UFCD 10793 - Fundamentos de Python

Notebook 01 - Conceitos básicos da linguagem Python

SILRMAR

2022

A linguagem Python

As **palavras reservadas** seguintes constituem uma parte significativa da linguagem Python:

In [1]:

```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Podes também obter a lista de todos os símbolos em Python a partir da execução da função *help()*:

help()

Funções embutidas

O interpretador do Python possui várias funções e tipos embutidos que sempre estão disponíveis. A lista de todas as funções embutidas no interpretador Python podem ser acedidas e estudadas em docd.python.org (https://docs.python.org/pt-br/3/library/functions.html), sendo a função `help()` uma delas.

Bibliotecas padrão Python

O ambiente Python tem por defeito um conjunto de módulos predefinidos que contém as funções normalmente aplicadas em diferentes domínios, tais como os módulos `math` , `time` , `random` , `string` , entre outras que definem bibliotecas de software e de funções que são aplicadas no desenvolvimento e implementação de programas.

Exemplos de bibliotecas padrão do Python:

In [ ]:

```
import os
#interação com o sistema operativo (os)

import sys
#interação com o sistema Python

import string
#string: biblioteca de funções de manipulação de strings (cadeia de carateres)
import math
#math:biblioteca de funções matemáticas
print(math.sqrt(25))
import time
?time
#time: biblioteca de funções data e hora
print(time.asctime())
```

Utilização de comentários

Uma das boas práticas de programação consiste em comentar o que escrevemos por forma a explicitar, para nós e principalmente para os outros, o significado de cada pedaço de código.
Os comentários podem ser usados para tornar o código mais legível ou explicá-lo.
Linhas começadas por **#** são consideradas comentários, e portanto não são avaliadas pelo interpretador.

Exemplos:

In [ ]:

```
#isto é um comentário
#é também um comentário
print("Bom ano letivo!")
```

In [ ]:

```
"""para várias linhas podemos usar também aspas triplas
para identificar um comentário"""
print("Bom ano letivo!")
```

Nomes (variáveis), valores e tipos de dados

Nomes em Python

Uma nome pode ser curto (como x e y) ou mais descritivo (nome, idade, num_id). Em Python, um nome é uma qualquer sequência de carateres (sem aspas, não se trata de uma string) que segue as seguintes regras:

- Deve começar com uma letra ou pelo caráter underscore (travessão baixo);
- Não pode começar com um número;
- Pode conter apenas carateres alfanuméricos e underscore (Az, 0-9 e _) ;
- São case sensitive (idade e IDADE são variáveis diferentes).

Na terminologia das linguagens de programação é usual chamar variáveis aos nomes. No entanto a linguagem Python dá um uso particular ao mecanismo de associação de valores a nomes que a distingue da maioria das linguagens de programação. Nomes começados por _ são possíveis, mas convenciona-se que têm um significado especial.
Convenciona-se ainda que nomes cujo primeiro carater é uma letra maiúscula são usados para definir classes de objetos. Para já, usaremos apenas nomes iniciados com uma letra minúscula.

Como já visto, há vários nomes reservados, predefinidos na linguagem Python, que não se podem usar.

Tipos de valores

Em Python, cada valor (objeto) pertence a um determinado tipo (classe). Esse tipo determina quais as operações que podem ser aplicadas a esse valor (objeto). Trata-se de uma linguagem tipada, mas onde os tipos não são explícitos, o que facilita bastante o trabalho do utilizador (apesar de poder trazer outro tipo de problemas, a jusante, na depuração de erros).

Os tipos básicos disponíveis mais úteis são:
números inteiros (`int`), números reais (`float`), complexos (`complex`) e valores lógicos (`bool`) que são tipos imutáveis (https://pt.linuxteaching.com/article/mutable_vs_immutable_objects_in_python#how_do_you_check_if_an_object_is_mutable_in_python).

-> Todos os objetos em Python têm o seu próprio `id` exclusivo, sendo-lhe atribuído quando este é criado. -> O id é o endereço de memória do objeto e será diferente a cada execução do programa (exceto para algum objeto que tem um id único constante, como inteiros de -5 a 256).

Abaixo exemplos de valores inteiros, reais e complexos:

A função *type(objeto)* indica o tipo de dados de um determinado objeto

In [ ]:

```
type(3)
```

```
The kernel failed to start as a dll could not be loaded.

