Criando seu Setup para Programação Python

Asimov Academy

ASIMOV

Conteúdo

01. O que você vai aprender

Neste curso, nosso foco está em **aprender a instalar Python no nosso computador e configurar programas para rodá-lo com sucesso**. Vamos também entender para quê serve um terminal, e como usá-lo para conseguir executar Python sem nenhum problema.

O que motivou este curso?

Existem diversas maneiras de instalarmos Python no nosso computador. Esta flexibilidade é algo positivo, mas para quem está começando, o número de sugestões diferentes pode ser mais confuso que útil. Às vezes, encontramos até mesmo opiniões conflitantes em sites e tutoriais espalhados por aí!

Somado a isso, para os alunos que ainda não possuem familiaridade com ambientes de programação em geral, há também o estranhamento inicial de utilizar um console, CMD ou terminal. O que são estas janelas pretas? Posso acabar fazendo algo de errado no terminal? O que é esse tal de "PATH" que fico ouvindo falar em outras aulas?

Se considerarmos ainda a infinidade de formas e programas para rodar Python, como por exemplo VS Code, PyCharm, Jupyter Notebook, console de Python, ... Temos uma grande fonte de dúvidas de iniciantes. Como configuro cada programa? Cada programa precisa instalar o seu próprio Python? Rodei um comando de `pip install` mas a biblioteca não é encontrada, o que aconteceu?

Por mais que estas questões sejam erros tangenciais à linguagem Python em si, quando nos deparamos com eles (e os iniciantes sempre se deparam com eles), isso torna nossa experiência muito mais frustrante que o necessário.

Na Asimov Academy, sentimos isso na pele. Vemos muitas dúvidas de setup dos nossos alunos que são simples de resolver para quem entende o que está acontecendo, mas completamente enigmáticas para iniciantes.

Foi pensando nisso que decidimos montar um curso **só sobre a configuração do seu setup de programação Python**. Aqui você não vai aprender a desenvolver uma habilidade específica, como construir um dashboard ou realizar uma tipo específico de análise de dados. No lugar disso, você terá um conhecimento profundo de como Python funciona por "debaixo dos panos", e como configurar a instalação de Python em qualquer sistema operacional, seja no seu computador, no dos seus colegas, ou na sua empresa.

Como o curso está estruturado?

Vamos começar falando sobre terminais/CMD, e entender alguns comandos básicos no terminal. Em seguida, aprenderemos a instalar Python nos 3 principais sistemas operacionais: Windows, Mac e Linux. Por fim, seguiremos para a instalação e configuração de programas para trabalhar com Python, como VS Code, PyCharm e notebooks do Jupyter.

Vamos trabalhar sempre usando máquinas virtuais (VMs). Para quem não conhece o conceito, uma VM simula um sistema operacional inteiro rodando dentro do seu computador. Isso será feito para que a instalação seja feita com sistemas "zerados", como se você estivesse instalando Python em um computador que acabou de ser iniciado pela primeira vez. Onde for relevante, vamos também mencionar as diferenças e particularidades de cada sistema operacional.

Não é estritamente necessário acompanhar 100% das aulas - por exemplo, se você sabe que só vai trabalhar em Windows, não necessariamente precisa assistir às instalações em Mac e Linux. Dito isso, ter conhecimento dos outros sistemas operacionais sempre é útil. Mesmo para quem trabalha em Windows, por exemplo, não é raro ter que interagir com servidores ou serviços Cloud que são hospedados em Linux. Nunca sabemos quando teremos que usar outro computador, ou ajudar alguém com alguma configuração!

Então chega de conversa, e vamos aprender!

02. O que é um terminal?

O **terminal** é a forma mais direta que você tem para se comunicar com seu computador. Através dele, é possível rodar programas e comandos que interagem diretamente com seu *hardware*: disco rígido, sistema de arquivos do computador, memória RAM, tela, dispositivos conectados…

Também é possível rodarmos programas convencionais através do terminal. Na realidade, sempre que usamos a interface gráfica do nosso computador para rodar um programa ou abrir um arquivo, por debaixo dos panos o nosso computador envia um comando a um terminal, com uma instrução como “execute este programa” ou “abra este arquivo com aquele programa”.

O conceito de terminal existe há muitos anos, e se mantém até hoje, tanto por motivos históricos quanto pela facilidade de uso. Quando estamos acostumados a usar mouse e cliques na tela para controlar ações do computador, pode parecer estranho ter que digitar um comando no terminal. Mas com a prática, usar o terminal para rodar algum programa ou comando acaba se tornando mais prático do que procurá-lo em alguma pasta ou no nosso Desktop.

A história de um terminal

Antes do surgimento dos computadores modernos, os terminais eram equipamentos físicos usados para executar comandos em computadores. Alguns terminais usavam um display eletrônico para comunicar com o computador, mas os mais antigos (*teletipos*) faziam a conexão através de teclas mecânicas e texto escrito em papel, similares a máquinas de escrever!

A ideia de “printar”, ou exibir algum valor no terminal, vem daqui. Nos teletipos antigos, a saída de um comando era literalmente impresso em papel!

Figure 1: Computador PDP–11/70 ao fundo da sala, com um terminal de vídeo e um teletipo à sua frente

Figure 2: Terminal VT100, usado para comunicar com o computador PDP-11/70

Terminais nos computadores modernos

Mesmo nos computadores desktop e nos notebooks de hoje em dia, a mesma ideia básica de terminal persiste: ele serve para comunicar com o sistema operacional. Essa funcionalidade foi preservada através das décadas porque continuam sendo relevantes hoje em dia, principalmente para programadores e desenvolvedores.

A diferença é que, enquanto nos anos 70 os terminais eram equipamentos físicos, hoje em dia são programas que rodam dentro do próprio computador.

Dessa forma, o nome correto para os terminais de hoje em dia seria **emulador de terminal**, já que mer-

amente “simulam” os equipamentos antigos. Dito isso, no dia a dia é muito mais comum chamarmos estes programas simplesmente de “terminal”.

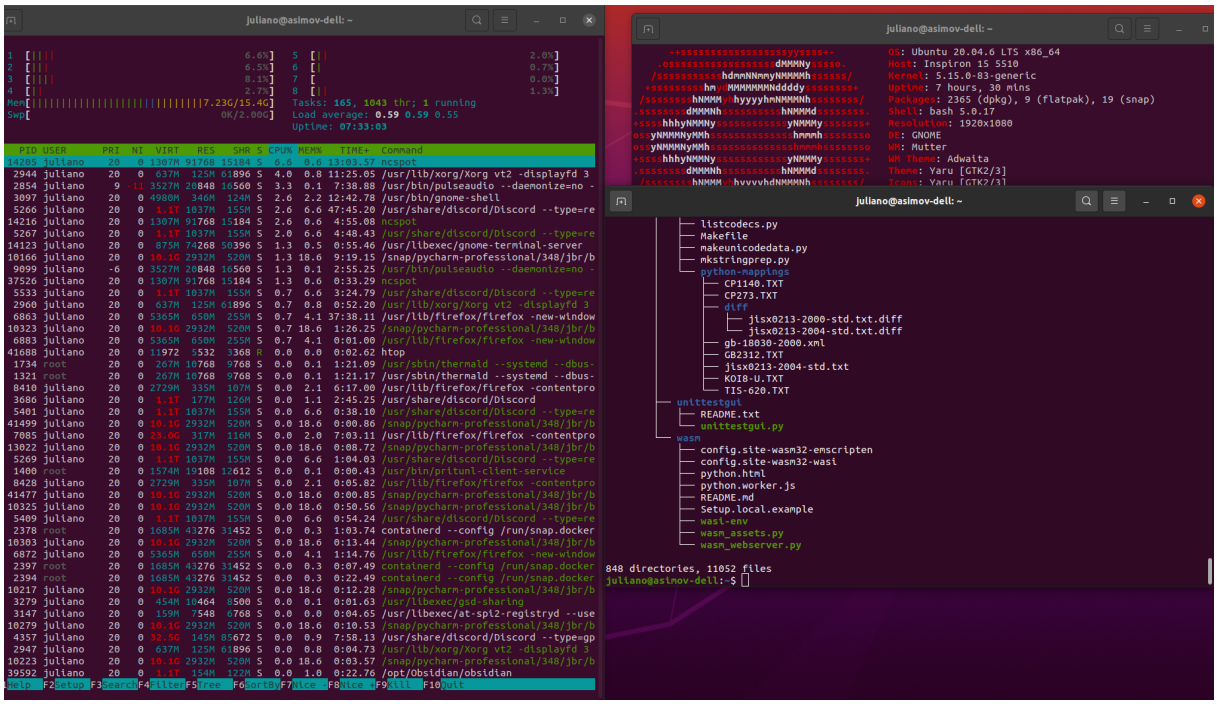


Figure 3: Múltiplos emuladores de terminal rodando em um sistema operacional Linux

Nomenclatura de terminais

Há alguns termos que surgem sempre quando falamos de terminais. Provavelmente você já ouviu falar:

- Terminal
- Shell
- Console
- Prompt de comando
- Linha de comando
- CMD
- …

Tecnicamente, cada termo possui uma origem e um significado específicos. Na prática, acabamos usando todos os termos de forma intercambiável, para representar o terminal. O único detalhe notável aqui é que CMD geralmente é utilizado para se referir à linha de comando do **Windows** (e não de Mac ou Linux).

A linguagem dos terminais

Quando falamos da "linguagem" usada pelos terminais (isto é, quais comandos os terminais aceitam, e como são usados), há uma clara separação entre Windows e Mac/Linux. Esta divisão tem motivos históricos, com base ao sistema que originou cada um destes 3 sistemas operacionais:

- O terminal do **Windows** é baseado no sistema **MS-DOS**.
- O terminal de **Mac e Linux** são baseados no sistema **UNIX**.

Isto significa que os comandos de terminal em Mac e Linux geralmente são idênticos, mas diferentes dos comandos em Windows.

A linguagem usada no terminal de Windows também é chamada de **CMD** (uma vez que o programa que executa se chama `cmd.exe`). Em Mac e Linux, a linguagem se chama **Bash**.

Em termos gerais, a linguagem Bash tem recursos e comandos mais avançados que o CMD. Este é um dos motivos pelo qual muitos desenvolvedores optam por Mac ou Linux: acesso a um terminal com mais recursos. Dito isso, em Windows há também o **PowerShell**, que é uma linguagem expandida baseada no CMD, e que inclui muitas outras funcionalidades.

Ao longo do curso, vamos abordar as diferenças entre sistemas operacionais quando necessário. Vamos agora começar com o terminal (ou CMD) do Windows!

03. O terminal do Windows

O terminal no Windows é acessível buscando pelo programa CMD na barra de busca.

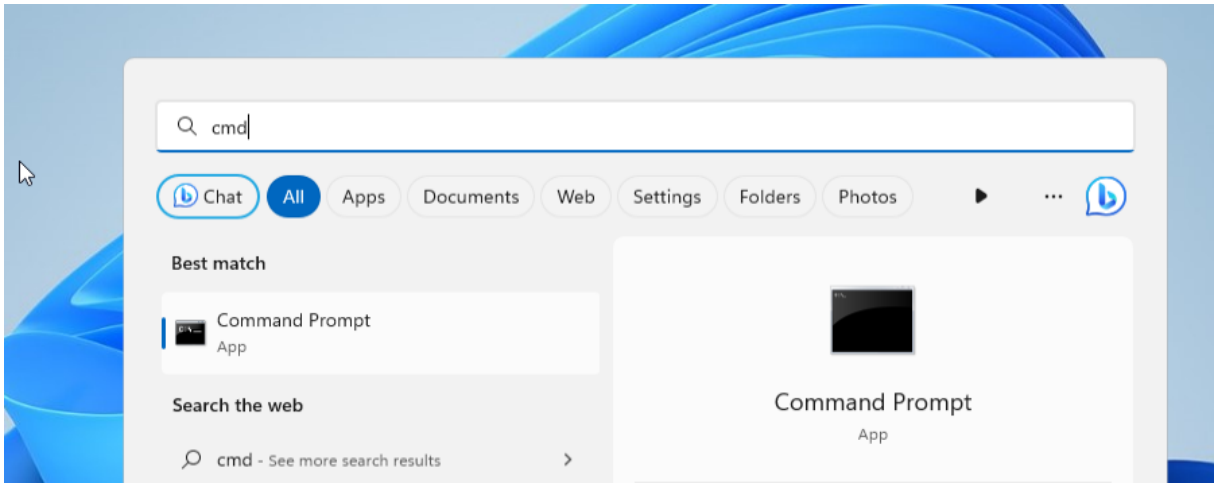


Figure 4: Abrindo o CMD no Windows

Também é possível usar o `PowerShell`, caso você tenha instalado no seu computador.

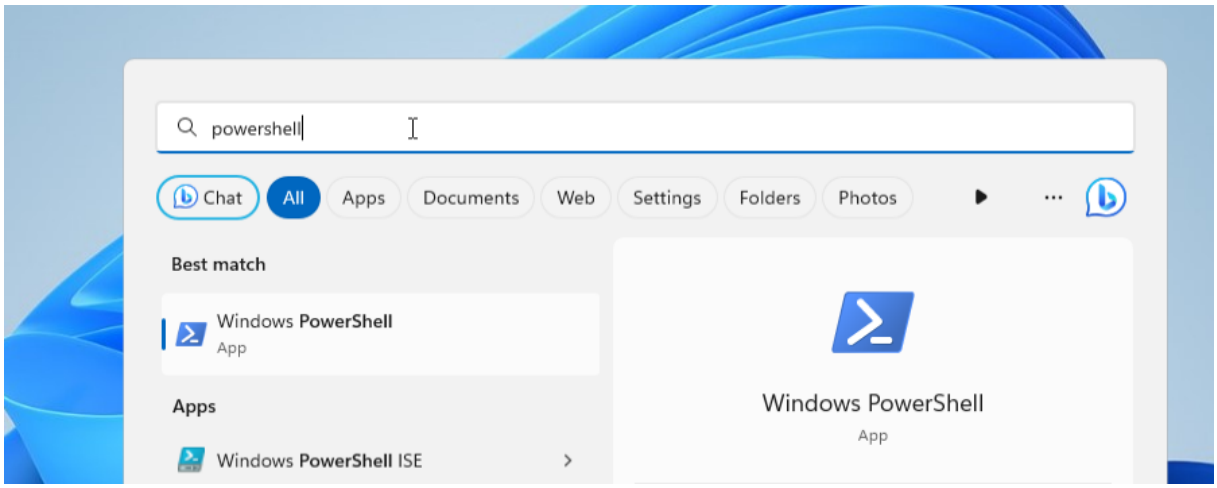


Figure 5: Abrindo o PowerShell no Windows

Comandos básicos

Ao abrir o CMD, você verá que ele estará executando da sua pasta de usuário (normalmente, no caminho `C:\Users\seu-nome-de-usuario`):

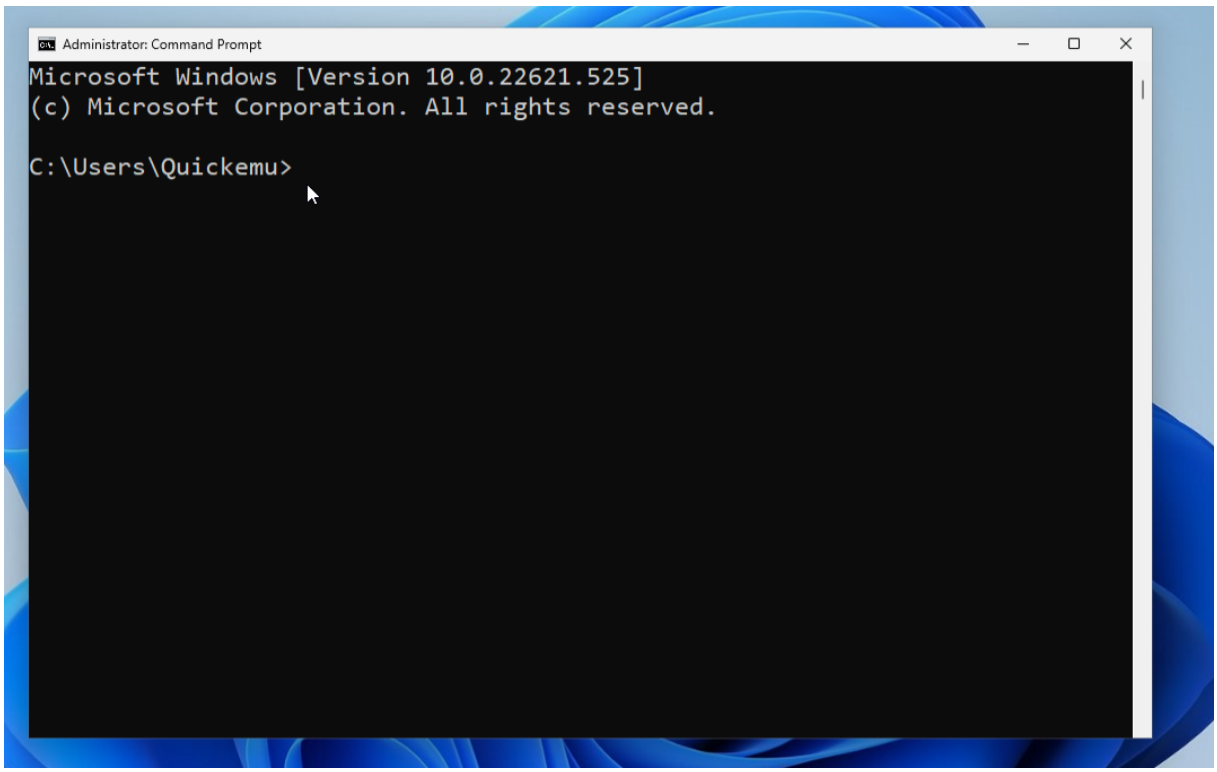


Figure 6: Tela padrão do CMD

O **comando `dir`** é usado para listar os conteúdos dentro da pasta atual. Veja que a sua pasta de usuário contém as principais pastas que você normalmente acessa pelo Windows Explorer, como `Documents`, `Desktop` e `Downloads`:

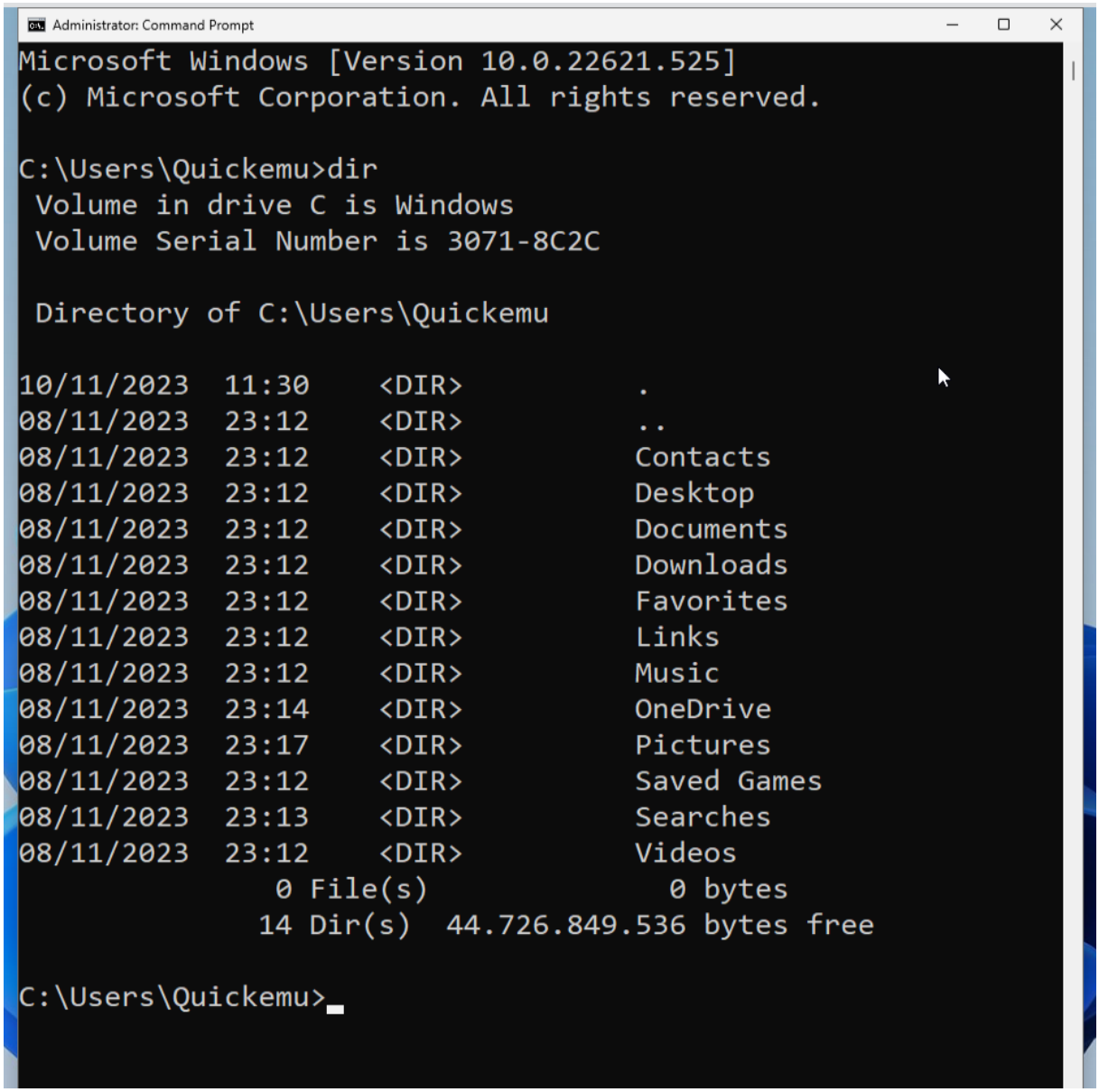


Figure 7: Comando `dir`

Já o **comando `cd`** é usado para mudar de pasta (o nome deriva de ***C**hange **D**irectory*). Use o comando `cd Downloads` para entrar na sua pasta de Downloads, e em seguida o comando `dir` para exibir seu conteúdo:

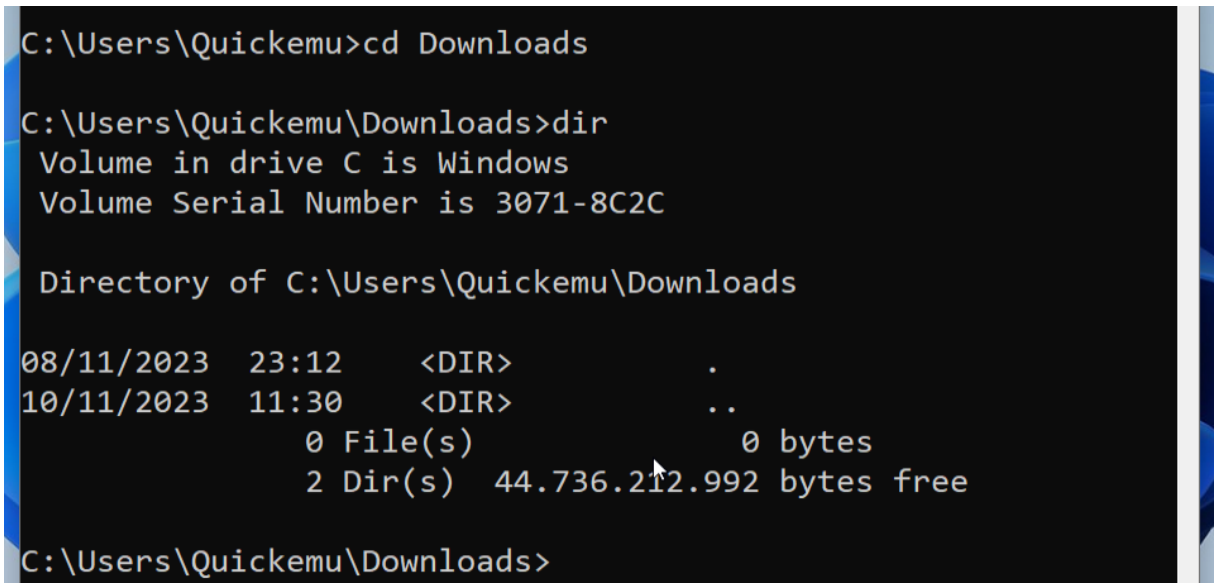


Figure 8: Comando `cd`

Mesmo quando uma pasta está vazia, aparecem duas pastas no comando `dir`: a pasta `.` (ponto), que representa a pasta atual, e a pasta `..` (ponto ponto), que representa a pasta acima da atual. Para ir para uma pasta acima da pasta atual, utilize o comando `cd ..`:

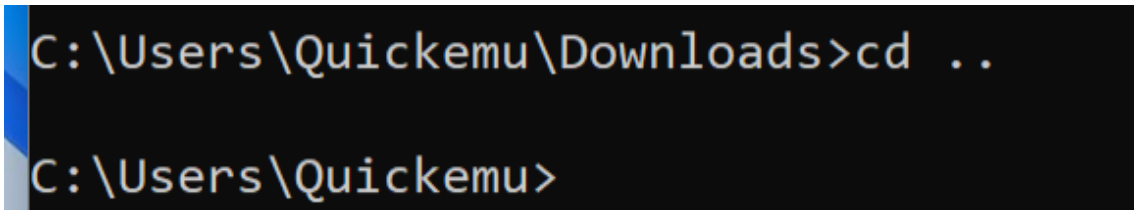


Figure 9: Voltando para pasta anterior com `cd ..`

Dica: use a tecla `Tab` para **autocompletar texto** no terminal. Por exemplo, digite `cd D` e vá apertando a tecla `Tab`. Você passará por todos os nomes de pastas que começam com a letra `D`!

Rodando programas pelo terminal

Através do terminal, é possível rodar programas diretamente. Como um exemplo simples, vamos rodar o Windows Explorer diretamente seguindo as instruções abaixo.

Entre na pasta `C:\Windows` usando os comandos `cd C:\` e `cd Windows`Em seguida, use o comando `dir` para examinar o conteúdo da pasta. Você deverá encontrar o arquivo `explorer.exe` na lista resultante:

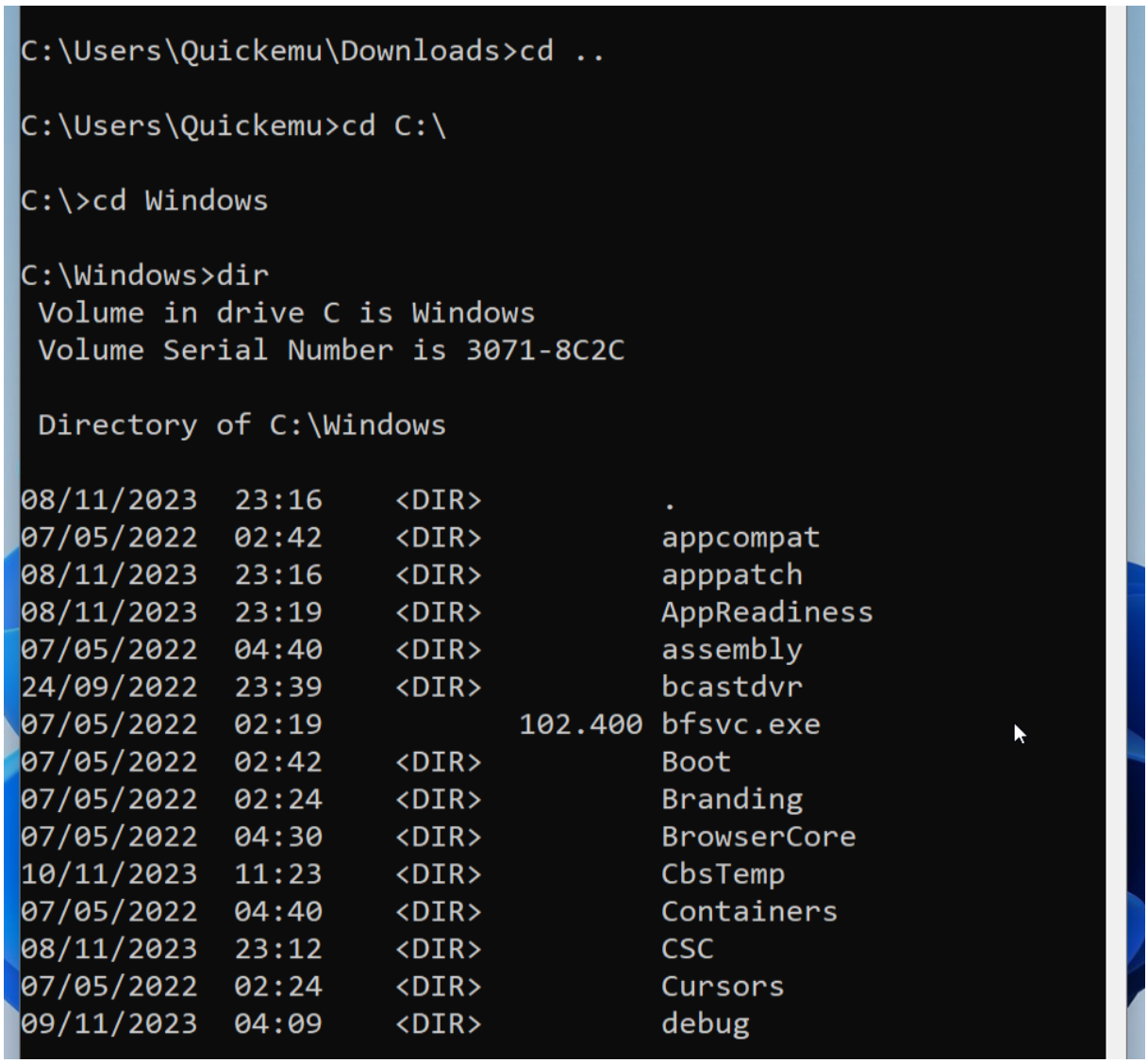


Figure 10: Listando o conteúdo da pasta Windows

Agora, simplesmente escreva `explorer.exe` e aperte `Enter` para rodar o Windows Explorer:

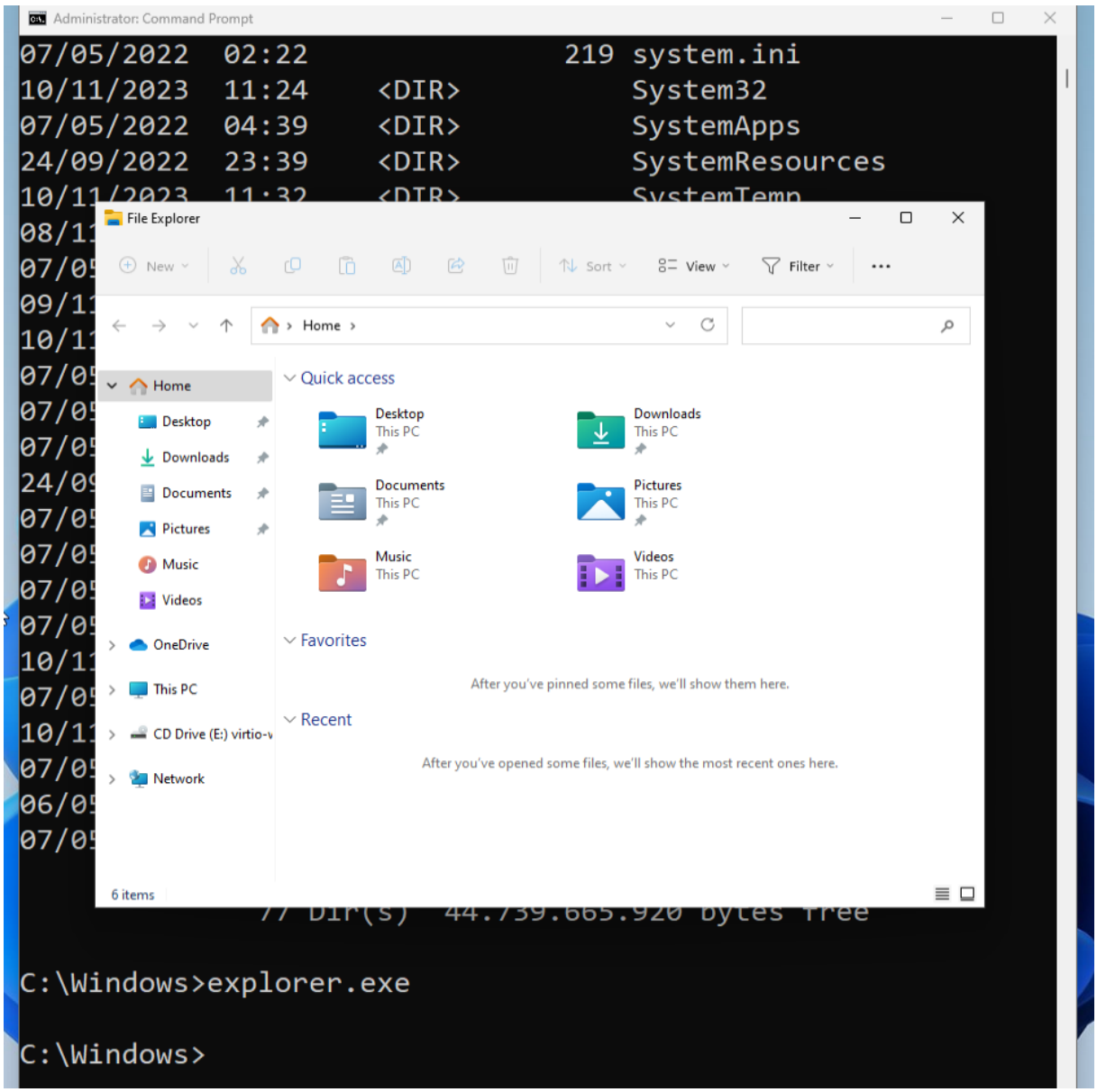


Figure 11: Rodando o Windows Explorer pelo terminal

Tente também rodar o comando `python`. Se você não tiver Python instalado, o comando não vai ser reconhecido e você será levado para a instalação de Python pela Microsoft Store:

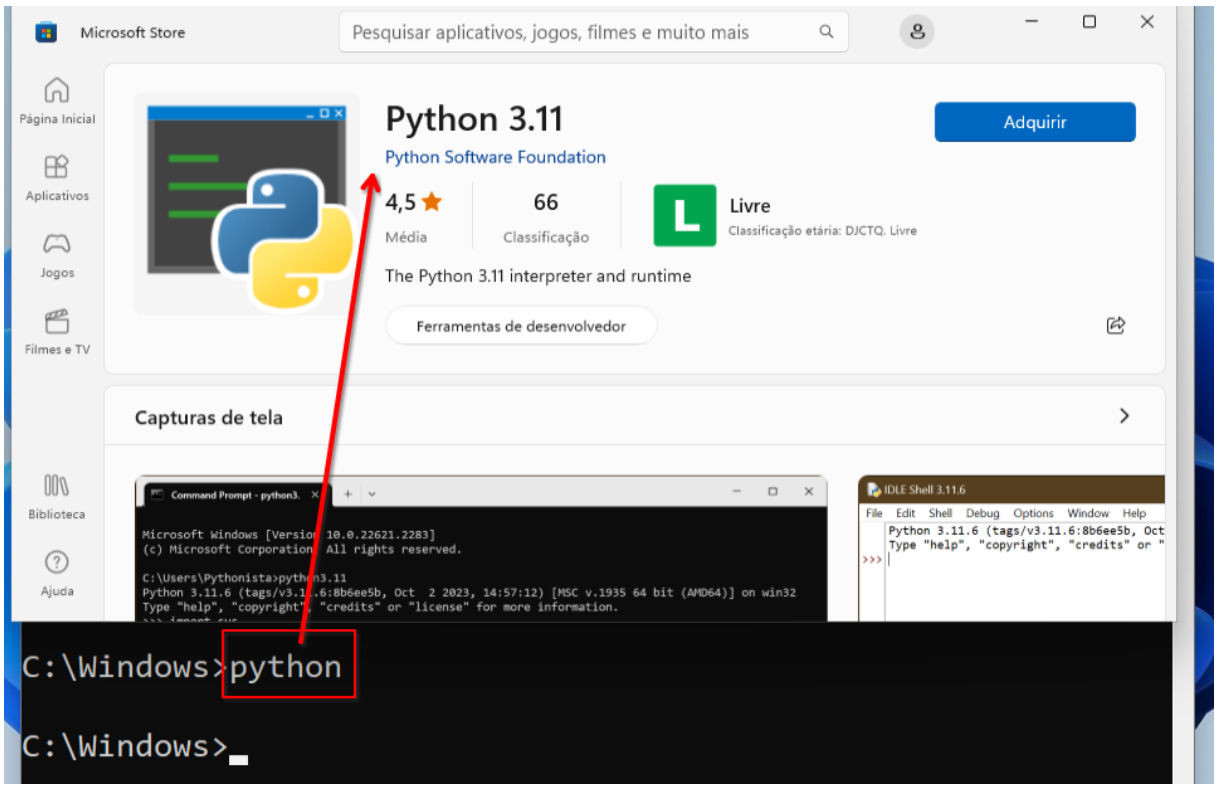


Figure 12: Tentando rodar Python sem estar instalado

Mas **não instale** Python por lá - mais para a frente do curso, veremos como instalar pelo instalador padrão de Python.

Variáveis de ambiente

Em todos os sistemas operacionais, existe o conceito de **variáveis de ambiente**. São valores que ajudam a configurar seu sistema operacional.

Em Windows, podemos acessá-las buscando por “ambiente” ou “environment” (dependendo da linguagem do seu sistema):

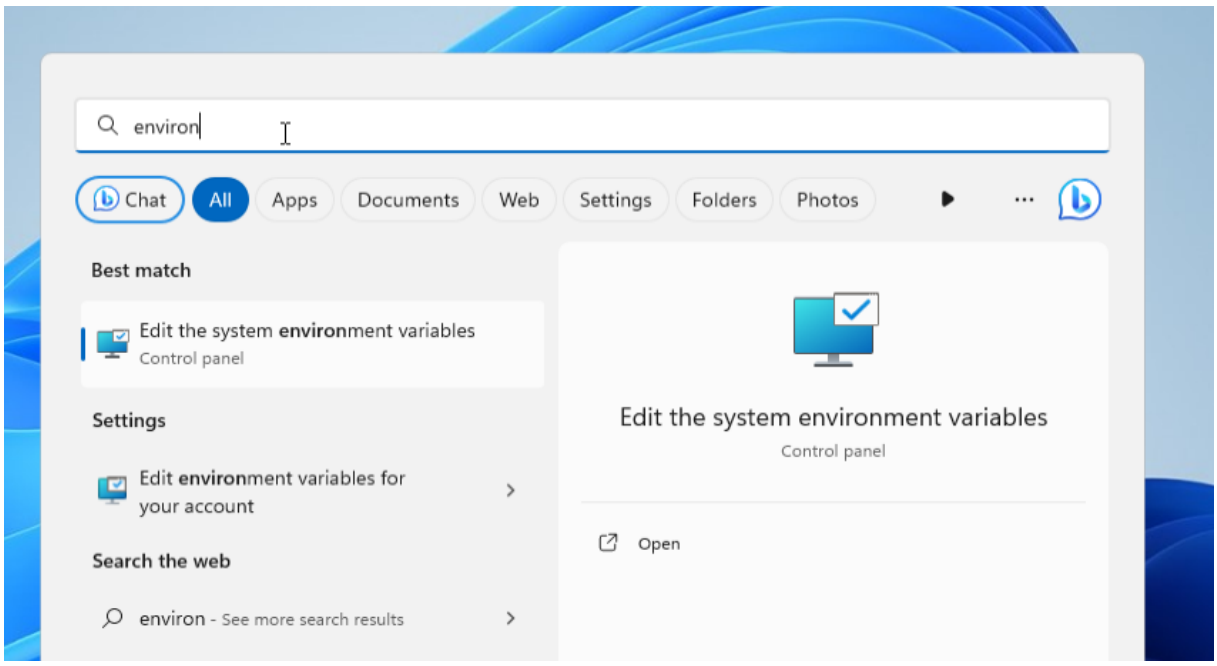


Figure 13: Acessando as variáveis de ambiente

Em seguida, clicamos no botão de “Variáveis de Ambiente” ou “Environment Variables” na janela que abre:

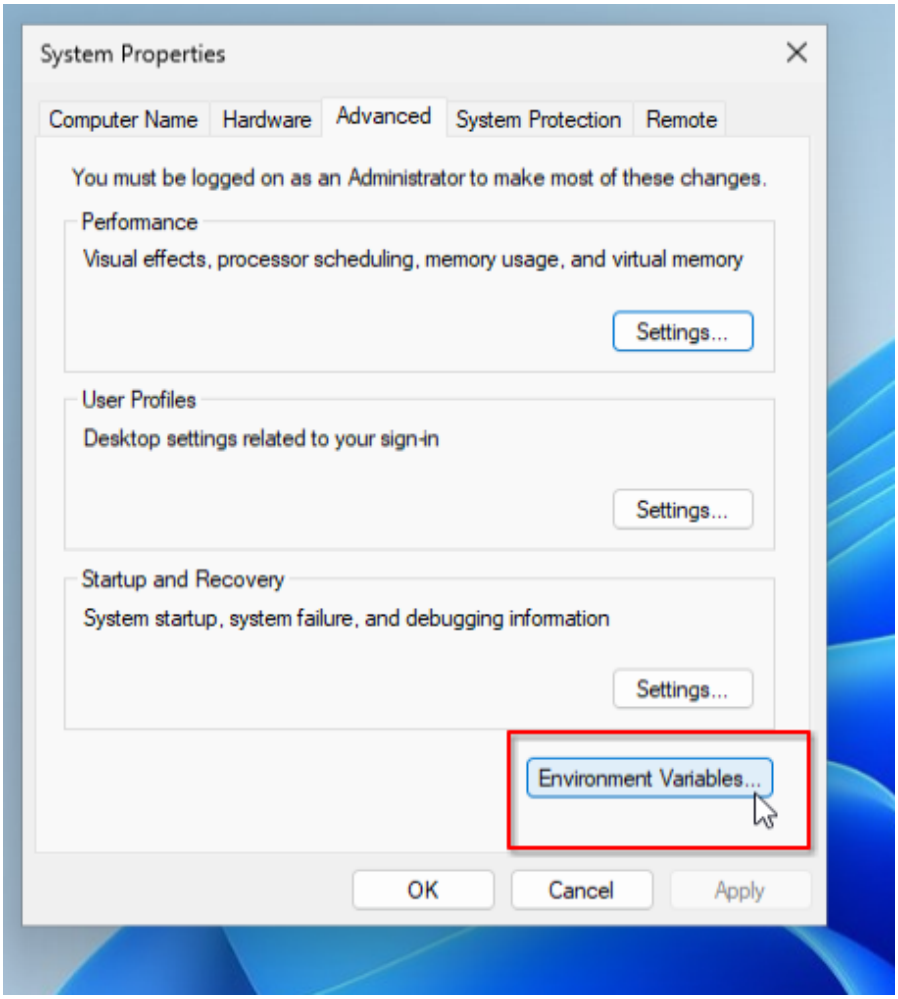


Figure 14: Acessando as variáveis de ambiente

A janela exibe as variáveis definidas para o usuário, e para o sistema como um todo.

Note a existência da variável `Path` (ou PATH). Esta variável é composta por uma lista de pastas separadas por ponto-e-vírgula. Estas pastas indicam os locais onde o Windows busca por programas para executar. Note que a pasta `C:\Windows` está lá dentro:

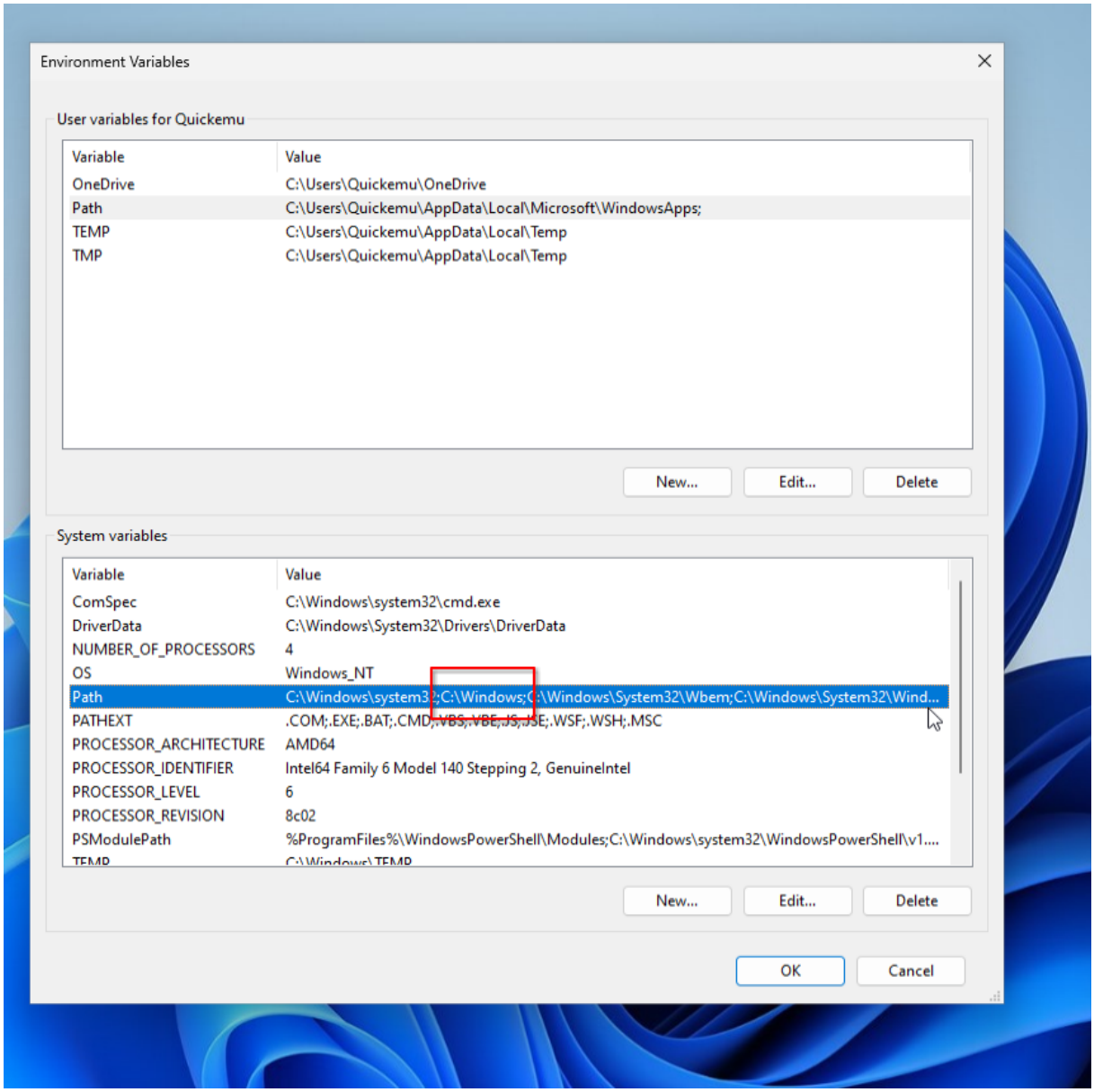


Figure 15: Vendo os valores das variáveis de ambiente

Isto significa que podemos executar o Windows Explorer (e qualquer outro programa na pasta `C:\Windows`) de qualquer lugar! Abra um novo CMD e digite `explorer.exe` para se certificar disso:

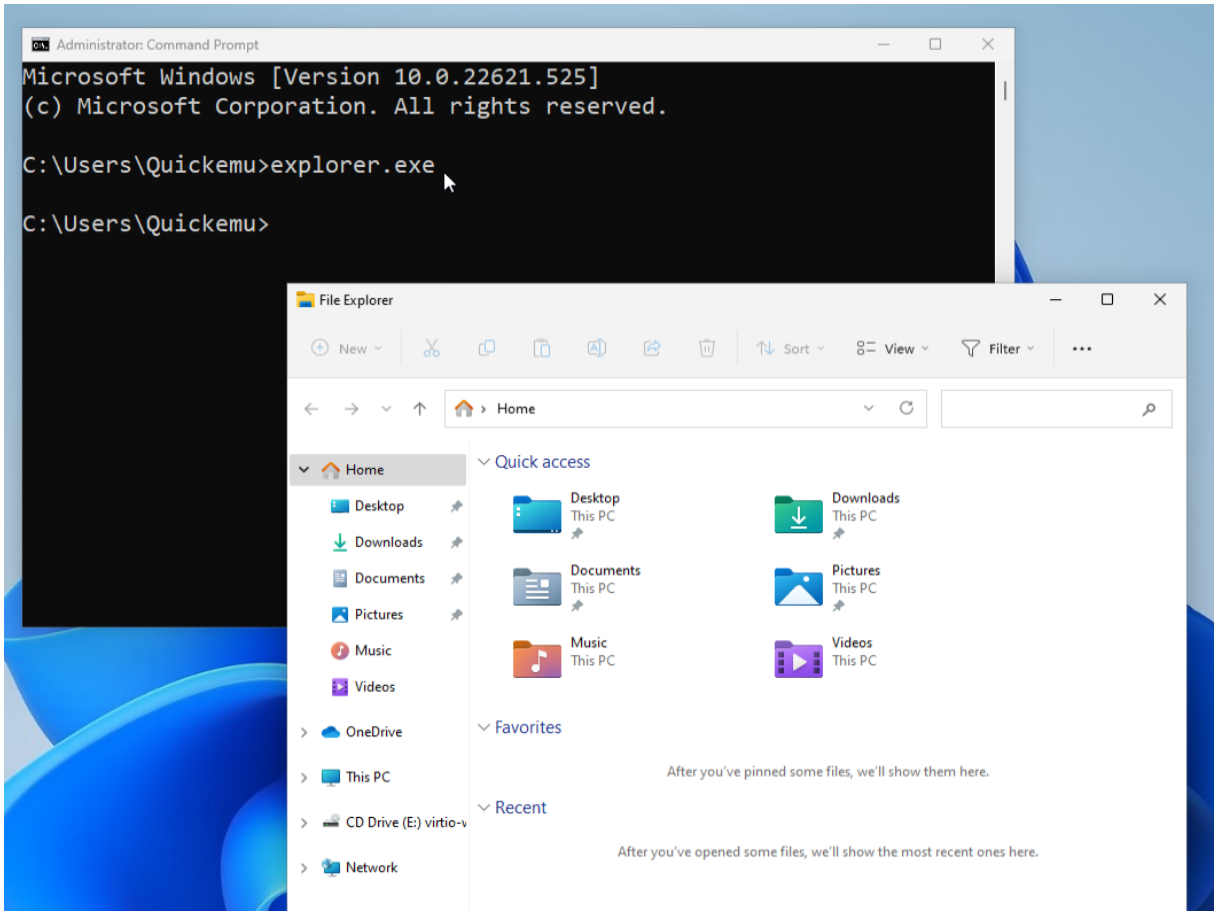


Figure 16: Executando programas que estão no `Path`

Esta informação é importante para que consigamos configurar Python no nosso computador: **se instalarmos Python mas não o adicionarmos a uma pasta conhecida pela variável de ambiente `Path`, então o Windows não encontrará o programa e não conseguirá executá-lo!**

No caso de um programa não ser encontrado (seja por não estar instalado, ou simplesmente por não estar no `Path`), o Windows retorna sempre a mensagem de erro `xxx não é reconhecido como um comando interno ou externo, um programa operável ou um arquivo em lotes`, conforme tela abaixo:

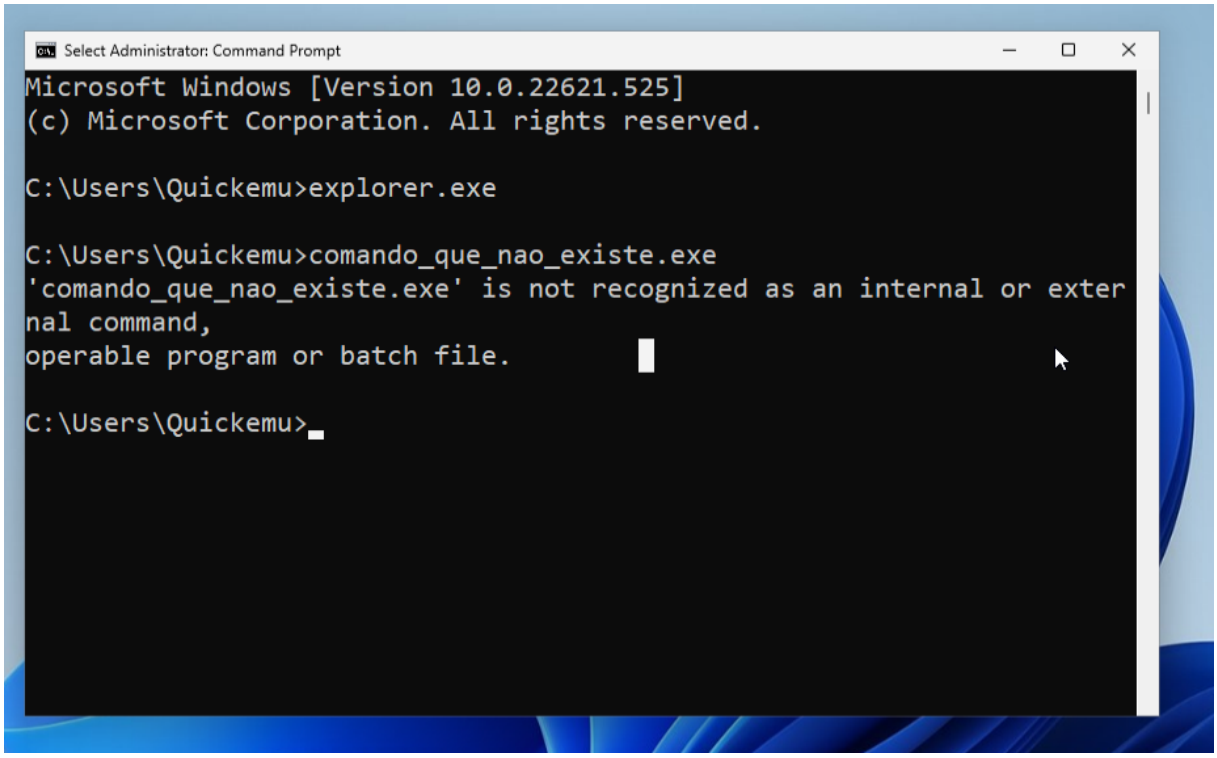


Figure 17: Executando programas fora do `Path`

04 Instalando Python no Windows

Instalação do Python

Vamos instalar Python a partir do instalador oficial. Para isso, acesse o site python.org e use a opção de Download no cabeçalho (ele já deve identificar seu sistema operacional e baixar a versão mais recente de Python):

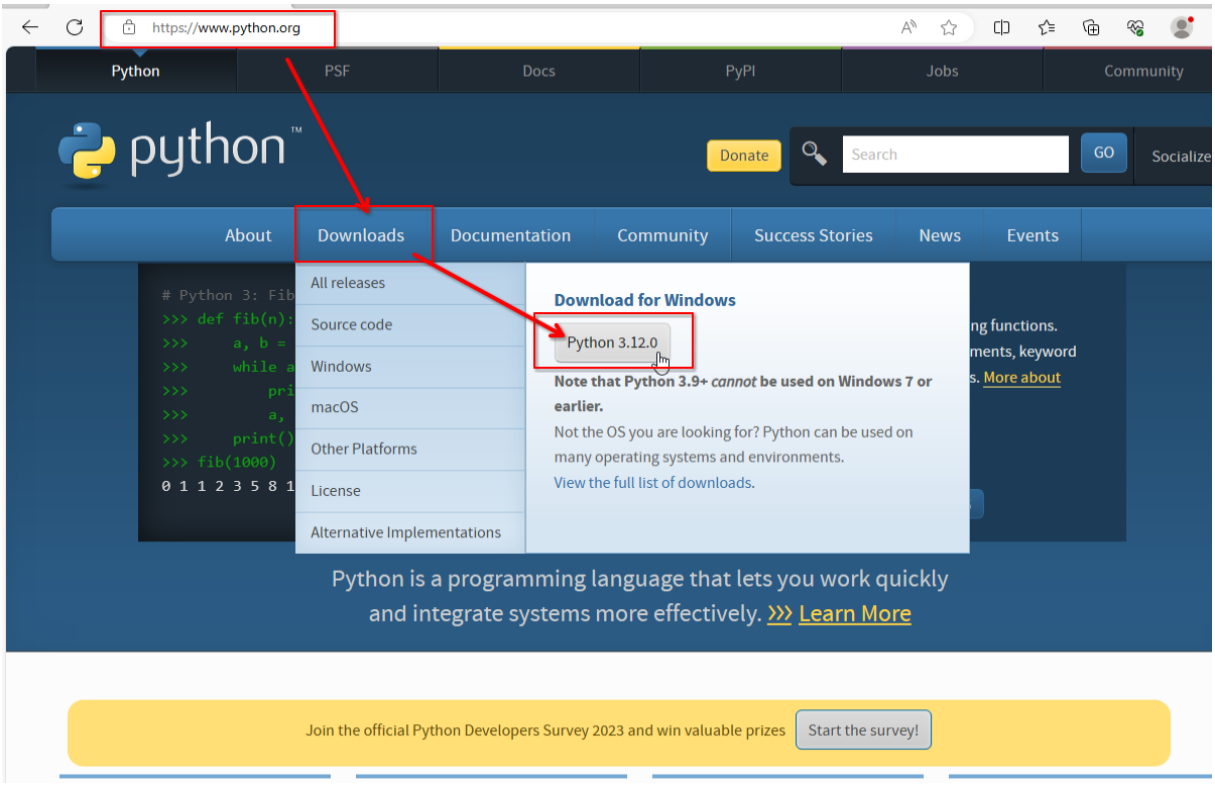


Figure 18: Baixando Python do site oficial

Após baixar o instalador, execute-o para chegar na tela abaixo:

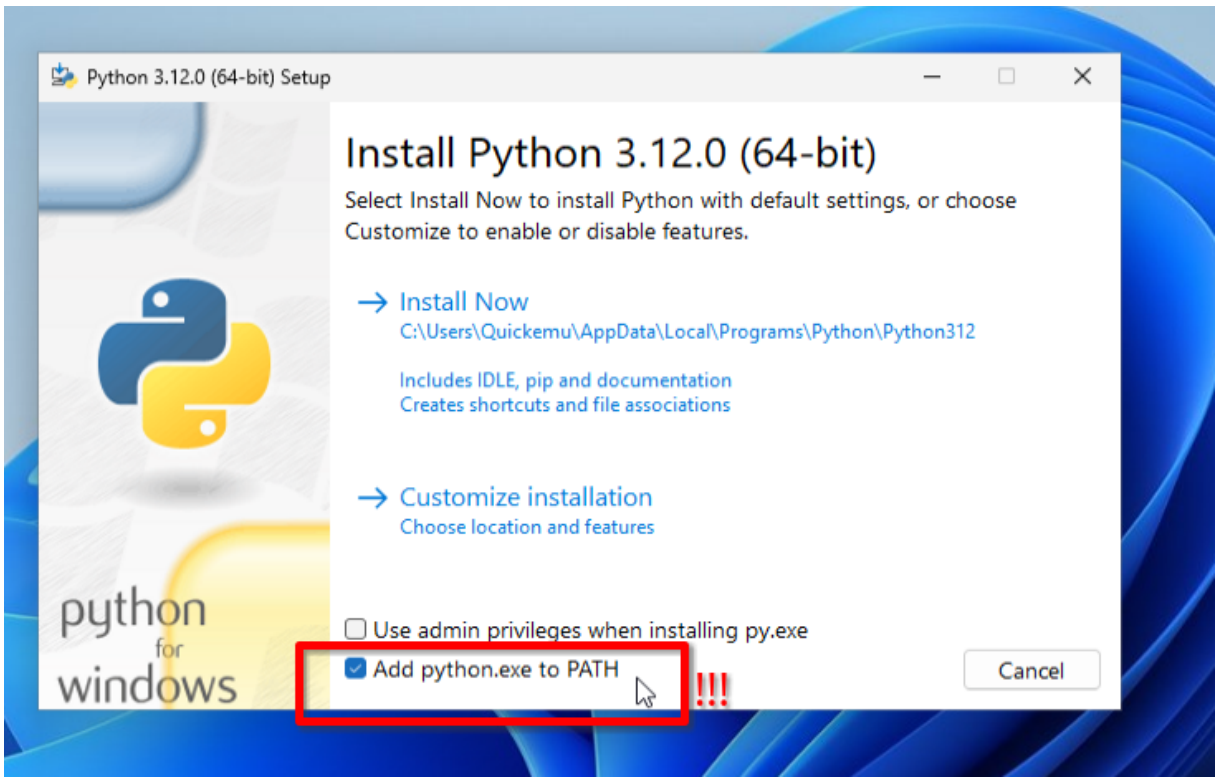


Figure 19: Executando o instalador de Python

Atenção: note que há uma opção para **adicionar Python ao PATH**. Marque a caixinha na parte de baixo do instalador para garantir que seu Windows encontrará sua instalação de Python!

Feito o passo acima, clique em `Instalar Agora` e espere o instalador finalizar.

Na tela final do instalador, há um botão com a opção para `Desabilitar o limite de comprimento de caminhos` do Windows. Clique neste botão para não ter problemas com caminhos de arquivos muito compridos (pode acontecer na sua instalação de Python).

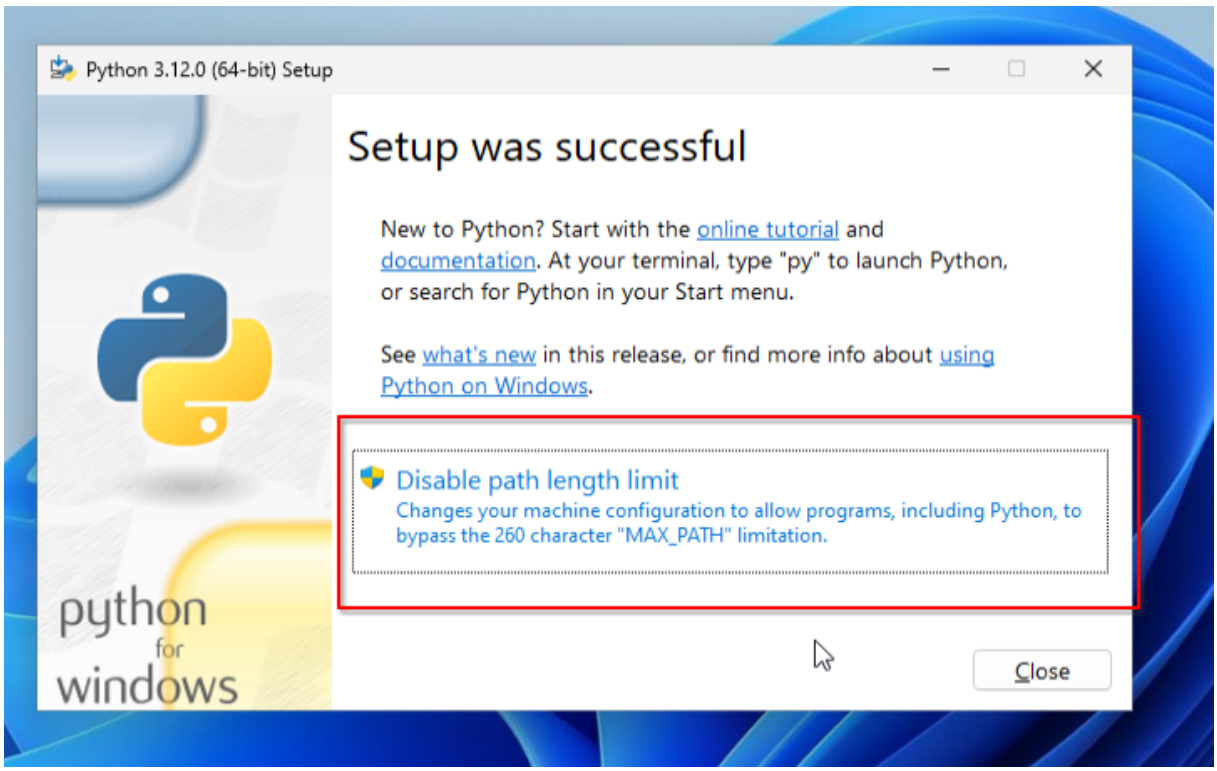


Figure 20: Tela final do instalador de Python

Pronto, seu Python está instalado! Caso você precise repetir ou modificar algum dos passos acima, é possível executar o instalador novamente para modificar (ou até mesmo desinstalar) sua instalação de Python:

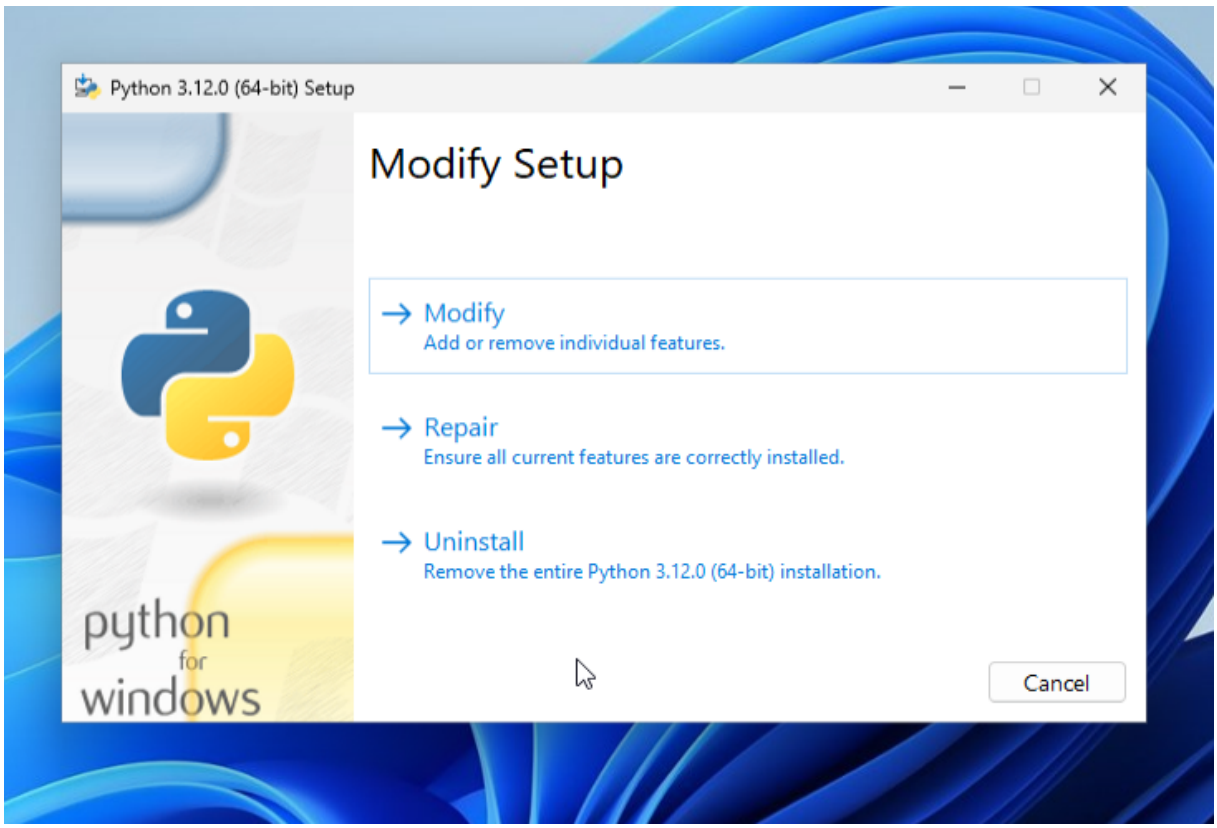


Figure 21: Modificando a instalação de Python

Verificando sua instalação

Se você marcou a opção de adicionar Python ao PATH, então sua instalação deve ser detectável pelo terminal.

Confirme isto entrando no CMD e digitando `python`. Se o console de Python aparecer, então a instalação funcionou! Você pode digitar comandos de Python para testá-lo, ou usar o comando `exit()` para voltar ao CMD.

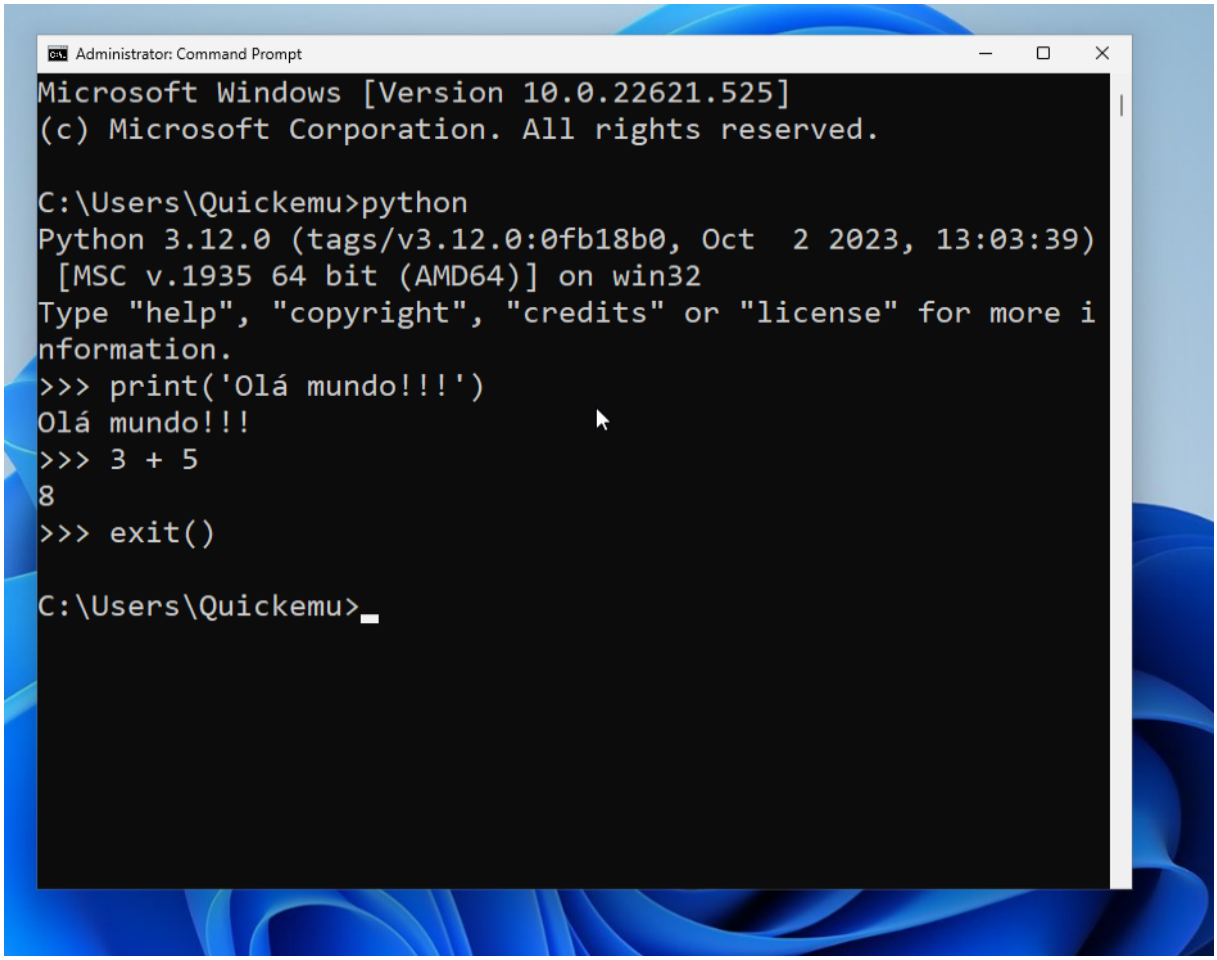


Figure 22: Testando a instalação de Python

Onde Python foi instalado?

Você pode usar o **comando `where python`** para checar a que executável o comando `python` está vinculado (lembre-se de executar este comando da interface do CMD, e não do console de Python):

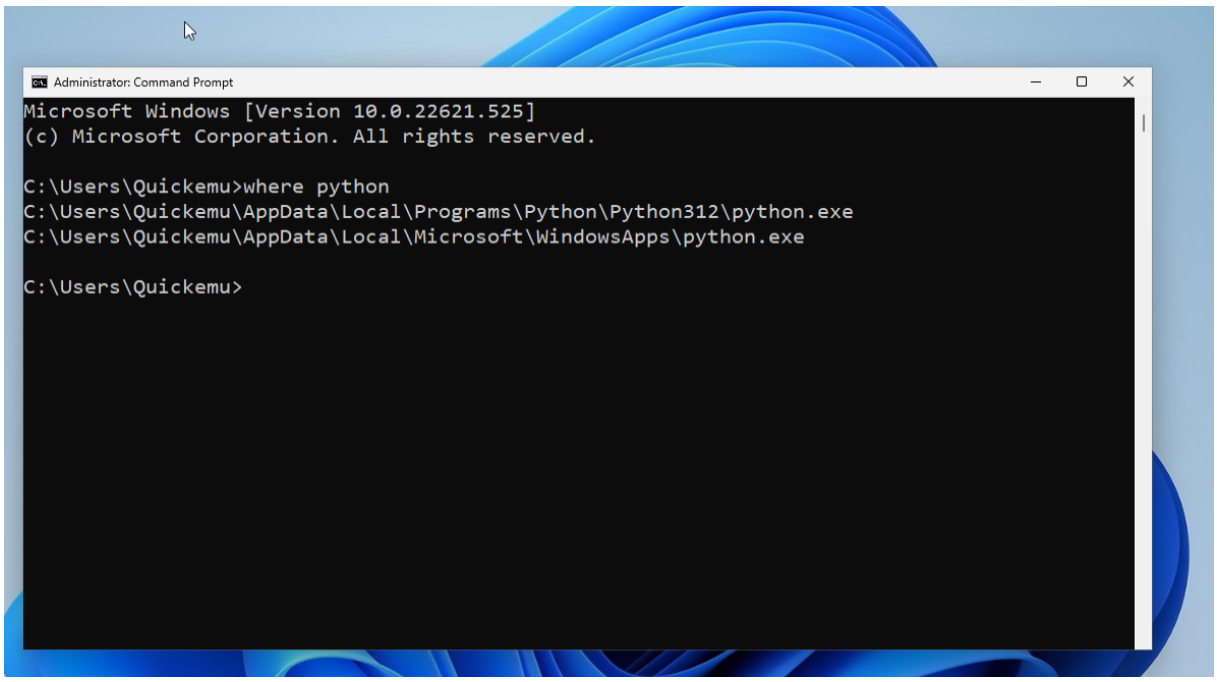


Figure 23: Comando `dir`

No caso da imagem acima, note que surgiram dois caminhos, pra dois executáveis distintos. O primeiro executável está no caminho onde acabamos de instalar Python. O segundo na realidade não se trata de uma instalação de Python, mas de um executável que abre a Microsoft Store (é o que acontece quando rodamos `python` sem ter feito sua instalação antes).

Note que o primeiro caminho retornado por `where python` tem preferência sobre os demais. Em outras palavras, o comando `python` irá sempre executar o Python que acabamos de instalar. Se você tiver múltiplas instalações de Python, fique atento a qual Python você se refere com este comando no CMD!

Instalando dependências

A instalação de Python inclui também o comando `pip`, que é usado para instalar bibliotecas externas de Python como o Pandas.

Podemos confirmar que o `pip` foi instalado e está no `Path` usando `where pip`:

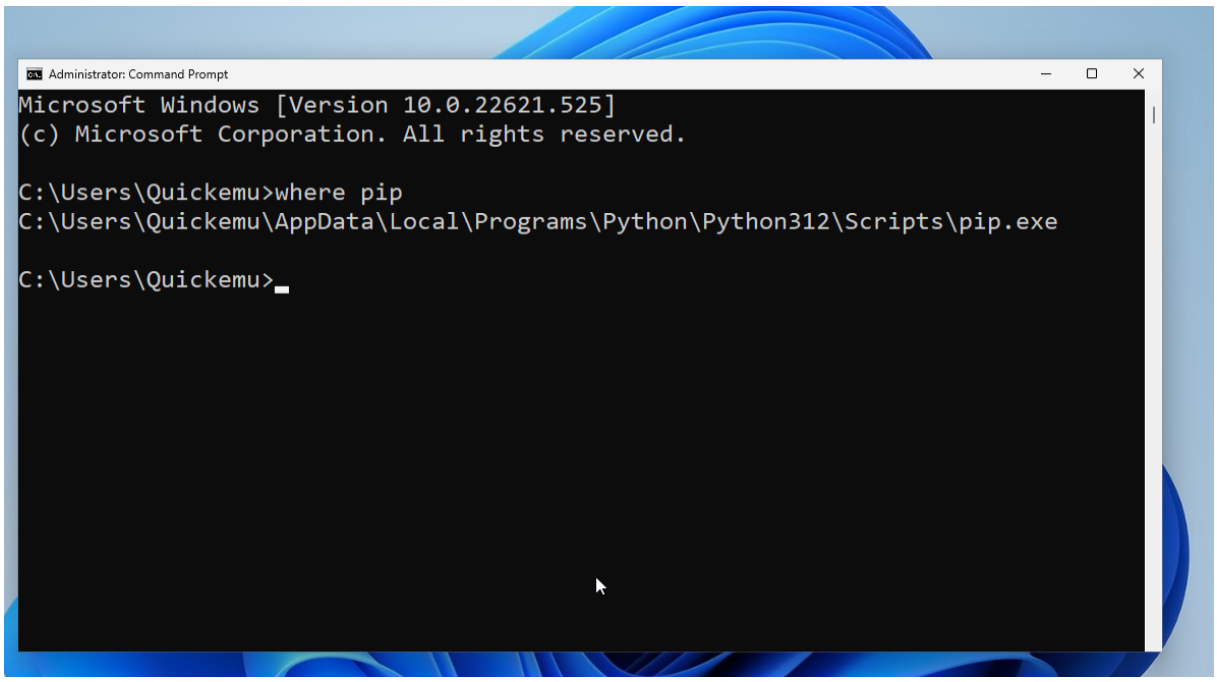


Figure 24: Verificando a instalação do `pip`

Usamos o comando `pip install xxx` para instalarmos a biblioteca `xxx`. Teste a instalação do Pandas com `pip install pandas`, e espere até aparecer a mensagem de que foi instalado com sucesso:

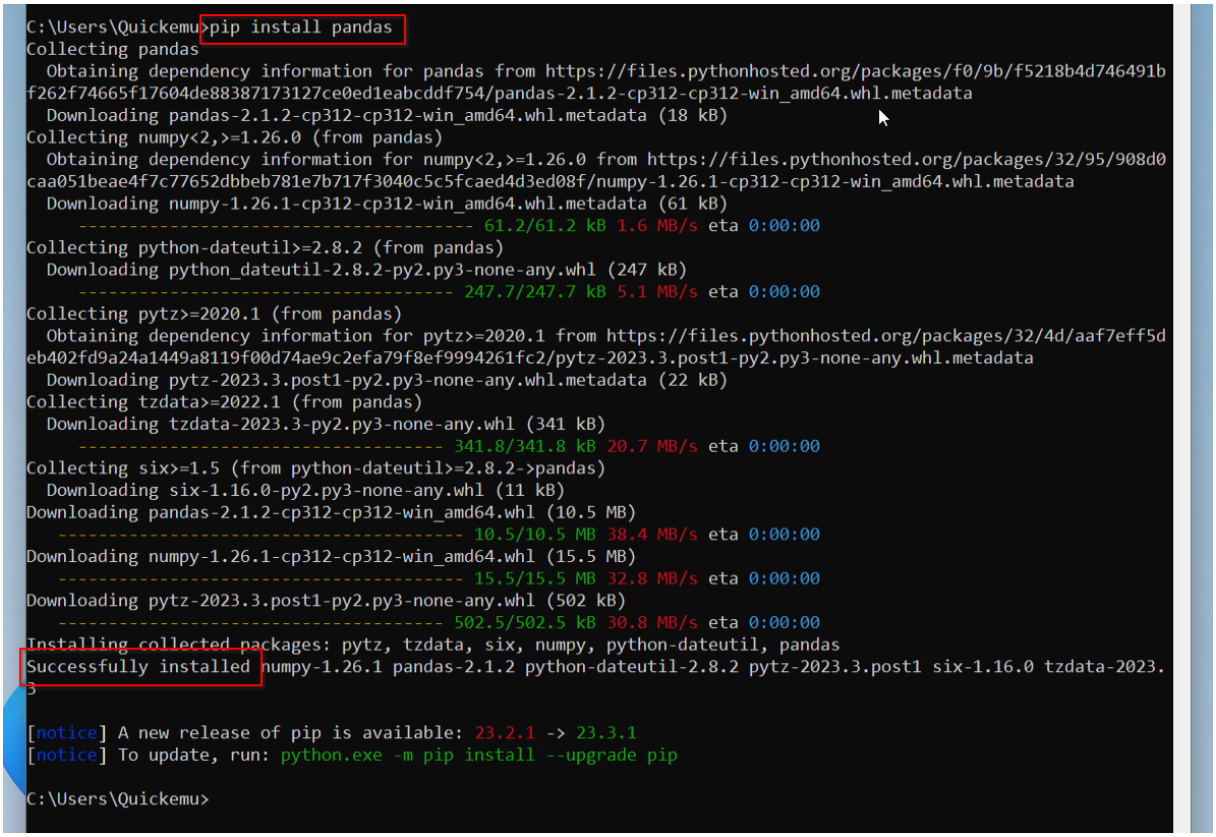


Figure 25: Instalando Pandas com `pip`

Se quisermos garantir que estamos rodando o `pip` de uma instalação específica de Python, podemos também executá-lo a partir do comando `python`, da seguinte forma: `python -m pip install xxx`. Abaixo, fazemos isto para instalar a biblioteca Matplotlib:

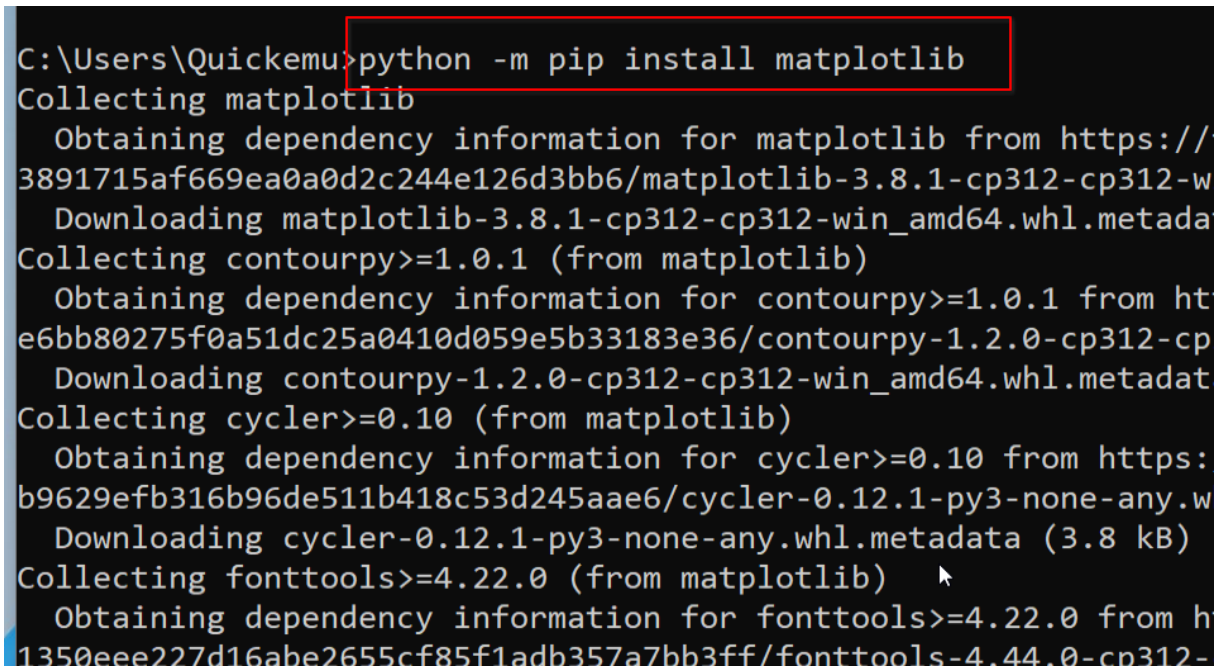


Figure 26: Instalando Matplotlib com `pip`

E para finalizar a verificação, podemos entrar no console de Python e importar as bibliotecas recém instaladas:

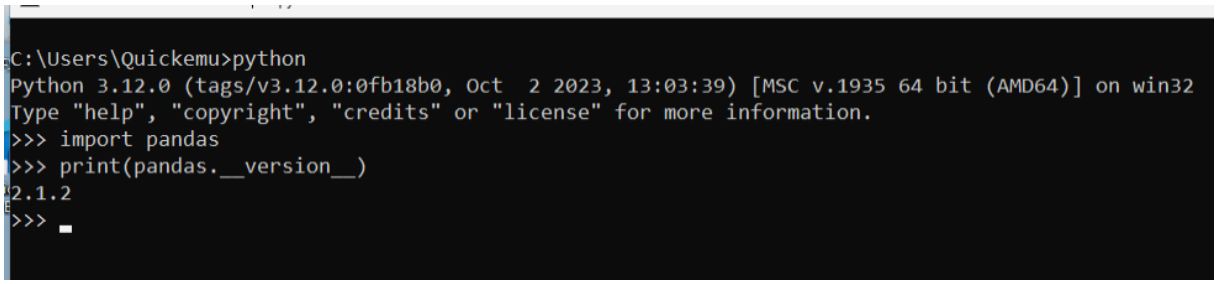


Figure 27: Verificando a instalação do Pandas

Rodando scripts de Python

Além de executar o console de Python, podemos usar o CMD também para rodar um script de Python.

Para isto, precisamos criar um **arquivo de texto simples** com a extensão `.py`, como por exemplo `meu_script.py`.

Há diversos programas chamados **editores de texto** que podem fazer isso. Muitos deles são bastante poderosos, como o VS Code, mas o próprio bloco de notas do Windows já quebra o galho.

Dentro do arquivo, incluímos todo o código de Python que queremos que seja executado. Então, pelo CMD, digitamos o comando `python` seguido do caminho até o script, ex: `python caminho\para\meu_script.py`.

Se você preferir, pode também ir até a pasta onde está o script (usando o comando `cd`) e aí digitar simplesmente `python nome_do_script.py`. Veja os dois exemplos no CMD da imagem abaixo:

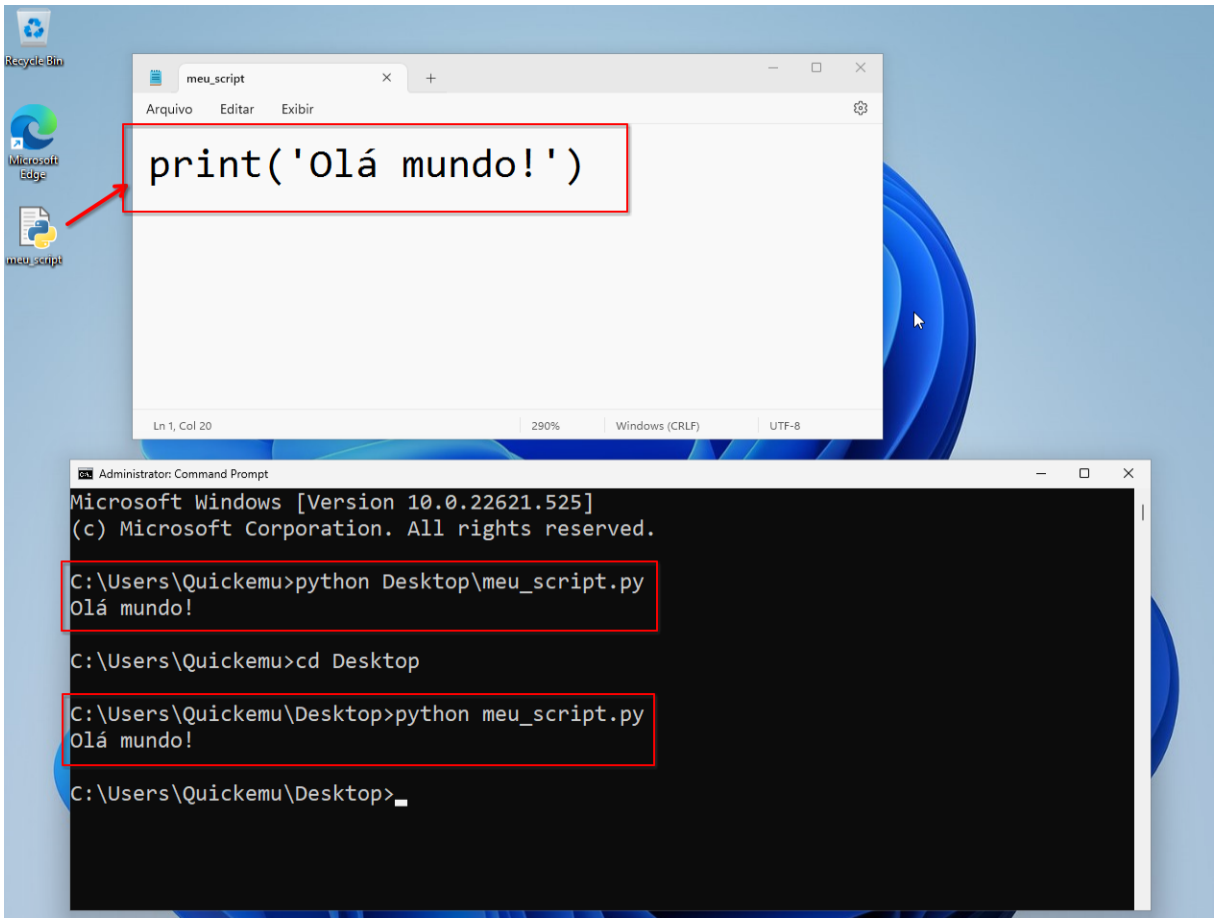


Figure 28: Exemplo de execução de script

05. O terminal do Mac

O terminal do Mac já vem instalado por padrão. Ele é acessível buscando pelo programa `Terminal` nos aplicativos instalados:

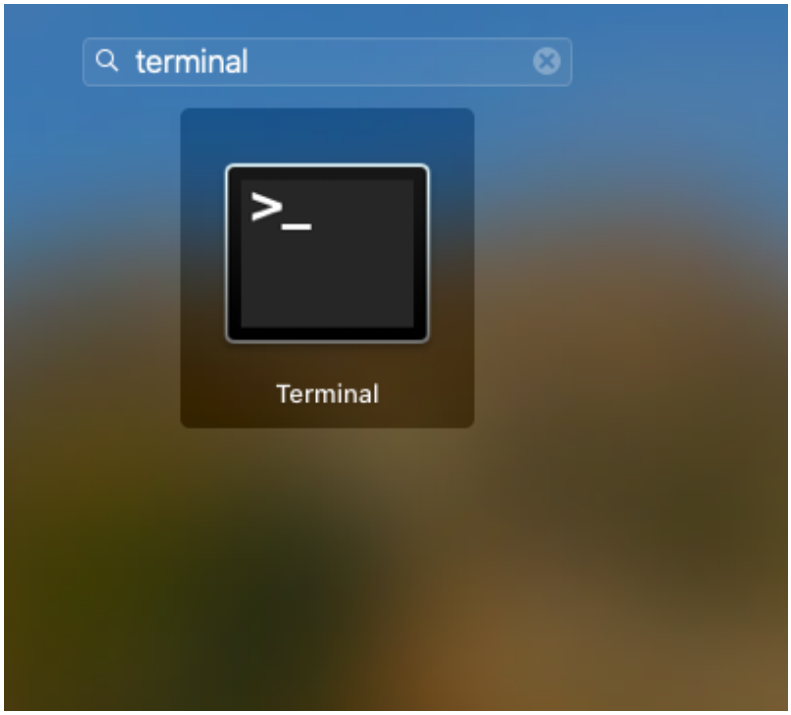


Figure 29: Abrindo o Terminal no Mac

Caminhos em sistemas UNIX

Como falamos anteriormente, o Mac, assim como o Linux, utiliza como base do seu sistema operacional o UNIX. Portanto, os comandos no terminal são diferentes que em Windows.

Assim como no Windows, o terminal de Mac executa a partir da sua pasta do usuário (em geral, o caminho `/Usuários/seu-nome-de-usuario`). Note que o caminho em sistemas UNIX são diferentes de caminhos no Windows em 2 pontos:

- Utilizam barras para a frente (`/`), e não barras para trás (`\`), para delimitar as pastas.
- A pasta raiz do sistema é simplesmente a pasta barra (`/`), e não o `C:\`.

Comandos básicos

Você pode usar o **comando `ls`** (de “listar”) para exibir as pastas e arquivos:

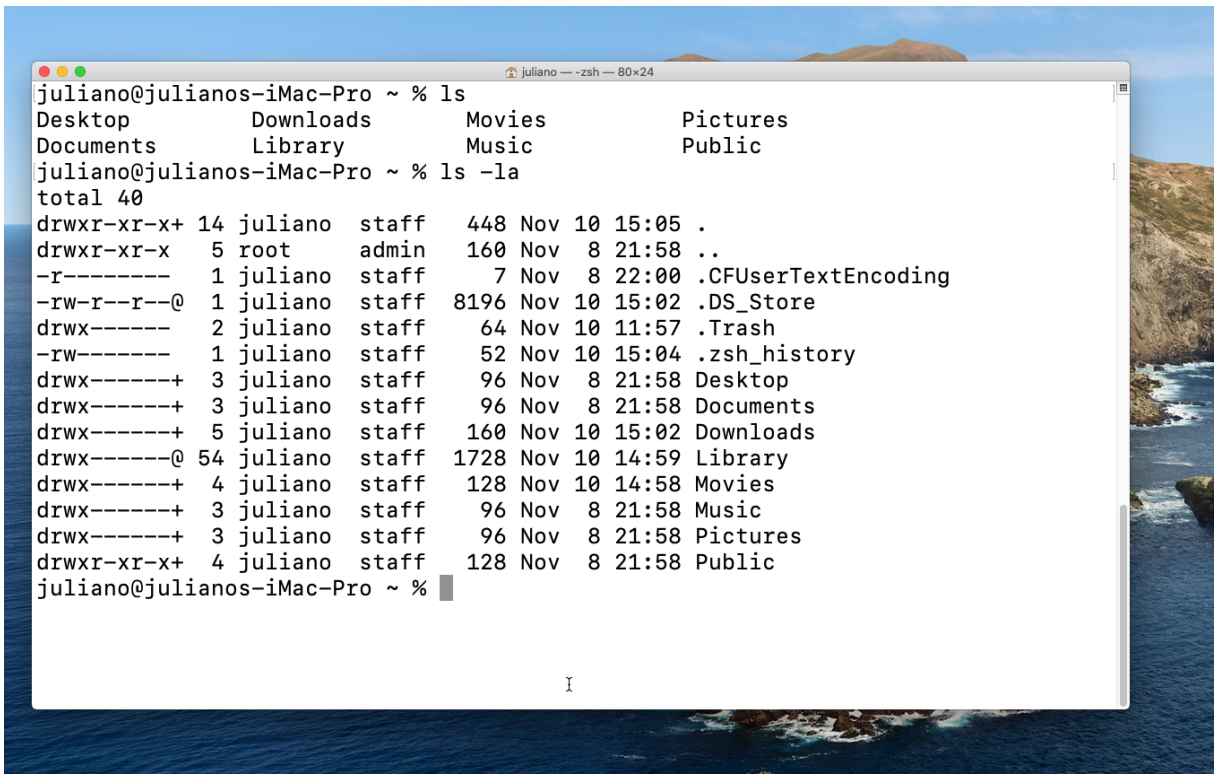


Figure 30: Comando `ls`

Note que na imagem acima executamos também o **comando `ls -la`**. Esta porção `-la` adicionada ao final do comando são **flags** que modificam o seu comportamento.

A flag `-l` faz com que os arquivos apareçam em uma lista vertical, similar ao comando `dir` de Windows. Já a flag `-a` faz com que pastas e arquivos ocultos (que em sistema UNIX, são aqueles com nome iniciado por ponto) seja exibidos também. Usar `-la` é apenas uma forma prática de combinar as flags `-l` e `-a`.

Para mudar de diretório, usamos o comando `cd` (do inglês *change directory*), que funciona da mesma forma como em Windows:

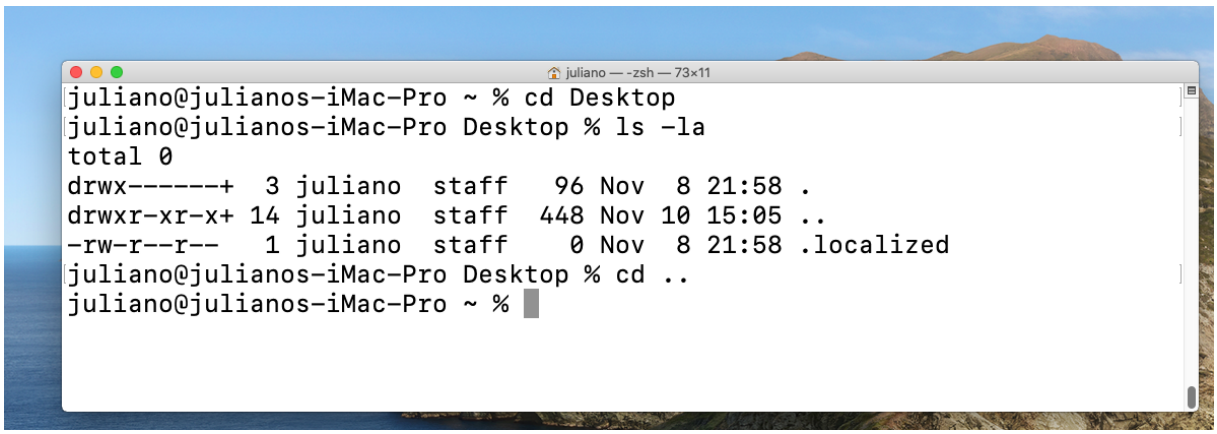


Figure 31: Comando `cd`

Executáveis e as variáveis de ambiente

Sistemas UNIX também possuem a variável de ambiente PATH. Assim como em Windows, essa variável determinar os locais onde o sistema operacional procura por programas para executar.

Podemos observar o valor da variável PATH no próprio terminal usando o comando `echo $PATH`:

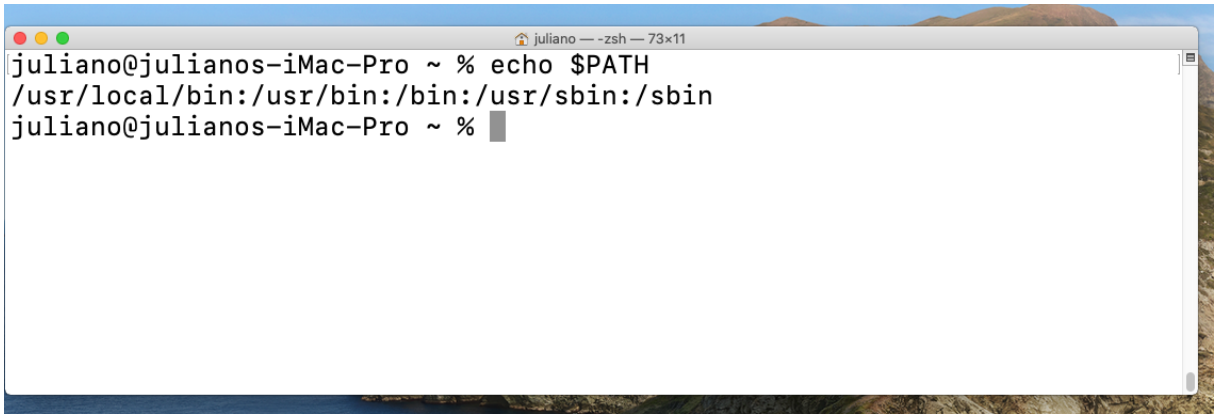


Figure 32: A variável de ambiente PATH

O resultado é uma lista de pastas do sistema, separadas por `:`.

Mesmo sem saber, já utilizamos o PATH ao usarmos o comando `ls`, já que ele é considerado um programa, que já vem instalado em sistemas UNIX.

Podemos até confirmar isso usando o **comando `which ls`**, que mostra o caminho onde o programa está instalado (é equivalente ao `where` que vimos no Windows).

Python em Mac

Se usarmos o comando `python` ou `python3` em Mac, possivelmente iniciaremos um interpretador de Python, mesmo sem instalar nada.

No exemplo da imagem abaixo, o comando `python` inicia o Python na versão 2.7, enquanto o comando `python3` redireciona para a instalação dos "Developer Tools" do Mac (os resultados dos comandos podem ser diferentes no seu Mac, dependendo da sua versão):

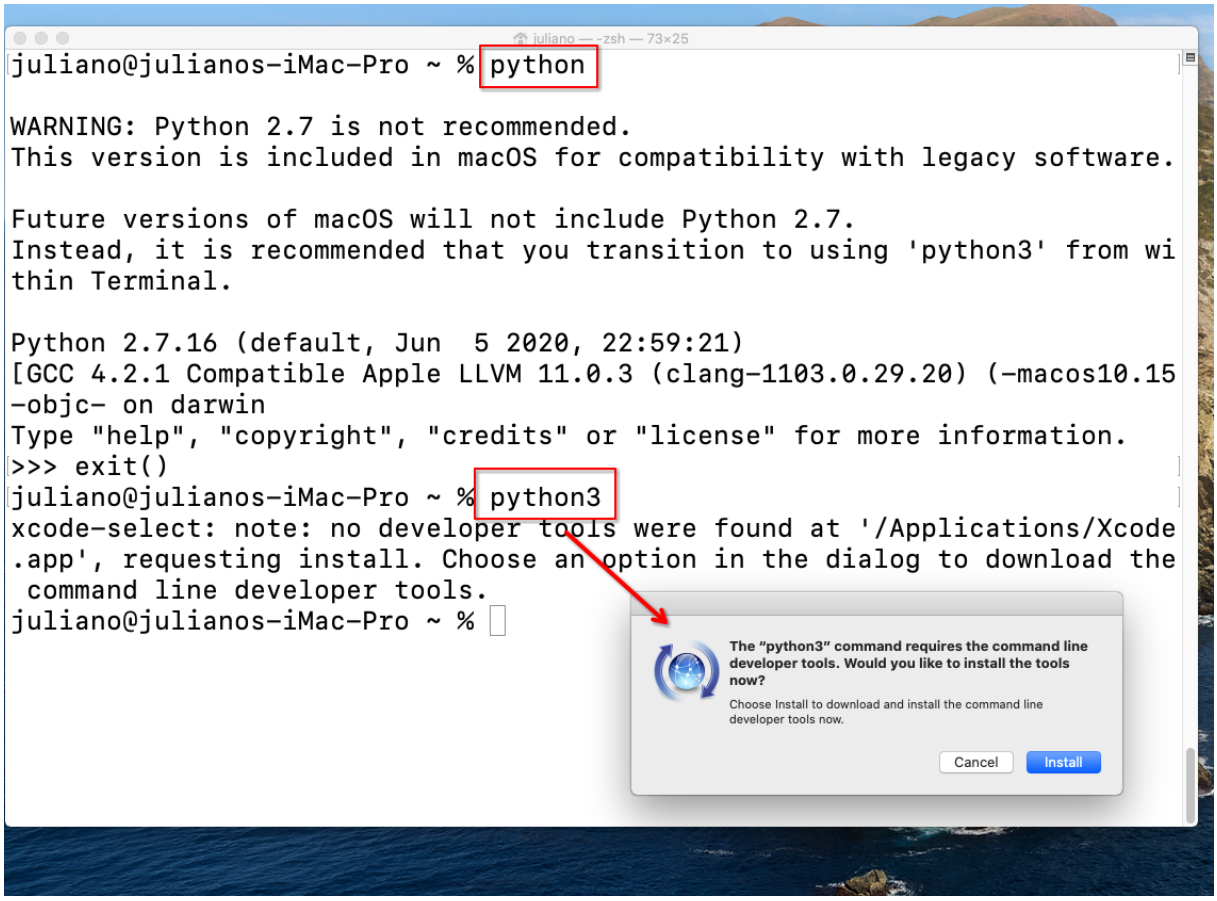


Figure 33: O comando `python` em Mac

Não queremos usar o Python já instalado por 2 motivos:

- A versão 2 do Python foi descontinuada em 2020, e portanto é altamente recomendável **não** utilizá-la hoje em dia.
- Mesmo se você possuir Python 3 previamente instalado no seu sistema, **o ideal é nunca mexer nas instalações de Python do seu sistema**. Isso porque parte do seu próprio sistema operacional pode depender de algum pacote, e fazer instalações, atualizações ou modificações pode

desconfigurá-lo! No lugar disso, vamos instalar Python pela fonte oficial na próxima aula.

06. Instalando Python no Mac

Vamos instalar Python a partir do instalador oficial. Para isso, acesse o site python.org e use a opção de Download no cabeçalho (ele já deve identificar seu sistema operacional e baixar a versão mais recente de Python):

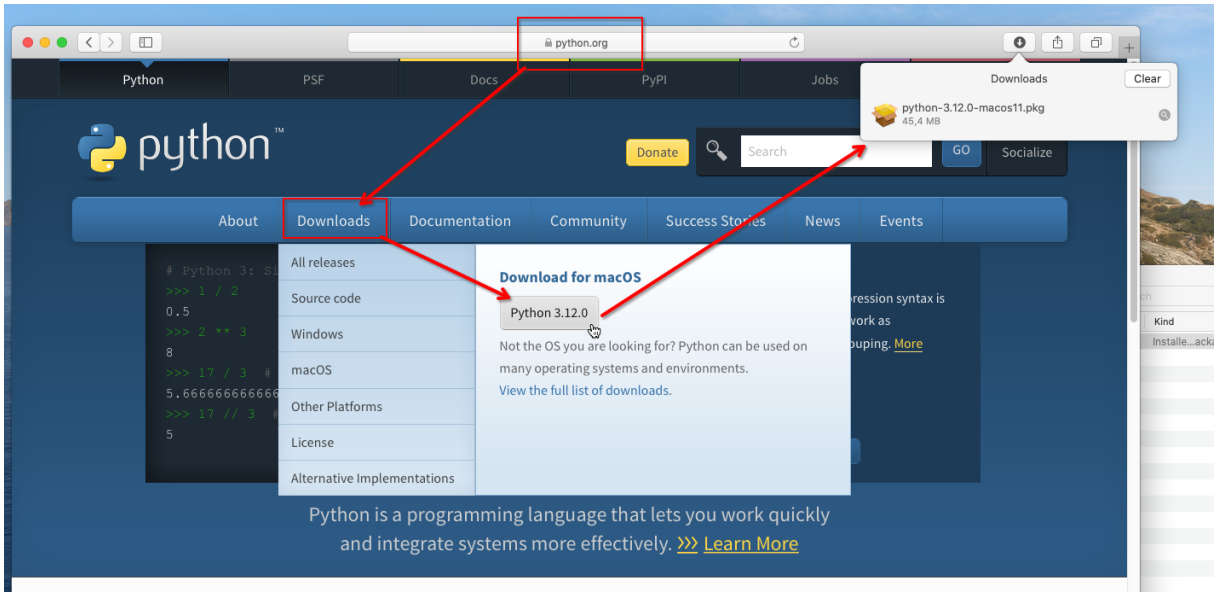


Figure 34: Baixando Python do site oficial

Abra o instalador e siga os seus passos (diferente do instalador de Windows, não há nenhum botão ou opção para marcar):

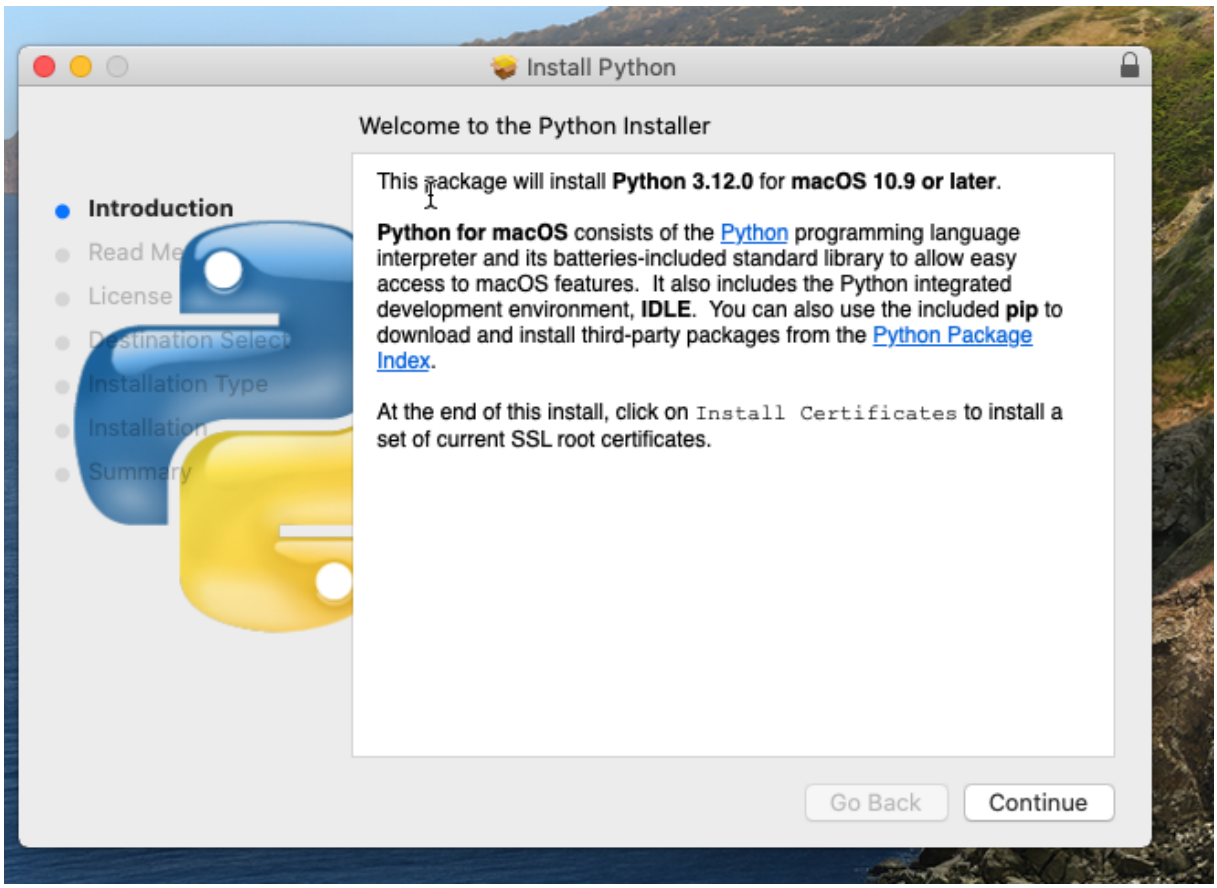


Figure 35: O instalador de Python

Ao final da instalação, você verá um aviso para executar o arquivo `Install Certificates`. Este passo é necessário para que o sua instalação de Python seja compatível com alguns pacotes que acessam a internet.

Para fazer a instalação, basta abrir a pasta da instalação de Python (em `Aplicativos` -> `Python3.xx`, onde `xx` é a versão de Python)...

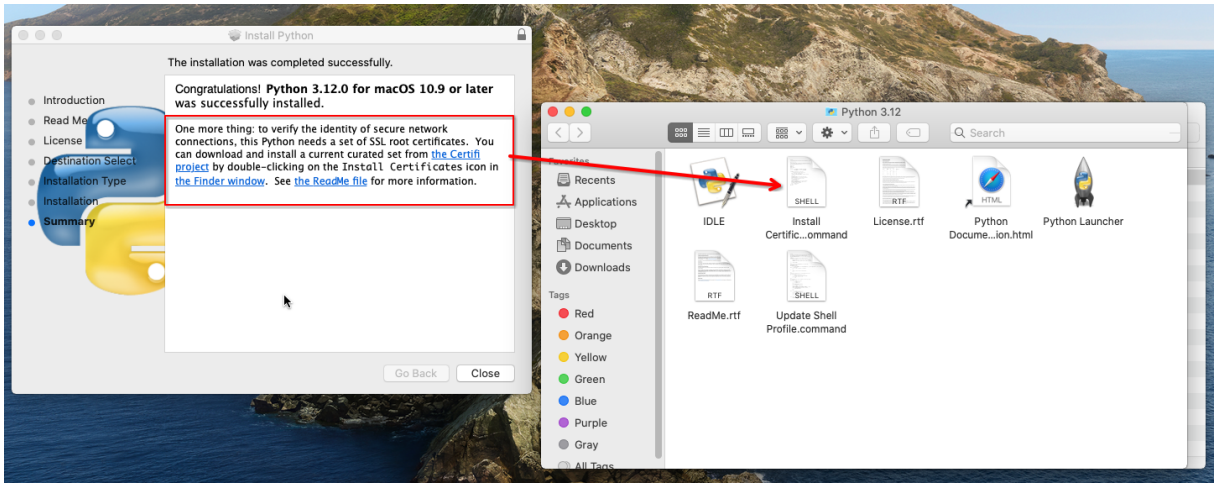


Figure 36: Localizando o `Install Certificates`

… E clicar duas vezes para rodar o arquivo e instalar os certificados:

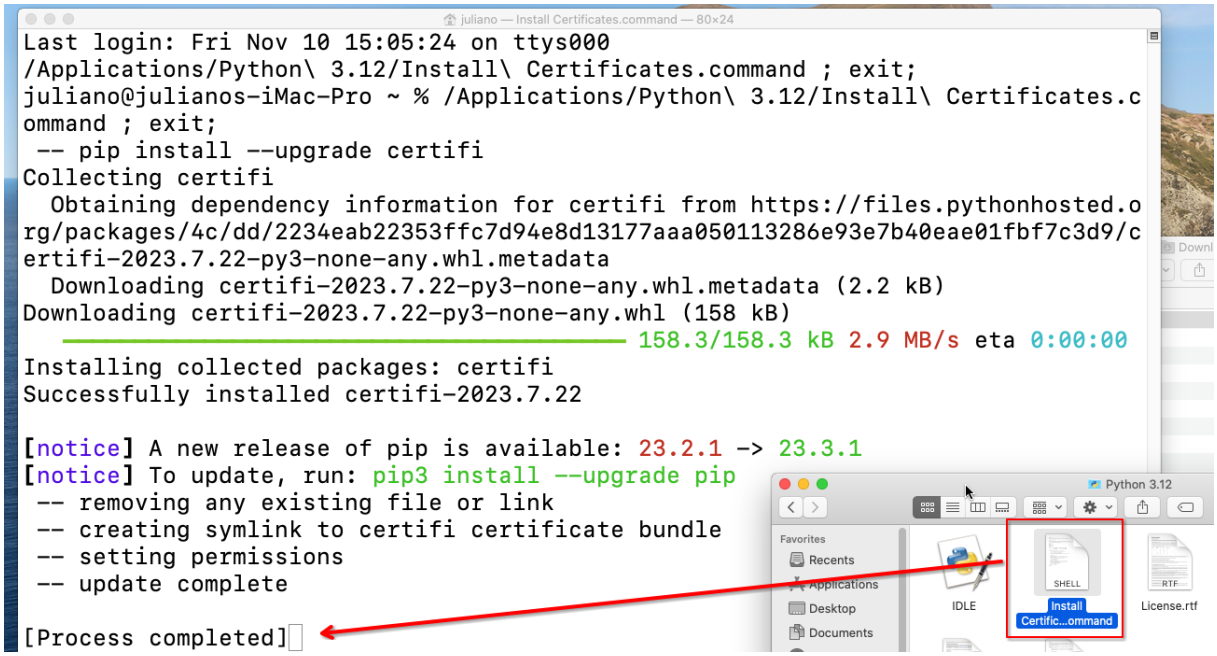


Figure 37: Rodando o `Install Certificates`

Verificando sua instalação

Abra novamente o terminal, digite o comando `python3` (atenção para incluir o 3 no comando) e confira que o console de Python inicia, para a versão que você acabou de instalar.

Nele, você pode inserir comandos de Python. Use o atalho Command + D (ou digite o comando `exit()`) para voltar para o terminal:

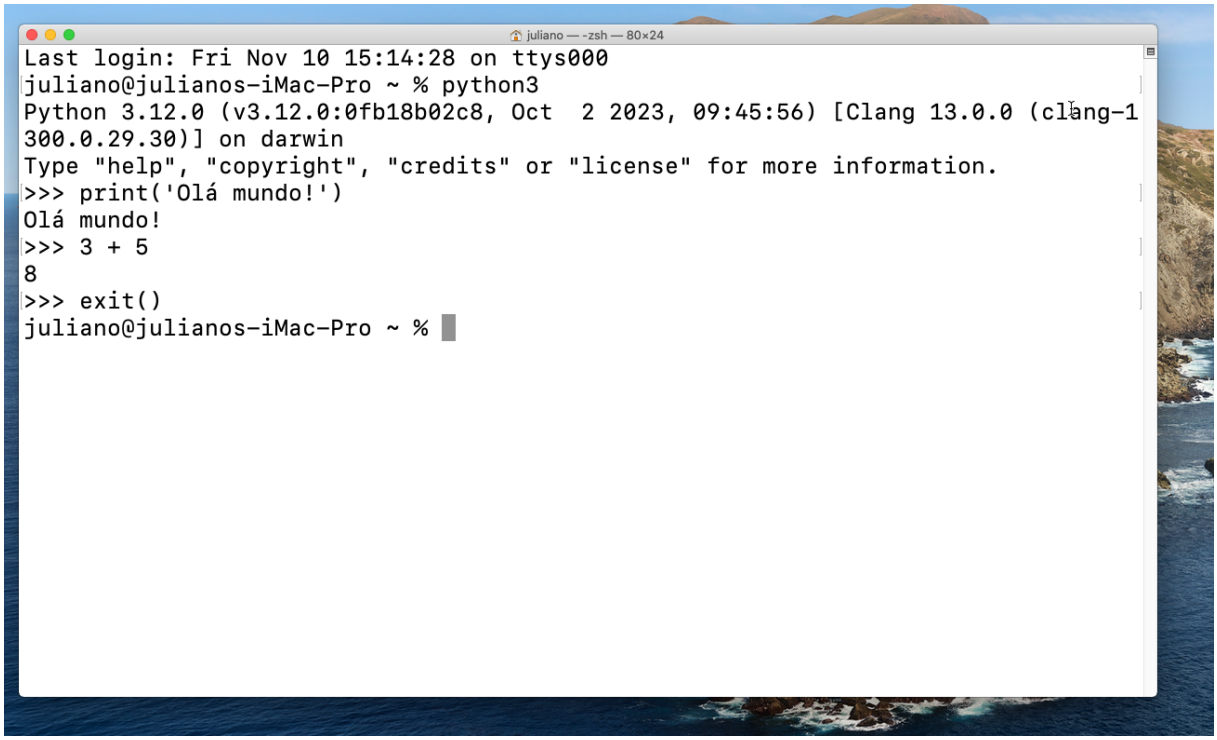


Figure 38: Rodando o console de Python

Onde Python foi instalado?

Use o **comando `which python3`** para checar onde está sua instalação de Python. Também é possível executar `which pip3` para checar onde está a instalação do `pip` (o instalador de pacotes do Python):

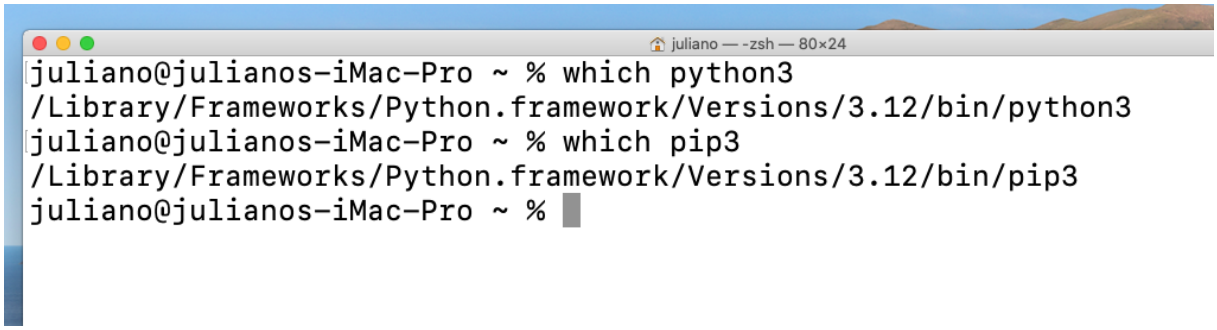


Figure 39: Checando o local de instalação do Python

Instalando dependências

Para instalar dependências, basta rodar `pip3 install` e o nome da biblioteca. No exemplo abaixo, instalamos a biblioteca `pandas` com `pip3 install pandas`:

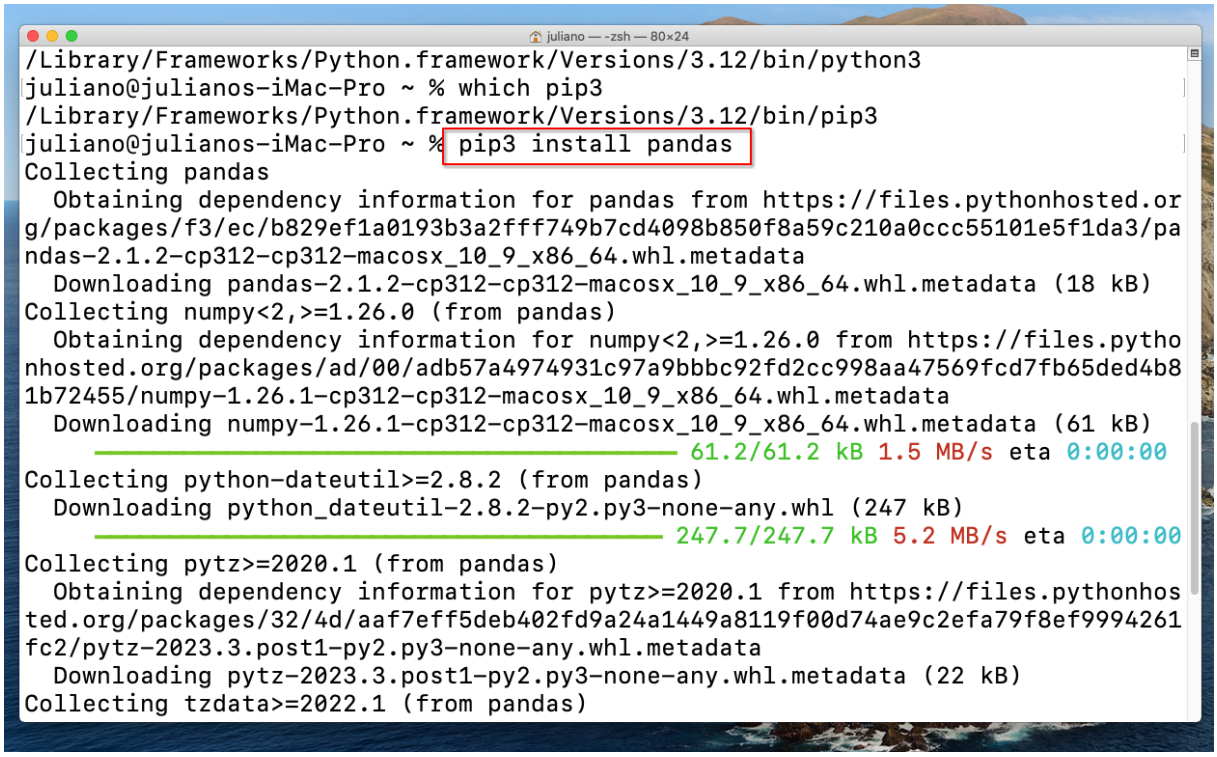


Figure 40: Instalando dependências com `pip3`

Após a instalação, podemos entrar no console de Python e importar as bibliotecas recém instaladas:

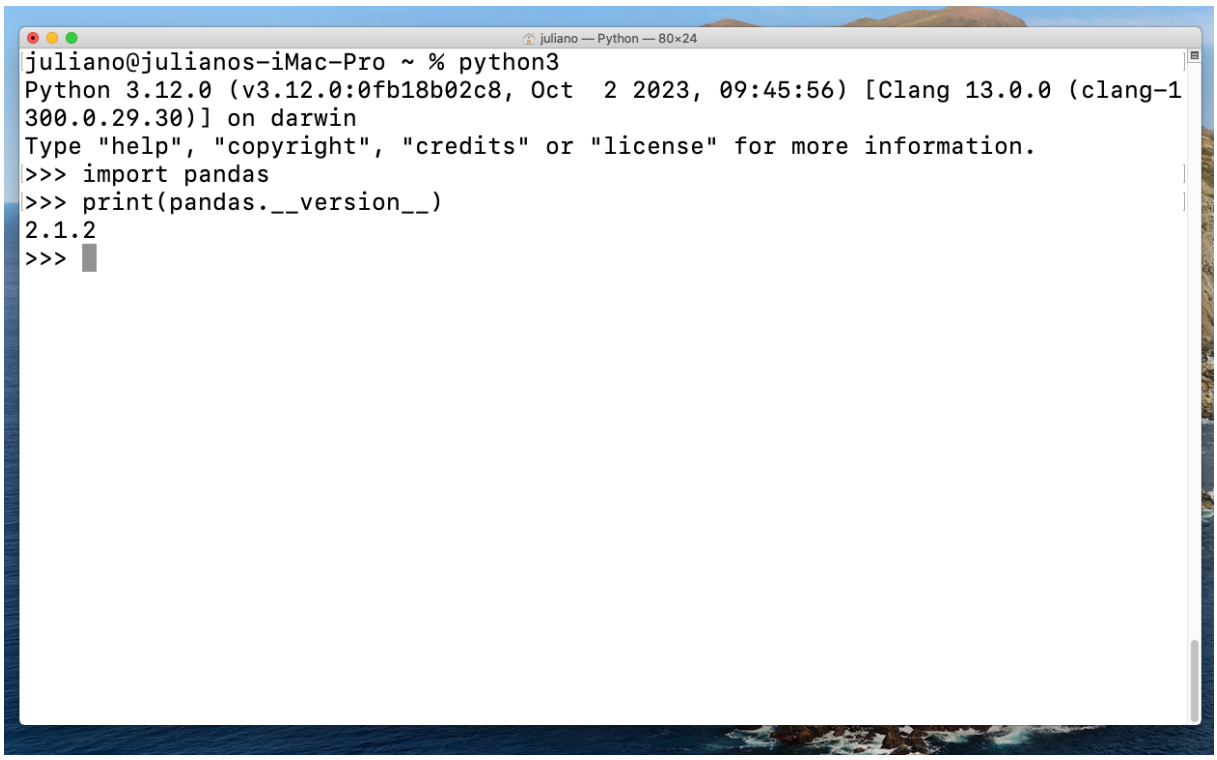


Figure 41: Checando a instalação do `pandas`

Rodando scripts de Python

Além de executar o console de Python, podemos usar o comando `python3` para rodar algum script de Python previamente criado.

O Mac não possui nenhum editor de texto bruto (o aplicativo `Notes` acaba salvando texto formatado, com informação de cores e fontes). Portanto, vamos usar o editor de texto via terminal chamado `nano` para este exemplo. Se preferir, pode baixar algum editor de texto como o VS Code, ou diversos outros que existem para Mac.

Abaixo, entramos na pasta Desktop e abrimos um arquivo novo com o `nano` chamado `meu_script.py`:

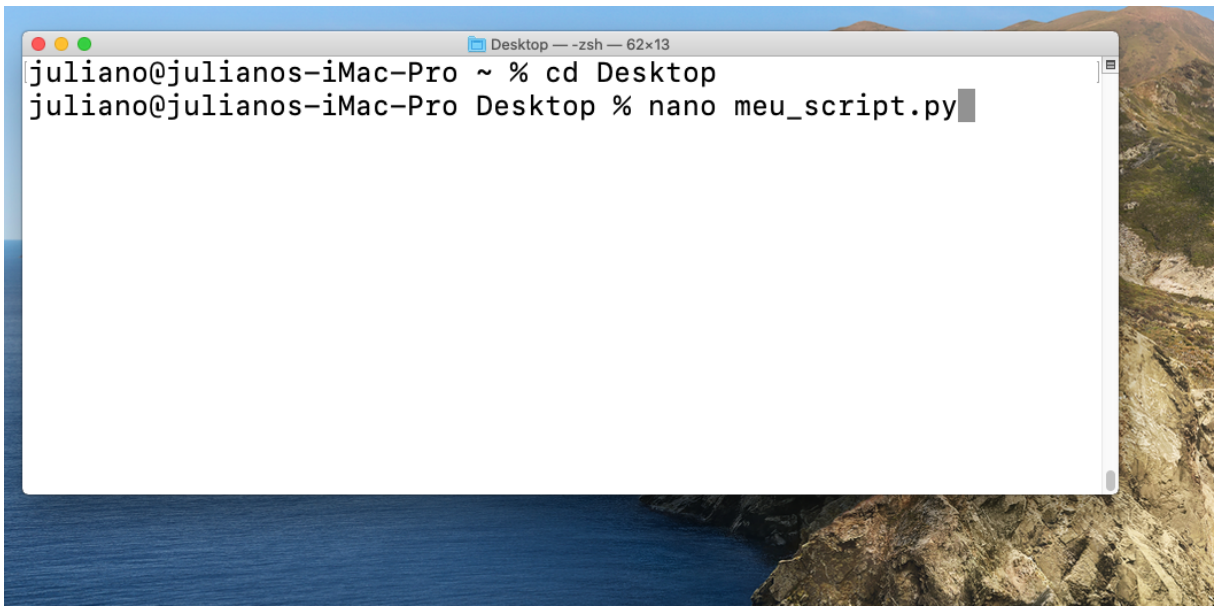


Figure 42: Acessando o editor de texto nano do terminal

A interface pode ser confusa para quem não está acostumado a rodar programas no terminal. No topo, aparece um cabeçalho com algumas informações, e no rodapé aparecem alguns atalhos de uso. Por exemplo, o atalho `Control` + X é usado para sair do nano e voltar ao terminal.

Escreva o script abaixo dentro do nano:

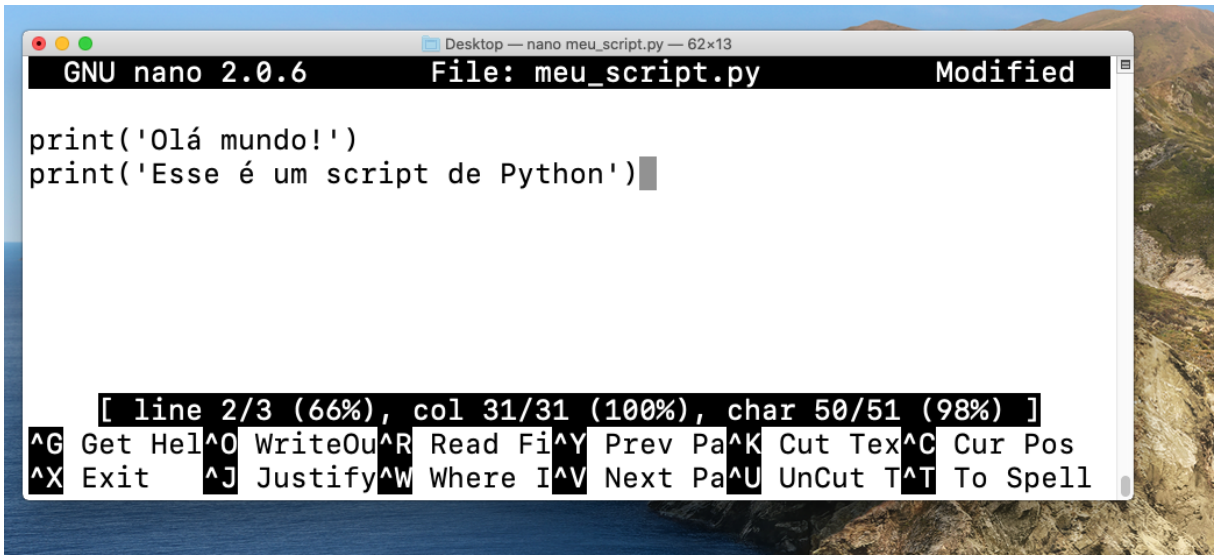


Figure 43: Editando o arquivo `meu_script.py` a partir do nano

Em seguida, dê `Control` + X para sair, e clique em Y para salvar as modificações:

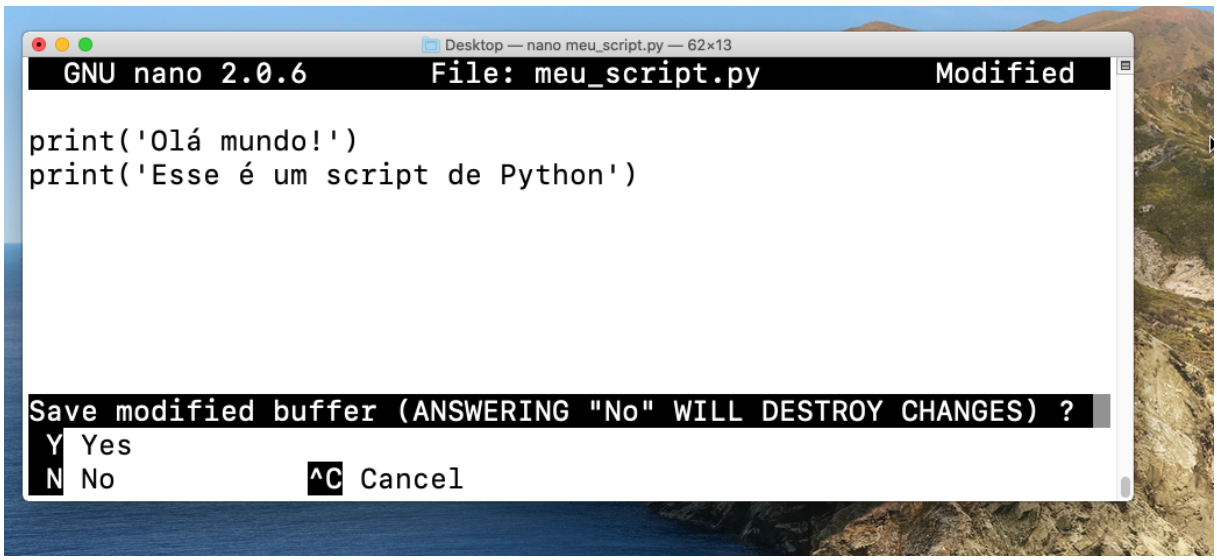


Figure 44: Salvando o script

O arquivo `meu_script.py` deverá aparecer no seu desktop. De volta ao terminal, use o comando `python3 meu_script.py` para executá-lo:

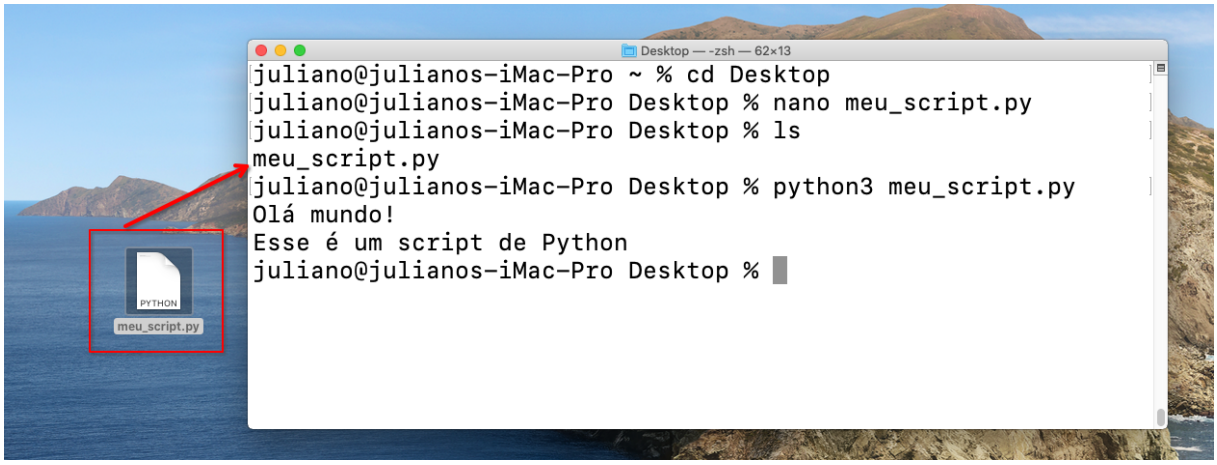


Figure 45: Rodando o script com Python

07. O terminal do Linux

O terminal do Linux é praticamente idêntico ao do Mac em termos de comandos. Isso se deve ao fato de ambos sistemas operacionais funcionarem com base no sistema UNIX.

Procure por um programa de terminal na sua instalação de Linux, ou utilize o atalho `Ctrl` + `Alt` + T já vem instalado por padrão. Ele é acessível buscando pelo programa `Terminal` nos aplicativos instalados:

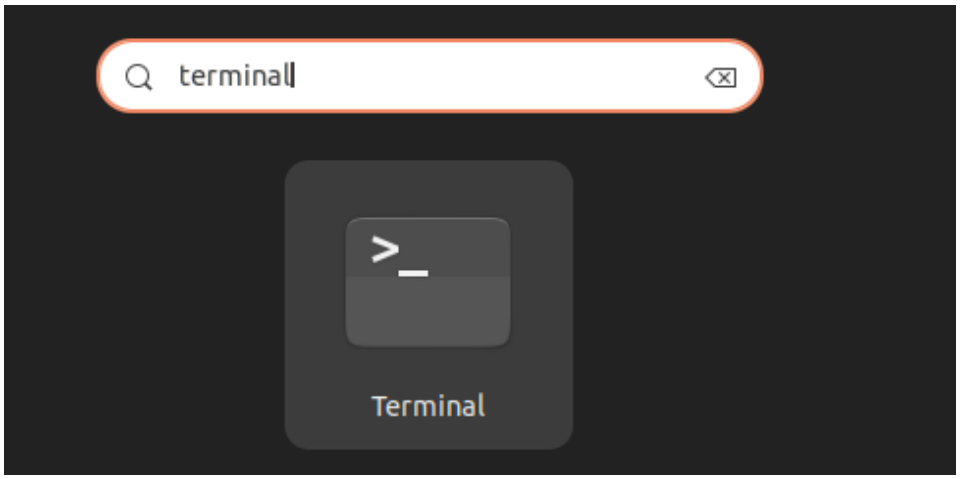


Figure 46: Abrindo o Terminal no Linux

Caminhos em sistemas UNIX

Assim como em Mac, os caminhos em Linux possuem duas diferenças em relação ao Windows:

- Utilizam barras para a frente (`/`), e não barras para trás (`\`), para delimitar as pastas.
- A pasta raiz do sistema é simplesmente a pasta barra (`/`), e não o `C:\`.

Comandos básicos

Você pode usar o **comando `ls`** (de “listar”) para exibir as pastas e arquivos. É o mesmo comando usado no Mac:

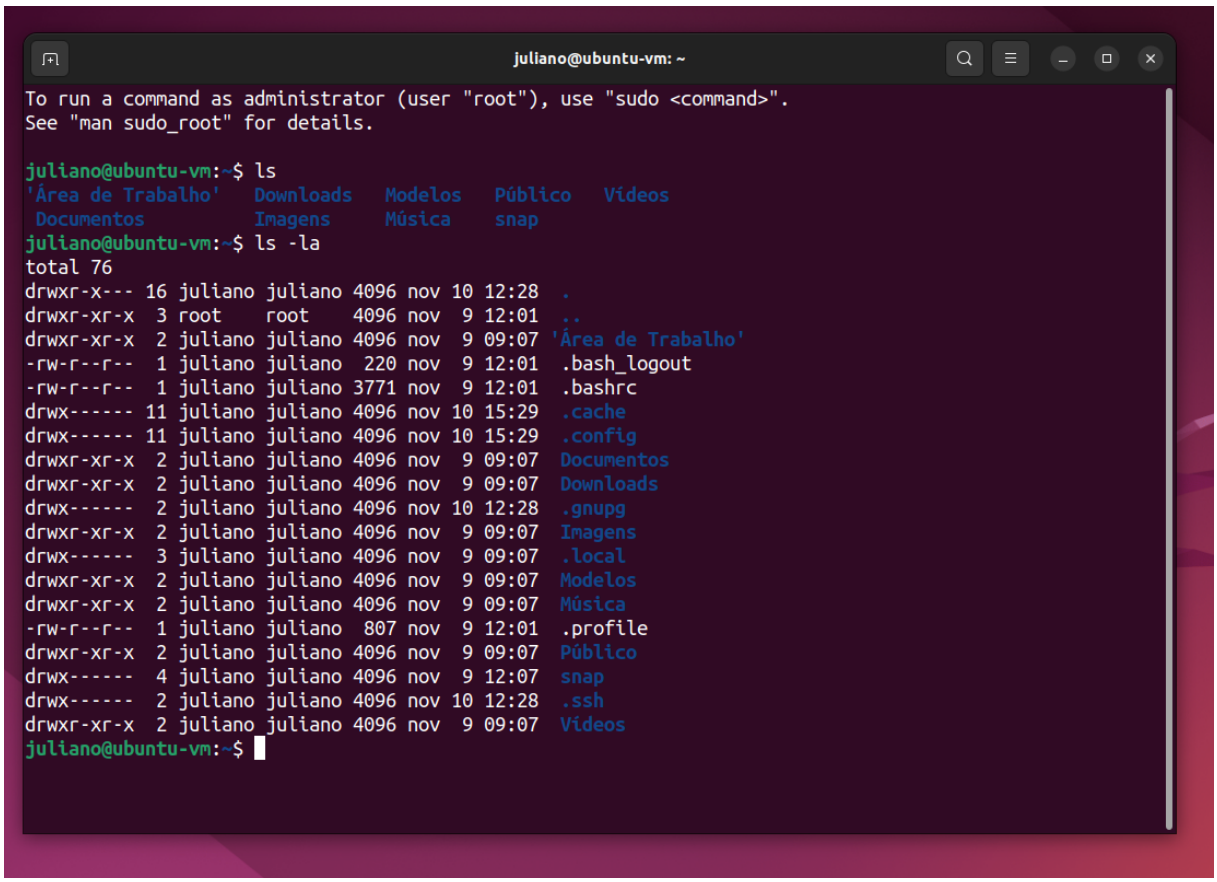


Figure 47: Comando `ls`

Note que na imagem acima executamos também o **comando `ls -la`**. Esta porção `-la` adicionada ao final do comando são **flags** que modificam o seu comportamento.

A flag `-l` faz com que os arquivos apareçam em uma lista vertical, similar ao comando `dir` de Windows. Já a flag `-a` faz com que pastas e arquivos ocultos (que em sistema UNIX, são aqueles com nome iniciado por ponto) seja exibidos também. Usar `-la` é apenas uma forma prática de combinar as flags `-l` e `-a`.

Para mudar de diretório, usamos o comando `cd` (do inglês *change directory*), que funciona da mesma forma como em Windows e Mac. Aqui, entramos na pasta `Documentos` e listamos seu conteúdo:

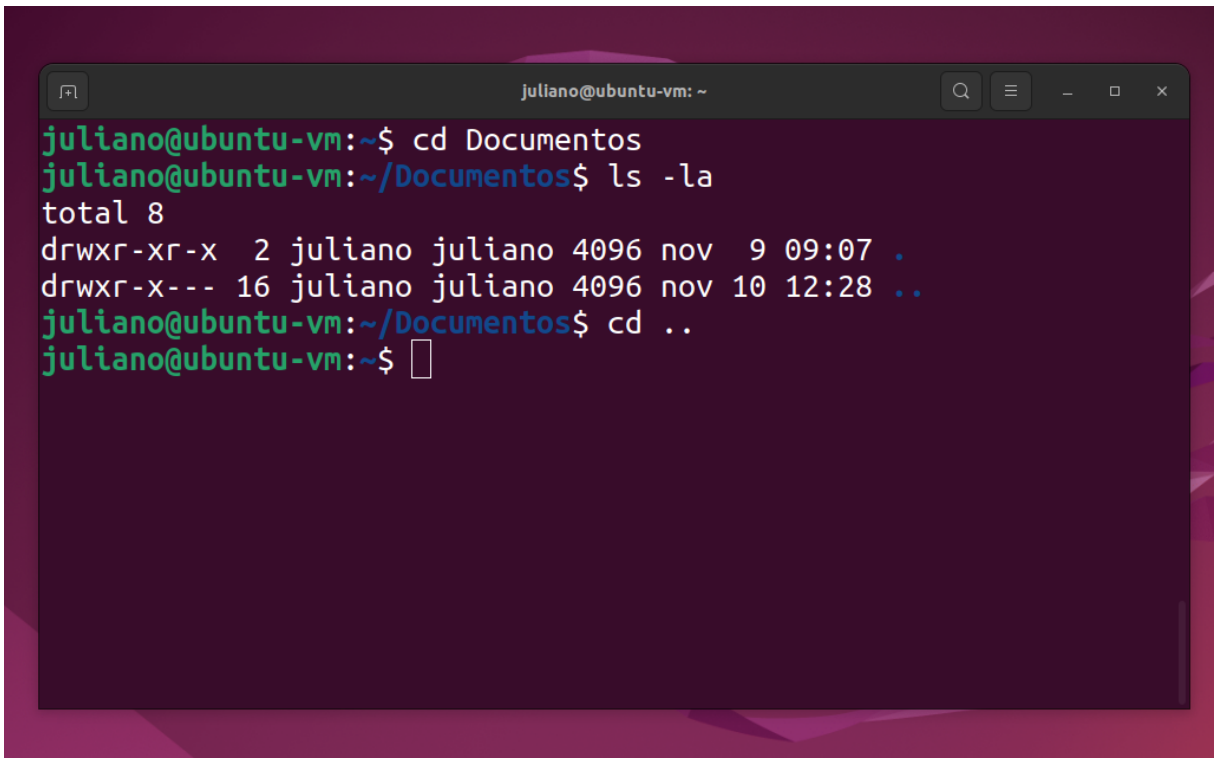


Figure 48: Comando cd

A pasta está vazia. Por isso, a listagem mostra apenas a pasta atual (representada pelo ponto `.`) e a pasta acima (representada pelo ponto-ponto `..`).

Note também que, em Linux, o caminho da pasta de usuário (geralmente `/home/seu-nome-de-usuario`) é abreviado com um caractere de til (~) no terminal.

Executáveis e as variáveis de ambiente

Sistemas UNIX também possuem a variável de ambiente PATH. Assim como em Windows, essa variável determinar os locais onde o sistema operacional procura por programas para executar.

Podemos observar o valor da variável PATH no próprio terminal usando o comando `echo $PATH`:

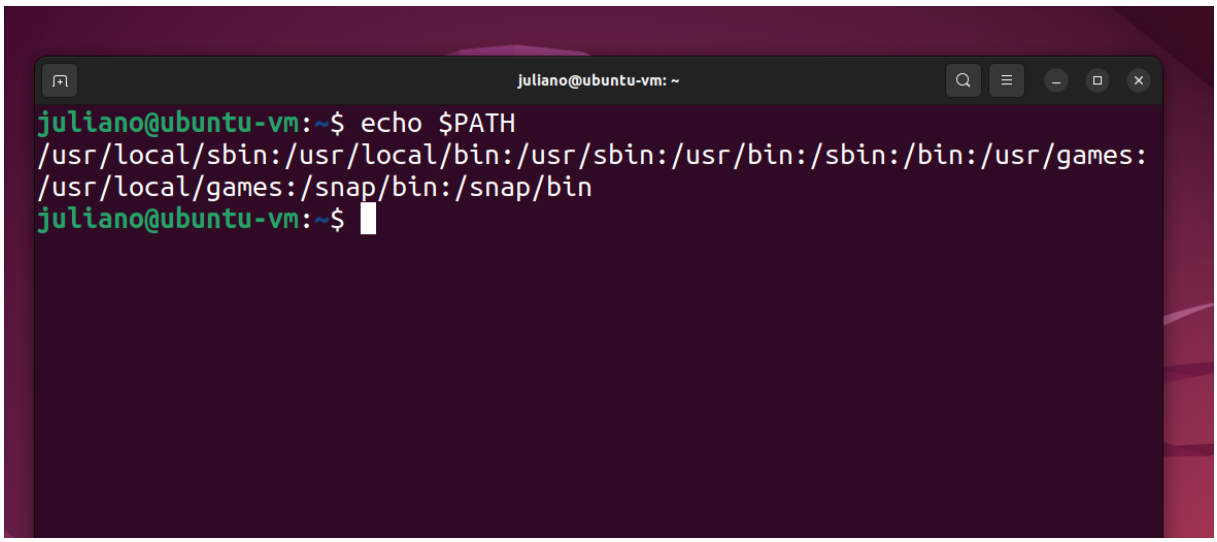


Figure 49: A variável de ambiente PATH

O resultado é uma lista de pastas do sistema, separadas por `:`.

Mesmo sem saber, já utilizamos o PATH ao usarmos o comando `ls`, já que ele é considerado um programa, que já vem instalado em sistemas UNIX.

Podemos até confirmar isso usando o **comando `which ls`**, que mostra o caminho onde o programa está instalado (é equivalente ao `where` que vimos no Windows).

Python em Linux

Nas versões mais recentes de Linux, o comando `python` não é reconhecido. Em versões mais antigas, ele pode ainda apontar para alguma instalação da versão 2 do Python.

Já o comando `python3` tende a ser reconhecido, apontando para uma instalação de Python 3 gerida pelo próprio sistema operacional:

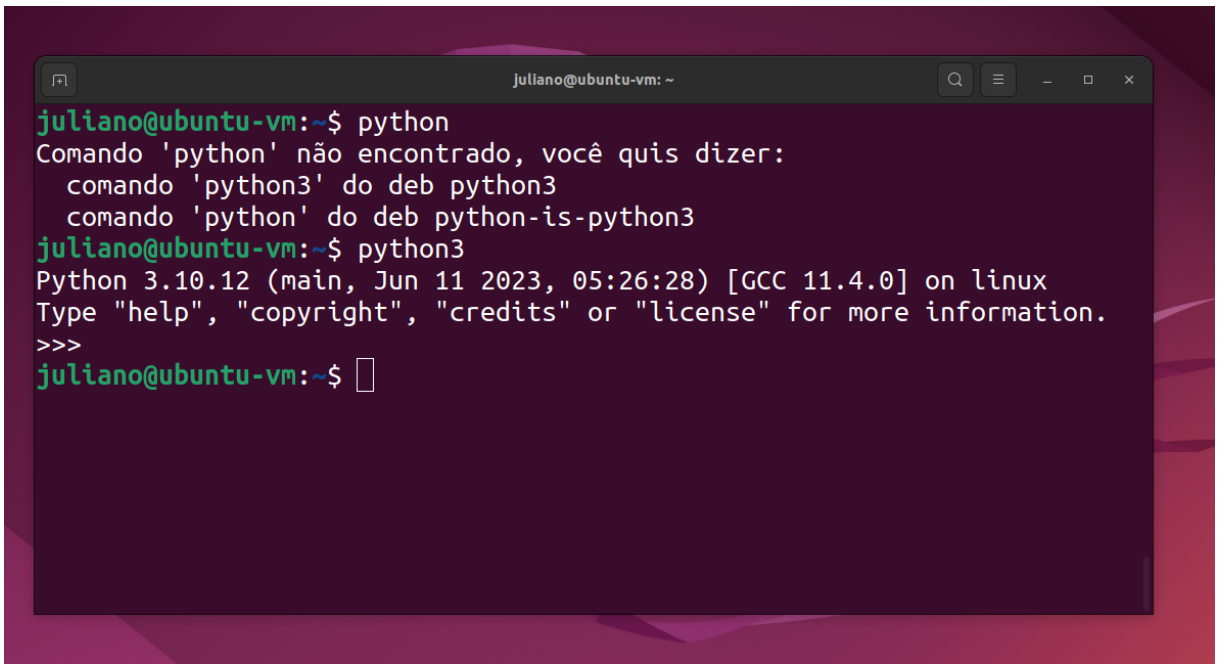


Figure 50: O comando `python` em Linux

Podemos usar o **comando `which python3`** para localizar onde está instalado o Python 3:

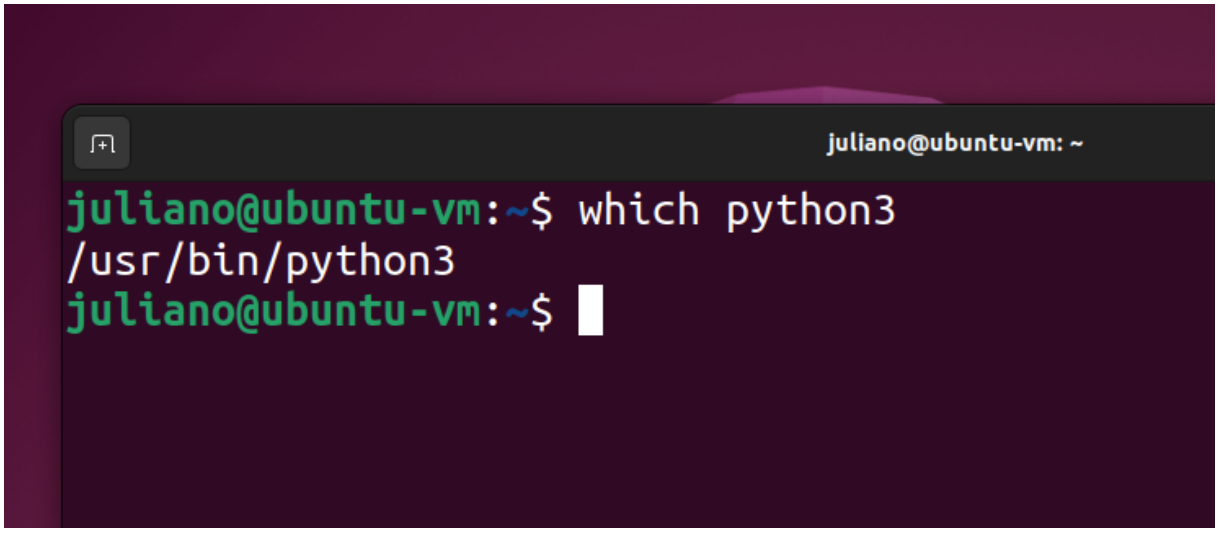


Figure 51: O comando `python` em Linux

Como podemos ver, em geral ele estará instalado em uma das pastas do sistema, uma vez que faz parte do sistema operacional.

Atenção: por mais que exista uma instalação de Python 3 no seu Linux, **o ideal é nunca mexer nas**

instalações de Python do seu sistema. Isso porque parte do seu próprio sistema operacional pode depender de algum pacote, e fazer instalações, atualizações ou modificações pode desconfigurá-lo!

Alguns tutoriais indicam que você use gerenciadores de pacotes do Linux, como o `apt`, para instalar Python. Esta é uma alternativa válida, mas existem 2 problemas com ela:

- Você só conseguirá instalar a versão de Python que o `apt` está configurado para baixar para a sua versão de Linux, que pode não ser nem a mais recente, ou a que você deseja:
- Algumas distribuições de Linux podem fazer alterações a esta instalação (por exemplo, mudando módulos da biblioteca padrão), e isto pode ser indesejável.

No lugar disso, vamos instalar Python pela fonte oficial na próxima aula.

08. Instalando Python no Linux

Como instalar Python pelo código-fonte

Não há instalador de Python para Linux. No lugar disso, temos que baixar o código-fonte e compilar o programa.

Este processo pode parecer um pouco inusitado, mas não se assuste! O processo envolve apenas alguns comandos no terminal, e é sempre uma boa ideia para usuários de Linux se acostumar a compilar programas a partir do seu código-fonte.

Note que, neste tutorial, estamos baixando a versão `3.12` de Python. Se você estiver baixando outra versão, lembre-se de adaptar os comandos onde aparece o texto `3.12` para a versão que você está baixando!

Baixando e extraindo código-fonte

Baixe o código-fonte do site python.org:

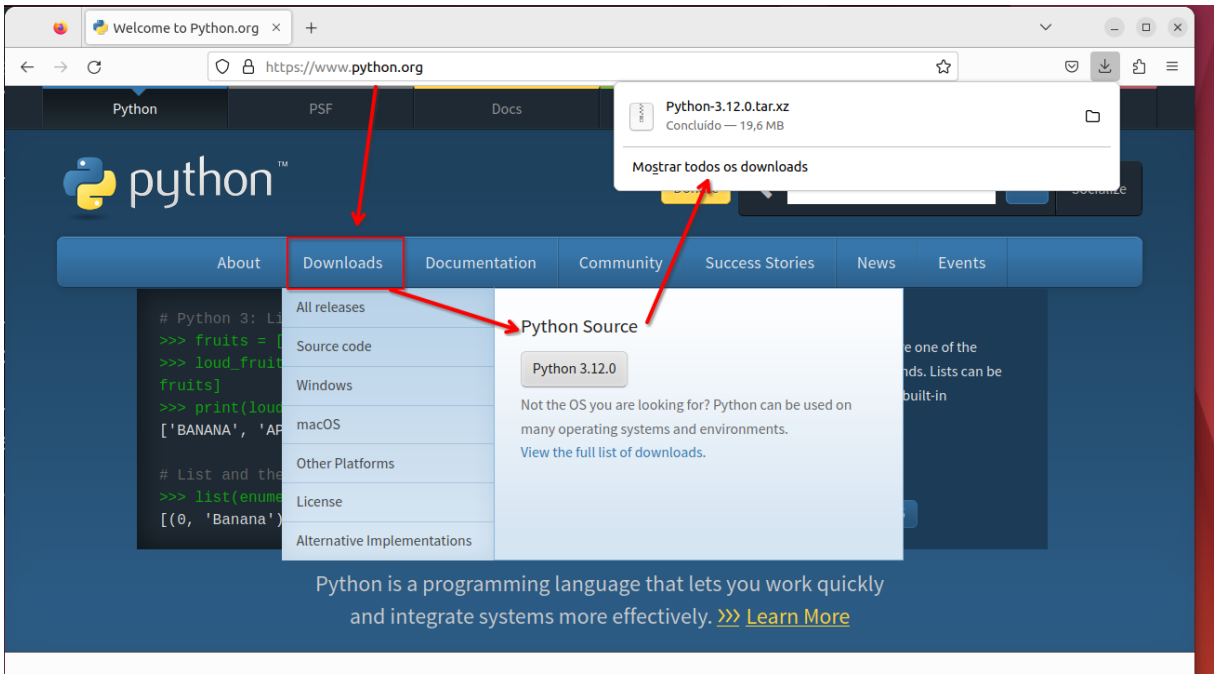


Figure 52: Baixando Python do site oficial

O arquivo baixado está compactado, portanto é preciso extrair seu conteúdo para uma pasta. Para isso, use o comando abaixo (lembrando de ajustar o número `3.12.0` de acordo com a versão baixada

de Python):

```
tar -xf Python-3.12.0.tar.xz
```

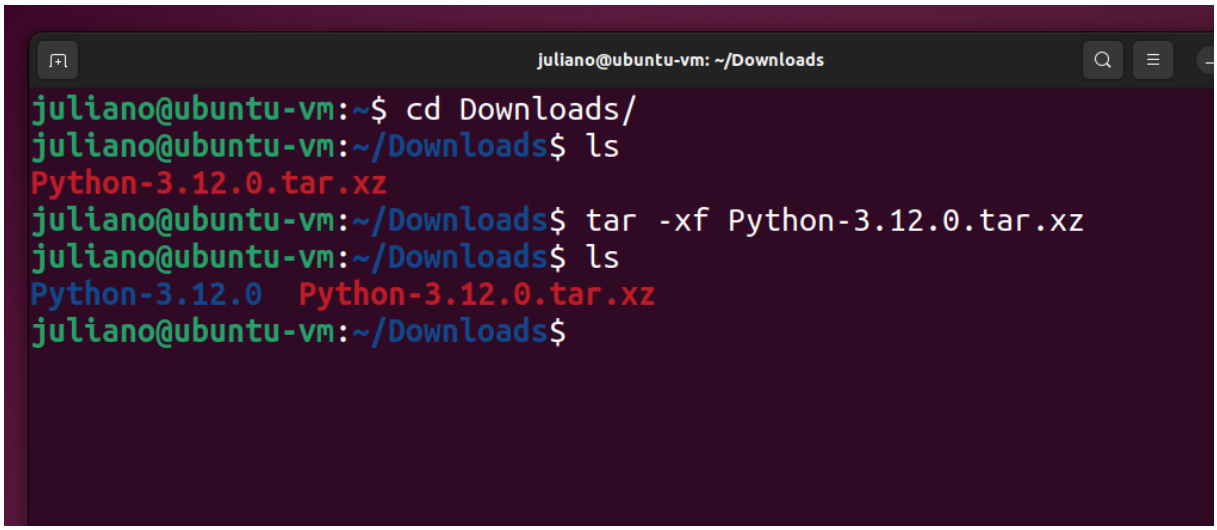


Figure 53: Descompactando o código fonte

Em seguida, use o comando `cd Python-3.12.0` para entrar na pasta contendo o conteúdo extraído (ajuste o número da versão de acordo com a versão que você baixou):

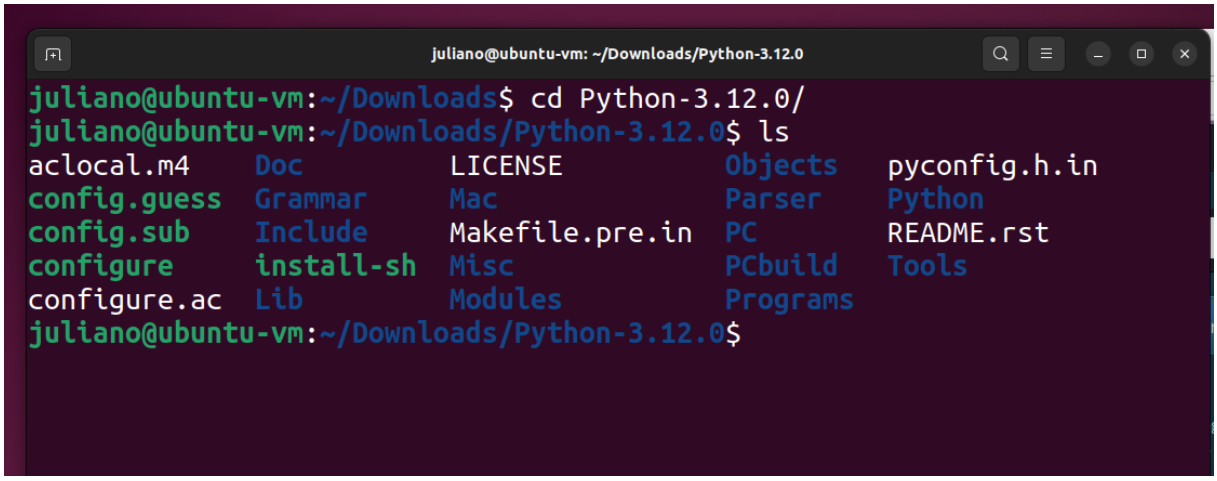


Figure 54: Entrando na pasta decompactada

Instalando as dependências de compilação

Para que seja possível compilar Python em Linux, é preciso baixar as dependências de compilação. Isto geralmente é feito instalando os pacotes pelo gerenciador de pacotes (`apt`, `yum`, `pacman`) da sua distribuição Linux.

Para Ubuntu e seus derivados, o comando abaixo faz a instalação de todas estas dependências de uma única vez através do gerenciador de pacotes `apt`:

```
sudo apt-get install build-essential gdb lcov pkg-config \
      libbz2-dev libffi-dev libgdbm-dev libgdbm-compat-dev liblzma-dev \
      libncurses5-dev libreadline6-dev libsqlite3-dev libssl-dev \
      lzma lzma-dev tk-dev uuid-dev zlib1g-dev
```

Para mais detalhes, veja a seção de instalação de dependências no Guia do Desenvolvedor Python (em inglês).

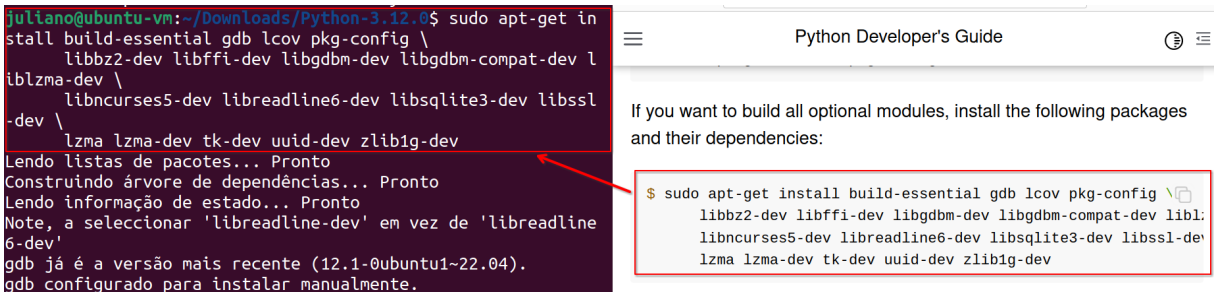


Figure 55: Instalando as dependências de compilação

Compilando Python

Uma vez baixadas as dependências, podemos configurar a compilação com o comando:

```
sudo ./configure
```

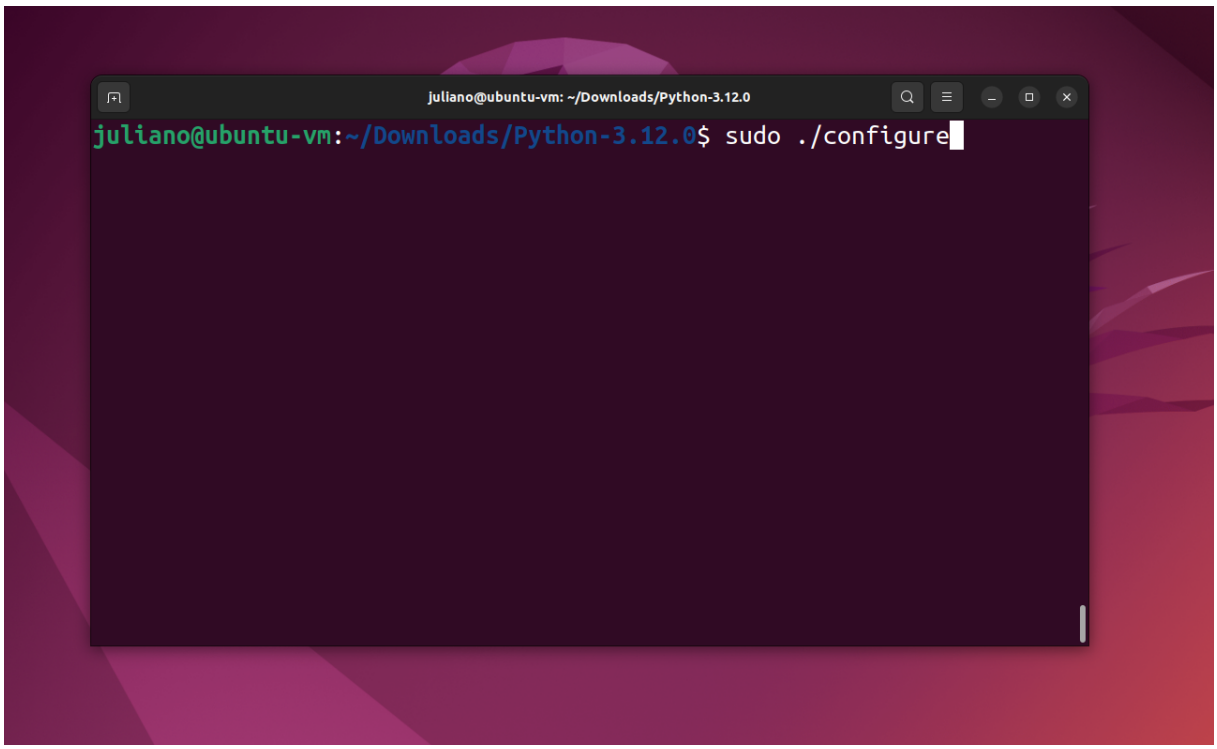


Figure 56: Configurando a compilação

O comando sudo faz com que o comando rode com privilégios de administrador. Isso requer que o seu usuário tenha estes privilégios. Além disso, será necessário digitar a senha do seu usuário para confirmar a operação (por segurança, nada aparece na tela quando você digita a senha).

Uma grande quantidade de texto irá aparecer no terminal. Espere o prompt retornar ao seu controle, e em seguida use o comando:

```
sudo make -j "$(nproc)"
```

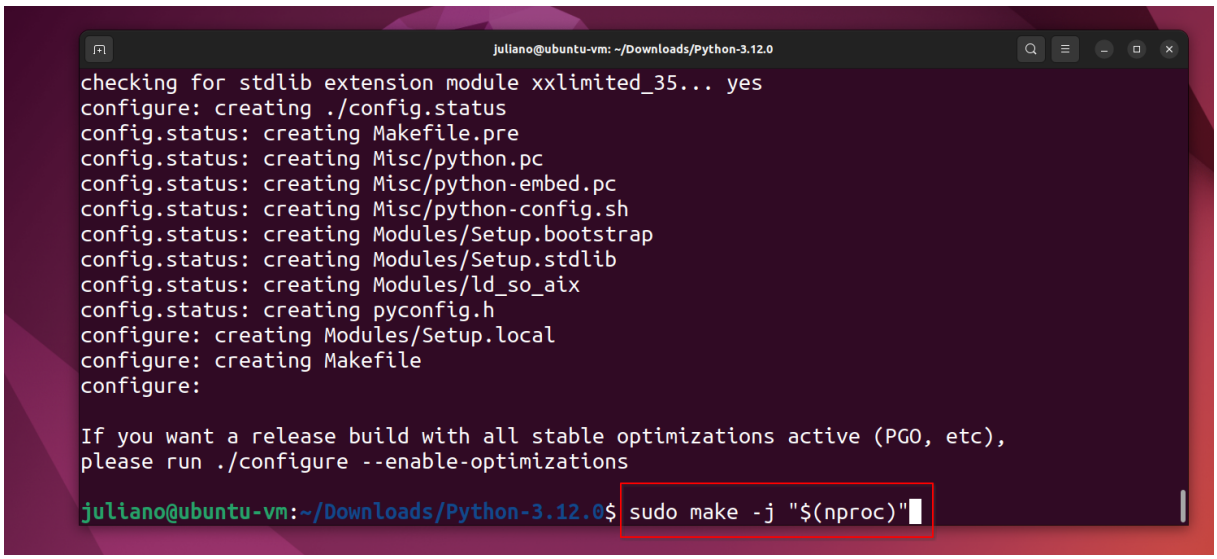


Figure 57: Compilando Python

Opções de instalação

Quando a compilação terminar, falta apenas instalar o Python! Há duas formas de fazer isso:

- `sudo make install`: instala Python e sobrescreve o comando `python3` do seu sistema. Em outras palavras, o comando `python3` passará a se referir a sua nova instalação de Python.
- `sudo make altinstall`: instala Python, porém **não** sobrescreve o comando `python3` do seu sistema. Você consegue acessar o Python do sistema usando o comando `python3`, e sua instalação de Python com o comando `python3.12` (ou o comando equivalente, de acordo com a versão instalada).

Não há forma "correta" de instalar. Se você quiser a comodidade de usar o comando `python3` para rodar a sua instalação de Python, use `sudo make install`. Por outro lado, se você quiser manter o acesso do Python do sistema e garantir que outros programas consigam acessá-lo, use `sudo make altinstall`.

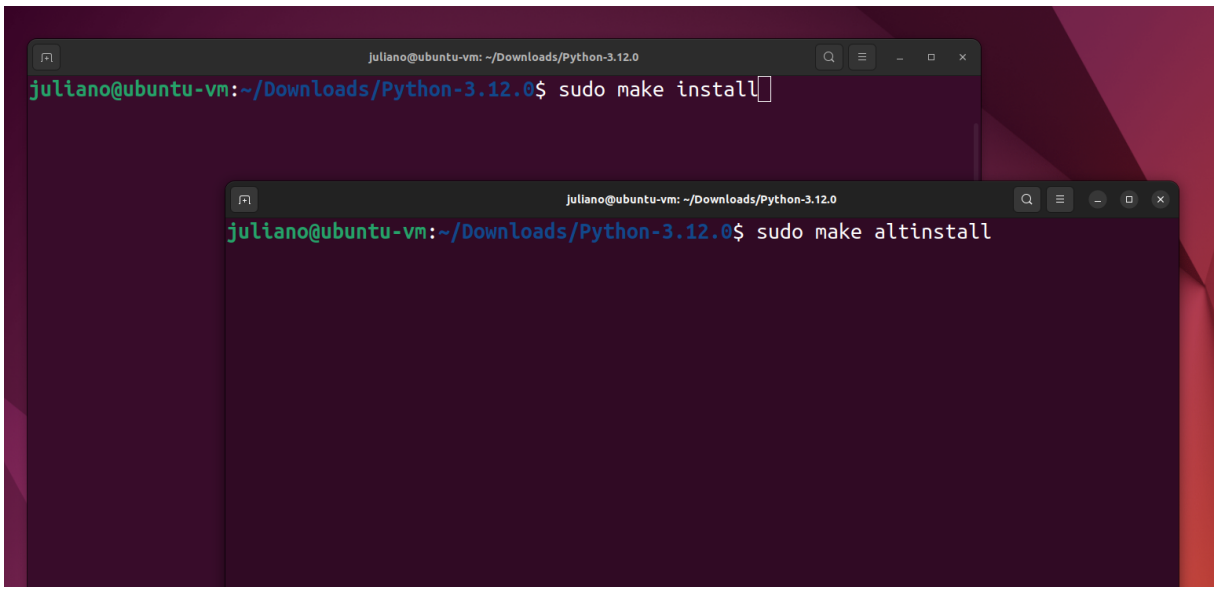


Figure 58: As opções de instalação de Python

Verificando sua instalação

Pronto! Agora já é possível acessar a nova instalação de Python pelo terminal. A instalação é feita em uma pasta padrão localizada pelo PATH, portanto ela deve ser encontrada pelo seu terminal.

A imagem abaixo mostra o resultado após a instalação com `sudo make altinstall`. Veja que `which python3`, aponta para a instalação do sistema, enquanto `which python3.12` aponta para a instalação nova:

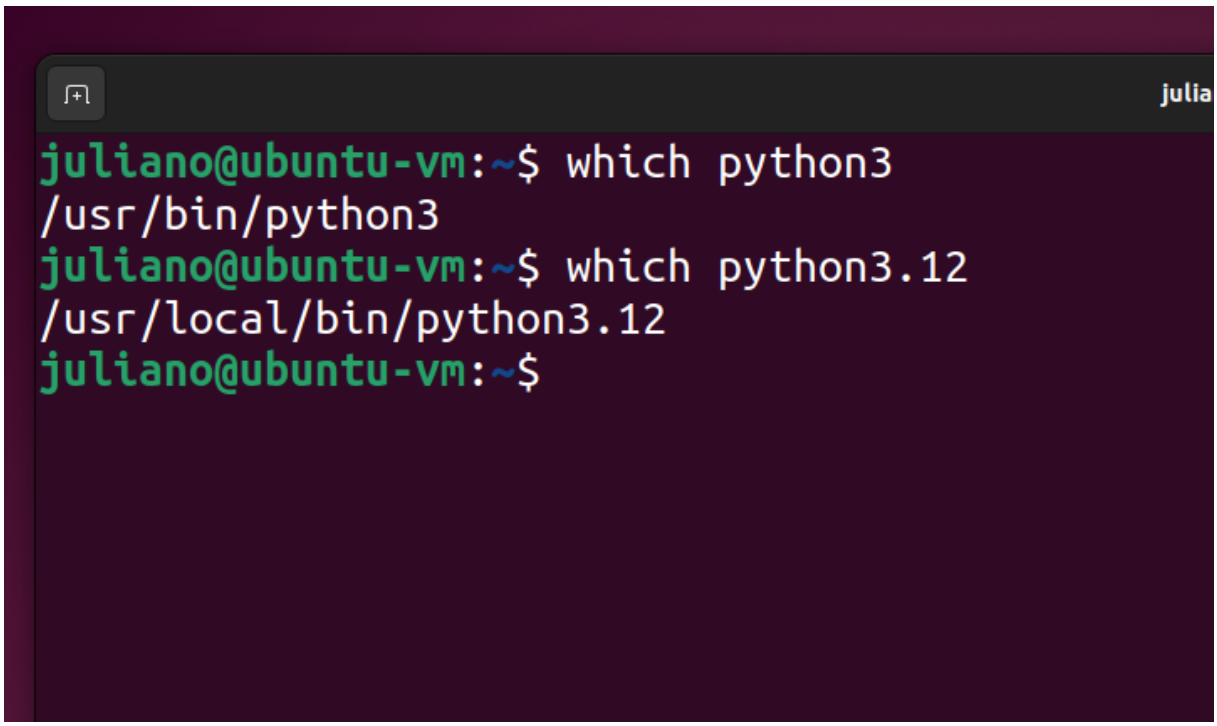


Figure 59: Conferindo a instalação

O mesmo acontece para o comando `pip` (veja que, com a instalação feita pelo `altinstall`, apenas o comando `pip3.12` foi criado):

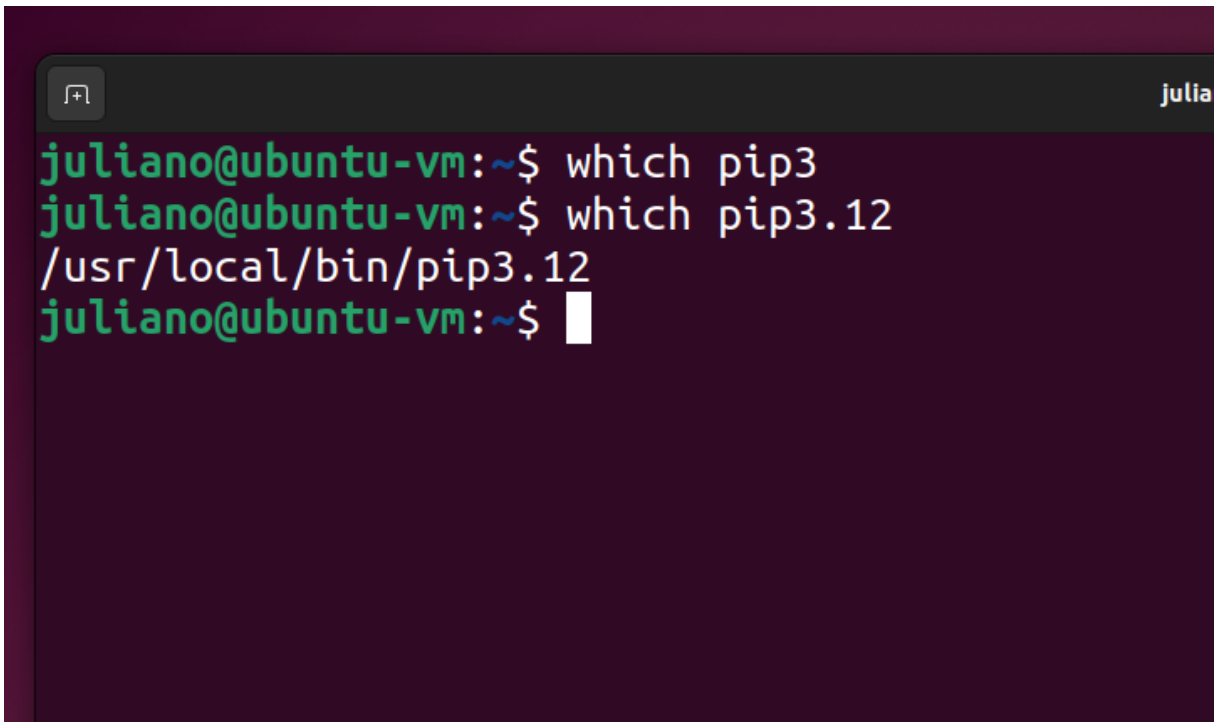


Figure 60: Conferindo a instalação do `pip`

Instalando dependências

Podemos usar o `pip install` para instalar dependências (Ao invés de usar `pip3.12 install`, você pode preferir o comando `python3.12 -m pip install`):

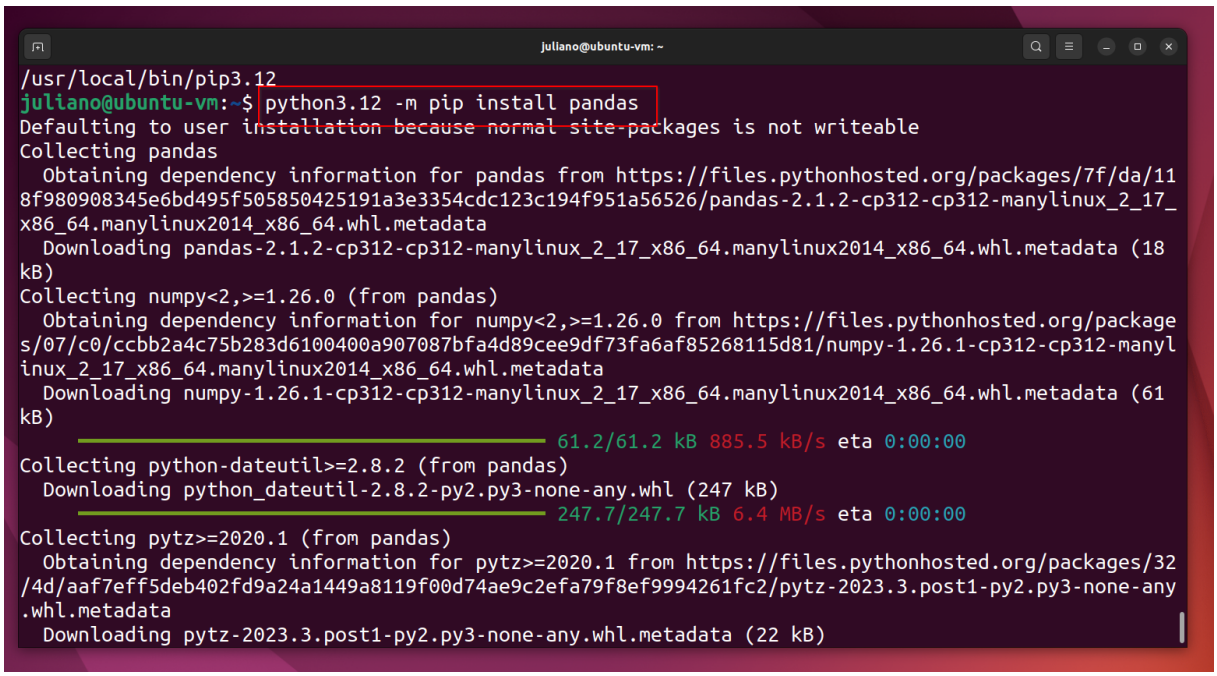


Figure 61: Instalando dependências com `pip`

Após a instalação, podemos entrar no console de Python e importar as bibliotecas recém instaladas:

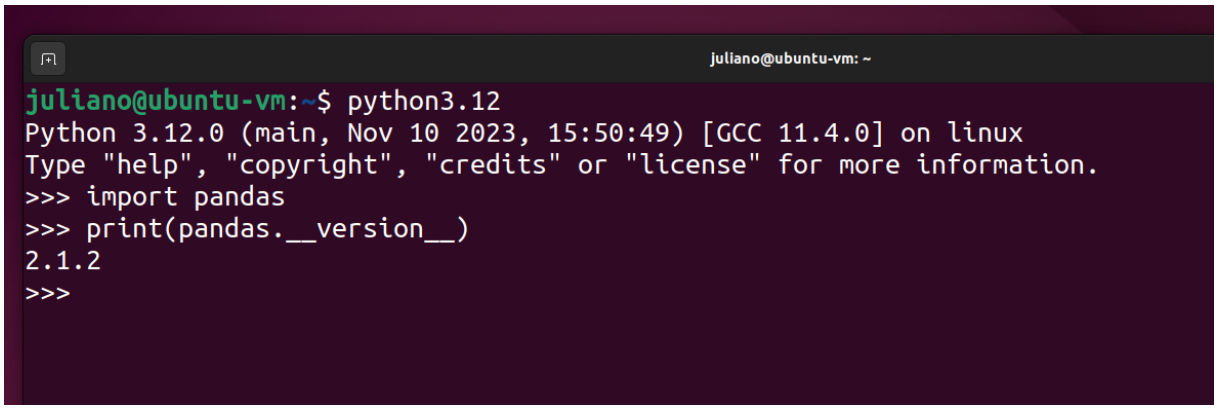


Figure 62: Checando a instalação das dependências

Rodando scripts de Python

Além de executar o console de Python, podemos usar o comando `python3.12` para rodar algum script de Python previamente criado.

Praticamente todas as distribuições de Linux possuem algum editor de texto pré-instalado. No caso

do Ubuntu, basta procurar por “Editor” na barra de busca de aplicativos.

Crie um script chamado `meu_script.py` e o salve na Área de Trabalho. Em seguida, rode o script com o comando `python meu_script.py`, conforme imagem abaixo (veja que é possível rodá-lo passando o caminho até o script, ou então usando `cd` para navegar até a sua pasta):

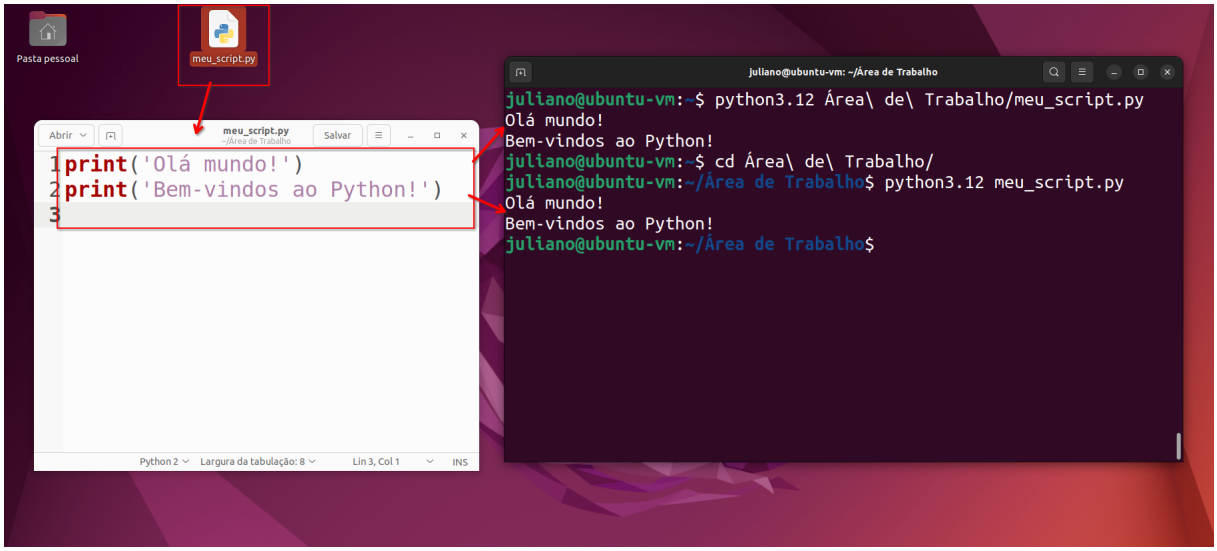


Figure 63: Executando um script de Python

09. Outras opções de instalação

Além do instalador oficial, há outras opções de instalação dependendo do seu sistema operacional. Por exemplo, no caso do Windows, já vimos que existe a opção de instalar a partir da Windows Store.

Uma das mais famosas e utilizadas é o **Anaconda**. O Anaconda é uma instalação completa de Python, incluindo as principais bibliotecas utilizadas para análise de dados e machine learning.

A instalação padrão inclui o **conda**, um programa de linha de comando usado para instalar e gerenciar outros pacotes de Python (e de outras linguagens também), e o **Anaconda Navigator**, um programa onde é possível gerenciar sua instalação de Python e de outros programas úteis também.

Instalando Anaconda

Vá até o site de download do Anaconda e clique no botão de `Download`:

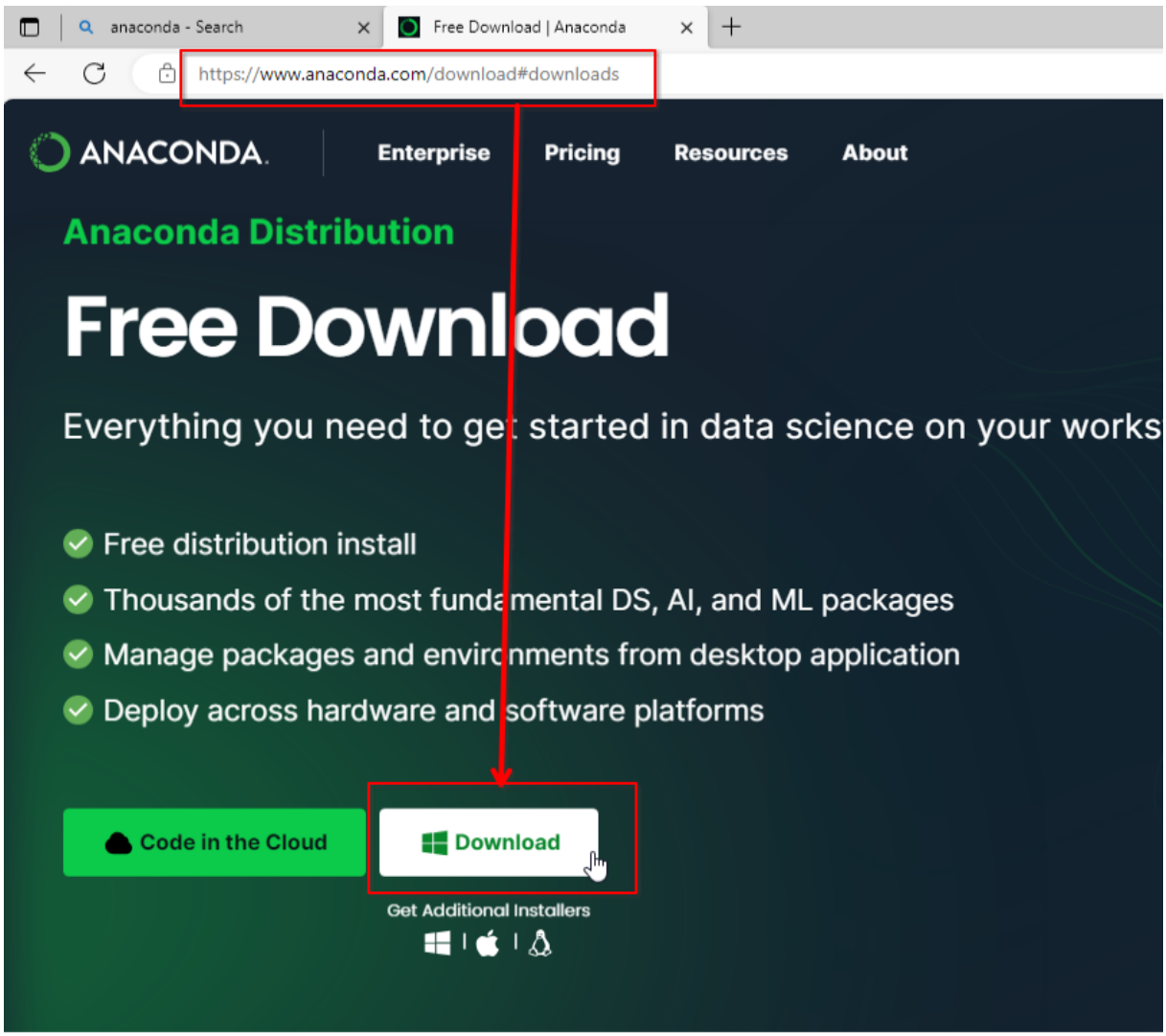


Figure 64: Download do Anaconda

Você também pode clicar em `Get Additional Installers` para ver todas as opções de instaladores do Anaconda. O instalador de Windows é gráfico. Em Mac, há a opção de instalador gráfico ou por linha de comando. Já em Linux, há apenas a opção de instalar por linha de comando.

Figure 65: Opções de instalação do Anaconda

Após o Download, siga os passos no instalador do Anaconda:

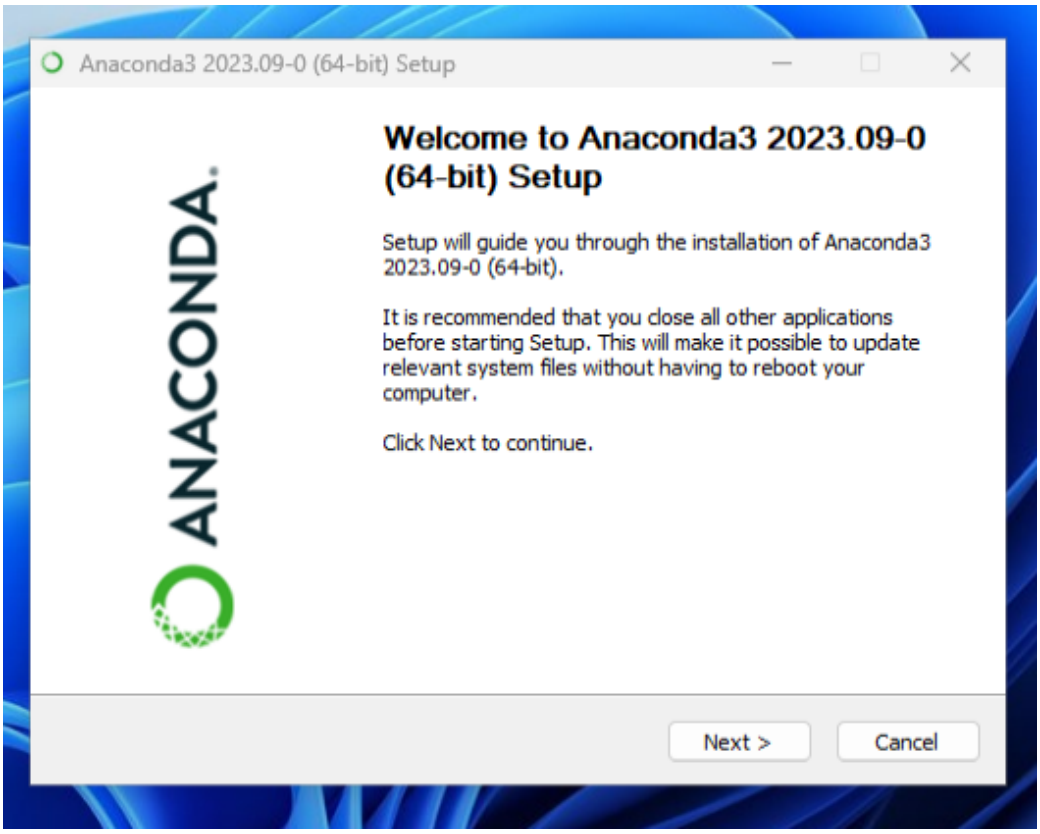


Figure 66: Instalador do Anaconda

Durante a instalação, note que há uma opção para registrar o Python do Anaconda como opção do seu sistema. Marcar ajuda o seu sistema a encontrar esta instalação de Python, mas cuidado para não sobrescrever a instalação anterior!

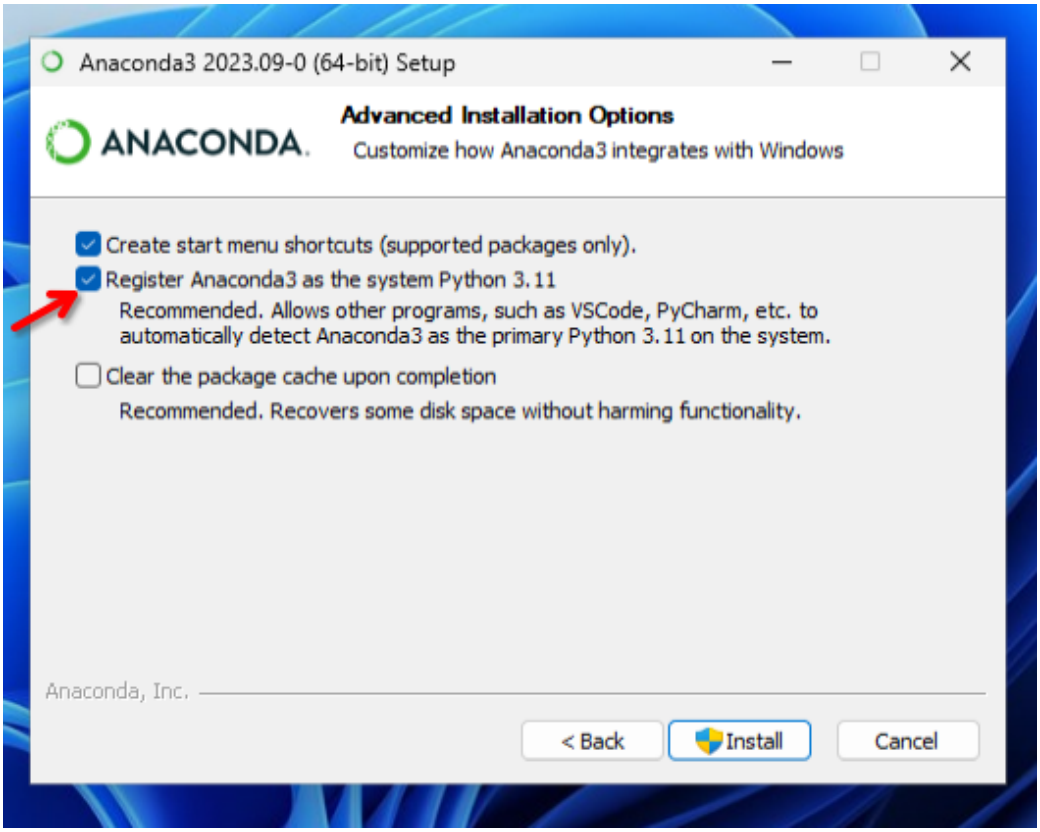


Figure 67: Opção para registrar o Python do Anaconda como a versão principal do sistema

Anaconda Navigator

Após a instalação, abra o **Anaconda Navigator**. Este programa serve como um menu interativo, a partir do qual é possível iniciar um CMD, gerenciar pacotes de Python, e instalar programas externos.

De forma geral, o foco do Anaconda é a **ciência de dados**. Por isso, ele já vem com um ambiente de Python pré-configurado com as principais bibliotecas usadas por quem quer trabalhar com ciência de dados. Além disso, muitos dos programas pré-instalados (ou sugeridos para instalação) são relacionados à análise de dados:

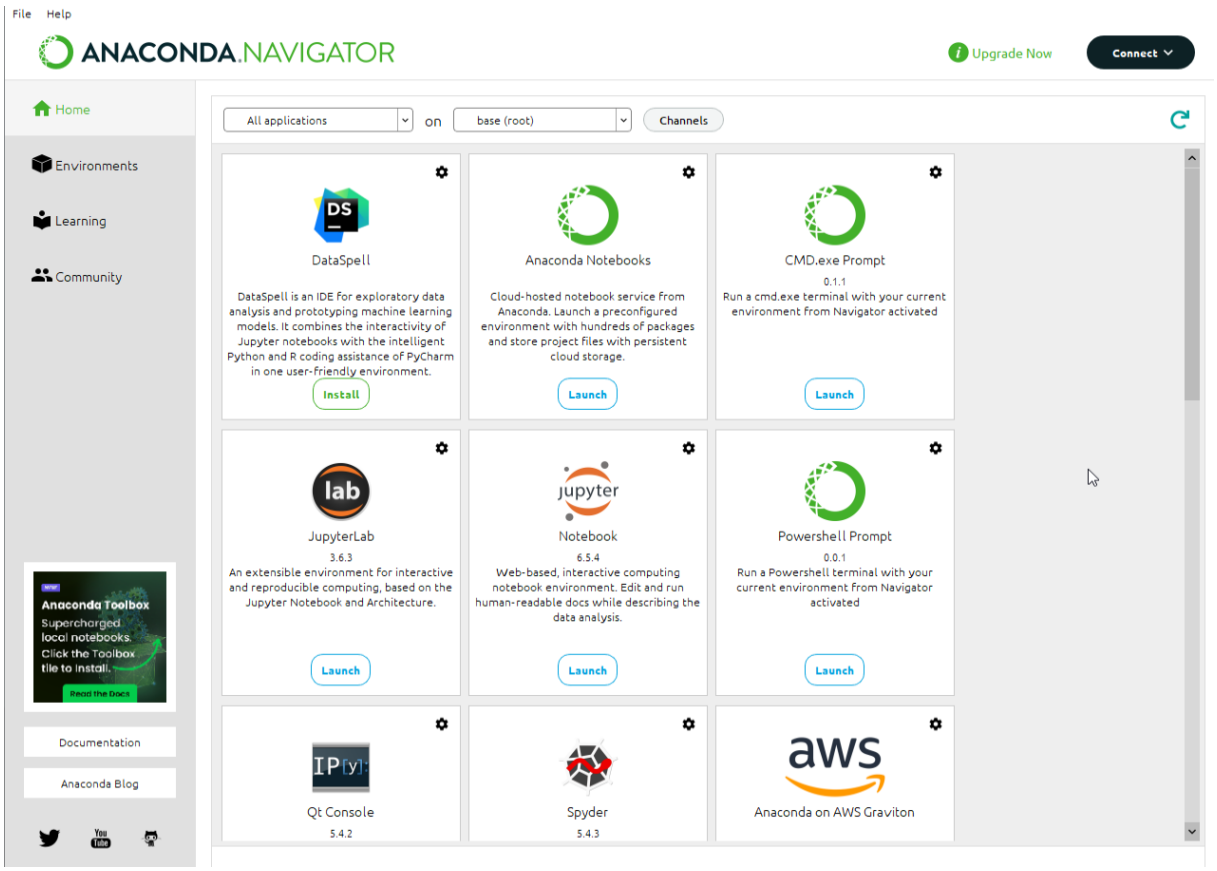


Figure 68: Tela inicial do Anaconda Navigator

Experimente abrir o `CMD.exe Prompt` e veja o output do comando `where Python`. Este terminal roda a versão de Python que foi instalada pelo Anaconda! O prefixo `(base)` indica isso também.

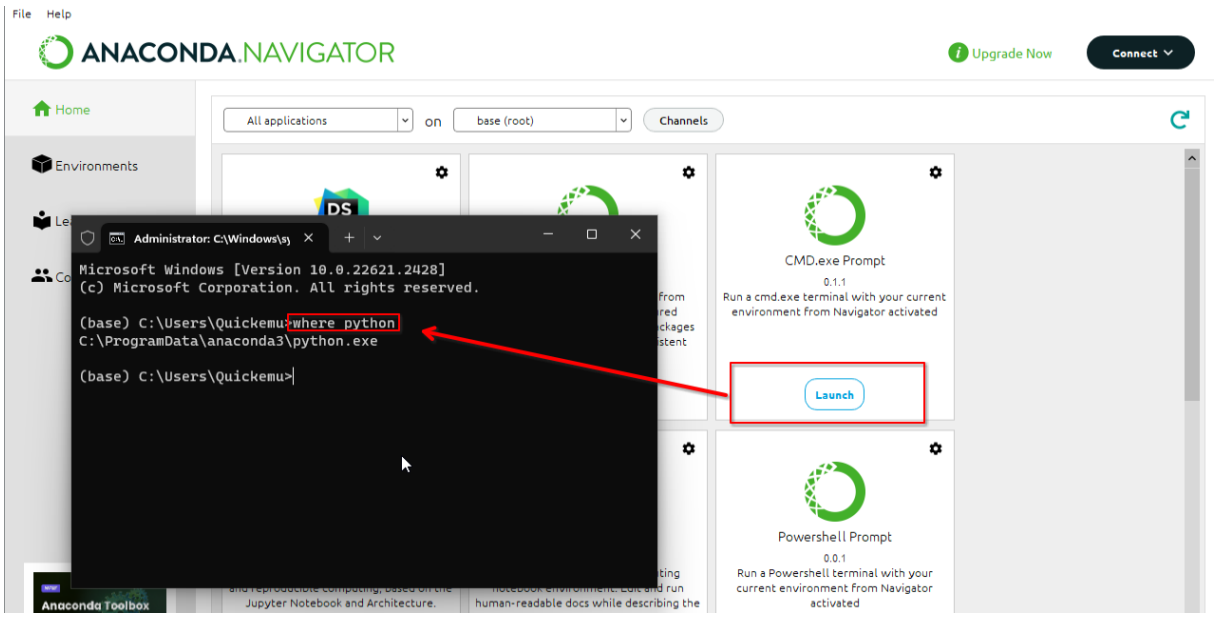


Figure 69: Rodando o CMD a partir do Anaconda Navigator

Nas outras opções, você pode instalar IDEs e outros programas relacionados à análise de dados. Note que é até possível instalar a linguagem de programação R e a IDE `RStudio` a partir do Anaconda Navigator!

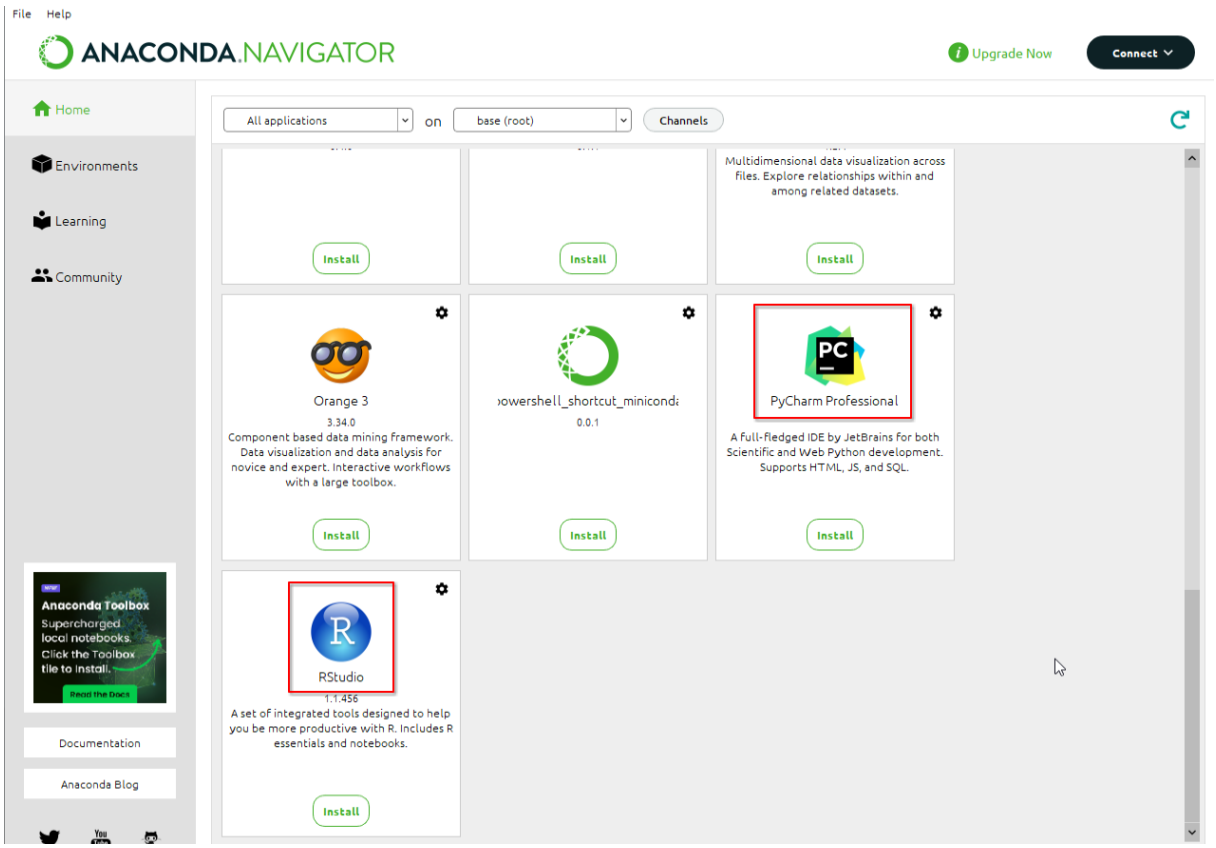
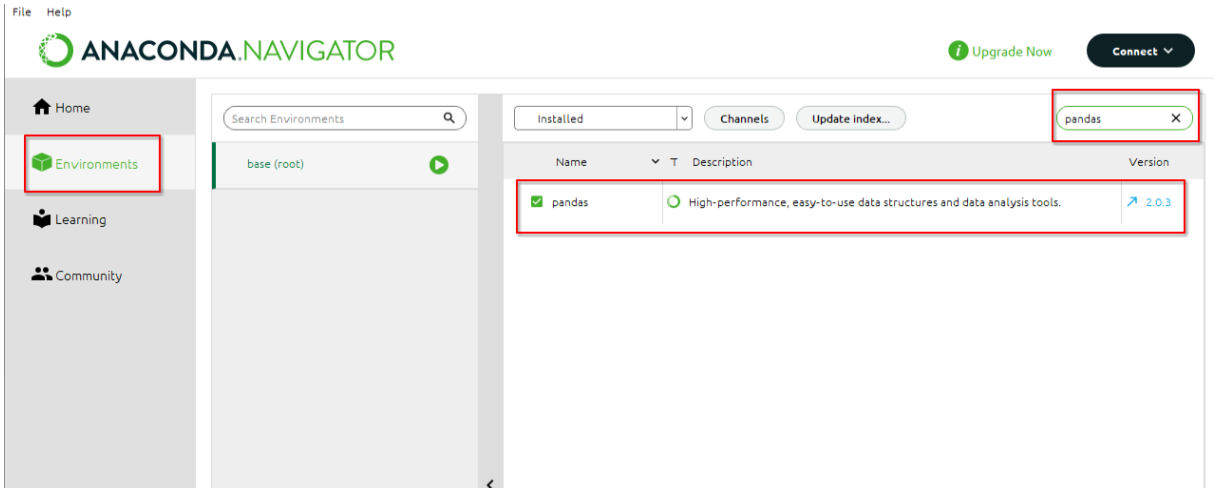


Figure 70: IDEs e outros programas do Anaconda Navigator

Na aba `Environments`, é possível ver os pacotes de Python instalados no ambiente do Anaconda. Também é possível criar ambientes diferentes, cada um com seus próprios pacotes. Por padrão, o Anaconda vem com apenas um ambiente configurado, chamado `base`.



Devo usar o Anaconda?

O Anaconda é de fato bastante poderoso, e bastante útil especialmente para quem quer trabalhar com ciência de dados. Ele permite ter um ambiente funcional de Python com um único download e instalação.

Por outro lado, o Anaconda acaba "escondendo" alguns detalhes sobre como Python funciona e como é instalado. E não é incomum termos alunos com dificuldades em fazer o Python do Anaconda interagir com outros programas, como VS Code, seja por alguma configuração ligeiramente diferente da esperada pelo Anaconda, ou por falta de entendimento do aluno sobre como estes programas funcionam e interagem.

Isso não é demérito do Anaconda, afinal nós mesmos recomendamos seu uso para alunos que estão começando a aprender a programar, e querem iniciar com Python com o menor número possível de cliques. Este é apenas um exemplo do equilíbrio que existe entre **flexibilidade e simplicidade**.

Para um programa como o Anaconda ser simples de instalar, os desenvolvedores tiveram de tomar várias decisões por você. E por mais que eles tenham pensado em todos os casos possíveis de todos os usuários, sempre há o risco de você chegar em um ponto onde deseja ter mais flexibilidade.

Por outro lado, flexibilidade demais pode ser uma armadilha, principalmente quando não se entende exatamente o que está acontecendo. Para quem está começando a programar, a ideia de terminais, CMD, Python, IDEs, … Tudo isso junto, pode confundir mais do que ajudar.

Esse mesmo tipo de ônus e bônus existe em ferramentas de low-code, em comparação à programação "tradicional". No fim das contas, cabe a cada um usar a ferramenta que faz sentido para a tarefa. O objetivo deste curso não é dizer que é melhor usar a ferramenta X ou Y, mas entender como cada uma funciona e qual seus limites, para que possamos tomar decisões informadas no futuro.

Feito este discurso, seguimos para a instalação e uso de IDEs!

10. O que são IDEs?

IDEs são programas desenvolvidos especialmente para desenvolver código. O nome significa Ambiente Integrado de Desenvolvimento (do inglês *Integrated Development Environment*).

Eles não são programas estritamente necessários para criar e executar códigos de Python. É perfeitamente possível fazer isso da forma como fizemos nas aulas anteriores, criando scripts em arquivos de texto, e depois os executando através do terminal. Dito isso, IDEs facilitam tanto a produtividade de desenvolvedores, que não faz sentido não utilizá-las.

Dependendo do ambiente que você estiver desenvolvendo (por exemplo, se estiver escrevendo código diretamente em um servidor através de um terminal), talvez não tenha IDEs disponíveis para usar. Neste caso, pode ser necessário utilizar formas mais simples, como editores de texto via terminal, para modificar seus códigos. Mas em todas as outras circunstâncias, faz muito mais sentido utilizar uma IDE!

Interface de uma IDE

A aparência de uma IDE pode mudar drasticamente entre programas distintos. Dito isso, é esperado que uma IDE possua algumas telas e ferramentas padrão.

A imagem abaixo (da IDE **PyCharm**) apresenta as principais áreas de uma IDE tradicional:

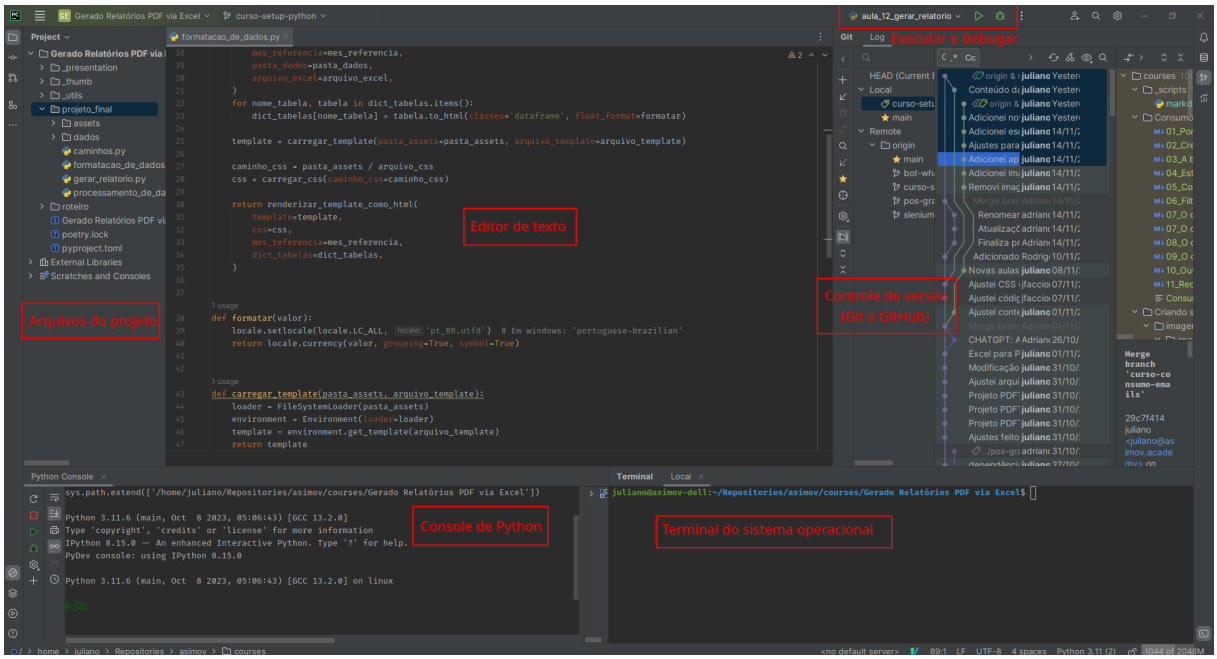


Figure 71: A interface de uma IDE

A seguir, uma explicação de cada um dos elementos:

- **Arquivos do projeto**: um explorador de arquivos do projeto. Tipicamente, é possível criar, acessar, modificar, mover e deletar os scripts e demais arquivos com que você está trabalhando.
- **Editor de texto**: área principal de uma IDE. Aqui, você pode editar o código de um arquivo. Muitas IDEs oferecem abas ou múltiplas telas no editor de texto, para facilitar o trabalho com diversos arquivos de código.
- **Executar e debugar (ou depurar)**: botões que permitem rodar o código do arquivo atual, ou testá-lo através do *debugger* (depurador) de código. Falaremos mais do *debugger* nas próximas aulas!
- **Controle de versão**: região responsável por integrar seu código com um sistema de versionamento. Neste curso, nós não nos aprofundaremos nesta funcionalidade, mas explicando em poucas palavras: versionamento de código significa criar versões de cada alteração feita no seu código, de forma que seja possível revisar e voltar no histórico, caso necessário. O sistema de versionamento de código mais famoso é o **git**, e a principal ferramenta para hospedá-lo e compartilhar código entre diversos desenvolvedores é o **GitHub**.
- **Console de Python**: Ambiente rodando um interpretador de Python, a partir do qual é possível inserir e testar comandos. Equivale a digitar `python` no CMD. Em uma IDE, sua função é ajudar a testar código rapidamente, para que seja possível integrá-lo rapidamente ao código do editor de texto.
- **Terminal do sistema operacional**: terminal equivalente ao CMD do Windows ou terminal de Mac e Linux. Permite acesso rápido a comandos do sistema, como já vimos anteriormente.

Vamos agora entender como instalar e configurar as principais IDEs de Python!

11. Instalando e configurando o VS Code

O Visual Studio Code (VS Code) é um editor de texto criado pela Microsoft. Em sua instalação base, ele funciona apenas como editor de texto de fato. É preciso usar extensões do VS Code para transformá-lo em uma IDE, capaz de executar código Python (e de outras linguagens também).

Instalação

Baixe o VS Code do site oficial. São oferecidos instalados para Windows, Mac e Linux:

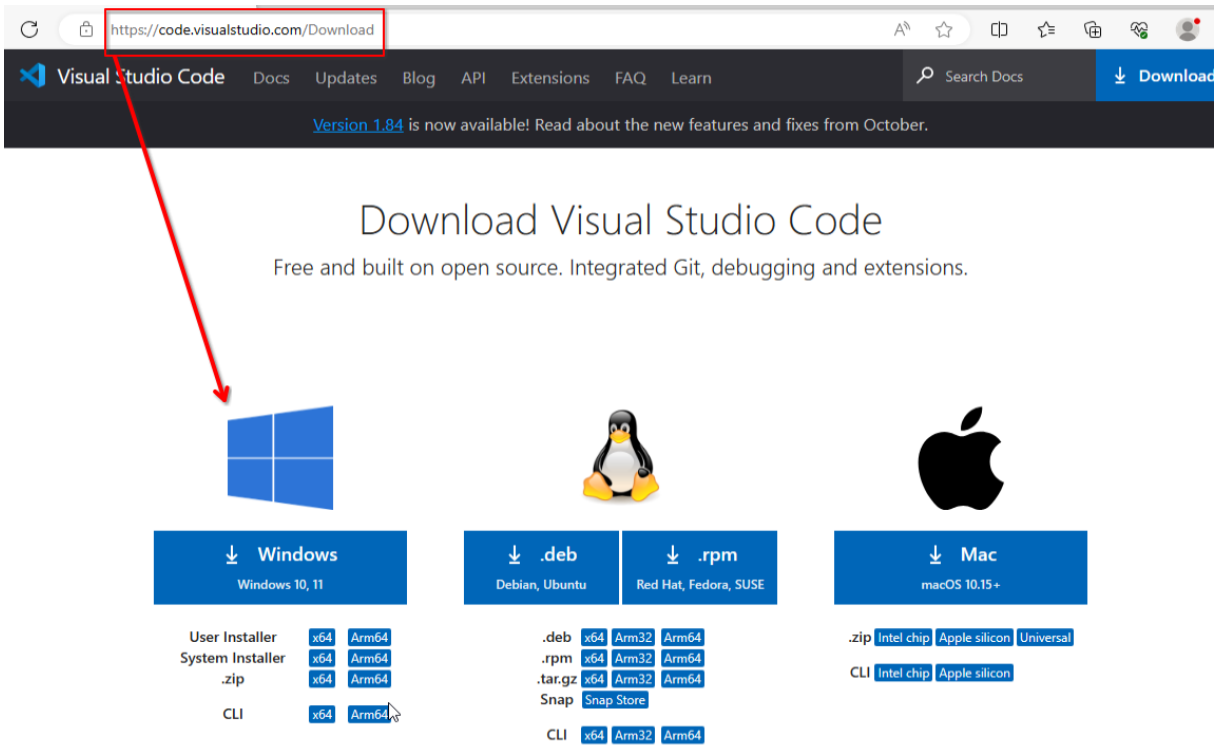


Figure 72: Download do VS Code

Abra o instalador e vá seguindo pelas telas:

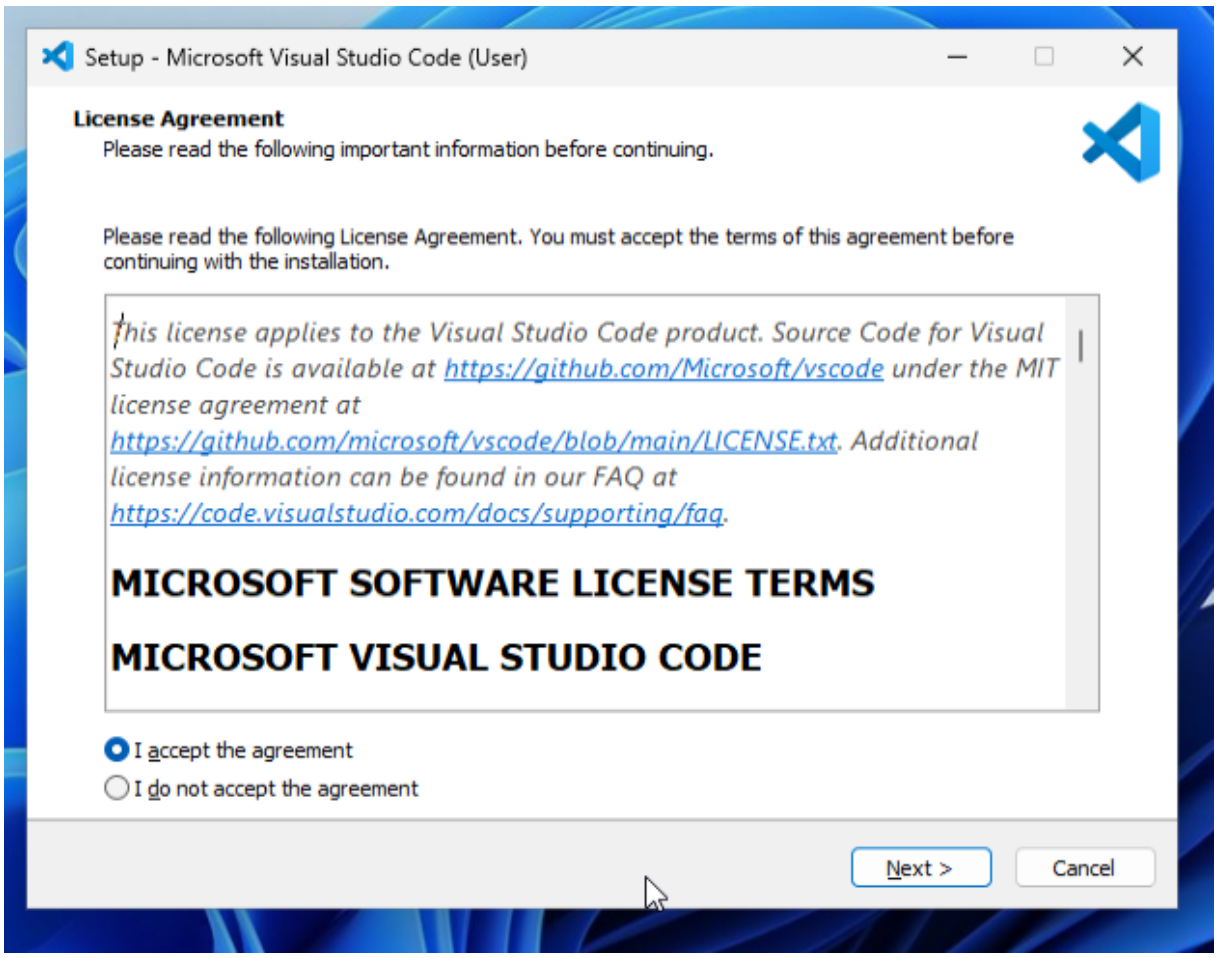


Figure 73: Instalador do VS Code

Na tela abaixo, marque as caixinhas para permitir que o VS Code seja aberto a partir do navegador de arquivarm os, e também para vincular a abertura de scripts diretamente com o VS Code:

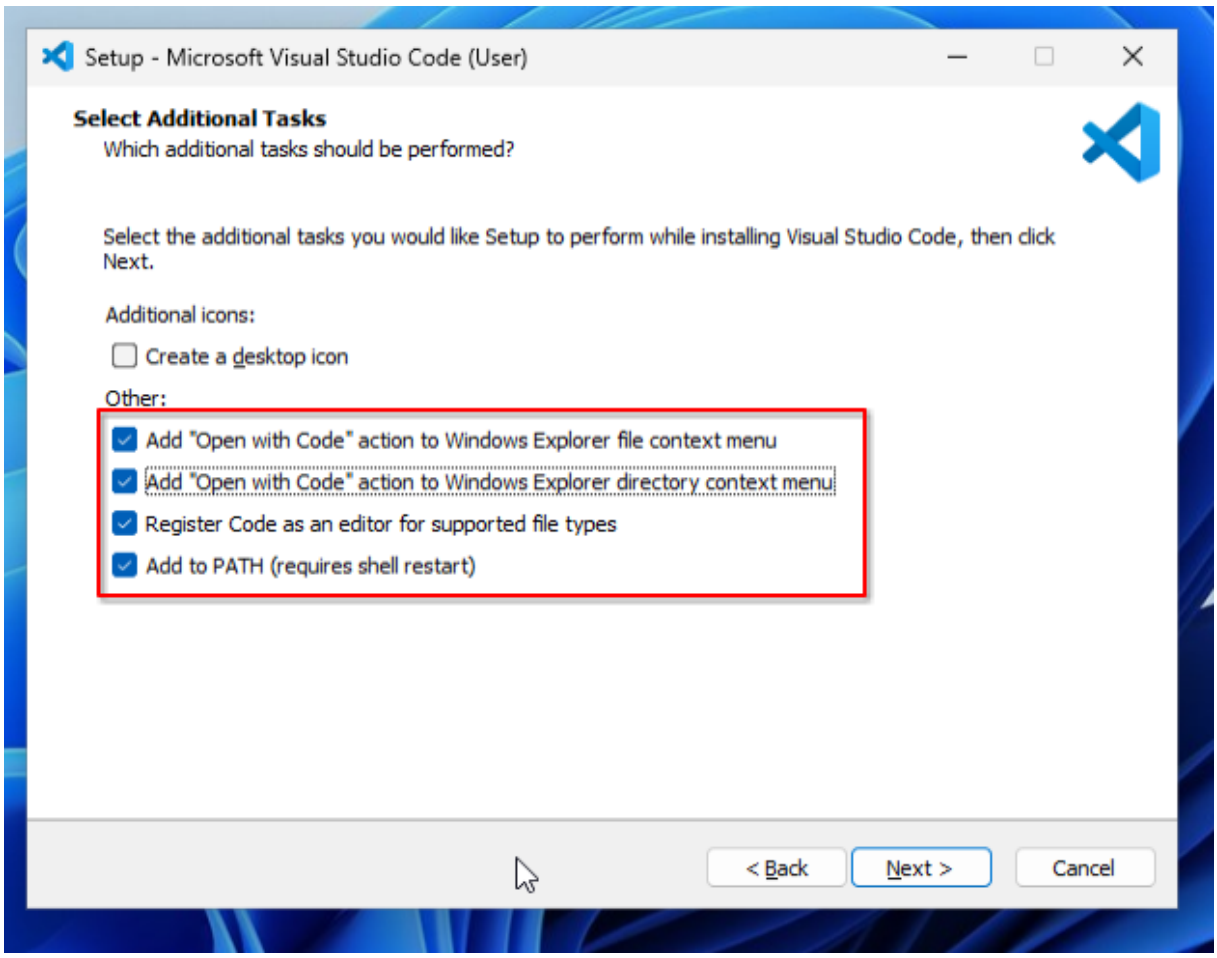


Figure 74: Opções de instalação do VS Code

Uso básico do VS Code

Com a instalação concluída, abra o VS Code. Você deve chegar na tela abaixo (pode explorar as opções de boas-vindas se desejar):

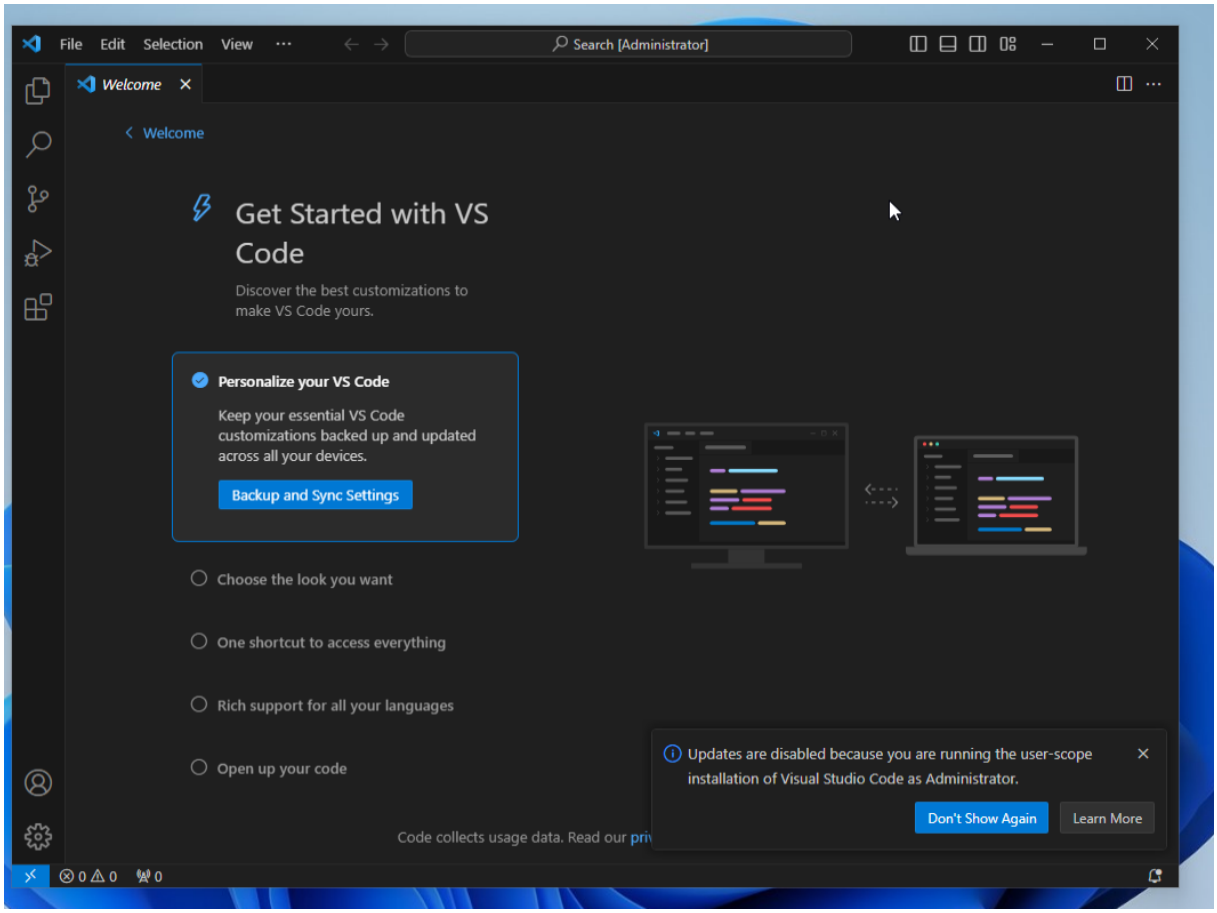


Figure 75: Tela inicial do VS Code

Clique em `Arquivo > Novo Arquivo de Texto` e escreva qualquer coisa no arquivo. Note que o tipo de arquivo identificado por padrão é texto sem formatação (*Plain Text*):

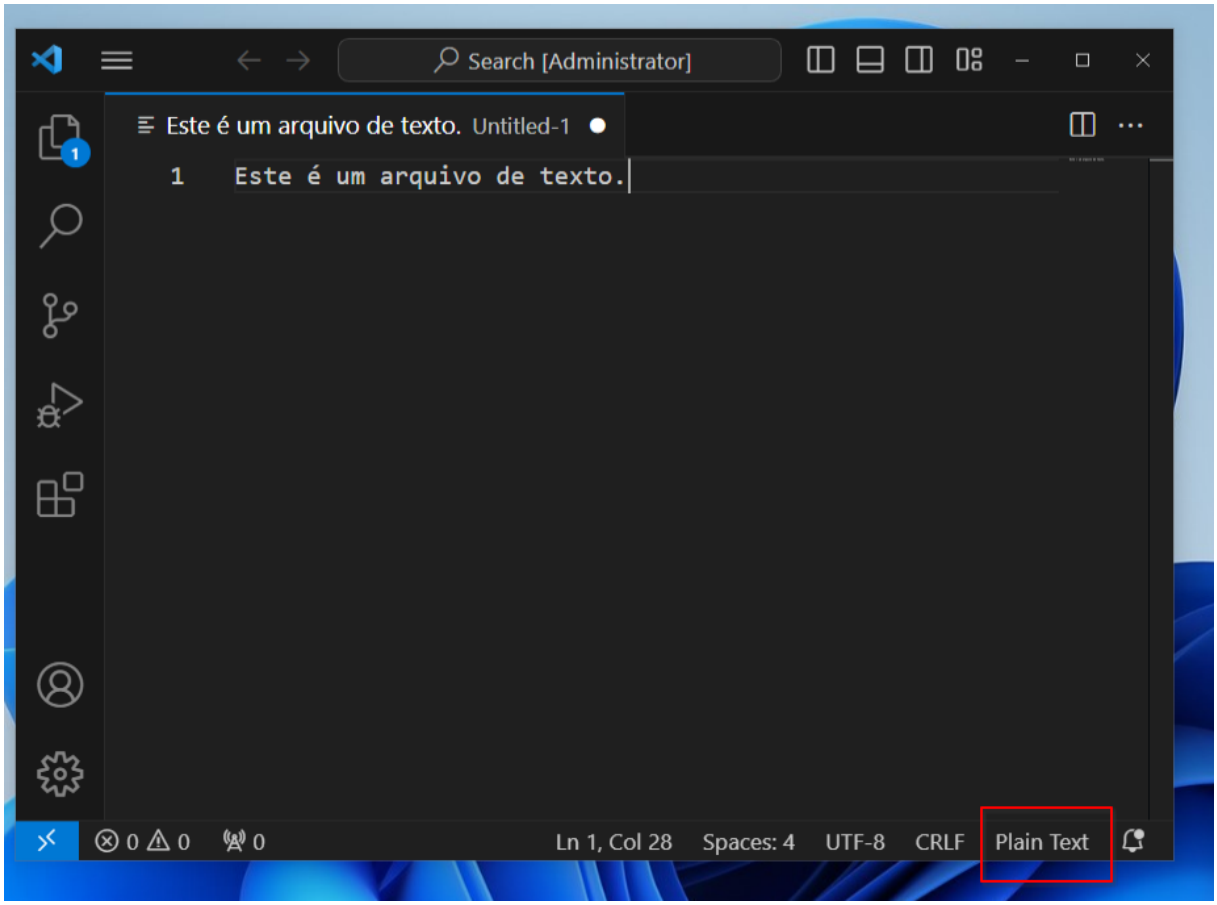


Figure 76: Arquivo de texto simples

Clique em cima do botão `Texto sem Formatação` e busque por Python. Com isso, o arquivo será identificado como um script de Python:

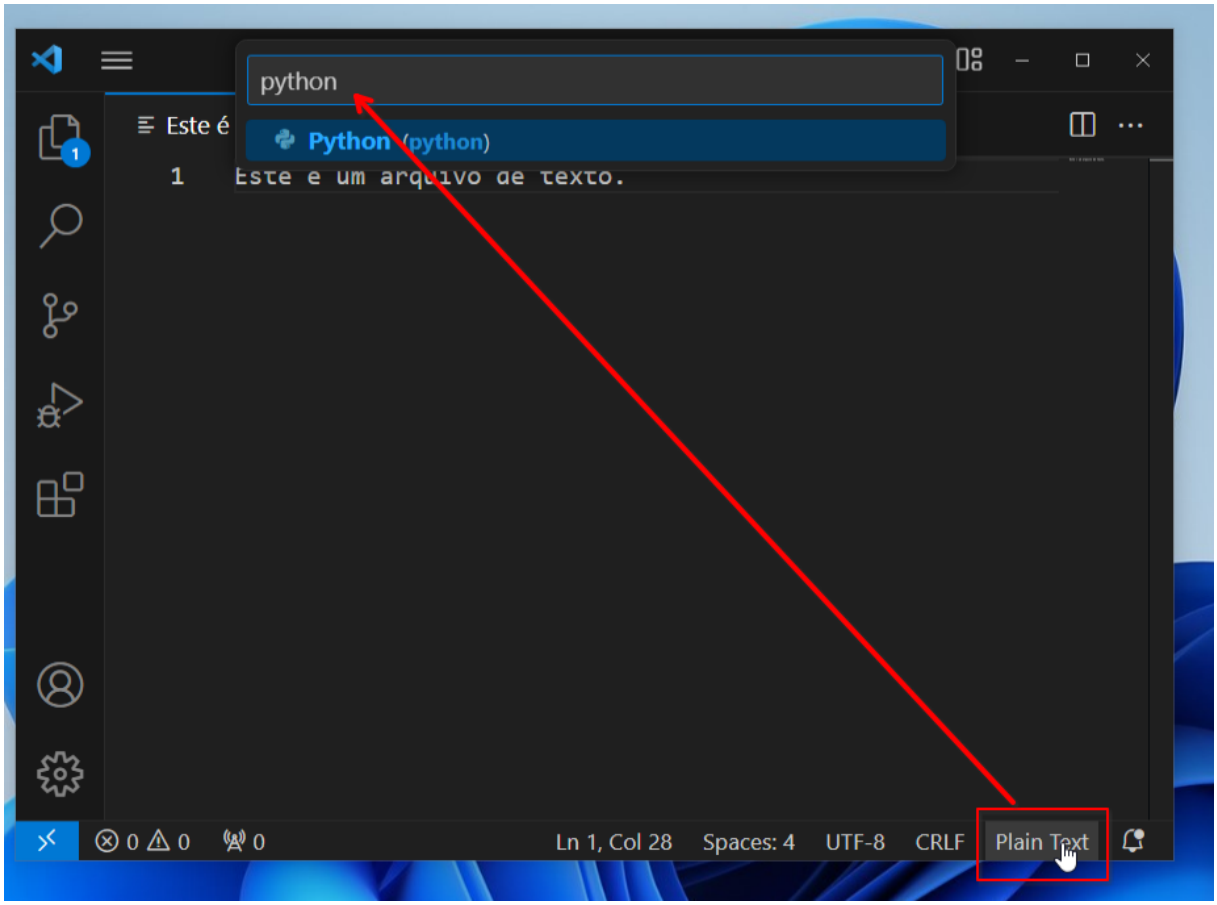


Figure 77: Escolhendo a sintaxe de Python

Modifique o texto para código Python (use a função `print()` por exemplo) e veja que o texto muda de cor para ajudar a escrever o código!

Salve o script (Atalho `Ctrl + S`) no Desktop. Lembre-se de salvá-lo como um script de Python (extensão `.py`):

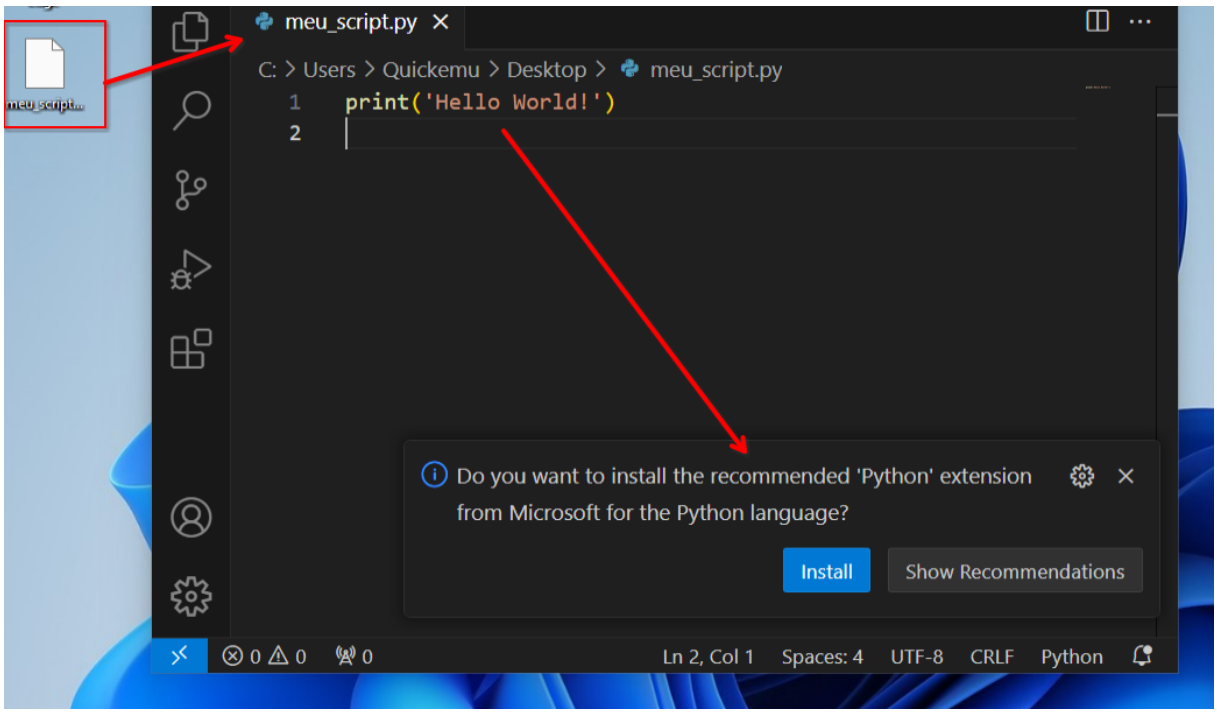


Figure 78: Salvando um script de Python

Durante os passos acima, o VS Code provavelmente sugeriu a instalação da extensão de Python. Isso porque sem ela, é impossível executar código Python no VS Code (lembre-se de que ele é apenas um editor de texto).

Configuração para código Python

Clique nos ícones de extensões (quadrados empilhados na barra à esquerda), digite Python na caixa de busca, e baixe a extensão de Python feita pela Microsoft (provavelmente será a primeira da lista e com o maior número de downloads):

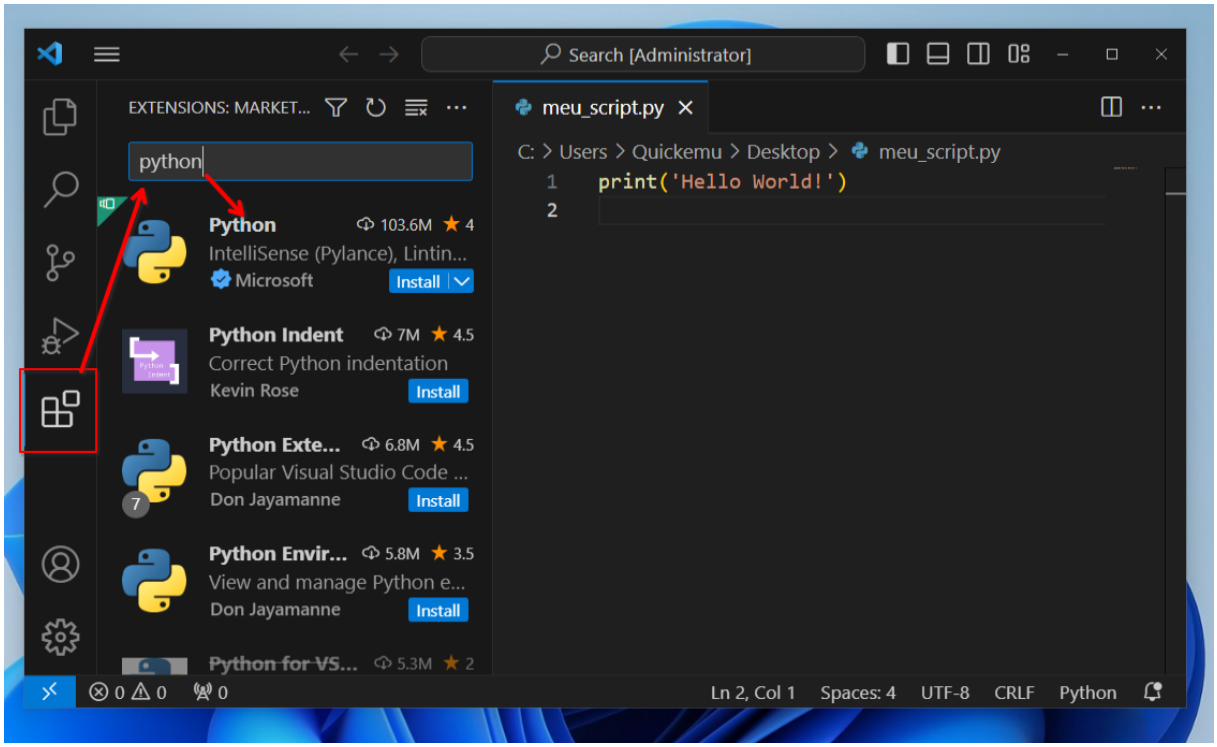


Figure 79: Instalando a extensão de Python do VS Code

Após a instalação, um botão no formato de seta deverá aparecer no canto superior direito. Clique no menu em dropdown ao lado do botão, e em seguida na opção “Executar Arquivo Python” para rodar seu script:

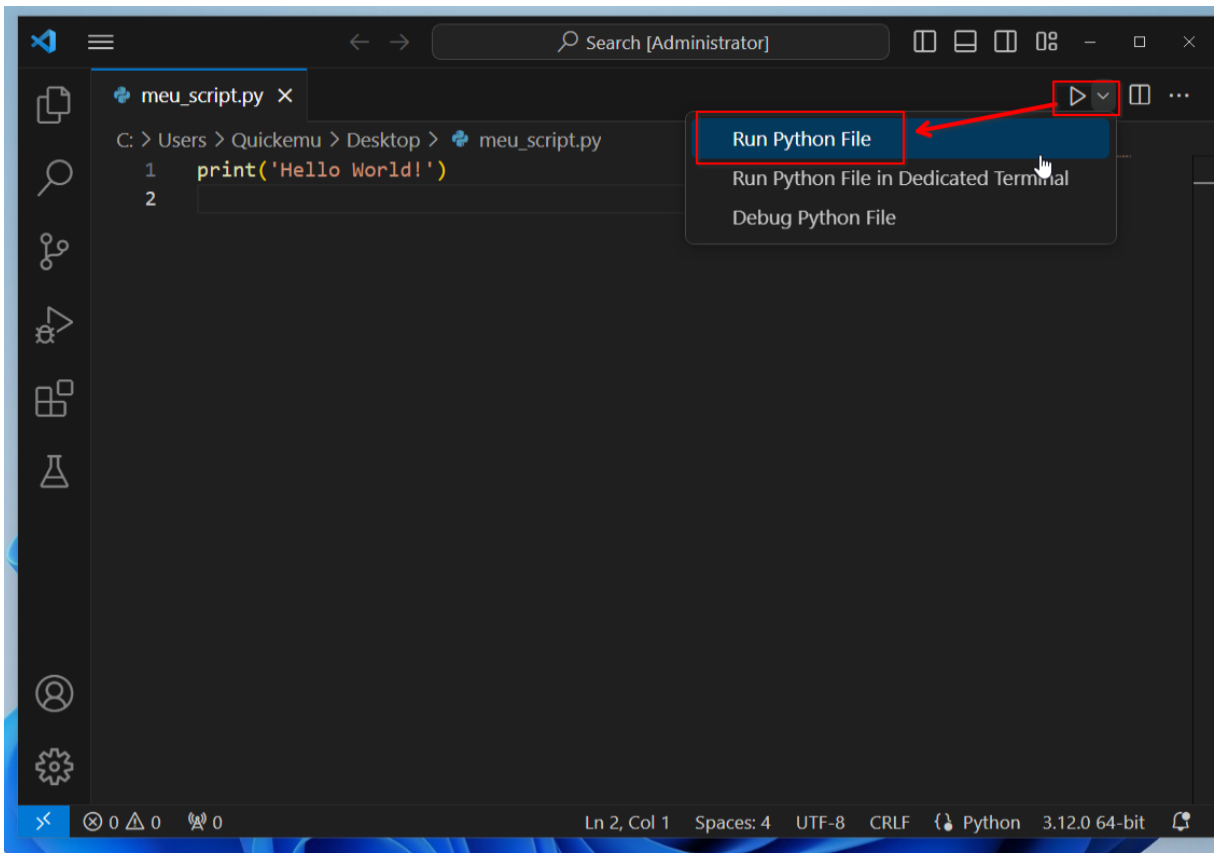


Figure 80: Executando um script de Python

Ao executar o código, uma aba com um terminal aparecerá na parte de baixo. Este terminal é usado para rodar o código e exibir seu output:

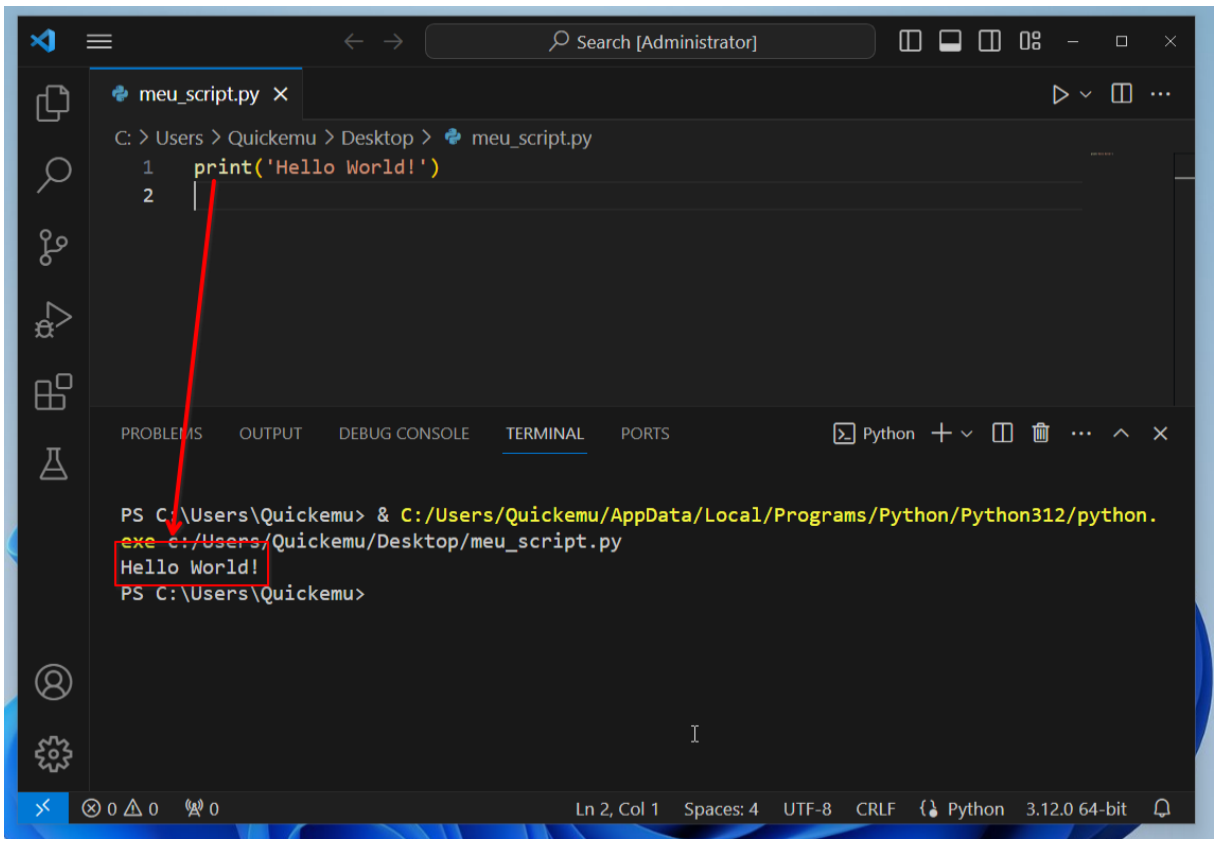


Figure 81: Output do script

Com a tela dividida entre editor e terminal, é possível abrir novos terminais, checar erros, modificar as abas, … Experimente com os botões da interface para entender como funcionam!

Depuração de código

A extensão Python do VS Code adiciona também a funcionalidade de debugar ou depurar código. Com isso, você consegue "pausar" o código em uma linha qualquer e acompanhar o que acontece com as variáveis a cada linha.

Para usar o debugger, é preciso adicionar um **break point** no seu código. Este ponto marca a posição em que o debugger deverá pausar o código. Clique ao lado do número da linha para adicionar um break point a ela (bola vermelha):

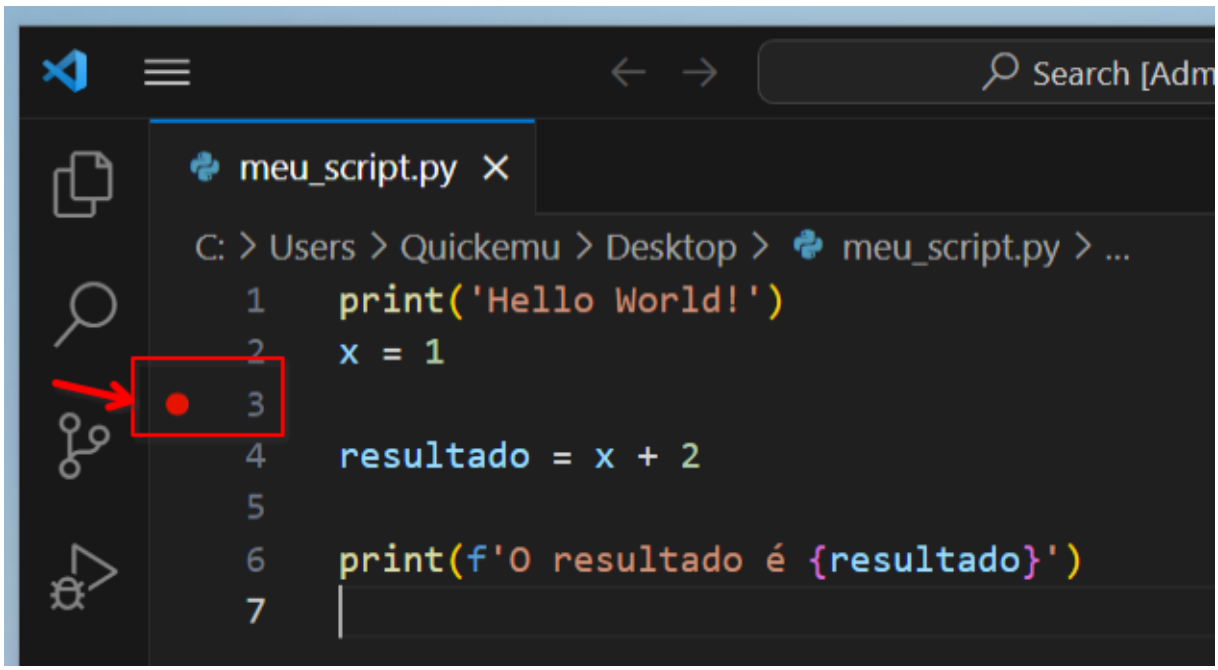


Figure 82: Adicionando um break point no código

Em seguida, no mesmo dropdown de execução que usamos anteriormente, selecione "Depurar Arquivo Python":

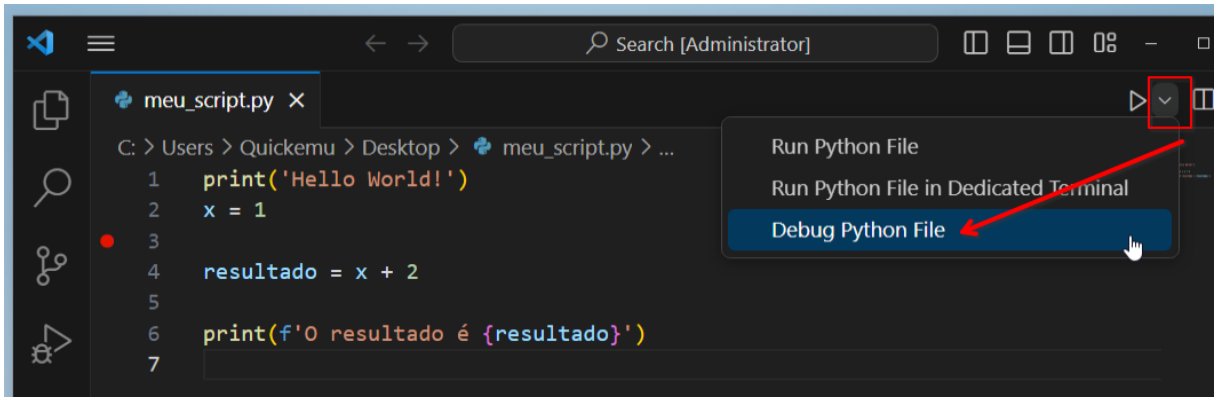


Figure 83: Depurando um script

Note que o código no editor ficará pausado na linha contendo o break point. Uma cor amarela no fundo da linha indica isso.

As janelas do debugger deverão aparecer na lateral e na base do editor, e o controlador para avançar para próxima linha fica no topo:

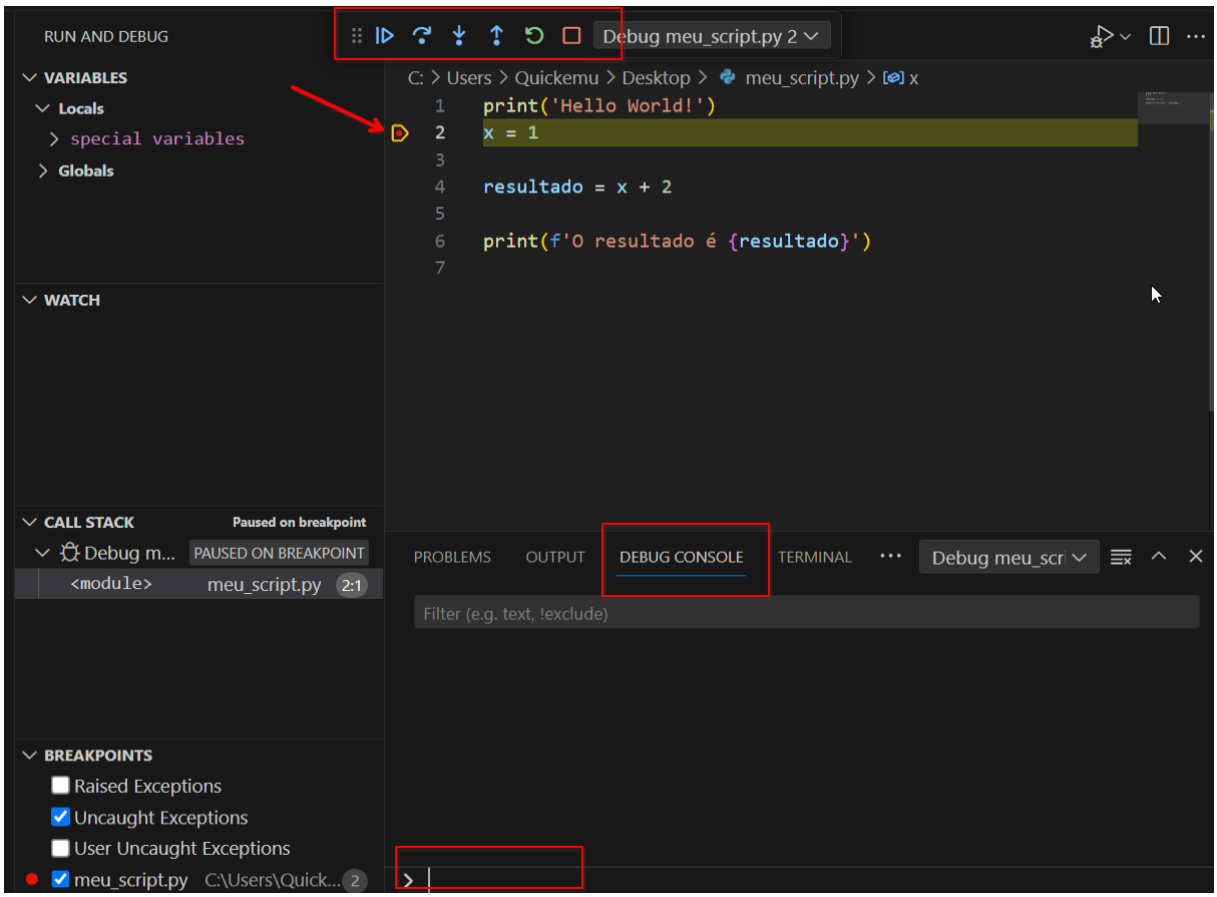


Figure 84: Os controles de depuração

Na aba de “Debug Console”, é possível executar código acessando as variáveis do script (até o ponto em que foi executado). Avance algumas linhas e veja que o valor das variáveis pode ser acessado neste console:

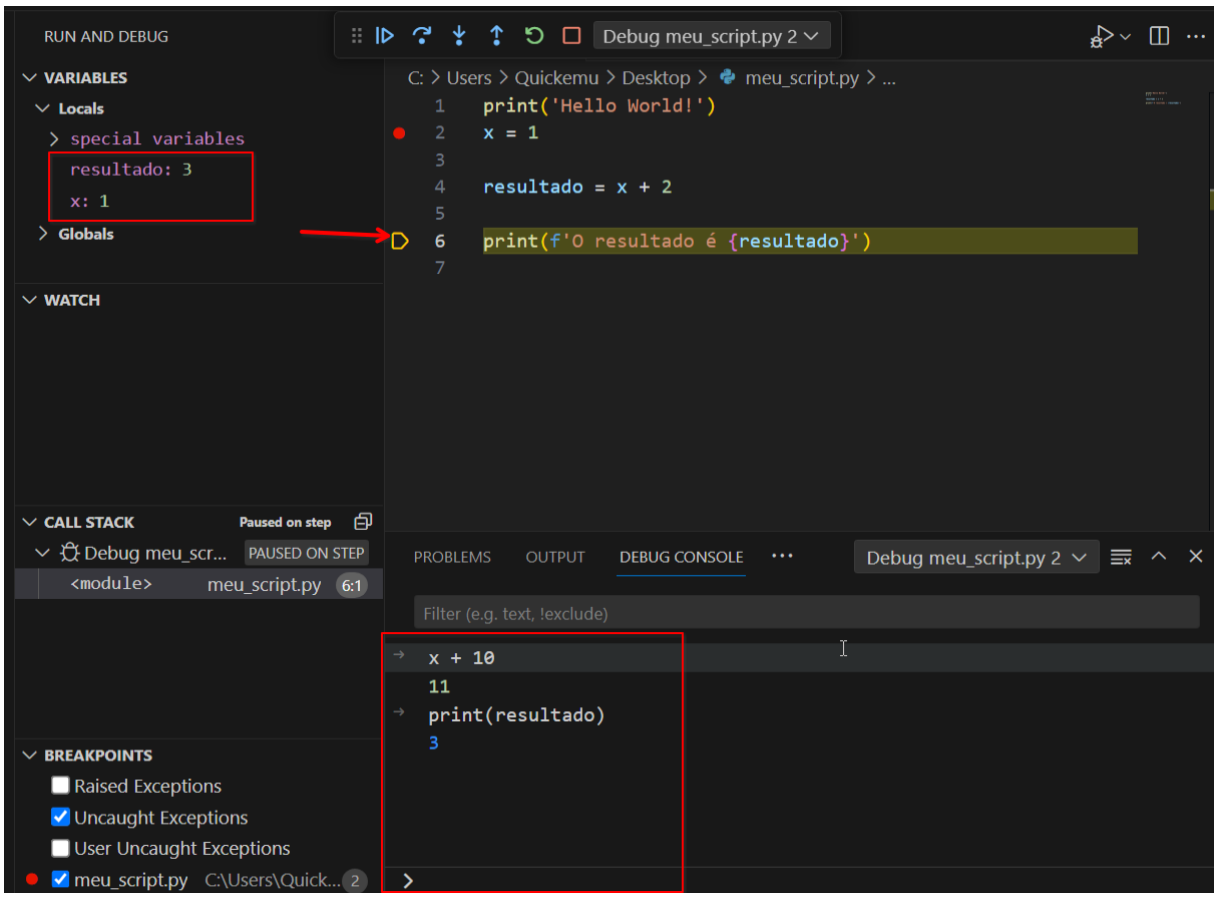


Figure 85: Acessando variáveis no código em depuração

Usando o debugger, é possível inspecionar o valor de qualquer variável durante a execução do código - muito mais interativo e prático que usar comandos de `print()` para exibir os valores!

Qual Python o VS Code está usando?

O VS Code está sempre executando com alguma instalação de Python do seu sistema. Se você está tentando importar pacotes como o `pandas`, mas na hora de executar no VS Code recebe erros de "Pacote não encontrado", provavelmente você instalou o pacote em um Python diferente que o usado no VS Code!

Para escolher qual Python utilizar, clique no número da versão de Python, no canto inferior direito do VS Code. Uma janela deverá abrir, exibindo todas as instalações de Python encontradas no seu sistema:

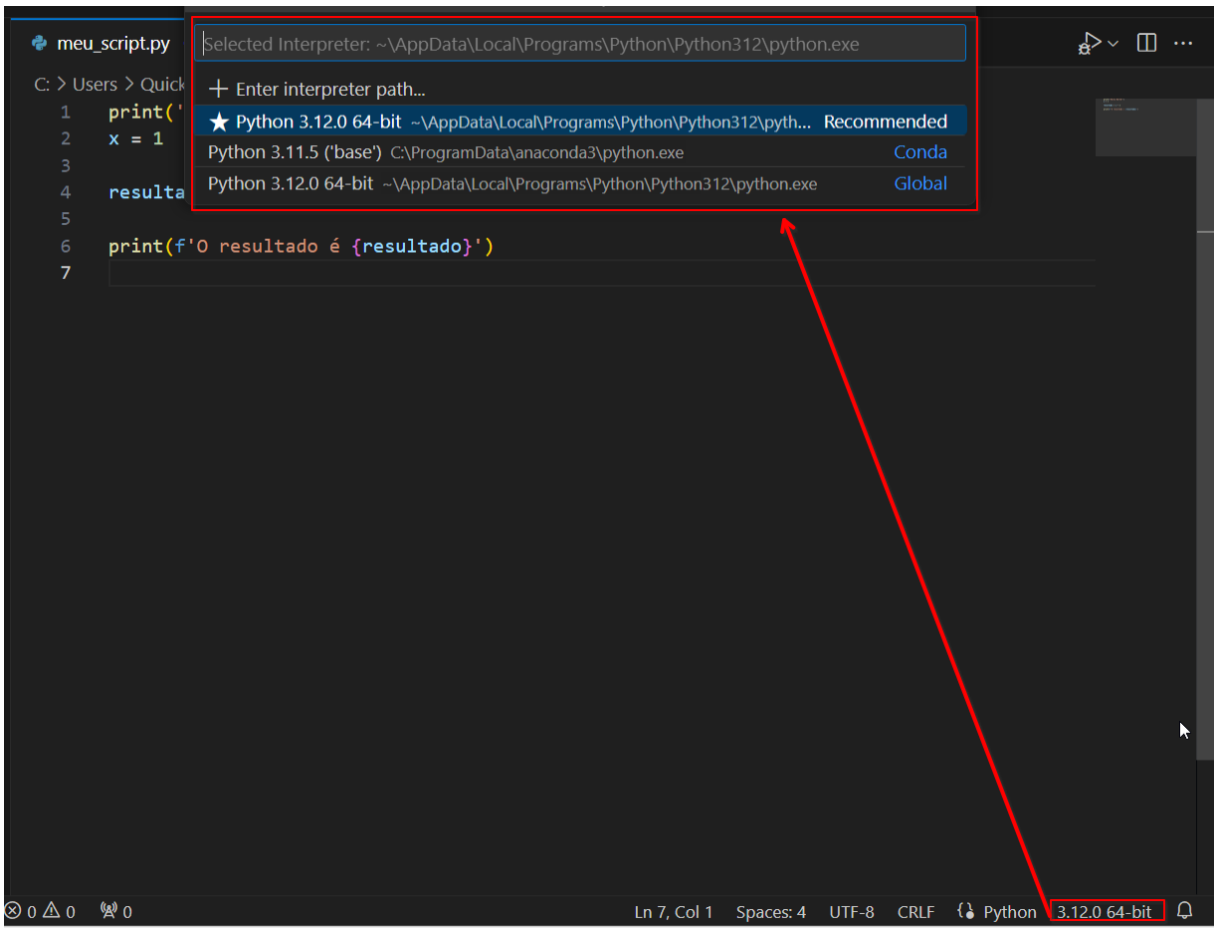


Figure 86: Escolhendo a instalação de Python

Se você ainda não encontrar a sua instalação desejada (possivelmente porque ela não está no PATH), é possível buscar pela instalação de Python no seu explorador de arquivos, ou digitar o caminho diretamente.

Configurações adicionais

Esta aula mostrou apenas as bases de como executar e depurar código Python no VS Code. Há ainda muitas extensões para testar e explorar! Se quiser algumas ideias adicionais, leia nosso artigo sobre como instalar e configurar o VS Code para Python.

12. Instalando e configurando o PyCharm

Enquanto o VS Code é uma ferramenta genérica, capaz de ser customizada para qualquer linguagem através de extensões, o PyCharm tem como foco o desenvolvimento em Python.

O PyCharm é desenvolvido pela empresa JetBrains, que possui uma grande quantidade de IDEs especializadas para linguagens diferentes.

Instalação

O PyCharm possui duas versões: Community (gratuita) e Professional (paga). A versão paga possui um teste grátis de 30 dias, e também é gratuita por 1 ano para estudantes com email institucional ativo.

Naturalmente, existem algumas diferenças entre a versão gratuita e paga. Uma das principais é a integração com Notebooks do Jupyter (que veremos mais pra frente). Mas para quem está começando, a versão gratuita dá conta de todas as demandas.

Para instalar, acesse a página de download do PyCharm e baixe a versão desejada (neste exemplo usaremos a gratuita):

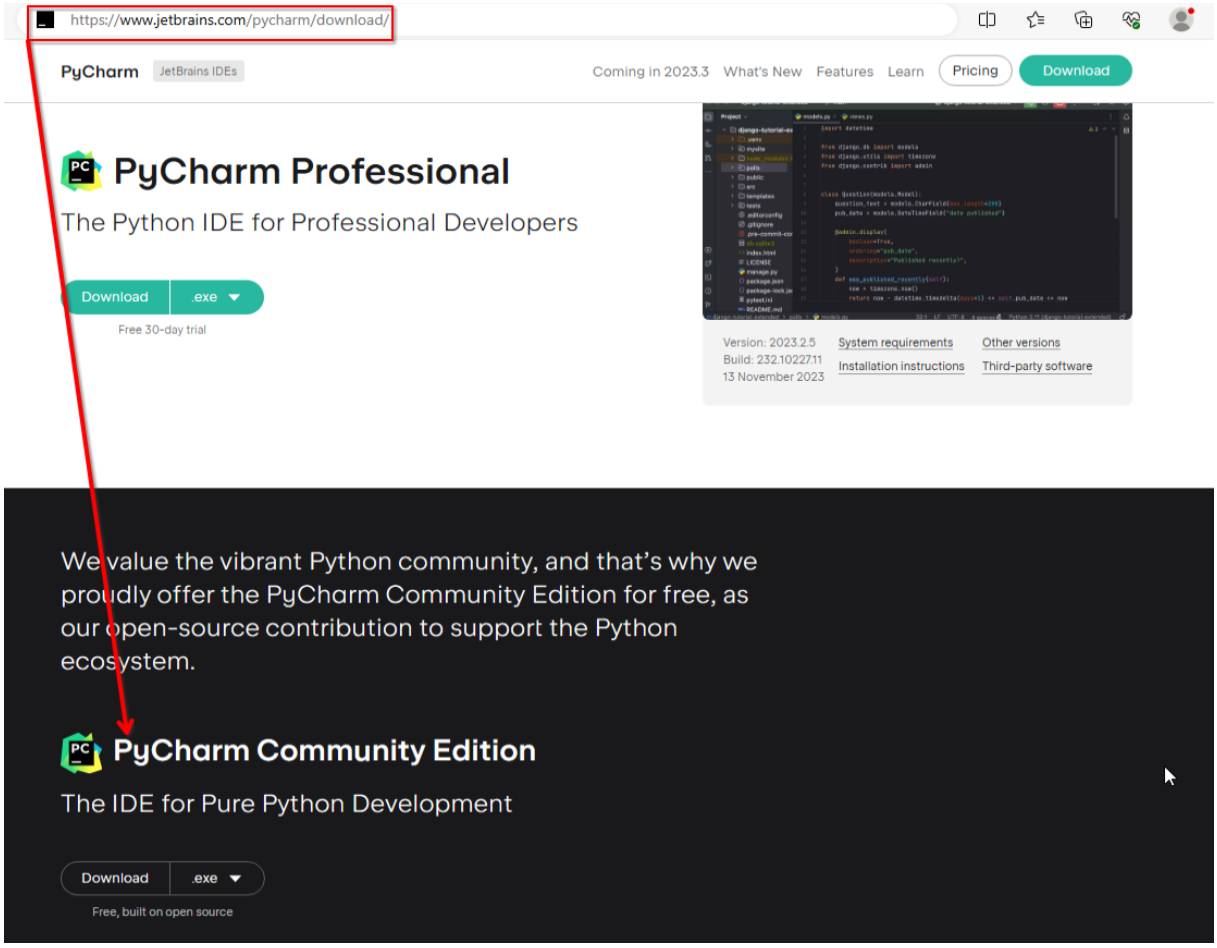


Figure 87: Download do PyCharm

Em seguida, siga os passos no instalador:

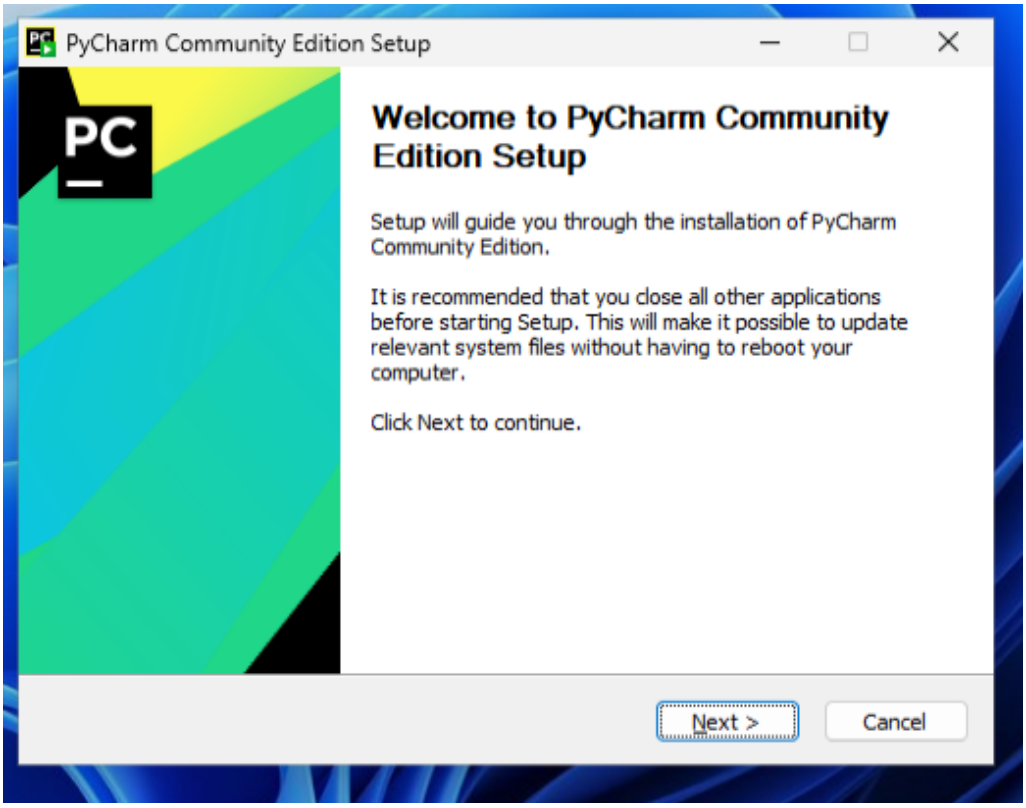


Figure 88: Instalador do PyCharm

Criando um projeto no PyCharm

Como o PyCharm já é pensado para desenvolvimento Python, a configuração necessária é mínima. Por outro lado, o PyCharm segue a lógica de **Projetos**, em que antes de abrir um script e começar a programar, é preciso determinar uma pasta para servir como "base" para o seu projeto.

Na tela introdutória abaixo, clique em "New Project":

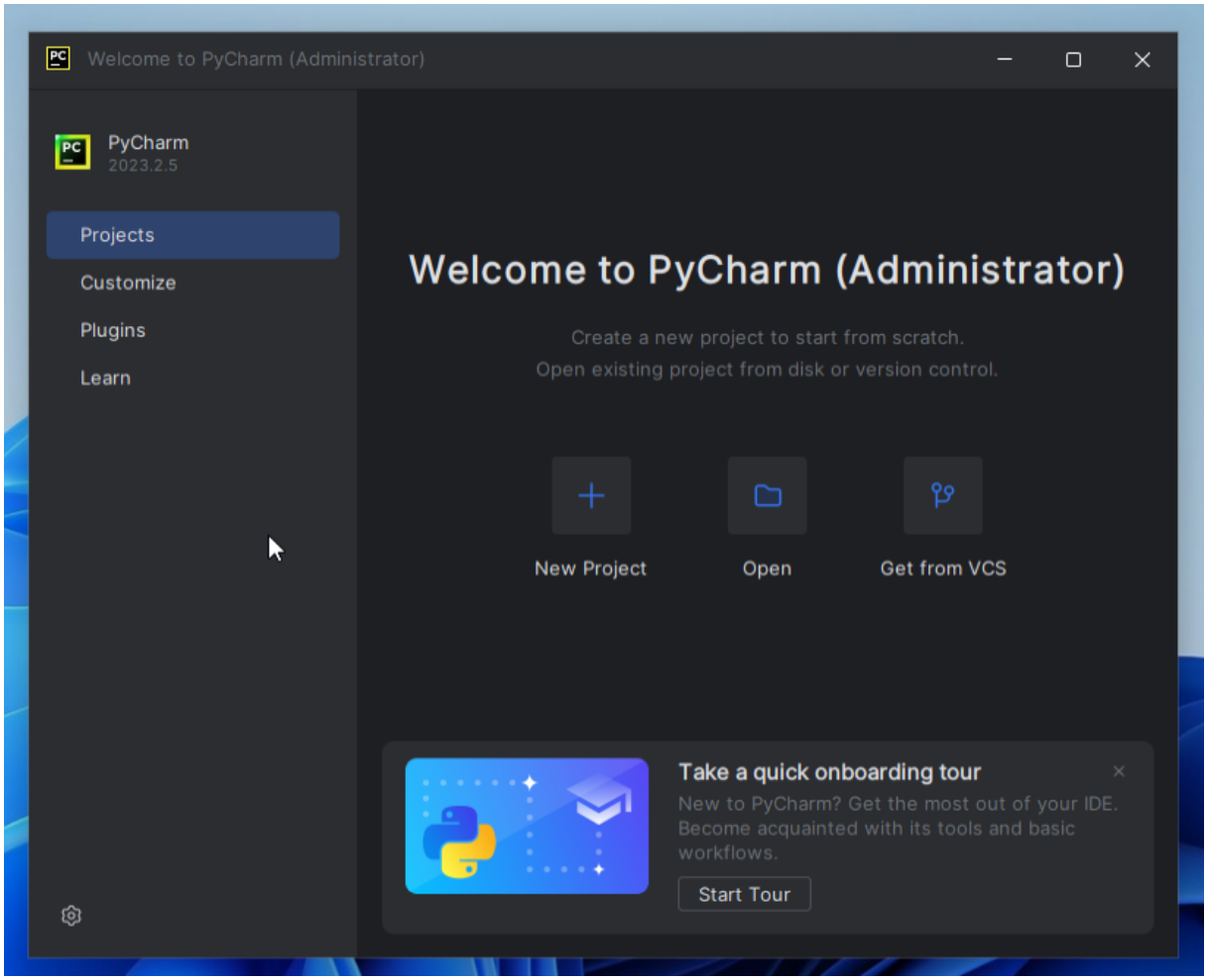


Figure 89: Tela de abertura do PyCharm

O PyCharm irá sugerir uma pasta onde criar seu projeto (modifique caso desejar). Além disso, ele irá sugerir criar um novo **Ambiente Virtual** vinculado ao seu projeto:

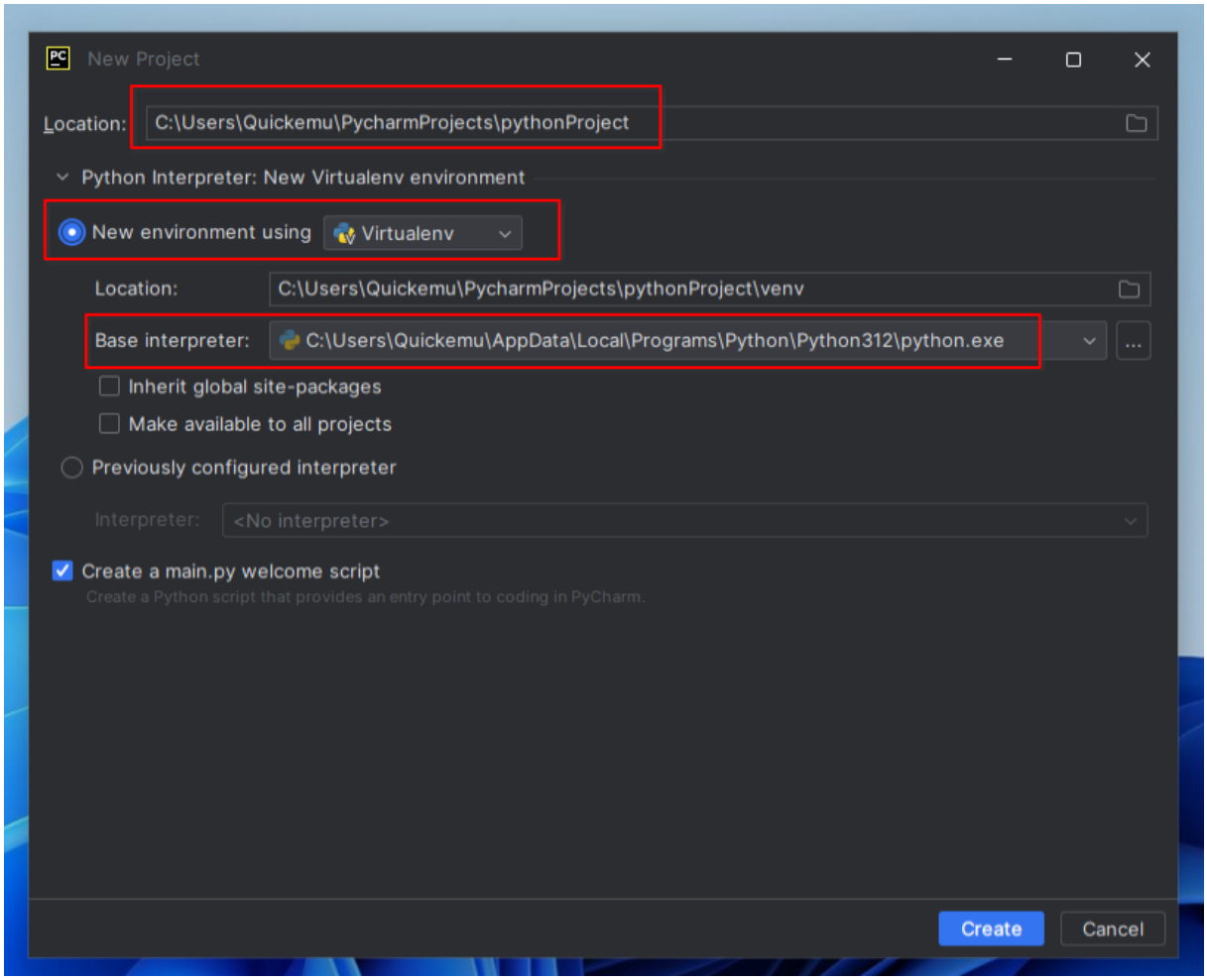


Figure 90: Criando um novo projeto

Um ambiente virtual funciona como uma cópia da sua instalação de Python, onde os pacotes instalados não afetam a sua instalação principal. Isso evita problemas com dependências incompatíveis quando você tiver múltiplos projetos de Python no mesmo computador.

Neste exemplo, vamos seguir sem ambientes virtuais, mas considere utilizá-los nos seus projetos futuros. Para escolher a instalação principal de Python, marque a opção `Previously configured interpreter`, e em seguida clique em `Add Interpreter > Add Local interpreter...`:

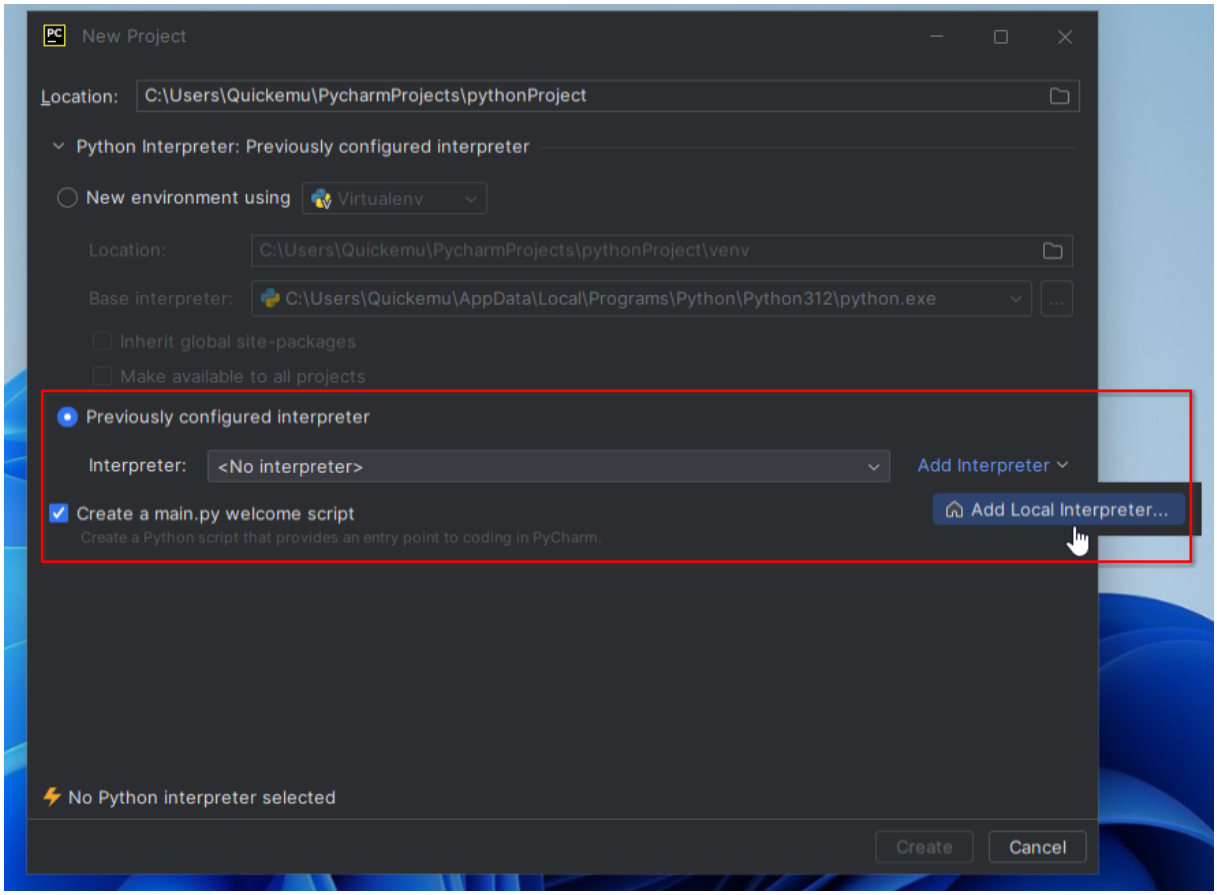


Figure 91: Escolhendo um interpretador de Python

Na janela que abrir, escolha `System Interpreter`, e em seguida a instalação de Python desejada:

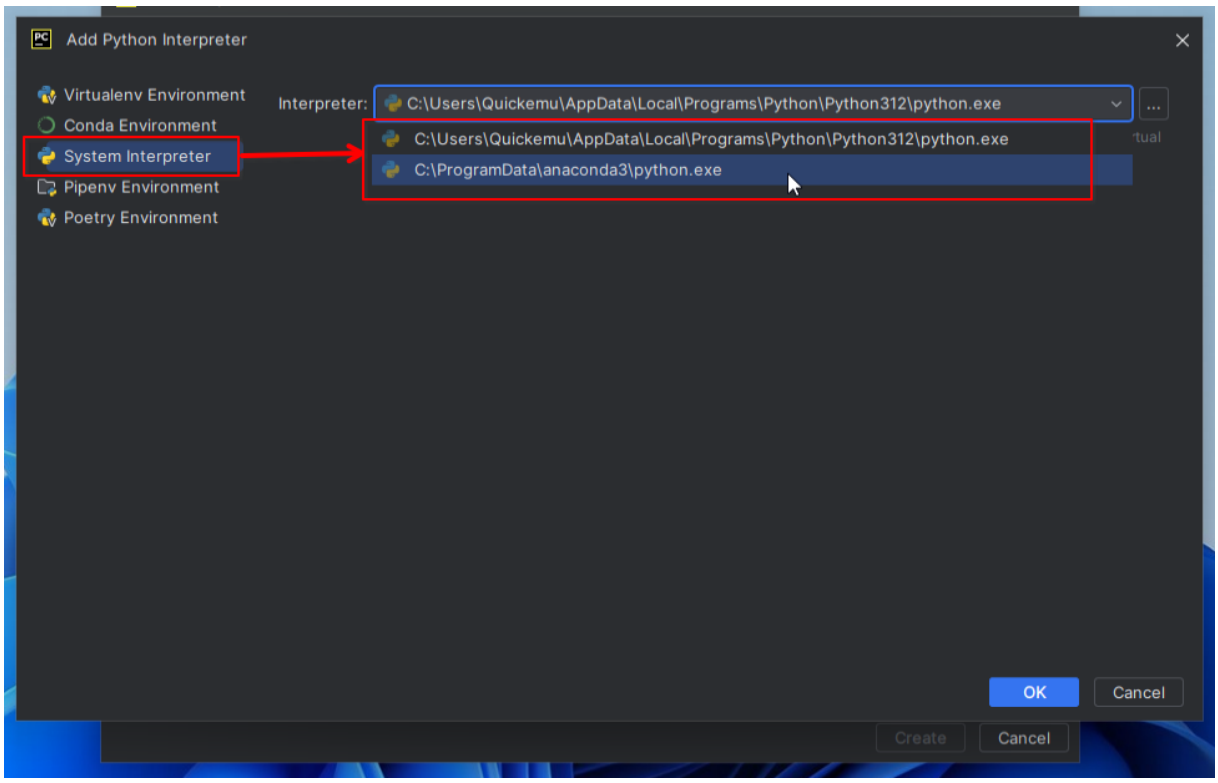


Figure 92: Escolhendo o interpretador do sistema

Uso básico do PyCharm

Ao criar o projeto, você verá a tela abaixo, já com um script chamado `main.py`:

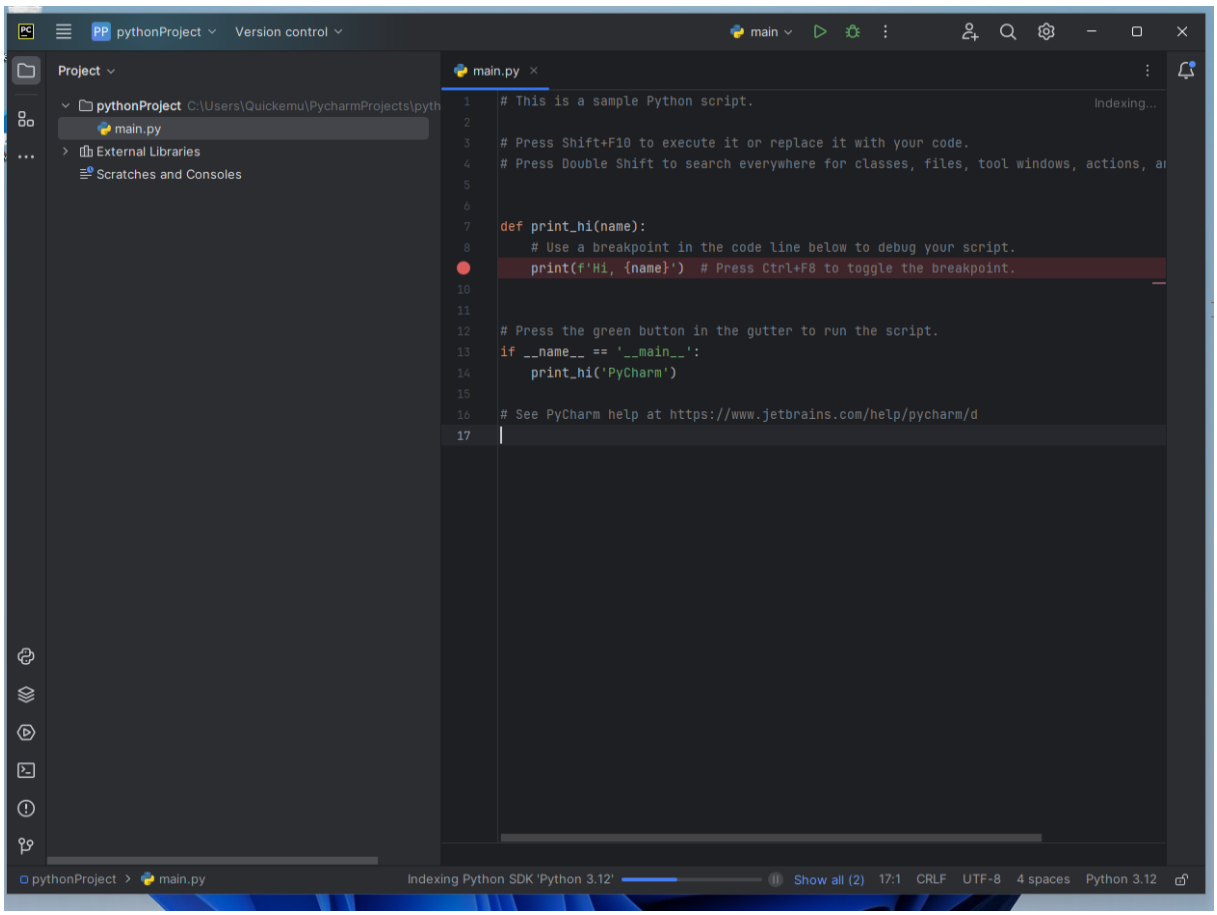


Figure 93: Tela inicial do projeto aberto

Delete o conteúdo dele e escreva algum código qualquer. Você vai perceber de cara que o PyCharm é mais estrito quanto à escrita de código: ele gera avisos até mesmo de erros de linhas em branco excessivas, ou erros de ortografia:

Figure 94: Script com avisos do PyCharm

Os botões de executar e depurar o código estão no canto superior direito:

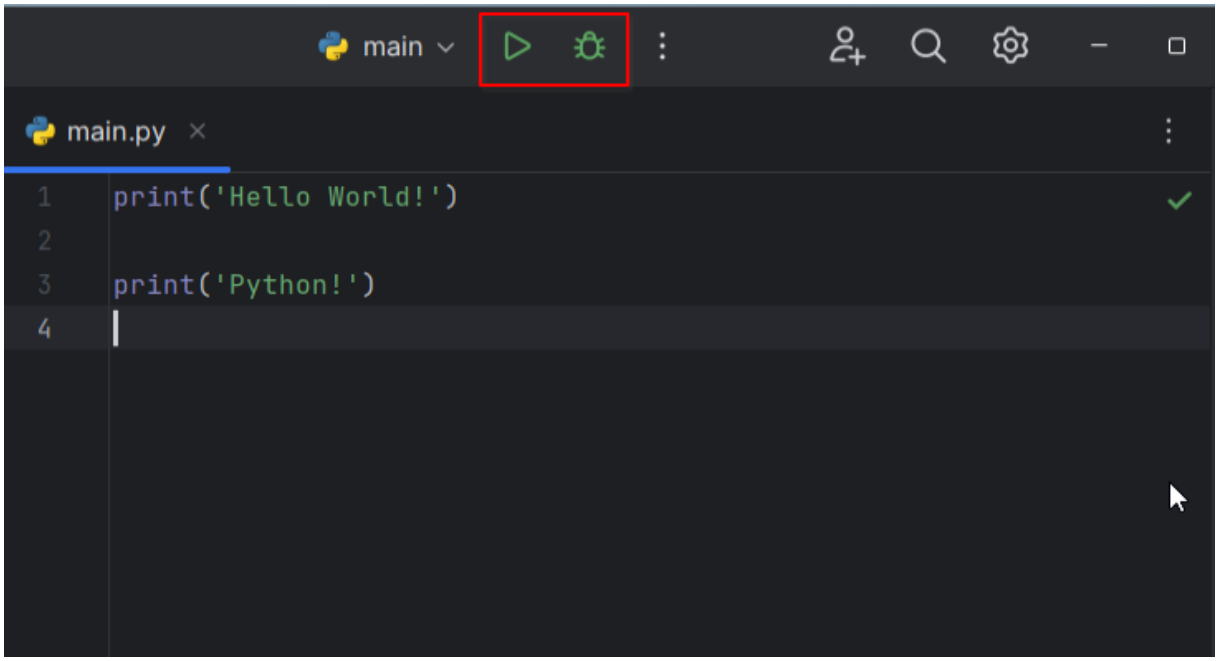


Figure 95: Botões de execução e depuração

Executando o código, o resultado aparece no terminal abaixo:

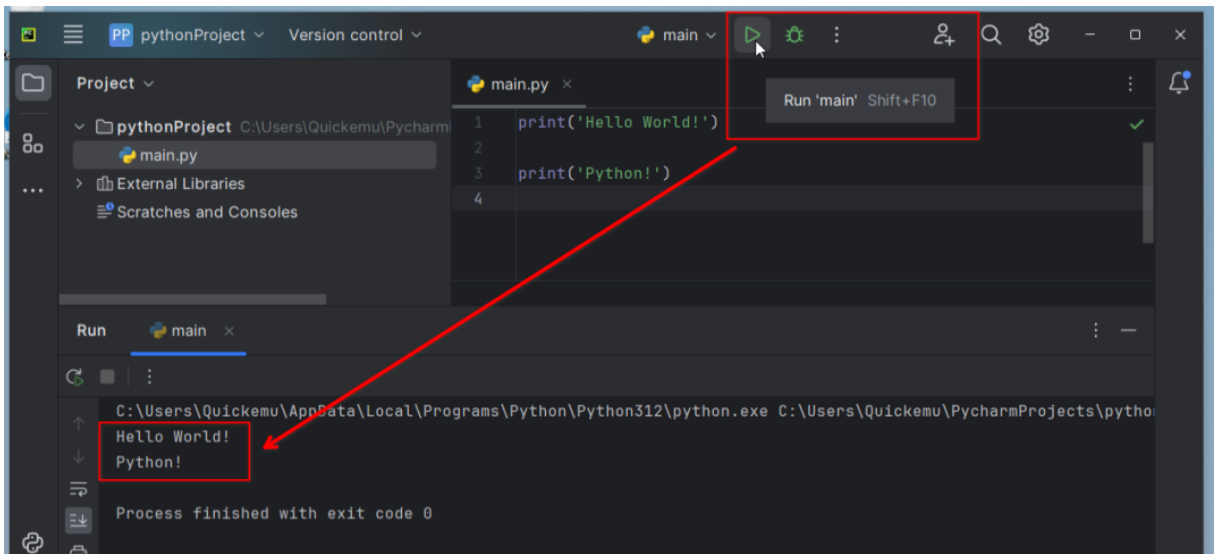


Figure 96: Rodando o script

Clicando o botão de depurar, menus com funcionalidade similar à depuração do VS Code aparecem:

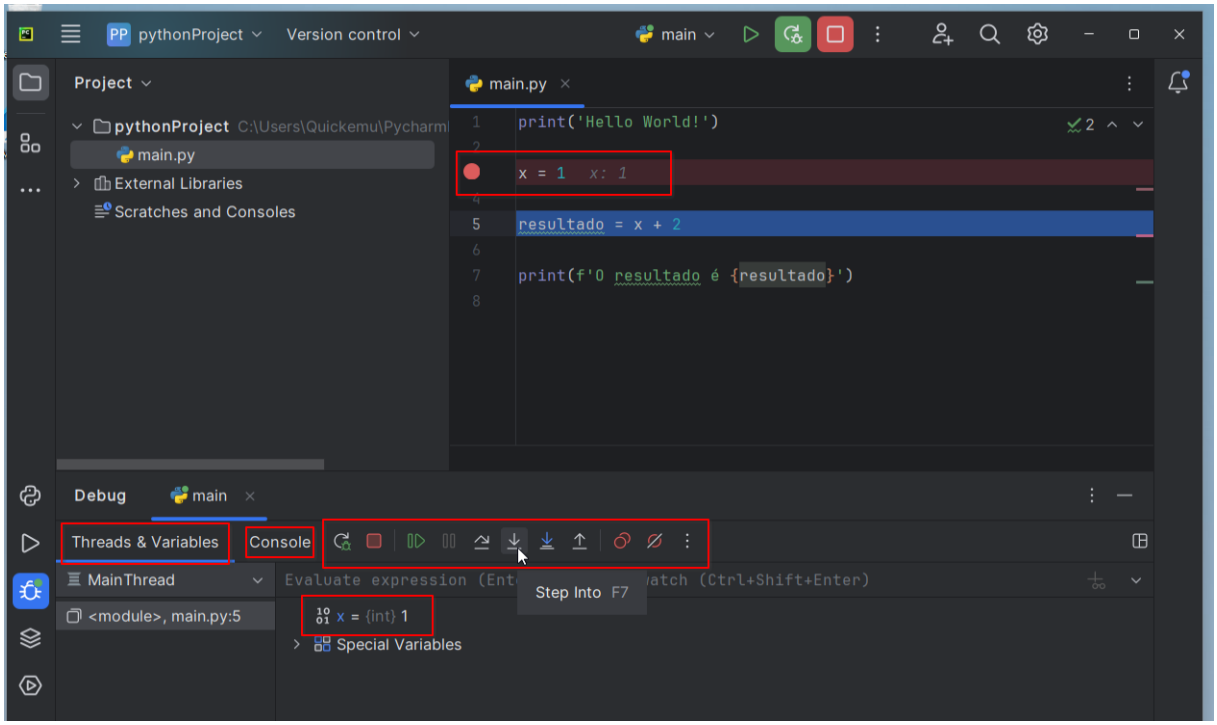


Figure 97: Depurando o script

Configurações diversas do PyCharm

É possível editar a forma de rodar um script clicando no dropdown ao lado do nome do arquivo executado e selecionando `Edit Configurations...` — o menu resultante permite escolher a pasta de onde executar, que script rodar, comandos ou tarefas adicionais, ... É possível deixar múltiplos scripts pré-configurados para execução a partir deste menu:

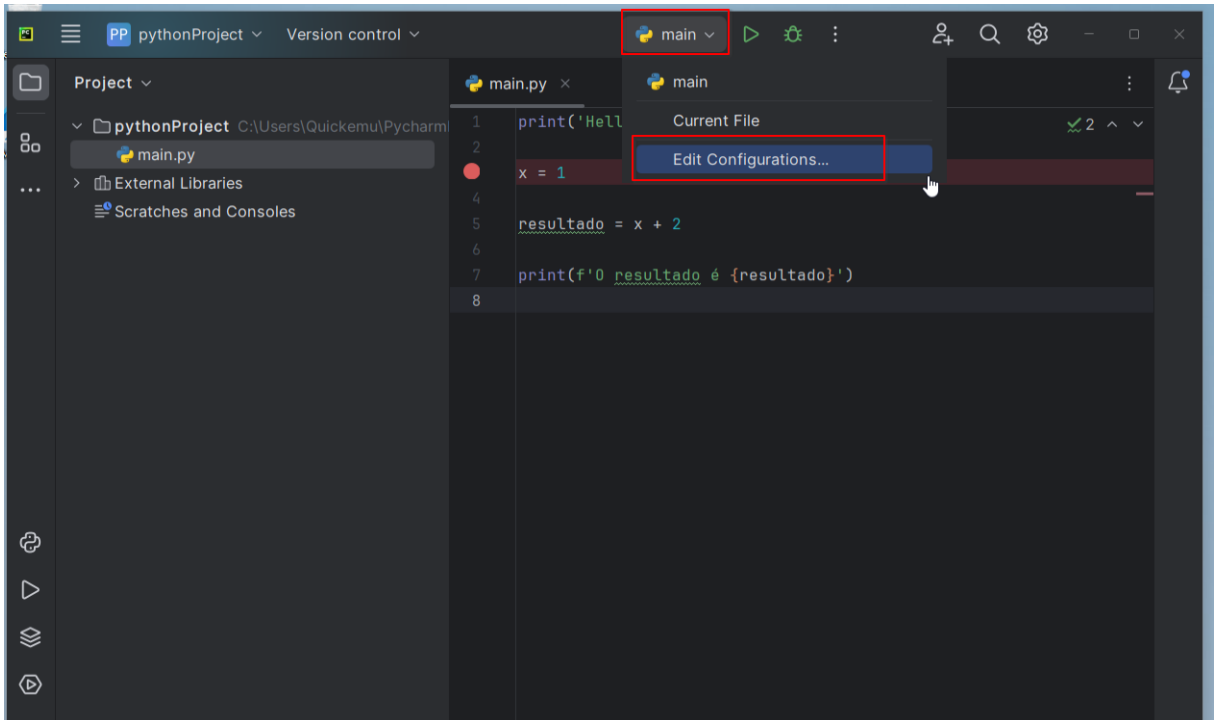


Figure 98: Configurações de execução

Como explicado anteriormente, a instalação de Python usada é configurada para o projeto como um todo. Para modificá-la, é possível acessar o menu no canto inferior direito, clicando na versão de Python:

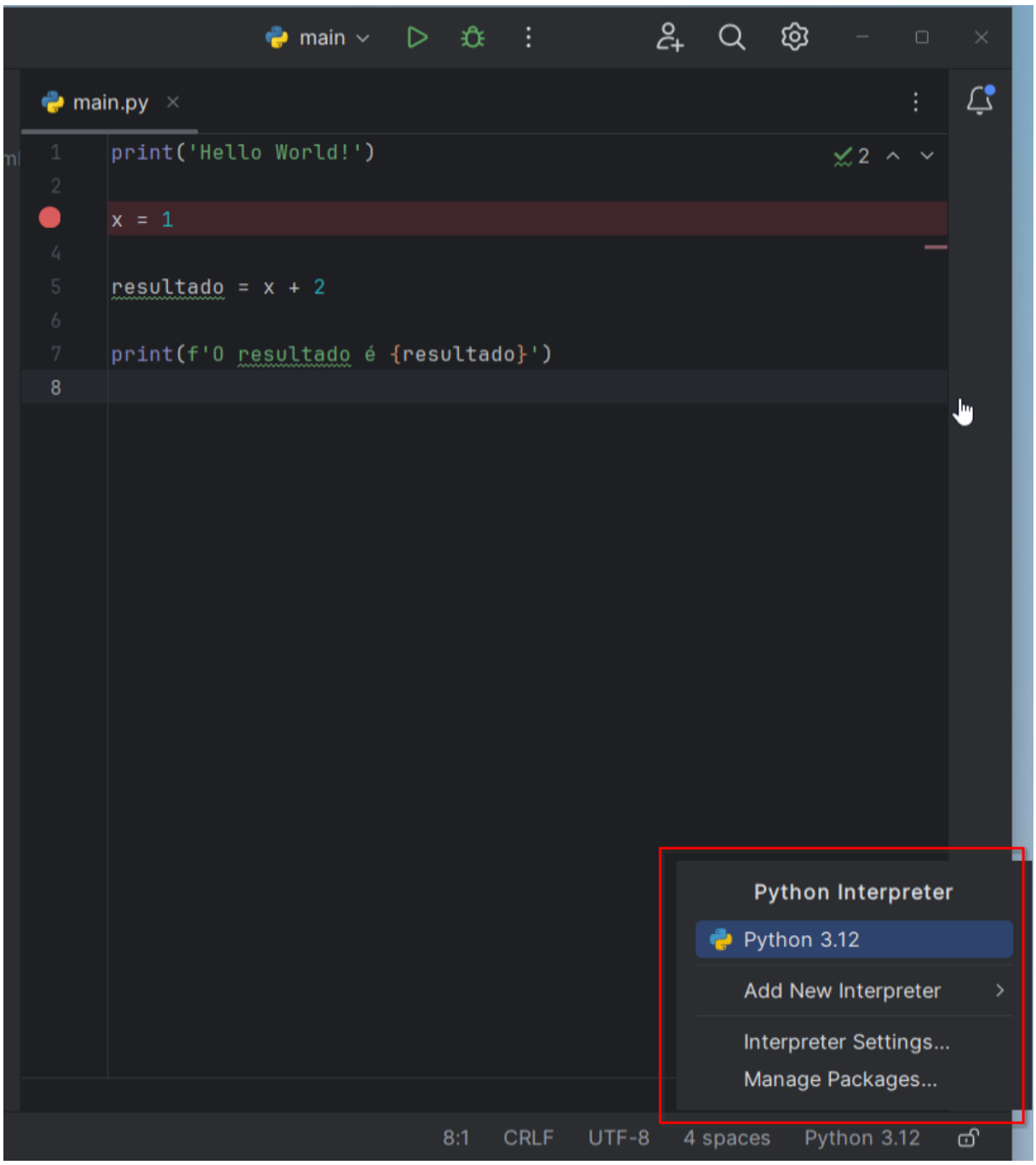


Figure 99: Verificando a instalação de Python

Por fim, muitas opções estão dentro dos menus no topo do programa. Alguns itens úteis são `File > New Project...` para criar projetos novos, ou `Recent Projects` para abrir algum projeto recente. Há também a opção de `Settings` para configurar todos os aspectos da IDE:

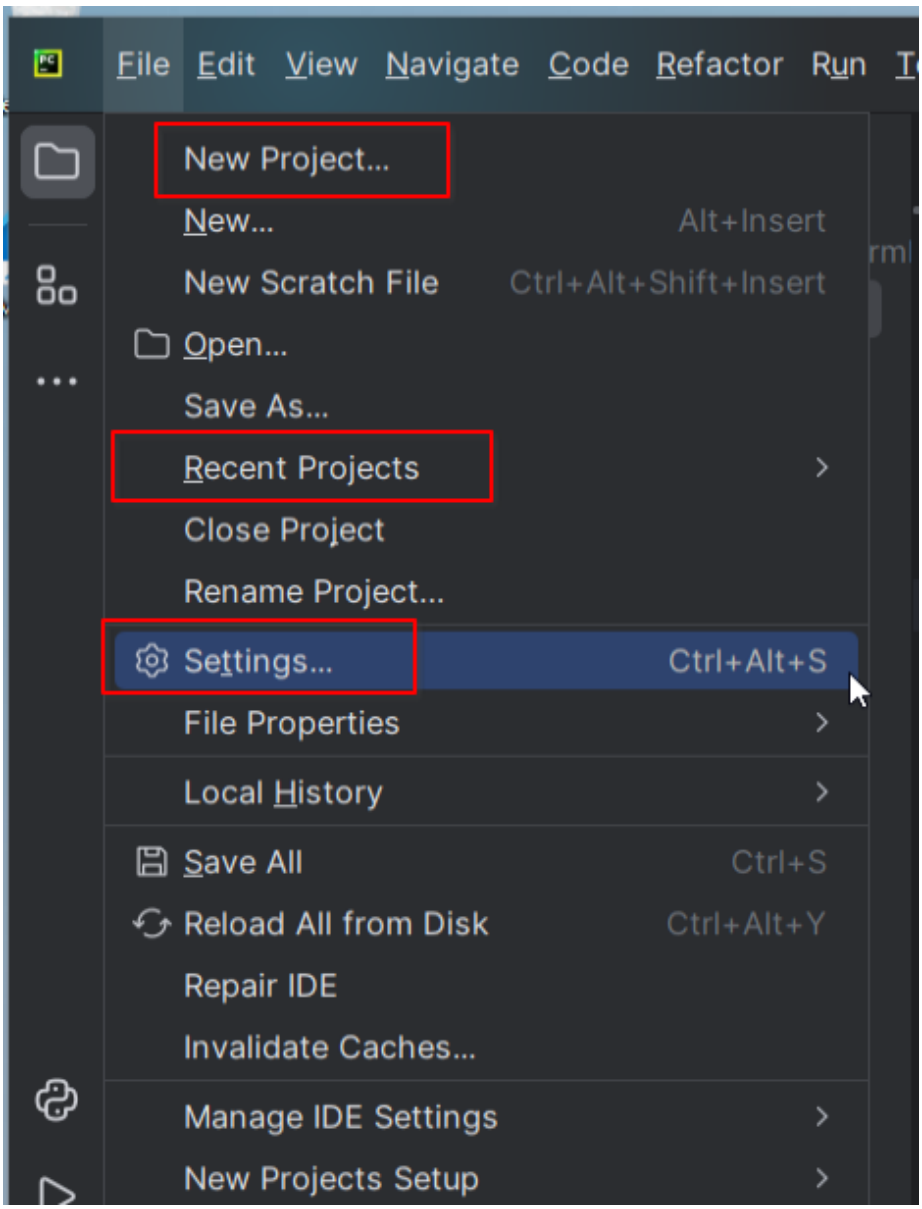


Figure 100: O menu `File`

Uma seção interessante na janela de Settings é `Plugins`. Estes são equivalentes às extensões do VS Code:

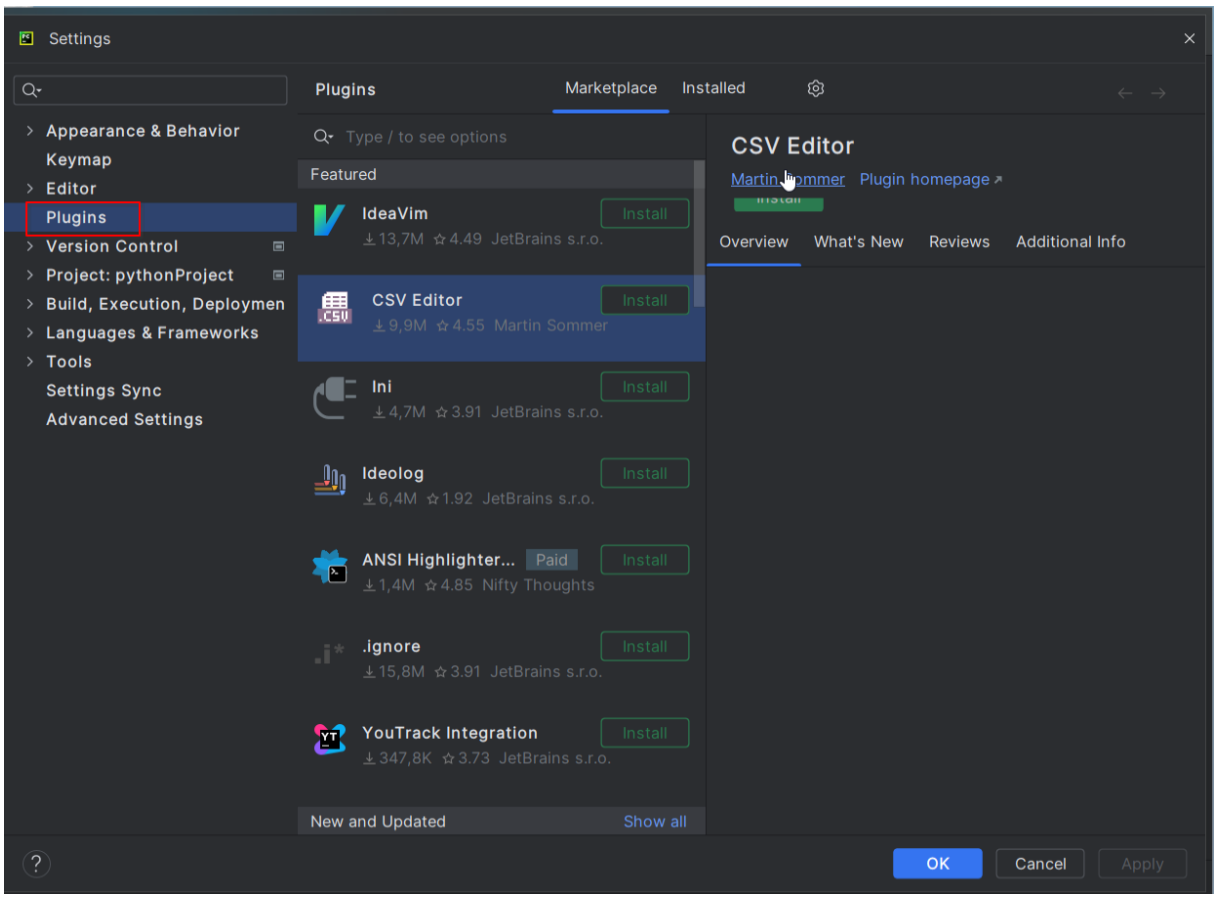


Figure 101: Instalação de plugins no PyCharm

13. O console do IPython

O console de Python tradicional (acessível através do comando `python` no terminal) é útil para testar códigos simples, mas não tem muitos recursos. Com o tempo, a comunidade desenvolveu um console mais interativo, chamado de **IPython**.

É possível instalá-lo com o comando `pip install ipython`:

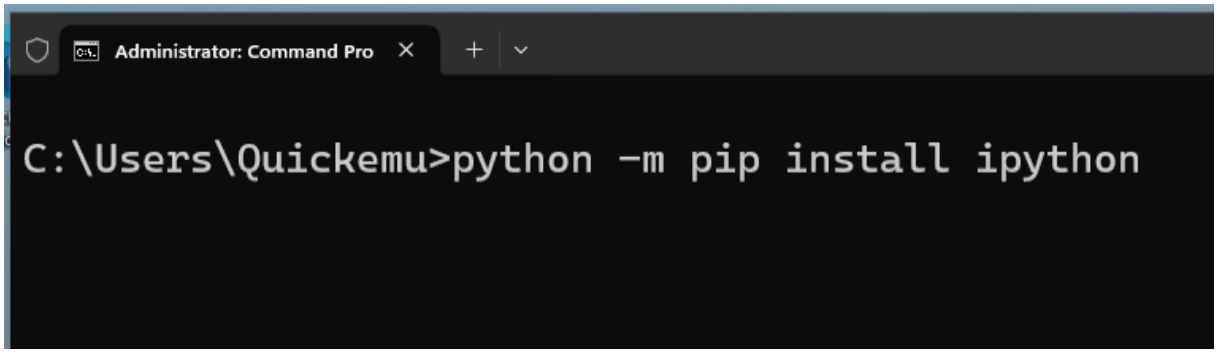


Figure 102: Instalando o IPython

Após a instalação, rode com `python -m IPython` (atenção para as letras maiúsculas):

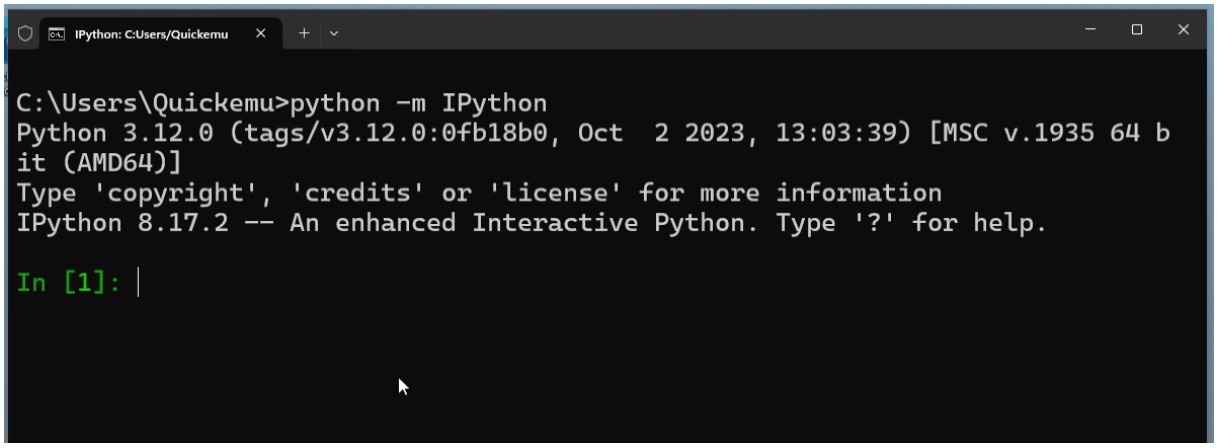


Figure 103: Executando o IPython

Note que comparado com o console tradicional, ele já vem com cores e linhas mais claras de input/output. Além disso, possui funcionalidade de autocompletar palavras anteriores (use o atalho `Ctrl + E` para aceitar a sugestão):

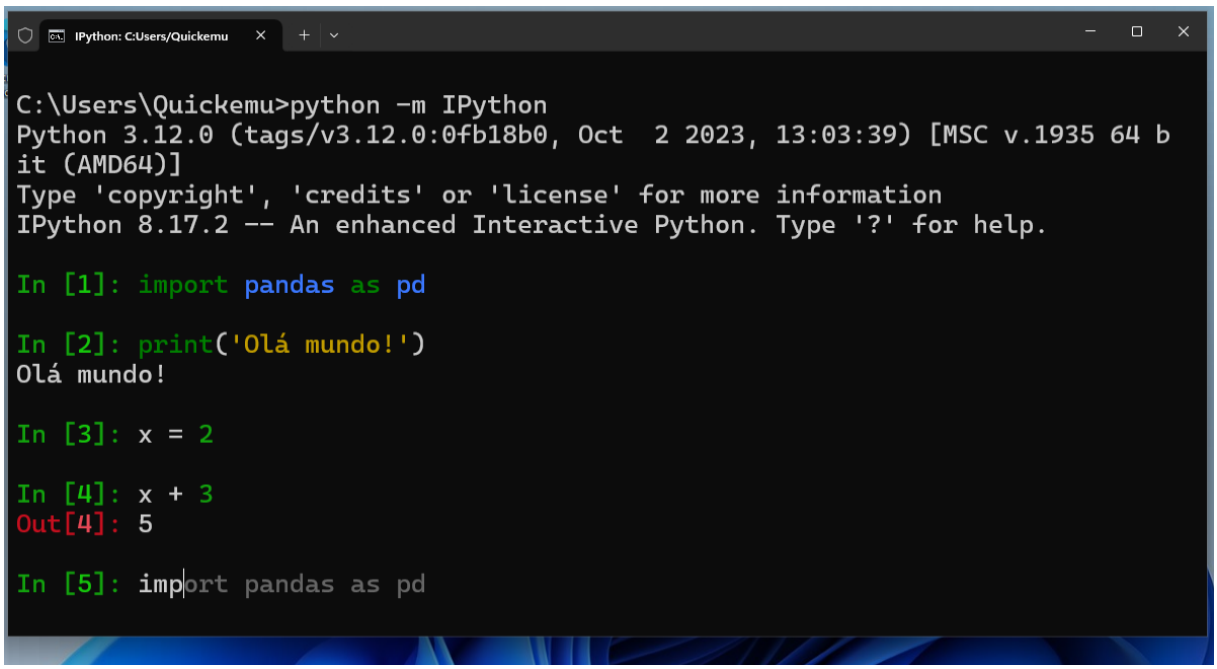


Figure 104: Exemplo das funcionalidades do IPython

Comandos mágicos de IPython

O IPython permite usarmos “comandos mágicos”, que começam com o sinal de porcentagem (`%`). Estes comandos não são código Python, mas ajudam a realizar alguns processos comuns.

Um exemplo é o comando `%timeit`, que permite medir quanto tempo uma porção de código leva pra executar:

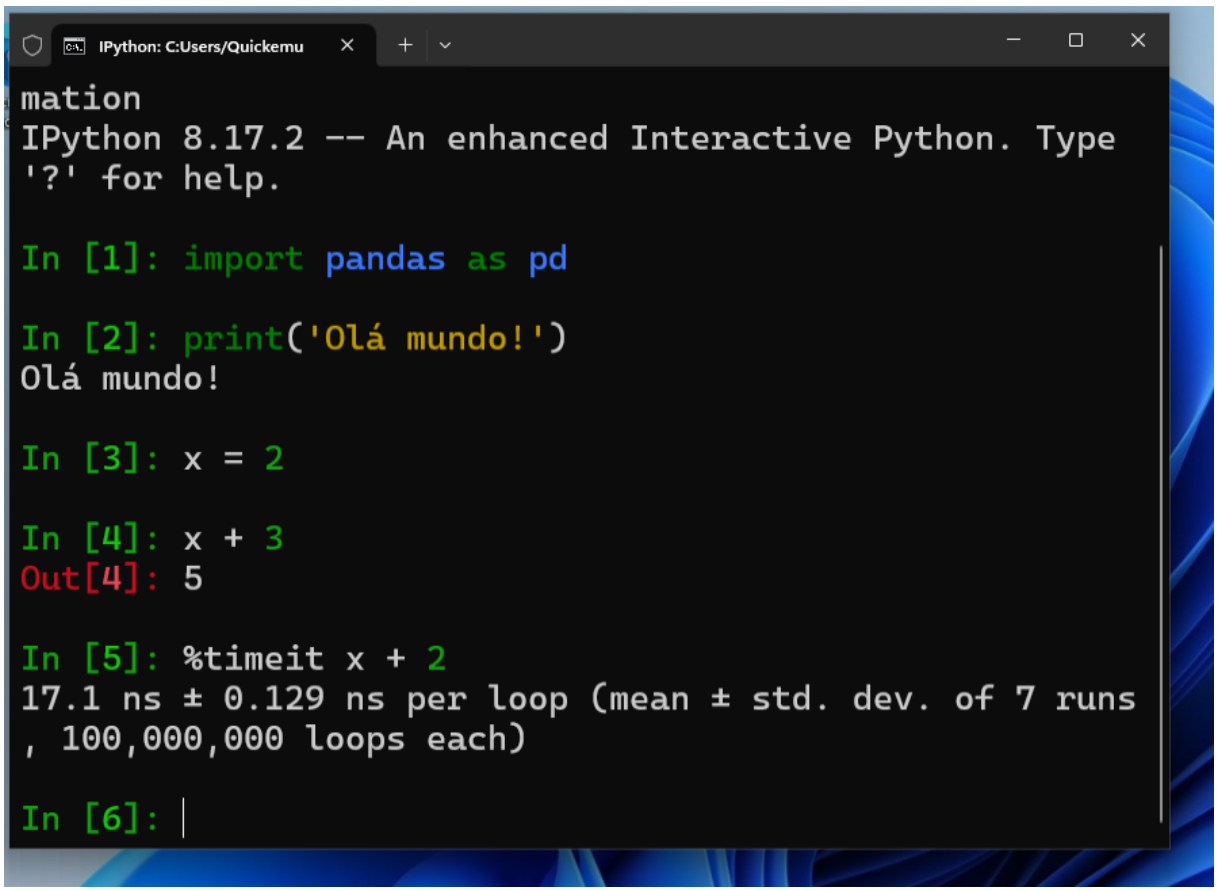


Figure 105: O comando mágico `%timeit`

Já o comando `%run` permite executar um script de Python, que será carregado para dentro do IPython. Dessa forma, é possível editar um script e carregar suas variáveis para dentro do IPython!

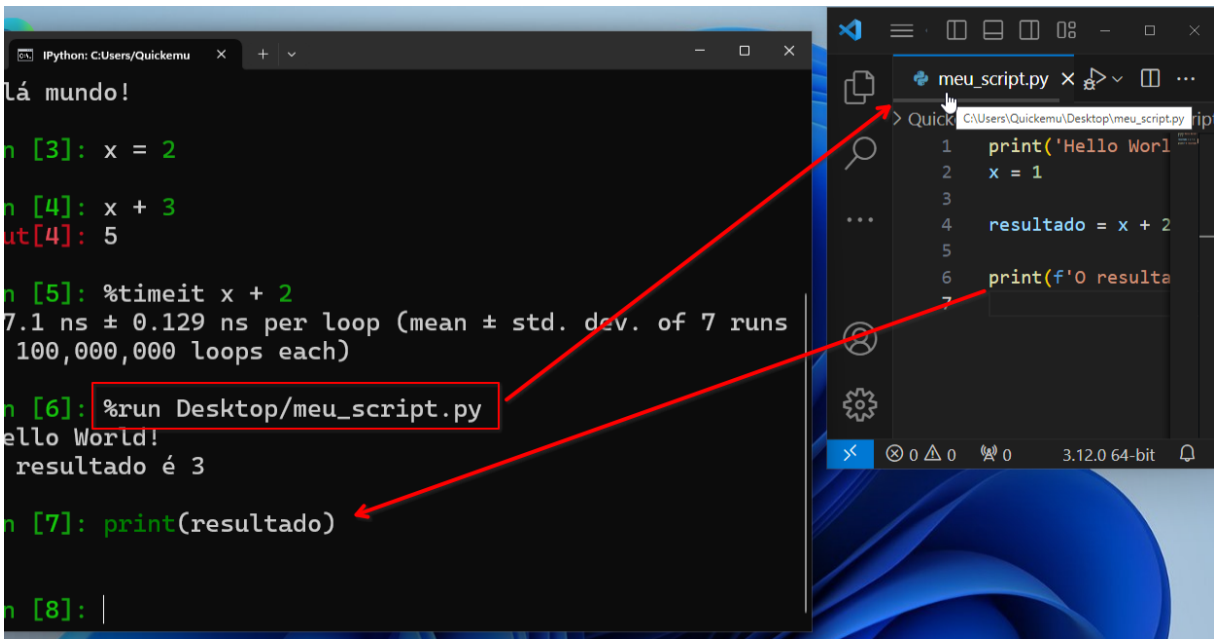


Figure 106: O comando mágico `%run`

IPython nas IDEs

Se o IPython estiver instalado no Python usado em alguma IDE, você terá acesso ao console de IPython dentro dela. Na imagem abaixo, o comando mágico `%run` foi usado para executar o script e carregar suas variáveis diretamente no PyCharm:

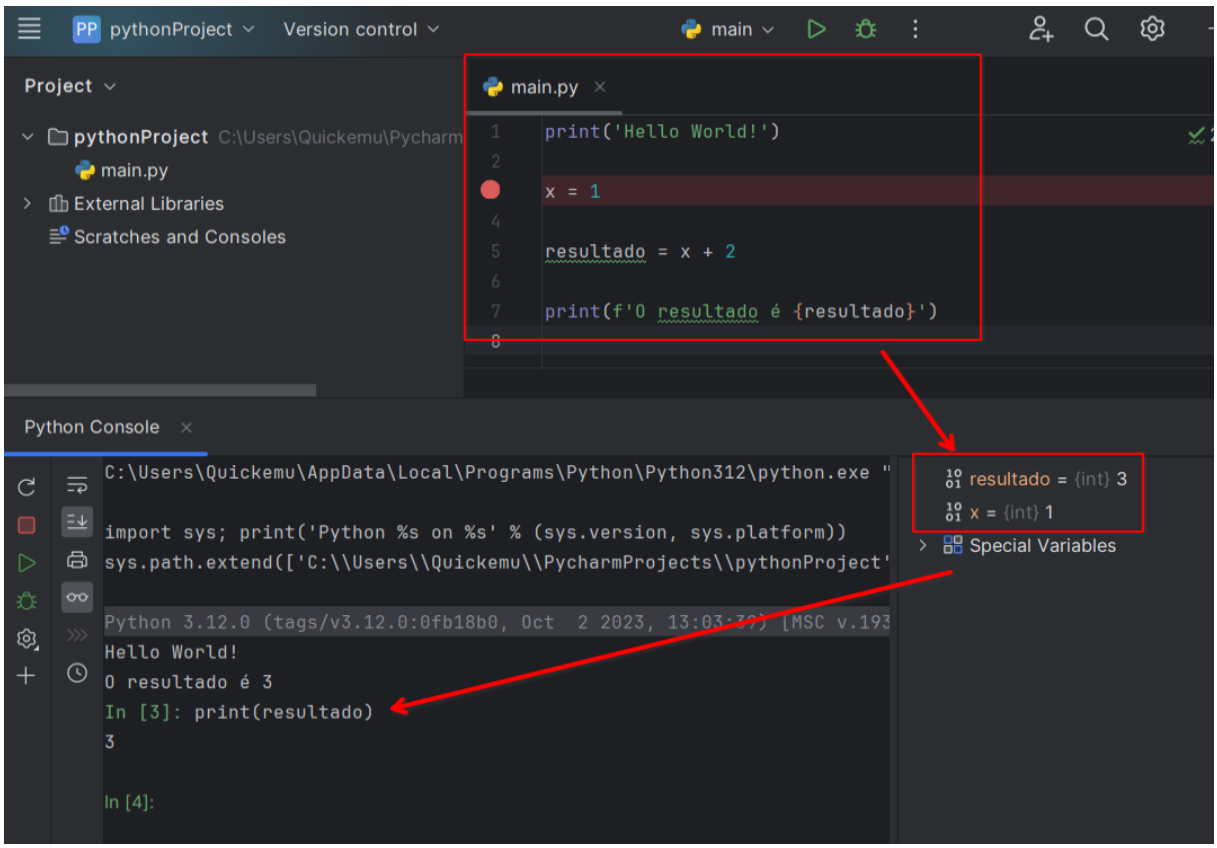


Figure 107: IPython integrado no PyCharm

14. Jupyter Notebook e Jupyter Lab

Os notebooks de Jupyter

Um notebook de Jupyter é uma forma de combinar blocos de código, imagens, e texto explicativo, tudo em um mesmo arquivo.

Para instalá-lo, use o comando `pip install jupyter`:

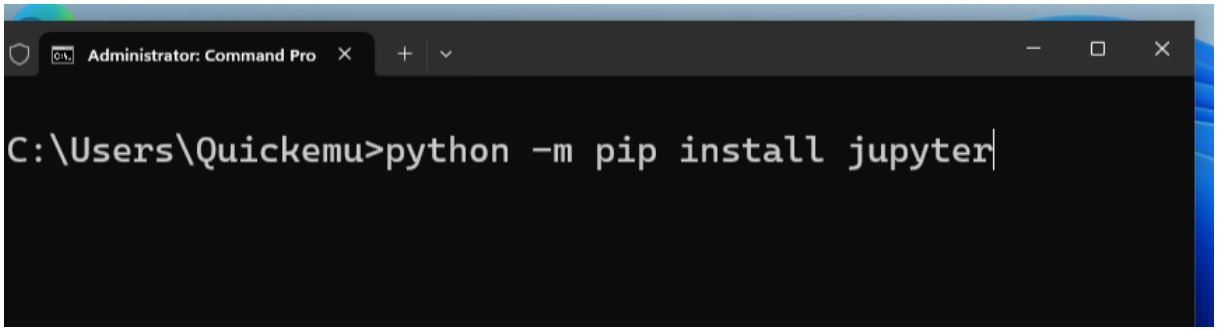


Figure 108: Instalando o Jupyter

Em seguida, use `jupyter-notebook` para iniciar o Jupyter:

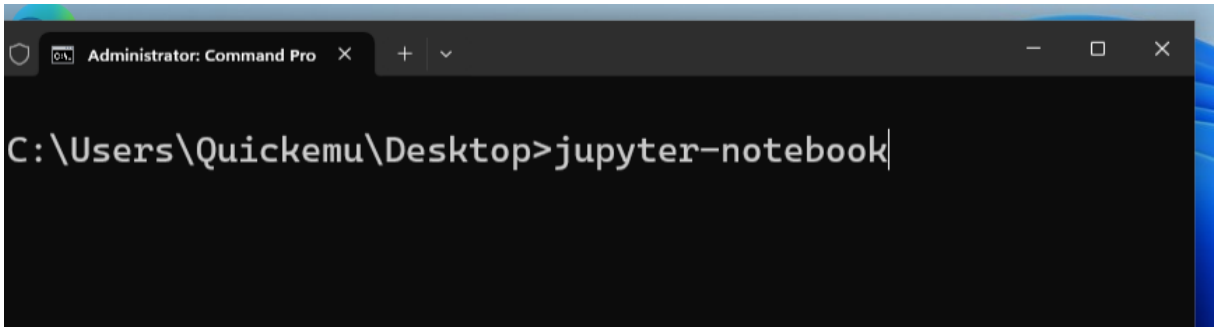


Figure 109: Inicializando o Jupyter Notebook

Se tudo der certo, aparecerá bastante texto no terminal, e seu computador abrirá o Jupyter automaticamente no seu navegador web.

Como o Jupyter funciona

O Jupyter funciona como um "servidor local" na sua máquina. Você precisa deixar o terminal aberto para continuar a ter acesso aos notebooks.

Caso você feche a janela do navegador, no texto do terminal há o link para acesso direto aos notebooks pelo seu navegador:

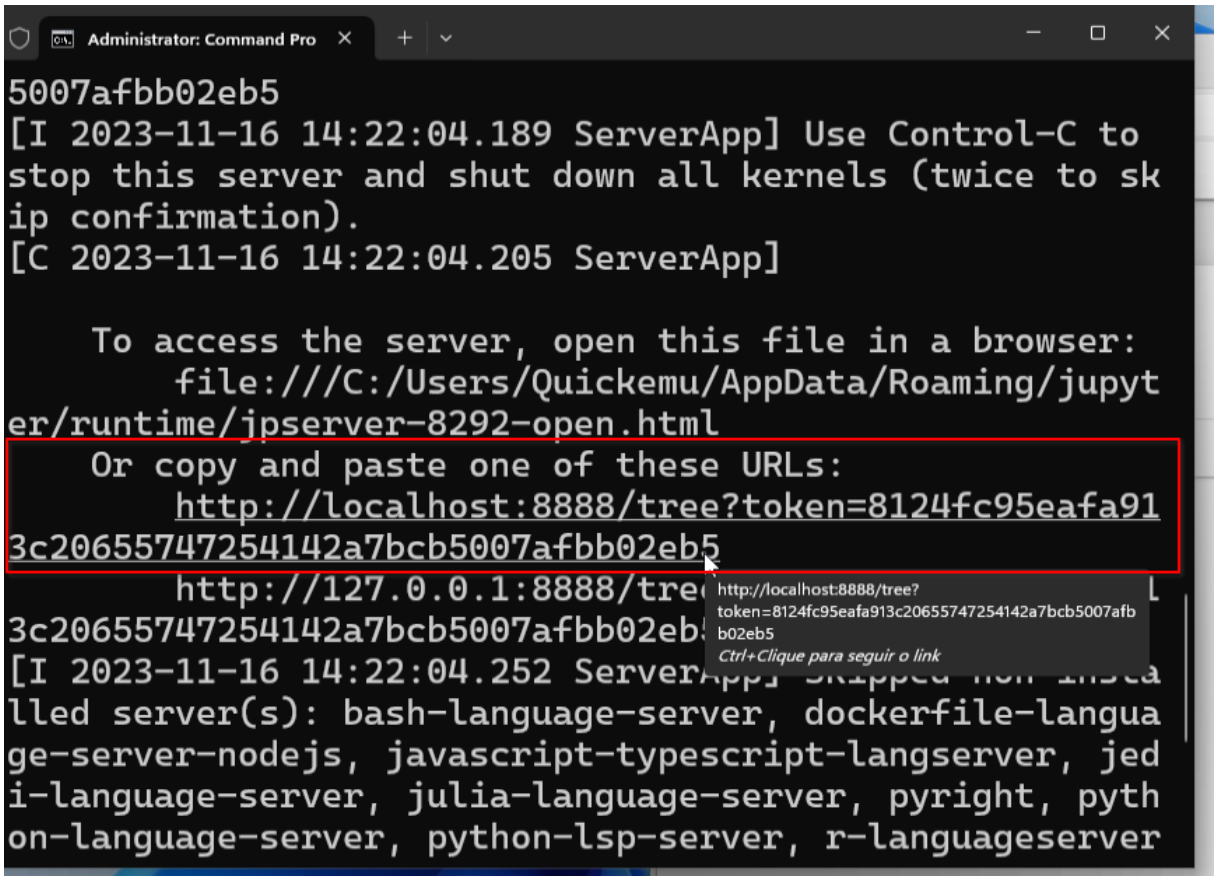


Figure 110: Link para acesso aos notebooks

Por debaixo dos panos, o Jupyter utiliza o mesmo IPython que vimos na aula anterior. A diferença é que ele é capaz de combinar a utilidade do IPython com uma interface que permite misturar código e texto. Assim, o resultado fica parecendo de fato um “caderno” de anotações.

Muitos cientistas de dados utilizam o Jupyter para testar código rapidamente, anotar os resultados, e compartilhá-los com colegas.

Criando e usando um notebook do Jupyter

No navegador, você verá os conteúdos da pasta em que estava no terminal:

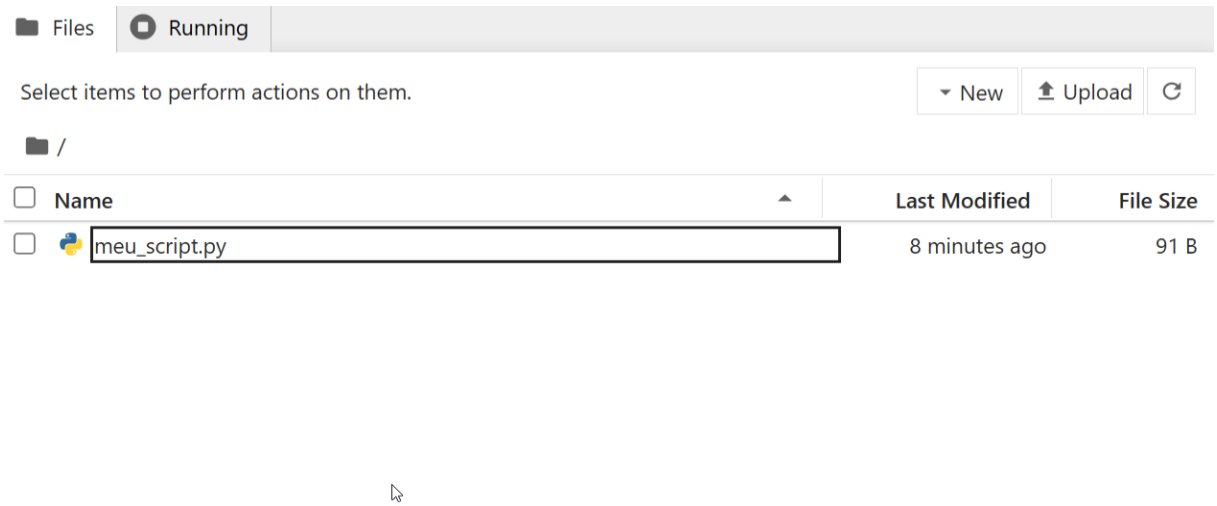


Figure 111: Interface inicial do Jupyter

Para criar seu primeiro notebook, clique no botão New e em seguida `Notebook`:

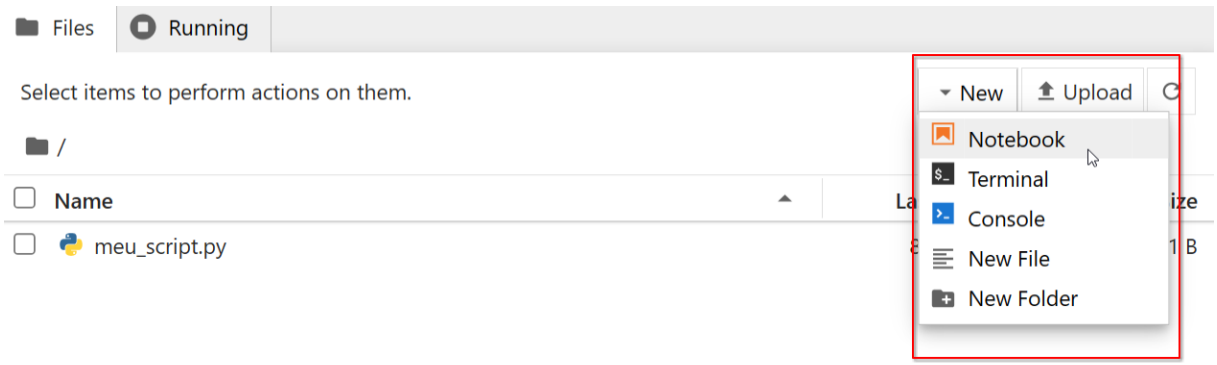


Figure 112: Criando um notebook novo

Ao abrir, o Jupyter perguntará qual kernel utilizar (além de Python, Jupyter pode ser usado para rodar código nas linguagens R e Julia). Escolha `Python 3` e clique em `Select`:

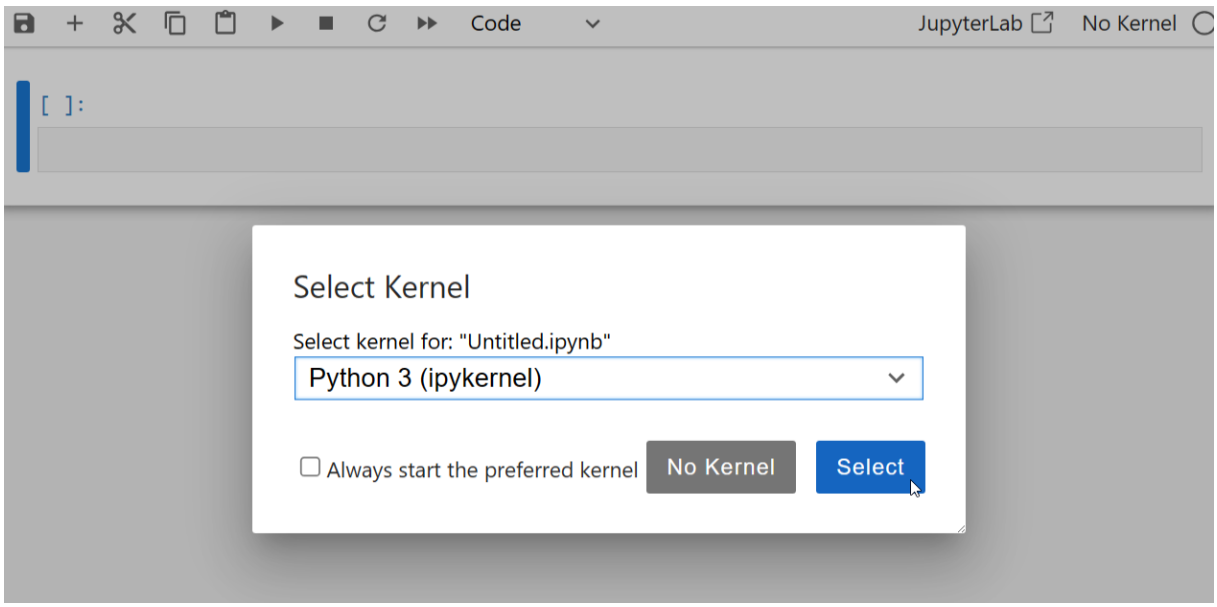


Figure 113: Escolhendo o kernel

Célula `Markdown`

O Jupyter funciona com **células**. Cada célula pode conter código Python (célula do tipo Code) ou texto formatado (célula do tipo `Markdown`).

Selecione a primeira célula e altere seu tipo para `Markdown`:

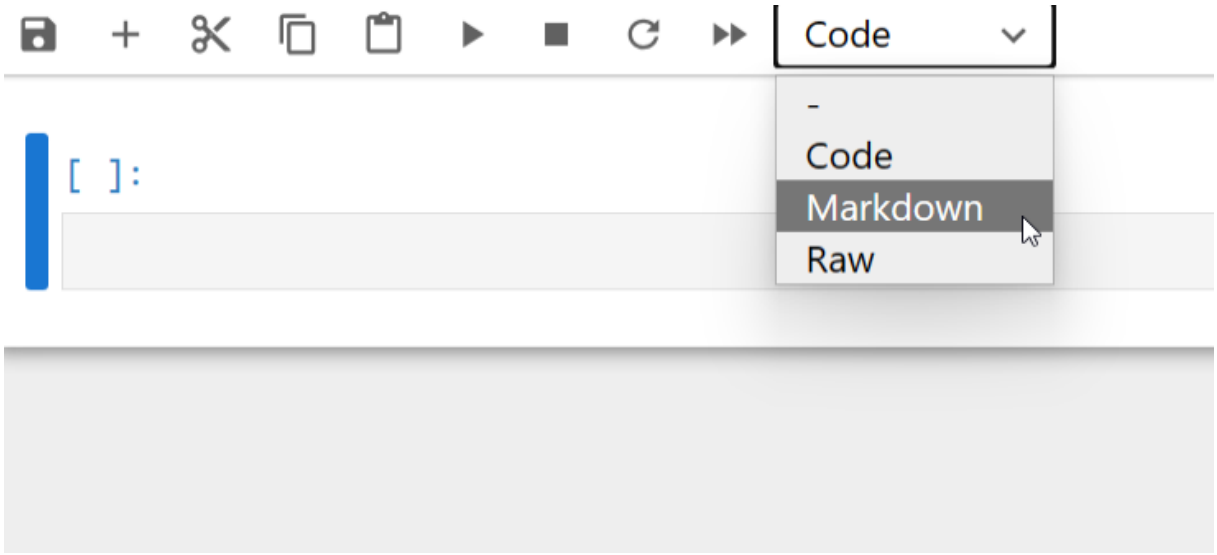


Figure 114: Alterando o tipo de célula

Escreva texto markdown dentro da célula. A linguagem markdown é bastante simples, e requer formatação mínima para ser exibida de forma elegante. Por exemplo, hashtags (#) são usados para controlar cabeçalhos, e hífens (−) delimitam listas:

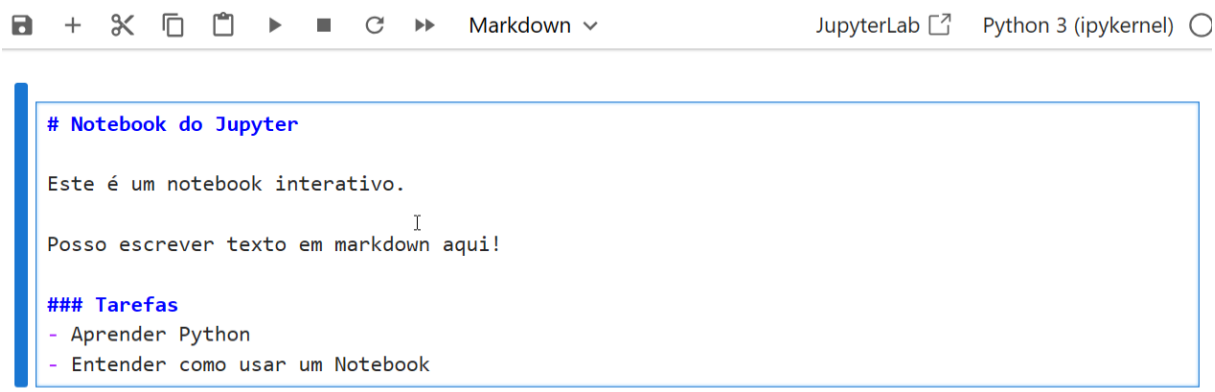


Figure 115: Escrevendo texto em markdown

Ao terminar de editar, execute a célula (símbolo de "Play" no topo do arquivo, ou atalho `Shift + Enter`). Você verá o texto markdown renderizado:

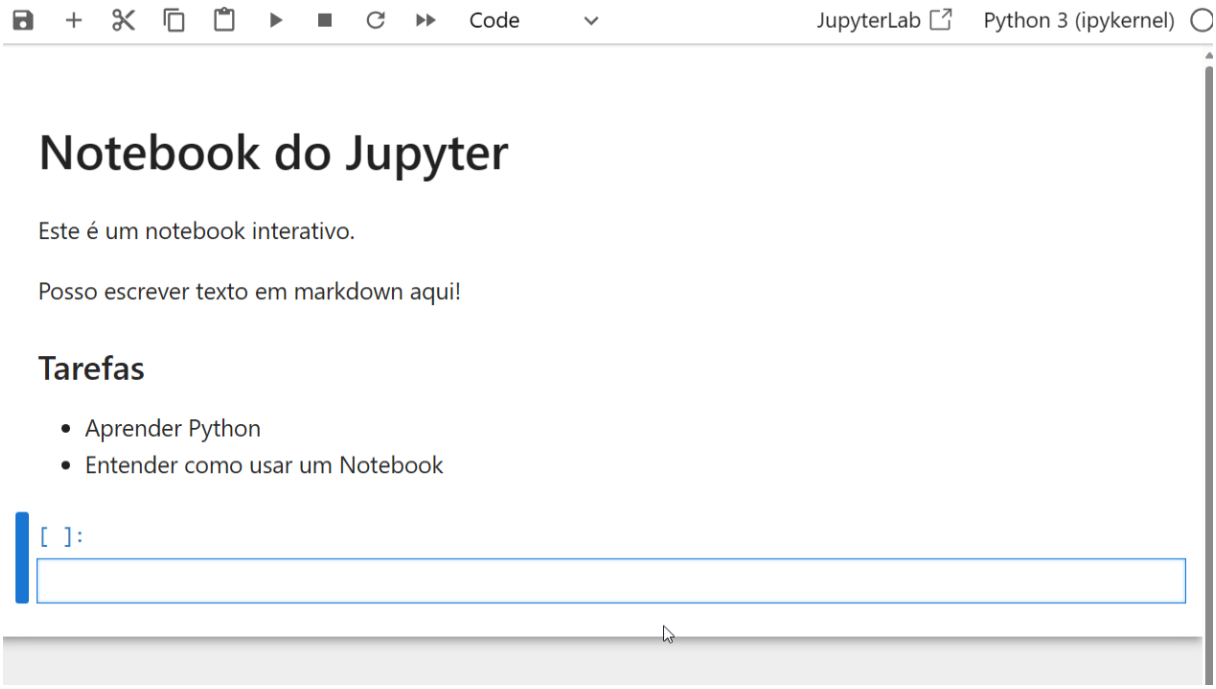


Figure 116: Célula markdown renderizada

Célula Code

Como esperado, você pode escrever e executar código Python nas células de tipo Code.

Crie células desse tipo com comandos de Python (**dica**: cada vez que você executa uma célula com `Shift + Enter`, uma nova célula abaixo já é criada):

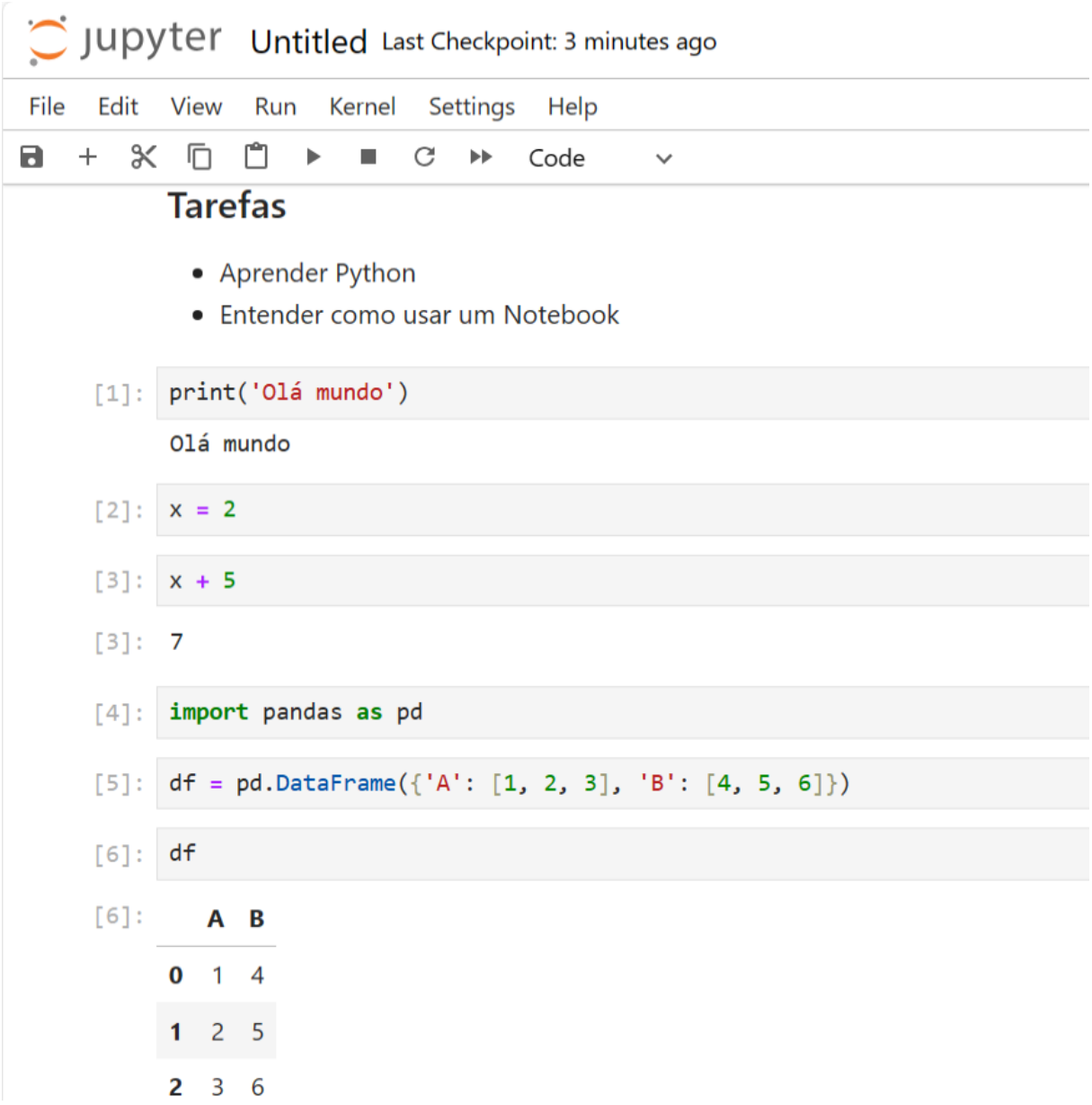


Figure 117: Rodando código em células do Jupyter

Salvando o arquivo do notebook

No topo do arquivo, é possível dar um nome ao seu notebook. A sugestão é dar um nome informativo, para evitar ficar com diversos notebooks chamados `Untitled` no seu computador!

Note também que a extensão de um arquivo de notebook `.ipynb` é diferente de um arquivo Python:

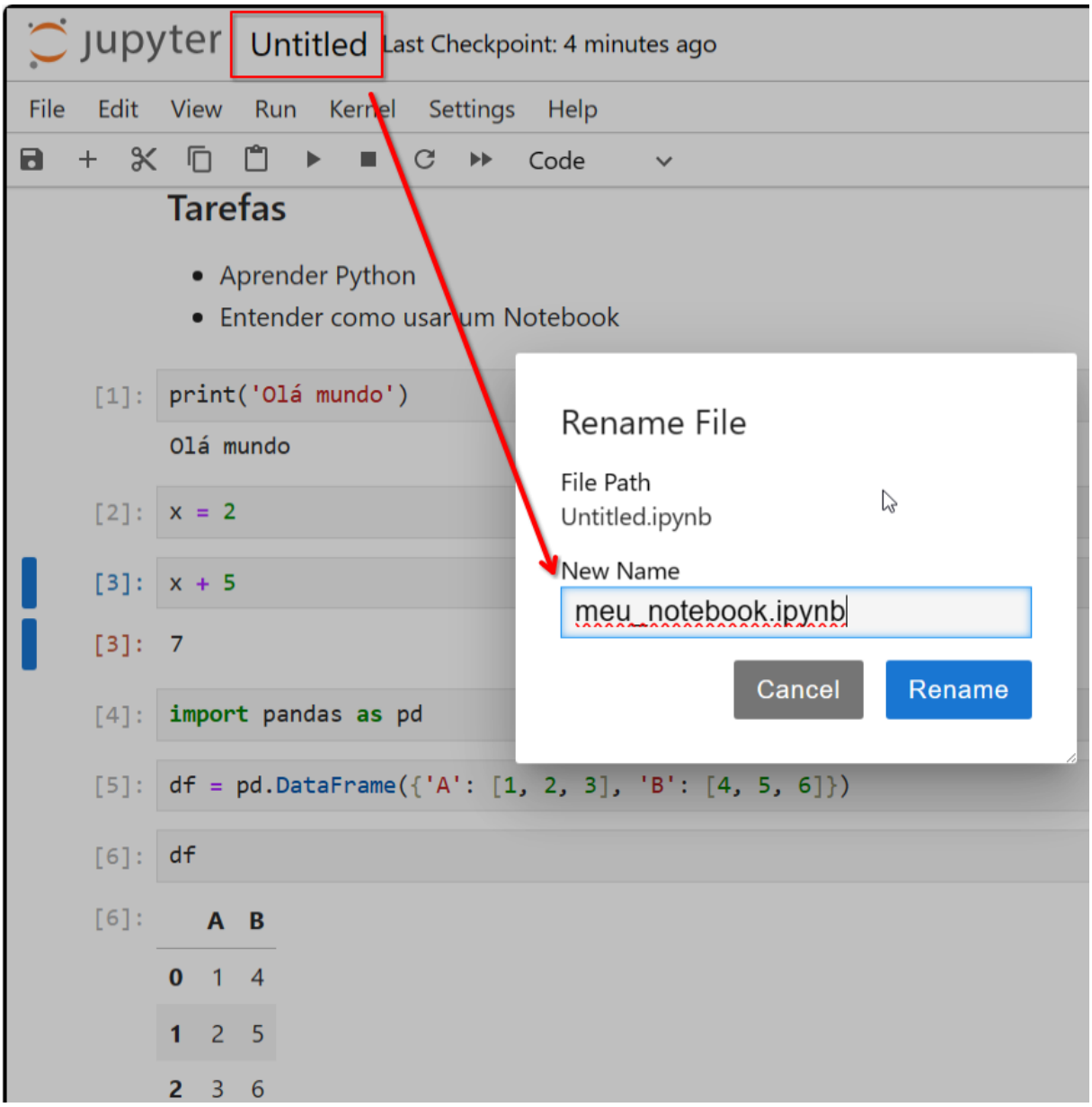


Figure 118: Modificando o nome do notebook

Após salvarmos o notebook, ele pode ser aberto nas IDEs (lembrando que o PyCharm só permite leitura

de notebooks na versão gratuita):

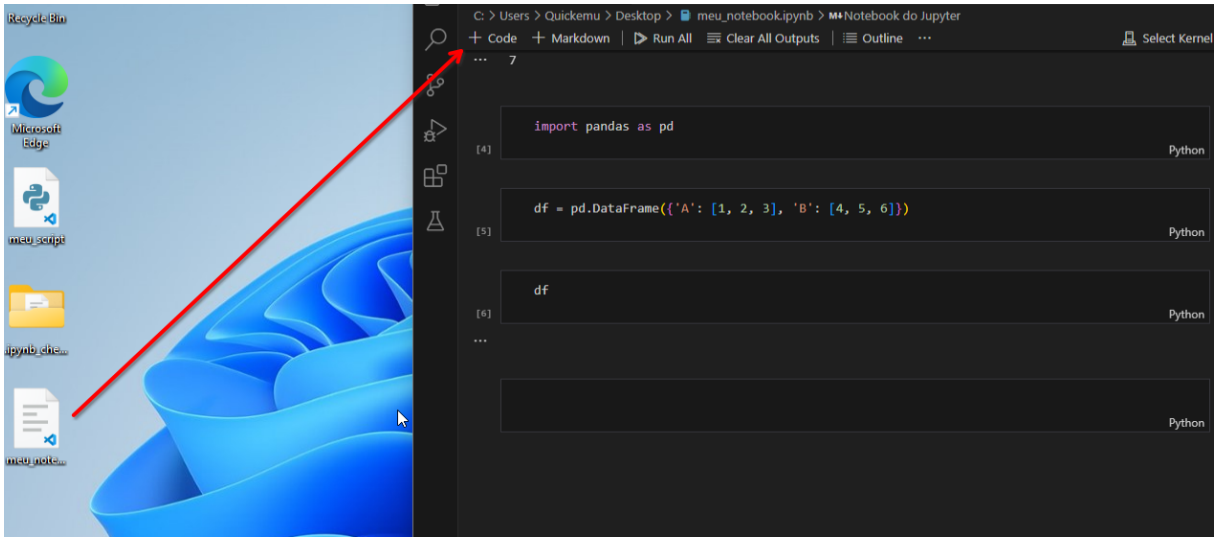


Figure 119: Abrindo o notebook no VS Code

Na realidade, o arquivo do notebook em si **não é código Python**, mas sim um arquivo com formatação similar a JSON:

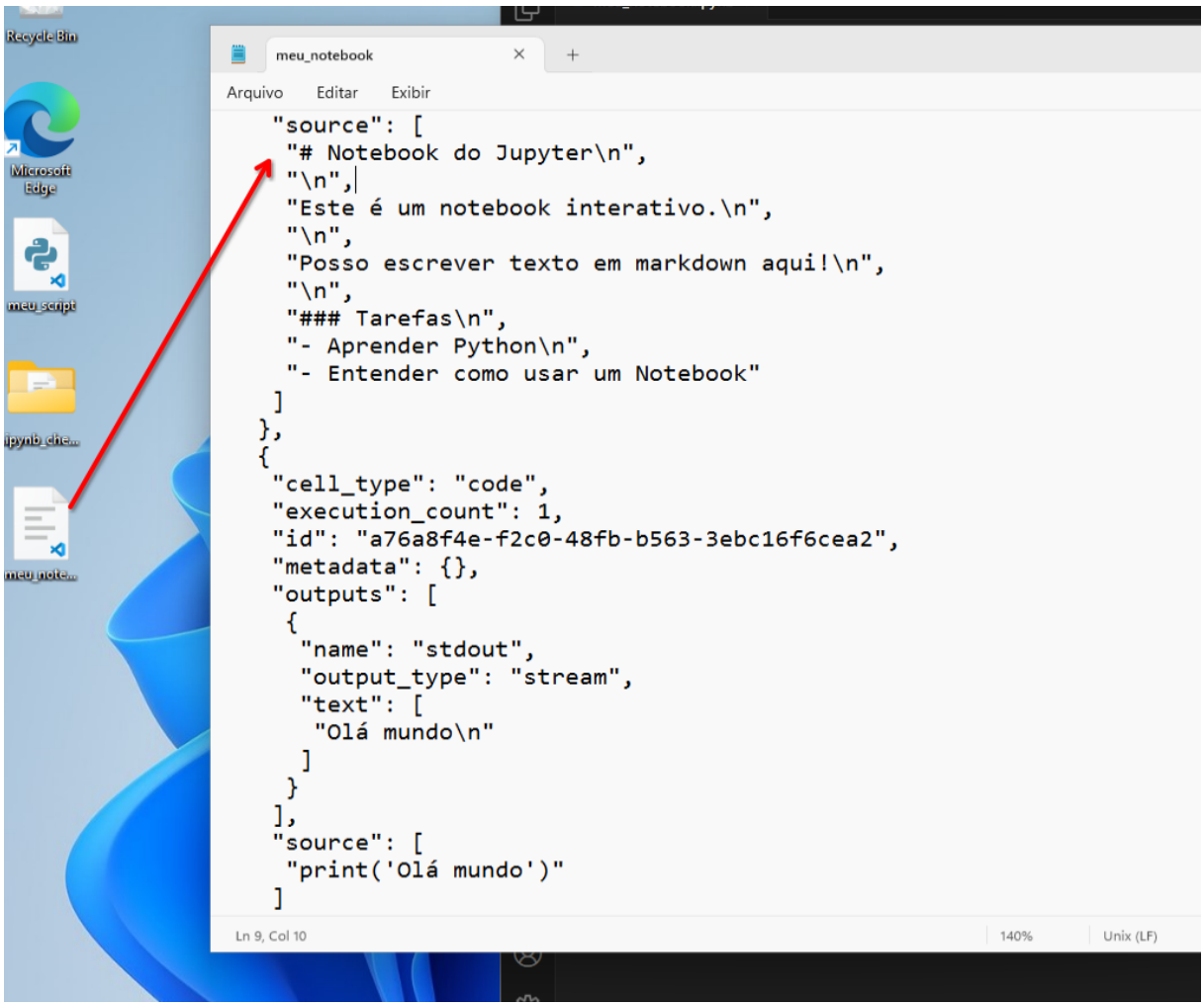


Figure 120: O texto bruto de um notebook

Este arquivo pode ser interpretado pelo Jupyter, mas não serve como código-fonte para seus programas de Python. Quando você for desenvolver um código mais sólido, lembre-se de passar as informações dentro do Jupyter para um script de Python de verdade, com extensão `.py`!

JupyterLab

Existe também o JupyterLab, um programa ligeiramente mais avançado para trabalhar com notebooks de Jupyter. Se pensarmos que os notebooks de Jupyter são equivalentes a um script de Python, então o JupyterLab seria equivalente a uma IDE.

O JupyterLab já foi instalado anteriormente, quando instalamos o Jupyter. Portanto, precisamos apenas usar o comando `jupyter-lab` para iniciar o servidor:

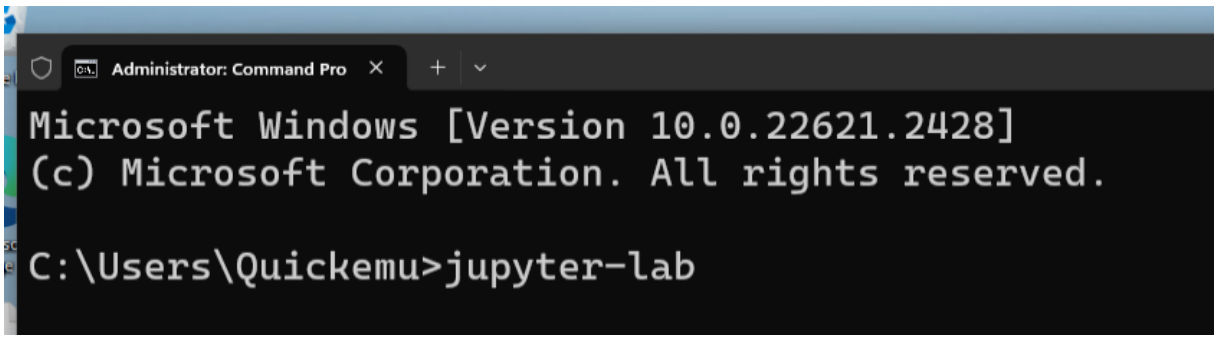


Figure 121: Rodando o JupyterLab

A interface é similar à dos notebooks padrão, porém há mais opções. Por exemplo, há um explorador de arquivos completo nos ícones à esquerda:

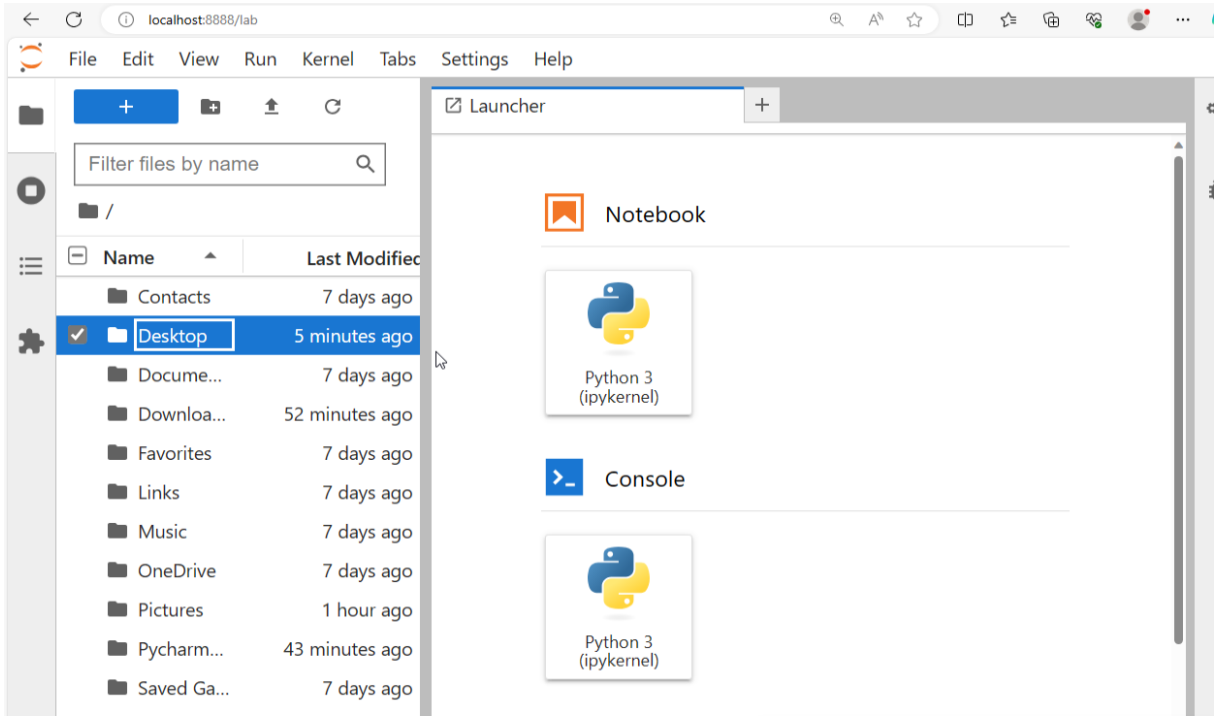


Figure 122: Explorador de arquivos do JupyterLab

Também é possível editar scripts de Python e notebooks em abas dedicadas, como em uma IDE tradicional:

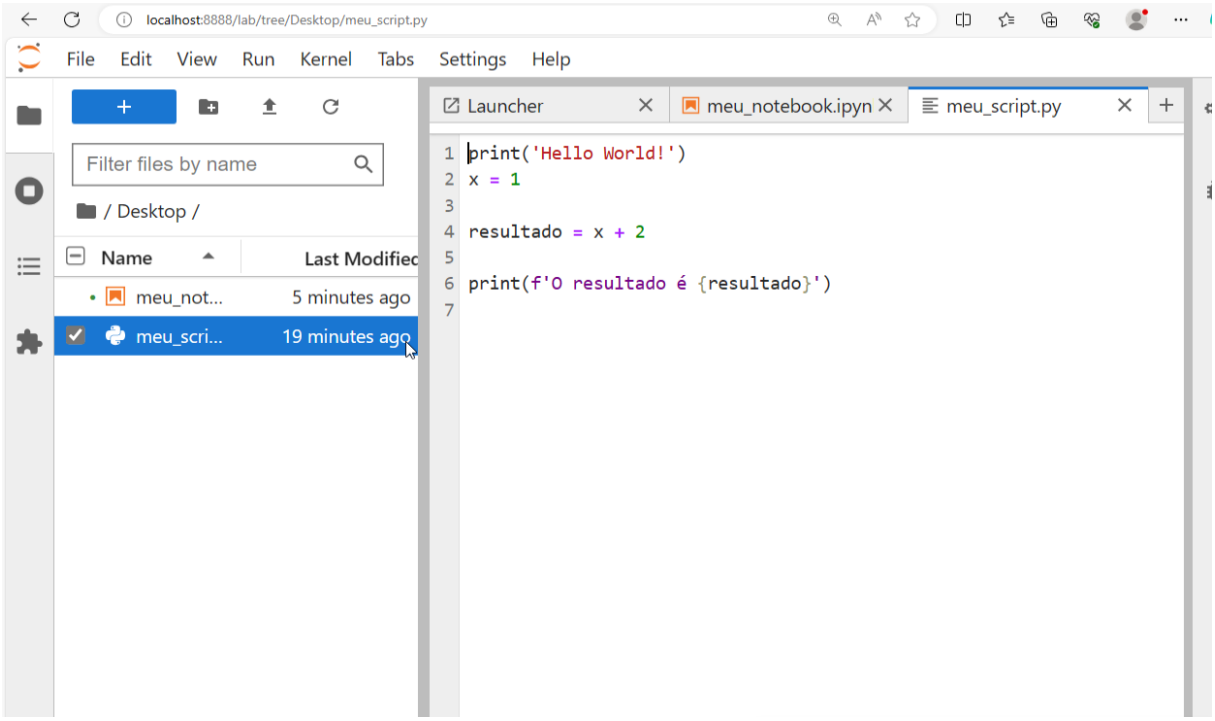


Figure 123: Código e abas do JupyterLab

Sempre que possível, opte por utilizar o JupyterLab, pois possui muito mais funcionalidades que os notebooks!

15. Outras opções de IDEs

Abordamos algumas das principais formas de trabalhar com Python, indo desde os terminais mais básicos aos programas mais avançados.

Dito isso, este curso não consegue abordar todas as formas existentes de rodar Python. Há uma infinidade de programas e IDEs para diferentes nichos, e novos métodos de execução de Python surgem a cada mês.

Nesta aula final, vamos ver algumas formas alternativas de executar Python, e qual o público-alvo de cada uma delas.

Spyder

O Spyder é uma IDE com foco na área científica, que é uma das grandes forças de Python. O Spyder possui áreas específicas na interface para exibição de gráficos e de variáveis, de forma similar ao RStudio (para quem conhece a linguagem R).

A forma indicada pelo próprio site do Spyder é a instalação através do Anaconda Navigator.

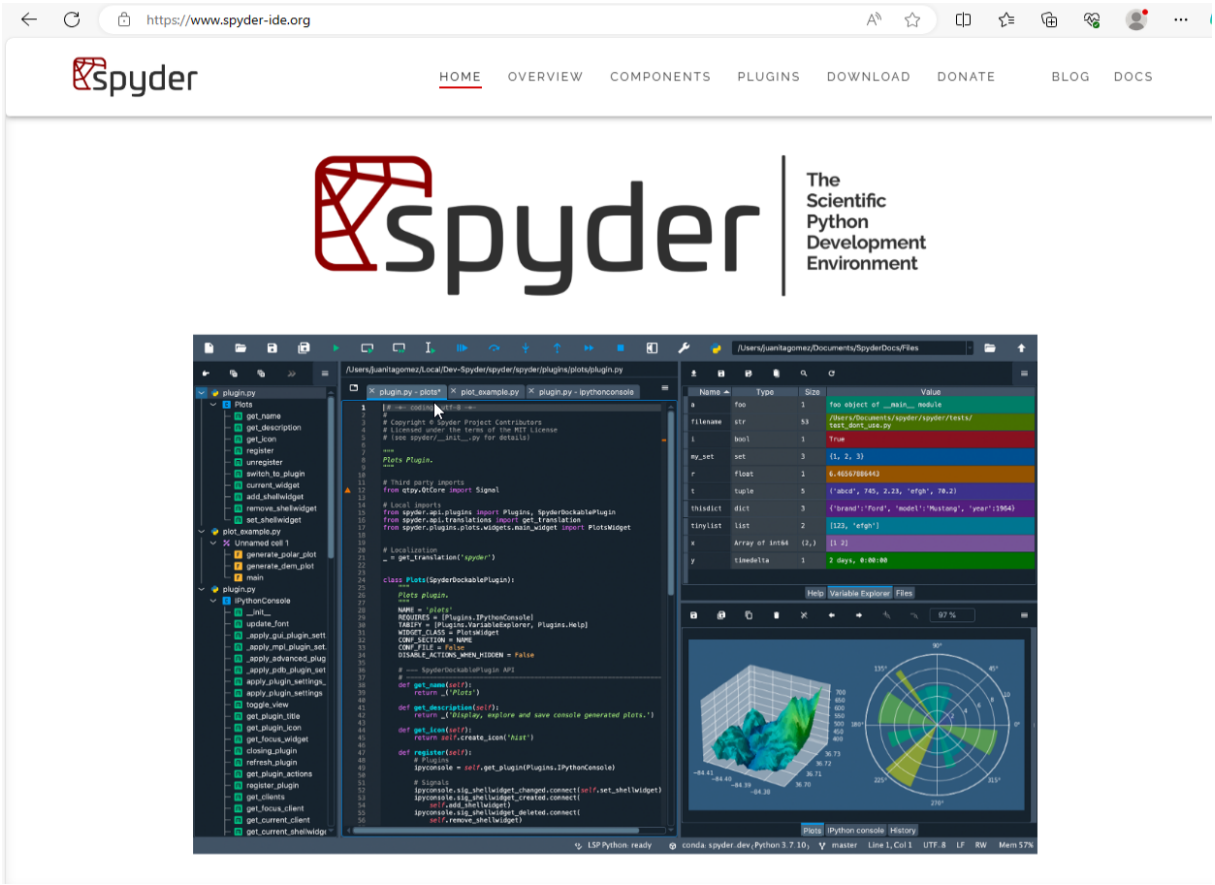


Figure 124: Página de apresentação da IDE Spyder

Google Colab

O Google Colab é uma forma de escrever notebooks de Jupyter que não ficam salvos no seu computador.

O ambiente de Python fica nos servidores do Google, e os notebooks gerados podem ser compartilhados facilmente e integrados com outras ferramentas do Google, como o Google Drive.

O serviço é gratuito para os níveis mais básicos, mas dependendo da sua necessidade (processamento com servidores de alto desempenho, ou análise de dados de larga escala), pode ser necessário pagar por ele.

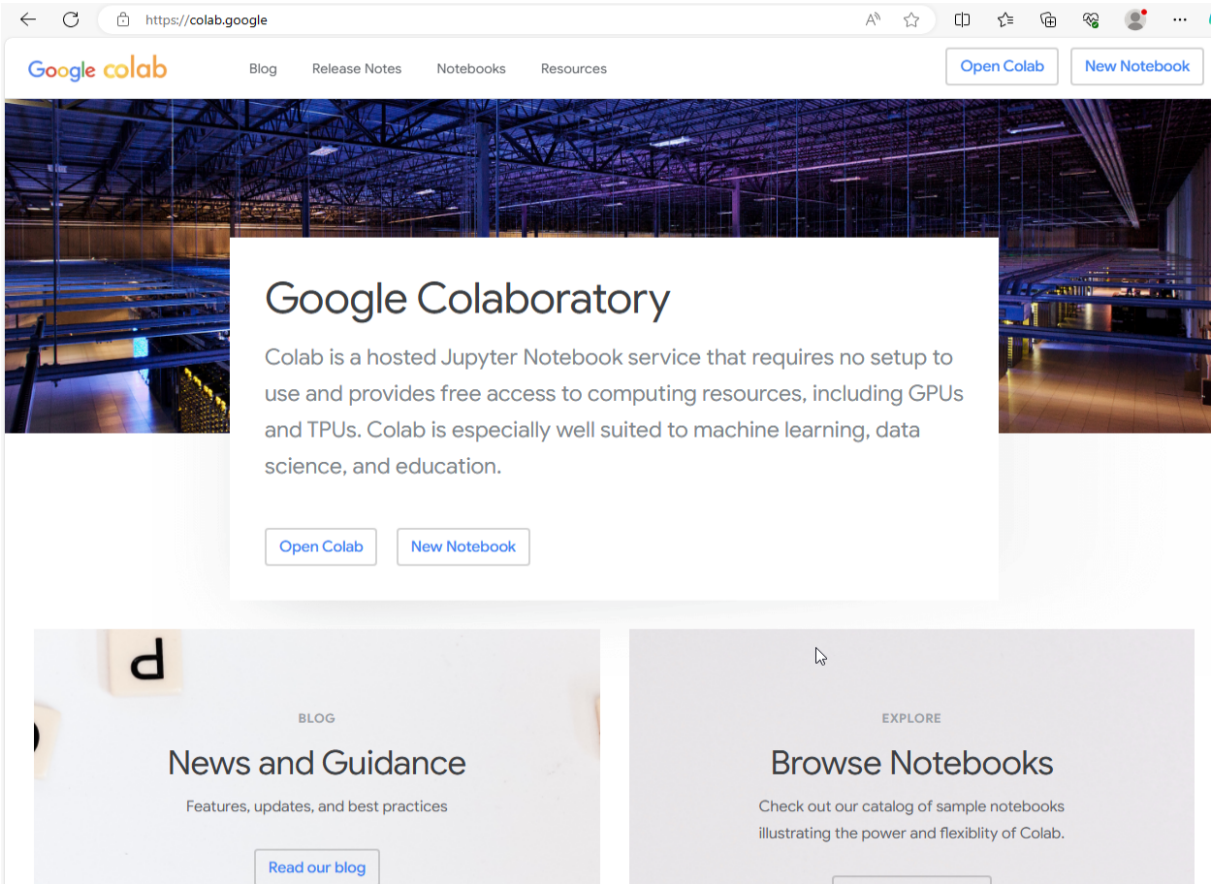


Figure 125: Página de apresentação do Google Colab

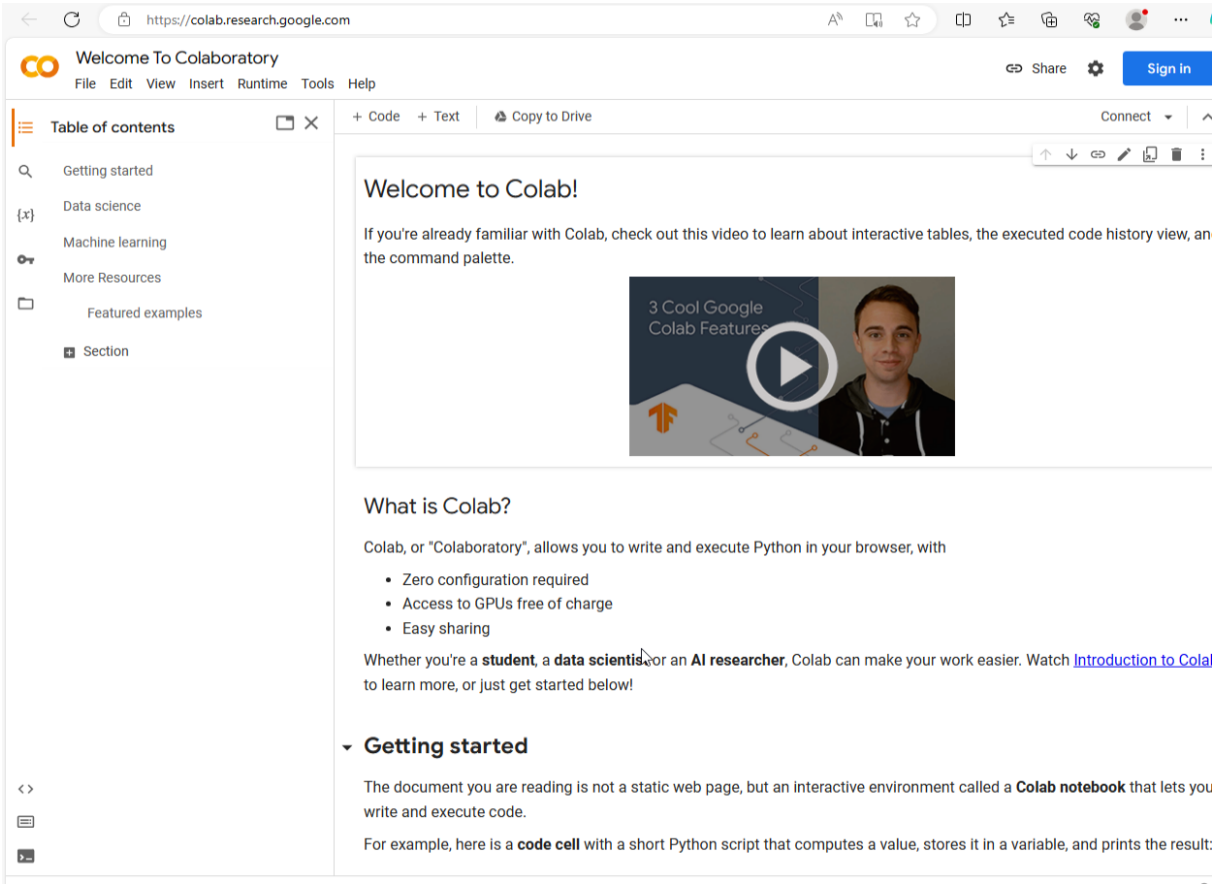


Figure 126: Notebook introdutório do Google Colab

PythonAnywhere

É um serviço de execução de Python na nuvem, oferecido pela Anaconda, a mesma empresa do Anaconda Navigator. Simplifica o gerenciamento do ambiente de Python e permite hospedagem de aplicativos ou qualquer outro tipo de código.

Assim como o Google Colab, oferece planos pagos de acordo com as necessidades de hospedagem e poder de processamento.

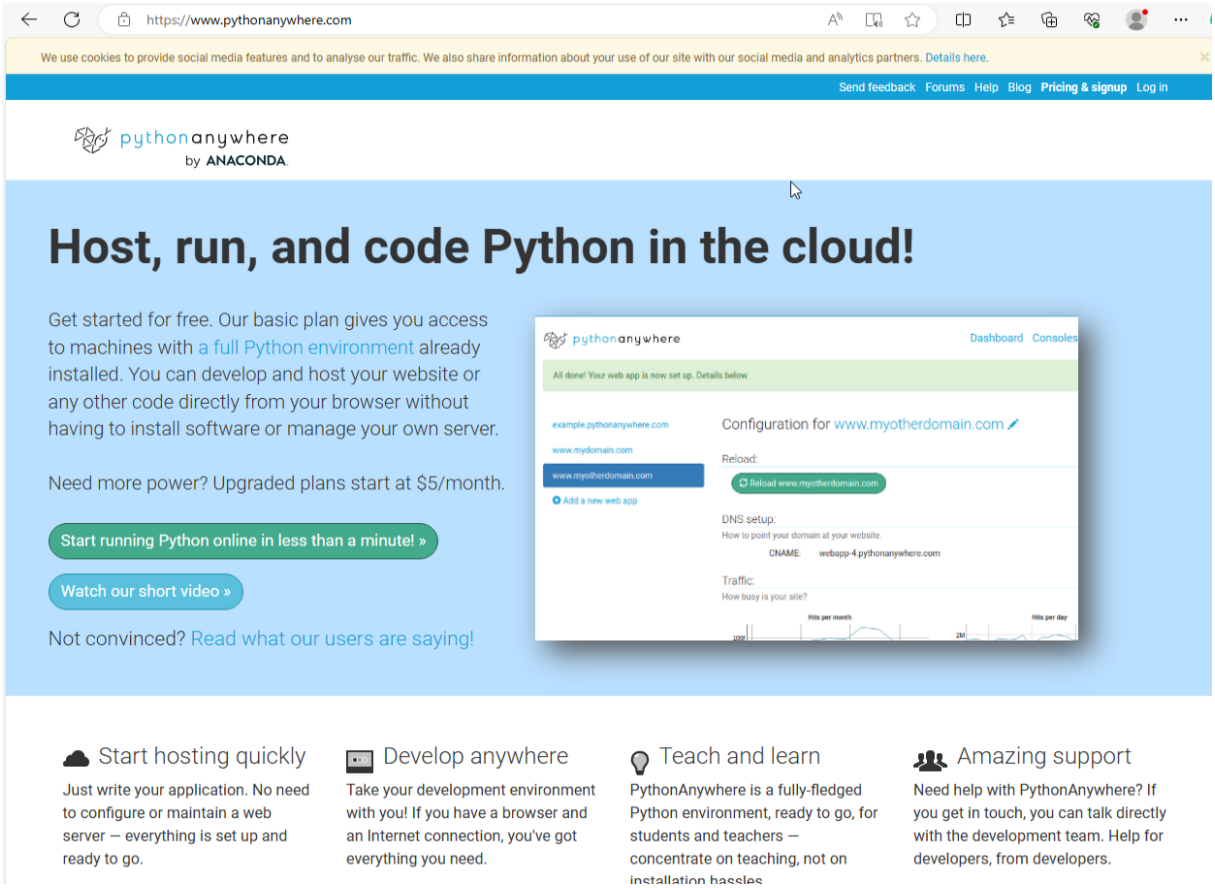


Figure 127: Página de apresentação do PythonAnywhere

Mu

É uma IDE simplificada, como foco na facilidade de uso para iniciantes em Python. Ela vem embutida de uma instalação completa de Python, e portanto não requer que Python já esteja instalado no sistema.

Os alunos do curso **Aprendendo Python: Conceitos Básicos** da Asimov Academy certamente estão familiarizados com ela!

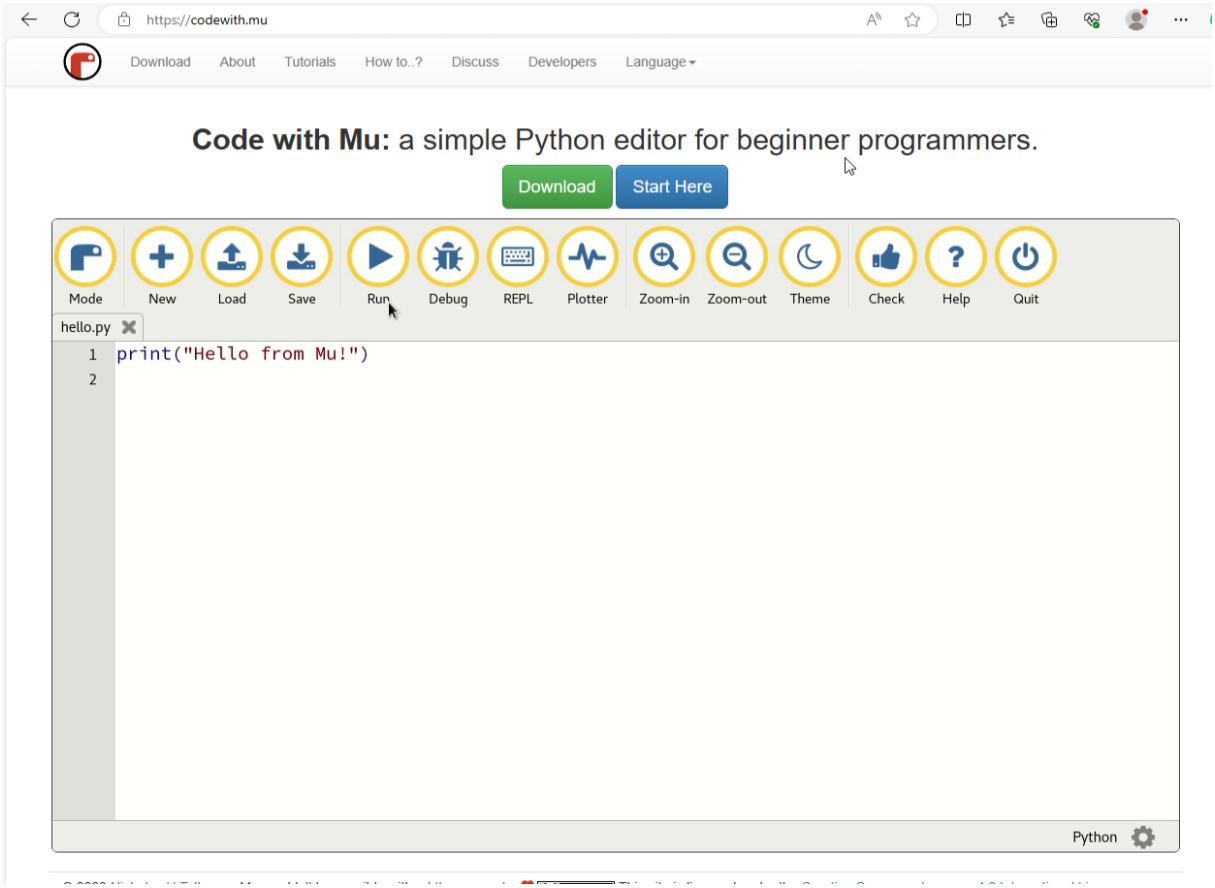


Figure 128: Página de apresentação da IDE Mu

16. O que é um ambiente virtual?

Antes de aprender como utilizar um ambiente virtual em Python, é importante entender para que ele serve e por que ele é necessário.

Imagine que você tem o Python 3.12 instalado em seu computador e deseja trabalhar em um projeto utilizando essa versão. Você instala a biblioteca **Pandas**, que, na época, está na versão 1.5.0. Seu projeto (Projeto A) é desenvolvido com sucesso utilizando essa versão.

Meses ou anos depois, um amigo compartilha um novo projeto Python com você (Projeto B). Esse projeto também utiliza a biblioteca Pandas, mas em uma versão mais recente, 2.2.1.

Ao tentar rodar o Projeto B, você percebe que o código não funciona corretamente porque a versão do Pandas é diferente. Para corrigir isso, você pode atualizar a biblioteca com o comando:

```
pip install --upgrade pandas
```

No entanto, ao atualizar o Pandas para rodar o Projeto B, você acaba quebrando o Projeto A, pois ele foi desenvolvido para funcionar com a versão 1.5.0. Isso cria um problema: como manter diferentes versões de bibliotecas para diferentes projetos sem conflitos?

A Solução: Ambientes Virtuais

A solução para esse problema é o uso de **ambientes virtuais**. Os ambientes virtuais (ou **venvs**, como são comumente chamados) permitem que você crie instalações isoladas do Python para cada projeto, evitando conflitos de dependências.

Cada ambiente virtual funciona como uma cópia independente da instalação do Python. Dentro desse ambiente, você pode instalar pacotes específicos sem interferir em outros projetos.

Assim, você pode ter:

- Um ambiente virtual para o **Projeto A**, rodando Python 3.12 e Pandas 1.5.0.
- Um ambiente virtual para o **Projeto B**, rodando Python 3.12 e Pandas 2.2.1.
- Outros ambientes para diferentes projetos, cada um com suas próprias dependências.

Isso garante que cada projeto funcione corretamente sem afetar os outros.

Benefícios dos Ambientes Virtuais

- **Isolamento de dependências**: Cada projeto possui suas próprias versões de bibliotecas.

- **Facilidade de reprodução**: Outras pessoas podem recriar o mesmo ambiente e rodar o código sem problemas.
- **Organização**: Mantém a instalação global do Python limpa e evita conflitos.

Conclusão

Os ambientes virtuais são uma ferramenta essencial para o desenvolvimento em Python, garantindo que cada projeto tenha um conjunto de dependências bem definido e isolado.

Na próxima aula, vamos aprender como criar e gerenciar esses ambientes virtuais na prática!

17 - Criando e Ativando seu Ambiente Virtual

Agora que compreendemos a importância dos ambientes virtuais e suas funcionalidades, vamos aprender na prática como criar e ativar um ambiente virtual para organizar nossas dependências de projeto.

Criando o Ambiente Virtual

Estrutura Inicial

Primeiramente, criaremos uma estrutura de diretórios para organizar nossos projetos. Vamos criar uma pasta chamada **“meus_projetos”** e, dentro dela, duas subpastas: **“projeto_A”** e **“projeto_B”**.

Criando um Ambiente Virtual para o Projeto A

1. Abra o terminal (CMD no Windows ou terminal no macOS/Linux).
2. Navegue até a pasta do projeto:

Windows:

```
 cd Desktop\meus_projetos\projeto_A
```

Linux / MacOS

```
cd Desktop/meus_projetos/projeto_A
```

- Verifique a versão do Python instalada:
- No windows:

```
where python
```

- No Linux / MacOS

```
which python
```

3. Confirme a versão do Python:

```
python --version
```

4. Liste os pacotes instalados globalmente:

```
pip list
```

5. Crie o ambiente virtual:

```
python -m venv venv-projeto-A
```

Isso criará uma pasta chamada **venv-projeto-A**, que conterá os arquivos do ambiente virtual.

Ativando o Ambiente Virtual

Para ativar o ambiente virtual:

- **No Windows (CMD ou PowerShell):**

  ```
  venv-projeto-A\Scripts\activate
  ```

- **No macOS/Linux:**

  ```
  source venv-projeto-A/bin/activate
  ```

Quando ativado, o nome do ambiente virtual aparecerá no início do prompt do terminal, indicando que ele está ativo.

Conferindo a Instalação

Verifique a instalação do Python dentro do ambiente virtual:

```
where python  # Windows
which python  # macOS/Linux
```

Liste os pacotes instalados:

```
pip list
```

Agora, qualquer instalação de pacotes com **pip install** será feita exclusivamente dentro do ambiente virtual.

Instalando Dependências no Ambiente Virtual

Vamos instalar o Pandas como exemplo:

```
pip install pandas
```

Verifique a instalação:

```
pip list
```

Os pacotes instalados estão contidos dentro do ambiente virtual e não afetam o sistema global.

Criando um Ambiente Virtual para o Projeto B

Agora, vamos repetir o processo para o **projeto B**:

1. Navegue até a pasta do projeto B:

```
cd ../projeto_B #Linux / MacOS
cd..\projeto_B #Windows
```

2. Crie o ambiente virtual:

```
python -m venv venv-projeto-B
```

3. Ative o ambiente virtual:

```
venv-projeto-B\Scripts\activate  # Windows
source venv-projeto-B/bin/activate  # macOS/Linux
```

4. Instale uma versão específica do Pandas:

```
pip install pandas==1.5.0
```

Gerenciando Dependências com `requirements.txt`

Para facilitar a instalação de pacotes em outro ambiente, podemos gerar um arquivo **requirements.txt** com todas as dependências:

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

Esse arquivo conterá a lista de pacotes e versões necessárias para o projeto.

Para instalar as dependências em outro ambiente virtual, basta executar:

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

Desativando o Ambiente Virtual

Para desativar o ambiente virtual, utilize:

```
deactivate
```

Com isso, o terminal volta ao ambiente global do sistema.

Conclusão

Aprendemos a criar e ativar ambientes virtuais, instalar dependências e gerenciá-las com o `requirements.txt`. Essa prática é essencial para manter projetos organizados e evitar conflitos de versões de bibliotecas.

Agora que entendemos como trabalhar com ambientes virtuais, seguimos para a próxima aula!

18. Selecionando o Interpretador de Python nas IDEs

Introdução

Uma das principais dúvidas de quem começa a programar em Python está relacionada à escolha do interpretador correto dentro das IDEs. Muitas vezes, os usuários instalam bibliotecas como o `pandas`, mas, ao executar o código na IDE, encontram erros informando que o módulo não foi encontrado. Isso ocorre porque a IDE pode estar rodando um interpretador diferente daquele onde a biblioteca foi instalada. Nesta aula, vamos aprender a selecionar o interpretador adequado no VS Code e no PyCharm.

Selecionando o Interpretador no VS Code

1. **Abrindo o projeto**
    - No VS Code, clique em `File` > `Open Folder` e selecione a pasta do seu projeto.
    - Exemplo: Vamos abrir a pasta do projeto B.
2. **Criando um novo script**
    - Crie um novo arquivo Python chamado `meu_script.py`.
    - No arquivo, escreva o seguinte código:

      ```
      import pandas as pd
      print(pd.__version__)
      ```

    - Salve o arquivo e tente executá-lo.
3. **Identificando o problema**

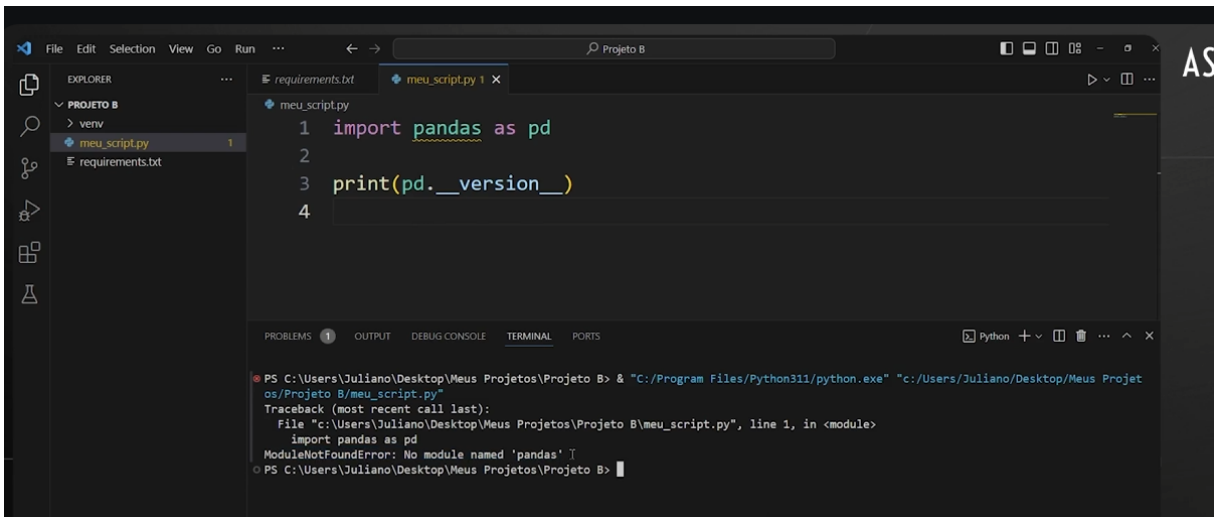


Figure 129: erro no interpretador

- Se a execução resultar em um erro `ModuleNotFoundError: No module named 'pandas'`, significa que o interpretador atual do VS Code não está apontando para o ambiente correto.

4. **Selecionando o interpretador correto**

- No canto inferior direito da tela, verifique o caminho do interpretador de Python em uso.

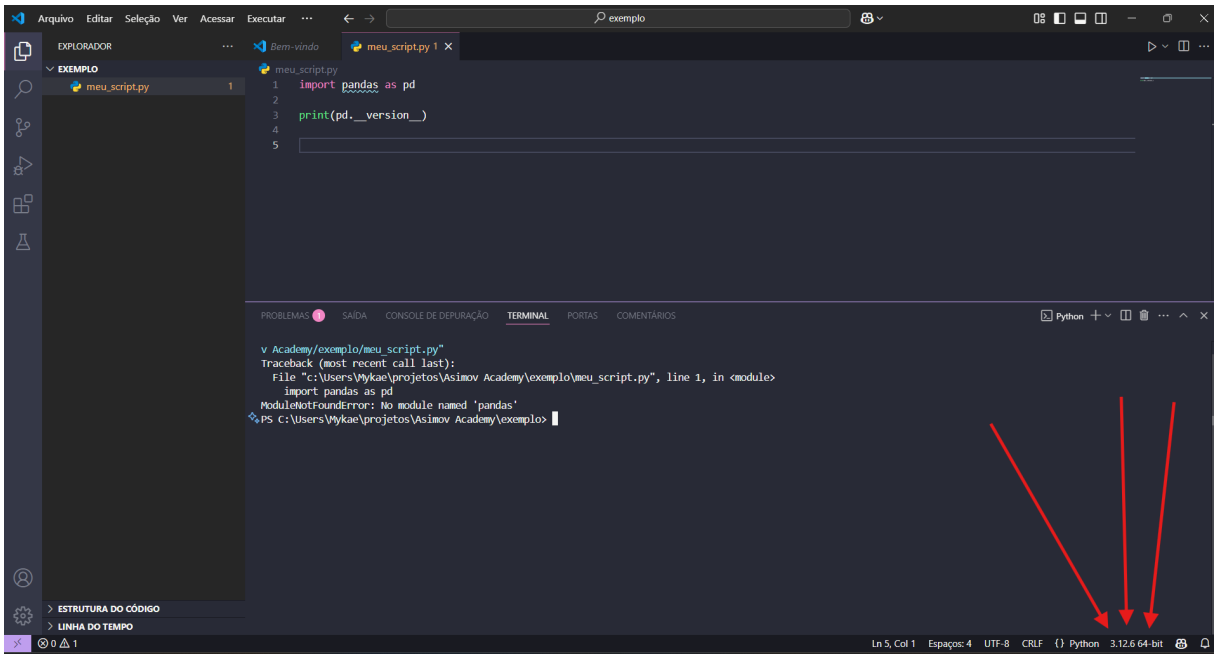


Figure 130: Localização interpretador

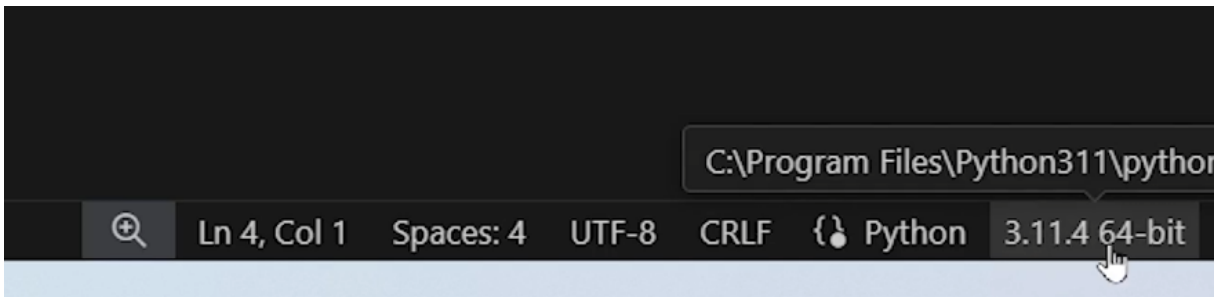


Figure 131: Localização interpretador

Clique nele para abrir a lista de interpretadores disponíveis.

- Selecione a opção correspondente ao ambiente virtual do projeto (geralmente localizado em `venv\Scripts\python.exe`).

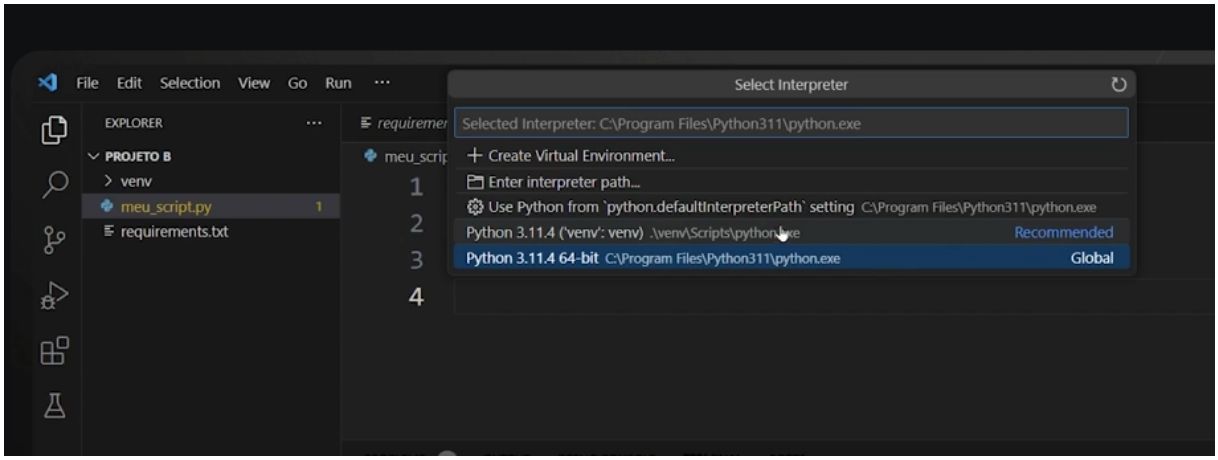


Figure 132: Seleção do interpretador

- Caso o interpretador correto não apareça, clique em `Enter interpreter path...` e navegue até o executável correto.

5. **Executando o script novamente**

   - Agora que o interpretador correto foi selecionado, rode o script novamente e veja se a biblioteca está funcionando corretamente.

Selecionando o Interpretador no PyCharm

6. **Abrindo o projeto**

- No PyCharm, clique em `File` > `Open` e selecione a pasta do projeto.
- Exemplo: Vamos abrir a pasta do projeto A.
- Confirme que confia no projeto quando solicitado.

7. **Criando um novo script**
   - Crie um novo arquivo Python chamado `meu_script.py`.
   - Adicione o seguinte código:

     ```
     import pandas as pd
     print(pd.__version__)
     ```

   - Salve e execute o arquivo.
8. **Verificando o interpretador**
   - No PyCharm, o interpretador padrão pode ser identificado no canto inferior direito da tela.
   - Se o interpretador correto já estiver selecionado, a execução ocorrerá sem problemas.
9. **Alterando o interpretador**
   - Clique no interpretador no canto inferior direito e selecione `Add Interpreter`.
   - Escolha `Virtualenv Environment` e selecione o caminho correto para o `python.exe` dentro do diretório `venv` do projeto.
   - Outras opções incluem usar um interpretador global ou um ambiente `conda` se disponível.
10. **Executando o script novamente**
    - Agora, com o interpretador correto, execute o script novamente e confira se a biblioteca está funcionando corretamente.

Conclusão

Selecionar corretamente o interpretador de Python dentro das IDEs é essencial para evitar erros de importação de módulos. Tanto no VS Code quanto no PyCharm, o processo é simples, mas requer atenção aos detalhes. Sempre que encontrar um erro de `ModuleNotFoundError`, verifique se sua IDE está usando o interpretador correto.

Agora que entendemos esse processo, nas próximas aulas abordaremos outras questões relacionadas à importação de códigos para facilitar ainda mais o seu trabalho com Python. Vamos para a próxima aula!

19. Importando arquivos com código – a variável name

Nesta aula, vamos abordar um conceito essencial para a estruturação de projetos Python: a variável `__name__`. O objetivo é entender como evitar problemas ao importar arquivos e como controlar a execução de códigos dentro de diferentes scripts.

O problema da importação de scripts

Imagine que temos dois scripts Python dentro de um mesmo projeto:

- **script_b.py**:

```
x = 1
print(x)
```

- **script_a.py**:

```
from script_b import x
print(f"O valor de x é: {x}")
```

Se rodarmos o `script_b.py`, o resultado será:

```
1
```

Porém, se rodarmos o `script_a.py`, a saída será:

```
1
O valor de x é: 1
```

Por que isso acontece? Ao importar x de `script_b.py`, o Python executa todo o conteúdo do arquivo importado, inclusive a linha `print(x)`. Isso pode ser problemático em projetos maiores.

A solução: Usando `__name__`

Para evitar a execução de comandos desnecessários ao importar um arquivo, usamos a variável `__name__`. O Python define automaticamente essa variável para cada arquivo executado:

- Se o script for executado diretamente, `__name__` será igual a `"__main__"`.
- Se o script for importado, `__name__` conterá o nome do arquivo.

Podemos modificar `script_b.py` para evitar a execução indesejada:

```
x = 1

def main():
    print(x)
```

```
if __name__ == "__main__":
    main()
```

Agora, ao rodarmos `script_b.py`, teremos:

```
1
```

E ao rodarmos `script_a.py`, a saída será apenas:

```
O valor de x é: 1
```

O bloco `if __name__ == "__main__"` impede que o `print(x)` seja executado quando `script_b.py` for importado, mantendo o código mais organizado e controlado.

Conclusão

A variável `__name__` é um mecanismo fundamental para diferenciar códigos que devem ser executados diretamente ou apenas quando importados. Isso evita execuções indesejadas e melhora a modularidade do seu projeto Python.

Agora que você compreende esse conceito, podemos partir para desafios mais complexos na próxima aula!

20. Estrutura do projeto e erros de importação

Quando estamos trabalhando com projetos mais complexos em Python, com várias pastas, subpastas e módulos, é comum enfrentar problemas relacionados a importações de pacotes e funções. Este capítulo visa explicar como organizar um projeto grande de Python de maneira eficiente e como solucionar problemas de importação que podem surgir, principalmente quando lidamos com pacotes e módulos em diferentes níveis de pastas.

Neste exemplo, vamos explorar uma situação em que temos um projeto com um pacote e diversos módulos, e como configurar o ambiente de forma que as importações funcionem corretamente em diferentes contextos.

Estrutura do Projeto

1. **Criando a Estrutura do Projeto**

   Vamos criar uma estrutura de projeto que inclui um ambiente virtual e um pacote. A estrutura inicial do nosso projeto será a seguinte:

```
My Project/
│
├── venv/          # Ambiente virtual (para gerenciar dependências)
│
└── meu_projeto/   # Pasta do código
    ├── modulo_a.py  # Módulo A
    ├── modulo_b.py  # Módulo B
    └── modulo_c.py  # Módulo C
```

Figure 133: Estrutura das pastas

No VScode, a estrutura deve ser mais ou menos essa aqui:

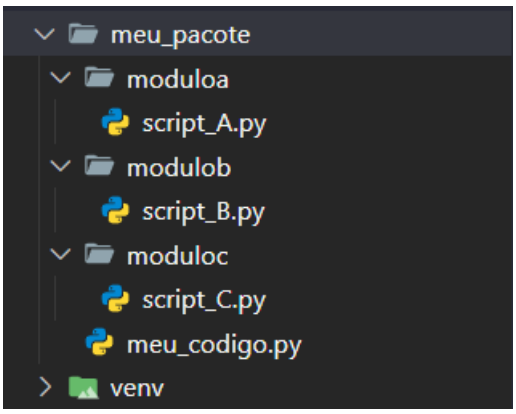


Figure 134: Estrutura das pastas

Dentro de cada módulo (A, B e C), teremos simples variáveis com valores numéricos. Aqui está um exemplo de como seria o `modulo_a.py`:

```
varA = 1
```

A ideia é que, em um projeto real, esses módulos teriam funcionalidades mais complexas.

2. **Criando o Script Principal**

No arquivo principal, vamos importar essas variáveis dos módulos e somá-las para verificar se as importações estão funcionando corretamente.

```
from meu_pacote.modulo_a import varA
from meu_pacote.modulo_b import varB
from meu_pacote.modulo_c import varC

def func():
    return varA + varB + varC

if __name__ == "__main__":
    print("Testando a função:", func())
```

Esse código deve funcionar sem problemas, desde que as importações estejam corretamente configuradas dentro da estrutura do projeto.

Problema Comum de Importação

1. **Criando o Script `main.py`**

   Quando você tenta criar um script principal (`main.py`) fora do pacote, você pode encontrar um erro de importação, já que o Python não consegue encontrar o módulo a partir da estrutura de

pastas. Isso ocorre porque o `main.py` está em um nível de diretório diferente e o Python tenta importar os módulos a partir do diretório atual.

Por exemplo, ao tentar rodar:

```
from meu_pacote.modulo_a import varA
```

O Python pode não encontrar o `meu_pacote` se ele não estiver configurado corretamente.

2. **Solução Temporária**

Uma forma de contornar esse problema seria adicionar a importação completa:

```
from my_project.meu_pacote.modulo_a import varA
```

Isso faz com que o Python encontre o módulo corretamente. No entanto, isso cria um novo problema, pois agora, ao rodar o código dentro do pacote, as importações não funcionarão mais corretamente.

Solução Definitiva: Instalar o Projeto no Ambiente Virtual

A maneira correta de resolver esse problema é instalar o próprio projeto dentro do ambiente virtual. Isso permitirá que o Python encontre os pacotes e módulos independentemente de onde o script está sendo executado.

Para isso, vamos adicionar o arquivo `__init__.py` ao diretório do pacote e configurar o projeto com a biblioteca `setuptools`.

- Crie o arquivo `__init__.py` dentro do diretório `meu_pacote` para indicar ao Python que essa pasta é um pacote.
- Instale o `setuptools` no ambiente virtual:

```
pip install setuptools
```

- Crie um arquivo `setup.py` na raiz do projeto para configurar o pacote:

```
from setuptools import setup, find_packages

setup(
    name="myproject",
    packages=find_packages(),
)
```

Agora, execute o comando para instalar o projeto de forma que o Python consiga importar os pacotes corretamente:

```
pip install -e .
```

Isso instala o projeto no ambiente virtual de forma editável, permitindo que você faça alterações no código e as veja refletidas imediatamente.

Conclusão

O processo de importar pacotes em projetos grandes de Python pode gerar erros, especialmente quando há uma estrutura de pastas complexa. A solução para evitar esses problemas é instalar o projeto dentro do ambiente virtual, o que garante que as importações sejam feitas sempre a partir da raiz do projeto.

Além disso, utilizar a biblioteca `setuptools` e o arquivo `setup.py` ajuda a configurar corretamente o ambiente para importação consistente, independentemente do local onde o script esteja sendo executado.

Agora, com essa configuração, as importações funcionarão corretamente, não importa onde o código seja executado dentro do projeto.