Chapter 09 - Testing and Debugging

May 8, 2021

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

- for most things, all you need to do is call `print` to see how the state of your program changes while it runs to understand where it goes wrong
- the `print` function outputs a human-readable string version of whatever you supply it

[1]:
```
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 human-redable string for a value does not make it clear what the actual type and its specific compositions are
- for example
- for example `print(5)` will not tell you if its an `int` or a `str`

- what you almost always want while debugging is to see the `repr` verision of an object
- the `repr` built-in function returns the printable representation of an object, which should be its most clearly understandable string representation

[2]:
```
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

[3]:
```
b = eval(repr(a))
assert a == b
```

- when you are debugging with `print` you should call `repr` on a value before printing to ensure that any difference in types is clear

[4]:
```
print(repr(5))
print(repr('5'))
```

```
5
'5'
```

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

[6]:
```
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-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 dont need to explicitly call `repr` on it
- unfortunately, the default implementation of the `repr` for `object` subclass isnt espically helpful

[12]:
```
class OpaqueClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

```
<__main__.OpaqueClass object at 0x000001F233B92340>
```

- the output cant be passed to the `eval` function, and it says nothing about the instance fields of the object

- there are two solution
- you can define your own `__repr__` special method that returns a string containing the Pythone expression that re-creates the objec

[13]:
```
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})'

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

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

- when you dont have control over the class definition, you can reach into the object instance dictionary, which is stored in the `__dict__` attribute

[11]:
```
obj = OpaqueClass(4, 'baz')
print(obj.__dict__)
```

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

0.2 Item 76 Verify Related Behaviors in `TestCase` Subclasses

- suppose we gave a `utils` function we would like to verify works correctly across a variety of inputs

[14]:
```
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)
```

[17]:
```
from unittest import TestCase, main


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'))
```

- 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` 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 of 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
    - `python3 utils_test.py UtilsTestCase.test_to_str_bytes`
- you can also invoke the debugger from directly withing test methods at specifc breakpoints in order to dig more deeply into the cause of failure
- `TestCase` class provides helper methods for making assertions in your tests, such as `assertEqual` for verifying equality, `assertTrue` for verifying Boolean expressions and others by typeing `help(TeseCase)`
- there is also `assertRaises` helper method for verifying exceptions that can be used as a context manager in `with` statements
- this appears similiar to a `try/except` statement and makes it abundantly clear where the exception is expected to be raised

[18]:
```python
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')
```

- you can define your own helper methods with compelx logic in `TestCase` subclasses to make your tests more readable
- just ensure that the method names dont begin with the word `test`

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

- it is good to define one `TestCase` subclass for each set of related tests
- sometimes you can have one `TestCase` subclass for each function that has many edge cases
- other times we 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 allows the test method to continue testing cases even after one oif them fails

[20]:
```python
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)
```

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

- you can override the `setUp` and the `tearDown` methods of `TestCase` subclasses
- these methods are called before and after each test method so that you can ensure that each test runs in isolation, which is an important best pratice of proper testing

- below we define a `TestCase` that creates a temporary dictionary before each test and deleted its contents after each test finishes

[22]:
```
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:
            ...
```

- when programs get complicated you want additional tests to verify the end-to-end interactions between your modules instead of only testing code in isolation
- this is the difference between `unit tests` and `integration tests`
- in python, its important to write both tyes 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 enviorment for integration tests can be computationally expensive and may require alot of wall-clock time
- for example, say you have to start up the database and tear it down every time
- its not praticle to do this for every `setUp` or `tearDown`

- `unittest` library provides you ways to configure an expensive resource a single time and then all `TestCase` classes and their test methods run without repeating that initialization
- the methods we can use are `setUpModule` and `teardownModule`

[23]:
```python
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')
```

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

- when writting tests, its common to use mocked functions and classes to simulate behavior when its too difficult or slow to use the real thing
- for example, lets say we need a program to maintain the feeding schedule for animals at the zoo
- we define a function to query a database for all the animals of a certain species and return when they most recently ate

[50]:
```python
class DatabaseConnection:
    ...

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

- to test this we could create a database and populate it with data but that takes alot of wall clock time
- its better to use mocks
- a `mock` lets you provide expected responses for dependent functions, given a set of expected call
- a `mock` is not a `fake`

  - a fake would provide most of the behavior of the DatabaseConnection class but with a simpler implementation, such as 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 we define a Mock instance that simulates the get_animals functions without actually connecting to the database

[75]:
```python
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 its used in the wrong way

- if we try to trat the mock function ad if it were a mock object with attributes, we get errors

[77]:
```python
try:
    mock.does_not_exist
except AttributeError as e:
    print(e)
```

```
Mock object has no attribute 'does_not_exist'
```

- once its created we can call the mock, get its return value and verify that what it returns matches expecations
- we use a unique object value as the database argument because it wont actually be used by the mock to do anything
- all we care about is that the database parameter was correctly plumbed through to any dependent functions that needed a DatabaseConenction instance in other work

[79]:
```python
database = object()
result = mock(database, 'Meerkat')
assert result == expected
```

- the code snippet above verifies that the mock responed correctly, but how do we 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 signle call with exactly the given parameters was made

[81]:
```
try:
    mock.assert_called_once_with(database, 'Meerkat')
except AssertionError as e:
    print(e)
```

```
Expected 'mock' to be called once. Called 2 times.
Calls: [call(<object object at 0x000001E074F8BF30>, 'Meerkat'),
 call(<object object at 0x000001E074F8BF40>, 'Meerkat')].
```

- if we supply the wrong parameters, an expection is raised and any `TestCase` that the assertion in used in fails

[82]:
```
try:
    mock.assert_called_once_with(database, 'Giraffe')
except AssertionError as e:
    print(e)
```

```
Expected 'mock' to be called once. Called 2 times.
Calls: [call(<object object at 0x000001E074F8BF30>, 'Meerkat'),
 call(<object object at 0x000001E074F8BF40>, 'Meerkat')].
```

- if you dont care about some individual parameters, such as exactly which `database` object was used, then we can indicate that any value is okay for an argument by using the `unittest.mock.ANY` constant
- we 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

[84]:
```
from unittest.mock import ANY

try:
    mock = Mock(spec=get_animals)
    mock('database 1', 'Rabbit')
    mock('database 2', 'Bison')
    mock('database 3', 'Meerkat')
    mock.assert_called_with(ANY, 'Meerkat')
except AssertionError as e:
    print(e)
```

- `ANY` is useful in tests when a parameter is not core to the behavior thats being tested
- its often work erring on the side of under-specifying tests used by `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
- all you have to do is use `side_effect`

[85]:
```
from unittest.mock import Mock

class MyError(Exception):
    pass
```

```python
try:
    mock = Mock(spec=get_animals)
    mock.side_effect = MyError('Whoops! Big problem')
    result = mock(database, 'Meerkat')
except MyError as e:
    print(e)
```

```
Whoops! Big problem
```

- use `help(unittest.mock.Mock)` to learn more
- below is an example of how to apply `Mock` to actual testing situations to show how to use it effectively in writing unit tests

[86]:
```python
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 foal of my testr is to verify that when `do_rounds` is run, the right animals get 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
- to do all this, we need to mock out `datetime.utcnow` so my tests have a stable time that isn't affected by daylight saving time and other ephermeral changes
- we need to mock out `get_food_period` and `get_animals` to return values that would hae come from the database
- all we need to mock out `feed_animal` to accept data that would have been written back to the database
- we also need to mock out `feed_animal` to accept data that would have been written back to the database
- the problem is even if we know how to create these mock functions and set expectations, how do we get the `do_round` function thats being tested to use the mock dependent functions instead of the real versions?

- one approach is to inject everything as keyword-only arguments

[87]:
```
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 `Mocks` instances upfront and set their expectantions:

[88]:
```
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 we can run the test by passing the mocks into the `do_rounds` function to override the defaults

[89]:
```
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 we can verify that all the calls to dependent functions matched our expectations

[90]:
```
from unittest.mock import call

try:
    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)
except AssertionError as e:
    print(e)
```

- we dont verify the parameters to the `datetime.utcnow` mock or how many times it was called because its indirectly verified by the return value of the function
- for `get_food_period` and `get_animals`, we verify a single call with the specified parameters by using `assert_called_once_with`
- for the `feed_animal` function we verify that two calls were amade- 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 using keyword-only arguments for injecting mocks works, buits its very verbose and requires changing every function you want to test
- the `unittest.mock.patch` family of functions makes injecting mockes easier
- it temporarily reassigns an attribute of a module or class, such as the database-accessing functions that we defined above

- for example, here we can override `get_animals` to be a mock using patch

[91]:
```
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 0x000001E075FAFF70>
Inside patch:  <MagicMock name='get_animals' id='2063563728496'>
```

```
Outside again: <function get_animals at 0x000001E075FAFF70>
```

- patch works for modules, classes and attributes
- it can be used in `with` statements, as a functional `decorator` or in the `setUp` and `tearDown` methods of TestCase classes

- `patch` doesent work in all cases
- to test `do_rounds` we need to mock out the current time returned by the `datetime.utcnow` class method
- python wont let me do this because the `datetime` class is defined in a `C`-extension module, which can't be modified in this way

[92]:
```python
fake_now = datetime(2019, 6, 5, 15, 45)

try:
    with patch('datetime.datetime.utcnow'):
        datetime.utcnow.return_value = fake_now
except TypeError as e:
    print(e)
```

```
can't set attributes of built-in/extension type 'datetime.datetime'
```

- to work around this, we can create a helper function to fetch time that can be patched

[93]:
```python
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 we can use a keyword-only argument for the `datetime.utcnow` mock and use `patch` for all of the other mocks

[100]:
```python
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
```

- we are going to go with the latter approach
- now we can use the `patch.multiple` function to create many mocks and set their expectations

[101]:
```
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))
    ]
```

- witht he setup ready, we can run the test and verify that the calls were correct inside the `with` statement that used `patch.multiple`

[102]:
```
try:
    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)
except TypeError as e:
    print(e)
```

```
'NoneType' object is not iterable
```

- when the setup is ready, we can run the test and veify that the calls were correct inside the `with` statement that used `patch.multiple`
- the keyword arguments to `patch.multiple` correspond to the names in the `__main__` module that we want to overreide during the test
- the `DEFAULT` value indicated that I want to standard `Mock` instance to be created for each name
- All of the generated mocks will adhere to the specification of the object they are meant to simultate, thanks to the `autospec=True` parameter

0.4.1 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 its 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 into the code being tested

0.5 Item 79: Encapsulate Dependencies to Facilitate Mocking and Testing

- 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 the functions as an argument
- its woth refactoring your code to use better abstractions because it facilitates creating mocks and writing tests

- below we define the various database helper functions from the previous item as methods on a class instead of as independent functions

[103]:
```
class ZooDatabase:
    ...

    def get_animals(self, species):
        ...

    def get_food_period(self, species):
        ...

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

- now we can redefine the `do_rounds` function to call method son a `ZooDatabase` object

[104]:
```
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 alot easier because we no longer need to use `unittest.mock.patch` to inject the mock into code being tested

- instead we 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 we can then use to set expectations and verify calls
- this makes it easy to mock out all of the methods of a class

[105]:
```python
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='2063563903328'>
```

- we can rewrite the `Mock` setup code by using the `Zoodatabase` encapsulation

[109]:
```python
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 we can returnt he function being tested and verify that all dependent methods were called as expected

[113]:
```python
try:
    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)
except AssertionError as e:
    print(e)
```

```
Expected 'get_food_period' to be called once. Called 4 times.
Calls: [call('Meerkat'), call('Meerkat'), call('Meerkat'), call('Meerkat')].
```

- using the `spec` parameter to `Mock` is especially useful when mocking classes because it ensures that the code under test doesnt 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 reveal itself in production

[114]:
```
try:
    database.bad_method_name()
except AttributeError as e:
    print(e)
```

```
Mock object has no attribute 'bad_method_name'
```

- if we want to test this program `end-to-end` with a mid-level integration `test`, we still need a way to inject a mock 'ZooDatabase into the program
- we can do this by creating a helper function that acts as a seam for `dependency injection`
- here we define a helper function that caches a `ZooDatabase` in module scope using a `global` statement

[118]:
```
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 we can inject the mock `ZooDatabase` using `patch`, run the test and verify the programs output
- were not using a mock `datetime.utcnow` but 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

[119]:
```
import io
import contextlib
from unittest.mock import patch

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

```python
    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
```

- creating this integration test was straightforward because we designed the implmentation to make it easier to test

0.5.1 Things to remember

- when unit test require alot of 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 atttribute then is accessed
- for `end-to-end` tests, its valuable to refactor your code to have more helper function that can act as explicit seams for injecting mock dependencies in tests

0.6 Item 80: Consider Interactive Debugging with pdb

- in 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

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

[5]:
```python
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
```

```python
    rmse = math.sqrt(mean_err)
    return rmse

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

```
0.5291502622129182
```

- 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` function and even modify parts of the running program- whatever you need to do to aid in your debugging

- 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 progra that led to the breakpoint
  - `down`: move your scope back down the execution call stack one level

- when your done inspecting the current state, you can use these five debugger commands to control the programs execution:
  - `step`: Run the program unitll 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 untill 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 untill the called function has returned
  - `return`: Run the program untill the current function returns and then returns control back to the debugger prompt
  - `continue`: Continue running the prompt untill the next `breakpoint` call or one added by a debugger command
  - `quit`: Exit the debugger and end the program. Run tis command if you've found the problem gone too far or need to make the program modifications and try again

- `post-mortem debugging` is a useful way to reach the debugger prompt
- this enables us to debug a program after its already raised an exception and c rashed
- this is helpful when you dont know where to put the break point
  - `python3 -m pdb -c continue postmortem_breakpoint.py`

- 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

```
>>> import my_module
>>> my_module.compute_stddev([5])
Traceback(...)
```

```
...
>>> `import pdb; pdb.pm())`
>>> (pdb) err_2_sum
```

0.6.1 Things to Remember

- 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()`)

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

- memory management in the default implementation of Python, `CPython`, uses refrence counting
- this ensures that as soon as all refrences 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
- you generally dont have to worry about memeory in a python application but in pratice you can run out of memeory due to no longer useful refrences still being held
- the first way to debug memory usage is to ask the `gc` built in module to list every object currently known by the garbace collector
- the tool is blunt but leys you quickly get a sense of where your programs memory is bing used

[13]:
```python
# 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 we run a program that uses the `gc` built-in module to print out how many objects were created during execution along with a sample of allocatged objects

[14]:
```python
# using_gc.py
import gc

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

# 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: 55746
```

- the problem with the `gc.get_objects` is that it does not tell you anything about how the objects were allocated
- `tracemalloc` built-in module helps us identify the code responsible for allocating the objects that were leaking memory

- `tracemalloc` makes it possible to connect an object back to where it was allocated
- you use it by taking befor and after snapshots of memory usage and comparing them to see whats changed
- we use this approach to print out the top three memory usage offenders in our application

[17]:
```python
# top_n.py
import tracemalloc

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



x = 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)
```

```
<ipython-input-13-9260f813853c>:5: size=2314 KiB (+2314 KiB), count=30000
(+30000), average=79 B
<ipython-input-13-9260f813853c>:10: size=469 KiB (+469 KiB), count=10000
(+10000), average=48 B
<ipython-input-13-9260f813853c>:11: size=84.4 KiB (+84.4 KiB), count=100 (+100),
average=864 B
```

- the size and count labels in the output make it immediately clear which objects are dominating my projects memory usage and where in the source code they were allocated

- `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 memeory usage in the program

[18]:
```python
# with_trace.py
import tracemalloc

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

x = 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 "c:\users\vicktree\appdata\local\programs\python\python39\lib\site-
packages\IPython\core\interactiveshell.py", line 2894
    result = self._run_cell(
  File "c:\users\vicktree\appdata\local\programs\python\python39\lib\site-
packages\IPython\core\interactiveshell.py", line 2940
    return runner(coro)
  File "c:\users\vicktree\appdata\local\programs\python\python39\lib\site-
packages\IPython\core\async_helpers.py", line 68
    coro.send(None)
  File "c:\users\vicktree\appdata\local\programs\python\python39\lib\site-
packages\IPython\core\interactiveshell.py", line 3165
    has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
  File "c:\users\vicktree\appdata\local\programs\python\python39\lib\site-
packages\IPython\core\interactiveshell.py", line 3357
    if (await self.run_code(code, result,  async_=asy)):
  File "c:\users\vicktree\appdata\local\programs\python\python39\lib\site-
packages\IPython\core\interactiveshell.py", line 3437
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-18-d63433c329f2>", line 7
    x = run()
  File "<ipython-input-13-9260f813853c>", line 17
    deep_values.append(get_data())
  File "<ipython-input-13-9260f813853c>", line 10
    obj = MyObject()
  File "<ipython-input-13-9260f813853c>", line 5
    self.data = os.urandom(100)
```

0.7.1 Things to Remember

- it can be difficult to understand how Python programs use the leak memory

- the `gc` module can help you understand which object exist, but it has no information about how they were allocated
- the `tracemalloc` built-in module provides powerful tools for understanding the source of memeory usage