Python Packaging and Project Structure

Joseph Hellerstein[1,3], **Bernease Herman[1]**, David Beck[1,2,3], Sam Gao[3]

[1]eScience Institute
[2]Chemical Engineering
[3]Computer Science

April 23, 2019
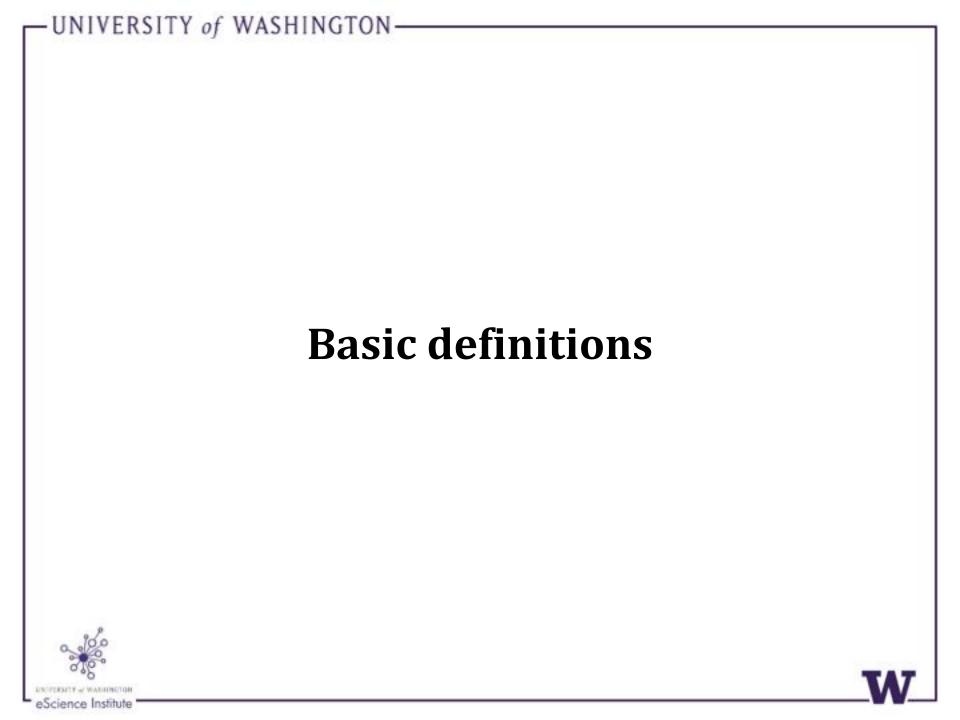
Agenda

1. Importing in Python

2. Dependency basics

3. Project directory structure for Python projects

4. Packaging Python projects and PyPI

Basic definitions

Basic Definitions

object: Most things in Python, e.g. function, variable, class.

module: A *.py script; carries the name as the file.

built-in module: A "module" that operates like a *.py module, but has been compiled directly into the Python interpreter; often created in C programming language.

namespace: A mapping of unique names to objects.

package: A directory-like concept that can hold multiple Python objects, subpackages under same namespace.

Importing packages

Imports

Order in which Python searches for modules to import:
1. built-in modules in the Python Standard Library (e.g. `math`, `os`)
2. modules or packages in a directory specified by `sys.path`:
    1. If the Python interpreter is run interactively:
        - `sys.path[0]` is the empty string `''`. This tells Python to search the current working directory from which you launched the interpreter, i.e. the output of `pwd` on Unix systems.
    2. If we run a script with `python <script>.py`:
        - `sys.path[0]` is the path to `<script>.py`
    3. directories in the `PYTHONPATH` environment variable
    4. default `sys.path` locations

Imports

There are 4 different syntaxes for writing import statements.
1. `import <package>`
2. `import <module>`
3. `from <package> import <module or subpackage or object>`
4. `from <module> import <object>`

Let `X` be whatever name comes after `import`.
- If `X` is the name of a module or package, then to use objects defined in `X`, you have to write `X.object`.
- If `X` is a variable name, then it can be used directly.
- If `X` is a function name, then it can be invoked with `X()`

Optionally, `as Y` can be added after any `import X` statement: `import X as Y`. The argument to the `import` function can be a single name, or a list of multiple names. Each of these names can be optionally renamed via `as`.

Absolute vs. Relative Imports

Absolute Imports (works in Python 2 and 3):

Import from the top-level Python package. e.g., `import <package>`

Relative Imports:
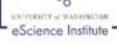
Import based on your `sys.path` location.

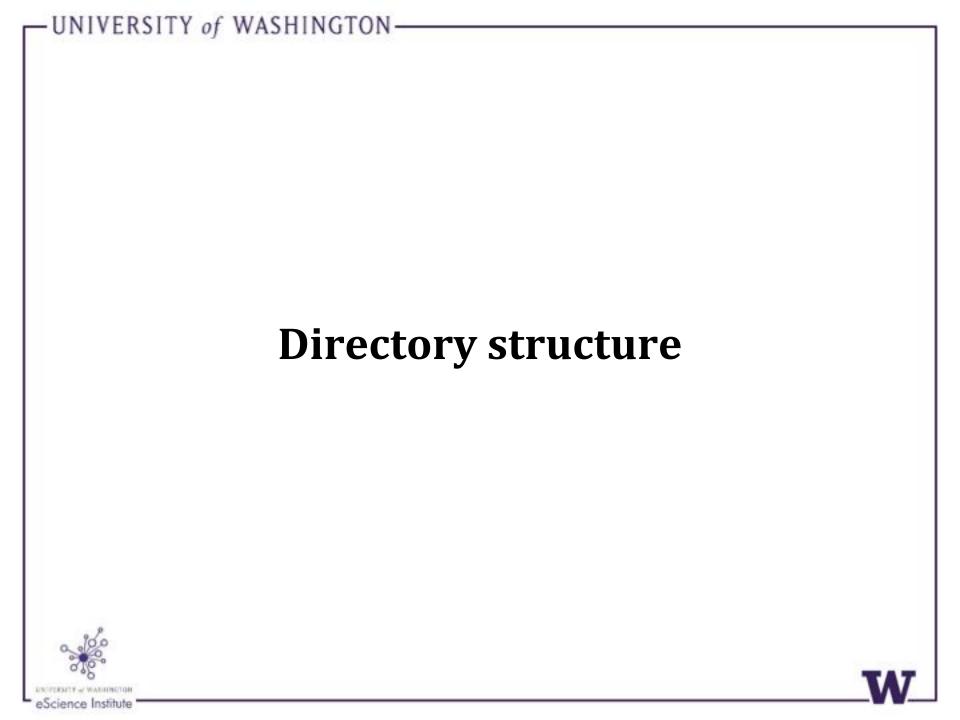### ***Explicit* Relative Imports (works in Python 2 and 3)**:
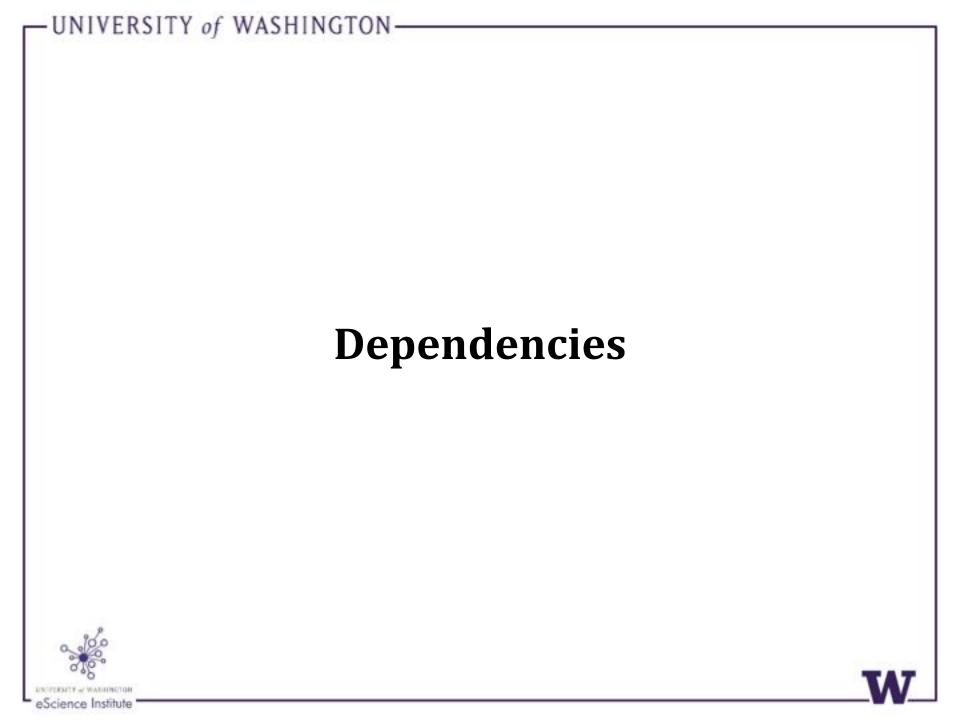
Import using . and .. notation.

***Implicit* Relative Imports (only Python 2)**:

Python searches down your path to find subpackages or modules you may be referring to. This discouraged and thus, discontinued.

Directory structure

Dependencies

Basic codebase in a Git directory

```
myproject/
    README.md
    LICENSE
    myproject/
        __init__.py
        core.py
       +submodule/
       -tests/
        __init__.py
        test_core.py
```

← This is your git repository, usually matching the name on Github

Basic codebase in a Git directory

`myproject/`

← This is your git repository, usually matching the name on Github

Basic codebase in a Git directory

```
myproject/
    README.md
```

← Markdown-formatted or plain text file describing the package.

Basic codebase in a Git directory

```
myproject/
    README.md
    LICENSE
```
← Software license specifying how others may use your code.

Basic codebase in a Git directory

```
myproject/
    README.md
    LICENSE
    myproject/
```

← The Python package, helpful for `import myproject` into separate namespace.

Basic codebase in a Git directory

```
myproject/
    README.md
    LICENSE
    myproject/
        __init__.py
```

← This module marks the directory (optional in 3.3+) as a Python package and is run upon import.

Basic codebase in a Git directory

```
myproject/
    README.md
    LICENSE
    myproject/
        __init__.py
        core.py
```

← Other modules in package containing importable code.

Basic codebase in a Git directory

```
myproject/
   README.md
   LICENSE
   myproject/
      __init__.py
      core.py
     +submodule/
```

← Packages can have subpackages (and sub-subpackages, etc.) to any depth. They contain their own `__init__.py` files.

Basic codebase in a Git directory

```
myproject/
   README.md
   LICENSE
   myproject/
      __init__.py
      core.py
     +submodule/
     -tests/          ← Unit tests go into their own submodule.
      __init__.py
      test_core.py
```
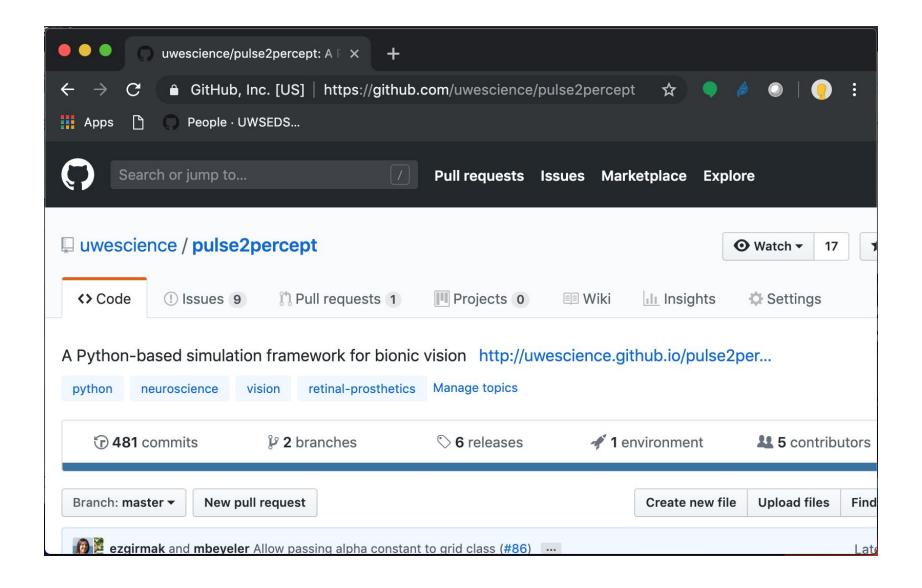
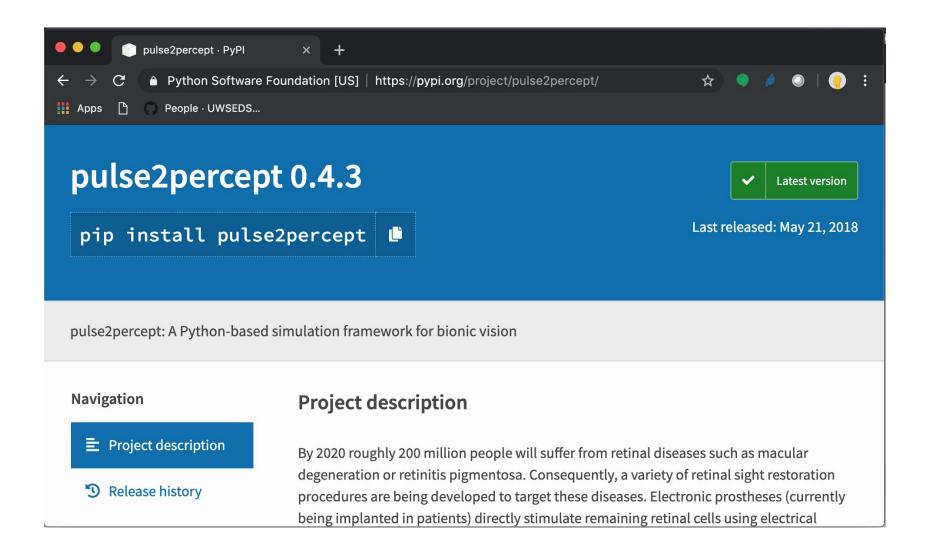# Packaging and distributing Python projects

GitHub

PyPI

Extending project structure for PyPI

```
myproject/
    README.md
    LICENSE
    myproject/
        __init__.py
        setup.py
        requirements.txt
        MANIFEST.in
        core.py
      +submodule/
      +tests/
```

← Contains metadata for the package. Often uses `distutils` or `setuptools` standards.

Can contain abstract vital dependencies.

Extending project structure for PyPI

```
myproject/
    README.md
    LICENSE
    myproject/
        __init__.py
        setup.py
        requirements.txt
        MANIFEST.in
        core.py
      +submodule/
      +tests/
```

← Contains absolute dependencies, especially useful in the application case.

Can generated using:

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

Extending project structure in PyPI

```
myproject/
    README.md
    LICENSE
    myproject/
        __init__.py
        setup.py
        requirements.txt
        MANIFEST.in
        core.py
       +submodule/
       +tests/
```

← Specify data and files that should also be packaged in addition to the Python modules.

Basic `setup.py` file

```python
import setuptools

setuptools.setup(
    name="example-pkg-your-username",
    version="0.0.1",
    author="Example Author",
    author_email="author@example.com",
    description="A small example package",
    install_requires=['docutils>=0.3'],
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/pypa/sampleproject",
    packages=setuptools.find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
)
```

Submitting your package to PyPI

Update your code and version number. Run your test suites and ensure your code works as intended.

Create your **source**, and if desired, **binary** distribution:

```
$ python setup.py bdist_egg upload [options]
$ python setup.py bdist_wininst [options]
$ python setup.py sdist [options]
```

Install `twine` package to submit builds to PyPI.
(Can install using conda install twine, pip install twine, etc.)

```
$ twine upload dist/*
```