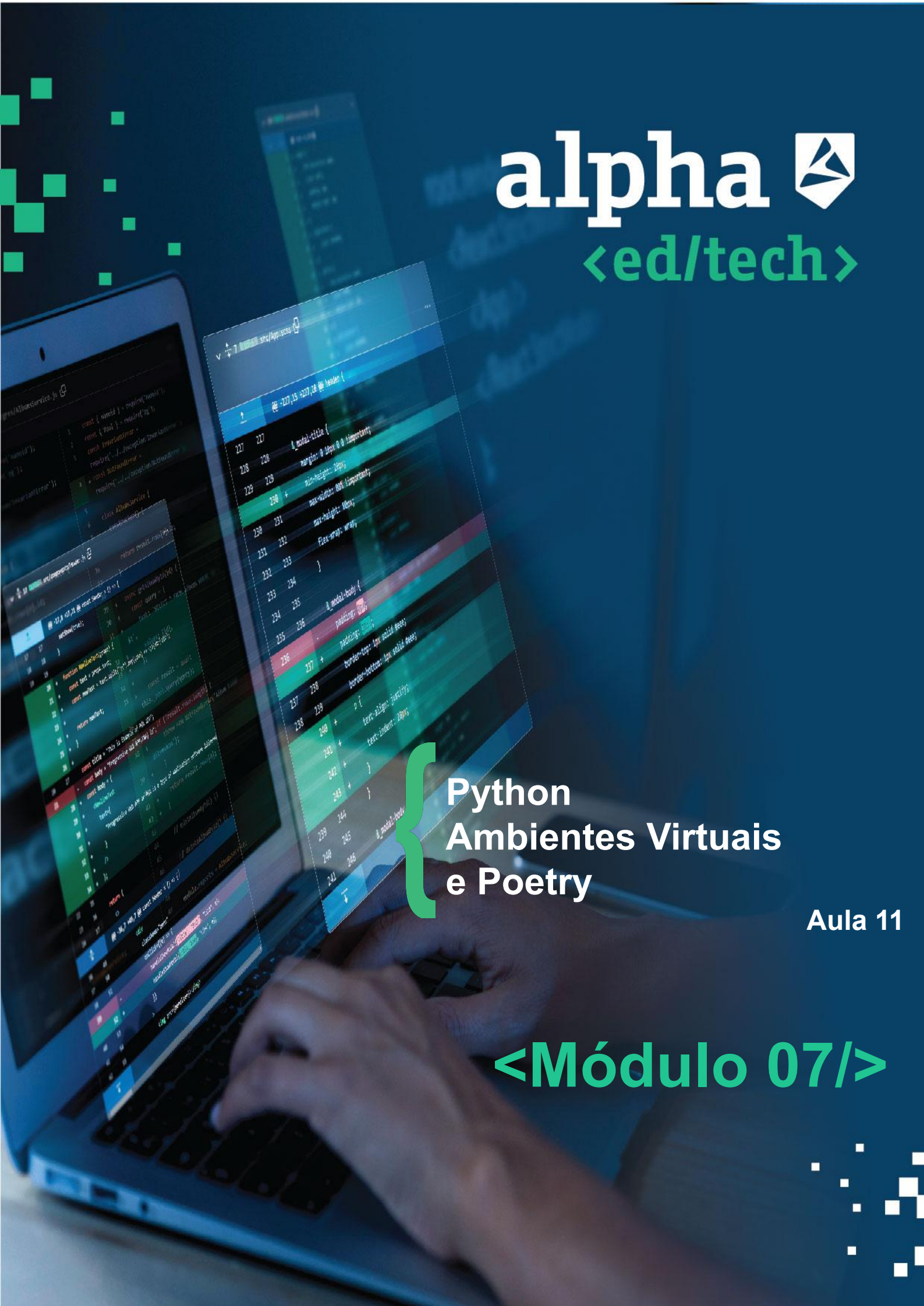


alpha
<ed/tech>

Python
Ambientes Virtuais e Poetry

Aula 11

<Módulo 07/>

Ambientes Virtuais e Poetry

Sumário

- Ambientes Virtuais e Gerenciadores de Dependências
- Gerenciando Versões do Python com o PyEnv
- Trocando a Versão do Python no VSCode
- Gerenciamento Manual de Dependências com pip
- Ambientes Virtuais: Isolando Dependências de Projetos
- Integração de Ambiente Virtual no VSCode
- Gerenciamento Avançado de Dependências com o Poetry
- Configurando o VSCode para Usar o Ambiente Virtual do Poetry
- Conclusão

Ambientes Virtuais e Poetry

Introdução

Nesta aula, vamos explorar o mundo dos ambientes virtuais e gerenciadores de dependências em Python.

Você pode estar se perguntando: "Por que eu preciso disso?".

A resposta é simples: organização e controle. À medida que você avança em seus projetos de programação, a complexidade aumenta.

Você começará a usar bibliotecas externas, diferentes versões do Python e, eventualmente, precisará compartilhar seu código com outras pessoas.

É aí que entram os ambientes virtuais e gerenciadores de dependências, ferramentas essenciais para manter seu código organizado, confiável e fácil de compartilhar.

Vamos começar nossa jornada entendendo o primeiro desafio: gerenciar diferentes versões do Python.

Gerenciando Versões do Python com o PyEnv

Motivação: Por que Precisamos de Múltiplas Versões do Python?

Imagine que você está trabalhando em um projeto pessoal e decide usar uma versão recente do Python, digamos, a 3.13. Tudo funciona perfeitamente.

Agora, imagine que você começa a colaborar em um projeto de uma empresa. Essa empresa, por razões de estabilidade e compatibilidade, usa Python 3.6 em seus servidores de produção (onde o código é executado para os usuários finais).

Problema: Se você desenvolver seu código usando recursos do Python 3.13 que não existem no 3.6, seu código simplesmente **não funcionará** quando for implantado nos servidores da empresa. Isso pode causar erros, bugs e, em última análise, perda de tempo e dinheiro.

Cenário hipotético: Sabe-se que na versão 3.8 do Python foi introduzido um novo operador `:=`, conhecido como "operador morsa". Ele permite realizar atribuições dentro de expressões (não se preocupe em entender o que ele faz, não é importante).

```
# Python 3.8 ou superior
lista = [1, 2, 3]
if (n := len(lista)) > 2:
    print(f"Lista com mais de 2 elementos, totalizando {n} elementos")
```

Se você tentar executar esse mesmo código em um ambiente com Python 3.6, que não tem o operador morsa, você obterá um erro de sintaxe:

```
  File "<stdin>", line 1
    if (n := len([1, 2, 3])) > 2:
          ^
SyntaxError: invalid syntax
```

Consequência: Você precisaria reescrever partes do seu código para torná-lo compatível com o Python 3.6, o que pode ser trabalhoso e propenso a erros.

Solução: Precisamos de uma maneira de **isolar** diferentes versões do Python em nossos computadores e alternar facilmente entre elas, dependendo das necessidades de cada projeto. Assim para trabalhar no projeto da empresa você trocaria para a versão 3.6 e ao voltar ao projeto pessoal você trocaria para a versão 3.13. É aí que entra o **PyEnv**.

O que é o PyEnv?

O PyEnv é uma ferramenta de linha de comando (ou seja, feita para ser usada pelo terminal) que permite instalar e gerenciar facilmente várias versões do Python em seu sistema. Ele atua como um intermediário, direcionando os comandos Python para a versão correta que você deseja usar em um determinado momento.

Como o PyEnv Funciona?

O PyEnv funciona interceptando a chamada do Intepretador Python (ou seja `python meu-script.py`) e redirecionando-a para a versão correta do Python, com base na configuração do PyEnv. Ele faz isso manipulando a variável de ambiente `PATH`, que é uma lista de diretórios onde o sistema operacional procura por executáveis (nesse caso o executável `python`).

Gerenciando Versões do Python com o PyEnv

Explicando a variável de ambiente PATH:

Quando você digita um comando no terminal, como `python` ou `pip`, o sistema operacional precisa saber onde encontrar o programa correspondente. Imagine que você está em uma biblioteca enorme e quer encontrar um livro específico. Você não vai procurar em todas as prateleiras, certo? Provavelmente, você consultaria o catálogo para descobrir em qual seção e prateleira o livro está.

A variável de ambiente `PATH` funciona de forma semelhante. Ela é como um **catálogo** que diz ao sistema operacional onde procurar por programas executáveis. É uma lista de pastas (diretórios) onde o sistema operacional deve procurar quando você digita um comando.

É importante entender que a variável `PATH` não é específica do Python. Ela é uma configuração fundamental do sistema operacional, usada para localizar qualquer programa executável, não apenas o Python.

Como a busca funciona:

1. **Você digita um comando:** Por exemplo, você digita `python` (ou qualquer outro comando, como `git`, `node`, etc.) no terminal e pressiona Enter.
2. **O sistema operacional consulta o `PATH`:** O sistema operacional olha para a variável de ambiente `PATH`, que contém uma lista de diretórios.
3. **Busca sequencial:** O sistema operacional começa a procurar pelo arquivo executável chamado `python` em cada diretório listado no `PATH`, **na ordem em que aparecem**.
4. **Primeiro encontrado, primeiro executado:** Assim que o sistema operacional encontra um arquivo executável chamado `python` em um dos diretórios do `PATH`, ele **para de procurar e executa esse arquivo**.

Exemplo na prática:

Digamos que digito `python` no terminal. Em qual pasta o sistema operacional vai encontrar o executável `python.exe` ?

Para descobrir, precisamos olhar para a variável de ambiente `PATH`.

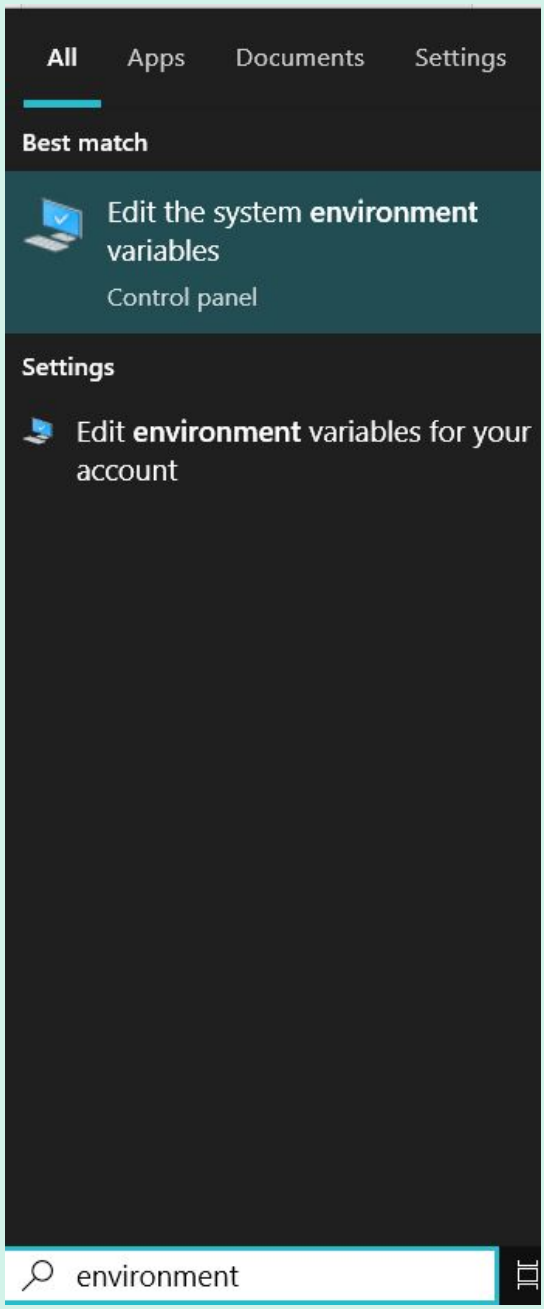
No linux, você pode ver o `PATH` com o comando `echo $PATH` num terminal.

No Windows, digite "environment" ou "ambiente" na busca do Windows e escolha a primeira opção:

(continua)

Gerenciando Versões do Python com o PyEnv

(continuação)

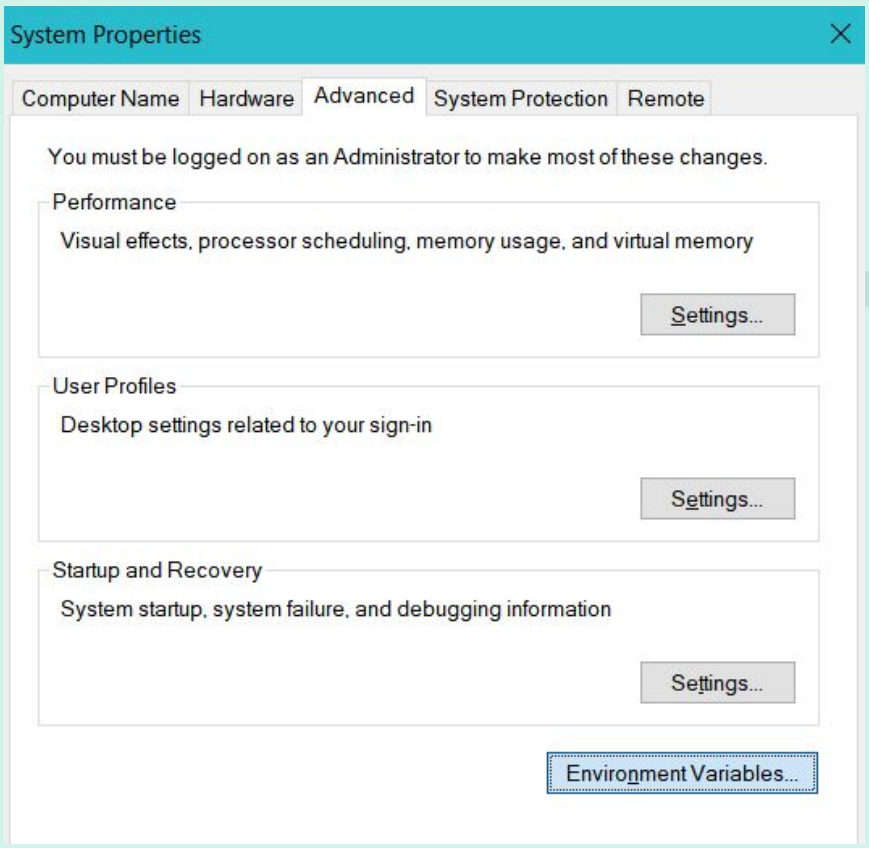


(continua)

Gerenciando Versões do Python com o PyEnv

(continuação)

Na janela que abriu, clique em "Environment Variables" / "Variáveis de ambiente":

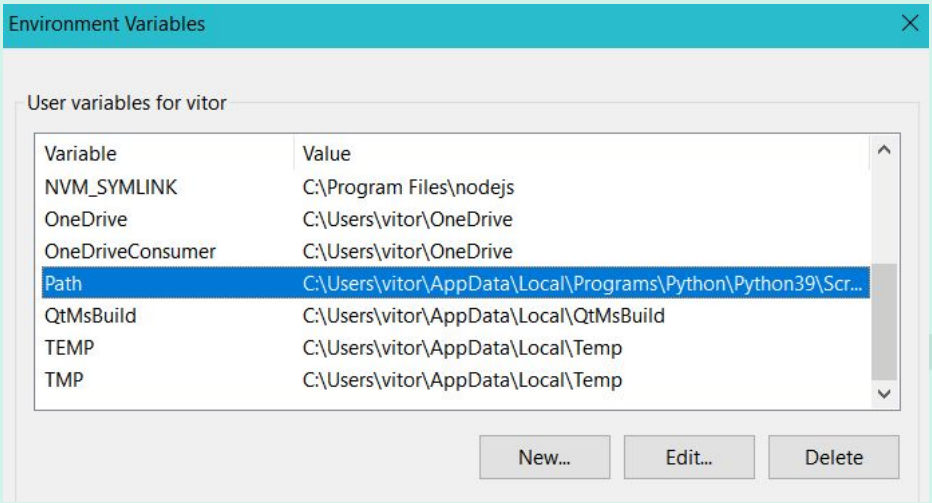


Agora role a primeira lista até encontrar "Path":

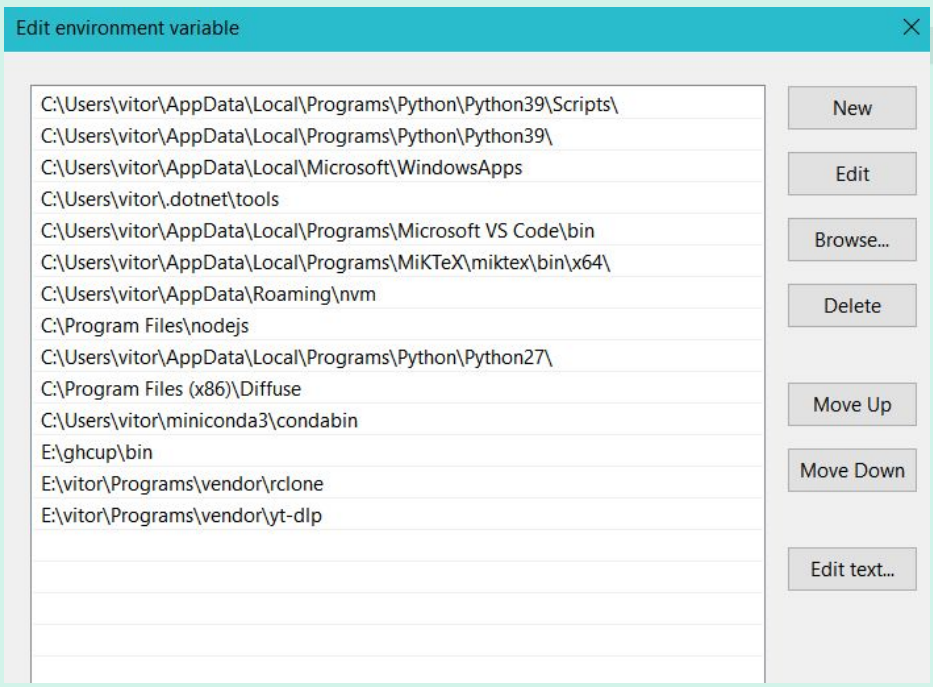
(continua)

Gerenciando Versões do Python com o PyEnv

(continuação)



Agora clique em "Edit" / "Editar":



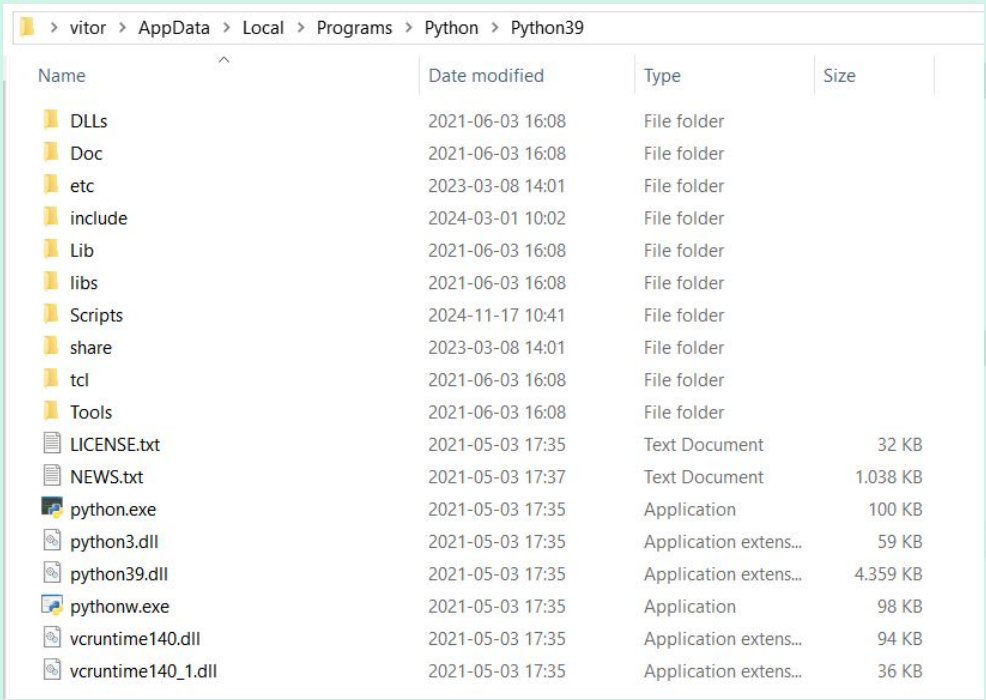
(continua)

Gerenciando Versões do Python com o PyEnv

(continuação)

Pronto ! Esta lista de pastas é a variável PATH.

Veja que no meu caso as primeiras duas pastas são pastas de instalação do Python (versão 3.9). Ao abrir a segunda pasta no explorador de arquivos, encontrei o "python.exe" que é o executável do Python:



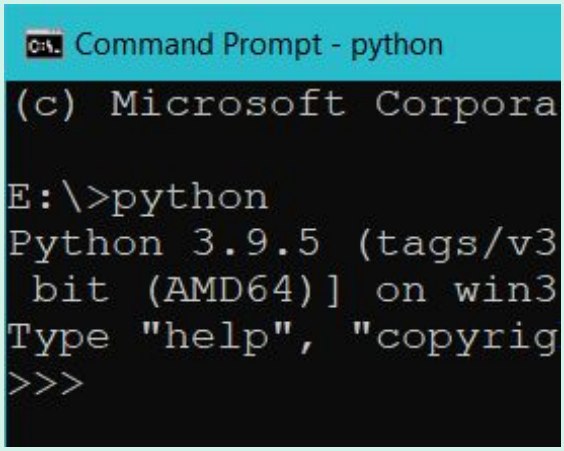
Portanto quando eu digito `python` (ou `python meu-arquivo.py` etc.) no meu terminal, o sistema operacional vai executar esse arquivo "python.exe" aí. Que é o python 3.9.

Posso comprovar isso rodando no terminal de fato e checando que o sistema operacional executa o Python 3.9.5:

(continua)

Gerenciando Versões do Python com o PyEnv

(continuação)



Obs: repare que no meu PATH tem também outra pasta onde está instalada outra versão do Python (python 2.7):

`C:\Users\vitor\AppData\Local\Programs\Python\Python27\`

Dentro dessa pasta também tem um "python.exe" mas ele nunca é executado quando eu digito `python` no terminal, porque o sistema operacional sempre executa o primeiro arquivo executável que encontra no `PATH` (nesse caso o python.exe 3.9).

Gerenciando Versões do Python com o PyEnv

O que o PyEnv faz?

O PyEnv insere seus próprios diretórios no **início** do `PATH`. Esses diretórios contêm pequenos scripts chamados "shims" que têm o mesmo nome dos comandos Python (como `python`, `pip`, etc.).

Quando você digita `python`, o sistema operacional encontra o "shim" do PyEnv primeiro (porque ele está no início do `PATH`). Esse "shim" então determina qual versão real do Python deve ser executada com base nas suas configurações do PyEnv e, finalmente, executa a versão correta do Python.

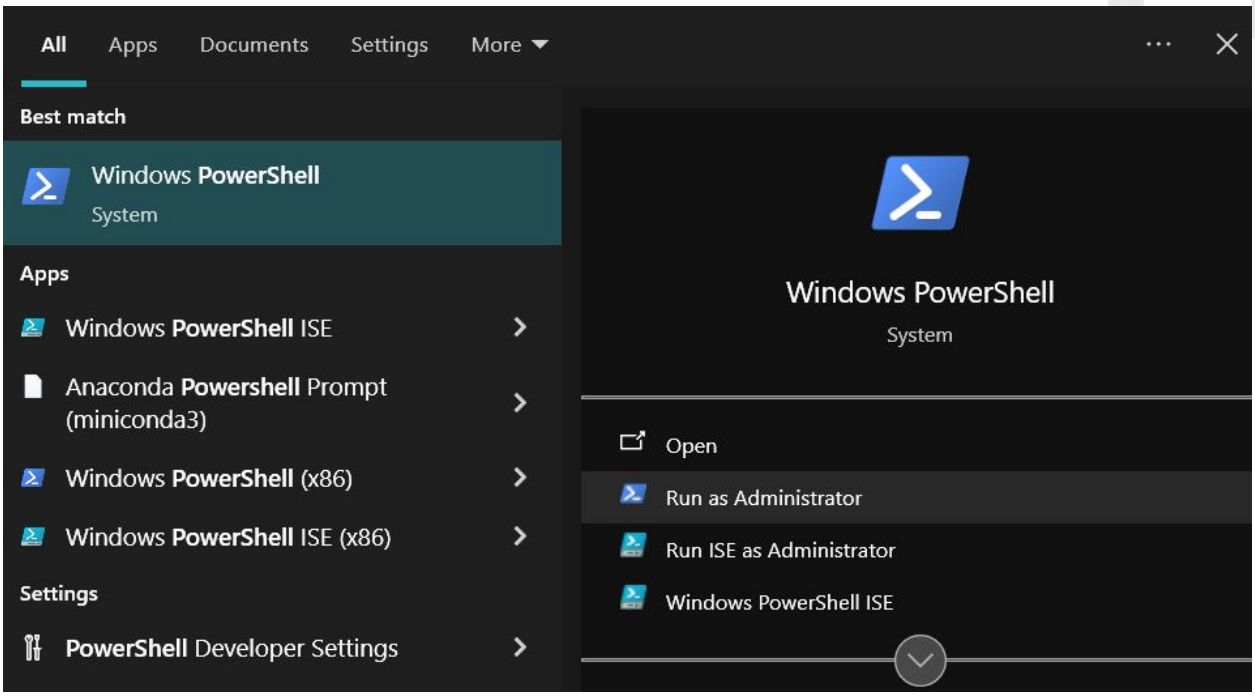
Em resumo: O `PATH` é uma lista de diretórios que o sistema operacional usa para procurar **qualquer programa executável**, não apenas o Python. O PyEnv manipula o `PATH` para garantir que a versão correta do Python seja executada quando você digita um comando Python no terminal, dando prioridade às suas próprias configurações. Isso permite que você alterne facilmente entre diferentes versões do Python para diferentes projetos.

Instalação do PyEnv

No Linux e MacOS, siga as instruções do repositório github do PyEnv:

- Linux: https://github.com/pyenv/pyenv?tab=readme-ov-file#linuxunix
- MacOS: https://github.com/pyenv/pyenv?tab=readme-ov-file#macos

No Windows, primeiro abra um powershell em modo Administrador. Para isso, digite powershell na busca do Windows e clique em Executar como Administrador:

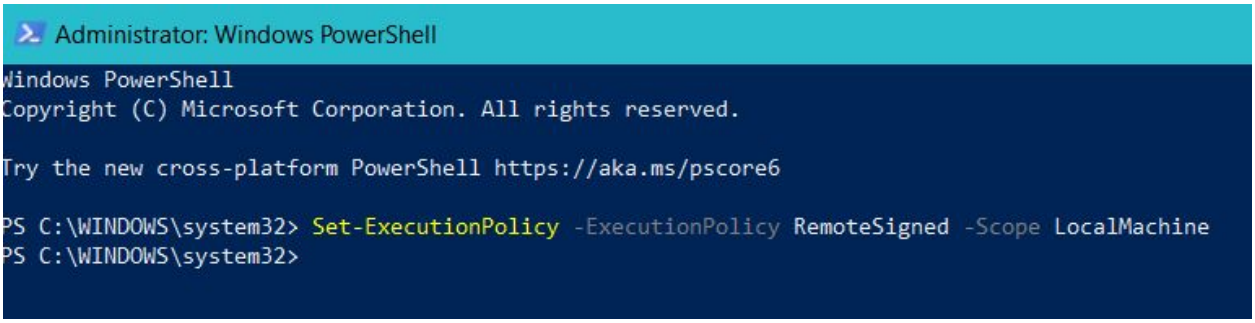


Gerenciando Versões do Python com o PyEnv

Nesse terminal, digite:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine
```

E aperte Enter:



Isso só é necessário fazer uma única vez na sua máquina, serve para permitir execução de scripts no powershell (porque para instalar o pyenv precisaremos rodar um script).

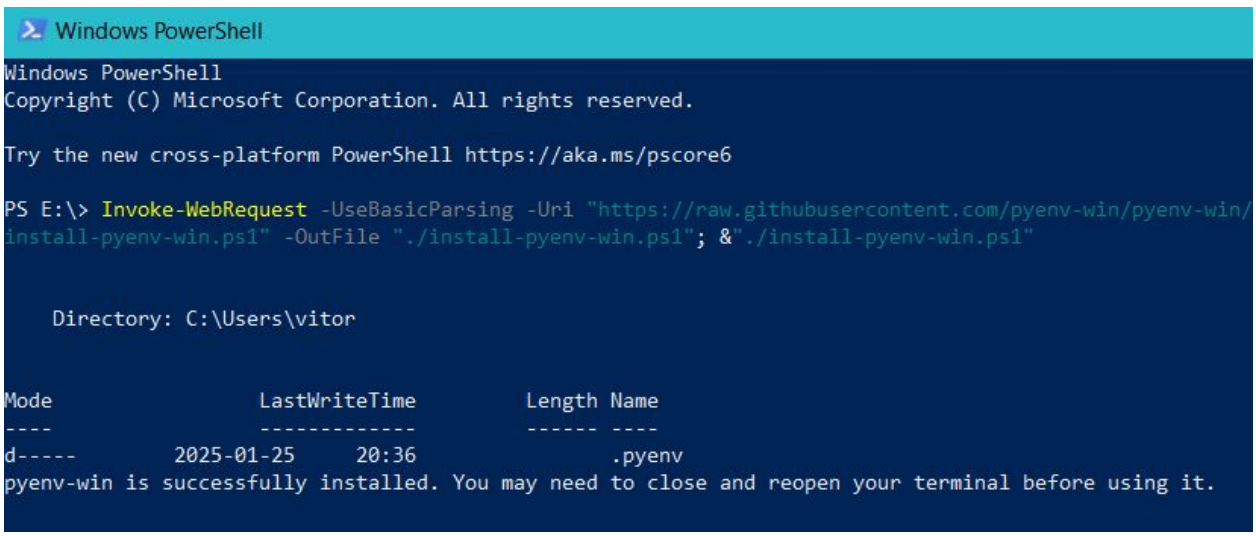
Agora **feche** esse powershell e abra outro que **não esteja em modo Administrador**.

Nesse novo terminal, digite:

```
Invoke-WebRequest -UseBasicParsing -Uri
"https://raw.githubusercontent.com/pyenv-win/pyenv-win/master/pyenv-win/install-pyenv
-win.ps1" -OutFile "./install-pyenv-win.ps1"; &"./install-pyenv-win.ps1"
```

E aperte Enter.

Espere alguns instantes e a instalação vai terminar:



O PyEnv foi instalado, ele não funcionará nesse mesmo terminal mas funcionará em todos os novos terminais que você abrir. Caso você use terminal pelo VSCode, terá que fechar todas as janelas do VSCode e abrir novamente para que o PyEnv seja encontrado e funcione.

Curiosidade: veja a variável PATH no seu sistema após a instalação do PyEnv e repare que ele colocou novas pastas no início do PATH como havíamos explicado.

Gerenciando Versões do Python com o PyEnv

Uso do PyEnv

1 — Instalando uma versão específica do Python:

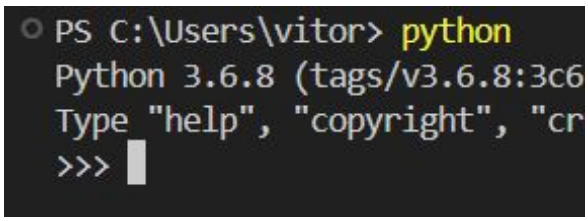
Para instalar por exemplo a versão 3.6.8:

```
pyenv install 3.6.8
```

Agora para trocar para essa versão, digite:

```
pyenv global 3.6.8
```

Agora ao digitar `python` será executado o Python 3.6.8:



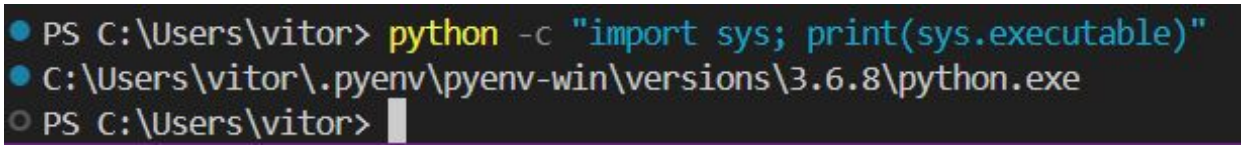
Isso é permanente, ou seja, se você abrir outro terminal e digitar `python` novamente, o Python 3.6.8 será executado (não precisa repetir o `pyenv global 3.6.8`).

2 — Onde o PyEnv instalou o Python ?

Para ver onde está o executável "python.exe" instalado pelo PyEnv, digite no terminal:

```
python -c "import sys; print(sys.executable)"
```

Isso vai executar o python e fazê-lo imprimir seu próprio nome de arquivo:



Gerenciando Versões do Python com o PyEnv

Vimos que foi instalado na pasta `C:\Users\vitor\.pyenv\pyenv-win\versions\3.6.8\`.

A pasta `C:\Users\vitor\.pyenv\pyenv-win\versions` é gerenciada pelo PyEnv. Quando você instala uma versão do python o PyEnv a armazena aí.

3 — Como instalar mais versões do Python ?

Para instalar mais versões do Python, basta repetir o comando `pyenv install <versão>`.

Daí para trocar permanentemente para essa versão, basta digitar `pyenv global <versão>`.

```
PS C:\Users\vitor> python -c "import sys; print(sys.executable)"
C:\Users\vitor\.pyenv\pyenv-win\versions\3.6.8\python.exe
PS C:\Users\vitor> pyenv install 3.13
:: [Info] ::  Mirror: https://www.python.org/ftp/python
:: [Info] ::  Mirror: https://downloads.python.org/pypy/versions.json
:: [Info] ::  Mirror: https://api.github.com/repos/oracle/graalpython/releases
:: [Downloading] ::  3.13.1 ...
:: [Downloading] ::  From https://www.python.org/ftp/python/3.13.1/python-3.13.1-amd64.exe
:: [Downloading] ::  To   C:\Users\vitor\.pyenv\pyenv-win\install_cache\python-3.13.1-amd64.exe
:: [Installing] ::  3.13.1 ...
:: [Info] :: completed! 3.13.1
PS C:\Users\vitor> pyenv global 3.13.1
PS C:\Users\vitor> python -c "import sys; print(sys.executable)"
C:\Users\vitor\.pyenv\pyenv-win\versions\3.13.1\python.exe
```

No exemplo acima:

- Imprimimos a localização do python, mostrando que estamos usando a versão 3.6.8
- Instalamos a versão 3.13.
- Trocamos para a versão 3.13.
- Imprimimos a localização do python novamente para confirmar que agora está sendo usada a versão 3.13.

Para voltar a usar a versão 3.6.8, basta digitar `pyenv global 3.6.8` (não deve repetir o `pyenv install` porque já foi instalado).

Todas as versões do python instaladas pelo PyEnv ficarão dentro da pasta `C:\Users\vitor\.pyenv\pyenv-win\versions\`. Essa é uma pasta gerenciada pelo PyEnv.

Dentro dessa pasta ficam as pastas de cada versão instalada:

- `C:\Users\vitor\.pyenv\pyenv-win\versions\3.6.8`
- `C:\Users\vitor\.pyenv\pyenv-win\versions\3.13.1`
- etc.

Cada uma delas tem seu próprio "python.exe" e outros arquivos e bibliotecas pertencentes à instalação do python.

Ao usar `pyenv global <versão>`, o PyEnv altera uma configuração que determina qual desses arquivos "python.exe" será executado quando você digitar `python` no terminal.

Gerenciando Versões do Python com o PyEnv

4 — Como retomar a versão do Python que estava sendo usada antes de instalar o PyEnv ?

Volte um pouco no texto, você deve se lembrar que antes de instalar o PyEnv, já existia no meu computador o Python 3.9.5 na pasta `C:\Users\vitor\AppData\Local\Programs\Python\Python39\`.

Agora que instalamos o PyEnv, **não é possível** fazer uso desse python mais, porque o comando `pyenv global <versão>` só permite trocar entre as versões de Python que foram instaladas pelo pyenv (com `pyenv install <versão>`).

Não significa que o python 3.9.5 foi desinstalado. Pelo contrário, a pasta dele continua intacta, o PyEnv não a modificou.

Mas como o PyEnv se insere no início da variável de ambiente PATH, ele ganha precedência sobre o python pré-instalado.

5 — Listando as versões instaladas pelo PyEnv:

```
pyenv versions
```

```
PS C:\Users\vitor> pyenv versions
  * 3.13.1 (set by C:\Users\vitor\.pyenv\pyenv-win\version)
    3.6.8
PS C:\Users\vitor>
```

O asterisco indica a versão ativa no momento.

6 — Como remover uma versão do Python ?

```
pyenv uninstall 3.6.8
```

```
PS C:\Users\vitor> pyenv versions
  * 3.13.1 (set by C:\Users\vitor\.pyenv\pyenv-win\version)
    3.6.8
PS C:\Users\vitor> pyenv uninstall 3.6.8
pyenv: Successfully uninstalled 3.6.8
PS C:\Users\vitor> pyenv versions
  * 3.13.1 (set by C:\Users\vitor\.pyenv\pyenv-win\version)
PS C:\Users\vitor>
```

Resumo do PyEnv

O PyEnv é uma ferramenta poderosa para gerenciar múltiplas versões do Python. Ele resolve o problema de conflitos entre versões e permite que você trabalhe em diferentes projetos com requisitos distintos de versão do Python.

Trocando a Versão do Python no VSCode

Motivação: VSCode e sua Versão Independente do Python

Até aqui, aprendemos a instalar e alternar entre diferentes versões do Python usando o PyEnv, e vimos como isso afeta o que é executado no terminal. No entanto, você pode encontrar situações em que o VSCode parece não reconhecer a versão do Python que você configurou com o PyEnv.

Problema:

Vamos supor que você tenha seguido os passos anteriores e instalado o Python 3.13.1 usando o PyEnv, e definido essa versão como global:

```
pyenv install 3.13.1
pyenv global 3.13.1
```

Você verifica no terminal e confirma que o Python 3.13.1 está sendo usado:

```
python -c "import sys; print(sys.executable)"
# Saída (na minha máquina):
# C:\Users\vitor\.pyenv\pyenv-win\versions\3.13.1\python.exe
```

Agora, você abre o VSCode e cria um arquivo chamado `main.py` com o seguinte código (não se preocupe se não entende o código, não é importante entender):

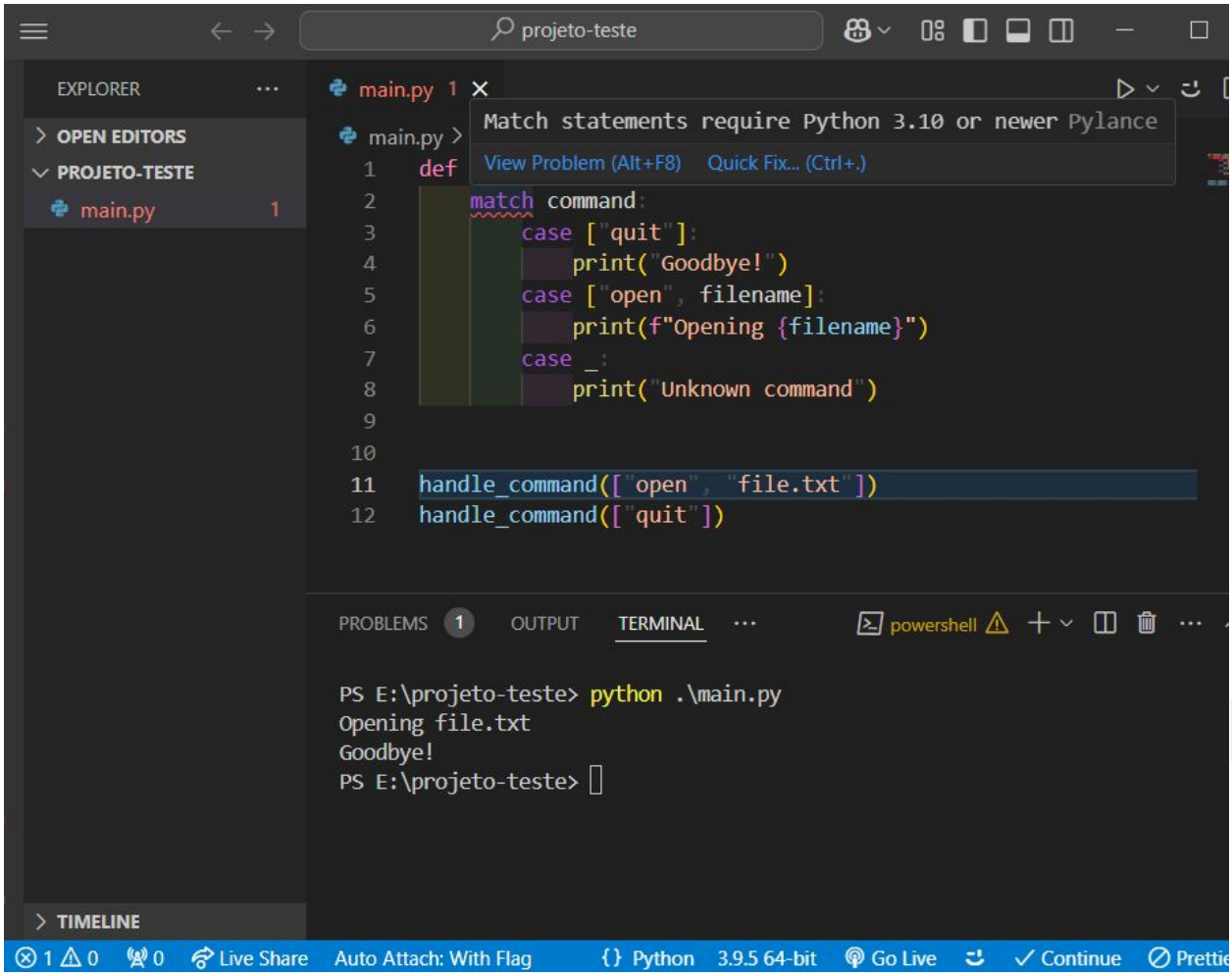
```
# main.py
def handle_command(command):
    match command:
        case ["quit"]:
            print("Goodbye!")
        case ["open", filename]:
            print(f"Opening {filename}")
        case _:
            print("Unknown command")

handle_command(["open", "file.txt"])
handle_command(["quit"])
```

Este código usa o comando `match/case`, que foi introduzido no Python 3.10. Se você executar este código no terminal usando `python main.py`, ele funcionará perfeitamente, pois o terminal está usando o Python 3.13.1.

No entanto, o VSCode pode mostrar um erro na linha do `match`:

Trocando a Versão do Python no VSCode



Repare que no terminal a execução do código funcionou, mas no VSCode ele acusa um erro no código.

Além disso, se você olhar na barra azul inferior do VSCode, repare que ele indica uma outra versão de python "3.9.5 64-bit".

Por que isso acontece?

Para fornecer recursos de **inteligência de código** (IntelliSense, como autocompletar, análise de erros, refatoração e navegação de código), o VSCode faz uso de uma das instalações de Python que ele detecta automaticamente em seu sistema, independentemente da versão que está configurada no seu terminal.

Em outras palavras, o VSCode escolhe qual interpretador Python usar de forma autônoma, independente da versão que o terminal está usando.

A versão do Python que o VSCode usa para esses recursos internos (IntelliSense) é **independente** da versão que você configurou com o PyEnv para o terminal.

Consequência:

O VSCode pode mostrar erros ou se comportar de forma inconsistente com o que você espera com base na versão do Python que você está usando no terminal.

Solução:

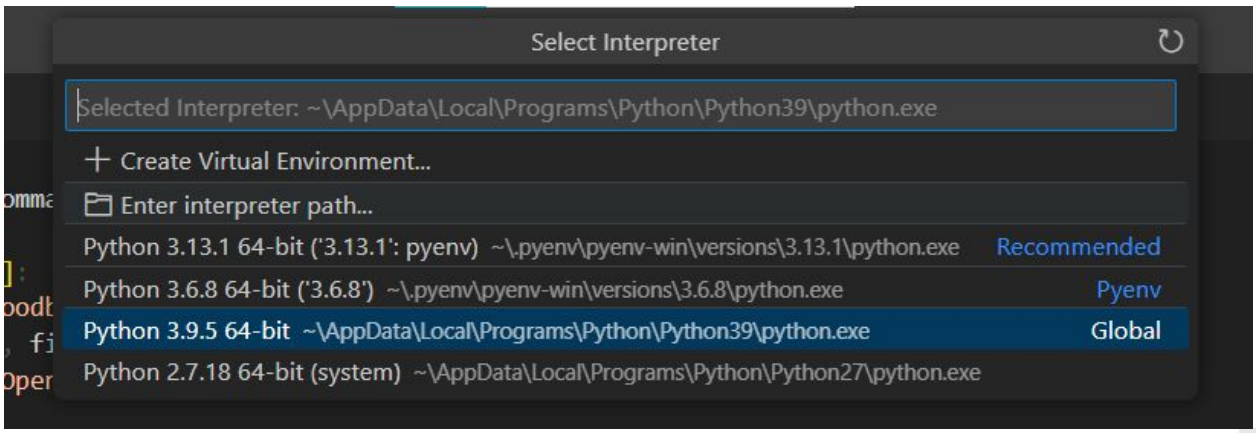
Precisamos configurar o VSCode para usar a mesma versão do Python que o terminal está usando (neste caso, 3.13.1).

Trocando a Versão do Python no VSCode

Como Configurar o VSCode para Usar a Mesma Versão do Terminal

Clique na barra inferior do VSCode onde ele mostra a instalação do Python que ele está usando (se isso não aparecer para você, aperte F1 e pesquise por "python interpreter", daí escolha a opção "Python: Select Interpreter").

Isso vai abrir uma janela onde você pode escolher qual instalação do python você quer que o VSCode use internamente para a Inteligência de Código:



Na imagem acima:

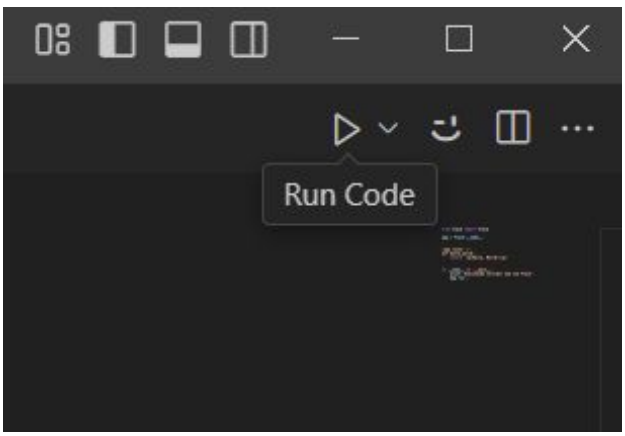
- 2.7.18 e 3.9.5 são versões que eu já tinha pré-instaladas antes de instalar o PyEnv.
- 3.6.8 e 3.13.1 são versões que instalei com o PyEnv.
- Obs: o "~" no começo de cada pasta é uma abreviação de `C:\Users\vitor`. É uma abreviação padrão de sistemas operacionais, existe em toda máquina.

Para trocar para a 3.13.1 basta clicar nela.

Com isso o VSCode vai começar a usar essa versão do Python para fazer a análise do código, então ele vai parar de reclamar sobre o comando `match`.

Não recomendamos a extensão Code Runner

Talvez você tenha instalada no seu VSCode a extensão Code Runner que permite executar código clicando num botão no lado direito superior do editor:



Trocando a Versão do Python no VSCode

Não recomendamos executar código python por meio dessa extensão (ou seja, clicando no botão) porque ela pode não usar a mesma instalação de Python que você configurou no VSCode (ou no terminal). Isso pode gerar erros de execução difíceis de entender, e são erros ilegítimos porque, se o código fosse executado no terminal com a instalação correta do Python, o erro poderia não ter acontecido.

Por isso recomendamos executar código python diretamente pelo terminal, assim você tem controle sobre qual instalação do Python será usada para executar o código.

Resumo

O VSCode usa um dos interpretadores Python instalados na máquina para fornecer recursos de inteligência de código.

Essa versão do Python é independente da versão configurada no terminal.

Para evitar inconsistências, é importante configurar o VSCode para usar a mesma versão do Python que você está usando no terminal.

Com essa configuração, você garante que o VSCode e o terminal estejam em sincronia, usando a mesma versão do Python, e você pode aproveitar todos os recursos do VSCode sem problemas de compatibilidade.

Por fim, não recomendamos usar a extensão Code Runner para executar código Python porque ela pode não usar a mesma instalação do Python que você configurou no VSCode/terminal, o que pode gerar erros de execução difíceis de entender. Prefira executar código Python diretamente pelo terminal.

Gerenciamento Manual de Dependências com pip

Agora que já vimos como gerenciar versões diferentes do Python, nossa próxima temática será sobre ambientes virtuais.

Mas antes de mergulharmos nos ambientes virtuais, vamos entender como gerenciar dependências manualmente usando o `pip`, o instalador de pacotes padrão do Python.

Dependências são bibliotecas e módulos que você usa em seus projetos Python.

Embora os ambientes virtuais sejam a prática recomendada, entender o `pip` e o gerenciamento manual de dependências é fundamental para compreender os problemas que os ambientes virtuais resolvem.

O que é o pip?

O `pip` é o gerenciador de pacotes padrão do Python. Ele permite que você instale, desinstale, atualize e liste pacotes (bibliotecas) Python. Quando você instala o Python, o `pip` é instalado junto.

Instalando Pacotes

Para instalar um pacote, use o comando `pip install <nome_do_pacote>`. Mas por que instalar um pacote? Vamos usar um exemplo prático.

Motivação:

Imagine que você queira criar um programa que busca informações sobre os livros de Harry Potter. Existe uma API pública chamada Potter API (https://potterapi-fedeperin.vercel.app/pt/books) que fornece esses dados. Para interagir com essa API, precisamos fazer requisições HTTP. Embora seja possível fazer isso usando bibliotecas nativas do Python, o processo pode ser trabalhoso. É aí que entra a biblioteca `requests`!

A biblioteca `requests` simplifica enormemente o processo de fazer requisições HTTP. Mas não é instalada junto com o Python.

Ao contrário, a biblioteca `requests` e muitas outras foram desenvolvidas pela comunidade (desenvolverdores independentes) e publicadas no PyPI (Python Package Index), que é um repositório público de pacotes Python onde qualquer pessoa pode publicar pacotes: https://pypi.org/

O pip é um programa instalado junto com o python, que permite baixar bibliotecas do PyPI.

Vamos instalar a biblioteca `requests` usando o `pip` e ver como ela funciona.

Instalando o `requests`:

Abra seu terminal e digite o seguinte comando:

```
pip install requests
```

O `pip` baixará o pacote `requests` do PyPI (Python Package Index), o repositório oficial de pacotes Python, e o instalará na máquina.

Gerenciamento Manual de Dependências com pip

Exemplo de uso do `requests`:

Agora, vamos criar um arquivo Python chamado `potter.py` e adicionar o seguinte código:

```
import requests

response = requests.get("https://potterapi-fedeperin.vercel.app/pt/books")
books = response.json()

for book in books:
    print(book["title"])
```

Este código faz uma requisição GET para a API da Potter API, obtém a lista de livros em formato JSON e imprime o título de cada livro.

Execute o código:

```
python potter.py
```

Você verá uma lista de títulos de livros de Harry Potter impressa no console:

```
Harry Potter e a Pedra Filosofal
Harry Potter e a Câmara Secreta
Harry Potter e o Prisioneiro de Azkaban
Harry Potter e o Cálice de Fogo
Harry Potter e a Ordem da Fênix
Harry Potter e o Enigma do Príncipe
Harry Potter e as Relíquias da Morte
Harry Potter e a Criança Amaldiçoada
```

Onde os Pacotes são Instalados?

Quando você usa o `pip` para instalar um pacote, como fizemos com o `requests`, ele é instalado no diretório de instalação do Python que está sendo usado no momento. Mais especificamente, os pacotes são instalados em um subdiretório chamado `Lib\site-packages` dentro da instalação do Python.

Importante: O `pip` está Vinculado à Instalação do Python

É importante entender que o `pip` não é um programa independente e isolado. Ele está vinculado à instalação do Python que está ativa no seu terminal no momento em que você executa o comando `pip`.

Em outras palavras:

- **Cada instalação do Python tem seu próprio `pip`:** Se você tem múltiplas versões do Python instaladas (por exemplo, Python 3.8, Python 3.9, Python 3.13, etc.), cada uma dessas instalações terá seu próprio `pip` dentro de sua pasta de instalação.
- **O `pip` instala pacotes na instalação do Python à qual ele pertence:** Quando você executa `pip install <pacote>`, o `pip` que é executado é aquele que pertence à instalação do Python que está ativa no seu terminal (ou seja, a primeira que é encontrada no seu `PATH`). Esse `pip` instalará o pacote no diretório `site-packages` **dessa mesma instalação do Python**.

Gerenciamento Manual de Dependências com pip

Exemplo:

Se você estiver usando o Python 3.9 no Windows, e essa versão do Python está instalada em `C:\Users\vitor\AppData\Local\Programs\Python\Python39\` , então:

- O `pip` correspondente a essa instalação estará em `C:\Users\vitor\AppData\Local\Programs\Python\Python39\Scripts\pip.exe`
- Quando você executar `pip install requests`, será esse `pip.exe` que será executado.
- A biblioteca `requests` será instalada na pasta `C:\Users\vitor\AppData\Local\Programs\Python\Python39\Lib\site-packages`.

Consequências:

- Se você alternar entre diferentes versões do Python usando ferramentas como o `pyenv`, você estará alternando qual `pip` será executado.

Mostrando o local de instalação

Para ver exatamente onde o pacote `requests` foi instalado, digite no terminal:

```
pip show requests # pip show <nome do pacote>
```

Isso vai mostrar algo assim:

```
Name: requests
Version: 2.32.3
Summary: Python HTTP for Humans.
Home-page: https://requests.readthedocs.io
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache-2.0
Location: C:\Users\vitor\AppData\Local\Programs\Python\Python39\Lib\site-packages
Requires: certifi, charset-normalizer, idna, urllib3
Required-by:
```

Observe a linha "Location", ela mostra a pasta onde o pacote foi instalado. Repare que realmente é dentro da pasta de instalação do Python como havíamos falado.

Importância da Instalação Global:

Como os pacotes são instalados globalmente (ou seja, na pasta de instalação do Python), eles podem ser importados e usados em **qualquer** script ou projeto Python, **desde que você esteja usando a mesma versão do Python na qual os pacotes foram instalados**. Isso é conveniente, pois você não precisa instalar o mesmo pacote várias vezes para diferentes projetos. No entanto, essa conveniência tem um custo, que discutiremos mais adiante quando falarmos sobre ambientes virtuais.

Gerenciamento Manual de Dependências com pip

Versionamento Semântico

Ao instalar pacotes, você frequentemente verá números de versão como `2.28.1`, `1.0.0`, `3.10.2`, etc. Esses números seguem um padrão chamado **Versionamento Semântico (SemVer)**.

O SemVer usa um formato de três partes: `MAJOR.MINOR.PATCH` (MAIOR.MENOR.CORREÇÃO).

- **MAJOR (MAIOR):** Alterações incompatíveis com versões anteriores. Quando a versão principal é incrementada, significa que a nova versão do pacote pode quebrar o código que funcionava com versões anteriores.
- **MINOR (MENOR):** Adição de funcionalidades de forma compatível com versões anteriores. Novas funcionalidades foram adicionadas, mas o código existente ainda deve funcionar sem modificações.
- **PATCH (CORREÇÃO):** Correções de bugs compatíveis com versões anteriores.

Exemplo:

- `requests 2.28.1`: Versão principal 2, versão secundária 28, patch 1.
- Se uma nova versão `requests 2.29.0` for lançada, ela provavelmente terá novas funcionalidades, mas não quebrará o código que usa a versão `2.28.1`.
- Se uma nova versão `requests 3.0.0` for lançada, ela pode conter mudanças que quebram a compatibilidade com a versão `2.x`.

Especificando Versões

Você pode especificar a versão exata de um pacote que deseja instalar usando o operador `==`.

Exemplo:

```
pip install requests==2.28.1
```

Isso instalará a versão `2.28.1` da biblioteca `requests`.

Gerenciamento Manual de Dependências com pip

Requisitos Flexíveis

Em vez de especificar uma versão exata, você pode usar operadores de comparação para especificar um intervalo de versões aceitáveis.

- `>=`: Maior ou igual a.
- `<=`: Menor ou igual a.
- `>`: Maior que.
- `<`: Menor que.
- `!=`: Diferente de.
- `~=`: Compatível com.

Exemplos:

- `pip install requests>=2.20.0`: Instala a versão `2.20.0` ou qualquer versão mais recente da série `2.x`.
- `pip install requests>=2.20.0,<3.0.0`: Instala a versão mais recente, mas apenas dentro do intervalo principal `2.x`.
- `pip install "requests~=2.28.0"`: Instala a versão 2.28.x mais recente, onde x é qualquer valor. Por exemplo, 2.28.0, 2.28.1, 2.28.2, etc. Mas não 2.29.0.

Gerenciamento Manual de Dependências com pip

Listando Pacotes Instalados

Para ver uma lista de todos os pacotes instalados e suas versões, use o comando `pip list` ou `pip freeze`.

- `pip list`**:** Lista todos os pacotes instalados em um formato de tabela.

  Exemplo de saída do `pip list`:

```
Package            Version
------------------ ----------
certifi            2024.12.14
charset-normalizer 3.4.1
idna               3.10
pip                24.3.1
requests           2.32.3
urllib3            2.3.0
```

- `pip freeze`**:** Lista os pacotes instalados em um formato que pode ser usado para recriar o ambiente em outro lugar.

  Exemplo de saída do `pip freeze`:

```
certifi==2024.12.14
charset-normalizer==3.4.1
idna==3.10
requests==2.32.3
urllib3==2.3.0
```

Você pode estar se perguntando: *"Se eu só instalei o `requests`, por que foram listados 5 pacotes ?"*

A resposta é que o `requests`, por sua vez, depende de outras bibliotecas para funcionar corretamente. Essas bibliotecas são chamadas de **dependências indiretas** ou **transitivas**.

Quando você instalou o `requests` com o `pip`, o `pip` automaticamente instalou também as suas dependências. No caso do `requests`, ele depende de:

- `certifi`**:** Fornece certificados CA (Certificate Authority) para verificar a autenticidade de conexões SSL.
- `charset-normalizer`**:** Ajuda a detectar a codificação de caracteres de um texto.
- `idna`**:** Suporta a conversão de nomes de domínio internacionalizados (IDNs).
- `urllib3`**:** Uma biblioteca poderosa para fazer requisições HTTP, que serve como base para o `requests`.

O `pip freeze` lista não apenas as bibliotecas que você instalou explicitamente (como o `requests`), mas também todas as suas dependências indiretas, ou seja, tudo que efetivamente está instalado na pasta `site-packages`.

Atenção:

Se você já instalou vários pacotes, essas listas podem ser bem grandes.

Gerenciamento Manual de Dependências com pip

Desinstalando Pacotes

Para desinstalar um pacote, use o comando `pip uninstall <nome_do_pacote>`.

Exemplo:

```
pip uninstall requests
```

O `pip` removerá o pacote do diretório `site-packages`.

Atenção:

Isso não vai desinstalar as dependências transitivas do `requests`, ou seja, `certifi`, `charset-normalizer`, `idna` e `urllib3` continuarão instalados (você pode comprovar com `pip freeze`, que ainda mostrará essas outras bibliotecas).

Arquivo requirements.txt

Imagine que você tenha criado um projeto Python que usa várias bibliotecas externas, como o `requests` que instalamos anteriormente.

Você quer compartilhar seu projeto com outras pessoas, mas elas precisam instalar as mesmas bibliotecas para que o código funcione corretamente.

É aqui que entra o arquivo `requirements.txt`.

O `requirements.txt` é um arquivo de texto simples que lista todas as dependências (e suas versões) do seu projeto. Ele serve como uma receita para que outras pessoas possam instalar facilmente os pacotes necessários.

Criando um arquivo `requirements.txt`:

A maneira mais fácil de criar um arquivo `requirements.txt` é usar o comando `pip freeze > requirements.txt`.

Exemplo:

Depois de instalar o `requests`, execute:

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

Isso criará um arquivo chamado `requirements.txt` no diretório atual, com o seguinte conteúdo (ou algo semelhante, dependendo das versões instaladas):

