Python-tutorial

September 4, 2019

1 Python Basics

This is a brief introduction to the basics of Python programming

TIP: To run the code in a line of code in Canopy, select the line of text you wish to run and hit Ctrl (on Mac) + Shift + R .

```
In [ ]: X = "Python is fun"
        print X
```

1.1 1. Variables

Variables are containers for storing and manipulating information. For example, we can assign the value "Python is fun" to the variable X. The = sign is an assignment operator.

In Python, variables are designed to hold specific types of information. For example, after the first command above is executed, the variable X is associated with the string type. There are several types of information that can be stored:

- Boolean. Variables of this type can be either True or False.
- Integer. An integer is a number without a fractional part, e.g. -4, 5, 0, -3.
- Float. Any rational number, e.g. 3.432.
- String. Any sequence of characters.

The string "5" and integer 5 are completely different entities to Python, despite their similar appearance. You'll see the importance of this in the next section.

Exercise: Write a program that stores the value 5 in a variable X and prints out the value of X, then stores the value 7 in X and prints out the value of X (4 lines.)

In order to check the type of the variable, we will use the function type.

```
In [ ]: type(X)
```

Let's try other variable types:

```
In [ ]: a = True
        b = 5
        c = 3.2
        d = "Math"
```

```
In [ ]: print(type(a))
        print(type(b))
        print(type(c))
        print(type(d))
```

1.2 2. Commands that operate the variables

Just storing data in variables isn't much use to us. Right away, we'd like to start performing operations and manipulations on data and variables.

There are three very common means of performing an operation on a variable.

1.2.1 2.1 Use an operator

All of the basic math operators work like you think they should for numbers. They can also do some useful operations on other things, like strings. There are also boolean operators that compare quantities and give back a `bool` variable as a result.

```
In [ ]: # Standard math operators work as expected on numbers
        a = 2.0
        b = 3
        print(a + b)
        print(a * b)
        print(a ** b)  # a to the power of b (a^b does something completely different!)
        print(a / b)
```

Can you see the mistake? What are the possible solutions?

```
In [ ]: # There are also operators for strings
        print 'hello' + 'world'
        print 'hello' * 3
```

```
In [ ]: # Boolean operators compare two things
        a = (1 > 3)
        b = (3 == 3)
        print a
        print b
        print a or b
        print a and b
        a = True
        b = True
        print a != b
```

1.2.2 2.2 Use a function

These will be very familiar to anyone who has programmed in any language, and work like you would expect.

```
In [ ]: # There are thousands of functions that operate on variables and objects
        print len('hello')
        print round(3.3)
```

TIP: To find out what a function does, you can type its name and then a question mark to get a pop up help window. Or, to see what arguments it takes, you can type its name, an open parenthesis, and hit shift+tab.

```
In [ ]: round?
        round(3.14159, 2)
```

TIP: Many useful functions are not in the Python built in library, but are in external scientific packages. These need to be imported into your Python notebook (or program) before they can be used. Probably the most important of these are numpy and matplotlib.

```
In [ ]: # Let's start with numpy
        import numpy as np

In [ ]: # To see what's in a package, type the name, a period, then hit tab
        #np?
        np.deg2rad(90)

In [ ]: # Some examples of numpy functions
        print np.sqrt(4)
        print np.pi  # Not a function, just a variable
        print np.sin(np.pi)
        np.add?
        print np.add(2,3)
```

1.2.3 2.3 Use a method

Before we get any farther into the Python language, we have to say a word about "objects". We will not be teaching object oriented programming in this tutorial, but you will encounter objects throughout Python (in fact, even seemingly simple things like ints and strings are actually objects in Python).

In the simplest terms, you can think of an object as a small bundled "thing" that contains within itself both data and functions that operate on that data. For example, strings in Python are objects that contain a set of characters and also various functions that operate on the set of characters. When bundled in an object, these functions are called "methods".

Instead of the "normal" `function(arguments)` syntax, methods are called using the syntax `variable.method(arguments)`.

```
In [ ]: # A string is actually an object
        a = 'python is fun'
        print type(a)

In [ ]: # Objects have bundled methods
        print a.capitalize()
        print a.replace('fun', 'very fun')
```

1.2.4 Exercise - Conversion

We want to calculate `Height in Metres`. The first thing we want to do is convert from an antiquated measurement system.

To change inches into metres we use the following equation (conversion factor is rounded)
$metre = \frac{inches}{39}$

1. Create a variable for the conversion factor, called `inches_in_metre`.

2. Create a variable (`inches`) for your height in inches, as inaccurately as you want.
3. Divide `inches` by `inches_in_metre`, and store the result in a new variable, `metres`.
4. Print the result

Bonus

Convert from feet and inches to metres.

TIP: To use Latex select cell type "markdown" and two dollar signs before and after the latex equation:

$$c = \sqrt{a^2 + b^2}$$

1.3 3. Collections of variables or objects

While it is interesting to explore your own height, in science we work with larger slightly more complex datasets. In this example, we are interested in the characteristics and distribution of heights. Python provides us with a number of objects to handle collections of things.

Probably 99% of your work in scientific Python will use one of four types of collections: `lists`, `dictionaries`, and `numpy arrays`. We'll look quickly at each of these and what they can do for you.

1.3.1 3.1 Lists

Lists are probably the handiest and most flexible type of container.

Lists are declared with square brackets [].

Individual elements of a list can be selected using the syntax `a[ind]`.

```
In [ ]: # Lists are created with square bracket syntax
        a = ['blueberry', 'strawberry', 'pineapple']
        print a, type(a)
```

```
In [ ]: # Lists (and all collections) are also indexed with square brackets
        # NOTE: The first index is zero, not one
        print a[0]
        print a[1]
```

```
In [ ]: ## You can also count from the end of the list
        print 'last item is:', a[-1]
        print 'second to last item is:', a[-2]
```

```
In [ ]: # you can access multiple items from a list by slicing, using a colon between indexes
        # NOTE: The end value is not inclusive
        print 'a =', a
        print 'get first two:', a[0:2]
```

```
In [ ]: # You can leave off the start or end if desired
        print(a[:2])
        print(a[2:])
        print(a[:])
        print(a[:-1])
```

```
In [ ]: # Lists are objects, like everything else, and have methods such as append
        a.append('banana')
        print a

        a.append(1,2)
        print a

        a.pop()
        print a
```

TIP: A 'gotcha' for some new Python users is that many collections, including lists, actually store pointers to data, not the data itself.

HELP: look into the `copy` module or work with list content

```
In [ ]: a = ['blueberry', 'strawberry', 'pineapple']
        b = a
        print 'original b:', b
        a.pop()
        print 'What is b after we change a ?', b
```

1.3.2 EXERCISE - Store a bunch of heights (in metres) in a list

Tom is 1.8m, Sarah is 1.6m, Brendan is 1.95m, Joan is 1.7m and Catherine in 1.75m. 1. Store these heights in a list called `heights`. 2. Append your own height, calculated above in the variable *metres*, to the list. 3. Get the first height from the list and print it.

Bonus

1. Extract the last value in two different ways: first, by using the index for the last item in the list, and second, presuming that you do not know how long the list is.

HINT: len() can be used to find the length of a collection

1.3.3 3.3 Dictionaries

Dictionaries are the collection to use when you want to store and retrieve things by their names (or some other kind of key) instead of by their position in the collection. A good example is a set of model parameters, each of which has a name and a value. Dictionaries are declared using {}.

```
In [ ]: # Make a dictionary of model parameters
        convertors = {'inches_in_feet' : 12,
                      'inches_in_metre' : 39}

        print convertors['inches_in_feet']
```

```
In [ ]: ## Add a new key:value pair
        convertors['metres_in_mile'] = 1609.34
        print convertors
```

```
In [ ]: # Raise a KEY error
        print convertors['blueberry']
```

1.3.4 3.4 Numpy arrays (ndarrays)

Even though numpy arrays (often written as ndarrays, for n-dimensional arrays) are not part of the core Python libraries, they are so useful in scientific Python that we'll include them here in the core lesson. Numpy arrays are collections of things, all of which must be the same type, that work similarly to lists (as we've described them so far). The most important are:

1. You can easily perform elementwise operations (and matrix algebra) on arrays
2. Arrays can be n-dimensional

Arrays can be created from existing collections such as lists, or instantiated "from scratch" in a few useful ways.

When getting started with scientific Python, you will probably want to try to use ndarrays whenever possible, saving the other types of collections for those cases when you have a specific reason to use them.

```
In [ ]: # We need to import the numpy library to have access to it
        # We can also create an alias for a library, this is something you will commonly see w
        import numpy as np
```

```
In [ ]: # Make an array from a list
        alist = [2, 3, 4]
        blist = [5, 6, 7]
        a = np.array(alist)
        b = np.array(blist)
        print a, type(a)
        print b, type(b)
```

```
In [ ]: # Do arithmetic on arrays
        print a**2
        print np.sin(a)
        print a * b
        print a.dot(b), np.dot(a, b)
```

```
In [ ]: # Boolean operators work on arrays too, and they return boolean arrays
        print a > 2
        print b == 6

        c = a > 2
        print c
        print type(c)
        print c.dtype
        print a.dtype
        print b.dtype
```

```
In [ ]: # Indexing arrays
        print a[0:2]

        c = np.random.rand(3,3)
```

```
        print c
        print c[1:3,0:2]

        c[0,:] = a
        print '\n'
        print c
```

```
In [ ]: # Arrays can also be indexed with other boolean arrays
        a = np.array([2,4,6])
        b = np.array([True, False, True])
        print a[b]

        print a
        print b
        print a > 2
        print a[a > 2]
        print b[a > 2]

        b[a == 3] = 77
        print b
```

```
In [ ]: # ndarrays have attributes in addition to methods
        #c.
        print c.shape
        print c.prod()
```

```
In [ ]: # There are handy ways to make arrays full of ones and zeros
        print np.zeros(5)
        print np.ones(5)
        print np.identity(5)
```

```
In [ ]: # You can also easily make arrays of number sequences
        print np.arange(0, 11, 2)
```

```
In [ ]: # Boolean arrays:
        x = np.ones(5, dtype = bool)
        print x
        x[3] = 0
        print x
        y = np.ones(5, dtype = bool)
        print y
        print np.logical_and(x, y)
        print x.sum()
        print y.sum()
```

1.3.5 EXERCISE 3 - Using Arrays for simple analysis

Revisit your list of heights

1. turn it into an array

2. calculate the mean
3. create a mask of all heights greater than a certain value (your choice)
4. find the mean of the masked heights

BONUS

1. find the number of heights greater than your threshold
2. mean() can take an optional argument called axis, which allows you to calculate the mean across different axes, eg across rows or across columns. Create an array with two dimensions (not equal sized) and calculate the mean across rows and mean across columns. Use 'shape' to understand how the means are calculated.

1.4 4. Repeating yourself

So far, everything that we've done could, in principle, be done by hand calculation. In this section and the next, we really start to take advantage of the power of programming languages to do things for us automatically.

We start here with ways to repeat yourself. The two most common ways of doing this are known as for loops and while loops. For loops in Python are useful when you want to cycle over all of the items in a collection (such as all of the elements of an array), and while loops are useful when you want to cycle for an indefinite amount of time until some condition is met.

The basic examples below will work for looping over lists, tuples, and arrays. Looping over dictionaries is a bit different, since there is a key and a value for each item in a dictionary. Have a look at the Python docs for more information.

```
In [ ]: # A basic for loop - don't forget the white space!
        wordlist = ['hi', 'hello', 'bye']
        for word in wordlist:
            print(word + '!')
        print('test')
```

Note on indentation: Notice the indentation once we enter the for loop. Every idented statement after the for loop declaration is part of the for loop. This rule holds true for while loops, if statements, functions, etc. Required identation is one of the reasons Python is such a beautiful language to read.

If you do not have consistent indentation you will get an `IndentationError`. Fortunately, most code editors will ensure your indentation is correction.

NOTE In Python the default is to use four (4) spaces for each indentation, most editros can be configured to follow this guide.

```
In [ ]: # Indentation error: Fix it!
        for word in wordlist:
            new_word = word.capitalize()
            print(new_word + '!') # Bad indent
```

```
In [ ]: # Sum all of the values in a collection using a for loop
        numlist = [1, 4, 77, 3]

        total = 0
```

```
        for num in numlist:
            total = total + num

        print "Sum is", total
```

```
In [ ]: # Often we want to loop over the indexes of a collection, not just the items
        print(wordlist)

        for i, word in enumerate(wordlist):
            print i, word, wordlist[i]

        j = 0
        for word in wordlist:
            print j, word, wordlist[j]
            j = j+1
```

```
In [ ]: # While loops are useful when you don't know how many steps you will need,
        # and want to stop once a certain condition is met.
        step = 0
        prod = 1
        while prod < 100:
            step = step + 1
            prod = prod * 2
            print step, prod

        print 'Reached a product of', prod, 'at step number', step
```

TIP: Once we start really generating useful and large collections of data, it becomes unwieldy to inspect our results manually. The code below shows how to make a very simple plot of an array. We'll do much more plotting later on, this is just to get started.

```
In [ ]: # Load up pylab, a useful plotting library
        %matplotlib inline
        import matplotlib.pyplot as plt

        # Make some x and y data and plot it
        y = np.arange(100)**2
        plt.plot(y)
```

1.5 5. Making choices

Often we want to check if a condition is True and take one action if it is, and another action if the condition is False. We can achieve this in Python with an if statement.

