STRINGFEST ANALYTICS

Five things Excel users should know about Python

Contents:

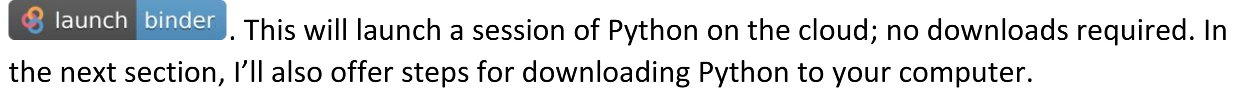
0. Getting Started ____ **2**
Companion repo ____ 2
1. Open source means a license to build and distribute ____ **3**
2. There's a package for that! ____ **5**
The Python Standard Library ____ 5
Installing packages from Anaconda and PyPI ____ 5
3. It's finnicky! ____ **6**
It's case sensitive ____ 7
Functions ____ 7
Objects ____ 7
You need to import packages to use them ____ 8
It counts from 0 ____ 9
It requires indentation ____ 11
Finnicky, or just logical? ____ 12
4. It can augment and automate Excel ____ **12**
5. It's not worth panicking over ____ **13**
Conclusion and next steps ____ **13**
Thank you ____ 14

0. Getting Started

Thank you for picking up this white paper. I hope that as an Excel user this paper both demystifies what Python can do for you and excites you to learn more.

Companion repo

All files used in this white paper are available at the GitHub repository. To follow along interactively, click the “launch binder” icon on the homepage of the repository:



. This will launch a session of Python on the cloud; no downloads required. In the next section, I'll also offer steps for downloading Python to your computer.

If, during any point of this white paper, you feel overwhelmed and that Python is too difficult for you, keep section 5 in mind: *don't panic*! You'll be in great shape for future learning by the end of this white paper, and I'll provide some additional resources in the conclusion.

Let's prance into Python!

1. Open source means a license to build and distribute

What is open source software? Yes, it's free – that's a nice touch and it significantly lowers the bar to entry. More broadly, however, open source software means that anyone is free to *use, purpose, modify and redistribute* it.

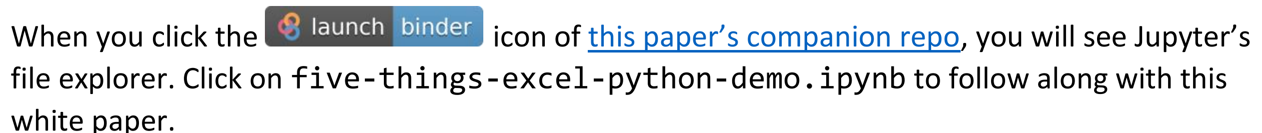
This will become apparent for Python in a couple of ways: first, untold thousands of contributors develop and share *packages* based on the source code. You'll learn more about packages in section 2.

Second, because Python is open source, anyone is free to *redistribute* it. The "official" Python code base is available from the Python Foundation at python.org.

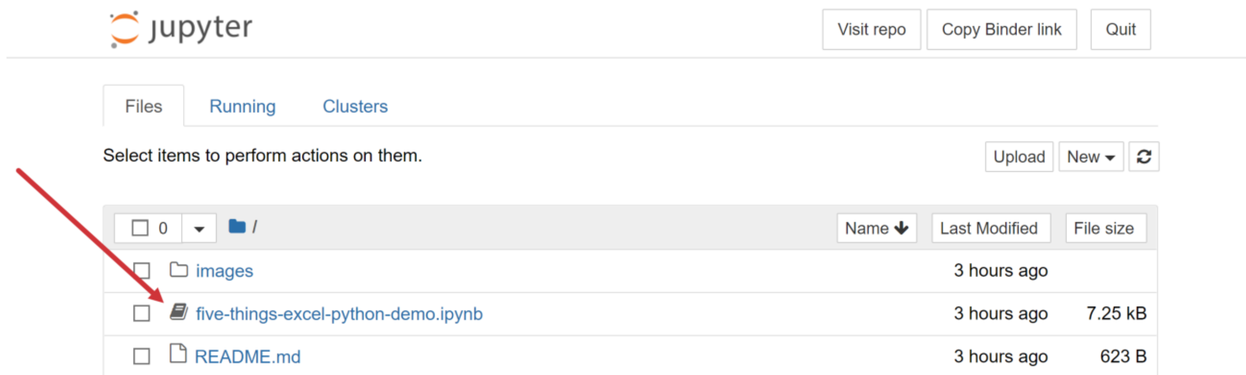
While you could download it from there, it's common especially in the data community instead to do so from Anaconda. Essentially, this for-profit company *redistributes* the Python code with various features and services included; hence, you'll often hear it referred to as the Anaconda "distribution" of Python. You can download the free Individual Edition on Anaconda's website.

