

Documentation

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON



Adam Spannbauer

Machine Learning Engineer at Eastman

Documentation in Python

- Comments

```
# Square the number x
```

- Docstrings

```
"""Square the number x

:param x: number to square
:return: x squared

>>> square(2)
4
"""
```

Comments

```
# This is a valid comment  
x = 2
```

```
y = 3 # This is also a valid comment
```

```
# You can't see me unless you look at the source code  
# Hi future collaborators!!
```

Effective comments

Commenting 'what'

```
# Define people as 5
people = 5

# Multiply people by 3
people * 3
```

Commenting 'why'

```
# There will be 5 people attending the party
people = 5

# We need 3 pieces of pizza per person
people * 3
```

Docstrings

```
def function(x):  
    """High level description of function  
  
    Additional details on function
```

Docstrings

```
def function(x):  
    """High level description of function  
  
    Additional details on function  
  
    :param x: description of parameter x  
    :return: description of return value
```

Example webpage generated from a docstring in the Flask package.

Docstrings

```
def function(x):  
    """High level description of function  
  
    Additional details on function  
  
    :param x: description of parameter x  
    :return: description of return value  
  
    >>> # Example function usage  
    Expected output of example function usage  
    """"  
    # function code
```

Example docstring

```
def square(x):  
    """Square the number x  
  
    :param x: number to square  
    :return: x squared  
  
>>> square(2)  
4  
"""  
  
# `x * x` is faster than `x ** 2`  
# reference: https://stackoverflow.com/a/29055266/5731525  
    return x * x
```


Example docstring output

```
help(square)
```

```
square(x)
    Square the number x

:param x: number to square
:return: x squared

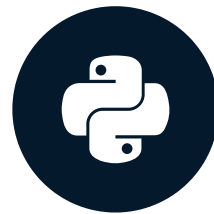
>>> square(2)
4
```

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Readability counts

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON



Adam Spannbauer
Machine Learning Engineer

The Zen of Python

```
import this
```

2. The Zen of Python

Luckily, Python has a big focus on readability, and it comes with a list of guidelines to help write good code. These guidelines are known as the 'Zen of Python', and you can view them through the command 'import this'. Here we see an abridged version. Notice the item stating, 'readability counts', and the rest of the items can guide us in this quest for readability.

The Zen of Python, by Tim Peters (abridged)

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

The complex is better than complicated.

Readability counts.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Descriptive naming

Poor naming

```
def check(x, y=100):  
    return x >= y
```

Descriptive naming

```
def is_boiling(temp, boiling_point=100):  
    return temp >= boiling_point
```

Going overboard

```
def check_if_temperature_is_above_boiling_point(  
    temperature_to_check,  
    celsius_water_boiling_point=100):  
    return temperature_to_check >= celsius_water_boiling_point
```

3. Descriptive naming

One way to aid in readability is with good naming. Look at these 2 functions. Both implement the same logic. However, if we believe 'readability counts', then the `is_boiling` function is a better option. The `is_boiling` function describes what it's doing thanks to descriptive naming. This is known as self-documenting code. However, remember that under-commented is a bigger issue than over-commented code; if you're ever in doubt, add comments. A warning about descriptive names. It's possible to go too far. Modern IDEs have auto-complete so it might not seem like a burden to type long names, but long names can make code hard to read, as you see here. Note, even if your code is self-documenting, your users will appreciate the ability to call help and see all the good information a docstring can provide.

Keep it simple

The Zen of Python, by Tim Peters (abridged)

Simple is better than complex.
Complex is better than complicated.

4. Keep it simple
Another way to keep your project readable is by writing in simple maintainable units of code. Let's take a look at an example by writing a function to make a pizza.



