Organizing and testing your Python code

Keeping it tidy

Modules and packages

What?

- **module**- a .py file
  - example from project 1: `loaddata.py`
- **package** - a collection of modules. AKA a folder with an `__init__.py`
- example from project 1: `project1/`

Why?

How?

- an `__init__.py` tells Python “this folder is a package”.
- It also gets executed whenever that package is imported.

Example layout for project 2:

```
project2/
    __init__.py  <- run on import
    data/        <- data lives in here
    loaddata.py  <- for loading from files + cleaning
    plotting.py  <- for making common plots
    reporting.py <- for printing out analysis results.
    models.py    <- for saving various models
    run.py       <- for importing stuff from the other modules
                    and running it from the command line
```

```
$> python project2/run.py
```

You can nest packages too. Could be helpful for splitting up work and trying things out on this project.

```
project2/
    __init__.py  <- run on import
    data/        <- data lives in here
    loaddata.py  <- for loading from files + cleaning
    plotting.py  <- for making common plots
    reporting.py <- for printing out analysis results.
    models/      <- for saving various models
        __init__.py
        brian.py
        irmak.py
    run.py       <- for importing stuff from the other
                    modules and running it from the
                    command line
```

```
from project2.models.brian import AwesomeModel
```

Assertions

Another way to fail helpfully

```
assert a.shape == b.shape

assert isinstance(datapoint, float)
```

Assertions

times these can be helpful, straight from the Python docs:

- checking parameter types, classes, or values
- checking data structure invariants
- checking “can’t happen” situations (duplicates in a list, contradictory state variables.)
- after calling a function, to make sure that its return is reasonable

Unit tests

Functions built around assertions

- These exercise your code in specific ways by defining expected behavior.
- In general, they can't exercise the code in *all* ways, so they can't *guarantee* it is bug free.
- When you refactor (add, optimize, "fix") your code later, tests give you confidence you didn't violate the expected behavior.

What makes a good unit test?

- tests one method
- well defined input and output
- doesn’t test things that are already tested

How do I make/run unit tests?

- Python has a builtin library, `unittest`
- Popular add-on test library: nose