Módulo 4: Maintainability

Documentation

- Comments

```
# Square the number x
```

- Docstrings

```
    """Square the number x

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

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

Coments

Los **comentarios** se usan en una línea de código para documentar qué está haciendo y por qué.

Los comentarios no serán vistos por los usuarios hasta que entren al código fuente.

El objetivo de los comentarios es hacer el código más fácil de leer para uno mismo y para futuros colaboradores.

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

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

Los comentarios deberían explicar el **por qué**, no el qué

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
```

Es mejor un código con muchos comentarios a uno con pocos.

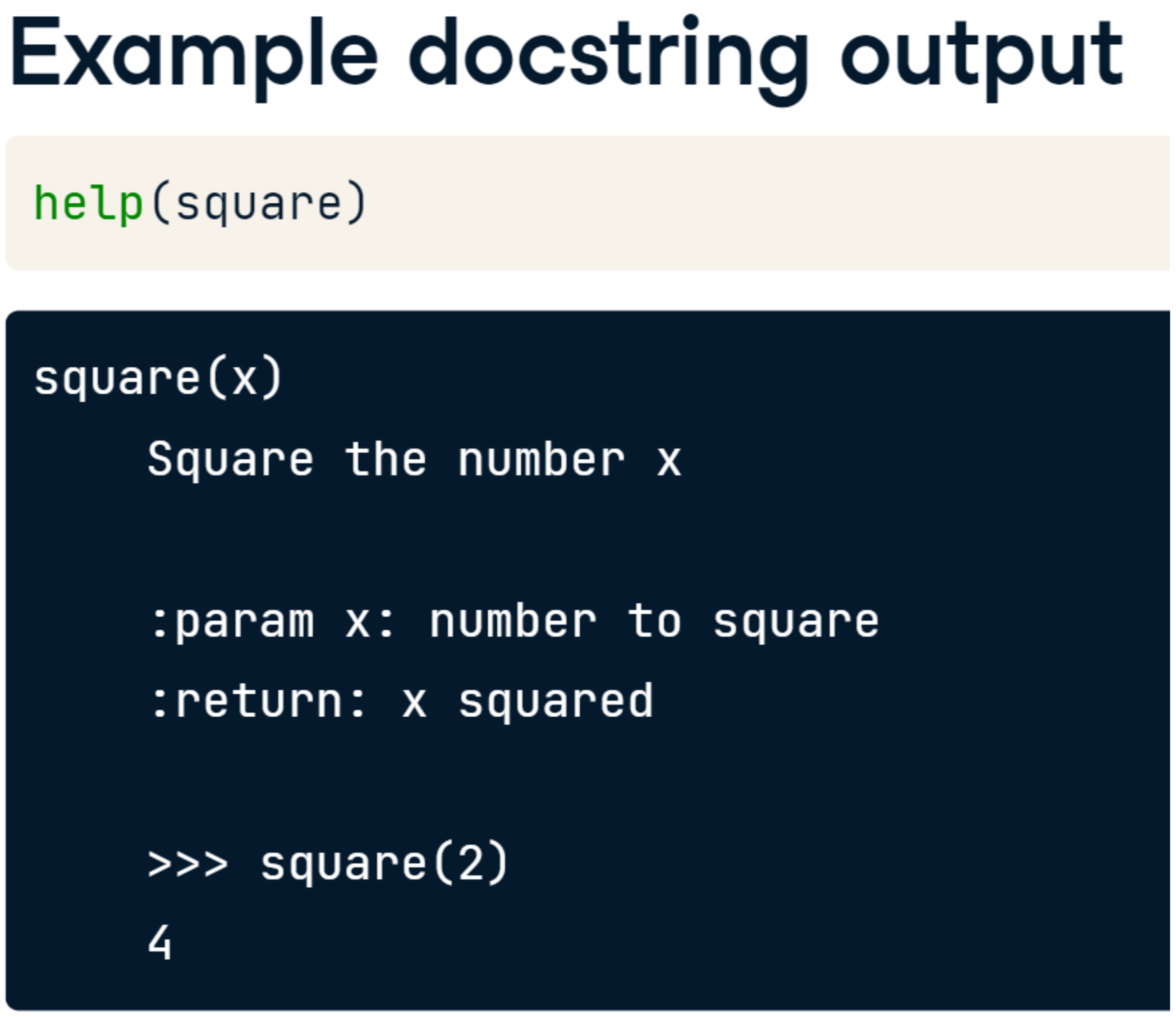
Docstrings

Los **docstrings** son documentación para los usuarios.

```python
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

