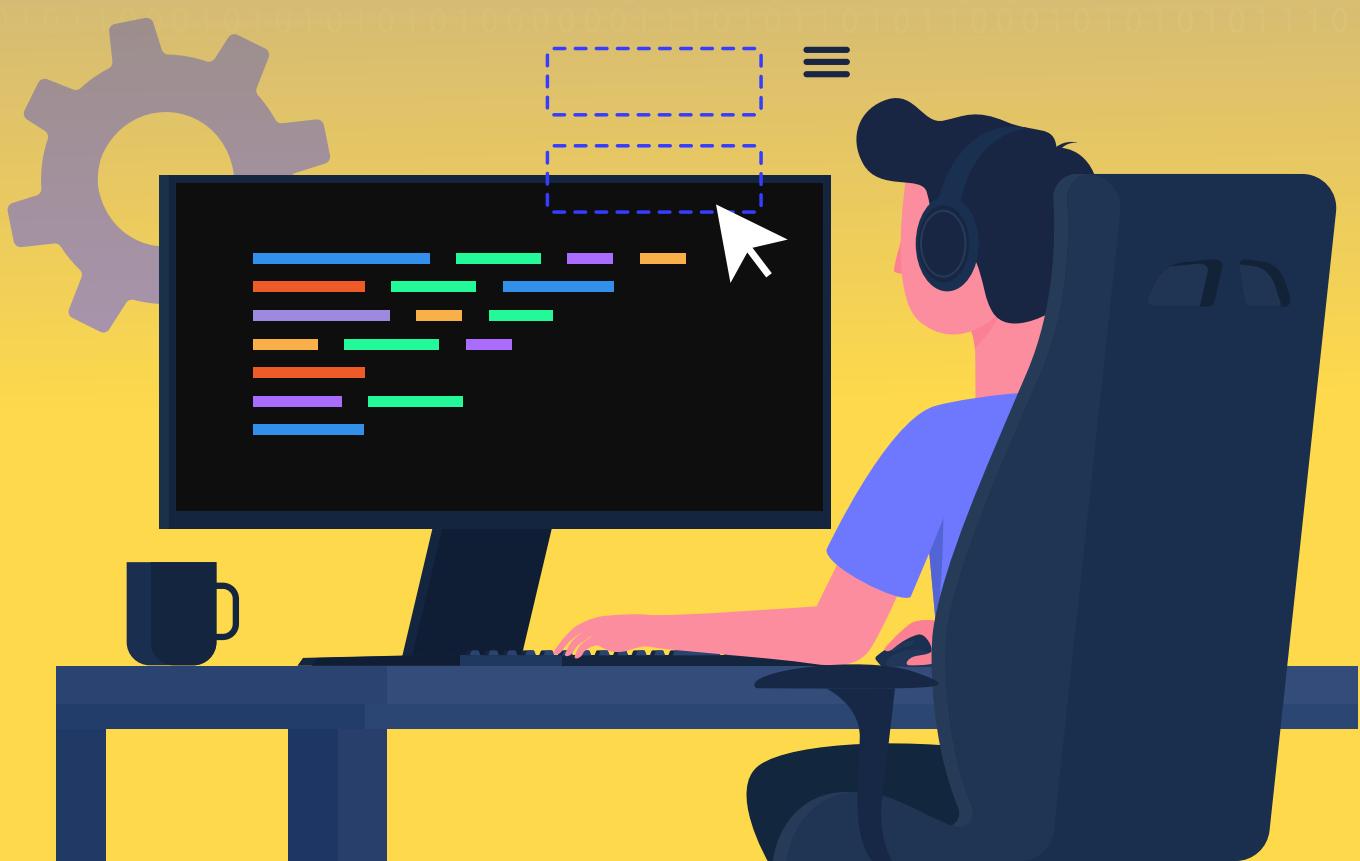




Começando com Python

Uma introdução acessível e prática
ao mundo da linguagem de
programação Python.

E-BOOK GRATUITO



SUMÁRIO INTERATIVO

Clique e seja redirecionado para a página



Introdução	04
O que é Python?	05
Características de tipagem do Python	06
Principais aplicações do Python	07
Instalação do Python e nosso primeiro Olá Mundo	09
Como instalar o Python no Windows	09
Como instalar o Python no Linux	10
Como instalar o pip	12
Como instalar o venv	12
Como instalar o Python no MacOS	13
Criando nosso primeiro Olá Mundo	13
Vamos executar nosso Olá Mundo	15
Principais IDEs para desenvolvimento Python	16
Conhecendo variáveis e constantes no Python	18
Tipos de Dados no Python	18
Regras de nomeação de variáveis	19
Declarando variáveis	20
Declarando constantes	20
Estruturas condicionais e estruturas de repetição em Python	21
Estruturas de condição	21
Estruturas de repetição	23
Orientação a objetos em Python	27
Declarando classes	27
Criando construtor da classe	28
Instanciando objetos	28
Declarando métodos	30
Declarando propriedades	31
Principais Estruturas de Dados no Python	36
Listas	36
Declarando Listas	36
Tuplas	37
Tuplas x Listas	37
Declarando Tuplas	38
Sets	38
Declarando Sets	39

SUMÁRIO INTERATIVO

Clique e seja redirecionado para a página



Dicionários	39
Declarando Dicionários	40
Como instalar um pacote com PIP e utilizá-lo em seu projeto	42
Afinal, o que é um Gerenciador de Pacotes?	42
Onde encontrar os pacotes?	42
Como definir os pacotes no projeto e instalar	44
Gerenciando pacotes em projetos Python com o PIP	45
O que é o PIP?	45
Gerenciando pacotes	46
Criando ambientes virtuais para projetos Python com o Virtualenv	48
O que é uma virtualenv?	48
Como funciona uma virtualenv?	49
Instalando a virtualenv	50
Criando uma nova virtualenv	50
Ativando uma virtualenv	51
Desativando uma virtualenv	52
Instalando pacotes	52
O ponto final ou apenas o começo?	54
Conheça a TreinaWeb	54

Introdução

Vamos abordar nesse ebook uma das linguagens de programação mais versáteis que temos: o **Python**.

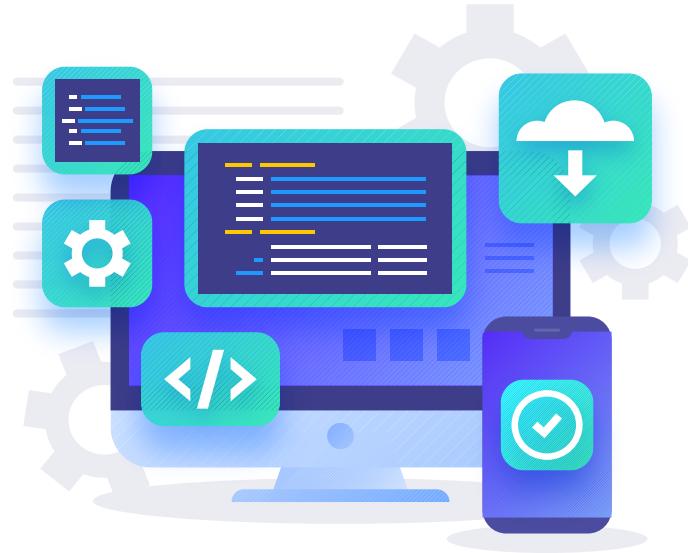
O Python é uma linguagem muito interessante principalmente por conta da sua polivalência, já que é possível utilizá-la nas áreas de Big Data, Machine Learning, Análise de Dados, desenvolvimento web e muitas outras, áreas essas que estão em constante crescimento nos últimos anos e tendem a continuar assim.

Por isso, se você tem interesse em desenvolvimento, é muito importante que você conheça a linguagem Python - **e você está no lugar certo**. Criamos este ebook para abordar conceitos fundamentais para que você possa começar seus estudos com Python da maneira correta, dando os primeiros passos para criar uma base sólida, o que te ajudará muito lá na frente.

Então, vamos iniciar neste mundo do Python?

Vamos lá!

O que é Python?



Criado por Guido Van Rossum em meados dos anos 90, o Python é uma linguagem de programação de alto nível, multiplataforma e open source, que ganhou destaque nos últimos anos por possuir uma ótima curva de aprendizagem, ser uma linguagem leve e ter se tornado uma das principais linguagens para o desenvolvimento de IA, Machine Learning e Big Data, áreas em grande crescimento nos últimos anos.

Python é uma linguagem de programação multiparadigma, ou seja, suporta diversos paradigmas de desenvolvimento, como orientado a objetos, funcional, imperativo, entre outros. Desta forma, com o Python um único programa poderá ser desenvolvido utilizando mais de um paradigma.

O Python possui as seguintes características:

Open Source

Multiplataforma

Linguagem Interpretada

Tipagem Dinâmica

Tipagem Forte

Multiparadigma

Multinicho

Além disso, a linguagem Python possui uma curva de aprendizado muito pequena e é bastante reconhecida pela sua comunidade diversa, acolhedora e bastante ativa.

