

CSIT121

Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 4 outline

- Modules and packages
- Third-party libraries
- Test-driven development
- Unittest module
- Code coverage

Modules

- For small programs, we can put all our classes into one file and add a little script at the end to start them interacting.
- For large programs, it can become difficult to find the one class that needs to be edited among the many classes we've defined.
- We need to use **modules**.
- Modules are simply Python files. One Python file equals one module.
- If we have multiple modules with different class definitions, we can load classes from one module to reuse them in the other modules.
E.g. put all classes and functions related to the databased access into the module 'database.py'.

Modules

- The '**import**' statement is used for importing modules or specific classes or functions from modules. E.g. we used the import statement to get Python's built-in math module and use its sqrt function in the distance calculation

Suppose we have:

- A module called 'database.py' which contains a Database class
- A second module called 'products.py' which responsible for product-related queries.
- 'products.py' needs to instantiate the Database class from 'database.py' module so that it can execute queries on the product table in the database.

We can import modules, classes or functions with different syntax

Modules

- To import the **whole database module** into the products namespace so any class or function in the database module can be accessed using the '**database.<something>**' notation.

Import database

db = database.Database()

Do queries on db

Modules

- To import just the one class (Database class) from the database module into the products namespace so all the class functions can be accessed directly.

from database import Database

db = Database()

Do queries on db

Modules

- If the products module already has a class called Database, and we don't want the two names to be confused, we can rename the imported class when used inside the products module

```
from database import Database as DB
```

```
db = DB()
```

```
# Do queries on db
```

Modules

- We can also import multiple items in one statement. If our database module also contains a Query class, we can import both classes as follows.

```
from database import Database, Query
```

```
db = Database()
```

```
query = Query()
```

```
# Do queries on db via query
```


Modules

- Don't do this

from database import *

Because,

- You will never know when classes will be used
- Not easy to check the class details (using help()) and maintain your program
- Unexpected classes and functions will be imported. For example, it will also import any classes or modules that were themselves imported into that file.

Modules

- For fun, try typing 'import this' in your interactive interpreter. You will get a poem about Python philosophy. 😊

```
import this
```

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
```

```
Explicit is better than implicit.
```

```
Simple is better than complex.
```

```
Complex is better than complicated.
```

```
Flat is better than nested.
```

```
Sparse is better than dense.
```

```
Readability counts.
```

```
Special cases aren't special enough to break the rules.
```

```
Although practicality beats purity.
```

```
Errors should never pass silently.
```

```
Unless explicitly silenced.
```

```
In the face of ambiguity, refuse the temptation to guess.
```

```
There should be one-- and preferably only one --obvious way to do it.
```

```
Although that way may not be obvious at first unless you're Dutch.
```

```
Now is better than never.
```

```
Although never is often better than *right* now.
```

```
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.
```

```
Namespaces are one honking great idea -- let's do more of those!
```

Package

- As a project grows into a collection of more and more modules, it is better to add another level of abstract.
- As modules equal files, one straightforward solution is to organise files with folders (called packages).
- A package is a collection of modules in a folder. The name of the package is the name of the folder.
- We need to tell Python that a folder is a package to distinguish it from other folders in the directory.
- We need to place a special file in the folder named `'__init__.py'`.
- If we forget this file, we won't be able to import modules from that folder.

Package

- Put our modules inside an **ecommerce** package in our working folder (parent_directory)
- The working folder also contain a **main.py** module to start the program.
- We can add another **payment** package inside the **ecommerce** package for various payment options.
- The folder hierarchy will look like this.

```
parent_directory/  
    main.py  
    ecommerce/  
        __init__.py  
        database.py  
        products.py  
        payments/  
            __init__.py  
            square.py  
            stripe.py
```

Package

In Python 3, there are two ways of importing modules: **absolute imports** and **relative imports**.

- Absolute imports specify the complete path to the module, function, or class we want to import
- Relative imports find a class, function, or module as it is positioned relative to the current module.

Package

Absolute imports

- If we need access to the Product class inside the products module, we could use any of these syntaxes to perform an absolute import.

```
import ecommerce.products  
product = ecommerce.products.Product()
```

//or

```
from ecommerce.products import Product  
product = Product()
```

//or

```
from ecommerce import products  
product = products.Product()
```

```
parent_directory/  
  main.py  
  ecommerce/  
    __init__.py  
    database.py  
    products.py  
    payments/  
      __init__.py  
      square.py  
      stripe.py
```

Package

Absolute imports

Which way is better? It depends.

```
import ecommerce.products  
product = ecommerce.products.Product()
```

//or

```
from ecommerce.products import Product  
product = Product()
```

//or

```
from ecommerce import products  
product = products.Product()
```

- The first way is normally use if you have some kind of name conflict from multiple modules. You have to specify the whole path before the function calls.
- If you only need to import one or two classes, you can use the second way. Easy to call functions.
- If there are dozens of classes and functions inside the module that you want to use, you can import the module using the third way.

Package

Relative imports

- If we are working in the products module and we want to import the Database class from the database module next to it, we could use a relative import.

from .database import Database

- The period '.' in front of database says 'using the database module inside the current package'.
- The current package refers to the package containing the module (products.py) we are currently working in, i.e., the ecommerce package.

```
parent_directory/  
  main.py  
  ecommerce/  
    __init__.py  
    database.py  
    products.py  
    payments/  
      __init__.py  
      square.py  
      stripe.py
```


Package

Relative imports

- If we were editing the *square.py* module inside the payments package, we want to use the database package inside the parent package.

from ..database import Database

- We use more periods to go further up the hierarchy.
- If we had an *ecommerce.contact* package containing an *email* module and wanted to import the `send_mail` function.

from ..contact.email import send_mail

```
parent_directory/  
  main.py  
  ecommerce/  
    __init__.py  
    database.py  
    products.py  
    payments/  
      __init__.py  
      square.py  
      stripe.py
```

Inside a module

- We specify variables, classes or functions inside modules.
- They can be a handy way to shore the global state without namespace conflicts.
- For example, it might make more sense to have only one database object globally available from the database module.
- The database module might look like this:

```
class Database:  
    # the database implementation  
    pass
```

```
database = Database()
```

Inside a module

- The we can use any of the import methods we've discussed to access the database object, such as

```
from ecommerce.database import database
```

- In some situations, it may be better to create objects until it is actually needed to avoid unnecessary delay of the program.

```
class Database:  
    # the database implementation  
    pass
```

```
database = None
```

```
def initialize_database():  
    global database  
    database = Database()
```

Inside a module

- All module-level code is executed immediately at the time it is imported.
- However, the internal code of functions will not be executed until the function is called.
- To simplify the process, we should always put our start-up code in a function (conventionally called 'main') and only execute that function when we know we are running the module as a script, but not when our code is being imported from a different script.
- We can do this by guarding the call to 'main' inside a conditional statement.

Inside a module

```
class UsefulClass:
    """This class might be useful to other modules."""

    pass


def main():
    """Creates a useful class and does something with it for our module."""
    useful = UsefulClass()
    print(useful)


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

Important note: Make it a policy to wrap all your scripts in an `if __name__ == "__main__":` `your_testing_code` pattern in case you write a function that you may want to be imported by other code at some point in the future.

Third-party libraries

- You can find third-party libraries on the Python Package Index (PyPI) at <http://pypi.python.org/>. Then you can install the libraries with a tool called 'pip'
- However, 'pip' is not pre-installed in Python, please follow the instructions to download and install 'pip': <http://pip.readthedocs.org/>
- For Python 3.4 and higher, you can use a built-in tool called 'ensurepip' by installing it with the command '\$python3 -m ensurepip'
- Then, you can install libraries via the command '\$pip install <library_name>'
- Then the third-party library will be installed directly into your system Python directory.

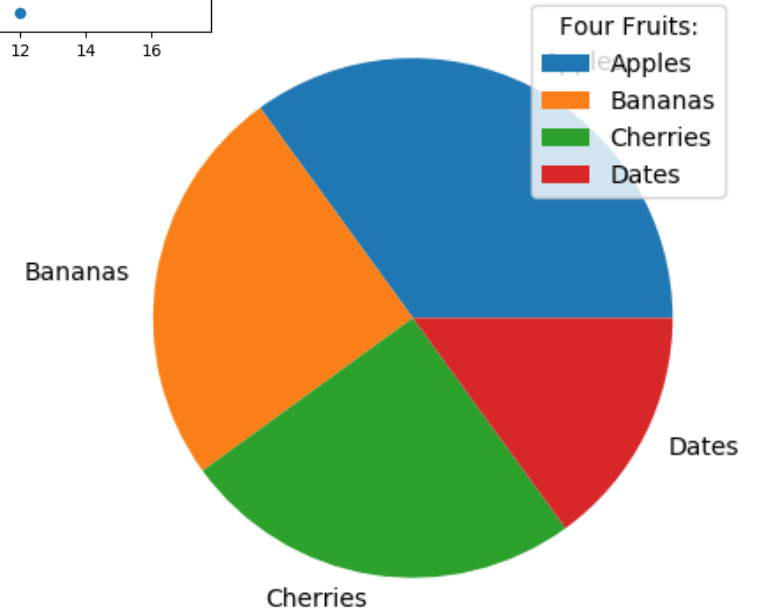
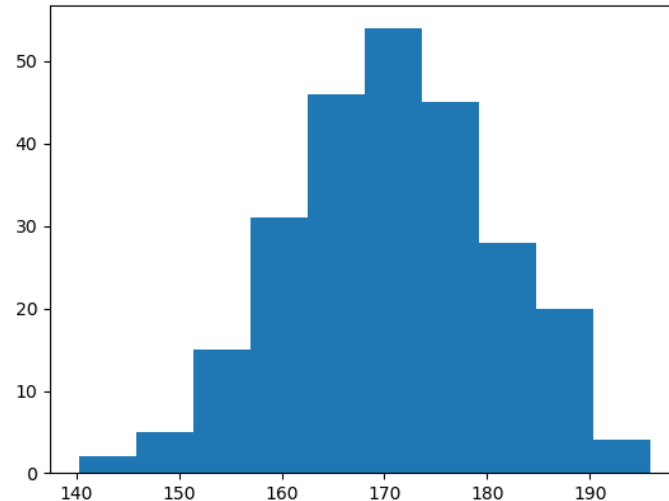
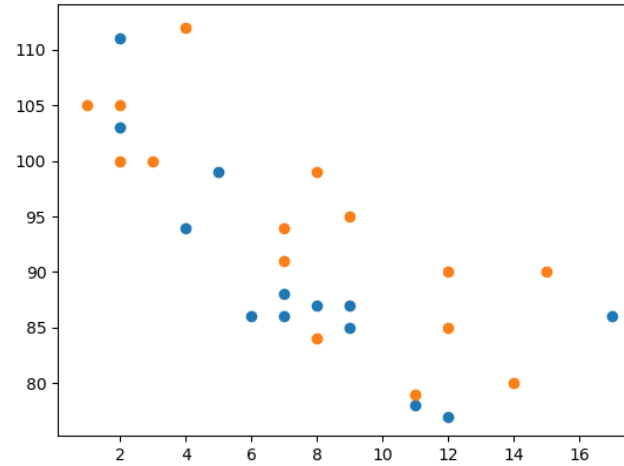
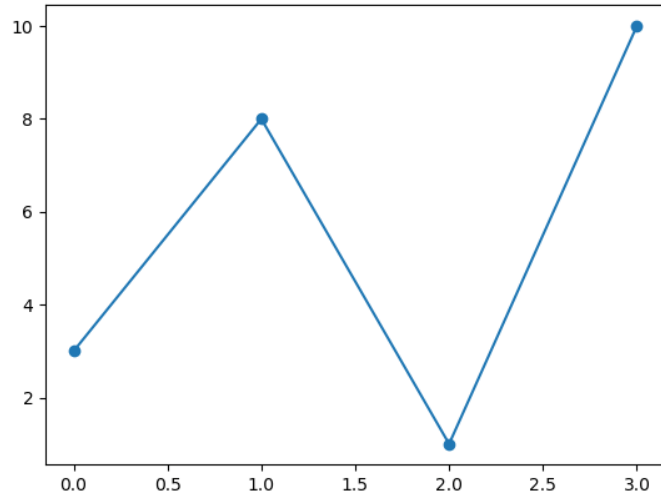
Third-party libraries

- If you don't want to touch the system Python directory, you can start a virtual environment in your working directory called 'mini Python'.
- When you activate the mini Python, commands related to Python will work on the working directory instead of the system directory.

```
cd project_directory
python -m venv env
source env/bin/activate    # on Linux or macOS
env/bin/activate.bat      # on Windows
```

Third-party libraries

We will learn Matplotlib (a low-level graph plotting library) in this subject.



Program Testing

Why test?

- To ensure that code is working the way the developer thinks it should
- To ensure that code continues working when we make changes
- To ensure that the developer understood the requirements
- To ensure that the code we are writing has a maintainable interface

Test-driven development

Principle

- Write tests for a segment of code
- Test your code (will fail because you don't write the code yet)
- Write your code and ensures the test passes
- Write another test for the next segment of code
- Write your code and ensures the test passes

...

It is fun. You build a puzzle for yourself first, then you solve it!

Test-driven development

The test-driven methodology

- ensure that tests really get written
- forces us to consider exactly how the code will be used. It tells us what methods objects need to have and how attributes will be accessed.
- helps us break up the initial problem into smaller, testable problems, and then to recombine the tested solutions into larger, also tested, solutions.
- Help us to discover anomalies in the design that force us to consider new aspects of the software.
- will not leave the testing job to the program users

Unit Test

- Same as Java, Python also has a built-in test library called 'unittest'
- 'unittest' provides several tools for creating and running unit tests.
- The most important one is the 'TestCase' class.
- 'TestCase' class provides a set of methods that allow us to compare values, set up tests, and clean up when the tests have finished.

Unit Test

- Create a subclass of TestCase (we will introduce the inheritance next lecture) and write individual methods to do the actual testing.
- These method names must all start with the prefix 'test'.
- TestCase class will automatically run all the test methods and report the test results

Unit Test

- Pass the test

```
1 # calculations.py
2 |
3 class Calculations:
4     def __init__(self, a, b):
5         self.a = a
6         self.b = b
7
8     def get_sum(self):
9         return self.a + self.b
10
11    def get_difference(self):
12        return self.a - self.b
13
14    def get_product(self):
15        return self.a * self.b
16
17    def get_quotient(self):
18        return self.a / self.b
```

```
1 import unittest
2 from calculations import Calculations
3
4 class TestCalculations(unittest.TestCase):
5
6     def test_sum(self):
7         calculation = Calculations(8, 2)
8         self.assertEqual(calculation.get_sum(), 10)
9
10 if __name__ == '__main__':
11     unittest.main()
```

```
= RESTART: /Users/fren/Library/Cl
g/CSIT121/Autumn_2023_Python/exar
```

.

Ran 1 test in 0.010s

OK

Unit Test

- fail the test

```
1 import unittest
2 from calculations import Calculations
3
4 class TestCalculations(unittest.TestCase):
5
6     def test_sum(self):
7         calculation = Calculations(8, 2)
8         self.assertEqual(calculation.get_sum(), 11)
9
10 if __name__ == '__main__':
11     unittest.main()
```

F

=====

FAIL: test_sum (__main__.TestCalculations.test_sum)

Traceback (most recent call last):

File "/Users/fren/Library/CloudStorage/OneDrive-UniversityofWollongong/MyTeaching/C
SIT121/Autumn_2023_Python/examples/CalculationsTest.py", line 8, in test_sum

self.assertEqual(calculation.get_sum(), 11)

AssertionError: 10 != 11

Ran 1 test in 0.013s

FAILED (failures=1)

Unit Test

- more tests

```
1 import unittest
2 from calculations import Calculations
3
4 class TestCalculations(unittest.TestCase):
5
6     def test_sum(self):
7         calculation = Calculations(8, 2)
8         self.assertEqual(calculation.get_sum(), 11)
9
10    def test_difference(self):
11        calculation = Calculations(8, 2)
12        self.assertEqual(calculation.get_difference(), 6)
13
14    def test_product(self):
15        calculation = Calculations(8, 2)
16        self.assertEqual(calculation.get_product(), 15)
17
18    def test_quotient(self):
19        calculation = Calculations(8, 2)
20        self.assertEqual(calculation.get_quotient(), 4)
21
22 if __name__ == '__main__':
23     unittest.main()
```

```
..F.F
=====
FAIL: test_product (__main__.TestCalculations.test_product)
=====
Traceback (most recent call last):
  File "/Users/fren/Library/CloudStorage/OneDrive-UniversityofWollongong/MyTeaching/CSIT121/Autumn_2023_Python/examples/CalculationsTest.py", line 16, in test_product
    self.assertEqual(calculation.get_product(), 15)
AssertionError: 16 != 15
=====
FAIL: test_sum (__main__.TestCalculations.test_sum)
=====
Traceback (most recent call last):
  File "/Users/fren/Library/CloudStorage/OneDrive-UniversityofWollongong/MyTeaching/CSIT121/Autumn_2023_Python/examples/CalculationsTest.py", line 8, in test_sum
    self.assertEqual(calculation.get_sum(), 11)
AssertionError: 10 != 11
=====
Ran 4 tests in 0.028s
FAILED (failures=2)
```


Unit Test

- Assert methods

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x)</code> is True
<code>assertFalse(x)</code>	<code>bool(x)</code> is False
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

Reducing boilerplate and cleaning up

- No need to write the same setup code for each test method if the test cases are the same.
- We can use the 'setUp()' method on the TestCase class to perform initialisation for test methods.

```
1 import unittest
2 from calculations import Calculations
3
4 class TestCalculations(unittest.TestCase):
5
6     def setUp(self):
7         self.calculation = Calculations(8,2)
8
9     def test_sum(self):
10         self.assertEqual(self.calculation.get_sum(), 10)
11
12     def test_difference(self):
13         self.assertEqual(self.calculation.get_difference(), 6)
14
15     def test_product(self):
16         self.assertEqual(self.calculation.get_product(), 16)
17
18     def test_quotient(self):
19         self.assertEqual(self.calculation.get_quotient(), 4)
20
21 if __name__ == '__main__':
22     unittest.main()
```

....

Ran 4 tests in 0.047s

OK

Organise your test classes

- We should divide our test classes into modules and packages (keep them organized)
- Python's discover module ('python3 -m unittest') can find any TestCase objects in modules if your tests module starts with the keyword 'test'.
- Most Python programmers also choose to put their tests in a separate package (usually named 'tests/' alongside their source directory).

```
examples -- -zsh -- 130x39
[fren@UW-XY06K990HQ examples % ls
Circle.py          LineSegment.uxf      Polygon.py           test_calculations.py
HelloWorld.py      LineSegments.py      __pycache__         test_str.py
HelloWorld2.py     LineSegmentsUserInput.py  addition.py
LineSegment.py     Point.py             calculations.py
[fren@UW-XY06K990HQ examples % python3 -m unittest
.....
-----
Ran 7 tests in 0.000s

OK
[fren@UW-XY06K990HQ examples % python3 -m unittest -v
test_difference (test_calculations.TestCalculations.test_difference) ... ok
test_product (test_calculations.TestCalculations.test_product) ... ok
test_quotient (test_calculations.TestCalculations.test_quotient) ... ok
test_sum (test_calculations.TestCalculations.test_sum) ... ok
test_isupper (test_str.TestStringMethods.test_isupper) ... ok
test_split (test_str.TestStringMethods.test_split) ... ok
test_upper (test_str.TestStringMethods.test_upper) ... ok
-----
Ran 7 tests in 0.000s

OK
[fren@UW-XY06K990HQ examples % python3 -m unittest test_calculations -v
test_difference (test_calculations.TestCalculations.test_difference) ... ok
test_product (test_calculations.TestCalculations.test_product) ... ok
test_quotient (test_calculations.TestCalculations.test_quotient) ... ok
test_sum (test_calculations.TestCalculations.test_sum) ... ok
-----
Ran 4 tests in 0.000s

OK
[fren@UW-XY06K990HQ examples % python3 -m unittest test_calculations.TestCalculations.test_sum -v
test_sum (test_calculations.TestCalculations.test_sum) ... ok
-----
Ran 1 test in 0.000s

OK
```

Ignoring broken tests

- Sometimes, a test is known to fail, but we don't want to report the failure.
- Python provides a few decorators to mark tests that are expected to fail or to be skipped under known conditions.
- 'expectedFailure()', 'skip(reason)', skipIf(condition, reason)', 'skipUnless(condition, reason)'

Ignoring broken tests

```
1 import unittest
2 import sys
3
4
5 class SkipTests(unittest.TestCase):
6     @unittest.expectedFailure
7     def test_fails(self):
8         self.assertEqual(False, True)
9
10    @unittest.skip("Test is useless")
11    def test_skip(self):
12        self.assertEqual(False, True)
13
14    @unittest.skipIf(sys.version_info.minor == 7, "broken on 3.7")
15    def test_skipif(self):
16        self.assertEqual(False, True)
17
18    @unittest.skipUnless(
19        sys.platform.startswith("linux"), "broken unless on linux"
20    )
21    def test_skipunless(self):
22        self.assertEqual(False, True)
23
24
25 if __name__ == "__main__":
26     unittest.main()
```

```
xsFs
=====
FAIL: test_skipif (__main__.SkipTests.test_skipif)
=====
Traceback (most recent call last):
  File "/Users/fren/Library/CloudStorage/OneDrive-UniversityofWollongong/MyTeaching/CSIT121/Autumn_2023_Python/Python-3-Object-Oriented-Programming-Third-Edition-master/Chapter12/test_skipping.py", line 16, in test_skipif
    self.assertEqual(False, True)
AssertionError: False != True
=====

Ran 4 tests in 0.075s

FAILED (failures=1, skipped=2, expected failures=1)
```

Tests results:

- The first test fails and is reported as an expected failure with the mark 'x'
- The second test is never run and marked as 's'
- The third and four tests may or may not be run depending on the current Python version and operation system.

Ignoring broken tests


```
fren@UW-XY06K990HQ examples % python3 -m unittest -v
test_difference (test_calculations.TestCalculations.test_difference) ... ok
test_product (test_calculations.TestCalculations.test_product) ... ok
test_quotient (test_calculations.TestCalculations.test_quotient) ... ok
test_sum (test_calculations.TestCalculations.test_sum) ... ok
test_fails (test_skipping.SkipTests.test_fails) ... expected failure
test_skip (test_skipping.SkipTests.test_skip) ... skipped 'Test is useless'
test_skipif (test_skipping.SkipTests.test_skipif) ... FAIL
test_skipunless (test_skipping.SkipTests.test_skipunless) ... skipped 'broken unless on linux'
test_isupper (test_str.TestStringMethods.test_isupper) ... ok
test_split (test_str.TestStringMethods.test_split) ... ok
test_upper (test_str.TestStringMethods.test_upper) ... ok
```

How much testing is enough?

- How can we tell how well our code is tested?
 - This is a hard question, and we actually do not know whether our code is tested properly and throughout.
- How do we know how much of our code is being tested and how much is broken?
 - This is an easy question, and we can use the code coverage to check.
 - We can check the number of lines that are in the program and get an estimation of what percentage of the code was really tested or covered.

How much testing is enough?

- In Python, the most popular tool for testing code coverage is called 'coverage.py'
- It can be installed using the 'pip3 install coverage' command

```
fren@UW-XY06K990HQ examples % pip3 install coverage
Collecting coverage
  Downloading coverage-7.2.5-cp311-cp311-macosx_11_0_arm64.whl (200 kB)
     200.6/200.6 kB 9.2 MB/s eta 0:00:00
Installing collected packages: coverage
Successfully installed coverage-7.2.5
```

- coverage.py works in three phases:
 - Execution: Coverage.py runs your code, and monitors it to see what lines were executed. (command 'coverage run <your_code>')
 - Analysis: Coverage.py examines your code to determine what lines could have run.
 - Reporting: Coverage.py combines the results of execution and analysis to produce a coverage number and an indication of missing execution. (command 'coverage report' or 'coverage html')

How much testing is enough?

- Execution: 'coverage run -m <your_code>'

```
fren@UW-XY06K990HQ examples % coverage run -m unittest
....xsFs...
=====
FAIL: test_skipif (test_skipping.SkipTests.test_skipif)
-----
Traceback (most recent call last):
  File "/Users/fren/Library/CloudStorage/OneDrive-UniversityofWollongong/MyTeaching/CSIT121/Autumn_
2023_Python/examples/test_skipping.py", line 16, in test_skipif
    self.assertEqual(False, True)
AssertionError: False != True
-----
Ran 11 tests in 0.001s

FAILED (failures=1, skipped=2, expected failures=1)
```

How much testing is enough?

- Analysis & Reporting: command 'coverage report -m' and 'coverage html'

```
[fren@UW-XY06K990HQ examples % coverage report -m
Name                               Stmts  Miss  Cover   Missing
-----
calculations.py                    12      0   100%
test_calculations.py               15      1    93%    22
test_skipping.py                   17      3    82%    12, 22, 26
test_str.py                        14      1    93%    20
-----
TOTAL                             58      5    91%
[fren@UW-XY06K990HQ examples % coverage html
Wrote HTML report to htmlcov/index.html
[fren@UW-XY06K990HQ examples %
```

How much testing is enough?

HTML reports

- calculations_py.html
- coverage_html.js
- favicon_32.png
- index.html
- keybd_closed.png
- keybd_open.png
- status.json
- style.css
- test_calculations_py.html
- test_skipping_py.html
- test_str_py.html

Coverage report: 91%

coverage.py v7.2.5, created at 2023-05-22 10:29 +1000

Module	statements	missing	excluded	coverage
calculations.py	12	0	0	100%
test_calculations.py	15	1	0	93%
test_skipping.py	17	3	0	82%
test_str.py	14	1	0	93%
Total	58	5	0	91%

coverage.py v7.2.5, created at 2023-05-22 10:29 +1000

Coverage for test_calculations.py: 93%

15 statements 14 run 1 missing 0 excluded

« prev ^ index » next coverage.py v7.2.5, created at 2023-05-22 10:29 +1000

```
1 import unittest
2 from calculations import Calculations
3
4 class TestCalculations(unittest.TestCase):
5
6     def setUp(self):
7         self.calculation = Calculations(8,2)
8
9     def test_sum(self):
10         self.assertEqual(self.calculation.get_sum(), 10)
11
12     def test_difference(self):
13         self.assertEqual(self.calculation.get_difference(), 6)
14
15     def test_product(self):
16         self.assertEqual(self.calculation.get_product(), 16)
17
18     def test_quotient(self):
19         self.assertEqual(self.calculation.get_quotient(), 4)
20
21 if __name__ == '__main__':
22     unittest.main()
```

« prev ^ index » next coverage.py v7.2.5, created at 2023-05-22 10:29 +1000

Suggested reading

Python 3 Object-Oriented Programming

- Preface
- Chapter 2: Objects in Python
- Chapter 12: Testing object-oriented programs

Python

- <https://www.python.org/>
- <https://docs.python.org/3/library/unittest.html#>