Introduction to python

Iván Andrés Trujillo Abella

What is involved in programming

- Creativity
- Abstraction
- Divisibility

key words

To name variables avoid uses words as:

```
def
class
elif
list
while
...
```

Arithmetic operators

Take in mind that a binary or unary operator has a level of hierarchy.

```
2**4 # Exponentiation
4 //2 # integer division
3%2 # Module
```

```
2**4+1
(2**4)+1
```

The exponentiation has the highest hierarchy, then will be executed first.

Operator precedence

From left to right major hierarchy. (),**, *, / , +-.

```python
a = 2
b = 4
c = 9
d =  5
d+a**c/b-d
```

Which is the value?, we can calculate it step by step:

```python
r = 2**2
r = r/4
r = r + 5
r = r -9
print(r)
```

Data types

Boolean

Booleans values have two types: True or False.

```
p = 10
q = 3
p%q  == 0
```

Boolean rules

Boolean rules of the conjuction and disjunction and negative (logical operators). Internally True is (1) and False (0).

```
and
or
not
```

- x and True == x
- x or False == x
- x or True == True
- x and False == False
- x and x == x
- x or x == x

Conditionals

Assess conditions return boolean values, then the program will execute a block if the value is True; otherwise, it will not be exectuted.

```
if condition:
    #Block
    statements
```

```
x = 3
if x > 2 :
    print('x-is a number grater than 2')
```

Conditionals

When appear *else* will be executed the else block if the condition is False.

```
if condition:
    # if block
    statements
else:
    # else-block
    statements
```

Conditionals

elif

In some cases we need assess more than one condition, for instance, check different ranges in a numerical variable.

```
if condition:
    #if - block
elif condition:
    # elif - block:
elif condition:
    # elif - block:
...
else:
    # else - block
```

Elif help to reduce the quantity of assessments. When an elif expression is True then the program skip to get out of else, nevertheless *if* check all conditions. The program without else works.

loops

while

Repeat code until reach a condition, for instance we need print in screen the number from 0 to 10.

```python
i = 0
while i < 11:
    print(i)
    i  = i + 1
```

Note here that the statements **print(i)** and $\mathbf{i = i + 1}$ will be executed until $\mathbf{i < 11}$ be **False**. Remember that python first assess the right hand and after assign to **i**.

loops

for

It is a short hand of while loop.

```python
for i in range(0,11):
    print(i)
```

Note here that we not initialize the variable **i=0** and also do not define a exit statement $\mathbf{i = i + 1}$. The sequence begin in 0 and end in $n - 1$.

for vs while

```
# While implementation
i = 0
while i < 10:
  k = 0
  while k < 10 :
    j = 0
    while j < 10 :
      print(i,k,j)
      j = j + 1
    k = k + 1
  i = i +1
#-------------------#
# for implementation
for x in range(10):
    for y in range(10):
        for z in range(10):
            print(x,y,z)
```

Example flag

```python
def searchk(k,l):
  flag = False
  counter = 0
  for x in l:
    if x == k :
      flag = True
    counter += 1
  return flag,counter
```

Example flag

```
def searchop(k,ls):
  i = 0
  counter = 0
  flag = False
  while i < len(lisn) and flag==False:
    if lisn[i] == k:
      flag = True
    counter +=1
    i +=1
  return flag, counter
```

Basic structures

- list
- dictionary
- tuples

Remember that list not keep the objects itself otherwise, keep the direction of memory (therefore it is neccesary uses copy method). The simple invocation:

```
lista = []
tupla = ()
dictionario = {}
```

Python Structure

LIST

Lists allow us to save elements of several data types: numeric, boolans, integers and so, even save another list.

list

Empty string

list it is a built -in.

```
list = []
```

and empty string, we can focus in several elements.

```
list=[1,2,True,'circle']
```

Adding one element,

```
list.append('String')
```

List

Copy or clone

```
array = [1,2,3,4]
array_ = array
array_c = array.copy()
print(id(array) == id(array_))
print(id(array) == (array_c))
```

try to see the result of

```
array_ci = array[:]
```

Numerical system

Uses as base, the ten(10) number, with the following digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. Not is the only one, you could uses:

- Binary
- Hexadecimal

Numerical system

Positional system

the value of each digit rely on in the place that occupy in the number.

Decimal representation

The value of each number is ten times more greater than the precedence.

$$\overset{\text{One thousands}}{\mathbf{1}}\ \underset{Hundreds}{\mathbf{1}}\ \overset{Tens}{\mathbf{0}}\ \underset{Units}{\mathbf{1}}$$

Fron list to decimal

```
lista = [0,1,2,3]
acum = 0
if lista[0] != 0:
  for i in range(len(lista)):
    acum  = acum + lista[i]*factor/10**i
else:
  for i in range(0,len(lista)):
    acum = acum + lista[i]*10**-i
acum
```

How to write a program?

- Write in a blank paper the general structure
- Probe to hand, step by step the solution
- Generalize the program
- Uses functions
- Parameters outside

Iterable object

An iterable object means that we can 'run' over each element that compound in python this start off in 0 index.

```
lista = ['A','B','C']
for j in lista:
    print(j)
```

a list of lists.

```
ListaS=[[1,2,3],[4,5,6],[7,8,9]]
i = 0
for j in listaS:
  i = i+1
  print(j,'row',i,'\n')
```

Iterable object

Dictionary

When uses a for loop we iterate over the *key_value*.

```
info = {'key_value_one:'1', 'Key_value_two:'2'}

for key in info:
    print(key)
```

the last code will be return:

```
Key_value_one
Key_value_two
```

Iterable object

Dictionary

We can access to the data associated with each *key_value*.

```
for key in info:
    print(info[key])
```

allow us to see the data stored in the array.

```
dco = {'key_one':'triangle,square,circle',
'key_two':['triangle','square','circle']}
```

Note that the dictionary allow us keep differente data structures or data types.

Dictionary

Take in mind that python it is based upon object oriented programming, we need take in mind that daya types is returned or we are using

```
type(object) # return data type
```

and therefore we can add to *key_two* a new element to the value associated in this key.

```
dco['Key_two'].append('rectangle')
```

Note that we uses **append()** that is a method of lists.

Example 1

Adding $i-th$ elements in list

```
matrix = [[0,2,10,30],[3,5,4,10],[0,1,2,3],[10,11,12,14]]
pairs=[]
sum=0
for x in range(len(matrix[0])):
    print(sum)
    pairs.append(sum)
    sum=0
    for j in range(len(matrix)):
      #print(j,x)
      sum = matrix[j][x]+sum
pairs.append(sum)
pairs = pairs[1:]
print(pairs)
```

Example 2

ordering each list inverse

```
new=[]
k= 1
for x in lista:
  row=[]
  j= (len(x) -1)
  while (j)!=-1:
    row.append(x[j])
    j = j -1
  k=k+1
  new.append(row)
print(new)
```

Functions

The **Functions** are very handy to work with repetitive processes.

```
def Function_name(k,f,h ...):
    statements
    return result
```

The example of function it is

```
def root(n,k):
    return n**(1/k)
```

signature

```python
def function(parameter_one: int, parameter_two:float) -> float
```

Docstring

The function documentation is called docstring:

```
def power(x,n):
  """
  This is docstring that will be invoked
  with the function help(power)
  ----------------
  power(x,n)
  x it is a any number that will be powered to n
  ---------------
  """
  return x**n
```

you can peek using the help function

```
help(power)
```

Factorial function

There are different ways of compute the same thing, the factorial it is a practical function in combinatorial analysis and statistics.

$$\begin{aligned} n! =& n(n-1)(n-2)(n-3)...1 \\ n! =& n(n-1)! \end{aligned} \tag{1}$$

For loop

Factorial function

the built in function type().

```
def factl(n):
    prod = 1
      for x in range(1,n):
      prod *= x
    return prod
```

While loop

Factorial function

```python
def fac2(n):
    prod = 1
    while n > 1:
        prod *= n
        n -= 1
    return prod
```

Count

```
def count_vowels(iterable):
  vowel=0
  for i in iterable:
    if i in ['a','e','i','o','u']:
      vowel += 1
  return vowel
print(count_vowels('abcdd'))
```

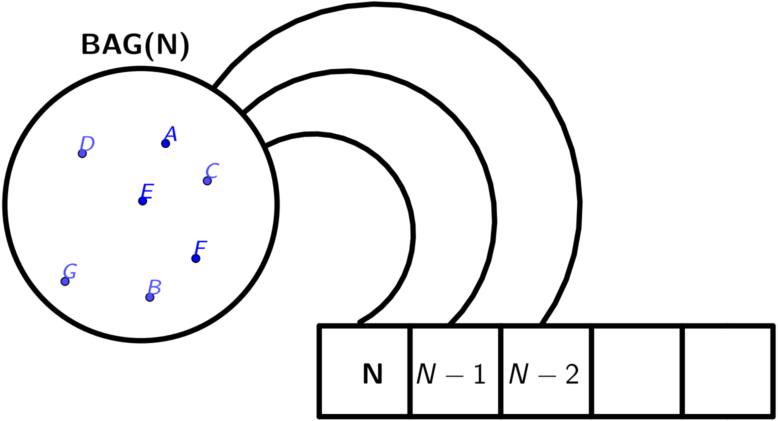
Recursion

It is a method to tackle problems that contain instances of a smaller problem of itself.

```
def fact(n):
    if n==1:
        return 1
    else:
        return n*fact(n-1)
```

```
def mlR(a,b):
  if b==1:
    return a
  else:
    return a + mlR(a,b-1)
```

Bag model ilustration



Permutation - Combination

In permutations there is not replacement and order matter.

$$P_k^n = N(N-1)(N-2)(N-3)...(N-k+1) \tag{2}$$

However for each arragement of k longitude there are $k!$ of each arragement, therefore the number of possible ways of different total elements it is a combination.

$$C_k^n = \frac{P_k^n}{k!} = \frac{n!}{(n-k!)k!} \tag{3}$$

Birthday paradox

Programming and mathematics are practical. Statistics it is a field in which we can interact with real data, the **birthday paradox** it is a practical example of this.
If we have k persons then the probability that two of them born in the same day is; if the year have 360 days:

$$1 - \frac{360.359.358...(360-k+1)}{360^k}$$

In the case of :

$$1 - \left(\prod_{i=311}^{360} i(360^{-50}) \right) \approx 0,97 \tag{4}$$

is 97% is a high probability that at least two person born the same day.

Birthday paradox

Simulation

```python
import random
def match(k,num_matchs,days):
  acum_array =[0]*days
  for _ in range(k):
    day = random.choice(range(days))
    acum_array[day] +=1
  return max(acum_array)>= num_matchs
def game(k,days, trials):
  counter = 0
  for _ in range(trials):
    counter = counter + int(match(k,2,days))
  return counter/trials
```

The following implementation is based on the uses of factorials, that are defined in *fact*() in a recursive way.

```
def permutation(n,k):
  return fact(n) / fact(n-k)

def paradox(n,k):
  return 1-(permutation(n,k)/(n**k))
```

list comprehension

Composed of expression, if - clauses, and loops.

```
[exp for i in iterable]
odds = [["par",i] if i%2==0 else ["impar",i] for i in
    range(1,11)]
print(odds)
```

Dynamic code

```
for x in range(3):
    exec(f'var_{x}=x')
```

Break statement

Sometimes we need stop the run or execution of a program, the **break**

```
k=222
limit = 19
ls=[]
for j in range(k):
    if j%2==0 :
        ls.append(j)
    if len(ls)>limit:
        break
print(ls)
print(len(ls))
```

The previous code will finish when the length of the list, reach a size greater than the limit.

Continue statement

In some cases we need avoid a condition, remaining in the cycle.

```
N=10
ls=[]
for l in range(N):
    if l==4 or l==6:
        continue
    elif l%2==0:
        ls.append(l)
print(ls)
```

Sorting

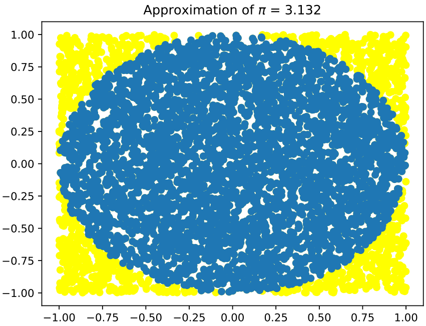
```
dat=[99,23,13,44,120,42,12]
new=[]
while len(dat)!=0 :
    k=dat[0]
    for j in dat:
        if k < j:
            k=j
    dat.remove(k)
    new.append(k)
print(new)
```

