CONCEPT

MODULARITY

- Derive the module that contain as much as it should be
- Easy to maintain and reuse

DOCUMENTATION

- How to use your project
- Prevent confusion
- Prevent FRUSTRATION WHO IS TOUCHING THE CODE

AUTOMATED TESTING

- Save time over manual testing
- Find and fix more bugs
- Run tests anytime anywhere

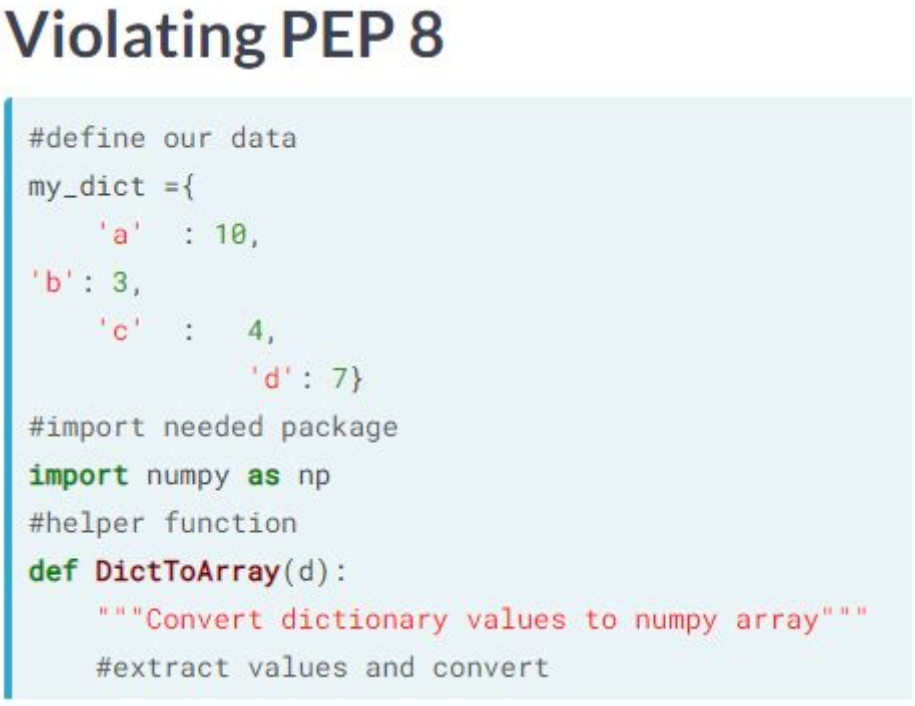
GUIDELINE

INTRO TO PACKAGES & DOCUMENTATION

- PYPI (python package index)
- help() function

CONVENTION AND PEP8

- PEP - PYTHON EXCHANGE PROTOCOL
- "**Code is read much more often than it is written"**
- VIOLATING PEP8 EXAMPLE



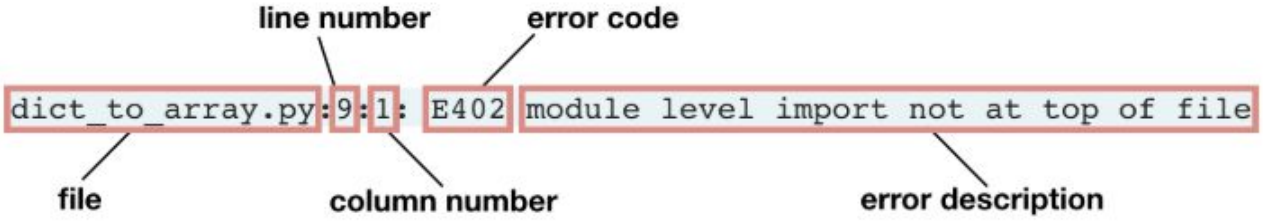
- FOLLOWING PEP8

Following PEP 8

```python
# Import needed package
import numpy as np

# Define our data
my_dict = {'a': 10, 'b': 3, 'c': 4, 'd': 7}


# Helper function
def dict_to_array(d):
    """Convert dictionary values to numpy array"""
    # Extract values and convert
    x = np.array(d.values())
```

PEP8 TOOLS - IDE, **pycodestyle package**

Output from pycodestyle



How to use pycodestyle to check your python script

```python
# Import needed package
import pycodestyle

# Create a StyleGuide instance
style_checker = pycodestyle.StyleGuide()

# Run PEP 8 check on multiple files
result = style_checker.check_files(['nay_pep8.py', 'yay_pep8.py'])

# Print result of PEP 8 style check
print(result.messages)

```

Some best practices from the PEP 8

PEP 8 in documentation

So far we've focused on how PEP 8 affects functional pieces of code. There are also rules to help make comments and documentation more readable. In this exercise, you'll be fixing various types of comments to be PEP 8 compliant.

