You should think of tests as an insurance policy on your code. Good tests give you confidence that your code is correct. If you refactor or expand your code, tests that verify behavior—*not* implementation—make it easy to identify what's changed. It sounds counterintuitive, but having good tests actually makes it easier to modify Python code, not harder.

## Item 75: Use `repr` Strings for Debugging Output

When debugging a Python program, the `print` function and format strings (see Item 4: "Prefer Interpolated F-Strings Over C-style Format Strings and `str.format`"), or output via the `logging` built-in module, will get you surprisingly far. Python internals are often easy to access via plain attributes (see Item 42: "Prefer Public Attributes Over Private Ones"). All you need to do is call `print` to see how the state of your program changes while it runs and understand where it goes wrong.

The `print` function outputs a human-readable string version of whatever you supply it. For example, printing a basic string prints the contents of the string without the surrounding quote characters:

```
print('foo bar')
```

```
>>>
foo bar
```

This is equivalent to all of these alternatives:

- Calling the `str` function before passing the value to `print`
- Using the `'%s'` format string with the `%` operator
- Default formatting of the value with an f-string
- Calling the `format` built-in function
- Explicitly calling the `__format__` special method
- Explicitly calling the `__str__` special method

Here, I verify this behavior:

```
my_value = 'foo bar'
print(str(my_value))
print('%s' % my_value)
print(f'{my_value}')
print(format(my_value))
print(my_value.__format__('s'))
print(my_value.__str__())
```

```
>>>
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar
```

The problem is that the human-readable string for a value doesn't make it clear what the actual type and its specific composition are. For example, notice how in the default output of `print`, you can't distinguish between the types of the number `5` and the string `'5'`:

```
print(5)
print('5')

int_value = 5
str_value = '5'
print(f'{int_value} == {str_value} ?')
```

```
>>>
5
5
5 == 5 ?
```

If you're debugging a program with `print`, these type differences matter. What you almost always want while debugging is to see the `repr` version of an object. The `repr` built-in function returns the *printable representation* of an object, which should be its most clearly understandable string representation. For most built-in types, the string returned by `repr` is a valid Python expression:

```
a = '\x07'
print(repr(a))
```

```
>>>
'\x07'
```

Passing the value from `repr` to the `eval` built-in function should result in the same Python object that you started with (and, of course, in practice you should only use `eval` with extreme caution):

```
b = eval(repr(a))
assert a == b
```

When you're debugging with `print`, you should call `repr` on a value before printing to ensure that any difference in types is clear:

```
print(repr(5))
print(repr('5'))
```

```
>>>
5
'5'
```

This is equivalent to using the `'%r'` format string with the `%` operator or an f-string with the `!r` type conversion:

```
print('%r' % 5)
print('%r' % '5')

int_value = 5
str_value = '5'
print(f'{int_value!r} != {str_value!r}')
```

```
>>>
5
'5'
5 != '5'
```

For instances of Python classes, the default human-readable string value is the same as the `repr` value. This means that passing an instance to `print` will do the right thing, and you don't need to explicitly call `repr` on it. Unfortunately, the default implementation of `repr` for `object` subclasses isn't especially helpful. For example, here I define a simple class and then print one of its instances:

```
class OpaqueClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = OpaqueClass(1, 'foo')
print(obj)
```

```
>>>
<__main__.OpaqueClass object at 0x10963d6d0>
```

This output can't be passed to the `eval` function, and it says nothing about the instance fields of the object.

There are two solutions to this problem. If you have control of the class, you can define your own `__repr__` special method that returns a string containing the Python expression that re-creates the object. Here, I define that function for the class above:

```
class BetterClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __repr__(self):
        return f'BetterClass({self.x!r}, {self.y!r})'
```

Now the repr value is much more useful:

```
obj = BetterClass(2, 'bar')
print(obj)
```

```
>>>
BetterClass(2, 'bar')
```

When you don't have control over the class definition, you can reach into the object's instance dictionary, which is stored in the __dict__ attribute. Here, I print out the contents of an OpaqueClass instance:

```
obj = OpaqueClass(4, 'baz')
print(obj.__dict__)
```

```
>>>
{'x': 4, 'y': 'baz'}
```

### Things to Remember

- Calling print on built-in Python types produces the human-readable string version of a value, which hides type information.
- Calling repr on built-in Python types produces the printable string version of a value. These repr strings can often be passed to the eval built-in function to get back the original value.
- %s in format strings produces human-readable strings like str. %r produces printable strings like repr. F-strings produce human-readable strings for replacement text expressions unless you specify the !r suffix.
- You can define the __repr__ special method on a class to customize the printable representation of instances and provide more detailed debugging information.

## Item 76: Verify Related Behaviors in TestCase Subclasses

The canonical way to write tests in Python is to use the unittest built-in module. For example, say I have the following utility function defined in utils.py that I would like to verify works correctly across a variety of inputs:

```
# utils.py
def to_str(data):
```

```
    if isinstance(data, str):
        return data
    elif isinstance(data, bytes):
        return data.decode('utf-8')
    else:
        raise TypeError('Must supply str or bytes, '
                        'found: %r' % data)
```

To define tests, I create a second file named test_utils.py or utils_test.py—the naming scheme you prefer is a style choice—that contains tests for each behavior that I expect:

```
# utils_test.py
from unittest import TestCase, main
from utils import to_str

class UtilsTestCase(TestCase):
    def test_to_str_bytes(self):
        self.assertEqual('hello', to_str(b'hello'))

    def test_to_str_str(self):
        self.assertEqual('hello', to_str('hello'))

    def test_failing(self):
        self.assertEqual('incorrect', to_str('hello'))

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

Then, I run the test file using the Python command line. In this case, two of the test methods pass and one fails, with a helpful error message about what went wrong:

```
$ python3 utils_test.py
F..
======================================================================
FAIL: test_failing (__main__.UtilsTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "utils_test.py", line 15, in test_failing
    self.assertEqual('incorrect', to_str('hello'))
AssertionError: 'incorrect' != 'hello'
- incorrect
+ hello


----------------------------------------------------------------------
```

```
Ran 3 tests in 0.002s

FAILED (failures=1)
```

Tests are organized into `TestCase` subclasses. Each test case is a method beginning with the word `test`. If a test method runs without raising any kind of `Exception` (including `AssertionError` from `assert` statements), the test is considered to have passed successfully. If one test fails, the `TestCase` subclass continues running the other test methods so you can get a full picture of how all your tests are doing instead of stopping at the first sign of trouble.

If you want to iterate quickly to fix or improve a specific test, you can run only that test method by specifying its path within the test module on the command line:

```
$ python3 utils_test.py UtilsTestCase.test_to_str_bytes
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

You can also invoke the debugger from directly within test methods at specific breakpoints in order to dig more deeply into the cause of failures (see Item 80: "Consider Interactive Debugging with `pdb`" for how to do that).

The `TestCase` class provides helper methods for making assertions in your tests, such as `assertEqual` for verifying equality, `assertTrue` for verifying Boolean expressions, and many more (see `help(TestCase)` for the full list). These are better than the built-in `assert` statement because they print out all of the inputs and outputs to help you understand the exact reason the test is failing. For example, here I have the same test case written with and without using a helper assertion method:

```
# assert_test.py
from unittest import TestCase, main
from utils import to_str

class AssertTestCase(TestCase):
    def test_assert_helper(self):
        expected = 12
        found = 2 * 5
        self.assertEqual(expected, found)
```

```
    def test_assert_statement(self):
        expected = 12
        found = 2 * 5
        assert expected == found

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

Which of these failure messages seems more helpful to you?

```
$ python3 assert_test.py
FF
======================================================================
FAIL: test_assert_helper (__main__.AssertTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "assert_test.py", line 16, in test_assert_helper
    self.assertEqual(expected, found)
AssertionError: 12 != 10

======================================================================
FAIL: test_assert_statement (__main__.AssertTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "assert_test.py", line 11, in test_assert_statement
    assert expected == found
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=2)
```

There's also an `assertRaises` helper method for verifying exceptions that can be used as a context manager in `with` statements (see Item 66: "Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior" for how that works). This appears similar to a `try/except` statement and makes it abundantly clear where the exception is expected to be raised:

```
# utils_error_test.py
from unittest import TestCase, main
from utils import to_str

class UtilsErrorTestCase(TestCase):
```

```
    def test_to_str_bad(self):
        with self.assertRaises(TypeError):
            to_str(object())

    def test_to_str_bad_encoding(self):
        with self.assertRaises(UnicodeDecodeError):
            to_str(b'\xfa\xfa')

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

You can define your own helper methods with complex logic in TestCase subclasses to make your tests more readable. Just ensure that your method names don't begin with the word test, or they'll be run as if they're test cases. In addition to calling TestCase assertion methods, these custom test helpers often use the fail method to clarify which assumption or invariant wasn't met. For example, here I define a custom test helper method for verifying the behavior of a generator:

```
# helper_test.py
from unittest import TestCase, main

def sum_squares(values):
    cumulative = 0
    for value in values:
        cumulative += value ** 2
        yield cumulative

class HelperTestCase(TestCase):
    def verify_complex_case(self, values, expected):
        expect_it = iter(expected)
        found_it = iter(sum_squares(values))
        test_it = zip(expect_it, found_it)

        for i, (expect, found) in enumerate(test_it):
            self.assertEqual(
                expect,
                found,
                f'Index {i} is wrong')

        # Verify both generators are exhausted
        try:
            next(expect_it)
        except StopIteration:
            pass
```

```
        else:
            self.fail('Expected longer than found')

        try:
            next(found_it)
        except StopIteration:
            pass
        else:
            self.fail('Found longer than expected')

    def test_wrong_lengths(self):
        values = [1.1, 2.2, 3.3]
        expected = [
            1.1**2,
        ]
        self.verify_complex_case(values, expected)

    def test_wrong_results(self):
        values = [1.1, 2.2, 3.3]
        expected = [
            1.1**2,
            1.1**2 + 2.2**2,
            1.1**2 + 2.2**2 + 3.3**2 + 4.4**2,
        ]
        self.verify_complex_case(values, expected)

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

The helper method makes the test cases short and readable, and the outputted error messages are easy to understand:

```
$ python3 helper_test.py
FF
======================================================================
FAIL: test_wrong_lengths (__main__.HelperTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "helper_test.py", line 43, in test_wrong_lengths
    self.verify_complex_case(values, expected)
  File "helper_test.py", line 34, in verify_complex_case
    self.fail('Found longer than expected')
AssertionError: Found longer than expected
```

```
======================================================================
FAIL: test_wrong_results (__main__.HelperTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "helper_test.py", line 52, in test_wrong_results
    self.verify_complex_case(values, expected)
  File "helper_test.py", line 24, in verify_complex_case
    f'Index {i} is wrong')
AssertionError: 36.3 != 16.939999999999998 : Index 2 is wrong

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=2)
```

I usually define one TestCase subclass for each set of related tests. Sometimes, I have one TestCase subclass for each function that has many edge cases. Other times, a TestCase subclass spans all functions in a single module. I often create one TestCase subclass for testing each basic class and all of its methods.

The TestCase class also provides a subTest helper method that enables you to avoid boilerplate by defining multiple tests within a single test method. This is especially helpful for writing data-driven tests, and it allows the test method to continue testing other cases even after one of them fails (similar to the behavior of TestCase with its contained test methods). To show this, here I define an example data-driven test:

```
# data_driven_test.py
from unittest import TestCase, main
from utils import to_str

class DataDrivenTestCase(TestCase):
    def test_good(self):
        good_cases = [
            (b'my bytes', 'my bytes'),
            ('no error', b'no error'),  # This one will fail
            ('other str', 'other str'),
            ...
        ]
        for value, expected in good_cases:
            with self.subTest(value):
                self.assertEqual(expected, to_str(value))
```

```
    def test_bad(self):
        bad_cases = [
            (object(), TypeError),
            (b'\xfa\xfa', UnicodeDecodeError),
            ...
        ]
        for value, exception in bad_cases:
            with self.subTest(value):
                with self.assertRaises(exception):
                    to_str(value)

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

The 'no error' test case fails, printing a helpful error message, but all of the other cases are still tested and confirmed to pass:

```
$ python3 data_driven_test.py
.
======================================================================
FAIL: test_good (__main__.DataDrivenTestCase) [no error]
----------------------------------------------------------------------
Traceback (most recent call last):
  File "testing/data_driven_test.py", line 18, in test_good
    self.assertEqual(expected, to_str(value))
AssertionError: b'no error' != 'no error'

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

> **Note**
>
> Depending on your project's complexity and testing requirements, the *pytest* (https://pytest.org) open source package and its large number of community plug-ins can be especially useful.

## Things to Remember

- You can create tests by subclassing the `TestCase` class from the `unittest` built-in module and defining one method per behavior you'd like to test. Test methods on `TestCase` classes must start with the word `test`.
- Use the various helper methods defined by the `TestCase` class, such as `assertEqual`, to confirm expected behaviors in your tests instead of using the built-in `assert` statement.

- Consider writing data-driven tests using the `subTest` helper method in order to reduce boilerplate.

## Item 77: Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule`, and `tearDownModule`

`TestCase` classes (see Item 76: "Verify Related Behaviors in `TestCase` Subclasses") often need to have the test environment set up before test methods can be run; this is sometimes called the *test harness*. To do this, you can override the `setUp` and `tearDown` methods of a `TestCase` subclass. These methods are called before and after each test method, respectively, so you can ensure that each test runs in isolation, which is an important best practice of proper testing.

For example, here I define a `TestCase` that creates a temporary directory before each test and deletes its contents after each test finishes:

```
# environment_test.py
from pathlib import Path
from tempfile import TemporaryDirectory
from unittest import TestCase, main

class EnvironmentTest(TestCase):
    def setUp(self):
        self.test_dir = TemporaryDirectory()
        self.test_path = Path(self.test_dir.name)

    def tearDown(self):
        self.test_dir.cleanup()

    def test_modify_file(self):
        with open(self.test_path / 'data.bin', 'w') as f:
            ...

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

When programs get complicated, you'll want additional tests to verify the end-to-end interactions between your modules instead of only testing code in isolation (using tools like mocks; see Item 78: "Use Mocks to Test Code with Complex Dependencies"). This is the difference between *unit tests* and *integration tests*. In Python, it's important to write both types of tests for exactly the same reason: You have no guarantee that your modules will actually work together unless you prove it.

One common problem is that setting up your test environment for integration tests can be computationally expensive and may require a lot of wall-clock time. For example, you might need to start a database process and wait for it to finish loading indexes before you can run your integration tests. This type of latency makes it impractical to do test preparation and cleanup for every test in the `TestCase` class's `setUp` and `tearDown` methods.

To handle this situation, the `unittest` module also supports module-level test harness initialization. You can configure expensive resources a single time, and then have all `TestCase` classes and their test methods run without repeating that initialization. Later, when all tests in the module are finished, the test harness can be torn down a single time. Here, I take advantage of this behavior by defining `setUpModule` and `tearDownModule` functions within the module containing the `TestCase` classes:

```
# integration_test.py
from unittest import TestCase, main

def setUpModule():
    print('* Module setup')

def tearDownModule():
    print('* Module clean-up')

class IntegrationTest(TestCase):
    def setUp(self):
        print('* Test setup')

    def tearDown(self):
        print('* Test clean-up')

    def test_end_to_end1(self):
        print('* Test 1')

    def test_end_to_end2(self):
        print('* Test 2')

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

```
$ python3 integration_test.py
* Module setup
* Test setup
* Test 1
```

```
* Test clean-up
.* Test setup
* Test 2
* Test clean-up
.* Module clean-up

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

I can clearly see that `setUpModule` is run by `unittest` only once, and it happens before any `setUp` methods are called. Similarly, `tearDownModule` happens after the `tearDown` method is called.

### Things to Remember

- It's important to write both unit tests (for isolated functionality) and integration tests (for modules that interact with each other).
- Use the `setUp` and `tearDown` methods to make sure your tests are isolated from each other and have a clean test environment.
- For integration tests, use the `setUpModule` and `tearDownModule` module-level functions to manage any test harnesses you need for the entire lifetime of a test module and all of the `TestCase` classes that it contains.

## Item 78: Use Mocks to Test Code with Complex Dependencies

Another common need when writing tests (see Item 76: "Verify Related Behaviors in `TestCase` Subclasses") is to use mocked functions and classes to simulate behaviors when it's too difficult or slow to use the real thing. For example, say that I need a program to maintain the feeding schedule for animals at the zoo. Here, I define a function to query a database for all of the animals of a certain species and return when they most recently ate:

```
class DatabaseConnection:
    ...

def get_animals(database, species):
    # Query the database
    ...
    # Return a list of (name, last_mealtime) tuples
```

How do I get a DatabaseConnection instance to use for testing this function? Here, I try to create one and pass it into the function being tested:

```
database = DatabaseConnection('localhost', '4444')

get_animals(database, 'Meerkat')
```

```
>>>
Traceback ...
DatabaseConnectionError: Not connected
```

There's no database running, so of course this fails. One solution is to actually stand up a database server and connect to it in the test. However, it's a lot of work to fully automate starting up a database, configuring its schema, populating it with data, and so on in order to just run a simple unit test. Further, it will probably take a lot of wall-clock time to set up a database server, which would slow down these unit tests and make them harder to maintain.

A better approach is to mock out the database. A *mock* lets you provide expected responses for dependent functions, given a set of expected calls. It's important not to confuse mocks with fakes. A *fake* would provide most of the behavior of the DatabaseConnection class but with a simpler implementation, such as a basic in-memory, single-threaded database with no persistence.

Python has the unittest.mock built-in module for creating mocks and using them in tests. Here, I define a Mock instance that simulates the get_animals function without actually connecting to the database:

```
from datetime import datetime
from unittest.mock import Mock

mock = Mock(spec=get_animals)
expected = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 45)),
]
mock.return_value = expected
```

The Mock class creates a mock function. The return_value attribute of the mock is the value to return when it is called. The spec argument indicates that the mock should act like the given object, which is a function in this case, and error if it's used in the wrong way.

For example, here I try to treat the mock function as if it were a mock object with attributes:

```
mock.does_not_exist
```

```
>>>
Traceback ...
AttributeError: Mock object has no attribute 'does_not_exist'
```

Once it's created, I can call the mock, get its return value, and verify that what it returns matches expectations. I use a unique `object` value as the `database` argument because it won't actually be used by the mock to do anything; all I care about is that the `database` parameter was correctly plumbed through to any dependent functions that needed a `DatabaseConnection` instance in order to work (see Item 55: "Use `Queue` to Coordinate Work Between Threads" for another example of using sentinel `object` instances):

```
database = object()
result = mock(database, 'Meerkat')
assert result == expected
```

This verifies that the mock responded correctly, but how do I know if the code that called the mock provided the correct arguments? For this, the `Mock` class provides the `assert_called_once_with` method, which verifies that a single call with exactly the given parameters was made:

```
mock.assert_called_once_with(database, 'Meerkat')
```

If I supply the wrong parameters, an exception is raised, and any `TestCase` that the assertion is used in fails:

```
mock.assert_called_once_with(database, 'Giraffe')
```

```
>>>
Traceback ...
AssertionError: expected call not found.
Expected: mock(<object object at 0x109038790>, 'Giraffe')
Actual: mock(<object object at 0x109038790>, 'Meerkat')
```

If I actually don't care about some of the individual parameters, such as exactly which `database` object was used, then I can indicate that any value is okay for an argument by using the `unittest.mock.ANY` constant. I can also use the `assert_called_with` method of `Mock` to verify that the most recent call to the mock—and there may have been multiple calls in this case—matches my expectations:

```
from unittest.mock import ANY
```

```
mock = Mock(spec=get_animals)
mock('database 1', 'Rabbit')
mock('database 2', 'Bison')
mock('database 3', 'Meerkat')

mock.assert_called_with(ANY, 'Meerkat')
```

ANY is useful in tests when a parameter is not core to the behavior that's being tested. It's often worth erring on the side of under-specifying tests by using ANY more liberally instead of over-specifying tests and having to plumb through various test parameter expectations.

The Mock class also makes it easy to mock exceptions being raised:

```
class MyError(Exception):
    pass

mock = Mock(spec=get_animals)
mock.side_effect = MyError('Whoops! Big problem')
result = mock(database, 'Meerkat')
```

```
>>>
Traceback ...
MyError: Whoops! Big problem
```

There are many more features available, so be sure to see help(unittest.mock.Mock) for the full range of options.

Now that I've shown the mechanics of how a Mock works, I can apply it to an actual testing situation to show how to use it effectively in writing unit tests. Here, I define a function to do the rounds of feeding animals at the zoo, given a set of database-interacting functions:

```
def get_food_period(database, species):
    # Query the database
    ...
    # Return a time delta

def feed_animal(database, name, when):
    # Write to the database
    ...

def do_rounds(database, species):
    now = datetime.datetime.utcnow()
    feeding_timedelta = get_food_period(database, species)
    animals = get_animals(database, species)
    fed = 0
```

```
    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_animal(database, name, now)
            fed += 1

    return fed
```

The goal of my test is to verify that when `do_rounds` is run, the right animals got fed, the latest feeding time was recorded to the database, and the total number of animals fed returned by the function matches the correct total. In order to do all this, I need to mock out `datetime.utcnow` so my tests have a stable time that isn't affected by daylight saving time and other ephemeral changes. I need to mock out `get_food_period` and `get_animals` to return values that would have come from the database. And I need to mock out `feed_animal` to accept data that would have been written back to the database.

The question is: Even if I know how to create these mock functions and set expectations, how do I get the `do_rounds` function that's being tested to use the mock dependent functions instead of the real versions? One approach is to inject everything as keyword-only arguments (see Item 25: "Enforce Clarity with Keyword-Only and Positional-Only Arguments"):

```
def do_rounds(database, species, *,
              now_func=datetime.utcnow,
              food_func=get_food_period,
              animals_func=get_animals,
              feed_func=feed_animal):
    now = now_func()
    feeding_timedelta = food_func(database, species)
    animals = animals_func(database, species)
    fed = 0

    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_func(database, name, now)
            fed += 1

    return fed
```

To test this function, I need to create all of the `Mock` instances upfront and set their expectations:

```
from datetime import timedelta
```

```
now_func = Mock(spec=datetime.utcnow)
now_func.return_value = datetime(2019, 6, 5, 15, 45)

food_func = Mock(spec=get_food_period)
food_func.return_value = timedelta(hours=3)

animals_func = Mock(spec=get_animals)
animals_func.return_value = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 45)),
]

feed_func = Mock(spec=feed_animal)
```

Then, I can run the test by passing the mocks into the `do_rounds` function to override the defaults:

```
result = do_rounds(
    database,
    'Meerkat',
    now_func=now_func,
    food_func=food_func,
    animals_func=animals_func,
    feed_func=feed_func)

assert result == 2
```

Finally, I can verify that all of the calls to dependent functions matched my expectations:

```
from unittest.mock import call

food_func.assert_called_once_with(database, 'Meerkat')

animals_func.assert_called_once_with(database, 'Meerkat')

feed_func.assert_has_calls(
    [
        call(database, 'Spot', now_func.return_value),
        call(database, 'Fluffy', now_func.return_value),
    ],
    any_order=True)
```

I don't verify the parameters to the `datetime.utcnow` mock or how many times it was called because that's indirectly verified by the return value of the function. For `get_food_period` and `get_animals`, I verify a single call with the specified parameters by using `assert_called_once_with`.

For the feed_animal function, I verify that two calls were made—and their order didn't matter—to write to the database using the unittest.mock.call helper and the assert_has_calls method.

This approach of using keyword-only arguments for injecting mocks works, but it's quite verbose and requires changing every function you want to test. The unittest.mock.patch family of functions makes injecting mocks easier. It temporarily reassigns an attribute of a module or class, such as the database-accessing functions that I defined above. For example, here I override get_animals to be a mock using patch:

```
from unittest.mock import patch

print('Outside patch:', get_animals)

with patch('__main__.get_animals'):
    print('Inside patch: ', get_animals)

print('Outside again:', get_animals)
```

```
>>>
Outside patch: <function get_animals at 0x109217040>
Inside patch:  <MagicMock name='get_animals' id='4454622832'>
Outside again: <function get_animals at 0x109217040>
```

patch works for many modules, classes, and attributes. It can be used in with statements (see Item 66: "Consider contextlib and with Statements for Reusable try/finally Behavior"), as a function decorator (see Item 26: "Define Function Decorators with functools.wraps"), or in the setUp and tearDown methods of TestCase classes (see Item 76: "Verify Related Behaviors in TestCase Subclasses"). For the full range of options, see help(unittest.mock.patch).

However, patch doesn't work in all cases. For example, to test do_rounds I need to mock out the current time returned by the datetime.utcnow class method. Python won't let me do that because the datetime class is defined in a C-extension module, which can't be modified in this way:

```
fake_now = datetime(2019, 6, 5, 15, 45)

with patch('datetime.datetime.utcnow'):
    datetime.utcnow.return_value = fake_now
```

```
>>>
Traceback ...
TypeError: can't set attributes of built-in/extension type
➥'datetime.datetime'
```

To work around this, I can create another helper function to fetch time that can be patched:

```
def get_do_rounds_time():
    return datetime.datetime.utcnow()

def do_rounds(database, species):
    now = get_do_rounds_time()
    ...

with patch('__main__.get_do_rounds_time'):
    ...
```

Alternatively, I can use a keyword-only argument for the `datetime.utcnow` mock and use `patch` for all of the other mocks:

```
def do_rounds(database, species, *, utcnow=datetime.utcnow):
    now = utcnow()
    feeding_timedelta = get_food_period(database, species)
    animals = get_animals(database, species)
    fed = 0

    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_func(database, name, now)
            fed += 1

    return fed
```

I'm going to go with the latter approach. Now, I can use the `patch.multiple` function to create many mocks and set their expectations:

```
from unittest.mock import DEFAULT

with patch.multiple('__main__',
                    autospec=True,
                    get_food_period=DEFAULT,
                    get_animals=DEFAULT,
                    feed_animal=DEFAULT):
    now_func = Mock(spec=datetime.utcnow)
    now_func.return_value = datetime(2019, 6, 5, 15, 45)
    get_food_period.return_value = timedelta(hours=3)
    get_animals.return_value = [
        ('Spot', datetime(2019, 6, 5, 11, 15)),
        ('Fluffy', datetime(2019, 6, 5, 12, 30)),
        ('Jojo', datetime(2019, 6, 5, 12, 45))
    ]
```

With the setup ready, I can run the test and verify that the calls were correct inside the `with` statement that used `patch.multiple`:

```
result = do_rounds(database, 'Meerkat', utcnow=now_func)
assert result == 2

food_func.assert_called_once_with(database, 'Meerkat')
animals_func.assert_called_once_with(database, 'Meerkat')
feed_func.assert_has_calls(
    [
        call(database, 'Spot', now_func.return_value),
        call(database, 'Fluffy', now_func.return_value),
    ],
    any_order=True)
```

The keyword arguments to `patch.multiple` correspond to names in the `__main__` module that I want to override during the test. The `DEFAULT` value indicates that I want a standard `Mock` instance to be created for each name. All of the generated mocks will adhere to the specification of the objects they are meant to simulate, thanks to the `autospec=True` parameter.

These mocks work as expected, but it's important to realize that it's possible to further improve the readability of these tests and reduce boilerplate by refactoring your code to be more testable (see Item 79: "Encapsulate Dependencies to Facilitate Mocking and Testing").

### Things to Remember

- The `unittest.mock` module provides a way to simulate the behavior of interfaces using the `Mock` class. Mocks are useful in tests when it's difficult to set up the dependencies that are required by the code that's being tested.
- When using mocks, it's important to verify both the behavior of the code being tested and how dependent functions were called by that code, using the `Mock.assert_called_once_with` family of methods.
- Keyword-only arguments and the `unittest.mock.patch` family of functions can be used to inject mocks into the code being tested.

## Item 79: Encapsulate Dependencies to Facilitate Mocking and Testing

In the previous item (see Item 78: "Use Mocks to Test Code with Complex Dependencies"), I showed how to use the facilities of the `unittest.mock` built-in module—including the `Mock` class and `patch`

family of functions—to write tests that have complex dependencies, such as a database. However, the resulting test code requires a lot of boilerplate, which could make it more difficult for new readers of the code to understand what the tests are trying to verify.

One way to improve these tests is to use a wrapper object to encapsulate the database's interface instead of passing a `DatabaseConnection` object to functions as an argument. It's often worth refactoring your code (see Item 89: "Consider `warnings` to Refactor and Migrate Usage" for one approach) to use better abstractions because it facilitates creating mocks and writing tests. Here, I redefine the various database helper functions from the previous item as methods on a class instead of as independent functions:

```
class ZooDatabase:
    ...

    def get_animals(self, species):
        ...

    def get_food_period(self, species):
        ...

    def feed_animal(self, name, when):
        ...
```

Now, I can redefine the `do_rounds` function to call methods on a `ZooDatabase` object:

```
from datetime import datetime

def do_rounds(database, species, *, utcnow=datetime.utcnow):
    now = utcnow()
    feeding_timedelta = database.get_food_period(species)
    animals = database.get_animals(species)
    fed = 0

    for name, last_mealtime in animals:
        if (now - last_mealtime) >= feeding_timedelta:
            database.feed_animal(name, now)
            fed += 1

    return fed
```

Writing a test for `do_rounds` is now a lot easier because I no longer need to use `unittest.mock.patch` to inject the mock into the code being tested. Instead, I can create a `Mock` instance to represent

a ZooDatabase and pass that in as the database parameter. The Mock class returns a mock object for any attribute name that is accessed. Those attributes can be called like methods, which I can then use to set expectations and verify calls. This makes it easy to mock out all of the methods of a class:

```
from unittest.mock import Mock

database = Mock(spec=ZooDatabase)
print(database.feed_animal)
database.feed_animal()
database.feed_animal.assert_any_call()
```

```
>>>
<Mock name='mock.feed_animal' id='4384773408'>
```

I can rewrite the Mock setup code by using the ZooDatabase encapsulation:

```
from datetime import timedelta
from unittest.mock import call

now_func = Mock(spec=datetime.utcnow)
now_func.return_value = datetime(2019, 6, 5, 15, 45)

database = Mock(spec=ZooDatabase)
database.get_food_period.return_value = timedelta(hours=3)
database.get_animals.return_value = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 55))
]
```

Then I can run the function being tested and verify that all dependent methods were called as expected:

```
result = do_rounds(database, 'Meerkat', utcnow=now_func)
assert result == 2

database.get_food_period.assert_called_once_with('Meerkat')
database.get_animals.assert_called_once_with('Meerkat')
database.feed_animal.assert_has_calls(
    [
        call('Spot', now_func.return_value),
        call('Fluffy', now_func.return_value),
    ],
    any_order=True)
```

Using the `spec` parameter to `Mock` is especially useful when mocking classes because it ensures that the code under test doesn't call a misspelled method name by accident. This allows you to avoid a common pitfall where the same bug is present in both the code and the unit test, masking a real error that will later reveal itself in production:

```
database.bad_method_name()
```

```
>>>
Traceback ...
AttributeError: Mock object has no attribute 'bad_method_name'
```

If I want to test this program end-to-end with a mid-level integration test (see Item 77: "Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule`, and `tearDownModule`"), I still need a way to inject a mock `ZooDatabase` into the program. I can do this by creating a helper function that acts as a seam for *dependency injection*. Here, I define such a helper function that caches a `ZooDatabase` in module scope (see Item 86: "Consider Module-Scoped Code to Configure Deployment Environments") by using a `global` statement:

```
DATABASE = None

def get_database():
    global DATABASE
    if DATABASE is None:
        DATABASE = ZooDatabase()
    return DATABASE

def main(argv):
    database = get_database()
    species = argv[1]
    count = do_rounds(database, species)
    print(f'Fed {count} {species}(s)')
    return 0
```

Now, I can inject the mock `ZooDatabase` using `patch`, run the test, and verify the program's output. I'm not using a mock `datetime.utcnow` here; instead, I'm relying on the database records returned by the mock to be relative to the current time in order to produce similar behavior to the unit test. This approach is more flaky than mocking everything, but it also tests more surface area:

```
import contextlib
import io
from unittest.mock import patch
```

```
with patch('__main__.DATABASE', spec=ZooDatabase):
    now = datetime.utcnow()

    DATABASE.get_food_period.return_value = timedelta(hours=3)
    DATABASE.get_animals.return_value = [
        ('Spot', now - timedelta(minutes=4.5)),
        ('Fluffy', now - timedelta(hours=3.25)),
        ('Jojo', now - timedelta(hours=3)),
    ]

    fake_stdout = io.StringIO()
    with contextlib.redirect_stdout(fake_stdout):
        main(['program name', 'Meerkat'])

    found = fake_stdout.getvalue()
    expected = 'Fed 2 Meerkat(s)\n'

    assert found == expected
```

The results match my expectations. Creating this integration test was straightforward because I designed the implementation to make it easier to test.

### Things to Remember

- When unit tests require a lot of repeated boilerplate to set up mocks, one solution may be to encapsulate the functionality of dependencies into classes that are more easily mocked.
- The `Mock` class of the `unittest.mock` built-in module simulates classes by returning a new mock, which can act as a mock method, for each attribute that is accessed.
- For end-to-end tests, it's valuable to refactor your code to have more helper functions that can act as explicit seams to use for injecting mock dependencies in tests.

## Item 80: Consider Interactive Debugging with pdb

Everyone encounters bugs in code while developing programs. Using the `print` function can help you track down the sources of many issues (see Item 75: "Use `repr` Strings for Debugging Output"). Writing tests for specific cases that cause trouble is another great way to isolate problems (see Item 76: "Verify Related Behaviors in `TestCase` Subclasses").

But these tools aren't enough to find every root cause. When you need something more powerful, it's time to try Python's built-in *interactive debugger*. The debugger lets you inspect program state, print local variables, and step through a Python program one statement at a time.

In most other programming languages, you use a debugger by specifying what line of a source file you'd like to stop on, and then execute the program. In contrast, with Python, the easiest way to use the debugger is by modifying your program to directly initiate the debugger just before you think you'll have an issue worth investigating. This means that there is no difference between starting a Python program in order to run the debugger and starting it normally.

To initiate the debugger, all you have to do is call the `breakpoint` built-in function. This is equivalent to importing the `pdb` built-in module and running its `set_trace` function:

```
# always_breakpoint.py
import math

def compute_rmse(observed, ideal):
    total_err_2 = 0
    count = 0
    for got, wanted in zip(observed, ideal):
        err_2 = (got - wanted) ** 2
        breakpoint()  # Start the debugger here
        total_err_2 += err_2
        count += 1

    mean_err = total_err_2 / count
    rmse = math.sqrt(mean_err)
    return rmse

result = compute_rmse(
    [1.8, 1.7, 3.2, 6],
    [2, 1.5, 3, 5])
print(result)
```

As soon as the `breakpoint` function runs, the program pauses its execution before the line of code immediately following the `breakpoint` call. The terminal that started the program turns into an interactive Python shell:

```
$ python3 always_breakpoint.py
> always_breakpoint.py(12)compute_rmse()
-> total_err_2 += err_2
(Pdb)
```

At the (Pdb) prompt, you can type in the names of local variables to see their values printed out (or use `p <name>`). You can see a list of all local variables by calling the `locals` built-in function. You can import modules, inspect global state, construct new objects, run the `help` built-in function, and even modify parts of the running program—whatever you need to do to aid in your debugging.

In addition, the debugger has a variety of special commands to control and understand program execution; type `help` to see the full list.

Three very useful commands make inspecting the running program easier:

- `where`: Print the current execution call stack. This lets you figure out where you are in your program and how you arrived at the `breakpoint` trigger.
- `up`: Move your scope up the execution call stack to the caller of the current function. This allows you to inspect the local variables in higher levels of the program that led to the breakpoint.
- `down`: Move your scope back down the execution call stack one level.

When you're done inspecting the current state, you can use these five debugger commands to control the program's execution in different ways:

- `step`: Run the program until the next line of execution in the program, and then return control back to the debugger prompt. If the next line of execution includes calling a function, the debugger stops within the function that was called.
- `next`: Run the program until the next line of execution in the current function, and then return control back to the debugger prompt. If the next line of execution includes calling a function, the debugger will not stop until the called function has returned.
- `return`: Run the program until the current function returns, and then return control back to the debugger prompt.
- `continue`: Continue running the program until the next breakpoint is hit (either through the `breakpoint` call or one added by a debugger command).
- `quit`: Exit the debugger and end the program. Run this command if you've found the problem, gone too far, or need to make program modifications and try again.

The `breakpoint` function can be called anywhere in a program. If you know that the problem you're trying to debug happens only under special circumstances, then you can just write plain old Python code to call `breakpoint` after a specific condition is met. For example, here I start the debugger only if the squared error for a datapoint is more than 1:

```
# conditional_breakpoint.py
def compute_rmse(observed, ideal):
    ...
    for got, wanted in zip(observed, ideal):
        err_2 = (got - wanted) ** 2
        if err_2 >= 1:  # Start the debugger if True
            breakpoint()
        total_err_2 += err_2
        count += 1
    ...
result = compute_rmse(
    [1.8, 1.7, 3.2, 7],
    [2, 1.5, 3, 5])
print(result)
```

When I run the program and it enters the debugger, I can confirm that the condition was true by inspecting local variables:

```
$ python3 conditional_breakpoint.py
> conditional_breakpoint.py(14)compute_rmse()
-> total_err_2 += err_2
(Pdb) wanted
5
(Pdb) got
7
(Pdb) err_2
4
```

Another useful way to reach the debugger prompt is by using *post-mortem debugging*. This enables you to debug a program *after* it's already raised an exception and crashed. This is especially helpful when you're not quite sure where to put the `breakpoint` function call.

Here, I have a script that will crash due to the `7j` complex number being present in one of the function's arguments:

```
# postmortem_breakpoint.py
import math

def compute_rmse(observed, ideal):
    ...
```

```
result = compute_rmse(
    [1.8, 1.7, 3.2, 7j],  # Bad input
    [2, 1.5, 3, 5])
print(result)
```

I use the command line `python3 -m pdb -c continue <program path>` to run the program under control of the `pdb` module. The `continue` command tells `pdb` to get the program started immediately. Once it's running, the program hits a problem and automatically enters the interactive debugger, at which point I can inspect the program state:

```
$ python3 -m pdb -c continue postmortem_breakpoint.py
Traceback (most recent call last):
  File ".../pdb.py", line 1697, in main
    pdb._runscript(mainpyfile)
  File ".../pdb.py", line 1566, in _runscript
    self.run(statement)
  File ".../bdb.py", line 585, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "postmortem_breakpoint.py", line 4, in <module>
    import math
  File "postmortem_breakpoint.py", line 16, in compute_rmse
    rmse = math.sqrt(mean_err)
TypeError: can't convert complex to float
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> postmortem_breakpoint.py(16)compute_rmse()
-> rmse = math.sqrt(mean_err)
(Pdb) mean_err
(-5.97-17.5j)
```

You can also use post-mortem debugging after hitting an uncaught exception in the interactive Python interpreter by calling the `pm` function of the `pdb` module (which is often done in a single line as `import pdb; pdb.pm()`):

```
$ python3
>>> import my_module
>>> my_module.compute_stddev([5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "my_module.py", line 17, in compute_stddev
    variance = compute_variance(data)
  File "my_module.py", line 13, in compute_variance
    variance = err_2_sum / (len(data) - 1)
```

```
ZeroDivisionError: float division by zero
>>> import pdb; pdb.pm()
> my_module.py(13)compute_variance()
-> variance = err_2_sum / (len(data) - 1)
(Pdb) err_2_sum
0.0
(Pdb) len(data)
1
```

### Things to Remember

- You can initiate the Python interactive debugger at a point of interest directly in your program by calling the `breakpoint` built-in function.
- The Python debugger prompt is a full Python shell that lets you inspect and modify the state of a running program.
- `pdb` shell commands let you precisely control program execution and allow you to alternate between inspecting program state and progressing program execution.
- The `pdb` module can be used for debug exceptions after they happen in independent Python programs (using `python -m pdb -c continue <program path>`) or the interactive Python interpreter (using `import pdb; pdb.pm()`).

## Item 81: Use `tracemalloc` to Understand Memory Usage and Leaks

Memory management in the default implementation of Python, CPython, uses reference counting. This ensures that as soon as all references to an object have expired, the referenced object is also cleared from memory, freeing up that space for other data. CPython also has a built-in cycle detector to ensure that self-referencing objects are eventually garbage collected.

In theory, this means that most Python programmers don't have to worry about allocating or deallocating memory in their programs. It's taken care of automatically by the language and the CPython runtime. However, in practice, programs eventually do run out of memory due to no longer useful references still being held. Figuring out where a Python program is using or leaking memory proves to be a challenge.

The first way to debug memory usage is to ask the `gc` built-in module to list every object currently known by the garbage collector. Although

it's quite a blunt tool, this approach lets you quickly get a sense of where your program's memory is being used.

Here, I define a module that fills up memory by keeping references:

```
# waste_memory.py
import os

class MyObject:
    def __init__(self):
        self.data = os.urandom(100)

def get_data():
    values = []
    for _ in range(100):
        obj = MyObject()
        values.append(obj)
    return values

def run():
    deep_values = []
    for _ in range(100):
        deep_values.append(get_data())
    return deep_values
```

Then, I run a program that uses the gc built-in module to print out how many objects were created during execution, along with a small sample of allocated objects:

```
# using_gc.py
import gc

found_objects = gc.get_objects()
print('Before:', len(found_objects))

import waste_memory

hold_reference = waste_memory.run()

found_objects = gc.get_objects()
print('After: ', len(found_objects))
for obj in found_objects[:3]:
    print(repr(obj)[:100])
```

```
>>>
Before: 6207
After:  16801
```

```
<waste_memory.MyObject object at 0x10390aeb8>
<waste_memory.MyObject object at 0x10390aef0>
<waste_memory.MyObject object at 0x10390af28>
...
```

The problem with `gc.get_objects` is that it doesn't tell you anything about *how* the objects were allocated. In complicated programs, objects of a specific class could be allocated many different ways. Knowing the overall number of objects isn't nearly as important as identifying the code responsible for allocating the objects that are leaking memory.

Python 3.4 introduced a new `tracemalloc` built-in module for solving this problem. `tracemalloc` makes it possible to connect an object back to where it was allocated. You use it by taking before and after snapshots of memory usage and comparing them to see what's changed. Here, I use this approach to print out the top three memory usage offenders in a program:

```
# top_n.py
import tracemalloc

tracemalloc.start(10)                       # Set stack depth
time1 = tracemalloc.take_snapshot()         # Before snapshot

import waste_memory

x = waste_memory.run()                      # Usage to debug
time2 = tracemalloc.take_snapshot()         # After snapshot

stats = time2.compare_to(time1, 'lineno')  # Compare snapshots
for stat in stats[:3]:
    print(stat)
```

```
>>>
waste_memory.py:5: size=2392 KiB (+2392 KiB), count=29994
➥(+29994), average=82 B
waste_memory.py:10: size=547 KiB (+547 KiB), count=10001
➥(+10001), average=56 B
waste_memory.py:11: size=82.8 KiB (+82.8 KiB), count=100
➥(+100), average=848 B
```

The size and count labels in the output make it immediately clear which objects are dominating my program's memory usage and where in the source code they were allocated.

The tracemalloc module can also print out the full stack trace of each allocation (up to the number of frames passed to the tracemalloc.start function). Here, I print out the stack trace of the biggest source of memory usage in the program:

```
# with_trace.py
import tracemalloc

tracemalloc.start(10)
time1 = tracemalloc.take_snapshot()

import waste_memory

x = waste_memory.run()
time2 = tracemalloc.take_snapshot()

stats = time2.compare_to(time1, 'traceback')
top = stats[0]
print('Biggest offender is:')
print('\n'.join(top.traceback.format()))
```

```
>>>
Biggest offender is:
  File "with_trace.py", line 9
    x = waste_memory.run()
  File "waste_memory.py", line 17
    deep_values.append(get_data())
  File "waste_memory.py", line 10
    obj = MyObject()
  File "waste_memory.py", line 5
    self.data = os.urandom(100)
```

A stack trace like this is most valuable for figuring out which particular usage of a common function or class is responsible for memory consumption in a program.

## Things to Remember

- It can be difficult to understand how Python programs use and leak memory.
- The gc module can help you understand which objects exist, but it has no information about how they were allocated.
- The tracemalloc built-in module provides powerful tools for understanding the sources of memory usage.

*This page intentionally left blank*

# 10 Collaboration

Python has language features that help you construct well-defined APIs with clear interface boundaries. The Python community has established best practices to maximize the maintainability of code over time. In addition, some standard tools that ship with Python enable large teams to work together across disparate environments.

Collaborating with others on Python programs requires being deliberate in how you write your code. Even if you're working on your own, chances are you'll be using code written by someone else via the standard library or open source packages. It's important to understand the mechanisms that make it easy to collaborate with other Python programmers.

## Item 82: Know Where to Find Community-Built Modules

Python has a central repository of modules (https://pypi.org) that you can install and use in your programs. These modules are built and maintained by people like you: the Python community. When you find yourself facing an unfamiliar challenge, the Python Package Index (PyPI) is a great place to look for code that will get you closer to your goal.

To use the Package Index, you need to use the command-line tool `pip` (a recursive acronym for "pip installs packages"). `pip` can be run with `python3 -m pip` to ensure that packages are installed for the correct version of Python on your system (see Item 1: "Know Which Version of Python You're Using"). Using `pip` to install a new module is simple. For example, here I install the `pytz` module that I use elsewhere in this book (see Item 67: "Use `datetime` Instead of `time` for Local Clocks"):

```
$ python3 -m pip install pytz
Collecting pytz
```