π Number

```python
import numpy as np
import matplotlib.pyplot as plt
dots = 5000
c1,c2=-1,1
x = np.random.uniform(c1,c2,
    size=dots)
y = np.random.uniform(c1,c2,
    size=dots)
coordenates_circle =
    (x**2)+(y**2) < 1
circle_y=y[coordenates_circle]
circle_x=x[coordenates_circle]
pi = 4*sum(coordenates_circle)
    / dots
plt.scatter(x,y, color='yellow')
plt.scatter(circle_x,circle_y)
```

What is the probability of a drop lands in the circle?

$$P(hit) = \frac{\pi r^2}{4r^2} \tag{5}$$



Matrix

```
matrix = [[1,2,3],[10,11,12],[18,19,29]]
```

Matrix

Sum

```
def sum_matrix(matA, matB):
  rows = len(matA)
  column =len(matA[0])
  if len(matA) != len(matB):
    raise Exception('Dont have the same number of rows')
  sum_mat = []
  for i in range(0,rows):
    row =[]
    for j in range(0,column):
      row.append(matA[i][j]+matB[i][j])
    sum_mat.append(row)
  return sum_mat
```

Matrix tranpose

```python
def transpose(matx):
  transpose_mat=[]
  for h in range(0,len(matx[0])):
    row =[]
    for j in matx:
      row.append(j[h])
    transpose_mat.append(row)
  return transpose_mat
```

Identity

```
def nrow(size,row_n):
    row = []
    for j in range(0,size):
      if j == row_n-1:
        row.append(1)
      else:
        row.append(0)
    return row
```

Identity

```
def indentity(matrix):  # we also could take min ( column, row)
    and return a square of these size
  # check if square
  check = len(matrix)
  for j in matrix:
    if len(j)!= check:
      raise Exception("Not is a square MATRIX")
  mat=[]
  for j in range(1,check+1):
    mat.append(nrow(check,j))
  return mat
```

Inner product

```
def inner(vectorA,vectorB):
  if len(vectorA) != len(vectorB):
    raise Exception('overcome dimensionality')
  column = len(vectorA)
  addedVector = []
  for j in range(0,column):
    if type(vectorB[j])==list:
      addedVector.append(vectorA[j] * vectorB[j][0])
    if type(vectorB[j])!=list:
      addedVector.append(vectorA[j] * vectorB[j])
  return sum(addedVector)
```

Product

```
def prod(matA, matB):
 row = len(matA)
 col = len(transpose(matB))
 resultado = []
 for j in range(0,row):
   row =[]
   for i in range(0,col):
    row.append(inner(matA[j],transpose(matB)[i]))
   resultado.append(row)
 return resultado
```

Numpy

What is numpy and why it is important? numpy is a open source project, that allow us work with arrays.
np.arrays are most fast than built-in lists.
some functions of numpy are written in C or C++.

Inner(dot) product

$$
\begin{aligned}
\vec{u}.\vec{v} &= \sum u_i v_i \\
\vec{u}.\vec{u} &= \frac{n(n+1)(2n+1)}{6}
\end{aligned} \tag{6}
$$

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

def sum_square(number):
  return number * (number+1) * (2*number +1 ) / 6
sum_square(6)
```

Matrix multiplication

@

When the arrays are of 1-D then uses*np.dot*() otherwise, 2-D arrays uses *np.matmult*() or @.

$$AB = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 5 & 2 \\ 4 & 2 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 3 & 7 \\ 4 & 2 \end{pmatrix}$$

```
A = np.array([[1,2,3], [1,5,2], [4,2,0]])
B = np.array([[1,0],[0,1],[1,0]])
A @ B
np.matmul(A,B)
```

Numpy

other functions

```
np.zeros((rows,columns)) #Return a array of zeros.
np.transpose(array) # Return the transpose matrix.
np.ones_like(array) # Return the array with the same size but
    filled with ones.
```

Switch

```
def funcion1():
  print('f1')
def funcion2():
  print('f2')
def funcion3():
  print('f3')
def switch(a):
  swithcer={1: funcion1,
            2: funcion2,
            3: funcion3
  }
  return swithcer.get(a)()
switch(1)
```

Type hints

We said previously that is important in object oriented programming, to know, what type of object will be returned by a function, this allow us read more easily the code.
To write better code

```
def factorial(n:int) -> int:
statements..
```

This is a way of write better code, but take time.

Type hints

```
def factorial(n:int) -> int:
acum: int = n
for x in range(1,n):
    acum: int = acum + x
return acum
```

This is a way of write better code, but take time.

To see documentation about type hints.