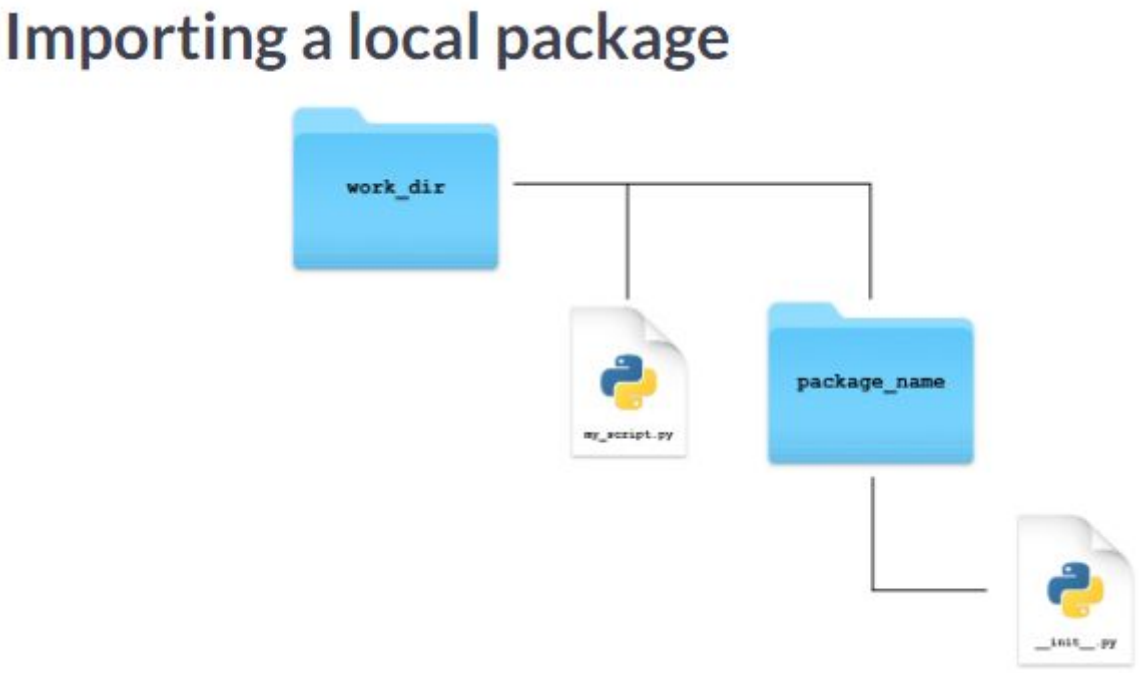
The result of a `pycodestyle` style check on the code can be seen below.

```
my_script.py:2:15: E261 at least two spaces before inline comment
my_script.py:5:16: E262 inline comment should start with '# '
my_script.py:11:1: E265 block comment should start with '# '
my_script.py:13:2: E114 indentation is not a multiple of four (comment)
my_script.py:13:2: E116 unexpected indentation (comment)
```

```
1  def print_phrase(phrase, polite=True, shout=False):
2      if polite:  # It's generally polite to say please
3          phrase = 'Please ' + phrase
4
5      if shout:  # All caps looks like a written shout
6          phrase = phrase.upper() + '!!'
7
8      print(phrase)
9
10
11 # Politely ask for help
12 print_phrase('help me', polite=True)
13 # Shout about a discovery
14 print_phrase('eureka', shout=True)
15
```

WRITING YOUR FIRST PACKAGE

"package_name" should be lower case and seperate with the underscore (_). To create package you have to create the __init__.py for let the python knows that this is the package. This structure is solid until python version 3.3 after that we still can import the package without the __init__.py structure.



Importing a local package

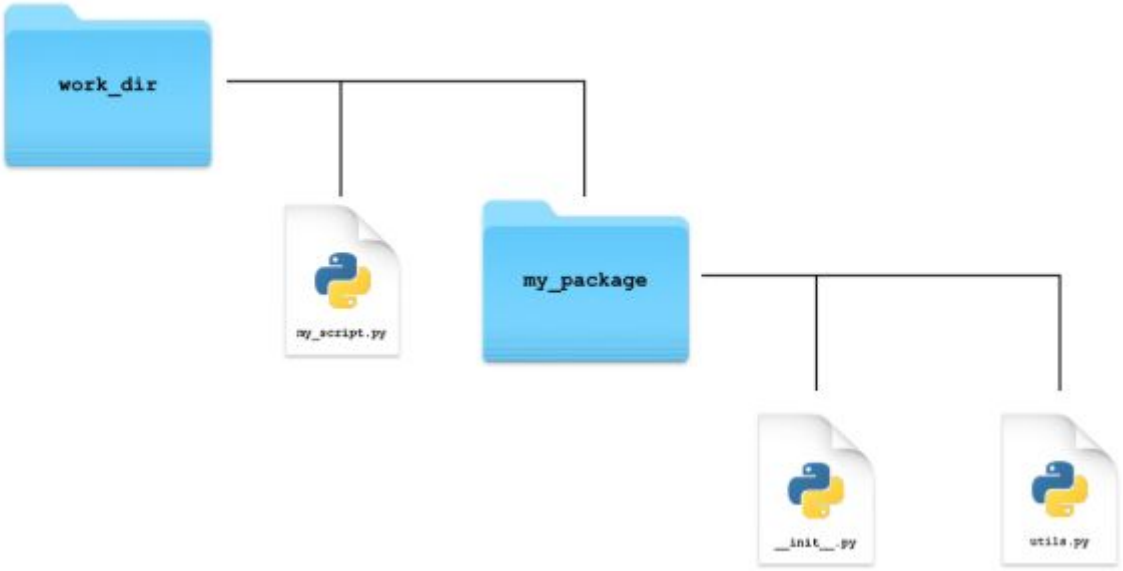
```
import my_package

help(my_package)
```

```
Help on package my_package:
NAME
    my_package

PACKAGE CONTENTS
FILE
    ~/work_dir/my_package/__init__.py
```

ADDING FUNCTIONALITY TO PACKAGES

The name convention of the function should be followed the PEP8 guideline

Package structure



We can add the function by the utils.py or add it in the __init__.py path in order to save time for importing it everytime you calls the package.

Adding functionality

working in `work_dir/my_package/utils.py`

```
def we_need_to_talk(break_up=False):
    """Helper for communicating with significant other"""
    if break_up:
        print("It's not you, it's me...")
    else:
        print('I <3 U!')
```

working in `work_dir/my_script.py`

```
# Import utils submodule
import my_package.utils
```

Importing functionality with __init__.py

working in `work_dir/my_package/__init__.py`

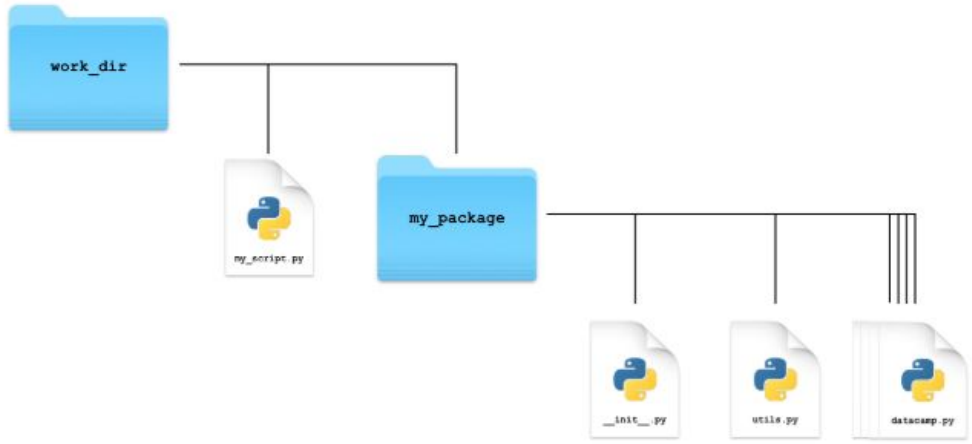
```
from .utils import we_need_to_talk
```

working in `work_dir/my_script.py`

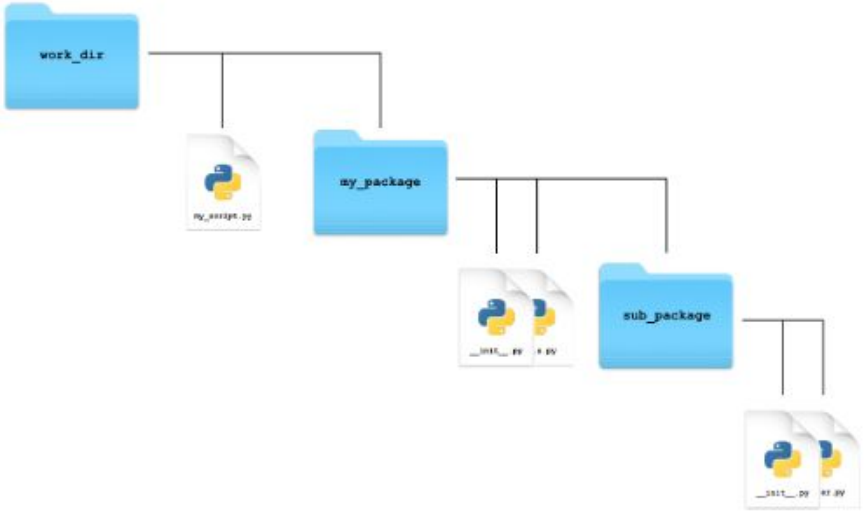
```
# Import custom package
import my_package

# Realize you're with your soulmate
my_package.we_need_to_talk(break_up=False)
```

When you extends the scope of package structure, you should consider about how easy the accessibles is, thus we recommend that you should have a central module like utils as above example and then you can use dot (.) to access the submodul inside the central module again.

Extending package structure



Extending package structure



For the relative syntax the . in front of the function.py is a must for importing it in __init.py__

Question

You just wrote two functions for your package in the file `counter_utils.py` named `plot_counter` & `sum_counters`. Which of the following lines would correctly import these functions in `__init__.py` using relative import syntax?

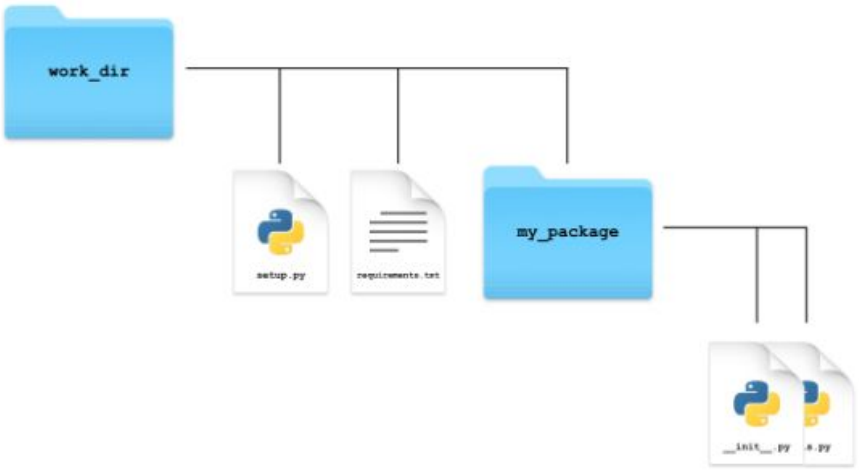
Possible Answers

- `from counter_utils import plot_counter, sum_counters`
- `from .counter_utils import plot_counter, sum_counters`
- `from . import plot_counter, sum_counters`
- `from .counter_utils import plot_counter & sum_counters`

MAKING YOUR PACKAGE PORTABLE

Create setup.py and requirements.txt. We have to put it in the work_dir.

Portable package structure



The content inside is as following.

Contents of requirements.txt

working in `work_dir/requirements.txt`

```
# Needed packages/versions
matplotlib
numpy==1.15.4
pycodestyle>=2.4.0
```

working with `terminal`

```
datacamp@server:~$ pip install -r requirements.txt
```

For example

```
requirements = """
matplotlib>=3.0.0
numpy==1.15.4
pandas<=0.22.0
pycodestyle
"""
```

The content of setup.py as following (we will use the package setup for help in creating package)

Contents of setup.py

```
from setuptools import setup

setup(name='my_package',
      version='0.0.1',
      description='An example package for DataCamp.',
      author='Adam Spannbauer',
      author_email='spannbaueradam@gmail.com',
      packages=['my_package'],
      install_requires=['matplotlib',
                        'numpy==1.15.4',
                        'pycodestyle>=2.4.0'])
```

If you use the install_requires option in setup.py the requirements.txt should be change as following

install_requires vs requirements.txt

working in `work_dir/requirements.txt`

```
# Specify where to install requirements from
--index-url https://pypi.python.org/simple/

# Needed packages/versions
matplotlib
numpy==1.15.4
pycodestyle>=2.4.0
```

Example of setup.py

```
# Import needed function from setuptools
from setuptools import setup

# Create proper setup to be used by pip
setup(name='text_analyzer',
      version='0.0.1',
      description='Perform and visualize a text anaylsis.',
      author='pat',
      packages=['text_analyzer'])
```

Then we can install our package by pip install .
(from the working directory)

ADDING CLASSES TO PACKAGE

How to use classes to strengthen your package

Anatomy of a class

working in `work_dir/my_package/my_class.py`

```
# Define a minimal class with an attribute
class MyClass:
    """A minimal example class

    :param value: value to set as the ``attribute`` attribute
    :ivar attribute: contains the contents of ``value`` passed in init
    """


    # Method to create a new instance of MyClass
    def __init__(self, value):
```

Using a class in a package

working in `work_dir/my_package/__init__.py`

```
from .my_class import MyClass
```

working in `work_dir/my_script.py`

```
import my_package

# Create instance of MyClass
my_instance = my_package.MyClass(value='class attribute value')

# Print out class attribute value
```

The self convention is to refer the instance even though we don't know how the use will name it. User doesn't have to pass any argument to the self argument. This is done behind the scene by python.

In the function body, we can use self to access of define attributes.

Question

You just completed writing the `Document` class for your package in `document.py`. Which of the following lines would correctly import this class in `__init__.py` using relative import syntax?

Possible Answers

- `from document import Document`
- `from . import Document`
- `from .document import Document` (selected)
- `from .document import document`

Leveraging class

Adding `_tokenize()` method

```
# Import function to perform tokenization
from .token_utils import tokenize

class Document:
    def __init__(self, text, token_regex=r'[a-zA-z]+'):
        self.text = text
        self.tokens = self._tokenize()

    def _tokenize(self):
        return tokenize(self.text)
```

Revising `__init__`

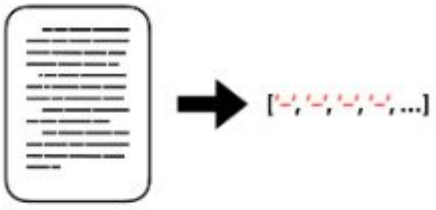
```
class Document:
    def __init__(self, text):
        self.text = text
        self.tokens = self._tokenize()

doc = Document('test doc')
print(doc.tokens)
```

```
['test', 'doc']
```

The reason they add the _tokenize method in the __init__ is to make it run at the first time when the user creates the object Document without the use have to think about it. Then this method doesn't have to be public to the user. Therefore, we can add the _function to the method as a private property.

Non-public methods

```
def __init__():
    ...
```

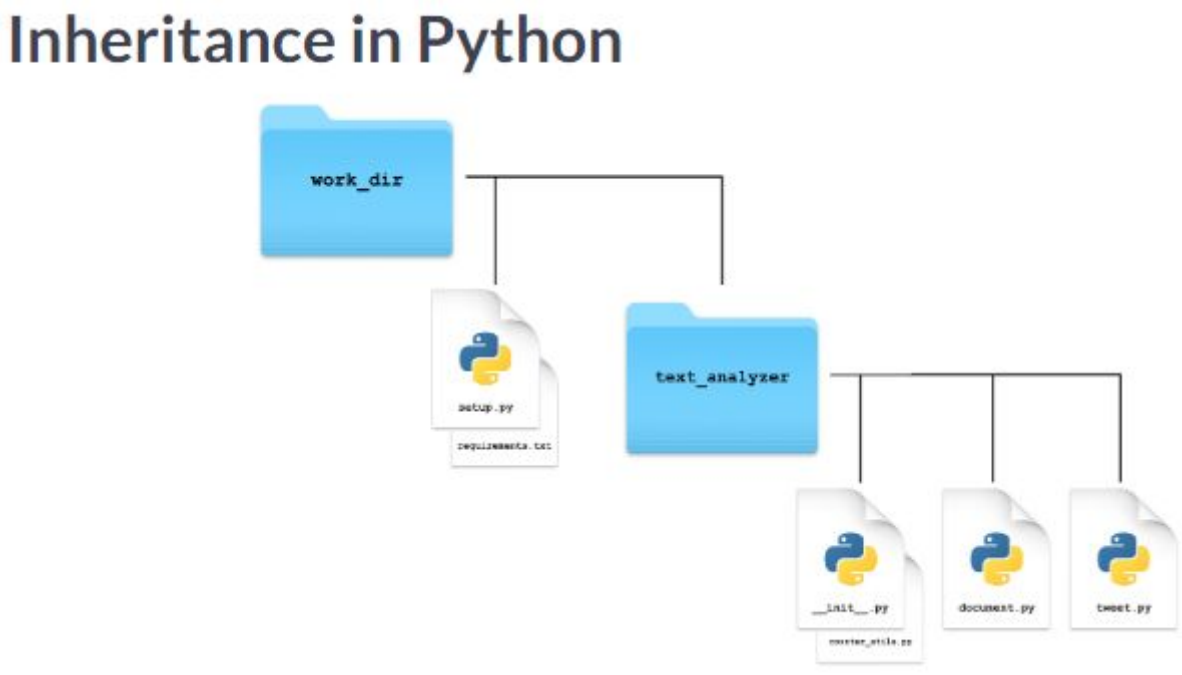


```python
class Document:
  def __init__(self, text):
    self.text = text
    # Tokenize the document with non-public tokenize method
    self.tokens = self._tokenize()
    # Perform word count with non-public count_words method
    self.word_counts = self._count_words()

  def _tokenize(self):
    return tokenize(self.text)

  # non-public method to tally document's word counts with Counter
  def _count_words(self):
    return Counter(self.tokens)

```

CLASS AND THE DRY PRINCIPLE

How to create a new class while preserving the context of the previous class as is. The DRY principle “DON’T REPEAT YOURSELF” fix it in 1 place and use it through the pipeline.

INHERITANCE in python




```python
# Define a SocialMedia class that is a child of the `Document class`
class SocialMedia(Document):
    def __init__(self, text):
        Document.__init__(self, text)

```

How to create a new class with inheritance

```
# Define a SocialMedia class that is a child of the `Document class`
class SocialMedia(Document):
    def __init__(self, text):
        Document.__init__(self,text)
        self.hashtag_counts = self._count_hashtags()
        self.mention_counts = self._count_mentions()

    def _count_hashtags(self):
        # Filter attribute so only words starting with '#' remain
        return filter_word_counts(self.word_counts, first_char='#')

    def _count_mentions(self):
        # Filter attribute so only words starting with '@' remain
        return filter_word_counts(self.word_counts, first_char='@')

```

How to use the chill class

```
# Import custom text_analyzer package
import text_analyzer

# Create a SocialMedia instance with datacamp_tweets
dc_tweets = text_analyzer.SocialMedia(text=datacamp_tweets)

# Print the top five most most mentioned users
print(dc_tweets.mention_counts.most_common(5))

# Plot the most used hashtags
text_analyzer.plot_counter(dc_tweets.word_counts)

```

Multilevel inheritance

Multilevel inheritance and super

```
class Parent:
    def __init__(self):
        print("I'm a parent!")


class Child(Parent):
    def __init__(self):
        Parent.__init__()
        print("I'm a child!")

class SuperChild(Parent):
    def __init__(self):
        super().__init__()
        print("I'm a super child!")
```

To keep the track of inherited attributes we can use function of help() to find the path when those class belong to so that we can see the structure cleary. dir() will help you in showing all the methods that class have so you might notice how this class comes.

The example of multiple inheritance.

```
1  # Define a Tweet class that inherits from SocialMedia
2  class Tweets(SocialMedia):
3      def __init__(self, text):
4          # Call parent's __init__ with super()
5          super.__init__(self,text)
6          # Define retweets attribute with non-public method
7          self.retweets = self._process_retweets()
8
9      def _process_retweets(self):
10         # Filter tweet text to only include retweets
11         retweet_text = filter_lines(self.text, first_chars='RT')
12         # Return retweet_text as a SocialMedia object
13         return SocialMedia(retweet_text)
14
```

DOCUMENTATION

Comments - use inline with the code

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!!
```

Docstrings - use to document function script

Documentation in Python

- Comments

```
# Square the number x
```

- Docstrings

```
"""Square the number x

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

>>> square(2)
4
```

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
```

You should comment why instead of what

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.

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
```

We can access the docstrings by using function help()

```
1  import re
2
3  def extract_0(text):
4      # match and extract dollar amounts from the text
5      return re.findall(r'\$\d+\.\d\d', text)
6
7  def extract_1(text):
8      # return all matches to regex pattern
9      return re.findall(r'\$\d+\.\d\d', text)
10
11 # Print the text
12 print(text)
13
14 # Print the results of the function with better commenting
15 print(extract_0(text))
```

The proper way to write a docstrings

```python
# Complete the function's docstring
def tokenize(text, regex=r'[a-zA-z]+'):
    """Split text into tokens using a regular expression

    :param text: text to be tokenized
    :param regex: regular expression used to match tokens using re.findall
    :return: a list of resulting tokens

    >>> tokenize('the rain in spain')
    ['the', 'rain', 'in', 'spain']
    """
    return re.findall(regex, text, flags=re.IGNORECASE)

# Print the docstring
help(tokenize)
```

Readability count

The Zen of python is the guideline for writing a python (read it by import this)

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
```

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)
    ...
```

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)
```

The example of readability counts

```
1  def polygon_perimeter(n_sides, side_len):
2      return n_sides * side_len
3
4  def polygon_apothem(n_sides, side_len):
5      denominator = 2 * math.tan(math.pi / n_sides)
6      return side_len / denominator
7
8  def polygon_area(n_sides, side_len):
9      perimeter = polygon_perimeter(n_sides, side_len)
10     apothem = polygon_apothem(n_sides, side_len)
11
12     return perimeter * apothem / 2
13
14 # Print the area of a hexagon with legs of size 10
15 print(polygon_area(n_sides=6, side_len=10))
```

Unit testing

To check that your script is working correctly
Ensure changes in one function don't break another
Protect against changes in dependency

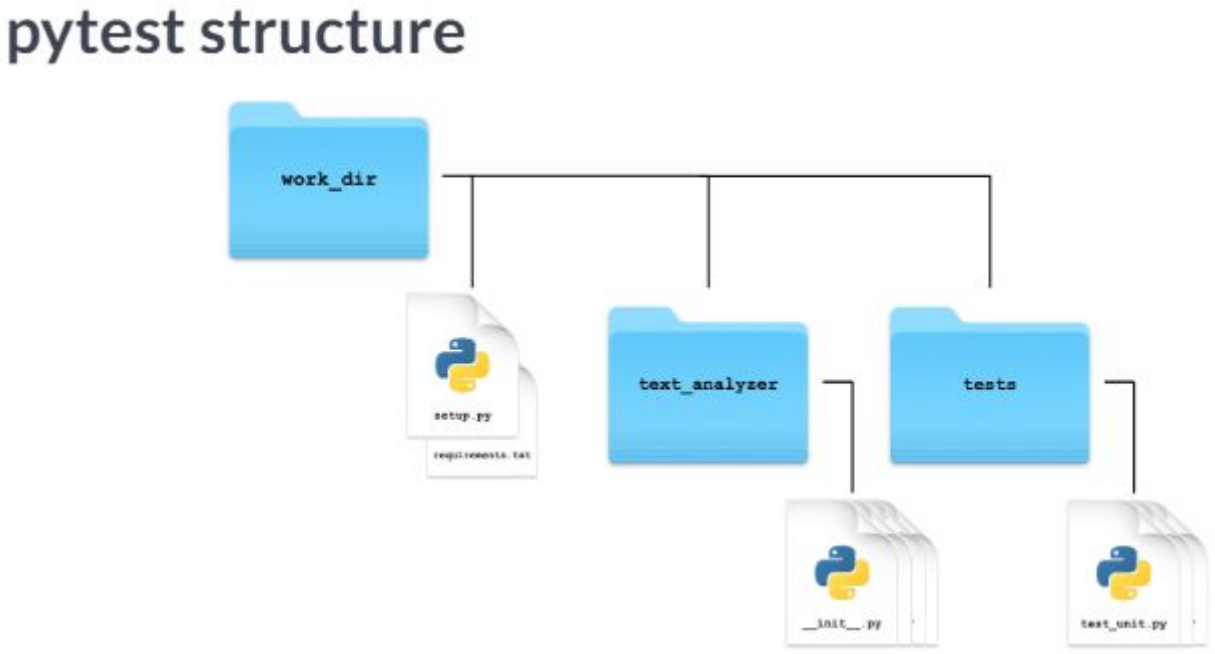
The 2 packages we can use for testing are
Doctest

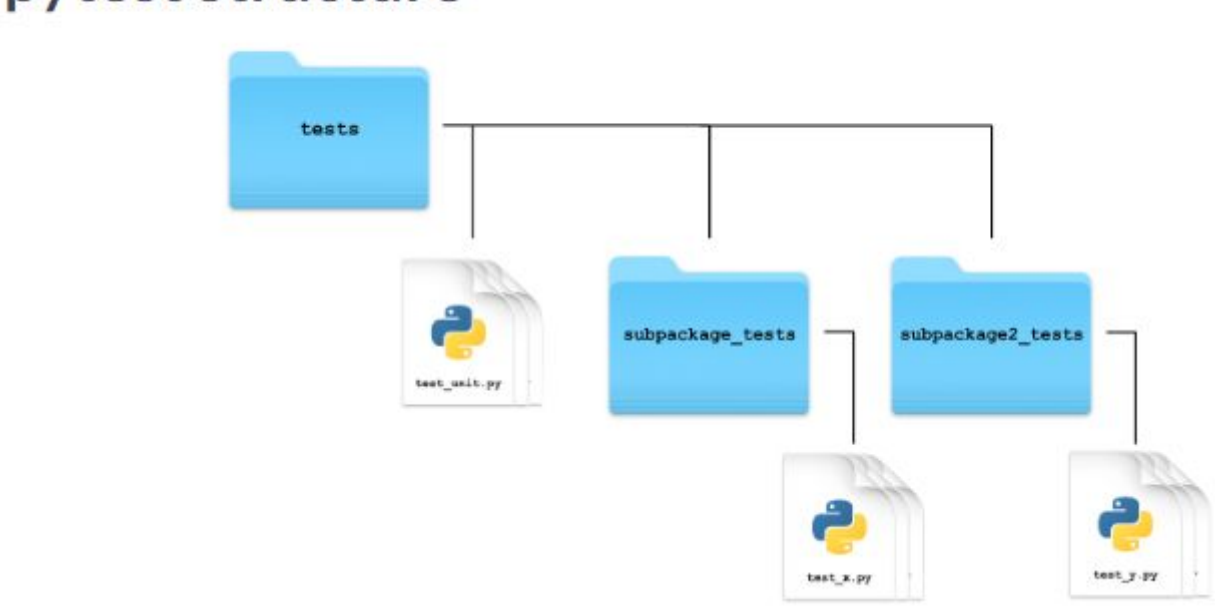
Using doctest

```
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()
```

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

Pytest - for the larger case





As long as the assertion is okay then it’s passed

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()
```

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']
```

We can't compare the 2 object directly, but we can compare the attribute of them by == operation

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)
```

```
False
True
True
```

To run the test

Running pytest

working with `terminal`

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

```
collected 2 items

tests/test_document.py ..

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

To running the test at only 1 file

Running pytest

working with `terminal`

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

```
collected 2 items

tests/test_document.py ..

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

```
def sum_counters(counters):
    """Aggregate collections.Counter objects by summing counts

    :param counters: list/tuple of counters to sum
    :return: aggregated counters with counts summed

    >>> d1 = text_analyzer.Document('1 2 fizz 4 buzz fizz 7 8')
    >>> d2 = text_analyzer.Document('fizz buzz 11 fizz 13 14')
    >>> sum_counters([d1.word_counts, d2.word_counts])
    Counter({'fizz': 4, 'buzz': 2})
    """
    return sum(counters, Counter())

doctest.testmod()
```

```python
from collections import Counter
from text_analyzer import SocialMedia

# Create an instance of SocialMedia for testing
test_post = 'learning #python & #rstats is awesome! thanks @datacamp!'
sm_post = SocialMedia(test_post)

# Test hashtag counts are created properly
def test_social_media_hashtags():
    expected_hashtag_counts = Counter({'#python': 1, '#rstats': 1})
    assert sm_post.hashtag_counts == expected_hashtag_counts

```

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 ======
```

Documenting projects with Sphinx

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()`)

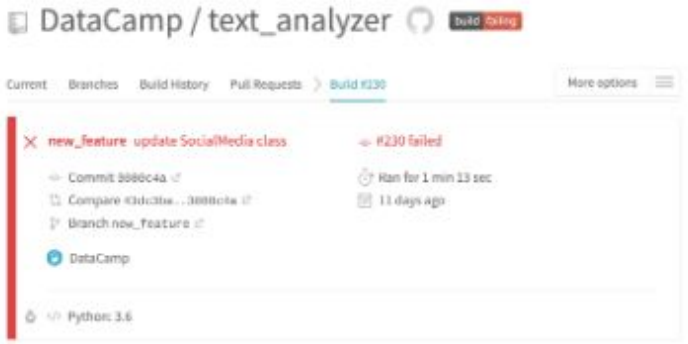
Returns: `None`; a plot is shown using `matplotlib`

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

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):
        ...
```

Continuous integration testing



Travis CI is the platform that will run the automated test for you once you update the code into the system.

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

```
from text_analyzer import Document

class SocialMedia(Document):
    """Analyze text data from social media

    :param text: social media text to analyze

    :ivar hashtag_counts: Counter object containing counts of hashtags used in text
    :ivar mention_counts: Counter object containing counts of @mentions used in text
    """
    def __init__(self, text):
        Document.__init__(self, text)
        self.hashtag_counts = self._count_hashtags()
        self.mention_counts = self._count_mentions()
```

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
    """
    ...
```