Pensamento Computacional

2022/23

Aula 7

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Objetivos

- Iteração
  - A declaração **`while`**
  - O commando **`break`**
  - A declaração **`for`**
  - A função **`range`**

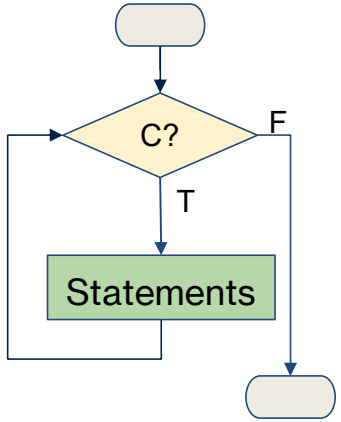
A declaração `while`

(cfr. enquanto...faça...)

- A declaração **`while`** dá indicações ao Python para **executar repetidamente** algum conjunto de declarações **enquanto uma dada condição for verdadeira**.

Sintaxe	Exemplo

`...`
`while condition:`
`    statements`
`...`
	`n = 3`
	`while n > 0:`
	`    print(n)`
	`    n = n-1`
	`print("Go!")`



A declaração `while`
(cfr. enquanto...faça...)

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

1. **Se a condição for verdadeira**, o bloco de commandos é executado.
2. Posteriormente, a condição é reavaliada e, **se ainda for verdadeira, o bloco de commandos é repetido**.
3. Quando a condição se torna **falsa, a execução avança** até à linha **imediatamente após** o bloco de comandos indentados.

- A condição deve ser uma expressão Booleana.
- Outros tipos de expressões são implicitamente convertidos para `bool`, desta forma, qualquer valor `null` ou vazio significará falso.

O comando `break`
(cfr. interrompa)

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- O **corpo do ciclo** deve alterar o valor de uma ou mais variáveis de forma a que, eventualmente, a condição se torne falsa e o ciclo termine.

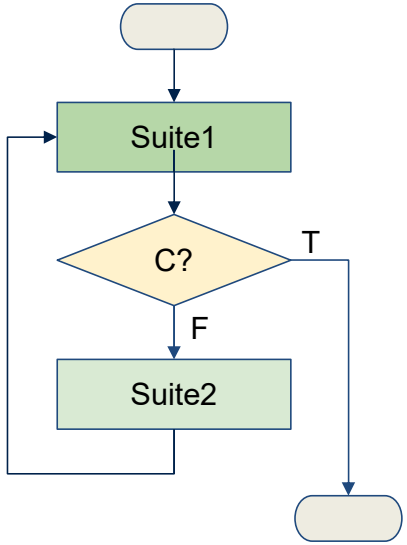
- Caso contrário, o ciclo irá repetir indefinidamente: *ciclo infinito*.

O comando `break` (cfr. interrompa)

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- Frequentemente, é necessário decidir se o ciclo deve terminar algures a meio. Nesse caso, usamos o commando **`break`** para saltar o ciclo.

```
while True:
    line = input('Enter text? ')
    if line == 'done':
        break
    print(line)
print('The end')
```



Ciclos com múltiplas saídas

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Por vezes, há várias condições que levam ao fim do ciclo e múltiplas localizações para as testar no corpo do ciclo.
- Para tal, usamos múltiplas declarações `if-break`.

```
while C1:
    Suite1
    if C2: break
    Suite2
    if C3: break
    Suite3
...
```

A declaração `for` (cfr. para...faça...)

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- Outro mecanismo de execução repetida é a declaração **`for`**.
- Esta repete um conjunto de comandos uma vez para cada item numa *coleção* de items, tal como uma lista, uma string ou um tuplo.

Sintaxe	Exemplo

`...`
`for var in collection:`
`    statements`
`...`
	`for n in [3, 1, 9]:`
	`    print(n)`
	`print("End")`

Expressão. Primeira coisa a ser avaliada.

Depois, o 1º item da coleção é atribuído à variável e o bloco de comandos é executado.

Posteriormente, o segundo item da coleção é atribuido a `var`, as declarações são novamente executadas, e por aí adiante até que a coleção tenha sido totalmente percorrida.

A função `range`

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- A função integrada `range` gera uma sequência de números em progressão aritmética.

```
list(range(4))  →  [0, 1, 2, 3]
```

- É frequentemente utilizada em ciclos **`for`**. Pode ser chamada com 1, 2 ou 3 argumentos:
  - `range(stop)`
  - `range(start, stop)`
  - `range(start, stop, step)`

```
for n in range(0, 4):
    print(n)
```

- Todos os argumentos devem ser **inteiros, positivos ou negativos**.
- Gera inteiro desde/até ao `stop`, **excluindo-o!**

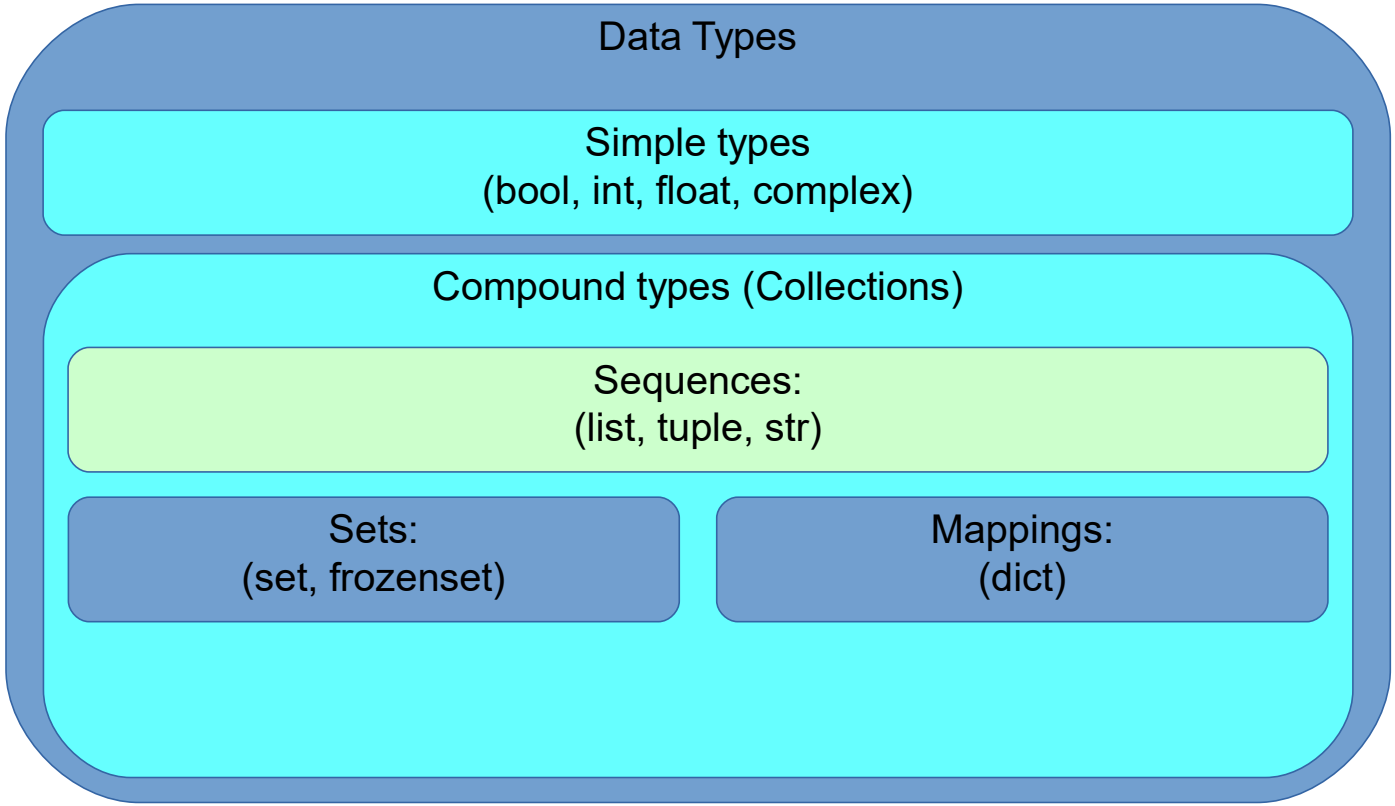
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Objetivos

- Estrutuuturas de dados em Python:
  - Sequências: Listas

Tipos de dados em Python

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO



Listas

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- A **lista** é uma sequência **mutável** de valores de qualquer tipo.
- Os valores numa lista são denominados *elementos* ou *items*.
- Os valores literais da lista são escritos entre parênteses retos.

```
numbers = [10, 20, 30, 40]
fruits = ['banana', 'pear', 'orange']
empty = []                          # uma lista vazia
things = ['spam', 2.0, [1, 2]]  # lista dentro de lista!
```

Listas

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- A função `len` retorna o comprimento (*length)* da coleção

```
numbers = [10, 20, 30, 40]
fruits = ['banana', 'pear', 'orange']
empty = []                              # uma lista vazia
things = ['spam', 2.0, [1, 2]]  # lista dentro de lista!


len(numbers)    #-> 4
len(empty)      #-> 0
len(things)     #-> 3
```

Indexação

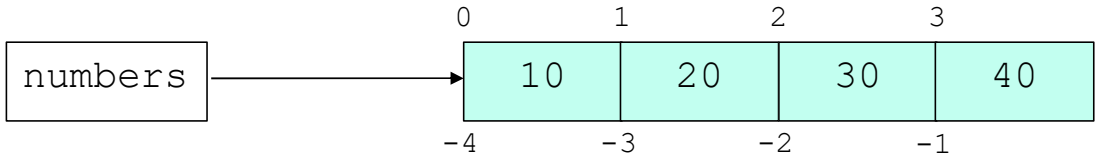
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Podemos aceder cada elemento de uma sequência usando o seu valor - o *índice.*

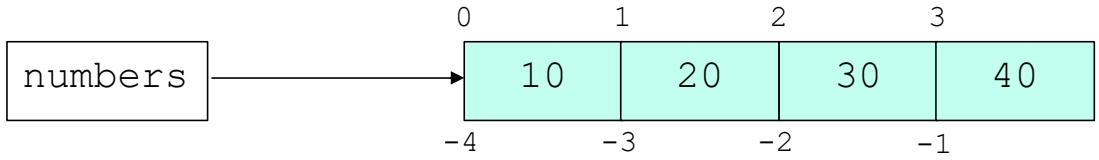
```
numbers[0]      #-> 10          (index starts at 0)
fruits[2]       #-> 'orange'
```

- Um índice negative conta regressivamente a partir do fim.

```
numbers[-1]     #-> 40
```



Indexação



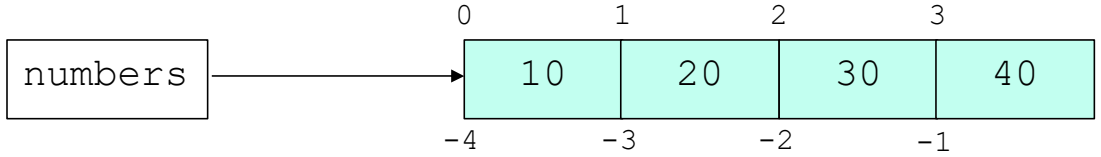
- Usar um índice fora dos limites da lista, irá gerar um erro.

```
numbers[4]     #-> IndexError
numbers[-5]    #-> IndexError
```

- Qualquer expressão inteira pode ser usada como índice.

```
numbers[(9+1)%4]   #-> 30
```

Partição



- Podemos extrair uma *subsequência* da lista usando **partições**.

```
numbers[1:3]   #-> [20, 30]
numbers[0:4:2] #-> [10, 30] (step = 2)
numbers[2:2]   #-> []
```

- Os índices negativos também podem ser usados.

```
numbers[-4:-2]  #-> [10, 20]
numbers[1:-1]   #-> [20, 30]
```

- Os índices iniciais e finais podem ser omitidos.

```
numbers[:2]  #-> [10, 20]
numbers[3:]  #-> [40]
numbers[:]   # a full copy of numbers
```

Percorrer uma lista

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- A forma mais comum de percorrer os elementos de uma lista é com um ciclo **`for`**.

```
for f in fruits:
    print(f)
```

```
banana
pear
orange
```

- Também podemos percorrer a sequência usando os índices:

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

- Neste caso também podemos usar o ciclo **`while`**.

```
i = 0
while i < len(numbers):
    print(i, fruits[i])
    i += 1
```

Operações em sequências

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- O operador **+** concatena e o * repete as sequências.

```
s = [1, 2, 3] + [7, 7]#-> [1, 2, 3, 7, 7]
s2 = [1, 2, 3]*2              #-> [1, 2, 3, 1, 2, 3]
s3 = 3*[0]                    #-> [0, 0, 0]
```

- O operador **`in`** verifica se um element está incluído na sequência. O operador **`not in`** faz o oposto.

```
7 in s        #-> True
4 not in s    #-> True
```

- Algumas das funções integradas podem ser aplicadas a sequências.

```
sum(s)    #-> 20
min(s)    #-> 1
max(s)    #-> 7
```

As listas são mutáveis

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- As listas são **mutáveis**, i.e., Podemos alterar o seu conteúdo.

```
numbers[1] = 99
numbers      #-> [10, 99, 20, 40]
```

- Podemos também alterar uma sublista.

```
numbers[2:3] = [98, 97]
numbers      #-> [10, 99, 98, 97, 40]
```

- Existem vários métodos que alteram o seu conteúdo:

```
lst = [1, 2]
lst.append(3)        # junta 3 ao fim de lst → [1, 2, 3]
x = lst.pop()        # lst → [1, 2], x → 3
lst.extend([4, 5])   # lst → [1, 2, 4, 5]
lst.insert(1, 6)     # lst → [1, 6, 2, 4, 5]
x = lst.pop(0)       # lst → [6, 2, 4, 5], x → 1
```

Aliasing e passagem de argumentos

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- *Aliasing* acontece quando passamos objetos como argumentos.
- Se o objeto é alterado dentro da função, isto é refletido também no exterior da mesma.

```
def grow(lst):
    lst.append(3)
    return lst
lst1 = [1, 2]
lst2 = grow(lst1)
print(lst1, lst2)  # What's the value of lst1 and lst2?
```

Aliasing e passagem de argumentos

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- Isto tem vantagens de eficiência, designadamente de memória.
- No entanto, caso não queiram que tal aconteça, é necessário fazer uma cópia antes de alterar.

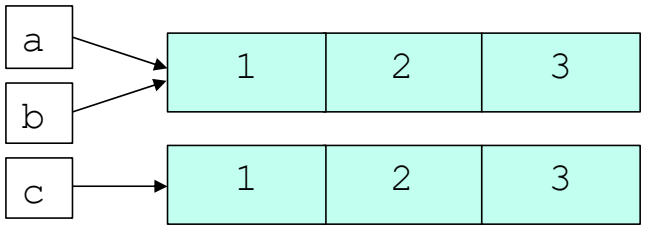
```
def grow(lst):
    r = lst[:]
    r.append(3)
    return r
```

Igualdade *versus* Identidade

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Os objetos podem ser *iguais* sem serem os *mesmos!*

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



- *Identidade* implica *igualdade*!
- Mas, igualdade não implica identidade.

Igualdade *versus* Identidade

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Testamos **igualdade com ==** (or **!=**).

```
a == b    #-> True          a != b    #-> False
a == c    #-> True          a != c    #-> False
```

- Testamos **identidade** com **is** (or **is not**).

```
a is b    #-> True          a is not b    #-> False
a is c    #-> False         a is not c    #-> True
```

Outros métodos úteis

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

`lst.append(item)`	Adicionar um item no final
`lst.insert(pos, item)`	Inserir um item numa dada posição
`lst.extend(collection)`	Adicionar todos os items da collection
`lst.pop()`	Remover o último item
`lst.pop(pos)`	Remover o item numa dada posição
`lst.remove(item)`	Remover a primeira ocorrência de um dado item (se existir algum)
`lst.index(item)`	Posição da primeira ocorrência de um dado item
`lst.count(item)`	Número de ocorrências de um dado item
`lst.sort()`	Oredenar os items de uma lista
`lst.reverse()`	Reverter a ordem dos items numa lista

Pensamento Computacional

2022/23

TPC: exercícios propostos aula 7