```python
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
```



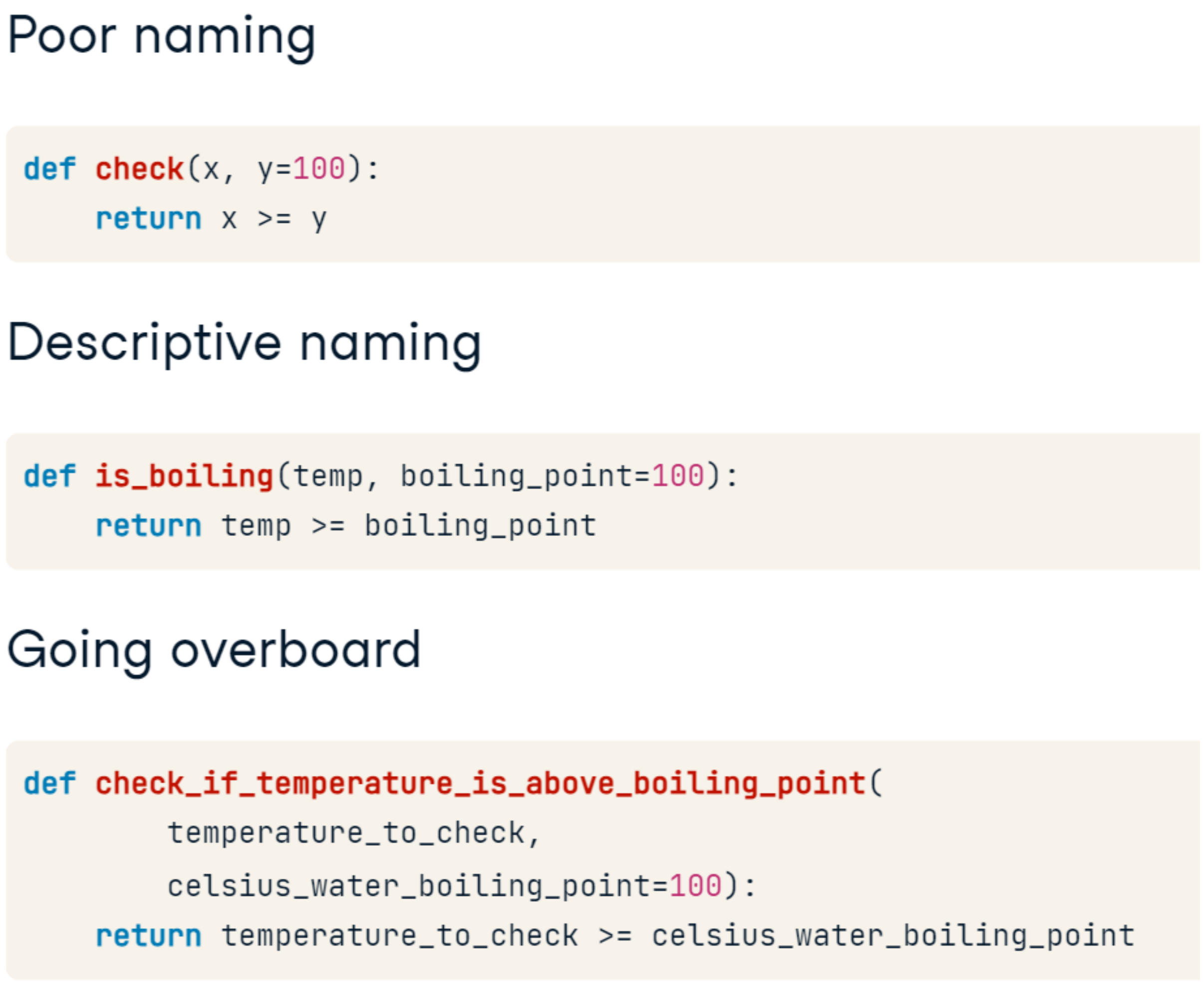
Example docstring output