To use Excel, you open a single application that contains the code base along with an end-user interface. In Python, these are decoupled so that you can run the code base from a variety of applications. We'll focus on a very common interface for working with Python: the Jupyter Notebook.

According to its parent organization Project Jupyter, the Jupyter Notebook is "an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text."

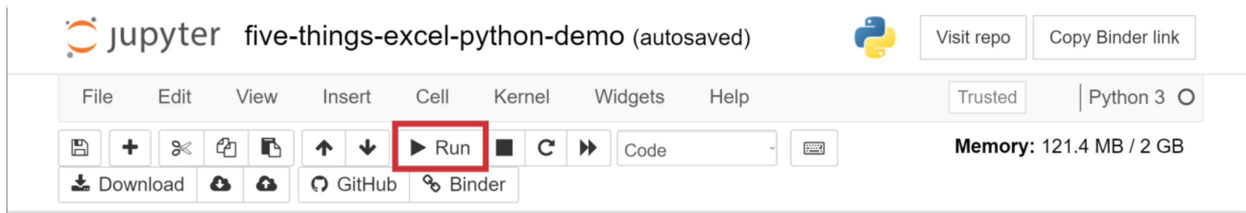
When you click the 

icon of this paper's companion repo, you will see Jupyter's file explorer. Click on `five-things-excel-python-demo.ipynb` to follow along with this white paper.



.ipynb is the extension for Jupyter Notebooks and stands for *IPython Notebook*. Jupyter was conceived as an interactive way (hence the I) to execute Python code, although it's now expanded to work with many other languages such as R, SAS and JavaScript.

Jupyter notebooks consist of a series of cells which can contain, for the most part, either Markdown text or (in this case) Python code. You can run the cells in our demo notebook by clicking "Run" on the toolbar. (And yes, there is a keyboard shortcut to run cells, along with many other commands – go to Help > Keyboard Shortcuts to find it out.)



To learn a bit more about operating in Jupyter notebooks, such as how to rename files and add text blocks, [check out this blog post](). You will quickly see why Jupyter has become such a popular interface for building beautiful, interactive data documents.

In this section you learned about what it means for Python to be open source. The next section will dig further into one feature: packages.

2. There's a package for that!

Imagine if you weren't able to download applications on your smartphone. You could make phone calls, browse the internet, and jot notes to yourself—still pretty handy. But the real power of a smartphone comes from its applications, or apps.

As an open source language, many thousands of contributors have built and shared Python *packages* which serve much like apps for smartphones. Packages are built to assist with everything from web development to artificial intelligence.

The Python Package Index (PyPI) serves as the official storehouse for these packages. How can you get your hands on them?

The Python Standard Library

Actually, before getting into these packages, let's take a moment to learn about the Standard Library. As the name suggests, these packages come with any download of Python. Learn more about the Standard Library here. While the Standard Library has some useful stuff, you'll likely need to branch further out to get far with data analysis or visualization.

Installing packages from Anaconda and PyPI

Many popular Python packages outside the Standard Library which are stored on PyPI come pre-installed with Anaconda; this is another "distribution" of code. You can check what's included for your version here. For example, `pandas` is available out of the box for most Anaconda downloads. We'll talk more about `pandas` later on in this paper.