```
certifi==2024.12.14
charset-normalizer==3.4.1
idna==3.10
requests==2.32.3
urllib3==2.3.0
```

Gerenciamento Manual de Dependências com pip

Observação Importante:

É crucial entender que o comando `pip freeze` **não tem a noção de "projeto"**. Ele simplesmente lista **todos** os pacotes instalados no ambiente Python atual (ou seja, na pasta de instalação do Python que está sendo usado), independentemente de quais pacotes foram realmente usados no seu projeto específico.

Isso significa que, se você tiver outros pacotes instalados globalmente que não são usados pelo seu projeto, eles também aparecerão no arquivo `requirements.txt`. Isso pode tornar o arquivo inchado e potencialmente causar problemas de compatibilidade se alguém tentar instalar todas as dependências listadas, mesmo que não sejam necessárias para o seu projeto.

Por exemplo, se você tiver o `numpy` instalado globalmente, mas não o estiver usando em seu projeto atual, ele ainda será incluído no `requirements.txt` gerado pelo `pip freeze`.

Esse é um dos principais motivos pelos quais os ambientes virtuais são importantes, como veremos na próxima seção. Eles permitem que você isole as dependências de cada projeto, evitando que pacotes desnecessários sejam incluídos no `requirements.txt`.

Para criar o `requirements.txt` sem esse problema de inchaço, pode ser melhor simplesmente escrever manualmente o arquivo, listando apenas as dependências do seu projeto específico.

Compartilhando seu projeto:

Agora, quando você compartilhar seu projeto com outras pessoas, basta incluir o arquivo `requirements.txt`. Outros desenvolvedores podem então instalar todas as dependências necessárias com um único comando.

Instalando a Partir de um Arquivo requirements.txt

Para instalar todas as dependências listadas em um arquivo `requirements.txt`, use o comando `pip install -r requirements.txt`.

Exemplo:

Se você recebeu um projeto com um arquivo `requirements.txt`, basta navegar até o diretório do projeto no terminal e executar:

```
pip install -r requirements.txt
```

Isso instalará as versões exatas de todos os pacotes listados no arquivo `requirements.txt` na sua instalação do Python, garantindo que você tenha o mesmo ambiente de desenvolvimento que a pessoa que criou o projeto.

Gerenciamento Manual de Dependências com pip

Limitações do Gerenciamento Manual

Embora o `pip` seja útil para instalar e desinstalar pacotes, o gerenciamento manual de dependências tem limitações significativas:

- **Conflitos de Dependências:** Diferentes projetos podem precisar de versões diferentes do mesmo pacote. Como os pacotes são instalados globalmente, isso pode levar a conflitos. Por exemplo, o Projeto A pode precisar da versão `1.0` de uma biblioteca, enquanto o Projeto B precisa da versão `2.0`. Se a versão `2.0` for instalada globalmente, o Projeto A poderá parar de funcionar.

- **Ambientes Não Isolados:** Todos os projetos compartilham o mesmo conjunto de pacotes instalados, o que pode causar problemas de compatibilidade. Alterar a versão de um pacote para um projeto pode afetar inadvertidamente outros projetos.

- **Dificuldade em Replicar Ambientes:** Recriar o ambiente de desenvolvimento em outra máquina pode ser difícil e propenso a erros se as versões exatas das dependências não forem cuidadosamente documentadas e gerenciadas através do `requirements.txt`.

Motivação para Ambientes Virtuais:

Essas limitações do gerenciamento manual de dependências nos levam à necessidade de **ambientes virtuais**, que serão abordados na próxima seção. Ambientes virtuais fornecem uma maneira de isolar as dependências de cada projeto, evitando conflitos e facilitando a reprodução de ambientes de desenvolvimento.

Ambientes Virtuais: Isolando Dependências de Projetos

Agora que sabemos como gerenciar diferentes versões do Python e como gerenciar dependências manualmente com o `pip`, vamos dar o próximo passo: isolar as dependências de cada um dos nossos projetos.

Motivação: Por que Precisamos de Ambientes Virtuais?

Imagine que você está trabalhando em dois projetos:

- **Projeto A:** Um web app usando o framework Flask na versão 1.0.
- **Projeto B:** Um script de análise de dados que usa uma versão mais antiga do Flask, a 0.12, porque depende de uma biblioteca específica que ainda não é compatível com versões mais recentes do Flask.

Problema: Se você instalar o Flask globalmente (ou seja, para todos os projetos), você só pode ter uma versão instalada por vez. Se você instalar o Flask 1.0 para o Projeto A, o Projeto B não funcionará, pois ele precisa do Flask 0.12. Se você instalar o Flask 0.12 para o Projeto B, o Projeto A poderá não funcionar corretamente, pois ele foi desenvolvido para o Flask 1.0.

Consequência: Você fica preso em um ciclo de instalação e desinstalação de dependências, o que é extremamente ineficiente e pode levar a erros difíceis de rastrear.

Solução: Precisamos de uma maneira de criar **ambientes isolados** para cada projeto, onde possamos instalar as dependências específicas de cada um, sem que elas interfiram umas nas outras. É aí que entram os **ambientes virtuais**.

O que são Ambientes Virtuais?

Um ambiente virtual é um diretório isolado (uma pasta) que contém uma instalação do Python e suas bibliotecas. Ele permite que você instale pacotes específicos para um projeto sem afetar outros projetos ou a instalação global do Python.

Como os Ambientes Virtuais Funcionam?

Quando você cria um ambiente virtual, ele cria um diretório que contém:

- Uma cópia (ou atalho) do executável do Python.
- Uma cópia do gerenciador de pacotes `pip`.
- Um diretório `site-packages` vazio, onde as bibliotecas do projeto serão instaladas.

Quando você ativa o ambiente virtual, ele modifica a variável de ambiente `PATH` para que o sistema operacional procure primeiro no diretório do ambiente virtual quando você executar comandos Python. Isso garante que você esteja usando a versão do Python e as bibliotecas instaladas dentro do ambiente virtual, e não a instalação global.

Analogia:

Pense em um ambiente virtual como uma sala de trabalho separada para cada um dos seus projetos. Cada sala tem sua própria mesa, suas próprias ferramentas e seus próprios materiais. Quando você entra em uma sala, você só usa o que está lá dentro. Você não precisa se preocupar com o que está nas outras salas ou no corredor (instalação global).

Ambientes Virtuais: Isolando Dependências de Projetos

Ferramentas para Criar Ambientes Virtuais: venv e virtualenv

Existem duas principais ferramentas para criar ambientes virtuais em Python:

- `venv`**:** Um módulo que vem embutido na biblioteca padrão do Python 3.3 em diante. É a forma recomendada para criar ambientes virtuais.
- `virtualenv`**:** Uma ferramenta externa mais antiga e com mais recursos que o `venv`. É útil para versões mais antigas do Python ou se você precisar de recursos avançados não disponíveis no `venv`.

Nesta aula, vamos focar no `venv`, pois ele é a solução padrão e suficiente para a maioria dos casos.

Exemplo Prático com venv

Vamos criar um ambiente virtual para o Projeto A, que usa o Flask 1.0:

1 — Navegue até o diretório do seu projeto:

```
cd /caminho/para/projeto_a # pasta do seu projeto
```

2 — Crie o ambiente virtual:

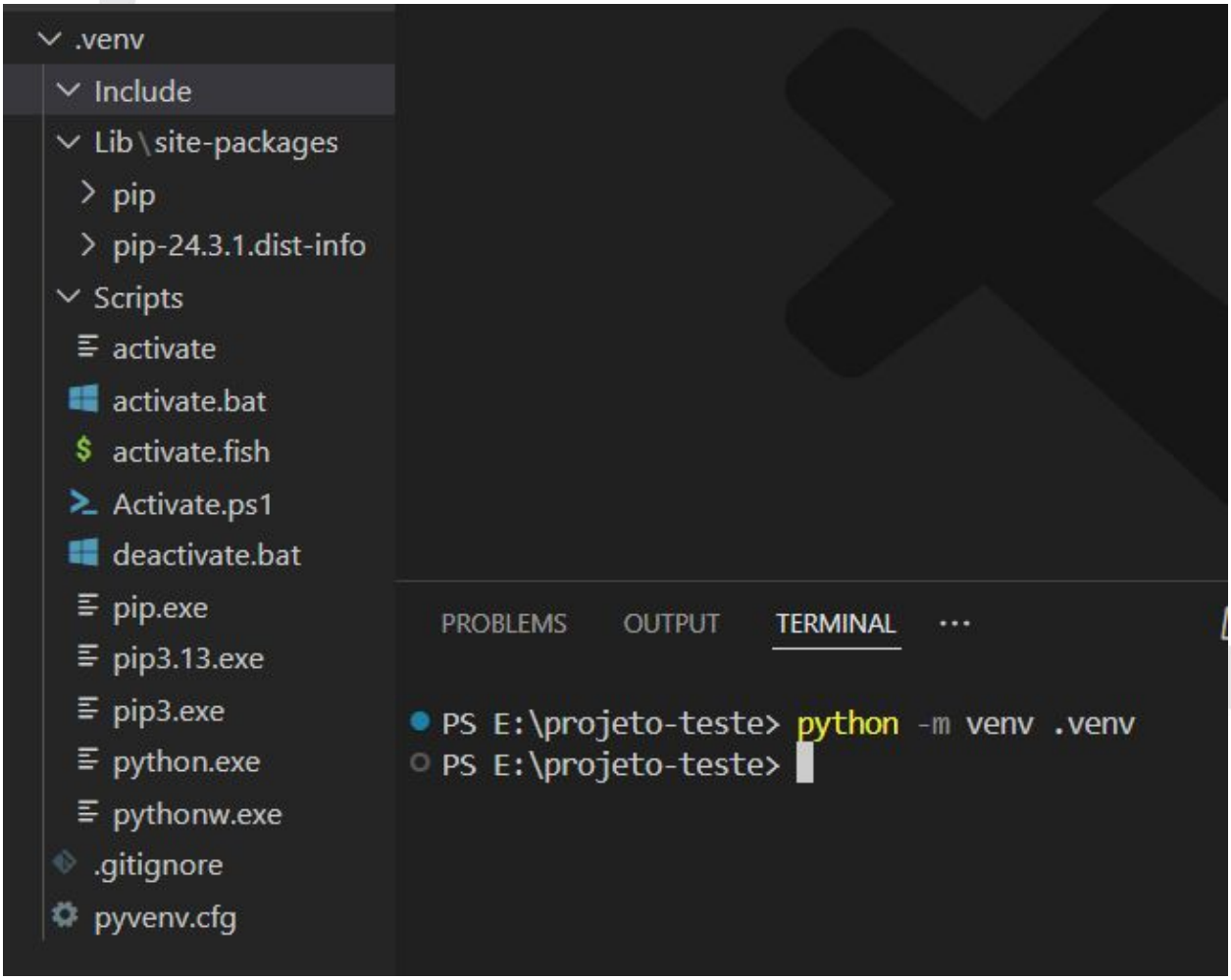
Obs: se você está no Ubuntu, talvez precise instalar o pacote venv digitando num terminal: `sudo apt install python3-venv`. Sem isso, é possível que o comando abaixo falhe.

Para criar um ambiente virtual, execute:

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

Isso criará um diretório chamado `.venv` dentro do diretório do seu projeto. O nome `.venv` é uma convenção comum, mas você pode usar qualquer nome que desejar.

Ambientes Virtuais: Isolando Dependências de Projetos



Repare que a pasta do ambiente virtual (`.venv`) tem seu próprio executável do python (python.exe), seu próprio `pip.exe` e seu próprio diretório `site-packages` para instalar pacotes, além de alguns scripts utilitários (activate, deactivate, etc.).

Quando você executa `python -m venv .venv`, o comando `venv` usa o interpretador Python que está atualmente ativo no seu terminal (aquele que é encontrado primeiro no seu `PATH`) para criar o ambiente virtual. Isso significa que:

- O Python do ambiente virtual é uma cópia (ou um link simbólico/atalho, dependendo do sistema operacional e das configurações) do Python que você usou para executar o comando `venv`.

- Se você quiser um ambiente virtual com uma versão específica do Python, você precisa primeiro ativar essa versão no seu terminal antes de criar o ambiente virtual.

Usando o PyEnv para Controlar a Versão do Python do Ambiente Virtual:

Lembra do **PyEnv**, que aprendemos anteriormente? Ele é uma ferramenta excelente para gerenciar múltiplas versões do Python. Você pode usar o PyEnv para alternar facilmente entre as versões do Python no seu terminal e, assim, controlar qual versão do Python será usada para criar o ambiente virtual.

Ambientes Virtuais: Isolando Dependências de Projetos

Exemplo com PyEnv:

1. Instale o Python 3.9 e 3.10 com o PyEnv:

```
pyenv install 3.9.7
pyenv install 3.10.12
```

2. Alterne para o Python 3.9 globalmente:

```
pyenv global 3.9.7
```

3. Crie um ambiente virtual:

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

Nesse caso, o ambiente virtual usará o Python 3.9.

4. Agora, alterne para o Python 3.10 globalmente:

```
pyenv global 3.10.12
```

5. Crie outro ambiente virtual (em outro diretório, para não sobrescrever o anterior):

```
python -m venv .venv310
```

Esse novo ambiente virtual usará o Python 3.10.

Em resumo:

O ambiente virtual herda a versão do Python que está ativa no seu terminal no momento da criação do ambiente.

Você pode controlar qual versão do Python será instalada no ambiente virtual alternando a versão do Python no seu terminal (por exemplo, usando o PyEnv) antes de executar o comando `venv`. Isso permite que você crie ambientes virtuais com diferentes versões do Python, conforme necessário para seus projetos.

Ambientes Virtuais: Isolando Dependências de Projetos

3 — Ative o ambiente virtual:

No Linux/macOS:

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

No Windows CMD:

```
.venv\\Scripts\\activate.bat
```

No Windows PowerShell:

```
.venv\\Scripts\\Activate.ps1
```

No Windows você pode distinguir se o terminal é CMD ou Powershell porque no Powershell aparece "PS" no início da linha do terminal.

Após ativar o ambiente virtual, você verá o nome dele entre parênteses no início da linha de comando, indicando que o ambiente virtual está ativo:

Ambientes Virtuais: Isolando Dependências de Projetos

O que realmente significa "ativar" um ambiente virtual?

Quando você executa o comando de ativação (por exemplo, `source .venv/bin/activate`, `.venv\Scripts\activate.bat` ou `.venv\Scripts\Activate.ps1`), você está, na verdade, executando um **script** (um pequeno programa) que está localizado dentro da pasta do ambiente virtual.

Esse script de ativação tem uma tarefa principal: modificar temporariamente a variável de ambiente `PATH` **dentro do seu terminal** para que ela aponte para os executáveis (como `python` e `pip`) que estão dentro do ambiente virtual, e não para os que estão fora dele (na instalação global do Python).

Lembrando o que é o PATH:

Como vimos anteriormente, o `PATH` é uma lista de diretórios que o sistema operacional usa para procurar programas executáveis quando você digita um comando no terminal.

Como o script de ativação modifica o PATH:

O script de ativação **adiciona o diretório `bin` (no Linux/macOS) ou `Scripts` (no Windows) do ambiente virtual ao *início* do** `PATH`. Isso significa que, quando você digitar `python`, `pip` ou qualquer outro comando relacionado ao Python, o sistema operacional encontrará primeiro os executáveis dentro do ambiente virtual.

Comprovando a mudança:

Você pode verificar qual instalação do Python está sendo usada pelo seu terminal usando o comando:

```
python -c "import sys; print(sys.executable)"
```

Antes de ativar o ambiente virtual, esse comando mostrará o caminho para o executável do Python global. Depois de ativar o ambiente virtual, ele mostrará o caminho para o executável do Python dentro do ambiente virtual.

Veja abaixo uma comparação entre antes e depois de ativar:

Antes de ativar, o terminal encontrava o executável "python.exe" em:

- `C:\Users\vitor\.pyenv\pyenv-win\versions\3.13.1\python.exe` (python global instalado com pyenv)

Depois de ativar, o terminal passou a usar o "python.exe" em:

- `E:\projeto-teste\.venv\Scripts\python.exe` (instalação do python dentro da pasta `.venv` do ambiente virtual)

Ambientes Virtuais: Isolando Dependências de Projetos

Atenção:

Quando você ativa o ambiente virtual, essa ativação só vale **nesse mesmo terminal**. Se você abrir outro terminal, o ambiente virtual não estará ativo. Para ativar o ambiente virtual em outro terminal, você precisará executar o script de ativação novamente.

Em resumo:

Ativar um ambiente virtual significa executar um script que modifica temporariamente o `PATH` do seu terminal para que os comandos Python sejam direcionados para os executáveis dentro do ambiente virtual.

Isso garante que você esteja usando a versão correta do Python e os pacotes instalados nesse ambiente específico, isolando seu projeto de outros projetos e da instalação global do Python.

A ativação vale somente dentro do terminal em que você está ativando, ao abrir outros terminais precisará ativar de novo.

Você pode garantir que o ambiente está ativado executando `python -c "import sys; print(sys.executable)"` e verificando se o caminho aponta para o executável dentro do ambiente virtual.

4 — Instale as dependências do projeto:

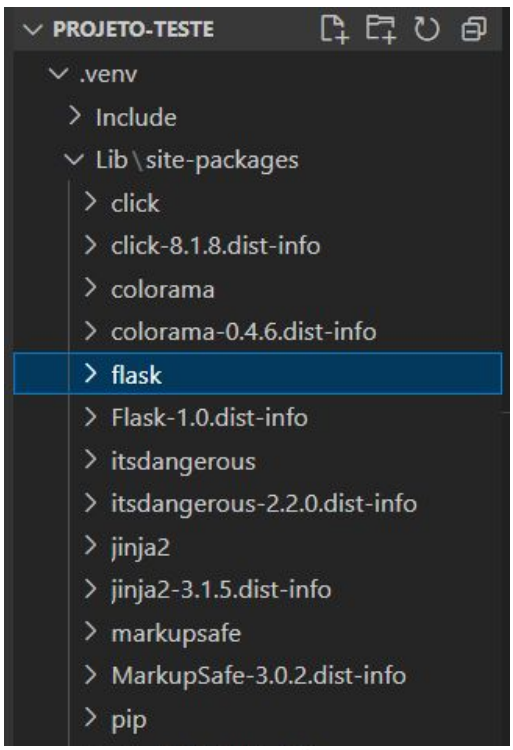
Imagine que seu projeto precisa de uma dependência, vamos exemplificar com o pacote `flask`.

Se você quer instalar o flask no seu ambiente virtual, execute num terminal **com o ambiente virtual ativado**:

```
pip install flask==1.0
```

Isso instalará o Flask 1.0 dentro do ambiente virtual, no diretório `site-packages`.

Você pode constatar que o `flask` está nessa pasta:



Ambientes Virtuais: Isolando Dependências de Projetos

Ou então pelo terminal com `pip show flask` (repare no "Location"):

```
(.venv) PS E:\projeto-teste> pip show flask
Name: Flask
Version: 1.0
Summary: A simple framework for building complex web applications.
Home-page: https://www.palletsprojects.com/p/flask/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD
Location: E:\projeto-teste\.venv\Lib\site-packages
Requires: click, itsdangerous, Jinja2, Werkzeug
Required-by:
```

5 — Verifique os pacotes instalados:

```
pip freeze
# Saída:
# Flask==1.0
# ... outras dependências do Flask ...
```

O comando `pip freeze` lista todos os pacotes instalados no ambiente virtual.

E aqui está uma das grandes vantagens de usar ambientes virtuais: como os pacotes são instalados especificamente neste ambiente, isolados da instalação global do Python e de outros projetos, o `pip freeze` agora mostrará **somente** os pacotes que foram instalados para este projeto específico.

Lembra do problema que mencionamos anteriormente, onde o `pip freeze` listava todos os pacotes instalados globalmente, mesmo aqueles que não eram usados pelo projeto? Com ambientes virtuais, esse problema é resolvido!

Agora, o arquivo `requirements.txt` gerado pelo `pip freeze` será muito mais limpo e preciso, contendo apenas as dependências reais do seu projeto. Isso torna muito mais fácil para outras pessoas (ou para você mesmo no futuro) replicar o ambiente e instalar as dependências corretas.

Exemplo:

Se você estivesse gerenciando as dependências manualmente, sem um ambiente virtual, e tivesse diversos pacotes instalados globalmente (como `numpy`, `pandas`, `requests`, etc.), o `pip freeze` listaria todos eles, mesmo que seu projeto usasse apenas o `Flask`.

Com um ambiente virtual ativado, o `pip freeze` mostrará uma lista muito mais enxuta, como no exemplo acima, contendo apenas o `Flask` e suas dependências diretas. Isso ocorre porque o ambiente virtual é isolado e não tem visibilidade dos pacotes instalados globalmente ou em outros ambientes virtuais.

Essa é uma das principais razões pelas quais os ambientes virtuais são essenciais para o desenvolvimento em Python: eles permitem que você gerencie as dependências de cada projeto de forma isolada e precisa, evitando conflitos e facilitando a reprodução do ambiente.

Ambientes Virtuais: Isolando Dependências de Projetos

6 — Desative o ambiente virtual:

```
deactivate
```

Isso desativará o ambiente virtual e retornará à instalação global do Python.

Sempre verifique que realmente foi desativado, ou seja, que o terminal está usando o Python global novamente.

Veja abaixo uma comparação primeiro com o ambiente ativado e depois desativado:

Com o ambiente ativado, o terminal encontrava o executável "python.exe" em:

- `E:\projeto-teste\.venv\Scripts\python.exe` (instalação do python dentro da pasta `.venv` do ambiente virtual)

Depois de desativar, o terminal passou a usar o "python.exe" em:

- `C:\Users\vitor\.pyenv\pyenv-win\versions\3.13.1\python.exe` (uma instalação global do python)

Se o `deactivate` não funcionar (terminal continua usando o "python.exe" da pasta `.venv`), simplesmente feche o terminal e abra outro.

Ambientes Virtuais: Isolando Dependências de Projetos

Onde Colocar os Arquivos do Projeto e Como Lidar com o Ambiente Virtual no Git

Agora que entendemos como criar e usar ambientes virtuais, surge uma pergunta importante: **onde devemos colocar os arquivos do nosso projeto (os arquivos `.py` que estamos codificando) em relação ao diretório do ambiente virtual (`.venv`)?**

A regra é simples: os arquivos do seu projeto devem ficar *fora* do diretório do ambiente virtual.

O diretório do ambiente virtual (`.venv` no nosso exemplo) deve conter apenas os arquivos relacionados ao próprio ambiente virtual, ou seja, o interpretador Python, as bibliotecas instaladas e os scripts de ativação. Ele não deve conter os arquivos de código-fonte do seu projeto.

Estrutura Recomendada:

```
meu_projeto/
├── .venv/              <-- Ambiente virtual (NÃO deve ser enviado ao Git)
├── .gitignore          <-- Arquivo para ignorar o .venv no Git
├── main.py             <-- Arquivo principal do seu projeto
├── utils.py etc.       <-- Outros arquivos/pastas do seu projeto
└── requirements.txt    <-- Lista de dependências (DEVE ser enviado ao Git)
```

Por que manter os arquivos do projeto fora do ambiente virtual?

1. **Separação de Conceitos:**
   O ambiente virtual é uma ferramenta para gerenciar dependências, enquanto os arquivos `.py` são o código-fonte do seu projeto. Mantê-los separados mantém uma estrutura de diretórios organizada e clara.

2. **Portabilidade:**
   Você deve poder mover seu projeto para outro local ou compartilhá-lo com outras pessoas sem carregar junto o ambiente virtual. O ambiente virtual pode ser recriado facilmente usando o arquivo `requirements.txt`.

3. **Controle de Versão (Git):**
   O ambiente virtual não deve ser incluído no controle de versão (como o Git). Ele contém muitos arquivos que são específicos da sua máquina e do seu ambiente de desenvolvimento. Incluí-lo no repositório Git aumentaria desnecessariamente o tamanho do repositório e poderia causar conflitos com outros desenvolvedores.

Ambientes Virtuais: Isolando Dependências de Projetos

Como lidar com o ambiente virtual no Git?

1. **Não envie o ambiente virtual para o Git:**

   Você deve adicionar o diretório do ambiente virtual ao arquivo `.gitignore` do seu projeto. O `.gitignore` é um arquivo que especifica quais arquivos e diretórios o Git deve ignorar e não incluir no controle de versão.

   Exemplo de `.gitignore`:

   ```
   .venv/
   ```

   Isso dirá ao Git para ignorar completamente o diretório `.venv` e seu conteúdo.

2. **Envie o `requirements.txt` para o Git:**

   O arquivo `requirements.txt`, que contém a lista de dependências do seu projeto, *deve* ser incluído no controle de versão. Isso permitirá que outras pessoas (ou você mesmo em outra máquina) recriem facilmente o ambiente virtual com as dependências corretas usando o comando `pip install -r requirements.txt` (depois de criar um ambiente virtual e ativá-lo).

Em resumo:

- Mantenha os arquivos do seu projeto (arquivos `.py`) fora do diretório do ambiente virtual.
- Use o arquivo `.gitignore` para evitar que o diretório do ambiente virtual seja enviado para o Git.
- Inclua o arquivo `requirements.txt` no controle de versão para permitir a reprodução fácil do ambiente virtual.

Seguindo essas práticas, você manterá seu projeto organizado, portável e fácil de colaborar.

Resumo de Ambientes Virtuais

Ambientes virtuais são essenciais para isolar as dependências de seus projetos Python.

O ambiente virtual é uma pasta contendo uma instalação do python independente, com suas próprias bibliotecas e executáveis.

Eles permitem que você instale diferentes versões de bibliotecas para cada projeto sem conflitos.

O `venv` é a ferramenta padrão para criar ambientes virtuais no Python 3.3+, enquanto o `virtualenv` é uma alternativa para versões mais antigas ou para usuários que precisam de recursos avançados.

Com ambientes virtuais, você pode manter seus projetos organizados, evitar conflitos de dependências e garantir que seu código funcione corretamente em diferentes ambientes.

Integração de Ambiente Virtual no VSCode

No capítulo anterior, aprendemos a criar e ativar ambientes virtuais na linha de comando.

No entanto, o VSCode possui sua própria maneira de lidar com interpretadores Python e, por padrão, ele não usa automaticamente o ambiente virtual que você ativou no terminal. Isso pode levar a inconsistências entre o código que você executa no terminal e o que o VSCode entende.

Problema: VSCode Não Detecta Automaticamente o Ambiente Virtual Ativado no Terminal

Vamos supor que você criou um ambiente virtual chamado `.venv` e o ativou no terminal:

```
python -m venv .venv
source .venv/bin/activate       # No Linux/macOS
.venv\Scripts\activate.bat      # No Windows CMD
.venv\Scripts\Activate.ps1      # No Windows PowerShell
```

Em seguida, você instalou o Flask neste ambiente virtual:

```
pip install flask==3.1.0
```

Agora, você cria um arquivo `app.py` com o seguinte código simples do Flask (exemplo ilustrativo, não precisa entender o código):

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
   return "<p>Hello, World!</p>"

if __name__ == "__main__":
   print("Executando servidor web com Flask")
   app.run()
```

Se você executar este código no terminal (com o ambiente virtual ativado), ele funcionará perfeitamente:

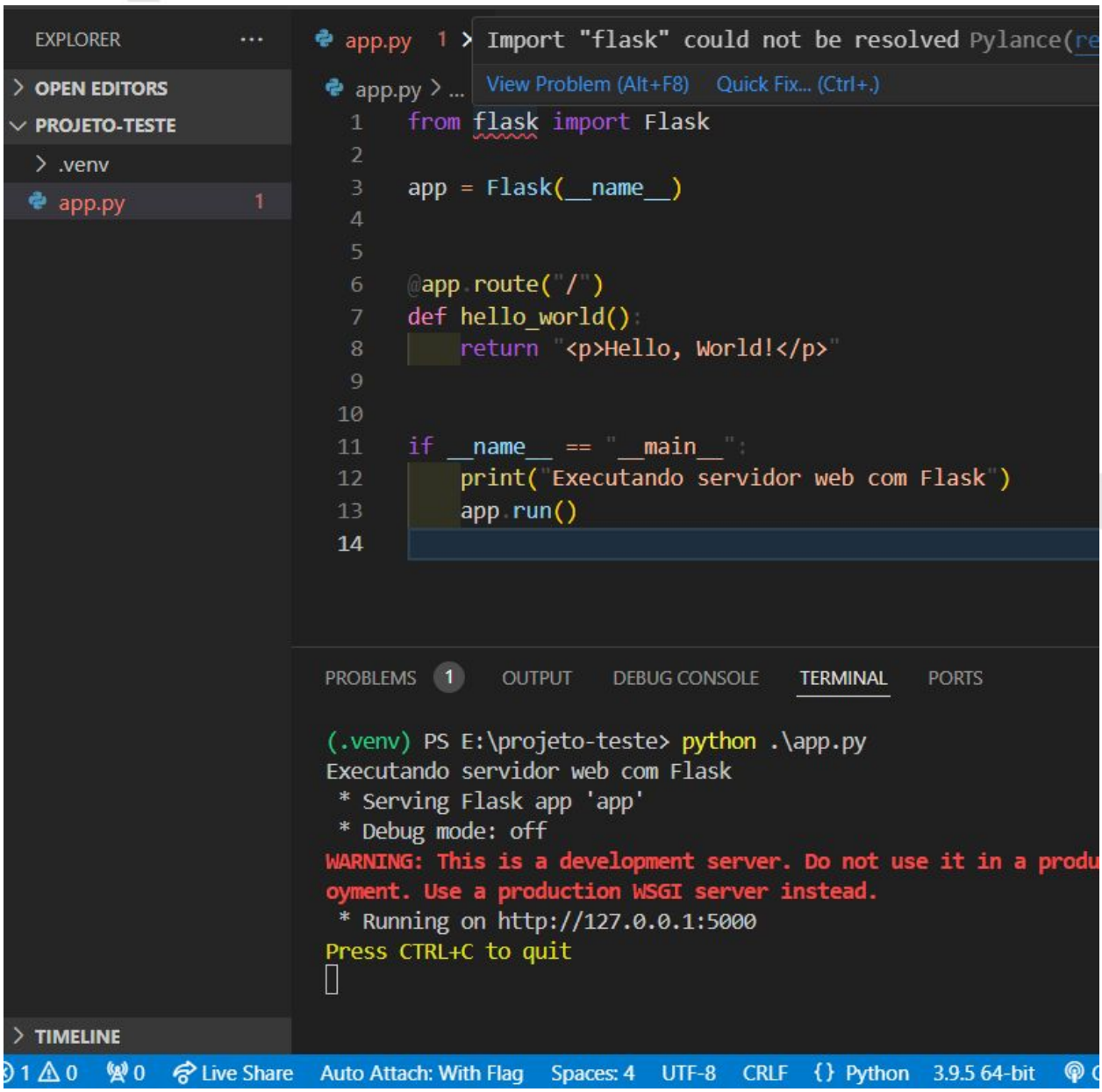
```
python app.py
```

```
(.venv) PS E:\projeto-teste> python .\app.py
Executando servidor web com Flask
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Integração de Ambiente Virtual no VSCode

Não é importante entender o código. Ele é um servidor web com Flask, se você acessar a URL que aparece no terminal (geralmente `http://127.0.0.1:5000/`), você verá a página "Hello, World!", comprovando que o código funciona. Aperte Ctrl-C no terminal para fechar o programa.

Então pelo terminal (com ambiente virtual ativado) o código funciona perfeitamente. No entanto, o VSCode pode mostrar um erro na linha `from flask import Flask`:



Ele diz que não consegue encontrar o pacote flask. Mas isso não pode estar certo, afinal quando você executou o código pelo terminal, o python conseguiu sim encontrar o pacote.

Integração de Ambiente Virtual no VSCode

Por que isso acontece?

Conforme explicado no capítulo "**Trocando a Versão do Python no VSCode**", o VSCode seleciona independentemente qual instalação do Python ele vai usar para alimentar seus recursos de inteligência de código (IntelliSense), como autocompletar, análise de erros e outros.

Nesse caso, o VSCode está usando uma instalação do Python que não é aquela instalada no ambiente virtual.

Consequência:

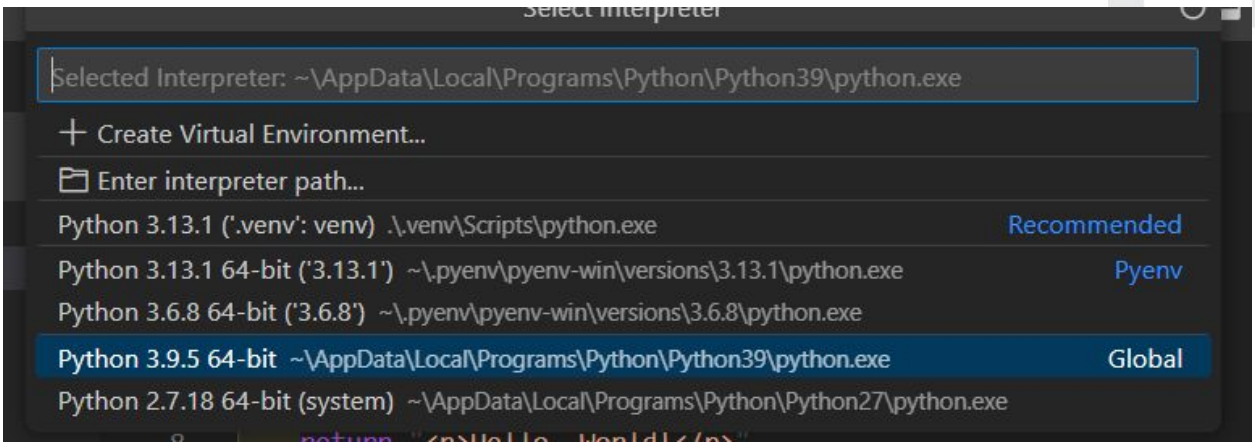
Isso pode causar confusão, pois o VSCode pode mostrar erros ou se comportar de forma inconsistente com o que você espera com base no ambiente virtual ativado no terminal.

Solução: Configurar o VSCode para Usar o Interpretador do Ambiente Virtual

Felizmente, é fácil configurar o VSCode para usar o interpretador Python do seu ambiente virtual.

Clique na barra azul inferior do VSCode onde aparece a instalação do Python que o VSCode está usando (se isso não aparecer para você, aperte F1 e pesquise por "python interpreter", daí escolha a opção "Python: Select Interpreter").

No meu computador aparece isso aqui:



Repare que a instalação do Python que meu VSCode está usando é aquela da pasta `~\AppData\Local\Programs\Python\Python39` (~ é uma abreviação de `C:\Users\Vitor`). Essa é uma instalação global.

Nesse caso eu devo selecionar a instalação do Python do ambiente virtual, que é a pasta `~\.venv\Scripts` (primeira opção na imagem acima).

Ao fazer isso, o VSCode passará a usar o "python.exe" do ambiente virtual e como o flask está instalado nesse ambiente virtual, o VSCode vai conseguir encontrar o flask e não mostrará mais o erro no import.

Integração de Ambiente Virtual no VSCode

Vantagem: VSCode Ativa Automaticamente o Ambiente Virtual em Novos Terminais

Ao contrário do que acontece quando configuramos uma instalação global do Python no VSCode (como vimos no capítulo "**Trocando a Versão do Python no VSCode**"), quando você configura o VSCode para usar o interpretador de um ambiente virtual, ele oferece uma vantagem adicional: **o VSCode ativará automaticamente o ambiente virtual para você em todos os novos terminais que você abrir dentro do VSCode.**

Isso significa que você não precisa mais ativar manualmente o ambiente virtual toda vez que abrir um novo terminal no VSCode. O VSCode cuidará disso para você.

Demonstração:

1. Configure o VSCode para usar o interpretador do ambiente virtual.
2. Feche todos os terminais abertos no VSCode.
3. Abra um novo terminal no VSCode (`Ctrl+J` ou `Terminal > New Terminal`).
4. Observe que o ambiente virtual é ativado automaticamente, isso pode não ser indicado pelo prompt (`.venv`) no início da linha de comando (teoricamente deveria aparecer mas no meu computador não apareceu), mas você pode executar:

```
python -c "import sys; print(sys.executable)"
```

E ele imprimirá o caminho para o executável do Python dentro do seu ambiente virtual, comprovando que o ambiente virtual está ativado:

Comparação com a Configuração de uma Instalação Global:

No capítulo "**Trocando a Versão do Python no VSCode**", vimos que, ao configurar o VSCode para usar uma instalação global do Python, ele **não** modificava o comportamento do terminal. Você ainda precisava trocar a instalação do Python no terminal usando `pyenv global <versão>`.

No entanto, ao configurar o VSCode para usar o interpretador de um ambiente virtual, ele **automaticamente** ativa o ambiente virtual em novos terminais.

Integração de Ambiente Virtual no VSCode

Em resumo:

- O VSCode não usa automaticamente o ambiente virtual que você ativou no terminal, a menos que você o configure para isso.
- Se o VSCode não está usando o ambiente virtual que você ativou no terminal, a análise de código do VSCode pode estar errada, por exemplo ele pode reclamar que certo pacote não existe porque está procurando o pacote na instalação global do Python, enquanto na verdade o pacote está instalado no ambiente virtual.
- Ao configurar o VSCode para usar o ambiente virtual, ele ativará automaticamente o ambiente em novos terminais que você abrir dentro do VSCode.

Essa integração torna o trabalho com ambientes virtuais no VSCode muito mais conveniente e consistente, garantindo que o terminal e os recursos de inteligência de código do VSCode estejam sempre usando o mesmo ambiente.

Com este capítulo, cobrimos a integração do ambiente virtual no VSCode, explicando como configurá-lo para usar o interpretador do ambiente virtual e destacando a vantagem da ativação automática do ambiente em novos terminais.

Também comparamos esse comportamento com o que acontece quando se configura uma instalação global do Python no VSCode, reforçando as diferenças.

Gerenciamento Avançado de Dependências com o Poetry

Até agora, aprendemos a gerenciar versões do Python com o PyEnv e a isolar dependências com ambientes virtuais usando `venv` e `pip`. Essa abordagem funciona bem para projetos simples, mas à medida que os projetos crescem em complexidade, o gerenciamento de dependências pode se tornar um desafio. Vamos explorar algumas limitações do `venv` + `pip` e como o Poetry surge como uma solução mais robusta e moderna.

Motivação: Além do venv e pip - Os Desafios do Gerenciamento de Dependências

Embora o `venv` e o `pip` sejam ferramentas essenciais para o desenvolvimento em Python, eles têm algumas limitações que se tornam evidentes em projetos maiores e mais complexos:

- **Resolução de Dependências Ineficiente:**

  O `pip` instala as dependências de forma linear, sem uma estratégia sofisticada para resolver conflitos entre as dependências e suas subdependências. Isso pode levar a situações em que duas bibliotecas precisam de versões incompatíveis da mesma subdependência, causando erros difíceis de depurar.

- **Arquivo `requirements.txt` Impreciso e Potencialmente Inchado:**

  O `pip freeze` gera um arquivo `requirements.txt` que lista todas as dependências instaladas no ambiente virtual, incluindo as dependências indiretas. Isso pode levar a um arquivo inchado com pacotes desnecessários e, pior, pode causar problemas de compatibilidade ao recriar o ambiente em outra máquina, se as versões exatas das dependências indiretas não forem compatíveis. Além disso, o `pip` não diferencia entre dependências de produção e de desenvolvimento.

- **Gerenciamento de Ambientes Virtuais Manual:**

  Embora o `venv` facilite a criação de ambientes virtuais, o processo ainda é manual. Você precisa criar, ativar e desativar os ambientes virtuais manualmente, o que pode ser tedioso e propenso a erros (por exemplo se esquecer de ativar o ambiente e instalar um pacote com `pip install`, não será instalado na pasta do ambiente)

- **Falta de Recursos para Publicação de Pacotes:**

  O `pip` e o `venv` não oferecem recursos integrados para facilitar a publicação de seus próprios pacotes Python no PyPI (Python Package Index).

Gerenciamento Avançado de Dependências com o Poetry

Exemplo de Problema de Resolução de Dependências:

Imagine que você está trabalhando em um projeto que usa duas bibliotecas: `biblioteca-a` e `biblioteca-b`.

- `biblioteca-a` requer `dependencia-comum>=1.0,<4.0`
- `biblioteca-b` requer `dependencia-comum>=3.0,<5.0`

Se você instalar `biblioteca-a` primeiro, o `pip` instalará a versão mais recente de `dependencia-comum` que satisfaça a restrição da `biblioteca-a`, digamos, `dependencia-comum 3.9`.

Em seguida, ao instalar `biblioteca-b`, o `pip` instalará a versão mais recente de `dependencia-comum` que satisfaça a restrição da `biblioteca-b`, digamos, `dependencia-comum 4.9`. Essa instalação **sobrescreve** a instalação da `dependencia-comum 3.9` porque não é possível manter duas versões da mesma biblioteca.

Resultado: a versão final instalada da `dependencia-comum` é `dependencia-comum 4.9`, que não é compatível com a versão requerida pela `biblioteca-a`.

Então quando você executar o código do seu projeto, a `biblioteca-a` não funcionará porque ela espera `dependencia-comum < 4.0`, mas você tem `dependencia-comum 4.9`.

O `pip` não vai avisar que existe esse conflito entre as duas `biblioteca-a` e `biblioteca-b`. Você só descobrirá que algo está errado quando executar o código do seu projeto e ele falhar. Aí terá que investigar por que falhou, eventualmente (com sorte) descobrindo que é porque existe um conflito de dependências, e então terá que resolver o conflito manualmente instalando uma versão de `dependencia-comum` que seja compatível tanto com `biblioteca-a` quando com `biblioteca-b`.

Em resumo, precisamos de uma ferramenta que:

- Resolva conflitos de dependências de forma inteligente.
- Gerencie ambientes virtuais de forma mais automatizada.
- Diferencie entre dependências de produção e desenvolvimento.
- Facilite a publicação de pacotes.
- Garanta a reprodutibilidade do ambiente de desenvolvimento.

É aí que entra o **Poetry**.

Gerenciamento Avançado de Dependências com o Poetry

O que é o Poetry?

O Poetry é uma ferramenta moderna de linha de comando para gerenciamento de dependências e pacotes em Python. Ele simplifica a criação de ambientes virtuais, a resolução de dependências, a instalação de pacotes e a publicação de projetos. O Poetry usa os arquivos `pyproject.toml` para definir as configurações do projeto e as dependências, e `poetry.lock` para garantir instalações consistentes.

Como o Poetry Funciona?

O Poetry funciona gerenciando um ambiente virtual para o seu projeto (semelhante ao `venv`) e instalando as dependências definidas no arquivo `pyproject.toml`. Ele resolve automaticamente as dependências e suas versões compatíveis, garantindo que o projeto funcione corretamente em diferentes ambientes.

Arquivo `pyproject.toml`:

Este arquivo é o coração do Poetry. Ele define as metainformações do projeto (nome, versão, descrição, etc.) e as dependências diretas.

Exemplo de `pyproject.toml`:

```toml
[tool.poetry]
name = "meu-projeto"
version = "0.1.0"
description = "Um projeto de exemplo"
authors = ["Seu Nome <seu.email@example.com>"]
readme = "README.md"


requires-python = ">=3.13"
dependencies = [
    "requests (>=2.32.3,<3.0.0)"
]

[tool.poetry.group.dev.dependencies]
pytest = "^8.3.4"

[build-system]
requires = ["poetry-core>=2.0.0,<3.0.0"]
build-backend = "poetry.core.masonry.api"
```

- `[tool.poetry]`: Define as metainformações do projeto.
- `requires-python = ">=3.13"`: Especifica que o projeto é compatível com Python 3.13 e versões superiores.
- `dependencies = ["requests (>=2.32.3,<3.0.0)"]`: Especifica que o projeto precisa da biblioteca `requests` na versão 2.32.3 ou superior, mas inferior à versão 3.0.0.
- `[tool.poetry.group.dev.dependencies]`: Lista as dependências necessárias apenas para desenvolvimento, como ferramentas de teste (por exemplo, `pytest`).
- `[build-system]`: Especifica as configurações de construção do projeto.

Gerenciamento Avançado de Dependências com o Poetry

Arquivo `poetry.lock`:

Este arquivo é gerado automaticamente pelo Poetry e contém as versões exatas de todas as dependências instaladas, incluindo as dependências indiretas. Ele garante que o ambiente virtual possa ser recriado de forma consistente em diferentes máquinas.

Como o Poetry Resolve os Problemas do venv + pip?

O Poetry aborda as limitações do `venv` + `pip` das seguintes maneiras:

1. **Resolução Inteligente de Dependências:**
   - O Poetry usa um algoritmo avançado de resolução de dependências que analisa todas as dependências e suas subdependências, criando uma árvore de dependências completa.
   - Ele tenta encontrar um conjunto de versões compatíveis entre si, levando em conta as restrições de versão de cada pacote.
   - Se houver conflitos, o Poetry informará quais dependências estão causando o problema e sugerirá possíveis soluções.
2. **Arquivo de Bloqueio (`poetry.lock`):**
   - O Poetry gera automaticamente um arquivo `poetry.lock` que lista as versões exatas de todas as dependências instaladas, incluindo as dependências indiretas, e seus hashes (códigos de verificação de integridade).
   - Isso garante que o ambiente virtual possa ser recriado de forma idêntica em qualquer máquina, garantindo a reprodutibilidade do ambiente de desenvolvimento.
   - Quando você executa `poetry install` em um projeto com um arquivo `poetry.lock`, o Poetry instala exatamente as mesmas versões de pacotes que foram instaladas originalmente, independentemente de novas versões terem sido lançadas.
3. **Gerenciamento de Ambientes Virtuais Integrado:**
   - O Poetry gerencia automaticamente os ambientes virtuais para você. Você não precisa criar ou ativar manualmente os ambientes virtuais com `venv`.
   - O comando `poetry shell` ativa o ambiente virtual associado ao projeto, e `poetry run` executa comandos dentro desse ambiente.
4. **Arquivo `pyproject.toml` Padronizado:**
   - O Poetry usa o arquivo `pyproject.toml` para armazenar as configurações do projeto e as dependências.
   - O `pyproject.toml` é um padrão emergente para configuração de projetos Python, centralizando todas as informações do projeto em um único lugar.
   - O formato TOML é mais legível e fácil de entender do que os arquivos de configuração tradicionais do Python, como `setup.py` ou `requirements.txt`.
5. **Separação de Dependências de Desenvolvimento:**
   - O Poetry permite que você defina dependências de desenvolvimento separadamente das dependências de produção no arquivo `pyproject.toml`.
   - Isso ajuda a manter os ambientes de produção mais enxutos, instalando apenas o que é realmente necessário para o projeto rodar.
6. **Facilidade de Publicação:**
   - O Poetry simplifica o processo de construção e publicação de pacotes Python no PyPI.
   - Os comandos `poetry build` e `poetry publish` automatizam as etapas necessárias.

Gerenciamento Avançado de Dependências com o Poetry

Poetry na Prática

1 — Instalação do Poetry:

Conforme as instruções da documentação oficial:
https://python-poetry.org/docs/#installing-with-the-official-installer

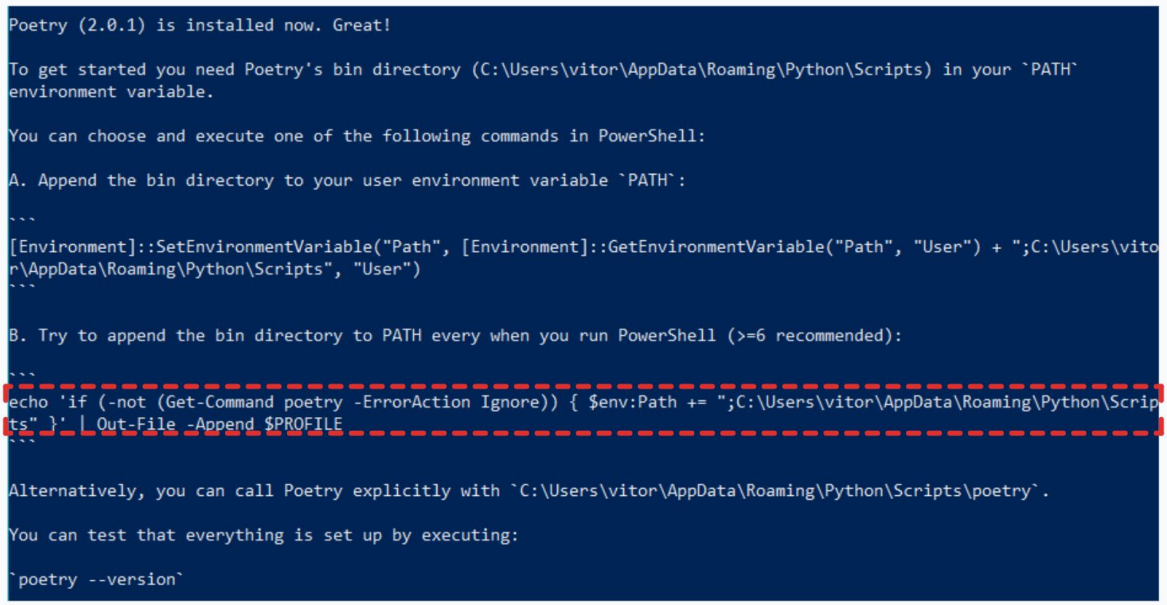
No Linux/MacOS:

```
curl -sSL https://install.python-poetry.org | python3 -
```

No Windows Powershell:

```
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content |
py -
```

Quando a instalação terminar, dentro do terminal haverá uma mensagem indicando que você deve executar mais algum comando para finalizar a configuração do Poetry na máquina. Esse comando varia conforme o seu sistema:



Copie e cole o comando no seu terminal e execute.

Com isso o Poetry foi instalado, ele não funcionará no mesmo terminal que você usou para instalação, mas funcionará em todos os novos terminais que você abrir. Caso você use terminal pelo VSCode, terá que fechar todas as janelas do VSCode e abrir novamente para que o Poetry seja encontrado e funcione.

Gerenciamento Avançado de Dependências com o Poetry

2 — Inicializando um novo projeto:

```
poetry new meu-projeto-poetry
```

O nome pode ser qualquer um, não precisa ser `meu-projeto-poetry` (mas recomendamos um nome sem espaços).

Isso criará um diretório chamado `meu-projeto-poetry` com a estrutura básica de um projeto Python gerenciado pelo Poetry, incluindo o arquivo `pyproject.toml`:

```
meu-projeto-poetry
├── pyproject.toml
├── README.md
├── meu_projeto_poetry
│   └── __init__.py
└── tests
    └── __init__.py
```

Este é um esqueleto básico de um projeto Python moderno, configurado pelo Poetry. Vamos entender o propósito de cada elemento:

Gerenciamento Avançado de Dependências com o Poetry

- `pyproject.toml`

  Este é o arquivo de configuração principal do seu projeto e o coração do gerenciamento de projetos com o Poetry. Ele contém todas as metainformações do projeto, como nome, versão, descrição, autor, dependências, dependências de desenvolvimento e scripts.

- `README.md`

  Este é um arquivo de texto no formato Markdown que serve como a documentação principal do seu projeto. Ele geralmente contém uma descrição do projeto, instruções de instalação, exemplos de uso e outras informações relevantes.

- `meu_projeto_poetry/` **(diretório com o nome do projeto)**

  Este diretório é onde o código-fonte principal do seu projeto será armazenado. O nome do diretório geralmente corresponde ao nome do projeto (neste caso, `meu_projeto_poetry`).

- `__init__.py` **(dentro do diretório** `meu_projeto_poetry`**)**

  Este arquivo, inicialmente vazio, transforma o diretório `meu_projeto_poetry` em um pacote Python.

  - **Marcar um diretório como um pacote Python:** Indica ao Python que o diretório deve ser tratado como um pacote, permitindo a importação de módulos e subpacotes.
  - **Executar código de inicialização (opcional):** Você pode colocar código de inicialização neste arquivo, que será executado quando o pacote for importado. No entanto, para projetos simples, ele geralmente permanece vazio.

- `tests/`

  Este diretório é onde você colocará os testes unitários do seu projeto.

  - **Armazenar testes:** Conter os arquivos de teste que verificam se o seu código está funcionando conforme o esperado.
  - **Promover boas práticas de desenvolvimento:** Incentiva a escrita de testes, o que ajuda a garantir a qualidade do código e a prevenir bugs.
  - **Facilitar a integração contínua:** Os testes podem ser executados automaticamente por ferramentas de integração contínua (CI) para garantir que novas alterações não quebrem o código existente (CI configurado à parte, não faz parte do Poetry, está fora do nosso escopo atual).

- `__init__.py` **(dentro do diretório** `tests`**)**

  Assim como o `__init__.py` no diretório do projeto, este arquivo transforma o diretório `tests` em um pacote Python. Permite que o Python reconheça o diretório como um pacote, facilitando a organização e a execução dos testes.

Gerenciamento Avançado de Dependências com o Poetry

Por que essa estrutura é criada?

O Poetry cria essa estrutura para incentivar boas práticas de desenvolvimento e facilitar a organização e a manutenção de projetos Python. Essa estrutura:

- **Promove a modularidade:**
  Ao organizar o código em pacotes, você incentiva a modularidade e a reutilização de código.
- **Facilita o teste:**
  A separação do código-fonte e dos testes em diretórios distintos torna mais fácil escrever, organizar e executar testes.
- **Simplifica a distribuição:**
  A estrutura segue as convenções para a criação de pacotes Python, tornando mais fácil distribuir seu projeto para outras pessoas.
- **Prepara para o crescimento:**
  Mesmo que seu projeto seja pequeno inicialmente, essa estrutura fornece uma base sólida para o crescimento futuro.

Em resumo, o esqueleto inicial criado pelo Poetry é um ponto de partida bem organizado e estruturado para o desenvolvimento de projetos Python, seguindo as melhores práticas da comunidade e preparando o terreno para um desenvolvimento eficiente e sustentável.

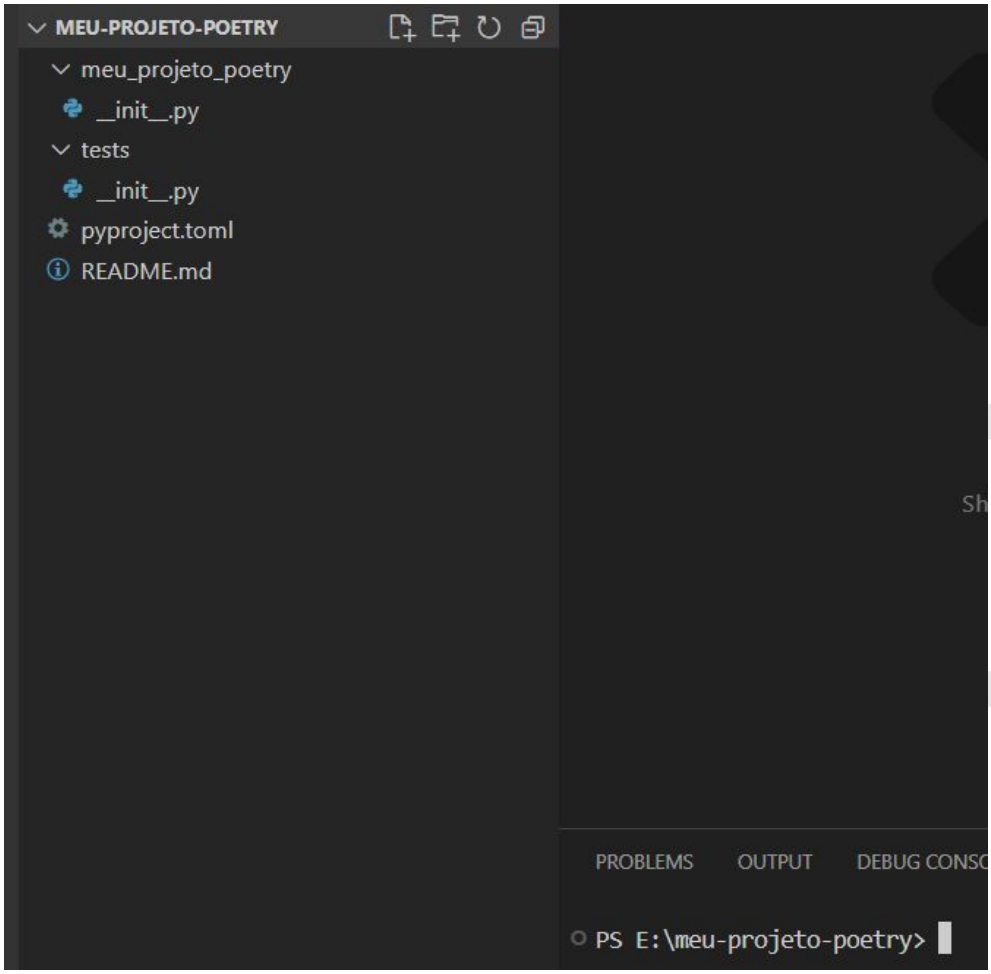
3 — Navegue até o diretório do projeto:

```
cd meu-projeto-poetry
```

Se quiser, pode abrir o VSCode neste diretório:

```
code .
```

Gerenciamento Avançado de Dependências com o Poetry



4 — Desconfigure o “modo pacote” do Poetry

Abra o `pyproject.toml` e no final do arquivo adicione as duas linhas abaixo:

```
[tool.poetry]
package-mode = false
```

Isso configura o Poetry para que ele não espere que seu projeto tenha a estrutura de diretórios de um pacote Python publicável.

Gerenciamento Avançado de Dependências com o Poetry

Explicação:

O Poetry tem funcionalidades tanto para gerenciar dependências quanto para ajudar na publicação de pacotes (se você quiser publicar seu projeto no PyPI para que outros desenvolvedores possam instalá-lo com `pip install`).

Para realizar essa funcionalidade de publicação de pacotes, o Poetry espera que seu projeto tenha uma estrutura de diretórios específica, aquela que foi criada pelo `poetry new`.

Como não estamos interessados em publicar pacotes, somente em gerenciar dependências, a linha `package-mode = false` desabilita as funções de publicação do Poetry, e com isso não precisamos mais seguir a estrutura de diretórios que o Poetry criou.

Ou seja, agora você pode apagar se quiser:

- Pasta `meu_projeto_poetry` com o `__init__.py` dentro.
- Pasta `tests` com o `__init__.py` dentro.
- Arquivo `README.md`

Só é obrigatório deixar o `pyproject.toml` porque é por meio desse arquivo que o Poetry entende as configurações do projeto para ser capaz de gerenciar as dependências.

5 — Adicionando dependências:

Exemplificando a instalação da biblioteca `requests` (você pode instalar o que quiser):

`poetry add requests`

Isso adicionará a biblioteca `requests` ao arquivo `pyproject.toml` e a instalará no ambiente virtual do projeto, além de criar o arquivo `poetry.lock`.

```
# trecho do pyproject.toml após instalar requests
dependencies = [
   "requests (>=2.32.3,<3.0.0)"
]
```

6 — Adicionando dependências de desenvolvimento:

Exemplo adicionando `pytest`, você pode adicionar qualquer dependência de desenvolvimento desejada:

```
poetry add pytest --dev
```

Isso adicionará a biblioteca `pytest` como uma dependência de desenvolvimento (`--dev`) ao arquivo `pyproject.toml`.

```
# trecho do pyproject.toml após instalação do pytest
[tool.poetry.group.dev.dependencies]
pytest = "^8.3.4"
```

Gerenciamento Avançado de Dependências com o Poetry

Onde as Dependências (de Desenvolvimento e Produção) Foram Instaladas?

Depois de instalar o `requests` como uma dependência de produção e o `pytest` como uma dependência de desenvolvimento usando os comandos `poetry add requests` e `poetry add pytest --group dev`, você pode estar se perguntando: onde essas bibliotecas foram instaladas? Afinal, ao contrário do `venv`, o Poetry não cria um diretório `.venv` óbvio dentro da pasta do projeto.

Ambientes Virtuais Gerenciados pelo Poetry:

O Poetry, de fato, cria um ambiente virtual para cada projeto, mas ele gerencia esses ambientes virtuais de forma centralizada, em um local fora da pasta do projeto. Isso ajuda a manter a pasta do projeto limpa.

Localização Padrão dos Ambientes Virtuais do Poetry:

A localização padrão dos ambientes virtuais do Poetry varia de acordo com o sistema operacional:

- **Windows:** `%LOCALAPPDATA%\pypoetry\Cache\virtualenvs` (geralmente `C:\Users\<seu_usuario>\AppData\Local\pypoetry\Cache\virtualenvs`)
- **macOS:** `~/Library/Caches/pypoetry/virtualenvs`
- **Linux:** `~/.cache/pypoetry/virtualenvs`

Se estiver na dúvida, você pode perguntar diretamente ao Poetry onde ele armazena os ambientes virtuais, digitando em qualquer terminal:

```
poetry config virtualenvs.path
```

No meu computador, isso mostra:

```
C:\Users\vitor\AppData\Local\pypoetry\Cache\virtualenvs
```

Encontrando o Ambiente Virtual do seu Projeto:

Dentro do diretório de ambientes virtuais do Poetry, você encontrará um diretório para cada projeto gerenciado por ele. O nome do diretório do ambiente virtual segue um padrão:

`{nome-do-projeto}-{hash}-{versão-do-python}`

- `{nome-do-projeto}`**:** O nome do seu projeto, com hifens substituindo espaços e caracteres especiais.
- `{hash}`**:** Um hash gerado pelo Poetry para identificar exclusivamente o projeto.
- `{versão-do-python}`**:** A versão do Python usada no ambiente virtual.

Exemplo:

Para o projeto `meu-projeto-poetry` que criei anteriormente, o ambiente virtual está localizado no meu computador em:

`C:\Users\vitor\AppData\Local\pypoetry\Cache\virtualenvs\meu-projeto-poetry-3hIRfkUB-py3.13`

(No Linux/macOS, o caminho seria semelhante, mas com a estrutura de diretórios correspondente).

Gerenciamento Avançado de Dependências com o Poetry

Verificando o Conteúdo do Ambiente Virtual:

Dentro do diretório do ambiente virtual, você encontrará a estrutura familiar de um ambiente virtual Python:

```
meu-projeto-poetry-3hIRfkUB-py3.13/
├── bin/ (ou Scripts/ no Windows)
│   ├── python
│   ├── pip
│   └── ...
├── lib/ (ou Lib/ no Windows)
│   └── python3.13
│       └── site-packages
│           ├── requests
│           ├── pytest
│           └── ...
└── ...
```

- `bin/` **(ou** `Scripts/` **no Windows):** Contém os executáveis do ambiente virtual, como `python`, `pip` e os scripts de ativação.
- `lib/` **(ou** `Lib/` **no Windows):** Contém as bibliotecas instaladas, incluindo as dependências de produção e desenvolvimento. Você encontrará o `requests`, `pytest` e suas dependências dentro do diretório `site-packages`.

Por que o Poetry usa esse local para ambientes virtuais?

- **Organização Centralizada:** Manter os ambientes virtuais em um local centralizado facilita o gerenciamento e a limpeza de ambientes virtuais antigos ou não utilizados.
- **Mantém a Pasta do Projeto Limpa:** Mantém a pasta do projeto focada no código-fonte e nos arquivos de configuração, sem a confusão de um diretório `.venv`.

Versão do Python instalada no ambiente virtual do Poetry

O Poetry cria o ambiente quando você executa pela primeira vez o comando `poetry add <pacote>` (porque ele é obrigado a criar um ambiente para ter onde instalar o pacote) ou o comando `poetry install` (veremos sobre esse comando adiante).

Nesse momento, o Poetry instala no ambiente virtual a versão do Python que está atualmente ativa. Então se você quer controlar qual versão do Python será instalada no ambiente virtual, faça o seguinte:

- Antes do `poetry new`, troque para a versão desejada do Python usando o PyEnv: `pyeng global <versão>`
- Agora pode criar o projeto com `poetry new` e então instalar os pacotes com `poetry add`. O Poetry vai criar o ambiente virtual instalando nele a versão ativa do python.
- Se você já fez o `poetry new` mas ainda não fez o `poetry add`, ainda dá tempo de trocar a versão do Python com PyEnv antes de fazer o `poetry add` (porque esse comando será o momento em que o Poetry vai criar o ambiente virtual)

Em resumo:

O Poetry gerencia os ambientes virtuais de forma centralizada, fora da pasta do projeto. Você pode encontrar o ambiente virtual do seu projeto no diretório de cache do Poetry, em uma pasta nomeada com o nome do projeto, um hash e a versão do Python. Dentro do ambiente virtual, você encontrará as dependências de produção e desenvolvimento instaladas no diretório `site-packages`. Embora essa abordagem possa parecer diferente no início, ela oferece vantagens em termos de organização, limpeza e gerenciamento de ambientes virtuais.

Gerenciamento Avançado de Dependências com o Poetry

7 — Executando Comandos no Ambiente Virtual com poetry run

Diferentemente do `venv`, o Poetry não ativa explicitamente o ambiente virtual no seu terminal. Em vez disso, ele gerencia o ambiente virtual nos bastidores e o utiliza temporariamente quando necessário, como ao instalar dependências com `poetry add` ou `poetry install`.

Isso significa que, se você simplesmente executar `python` ou `pip` no seu terminal, **não** estará usando os executáveis do ambiente virtual gerenciado pelo Poetry. Você estará usando os executáveis Python globais (ou os de outro ambiente que esteja ativo no momento).

Como executar comandos dentro do ambiente virtual do Poetry?

Para executar comandos usando o interpretador Python e as bibliotecas instaladas no ambiente virtual do seu projeto, você deve usar o comando `poetry run`.

Sintaxe:

```
poetry run <comando>
```

O `poetry run` garante que o `<comando>` seja executado dentro do ambiente virtual do projeto, usando o Python e os pacotes instalados nesse ambiente.

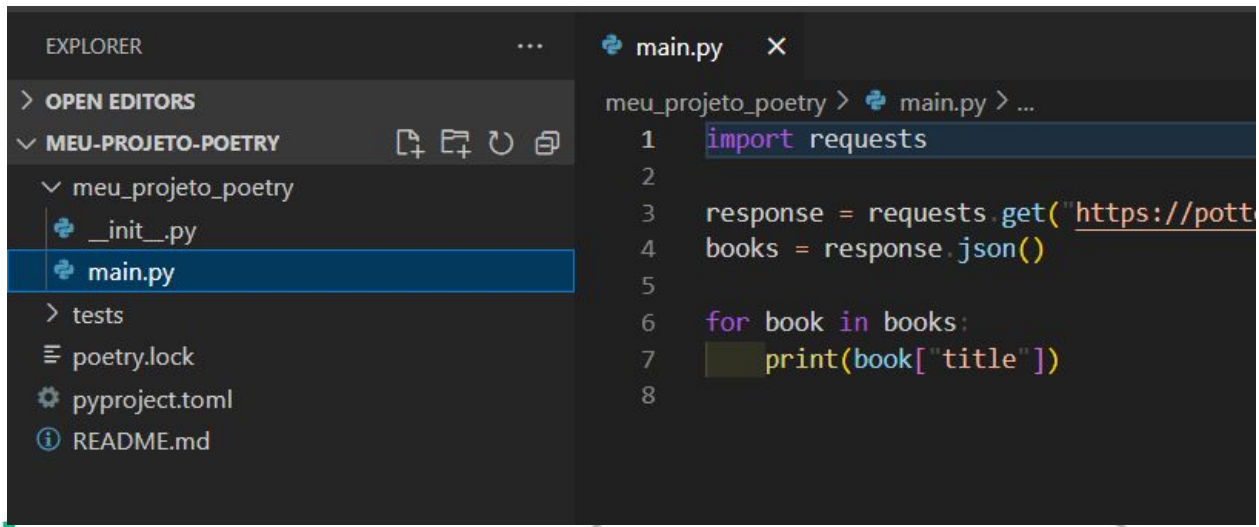
Exemplo Prático:

Vamos criar um arquivo `main.py` no nosso projeto (`meu-projeto-poetry`) com o seguinte conteúdo, que importa a biblioteca `requests` (que instalamos anteriormente com `poetry add requests`) e busca os livros da API do Harry Potter:

```python
# meu_projeto_poetry/main.py
import requests

response = requests.get("https://potterapi-fedeperin.vercel.app/pt/books")
books = response.json()

for book in books:
    print(book["title"])
```



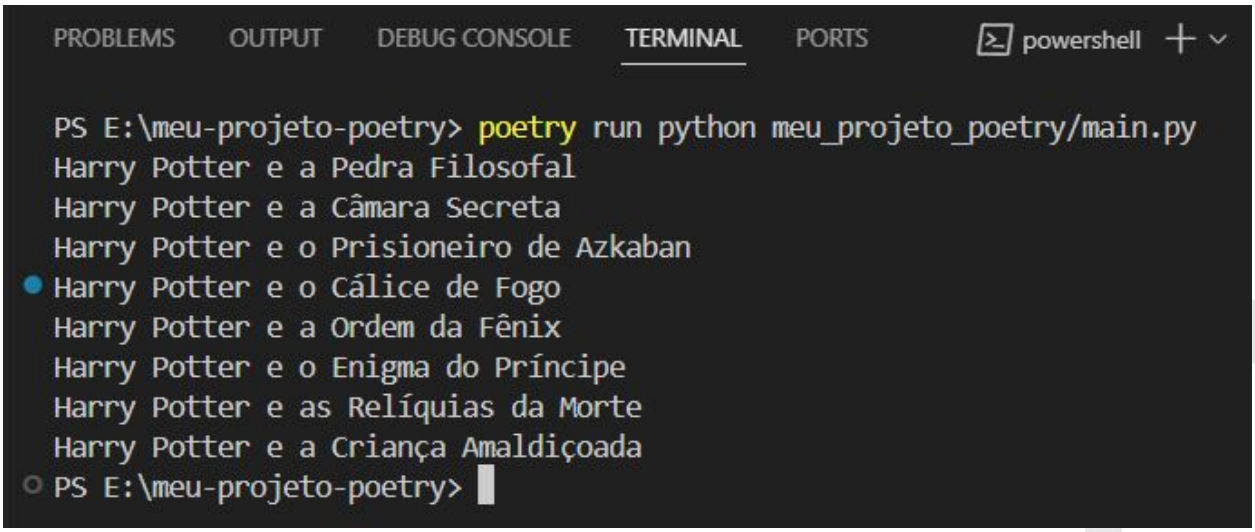
Gerenciamento Avançado de Dependências com o Poetry

Se tentarmos executar esse script diretamente com `python meu_projeto_poetry/main.py`, provavelmente obteremos um erro, pois o `requests` não está instalado no ambiente global (e sim, no ambiente virtual do Poetry).

Para executar o script corretamente, usando o ambiente virtual do Poetry, devemos usar:

```
poetry run python meu_projeto_poetry/main.py
```

Este comando executará o script `main.py` usando o interpretador Python do ambiente virtual e as bibliotecas instaladas nele, incluindo o `requests`. Você verá a lista de livros do Harry Potter impressa no console.



Verificando qual Python está sendo usado:

Você pode usar o comando `python -c "import sys; print(sys.executable)"` para verificar qual interpretador Python está sendo usado.

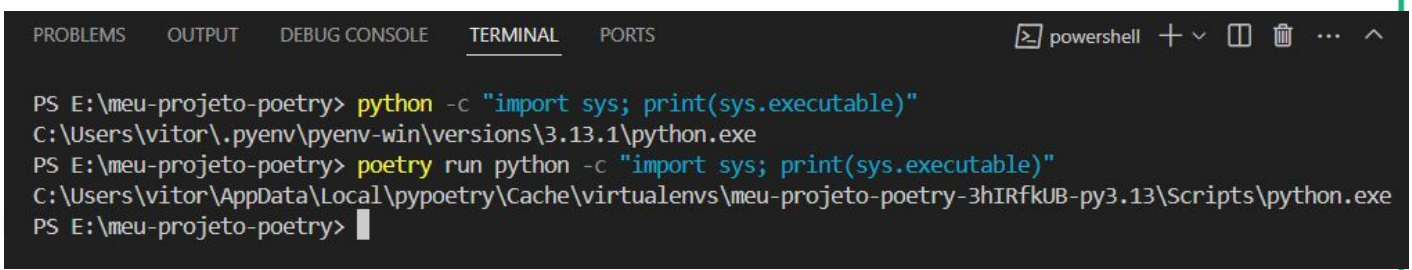
- **Fora do ambiente virtual (usando o Python global):**

```
python -c "import sys; print(sys.executable)"
```

- **Dentro do ambiente virtual (usando `poetry run`):**

```
poetry run python -c "import sys; print(sys.executable)"
```

Compare os resultados:



Gerenciamento Avançado de Dependências com o Poetry

Na imagem acima:

- `C:\Users\vitor\.pyenv\pyenv-win\versions\3.13.1\python.exe` é uma instalação global do python

- `C:\Users\vitor\AppData\Local\pypoetry\Cache\virtualevns\meu-projeto-poetry-3hIRfkUB-py3.13\Scripts\python.exe` é o ambiente virtual do Poetry.

Em resumo:

O Poetry não ativa o ambiente virtual da maneira tradicional. Em vez disso, ele o utiliza temporariamente quando necessário. Para executar comandos dentro do ambiente virtual do seu projeto, use `poetry run <comando>`. Isso garante que você esteja usando o interpretador Python e as bibliotecas instaladas no ambiente virtual, e não os globais. O comando `python -c "import sys; print(sys.executable)"` é uma ferramenta útil para verificar qual interpretador Python está sendo usado em um determinado contexto.

Gerenciamento Avançado de Dependências com o Poetry

Compartilhando um projeto Poetry com Outras Pessoas

Se você quiser compartilhar o projeto com outra pessoa, deve passar para ela sua pasta incluindo o `pyproject.toml` e `poetry.lock`.

Como essa pessoa não terá na máquina dela um ambiente virtual como o seu (que já tem pacotes instalados), ela precisará mandar o Poetry criar o ambiente e instalar as dependências nele.

Para isso na pasta do projeto, deverá fazer:

```
poetry install
```

Este comando tem um papel fundamental no gerenciamento de dependências com o Poetry:

- **Lê o arquivo `pyproject.toml`:**

  O Poetry analisa o arquivo `pyproject.toml` para identificar as dependências do projeto (tanto de produção quanto de desenvolvimento) e suas restrições de versão.

- **Resolve as dependências:**

  Com base nas informações do `pyproject.toml`, o Poetry resolve as dependências, determinando quais versões de cada pacote devem ser instaladas para satisfazer todas as restrições e garantir a compatibilidade entre os pacotes.

- **Verifica o arquivo `poetry.lock`:**

  O Poetry verifica se existe um arquivo `poetry.lock`.

  - **Se o arquivo `poetry.lock` existir:**

    O Poetry instala as versões exatas dos pacotes especificados no arquivo `poetry.lock`, garantindo que o ambiente seja reproduzível. Isso é crucial para a consistência entre os ambientes de desenvolvimento, teste e produção.

  - **Se o arquivo `poetry.lock` não existir:**

    O Poetry resolve as dependências com base nas restrições do `pyproject.toml`, instala as versões mais recentes que satisfaçam essas restrições e cria o arquivo `poetry.lock` para registrar as versões exatas instaladas.

- **Instala ou atualiza os pacotes:**

  O Poetry instala ou atualiza os pacotes no ambiente virtual do projeto, conforme necessário. Se o ambiente virtual ainda não existe, o Poetry cria.

Gerenciamento Avançado de Dependências com o Poetry

`pyproject.toml` e `poetry.lock` no Controle de Versão (Git):

É **essencial** adicionar os arquivos `pyproject.toml` e `poetry.lock` ao seu sistema de controle de versão (como o Git).

- `pyproject.toml`**:** Define as dependências do projeto e suas restrições de versão. Ele atua como a fonte da verdade para as dependências do seu projeto.

- `poetry.lock`**:** Garante que as mesmas versões exatas das dependências sejam instaladas em todos os ambientes, tornando o ambiente de desenvolvimento reproduzível.

Fluxo de Trabalho Recomendado:

1. **Adicione** `pyproject.toml` **e** `poetry.lock` **ao Git:**

   ```
   git add pyproject.toml poetry.lock
   git commit -m "Adiciona arquivos de configuração do Poetry"
   ```

2. **Quando outra pessoa (ou você mesmo em outra máquina) clonar o projeto, ela deve executar:**

   ```
   poetry install
   ```

   Como o arquivo `poetry.lock` está presente, o Poetry instalará as versões exatas das dependências registradas no arquivo, garantindo um ambiente consistente.

Comportamento do `poetry install` em Diferentes Cenários:

- **Primeira instalação em um novo ambiente:** Se você acabou de clonar um projeto com `pyproject.toml` e `poetry.lock` e executar `poetry install`, o Poetry criará um novo ambiente virtual (se ainda não existir) e instalará todas as dependências listadas no `poetry.lock` nas versões exatas especificadas.

- **Ambiente já configurado (com dependências instaladas):**
  Se você já executou `poetry install` anteriormente e as dependências já estão instaladas, ao executar `poetry install` novamente, o Poetry verificará se as versões instaladas ainda correspondem às especificações do `poetry.lock`.

  - **Se tudo estiver atualizado:**
    O Poetry informará que não há dependências para instalar ou atualizar. Você verá uma mensagem como:

    ```
    Installing dependencies from lock file (no dependencies to install or update)
    ```

    Isso indica que seu ambiente virtual já está sincronizado com o arquivo `poetry.lock`.

  - **Se houver atualizações disponíveis (mas não exigidas pelo** `poetry.lock`**):**
    O Poetry não atualizará as dependências automaticamente. Você precisará usar `poetry update` para atualizar as dependências para as versões mais recentes permitidas pelo `pyproject.toml` e atualizar o `poetry.lock`.

Gerenciamento Avançado de Dependências com o Poetry

Em resumo:

O comando `poetry install` é a maneira recomendada de instalar as dependências do seu projeto, garantindo a consistência entre diferentes ambientes. Os arquivos `pyproject.toml` e `poetry.lock` devem ser adicionados ao controle de versão para que outros desenvolvedores (ou você mesmo em outra máquina) possam reproduzir facilmente o ambiente do projeto.

O `poetry install` se comporta de forma diferente dependendo se as dependências já estão instaladas e se o `poetry.lock` está atualizado, mas sempre garante que o ambiente virtual esteja de acordo com as especificações do `poetry.lock`.

Criando Ambientes Virtuais Dentro da Pasta do Projeto com Poetry

Por padrão, o Poetry cria e gerencia os ambientes virtuais em um diretório centralizado, fora da pasta do projeto. No entanto, você pode configurar o Poetry para criar os ambientes virtuais dentro da pasta do projeto, de forma semelhante ao `venv`.

Configurando o Poetry para Criar Ambientes Virtuais Localmente:

Para fazer com que o Poetry crie ambientes virtuais dentro da pasta do projeto, você precisa alterar a configuração `virtualenvs.in-project` para `true`. Abra um terminal (não precisa ser dentro de um projeto específico, pois essa é uma configuração global do Poetry) e execute o seguinte comando:

```
poetry config virtualenvs.in-project true
```

Este comando configura o Poetry globalmente. A partir de agora, todos os **novos** projetos criados ou inicializados com o Poetry terão seus ambientes virtuais criados dentro da pasta do projeto, em um diretório `.venv`.

Observação: Essa configuração afeta apenas os **novos** ambientes virtuais criados pelo Poetry. Os ambientes virtuais existentes, criados antes dessa configuração, permanecerão em seus locais originais.

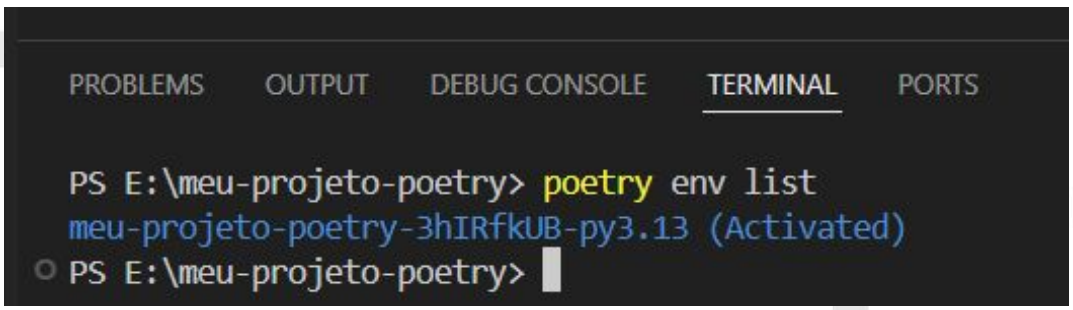
Excluindo um Ambiente Virtual Antigo (Gerenciado pelo Poetry):

Se você já tinha um projeto gerenciado pelo Poetry e deseja que o ambiente virtual seja criado localmente, primeiro você precisa excluir o ambiente virtual antigo (aquele que foi criado no diretório centralizado do Poetry).

Gerenciamento Avançado de Dependências com o Poetry

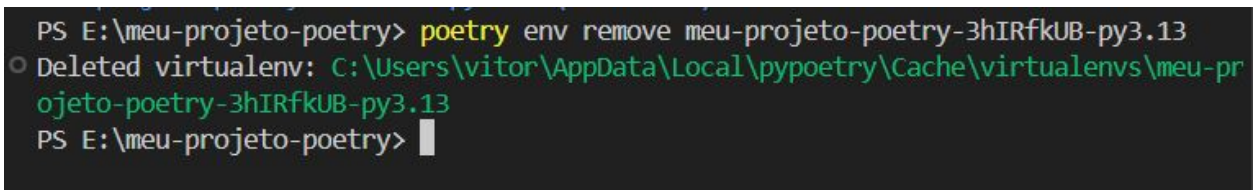
1. **Encontre o nome do ambiente virtual antigo:**
   Você pode usar o comando `poetry env list` para obter informações sobre o ambiente virtual do seu projeto.

