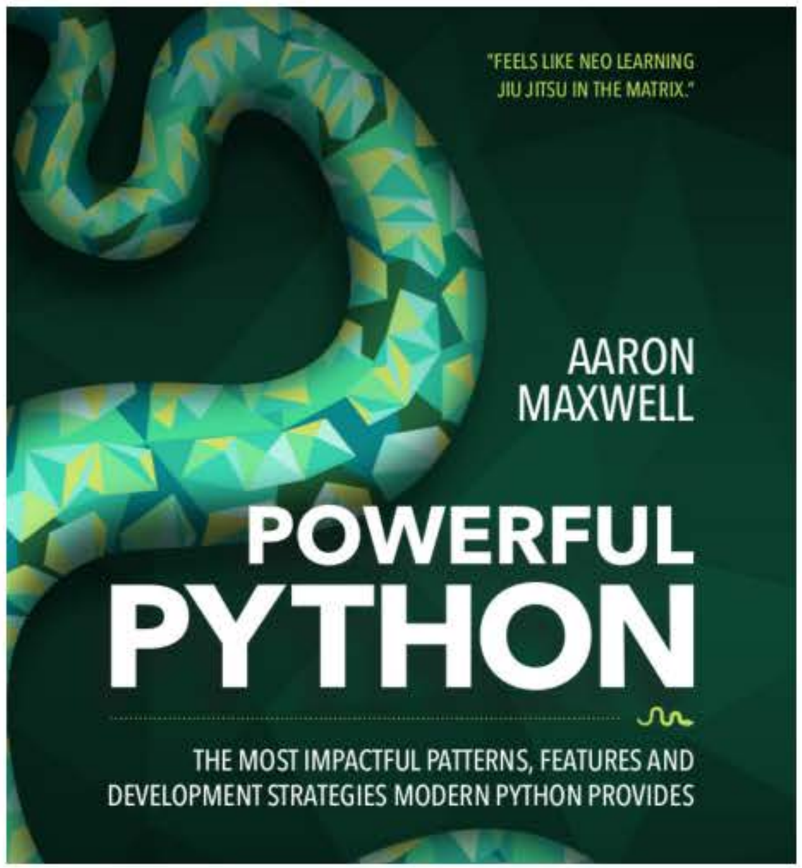
Python For Applications: Beyond Scripts

Welcome

I'm your host, Aaron Maxwell.

- Author of **Powerful Python**
- aaron@powerfulpython.com
- On twitter:
  - @powerfulpython (professional)
  - @redsymbol (personal)



Our focus in this class: Python's features that are **exceptionally and powerfully useful** when writing larger Python applications, rather than small scripts.

Python Scripts

Many people use Python for **scripts**.

- Small enough to easily fit in one file
- Often written and maintained by one developer
- Frequently written to solve a specific problem
- If in version control, work is done in mainline branch
- Often does NOT have unit tests
- Often is not code reviewed

Python Applications

For larger **applications**, it's a whole different game.

- Split across many different files. Too many LOC for one
- Worked on concurrently by a team of developers
- All work done in feature branches, merged into mainline
- Version control an **absolute must**
- Extensive test coverage
- Likely to be code reviewed
- Static code analysis

At this scale, sheer complexity becomes a barrier to progress.

We **explode** through that barrier by leveraging certain features of Python.

Broad Itinerary

Today:

- The key distinctions between **Scripts** and **Applications**
- Logging in Python
- Module Organization
- Dependency Management
- And... Homework!

Tomorrow:

- Errors and Exceptions
- Building Command-line Programs
- Test-driven development
- And... More homework!

How we will proceed

Download courseware ZIP:
https://powerfulpython.com/courseware-app.zip

What's included:

- PDF course book
- Slides
- README.txt with pointers
- Labs (i.e., programming exercises - more on that later)

Give you a break every hour (10 minutes).

Give me a thumbs up. (Let's try it now)

Ask questions anytime.

Python versions

Most code I show you will run in both Python 2 and 3.

Where it's different, I'll code in Python 3, and point out the differences. (There won't be many.)

You can do the programming exercises in either 3, or 2.7.

What makes perfect?

Practice, practice, practice.

- Practice syntax (typing things in)
- Practice programming (higher-level labs)

I expect you to do your part!

You **exponentially** get out of this what you put into it.

GO FOR IT.

Running the labs

Labs are the main programming exercises. You are given a failing automated test; your job is to write Python code to make it pass.

Simply run it as a Python program, any way you like. (For example, "python3 helloworld.py")

Run unmodified first, so you can see the failure report.

When done, click the thumb's up, and find someone to high-five.

Then: Move on to the extra credit.

Lab: helloworld.py

Let's do our first lab now: 'helloworld.py`

- In `labs/py3` for Python 3.x, or `labs/py2` for 2.7

When you finish:

- Thumbs up, so I know you're done.
- Find someone to high five.
- Proceed to `helloworld_extra.py`

You'll know the tests pass when you see:

```
*** ALL TESTS PASS ***
Give someone a HIGH FIVE!
```

Getting the most

We'll take some class time for each lab. You may not finish, but it's **critically important** that you at least start when I tell you to.

After we're done for the day, find time to finish all the main labs before tomorrow.

Solutions are provided. Use them wisely, not foolishly:

- After you get the lab passing, compare your code to the official solution.
- Other than that, don't look at them if you can avoid it.
- The more work you can do on your own, the more you will learn. Peek at the solution to get a hint when you really need it.

Optional (**only** for future master Pythonistas): Do all the extra labs as well, as soon as you can manage.

Logging

What's Logging For?

Logging provides useful data about what your application is doing.

It can be as detailed or coarse as you want, changeable at any time.

- Gives insight into **business logic** (unlike automated tracing tools)
- Valuable **telemetry** for monitoring
- Critical for **troubleshooting** and debugging

The larger your application, the more important logging becomes.

Two Ways

You can use Python's `logging` module in two broad ways.

The Basic Interface - Simpler. Useful for scripts and some mid-size applications.

Logger Objects - More complex to set up. But FAR more powerful, and invaluable with larger apps. Scales to any size.

We'll focus on the first.

The Basic Interface

The easiest way:

```
import logging
logging.warning('Look out!')
```

Save this in a script and run it, and you'll see this printed to standard error:

```
WARNING:root:Look out!
```

You call `logging.warning()`, and the output line starts with `WARNING`. There's also `error()`:

```
logging.error('Look out!')
```

```
ERROR:root:Look out!
```

Log Level Spectrum

debug: Detailed information, typically of interest only when diagnosing problems.

info: Confirmation that things are working as expected.

warning: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.

error: Due to a more serious problem, the software has not been able to perform some function.

critical: A serious error, indicating that the program itself may be unable to continue running.

Thresholds

Run this as a program:

```
logging.debug("Small detail. Useful for troubleshooting.")
logging.info("This is informative.")
logging.warning("This is a warning message.")
logging.error("Uh oh. Something went wrong.")
logging.critical("We have a big problem!")
```

And here's the output:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

What's missing? Why?

Log Levels

Python loggers have a *logging threshold*. And the default threshold is `logging.WARNING`.

You can change it with `logging.basicConfig()`.

```
logging.basicConfig(level=logging.INFO)
logging.info("This is informative.")
logging.error("Uh oh. Something went wrong.")
```

Run this new program, and the `INFO` message gets printed:

```
INFO:root:This is informative.
ERROR:root:Uh oh. Something went wrong.
```

Two Meanings

The phrase "log level" means two different things:

1) The **severity** of a message.

The order is `debug()`, `info()`, `warning()`, `error()` and `critical()`, from lowest to highest severity.

2) It can mean the **threshold** for ignoring messages.

Ignores everything less severe than `logging.INFO`:

```
logging.basicConfig(level=logging.INFO)
```

And this means "show me everything":

```
logging.basicConfig(level=logging.DEBUG)
```

Configuring Basic

Configure the basic interface by passing arguments to `logging.basicConfig()`.

`level`
The log level threshold, described above.

`format`
The format of log records.

`filename`
Write log messages to the given file, rather than stderr.

`filemode`
Set to "a" to append to the log file (the default), or "w" to overwrite.

