Software Project Management – IS212

Automated Testing

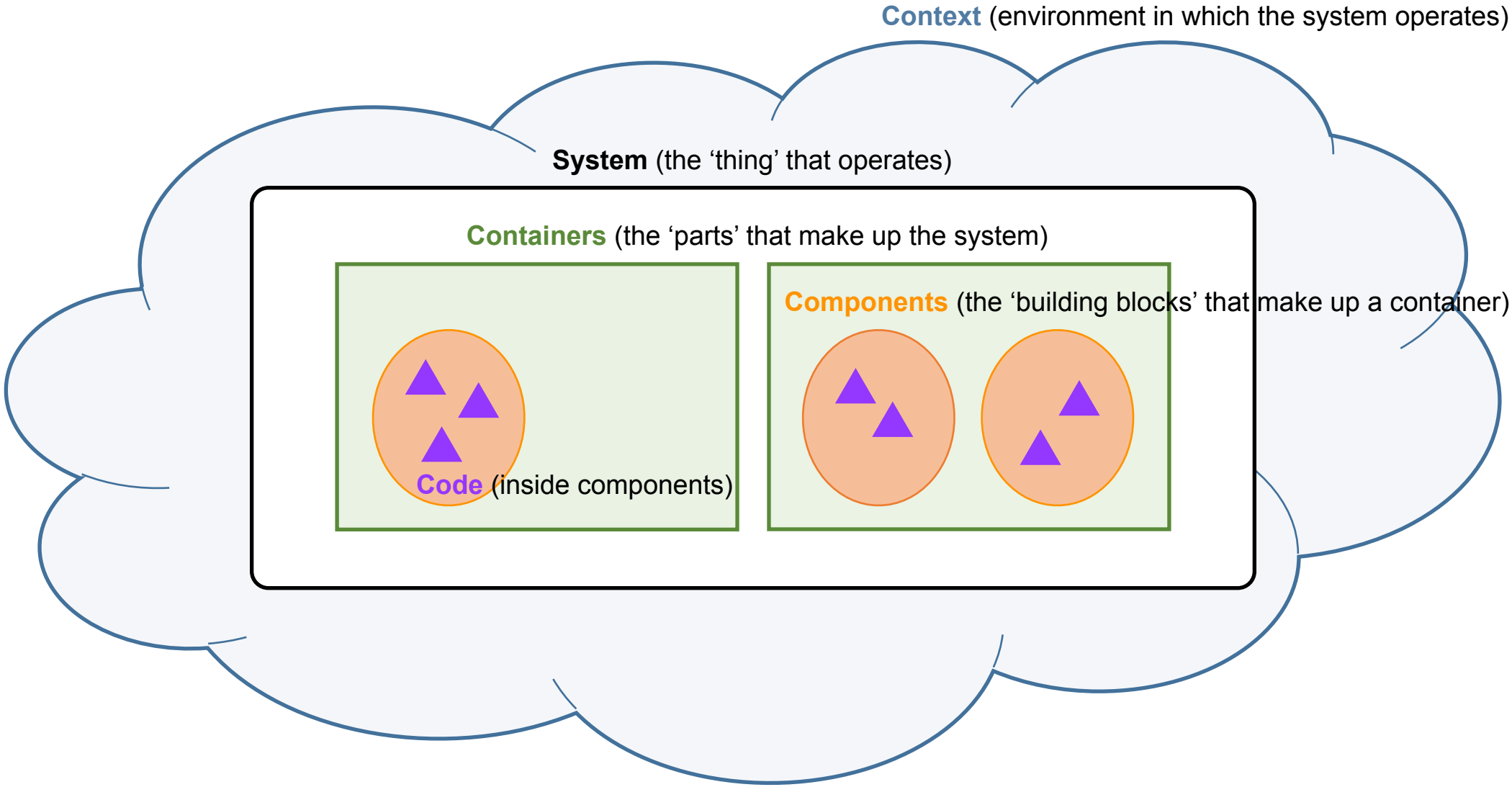
Announcements

Announcements – Week 7 Quiz

- **Week 5 Quiz results are out –** *please ask us if any doubts*
  - *Of the 4 quizzes, only your top 3 quizzes retained*
- Our third quiz will take place at the start of your Week 7 class
  - *Starts promptly at the beginning of class!*
  - *No extra time if you arrive to class late*
- Need to be physically present (no make-ups; see Week 1 slides)
- Requires Respondus LockDown Browser
- Short (~10 mins); covers the Weeks 5–6 official materials
  - *MCQ-style questions (no written)*

Recap

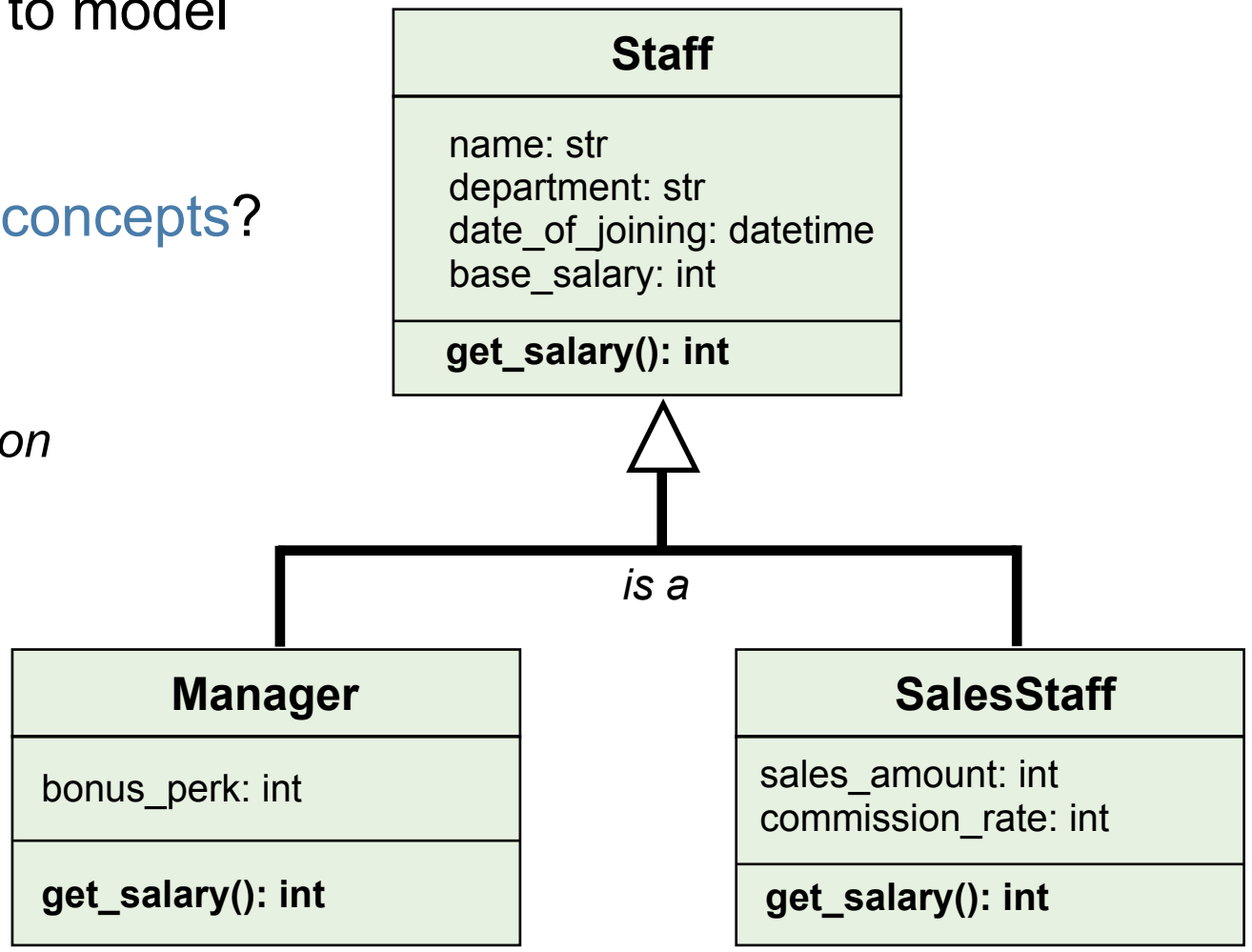
Week 5 Recap – C4 Model



Week 5 Recap – Class Diagrams

- What do UML class diagrams allow us to model about code?

- What are the following object-oriented concepts?
  - *Data abstraction*
  - *Encapsulation*
  - *Association, Aggregation, and Composition*
  - *Inheritance*
  - *Polymorphism*



Today: Automated Testing

Objectives

On completing this class, you will be able to:

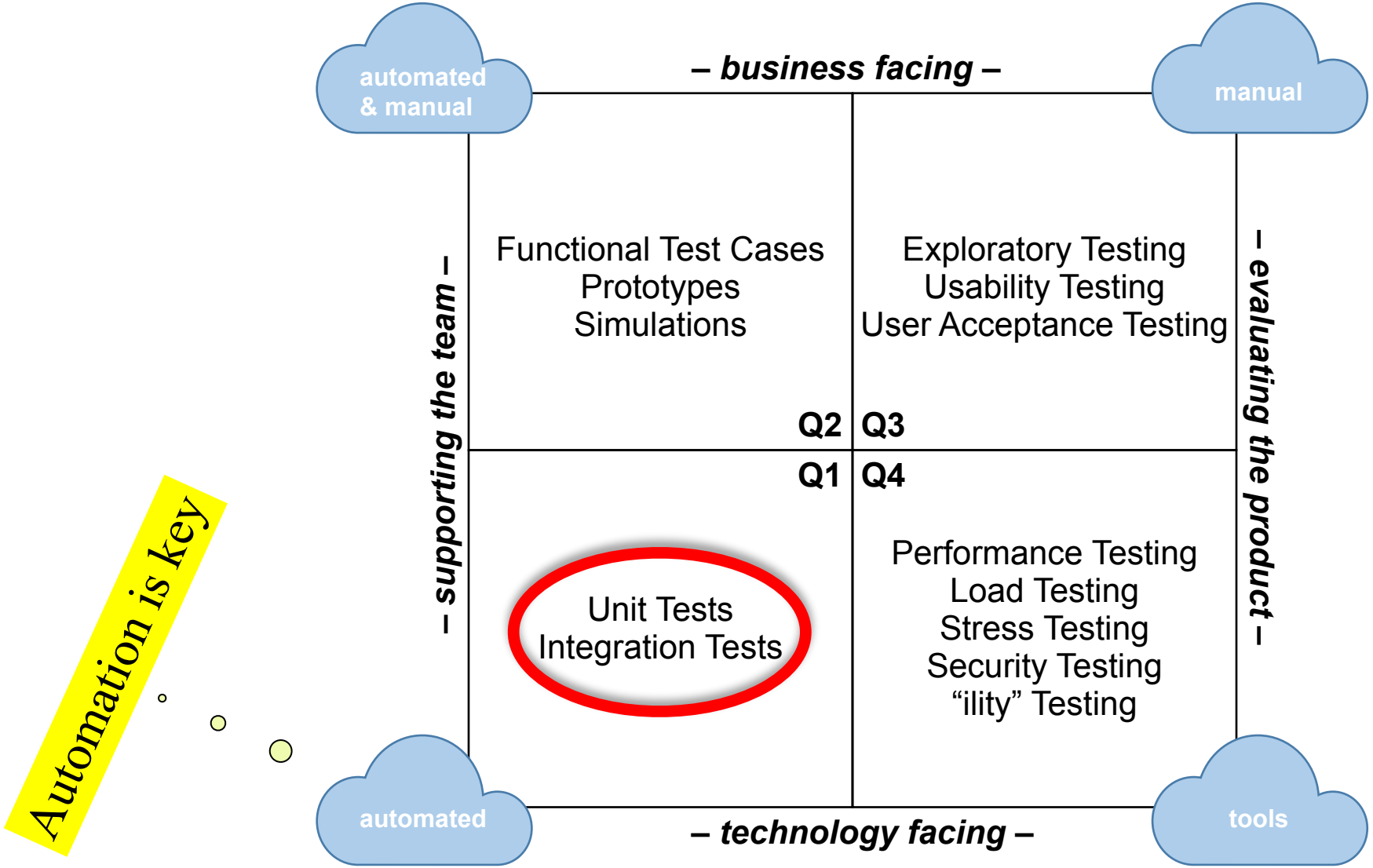
- Implement unit and integration tests that satisfy the FIRST principles
- Conduct Test-Driven Development (TDD) and explain its benefits
- Use a testing framework in Python to produce well-tested and clean code

Outline

- ***Test Automation* –** unit tests, integration tests, FIRST principles, coverage
- ***TDD* –** Red-Green-Refactor cycle, TDD in scrum, worked examples
- ***unittest Framework* –** assertions, fixtures, mocks

Test Automation

Recap – Agile Testing Quadrant



Why Automate Tests?

Repeatable and One-Click

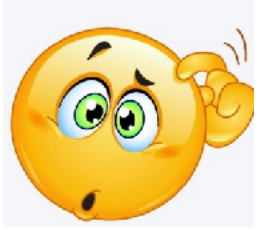
Early Feedback

Safety for Refactoring

Tool Support

Automated Testing – Example

```
def is_palindrome(s):
    cleaned_string = ''.join(char.lower() for char in s if char.isalnum())
    return cleaned_string == cleaned_string[::-1]
```

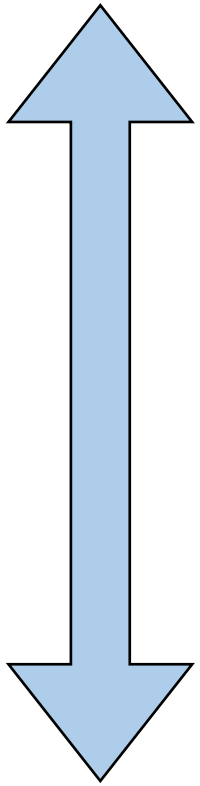


What might automated tests look like for this function … ?

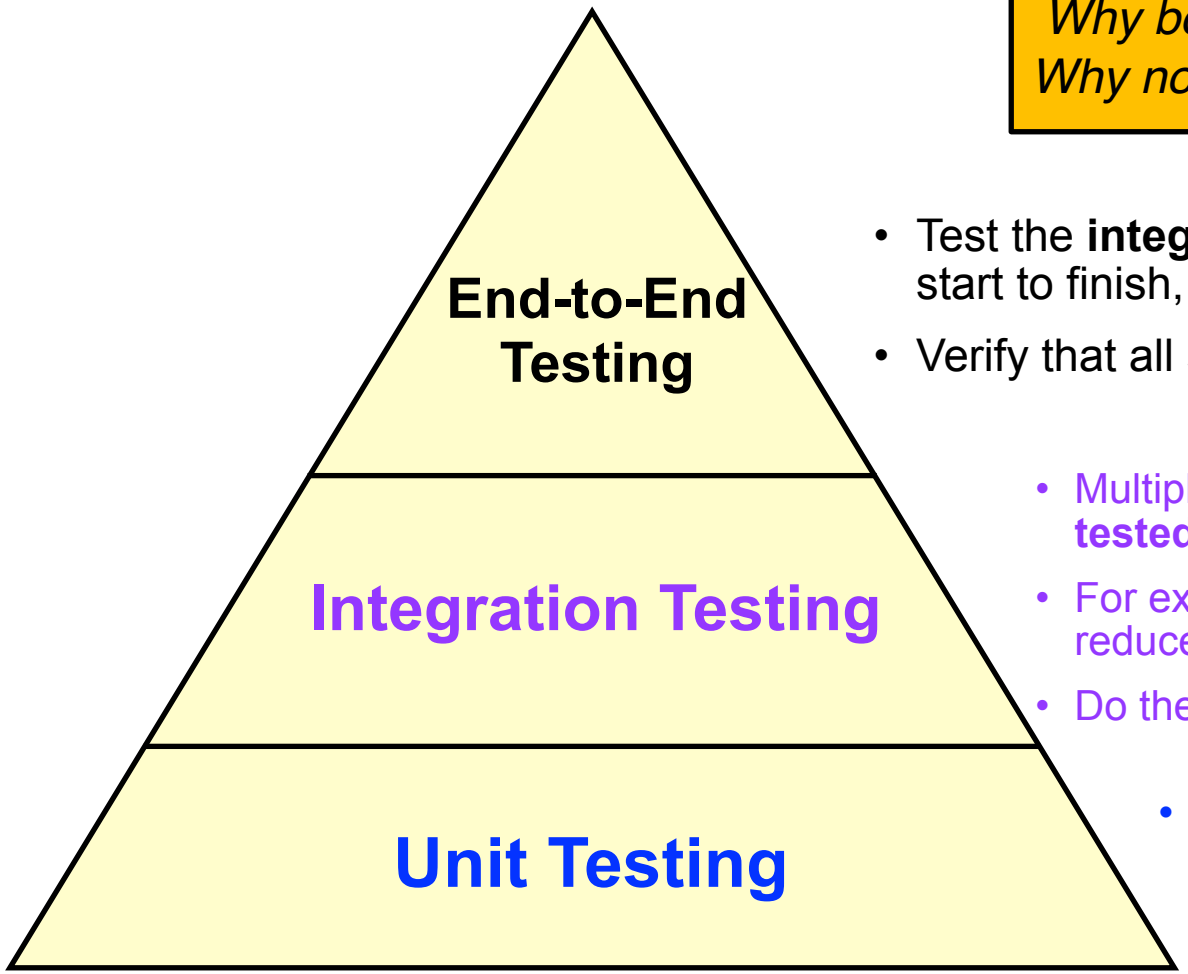
```
assert is_palindrome("radar")
assert is_palindrome("")
assert not is_palindrome("Phris")
assert not is_palindrome("123")
assert is_palindrome("A Santa lived as a devil at NASA")
```

Automated Testing Pyramid

- more "realistic" tests
- slower, more brittle



- faster, more reliable
- smaller scope



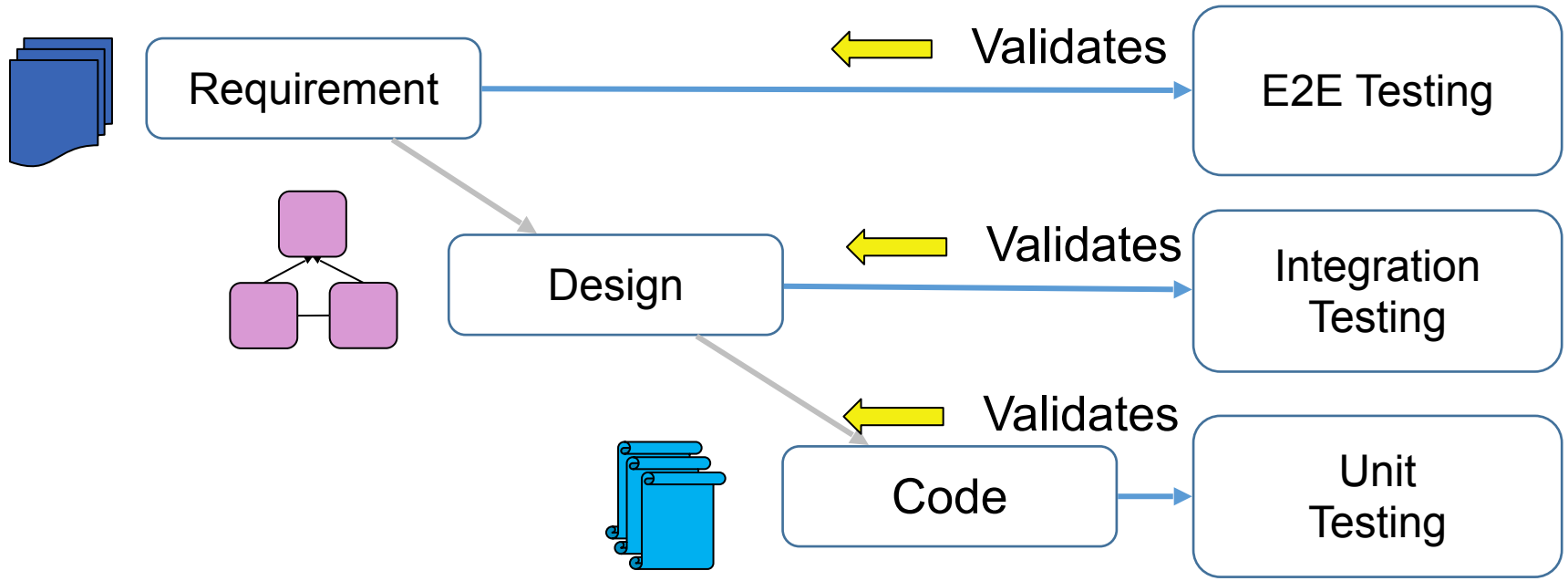
Why bother with unit tests?
Why not just do end-to-end?

- Test the **integrated flow** of the application from start to finish, simulating real user scenarios
- Verify that all subsystems work together correctly

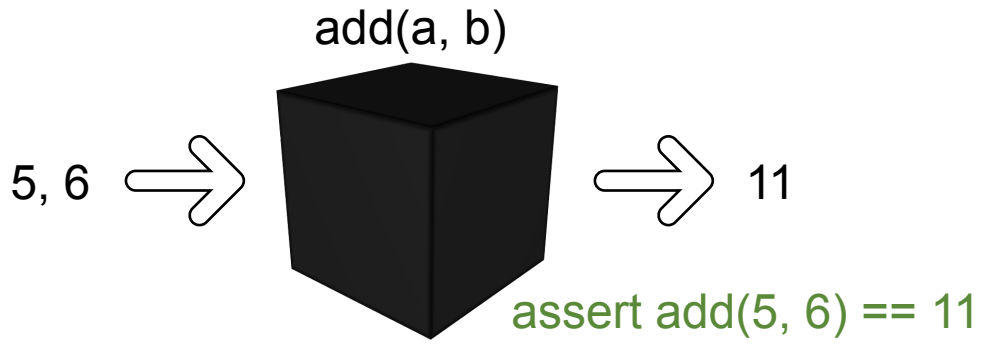
- Multiple components (modules) of a system are **tested together**
- For example, "does placing an **order** correctly reduce the remaining **inventory**?"
- Do the components correctly integrate with the DB?