Making a pizza - complex

```
def make_pizza(ingredients):  
    # Make dough  
    dough = mix(ingredients['yeast'],  
                ingredients['flour'],  
                ingredients['water'],  
                ingredients['salt'],  
                ingredients['shortening'])  
  
    kneaded_dough = knead(dough)  
    risen_dough = prove(kneaded_dough)  
  
    # Make sauce  
    sauce_base = sautee(ingredients['onion'],  
                        ingredients['garlic'],  
                        ingredients['olive oil'])  
  
    sauce_mixture = combine(sauce_base,  
                            ingredients['tomato_paste'],  
                            ingredients['water'],  
                            ingredients['spices'])  
  
    sauce = simmer(sauce_mixture)  
    ...
```

5. Making a pizza - complex

Take a look at this code to make a pizza. Notice that the full definition doesn't fit on the slide. Although we're working with a small screen, not being able to fit your function on the screen is a sign that it maybe should be refactored. A less obvious issue is that we have a few different processes happening that could stand on their own. We're making dough and then making a

Making a pizza - simple

```
def make_pizza(ingredients):  
    dough = make_dough(ingredients)  
    sauce = make_sauce(ingredients)  
    assembled_pizza = assemble_pizza(dough, sauce, ingredients)  
  
    return bake(assembled_pizza)
```

6. Making a pizza - simple

Take a second to look at this refactored function. It's easier to see what's happening, in both a functional sense and a literal one since we were able to fully fit it on the screen. By breaking processes out into their own descriptively named functions we can see the high-level process of making a pizza at a glance. An additional benefit of defining these smaller functions is that we now have more modular code that we can easily plug into different recipes that might call for our homemade marinara. These also make writing tests easier, which we'll soon be covering.

When to refactor

Poor naming

```
def check(x, y=100):  
    return x >= y
```

Descriptive naming

```
def is_boiling(temp, boiling_point=100):  
    return temp >= boiling_point
```

Going overboard

```
def check_if_temperature_is_above_boiling_point(  
    temperature_to_check,  
    celsius_water_boiling_point=100):  
    return temperature_to_check >= celsius_water_boiling_point
```

7. When to refactor

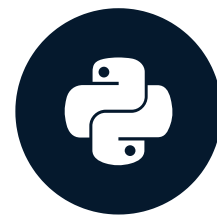
Our rewrites came about thanks to a couple warning signs. They were that our function was a bit long and we had separate processes happening. You should strive for your functions to accomplish one and only one thing. In our pizza example, we were using comments as 'section headers' to denote different processes; if you're ever doing that then it's a good bet that your code should be split into smaller functions. Another warning sign that your function is doing too much is if it's hard to think of a good meaningful name for it. Unfortunately, you might not notice your code is hard to read until trying to read it the next day. Just strive to do your best, and to repeat once more: document your code with comments and docstrings.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Unit testing

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON



Adam Spannbauer

Machine Learning Engineer at Eastman

Why testing?

- Confirm code is working as intended
- Ensure changes in one function don't break another
- Protect against changes in a dependency

