Setting Up Reproducible Environments & Application Deploys

1. **Welcome & Course Overview** ✅
2. **Managing Third-Party Dependencies With pip** ✅
3. **Isolating Dependencies With Virtual Environments** ✅
4. **Finding Quality Python Packages** ✅
5. **Setting Up Reproducible Environments & Application Deploys**
6. Course Conclusion

Challenge

You're working on a Python program...

The project is ready for deployment

But what if the project includes third-party packages?

How can you make sure that someone else is getting the exact same set of dependencies?

Even slight version conflicts can make installing or deploying a Python program a frustrating experience

How to **reliably specify all of the dependencies** a Python program needs?

Challenge

Many environments your Python program can run in:

- Local dev environment
- Automated tests (CI)
- Deployment targets (staging/prod)

Challenge (cont'd)

Goal:

All environments should use the same set of dependencies to avoid surprises

How to make dependency installs **repeatable**?

Setting Up Reproducible Environments & Application Deploys

1. Introduction to Requirements Files
2. Capturing project dependencies
3. Restoring captured dependencies
4. Separating development and production dependencies
5. Requirements files best practices

Requirements files to the rescue

Introduction to Requirements Files

Requirements files

- `requirements.txt` (or `requirements.pip`)
- ≔ *"A list of 'pip install' arguments placed in a text file"*

Requirements files

- `requirements.txt` (or `requirements.pip`)
- ≔ *"A list of 'pip install' arguments placed in a text file"*

Example:

```
# This is a comment.
requests==2.13.0
schedule==0.4.2
```

Requirements files

- Capture all of the third-party dependencies a Python program needs to run
- They (usually) specify exact package versions
- Allow Python environments to be reproduced in exactly the same way on another machine or build environment (*repeatability*)

Capturing project dependencies

Capturing dependencies

```
$ pip freeze

$ pip freeze > requirements.txt
```

`pip freeze` captures **all** dependencies, including **secondary dependencies** and their **exact version numbers**

→ This is **important** for achieving repeatability

Restoring dependencies

Restoring dependencies

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

Capturing & Restoring dependencies

Quick Review

Capturing & Restoring dependencies

Step 1: Install necessary dependencies during development

```
$ pip install somepackage
```

Capturing & Restoring dependencies

Step 1: Install necessary dependencies during development

```
$ pip install somepackage
```

Step 2: Capture dependencies in requirements file

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

Capturing & Restoring dependencies

Step 1: Install necessary dependencies during development

```
$ pip install somepackage
```

Step 2: Capture dependencies in requirements file

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

Step 3: Restore dependencies from requirements file

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

Separating **development** and **production** dependencies

Challenge

Development and **Continuous Integration** environments need additional dependencies:

- testing frameworks, debuggers, profilers, ...

But: **Production** environment should run "lean and mean"

Separating development and production dependencies

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

```
|----------------------|         |----------------------|
| requirements-dev.txt |    +--->| requirements.txt     |
|----------------------|    |    |----------------------|
| -r requirements.txt  |----+    | requests==2.1.3      |
| pytest==2.0.0        |         | # ...                |
| # ...                |         |----------------------|
|----------------------|
```

Demo

Requirements files best practices

Requirements files best practices:

Versioning

- Use exact version (`requests==2.13.0`) most of the time
- Include secondary dependencies
- Some dev dependencies may go without a version specifier (e.g. `ipdb` debugger)

Requirements files best practices:

Naming

- Most popular choice: `requirements.txt` and `requirements-dev.txt`
- Also: `requirements.pip` or `requirements.lock`
- Typically placed in the *root folder* of the project

Requirements files best practices:

Comments

```
# This is a comment.
requests==2.13.0
schedule==0.4.2
```

- Good idea to use them
- Example: Explain `requirements.txt` vs `requirements-dev.txt` split

Requirements files best practices:

Ordering Dependencies

```
# Direct dependencies
# (sorted alphabetically)
Flask==0.12
requests==2.13.0
schedule==0.4.2

# Secondary dependencies
# (sorted alphabetically)
Jinja2==2.9.5
Werkzeug==0.12
```

Requirements files best practices:

Dev/Prod Split

```
|----------------------|         |----------------------|
| requirements-dev.txt |    +--->| requirements.txt     |
|----------------------|    |    |----------------------|
| -r requirements.txt  |----+    | requests==2.1.3      |
| pytest==2.0.0        |         | # ...                |
| # ...                |         |----------------------|
|----------------------|
```

Requirements files best practices:

1. Versioning
2. Naming
3. Comments
4. Ordering Dependencies
5. Dev/Prod Split

Setting Up Reproducible Environments & Application Deploys ✅

1. **Introduction to Requirements Files** ✅
2. **Capturing project dependencies** ✅
3. **Restoring captured dependencies** ✅
4. **Separating development and production dependencies** ✅
5. **Requirements files best practices** ✅

Summary

- Requirements files allow you to specify the **third-party dependencies** of a Python program.
- This makes dependency installs and application deployments **repeatable**.
- Dependencies can be **captured** (`pip freeze`) and **restored** (`pip install -r`) with **pip**.