More Fun with Python

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Mutation testing: how we autograde your tests

"Mutation testing": programmers insert small, common bugs into their code and check whether the tests catch them.

We grade student tests by checking that they:

1. Pass on correct code
2. Fail on incorrect code

(1) is needed before moving on to (2)

Disclaimer: All bugs that we inserted into the incorrect code are intended to be simple and common. If the autograder says there is a bug that remains undetected by your tests, look for large missing test cases, rather than digging into obscure ways code can run incorrectly.

```python
def add(a: int, b: int) -> int:
    """Returns the sum of two integers."""
    return a + b

class TestAddFunction(unittest.TestCase):
    """Unit tests for the add function."""

    def test_add_positive_numbers(self) -> None:
        """Test adding two positive numbers."""
        self.fail()

    def test_add_negative_numbers(self) -> None:
        """Test adding two negative numbers."""
        self.assertEqual(add(-1, -1), -2)

    def test_add_mixed_numbers(self) -> None:
        """Test adding a positive and a
        negative number."""
        self.assertEqual(add(-1, 1), 0)

    def test_add_zero(self) -> None:
        """Test adding zero to a number."""
        self.assertEqual(add(0, 5), 5)
        self.assertEqual(add(5, 0), 5)
```

Poll: Why is this assignment submission not receiving full points?

1. The student implemented `add()` incorrectly.
2. Mutation testing: the student's tests don't cover enough cases.
3. The student's test fails on correct code, so mutation tests are not run.
4. It's something else -- pylint warnings, infinite loop, etc.

How many tests do I need?

For the grade? Enough to pass the mutation tests

In life? ... Consider all the ways the function might behave:

- Normal / happy case (expected inputs)
    - `assertEqual(5, add(2, 3))`
    - `assertNotEqual(1, add(2, 3))`
    - `assertEqual('A', calculateGrade(96))`
- Invalid inputs
    - `with self.assertRaises(ValueError): calculateGrade(-600)`
    - `with self.assertRaises(ValueError): add('two', 3)`
    - `with self.assertRaises(ValueError): get_area_of_rectangle(-1, 4)`

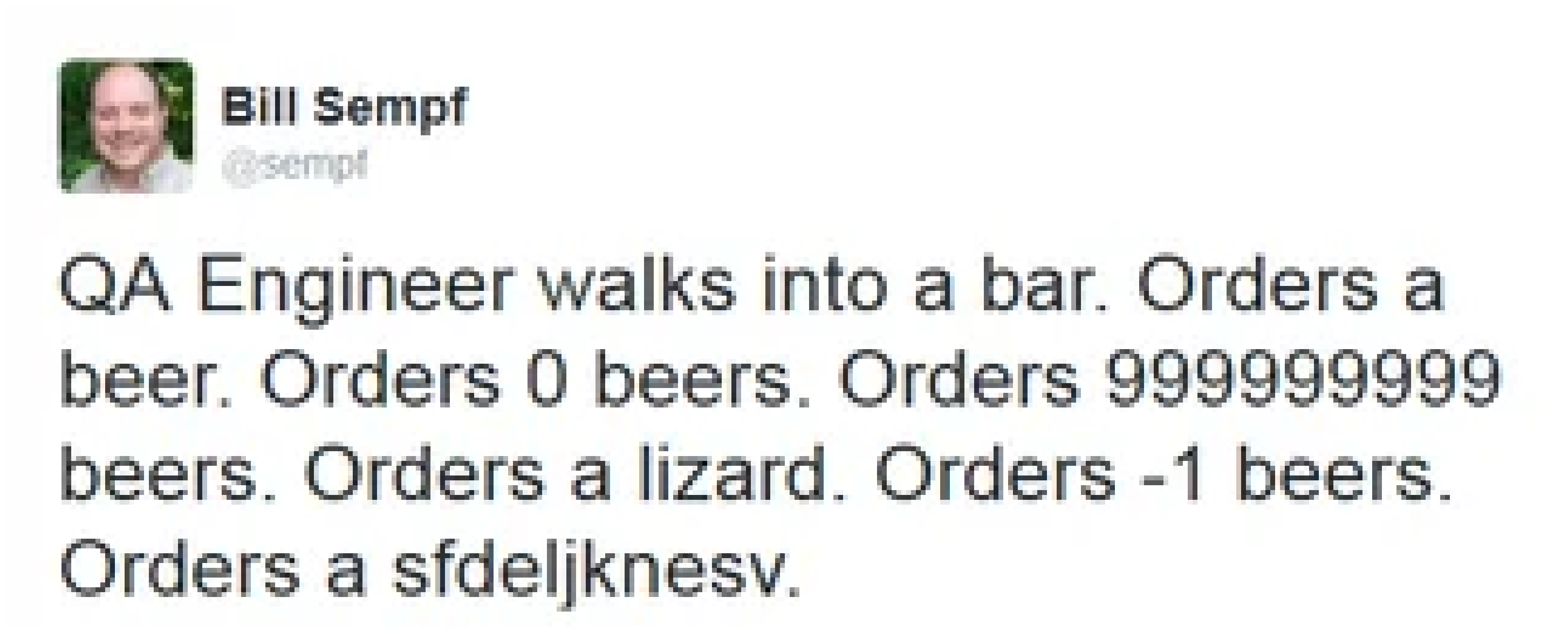
Edge cases

- Edge cases at the boundaries (almost invalid, but not quite)
    - `assertEqual(0, get_area_of_rectangle(0, 4))`
    - `assertEqual(0, divide(0, 1))`

If the function has conditionals, make sure to have test cases for each branch.

