Python Technology

Agenda

1. Checking Python version
2. PyPI & pip
3. Virtual environments
4. Where do packages reside?
5. PyInstaller

Checking Python version

Before we start

Before starting this chapter, check the following:

```
$ python --version
Python 2.7.16
$ python3 --versionPython 3.7.4
$ pip --version
$ pip3 --version
pip 19.1.1 from /usr/lib/python3.7/site-packages/pip (python 3.7)
```

There is a chance you have older versions of Python (2.7) and pip installed.

The many versions

Python allows you to have multiple versions co-existing. You can specify the exact version by its aliases, like so:

```
$ python
$ python2
$ python3
$ python2.7
$ python3.6
$ python3.7
```

Python Package Index & pip

Python Package Index & pip

> pip is the reference Python package manager. It's used to install and update packages.
>
> *python.org*

> The Python Package Index (PyPI) is a repository of software for the Python programming language.
> PyPI helps you find and install software developed and shared by the Python community
>
> *pypi.org*

Python Package Index & pip

Put simply, you can use pip to install, upgrade, downgrade and uninstall packages from various sources. PyPI is the go-to repository where pip downloads packages from.

Pip usage

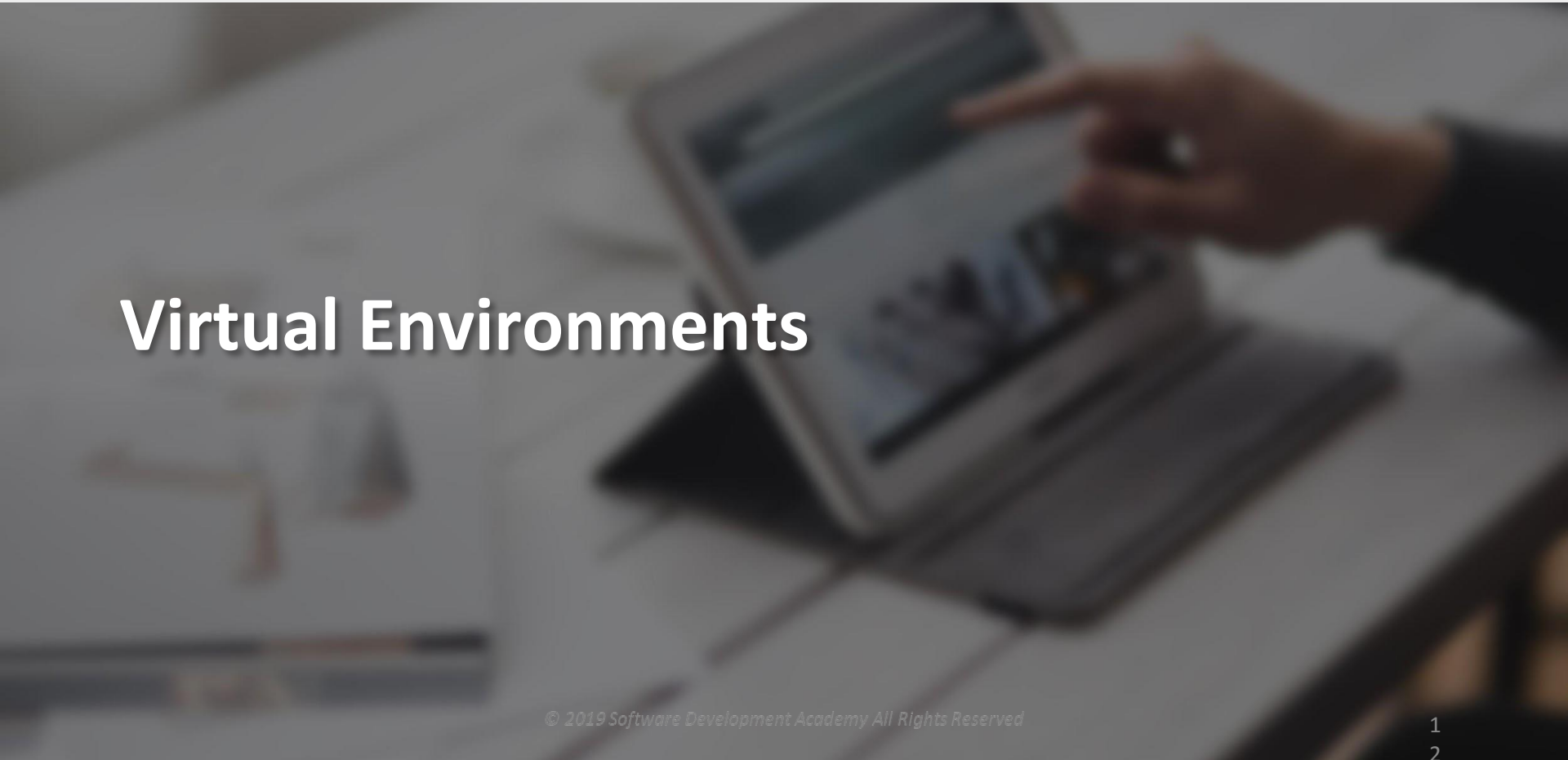
1. List installed packages using `pip list`
2. Install new packages using `pip install <package1> <package2>...`
3. You can specify installed package's version:
   `pip install x==1.0, pip install x>1.0, pip install x<=1.5`
4. You can upgrade packages using `pip install --upgrade <package>`
5. Uninstall package using `pip uninstall <package>`

Example – pip usage

```
$ pip list
Package    Version
---------- -------
pip        19.0.3
setuptools 40.8.0
$ pip install requests
Collecting requests
  (...)
Successfully installed certifi-2019.6.16 chardet-3.0.4 idna-2.8 requests-2.22.0 urllib3-1.25.3
$ pip uninstall requests
Uninstalling requests-2.22.0:
  (...)
  Successfully uninstalled requests-2.22.0
$ pip install requests==2.21.0
  (...)
$ pip list
Package    Version
---------- ---------
certifi    2019.6.16
chardet    3.0.4
idna       2.8
pip        19.0.3
requests   2.21.0
setuptools 40.8.0
urllib3    1.24.3
$ pip install --upgrade requests
   (...)
Successfully installed requests-2.22.0
```

Requirements.txt

1. It is customary for projects to keep their package requirements in `requirements.txt` files
2. pip can “freeze” installed package versions, generating such a file, by using `pip freeze > requirements.txt`
3. pip can then install requirements based on such file by using `pip install -r requirements.txt`



Virtual Environments

Virtual environments

1. pip installs packages in one place by default
2. Using global pip for all your projects is considered a bad practice
3. Installing too many packages globally can lead to package version conflicts
4. Luckily Python supports *virtual environments*

Virtual environments

5. A *virtual environment* is set up by creating separate site directories, containing a copy of Python interpreter and libraries
6. Virtual environments are often created per project
7. They provide activation scripts for various shells that “switch” their contexts to use environment’s Python interpreter and pip instead of system-wide one
8. Virtual environments can safely co-exist with different Python/package versions

Creating virtual environments

To create virtual environment, use `python -m venv <virtualenv_path>`. Virtual environments are often created inside a `venv` directory in project's root directory.

Exercise – creating and activating venv

1. Create a new directory called `my_project` and enter it.
2. Create a virtual environment for `my_project` using `python3.7 -m venv venv`
3. Assuming you are in your project's directory, run an appropriate command in your shell. Environment's name will appear next to your command prompt.

Platform	Shell	Command to activate virtual environment
Posix	bash/zsh	`$ source venv/bin/activate`
	fish	`$ . venv/bin/activate.fish`
	csh/tcsh	`$ source venv/bin/activate.csh`
Windows	cmd.exe	`C:\> venv\Scripts\activate.bat`
	PowerShell	`PS C:\> venv\Scripts\Activate.ps1`

Is your venv active?

Now that the environment is activated, you can install packages directly in it using pip. You can check active pip's path using `pip --version`.

Exercise – using venv

1. Install `django` package in your virtual environment.
2. Use `pip list` to verify what other packages are installed.

Deactivating venv

After you are done working in a particular environment, you can deactivate it using `deactivate`. This will not undo any work, it just resets context to use global interpreter and pip. You can always come back to your environment by activating it again.

Exercise – using deactivate

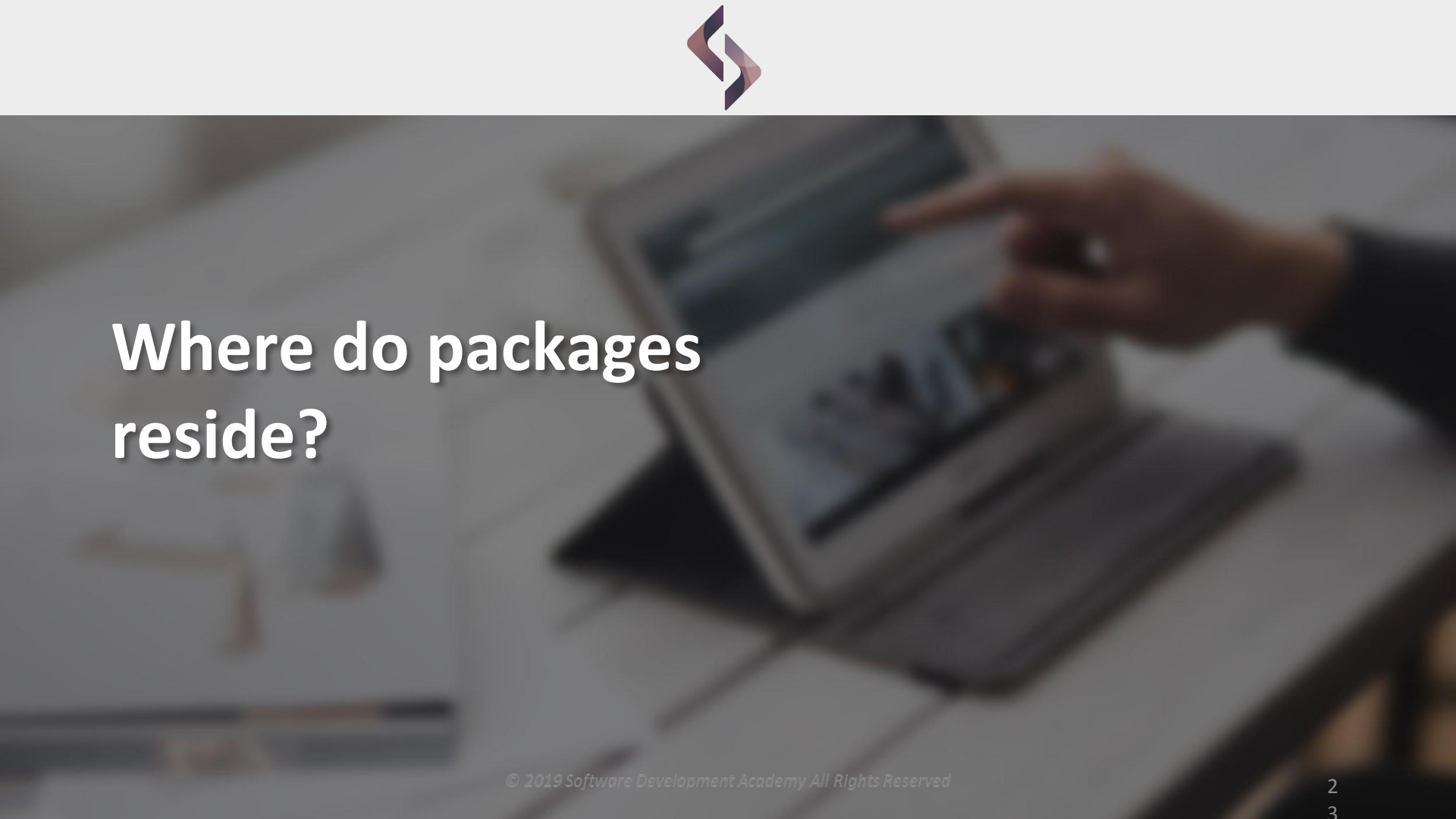
1. Deactivate your environment using `deactivate`
2. Re-activate it
3. Use `pip list` to verify the environment's state has not changed
4. Create a second virtual environment and install `flask` in it. Remember to deactivate current environment first!

Deleting venv

Deleting an environment is very simple. Make sure it is deactivated and simply remove its whole directory.

Exercise – delete your environment

1. Delete venv environment in `my_project` by removing its directory



Where do packages reside?

Package location

Now you know how to install and import packages. But what happens when you install a package? Where exactly is it installed? How to use your own libraries? The answer to that is Python Path.

Nomenclature

Just to clarify:

- When we say *a module*, we mean a single `.py` file
- When we say *a package*, we mean a directory containing one or more modules
- A *library* is not as well defined, but usually means *a package*.

Remember import?

```
import urllib # urllib.request.urlopen("https://google.com")
import urllib.request # utllib.request.urlopen("https://amazon.com")
from urllib import request # request.urlopen("https://ebay.com")
from urllib.request import urlopen # urlopen("https://wikipedia.com")
from urllib.request import * # urlopen("https://gmail.com")
```

What does import do?

```
from urllib.request import urlopen
```

1. `importlib` calls `import_module` function, which parses given path: `import urlopen object from request module of urllib package`
2. It looks in `sys.path` for `urllib` package
3. It executes urllib's `__init__` statements (if any)
4. It finds `request` module
5. It executes its `__init__` statements (if any)
6. It adds imported objects (`urlopen`) to `globals()`, so that they are available in importing module.

Exercise – 1-pythonpath/exercise-01

1. Examine given example. Look at its modules, `__init__.py` and `__main__.py` files.
2. Write a simple script (it has to be in `exercise-01` directory) which imports `happy_symbol` from `examplepackage.symbols`. What happens when you run it?
3. Try importing `from examplepackage.symbols import *`. What did it import?
4. Try executing `examplepackage: python3 -m examplepackage`. What does it do?

Looking for a package

1. `import` looks for packages in `sys.path`
2. `sys.path` is traversed in order to check whether it contains a directory named like the package
3. Found directory has to contain `__init__.py`, even if it is empty, to be recognized as package
4. Empty path `""` means current folder - this is how Python imports locally
5. You can extend `sys.path` by appending to it

Exercise – 1-pythonpath/exercise-02

1. Try printing `sys.path`
2. Append `"../exercise-01/"` to sys.path
3. Import `examplepackage`. It should work, now that the `path` is extended.

PYTHONPATH

- Appending to `sys.path` in code is not the only way to extend it.
- If `PYTHONPATH` environment variable is set, it will be used by Python and added to `sys.path` at script start
- Entries in `PYTHONPATH` have to be separated by system path separator - `;` for Windows, `:` for Linux
- `(...)/Python3.7/lib/site-packages` is where packages installed by `pip` reside

Exercise – 1-pythonpath/exercise-03

1. Set `PYTHONPATH`, adding full path to exercise-01 (in my case `/home/jakub/python-technology/exercise-01`). You can run `pwd` inside `exercise-01` folder to get the full path.
2. Run `python3 exercise-03.py`.

Summary

- Installed packages live in `dist-packages` or `site-packages` in `PythonX.Y/lib` directory
- Python searches for packages/modules in current folder and in `sys.path`
- You can extend `sys.path` directly in your code or by using `PYTHONPATH` environment variable
- To import your own modules, you have to:
  - Place them in the same folder you want to use them in OR
  - Add their parent path to `sys.path` in your code OR
  - Add their parent path to `PYTHONPATH`
- Remember to add `__init__` for your packages!

PyInstaller

Packaging using PyInstaller

Now that you know how to separate project's dependencies, let's see how you can package your project for distribution.
There are many projects providing such capabilities, but we will be using PyInstaller.

Installing PyInstaller

- `pip install pyinstaller`
- Be advised: PyInstaller is platform, architecture and interpreter specific.
- It means that projects packaged on 64-bit Linux distribution using Python 3.7 will run only on 64-bit Linux distributions, and will use packaged Python 3.7
- To package project for a different distribution/architecture or with different Python, you have to package it in that specific environment.

One-folder package

1. PyInstaller creates a one-folder package by default.
2. When you run `pyinstaller myscript.py`, it:
   - creates a `build` folder and writes temporary files there
   - creates a `dist` folder and prepares the package
   - writes a `myscript` (Linux) or `myscript.exe` executable to dist folder
3. The `dist` folder is your package ready for distribution
4. If you want to package your application once again, remove `build` and `dist` folders, then run `pyinstaller`

Exercise – package flask-basic

1. Enter `2-pyinstaller/flask-basic`
2. Try to package `flask-basic/main.py` using PyInstaller
3. Run resulting executable
4. Open `localhost:8080` in your browser

One-file package

- Try running `pyinstaller --onefile myscript.py` to package the whole application to a single executable.
- You can add additional files, like images or html files using:
  - `--add-data=<source>;<destination>` for Windows
  - `--add-data=<source>:<destination>` for Linux
- For example

```
pyinstaller --onefile --add-data="README:." --add-data="img.png:."
myscript.py
```

Packaging script

You can avoid needless repetition by creating a script:

```
import os
import PyInstaller.__main__

PyInstaller.__main__.run([
    "--name=package_name", "--onefile", "--windowed",
    f"--add-binary={os.path.join("resource", "path", "*.png")}",
    f"--add-data={os.path.join("resource", "path", "*.txt")}",
    f"--icon={os.path.join("resource", "path", "icon.ico")}",
    os.path.join("my_package", "__main__.py"),
])
```

Exercise – package flask-with-files

1. Enter `2-pyinstaller/flask-with-files`
2. Try to package `flask-basic/main.py` using PyInstaller
3. You have to package `templates` and `static` also
4. Run resulting executable
5. Open `localhost:8080` in your browser