Uma introdução à linguagem Python

Marcelo Bezerra

15 de outubro de 2023

Conteúdo

Listings

Capítulo 1

Instalação

Muitos sistemas são entregues ao usuário final com alguma versão da linguagem Python pré-instalada. Você pode tentar executar o comando `python`, que inicia uma sessão interativa do interpretador de comandos da linguagem, e, assim, checar se o mesmo já se encontra instalado.

A seguir cobriremos uma maneira de se instalar a linguagem em sistemas Linux, tanto Debian GNU/Linux quanto Fedora Linux. Se você é um usuário do Microsoft Windows, por favor considere a adoção de um sistema de código aberto. Até lá, recomendamos o uso do WSL (Windows Subsystem for Linux). No macOS, pode-se usar o gerenciador de pacotes Homebrew.

1.1 Distribuições Linux

Em sistemas Linux os softwares são, geralmente, chamados de *pacotes*, distribuídos através de *repositórios* e gerenciados por meio de *gerenciadores de pacotes*. Cada distribuição Linux possui seu próprio gerenciador de pacotes (e, sim, o Slackware Linux possui o `pkgtools`). Em caso de dúvida, consulte a documentação para o seu sistema.

1.2 Debian GNU/Linux

No Debian, vamos utilizar o gerenciador de pacotes `apt`.

```
1 > sudo apt install python3
```

Listing 1.1: Instalando Python 3 no Debian

1.3 Fedora Linux

No Fedora, vamos utilizar o gerenciador de pacotes `dnf`.

```
1 > sudo dnf install python3
```

Listing 1.2: Instalando Python 3 no Fedora

Capítulo 2

Gerenciamento de módulos

Este capítulo mostrará como instalar e utilizar ferramentas necessárias para o funcionamento de suas aplicações. Tenha em mente que a linguagem Python é utilizada para diversos fins, e que a escolha de como você irá gerenciar suas dependências vai depender de como você planeja distribuir o seu programa. A maneira aqui apresentada se aplica ao gerenciamento de ambientes de desenvolvimento e depuração de aplicações dos mais diversos matizes, incluindo aquelas para a Internet.

2.1 Ambientes virtuais

A primeira coisa a se fazer é certificar-se de que seja possível criar ambientes virtuais que possam isolar as dependências da aplicação em desenvolvimento do restante do sistema. Imagine, por exemplo, que você esteja utilizando uma certa biblioteca para a linguagem Python, numa versão X em seu trabalho, numa versão Y para um trabalho da faculdade, e, finalmente, numa versão Z, mais recente do que as outras, para acompanhar o desenvolvimento de tal biblioteca. Eis a pergunta que surge: como manter tantas versões de uma mesma biblioteca em um único sistema? A resposta é simples: *ambientes virtuais*.

2.1.1 Instalando o pip

`pip` significa *package installer for Python* e você pode usá-lo para instalar pacotes disponíveis no PyPi, que significa *Python Package Index*. Existe mais de uma maneira de se instalar o `pip` em seu sistema. Aqui, vamos instalá-lo através do gerenciador de pacotes em nosso sistema.

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

Listing 2.1: Instalando o `pip` no Debian

```
1  > sudo dnf install python3-pip python3-wheel
```

Listing 2.2: Instalando o `pip` no Fedora

2.1.2 Criando ambientes virtuais

A criação de ambientes virtuais é muito simples. A fim de sermos mais didáticos, iremos supor que estejamos interessados em aprender a desenvolver sistemas *web* com Django. Para começar, vamos criar uma pasta chamada *projects*, que irá conter todos os projetos desenvolvidos neste computador. O comando para isto é o `mkdir`:

```
1 > mkdir -p projects
```

Listing 2.3: Criando o diretório de projetos

Feito isto, vamos mudar para o diretório recém criado e, então, criaremos um novo diretório, digamos, *meu_primeiro_projeto_django*, para abrigar o desenvolvimento da nossa aplicação. Fica assim:

```
1 > cd projects
2 > mkdir -p meu_primeiro_projeto_django
```

Listing 2.4: Criando o diretório que abrigará nossa aplicação

A seguir, adentramos o diretório recém criado, **criamos** e **ativamos** um ambiente virtual chamado `venv` (pouco criativo, certo?).

```
1 > cd meu_primeiro_projeto_django
2 > python3 -m venv venv
```

Listing 2.5: Criando o diretório que abrigará nossa aplicação

A opção $-m$ é como se evoca um módulo de maneira não interativa em Python. O primeiro `venv` é, portanto, o módulo de mesmo nome em Python enquanto que o segundo `venv` é o nome escolhido para o ambiente virtual em questão. Procedendo assim, dá-se o nome de `venv` para todos os ambientes virtuais existentes na máquina. Mas, então, não há o risco de confusão? Não, não há. Lembre-se de que o `venv` recém criado reside na pasta chamada *meu_primeiro_projeto_django*.

Agora, é hora de ativar o ambiente virtual e instalar as dependências para o nosso projeto, neste caso Django. A maneira de se fazer isto é com o comando `source`.

```
1 > source venv/bin/activate
```

Listing 2.6: Ativando o ambiente virtual – parte 1

Enquanto isto irá funcionar quando o *shell* utilizado for o **bash**, não podemos afirmar com certeza que todas as pessoas no planeta estarão a utilizá-lo, não é mesmo? E quando o shell em questão for o excelente **ksh**, por exemplo? Substitua o comando `source` por um ponto:

```
1 > . venv/bin/activate
```

Listing 2.7: Ativando o ambiente virtual – parte 2

Feito isto, vamos finalmente instalar Django:

```
1 > pip install django
```

Resumindo:

1. Criamos uma pasta para abrigar a nossa aplicação;
2. Entramos nesta pasta e, então, criamos um ambiente virtual para instalar as dependências do projeto de maneira isolada do restante do sistema;
3. Ativamos o ambiente virtual criado no passo anterior e, por fim,
4. Instalamos as dependências.

São muitos passos, não? Além disso, muitos dirão que estamos no século XXI, que usar o terminal é coisa do passado e, principalmente, que tem-se que digitar muita coisa! Prezados, *makefile*'s são nossos amigos! Como qualquer explicação sobre o GNU Make escapa aos propósitos iniciais destas notas, gostaríamos apenas de dizer que `make` pode ser utilizado para automatizar o processo de se transformar códigos em produtos. Em verdade, esta explicação não faz juz ao incrível poder de `make`. Por exemplo, eu o utilizei para criar o PDF que você lê agora!

Para dar um singelo exemplo, digamos que o nosso projeto utilize, além do já mencionado Django, o excelente `black`. Nós podemos, então, criar um arquivo chamado *requirements.txt* na raiz de nosso projeto e com ele listar as dependências em nosso projeto, uma por linha. Ficaria assim:

```
black
django
```

Listing 2.8: Um exemplo de requirements.txt

Feito isto, poderíamos criar um `makefile`, também na raiz de nosso projeto, com a seguinte regra:

```
ready:
  python3 -m venv venv; \
  . venv/bin/activate; \
  pip install -U pip; \
  pip install -r requirements.txt; \
  deactivate
```

Listing 2.9: Exemplo de makefile

Deste modo, quando quisermos criar um ambiente virtual e com ele instalar as dependências de nosso projeto, bastaria agora digitar um único comando:

```
> make ready
```

Listing 2.10: Reduzindo-se a quantidade de comandos digitados

A esta altura alguns poderiam perguntar:

Mas, eu precisaria criar um makefile com uma regra como esta para cada novo projeto que eu iniciar?

Boa pergunta! Sim, isto é verdade. No meu caso, tenho snippets de códigos em meu editor de textos favorito, que automatizam o processo de criação de makefiles (e tantas outras coisas) nas linguagens de programação que mais utilizo, como C, Python e LaTeX (e, sim, eu penso em LaTeX como uma linguagem de programação; o que difere neste caso é o propósito do programa que estou a escrever).

Eu não quero causar a impressão equivocada de que o `make` apenas serve para se reduzir a quantidade de comandos digitados ao terminal, pois este é um efeito do seu uso. O GNU Make é muito eficiente em detectar o que não foi modificado desde a última compilação e, portanto, economiza recursos da máquina e reduz tempo de execução de modo que em grandes projetos o seu uso se torna obrigatório (ou, então, alguma de suas alternativas).