

A white line-art pattern of a circuit board on a white background, featuring various traces, pads, and components.

8

# TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

## Texto base

# 8

## Dependências externas e bibliotecas de terceiros

Prof. MSc. Rafael Maximo Carreira Ribeiro

### *Resumo*

*Neste capítulo veremos como instalar e utilizar módulos e pacotes externos, comumente chamados simplesmente de bibliotecas, fazendo uso do gerenciador de pacotes do Python. Aprenderemos também um pouco sobre os ambientes virtuais e sobre as ferramentas para gerenciamento de dependências.*

### **8.1. Introdução**

Podemos dizer que boa parte dos problemas que enfrentamos no dia a dia da programação são releituras de problemas comuns. Por exemplo, cada aplicação web existente é única e feita para atender a uma regra de negócio específica, mas ao mesmo tempo, boa parte do que ela precisa fazer é comum a todas as aplicações, como criar rotas, renderizar uma template html, gerar um arquivo JSON, fazer uma requisição http, etc. Portanto, existem ferramentas como Flask e Django, que são exemplos famosos no caso de aplicações web, que facilitam nosso trabalho de desenvolvimento, automatizando e abstraindo boa parte das funções comuns, de modo que podemos focar em desenvolver a parte da aplicação que é única para o nosso problema.

Considerando a maturidade da linguagem Python, que existe há mais de 30 anos e conta com uma comunidade global extremamente ativa e participativa, é seguro dizer que, para todos os problemas que precisamos resolver, provavelmente alguém em algum lugar já precisou resolver um problema parecido e fez um módulo para isso. E no raro caso de isso não ser verdade, esta pode ser uma ótima oportunidade para criar seu primeiro módulo e disponibilizá-lo para a comunidade. Qualquer um pode fazer isso e

não é preciso décadas de experiência para tanto, pois uma vez lançado, seu projeto poderá ganhar outros contribuidores que o ajudarão a avançar no seu desenvolvimento<sup>1</sup>.

Além de aprender sobre a instalação e utilização de pacotes, outro conceito muito importante é o de ambientes virtuais, *virtualenv*'s ou *venv*'s<sup>2</sup> como são muitas vezes chamados. Eles são essenciais quando precisamos trabalhar em mais de um projeto ou aplicação, pois é praticamente certo que em algum momento elas apresentarão uma incompatibilidade de dependências, então o uso de um ambiente virtual nos permite isolar não só os módulos mas também o interpretador do Python que será usado para cada aplicação ou projeto, que podem ter inclusive versões diferentes.

## 8.2. Ambientes virtuais

Imagine a seguinte situação, um projeto A precisa trabalhar com a versão 1.3 de um determinado módulo, para ser compatível com a interface que foi desenvolvida no seu lançamento por exemplo, mas outro projeto B, que está começando agora poderá utilizar a versão 3.0, mais recente e que traz novas ferramentas, mas que não é retrocompatível com as versões anteriores do módulo.

Nesta situação, não é possível ter os requisitos de ambos os projetos satisfeitos simultaneamente, pois ou instalamos a versão 1.3 para que o projeto A possa rodar ou instalamos a versão 3.0 para desenvolver o projeto B, e não conseguimos mais rodar o projeto A.

Para isso podemos criar um ambiente virtual, que nada mais é do que uma pasta com uma cópia da instalação do Python, na qual serão instalados os módulos e que será utilizada para rodar o projeto. Cada projeto ou aplicação passa então a ter o seu próprio ambiente virtual, de modo que evitamos qualquer tipo de conflito entre suas dependências.

Outra vantagem que decorre do uso de ambientes virtuais é que podemos manter a nossa instalação principal do Python limpa, apenas com os módulos principais e mais genéricos, como por exemplo o IPython, que é uma *Shell* alternativa ao IDLE.

Para criar um ambiente virtual do Python, podemos usar o módulo integrado `venv`, como mostra a Codificação 8.1.

### Codificação 8.1: Sintaxe para criação de um ambiente virtual do Python

```
> <executável do python> -m venv <nome do ambiente a ser criado>
```

Fonte: do autor, 2021

<sup>1</sup> Essa é uma das principais vantagens de se trabalhar com uma linguagem de código aberto, pois o desenvolvimento de novas ferramentas não fica limitado aos recursos e interesses de uma única empresa ou companhia, e todos podem contribuir.

<sup>2</sup> `venv` é a abreviação de *Virtual Environment*, em inglês.

A Codificação 8.2 mostra alguns exemplos desse comando para Windows e sistemas Unix (Linux e Mac).

#### Codificação 8.2: Exemplos de criação de um ambiente virtual

```
> py -m venv meu-venv # Windows (1)
> C:\Program Files\Python39\python -m venv venv39 # Windows (2)
> C:\Program Files\Python37\python -m venv venv37 # Windows (3)
> phython3 -m venv meu-venv # Linux e Mac (1)
> /usr/bin/phython3.5 -m venv venv35 # Linux e Mac (2)
> /usr/bin/phython3.8 -m venv venv38 # Linux e Mac (3)
```

Fonte: do autor, 2021

Nos exemplos da Codificação 8.2 marcados com (1), será criado um ambiente virtual com a versão que estiver configurada na variável de ambiente PATH do sistema, em geral é a versão mais recente do Python. Nos exemplos marcados com (2) e (3), estamos passando o caminho completo para o binário (executável) do Python<sup>3</sup>, portanto o ambiente virtual será criado com uma cópia do binário usado para sua criação. Dessa forma podemos controlar de maneira muito fácil qual versão do Python será usada em cada aplicação, bastando que tenhamos tal versão já instalada em nosso computador.