Testing in Python

- `doctest`
- `pytest`



pytest

Using doctest

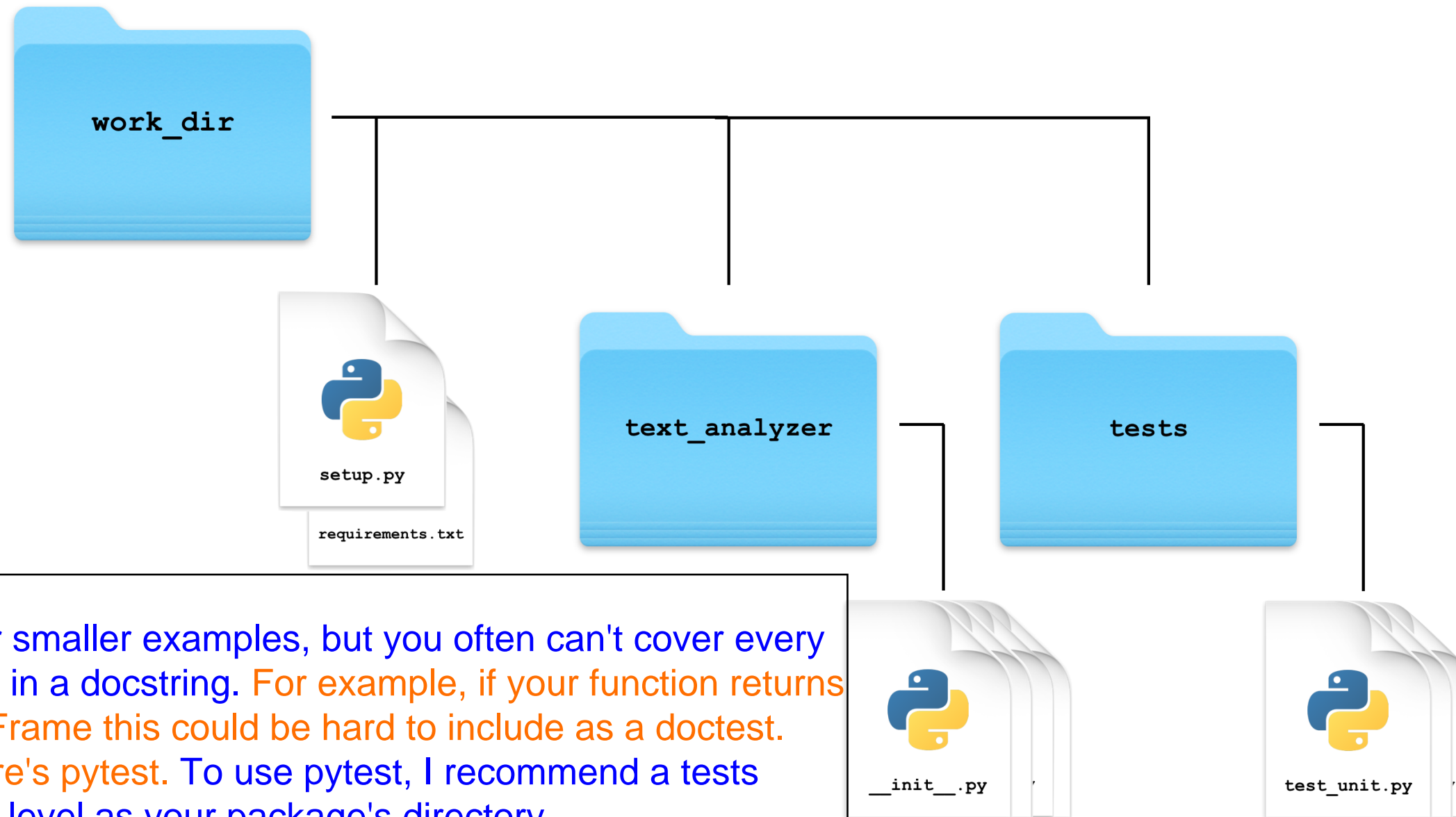
```
def square(x):  
    """Square the number x  
  
    :param x: number to square  
    :return: x squared  
  
    >>> square(3)  
    9  
    """  
    return x ** x  
  
import doctest  
doctest.testmod()
```

4. Using doctest

Thanks to writing informative docstrings with examples included you've already written tests that can be run with doctest! Let's look at an example with this square function. To use doctest we need to import it and run the testmod command to test our module's examples. When we run the tests we see that it failed. Looks like there was a typo in the docstring that was caught thanks to doctest. If there were no failed tests the output from testmod would be blank.

```
Failed example:  
    square(3)  
Expected:  
    9  
Got:  
    27
```