```
help(square)
```

```
square(x)
    Square the number x

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

    >>> square(2)
    4
```

Readability counts

Descriptive naming



Poor naming

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

Descriptive naming

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

Going overboard

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

Keep it simple

Si el código no cabe en la pantalla, deberíamos pensar en refactorizarlo.

Making a pizza - complex

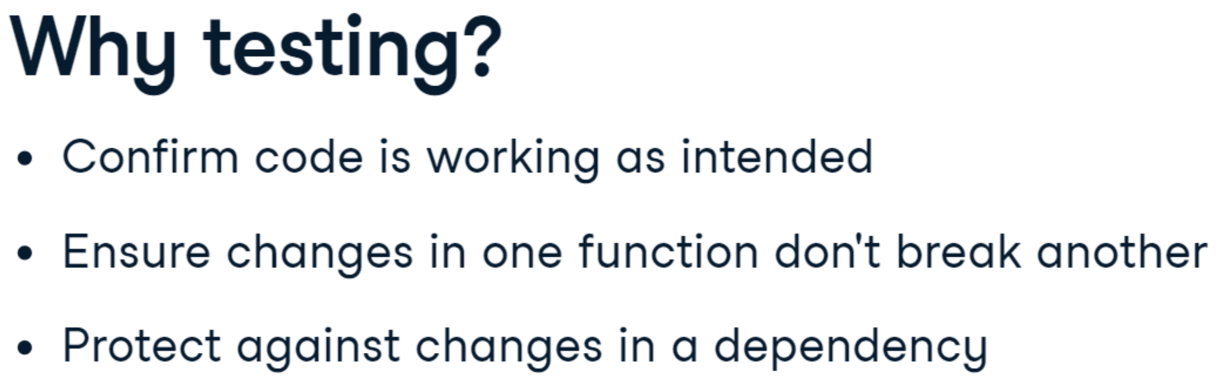
```python
def make_pizza(ingredients):
    # Make dough
    dough = mix(ingredients['yeast'],
                ingredients['flour'],
                ingredients['water'],
                ingredients['salt'],
                ingredients['shortening'])

    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)
```

Making a pizza - simple

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

    return bake(assembled_pizza)
```

Podemos partir los procesos que se realizan en la función en otras funciones. Cada función debe hacer una única tarea.

Unit testing



Why testing?

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

doctest

doctests prueba utilizando los docstrings.