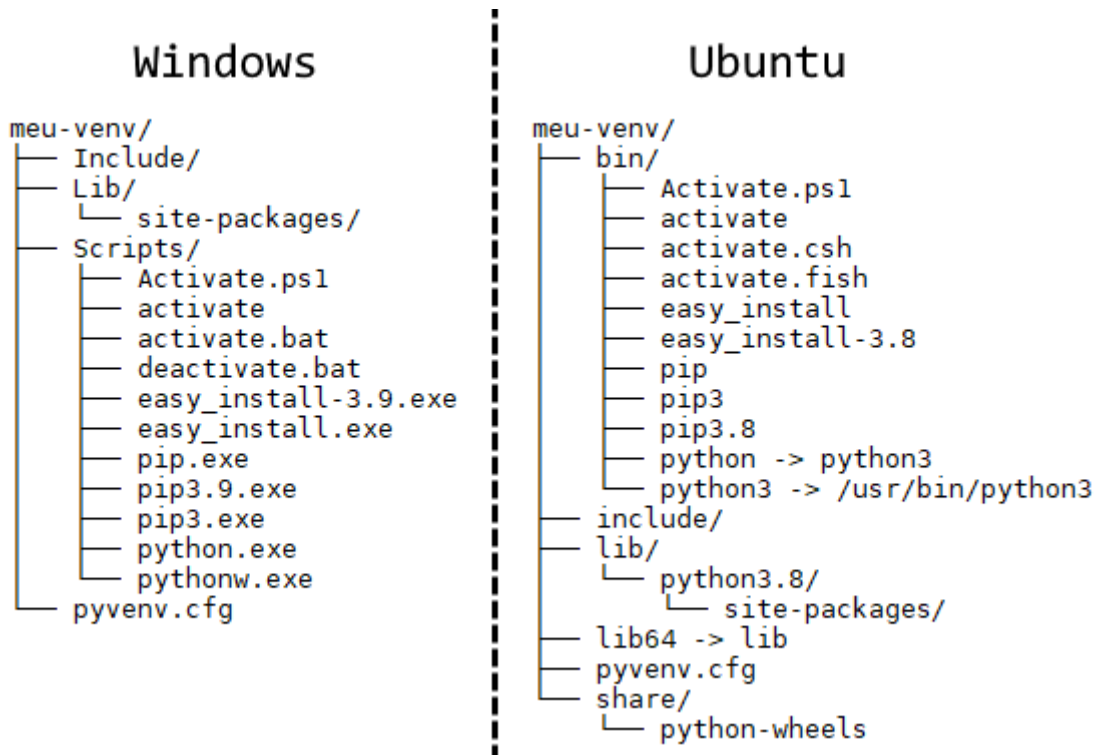
A Figura 8.1 mostra uma comparação entre a hierarquia inicial das pastas do ambiente virtual criado em um sistema Windows e Ubuntu, com a execução dos comandos `> py -m venv meu-venv` e `> python3 -m venv meu-venv`, respectivamente.

Nela podemos notar que há uma pasta com o executável do Python (`Scripts` no Windows e `bin` no Ubuntu), junto com alguns outros scripts que falaremos em breve, e há também uma pasta para a instalação dos pacotes que forem adicionados a este ambiente (`Lib/site-packages` no Windows e `lib/python3.8/site-packages` no Ubuntu).

A estrutura de diretórios criada para o ambiente virtual pode sofrer variações de acordo com o sistema operacional ou a versão do Python, mas o único momento que iremos acessar diretamente uma destas pastas é para ativar o ambiente virtual, portanto essas diferenças não são importantes. A interface que utilizaremos para instalar os pacotes será a mesma independente do sistema operacional ou versão do Python, e essa é uma das vantagens da portabilidade do Python, que irá internamente realizar as operações corretas em cada situação.

<sup>3</sup> O caminho exato poderá ser diferente dependendo do sistema operacional específico e das configurações feitas na instalação do Python, o exemplo mostra a localização mais comum para o executável do Python.

Figura 8.1: Pasta do ambiente virtual criado com o módulo venv do Python, comparação entre os sistemas operacionais Windows e Ubuntu



Fonte: do autor, 2021

Após criado, o ambiente virtual precisa ser ativado para que possamos começar a utilizá-lo. O processo de criação apenas faz a cópia do arquivo binário do Python e demais arquivos necessários para seu funcionamento. O modo de ativar o ambiente virtual vai depender novamente do sistema operacional e de qual terminal, também chamado de *Shell*, estamos usando, como mostrado na Tabela 8.1 (PSF, 2021a).

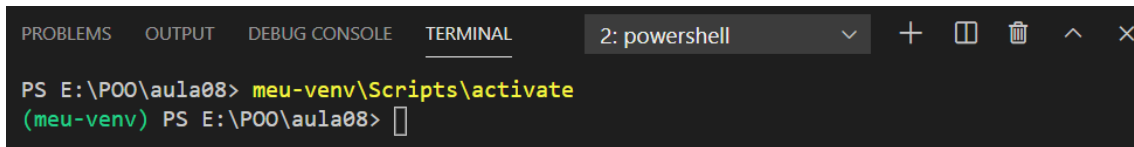
Tabela 8.1: Comandos para ativar o ambiente virtual por plataforma e terminal utilizado. Fonte: PSF (2021a)

Plataforma	Terminal / Shell	Comando para ativar o ambiente virtual
POSIX (Linux e Mac)	bash/zsh	\$ source <venv>/bin/activate
	fish	\$ source <venv>/bin/activate.fish
	csh/tcsh	\$ source <venv>/bin/activate.csh
	PowerShell Core	\$ <venv>/bin/Activate.ps1
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

Fonte: do autor, 2021

Após a ativação do ambiente virtual, a *Shell* irá mostrar o nome do ambiente em uso entre parênteses no começo da linha, como mostrado na Figura 8.2.

