

Introdução Prática à Programação em Python

Minicurso

Rodrigo Amaral

Nassau Tech Week
24 de abril de 2019
Aracaju - Sergipe



UNINASSAU

<http://rodrigoamaral.net/minicurso-python>

Sobre o instrutor

Rodrigo Amaral

- Analista de Tecnologia da Informação no TRT da 20ª Região (SE)
- Mestre em Ciência da Computação pela UFS
- Coorganizador do Python User Group Sergipe (PUG-SE)
- Fã de música, livros e basquete 🎸 📚 🏀



rodrigo@rodrigoamaral.net



amaral101



amaral101



rodrigoamaral



rodrigoamaral



O PUG-SE é uma iniciativa comunitária que tem o objetivo de reunir os desenvolvedores e demais interessados na linguagem de programação Python e em suas tecnologias associadas.

<http://se.python.org.br>



[@pugse](https://t.me/pugse)

Um pouco de história

- Guido van Rossum (Holanda)
- Inspirada na linguagem ABC
- Implementação iniciada em dezembro de 1989, primeira versão pública em fevereiro de 1991
- Nome inspirado no grupo de comédia Monty Python



Por que Python?

Python é uma linguagem de propósito geral que se destaca por ser **poderosa, versátil, legível, fácil de aprender** e por contar com uma vasta e acolhedora **comunidade mundial** de usuários e desenvolvedores.

O aumento do interesse geral por temas como **inteligência artificial, ciência de dados e aprendizado de máquina** vem atraindo o interesse de uma parcela crescente do público de tecnologia por Python.

Os recursos, bibliotecas e ferramentas desenvolvidas na linguagem – tais como **numpy, pandas, scikit-learn, TensorFlow/Keras, e PyTorch**, entre outros – estão entre os mais utilizados por organizações de todos os tamanhos para implementar soluções inteligentes.

Quem usa Python

Organizações de todos os tamanhos e áreas de atuação, órgãos governamentais, universidades, escolas, centros de pesquisa e profissionais liberais descobriram na linguagem Python um diferencial importante para seus negócios e atividades. Entre os casos mais famosos estão **Google, Intel, Facebook, Instagram, Netflix, Spotify, Dropbox, PayPal, NASA, Globo.com** etc.

Principais características

- Interpretada
- Tipagem dinâmica
- Multiplataforma
- Multiparadigma
- “Baterias incluídas”

Interpretador interativo

Python vem com um interpretador interativo que permite executar **uma instrução de cada vez** e mostrar seu **resultado** instantaneamente. Com ele você pode **experimentar** pequenos trechos de código para ver como eles vão se comportar **antes de usá-los** em programas reais.

```
C:\Users\rodrigo>python
```

```
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] on win32
```

```
Type 'help', 'copyright', 'credits' or 'license' for more information.
```

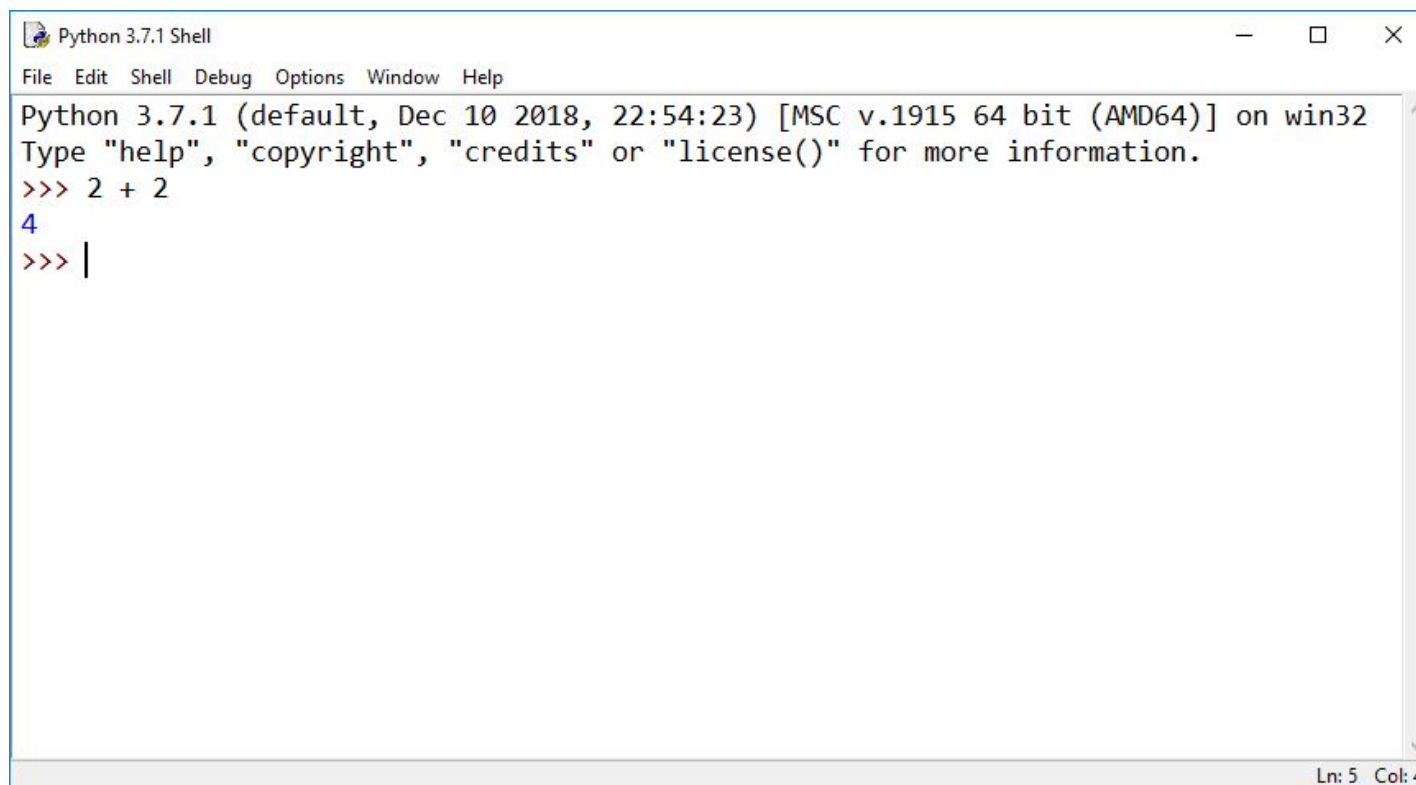
```
>>> 2 + 2
```

```
4
```

```
>>>
```

