Pensamento Computacional

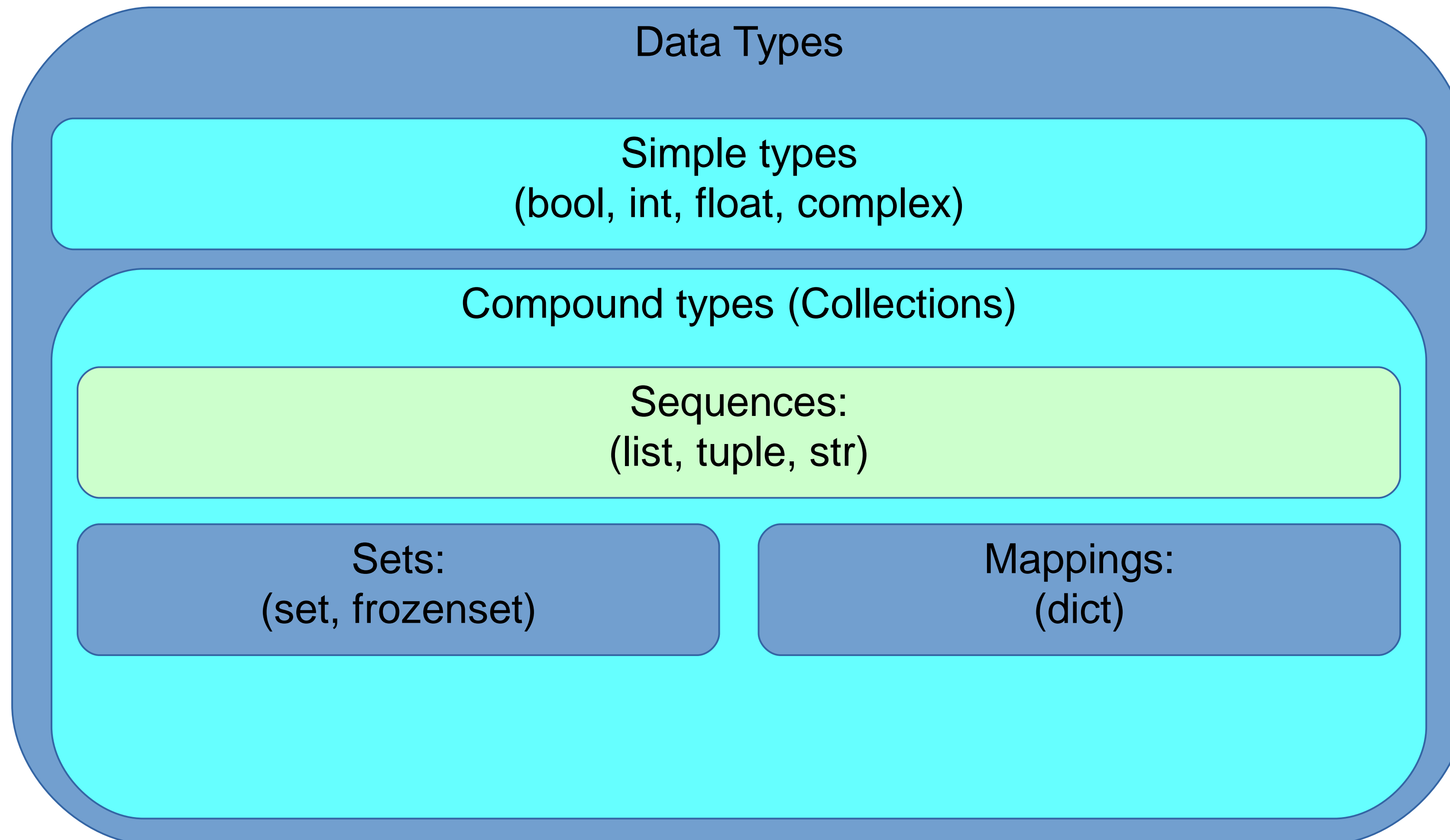
Aula 8

Objetivos

- Estrututuras de dados em Python:
  - Sequências:
    - Listas
    - ✓ Strings – vamos relembrar!
    - Tuplos

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
```

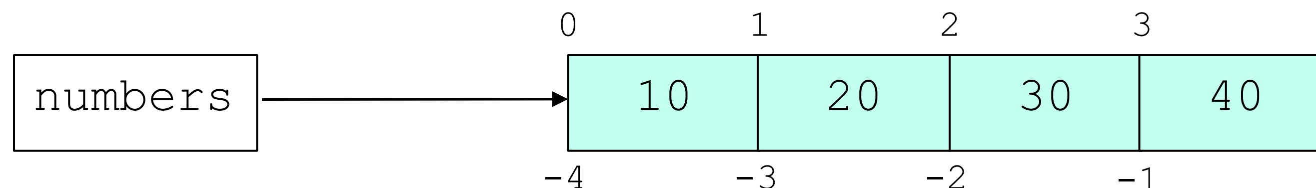
Indexação

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

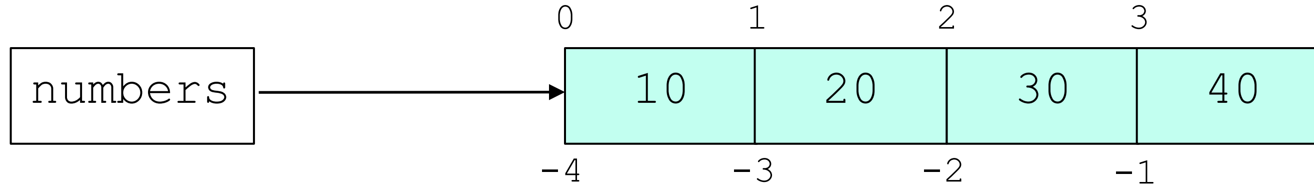
```
numbers[0]     #-> 10          (index starts at 0)
fruits[2]      #-> 'orange'
```

- Um índice negativo conta regressivamente a partir do fim.

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



Indexação



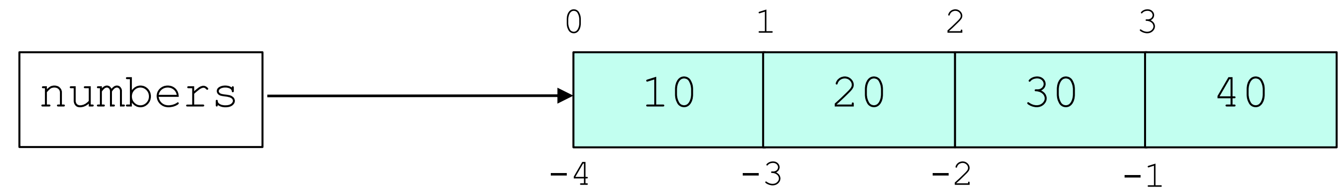
- Usar um índice fora dos limites da lista, irá gerar um erro.

```
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(fruits):
    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

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

```
numbers[1] = 99
numbers        #-> [10, 99, 30, 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.

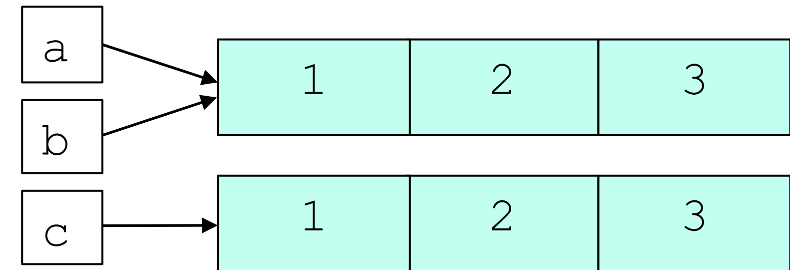
```
def grow(lst):
    r = lst[:]
    r.append(3)
    return r
```

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

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Strings – relembrar!

- *Strings* são sequências de carateres.
- Literais string são delimitados por aspas simples ou duplas..

```
fruit = 'orange'
```

- Função que retorna o número de carateres:

```
len(fruit)           #-> 6
```

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Strings – relembrar!

- Podemos aceder a um conjunto de carateres através da sua posição (**a primeira posição tem indíce 0**)

```
fruit[1]        #-> 'r'
fruit[3:5]      #-> 'ng'
fruit[:-1]      #-> 'orang'
fruit[::-1]     #-> 'egnaro'
```

- Podemos concatenar e repetir *strings*

```
name = 'tom' + 'cat'
#-> 'tomcat'
gps = 2 * 'tom'
#-> 'tomtom'
```

Strings - Relembrar!

- Ao contrário das listas, strings em *Python* são **imutáveis**. Quando uma string é criada não pode ser modificada.

```
fruit[0] = 'a'        #-> TypeError
```

- Mas, podemos criar novas strings através da combinação das existentes.

```
ape = fruit[:-1]+'utan'    #-> 'orangutan'
```

- Na realidade, mesmo os métodos que implicam alterações, apenas retornam um novo objeto string.

```
fruit.upper()             #-> 'ORANGE'
fruit.replace('a', 'A')   #-> 'orAnge'
fruit                     #-> 'orange' (not changed)
```

Percorrer strings

- Uma forma de percorrer strings é com um ciclo `for`:

```
fruit = 'banana'
for char in fruit:
    print(char)
```

- Outra forma:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- Outro exemplo:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

Exemplos

- O seguinte programa conta o número de vezes que a letra `'a’` aparece numa string:

```
word = 'banana'; count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

- Em strings, o operador `in` retorna `True` **se e só se (sse)** a primeira string aparece como uma substring na segunda.

```
for letter in word1:
    if letter in word2:
        print(letter)
```

Tuplos

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Um **tuplo** é uma **sequência imutável** de valores de qualquer tipo.
- Os valores num tuplo são indexados por inteiros, tal como nas listas. A diferença mais importante é que **os tuplos são imutáveis**.
- Sintaticamente, um tuplo é uma lista de valores separada por vírgulas.

```
t = 'a', 'b', 'c', 'd', 'e'
```

Tuplos

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- É comum, e por vezes necessário, rodear os tuplos em parênteses.

```
t = ('a', 'b', 'c', 'd', 'e')
```

- Para criar um tuplo com um único elemento, é necessário adicionar a vírgula final:

```
t1 = ('a',)
type(t1)     #->  <type 'tuple'>
```

- Outra forma de criar um tuplo,é usar a função incluída `tuple`. Quando não é passado qualquer argumento, esta cria um tuplo vazio:

```
t = tuple()                    # t → ()
```

Tuplos

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Se o argumento é uma sequência (string, list or tuple), o resultado é o tuplo com os elementos da sequência:

```
t = tuple('ape')        # t → ('a', 'p', 'e')
t = tuple([1, 2])       # t → (1, 2)
```

- A maior parte dos operadores de lista também funcionam em tuplos.
- Não podemos alterar os elementos de um tuplo, mas podemos substituir um tuplo por outro.

```
t = t + (3, 4)               # t → (1, 2, 3, 4)
```

Listas e Tuplos

- A função integrada `zip` aceita duas ou mais sequências e gera uma sequência de tuplos, em que cada contem um elemento de cada sequência.

```
s = 'abc'
t = [4, 3, 2]
list(zip(s, t)) # → [('a', 4), ('b', 3), ('c', 2)]
```

- A função `enumerate` gera uma sequência de pares (índice, item).

```
enumerate('abc')    # → (0, 'a'), (1, 'b'), (2, 'c’)
```

- Pode usar-se num ciclo **`for`** para percorrer uma sequência de tuplos:

```
s = 'somestuff'
for i, c in enumerate(s):
    print(i, c)
```

Objetivos

- Ficheiros
- Exceções e afirmações (*assertions*)

Ficheiros de texto

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Os programas que vimos até à data são transitórios, i.e., correm num período reduzido, recebem um entrada e produzem uma saída. No entanto, quando terminam, tudo desaparece!
- De forma a manter os dados relacionados, podemos usar **ficheiros de texto**.
  - Um ficheiro de texto é uma sequência de carateres guardados num local persistente como um disco, ou um cartão de memória.
  - Os carateres são *codificados* em bytes de acordo com uma tabela de código standard, tal como ASCII, Latin-1 or UTF-8.

Abrir e fechar ficheiros

- Antes de usar um ficheiro, temos de o preparar para a leitura e escrita.
- A função integrada `open` recebe o nome de um ficheiro e devolve um objeto `file` que podemos usar para aceder ao mesmo.

```
fileobj = open(file_name, 'r')   # open for reading
fileobj = open(file_name, 'w')   # open for writing
```

- Outros modos: `'r'`, `'w'`, `'a'`, `'r+'`, `'w+'`, `'a+'`, ...

Abrir e fechar ficheiros

- Após o uso do ficheiro, deve lembrar-se de o **fechar (close)**!

```
fileobj.close()
```

- A declaração **`with`** permite fechar automaticamente os ficheiros.

```
with open(file_name, mode) as fileobj:
    statements to read/write fileobj
# fileobj.close() not required!
```

Leitura de um ficheiro

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Podemos usar um ciclo `for` para ler um ficheiro *linha a linha*.

```
fin = open('words.txt')
for line in fin:          # for each line from the file
    print(repr(line))     # do something with it
fin.close()
```

Leitura de um ficheiro

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Também podemos usar o método `readline`:

```
while True:
    line = fin.readline()     # returns line to the end
    if line == "": break      # empty means end-of-file
    print(repr(line))
```

- Podemos, também, ler o ficheiro inteiro como uma *string*.

```
text = fin.read()  # read as much as possible (up to EOF)
```

- Ou ler os primeiros N carateres.

```
str = fin.read(10) # read upto 10 chars (empty means EOF)
```

Mover a posição do objeto file

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- Regra geral, lemos e escrevemos sequencialmente, do início ou fim. Mas, por vezes, precisamos de “saltar” conteúdo.
- O método `tell()` indica a posição atual no ficheiro.
- Por outro lado, o método `seek(offset)` altera a posição atual no ficheiro para a posição `offset` bytes do início.

```
a0 = f.readline() # read a line
pos = f.tell()         # store position
a1 = f.readline() # read second line
f.seek(pos)            # return to stored position
a2 = f.readline() # read second line again (a2==a1)
```

Escrita num ficheiro

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Para escrever num ficheiro, precisamos de o abrir no modo `'w'` (ou `'a'`).

```
fout = open('output.txt', 'w', encoding='utf-8')
```

- Ao abrir um ficheiro em modo `'w'`, criamos um novo ficheiro ou truncamos um já existente. Ao abrir em modo `'a'` o ficheiro não é truncado.

- O método `write` permite adicionar dados ao ficheiro.

```
line1 = "To be or not to be,\n"
fout.write(line1)
```

Escrita num ficheiro


```
line1 = "To be or not to be,\n"
fout.write(line1)
```

- Mais uma vez, o objeto ficheiro guarda a informação sobre a localização atual para que, ao chamar novamente `write`, este adicione os novos dados no final do ficheiro.

```
line2 = "that is the question.\n"
fout.write(line2)
```

Escrita num ficheiro

- **Lembre-se:** O argumento de `write` tem de ser uma string! Logo, temos de converter valores de outros tipos.

```
x = 0.75
fout.write('X: ' + str(x))
```

- Ou usar o método de formatação de strings.

```
fout.write('{} costs {:.2f}€.'.format('tea', x))
```

Escrita num ficheiro

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- Também pode usar o `print` com o argumento `file=` .

```
print('X:', x, file=fout)
print('{} costs {:.2f}€.'.format('tea', x), file=fout)
```

- Quando terminar a escrita no ficheiro, lembre-se de fechar o mesmo!

```
fout.close()    # OR use the with statement
```

Exceções

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- Em Python existe uma ferramenta importante para lidar com eventos inesperados no programa - **exceções**.

- Já terá visto algumas exceções na resolução dos exercícios das aulas práticas de PC:

```
int(“hello world")        #-> ValueError: invalid literal for
                          # int()

open("foo")          #-> FileNotFoundError: No such file…
```

Exceções

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Quando é encontrada uma situação com a qual não consegue lidar, o Python **identifica/levanta** uma exceção.

- Tal, interrompe o fluxo normal da execução: a tarefa atual é interrompida, e todas as seguintes de forma sucessiva até à interrupção do próprio programa.

- A informação sobre o evento que levou a esta interrupção é transmitida num objeto *exception*.

Lidar com exceções

A declaração `try`

- Podemos intercetar exceções selecionadas e regressar à normal execução do programa através da declaração **`try`**.

- **Example: handle errors accessing files:**

```
try:
    fh = open("testfile", "r")
    content = fh.read()
except IOError:
     print("Error: could not open file or read data")
else:
     print("This executes iff no exception occurred")
    fh.close()
```

- **`except`** pode nomear múltiplas exceções.
- um **`except`** que não identifique uma exceção especifica, apanhará todo o tipo de exceções.

Informação de uma exceção

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

- Uma exceção pode ter um *argumento* que nos permita obter informação adicional sobre o problema.

```
def temp_convert(var):
    try:
        return int(var)
    except ValueError as e:
        print("Not numeric:", e)

temp_convert("123")
temp_convert("xyz")
```

Levantar exceções

- É possível levantar exceções (de qualquer tipo) usando a declaração **`raise`**.

```
def checkLevel( level ):
    if level < 1:
        raise Exception(f"level={level} is too low!")
    # code here is not executed if we raise the exception
    return level

try:
    v = checkLevel(-1)
    print("level = ", v)
except Exception as e:
    print("Error:", e)
```

Afirmações (*Assertions*)

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

- Há alturas em que sabemos e, por vezes, exigimos que uma dada condição seja verdadeira em algum momento do programa – uma **afirmação** ou **assertion.**
- Podemos usar a declaração **`assert`** para validar condições. Caso a mesma seja falsa, é levantada uma exceção.
- Podemos posicionar estas validações no início de uma função – para garantir um input válido – ou após uma função – neste caso, para garantir um output válido.

Afirmações (*Assertions*)

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Exemplo

```
def KelvinToFahrenheit(Temperature):
     assert Temperature >= 0, "Should exceed absolute zero!"
    return ((Temperature-273)*1.8)+32
print(KelvinToFahrenheit(-5))
          #-> AssertionError: Should exceed absolute zero!
```

Pensamento Computacional

TPC: exercícios propostos aulas 7 e 8