COMPSCI 590N
Lecture 12: Documentation and Testing

Roy J. Adams

College of Information and Computer Sciences
University of Massachusetts Amherst

Outline

Commenting your code

- **Comments** are lines of code that are ignored by the interpreter and serve to document the code for yourself and others.

Commenting your code

- **Comments** are lines of code that are ignored by the interpreter and serve to document the code for yourself and others.
- As you are writing code it can feel like there is no way you will forget what certain functions do or variables represent. Give it a month. You will forget.

Commenting your code

- **Comments** are lines of code that are ignored by the interpreter and serve to document the code for yourself and others.
- As you are writing code it can feel like there is no way you will forget what certain functions do or variables represent. Give it a month. You will forget.
- Reading uncommented code is tedious and time consuming.

Commenting your code

- **Comments** are lines of code that are ignored by the interpreter and serve to document the code for yourself and others.
- As you are writing code it can feel like there is no way you will forget what certain functions do or variables represent. Give it a month. You will forget.
- Reading uncommented code is tedious and time consuming.
- If you are using code for science, it is good practice to treat comments like a lab journal. This documents the experiments you are running for reproducibility.

Comments in Python

There are two types of comments in Python:

```
# Single line comments start
# with a # sign.

"""
Multiline comments start
and end with triple quotes.
"""
```

Documentation

- Code **documentation** describes the user facing reusable parts of your code (i.e. functions and classes) so that other people can use them.

Documentation

- Code **documentation** describes the user facing reusable parts of your code (i.e. functions and classes) so that other people can use them.
- For example: The documentation for a function should describe the function, its inputs, its outputs, and any exceptions it might raise.

Documentation

- Code **documentation** describes the user facing reusable parts of your code (i.e. functions and classes) so that other people can use them.
- For example: The documentation for a function should describe the function, its inputs, its outputs, and any exceptions it might raise.
- If you want anyone to use your code, it needs to be commented and documented.

Docstrings

- **Docstrings** are a special type of comment that the Python interpreter doesn't ignore.

Docstrings

- **Docstrings** are a special type of comment that the Python interpreter doesn't ignore.
- Instead Python reads and stores docstrings and makes them accessible through the `help` function.

Docstrings

- **Docstrings** are a special type of comment that the Python interpreter doesn't ignore.
- Instead Python reads and stores docstrings and makes them accessible through the `help` function.
- Docstrings can be used to provide documentation for scripts, functions, and classes.

Docstrings

- **Docstrings** are a special type of comment that the Python interpreter doesn't ignore.
- Instead Python reads and stores docstrings and makes them accessible through the `help` function.
- Docstrings can be used to provide documentation for scripts, functions, and classes.
- Docstrings must use triple quotes: `""" comment """`

Docstrings

```
"""
    Place a docstring at the top of a script to document
        usage.

    Usage:
        python -h doc_script.py
"""
```

Docstrings

```
def fun(args):
"""
    Place a docstring under a def statement to document
        a function.
    Call help(fun) to view the docstring.

    Inputs: Describe the inputs.

    Outputs: Describe the outputs.

    Raises: Describe the possible exceptions.
"""
```

Docstrings

```
class Foo:
"""
    Place a docstring under a class statement to
        document a class.
    Call help(Foo) to view the docstring.

    Inputs: Describe the inputs to __init__.

    Members: Describe the members functions and
        attributes.
"""
```

Docstrings

Demo

Outline

Testing

- Non-trivial code almost never works the way you expect it to the first time.

Testing

- Non-trivial code almost never works the way you expect it to the first time.
- We have talked about debugging, but debugging is only useful when you know that your program has a bug.

Testing

- Non-trivial code almost never works the way you expect it to the first time.
- We have talked about debugging, but debugging is only useful when you know that your program has a bug.
- **Testing** is how you find out your program is wrong.

Testing

- Non-trivial code almost never works the way you expect it to the first time.
- We have talked about debugging, but debugging is only useful when you know that your program has a bug.
- **Testing** is how you find out your program is wrong.
- Writing good tests for your code is hard, but is extremely important. It saves you time in the long run and saves you a lot of embarrassment.

Testing

- Non-trivial code almost never works the way you expect it to the first time.
- We have talked about debugging, but debugging is only useful when you know that your program has a bug.
- **Testing** is how you find out your program is wrong.
- Writing good tests for your code is hard, but is extremely important. It saves you time in the long run and saves you a lot of embarrassment.
- Many companies (e.g. Google) have strict testing guidelines.

Testing: Minor Digression

- Testing is important everywhere, but in an industry setting, there are usually checks to make sure you have done appropriate testing.

Testing: Minor Digression

- Testing is important everywhere, but in an industry setting, there are usually checks to make sure you have done appropriate testing.
- This is not true in science, yet we are making scientific assertions premised on the the correctness of our code.

Testing: Minor Digression

- Testing is important everywhere, but in an industry setting, there are usually checks to make sure you have done appropriate testing.
- This is not true in science, yet we are making scientific assertions premised on the the correctness of our code.
- Notable examples of failure:

Testing: Minor Digression

- Testing is important everywhere, but in an industry setting, there are usually checks to make sure you have done appropriate testing.
- This is not true in science, yet we are making scientific assertions premised on the the correctness of our code.
- Notable examples of failure:
  - The Geweke test.

Testing: Minor Digression

- Testing is important everywhere, but in an industry setting, there are usually checks to make sure you have done appropriate testing.
- This is not true in science, yet we are making scientific assertions premised on the the correctness of our code.
- Notable examples of failure:
  - The Geweke test.
  - “Growth in a Time of Debt”.

Testing: Minor Digression

- Testing is important everywhere, but in an industry setting, there are usually checks to make sure you have done appropriate testing.
- This is not true in science, yet we are making scientific assertions premised on the the correctness of our code.
- Notable examples of failure:
  - The Geweke test.
  - “Growth in a Time of Debt”.
  - Numerous times in my own attempts to recreate other’s experiments.

Strategies for Testing: Assertions

- **Assertions** are special statements that evaluate a logical expression and cause an error if the statement is false.

Strategies for Testing: Assertions

- **Assertions** are special statements that evaluate a logical expression and cause an error if the statement is false.
- Assertions are a good, cheap way to add sanity checks to you program.

Strategies for Testing: Assertions

- **Assertions** are special statements that evaluate a logical expression and cause an error if the statement is false.
- Assertions are a good, cheap way to add sanity checks to you program.
- For example: If you are working with probabilities, it is common to place assertions that check that all probability values are positive.

Strategies for Testing: Assertions

- **Assertions** are special statements that evaluate a logical expression and cause an error if the statement is false.
- Assertions are a good, cheap way to add sanity checks to you program.
- For example: If you are working with probabilities, it is common to place assertions that check that all probability values are positive.
- One way to think of assertions is as preemptively doing the check we might do when debugging.

Strategies for Testing: Assertions

- **Assertions** are special statements that evaluate a logical expression and cause an error if the statement is false.
- Assertions are a good, cheap way to add sanity checks to you program.
- For example: If you are working with probabilities, it is common to place assertions that check that all probability values are positive.
- One way to think of assertions is as preemptively doing the check we might do when debugging.
- Python uses the `assert` statement:

Strategies for Testing: Assertions

- **Assertions** are special statements that evaluate a logical expression and cause an error if the statement is false.
- Assertions are a good, cheap way to add sanity checks to you program.
- For example: If you are working with probabilities, it is common to place assertions that check that all probability values are positive.
- One way to think of assertions is as preemptively doing the check we might do when debugging.
- Python uses the `assert` statement:

Strategies for Testing: Assertions

- **Assertions** are special statements that evaluate a logical expression and cause an error if the statement is false.
- Assertions are a good, cheap way to add sanity checks to you program.
- For example: If you are working with probabilities, it is common to place assertions that check that all probability values are positive.
- One way to think of assertions is as preemptively doing the check we might do when debugging.
- Python uses the `assert` statement:

```
assert <logical_statement>
```

Strategies for Testing: Unit Testing

- **Unit testing** is a form of program testing that breaks the program into small portions called **units** and tests each one separately.

Strategies for Testing: Unit Testing

- **Unit testing** is a form of program testing that breaks the program into small portions called **units** and tests each one separately.
- Ideally, a unit is the smallest block of testable code.

Strategies for Testing: Unit Testing

- **Unit testing** is a form of program testing that breaks the program into small portions called **units** and tests each one separately.
- Ideally, a unit is the smallest block of testable code.
- Test input/output pairs for each unit.

Strategies for Testing: Unit Testing

- **Unit testing** is a form of program testing that breaks the program into small portions called **units** and tests each one separately.
- Ideally, a unit is the smallest block of testable code.
- Test input/output pairs for each unit.
- Why use unit testing?

Strategies for Testing: Unit Testing

- **Unit testing** is a form of program testing that breaks the program into small portions called **units** and tests each one separately.
- Ideally, a unit is the smallest block of testable code.
- Test input/output pairs for each unit.
- Why use unit testing?
  - Easier to write tests.

Strategies for Testing: Unit Testing

- **Unit testing** is a form of program testing that breaks the program into small portions called **units** and tests each one separately.
- Ideally, a unit is the smallest block of testable code.
- Test input/output pairs for each unit.
- Why use unit testing?
  - Easier to write tests.
  - Easier to identify which part of the program is causing an error.

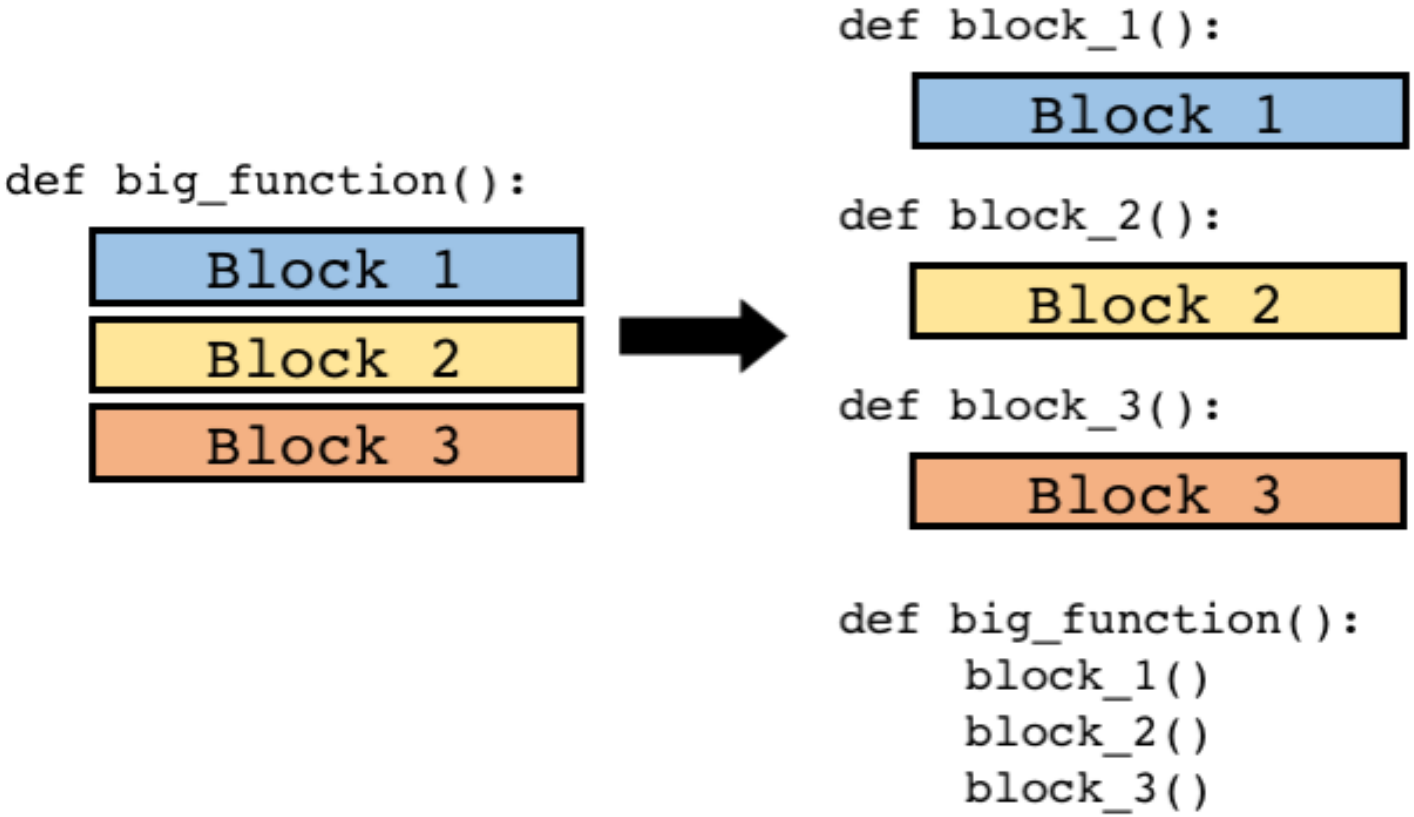
Strategies for Testing: Unit Testing

- **Unit testing** is a form of program testing that breaks the program into small portions called **units** and tests each one separately.
- Ideally, a unit is the smallest block of testable code.
- Test input/output pairs for each unit.
- Why use unit testing?
  - Easier to write tests.
  - Easier to identify which part of the program is causing an error.
  - If changes are made only to a single unit, you can be confident that all of the other units are still valid.

Strategies for Testing: Unit Testing

```
def big_function():
    Block 1
    Block 2
    Block 3
```

Strategies for Testing: Unit Testing



unittest

The basic Python unit testing module is called `unittest`.

unittest

The basic Python unit testing module is called `unittest`.

```
import unittest

def fun(x):
    return x**2

class FunTester(unittest.TestCase):
    def test_case_1(self):
        got = fun(3)
        expected = 9
        self.assertEqual(got,expected)

if __name__=="__main__":
    unittest.main()
```

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.
- Includes many custom assertions for testing different properties of the output:

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.
- Includes many custom assertions for testing different properties of the output:
  - `assertTrue(a)` - Assert that a == True.

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.
- Includes many custom assertions for testing different properties of the output:
  - `assertTrue(a)` - Assert that a == True.
  - `assertEqual(a,b)` - Assert that a == b.

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.
- Includes many custom assertions for testing different properties of the output:
  - `assertTrue(a)` - Assert that a == True.
  - `assertEqual(a,b)` - Assert that a == b.
  - `assertAlmostEqual(a,b)` - Similar to isclose.

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.
- Includes many custom assertions for testing different properties of the output:
  - `assertTrue(a)` - Assert that a == True.
  - `assertEqual(a,b)` - Assert that a == b.
  - `assertAlmostEqual(a,b)` - Similar to isclose.
  - `assertCountEqual` - Assert that the two sequences contain the same items, regardless of order.

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.
- Includes many custom assertions for testing different properties of the output:
  - `assertTrue(a)` - Assert that a == True.
  - `assertEqual(a,b)` - Assert that a == b.
  - `assertAlmostEqual(a,b)` - Similar to isclose.
  - `assertCountEqual` - Assert that the two sequences contain the same items, regardless of order.
  - Many more.

unittest

- `unittest.main()` searches all instance of `unittest.TestCase` and runs all member functions that start with “test”.
- Includes many custom assertions for testing different properties of the output:
  - `assertTrue(a)` - Assert that a == True.
  - `assertEqual(a,b)` - Assert that a == b.
  - `assertAlmostEqual(a,b)` - Similar to isclose.
  - `assertCountEqual` - Assert that the two sequences contain the same items, regardless of order.
  - Many more.
- Many modules extend unittest to either add features or simplify the syntax: py.test, nose, testify, etc.

doctest

`doctest` allows you to write simple test directly in a docstring.

doctest

`doctest` allows you to write simple test directly in a docstring.

```
def fun(x):
"""
    >>> fun(3)
    9
"""
    return x**2
```

doctest

`doctest` allows you to write simple test directly in a docstring.

```
def fun(x):
"""
    >>> fun(3)
    9
"""
    return x**2
```

To run the tests call doctest on the script:

```
python -m doctest script.py
```