Introduction to Python

Víctor Sojo (vsojo@amnh.org)

Contents

I. A few words on formatting before we start

Throughout this document, I will use a few types of formatting. Normal text like this is just me rambling on insufferably, and **you can ignore most of this black-and-white text if you wish— things should mostly work**. You probably won't understand much of what you're doing, though, so I wouldn't ignore this black-and-white text if I were you, but you can if you wish.

Secondly, I will have to-the-point text for very specific instructions that you should follow:

1. Follow the instructions in these blocks closely.

You can essentially just follow those blocks of grey text and you'd be okay…ish.

These specific instructions will be numbered continuously throughout this document, so that we can always track how far along we are in case someone gets stuck.

Code blocks will be pretty obvious—in monospace font and coloured in a funky way:

```python
DEADLINE = 23
happy = True
day = 24


# am I happy and is this day before the deadline?
if happy and day <= DEADLINE:
    print("Hello World!")
else:
    # Shoot, I either missed the deadline, or I'm unhappy :-(
    print("Good bye cruel world, I'm leaving you today." + 3*" Good bye...")
    # ...extra points if you can tell where that text above came from.
```

I emphatically recommend that you **do not copy and paste my code into your editor**. Instead, **type the code yourself from scratch**. Apparently, neurologists say that this should improve your learning process 🙄. In fact, if you want to get really serious about this, write code on paper!

I may occasionally panic and give you a stark warning, such as:

You can ignore most black & white text, but please **do not** ignore these blocks or the grey ones!

And with that, let's get started.

II. Installing Python and setting up

To write and run code in Python, we would generally need three things:

- The Python 3.7 engine itself, i.e. the software that interprets Python code.
- A text editor to write the code in.
- A place to run this code using the engine.

The latter two tasks are typically combined into something called an **Integrated Development Environment (IDE)**, which is some sort of fancy editor in which you can both write your code and run it (as well as "debug" it, i.e. use some mighty tools to trace errors in your code).

Actually, if you're on a Mac or Linux machine, chances are you already have Python. You could go and use the basic text editor app that came pre-installed on your machine to write your code (you can't use MS Word because you need "plain text"), and you could run your code directly from the Terminal or Console. However, this scenario isn't necessarily the most convenient.

Alternatively, you *could* (but shouldn't) go to the language's website itself (www.python.org) and download Python, which comes with its own IDE. However, please **don't download Python from their website**. There is a far better way of doing all this: Anaconda.

Don't get Python from python.org—we'll use Anaconda instead.

II.1.   Downloading Anaconda

We will download Python v3.7 through Anaconda, a super useful software that contains Python and a bunch of tools already included, most notably a powerful package manager and two separate IDEs (Jupyter Notebook and Spyder).

2.   Go to www.anaconda.com and follow the links to **Download** Anaconda for your platform. Make sure to download the **3.7 version**.

Install and so on, you know what to do…

Get **3.7! Do not** get the 2.7 version. Python 2.x has been formally discontinued.

If you're on Windows, I strongly suggest that you follow the recommendation of Anaconda and **install only into your local user account** (not for all users). It will make your life easier.

Let's make sure Anaconda is working:

3.   From your apps, launch the **Anaconda Navigator**, which must be installed now if everything went well.

Easiest way to do this: on Windows, push the Windows key, and on Mac press Cmd+Space, then start typing "Anac…" and it should appear. You should see a screen like this:
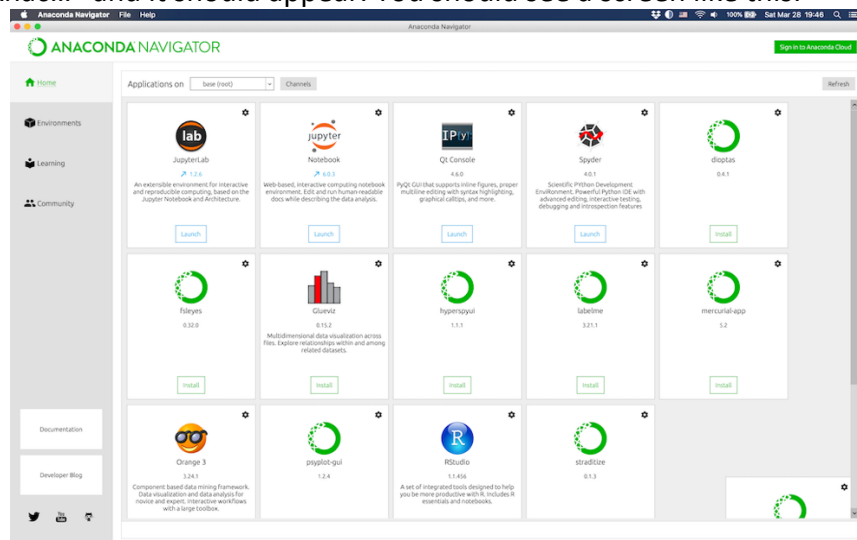
II.2.   Creating an Anaconda environment

Great, we have Anaconda! Now we want to create something called an "environment", which is our own little Python world with its own Python version and packages (like plugins and apps). You can make many of these. This way you can have a Python 2.7 environment if you inherited old code from the previous PostDoc in your lab, but you can still keep a separate environment with your shiny Python 3.7 (or whatever is cool when you read this document). Actually, Python 3.8 has already been out there for a while, but at the time of writing (April 2020) most scientific computing tools, including Anaconda itself, are still optimised for 3.7, so let's stick to that.

4.   Click on "Environments" on the left.

There should already be an environment there called base (root). That's the base Anaconda environment and it works perfectly fine right out of the box. Some people just work there and don't bother creating an environment. That's ok, but it's much better to have environments. Why? Well, imagine you want to try your code on Python 3.8 to see if the packages that you are

using already work there. Instead of messing up your nicely working 3.7 environment, you just create a new environment with 3.8, try it, and if the code doesn't work you can just go back to your tried-and-tested 3.7 until somebody convinces you that 3.8 should now work.

5.   Click "Create" at the bottom.
6.   Enter `pyworkshop` as the Name.
7.   In the "Packages" section, check Python and select version 3.7. Don't select R.

Make absolutely sure to select **Python 3.7** if you're given multiple options.

8.   Click "Create". This should take roughly about a minute or so.

If Anaconda Navigator is behaving strangely and not letting you choose a Python version or click "Create", just restart it. You may also have to restart your computer.
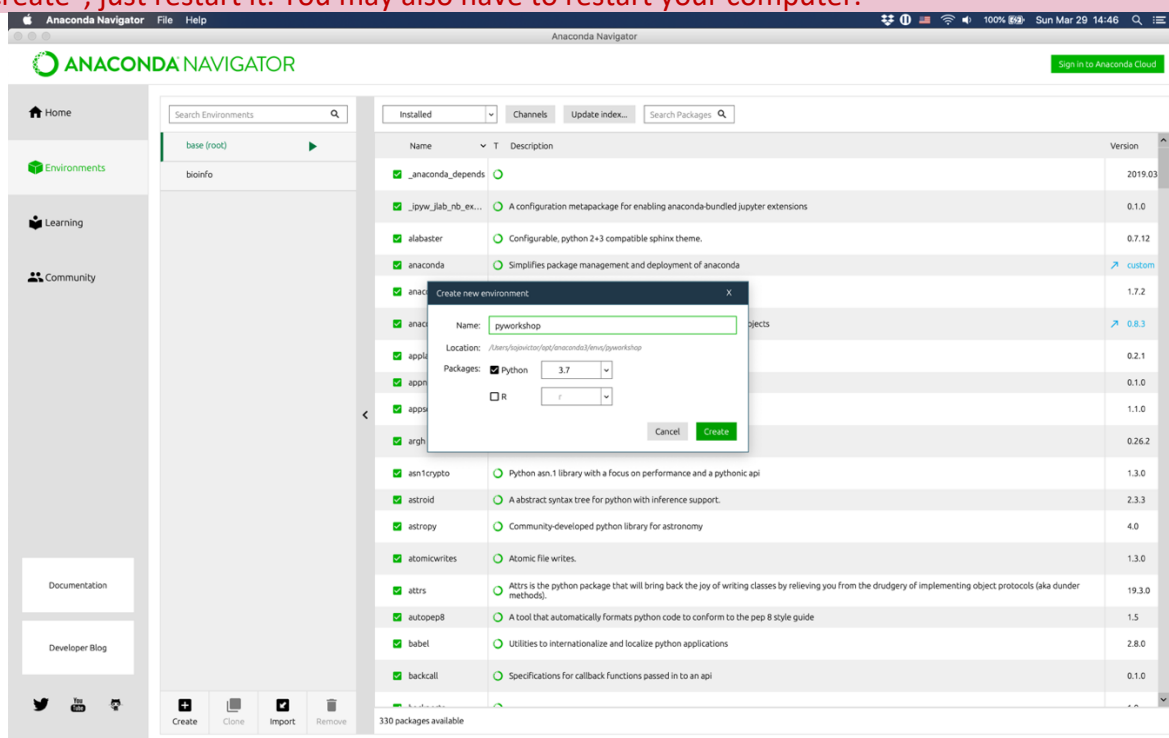
You'll see in the image above that I already have another environment there called `bioinfo`, where I do essentially all of my coding. That works well for me because all my projects have similar requirements, but some people have many environments for different purposes.

You will have noticed that you could have added R as well as Python to your new environment. Not only can you write both R and Python code in your Anaconda environment, you can actually write them together in the same document! We'll probably talk about that very briefly later in the workshop.

II.3.   Adding packages to your Anaconda environment

I asked you not to add R when you created your environment. That's because we can always add it in later. In fact, **Anaconda environments are highly—and robustly—customisable**, which is the main reason Anaconda has become the standard for most scientific computing.

Incidentally, the best place to write R code together with Python code is the **Jupyter Notebook**, a great browser-based tool for code development that makes your projects much more reproducible and easier to share. Let's install this package into our environment.

II.3.1.   Adding channels to your Anaconda

But before we add packages to our environment, a bit of housekeeping. We want to have the appropriate "Channels" set up. These channels are the online repositories where Anaconda will look for new software for our environments.

