# CS 61A Object-Oriented Programming, String Representation

Fall 2021 Discussion 7: October 13, 2021 Solutions

# OOP

**Object-oriented programming** (OOP) is a programming paradigm that allows us to treat data as objects, like we do in real life.

For example, consider the **class** `Student`. Each of you as individuals is an **instance** of this class.

Details that all CS 61A students have, such as `name`, are called **instance variables**. Every student has these variables, but their values differ from student to student. A variable that is shared among all instances of `Student` is known as a **class variable**. For example, the `max_slip_days` attribute is a class variable as it is a property of all students.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are called **methods**. In this case, these actions would be methods of `Student` objects.

Here is a recap of what we discussed above:

- **class**: a template for creating objects
- **instance**: a single object created from a class
- **instance variable**: a data attribute of an object, specific to an instance
- **class variable**: a data attribute of an object, shared by all instances of a class
- **method**: a bound function that may be called on all instances of a class

Instance variables, class variables, and methods are all considered **attributes** of an object.

### Q1: WWPD: Student OOP

Below we have defined the classes `Professor` and `Student`, implementing some of what was described above. Remember that Python passes the `self` argument implicitly to methods when calling the method directly on an object.

```
class Student:

    max_slip_days = 3 # this is a class variable

    def __init__(self, name, staff):
        self.name = name # this is an instance variable
        self.understanding = 0
        staff.add_student(self)
        print("Added", self.name)

    def visit_office_hours(self, staff):
        staff.assist(self)
        print("Thanks, " + staff.name)

class Professor:

    def __init__(self, name):
        self.name = name
        self.students = {}

    def add_student(self, student):
        self.students[student.name] = student

    def assist(self, student):
        student.understanding += 1

    def grant_more_slip_days(self, student, days):
        student.max_slip_days = days
```

What will the following lines output?

```
>>> callahan = Professor("Callahan")
>>> elle = Student("Elle", callahan)
```

Added Elle

```
>>> elle.visit_office_hours(callahan)
```

Thanks, Callahan

```
>>> elle.visit_office_hours(Professor("Paulette"))
```

Thanks, Paulette

```
>>> elle.understanding
```

2

```
>>> [name for name in callahan.students]
```

['Elle']

```
>>> x = Student("Vivian", Professor("Stromwell")).name
```

Added Vivian

```
>>> x
```

'Vivian'

```
>>> [name for name in callahan.students]
```

['Elle']

```
>>> elle.max_slip_days
```

3

```
>>> callahan.grant_more_slip_days(elle, 7)
>>> elle.max_slip_days
```

7

```
>>> Student.max_slip_days
```

3

### Q2: Keyboard

We'd like to create a `Keyboard` class that takes in an arbitrary number of `Button`s and stores these `Button`s in a dictionary. The keys in the dictionary will be ints that represent the postition on the `Keyboard`, and the values will be the respective

`Button`. Fill out the methods in the `Keyboard` class according to each description, using the doctests as a reference for the behavior of a `Keyboard`.

```
class Button:
    def __init__(self, pos, key):
        self.pos = pos
        self.key = key
        self.times_pressed = 0

class Keyboard:
    """A Keyboard takes in an arbitrary amount of buttons, and has a
    dictionary of positions as keys, and values as Buttons.
    >>> b1 = Button(0, "H")
    >>> b2 = Button(1, "I")
    >>> k = Keyboard(b1, b2)
    >>> k.buttons[0].key
    'H'
    >>> k.press(1)
    'I'
    >>> k.press(2) # No button at this position
    ''
    >>> k.typing([0, 1])
    'HI'
    >>> k.typing([1, 0])
    'IH'
    >>> b1.times_pressed
    2
    >>> b2.times_pressed
    3
    """
    def __init__(self, *args):
        self.buttons = {}
        for button in args:
            self.buttons[button.pos] = button

    def press(self, info):
        """Takes in a position of the button pressed, and
        returns that button's output."""
        if info in self.buttons.keys():
            b = self.buttons[info]
            b.times_pressed += 1
            return b.key
        return ''

    def typing(self, typing_input):
        """Takes in a list of positions of buttons pressed, and
        returns the total output."""
        accumulate = ''
        for pos in typing_input:
            accumulate+=self.press(pos)
        return accumulate
```

# Inheritance

To avoid redefining attributes and methods for similar classes, we can write a single **base class** from which the similar classes **inherit**. For example, we can write a class called **Pet** and define **Dog** as a **subclass** of **Pet**:

```
class Pet:

    def __init__(self, name, owner):
        self.is_alive = True    # It's alive!!!
        self.name = name
        self.owner = owner

    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")

    def talk(self):
        print(self.name)

class Dog(Pet):

    def talk(self):
        super().talk()
        print('This Dog says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class **is a** more specific version of the other: a dog **is a** pet (We use **is a** to describe this sort of relationship in OOP languages, and not to refer to the Python `is` operator).

Since `Dog` inherits from `Pet`, the `Dog` class will also inherit the `Pet` class's methods, so we don't have to redefine `__init__` or `eat`. We do want each `Dog` to `talk` in a `Dog`-specific way, so we can **override** the `talk` method.

We can use `super()` to refer to the superclass of `self`, and access any superclass methods as if we were an instance of the superclass. For example, `super().talk()` in the `Dog` class will call the `talk()` method from the `Pet` class, but passing the `Dog` instance as the `self`.

This is a little bit of a simplification, and if you're interested you can read more in the Python documentation on `super`.

### Q3: Cat

Below is a skeleton for the `Cat` class, which inherits from the `Pet` class. To complete the implementation, override the `__init__` and `talk` methods and add a new `lose_life` method.

> Hint: You can call the `__init__` method of `Pet` (the superclass of `Cat`) to set a cat's `name` and `owner`.

