

Herança e Classe abstrata

[2]

Orientação a Objetos II

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Prof. Felipe Scheidt – IFPR – Campus Foz do Iguaçu

2025

Última aula

- Sintaxe do Python
- Declaração de variáveis e inicialização
- Funções nativas:
 - len, sorted, range, print, input, enumerate
- Declarar função (**def**)
- Declarar uma Classe (**class**)

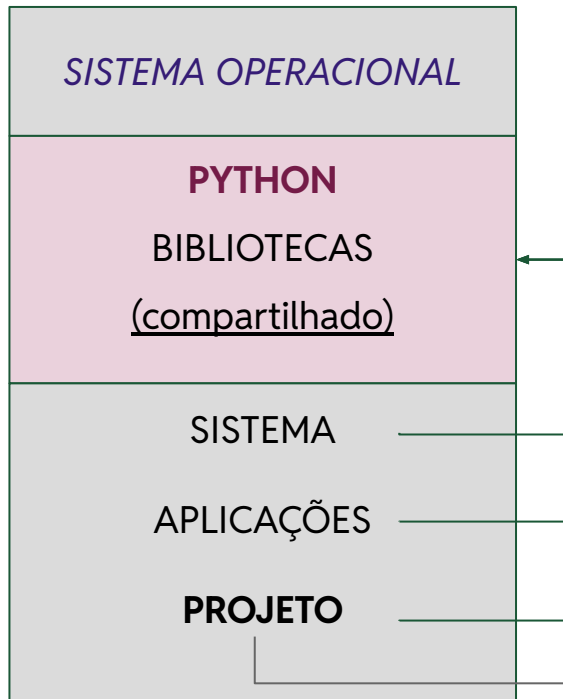
Ambiente virtual

Para desenvolver em python é fundamental "isolar" o ambiente de desenvolvimento do projeto.

- O ambiente virtual é definido *por* projeto.
- Cada projeto possui a sua própria versão do python e bibliotecas específicas.
- Evitar conflito com o python instalado no sistema operacional.

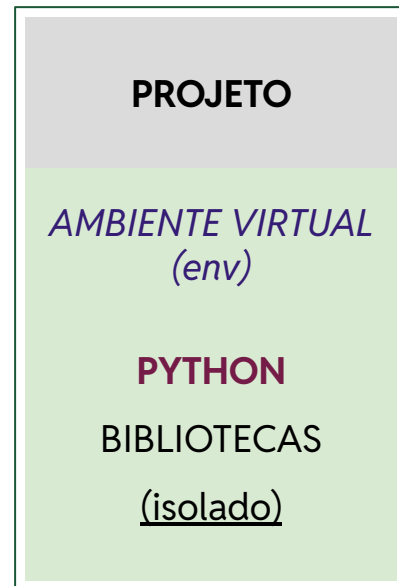
Ambiente virtual

Compartilhado



```
$ which python3  
/usr/bin/python
```

✓ Ambiente isolado



Configuração do ambiente virtual

Abrir o terminal e criar a pasta do projeto para armazenar os arquivos e o **env**

```
$ mkdir python-projects/oo2
$ cd python-projects/oo2
$ pwd
/home/ifpr/python-projects/oo2
```

```
$ python3 -m venv env
$ source env/bin/activate
(env) $
```

- 1 Cria o ambiente virtual nomeado "env"
- 2 Ativa o ambiente virtual

```
(env) $ pip install rich
(env) $ code .
```

- 3 pip - instala a biblioteca "rich" no env

Estrutura do projeto

```
.
├── aulas
├── env
├── funcoes.py
├── hello.py
├── listas.py
├── pessoa.py
└── README.md
```

```
$ which python3
/home/ifpr/python-projects/oo2/bin/python
```

```
.
├── aulas
│   └── 01
├── env
│   ├── bin
│   ├── include
│   ├── lib
│   ├── lib64 -> lib
│   └── pyenv.cfg
├── funcoes.py
├── hello.py
├── listas.py
├── pessoa.py
└── README.md
```

Classe concreta

Pessoa
+ nome: string + renda: float + email: string
+ exibe_dados()

Construtor `__init__`

O método construtor permite **inicializar** os atributos das instâncias. No Python, o construtor também é usado para **declarar** os atributos da classe. Além disso, o construtor permite definir:

- Atributos **obrigatórios**
- Atributos opcionais
- Atributos com valores default

Pessoa
+ nome: string + renda: float + email: string + estrangeiro: bool
+ exibe_dados()

Herança

Herança oferece um mecanismo para derivar classes especializadas, permitindo **aproveitamento** de código e evitar **duplicação** de código.

```
class Usuario:
    def __init__(self, email: str):
        self.email = email
    def login(self):
        print(f"[{datetime.now()}] login: {self.email}")
```




```
class UsuarioPro(Usuario):
    pass
```

```
class UsuarioAdmin(Usuario, SystemAdm):
    pass
```

Classe abstrata

No Python classe abstrata é definida usando a classe **ABC** importada do módulo **abc**. Por exemplo, para definir que Pessoa é uma classe abstrata, basta declarar que Pessoa **herda** de **ABC** (*abstract base class*):

```
from abc import ABC, abstractmethod
class Pessoa(ABC):
    |   pass
```

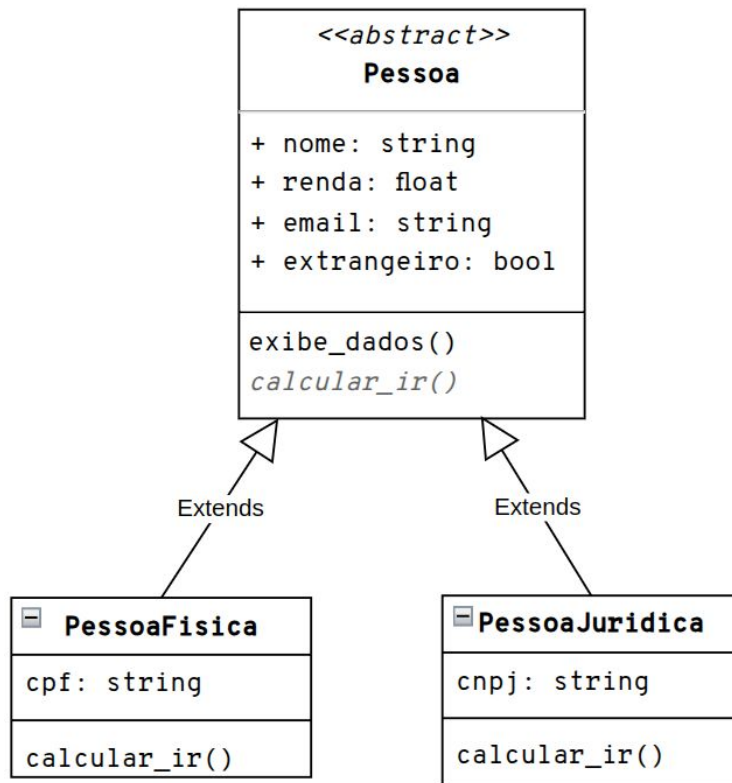


<<abstract>> Pessoa
+ nome: string + renda: float + email: string + estrangeiro: bool
exibe_dados() calcular_ir()

Método abstrato

`@abstractmethod` é um decorador que permite "anotar" um **método** como sendo **abstrato**:

```
from abc import ABC, abstractmethod  
class Pessoa(ABC):  
    → @abstractmethod  
    def calcula_ir(self) -> float:  
        pass
```



Acesso ao construtor da Superclasse

Se a **mesma** lógica de inicialização é usada para *todas* as **sub-classes** de Pessoa, podemos encapsular esse código no **construtor** da classe abstrata.

Na sub-classe PessoaFisica pode-se chamar o construtor de Pessoa usando a função **super()**

```
class PessoaFisica(Pessoa):  
    def __init__(self, nome: str):  
        # chama o construtor de Pessoa  
        super().__init__(nome)
```