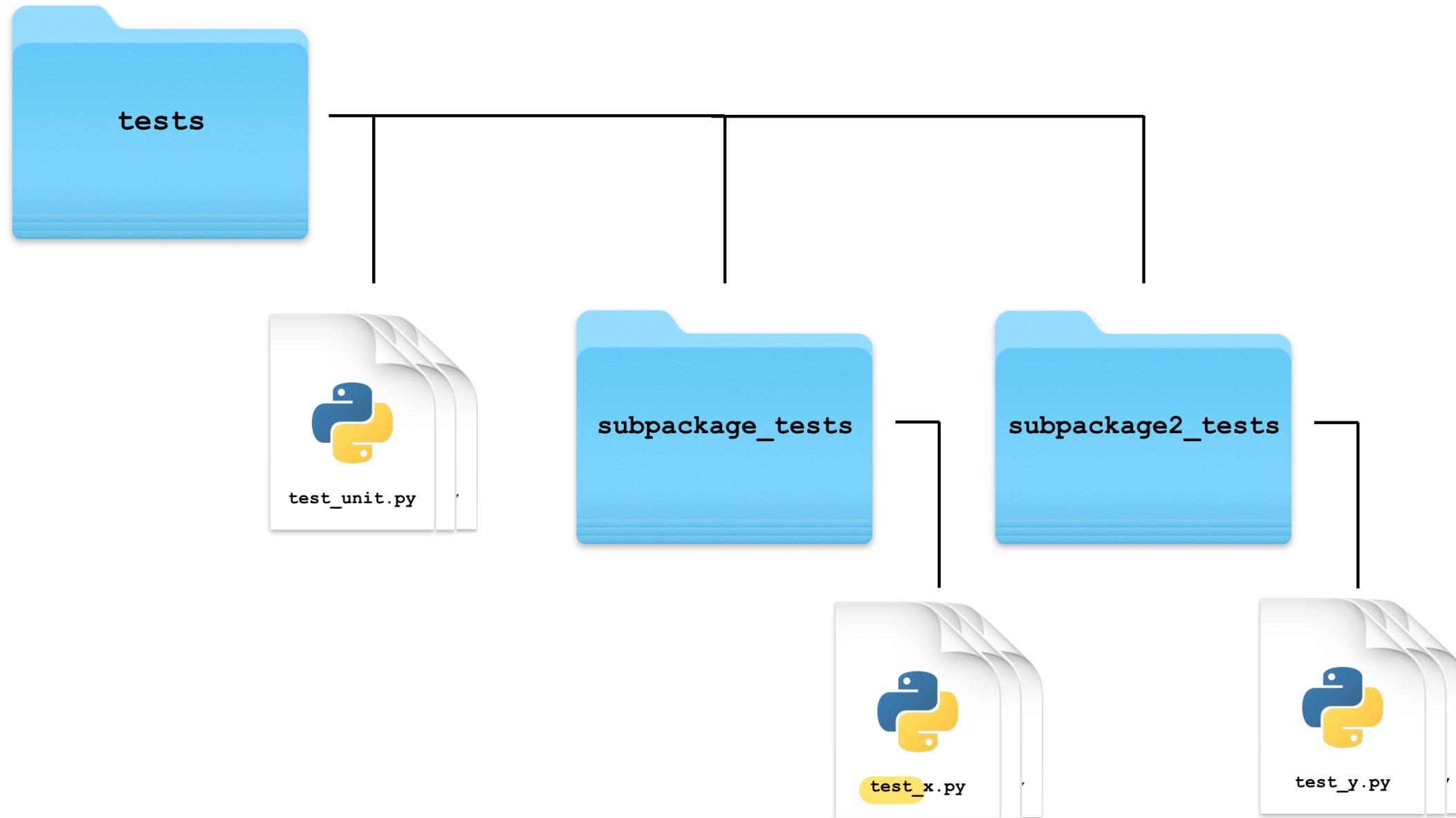
pytest structure



5. pytest structure

doctests are great for smaller examples, but you often can't cover every case you want to test in a docstring. For example, if your function returns a large pandas DataFrame this could be hard to include as a doctest. For larger cases, there's pytest. To use pytest, I recommend a tests directory at the same level as your package's directory.