- Individual "units" (functions/classes) are tested **in isolation** (i.e. without executing other functions, databases, …)

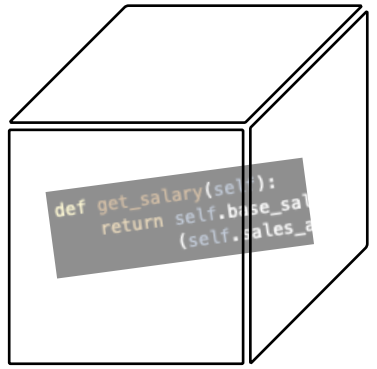
What Tests Validate



Designing Tests – **Black Box** vs. **White Box**

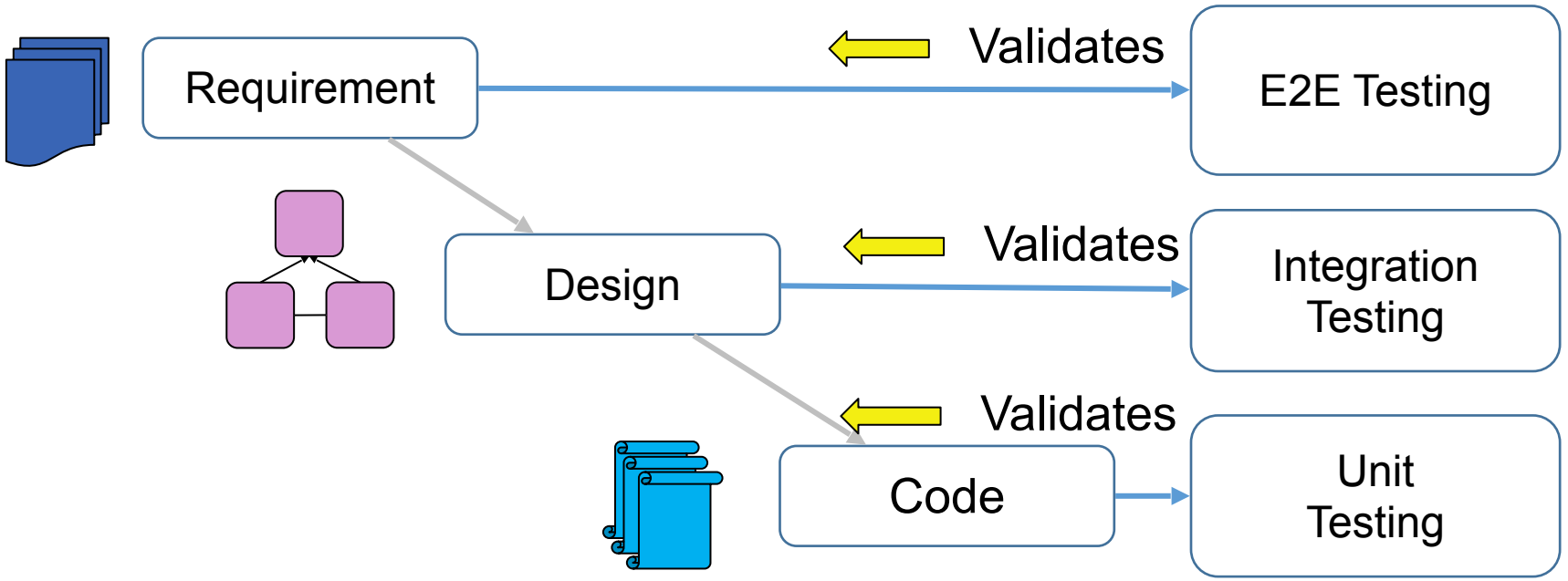


- Design tests based on inputs / outputs without considering the implementation
- Focus on observed behaviour rather than the internal workings of the code



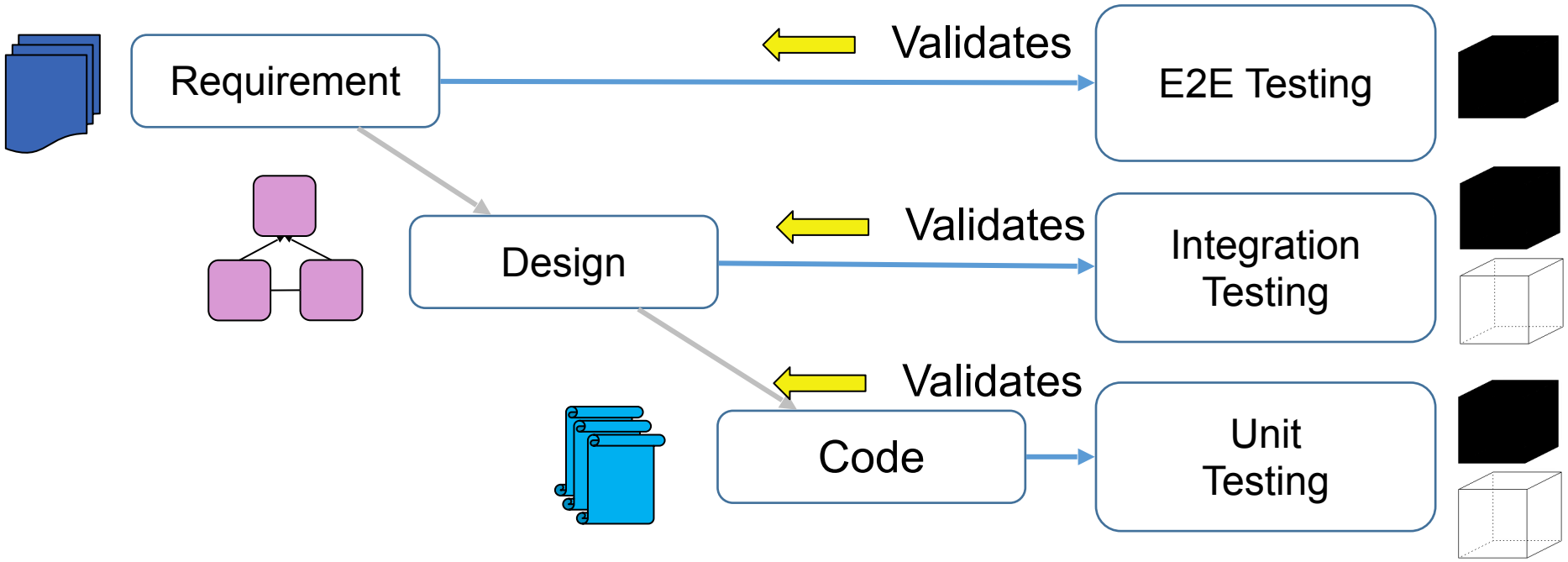
- Take the implementation of functions (i.e. source code) into consideration when designing tests
- Ensure that all internal paths, branches, and conditional statements are working correctly

What Tests Validate



Which levels are black/white box testing suitable for?

What Tests Validate



Which levels are black/white box testing suitable for?

FIRST Principles of Testing

- The "FIRST" mnemonic summaries the expected qualities of automated tests

F*ast*

I*solated*

R*epeatable*

S*elf-Validating*

T*imely*

FIRST Principles of Testing

- **Fast –** production systems have 1000s of tests; need to be fast
- **Isolated –** should be possible to execute tests in any order (a test should not fail because a certain other test was run first)
- **Repeatable –** executing the same test multiple times shouldn't change the outcome, otherwise it is a **flaky** test
- **Self-Validating –** test cases themselves should determine their outcome by using assertions (no manual inspection required)
- **Timely –** test cases can be written any time (even before implementing the function)

Frameworks for Automated Tests

- Rather than writing ad hoc assert statement, we typically make use of testing frameworks that provide useful features:
  - *Test runners, automated test discovery, test fixtures, advanced assertions, …*

- For unit and integration tests in Python, we will use "unittest", but another popular choice is "pytest"
  - *For other languages, see: JUnit, NUnit, MSTest, CppUnit, PhpUnit, etc.*

Test-Driven Development

Test-Driven Development

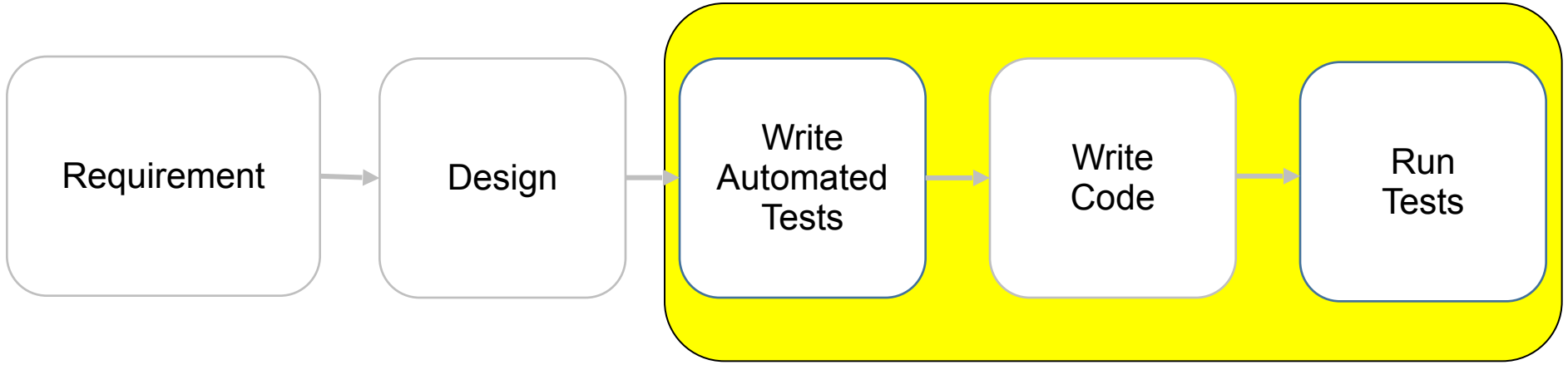
Test-Driven Development is an approach in which we **build a test first**, then **fail the test** and finally refactor our code to **pass the test**

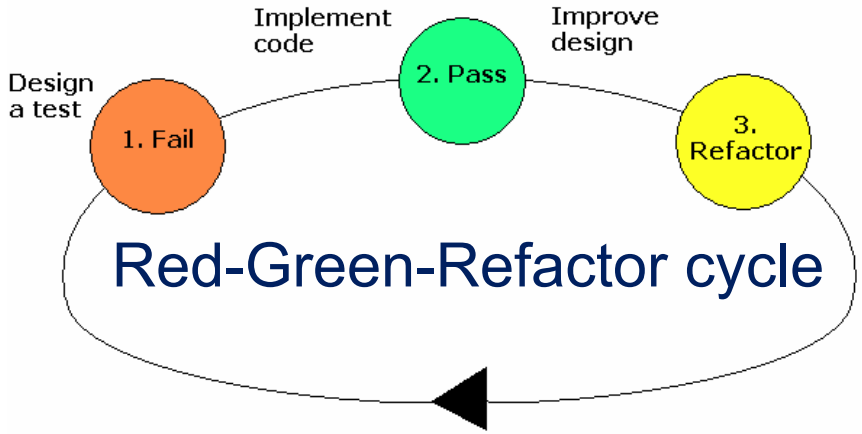
Test first

Test automation

*The goal of TDD is '**clean code that works**'*

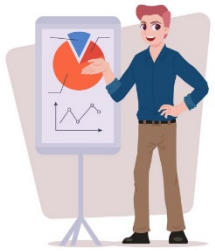
Test-Driven Development - Process



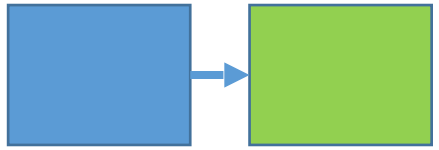


"Refactor"
aka "improve"

Test-Driven Development - Challenges



Management Support

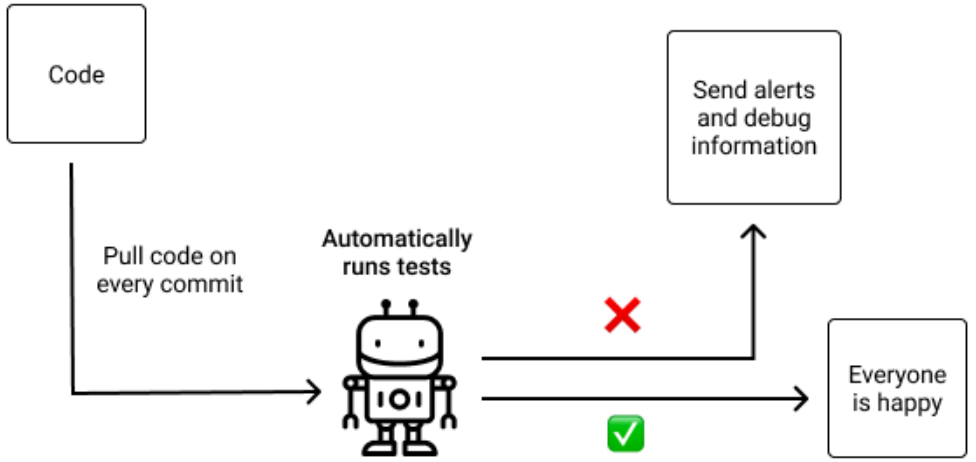


Code with dependencies are difficult to unit test

Test Design and Management

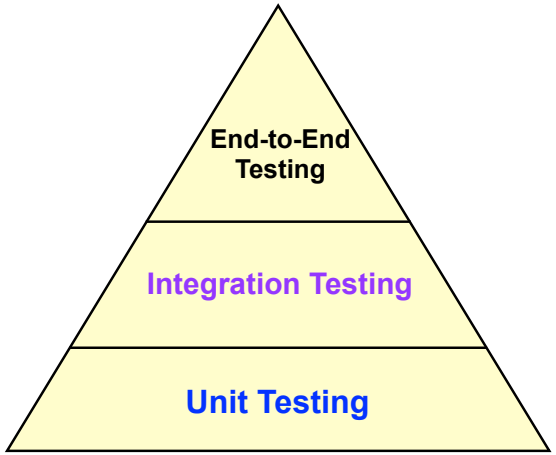
Automated Testing in the IS212 Project

- TDD can be used when implementing stories in your remaining sprint(s)
- Writing tests ***first*** isn't your style? You can still write automated unit and/or integration tests after the initial implementation – *they may help you uncover hidden defects, especially at boundaries!*
- Next week, we'll show you how to automate the execution of your unit and/or integration tests in a CI pipeline, allowing developers to check that new code doesn't break the functionality of existing code



Writing Unit Tests by Example

…using TDD and "unittest"



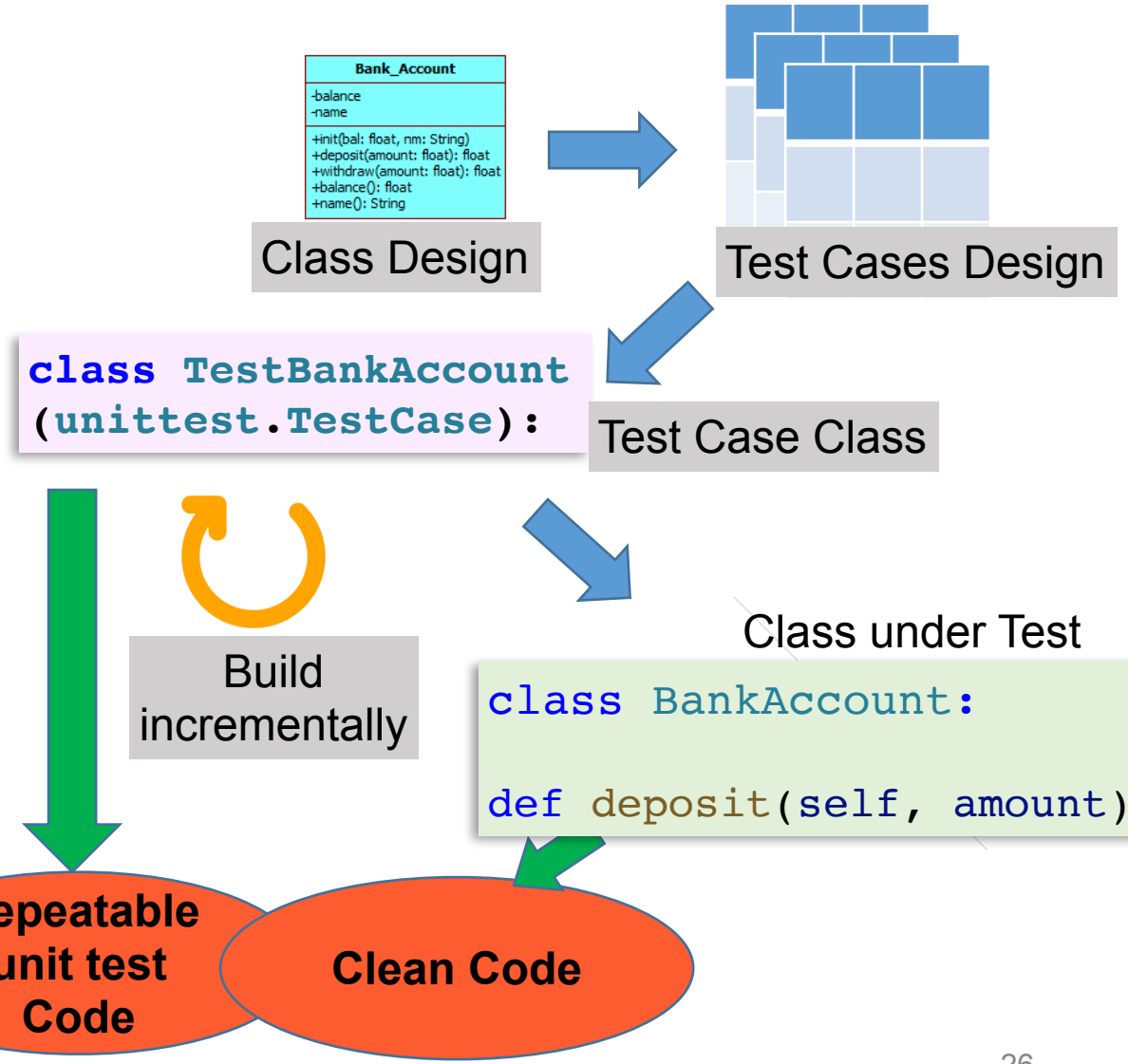
Typical process of using **unittest** for testing

Design ***test cases*** that will test the testable properties of the class

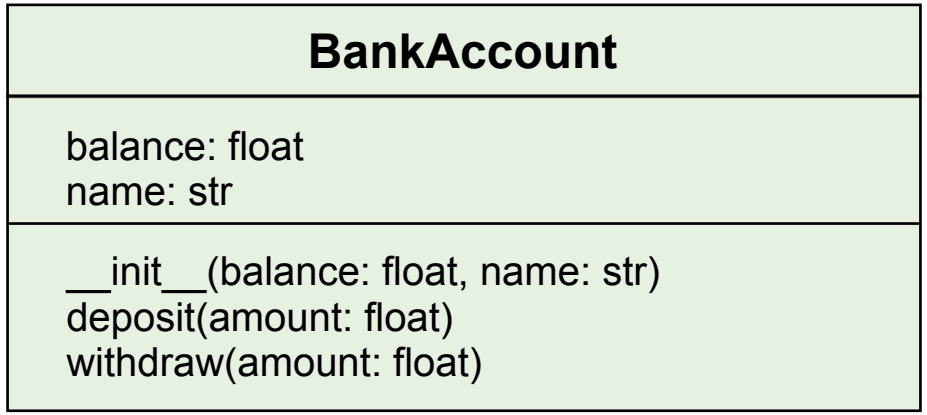
Build TestCase ***classes*** that contain ***test methods*** that implement the test cases

In the test methods introduce ***assertions*** to test the expected vs actual test results

Build the class under test to '**pass**' the test cases



TDD Example: Account Class Development
Class diagram



TDD Example: Account Class Development
Design Test Cases

Deposit Method:

1. Deposit amount = 0
   - Balance same as before
2. Deposit amount = positive number
   - Balance increased by the positive number
3. Deposit amount = negative number
   - Exception raised

Withdraw Method:

1. Withdraw amount = 0
   - Balance same as before
2. Withdraw amount = positive number less than balance
   - Balance decreased by the positive number
3. Withdraw amount = positive number greater than balance
   - Exception raised
4. Withdraw amount = negative number
   - Exception raised

Other test cases:

1. Test for same account objects
2. Test for different account objects
3. Test for different account objects with same attribute values

Add test cases iteratively

Live Demo

Let's develop BankAccount together using TDD!

BankAccount
balance: float name: str
__init__(balance: float, name: str) deposit(amount: float) withdraw(amount: float)

TDD Example: Account Class Development
BankAccount class: Initial Code

```
class BankAccount:
    def __init__(self, bal=0, nm=""):
        self.balance = bal
        self.name = nm

    def deposit(self, amount):
        pass

    def withdraw(self, amount):
        pass
```

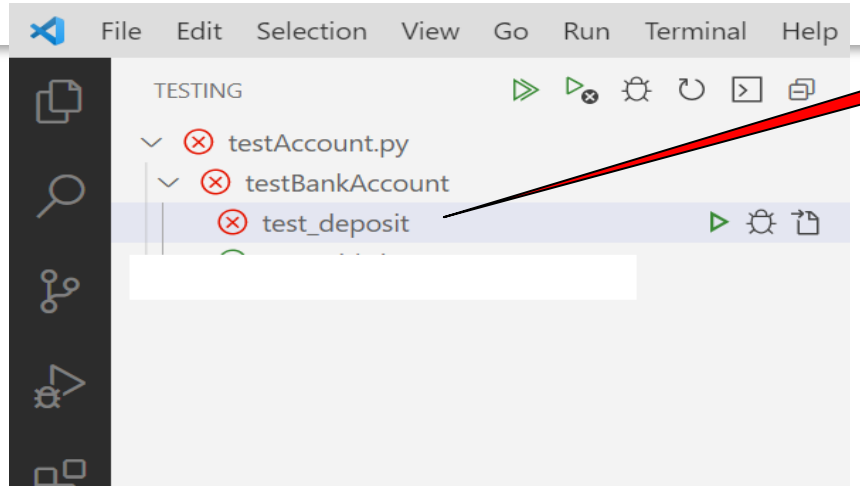
Empty methods that would fail tests

TDD Example: Account Class Development using 'unittest' framework
TestAccount class: To test BankAccount class

```
import unittest
from bank_account import BankAccount

class TestBankAccount(unittest.TestCase):
 def test_deposit(self):
    s = BankAccount(1000, "Chris")
    bal = s.balance
    self.assertEqual(bal, 1000)
    self.assertEqual(s.name, "Chris")
    s.deposit(500)
    self.assertEqual(s.balance, 1500)
```

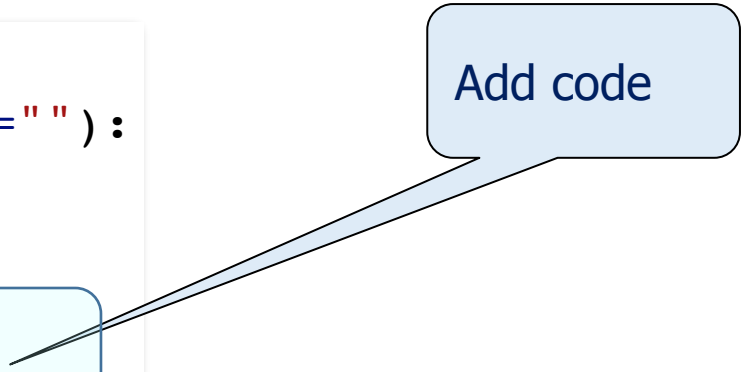
Import unittest and BankAccount

failed test



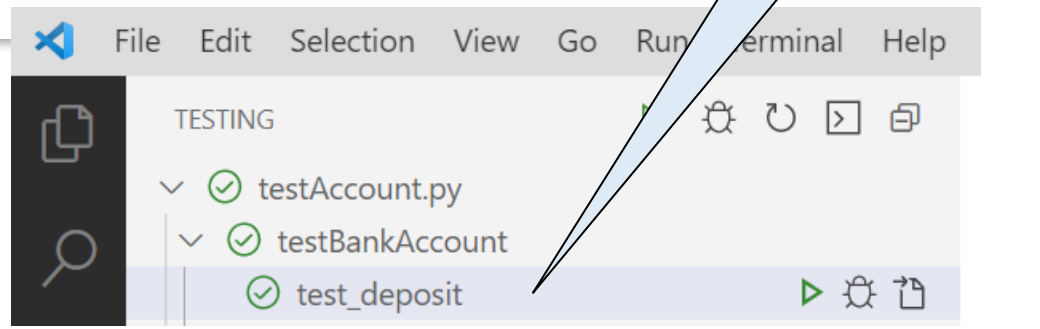
TDD Example: Account Class Development
BankAccount class: Incremental TDD Code

```
class BankAccount:
    def __init__(self, bal=0, nm=""):
        self.balance = bal
        self.name = nm

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        pass
```



TDD Example: Account Class Development using ‘unittest’ framework
TestAccount class: To test BankAccount class

```
import unittest
from bank_account import BankAccount;

class TestBankAccount(unittest.TestCase):
 def test_deposit(self):
    s = BankAccount(1000, "Chris")
    bal = s.balance
    self.assertEqual(bal, 1000)
    self.assertEqual(s.name, “Chris")
    s.deposit(500)
    self.assertEqual(s.balance, 1500)
```

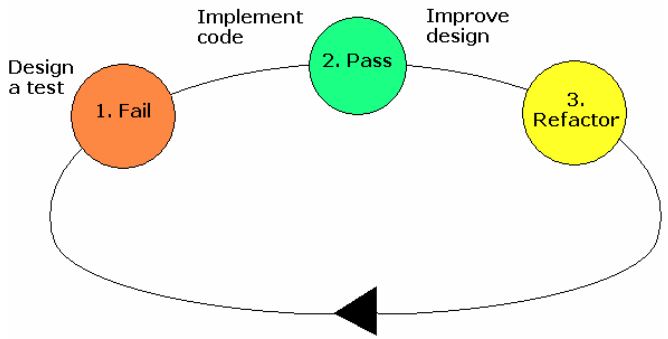


TDD Example: Account Class Development using 'unittest' framework
TestAccount class: To test BankAccount class

```
import unittest
from bank_account import BankAccount

class TestBankAccount(unittest.TestC
  def test_deposit(self):
     s = BankAccount(1000, "Chris")
     bal = s.balance
     self.assertEqual(bal, 1000)
     self.assertEqual(s.name, "Chris")
     s.deposit(500)
     self.assertEqual(s.balance, 1500)
 def test_withdraw(self):
     s = BankAccount(1500, "Chris")
     s.withdraw(1000)
     self.assertEqual(s.balance, 500)
     self.assertRaises(Exception, s.withdraw, 600)
```

```
class BankAccount:
. . .
    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
        else:
            raise Exception ("Insufficient balance"
```

Add more tests and code in the **TDD fashion**



TDD Example: Account Class Development
BankAccount class: TDD Code

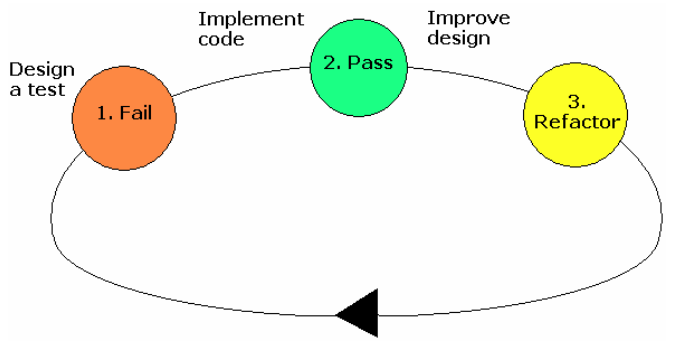
```
class BankAccount:
    def __init__(self, bal=0, nm=''):
        self.balance = bal
        self.name = nm

    def deposit(self, amount):
        self.balance += amount
```

```
    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
        else:
            raise Exception ("Insufficient balance")
```

TDD Example: Account Class Development using 'unittest' framework
TestAccount class with more tests: To test BankAccount class

```
import unittest
from bank_account import BankAccount;

class TestBankAccount(unittest.TestCase):
  def test_deposit(self):
     s = BankAccount(1000, "Chris")
     bal = s.balance
     self.assertEqual(bal, 1000)
     self.assertEqual(s.name, "Chris")
     s.deposit(500)
     self.assertEqual(s.balance, 1500)
 def test_withdraw(self):
     s = BankAccount(1500, "Chris")
     s.withdraw(1000)
     self.assertEqual(s.balance, 500)
     self.assertRaises(Exception, s.withdraw, 6
```

Add more tests and code in the **TDD fashion**




```
def test_EqualAccounts(self):
     s = BankAccount(1000, "Phris")
     s1 = BankAccount(1000, "Coskitt")
     self.assertTrue(s.balance > 0)
     self.assertIsNot(s, s1)
     self.assertIs(s, s)
```

TDD Example: Account Class Development
BankAccount class: TDD Code

```
class BankAccount:
    def __init__(self, bal=0, nm=''):
        self.balance = bal
        self.name = nm

    def deposit(self, amount):
        self.balance += amount
```

Clean Code!

```
    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
        else:
            raise Exception ("Insufficient balance")
```

unittest framework – Assert methods

- Basic Boolean Asserts
- Comparative Asserts
- Asserts for Collections

https://www.tutorialspoint.com/unittest_framework/unittest_framework_assertion.htm

unittest framework
Some Basic Boolean methods

Method	
assertEqual(arg1, arg2, msg = None)	Test that *arg1* and *arg2* are equal.
assertNotEqual(arg1, arg2, msg = None)	Test that *arg1* and *arg2* are not equal.
assertTrue(expr, msg = None)	Test that *expr* is true.
assertFalse(expr, msg = None)	Test that *expr* is false.
assertIs(arg1, arg2, msg = None)	Test that *arg1* and *arg2* evaluate to the same object.
assertIsNot(arg1, arg2, msg = None)	Test that *arg1* and *arg2* don't evaluate to the same object.
Others	

https://www.tutorialspoint.com/unittest_framework/unittest_framework_assertion.htm

unittest framework
Some Comparative assert methods

Method	
assertAlmostEqual (first, second, places = 7, msg = None, delta = None)	Test that *arg1* and *arg2* are equal.
assertNotAlmostEqual (first, second, places, msg, delta)	Test that first and second are not approximately equal by computing the difference
assertGreater (first, second, msg = None)	Test that *first* is greater than *second* depending on the method name.
assertLess (first, second, msg = None)	Test that *first* is less than *second* depending on the method name.
assertRegexpMatches (text, regexp, msg = None)	Test that a regexp search matches the text.
Others	

https://www.tutorialspoint.com/unittest_framework/unittest_framework_assertion.htm

unittest framework
Some Assert for Collection methods

Method	
assertListEqual (list1, list2, msg = None)	Tests that two lists are equal.
assertTupleEqual (tuple1, tuple2, msg = None)	Tests that two tuples are equal.
assertSetEqual (set1, set2, msg = None)	Tests that two sets are equal.
assertDictEqual (expected, actual, msg = None)	Test that two dictionaries are equal.

https://www.tutorialspoint.com/unittest_framework/unittest_framework_assertion.htm

Test Coverage

Test Coverage

- Test coverage measures how much of the software's code has been "exercised" by your test suite

- There are different types of coverage:
  - ***Statement coverage*** *– % of executable statements that have been executed*
  - ***Branch coverage*** *– % of decision branches (**if-then-else**) that have been executed*
  - ***Function coverage*** *– % of functions/methods that have been called*

- Increasing coverage helps ensure a larger portion of the codebase is tested
  - *E.g. by ensuring that buggy / under-tested branches aren't missed*

- Note that high coverage does not necessarily mean that the test suite is effective, see e.g. https://www.cs.ubc.ca/~rtholmes/papers/icse_2014_inozemtseva.pdf

Test Coverage – Example

```python
def calculate_discount(price, discount_rate):
    """Calculate the discounted price."""
    if price <= 0:
        raise ValueError("Price must be positive.")
    if discount_rate < 0 or discount_rate > 1:
        raise ValueError("Discount rate must be between 0 and 1.")

    discount = price * discount_rate
    discounted_price = price - discount
    return discounted_price

def apply_tax(price, tax_rate):
    """Apply tax to the price."""
    if tax_rate < 0 or tax_rate > 1:
        raise ValueError("Tax rate must be between 0 and 1.")
    return price * (1 + tax_rate)
```

```python
def test_calculate_discount(self):
    self.assertEqual(calculate_discount(100, 0.2), 80)
    self.assertRaises(ValueError, calculate_discount, -10, 0.2)
    self.assertRaises(ValueError, calculate_discount, 100, 1.5)

def test_apply_tax(self):
    self.assertEqual(apply_tax(100, 0.1), 110)
    self.assertRaises(ValueError, apply_tax, 100, -0.1)
    self.assertRaises(ValueError, apply_tax, 100, 1.5)
```

*What is the **statement**, **branch**, and **function coverage**?*

Test Coverage

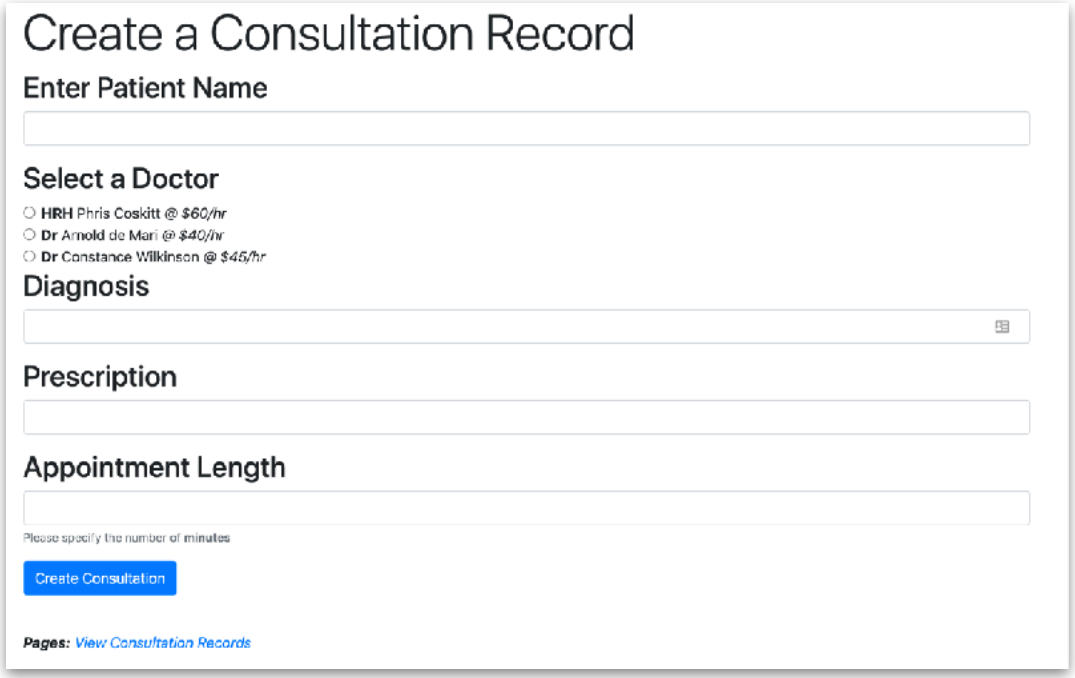
- The **Coverage.py** tool can be used to measure the test coverage of your Python programs
  - *https://coverage.readthedocs.io/*

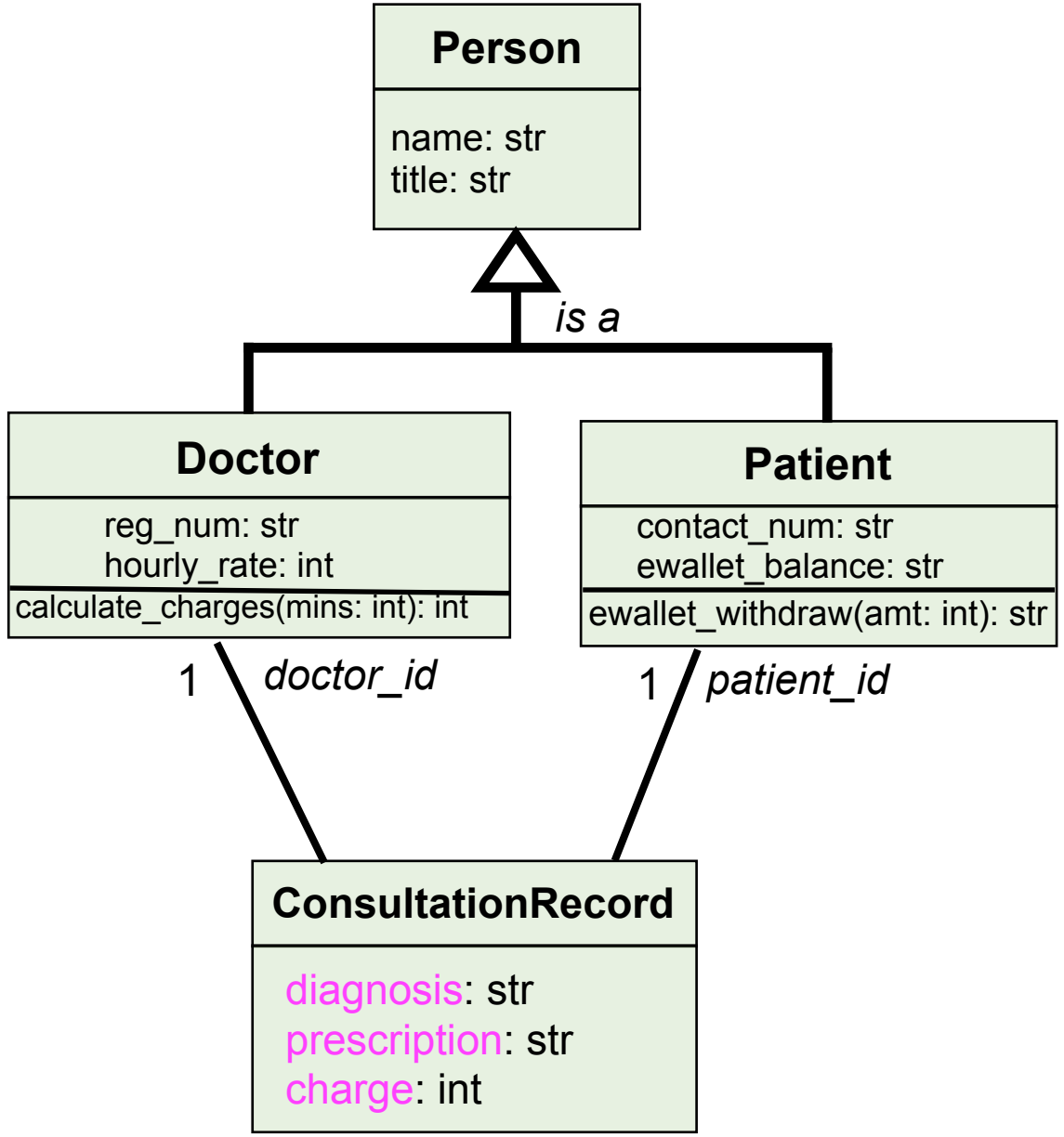
- It can be executed in a terminal:
  1. **Install:** python -m pip install coverage
  2. **Run:** coverage run -m unittest your_test_file.py
  3. **Generate Report:** coverage report
  4. **Generate HTML Report:** coverage html

Demo!

- **Tip:** use "# pragma: no cover" to exclude code from the analysis
  - *https://coverage.readthedocs.io/en/latest/excluding.html*

Recall our Flask app





*How do we **unit test** our Flask app?*

Demo!

Test Fixtures & Mocks

unittest framework – Test Fixtures

- Test methods often use same test data that need to be
  - Initialised at the start of the test method
  - Uninitialised at the end of the test method

- Common **Test Data** is called **Test Fixtures**

- **unittest** provides the following
  two methods to initialise and uninitialised Test Fixtures
  - **setUp()** – to initialise [called before every test_method()]
  - **tearDown()** – to uninitialise [called after every test_method()]

Live Demo

adding fixtures!

BankAccount
balance: float name: str
__init__(balance: float, name: str) deposit(amount: float) withdraw(amount: float)

TDD Example: Test Fixtures
TestAccount Class: setUp() and TearDown()

```
import unittest
from bank_account import BankAccount;

class TestBankAccount(unittest.TestCase)
    def setUp(self):
        self.s = BankAccount(1000, "Chris")
        self.s1 = BankAccount(1500, "Hyacin
        self.s2 = BankAccount(1000, "Daniel
    def tearDown(self):
        self.s = None
        self.s1 = None
        self.s2 = None
```

setUp() called **before every test_method**

tearDown() called **after every test_ method**

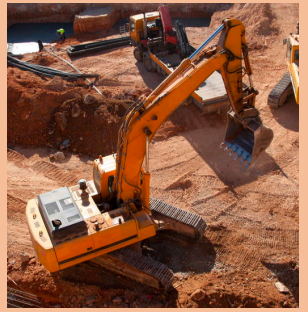
```
    def test_deposit(self):
        bal = self.s.balance
        self.assertEqual(bal, 1000)
        self.assertEqual(self.s.name, "Chris")
        self.s.deposit(500)
        self.assertEqual(self.s.balance, 1500)
    def test_withdraw(self):
        self.s1.withdraw(1000)
        self.assertEqual(self.s1.balance, 500)
        self.assertRaises(Exception, self.s1.withdraw, 600)
    def test_EqualAccounts(self):
        self.assertTrue(self.s.balance > 0)
        self.assertIsNot(self.s, self.s2)
        self.assertIs(self.s, self.s)
```

Test Fixtures in one place: Clean Code!

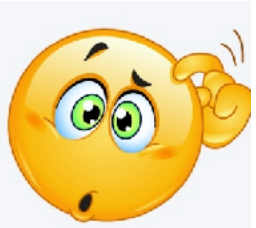
unittest framework – Mock framework

Not yet implemented

BankAccount
balance: float name: str
__init__(balance: float, name: str) deposit(amount: float) withdraw(amount: float) **interest(rate_computer: InterestRateComputer): float**

InterestRateComputer


How to test this method?



unittest framework – Mock framework
Need to Mock without creating Stubs…

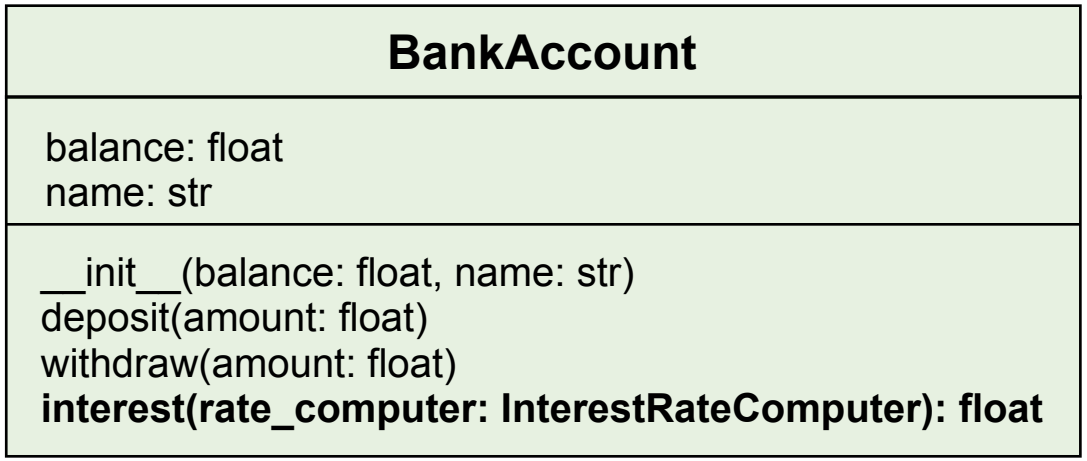
```
class BankAccount:
    def __init__(self, bal, nm):
. . .
    def deposit(self, amount):
. . .
    def withdraw(self, amount):
. . .
    def interest(self, rate_computer):
        return rate_computer.get_rate() * self.balance
```

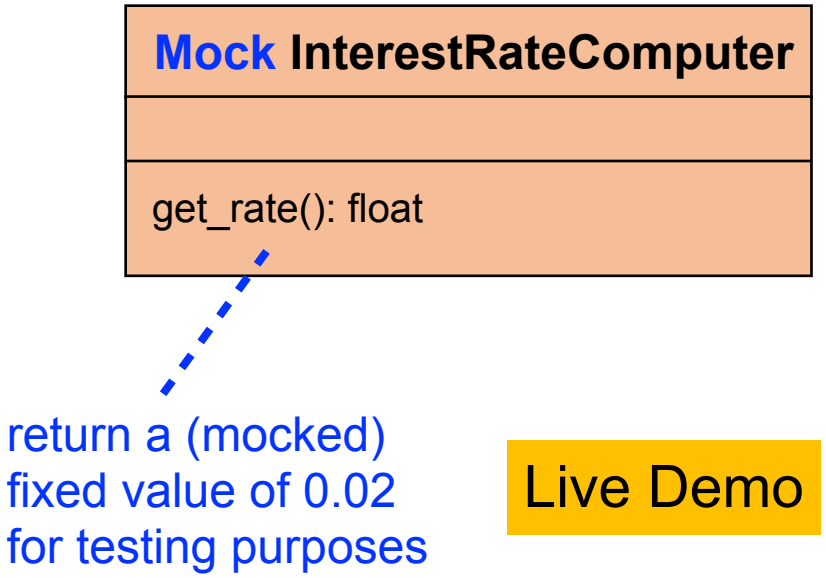
YET TO BE IMPLEMENTED!

```
class InterestRateComputer:
    #To be implemented
    def get_rate(self):
        pass
```

unittest framework – Mock framework

Need to Mock without creating Stubs…





Live Demo

unittest framework – Mock framework

- Mock class: Python ships with unittest.mock, a powerful part of the standard library
  - Used for stubbing dependencies and test doubles
  - Supports mocking side effects
- MagicMock is a subclass of Mock with all the magic methods pre-created and ready to use
- @Patch decorator is an alternative

https://docs.python.org/3/library/unittest.mock.html#patching-descriptors-and-proxy-objects
https://yeraydiazdiaz.medium.com/what-the-mock-cheatsheet-mocking-in-python-6a71db997832
https://semaphoreci.com/community/tutorials/getting-started-with-mocking-in-python

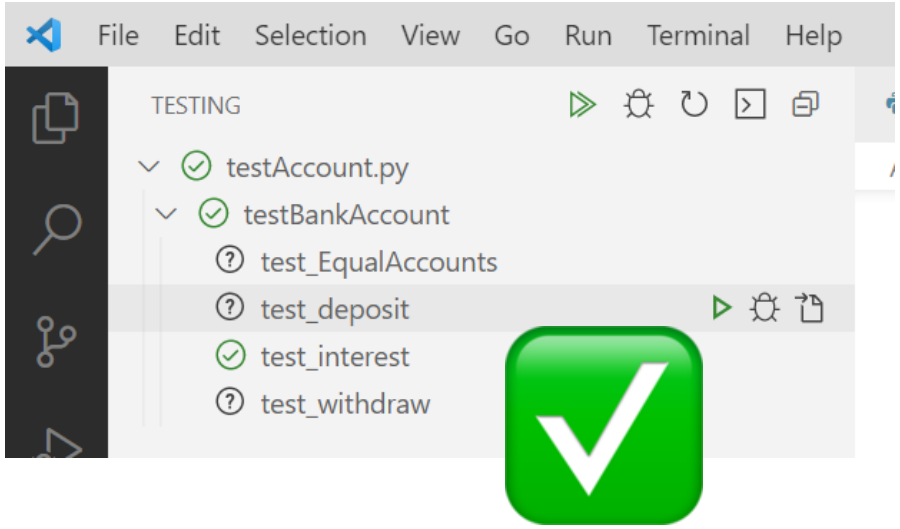
unittest framework – Mock framework
Mock using **MagicMock** class

```
class TestBankAccount(unittest.TestCase):
    . . .
    def test_interest(self):
        interest_computer_mock = InterestRateComputer()
        interest_computer_mock.get_rate = MagicMock(return_value = 0.02)
        self.assertEqual(self.s.interest(interest_computer_mock), self.s.balance*0.02)
```

Mock InterestRateComputer.get_rate() method to return a fixed value 0.02

YET TO BE IMPLEMENTED!

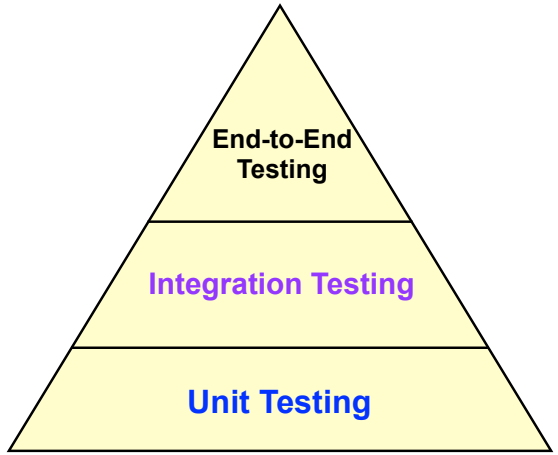
```
class InterestRateComputer:
    #To be implemented
    def get_rate(self):
        pass
```



Mocks in Unit Testing – Discussion

- Mock objects are useful in unit testing when it is impossible or impractical to incorporate a real object into the test
- E.g., objects that are not yet implemented; that supply non-deterministic results (time, temperature, random values); that interface with external infrastructure (databases, brokers, …)
- However – over-use of mocks in a test suite can dramatically increase the maintenance required on the tests themselves
- For example, bugs may be missed if the real class is refactored at some point but the mocked objects are not

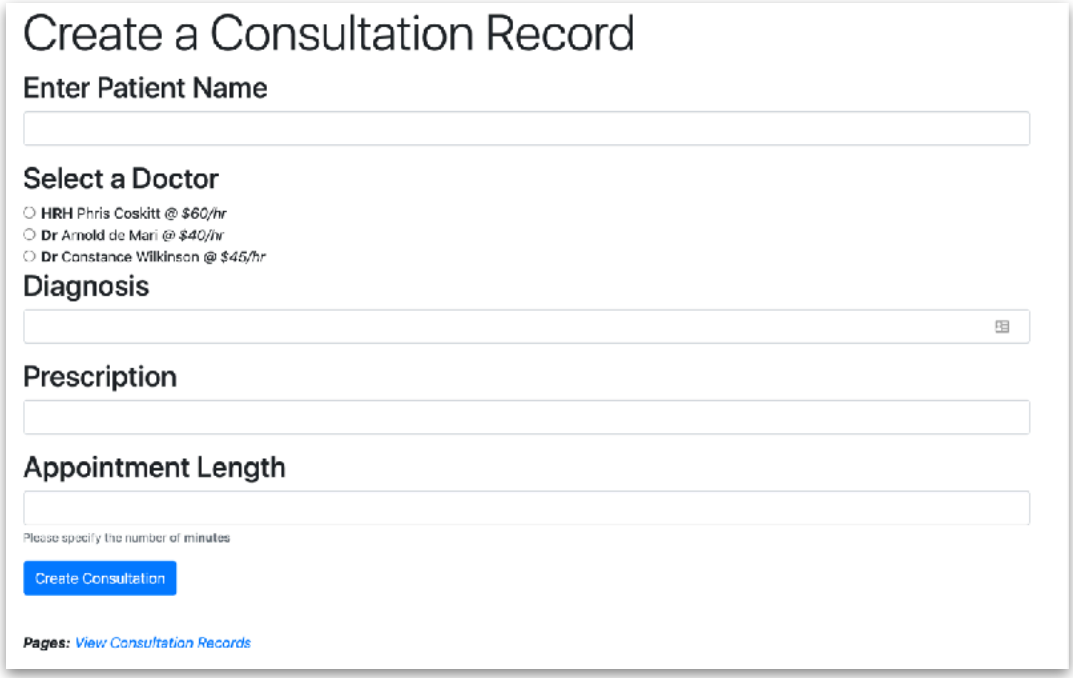
Integration and End-to-End Tests

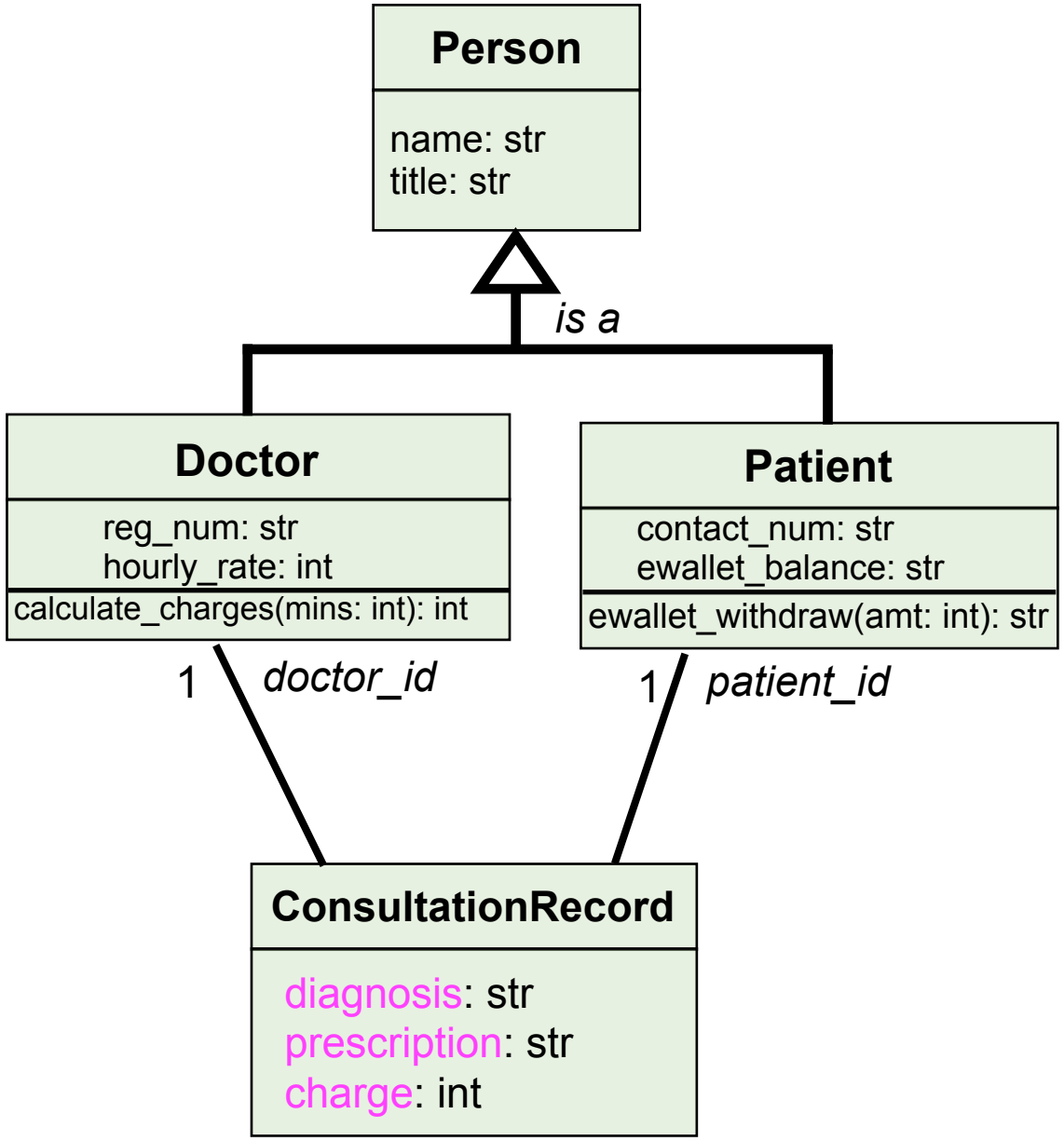


Integration Testing

- Unit testing helps to find bugs in individual classes and functions
- It's also important to test that the "parts" of your system integrate
  - *this is the purpose of automated integration testing*
- For example: do the Shopping Cart and Inventory Management modules of an online shopping app work together seamlessly?
  - *e.g. can you test that placing an order correctly reduces the stock?*
- Integration tests often involve databases and other infrastructure
  - *how do we satisfy the Isolation/Repeatability principles of FIRST?*

Back to our Flask app…





*How do we create **integration tests** for a Flask app?*

Demo!

End-to-End Testing

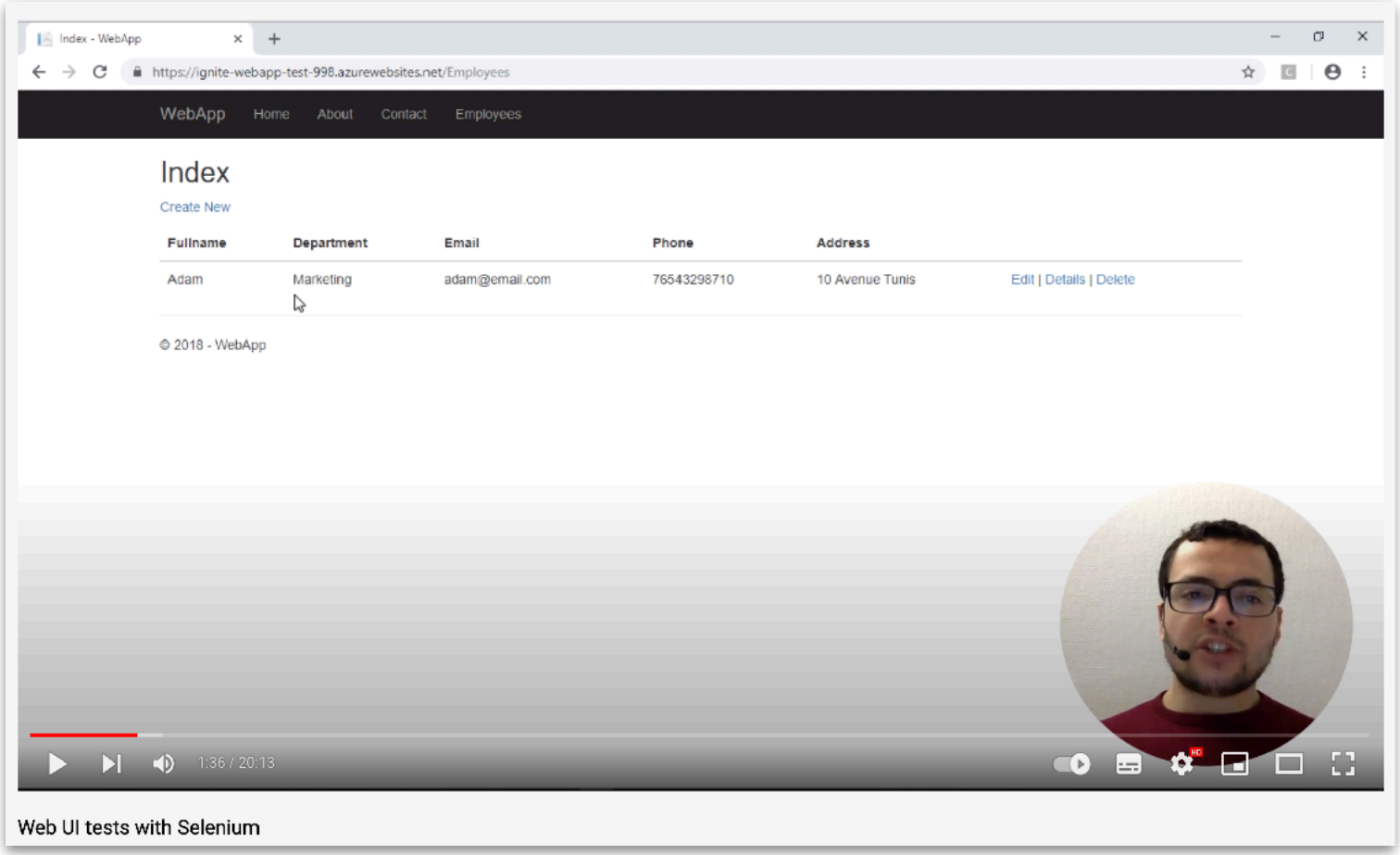
- End-to-end testing simulates real user workflows from start to finish
  - *Often involves interacting with the UI (e.g. simulating button clicks)*

- Goal is to validate that all subsystems work together as expected and to catch any issues missed by unit and integration testing

- Popular end-to-end testing frameworks include Selenium, Cypress, and Playwright
  - *These tools automate interactions through a real or simulated web browser*

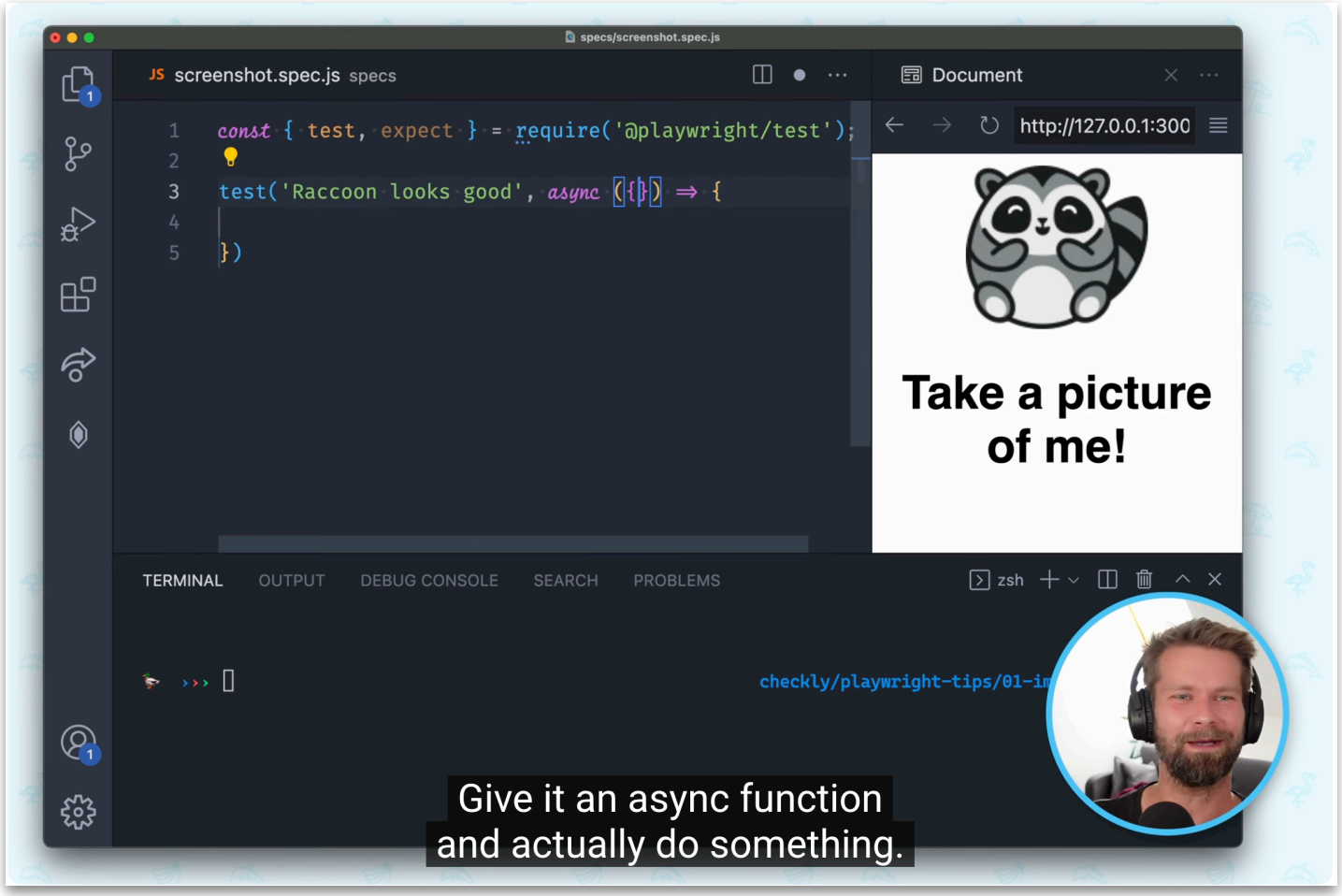
*Should we **only** do end-to-end testing and skip unit/integration?*

Example – End-to-End Testing with Selenium



https://youtu.be/mxDG4rzfEOI?t=92 **(1:32)**

Example – End-to-End Testing with Playwright

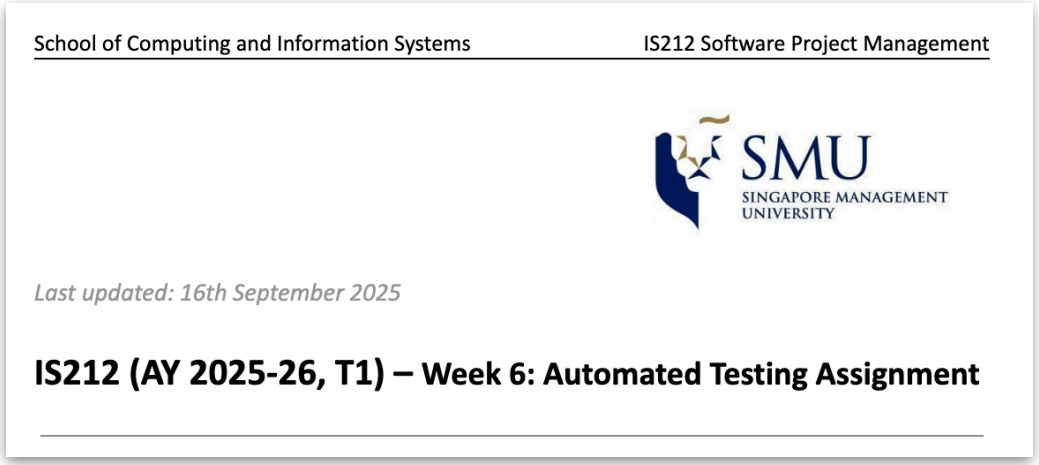


https://www.youtube.com/watch?v=Y1eTK-j66PU

Assignment & Reflections

Individual Assignment #2 – Automated Testing

- On **Friday 26th Sept @ 3:15pm**, the second individual assignment will be released
- Your task is to write unit tests for some provided code that achieve high code coverage and can detect injected faults
- Worth 4% of the grade for IS212
- Assignment is **auto-graded**, so please follow the instructions carefully
  - *Your tests **must** all run and pass on our code (as instructed), otherwise they receive 0 marks*



School of Computing and Information Systems — IS212 Software Project Management

SMU SINGAPORE MANAGEMENT UNIVERSITY

Last updated: 16th September 2025

IS212 (AY 2025-26, T1) – Week 6: Automated Testing Assignment

Due 23:59 on Friday Week 7
(1 sec late = 50% pen.)

Reminder – *Reflections*

- Please start filling in your weekly reflections now
- Complete them by two days after this class so that we have time to read them before we next meet ☺
- **http://smu.sg/is212-reflections-2025**