# STOP
# USING
# MOCK OBJECTS

Let's write meaningful, maintainable tests

# STOP
# USING
quite so many
# MOCK OBJECTS
as the test tool of first resort
also, try to reduce the use of patch()

Let's write meaningful, maintainable tests

# Lot of nuance here

- I am NOT proposing a blanket ban of mock objects in tests
- They are an important tool to have in the toolkit
- When you need them, you need them
- But *you don't need them often*

# Some code is easy to test

```
def count_vowels(s: str) -> int:
    """Return the number of vowels in s"""

def calc_area(r: float) -> float:
    """Return the area of a circle with radius r"""
```

- These are pure functions: they have no implicit inputs, no side effects, and the same inputs always return the same output
- Not interesting: we're talking about code that is *hard* to test

# What makes code hard to test?

- Implicit inputs
    - Global variables
    - Flask request context (which is just a fancy global variable)
    - Read-only API calls
    - Database content
- Side effects (implicit outputs)
    - Read-write API calls
    - Database writes
    - Caching

# That one weird trick for hard-to-test code

- *Test doubles*: test-specific objects that replace objects in the code being tested
- There are many types of test doubles!
    - Fakes (working implementation, but not the real thing)
    - Stubs (hardcoded responses specific to your tests)
    - Dummies (do-nothing objects)
    - Spies (keeps track of what is done to it)
    - Mocks (my bête noire)

# Sources

- Great blog post by Martin Fowler from 2007: https://martinfowler.com/articles/mocksArentStubs.html
- Book by Gerard Meszaros: XUnit Test Patterns http://xunitpatterns.com/
- Disclaimer: I've read the blog post many times, but not the book

# What's so bad about mock objects?

1. Hard to write, harder to understand
2. Self-fulfilling prophecy
3. Tests tightly coupled to code under test
4. Python's `unittest.mock` is too permissive by default

# 1. Hard to write, hard to understand, even harder to modify

- Writing feature code is easy
- Writing normal tests is harder
- Writing mock-based tests is harder than that
- Understanding mock-based tests is harder than that
- Modifying mock-based tests is the hardest of all

# 2. Self-fulfilling prophecy

- A common mistake: the test simply recapitulates what the code under test does
- Code under test: API GET, DB read, API POST
- Test code: expect API GET, expect DB read, expect API POST
- All this proves is that you can write the same code twice
    - Once the normal way
    - Second time in the "expect this next" way
    - Doesn't really tell you if the code is correct!

# 3. Tests tightly coupled to implementation

- Does it *matter* if the code does things in a certain order?
- Does it *matter* that it does X exactly once, and Y not at all?
- Sometimes, maaaybe, and that's when mocks *might* be OK
- But now any change to the implementation breaks the tests
    - Which means you must adapt the test to the new implementation
    - And we've already established that modifying mock-based tests is hard

# 4. unittest.mock is too permissive

- unittest.mock provides two classes for mock objects
  - Mock is very permissive: any attribute, any method call works
  - MagicMock is insanely permissive: Mock, plus any operator overload

# How permissive is unittest.mock.Mock?

```
>>> from unittest.mock import Mock
>>> m = Mock()
>>> m.asdfasdf
<Mock name='mock.asdfasdf' id='127394004839760'>
>>> m.rg932t = "foobar"
>>> m.garbage(2, "", None)
<Mock name='mock.garbage()' id='127393993336016'>
>>> m.garbage("", [])
<Mock name='mock.garbage()' id='127393993336016'>
>>> m + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Mock' and 'int'
```

# How permissive is unittest.mock.MagicMock?

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock()
>>> m.dvjq235
<MagicMock name='mock.dvjq235' id='127393993419600'>
>>> m.bdrgas = 54
>>> m.bogus({}, (34,1.0))
<MagicMock name='mock.bogus()' id='127394004475856'>
>>> m.bogus("hello")
<MagicMock name='mock.bogus()' id='127394004475856'>
>>> m + 4
<MagicMock name='mock.__add__()' id='127393993488848'>
```

# Case study 1: some code we need to test

```
class Worker:
    def get_foobar(self) -> str:
        """Make a slow, error-prone API call."""

    def process_foobar(self, a: str) -> str:
        """Do a lot of expensive processing"""

def do_work(worker: Worker) -> str:
    a = worker.get_foobar()
    b = worker.process_fooba(a)    # should FAIL!
    return a + b
```



What does "correct" mean?

- It doesn't crash with an AttributeError (TypeError, etc.)
- It returns a new string built from the outputs of the Worker object

# Bad test: overly permissive mock object

```
mock_worker = mock.Mock()
result = do_work(mock_worker)  # tested code is incorrect, but does not fail
```

# Bad test: self-fulfilling prophecy

```
mock_worker = mock.Mock(spec=Worker)
mock_worker.get_foobar.return_value = "abc"
mock_worker.process_foobar.return_value = "def"

result = do_work(mock_worker)
assert result == "abcdef"

mock_worker.get_foobar.assert_called_once_with()
mock_worker.process_foobar.assert_called_once_with("abc")
```

hooray: I can write the same code twice: good for me

(also: tightly coupled to implementation)

# Preliminary conclusions

- Maybe mock objects aren't such a great idea after all
- Maybe we should not use mock objects in *every single test* that involves implicit inputs and/or side effects
- Maybe we could find a better way to write some of those tests

# Maybe we could do ... this?

```
class StubWorker:
    def get_foobar(self) -> str:
        return "abc"

    def process_foobar(self, a: str) -> str:
        return "def"

stub_worker = StubWorker()
result = do_something(stub_worker)
assert result == "abcdef"
```

- For a single test, this isn't a huge difference
- But when you have multiple tests that use the same stub, it can be a big win
- It's even more fun with a fake

# Case study 2: more complex worker

```
class Worker:
    def get_foobar(self, id: int) -> str:
        """Lookup the requested foobar."""

    def process_foobar(self, a: str) -> str:
        """Do a lot of expensive processing."""

def do_work(worker: Worker, id: int, n: int) ->
str:
    a = worker.get_foobar(id)
    b = worker.process_foobar(a)
    return (a * n) + b
```

What does "correct" mean?

- It doesn't crash with an AttributeError (TypeError, etc.)
- It returns a new string built from the outputs of the Worker object

# Testing with a fake

```
class FakeWorker:
    def __init__(self):
        self.foobars = {1: "abc", 2: "bep", 5: "qux"}

    def get_foobar(self, id: int) -> str:
        return self.foobars[id]

    def process_foobar(self, a: str) -> str:
        return "def"

fake_worker = FakeWorker()
result = do_something(fake_worker, 5, 2)
assert result == "quxquxdef"
```

# Intermission: mock ≠ patch

- unittest.mock provides Mock, MagicMock, and patch()
- By default, patch() creates a MagicMock
- It's easy to get confused and think they must be used together
- But it's not so! You can patch() in a stub, or you can pass a mock directly
- Sometimes, code cannot be tested without patch() 😮

CODE SMELL

# Case study 3:
# Testing side effects (implicit outputs)

```
_source: ThingSource = None
_cache: dict[str, Thing] = {}

def get_thing(id: int) -> Thing:
    global _cache, _source
    if id not in _cache:
        if _source is None:
            _source = create_thing_source()
        _cache[id] = _source.get_thing(id)
    return _cache[id]
```

What is correct?

- Return the requested Thing (or raise exception)
- First call with new `id` updates the cache
- Repeated calls with the same `id` only call ThingSource.get_thing() once

# Testing "only call method once"

- Could implement a custom spy object
- Or could just use a Mock ... it's there! this is its strength!
- Code smell: have to patch() the ThingSource ;-(

# First attempt

```
import things

def test_get_thing():
    source = mock.Mock(spec=ThingSource)
    with mock.patch("things._source", new=source):
        thingA = things.get_thing(1)
        assert things._cache == {1: thingA}
        thingB = things.get_thing(1)
        assert things._cache == {1: thingA}
        assert thingA is thingB

source.get_thing.assert_called_once_with(1)
```

# WRONG! Mocks make this test harder

- What was the first item on the "correctness" list?
    - Return the requested Thing (or raise exception)
- This  Mock does neither!
- Repeated calls to `source.get_thing()` return the same object, because *that is how mocks work*
- Overriding this means implementing our own source.get_thing() ... in which case, why are we bothering with a Mock?

# Second attempt

```
import things

class SpyThingSource:
    def get_thing(self, id: int) -> Thing:
        self.calls.append(id)
        return things.Thing(id=id)

def test_get_thing():
    source = SpyThingSource()
    with mock.patch("things._source", new=source):
        thingA = things.get_thing(1)
        assert thingA.id == 1
        assert things._cache == {1: thingA}
        thingB = things.get_thing(1)
        assert thingA is thingB
        assert source.calls == [1]

        thingC = things.get_thing(3)
        assert thingC.id == 3
        assert thingC is not thingA
```

# Refactor for testability (can we test without patch()?)

```
_cache: dict[str, Thing] = {}

def get_thing(source: ThingSource, id: int) ->
Thing:
    global _cache
    if id not in _cache:
        _cache[id] = source.get_thing(id)
    return _cache[id]
```

# Third attempt

```
import things

class SpyThingSource:
    [...unchanged...]

def test_get_thing():
    source = SpyThingSource()
    thingA = things.get_thing(source, 1)
    assert thingA.id == 1
    assert things._cache == {1: thingA}
    thingB = things.get_thing(source, 1)
    assert thingA is thingB
    assert source.calls == [1]

    thingC = things.get_thing(source, 3)
    assert thingC.id == 3
    assert thingC is not thingA
```

# Case study 4: Multiple dependencies

```
def notify_user(
    user_id: int,
    event: UserEvent,
):
    api_config = api.ConfigAPIClient()
    api_users = api.UserAPIClient()
    api_employees = api.EmployeeAPIClient()

    [...business logic that needs to be tested...]
```

# Design flaws (pre-refactoring)

- Cannot be tested without patch()ing several factory functions
- Coupled to concrete implementation types:
  - the code knows what api_config is and how to construct it
  - same for api_users, api_employees
  - it's a pretty weak coupling, but still

# Not quite the right refactoring

No `patch()`ing required! But, yuck, too many parameters

```
def notify_user(
    api_config: ConfigAPIClient,
    api_users: UserAPIClient,
    api_employees: EmployeeAPIClient,
    user_id: int,
    event: UserEvent,
):
    [...business logic that needs to be tested...]
```

# Worker objects can be awesome

```
class NotificationWorker:
    def __init__(self, api_config, api_users, api_employees):
        self.api_config = api_config
        self.api_users = api_users
        self.api_employees = api_employees

    def notify_user(self, user_id, event):
        [...business logic that needs to be tested...]
```

# Good thing about both refactored versions

- The code that is responsible for constructing a notification and sending it has *no idea* what api_config, api_patients, etc. are
- They're just *objects* that implement a particular *interface*
- In Java or Go, you are forced to define that interface formally
- In Python, nobody forces you, but you really should
- ONE refactoring fixes BOTH design flaws

# Wrap-up:
# Some guidelines on using mock objects

1. DON'T. Just don't do it if you can avoid it
2. Always use spec, or spec_set, or autospec
3. Avoid MagicMock unless you need to test operator overloading
4. NEVER use unittest.mock.patch() with the defaults
    - creates a MagicMock
    - with no spec
    - it's totally unconstrained! it will do anything!