Organizing your code

(Modules and Packages in Python)

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
```

Logging

Logging

How do I know when this thing is done? How do I know what happened?

A good way:

```
print “Done with %d of %d” % (i, total)
```

Better:

```
import logging

# will print a message to the console
logging.warning('Watch out!')

# will not print anything
logging.info('I told you so')
```

Levels of logging

- **DEBUG** - notes for you, while working on code
- **INFO** - notes that everything is going OK
- **WARNING** - something seems wrong but it's not urgent. running out of disk space, or data is probably too small, or things like that

- **ERROR** - something broke. things did not work.
- **CRITICAL** - oh 💩. things are seriously screwed. the whole program probably stopped running.

Best:

```
import logging
logging.basicConfig(filename=‘example.log’,
                    level=logging.DEBUG)
logging.debug(‘This message should go to the log file’)
logging.info(‘So should this’)
logging.warning(‘And this, too’)
```

That's just the beginning.

You can customize the information it includes (add things like dates), specify policies about logging to multiple files, and more.

Python `logging` howto

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 aren't *really* for *finding* bugs– the best way to do that is to use your functions.

They're for:

- defining the expectations of what your functions need to do
- ensuring that if you refactor your code later, you don't violate those expectations

What makes a good unit test?

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

How do I make/run unit tests?

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