Other packages do not come pre-installed by Anaconda, but can be downloaded from there. This can be done with the command `conda install packagename`. As a command line execution, it can be run in Jupyter with an exclamation mark.

For this example, we'll install `plotly`, a popular Python visualization language:

```
In [1]: #!conda install plotly
```

If you're following along with the demo file, you'll see that not much happens when you run this block. That's because it's been *commented out*, as indicated by the hash mark #. In other words, this bit of code has not been executed in Python. Because installing a package can take some time and is only meant to be done once, it's considered bad etiquette to leave it as code. Comments are also used to write notes providing context, assumptions and so forth.

If you're looking to download a package that's not distributed by Anaconda, you can download it easily enough directly from PyPI. The command line execution here will be `pip install` *`packagename`*. For example, let's install the `pyxlsb` package, which can be used to work with *.xlsb* Excel workbooks (currently commented out):

```
In [2]: #!pip install pyxlsb
```

Check out this post for a complete decision tree on sourcing Python packages from Anaconda versus directly from PyPI.

OK, enough about *how* to get packages – what packages should you know about? You won't get far in data analysis and visualization without those in the following table, all of which come pre-installed with Anaconda:

Package	Description
`numpy`	Designed for *numerical computing*
`pandas`	Designed to work with *panel data* and other tabular data structures (think rows and columns). This package leverages code from `numpy`.
`matplotlib`	a popular package for data visualization
`seaborn`	another package for data visualization, built on top of `matplotlib` and designed to work well with `pandas`.

However, as you will see in the next section, having a package *installed* isn't enough to use it – you'll need to import it too.

3. It's finnicky!

Python is a programming language, and every language has its features that take some getting used to. Fortunately there are sound reasons behind these features, and you will come to respect them. But coming from Excel, these in particular may feel jarring:

It's case sensitive

To learn more about case sensitivity, let's take a look at functions and objects:

Functions

Some Python look quite similar to Excel. For example, we can calculate the absolute value using the `abs()` function like so:

```
In [3]: abs(-10)
Out[3]: 10
```

However, `Abs()` or `ABS()` don't work:

```
In [4]: Abs(-10)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-eee8325e95e8> in <module>
----> 1 Abs(-10)

NameError: name 'Abs' is not defined
```

```
In [5]: ABS(-10)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-ec7b96a10197> in <module>
----> 1 ABS(-10)

NameError: name 'ABS' is not defined
```

This is because *Python is case sensitive*. In Excel, you can type `ABS()` in any case and get the absolute value. Not so Python.

Objects

Another, perhaps even more important example is objects. Rather than simply run functions in Python, you'll typically assign most data to *objects* and work from there, such as running functions on them. Objects are assigned with the equals operator, =.

For example, rather than taking the absolute value of -10 directly, we could assign -10 as an object a and then take the absolute value of a:

```
In [6]:  a = -10

         abs(a)

Out[6]:  10
```

What happens if we take the absolute value of the object A? *We get an error*, again because Python is case sensitive.

```
In [7]:  abs(A)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-503839f8ecd6> in <module>
----> 1 abs(A)

NameError: name 'A' is not defined
```

To learn more about functions and objects, check out the resources in the conclusion of this paper.

You need to import packages to use them

Excel features lots of functions right out of the box to help you work with data. So does Python, but most of them reside in outside packages. For example, a Python function `sqrt()` does exist to find the square root, similar to Excel.

But try it, and you'll get an error:

```
In [8]:  sqrt(25)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-1bf613b64533> in <module>
----> 1 sqrt(25)

NameError: name 'sqrt' is not defined
```

How come? Effectively, Python can't identify the code that defines `sqrt()` as a function. It turns out this code resides in one of the packages, or modules, of the Standard Library, `math`. (While technically a package is a way to organize modules, you'll often hear these terms used interchangeably.)

While `math` has already been downloaded as part of the Standard Library, you still need to *import* it into your session to use it – just like you'd need to open an app to use it on your phone even though you've already downloaded it.

