Lista, tupla, conjunto e controle de fluxo com `for`

Vimos em aulas anteriores os tipos de dados mais comuns, como `str`, `int`, `float` e `bool`. E vimos como armazenar dados na memória com o uso de variáveis. Exemplo:

```
num = 40
nome = "Rodolfo"
print("{} tem {} anos.".format(nome, num))
```

Muitas vezes, porém, precisamos armazenar mais de um valor numa variável. Por exemplo, uma coleção de preços, sendo maçã a 1 real, abacate a 1,25 real, uva a 3 reais. E agora? Como armazenamos múltiplos dados numa variável?

Para isso existem as **coleções de dados** do Python!

Há quatro coleções muito comuns, cada uma com características e funções próprias. Veremos três hoje.

Lista

A primeira é a **lista** (classe `list`), feita com valores dentro de colchetes (`[` e `]`) ou simplesmente chamando a função `list()`.

In [1]:
```
nomes = ["Rodolfo", "Cassie", "Rafael", "Nath"]
print(nomes)
print(type(nomes))
```

```
['Rodolfo', 'Cassie', 'Rafael', 'Nath']
<class 'list'>
```

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

```
[1, 2, 3, 4]
<class 'list'>
```

Posso também misturar tipos de dados dentro de uma lista...

In [3]:
```
mix = [12, "Cenoura", 3.72, True]
print(mix)
```

```
[12, 'Cenoura', 3.72, True]
```

...fazer lista de listas...

In [4]:
```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
listona1 = [lista1, lista2]
print(listona1)
```

```
[[1, 2, 3], [4, 5, 6]]
```

...e juntar várias listas numa só.

In [5]:
```
listona2 = lista1 + lista2
print(listona2)
```

```
[1, 2, 3, 4, 5, 6]
```

Para acessar cada elemento da lista, é preciso usar a posição do elemento dentro de colchetes. Mas lembre-se: **Python começa a contagem no índice 0** (ou seja, o primeiro elemento é 0, o segundo é 1, o terceiro é 2...).

In [6]:
```
print(listona2)
print(listona2[1])
```

```
[1, 2, 3, 4, 5, 6]
2
```

In [7]:
```
print(listona2[2] * listona2[4])
```

```
15
```

É possível acessar múltiplos elementos passando o índice de começo e de fim (mas o resultado exclui o último item):

In [8]:
```
print(listona2[2:4])  # [índice de começo:índice de fim - 1]
```

```
[3, 4]
```

"E como funciona essa indexação quanto temos uma lista de listas?" Vamos ver com um exemplo:

```
exemplo = [["vermelho", "amarelo", "azul"], ["verde", "roxo", "preto"]]
```

Neste caso, temos duas listas dentro de uma lista. Vamos chamas de "listas internas" e "lista externa". a "lista externa" tem dois elementos. Então, se eu chamar isso...

```
exemplo[1]
```

...terei como retorno isso:

```
["verde", "roxo", "preto"]
```

Se eu quero acessar `preto`, preciso então indicar o índice dentro da "lista interna" que acesso com `exemplo[1]`. Fica assim:

```
exemplo[1][2]
```

Ou seja, o terceiro elemento dentro da segunda lista de `exemplo`.

In [9]:
```
exemplo = [["vermelho", "amarelo", "azul"], ["verde", "roxo", "preto"]]
print(exemplo[1])
print(exemplo[1][2])
```

```
['verde', 'roxo', 'preto']
preto
```

Como na aula passada vimos `if-elif-else`, vale a gente ver o uso de controle de fluxo

com listas e apresentar o operador `in` ("está contido em") e `not in` ("não está contido em"):

```
num = 3
lista = [3, 5, 7, 9]
if num in lista:
    print("O número está na lista.")
else:
    print("O número não está na lista.")
```

In [10]:
```
num = 3
lista = [3, 5, 7, 9]
if num in lista:  # "Se o número estiver contido na lista..."
    print("O número está na lista.")
else:  #
    print("O número não está na lista.")
```

```
O número está na lista.
```

In [11]:
```
num = 6
lista = [3, 5, 7, 9]
if num in lista:
    print("O número está na lista.")
else:
    print("O número não está na lista.")
```

```
O número não está na lista.
```

In [12]:
```
nome = "Claudio"
lista_nomes = ["Renato", "Ana", "Fernanda"]
if nome not in lista_nomes:  # "Se o nome não estiver cna lista..."
    print("Nome não está na lista.")
else:
    print("Nome está na lista.")
```

```
Nome não está na lista.
```

In [13]:
```
nome = "Ana"
lista_nomes = ["Renato", "Ana", "Fernanda"]
if nome not in lista_nomes:
    print("Nome não está na lista.")
else:
    print("Nome está na lista.")
```

```
Nome está na lista.
```

As listas são **mutáveis**: posso adicionar e excluir elementos, mostrar em ordem reversa etc. com algumas funções:

- `.append(x)` para adicionar um elemento `x`
- `.pop(i)` para tirar da lista um elemento de índice `i` e mostrar esse elemento
- `.remove(x)` para tirar um elemento `x`
- `.reverse()` para inverter a ordem
- `.count(x)` para contar quantas vezes o elemento `x` aparece na lista
- `.sort([reverse=True])` para organizar os elementos do menor ao maior (ou do maior ao menor, se usar `reverse=True`)

In [14]:
```
lista = ["Carlos", "Antonio", "Cesar"]
print(lista)
lista.append("Rodolfo")
print(lista)
```

```
['Carlos', 'Antonio', 'Cesar']
['Carlos', 'Antonio', 'Cesar', 'Rodolfo']
```

In [15]:
```
print(lista.pop(2))
print(lista)
```

```
Cesar
['Carlos', 'Antonio', 'Rodolfo']
```

In [16]:
```
lista.remove("Antonio")
print(lista)
```

```
['Carlos', 'Rodolfo']
```

In [17]:
```
lista.reverse()
print(lista)
```

```
['Rodolfo', 'Carlos']
```

In [18]:
```
print(lista.count("Rodolfo"))
```

```
1
```

In [19]:
```
lista.append("Rodolfo")  # Adicionei mais um elemento "Rodolfo"
print(lista)
print(lista.count("Rodolfo"))
```

```
['Rodolfo', 'Carlos', 'Rodolfo']
2
```

In [20]:
```
lista.sort()
print(lista)
```

```
['Carlos', 'Rodolfo', 'Rodolfo']
```

In [21]:
```
lista.sort(reverse=True)
print(lista)
```

```
['Rodolfo', 'Rodolfo', 'Carlos']
```

Tupla

A segunda é a **tupla** (classe `tuple`), feita com valores dentro de parênteses (`(` e `)`) ou com a função `tuple()`.

In [22]:
```
valores = (1, 2, 99)
print(valores)
print(type(valores))
```

```
(1, 2, 99)
<class 'tuple'>
```

In [23]:
```
valores = (7.5, False, "Python")
print(valores)
print(type(valores))
```

```
(7.5, False, 'Python')
<class 'tuple'>
```

In [24]:
```
tupla = ((1, 2, 3), ("Anderson", "Matias", "José"))
print(tupla)
```

```
((1, 2, 3), ('Anderson', 'Matias', 'José'))
```

A forma de encontrar elemento é similar à de listas:

In [25]:
```
print(tupla[1])
```

```
('Anderson', 'Matias', 'José')
```

In [26]:
```
print(tupla[1][2])
```

```
José
```

Entretanto, as semelhanças acabam aí. Ao contrário de listas, **tuplas são imutáveis**. Ou seja, elementos não podem ser removidos, adicionados, reordenados etc.

In [27]:
```
vegetais = ("acelga", "repolho", "alface")
print(vegetais)
```

```
('acelga', 'repolho', 'alface')
```

In [28]:
```
vegetais.remove("acelga")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-28-a8d8c72289fb> in <module>
----> 1 vegetais.remove("acelga")

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

In [29]:
```
vegetais.append("rúcula")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-29-97283b0b32a0> in <module>
----> 1 vegetais.append("rúcula")

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

Daí a vantagem de tuplas: quando precisamos nos certificar de que os elementos de um conjunto não foram ou não serão alterados, elas são bastante úteis.

Conjunto

A terceira é o **conjunto** (classe `set`), feita com valores dentro de chaves (`{` e `}`) ou com a função `set()`.

In [30]:
```
conjunto = {1, 2, 3, 4, 5}
print(conjunto)
print(type(conjunto))
```

```
{1, 2, 3, 4, 5}
<class 'set'>
```

A diferença mais significativa entre listas e conjuntos é que conjuntos, ao contrário de listas, não retorna repetições.

In [31]:
```
conjunto = {1, 2, 2, 2, 3, 3, 4, 5}
print(conjunto)
```

```
{1, 2, 3, 4, 5}
```

Também ao contrário de listas, conjuntos não aceitam indexação...

In [32]:
```
print(conjunto[2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-929524e0cdef> in <module>
----> 1 print(conjunto[2])

TypeError: 'set' object is not subscriptable
```

...mas aceitam **iteração**. Vamos falar sobre isso.

Controle de fluxo com `for`

Agora que vimos três coleções de dados (`list`, `tuple` e `set`), podemos imaginar:

> E se o programador quiser realizar a mesma operação para cada item da coleção? Deve escrever tudo de novo?

A resposta é: **não**. Podemos usar controle de fluxo com `for` (ou `for-loop`, como é conhecido).

O `for-loop` itera (ou seja, repete) a operação ou o comando para cada item da coleção. Sua sintaxe é assim:

```
for elemento in colecao:
    print(elemento)
```

A tradução seria algo:

```
para cada elemento na coleção de elementos:
```

faça isso

Por exemplo:

> Tenho uma lista com cinco número e quero cada um elevado ao quadrado.

In [33]:
```
lista = [2, 4, 6, 8, 10]
for n in lista:
    print(n ** 2)
```

```
4
16
36
64
100
```

No exemplo acima, o computador fez o cálculo desejado (`n ** 2`) para cada item de `lista` .

E aqui eu posso ordenar ao sistema que realize qualquer operação:

In [34]:
```
nomes = ["José", "Manuel", "Carlos"]
for x in nomes:
    primeiras_letras = x[0:2]
    print(primeiras_letras.lower())
```

```
jo
ma
ca
```

In [35]:
```
nums = (6787, 6781, 789876, 43644)
for n in nums:
    if n % 2 == 0:
        valor = n / 2
        print(valor)
    else:
        valor = (n + 1) / 2
        print(valor)
```

```
3394.0
3391.0
394938.0
21822.0
```