9.   Make sure the `pyworkshop` environment is selected (it should have a green triangle).
10. Click on "Channels" at the top.
11. Click on "Add…", type `bioconda` and hit Enter/Return.
12. Now add `conda-forge` the same way.
13. Click "Update channels".

Your channels should now look something like this:
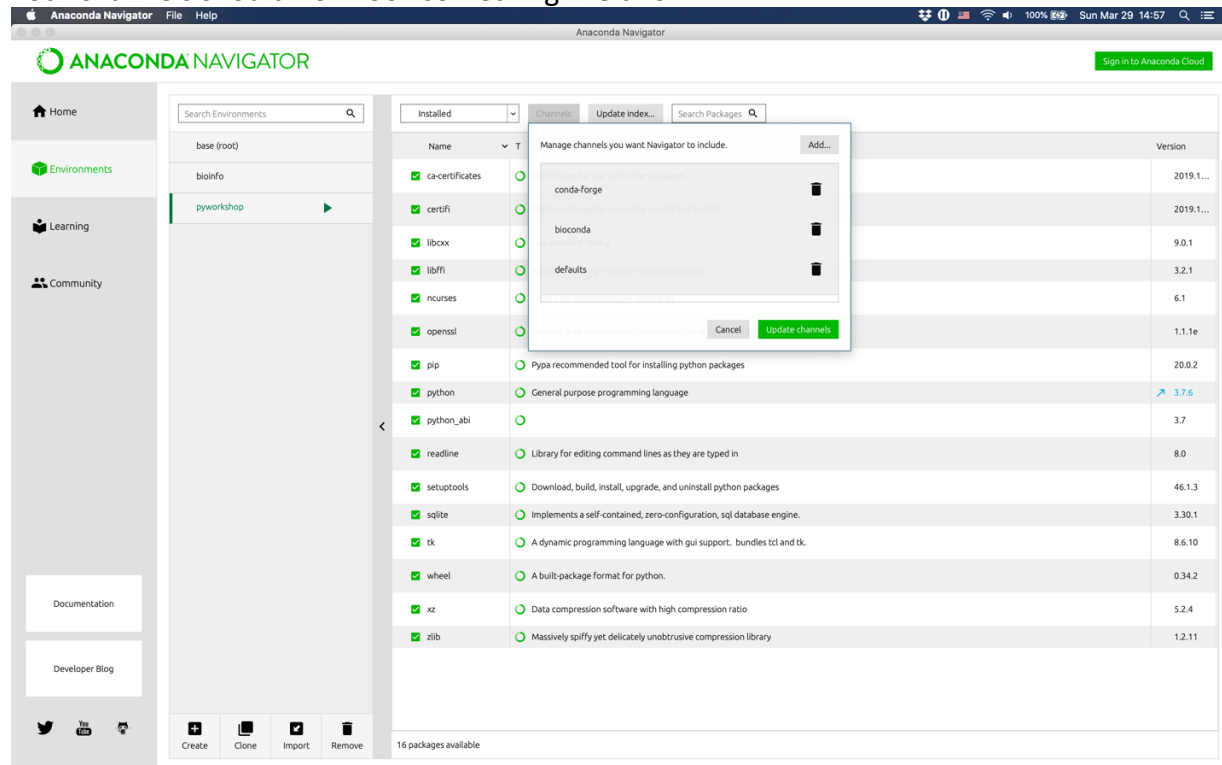
II.4.   Installing Jupyter Notebooks

Ok, now we can go ahead and add in Jupyter Notebooks to our `pyworkshop` environment.

14. Change from "Installed" to "Not installed" in the drop-down list at the top.
15. In the search box, look up `jupyter`

Yes, "jupyter" with a "y"!

There's a ton of options, make sure to choose the one that just says "`jupyter`" and nothing else (this one brings a lot of nice plug-ins with it):
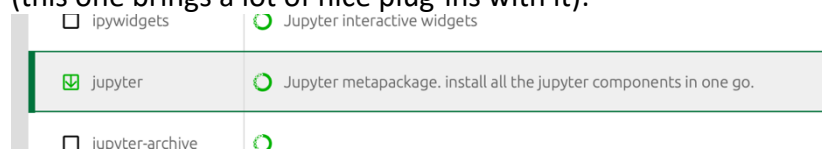
16. Activate the checkbox for `jupyter` and click "Apply" in the bottom right.

Anaconda can be buggy, heavy and annoying at times, but here is why it is so cool: It goes and checks for all the requirements of jupyter, figures them out, resolves any conflicts with the

packages that you already have installed, and it may even *downgrade* any conflicting packages as necessary so that all of your code still works. It's wonderful:
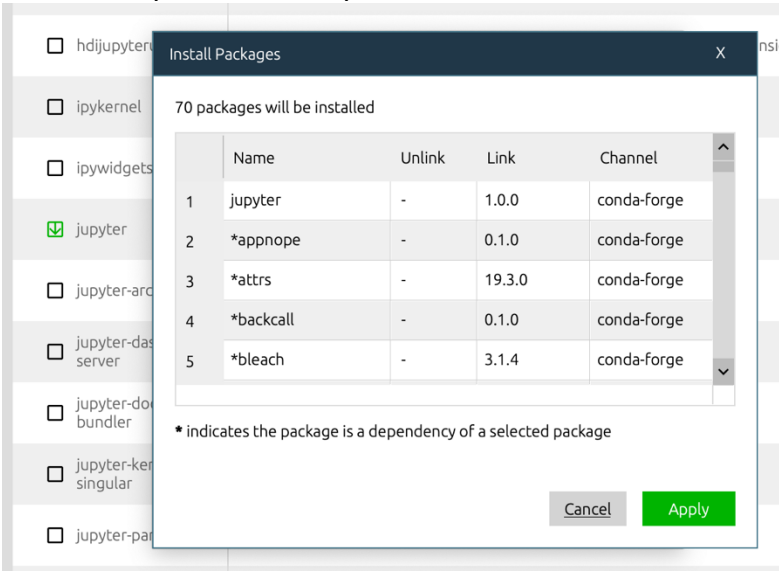
17. Go ahead and click "Apply" inside the pop-up box.

Anaconda will now download, install and update or downgrade any necessary packages. This may take a few minutes, depending on your connection. Once it's done, jupyter will disappear from the list of not-installed software. You should see it if you change back to "Installed".

Better still, let's see if it works!

18. Click on "Home" at the top left.
19. Make sure `pyworkshop` is still the selected environment.
20. You should see a box for Jupyter Notebook with a "Launch" button. Click it.

If Jupyter Notebook fails to start, you may have an **ad blocker** in your browser. Read on:

You can try to disable the blocker or temporarily change your default browser to one that doesn't have a blocker. More elegant still:

21. To start Jupyter Notebook without the Anaconda Navigator,
     Mac/Linux: open the **Terminal** (a.k.a. Console). **Windows:** hit the Windows button and start typing "**Anaconda Prompt**" until you see it offered; select it.
22. Once in there, type the following, hitting enter after each line:

```
conda activate pyworkshop
jupyter notebook
```

The first line changes your environment to the one we created for the workshop. The second line should start a browser with the Jupyter notebook. This not only is a workaround to the ad-blocker problem, it's actually how most coders start their jupyter sessions (or so I like to think because that's how I do it, albeit using a nifty "alias"; ask me if you want to know). It also prevents you from having to start the heavy and slow Anaconda Navigator.

II.4.1. Testing that Jupyter is working

Let's assume it all went well. You should see a browser window pop up, and it will probably show you your computer's file structure. Don't be scared, you're not exposing yourself to hackers on the internet. This is your own data being loaded and shown *only* in your own computer; nobody else can see that. That's why the http address says `localhost`.

23. Create a New… > Notebook > Python 3
24. Inside the textbox, type:

```python
print("Hello World!")
```

25. With the cursor still inside the cell, click the "Run" button at the top, or hit Shift+Enter.
The computer should tell you "Hello World!". And that's it, we've got a running Python and IDE!
If you're really bored, and **just this one time**, copy and paste the code below Instruction #1 at
the beginning of this guide, and try to get it to run. Otherwise, I emphatically recommend:

Do not copy-paste my code into your editor. Type it yourself; it will pay out in the long run.

You'll probably get an IndentationError, so the code will not run. If you do encounter this,
compare your code with mine and see if it pasted appropriately or if something seems to be off.
That's it for now. Let's close Jupyter, we'll return to it later:

26. Close the browser window with the code we just created.

No need to save anything (actually, it saves itself, even if you don't want it to).

27. In the main Jupyter window, click "Quit" to shut down the Jupyter server.

The Jupyter server is the software running behind the scene that is making your computer behave
like a pretend internet server. You don't need to understand any of it for now; just remember
that if you switched the server off, you'll have to launch it again if you want to keep coding in
Jupyter. If you click on any link to open a notebook again, it should fail. Feel free to close the
browser window now.

II.5.   Installing Spyder

We also want to install Spyder, which is a more traditional integrated development environment.

28. Add spyder to your pyworkshop environment just like we added jupyter.

Many options again. Add the one with just "spyder" and nothing else.

29. Go to your Anaconda Navigator, make sure your pyworkshop environment is
selected, and click "Home".
30. Find Spyder and start it by clicking the "Launch" button.
31. **Alternatively** to 29&30, or for future reference so that you can start Spyder without
the Navigator: open a Terminal/Console/Prompt (see #21), change to the
pyworkshop environment (enter "conda activate pyworkshop"), then enter
"spyder".
32. There should be a "Console" on Spyder's bottom-right side. Click inside it and type:

```python
print(1 + 2)
```

That's it. We're ready to go!

II.6.   Updating your Anaconda installation (semi-optional)

It is generally wise to keep your Anaconda up to date. You should do so immediately after installation so that you have all the current packages. The instructions below are valid at the time of writing (April 2020), but I recommend that you check what's current when you read this.

33. Start a _new_ **Terminal** window (**Mac**) or **Anaconda Prompt** (**Win**). See how above in instruction #21.

34. Make sure it says (base) at the beginning of the prompt line. If it doesn't, you need to change the environment:

```
conda activate base
```

35. Execute the following command to update the base environment:

```
conda update --all
```

You will be shown the list of packages that will be downloaded, updated, downgraded, superseded and so on. Proceed if you agree.

This will probably take a while... ...done? Great! But...