Ambiente de desenvolvimento

O **IDLE** é um ambiente de desenvolvimento integrado bastante simples, que acompanha a instalação do Python. Vamos usá-lo para escrever o código dos nossos programas.

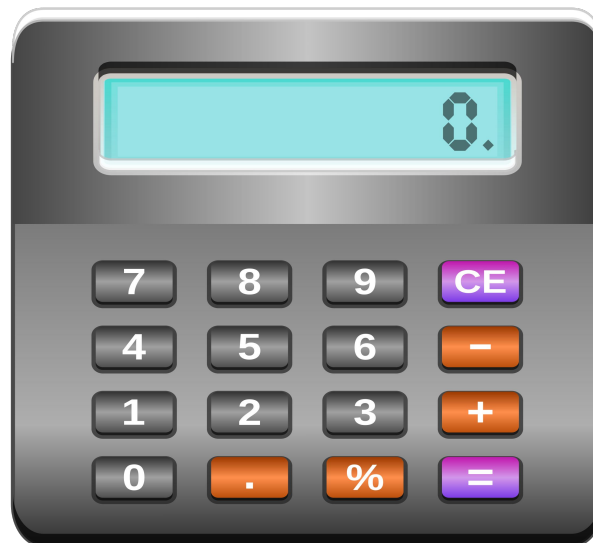


```
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>> |
```

Ln: 5 Col: 4

Exercício: Usando Python como calculadora

1. Quantos segundos há em **70 minutos e 42 segundos**?
2. Converta a seguinte expressão em código Python:
 $(5^4 + 3) \times 7 - 2$
3. Qual o **resto** da divisão de **3964** por **14**?



Operações matemáticas

Operador	Operação	Exemplo	Resultado
**	Potenciação	$2 ** 3$	8
%	Módulo/Resto	$22 \% 8$	6
//	Divisão de inteiros	$22 // 8$	2
/	Divisão	$22 / 8$	2.75
*	Multiplicação	$3 * 5$	15
-	Subtração	$5 - 2$	3
+	Soma	$2 + 2$	4

Guardando dados na memória com variáveis

Uma variável é um 'apelido' que damos para a referência a um valor na memória do computador. Imagine que podemos colar uma etiqueta para nos lembrar o que significa um determinado valor.



Os dados têm tipos

- Números inteiros → **int**
 - 12
 - 86400
- Números decimais → **float**
 - 0.33333
 - 3.14159
- Valores lógicos → **bool**
 - True
 - False
- Textos → **str**
 - 'Maria'
 - 'Python é divertido!'

O comando **type()** retorna o tipo do dado:

```
>>> type('Python é divertido')  
<class 'str'>  
>>> type(3.14159)  
<class 'float'>
```

Atribuição de variáveis

O tipo é definido no momento da atribuição de acordo com o valor atribuído (tipagem dinâmica). Uma instrução de atribuição cria uma nova variável e atribui um valor a ela.

