



O Manual Python

Índice

- [Índice](#)
- [1. Introdução ao Python](#)
- [2. Instalando o Python](#)
- [3. Executando programas Python](#)
- [4. Python 2 vs Python 3](#)
- [5. Os conceitos básicos de trabalhar com Python](#)
 - [5.1. Variável](#)
 - [5.2. Expressões e declaração](#)
 - [5.3. Comentar](#)
 - [5.4. Recuo](#)
- [6. Tipos de dados](#)
 - [7. Operadores](#)
 - [7.0.1. Operador de atribuição](#)
 - [7.0.2. Operador aritmético](#)
 - [7.0.3. Operador de comparação](#)
 - [7.0.4. Operador booleano](#)
 - [7.0.5. Operador bit a bit](#)
 - [7.0.6. é e em](#)
 - [8. O Operador Ternário](#)
 - [9. Cordas](#)
 - [10. Booleanos](#)
 - [11. Números](#)
 - [11.0.1. Número inteiro](#)
 - [11.0.2. Número do ponto flutuante](#)
 - [11.0.3. Número complexo](#)



The banner features the Auth0 logo (a blue hexagon with a white star) and the text "auth0 by Okta". Below the logo, there is a promotional message: "Implement Auth0 in any app in just 5 minutes. Start building today." At the bottom right of the banner, it says "ADS VIA CARBON".

- [11.0.4. Operações aritméticas em número](#)
- [11.0.5. Função incorporada](#)
- [12. Constantes](#)
- [13. Enums](#)
- [14. Entrada do usuário](#)
- [15. Instruções de controle](#)
- [16. Listas](#)
- [17. Tuplas](#)
- [18. Dicionários](#)
- [19. Conjuntos](#)
- [20. Funções](#)
- [21. Objetos](#)
- [22. Loops](#)
 - [22.1. enquanto loop](#)
 - [22.2. para loop](#)
 - [22.3. Interromper e continuar](#)
- [23. Aulas](#)
- [24. Módulos](#)
- [25. A Biblioteca Padrão Python](#)
- [26. O guia de estilo do Python PEP8](#)
- [27. Depuração](#)
- [28. Âmbito das variáveis](#)
- [29. Aceite argumentos da linha de comando](#)
- [30. Funções do Lambda](#)
- [31. Recursão](#)
- [32. Funções aninhadas](#)
- [33. Encerramentos](#)
- [34. Decoradores](#)
- [35. Docstrings](#)
- [36. Introspecção](#)
- [37. Anotações](#)
- [38. Exceções](#)
- [39. A declaração com](#)
- [40. Instalando pacotes 3rd party usando pip](#)
- [41. Listar comprehensões](#)
- [42. Polimorfismo](#)
- [43. Sobrecarga do operador](#)

- 44. Ambientes Virtuais

1. Introdução ao Python

Python está literalmente comendo o mundo da programação. Está crescendo em popularidade e uso de maneiras que são praticamente sem precedentes na história dos computadores.

Há uma enorme variedade de cenários em que o Python se destaca. **Shell scripting, automação de tarefas, desenvolvimento Web** são apenas alguns exemplos básicos.

Python é a linguagem de escolha para **análise de dados e aprendizado de máquina**, mas também pode se adaptar para criar jogos e trabalhar com dispositivos embarcados.

Mais importante ainda, é a linguagem de escolha para cursos introdutórios de **ciência da computação** em universidades de todo o mundo.

Muitos alunos aprendem Python como sua primeira linguagem de programação. Muitos estão aprendendo agora, muitos aprenderão no futuro. E para muitos deles, Python será a única linguagem de programação que eles precisam.

Graças a essa posição única, o Python provavelmente crescerá ainda mais no futuro.

A linguagem é simples, expressiva e bastante direta.

O ecossistema é enorme. Parece haver uma biblioteca para tudo o que você pode imaginar.

Python é uma linguagem de programação de alto nível adequada para iniciantes graças à sua sintaxe intuitiva, sua enorme comunidade e ecossistema vibrante.

Também é apreciado por profissionais em muitos campos diferentes.

Tecnicamente falando, é uma linguagem interpretada que não tem uma fase de compilação intermediária como uma linguagem compilada, por exemplo, C ou Java.

E, como muitas linguagens interpretadas, ele é digitado dinamicamente, o que significa que você não precisa indicar os tipos das variáveis que você usa, e as variáveis não estão vinculadas a um tipo específico.

Isso tem prós e contras. Em particular, podemos mencionar que você escreve programas mais rapidamente, mas, por outro lado, você tem menos ajuda das ferramentas para evitar possíveis bugs e você descobrirá sobre alguns tipos de problemas apenas executando o programa em tempo de execução.

Python suporta uma ampla variedade de diferentes paradigmas de programação, incluindo programação processual, programação orientada a objetos e programação funcional. É flexível o suficiente para se adaptar a muitas necessidades diferentes.

Criado em 1991 por Guido van Rossum, tem vindo a aumentar em popularidade - especialmente nos últimos 5 anos, como mostra este infográfico do Google Trends:

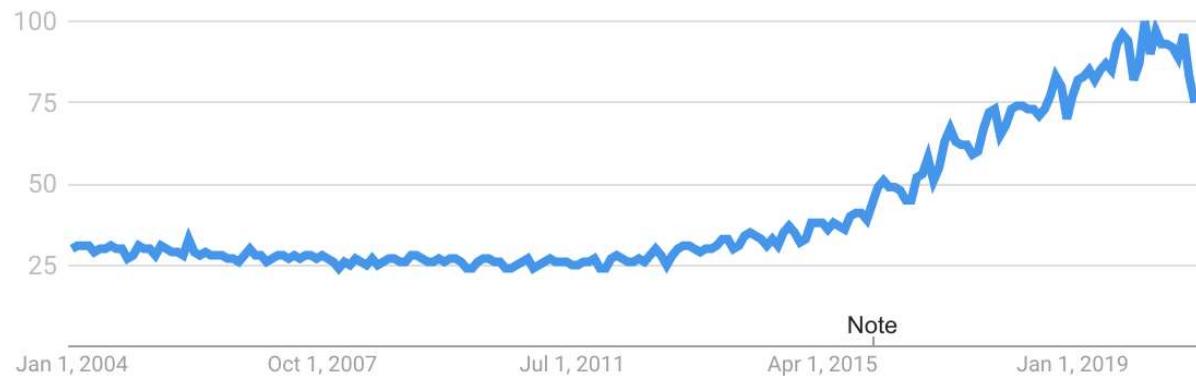
- Python
Programming language



+ COMPARE

— Worldwide, 2004 - present

Interest over time



Começar com Python é muito fácil. Tudo o que você precisa é instalar o pacote oficial do [python.org](https://www.python.org), para Windows, macOS ou Linux, e você está pronto para ir.

Se você é novo em programação, nos próximos posts vou orientá-lo a ir do zero para se tornar um programador Python.

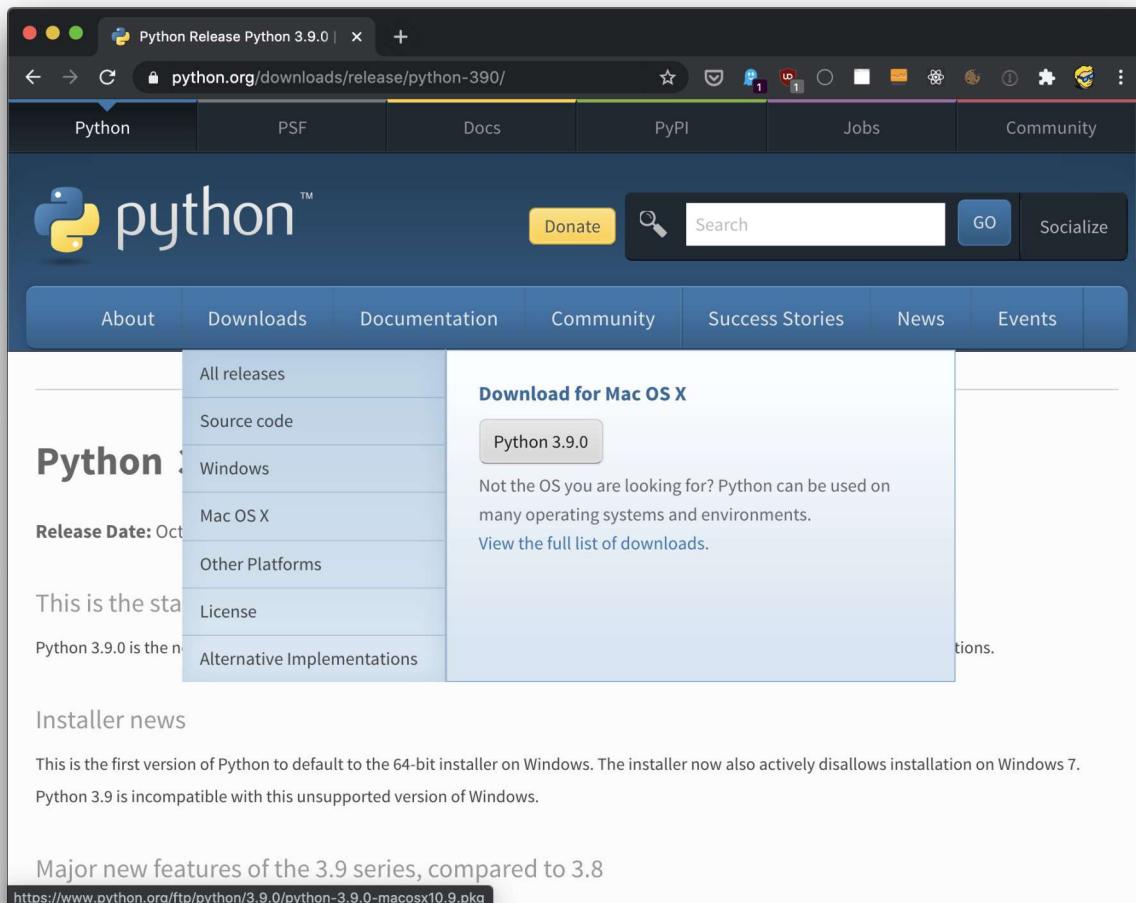
E mesmo que você seja atualmente um programador especializado em outra linguagem, Python é uma linguagem que vale a pena conhecer, porque acho que estamos apenas no começo.

Linguagens de nível mais baixo, como C++ e Rust, podem ser ótimas para programadores especializados, mas assustadoras para começar, e levam muito tempo para dominar. Python, por outro lado, é uma linguagem de programação para programadores, é claro, mas também para os não-programadores. Os estudantes, as pessoas que fazem o seu trabalho diário com o Excel, os cientistas.

A linguagem que todos os interessados em codificação devem aprender primeiro.

2. Instalando o Python

Vá para <https://www.python.org>, escolha o menu Downloads, escolha seu sistema operacional e um painel com um link para baixar o pacote oficial aparecerá:



Certifique-se de que segue as instruções específicas para o seu sistema operativo. No macOS, você pode encontrar um guia detalhado sobre <https://flaviocopes.com/python-installation-macos/>.

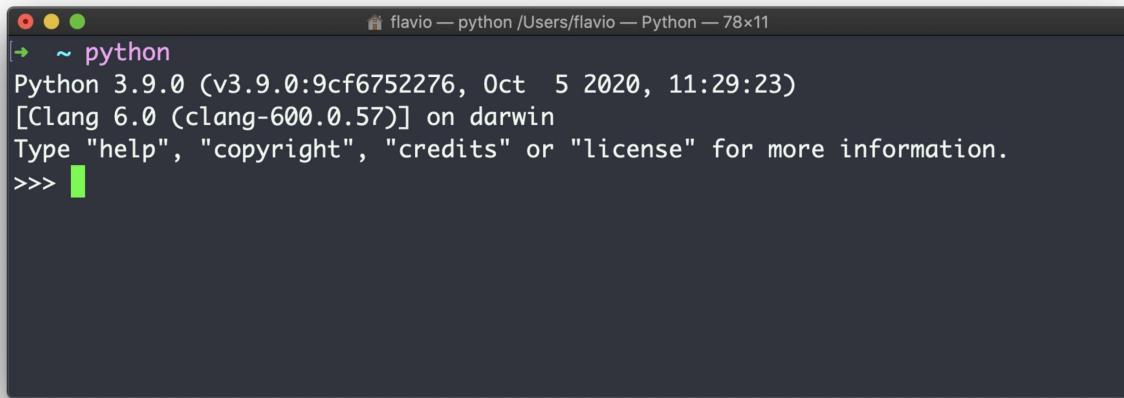
3. Executando programas Python

Existem algumas maneiras diferentes de executar programas Python.

Em particular, você tem uma distinção entre usar prompts interativos, onde você digita o código Python e ele é executado imediatamente, e salvar um programa Python em um arquivo e executá-lo.

Vamos começar com prompts interativos.

