python_intro_01

December 5, 2021

1 Introdução ao Python

Iremos usar este notebook para explorar os principais elementos do Python. Como seria de esperar, nem todos operadores e tipos de dados serão abrangidos nesta introdução, por essa razão será sempre útil ter à mão a documentação oficial do Python.

Nota: Notebooks jupyter não são consideradas ferramentas de programação, são antes ferramentas usadas na exploração preliminar de dados e são especialmente úteis em contextos de formação, como é o nosso caso.

1.1 Variáveis

Em Python, as variáveis são etiquetas com nomes que apontam para valores(objectos). A contrário de outras linguagens de programação, em Python, a declaração de uma variável e atribuição de um valor é feita num único passo usando o operador `=`. Ao longo do tempo de vida de um programa ou script a mesma variavel pode mudar de valor e ate mudar de tipo de valores.

[ ]:
```
a = 10
b = 2
nome = 'Artur'

print(a, b, nome)

# Mudar valor da variavel
nome  = 'Lancelot'

print(a, b, nome)
```

```
10 2 Artur
10 2 Lancelot
```

> **Nota - Comentarios** Se quisermos adicionar comentarios no nosso codigo python, pratica muito comum e desejavel, fazemos com recurso ao caracter `#`. Em Python, na mesma linha, tudo o que vier a direita de um # e ignorado

As regras fundamentais para nomear as variáveis são:

1. O nome de uma variável deve começar com uma letra ou um underscore
2. Os nomes de variáveis só devem ter caracteres alfanuméricos (a-z, A-Z, 0-9) e underscores.

Para além disso, são recomendadas as seguinte regras na escolha de nomes de variáveis:

1. Um bom nome de uma variável tem um significado explícito - tem de ser possível adivinhar o objectivo da mesma apartir do seu nome.
2. Um bom nome de uma variável deve ser curto para evitar a poluição visual

Por exemplo, compare:

```
a = b * c
```

Com:

```
area_rectangulo = altura * largura
```

Uma variável para existir, terá de ser declarada anteriormente com um valor. Se isso não for feito, ao tentarmos usar uma variável que não foi iniciada, vamos obter um erro.

[ ]:
```
print(z)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_24887/557461111.py in <module>
----> 1 print(z)

NameError: name 'z' is not defined
```

Quando precisamos de criar uma variável "vazia" usamos a palavra reservada `None`.

[ ]:
```
z = None
print(z)
```

```
None
```

A gestão de memória no Python e automática, quando um determinado valor/objecto deixa de ser usado, o gestor de memória encarrega-se de o eliminar da memória para libertar espaco para outras coisas.

1.2 Tipos de dados

Em python existem os seguintes tipos de dados:

- Números (inteiros, float e complexo)
- Textos (uma cadeia de caracteres)
- Boleanos (valores logicos: `True` e `False`)

Cada data type define um domínio (i.e., conjunto aceitável de valores) e operadores que podem trabalhar com eles.

1.2.1 Números

Tipos de números Os números em python podem ser inteiros, float ou complexos

Exemplos de numero inteiros:

[ ]:
```
n1 = 0
n2 = 24
n3 = -50
n4 =  34453536363
```

Exemplos de numeros float (reais):

[ ]:
```
f1 = 0.1
f2 = -1.212
f3 = 1231212312.0
f4 = -1214456.12123
```

Podemos ver o tipo de numero/dado usando a funcao type()

[ ]:
```
print(type(n1))
print(type(f1))
```

```
<class 'int'>
<class 'float'>
```

Exercisio: Analise e corra o seguinte codigo:

[ ]:
```
number = 50
number = -22
number = 1.2345
```

Consegue adivinhar qual o valor final da variável `number`? Escreva em baixo o código que confirma a sua resposta.

[ ]:
```
# Escreva o codigo abaixo desta linha
```

Operadores aritméticos O Python trás nativamente os operadores aritméticos mais comuns, como sejam:

nome	Operador
Adicao	`+`
Subtraccao	`-`
Multiplicação	`*`
Divisão	`/`
Quoficiente da divisão inteira	`//`
Resto da divisão inteira	`%`
Potência	`**`

[ ]:
```
n1 = 10
n2 = 3
```

```
print(n1 + n2)
print(n1 * n2)
print(n1 / n2)
print(n1 // n2)
print(n1 % n2)
print(n1 ** n2)
```

```
13
30
3.3333333333333335
3
1
1000
```

Com a excepção dos operadores de quociente e resto, todos os outros operadores podem ser usados com outros tipos de números.

Paralelamente, através da instalação de bibliotecas especializadas, é possível adicionar outros tipos de dados numéricos e respectivos operadores. Veremos como carregar outras bibliotecas mais à frente no curso.

1.2.2 Strings (text)

Como representar Strings As strings, habitualmente traduzidas como cadeia de caracteres ou texto, servem para guardar caracteres individuais, palavras, frases, textos. São objectos imutáveis.

Representam-se de três formas possíveis:

- Pelicas
- Aspas
- Aspas tripas

[ ]:
```
s1 = 'Hello, there! I am a string defined with single quotes. Therefore, I have␣
 ↪no problems representing double quotes (")!'
s2 = "Hello, there! I'm a string defined with double quotes. Therefore, I have␣
 ↪no problems representing single quotes (')!"
s3 = """
Hello, there! I'm a string defined with triple quotes.
Not only I don't have problems representing double quotes (") or single quotes␣
 ↪(')
but I am also multi line by nature. """
```

A escolha entre pelicas ou aspas é uma questão de estilo ou escolha pessoal. É no entanto conveniente ser-se coerente com essa escolha. Pessoalmente, prefiro o uso das pelicas por serem mais rápidas de escrever, mas tornam a escrita em inglês mais complicada.

O uso das asplas triplas guarda-se geralmente para as doc-strings, onde descrevemos as funcionalidade de uma função.

Se numa string representada por meio de pelicas, quisermos representar uma pelica, podemos usar o caractere de escape . Este caractere também é necessário de usar para caminhos de ficheiros em

windows.

```
[ ]: s5 = 'Hello, there! I'm a string defined with single quotes.'
print(s5)
```

```
  File "/tmp/ipykernel_24887/148993037.py", line 1
    s5 = 'Hello, there! I'm a string defined with single quotes.'
                                                               ^
SyntaxError: unterminated string literal (detected at line 1)
```

```
[ ]: s5 = 'Hello, there! I\'m a string defined with single quote with a escape␣
 ↪caracter.'
print(s5)
```

```
Hello, there! I'm a string defined with single quote with a escape caracter.
```

```
[ ]: path = 'c:\users\arthur\deskop\my_file.py'
print(path)
```

```
  File "/tmp/ipykernel_24887/4105865062.py", line 1
    path = 'c:\users\arthur\deskop\my_file.py'
                                             ^
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in␣
 ↪position 2-3: truncated \uXXXX escape
```

```
[ ]: path = 'c:\\users\\arthur\\deskop\\my_file.py'
print(path)
```

```
c:\users\arthur\deskop\my_file.py
```

Operadores sobre strings As string são indexadas por inteiros com começo em zero, e podem ser acedidas com base nesse índice.

```
[ ]: nome = 'Holy grail'

print(nome[0]) # indice 0 Para obter o primeiro caracter
print(nome[-1]) # indice -1 para obter o último caracter
print(nome[5:]) # Para obter os primeiros X caracteres
print(nome[:-4]) # Para obter os ultimos X caracteres
print(nome[2:6]) # Para obter os caracteres dentro de um intervalo. Atenção que␣
 ↪o último caracter é excluído.
```

```
H
l
grail
```

```
Holy g
ly g
```

Algumas operações aritméticas também funcionam com strings. Por exemplo, a concatenação de duas strings pode ser feita com o operador `+`

[ ]:
```
s1 = 'Olá, o meu nome é '
s2 = 'Zeferino'

s3 = s1 + s2

print(s3)
```

```
Olá, o meu nome é Zeferino
```

Também podemos efectuar a multiplicação de uma string por um inteiro.

[ ]:
```
print("Olá! " * 3 + 'Zeferino.')
```

```
Olá! Olá! Olá! Zeferino.
```

Existe uma longa lista de métodos (funções intrínsecas de um objecto) para strings que podem ser consultadas aqui. Por exemplo, se quisermos converter uma string para maiúsculas podemos usar o método `upper`, mas também podemos usar o método `lower` para tornar todos os caracteres em minúsculas.

[ ]:
```
nome = 'Holy grail'

print(nome.upper())
print(nome.lower())
print(nome.title())
```

```
HOLY GRAIL
holy grail
Holy Grail
```

Podemos determinar o comprimento de uma string através da função `len()`.

[ ]:
```
s1 = 'paralelipípedo'
print(len(s1))
```

```
14
```

f-strings são string parametrizáveis. Quer isso dizer que podemos inserir valores em partes específicas do texto.

[ ]:
```
nome = 'Pedro'
altura = 1.78

print(f"O {nome} tem {altura} m de altura.")
```

```
nome = 'Joaquim'
altura = 1.96'

print(f"O {nome} tem {altura} m de altura.")
```

```
  File "/tmp/ipykernel_24887/4083152117.py", line 7
    altura = 1.96'
                  ^
SyntaxError: unterminated string literal (detected at line 7)
```

Uma outra forma de inserir valores dentro de strings é usar o método `format()`. Neste exemplo, ainda decidimos formatar o número com apenas 2 casas décimais.

[ ]:
```
nome = 'Pedro'
altura = 1.785

s = "O {} tem {:.2f} m de altura.".format(nome, altura)
print(s)
```

```
O Pedro tem 1.78 m de altura.
```

1.2.3 Boleanos

Como definir valores boleanos Apenas existem dois valores boleanos diferentes: `True` e `False`.

Em programação é comum termos de comparar coisas, é maior, é menor, é igual, etc...

A comparação segundo a order alfabética ou numérica é feita como em matemática.

- `a > b` - a é maior b
- `a >= b` - a é maior ou igual a b
- `a < b` - a é menor que b
- `a <= b` - a é menor ou igual a b
- `a == b` - a é igual a b
- `a != b` - a é different de b

Exercício Teste a comparação entre alguns números e textos

[ ]:
```
# Colocar o código abaixo desta linha
```

Operadores com boleanos Os operadores boleanos são o and, or e o not, entre outros.

[ ]:
```
verdadeiro = True
falso = False

print(verdadeiro and falso)
print(verdadeiro or falso)
```

```
print(not verdadeiro)
```

```
False
True
False
```

1.3 Estruturas de dados

O que são estruturas de dados?

Em programação, estruturas de dados são uma forma de organizar, gerir e guardar dados que permitam um acesso e modificação eficientes. Para além disso, tal como os tipos de dados, cada tipo de estrutura de dados tem operadores específicos para os manipular.

Em Python, as estruturas de dados mais comuns são as seguintes:

1. Listas
2. Dicionários
3. Tuplos
4. Set

1.3.1 Listas

As listas são conjuntos de objectos guardados de forma indexada. As listas definem-se usando parênteses rectos e virgulas para separar os elementos.

[ ]:
```
nomes = ["Pedro", "Joaquim", "Gonçalo", "Henrique"]
```

As listas não têm necessariamente de ser compostas por elementos com tipos de dados iguais.

[ ]:
```
numeros = [1, 2, 'III', 'quatro', 5.0]
```

Tal como vimos nas strings, os elementos de uma lista pode ser acedidos através do índice de base zero.