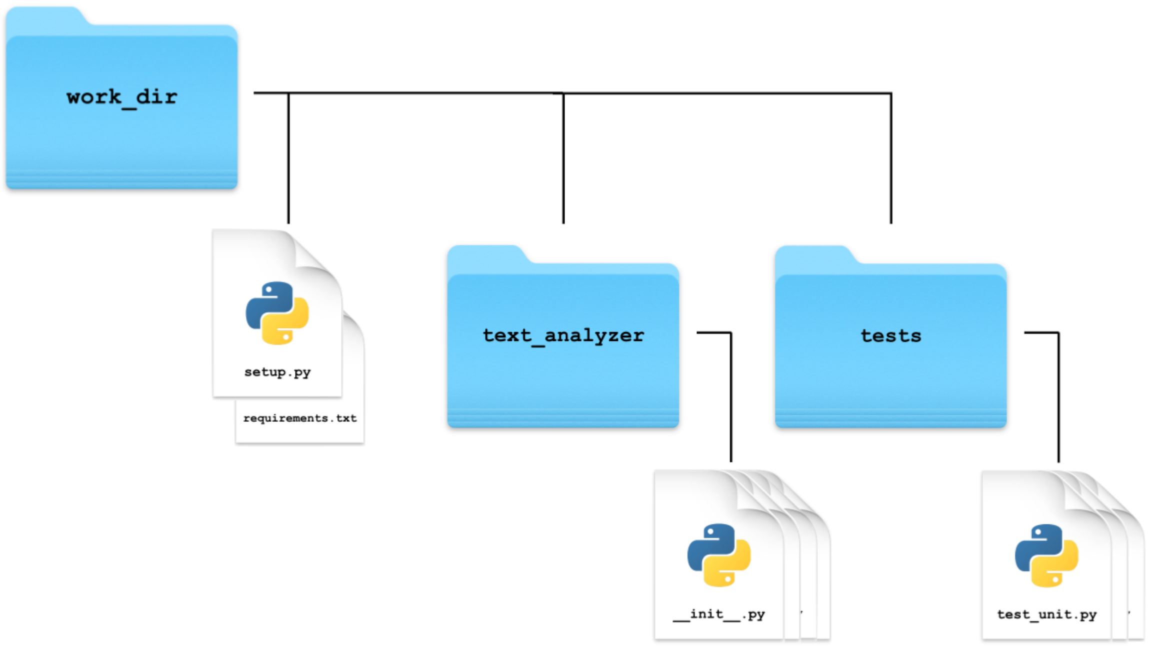
```python
def square(x):
    """Square the number x

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

    >>> square(3)
    9
    """
    return x ** 3


import doctest
doctest.testmod()
```

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

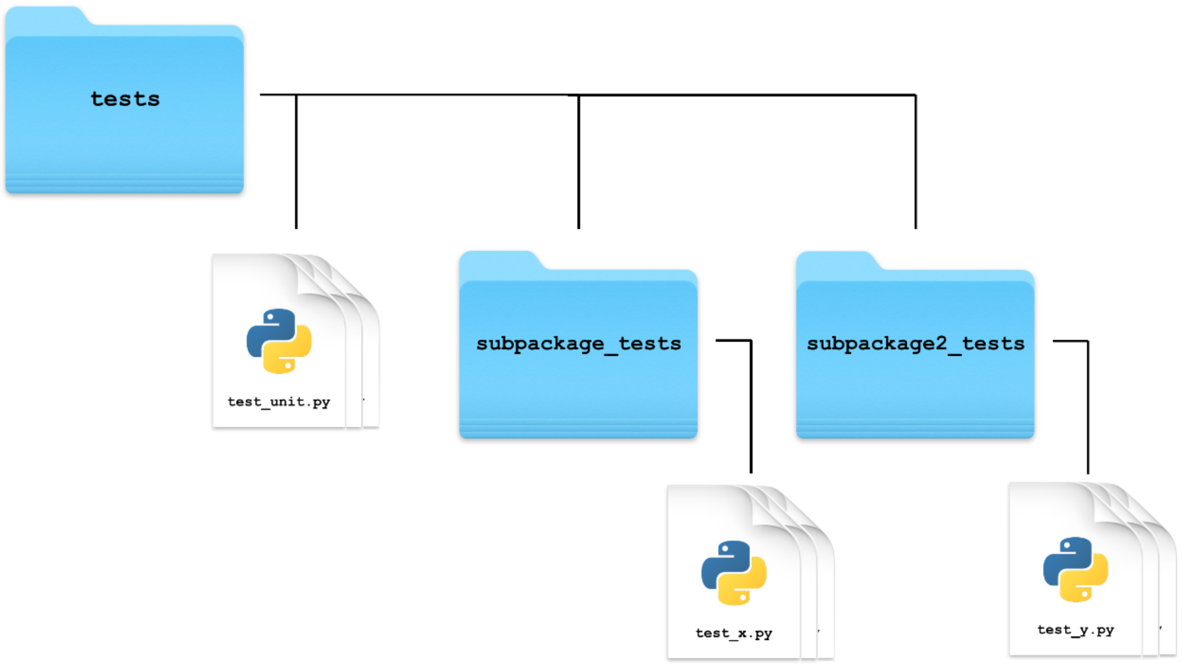
doctests es muy útil para proyectos pequeños.

Pytest

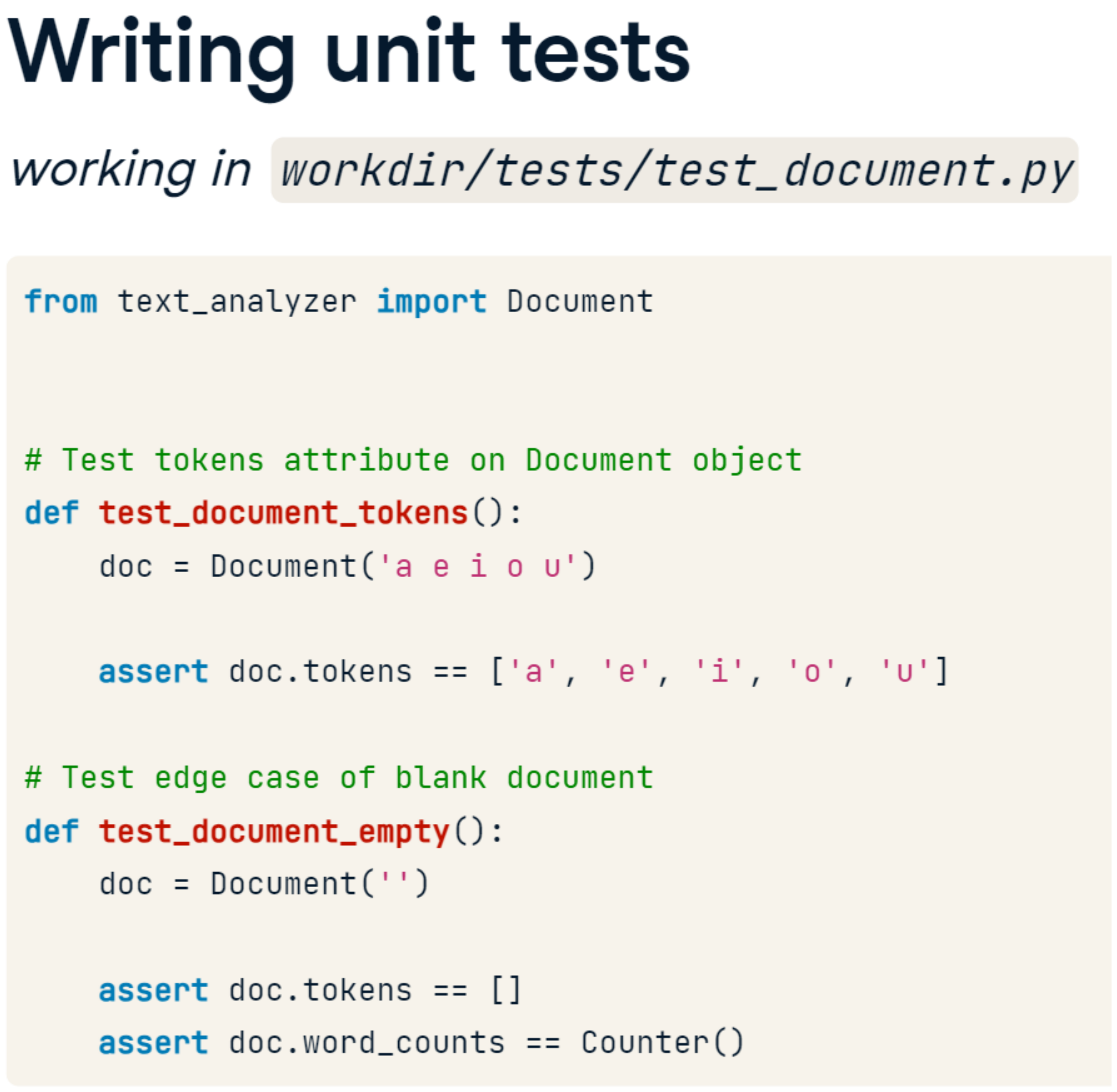
Se recomienda una estructura similar a la siguiente.