```
class Cat(Pet):

    def __init__(self, name, owner, lives=9):
        super().__init__(name, owner)
        self.lives = lives

    def talk(self):
        """Print out a cat's greeting.

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
        print(self.name + ' says meow!')

    def lose_life(self):
        """Decrements a cat's life by 1. When lives reaches zero,
        is_alive becomes False. If this is called after lives has
        reached zero, print 'This cat has no more lives to lose.'
        """
        if self.lives > 0:
            self.lives -= 1
            if self.lives == 0:
                self.is_alive = False
        else:
            print("This cat has no more lives to lose.")
```

**Q4: NoisyCat**

More cats! Fill in this implementation of a class called `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot – twice as much as a regular `Cat`! If you'd like to test your code, feel free to copy over your solution to the `Cat` class above.

```
class NoisyCat(Cat): # Fill me in!
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        super().__init__(name, owner, lives)
        # No, this method is not necessary because NoisyCat already
    inherits Cat's __init__ method

    def talk(self):
        """Talks twice as much as a regular cat.
        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
        super().talk()
        super().talk()
```

# Class Methods

Now we'll try out another feature of Python classes: class methods. A method can be turned into a class method by adding the classmethod decorator. Then, instead of receiving the instance as the first argument (`self`), the method will receive the class itself (`cls`).

Class methods are commonly used to create "factory methods": methods whose job is to construct and return a new instance of the class.

For example, we can add a `robo_factory` class method to our `Dog` class that makes robo-dogs:

```
class Dog(Pet):
    # With the previously defined methods not written out
    @classmethod
    def robo_factory(cls, owner):
        return cls("RoboDog", owner)
```

Then a call to `Dog.robo_factory('Sally')` would return a new `Dog` instance with the name "RoboDog" and owner "Sally".

### Q5: Cat Adoption

Now you can implement the `adopt_random_cat` method below, which should construct a cat with a random name and lives. To generate random values, you can use functions like `random.choice` and `random.randint` from the `random` module.

```
import random as random

class Cat(Pet):
    def __init__(self, name, owner, lives=9):
        super().__init__(name, owner)
        self.lives = lives

    # Insert other previously defined methods here

    @classmethod
    def adopt_random_cat(cls, owner):
        """
        Returns a new instance of a Cat with the given owner,
        a randomly chosen name and a random number of lives.
        >>> randcat = Cat.adopt_random_cat("Ifeoma")
        >>> isinstance(randcat, Cat)
        True
        >>> randcat.owner
        'Ifeoma'
        """
        cat_name = random.choice(["felix", "bugs", "grumpy"])
        num_lives = random.randint(1, 20)
        return cls(cat_name, owner, num_lives)
```

# Representation: Repr, Str

There are two main ways to produce the "string" of an object in Python: `str()` and `repr()`. While the two are similar, they are used for different purposes.

`str()` is used to describe the object to the end user in a "Human-readable" form, while `repr()` can be thought of as a "Computer-readable" form mainly used for debugging and development.

When we define a class in Python, `__str__` and `__repr__` are both built-in methods for the class.

We can call those methods using the global built-in functions `str(obj)` or `repr(obj)` instead of dot notation, `obj.__repr__()` or `obj.__str__()`.

In addition, the `print()` function calls the `__str__` method of the object, while simply calling the object in interactive mode calls the `_repr__` method.

Here's an example:

```
class Rational:

    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return f'{self.numerator}/{self.denominator}'

    def __repr__(self):
        return f'Rational({self.numerator},{self.denominator})'

>>> a = Rational(1, 2)
>>> str(a)
'1/2'
>>> repr(a)
'Rational(1,2)'
>>> print(a)
1/2
>>> a
Rational(1,2)
```

**Q6: WWPD: Repr-esentation**

```
class A:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
         return self.x

    def __str__(self):
         return self.x * 2

class B:
    def __init__(self):
         print('boo!')
         self.a = []

    def add_a(self, a):
         self.a.append(a)

    def __repr__(self):
         print(len(self.a))
         ret = ''
         for a in self.a:
             ret += str(a)
         return ret
```

Given the above class definitions, what will the following lines output?

```
>>> A('one')
```

one

```
>>> print(A('one'))
```

oneone

```
>>> repr(A('two'))
```

'two'

```
>>> b = B()
```

boo!

```
>>> b.add_a(A('a'))
>>> b.add_a(A('b'))
>>> b
```

2

aabb