



devaria

Programação – Python

Parte 4

Parte 1



Resumo



Design Patterns

Encapsulamento

```
class ContaBancaria:
    def __init__(self, tipo, numero, agencia, saldo):
        self.tipo = tipo
        self.numero = numero
        self.agencia = agencia
        self.__saldo = saldo

    def exibirDadosConta(self):
        print("Conta, tipo: {self.tipo},
        agencia: {self.agencia} e numero: {self.numero}")

    def exibirSaldo(self):
        print("Seu saldo é: {self.__saldo}")
```



python

```
11
12
13 ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)
14
15
16 class Base(ContainerAware, metaclass=abc.ABCMeta):
17     """The base class for all controllers.
18     Attributes:
19         __action__ (string): The last action that was
20         executed.
21     """
22     def execute(self, **kwargs):
23         method = self.get_execute_method(**kwargs)
24         self.__action__ = method
25         return method(**kwargs) or {}
```

Encapsulamento

```
conta1 = ContaBancaria("Corrente", 35662, "0001", 0)
```

```
conta1.tipo # Funciona porque o atributo é publico
```

```
conta1.numero = 234123 # Funciona porque o atributo é publico
```

```
conta1.__saldo # Não funciona pegar o valor porque é privado
```

```
conta1.exibirSaldo() # Funciona porque o método é publico
```



python

Herança

```
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def Respirar(self):
        print("O animal {self.nome} está respirando")

class Mamifero(Animal):
    def __init__(self, nome):
        super().__init__(nome)

    def Mamar(self):
        print("O animal {self.nome} está se alimentando")

class Reptil(Animal):
    def __init__(self, nome):
        super().__init__(nome)

    def BotarOvo(self):
        print("O animal {self.nome} está botando ovo")
```



python

```
11
12
13 ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)
14
15
16 class Base(ContainerAware, metaclass=abc.ABCMeta):
17     """The base class for all controllers.
18     Attributes:
19         __action__ (string): The last action that was
20         executed.
21     """
22     def execute(self, **kwargs):
23         method = self.get_execute_method(**kwargs)
24         self.__action__ = method
25         return method(**kwargs) or {}
```

Polimorfismo

```
class FormaGeometrica:
    def __init__(self):
        pass

    def Desenhar(self):
        print("Você está desenhando uma forma geométrica.")
```

```
class Circulo(FormaGeometrica):
    def __init__(self):
        pass

    def Desenhar(self):
        print("Você está desenhando um Circulo")
```

```
class Quadrado(FormaGeometrica):
    def __init__(self):
        pass

    def Desenhar(self):
        print("Você está desenhando um Quadrado")
```



python

Polimorfismo

```
formas = [  
    Quadrado(),  
    Circulo()  
]  
  
for forma in formas:  
    forma.Desenhar()
```



python

```
11  
12  
13 ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)  
14  
15  
16 class Base(ContainerAware, metaclass=abc.ABCMeta):  
17     """The base class for all controllers.  
18     Attributes:  
19     __action__ (string): The last action that was executed.  
20     """  
21  
22     def execute(self, **kwargs):  
23         method = self.get_execute_method(**kwargs)  
24         self.__action__ = method  
25         return method(**kwargs) or {}
```


Enums

Um tipo de enumeração (ou enum) é um tipo de valor definido por um conjunto de constantes nomeadas do tipo numérico integral subjacente. Para definir um tipo de enumeração, use a enum palavra-chave e especifique os nomes dos membros de enumeração:

```
class EstacoesDoAno(Enum):  
    VERA0 = 'Verão'  
    OUTONO = 'Outono'  
    INVERNO = 'Inverno'  
    PRIMAVERA = 'Primavera'
```

```
class CodigosDeErro(Enum):  
    Vazio = 0  
    Desconhecido = 1  
    UsuarioInvalido = 2  
    DadosInvalidos = 3
```



python

Interfaces/Implementações

Um tipo de enumeração (ou enum) é um tipo de valor definido por um conjunto de constantes nomeadas do tipo numérico integral subjacente. Para definir um tipo de enumeração, use a enum palavra-chave e especifique os nomes dos membros de enumeração:

```
import abc
```

```
class Formulario (abc.ABC):
```

```
    @abc.abstractmethod  
    def validarCampos(self):  
        pass
```

```
class Impressao (abc.ABC):
```

```
    @abc.abstractmethod  
    def imprimirComprovante(self, produtos):  
        pass
```



python

```
11  
12  
13 ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)  
14  
15  
16 class Base(ContainerAware, metaclass=abc.ABCMeta):  
17     """The base class for all controllers.  
18     Attributes:  
19     __action__ (string): The last action that was  
20     executed.  
21     """  
22     def execute(self, **kwargs):  
23         method = self.get_execute_method(**kwargs)  
24         self.__action__ = method  
25         return method(**kwargs) or {}
```

Interfaces/Implementações

classe `ContaBancariaFormulario` (`Formulario`,
`Impressao`):

```
    def validarCampos():  
        print(f"Campos validados com  
sucesso")  
  
    return True
```

```
    def imprimirComprovante(produtos):  
        print(f"Imprimindo comprovante dos  
produtos")
```



python

```
11  
12  
13 ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)  
14  
15  
16 class Base(ContainerAware, metaclass=abc.ABCMeta):  
17     """The base class for all controllers.  
18     Attributes:  
19     __action__ (string): The last action that was  
20     executed.  
21     """  
22     def execute(self, **kwargs):  
23         method = self.get_execute_method(**kwargs)  
24         self.__action__ = method  
25         return method(**kwargs) or {}
```

Design Patterns

Padrões de projeto (design patterns) são soluções típicas para problemas comuns em projetos de software. Cada padrão é como uma planta de construção que você pode customizar para resolver um problema de projeto particular em seu código.

Padrões são como um conjunto de ferramentas para soluções de problemas comuns em design de software. Eles definem uma linguagem comum, que ajuda sua equipe a se comunicar mais eficientemente.



python

Design Patterns

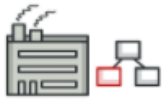

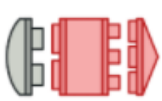




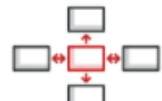

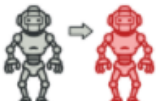

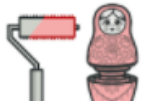
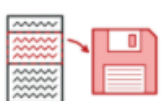

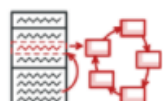
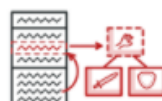






Geralmente separamos os Design Patterns em três categorias:

- Criação
- Estruturais
- Comportamentais



python

Design Patterns

| | | | | | | | |
|---|---|--|--|---|--|---|---|
|  Factory Method |  Abstract Factory |  Adapter |  Bridge |  Chain of Responsibility |  Command |  Iterator |  Mediator |
|  Builder |  Prototype |  Composite |  Decorator |  Memento |  Observer |  State |  Strategy |
|  Singleton | |  Facade |  Flyweight |  Template Method |  Visitor | | |
| |  Proxy | | | | | | |



python

```
11
12
13 ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)
14
15
16 class Base(ContainerAware, metaclass=abc.ABCMeta):
17     """The base class for all controllers.
18     Attributes:
19         __action__ (string): The last action that was
20         executed.
21     """
22     def execute(self, **kwargs):
23         method = self.get_execute_method(**kwargs)
24         self.__action__ = method
25         return method(**kwargs) or {}
```


Design Patterns

- Controller – Normalmente utilizado na arquitetura MVC, esse pattern indica que qualquer arquivo concluído com Controller tem a responsabilidade de controlar as requisições do usuário e direcionar para o resultado correto;
- Factory – Esse pattern tem a responsabilidade de auxiliar na gestão de resoluções que envolvem caminhos muito diferentes, normalmente utilizado para polimorfismo é uma classe que instancia de acordo com parâmetros uma determinada regra de negócio a ser executada, deixando o código genérico e resiliente;
- Entity / Model – Um dos patterns mais utilizados, qualquer arquivo model ou entity costuma ser um espelho da tabela no banco de dados. Assim, quando o banco for acessado, esse arquivo será o arquivo ideal para preencher com os dados proveniente do banco;



Design Patterns

- Service – Esse pattern, também muito utilizado, serve para que possamos guardar as regras de negócio complexas em um único método para que outros arquivos possam chamar esse método exclusivamente para acessar toda regra de negócio de uma funcionalidade, deixando nosso código com uma manutenção melhor;
- Repository – Assim como os models/entity são o espelho do banco de dados, os repositories são os arquivos que manipulam esses modelos e entidades a fim de gerar um melhor resultado na integração com o banco. Normalmente, são os arquivos onde as queries e demais scripts de banco estão inseridas;
- DTO – Muito utilizado no backend, dto significa data transfer object, ou seja, um objeto de transferência de dados. Não são modelos, são apenas objetos onde juntamos uma série de dados proveniente de diversas fontes a fim de facilitar o resultado para o usuário;



Design Patterns

- Util – Normalmente, usamos esse pattern para atividades diversas que chamamos de utilitários, como formatações de campos, validações de senha, cpf, entre outros. Para qualquer função/método que desempenhe um papel de utilitário para o sistema inteiro, pode ser utilizado esse pattern;
- Singleton – Gerenciar memória em linguagens com orientação a objetos é sempre um desafio, senão ficamos instanciando milhares de objetos iguais sem necessidade. Um singleton é um pattern que identifica que aquela classe é instanciada apenas uma vez no sistema deixando-o mais performático;
- Transaction – Existem os serviços que, como já falamos, são a base de regra de negócio dos nossos sistemas. Porém, às vezes, muitos serviços precisam ser chamados, e a falha de um impacta nos demais. Para este caso, utilizamos o pattern de transaction, para que tudo dentro de uma transação ou ocorra 100% ou não ocorra nada.



Design Patterns

- Site para consultar patterns:

<https://sourcemaking.com>



python



Café ou Dúvidas?

Para que a aula não fique cansativa, a cada conclusão de assunto teremos 15 minutos para que vocês possam tirar as dúvidas através dos comentários no vídeo e responderemos ao vivo.

Parte 2

- Particularidades do Python

Particularidades do Python

Até agora vimos coisas em comum em todas as linguagens para facilitar o entendimento, agora vamos apresentar o que cada linguagem tem de particular (um ou outro termo podemos até compartilhar entre uma linguagem e outra, porém não é algo que as 4 linguagens apresentadas tenham).



python

Particularidades do Python

- **Decorators** – Um decorador em Python é um objeto que estende/modifica a funcionalidade de uma função (ou método) em tempo de execução e conceitualmente está mais próximo da anotação do Java que do decorador da orientação a objetos.

Na prática, o decorador age como uma embalagem de presente, acondicionando a função sem alterar seu conteúdo (ele continua sendo um presente) mas deixando-o mais bonito.



Particularidades do Python

```
def decora_funcao(func):  
    """ Criando um decorator """  
  
    def decoracao():  
        print("Executa algo antes de executar a soma")  
        func()  
  
    return decoracao()
```

```
@decora_funcao  
def soma(n1, n2):  
    """ Esta função realiza a soma de dois números """  
  
    return n1 + n2
```



python

```
11  
12  
13 ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)  
14  
15  
16 class Base(ContainerAware, metaclass=abc.ABCMeta):  
17     """The base class for all controllers.  
18     Attributes:  
19     __action__ (string): The last action that was  
20     executed.  
21     """  
22     def execute(self, **kwargs):  
23         method = self.get_execute_method(**kwargs)  
24         self.__action__ = method  
25         return method(**kwargs) or {}
```


Particularidades do Python

- **Ambientes Virtuais** – Aplicações em Python normalmente usam pacotes e módulos que não vêm como parte da instalação padrão. Aplicações às vezes necessitam uma versão específica de uma biblioteca, porque ela requer que algum problema em particular tenha sido consertado ou foi escrita utilizando-se de uma versão obsoleta da interface da biblioteca.



Particularidades do Python

- **Docstring** – Uma docstring é uma string literal presente na primeira linha da definição de um módulo, classe ou função. O docstring de qualquer objeto pode ser acessado através de um atributo especial chamado `__doc__`.

```
def soma(n1, n2):  
    """ Esta função realiza a soma de dois números """  
  
    return n1 + n2
```

```
print(soma.__doc__) # Esta função realiza a soma de dois números
```





Café ou Dúvidas?

Para que a aula não fique cansativa, a cada conclusão de assunto teremos 15 minutos para que vocês possam tirar as dúvidas através dos comentários no vídeo e responderemos ao vivo.

Parte 3

 Frameworks

 Gestão de Dependência

Frameworks

Como já falamos, hoje em dia fazer tudo do zero é, além de demorado e custoso, uma perda de tempo.

Boa parte dos problemas básicos na programação já foram solucionados pela comunidade, e compartilhados para que realmente nosso esforço na programação fique em resolver os problemas solicitados, e não em perder tempo em configuração.

Para isto, temos as bibliotecas e frameworks que nos auxiliam a resolver estes problemas sem termos que fazer toda a solução do zero.

Vamos apresentar aqui os principais frameworks em Python.



Frameworks

django

Django é um framework para desenvolvimento rápido para web, escrito em Python, que utiliza o padrão model-template-view (MTV).

Django utiliza o princípio DRY (Don't Repeat Yourself), onde faz com que o desenvolvedor aproveite ao máximo o código já feito, evitando a repetição.

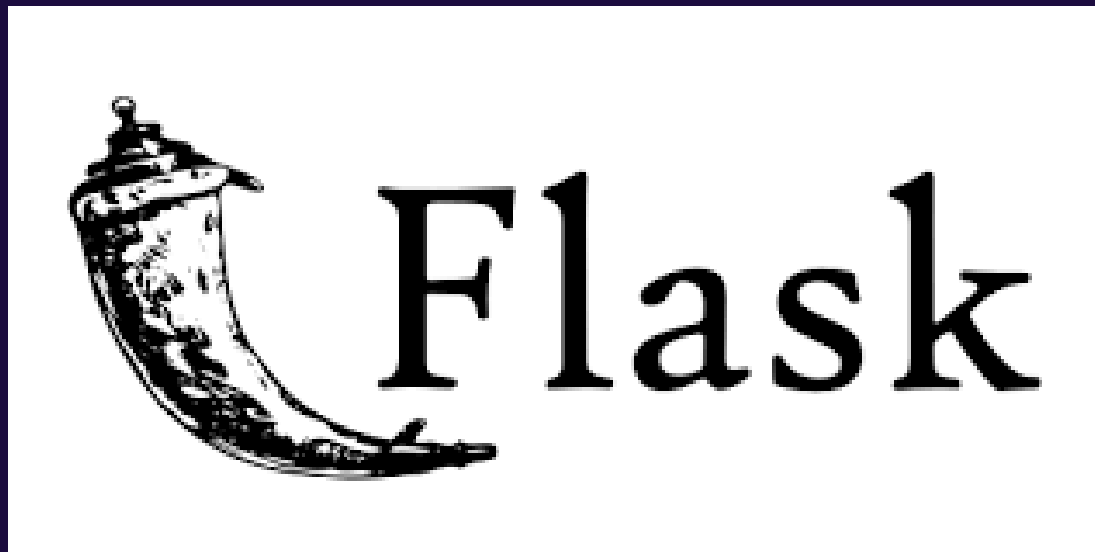
Principais Características:

- Mapeamento Objeto-Relacional (ORM)
- Interface Administrativa
- Formulários
- URLs Amigáveis
- Sistema de Templates
- Sistema de Cache
- Internacionalização



python

Frameworks



O principal objetivo do Flask é prover um modelo simples para desenvolvimento web, ao mesmo tempo que tem a flexibilidade no uso da linguagem Python.

Não possui camada de abstração de banco de dados, validação de formulários ou mesmo qualquer outro componente, no entanto bibliotecas de terceiros provêm essas funcionalidades.



Frameworks



FastAPI é um moderno e rápido (alta performance) framework web para construção de APIs com Python 3.6 ou superior, baseado nos type hints padrões do Python.

Documentação Interativa da API.

Um dos frameworks mais novos do Python para web.



Frameworks



É um micro-framework WSGI rápido, simples e leve projetado e distribuído como um módulo de arquivo único e sem dependências além da biblioteca padrão do Python. Ele pode ser executado com Python 2.5+ e 3.x.

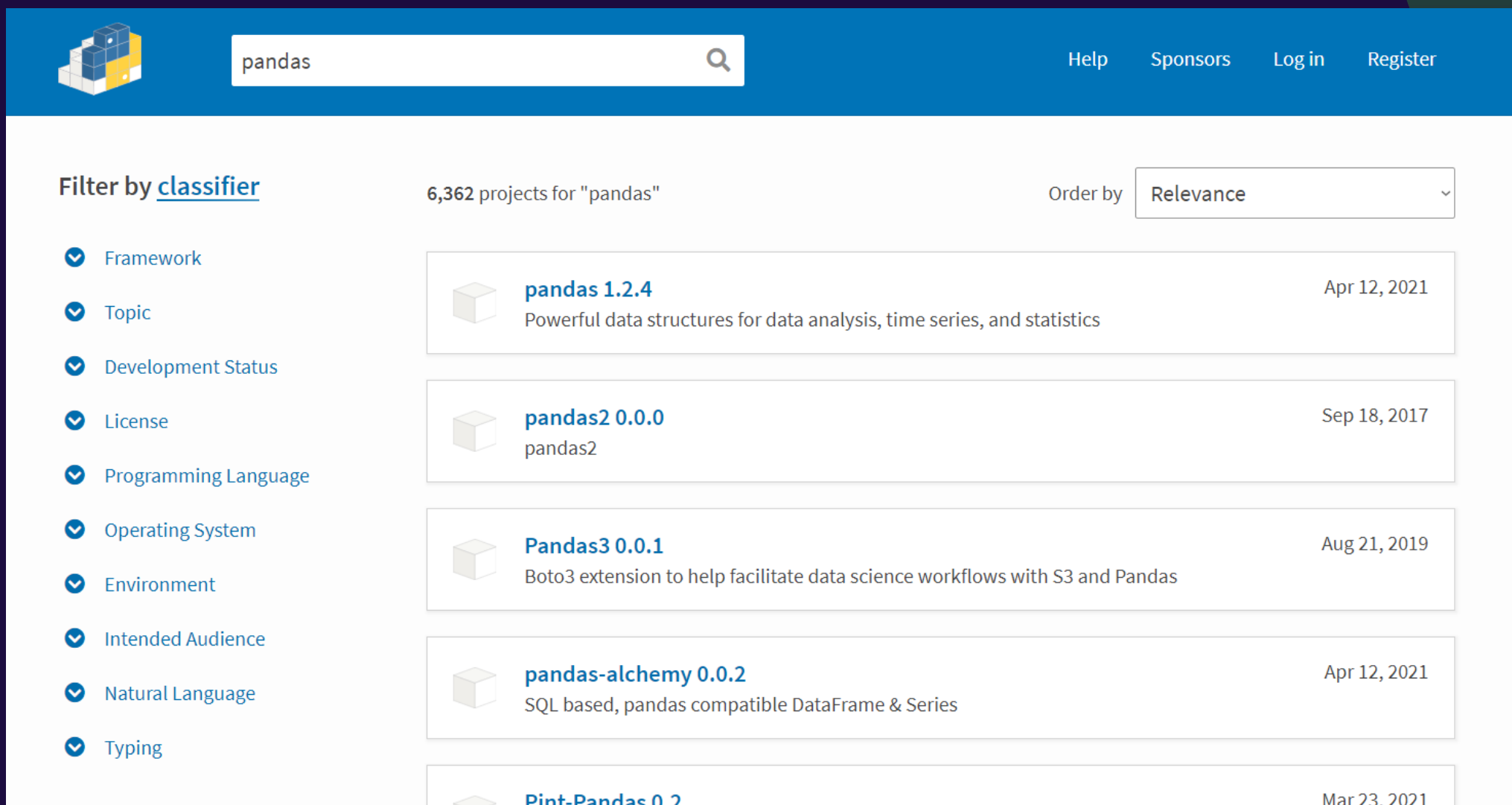
Originalmente, foi desenvolvido para a criação de APIs Web, incluindo funcionalidades prontas para roteamento para URLs limpas e dinâmicas, suporte para templates mako, jinja2 e cheetah.



Gestão de Dependências

O PIP é um gerenciador de pacotes para projetos Python. É com ele que instalamos, removemos e atualizamos pacotes em nossos projetos.

Site:
<https://pypi.org>



The screenshot shows the PyPI search results for the package 'pandas'. The search bar at the top contains the text 'pandas'. Below the search bar, there are filters on the left, a search result count of '6,362 projects for "pandas"', and a dropdown menu for 'Order by' set to 'Relevance'. The search results are listed in a table with columns for the package name, description, and release date.

| Package Name | Description | Release Date |
|-----------------------------|--|--------------|
| pandas 1.2.4 | Powerful data structures for data analysis, time series, and statistics | Apr 12, 2021 |
| pandas2 0.0.0 | pandas2 | Sep 18, 2017 |
| Pandas3 0.0.1 | Boto3 extension to help facilitate data science workflows with S3 and Pandas | Aug 21, 2019 |
| pandas-alchemy 0.0.2 | SQL based, pandas compatible DataFrame & Series | Apr 12, 2021 |
| Pint-Pandas 0.2 | | Mar 23, 2021 |



python

Gestão de Dependências

Para a instalação de novos pacotes utilizando o PIP, temos o seguinte comando:

```
pip install nome_do_pacote
```

Para a listagem dos pacotes instalado, utilizamos o comando:

```
pip freeze
```

Para exportar os pacotes instalados em um txt, utilizamos o comando:

```
pip freeze > requirements.txt
```

Para a atualização dos pacotes que estão instalados, utilizamos o comando:

```
pip install --upgrade nome_do_pacote
```

Para a remoção dos pacotes que estão instalados, utilizamos o comando:

```
pip uninstall nome_do_pacote
```



python

Gestão de Dependências

Os pacotes sempre são instalados no ambiente virtual ativo do projeto.

As dependências instaladas são salvas no arquivo requirements.txt



python

Hora da prática

Agora que já entendemos como funcionam frameworks e as particularidades do Python, vamos exercitar algumas coisas:

- Criar uma api;
- Criar uma classes e comportamentos;
- Adicionar dependências;
- Retornar para o usuário o resultado no endpoint solicitado.



python

Não se esqueça de commitar
Os programas desenvolvidos no seu Git =D





Obrigado pela participação

Esperamos que você tenha gostado, fique à vontade para nos enviar seus feedbacks sobre esta aula.
Esperamos você na próxima live, terça-feira 17/05 as 19:30. Não se esqueça de consultar o calendário e anote na sua agenda.

Se inscreva no canal e siga-nos nas rede sociais:



www.youtube.com/c/Devaria



[@devaria_oficial](https://www.instagram.com/devaria_oficial)

[@rafamazucato](https://www.instagram.com/rafamazucato)

[@castellodaniel](https://www.instagram.com/castellodaniel)

[@oliveirandouglas](https://www.instagram.com/oliveirandouglas)