Poll: We're testing a function `calculateGrade(score: int) -> str` that returns a letter grade given a percentage. Which test case is most important to include?

1. `assertEqual('B+', calculateGrade(87))`
2. `assertEqual('F', calculateGrade(0))`
3. `with self.assertRaises(ValueError): calculateGrade(-600)`
4. All of these are equally important



Bill Sempf
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

Source: https://www.reddit.com/r/QualityAssurance/comments/3na0fq/qa_engineer_walks_into_a_bar

Let's write a function that...

Takes the name of a text file, and replaces its contents with the sarcastic version of the text.

We'll need to:

- Read all the content of a given text file
- Sarcastify the string (we know how to do that)
- Overwrite the text file with a new file that has the same name, with the sarcastic contents

Read and write data from text files

Read data from a file:

```python
with open('story.txt', 'r', encoding="utf-8") as file:
    for line in file.readlines():
        print(line)
```

Write to a file instead of reading --> use an option other than `'r'`:

- `open('story.txt', 'r')`: read the file
- `open('story.txt', 'w')`: write the file (overwrite it if it already exists)
- `open('story.txt', 'a')`: append to the end of the file (and create the file if it doesn't exist)

Write to the file using `file.write("Line to write to file")`.

None

`None` works like a value that represents the absence of a value.

```
bodyguard_name: str = None # doesn't have a value -- I don't have a bodyguard
```

It's different from "" or 0 (see Null Island)

Can store `None` in a list

```
grades: List[int] = [5, None, 0]
```

Cannot add `None` to a number or string

- `None + "hi"` does not work
- `len(None)` does not work

Use `Optional` to specify that a type might be `None`

```python
from typing import Optional

def get_number_or_None(hopefully_a_number: str) -> Optional[int]:
    try:
        return int(hopefully_a_number)
    except ValueError:
        return None
```

Using `try` / `except` safely

A control structure that we have not introduced until now:

```python
a: int = 4
b: int = 0

try:
    result = a / b
    print(result)
except ZeroDivisionError:
    print("Cannot divide by zero")
```

- Allows us to try to run risky code
- If an error is raised during that risky code, it jumps to the corresponding `except` block
- Only use it when absolutely necessary -- do not avoid fixing bugs with `try`

Places where `try` / `except` is commonly used:

- Converting values

```python
def get_user_age() -> int:
    """Get a numerical age from the user"""
    user_input: str = input("Enter your age: ")
    try:
        age: int = int(user_input)
        return age
    except ValueError:
        print("Please enter a valid number")
        return -1

def parse_json_safely(json_string: str) -> Any:
    """Convert data from JSON to a readable format"""
    try:
        return json.loads(json_string)
    except json.JSONDecodeError as e:
        print(f"Invalid JSON: {e}")
        return {}
```

- Operations that rely on external things like network requests or database operations
- File I/O (though `with` is better)
- `try` / `except` is an acceptable alternative to the built-in `self.assertRaises()`

Keywords in a `try` / `except` block:

- Each error type gets its own `except`
    - Error types as specific as possible (`ValueError` , not `Error`)
    - Okay to have multiple `except` s for the same `try`
    - One `except` block can handle multiple errors, if they require the same process: `except (ValueError, TypeError) as e:`
- Inside an `except` block, we may choose to `raise` a different error
- If there is a `finally` at the end, it is always run (whether the `try` was fully executed, or it jumped to the `except`)
- If there is an `else` at the end, then it is run only if the `try` was fully successful

Poll: What is output?

```python
def noodle(hopefully_a_number: str) -> None:
    try:
        num: int = int(hopefully_a_number)
        print('Cats rule')
    except AssertionError as e:
        print(f'{hopefully_a_number} is not a number')

noodle('hello')
```

1. Cats rule
2. hello is not a number
3. Cats rule

   hello is not a number
4. No output - it raises the error

Poll: What is output?

```python
def noodle(hopefully_a_number: str) -> None:
    try:
        num: int = int(hopefully_a_number)
        print('Cats rule')
    except ValueError as e:
        print(f'{hopefully_a_number} is not a number')

noodle('hello')
```

1. Cats rule
2. hello is not a number
3. Cats rule

   hello is not a number
4. No output - it raises the error

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?