Apostila
Curso de *Python 3*

Escola Padre Constantino de Monte
Escola da Autoria - Tempo Integral
Curso Técnico em Informática Integrado ao Ensino Médio

Prof. coordenador Jhonathan Paulo Banczek

Nota

- Esta apostila foi elaborada para uso na disciplina de Lógica e Linguagem de Programação, do curso técnico em Informática Integrado ao Ensino Médio, da Escola Padre Constantino de Monte. Maracaju-MS.
- Os códigos foram testados em Python 3.7, no Windows 10 pro, 64bit.
- Todo material será disponibilizado no github: github.com/jhoonb/autoria
- Dicas são assinaladas pelo símbolo: ➢ **DICA.**

Sumário

Informações sobre a linguagem

- *Python* é uma linguagem de programação de ***alto nível***, ***multiparadigma***, ***interpretada*** e de ***propósito geral***.
- *Python* é uma linguagem de ***tipagem dinâmica*** e ***forte***.
- Python é ***multiplataforma.***
- Em *Python* os ***blocos de instrução*** são separados por ***indentação***.
- Duas grandes ***versões*** de *Python*: O *Python* 2 e o *Python* 3.
- *Python 3* foi lançado em 2008. *Python 2* perderá o suporte em 2020. *Python 3* não é retrocompatível.
- Usaremos *Python 3* em nossas aulas!
- Site oficial da linguagem (conteúdo em *inglês*): https://python.org
    - Documentação Oficial do *Python 3*: https://docs.python.org/3/
    - Biblioteca Padrão da linguagem (*Standard Library*) https://docs.python.org/3/library/index.html
    - Referência da Linguagem (*Language Reference*) https://docs.python.org/3/reference/index.html
    - Download do instalador: https://www.python.org/downloads/

Instalando Python 3 e o IDLE

Baixe o instalador do *Python 3*. Lembre de baixar a versão correta pro seu sistema operacional (x86 ou 64). Execute em seu computador, veja o passo-a-passo abaixo:

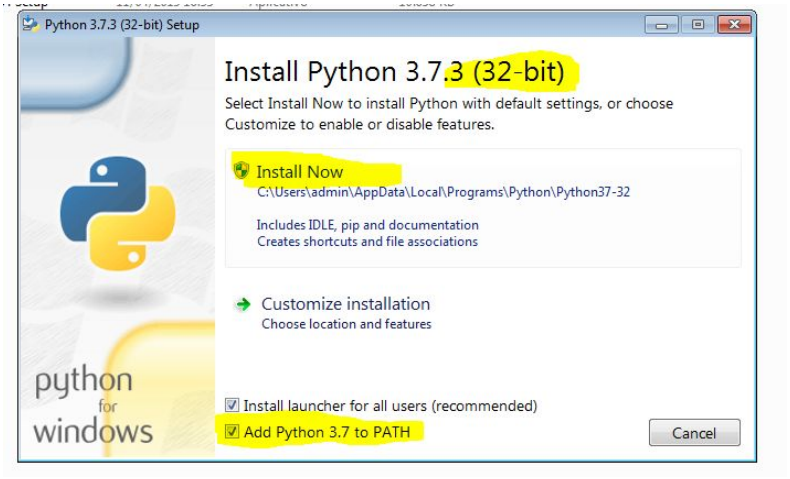


Imagem 1 (tela do instalador)

Marque a caixa "*Add Python to PATH*" e clique em *Install Now*.

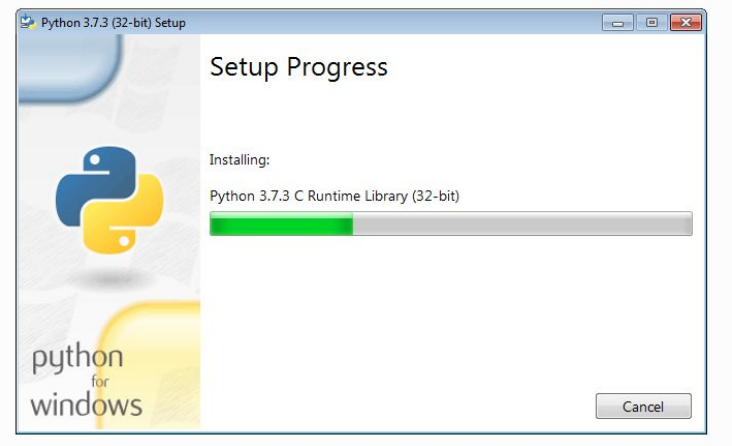


Imagem 2 (Espere o instalar concluir).

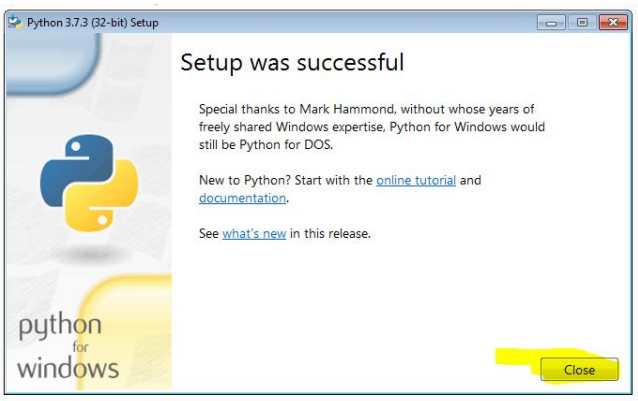


*Imagem 3 (*Clique em *close* para finalizar a instalação).

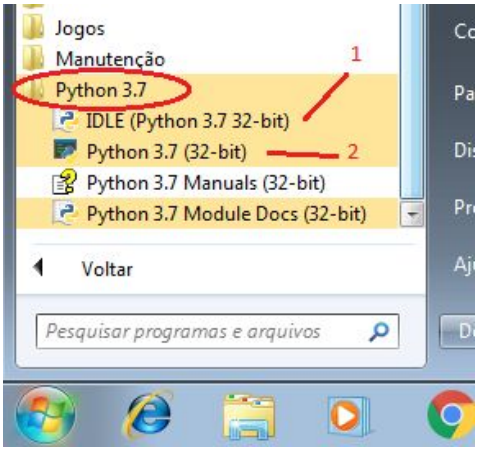


Imagem 4 (menu iniciar com o Python instalado)

No Menu Iniciar do Windows podemos ver a pasta do *Python 3*, em 1) temos o IDLE que é o **Ambiente de Desenvolvimento Integrado para Python.** Em 2) podemos acessar o Python pelo **interpretador na linha de comando do Windows**.

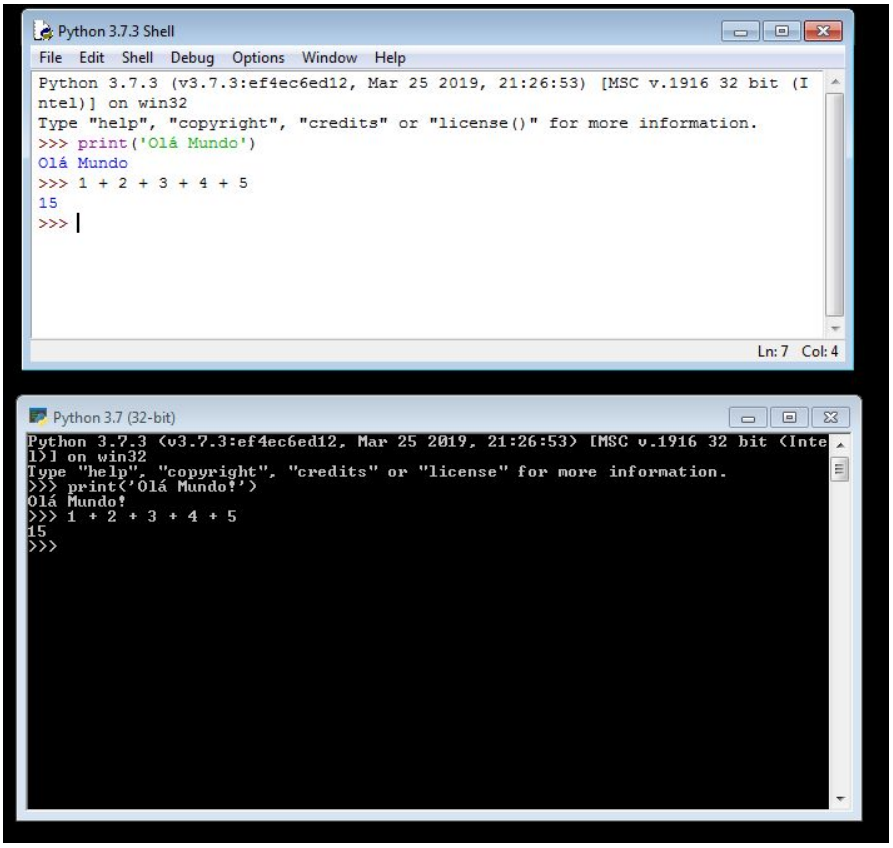


Imagem 5 (IDLE e o interpretador com o mesmo código executado)

Veja abaixo como criar um novo arquivo de código Python usando o próprio editor de código do IDLE:

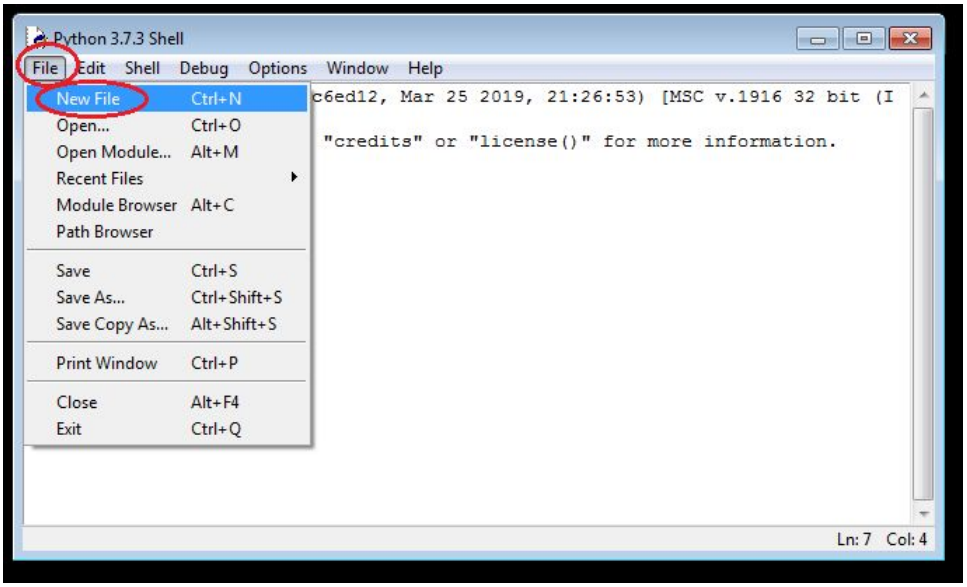


Imagem 6

Para criar um novo arquivo, clique em “File” (arquivo) e em “New File” (Novo Arquivo).

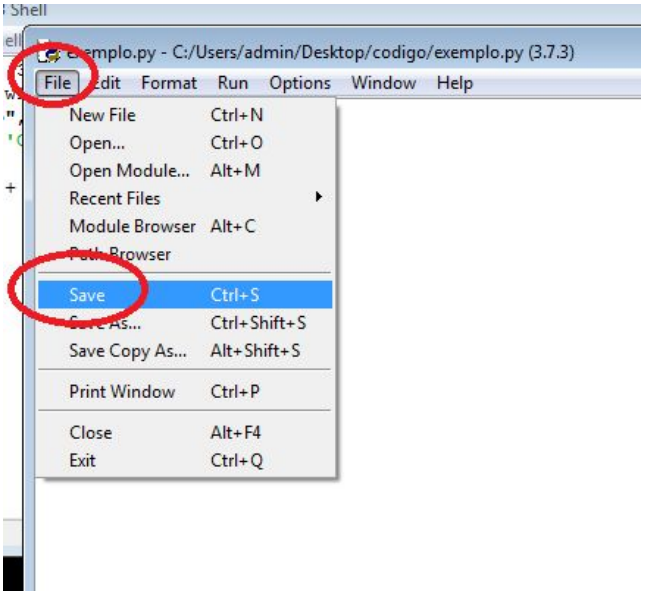


*Imagem 7 (*Clique em “File” e em “Save” (salvar) para salvar um arquivo)

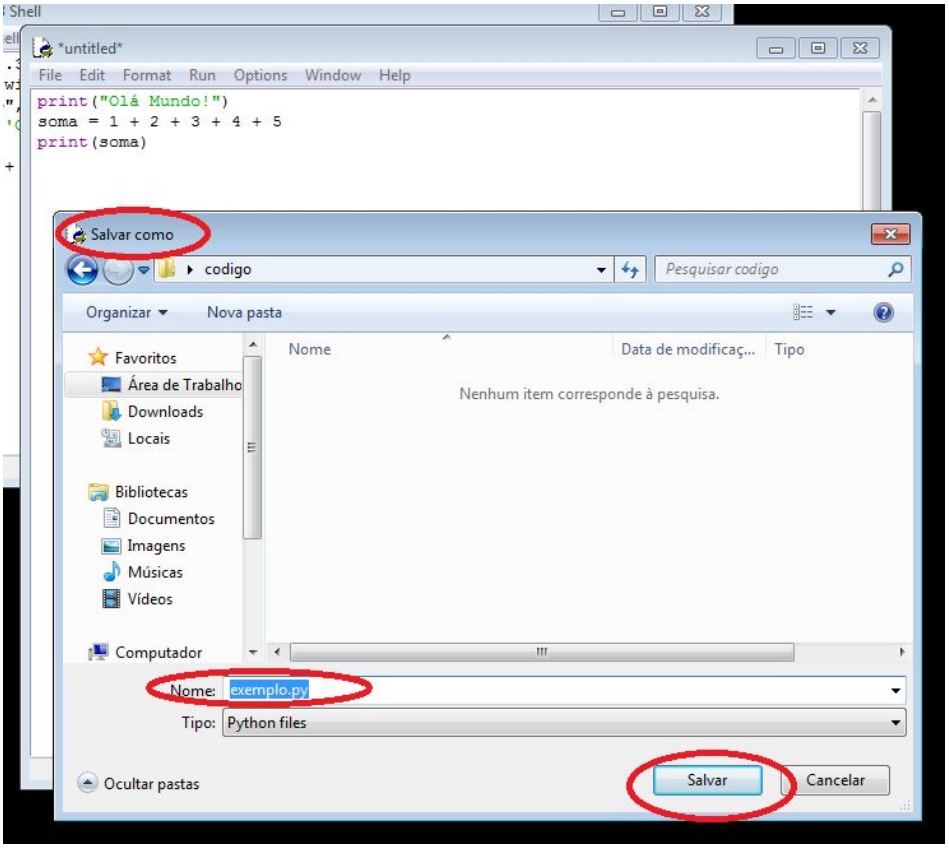


Imagem 8

Dê um nome para o arquivo e coloque a extensão de arquivo Python: .py, salve na pasta que deseja.

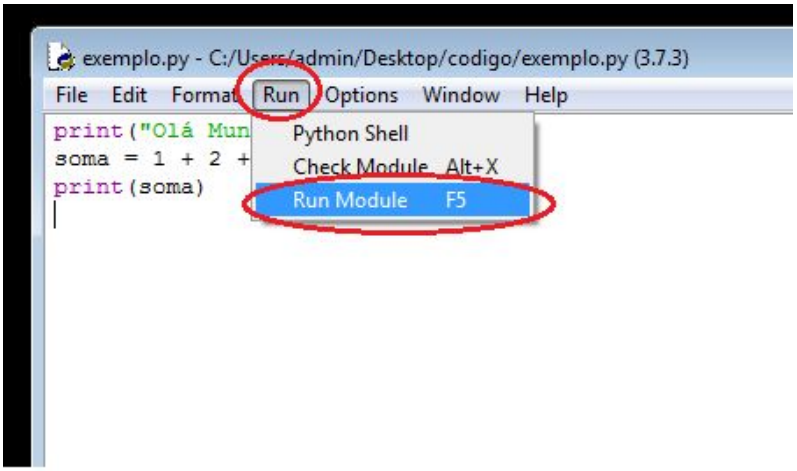


Imagem 9

Para executar um programa *Python* pelo **IDLE**, apenas crie um arquivo, salve e clique em "*Run*" e depois em "*Run Module*", ou utilize a tecla de atalho **F5**

Instalando o editor de texto *Sublime Text 3*

O Sublime Text é um editor multiplataforma para edição de textos, suporta dezenas de linguagens, como Python, Java, HTML, CSS, Javascript, Ruby, Perl, PHP, Lua, Go, C++, C, Pascal, etc. Possui inúmeros recursos como coloração da sintaxe, auto completar, compilação (*build*), temas, criação de janelas, minimap, etc. Conta também com um poderoso recurso de extensão via plugins.

Site oficial do programa: https://www.sublimetext.com/. É um editor pago, porém podemos testar de maneira gratuita.

Vamos ver como proceder na instalação. Baixe o *Sublime* pelo link acima, no menu "*Download*". Siga os passos abaixo:

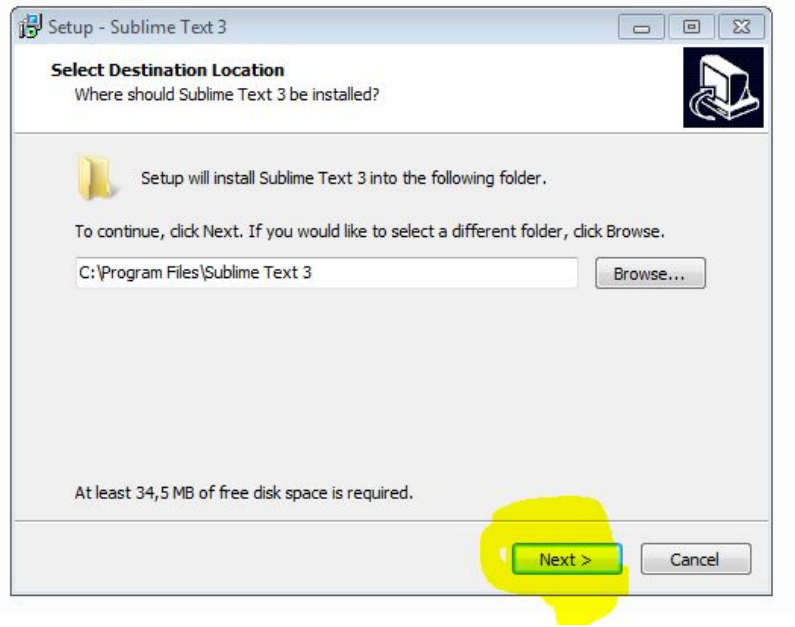


Imagem 10

No instalador clique em "*Next*" (próximo).

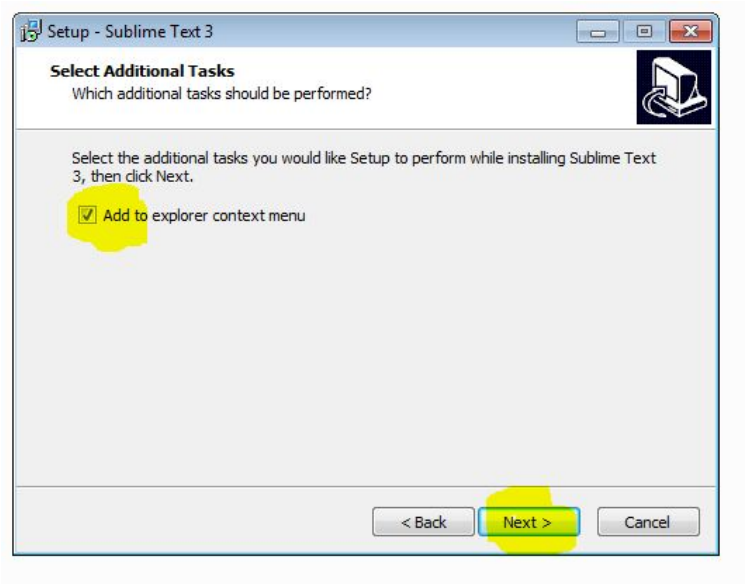


Imagem 11

Marque a opção "*Add to explorer context menu*". Clique em "Next" e depois, na próxima tela clique em "*Install*" (instalar). Aguarde a instalação, por fim clique em "*Finish"*.