[ ]:
```
nomes = ["Pedro", "Joaquim", "Gonçalo", "Henrique"]

print(nomes[0])
print(nomes[1])
print(nomes[-1])
print(nomes[-2])
print(nomes[-3])
```

```
Pedro
Joaquim
Henrique
Gonçalo
Joaquim
```

Também como as strings, podemos obter um subset de uma lista usando a seguinte forma:

[ ]:
```
print(nomes[1:3])
```

```
['Joaquim', 'Gonçalo']
```

Ao contrário das strings, as listas são alteráveis. Podemos substituir valores em determinado índice ou adicionar novos elementos à lista.

[ ]:
```
print(nomes)

nomes[0] = 'O grande ' + nomes[0]
nomes[1] = 'Joana'
nomes.append('Ricardo')

print(nomes)
```

```
['Pedro', 'Joaquim', 'Gonçalo', 'Henrique']
['O grande Pedro', 'Joana', 'Gonçalo', 'Henrique', 'Ricardo']
```

Também podemos remover elementos de uma lista usando o operador `del`. Assim como obter e remover o último elemento da lista usando o método pop().

[ ]:
```
nomes = ["Pedro", "Joaquim", "Gonçalo", "Henrique"]

del nomes[1]

print(nomes)

print(nomes.pop())

print(nomes)
```

```
['Pedro', 'Gonçalo', 'Henrique']
Henrique
['Pedro', 'Gonçalo']
```

Podemos concatenar duas listas.

[ ]:
```
nomes1 = ["Pedro", "Joaquim", "Gonçalo", "Henrique"]
nomes2 = ["Manuel", "Eduardo"]

nomes_totais = nomes1 + nomes2

print(nomes_totais)

#Alternativamente, podemos alterar uma lista extendendo-a com outra

nomes1.extend(nomes2)

print(nomes1)
```

```
['Pedro', 'Joaquim', 'Gonçalo', 'Henrique', 'Manuel', 'Eduardo']
['Pedro', 'Joaquim', 'Gonçalo', 'Henrique', 'Manuel', 'Eduardo']
```

Por fim, é importante referir que as listas podem guardar tudo o que seja representável em Python. Por isso, é possivel termos uma lista de listas, ou uma lista de dicionários.

[ ]:
```
list_inception = [[1, 2, 3], ["4", "5", "6"], ["sete", "eight", "neuf"]]

print(list_inception[0])
print(list_inception[0][1])
```

```
[1, 2, 3]
2
```

O conjunto de funções que permitem manipular listas é enorme, podem encontrar mais informação aqui.

1.3.2 Dicionários

Dicionários guardam valores e chaves para aceder a esses mesmos valores. As chaves têm de ser strings, mas os valores podem ser tudo o que é representável em Python. Ao contrário das lista, os elementos de um dicionário, não têm uma ordem específica.

Para definir um dicionário, usamos parentesis encaracolados, dois pontos para separar a chave do valor e virgulas para separar cada par chave-valor.

[ ]:
```
dados = {'nome':'John', 'apelido':'Cleese', 'nacionalidade':'Inglês',␣
 ↪'ano_nascimento':1961, 'ocupacao':['Escrito', 'Actor', 'Comediante']}

# Mais fácil de ler
dados = {
    'nome':'John',
    'apelido':'Cleese',
    'nacionalidade':'Inglês',
    'ano_nascimento':1961,
    'ocupacao':[
        'Escrito',
        'Actor',
        'Comediante'
        ]
}

print(dados)
print(f"Seu nome é {dados['nome']} {dados['apelido']}.")
print(f"É {dados['nacionalidade']} e nasceu em {dados['ano_nascimento']}.")
print(f"Tem {2021 - dados['ano_nascimento']} anos")
```

```
{'nome': 'John', 'apelido': 'Cleese', 'nacionalidade': 'Inglês',
'ano_nascimento': 1961, 'ocupacao': ['Escrito', 'Actor', 'Comediante']}
Seu nome é John Cleese.
```

```
É Inglês e nasceu em 1961.
Tem 60 anos
```

Para melhor visualizarmmos os dados podemos mudar de linha

[ ]:

[ ]: