00_python_refresher

October 29, 2020

1 Welcome to Python

You have two options for this class:

1. You can run the code online, using MyBinder. This means you only need your browser, and not necessarily a local Python installation. (At least not for now, which could be useful if lockdown is a limiting factor for you.)
2. You can install Python on your computer. If you're on Windows or OS X, I recommend the Anaconda distribution. It comes with a visual installer that will install basic Python, and a number of additional packages you will need for this class.

Both of these options are explained in further detail below. Choose one, and scroll down to the relevant section.

1.1 Welcome to MyBinder

This could say "Welcome to Jupyter Notebooks", as that's what you're really looking at. This is a tool for creating interactive worksheets that combine text (which you're reading right now) and code (which you'll see a bit later).

Jupyter notebooks can be rendered in different ways. You might be reading this as a PDF document, via GitHub, or in an interactive session running on your own computer. Alternatively, you might already be looking at an interactive session on MyBinder.

MyBinder is a website that can run Jupyter Notebooks within a browser. This is great, because it means that you can run Python code in your browser *without* having a local Python installation. If your computer doesn't allow you to install anything on it (e.g. because it's locked down by an IT department), this is the option for you.

The link to this worksheet on MyBinder is:

mybinder.org/v2/gh/esdalmaijer/Cambridge_SSRMP_Python/master

1.1.1 Using MyBinder

If you clicked on the link above, you should have been taken to a page with several folders. You're looking for the folder `00_welcome_to_python`, and in that folder the notebook `welcome_to_python` (this one!).

The Jupyter Notebook interface has this text and code snippets, and a bar on top with several options. The text bits are pretty simple: You can read them using your eyeballs, and/or text-to-voice or text-to-touch software on your computer. More interesting are the code bits.

Code can be run in several ways. You can hover over a code snippet, and an arrow-like symbol will appear to the left. Clicking on this makes the code run. Try it on the snippet below!

[ ]:
```
print("Test me!")
```

If all went well, you should now see the text "Test me!" directly below the code snippet.

Another way of running code is by using the "Run" button in the menu on the top of the page.

One important thing to keep in mind is that any variables in memory are created *as you run snippets*. This means that the order in which *you* run things is important, not the order in which snippets appear. For example, try running the two snippets below in order (top first, bottom second):

[ ]:
```
print(a)
```

[ ]:
```
a = 3
```

If you did it in the top-to-bottom order, the first snippet should have caused an error.

Now do it again, but run the `a = 3` snippet before the `print(a)` snippet.

If all went well, the `print(a)` snippet didn't result in an error now, because the variable it referenced was created by the `a = 3` snippet.

This might not feel like a problem NOW, but inevitably you'll run into a situation where you forgot to run a snippet, or are trying to run a previous snippet after running later snippets. This WILL result in issues.

If you run into a weird issue and you think it might be to do with the order in which snippets were run, the best thing is to simply restart the whole worksheet. You do this by clicking on the "Kernel" option in the menu, and then click "Restart and clear output". This will simply restart the Python session, and thus removes all potential weirdness that was built up in memory.

Finally, choosing "Kernel" and then "Restart and run all" will make all snippets in this worksheet run from the top.

