CS-345/M45 Big Data and Machine Learning Lab Class
Python Cheat Sheet(s)

Mike Edwards

The following is a quick cheat sheet to serve as a reference for the basics of the Python 3 programming language. It is intended to get you used to core components of the syntax. It is by no means exhaustive and you are strongly encouraged to dive deeper into using and understanding the language.
The first lab session will help familiarise you with the syntax. Following labs will provide more focus on machine learning principles and you are expected to be competent in understanding the Python language code used.

1 Python

In these labs we will use Python 3, if you are installing Python on your own computer to work from home then make sure you are installing the correct version.

We are also using a number of external packages which can be installed via the `pip` installer. Packages relevant to each lab session will be highlighted on the lab sheet.

If you are working with Python on your own computer then we recommend the usage of virtual environments as a method of controlling and encapsulating the installation of various packages which may conflict with each other. `virtualenv` or `anaconda` are common virtual environments in Python development.

2 Running Python Programs

A Python file is a Python language source file with a `.py` extension. A number of methods to write and execute Python programs are available, including:

1. Text editor and Python console
2. IDLE - a Python IDE
3. PyCharm - an IDE from JetBrains which has a version available to students
4. Jupyter - a "notebook" style browser-based environment

More often than not we will be writing a script which will be the main body of our program, this script will define its own functionality and can also call functions from external packages which are imported.

3 Basic Syntax

Variable initialisation

Python allows you to create variables and assign values of built-in types without explicit type declarations:

```
a = 1
b = 3.14
thisStatementIsFalse = True
```

You can initialise a String with single or double quotes:

```
myStr1 = "Hello"
myStr2 = 'World'
```

Code comments can be provided with the hash (#) symbol:

```
date = "01/02/2020" # States the date in British format
```

Operators

There are a number of built-in arithmetic operators to Python 3. Notably:

```
+       # Addition
-       # Subtraction
*       # Element-wise multiplication
/       # Element-wise division
**      # Raise to power
//      # Floor division
%       # Modulus
@       # Matrix multiplication
=       # Assignment
```

We can combine most of the arithmetic operators with the assignment operator to provide an in-place assignment.

```
+=      # Addition and assign
-=      # Subtraction and assign
*=      # Element-wise multiplication and assign
/=      # Element-wise division and assign
**=     # Raise to power and assign
//=     # Floor division and assign
%=      # Modulus and assign
```

Relational operators:

```
<       # Less than
>       # Greater than
<=      # Less than or equal to
>=      # Greater than or equal to
==      # Equal to
!=      # Not equal to
```

Logical operators:

```
and     # True if both operands are True
or      # False if both operands are False
not     # Inversion of the Boolean value
```

There are other operators, including membership and and identity operators. We refer you to the Python documentation.

Blocks, Scope and Indentation

Python uses indentation level as its syntax for defining block scopes. This can be structured with tabs or spaces. Incorrect indentation can often result in the errors:

```
def myFunc():
print('Inside myFunc')
^
IndentationError: expected an indented block
```

This error has occurred because an indentation is required to define the scope of the function being defined. Increase the indentation level of the second line to fix this.

```
a = 1
        b = 2
        ^
IndentationError: unexpected indent
```

This error has occurred because the indentation level is not consistent, due to the increased indentation level of the second line of code. Remove the indentation to fix this issue.

These errors can be common occurrences while you are first getting used to programming, especially if you are coming from languages with explicit block defining syntax such as braces in Java or C++.

4 Data Structures

Lists

A list is one of the 4 collection data types within Python, the others being tuple, set and dictionary. A list is ordered, changeable and allows duplicate entries. Lists are defined with square brackets as follows:

```
myEmptyList = [] # An empty list
myPopulatedList = [‘Cat’, ‘Dog’, ‘CatDog’]
```

We can access elements of a list using their index (position) within the ordered collection by using square brackets and providing the index as such:

```
myPopulatedList = [‘Cat’, ‘Dog’, ‘CatDog’]
print(myPopulatedList[0])

output: Cat
```

We can edit the value of an element by assigning a new value to the index of a list:

```
myPopulatedList = [‘Cat’, ‘Dog’, ‘CatDog’]
myPopulatedList[0] = ‘Fish’
print(myPopulatedList[0])

output: Fish
```

We can add new elements to the end of a list using the `append` method as follows:

```
myPopulatedList = [‘Cat’, ‘Dog’, ‘CatDog’]
myPopulatedList.append(‘Fish’)

print(myPopulatedList[3])
output: Fish
```

For further information on list functionality we refer to the Python documentation.

Dictionary

A dictionary is another collection data type in Python, and represents an unordered, changeable and indexed collection of elements. Dictionaries are written with braces (curly brackets) and control the access to data through `keys`. We declare a dictionary as:

```
myEmptyDictionary = {}

myPopulatedDictionary = {
        ‘ModuleCode’ : ‘CSC345/M45’,
        ‘NumberEnroled’ : 130,
        ‘OtherKey’ : ‘OtherValues’
        }
```

Instead of indexing into the collection by index as with the `list` structure above, we use the keys of the dictionary to query the values:

```
myPopulatedDictionary = {
        'ModuleCode' : 'CSC345/M45',
        'NumberEnroled' : 130,
        'OtherKey' : 'OtherValues'
        }

print(myPopulatedDictionary['ModuleCode'])

output: CSC345/M45
```

We can assign values to a key in a dictionary:

```
myPopulatedDictionary = {
        'ModuleCode' : 'Not Selected'
        }
myPopulatedDictionary['ModuleCode'] = 'CSC345/M45'

print(myPopulatedDictionary['ModuleCode'])

output: CSC345/M45
```

5 Control Flow

Conditionals

A Python conditional block is given by the `if` keyword, followed by the conditional expression, a colon operator and the statement block on the next indentation level. For example:

```
if a == 10:
        print('a is equal to 10')
```

We can produce an if-else block using the `else` and `elif` keywords:

```
if a == 10:
        print('a is equal to 10')
elif a == 100:
        print('a is equal to 100')
else:
        print('a is not equal to 10 or 100')
```

We can provide logical operations using the keywords `not`, `and`, and `or`:

```
if a == 10 and b == 10:
        print('Both a and b are equal to 10')
```

Loops

We define for loops using the `for` keyword, followed by an iterator variable name, the `in` keyword, an array of values to iterate, and the colon operator. The body of the loop follows on the next indentation level. For example:

```
for i in [1, 2, 3, 4, 5]:
        print(i)

output: 1
        2
        3
        4
        5
```

We can also use the `range` function to construct a vector to iterate across if we know a start and stop value. We can provide the function with a step size if we wish.

```
for i in range(0, 10, 2):
        print(i)

output: 0
        2
        4
        6
        8
```

While loops are defined using the `while` keyword, followed by a conditional expression, the colon operator and the body of the loop on the next indentation level:

```
a = 5
while a > 0:
        print(a)
        a -= 1

output: 5
        4
        3
        2
        1
```

6 Functions

Defining Functions

We can define a function using the keyword `def` followed by the function handle, an open parenthesis, the comma-separated argument list, a close parenthesis and a colon operator. We then provide the statement block of the function on the next indentation level. The `return` keyword can be used to return variables to the calling workspace. For example:

```
def add2Numbers(a, b):
        c = a + b
        return c

print(add2Numbers(1, 1))

output: 2
```

Method chaining is the ability to call a method directly on another method call due to the order of evaluation. For example: we may want to take a String, cast it to uppercase and then split it by whitespace to gain a list of individual words. We can either do this in several steps:

```
str = "This is my string"
str = str.upper() # Cast to uppercase
str = str.split() # Split on whitespace
print(str)
```

or we can use method chaining:

```
str = "This is my string"
print(str.upper().split())
```

These both print the same output: ['THIS', 'IS', 'MY', 'STRING']

Importing Packages and External Functionality

To provide additional functionality from external sources we can use the `import` command to include external code within our current program. In the following example we will import the `NumPy` package to allow us to utilise the NumPy random number generator functionality:

```
import numpy
print(numpy.random.randint(0, 2))
```

We can import a specific component of a package to call it directly:

```
from numpy.random import randint
print(randint(0, 2))
```

We can also import a module to an alias name using the `as` keyword.

```
import numpy as np
print(np.random.randint(0, 2))
```

7 NumPy and Matplotlib

Within the Big Data and Machine Learning labs we will be making use of several external packages, especially NumPy and Matplotlib. NumPy is designed for the efficient handling of multi-dimensional numerical arrays and a range of mathematical operations which are optimized for scientific use. Matplotlib is a plotting package for Python which helps to produce high quality figures and visualisations. In this section we will look at a very small sample of functionality from within these two packages, and we direct you to further documentation for the respective package.

NumPy

We first want to import numpy, and often this is given the alias `np`. We can then call functionality from numpy using the np alis; including sampling random numbers with `numpy.random`, creating N-Dimensional arrays with `ndarray`, and broadcasting common operations to high dimensions. In real world applications we can use higher dimensionality structures to represent a wide range of data; from 4D stacks of images to filter banks in a neural network.

In the following example we will import numpy, create a 3D array of random integers with a shape of 2x3x5, and query its size. We can then index into this ndarray using the list indexing notation (square brackets and positional indices) to print out a slice through the third dimension of the array.

```
import numpy as np

X = np.random.rand(2, 3, 5)

print(X.shape)

output: (2, 3, 5)

print(X[:, :, 0])

output: [[0.14712367 0.80746693 0.64743047]
         [0.24978265 0.85477356 0.51286484]]
```

From this we can see that a numpy ndarray's data is actually formatted as nested lists, providing higher dimensionality structures.

There are numerous attributes associated with the ndarray object; including the `shape`, `ndim`, and `size` attributes.

There are also numerous methods associated with the ndarray object; including `reshape`, `tolist`, `transpose`, `flatten`, `sum`, and `mean`.

A lot of other packages in Python can make use of the numpy package, it's very handy to have installed and to take the time to understand.

`numpy` also has methods available to load from, and save to, disk a `.npy` file containing an ndarray:

```
import numpy as np

x = np.arange(5) # Create array containing integer values [0-5)
np.save('myFile.npy', x)

y = np.load('myFile.npy')
print(y)

output: [0, 1, 2, 3, 4]
```

We refer you to the documentation for NumPy for further details.

Matplotlib

Matplotlib provides two main ways to access its functionality in order to create detailed and finely tuned figures. The first is to use the object-oriented low level functionality, allowing a high degree of control over plot elements. The second is the much more user friendly `pyplot` module, which provided functions which are used to add plot elements to a figure. In this cheat sheet we will look at the `matplotlib.pyplot` functionality in order to create some nice looking plots.

The first figure is a 2D scatter plot with some labelled axes and a title. The second plots several line series plots of the output from different functions, with a background grid and legend describing the series.

```
# A 2D scatter plot of some random data
import matplotlib.pyplot as plt
import numpy as np

data = np.random.randn(150,2) # 150 observations with 2 features

plt.figure() # Create a new canvas
plt.scatter(data[:,0], data[:,1]) # Plot data as a scatter plot

plt.xlabel('Feature 1') # Add x axis label
plt.ylabel('Feature 2')
plt.title('Scatter plot of 2D data')

plt.show() # Display plot to screen
```

```
# A line plot of an exponential growth curve
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 3, 50)
y = np.exp(x)
y2 = np.exp(2*x)

plt.figure()
plt.plot(x, y, label="y = exp(x)") # Plot y against x
plt.plot(x, y2, label="y = exp(2x)") # Plot y2 against x

plt.xlabel('x')
plt.ylabel('y')
plt.title('Exponential curve')

plt.legend()
plt.grid()

plt.show() # Display plot to screen
```

outputs: