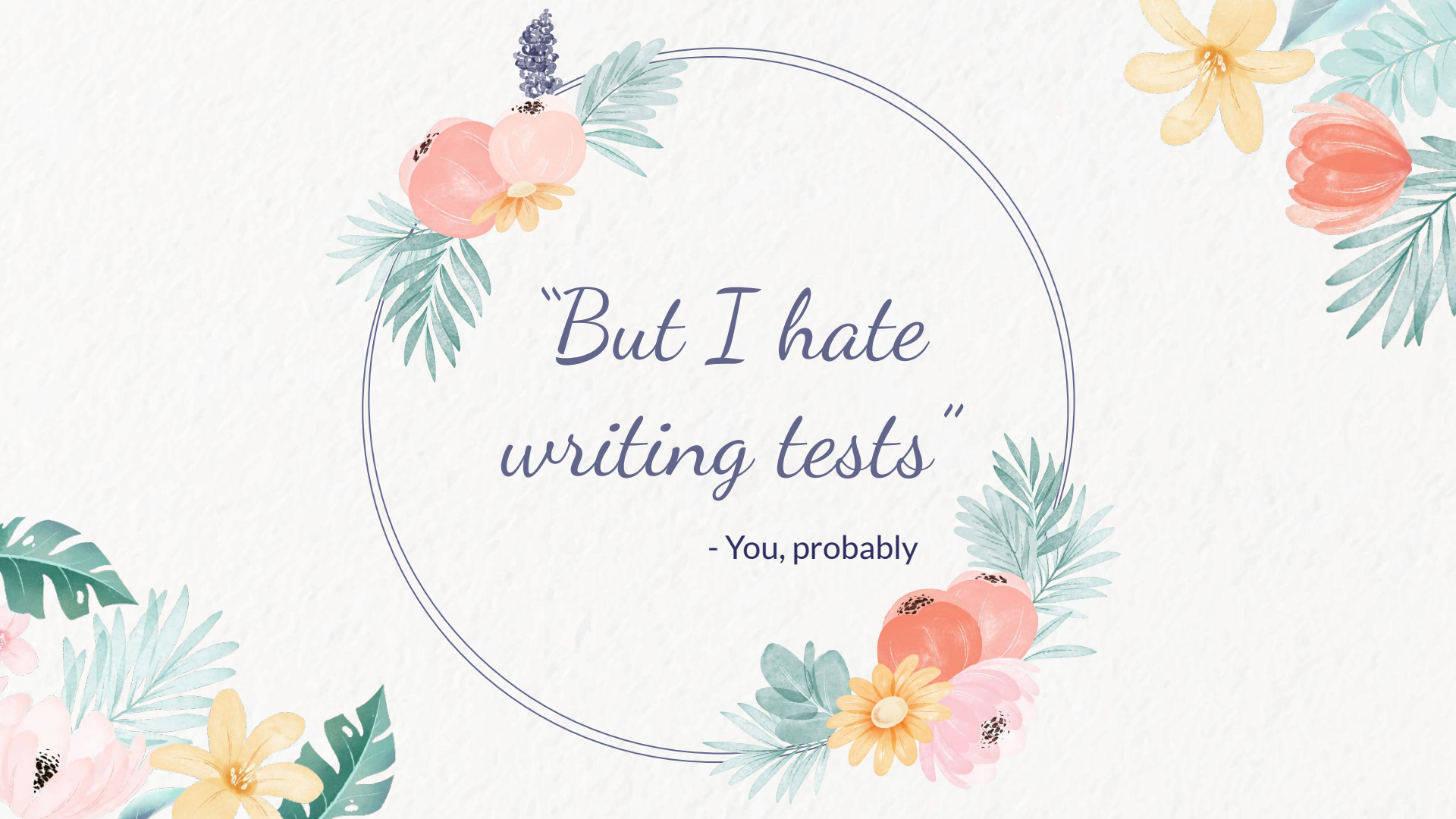
Writing Effective Unit Tests

Jessie Newman
ConfFoo Montreal 2024

Jessie Newman

jessiedotjs

linkedin.com/in/newmanjessie


"But I hate writing tests"
- You, probably

How does good testing benefit YOU?

Good testing increases the impact of your code

- Tests serve as documentation of the code
- People are more likely to integrate with code that they aren't afraid of breaking

How does good testing benefit YOU?

Good testing increases the lifetime of your code

- Tests prevent the logic from becoming stale
- People feel safer making updates to tested code

How does good testing benefit YOU?

Good testing lets your take on more projects

- Easier to ramp up new developers on tested code
- Less breakage means less time maintaining it

Types of Tests

	Unit Tests	*Integration Tests*	*End to End Tests*
Scope	Individual “unit”	Connection between two components	Entire user story
Ease to write, debug, & maintain	Easiest		Hardest
Number of tests	Most		Least


What makes a
good unit test?

Unit Tests Should...

01 Be Readable

02 Only Test One Thing

03 Be Deterministic

04 Cover All Use Cases & Edge Cases

05 Contain as Little Logic as Possible

06 Not Test Implementation Details

Unit Tests Should Be Readable

Arrange - Act - Assert

```
def testCheckoutBook():
    library = Library()
    book = library.addBook(title="Way of Kings", author="Sanderson", copies=4)
    library.addCustomer(username="valid-username")
    assert library.getNumCopiesAvailable(book=book) == 4

    library.checkout(book=book, username="valid-username")

    assert library.getNumCopiesAvailable(book=book) == 3
    checked_out_books = library.getCheckedOutBooks(username="valid-username")
    assert len(checked_out_books) == 1
    assert checked_out_books[0].title == "Way of Kings"
    assert checked_out_books[0].author == "Sanderson"
```

Arrange

Act

Assert

Unit Tests Should Be Readable

One Condition Per Test

```
def testCheckoutBook()
def testCheckoutBook__noCopiesAvailable()
def testCheckoutBook__invalidCustomer()
def testCheckoutBook__customerAtCheckoutLimit()
def testCheckoutBook__customerAccountIsSuspended()
```

Unit Tests Should Be Readable

Good Function & Variable Naming

```
def testCheckoutBook__invalidCustomer():
    library = Library()
    book = library.addBook(title="Way of Kings", author="Sanderson", copies=4)
    assert library.getNumCopiesAvailable(book=book) == 4

    with pytest.raises(Exception, match="No customer found with username invalid-username"):
        library.checkout(book=book, username="invalid-username")

    assert library.getNumCopiesAvailable(book=book) == 4
```

Unit Tests Should Be Readable

Consider when to hide “Arrange” code

```
@fixture
def library():
    library = Library()
    book = library.addBook(title="Way of Kings", author="Sanderson", copies=4)
    book = library.addBook(title="Words of Radiance", author="Sanderson", copies=0)
    book = library.addBook(title="Edgedancer", author="Sanderson", copies=12)
    book = library.addBook(title="Oathbringer", author="Sanderson", copies=3)
    library.addCustomer(username="customer1")
    library.addCustomer(username="customer2")
    library.addCustomer(username="customer3")
    return library
```

Unit Tests Should Be Readable

Consider when to hide “Arrange” code

```
@fixture
def library():
    library = Library()
    book = library.addBook(title="Way of Kings", author="Sanderson", copies=4)
    book = library.addBook(title="Words of Radiance", author="Sanderson", copies=0)
    book = library.addBook(title="Edgedancer", author="Sanderson", copies=12)
    book = library.addBook(title="Oathbringer", author="Sanderson", copies=3)
    library.addCustomer(username="cus
    library.addCustomer(username="cus
    library.addCustomer(username="cus
    return library
```

```
def testCheckoutBook2(library):
    book = library.getBook(title="Way of Kings", author="Sanderson")
    assert library.getNumCopiesAvailable(book=book) == 4

    library.checkout(book=book, username="customer1")

    assert library.getNumCopiesAvailable(book=book) == 3
    checked_out_books =
library.getCheckedOutBooks(username="customer1")
    assert len(checked_out_books) == 1
    assert checked_out_books[0] == book
```

Unit Tests Should...

01 Be Readable

02 Only Test One Thing

03 Be Deterministic

04 Cover All Use Cases & Edge Cases

05 Contain as Little Logic as Possible

06 Not Test Implementation Details

Unit Tests Should Only Test One Thing

Testing too many functions makes a failure hard to debug.

```
def testSignUpForLibraryCard():
    library = Library()
    book = library.addBook(title="Way of Kings", author="Sanderson", copies=4)
    customer = library.addCustomer(username="new-customer")
    customer.sendVerificationEmail()
    customer.verify()
    library.setBookLimitForCustomer(username=customer, limit=8)
    assert library.getNumCopiesAvailable(book=book) == 4
    library.checkout(book=book, username="new-customer")
    assert library.getNumCopiesAvailable(book=book) == 3
    checked_out_books = library.getCheckedOutBooks(username="new-customer")
    assert len(checked_out_books) == 1
    assert checked_out_books[0].title == "Way of Kings"
    assert checked_out_books[0].author == "Sanderson"
```

Unit Tests Should Only Test One Thing

Calling an API is outside the scope of a unit test.

```
def addBookReview(self, username: str, book: Book, stars: int, review: str):
    if (
        book.id not in self.customers_to_checked_out_books[username] and
        book.id not in self.customers_to_returned_books[username]
    ):
        raise Exception("You cannot review a book you have not read. ")
    db = mysql.connector.connect(
        host="localhost",
        user="fakeusername",
        password="fakepassword",
        database="fakedatabase",
    )
    mycursor = db.cursor()
    mycursor.execute(
        "INSERT INTO reviews (book_id, username, rating, text) ",
        (book.id, username, stars, review)
    )
    db.commit()
```

Unit Tests Should Only Test One Thing

Calling an API is outside the scope of a unit test.

```
def testAddReviewBook():
    library = Library()
    library.addCustomer(username="username")
    book = library.addBook(title="Way of Kings", author="Sanderson", copies=4)
    library.checkout(book=book, username="username")

    library.addBookReview(
        username="username",
        book=book,
        stars=5,
        review="What a fun book!"
    )

    book_review = library.getBookReview(book)[-1]
    assert book_review.username == "username"
    assert book_review.stars == 5
    assert book_review.review == "What a fun book!"
```

Unit Tests Should Only Test One Thing

Calling an API is outside the scope of a unit test.

```
@patch("library.mysql.connector.connect")
def testAddReviewBook(fake_connect: MagicMock):
    def fake_execute(sql, values):
        assert sql == "INSERT INTO reviews (book_id, username, rating, text)"
        assert values == (0, 'username', 5, 'What a fun book!')

    fake_connect.return_value.cursor.return_value.execute.side_effect =  fake_execute

    library = Library()
    library.addCustomer(username="username")
    book = library.addBook(title="Way of Kings", author="Sanderson", copies=4)
    library.checkout(book=book, username="username")

    library.addBookReview(
        username="username",
        book=book,
        stars=5,
        review="What a fun book!"
    )
```

Unit Tests Should...

01 Be Readable

02 Only Test One Thing

03 Be Deterministic

04 Cover All Use Cases & Edge Cases

05 Contain as Little Logic as Possible

06 Not Test Implementation Details

Unit Tests Should Be Deterministic

```
def addCustomer(self, username: str) -> Customer:
    self.customers.add(username)
    library_card_number = self.generateLibraryCardNumber()
    self.customer_to_library_card[username] = library_card_number
    return Customer(username, library_card_number)

def generateLibraryCardNumber(self):
    library_card_number = random.randint(1111111111, 9999999999)
    while library_card_number in self.library_cards:
        # This number is already used. Try again.
        library_card_number = random.randint(1111111111, 9999999999)
    return library_card_number
```

Unit Tests Should Be Deterministic

```
@patch("library.random.randint")
def testLookupCustomerByLibraryCardNumber(fake_randint: MagicMock):
    fake_randint.side_effect = [1111111111, 2222222222, 3333333333]
    library = Library()
    customer1 = library.addCustomer("customer1")
    customer2 = library.addCustomer("customer2")
    customer3 = library.addCustomer("customer3")

    assert library.lookupCustomerByLibraryCardNumber(2222222222).username == "customer2"
```

Unit Tests Should...

01 Be Readable

02 Only Test One Thing

03 Be Deterministic

04 Cover All Use Cases & Edge Cases

05 Contain as Little Logic as Possible

06 Not Test Implementation Details

Unit Tests Should Cover All Edge Cases

```
# Valid moves
def test_play_tic_tac_toe_move__causes_horizontal_win()
def test_play_tic_tac_toe_move__causes_vertical_win()
def test_play_tic_tac_toe_move__causes_diagonal_win()
def test_play_tic_tac_toe_move__does_not_cause_win()

# Invalid input
def test_play_tic_tac_toe_move__invalid_player()
def test_play_tic_tac_toe_move__x_y_out_of_bounds()
def test_play_tic_tac_toe_move__x_y_already_played()
def test_play_tic_tac_toe_move__game_already_over()
```

Unit Tests Should Cover All Edge Cases

```
@pytest.mark.parametrize("input,expected", [
    ("4+5", 9), # positive numbers
    ("-3+7", 4), # left negative number
    ("7+-3", 4), # right negative number
    ("-3+-6", -9), # both negative numbers,
    (" 7 + 3 ", 10), # contains spaces
])
def testCalculator_addition(input, expected):
    calculator = Calculator()
    actual = calculator.solve(input)
    assert actual == expected
```

Unit Tests Should...

01 Be Readable

02 Only Test One Thing

03 Be Deterministic

04 Cover All Use Cases & Edge Cases

05 Contain as Little Logic as Possible

06 Not Test Implementation Details

Unit Tests Should Not Contain Logic

```
def testInSameDirectory():
    path1 = os.path.join("directory", "/folder1")
    path2 = os.path.join("directory", "/folder2")
    assert inSameDirectory(path1, path2)
```

Unit Tests Should Not Contain Logic

```
def testInSameDirectory():
    path1 = os.path.join("directory", "/folder1")
    path2 = os.path.join("directory", "/folder2")
    assert inSameDirectory(path1, path2)
```



```
def testInSameDirectory():
        assert inSameDirectory("/folder1", "folder2")
```

Unit Tests Should Not Contain Logic

```
def test_in_same_directory():
    directory1 = "directory2"
    directory2 = "directory2"
    assert inSameDirectory(directory1, directory1)
    assert not inSameDirectory(directory1, directory2)

    folder1 = "/folder1"
    directory1_path = os.path.join(directory1, folder1)
    directory2_path = os.path.join(directory2, folder1)
    assert inSameDirectory(directory1, directory1_path)
    assert not inSameDirectory(directory2, directory1_path)
    assert not inSameDirectory(directory1_path, directory2_path)

    file1 = "/file.txt"
    directory1_path_to_file = os.path.join(directory1_path, file1)
    directory2_path_to_file = os.path.join(directory2_path, file1)
    assert inSameDirectory(directory1, directory1_path_to_file)
    assert inSameDirectory(directory1_path, directory1_path_to_file)
    assert not inSameDirectory(directory1_path_to_file, directory2_path_to_file)
```

Unit Tests Should...

01 Be Readable

02 Only Test One Thing

03 Be Deterministic

04 Cover All Use Cases & Edge Cases

05 Contain as Little Logic as Possible

06 Not Test Implementation Details

Unit Tests Should Not Test Implementation Details

```
def isEven(num: int) -> bool:
    return num%2 == 0

def removeOddNumbers(nums: List[int]) -> List[int]:
    return [num for num in nums if isEven(num)]
```

```
def testIsEven():
    assert isEven(4)
    assert not isEven(5)

def testRemoveOddNumbers():
    assert removeOddNumbers([1,1,2,3,4,5,8]) == [2,4,8]
```

Unit Tests Should Not Test Implementation Details

```python
def isOdd(num: int) -> bool:
    return num%2 == 1

def removeOddNumbers(nums: List[int]) -> List[int]:
    return [num for num in nums if isOdd(num)]
```

```python
def testIsEven():
    assert isEven(4)
    assert not isEven(5)

def testRemoveOddNumbers():
    assert removeOddNumbers([1,1,2,3,4,5,8]) == [2,4,8]
```

Thank You