# Designing Classes and Test Classes

## Welcome back to CS 2100!

Prof. Rasika Bhalerao

# Classes

- "nouns" (versus functions which are "verbs")
- encapsulate data and code
- achieve abstraction (mask details of implementation)
    - e.g., like how we push a button/turn a key to start a car without knowing how exactly it works
- Allow us to create our own new custom *type*

Here are some classes that are built in to Python (types that we already use):

| **Class (data type)** | **Object (an instance of a class)** |
|---|---|
| str | word: str = "hello" |
| list | items: list[int] = [1, 2, 3] |

# How to make a class

- Class header is `class Name:` (capital letter)
- Methods: functions inside a class
  - First parameter is `self`
- Attributes: variables shared among all methods
  - Name starts with `self.`
- Constructor: special method that is called when the object is "instantiated"
  - To initialize the attributes
  - `def __init__(self, <args>):`

```
class Pet:
    """Represents a household pet"""

    def __init__(self,
                 pet_name: str,
                 owner_name: str,
                 animal: str):

        self.name: str = pet_name
        self.owner: str = owner_name

        if animal == 'cat':
            self.sound: str = 'meow'
        elif animal == 'dog':
            self.sound = 'bark'
        else:
            self.sound = 'hello'

    def make_sound(self) -> str:
        """Returns the pet's sound"""
        return self.sound
```

```
class Pet:
    """Represents a household pet"""

    def __init__(self,
                 pet_name: str,
                 owner_name: str,
                 animal: str):

        self.name: str = pet_name
        self.owner: str = owner_name

        if animal == 'cat':
            self.sound: str = 'meow'
        elif animal == 'dog':
            self.sound = 'bark'
        else:
            self.sound = 'hello'

    def make_sound(self) -> str:
        """Returns the pet's sound"""
        return self.sound
```

- Instantiate an object (an instance) by putting parentheses after its name with the constructor's args
- Call its methods using the "dot operator" ( `.` )

Below:

1. Instantiate a `Pet` variable called `mini`
2. Call `mini`'s `make_sound()` method

```
mini: Pet = Pet(
    'Mini', 'Rasika', 'cat'
)
print(mini.make_sound())
```

Exercise: Let's define a class called `Cat`

- Attributes: self.name, self.age
- Constructor:
  - Take name as parameter
  - Make self.age equal 0
- Methods:
  - `birthday()` increments `self.age`
  - `make_sound()` returns the string `'meow'`, multiplied by the cat's age (with spaces in between)

Solution:

```
class Cat:
    """Represents a cat with a name"""
    def __init__(self, name: str):
        self.name = name
        self.age = 0

    def birthday(self) -> None:
        """Increments cat's age"""
        self.age += 1

    def make_sound(self) -> str:
        """Returns 'meow' multiplied by cat's age, with spaces in between"""
        return ('meow ' * self.age).strip()
```

## Poll: What does this output?

```
mini: Cat = Cat('Mini')
for year in range(3):
    mini.birthday()
print(mini.make_sound() + Cat('Mega').make_sound())
```

1. meow meow meow
2. (blank line)
3. Mini Mini Mini Mega
4. Mini Mega

# We give each class a corresponding test class

1. To write tests for a class named `Class`, create a class called `TestClass(unittest.TestCase)`
2. Put all the tests for `Class` inside `TestClass`
    - `self.assertEqual()` takes two args. If they are equal, it does nothing. If they are not equal, it raises an error
    - `self.assertAlmostEqual()` allows a small difference if used for `float`s
    - `self.assertRaises()` takes an error as an argument, and does nothing if the block of code raises that error. Else raises `AssertionError`
    - The name of each method that has tests in it should start with `test_`
3. Then outside of `TestClass`, call `unittest.main()`
4. Don't forget to `import unittest` at the top of the file

# Exercise: Let's write these classes and corresponding tests

### A class that represents a message in a group chat

- Constructor takes the contents of the message and the writer's user ID, and stores both in attributes
- Constructor also stores current time
  - `from datetime import datetime, UTC` and `now_utc = datetime.now(UTC)`
- A `display()` method that returns a `str` with the userID, timestamp, and message

### A class that represents a group chat

- Constructor takes no arguments, but stores a timestamp and starts a log of messages
- An `add_message()` method that takes a message and user ID and adds a message to the log
- A `display()` method that returns a `str` with all messages, formatted nicely

# Notes from that exercise:

- Still need `main()` to instantiate objects and run stuff
- `self.assertEqual()` calls `__eq__()` to check if the two things are equal
  - `==` also uses `__eq__()`

## Poll: What's wrong with this test?

```
def test_make_sound_works_after_four_years(self) -> None:
    self.assertEqual("", Cat('giga').make_sound())
```

1. The test runs, but it fails (that's not how the implementation is supposed to work)
2. Not all of the tests in this function always get executed
3. The function's name doesn't reflect what it tests
4. It's using the wrong type of test

# Poll: What's wrong with this test?

```
def test_make_sound_works_during_first_four_years(self) -> None:
    large: Cat = Cat('large')
    meows: str = ""
    for _ in range(4):
        self.assertEqual(meows, large.make_sound())
        large.birthday()
        meows = (meows + " meow").strip()
```

1. The test runs, but it fails (that's not how the implementation is supposed to work)
2. Not all of the tests in this function always get executed
3. The function's name doesn't reflect what it tests
4. It's using the wrong type of test

## Poll: What's wrong with this test?

```
def test_negative_area(self) -> None:
    with self.assertRaises(ValueError):
        self.assertEqual(-400, get_area_of_rectangle(-4, 100))
```

1. The test runs, but it fails (that's not how the implementation is supposed to work)
2. Not all of the tests in this function always get executed (it is possible for some tests to not run)
3. The function's name doesn't reflect what it tests
4. It's using the wrong type of test

# Using setUp and tearDown

`unittest` comes with methods that reduce redundancy / write cleaner tests:

- `def setUp(self) -> None:` runs before each test
- `def tearDown(self) -> None:` runs after each test
- `def setUpClass(cls) -> None:` runs once before any tests have run
- `def tearDownClass(cls) -> None:` runs once after all tests have run

Notes about `setUpClass(cls)` and `tearDownClass(cls)`:

- Need decorator `@classmethod` above the method
- arg is `cls`, not `self`

## Poll: Why does this break? Why is it better to use `setUp()`?

```
class TestShirt(unittest.TestCase):
    def __init__(self) -> None:
        self.shirt = Shirt(500, 'green')

    def test_set_size_works_for_positive_values(self) -> None:
        self.shirt.set_size(600)
        self.assertEqual(600, self.shirt.size)

    def test_cannot_set_size_to_negative_value(self) -> None:
      self.assertEqual(500, self.shirt.size)
      self.shirt.set_size(-700)
      self.assertEqual(500, self.shirt.size)
```

1. It unnecessarily tests the same thing multiple times
2. It requires the tests to be run in a certain order, which is not gurarantee
3. It doesn't test what the name implies it is testing
4. It is possible for some tests to not be run

## Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?