This did not update your other environments! You need to update those one by one. So:

36. Let's update the pyworkshop environment also from within the same window. You don't need to open a new window, you just have to change the environment:

```
conda activate pyworkshop
conda update -n pyworkshop --all
```

II.7.   Other IDEs (optional)

There are probably hundreds of other IDEs (fancy code editors/runners/debuggers). Two worthy of mention are **Visual Studio Code** (code.visualstudio.com) and **PyCharm Community Edition (CE)** (jetbrains.com/pycharm, make sure to follow the link to "Community", not "Professional"). Both are free and open source, and both are great. We won't really need them here because we already have two separate IDEs (Spyder and Jupyter Notebook), but you may want to consider them for the future work if you become a formal Python coder. Spyder is similar to VS Code and PyCharm, so if you end up liking Spyder over Jupyter in this workshop, I encourage you to try the other two also and see which one suits you better (warning: PyCharm has a bit of a learning curve). In any case, other than being light yet powerful, VS Code lets you easily copy and paste nicely formatted code into Word (which incidentally is how I did the beautifully coloured code snippets in this document). This isn't as easily done with the other two that we have installed. **For any other intents and purposes in this workshop, Jupyter and Sypder are perfect**.

VS code and PyCharm are great to code in in general but we won't be using them here, so it's your choice whether you install them or not.

All of that said, **Jupyter Notebooks are quickly becoming a (if not _the_) standard for sharing computational-science protocols, at least for Python code**, so please do try to learn as much Jupyter as you can (there's plenty of help online, and I will aim to show you some of it here if there's time).

III. Mathematical operators

37. Take a look at the list of basic mathematical operators in Python:

Mathematical operators		Example	Result
Addition	+	2 + (−5)	−3
Subtraction	−	−7 + 9	2
Multiplication	*	3 * 2	6
"Floating-point" (real) division	/	6 / 2	3.0
Integer division **quotient**	//	13 // 5	2
Integer division **remainder** or "modulo"	%	13 % 5	3
Exponentiation (a^b)	**	2 ** 3	8

Operation precedence works just like it did in middle school. If in doubt, just throw in a pair of parentheses, even if you think it's unnecessary but will make the code easier to read. Don't overdo it, though—try to keep your code readable, clean and pretty, in that order.

38. Go ahead and try all the examples in the table above in Spyder's Python console.

Note: here, Python won't care if you leave spaces or not. For other things that we will see later, Python is *extremely* nitpicky with spaces. I'll tell you when it becomes relevant.

You'll see that the **Python console** gives you the answer as screen output. It essentially behaves as a calculator.

There are some funny things happening, though. If you try 3*2 you get the integer 6, but if you try to reverse it with 6/2 you get 3.0, with a decimal point. That's because Python 3.x (but not Python 2.x), defaults to floating-point (float) division, i.e. it treats all numbers as ℝeal and not integers. Conversely, multiplication defaults to integer (int) if all numbers being multiplied are integers, but float if at least one of them is a float (i.e. has decimals, even if the decimal is zero). You'll see in the table that you can actually get integer division by using a double //, but be careful:

// only gives you the exact value if the first integer is divisible by the second one (e.g. 4//2).

39. Try each of the following exercises, but **predict their output before you run them**.

| 6*3 | 6/3 | 6//4 | 6%4 | (50−5*6)/4 | 3**2 | 9**(1/2) | 9**(1//2) |

IV. The `print()` statement: the programmer's best friend

40. Let's do some of the same code above, but wrapped around a `print()` statement:

```python
print(2 + 4)
print((50 − 5*6) / 4)
print(9**(1/2))
```

So far so good, we get essentially the same result as without the print statement. So why would we want to use `print()` at all? Well…

41. Try this:

```python
print("I am", 3**2 − 7 + 38//3, "years old")
```

Pretty cool, innit? We will see more reasons to use `print()`, chiefly that when we write proper **script files**, Python won't print anything unless we actively tell it to.

The print statement is arguably the programmer's best friend. It is often used as a quick and cheap way of **debugging**, i.e. finding errors in code. There are fancier ways of debugging, but we

Víctor Sojo (vsojo@amnh.org)   10

won't be covering them here. I encourage you to read about it online if you get serious about coding. I'll limit myself to giving you the best advice I've ever read on the topic:

When debugging, don't try to figure out why your code ***isn't*** doing what you want it to do. Instead, try to understand why it ***is*** doing what it's doing.

V. Don't ever use the console, ever again – write scripts instead!

42. Let's make a deliberate error. Type and run the following in the Python console:

↓

```python
print("I am", 3**2 – 7 + 38//3, "years old)
```

↑

Yes, it's the same code as before but leaving out the closing double-quote before the parenthesis.

Python will complain, and it will try to help us figure out what we did wrong. In this case the error message is not particularly helpful unless you already know a bit of coding:

`SyntaxError:` `EOL while scanning string literal`

What on Earth could that mean?

EOL means "End Of Line". It seems Python was scanning a "string literal" (whatever that may mean) and it reached the end of the line that we told it to execute before this "string literal" thing ended. Since we know what we broke, we can deduce what the message means: the string literal was the text, and Python reached the end of the line without understanding what was going on because we never told it where the text ended.

Annoyingly, we need to type or at best copy and paste the code again, correct the error, and run it again.

There's a trick, though. In most consoles, **you can use the up arrow in your keyboard to recover the preceding commands**.

43. Use the up arrow in your keyboard to recover the previous command and correct it.

But every time we run a line of code, it goes into the console and after a while we lose it. Sure, we can scroll up or hit the up arrow and recover it, but what if we want to build a more complex program? Writing and testing it line by line would be a real pain.

Instead, we write a **script**. A script is a text file with code that gets interpreted by the computer only when you tell it to, not every time you hit enter like in the console.

Don't ever use the console again, for anything other than some extremely basic Python command. If you want to use a calculator, just use the one that came with your operating system.

A script is just a text file. You can send it to your collaborators, share it with the community, update it, and so on. And you can keep all your code there and later delete anything that you don't want. Conversely, trying to get code out of the console can be very annoying and sometimes impossible.

So, if Spyder doesn't already have a text file open on the left, hit **File > New File** or **Cmd+N** on Mac, Ctrl+N on Windows/Linux.

44. Type or copy and paste some of your old code into the script in Spyder.
45. To run it, do not hit Enter, but **Ctrl+Enter**. Enter just adds a new line, like in MS Word. You can also hit the green "play" button at the top, near the left side.

You will see the output of your command on the console.

And that's it, we won't be using the console again here except to examine the output of our code.

VI. # Comment your darn code!

Comments are extremely important, and we will use them very generously here. They make your code readable—by you and by others—and they help you think about the code you're writing and why. Often while writing comments you will see that the code you wanted to write doesn't make sense or isn't necessary. I therefore recommend that you:

Write lots of comments and, as much as possible, write them **before** you even write the code that you are commenting!

Almost all programming languages have a way of adding comments. In Python there are two main ways: single-line comments, and multi-line comments.

VI.1.   # Single-line comments

Created using the "hash", "numeral" or "number" sign # (for some strange reason called "pound" in some parts of North America). The convention is to leave a space after the #:

```
# like this, with a space after the hash
#not like this (this is ugly and confusing, see next)
```

The space is not obligatory, but it is strongly advisable because we normally **do not leave a space when we're "commenting code out"**, i.e. when we have a line of code that we want to disable temporarily:

```
print("This code is fine.")
#print("But I don't want this code to run")
```

VI.2.   """Multi-line comments"""

These are created between triple double-quotes:

```
"""
This is a multi-line comment
I can write whatever I want here...
#%#%%@^^!(!*!%!% Fja jas ;ljfl jdsfoae ou234@$@#$!$$! Absurd nonsensefd f dj jl!
print("...and none of this will run")
x = "because it's in a multi-line comment")
print(x, "so Python will ignore all of it!") BWAhahaha
"""
```

These comments can be very useful for temporarily commenting out large blocks of code, but other than that:

Refrain from using multi-line comments except to disable code temporarily. Triple-double-quote comments are typically reserved to describe functions and other things. We'll see how later.

In fact, you may have noticed that Spyder added a multi-line comment at the top of our script file. We're supposed to write into it what this file is supposed to do, who made it, when, for whom, and so on.

VII.   Variables

Like in mathematics, **variables** in programming languages are symbols that represent values— but they can also represent other things. In Python, we use the equals sign = to **assign** a **value** to a **variable**.

Yes, equals = is used to **assign values to a variable**, not to compare things. For comparisons we use the double equals ==, as we will see later.

46. Predict the output of the following script, then write and run it:

```
width = 10
length = 2
height = 5
volume = width * length * height
```

…actually, if you're in Spyder, this script doesn't give you any output at all. Can you guess why? The reason is that we just told Python to multiply those three things, and it did it; fine. If we want to see the value, we have to tell Python to show it to us:

```
print(volume)
```

And now it does show it. In case you're wondering what would happen if you just throw volume on its own, i.e. removing the print statement:

```
volume
```

Go ahead try it. You'll get absolutely nothing. The reason is that computers respond only to commands, and you didn't give it any. If you tell it "volume", the computer just thinks "aha, volume, it's worth 100, sure thing, I know what you mean". You didn't really tell it to do anything with that number, so it just looks at it, and then it just keeps going.

There's a caveat: if you try typing "volume" directly in the console instead, or in Jupyter, you'll see that it does actually print the result. This is because the console is meant as an interactive environment for immediate coding, so it was built to assume that you always want to see the value of a variable, even if you don't ask to print it. Once again: the console is not really meant to create lasting code, only for short tests. The same happens in Jupyter, but that's because the Notebooks were created for visualisation, so they show as much as possible.

One other thing is worth noting in the code above. If you try printing any of the three variables, you'll see that their values didn't change. So:

```
x = 1
print(x + 1)
print(x) # still the same original value
x = x + 1
print(x) # now it did change
# challenge: try x += 1
```

VII.1. Naming variables

Variable names must start with an English-language letter or underscore, and other than upper and lowercase letters, they can have numbers or underscores within. **No dashes or spaces**.

You don't have to, but please **always start your variable names with lowercase letters**. Starting uppercase letters are used for something else (Classes, if you must know). **Python is case-sensitive**, i.e. the variables pYthon and pyThon are two different things. Needless to say, distinguishing your variables only by casing, although technically allowed, is a horrendous coding practice, so don't do it. In any case (forgive the pun), the convention in Python is to write every name in lowercase unless you have a compelling reason to do otherwise.

Constants, however, are normally written in all uppercase: `AVOGADRO = 6.022e23`

If you have a long variable name, `somePeopleUseCamelCasing`, whereas others `prefer_to_use_snake_casing`. The convention in Python is to **use** `snake_casing` **for most things**. `CamelCasing` is used only when defining classes in object-oriented programming.

VII.1.1. "Reserved" (i.e. forbidden) names

You can't use any name that Python is already using, such as `print`, `if`, `else`, `for`, `while`, `and`, `or`, and many others. The colour in your IDE often helps you a bit. Here's a sample from VS Code:

```
name = "NYC"
desc = "awesome"
type = "metropolis" # hm... maybe we need to use "kind" instead of "type"
```

VII.1.2. Using descriptive names

By all means, **give descriptive names to your variables**, but do try to keep them relatively short. I tend to prefer longer names like `population_density` far better than a confusing `p`, but some people prefer shorter code. Either way, **comment your code thoroughly** to indicate what each variable represents, even if you think it's obvious—it may not be obvious to everybody:

```
# Population density of New York City:
popdens_nyc = 10947 # persons/km2
```

Now whoever looks at our code knows that we're using persons/km^2, as opposed to persons/mi^2. Also, this:

```
popdens_bronx = 13231
popdens_brooklyn = 14649
popdens_manhattan = 27826
popdens_queens = 8354
popdens_staten_island = 3132
```

is much easier to read than this:

```
bronx_popdens = 13231
brooklyn_popdens = 14649
manhattan_popdens = 27826
queens_popdens = 8,354
staten_island_popdens = 3132
```

All in all, please choose variable names that anybody can identify immediately if possible, even if they're completely unfamiliar with your code, or even with programming.

VII.2. Assigning multiple variables at the same time

Python is built for efficiency. You can assign multiple variables in one go. You just have to give a list of desired variables and their values.

47. Type and run the following (but predict the output before you run it!):

```
a, b, c = 1, 2, 3
print(c, a)
```

This is perhaps a little advanced and you may never need it, but the numbers don't really have to match, you can just make one of the variables a container for "everything else":

```
a, *b, c = 1, 2, 3, 4, 5, 6
print(a)
print(b)
print(c)
```

So, a takes the first value, c takes the last one, and since we gave b an asterisk, it takes everything else. Other than this exception, the numbers of variables and values typically must match.

The following will give an error:

```
a, b, c = 1, 2, 3, 4, 5, 6 # THIS WILL FAIL!
```

48. But… you can actually define multiple variables to the same value. Try the following:

```
x = y = z = –5
print(x, y, z)
```

VII.3. CHALLENGE: reassigning variables

49. Define two variables, x = 4 and y = 5. Now, **without** writing 4 or 5 again, make x have the original value of y (i.e. 5), and y the original value of x (i.e. 4).

Bonus points if you can do it in one line, but the traditional solution would have three.

VIII.   Data types

50. Try the following:

```
print("Mary had " + "one" + " little lamb")
```

Cool. You can join bits of text using the + operator. This is very worth remembering. If you care to know, the process of joining text is almost always called **concatenation** in computer science.

51. Now try the following:

```
print("Mary had " + 1 + " little lamb") # will fail!
```

You'll get an interesting and very informative error. It tells you that you can only concatenate a str to another str, not an int. "str" is short for string, which is what computer people call text (think of text as being a *string* of characters, one after the other). "int" of course is short for integer. Essentially, you can't tell Python to just put these two things together, it doesn't understand what you mean. There are a few tricks you can use to solve this problem. The first:

```
print("Mary had", 1, "little lamb") # Good trick to remember: use comma instead of +
```

This will work, because the print function internally handles any **type conversion** necessary. It receives three separate bits, handles each of them independently making sure that they are all converted to string, and then concatenates them. But we can do the type conversion ourselves.

VIII.1. Type conversion

52. Use the str() function to convert an integer to string:

```
print("Mary had " + str(1) + " little lamb") # will fail!
```

Nice! Now it works.

So, how many data types are there? Well, many. But here are the main ones:

53. Use the type() function to find out the main types of data in Python

```
type(1) # int
type(1.0) # float
type(True) # bool
type("número uno") # str
```

You can run `type()` on any variable in Python, so it's a good function to remember.

Try to remember those **four main data types:** `int`, `float`, `bool`, and `str`.

You'll be using them quite a bit whenever you write Python code.

Each of those types has its own type conversion or type casting function, called the same as the type itself, so…

VIII.2. CHALLENGE: type casting (a.k.a. type conversion, or type parsing)

54. Using either **Jupyter** or the **Spyder console**, predict the output of the following code snippets, **including quotes ' ' and decimals**, if you think there will be any.

```python
float(1)
int(2.0)
int(2.9999)
str(2)
str(2) + str(3)
str(2) + float(3)
int(str(2) + str(float(3)))
int(True) + float(False)
```

VIII.3. Double quotes or single quotes?

Sometimes you'll see people write Python strings `"like this"`, but sometimes `'like this'`. Briefly, both are correct, but please use the double ones. There are two reasons for this. The first reason is that with double quotes you can easily write English-language strings such as `"He ain't heavy, he's my brother"`. But there's another good reason to use double quotes for most of your strings, which I'll explain in the following sections (in short, we'll use them as the *index* or *key* of dictionaries).

IX. `list`, `dict`, `tuple`: Data structures

IX.1. Lists

Lists are collections of data, *typically* the same kind of data.

55. Create a list:

```python
fruits = ["apple", "orange", "pineapple", "passion fruit", "coconut", "kiwi"]
```

Try printing the list with `print(fruits)`. Nice. But what's coolest about lists is that you can do a lot of stuff with their individual members.

56. Let's access an element inside a list:

```python
print(fruits[1])
```

Great! Or… hang on a sec… the 1st element was not the orange! The geekiest amongst you will know what's going on:

Many computer languages (**including Python**, but **not R**), start counting at zero!

This is crucial, and it's a major cause of headaches for beginning programmers, and for those of us who frequently have to switch between Python and R.

Anyhow, we can access elements inside a list by giving their index (remembering to subtract 1).

57. Cooler still, we can also access items backwards by giving a negative index:

```python
print(fruits[-1]) # the last fruit (kiwi)
```

Ok, here the first index is actually -1 (not -0, which would make no sense)… just another thing to remember. The second one from the back is -2 and so on.

```python
print(fruits[-2]) # the coconut
```

IX.1.1.   Modifying values in a list

58. You can just change the value of any item in a list:

```python
print(fruits)
fruits[2] = "tamarillo"
print(fruits)
```

IX.1.2.   *Slicing* and *stepping* in a list

You can get "slices" of a list by using a colon:

59. Slice a list by using a colon, as in: `my_list[start:stop]`

```python
print(fruits[2:4]) # This doesn't change the list itself, though
```

Nice! Oh… hang on again… index 4 is actually the 5th element, isn't it? So, that should have been the coconut! Why did we get only up to the previous one? Well… that's something you'll just have to learn to live with: when you're slicing or sub-setting a list (or tuple, or string), Python gets you up to the item *before* the one you gave it as stopping point.

60. You can slice backwards too:

```python
print(fruits[-4:-2]) # Going backwards, Python also stops one earlier than you'd think
print(fruits[-4:-5]) # Nothing. That's an empty set. It's similar to doing this:
print(fruits[5:4]) # A nonsensical slice that will give you nothing, an empty set
```

61. You can indicate "everything else" by just leaving either of the two indices empty:

```python
print(fruits[:4]) # start at the beginning, go until the 4th item
print(fruits[3:]) # start at the 4th item, go until the end
print(fruits[-3:]) # start at the antepenultimate item, go until the end
print(fruits[:]) # start at the beginning, go until the end, i.e., the whole list
```

Two weird things about Python indices to remember:

- Start counting at 0 if going forwards, but at -1 if going backwards.

- Sub-setting works until the number *before* the one you were aiming for.

This applies to strings, lists, tuples, and more.

62. Jump by specifying a step: `my_list[start:stop:step]`

```python
print(fruits[:4:2]) # start at the beginning, go until the 5th item, but skip 2 by 2
print(fruits[::2]) # the whole list, but leaving every other item out
print(fruits[::-1]) # the whole list skipping backwards by 1... so, the reversed list!
```

Mind you, that last one is not the most efficient way to reverse a list, but coders sometimes use it because it makes them look cool, I suppose. We'll see the best way to reverse a string in the next section.

IX.1.3.   Some very useful list functions

Note: many of the functions below also work for `string`s and `tuple`s.

63. Create the following list

```python
squares = [0, 1, 4, 9, 25, 36, 49, 64] # the squares from 0 to 8
print(squares)
```

64. Use the `len()` function to get the length of a list

```python
print(len(squares))
```

Hm… the length is 8, but it should have been 9. Oh, I see what I did wrong, I forgot to add the square of 4 (i.e. 16). We can use the `insert` method for this.

65. Use `.insert(index, object_to_add)` to add items to a list:

```python
squares.insert(4, 16) # insert 16 at the 5th position (remember we start from zero)
print(squares)
```

66. Use `.pop()` to remove items from a list and get the item removed at the same time:

```python
squares.pop() # removes the last item
print(squares)
x = squares.pop(0) # removes the 0th (first) item. You can also use negative indices
print(squares)
print(x)
```

As you can see, pop does not just remove an item, it also "returns" the item removed, which you can assign to a variable. This is extremely useful and very much worth remembering.

67. Use `.append()` to add elements to the end of a list:

```python
squares.append(8**2) # adding the 64 back in, at the end of the list
print(squares)
```

68. Now that we're at it, let's append the squares of 9 and 10. But let's do it in one go:

```python
squares.append([81, 100])
print(squares)
```

Hm… that didn't quite work out. *But*, we learned something useful: you can have lists inside other lists! In fact, **you can have all sorts of stuff inside lists**. We'll discuss this at the end of this section.

69. **Mini-challenge:** let's repair the squares list. Remove that last thing that we added.

```python
squares.pop() # remove the last item
print(squares) # repaired now!
```

70. Use `+` or `.extend()` to join a list to another list:

```python
squares.extend([81, 100]) # or:
squares += [121, 144] # same as squares = squares + [121, 144]. Print to see
```

71. Use the Boolean operator `in` to find out if something is in a list or not

```python
print(4 in squares) # True
print(5 in squares) # False
```

72. Use `.index()` to find where exactly stuff is in a list, if you're sure it's there:

```python
squares.insert(3, 'papaya')
print(squares) # oops, it seems we put a fruit into the wrong list. Let's remove it.
# Let's assume we don't know where we put it. We can use .index() to find out:
idx = squares.index('papaya') # idx is just a variable to store the index that we need
squares.pop(idx)
print(squares) # great, fixed now!
```

Note: index() only works if the element you're looking for is definitely in the list. If you try squares.index('papaya') again, you'll get an error, because papaya is no longer there.

73. Use the `.reverse()` method to reverse a list

```python
revsquares = squares
revsquares.reverse()
print(revsquares) # nice! The new list is reversed
```

IX.1.4.   Copying a list

Let's recover the code from the last exercise.

```python
squares = [0, 1, 4, 9, 16, 25]
revsquares = squares # First we make a copy of the original list (...or do we?...)
revsquares.reverse() # then we reverse it
print(revsquares) # nice! revsquares is reversed now :)
print(squares) # what?!! We didn't do anything to squares! Why did it get reversed?!
```

Unlike for simple variables, using = on a list copies only the ***pointer*** to that list, not the list itself. The idea is that lists can be enormous, taking up a lot of memory. So, by default, Python doesn't want to copy the whole thing itself, only the bit that *points* to where it is stored in memory, which is known in computer science as a *pointer*.

74. So, to copy a list, use the `.copy()` method instead

```python
# This is the preferred method on Python 3:
newlist = oldlist.copy() # best and clearest
# ...or you can use the list function:
newlist = list(oldlist) # a little more obscure, but still clear enough
# ...or you could use the old method of slicing from beginning to end:
newlist = oldlist[:] # uglier but typical in Python 2, so old Pythoners use it
```

These, however, give you what's called a "shallow copy" of the list. This is getting pretty technical, but briefly: if any of the elements of the list was itself a pointer to another list, then you only copied the pointer, not the internal list. And on that note…

75. You can throw all sorts of rubbish into the same list:

```python
x = [1234, "some text", ["another", "list"], {'a': "dict"}, ("and a", "tuple"), 3.14]
```

Many purists (particularly people coming into Python from C, C++ or Java) would recommend that you only put the same kind of data into a `list`. If you want to have mixed types of data within the same data structure, you should be using a `tuple`, they say. The reality is that most Python coders freely put all sorts of things into any of their data structures, and you can too unless you have a good reason to follow a tidier practice. If you can be tidy, by all means do so, but don't hesitate to use a dirty list if that's what's convenient.

IX.1.5.   A few more list functions and properties

76. You can use the `list()` function itself to turn a `string` into a `list`:

```python
mytext = "Hello World!"
mylist = list(mytext)
print(mytext)
print(mylist)
```

77. If you only have numbers in a list, you can find their maximum and minimum easily:

```python
geekyconstants = [2.71828, 3.14159, 6.022e23, −273.15]
print(max(geekyconstants))
print(min(geekyconstants))
```

78. There can be multiple repetitions of the same item in a list, and you can count them:

```python
fruits = ["apple", "orange", "apple", "kiwi", "apples"] # someone really likes apples!
print(fruits.count("apple")) # only 2? Interesting...
```

Note that "apples" was not counted as matching "apple". Computers are very obedient… sometimes too much.

IX.2. Dictionaries (`key: value` pairs)

Dictionaries, or `dict`, are one of the coolest data structures in the galaxy. They work as key-value pairs, i.e. you can access the value using a key or index. This is more easily seen with an example.

79. Create two simple dictionaries

```python
cat = {'lives': 9, 'goes': "meow", 'likes': "murdering beautiful things"}
dog = {'lives': 1, 'goes': "woof", 'likes': ["loving humans", "chasing cats"]}
```

Note: it's a very bad idea to have the `'likes'` value be a simple `str`ing for the cat but a `list` for the dog. However, Python will not complain. It's up to us to write consistent code.

We probably should have put the `'likes'` of the cat into a list, even if it has only one item.

Anyhow, let's keep going. Oh, but before we go on, a short digression to discuss a main reason why I prefer using double quotes for normal text.

IX.2.1. So… "double quotes" or 'single quotes'?

This is very much a matter of personal choice. My own convention, which I highly recommend, is to use double quotes for any "free" text, `"such as this one"`, and single quotes for strings that represent data, such as `dictionary['keys']` or `'file_names.txt'`. Using this convention helps me realise when a string is not supposed to contain spaces, for example, and when it is supposed to have a very specific content that can't change freely (file names and dict indices don't typically change too often).

Easy rule of thumb for double vs. single quotes: if the text is for humans, double quotes; if it is for the computer, single.

OK, back to dictionaries.

IX.2.2. Accessing items inside a dictionary

80. Access the `value`s of `item`s inside a `dict`ionary by giving the `key`:

```python
print(cat) # the whole dict
print(cat['lives']) # get only one value, using its key
print(dog['goes'])
```

And you can use the `print()` function to get nice output:

```python
animal = cat
print("Here's a riddle for you:\nThis animal has", animal['lives'], "lives, it goes",
...     animal['goes'], "and it likes", animal['likes'], "\nCan you guess what it is?")
```

The three dots on the left mean that I ran out of space and the whole thing should be written in one line. Note that it is generally not great style to have a very long line. We could have done:

```python
animal = cat
print("Here's a riddle for you:\nThis animal has"
    , animal['lives'], "lives, it goes", animal['goes']
    , "and it likes", animal['likes']
    , "\nCan you guess what it is?"
)
```

…which is much nicer to read. Python won't care much about any "whitespace" (that's the geeky name for it) added *inside* a **function call** (`print()` is the function here), so add it freely so that your code is easier to read. It *will*, however, care *very much* about whitespace when we start writing `if` statements and `for` loops. I'll discuss it then.

IX.3. Nested data structures

We can have dictionaries, lists, tuples, and pretty much any kind of object inside a dictionary, or inside a list, or inside a list that's inside another list inside a tuple inside a dictionary, and so on.

81. Try the following

```python
FAVOURITE = 0
animals = {
     'cat': {'lives': 9, 'goes': "meow", 'likes': ["murdering beautiful things"]}
    ,'dog': {'lives': 1, 'goes': "woof", 'likes': ["loving humans", "chasing cats"]}
}
animal = 'cat'
print("The", animal, "'s favourite thing is", animals[animal]['likes'][FAVOURITE])
```

In the interest of full disclosure, I'd probably go with object-oriented programming for something like this. It would be way more versatile, more useful, and just prettier. But ignore that for now.

You'll notice an annoying space before the apostrophe. You can remove it using the `sep=''` parameter of the `print()` function, but then you'll have to add extra spaces. Don't worry too much about it right now, we'll see a better way to print later.

Also, you can now change the value of `animal` to `'dog'`, but it would be nice to have a way to see the data for each of the animals more efficiently. There is, and that's what loops are for (see what I did there? 😉). We'll get there shortly, but first, tuples.

IX.4. Tuples

Tuples look pretty much the same as lists, but they have one main technical difference: they are "immutable": once you put something in, you can't change it. The only caveat is you can still add stuff to the tuple, but once it's in, it stays as is for life:

```python
# Lists
mylist = [1, 2, 99, 4, 5, 6] # SQUARE brackets [] to create and access
mylist[2] = 3 # let's fix that error
print(mylist) # nice, it's fixed now :)
# Tuples
mytuple = (1, 2, 99, 4, 5, 6) # ROUND brackets () to create, but square ones to access
mytuple[2] = 3 # ERROR! tuple can't be changed ...we can't fix it :(
```

IX.4.1. So, why on Earth would anybody ever want to use a tuple?!

Well, since tuples are immutable, you can use them as keys for a dictionary:

```python
mymatrix = {
     (0,0): 2.2
    ,(0,1): 1.7
    ,(1,0): 1.8
    ,(1,1): 1.3
}
i, j = 1, 0
print(mymatrix[(i, j)])
```

Admittedly, that's a pretty niche usage, so most coders don't really use tuples all that much. But, it can be very handy for, say, longitudes and latitudes of ecological locations, and what you found there. Below I discuss another good reason for tuples (spoiler: they use less memory).

IX.5.   CHALLENGE: a `dict` indexed (`keyed`) with `tuple`s to `list` value`s`

82. Make a dictionary whose keys are tuples of potentially made-up latitudes and longitudes (in decimal degrees), and the value is an animal that you have seen there. For example, at the Museum, which is located at -73.974 degrees (west, but that's what the minus is for) and 40.781 degrees (north, so positive), I have seen a T. rex; and in my home city of Caracas I have seen a couple armadillos. Add as many animals and locations as you wish.

In all fairness, I'd rather just have the key of the dictionary be `'amnh'`, and have the value of that key be an inner dictionary that itself contains the latitude and longitude as elements, and a list of animals as a third element. This would be very much like what we did with the dog and cat in the Nested data structures section. But hey, I had to justify tuples somehow. There are also `set`s, but those I won't even bother with here other than to say:

The main value of immutable collections such as `tuple`s and `set`s is that, since they cannot change, **they are a lot more efficient in how Python stores and accesses them in memory**. Because `list`s and `dict`ionaries can contain anything and change at any time, Python has to be a lot more flexible with memory, and therefore they are slower. Imagine you had a mouse in this pocket of your jacket, and now you go and change the mouse to an elephant: it has to be a pretty flexible pocket indeed. With a tuple or set, the mouse is a mouse forever, so Python knows it will never need more space (memory) to store it. For most purposes, though, lists and dicts will do the job just fine, and that's what most coders use most of the time.

Note: when computer scientists talk about memory, they almost always mean the active memory, such as the RAM (random access memory), not the more passive hard-drive space.

X.   `for` and `while` loops: iterative statements to do repetitive tasks

Loops are absolutely awesome. Together with control statements (`if` and `else`), they form the core of computer programming. The main type of loop, at least as Python is concerned, is the `for` loop, so we start with that. There is also the `while` loop, which we will mention briefly.

