Python guide / reference sheet

Overview of Python

Python is an object-oriented programming language that is relatively simple to use and quick to get working. As such, it's well suited to being someone's first or "learning" programming language. In addition to its simple syntax and set of features, it has a very extensive set of libraries, including ones useful for working with matrices/linear algebra and other things important for machine learning, which is why we are using it in these labs.

The notebooks provide code examples and a place where you can play around with Python code, so hopefully will be helpful for learning programming. Note that it's OK if you don't understand everything going on with them. If you run into issues running the notebooks or writing code, please ask.

That said, even though Python is a relatively straightforward language, learning programming for the first time is a steep learning curve, and there's many small details about the way Python works that will inevitably trip you up if this is your first time using the language. Beyond this guide, I recommend taking one of the many Python or Intro Programming courses out there on LinkedIn Learning, which you have access to for free as a UCL student.

Anaconda and Jupyter Notebook

Anaconda is a distribution of Python. Python comes in many versions and with the ability to install a wide range of libraries. Anaconda is a version of Python that comes pre-bundled with many useful libraries, including Jupyter Notebook. As Anaconda is installed on lab machines, I recommend using it (search for the **Anaconda Prompt**). You can also of course install it on your own laptop.

Jupyter Notebook is a software that allows you to create interactive notebooks that embed runnable snippets of code in blocks called *cells*. There are versions for many different programming languages – we'll be using the one for Python. In order for a Jupyter Notebook to run snippets of Python code, it needs to have a Python runtime engine/interpreter running which kicks in when the user runs the code – this is referred to as the *kernel*.

In order to run Jupyter Notebook, start the Anaconda Prompt. Then (optionally) change the directory you're in (**`cd FolderName`** will change directory; use **`dir`** (Windows) or **`ls`** (Mac/Linux) to list directories beneath this one; **`cd`** (no argument, for Windows) or **`pwd`** (Mac/Linux) will print the current working directory (print out the current path)). If you're happy with the directory you're in, type **`jupyter notebook`**.

This will open a tab in your web browser that gives you a view of the directory matching your current directory in the prompt. You can browse the directories in order to find wherever you saved your notebook files (look for .ipynb files). Clicking on them will open a new tab with the interactive notebook.

Alternatively, there are many websites that allow you to open and work with Jupyter notebooks through your web browser *without installing any software* (Jupyter notebooks as a cloud service). In this case, you want to upload the .ipynb file to that site, which will allow you to open and view the notebook. Note that it is also important to upload any files that the Jupyter notebook is dependent on! If there are other files bundled in the zip file for a particular week, they ALL have to be uploaded to the site in the same directory in order for the code to run properly.

Importing modules

At the beginning of a notebook, you will often see a cell containing a bunch of import statements. These statements allow you to make use of external libraries and code files. As such, this cell has to be run once before moving onto the rest of the notebook's contents. In particular, you can use the following syntax to give the libraries a nickname when importing:

```
import numpy as np
import matplotlib.pyplot as plt
```

Normally, in order to use matplotlib's plotting functions, you would have to say `matplotlib.pyplot.show()` (for example). However, because we have imported matplotlib.pyplot as plt, we can just say `plt.show()` as shorthand.

NumPy (documentation: https://numpy.org/doc/stable/user/whatisnumpy.html) is a big, comprehensive mathematical library in Python that provides many useful functions necessary for scientific computation applications, such as linear algebra and machine learning. The most confusing aspect of NumPy, however, is that it has its own way of specifying multidimensional arrays, which are defined and indexed/accessed *in different ways from native Python arrays*. Just a heads up that learning Python that makes use of NumPy arrays can often be very difficult because you have to learn two different types of arrays at the same time, and the arrays you see all over the place are NOT the same as the default/standard Python arrays.

Matplotlib (documentation: https://matplotlib.org/stable/api/index) is a library that provides tools for plotting graphs in Python.

Notes on using Python – Python reference

This section contains very brief notes on how some aspects of Python work. It is not meant as a complete tutorial but more like a basic intro and a reference sheet for you to look back at if you forget about certain parts of Python.

Python documentation: https://www.w3schools.com/python , https://docs.python.org/

Python indentation

Indentation refers to the spaces at the beginning of a code line. In other programming languages, typically, indentation in code is for readability only. However, Python is somewhat unusual in that the indentation has semantic meaning and is thus important for the behavior of the program. For example:

```
if 5 > 2:
  print("Five is greater than two!")
```

Python uses indentation to indicate a block of code. In the above example, the `print` statement (second line) only happens if the `if` statement (first line) evaluates to True. If there were more indented lines inserted after the second one, they would also be considered part of the `if` statement's "block" as well. Indentation also defines the body of functions. In other languages, blocks of code are typically indicated using curly brackets or with begin/end keywords.

What this means is that Python files have to be consistent about the indentation they use throughout. Either tabs and spaces are able to be used as indentation, but you need to choose which one and use it consistently. If spaces are used, then the number of spaces indicating a single indentation has to be consistent. If a line of code is (or is not) meant to be part of a function or if/else/for/while block, but it has been indented as if it were not (or were), then this can cause unexpected/incorrect behavior when running your program including causing an error to be thrown.

Semicolons

In some languages, semicolons are important tokens that indicate the end of a statement and are used for parsing code. Python is unusual in that semicolons *can* be used to mark the end of the statement but it is not necessary. They are entirely optional. Therefore, if you see a stray semicolon at the end of a statement, you can disregard it – it has no particular meaning. Colons at the end of lines, however, do have important meaning and *are* required in order to e.g. indicate the start of certain code blocks as shown in the previous example.

Declaring variables in Python

Python features a very simple/succinct way of declaring variables. (Variables are named pieces of data, which can then be easily manipulated and combined by referring to them by name.)

Example code:

```
a = 14 - 3 # You can put comments after pound symbol as this is ignored
b = "Welcome" # Strings can be defined with single ' or double " quotes
              # Surround multiline strings with triple quotes """text"""
c = 3
if c == 3 or b == "Welcome":
  a = a + 1
  b = b + " to Python" # concatenation operation using + operator
print a
print b
```

Variable types

Python has various data types: numbers (includes integers and floating point numbers), strings, Booleans (`True`/`False`), lists, tuples, and dictionaries. The first two were demonstrated above. The last three are a bit more complicated but useful.

Lists

Lists are a simple sequence of items. In Python, the elements of a list can be *mixed* types. They are defined using square brackets and separated by commas. Example:

```
mylist = ["abc", 34, 4.34, True]
```

Accessing an element:

```
mylist[1] # returns 34
```

Iterating through all the elements of a list:

```
for li in mylist:
  print li # li's value changes with each iteration through the loop
```

There are many ways to write for loops in Python. The above is seen as a characteristically Python-like way to do it. Another way to iterate is to do it over list *indices* rather than list items:

```
for i in range(len(mylist)):
  print mylist[i] # same output as above
```

"`len`" is a function that returns the number of items in a list (here, 4). "`range`" is a function that returns a special kind of range object. When it takes one argument (e.g. `range(4)`) it will output a sequence consisting of the first n numbers starting with zero (e.g. 0, 1, 2, 3). When used with two arguments, the two arguments specify the lower bound and upper bound. Iteration stops short of the upper bound (e.g. `range(8, 12)` will iterate over 8, 9, 10, 11). An optional third argument will specify the step (e.g. `range(1, 11, 2)` will iterate over 1, 3, 5, 7, 9).

Finally, you can easily iterate over *both* indices and items using the `enumerate()` function:

```
for i, li in enumerate(mylist):
  print("Val at index " + str(i) + " is " + str(li))
```

The `str()` function is a convenient function for converting non-string types to strings.

Tuples

Tuples are similar to lists except that they are **immutable** – once their value is set, it can't be altered later on. However, like lists, they also are allowed to be mixed type including collection types. They are defined with regular brackets. Example:

```
mytuple = (23, 'abc', 4.56, (2,3), 'def')
```

Accessing and iteration works the same as with lists. Notably, when you return multiple values from a function, they will be returned as a tuple.

Dictionaries

Dictionaries are collections of comma-separated **key-value** pairs. Key-value pairs associate a *key* (a name or index for looking up data) with a piece of data (the *value*). Like lists, they are mutable. The keys in dictionaries are typically strings, but they can be any immutable type like numbers or tuples. Earlier versions of dictionaries in Python were unordered but nowadays they are ordered. They are defined with curly brackets with pairs in the form "key: value". Example:

```
mydict = {
  "apple": 23,
  "orange": -5.2,
  "banana": "Hello"
}
```

Accessing an element:

```
print (mydict["apple"]) # prints 23
```

Iterating through all the data in a dictionary:

```
for k, v in mydict.items():
  print(k, v)
```

You give two variable names, which will be used to store each key-value pair that is active in each iteration of the loop. This is similar to the way lists work except for the pair of variables needed and the need for the `.items()` call.

Multidimensional Python and NumPy arrays

Matrices form the backbone of many machine learning algorithms and are a focus of these lab practicals. Somewhat confusingly, you can easily create matrices (more generally, *n*-dimensional arrays) both in Python and NumPy, and they both share some functions in common while other behavior is specific only to one type. It is therefore important to carefully note when you are using a Python multidimensional list vs. a NumPy array, as it affects how the variable can be used.

If you understand the previous section on data types, then you can see that you can easily create a piece of data representing a matrix in Python:

```
x = [[1, 2, 3], [4, 5, 6]]
print(x[1]) # prints [4, 5, 6]
print(x[0][2]) # prints 3
```

With just a bit of tweaking, the same can be done with a NumPy array:

```
y = np.array([[1, 2, 3], [4, 5, 6]])
print(y[1]) # prints [4 5 6]
print(y[0][2]) # prints 3
```

Note the syntax which creates y as a NumPy array, not a Python list. When examining code, it's good to be able to tell which type of matrix is being created. Note that both x and y behave almost identically and as you might expect, at least in this example.

NumPy arrays additionally have a lot of functionality that Python arrays don't have, sometimes with specialized operators. For example, NumPy arrays can be sliced:

```
print(y[:, 1]) # prints [2 5]
print(y[1, :]) # prints [4 5 6]

print(x[:, 1]) # Both of these yield a TypeError, although it may be
print(x[1, :]) # possible to get the information with different syntax
```

The colon here indicates "every index." NumPy also implements ordinary matrix multiplication through the `.dot()` function. Example:

```
z = np.array([[7, 8], [9, 10], [11, 12]])
print(y.dot(z)) # prints [[ 58  64]
                #         [139 154]]
print(z.dot(y)) # prints [[ 39  54  69]
                #         [ 49  68  87]
                #         [ 59  82 105]]
```

It is recommended you use NumPy arrays, especially when handling data that you'll be performing extensive calculations on as they have more functionality and calculations involving them are quicker to execute. However, be aware that the lab notebooks sometimes use both types of array, and they resemble each other but don't behave in quite the same way.

Control flow constructs

Python, like most languages, provides ways of specifying standard constructs that allow programmers to intuitively control the flow of their program. These include if/then/else blocks, for loops, and while loops.

If/then/else

Constructs of this type allow blocks of code to be executed only if certain conditions are met.

```
if condition:
  # do work
else:
  # do something else
```

The "else" here executes for everything that fails the "if" condition (and only those things). Indentation is important for grouping lines of code together. "Else if" (`elif` in Python) is a construct that combines "else" and "if" behavior: it only kicks in when previous if(s) have failed AND when the specified conditions are met.

Example:

```
if x == 0:
  # do something
elif x > 3 and x <= 6:
  # do something else
else:
  # do another thing if both of the above conditions have failed
```

Note that multiple conditions can be combined using the plain-English keywords “and” and “or”.

For loops

Several types of for loops were demonstrated in the Lists section above. For loops generally have the following form:

```
for iterator variable in range()/collection variable:
  # Do something
```

The for loop will run once for every value in the range or for every item in the collection, in order.

While loops

While loops run continuously until the conditions at the top of the loop no longer hold true. At that point, the while loop immediately exits. While loops have the following form:

```
while condition:
  # do work
```

Note that it is possible to have while loops that loop infinitely and do not exit.

`break` and `continue`

For both for loops and while loops, execution flow can be further controlled using the `break` and `continue` keywords. These can be placed within the body of a for/while loop, by themselves on their own line. “`break`” immediately exits the loop, moving on to the next line of code after the loop. “`continue`” immediately jumps back to the top of the loop for the next iteration, skipping the execution of any code that happens after the `continue`.

These control constructs can all be nested within each other in any configuration. For example, the following code runs a for loop within an infinite while loop, skips some runs of the for loop, and breaks out of the while loop when some condition is met:

```
counter = 0
while True: # this is an infinite loop!
  for x in mylist:
    if x == "Hi":
      continue # "Hi" items are skipped; loop moves onto next item in mylist
    print("Not hi") # Only does these on non-"Hi" items. Could wrap them in
    counter += 1    # an else but it wouldn't change behavior.

  # This code executes after all the items in mylist have been examined:
  if counter > 25:
    break # This will break out of the while loop

print("We made it out!")
```

In the above example, the for loop will keep on running over and over until “counter” exceeds 25. If it never exceeds 25, then the program will not terminate.

Functions

Functions are named, reusable chunks of code that perform some work. They can optionally take in *arguments* (or *parameters*), which can be used to alter the way a function works while the body of the function itself remains unchanged. Functions have the following structure:

```
def function_name (arg1, arg2, ..., argN):
  <statement>
  return <value>
```

A concrete example:

```
def times(x, y):
  return x*y

# Using above function:
x = times(2, 3) # This x exists in a different scope as the one above, so
                # reusing it here and also above doesn't cause problems
print x # Prints 6
```

The `def` keyword creates a function and assigns it a name, which can then be used elsewhere. Functions do not have to return values, but if they want to, they can send back data to the calling context using the `return` keyword followed by the value to be returned. Multiple values can easily be returned by putting a tuple after the `return`. The types of arguments and return values are not declared.

Note: When arguments pass to functions, the function gets a local copy of the value passed in. We can modify the local copy in the function but those changes cannot be seen outside of the function. There is an exception to that rule and that is when data structures such as lists, sets, and dictionaries are used as arguments.

Python 2 vs. Python 3

One more potentially tricky aspect of Python is that there are two main versions of Python still frequently in use: Python 2 (usually some version like 2.6 or 2.7) and Python 3 (the latest stable version is 3.10). The reason why this is is because Python 3 is mostly but not *entirely* backwards compatible with Python 2. In particular, the main change that tends to cause problems is that the way print statements work. In Python 2, you can print values by using "`print x`" or "`print(x)`", whereas in Python 3, only the latter form with enclosing parentheses is allowed. So if Python trips up on a print statement (perhaps you've copied some code from the internet), it might be because you're running Python 3 and it is rejecting Python 2-style print statements. You can check the version of python you're using with
`python --version`

This Python 2 / Python 3 incompatibility also has some downstream affects, like some libraries are only Python 3 compatible after a certain point, so if you have difficulty installing a later version of a package, it's possible you have Python 2 and a newer version is not available/compatible.

Classes and objects

As an object-oriented language, Python allows the creation and use of custom objects that come with their own regular set of member data and provide member functions that can be performed on objects of this type. These member data and member functions can then be inherited and possibly overridden by child classes (inheritance).

The following code creates a Vehicle class:

```
class Vehicle():
  def __init__(self, args):
    # Put initialization actions here
    self.acc_value = 0; # This creates a member variable and initializes
                        # its value

  def set_acceleration(self, acc_value):
    self.acc_value = acc_value;
```

The function `__init__` is a special function that gets called when a new instance of an object is created (a constructor). Note the way indentation is used to indicate member functions that belong to the object, and again to indicate the body of each of those member functions. The argument "self" passes a reference to this object, which is used e.g. to set member data.

In order to create a new object of type Vehicle, you do this:

```
myVehicle = Vehicle() # args go inside the parentheses
```

And the following code creates a child (subclass) of the Vehicle class called “Car”:

```
class Car(Vehicle):
  def __init__(self, args): # Declaring this function will cause the parent
                            # (Vehicle) init function to be overridden
    super().__init__() # This calls the original method defined by the
                       # parent class (superclass) (optional)
    # Can do extra car-specific actions here as needed
```