2

July 23, 2021

# 1 Básicos da Programação em Python

## 1.1 Sobre o Python

Não é por acaso que a linguagem de programação Python tem se tornado cada vez mais popular. Além do seu uso na computação, que ganhou um novo *hype* por conta da Inteligência Artificial, a linguagem é uma ótima forma de introduzir uma pessoa à programação, de modo geral.

A linguagem apresenta diversas facilidades em sua sintaxe, que não somente contibuem para a facilidade da escrita, mas também contribuem para o entendimento e documentação do código.

Novamente, para não fugir do escopo do curso, não acredito que seja necessário listar as especificações técnicas da linguagem. Acredito que há somente alguns conceitos simples que são necessários para o uso da linguagem.

- O Python é uma linguagem interpretada.

Ou seja, o software gerado pelo código é o próprio código interpretado durante a execução

- O Python interpretará linha após linha, do início ao fim do script (chunk, para nós).

Para novatos em programação, isso deve ser fácil de aceitar. Mas pessoas com certa experiência podem questionar essa afirmação. Nesse caso, estou me referindo a forma que vamos utilizar a linguagem no curso e até onde iremos com as estruturas de programação da linguagem.

- O Python diferencia letras maíusculas de minúsculas.

Isso é chamado de "sensitive case". Tome cuidado ao reproduzir os exemplos e ao programar, de modo geral.

- O Python reconhece espaços vazios.

Isso é utilizado para definir o escopo dessa linha de código. Não se preocupe caso não tenha entendido, é mais simples do que aparenta ser. Isso basicamente é pra mostrar o "quão dentro" uma linha está de uma estrutura. Abordaremos isso mais tarde.

- Texto precedido por `#` não são interpretadas

Tudo que vier após um `#` em uma linha é chamado de comentário. Ele não é compilado e utilizaremos para comentar o código.

## 1.2 Definindo variáveis

Uma variável é uma forma de armazenar certo valor e reutilizá-lo. Se quiser fazer uma comparação com a Matemática, apenas enxergue ele como um $x_0$ não um $x$ somente. Ou seja, ele vai ter um

valor específico que pode mudar. Não necessáriamente ele seria "todos os valores ao mesmo tempo" como uma variável na matemática.

Por exemplo, vamos criar a variável `name` que vai armazenar meu nome.

[1]:
```
name = "Eduardo"
```

Agora, eu posso utilizar name em qualquer lugar do meu código. No caso, se eu mudar o valor de name, todo lugar onde name está terá seu valor trocado.

Por exemplo:

[2]:
```
name = "Eduardo"
name
```

[2]:
```
'Eduardo'
```

[3]:
```
name = "Adame"
name
```

[3]:
```
'Adame'
```

Para ficar mais claro, podemos ver algo como:

[4]:
```
name = "Eduardo"
name = "Adame"
name
```

[4]:
```
'Adame'
```

Como pôde ver no *chunk* acima, embora eu tenha atribuído o valor `"Eduardo"` ao `name` na primeira linha, o valor dele foi sobrescrito pela linha seguinte.

## 1.3 Tipos de Variáveis

Acredito que deve ter notado que colocamos o valor entre `" "`. Por quê?

Isso tem a ver com o tipo do valor. No Python, os tipos são dinamicamente interpretados através de sua sintaxe. E, nesse caso, nós inserimos uma `string`, que é utilizada para armazenar texto.

Vamos falar sobre os principais tipos de variáveis que já vêm com o Python e como defini-las.

### 1.3.1 Inteiros

Os inteiros são simplesmente definidos ao igualar uma variável a um número que pertença ao conjunto dos números inteiros, seja positivo ou negativo.

[5]:
```
my_integer = 2
my_integer
```

[5]:
```
2
```

### 1.3.2 Floats

Os floats, diferentemente dos inteiros, podem ser qualquer número real. Utilizamos ponto (`.`) para separar as casas decimais.

[6]:
```
my_float = 2.5
my_float
```

[6]:
```
2.5
```

### 1.3.3 Strings

Como dito anteriormente, strings são utilizados para textos. Basta envolver seu texto em aspas (`" "`) ou (`' '`). Para o Python, tanto faz se elas são simples ou duplas.

[7]:
```
my_string = "Textos são bem úteis, não acha?"
my_string
```

[7]:
```
'Textos são bem úteis, não acha?'
```

### 1.3.4 Booleanos

Esse tipo de valor é utilizado para condições. Eles podem ser "Falso" (`False`) ou "Verdadeiro" (`True`). Preste atenção no sensitive case.

[8]:
```
do_i_like_math = True
do_i_like_rlang = False
do_i_like_math, do_i_like_rlang
```

[8]:
```
(True, False)
```

### 1.3.5 Listas

Nós utilizamos listas para agrupar outros valores. Uma lista pode armazenar qualquer tipo e em qualquer quantidade. Inclusive podemos fazer listas de listas.

[9]:
```
names = ["Eduardo", "Marcos", "Ruan", "João"]
names
```

[9]:
```
['Eduardo', 'Marcos', 'Ruan', 'João']
```

[10]:
```
my_id = ["Eduardo", 18, 1.78]
my_id
```

[10]:
```
['Eduardo', 18, 1.78]
```

Para acessar um valor em específico na lista utilizamos `[]` com o índice desse valor. A lista começa a contagem em 0. Veremos algumas outras coisas necessárias para o uso de listas posteriormente.

[11]:
```
names[1] # "Eduardo" é o 0
```

[11]:
```
'Marcos'
```

[12]:
```
my_id[2]
```

[12]:
```
1.78
```

## 1.4 Operações

Os operadores e as operações para cada tipo são diferentes. Principalmente quando ocorrem entre dois tipos diferentes. Os principais para cada um deles são:

### 1.4.1 Aritmética

Com `int` e `float` é possível utilizar os operadores comuns da Matemática. Quando ocorre entre eles, o resultado será um `float`.

Os principais operadores são:

- `+`/`-` para adição/sutração.
- `*`/`//` para multiplicação/divisão.
- `**` para potenciação.
- `%` para módulo (resto).

Para alguns exemplos, como esse, não criarei variáveis, mas é possível criar para qualquer operação.

[13]:
```
5 + 2 + 4 * 10
```

[13]:
```
47
```

[14]:
```
2 ** (1/2) # Raíz de 2
```

[14]:
```
1.4142135623730951
```

Você também pode utilizar esses operadores em cima de uma variável. Por exemplo, você tem uma variável e quer acrescer 2 nela.

[15]:
```
my_num = 14
my_num = my_num + 2
my_num
```

[15]:
```
16
```

O exemplo acima é bem convincente e, certamente, bem útil. Mas há uma forma reduzida desse tipo de expressão.

[16]:
```
my_num = 14
my_num += 2
my_num
```

[16]:
```
16
```

Esse tipo de "operador" pode ser criado ao unir um desses operadores listados acima com o símbolo de =.

[17]:
```
my_num = 2
my_num *= 10
my_num
```

[17]: 20

### 1.4.2 Booleanas

As operações booleanas são expressões que retornam ou `True` ou `False`, são utilizadas para condições.

No Python, costumamos escrever por extenso a maioria das condições.

- `and` para operação "e"
- `or` para operação "ou"
- `not` para operação de negação (inverte o valor)
- `>`/`>=` maior/maior ou igual
- `<`/`<=` menor/menor ou igual
- `==` igual
- `in`confere está em uma lista

Exemplos:

[18]:
```
30 > 15
```

[18]: True

[19]:
```
30 < 15
```

[19]: False

[20]:
```
14 == 14
```

[20]: True

[21]:
```
30 < 15 and 14 == 14
```

[21]: False

[22]:
```
30 <= 15 or 14 == 14
```

[22]: True

[23]:
```
14==14 and not 30 < 15
```

[23]: True

[24]:
```
lst = [1,2]
2 in lst
```

[24]: True

[25]:
```
True and False
```

[25]: False

[26]:
```
False or not False
```

[26]: True

### 1.4.3 Strings

Podemos formatar strings e fazer algumas operações com elas

- `+` para concatenação
- `*` para concatenação repetidas vezes

Exemplos:

[27]:
```
'Eduardo ' + 'Adame'
```

[27]: 'Eduardo Adame'

[28]:
```
'du'*2
```

[28]: 'dudu'

**Strings formatadas** As strings formatadas são um modo de inserir variáveis em strings. Adicionamos um `f` antes das áspas e colocamos a variável entre chaves `{}`.

Exemplo:

[29]:
```
nome = 'Eduardo'
sobrenome = 'Adame'
idade = 18
f'Meu nome é {nome} {sobrenome} e tenho {idade} anos'
```

[29]: 'Meu nome é Eduardo Adame e tenho 18 anos'

## 1.5 Conversão de Tipos

Basta utilizar a função (veremos mais tarde funções em específico) que tem o nome do tipo.

- `int()` para converter para `inteiro`
- `str()` para converter para `string`
- `bool()` para converter para `booleano`
- `float()` para converter para `float`

[30]:
```
int('10')
```

[30]: 10

[31]:
```
int(20.0)
```

[31]: 20

[32]:
```
float('12.23')
```

[32]: 12.23

[33]:
```
str(30.2)
```

[33]: '30.2'

[34]:
```
bool(0)
```

[34]: False

[35]:
```
bool(1) #ou qualquer outro número diferente de 0
```

[35]: True

[36]:
```
bool('')
```

[36]: False

[37]:
```
bool('a') #ou qualquer string não vazia
```

[37]: True

### 1.6 Recebendo entrada do usuário

Para isso, utilizamos a função `input()`, que recebe uma `string` como parâmetro para a mensagem impressa, e retorna a entrada como uma `string` (devemos converter, se necessário). Para imprimir uma mensagem utilizamos `print()` que imprime qualquer coisa que for passada, costuma-se utilizar as operações vistas anteriormente.

Exemplo:

[38]:
```
number = input('Insira um número: ')
number * 2
```

```
Insira um número:  10
```

[38]: '1010'

Por que ao invés de 20 recebemos '1010'? Porque 10 foi recebido como '10', ou seja, uma `string`.

Logo, o exemplo correto seria:

[39]:
```
number = float(input('Insira um número: ')) #float para aceitar qualquer número
number * 2
```

```
Insira um número:  10
```

[39]:
```
20.0
```

## 1.7 Condições

Em Python, como na maioria das linguagens de programação podemos utilizar estruturas condicionais para interpretar certas linhas de código somente se certa condição for atendida.

No caso, aqui faremos o uso das palavras-chave `if`, `else` e `elif`.

A partir de agora, veremos que os espaços em branco são importantes para denotar se certa linha está dentro de uma estrutura.

### 1.7.1 If

[40]:
```
word = "Renato"
if 2 > 1:
    word = "Eduardo"
word
```

[40]:
```
'Eduardo'
```

Como visto acima, o if verificou se o booleano (resultado da operação) era verdadeiro. Como era, executou a linha que definia o novo valor para `word`. Caso fosse falso teríamos o seguinte resultado:

[41]:
```
word = "Renato"
if 1 > 2:
    word = "Eduardo"
word
```

[41]:
```
'Renato'
```

Agora, se quisermos fazer algo se tal condição for real (como vimos acima) mas, se caso contrário, fazer outra ação? Utilizaremos o `else`.

[42]:
```
word = "Renato"
if 1>2:
    word = "Eduardo"
else:
    word = "Flávio"
word
```

[42]:
```
'Flávio'
```

A estrutura acima é muito simples e frequentemente utilizada. Ela executa o primeiro bloco caso a condição for verdadeira, caso contrário, executa a segunda.

Mas se quisermos criar condições intermediárias (ou específicas) utilizamos o `elif`. Ele significa algo como: "Se o anterior for falso, verifica se essa outra condição é verdadeira antes de ir pro `else`".

[43]:
```
lst = [0,1,2]

if 3 in lst:
    print("3 está!") # Em notebooks o print é desnecessário, mas esse é um␣
 ↪exemplo geral.
elif 2 in lst:
    print("2 está!")
else:
    print("3 e 2 não estão!")
```

```
2 está!
```

## 1.8 Métodos

Métodos são funções chamadas a partir de um objeto de um tipo específico. Novamente, não precisa se preocupar caso não tenha entendido, é mais fácil do que aparenta.

Esses métodos são acessados através de `.`. E cada tipo terá os seus, ou seja, `str` tem seus próprios métodos, assim como `list` tem seus próprios.

Não é viável tratar cada um deles aqui, mas é interessante que sempre busque pela documentação quando sentir necessidade. Por exemplo, aqui estão todos os métodos de `list`.

Alguns exemplos:

[44]:
```
# Todas as letras maiúsculas em uma string
word = "eDuArdO"
word = word.upper()
word
```

[44]: `'EDUARDO'`

[45]:
```
# Todas as letras minúsculas em uma string
word = "eDuArdO"
word = word.lower()
word
```

[45]: `'eduardo'`

[46]:
```
# Somente a primeira letra maiúscula em uma string
word = "eDuArdO adaME"
word = word.title()
word
```

[46]: `'Eduardo Adame'`

Esses métodos de strings são muito úteis em condições. Note que `'eduardo'` é diferente de `'Eduardo'`. Outros exemplos são mais úteis para limpeza de dados e/ou correção, como o abaixo.

[47]:
```
# Substitui todas ocorrências de uma substring
phrase = "Penso, logo existo"
phrase = phrase.replace('logo', 'portanto')
phrase
```

[47]:
```
'Penso, portanto existo'
```

Para listas, os seus métodos são suas principais "operações".

[48]:
```
lst = [0,2,4]
# Adiciona um item a uma lista
lst.append(6)
lst # Note que ele altera a variável diretamente
```

[48]:
```
[0, 2, 4, 6]
```

[49]:
```
if 2 in lst: # Forma de evitar erros
    lst.remove(2) # Remove um item da lista
lst
```

[49]:
```
[0, 4, 6]
```

[50]:
```
# Remover um item pelo seu índice
lst.pop(2)
lst
```

[50]:
```
[0, 4]
```

[51]:
```
# Extende a lista com outra (semelhante a uma soma)
lst.extend([2,6])
lst
```

[51]:
```
[0, 4, 2, 6]
```

[52]:
```
# Limpa a lista
lst.clear()
lst
```

[52]:
```
[]
```

### 1.9 Estruturas de Repetição

Esse tipo de estrutura, como diz o seu nome, é utilizada para repetir certo bloco de código. No caso, há duas formas de criar esse tipo de *loop*.

- `while`: Repete o bloco enquanto certa condição verdadeira
- `for`: Repete o bloco para cada item em uma lista

Nesse ponto do curso gostaria de fazer uma menção sobre uma decisão que tomei quanto ao método de ensino. Utilizar listas não é a unica forma de agrupar elementos, por exemplo, existem os `sets` e as `tuples`. Contudo, todos eles são iteráveis. Ou seja, podemos navegar por seus elementos. Portanto, tudo que se refere a iterabilidade da lista, também serve para esses outros tipos (que pretendo falar posteriomente).

Exemplos:

[61]:
```
entry = int(input('Digite um inteiro: '))
while entry != 7: # Verifica se a entrada é igual a 7.
    entry = int(input('Digite um inteiro: '))
print("Loop encerrado!")
```

```
Digite um inteiro:  2
Digite um inteiro:  3
Digite um inteiro:  4
Digite um inteiro:  8
Digite um inteiro:  7

Loop encerrado!
```

Tome cuidado com loops infinitos! A estrutura `while` é bem propícia a isso.

[54]:
```
lst = [0,2,4,6]
for item in lst: # Para cada item da lista
    print(item ** 2) # Imprima o dobro do item
```

```
0
4
16
36
```

É possível criar coisas muito poderozas com o for, inclusive na visualização de dados (em texto).

[55]:
```
lst = [0,2,3,5,6]
for item in lst: # Para cada item da lista
    sq = item **2
    if item % 2 == 0: # Checa se o número é divisível por 2 (par)
        is_even = "par"
    else:
        is_even = "ímpar"
    print(f'O número {item} é {is_even}, e seu quadrado é {sq}') # Imprima o␣
 ↪dobro do item
```

```
O número 0 é par, e seu quadrado é 0
O número 2 é par, e seu quadrado é 4
O número 3 é ímpar, e seu quadrado é 9
O número 5 é ímpar, e seu quadrado é 25
O número 6 é par, e seu quadrado é 36
```

## 1.10 Funções

Funções são blocos de código que podem ser reutilizados. Elas podem ser definidas de duas formas:

- Utilizando `def`
- Utilizando `lambda` (chamadas funções anônimas)

O uso mais comum, entretanto, é utilizando o `def`.

Exemplo:

[64]:
```
def square(x):
    return x**2

square(89)
```

[64]:
```
7921
```

Note que, ao utilizar `def`, nós criamos um bloco. Nós devemos nomear uma função e colocar entre parênteses seus parâmetros (ou deixá-lo vazio). É possível definir valores padrões para certos parâmetros. As variáveis criadas dentro do bloco só existirão dentro dele (inclusive sobrescrevendo temporariamente uma variável global).

O termo return se refere a linha que define o valor retornado por essa função. Quando utilizamos notebooks, o Jupyter imprime o retorno da última linha, por isso não utilizamos print. Mas, utilizando esse exemplo, se estivéssemos no desenvolvimento regular, seria algo como `print(square(89))`.

Alguns exemplos com `def`:

[65]:
```
def count_even(lst):
    count = 0
    for item in lst:
        if item % 2 == 0:
            count += 1
    return count

count_even([1,2,4,5,8])
```

[65]:
```
3
```

[69]:
```
# Como utilizar variáveis globais

breads = 100
currency = 0
def sell():
    global breads, currency # Crie as variáveis com a palavra-chave global
    breads -=1
    currency += .5

sell()
```

```
breads,currency
```

[69]: (99, 0.5)

[89]:
```
def print_id(name, age, course = ''):
    result = f' Nome: {name} \n Idade: {age}' # \n para quebrar a linha
    if course:
        result += f' \n Curso: {course}'
    return print(result) # Necessário para \n funcionar

print_id('Eduardo', 18)
```

```
 Nome: Eduardo
 Idade: 18
```

[87]:
```
print_id('Eduardo', 18, 'Ciência de Dados')
```

```
 Nome: Eduardo
 Idade: 18
 Curso: Ciência de Dados
```

Para funções de linha única, o `lambda` pode ser uma opção melhor. Ou quando uma função recebe outra como parâmetro. Muitos cursos não abordam esse tipo de função, mas acredito que passar rapidamente por ela pode ser bom.

Anteriormente definimos a função `square()`, podemos definir uma função que faz o mesmo trabalho da seguinte forma:

[88]:
```
sqr_anon =  lambda x: x**2

sqr_anon(89) # É invocada da mesma forma.
```

[88]: 7921

Nesse caso, a criamos como se fosse um valor atribuído. O que está entre `lambda` e `:` são os parâmetros e o que está depois dos `:` é o retorno. Nós perderemos o poder de criar o parâmetro opcional como em `def`, mas podemos fazer algo parecido com `print_id()`.

[90]:
```
id_anon = lambda name, age: f' Nome: {name} \n Idade: {age}' # Decidi deixar o␣
 ↪print de fora.
print(id_anon('Eduardo', 18))
```

```
 Nome: Eduardo
 Idade: 18
```

## 1.11 Exercícios

Agora você sabe mais que o suficiente para utilizar o `sympy`. Para verificar que absorveu o aprendido aqui, tente resolver os seguintes exercícios:

1. Crie uma função que recebe dois números (floats) e retorna o menor elevado pelo maior.
2. Crie uma função que recebe duas listas, uma com o ponto inicial e outra com o ponto final `[x,y]`, e calcule a distância entre eles.
3. Crie uma função que recebe números (floats) de quantidade indefinida (pesquise sobre `*args`) e retorne a soma deles.
4. Crie um loop que faz o mesmo que a função acima, contudo, ele deverá encontrar o valor total quando o usuário inserir `'s'`.
5. Crie uma função que calcula as raízes reais de uma equação do segundo grau a partir dos coeficientes `a`, `b`, `c`.