pytest structure



Writing unit tests

working in `workdir/tests/test_document.py`

```
from text_analyzer import Document

# Test tokens attribute on Document object
def test_document_tokens():
    doc = Document('a e i o u')

    assert doc.tokens == ['a', 'e', 'i', 'o', 'u']

# Test edge case of blank document
def test_document_empty():
    doc = Document('')

    assert doc.tokens == []
    assert doc.word_counts == Counter()
```

7. Writing unit tests

Let's write some tests for our Document class. First, notice the test underscore prefix. **pytest searches for tests by first looking for files that start or end with the word test, and then pytest runs all the functions in these files who's name follow the same pattern.** Within our test method, we've then created an instance of document as our test case. Last, we run the actual test with the assert keyword. As long as the assertion is True, our test passes. When testing, it's also a good idea to test for the 'edge cases' that you can think of. For example, here we could test the expected attributes of a blank Document.

Writing unit tests

```
# Create 2 identical Document objects
doc_a = Document('a e i o u')
doc_b = Document('a e i o u')

# Check if objects are ==
print(doc_a == doc_b)
# Check if attributes are ==
print(doc_a.tokens == doc_b.tokens)
print(doc_a.word_counts == doc_b.word_counts)
```

8. Writing unit tests

Note that when testing class objects, it's not wise to compare two objects with double equals. Here we create two identical Document objects and test equality. Our test shows the objects are not the same. Instead, to compare objects we can compare attributes as we see here.

False

True

True

Running pytest

working with `terminal`

9. Running pytest

To run our tests we head to the terminal in our `work_dir` directory, run the command "pytest", and wait for our output. We see that all of our tests passed and our code is running as expected.

```
datacamp@server:~/work_dir $ pytest
```

```
collected 2 items
```

```
tests/test_document.py .. [100%]
```

```
===== 2 passed in 0.61 seconds =====
```

Running pytest

working with `terminal`

10. Running pytest

If we just wanted to run the tests in one file our command might look like this and we'd only get the results for that file's tests.

```
datacamp@server:~/work_dir $ pytest tests/test_document.py
```

```
collected 2 items
```

```
tests/test_document.py .. [100%]
```

```
===== 2 passed in 0.61 seconds =====
```

Failing tests

working with `terminal`

```
datacamp@server:~/work_dir $ pytest
```

```
collected 2 items
```

```
tests/test_document.py F.
```

```
===== FAILURES =====
```

```
_____ test_document_tokens _____
```

```
def test_document_tokens(): doc = Document('a e i o u')
```

```
assert doc.tokens == ['a', 'e', 'i', 'o']
```

```
E AssertionError: assert ['a', 'e', 'i', 'o', 'u'] == ['a', 'e', 'i', 'o']
```

```
E Left contains more items, first extra item: 'u'
```

```
E Use -v to get the full diff
```

```
tests/test_document.py:7: AssertionError
```

```
===== 1 failed in 0.57 seconds =====
```

11. Failing tests

So far all of our tests have passed. Let's look at some output of when things go wrong. In the output, we see exactly which test failed and even how it failed. This information can be a lifesaver when your projects grow and tracking down bugs becomes a more time-consuming process.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Documentation & testing in practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON



Adam Spannbauer

Machine Learning Engineer at Eastman

Documenting projects with Sphinx

text_analyzer

Navigation

Classes

Utility Functions

Quick search

Classes

`class text_analyzer.Document(text)`

Analyze text data

Parameters: `text` – text to analyze

Variables:

- `text` – Contains the text originally passed to the instance on creation
- `tokens` – Parsed list of words from `text`
- `word_counts` – `Counter` object containing counts of hashtags used in text

`plot_counts(attribute='word_counts', n_most_common=5)`

Plot most common elements of a `collections.Counter` instance attribute

Parameters:

- `attribute` – name of `Counter` attribute to use as object to plot
- `n_most_common` – number of elements to plot (using `Counter.most_common()`)

Returns: `None`; a plot is shown using `matplotlib`

```
>>> doc = Document("duck duck goose is fun")
>>> doc.plot_counts('word_counts', n_most_common=5)
```

2. Documenting projects with Sphinx

Sphinx was mentioned earlier as a tool that transforms docstrings into beautiful documentation. Here we see an example of what your `Document` class might look like as an HTML page generated by Sphinx. All of this benefit just from writing complete docstrings! If you're managing your project with GitHub or GitLab then you can even host your Sphinx-built documentation for free using the products' services. That way users can search the web to find your helpful documentation.

