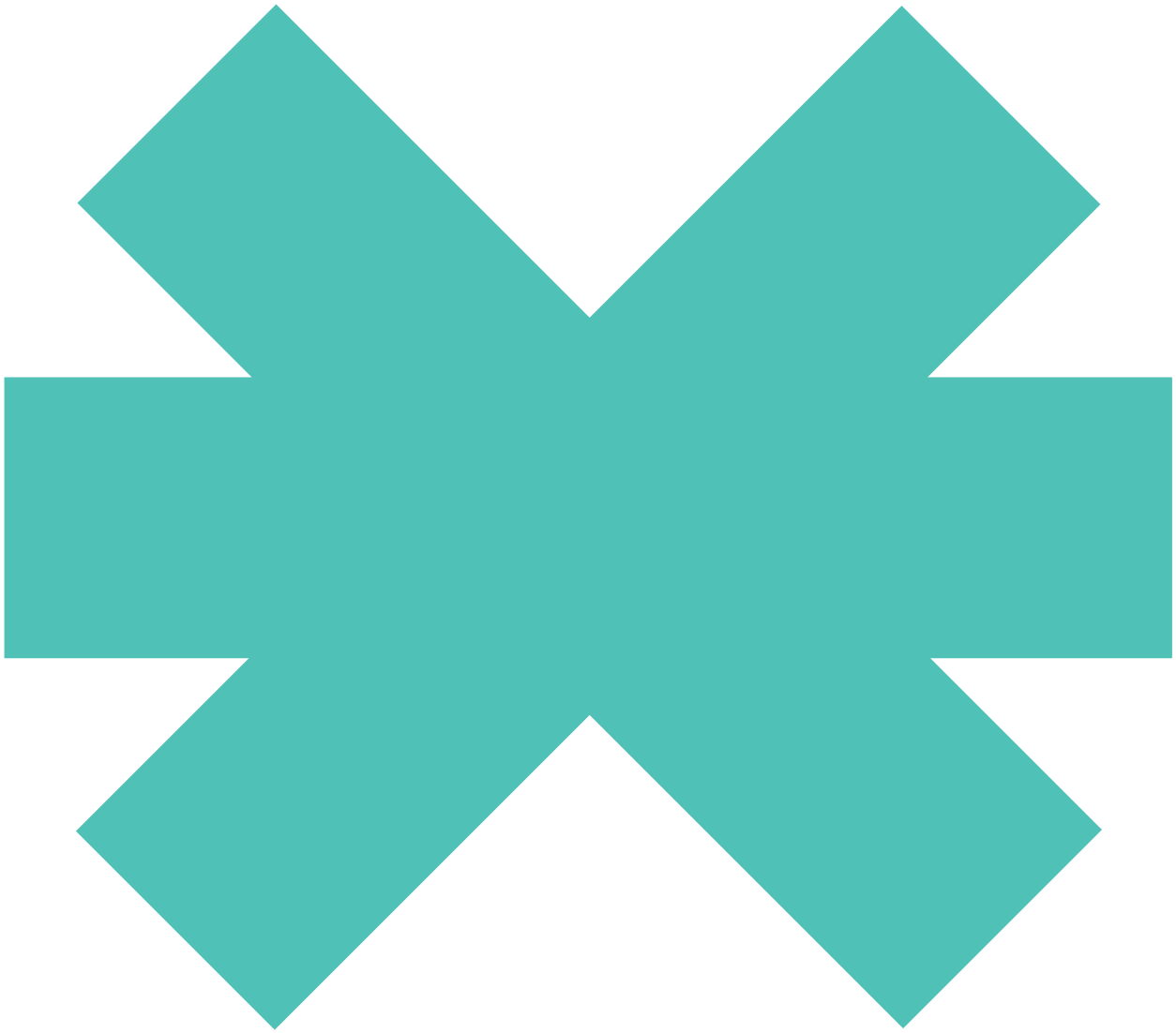
MÓDULO 2
Características básicas da linguagem
Programação Python

4

Tipos de dados avançados

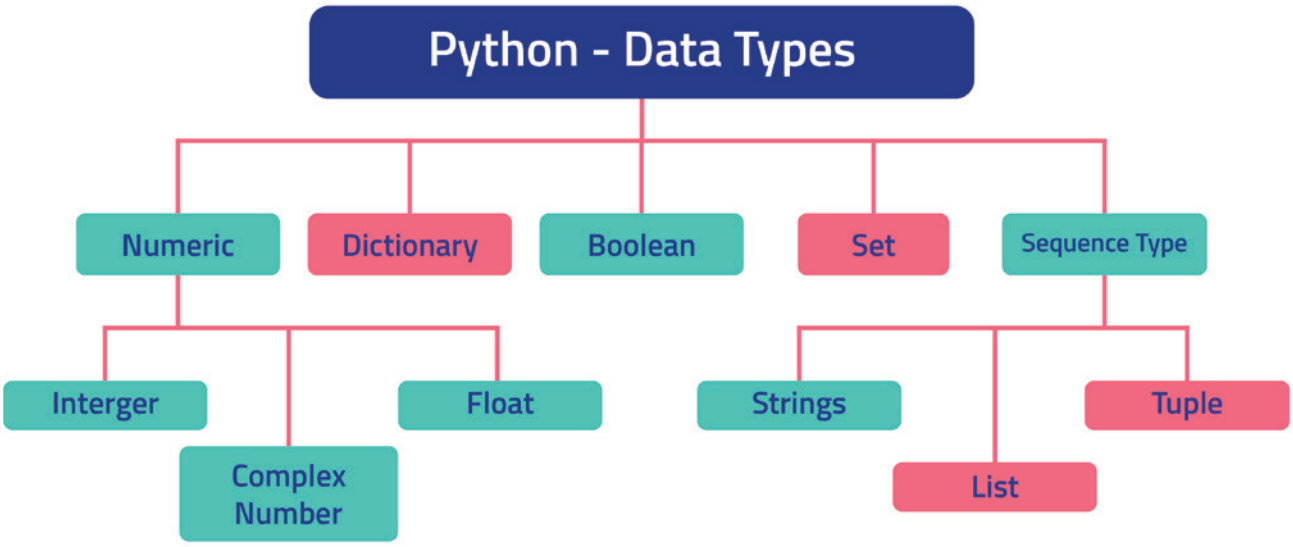
New Technology School

Tokio.



4 Tipos de dados avançados

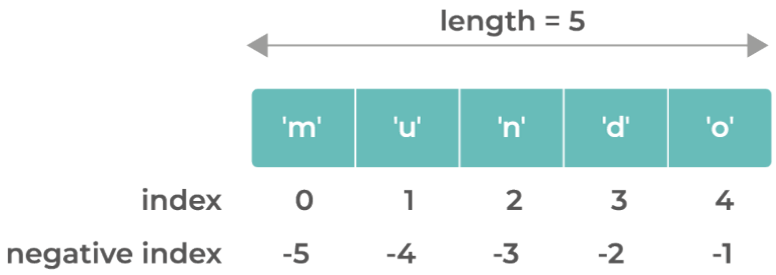
Sumário


Python - Data Types
Numeric
Dictionary
Boolean
Set
Sequence Type
Interger
Float
Complex Number
Strings
Tuple
List

4.1 Listas

Uma **lista** em Python é uma estrutura de dados, formada por uma sequência ordenada de objetos, é uma coleção organizada e transformável, que permite membros duplicados.

As listas em Python são heterogéneas porque podem ser formadas por elementos de distintos tipos, incluídos noutras listas e mutáveis, porque os seus elementos podem ser modificados.



Através dos índices de uma lista podemos modificar o valor dos seus elementos, gerando, desta forma, a lista mutável.

Os valores que compõem uma lista são dispostos entre [ ] e separados por vírgulas. Seguidamente, veremos como declarar uma lista, como a mostrar e verificar de que tipo é o elemento criado:

```
In [102]: numeros = [1,2,3,4]
          print(numeros)

          [1, 2, 3, 4]

In [103]: dados = [4,"Uma cadeia",-15,3.14,"Outra cadeia"]
          print(dados)

          [4, 'Uma cadeia', -15, 3.14, 'Outra cadeia']

In [104]: print(type(dados))

          <class 'list'>
```

Tanto o acesso aos elementos, realizado através de índices, como o *slice*, são realizados de forma muito semelhante às cadeias de caracteres, podendo também aceder aos elementos do final, com índices negativos, como vimos nas cadeias:

```
In [105]: print(dados)
          print(dados[0])

          [4, 'Uma cadeia', -15, 3.14, 'Outra cadeia']
          4

In [106]: print(dados[-1])

          Outra cadeia

In [107]: print(dados[-2:])

          [3.14, 'Outra cadeia']

In [108]: print(dados[1:3])

          ['Uma cadeia', -15]
```

As listas também aceitam o operador de soma, cujo resultado é uma nova lista que inclui todos os itens:

```
In [11]: numeros = numeros + [5,6,7,8]
         print(numeros)
         [1, 2, 3, 4, 5, 6, 7, 8]
```

As listas, tal como indicámos anteriormente, são modificáveis, e para isso alteramos o valor através dos seus índices:

```
In [6]: pares = [0,2,4,5,8,10]
        print(pares)
        [0, 2, 4, 5, 8, 10]

In [7]: pares[3]= 6
        print(pares)
        [0, 2, 4, 6, 8, 10]
```

Além disso, integram funcionalidades internas, como o método *.append*() usado para adicionar um item ao final da lista:

```
In [9]: pares.append(12)

In [10]: print(pares)
         [0, 2, 4, 6, 8, 10, 12]

In [11]: pares.insert(0, 7)

In [17]: pares.append(7*2)

In [12]: print(pares)
         [7, 0, 2, 4, 6, 8, 10, 12]
```

E têm uma peculiaridade; aceitam atribuição com *slice* para modificar vários itens em conjunto:

```
In [20]: letras = ['a','b','c','d','e','f']

In [21]: print(letras[:3])
         ['a', 'b', 'c']

In [22]: letras[:3] = ['A','B','C']

In [23]: print(letras)
         ['A', 'B', 'C', 'd', 'e', 'f']
```

Se queremos apagar o conteúdo de uma lista, apenas temos que atribuir-lhe uma lista vazia:

```
In [26]: letras[:3] = []

In [27]: print(letras)
         ['d', 'e', 'f']

In [28]: letras = []

In [29]: print(letras)
         []
```

Tal como acontece com as cadeias, a função *len*() retorna a quantidade de elementos que a lista contém, ou seja, o seu comprimento:

```
In [30]: print(len(letras))
         0

In [31]: print(len(pares))
         6
```

Para verificar a existência de um elemento dentro de uma lista usaremos o operador *in*, que nos retornará verdadeiro ou falso, em função de o elemento se encontrar ou não dentro da lista:

```
In [40]: print(pares)
         [0, 2, 4, 6, 8, 10]

In [41]: 2 in pares
Out[41]: True

In [42]: 7 in pares
Out[42]: False
```

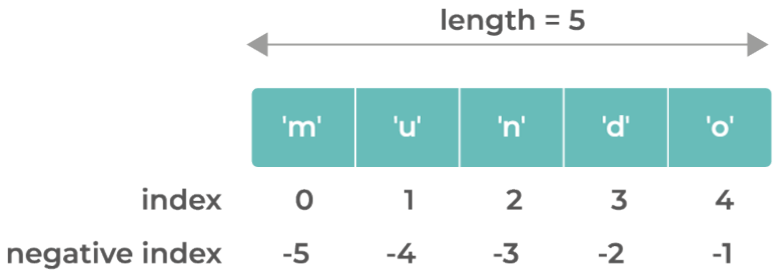
Podemos incluir listas dentro de listas, denominadas listas aninhadas (nested), e manipulá-las-emos facilmente através da utilização de índices múltiplos, como se nos referíssemos às linhas e colunas de uma tabela:

```
In [174]: a = [1,2,3]
          b = [4,5,6]
          c = [7,8,9]
          r = [a,b,c]

In [175]: print(r)
          [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

In [176]: print(r[0]) # Primeira sublista
          [1, 2, 3]

In [177]: print(r[-1]) # Ultima sublista
          [7, 8, 9]

In [178]: print(r[0][1]) # Segundo item da primeira sublista
          2

In [179]: print(r[1][1]) # Segundo item da segunda sublista
          5

In [180]: print(r[2][2]) # Terceiro item da segunda sublista
          9

In [181]: print(r[-1][-1]) # Ultimo item da ultima sublista
          9
```

4.2 Tuplas

Uma **tupla,** em Python, é uma estrutura de dados formada por uma sequência ordenada de objetos. Uma coleção organizada e imutável, que permite membros duplicados.

Podemos afirmar que as tuplas são listas imutáveis, que não podem ser modificadas depois de criadas.



Trabalha-se com as tuplas exatamente da mesma forma que com as listas. A única diferença é que as tuplas são imutáveis, não é possível modificar o seu conteúdo. Os valores que compõem uma tupla são dispostos entre [ ] e separados por vírgulas. Seguidamente, veremos como declarar uma tupla, como a mostrar e verificar de que tipo é o elemento criado:

```
In [1]: numeros = [1,2,3,4]
        print(numeros)

        [1, 2, 3, 4]

In [2]: dados = [4,"Uma cadeira",-15,3.14,"Outra cadeira"]
        print(dados)

        [4, 'Uma cadeira', -15, 3.14, 'Outra cadeira']

In [3]: print(type(dados))

        <class 'list'>
```

No que toca a índices e *slice*, as tuplas funcionam de uma forma muito semelhante às cadeias de caracteres e às listas:

```
In [4]: print(dados)
        print(dados[0])

        [4, 'Uma cadeira', -15, 3.14, 'Outra cadeira']
        4

In [5]: print(dados[-1])

        Outra cadeira

In [6]: print(dados[-2:])

        [3.14, 'Outra cadeira']

In [7]: print(dados[1:3])

        ['Uma cadeira', -15]
```

Também podemos realizar a soma de tuplas, cujo resultado será uma nova tupla que inclui todos os itens:

```
In [10]: numeros = numeros + (5,6,7,8)
         print(numeros)

         (1, 2, 3, 4, 5, 6, 7, 8)
```

É muito importante ter em conta que os elementos das tuplas não são modificáveis, qualquer tentativa de alteração dos mesmos produzirá um erro:

```
In [11]: pares = (0,2,4,5,8,10)
         print(pares)

         (0, 2, 4, 5, 8, 10)

In [12]: pares[3]= 6
         print(pares)

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-12-e23cd326fffe> in <module>
         ----> 1 pares[3]= 6
               2 print(pares)

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

Por não serem modificáveis não incluem o método *append*:

```
In [13]: pares.append(12)

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         <ipython-input-13-441fff7e2acb> in <module>
         ----> 1 pares.append(12)

         AttributeError: 'tuple' object has no attribute 'append'
```

E também não aceitam a atribuição com *slice*, para modificar vários itens em conjunto:

```
In [22]: letras = ('a','b','c','d','e','f')

In [19]: print(letras[:3])

         ('a', 'b', 'c')

In [20]: letras[:3] = ('A','B','C')

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-20-322fe606ae62> in <module>
         ----> 1 letras[:3] = ('A','B','C')

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

A função *len*() também funciona com as tuplas, da mesma forma que nas listas e cadeias de caracteres:

```
In [23]: print(len(letras))

         6
```

Tal como nas listas, podemos verificar a existência de um elemento com o operador *in*:

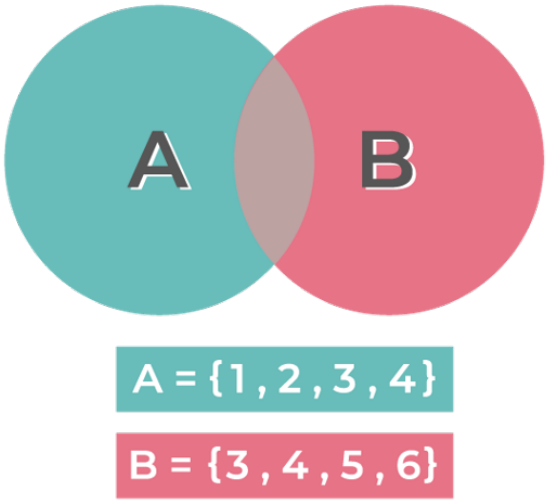
```
In [25]: print(pares)
         (0, 2, 4, 5, 8, 10)

In [26]: 2 in pares
Out[26]: True

In [27]: 7 in pares
Out[27]: False
```

E também podemos manipular tuplas aninhadas através da utilização de índices múltiplos, como se nos referíssemos às linhas e colunas de uma tabela:

```
In [189]: a = (1,2,3)
          b = (4,5,6)
          c = (7,8,9)
          r = (a,b,c)

In [190]: print(r)
          ((1, 2, 3), (4, 5, 6), (7, 8, 9))

In [191]: print(r[0]) # Primeiro subtuplo
          (1, 2, 3)

In [192]: print(r[-1]) # Ultimo subtuplo
          (7, 8, 9)

In [193]: print(r[0][1]) # Segundo item da primeiro subtuplo
          2

In [194]: print(r[1][1]) # Segundo item da segundo subtuplo
          5

In [195]: print(r[2][2]) # Terceiro item da segundo subtuplo
          9
```

4.3 Sets

Um **conjunto ou *set*** é uma coleção desordenada e não indexada na qual não são permitidos elementos repetidos. A utilização comum destes conjuntos inclui verificação de objetos e eliminação de entradas duplicadas.



Os valores que compõem um conjunto são dispostos entre { } e separados por vírgulas. Seguidamente, veremos como declarar um conjunto, como o mostrar e verificar de que tipo é o elemento criado:

```
In [208]: numeros = {1,2,3,4}
          print(numeros)

          {1, 2, 3, 4}

In [209]: dados = {4,"Uma cadeia",-15,3.14,"Outra cadeia"}
          print(dados)

          {3.14, 'Outra cadeia', 4, -15, 'Uma cadeia'}

In [210]: print(type(numeros))

          <class 'set'>

In [211]: print(type(dados))

          <class 'set'>
```

Não podemos aceder aos elementos de um **conjunto *set*** fazendo referência a um índice, pois os conjuntos não estão ordenados e devido a esse facto, os elementos não têm índice. Contudo, podemos examinar os elementos do conjunto, utilizando um *loop for*, que veremos posteriormente, ou perguntar se um valor específico está presente num conjunto, utilizando a palavra chave *in*:

```
In [214]: linguagem = {'Python' , 'C++' , 'Java'}
          for x in linguagem:
              print(x)

          Java
          Python
          C++

In [215]: print("Python" in linguagem)

          True
```

Uma vez criado um conjunto *set* não é possível alterar os seus elementos, mas podemos adicionar novos, através do método *add()*, para juntar um elemento a um conjunto ou através do método *update()*, para juntar mais de um elemento a um conjunto. Podemos observar como a ordem é totalmente aleatória e decidida pela linguagem e, por não aceitar elementos repetidos, se adicionarmos novamente um elemento que já existe, não será adicionado como um elemento novo:

```
In [7]: print(dados[1:3])
        ['Uma cadeira', -15]

In [8]: linguagens = {"Python" , "C++", "Java"}
        print(linguagens)
        {'Java', 'C++', 'Python'}

In [9]: linguagens.add("C#")
        print(linguagens) # Como se pode ver, a ordem é totalmente aleatória e decidido pela linguagem
        {'Java', 'C++', 'Python', 'C#'}

In [10]: linguagens.add("C#")
         print(linguagens) # Como se pode ver, a ordem é totalmente aleatória e decidido pela linguagem
         {'Java', 'C++', 'Python', 'C#'}

In [12]: linguagens.update(["Go","Javascript","PHP"])
         print(linguagens)
         {'PHP', 'C#', 'Java', 'C++', 'Python', 'Go', 'Javascript'}
```

A função *len*() também funciona para os conjuntos:

```
In [222]: print(len(linguagem))
          7
```

Para eliminar elementos do conjunto podemos utilizar dois métodos, *discard()* ou *remove()*, indicando entre parêntesis o elemento que queremos eliminar:

```
In [223]: linguagem = {'PHP', 'Go', 'Java', 'C#', 'Python', 'C++', 'Javascript'}
          print(linguagem)

          linguagem.remove("Go")
          print(linguagem)

          linguagem.discard("PHP")
          print(linguagem)
          {'PHP', 'Go', 'Java', 'C#', 'Python', 'C++', 'Javascript'}
          {'PHP', 'Java', 'C#', 'Python', 'C++', 'Javascript'}
          {'Java', 'C#', 'Python', 'C++', 'Javascript'}
```

Se queremos verificar a existência dentro do conjunto fá-lo-emos com o operador *in*:

```
In [224]: print(linguagem)
          {'Java', 'C#', 'Python', 'C++', 'Javascript'}

In [225]: "C#" in linguagem
Out[225]: True
```

E se o que pretendemos é eliminar todo o conteúdo de um conjunto, devemos utilizar o método *clear()*:

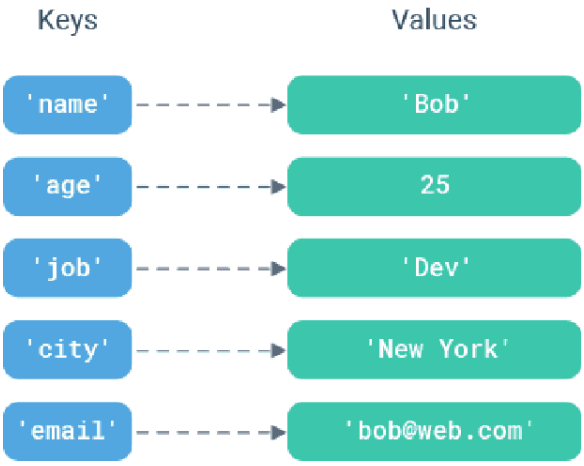
```
In [226]: linguagem = {'PHP', 'Go', 'Java', 'C#', 'Python', 'C++', 'Javascript'}
          print(linguagem)

          linguagem.clear()
          print(linguagem)

          {'PHP', 'Go', 'Java', 'C#', 'Python', 'C++', 'Javascript'}
          set()
```

Por último, devemos saber que os conjuntos não são aninháveis, logo não pode haver conjuntos dentro de outros conjuntos:

```
In [31]: a = {1,2,3}
         b = {4,5,6}
         c = {7,8,9}
         r = {a,b,c}

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-31-c6b53e85afab> in <module>
               2 b = {4,5,6}
               3 c = {7,8,9}
         ----> 4 r = {a,b,c}

         TypeError: unhashable type: 'set'
```

4.4 Dicionários

Um **dicionário**, em Python, é uma coleção desordenada, modificável indexada na qual não são permitidos elementos repetidos. Um dicionário define una relação única entre chaves e valores.



Os valores que compõem um dicionário são dispostos entre { } e separados por vírgulas. A estrutura principal é *chave* : *valor*. Seguidamente, veremos como declarar um dicionário, como o mostrar e verificar de que tipo é o elemento criado:

```
In [227]: veiculos = {
              "brand" : "Ford",
              "model" : "Mustang",
              "year" : 1984
          }
          print(veiculos)

          {'brand': 'Ford', 'model': 'Mustang', 'year': 1984}

In [228]: print(type(veiculos))

          <class 'dict'>
```

Para aceder aos elementos de um dicionário temos duas formas, fazendo referência à sua chave, ou utilizando o método *get()*. Ambas as formas retornarão o valor correspondente, como verificamos seguidamente:

```
In [231]: valorQueInteressa = veiculos["model"]
          print(valorQueInteressa)

          Mustang

In [232]: valorQueInteressa = veiculos.get("model")
          print(valorQueInteressa)

          Mustang
```

Para verificar a existência de uma chave num dicionário utiliza-se o operador *in* (serve apenas para verificar a existência de chaves):

```
In [233]: print("model" in veiculos)
```

```
True
```

Para modificar um valor faremos referência à sua chave:

```
In [234]: veiculos = {
              "brand" : "Ford",
              "model" : "Mustang",
              "year" : 1984
          }
          print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
```

```
In [235]: veiculos["year"] = 2020
          print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Para saber o comprimento de um dicionário usaremos a função *len()*, que também funciona com os dicionários:

```
In [236]: print(len(veiculos))
```

```
3
```

Para examinar um dicionário utilizaremos o mesmo método utilizado nos conjuntos:

```
In [245]: for x in veiculos:
              print(x)
```

```
brand
model
year
```

```
In [246]: for x in veiculos:
              print(veiculos[x])
```

```
Ford
Mustang
2020
```

```
In [247]: for x in veiculos:
              print(x, ": ", veiculos[x])
```

```
brand :  Ford
model :  Mustang
year :  2020
```

```
In [248]: for x in veiculos.values():
              print(x)
```

```
Ford
Mustang
2020
```

Existe um método dos dicionários, que nos facilita a leitura em chave e o valor dos elementos, porque retorna automaticamente, ambos os valores, em cada repetição:

```
In [249]: for x, y in veiculos.items():
              print(x, ":", y)
```

```
brand : Ford
model : Mustang
year : 2020
```

Para juntar elementos a um dicionário utilizaremos uma nova chave de índice e atribuir-lhe-emos um valor:

In [252]:
```
veiculos = {
    "brand" : "Ford",
    "model" : "Mustang",
    "year" : 1984
}
print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
```

In [253]:
```
veiculos["color"] = "red"
print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984, 'color': 'red'}
```

Para eliminar elementos do dicionário utilizaremos um destes três métodos, que nos seja mais conveniente: *clear*() para apagar todo o dicionário, *pop*() para eliminar o elemento com o nome de chave especificado e *popitem*() para eliminar o último elemento inserido. Há que ter em conta que *popiten()*, em versões anteriores à 3.7 do interpretador, elimina um elemento aleatório do dicionário em vez de eliminar o último elemento inserido:

In [254]:
```
veiculos = {
    "brand" : "Ford",
    "model" : "Mustang",
    "year" : 1984
}
print(veiculos)

veiculos.popitem()
print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
{'brand': 'Ford', 'model': 'Mustang'}
```

In [255]:
```
veiculos = {
    "brand" : "Ford",
    "model" : "Mustang",
    "year" : 1984
}
print(veiculos)

veiculos.pop("model")
print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
{'brand': 'Ford', 'year': 1984}
```

In [256]:
```
veiculos = {
    "brand" : "Ford",
    "model" : "Mustang",
    "year" : 1984
}
print(veiculos)

veiculos.clear()
print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
{}
```

Um dicionário não se pode copiar realizando uma atribuição entre dois dicionários da forma dict2 = dict1, porque o dicionário dict2 será apenas uma referência a dict1, e as modificações realizadas em dict1 também se realizarão automaticamente em dict2. Para copiar há que utilizar o método *copy()*:

In [257]:
```
veiculos = {
    "brand" : "Ford",
    "model" : "Mustang",
    "year" : 1984
}
print(veiculos)

veiculos_copia = veiculos.copy()
print(veiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
```

In [258]:
```
veiculos.pop("model") # Elimina o elemento com o nome de chave especifico
print(veiculos)
print(veiculos_copia) # Apesar do original ter aliminado um elemento, a copia mantem se intacta
```

```
{'brand': 'Ford', 'year': 1984}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1984}
```

Podem aninhar-se dicionários; incluir dicionários dentro de outros dicionários. Na imagem seguinte, o dicionário família contém outros três dicionários:

In [263]:
```
familia = {
    "child1" :{
        "name" : "Paulo",
        "year" : 2004
    },
    "child2" :{
        "name" : "Carlos",
        "year" : 2007
    },
    "child3" :{
        "name" : "Joana",
        "year" : 2011
    }
}

print(familia)
```

```
{'child1': {'name': 'Paulo', 'year': 2004}, 'child2': {'name': 'Carlos', 'year': 2007}, 'child3': {'name': 'Joana', 'year': 201
1}}
```

E para aceder aos dicionários "internos" faremos da forma seguinte:

In [264]:
```
valorQueInteressa = familia["child1"]
print(valorQueInteressa)
```

```
{'name': 'Paulo', 'year': 2004}
```

In [265]:
```
valorQueInteressa = familia["child3"]["name"]
print(valorQueInteressa)
```

```
Joana
```

Por último, nestas duas imagens vemos como incluir listas nos nossos dicionários:

In [268]:
```
dicionario = {
    'nome' : "Sara",
    'idade' : 22 ,
    'cursos' : ["Python", "Django" , "Javascript"]

}

print("Dicionario completo: ")
print(dicionario)

print("\nElementos do dicionario: ")
print(dicionario['nome'])
print(dicionario['idade'])
print(dicionario['cursos'])

print("\nItems da lista de cursos: ")
print(dicionario['cursos'][0])
print(dicionario['cursos'][1])
print(dicionario['cursos'][2])

print("\nListar o dicionario com um loop For")

for key in dicionario:
    print(key, ":", dicionario[key])
```

```
Dicionario completo:
{'nome': 'Sara', 'idade': 22, 'cursos': ['Python', 'Django', 'Javascript']}

Elementos do dicionario:
Sara
22
['Python', 'Django', 'Javascript']

Items da lista de cursos:
Python
Django
Javascript

Listar o dicionario com um loop For
nome : Sara
idade : 22
cursos : ['Python', 'Django', 'Javascript']
```

In [270]:
```
clientes = {
    'nome' : ["Tiago" , "Nuno" , "Maria"] ,
    'idade' : [32,24,45] ,
    'linguagem_favorita' : ["Python", "Django" , "Javascript"]

}

print("Dicionario completo: ")
print(clientes)

print("\nMostrar todos os dados do primeiro cliente")
print(clientes['nome'][0])
print(clientes['idade'][0])
print(clientes['linguagem_favorita'][0])

print("\nMostrar todos os dados do segundo cliente")
print(clientes['nome'][1])
print(clientes['idade'][1])
print(clientes['linguagem_favorita'][1])

print("\nMostrar todos os dados do terceito cliente")
print(clientes['nome'][2] , end = ", ")
print(clientes['idade'][2] , end = ", ")
print(clientes['linguagem_favorita'][2])
```

```
Dicionario completo:
{'nome': ['Tiago', 'Nuno', 'Maria'], 'idade': [32, 24, 45], 'linguagem_favorita': ['Python', 'Django', 'Javascript']}

Mostrar todos os dados do primeiro cliente
Tiago
32
Python

Mostrar todos os dados do segundo cliente
Nuno
24
Django

Mostrar todos os dados do terceito cliente
Maria, 45, Javascript
```