O operador de atribuição é o sinal =

```
pi = 3.141592653589793  
filme = 'Monty Python e o Cálice Sagrado'  
sabe_programar = True
```

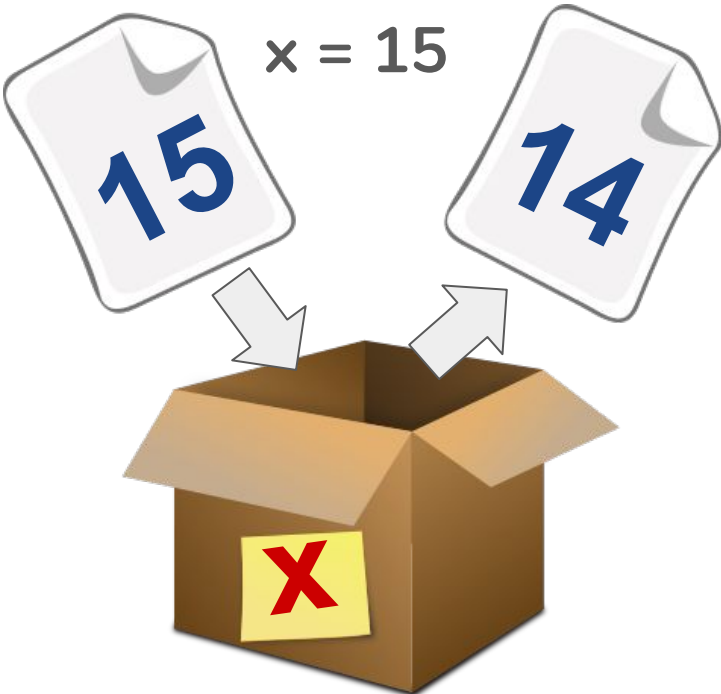
$$x = 7$$

$$x = 14$$

$$y = x$$



$$x = 15$$



Nomeando variáveis

Os nomes de variáveis em Python precisam obedecer algumas regras para serem considerados válidos pelo interpretador:

- Podem ter letras e números
- Não podem iniciar por um número
- O único caractere especial permitido é o *underscore* _
- Não pode coincidir com alguma **palavra reservada** do Python



Palavras reservadas

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
<i>class</i>	False	in	pass	yield
continue	finally	is	raise	
<i>def</i>	for	<i>lambda</i>	return	

Imprimindo dados na tela

A função **print()** mostra na tela o valor entre parênteses.

```
>>> print('Hello world!')  
Hello world!
```

```
>>> print(31 + 11)  
42
```



Lendo entrada de dados do usuário

A função **input()** espera até que o usuário digite algum texto no teclado e pressione ENTER.

```
>>> filme = input()
```

```
>>> print(filme)
```

```
Batman
```



Convertendo tipos

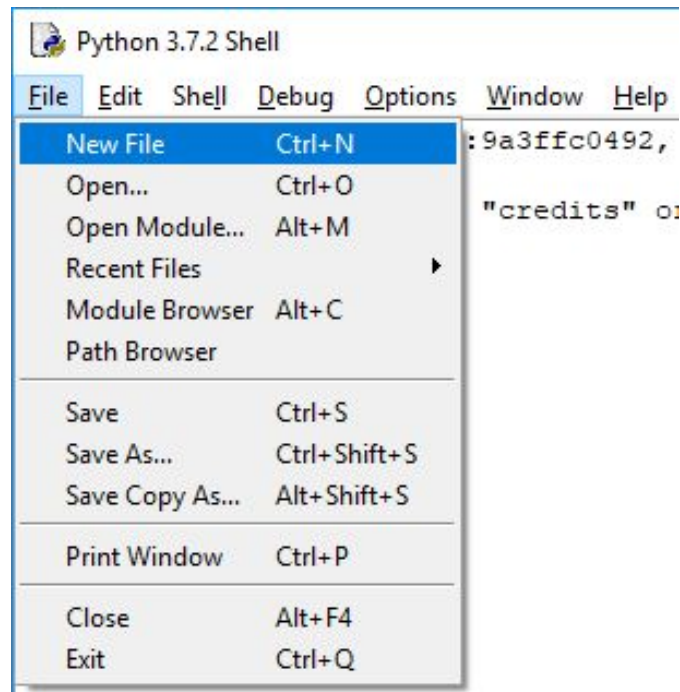
A função **input()** sempre retorna valores do tipo string.

Quando queremos que o valor informado pelo usuário seja usado como número, precisamos convertê-lo para o tipo correspondente.

As funções **int()**, **float()**, **bool()** e **str()** convertem para inteiros, decimais, booleanos e strings, respectivamente.

