Python - Introduction

Python

Python is a **Programming language**:

1. **General-purpose** - write software in variety of application domains
2. **High-Level** - abstraction from the details of the computer
3. **Multi-Paradigm** - procedural, object-oriented, ecc.
4. **Interpreted** - the interpreter shall be installed
5. **Open Source** - free to use and modify
6. **Cross-Platform** - as it is interpreted

History

- **1990**: First implementation by Guido Van Rossum
- **2001**: Python Software Foundation License (Python 2.1)
- **2008**: Modern Python 3.0 is released
- **Today**: Python 3.14 released

Other Information

- Documentation: https://docs.python.org/3/
- Exercises: https://www.w3resource.com/python-exercises/
- Installation: https://www.python.org/downloads/

Is Python trendy? https://www.tiobe.com/tiobe-index/.

In [1]:
```
!python3 --version
```

```
Python 3.10.12
```

Hello world

In [2]:
```
import builtins as blt

# Main
if __name__ == '__main__':
    blt.print('Hello World!')
```

```
Hello World!
```

- **Modules** - single file with group of functionalities
- **Packages** - group of Modules

In [ ]:
```
import mdlName as mdlAlias
```

Readability

> Readability of the code is **mandatory**, avoid Spaghetti Code!

Python has the *Python Enhancement Proposal* (**PEP20**) rules for "*beautiful coding*"

1. Beautiful is better than ugly
2. Explicit is better than implicit
3. Simple is better than complex
4. ...

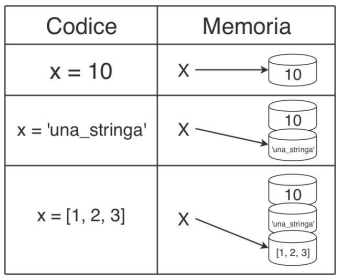
And there exists different **Style Guidelines**, for example

- `PEP08` : https://peps.python.org/pep-0008/

Basics

Variables

> **NOTE**: In Python all the variables are **"pointers"** without a fixed type.

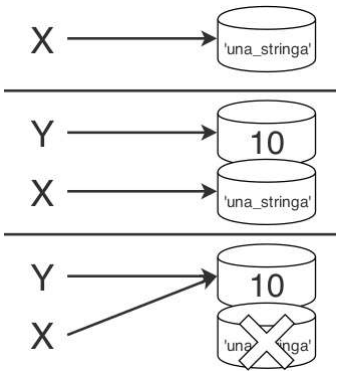


In [3]:
```
# Definition of the variable --> Python manages the memory with the garbage coll
x = None # point to nothing
print("Type", type(x), "Value", x)
x = 10 # point to int variable
print("Type", type(x), "Value", x)
x = 13.2 # point to double variable
print("Type", type(x), "Value", x)
x = 'a string' # point to string variable
print("Type", type(x), "Value", x)
x = [1, 2, 3] # point to an array
print("Type", type(x), "Value", x)
```

```
Type <class 'NoneType'> Value None
Type <class 'int'> Value 10
Type <class 'float'> Value 13.2
Type <class 'str'> Value a string
Type <class 'list'> Value [1, 2, 3]
```

Garbage collector

Automatic memory management with the Garbage Collector (**GC**).



> **NOTE**: Two variables can refers to the **same memory location**.

Blocks

The Block is the **basic unit** of the code

In [4]:
```
# Block identified by identation
# When the block is finished, the variables are deleted by the GC
y: int = 0;
def ABlock():
    x: int = 0;
    # Here x exists

# Here x does not exists
# Here y exists
print(y)
print(x)
```

```
0
[1, 2, 3]
```

> **GARBAGE COLLECTOR**: All the variables declared in a block are automatically deleted at the end of the block.

Basic Conditionals

Syntax for basic conditionals `if`, `else`, `elif`.

- Boolean operators `and`, `or`, `!`

> **REMARK:** Pay attention to indentation (**blocks**) and colon `:`.

In [6]:
```
petType = "fish"

if petType == "dog" or petType == "Dog": # The colon are very important
```

```
    print("You have a dog.")
elif petType == "fish" and petType != "hamster":
    print("You have a fish")
else:
    print("Not sure!")
```

```
You have a fish
```

The switch case

> Before **Python 3.10** the switch-case operator did not exist.

Switch operator is the `match` operator

In [7]:
```
lang = input("What's the programming language you want to learn? ")

match lang:
    case "JavaScript":
        print("You can become a web developer.")
    case "Python" | "python":
        print("You can become a Data Scientist")
    case "PHP":
        print("You can become a backend developer")
    case "Solidity":
        print("You can become a Blockchain developer")
    case item if item in ["Java", "java"]:
        print("You can become a mobile app developer")
    case _:
        print("The language doesn't matter, what matters is solving problems.")
```

```
What's the programming language you want to learn?  PHP
You can become a backend developer
```

Loops - while, for

Syntax for loops `while`, `for`.

> **REMARK:** Pay attention to indentation (**blocks**) and colon `:`.

In [8]:
```
i = 1
while i < 6:
    if i % 2 == 0:
        i += 1
        continue
    print(i)
    if i == 3:
        break
    i += 1

for i in range(1,6):
    if i % 2 == 0:
        continue
    print(i)
    if i == 3:
        break
```

```
1
3
1
3
```

Functions

Syntax for basic `functions`:

In [11]:
```python
# void function - Declaration and Implementation
def printTest(): # The keyword "def" defined the function
    print("print test")

# argument function - Declaration and Implementation
# This is a function with two input value that also have default value 0
def sumAndDiff(a = 0, b = 0): # This is the signatude of the function
    return (a + b), (a - b)

# main
if __name__ == '__main__':
    printTest()
    [s, d] = sumAndDiff(1, 3)
    print("Sum:", s, "Diff:", d)
```

```
print test
Sum: 4 Diff: -2
```

Function **arguments** can be:

- **Immutable**: copy of original variable -> pass to the function by value
- **Mutable**: the original variable -> pass to the function by pointer

> **REMARK**: you cannot choose if an argument is mutable or not. It depends from the variable type. Example: `int` is Immutable

In [12]:
```python
def sumABC(a, b, c):
    b = a + c
    print(type(a))
    match a:
        case int():
            a = 300
        case float():
            a = 258.55
        case str():
            a = "HELLO"
        case list():
            a[0] = 1528
        case _:
            raise Exception("Not implemented yet")

if __name__ == '__main__':
    a = 10
    b = 0
    sumABC(a, b, 1)
    print("a:", a, "b:", b) # b is not 11!
```

```
    a = 10.2
    b = 0.0
    sumABC(a, b, 1.2)
    print("a:", a, "b:", b) # b is not 11.4!

    a = "a"
    b = ""
    sumABC(a, b, "c")
    print("a:", a, "b:", b) # b is not "ac"!

    a = [1,2]
    b = []
    sumABC(a, b, [3])
    print("a:", a, "b:", b) # b is not [1,2,3]!
```

```
<class 'int'>
a: 10 b: 0
<class 'float'>
a: 10.2 b: 0.0
<class 'str'>
a: a b:
<class 'list'>
a: [1528, 2] b: []
```

Generic Types

Python works with **generic types**

> **DUCK TYPING**: "If it walks like a duck and it quacks like a duck, then **it must** be a duck".

In [13]:
```
def maxFunc(a, b):
    return a if a > b else b

def maxFuncType(a, b, required_type):
    return "ERROR" if (required_type is str) else required_type(maxFunc(a, b))

if __name__ == '__main__':
    print(maxFunc(10.2, 12.9)) # Output: 12.9
    print(maxFuncType(10.2, 12.9, int)) # Output: 12
    print(maxFunc(10, 12)) # Output: 12
    print(maxFuncType("T1", "T2", str)) # Output: ERROR
```

```
12.9
12
12
ERROR
```

Casting Operators

A **cast** is a special operator that forces one data type to be converted into another.

> **REMARK**: Pay attention to the **rounding number** operations.

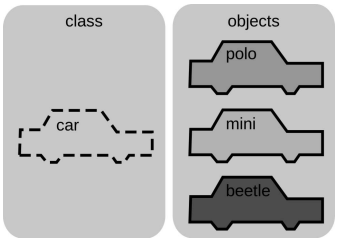
In [14]:
```
# Cast of numbers
a: int = int(21.99399)
b: int = int(10.20)
# Cast to string
c: str = str(21.99399)
print("a:", a, "b:", b, "c:", c)
```

```
a: 21 b: 10 c: 21.99399
```

Object Oriented

Object-Oriented programming (**OOP**) is based on the concept of **class** and **object**.

- **Class** - the definition for
  - **attributes** - the properties
  - **methods** - the procedures
- **Object** - instance of a class



In [15]:
```
class Person:
    """
    A class used to represent a Person

    Attributes
    ----------
    _name : str
        the name of the person
    _age : int
        the age of the person
    """

    def __init__(self, name, age):
        self._name = name # name attribute
        self._age = age # age attribute

    def introduce_yourself(self): # method example
        """Prints the name and the age of the person"""
        print("My name is", self._name, "and I am", self._age)

if __name__ == '__main__':
    help(Person)

    p1 = Person("John", 36)
    p1.introduce_yourself()
```

```
Help on class Person in module __main__:

class Person(builtins.object)
 |  Person(name, age)
 |
 |  A class used to represent a Person
 |
 |  Attributes
 |  ----------
 |  _name : str
 |      the name of the person
 |  _age : int
 |      the age of the person
 |
 |  Methods defined here:
 |
 |  __init__(self, name, age)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  introduce_yourself(self)
 |      Prints the name and the age of the person
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

```
My name is John and I am 36
```

Constructor and Destructor

- An object is always created using a method called **Constructor** `__init__`.
- An object is always deleted using a method called **Destructor** `__del__`.

In [16]:
```
class Something:
    def __init__(self):
        print("Creating...")
    def __del__(self):
        print("Removing...")

if __name__ == "__main__":
    test = Something()
```

```
Creating...
```

Linear Algebra - `numpy` and `scipy`

`NumPy` and `SciPy` are Python libraries used for working with linear algebra (arrays, matrices, linear systems...).

> **NOTE**: NumPy and SciPy are written partially in Python, but most of the parts that require fast computation are written in C or C++.

In [17]:
```
import numpy as np
import scipy as sp

if __name__ == "__main__":
    print(np.__version__)
    print(sp.__version__)
```

```
2.2.3
1.15.2
```

Arrays

The array class in `NumPy` is called `ndarray`.

- The size of the array is given by the `shape` attribute.

> It supports array of n-dimension, taken with attribute `ndim`.

The sparse array library in `SciPy` is called `sparse`

> It supports different formats. Usually we use `csr_matrix` Compressed Sparse Row matrix.

In [20]:
```
# How to create n-dimensional array
import numpy as np
import scipy.sparse

if __name__ == "__main__":
    arr0D = np.array(42); # Zero dimension
    print("Array 0D Dim:", arr0D.ndim, "size:", arr0D.shape, ":\n", arr0D) # Wit
    arr1D = np.array([1, 2, 3, 4, 5]) # One dimension
    print("Array 1D Dim:", arr1D.ndim, "size:", arr1D.shape, ":\n", arr1D)
    arr2D = np.array([[1, 2, 3, 4, 5],[1, 2, 3, 4, 5]]) # Two dimension
    print("Array 2D Dim:", arr2D.ndim, "size:", arr2D.shape, ":\n", arr2D)
    arr5D = np.array([1, 2, 3, 4], ndmin=5) # And so on
    print("Array 5D D:", arr5D.ndim, "size:", arr5D.shape, ":\n", arr5D)

    row_ind = np.array([0, 1, 1, 3, 4])
    col_ind = np.array([0, 2, 4, 3, 4])
    values = np.array([1, 2, 3, 4, 5], dtype=float)
    # To build sparse matrices
    S = scipy.sparse.csr_matrix((values, (row_ind, col_ind)))
    print("S\n", S)
    print(type(S))
```

```
Array 0D Dim: 0 size: () :
 42
Array 1D Dim: 1 size: (5,) :
 [1 2 3 4 5]
Array 2D Dim: 2 size: (2, 5) :
 [[1 2 3 4 5]
 [1 2 3 4 5]]
Array 5D D: 5 size: (1, 1, 1, 1, 4) :
 [[[[[1 2 3 4]]]]]
S
 <Compressed Sparse Row sparse matrix of dtype 'float64'
       with 5 stored elements and shape (5, 5)>
  Coords        Values
  (0, 0)        1.0
  (1, 2)        2.0
  (1, 4)        3.0
  (3, 3)        4.0
  (4, 4)        5.0
<class 'scipy.sparse._csr.csr_matrix'>
```

Array element access

- Square brackets `[]` are used to access to array elements;
- Colon symbol `:` is used to get more than one element.

> **NOTE** The indices always starts from 0.

In [21]:
```
import numpy as np

if __name__ == "__main__":
    arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
    print("matrix\n", arr)
    print("5th element on 2nd row:", arr[1, 4])
    print("Last element:", arr[1, -1])
    print("Access to first two elements of second row:",arr[1,0:2])
    print("Access to last two elements of all the rows:\n",arr[:,3:]) # To acces
    print("Access to first 4 elements of all the rows:\n",arr[:,0:-1])
    print("Access to even columns:\n",arr[:,0:6:2])
```

```
matrix
 [[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
5th element on 2nd row: 10
Last element: 10
Access to first two elements of second row: [6 7]
Access to last two elements of all the rows:
 [[ 4  5]
 [ 9 10]]
Access to first 4 elements of all the rows:
 [[1 2 3 4]
 [6 7 8 9]]
Access to even columns:
 [[ 1  3  5]
 [ 6  8 10]]
```

Copy vs View

- The `copy` method is used to create a copy of the array to an other variable.
- The `reshape` method is used to read the array into an other shape.

> **REMEMBER**: In Python all the variables are **pointers**.

The `base` attribute gives you the original array. For the original array the property is empty.

In [22]:
```
import numpy as np

if __name__ == "__main__":
    a = np.array([1, 2, 3, 4, 5, 6]) # Mutable
    b = a # We refere to the same piece of memory
    c = a.copy() # Here we have to difference portion of memory, two copy of the
    d = a.reshape(2, 3) # a, b, d point at the same portion of memory
    a[0] = 42

    print("a:", a, "b", b, "c", c)
    print("a.base", a.base, "b.base", b.base, "d.base", d.base, "d\n", d)
```

```
a: [42  2  3  4  5  6] b [42  2  3  4  5  6] c [1 2 3 4 5 6]
a.base None b.base None d.base [42  2  3  4  5  6] d
 [[42  2  3]
 [ 4  5  6]]
```

Array iterations

There are different way to iterate on an array

In [23]:
```
import numpy as np

if __name__ == "__main__":
    arr = np.array([[1, 2, 3], [4, 5, 6]])

    for x in arr: # iteration on rows
        print(x)

    for x in arr: # iteration on each element
        for y in x:
            print(y)

    for x in np.nditer(arr): # iteration on each element
        print(x)

    for idx, x in np.ndenumerate(arr): # iteration on each element with index
        print(idx, x, arr[idx])
```

```
[1 2 3]
[4 5 6]
1
2
3
4
5
6
1
2
3
4
5
6
(0, 0) 1 1
(0, 1) 2 2
(0, 2) 3 3
(1, 0) 4 4
(1, 1) 5 5
(1, 2) 6 6
```

Other array operations

- `concatenate` : join togheter two arrays, also in different dimensions;
- `hstack` , `vstack` , `dstack` : concatenation of arrays in different directions;
- `array_split` : split and array in sub-arrays;
- `hsplit` , `vsplit` , `dsplit` : the opposite of stack functions;
- `where` : search elements in an array, returns the index and can be use for filtering
- `sort` : sort an array

In [24]:
```
import numpy as np

if __name__ == "__main__":
    arr1 = np.array([1, 2, 3])
    arr2 = np.array([4, 5, 6])
    print("concatenate -", np.concatenate((arr1, arr2)))
    print("hstack -", np.hstack((arr1, arr2)))
    print("vstack -", np.vstack((arr1, arr2)))
    print("dstack -", np.dstack((arr1, arr2)))

    arr1 = np.array([[1, 2], [3, 4]])
    arr2 = np.array([[5, 6], [7, 8]])
    print("concatenate axis 0 -", np.concatenate((arr1, arr2), axis=0))
    print("concatenate axis 1 -", np.concatenate((arr1, arr2), axis=1))

    arr = np.array([1, 2, 3, 4, 5, 6])
    print("array_split -", np.array_split(arr, 3))
    arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],
    print("vsplit -", np.vsplit(arr, 3))

    arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
    print("where -", np.where(arr % 2 == 1, True, False))
    print("filter -",arr[np.where(arr % 2 == 1, True, False)])

    arr = np.array([3, 2, 0, 1])
    print("sort - ", np.sort(arr))
```

```
concatenate - [1 2 3 4 5 6]
hstack - [1 2 3 4 5 6]
vstack - [[1 2 3]
 [4 5 6]]
dstack - [[[1 4]
  [2 5]
  [3 6]]]
concatenate axis 0 - [[1 2]
 [3 4]
 [5 6]
 [7 8]]
concatenate axis 1 - [[1 2 5 6]
 [3 4 7 8]]
array_split - [array([1, 2]), array([3, 4]), array([5, 6])]
vsplit - [array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
where - [ True False  True False  True False  True False]
filter - [1 3 5 7]
sort -  [0 1 2 3]
```

Matrix operations

- `rand` : generate random matrix
- `transpose` or `T` : transpose of a matrix
- `@` : matrix multiplication
- `*` , `/` , `**` : element-wise multiplication, division and exponential
- `zeros` , `ones` , `eye` , `diag` : create zero, all ones, identity and diagonal matrix

In [26]:
```
import numpy as np

if __name__ == "__main__":
    A =  np.trunc(np.random.rand(3, 3) * 10.0 + 1.0)
    B =  np.trunc(np.random.rand(3, 3) * 10.0 + 1.0)
    print("A\n", A, "\nB\n", B)
    print("A.T\n", A.T, "\nB.transpose\n", B.transpose())
    # Important: matrix moltiplication!!
    print("A @ B\n", A @ B, "\nA * B\n", A * B, "\nA / B\n", A / B, "\nA ** 2\n"
    print("zeros\n", np.zeros((2, 2)), "\nones\n", np.ones((2, 2)), "\neye\n", n
    print("diag 0\n", np.diag([1,2,3], 0), "\ndiag 1\n", np.diag([1,2,3], 1), "\
```

```
A
 [[ 7. 10.  2.]
 [10. 10.  6.]
 [ 5.  1.  7.]]
B
 [[ 6.  7. 10.]
 [ 9.  9.  5.]
 [ 8.  1.  8.]]
A.T
 [[ 7. 10.  5.]
 [10. 10.  1.]
 [ 2.  6.  7.]]
B.transpose
 [[ 6.  9.  8.]
 [ 7.  9.  1.]
 [10.  5.  8.]]
A @ B
 [[148. 141. 136.]
 [198. 166. 198.]
 [ 95.  51. 111.]]
A * B
 [[42. 70. 20.]
 [90. 90. 30.]
 [40.  1. 56.]]
A / B
 [[1.16666667 1.42857143 0.2       ]
 [1.11111111 1.11111111 1.2       ]
 [0.625      1.         0.875     ]]
A ** 2
 [[ 49. 100.   4.]
 [100. 100.  36.]
 [ 25.   1.  49.]]
zeros
 [[0. 0.]
 [0. 0.]]
ones
 [[1. 1.]
 [1. 1.]]
eye
 [[1. 0.]
 [0. 1.]]
diag 0
 [[1 0 0]
 [0 2 0]
 [0 0 3]]
diag 1
 [[0 1 0 0]
 [0 0 2 0]
 [0 0 0 3]
 [0 0 0 0]]
diag -1
 [[0 0 0 0]
 [1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]]
```

Linear algebra

- `np.linalg.norm` : norm of array and matrices

- `np.linalg.inv` , `linalg.pinv` : inversion and pseudo-inversion of matrix
- `np.linalg.matrix_rank` : rank of the matrix
- `np.linalg.solve` , `linalg.lstsq` : solvers for square/rectangular linear system
- `sp.linalg.lu` : PA = LU decomposition, returns `P,L,U`
- `np.linalg.cholesky` : Cholesky decomposition, returns `L`
- `np.linalg.qr` : QR decomposition, returns `[Q, R]`
- `np.linalg.svd` : svd decomposition, returns `[U, S, Vh]` , with `V=Vh.T`
- `np.linalg.eig` , `scipy.sparse.linalg.eigs` : eigenvalues and eigenvectors, returns `D,V`

In [ ]:
```
import numpy as np
import scipy.linalg
import scipy.sparse.linalg

if __name__ == "__main__":
    B =  np.trunc(np.random.rand(3, 3) * 10.0 + 1.0)
    A = B.T @ B
    b = A.sum(axis=1)
    print("A\n", A)
    print("A.sum axis 0:", b, "A.sum axis 1:", b)
    print("l2 norm(b):", np.linalg.norm(b), "fro norm(A):", np.linalg.norm(A))
    print("inv(A)*A=I\n", np.linalg.inv(A) @ A)
    print("rank(A)", np.linalg.matrix_rank(A))
    print("x = inv(A) * b =", np.linalg.solve(A, b))
    print("L,U,P\n", scipy.linalg.lu(A))
    print("Cholesky factor\n", np.linalg.cholesky(A))
    print("Q, R\n", np.linalg.qr(A))
    print("U, S, Vh\n", np.linalg.svd(A))
    print("D, V\n", np.linalg.eig(A))
    print("D, V\n", scipy.sparse.linalg.eigs(A, k=1))
```