Características de tipagem do Python

Toda linguagem de programação possui uma tipagem, que é basicamente um conjunto de regras que as define como: Tipagem estática ou dinâmica e forte ou fraca.

O Python é uma das principais linguagens que possui tipagem dinâmica. A tipagem dinâmica é a característica que muitas linguagens de programação possuem por não exigirem que os tipos de dados sejam declarados, pois são capazes de realizar esta escolha dinamicamente. Desta forma, durante a execução do programa ou até mesmo durante a sua compilação, o tipo de uma variável poderá ser alterado.

Além disso, o Python possui como característica a tipagem forte, ou seja, a linguagem não realiza conversões automaticamente entre os tipos suportados.

Principais aplicações do Python

Quando decidimos iniciar uma nova linguagem, nada melhor que conhecer as suas aplicações, para assim, decidir se esta escolha nos levará ao que procuramos.

Com o Python não é diferente, e dentre suas principais aplicações podemos citar:

Data Science

Data Science ou Ciência de dados, é uma área voltada ao estudo da análise de dados, ou seja, dada uma quantidade de dados, as análises dos mesmo e as conclusões que serão tiradas a partir da sua extração tem se tornado bastante importante para diversas empresas. Nesta área, o Python é uma das principais tecnologias. Muito disso por conta de sua simplicidade e bibliotecas para trabalhar com análise de dados, como o Pandas, uma das principais e mais poderosas do mercado.

Machine Learning

Machine Learning ou “Aprendizado de máquina”, é a área da ciência da computação que tem como objetivo a análise de dados que automatiza a construção de modelos analíticos. Assim como o Big Data, o Python é uma das principais tecnologias para trabalhar com Machine Learning, uma área de grande crescente na última década.

Big Data

Big Data é a análise e interpretação de grandes volumes de dados. Assim como o Data Science e Machine Learning, o Python também está pronto e possui diversas bibliotecas para trabalhar com Big Data, já que estas áreas estão estreitamente relacionadas. Várias são as bibliotecas para trabalhar com Big Data no Python, como a Pandas (citada anteriormente), NumPy, Matplotlib, Scikit-Learn, entre outras.

Desenvolvimento Web (Django e Flask)

Desenvolvimento web é o termo utilizado para definir sites ou aplicativos que podem ser acessados diretamente pelo navegador web, seja em computadores ou dispositivos móveis. Além de todas as outras aplicações do Python citadas anteriormente, a linguagem ainda é uma ótima alternativa para a criação de aplicações web de forma simples e poderosa. Para isso, o Python conta com dois dos principais frameworks para desenvolvimento web do mercado: o Django e o Flask. Com esses dois frameworks, podemos desenvolver aplicações web poderosas e que, com certeza, irão atender todas as demandas do mercado.

Instalação do Python e nosso primeiro Olá Mundo

Agora que já sabemos o que é o Python e suas principais características, estamos prontos para realizar a instalação e executar o nosso primeiro exemplo com a linguagem.

Como instalar o Python no Windows

A instalação do Python no Windows segue o padrão da maioria dos programas instalados no sistema operacional em questão (next, next, next, finish) com uma única ressalva: no início do processo de instalação, deve-se selecionar a opção “Add Python 3.8 to PATH”. Com isso, o Windows saberá onde está localizado o interpretador do Python e, assim, conseguiremos utilizá-lo sem problemas.

Acesse a página oficial para realizar o download do instalador do Python na versão 3.8.

Vá até a pasta na qual foi feito o download do instalador do Python 3.8.

Clique com o botão direito em cima do instalador.

Clique na opção “Executar como Administrador”.

Com o instalador aberto tenha a certeza de ter marcado as opções “Add Python 3.8 to PATH” para que o comando python fique disponível.

Por fim clique em “Install Now” e siga o processo padrão de instalação de programas no Windows (next, next, next, finish).



Para verificar se a instalação foi realizada com sucesso basta abrir algum terminal do Windows (Prompt de Comando ou Power Shell) e digitar o comando abaixo:

```
python --version
```

Caso nenhum erro seja exibido, isso significa que a instalação do Python foi realizada com sucesso.

Como instalar o Python no Linux

A instalação do Python em ambientes Linux também é bem simples. Por padrão, o Python já vem instalado nos sistemas baseados em Debian (como o Ubuntu e o Mint), porém dependendo da versão do seu sistema você terá uma versão diferente do Python. Sendo assim,

primeiramente devemos verificar se o Python está instalado na nossa máquina e em qual versão. Para isso, execute o comando abaixo para verificar a existência (ou não) do Python 3.8:

```
python3 --version
```

Ao executar o comando acima, será retornado a versão do Python 3 instalado em sua máquina. Caso seja retornado algum erro, isso indica que o interpretador do Python ainda não está instalado.

Caso seja retornada uma versão inferior ao Python 3.8 ficando sendo sua decisão instalar uma versão mais recente ou não, qualquer versão superior ou igual a versão 3.6 já é o suficiente para desenvolver seus projetos utilizando a grande maioria dos recursos da linguagem. Sendo assim, caso necessite realizar a instalação, basta executar o seguinte comando:

```
sudo apt install python3.8
```

Este comando irá instalar o interpretador do Python em sua versão 3.8.x. Caso você já tenha uma versão do Python instalada no seu Linux e mesmo assim optou por instalar a versão 3.8, o seu sistema operacional terá duas versões do Python, uma versão é o padrão do sistema que pode ser acessado através do comando `python3` e a outra versão é a que foi instalada por você que será acessível através do comando `python3.8`.

Como instalar o pip

Além disso precisamos instalar o pip, pois diferente dos assistentes de instalação do Python para os sistemas Windows e MacOS ao instalar o Python via apt no Linux a ferramenta pip não é instalada em conjunto.

O pip é um gerenciador de pacotes para projetos Python, é através dele que podemos instalar, remover e atualizar pacotes em nossos projetos. Para realizar a instalação do pip execute o comando abaixo:

```
sudo apt install python3-pip
```

Como instalar o venv

Caso você esteja em um sistema derivado do Debian como o Ubuntu por exemplo, é necessário instalar os binários da biblioteca venv caso você queira utilizar ambientes virtuais em seu ambiente de desenvolvimento.

Para realizar a instalação dos binários da biblioteca venv basta executar o comando abaixo:

```
sudo apt install python3-venv
```

Como instalar o Python no MacOS

A instalação do Python em ambientes MacOS segue a mesma ideia do Windows, onde o processo de instalação é o “padrão” (next, next, next, finish), para realizar o download do instalador acesse o site oficial do Python. Ao final do processo, podemos abrir o terminal do Mac e com o comando abaixo, verificar se a instalação foi feita com sucesso.

```
python3 --version
```

Caso nenhum erro seja exibido, significa que a instalação do Python foi realizada com sucesso.

Criando nosso primeiro Olá Mundo

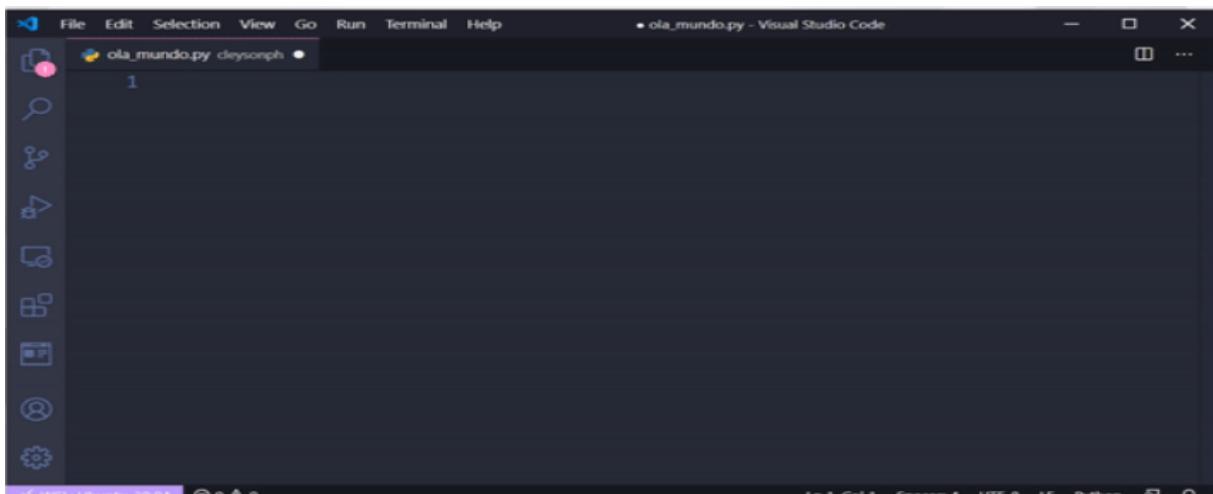
Agora que já temos o Python instalado em nossa máquina, vamos escrever nosso primeiro código em Python que será um simples **Olá Mundo**. Esse simples código que vai exibir a mensagem “Olá Mundo” tem como objetivo principal verificar se nosso ambiente de desenvolvimento está configurado de maneira adequada.

Para escrevermos nosso código vamos precisar de algum editor de código fonte ou uma IDE. Aqui iremos utilizar o editor de código Visual Studio Code, mas veremos mais a frente sobre IDEs.

Para desenvolver um código em Python é necessário criar um arquivo com a extensão **.py**. Então, para escrever o nosso código irei criar um arquivo chamado **ola_mundo.py** e abri-lo no VS Code, sendo possível realizar ambas as ações com um único comando:

```
code ola_mundo.py
```

Após executar o comando acima o VS Code irá abrir já com o arquivo criado e aberto para que possamos escrever nosso código.



Na linguagem Python quando queremos exibir alguma mensagem na tela durante a execução do nosso código utilizamos a função **print** e passamos como parâmetro dessa função uma String contendo a mensagem a ser exibida.

```
print("Olá Mundo")
```

E com uma única linha de código o nosso Olá Mundo já está pronto, essa é uma das grandes vantagens do Python, ele nos permite desenvolver nossas aplicações com pouquíssimas linhas de código e não precisamos colocar o ponto e vírgula no final da instrução.

Vamos executar nosso Olá Mundo

Agora que nosso código já está pronto precisamos “pedir” para o interpretador do Python executar o nosso arquivo `ola_mundo.py`, para isso execute o comando abaixo no seu terminal, mas antes de executar o comando não esqueça de salvar o arquivo.

```
python3 ola_mundo.py
```

Caso você esteja no Windows deverá utilizar o comando `python` ao invés de `python3`. Pronto, com isso nosso código será executado e a mensagem “Olá Mundo” será exibida no seu terminal.

A screenshot of a terminal window titled "Terminal". The window shows a command-line interface with a dark background and light-colored text. The user has entered the command "python3 ola_mundo.py" and pressed Enter. The terminal then displays the output "Olá Mundo", which is the text printed by the Python script. The cursor is currently at the end of the line where the command was typed.

```
python3 ola_mundo.py
Olá Mundo
```

Principais IDEs para desenvolvimento Python

Para facilitar o desenvolvimento de programas com o Python, a melhor maneira é utilizando IDEs. São elas que proporcionam mecanismos que facilitam toda a escrita, testes e execução do nosso código.

IDE ou **Integrated Development Environment** (*Ambiente de Desenvolvimento Integrado*) é um software que auxilia no desenvolvimento de aplicações, muito utilizado por desenvolvedores com o objetivo de facilitar diversos processos (ligados ao desenvolvimento), que combinam ferramentas comuns em uma única interface gráfica do usuário (GUI). Desta forma, como principais IDEs para desenvolvimento Python podemos citar:

Eclipse

O Eclipse é uma excelente IDE, muito utilizada no mercado. Seu uso facilita a criação de aplicações Python tanto para Desktop ou Web. O download do Eclipse poderá ser realizado em seu próprio site.

PyCharm

PyCharm conta com desenvolvimento multitecnologias, onde além do Python, oferece suporte para CoffeeScript, TypeScript, Cython, JavaScript, SQL, HTML/CSS, linguagens de modelo, AngularJS, Node.js e muitas outras. O download do PyCharm é feito em seu próprio site, onde é possível acompanhar todas as suas novidades, recursos, suporte e muito mais.

Jupyter Notebook

Criada em 2014, derivado do IPython, o Jupyter Notebook é baseada na estrutura servidor-cliente, que permite a manipulação de documentos. O Jupyter Notebook independe de linguagem e suporta diversos ambientes de execução, entre elas: Julia, R, Haskell, Ruby e o próprio Python. Para instalar o Jupyter Notebook basta acessar o seu site, onde você também encontrará toda a sua documentação, blog, novidades e muito mais.

Spyder

O Spyder é outra opção para desenvolvedores Python, muito utilizado principalmente por cientistas de dados, já que possui integração com as principais bibliotecas como NumPy, SciPy, Matplotlib e IPython. O download do Spyder poderá ser feito em seu site, onde também é possível verificar seus plugins, componentes e muito mais.

Conhecendo variáveis e constantes no Python

Tipos de Dados no Python

Como vimos anteriormente, o Python possui tipagem dinâmica, ou seja, suas variáveis podem armazenar dados de diferentes tipos. No Python os tipos suportados são:

Inteiro

O tipo inteiro é representado por toda e qualquer informação numérica que pertença ao conjunto de números inteiros relativos (números positivos, negativos ou o zero). Exemplo:

```
inteiro = 15  
print(type(inteiro)) # <class 'int'>
```

Float

O tipo float é representado por números decimais, ou seja, números que possuem partes fracionadas. Exemplo:

```
ponto_flutuante = 25.10  
print(type(ponto_flutuante)) # <class 'float'>
```

String

String é uma cadeia de caracteres que representam textos. Exemplo:

```
nome = "Treinaweb"  
print(type(nome)) # <class 'str'>
```

Booleano

Variáveis booleanas armazenam valores lógicos que podem ser verdadeiro ou falso. Exemplo:

```
booleano = True  
print(type(booleano)) # <class 'bool'>
```

Tipo complexo

As variáveis de tipo complexo armazenam dados com formato misto, ou seja, dados de diferentes tipos em uma mesma sentença. Exemplo:

```
complexo = 25 + 3j  
print(type(complexo)) # <class 'complex'>
```

Regras de nomeação de variáveis

A linguagem de programação Python possui como regra de nomeação de variáveis o formato `snake_case`.

Este formato determina que o nome das variáveis possuam um padrão em que as palavras sejam definidas em letras minúsculas e, caso possua, os espaços serão substituídos por “underline” conforme podemos verificar abaixo:

```
variavel_com_nome_composto = "Treinaweb"  
print(variavel_com_nome_composto) # Treinaweb
```

Declarando variáveis

A declaração de variáveis no Python é um processo muito simples. Por não possuir um escopo padrão para seus scripts, basta definir o nome da variável e, em seguida, atribuir o valor desejável:

```
nome_da_variavel = "valor_da_variavel"
```

Declarando constantes

A regra de nomeação das constantes no Python segue um padrão parecido com as de variáveis, com a diferença de que todas as letras são maiúsculas e separadas por underline “_”. Porém, por possuir tipagem dinâmica, os valores atribuídos à constantes podem ser alterados sem problemas:

```
MINHA_CONSTANTE = 10  
print(MINHA_CONSTANTE) # 10  
  
MINHA_CONSTANTE = 15  
print(MINHA_CONSTANTE) # 10
```

Estruturas condicionais e estruturas de repetição em Python

Estruturas de condição

Estruturas de condição são artifícios das linguagens de programação para determinar qual bloco de código será executado a partir de uma determinada condição. No Python, assim como em outras linguagens, podemos trabalhar com as estruturas de condição utilizando o if/else como veremos abaixo.

if/else

O if e o else são comandos que verificam determinada condição na programação. O uso do if em um programa em Python visa verificar se determinada ação é verdadeira e executar o bloco de código contido em seu escopo. Basicamente é feita da seguinte forma:

```
media = 7
if media > 6.9:
    print ("Você foi aprovado")
```

No exemplo do código acima, utilizamos apenas o if para verificar se a variável **media** é maior que 6.9. Como esta condição é verdadeira, imprimimos a mensagem na tela “**Você foi aprovado**”. Caso esta condição fosse falsa, o código seguiria normalmente ignorando, desta forma, a linha 3, o nosso print.

Já o uso o **if/else** fará com que uma das ações sejam executadas, já que se a condição dentro do **if** não for verdadeira, será executado o código contido no **else**. O **if/else** irá testar caso a condição seja verdadeira e executar uma determinada ação ou caso a mesma não seja executar outra.

```
media = 7
if media < 6.9:
    print ("Você foi reprovado")
else:
    print ("Você foi aprovado")
```

O código acima contém dois códigos a serem executados. Caso a **media** informada seja menor que 6.9, a pessoa será reprovada na disciplina, porém, caso a condição contida no **if** falhar, o código contido no **else** será executado. Desta forma, será exibido em tela que o aluno foi aprovado, já que a condição do **if** é falsa.

if...elif...else

O uso de **if/elif/else** serve para quando mais de uma condição precisar ser verificada. Imagine que possuímos duas condições: a primeira, se o aluno possuir uma média menor que 5 e a segunda, se a média for menor que 7 e maior que 5. Vimos anteriormente que utilizamos o **if** para testar se uma condição é verdadeira, porém, quando precisamos verificar uma segunda condição utilizamos o **elif** e adicionamos a condição a ser testada.

```
media = 7
if media < 5:
    print ("Você foi reprovado")
elif media > 5 and media < 7:
    print ("Você fará a recuperação")
else:
    print ("Você foi aprovado")
```

No código acima, o primeiro passo é verificar se a média do aluno é menor que 5 e, caso positivo, imprimir a mensagem que o mesmo foi reprovado. Porém, caso essa condição seja falsa, precisamos exibir se ele foi aprovado ou fará a recuperação. Para isso, utilizamos o **elif** para testar se a média está entre os valores 5 e 7 e, caso positivo, imprimir a mensagem "Você fará a recuperação". Se a condição contida no **if** e **elif** forem falsas, o código contido no **else** será executado e imprimirá a mensagem "Você foi aprovado".

Estruturas de repetição

Estruturas de repetição são artifícios das linguagens de programação para executar um determinado bloco de código por uma certa quantidade de vezes, seja ela definida (utilizando o **for**) ou a partir de uma condição (utilizando o **while**).

while

O **while** é uma estrutura de repetição que permite executar um determinado bloco de código enquanto uma condição for verdadeira. É muito similar ao **if**, com a diferença que o bloco será executado enquanto a condição for verdadeira, e não se a condição for verdadeira.

Para isso, após o comando **while** precisamos definir uma condição que será testada a cada execução do seu loop.

O **while** só será finalizado quando essa condição não for mais atendida. Imagine que estamos desenvolvendo um controle de gastos e que, enquanto os gastos não somarem R\$ 1000,00, nós poderemos adicionar novas contas. Este é um ótimo exemplo do uso do while, já que o bloco de código que será responsável por incrementar a quantidade dos gastos será executado enquanto a soma de todos os valores não for menor que 1000:

```
gastos = 0
valor_gasto = 0
while gastos < 1000:
    valor_gasto = int(input("Digite o valor do novo gasto"))
    gastos = gastos + valor_gasto

print(gastos)
```

O código acima irá executar o bloco contido no while enquanto sua condição for verdadeira, ou seja, enquanto a soma de gastos for menor que 1000, como podemos ver na imagem abaixo:

The screenshot shows a code editor interface with a tab labeled "main.py". The code itself is as follows:

```
1 gastos = 0
2 valor_gasto = 0
3 while gastos < 1000:
4     valor_gasto = int(input("Digite o valor
        do novo gasto: "))
5     gastos = gastos + valor_gasto
6 else:
7     print("Você chegou ao limite de gastos")
8 print(gastos)
```

To the right of the code, there is a "Run" button and a "Shell" output window. The "Shell" window displays the following interaction:

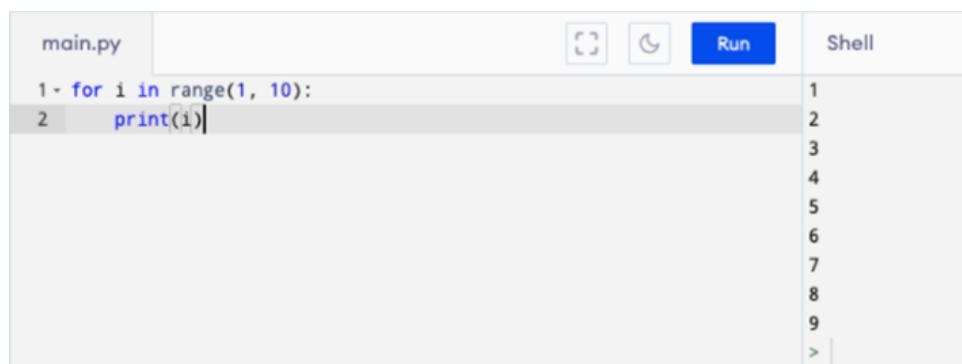
```
Digite o valor do novo gasto: 500
Digite o valor do novo gasto: 300
Digite o valor do novo gasto: 200
Você chegou ao limite de gastos
1000
>
```

for...in

Diferente do while, o **for** executará um determinado bloco de código por um número definido de vezes. Esta estrutura é muito útil quando já sabemos a quantidade de vezes que precisamos executar determinado bloco de código. Diferente da maioria das linguagens, para criar um intervalo de vezes que o **for** será executado, precisamos utilizar a função **range** e definir o intervalo, como podemos ver abaixo:

```
for i in range(1, 10):
    print(i)
```

O código acima irá executar o comando **print** enquanto o intervalo entre 1 e 10 não finalizar, como podemos ver na imagem abaixo:



The screenshot shows a code editor interface with a tab labeled "main.py". The code in the editor is:

```
1 - for i in range(1, 10):
2     print(i)|
```

To the right of the editor, there is a "Run" button and a "Shell" window. The "Shell" window displays the output of the code execution, showing the numbers 1 through 9, each on a new line.

Shell
1
2
3
4
5
6
7
8
9
>

for else e while else

O Python também permite adicionar o comando **else** depois de uma estrutura de repetição, seja ela um **for** ou um **while**. Este **else** serve para executar um determinado bloco de código imediatamente após a estrutura de repetição finalizar:

main.py		Run	Shell
1 - for i in range(1, 10): 2 print(i) 3 - else: 4 print("fim do loop")			1 2 3 4 5 6 7 8 9 fim do loop >

main.py		Run	Shell
1 gastos = 0 2 valor_gasto = 0 3 while gastos < 1000: 4 valor_gasto = int(input("Digite o valor do novo gasto: ")) 5 gastos = gastos + valor_gasto 6 - else: 7 print("você gastou demais!") 8 print(gastos) 9			Digite o valor do novo gasto: 300 Digite o valor do novo gasto: 500 Digite o valor do novo gasto: 200 você gastou demais! 1000 >

O **else** dos códigos acima irão imprimir a mensagem “fim do loop” e “você gastou demais” imediatamente após o **for** e o **while** se encerrarem.

Orientação a objetos em Python

O paradigma de programação orientado à objetos é um dos principais paradigmas das linguagens de programação. Muito utilizado no mercado, entender como funciona e como implementar este paradigma é essencial para todo desenvolvedor de software.

No Python, o paradigma orientado à objetos funciona de forma similar às outras linguagens, porém com algumas mudanças em sua implementação.

Declarando classes

No paradigma orientado à objetos, uma classe é a representação de algo do mundo real. No Python, o uso de classes é algo constante no desenvolvimento de programas. Sendo assim, para declarar uma classe no Python é bem simples, como podemos ver abaixo:

```
class Pessoa():
    # Atributos e métodos da classe
```

Como vimos acima, para declarar uma classe no Python, utilizamos a palavra reservada **class** seguido do nome desta classe.

No Python, todas as classes devem, por boas práticas, possuir nomes que comecem com letra maiúscula e, caso sejam compostos, a primeira letra de cada palavra deve ser maiúscula, o que chamamos de formato CamelCase:

```
class PessoaFisica():
    # Atributos e métodos da classe
```

Criando construtor da classe

Uma classe é representada por atributos e métodos. Os atributos de uma classe representam as características que esta classe possui, já os métodos representam o comportamento da classe.

Para declarar um atributo em uma classe no Python é bem simples, basta definir o nome do atributo no método especial chamado `__init__`, este método define o construtor da classe, ou seja, é onde definimos como uma nova pessoa será criada em nosso programa.

Para definir os atributos de uma classe em seu construtor, basta passá-los como parâmetro, como podemos ver abaixo:

```
class Pessoa:
    def __init__(self, nome, sexo, cpf):
        self.nome = nome
        self.sexo = sexo
        self.cpf = cpf
```

Agora, estamos indicando que toda pessoa que for criada em nosso programa e que utilize como base a classe `Pessoa` deverá possuir um nome, sexo e cpf.

Instanciando objetos

Como vimos anteriormente, as classes representam a estrutura de um elemento no mundo real, porém ela é apenas o modelo destes elementos.

Sempre que precisamos criar “algo” com base em uma classe, dizemos que estamos “instanciando objetos”. O ato de instanciar um objeto significa que estamos criando a representação de uma classe em nosso programa.

Para instanciar um objeto no Python com base em uma classe previamente declarada, basta indicar a classe que desejamos utilizar como base e, caso possua, informar os valores referentes aos seus atributos, como podemos ver abaixo:

```
1 class Pessoa:  
2     def __init__(self, nome, sexo, cpf):  
3         self.nome = nome  
4         self.sexo = sexo  
5         self.cpf = cpf  
6  
7     if __name__ == "__main__":  
8         pessoa1 = Pessoa("João", "M", "123456")  
9         print(pessoa1.nome)
```

Ao executar a linha `pessoa1 = Pessoa("João", "M", "123456")` estamos criando um objeto do tipo pessoa com nome “João”, sexo “M” e cpf “123456”.

Com isso, agora possuímos uma forma de criar diversas pessoas utilizando a mesma base, a classe Pessoa. Ao executar o código acima e imprimir o nome dessa pessoa `print(pessoa1.nome)`, teremos o seguinte retorno:

```
1 class Pessoa:  
2     def __init__(self, nome, sexo, cpf):  
3         self.nome = nome  
4         self.sexo = sexo  
5         self.cpf = cpf  
6  
7     if __name__ == "__main__":  
8         pessoa1 = Pessoa("João", "M", "123456")  
9         print(pessoa1.nome)
```

João

Declarando métodos

Como vimos anteriormente, uma classe possui **atributos** (que definem suas características) e **métodos** (que definem seus comportamentos).

Imagine que possuímos um atributo **ativo** na classe **Pessoa**. Toda pessoa criada em nosso sistema é inicializado como ativo, porém, imagine que queremos alterar o valor deste atributo e, assim, “desativar” a pessoa em nosso sistema e, além disso, exibir uma mensagem de que a pessoa foi “desativada com sucesso”.

Para isso, precisamos definir um comportamento para essa pessoa, assim, agora ela poderá ser “desativada”. Sendo assim, precisamos definir um método chamado “desativar” para criar este comportamento na classe **Pessoa**, como podemos ver abaixo:

```
class Pessoa:  
    def __init__(self, nome, sexo, cpf, ativo):  
        self.nome = nome  
        self.sexo = sexo  
        self.cpf = cpf  
        self.ativo = ativo  
  
    def desativar(self):  
        self.ativo = False  
        print("A pessoa foi desativada com sucesso")  
  
if __name__ == "__main__":  
    pessoa1 = Pessoa("João", "M", "123456", True)  
    pessoa1.desativar()
```

Para criarmos este “comportamento” na classe **Pessoa**, utilizamos a palavra reservada **def**, que indica que estamos criando um método da classe, além do nome do método e seus atributos, caso possuam.

Depois disso, é só definir o comportamento que este método irá reali-

zar. Neste caso, o método vai alterar o valor do atributo “ativo” para “False” e imprimir a mensagem “A pessoa foi desativada com sucesso”, como podemos ver abaixo:

```
1 - class Pessoa:  
2 -     def __init__(self, nome, sexo, cpf, ativo):  
3 -         self.nome = nome  
4 -         self.sexo = sexo  
5 -         self.cpf = cpf  
6 -         self.ativo = ativo  
7 -  
8 -     def desativar(self):  
9 -         self.ativo = False  
10 -        print("A pessoa foi desativada com sucesso")  
11 -  
12 - if __name__ == "__main__":  
13 -     pessoal = Pessoa("João", "M", "123456", True)  
14 -     pessoal.desativar()
```

A pessoa foi desativada com sucesso
>

Declarando propriedades

Aparentemente o código acima funciona normalmente. Porém, temos um pequeno problema com o atributo “ativo”: ele é acessível para todo mundo. Ou seja, mesmo possuindo o método “desativar”, é possível alterar o valor do atributo “ativo” sem qualquer problema:

```
1 - class Pessoa:  
2 -     def __init__(self, nome, sexo, cpf, ativo):  
3 -         self.nome = nome  
4 -         self.sexo = sexo  
5 -         self.cpf = cpf  
6 -         self.ativo = ativo  
7 -  
8 -     def desativar(self):  
9 -         self.ativo = False  
10 -        print("A pessoa foi desativada com sucesso")  
11 -  
12 - if __name__ == "__main__":  
13 -     pessoal = Pessoa("João", "M", "123456", True)  
14 -     pessoal.desativar()  
15 -     pessoal.ativo = True  
16 -     print(pessoal.ativo)
```

A pessoa foi desativada com sucesso
True
>

Este comportamento do nosso programa dá brechas para que um usuário possa ser ativado ou desativado sem passar pelo método

responsável por isso. Sendo assim, para corrigir este problema, devemos recorrer a um pilar importantíssimo da Orientação à Objetos: o **encapsulamento**. Basicamente, o encapsulamento visa definir o que pode ou não ser acessado de forma externa da classe.

Existem três tipos de atributos de visibilidade nas linguagens orientadas a objetos, que são:

Public: Atributos e métodos definidos como públicos poderão ser invocados, acessados e modificados através de qualquer lugar do projeto;

Private: Atributos e métodos definidos como privados só poderão ser invocados, acessados e modificados somente por seu próprio objeto.

Protected: Atributos e métodos definidos como protegidos só poderão ser invocados, acessados e modificados por classes que herdam de outras classes através do conceito de Herança. Sendo assim, apenas classes “filhas” poderão acessar métodos e atributos protegidos.

No Python, diferente das linguagens completamente voltadas ao paradigma da orientação à objetos (Java, C#, etc.), estes atributos existem, mas não da forma “convencional”.

Para definir um atributo público, não há necessidade de realizar nenhuma alteração, por padrão, todos os atributos e métodos criados no

Python são definidos com este nível de visibilidade. Já se precisarmos definir um atributo como privado, adicionamos dois underlines (_) antes do nome do atributo ou método:

```
1 class Pessoa:
2     def __init__(self, nome, sexo, cpf, ativo):
3         self.nome = nome
4         self.sexo = sexo
5         self.cpf = cpf
6         self.__ativo = ativo
7
8     def desativar(self):
9         self.__ativo = False
10        print("A pessoa foi desativada com sucesso")
11
12 if __name__ == "__main__":
13     pessoa1 = Pessoa("João", "M", "123456", True)
14     pessoa1.desativar()
15     pessoa1.ativo = True
16     print(pessoa1.ativo)
```

Porém, isso é apenas uma “convenção” do Python, ou seja, mesmo definindo o atributo com visibilidade privada (utilizando dois underlines antes de seu nome), ele poderá ser acessado de fora da classe:



```
1 class Pessoa:
2     def __init__(self, nome, sexo, cpf, ativo):
3         self.nome = nome
4         self.sexo = sexo
5         self.cpf = cpf
6         self.__ativo = ativo
7
8     def desativar(self):
9         self.__ativo = False
10        print("A pessoa foi desativada com sucesso")
11
12 if __name__ == "__main__":
13     pessoal = Pessoa("João", "M", "123456", True)
14     pessoal.desativar()
15     pessoal.ativo = True
16     print(pessoal.ativo)
```

A pessoa foi desativada com sucesso
True
>

Isso ocorre porque estamos falando de “convenções”, ou seja, padrões que devem ser seguidos por desenvolvedores Python.

Porém, caso precisemos acessar os atributos privados de uma classe, o Python oferece um mecanismo para construção de propriedades em uma classe e, dessa forma, melhorar a forma de encapsulamento dos atributos presentes. É comum que, quando queremos obter ou alterar os valores de um atributo, criamos métodos **getters** e **setters** para este atributo:

```
class Pessoa:  
    def __init__(self, nome, sexo, cpf, ativo):  
        self.__nome = nome  
        self.__sexo = sexo  
        self.__cpf = cpf  
        self.__ativo = ativo  
  
    def desativar(self):  
        self.__ativo = False  
        print("A pessoa foi desativada com sucesso")  
  
    def get_nome(self):  
        return self.__nome  
  
    def set_nome(self, nome):  
        self.__nome = nome  
  
if __name__ == "__main__":  
    pessoa1 = Pessoa("João", "M", "123456", True)  
    pessoa1.desativar()  
    pessoa1.ativo = True  
    print(pessoa1.ativo)  
  
    # Utilizando getters e setters  
    pessoa1.set_nome("José")  
    print(pessoa1.get_nome())
```

Porém, ao tentar acessar o valor do atributo **nome** presente na classe, fica evidente que estamos obtendo esse dado através de um método. Pensando nisso, o time de desenvolvimento criou as **Properties** para prover um meio mais “elegante” para obter e enviar novos dados aos atributos de uma classe:

```

class Pessoa:
    def __init__(self, nome, sexo, cpf, ativo):
        self.__nome = nome
        self.__sexo = sexo
        self.__cpf = cpf
        self.__ativo = ativo

    def desativar(self):
        self.__ativo = False
        print("A pessoa foi desativada com sucesso")

        def get_nome(self):
            return self.__nome

    def set_nome(self, nome):
        self.__nome = nome

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, nome):
        self.__nome = nome

if __name__ == "__main__":
    pessoa1 = Pessoa("João", "M", "123456", True)
    pessoa1.desativar()
    pessoa1.ativo = True
    print(pessoa1.ativo)

    # Utilizando geters e setters
    pessoa1.set_nome("José")
    print(pessoa1.get_nome())

    # Utilizando properties
    pessoa1.nome = "José"
    print(pessoa1.nome)

```

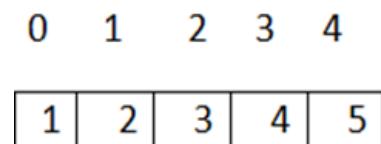
Com isso, podemos ver o quanto mais “limpo” é o uso das properties para acessar ou alterar o valor de uma propriedade privada das classes no Python.

Principais Estruturas de Dados no Python

Listas

Uma lista é a estrutura de dados mais básica do Python e armazena os dados em sequência, onde cada elemento possui sua posição na lista, denominada de **índice**. O primeiro elemento é sempre o índice zero e a cada elemento inserido na lista esse valor é incrementado.

No Python, uma lista pode armazenar qualquer tipo de dado primitivo (string, inteiro, float, etc). Na imagem abaixo podemos ver como uma lista se comporta:



Declarando Listas

Para a criação de uma lista no Python, a sintaxe é a seguinte:

```
nome_da_lista = [] # Criação de uma lista vazia
nome_da_lista = [1, 2, 3] # Criação de uma lista de inteiros
nome_da_lista = [1, "Olá, mundo!", 1.1] # Criação de uma lista com vários tipos diferentes
```

Podemos também criar listas dentro de outras listas. Essas são chamadas de **nested lists** e sua sintaxe é a seguinte:

```
nome_da_lista = ["Olá, mundo", [1, 2, 3], ["outra_lista"]]
```

Tuplas

Uma tupla é uma estrutura bastante similar a uma lista, com apenas uma diferença: os elementos inseridos em uma tupla não podem ser alterados, diferente de uma lista onde podem ser alterados livremente.

Sendo assim, em um primeiro momento, podemos pensar em uma tupla como uma lista que não pode ser alterada, mas não é bem assim... É certo que as tuplas possuem diversas características das listas, porém os usos são distintos. As listas são destinadas a serem sequências homogêneas, enquanto que as Tuplas são estruturas de dados heterogêneas.

Sendo assim, apesar de bastante similares, a tupla é utilizada para armazenar dados de tipos diferentes, enquanto que a lista é para dados do mesmo tipo.

Tuplas x Listas

As tuplas possuem algumas vantagens com relação às listas, que são:

Como as tuplas são imutáveis, a iteração sobre elas é mais rápida e, consequentemente, possuem um ganho de desempenho com relação às listas;

Tuplas podem ser utilizadas como chave para um dicionário, já que seus elementos são imutáveis. Já com a lista, isso não é possível;

Se for necessário armazenar dados que não serão alterados, utilize uma tupla. Isso garantirá que esses sejam protegidos de alterações posteriores.

Declarando Tuplas

A sintaxe para criação de uma tupla, assim como uma lista, também é bem simples. Ao invés de se utilizar colchetes (listas), são utilizados parênteses, como podemos ver abaixo:

```
nome_da_tupla = (1, 2, 3) #tupla de inteiros
nome_da_tupla = (1, "olá", 1.5) #tupla heterogênea
```

Sets

No Python, os sets são uma coleção de itens desordenada, parcialmente imutável e que não podem conter elementos duplicados. Por ser parcialmente imutável, os sets possuem permissão de adição e remoção de elementos.

Além disso, os sets são utilizados, normalmente, com operações matemáticas de união, interseção e diferença simétrica, conforme veremos nos próximos tópicos.

Declarando Sets

Para a criação de um set no Python há duas formas: a primeira é bem similar às listas e tuplas, porém utilizando chaves {} para determinar os elementos do set:

```
nome_do_set = {1, 2, 3, 4}
```

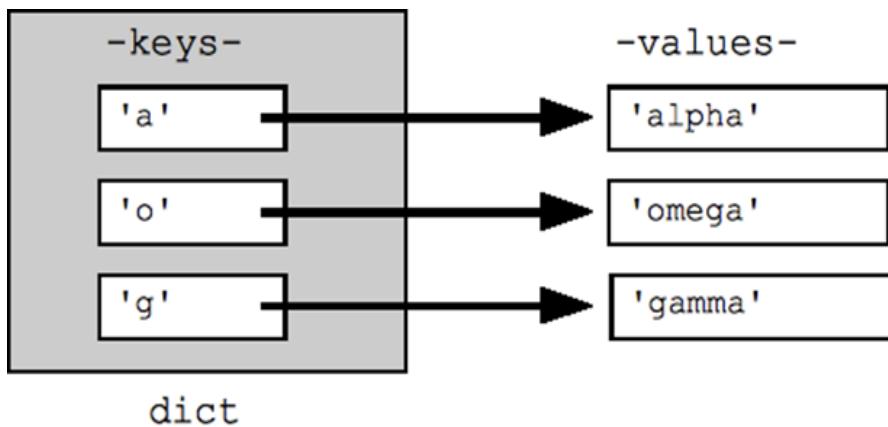
A segunda é utilizando o método set presente no Python:

```
nome_do_set = set([1, 2, 3, 4])
```

Dicionários

No Python, os dicionários são coleções de itens desordenados com uma diferença bem grande quando comparados às outras coleções (lists, sets, tuples, etc): um elemento dentro de um dicionário possui uma chave atrelada a ele, uma espécie de identificador. Sendo assim, é muito utilizado quando queremos armazenar dados de forma organizada e que possuem identificação única (como acontece em bancos de dados).

Um dicionário, portanto, pode ser visto como a figura abaixo:



Onde cada valor é “atrelado” à uma chave, seu identificador. Vale lembrar que, por necessitar que cada valor possua uma chave relacionada a ele, as chaves dos dicionários devem ser únicas para que possam ser acessadas também através do seu índice.

As chaves também não são armazenadas em qualquer ordem, elas apenas são associadas aos valores que pertencem.

Declarando Dicionários

Existem duas formas de se criar um dicionário utilizando o Python. A primeira delas é utilizando chaves ({}) e separando os elementos das chaves com dois pontos (:) e os outros elementos por vírgula (,):

```
nome_do_dicionario = {1: 'João', 2: 'José'}
nome_do_dicionario = {'nome': 'João', 'sobrenome': 'Silva'}
```

A segunda forma é utilizando o método `dict()` com o dicionário sendo passado como parâmetro:

```
nome_do_dicionario = dict({1: 'João', 2: 'José'})  
nome_do_dicionario = dict({'nome': 'João', 'sobrenome': 'Silva'})
```

Como instalar um pacote com PIP e utilizá-lo em seu projeto

O PIP é uma ferramenta para gerenciamento de pacotes de software escrito em Python. Veremos mais a fundo sobre ele mais a frente.

Afinal, o que é um Gerenciador de Pacotes?

Com o objetivo de gerenciar bibliotecas externas em projetos, um gerenciador de pacotes, de forma resumida, nada mais é que um facilitador para instalação, remoção e atualização de pacotes externos em projetos.

Desta forma, um pacote contém todos os arquivos necessários para um módulo, e os módulos, por sua vez, são bibliotecas de código Python que você pode incluir em seu projeto.

Onde encontrar os pacotes?

No site do PIP é possível encontrar todos os pacotes disponíveis para sua instalação. Nele basta pesquisar pelo pacote desejado. No exemplo abaixo, estaremos buscando o pacote “mysqlclient”, responsável pela conexão de banco de dados MySQL em scripts Python.



A busca retornará todos os pacotes que possuem alguma relação com o pacote buscado.

Filter by [classifier](#)

80 projects for "mysqlclient"

Order by [Relevance](#)

Project	Version	Published
mysqlclient 2.0.3	Python interface to MySQL	Jan 1, 2021
mysqlclient-deps 0.0.2		Jun 2, 2017
qbirthday 0.7.0b2	QBirthday birthday reminder	Jan 11, 2018
multy 1.0.1	Easily implement bulk insert and ON DUPLICATE KEY UPDATE statements with MySQL and MariaDB	Dec 2, 2019
dbConnect 2.1	Database for Humans	Jun 14, 2018

Ao selecionar o pacote desejado, será exibido uma página com toda a descrição do pacote (formas de instalação, documentação, versões, etc).

mysqlclient 2.0.3

[pip install mysqlclient](#)

Latest version

Released: Jan 1, 2021

Python interface to MySQL

Navigation

- [Project description](#) (selected)
- [Release history](#)
- [Download files](#)

Project description

mysqlclient

[Build unknown](#)

This is a fork of [MySQLdb](#).

This project adds Python 3 support and bug fixes. I hope this fork is merged back to MySQLdb like distribute was merged back to setuptools.

Support

Do Not use Github Issue Tracker to ask help. OSS Maintainer is not free tech support

When your question looks relating to Python rather than MySQL:

- Python mailing list [python-list](#)
- Slack [pythondev.slack.com](#)

Or when you have question about MySQL:

Como definir os pacotes no projeto e instalar

O processo de instalação de um pacote com o PIP em um projeto Python é bem simples. Basta utilizar o comando **pip install** seguido do nome do pacote que o próprio gerenciador se encarregará de baixá-lo e realizar sua instalação.

Imagine que precisamos consumir um serviço REST com o Python, para isso existe uma ótima biblioteca chamada **requests**. Para realizar sua instalação, basta executar o seguinte comando no terminal: **pip install requests**. Agora, com o **requests** instalado, basta importá-lo e utilizá-lo, como podemos ver no exemplo abaixo:

```
import requests
import json

def buscar_dados():
    request = requests.get("http://localhost:3002/api/todo")
    todos = json.loads(request.content)
    print(todos)

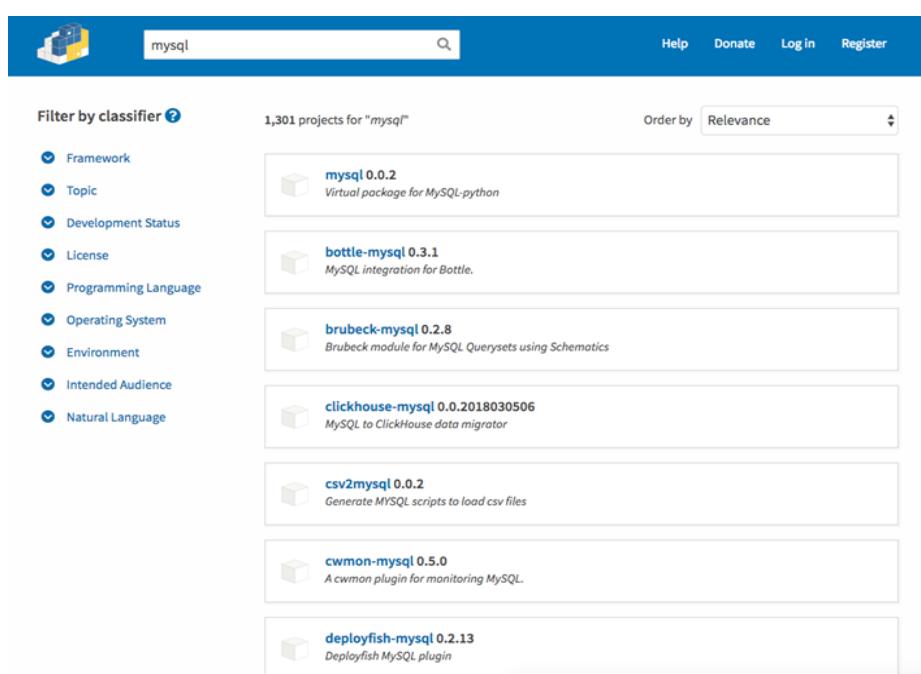
if __name__ == '__main__':
    buscar_dados()
```

Gerenciando pacotes em projetos Python com o PIP

Vimos que é comum que no desenvolvimento de projetos Python, precisemos instalar diversas bibliotecas para diferentes necessidades, como a comunicação com algum banco de dados ou até a utilização de testes unitários. Porém, não é viável que a instalação dessas bibliotecas seja feita de forma manual, já que o processo de cada uma delas podem ser, no mínimo, complicadas. Para isso, vamos nos aprofundar no PIP.

O que é o PIP?

O PIP é um gerenciador de pacotes para projetos Python. É com ele que instalamos, removemos e atualizamos pacotes em nossos projetos. É similar aos conhecidos npm e composer (php), por exemplo. O PIP possui uma página onde nós conseguimos buscar os pacotes disponíveis para a utilização. Nela podemos pesquisar por um pacote específico ou até uma palavra chave:

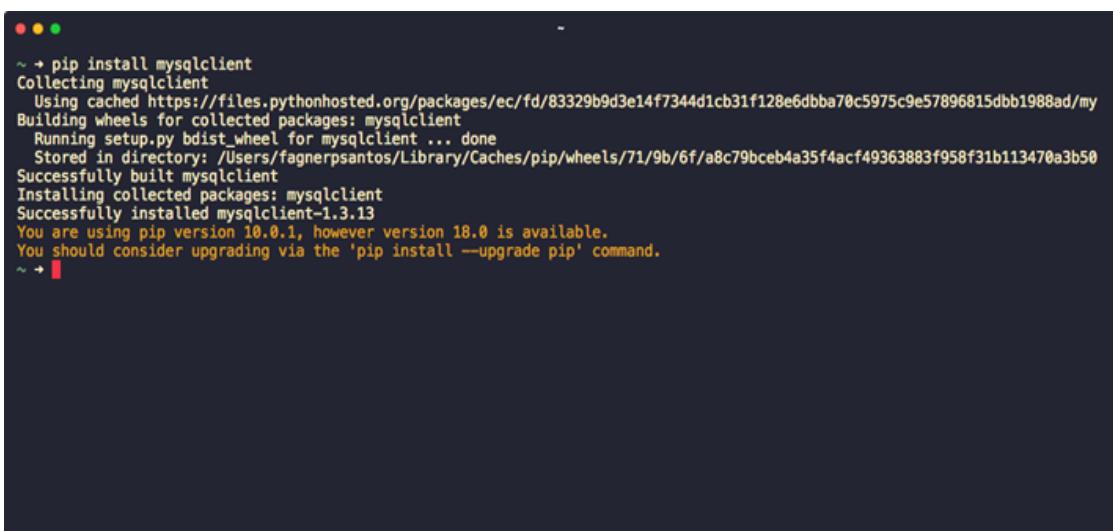


The screenshot shows the Python Package Index (PyPI) search results for the query "mysql". At the top, there's a navigation bar with a logo, a search input field containing "mysql", and links for Help, Donate, Log in, and Register. Below the search bar, a filter section titled "Filter by classifier" is visible, listing categories like Framework, Topic, Development Status, License, Programming Language, Operating System, Environment, Intended Audience, and Natural Language, each with a checked checkbox. To the right of the filter, it says "1,301 projects for 'mysql'" and "Order by Relevance". The main content area displays a list of packages, each with a thumbnail icon, name, version, and a brief description. The listed packages are: mysql 0.0.2 (Virtual package for MySQL-python), bottle-mysql 0.3.1 (MySQL integration for Bottle.), brubeck-mysql 0.2.8 (Brubeck module for MySQL Querysets using Schematics), clickhouse-mysql 0.0.2018030506 (MySQL to ClickHouse data migrator), csv2mysql 0.0.2 (Generate MySQL scripts to load csv files), cmon-mysql 0.5.0 (A cmon plugin for monitoring MySQL.), and deployfish-mysql 0.2.13 (Deployfish MySQL plugin).

Gerenciando pacotes

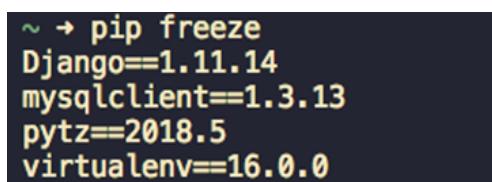
Após instalar o PIP em nosso SO, podemos utilizá-lo para diferentes tarefas, como instalar, remover, listar e atualizar pacotes. Veremos agora como realizar cada uma dessas tarefas.

Para a instalação de novos pacotes utilizando o PIP, temos o seguinte comando: **pip install nome_do_pacote**. Este comando irá baixar o pacote desejado e instalar em nosso SO, como podemos ver na figura abaixo. O nome do pacote pode ser encontrado na página oficial do PIP.



```
~ → pip install mysqlclient
Collecting mysqlclient
  Using cached https://files.pythonhosted.org/packages/ec/fd/83329b9d3e14f7344d1cb31f128e6dbba70c5975c9e57896815dbb1988ad/mysqlclient-1.3.13-py2.py3-none-any.whl
Building wheels for collected packages: mysqlclient
  Running setup.py bdist_wheel for mysqlclient ... done
  Stored in directory: /Users/fagnergsantos/Library/Caches/pip/wheels/71/9b/6f/a8c79bceb4a35f4acf49363883f958f31b113470a3b50
Successfully built mysqlclient
Installing collected packages: mysqlclient
Successfully installed mysqlclient-1.3.13
You are using pip version 10.0.1, however version 18.0 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
~ →
```

Para a listagem dos pacotes instalados, utilizamos o comando **pip freeze**:



```
~ → pip freeze
Django==1.11.14
mysqlclient==1.3.13
pytz==2018.5
virtualenv==16.0.0
```

Para a atualização dos pacotes que estão instalados, utilizamos o comando **pip install --upgrade nome-do-pacote**:

```
~ → pip install --upgrade django
Collecting django
  Downloading https://files.pythonhosted.org/packages/f8/1c/31112c778b7a56ce18e3fff5e8915719fbe1cd3476c1eef557ddacfac8b/django-1.11.15-py2.py3-none-any.whl (6.9MB)
    100% |██████████| 7.0MB 1.9MB/s
Requirement not upgraded as not directly required: pytz in /usr/local/lib/python2.7/site-packages (from django) (2018.5)
Installing collected packages: django
  Found existing installation: Django 1.11.14
    Uninstalling Django-1.11.14:
      Successfully uninstalled Django-1.11.14
Successfully installed django-1.11.15
```

Para a remoção dos pacotes que estão instalados, utilizamos o comando **pip uninstall nome-do-pacote**:

```
~ → pip uninstall mysqlclient
Uninstalling mysqlclient-1.3.13:
  Would remove:
    /usr/local/lib/python2.7/site-packages/MySQLdb/*
    /usr/local/lib/python2.7/site-packages/_mysql.so
    /usr/local/lib/python2.7/site-packages/_mysql_exceptions.py
    /usr/local/lib/python2.7/site-packages/mysqlclient-1.3.13.dist-info/*
Proceed (y/n)? y
  Successfully uninstalled mysqlclient-1.3.13
```

Criando ambientes virtuais para projetos Python com o Virtualenv

Quando estamos desenvolvendo diversos projetos em Python, é comum utilizarmos diferentes versões de uma mesma biblioteca entre estes projetos. Por exemplo, imagine que estamos desenvolvendo o projeto x com a biblioteca mysqlclient em sua versão 1.0 e o projeto y com a mesma biblioteca, porém na versão 2.0. Como faríamos para gerenciar estas dependências e o sistema operacional saber qual versão correta executar quando estivermos em cada projeto?

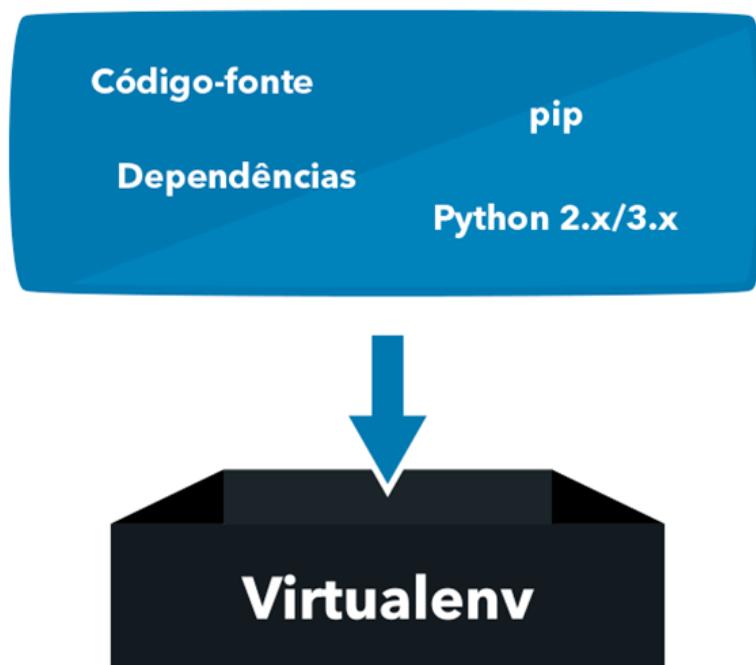
Esta é uma tarefa um tanto quanto complexa para o SO gerenciar, podendo acarretar em conflitos entre as versões e causar muita dor de cabeça. Sendo assim, o mais comum é utilizarmos diferentes ambientes virtuais, chamados de **virtualenvs**, um para cada projeto.

O que é uma virtualenv?

Como dito acima, um problema muito comum é quando precisamos utilizar diversas versões de uma mesma biblioteca em diferentes projetos Python. Isso pode acarretar em conflitos entre as versões e muita dor de cabeça para o desenvolvedor. Para resolver este problema, o mais correto é a criação de um ambiente virtual para cada projeto.

Basicamente, um ambiente virtual empacota todas as dependências que um projeto precisa e armazena em um diretório, fazendo com que

nenhum pacote seja instalado diretamente no sistema operacional. Sendo assim, cada projeto pode possuir seu próprio ambiente e, consequentemente, suas bibliotecas em versões específicas.



Como funciona uma virtualenv?

O funcionamento de uma virtualenv é bem simples. Basicamente, uma cópia dos diretórios necessários para que um programa Python seja executado é criada, incluindo:

- PIP (Gerenciador de pacotes);
- A versão do Python utilizada (2.x ou 3.x);
- Dependências instaladas com o pip (armazenadas no diretório site-packages);
- Seu código fonte;

Bibliotecas comuns do Python.

Isso faz com que as dependências sejam sempre instaladas em uma virtualenv específica, não mais no sistema operacional. A partir daí, cada projeto executa seu código-fonte utilizando sua virtualenv própria.

Instalando a virtualenv

A instalação de uma virtualenv é feita utilizando o pip, gerenciador de pacotes do Python. Após instalar o pip, utilizamos o comando abaixo para instalar o pacote virtualenv em nosso computador:

```
pip install virtualenv
```

Feito isso, o pacote estará instalado e pronto para ser utilizado. Agora já podemos criar e gerenciar nossos ambientes virtuais.

Criando uma nova virtualenv

O processo de criação de uma virtualenv é bastante simples e pode ser feito utilizando um único comando, como podemos ver abaixo:

```
virtualenv nome_da_virtualenv
```

O mais comum é criar a virtualenv na raiz do projeto que ela irá pertencer. Isso permite uma organização maior das virtualenvs que

possuímos em nosso computador:

```
→ projeto_python virtualenv venv
Using base prefix '/usr/local/Cellar/python/3.7.0/Frameworks/Python.framework/Versions/3.7'
New python executable in /Users/fagnerpsantos/Desktop/projeto_python/venv/bin/python3.7
Also creating executable in /Users/fagnerpsantos/Desktop/projeto_python/venv/bin/python
Installing setuptools, pip, wheel...
done.
→ projeto_python ls
venv
```

Com isso, criamos a virtualenv do projeto chamada “venv”. É ela quem vai comportar todos os pacotes necessários para a execução do projeto.

Ativando uma virtualenv

Após criar uma virtualenv, precisamos ativá-la para que possamos instalar os pacotes necessários do projeto. Para isso, utilizamos o seguinte comando:

source nome_da_virtualenv/bin/activate (Linux ou macOS)
nome_da_virtualenv/Scripts/Activate (Windows)

O comando acima irá ativar a virtualenv e teremos um retorno similar ao ilustrado abaixo:

```
→ projeto_python source venv/bin/activate
(venv) → projeto_python
```

Note que o nome da virtualenv, agora, é exibido antes do caminho do diretório em que estamos. Isso indica que a virtualenv foi ativada com sucesso.

Desativando uma virtualenv

Para desativar uma virtualenv utilizamos o comando deactivate, como podemos ver abaixo:

```
(venv) ➔ projeto_python deactivate  
➔ projeto_python
```

Instalando pacotes

Com a virtualenv ativada, podemos instalar os pacotes necessários do projeto utilizando o próprio PIP:

```
(venv) ➔ projeto_python pip install django  
Collecting django  
  Downloading https://files.pythonhosted.org/packages/d1/e5/2676be45ea49cf09a663f289376b3888acc57ff06c953297bfdee1fb08/Django-2.1.3-py3-none-any.whl (7.3MB)  
    100% |██████████| 7.3MB 1.9MB/s  
Collecting pytz (from django)  
  Downloading https://files.pythonhosted.org/packages/f8/0e/2365ddc010afb3d79147f1dd544e5ee24bf4ece58ab99b16fbb465ce6dc0/pytz-2018.7-py2.py3-none-any.whl (506kB)  
    100% |██████████| 512kB 9.3MB/s  
Installing collected packages: pytz, django  
Successfully installed django-2.1.3 pytz-2018.7
```

Com isso, instalamos o pacote **Django**, em sua versão mais atual, na virtualenv do projeto “projeto_python”. Agora, se precisarmos instalar uma outra versão do Django em outro projeto, bastaria criar uma nova virtualenv e realizar o mesmo processo:

```
➔ projeto_python_2 source venv/bin/activate  
(venv) ➔ projeto_python_2 pip install django==1.11  
Collecting django==1.11  
  Downloading https://files.pythonhosted.org/packages/47/a6/078ebcbd49b19e22fd560a2348fc5cec9e5dcfe3c4fad8e64c9865135bb/Django-1.11-py2.py3-none-any.whl (6.9MB)  
    100% |██████████| 6.9MB 2.2MB/s  
Collecting pytz (from django==1.11)  
  Using cached https://files.pythonhosted.org/packages/f8/0e/2365ddc010afb3d79147f1dd544e5ee24bf4ece58ab99b16fbb465ce6dc0/pytz-2018.7-py2.py3-none-any.whl  
Installing collected packages: pytz, django  
Successfully installed django-1.11 pytz-2018.7
```

Com isso, teremos diferentes versões do Django instaladas no mesmo sistema operacional, porém isoladas em cada ambiente. Sendo assim, o projeto “projeto_python” utiliza a versão mais recente (2.1.3) e o “projeto_python_2” utiliza a versão 2.11, tudo de forma isolada entre os projetos:

```
(venv) ➔ projeto_python_2 pip freeze
Django==1.11
pytz==2018.7
```

O ponto final ou apenas o começo?

Neste ebook compilamos os principais conceitos que todos que querem começar no Python devem conhecer, desde conceitos mais iniciais, como criar seu primeiro exemplo e até mesmo como instalar dependências, gerenciar pacotes com o PIP e como utilizar a Virtualenv.

Mas vale lembrar que a partir de cada conceito você pode se aprofundar muito mais para iniciar no mundo do Python da melhor maneira.

Temos diversos materiais em nosso blog, como artigos mais teóricos, práticos, tutoriais, dicas de carreira, aprendizado, estudos... muita coisa!

É totalmente gratuito caso você tenha interesse em continuar seus estudos de maneira mais autodidata. Mas, se você gostaria de ter um guia especializado, alguém que irá explicar todos os próximos passos para você prosseguir nesta carreira, temos algo muito legal pra você: a nossa mentoria personalizada.

Com nossa mentoria, além de todos os cursos, formações e suporte que você já encontra em nossa plataforma, você ainda terá um apoio que te ajudará a alcançar seus objetivos profissionais. Vamos explorar o mundo do Python juntos.

Então, vamos nessa?

Conheça a TreinaWeb

Com mais de 15 anos de história, a TreinaWeb é uma plataforma de ensino online que oferece cursos voltados para a tecnologia.

Além da diversidade de cursos e formações, você ainda conta com um direcionamento e suporte com os professores, para te ajudar a guiar sua carreira e também para sanar suas dúvidas técnicas.

Aprenda tudo o que você precisa para entrar ou se desenvolver na área que mais tem demanda por novos profissionais. Vá do zero ao profissional nas linguagens, frameworks, ferramentas e metodologias mais utilizadas no mercado atual.

Não importa seu nível de conhecimento. Vá do zero ao mercado de trabalho com a gente!

ACESSE O NOSSO SITE

