Python Re-fresh

Dr Rob Collins

Version 6, 31st July 2022

1 Python Coding re-fresh

1.1 General Introduction

This Jupyter notebook contains some core material on the Python language. We will be using Python as the main programming language on this Machine Learning course. The material in this notebook is *not* intended to be a full Python course - there are many, many widely-available resources to provide that. Rather, this is intended to be a relatively simple 're-fresh' of core Python that will be useful in the Machine Learning course.

You should be familiar with most of the material in this notebook before starting the Machine Learning course. You are encouraged to work through this Jupyter notebook, replicating each code block and satisfying yourself that you understand what the code is doing and how it works. You will need many of the commands, statements and functions that appear in this workbook during the course.

If you find you do not understand any of the material in this workbook then you are encouraged to search for reference / training material on the Internet to help your understanding (teaching Python is outside of the scope of the current course).

Some of this workbook is based on the "Python Book" developed and published by Dave Kuhlman: http://www.davekuhlman.org/python_book_01.html. I have retained the section numbering from the original book to enable cross-referencing. Note however, that Dave Kuhlman's book uses an earlier version of Python (Python 2) which has some important differences to the version you will be using on the course (Python 3). Therfore, you advised to refer to the current notebook, and other more up-to-date sources than the original by Dave Kuhlman.

1.2 Instructions for Students

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook

- Note that 'code blocks' are the grey blocks below that start with In [n]: ... where 'n' is a number

1. **Execute each of the code blocks** to check your understanding
2. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
3. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
4. You **may** add further code blocks to experiment with the commands and try out other things

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

1.3 Preliminaries

The Python built-in command to display information is the 'print()' function

```
print ("one", "two", 'three')
```

In Jupyter notebooks you can also include just the name of a variable as the last line of a code block. The contents of this variable will then be displayed when the code block is executed. Note that lists can contain both numbers and character strings.

```
my_list = [1,2,3,'one','two', 'three']
my_list
```

Python provides a type of for loop to iterate over a list and print out each element in that list:

```
for item in my_list:
    print (item)
```

Python includes various data types including .. integers ('int'), double-precision decimal numbers ('float'), strings ('str') and lists ('list'). You can discover the type of a variable using the 'type' function:

```
a = 1
b = 1.0
c = "hello"
my_complex_number = complex(3,2)
print(type(a), type(b), type(c), type(my_list), type(my_complex_number))
```

You can find help on built-in functions using the Python 'help' function:

```
help(print)
```

1.4 Built-in datatypes

1.4.2 List and Tuples

1.4.2.1 Lists

As shown above, lists can be created explicitely:

```
my_list = [1,2,3,4,5]
my_list
```

The values in a list can be changed. (Using technical language they are 'mutable')

```
my_list[2] = 10000
my_list
```

There are some 'short-cut' expressions that can shorten the defintion of lists. For example, to create a list containing all of the same value:

```
my_list = [0] * 20;
my_list
```

Or create a list with a repeating pattern of data:

```
my_list = [1,2,3,4] * 4
my_list
```

1.4.2.1 Tuples

A tuple is a sequence. A tuple is similar in many ways to a list - however it cannot be modified. (Using technical language it is 'immutable':

```
my_tuple = ("red", "blue", "green")
my_tuple
```

When you execute the following block of code you should get an error message - because Tuples are immutable

```
my_tuple[2] = "BROKEN"    # This is not allowed!
```

1.4.2.1 List Operators

Python includes a powerful set of operators for manupulating lists. This is one of the reasons that Python is a common choice for Machine Learning applications - it is easy to manipulate and 're-shape' data.