TIP: You can use any expression that returns a boolean value (True or False) in an if statement. Common boolean operators are ==, !=, <, <=, >, >=. You can also use `is` and `is not` if you want to check if two variables are identical in the sense that they are stored in the same location in memory.

```
In [ ]: # A simple if statement
        x = 7
        if x > 0:
            print 'x is positive'
        elif x < 0:
            print 'x is negative'
        else:
            print 'x is zero'
```

```
In [ ]: # If statements can rely on boolean variables
        x = 1
        test = (x > 0)
        print type(test); print test

        if test == True:
            print 'Test was true'
```

1.6 6. Creating chunks with functions and modules

One way to write a program is to simply string together commands, like the ones described above, in a long file, and then to run that file to generate your results. This may work, but it can be cognitively difficult to follow the logic of programs written in this style. Also, it does not allow you to reuse your code easily - for example, what if we wanted to run our logistic growth model for several different choices of initial parameters?

The most important ways to "chunk" code into more manageable pieces is to create functions and then to gather these functions into modules, and eventually packages. Below we will discuss how to create functions and modules. A third common type of "chunk" in Python is classes, but we will not be covering object-oriented programming in this workshop.

```
In [ ]: # We've been using functions all day
        x = 3.333333
        print round(x, 2)
        print np.sin(x)
```

```
In [ ]: # It's very easy to write your own functions
        def multiply(x, y):
            return x*y
```

```
In [ ]: # Once a function is "run" and saved in memory, it's available just like any other fun
        print type(multiply)
        print multiply(4, 3)
```

```
In [ ]: # It's useful to include docstrings to describe what your function does
        def say_hello(time, people):
            '''
            Function says a greeting. Useful for engendering goodwill
            '''
            return 'Good ' + time + ', ' + people
```

Docstrings: A docstring is a special type of comment that tells you what a function does. You can see them when you ask for help about a function.

```
In [ ]: say_hello('afternoon', 'friends')
```

```
In [ ]: # All arguments must be present, or the function will return an error
        say_hello('afternoon')
```

```
In [ ]: # Keyword arguments can be used to make some arguments optional by giving them a defau
        # All mandatory arguments must come first, in order
        def say_hello(time, people='friends'):
            return 'Good ' + time + ', ' + people
```

```
In [ ]: say_hello('afternoon')
```

```
In [ ]: say_hello('afternoon', 'students')
```

1.7 7. Plotting your data

```
In [ ]: %matplotlib inline
        import matplotlib.pyplot as plt
```

7.1 Time series

```
In [ ]: time = np.arange(0.0, 5.0, 0.1)

        def func(t, x, f):
            return np.exp(-x*t) * np.cos(2*f*np.pi*t)

        plt.figure(figsize=(5,4))
        plt.plot(time, func(time, 1, 1),  '.--b', label = 'x = 1, f = 1')
        plt.plot(time, func(time, 0.5, 0.5), '.--r', label = 'x = 0.5, f = 0.5')
        plt.xlabel(r'time', fontsize=14, color='black')
        plt.ylabel('$e^{-xt} \cos{2 \pi ft}$', fontsize=14)
        plt.legend()
        plt.title('Functions')
        plt.grid(True)
```

7.2 Bar chart In object recognition experiment, participants viewed images of real and scrambled objects. Their recognition accuracy was 80 percent for real images 10.5 percent for scrambled. Plot the results.

```
In [ ]: accuracy = {"real":80.,"scrambled":10.5}

        plt.figure(figsize=(2,4))
        plt.bar(accuracy.keys(), accuracy.values(), width = 0.9, color = 'red')
        plt.ylabel('Accuracy (%)', fontsize=14)
        plt.xticks(fontsize=12)
        plt.ylim(0,100)
```

7.3 Scatter Plot

```
In [ ]: x = np.arange(0.0, 50.0, 2.0)
        y = x ** 1.3 + np.random.rand(*x.shape) * 30.0
        s = np.random.rand(*x.shape) * 800 + 100

        plt.figure(figsize=(3,3))
        plt.scatter(x, y, s, c="g", alpha=0.5, marker='.')
        plt.xlabel("X", fontsize=16)
        plt.ylabel("Y", fontsize=16)
```

```
In [ ]: plt.show()
```

```
In [ ]:
```