Sumário

[Trabalhando com Python 1](#_Toc139141916)

[Python package manager 1](#_Toc139141917)

[Virtual environment 1](#_Toc139141918)

[Atualizar o pip 1](#_Toc139141919)

[Instalar virtualenv (Python version< 3.3) 2](#_Toc139141920)

[Criar um ambiente virtual 2](#_Toc139141921)

[Instalar pacotes 2](#_Toc139141922)

[PipFile 2](#_Toc139141923)

[Pipenv 3](#_Toc139141924)

# Trabalhando com Python

Conforme visto utilizando o Node.Js, será demonstrado como configurar o Python para trabalhar conforme as características de um projeto Node.js, sendo assim, é importante ter conhecimento de como:

* Utilizar o comando **pip**
* Criar um **ambiente virtual (versão >3 venv)**
* Configurar um **requirements.txt**, ou listagem de pacotes
* Comando **pip freeze**
* Como instalar pacotes pelo **requirements.txt**

# Python package manager

Há dois tipos de comandos, conforme este [link](https://www.binarystudy.com/2023/05/difference-between-pip-and-pip3.html#:~:text=Another%20difference%20between%20pip%20and,x.):

* pip - utilizado para versões 2.x.x do Python
* pip3 - utilizado para versões atuais do Python 3.x.x

# Virtual environment

Ambientes virtuais são uma boa forma de conseguir criar configurações específicas para o mesmo projeto rodar em diferentes máquinas sem depender das configs globais já setadas na máquina, o que pode conflitar com os módulos utilizados durante o desenvolvimento da mesma aplicação. Assim, é sempre **recomendado criar um ambiente virtual enquanto desenvolver uma aplicação em Python**, conforme a [documentação](https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/):

## Atualizar o pip

py -m pip install --upgrade pip  
py -m pip --version

## Instalar virtualenv (Python version< 3.3)

py -m install --user virtualenv  
Para versões >3.3, o venv já vem instalado

## Criar um ambiente virtual

py -m venv env  
Cria um ambiente na pasta env  
Posso excluir ele com o .gitignore

source ./env/Scripts/activate bash  
env\Scripts\activate cmd  
Ativar o ambiente virtual  
Agora pip instala pacotes somente neste ambiente

Where python  
Verificar o interpretador do Python

deactivate  
Sair do ambiente

## Instalar pacotes

py -m pip install requests  
py -m pip install “requests>=2.0.0,<3.0.0”  
py -m pip install requests-2.18.4.tar.gz  
py -m pip install --upgrade requests

Utilizando o requirements.txt  
py -m pip install -r requirements.txt  
pip install -r requirements.txt  
  
Exportar as dependências instaladas  
py -m pip freeze > requirements.txt  
pip freeze > requirements.txt  
env/Scripts/python.exe env/Scripts/pip.exe freeze > requirements.txt

# PipFile

O PipFile é uma proposta criada para corrigir alguns dos problemas que o requirements.txt trouxe com o gerenciamento de dependências. Muitos utilizaram por bastante tempo o requirements.txt, porém é difícil ter ideia quais pacotes no arquivo tem dependências diretas, ou é uma dependência transitiva. Além disso, se não pinar estas dependências transitivas, qualquer atualização feita pelo criador na mesma pode quebrar a sua aplicação se for atualizada no seu projeto.

Logo o Pipfile resolve este problema, conhecido como deterministic build. É basicamente um package.json, no qual há scripts, dependências de desenvolvimento e produção (Pipfile), além do .lock com as dependências transitivas (Pipfile.lock). A única forma de cria-lo e gerenciar dependências por ele é utilizando o Pipenv.

A estrutura de um arquivo Pipfile utiliza do formato [TOML](https://github.com/toml-lang/toml), no qual tem a mesma ideia do JSON (facilidade de leitura para humanos) e o YAML (possibilita inserir comentários), porém com uma semântica que facilita a leitura, sem ser no formato de objetos

# [Pipenv](https://pipenv.pypa.io/en/latest/)

É a ferramenta que gerencia ambientes virtuais juntando com todas as funcionalidades do python e do pip. Além disso, gerencia todos os pacotes em um Pipflie e no Pipflie.lock. A ideia é manter um ambiente consistente, sem se preocupar com a administração de pacotes, versões e ambientes virtuais.

Passa-se a utilizar pipenv ao invés de pip, não há mais necessidade do requirements.txt. Foca em buscar as versões mais antigas das dependências quando baixadas pela primeira vez, nos quais são pinadas no .lock. Possibilita um workflow mais dinâmico (.env)

pip install pipenv --user

pip install --user --upgrade pipenv

pipenv --python 3.10 cria o ambiente

Ele não mostra a pasta com o ambiente virtual igual ao venv, mas o ambiente pode ser executado a partir dos seguintes comandos.

pipenv install <pacote> (install + update) instala o pacote, criando os arquivos Pipfile

pipenv uninstall <pacote> desinstala pacotes

pipenv sync (install) instala as dependências do .lock sem alterar versões

pipenv lock (update) cria e atualiza o .lock para as versões mais recentes

pipenv update <package> atualiza o .lock de uma dependência e suas sub-dependências e instala as atualizações

pipenv upgrade <package> update mas não instala as atualizações

pipenv shell executar o venv, sair com o exit

pipenv run command executar scripts

pipenv scripts ver scripts disponiveis

pipenv graph visualizar o gráfico de dependências

pipenv run python app.py

pipenv run pip freeze

pipenv install -r path/to/requirements.txt criar um pipfile a partir de um requirements

pipenv requirements > requirements.txt exportar dependências

Segundo a documentação neste [link](https://pipenv.pypa.io/en/latest/advanced/#community-integrations), é possível criar integrações contínuas para deploy das aplicações criadas com o pipenv, sendo possível utilizar plataformas do [Heroku](https://www.heroku.com/python), [Platform.sh](https://platform.sh/marketplace/python/) e Github