```
factorial.__annotations__
```

types

```
int, float,str, NoneType
```

Unicode

is a standard to codify characters, in python the function ord() return its code. ASCII values..

Prime number

```
def prime(k):
  i =1
  flag = False
  while flag==False and i<k-1:
    i = i+1
    if k%i==0:
      flag=True
  if flag==True:
    return 'no primo'
  else:
    return 'primo'
prime(7)
```

Split own implementation

```
def split(text, char):
  joint, result = '', []
  for letter in text:
    if letter == char:
      result.append(joint)
      joint = ''
    else:
      joint = joint + letter
  return result
```

Ceaser code criptography

```
def cripto(text,constant):
  result = ''
  for letter in text:
    letter_coded = str(ord(letter)+ constant) + '-'
    result = result + letter_coded
  result = result[0:-1]
  return result

def decifre(text,constant):
  result = ''
  for code in text:
    decode = chr(int(code)-constant)
    result = result + decode
  return result
```

Dynamic vs static typing

we said that the language follow a static programming style if the variables must be defined in compilation time. Otherwise, dynamic programming define variables in execution time.

```
int c = 10
```

strong and weak typing: python have a strong typing for instance

```
a = 10
b = '1'
print(a+b)
```

this will araise a error, in a weak language one variable cast to compatible type for instance to concatenate, '101'.
Python allow us a static system with type hints, for instance;

```
var: float = 10.1
```

mypy

spite of the ability of python to be explicit definition of type, not guaranteed that the parser arise a error if the variable change of type in execution time. we can uses mypy to check the consistency.
if we want a variable that change in the execution then we could define as:

```python
from typing import Any
var: Any = 10
```

Then mypy not raises a error.

Delete functions

```
del function_Name
```

Modules and packages

import it is a keyword. the package will be loaded, if there is in path, python search the module in the current directory, and after in PYTHONPATH.

```
sys.path # Directories to load modules
```

Namespace

Could exist two or more variables with the same name, living in different namespace. The scope of a variable play a key role: **global** variable could be invoked inside or outside of a function, otherwise a **local** variable only could be invoked inside the function(this imply that the variable was defined in the same function).

local and global scope

global refer to the ability to access in all program, and **local** only for parts of code. When you define a variable, or assign a value inside a function, its scope is local. If two variables have the same name, local variable override the global.

Namespace and scope

variables are identifiers mapping to the objects stored in memory, a dictionary of variables names is called **Namespace**. Each function has its corresponding namespace.

Namespace and scope

In each invocation of a function, it was created a scope, and this is destroyed when appear return.

```python
def function(x):
    result = ...
def function_(x):
    result = ...
```

Notice that **result** do not clash or rise a error, due both variables are defined in local scope. this means that is not allowed be invoked from another side of its own scope.

LEGB rule

Local,Enclosing,Global, Built-in

paradigm of Scope this mechanism avoid collision by names, python names could came from: Assignments, import , def and Class.

- Local scope: python functions or lambda expressions
- Enclosing or nolocal: Nested functions
- Global: related to the module, visible in all program.
- Built-in: have keywords, functions, exceptions, that are built in.

This rule is a hierarchical structure for searching or call a variable.

Symbol table

Is a data structure that contain information about; Methods,classes, variables, and so on of a program. there are global and local tables:

```
globals()
locals()
```

When you import a module this not is loaded automatically in symbol table only the module name, therefore is needed access by the module name; for instance **np.float()** It is important to know that each module have its own symbol table.

dir

the **dir()** function tell us that things are inside of a package.

```
dir(module_name)
```

could be useful to remember methods and attribute inside a package.

__main__

Sometimes you write a module to be imported and its behavior must be change regarding if itself is the main program o will be part of another program.

```
if __name__=='main':
    print('the main program never will be imported')

else:
    print('was loaded and not be a main program')
```

save the last snippet as **two.py**

```
import two
print('this is main and was be executed form shell')
```

save as **one.py** and execute in shell

`python3`