Imagem 12

Para abrir o programa, no menu iniciar pesquise por "Sublime" e o programa aparecerá na lista. Ou procure o atalho na Área de Trabalho.

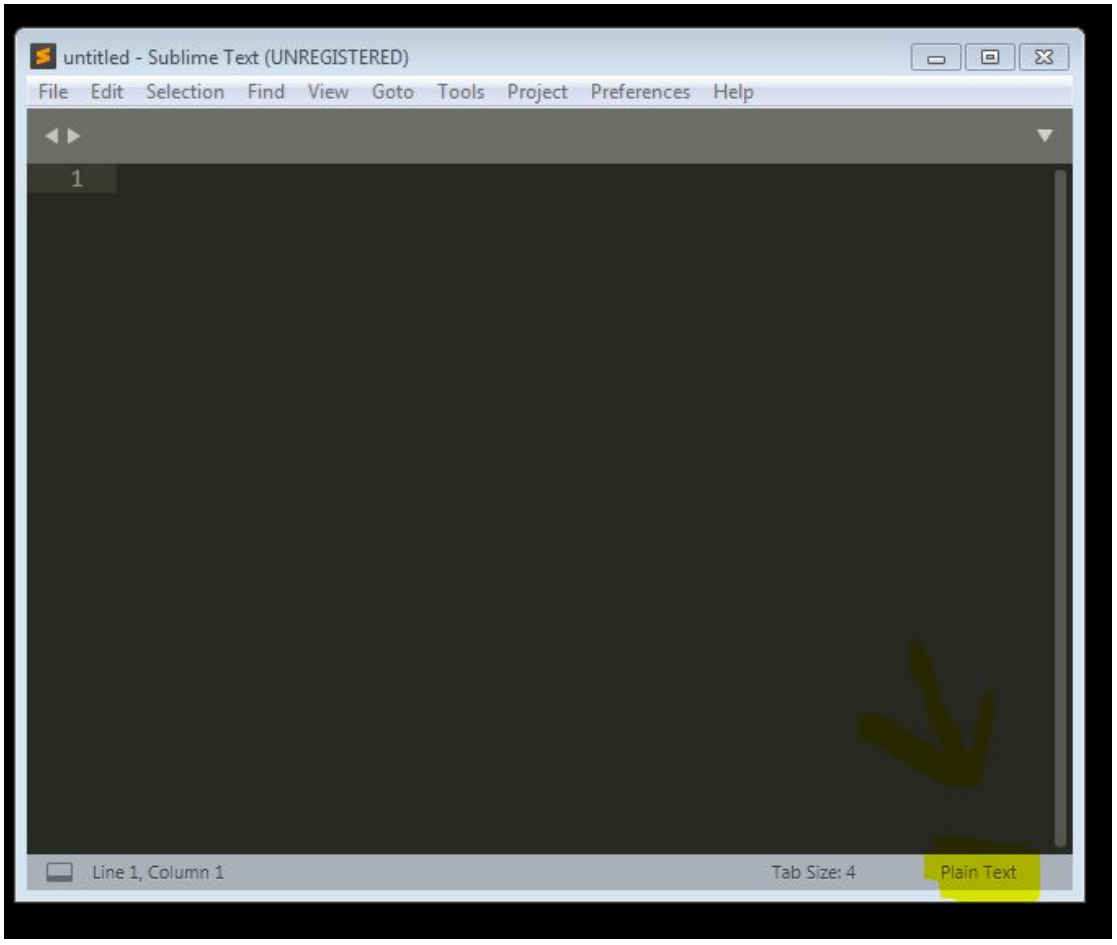


Imagem 13

Tela do programa, observe que do lado inferior direito existe um menu, onde podemos selecionar qual tipo de linguagem o editor deve se configurar. Clique no menu e escolha a sua linguagem de escolha (imagem 14).

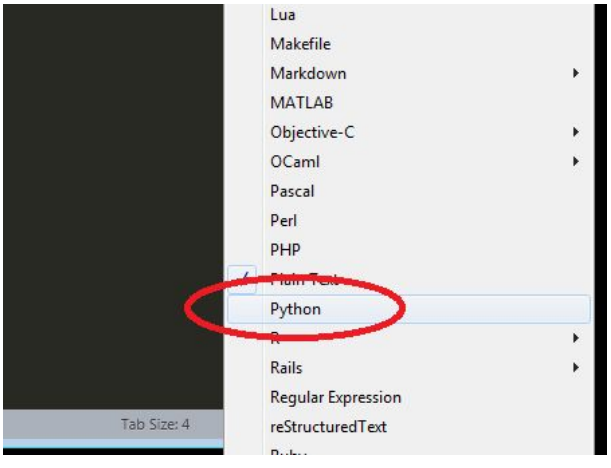


Imagem 14

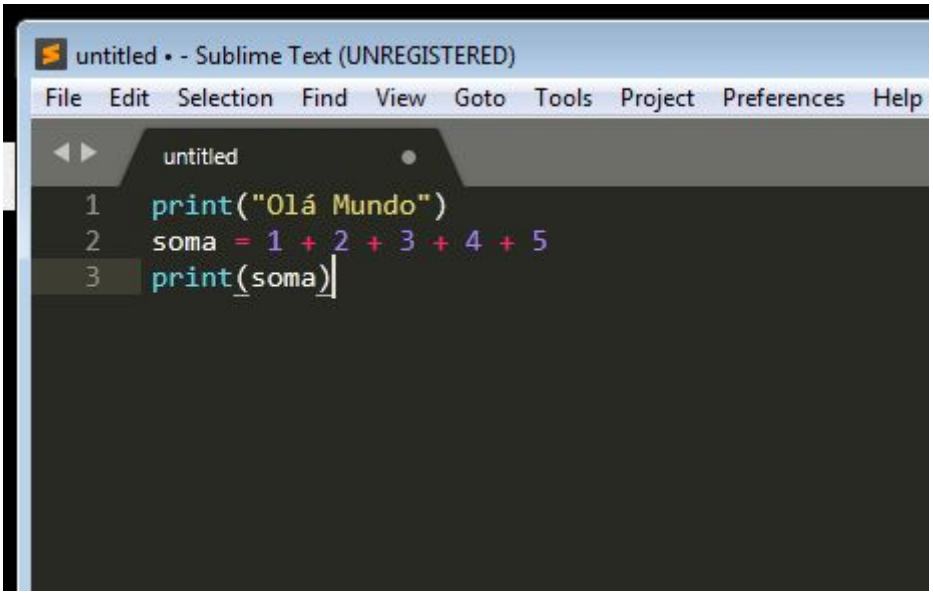


Imagem 15

Na imagem 15 vemos como o editor mudou a coloração para mostrar um código em

Python.

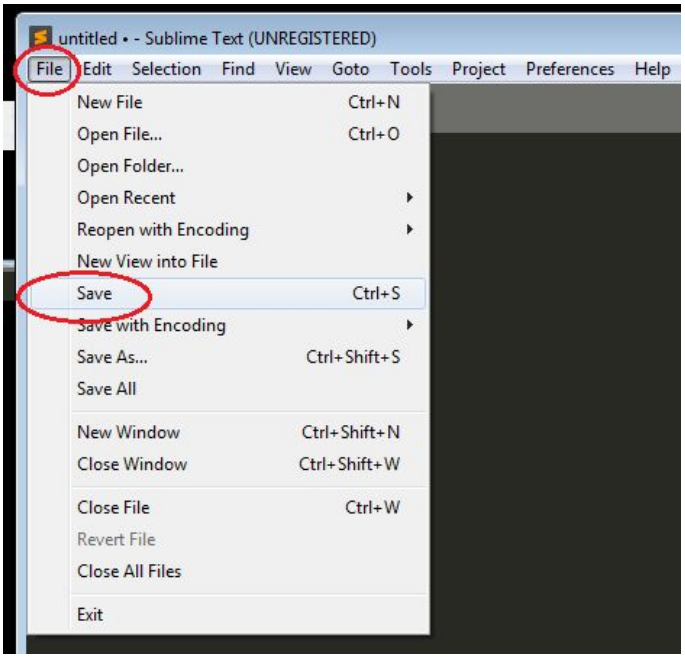


Imagem 16

Para salvar um arquivo, clicamos em “File”, depois em “Save”. Ou o atalho Control + S.

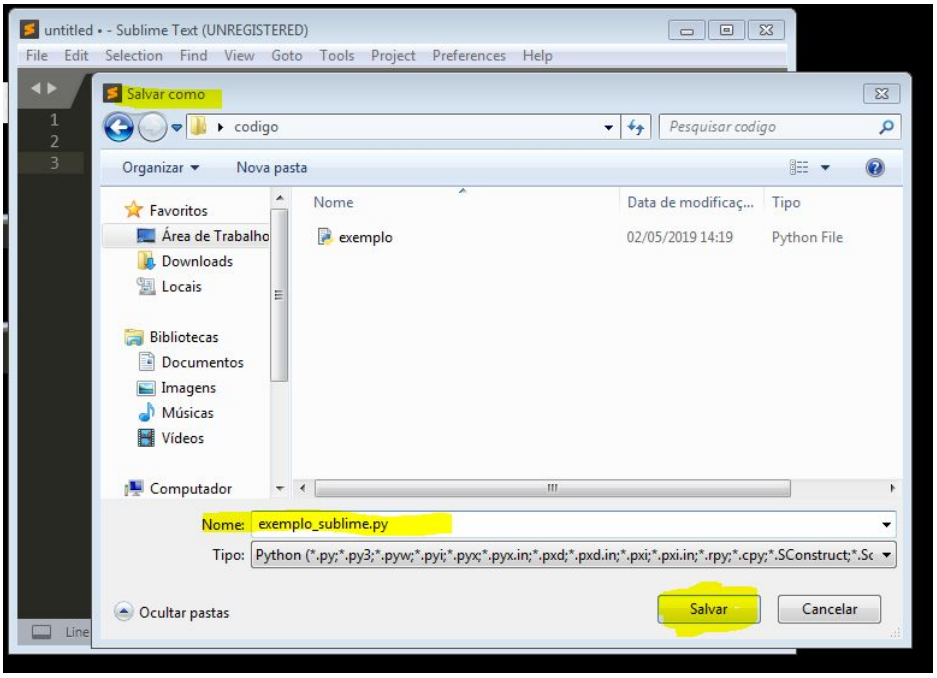


Imagem 17

Selecione a pasta onde irá salvar o código e de um nome para o arquivo, com a extensão .py.

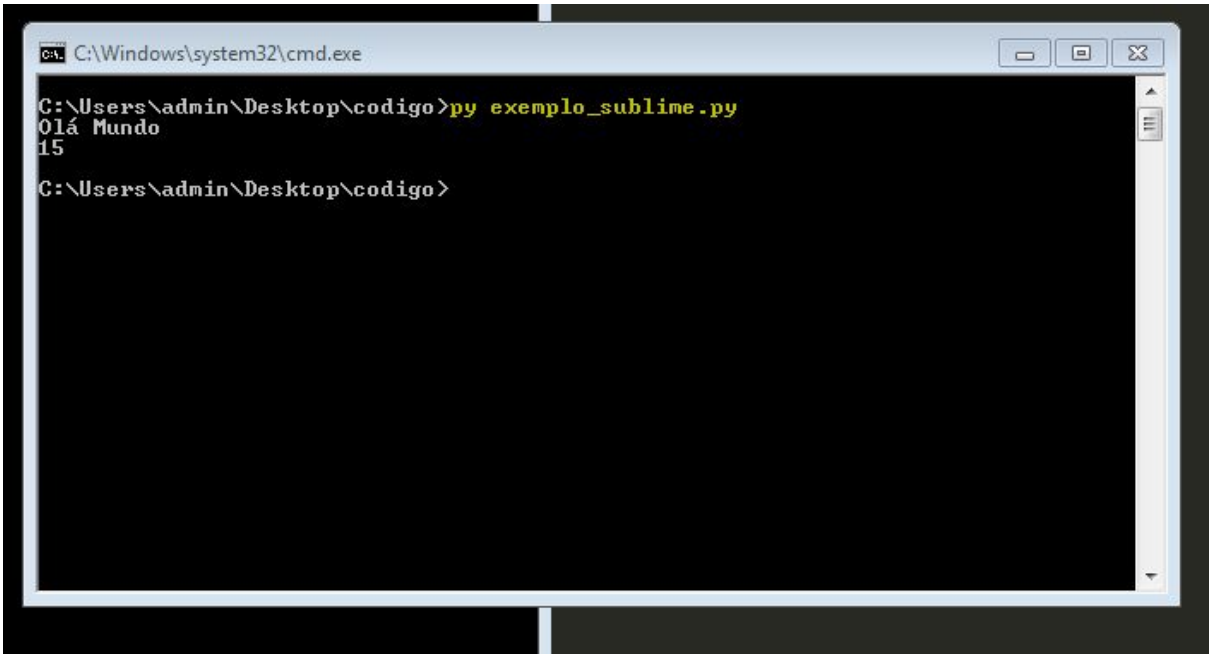


Imagem 18

Execute o programa Python pelo prompt de comando, navegando até a pasta onde salvou o arquivo e digitando: py <nome do arquivo.py>, ou “python”. Podemos executar também pelo IDLE, indo em “File” e “Open”.

➤ **DICA:** Existem outros editores como: Gedit, Atom, Notepad ++, Brackets, Microsoft Visual Studio Code. etc.

Utilizando Python como uma calculadora

Vamos iniciar no mundo da programação primeiro utilizando a linguagem *Python* como uma calculadora.
Abra o IDLE do Python (Imagem 4), no interpretador cada comando é *avaliado* e *executado,* resultando na próxima linha com a resposta (ou saída).

➢ **DICA:** Uma boa maneira de aprender a programar é usando um interpretador em modo *interativo*. Dessa forma você pode digitar comandos linha por linha e observar a cada passo como o computador interpreta e executa esses comandos.

Para fins didáticos, usaremos o sinal de `>>>` para indicar que estamos usando o IDLE Shell do *Python*.

```
>>> 1 + 2
3
```

Ao executarmos a operação `1 + 3` o interpretador nos dá a saída `4`. Caso aconteça alguma operação errada receberemos uma mensagem de erro, *exception* (uma exceção). Como exemplo, some 1 ao caracter 'a':

```
>>> 1 + 'a'
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    1 + 'a'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

O Python avaliou o trecho do código acima e tentou executar, essa operação não é suportada, pois estamos somando um número inteiro(int) a uma letra (str).

Observe na última linha da mensagem, a primeira palavra diz o tipo de erro (exceção) "**`TypeError`**", depois informa a descrição do erro "***`unsupported operand`***", também observamos *onde* ocorreu o erro "`line 1`", ou seja, na primeira linha.

Abaixo tentaremos efetuar uma divisão: dividir um número inteiro por zero:

```
>>> 10/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    10/0
ZeroDivisionError: division by zero
```

Na última linha note o tipo de erro: "`ZeroDivisionError`" e a descrição "`division by zero`".

➤ **DICA:** Sempre devemos ler e entender os erros para acharmos onde há problemas em nosso código. Aprenda com o erro!

Vamos continuar com nossos testes, observe a operação abaixo:

```
>>> 1 + 2 * 3
7
>>>
```

Note que o Python segue uma regra de operação, uma **precedência de operação**. Primeiro efetuou a multiplicação: `2 * 3`, o resultado dessa multiplicação é somado ao 1. Isso acontece por causa da precedência, a multiplicação é efetuada antes de uma soma. Se executarmos o mesmo código, porém agora com parênteses, teremos um resultado diferente, veja:

```
>>> (1 + 2) * 3
9
>>>
```

Regra geral para a ordem das operações: parênteses, exponenciação, multiplicação, divisão, soma e subtração.

➤ **DICA:** Podemos resumir a regra a uma fácil memorização no seguinte acrônimo: **P.E.M.D.A.S.**

Cada letra é uma operação: **P**arênteses**, E**xponenciação, **M**ultiplicação, **D**ivisão, **S**oma **S**ubtração. Dessa maneira fica fácil decorar.

Conhecendo os operadores

Python possui operadores de comparação, operadores lógicos (booleanos), operadores aritméticos e operadores bit-a-bit..

Operadores de comparação

operador	descrição	exemplo
<	menor que	2 < 3
<=	menor igual que	2 <= 2
>	maior que	3 > 2
>=	maior igual que	3 >= 3
==	igual	10 == 10
!=	diferente	10 != 9
is	é (identidade do objeto)	“oi” is “oi”
is not	não é (negação de identidade do objeto)	“oi” is not “ei”

Operadores aritméticos

Operador	Descrição	Observação
x + y	soma x e y	
x - y	diferença entre x e y	
x * y	produto de x por y	
x / y	divisão de x por y	
x // y	divisão inteira de x por y	
x % y	resto da divisão de x por y	operador módulo

-x	valor negativo de x	
+x	valor positivo de x	
abs(x)	valor absoluto de x	
int(x)	converte x para tipo int	
float(x)	converte x para tipo float	
divmod(x, y)	retorna o quociente e o resto da divisão de x por y	divmod(10, 2) -> (5, 0)
pow(x, y)	x elevado a y	potência igual x ** y
x ** y	x elevado a y	potência igual pow(x, y)
complex(re, im)	um número complexo com a parte real *re*, parte imaginária *im*. *im* por padrão é zero.	
c.conjugate()	Conjugado de um número complexo c	

Operadores Lógicos (booleanos)

Operação	Resultado	Notas
x **or** y	se x é falso, então y, senão x	(1)
x **and** y	se x é falso, então x, senão y	(2)
not x	se x é falso, então verdadeiro, senão falso	(3)

Notas

(1) Avaliação de curto-circuito em Python. O segundo argumento só é avaliado se o primeiro for falso (ou seja, o segundo).

(2) Avaliação de curto-circuito em Python. O segundo argumento só é avaliado se o primeiro é verdadeiro.

(3) not, operador de negação. É um operador de baixa prioridade. Na expressão `not a == b` é interpretado como `not(a == b)`.

Operadores bit-a-bit (bitwise)

São utilizados quando precisamos realizar operações a nível de bits com números inteiros, ou seja, trabalhar com representação a binária.

Operador	Descrição
`x << y`	`retorna x com os seu bits movidos y casas à esquerda`
`x >> y`	`retorna x com o seus bits movidos y casas à direita`
`x & y`	`faz um “E” binário.`
`x | y`	`faz um “OU” binário.`
`^`	`faz um “OU Exclusivo” binário`
`~x`	`retorna o complemento de x`

Tipos básicos

Antes de você aprender sobre os tipos básicos, é necessário compreender como o *Python* testa os valores verdade. Por teste de valor verdade entendemos aquelas operações que nos retornam sempre Verdadeiro ou Falso. (`True, False`).

Qualquer objeto em Python pode ser testado quanto ao valor verdade em uma condição (`if`), repetição (`while`) ou **expressões** que utilizam operações booleanas.

Padrões de Verdadeiro e Falso:

- Por padrão um objeto é considerado Verdadeiro (*`True`*), a menos que em sua classe seja definida um método `__bool__()` que retorne *`False`* ou um método `__len__()` que retorne `0` (zero) (Não se preocupe agora em entender este ponto).
- `0` (zero) de qualquer tipo numérico sempre é *`False`*.
- Constantes definidas como *`False`*: *`False`* e *`None`*.
- Sequências de coleção vazias são sempre *`False`*: `{}, [], (), set(), range(0)`
- Operações e built-in functions (funções internas) que possuem resultado booleano sempre retornam `0` ou `False` para `False` e `1` ou `True` para `True`, a menos que seja indicado o contrário. (Um exceção importante: As operações booleanas `and, or` sempre retornam um de seus operandos).

➤ **DICA:** Faça vários testes com expressões booleanas para compreender bem o funcionamento deste recurso em Python. Observe no tipo básico *`bool`* na página Tipo bool.

Variáveis são **espaços de memória**, utilizamos para guardar **valores**. Os valores, em Python podem ser de vários **tipos**, como: *`int, float, bool, str, list, tuple, set, dict, complex`* (existem outros, por enquanto trataremos destes).

tipo	descrição	exemplo
`int`	`inteiro`	`42`
`float`	`decimal/ponto flutuante/ número real`	`42.57`
`bool`	`booleano / verdadeiro ou falso`	`True, False`

str	cadeia de caracteres, texto, uma string	*“Olá Mvndo”*
list	lista heterogênea de elementos	[2, “olá”, 42.74]
tuple	lista heterogênea imutável de elementos	(2, “olá”, 42.74)
set	conjunto ordenado e sem repetição de elementos	{1, 2, 5, 6, 7}
dict	dicionário, array associativo	{“telefone”: “9999-1234”}
complex	número complexo. (real e imaginária)	2+3j

Veremos a seguir, através do interpretador interativo como usar cada tipo de dado com os operadores.

Funções embutidas (built-in Functions)

Funções embutidas são funções que sempre estão disponíveis, podemos usar em qualquer momento.

Função `len(iter)`

```
# função len(iter) retorna o tamanho do objeto, (um int)

nome = "isaura"
len(nome) # 6

lista = [1, 2, "b", 3, "a"]
len(lista) # 5

tupla = (1,2,3,4, "abc")
len(tupla) # 5

dicionario = {'joao': 12, 'pedro': 18}
len(dicionario) # 2
```

Função `range(parada)`
Retorna uma sequência de 0 até parada (objeto do tipo range). Para converter para lista use a função list().

```
# usando o range, lembrando que retorna um iterável
```

```
range(5) # range(0, 5)
range(1,10) # range(1, 10)
range(4,6) # range(4, 6)
range(1, 10, 1) # range(1, 10)
range(1, 10, 2) #range(1, 10, 2)

# convertendo para list
list(range(5)) # [0, 1, 2, 3, 4]
list(range(1, 10)) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
list(range(4, 6)) # [4, 5]
list(range(1, 10, 1)) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
list(range(1, 10, 2)) # [1, 3, 5, 7, 9]
```

Função `print()`

Imprime os objetos passados por parâmetro. Pode possuir 1 ou mais parâmetros, separados por vírgula. A função print também contém um parâmetro nomeado ***`end='\n'`*** *que* por padrão faz uma quebra de linha cada imprime, pode-se alterá-lo.

```
print("olá mvndo") # olá mvndo
print("olá", 'mvndo') #olá mvndo
print('ola', 2, 3, 'mvndo') # ola 2 3 mvndo
print('ola', [2,3,"lista"], "mvndo") # ola [2, 3, 'lista'] mvndo
print('olá', range(3)) # olá range(0, 3)

# mudando o parâmetro end
print(1, 2, 3, end='...') # 1 2 3...

# em um for
for i in range(5):
    print(i, end='...')

# saída
# 0...1...2...3...4...
```

Função `input(str)`

Essa função captura o que foi informado via teclado e retorna o conteúdo em tipo `str`. Recebe por parâmetro uma `str` que será impressa. `input()` sempre retorna uma `str`, para fazer a conversão de tipos, faça explicitamente.

```
# retorna str
x = input('digite algo: ') # digite algo:
# retorna int
x = int(input("numero: ")) # numero:
# retorna floar
x = float(input("numero: ")) # numero
```

Função `max(iter)`
retorna o maior elemento de um iterador.
Função `min(iter)`
retorna o menor elemento de um iterador
Função `sum(iter)`
retorna o somatório de todos os elementos de um iterador

```
d = [1, 2, 7, 10, 68, 14, 7, 8]
max(d) # 68
min(d) # 1
sum(d) # 117
```

Função `str(obj)`
Converte o objeto em str

```
# int pra str
str(1000) # '1000'
str(10_000) # '10000'

# float pra str
str(67.8) # '67.8'

# list pra str
str([1,2,3]) # '[1, 2, 3]'

# tuple pra str
str((4,5)) # '(4, 5)'

# range pra str
str(range(5)) # 'range(0, 5)'

# range pra list para str
str(list(range(5))) # '[0, 1, 2, 3, 4]'

# dict pra str
str({1:'um', 2: 'dois'}) # "{1: 'um', 2: 'dois'}"
```

Função `enumerate(iter)`
Retorna uma tupla numerada com os elementos do objeto iterável

```
lista = ['homem', 'macaco', 'veado']

# retorna um objeto enumerate
enumerate(lista) # <enumerate object at 0x000001FE42E5E048>
```

```
# gera um lista
list(enumerate(lista))
# [(0, 'homem'), (1, 'macaco'), (2, 'veado')]
```

Ou em um **for**:

```
lista = ['homem', 'macaco', 'veado']

for indice, valor in enumerate(lista):
     print(indice, valor)

# saída:
# 0 homem
# 1 macaco
# 2 veado
```

➤ **DICA:** Existem outras inúmeras funções embutidas, pode ver mais delas no site oficial pelo link: docs.python.org/3/library/functions.html

➤ **DICA:** quando falamos em **iteradores**, estamos falando de objetos que podem ser percorridos por um loop. Por exemplo, **`str, list, tuple, dict, range, set`**... são todos iteráveis. Por enquanto esta definição é suficiente.

Tipo int

Observe abaixo a manipulação de tipo inteiro, com operações de soma, subtração, multiplicação, divisão, etc... São autoexplicativas.

```
>>> 12345677889233
12345677889233
>>> 34
34
>>> 10 + 2
12
>>> 10 - 2
8
>>> 2 - 10
-8
>>> 10 * 2
20
>>> 10 / 2
5.0
```

```
>>> 10 // 2
5
>>> 10 % 2
0
>>> divmod(10, 2)
(5, 0)
>>> abs(2 - 10)
8
>>> int('345')
345
>>> float(10)
10.0
>>> pow(10,2)
100
>>> 10 ** 2
100
>>> complex(10,2)
(10+2j)
>>> complex(10,2).conjugate()
(10-2j)
```

Tipo float

Observe abaixo a manipulação de tipo float, com operações de soma, subtração, multiplicação, divisão, etc... São autoexplicativas.

```
>>> 15.4 + 2.01
17.41
>>> 15.4 - 2.01
13.39
>>> 2.01 - 15.4
-13.39
>>> 15.4 * 2.01
30.953999999999997
>>> 15.4 / 2.01
7.661691542288558
>>> 15.4 // 2.01
7.0
>>> 15.4 % 2.01
1.3300000000000018
>>> divmod(15.4, 2.01)
(7.0, 1.3300000000000018)
>>> abs(2.01 - 15.4)
13.39
>>> int(15.4)
15
```

```
>>> float('15.4')
15.4
>>> pow(15.4, 2.01)
243.7342991192041
>>> 15.4 ** 2.01
243.7342991192041
>>> complex(15.4, 2.01)
(15.4+2.01j)
>>> complex(15.4, 2.01).conjugate()
(15.4-2.01j)
```

Tipo bool

Observe abaixo a manipulação de tipo booleano, com operações de comparação.

```
>>> True > False
True
>>> True < False
False
>>> True >= False
True
>>> True <= False
False
>>> True == False
False
>>> True != False
True
>>> not True
False
>>> not False
True
>>> True and True
True
>>> True or True
True
>>> True and False
False
>>> True and False
False
>>>
```

```
>>> 1 and True
True
>>> 1 or False
1
>>> 1 or True
1
>>> not 1
False
>>> not 1 and True
False
>>> [] or {} or () or range(0) or True
True
>>> True or 1 or 2 or ''
True
>>> not ''
True
>>> not []
True
>>> not {}
True
>>> not ()
True
>>> not range(0)
True
>>> not set()
True
```

Tipo str

O tipo ***str*** (string) é uma sequência de caracteres imutáveis.
Podem ser definidas usando aspas simples, duplas e triplas (simples/triplas).

O tipo de dado ***str***, possui ***métodos***.
Métodos são como funções que manipulam o próprio objeto. Por exemplo, como uma variável ***str*** é imutável, se usarmos o método ***.upper()*** retornará uma nova ***str***, que será necessário atribuir a uma variável (podendo ser ela mesma):

```
1    >>> nome = "chapolin"
2    >>> print(nome)
3    chapolin
4    >>> print(nome.upper())
5    CHAPOLIN
6    >>> print(nome)
7    chapolin
8    >>> nome = nome.upper()
9    >>> print(nome)
10   CHAPOLIN
```

Na ***linha 1***, criamos uma variável do tipo str com o valor "`chapolin`".
Na ***linha 2*** imprimos a variável, o resultado sai na ***linha 3.***
Na linha 4, usamos o método ***.upper()*** que converte todas as letras minúsculas em maiúsculas, e usamos a função print para imprimir o resultado disso na tela (***linha 5***).
Na ***linha 6***, imprimimos novamente a variável ***nome***, observe que seu conteúdo não foi alterado, pois str é um tipo imutável. É necessário fazer uma atribuição novamente (***linha 8***).
Note que o método ***.upper()*** retorna uma nova str, que será atribuída a variável nome.

Podemos acessar uma string pelo seu ***index***. (índice)

```
"python"[0] # 'p'
"python"[1] # 'y'
"python"[2] # 't'
"python"[3] # 'h'
"python"[4] # 'o'
"python"[5] # 'n'
```

ou

```
ling = "python"
ling[0] # 'p'
ling[1] # 'y'
ling[2] # 't'
ling[3] # 'h'
ling[4] # 'o'
ling[5] # 'n'
```

`str` é um tipo ***iterável***, ou seja podemos acessar ele usando o loop ***`for`***:

```
for letra in "python":
     print(letra)


'''
p
y
t
h
o
n
'''
```

Podemos "fatiar" (slice):

```
s = "constantino"
s[0:5] # 'const'
s[0:3] # 'con'
s[:3]  # 'con'
s[1:3] # 'on'

# Acessando pelo final
s[-1] # 'o'
s[-2] # 'n'

# Acessando completamente
s     # 'constantino'
s[:]  # 'constantino'
s[0:] # 'constantino'
```

Pesquisando na `str`

Podemos pesquisar um string com o operador ***`in`***, exemplo

```
"cons" in "constantino"   # True
"a" in "constantino"      # True
"d" not in "constantino"  # True
```

```
"b" in "constantino"      # False
"ino" in "constantino"    # True
"es" in "constantino"      # False
```

Métodos comuns para `str`

Função `len(obj)`

parâmetro obj: qualquer objeto iterável
retorno int
Mostra o tamanho da string

```
s = "foo"
len(s)
# 3
```

Método `.upper()`

Retorno str:
Caixa alta.

```
"foo".upper() # FOO
```

Método `.lower()`

Retorno str:
Caixa baixa.

```
"ALFA".lower() # alfa
```

Função `str(obj)`

Converte em string.

```
num = 123
type(num)      # <class 'int'>
type(str(num)) # <class 'str'>
```

Método .isalpha()

Retorna False se a string contiver algum caractere que não seja letras

```
"abc".isalpha()  # True
"1fg".isalpha()  # False
"123".isalpha()  # False
"/+)".isalpha()  # False
```

Método .strip()

Retira espaços em branco no começo e no fim

```
" sobrando espaços ".strip()      # 'sobrando espaços'
"  sobrando espaços   ".strip()   # 'sobrando espaços'
```

Método .join()

Junta cada item da string com um delimitador especificado.
É o inverso do split().

```
", ".join("abc")
# 'a, b, c'
# Aceita listas.
"-".join(['papa', 'sao', 'pio', 'x'])
# 'papa-sao-pio-x'
```

Método .split(delimitador)

Separa uma string conforme um delimitador.
É o inverso do join().

```
s = 'n o m e'
s.split()      # ['n', 'o', 'm', 'e']
s.split(" ")   # ['n', 'o', 'm', 'e']
s.split("")    # ValueError: empty separator
```

Método .replace(x, y)

retorna uma **cópia** da ***str*** com todos os valores de ***x*** substituídos pelo valor de ***y***.

```
frase_antiga = 'eu gosto da colégio Constantino'
frase_nova = frase_antiga.replace('colégio', 'escola')
print(frase_nova)
# 'eu gosto da escola Constantino'
```

Método .find(sub)

parâmetro sub: str, uma substring que será pesquisada na variável str
retorno int: posição (índice) da primeira ocorrência de sub
Se não encontrar a ocorrência retorna -1

```
>>> "constantino".find("o")
1
>>> "constantino".find("tino")
7
>>> "constantino".find("tan", 3,10)
4
>>> "constantino".find("tan", 9,10)
-1
```

Note que podemos passar como parâmetro a posição inicial e final de onde queremos começar a procurar sub dentro da str: .find(sub, inicio, fim)

Método .format()

O método `format()` serve para criar uma `str` que contém campos entre chaves a serem substituídos pelos argumentos de format. Exemplo:

```
mensagem = "Olá {0}, uma {1}!"
print(mensagem.format('Jhon', 'Boa tarde'))

'''
Olá Jhon, uma Boa tarde!
'''

'parametro 1: {0}  parametro 2: {1}'.format('peixe', 'pássaro')

'''
```

```
parametro 1: peixe  parametro 2: pássaro
'''

msg = 'nome: {0}, idade: {1}, sexo: {2}'

pessoa = msg.format('arthur', 17, 'masculino')

print(pessoa) # nome: arthur, idade: 17, sexo: masculino
```

Tipo list

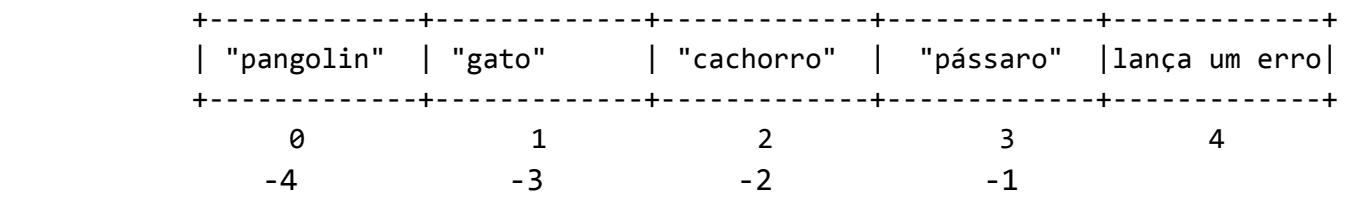
List é uma estrutura de dados heterogênea, ou seja, pode ter qualquer tipo de dados dentro dela.
List é mutável, podemos mudar seu conteúdo.
Veja os exemplos:

```
# criar uma lista vazia
animais = []

# ou usa a função list() que retorna uma lista vazia
animais = list()

# atribui diretamente os valores
animais = ["pangolin", "gato", "cachorro", "pássaro"]
animais[0] # 'pangolin'
animais[1] # 'gato'
animais[2] # 'cachorro'
animais[3] # 'pássaro'
animais[-1] # 'pássaro'
animais[-2] # 'cachorro'
animais[-3] # 'gato'
animais[:]  # '[pangolin', 'gato', 'cachorro', 'pássaro']
animais[2:] # ['cachorro', 'pássaro']
animais[2:3] ['cachorro']
animais[2:10] ['cachorro', 'pássaro']
```

Contamos o índice da lista a partir do zero (e também de maneira reversa), veja:



Ao acessar um índice inexistente recebemos um erro.

```
animais[4]   # IndexError: list index out of range
```

Percorrendo uma lista usando o loop **for**:

```
animais = ["pangolin", "gato", "cachorro", "pássaro"]
for animal in animais:
      print(animal)

# saída:
# pangolin
# gato
# cachorro
# pássaro
```

Percorrendo uma lista usando loop **while:**

```
animais = ["pangolin", "gato", "cachorro", "pássaro"]

indice = 0 # contador pro índice

tamanho = len(animais) # tamanho total da lista

while indice < tamanho:
      print(animais[indice])
      indice += 1 # incremento do contador

# saída:
# pangolin
# gato
# cachorro
# pássaro
```

Usando a função **enumerate** para pegar o índice:

```
for indice, animal in enumerate(animais):
      print(indice, animal)
```

```
# saída:
# 0 pangolin
# 1 gato
# 2 cachorro
# 3 pássaro
```

Métodos da list

Método `.append(valor)`

Insere no fim da lista o valor passado por parâmetro.

```
minha_lista = []
minha_lista.append("caderno")
minha_lista.append("lapis")
print(minha_lista) # ['caderno', 'lapis']
```

Método `.pop()`

Remove e retorna o último elemento da lista:

```
minha_lista = ["caderno", "lapis"]
ultimo = minha_lista.pop()
print(ultimo) # lapis
```

Podemos usar o **`.pop()`** passando o índice do elemento que desejamos retornar e remover, veja:

```
minha_lista = ['caderno', 'lapis', 'borracha']
elemento = minha_lista.pop(1) # o segundo elemento da lista
print(elemento) # lapis
```

Método `.insert(indice, valor)`

Insere o parâmetro ***`valor`*** na posição indicada pelo parâmetro ***`indice`***:

```
minha_lista = ['caderno', 'lapis', 'borracha']
minha_lista.insert(0, 'caneta')
print(minha_lista) # ['caneta', 'caderno', 'lapis', 'borracha']
```

Método `.count(valor)`

Retorna a quantidade de ocorrências de ***`valor`*** na lista:

```
lista = ['a', 'b', 'c', 'a', 'a', 'd', 'e']
q = lista.count('a')
print(q) # 3
```

Método `.remove(valor)`

Remove a primeira ocorrência do parâmetro ***`valor`*** na lista:

```
lista = ['a', 'b', 'c', 'a', 'a', 'd', 'e']
lista.remove('a')
print(lista) # ['b', 'c', 'a', 'a', 'd', 'e']
lista.remove('a')
print(lista) # ['b', 'c', 'a', 'd', 'e']
```

Método `.index(valor)`

Retorna o índice (posição) da primeira ocorrência do parâmetro ***`valor`*** na lista:

```
lista = ['b', 'c', 'a', 'd', 'e']
indice = lista.index('a')
print(indice) # 2
```

Podemos passar por parâmetro o início e o fim do índice dentro da lista que queremos procurar, por exemplo:

```
lista = ['b', 'c', 'a', 'd', 'e', 'a']

# começa a procurar do índice 3 até o 6
indice = lista.index('a', 3, 6)
print(indice) # 5
```

Método `.clear()`

remove todos os elementos da lista, deixando-a vazia:

```
lista = ['b', 'c', 'a']
lista.clear()
print(lista) # []
```

Método `.copy()`

Copia a lista, sem manter a referência.
Temos duas maneiras de copiar uma lista em Python: uma é atribuindo uma nova variável a lista, por exemplo: `lista2 = lista1`
O que mantém a referência, isto é: o que modificar em `lista1` será também modificado em `lista2`. A outra maneira é usando o método **`.copy()`**. Veja o exemplo dos dois casos:

```
# copia mantendo a referência

lista1 = ['a', 'b', 'c']
lista2 = lista1

lista1.remove('a')

print(lista1) # ['b', 'c']
print(lista2) # ['b', 'c']

# copia sem manter a referência

lista1 = ['a', 'b', 'c']
lista2 = lista1.copy()

lista1.remove('a')

print(lista1) # ['b', 'c']
print(lista2) # ['a', 'b', 'c']
```

Método `.extend(lista)`

Insere cada elemento da lista passada por parâmetro:

```
lista = ['a', 'b', 'c']
lista2 = ['d', 'e']
lista.extend(lista2)

print(lista) # ['a', 'b', 'c', 'd', 'e']
```

Método `.reverse()`

Inverte a ordem (inverso) dos elementos da lista (método *in place*), veja:

```
lista = ['d', 'c', 'b', 'a']
```

```
lista.reverse()
print(lista) # ['a', 'b', 'c', 'd']
```

Método `.sort()`

Ordena os elementos da lista(método *in place*) do menor para o maior , veja:

```
lista = [10, 9, 4, 5, 6, 1, 2, 2, 7]
lista.sort()
print(lista) # [1, 2, 2, 4, 5, 6, 7, 9, 10]
```

Para ordenar do maior para menor use o parâmetro `reverse=true`:

```
lista = [10, 9, 4, 5, 6, 1, 2, 2, 7]
lista.sort(reverse=True)
print(lista) # [10, 9, 7, 6, 5, 4, 2, 2, 1]
```

Tipo tuple

O tipo **`tuple`** é semelhante ao tipo **`list`**, é uma sequência de elementos, a diferença consiste em que as tuplas são imutáveis. Elas podem ser acessadas também por índice:

```
# cria uma tupla vazia
tupla = ()
# outra forma
tupla = tuple()
# usa-se parênteses
tupla = (1, 2, 3)
tupla = ('a', 'b', 555)
```

Método `.index(valor)`

Retorna o índice (posição) da primeira ocorrência do parâmetro ***`valor`*** na tupla:

```
tupla = (1, 2, 'dragão', 1)

tupla.index('dragão') # 2
tupla.index(1) # 0
```

Método `.count(valor)`

Retorna a quantidade de ocorrências de ***`valor`*** na tupla:

```
tupla = (1, 2, 'dragão', 1)
tupla.count(1) # 2
```

O acesso por índice, e o uso do fatiamento (slice) é idêntico ao do tipo `list`.

Tipo dict

Dicionário (**`dict`**) é uma poderosa estrutura de dados em Python, são conhecidas como tabela hash (tabela de dispersão ou *hash table*).
É uma estrutura de dados especial, que associa chaves de pesquisa a valores. Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado.
Ela é composta de chaves e valores. As chaves do dicionário são únicas (não podemos ter um dicionário com chaves repetidas) e os valores de qualquer tipo, o tipo da chave pode ser **`int`** ou **`str`**. Vamos ao exemplo:

```
# para criar um dicionário vazio usamos chaves
idade = {}
# ou criando um novo objeto com a função dict()
idade = dict()
# com parâmetros e valores na função dict
idade = dict(maria=10, pedro=12, amanda=17)

# separamos as chaves e os valores por dois pontos ':'
idade = {'maria': 10, 'pedro': 12, 'amanda': 17}

# para acessar o valor, colocamos a chave do dicionário entre
# colchetes, como na lista.
idade['pedro'] # retorna: 12
idade['amanda'] # 10

# para criar novas chaves e valores no mesmo dicionário
# podemos fazer isso apenas atribuindo, veja:
idade['jair'] = 17

# podemos usar inteiros como chave
idade[17] = 'jair'

# tamanho do dicionário usando len()
len(idade) # 3
```

Para verificar se existe uma chave no dicionário usamos o operador **in**

```
telefone = {'joao': '9999-1234', 'simao': '9999-4321'}
'joao' in telefone  # True
'cesar' in telefone # False
'joao' not in telefone # False
'cesar' not in telefone # True
```

Para **deletar** uma chave em um dicionário usamos o **del**:

```
telefone = {'joao': '9999-1234', 'simao': '9999-4321'}
del telefone['joao']
print(telefone) # {'simao': '9999-4321'}
```

Métodos

Método	**Descrição**
dict.clear()	Remove todos os elementos do dicionário.
dict.copy()	Retorna uma cópia sem referência.
dict.fromkeys(chaves, valor)	cria um novo dicionário com as chaves (iterável) com os valores = valor.
dict.get(chave, valor)	Retorna o valor que contém a chave passada, caso não exista a chave retorna valor.
dict.keys()	Retorna uma lista contendo as chaves do dicionário
dict.items()	Retorna uma lista com os pares de chave-valor em tupla
dict.setdefault(chave, valor)	se a chave estiver em dict, retorna o valor, senão adiciona a chave com o valor

dict.update(dict2) dict.update(chave=valor)	Atualiza o dicionário. Cada par chave-valor em dict2 será copiado ou os pares de chave-valor.
dict.values()	Retorna uma lista com os valores do dicionário. (obj dict_values), para converter para lista use list()
dict.pop(chave, valor)	remove a chave do dicionário e retorna o seu valor, caso não exista a chave retorna o valor passado por parâmetro, se não for informado valor ocorre um erro KeyError.

Dicionários suportam comparação de igualdade ==, outras comparações (‘<’, ‘<=’, ‘>=’, ‘>’) levanta um erro *TypeError*.
Dicionários preservam a ordem de inserção. Veja o uso dos métodos:

```
# criando um dicionário pela função dict()
d = dict(um=1, dois=2, tres=3)

# criando diretamente com chaves
d = {'um': 1, 'dois': 2, 'tres': 3}

# remove todos os elementos
d.clear()
print(d) # {}

# atribuindo novo par chave-valor
d['tres'] = 3

# copia sem referência
d2 = d.copy()

# comparação
d2 == d # True

# método .fromkeys retorna novo dicionário
d = d.fromkeys(('joao', 'pedro'), 1) # {'joao': 1, 'pedro': 1}

# retorna valor da chave
d.get('joao') # 1

# se a chave não existir pode retornar o valor
# passado no segundo parâmetro
```

```
d.get('goku', 'não tem') # não tem

# lista com as chaves (dict_keys object)
chaves = d.keys() # ['joao', 'pedro']

# lista com tuplas chave-valor
pares = d.items() # [('joao', 1), ('pedro', 1)]

# atualiza
d.update(maria=123) # {'joao': 1, 'pedro': 1, 'maria': 123}

# ou
novo = {'um': 1, 'dois': 2}
d.update(novo)

# retorna uma lista com os valores
d.values()

# remove a chave e retorna o valor
d.pop('um') # 1

# se não existir a chave, levanta um erro
d.pop('banana') # KeyError: 'banana'

# para retornar mesmo não existindo a chave uso o 2º parâmetro
d.pop('banana', 123) # 123
```

Percorrendo um dicionário usando **`for loop:`**

```
# apenas as chaves
for chave in dicionario:
     print(chave)

# chave e valor
for chave, valor in dicionario.items():
    print(chave, valor)
```

Tipo Complex

O tipo `complex` será visto em breve.

Tipo Set

O tipo `set` será visto em breve.

Estrutura de Controle

A estrutura de decisão: `if, elif e else`.

Simplesmente conhecer os tipos básicos não é suficiente para programar, precisamos de estruturas que manipulam a lógica do nosso programa. Essas estruturas são conhecidas como estrutura de controle.

`if`

O **`if`** é uma estrutura de controle (de condição) que permite avaliar uma expressão e, de acordo com seu resultado, executar um bloco de comando.
A avaliação da expressão em relação ao *valor verdade.*

```
idade = 19
if idade >= 18:
    print('maior de idade')
```

`else`

Vimos anteriormente como utilizar o **`if`** para executar uma ação caso uma condição seja atendida. No entanto, nenhum comportamento específico foi definido para o caso de a condição não ser satisfeita. Quando isso é necessário, precisamos utilizar a reservada **`else`**.

```
idade = 17
if idade >= 18:
    print('maior de idade')
else:
    print('menor de idade')
```

`elif`

Adicionalmente, se existir mais de uma condição alternativa que precisa ser verificada, devemos utilizar o **`elif`** para avaliar as expressões intermediárias antes de usar o **`else`**:

```
idade = 18
if idade < 12:
    print('criança')
elif idade < 18:
    print('adolescente')
elif idade < 60:
    print('adulto')
else:
    print('idoso')
```

Estrutura de repetição `while` e `for` (loop)

É comum que uma mesma instrução (ou conjunto delas) precise ser executada várias vezes seguidas. Nesses casos, utilizamos um loop (ou laço de repetição), que permite executar um bloco de código repetidas vezes, enquanto uma dada *condição* é atendida.

`while`

O loop **`while`** avalia uma expressão e continua repetindo seu bloco de código enquanto a expressão for *verdadeira*. Sempre observe a ***condição de parada*** do seu loop, para que seu laço de repetição não fique em loop infinito. Veja algumas maneiras de usar o loop:

```
while True:
    print("olá mvndo")
    break
```

Esse loop acima tem uma expressão de condição que é lida como *"enquanto for verdadeiro … repita*", sempre `True` será `True`, o que causaria um loop infinito. Note que depois do `print` há um comando **`break`**, é com ele que paramos um loop.

➢ **DICA:** A instrução **`break`** e a instrução **`continue`** são ferramentas das estruturas de repetição (`while`, `for`) permitindo a interrupção de um único ciclo ou do laço de repetição. A instrução **`break`** interrompe não somente o ciclo em execução, mas sim, **todo o laço**, enquanto que a instrução **`continue`** finaliza um único laço, fazendo com que o Cursor de execução vá para o cabeçalho da estrutura de repetição.

Podemos usar o `while` com um contador para controlar a parada do loop.

```
contador = 0
while contador < 10:
      print(contador)
      contador += 1

# saída: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

a variável contador é usada na **expressão** do loop `while` e dentro do loop essa variável é **incrementada** usando o operador incremento `+=` que é um açúcar sintático (*syntax sugar*) para `contador = contador + 1`

Podemos usar o operador de **decremento**:

```
contador = 10
while contador > 0:
      print(contador)
      contador -= 1

# saída: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

a variável contador é usada na **expressão** do loop `while` e dentro do loop essa variável é **subtraída seu valor** usando o operador de decremento `-=` que é um açúcar sintático (*syntax sugar*) para `contador = contador - 1`

Outra maneira de controlar o loop é usar uma estrutura de decisão dentro do laço e checar alguma condição, e com isso inserir um `break` para encerrar o loop.

```
while True:
    nome = input('informe seu nome: ')
    if nome == 'goku':
          print('oi, eu sou o goku!')
          break
```

➤ **DICA:** Sempre pense quando construir seu loop:

- *Qual é a minha condição de parada?*
- *É possível satisfazer essa condição?*

for

O `for` é um loop um pouco diferente do `while`, ele não itera sobre os índices, ele itera sobre os elementos de um iterável. Veja no exemplo:

```
# usando listas
frutas = ['banana', 'maçã', 'tomate']

for fruta in frutas:
    print(fruta)

'''
banana
maçã
tomate
'''

# usando str
vogais = 'aeiou'

for vogal in vogais:
```

```
    print(vogal)

'''
a
e
i
o
u
'''

# tupla
tupla = ('a', 1, 'eggs', 78)
for i in tupla:
    print(i)

'''
a
1
eggs
78
'''

# objeto range
for x in range(5):
     print(x * 2)

'''
0
2
4
6
8
'''

# parando o loop com break
frutas = ['banana', 'maçã', 'tomate']

for fruta in frutas:
    if fruta == 'maçã':
        break
    print(fruta)

'''
banana
'''
```

Algumas técnicas de loop

Usando dicionário

```
tabela_periodica = {'ferro': 26,
'sodio': 11, 'radio': 88, 'litio': 3}

for e, na in tabela.periodica.items():
     print("elemento: ", e, ' n. atomico: ', na)
# saída
'''
elemento:  ferro  n. atomico:  26
elemento:  sodio  n. atomico:  11
elemento:  radio  n. atomico:  88
elemento:  litio  n. atomico:  3
'''
```

Ao fazer um loop em uma sequência (objeto iterável) podemos recuperar o índice (posição) e o valor usando a função embutida **`enumerate`**()

```
for i, v in enumerate(['tic', 'tac', 'toc']):
    print(i, v)

'''
0 tic
1 tac
2 toc
'''
```

Para repetir duas ou mais sequências ao mesmo tempo, as entradas podem ser emparelhadas com a função **`zip()`**

```
perguntas = ['nome', 'QI', 'objetivo']
respostas = ['Arthur', '180', 'ganhar um gato']

for p, r in zip(perguntas, respostas):
    print('Qual é seu {0}?  É {1}.'.format(p, r))
'''
Qual é seu nome?  É Arthur.
Qual é seu QI?  É 180.
Qual é seu objetivo?  É ganhar um gato.
'''
```

Para fazer o loop de uma sequência invertida, primeiro especifique a sequência em uma direção para frente e, em seguida, chame a função **`reversed()`**.

```
for i in reversed(range(1, 10, 2)):
    print(i)
'''
9
7
5
3
1
'''
```

Para fazer um loop com uma sequência ordenada use a função **sorted()** que retorna uma nova lista ordenada enquanto deixa a fonte inalterada:

```
cesta = ['laranja', 'maça', 'pera', 'banana', 'amora', 'morango']
for f in sorted(set(cesta)):
     print(f)

'''
amora
banana
laranja
maça
morango
pera
'''
```

Exercícios Resolvidos

Para aprender a programar é necessário resolver centenas de problemas.
Não existe outra maneira, ou caminho mais curto.
A arte de programar consiste em primeiro, ter um problema, avaliar suas incógnitas, extrair os condicionais e as variáveis, e por fim aplicar um algoritmo lógico para solucionar aquele determinado problema.
Agora que você já conhece as estruturas de dados em Python iremos resolver alguns problemas.

Problema 1: Média de notas de uma sala.

Você precisa criar um programa que vai ler N notas, só pare de receber as notas quando for informado um valor igual a -1 (não computar esse valor).
Após isso, calcular a média e exibir na tela. O total de notas lidas e a média.

```
# cria variável
```

```
contador = 0
somatorio = 0.0

# repita até que nota seja == -1
while True:
    nota = float(input('informe a nota (ou -1 para sair): '))

    # condição de parada do nosso loop
    if nota == -1:
        break
    # se nao entrou no if calcula o somatório
    # e incrementa o contador
    somatorio += nota
    contador += 1

# depois de receber as notas no loop verificamos se houve
# ao menos 1 nota, pra nao dar divisão por zero
if contador >= 1:
    media = somatorio / contador
    print('quantidade de notas lidas: ', contador)
    print('media = ', media)
else:
    print('nenhuma nota lida')
```

Você consegue entender o programa? comente com seus colegas e com o professor se teve alguma dificuldade no código.

Veja que o código acima não usa nenhuma estrutura de dados. Podemos refazer o código acima usando as estruturas de dados e funções/métodos aprendidos anteriormente.

Neste próximo código implementamos usando listas (`list`) e funções embutidas.

```
notas = []

while True:
    nota = float(input('informe a nota (ou -1 para sair): '))
    if nota == -1:
        break
    notas.append(nota)

contador = len(notas)
if contador >= 1:
    media = sum(notas) / contador
    print('quantidade de notas lidas: ', contador)
```

```
    print('media = ', media)
else:
    print('nenhuma nota lida')
```

Você consegue entender o programa? comente com seus colegas e com o professor se teve alguma dificuldade no código.

Problema 2: Taxa de crescimento de uma cidade

Temos dois países, o país A e o país B. Cada um desses países tem um número x de habitantes e um número y de taxa de anual de crescimento. Desenvolva um programa que calcule e escreva o número de anos necessários para que a população do país A ultrapasse ou iguale a população do país B, mantidas as taxas de crescimento. Como exemplo de entrada do programa, use:
País A -> População: 700 mil; Taxa: 1.1
País B -> População 370 mil; Taxa 0.4
Quantos anos são necessários para A ultrapassar B?

Observe o código, e veja que temos 2 loops, um `while` externo e um interno. O `while` externo faz repetir a inserção de dados da cidade e taxa, o `while` interno serve para iterar os anos e calcular quando a população do país A supera ou iguala a população do País B.

```
while True:
    ano = 1
    populacao_a = int(input('População do País A: '))
    populacao_b = int(input('População do País B: '))
    tx_a = float(input('Taxa crescimento do País A: '))
    tx_b = float(input('Taxa crescimento do País B: '))

    while True:
        print("calculando para ", ano, " anos.")
        # regra de três, aumento em numeros da populacao
        pop_a = populacao_a * (tx_a * 0.10)
        pop_b = populacao_a * (tx_b * 0.10)

        # acrescenta as pessoas ao total da populaçao
        populacao_a += pop_a
        populacao_b += pop_b

        # condicao de parada do loop, se população
        # pais A >= pais B
        if populacao_a >= populacao_b:
            print('Pais A: ', populacao_a)
            print('Pais B: ', populacao_b)
            print('Pais A - Pais B: ', populacao_a - populacao_b)
            print("Em: ", ano, " anos.")
            break
        # proximo ano
        ano += 1

    opc = input("inserir outro país? (S) sim ou (N) nao: ")
    if opc in ('N', 'n'):
        break
```

Você consegue entender o programa? comente com seus colegas e com o professor se teve alguma dificuldade no código.

Problema 3: programa de cadastro e venda de produto

Você precisa criar um programa que dê as seguintes opções para o usuário
1 - cadastrar produto
2 - remover produto
3 - vender produto
4 - listar produtos
5 - sair do programa

```
menu = '''1 - cadastrar produto \n
2 - remover produto
3 - vender produto
4 - listar produtos
5 - sair do programa
:
'''
produtos = {}
while True:
    opcao = input(menu)
    if opcao == '1':
        nome = input("qual nome do produto: ")
        preco = float(input("preco: "))
        produtos[nome] = preco # insere

    elif opcao == '2':
        nome = input("qual nome do produto: ")
        # verifica se existe o produto no dicionário
        if nome in produtos:
            del produtos[nome] # remove
        else:
            print('produto: ', nome, ' nao existe no cadastro')

    elif opcao == '3':
        # imprime a lista de produtos
        for nome, preco in produtos.items():
            print('produto: ', nome, ' | preco R$: ', preco)

        nome = input("qual nome do produto: ")
        qnt = float(input("quantidade: "))
        # verifica se existe o produto no dicionário
        if nome in produtos:
            preco = produtos[nome]
            preco_final = qnt * preco
            print("Valor total foi de R$: ", preco_final)
        else:
            print('produto: ', nome, ' não existe no cadastro')

    elif opcao == '4':
        # imprime o nome e o preco dos produtos
        for nome, preco in produtos.items():
            print('produto: ', nome, ' | preco R$: ', preco)
    else:
        print('programa encerrado')
        break
```

Nesse programa usamos a estrutura de dados `dict` (dicionário), os loops `for` e `while`, e estrutura de decisão `if, elif e else.`
Você consegue entender o programa? comente com seus colegas e com o professor se teve alguma dificuldade no código.

Escreva todos esses programas em seu computador e execute para ver o resultado.

Funções (functions)

Uma função é um bloco de código organizado e reutilizável usado para executar uma única ação relacionada. Funções fornecem melhor modularidade para sua aplicação e um alto grau de reutilização de código.
Como você já sabe, o Python oferece muitas funções internas, como `print()`, `input()`, `str()`, `list()`, etc., mas você também pode criar suas próprias funções. Essas funções são chamadas de funções definidas pelo usuário.
Uma função em Python é criada através da declaração **`def`** com a seguinte estrutura:

```
def nome_da_funcao(parametros):
    <bloco de código>
    return
```

Uma função é composta por: nome, parâmetros, o bloco de código e retorno.
Os parâmetros são opcionais e pode ter um ou mais. O comando de retorno é opcional e pode ter apenas 1, porém pode retornar vários valores.

```
def bem_vindo(nome):
    print('Seja bem vindo', nome)

nome = input('Entre com seu nome: ')
bem_vindo(nome) # chamada da função
```