**Figura 8.2: Ativação do ambiente virtual no terminal do VSCode (PowerShell)**

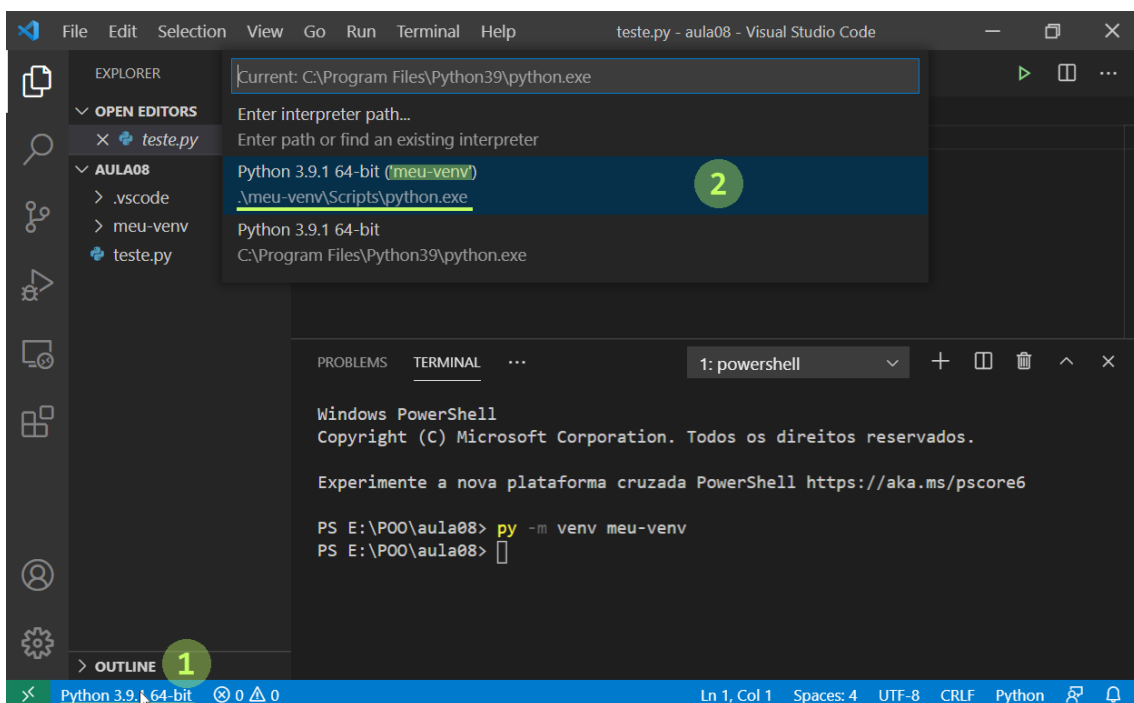


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: powershell
PS E:\P00\aula08> meu-venv\Scripts\activate
(meu-venv) PS E:\P00\aula08> █
```

Fonte: do autor, 2021

Caso você queira executar o modo de depuração do VSCode em um ambiente virtual, deverá escolher o interpretador que será utilizado. Caso o ambiente virtual tenha sido criado na raiz da pasta que está aberta no VSCode, ele irá automaticamente identificar a existência de um ambiente virtual e podemos então selecionar o executável abrindo um arquivo Python qualquer e clicando no canto inferior esquerdo da tela, como mostra a Figura 8.3.

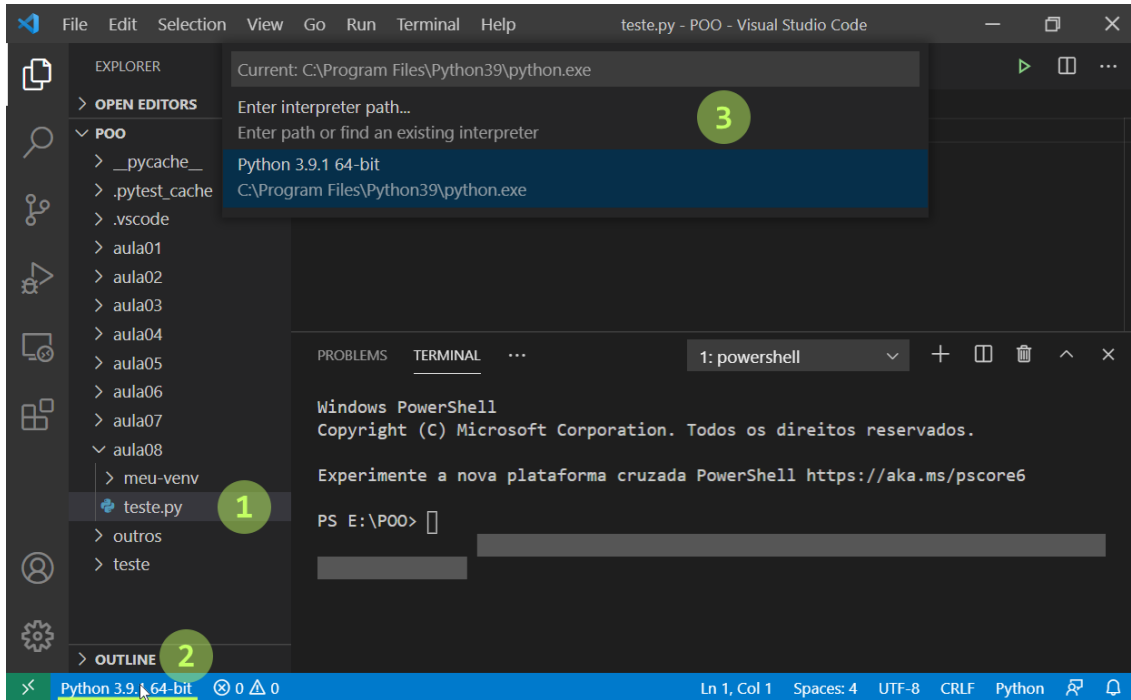
**Figura 8.3: Seleção do interpretador do Python no VSCode.**



Fonte: Elaborado pelo autor

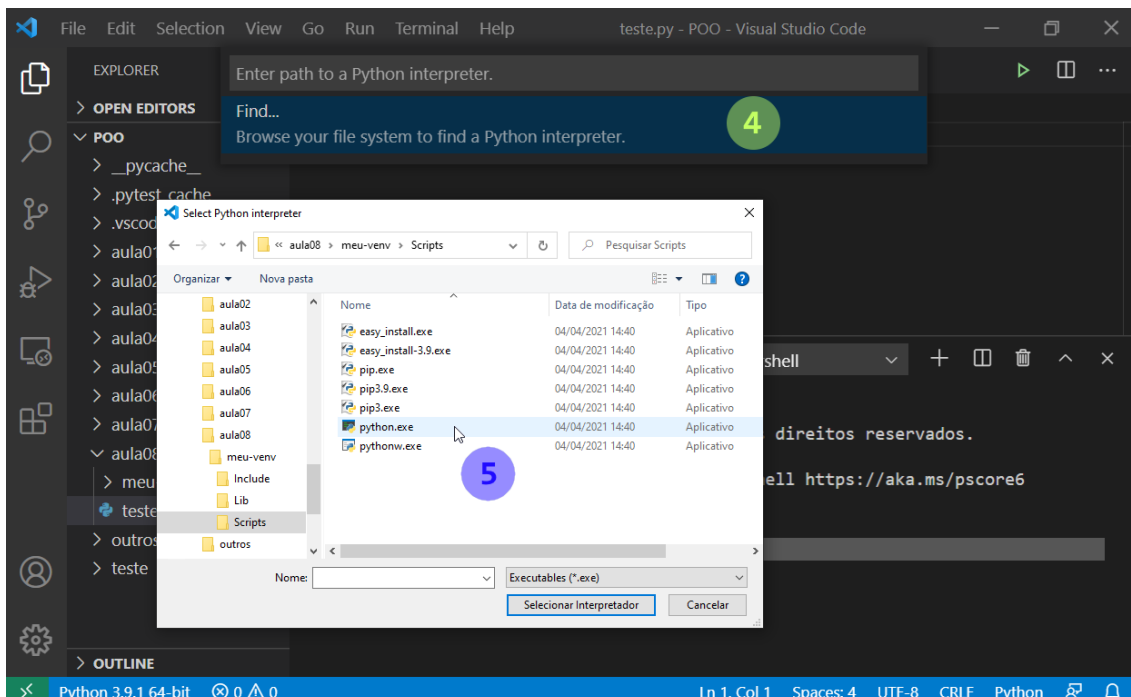
Se você estiver com outra pasta aberta ou pretende usar um ambiente virtual que não esteja na raiz, é possível indicar manualmente o caminho para o executável, como mostrado nas Figuras 8.4 e 8.5.

Figura 8.4: Seleção manual do interpretador do Python no VSCode - parte 1



Fonte: Elaborado pelo autor

Figura 8.5: Seleção manual do interpretador do Python no VSCode - parte 2



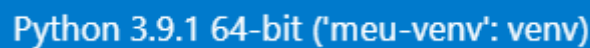
Fonte: Elaborado pelo autor

Primeiramente crie ou abra um arquivo Python (passo 1), em seguida clique na informação sobre o interpretador, no canto inferior esquerdo (passo 2), e então clique em “entrar caminho para o interpretador”. Aqui você pode digitar o caminho se souber,

ou então fazer como no passo 4 da Figura 8.4 e clicar em “encontrar” para navegar até o interpretador do Python (arquivo python.exe no Windows ou apenas python no Linux e Mac) que se encontra no seu ambiente virtual, pasta Scripts no Windows ou bin no Linux e Mac.

Após a alteração, a barra de status inferior do VSCode será atualizada para exibir o interpretador do Python selecionado, como mostra a Figura 8.6.

**Figura 8.6: Visualização do interpretador selecionado no canto inferior direito do VSCode.**



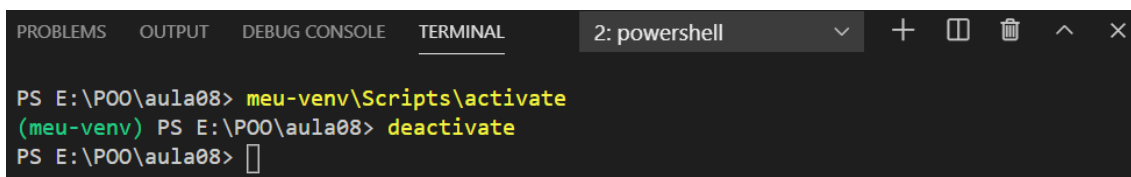
Python 3.9.1 64-bit ('meu-venv': venv)

Fonte: do autor, 2021

Um nome comumente utilizado para essa pasta é `.venv`, em especial em sistemas Unix, nos quais o terminal trata nomes que começam com um ponto como arquivos ou pastas ocultos.

Por fim, para encerrar o ambiente virtual, apenas execute o comando `deactivate`, como mostrado na Figura 8.7.

**Figura 8.7: Desativando o ambiente virtual.**



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: powershell
PS E:\P00\aula08> meu-venv\Scripts\activate
(meu-venv) PS E:\P00\aula08> deactivate
PS E:\P00\aula08> 
```

Fonte: do autor, 2021

### 8.3. Observações

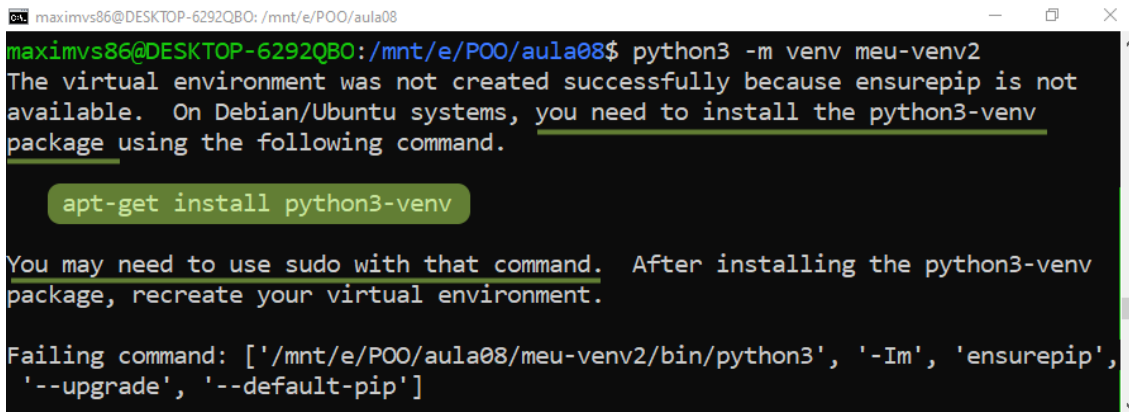
Caso você esteja usando Linux, é possível que o módulo `venv` não esteja completamente instalado, então ao tentar criar o ambiente você verá um erro. Em geral o próprio sistema avisa o que deve ser feito para corrigir o problema, com uma mensagem semelhante a da Figura 8.8. Se for esse o caso, execute o comando dado como administrador (`sudo`) para instalar o módulo. No exemplo da figura, o comando<sup>4</sup> é:

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

<sup>4</sup> Pode ser usado tanto `apt` quanto `apt-get`, com o mesmo resultado.



Figura 8.8: Erro ao criar um ambiente virtual pela primeira vez em sistemas Debian/Ubuntu



```
maximvs86@DESKTOP-6292QBO: /mnt/e/POO/aula08
maximvs86@DESKTOP-6292QBO:/mnt/e/POO/aula08$ python3 -m venv meu-venv2
The virtual environment was not created successfully because ensurepip is not
available. On Debian/Ubuntu systems, you need to install the python3-venv
package using the following command.

apt-get install python3-venv

You may need to use sudo with that command. After installing the python3-venv
package, recreate your virtual environment.

Failing command: ['/mnt/e/POO/aula08/meu-venv2/bin/python3', '-Im', 'ensurepip',
'--upgrade', '--default-pip']
```

Fonte: Elaborado pelo autor

Caso você esteja usando a PowerShell no Windows, é possível que precise alterar as permissões de execução de scripts para poder ativar o ambiente virtual. Para isso, abra como administrador uma janela da PowerShell independente e execute o comando da Codificação 8.2. Para executar como administrador, vá ao menu iniciar, pesquise por “PowerShell” e clique em “Executar como administrador”, como mostra a Figura 8.9.

