Coding Workshop

readr 2017

What you need

- Caffeine to stay awake
- Some knowledge of Python
- Maybe some knowledge of git

Examples will require git + PyCharm + python + Github account to follow along with. *Windows users will need to install git manually*.

I assume everyone will have python. Some windows users might need to install git. Please make sure everyone has a Github account, and to follow along please make sure everyone has PyCharm. If people don't have PyCharm, I recommend starting the download now so that it will be done by lunch. Plenty of time, no pressure.

Coding Practises

Or “Things that annoy me”

But my code ‘works’...

- Ever asked for help?
- Ever sent your code to someone else?
- Ever put code online? (Do it, it’s great fun)
- Ever taken a break from a project and come back to it?
- Or just want to code more efficiently?

Fundamentals

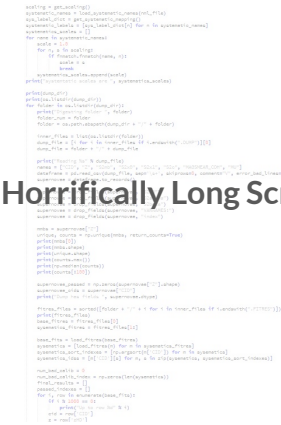
- Single Responsibility Principle
- Immutable vs mutable variables
- Decoupling of input/output
- Variable names that won't make you cry
- Minimising code duplication
- Code Linters

Single Responsibility Principle

*Every class, function and method should handle **one** thing, and **one** thing only.*

Single Responsibility Principle

Which is easier to read, understand, and update?



Horrifically Long Script

```
if __name__ == "__main__":
    filename_data = "supernovae.txt"
    filename_plot = "cosmology.png"

    supernovae_data = load_data(filename_data)
    chains = fit_cosmology(supernovae_data)
    plot_cosmology(chains, filename_plot)
```

Makes it WAY easier to navigate code and contribute to it. Another big benefit is that it allows you to document code without writing documentation. By writing code that follows the SRP, often simply the function or class name is sufficient to explain what the code is doing. The right hand code's function can be arbitrarily long, but you still know what is going on!

Finally, its also a good way to get coding when you aren't sure how to start. List the tasks or things you need done. Great, now theyre all functions. For each one, figure out the steps you need. And those will either be the lines of code, or more functions. At the end, with your code scaffolded out with empty functions, you can go back and fill in the blanks.

Immutability

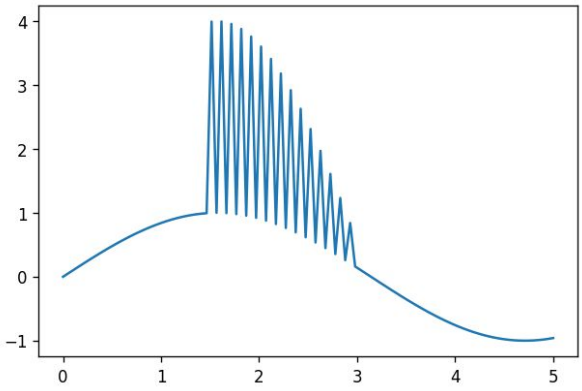
```
class Favourites(object):
    def __init__(self):
        self.set_default_favs()
    def get_favs(self):
        return self.favs
    def add_fav(self, fav):
        self.favs.append(fav)
    def set_default_favs(self, default=["pizza"]):
        self.favs = default

f = Favourites()
print(f.get_favs())                    → ['pizza']

f.add_fav("chocolate")
print(f.get_favs())                    → ['pizza', 'chocolate']

f.set_default_favs()
print(f.get_favs())                    → ['pizza', 'chocolate']

f2 = Favourites()
print(f2.get_favs())                   → ['pizza', 'chocolate']
```

A less contrived example, showing the effect in a given class instance, and the effect over separate instances

Immutability

Be wary when passing mutable objects by reference.

```
import numpy as np
import matplotlib.pyplot as plt

def get_mult_sum(data):
    data *= 4
    return data.sum()

xs = np.linspace(0, 5, 100)
ys = np.sin(xs)
print(get_mult_sum(ys[30:60:2]))
plt.plot(xs, ys)
```



Immutability

As primitives are immutable objects in python, don't have to worry about a lot of mutability issues such as you'd get in *c*.

Immutable objects:

- Int
- Float
- Bool
- String
- Tuple
- Range
- etc

Mutable objects:

- List
- Dict
- Set
- User defined classes
- bytearrays

So everything on the left is fine to go in a method signature, everything on the right should be avoided. And be careful when you expose mutable variables to other people / code, best to make a copy or trust that they dont change it!

Decoupling Input

Note: Extremely simplified.

```
import numpy as np

def get_data():
    return np.load("data.npy")

def crunch_data():
    data = get_data()
    return data.sum(axis=0)

def save_results(data):
    np.savetxt("output.txt", data)

def analyse():
    crunched = crunch_data()
    save_results(crunched)
```

In a large projects these functions might be found hundreds of lines away or in different files.

So the idea is that input and output should be decoupled from algorithmic logic as much as possible. In this example, imagine trying to test crunch_data on a different dataset - you cant do it without editting some random get_data function

Decoupling Input

Note: Extremely simplified.

```
import numpy as np

def get_data(file):
    return np.load(file)

def crunch_data(data):
    return data.sum(axis=0)

def save_results(data, file):
    np.savetxt(file, data)

def analyse(data_file, output_file):
    data = get_data(data_file)
    crunched = crunch_data(data)
    save_results(crunched)

analyse("test_data.npy", "test_output.txt")
```

Fully decoupled input and output.

Not always necessary - depends on what level your logic is.

So in the refactored example I've moved the input and output right to the top, but this isnt always necessary. If you have a function that uses data, but the rest of the application doesnt, just decouple that specific function. The goal is you should be able to quickly test whatever algorithm uses the data with a different dataset (such as a mock). Because this is such a simple example it goes right to the top.

The other advantage is you dont have to go hunting. Make the most likely to change code right at your fingertips

Other notes

1. **Don't** redefine variables
2. **Don't** access globals from inside functions / classes
3. **Don't** write global code and also define a main statement
4. **Always** use **`with`** statements when you can… ie files
5. **Separate** plotting and analysis

```
# Bad
f = open("file.txt", "w")
f.write("Hello")
f.close()

# Good
with open("file.txt", "w") as f:
    f.write("Hello")
```

Other things to quickly point out

REASONS TO USE AN IDE

Variable Names and Reserved Words

Resist the temptation to just press a random key.

If you do, ***refactor.*** Don't waste your precious time doing it yourself!

Don't just avoid keywords, avoid shadowing builtins! If you don't want a variable, use an underscore.

```
import builtins
dir(builtins)
```

```
... Exception, False, None, True,
Warning, abs, all, any, ascii, bin,
bool, bytearray, bytes, callable, chr,
compile, complex, dict, dir, enumerate,
eval, exec, float, format, getattr,
globals, hash, help, hex, id, input,
int, isinstance, issubclass, iter, len,
list, locals, map, max, min, next,
object, oct, open, ord, pow, print,
property, range, repr, reversed,
round, set, slice, sorted, str, sum,
super, tuple, type, vars, zip
```

Often bad variable names come through because Im not sure if something is going to work, so write up really quick, low effort code to check. And if it does, its easy to not go back and update things into legibility, and I end up with d, dd, d2, ddd all in one function.

This is why I recommend using an IDE like PyCharm. Got a bad variable name, just press Shift+F6 on the variable and rename it. The IDE will rename all usages of it for you, you dont need to go through you code and change them yourself, miss one, and thus break you code.

Mostly I just want to discourage people from overwriting the builtins. I have seen bugs introduced by this, even done it myself when I shadowed 'id' and then was doing some memory management stuff which invoked id.

Code Duplication

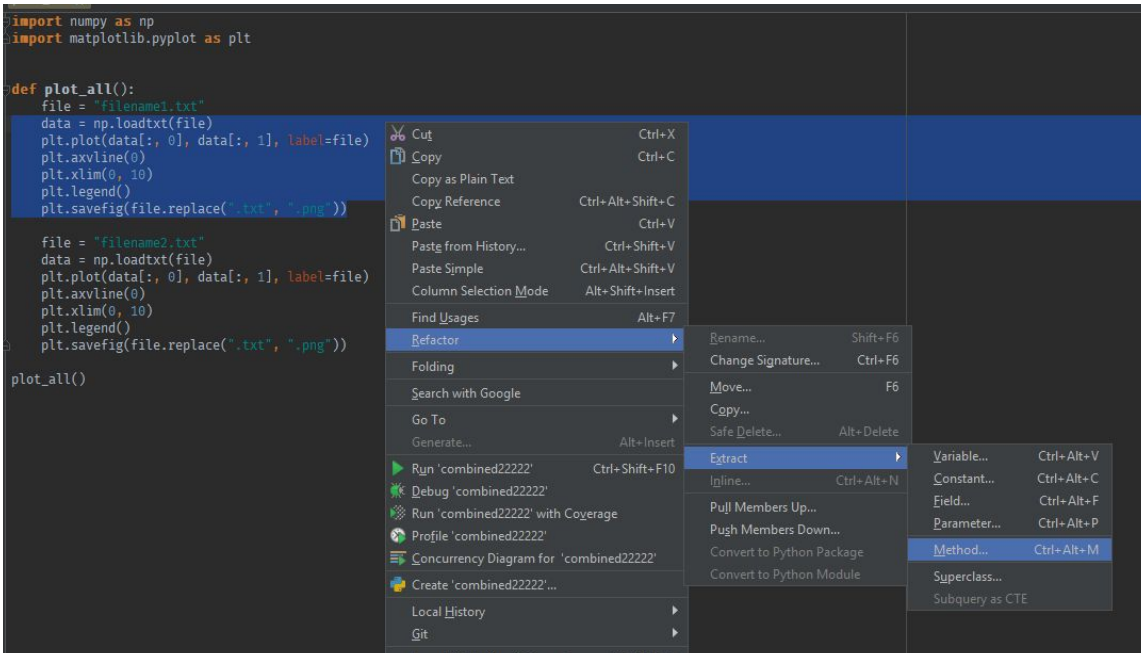
Don't know how many times I've seen this sort of code:

```
import numpy as np
import matplotlib.pyplot as plt

def plot_all():
    file = "filename1.txt"
    data = np.loadtxt(file)
    plt.plot(data[:, 0], data[:, 1], label=file)
    plt.axvline(0)
    plt.xlim(0, 10)
    plt.legend()
    plt.savefig(file.replace(".txt", ".png"))

    file = "filename2.txt"
    data = np.loadtxt(file)
    plt.plot(data[:, 0], data[:, 1], label=file)
    plt.axvline(0)
    plt.xlim(0, 10)
    plt.legend()
    plt.savefig(file.replace(".txt", ".png"))

plot_all()
```

Minimise the work you have to do to refactor. Use an IDE.

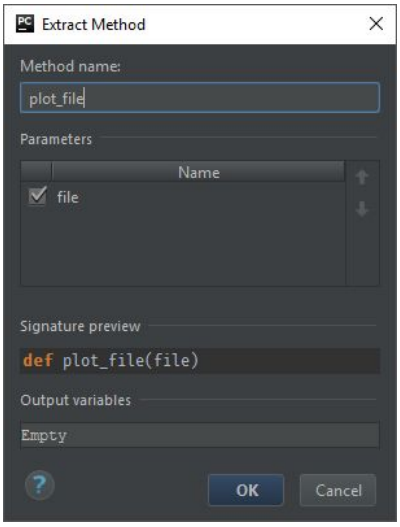
Another recommendation for an IDE. Code duplication occurs because its easier to copy paste and check something works rather than refactoring yourself. But after it works, its hard to motivate yourself to go through and tidy up.

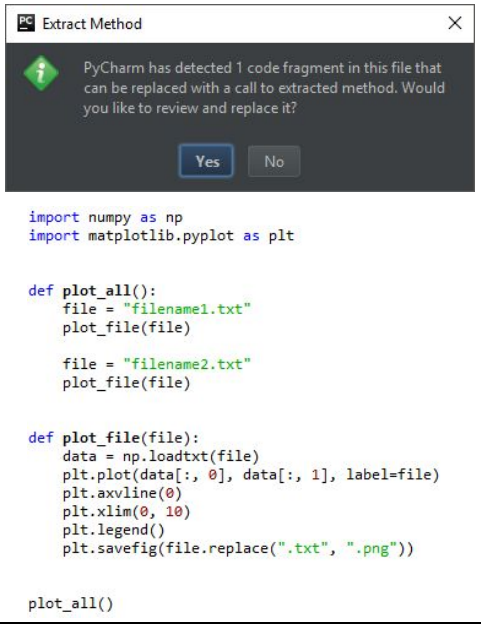
Modern IDEs have refactoring tools though that do it for you.



Another recommendation for an IDE. Code duplication occurs because its easier to copy paste and check something works rather than refactoring yourself. But after it works, its hard to motivate yourself to go through and tidy up.

Modern IDEs have refactoring tools though that do it for you.





Another recommendation for an IDE. Code duplication occurs because its easier to copy paste and check something works rather than refactoring yourself. But after it works, its hard to motivate yourself to go through and tidy up.

Modern IDEs have refactoring tools though that do it for you.

Code Linters

Code linters will analyse the code you write to:

1. Improve readability
2. Pre-emptively detect bugs
3. Find unused code

Can run these manually, run them in a build pipeline (covered later) or use them integrated into an IDE

```
import numpy as np
def get_floor_max(a,b,c):
    if a>=b:
        maxv=a
    else:
        maxv=b
    sum=np.floor(maxv)
    return sum
if __name__=="__main__":
    d=get_floor_max(1,2,3)#Sums stuff
    print (d)
```

Code Linters

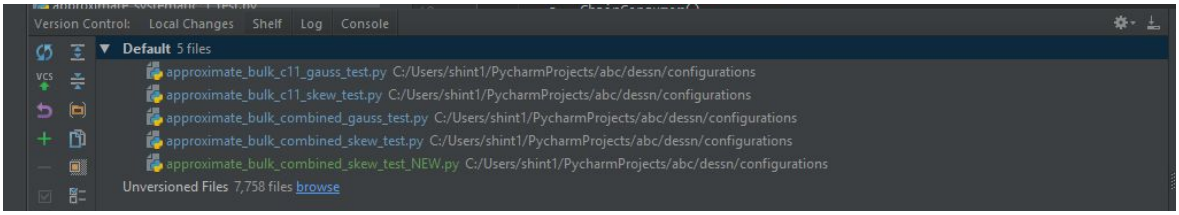
PEP8 standard enabled by default in PyCharm

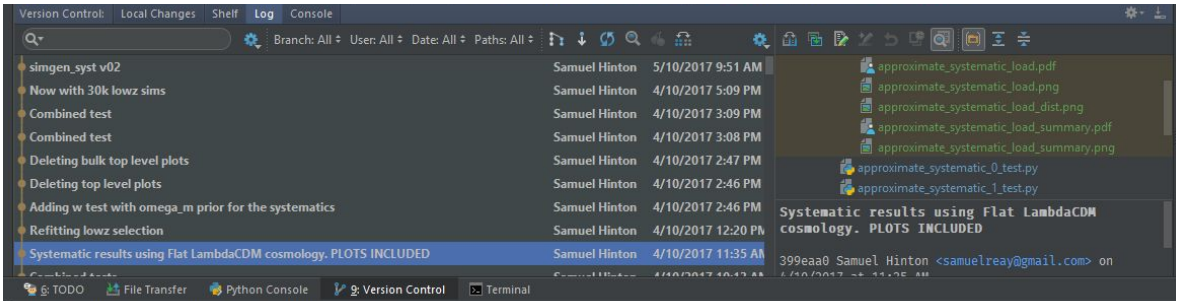
```
import numpy as np
def get_floor_max(a,b,c):
    if a>=b:
        maxv=a
    else:
        maxv=b
    sum=np.floor(maxv)
    return sum
if __name__=="__main__":
    d=get_floor_max(1,2,3)#Sums stuff
    print(d)
```

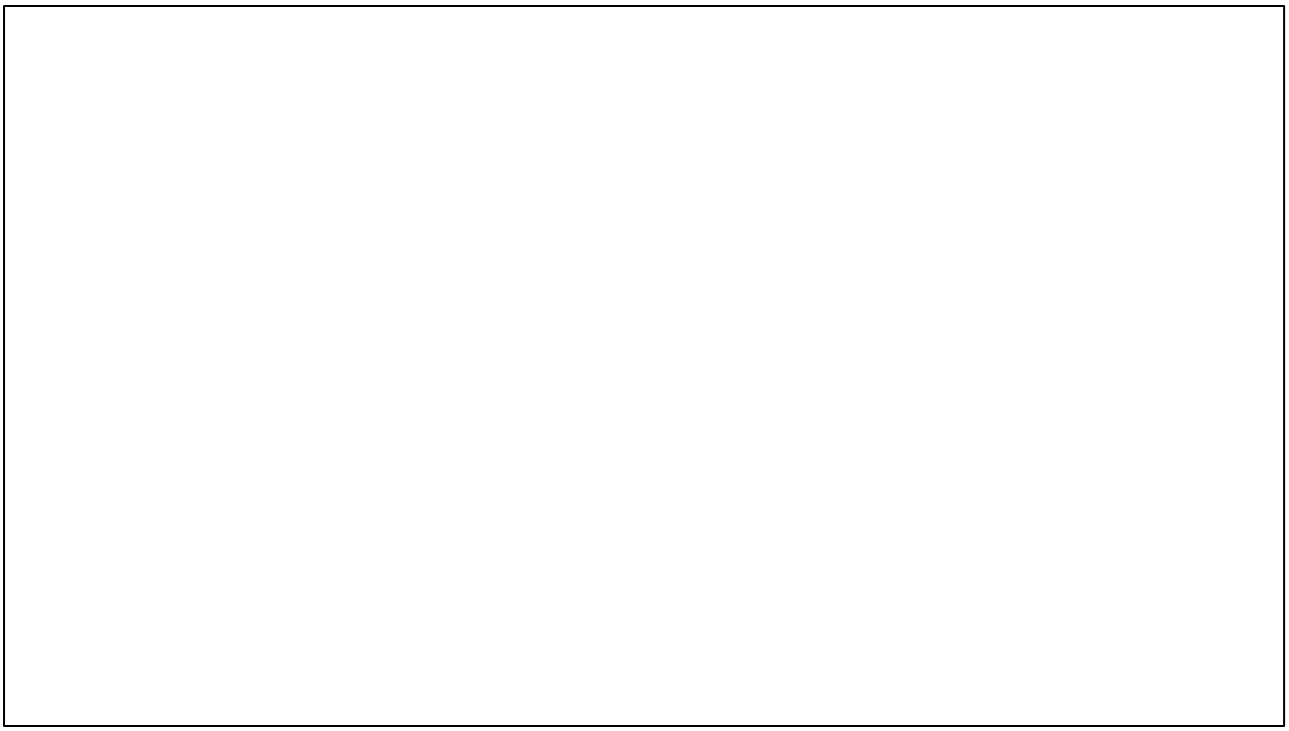
```
import numpy as np


def get_floor_max(a, b, c):
    if a >= b:
        maxv = a
    else:
        maxv = b
    sum = np.floor(maxv)
    return sum


if __name__ == "__main__":
    d = get_floor_max(1, 2, 3)  # Sums stuff
    print(d)
```

Can go to Code->Reformat file and have it format the code for you to take the hassle out of it. Note that some things like “c” and “sum” are still underlined. Can hover for why - c is not used, and sum shadows a builtin. IDEs really try to make your life simpler.

Git Integration







Q1: What does this code do?

Q2: How many ways can you improve this?

```
import numpy as np
from numpy import *
f = open("/home/shint/6cols.txt", "r")
for line in f:
    headers = line.split()
    id, ra, dec, z, mag, mask = headers
    break

d = np.loadtxt("/home/shint/6cols.txt", skiprows=1)
mask = d[:, headers.index(mask)] == 1
zs = d[mask,headers.index(z)]
mags = d[mask, headers.index(mag)]
b = np.linspace(zs.min(), zs.max(), 10)
b[0] -= 0.000001
bc = 0.5*(b[1:]+b[:-1])
sums = array([0.0]*9)
cs = [0]*9
for i in range(len(zs)):
    z, mag = zs[i], mags[i]
    index = (z>b).sum()-1
    sums[index] += mag
    cs[index] += 1
print np.array(sums) / np.array(cs)
```

I guinea pigged this activity twice at Brisbane. I normally give five minutes to figure out what the code is doing, ask a few groups what they think, and then give between 5 and 10 minutes to come up with some updates. The next slide has the ones I find most annoying, but there are actually a few I considered too pedantic to list. Works well to go through each group asking what they would change in order or severity until you have gone through everyone and then jump to the next slide.

What the code does is calculate the mean magnitude for each redshift bin.

Yay
Double import. Using * import
Filename is OS / user dependent

Assumes file structure and order
Shadows id, unused vars
Doesn't close file

Duplicate filename
Self shadowing, float/int comparison
Inconsistent whitespacing
Redundant code
Magic numbers (repeatedly)
Dirty hacks
Awful variable names
Unpythonic declarations
Mixing data types
"Traditional" for loop
Zip it
Needlessly obtuse

Amateur hour
No return object. And... PYTHON2

```
import numpy as np
from numpy import *
f = open("/home/shint/6cols.txt", "r")
for line in f:
    headers = line.split()
    id, ra, dec, z, mag, mask = headers
    break

d = np.loadtxt("/home/shint/6cols.txt", skiprows=1)
mask = d[:, headers.index(mask)] == 1
zs = d[mask,headers.index(z)]
mags = d[mask, headers.index(mag)]
b = np.linspace(zs.min(), zs.max(), 10)
b[0] -= 0.000001
bc = 0.5*(b[1:]+b[:-1])
sums = array([0.0]*9)
cs = [0]*9
for i in range(len(zs)):
    z, mag = zs[i], mags[i]
    index = (z>b).sum()-1
    sums[index] += mag
    cs[index] += 1
print np.array(sums) / np.array(cs)
```

Double import obviously bad
Star import is bad because you lose track of where things come from
OS and user dependent filenames are a cardinal sin

The next section, assumes file structure to "Doesnt close file" is useless. Because I hardcode the order of the variables in the header assignment, theres literally no point doing this.

Refactored and cleaned code.
Of course, just use `scipy.stats.binned_statistic`

```python
import numpy as np

def get_masked_data(filename):
    data = np.genfromtxt(filename, dtype=None, names=True)
    mask = data["mask"] == 1
    return data[mask]

def get_mean_per_bin(x, y, bins=9):
    bin_edges = np.linspace(x.min(), x.max(), bins + 1)
    # Map each entry to which x bin it belongs to
    mapping = np.digitize(x, bin_edges) - 1
    avgs = np.array([np.mean(y[mapping == i]) for i in range(bins)])
    return bin_edges, avgs

if __name__ == "__main__":
    data = get_masked_data("6cols.txt")
    _, avgs = get_mean_per_bin(data["z"], data["mag"])
    print(avgs)
```

So have broken things into functions to follow SRP. The avgs = line requires stepping through it. Also good to draw attention to the destructive assignment in the second last line.

FUN PYTHON FEATURES

Logging

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


def do_something(a,b):
    logging.info("<-- Look at the extra information!")
    return "%s %s" % (a, b)

print(do_something("No", "way"))
```

```
INFO:root:<-- Look at the extra information!
No way
```

As much fun as throwing a million print statements everywhere is, it can get confusing to keep track of them, especially if you're like me and often have print("EUGH"), print("sdfghjksfdhjk") and print("1"), print("2"), when youre desperately hunting for a bug. So use logging to keep track of all our output better!

The benefits of logging are: custom formats, differing log levels (critical, error, warning, info, debug), easily redirect them all into a different file and a ton more.

Logging

```
import logging

fmt = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
logging.basicConfig(level=logging.INFO, format=fmt)

class SomeClass(object):
    def __init__(self):
        self.logger = logging.getLogger(self.__class__.__name__)

    def do_something(self, a, b):
        self.logger.info("WHAT A USEFUL MESSAGE")
        if a == "No":
            self.logger.warning("WHY SO NEGATIVE?")
        return "%s %s" % (a, b)

print(SomeClass().do_something("No", "way"))
```

```
2017-11-16 12:45:47,521 - SomeClass - INFO - WHAT A USEFUL MESSAGE
2017-11-16 12:45:47,522 - SomeClass - WARNING - WHY SO NEGATIVE?
No way
```

As much fun as throwing a million print statements everywhere is, it can get confusing to keep track of them, especially if you're like me and often have print("EUGH"), print("sdfghjksfdhjk") and print("1"), print("2"), when youre desperately hunting for a bug. So use logging to keep track of all our output better!

The benefits of logging are: custom formats, differing log levels (critical, error, warning, info, debug), easily redirect them all into a different file and a ton more. In this example, I have a class logger which has the same name as the class itself. You can create new loggers whenever you want with different names, it doesnt have to be in the class

Args and kwargs

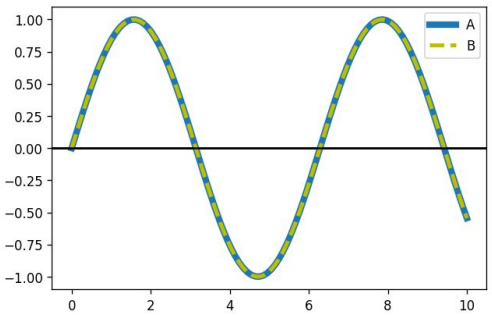
```
def do_math(a, b, c):
    return a * (b ** c)

vals = [1, 2, 3]
print(do_math(vals[0], vals[1], vals[2]))
print(do_math(*vals))
```

```
def alt_max(*vals):
    # vals is a tuple
    return max(vals)

print(alt_max(3, 5, 7, 4, 3))
```

You can add args to functions and methods so they can take arbitrary number of arguments. However, the more common usecase is to use the splat operator to unwrap a list of options when the function you are calling wants them separate.

Args and kwargs

```
def plot_with_options(x, y, label="Data", **kwargs):
    plt.plot(x, y, label=label, **kwargs)
    plt.axhline(0, c='k')
    plt.legend()

x = np.linspace(0, 10, 100)
y = np.sin(x)
opts = {"c": "y", "ls": "--"}

plot_with_options(x, y, lw=5.0, label="A")
plot_with_options(x, y, label="B", lw=3.0, **opts)
```



Like args, there are kwargs. Example is chosen to show both how they can be used in a function signature, and how you can unpack dictionaries to pass them in as kwargs. The example use is something I see a lot of, where plotting calls or similar have duplicate entries (like all plots are made with linewidth thicker for example). Instead of having to change each call if you decide to update, you can put the common arguments into a dictionary and then unpack them. Included the linewidth in the “B” plot call to illustrate that you can combine inline keywords with unpacked keywords, you dont need to have one or the other.

Args and kwargs - tools of denial

```
# Almost everyone does this
x = np.random.normal(0, 1, 1000)
hist, bins = np.histogram(x, 100)

# When the actual syntax uses kwargs
x = np.random.normal(loc=0, scale=1, size=1000)
hist, bins = np.histogram(x, bins=100)

# There are dangers, so can shut them down like this
def histogram(data, *args, bins=10):
    assert len(args) == 0, "WAT U DOIN?"
    return np.histogram(data, bins=bins)

hist, bins = histogram(x, bins=100)
hist, bins = histogram(x, 100)  # ASSERTION ERROR
```

A feature of python is that if you finish specifying args and there are kwargs left, you can just keep specifying them without the keywords. HOWEVER, if any future update reorders the keywords, or adds a new one not at the end, or if there are any more arguments added, all previous code that doesnt specify the keywords will break.

If this might happen to code you are writing, you can deny this feature by routing all the non-keyword extra arguments into a tuple of *args - the only way to set the kwargs is then to use the keywords.

Unpacking (more splat)

Whilst we're splatting, you can use them to unpack better in Python3.

```
h, e, l, _, o = "hello"
h, *ell, o = "hello"
*hell, o = "hello"
```

Underscore is a placeholder, not a variable name. Use it to discard variables you dont want. Variable names indicate whats in them.

Generators

```
xs = range(1000000000) # ONE BILLION NUMBERS PLS
```

With Python3, you can run this. Python2, you will get a MemoryError

Traditional

```
def fibonacci(n):
    res = [0, 1]
    for i in range(2, n):
        res.append(res[-1] + res[-2])
    return res

for number in fibonacci(10):
    print(number)
```

Generator

```
def fibonacci2(n):
    a, b = 0, 1
    for i in range(n):
        yield a
        a, b = b, a + b

for number in fibonacci2(10):
    print(number)
```

Generator v2

```
def fibonacci3():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

So python2 allocates a huge array, runs of memory, and crashes. Generators dont preallocate everything, they only give you what you want when you ask for it (when you iterate through it)

Yes the traditional assumes n>2.

Two different ways of generating fibonacci numbers. The first is using the traditional approach like you would in languages like c. If you call this with a large n, like 10000, you will get a big hangup initially and then itll print everything out at once. With the generator method you start printing immediately, because you dont have to precompute everything. You only calculate the nth number when you request it, so its more efficient. The third example is showing that because of the stateful nature of the iterator you dont need to set an upper limit, you can just keep calling it however many times you like.

f-strings

Welcome to python 3.6

```
user = "Sam"
num = 20
cool_string = f"User {user} has {num} bugs to fix. Damn"
print(cool_string)
```

```
User Sam has 20 bugs to fix. Damn
```

```
name, age = "samuel hinton", 25
print(f"{name.title()} is {12*age} months old.")
```

```
Samuel Hinton is 300 months old.
```

```
good = ["pizza", "chocolate"]
name = "Bob"
print(f"{name} likes {good[0]} the most")
```

```
Bob likes pizza the most
```

f-strings are amazing, but youll need the latest python to get them. You can do a lot in the braces, because its evaluated directly (and cast to string). No more temp variables you just create for string formatting, and no more verbose formatting!

lru_cache

```
from functools import lru_cache

def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

@lru_cache(maxsize=None)
def fib2(n):
    if n < 2:
        return n
    return fib2(n-1) + fib2(n-2)

%time fib(35)
%time fib2(35)
print(fib2.cache_info())
```

```
Wall time: 3.94 s
Wall time: 0 ns
CacheInfo(hits=33, misses=36,
      maxsize=None, currsize=36)
```

What a beauty.

So lru_cache caches the last n (maxsize) number of calls to the function and the output. Set maxsize to a power of 2, or leave it to None to let it grow forever! If you have a stateless function you call a lot throwing the lru_cache annotation on it will speed things up a ton!

Decorators

```
import time

def timed(fn):
    def timer(*args, **kwargs):
        t_start = time.time()
        result = fn(*args, **kwargs)
        tdiff = time.time() - t_start
        print("%s took %0.2f sec" %
            (fn.__name__, tdiff))
        return result
    return timer

@timed
def stupid_sum():
    x = 0
    for i in range(10000000):
        x += i
    return x

val = stupid_sum()
```

```
def counted(fn):
    def inner(*args, **kwargs):
        inner.count += 1
        print("%s called %d times" %
            (fn.__name__, inner.count))
        return fn(*args, **kwargs)
    inner.count = 0
    return inner

@counted
def blah():
    pass

#blah = counted(blah)

for i in range(5):
    blah()
```

Right, so decorators are something I never see used but can actually be useful. Here are two simple examples, one to time a function, and another that counts the number of times a function is invoked.

The commented out line on the right is a way to think of what is happening. The decorator overwrites the original function pointer to a point of the inner function returned. That inner function generally invokes the original function (although you can make a decorate which calls a separate function instead or does nothing if you want). On the left is a stock standard function, on the right we use the fact a function is an object and attach a variable to the function called count.

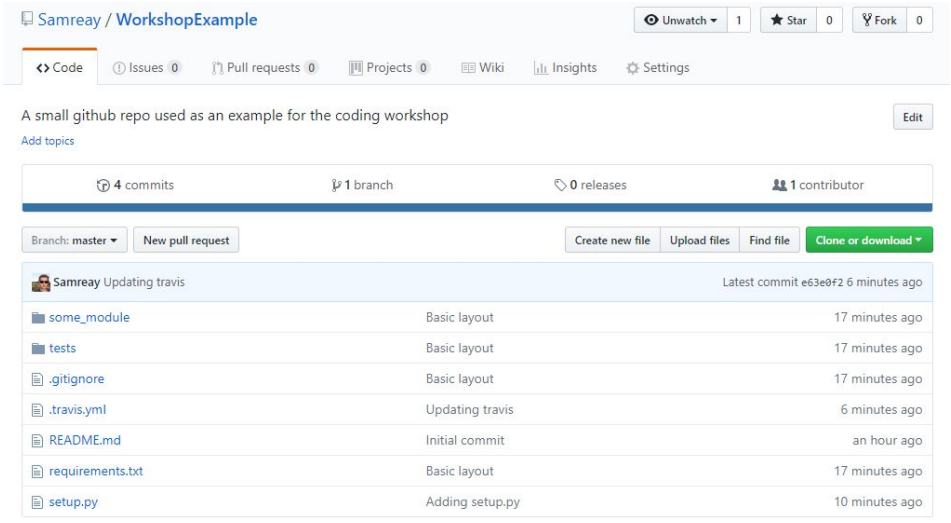
These decorators are examples and not very useful, useful ones are like lru_cache

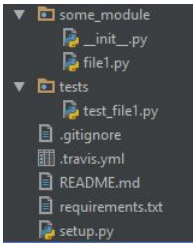
Summary

Write good code.

Forking a Github Repo

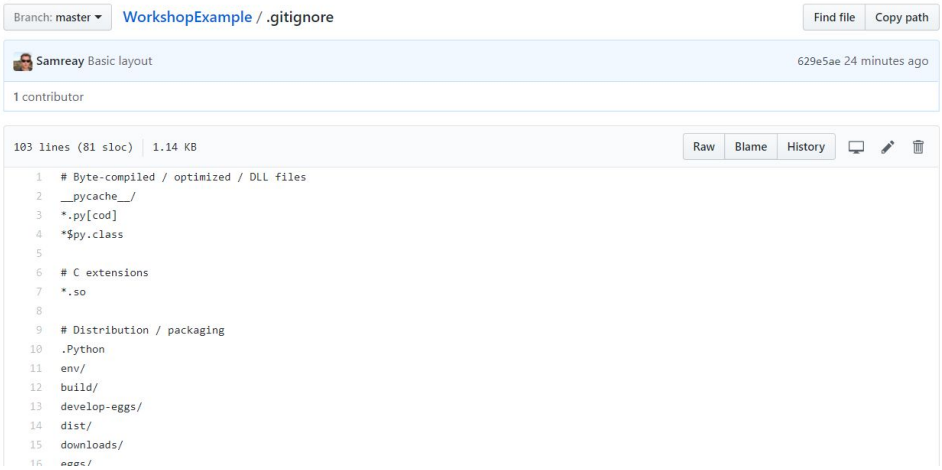
Setting up a project





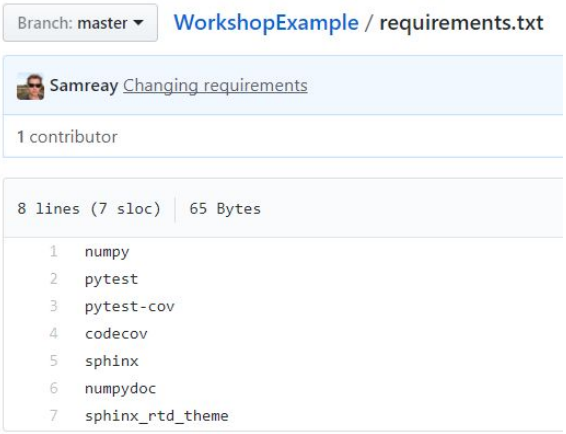
Made a Github project to show a build up. There is some module, and some tests, with the normal files - gitignore, setup.py, requirements.txt. Lets run through them real quick.

Setting up a project

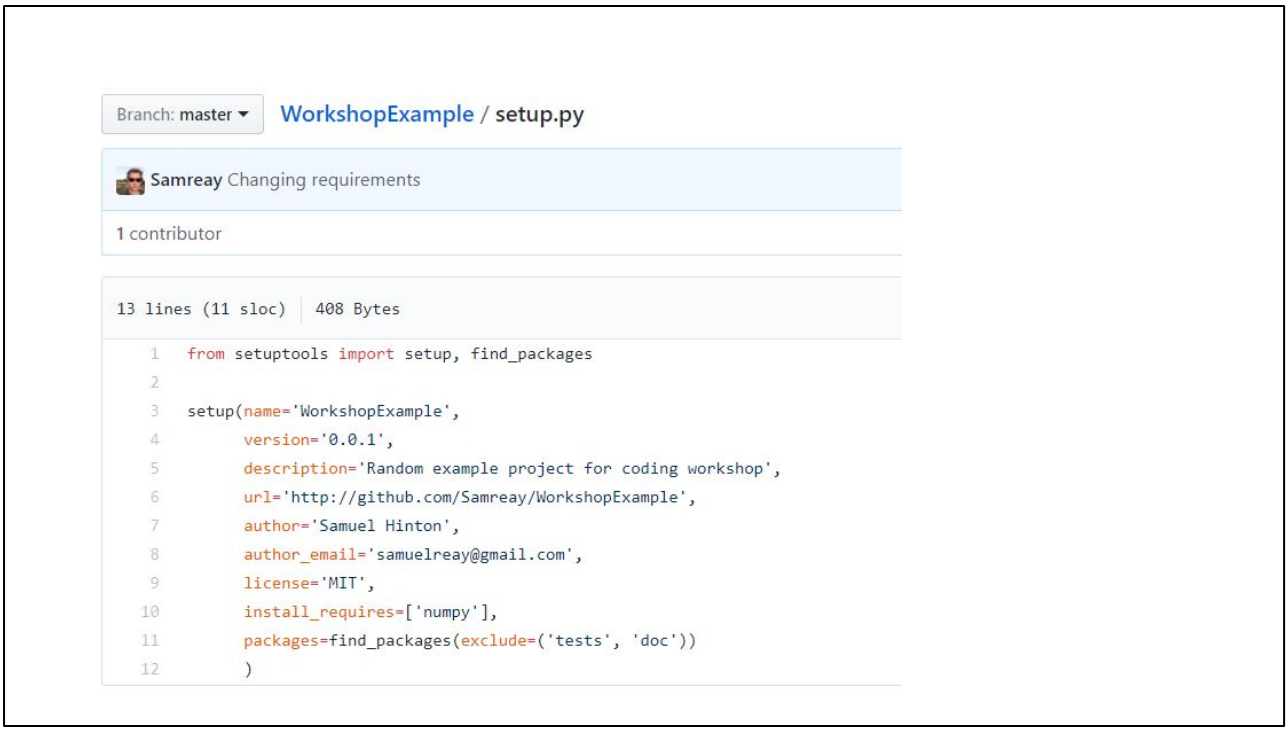


Gitignore. Github will generate this for you if you pick a language when you create the repo.

Setting up a project



Requirements file. Should list every dependency you need to install to get your project running, tested, documented. Everything.



Okay, so most of this is self explanatory. Two things which are worth pointing out - normally you can just write the packages (in this case, it would be some_module), but to make it more general Im using the find_packages function, and excluding the tests and doc directory (doc doesnt exist right now, but soon).

The main thing to point out here is the difference between requirements.txt and install_requirements. Install_requirements should document the most general, loosest possible requirements for installing and running the package. This is what PyPI uses to install dependencies when you pip install. Requirements.txt on the other hand is meant to declare “Using these specific packages/versions, you can define a working development/test environment”. So one is general user focused, and one is dev focused.

Task: Forking a Github Project

https://github.com/Samreay/WorkshopExample

On Github, everyone fork the project, clone it using PyCharm and commit a small change.

Ensure you find/replace my username with yours at the very least.

And now, lets get everyone to open this repo, fork it, clone it and get commits working. What I found useful to do is grab the laptop of someone already familiar with git/python (that would be able to do this on their own) and do it live with their github account.

The important step for them to do initial is to find and replace (right click WorkshopExample in Pycharm, replace in path, samreay -> their_username, done) my username with theirs, so that things point to their repo and not mine.

And then onto code testing

Code Testing

Fail hard. Fail fast.

Code without guilt or worry

This is the main point - modify, change, add, remove, do whatever you want with your code and dont worry about breaking current features or current workflow when it might impact your results, youll know if anything changes right away and diagnose issues right where the change has happened instead of spending days tracking down an issue because a number spat out at the end of your analysis is now wrong.

Fundamentals

- Unit tests vs integration tests
- How to test
- Testing pipelines and utilities

Categories

Unit Tests

- One test on the ***smallest*** scale
- Each test tests ***one*** thing
- Cheap and fast to run

Integration Tests

- Tests how pieces interconnect
- Tests from ***start to end***
- Slow and expensive to run

So just a brief overview of the difference between unit tests and integration tests so that I can quickly go straight to unit tests and the context is clear

Unit Tests

```
def count_occurences(search, document):
    """ Count the occurences of one string in another, case insensitive

    Finds the number of occurences, including overlapping cases,
    for how many times one string is present inside another. Uses
    a simple brute force algorithm, so try not to pass in massive strings

    Parameters
    ----------
    search : str
        The string to look for
    document : str
        The string to look for the `search` term inside

    Returns
    -------
    int
        The number of occurences

    Examples
    --------
    >>> count_occurences("bob", "Hey Bob, Bobby, Bobo, Bobobo")
    5

    """
    count = 0
    document = document.lower()
    n = len(search)
    for i in range(len(document)):
        if document[i:i+n] == search:
            count += 1
    return count
```

So let's be clear, such a simple method doesn't need so much docstring, but for posterity and repeating how to do it, here it is. But as to "What does this method do" we now read a sentence instead of a section of code.

In terms of what should be tested, the obvious one here is to test the given example. There are at least two bugs in the above code, one of which Ill point out and fix in later slides (empty string search matches everything, the other is also fairly obvious - search should be cast to lowercase but isnt). The point is eventually we will get 100% test coverage on our code and yet there is still a bug, because 100% coverage != 100% tested

Unit Tests

How would you test the following function?

```
def count_occurence(search, document):
    count = 0
    document = document.tolowercase()
    n = len(search)
    for i in range(len(document)):
        if document[i:i+n] == search:
            count += 1
    return count
```

So thought it would be good to jump straight in. The function above just counts the number of occurrences of one string inside another by brute force. But dont spend time on this slide, jump to the next because documentation, reinforce the point of why doco is useful.

Unit Tests

General Principle: **0, 1 and many**

Catch the edge cases!

```
def test_count_occurence_simple():
    assert count_occurences("bob", "Hey Bob, Bobby, Bobo, Bobobo") == 5

def test_count_occurence_letter():
    assert count_occurences("a", "a") == 1

def test_count_occurence_empty_search():
    assert count_occurences("", "hello") == 0

def test_count_occurence_empty_document():
    assert count_occurences("a", "") == 0

def test_count_occurence_empty_both():
    assert count_occurences("", "") == 0
```

So lets say we come up with these five test cases to start with, but havent run them yet. Now the point I want to get across here is that *we dont need to method to exist to write the test cases*. In some cases in code development, you write the tests *before* you write the algorithm itself, because the tests should define how the function acts. The above list of five isnt an exhaustive list of tests,Ill add something a bit more complicated later on, but these are basic tests that we can probably all agree on. Test case sensitvity, multiple matches, single matches, empty string perumations. Have no tested type checking.

The other important thing to note is that unit tests need to have more coverage on API or external methods, its not necessary to unit test every function you ever write, and in some cases for internal functions the behaviour may be explicitly undefined if the inputs are wrong (type/value, etc). Because those functions are called by your code, and not an unknown user, you dont need to be as defense.

Running Unit Tests

```
test_count_occurence_simple()
test_count_occurence_letter()
test_count_occurence_empty_search()
test_count_occurence_empty_document()
test_count_occurence_empty_both()
```

Never, never do this.

No metrics. Will abort on first failure. **WILL RUN ON IMPORT.**

Running Unit Tests

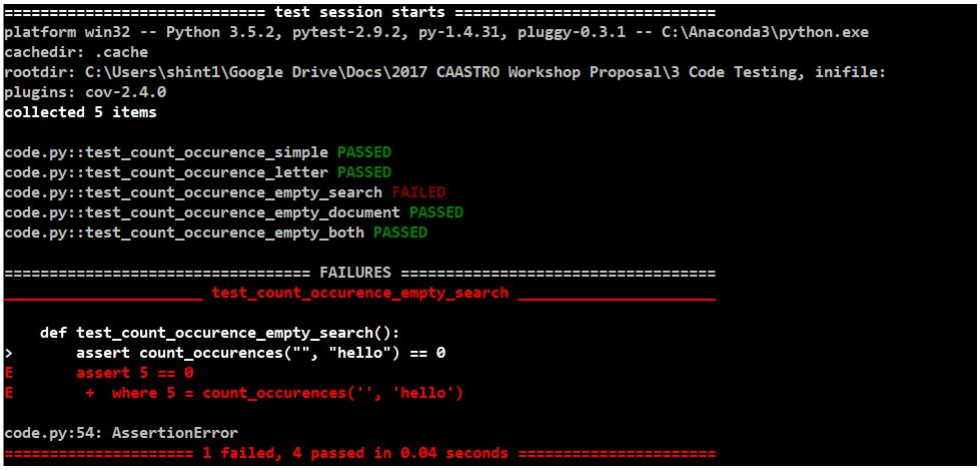
```
if __name__ == "__main__":
    test_count_occurence_simple()
    test_count_occurence_letter()
    test_count_occurence_empty_search()
    test_count_occurence_empty_document()
    test_count_occurence_empty_both()
```

Never, never do this.

No metrics. Will abort on first failure.

Running Unit Tests With PyTest

`pytest code.py` if not on path: `python -m pytest -v code.py`



Yay for pytest

So you dont need to write anything other than the functions in your file. Pytest will look through the file, and run any function that starts with “test_”. It will do the same for classes like “TestClass” and execute “test_*” methods. The -v is for verbose so you can see the passed ones as well. See that it runs all tests, and instead of just saying “AssertionError” the previous two slides, it actually breaks down why the assertion failed! How useful!

Actually they dont even need to be in the same file. We can remove the “code.py” part of our call, and make a file called test_code.py, and because it matches the “test_” discovery pattern pytest will pick it up automatically. (Next slide has those patterns on it)

PyTest Discovery Rules

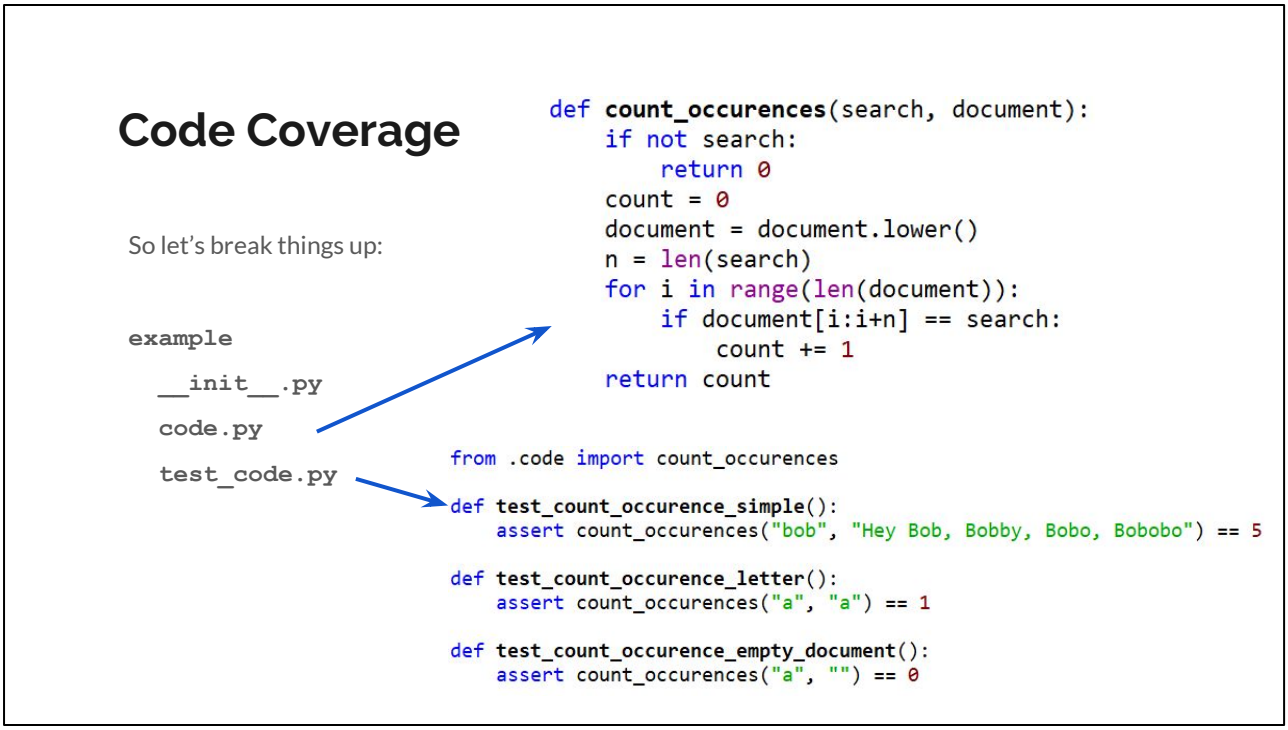
https://docs.pytest.org/en/latest/goodpractices.html#test-discovery

`test_*.py`

`*_test.py`

(Recursive directory search)

```
def test_*
class Test*:
    def test_*
```

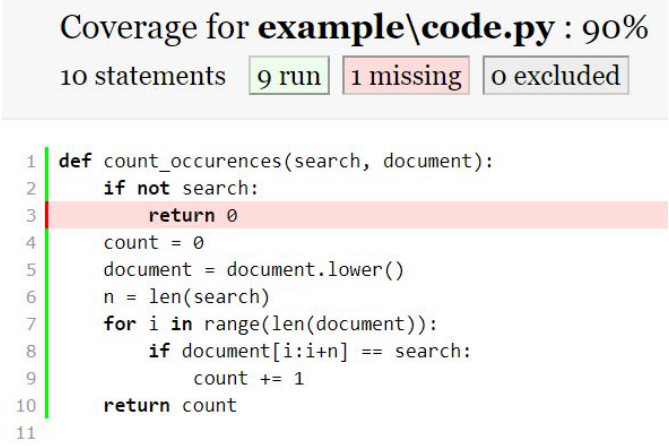


Using the previous example, lets modify two things. First, to deal with that test failure, lets check for search being empty, and return 0 if that is the case. Secondly, Im going to remove the two tests which actually used an empty search, so I can demonstrate code coverage better. Ive also put the tests in a separate file, and put everything in a folder with an __init__.py, so that we have an example module. Might be worthwhile here to explain what __init__.py does - it tells python to treat that directory as a module (ie treat it as a source directory and let imports work, etc)

Code Coverage

```
python -m pytest --cov-report html --verbose --cov example
```

Needs **`pytest-cov`** to be installed, but we can now say "Give us an HTML coverage report of the example module." And suddenly a htmlcov directory exists and we open the index.html:



So we can ask for an html coverage report with the command above, and itll show you all the files and how much of the code is covered. We can see that by removing those two test cases our tests never enter the if statement, and so the coverage report is telling us to add them back in! In the main index.html itll also show the coverage for the test file (which should be 100% because pytest ran all the tests). However, this isnt a useful file to show, and we can specify which files to compute coverage on using a .coveragerc file and excluding anything with test in the name, or exclude a test directory, etc.

Overdoing it

```
def count_occurences(search, document):
    assert isinstance(search, str), \
        "search should be a string"
    assert isinstance(document, str), \
        "document should be a string"
    if not search:
        return 0
    count = 0
    document = document.lower()
    n = len(search)
    for i in range(len(document)):
        if document[i:i+n] == search:
            count += 1
    return count
```

```
from .code import count_occurences
import pytest

def test_count_occurence_simple():
    assert count_occurences("bob", "Bob,Bobby,Bo

def test_count_occurence_letter():
    assert count_occurences("a", "a") == 1

def test_count_occurence_empty_document():
    assert count_occurences("a", "") == 0

def test_count_occurence_empty_search():
    assert count_occurences("", "hello") == 0

def test_count_occurence_empty_both():
    assert count_occurences("", "") == 0

def test_count_occurence_not_string_search():
    with pytest.raises(AssertionError):
        count_occurences(0, "hello")

def test_count_occurence_not_string_document():
    with pytest.raises(AssertionError):
        count_occurences("a", {})
```

So here are a set of tests which will get 100% code coverage. Its probably also over the top, imagine trying to do this for *every* function you write - its just not going to happen. The style of coding shown on the left, type checking, boundary checking, etc, is known as defensive coding, and it only viable or needed in specific circumstances.

Note here we have 100% test coverage, but there is still a bug - search is never cast to lwoer

Defensive Coding

Trust nothing. Plan for everything.

Many sorts of defensive coding, but main one in astro context is “Validating input”. Assume you will be given garbage, and either check for garbage or ensure your algorithm can handle it.

ONLY *worth the effort for external APIs.*

Private functions are rarely worth the effort!

So defensive coding is needed in high stake applications where you need to make sure nothing the user does crashes your program (like trading on the stockmarket). But this isnt too relevant to our use cases, most of what we want to do testing for is for our own benefit. Ie if we write a numerical integration function, to test it on functions with analytic solutions so that we can gain confidence in our own code.

Integration Tests

Relevant for projects with multiple connecting pieces, where you test how the pieces interact.

Each individual component should already be unit tested.

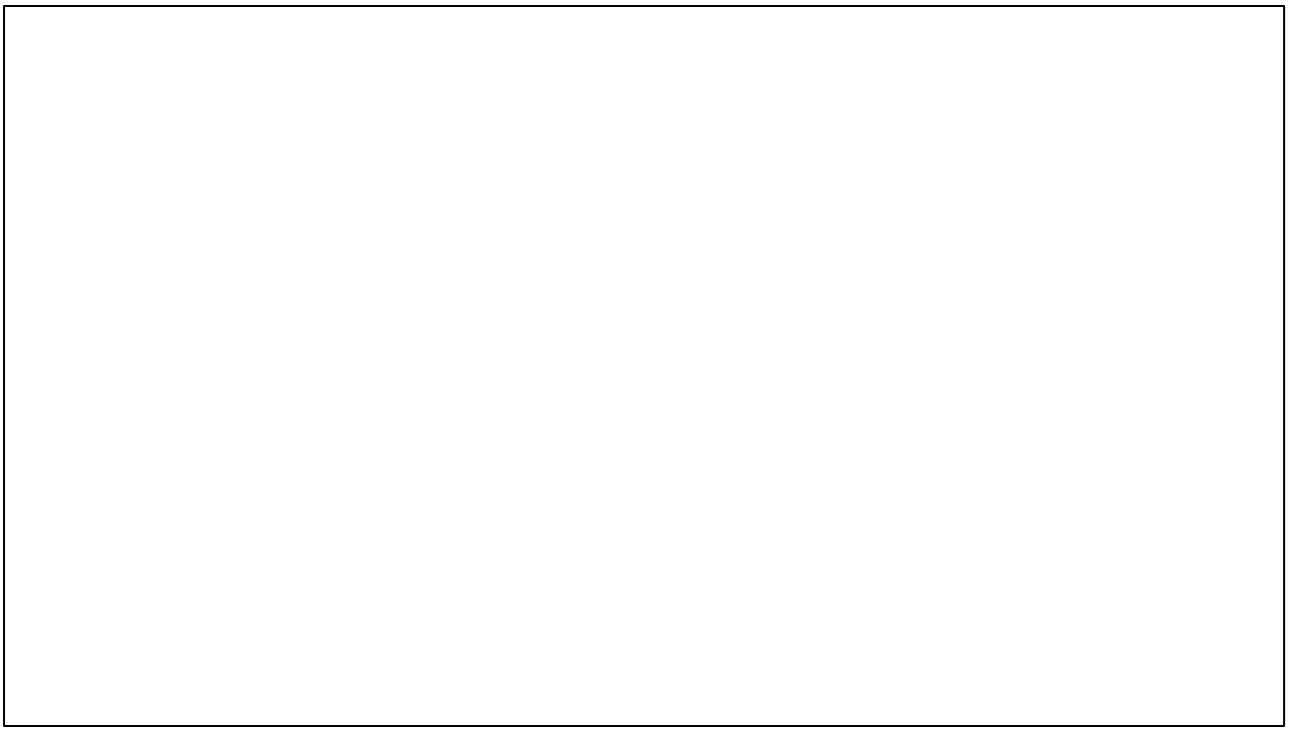
```python
class DataLoader(object):
    def load_data(self, filename):
        pass # Return data

class Parser(object):
    def parse_data(self, data):
        pass # Return some parsed data

def test_data_integration():
    filename = 'test_data.txt'
    data_loader = DataLoader()
    data = data_loader.load_data(filename)
    parser = Parser()
    parsed_data = parser.parse_data(data)
    assert parsed_data is not None
    # Assert parsed_data is a valid product
```

Integration testing isnt too common or needed in astro. But if youre making a data pipeline with multiple components, this would be what you want to have! Or anything which properly separates out things like data loaders, serialisers, and an analysis. Integration testing is, as the name suggestions, about testing how the difference pieces you have constructed fit and integrate together.

System Tests

Consider a project like a data pipeline. A system test would validate the entire pipeline, feeding in test data and validating the output products.

The layer above this integration testing is system testing, where you essentially test everything, start to finish. Ie a data pipeline would have system testing where fake ‘raw telescope data’ is fed in one end, and processed galaxy properties come out the other. Useful normally only in larger projects, would not be done for scripts.



Running Tests

1. Have the repository cloned and ready to go
2. Open cmd/terminal/shell of choice in the top directory
3. Make sure you have all deps: **`pip install -r requirements.txt`**
4. Ensure tests work:
   a. **`pytest -v --cov`** (nix) **`python -m pytest -v --cov`** (Windows)
5. Have a function **`integrate_trapz`** in **`code.py`**
   a. Lets add another test - integrating a triangle
   b. If you're shooting ahead, there is another function in **`code.py`** you can test. Add it to the **`__init__.py`** to see it though!

Code Documentation

Just do it

Fundamentals

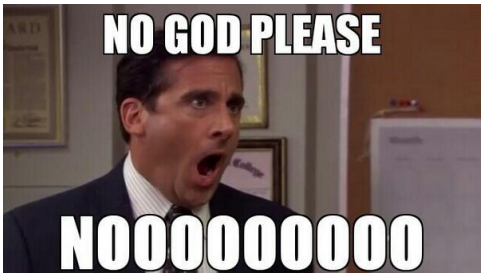
- What Not To Do
- Self Documenting Code
- Comments vs Docstring
- How to Docstring

Signposting.

What Not To Do

If you've ever seen an *Introduction to Coding* assignment with marks for code comments you know where I'm going.

```
# initialise variables
acc = 0
# Loop over all items
for item in alist:
    # Add them up
    acc += item
# end loop
```



Comments should never turn basic code into english.

Self Documenting Code

Remember SRP? Good, we're done here.

```
if __name__ == "__main__":
    filename_data = "supernovae.txt"
    filename_plot = "cosmology.png"

    supernovae_data = load_data(filename_data)
    chains = fit_cosmology(supernovae_data)
    plot_cosmology(chains, filename_plot)
```

```
if __name__ == "__main__":
    directory = "data"
    simulations = [load_simulation(f) for f in get_files(directory)]
    cor = get_correlation_function(simulations)
```

Additionally, each time you feel like you should add a comment, consider refactoring instead. Note in the examples here the overall flow of the program or script is obvious because all the logic and algorithmic trickery is hidden inside functions.

Inline Comments vs Docstring

Inline comments:

- SPARSE
- Short

Docstring:

- Well structured
- Verbose
- Literally everything

Inline comments should be used sparsely, to either provide a “why” you’re doing something, or to summarise a code block which for some reason hasnt been refactored out into its own function. Generally the comment is something that *should* be its own method, but due to a large number of dependent variables (that would create a massive signature) or a high degree of state (having to pass back a dozen things) makes it better to leave inline.

Docstring is the opposite. For those used to linux and similar, think of docstring like you would `man`.

The structure of docstring

Focusing on **reStructuredText** docstring here (PEP287), specifically **Numpydoc** syntax.

1. Opening line - one sentence description of method. This is what becomes a tooltip.
2. A more verbose section explaining what the method does. Can include math or visual aid.
3. Details of input parameters - data type, defaults, valid inputs, etc.
4. The return value of the method
5. Potential code examples
6. Potential references
7. Potential outbound links to similar functions

https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

So what can docstring not do? So note that there are a bunch of valid documentation formats (epytext, reST, Google, Numpydoc, more). Going to focus on numpydoc as its used by numpy, scipy and matplotlib, so very easy to find examples and look at those libraries for reference.

Very Basics

```
def add(x, y):
    """ The sum of two numbers.

    Adds the two supplied input arguments and returns their summed value.
    The input arguments must be numeric, no type checking is performed.

    Parameters
    ----------
    x : numeric
        The first argument to add.
    y : numeric
        The second argument to add.

    Returns
    -------
    numeric
        The summation of `x` and `y`.
    """
    return x + y
```

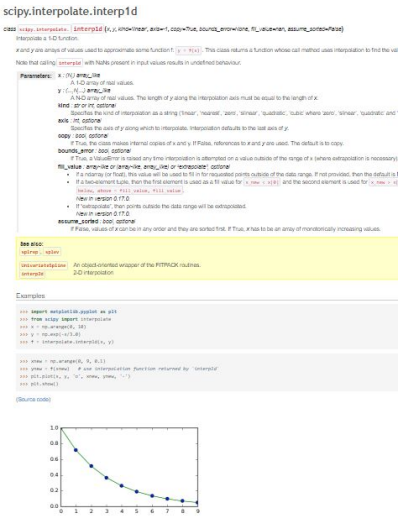
Let's get fancy

```
def get_mean_per_bin(x, y, bins=10):
    """ Returns the mean of `y`, binned in `x`.

    Computes the binned mean statistic for `y` for an arbitrary
    number of bins of `x`.

    Parameters
    ----------
    x : array-like[numeric]
        The vector to bin.
    y : array-like[numeric]
        The values to take the mean of.
    bins : int, optional
        The number of bins to use, must be a positive integer.
        Defaults to 10.

    Returns
    -------
    np.ndarray
        The bin edges.
    np.ndarray
        The binned averages.

    See Also
    --------
    scipy.stats.binned_statistic: General binned statistics

    Notes
    -----
    This method is totally redundant. What is a mean? From [1]_:

    .. math::
        \mu = \frac{1}{N} \sum_{i=0}^N y_i

    References
    ----------
    .. [1] A. Child, "Primary School Maths Class", School, 2017.

    Examples
    --------
    >>> x, y = np.arange(101), np.arange(101)
    >>> get_mean_per_bin(x, y, bins=2)
    (array([   0.,   50.,  100.]), array([ 24.5,  74.5]))

    See, simple!
    """
```

So what can docstring not do? So note that there are a bunch of valid documentation formats (epytext, reST, Google, Numpydoc, more). Going to focus on numpydoc as its used by numpy, scipy and matplotlib, so very easy to find examples and look at those libraries for reference.

WHO CAN BE BOTHERED?

Numpy, Astropy, Scipy, Matplotlib… everyone



https://github.com/scipy/scipy/blob/v0.19.1/scipy/interpolate/interpolate.py#L321-L647

```
class interp1d(_Interpolator1D):
    """
    Interpolate a 1-D function.

    `x` and `y` are arrays of values used to approximate some function f:
    ``y = f(x)``.  This class returns a function whose call method uses
    interpolation to find the value of new points.

    Note that calling `interp1d` with NaNs present in input values results in
    undefined behaviour.

    Parameters
    ----------
    x : (N,) array_like
        A 1-D array of real values.
```

Also important to note here that the methods which you expect users to use get the most documentation. Internal methods definitely do not need to have examples and detailed notes!

DOCSTRING ⟶ DOCS

TECHNICAL IMPLEMENTATION

I should probably address how to go from one to the other shouldn't I?

Sphinx

http://www.sphinx-doc.org/en/stable/

http://www.sphinx-doc.org/en/stable/tutorial.html

- Write your own doc pages
- Automatically create pages from docstring
- Publish to HTML/PDF/ePub/PlainText

```
pip install Sphinx

pip install numpydoc

sphinx-quickstart
```

So two useful links for starting with Sphinx. Installing and setting up a new project is easy, run the quickstart in the top level folder.

.rst

Sphinx doc pages are .rst - **reStructured Text**

See the index.rst, fairly empty.

```
make html
```

```
make latex
```

```
.. Test documentation master file, created by
   sphinx-quickstart on Mon Nov 13 19:11:02 2017.
   You can adapt this file completely to your
   liking, but it should at least
   contain the root `toctree` directive.

Welcome to Test's documentation!
================================

Contents:

.. toctree::
   :maxdepth: 2


Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

So this is the base file that gets generated. Empty page essentially. Can make it as a website and as a pdf for demonstration

.rst

And this is the website itself.

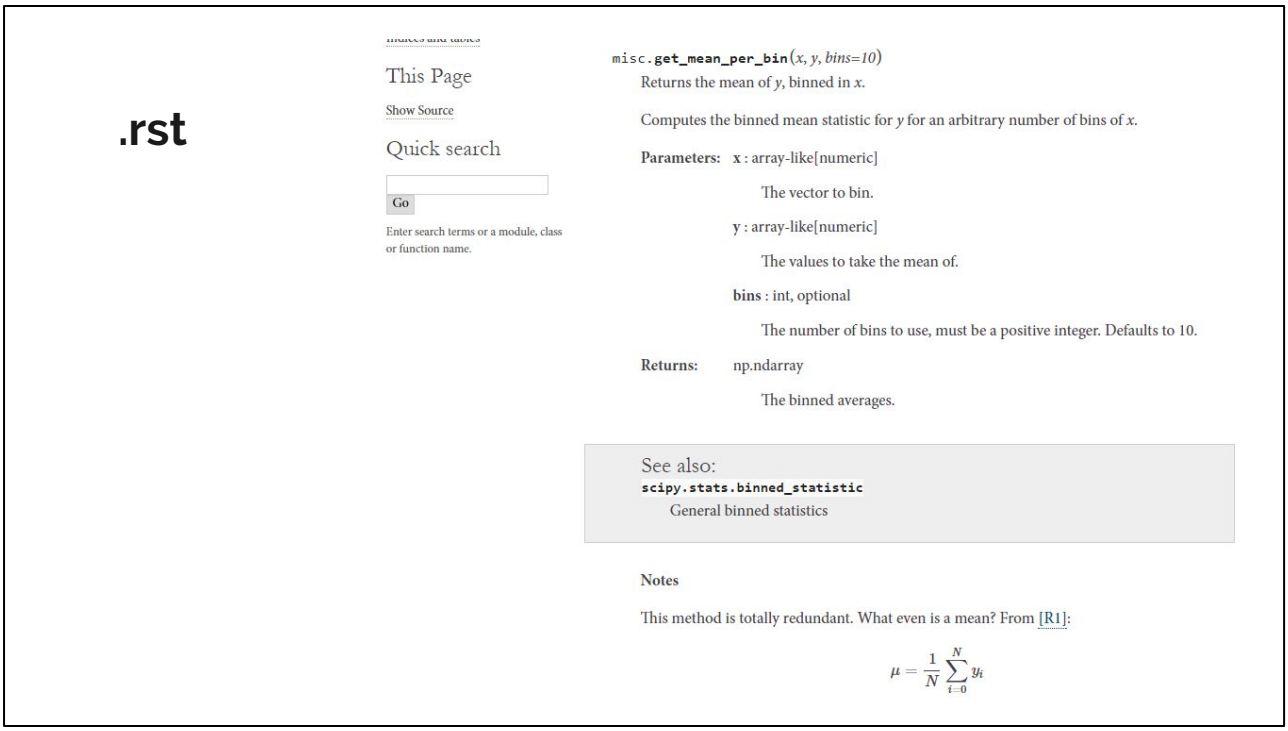
.rst

```
Welcome to Test's documentation!
================================

Contents:

.. toctree::
   :maxdepth: 2

.. automodule:: misc
   :members:

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

```
# If extensions (or modules to document with a
# add these directories to sys.path here. If t
# documentation root, use os.path.abspath to m
# sys.path.insert(0, os.path.abspath('.'))
sys.path.insert(0, os.path.abspath('./src'))

# -- General configuration ------------------

# If your documentation needs a minimal Sphinx
#needs_sphinx = '1.0'

# Add any Sphinx extension module names here,
# extensions coming with Sphinx (named 'sphinx
# ones.
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.mathjax',
    'numpydoc'
]
```

So to get our docstring in there we need to add it to the index.rst, and also tell the newly created conf.py where our code is. Also lets not forget to activate numpydoc in conf.py

.rst



This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

`misc.get_mean_per_bin(x, y, bins=10)`

Returns the mean of *y*, binned in *x*.

Computes the binned mean statistic for *y* for an arbitrary number of bins of *x*.

Parameters: **x** : array-like[numeric]

The vector to bin.

y : array-like[numeric]

The values to take the mean of.

bins : int, optional

The number of bins to use, must be a positive integer. Defaults to 10.

Returns: np.ndarray

The binned averages.

See also:

`scipy.stats.binned_statistic`

General binned statistics

Notes

This method is totally redundant. What even is a mean? From [R1]:

$$\mu = \frac{1}{N} \sum_{i=0}^{N} y_i$$

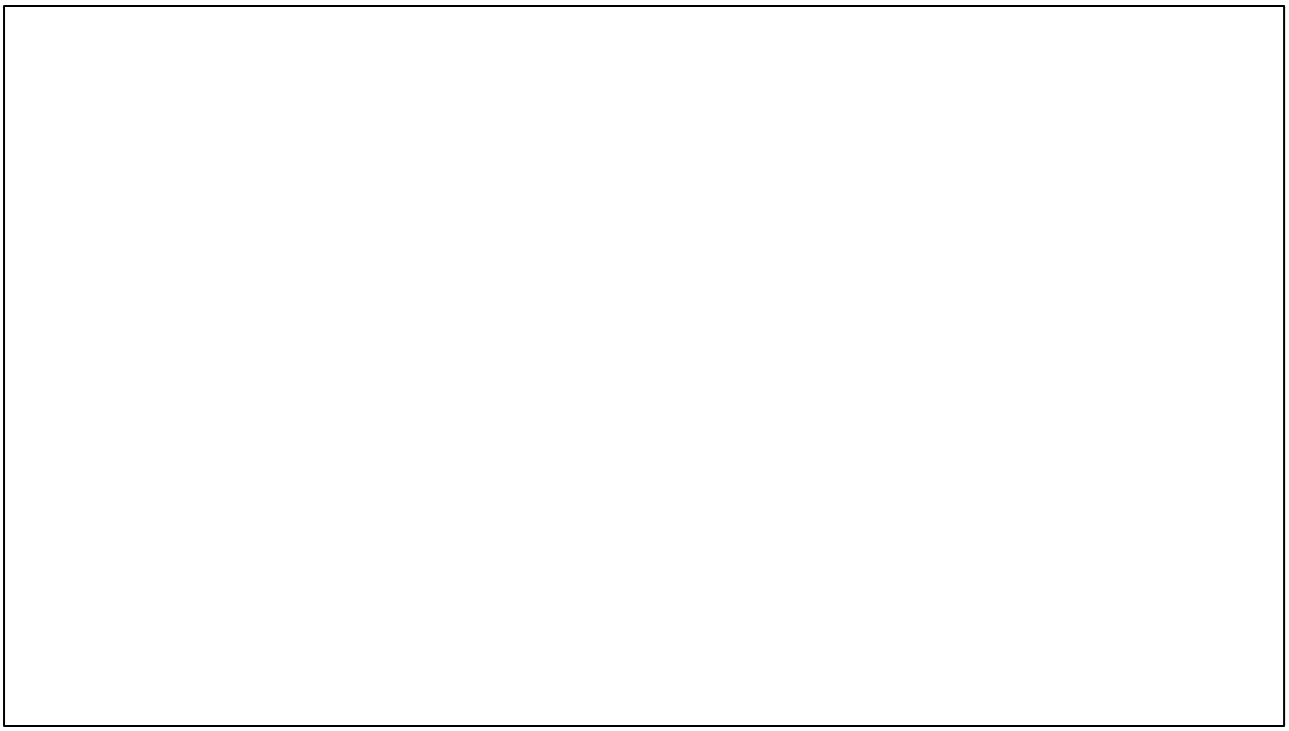
This is the output after running make html again

Summary

Documentation is a necessary evil. Please do it.

Sphinx makes it easy, numpydoc has a million examples to make it even simpler.

Will do an interactive sphinx example alongside testing at the end.



Making doco

1. Navigate to do the doc directory
2. `make html`
3. Either open doc/_build/html/index.html yourself, or open as a server in PyCharm
4. Document the `integrate_trapz` method in `code.py`
5. Observe the fruits of your labour!
6. For extra fun, you can either document the other method, or try `make latex`

Shipping Code

Get the message out to the masses!

The story so far...

Have a project. Tested.

Documented.

Finally happy for someone else

to see it!

But... how?

Right, so hopefully this section should be quick so we can do the big interactive thing. Will cover PyPI, Zenodo and project distribution methods.

Go both methods!

PyPI

Get your packages pip installable.

Zenodo

Get your code archived and citable.

Its important to realise PyPI and Zenodo are completely different, so Ill just state it outright. And then move onto PyPI

PyPI

Register for user: https://pypi.python.org/pypi?%3Aaction=register_form

Variety of tutorials:

- https://blog.jetbrains.com/pycharm/2017/05/how-to-publish-your-package-on-pypi/
- https://www.codementor.io/arpitbhayani/host-your-python-package-using-github-on-pypi-du107t7ku
- http://peterdowns.com/posts/first-time-with-pypi.html

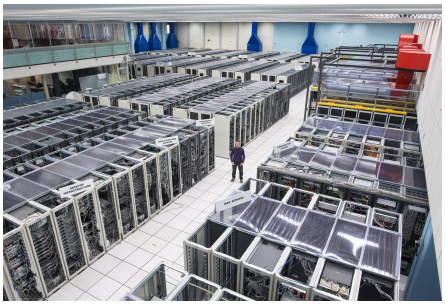
I wont do the PyPI stuff interactively, because (unless youre on the test servers), you cant screw around with PyPI. There is no undo, no delete forever, its serious stuff, so for posterity here are a bunch of useful tutorials before I go over it very quickly myself.

PyPI

1. Become a PyPI user
2. Setup your **`.pypirc`** as per the tutorials
3. Ensure your `setup.py` is good (package name, dependencies, it all comes from that).
4. Check you can install it manually - **`python setup.py install`**
5. Register your package - **`python setup.py register`**
   a. **`python setup.py register -r`** **https://testpypi.python.org/pypi** (for the test servers)
6. Build your source distribution - **`python setup.py sdist`**
   a. (This will be in the dist folder now)
7. Upload your package - **`twine upload dist/packagename.zip`**
   a. **`twine upload -r test dist/packagename.zip`** (if you registered to the test servers)

So there are a few things to note here above the actual steps. Most tutorials will recommend you upload via `python setup.py sdist upload`, but note that this sends your password and username plaintext. Imma pass on that, and twine was created to more securely upload your packages. The other thing to note is that the setup.py we have at the moment is very, very minimalistic. There are a bunch of extra fields we can add if we want more detail. The other big thing we should do is manage the versions better, at the moment the version is hardcoded into the setup.py, but better if the version is defined in the package (ie numpy.__version__) and setup.py uses this. You can see how to do that if you look at the setup.py for chainconsumer - it searches the main file for __version__ with regex and extracts the version.

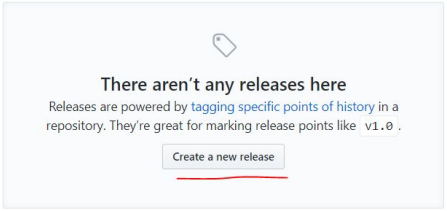
Zenodo



- Versioned *digital* archive
  - Code
  - Datasets
- Generates DOI - makes code **citable**
- Very robust servers
- Integrates with Github
  - https://guides.github.com/activities/citable-code/

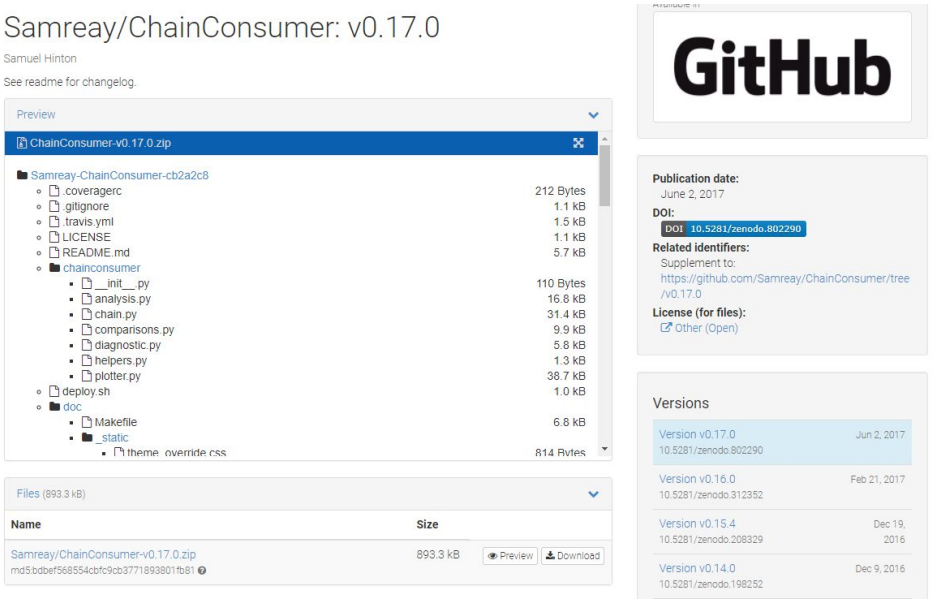
Zenodo

1. Log into zenodo, via Github
2. Pick the repository
3. In Github, create a release (which uses git tags)



Zenodo

Example:



Project Distributions

Project services

1. Set up an environment
2. Install dependencies
3. **Install project**
4. Download data
5. Run on the data
6. Create output products

We've focused on getting step 3 down pat. But what about the other steps?

Project services

Anaconda Projects

Anaconda Project encapsulates data science projects and makes them easily portable. Project automates setup steps such as installing the right packages, downloading files, setting environment variables and running commands.

Docker

The Docker platform is the only container platform to build, secure and manage the widest array of applications from development to production both on premises and in the cloud

Project services

Anaconda Projects

- Python focused
- Relatively new
- Comes with conda install
- Integrated Jupyter notebook support

Docker

- Generic applications, not just python
- Highly used and well tested
- Separate install
- Jupyter notebook support with right image

Project services

Anaconda Projects Example

https://blog.daftcode.pl/how-to-wrap-your-repo-with-anaconda-project-c7ee2259ec42

Docker Example

https://docs.docker.com/samples/library/python/#create-a-dockerfile-in-your-python-app-project

For setting up iPython notebook:

https://www.dataquest.io/blog/docker-data-science/

Global Summary

1. Follow SRP to write readable, concise code.
2. Test the code for your own piece of mind
3. Add docstring to relevant functions
4. Ship it to PyPI/Zenodo, ship distributions using Projects/Docker

We've focused on getting step 3 down pat. But what about the other steps?

Fin

Because we simply dont have the time or bandwidth to give an interactive example of a full pipeline!