```
poetry env list
```



   Anote o nome do ambiente, no meu caso é "meu-projeto-poetry-3hIRfkUB-py3.13"

2. **Remova o ambiente virtual antigo:**
   Você pode usar o comando `poetry env remove <nome do ambiente>`



3. **Crie um novo ambiente virtual localmente (após configurar `virtualenvs.in-project`):**
   Após configurar o Poetry para criar ambientes virtuais localmente (com `poetry config virtualenvs.in-project true`) e remover o ambiente virtual antigo, você pode criar um novo ambiente virtual dentro da pasta do projeto executando:

```
poetry install
```

   Isso criará um novo diretório `.venv` dentro da pasta do seu projeto, contendo o ambiente virtual. E o Poetry irá instalar as dependências nesse novo ambiente virtual (conforme a operação do `poetry install` que já explicamos).

Gerenciamento Avançado de Dependências com o Poetry

Integrando o Poetry a um Projeto Existente

Se você já tem um projeto Python com arquivos de código, esse projeto não estava usando o Poetry (por exemplo você estava usando o `pip` para instalar dependências) e deseja começar a usar o Poetry para gerenciá-lo, você pode fazer isso facilmente.

Inicializando o Poetry em um Projeto Existente:

1. Navegue até o diretório raiz do seu projeto no terminal.

2. Execute o comando:

   ```
   poetry init
   ```

   Este comando iniciará um assistente interativo que irá guiá-lo através da criação de um arquivo `pyproject.toml` para o seu projeto. Ele fará perguntas sobre o nome do projeto, versão, descrição, autor, etc. Você pode aceitar os valores padrão (pressionando Enter) ou fornecer suas próprias informações.

3. **Desabilitar package-mode:** Ao arquivo `pyproject.toml` que foi gerado e adicione as linhas abaixo no final do arquivo:

   ```
   [tool.poetry]
   package-mode = false
   ```

   Isso configura o Poetry para que ele não espere que seu projeto tenha a estrutura de diretórios de um pacote Python publicável (já explicamos sobre o "package-mode = false").

Os próximos passos dependem se o seu projeto já tinha um ambiente virtual ou não:

Gerenciamento Avançado de Dependências com o Poetry

Lidando com Ambientes Virtuais Existentes:

- **Se o seu projeto não tinha um ambiente virtual:**
  Após executar `poetry init` e configurar `package-mode = false`, você pode simplesmente executar `poetry install` para que o Poetry crie um ambiente virtual (no local padrão ou dentro da pasta do projeto, dependendo da sua configuração de `virtualenvs.in-project`) e instale as dependências do projeto.

- **Se o seu projeto já tinha um ambiente virtual (criado com `venv` ou `virtualenv`):**
  Você tem duas opções:

  1. **(Recomendado) Excluir o ambiente virtual antigo e deixar o Poetry criar um novo:**
     Exclua o diretório do ambiente virtual antigo (geralmente `.venv` ou `env`).
     Execute `poetry install` para que o Poetry crie um novo ambiente virtual e instale as dependências.

  2. **Dizer ao Poetry para usar o ambiente virtual existente:**
     Verifique o caminho para o executável "python.exe" que está dentro da pasta do ambiente virtual.
     Geralmente no Windows é `.venv\Scripts\python.exe` e no Linux/Mac é `.venv/bin/python`.
     Configure o Poetry para usar o "python.exe" do ambiente virtual existente:

     ```
     poetry env use .venv\Scripts\python.exe
     ```

     Isso fará com que o Poetry use o ambiente virtual existente em vez de criar um novo.

Em resumo:

Você pode facilmente integrar o Poetry a projetos existentes usando `poetry init`.

Se o projeto não tinha um ambiente virtual, o Poetry pode criar um novo.

Se o projeto já tinha um ambiente virtual, você pode optar por excluir o antigo e deixar o Poetry criar um novo ou configurar o Poetry para usar o ambiente virtual existente.

Lembre-se de configurar o `package-mode = false` no `pyproject.toml` para projetos que não seguem a estrutura de pacotes Python publicáveis.

Gerenciamento Avançado de Dependências com o Poetry

Comparação: Poetry vs. venv + pip

Recurso	Poetry	`venv` + `pip`
Gerenciamento de Dependências	Automático, com resolução de conflitos	Manual, com `requirements.txt`
Arquivo de Bloqueio	`poetry.lock` para instalações consistentes	`pip freeze` pode gerar `requirements.txt` imprecisos
Ambientes Virtuais	Gerenciados automaticamente	Criados manualmente com `venv`
Arquivo de Configuração	`pyproject.toml` (padrão moderno)	`setup.py` (tradicional) ou `requirements.txt` (limitado)
Separação de Dependências	Suporta dependências de desenvolvimento	Não suporta nativamente
Publicação de Pacotes	Integrado	Requer ferramentas adicionais como `setuptools` e `wheel`

Vantagens do Poetry:

- **Gerenciamento de Dependências Simplificado:** Resolve automaticamente conflitos de dependências e garante instalações consistentes.
- **Ambientes Virtuais Integrados:** Gerencia automaticamente os ambientes virtuais para você.
- **Facilidade de Publicação:** Simplifica o processo de publicação de pacotes no PyPI.
- **Arquivo de Configuração Moderno:** Usa o padrão `pyproject.toml` para uma configuração mais limpa e organizada.

Desvantagens do Poetry:

- **Curva de Aprendizado:** Pode levar algum tempo para se acostumar com os comandos e a estrutura do Poetry.
- **Ferramenta Externa:** Requer instalação separada, ao contrário do `venv` que já vem com o Python.
- **Menos Flexível em Alguns Casos:** Pode ser menos flexível do que usar `venv` e `pip` separadamente em alguns casos de uso muito específicos.

Gerenciamento Avançado de Dependências com o Poetry

Aprofundando no assunto

O Poetry é vasto, esse texto é necessariamente limitado e não aborda todas as possibilidades da ferramenta.

Para ver mais, consulte os guias oficiais:

- https://python-poetry.org/

Resumo do Poetry

O Poetry é uma ferramenta poderosa e moderna para gerenciamento de dependências e pacotes em Python.

Ele resolve muitas das limitações do `venv` + `pip`, oferecendo resolução inteligente de dependências, um arquivo de bloqueio para instalações consistentes, gerenciamento automático de ambientes virtuais e um arquivo de configuração padronizado.

Se você está procurando uma solução mais robusta e eficiente para gerenciar seus projetos Python, o Poetry é uma excelente escolha.

Ele simplifica o fluxo de trabalho de desenvolvimento, automatiza tarefas complexas e ajuda a garantir que seus projetos sejam fáceis de instalar, manter e distribuir.

Configurando o VSCode para Usar o Ambiente Virtual do Poetry

Assim como acontece com o `venv`, você precisa configurar o VSCode para usar o interpretador python do ambiente virtual criado pelo Poetry.

Se você configurou o Poetry para criar ambientes virtuais dentro da pasta do projeto (usando `poetry config virtualenvs.in-project true`), o VSCode *provavelmente* encontrará o ambiente virtual automaticamente.

No entanto, se você deixou o Poetry gerenciar os ambientes virtuais em seu local padrão, o VSCode pode não encontrá-los automaticamente.

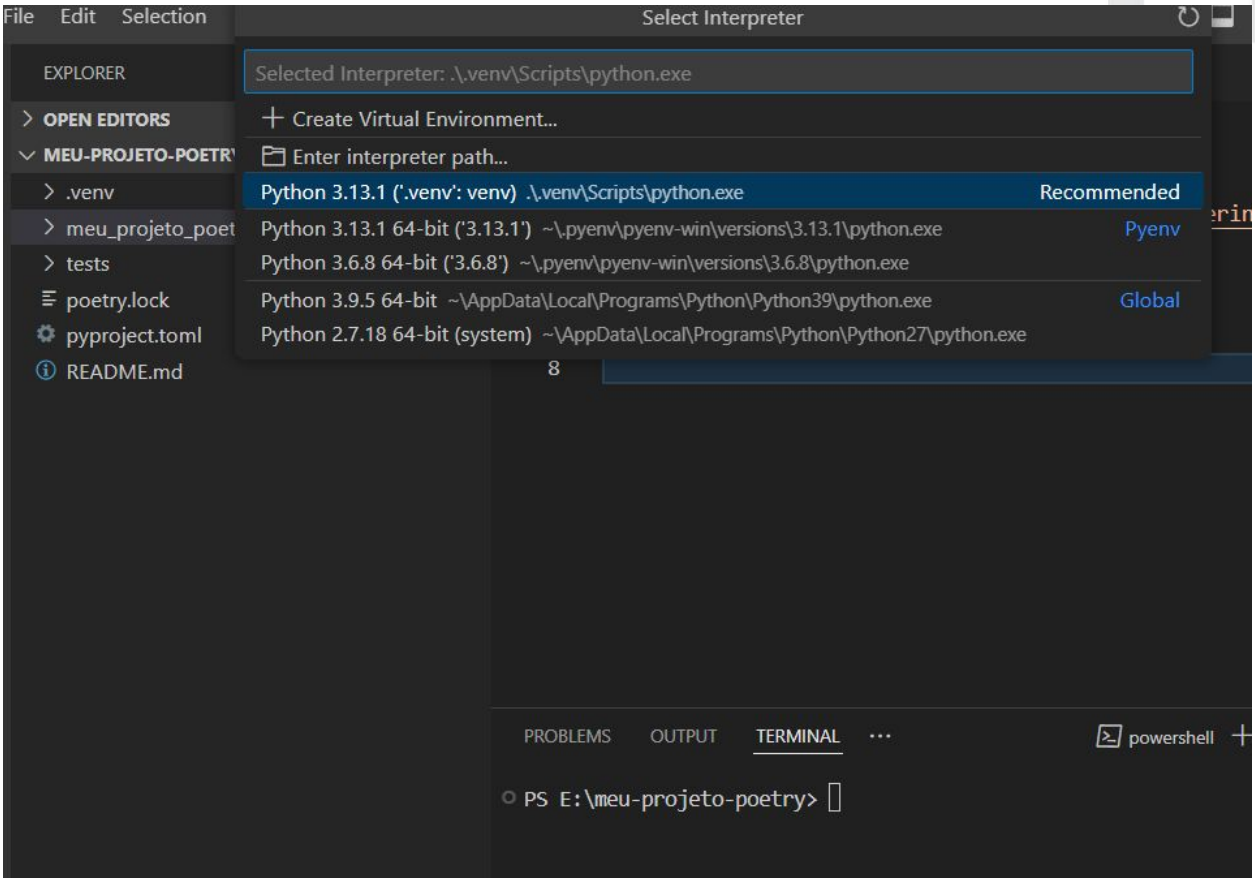
VSCode e Ambientes Virtuais Dentro da Pasta do Projeto

Se você configurou o Poetry para criar ambientes virtuais dentro da pasta do projeto (com o comando `poetry config virtualenvs.in-project true`), o VSCode geralmente detecta o ambiente virtual automaticamente.

Como verificar:

Clique na barra azul inferior do VSCode onde ele mostra a versão do Python que ele está usando (se isso não aparecer para você, aperte F1 e pesquise por "python interpreter", daí escolha a opção "Python: Select Interpreter").

Veja se aparece o "python.exe" do seu ambiente virtual:



Configurando o VSCode para Usar o Ambiente Virtual do Poetry

VSCode e Ambientes Virtuais Gerenciados pelo Poetry

Mas se você deixou o Poetry gerenciar os ambientes virtuais em seu local padrão (fora da pasta do projeto), o VSCode pode não detectá-los automaticamente. Nesse caso, você precisará adicionar o diretório de ambientes virtuais do Poetry às configurações do VSCode.

Passos para configurar o VSCode:

1. **Encontre o diretório de ambientes virtuais do Poetry:**
   Abra um terminal (qualquer terminal, pode ser fora do VSCode) e execute:

   ```
   poetry config virtualenvs.path
   ```

   Este comando imprimirá a pasta onde o Poetry armazena os ambientes virtuais que ele gerencia.
   No meu computador, isso mostra:

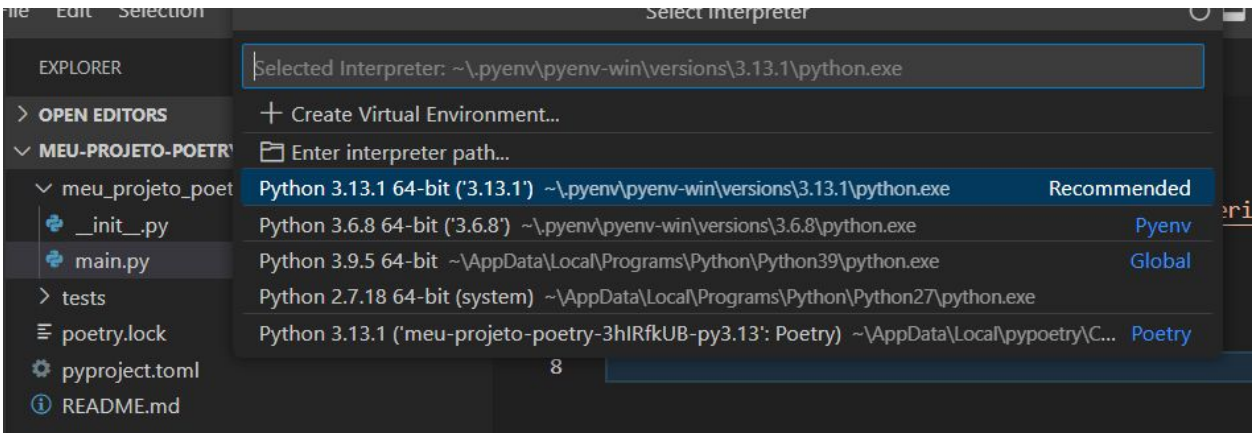
   ```
   C:\Users\vitor\AppData\Local\pypoetry\Cache\virtualenvs
   ```

2. **Abra as configurações do usuário no VSCode:**
   - No VSCode, rressione `F1` para abrir a Paleta de Comandos.
   - Digite "settings" e selecione "Preferences: Open User Settings".

3. **Adicione o diretório do Poetry às configurações de `venvFolders`:**
   - Na barra de pesquisa das configurações, digite "venvFolders".
   - Você verá uma configuração chamada "Python: Venv Folders".
   - Clique em "Add Item".
   - Cole o caminho para o diretório de ambientes virtuais do Poetry que você copiou na etapa 1.
   - Clique em OK.

4. **Reinicie o VSCode, feche todas as janelas e abra de novo**.

5. **Selecione o interpretador do ambiente virtual:**
   Agora ao clicar na barra azul inferior do VSCode onde ele mostra a versão de python que ele está usando, a lista vai incluir os ambientes virtuais gerenciados pelo Poetry (se isso não aparecer para você, aperte F1 e pesquise por "python interpreter", daí escolha a opção "Python: Select Interpreter"):



Na imagem acima, a última opção é o ambiente virtual do Poetry.

Configurando o VSCode para Usar o Ambiente Virtual do Poetry

Após seguir esses passos, o VSCode usará o interpretador Python do ambiente virtual do Poetry para os recursos de inteligência de código e para os novos terminais que você abrir dentro do VSCode.

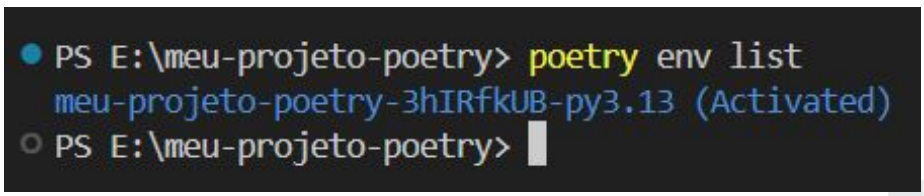
Atenção:

Se você faz vários projetos com o Poetry, para cada projeto o Poetry cria mais um ambiente virtual.

Então ao listar os ambientes no VSCode, vão aparecer várias opções de ambiente virtual, e claramente você deverá escolher o ambiente do projeto onde você está trabalhando agora.

Se não sabe qual dos ambientes do Poetry pertence ao seu projeto, digite num terminal no seu projeto:

```
poetry env list
```



Isso vai mostrar o nome do ambiente virtual correspondente ao seu projeto. No meu caso (imagem acima) apareceu o nome "meu-projeto-poetry-3hIRfkUB-py3.13", por isso sei que esse é o nome que devo escolher na lista do VSCode.

Resumo

- Se o ambiente virtual do Poetry estiver dentro da pasta do projeto, o VSCode provavelmente o encontrará automaticamente.
- Se o ambiente virtual estiver no local gerenciado pelo Poetry, você precisará adicionar o diretório de ambientes virtuais do Poetry às configurações do VSCode (`Python: Venv Folders`).
- Agora o VSCode conseguirá listar os ambientes virtuais gerenciados pelo Poetry e você pode escolher o seu ambiente. Com isso o VSCode usará o interpretador do ambiente virtual do Poetry e ativará automaticamente o ambiente em novos terminais.

Com essa configuração, você garante que o VSCode esteja sempre em sincronia com o ambiente virtual do seu projeto gerenciado pelo Poetry, independentemente de onde o ambiente virtual esteja localizado. Isso proporciona uma experiência de desenvolvimento mais consistente e eficiente.

Por facilidade, recomendamos configurar o Poetry para sempre criar ambiente virtual na mesma pasta do projeto (`poetry config virtualenvs.in-project true`).

Conclusão

Chegamos ao final desta jornada pelo universo do gerenciamento de versões do Python, ambientes virtuais e gerenciamento de dependências! Vamos recapitular os principais conceitos e ferramentas que exploramos:

1. PyEnv:

- **Problema:** Lidar com múltiplas versões do Python no mesmo sistema, especialmente quando diferentes projetos exigem versões distintas.
- **Solução:** O PyEnv permite instalar e alternar facilmente entre diferentes versões do Python, globalmente ou por projeto, resolvendo conflitos de versão.

2. Gerenciamento Manual de Dependências com `pip`:

- **Problema:** Instalar pacotes globalmente pode levar a conflitos de dependências entre projetos e tornar a replicação de ambientes de desenvolvimento difícil. O arquivo `requirements.txt` gerado pelo `pip freeze` pode ser impreciso e incluir pacotes desnecessários.
- **Solução:** O `pip` é útil para instalar pacotes, mas tem limitações no gerenciamento de dependências em projetos complexos. É melhor usar ambientes virtuais e gerenciadores de dependência como o Poetry.

3. Ambientes Virtuais com `venv`:

- **Problema:** Sem isolamento, os pacotes instalados para um projeto podem afetar outros projetos, porque os pacotes são instalados na pasta de instalação global do Python.
- **Solução:** Ambientes virtuais criam ambientes isolados para cada projeto, permitindo instalar dependências específicas sem conflitos. O `venv` é a ferramenta padrão do Python para criar ambientes virtuais.

4. Integração de Ambientes Virtuais no VSCode:

- **Problema:** O VSCode usa um interpretador Python independente para seus recursos de inteligência de código, o que pode levar a inconsistências com o ambiente virtual ativado no terminal.
- **Solução:** Configurar o VSCode para usar o interpretador Python do ambiente virtual garante consistência e ativa automaticamente o ambiente em novos terminais dentro do VSCode.

5. Poetry:

- **Problema:** O gerenciamento de dependências com `venv` e `pip` pode se tornar complexo em projetos maiores, com resolução de dependências ineficiente, arquivos `requirements.txt` imprecisos e gerenciamento manual de ambientes virtuais.
- **Solução:** O Poetry oferece uma solução moderna e robusta, com resolução inteligente de dependências, um arquivo de bloqueio (`poetry.lock`) para instalações consistentes, gerenciamento automático de ambientes virtuais e um arquivo de configuração padronizado (`pyproject.toml`). Ele também facilita a publicação de pacotes e o gerenciamento de dependências de desenvolvimento.

Em resumo, aprendemos a:

- Gerenciar múltiplas versões do Python com o **PyEnv**.
- Instalar pacotes manualmente com o `pip` e entender suas limitações.
- Criar e gerenciar ambientes virtuais com o `venv` para isolar dependências.
- Configurar o VSCode para usar o interpretador Python de um ambiente virtual.
- Utilizar o **Poetry** para um gerenciamento de dependências e ambientes virtuais mais eficiente e moderno.