**Codificação 8.3: Alteração das permissões da PowerShell no Windows para permitir a execução do script de ativação do ambiente virtual**

```
PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Fonte: do autor, 2021

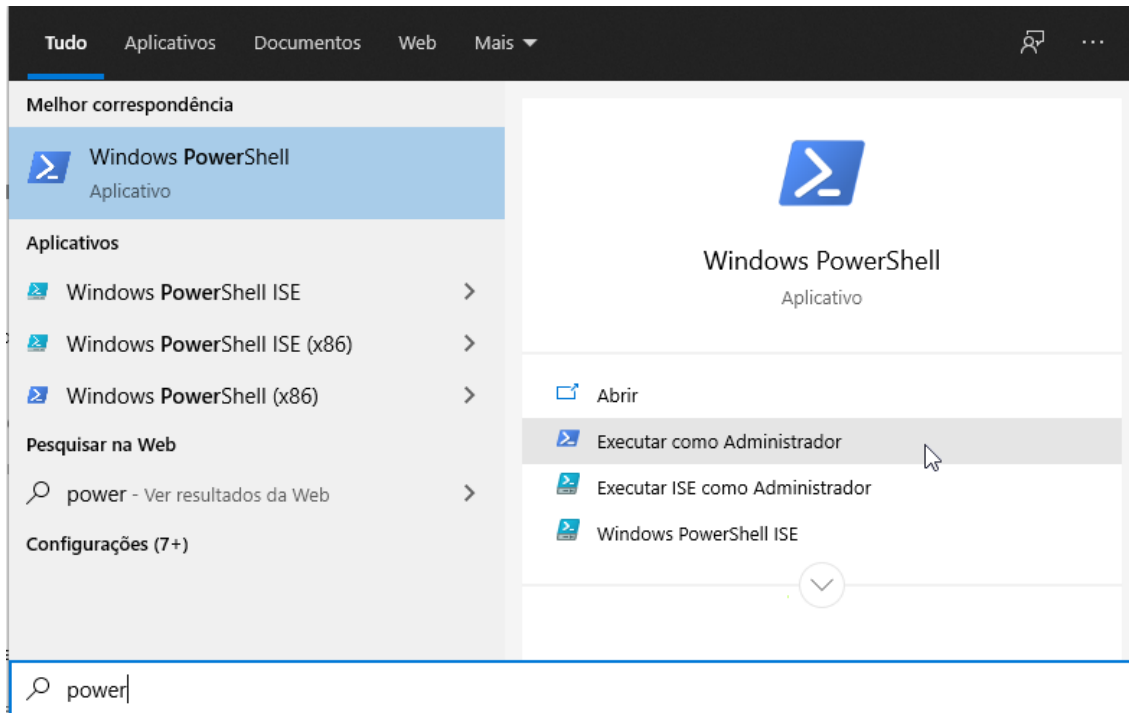
Para voltar às configurações padrão das permissões de execução de scripts na PowerShell, execute no terminal com privilégios de administrador o comando da Codificação 8.3. Lembrando que isso efetivamente impede que um ambiente virtual seja ativado via um terminal da PowerShell.

**Codificação 8.4: Restabelece a configuração alterada pelo comando da Codificação 8.2 para o valor padrão**

```
PS C:> Set-ExecutionPolicy -ExecutionPolicy Undefined -Scope CurrentUser
```

Fonte: do autor, 2021

Figura 8.9: Executar terminal da PowerShell como administrador



Fonte: do autor, 2021

#### 8.4. Instalando pacotes com PIP

Uma vez com o ambiente virtual ativado, o gerenciador de pacotes do Python, `pip`, irá automaticamente instalar os pacotes e módulos no lugar certo, portanto independente de onde estamos instalando um módulo Python, seja na instalação do sistema ou em um ambiente virtual, os comandos são exatamente os mesmos.

O `pip` consegue instalar pacotes do índice PyPI, de urls de projetos em sistemas de versionamento (github, gitlab, etc.), projetos locais e arquivos fonte locais ou remotos. Além disso, é possível instalar com um único comando todos os pacotes listados em um arquivo texto de requisitos, comumente nomeado `requirements.txt`, ou alguma variação, como `dev-requirements.txt`, por exemplo, para indicar a finalidade do arquivo em questão.

A lista completa de comandos e opções disponíveis para gerenciar pacotes com o `pip` pode ser vista na documentação oficial (PyPA, 2021), mas os comandos mais utilizados são:

- `pip install`: para instalar um módulo ou uma lista de módulos em um arquivo de requisitos, o comando a seguir instala o módulo `requests`:  

```
> pip install requests
```

Já o comando a seguir instala todos os módulos listados no arquivo `meus_requisitos.txt`, que está na mesma pasta a partir da qual o comando está sendo executado.

```
> pip install -r meus_requisitos.txt
```

- `pip show`: mostra informações sobre um módulo específico, como por exemplo:

```
> pip show requests
Name: requests
Version: 2.25.1
Summary: Python HTTP for Humans.
Home-page: https://requests.readthedocs.io
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: e:\poo\aula08\meu-venv\lib\site-packages
Requires: certifi, urllib3, chardet, idna
Required-by:
```

- `pip list`: lista todos os módulos atualmente instalados no ambiente em questão, como por exemplo:

```
> pip list
Package      Version
-----
certifi      2020.12.5
chardet      4.0.0
idna         2.10
pip          20.2.3
requests     2.25.1
setuptools   49.2.1
urllib3      1.26.4
```

Observe que nesse ambiente foi instalado apenas o módulo `requests`, mas como este módulo depende de outros para funcionar, o `pip` os instalou automaticamente.

- `pip freeze`: cria uma imagem da situação atual do ambiente, listando todos os módulos que estão instalados e qual versão de cada módulo. O retorno é mostrado no próprio terminal, como a seguir:

```
> pip freeze
certifi==2020.12.5
idna==2.10
requests==2.25.1
urllib3==1.26.4
```

Observe que nesta lista não aparecem o `pip` e o `setuptools`, mas eles apareciam com o comando `list`, isso ocorre pois ambos os módulos são instalados pelo Python durante a criação do ambiente virtual, então não há necessidade de serem incluídos na saída deste comando.

Uma forma fácil de criar um arquivo com essa lista é usar o mecanismo da própria *Shell* ou terminal para redirecionar e salvar o retorno de um comando para um arquivo, que é feito com o sinal de maior, como a seguir:

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

- `pip uninstall`: remove um módulo diretamente ou todos os módulos listados em um arquivo. A utilização é análoga ao comando `pip install`, mas com o efeito inverso.

```
> pip uninstall requests
Found existing installation: requests 2.25.1
Uninstalling requests-2.25.1:
  Would remove:
    e:\poo\aula08\meu-venv\lib\site-packages
      \requests-2.25.1.dist-info\*
    e:\poo\aula08\meu-venv\lib\site-packages\requests\*
Proceed (y/n)? y
Successfully uninstalled requests-2.25.1
```

Se você testar o comando `pip list` novamente, verá que apenas o módulo `requests` foi removido, mas não as suas dependências. Isso ocorre porque o `pip` é um gerenciador de pacotes mais simples e não faz uma verificação completa de todas as dependências que não estão sendo utilizadas. Uma forma simples de remover os demais módulos é gerar um arquivo de requisitos com o `pip freeze` (para um arquivo `remover.txt`, por exemplo), editá-lo para conter apenas os módulos que deseja remover, e por fim executar o seguinte comando:

```
> pip uninstall -r remover.txt
```

## 8.5. Gerenciando dependências

Para um projeto simples, com poucas dependências, é possível fazer esse gerenciamento manualmente, usando o comando `pip freeze` que vimos. Ele cria uma imagem exata da condição atual do ambiente de desenvolvimento, fixando as versões exatas de cada módulo instalado para que o ambiente possa ser replicado com exatidão em outra máquina ou servidor.

No entanto, conforme o projeto cresce em complexidade e número de bibliotecas externas utilizadas, fica mais difícil gerenciar manualmente a compatibilidade de todas as subdependências do projeto. Nesta situação, é recomendado o uso de outras ferramentas que fazem uma verificação muito mais detalhada das lista de pacotes requeridos, avisando por exemplo se for encontrada alguma incompatibilidade entre as dependências que não seja possível resolver de maneira automática.

A ferramenta recomendada pela documentação do Python (PSF, 2021c) para uso geral é o Pipenv, que possibilita o gerenciamento de ambientes virtuais, dependências e importação de pacotes por uma interface de linha de comando mais avançada que o pip, pois inclui também o Pipfile<sup>5</sup> e o `virtualenv`. Ou seja, com uma única ferramenta é possível gerenciar a criação de ambientes virtuais, instalação e verificação de dependências e a lista de pacotes do ambiente. Para aprender mais sobre o uso do Pipenv, leia a documentação citada neste parágrafo.

Caso o Pipenv não atenda as necessidades do projeto, o que pode ocorrer por diferentes motivos, a documentação recomenda também outras ferramentas similares:

- **poetry**: é uma ferramenta muito semelhante ao Pipenv, mas com foco em projetos que serão distribuídos como pacotes do Python.
- **hatch**: ferramenta com mais opções, como incremento de versão, aplicação de tags de lançamento e criação de templates.
- **pip-tools**: ferramenta para construção de processos de gerenciamento de dependências personalizados.
- **micropipenv**: Ferramenta que adiciona suporte para trabalhar com arquivos das demais ferramentas em um mesmo projeto, possibilitando a conversão entre os arquivos Pipenv, Poetry lock, requirements.txt e arquivos compatíveis com pip-tools.

---

<sup>5</sup> É a lista de pacotes do projeto, que o Pipenv utiliza para avaliar as dependências e gerar uma versão com as versões travadas, calculando as hashes dos arquivos e salvando todas essas informações em um arquivo `Pipfile.lock`, que pode ser usado em seguida para replicar a configuração do ambiente com muito mais confiabilidade que um arquivo padrão de requisitos (`requirements.txt`).

## 8.6. PEP 8

Para melhorar a legibilidade dos módulos que estão sendo importados, a PEP 8 define (PSF, 2021e) as seguintes recomendações relativas a como escrever os comandos de importação.

- Importações devem estar em linhas separadas, com uma para cada módulo:

```
# correto:  
import os  
import sys
```

```
# incorreto:  
import os, sys
```

- No entanto, importações parciais podem estar na mesma linha:

```
# correto:  
from math import pi, sin, cos, tan
```

- Importações devem estar sempre no topo do arquivo, imediatamente após os comentários ou *strings* de documentação do módulo e antes da definição das variáveis e constantes globais.
- As importações devem ser agrupadas em 3 grupos e separadas por uma linha em branco, na seguinte ordem:
  - Módulos da biblioteca padrão;
  - Módulos de terceiros;
  - Módulos locais específicos da aplicação.
- **Dica:** no VSCode, é possível clicar com o botão direito em um arquivo Python e selecionar a opção “*Sort Imports*”, que ele irá automaticamente organizar os comandos de importação seguindo as recomendações da PEP 8.
- Para importações de módulos locais, deve-se dar preferência à importação absoluta, deixando a importação relativa explícita para situações específicas.

```
# importação absoluta  
from meuapp.modulos.texto import format_cpf
```

```
# importação relativa explícita  
from .texto import format_cpf
```

- Importação com asterisco deve ser evitada, pois dessa forma só é possível determinar quais nomes serão carregados para o espaço de nomes em tempo de execução, e isso confunde tanto leitores do código quanto ferramentas automatizadas de verificação e de autocompletar código, por exemplo.

## 8.7. Bibliotecas notáveis

Para buscar módulos no índice do PyPI é possível utilizar o próprio pip, com o comando `pip search`, mas em geral é mais prático fazer a busca no site do índice (<https://pypi.org/>).

Uma rápida pesquisa na internet irá te trazer dezenas de páginas listando as 10, 20 ou 50 bibliotecas que todo desenvolvedor precisa saber. Enquanto por um lado é sempre bom ter curiosidade e estudar por conta própria, tentar aprender sobre todos os principais módulos sem ter um objetivo ou necessidade específica tem um potencial grande para te deixar frustrado e desanimado, pois há milhares de módulos interessantes e bons, e em geral é preciso um certo tempo trabalhando com um módulo para entender suas nuances.

Sendo assim, é melhor aprender bem alguns poucos módulos e pesquisar e estudar os demais conforme a necessidade surgir, assim você sentirá seu esforço recompensado ao conseguir aplicar aquilo que aprendeu para resolver um problema real que estava enfrentando.

A seguir há uma pequena lista, definitivamente não exaustiva, de alguns módulos bastante utilizados em Python, sendo que muitas vezes vários módulos são usados em conjunto, agregando funcionalidades para resolver o problema em questão.

- Requests: para realizar requisições HTTP;
- Django, Flask e FastAPI: para criar aplicações web e API's;
- SQLAlchemy: para gerenciar bancos de dados;
- NumPy: para qualquer problema que envolva o processamento de grandes quantidades de dados de maneira eficiente;
- SciPy: para processamento de imagem e machine learning;
- Matplotlib: para criação de gráficos;
- Tkinter: para criação de interfaces gráficas;
- NLTK: para processamento de linguagem natural;
- Pytest: para criação de testes automatizados;
- Selenium: para criação de testes automatizados que simulam a navegação em uma página da internet;
- Pillow: para manipulação de imagens;
- PyGame: para criação de jogos;

Lembrando que esta lista mostra apenas alguns exemplos, mas existem muitas bibliotecas para resolver o mesmo tipo de problema, cada uma com a sua própria abordagem e com vantagens e desvantagens em diferentes contextos.

## Bibliografia

MARTIN, R. C. **Design principles and design patterns**. 2000. Disponível em: <[https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles\\_and\\_Patterns.pdf](https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)>. Acesso em: 15 fev. 2021.

PSF. **Venv**: criação de ambientes virtuais. 2021a. Disponível em: <<https://docs.python.org/pt-br/3/library/venv.html>>. Acesso em: 04 abr. 2021.

PSF. **O Tutorial python**: ambientes virtuais e pacotes. 2021b. Disponível em: <<https://docs.python.org/pt-br/3/tutorial/venv.html>>. Acesso em: 06 abr. 2021.

PSF. **Python packaging user guide**: Managing application dependencies. 2021c. Disponível em: <<https://packaging.python.org/tutorials/managing-dependencies/>>. Acesso em: 06 abr. 2021.

PSF. **Python packaging user guide**: installing packages. 2021d. Disponível em: <<https://packaging.python.org/tutorials/installing-packages/>>. Acesso em: 06 abr. 2021.

PSF. **Style guide for python code**: imports. 2021e. Disponível em: <<https://www.python.org/dev/peps/pep-0008/#imports>>. Acesso em: 06 abr. 2021.

PyPA. **PIP documentation**: reference guide. 2021. Disponível em: <<https://pip.pypa.io/en/stable/reference/#>>. Acesso em: 06 abr. 2021.