Effective Excel and preparing for Python

Introduction

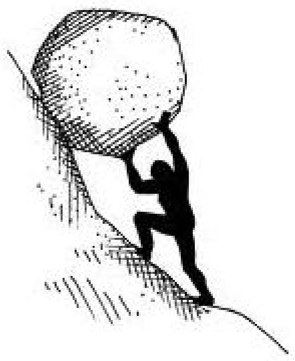
Given some data problem, your first instinct may be to approach it using a spreadsheet program, like Microsoft Excel. While Excel is great for data entry, it is more problematic to use it for processing data, generating tables for publications, summary statistics, and for making figures. The drag and drop nature of spreadsheet programs means that it can be very difficult, if not impossible, to replicate your analysis.

This document is intended to help standardise and improve your spreadsheet use. It will also lay the foundations for a transition to a fully programmatic analysis approach to your research, either by your future self or by others seeking to continue your work. To do this, we describe approaches to workbook organization, data cleaning, quality control practices, and how to export data for automated use. We also demonstrate the reading of data into Python, and how to connect Python programs directly to your spreadsheets, similar to the concept of macros, to create workflows that move beyond what is practical to achieve with Excel alone.

There are several spreadsheet programs you may be using, including: LibreOffice, Microsoft Excel, Gnumeric, OpenOffice.org. We have no preference over the program you select, and the contents of this document are generally applicable, although we will refer to Excel in this document. For those of you who are interested in moving into an automated workflow, we encourage the use of Anaconda Python (Python Version 3). This will work regardless of your Operating System, and you do not need Administrator privileges on your machine to install and run it. Although this document describes several uses of Python, we don't provide an introduction to the Python language itself, as many excellent and free resources doing exactly this already exist. For a good place to start we recommend the Software Carpentry training events, which run regularly at UCL. If you have not yet been invited to attend this course, but wish to, please keep an eye on upcoming UCL training courses and apply to a future Software Carpentry event.

1. Pay it forward

We encourage you to think of others first when creating any analysis. Your work should ideally be useful to your group and other researchers. It is likely that you will have started your own research based on a foundation of someone else's work, and, in turn, someone else may likewise start their work based on your own. Everyone in the research chain benefits from considering that, while a piece of work is only written once, it may be read an unlimited number of times.



As you have no real way of knowing how people in the future will use your work, or with what technology, you should aim to be as clear, consistent, and compatible as possible. Keep this principle in mind first and foremost, or else your work will likely be abandoned at the end of your research project, and each successive generation of researchers will need to reconstruct what you have done in order to move forward; a fate reminiscent of Sisyphus!

As an added benefit, this 'others first' mindset will also benefit your future self: when you come back to work later, or have to adapt some analyses to satisfy a reviewer, you will be grateful for a clear description of what you did originally to build on. These practices also increase the reach and impact of your research, thus enhancing your academic career prospects.

Summary: *Think of others and try to avoid creating 'abandonware'.*

2. Documentation

In addition to your Workbook file, you should include a supporting 'readme' document. At minimum this should be a simple text file, containing:

- Author(s) name(s), affiliations, and basic contact information
- Date of creation
- Project information (e.g. name, grant, primary investigator, or other identifying info)
- Intended license of the Workbook (see https://en.wikipedia.org/wiki/Software_license)
- Info about the raw data (original source, units, missing values, and supporting notes)
- References (e.g. papers or books from which you took formulas for processing)

Summary: *Make a readme file to go with your spreadsheet.*

3. Data Consistency

> *"A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines."*
> – Ralph Waldo Emerson

The most common mistake made is treating a workbook like a lab notebook; that is to say, relying on context, notes in the margin, and the spatial layout of data and fields to convey information. A workbook laid out in such a manner will be almost impossible to use in an automated workflow, and it may even be difficult for you the author to understand it after some time has passed, let alone another person!

Consistent design is the foundation of creating work that is clear and reusable. Here are some key patterns to keep in mind:

- Never modify your original dataset or you may become confused in the future, create a copy and modify that instead.
- If you are creating a modified/cleaned dataset, document the steps you took.
- Spreadsheet data should be organised so variables (e.g. weight or temperature) are in columns, while observations (instances of weight or temperature) should be in the rows.
- Do not combine multiple pieces of information into single cells. E.g. Information like “True, Red, 10” should be in three separate cells.
- You should clearly partition the sheets in a workbook by purpose. Sheets that contain raw data should be separated from sheets that process these data, i.e. all cells in the raw data sheet should be hardcoded values, not formulas. Any values derived from applying formulas to your raw data should be in a separate sheet.
- Separate tables should be on separate sheets, with distinguishing sheet names.
- Headings for tables should always be included, and be in the first row only, without spaces in the names. You may use camelcase or underscores to join multi-word heading names. E.g. “sample 1” should be ‘sample_1’. Names should be meaningful.
- If columns become too long, you can Freeze headers. Do not repeat headers inside the tables.
- Documentation describing the the raw data’s origin, units, missing values, and any other relevant information should be included, in a linked document or readme file kept in the same folder as the workbook.
- Data tables should not contain whitespace (blank cells). Instead, use missing values to indicate a lack of data. Also, note that a blank cell is not equivalent to a 0.
- Do not mix missing values. E.g. do not change between the use of #N/A and -999.
- Formatting should not be used to convey information. E.g. instead of highlighting cells in a table to denote whether a measurement was calibrated or not, a new column should be added with a heading ‘Calibrated’, and binary values indicated True or False. If formatting is required, it can be applied to columns via conditional formatting rules: to do this, select a range of cells, click ‘home’ > ‘conditional formatting’ > ‘new rule’.
- Information added as comments or units in cells will be lost when reading the data automatically. These data should be included in readme notes instead.
- Use plain text characters only: don’t use formatted, or special characters in your headings, data, or notes.
- If your data includes dates or times it is best to store it in separate columns: e.g. instead of entering ‘1985-Jan-11’ into a row of a column called ‘Dates’, you should, create three columns ‘Year’, ‘Month’, ‘Day’, and enter 1985, 1, 11 respectively into the rows. This will make life easier for you in the future, and prevent unexpected behaviour[1].
- As the aim is to write your spreadsheet files so they can be read by either humans or programs, you should avoid unusual formatting to make the sheets look pleasing: it may compromise the ability of programs to read the spreadsheets.

[1] More info at: http://www.datacarpentry.org/spreadsheet-ecology-lesson/03-dates-as-data.html

Pro tip: By using missing values your tables should be contiguous, containing no whitespace. Because of this, you can rapidly move and select data from your tables with the tab, shift and arrow keys. These 'Shortcut Keys' are usually common among all Spreadsheet programs. If you find yourself using the mouse wheel excessively to navigate and select your data, try using the Shortcut Keys instead.

Summary: *Isolate input data into separate Sheets with consistent style.*

4. Quality Assurance

When you have a well-structured data table, you can use several simple techniques within your spreadsheet to ensure the data you enter is free of errors during the data entry phase. For example, if research is being conducted at sites A, B, and C, then the value V (which is right next to B on the keyboard) should never be entered. Likewise if one of the kinds of data being collected is a count, only integers greater than or equal to zero should be allowed. We can set cells to only accept pre-approved values.

To control the kind of data entered into a spreadsheet we use Data Validation in Excel.

1. Select the cells or column you want to validate
2. On the Data tab select Data Validation
3. In the Allow box select the kind of data that should be in the column. Options include whole numbers, decimals, lists of items, dates, and other values.
4. After selecting an item enter any additional details. E.g. if you've chosen a list of values, enter a comma-delimited list of allowable values in the Source box.

Quality assurance can make data entry easier as well as more robust. For example, if you use a list of options to restrict data entry, the spreadsheet will provide you with a drop-down list of the available items. So, instead of trying to remember how to spell "*Drosophila pseudoobscura*", you can just select the right option from the list.

Summary: *Manual data entry can be automatically supervised to minimize errors.*

5. Quality Control

While Quality Assurance examines data for correctness prior to being entered, Quality Control refers to the process of checking your data for errors after you have the dataset[2]. You can use filters to sort your data, or to apply conditional formatting as a means of Quality Control. It is advisable to save a copy of your spreadsheet before performing any Quality Control operations, and ensure your data is stored as values and not as formulas. Because formulas refer to other cells, and you may be moving cells around, you may compromise the integrity of your data if you do not take this step. It is also crucial to document all steps taken in Quality Control in a readme file.

[2] http://www.datacarpentry.org/spreadsheet-ecology-lesson/04-quality-control.html

Sorting: Bad values often sort to bottom or top of the column. For example, if your data should be numeric, then alphabetical and null data will group at the ends of the sorted data. Sort your data by each field, one at a time using a filter. Scan through each column, but pay the most attention to the top and the bottom of a column. If your dataset is well-structured and does not contain formulas, sorting should never affect the integrity of your dataset.

Conditional formatting: This is a great way to flag inconsistent values when entering data. This effectively color codes your values by some criteria or lowest to highest, making it easy to scan your data for outliers. In Excel, the option to apply conditional formatting is under **Format** then **Conditional Formatting**.

While it is useful to be able to perform these types of quality checks in spreadsheet programs, and to be aware that these features exist, it is usually better to do this in a programming language.

***Summary**: Use techniques like filters and conditional formatting to examine your dataset.*

6. Identify and isolate your parameters and constants

You should identify any values that your workflow depends upon, such as constants (a.k.a. magic numbers), or parameters that you (or others) may wish to alter in the course of an analysis. These values should be clearly listed in a single table. Explanations specifying what these values are, and why someone may wish to vary them, should be included in a separate readme file or contained in the sheet (in a 'notes' column). Crucially, **these values should only appear once in your entire workbook.** All uses of these values should be tied to the single cell where they appear, so that if you alter the value, the entire workbook automatically updates to use the altered value.

You should apply some common sense to identifying your parameters. For example, if your calculations require long lists of column or row numbers (indexes) these should probably not be placed in the table of variables.

Summary: *All parameters should be collected into a single table in a unique sheet and appear only once in the entire workbook.*

7. Dynamic Processing

Data analysis in spreadsheets usually requires a lot of manual work. If you want to change a parameter or run an analysis with a new dataset, you usually have to redo everything by hand. It is also difficult to track or reproduce statistical or plotting analyses done in spreadsheet programs, which can cause significant difficulties when you want to go back over your workflow, or if someone asks for specific details of your analysis. Although there is no way to totally overcome these problems, we can try to minimize them by creating dynamic workbooks.

Essentially, you should aim to write your workbooks so that if the raw input data were updated, the workbook would automatically process these data and produce updated results. While this may not be 100% achievable in every case, you should try and get as close to this as possible, and flag in your documentation the places the places in your workflow (including sheets and cell references) where further manual operations are required to produce results.

This dynamic updating is achieved by linking cells together, across different sheets, so that as input data or variables are updated, a cascade of changes are triggered. A typical workflow may have a sheet containing raw data, another sheet containing variables, and at least one sheet of cells that read these inputs and use them to perform some calculation (a processing sheet). Finally you should have a sheet for displaying summary results, tables and/or graphs as needed, linked to the processing sheet.

If complex processing is required, beyond the functions of Excel, and your workflow is broken up by the need to run Visual Basic macros, we would encourage you to move towards either: 1) exporting your data and working in Python fully, or 2) writing your macro in Python via the xlwings package. It will likely be far easier to meet your needs using Python than the VBA language. You should take the need for macros as an indication that your analysis requirements have moved beyond the realm of spreadsheets.

Summary: *Minimise the amount of manual work required to go from your data to your results.*

8. Importing and Exporting data

Sometimes data you import will require parsing, or formatting, before it appears in the row and column layout that spreadsheets are designed for. You may have encountered this when importing data from a file, only to find that your spreadsheet program condensed every item in a row to a single cell. (e.g. placing “1,2,3,4,5” in one cell instead of 5 separate cells). There is usually an operation called “Text to Columns” built into spreadsheet programs that you can use to tell the software how to convert these data into individual cells (e.g. by separating cells for every comma). For example, in Microsoft Excel (version 14), this operation is built into the Data tab.

When you want to export your data tables from Excel to be read by software such as Python, used by other people, or included with journal publications, you will need to export it into a suitable format. Usually, for data in tables this should be in CSV format (meaning comma separated values). It is generally not a good idea to try and share data with colleagues in spreadsheet format for a variety of reasons, including excessive file size, format limitations, and the inadvertent transmission of metadata. We recommend keeping both your original data files, and any exported data files you may create in your project folder. These files can be placed in a subfolder and compressed to save space if required.

To export a table: the sheet containing the table you wish to save should have nothing in it except for the table of values. The cells should be values, not formulas. If your table contains formulas, and you need to convert the cell contents to values, you can copy and paste the table, overwriting itself, using ‘Paste Special’ instead of a regular paste command. This will enable you to select ‘Values’, pasting hardcoded values instead of formulas. There should not be supporting text anywhere in the sheet, i.e. it must simply be a table, with a heading, hardcoded value entries, and no blank spaces. When the sheet is selected, you should then be able to ‘export’ or ‘save as’ in CSV format. This will save the active/selected sheet only, as a plain text file, with the cell values separated by commas. Be careful to name the CSV document so as not to conflict with any other project files. Name the CSV file something that relates to the file contents in a meaningful way, without whitespace in the name, e.g. ‘raw_data_series_1.csv’.

If you have created a cleaned or modified dataset in your work, you should ensure that both the original and your modified dataset are saved to CSV. You should also ensure the steps you took to modify the data are documented in your readme file.

Summary: *Maintain original data, and export tables in CSV format for broader use.*

9. Reading exported data in Python

Spreadsheets are good for data entry, but often people tend to use them for much more, such as: generating data tables for publications, summary statistics, and for making figures. Because of the graphical, drag and drop nature of spreadsheet programs, it can be very difficult, if not impossible, to replicate your steps (much less retrace anyone else's), particularly if your stats or figures require you to do more complex calculations. Furthermore, in doing calculations in a spreadsheet, it’s easy to accidentally apply a slightly different formula to multiple adjacent cells.

Conversely, when using a command-line based statistics program like Python, it’s practically impossible to accidentally apply a calculation to one observation in your dataset but not another unless you’re doing it on purpose. However, there are circumstances where you might want to use a spreadsheet program to produce “quick and dirty” calculations or figures, or perhaps do some data cleaning, prior to importation into a statistical analysis program.

We will not cover a basic introduction to Python in this document, as many excellent and free online resources are avialable (we recommend starting with http://swcarpentry.github.io/python-novice-inflammation/). We will assume that you have Anaconda Python installed, and can start a shell session. We will also assume that you have exported an example table from a spreadsheet in CSV format, randomly created for this demonstration, called ‘mydata.csv’. It has two columns, labeled A and B, of integers and booleans, with 6 rows of data.

To read this hypothetical data file into Python, you need to start a Python session, e.g. by using the Anaconda Launcher, and then type the following:

```
In [1]: import pandas
In [2]: df = pandas.read_csv('/Users/Julia/Data/mydata.csv')
In [3]: print(df)
Out[3]:
   A      B
0  1   True
1  2  False
2  3   True
3  4  False
4  5   True
5  6  False
```

The end result of this operation is to store the table into a data structure in memory which we have called 'df'. Note, that we are using a tool called Pandas to read our data. It is able to turn them into a data structure that is reminiscent of a spreadsheet array, and will be very convenient for you to work from. We encourage you to read about Pandas further and to use it in your work (http://pandas.pydata.org). Thanks to community tools like Pandas, it is usually painless for you to read complex datasets into languages like Python as we have just done!

You may also note that we have explicitly specified the path to find the file 'mydata.csv' in our example: our file was stored in a location accessible from '/Users/Julia/Data', but of course this path will be different on your computer.

Summary: *When your data is correctly formatted, it is simple to read it into a programming language, ready for advanced use by using existing tools.*

10. Writing a simple macro in Python

A macro-instruction (or macro for short) is a list of instructions that you want a computer to execute. In essence, it is a computer program. To write a macro, you must therefore use a programming language. In Excel, macros are traditionally written in Visual Basic (often abbreviated to VBA). There are several problems with putting your instructions in these macros: they can be hard to read, difficult to check for errors and logical flaws, hard to maintain, and may be difficult to run in the future. As we mentioned earlier, if you find a need to use many macros, this is a clear indication that you have reached the limits of what spreadsheets can do for your analysis, and need to move on to a more advanced toolset.

However, a good option for those who are not ready/able to change to a fully programmatic workflow can be to write your macros in Python rather than VBA. You will likely find that this is a much easier and more expressive language to work in than VBA. We will demonstrate an example of creating a macro in Python now.

We will use the tool xlwings (https://www.xlwings.org) to connect Python to a spreadsheet. You can download it using Anaconda Python by opening the Anaconda terminal (https://docs.continuum.io/ae-notebooks/user/terminal), and typing:

```
$ conda install xlwings
```

This adds the xlwings package to your Python library (the collection of packages installed on your local system which you can import from).

Imagine we have a workbook called 'myworkbook.xlsx' located on '/Users/Julia/Desktop/'. This workbook has two sheets: a sheet called 'Inputs', and a sheet called 'Outputs'. The 'Inputs' sheet has three columns, with two rows: a header row, and a row of values. We can access the workbook as follows.

First, start a Python session. Then we type:

```
In [1]: import xlwings
In [2]: xlwings.Book('/Users/Julia/Desktop/myworkbook.xlsx')
Out[2]: <Book [myworkbook.xlsx]>
```

You will note that, upon making a connection to the workbook, the spreadsheet program opens. We can setup a connection to specific sheets like this:

```
In [3]: inputsheet = xlwings.sheets[‘Inputs’]
In [4]: outputsheet = xlwings.sheets[‘Outputs’]
```

N.b. You could also reference the sheet with integers (from 0:n), indicating the order of the sheets, rather than call them by sheet name.

We can then access cell contents like this:

```
In [5]: print(inputsheet.range("A1").value)
Out[5]: sample1
In [6]: print(inputsheet.range("A2").value)
Out[6]: 5.6
```

If we wanted to, we could write another sample to the Input sheet like this:

```
In [7]: inputsheet.range("D1").value = 'sample4'
In [8]: inputsheet.range("D2").value = 4.2
```

A probable task we would wish to do is gather all the cells in a given range, perform an operation on them, and write the results to the 'Output' sheet. We could grab the data for our four samples and put them into a Python list, which i have called 'cells', like this:

```
In [9]: cells = inputsheet.range("A2:D2").value
In [10]: print(cells)
Out[10]: [5.6, 3.2, 1.1, 4.2]
```

Perhaps we wish to examine all the values in our samples, see when they are greater than 3.5, and, if they are, find the average value of the samples. We could design some simple Python code to do that task, and write the value to the spreadsheet's Output sheet like this:

```
In [9]: count = 0
In [10]: total = 0
In [11]: for cell in cells:
    ...:     if cell > 3.5:
    ...:         total = total + cell
    ...:         count = count + 1
In [12]: avg = total / count
In [13]: outputsheet.range("A1").value = 'Python Result'
In [14]: outputsheet.range("A2").value = avg
```

The above makes use of a for loop. If you are confused by this, and the meaning of the line "for cell in cells" above, then we encourage you to read about for loops at the following link: http://swcarpentry.github.io/python-novice-inflammation/02-loop/

You can save your Python instructions in files, and re-run them as you need. A range of software development practices can be applied to these files, which will ultimately make them into powerful vehicles for your scientific analysis.

Summary: *Complex spreadsheet tasks can be performed by Python.*

11. Abstraction

"No problem can be solved until it is reduced to some simple form. The changing of a vague difficulty into a specific, concrete form is a very essential element in thinking."
– J. P. Morgan (1837 - 1913)

Abstraction allows us to repeatedly and reliably solve problems. In our case, it is one of the main impetuses to keep a consist spreadsheet design, distinguishing between inputs (sheets with raw data and documentation describing these data), processing (sheets with formulas and notes describing what was done) and outputs (tables, graphs and summary values). This pattern will allow us to understand how to solve the manual workflows you are creating in your spreadsheet tools in a general way.

Identifying and separating the inputs, variables, and processing stages is the beginning of your journey towards abstracting your research problem. You will likely find that to process your data you have written a general solution that you can copy and paste to many cells to produce output. We previously introduced the idea of using Python to create macros when you have a complex task to solve, but, once you are comfortable with Python and have created a well laid-out spreadsheet it will be relatively simple to use it to code the specific solutions you implemented in your spreadsheet cells into an abstract solution. This has considerable advantages over a cell-wise implementation, as you need only write this solution in Python a single time. Once defined, you can call upon it repeatedly, varying the inputs.

For example, consider a simple function that requires you to add two numbers $f(x,y) = x + y$ in a spreadsheet, you need to find a way to write out all of the formulas directly: perhaps you may have two columns of numbers side-by-side and copy-paste something like "=$A1 + $B1", for the entire length of the columns. In Python, however, you would define your function once in the following way:

```
def f(x, y):
        return x + y
```

After, you can simply use it like this an unlimited number of times:

```
>>> f(1, 1)
2
```

Some benefits of a functional (or abstract) approach are as follows:

Explicit Solution	**Abstract Solution**
● Many implementations of same solution = many potential points of failure ● Hard to modify ● Nearly impossible to fully test	● Ideally one implementation, reused many times ● Easy to modify ● Easy to test ● Powerful and time saving

Summary: *It is more robust and powerful to solve your problems in an abstract way.*

Definition of terminology

- Spreadsheet: *An instance of a program such as LibreOffice or Microsoft Excel.*
- Workbook: *The largest unit of a running Spreadsheet file, which is comprised of one or more Sheets.*
- Sheet: *An individual page in a Workbook, made up of of a set of unique rows and columns. Sheets have names that are unique in their Workbooks, and can usually be tabbed through like a browser.*
- Cell: *The smallest unit in a Sheet, wherein text, data, or formulas may be entered.*

Acknowledgements

This work was commissioned by Prof. Suzanne Farid, and is loosely based on materials from the Data Carpentry and Software Carpentry organizations.