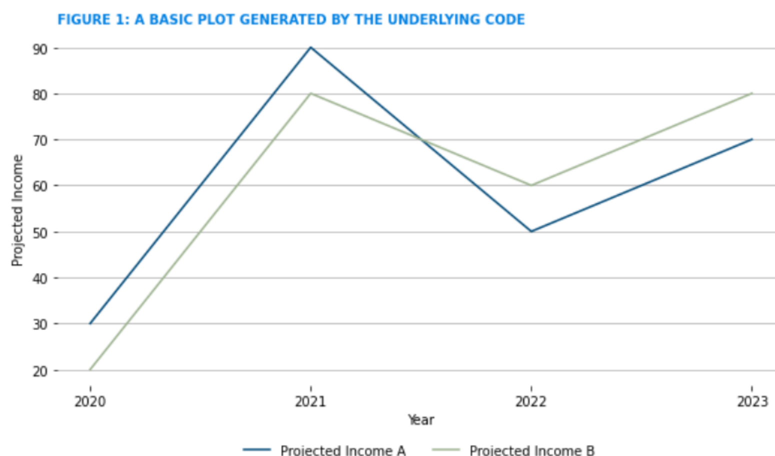
Year	Projected Income A	Projected Income B
2020	30	20
2021	90	80
2022	50	60
2023	70	80

One of Python's main strengths is data visualisation. Python has a host of libraries such as Matplotlib, Seaborn, and Bokeh, which are used to create highly customized data graphs. Many different types of graphs are possible, such as bar charts, line graphs, pie charts, scatter plots, histograms, and interactive charts.

We used the Seaborn package to create the graph below. We adjusted the default chart settings with the following formatting modifications:

- The chart uses the Milliman colouring scheme.
- The font size of the chart axis, legend, and title is set to 10 pt.
- The border of the chart is removed.
- Gridlines are included on the Y axis only.
- A different colour coding is used for the chart title. Bold font is also used and the title is positioned in the top left.
- The legend is positioned underneath the chart in two columns.
- The size dimensions of the chart can be adjusted.

FIGURE 1: A BASIC PLOT GENERATED BY THE UNDERLYING CODE



The code to create the graph using Seaborn is relatively straightforward and the formatting modifications can be made using only Python code. In this example, CSS was not needed for any reformatting.

In summary, the prototype is capable of producing tables and graphs that can be used for reporting. Once these are set up using Python and a suitable CSS template, the process is automated and the charts can be embedded in the document without any manual intervention such as refreshing data in MS Excel and pasting into MS Word. The main challenge is that any further refinements to the charts would require some basic Python and/or CSS skills to implement, however this should not be a major issue for regular reports that do not change on a frequent basis.

6. Footnotes / Endnotes / Hyperlinks / Watermark / Captions

In the sections above, we found it straightforward to add the main elements of a report (text, headings, tables, graphs) in Jupyter Notebook. In practice, actuarial reports may also require other elements such as footnotes, endnotes, hyperlinks to other documents, captions, and watermarks. We found that some of these (endnotes, hyperlinks, and captions) were relatively straightforward to include, and other elements, such as footnotes and watermarks, more challenging to add within the timeframe of the research project. Any company who adopts this system should consider these items carefully during the initial design stage.

7. Embedding Graphics

Embedding graphics such as a company logo into a Jupyter Notebook is straightforward. The only steps required are to convert the Jupyter Notebook cell to a Markdown cell, and drag and drop the image to be embedded (e.g., png file). Alternatively, we could use CSS to embed the graphics. This is a particularly useful feature for customising the appearance of reports according to company branding.

ABILITY TO PRINT AS AN A4 OR PDF DOCUMENT

With Jupyter Notebook, the prototype document can be converted into either a standalone HTML document or a PDF document. Although we were able to produce a professional-looking HTML document that could be viewed in a web browser, it was more difficult to produce a professional-looking printed document in A4 size. This is perhaps not surprising, given that Jupyter is an application designed for viewing in web browsers rather than for creating printed pages. Our prototype has most of the functionality required to create printed pages, however there were some elements that did not come by default and were more difficult to implement in the prototype. For example, printed documents typically have headers and footers on each page. In our example, we could easily apply a single header at the start of the document and a single footer at the end of the document, however it was more difficult to create headers and footers on each A4 page. We generally believe that it is possible to create printed documents using this system that are as professional-looking as traditional MS Word reports. Appendix A contains an example of a report generated using the prototype.

AUTOMATION CAPABILITIES

Another area we focused on in the prototype testing was automation—this is an essential capability of any automated report writing system.

1. Importing Data

Data for reports is commonly compiled from a variety of sources, such as MS Excel files, CSV files, and SQL databases. One of Python's main strengths is its ability to interact with other languages and software, as it can easily import data from Excel, SQL, and CSV files. The Python library called Pandas is commonly used for his type of work. The following code loads the library, imports data from an MS Excel file (used to populate the table presented earlier) and prints out the first four rows. Note how easy it is to import the data into the Notebook.

```
import pandas
proj_income_table = pandas.read_excel("Projected Income A.xlsx")
proj_income_table.head()
```

	Year	Projected Income A
0	2020	30
1	2021	90
2	2022	50
3	2023	70

In addition, Python has very strong calculation capabilities. Therefore, if there is a lot of manual work to generate tables and graphs for a report using the data available, most or all of the calculations can be done within Python itself. This means there is less reliance on other tools such as MS Excel, which in turn reduces operational risk and leads to a more automated process.

2. Text Interpolation

Python supports text interpolation (also known as string interpolation). This can be used to populate text with values from the data. For example, for a sentence in the main body of the report that says, “Projected income for company A in the first year is €30,” it would not be necessary to type €30 in the text—this can be sourced directly from the data using Python code. The code is shown below—this imports Markdown, which is used for text interpolation, and finds the first row of the projected income A table and includes this value as a string in a sentence. The output is shown under the code:

```
from IPython.display import Markdown
a = df.at[0,'Projected Income A']
Markdown("Projected income for company A in the first year is €"+str(a))
```

Projected income for company A in the first year is €30

You can also populate text with variables in the notebook. For example, a variable could be created called “date” set equal to “31 December 2020.” The report would then reference this variable in the text so that inserting it manually in multiple places would not be necessary. Similarly, you could add logic to the text using text interpolation. For example, it is possible to have a variable that calculates the difference in percentage terms between projected income in year 1 and year 2 in the table above. If this number is positive, then the text would return “Projected income in year 2 is x% higher than year 1” where “x” and “higher” are inserted using variables. Alternatively, if this number is negative, then the text would return “Projected income in year 2 is x% lower than year 1”.

In this way, Python can be used to automatically generate the text with data. Please note that, while the screenshots above contain code, in practice we would hide the code before creating the report. The final report will show the output from the code but not the code itself.

3. Saving to the Server

We found that it was quite straightforward to save the code, audit trail(s), and report(s) to the server after running the prototype.

4. Automated Workflow

We used a Python library called Papermill to automate the report generating process. Papermill allows you to execute the same Jupyter Notebook, but with different variables defined in a master notebook. You can use Papermill to generate the same report but with a different set of inputs. For example, if you need to produce a report on the monthly performance of each fund in a portfolio, you can build a “for” loop that loops through the inputs and generates multiple fund reports, all created by a single run on Python.

PROTOTYPE IN PRODUCTION

Another important consideration is whether the prototype is robust and can be used in a production environment. We believe that all elements of our prototype are quite stable and are suitable for this purpose as Python is used in production by thousands of companies around the world. Jupyter is one of the most widely used tools in the data science community. CSS is a reliable and stable language and is widely used in production by many leading organisations across the globe.

The main benefit of automation is often seen as faster processes and less manual work, but these processes also perform tasks more accurately. Most human errors in MS Word are due to the user forgetting to refresh the data, pasting in incorrect data, “fat-finger” mistakes, and so on. However, technology can remove most of these manual data entry errors. The prototype addresses many of these issues. Python has good run log and audit trail capabilities, which improves the robustness of the process. Inputs to reports are clearly documented and there is an automated process for updating data in reports, making them less error prone.

Conclusion

This research has demonstrated that it is feasible for insurers to build automated reporting systems using currently available open-source tools such as Jupyter Notebook and Python. The report produced by the prototype has most of the main features you would expect in an MS Word document: it can be structured in a similar format with headings and paragraphs; a variety of text formatting options are possible; tables and graphs are informative and professional-looking; corporate branding guidelines can be applied; and so on.

The prototype has some strengths and weaknesses compared to MS Word. We believe that MS Word is likely to continue to be the tool of choice for producing infrequent, ad-hoc reports. It is well known, easy to use, and extremely flexible.

The prototype offers a general purpose system which can be used to create many different types of reports. One strength of this system is that the report writer can do all of the work in a single place. They can import all of the required data and do all of the data processing and visualisation within the document itself. Checks and controls can be embedded in the process and an audit trail can be produced automatically. Less manual intervention means that reports can be produced more quickly and are less error-prone.

The system is easier to automate than MS Word. It is relatively straightforward to update the system so that it can create multiple reports on a single day (e.g., a different report for each individual product or fund), or regular reports (e.g., daily or weekly reports).

However, the Jupyter/Python system is less flexible than MS Word, as MS Word enables users to make manual changes at any time. In contrast, our prototype uses a one-size-fits-all CSS template and the final output from our prototype is essentially a read-only document. Users have no ability to change it after it has been created. For example, if the report writer wanted to change the column width of a table, it is trivial to resize the column in MS Word, however it would be difficult to do so in our prototype as they would need to change the CSS file to achieve the desired width and then regenerate the report with the updated CSS file.

In general, we expect most actuaries would be able to learn how to use calculation software such as Jupyter and Python quite quickly. From an actuarial perspective, we expect the main difficulty faced by any actuarial department who adopts this system would be in the creation of the CSS template. CSS is not directly relevant to actuarial work and we expect few actuaries currently know how to set up a CSS template. Companies may need to get external help, which should not be difficult to find, as CSS is widely used around the world. Given that corporate branding guidelines tend to remain quite stable over time, we would expect the template to remain stable as well. Therefore, most of the difficulties would likely be overcome during the initial design phase with little ongoing overhead in maintaining the template. Companies who adopt this system should ensure that they carefully design the initial CSS template, covering all of the elements they expect to include in future reports, to minimise the need to substantially change the CSS template in future.

Please note that most users will not need to know how to update the CSS template. Report writers can write the report in Jupyter using Python, and can simply run a single line of code to apply the existing one-size-fits-all CSS template to the Jupyter document.

MS Word is a powerful tool that will continue to be used for report writing for many years to come. However, there are clear benefits to automated reporting and it will be interesting to see if companies will embrace new technologies to automate these processes.

Milliman is among the world's largest providers of actuarial and related products and services. The firm has consulting practices in life insurance and financial services, property & casualty insurance, healthcare, and employee benefits. Founded in 1947, Milliman is an independent firm with offices in major cities around the globe.

ie.milliman.com

CONTACT

Donal McGinley
donal.mcginley@milliman.com

Joseph Sloan
joseph.sloan@milliman.com

Appendix A: Example Report Created Using Our Prototype

Sample Output from Jupyter/Python/CSS Prototype

This report has been generated using the Jupyter Notebook/Python/CSS prototype. Some of the features are shown below.

Code cells are used to run Python code e.g. import data, perform calculations on the data, create summary graphs or tables of the calculation results and so on. The code cells are hidden before saving to HTML or PDF and are not visible in the output. The saved file will only contain the output e.g. text, tables and graphs. The other type of cell is Markdown - these contain headings or explanatory text. This paragraph is written in a Markdown cell.

Heading 1

Heading 2

If we want the text to be shown as a first-level heading, we add a hash mark and a space ("# ") to the beginning of the line. This informs Jupyter notebook that the line should be treated as a heading rather than plain text. If we want to add a second-level heading, we add two hash marks and a space ("## ")to the beginning of the line. The headings font colour can be changed to align with branding guidelines e.g. for the purposes of demonstration, we specified in the CSS template that the first level headings should be shown in the red, and that that second level headings should be smaller and shown in blue colour often used in Milliman reports.

Text can also be ***emphasized*** in Markdown cells e.g. the word "emphasized" in this sentence is in bold and italics.

You can use text interpolation to merge the text with variable values. There is code cell in this notebook which sets $\boldsymbol{a} = \mathbf{1}$. The Python formula will automatically extract the values from the results, the user does not need to manually type a = 1 in the text i.e. the digit "1" is not typed in this paragraph, instead a Python formula uses the value of the variable "a" from the code cell. In this way Python can be used to to autogenerate the text with the results.

Nested bullet points and numbered lists are also possible:

- A bulletpoint
- Another bulletpoint
  - Nested bulletpoint
- Another bulletpoint

1. Ordered List
   A. Ordered List
   B. Ordered List
2. Ordered List

Similarly, check boxes can be used:

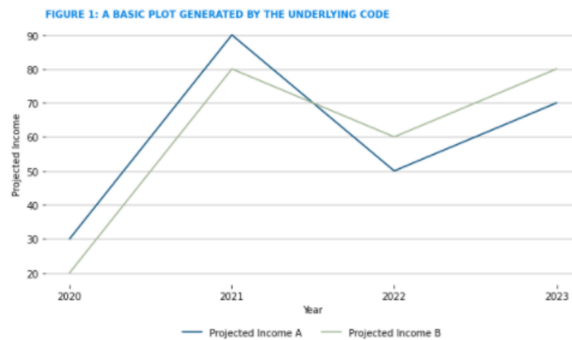
- [x] Some task
- [ ] a different task

Here is a basic table generated by the underlying code. The table is based on a dummy dataset and is constructed using the Pandas Python library. CSS is used to format the table, for example, the Company colour scheme is applied and the font of the column names is in white.

Year	Projected Income A
2020	30
2021	90
2022	50
2023	70

Year	Projected Income B
2020	20
2021	80
2022	60
2023	80

Here is a basic plot using the data in the tables. The graph was created using the Seaborn Python library - the graph was reformatted to look like a standard Milliman graph.



Headers and footers can be implemented in the document. A footer is created below by adding an image to the end of the document.

ABC Insurance Ltd.
Actuarial Function Report as at 31 December 2020
1 April 2021
4

© 2021 Milliman, Inc. All Rights Reserved. The materials in this document represent the opinion of the authors and are not representative of the views of Milliman, Inc. Milliman does not certify the information, nor does it guarantee the accuracy and completeness of such information. Use of such information is voluntary and should not be relied upon unless an independent review of its accuracy and completeness has been performed. Materials may not be reproduced without the express consent of Milliman.