Consider two lists:

```
list_1 = [1,2,3,4,5]
list_2 = [10,20,30,40]
```

One can be appended to the end of the other using the '+' operator

```
list_1 + list_2
```

As shown above, lists can be extended in repeating patterns like this:

```
list_1 * 5
```

There is a short-hand operator for appending a list to an existing list:

```
list_1 += list_2
print(list_1)
```

You can access a specific item in a list using an integer index. But note that this index starts at zero! Thus the following line of code, somewhat confusingly accesses the sixth (6th) item in the list:

```
list_1[5]
```

You can also access an item by indexing from end of a sequence, using the '-' symbol:

```
list_1[-3]
```

A common and powerful operator ':' allows you to 'slice' lists into sections. The indexing here is somewhat confusing and rather prone to user-error. Notice that again the index is 'zero based' (the first element is indexed by '0'. But also notice that the number after the ':' operator has an index one greater than index of the last item returned.

This in the following the index '4:7' actually extracts the 5,6 and 7th item!

I would read this as X:Y -> "For index i=X to Y-1":

```
sub_list = list_1[4:7]
sub_list
```

(Which is obviously not what we would expect!)

We can also slice with 'strides'. Strides are 'steps' .. so that you can select 'every 2nd item' or 'every 3rd item' etc.:

```
stride_sub_list = list_1[0:10:2]
print(list_1, '\n',stride_sub_list)
```

You can find out if an item is included a list by using the 'in' operator:

```
if (5 in list_1):
    print ("Yup! 5 is in list 1")
if (6 in list_1):
    print ("Weird!!!!")
```

And determine the length of a list using the 'len' function.....

```
len(list_1)
```

Append an new items to the end of a list using 'append()':

```
list_1.append(99)
print(list_1)
```

Insert an item at a particular index. Note that this does not **replace** an item, but rather **inserts** a new item, shuffling up the other items and extending the length of the list by 1:

```
list_1.insert(3,888)
list_1
```

Remove an item from a list using 'remove()':

```
list_1.remove(4)
print(list_1)
```

The 'pop' function, returns the last item in a list **and also** deletes that item from the list:

```
print(list_1.pop())
```

```
print(list_1)
```

1.4.3 Strings

1.4.4 Dictionaries

A dictionary is a collection, whose values are accessible by key. It is a collection of namevalue pairs. Dictionaries are indicated by curley-brackets: {....}

Dictionaries are a powerful data-structure that have a number of uses in Machine Learning. They are used to easily build complex data-structures and, for example, to provide structured parameters for some Machine Learning library functions.

The order of elements in a dictionary is undefined. But, we can iterate over (1) the keys, (2) the values, and (3) the items (keyvalue pairs) in a dictionary. We can set the value of a key and we can get the value associated with a key.

Note: Keys must be immutable objects: ints, strings, tuples, ...

Literals for constructing dictionaries:

```
d1 = {}
d2 = {'key1': 4, 'key2': 66, }
print(d2['key1'])
```

When iterating over dictionaries, use methods keys(), values() and items(). Example:

```
d = {'a': 111, 'b': 222, 'c': 333}
for key in d.keys():
    print(key)
```

```
for value in d.values():
    print(value)
```

```
for item in d.items():
    print (item)
```

```
for key,item in d.items():
    print('Key = ', key, 'Item = ', item)
```

To test for the existence of a key in a dictionary, use the 'in' operator:

```
d = {'height': 220, 'age': 34, 'gender': 'Female'}
notAKey = 'Fish'
if (notAKey in d):
    print (notAKey, "is a key in d")
else:
    print (notAKey, "is not a key in d")
```

```
aKey = 'gender'
if (aKey in d):
    print (aKey, "is a key in d")
else:
    print (aKey, "is not a key in d")
```

You can often avoid the need for a test by using method get:

```
theData = d.get(notAKey)
if (theData is None):
    print ("No data with that key")
else:
    print ("The data is", theData)
```

```
theData = d.get(aKey)
if (theData is None):
    print ("No data with that key")
else:
    print ("The data for key", aKey, "is", theData)
```

1.4.6 Other built-in types

1.4.6.1 The None value/type The unique value None is used to indicate "no value", "nothing", "nonexistence", etc. There is only one None value; in other words, it's a singleton.

```
flag = None
if flag is None:
    print ("The variable 'flag' is defined as a 'None' value")
if flag is not None:
    print ("The variable 'flag' has a value")
```

1.4.6.2 Boolean values

True and False are the boolean values.