1.2 Welcome to Anaconda

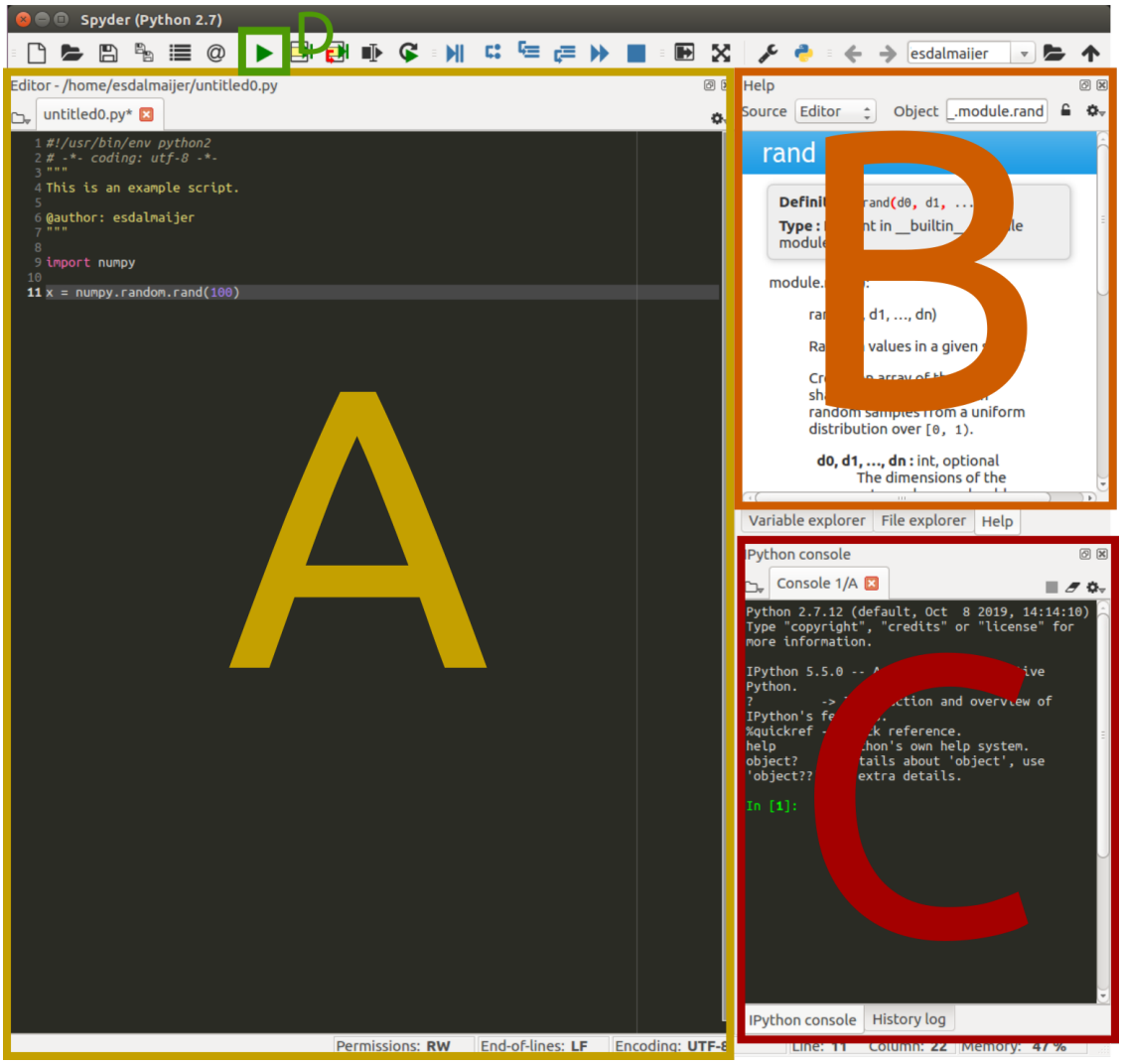
If you're choosing to run code on your own computer, your best bet is to download the Anaconda Python distribution. This is maintained by the company Continuum. It incorporates basic Python, and many additional packages that are useful in scientific contexts.

You can download this distribution here: www.anaconda.com/products/individual (make sure to get the Python 3.7 version!)

The download and installation procedure can take a while, so feel free to watch some cat videos while you're waiting.

1.2.1 Using Anaconda and Spyder

Anaconda comes with the Spyder code editor pre-installed. This looks as follows:



- Area A (yellow outline) is the code editor. This is where you will be writing script files. These are whole files that will not be run automatically, but only when you decide to run the whole file.
- Area B (orange outline) is the help window. If you're confused about a function in the script, hit Ctrl+I, and info will appear in this window.
- Area C (red outline) is the **console**. This is an interactive Python session. If you type a line in here and hit Enter, it will run straight away. Restarting the console (by clicking on the little cross next to its name) will wipe the memory.
- Area D (green outline; all the way on the top) is the "Run" button. This will make the current script run in its entirety. It will be run line-by-line, using the active console. Any visible results will appear here too.

Spyder is a very decent editor, especially if you're used to Matlab or R Studio (both look very similar), or if you're completely new to coding. Code Editors are like text processors or slideshow

software: There are several options, and each come with their own benefits and drawbacks. Like text or presentation files, script files are to construct a complete narrative to your computer.

For example, you could use the console as a quick calculator or to try out a few things. The code editor is for larger files, which could be entire analysis scripts, experiments, or a coherent set of functions for later use. Python scripts should be saved with the `.py` extension.

2 Python Basics

This rest of this worksheet will cover the basics of Python: types of values, variables, statements, functions, and classes. First, let's get some jargon out of the way.

- **value type**: Programming languages can recognise several specific value types, including numbers and text. We'll go through all important types in this worksheet.
- **variable**: A variable has a name, and is linked to a value. It's a way of keeping track of specific values in memory. For example, you could say `a = 3`. The value `3` is now linked to the variable `a`. You can refer to `a` later in your script.
- **statement**: There are several types of statements. They are the building blocks of a programming language, and allow you to repeatedly run small pockets of code, or to run a bit of code only if a certain condition is met.
- **function**: A function is a defined collection of code. It usually takes some input, processes this, and then returns some output. For example, the function `sum` takes as input a collection of numbers, adds them all together, and then returns the sum. Functions can be short and simple, or long and complex, and anything in between. Some functions are built into Python, some are inlcuded in external packages, and you can also write your own.
- **class**: A class is a collection of variables and functions, all linked to the same **instance**. For example, you can think of a car as an *instance*: It has its own properties (like its top speed), and its own functions (like driving). The blueprint for a car is more like a *class*: It describes how a car should be built, for example with four wheels. Even when produced with the same blueprint (class), cars (instances) can be different from each other. For example, they could have different wheels, or a different colour.

If these descriptions seem a bit unclear still, don't worry: We're going to go through them in more detail in this worksheet.

2.1 Numbers

Numbers come in two formats: integers and floats. **Integers** are whole, undivided numbers like -2, 0, and 15. **Floats** (floating point) numbers are fractions, for example 3.1, 5.43, and 0.6666666666666667.

2.1.1 Integers

You can use integers in basic operations like addition, subtraction, multiplication, and division:

[ ]:
```
1 + 1
```

[ ]:
```
5 - 3
```

[ ]:
```
10 - 900
```

[ ]:
```
2 * 10
```

[ ]:
```
10 / 5
```

Let's try allocating an integer to a **variable**, and do operations with that:

[ ]:
```
a = 3
a * 2
```

2.1.2 Floats

Floats can also be used in basic operations:

[ ]:
```
0.5 + 0.1
```

[ ]:
```
1.5 - 0.5
```

[ ]:
```
5.0 * 0.25
```

[ ]:
```
1.25 / 5.0
```

You can do the same by assigning floats to variables:

[ ]:
```
a = 0.5
b = 3.0
a * b
```

2.1.3 Operations

Aside from the basic addition (`+`), subtraction (`-`), multiplication (`*`), and division (\), there are also operators to do exponentiation (`**`), integer division (\\), and to compute the remainder after division (`%`). Let's see them in action:

Exponentiation Three square (3^2) and two to the power of eight (2^8):

[ ]:
```
3**2
```

[ ]:
```
2**8
```

Division (integer and remainder) The result of a division can be a fraction, which can be expressed as a float. However, sometimes it can be useful to know the result as an integer and the remainder. For example, the outcome of five divided by two is 2.5, but can also be expressed as 2 with a remainder of 1. In Python, you can compute these as follows:

[ ]:
```
5 // 2
```

[ ]:
```
5 % 2
```

2.1.4 Should I be aware of the difference between int and float?

Previously, divisions of two integers ignored the remainder. So `5/2` would be the same as `5//2`, both resulting in `2`. In Python 3 (the newest version), this is no longer the case: `5/2` should result in `2.5`. While this is an intuitive result to humans, many programming languages are more fussy about variable types. That an operation involving two integers results in a float could thus be considered a bit weird.

As a human, you might be relieved that Python's behaviour maps onto your own, rather than on a more rigid interpretation of what numbers are. There are people who argue that this means you don't have to play attention to the distinction between integers and floats.

Those people are wrong. At some point, you'll encounter functions that require integers, and can't deal with floats. This is important if you're dealing with undivisible quantities, like pixels: there's no such thing as half a pixel.

That means that while `5` and `3` are both OK pixel values, `5/3` is NOT. This is easy to spot with numerical values, but less so if you can only see variable names `a/b`.

2.2 Text strings

Python can also deal with text values, in the form of the **string** type. Strings are denoted by single (`'`) or double (`"`) quotes, or even three quotes (`'''` or `"""`). Par example:

[ ]:
```
"This is a string"
```

[ ]:
```
'This is also a string'
```

[ ]:
```
"""This is a single string too"""
```

[ ]:
```
'''This is not a string. Jokes, this totally is a string!'''
```

Strings can also be empty:

[ ]:
```
""
```

You can combine strings using the `+` operator:

[ ]:
```
"Strings can be" + " combined!"
```

And you can even use the `*` operator to multiply a string!

[ ]:
```
"Cool," + 3 * " cool"
```

2.2.1 Introducing print

At this point, it might be good to learn about the `print` function. We'll get to functions n more detail later, but this one is good to known about in the context of strings. When running a script, output is not usually produced *unless you specifically request it*. The `print` function is a way to make this happen.

For example, if you assign a value to variable `ghost` (either in a console or in a Jupyter Notebook), you don't get any visible feedback:

[ ]:
```
ghost = "man wearing a sheet"
```

Let's use print to shine some light on what `ghost` is:

[ ]:
```
print(ghost)
```

The `print` function can be used on any variable, not just on strings. It can be useful in scripts to check the value of a particular variable, or to give you periodic updates of what is happening.

2.2.2 String formatting

Strings come with their own formatting options. Specifically, you can use `{}` as a placeholder for a value that you might want to add later. An example:

[ ]:
```
s = "I am {} years old"
```

To assign a value to the wildcard, you can use the sting's `format` method:

[ ]:
```
print(s)
print(s.format(42))
print(s.format("twenty"))
print(s.format(2.5))
```

2.3 Collections

So far you've only seen single values, whether numbers or texts. (Although strictly speaking a string could be considered a collection of single characters!) There are also variable types that allow you to combine those.

A **tuple** is a collection of values:

[ ]:
```
a = (1, 2, 3)
```

You can get at specific values in a list by using their **index**. For this, Python starts counting at 0. So, it you would like to get at the first value, you would use `a[0]`. The second value is at `a[1]`, and so on:

[ ]:
```
print(a[0])
print(a[1])
print(a[2])
```

Tuples cannot be altered (they can be completely overwriten)

[ ]:
```
a[0] = 2
```

An alternative that CAN be altered is the **list**:

[ ]:
```
a = [1, 2, 3]
```

You can index lists in the same way as tuples:

[ ]:
```
print(a[0])
print(a[1])
print(a[2])
```

You can alter any value in a list directly:

[ ]:
```
a[0] = a[0] + 5
print(a)
```

Altering the entire list at once is NOT possible:

[ ]:
```
a = a + 5
```

2.4 For loops

If you do want to change all values in a list, you can do so one by one. (In the example below, the `+=` operator is used. This means "add to this variable". That means that `a = a + 5` is the same as `a += 5`)

[ ]:
```
a = [1, 2, 3]

a[0] += 5
a[1] += 5
a[2] += 5
```

For three values, this is still doable. But if your list has hundreds of values, having to type hundreds of additional lines would take FOR EVER! This is where loops come in!

You can use a **for loop** to cycle through all values in a collection:

[ ]:
```
for value in a:
    print(value)
```

To be able to change the entire `a` list, you would need to be able to get all indices. There is a function that can do this! It's called `range`, and it provides a list with values from a starting point and an ending point. Here, the starting point should be 0 (the first index), and the ending point should be the length of the list. The length of the list can be computed with the `len` function.

[ ]:
```
start = 0
end = len(a)
indices = range(start, end)
print(indices)
```

Now that you know how to get the indices, you can use them to index all values in `a` in a single for loop:

[ ]:
```
for i in range(0, len(a)):
    print(a[i])
```

Instead of printing all values in `a`, our plan was to change all values in `a`. So let's do that with our newfound ability to loop through all values in the list:

[ ]:
```
print(a)
for i in range(0, len(a)):
    a[i] += 5
print(a)
```

2.5 Booleans and logic

A *Boolean* is a variable that can only be one of two things: `True` or `False`, which is equivalent to `1` or `0`.

[ ]:
```
a = True
b = False
```

The fact that Booleans are really `0` and `1` can lead to odd behaviour, for example when you multiple True with any number:

[ ]:
```
print(3 * True)
print(-1 * True)
```

If you want to combine Booleans in a more sensible way, you can use *logical operators*. These include `and`, `or`, and `not`.

The `and` operator only returns `True` if ALL conditions are `True`:

[ ]:
```
print(True and True)
print(True and False)
```

```
print(False and False)
```

The `or` operator returns `True` if ANY conditions are 'True:

[ ]:
```
print(True or True)
print(True or False)
print(False or False)
```

The `not` operator can reverse a Boolean:

[ ]:
```
print(not True)
print(not False)
```

2.5.1 Comparisons

Booleans can be the result of comparisons. You can check whether any two variables are the same by using the `==` operator:

[ ]:
```
print(3 == 5)
print(10 == 10)
print("a" == "b")
```

Other comparisons are "smaller than" (`<`), "greater than" (`>`), "smaller than or equal to" (`<=`), "greater than or equal to" (`>=`), and "does not equal" (`!=`).

[ ]:
```
print(2 < 3)
print("a" != "b")
print(2 + 3 <= 8 - 3)
```

The `and` operator can also be written as `&`, and the `or` operator can be written as `|`:

[ ]:
```
print(True and False)
print(True & False)
print(True or False)
print(True | False)
```

2.5.2 If statements

Booleans can be used in logic statements. One of these is the `if` statement, which can be used to run specific lines of code depending on a situation. For example, you might want to check if a value is even. An even value is divisible by two without a remainder.

[ ]:
```
a = 4

if a % 2 == 0:
    print("This value is even!")
```

You can add alternative options to your if statement using `elif` (short for "else if"). In this case, you could check if a value is odd, meaning it will leave a remainder of 1 after being divided by two.

[ ]:
```
a = 5

if a % 2 == 0:
    print("This value is even!")
elif a % 2 == 1:
    print("This value is odd!")
```

In this case, the if statement covers all options, provided the variable `a` points to an integer. In cases where there are more options, you can add as many `elif` statements as you would like.

In addition to `if` and `elif`, there is a "catch-all" type of statement too. You can use this to cover any options that hadn't yet been covered in your `if`s and `elif`s:

[ ]:
```
a = "a"

if a < 0:
    print("This value is negative.")
elif a > 0:
    print("This value is positive.")
else:
    print("This value must be 0!")
```

The above statement first checks if a value is below zero, then if it's above 0, and finally concludes that if the value was neither below nor above zero, it must be zero!

You might have noticed that it is assumed that `a` points to a numerical value. You could try to make it a string or a Boolean, and see what happens. Seriously, try it! You might find that a string is misrecognised as a positive value, that `True` is also a positive value, and that `False` is zero.

One way to avoid this mess, is to check whether `a` is actually a number. Checking a variable's type can be done with the `type` function:

[ ]:
```
a = 5

if type(a) == int or type(a) == float:
    print("This is a number!")
else:
    print("This is not a number...")
```

2.5.3 Nesting and indentation

As you might have noticed, the if statements are followed by lines of code that are **indented**: They are four spaces ahead of the previous line. These four spaces are how Python know what should be run by the if statement.

You can use this to **nest** if statements: Write one statement within another. Let's try this by combining the previous examples:

[ ]:
```
a = 5

if type(a) == int or type(a) == float:
    print("This is a number!")
    if a < 0:
        print("This value is negative.")
    elif a > 0:
        print("This value is positive.")
    else:
        print("This value must be 0!")
else:
    print("This is not a number...")
```

2.6 Functions

You already encountered four functions from basic Python: `print`, `type`, `len`, and `range`. You can also create your own! For example, you could take rewrite previous bit of code as a function.

Function's require three things: a name (duh!), inputs, and outputs. Here, your function name could be `pos_or_neg`, because the code determines whether a value is positive, negative, or 0. The input is a numerical value (`number`, and the output is a string (`output`).

In Python, you can write this using `def`:

[ ]:
```
def pos_or_neg(number):
    if number < 0:
        output = "neg"
    elif number > 0:
        output = "pos"
    else:
        output = "0"
```

We can now use this function:

[ ]:
```
print(pos_or_neg(5))
print(pos_or_neg(-10))
```

OK, so you didn't actually see any output. Maybe it helps if you try printing the `output` variable directly?

[ ]:
```
print(output)
```

Oh, no! An error!

The variable `output` is **local** to the function. It exists within the function, but cannot be accessed outside of the function. OOPS!

To make the output available outside of the function, it needs to be **returned**:

[ ]:
```
def pos_or_neg(number):
    if number < 0:
        output = "neg"
    elif number > 0:
        output = "pos"
    else:
        output = "0"
    return output
```

Let's try it now:

[ ]:
```
print(pos_or_neg(5))
print(pos_or_neg(-10))
print(pos_or_neg(0))
```

We currently assume that the `number` variable passed by whoever uses the function is in fact a number. However, you can just as easily pass something else, and end up in trouble:

[ ]:
```
print(pos_or_neg("minus twelve"))
print(pos_or_neg("0"))
```

The same solution as before applies here too: You need to check whether the passed input value is actually a number. Only if it is a number, you can return something sensible. If it is not a number, you can pass `None`. This is a special variable type that basically is nothing.