Click <a href='https://aka.ms/kernelFailuresDllLoad'>here</a> for more info.
```

A função *id(objeto)* indica o ID exclusivo para o objeto especificado

In [ ]:

```
id(1)
```

In [ ]:

```
type(1.0)
```

In [ ]:
```
id(1.0)
```

In [ ]:
```
id(1j)
```

In [ ]:
```
type(1j)
```

Podes conhecer outras funções incorporadas no Python em W3Schools (https://www.w3schools.com/python/python_ref_functions.asp)

Operadores e suas prioridades

Operadores

- Operadores aritméticos: `+` , `-` , `*` , `/` , `//` (divisão inteira), `**` (potênciação) , `%` (resto da divisão inteira)
- Operadores lógicos (mais utilizados): `and` (conjunção), `not` (negação), `or` (disjunção)
- Operadores relacionais (de comparação), como os da matemática: `<` , `>` , `<=` , `>=` , `==` , `!=`
- Operador de atribuição: `=` , `-=` , `=` , `/=` , `//=` , `%=` , `*=`
- Operadores de identidade (servem para a comparação de objetos): `is` , `is not`
- Operadores de associação (são utilizados para verificar se uma sequência contém um objeto): `in` , `not in`

> **Tarefas:**
>
> - Consulta a lista de operadores e operações com operadores em W3Schools (https://www.w3schools.com/python/python_operators.asp)
> - Deves rever a tabela de verdade para as operações de conjunção, disjunção e negação.

Prioridade dos Operadores (da maior para a menor prioridade):

`() ->Parêntesis`

`**, not`

`* , /, %, //, and`

`+, -, or`

`<=, <, >, >=`

`==, !=`

`=, %=, /=, //=, -=, +=, *=`

Expressões numéricas

Exemplos:

In [ ]:
```
1+1-3
```

In [ ]:
```
2*(3%5)
```

In [ ]:
```
7/3*5.9
```

In [ ]:
```
7//3
```

In [ ]:
```
10**3
```

In [ ]:

```
abs(-1)*max(1,2,3)+min(1,2,3)+round(2.6)
```

In [ ]:

```
1/3+1/2
```

São exemplos de números reais utilizando a `notação científica` , 4.2e5 (=420000.0) e -6e-8 (=-0.00000006).
A notação científica é utilizada, principalmente, para representar números muito grandes ou muito pequenos.
e corresponde à versão simbólica do nº de Napier e faz parte das funcionalidades para cálculo simbólico, suportadas pela extensão Sympy.

In [ ]:

```
from sympy import *
```

e+E

In [ ]:

```
200000000000000000000000000000000.0*2  # 4e+35
```

In [ ]:

```
.00000000000000000001 # 1e-20
```

Nomes e atribuição

É possível, frequente e desejável, associar nomes a valores (objetos), por forma a memorizá-los e ter um mecanismo simples para os referir. O mecanismo de associação de um nome a um valor é conhecido por atribuição.

Exemplos:

In [ ]:

```
and=3
```

In [ ]:

```
a=5
```

In [ ]:

```
a+a**2 #o mesmo que print(a+a**2)
```

In [ ]:

```
b=a
```

In [ ]:

```
b # o mesmo que print(b)
```

In [ ]:

```
a=a+1
```

O objeto que representa o valor 5 vai sempre representar o valor 5. A atribuição a=a+1 associa então ao nome ao objeto que corresponde a avaliar 5+1, que representa o valor 6. No entanto, o objeto associado a b não é alterado, mantendo-se o valor 5.

In [ ]:

```
a
```

In [ ]:

```
b
```

In [ ]:

```
frutas = ["banana","laranja","uva","ameixa"] # lista de objetos

fruta_1 = "ameixa"
fruta_2 = "melancia"

print(fruta_1 in frutas) # True
print(fruta_2 in frutas) # False
```

In [ ]:

```
import math
val=math.pi
math.cos(val)
```

Muitos valores para múltiplas variáveis:

In [ ]:

```
x, y, z = "Laranja", "Banana", "Cereja"
print(x)
print(y)
print(z)
print(x, "", y," ", z)
```

Múltiplas variáveis com o mesmo valor:

In [ ]:

```
x = y = z = "Banana"
print(x)
print(y)
print(z)
```

Tipo *bool*

O tipo bool (booleano ou lógico) disponibiliza as constantes `True` e `False`.

Exemplos da sua utilização abaixo:

In [ ]:

```
(1==1)
```

In [ ]:

```
type(1==1)
```

In [ ]:

```
1!=3 and (3<1 or (not 0==1))
```

In [ ]:

```
2 == 1+1
```

In [ ]:

```
2==2.0
```

Os operadores lógicos `is` e `is not` comparam os identificadores dos objetos em causa:

In [ ]:

```
carlos=1
luciano=carlos
if carlos is luciano:
    print("carlos - luciano = mesmo objeto")
else:
    print("carlos - luciano = objetos diferentes")
```

In [ ]:

```
2 is 3 # id(2)==id(3)
```

Diferença entre os operadores `is` e `==`

`==` é para igualdade de valor. É usado para saber se dois objetos têm o mesmo valor. `is` é para igualdade de referência. Determina se dois objetos são idênticos, isto é, se têm o mesmo endereço de memória.

A declaração `is` , aplicada aos objetos **a** e **b** é equivalente a: `id(a) == id(b)`

Exemplo, onde podemos observar a diferença descrita acima:

In [ ]:

```
a=[1,2,3]  # isto é uma lista
```

In [ ]:

```
b=a
```

In [ ]:

```
c=a[:] # c=[1,2,3] mas não necessáriamente armazenado nos mesmos endereços de memória
print(c)
id(c)
```

In [ ]:

```
id(a)
```

In [ ]:

```
a==b
```

In [ ]:

```
a==c
```

In [ ]:

```
a is b
```

In [ ]:

```
a is c
```

A função `isinstance(objeto, classe)` testa a pertença de um valor (objeto) a um tipo (classe).

Exemplos:

In [ ]:

```
isinstance(2.0,int)
```

In [ ]:

```
x = isinstance(6, float)
print(x)
```

Outros tipos de valores

Além dos tipos básicos, há também os tipos sequenciais, ou iteráveis: cadeias de carateres (`str`), listas (`list`), vetores (`tuple`), registos (`dictionary`), conjuntos (`set`), e ficheiros (`files`).
Cadeias de carateres, vetores e ficheiros são tipos imutáveis , enquanto que listas, registos e conjuntos são tipos mutáveis (embora possam ser constituidos por tipos imutáveis). É possível ainda introduzir novos tipos, por definição das respetivas classes.

Daremos de seguida atenção às cadeias de carateres (deixando os restantes tipos referidos para mais tarde).

As cadeias de carateres, ou *strings*, são representadas em Python usando aspas (") ou plicas ('). Estão disponíveis diversas operações para manipular *strings*.

Plicas simples: 'permitem "aspas", internas'

Aspas: "permitem 'plicas' internas"

Plicas triplas: '''Três plicas simples''', """Três aspas"""

Exemplos:

In [ ]:
```
type("bom")
```

In [ ]:
```
"bom"+" "+"dia" #concatena as duas string
```

In [ ]:
```
"bom dia"[0] # obtém o carater que se encontra na posição 0 (o 1.º)
```

In [ ]:
```
"bom dia"[1] # obtém o caráter que se encontra na posição 1 (o 2.º)
```

In [ ]:
```
"bom dia"[-2] # obtém o caráter que se encontra duas posições antes do fim da string (fora da string)
```

In [ ]:
```
"bom dia"[2:5] # obtém os carateres da posição 2 (3.º) à posição 5-1
```

In [ ]:
```
"bom dia"[4:5]
```

In [ ]:
```
"bom"/2
```

As cadeias de carateres podem ainda ser manipuladas recorrendo a métodos específicos, como veremos posteriormente.

Casting | conversão de tipos

O Python disponibiliza também várias funcionalidades que permitem converter valores entre diferentes tipos.

Exemplos:

In [ ]:
```
print (int("523")+1) #será exibido 524
y = int(2.8) # y será 2
z = int("3") # z será 3
print (y,"\t",z)
```

In [ ]:
```
int(2.9)
```

In [ ]:
```
float(2)
```

In [ ]:
```
"depois de "+str(123)+" vem "+str(124)
```

In [ ]:
```
chr(123) # "123"
```

In [ ]:
```
round(3.6)
```

In [ ]:
```
round(3.2)
```

In [ ]:
```
bool("") #false
```

In [ ]:
```
bool("abc") #true
```

In [ ]:
```
bool(0)
```

In [ ]:
```
str(4.78)
```

In [ ]:
```
float("4.78")
```

Expressões alternativas

Uma forma útil de construir expressões, é a composição alternativa.
Exemplos:

In [ ]:
```
2 if 1==1 else 3
```

In [ ]:
```
2 if 1==2 else 3 if 1==1 else 4
```

Reforça-se que os operadores de atribuição (`=`) e de comparação (`==`) são distintos, e têm papéis muito diferentes, pelo que não devem ser confundidos.

Instruções de leitura e escrita (*input/output*)

Os mecanismos de leitura e escrita de dados (*input/output*) são fundamentais para a interação entre os nossos programas e os seus utilizadores. A leitura de dados é feita usando a função `input` , e a saída de dados, como já vimos, é feita usando a função `print` . Ambas as funções têm por valor uma cadeia de carateres.

Exemplos:

In [ ]:
```
input("valor? ")
```

In [ ]:
```
int(input("valor? "))+1
```

In [ ]:
```
int(input("valor? "))+1
```

In [ ]:
```
v=input("valor? ")
```

In [ ]:
```
print("valor =",v,", quadrado =",int(v)**2)
```

Como o valor da função input é uma string, como podemos avaliar o tipo de valores lidos? Recorre-se à função embutida *eval* ! A função *eval* recebe uma string e devolve o resultado da avaliação dessa string como sendo uma expressão.
Assim, podemos dispensar a conversão para o tipo pretendido, usando o casting, fazendo uso da função *eval*, que evitará erros na avaliação de strings, que representam valores numéricos, inseridas pelo utilizador.

In [ ]:
```
eval('3+8')
```

In [ ]:
```
eval('5 * 3.2 + 1')
```

In [ ]:
```
eval('a função eval')
```

Combinando a função de avaliação eval com a função input podemos obter o seguinte resultado:

In [ ]:
```
eval(input('insira um número: '))
```

Carateres de escape e seu significado:

\\ Barra ao contrário ()
\' Plica (')
\" Aspas (")
\a Toque de campainha
\b Retrocesso de um espaço
\f Salto de página
\n Salto de linha
\r "Return"
\t Tabulação horizontal
\v Tabulação vertical

In [ ]:
```
v=input("Insira um número\n-> ")
```

In [ ]:
```
print("valor =",v,"\tquadrado =",int(v)**2)
```

Pode-se formatar o resultado de um comando print de formas mais sofisticadas, usando o método format.

In [ ]:
```
"conseguir fazer {} dos {} exercícios do teste".format(3,5)
```

In [ ]:
```
"conseguir fazer "+ "3"+ " dos "+"5"+" exercícios do teste"
```

In [ ]:
```
help('FORMATTING')
```

Estrutura de controlo sequencial

O mais simples dos mecanismos de composição de instruções é a denominada *sequenciação*. Consiste, essencialmente, na execução consecutiva de várias instruções, na ordem especificada no programa.

Exemplos:

In [ ]:
```
x=5;y=x+1;media=(x+y)/2;print("A média é: ", media)
```

(;) termina a execução sequencial das várias atribuições, da esquerda para a direita. Apesar de suportada, e muito comum noutros ambientes, esta sintaxe é altamente desaconselhada em Python, por razões que promovem a legibilidade do código, bem como os princípios essenciais da programação estruturada. Deve-se, portanto, expressar a sequência das instruções com a mudança de linha.

In [ ]:

```
x=5
y=x+1
media=(x+y)/2
print("A média é: ", media)
```

Bibliografia

Caleiro.C., Ramos.J.(2016). Notebook 01 - Conceitos básicos da linguagem Python Dep. Matemática -IST.Lisboa:IST
https://www.w3schools.com/python/default.asp (https://www.w3schools.com/python/default.asp)
Vasconcelos, J. (2015).Python - Algoritmia e Programação Web. Lisboa:FCA
Martins. J. (2012). Programação em Python:Introdução à Programação utilizando Múltiplos Paradigmas.Lisboa:Departamento de Engenharia Informática-IST
https://algoritmosempython.com.br/cursos/programacao-python/operadores/ (https://algoritmosempython.com.br/cursos/programacao-python/operadores/)
https://www.devmedia.com.br/operadores-no-python/40693 (https://www.devmedia.com.br/operadores-no-python/40693)
https://towardsdatascience.com/whats-the-difference-between-is-and-in-python-dc26406c85ad (https://towardsdatascience.com/whats-the-difference-between-is-and-in-python-dc26406c85ad)
https://docs.python.org/pt-br/3/contents.html (https://docs.python.org/pt-br/3/contents.html)
https://docs.python.org/pt-br/3/library/time.html#time.struct_time (https://docs.python.org/pt-br/3/library/time.html#time.struct_time)
Martins. M.(2021), Ficha de Trabalho 13 -UFCD 0809 Programação em C/C++ - fundamentos.Covilhã:AEFHP