Documenting classes

```
class Document:
    """Analyze text data

    :param text: text to analyze

    :ivar text: text originally passed to the instance on creation
    :ivar tokens: Parsed list of words from text
    :ivar word_counts: Counter containing counts of hashtags used in text
    """
    def __init__(self, text):
        ...
```

3. Documenting classes

You might have noticed that the example Sphinx page showed documentation for a class and included information on attributes... We document the parameters for the init method in the class docstring. This way users can see how to initialize a new instance by calling help on the class itself. Next, we document attributes using the 'ivar' keyword. This abbreviation stands for 'instance variable'. With this documentation in place, users can easily see how to use your class.

Continuous integration testing



DataCamp / text_analyzer  build failing

Current Branches Build History Pull Requests > Build #230

More options 

✖ new_feature update SocialMedia class

🔗 #230 failed

🔗 Commit 3080c4a 

🕒 Ran for 1 min 13 sec

🔗 Compare 43dc3ba...3080c4a 

📅 11 days ago

🔗 Branch new_feature 



 DataCamp

🔗 </> Python: 3.6

4. Continuous integration testing

By using pytest extensively you'll be able to have most, if not all, of your project's code being tested, but it can be a bit of a bother to have to always run the tests repeatedly from the command line. This is where tools like Travis CI come in. The CI stands for continuous integration, and it just means that you are continually adding new code to your project. When adding this new code, Travis can automatically run your tests for you and alert you if your changes broke something.

Continuous integration testing

 DataCamp / text_analyzer  build passing

Current Branches Build History Pull Requests > Build #231 More options

✓ new_feature fix bug in SocialMedia

Commit 09eb5e9

Compare 3080c4a...09eb5e9

Branch new_feature

DataCamp

Python: 3.6

#231 passed

Ran for 1 min 39 sec

11 days ago

5. Continuous integration testing

Then when you push a fix, Travis CI will run your tests again to double check that your fix was successful. Another great aspect of Travis CI and other CI testing tools is scheduled builds. By scheduling a build your tests can be run periodically even without you adding new code. Scheduled tests are a great way to catch bugs introduced by updates in one of your dependencies.

Links and additional tools

- **Sphinx** - Generate beautiful documentation
- **Travis CI** - Continuously test your code
- **GitHub** & **GitLab** - Host your projects with git
- **Codecov** - Discover where to improve your projects tests
- **Code Climate** - Analyze your code for improvements in readability

6. Links and additional tools

To close out, let's list out some links for the tools we've covered and some additional ones. First, here are links to the tools we just covered. Next, GitHub and GitLab are great for storing open source projects. They are based on the tool, 'git'. Codecov is a tool that lets you explore which parts of your code are being tested by your automatic tests; this is known as test coverage. By keeping your project's test coverage high you will be less prone to surprise bugs. Code Climate is a tool that will analyze your code's readability. It can point out when a function is getting too long or even if your code is a little confusing in spots. Fixing the issues it points out, will help keep your project more maintainable. Both Codecov and Code Climate can be integrated with Travis CI to run automatically on a schedule and with each addition of new code.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Final Thoughts