X.1.   A simple `for` loop

83. Try the following, very basic `for` loop.

```python
for i in range(5):
    print("Hello", i, "times!")
```

Pretty simple, but extremely powerful, so do not underestimate it. You'll probably be writing thousands upon thousands of these in your life. Two things to note: First, remember that Python starts counting at 0. Also, you don't have to use `i` at all, but it's extremely common to do so for basic loops in which you just need a simple counter and have no reason to call it anything else.

CRUCIAL: that space before `print()` is extremely important. If you don't put it there, or if you use that space inconsistently, you'll get an `IndentationError`, a bane of Python coding.

84. You can specify from where to where, and by how much, the range goes:

```python
for i in range(20, 5, -2):
    print("Hello", i, "times!")
```

So, the first number is the start, the second is the st**o**p, and the third is the st**e**p. As you can see, we can even go backwards if we wish.

85. See what happens if you swap around the 20 and the 5 in the last example. Why?

X.2. Using a `for` loop to go over a `list`

Since we can use the `len()` function to find out how long a list is, we can do the following:

86. Go over a list the poor way:

```python
fruits = ["apple", "orange", "pineapple", "passion fruit", "kiwi"]
for i in range(len(fruits)):
    print("I want to eat a delicious", fruits[i])
```

Fine… but Python is far too cool for that!

87. Go over a list the amazing Pythonic way:

```python
for fruit in fruits:
    print("I want to eat a delicious", fruit)
```

Ain't that neat?

But what if you want to have a number as well as the item itself, for whatever reason?

88. Use `enumerate()` to go over the list items **and** also get a counter at each step!:

```python
for i, fruit in enumerate(fruits):
    print("On day", i+1, " I want to eat a delicious", fruit)
```

You've got to be enjoying this stuff (the coding, I mean; the fruits also if you wish)! I do, anyway.

X.3. Using a `for` loop to go over a `dict`

```python
basket = {'loo rolls': 96, 'bottles of sanitiser': 25, 'packs of flour': 17}
print("On my next coronapanic shopping spree, I will get:")
for product in basket:
    print("+", product)
```

Hm… that's all right, but we didn't quite get the numbers of each. We can just use the properties of dictionaries:

```python
print("On my next coronapanic shopping spree, I will get:")
for product in basket:
    print("+", basket[product], product)
```

Nice! But Python can do better.

89. Use the `.items()` method of dictionaries to get both the keys and their values at the same time:

```python
print("On my next coronapanic shopping spree, I will get:")
for product, howmany in basket.items():
    print("+", howmany, product)
```

Beautiful.

X.4. CHALLENGE: using a `for` loop to go over `dict`s inside a `dict`

90. Let's bring back our murderous cats. Iterate over them with a for loop the same way we did above, and print a nice string that tells us something about each animal.

```python
animals = {
     'cat': {'lives': 9, 'goes': "meow", 'likes': ["murdering beautiful things"]}
    ,'dog': {'lives': 1, 'goes': "woof", 'likes': ["humans", "bones", "chasing cats"]}
    ,'pig': {'lives': 0, 'goes': "oink", 'likes': ["not being turned into bacon"]}
}
# your loop here. Tell us something about what each animal says, likes and so on
```

We may want to do something nicer with that list of things that the animal likes, but without knowing a little more Python, it would be rather tricky.

X.5. A simple `while` loop

While loops operate as long as a condition is True… hence the name *while*.

91. Try the following `while` loop:

```python
fruits = ["apple", "orange", "apple", "kiwi", "níspero"]
while len(fruits):
    fruit = fruits.pop()
    print("Ñam, ñam, I ate a delicious", fruit)
print("\nNow I have", len(fruits), "fruits left :(\n")
```

Important side note: the uppercase Ñ (an awesome letter, by the way), does not seem to print in my version of VS Code, but it does in Spyder and Jupyter Notebook. If you'll be using non-English characters, you need to check that they work in the medium you intend them to.

We'll leave `while` loops at that, other than to say that they can get dangerous if you're sloppy:

```python
x = 5
while x > 0:
    function_that_fills_up_the_clusters_memory_with_huge_things()
    # will run forever and crash the cluster because x will always be greater than 0
```

So, we should have done something like `x -= 1` at the end of the while loop (remember that `x -= 1` is the same thing as `x = x - 1`, just a tiny bit shorter).

Since `for` loops take care of this kind of controls for us, it is advisable to use `for` whenever possible. But don't disregard `while` loops entirely, they can be very useful sometimes.

In any case, all this discussion of while loops running only while a condition is True, brings us to our next big topic: Boolean (i.e. True/False) logic.

XI. Logical (Boolean) operators

Boolean logic is a stunningly beautiful and crucial sub-universe of human thinking, and we only have time to scratch its surface here. Suffice it to say that computers are based on it, so you will encounter quite a bit of it in coding. In brief, this means questions of whether something is strictly true or not, 1 or 0, on or off. In Python terms, this is typically expressed as True or False (mind the first uppercase letters).

The table below presents the main list of logical operators in Python

Try to think of each of these Boolean operators as a **question**, not an affirmation.

Logical operators	¿?	Example	Result
Equality (is 1st equal to 2nd?)	==	2 == 1+1	True
Inequality (≠)	!=	2 != 1+1	False
Greater than (>)	>	2 > 1	True
Greater than or equal to (≥)	>=	1 >= 1	True
And (are 1st and 2nd both true?)	and	0 > 3 and 2 > 1	False
Or (is either of the two true?)	or	0 > 3 or 2 > 1	True
In (is 1st contained in 2nd?)	in	2 in [0, 1, 2, 3]	True

Important: Remember that we use *one* equals to assign a value to a variable, *two* to compare:

```
x = 5 # means that x is DEFINITELY equal to 5, because we just made it so
x == 5 # is wondering whether x is equal to 5 or not
```

92. Try all of the examples in the table above, one by one.

There is also a **Boolean inverter**, not, which takes any Boolean value and flips it from True to False, or vice versa.

93. Try printing the following, one by one, predicting the output before you run them

```
not True
not False
not 5 > 4
not 1 + 1 == 2
not 5 in [1, 2, 3, 4]
```

Cool! Let's look at a few other interesting examples:

```
not 1
not 0
not -5
not 5
not None
not "" # an empty str
not [] # an empty list
not () # an empty tuple
not {} # an empty dict
```

Remember that True, False and None must start with uppercase, but or, and, in and not must start with lowercase. Python is very finicky about this.

These last examples are very interesting. Let's see what they mean. First, 1 and 0: for a computer, this is fundamental – **zero means nothing, so it's** False. **Any other number is** True (including negative numbers).

That's why the first while **loop above in section X.5 works!** Python computes the length of the list, and as long as there are one or more things in there, the number is not zero, so the condition is True and the loop keeps going. Eventually, we manage to empty the list, its length becomes zero, and so the condition fails (it is False), so we exit the loop.
 this applies to many programming languages, but not all.

Ok, back to the list here. Next after the numbers is a custom Python value, None, and it does what it says on the tin. Then we have an empty string "", i.e. empty text, which is considered False in Python. Finally, the square [], round () and curly brackets {} are an empty list, tuple and dictionary, respectively. Since they're empty, Python considers them False.

You may be wondering why we've been covering all of this technical computer-sciencey logical mathsy stuff. Well, it is essential for a major pillar of computer programming: conditional statements, i.e. whether to do something or not depending on specific conditions.

XII.   if, elif, else: determining what happens with control statements

We have come to the other main pillar of computer programming (together with loops). **Control statements** allow us to decide what happens given certain conditions, so obviously programmers use them extremely frequently.

XII.1. A simple `if`

94. Write the following simple `if`:

```
x = 1 # try changing this to 1+1, 2, -2, and run again
if x == 2: # note the double equals for comparisons!
    print("x equals two")
    print("We're still inside the if")
print("And now we're outside the if")
```

That should be pretty obvious. One thing to notice is again the spacing at the beginning of each line inside the if. **Be extremely careful with these indentations**. Make sure they are the same for blocks of code that belong together, e.g. in this case everything that goes inside the loop should be indented in once.

XII.2. A simple `else`

95. You can use `else` to specify what happens `if` the condition is `not True` (i.e. `False`):

```
x = 1 # try changing this to other numbers and run again
if x < 0:
    print(x, "is a negative number.")
else:
    print("Maybe", x,"is a positive number, maybe it's zero, I dunno!")
# end if: is x positive or negative? # This line is not necessary, but I like it
```

As I mentioned at the end, that last line is just a comment, so it's certainly not necessary. However, I really like closing my `if`, `for`, `while`, and `def`, and with blocks (we'll see the latter two soon) with such a comment. I think it makes my code much easier to read, but this is by no means a standard practice (most people in fact don't do it).

XII.3. Using `elif` to provide intermediate conditions

```
x = 1 # try testing this with 1-1, -5, -2**0, 0**0 and run again
if x < 0:
    print(x, "is a negative number.")
elif x > 0:
    print(x, "is a positive number.")
else:
    print("I think x has to be zero!")
    print("...or... hang on a sec, let me check...")
    if x == 0: # an unnecessary nested if block inside the other if block
        print("Yep, x is zero, look: x =", x)
    # end if: confirming that x really is zero
# end if: is x positive, negative, or zero?
```

Hm… some weird behaviour with the powers of zero there, but other than that, it works like a charm! Obviously, **the inner `if` is not necessary**, because any self-respecting number is either positive, negative, or zero. There are, of course, imaginary numbers, but those are no self-respecting numbers, as everyone knows. Try to keep your code **robust**, but don't write unnecessary clutter. Go on and get rid of that inner if. I just wrote it to show you that **you can have "nested" `if` blocks inside other `if` blocks!** Note how the indentation got one level deeper.

XII.4. CHALLENGE

96. Generate a dictionary called `oddsquares` that contains the squares of all the odd numbers between `0` and `N`, where `N` can be altered at will. Use the odd number as the dictionary key and its square as the value, i.e. it should be something like `{1: 1, 3: 9, 5: 25, …}`. Test it with `N = 20`.

Hint: you may need to create an empty dict (or "declare" it, in computer jargon) before the for loop, then proceeding to fill it in inside the loop.

XII.5. List/dict comprehensions with conditionals

Ok, I'm sure your solution to the last challenge was very cool, but take a look at mine 😎:

```
N = 20
oddsquares = {num:num**2 for num in range(N+1) if num%2 != 0}
print(oddsquares)
```

My solution is what's called a **comprehension**, in this case a dictionary comprehension. Here's a simpler one without the `if` bit, in this case a list comprehension:

```
doubles = [2*i for i in range(10)]
print(doubles)
```

We don't really have time to delve any deeper into comprehensions, but they are one of the major elements of the concept of "**pythonicness**", i.e. writing the most Python-like code possible. Sometimes that gets dangerously close to snobbism in my opinion: it can render code that's impossible to read for anyone who is not very familiar with Python. But sometimes it really gives you much more efficient and prettier code, so it's worth keeping comprehensions in mind.

## XIII.   Functions and their arguments, parameters and returns: reusing code

It is a **very** good habit to write a `"""long multi-line comment"""` at the beginning of a function `def`inition, describing what it does, what it takes in as **arguments**, and what it `return`s. Most people write this comment only after they're done with writing their function, if at all. I strongly advise and encourage you to do it the other way around: always write what the function is supposed to do, what it takes in and what it spits out first, before you write any code into it. The reason is that this helps guide the code that you will write, and often prevents you from making silly mistakes or writing a function that you don't really need.

Actually, I'll make a brief stop here and talk a little about software engineering. Just a minute, please bear with me.

97. Use the black-box approach to design functions and code in general.

By this I mean, imagine the code as a magical black-box that does anything it needs to do, perfectly. What you really need to know before you write any of that code is two things:

1. What data do you have or need? (actually, think about this second).
2. What do you want to do to it? What should the code give you? (**start here**)
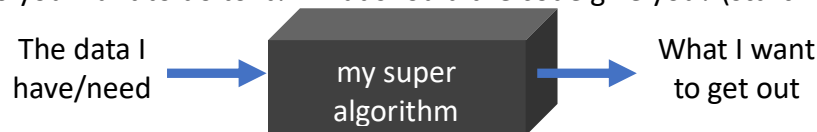
I know it sounds obvious, but it is extremely common to skip this and start coding before you even know exactly what kind of data you have or even what exactly you want to do to it. Don't.

Always start your code with pencil and paper… i.e. without any code at all!

98. **Define** (i.e. write) the following simple function, then **call** it (i.e. use it):

```python
# The function DEFINITION:
def say_hello():
    print("Hello!")


# The function CALL:
say_hello()
```

…well that's truly one rubbish function. Surely, we can do better than that.

99. Pass an **argument** or **parameter** to a function, inside the parenthesis:

```python
def say_something(thing):
    """This function says anything you want it to say, using the argument <thing>"""
    print(thing)
# end def say_something(): a function to say something arbitrary


say_something("Fare thee well!")
```

Other than the new argument `thing`, whose behaviour should hopefully be quite obvious, I have added two optional bits:

1. A multi-line comment that describes the function (ok, it only has one line here, but it could have many).
2. A closing comment that marks where the function definition ends. This is very much a personal choice of mine. Most Python coders don't use it, but I think it makes code easier (for me) to read.

Suit yourself regarding the second, but please, and to reiterate:

Do add a multi-line comment at the beginning of every function you write.

XIII.1. Docstrings

The reason for adding that multi-line comment at the very beginning of a function `def` is not only that it makes your code much easier to read and maintain but, significantly, that many IDEs actually use that text to give users information about the function. Here are Spyder and VS Code:

Spyder

VS Code

VS Code (lower) seems to make better use of the information than Spyder, but both are helpful. Jupyter doesn't seem to use the information at all, but then again Jupyter is meant more for sharing code than for writing it. Actually, it will show us quite a bit of information if we hit Shift+Tab inside a function call:

```
In [1]: def say_something(thing):
            """This function says anything you want it to say, using the argument <thing>"""
            print(thing)
        # end def say_something(): a function to say something arbitrary

In [ ]: say_something()
```
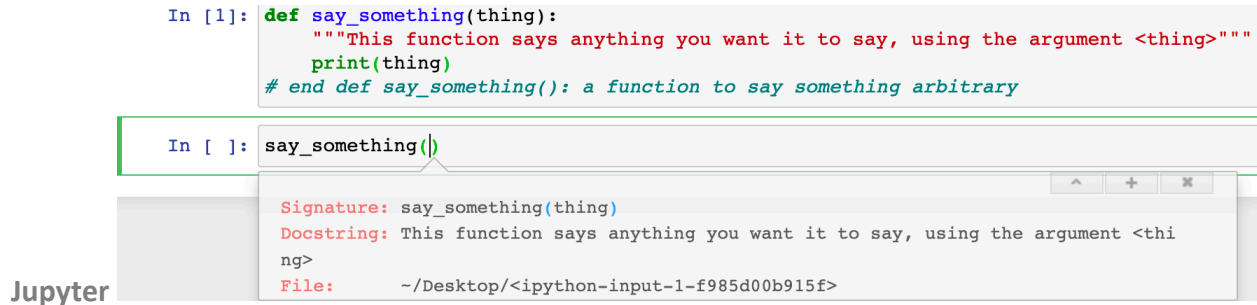
```
Signature: say_something(thing)
Docstring: This function says anything you want it to say, using the argument <thi
ng>
File:      ~/Desktop/<ipython-input-1-f985d00b915f>
```

Jupyter

So… do write the """DOCSTRING""" (that's its name, from "documentation string").

Remember that the docstring must always start on the very first line. Ok, back to functions.

XIII.2. Default and multiple values

100. Give the parameter (or argument) a **default value**:

```python
def say_something(thing="Hello Roger!"):
    """

    Says anything you want it to say, using the argument <thing>,
    which defaults to "Hello Roger!" if you don't specify anything.
    """

    print(thing)
# end def say_something(): a function to say something arbitrary
say_something(thing = "Good bye, Roger darling.")
say_something("Good bye, Roger darling.")
say_something()
```

You'll see that you can either specify the argument name or not. For this function, it's fine not to bother specifying the name of the argument because there's only one, and from the name of the function it's pretty obvious what will happen. But when we start defining more complex functions, or using those created by other people, at times it may be best to specify the arguments by name, so that your code is easier to read (by you and others).

101. When defining a function with multiple arguments, those without a default value must come *before* those with a default:

```python
def say_something_many_times(thing, times=1):
    """

    Says any <thing> you want it to say
    and repeats it a specified number of <times>.
    <thing> has no default, you must specify it!
    <times> defaults to 1 if you don't specify anything else.
    """

    for i in range(times):
        print(thing, i+1, "times.")
    # end for: printing the message <times> times
# end def say_something_many_times(): repeats an arbitrary message
```

102. Good. Let's test it now:

```
say_something_many_times(thing="Farewell", times=4)
say_something_many_times("Farewell", 4) # same as above, still works
say_something_many_times(times=4, thing="Good bye") # this also works!
say_something_many_times(4, "Adieu") # but this will fail, wrong order
say_something_many_times("Adieu") # this is fine, <times> will use its default of 1
say_something_many_times() # another error, <thing> always needs a value
say_something_many_times(4) # MINI-CHALLENGE: will this work or fail?
```

You'll see that you can actually invert the order of the arguments as long as you specify their names.

Note: Whether you call the stuff that the function takes in an **argument** or a **parameter** is a mathematical/comp-sci discussion that I will not be dragged into here. Call them what you will. Ok, fine: a **parameter** is formally the variable that you put into the function *definition*. An **argument** is the data that you send (or *assign*) into that variable when you *call* the function.
I think mathematicians actually have a different opinion about this, but I'll leave it at that.

XIII.3. return**ing values from a function**

Besides performing a job, you can have your function give a return value. This is really useful:

103. Create a function that returns the maximum between two numbers

```python
def max_between_two_nums(num1, num2):
    """Returns the maximum between two numbers.
    Fails if the numbers are the same"""
    maximum = None
    if num1 > num2:
        maximum = num1
    elif num2 > num1:
        maximum = num2
    else:
        raise ValueError("ARGH, I don't know what to do, the numbers are the same!")
    # end if: testing which number is larger
    return maximum
# end def max_between_two_nums()
```

104. Ok, now try it:

```python
print(max_between_two_nums(5, 23))
print(max_between_two_nums(5, -23))
print(max_between_two_nums(5, 5))
```

Hopefully that should be clear. You'll notice that we never used the word max, which is short and neat and would have been a great function or variable name. The thing is, Python is already using that name (for precisely the same purpose), so we can't use it to name our own stuff.

You'll see that we raised an error if the numbers are the same. The best solution if the numbers are the same would of course be to just pick one of the two and return that one, but I wanted to show you the principle of raising errors. Python comes with a bunch of pre-defined errors (such as ValueError), and you can define your own freely to match the kind of data you expect your users to give you. We unfortunately don't have time to cover errors more formally here, but:

Defining (or borrowing) and raising informative errors to your users is a very good practice.

105. You can return multiple values:

```python
def triplet_inverter(x, y, z):
    """Gets three values as input, and returns them swapped around"""
    return z, y, x
# end def: triplet_inverter()


print(triplet_inverter(1,2,3))
```

Not my most impressive function, I'll admit, but it does the job.

XIII.4. CHALLENGE: a function to get the minimum and maximum in a list

106. Create a function `minmax(the_list)` that returns the minimum and maximum from a list of numbers, such as nums = [1, 45, 3.9, -7, 8].

We'll have to go over the whole list and check if what we thought was the maximum is still the maximum, and same for the minimum.

XIII.5. Getting variable numbers of arguments with ∗args and ∗kwargs

This is a little advanced, but, wouldn't it be cool if we could take our `triplet_inverter()` function above and turn it into a `multiplet_inverter()` that can take any number of arguments?

107. Use ∗args to tell Python that you're probably expecting some arguments, but you don't know how many:

```python
def multiplet_inverter(*args):
    """Gets a tuple of any length as input, and returns its contents swapped around"""
    #print(type(args)) # it's a tuple!
    outlist = []
    for num in args:
        outlist.insert(0, num)
    # end for: adding each
    # because we received a tuple as input, it's fair to give a tuple back
    return tuple(outlist)
# end def: multiplet_inverter()
print(multiplet_inverter(1, 2, 3, 4, 5))
```

Of course, there are better ways to invert a tuple, but this was just a proof of principle.

Cooler still, you can have an unspecified number of named arguments also!

108. Use ∗∗kwargs to specify arguments with a unique name, even without defining them into the function as parameters.

```python
def privacy_violator(name, **kwargs):
    """Can deal with any number of named arguments"""
    print("name =", name)
    for stuff in kwargs:
        print(stuff, "=", kwargs[stuff])
    # end for: printing all the stuff in kwargs
# end def: privacy_violator()
privacy_violator("Jonnie", hometown="NYC", mother="Gina", password="12345")
```

Actually, the names ∗args and ∗∗kwargs are purely convention, you can change them if you wish (don't). What really matters are the ∗single and ∗∗double asterisks preceding them.

XIV.   Working with strings and text

XIV.1. Common properties between lists and strings

Strings are seen by many programmers as lists of characters, so the creators of Python made many of the methods, functions and behaviours the same between the two classes.

109. Try the following examples, in which strings behave just like lists:

```python
mystr = "New York City"
print("ork" in mystr) # oh no, there's an ork in New York City! :-O
print(mystr.index("ork")) # start counting at zero!
print(mystr[2:6]) # remember the weird Python index counting
print(mystr[::-3]) # print every 3rd character, going backwards
```

XIV.2. Some useful string functions

110. Use `.replace()` to … erm… replace:

```python
# Let's get rid of that evil ork from NYC and replace it with a lovely elf:
print(mystr.replace("ork", "elf"))
print(mystr) # oh well, it seems the original remains unchanged
# What if we try to replace something that appears many times?:
print(mystr.replace(" ", "%")) # all the spaces get replaced
print(mystr.replace(" ", "%", 1)) # you can limit how many times you replace
```

Most of these functions don't really change the original string. You'd need to store the changes into a new variable (or replace the original on the fly).

111. Use `.split()` to turn a string into a `list`, splitting at whichever character (or string) you choose:

```python
mystr = "Methinks it is like a weasel"
print(mystr.split())
print(mystr) # good to know that the original remains unchanged by .split()
```

So, if you don't specify anything to split, it just breaks at any space. Actually, it breaks at any **whitespace**, which includes new lines (\n), tabs (\t), carriage returns (\r), and simple spaces. But nothing stops you from using some other character if you wish:

```python
print(mystr.split('i'))
```

Actually, you don't have to use a single character. You can use a longer string:

```python
print(mystr.split('s '))
print(mystr.split('thinks it '))
```

112. The good brother of the evil `.split()`, is `.join()`, which needs a joining string as its base, and a `list` (or technically, any `iterable` object) as its input:

```python
mylist = ["1", "2", "tomato", "more things", "8", "9"]
print(' AND '.join(mylist))
```

Both of these functions will be extremely useful when we read and write comma-separated values (CSV) files (or tab-separated, which I like to call TSV, although that's not a standard).

113. The good brother of the evil `.split()`, is `.join()`, which needs a joining string as its base, and a `list` (or technically, any `iterable` object) as its input:

```python
mylist = ["1", "2.07", "tomato", "more things", "-8", "9"]
print(' AND '.join(mylist))
```

All members of the list must be strings themselves, that's why the quotes around the numbers.

But we could use a fancy **list comprehension** if we have numbers that need to be **parsed** to `str`:

```python
mylist = [1, 2.07, "tomato", "more things", -8, 9]
print(' AND '.join([str(item) for item in mylist]))
```

There are many more string functions, such as `strip()` to remove whitespace to the sides (we will use it later), and a bunch more. Do search for "python string functions" if you wish.

XIV.3. Printing pretty output with f"**strings**"

114. Precede a string with `f` to use `{variables}` inside it:

```python
x = 5
day = "Friday"
print(f"I called you {x} times last {day} and you didn't pick up!")
```

115. You can use functions and do operations inside an f-string:

```python
ages = [17, 72, 34, 55]
print(f"The oldest person is at least {max(ages) * 12} months old.")
```

But f-strings are much cooler than that, you can format strings in very advanced ways:

```python
nums = {'money':752, 'base rate':1.4, 'c':0.4, 'age':19, 'pi':3.14159}
for num, val in nums.items():
    print(f"#{num:^12.3}:{val:>010.2f}") # try changing the > to ^ or <
```

We won't go into further details, just search for "python f string numbers" or something like that.

XV.   Getting input from the user

116. Getting input from the user is very easily done in Python:

```python
name = input("Please enter your name: ")
age  = input("Now please enter your age: ")
print("Thank you, " + name + ". I hope you had a wonderful " + age + "th birthday!")
```

Actually, now that we know how to use f-strings, we can make that code prettier:

117. Change the last line to an f-string:

```python
print(f"Thank you, {name}. I hope you enjoyed your {age}th birthday!")
```

Maybe we could improve that a bit, to respond correctly to years ending in 1, 2, or 3, so that they get "st", "nd", and "rd" as opposed to "th"… but actually only if the number is not 11, 12, or 13, in which case it actually should still end with "th"… anyhow, it can get tricky, but you're an `if-elif-else` master by now, so you could do it without much problem ☺

We actually have to trust that the user gives us a proper name and particularly an integer as their age. This would be handled with `Exception`s, `assert`ions, `try`s, and `Error`s, which we do not have time to cover here.

118. Mini-challenge: give the user good wishes for their *next* birthday instead.

XV.1. CHALLENGE: read and process user input

119. Write a script that asks the user how many days ago they followed the installation instructions at the beginning of this guide. The script should distinguish between and give different messages to four groups of people: those who installed only up to 1 day ago, those who installed between 2 and 4 days ago, those who installed between 5 and 9 days ago, and those who installed 10 days ago or earlier.

The truth is, most people don't take user input very often, not in Python code anyway.

XVI. Reading and writing files

120. Go to my GitHub (https://github.com/vsojo/Python_Workshop)and get two files:
    - A list of (fake) attendees to the workshop.
    - Global population data from the UN.

If the size of the files is not too large, it's always a good idea to try to open them to take a look. Open them in a spreadsheet editor (e.g. Excel), and in a simple text editor—you can use Spyder itself, vi/vim/emacs if you're a geek, or you could consider downloading the awesome Notepad++ for Windows or BB-Edit for Mac if you don't have them already.

Note: these files are both small, but if the file is very large (several MB or more), you may want to use the head command on the Linux or Mac terminal to take a peek.

XVI.1. Opening files using `with` and `open`

There are many ways to open files in Python. Technically, you have to open the file, read it (or write into it), and then close it. My understanding is that the best way to do that is to use a `with` statement, which takes care of the closing safely even if something fails within your code. So, we'll do that.

121. To open a file, use `with` and `open`:

```python
wd = '/Where/ever/you/put/the/files/'
with open(wd + 'PythonWorkshop_Attendees.csv') as f:
    for line in f:
        print(line)
    # end for: reading the file line by line
# end with: opening the file
```

Remember that the last two lines are just my personal preference. Most coders don't use that kind of closing comment, but I just think it makes my code easier to read, particularly when the blocks get very long. That said, I learned to code in C++, so it may just be an old habit.

If you put the files in the same place as your script, and you're coding on Spyder or Jupyter, you should be able to just leave the working directory variable blank (`wd = ''`) or out entirely. Either way, I'll assume you know how to find your stuff.

The script above reads and then prints every line of the file, one by one. Alternatively, we could have just read the entire file in one go using a method called readlines(), but I won't even bother explaining that here because reading a file line by line as we're doing above is way more efficient. Imagine you're processing human genomes, or transcriptomic data from a population study with thousands of individuals… you most certainly do not want to upload all of that stuff into memory at the same time unless you really need it all at the same time. In this case—and in most cases— we don't, so we can just process each line one at a time, get what we need out of it, then forget about it—quite literally, since we remove it from memory by loading the next line into the same variable (which we called `line` here, but we can call it anything we wish).

The lines look pretty spaced out, don't they? That's because they all end with a newline character "\n", and Python kept it when reading. We could just cut that last character by doing something like `line = line[:-1]`, but it's more elegant to use Python's in-built method `.strip()`:

122. Add the following line before the print call in the code above (inside the for loop):

```python
    line = line.strip()
```

This removes any "whitespace" (all kinds of blank contents) at both ends of a string.

The file is a CSV, i.e. a comma-separated-values file. That means—surprise—that the values are... separated... by commas... Anyhow, let's separate that data:

123. Use `.split(',')` to split a line of a csv file into a list. Actually, go ahead and assign values on the fly:

```python
with open(wd + 'PythonWorkshop_Attendees.csv') as f:
    for line in f:
        line = line.strip()
        name, age, institution = line.split(',')
        print(name, age, institution)
```

Hm, it seems the first four lines worked without problem, but the fifth one failed. If we take another look at the raw text file, we will see that we don't have an institution for Mike, so that line only has 2 values. Here, we could go and add "Unknown" or something like that, but what if we have thousands of entries? Maybe it's best to store the values in a list and process case by case, checking whether the age and institution are missing:

124. Store the split fragments into a list, and process the cases in which

```python
with open(wd + 'PythonWorkshop_Attendees.csv') as f:
    for line in f:
        line = line.strip()
        fields = line.split(',')
        if len(fields) == 3:
            name, age, institution = fields
        elif len(fields) == 2:
            name, age = fields
            institution = "(unknown)"
        elif len(fields) == 1:
            name = fields[0]
            age = -999
            institution = "(unkown)"
        else:
            # fields should not have any other length
            raise IndexError(f"This file has too many values per line: {len(fields)}."
                +"\nAre you sure this is the right file?")
        # end if: testing if the line has the right number of values
        print(f"{name} is {age} years old and works at {institution}")
    # end for: reading the file line by line
# end with: opening the file
```

Cool! An age of -999 is so ludicrous that it will make you realise that something is wrong or missing. I like to use numbers like that when something is missing or cannot be.

XVI.2. CHALLENGE: operations on data from files

125. Calculate the mean age of the workshop attendees.

XVII. Where to from here

NumPy, Pandas, MatPlotLib, BioPhython, RegEx, GeoPy, and a whole lot more!