The following values also count as 'false'(for example, when they appear an if statement):

- numeric zero
- None
- the empty string
- an empty list
- an empty dictionary
- any empty container

All other values, including True, act as true values.

1.4.6.3 Sets and frozensets

A set is an unordered collection of immutable objects. A set does not contain duplicates. Sets support several set operations, for example: union, intersection, difference, ...

```
a_set = set()
a_set
```

You can add items to a set using 'add':

```
a_set.add('apple')
a_set
```

```
a_set.add('orange')
print(a_set)
```

Sets only represent the inclusion of single items. Thus adding multiple instances of the same thing to a set does not change it:

```
a_set.add('apple')
a_set.add('apple')
a_set.add('apple')
a_set.add('apple')
a_set.add('apple')

print(a_set)
```

You can add an item to a set using the 'add' function:

```
another_set = set(['fish','meat'])
another_set.add('apple')
another_set
```

Set intersection is achieved using the 'intersection' function:

```
a_set.intersection(another_set)
```

```
print (a_set.union(another_set))
```

1.5 Not Used

1.6 Statements

1.6.1 Assignment statement

Tuple or lists Can be nested. Left and right sides must have equivalent structure.

```
x,y,z = 11,22,33
print(x,y,z)
```

```
[x2,y2,z2] = 11,22,33
print(x2,y2,z2)
```

This feature is sometimes used simulate the enumerated type ('enum') that appears in other languages. I.e. an ordered set of named items:

```
LITTLE, MEDIUM, LARGE = range(1,4)
print(LITTLE)
print(LARGE)
```

Subscription of a sequence, dictionary, etc. Example

```
a = [0,1,2,3,4,5,6,7,8,9]
a
```

```
a[3] = 'something else'
print(a)
```

A slice of a sequence Note that the sequence must be mutable. Example

```
print (a[2:5])
```

Assignment can also cause sharing of an object.

Note that in my (Rob Collins') view whilst this feature can have uses (for example manipulation of complex tree and list data-structures), it seems to often leads to programming defects. That is, it can be useful, but programmers seem to make frequent mistakes when the use it unless they are quite careful:

```
obj1 = [1,2,3,4,5,6,7,8]
obj2 = obj1
obj2[3] = "elephant"
print(obj1)
```

You can also do multiple assignment in a single statement. Example:

```
a = b = 123
print(a,b)
```

You can interchange (swap) the value of two variables using assignment and packing/unpacking:

```
a = 10
b = 20
a, b = b,a
print(a,b)
```

1.6.2 import statement

We will make extensive use of pre-defined libraries during the Machine Learning course. Pre-define libraries give you access to a huge range of powerful functions that go beyond the core funcitonality of Python.

We will cover the core libraries you need to use in a different Jupyter notebook. However, for now, we introduce the basic method for importing and using library functions.

First consider the 'Math' library. Python does not have built-in trigonometric functions. Therefore, before you import the 'Math' library, the function below will give an error:

```
x = math.sin(0.1)
```

However, if you 'import' the Math library, this function should work:

```
import math
x = math.sin(0.1)
x
```

Note that the above 'import' function imports the whole of the 'math' library into your programme. This is often inefficient .. why import a large amount of code that you don't need?

It is thus often better to specifically import only those functions that you need from a library. This is achieved using the 'from' statement:

```
from random import randrange
print (randrange(30))
```

You will also see that developers frequently 'rename' library functions when they import them into their code. They may do this because they have already used the name of the library function in their existing code. However, it is more frequently the case that the renaming just allows a short version of the name to be used in their code:

```
from datetime import datetime as dt
print(dt.now())
```

1.6.3 print statement

We have already been making extensive use of Python's 'print()' function above. As mentioned, you will often not need a print() when using Jupyter Notebooks as, if a variable is included as the last line of a code block on its own, then a formatted version of the variable contents is displayed in your browser. This is particularly useful later when we are printing tables, as Jupyter prints these attractively and allows scrolling.

However, for now, we will further explore the use of 'print()':

As shown above, print can be used to print literal values and the contents of variables:

```
print("This is a literal")
```

```
my_string_variable = "This is a string variable"
print(my_string_variable)
```

```
my_integer_variable = 4
print(my_integer_variable)
```

You can also print out multiple values with a single print() statement:

```
x = 5
z = 4
print ("The result of", x, "+", z , "=", x+z)
```

Notice that hear, Python is adding a space character between each argument that it prints. This can make text readable but for example, if you want to read data into another programme, then you may wish to separate the data with some other character. This can be achieved using the 'sep' argument to print():

```
print ("The result of", x, "+", z , "=", x+z, sep=',')
```

Or maybe, show each data item on a separate line:

```
print ("The result of", x, "+", z , "=", x+z, sep='\n')
```

Sometime you may wish multiple calls to print() to output their values to the same line in the browser. This can be acheived using another argument 'end'. The 'end' argument, defines the character that the print() function displays after the last item of data:

```
message_1 = "Some text "
print(message_1, end="")
message_2 = ".... some more text "
print(message_2, end="")
print(x)
```

1.6.4 if: elif: else: statement

Logical decisions can be written using the 'if', 'else' and 'elif' statements:

```
if (5==3):
    print("Something bad happened in the world of arithmatic")
elif (5==5):
    print("That is correct")
elif (5==2):
    print("Definately not")
else:
    print("something bad happened")
```

1.6.5 for: statement

'For' statements are used to loop - technically speaking 'iterate'.

They can iterate through lists:

```
my_list = [1,2,3,4,5,6]
for a_number in my_list:
    print(a_number)
```

A class that implements the iterator protocol Example:

They can iterate through strings:

```
for a_character in "Hello Dr. Collins":
    print (a_character)
```

You will often need to loop (iterate) through a set number of values. The 'for' statement can be use in conjunction with the 'range()' function to achieve this in a flexible manner:

```
for x in range(10):
    print(x)
```

Note than when an upper bound is given in the range function it iterates up to, but not including, the upper bound:

```
for x in range(5,10):
    print(x)
```

It is also to iterate in steps other than 1:

```
for x in range(0,20,5):
    print(x)
```

Loops can be nested one inside the other:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

1.6.5.1 List comprehensions

Since list comprehensions create lists, they are useful in for statements. (Although, when the number of elements is large, you should consider using a generator expression instead).

A list comprehension looks a bit like a for: statement, but is inside square brackets, and it is an expression, not a statement:

```
a_dict = {'aa': 11, 'cc': -33, 'dd': 55, 'bb': 22}

# Returns all the keys in a dictionary whose associated values are greater than zero ...

filtered = [x[0] for x in a_dict.items() if x[1] > 0]
print(filtered)
```

1.6.5.2 Break and Else Statements

The 'break' and 'else' statements are often useful in a for statement.

The 'for' statement can also have an optional else: clause. The else: clause is executed if the for statement completes normally, that is if a break statement is not executed. Example:

```
list_of_nums = [12,14,43,54,34,23,12,23,43,23,12,23]
for item in list_of_nums:
    if item > 100:
        value1 = item
        break
else:
    value1 = 'not found'
print ('value1:', value1)
```

1.6.6 while: statement

The 'while:' statement is not often used in Python because the for: statement is usually more convenient, more idiomatic, and more natural:

```
count = 0
while count < 5:
    count += 1
    print (count)
```

1.6.7 continue and break statements

The break statement exits from a loop.
The continue statement causes execution to immediately continue at the start of the loop. Both can be used in 'for:' and 'while:' statements.

When the for: statement or the while: statement has an else: clause, the block in the else: clause is executed only if a break statement is **not** executed.

1.6.8 try: except: statement

Exceptions are a systematic and consistent way of processing errors and "unusual" events in Python.

Caught and uncaught exceptions
Uncaught exceptions terminate a program.
The try: statement catches an exception.
Almost all errors in Python are exceptions.

Evaluation (execution model) of the try statement
When an exception occurs in the try block, even if inside a nested function call, execution of the try block ends and the except clauses are searched for a matching exception.

(**Note**: Kuhlman's book is out of date with regards to exception handling .. It focusses on Python 2 wheras we are using Python 3. So this section is based on Chapter 25 of Brian Heinold "A Practical Introduction to Python Programming"

First .. a rather silly error. (The following should give you an error message when you execute it):

```
a = 3
b = 0
c = a/b
print('Hi there')
```

Code is better if it is protected with 'try:' and 'except' statements which catch errors:

```
a = 3
b = 0
try:
    c=a/b
except ZeroDivisionError:
    print('Calculation error')
print('The code ended without crashing')
```

We can have multiple statements in the try block and also and multiple except blocks.

You can test the following code by running it multople times with the values '0', '6' and 'fred':

```
try:
    a = eval(input('Enter a number: '))
    print (3/a)
except NameError:
    print('Please enter a number.')
except ZeroDivisionError:
    print("Can't enter 0, as divide by zero is infinity!.")
```

You are not **forced** to name each exception. The following is legal but is not recommended:

```
try:
    a = eval(input('Enter a number: '))
    print (3/a)
except:
    print('A problem occurred.')
```

It is generally not recommended that you do this, however, as this will catch every exception. This may include defects that maybe you aren't anticipating when you write the code. This will make it hard to debug your program.

Using the exception When you catch an exception, information about the exception is stored in an Exception object. Below is an example that passes the name of the exception to the user:

```
try:
    c = a/0
except Exception as e:
    print(e)
```

1.6.8.1 Try/except/else

You can use an else clause along with try/except. Here is an example:

```
try:
    file = open('filename.txt', 'r')
except IOError:
    print('Could not open file')
else:
    s = file.read()
    print(s)
```

1.6.8.2 try/finally and with/as

There is one more block called 'finally'. Code in the finally block must be executed whether or not an exception occurs. So even if an exception occurs and your program crashes, the statements in the finally block will be executed.

An example of this is closing any files at the end of a program.

```
f = open('filename.txt', 'w')
s = 'hi'
try:
    print("hello")
finally:
    f.close()
```

1.7 Functions, Modules, Packages, and Debugging

1.7.1 Functions

It is often useful to define your own functions when programming. This enables you to develop modular, named code that can be re-used. The use of functions can significantly simplify the structure of computer programmes and enable re-use.

```
def myAdd(x,y):
    return x+y

def twoReturns(x,y):
    return 2 * x, 3 * y

def gizzaList(x,y,z):
    return [x,y,z]

print(myAdd(3,4))
a, b = twoReturns(2,3)
print(a,b)
print(gizzaList(1,2,3))
```

1.7.1.3 Parameters

Default values can be defined for function arguments as follows:

```
def t( num_items = 5):
    for val in range(num_items):
        print (val)
```

```
t(7)
```

If no argument is given to the function, it defaults to the value given in the function definition:

```
t()
```

1.7.1.4 Arguments

When calling a function, values may be passed to a function with positional arguments or keyword arguments.

Positional arguments must placed before (to the left of) keyword arguments.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

1.7.1.5 Local variables

Creating local variables Any binding operation creates a local variable. Examples are:

- parameters of a function
- assignment to a variable in a function
- the import statement

Variable lookup The LGB/LEGB rule The local, enclosing, global, builtin scopes are searched in that order. See: http://www.python.org/dev/peps/pep0227/

The global statement Inside a function, we must use global when we want to set the value of a global variable:

```
a_global_variable_1 = 54
a_global_variable_2 = 45

def myFun():
    a_global_variable_1 = 999        # it isn't really global!!
    global a_global_variable_2
    a_global_variable_2 = 0

print(a_global_variable_1, a_global_variable_2)
myFun()
print(a_global_variable_1, a_global_variable_2)
```