Log Destination

By default, log messages are written to stderr.

Use `filename` to write them to a file instead:

```
# Appends messages to the file, one at a time.
logging.basicConfig(filename="log.txt")
logging.error("oops")
```

You can make your program *clobber* the log file each time by setting `filemode` to "w":

```
# Wipes out previous log entries when program restarts
logging.basicConfig(filename="log.txt", filemode="w")
logging.error("oops")
```

Or set it to "a" for "append". That's the default.

Prod Vs. Dev

```
log_file = 'myapp.log'
# mode can be set by an environment variable, command-line option, etc.
if mode == 'development':
    log_level = logging.DEBUG
    log_mode = 'w'
else:
    log_level = logging.WARNING
    log_mode = 'a'

logging.basicConfig(
    level = log_level,
    filename = log_file,
    filemode = log_mode)

logging.debug('debug message')
logging.warning('look out!')
logging.critical('we have a problem here')
```

What does this print out if mode is "development"? What if it's "production"?

Practice: `basiclog.py`

```
# Create a new file name `basiclog.py`. Type in this program:
mode = 'development'
log_file = 'myapp.log'
if mode == 'development':
    log_level = logging.DEBUG
    log_mode = 'w'
else:
    log_level = logging.WARNING
    log_mode = 'a'
logging.basicConfig(level=log_level, filename=log_file, filemode=log_mode)
logging.debug('debug message')
logging.warning('look out!')
logging.critical('we have a problem here')
```

Run to verify `myapp.log` contains `DEBUG`, `WARNING` and `CRITICAL`. Then change `mode` to "production", and re-run several times. Verify it appends only `WARNING` and `CRITICAL` to `myapp.log` each run.

EXTRA CREDIT: Make `mode` controlled by an environment variable.

Environment Switch

```
import os
mode = os.environ.get('MODE', 'production')
print('Running in mode: ' + mode)
if mode == 'development':
    # ...
```

```
$ python3 myprogram.py
Running in mode: production
```

```
$ export MODE=development
$ python3 myprogram.py
Running in mode: development
```

More here: http://powerfulpython.com/blog/nifty-python-logging-trick/

Common variation: Use a local config file.

Why not use print()?

If logging is new you to you, you may wonder why we don't just sprinkle the code with `print()` statements.

Essentially: some of those `print()` statements are more important than others. Python's logging module makes that hierarchy explicit.

And: you can change the threshold as needed.

Also: `print()` only writes to stdout. The logging module allows other destinations.

Efficiency

Freely write as many logging statements as you want, without affecting performance. Any lines below the threshold are cheaply ignored.

```
# Only log INFO messages or higher
logging.basicConfig(log_level = logging.INFO)
# Python essentially skips over this statement.
logging.debug("Received a message from X")
```

Often you will introduce many `debug()` statements while troubleshooting, and leave them in after the issue is resolved.

Interlude: String Formatting

"String Formatting" means inserting parameters into a template string at runtime, to get a final, calculated string.

Every language has it. C does it with `snprintf()`, `sprintf()`, `printf()`, etc.:

```c
int main() {
    char message[BUFFER_SIZE];
    char* template = "Your %s costs $%0.2f.";
    snprintf(message, BUFFER_SIZE, template, "cup of coffee", 1.75);
    printf("%s\n", message);
}
/*
  Output:
  Your cup of coffee costs $1.75.
 */
```

In Python, the situation with string formatting is complicated.

Modern String Formatting

In all modern versions of Python (2 and 3), you can use `str.format()`.

```
>>> template = "Your {} costs ${:0.2f}."
>>> output = template.format("cup of coffee", 1.75)
>>> print(output)
Your cup of coffee costs $1.75.
```

In 3.6 and later, you can use f-strings:

```
>>> item = "cup of coffee"
>>> price = 1.75
>>> print(f"Your {item} costs ${price:0.2f}.")
Your cup of coffee costs $1.75.
```

Percent Formatting

Python's original string interpolation. Inspired by C.

Uses `%s` for string; `%f` for float; `%d` for integer.

```
>>> template = "Your %s costs $%0.2f."
>>> output = template % ("cup of coffee", 1.75)
>>> print(output)
Your cup of coffee costs $1.75.

>>> # Or as a dictionary:
... data = {"item": "burger", "price": 6.95}
>>> print("Your %(item)s costs $%(price)0.2f." % data)
Your burger costs $6.95.
```

Largely outdated now, but still important in logging. Even in the latest Python 3.

Log Message Parameters

In many (most) log messages, you'll want to inject runtime data.

There's a good way and a bad way. The good way uses a *variant* of percent formatting:

```
logging.info("Your %s costs $%0.2f.", item, price)
```

- First argument: a percent-style template string
- 2nd and subsequent arguments: the parameters

The Bad Way

The good way again:

```
logging.info("Your %s costs $%0.2f.", item, price)
```

The bad way:

```
logging.info("Your %s costs $%0.2f." % (item, price))
```

Why is that bad?

Avoid Unnecessary Calculations

Imagine the log threshold is set to `logging.WARNING`. Then neither of these will emit any message:

```
logging.info("Your %s costs $%0.2f.", item, price)
logging.info("Your %s costs $%0.2f." % (item, price))
```

The second incurs a computational cost, to calculate the log message that's thrown away. But the first is very cheap.

This removes any hesitation to introducing log statements. Especially for INFO and lower, more tends to be better.

Why Percent Formatting?

Basically: Legacy constraints.

Original plan was to remove percent formatting completely. But it's not practical to convert old logging code to `str.format()` or f-strings.

Rather than render large groups of Python programs practically impossible to upgrade to Python 3, percent formatting is here to stay.

It's less painful to use the alternatives in NEW logging code. But I recommend you just use percent formatting.

Log Formats

Each call to `logging.debug()`, `.warning()`, etc. creates one log record - one line in the log file.

What info that record includes, and in what order, is determined by the *log format*.

```
logging.basicConfig(
    format="Log level: %(levelname)s, msg: %(message)s")
logging.warning("Collision imminent")
```

Run this as a program, and you get the following log line:

```
Log level: WARNING, msg: Collision imminent
```

Contrast with the default format:

```
WARNING:root:Collision imminent
```

Log Format Attributes

Attribute	Format	Description
asctime	`%(asctime)s`	Human-readable date/time
funcName	`%(funcName)s`	Function containing the logging call
lineno	`%(lineno)d`	The line number of the logging call
message	`%(message)s`	The log message
pathname	`%(pathname)s`	Full pathname of the source file of the logging call
levelname	`%(levelname)s`	Text logging level for the message (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*)
name	`%(name)s`	The logger's name

Lab: Basic Logging

Lab file: `logging/logging_basic.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `logging/logging_basic_extra.py`

Logger Objects

Richer interface to logging.

For details, read PythonForApps.pdf.

```
import logging
logger = logging.getLogger()
# StreamHandler defaults to using sys.stderr
console_handler = logging.StreamHandler()
logger.addHandler(console_handler)
# Now the file handler:
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)
logger.warning("This goes to both the console, AND log.txt.")
```

Benefits of Logging

"How Logging Made me a Better Developer", by Erik Hazzard

http://vasir.net/blog/development/how-logging-made-me-a-better-developer

- Visibility into code helps manage complexity
- Visibility after shipping
- Visibility while developing
- Communication
- Explicit comments

And I'll add: Dramatically accelerates troubleshooting.

Package & Module Structure

Python Modules

Python's module system is a delight - easy to use, well designed, and extremely flexible.

```
from greputils import grepfile
grepfile("pattern to match", "/path/to/file.txt")
```

Let's look at how it might evolve, from simple to rich and complex.

Start With A Little Script

```
# findpattern.py
import sys

def grepfile(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')

pattern, path = sys.argv[1], sys.argv[2]
for line in grepfile(pattern, path):
    print(line)
```

This also creates a module called `findpattern`.

Reuse Some Code

```
# finderrors.py
import sys
from findpattern import grepfile

path = sys.argv[1]
for line in grepfile('ERROR:', path):
    print(line)
```

```
$ python3 finderrors.py log1.txt
Traceback (most recent call last):
  File "finderrors.py", line 3, in <module>
    from findpattern import grepfile
  File "findpattern.py", line 10, in <module>
    pattern, path = sys.argv[1], sys.argv[2]
IndexError: list index out of range
```

What's the error?

Main Guard

The solution: use a "main guard". Original tail of `findpattern.py`:

```
pattern, path = sys.argv[1], sys.argv[2]
for line in grepfile(pattern, path):
    print(line)
```

Replace with:

```
if __name__ == "__main__":
    pattern, path = sys.argv[1], sys.argv[2]
    for line in grepfile(pattern, path):
        print(line)
```

Now both programs work:

```
$ python3 finderrors.py log1.txt
ERROR: out of milk
ERROR: alien spacecraft crashed
```

Separate Libraries

Let's refactor to have a common library, so we can add extra functions.

```
# greputils.py
# Search for matching lines in file.
def grepfile(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')
# Case-insensitive search.
def grepfilei(pattern, path):
    pattern = pattern.lower()
    with open(path) as handle:
        for line in handle:
            if pattern in line.lower():
                yield line.rstrip('\n')
```

Then `findpattern.py` and `finderrors.py` will have the line:

```
from greputils import grepfile
```

Expanding Libraries

Suppose `greputils` keeps adding functions, like `contains`:

```
def contains(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                return True
    return False
```

(And also `containsi`, for case-insensitive matching.)

As we add more, at some point we'll want to split up `greputils.py`. How?

Multifile Modules

There's more than one way to provide `greputils`. Let's split it into multiple files:

```
greputils/
greputils/__init__.py
greputils/files.py
greputils/contain.py
```

The `grepfile` and `grepfilei` functions are in `greputils/files.py`; `greputils/contain.py` has the `contains` and `containsi` functions.

The module directory generally must have an `__init__.py` file. This defines the interface for others importing the module.

__init__.py

```
from .files import (
    grepfile,
    grepfilei,
)
from .contain import (
    contains,
    containsi,
    )
```

Note:

- Split over multiple lines, using parenthesis.
- Uses "`from .files import`". "`from grepfile.files import`" will also work, but is less maintainable.
- "`from files import`" works in Python 2 only. But it's ambiguous, which is why Python 3 doesn't allow it.

Nesting

You can break up into different folders however you like:

```
greputils/
greputils/__init__.py
greputils/files.py
greputils/contain.py
greputils/net/__init__.py
greputils/net/html.py
greputils/net/text.py
greputils/net/json.py
```

```
# in greputils/__init__.py
# ...
from .net.html import (
    grep_html,
    grep_html_as_text,
)
```

Note the module interface doesn't change!

Antipattern Warning!

Sometimes you will see this:

```
from greputils import *
from otherlib import *
```

Don't do that - ESPECIALLY in your application code. It lets collisions and subtle bugs sneak in.

Importing

In your code you have a choice.

```
from greputils import grepfile
grepfile("pattern to match", "/path/to/file.txt")
```

Versus:

```
import greputils
greputils.grepfile("pattern to match", "/path/to/file.txt")
```

Use this to namespace when you're using many objects from the module.

Renaming Imports

When importing the module to namespace, use an abbreviation to save yourself some typing.

```
# Commonly used:
import numpy as np
import pandas as pd
import tensorflow as tf
```

Also works with imported functions, classes, etc.:

```
# Import a function and rename it:
from greputils import grepfilei as ci_grep

# Similar to:
from greputils import grepfilei
ci_grep = grepfilei
```

More on __init__.py

`__init__.py` can, when it makes sense, execute init code.

In general, avoid import-time side effects, unless you have a good reason to. Principle of Least Surprise

`__init__.py` can be an empty file. In that case, users will import sub-modules:

```
from greputils.files import grepfile

# Or:
import greputils.files
for line in greputils.files.grepfile(pattern, path):
    # ...
```

Terminology

The official terminology:

Reusable code in a single file is a **module**.

If that exact same code is split into multiple files, it's called a **package**.

Lab: Create A Package

Lab file: `modules/modules.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up

See also: `modules/greputils_start.py`

Unlike other labs, you do NOT modify `modules.py` at all. Instead, create a `greputils` directory, and populate it as a package, using the functions in `greputils_start.py`.

IMPORTANT: Make sure your current directory is the one containing `modules.py` and your `greputils` folder. If you're using an IDE, it might be easier to run this lab on the command line.

Dependency Management

Virtual Environments

Python lets you create a *virtual environment*. (A.k.a. "virtualenv" or "venv".)

Think of it as a lightweight container for a Python app:

- Dependencies precisely tracked and specified
- And kept in version control, for easy reproducability
- Two different Python applications with conflicting dependencies can peacefully coexist on the same machine.
- Python packages can be installed without requiring elevated system privileges.

Creating a VENV

For Python 3:

```
# The recommended method in Python 3.
$ python3 -m venv webappenv

# Does the same thing, but deprecated.
$ pyvenv webappenv
```

```
# Same as the above, in Python 2.
$ virtualenv webappenv
```

All create folder named `webappenv`.

Activating The VENV

The new folder contains lots of goodies.

To access them, activate the virtual environment:

```
# In macOS and Linux/Unix
$ source webappenv/bin/activate
(webappenv)$
```

On Windows:

```
C:\labs> webappenv\Scripts\activate.bat
(webappenv) C:\labs>
```

Notice your prompt has changed!

What's in the VENV?

With the virtual environment activated, you have your own local copy of the Python interpreter:

```
(webappenv)$ which python
/Users/sam/mywebapp/webappenv/bin/python
(webappenv)$ python -V
Python 3.6.0
```

And other tools, like `pip`:

```
(webappenv)$ which pip
/Users/sam/mywebapp/webappenv/bin/pip
```

Deactivate with `deactivate`:

```
(webappenv)$ deactivate
$ which python
/usr/bin/python
```

Multiple Environments

Each Python application can have its own virtual environment. Each can have a completely different version of Python.

```
$ cd /Users/sam/mediatag
$ virtualenv mediatagenv
$ source mediatagenv/bin/activate
(mediatagenv)$ which python
/Users/sam/mywebapp/mediatagenv/bin/python
(mediatagenv)$ python -V
Python 2.7.13
```

Using Pip

Modern Python provides an application called `pip`.

It lets you easily install 3rd-party Python libraries.

Your virtual environment has it:

```
$ source venv/bin/activate
(venv)$ python -V
Python 3.6.0
(venv)$ which pip
/Users/sam/myapp/venv/bin/pip
```

pip install

Use `pip install` to install opensource libraries:

```
pip install requests
```

These are fetched from PyPI: https://pypi.python.org/

Upgrade an installed package, with `-U` or `--upgrade`:

```
pip install --upgrade pip
```

Or uninstall:

```
pip uninstall requests
```

Reproducible Builds

Don't put the venv itself in version control. Some of those files are HUGE, and very platform-dependent.

Other problems:

- How to track exact version dependencies?
- How to specify the precise set of libraries, so that other devs can build it on their machines?
- How to manage upgrades (or even downgrades) over time?

`pip` provides a set of tools for this: `requirements.txt`.

Freeze

Start by using `pip freeze`:

```
(venv)$ pip freeze
requests==2.7.0
```

Prints what was installed from Pypi, one per line.
Place this in a file named `requirements.txt`:

```
(venv)$ pip freeze > requirements.txt
(venv)$ cat requirements.txt
requests==2.7.0
```

`requirements.txt` is what you actually check into version control!

Rebuild

You can pass these requirements to pip, using `-r`:

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv)$ pip install -r requirements.txt
Collecting requests==2.7.0 (from -r requirements.txt (line 1))
```

This allows the environment to be recreated on any server or machine.

Not a perfect solution, but often works very well.

Evolving Requirements

As dependencies evolve, update `requirements.txt` as you go along:

```
(venv)$ pip install django
Installing collected packages: pytz, django
Successfully installed django-1.11.7 pytz-2017.3

(venv)$ pip freeze > requirements.txt
(venv)$ git diff requirements.txt
diff --git a/requirements.txt b/requirements.txt
index ac2cf62..354542c 100644
--- a/requirements.txt
+++ b/requirements.txt
@@ -1 +1,3 @@
+Django==1.11.7
+pytz==2017.3
 requests==2.7.0

(venv)$ git add requirements.txt; git commit -m 'Install Django'
[master 8a3ec09] Install Django
 1 file changed, 2 insertions(+)
```

Naming the VENV

Two schools of thought. Pros and cons for each:

1) Pick a consistent name. "venv" is popular:

```
python3 -m venv venv
```

Upside: Consistency. Easy to make git ignore it.

2) Or pick a name that has to do with the application.

```
python3 -m venv mywebappenv
```

Upside: More descriptive prompt.

Downside: Inconsistent.

Errors And Exceptions

The Basic Idea

An **exception**:

- Is a way to interrupt the normal flow of code
- Can be **raised** at any point. Even in the middle of a line

```
import requests
from json.decoder import JSONDecodeError

resp = requests.get(api_url)
try:
    # Parse response body as JSON
    obj = resp.json()
except JSONDecodeError:
    # Response does not encode a valid JSON object.
    obj = {}
```

Flow Control

An exception often means an error. But it doesn't have to.

```
# Use "speedyjson" if available. If not fall back to "json" in the
# standard library.

try:
    from speedyjson import load
except ImportError:
    from json import load
```

The `load` function called in code will be the best available version.

Built-In Exceptions

Most errors you see in Python are exceptions:

- `KeyError` for dictionaries
- `IndexError` for lists
- `TypeError` for incompatible types
- `ValueError` for bad values
- `NameError` for an unknown identifier

Even `IndentationError` is an exception.

Multiple Except Blocks

```
try:
    value = int(user_input)
except ValueError:
    print("Bad value from user")
except TypeError:
    print("Invalid type (probably a bug)")
```

Often useful with logging:

```
try:
    value = int(user_input)
except ValueError:
    logging.error("Bad value from user: %r", user_input)
except TypeError:
    logging.critical(
        "Invalid type (probably a bug): %r", user_input)
```

`finally` and `except`

Sometimes you have a block of code that must ALWAYS be executed, no matter what.

```
conn = open_db_connection()
try:
    do_something(conn)
finally:
    conn.close()
```

You can also have one (or more) except clauses:

```
conn = open_db_connection(CONFIG)
try:
    do_something(conn)
except ValueError:
    logging.error("Bad data read from DB")
finally:
    conn.close()
```

Cloud Computing

```
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Imagine `running_job.wait()` raises a network-timeout exception. Now `fleet.terminate()` is never called.

Whoops. Expen$ive.

Save Your Bank Account/Job

Protect against this with `finally`:

```
fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
    running_job.wait()
finally:
    fleet.terminate()
```

Exceptions Are Objects

An exception is an instance of an exception class.

Often your `except:` clause will just specify the class. But sometimes you need the actual exception object.

Catch with "as":

```
try:
    do_something()
except ExceptionClass as exception_object:
    handle_exception(exception_object)
```

Exception Object Info

Exception objects have helpful info. The attributes vary, but it will almost always have an `args` attribute.

```
# Atomic numbers of noble gasses.
nobles = {'He': 2, 'Ne': 10,
  'Ar': 18, 'Kr': 36, 'Xe': 54}
def show_element_info(elements):
    for element in elements:
        print('Atomic number of {} is {}'.format(
              element, nobles[element]))
try:
    show_element_info(['Ne', 'Ar', 'Br'])
except KeyError as err:
    missing_element = err.args[0]
    print('Missing data for element: ' + missing_element)
```

```
Atomic number of Ne is 10
Atomic number of Ar is 18
Missing data for element: Br
```

Raising Exceptions

```
def positive_int(value):
    "Converts string value into a positive integer."
    number = int(value)
    if number <= 0:
        raise ValueError("Bad value: " + str(value))
    return number
```

Focus on the `raise` line:

- `raise` takes an exception object
- You instantiate `ValueError` inline

```
>>> positive_int(-7.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in positive_int
ValueError: Bad value: -7.0
```

(What does `positive_int("not a number")` do?)

Lab: Exceptions

Lab file: `exceptions/exceptions.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up.

When that's done: Open and skim through **PythonForApps.pdf** - just notice what topics interest you.

Example: Money

Raising exceptions can lead to more understandable error situations.

Here's a `Money` class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __repr__(self):
        'Renders the object nicely on the prompt.'
        return "Money({},{})".format(
            self.dollars, self.cents)
    # Plus other methods.
```

Money Factory

A factory helper function:

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

Huh?

What happens if you pass it bad input? The error isn't very informative.

```
>>> money_from_string("$140.75")
Money(140,75)
>>> money_from_string("$12.30")
Money(12,30)
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in money_from_string
AttributeError: 'NoneType' object has no attribute 'group'
```

Imagine finding this error deep in a stack trace. We have better things to do than decrypt this.

Better Errors

Add a check on the `match` object. If it's none, meaning `amount` doesn't match the regex, raise a `ValueError`.

```
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    # Adding the next two lines here
    if match is None:
        raise ValueError("Invalid amount: " + repr(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

More Understandable

```
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in money_from_string
ValueError: Invalid amount: 'Big money'
```

This is MUCH better. The exact nature of the error is immediately obvious.

Catch And Re-Raise

In an `except` block, you can re-raise the current exception.

Just write `raise` by itself:

```
try:
    do_something()
except ExceptionClass:
    handle_exception()
    raise
```

It's a shorthand, equivalent to this:

```
try:
    do_something()
except ExceptionClass as err:
    handle_exception()
    raise err
```

Interject Behavior

One pattern this enables: inject but delegate.

```
try:
    process_user_input(value)
except ValueError:
    logging.info("Invalid user input: %s", value)
    raise
```

It enables other patterns too.

Creating directories

`os.makedirs()` creates a directory.

```
# Creates the directory "riddles", relative
# to the current directory.
import os
os.makedirs("some-directory")
```

But if the directory already exists, it raises `FileExistsError`.

```
>>> os.makedirs("some-directory")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/lib/python3.6/os.py", line 220, in makedirs
    mkdir(name, mode)
FileExistsError: [Errno 17] File exists: 'some-directory'
```

Using In Code

Suppose if that happens, we want to log it, but then continue:

```
# First version....
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError:
        logging.error(
            "Upload dir for new user already exists")
```

This works, but the log message is not informative:

```
ERROR: Upload dir for new user already exists
```

Logging The directory

`FileExistsError` objects have an attribute called `filename`. Let's use that to create a useful log message:

```
# Better version!
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

MUCH better:

```
ERROR: Upload dir already exists: /var/www/uploads/joe
```

OSError

`FileExistsError` is only in Python 3. In Python 2, `os.makedirs()` instead raises `OSError`.

But `OSError` can indicate many other problems:

- filesystem permissions
- a system call getting interrupted
- a timeout over a network-mounted filesystem
- And the directory already existing.. the only one we care about.

How do you distinguish between these?

errno

`OSError` objects set an `errno` attribute. It's essentially the `errno` variable from C.

The standard constant for "file already exists" is `EEXIST`:

```
from errno import EEXIST
```

Game plan:

- Optimistically create the directory.
- if `OSError` is raised, catch it.
- Inspect the exception's `errno` attribute. If it's equal to `EEXIST`, this means the directory already existed; log that event.
- If `errno` is something else, it means we don't want to catch this exception here; re-raise the error.

create_upload_dir() in 2.x

```
# How to accomplish the same in Python 2.
import os
import logging
from errno import EEXIST
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except OSError as err:
        if err.errno != EEXIST:
            raise
        logging.error("Upload dir already exists: %s",
            err.filename)
```

The Most Diabolical Python Anti-Pattern

Design Patterns are good.

Anti-Patterns are bad.

And in Python, one Anti-Pattern most harmful of all.

I wish I could not even tell you about it. But I must.

TMDPAP

Here's the most self-destructive code a Python developer can write:

```
try:
    do_something()
except:
    pass
```

This creates the worst kind of bug.

After a FULL WEEK

After a full WEEK of engineer time, I was able to isolate the bug to a single block of code:

```
try:
    extract_address(location_data)
except:
    pass
```

Why???

Why do people do this?

1) Because they expect an exception to occur that can be safely ignored.

That's fine; the problem is being overbroad. Just target narrowly instead:

```
try:
    extract_address(location_data)
except ValueError:
    pass

# Variation: Insert logging.
try:
    extract_address(location_data)
except ValueError:
    logging.info(
        "Invalid location for user %s", username)
```

Why?

2) Because a code path must continue running regardless of what exceptions are raised.

In that case, this is better:

```
import logging
def get_number():
    return int('foo')
try:
    x = get_number()
except:
    logging.exception('Caught an error')
```

logging.exception()

Example stack trace:

```
ERROR:root:Caught an error
Traceback (most recent call last):
  File "example-logging-exception.py", line 5, in <module>
    x = get_number()
  File "example-logging-exception.py", line 3, in get_number
    return int('foo')
ValueError: invalid literal for int() with base 10: 'foo'
```

Command Line Programs

A Key To Automation

Much of the power of software comes from its automation.

A good command line interface makes the software you write more automatable. This ALWAYS increases the value of your program.

```
find . -name __pycache__ -type d
ls -l
git commit -m "Fix for #742"
grep -i 'python' *.txt
grep '^WARNING:' logs/*.log
```

sys.argv

One approach: use `sys.argv`.

```
# findpattern.py
import sys
def grepfile(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')

if __name__ == "__main__":
    pattern, path = sys.argv[1], sys.argv[2]
    for line in grepfile(pattern, path):
        print(line)
```

```
$ python3 findpattern.py  'WARNING' webserver.log
```

This works. But it makes you do the heavy lifting.

sys.argv

Also, the error messages are weird:

```
$ python3 findpattern.py
Traceback (most recent call last):
  File "findpattern.py", line 10, in <module>
    pattern, path = sys.argv[1], sys.argv[2]
IndexError: list index out of range
```

Optional Arguments

If you have multiple optional arguments, it gets really complex:

- `-i` for case-insensitive search
- `-c` for line count (instead of printing matching lines)
- Detect their presence in any order

```
$ python3 findpattern.py -i 'WARNING' webserver.log -c
```

The complexity explodes. We need a better approach.

argparse

The modern tool for building a command line interface is `argparse`.

- Easily specify complex sets of required and optional arguments
- Uses declarative syntax. Python handles the logic for you
- Boolean flags and other type conversions
- Automatic, user-friendly online help
- Clear and understandable error messages

```
import argparse
parser = argparse.ArgumentParser()
```

Using argparse

```
# findpattern.py, version 2:
# Use the same grepfile() function.

import argparse # instead of "import sys"

parser = argparse.ArgumentParser()
parser.add_argument('pattern')
parser.add_argument('path')

if __name__ == "__main__":
    args = parser.parse_args()
    for line in grepfile(args.pattern, args.path):
        print(line)
```

Parsing Args

`parse_args()` returns the parsed arguments:

```
>>> from findpattern import parser
>>> args = parser.parse_args(["WARNING", "webserver.log"])
>>> args
Namespace(path='webserver.log', pattern='WARNING')
>>> args.pattern
'WARNING'
>>> args.path
'webserver.log'
```

`parse_args()` operates on `sys.argv` by default:

```
# These two are equivalent.
parser.parse_args()
parser.parse_args(sys.argv[1:])
```

Helpful Errors

`argparse` includes automatic, user-friendly validation:

```
$ python3 findpattern.py
usage: findpattern.py [-h] pattern path
findpattern.py: error: the following arguments are required: pattern, path
```

And look, there's a -h option!

```
$ python3 findpattern.py  -h
usage: findpattern.py [-h] pattern path

positional arguments:
  pattern
  path

optional arguments:
  -h, --help  show this help message and exit
```

More Help

```
parser = argparse.ArgumentParser(description='Find patterns in file')
parser.add_argument('pattern', help='Substring pattern')
parser.add_argument('path', help='File to search in')
```

```
$ python3 findpattern.py -h
usage: findpattern.py [-h] pattern path

Find patterns in file

positional arguments:
  pattern     Substring pattern
  path        File to search in

optional arguments:
  -h, --help  show this help message and exit

Similar to grep, but with substring matching instead of regexes.
```

Boolean Flags

```
parser.add_argument('pattern', help='Substring pattern')
parser.add_argument('path', help='File to search in')
parser.add_argument('-i', '--ignore-case', default=False,
action='store_true')
```

```
>>> args = parser.parse_args(["-i", "WARNING", "webserver.log"])
>>> args.ignore_case
True

>>> args = parser.parse_args(["--ignore-case", "WARNING", "webserver.log"])
>>> args.ignore_case
True

>>> args = parser.parse_args(["WARNING", "webserver.log"])
>>> args.ignore_case
False
```

You can also do `default=True` and `action="store_false"`.

Typed Arguments

By default, arguments are parsed as strings. But you can specify a type.

```
parser.add_argument('--limit', default=None, type=int,
    help='Show only this many matches. Default is show all')
```

```
>>> args = parser.parse_args(["--limit", "42", "WARNING", "webserver.log"])
>>> print(args.limit)
42
>>> type(args.limit)
<class 'int'>

>>> args = parser.parse_args(["WARNING", "webserver.log"])
>>> print(args.limit)
None
```

```
$ python3 findpattern.py --limit notanumber WARNING webserver.log
usage: findpattern.py [-h] [-i] [--limit LIMIT] pattern path
findpattern.py: error: argument --limit: invalid int value: 'notanumber'
```

Custom Validation

```
parser = argparse.ArgumentParser()
parser.add_argument('-c', '--count', default=False, action='store_true',
                    help='Show # of matches instead of matching lines')
parser.add_argument('--limit', default=None, type=int,
                    help='Show only this many matches. Default is show all')
args = parser.parse_args()
if args.count and args.limit:
    parser.error('Cannot combine --limit and --count options')
```

```
$ python3 findpattern.py --limit 7 -c WARNING webserver.log
usage: findpattern.py [-h] [-i] [-c] [--limit LIMIT] pattern path
findpattern.py: error: Cannot combine --limit and --count options
```

Encapsulating

If your parser gets a little complex, encapsulate in a function.

```
import argparse
def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('pattern', help='Substring pattern')
    parser.add_argument('path', help='File to search in')
    parser.add_argument('-i', '--ignore-case', default=False,
                        action='store_true')
    parser.add_argument('-c', '--count', default=False, action='store_true',
                        help='Show # of matches instead of matching lines')
    parser.add_argument('--limit', default=None, type=int,
        help='Show only this many matches. Default is show all')
    args = parser.parse_args()
    if args.count and args.limit:
        parser.error('Cannot combine --limit and --count options')
    return args

if __name__ == "__main__":
    args = get_args()
```

Better UX

```
if __name__ == "__main__":
    args = get_args()
```

For a better user experience, parse the args immediately inside the main block.

Quicker feedback from `--help`, error checking, etc.

Lab: Command Line Arguments

Lab file: `commandline/argparselab.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up.

Reference:

https://docs.python.org/3/library/argparse.html

(Link in top of your lab file, for easy copy-pasting)

Test-Driven Development

Automated tests

An *automated test* is a program that tests another program.

Some of you have some experience with this already.

There are different flavors: unit tests, integration tests, etc.

We'll focus on the common basis, looking mainly at the simpler unit tests.

Why Write Tests?

It's basically a superpower.

Writing automated tests is one of the keys that separate average developers from world-class software engineers.

The ceiling of software complexity you can gracefully handle is *several quantum leaps higher* once you master unit tests.

This is well worth your while.

A Simple Test

Let's write an automated test for this function:

```
# Split a number into portions, as evenly as possible. (But it has a bug.)
def split_amount(amount, n):
    portion, remain = amount // n, amount % n
    portions = []
    for i in range(n):
        portions.append(portion)
        if remain > 1:
            portions[-1] += 1
            remain -= 1
    return portions
```

How it ought to work:

```
>>> split_amount(4, 2)
[2, 2]
>>> split_amount(5, 3)
[2, 2, 1]
```

The Test Function

Here's a function that will test it:

```
def test_split_amount():
    assert [1] == split_amount(1, 1)
    assert [2, 2] == split_amount(4, 2)
    assert [2, 2, 1] == split_amount(5, 3)
    assert [3, 3, 2, 2, 2] == split_amount(12, 5)
    print("All tests pass!")
# And of course, invoke it.
test_split_amount()
```

If any assertions fail, you'll see a stack trace:

```
Traceback (most recent call last):
  File "demo1.py", line 22, in <module>
    test_split_amount()
  File "demo1.py", line 18, in test_split_amount
    assert [2, 2, 1] == split_amount(5, 3)
AssertionError
```

Detecting the Error

The assertion that failed is:

```
assert [2, 2, 1] == split_amount(5, 3)
```

The good: Tells you an input that breaks the function.

The bad: Doesn't tell you anything else.

- What was the incorrect output?
- What other tests fail? The testing stops immediately, even if there are other assertions.
- Your large applications will have MANY tests. How do you reliably make sure you're running them all?
- Can we improve on the *reporting* of the test results?
- What about different assertion types?

Python's `unittest` module solves all these problems.

import unittest

Here's a basic unit test.

```
# test_splitting.py

from unittest import TestCase
from splitting import split_amount

class TestSplitting(TestCase):
    def test_split_amount(self):
        self.assertEqual([1], split_amount(1, 1))
        self.assertEqual([2, 2], split_amount(4, 2))
        self.assertEqual([2, 2, 1], split_amount(5, 3))
        self.assertEqual([3, 3, 2, 2, 2], split_amount(12, 5))
```

Running The Test

```
$ python3 -m unittest test_splitting.py
F
======================================================================
FAIL: test_split_amount (test_splitting.TestSplitting)
----------------------------------------
Traceback (most recent call last):
  File "test_splitting.py", line 8, in test_split_amount
    self.assertEqual([2, 2, 1], split_amount(5, 3))
AssertionError: Lists differ: [2, 2, 1] != [2, 1, 1]
First differing element 1:
2
1

- [2, 2, 1]
?     ^
+ [2, 1, 1]
?     ^

----------------------------------------
Ran 1 test in 0.001s
FAILED (failures=1)
```

Corrected Function

```
def split_amount(amount, n):
    'Split an integer amount into portions, as even as possible.'
    portion, remain = amount // n, amount % n
    portions = []
    for i in range(n):
        portions.append(portion)
        if remain > 0: # Was "remain > 1"
            portions[-1] += 1
            remain -= 1
    return portions
```

```
$ python3 -m unittest test_splitting.py
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

What's happening?

```
python3 -m unittest test_splitting.py
```

`unittest` is a standard library module. `test_splitting.py` is the file containing tests.

Inside is a class called `TestSplitting`. It subclasses `TestCase`.

(The name doesn't have to start with "Test", but often will.)

It has a method named `test_split_amount()`. That *test method* contains assertions.

Test methods **must** start with the string "test", or they won't get run.

Test Modules

To run code in a specific file:

```
python3 -m unittest test_splitting.py
```

OR a module name:

```
python3 -m unittest test_splitting
```

`test_splitting` is a **module**. It can be implemented as one or many files, just like any module.

In Python 2, you **must** pass the module argument, NOT the filename.

Test Discovery

You can also just run:

```
python3 -m unittest
```

This will locate all test code under the current directory.

This is called **test discovery**.

Restriction: the module/filename **must** start with "test" to be discovered.

To see options, run with `-h`:

```
python3 -m unittest -h
```

Assertions

`TestSplitting` uses the `assertEqual` method.

```
class TestSplitting(TestCase):
    def test_split_amount(self):
        self.assertEqual([1], split_amount(1, 1))
        self.assertEqual([2, 2], split_amount(4, 2))
        self.assertEqual([2, 2, 1], split_amount(5, 3))
        self.assertEqual([3, 3, 2, 2, 2], split_amount(12, 5))
```

Notice the expected value is always first. Consistency.

You can also make it always second. Just don't alternate in the same codebase.

Other Assertions

There are many different assertion methods. You'll most often use `assertEqual`, `assertNotEqual`, `assertTrue`, and `assertFalse`.

```
class TestDemo(TestCase):
    def test_assertion_types(self):
        self.assertEqual(2, 1 + 1)
        self.assertNotEqual(5, 1 + 1)
        self.assertTrue(10 > 1)
        self.assertFalse(10 < 1)
```

Full list:

https://docs.python.org/3/library/unittest.html#test-cases

Test Methods And Assertions

A single test method will stop at the first failing assertion.

Group related assertions in one test method, and separate other groups into new methods.

```
class TestSplitting(TestCase):
    def test_split_evenly(self):
        '''split_evenly() splits an integer into the smallest
           number of even groups.'''
        self.assertEqual([2, 2], split_evenly(4))
        self.assertEqual([5], split_evenly(5))
        self.assertEqual([6, 6], split_evenly(12))
        self.assertEqual([5, 5, 5], split_evenly(15))
    def test_split_amount(self):
        self.assertEqual([1], split_amount(1, 1))
        self.assertEqual([2, 2], split_amount(4, 2))
        self.assertEqual([2, 2, 1], split_amount(5, 3))
        self.assertEqual([3, 3, 2, 2, 2], split_amount(12, 5))
```

Test Methods and Failures

```
FF
======================================================================
FAIL: test_split_amount (test_splitting.TestSplitting)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_splitting.py", line 12, in test_split_amount
    self.assertEqual([1], split_amount(1, 1))
AssertionError: Lists differ: [1] != []

======================================================================
FAIL: test_split_evenly (test_splitting.TestSplitting)
split_evenly() splits an integer into the smallest # of even groups.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_splitting.py", line 7, in test_split_evenly
    self.assertEqual([2, 2], split_evenly(4))
AssertionError: Lists differ: [2, 2] != []

----------------------------------------------------------------------
Ran 2 tests in 0.001s
```

TDD

The idea of **Test-Driven Development**.

1. Write the test.
2. Run it, and watch it fail.
3. THEN write code to make the test pass.

This has some surprising benefits:

- Code clarity
- State of Flow
- Generally more robust software

And some downsides.

To TDD or Not?

People get religious about this. Be gentle with the zealots.

If you're new to writing tests, strictly following TDD for a while is a great way to get very good, very quickly. And remember, writing good tests is a critical skill.

Once you're fairly good at it: Consider following the 80-20 rule.

Lab: Unit Tests

In this self-directed lab, you implement a small library called `textlib`, and a test module named `test_textlib`.

Instructions: `testing/lab.txt`

- In labs/py3 for 3.x; labs/py2 for 2.7
- First follow the instructions to write `textlib.py` and `test_textlib.py`
- When you are done, give a thumbs up...
- ... then follow the extra credit instructions

Remember, in Python 2, you MUST omit the `.py`:

```
python2.7 -m unittest test_textlib
```

In Python 3, you can pass the file name or the module name:

```
python3 -m unittest test_textlib.py
```

Fixtures

As you write more tests, you'll create test case classes whose methods start or end with the same lines of code.

Consolidate these in `setUp()` and `tearDown()`.

```
class TestSample(unittest.TestCase):
    def setUp(self):
        "Run before every test methods starts."
        self.conn = connect_to_test_database()
    def tearDown(self):
        "Run after every test method ends."
        self.conn.close()
    def test_create_tables(self):
        from mylib import create_tables
        create_tables(self.conn)
```

Watch Out: It's "setUp", not "setup". "tearDown", not "teardown".

Example

Imagine writing a program that saves it state between runs. It saves it to a special file, called the "state file".

```
# statefile.py
class State:
    def __init__(self, state_file_path):
        # Load the stored state data, and save
        # it in self.data.
        self.data = { }
    def close(self):
        # Handle any changes on application exit.
```

Planning The Test

Tests on the `State` class should verify:

- If you add a new key-value pair to the state, it is recorded correctly in the state file.
- If you alter the value of an existing key, that updated value is written to the state file.
- If the state is not changed, the state file's content stays the same.

test_statefile: initial code

```
# test_statefile.py
import os
import unittest
import shutil
import tempfile
from statefile import State

INITIAL_STATE = '{"foo": 42, "bar": 17}'
```

test_statefile: setUp and tearDown

```
class TestState(unittest.TestCase):
    def setUp(self):
        self.testdir = tempfile.mkdtemp()
        self.state_file_path = os.path.join(
            self.testdir, 'statefile.json')
        with open(self.state_file_path, 'w') as outfile:
            outfile.write(INITIAL_STATE)
        self.state = State(self.state_file_path)

    def tearDown(self):
        shutil.rmtree(self.testdir)
```

test_statefile: the tests

```
    def test_change_value(self):
        self.state.data["foo"] = 21
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertEqual(21,
            reloaded_statefile.data["foo"])

    def test_remove_value(self):
        del self.state.data["bar"]
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertNotIn("bar", reloaded_statefile.data)

    def test_no_change(self):
        self.state.close()
        with open(self.state_file_path) as handle:
            checked_content = handle.read()
        self.assertEqual(checked_content, INITIAL_STATE)
```

Expecting Exceptions

Sometimes your code is *supposed* to raise an exception. And it's an error if, in that situation, it does not.

Use `TestCase.assertRaises()` to verify.

Imagine a `roman2int()` function:

```
>>> roman2int("XVI")
16
>>> roman2int("II")
2
>>> roman2int("a thousand")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in roman2int
ValueError: Not a roman numeral: a thousand
```

Asserting Exceptions

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("This is not a valid roman numeral.")
```

Catching The Error

If `roman2int()` does NOT raise `ValueError`:

```
$ python3 -m unittest test_roman2int.py
F
======================================================================
FAIL: test_roman2int_error (test_roman2int.TestRoman)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_roman2int.py", line 7, in test_roman2int_error
    roman2int("This is not a valid roman numeral.")
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Subtests

Python 3 only. And really valuable.

Imagine a function `numwords()`, counting unique words:

```
>>> numwords("Good, good morning. Beautiful morning!")
3
```

Testing numwords()

```
class TestWords(unittest.TestCase):
    def test_whitespace(self):
        self.assertEqual(2, numwords("foo bar"))
        self.assertEqual(2, numwords("    foo bar"))
        self.assertEqual(2, numwords("foo\tbar"))
        self.assertEqual(2, numwords("foo   bar"))
        self.assertEqual(2, numwords("foo bar    \t   \t"))
        # And so on, and so on...
```

This has two problems.

Less repetition...

```
    def test_whitespace_forloop(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            self.assertEqual(2, numwords(text))
```

More maintainble. But...

... but problematic

That approach creates more problems than it solves.

```
$ python3 -m unittest test_words_forloop.py
F
======================================================
FAIL: test_whitespace_forloop (test_words_forloop.TestWords)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_forloop.py", line 17, in test_whitespace_forloop
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Pop quiz: what exactly went wrong?

A Better Way

We need something that is (a) maintainable, and (b) clear in the error reporting.

Python 3.4 solves this with **subtests**.

```
    def test_whitespace_subtest(self):
        texts = [
        "foo bar",
        "    foo bar",
        "foo\tbar",
        "foo   bar",
        "foo bar    \t   \t",
        ]
    for text in texts:
        with self.subTest(text=text):
            self.assertEqual(2, numwords(text))
```

self.subTest()

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

`self.subTest()` creates a context for assertions.

Even if that assertion fails, the test continues through the for loop.

ALL failures are collected and reported at the end, with clear information identifying the exact problem.

Subtest Reporting

```
$ python3 -m unittest test_words_subtest.py
======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo\tbar')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo bar    \t   \t')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 4

------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures=2)
```

Subtest Reporting

Behold the opulence of information in this output:

- Each individual failing input has its own detailed summary.
- We are told what the full value of `text` was.
- We are told what the actual returned value was, clearly compared to the expected value.
- No values are skipped. We can be confident that these two are the *only* failures.

In detail...

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

The key-value pairs to `subTest()` are used in reporting the output. They can be anything you like.

Pay attention: the symbol `text` has *two different meanings* on these lines.

- The argument to `numwords()`
- A field in the failure report

Reporting Fields

Suppose you wrote:

```
for text in texts:
    with self.subTest(input_text=text):
        self.assertEqual(2, numwords(text))
```

Then the failure output might look like:

```
FAIL: test_whitespace_subtest (test_words_subtest.TestWords)
(input_text='foo\tbar')
```

Lab: Intermediate Unit Tests

Instructions: `testing/lab-intermediate.txt`

- In labs/py3 for 3.x; labs/py2 for 2.7
- First follow the instructions to write `textlib.py` and `test_textlib.py`
- When you are done, give a thumbs up.

Next, you can:

- Follow the extra credit instructions in `lab-intermediate.txt`.
- OR, if you're using Python 3, you can do `lab-subtests.txt`.

If you are new to `unittest`, focus on finishing `lab.txt` first.

Alternatives

`unittest` isn't the only game in town.

- doctest
  - Also in standard library
  - Most of your labs use this!
  - But only suitable for simpler code.

- pytest
  - Currently Python's most popular 3rd-party testing tool
  - Arguably better than unittest. But adds a separate dependency, and not universally used

- nose and nose2
  - Largely inactive now. Sometimes you'll still see it, though, especially with older projects.

Python Collections

import collections

The `collections` module provides some great tools.

- namedtuple
- defaultdict
- OrderedDict
- dequeue
- Counter
- ChainMap
- UserDict, UserList, and UserString

namedtuple

A great boon for readability. Use it!

```
>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name gpa major')
>>>
>>> student = Student('Joe Smith', 3.7, 'Computer Science')
>>> student.name
'Joe Smith'
>>> student.gpa
3.7
>>> student.major
'Computer Science'
```

It's a tuple

```
>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name gpa major')
>>> student = Student('Joe Smith', 3.7, 'Computer Science')
>>> # like student.name
... student[0]
'Joe Smith'
>>> # like student.gpa
... student[1]
3.7
>>> # like student.major
... student[2]
'Computer Science'
```

It's a shortcut

```
# Compare to:
# Student = namedtuple('Student', 'name gpa major')
class Student:
    def __init__(self, name, gpa, major):
        self.name = name
        self.gpa = gpa
        self.major = major

student = Student('Joe Smith', 3.7, 'Computer Science')
```

It's Immutable

Just like a regular tuple.

Think of `nametuple` as just like a normal tuple, except you have the convenient option to refer to fields by name.

```
>>> Student = namedtuple('Student', 'name gpa major')
>>> student = Student('Joe Smith', 3.7, 'Computer Science')
>>> student.name
'Joe Smith'
>>> student.name = 'John Doe'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

2 Ways To Create It

```
>>> from collections import namedtuple
>>> # Pass in a string...
... Student1 = namedtuple('Student1', 'name gpa major')
>>>
>>> # ... or a list of strings.
... fields = ['name', 'gpa', 'major']
>>> Student2 = namedtuple('Student2', fields)
>>>
>>> # They work the same. Some prefer one or the other.
... student1 = Student1('Joe Smith', 3.7, 'Computer Science')
>>> student1.name
'Joe Smith'
>>> student2 = Student2('Joe Smith', 3.7, 'Computer Science')
>>> student2.name
'Joe Smith'
```

First Argument is Repetitive

The first argument doesn't HAVE to be the same as what you assign it to. But there is almost never a reason to make them different.

```
>>> # Confusing...
... Foo = namedtuple('Bar', 'x y z')
>>> baz = Foo(1,2,3)
>>> type(baz)
<class '__main__.Bar'>
```

Where Do You Use It?

Anywhere you use tuples, and keeping track of indexes is slightly confusing.

```
>>> t = get_book_info_tuple('978-0439064873')
>>> # author
... t[0]
'J.K. Rowling'
>>> # title
... t[1]
'Harry Potter and the Chamber of Secrets'
>>> # pub date
... t[2]
2000
>>> # page count
... t[3]
341
>>> # rating
... t[4]
4.7
```

Clearer APIs

Note also: `namedtuple` is backwards-compatible with regular tuples.

```
BookInfo = namedtuple('BookInfo', [
  'author',
  'title',
  'pub_date',
  'page_count',
  'rating',
])
```

```
>>> info = get_book_info_namedtuple('978-0439064873')
>>> info.title
'Harry Potter and the Chamber of Secrets'
>>> info[1]
'Harry Potter and the Chamber of Secrets'
>>> info.author
'J.K. Rowling'
>>> info.page_count
341
```

Convenience

Sometimes it's even more useful with simpler types. Notice the clarity here.

```
>>> # latitude & longitude
... Location = namedtuple('Location', 'latitude longitude')
>>> place = Location(37.77, -122.46)
>>> place.latitude
37.77
>>> place.longitude
-122.46
>>>
>>> # Compare to:
... place = (37.77, -122.46)
>>> place[0]
37.77
>>> place[1]
-122.46
```

defaultdict

Word counting problem:

```
>>> words = 'beauty is truth truth beauty'.split()
>>> counts = {}
>>> for word in words:
...     if word in counts:
...         counts[word] += 1
...     else:
...         counts[word] = 1
...
>>> print(counts)
{'is': 1, 'beauty': 2, 'truth': 2}
>>>
>>>
```

Notice the "if".

Code Compress

We can compress 4 lines to 1 with `defaultdict`.

```
>>> from collections import defaultdict
>>> counts = defaultdict(int)
>>> for word in words:
...     counts[word] += 1
...
>>> print(counts)
defaultdict(<class 'int'>, {'is': 1, 'beauty': 2, 'truth': 2})
```

Different Default Behavior

`defaultdict` does not raise a `KeyError`, unlike regular dictionaries.

```
>>> regular = dict()
>>> regular['apple']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'apple'
>>>
>>> from collections import defaultdict
>>> default = defaultdict(int)
>>> default['apple']
0
```

Of course, you can insert just like a regular dictionary.

```
>>> default['orange'] = 7
>>> default['orange']
7
```

The new-item-generator

The argument to `defaultdict` is a *callable*:

```
>>> my_ints = defaultdict(int)
>>> my_floats = defaultdict(float)
>>> my_strings = defaultdict(str)
>>>
>>> my_ints['foo']
0
>>> my_floats['foo']
0.0
>>> my_strings['foo']
''
```

That callable is automagically called, with no arguments, to insert the default value.

Question: How would you create a `defaultdict` that defaults to 1 (instead of 0)?

Custom Function

```
>>> def one():
...     return 1
...
>>> my_numbers = defaultdict(one)
>>>
>>> my_numbers['foo']
1
>>> my_numbers['bar'] += 1
>>> my_numbers['bar']
2
>>> my_numbers['baz'] = 5
>>> my_numbers['baz']
5
```

Example

```
>>> grams_of_metal = defaultdict(float)
>>> # Found an iron nugget
... grams_of_metal['iron'] += 97.3
>>> # Oooh! A silver piece!
... grams_of_metal['silver'] += 10.1
>>> # Another iron nugget
... grams_of_metal['iron'] += 72.8
>>> grams_of_metal['iron']
170.1
>>> grams_of_metal['silver']
10.1
>>> grams_of_metal['gold']
0.0
```

Another example

```
>>> foods = defaultdict(str)
>>> foods['Poland'] += 'Zupa,'
>>> foods['Australia'] += 'Macadamia nuts,'
>>> foods['USA'] += 'Burger,'
>>> foods['Australia'] += 'Meat pies,'
>>> foods['Poland'] += 'Pierogi,'
>>> for country, food in foods.items(): print(country + ": " + food)
...
USA: Burger,
Australia: Macadamia nuts,Meat pies,
Poland: Zupa,Pierogi,
```

Actually, for this data, what we really want is not a string, but a list or set. Let's do that!

Default collections

```
>>> foods = defaultdict(list)
>>> foods['Poland'].append('Zupa')
>>> foods['Australia'].append('Macadamia nuts')
>>> foods['USA'].append('Burger')
>>> foods['Australia'].append('Meat pies')
>>> foods['Poland'].append('Pierogi')
>>> for country, food in foods.items():
...     print(country + ": " + ", ".join(food))
...
USA: Burger
Australia: Macadamia nuts, Meat pies
Poland: Zupa, Pierogi
```

We also could have used `defaultdict(set)`.

Ordered Dictionary

Dictionaries in Python are unordered.

```
>>> # Atomic numbers of noble gasses.
... nobles = {'He': 2, 'Ne': 10,
...    'Ar': 18, 'Kr': 36, 'Xe': 54}
>>> # This effectively prints them in random order.
... for atom, number in nobles.items():
...     print('{} {}'.format(atom, number))
...
Ne 10
Kr 36
Xe 54
Ar 18
He 2
```

The OrderedDict class

```
>>> from collections import OrderedDict
>>> nobles = OrderedDict()
>>> nobles['He'] =  2
>>> nobles['Ne'] = 10
>>> nobles['Ar'] = 18
>>> nobles['Kr'] = 36
>>> nobles['Xe'] = 54
>>> # .keys(), .values() and .items() produce
... # data in the order of insertion.
... for atom, number in nobles.items():
...     print('{} {}'.format(atom, number))
...
He 2
Ne 10
Ar 18
Kr 36
Xe 54
```

Constructing OrderedDict's

You can also pass in a list (or any iterable) to the `OrderedDict` constructor.

```
>>> data = [
...    ('He', 2), ('Ne', 10), ('Ar', 18),
...    ('Kr', 36), ('Xe', 54)
...    ]
>>> nobles = OrderedDict(data)
>>>
>>> for atom, number in nobles.items():
...     print('{} {}'.format(atom, number))
...
He 2
Ne 10
Ar 18
Kr 36
Xe 54
```

OrderedDict is a dictionary

`OrderedDict` has all the same methods as regular dictionaries.

But the ordering adds some quirks.

```
>>> # Deleting and re-inserting changes the order.
... od = OrderedDict([('a', 7), ('b', 3), ('c', 11)])
>>> print(list(od.keys()))
['a', 'b', 'c']
>>> del od['b']
>>> od['b'] = 7
>>> print(list(od.keys()))
['a', 'c', 'b']
>>> # Updating does NOT change the order, though.
... od['a'] = 8
>>> print(list(od.keys()))
['a', 'c', 'b']
>>> # reversed() plays nice with OrderedDict.
... backwards = reversed(od.items())
>>> type(backwards)
<class 'odict_iterator'>
>>> print(list(backwards))
[('b', 7), ('c', 11), ('a', 8)]
```

Additional Methods

popitem(True)

Remove (key, value) pair, LIFO

popitem(False)

Remove (key, value) pair, FIFO

move_to_end(key, True)

Move existing key to end.

move_to_end(key, False)

Move existing key to beginning.

Beware!

`OrderedDict` lets you instantiate with keyword arguments. But it leaves the ordering completely undefined.

```
>>> bad = OrderedDict(a=2, b=3, c=7)
>>> list(bad.keys())
['b', 'a', 'c']
>>> # The .update() method works fine if you pass in an OrderedDict...
... new_ordered = OrderedDict([('d',9), ('e',6), ('f',32)])
>>> bad.update(new_ordered)
>>> list(bad.keys())
['b', 'a', 'c', 'd', 'e', 'f']
>>> # ... but doesn't define any ordering if you pass a regular dict.
... new_unordered = {'x': 14, 'y': 2, 'z': 23}
>>> bad.update(new_unordered)
>>> list(bad.keys())
['b', 'a', 'c', 'd', 'e', 'f', 'z', 'y', 'x']
```

More Collections

deque	Double-ended queue. Like list, with fast append/pop on each end
ChainMap (py3)	Single dict-like view of multiple mappings
Counter	for counting hashable objects
UserDict, UserList, UserString	Easier subclassing of built-ins

Details in the standard library:
https://docs.python.org/3/library/collections.html

Python 2 differences

- `ChainMap` only exists in Python 3.
- Python 2 puts `UserDict`, `UserList`, and `UserString` each in their own module.
- Some useful methods (like OrderedDict.move_to_end) only exist in Python 3.

Lab: Collections

Lab file: `collections/maincollections.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up.

Collections reference:
https://docs.python.org/3/library/collections