Executando programas

Podemos usar um editor de texto para escrever o código-fonte dos nossos programas. Os arquivos de código Python têm a extensão **.py**. Vamos usar o IDLE para criar e executar o nosso primeiro programa. Selecione a opção de menu **File -> New Window**:



Exercício: Variáveis e referências

Escreva um programa que execute o trecho de código abaixo e, em seguida, imprima o **valor** e o **tipo** das variáveis **x**, **y** e **z** ao final da execução. Salve com o nome de **variaveis.py**.

```
x = 5
y = 7.5
z = x
s = 'exemplo'
x = y + 5
y = z - 1
z = x * 2
t = s
s = y // 2
```

Comentando o código

Um comentário é uma maneira de fazer que um ou mais trechos do código-fonte não sejam executados. Sua principal utilidade é documentar o que você está fazendo em seu código.

Em Python, tudo o que vem depois de um caractere # até o final da linha é considerado um comentário.

```
# Isto é um comentário antes de um trecho de código  
mensagem = 'Olá, Python!'  
a = 2 + 3 # Isto é um comentário in-line
```


Exercício: Calcular a idade de uma pessoa

Escreva um programa que peça nome e ano de nascimento do usuário, calcule a idade e imprima o resultado no formato abaixo.

'<nome do usuário> tem <idade> anos.'

Salve o programa em um arquivo chamado **idade.py**.

Controlando o fluxo do programa

Podemos escolher a sequência de ações que nossos programas executam a depender do resultado da verificação do valor de uma ou mais variáveis.



Comparação de valores

A comparação entre dois valores retorna um valor do tipo **Boolean (bool)**, ou seja, **True** ou **False**. Os **operadores de comparação** de Python são:

Operador	Significado
==	Igual a
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a



Operadores lógicos (ou booleanos)

A comparação entre valores lógicos (ou *booleanos*) é feita usando os operadores **and**, **or** e **not**, que também retornam **True** ou **False**.

AND

True	True	True
True	False	False
False	True	False
False	False	False

OR

True	True	True
True	False	True
False	True	True
False	False	False

NOT

True	False
False	True



if

A instrução **if** é responsável pela estrutura de decisão que serve para selecionar quando uma parte do programa deve ser executada.

A **expressão booleana** depois do if é chamada de **condição**. Se for verdadeira, o bloco de código correspondente é executado.

```
a = 2
b = 7
if a < b:
    print('a é menor que b.')
```



Indentação e espaços em branco

Em Python os blocos de código são definidos pela sua indentação em relação ao início da linha.

Não existem delimitadores de bloco como `{ ... }` ou `begin ... end`.

O padrão mais aceito é usar **4 espaços** para indentação.

Também é possível usar **Tab** ou qualquer outra quantidade de espaços, desde que seja mantida consistente dentro do mesmo bloco.

Exercício: Cálculo do valor de multa

Escreva um programa que pergunte a velocidade do carro de um usuário. Caso ultrapasse 80 km/h, exiba uma mensagem dizendo que o usuário foi multado. Nesse caso, exiba o valor da multa cobrando R\$ 5,00 por km/h acima de 80.

Fonte: MENEZES, Nilo Ney Coutinho. Introdução à Programação com Python. Novatec, 2010.



else

A instrução **else** indica que o trecho de código seguinte deve ser executado quando o resultado do teste feito no **if** for falso (**False**).

```
a = 2
b = 7
if a == b:
    print('a é igual a b.')
else:
    print('a é diferente de b.')
```



elif

A instrução **elif** indica que queremos testar outra condição quando o **if** for falso. É o equivalente a um **else: if** e serve para tornar o código mais fácil de ler quando temos “ifs” aninhados.

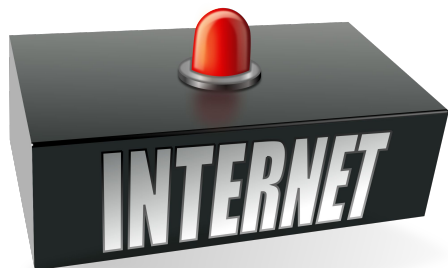
```
if pontos >= 80:
    nota = 'A'
else:
    if pontos >= 65:
        nota = 'B'
    else:
        if pontos >= 50:
            nota = 'C'
        else:
            nota = 'D'
```

```
if pontos >= 80:
    nota = 'A'
elif pontos >= 65:
    nota = 'B'
elif pontos >= 50:
    nota = 'C'
else:
    nota = 'D'
```



Exercício: Provedor de internet

Escreva um programa que calcule o preço a pagar pelo consumo de internet. Pergunte ao usuário qual a **quantidade consumida em GB** e o **tipo de plano**: **R** para residencial ou **E** para empresarial. A tabela de preços é a seguinte:



Tipo de Plano	Faixa de consumo (em GB)	Preço por GB (em R\$)
Residencial	Até 10	8,00
	Acima de 10	12,00
Empresarial	Até 100	10,00
	Acima de 100	15,00



Repetindo a execução de partes do programa

Uma das vantagens do uso de computadores é que eles são capazes de repetir tarefas inúmeras vezes de forma eficiente.

Por isso, em quase todos os programas precisamos executar algum trecho de código mais de uma vez.

Cada repetição de um trecho de código é chamada de **laço** ou **loop**.

while

O comando while repete um bloco de instruções até que uma determinada condição seja verdadeira.

```
contagem = 10
while contagem > 0:
    print(contagem)
    contagem = contagem - 1
print('Decolar!')
```

Modificando o laço de repetição

break

Sai do laço mais próximo que envolve o bloco.

continue

Interrompe a repetição atual e pula para o início da próxima.

pass

Usado quando não queremos que se faça nada (bloco vazio).

else

Executa bloco de código após encerradas todas as repetições.

Exercício: Pedra, papel e tesoura

Faça um jogo de Pedra, Papel e Tesoura para dois jogadores. O programa deve pedir que cada jogador digite sua jogada. Em seguida, deve comparar as jogadas, mostrar uma mensagem de parabéns ao vencedor e perguntar se os jogadores querem começar uma nova partida.

Lembre-se das regras:

- **Pedra** ganha da **tesoura**
- **Tesoura** ganha do **papel**
- **Papel** ganha da **pedra**



Sequências: armazenando múltiplos valores

Muitos problemas que podem ser resolvidos por computadores envolvem guardar múltiplos valores agrupados.

Em Python, as estruturas de dados que tem tamanho determinado e cujos itens podem ser acessados por meio de índices são chamadas de **sequências**.

Os principais tipos de sequências são **listas**, **tuplas** e **strings**.

Operadores de sequências

in

Verifica se um valor está contido em uma sequência (retorna um valor booleano)

+

Concatena duas sequências

Repete a sequência

[i]

Retorna o item da sequência que está na posição *i*

[i:j]

Retorna a fatia da sequência entre as posições *i* e *j*.

[i:j:k]

Retorna a fatia da sequência entre as posições *i* e *j* pulando de *k* em *k* elementos.

len(s)

Retorna o tamanho da sequência *s*.

min(s)

Retorna o menor valor da sequência *s*.

max(s)

Retorna o maior valor da sequência *s*.

s.index(x)

Retorna o índice da primeira ocorrência do valor *x* na sequência *s*.

s.count(x)

Retorna a quantidade de ocorrências de *x* na sequência *s*.

Listas

Uma lista é um valor que corresponde a uma estrutura que contém múltiplos valores em uma **sequência ordenada**.

Por ser um valor, pode ser referenciada por uma variável e utilizada em expressões.

```
>>> animais = ['gato', 'rato', 'cachorro', 'esquilo',  
'cavalo', 'pombo']  
>>> animais[2]  
'cachorro'  
>>>
```

Operadores de listas

```
>>> frutas = ['laranja', 'maçã', 'pera', 'banana', 'kiwi', 'maçã',  
'banana']  
>>> frutas.count('maçã')  
2  
>>> frutas.count('tangerina')  
0  
>>> frutas.index('banana')  
3  
>>> frutas.index('banana', 4)  
6  
>>> frutas.reverse()  
>>> frutas  
['banana', 'maçã', 'kiwi', 'banana', 'pera', 'maçã', 'laranja']
```

Operadores de listas (2)

```
>>> frutas.append('uva')
>>> frutas
['banana', 'maçã', 'kiwi', 'banana', 'pera', 'maçã', 'laranja',
'uva']
>>> frutas.sort()
>>> frutas
['banana', 'banana', 'kiwi', 'laranja', 'maçã', 'maçã', 'pera',
'uva']
>>> frutas.pop()
'uva'
>>> frutas
['banana', 'banana', 'kiwi', 'laranja', 'maçã', 'maçã', 'pera']
>>>
```

Listas são objetos mutáveis!

Qual o valor de b?

```
>>> a = ['Python', 'Java', 'C#']
>>> b = a
>>> a.append('JavaScript')
>>> print(b)
```

Para fazer cópias de listas:

```
>>> a = ['Python', 'Java', 'C#']
>>> b = a[:]
>>> a.append('JavaScript')
>>> print(b)
['Python', 'Java', 'C#']
>>> print(a)
['Python', 'Java', 'C#', 'JavaScript']
```

for

O comando **for** serve para **percorrer uma sequência** de itens.

Para cada item percorrido, um bloco de código é executado.

```
>>> animais = ['gato', 'rato', 'cachorro', 'esquilo', 'cavalo',  
'pombo']  
>>> for animal in animais:  
    print(animal)
```

```
gato  
rato  
cachorro  
esquilo  
cavalo  
pombo  
>>>
```

range()

A função **range()** serve para gerar sequências de números.

```
>>> for i in range(5):  
    print(i * 2)
```

0

2

4

6

8

```
>>>
```

```
>>> r = range(25, 0, -5)  
>>> for i in r:  
    print(i, "/ 2 =", i / 2)
```

25 / 2 = 12.5

20 / 2 = 10.0

15 / 2 = 7.5

10 / 2 = 5.0

5 / 2 = 2.5

```
>>>
```

Exercício: Palavras no plural

Dada a lista abaixo, escreva um programa que gere uma nova lista em que as palavras que estão nas **posições ímpares** fiquem no **plural**. Salve o arquivo com o nome **plural.py**.

```
palavras = ["carro", "melancia", "balde",  
            "navio", "gato", "floresta"]
```



List comprehensions: fazendo mágica com apenas uma linha

Em Python é possível gerar uma nova lista a partir da aplicação do resultado de uma expressão para cada elemento da lista original usando apenas uma linha de código.

```
>>> numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> dobro = [i * 2 for i in numeros]
>>> dobro
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> dobro_pares = [i * 2 for i in numeros if i % 2 == 0]
>>> dobro_pares
[4, 8, 12, 16, 20]
```



Tuplas

São semelhantes a listas, porém são **imutáveis**. Entre outras coisas, isso quer dizer que não podemos inserir, remover ou atribuir novos valores a itens de tuplas.

```
>>> filme = ('O Poderoso Chefão', 1972, 'Francis Ford Coppola')
>>> lancamento = filme[1]
>>> lancamento
1972
>>> filme[1] = 1973
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    filme[1] = 1973
TypeError: 'tuple' object does not support item assignment
>>>
```

Convertendo listas x tuplas

As funções `list()` e `tuple()` servem para converter sequências para listas e tuplas, respectivamente.

```
>>> filme = ('O Poderoso Chefão', 1972, 'Francis Ford Coppola')
>>> type(filme)
<class 'tuple'>
>>> filme[1] = 1973
Traceback (most recent call last):
  File "<pyshell#91>", line 1, in <module>
    filme[1] = 1973
TypeError: 'tuple' object does not support item assignment
>>> filme = list(filme)
>>> filme[1] = 1973
>>> filme
['O Poderoso Chefão', 1973, 'Francis Ford Coppola']
>>>
```

Strings

As strings também são sequências **imutáveis** na qual **cada item é um caracter**.

Os mesmos operadores comuns a sequências também funcionam com strings. Isso quer dizer que podemos referenciar cada caracter pelo índice, fatiar strings, obter tamanho, contar o número de ocorrências de caracteres etc.

"Python"



'P'	'y'	't'	'h'	'o'	'n'
0	1	2	3	4	5

Strings

"Python"



'P'	'y'	't'	'h'	'o'	'n'
0	1	2	3	4	5

```
>>> s = "Python"
>>> len(s)
6
>>> s.count('y')
1
>>> s[1:3]
'yt'
>>> s.index('o')
4
```

```
>>> s + " é super legal!"
'Python é super legal!'
>>> s * 3
'PythonPythonPython'
>>>
```

Mais truques com strings

- `split()`, `join()`
- `upper()`, `lower()`
- `startswith()`, `endswith()`
- `find()`, `replace()`
- `strip()`

Formatação de strings

format()

Strings de formatação incluem campos de substituição delimitados por chaves `{}`. Qualquer coisa que não estiver entre chaves é considerada texto literal.

```
>>> "{} marcou {} pontos na partida.".format("LeBron James", 32)
'LeBron James marcou 32 pontos na partida.'
>>> "Hoje é {2} de {1} de {0}".format(2019, "abril", 24)
'Hoje é 24 de abril de 2019'
>>> artista = "Freddie Mercury"
>>> musica = "We Are The Champions"
>>> banda = "Queen"
>>> 'O vocalista {a} da banda {b} gravou "{m}"'.format(m=musica,
a=artista, b=banda)
'O vocalista Freddie Mercury da banda Queen gravou "We Are The
Champions"'
```

Exercício: Strings

1. Escreva um programa que peça uma frase e a imprima, em minúsculas, substituindo os espaços por hífen (-).
2. Escreva um programa que peça o nome do usuário e o imprima de trás para frente em letras maiúsculas.

Dicionários

Em Python, um **dicionário** é uma estrutura de dados na qual os items são armazenados e localizados por uma **chave**, em vez de índices por deslocamento posicional.

Em outras linguagens: *maps, hashmaps, arrays associativos...*

```
>>> contatos = {}
>>> contatos['Alice'] = '3232-3232'
>>> contatos['Bob'] = '3434-3434'
>>> contatos['Carol'] = '3535-3535'
>>> contatos
{'Alice': '3232-3232', 'Bob': '3434-3434', 'Carol': '3535-3535'}
>>> contatos['Bob']
'3434-3434'
```


Dicionários: alguns métodos úteis

- **d.keys()**
 - Retorna lista com as chaves do dicionário
- **d.values()**
 - Retorna lista com os valores do dicionário
- **d.items()**
 - Retorna lista das tuplas de chave e valor do dicionário
- **d.get(k, valor_padrao)**
 - Retorna o valor correspondente à chave k ou o valor_padrao caso a chave não exista no dicionário
- **d.setdefault(k, valor_padrao)**
 - Atribui valor_padrao à chave k caso ela não exista no dicionário

Exercício: Contando ocorrências de caracteres

Escreva um programa que leia uma frase informada pelo usuário e conte a quantidade de ocorrências de cada caractere, armazenando a contagem em um dicionário.



Funções: Melhorando a estrutura do código

Geralmente precisamos reutilizar o mesmo trecho de código com dados diferentes. Para isso, podemos usar o conceito de **funções**.

Funções são como mini-programas dentro do seu programa. Elas podem receber dados de **entrada** e retornar dados de **saída**.

```
def gorjeta(valor):  
    resultado = (valor * 10) / 100  
    return resultado
```

PALAVRA-CHAVE QUE
INDICA DEFINIÇÃO DE
FUNÇÃO

NOME DA FUNÇÃO

ARGUMENTOS
(ENTRADAS)

def **faz_algo** **(a, b, c):**

linha 1

...

linha n

return resultado

PALAVRA-CHAVE QUE
INDICA O RESULTADO
DA FUNÇÃO

VALOR DE
RETORNO (SAÍDA)

Exercício: Cálculo da média

Escreva uma função chamada **media()** que receba uma lista de números e retorne a média dos seus valores.

Salve o código em um arquivo chamado **util.py**.



Exercício: Estruturando dados

Escreva uma função chamada `gerar_registro()` que receba uma string no formato

"Jennifer;Aracaju;25;3500"

e retorne um dicionário com a seguinte estrutura

```
{nome: 'Jennifer', cidade: 'Aracaju', idade: 25, renda: 3500}
```

Salve o código no mesmo arquivo **util.py** do exercício anterior.



Arquivos

Para abrir um arquivo, usamos a função **open(nome_arquivo, modo_acesso)**:

```
estados = open('nordeste.txt', 'r')
```

Os modos de acesso podem ser **'r'** (leitura), **'w'** (escrita), **'a'** (adicionar ao final) e **'b'** (binário).

Operações em arquivos:

- **file.read(), readline(), readlines(),**
- **file.write(), writelines(),**
- **file.close()**

Arquivos

Leitura:

```
>>> arquivo = open('nordeste.txt', 'r')
>>> estados = arquivo.readlines()
>>> estados
['Alagoas\n', 'Bahia\n', 'Ceará\n', 'Maranhão\n', 'Paraíba\n',
'Pernambuco\n', 'Piauí\n', 'Rio Grande do Norte\n', 'Sergipe\n']
>>> arquivo.close()
```

Escrita:

```
>>> arquivo = open('sudeste.txt', 'w')
>>> estados = ['Espírito Santo\n', 'Minas Gerais\n', 'Rio de Janeiro\n',
'São Paulo\n']
>>> arquivo.writelines(estados)
>>> arquivo.close()
```


Módulos

Cada arquivo com extensão **.py** que criamos até aqui pode ser chamado de **módulo**. Módulos são usados para agrupar definições de variáveis, constantes, funções ou classes que podem ser reutilizadas por outros programas.

O comando **import** serve para fazer com que o conteúdo de um módulo se torne disponível no programa que estamos escrevendo.

```
>>> import util
>>> util.media([1, 2, 3, 4])
2.5
>>> from util import media
>>> media([5, 6, 7, 8])
6.5
```

Exercício: Estatísticas de clientes

Uma empresa quer manter um cadastro dos seus clientes para obter estatísticas sobre eles. Crie um arquivo texto chamado **clientes.txt** com o conteúdo abaixo. Em seguida crie um módulo **clientes.py** com um programa que importe o módulo **util.py** dos exercícios anteriores, leia o arquivo de dados e mostre na tela as médias de idade e de renda mensal dos clientes.

Jennifer;Aracaju;25;3500

Pedro;Aracaju;32;4200

Carlos;Salvador;28;2900

Maria;Recife;41;5000



Baterias incluídas: a biblioteca padrão do Python

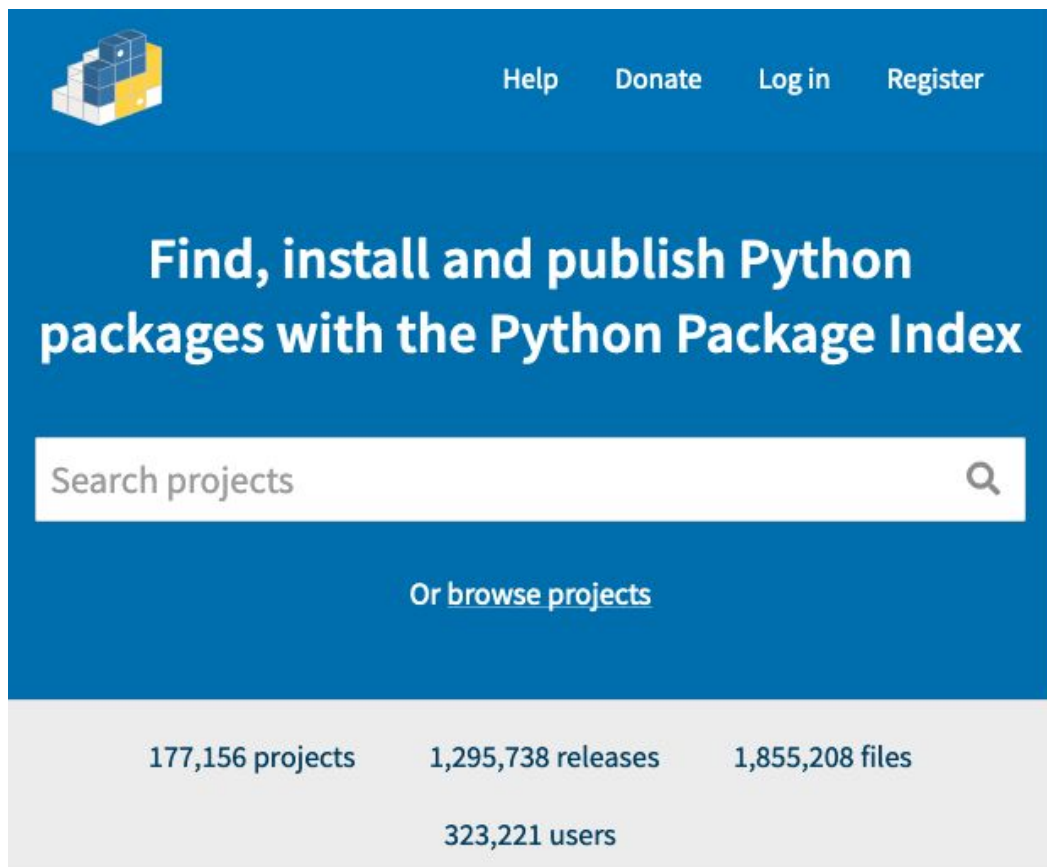
Um dos maiores diferenciais de Python é a sua vasta biblioteca padrão (**standard library**), que contém módulos para as mais diversas finalidades. Alguns módulos úteis são:

- **sys:** version, getsizeof(), argv
- **os:** getcwd()
- **datetime:** date.today(), timedelta()
- **math:** pi, sqrt(), sin(), cos(), factorial()
- **random:** random(), randint(), shuffle(), choice()
- **itertools:** accumulate(), combinations(), product(), permutations()
- **collections:** Counter(), OrderedDict(), deque()

Bibliotecas de terceiros: ampliando o poder de Python com open source

O repositório oficial de pacotes de terceiros é o **PyPI (Python Package Index)**

- Django
- Flask
- Requests
- BeautifulSoup
- Numpy
- Pandas
- Scikit-learn



Programação Orientada a Objetos

Python mistura diferentes estilos de programação (procedural, orientado a objetos, funcional etc.)

No entanto, quando falamos dos fundamentos de sua estrutura, Python é orientada a objetos. Em Python tudo é um objeto.

Um objeto é uma coleção de dados e instruções mantidas em memória, que consiste em:

- Tipo
- Identificador único
- Dados
- Métodos

Classes

Uma **classe** é a definição de um novo tipo de dado que agrupa dados e operações em uma única estrutura. Para representar um tocador de música simples em Python, podemos fazer algo como:

```
class Player:  
    def __init__(self):  
        self.estado = 'parado'  
        self.volume = 5  
        self.musica = ''  
  
    def play(self):  
        self.estado = 'tocando'  
  
    def stop(self):  
        self.estado = 'parado'
```

```
    def pause(self):  
        if self.estado == 'pausado':  
            self.estado = 'tocando'  
        elif self.estado == 'tocando':  
            self.estado = 'pausado'  
  
    def aumentar_volume(self):  
        self.volume += 1  
  
    def diminuir_volume(self):  
        self.volume -= 1
```

Obrigado!

Dúvidas?

Perguntas?

Referências

1. MENEZES, Nilo Ney Coutinho. Introdução à Programação com Python. 2a ed. Novatec, 2019.
2. SWEIGART, Al. Automate the Boring Stuff with Python. No Starch Press, 2015. Disponível em <<http://automatetheboringstuff.com/>>.
3. DRISCOLL, Mike. Python 101. Leanpub, 2014.
4. DOWNEY, Allen B. Pense em Python. Novatec, 2016.
5. Python Software Foundation. The Python Tutorial. Version 3.7.3. Disponível em <<https://docs.python.org/3/tutorial/>>.
6. SARGENT, Thomas J. e STACHURSKI, John. Quantitative Economics with Python. Disponível em <<https://lectures.quantecon.org/py/>>.
7. Python Basics <<https://pythonbasics.org>>