Intro to Python

Modules

Modules

- We have already seen some modules such as **re**, **random**, **pathlib** etc.
- Higher level structural building block which can contain, variables, classes, functions and statements
- All **.py** files are modules, the name of a module is the name omitting the **.py** extension
- As we know the **import** keyword is used to bring the functionality of an external module to your own code

```
import MODULE
```

Importing Modules

- Module themselves define a namespace and by importing a module, the user gains access to the name space
- You may either import using the **from** or **import** keywords:

```
import re

from pathlib import Path
```

- Import multiple modules on the same line is possible as well:

```
import MODULE_A, MODULE_B, MODULE_C
```

Import vs from … import

- Simply using **import** keyword alone imports the module as a single object in your name space:

```
import random
random.randint(1,6) #all names accessed via random
```

- Whereas **from … import** imports individual names from the module into the current namespace:

```
from random import randint
randint(1,6) #the randint name has been imported into the namespace
```

from … import *

- **from … import** * will import all available names into your current name space, for example:

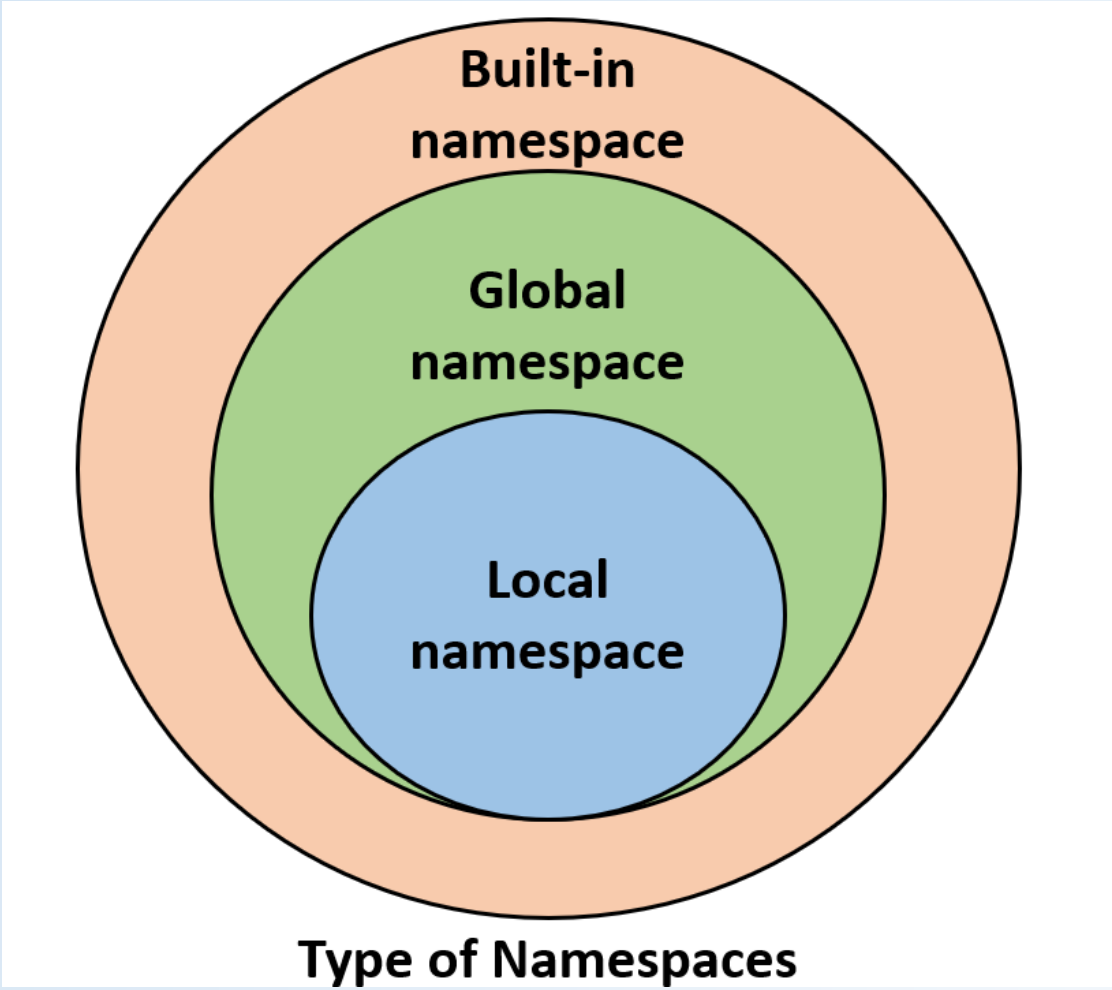
```
from random import *
randint(1,6)
```

- **from** modifies import to allow to import specific names from within an external module.

```
from MODULE import NAME
```

```
from MODULE import NAME_A, NAME_B
```

Name spaces visualised



from … import … **as**

- The main reason for the word **from** is to remove the need to have many references to a module
- Its best used when a module has many names/sub-modules and you only actually need one of them and there is no point of importing the whole module
- The **as** keyword allows to create an alternative name (or alias) for an imported name:

```
import MODULE as MODULE_ALIAS
from MODULE import NAME as NAME_ALIAS
```

- The keyword **as** allows you to import some module and refer to it as another name

Importing Modules

- Whenever an import statement is written, Python will look through a list of directories to find it
- This list can be expected via the **sys** module:

```
import sys
print(sys.path)
```

Output:

```
['C:\\Users\\Lucas\\Desktop', 'C:\\Python\\python38.zip', 'C:\\Python\\DLLs', 'C:\\Python\\lib', 'C:\\Python',
```

Packages

- Larger libraries are sometimes referred to as packages which is essentially a bundle of modules together in a directory:

```
import sounds.effects.noise
```

- Here is an example of importing the sounds package but as well exploring sub packages such as effects which are nested inside of this package.
- This means that somewhere in the Python Path there is a directory called sounds, and under this there is another directory called effects which contains various Python modules, one of which is called noise, notice how the “.” is used as a directory separator.

Modules – Exercise 1

- Create a file called **coin_toss_module.py** containing the coin_toss function we did a some classes ago, sample code attached in zip folder (coin_toss.py)
- Then create a new file called **coin_toss_test.py**
- Call the **coin_toss** function from within the **coin_toss_test.py** file.

Date & time module

- The **datetime** module supplies some classes for extracting the current date and time **(datetime.py),** for example:

```
from datetime import datetime
current_date = datetime.now()
print (current_date)
```

- The above will print the current date and time with exact precision.
- Read more on the **datetime** module here

Installing External Modules

- So far we have seen that modules are powerful and that there are variety of them that allows to harness and enhance the full power of Python.
- The most common external modules were pre-installed with Python, but what happens when we deal with a module that is not already installed?
- So lets learn how to install external modules!

Python Package Managers - **PIP**

- Python package managers make the job much easier to install external modules
- **PIP** is the most common package manager system for Python and can install modules by using one command
- There are modules that can solve all kinds of problems
- Biopython: contains various tools for bioinformatics
- Pandas: module for data manipulation and analysis
- Tensorflow: machine learning library focusing on deep learning and neural nets.

PIP

- PIP allows easy installation of external packages outside of Python with one command
- To get and install PIP follow the instructions here
- Basic use of PIP: **pip install some_package**
- If you have permission issues try: **sudo pip install some_package**
- And to uninstall packages: **pip uninstall some_package**
- It should be installed by default but if not check the link above to install

Numpy

- **Numpy** is an external Python package that make it easy to perform operations on a collection of numbers
- **Numpy** allows for easy use of data manipulation with matrices and linear algebra
- Much faster than using loops traditionally in Python
- Whenever dealing with a list of numbers, then consider using numpy

Numpy - importing

- **Numpy** is commonly imported by the following commands:

```
import numpy
#OR
import numpy as np
```

Numpy - arrays

- Numpy is known for its **numpy** array, when wanting to create an array from a list or tuple:

```
a = np.array([1,2,3,4])
b = np.array((4,3,2,1))
```

- When comparing to lists, numpy arrays are designed to contain elements of the same type and can be specified explicitly:

```
a = np.array([1,2,3,4], np.bool)
```

- Other types can be used as well **np.int**, **np.float** etc. and if not specified numpy will try to guess the type.

Numpy - Arrays

- There are a multitude ways of initializing numpy arrays:

For range of numbers:

```
print(np.arange(1.0, 1.3, 0.1))
```

➔

```
[1.  1.1 1.2 1.3]
```

Zero's:

```
print(np.zeros(4))
```

➔

```
[0. 0. 0. 0.]
```

One's:

```
print(np.ones(4))
```

➔

```
[1. 1. 1. 1.]
```

Numpy Array Operations

- The same common operators we have seen are available with numpy:

```
a = np.arange(1,4)

print(a)
print(3*a)
print(a+a)
print(a*a)
print(a/a)
print(np.cos(a))
```

Output:

```
[1 2 3]
[3 6 9]
[2 4 6]
[1 4 9]
[1. 1. 1.]
[ 0.54030231 -0.41614684 -0.9899925 ]
```

- Note how it automatically applied to each element

Numpy – Random Arrays

- You can create arrays with random numbers
- Random floating point numbers are between 0 and 1 as we have seen in the past:

```
print(np.random.rand(2))
```

```
[0.82193829 0.28537135]
```

- Printing random integers:

```
#4 numbers between 1 and 3 (3 not included)
print(np.random.randint(1,3,4))
```

```
[1 1 1 2]
```

Numpy – Exercise 2

- Create a program that is one line of code that calculates the average of 10,000 throws of the sum of 2 dice
- HINT: Use **np.average()** to calculate the average over an array