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON



Adam Spannbauer

Machine Learning Engineer at Eastman

Looking Back

- Modularity

```
def function()  
    ...
```

```
class Class:  
    ...
```



Looking Back

- Modularity
- Documentation

```
"""docstrings"""
```

```
# Comments
```



Looking Back

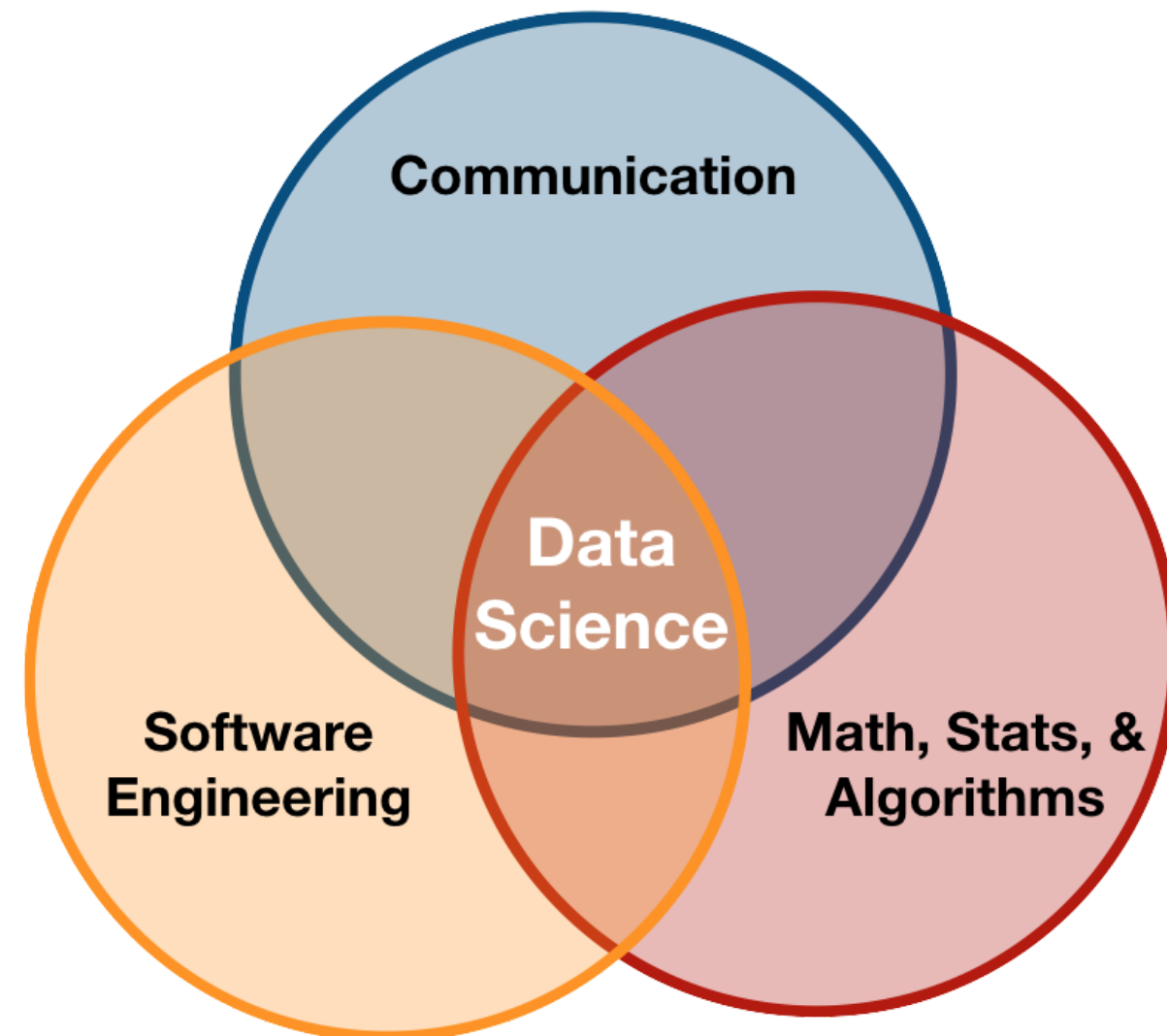
- Modularity
- Documentation
- Automated testing



```
def f(x):  
    """  
  
    >>> f(x)  
    expected output  
    """  
    ...
```



Data Science & Software Engineering



Good Luck!

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON