Hands on Machine Learning for Fluid Dynamics

Lecture 2: An Introduction to Python Programming

J. Christophe

07/02/2022

von KARMAN INSTITUTE
FOR FLUID DYNAMICS

Before we start ...

python

This introduction to Python is not covering all concepts you could find in a programming course but ...

.... It includes all the necessary concepts required in the other lectures of this week

- If you are an experienced Python user, be patient, things will get more complicated later...
- If you are not, no worries, we start from the really beginning ...

Content

- Variables and manipulations
- Flow control
- Functions
- Basic OOP

Content

- Variables and manipulations
- Flow control
- Functions
- Basic OOP

Data types in python

Python has several fundamental data types
Some of the most used ones are:

- Numerical: integers, float, complex, bool
- Containers: string, list, dict, tuples, sets
- Special types used in particular packages: numpy array

Data can be assigned to a variable using the `=` operator

Fundamental types:

Fundamental types:

In [1]:

```
i = 2          # integer
f = 2.0        # float
b1 = True      # boolean
b2 = False     # boolean
c = 1.0 -2.0j # complex

print(i, f, b1, b2, c)
print(type(i))
print(type(f))
print(type(b1))
print(type(b2))
print(type(c))
```

```
2 2.0 True False (1-2j)
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'bool'>
<class 'complex'>
```

Fundamental types:

In [1]:

```python
i = 2          # integer
f = 2.0        # float
b1 = True      # boolean
b2 = False     # boolean
c = 1.0 -2.0j # complex

print(i, f, b1, b2, c)
print(type(i))
print(type(f))
print(type(b1))
print(type(b2))
print(type(c))
```

```
2 2.0 True False (1-2j)
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'bool'>
<class 'complex'>
```

Note that Python is a dynamically typed language, no need to define the type before assignment

Fundamental types:

In [1]:

```
i = 2         # integer
f = 2.0       # float
b1 = True     # boolean
b2 = False    # boolean
c = 1.0 -2.0j # complex

print(i, f, b1, b2, c)
print(type(i))
print(type(f))
print(type(b1))
print(type(b2))
print(type(c))
```

```
2 2.0 True False (1-2j)
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'bool'>
<class 'complex'>
```

Note that Python is a dynamically typed language, no need to define the type before assignment

Multiple assignment is also possible:

Fundamental types:

In [1]:

```
i = 2          # integer
f = 2.0        # float
b1 = True      # boolean
b2 = False     # boolean
c = 1.0 -2.0j # complex

print(i, f, b1, b2, c)
print(type(i))
print(type(f))
print(type(b1))
print(type(b2))
print(type(c))
```

```
2 2.0 True False (1-2j)
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'bool'>
<class 'complex'>
```

Note that Python is a dynamically typed language, no need to define the type before assignment

Multiple assignment is also possible:

In [2]:

```
c, d = (i*2, f+2); print(c, d)
```

```
4 4.0
```

Type Casting

If needed, a variable can be converted in other types

Type Casting

If needed, a variable can be converted in other types

In [3]:

```
flt_num = 5.5
str_num = str(flt_num)
print(str_num)
type(str_num)
```

```
5.5
```

Out[3]:

```
str
```

Operations

Addition (+), subtraction (-), multiplication (*), division (/), integer division (//), power (**)

In [4]:

```
1+4, 5-1, 4/3, 4//3, 4**2
```

Out[4]:

```
(5, 4, 1.3333333333333333, 1, 16)
```

In [5]:

```
type(4/3)
```

Out[5]:

```
float
```

In [6]:

```
1.0+4.0, 5.0-1.0, 4.0/3.0, 4.0//3.0, 4.0**2.0
```

Out[6]:

```
(5.0, 4.0, 1.3333333333333333, 1.0, 16.0)
```

Comparison operators

Greater (>), less (<), greater or equal (>=), less or equal (<=), equality (==), identity (is)

In [7]:
```
4 > 2, 2 < 4, 4 > 4, 4 < 4, 4 <= 4, 4 >=4
```

Out[7]:
```
(True, True, False, False, True, True)
```

In [8]:
```
1.0 == 1.0
```

Out[8]:
```
True
```

In [9]:
```
i1 = 2.0
i2 = 2.0
i1 == i2
```

Out[9]:
```
True
```

In [10]:
```
i1 is i2
```

Out[10]:

```
False
```

In [11]:

```python
i1 = i2
i1 is i2
```

Out[11]:

```
True
```

In [12]:

```python
a1 = 'a'
a2 = 'a'
a1 is a2
```

Out[12]:

```
True
```

Boolean operators

In [13]:

```
4 > 2 and 3 > 2
```

Out[13]:

```
True
```

In [14]:

```
not True
```

Out[14]:

```
False
```

In [15]:

```
4 > 2 or 2 > 2
```

Out[15]:

```
True
```

String type

Used to store text in a variable

In [16]:

```
s = "Hello VKI"
type(s)
```

Out[16]:

```
str
```

In [17]:

```
len(s)
```

Out[17]:

```
9
```

String type

Used to store text in a variable

In [16]:
```
s = "Hello VKI"
type(s)
```

Out[16]:
```
str
```

In [17]:
```
len(s)
```

Out[17]:
```
9
```

You can index a character in a string using **[ ]** or slice using **[start:stop]**
Warning: index in python starts from 0 !

In [18]:
```
s[4]
```

Out[18]:
```
'o'
```

In [19]:

```
s[6:9]
```

Out[19]:

```
'VKI'
```

In [20]:

```
s[-2]
```

Out[20]:

```
'K'
```

Lists

Is very similar to strings, except that each element can be of any type

In [21]:

```
l = ['VKI', 2.0, 1+2j]
print(l)
print(type(l))
```

```
['VKI', 2.0, (1+2j)]
<class 'list'>
```

Lists

Is very similar to strings, except that each element can be of any type

In [21]:

```
l = ['VKI', 2.0, 1+2j]
print(l)
print(type(l))
```

```
['VKI', 2.0, (1+2j)]
<class 'list'>
```

on which you can use same slicing techniques to manipulate them

In [22]:

```
l[2]
```

Out[22]:

```
(1+2j)
```

Lists

Is very similar to strings, except that each element can be of any type

In [21]:

```
l = ['VKI', 2.0, 1+2j]
print(l)
print(type(l))
```

```
['VKI', 2.0, (1+2j)]
<class 'list'>
```

on which you can use same slicing techniques to manipulate them

In [22]:

```
l[2]
```

Out[22]:

```
(1+2j)
```

and can be arbitrarily nested

In [23]:

```
nl = [1, 3.0, ['V', 'K', 'I'], 1+2j]
print(nl)
```

```
[1, 3.0, ['V', 'K', 'I'], (1+2j)]
```

Manipulation of lists

In [24]:

```
print(l)
```

```
['VKI', 2.0, (1+2j)]
```

Manipulation of lists

In [24]:

```
print(l)
```

```
['VKI', 2.0, (1+2j)]
```

- Add an element at the end

In [25]:

```
l.append('a'); print(l)
```

```
['VKI', 2.0, (1+2j), 'a']
```

Manipulation of lists

In [24]:
```
print(l)
```

```
['VKI', 2.0, (1+2j)]
```

- Add an element at the end

In [25]:
```
l.append('a'); print(l)
```

```
['VKI', 2.0, (1+2j), 'a']
```

- Modify lists by assigning new values to elements (because list is *mutable*)

In [26]:
```
l[2] = 'p'; print(l)
l[2:4] = [2.1, 'b']; print(l)
```

```
['VKI', 2.0, 'p', 'a']
['VKI', 2.0, 2.1, 'b']
```

Manipulation of lists

In [24]:

```
print(l)
```

```
['VKI', 2.0, (1+2j)]
```

- Add an element at the end

In [25]:

```
l.append('a'); print(l)
```

```
['VKI', 2.0, (1+2j), 'a']
```

- Modify lists by assigning new values to elements (because list is *mutable*)

In [26]:

```
l[2] = 'p'; print(l)
l[2:4] = [2.1, 'b']; print(l)
```

```
['VKI', 2.0, 'p', 'a']
['VKI', 2.0, 2.1, 'b']
```

- Insert an element at a particular position

In [27]:

```
l.insert(2, 4+2j); print(l)
```

```
['VKI', 2.0, (4+2j), 2.1, 'b']
```

Manipulation of lists

In [24]:

```
print(l)
```

```
['VKI', 2.0, (1+2j)]
```

- Add an element at the end

In [25]:

```
l.append('a'); print(l)
```

```
['VKI', 2.0, (1+2j), 'a']
```

- Modify lists by assigning new values to elements (because list is *mutable*)