Se você abrir seu terminal e digitar , você verá uma tela como esta:python



```
[~] flavio — python /Users/flavio — Python — 78x11
[~] ~ python
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Este é o Python REPL (Read-Evaluate-Print-Loop)

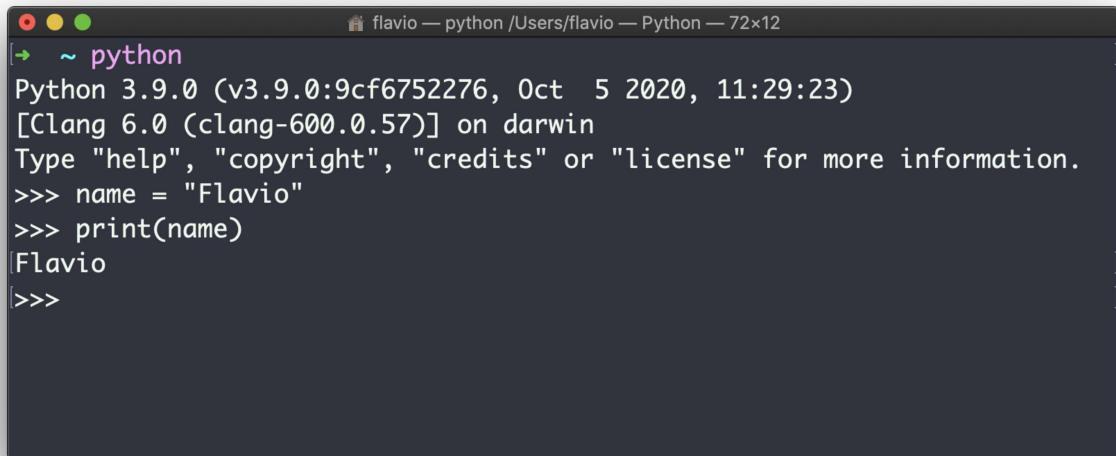
Observe o símbolo e o cursor depois disso. Você pode digitar qualquer código Python aqui e pressionar a tecla para executá-lo.>>>enter

Por exemplo, tente definir uma nova variável usando

```
name = "Flavio"
```

e, em seguida, imprima seu valor, usando:`print()`

```
print(name)
```



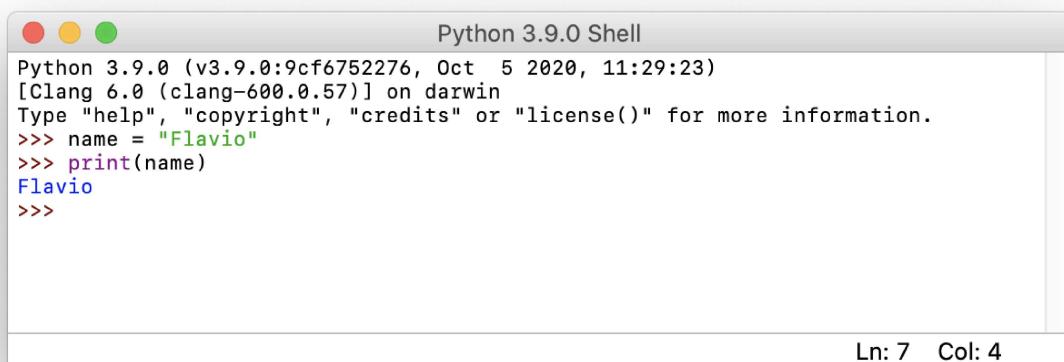
```
flavio — python /Users/flavio — Python — 72x12
[~] ~ python
Python 3.9.0 (v3.9.0:9cf6752276, Oct 5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> name = "Flavio"
>>> print(name)
Flavio
>>>
```

Nota: no REPL, você também pode simplesmente digitar , pressionar a tecla e você receberá o valor de volta. Mas em um programa, você não vai ver nenhuma saída se você fizer isso - você precisa usar em vez disso `name.print()`

Qualquer linha de Python que você escrever aqui será executada imediatamente.

Digite para sair deste Python REPL.`quit()`

Você pode acessar o mesmo prompt interativo usando o aplicativo IDLE que é instalado pelo Python automaticamente:



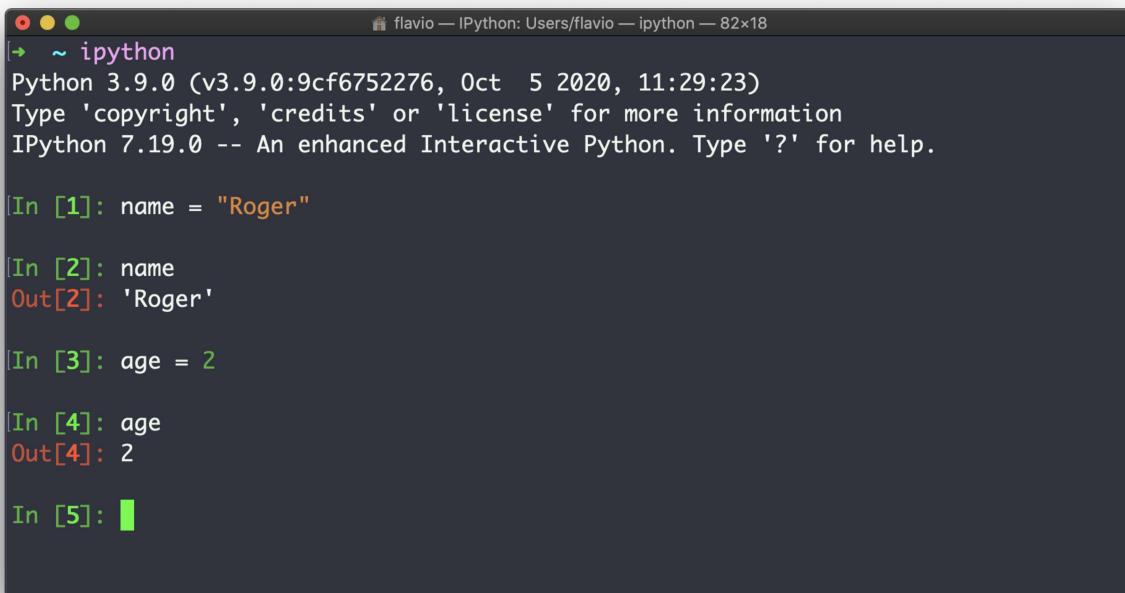
Isso pode ser mais conveniente para você, porque com o mouse você pode se mover e copiar / colar mais facilmente do que com o terminal.

Esses são os conceitos básicos que vêm com o Python por padrão. No entanto, eu recomendo instalar o IPython, provavelmente o melhor aplicativo REPL de linha de comando que você pode encontrar.

Instale-o com

```
pip install ipython
```

Verifique se os binários pip estão no caminho e execute:`ipython`



The screenshot shows a terminal window titled "flavio — IPython: Users/flavio — ipython — 82x18". It displays the following IPython session:

```
[~ ipython
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: name = "Roger"
Out[1]: 'Roger'

In [2]: name
Out[2]: 'Roger'

In [3]: age = 2
Out[3]: 2

In [4]: age
Out[4]: 2

In [5]:
```

`ipython` é outra interface para trabalhar com um Python REPL, e fornece alguns recursos interessantes, como realce de sintaxe, conclusão de código e muito mais.

A segunda maneira de executar um programa Python é escrever o código do programa Python em um arquivo, por exemplo:`program.py`

```
python — nano /Users/flavio/dev/python — nano program.py — 76x10
GNU nano 2.0.6          File: program.py          Modified

name = "Flavio"
print(name)

[ Read 2 lines ]
^G Get Help ^O WriteOut ^R Read File^Y Prev Page^K Cut Text ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is ^V Next Page^U UnCut Tex^T To Spell
```

e, em seguida, execute-o com `python program.py`

```
python — fish /Users/flavio/dev/python — -fish — 76x10
↳ python python program.py
Flavio
↳ python
```

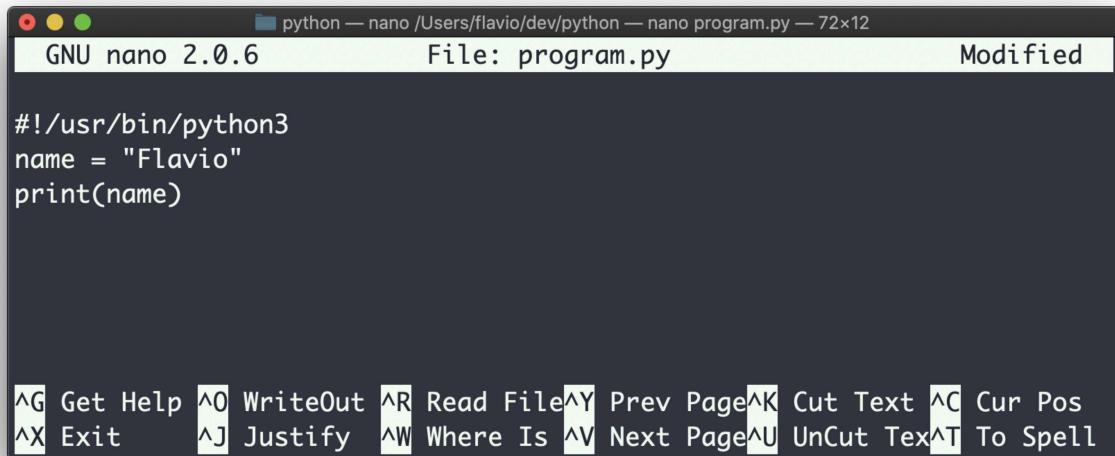
Note que salvamos programas Python com a extensão, isso é uma convenção..py

Neste caso, o programa é executado como um todo, não uma linha de cada vez. E é tipicamente assim que executamos programas.

Usamos o REPL para prototipagem rápida e para aprendizado.

No Linux e macOS, um programa Python também pode ser transformado em um shell script, substituindo todo o seu conteúdo com uma linha especial que indica qual executável usar para executá-lo.

No meu sistema, o executável Python está localizado em , então eu digito na primeira linha:`/usr/bin/python3#!/usr/bin/python3`



```
#!/usr/bin/python3
name = "Flavio"
print(name)
```

Em seguida, posso definir a permissão de execução no arquivo:

```
chmod u+x program.py
```

e eu posso executar o programa com

```
./program.py
```

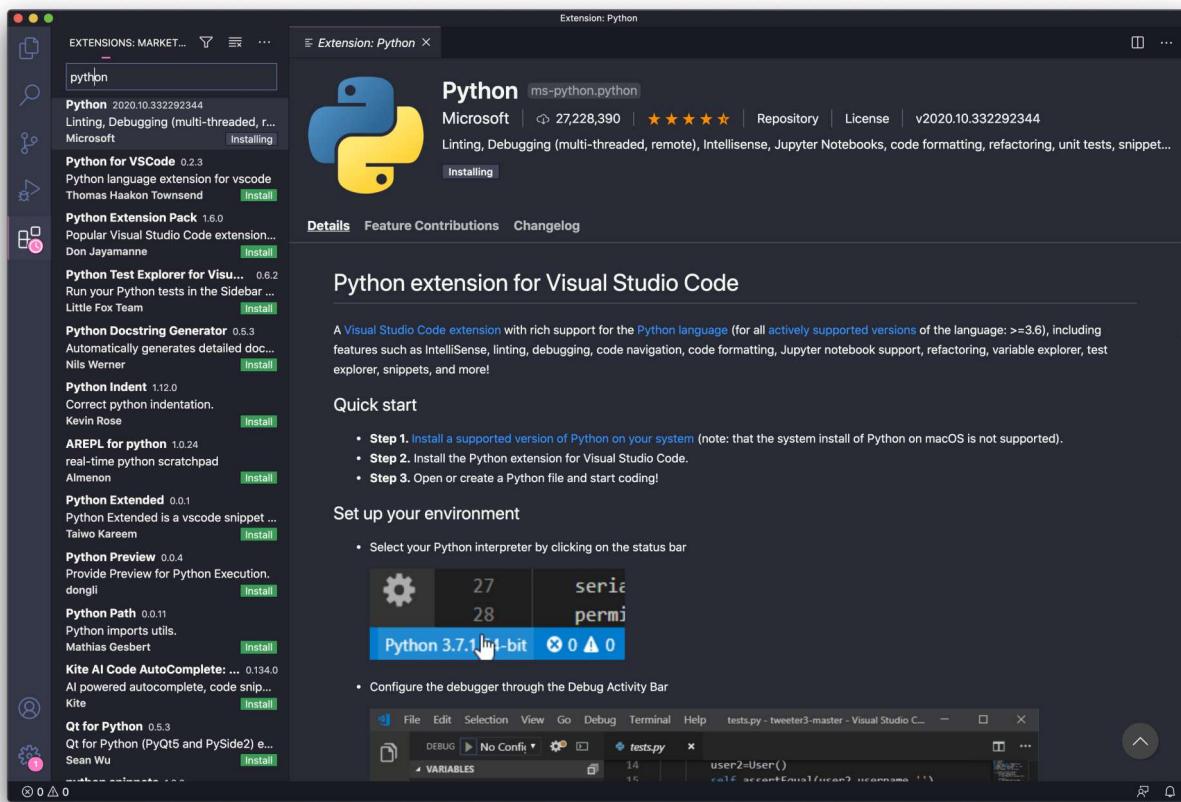


```
[→ python ./program.py
Flavio
→ python ]
```

Isso é especialmente útil quando você escreve scripts que interagem com o terminal.

Temos muitas outras maneiras de executar programas Python.

Um deles é usar o VS Code e, em particular, a extensão oficial Python da Microsoft:



Depois de instalar esta extensão, você terá auto-completar de código Python e verificação de erros, formatação automática e alinhamento de código com , e alguns comandos especiais, incluindo:pylint

Python: Inicie o REPL para executar o REPL no terminal integrado:

The screenshot shows a dark-themed code editor window. At the top, there's a toolbar with icons for file operations, search, and other functions. Below the toolbar, the title bar displays 'program.py X' and 'Python - Get Started'. The main area has tabs for 'OUTPUT', 'TERMINAL', 'DEBUG CONSOLE', and 'PROBLEMS'. The 'TERMINAL' tab is currently selected, showing the following text:

```
Welcome to fish, the friendly interactive shell
Type `help` for instructions on how to use fish
> ~ /usr/local/bin/python3
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the bottom of the editor, it says 'Python 3.9.0 64-bit' and shows status information: 'Ln 3, Col 1', 'Spaces: 2', 'UTF-8', 'LF', 'Python', and some small icons.

Python: **Execute o arquivo Python no Terminal** para executar o arquivo atual no terminal:

This screenshot is identical to the one above, showing the same code editor interface and terminal output. The terminal window displays the execution of 'program.py' and the resulting output 'Flavio'.

Python: **Execute o arquivo atual na janela interativa do Python:**

A screenshot of the Visual Studio Code (VS Code) interface. On the left is a dark-themed sidebar with icons for file operations like Open, Save, Find, and Run. The main area shows a Python file named "program.py" with the following code:

```
1 name = "Flavio"
2 print(name)
3 
```

To the right of the editor is a "Python Interactive - #1" terminal window displaying the output of the code execution:

```
x Flavio
```

Below the editor, the status bar shows "Python 3.9.0 64-bit" and other details like line and column numbers.

e muitos mais. Basta abrir a paleta de comandos (View -> Command Palette, ou Cmd-Shift-P) e digitar para ver todos os comandos relacionados ao Python:python

A screenshot of the Visual Studio Code interface showing the Command Palette open. The search bar at the top contains the text ">python". A list of Python-related commands is displayed:

- Python: Open Start Page
- Python: Reset Stored Info for Untrusted Interpreters
- Python: Restart Language Server
- Python: Run All Tests
- Python: Run Current Test File
- Python: Run Failed Tests
- Python: Run Linting
- Python: Run Python File in Terminal python.execInTerminal-icon

The rest of the interface is similar to the previous screenshot, showing the "program.py" file and its output in the terminal.

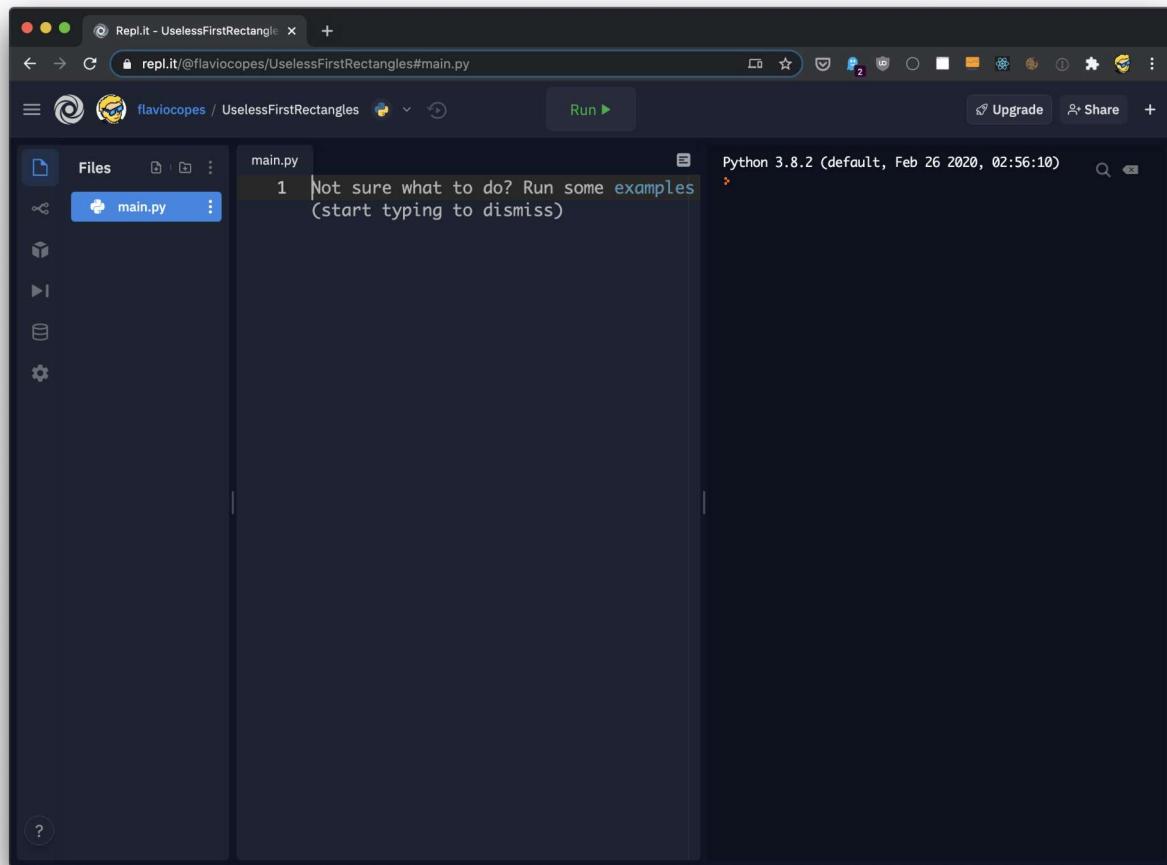
Another way to easily run Python code is to use repl.it, a very nice website that provides a coding environment you can create and run your apps on, in any language, Python included:

The screenshot shows the Repl.it homepage with a dark blue background. At the top, there's a navigation bar with links for 'Features', 'Jobs', 'Blog', 'Pricing', and 'Jam'. On the right side of the bar are 'Log in' and 'Sign up' buttons. Below the navigation, the main heading reads 'Code, create, and learn together' in large white font. A subtext below it says 'Use our free, collaborative, in-browser IDE to code in 50+ languages — without spending a second on setup.' A blue 'Start coding' button with a double arrow icon is centered. Below this, a large screenshot of the Repl.it IDE is displayed. The IDE has a dark theme with a sidebar on the left showing file navigation. In the center, there's a code editor with Python code, a terminal window showing the Python environment, and a chat interface at the bottom where users are discussing matrix operations.

Signup (it's free), then under "create a repl" click Python:

The screenshot shows the Repl.it home page. On the left, there's a sidebar with navigation links: Home, My repls, Talk, Notifications, Languages, Templates, Tutorials, and Teams (BETA). The main area has a search bar at the top right. Below it, there are sections for "create a repl" (Python, Node.js, C) and "connect GitHub to run your repos". A "Connect GitHub" button is present. The "explore trending repls" section features three examples: "RYANTADIPARTHI / Anger Test" (Python, 51 views, 133 likes), "KobeFF / Breakout" (HTML, CSS, JS, 145 views, 240 likes), and "SmartOne / [GAME] Among Us *TESTING*" (Python, 136 views, 166 likes). At the bottom of the page, there are links for Docs, Feedback, Bugs, Questions, Blog, About, Jobs, Pricing, and Discord.

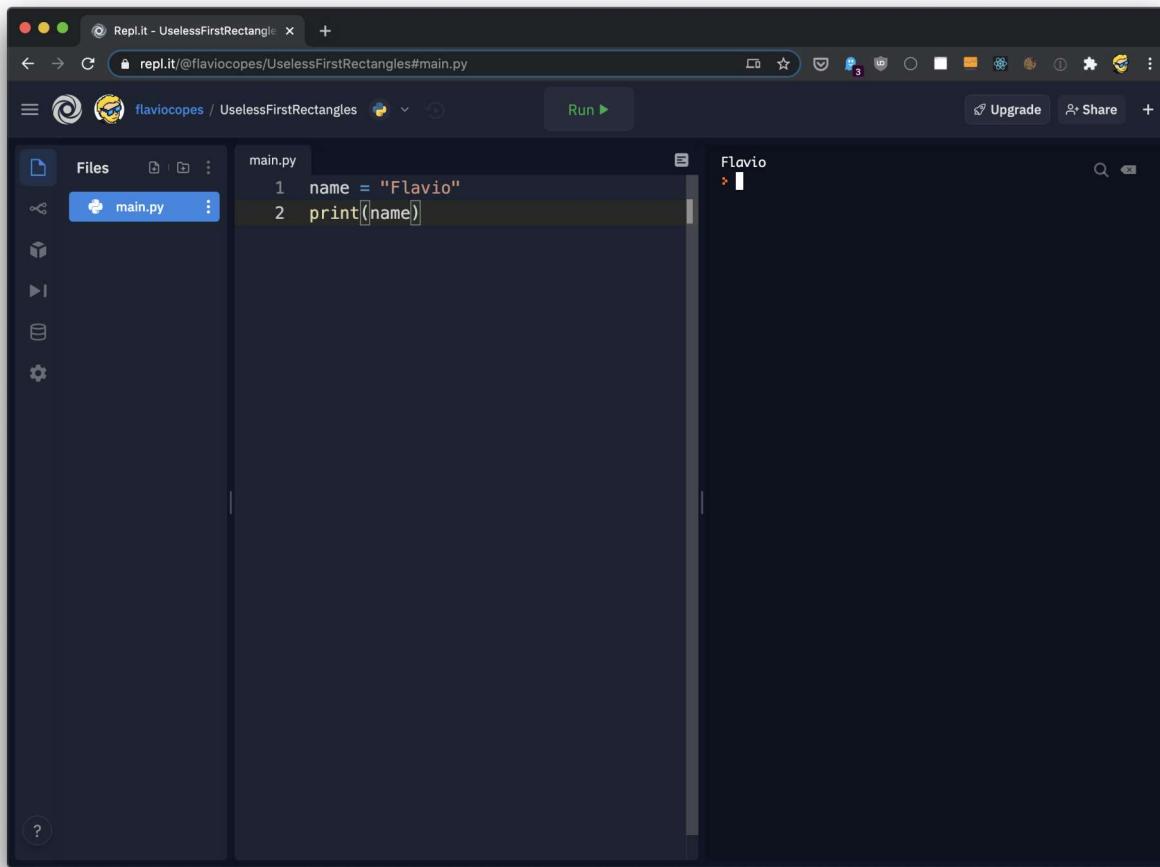
and you will be immediately shown an editor with a file, ready to be filled with a lot of Python code: `main.py`

A screenshot of the Repl.it web-based development environment. The interface is dark-themed. On the left, there's a sidebar with icons for Files, Run, and Settings. The main area shows a file named 'main.py' with the following content:

```
1 Not sure what to do? Run some examples  
(start typing to dismiss)
```

The status bar at the top indicates 'Python 3.8.2 (default, Feb 26 2020, 02:56:10)'. On the right, there's a terminal window with a single character '>' visible.

Once you have some code, click “Run” to run it on the right side of the window:



The screenshot shows a web-based Python code editor interface. At the top, there's a header with the title "Repl.it - UselessFirstRectangle" and a URL "repl.it/@flaviocopes/UselessFirstRectangles#main.py". Below the header is a toolbar with icons for "Run", "Upgrade", and "Share". On the left, there's a sidebar titled "Files" with a single file named "main.py" selected. The main workspace contains the following Python code:

```
1 name = "Flavio"
2 print(name)
```

The code is displayed in a monospaced font, with the first line starting with a number 1. The variable "name" and its value "Flavio" are highlighted in green, while the print statement and the second line are in white. In the bottom right corner of the workspace, there's a small profile icon labeled "Flavio".

I think repl.it is handy because:

- you can easily share code just by sharing the link
- multiple people can work on the same code
- it can host long-running programs
- you can install packages
- it provides you a key-value database for more complex applications

4. Python 2 vs Python 3

One key topic to talk about, right from the start, is the Python 2 vs Python 3 discussion.

Python 3 was introduced in 2008, and it's been in development as the main Python version, while Python 2 continued being maintained with bug fixes and security patches until early 2020.

On that date, Python 2 support was discontinued.

Many programs are still written using Python 2, and organizations still actively work on those, because the migration to Python 3 is not trivial and those programs would require a lot of work to upgrade those programs. And large and important migrations always introduce new bugs.

But new code, unless you have to adhere to rules set by your organization that forces Python 2, should always be written in Python 3.

This book focuses on Python 3.

5. The basics of working with Python

5.1. Variable

We can create a new Python variable by assigning a value to a label, using the assignment operator.=

In this example we assign a string with the value "Roger" to the label:name

```
name = "Roger"
```

Here's an example with a number:

```
age = 8
```

A variable name can be composed by characters, numbers, the underscore character. It can't start with a number. These are all **valid** variable names:_

```
name1  
AGE  
aGE  
a11111  
my_name  
_name
```

These are **invalid** variable names:

```
123  
test!  
name%
```

Other than that, anything is valid unless it's a Python **keyword**. There are some keywords like `, , , and more.` `for` `if` `while` `import`

There's no need to memorize them, as Python will alert you if you use one of those as a variable, and you will gradually recognize them as part of the Python programming language syntax.

5.2. Expressions and statement

We can *expression* any sort of code that returns a value. For example

```
1 + 1  
"Roger"
```

A statement on the other hand is an operation on a value, for example these are 2 statements:

```
name = "Roger"  
print(name)
```

A program is formed by a series of statements. Each statement is put on its own line, but you can use a semicolon to have more than one statement on a single line:

```
name = "Roger"; print(name)
```

5.3. Comment

In a Python program, everything after a hash mark is ignored, and considered a comment:

```
#this is a commented line  
  
name = "Roger" # this is an inline comment
```

5.4. Indentation

Indentation in Python is meaningful.

You cannot indent randomly like this:

```
name = "Flavio"  
    print(name)
```

Algumas outras linguagens não têm espaços em branco significativos, mas em Python, o recuo é importante.

Nesse caso, se você tentar executar este programa, você obteria um erro, porque o recuo tem um significado especial.`IndentationError: unexpected indent`

Tudo recuado pertence a um bloco, como uma instrução de controle ou bloco condicional, ou uma função ou corpo de classe. Veremos mais sobre isso mais adiante.

6. Tipos de dados

Python tem vários tipos embutidos.

Se você criar a variável atribuindo-lhe o valor "Roger", automaticamente essa variável agora está representando um tipo **de dados String**.`name`

```
name = "Roger"
```

Você pode verificar qual tipo uma variável está usando a função, passando a variável como um argumento e, em seguida, comparando o resultado com:`type()`

```
name = "Roger"  
type(name) == str #True
```

Ou usando `:isinstance()`

```
name = "Roger"  
isinstance(name, str) #True
```

Observe que para ver o valor em Python, fora de um REPL, você precisa envolver esse código dentro de `print()`, mas por razões de clareza eu evito usá-lo. Use `print(str())`

Usamos a classe aqui, mas o mesmo funciona para outros tipos de dados.

Primeiro, temos números. Os números inteiros são representados usando a classe. Os números de ponto flutuante (frações) são do tipo:

```
age = 1  
type(age) == int #True  
  
fraction = 0.1  
type(fraction) == float #True
```

Você viu como criar um tipo a partir de um valor literal, assim:

```
name = "Flavio"  
age = 20
```

Python detecta automaticamente o tipo a partir do tipo de valor.

Você também pode criar uma variável de um tipo específico usando o construtor de classe, passando um valor literal ou um nome de variável:

```
name = str("Flavio")
anotherName = str(name)
```

Você também pode converter de um tipo para outro usando o construtor de classe. Python tentará determinar o valor correto, por exemplo, extraíndo um número de uma cadeia de caracteres:

```
age = int("20")
print(age) #20

fraction = 0.1
intFraction = int(fraction)
print(intFraction) #0
```

Isso é chamado **de casting**. É claro que essa conversão nem sempre funciona, dependendo do valor passado. Se você escrever em vez de na cadeia de caracteres acima, você receberá um erro.`ValueError: invalid literal for int() with base 10: 'test'`

Esses são apenas os conceitos básicos de tipos. Temos muito mais tipos em Python:

- `complex` para números complexos
- `bool` para booleanos
- `list` para listas
- `tuple` para tuplas
- `range` para intervalos
- `dict` para dicionários
- `set` para conjuntos

e muito mais!

Vamos explorá-los todos em breve.

7. Operadores

Operadores Python são símbolos que usamos para executar operações sobre valores e variáveis.

Podemos dividir os operadores com base no tipo de operação que eles realizam:

- operador de atribuição
- operadores aritméticos
- operadores de comparação
- operadores lógicos
- operadores bit a bit

além de alguns interessantes como e .isin

7.0.1. Operador de atribuição

O operador de atribuição é usado para atribuir um valor a uma variável:

```
age = 8
```

Ou para atribuir um valor de variável a outra variável:

```
age = 8  
anotherVariable = age
```

Desde o Python 3.8, o *operador morsa* é usado para atribuir um valor a uma variável como parte de outra operação. Por exemplo, dentro de um ou na parte condicional de um loop. Mais sobre isso mais tarde.:=if

7.0.2. Operador aritmético

Python tem um número de operadores aritméticos: , , , (divisão), (restante), (exponenciação) e (divisão de piso):+-*/%**//

```

1 + 1 #2
2 - 1 #1
2 * 2 #4
4 / 2 #2
4 % 3 #1
4 ** 2 #16
4 // 2 #2

```

Observe que você não precisa de um espaço entre os operandos, mas é bom para a legibilidade.

- também funciona como um operador unário menos:

```
print(-4) #-4
```

- + também é usado para concatenar valores de String:

```
"Roger" + " is a good dog"
#Roger is a good dog
```

Podemos combinar o operador de atribuição com operadores aritméticos:

- +=
- -=
- *=
- /=
- %=
- .. e assim por diante

Exemplo:

```

age = 8
age += 1
# age is now 9

```

7.0.3. Operador de comparação

Python define alguns operadores de comparação:

- ==
- !=
- >
- <
- >=
- <=

Você pode usar esses operadores para obter um valor booleano (ou) dependendo do resultado:TrueFalse

```
a = 1  
b = 2  
  
a == b #False  
a != b #True  
a > b #False  
a <= b #True
```

7.0.4. Operador booleano

Python nos dá os seguintes operadores booleanos:

- not
- and
- or

Ao trabalhar com ou atributos, eles funcionam como lógico E, OU e NÃO, e são frequentemente usados na avaliação de expressão condicional:TrueFalseif

```
condition1 = True  
condition2 = False
```

```
not condition1 #False
condition1 and condition2 #False
condition1 or condition2 #True
```

Otherwise, pay attention to a possible source of confusion.

or used in an expression returns the value of the first operand that is not a falsy value (,,,.). Otherwise it returns the last operand.`False0''[]`

```
print(0 or 1) ### 1
print(False or 'hey') ### 'hey'
print('hi' or 'hey') ### 'hi'
print([] or False) ### False
print(False or []) ### []
```

The Python docs describe it as `.if x is false, then y, else x`

and only evaluates the second argument if the first one is true. So if the first argument is falsy (,,,.), it returns that argument. Otherwise it evaluates the second argument:`False0''[]`

```
print(0 and 1) ### 0
print(1 and 0) ### 0
print(False and 'hey') ### False
print('hi' and 'hey') ### 'hey'
print([] and False) ### []
print(False and []) ### False
```

The Python docs describe it as `.if x is false, then x, else y`

7.0.5. Bitwise operator

Some operators are used to work on bits and binary numbers:

- & performs binary AND

- | performs binary OR
- ^ performs a binary XOR operation
- ~ performs a binary NOT operation
- << shift left operation
- >> shift right operation

Bitwise operators are rarely used, only in very specific situations, but they are worth mentioning.

7.0.6. and `is`

`is` is called the **identity operator**. It is used to compare two objects and returns true if both are the same object. More on objects later.

`in` is called the **membership operator**. Is used to tell if a value is contained in a list, or another sequence. More on lists and other sequences later.

8. The Ternary Operator

The ternary operator in Python allows you to quickly define a conditional.

Let's say you have a function that compares an variable to the value, and return True or False depending on the result.

Instead of writing:

```
def is_adult(age):
    if age > 18:
        return True
    else:
        return False
```

You can implement it with the ternary operator in this way:

```
def is_adult(age):
```

```
return True if age > 18 else False
```

First you define the result if the condition is True, then you evaluate the condition, then you define the result if the condition is false:

```
<result_if_true> if <condition> else <result_if_false>
```

9. Strings

A string in Python is a series of characters enclosed into quotes or double quotes:

```
"Roger"  
'Roger'
```

You can assign a string value to a variable:

```
name = "Roger"
```

You can concatenate two strings using the operator:+

```
phrase = "Roger" + " is a good dog"
```

You can append to a string using :+=

```
name = "Roger"  
name += " is a good dog"
```

```
print(name) #Roger is a good dog
```

You can convert a number to a string using the class constructor:str

```
str(8) #"8"
```

This is essential to concatenate a number to a string:

```
print("Roger is " + str(8) + " years old") #Roger is 8 years old
```

A string can be multi-line when defined with a special syntax, enclosing the string in a set of 3 quotes:

```
print("""Roger is  
8  
years old  
""")  
  
#double quotes, or single quotes
```

```
print(''  
Roger is  
  
8  
  
years old  
''')
```

A string has a set of built-in methods, like:

- `isalpha()` to check if a string contains only characters and is not empty
- `isalnum()` to check if a string contains characters or digits and is not empty
- `isdecimal()` to check if a string contains digits and is not empty
- `lower()` to get a lowercase version of a string
- `islower()` to check if a string is lowercase
- `upper()` to get an uppercase version of a string
- `isupper()` to check if a string is uppercase
- `title()` to get a capitalized version of a string
- `startswith()` to check if the string starts with a specific substring

- `endswith()` to check if the string ends with a specific substring
- `replace()` to replace a part of a string
- `split()` to split a string on a specific character separator
- `strip()` to trim the whitespace from a string
- `join()` to append new letters to a string
- `find()` to find the position of a substring

and many more.

None of those methods alter the original string. They return a new, modified string instead. For example:

```
name = "Roger"  
print(name.lower()) #"roger"  
print(name) #"Roger"
```

You can use some global functions to work with strings, too.

In particular I think of `, which gives you the length of a string:len()`

```
name = "Roger"  
print(len(name)) #5
```

The operator `lets you check if a string contains a substring:in`

```
name = "Roger"  
print("ger" in name) #True
```

Escaping is a way to add special characters into a string.

For example, how do you add a double quote into a string that's wrapped into double quotes?

```
name = "Roger"
```

"Ro" "Ger" will not work, as Python will think the string ends at ."Ro"

The way to go is to escape the double quote inside the string, with the backslash character:\

```
name = "Ro\"ger"
```

This applies to single quotes too , and for special formatting characters like for tab, for new line and for the backslash.\ '\t\n\\

Given a string, you can get its characters using square brackets to get a specific item, given its index, starting from 0:

```
name = "Roger"  
name[0] #'R'  
name[1] #'o'  
name[2] #'g'
```

Using a negative number will start counting from the end:

```
name = "Roger"  
name[-1] #"r"
```

You can also use a range, using what we call **slicing**:

```
name = "Roger"  
name[0:2] #"Ro"  
name[:2] #"Ro"  
name[2:] #"ger"
```

10. Booleans

Python provides the type, which can have two values: and (capitalized)boolTrueFalse

```
done = False  
done = True
```

Booleans are especially useful with conditional control structures like statements:`if`

```
done = True  
  
if done:  
    # run some code here  
else:  
    # run some other code
```

When evaluating a value for `or`, if the value is not a we have some rules depending on the type we're checking:`True``False``bool`

- numbers are always unless for the number `True``0`
- strings are only when empty `False`
- lists, tuples, sets, dictionaries are only when empty `False`

You can check if a value is a boolean in this way:

```
done = True  
type(done) == bool #True
```

Or using `, passing 2 arguments: the variable, and the class:``isinstance()``bool`

```
done = True  
isinstance(done, bool) #True
```

The `global` function is also very useful when working with booleans, as it returns if any of the values of the iterable (list, for example) passed as argument are `:any()``True``True`

```
book_1_read = True  
book_2_read = False  
  
read_any_book = any([book_1_read, book_2_read])
```

A função global é a mesma, mas retorna se todos os valores passados para ela forem :all()
TrueTrue

```
ingredients_purchased = True  
meal_cooked = False  
  
ready_to_serve = all([ingredients_purchased, meal_cooked])
```

11. Números

Os números em Python podem ser de 3 tipos: , e .intfloatcomplex

11.0.1. Número inteiro

Os números inteiros são representados usando a classe. Você pode definir um inteiro usando um literal de valor:int

```
age = 8
```

Você também pode definir um número inteiro usando o construtor:int()

```
age = int(8)
```

Para verificar se uma variável é do tipo , você pode usar a função global:inttype()

```
type(age) == int #True
```

11.0.2. Número do ponto flutuante

Os números de ponto flutuante (frações) são do tipo . Você pode definir um inteiro usando um literal de valor:`float`

```
fraction = 0.1
```

Ou usando o construtor:`float()`

```
fraction = float(0.1)
```

Para verificar se uma variável é do tipo , você pode usar a função global:`floattype()`

```
type(fraction) == float #True
```

11.0.3. Número complexo

Os números complexos são do tipo `.complex`

Você pode defini-los usando um literal de valor:

```
complexNumber = 2+3j
```

ou usando o construtor:`complex()`

```
complexNumber = complex(2, 3)
```

Depois de ter um número complexo, você pode obter sua parte real e imaginária:

```
complexNumber.real #2.0  
complexNumber.imag #3.0
```

Novamente, para verificar se uma variável é do tipo , você pode usar a função global:`complexType()`

```
type(complexNumber) == complex #True
```

11.0.4. Operações aritméticas em número

Você pode executar operações aritméticas em números, usando os operadores aritméticos: , , , (divisão), (restante), (exponenciação) e (divisão de piso):`+-*/%**//`

```
1 + 1 #2
2 - 1 #1
2 * 2 #4
4 / 2 #2
4 % 3 #1
4 ** 2 #16
4 // 2 #2
```

e você pode usar os operadores de atribuição compostos

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- .. e assim por diante

para executar rapidamente operações em variáveis também:

```
age = 8
age += 1
```

11.0.5. Função incorporada

Existem 2 funções internas que ajudam com números:

`abs()` retorna o valor absoluto de um número.

`round()` dado um número, retorna seu valor arredondado para o inteiro mais próximo:

```
round(0.12) #0
```

Você pode especificar um segundo parâmetro para definir a precisão dos pontos decimais:

```
round(0.12, 1) #0.1
```

Several other math utility functions and constants are provided by the Python standard library:

- the `math` package provides general math functions and constants
- the `cmath` package provides utilities to work with complex numbers
- the `decimal` package provides utilities to work with decimals and floating point numbers
- the `fractions` package provides utilities to work with rational numbers

We'll explore some of those separately later on.

12. Constants

Python não tem como impor uma variável para ser uma constante.

O mais próximo que você pode ir é usar um enum:

```
class Constants(Enum):  
    WIDTH = 1024  
    HEIGHT = 256
```

E chegar a cada valor usando, por exemplo, `.Constants.WIDTH.value`

Ninguém pode reatribuir esse valor.

Caso contrário, se você quiser confiar em convenções de nomenclatura, você pode aderir a esta: declarar variáveis que nunca devem mudar maiúsculas:

```
WIDTH = 1024
```

Ninguém impedirá de substituir esse valor, e o Python não o impedirá.

Isso é o que faz a maioria do código Python que você verá.

13. Enums

Enums são nomes legíveis que estão vinculados a um valor constante.

Para usar enums, importe do módulo de biblioteca padrão:`Enum`

```
from enum import Enum
```

Em seguida, você pode inicializar uma nova enum da seguinte maneira:

```
class State(Enum):
    INACTIVE = 0
    ACTIVE = 1
```

Depois de fazer isso, você pode referenciar e , e eles servem como
`constants.State.INACTIVE``constants.State.ACTIVE`

Agora, se você tentar imprimir, por exemplo:`State.ACTIVE`

```
print(State.ACTIVE)
```

ele não vai voltar , mas `.1State.ACTIVE`

O mesmo valor pode ser alcançado pelo número atribuído no enum:
retornará . O mesmo para usar a notação de colchetes
.print(State(1))State.ACTIVESTate['ACTIVE']

No entanto, você pode obter o valor usando o .State.ACTIVE.value

Você pode listar todos os valores possíveis de um enum:

```
list(State) # [<State.INACTIVE: 0>, <State.ACTIVE: 1>]
```

Você pode contá-los:

```
len(State) # 2
```

14. Entrada do usuário

Em um aplicativo de linha de comando Python, você pode exibir informações para o usuário usando a função:print()

```
name = "Roger"  
print(name)
```

Também podemos aceitar a entrada do usuário, usando:input()

```
print('What is your age?')  
age = input()  
print('Your age is ' + age)
```

Essa abordagem recebe entrada em tempo de execução, o que significa que o programa interromperá a execução e aguardará até que o usuário digite algo e pressione a tecla.enter

Você também pode fazer um processamento de entrada mais complexo e aceitar a entrada no momento da invocação do programa, e veremos como fazer isso mais tarde.

Isso funciona para aplicativos de linha de comando. Outros tipos de aplicativos precisarão de uma maneira diferente de aceitar entradas.

15. Instruções de controle

O que é interessante fazer com booleanos, e expressões que retornam um booleano em particular, é que podemos tomar decisões e tomar caminhos diferentes, dependendo de seu valor ou valor.`True``False`

Em Python fazemos isso usando a instrução:`if`

```
condition = True

if condition == True:
    # do something
```

Quando o teste de condição é resolvido para `True`, como no caso acima, seu bloco é executado.

O que é um bloco? Um bloco é a parte que é recuada um nível (4 espaços geralmente) à direita:

```
condition = True

if condition == True:
    print("The condition")
    print("was true")
```

O bloco pode ser formado por uma única linha, ou várias linhas também, e termina quando você volta para o nível de recuo anterior:

```
condition = True

if condition == True:
    print("The condition")
    print("was true")
```

```
print("Outside of the if")
```

Em combinação com você pode ter um bloco, que é executado se o teste de condição de resultados para:`if``else``if``False`

```
condition = True

if condition == True:
    print("The condition")
    print("was True")

else:
    print("The condition")
    print("was False")
```

E você pode ter diferentes verificações vinculadas com o , que são executadas se a verificação anterior tiver sido:`if``elif``if``False`

```
condition = True
name = "Roger"

if condition == True:
    print("The condition")
    print("was True")
elif name == "Roger":
    print("Hello Roger")
else:
    print("The condition")
    print("was False")
```

O segundo bloco, neste caso, é executado se for , e o valor da variável é "Roger".`condition``False``name`

Em uma instrução, você pode ter apenas uma e verifica, mas várias séries de verificações:`if``if``else``elif`

```
condition = True
name = "Roger"

if condition == True:
    print("The condition")
    print("was True")
elif name == "Roger":
    print("Hello Roger")
elif name == "Syd":
    print("Hello Syd")
elif name == "Flavio":
    print("Hello Flavio")
else:
    print("The condition")
    print("was False")
```

`if` e também pode ser usado em um formato embutido, o que nos permite retornar um valor ou outro com base em uma condição.`else`

Exemplo:

```
a = 2
result = 2 if a == 0 else 3
print(result) # 3
```

16. Listas

As listas são uma estrutura de dados Python essencial.

O permite agrupar vários valores e fazer referência a todos eles com um nome comum.

Por exemplo:

```
dogs = ["Roger", "Syd"]
```

Uma lista pode conter valores de diferentes tipos:

```
items = ["Roger", 1, "Syd", True]
```

Você pode verificar se um item está contido em uma lista com o operador:`in`

```
print("Roger" in items) # True
```

Uma lista também pode ser definida como vazia:

```
items = []
```

Você pode fazer referência aos itens em uma lista por seu índice, começando de zero:

```
items[0] # "Roger"  
items[1] # 1  
items[3] # True
```

Usando a mesma notação, você pode alterar o valor armazenado em um índice específico:

```
items[0] = "Roger"
```

Você também pode usar o método:`index()`

```
items.index("Roger") # 0  
items.index("Syd") # 2
```

As with strings, using a negative index will start searching from the end:

```
items[-1] # True
```

You can also extract a part of a list, using slices:

```
items[0:2] # ["Roger", 1]
items[2:] # ["Syd", True]
```

Get the number of items contained in a list using the global function, the same we used to get the length of a string:`len()`

```
len(items) #4
```

You can add items to the list by using a list method:`append()`

```
items.append("Test")
```

or the `extend()` method:

```
items.extend(["Test"])
```

Você também pode usar o operador:`+=`

```
items += ["Test"]
# items is ['Roger', 1, 'Syd', True, 'Test']
```

Dica: com ou não se esqueça dos colchetes. Não faça ou ou Python irá adicionar 4 caracteres individuais à lista, resultando em `extend()+=items += "Test"`
`items.extend("Test")`['Roger', 1, 'Syd', True, 'T', 'e', 's', 't']

Remova um item usando o método:`remove()`

```
items.remove("Test")
```

Você pode adicionar vários elementos usando

```
items += ["Test1", "Test2"]

#or

items.extend(["Test1", "Test2"])
```

Estes acrescentam o item ao final da lista.

Para adicionar um item no meio de uma lista, em um índice específico, use o método:`insert()`

```
items.insert(1, "Test") # add "Test" at index 1
```

Para adicionar vários itens em um índice específico, você precisa usar fatias:

```
items[1:1] = ["Test1", "Test2"]
```

Classifique uma lista usando o método:`sort()`

```
items.sort()
```

Dica: `sort()` só funcionará se a lista contiver valores que possam ser comparados. Cadeias de caracteres e inteiros, por exemplo, não podem ser comparados, e você receberá um erro como se tentasse.`TypeError: '<' not supported between instances of 'int' and 'str'`

Os métodos ordenam letras maiúsculas primeiro, depois letras minúsculas. Para corrigir isso, use:`sort()`

```
items.sort(key=str.lower)
```

em vez de.

A classificação modifica o conteúdo da lista original. Para evitar isso, você pode copiar o conteúdo da lista usando

```
items_copy = items[:]
```

ou use a função `global:sorted()`

```
print(sorted(items, key=str.lower))
```

que retornará uma nova lista, classificada, em vez de modificar a lista original.

17. Tuplas

Tuplas são outra estrutura de dados Python fundamental.

Eles permitem que você crie grupos imutáveis de objetos. Isso significa que, uma vez que uma tupla é criada, ela não pode ser modificada. Não é possível adicionar ou remover itens.

Eles são criados de forma semelhante às listas, mas usando parênteses em vez de colchetes:

```
names = ("Roger", "Syd")
```

A tuple is ordered, like a list, so you can get its values referencing an index value:

```
names[0] # "Roger"  
names[1] # "Syd"
```

You can also use the method:`index()`

```
items.index("Roger") # 0  
items.index("Syd") # 2
```

As with strings and lists, using a negative index will start searching from the end:

```
names[-1] # True
```

You can count the items in a tuple with the function:`len()`

```
len(names) # 2
```

You can check if an item is contained into a tuple with the operator:`in`

```
print("Roger" in names) # True
```

You can also extract a part of a tuple, using slices:

```
names[0:2] # ('Roger', 'Syd')  
names[1:] # ('Syd',)
```

Obtenha o número de itens em uma tupla usando a função global, a mesma que usamos para obter o comprimento de uma cadeia de caracteres:`len()`

```
len(names) #2
```

Você pode criar uma versão classificada de uma tupla usando a função global:`sorted()`

```
sorted(names)
```

Você pode criar uma nova tupla a partir de tuplas existentes usando o operador:`+`

```
newTuple = names + ("Vanille", "Tina")
```

18. Dicionários

Os dicionários são uma estrutura de dados Python muito importante.

Enquanto as listas permitem que você crie coleções de valores, os dicionários permitem que você crie coleções de **pares de chave / valor**.

Aqui está um exemplo de dicionário com um par chave/valor:

```
dog = { 'name': 'Roger' }
```

A chave pode ser qualquer valor imutável, como uma cadeia de caracteres, um número ou uma tupla. O valor pode ser o que você quiser.

Um dicionário pode conter vários pares chave/valor:

```
dog = { 'name': 'Roger', 'age': 8 }
```

Você pode acessar valores de chave individuais usando esta notação:

```
dog['name'] # 'Roger'  
dog['age'] # 8
```

Usando a mesma notação, você pode alterar o valor armazenado em um índice específico:

```
dog['name'] = 'Syd'
```

E outra maneira é usando o método, que tem uma opção para adicionar um valor padrão:`get()`

```
dog.get('name') # 'Roger'  
dog.get('test', 'default') # 'default'
```

O método recupera o valor de uma chave e, posteriormente, exclui o item do dicionário:pop()

```
dog.pop('name') # 'Roger'
```

O método recupera e remove o último par chave/valor inserido no dicionário:popitem()

```
dog.popitem()
```

Você pode verificar se uma chave está contida em um dicionário com o operador:in

```
'name' in dog # True
```

Obtenha uma lista com as chaves em um dicionário usando o método, passando seu resultado para o construtor:keys().list()

```
list(dog.keys()) # ['name', 'age']
```

Obtenha os valores usando o método e as tuplas de pares chave/valor usando o método:values().items()

```
print(list(dog.values()))  
# ['Roger', 8]  
  
print(list(dog.items()))  
# [('name', 'Roger'), ('age', 8)]
```

Obtenha um comprimento de dicionário usando a função global, a mesma que usamos para obter o comprimento de uma cadeia de caracteres ou os

itens em uma lista:`len()`

```
len(dog) #2
```

Você pode adicionar um novo par chave/valor ao dicionário da seguinte maneira:

```
dog['favorite food'] = 'Meat'
```

Você pode remover um par chave/valor de um dicionário usando a instrução:`del`

```
del dog['favorite food']
```

Para copiar um dicionário, use o método `copy()`:

```
dogCopy = dog.copy()
```

19. Conjuntos

Os conjuntos são outra importante estrutura de dados Python.

Podemos dizer que funcionam como tuplas, mas não são ordenadas e são **mutáveis**. Ou podemos dizer que funcionam como dicionários, mas não têm chaves.

Eles também têm uma versão imutável, chamada `.frozenset`

Você pode criar um conjunto usando esta sintaxe:

```
names = {"Roger", "Syd"}
```

Os conjuntos funcionam bem quando você pensa neles como conjuntos matemáticos.

Você pode cruzar dois conjuntos:

```
set1 = {"Roger", "Syd"}  
set2 = {"Roger"}  
  
intersect = set1 & set2 #{'Roger'}
```

Você pode criar uma união de dois conjuntos:

```
set1 = {"Roger", "Syd"}  
set2 = {"Luna"}  
  
union = set1 | set2  
#{'Syd', 'Luna', 'Roger'}
```

Você pode obter a diferença entre dois conjuntos:

```
set1 = {"Roger", "Syd"}  
set2 = {"Roger"}  
  
difference = set1 - set2 #{'Syd'}
```

Você pode verificar se um conjunto é um superconjunto de outro (e, claro, se um conjunto é um subconjunto de outro)

```
set1 = {"Roger", "Syd"}  
set2 = {"Roger"}  
  
isSuperset = set1 > set2 # True
```

Você pode contar os itens em um conjunto com a função global: len()

```
names = {"Roger", "Syd"}  
len(names) # 2
```

Você pode obter uma lista dos itens em um conjunto passando o conjunto para o construtor:`list()`

```
names = {"Roger", "Syd"}  
list(names) #['Syd', 'Roger']
```

Você pode verificar se um item está contido em um conjunto com o operador:`in`

```
print("Roger" in names) # True
```

20. Funções

Uma função nos permite criar um conjunto de instruções que podemos executar quando necessário.

As funções são essenciais em Python e em muitas outras linguagens de programação para criar programas significativos, porque nos permitem decompor um programa em partes gerenciáveis, promovem a legibilidade e a reutilização de código.

Aqui está um exemplo de função chamada que imprime "Olá!":`hello`

```
def hello():  
    print('Hello!')
```

Esta é a **definição da função**. Há um nome () e um corpo, o conjunto de instruções, que é a parte que segue os dois pontos e é recuada um nível à direita.`hello`

Para executar essa função, devemos chamá-la. Esta é a sintaxe para chamar a função:

```
hello()
```

Podemos executar essa função uma ou várias vezes.

O nome da função, , é muito importante. Ele deve ser descritivo, para que qualquer um que o chame possa imaginar o que a função faz.`hello`

Uma função pode aceitar um ou mais parâmetros:

```
def hello(name):  
    print('Hello ' + name + '!')
```

Neste caso, chamamos a função de passar o argumento

```
hello('Roger')
```

Chamamos *de parâmetros* os valores aceitos pela função dentro da definição da função e *argumentamos* os valores que passamos para a função quando a chamamos. É comum ficar confuso sobre essa distinção.

Um argumento pode ter um valor padrão que é aplicado se o argumento não for especificado:

```
def hello(name='my friend'):  
    print('Hello ' + name + '!')
```

```
hello()  
#Hello my friend!
```

Veja como podemos aceitar vários parâmetros:

```
def hello(name, age):  
    print('Hello ' + name + ', you are ' + str(age) + ' years old!')
```

Neste caso, chamamos a função de passar um conjunto de argumentos:

```
hello('Roger', 8)
```

Os parâmetros são passados por referência. Todos os tipos em Python são objetos, mas alguns deles são imutáveis, incluindo inteiros, booleanos, flutuadores, cadeias de caracteres e tuplas. Isso significa que, se você passá-los como parâmetros e modificar seu valor dentro da função, o novo valor não será refletido fora da função:

```
def change(value):
    value = 2

val = 1
change(val)

print(val) #1
```

Se você passar um objeto que não seja imutável e alterar uma de suas propriedades, a alteração será refletida do lado de fora.

Uma função pode retornar um valor, usando a instrução. Por exemplo, neste caso, retornamos o nome do parâmetro:`return name`

```
def hello(name):
    print('Hello ' + name + '!')
    return name
```

Quando a função atende à instrução, a função termina.`return`

Podemos omitir o valor:

```
def hello(name):
    print('Hello ' + name + '!')
    return
```

Podemos ter a instrução `return` dentro de uma condicional, que é uma maneira comum de encerrar uma função se uma condição inicial não for atendida:

```
def hello(name):
    if not name:
        return
    print('Hello ' + name + '!')
```

Se chamarmos a função passando um valor que é avaliado como , como uma cadeia de caracteres vazia, a função é encerrada antes de chegar à instrução.`Falseprint()`

Você pode retornar vários valores usando valores separados por vírgula:

```
def hello(name):
    print('Hello ' + name + '!')
    return name, 'Roger', 8
```

Nesse caso, chamar o valor de retorno (observação: não o que está impresso na tela, mas o valor de retorno) é uma tupla contendo esses 3 valores:`hello('Syd')('Syd', 'Roger', 8)`

```
def hello(name):
    print('Hello ' + name + '!')
    return name, 'Roger', 8
```

```
print(hello('Syd')) #('Syd', 'Roger', 8)
```

21. Objetos

Tudo em Python é um objeto.

Mesmo valores de tipos primitivos básicos (int, str, float..) são objetos. Listas são objetos, tuplas, dicionários, tudo.

Os objetos têm **atributos** e **métodos** que podem ser acessados usando a sintaxe de pontos.

Por exemplo, tente definir uma nova variável do tipo:int

```
age = 8
```

age agora tem acesso às propriedades e métodos definidos para todos os objetos.int

Isso inclui, por exemplo, o acesso à parte real e imaginária desse número:

```
print(age.real) # 8
print(age.imag) # 0

print(age.bit_length()) #4

# the bit_length() method returns the number of bits necessary
# to represent this number in binary notation
```

Uma variável que contém um valor de lista tem acesso a um conjunto diferente de métodos:

```
items = [1, 2]
items.append(3)
items.pop()
```

Os métodos dependem do tipo de valor.

A função global fornecida pelo Python permite que você inspecione o local na memória de um objeto específico.id()

```
id(age) # 140170065725376
```

Seu valor de memória vai mudar, estou apenas mostrando-o como um exemplo

Se você atribuir um valor diferente à variável, seu endereço será alterado, porque o conteúdo da variável foi substituído por outro valor armazenado em outro local na memória:

```
age = 8

print(id(age)) # 140535918671808

age = 9

print(id(age)) # 140535918671840
```

Mas se você modificar o objeto usando seus métodos, o endereço permanecerá o mesmo:

```
items = [1, 2]

print(id(items)) # 140093713593920

items.append(3)

print(items) # [1, 2, 3]
print(id(items)) # 140093713593920
```

O endereço só será alterado se você reatribuir uma variável a outro valor.

Alguns objetos são *mutáveis*, alguns são *imutáveis*. Isso depende do próprio objeto. Se o objeto fornecer métodos para alterar seu conteúdo, ele será mutável. Caso contrário, é imutável. A maioria dos tipos definidos pelo Python são imutáveis. Por exemplo, um é imutável. Não há métodos para alterar seu valor. Se você incrementar o valor usando `int`

```
age = 8
age = age + 1

# or
```

```
age += 1
```

e você verificar com você vai encontrar que aponta para um local de memória diferente. O valor original não sofreu mutação, mudamos para outro valor.id(age)age

22. Loops

Os loops são uma parte essencial da programação.

Em Python temos 2 tipos de loops: **enquanto loops** e **para loops**.

22.1. Loopwhile

while loops são definidos usando a palavra-chave e repetem seu bloco até que a condição seja avaliada como: while False

```
condition = True
while condition == True:
    print("The condition is True")
```

Este é um **loop infinito**. Nunca acaba.

Vamos interromper o loop logo após a primeira iteração:

```
condition = True
while condition == True:
    print("The condition is True")
    condition = False

print("After the loop")
```

Nesse caso, a primeira iteração é executada, à medida que o teste de condição é avaliado para , e na segunda iteração o teste de condição é

avaliado como , de modo que o controle vai para a próxima instrução, após o loop.`True``False`

É comum ter um contador para interromper a iteração após algum número de ciclos:

```
count = 0
while count < 10:
    print("The condition is True")
    count = count + 1

print("After the loop")
```

22.2. Loop for

Usando loops, podemos dizer ao Python para executar um bloco por uma quantidade predeterminada de vezes, antecipadamente e sem a necessidade de uma variável separada e condicional para verificar seu valor.`for`

Por exemplo, podemos iterar os itens em uma lista:

```
items = [1, 2, 3, 4]
for item in items:
    print(item)
```

Ou, você pode iterar uma quantidade específica de vezes usando a função:`range()`

```
for item in range(0, 4):
    print(item)
```

`range(4)` cria uma sequência que começa a partir de 0 e contém 4 itens: `[0, 1, 2, 3]`

Para obter o índice, você deve encapsular a sequência na função:`enumerate()`

```
items = [1, 2, 3, 4]
for index, item in enumerate(items):
    print(index, item)
```

22.3. Interromper e continuar

Ambos os loops podem ser interrompidos dentro do bloco, usando duas palavras-chave especiais: `while`, `for`, `break` e `continue`.

`continue` interrompe a iteração atual e diz ao Python para executar a próxima.

`break` interrompe o loop completamente e continua com a próxima instrução após o término do loop.

O primeiro exemplo aqui imprime . O segundo exemplo imprime: 1, 3, 41

```
items = [1, 2, 3, 4]
for item in items:
    if item == 2:
        continue
    print(item)
```

```
items = [1, 2, 3, 4]
for item in items:
    if item == 2:
        break
    print(item)
```

23. Aulas

Além de usar os tipos fornecidos pelo Python, podemos declarar nossas próprias classes e, a partir de classes, podemos instanciar objetos.

Um objeto é uma instância de uma classe. Uma classe é o tipo de um objeto.

Defina uma classe da seguinte maneira:

```
class <class_name>:  
    # my class
```

Por exemplo, vamos definir uma classe Dog

```
class Dog:  
    # the Dog class
```

Uma classe pode definir métodos:

```
class Dog:  
    # the Dog class  
    def bark(self):  
        print('WOF!')
```

self como o argumento do método aponta para a instância do objeto atual e deve ser especificado ao definir um método.

Criamos uma instância de uma classe, um **objeto**, usando esta sintaxe:

```
roger = Dog()
```

Agora é um novo objeto do tipo Dog.roger

Se você executar

```
print(type(roger))
```

Você terá <class '__main__.Dog'>

Um tipo especial de método, é chamado construtor, e podemos usá-lo para inicializar uma ou mais propriedades quando criamos um novo objeto a partir dessa classe:`__init__()`

```
class Dog:  
    # the Dog class  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def bark(self):  
        print('WOF!')
```

Utilizamo-lo da seguinte forma:

```
roger = Dog('Roger', 8)  
print(roger.name) # 'Roger'  
print(roger.age) # 8  
  
roger.bark() # 'WOF!'
```

Uma característica importante das classes é a herança.

Podemos criar uma classe Animal com um método:walk()

```
class Animal:  
    def walk(self):  
        print('Walking..')
```

e a classe Dog pode herdar de Animal:

```
class Dog(Animal):  
    def bark(self):
```

```
print('WOF!')
```

Agora, a criação de um novo objeto de classe terá o método como ele é herdado de:Dogwalk()Animal

```
roger = Dog()  
roger.walk() # 'Walking..'  
roger.bark() # 'WOF!'
```

24. Módulos

Cada arquivo Python é um módulo.

Você pode importar um módulo de outros arquivos, e essa é a base de qualquer programa de complexidade moderada, pois promove uma organização sensata e a reutilização de código.

No programa Python típico, um arquivo atua como o ponto de entrada. Os outros arquivos são módulos e expõem funções que podemos chamar de outros arquivos.

O arquivo contém este código:dog.py

```
def bark():  
    print('WOF!')
```

Podemos importar esta função de outro arquivo usando , e uma vez que o fazemos, podemos referenciar a função usando a notação de ponto, :importdog.bark()

```
import dog
```

```
dog.bark()
```

Ou, podemos usar a sintaxe e chamar a função diretamente:from .. import

```
from dog import bark
```

```
bark()
```

A primeira estratégia nos permite carregar tudo o que é definido em um arquivo.

A segunda estratégia nos permite escolher as coisas de que precisamos.

Esses módulos são específicos para o seu programa, e a importação depende do local do arquivo no sistema de arquivos.

Suponha que você coloque em uma subpasta `dog.pylib`

Nessa pasta, você precisa criar um arquivo vazio chamado `.`. Isso informa ao Python que a pasta contém módulos. `__init__.py`

Agora você pode escolher, você pode importar de: `doglib`

```
from lib import dog
```

```
dog.bark()
```

ou você pode fazer referência à função específica do módulo importando de: `doglib.dog`

```
from lib.dog import bark
```

```
bark()
```

25. A Biblioteca Padrão Python

Python expõe um monte de funcionalidade embutida através de sua **biblioteca padrão**.

A biblioteca padrão é uma enorme coleção de todos os tipos de utilitários, desde utilitários matemáticos até depuração e criação de interfaces gráficas de usuário.

Você pode encontrar a lista completa de módulos de biblioteca padrão aqui: <https://docs.python.org/3/library/index.html>

Alguns dos módulos importantes são:

- `math` para utilitários de matemática
- `re` para expressões regulares
- `json` para trabalhar com JSON
- `datetime` para trabalhar com datas
- `sqlite3` para usar o SQLite
- `os` para utilitários do sistema operacional
- `random` para geração de números aleatórios
- `statistics` para utilitários de estatísticas
- `requests` para executar solicitações de rede HTTP
- `http` para criar servidores HTTP
- `urllib` para gerenciar URLs

Vamos apresentar como *usar* um módulo da biblioteca padrão. Você já sabe como usar módulos que você cria, importando de outros arquivos na pasta do programa.

Bem, isso é o mesmo com os módulos fornecidos pela biblioteca padrão:

```
import math  
  
math.sqrt(4) # 2.0
```

ou

```
from math import sqrt  
  
sqrt(4) # 2.0
```

Em breve, exploraremos os módulos mais importantes individualmente para entender o que podemos fazer com eles.

26. O guia de estilo do Python PEP8

Quando você escreve código, você deve aderir às convenções da linguagem de programação que você usa.

Se você aprender as convenções corretas de nomenclatura e formatação desde o início, será mais fácil ler o código escrito por outras pessoas e as pessoas acharão seu código mais fácil de ler.

Python define suas convenções no guia de estilo PEP8. PEP significa Python *Enhancement Proposals* e é o lugar onde todos os aprimoramentos e discussões da linguagem Python acontecem. Há muitas propostas de PEP, todas disponíveis em <https://www.python.org/dev/peps/>.

A PEP8 é uma das primeiras e uma das mais importantes também. Ele define a formatação e também algumas regras sobre como escrever Python de forma "pythonic".

Você pode ler seu conteúdo completo aqui:

<https://www.python.org/dev/peps/pep-0008/> mas aqui está um breve resumo dos pontos importantes com os quais você pode começar:

- Recuar usando espaços, não tabulações
- Recuo usando 4 espaços.
- Os arquivos Python são codificados em UTF-8
- Use no máximo 80 colunas para seu código
- Escreva cada instrução em sua própria linha
- Funções, nomes de variáveis e nomes de arquivos são minúsculos, com sublinhados entre palavras (`snake_case`)
- Os nomes das classes são capitalizados, palavras separadas são escritas com a letra maiúscula também, (`CamelCase`)
- Os nomes dos pacotes são minúsculos e não têm sublinhados entre as palavras

- As variáveis que não devem ser alteradas (constantes) são escritas em maiúsculas
- Os nomes das variáveis devem ser significativos
- Adicione comentários úteis, mas evite comentários óbvios
- Adicionar espaços ao redor dos operadores
- Não use espaços em branco desnecessários
- Adicionar uma linha em branco antes de uma função
- Adicionar uma linha em branco entre métodos em uma classe
- Dentro de funções/métodos, linhas em branco podem ser usadas para separar blocos de código relacionados para ajudar na legibilidade

27. Depuração

A depuração é uma das melhores habilidades que você pode aprender, pois irá ajudá-lo em muitas situações difíceis.

Cada idioma tem seu depurador. Python tem , disponível através da biblioteca padrão.pdb

Você depura adicionando um ponto de interrupção em seu código:

```
breakpoint()
```

Você pode adicionar mais pontos de interrupção, se necessário.

Quando o interpretador Python atingir um ponto de interrupção em seu código, ele será interrompido e informará qual é a próxima instrução que ele executará.

Então e você pode fazer algumas coisas.

Você pode digitar o nome de qualquer variável para inspecionar seu valor.

Você pode pressionar para avançar para a próxima linha na função atual. Se o código chamar funções, o depurador não entrará nelas e as considerará "caixas pretas".n

Você pode pressionar para avançar para a próxima linha na função atual. Se a próxima linha for uma função, o depurador entrará nela e você poderá executar uma instrução dessa função de cada vez.

Você pode pressionar para continuar a execução do programa normalmente, sem a necessidade de fazê-lo passo-a-passo.c

Você pode pressionar para interromper a execução do programa.q

A depuração é útil para avaliar o resultado de uma instrução, e é especialmente bom saber como usá-la quando você tem iterações ou algoritmos complexos que deseja corrigir.

28. Âmbito das variáveis

Quando você declara uma variável, essa variável é visível em partes do programa, dependendo de onde você a declara.

Se você declará-la fora de qualquer função, a variável ficará visível para qualquer código em execução após a declaração, incluindo funções:

```
age = 8

def test():
    print(age)

print(age) # 8
test() # 8
```

Chamamos isso de **variável global**.

Se você definir uma variável dentro de uma função, essa variável será uma **variável local** e só será visível dentro dessa função. Fora da função, ele não é acessível:

```
def test():
    age = 8
    print(age)

test() # 8

print(age)
# NameError: name 'age' is not defined
```

29. Aceite argumentos da linha de comando

Python oferece várias maneiras de lidar com argumentos passados quando invocamos o programa a partir da linha de comando.

Até agora, você executou programas a partir de um REPL ou usando

```
python <filename>.py
```

Você pode passar argumentos e opções adicionais ao fazer isso, como este:

```
python <filename>.py <argument1>
python <filename>.py <argument1> <argument2>
```

Uma maneira básica de lidar com esses argumentos é usar o módulo da biblioteca padrão.sys

Você pode obter os argumentos passados na lista:sys.argv

```
import sys
print(len(sys.argv))
print(sys.argv)
```

A lista contém como primeiro item o nome do arquivo que foi executado, por exemplo, `.sys.argv['main.py']`

Esta é uma maneira simples, mas você tem que fazer muito trabalho. Você precisa validar argumentos, certificar-se de que seu tipo está correto, você precisa imprimir comentários para o usuário se eles não estão usando o programa corretamente.

Python fornece outro pacote na biblioteca padrão para ajudá-lo: `.argparse`

Primeiro você importa e você chama , passando a descrição do seu programa:`argparse.ArgumentParser()`

```
import argparse

parser = argparse.ArgumentParser(
    description='This program prints the name of my dogs'
)
```

Em seguida, você prossegue para adicionar argumentos que deseja aceitar. Por exemplo, neste programa, aceitamos uma opção para passar uma cor, como esta: `-c python program.py -c red`

```
import argparse

parser = argparse.ArgumentParser(
    description='This program prints a color HEX value'
)

parser.add_argument('-c', '--color', metavar='color', required=True,
    help='the color to search for')

args = parser.parse_args()

print(args.color) # 'red'
```

Se o argumento não for especificado, o programa gerará um erro:

```
→ python python program.py
usage: program.py [-h] -c color
program.py: error: the following arguments are required: -c
```

Você pode definir uma opção para ter um conjunto específico de valores, usando:`choices`

```
parser.add_argument('-c', '--color', metavar='color', required=True,
    choices=['red', 'yellow'], help='the color to search for')
```

```
→ python python program.py -c blue
usage: program.py [-h] -c color
program.py: error: argument -c/--color: invalid choice: 'blue'
(choose from 'yellow', 'red')
```

Há mais opções, mas essas são as básicas.

E também existem pacotes da comunidade que fornecem essa funcionalidade, como o [Click](#) e o [Python Prompt Toolkit](#).

30. Funções do Lambda

Funções lambda (também chamadas de funções anônimas) são pequenas funções que não têm nome e têm apenas uma expressão como seu corpo.

Em Python eles são definidos usando a palavra-chave:`lambda`

```
lambda <arguments> : <expression>
```

O corpo deve ser uma única expressão. Expressão, não uma declaração.

Essa diferença é importante. Uma expressão retorna um valor, uma instrução não.

O exemplo mais simples de uma função lambda é uma função que dobra o valor de um número:

```
lambda num : num * 2
```

As funções do Lambda podem aceitar mais argumentos:

```
lambda a, b : a * b
```

As funções do Lambda não podem ser invocadas diretamente, mas você pode atribuí-las a variáveis:

```
multiply = lambda a, b : a * b
```

```
print(multiply(2, 2)) # 4
```

A utilidade das funções lambda vem quando combinada com outras funcionalidades do Python, por exemplo, em combinação com , e .map()filter()reduce()

31. Recursão

Uma função em Python pode se chamar. Isso é o que é recursão. E pode ser bastante útil em muitos cenários.

A maneira comum de explicar a recursão é usando o cálculo factorial.

O factorial de um número é o número mutado por , multiplicado por ... e assim por diante, até atingir o número: $n \cdot (n-1) \cdot (n-2) \cdots 1$

$3! = 3 * 2 * 1 = 6$

$4! = 4 * 3 * 2 * 1 = 24$

$5! = 5 * 4 * 3 * 2 * 1 = 120$

Usando a recursão podemos escrever uma função que calcula o fatorial de qualquer número:

```
def factorial(n):
    if n == 1: return 1
    return n * factorial(n-1)

print(factorial(3)) #   6
print(factorial(4)) #  24
print(factorial(5)) # 120
```

Se dentro da função você chamar em vez de , você vai causar uma recursão infinita. O Python por padrão interromperá as recursões em 1000 chamadas e, quando esse limite for atingido, você receberá um
erro.factorial()factorial(n)factorial(n-1)RecursionError

A recursão é útil em muitos lugares e nos ajuda a simplificar nosso código quando não há outra maneira ideal de fazê-lo, por isso é bom conhecer essa técnica.

32. Funções aninhadas

Funções em Python podem ser aninhadas dentro de outras funções.

Uma função definida dentro de uma função é visível apenas dentro dessa função.

Isso é útil para criar utilitários que são úteis para uma função, mas não úteis fora dela.

Você pode perguntar: por que eu deveria estar "escondendo" essa função, se ela não prejudica?

Primeiro, porque é sempre melhor ocultar a funcionalidade que é local para uma função e não é útil em outro lugar.

Além disso, porque podemos fazer uso de fechamentos (mais sobre isso mais tarde).

Aqui está um exemplo:

```
def talk(phrase):
    def say(word):
        print(word)

    words = phrase.split(' ')
    for word in words:
        say(word)

talk('I am going to buy the milk')
```

Se você quiser acessar uma variável definida na função externa a partir da função interna, primeiro você precisa declará-la como:`nonlocal`

```
def count():
    count = 0

    def increment():
        nonlocal count
        count = count + 1
        print(count)

increment()

count()
```

Isso é útil especialmente com fechamentos, como veremos mais adiante.

33. Encerramentos

Se você retornar uma função aninhada de uma função, essa função aninhada terá acesso às variáveis definidas nessa função, mesmo que essa função não

esteja mais ativa.

Aqui está um exemplo de contador simples.

```
def counter():
    count = 0

    def increment():
        nonlocal count
        count = count + 1
        return count

    return increment

increment = counter()

print(increment()) # 1
print(increment()) # 2
print(increment()) # 3
```

Nós retornamos a função interna, e que ainda tem acesso ao estado da variável mesmo que a função tenha terminado.`increment()`

34. Decoradores

Os decoradores são uma maneira de mudar, melhorar ou alterar de qualquer forma como uma função funciona.

Os decoradores são definidos com o símbolo seguido do nome do decorador, pouco antes da definição da função.`@`

Exemplo:

```
@logtime
def hello():
    print('hello!')
```

Esta função tem o decorador atribuído `hellologtime`

Sempre que ligarmos, o decorador vai ser chamado `hello()`

Um decorador é uma função que toma uma função como parâmetro, envolve a função em uma função interna que executa o trabalho que ela tem que fazer e retorna essa função interna. Em outras palavras:

```
def logtime(func):
    def wrapper():
        # do something before
        val = func()
        # do something after
        return val
    return wrapper
```

35. Docstrings

A documentação é extremamente importante, não apenas para comunicar a outras pessoas qual é o objetivo de uma função / classe / método / módulo, mas também para si mesmo.

Quando você voltar ao seu código daqui a 6 ou 12 meses, talvez não se lembre de todo o conhecimento que está segurando em sua cabeça, e ler seu código e entender o que ele deve fazer será muito mais difícil.

Os comentários são uma maneira de fazê-lo:

```
# this is a comment

num = 1 #this is another comment
```

Outra maneira é usar **docstrings**.

A utilidade dos docstrings é que eles seguem convenções e, como tal, podem ser processados automaticamente.

É assim que você define um docstring para uma função:

```
def increment(n):
    """Increment a number"""
    return n + 1
```

É assim que você define um docstring para uma classe e um método:

```
class Dog:
    """A class representing a dog"""

    def __init__(self, name, age):
        """Initialize a new dog"""

        self.name = name
        self.age = age

    def bark(self):
        """Let the dog bark"""
        print('WOF!')
```

Documente um módulo colocando um docstring na parte superior do arquivo, por exemplo, supondo que isso seja:dog.py

```
"""Dog module
```

This module does ... bla bla bla and provides the following classes:

- Dog

- ...

- """

```
class Dog:
    """A class representing a dog"""

    def __init__(self, name, age):
        """Initialize a new dog"""

        self.name = name
        self.age = age
```

```
def bark(self):
    """Let the dog bark"""
    print('WOF!')
```

Docstrings podem abranger várias linhas:

```
def increment(n):
    """Increment
    a number
    """
    return n + 1
```

Python irá processá-los e você pode usar a função global para obter a documentação para uma classe / método / função / módulo.`help()`

Por exemplo, chamar `help(increment)`

```
Help on function increment in module
__main__:

increment(n)
    Increment
    a number
```

Existem muitos padrões diferentes para formatar docstrings, e você pode optar por aderir ao seu favorito.

Eu gosto do padrão do Google:

<https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings>

O padrão permite ter ferramentas para extrair docstrings e gerar automaticamente a documentação para o seu código.

36. Introspecção

Funções, variáveis e objetos podem ser analisados usando **introspecção**.

Primeiro, usando a função global, podemos obter a documentação se fornecida na forma de docstrings.`help()`

Em seguida, você pode usar `print()` para obter informações sobre uma função:

```
def increment(n):
    return n + 1

print(increment)

# <function increment at 0x7f420e2973a0>
```

ou um objeto:

```
class Dog():
    def bark(self):
        print('WOF!')

roger = Dog()

print(roger)

# <__main__.Dog object at 0x7f42099d3340>
```

A função nos dá o tipo de um objeto:`type()`

```
print(type(increment))
# <class 'function'>

print(type(roger))
# <class '__main__.Dog'>

print(type(1))
```

```
# <class 'int'>

print(type('test'))
# <class 'str'>
```

A função global nos permite descobrir todos os métodos e atributos de um objeto:`dir()`

```
print(dir(roger))
```

```
# ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
# '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
# '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
# '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
# '__repr__', '__setattr__', '__sizeof__', '__str__',
# '__subclasshook__', '__weakref__', 'bark']
```

A função global nos mostra a localização na memória de qualquer objeto:`id()`

```
print(id(roger)) # 140227518093024
print(id(1))     # 140227521172384
```

Pode ser útil verificar se duas variáveis apontam para o mesmo objeto.

O módulo de biblioteca padrão nos dá mais ferramentas para obter informações sobre objetos, e você pode conferir aqui:
<https://docs.python.org/3/library/inspect.html#inspect>

37. Anotações

Python é digitado dinamicamente. Não precisamos especificar o tipo de uma variável ou parâmetro de função, ou um valor de retorno de função.

As anotações nos permitem (opcionalmente) fazer isso.

Esta é uma função sem anotações:

```
def increment(n):  
    return n + 1
```

Esta é a mesma função com anotações:

```
def increment(n: int) -> int:  
    return n + 1
```

Você também pode anotar variáveis:

```
count: int = 0
```

O Python ignorará essas anotações. Uma ferramenta separada chamada `mypy` pode ser executada de forma autônoma ou integrada pelo IDE, como o VS Code ou o PyCharm, para verificar automaticamente se há erros de tipo estaticamente, enquanto você está codificando, e ajudará você a detectar bugs de incompatibilidade de tipo antes mesmo de executar o código.

Uma grande ajuda, especialmente quando seu software se torna grande e você precisa refatorar seu código.

38. Exceções

É importante ter uma maneira de lidar com erros.

Python nos dá tratamento de exceções.

Se você encapsular linhas de código em um bloco:try:

```
try:  
    # some lines of code
```

Se ocorrer um erro, o Python irá alertá-lo e você pode determinar que tipo de erro ocorreu usando um bloco:`except`

```
try:  
    # some lines of code  
except <ERROR1>:  
    # handler <ERROR1>  
except <ERROR2>:  
    # handler <ERROR2>
```

Para capturar todas as exceções que você pode usar sem qualquer tipo de erro:`except`

```
try:  
    # some lines of code  
except <ERROR1>:  
    # handler <ERROR1>  
except:  
    # catch all other exceptions
```

O bloco será executado se nenhuma exceção for encontrada:`else`

```
try:  
    # some lines of code  
except <ERROR1>:  
    # handler <ERROR1>  
except <ERROR2>:  
    # handler <ERROR2>  
else:  
    # no exceptions were raised, the code ran successfully
```

Um bloco permite que você execute alguma operação em qualquer caso, independentemente de ter ocorrido um erro ou não:`finally`

```
try:  
    # some lines of code  
except <ERROR1>:  
    # handler <ERROR1>  
except <ERROR2>:  
    # handler <ERROR2>  
else:  
    # no exceptions were raised, the code ran successfully  
finally:  
    # do something in any case
```

O erro específico que ocorrerá depende da operação que você está executando.

Por exemplo, se você estiver lendo um arquivo, poderá obter um arquivo . Se você dividir um número por zero, você receberá um . Se você tiver um problema de conversão de tipo, poderá obter um arquivo .
.EOFErrorZeroDivisionErrorTypeError

Tente este código:

```
result = 2 / 0  
print(result)
```

O programa será encerrado com um erro

```
Traceback (most recent call last):  
  File "main.py", line 1, in <module>  
    result = 2 / 0  
ZeroDivisionError: division by zero
```

e as linhas de código após o erro não serão executadas.

Adicionar essa operação em um bloco nos permite recuperar graciosamente e seguir em frente com o programa:try:

```
try:  
    result = 2 / 0  
except ZeroDivisionError:  
    print('Cannot divide by zero!')  
finally:  
    result = 1  
  
print(result) # 1
```

Você também pode gerar exceções em seu próprio código, usando a instrução: `raise`

```
raise Exception('An error occurred!')
```

Isso gera uma exceção geral, e você pode interceptá-la usando:

```
try:  
    raise Exception('An error occurred!')  
except Exception as error:  
    print(error)
```

Você também pode definir sua própria classe de exceção, estendendo-se de `Exception`:

```
class DogNotFoundException(Exception):  
    pass
```

`pass` aqui significa "nada" e devemos usá-lo quando definimos uma classe sem métodos, ou uma função sem código, também.

```
try:  
    raise DogNotFoundException()  
except DogNotFoundException:  
    print('Dog not found!')
```

39. A declaração with

A declaração é muito útil para simplificar o trabalho com o tratamento de exceções.`with`

Por exemplo, ao trabalhar com arquivos, cada vez que abrimos um arquivo, devemos nos lembrar de fechá-lo.

`with` torna esse processo transparente.

Em vez de escrever:

```
filename = '/Users/flavio/test.txt'

try:
    file = open(filename, 'r')
    content = file.read()
    print(content)
finally:
    file.close()
```

Você pode escrever:

```
filename = '/Users/flavio/test.txt'

with open(filename, 'r') as file:
    content = file.read()
    print(content)
```

Em outras palavras, temos um tratamento de exceção implícito embutido, como será chamado automaticamente para nós.`close()`

`with` não é apenas útil para trabalhar com arquivos. O exemplo acima destina-se apenas a introduzir suas capacidades.

40. Instalando pacotes de terceiros usando pip

A biblioteca padrão Python contém um grande número de utilitários que simplificam nossas necessidades de desenvolvimento Python, mas nada pode satisfazer *tudo*.

É por isso que indivíduos e empresas criam pacotes e os disponibilizam como software de código aberto para toda a comunidade.

Esses módulos são todos coletados em um único lugar, o **Python Package Index** disponível em <https://pypi.org>, e eles podem ser instalados em seu sistema usando o .`pip`

Existem mais de 270.000 pacotes disponíveis gratuitamente, no momento da redação.

Você já deve ter instalado se seguiu as instruções de instalação do `Python.pip`

Instale qualquer pacote usando o comando :`pip install`

```
pip install <package>
```

ou, se você tiver problemas, você também pode executá-lo através de:`python -m`

```
python -m pip install <package>
```

Por exemplo, você pode instalar o pacote de solicitações, uma biblioteca HTTP popular:

```
pip install requests
```

e uma vez que você fizer isso, ele estará disponível para todos os seus scripts Python, porque os pacotes são instalados globalmente.

A localização exata depende do seu sistema operacional.

No macOS, executando o Python 3.9, o local é
./Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages

Atualize um pacote para sua versão mais recente usando:

```
pip install -U <package>
```

Instale uma versão específica de um pacote usando:

```
pip install <package>==<version>
```

Desinstale um pacote usando:

```
pip uninstall <package>
```

Mostre os detalhes de um pacote instalado, incluindo versão, site de documentação e informações do autor usando:

```
pip show <package>
```

41. Listar comprehensões

As comprehensões de listas são uma maneira de criar listas de uma maneira muito concisa.

Suponha que você tenha uma lista:

```
numbers = [1, 2, 3, 4, 5]
```

Você pode criar uma nova lista usando uma compreensão de lista, composta pelos elementos da lista, poder 2:numbers

```
numbers_power_2 = [n**2 for n in numbers]
# [1, 4, 9, 16, 25]
```

List comprehensions are a syntax that's sometimes preferred over loops, as it's more readable when the operation can be written on a single line:

```
numbers_power_2 = []
for n in numbers:
    numbers_power_2.append(n**2)
```

and over :map()

```
numbers_power_2 = list(map(lambda n : n**2, numbers))
```

42. Polymorphism

Polymorphism generalizes a functionality so it can work on different types. It's an important concept in object-oriented programming.

We can define the same method on different classes:

```
class Dog:
    def eat():
        print('Eating dog food')
```

```
class Cat:
    def eat():
        print('Eating cat food')
```

Then we can generate objects and we can call the method regardless of the class the object belongs to, and we'll get different results:eat()

```
animal1 = Dog()  
animal2 = Cat()  
  
animal1.eat()  
animal2.eat()
```

We built a generalized interface and we now do not need to know that an animal is a Cat or a Dog.

43. Operator Overloading

Operator overloading is an advanced technique we can use to make classes comparable and to make them work with Python operators.

Let's take a class Dog:

```
class Dog:  
    # the Dog class  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Let's create 2 Dog objects:

```
roger = Dog('Roger', 8)  
syd = Dog('Syd', 7)
```

We can use operator overloading to add a way to compare those 2 objects, based on the property:age

```
class Dog:  
    # the Dog class  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

```
def __gt__(self, other):
    return True if self.age > other.age else False
```

Now if you try running you will get the result `.print(roger > syd)`True

In the same way we defined (which means greater than), we can define the following methods: `__gt__()`

- `__eq__()` to check for equality
- `__lt__()` to check if an object should be considered lower than another with the operator<
- `__le__()` for lower or equal (`<=`)
- `__ge__()` for greater or equal (`>=`)
- `__ne__()` for not equal (`!=`)

Then you have methods to interoperate with arithmetic operations:

- `__add__()` respond to the operator`+`
- `__sub__()` respond to the operator`-`
- `__mul__()` respond to the operator`*`
- `__truediv__()` respond to the operator`/`
- `__floordiv__()` respond to the operator`//`
- `__mod__()` respond to the operator`%`
- `__pow__()` respond to the operator`**`
- `__rshift__()` respond to the operator`>>`
- `__lshift__()` respond to the operator`<<`
- `__and__()` respond to the operator`&`
- `__or__()` respond to the operator`|`
- `__xor__()` respond to the operator`^`

There are a few more methods to work with other operators, but you got the idea.

44. Virtual Environments

It's common to have multiple Python applications running on your system.

When applications require the same module, at some point you will reach a tricky situation where an app needs a version of a module, and another app a different version of that same module.

To solve this, you use **virtual environments**.

We'll use `.venv`. Other tools work similarly, like `.venv` `pipenv`

Create a virtual environment using

```
python -m venv .venv
```

in the folder where you want to start the project, or where you already have an existing project.

Then run

```
source .venv/bin/activate
```

Use on the Fish shell `source .venv/bin/activate.fish`

A execução do programa ativará o ambiente virtual Python. Dependendo da sua configuração, você também pode ver o prompt do terminal mudar.

O meu mudou de

→ folder

Para

(`.venv`) → folder

Agora, a execução usará esse ambiente virtual em vez do ambiente global `pip`