An introduction to coding (with Python)

Throughout most of this course, we will be using the programming language **Python** and its libraries for scientific computing to study, implement and test various numerical methods of great relevance to applications. Throughout the course, we will stress programming best practices that can make the coding process less bug-prone and can make scientific codes more readable, re-usable, efficient and reliable.

For most codes made available in this course, an equivalent **Matlab** code will also be provided. Regardless of prior familiarity with these languages, it is useful to compare these: most of these codes will be similar in structure, yet will differ in syntax / grammar.

Towards the end of the course, we will introduce the interface between Python and lower level, compiled language routines (**C, C++, Fortran**) to optimize and parallelize code.

1 What kind of language is Python?

Python is an *interpreted* language, meaning it does not need to be *compiled* before executing it. It can be used *interactively* via an interpreter (python distributions, IPython, Jupyter, Spyder, etc) in which commands and scripts can be executed, or scripts can be run directly in the command line.

Pros

- Rich set of scientific and numerical libraries (e.g. **numpy, scipy, matplotlib**)
- Free, open-source software with an active developer community
- Relatively easy to learn and to produce readable, structured code, "what-you-see-is-what-you-get".
- Flexible scripting language, environments such as **IPython, Spyder, Jupyter, Pycharm, Visual Studio Code**.

Cons

- Not all algorithms in specialized software or toolboxes are included
- For heavy-duty / large-scale computing, compiled languages (**C,C++,Fortran**) are much faster and provide much greater control and potential for optimization.

Python (with numpy/scipy) and **Matlab** share a number of similarities. Being commercial software, **Matlab** enjoys a number of advanced libraries and toolboxes, and can be a pleasant prototyping environment. However, its base language has some restrictions, and unlike **Python**, interfacing with lower level languages can be painful (e.g. through MEX wrappers).

Other languages for scientific computing include **Julia, Scilab, Octave, R,** etc.

1.1 Installing a working Python environment

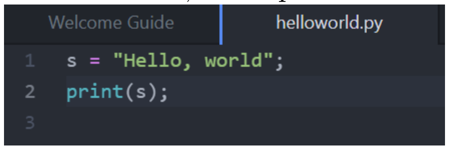
Using your OS package manager, install **Python3** or a well-known Python3 distribution such as **Anaconda, EPD,** or **WinPython**. After installation, test that you are able to call python in the command line (start an interactive session, which should change your command line prompt to >>>).

Interactive environments and text editors in this course, we will focus on writing scripts / functions and running code directly from the command line. While any text editor will do the job, trying freely available code editors such as **Atom** or **Visual Studio Code** is recommended (these will, for instance, present code with line markers, color coding, etc and might point out obvious syntax errors).

Besides the native interactive interface most Python distributions will provide, some of you might want to explore (or might already be familiar with) notebook-like environments such as **IPython** or **Jupyter**. You can use them to try ideas out and enhance your learning and development environment, but be mindful that this course's focus is on writing software / python scripts (and not on the notebook interface).

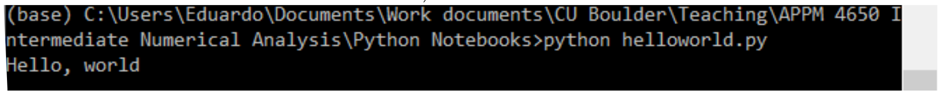
2 Python language: the basics

As a first exercise, let's open our text editor and write our first Python script, *helloworld.py*.



```
s = "Hello, world";
print(s);
```

Once we have saved this text file, we can run it in the command line:



```
(base) C:\Users\Eduardo\Documents\Work documents\CU Boulder\Teaching\APPM 4650 I
ntermediate Numerical Analysis\Python Notebooks>python helloworld.py
Hello, world
```

Each one of these commands can be written and executed at a time in an interactive session. As we will learn later, scripts can be run or *imported* in suc a session, as well.

Commenting code: in Python, the octothorpe (aka hashtag) symbol # is used to write single-line comments. Adding informative comments at the beginning of each code (e.g. an explanation of what this code / function does), as well as throughout the code is of crucial importance to make it readable and user-friendly, for yourself and for others.

2.1 Some basic instructions

Let's look at some basic instructions using the interactive session. Refer to the script *example_basic.py*.

- Variable definition and types (int, float, complex, bool,...)
- Basic operations $(+, -, *, /, **, \ldots)$ and comparisons $(==, <=, !=)$

- Typecasting
- Variable assignment, difference between mutable and immutable objects.

Containers there are several basic types of "containers" in python, in which collections of objects can be stored. Some initial things to be aware of: for a container or dimension of size N, indexing in python goes from 0 to $N-1$ (unlike, say, in Matlab where it starts at 1). It is also possible to access / index arrays backwards, starting with -1 (which would be the last entry).

List: an ordered collection of objects that can be of different types. Refer to the script *example_lists.py*

- Defining lists, such as $colors = ['blue','green','yellow','cerulean','magenta']$
- Accessing elements $colors[0]$, $colors[2]$, $colors[-1]$, etc
- Sublists of regularly spaced elements (*slices*): $colors[start : stop : stride]$
- Setting elements / sublists to new values
- Other operations: append, pop, extend, reverse, concatenate (+), repeat (*), sort.

Strings: character strings often useful for comments, printing, setting options, etc. They can be set between single quote ($'$), double quote (") or triple quote ($'''$ or """) marks, with slightly different functionality. Single and double are almost interchangeable, while triple allows for multiple lines to be printed and is often used in comments or documentation.

- Strings are collections of characters, and as such they can be accessed / sliced like lists can.
- One cannot modify elements directly, but new strings can be created from old ones (e.g. using replace, concatenate, etc)
- Using special formats, strings are extremely useful for many purposes like plotting (using matplotlib).

Other useful containers:

- **Dictionaries:** unordered container mapping key names to values. It is useful to set functions with variable inputs: e.g. $vars = \{'x' : 3572, 'y' : 0.5, 'z' : -10\}$
- **Tuples:** immutable lists, written with parenthesis and comma-separated. Be sure to use brackets if you want lists! e.g. $t = ('blue', 2, 'hello')$;
- **Sets:** unordered collection of unique items, e.g. $s = set(('a','b','c','a')$

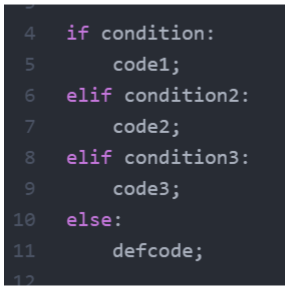
2.2 Control flow / basic logic

We will quickly go over how basic control flow operations are written in Python. These show an important feature of this language: code blocks are delimited by indentation, and so respecting **indentation depth** is compulsory in both scripts and interactive sessions. If you are used to a different language, this might take some getting used to. It is strongly recommended to separate depths of indentation using the Tab key (equivalent to 4 spaces).

Before diving into these, let us briefly go over how booleans / conditions are evaluated in Python:

- Any numerical type equal to 0, any empty container, a boolean equal to False or None will be evaluated as **False**. Anything else will be evaluated as **True**.
- Equality ==, Inequality <=, <, >=, >, Identity (is) and belonging to a collection (in).

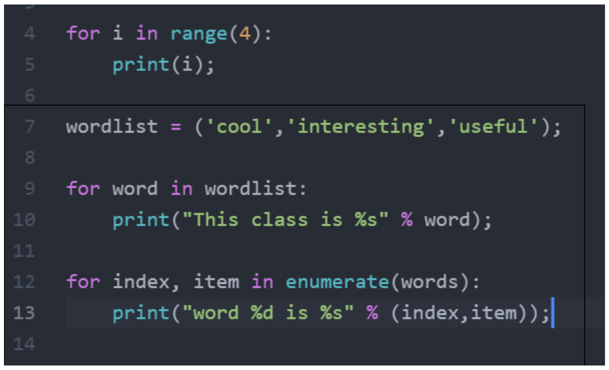
if/elif/else statements: The general formatting is as follows:



```
if condition:
    code1;
elif condition2:
    code2;
elif condition3:
    code3;
else:
    defcode;
```

Let's practice a few examples.

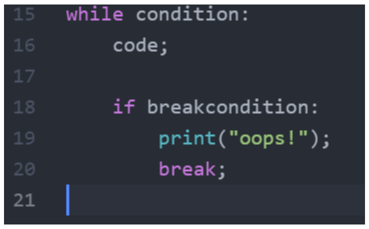
for loops: Somewhat unique to Python is that for loops can iterate over indices in a range, as well as over values in any container: lists, strings, sets, dictionaries, lines on a file, etc. It is even possible to iterate over both! Let's take a look at the following 3 examples: (Note the use of indentation)



```
for i in range(4):
    print(i);

wordlist = ('cool','interesting','useful');

for word in wordlist:
    print("This class is %s" % word);

for index, item in enumerate(words):
    print("word %d is %s" % (index,item));
```

What do each of these do?

while loops: Unlike for loops, which iterate over a set range or container of a given length, while loops iterate until a certain condition is met. As such, they can be dangerous and should be written with care (don't want infinitely looping code).



```
while condition:
    code;

    if breakcondition:
        print("oops!");
        break;

```

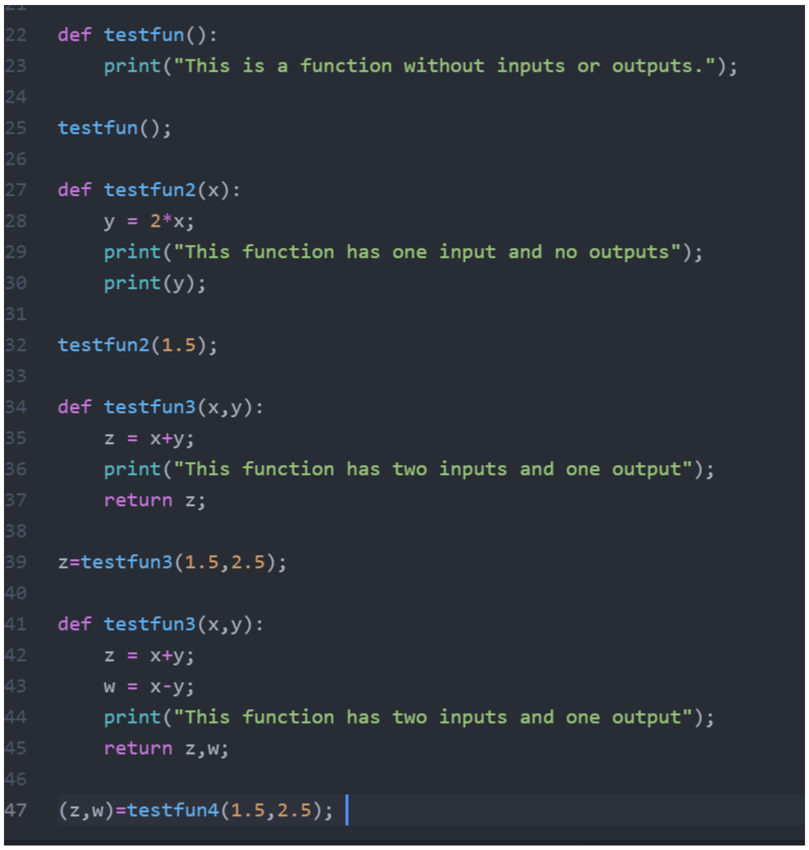
Useful instructions such as continue (to skip one iteration).

2.3 Defining functions

One of the keys to make code more readable and modular, and to reduce bugs / errors is to define functions we can call as necessary. As you may imagine, this also naturally allows us to define mathematical functions with all sorts of inputs and outputs.

If a function is defined on an interactive session, it may be used any time after its definition and until it is cleared or the session is over. On a script, it may be used anytime after it is defined (and until the script finishes running). Finally, we may *import* functions from *modules* and libraries (Python's libraries are examples of this).

Functions are defined with the **def** command, with optional input and output parameters. Note that, once again, the code within a function must be indented.



```
def testfun():
    print("This is a function without inputs or outputs.");

testfun();

def testfun2(x):
    y = 2*x;
    print("This function has one input and no outputs");
    print(y);

testfun2(1.5);

def testfun3(x,y):
    z = x+y;
    print("This function has two inputs and one output");
    return z;

z=testfun3(1.5,2.5);

def testfun3(x,y):
    z = x+y;
    w = x-y;
    print("This function has two inputs and one output");
    return z,w;

(z,w)=testfun4(1.5,2.5);
```

Some useful things to know:

- **Default values** can be set for one or more input parameters, as long as they are at the end of the list of inputs. The order of these arguments does not matter. If these variables are unset when calling the function, the default value is used. These are sometimes called "keyword / named / optional arguments".

Default values are evaluated when the function is defined, not when it is called. Immutable types (e.g. constants) will persist across function calls, mutable types (lists, dictionaries) can be modified inside of the function.

- **Local and Global variables:** variables declared outside the function can be referenced within the function. However, they may not be modified unless they are explicitly declared as global.
- **Variable arguments** Functions can be defined with variable number of both regular (positional) and keyword arguments. An example of this is *def variable_args*$(*args, **kwargs)$
- **Functions in python are objects**, which means they can be assigned to a variable, passed as arguments to other functions, be included as items in a container, etc.

3 Intro to Numpy and Scipy

The core tool to perform numerical computing in python is **numpy**. At its core, it is an extension package for python based on array oriented computing, providing efficient, vectorized operations for multi-dimensional arrays like vectors, matrices and tensors. Because this philosophy and the intended use in scientific computing are close to Matlab's, there are many analogues between numpy and Matlab instructions.

For this class, we will be mainly using routines from numpy and matplotlib (for visualization). Scipy is a library containing more advanced numerical modules (often working in synergy with numpy), and its linear algebra routines are particularly efficient (BLAS / LAPACK under the hood).

Importing libraries: In order to use numpy, we must first *import* it. All numpy functions must then be preceded by *numpy* and a period. As this can make code cumbersome, python allows us to rename this prefix. The standard is to write: *import numpy as np.*

3.1 Numpy arrays

Numpy arrays are lists of numerical values (int, float, complex) as well as others (bool, string) of definite number of dimensions and shape. They are stored as *row vectors*, and interpreted as 1D (vectors), 2D (matrices) or ND (tensors). Let's view some examples, testing attributes such as ndim, shape and len.

Some basic attributes of arrays and their operations include:

- Common functions to create arrays: **arange, linspace, ones, zeros, eye, diag**
- Arrays with random entries (rand, randn)
- Basic array indexing, slicing.
- Array operations: elementwise, reductions, shape manipulation.

- Difference between views and copies.
- Fancy indexing (using boolean and index arrays)

Refer to the examples in the script *examples_numpyarrays.py*.

3.2 Basic visualization using matplotlib and pyplot

We will import the basic set of plotting routines with the command: *from matplotlib import pyplot as plt*. The main function we will be using is plt.plot, a 1D plotting function provided arrays of x and y values and a set of extra keyword arguments.

Some basic functionality includes:

- Changing line color and width, markers.
- Setting x and y limits, ticks, labels, title
- Adding plot legends
- Using subplots (multiple plots in one figure)
- Other kinds of plots

Refer to the examples in the script *examples_basicplots.py*.

4 References and more information

Programming languages are best learnt by doing, and it is expected that you will consult this and many other tutorials and documentation throughout this course. For future reference, here is a list of suggested sources of information:

- **Scipy lectures**: well-documented intro to python, numpy and scipy.
- **Intro to Numpy and Matplotlib**: introduction to numpy for scientific computing, ML and statistics with videos.
- Numpy documentation website.
- **Numpy for Matlab users**: part of the numpy documentation with parallels between Numpy and Matlab commands.