The `import` statement is used to do this. Let's call in `math` and try the square root function again:

```
In [9]:  import math

         sqrt(25)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-230c924ba206> in <module>
      1 import math
      2 
----> 3 sqrt(25)

NameError: name 'sqrt' is not defined
```

Turns out that just calling `math` into our session wasn't enough. We need to point to it *each time* we want to use it. We'll do that by prefixing the function with `math`:

```
In [10]:  math.sqrt(25)
Out[10]:  5.0
```

It counts from 0

So far we've been working with single pieces of data in Python, such as taking the square root of one number. This is similar to working with one cell in Excel.

But what about working with a larger range of cells? This can be done in Python using one of its many *collection* data types, such as a list.

Here we'll create a list of five elements; each of them happen to be *string* or character data, but this doesn't have to be the case. We'll place our comma-separated entries inside square brackets to define it as a list, named `leaders`.

```
In [11]:  # Assign to list
          leaders = ['Barry', 'Hank', 'Babe', 'Alex', 'Albert']
```

Assigning data to an object is like putting some information into a shoebox. To see what's inside the shoebox, you can *print* the object. This can be done in Jupyter simply by running the object's name:

```
In [12]: # Print list
         leaders
Out[12]: ['Barry', 'Hank', 'Babe', 'Alex', 'Albert']
```

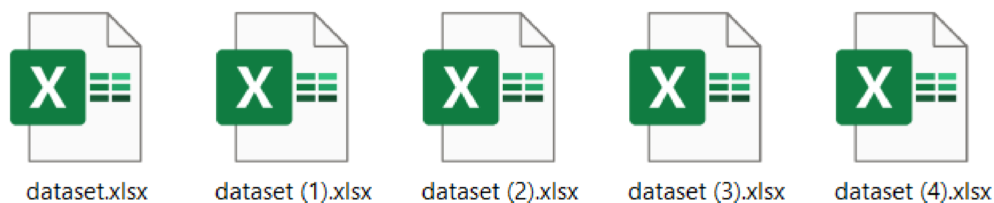
If you'd like to learn more about working with lists and other data types in Python, check out the resources in the conclusion. For now, imagine that this is for all intents and purposes a named range in Excel. If this were a named range, we could access its third element with the `INDEX()` function:

E3 =INDEX(leaders, 3)

	A	B	C	D	E	F
1	Barry	Hank	Babe	Alex	Albert	
2						
3					Babe	
4						

Indexing is also a very common procedure in Python, but it works a little differently than in Excel.

To orient you to this new way, imagine that you were so excited to get your hands on a file like this that you click download several times in haste. Doing so leaves you with a set of files like such:



In this example, our second dataset is *dataset (1).xlsx*, our third is *dataset (2).xlsx*, and so forth. In a way, we could say the first file is *dataset (0).xlsx*, although it's not stated as such. In other words, we are *counting from zero* here to index these files. Python counts this way *all the time*, and it's known more formally as *zero-based indexing.*

We can index a Python list by placing the desired position number in square brackets next to the list name. Think for a moment about what `leaders[3]` will return, then run it:

```
In [13]: leaders[3]
Out[13]: 'Alex'
```

While it seems like this would be the equivalent of Excel's =INDEX(leaders, 3), this actually pulls the *fourth* element of the list, not the third. That's zero-based indexing for you! To learn more, check out this blog post.

It requires indentation

Disoriented yet? Here's one more curveball.

Ever heard how sometimes it's not what you say that matters, but what you *don't* say? Python lives by this sentiment: not just code, but white space, can determine how code is run! Particularly, *indentation* tells Python what parts of code belong together and should be executed as such.

For an example, let's look at a loop. You may be familiar with loops if you've coded in VBA. No worries if not; just focus on the stated code error and resolution.

In this example we're assigning a list `languages`. We then want to iterate through each element of the list and make them lowercase. While in a loop, we'll need to explicitly use the `print()` function to print each element's result:

```
In [14]: languages = ['R', 'Python', 'VBA']

         for l in languages:
         print(l.lower())

  File "<ipython-input-14-52d2654bff39>", line 4
    print(l.lower())
        ^
IndentationError: expected an indented block
```

One thing to appreciate about Python is that error messages are generally intelligible. In this case, we're told explicitly here that indentation was expected at `print()`.

Let’s fix that and try again:

```
In [15]: languages = ['R', 'Python', 'VBA']

         for l in languages:
             print(l.lower())

r
python
vba
```

Bingo! Indentation can be done using either tabs or spaces, although most programmers prefer spaces.

Finnicky, or just logical?

At first, these features may feel like annoying quirks. But over time, you’ll come to appreciate the logical consistency of Python these deliver. Regardless of the packages you’re using or problems you’re trying to solve, principles like zero-based indexing and case sensitivity set consistent rules of Python and paradoxically make debugging that much easier.

4. It can augment and automate Excel

In section 2 you learned about some common Python packages for general data analysis. Here we’ll look at some packages built specifically for working with Excel.

Whether it’s importing or exporting spreadsheet data, calling VBA procedures from Python, or creating user-defined functions in Excel using Python features, these packages work to both augment and automate Excel’s capabilities.

The following table lists some standard Excel <> Python packages along with some basic pros and cons:

Package	Pros	Cons
`xlsxwriter`	Write almost anything from Excel from Python (data, formats, workbook settings, etc.)	Writes to Excel only/no reading
`xlwings`	Feature-rich: write data/UDFs, call VBA procedures, robust debugging tools	Local Python/Excel downloads needed
`openpyxl`	Read and write *.xlsx*, *.xlsm* Excel files	Limited ability to edit files
`pyxlsb`	Read and write *.xlsb* files	Limited features
`xlrd, xlwt, xlutils`	Can work with *.xls* files	Limited features

In keeping with what you learned in section 2, some of these packages come with Anaconda, some can be downloaded from there, and yet others are available from PyPI.

It can take some practice to see how all the pieces of a Python <> Excel workflow fit together. For example, it's not uncommon to pair these Excel-specific packages with ones like `pandas` and `seaborn` to build rich data analyses and visualizations in Python, then make them available in Excel. To see what a basic Python <> Excel workflow might look like, check out this blog post.

5. It's not worth panicking over

Python is a language -- of the programming variety -- and just like with any language there is *always* more to learn and consume. It can be distressing and overwhelming.

Don't panic! No one knows everything about Python, and **you should never be embarrassed about not knowing the answer to something. You *should*, however, always have a game plan to get unblocked.**

There are plenty of good resources, but it's easy to aim driftlessly across the web if you're not consistent in your Python troubleshooting workflow. Check out this post on five ways to consistently look for help in Python. While using forums like Reddit or StackOverflow are a valid way to get help, you'll need to do some homework first; the post covers these steps.

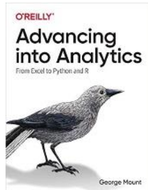
Conclusion and next steps

Congrats for reading to the end. With these five tips at your disposal, you're well on your way to becoming a true Excel-Pythonista.

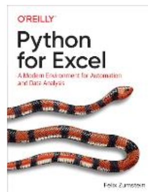
But just like anything worth pursuing, there's always more to learn. Here are a few resources for you:

PYTHON-BLOGGERS
Data science news and tutorials - contributed by Python bloggers

- *Subscribe to* python-bloggers.com. This is a daily roundup of Python posts from across the web, particularly for data users.



- Read *Advancing into Analytics: From Excel to Python and R* by George Mount (*sigh*… yes, that guy.). You will learn more about Jupyter notebooks, how to manipulate and visualize data, and conduct hypothesis testing.



- Read *Python for Excel: A Modern Environment for Automation and Data Analysis* by Felix Zumstein. Here, you will learn more intermediate Python programming techniques and how to fully automate Excel.

Thank you

Thanks for picking up this white paper as a guide for including Python along with Excel in your data toolkit.

I invite you to continue reading my newsletter and perusing my blog for more analytics content. You're also welcome to get in touch if I can help your organization with the services listed here.

One last thing: please give me a follow on Twitter and LinkedIn. Your support is appreciated!