1. Equivalence Testing in Python

Run equivalence.py which is an implementation of equivalence partitioning. This test partitions integers [-3,5] into equivalence classes based on *lambda x, y: (x-y)%4 == 0.* In the output, you should be able to see how a set of objects to be partitioned are considered, and a function evaluates if the two objects are equivalent before printing the result. test_equivalence_partition() produces the following output: set([1, -3]) set([2, -2]) set([3, -1]) set([0, 4]) 0 : set([0, 4]) 1 : set([1, -3]) 2 : set([2, -2]) 3 : set([3, -1]) 4 : set([0, 4]) -2 : set([2, -2]) -3 : set([1, -3]) -1 : set([3, -1]). You should carry out further investigations on the code and experiment with it.

equivalence.py

Edit Find View Tools Education Help Run Project Index (static) Debug ESARMAD

equivalence.py

```python
# CODE SOURCE: https://stackoverflow.com/questions/38924421/is-there-a-standard-way-to-pa

def equivalence_partition(iterable, relation):
    """Partitions a set of objects into equivalence classes

    Args:
        iterable: collection of objects to be partitioned
        relation: equivalence relation. I.e. relation(o1,o2) evaluates to True
            if and only if o1 and o2 are equivalent

    Returns: classes, partitions
        classes: A sequence of sets. Each one is an equivalence class
        partitions: A dictionary mapping objects to equivalence classes
    """
    classes = []
    partitions = {}
    for o in iterable:  # for each object
        # find the class it is in
        found = False
        for c in classes:
            if relation(next(iter(c)), o):  # is it equivalent to this class?
                c.add(o)
                partitions[o] = c
                found = True
                break
        if not found:  # it is in a new class
            classes.append(set([o]))
            partitions[o] = classes[-1]
    return classes, partitions


def equivalence_enumeration(iterable, relation):
    """Partitions a set of objects into equivalence classes

    Same as equivalence_partition() but also numbers the classes.

    Args:
        iterable: collection of objects to be partitioned
        relation: equivalence relation. I.e. relation(o1,o2) evaluates to True
            if and only if o1 and o2 are equivalent
```

```python

    Returns: classes, partitions, ids
        classes: A sequence of sets. Each one is an equivalence class
        partitions: A dictionary mapping objects to equivalence classes
        ids: A dictionary mapping objects to the indices of their equivalence classes
    """
    classes, partitions = equivalence_partition(iterable, relation)
    ids = {}
    for i, c in enumerate(classes):
        for o in c:
            ids[o] = i
    return classes, partitions, ids


def check_equivalence_partition(classes, partitions, relation):
    """Checks that a partition is consistent under the relationship"""
    for o, c in partitions.items():
        for _c in classes:
            assert (o in _c) ^ (not _c is c)
    for c1 in classes:
        for o1 in c1:
            for c2 in classes:
                for o2 in c2:
                    assert (c1 is c2) ^ (not relation(o1, o2))


def test_equivalence_partition():
    relation = lambda x, y: (x - y) % 4 == 0
    classes, partitions = equivalence_partition(
        range(-3, 5),
        relation
    )
    check_equivalence_partition(classes, partitions, relation)
    for c in classes: print(c)
    for o, c in partitions.items(): print(o, ':', c)


if __name__ == '__main__':
    test_equivalence_partition()
```

Equivalence Partitioning is a testing technique that can be used when the data is too much to test. So, the data is divided into partitions and just samples of each partition will be tested.

The code in ‘equivalence.py’:

- First function (equivalence_partition):
  - Makes a class using two objects from iterables. (Two objects have special style which has been mentioned in ‘relation’)
  - Appends every class to a dictionary called ‘partitions’.
  - To do the above steps, two loops have been used.
  - Returns created classes and partition

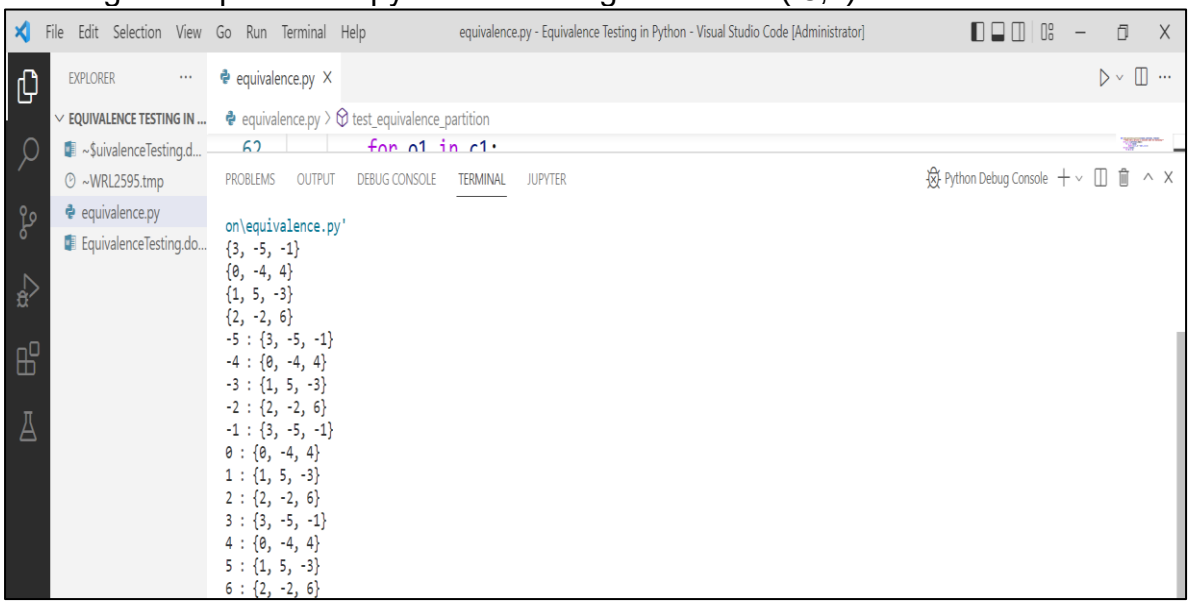
  This function has been called in two other functions (equivalence_enumeration and test_equivalence_partition).

- Second function (equivalence_enumeration):
  - Represents the iterables using objects that acts like counters
  - Creates a dictionary called ‘ids’ and stuff it with the indices of the classes using two loops.
  - Returns created classes, partition, and ids

- Third function (check_equivalence_partition):
  - Checks for objects in classes and classes in the ‘partitions’ using assertions.

- Fourth function (test_equivalence_partition):
  - Mentions the relationships between two objects using the formula (lambda x, y: (x - y) % 4 == 0)
  - Displays the range of numbers for partitioning
  - Prints every class
  - Prints every object in every class and every class in the ‘partitions’ following this pattern: ‘object: class’

Running the ‘equivalence.py’ with new ranges such as (-5,7):



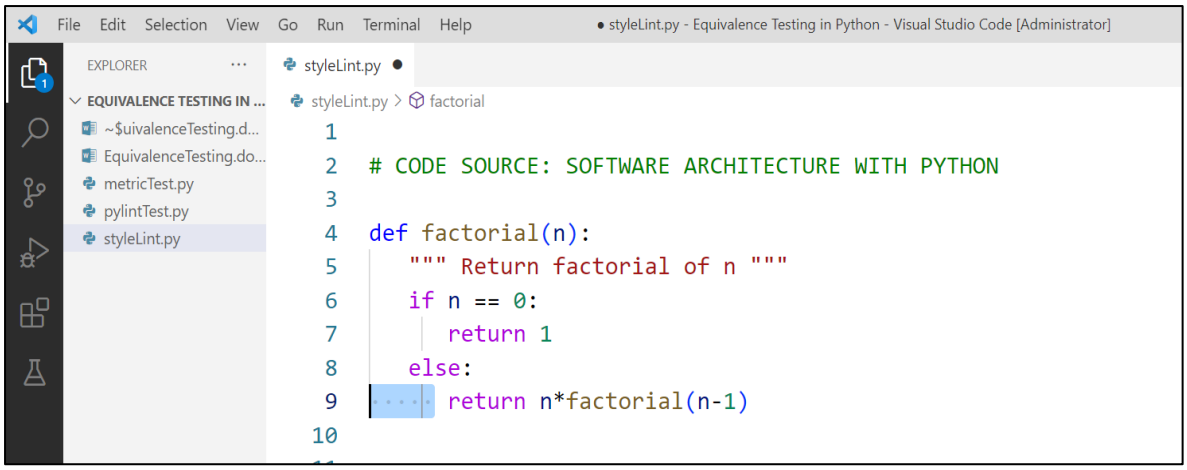
2. Testing with Python

Exploring Linters to Support Testing in Python: Question 1

Run styleLint.py

- **What happens when the code is run?** There is an Indentation error.
- **Can you modify this code for a more favourable outcome?** Yes
- **What amendments have you made to the code?** I indented every line.

styleLint.py



3. Testing with Python

Exploring Linters to Support Testing in Python: Question 2

pip install pylint

Run pylint on pylintTest.py

- **Review each of the code errors returned.**

  The errors are as follows:

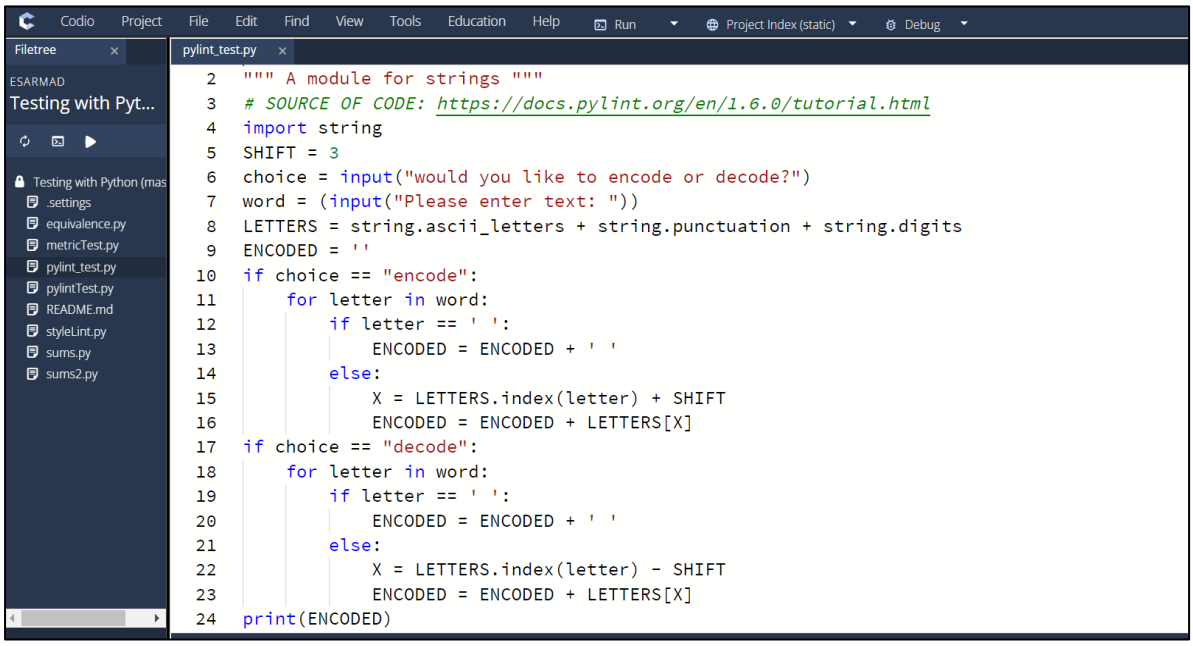
  - Module name "styleLint" doesn't conform to snake_case naming
  - Missing module docstring pylint (missing-module-docstring)
  - Constant name "shift" doesn't conform to UPPER_CASE naming style pylint (invalid-name)
  - Bad indentation. Found 2 spaces, expected 4 pylint (bad-indentation)
  - Trailing whitespace pylint (trailing-whitespace)
  - print(*values: object, sep: str | None = ..., end: str | None = ..., file: SupportsWrite[str] | None = ..., flush: Literal[False] = ...) -> None

- **Can you correct each of the errors identified by *pylint*?**

  - For snake-case naming convention, we should change the name of the file into 'pylint_test.py'.
  - For module docstring, we should add “““ ”””” before import module.

- For upper-case naming style for constants, we should change the names into ‘SHIFT’, ‘LETTERS’, ‘ENCODED’ and ‘X’.
- For bad indentation, we should change the indentation to four spaces
- For trailing whitespace, we should delete whitespaces.
- For the last statement in the code, we should change it into ‘print(ENCODED)’.

pylint_test.py



4. Testing with Python

Exploring Linters to Support Testing in Python: Question 3

```
pip install flake8
```

Run flake8 on pylintTest.py

- **Review the errors returned. In what way does this error message differ from the error message returned by *pylint*?**

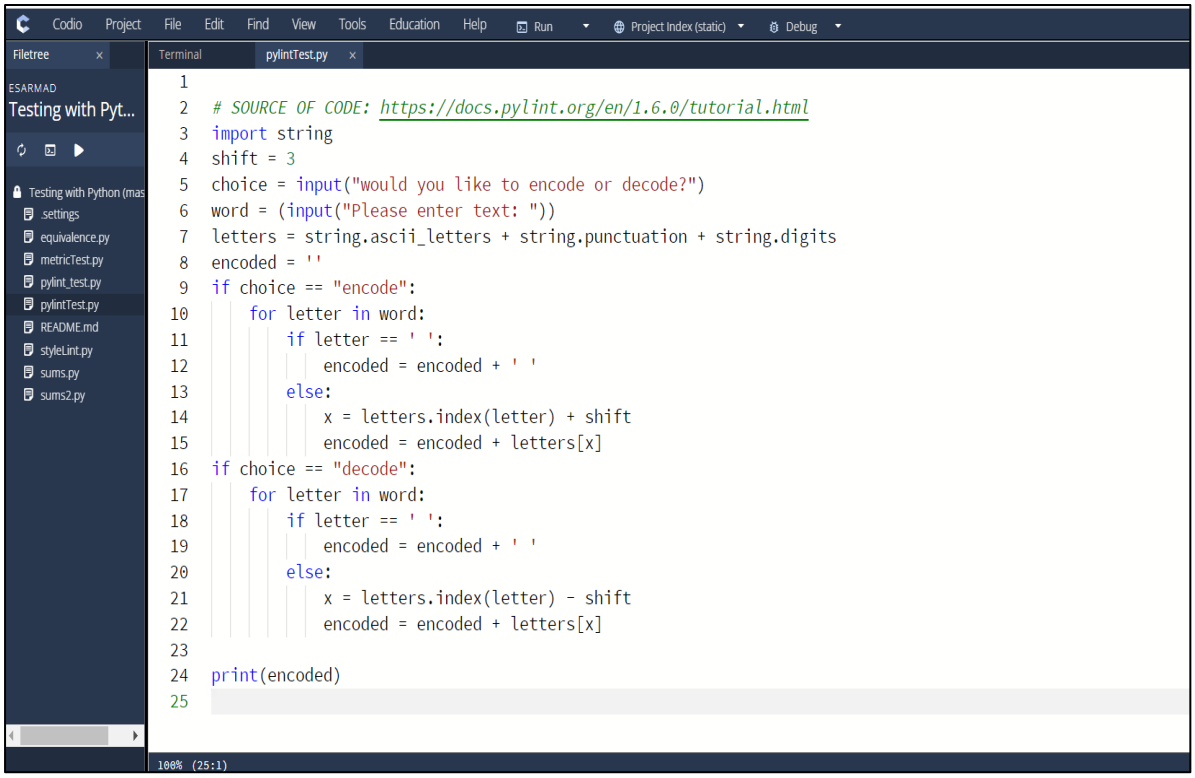
  The errors are as follows:

- Whitespace error
- "raw_input" is not defined Pylance (reportUndefinedVariable)
- indentation is not a multiple of 4 flake8(E111)
- missing whitespace around operator flake8(E225)
- no newline at end of file flake8(W292)

The corrections are as follows:

- For whitespace error, we should remove the whitespaces after import module
- For raw-input(), we should change it into ‘input()’.
- For indentation error, we should change the indentation into 4 spaces.
- For missing whitespace, we should put whitespace around = operator like this: ‘encoded = encoded + letters[x]’
  For newline error, we should change the place of cursor from this line (print(encoded)) to the next line.

pylintTest.py



Run flake8 on metricTest.py

- **Can you correct each of the errors returned by flake8?**

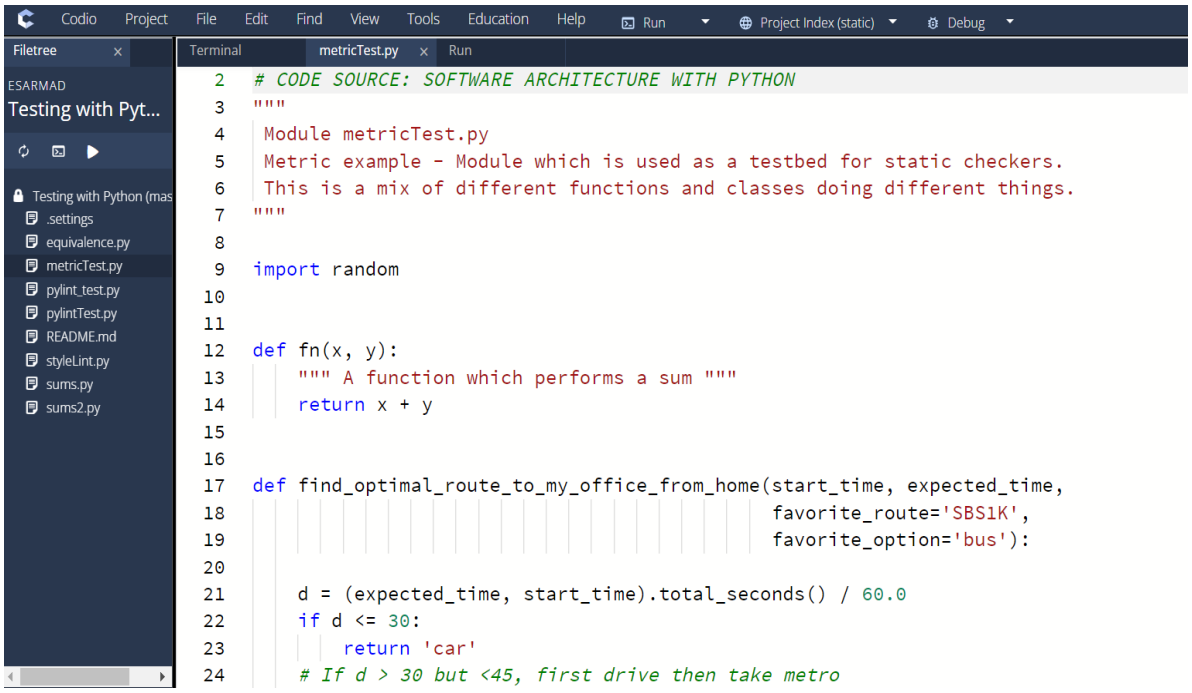
The errors are as follows:

- block comment should start with '# ' flake8(E265)
- trailing whitespace flake8(W291)
- expected 2 blank lines, found 1 flake8(E302)
- missing whitespace after ',' flake8(E231)
- line too long (121 > 79 characters) flake8(E501)
- indentation is not a multiple of 4flake8(E111)
- IndentationError: unindent does not match any outer indentation levelflake8(E999)

- **What amendments have you made to the code?**

  - For block comment with '#' error, we should put a space after '#'.
  - For trailing whitespace error, we should remove spaces at the end of the line.
  - For 2 blank lines, we should put two blank lines before functions,..
  - For missing whitespace after ',', we should put ',' after the name of variable such as 'start_time, expected_time'.
  - For too long line error, we should just put not more than 79 characters in a line.
  - For indentation error, we should change the indentation into 4 spaces.
  - For indentationError, we should change 'if d>45:' into 'if d > 45'.

metricTest.py




```
# CODE SOURCE: SOFTWARE ARCHITECTURE WITH PYTHON
"""
 Module metricTest.py
 Metric example - Module which is used as a testbed for static checkers.
 This is a mix of different functions and classes doing different things.
"""

import random


def fn(x, y):
    """ A function which performs a sum """
    return x + y


def find_optimal_route_to_my_office_from_home(start_time, expected_time,
                                              favorite_route='SBS1K',
                                              favorite_option='bus'):

    d = (expected_time, start_time).total_seconds() / 60.0
    if d <= 30:
        return 'car'
    # If d > 30 but <45, first drive then take metro
```

ESARMAD
Testing with Pyt...

Testing with Python (mas
.settings
equivalence.py
metricTest.py
pylint_test.py
pylintTest.py
README.md
styleLint.py
sums.py
sums2.py

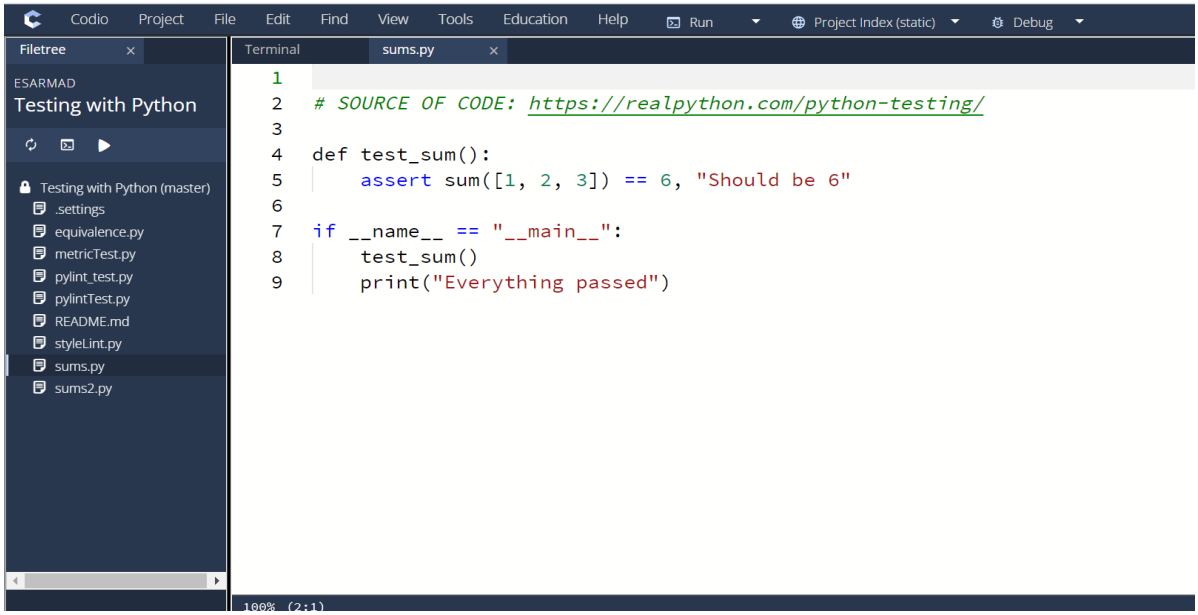
```python
    if d > 30 and d < 45:
        return ('car', 'metro')

    """If d>45 there are a combination of optionsWriting Modifiable
      and Readable Code"""

    if d > 45:
        if d < 60:
            # First volvo,then connecting bus
            return ('bus: 335E', 'bus: connector')
        elif d > 80:
            # Might as well go by normal bus
            return random.choice(('bus: 330', 'bus: 331', ':'.join((
                favorite_option, favorite_route))))
        elif d > 90:
            # Relax and choose favorite route
            return ':'.join((favorite_option, favorite_route))


class C(object):
    """ A class which does almost nothing """

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def f(self):
        pass

    def g(self, x, y):
        if self.x > x:
            return self.x+self.y
        elif x > self.x:
            return x + self.y


class D(C):
    """ D class """
    def __init__(self, x):
        self.x = x

    def f(self, x, y):
        if x > y:
            return x-y
        else:
            return x+y

    def g(self, y):
        if self.x > y:
            return self.x + y
        else:
            return y - self.x
```

5. Testing with Python

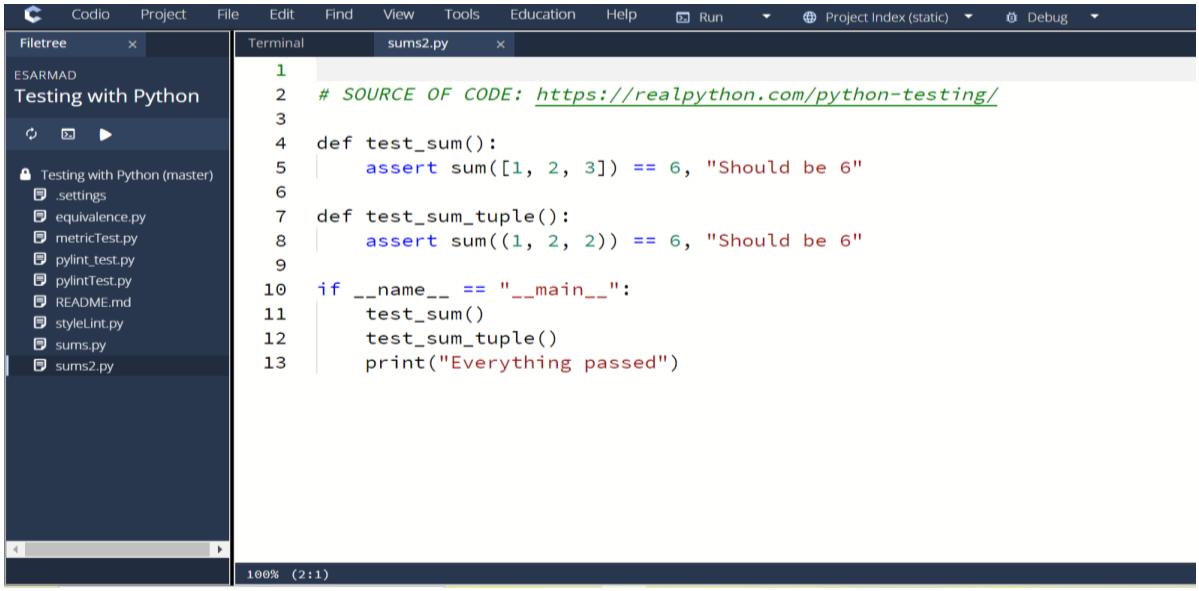
Python: Question 4

```
pip install mccabe
```

sums.py



sums2.py



Run mccabe on sums.py. what is the result?

```
$ python -m mccabe --min 1 sums.py
3:0: 'test_sum' 1
If 6 2
```

Run mccabe on sums2.py. what is the result?

```
$ python -m mccabe --min 1 sums2.py
4:0: 'test_sum' 1
7:0: 'test_sum_tuple' 1
If 10 2
```

- **What are the contributors to the cyclomatic complexity in each piece of code?**

  In ‘sums.py’, the cyclomatic complexity numbers for ‘test_sum’ function is 3 which means that the function is easy to understand and test. The reason behind it is that the function is small, and it has just used one assertion and one function invocation.

  In ‘sums2.py’, the cyclomatic complexity numbers for ‘test_sum’ and ‘test_sum_tuple’ are less than 10. It means that the functions are considered as a medium complexity because of using 2 assertions and 2 function invocations. So, there are more lines of code.