Breve introdução ao python básica Foco apenas nos tipos de dados, built-ins, funções, estruturas de controle

Curso será feito usado Jupyter Notebook Na próxima aula falarei sobre as bibliotecas que utilizaremos

Python:

- Estrutura básica
- Estruturas de controle
- Loops
- Importanto módulos
- Tipos de dados

Antes, mostrar Jupyter

Estrutura básica

- Linguagem de tipagem dinâmica
  - Tipo de dado definido na atribuição
  - Interpretador que infere o tipo do dado
- Tipagem forte
  - interpretador avalia as expressões e não faz coerções automáticas entre tipos não compatíveis
- Blocos de código definidos pelo aninhamento
- Built-in help()
  - operadores in, is ...
- Definição de função (lambda)

In [2]:

```
i, j, k = 10, "Rafael", "Pedro"
print(i)
print(j)
print("O primeiro resultado é: ", j + k)
print("O segundo resultado é: ", i + j)
```

```
10
Rafael
O primeiro resultado é:  RafaelPedro

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-af20671790f7> in <module>()
      3 print(j)
      4 print("O primeiro resultado é: ", j + k)
----> 5 print("O segundo resultado é: ", i + j)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [3]:

```
print("O segundo resultado é: ", str(i) + j)
```

```
O segundo resultado é:  10Rafael
```

In [4]:

```
condicao = True
if condicao:
    print('Bloco do if definido pelo aninhamento')
print('Esse print está fora do if')
```

```
Bloco do if definido pelo aninhamento
Esse print está fora do if
```

In [5]:

```
def funcao():
    print('Dentro da função')
print('Fora da função')
funcao()
```

```
Fora da função
Dentro da função
```

In [6]:

```
def funcao():
    print('Dentro da função 1')
   print('Dentro da função 2')
print('Fora da função')
funcao()
```

```
  File "<ipython-input-6-40655b4eaecb>", line 3
    print('Dentro da função 2')
                               ^
IndentationError: unindent does not match any outer indentation level
```

In [7]:

```
def funcao():
    print('Dentro da função 1')
    print('Dentro da função 2')
print('Fora da função 1')
    print('Fora da função 2')
funcao()
```

```
  File "<ipython-input-7-26ee16a04696>", line 5
    print('Fora da função 2')
    ^
IndentationError: unexpected indent
```

```
In [8]: help({})
```

```
Help on dict object:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Methods defined here:
 |
 |  __contains__(self, key, /)
 |      True if D has a key k, else False.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signatur
e.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(...)
 |      D.__sizeof__() -> size of D in memory, in bytes
 |
 |  clear(...)
 |      D.clear() -> None.  Remove all items from D.
```

In [9]:

```
a = 10
help(a)
```

```
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |  
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |  
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |  
 |  Methods defined here:
 |  
 |  __abs__(self, /)
 |      abs(self)
 |  
 |  __add__(self, value, /)
 |      Return self+value.
 |  
 |  __and__(self, value, /)
 |      Return self&value.
 |  
 |  __bool__(self, /)
 |      self != 0
 |  
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |  
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |  
 |  __eq__(self, value, /)
 |      Return self==value.
 |  
 |  __float__(self, /)
 |      float(self)
 |  
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |  
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |  
 |  __format__(...)
 |      default object formatter
 |  
 |  __ge__(self, value, /)
 |      Return self>=value.
 |  
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |  
 |  __getnewargs__(...)
 |  
 |  __gt__(self, value, /)
 |      Return self>value.
 |  
 |  __hash__(self, /)
 |      Return hash(self).
 |  
 |  __index__(self, /)
```

Definição de função

```
In [10]: def nome_funcao():
             pass
```

```
In [11]: def f1(p1, p2, p3=10):
             print(p1, p2, p3)
```

```
In [12]: f1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-b27bf7c7aafe> in <module>()
----> 1 f1()

TypeError: f1() missing 2 required positional arguments: 'p1' and 'p2'
```

```
In [13]: f1(1,2)
```

```
1 2 10
```

```
In [14]: f1(p2=100, p1=30)
```

```
30 100 10
```

```
In [15]: f1(p2=100, p1=30, p3=1)
```

```
30 100 1
```

```
In [16]: x = 3
         def func(x):
             x = 7 # defining a local x, not changing the global one
             print(x)
         func(x)
         print(x)
```

