Módulo de Programação Python

Trilha Python - Aula 2: Ambientes Virtuais: Criando e Gerenciando Módulos e Pacotes

Objetivo: Introduzir o conceito de ambientes isolados em Python para desenvolvimento de projetos. Apresentar ferramentas para criar e gerenciar ambientes virtuais. Gerenciar pacotes e módulos nos ambientes virtuais.

Na aula anterior foi apresentado o conceito de módulos e pacotes em **Python** e como utilizar os mesmos para estruturar o código. Muitos módulos e pacotes estão disponíveis em para desenvolvimento de diferentes tipos de aplicações e isso é uma da grandes vantagens de se utilizar **Python**. Mas como instalar novos pacotes e gerenciar os que já estão disponíveis?

A ferramenta mais utilizada hoje para gerenciamento de pacotes em **Python** é o `pip`.

Quando você instala **Python** em seu computador, ele vem com uma ampla coleção de pacotes e módulos adicionais que você pode usar a qualquer momento. Trata-se da chamada **Biblioteca Padrão Python** que é formada por um extenso conjunto de pacotes integrados que fornece soluções padronizadas para muitos problemas que ocorrem na programação cotidiana.

No entanto, às vezes você não encontrará a ferramenta que procura em **Python** ou em sua biblioteca padrão. Nesses casos, você precisará adquirir novas ferramentas em outro lugar. Felizmente, na internet é é possível encontrar centenas de milhares de pacotes desenvolvidos em **Python** para todos os propósitos. A grande maioria desses pacotes é de uso gratuito.

Embora pacotes de terceiros possam ser hospedados em locais diferentes, o repositório mais popular e abrangente é o **Python Package Index** (**PyPi**). Com mais de 300.000 pacotes **Python** disponíveis, PyPI é um repositório online gigante de pacotes aceitos pela comunidade **Python**.

Depois de identificar o pacote que procura, você precisará baixá-lo e instalá-lo em seu computador para usá-lo. Neste momento que os gerenciadores de pacotes entram em ação.

Gerenciador de pacotes `pip`

Um sistema de gerenciamento de pacotes é uma ferramenta que automatiza o processo de instalação, atualização, configuração e remoção de pacotes de um computador de maneira consistente.

Os gerenciadores de pacotes são projetados para eliminar a necessidade de instalações e atualizações manuais, garantindo assim que um pacote seja instalado junto com todas as dependências necessárias para funcionar. Da mesma forma, como os gestores de pacotes aproveitam as informações armazenadas em repositórios de pacotes certificados, como PyPi e Anaconda, eles garantem a integridade e autenticidade dos pacotes.

O gerenciador de pacotes mais popular para **Python** é o `pip` (um acrônimo de "pip Install Packages") que se tornou hoje a ferramenta padrão para instalar pacotes **Python** e suas dependências de maneira segura.

Pip é uma ferramenta poderosa e fácil de usar que permite gerenciar pacotes **Python** usando vários comandos. Embora pip use PyPi como repositório padrão para buscar pacotes, ele também tem a capacidade de instalar pacotes de outras fontes, incluindo:

- Sistemas de controle de versão como Github, Mercurial, Subversion e Bazaar.
- Arquivos de requisitos. Normalmente, os pacotes Python requerem vários pacotes para serem executados. Para instalar todos os pacotes necessários de uma vez, o pip usa o chamado require.txt, que contém uma lista dos pacotes necessários, bem como as versões corretas.
- Arquivos de distribuição. Esses são arquivos versionados e prontos para instalação contendo pacotes **Python**, módulos e outros arquivos de recursos necessários para o funcionamento de um pacote. Eles vêm em duas formas:
  - Source distribution (geralmente abreviada para "sdist")
  - Wheel distribution (geralmente abreviada para "wheel")

Para usar o pip e começar a gerenciar pacotes, primeiro você precisa garantir que ele esteja instalado no seu computador.

In [ ]:
```
#sudo apt-get install python3-pip
```

In [ ]:
```
#pip --version
```

Aqui costumamos ter um problema quando utilizamos python num sistema sem permissão de administrador. Precisamos de um ambiente isolado e seguro onde possamos trabalhar com **Python** e seus módulos e pacotes.

Ambiente virtual ou isolado

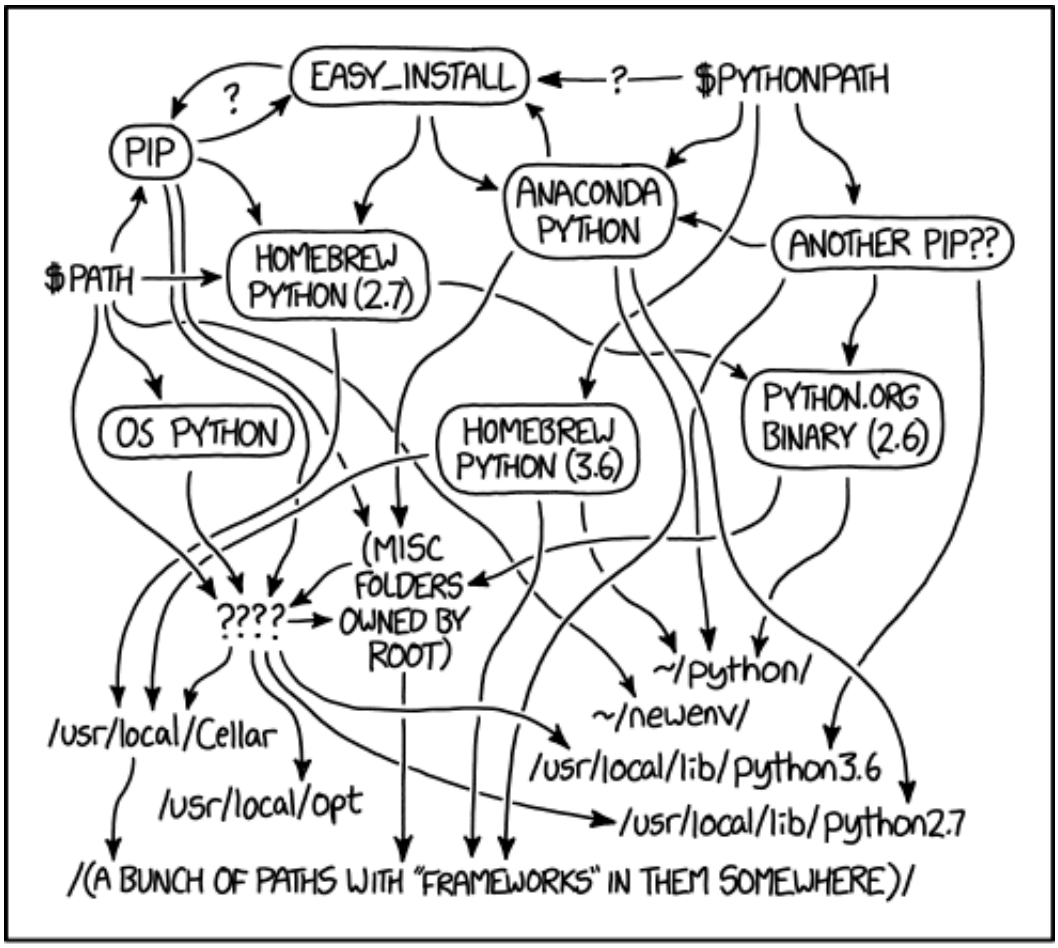
Para o desenvolvimento de aplicativos em **Python**, muitas vezes se faz necessário usar pacotes e módulos que não fazem parte da biblioteca padrão.

Às vezes, os aplicativos precisarão de uma versão específica de uma biblioteca, porque:

- o aplicativo pode exigir que um bug específico, que existe em versões anteriores da biblioteca, tenha sido corrigido;
- o aplicativo pode ser escrito usando uma versão obsoleta da interface da biblioteca;

Isso significa que pode não ser possível que uma instalação do **Python** atenda aos requisitos de desenvolvimento de cada aplicativo. Se, por exemplo, o aplicativo A precisar da versão 1.0 de um módulo específico, mas o aplicativo B precisar da versão 2.0, os requisitos estarão em conflito e a instalação da versão 1.0 ou 2.0 deixará um dos aplicativos incapaz de ser executado.

Tentar resolver este tipo de conflitos no ambiente padrão do seu sistema pode gerar um verdadeiro caos.



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

A solução para esse problema é criar um ambiente virtual, uma árvore de diretórios independente que contenha uma instalação de uma versão específica de **Python**, além de vários pacotes adicionais.

O processo de desenvolvimento de diferentes aplicativos pode então usar diferentes ambientes virtuais. Desta forma o exemplo anterior pode ser resolvido se o aplicativo A for executado no seu próprio ambiente virtual com a versão 1.0 instalada, enquanto o aplicativo B é executado em outro ambiente virtual com a versão 2.0. Se o aplicativo B exigir que o módulo seja atualizada para a versão 3.0, isso não afetará o ambiente do aplicativo A.

Seja organizado e crie ambientes virtuais **Python** e tente não poluir o ambiente padrão do seu sistema, ou ele ficará confuso.

Devido ao grande potencial dos ambientes virtuais, surgiram inúmeras ferramentas para ajudar no gerenciamento dos mesmos.

Vamos explorar alguns deles hoje.

venv

Vamos começar usando o `venv`. Trata-se de um módulo que foi integrado a partir de **Python 3.3** e versões posteriores e que permite criar ambientes Python isolados. É semelhante ao `virtualenv`, mas é instalado por padrão com Python. de fato o comando `virtualenv` é um "superconjunto" do comando nativo `python -m venv`. Na verdade, se você deseja a velocidade e os recursos adicionais do `virtualenv`, basta substituir por `virtualenv` em qualquer lugar onde você encontrar `python -m venv`.

Para começar vamos criar um ambiente na pasta `.venv`

In [ ]:
```
#sudo apt-get install python3.11-venv
#python3 -m venv .venv
```

Analisando este comando de forma mais detalhada, ele significa: use o executável **Python** como a versão **Python** que você deseja usar no ambiente virtual. Isso pode ser feito chamado python, python3, python3.8 ou python3.9, dependendo das versões que estejam disponíveis no seu sistema. Em seguida, diga para ele executar o módulo `venv`, seguido do nome do diretório no qual você deseja que o ambiente virtual resida. Aqui escolhemos a pasta `.venv` mas poderíamos usar uma pasta visível como `venv`. De fato você pode usar qualquer nome de diretório que desejar em vez de .venv.

Agora vamos ativar o ambiente virtual

In [ ]:
```
# source .venv/bin/activate
```

Repare que o prompt do sistema muda quando o ambiente virtual está ativado. A forma, claro, depende do **SO** e do terminal que você está utilizando.

Repare também que, independentemente de você ter utilizado o comando `python3` para criar o ambiente isolado, agora basta utilizar `python` para executar seu *script*. Podemos então explorar o ambiente virtual.

In [ ]:
```
#(.venv) $ python
import sys
sys.executable
```

Para desativar o ambiente virtual:

In [ ]:
```
#(.venv) $ deactivate
```

Dentro do ambiente virtual podemos instalar os pacotes e módulos necessários com `pip`

In [ ]:
```
#pip install numpy
#python -m pip install --upgrade pip
```

Pacotes e dependências devem ser instalados e então você pode importar e usar nos seus programas. Você pode fazer logoff, esquecer o **Python**, voltar em algumas semanas e reativar seu ambiente virtual. Os pacotes ainda serão instalados. Mas apenas neste ambiente virtual. Ele não polui o ambiente Python do seu sistema ou outros ambientes virtuais.

Outra vantagem de utilizar ambientes isolados desta forma é que os mesmos podem ser removidos ou desinstalados com facilidade. Basta apagar a pasta onde foram criados.

In [ ]:
```
# rm -rf .venv
```

Muito importante lembrar da lista de pacotes antes de remover o ambiente. Podemos fazer isto com `pip freeze` ou `pip list` . Com a lista é possível recriar o ambiente posteriormente.

In [ ]:
```
#pip freeze
```

virtualenv

O `virtualenv` é uma ferramenta desenvolvida por terceiros que permite criar ambientes Python isolados. Ele não é instalado por padrão com **Python**, mas pode ser facilmente adicionado usando `pip`.

A ferramenta `virtualenv` é muito semelhante ao `python -m venv`. Na verdade, o módulo `venv` do **Python** é baseado em `virtualenv`. No entanto, usar `virtualenv` no lugar de `python -m venv` tem algumas vantagens:

- Usar `virtualenv` é geralmente mais rápido que `python -m venv`
- Ferramentas como `pip`, `setuptools` e `wheel` costumam ser melhor atualizadas e armazenadas em cache.

In [ ]:
```
#sudo pip install virtualenv
#virtualenv --upgrade-embed-wheels
```

Agora podemos utilizar os mesmos comandos que apresentamos anteriormente com `venv`

In [ ]:
```
#virtualenv .venv
```

pipenv

O `pipenv` (https://pipenv.pypa.io/en/latest/) é uma ferramenta que combina `virtualenv` com `pip`. Ele permite criar ambientes virtuais e gerenciar pacotes Python para seus projetos. Ele foi projetado para ser mais fácil de utilizar do que usar `virtualenv e pip` separadamente.

In [ ]:
```
# sudo apt-get install pipenv
```

Para utilizar esta ferramente, primeiramente é necessário criar uma pasta para o projeto e então ativar o ambiente virtual:

In [ ]:
```
# pipenv shell
# para sair do shell
# exit
```

Reparem que ele funciona um pouco diferente que as ferramentas anteriores. Agora é necessário sair do *shell* com o comando `exit`. Ele também cria uma pasta `Pipfile` para gerenciar os pacotes deste ambiente isolado. A instalação de pacotes também é feita de forma diferente:

In [ ]:
```
# pipenv install numpy
```

O comando anterior vai instalar o `numpy` e suas dependências na pasta `Pipfile` da pasta atual.

Vamos a voltar a falar do `pipenv`, quando falemos de gerenciamento de projetos e ambientes virtuais com `poetry`.

Existem outras inúmeras soluções para gerenciamento de ambientes isolados **Python** como:

- `pyvenv` é um script que vem com **Python 3.3** e versões posteriores que permite criar ambientes virtuais. É um *wrapper* em torno do módulo `venv` e fornece a mesma funcionalidade dele.
- `pyenv` é uma ferramenta de terceiros que permite gerenciar várias versões do Python na mesma máquina e criar ambientes virtuais. Não está relacionado ao `venv` ou `virtualenv`, mas fornece funcionalidade semelhante.

Finalmente temos o Conda.

Conda

Conda não é apenas mais um pacote **Python** ou gerenciador de ambiente, trata-se de um ecossistema **Python** alternativo. O repositório de pacotes para **Conda** é diferente do repositório **PyPI** usado pela maioria dos gerenciadores de pacotes/projetos. O repositório **Conda** possui cerca de 1.500 pacotes. O repositório **PyPI** tem aproximadamente 150.000. Entretanto o **Conda** pode ser usado com `pip` se for necessário.

Para criar um novo ambiente virtual, especifique o nome do ambiente virtual que deseja usar e, opcionalmente, a versão do python:

In [ ]:
```
#conda create --name tic18 python=3.10
```

Uma vez criado, é possível ativar o ambiente virtual.

In [ ]:
```
#conda activate tic18
```

Já para desativar o ambiente virtual

In [ ]:
```
#conda deactivate
```

A instalação de novos pacotes pode ser feita via repositório conda

In [ ]:
```
#conda install numpy
```

Ou via comando `pip` que pucha do **PyPi**.

Uma vez instalados os ambientes virtuais podemos utilizar estes ambientes isolados tanto do **VSCode** quanto de **Jupyter** ou do **Jupyter-lab**.

Selecione as ferramentas que vai utilizar no futuro e monte os ambientes virtuais para seus projetos. Voltaremos a falar de gerenciamento de pacotes em breve.

Vamos apenas revisar o uso do comando `pip`

Para instalar um pacote podemos utilizar de forma simples:

In [ ]:
```
#pip install numpy
```

De esta forma o `pip` procura e instala a última versão disponível no repositótio do **PyPi** para a versão de **Pythopn** que que está sendo utilizada.

Lembre sempre que `pip` é um módulo **Python** e que, como vimos na aula anterior, pode ser executado também via:

In [ ]:
```
#python -m install numpy
```

Se o projeto precisa de uma versão específica do pacote isto pode ser colocado utilizando a seguinte sintaxes:

In [ ]:
```
#python -m install numpy==1.20.3
```

Já se a restrição é para versões maiores que uma versão específica e ainda num determinado intervalo

In [ ]:
```
#pip install numpy>=1.20.3,<1.24.2
```

Pip permite que você instale uma lista de pacotes de uma vez usando um arquivo de requisitos.

In [ ]:
```
#pip install -r requirements.txt
```

Onde o arquivo de requisitos contem os pacotes e, eventualmente as versões específicas `numpy==1.24.1 pandas==1.5.3`

Eventualmente pode ser necessário atualizar um determinado pacote Neste caso utilizamos:

In [ ]:
```
#pip install --upgrade numpy
```

O próprio pip pode precisar ser atualizado

In [ ]:
```
#pip install –upgrade pip setuptools wheel
```

Para desinstalar um pacote

#pip uninstall numpy

Agora pode brincar de instalar e desinstalar pacotes e módulos no seus ambientes isolados, sem afetar poluir ou corromper o ambiente nativo do sistema. Boa sorte.