In [26]:

```
l[2] = 'p'; print(l)
l[2:4] = [2.1, 'b']; print(l)
```

```
['VKI', 2.0, 'p', 'a']
['VKI', 2.0, 2.1, 'b']
```

- Insert an element at a particular position

In [27]:

```
l.insert(2, 4+2j); print(l)
```

```
['VKI', 2.0, (4+2j), 2.1, 'b']
```

- delete an element

In [28]:

```
del l[0]; print(l)
```

```
[2.0, (4+2j), 2.1, 'b']
```

Manipulation of lists

In [24]:

```
print(l)
```

```
['VKI', 2.0, (1+2j)]
```

- Add an element at the end

In [25]:

```
l.append('a'); print(l)
```

```
['VKI', 2.0, (1+2j), 'a']
```

- Modify lists by assigning new values to elements (because list is *mutable*)

In [26]:

```
l[2] = 'p'; print(l)
l[2:4] = [2.1, 'b']; print(l)
```

```
['VKI', 2.0, 'p', 'a']
['VKI', 2.0, 2.1, 'b']
```

- Insert an element at a particular position

In [27]:

```
l.insert(2, 4+2j); print(l)
```

```
['VKI', 2.0, (4+2j), 2.1, 'b']
```

- delete an element

In [28]:

```
del l[0]; print(l)
```

```
[2.0, (4+2j), 2.1, 'b']
```

- use help(list) for more details
- for vector/matrix manipulations of numbers, numpy offers more flexible types (np.array), see later

Tuples

Tuples are very similar to lists, but they are inmutable.

This means that they cannot be changed "on the fly"

In [29]:

```
a = (0, 1, 5) # --> Tuples use (), while lists use []
```

In [30]:

```
a[0] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_28535/3686271957.py in <module>
----> 1 a[0] = 1

TypeError: 'tuple' object does not support item assignment
```

Dictionnaries

Similar to lists, except that each element is key-value pair.

They can contain any types inside, including other dictionaries.

In [31]:

```
data = {'num_int': 12, 'type': 'data', 'num_float': 2.0, 'a_list': [1, 'a', 8], 'a_dict_in_dict':{'a':1,'b':3}}
print(data)
print(type(data))
```

```
{'num_int': 12, 'type': 'data', 'num_float': 2.0, 'a_li
st': [1, 'a', 8], 'a_dict_in_dict': {'a': 1, 'b': 3}}
<class 'dict'>
```

In [32]:

```
data['num_float']
```

Out[32]:

```
2.0
```

Content

- Variables and manipulations
- Flow control
- Functions
- Basic OOP

Flow control

Program blocks are defined by their indentation level, for example:

In [33]:

```
# out of block code
value = 6

# start of block
if value > 5:
    # inside of block
    print(value)

# out of block
print(2.0)
```

```
6
2.0
```

if, elif, else conditional statements

In [34]:

```
number = 3

# If it is smaller than 5 or not
if number < 5:
    print("the number is less than 5")
elif number >= 5 and number <= 10:
    print("the number is between 5 and 10")
else:
    print("the number is greater than 10")
```

```
the number is less than 5
```

Loops

The `for` loop is used on iterable objects (list, dict, string).

In [35]:

```
a = range(5)
print(type(a))
for i in a:
    print(i)
```

```
<class 'range'>
0
1
2
3
4
```

Note that the range function starts at 0 and 5 is not included, so the loop does covers the interval: [0, 5)

Loops

The `for` loop is used on iterable objects (list, dict, string).

In [35]:

```
a = range(5)
print(type(a))
for i in a:
    print(i)
```

```
<class 'range'>
0
1
2
3
4
```

Note that the range function starts at 0 and 5 is not included, so the loop does covers the interval: [0, 5)

Use on lists:

In [36]:

```
meas_techniques = ['PIV', 'LDV', 'hotwire']
for m in meas_techniques:
    print(m)
```

```
PIV
LDV
```

hotwire

Loops

Loops

And if you need to access to the indices when iterating over the list:

In [37]:

```
for i, m in enumerate(meas_techniques):
    print(i, m)
```

```
0 PIV
1 LDV
2 hotwire
```

Loops

And if you need to access to the indices when iterating over the list:

In [37]:

```
for i, m in enumerate(meas_techniques):
    print(i, m)
```

```
0 PIV
1 LDV
2 hotwire
```

Use the zip function when iterating over 2 iterable objects of same size:

In [38]:

```
params = ['param1', 'param2', 'param3']
values = [1.0, 2.0, 3.0]
for p, v in zip(params, values):
    print(p, v)
```

```
param1 1.0
param2 2.0
param3 3.0
```

If they are of different size, it stops at the smaller size

See `itertools` functions `cycle`, `repeat`, `multiply`, etc

To iterate over key-value pairs of a dictionary

In [39]:

```
print(data)
```

```
{'num_int': 12, 'type': 'data', 'num_float': 2.0, 'a_li
st': [1, 'a', 8], 'a_dict_in_dict': {'a': 1, 'b': 3}}
```

In [40]:

```
for key, value in data.items():
    print(key, value)
```

```
num_int 12
type data
num_float 2.0
a_list [1, 'a', 8]
a_dict_in_dict {'a': 1, 'b': 3}
```

While loops

In [41]:

```
i = 0
while i < 3:
    print(i)
    i +=1
```

```
0
1
2
```

Content

- Variables and manipulations
- Flow control
- Functions
- Basic OOP

What is a function?

A function is a collection of statements self contained in a sub-program that can be executed whenever you want in the main program

Why is it important?

- **Avoid duplicating** code snippets by putting all similar snippets in a common place
- **Save typing** and make it easier to change the program later
- **Split long programs** into smaller, more manageable pieces

Some definitions :

- A function takes inputs as **arguments**
- The first line, or the **header**, define the function name and inputs
- The **function body** are all the lines within the function
- The outputs of a function are written in the **return** line

Let’s write our first function

Let's write our first function

In [42]:

```
# h the input to the function is called the function .....
def hH2OtoPa(h): # this first line is defined as the function ....
    # the lines within the function are defined as the function ....
    rho = 997.05 #[kg/m^3] at 25°C
    g = 9.80665 #[m^2/s]
    p = rho*g*h #[Pa]
    # end of the ....
    return p # this is the output of the ....
```

Let's write our first function

In [42]:

```
# h the input to the function is called the function .....
def hH2OtoPa(h): # this first line is defined as the function ....
    # the lines within the function are defined as the function ....
    rho = 997.05 #[kg/m^3] at 25°C
    g = 9.80665 #[m^2/s]
    p = rho*g*h #[Pa]
    # end of the ....
    return p # this is the output of the ....
```

In [43]:

```
#To use a function within a code you must call it
Water_height = 0.0045 # This is the value of the argument
Called_function = hH2OtoPa(Water_height) # this is the call to the function

#display results
print('%.4f m of H2O corresponds to %.5f Pa' %(Water_height, Called_function))
```

```
0.0045 m of H2O corresponds to 43.99974 Pa
```

A function can have multiple inputs/outputs

and the results can be saved as tuples or single variables

--> They are positional: order matters!

A function can have multiple inputs/outputs

and the results can be saved as tuples or single variables

--> They are positional: order matters!

In [44]:

```
def hH2OtoPa(h, rho, g):
    pa = rho*g*h #[Pa]
    psi = pa/6894.76 #[Psi]
    return pa, psi

Water_height = 0.006

# Result saved as a tuple
Result_tuple = hH2OtoPa(Water_height, 997.05, 9.80665)
# Results saved as single variables
Result_Var1, Result_Var2 = hH2OtoPa(Water_height, 997.05, 9.80665)

print('%.4f m of H2O corresponds to %.5f Pa or %.5f Psi' \
      %(Water_height, Result_tuple[0], Result_tuple[1]))
```

```
0.0060 m of H2O corresponds to 58.66632 Pa or 0.00851 P
si
```

Keyword arguments

Some arguments can be defined as optional. If they are not specified, they will take their default value.

Keyword arguments

Some arguments can be defined as optional. If they are not specified, they will take their default value.

In [45]:

```
def hH2OtoPa(h, rho=997.05, g=9.80665): # h is an positional argument while rho, and g are keyword
    pa = rho*g*h #[Pa]
    psi = pa/6894.76 #[Psi]
    return pa, psi

Water_height = 0.01

P1, P2 = hH2OtoPa(Water_height) # Keyword not specified, values taken as default
print('%.4f m of H2O corresponds to %.5f Pa or %.5f Psi' %(Water_height, P1, P2))

P1, P2 = hH2OtoPa(Water_height, rho=1000.0, g=10.0) # Keyword specified
print('%.4f m of H2O corresponds to %.5f Pa or %.5f Psi' %(Water_height, P1, P2))
```

```
0.0100 m of H2O corresponds to 97.77720 Pa or 0.01418 Psi
0.0100 m of H2O corresponds to 100.00000 Pa or 0.01450 Psi
```

Unpacking arguments

We can pass elemental types to functions

In [46]:

```
def hH2OtoPa(h, rho, g):
    pa = rho*g*h #[Pa]
    return pa
```

In [47]:

```
arguments_tuple = (0.01, 997.05, 9.80665)
hH2OtoPa(*arguments_tuple)
```

Out[47]:

```
97.777203825
```

In [48]:

```
arguments_dict = {'rho':997.05, 'g':9.80665, 'h':0.01} # --> Order does not matter!
hH2OtoPa(**arguments_dict)
```

Out[48]:

```
97.777203825
```

More information on unpacking here

This also allows for variable number of function inputs, info here

Local and global variables

The variables used inside the body of a function are the **local variables**, they do not exist outside the function.

On the contrary, variables defined outside the function can be accessed everywhere in a program. Those are called **global variables**.

Local and global variables

Local and global variables

In [49]:

```
rho = 1 # Global variable definition

def hH2OtoPa(h):
    rho = 997.05 #[kg/m^3] at 25°C # local variable
    print('rho - Local variable = %0.1f' %(rho))
    g = 9.80665 #[m^2/s]
    p = rho*g*h #[Pa]
    return p

Water_height = 0.001
Called_function = hH2OtoPa(Water_height)

print('rho - Global variable = %0.1f' %(rho))
print('%.4f m of H2O corresponds to %.5f Pa' %(Water_height, Called_function))
```

```
rho - Local variable = 997.0
rho - Global variable = 1.0
0.0010 m of H2O corresponds to 9.77772 Pa
```

A quick comment about documentation...

... always comment your functions

A quick comment about documentation...

... always comment your functions

In [50]:

```python
def hH2OtoPa(h,rho=997.05,g=9.80665):
    """
    Convert a height in [m] of H2O (h) into a difference of pressure
    express in [N/m^2] (pa) or in pound-force per square inch (psi) computed
    using the Bernoulli equation : Dp = rho * g * h

    h : height of the H20 column [m]
    rho : water density [kg/m^3]
    g: gravitational acceleration [m/s^2]
    return: pa,psi pressure difference express in Pa and Psi
    """
    pa = rho*g*h #[Pa]
    psi = pa/6894.76 #[Psi]
    return pa, psi
```

Print the documentation

Print the documentation

In [51]:

```
#this print the documentation of the function
print(hH2OtoPa.__doc__)
```

```

    Convert a height in [m] of H2O (h) into a differenc
e of pressure
    express in [N/m^2] (pa) or in pound-force per squar
e inch (psi) computed
    using the Bernoulli equation : Dp = rho * g * h

    h : height of the H20 column [m]
    rho : water density [kg/m^3]
    g: gravitational acceleration [m/s^2]
    return: pa,psi pressure difference express in Pa an
d Psi
```

Lambda functions

There is a quick one-line construction of functions that is often convenient to make Python code compact called **lambda function**

Lambda functions

There is a quick one-line construction of functions that is often convenient to make Python code compact called **lambda function**

In [52]:

```
f = lambda x: x**2 + 4  # here lambda is equivalent to the matlab's @ for inline functions.

Xval = 5
print('the function f(%.1f) = %.1f'%(Xval, f(Xval)))
```

```
the function f(5.0) = 29.0
```

Lambda functions

There is a quick one-line construction of functions that is often convenient to make Python code compact called **lambda function**

In [52]:

```
f = lambda x: x**2 + 4  # here lambda is equivalent to the matlab's @ for inline functions.

Xval = 5
print('the function f(%.1f) = %.1f'%(Xval, f(Xval)))
```

```
the function f(5.0) = 29.0
```

In [53]:

```
def f(x):
    return x**2 + 4
print('the function f(%.1f) = %.1f'%(Xval, f(Xval)))
```

```
the function f(5.0) = 29.0
```

Content

- Variables and manipulations
- Flow control
- Functions
- Basic OOP

OOP concept

Procedural programming

Write functions that perform operations on data

OOP concept

Procedural programming

Write functions that perform operations on data

In [55]:

```
myData = np.array([1.0, 2.5, 4.0])
s = mean(myData)
print(s)
```

```
2.5
```

OOP concept

Procedural programming

Write functions that perform operations on data

In [55]:
```
myData = np.array([1.0, 2.5, 4.0])
s = mean(myData)
print(s)
```

```
2.5
```

Vs.