```
7
3
```

```
In [17]: x = 1
         del x
         def func(x):
             x = 7 # defining a local x
             print(x)
         func(x=3)
         print(x)
```

```
7

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-17-68673fd165ef> in <module>()
      5     print(x)
      6 func(x=3)
----> 7 print(x)

NameError: name 'x' is not defined
```

In [18]:

```
x = [1, 2, 3]
def func(x):
    x[1] = 42
func(x) # this affects the caller!
print(x)
```

```
[1, 42, 3]
```

"Assigning an object to an argument name within a function doesn't affect the caller"

In [19]:

```
x = [1, 2, 3]
def func(x):
    x[1] = 42 # this changes the caller!
    x = 'something else' # this points x to a new string object
func(x)
print(x)
```

```
[1, 42, 3]
```

Avoid the trap! Mutable defaults One thing to be very aware of with Python is that default values are created at def time, therefore, subsequent calls to the same function will possibly behave differently according to the mutability of their default values. Let's look at an example:

In [20]:

```
def func(a=[], b={}):
    print(a)
    print(b)
    print('#' * 12)
    a.append(len(a)) # this will affect a's default value
    b[len(a)] = len(a) # and this will affect b's one
func()
func()
func()
```

```
[]
{}
############
[0]
{1: 1}
############
[0, 1]
{1: 1, 2: 2}
############
```

Procurar por memoization techniques

Retornando valores de funções

Você pode retornar qualquer coisa de uma função. Um objeto, dois objetos, uma lista, uma tupla .... ou nenhum

In [21]:

```
def func():
    pass
func() # the return of this call won't be collected. It's lost.
```

In [22]:

```
a = func() # the return of this one instead is collected into `a`
print(a)
```

```
None
```

```
In [23]: def moddiv(a, b):
             return a // b, a % b

         print(moddiv(20, 7))
```

```
(2, 6)
```

Funções também são objetos, logo também têm atributos

```
In [24]: def multiplication(a, b=1):
             """Return a multiplied by b. """
             return a * b

         special_attributes = [
             "__doc__", "__name__", "__qualname__", "__module__",
             "__defaults__", "__code__", "__globals__", "__dict__",
             "__closure__", "__annotations__", "__kwdefaults__",
         ]

         for attribute in special_attributes[:6]:
             print(attribute, '->', getattr(multiplication, attribute))
```

```
__doc__ -> Return a multiplied by b.
__name__ -> multiplication
__qualname__ -> multiplication
__module__ -> __main__
__defaults__ -> (1,)
__code__ -> <code object multiplication at 0x7fbec771a780, file "<ipython-inp
ut-24-9e085e33ac57>", line 1>
```

Função anônima, lambda

```
In [25]: l = lambda x: x**3
```

```
In [26]: l(4)
```

```
Out[26]: 64
```

```
In [27]: print((lambda x,y: 2*x+ y+5)(10, 20))
```

```
45
```

Funções built-in

any , bin , bool , divmod , filter , float , getattr , id , int , len , list , min , print , set , tuple , type , e zip

sads

Estrutura de controle

- if, elif, else
- operador ternário

In [230]:

```
late = True
if late:
    print('Estou atrasado!')
```

```
Estou atrasado!
```

In [29]:

```
late = False
if late:
    print('Estou atrasado!')
else:
    print('Ainda tenho tempo')
```

```
Ainda tenho tempo
```

In [30]:

```
income = 15000
if income < 10000:
    tax_coefficient = 0.0
elif income < 30000:
    tax_coefficient = 0.2
elif income < 100000:
    tax_coefficient = 0.35
else:
    tax_coefficient = 0.45
print('I will pay:', income * tax_coefficient, 'in taxes')
```

```
I will pay: 3000.0 in taxes
```

Operador ternário

In [31]:

```
order_total = 247 # GBP
# classic if/else form
if order_total > 100:
    discount = 25 # GBP
else:
    discount = 0 # GBP
print(order_total, discount)
```

```
247 25
```

In [32]:

```
discount = 25 if order_total > 100 else 0
print(order_total, discount)
```

```
247 25
```

asda

Loop

- for, while
- break, continue

For

```
In [33]: for number in [1,2,3,4]:
             print(number)

         1
         2
         3
         4
```

```
In [34]: for number in range(4):
             print(number)

         0
         1
         2
         3
```

```
In [35]: list(range(10)) # one value: from 0 to value (excluded)

Out[35]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [36]: list(range(3, 8)) # two values: from start to stop (excluded)

Out[36]: [3, 4, 5, 6, 7]
```

```
In [37]: list(range(-10, 10, 4)) # three values: step is added

Out[37]: [-10, -6, -2, 2, 6]
```

```
In [38]: range(5)

Out[38]: range(0, 5)
```

```
In [39]: print(range(5))

         range(0, 5)
```

```
In [40]: surnames = ['Rivest', 'Shamir', 'Adleman']
         for position in range(len(surnames)):
             print(position, surnames[position])

         0 Rivest
         1 Shamir
         2 Adleman
```

```
In [41]: surnames = ['Rivest', 'Shamir', 'Adleman']
         for position, surname in enumerate(surnames):
             print(position, surname)

         0 Rivest
         1 Shamir
         2 Adleman
```

```
In [42]: x = 3
         for i in range(5):
             x=i
         print(x)

         4
```

While

```
In [43]: n = 5
         while n > 0:
             print(n)
             n -= 1
```

```
5
4
3
2
1
```

Continue e break

```
In [44]: for i in range(5):
             if i == 3:
                 continue
             print(i)
```

```
0
1
2
4
```

```
In [45]: for i in range(5):
             if i == 3:
                 break
             print(i)
```

```
0
1
2
```

Importanto módulos

There are many different ways to import objects into a namespace, but the most common ones are just two:

import module_name

and

from module_name import function_name

```
In [46]: factorial(5)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-46-637175d621a4> in <module>()
----> 1 factorial(5)

NameError: name 'factorial' is not defined
```

```
In [47]: from math import factorial
```

```
In [48]: factorial(5)
```

```
Out[48]: 120
```

```
In [49]: factorial(120)

Out[49]: 66895029134491270575881180540903725867527463331380298102956713523016335572449
         62989366874165271984981308157637893214090552534408589408121859898481114389650
         005964960521256960000000000000000000000000000
```

```
In [50]: import numpy as np
         np.sqrt(10)

Out[50]: 3.1622776601683795
```

Relative imports

The imports we've seen until now are called absolute, that is to say they define the whole path of the module that we want to import, or from which we want to import an object. There is another way of importing objects into Python, which is called relative import. It's helpful in situations in which we want to rearrange the structure of large packages without having to edit sub-packages, or when we want to make a module inside a package able to import itself. Relative imports are done by adding as many leading dots in front of the module as the number of folders we need to backtrack, in order to find what we're searching for. Simply put, it is something like this:

from .mymodule import myfunc

Tipos de dados

- Todos os tipos de dados são objetos
  - Todos terão atributos e funções associados
- Mutáveis e imutáveis
- Numérico, int, float, complex (Numbers are immutable objects.)
- String
- list, tuple, set, dict, object, bool

Since everything in Python is an Object, every variable holds an object instance. When an object is initiated, it is assigned a unique object id. Its type is defined at runtime and once set can never change, however its state can be changed if it is mutable. Simple put, a mutable object can be changed after it is created, and an immutable object can't.

```
In [51]: a = [1,2,4,5,2]
         # so atribuir não mostra nada na saída
```

```
In [52]: # mas se criar diretamente ou chamar o objeto será mostrado sua representaçã
         o em string
         # chamada das funções __str__() ou __repr__()
         a

Out[52]: [1, 2, 4, 5, 2]
```

```
In [53]: [46,6,3,3,5]

Out[53]: [46, 6, 3, 3, 5]
```

```
In [54]: # só funciona para o último objeto
         [4,6,34,4,6]
         a

Out[54]: [1, 2, 4, 5, 2]
```

```
In [55]: a.__str__()

Out[55]: '[1, 2, 4, 5, 2]'

In [56]: a.__repr__()

Out[56]: '[1, 2, 4, 5, 2]'
```

Pode ser útil sobreescrever ou criar o seu método __str**() ou** \repr__()

Números:

Python integers have unlimited range, subject only to the available virtual memory. This means that it doesn't really matter how big a number you want to store: as long as it can fit in your computer's memory, Python will take care of it. Integer numbers can be positive, negative, and 0 (zero). They support all the basic mathematical operations, as shown in the following example:

```
In [57]: a = 12
         b = 3
         a + b
         # addition

Out[57]: 15

In [58]: b - a # subtraction

Out[58]: -9

In [59]: a // b # integer division

Out[59]: 4

In [60]: a / b # true division

Out[60]: 4.0

In [61]: a * b # multiplication

Out[61]: 36

In [62]: b ** a # power operator

Out[62]: 531441

In [63]: 2 ** 1024 # a very big number, Python handles it gracefully

Out[63]: 17976931348623159077293051907890247336179769789423065727343008115773267580550
         09631327084773224075360211201138798713933576587897688144166224928474306394741
         24377767893424865485276302219601246094119453082952085005768838150682342462881
         47391311054082723716335051068458629823994724593847971630483535632962422413721
         6
```

The preceding code should be easy to understand. Just notice one important thing: Python has two division operators, one performs the so-called true division (/), which returns the quotient of the operands, and the other one, the so-called integer division (//), which returns the floored quotient of the operands. See how that is different for positive and negative numbers:

```
In [64]: 7 / 4
         # true division
Out[64]: 1.75
```

```
In [65]: 7 // 4
         # integer division, flooring returns 1
Out[65]: 1
```

```
In [66]: -7 / 4
         # true division again, result is opposite of previous
Out[66]: -1.75
```

```
In [67]: -7 // 4
         # integer div., result not the opposite of previous
Out[67]: -2
```

```
In [68]: 10 % 3
         # remainder of the division 10 // 3
Out[68]: 1
```

```
In [69]: 10 % 4
         # remainder of the division 10 // 4
Out[69]: 2
```

This is an interesting example. If you were expecting a -1 on the last line, don't feel bad, it's just the way Python works. The result of an integer division in Python is always rounded towards minus infinity. If instead of flooring you want to truncate a number to an integer, you can use the built-in int function, like shown in the following example:

```
In [70]: int(1.75)
Out[70]: 1
```

```
In [71]: int(-1.75)
Out[71]: -1
```

Reals Real numbers, or floating point numbers, are represented in Python according to the IEEE 754 double-precision binary floating-point format, which is stored in 64 bits of information divided into three sections: sign, exponent, and mantissa

```
In [72]: import sys
         sys.float_info
Out[72]: sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min
         =2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53
         , epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

```
In [73]: 2**1024
Out[73]: 179769313486231590772930519078902473361797697894230657273430081157732675805500
         963132708477322407536021120113879871393357658789768814416622492847430639474124
         377767893424865485276302219601246094119453082952085005768838150682342462881473
         913110540827237163350510684586298239947245938479716304835356329624224137216
```

```
In [74]: 10**308
```

```
Out[74]: 10000000000000000000000000000000000000000000000000000000000000000000000000000
         00000000000000000000000000000000000000000000000000000000000000000000000000000
         00000000000000000000000000000000000000000000000000000000000000000000000000000
         00000000000000000000000000000000000000000000000000000000000000000000000000000
         0
```

Funções round, abs

```
In [75]: print(round(10.12345, 0))
         print(round(10.12345, 1))
         print(round(10.12345, 3))
```

```
10.0
10.1
10.123
```

```
In [76]: print(abs(-10))
         print(abs(-10 + 20j))
```

```
10
22.360679774997898
```

Também tem o tipo para número complexo

```
In [77]: a = 10 + 20j
         b = 20 + 30j
         a
```

```
Out[77]: (10+20j)
```

```
In [78]: a + b
```

```
Out[78]: (30+50j)
```

```
In [79]: a.real
```

```
Out[79]: 10.0
```

```
In [80]: a.imag
```

```
Out[80]: 20.0
```

```
In [81]: a.conjugate()
```

```
Out[81]: (10-20j)
```

```
In [82]: a ** 2
```

```
Out[82]: (-300+400j)
```

Boleanos

- True e False
- Subclasse dos inteiros e se comportam como 0 e 1

```
In [83]: True
Out[83]: True

In [84]: False
Out[84]: False

In [85]: int(True)
Out[85]: 1

In [86]: int(False)
Out[86]: 0

In [87]: bool(1)
Out[87]: True

In [88]: bool(0)
Out[88]: False

In [89]: bool(-34)
Out[89]: True

In [90]: not True
Out[90]: False

In [91]: not False
Out[91]: True

In [92]: True and False
Out[92]: False

In [93]: True or False
Out[93]: True
```

Sequencias imutáveis: String, tuples

Strings

```
In [94]: print('String com aspas simples')
         print("String com aspas duplas")
         print("""String
         com
         quebra de linha""")
```

```
String com aspas simples
String com aspas duplas
String
com
quebra de linha
```

```
In [95]: # imutável
         a = 'String'
         print(len(a))
         print(a)
         print(a[3])
         a[3] = 'O'
```

```
6
String
i

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-95-809ac0a9504b> in <module>()
      4 print(a)
      5 print(a[3])
----> 6 a[3] = 'O'

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

Indexing and slicing strings

- Mesmo mecanismo para lista

```
In [96]: s = "The trouble is you think you have time."
```

```
In [97]: s[:4] # slicing, we specify only the stop position

Out[97]: 'The '
```

```
In [98]: s[4:] # slicing, we specify only the start position

Out[98]: 'trouble is you think you have time.'
```

```
In [99]: s[2:14] # slicing, both start and stop positions

Out[99]: 'e trouble is'
```

```
In [100]: s[2:14:3] # slicing, start, stop and step (every 3 chars)

Out[100]: 'erb '
```

```
In [101]: s[:] # quick way of making a copy
          # usa valores default de start, stop e step

Out[101]: 'The trouble is you think you have time.'
```

Funções de string

In [102]:
```
st = 'abcDefg'
print(dir(s))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getne
wargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__
reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__'
, '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'cen
ter', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_
map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join
', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', '
startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

In [103]:
```
print(st.capitalize())
print(st)
```

```
Abcdefg
abcDefg
```

In [104]:
```
print(st.find('f'))
print(st.find('h'))
```

```
5
-1
```

In [105]:
```
print(st.replace('b', 'TT'))
```

```
aTTcDefg
```

In [106]:
```
print(st.split('c'))
```

```
['ab', 'Defg']
```

In [107]:
```
print(st.join(['-', '_', '*']))
```

```
-abcDefg_abcDefg*
```

In [108]:
```
print(st.upper())
```

```
ABCDEFG
```

In [109]:
```
variavel_str = 'VaRiAvEl'
variavel_int = 10
variavel_float = 0.12654
print('%s %d %.3f'%(variavel_str, variavel_int, variavel_float))
```

```
VaRiAvEl 10 0.127
```

Tuples

The last immutable sequence type we're going to see is the tuple. A tuple is a sequence of arbitrary Python objects. In a tuple, items are separated by commas. They are used everywhere in Python, because they allow for patterns that are hard to reproduce in other languages. Sometimes tuples are used implicitly, for example to set up multiple variables on one line, or to allow a function to return multiple different objects (usually a function returns one object only, in many other languages), and even in the Python console, you can use tuples implicitly to print multiple elements with one single instruction. We'll see examples for all these cases:

In [110]:
```
t = ()
# empty tuple
```

```
In [111]: type(t)

Out[111]: tuple
```

```
In [112]: one_element_tuple = (42, ) # you need the comma!
```

```
In [113]: three_elements_tuple = (1, 3, 5)
```

```
In [114]: a, b, c = 1, 2, 3
```

```
In [115]: a, b, c
          # tuple for multiple assignment
          # implicit tuple to print with one instruction

Out[115]: (1, 2, 3)
```

```
In [116]: 3 in three_elements_tuple # membership test

Out[116]: True
```

Notice that the membership operator ***in*** can also be used with lists, strings, dictionaries, and in general with collection and sequence objects.

Because they are immutable, tuples can be used as keys for dictionaries (we'll see this shortly). The dict objects need keys to be immutable because if they could change, then the value they reference wouldn't be found any more (because the path to it depends on the key). If you are into data structures, you know how nice a feature this one is to have. To me, tuples are Python's built-in data that most closely represent a mathematical vector. This doesn't mean that this was the reason for which they were created though. Tuples usually contain an heterogeneous sequence of elements, while on the other hand lists are most of the times homogeneous. Moreover, tuples are normally accessed via unpacking or indexing, while lists are usually iterated over.

Sequências mutáveis

- list
- set
- dict

Listas

Lists are commonly used to store collections of homogeneous objects, but there is nothing preventing you to store heterogeneous collections as well.

```
In [117]: [] # empty list

Out[117]: []
```

```
In [118]: list() # same as []

Out[118]: []
```

```
In [119]: [1, 2, 3] # as with tuples, items are comma separated

Out[119]: [1, 2, 3]
```

```
In [120]: [x + 5 for x in [2, 3, 4]] # list comprehension
Out[120]: [7, 8, 9]

In [121]: list((1, 3, 5, 7, 9))
Out[121]: [1, 3, 5, 7, 9]

In [122]: list('hello')
Out[122]: ['h', 'e', 'l', 'l', 'o']
```

Funções de listas

```
In [123]: a = [1, 2, 1, 3]

In [124]: a.append(13) # we can append anything at the end

In [125]: a
Out[125]: [1, 2, 1, 3, 13]

In [126]: a.count(1) # how many `1` are there in the list?
Out[126]: 2

In [127]: a.extend([5, 7]) # extend the list by another (or sequence)

In [128]: a
Out[128]: [1, 2, 1, 3, 13, 5, 7]

In [129]: a.index(13) # position of `13` in the list (0-based indexing)
Out[129]: 4

In [130]: a.insert(0, 17) # insert `17` at position 0

In [131]: a
Out[131]: [17, 1, 2, 1, 3, 13, 5, 7]

In [132]: a.pop() # pop (remove and return) last element
Out[132]: 7

In [133]: a
Out[133]: [17, 1, 2, 1, 3, 13, 5]

In [134]: a.pop(3) # pop element at position 3
Out[134]: 1

In [135]: a
Out[135]: [17, 1, 2, 3, 13, 5]

In [136]: a.remove(17) # remove `17` from the list
In [137]: a

Out[137]: [1, 2, 3, 13, 5]

In [138]: a.reverse() # reverse the order of the elements in the list

In [139]: a

Out[139]: [5, 13, 3, 2, 1]

In [140]: a.sort() # sort the list

In [141]: a

Out[141]: [1, 2, 3, 5, 13]

In [142]: a.clear() # remove all elements from the list

In [143]: a

Out[143]: []

In [144]: dir(a)[-8:]

Out[144]: ['count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

In [145]:

```
help(a)
```

```
Help on list object:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |  
 |  Methods defined here:
 |  
 |  __add__(self, value, /)
 |      Return self+value.
 |  
 |  __contains__(self, key, /)
 |      Return key in self.
 |  
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |  
 |  __eq__(self, value, /)
 |      Return self==value.
 |  
 |  __ge__(self, value, /)
 |      Return self>=value.
 |  
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |  
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |  
 |  __gt__(self, value, /)
 |      Return self>value.
 |  
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |  
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |  
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  __iter__(self, /)
 |      Implement iter(self).
 |  
 |  __le__(self, value, /)
 |      Return self<=value.
 |  
 |  __len__(self, /)
 |      Return len(self).
 |  
 |  __lt__(self, value, /)
 |      Return self<value.
 |  
 |  __mul__(self, value, /)
 |      Return self*value.n
 |  
 |  __ne__(self, value, /)
 |      Return self!=value.
 |  
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signatur
e.
 |  
 |  __repr__(self, /)
 |      Return repr(self).
 |  
 |  __reversed__(...)
 |      L.__reversed__() -- return a reverse iterator over the list
```

```
In [146]: a = list('hello') # makes a list from a string

In [147]: a

Out[147]: ['h', 'e', 'l', 'l', 'o']

In [148]: a.append(100) # append 100, heterogeneous type

In [149]: a

Out[149]: ['h', 'e', 'l', 'l', 'o', 100]

In [150]: a.extend((1, 2, 3)) # extend using tuple

In [151]: a

Out[151]: ['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3]

In [152]: a.extend('...') # extend using string

In [153]: a

Out[153]: ['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3, '.', '.', '.']
```

Outras funções aplicadas às listas

```
In [154]: a = [1, 3, 5, 7]

In [155]: min(a) # minimum value in the list

Out[155]: 1

In [156]: max(a) # maximum value in the list

Out[156]: 7

In [157]: sum(a) # sum of all values in the list

Out[157]: 16

In [158]: len(a) # number of elements in the list

Out[158]: 4
```

Sobrecarga de operadores + e *

```
In [159]: b = [6, 7, 8]
          a + b # `+` with list means concatenation

Out[159]: [1, 3, 5, 7, 6, 7, 8]

In [160]: a * 2 # `*` has also a special meaning

Out[160]: [1, 3, 5, 7, 1, 3, 5, 7]
```

Exemplos mais complicados de manipulação em listas

```
In [161]: from operator import itemgetter
          a = [(5, 3), (1, 3), (1, 2), (2, -1), (4, 9)]
          sorted(a)

Out[161]: [(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]
```

```
In [162]: sorted(a, key=itemgetter(0))

Out[162]: [(1, 3), (1, 2), (2, -1), (4, 9), (5, 3)]
```

```
In [163]: sorted(a, key=itemgetter(1))

Out[163]: [(2, -1), (1, 2), (5, 3), (1, 3), (4, 9)]
```

```
In [164]: sorted(a, key=itemgetter(0, 1))

Out[164]: [(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]
```

```
In [165]: sorted(a, key=itemgetter(1), reverse=True)

Out[165]: [(4, 9), (5, 3), (1, 3), (1, 2), (2, -1)]
```

O método a.sort faz a mesma coisa mas a ordemanção é feita inplace

Ao invés de usar o método itemgetter, poderiamos utilizar uma função lambda Pode ser utilizada para fazer algum cálculo complexo

```
In [166]: sorted(a, key=lambda x:x[0]) # ordena pelo primeiro elemento de x, e x será
          substituído por cada elemento da lista

Out[166]: [(1, 3), (1, 2), (2, -1), (4, 9), (5, 3)]
```

```
In [167]: sorted(a, key=lambda x:x[1])

Out[167]: [(2, -1), (1, 2), (5, 3), (1, 3), (4, 9)]
```

Conjuntos (set)

dois tipos de objetos, *set* (mutável) e *frozenset* (imutável)

Hashability is a characteristic that allows an object to be used as a set member as well as a key for a dictionary, as we'll see very soon.

```
In [168]: small_primes = set() # empty set
          small_primes.add(2) # adding one element at a time
          small_primes.add(3)
          small_primes.add(5)
          small_primes

Out[168]: {2, 3, 5}
```

```
In [169]: small_primes.add(1) # Look what I've done, 1 is not a prime!
```

```
In [170]: small_primes

Out[170]: {1, 2, 3, 5}
```

```
In [171]: small_primes.remove(1) # so let's remove it
In [172]: 3 in small_primes

Out[172]: True

In [173]: 4 in small_primes

Out[173]: False

In [174]: 4 not in small_primes # negated membership test

Out[174]: True

In [175]: small_primes.add(3) # trying to add 3 again
          small_primes

Out[175]: {2, 3, 5}

In [176]: bigger_primes = set([5, 7, 11, 13]) # faster creation

In [177]: small_primes | bigger_primes # union operator `|`

Out[177]: {2, 3, 5, 7, 11, 13}

In [178]: small_primes & bigger_primes # intersection operator `&`

Out[178]: {5}

In [179]: small_primes - bigger_primes # difference operator `-`

Out[179]: {2, 3}

In [180]: {1,2,3}

Out[180]: {1, 2, 3}
```

you can create a set from a list or tuple (or any iterable) and then you can add and remove members from the set as you please.

```
In [181]: small_primes = frozenset([2, 3, 5, 7])

In [182]: bigger_primes = frozenset([5, 7, 11])

In [183]: small_primes.add(11) # we cannot add to a frozenset

          ---------------------------------------------------------------------------
          AttributeError                            Traceback (most recent call last)
          <ipython-input-183-97c9bb65a91e> in <module>()
          ----> 1 small_primes.add(11) # we cannot add to a frozenset

          AttributeError: 'frozenset' object has no attribute 'add'

In [184]: small_primes.remove(2) # neither we can remove

          ---------------------------------------------------------------------------
          AttributeError                            Traceback (most recent call last)
          <ipython-input-184-e8c2b586ad6c> in <module>()
          ----> 1 small_primes.remove(2) # neither we can remove

          AttributeError: 'frozenset' object has no attribute 'remove'
```

```
In [185]: small_primes & bigger_primes # intersect, union, etc. allowed

Out[185]: frozenset({5, 7})
```

Dicionários (dict)

```
In [186]: a = dict(A=1, Z=-1)
          b = {'A': 1, 'Z': -1}
          c = dict(zip(['A', 'Z'], [1, -1]))
          d = dict([('A', 1), ('Z', -1)])
          e = dict({'Z': -1, 'A': 1})
          a == b == c == d == e # are they all the same?

Out[186]: True

In [187]: a is b # podemos utilizar o operador is para comparar objetos também

Out[187]: False
```

Mas esse operador checa se os objetos tem o mesmo id, não o mesmo valor. Por isso é melhor comparar usando ==

```
In [188]: 10 is 10

Out[188]: True

In [189]: a = 10
          b = 10
          a is b

Out[189]: True

In [190]: a = 300
          b = 300
          a is b

Out[190]: False

In [191]: list(zip(['h', 'e', 'l', 'l', 'o'], [1, 2, 3, 4, 5]))

Out[191]: [('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]
```

Mais funções de diconário

```
In [192]: d = {}
          d['a'] = 1 # let's set a couple of (key, value) pairs
          d['b'] = 2
          len(d) # how many pairs?

Out[192]: 2

In [193]: d['a'] # what is the value of 'a'?

Out[193]: 1

In [194]: d # how does `d` look now?

Out[194]: {'a': 1, 'b': 2}
```

```
In [195]: del d['a'] # let's remove `a`
```

```
In [196]: d

Out[196]: {'b': 2}
```

```
In [197]: d['c'] = 3
```

```
In [198]: 'c' in d # let's add 'c': 3
          # membership is checked against the keys

Out[198]: True
```

```
In [199]: 3 in d # not the values

Out[199]: False
```

```
In [200]: 'e' in d

Out[200]: False
```

```
In [201]: d.clear()
```

```
In [202]: d

Out[202]: {}
```

Funções .keys(), .values() .items()

```
In [203]: d = dict(zip('hello', range(5)))
          d

Out[203]: {'h': 0, 'e': 1, 'l': 3, 'o': 4}
```

```
In [204]: d.keys()

Out[204]: dict_keys(['h', 'e', 'l', 'o'])
```

```
In [205]: d.values()

Out[205]: dict_values([0, 1, 3, 4])
```

```
In [206]: d.items()

Out[206]: dict_items([('h', 0), ('e', 1), ('l', 3), ('o', 4)])
```

```
In [207]: d.items()[0]

          ---------------------------------------------------------------------------
          TypeError                                 Traceback (most recent call last)
          <ipython-input-207-62707639c934> in <module>()
          ----> 1 d.items()[0]

          TypeError: 'dict_items' object does not support indexing
```

```
In [208]: list(d.items())

Out[208]: [('h', 0), ('e', 1), ('l', 3), ('o', 4)]
```

```
In [209]: 3 in d.values()

Out[209]: True
```

```
In [210]: ('o', 4) in d.items()

Out[210]: True
```

A ordem dos objetos do dicionário não é garantida.

Mais funções de dicionários

```
In [211]: d

Out[211]: {'h': 0, 'e': 1, 'l': 3, 'o': 4}
```

```
In [212]: d.popitem() # removes a random item

Out[212]: ('o', 4)
```

```
In [213]: d

Out[213]: {'h': 0, 'e': 1, 'l': 3}
```

```
In [214]: d.pop('l') # remove item with key `l`

Out[214]: 3
```

```
In [215]: d.pop('not-a-key') # remove a key not in dictionary: KeyError

          ---------------------------------------------------------------------------
          KeyError                                  Traceback (most recent call last)
          <ipython-input-215-9cda3d027920> in <module>()
          ----> 1 d.pop('not-a-key') # remove a key not in dictionary: KeyError

          KeyError: 'not-a-key'
```

```
In [216]: d.pop('not-a-key', 'default-value')
          # with a default value?
          # we get the default value

Out[216]: 'default-value'
```

```
In [217]: d.update({'another': 'value'})
          d.update(a=13)
          # we can update dict this way
          # or this way (like a function call)
          d

Out[217]: {'h': 0, 'e': 1, 'another': 'value', 'a': 13}
```

```
In [218]: d.get('a') # same as d['a'] but if key is missing no KeyError

Out[218]: 13
```

```
In [219]: d.get('a', 177) # default value used if key is missing

Out[219]: 13
```

```
In [220]: d.get('b', 177) # like in this case

Out[220]: 177
```

```
In [221]: d.get('b') # key is not there, so None is returned
```

Slices também podem ser usados para atribuição

```
In [222]: a = list(range(10))
          a

Out[222]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [223]: len(a)

Out[223]: 10

In [224]: a[5:8]

Out[224]: [5, 6, 7]

In [225]: a[5:8] = [10,20,30]

In [226]: a

Out[226]: [0, 1, 2, 3, 4, 10, 20, 30, 8, 9]
```

Número negativos podem ser usados para indexar a lista

```
In [227]: a[-1]

Out[227]: 9

In [228]: a[-2]

Out[228]: 8

In [229]: a[-5:-2]

Out[229]: [10, 20, 30]
```