Se pueden crear test para subpaquetes.



Pytest busca los tests primero en archivos que empiecen o terminen con la palabra **test;** luego, ejecuta todas las funciones en esos archivos cuyo nombre siga el mismo patrón de comenzar o terminar con **test.**



Writing unit tests

working in `workdir/tests/test_document.py`

```python
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()
```

No es buena idea comparar dos instancias de una clase usando == ; es mejor comparar los atributos.

```python
# 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)
```

```
False
True
True
```

Running pytest

working with `terminal`

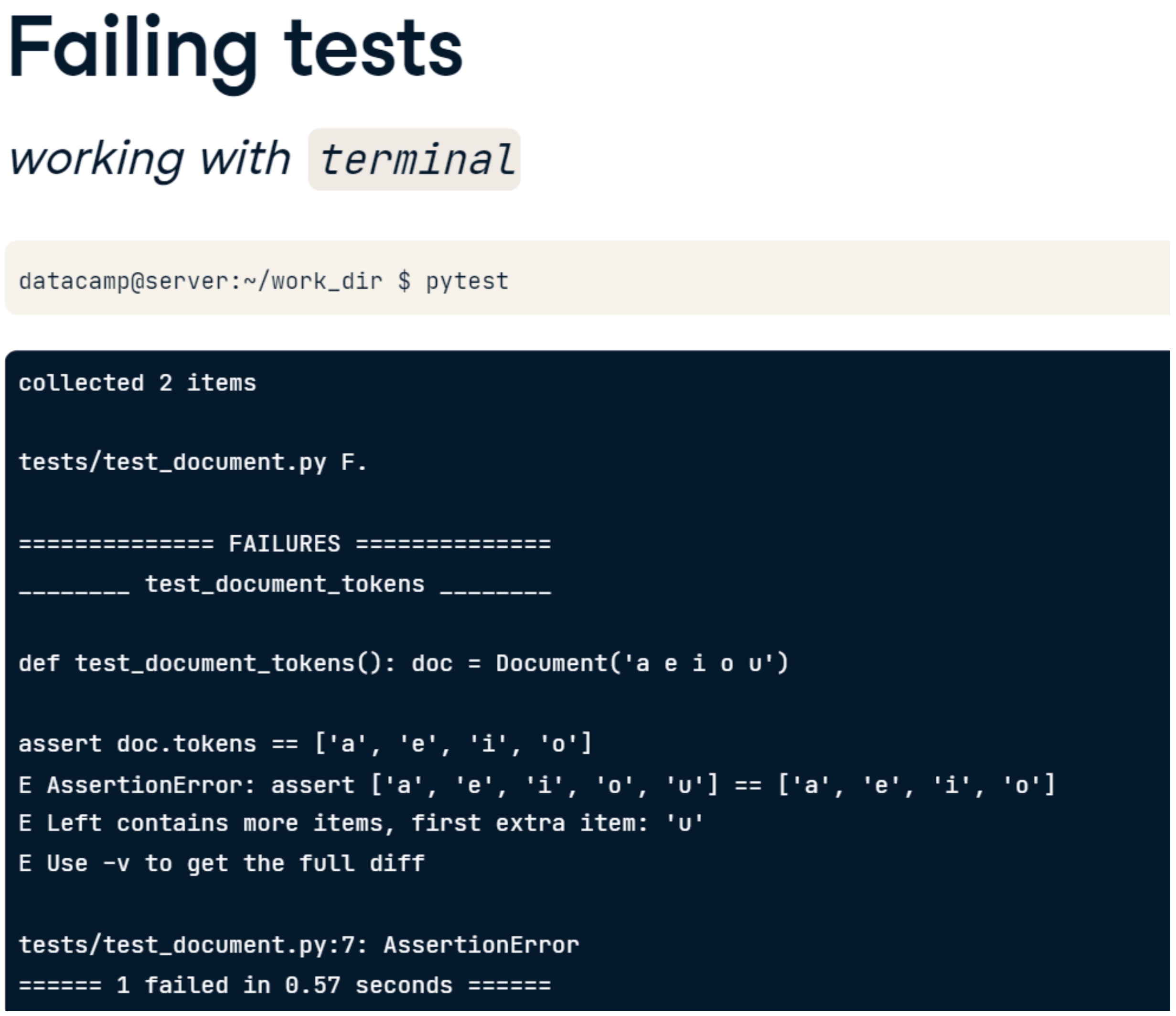
```
datacamp@server:~/work_dir $ pytest
```

```
collected 2 items

tests/test_document.py ..                              [100%]

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

working with `terminal`

```
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`

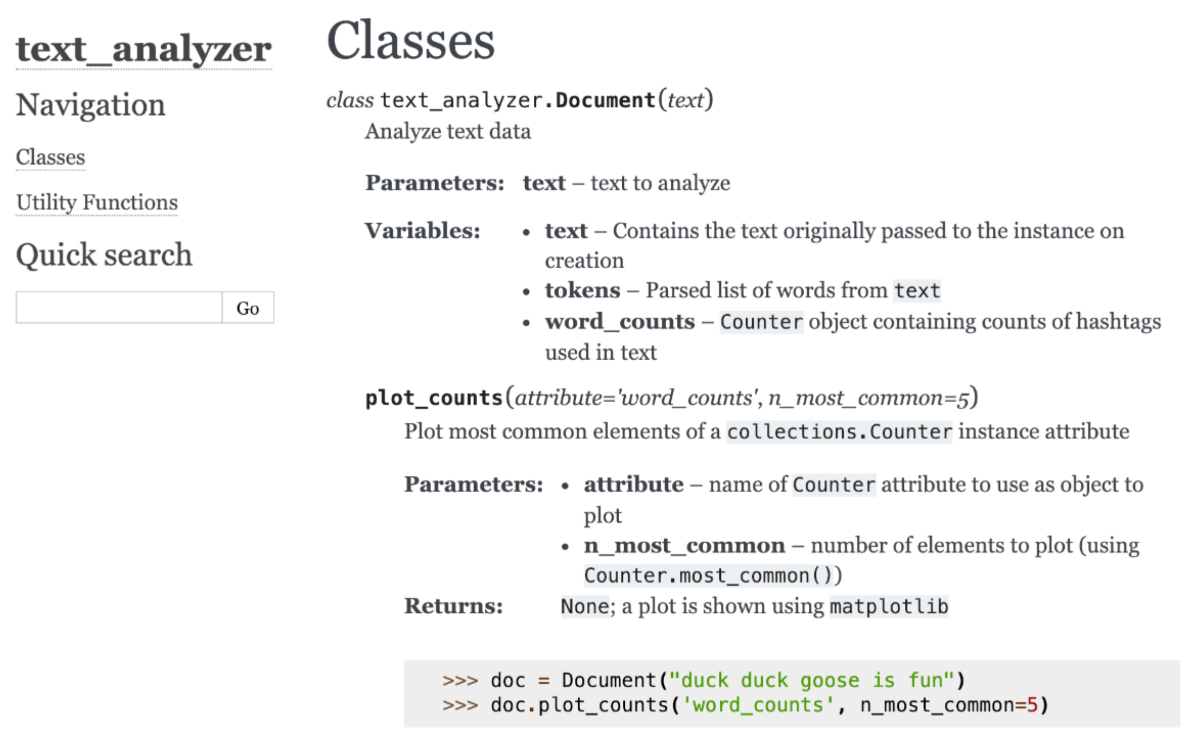
```
datacamp@server:~/work_dir $ pytest
```

```
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 ======
```

Documentation and testing in practice

Sphinx

Es una herramienta que transforma dosctrings en documentación.



text_analyzer

Navigation

Classes

Utility Functions

Quick search

Go

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)
```

Al documentar una clase, documentamos los parámetros del constructor en el docstring de la clase. **ivar** es una abreviación para instance variable.

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):
        ...
```