OOP = Oriented-object programming

Create objects that contain both data and functions

In [56]:
```
myData = np.array([1.0, 2.5, 4.0])
d = Data(myData)
s = d.mean()
print(s)
```

```
2.5
```

Why OOP?

- **Encapsulation:** in OOP, you bundle code into a single unit where you can determine the scope of each piece of data.
- **Abstraction:** by using classes, you are able to generalize your object types, simplifying your program.
- **Inheritance:** because a class can inherit attributes and behaviors from another class, you are able to reuse more code.
- **Polymorphism:** one class can be used to create many objects, all from the same flexible piece of code. https://www.codecademy.com/articles/cpp-object-oriented-programming

But OOP is not always the optimal way of implementing! It depends on the scope, complexity, repeatability of task, … so think about the structure of your code before implementing …

An over simplified example

- A **class** is a template/blueprint for objects --> CAR
- An **object** is an instance of a class --> Volvo XC40, Alfa Giulia, Mercedes Class A

When the individual objects are created, they inherit all the attributes and methods from the class.

In real life, the car has:

- **attributes** --> as weight and color
- **methods** --> as drive and brake.

Everything is an object (in the Python World)

In [57]:

```
yo = 1
print(type(yo))
print(yo.real)
print(yo.__abs__())
```

```
<class 'int'>
1
1
```

In [58]:

```
from numpy import array
a = array([1.0, 2.0])
print(type(a))
print(a.size)
print(a.max())
```

```
<class 'numpy.ndarray'>
2
2.0
```

Create a class and definitions

In [59]:

```
class Data:    # Defines a class, having a name starting by a capital letter (usual python convention)
    """Class for data array manipulation"""

    name = 'data'    # Class attribute

    def __init__(self, data):    # Instance constructor — this is called when you create the object
        """Initialize"""
        self.data = data    # Instance attribute

    def mean(self):    # Instance method
        """Perform mean over data vector"""
        return np.sum(self.data)/len(self.data)
```

Class and instance attributes in practice

In [60]:

```
# data sets
myData1 = np.array([1.0, 2.5, 4.0])
myData2 = np.array([2.0, 5.0, 8.0])

# Instantiate objects
d1 = Data(myData1)
d2 = Data(myData2)

print(f"Class attribute of d1: {d1.name}")
print(f"Class attribute of d2: {d2.name}")
print(f"Instance attribute of d1: {d1.data}")
print(f"Instance attribute of d2: {d2.data}")
```

```
Class attribute of d1: data
Class attribute of d2: data
Instance attribute of d1: [1.  2.5 4. ]
Instance attribute of d2: [2. 5. 8.]
```

- The class attributes are the same for all objects
- Each object has its own instance attributes
- Changing class attribute through the class name reflects it to all instances
- Changing class attribute through an instance only reflects it to that particular instance (!!! You overwrite the class attribute !!!)

In [61]:

```
Data.name = 'signal'
print(d1.name)
print(d2.name)
```

```
signal
signal
```

In [62]:

```
d1.name = 'signal2'
print(d1.name)
print(d2.name)
```

```
signal2
signal
```

Inheritance

Inheritance consists in defining a class that inherits all the methods and properties/attributes from another class

- Parent class is the class being inherited from
- Child class is the class that inherits from another class

This allows to re-use parts of code from Parent to Child, and extend functionalities in the Child class compared to Parent

Example of class inheritance

In [63]:

```
class VKIEmployee:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def print_name(self):
        print(f"Name: {self.first_name} {self.last_name}")

julien = VKIEmployee('Julien', 'Christophe')
julien.print_name()
```

```
Name: Julien Christophe
```

In [64]:

```
class Student(VKIEmployee):
    def __init__(self, first_name, last_name, dept):
        VKIEmployee.__init__(self, first_name, last_name)
        self.dept = dept
        self.courses = []

    def add_course(self, course):
        self.courses.append(course)

jack = Student('Jack', 'Jones', 'EA')
jack.print_name()
jack.add_course('NSIP2')
jack.add_course('TSC')
jack.courses
```

```
Name: Jack Jones
```

Out[64]:

```
['NSIP2', 'TSC']
```

In [65]:

```
julien.courses
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_28535/1899347557.py in <module>
----> 1 julien.courses

AttributeError: 'VKIEmployee' object has no attribute 'courses'
```