[ ]:
```
def pos_or_neg(number):
    if type(number) is int or type(number) is float:
        if number < 0:
            output = "neg"
        elif number > 0:
            output = "pos"
        else:
            output = "0"
    else:
        output = None
    return output
```

Let's try!

[ ]:
```
print(pos_or_neg(5))
print(pos_or_neg(0))
print(pos_or_neg(-10))
print(pos_or_neg("5"))
print(pos_or_neg(True))
print(pos_or_neg(None))
```

2.7 Comments

So far, you've only seen code that is runable. You could read it, but the main purpose was for the computer to read it. In addition to this, there is also code that is *not* intended for the computer. This is text that you add to your scripts specifically for humans who read the code. These can be notes to yourself, or to anyone else who might have to work with your code.

Comments are super useful. You might not appreciate this right now, but human memory is fallible. Code that you read or wrote and understand now, is going to be completely unintelligible in a few weeks time. (Sometimes I don't even understand what I did before lunch...) Comments allow you to explain what each part of your code does.

Let's add some comments to the function you wrote above. Comments are preceded by the pound sign (#), and Python will ignore them:

[ ]:
```
# Function definitions start with `def`, and require
# that the input is specified between brackets. Here,
# `number` is the input.
def pos_or_neg(number):
    # We can't assume that the passed value for `number`
    # is in fact a number. Whether it is should thus be
    # tested. The `type` function returns the type of
    # the passed variable. Numbers can be of type `int`
    # or `float`.
    if type(number) is int or type(number) is float:
        # All numbers below 0 are negative.
        if number < 0:
            output = "neg"
        # All numbers above zero are positive.
        elif number > 0:
            output = "pos"
        # This else statement should only be run if the
        # value was not below 0, but also not above 0.
        # The only number for which this is true is 0.
        else:
            output = "0"
    # This else statement is run if the first if
    # statement isn't True. This only hapens if the
    # passed value for `number` was not an int or a
    # float. In these cases, `None` should be returned.
    else:
        output = None
    # The return statement makes sure that the output is
    # available outside of this function. This means
    # that the function can be called like this:
    # `out = pos_or_neg(5)`
    return output
```

There is no such thing as *too* elaborate commenting. One general guideline is that the ratio of

comments to code should be about 2:1. That means two-thirds of your script would be non-functional comments. (It is very rare to actually see this in practice, but it's much less rare to curse code that is under-commented!)

3 Modules that extend Python

All of the above is available in Python without having to do anything. However, there are also functions that are tucked away in modules. Basic Python comes with a lot of these, each adding specific functionality. This includes, for example, the `math` module for mathematical operations (think `sin`, `cos`, `tan`, etc.), the `os` module for things relating to the operating system, `multiprocessing` and `threading` for parallel computing needs, and MANY more.

These modules offer quite specific functionality, so there's a good chance you won't need them. To avoid any confusion or accidental overwriting of function/variable names, the additional modules are thus not loaded by default. Instead, you need to load them explicitly.

For example, say that you wanted to use the number π in your script. It's a part of the `math` module, which you have to load before using it:

[ ]:
```
import math

print(math.pi)
```

3.1 NumPy

In addition to the built-in modules, there exist *many* external packages that offer even more functionality. One of the most important ones is **NumPy**, short for numeric Python. This package is a crucial tool for many people, and basically unmissable to scientists.

One of the cool elements of NumPy is the `array`. This is a variable type that can represent vectors or matrices of values, usually numerical value. (But they can also be strings, or any other variable type.) One of the main advantages of the NumPy array is that you can use arithmetic operations on the whole array. This is unlike the `list`, which had to be looped through. Side-by-side comparison:

[ ]:
```
import numpy

# Change all values in a list:
l = [1, 2, 3]
for i in range(len(l)):
    l[i] += 5
print(l)

# Change all values in a NumPy array:
a = numpy.array([1,2,3])
a += 5
print(a)
```

Arithmetic isn't the only thing you can do with a whole array. You can also compare all values in the whole array at the same time against a single value, or you can compare all values in one array against all values in an array of the same shape:

[ ]:
```
a = numpy.array([1, 2, 3, 4])
b = numpy.array([0, 2, 0, 4])

print(a == 2)
print(a == b)
```

The results of these comparisons are **Boolean arrays**. You can combine these in the same way as single Booleans:

[ ]:
```
a = numpy.array([True, False, True, False])
b = numpy.array([True, True, False, False])

print(a | b)
print(a & b)
```

NumPy is full of useful functions, and you will be seeing a lot more of it today!