CS 61A Fall 2023

OOP, String Representation

Discussion 7: October 11, 2023

Note: For formal explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

OOP

Here's a recap of the OOP vocab we've learned so far:

- **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: Legally Blonde OOP

Below we have defined the classes `Student` and `Professor`. Remember that Python passes the `self` argument implicitly to methods when calling the method directly on an object.

```
class Student:

    extension_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_extension_days(self, student, days):
        student.extension_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'

```
>>> elle.extension_days
```

3

```
>>> callahan.grant_more_extension_days(elle, 7)
>>> elle.extension_days
```

7

```
>>> Student.extension_days
```

3

In OOP problems like this one, it is often helpful to draw an environment diagram to keep track of how instance/class variables change as we execute the code (Note: Environment diagrams related to OOP are not in scope. They are just a helpful tool.). In OOP environment diagrams, we typically draw boxes to represent the different objects created in the problem and their instance variables. For this problem, we would draw two boxes. One box to represent the `callahan` object and its instance variables (`name` and `students`). Another box to represent the `elle` object and its instance variables (`name` and `understanding`). We also typically draw boxes to represent the different classes defined in the problem and their class variables. For this problem, we would draw two more boxes. One to represent the `Student` class and its class variable (`extension_days`). Another one to represent the `Professor` class and its class variables (it has none). As we execute the code line by line, we should update the instance/class variables of each object/class in our environment diagram accordingly.

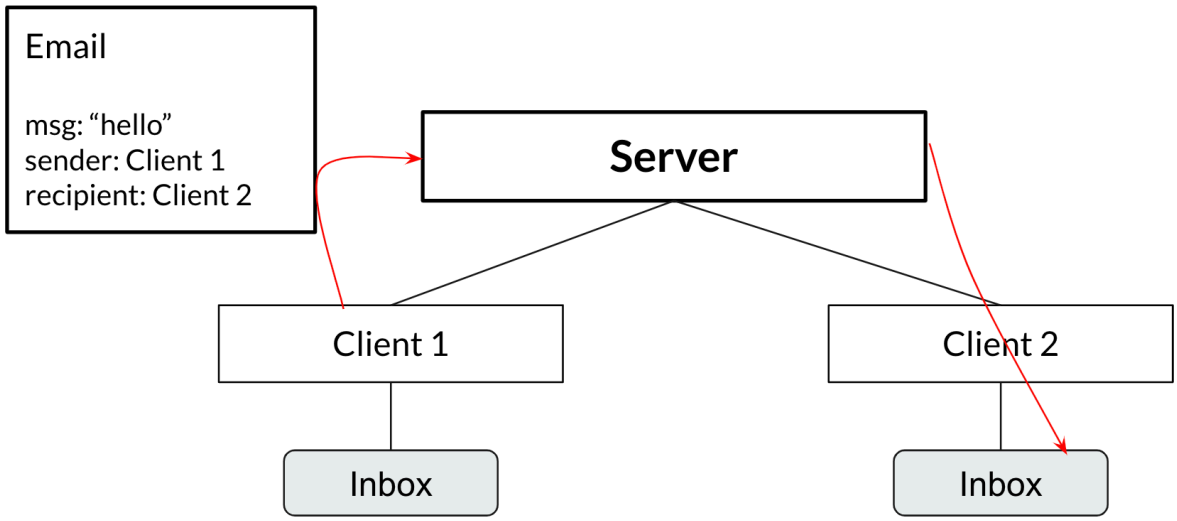
Here is a link to a very detailed environment diagram for this problem. You're environment diagram doesn't have to be as detailed as this one, nor should it be. The important thing is to pay attention to how to the instance/class variables change as we execute each line of code.

Q2: Email

We would like to write three different classes (`Server`, `Client`, and `Email`) to simulate a system for sending and receiving emails. A `Server` has a dictionary mapping client names to `Client` objects, and can both send `Email`s to `Client`s in the `Server` and register new `Client`s. A `Client` can both compose emails (which first creates a new `Email` object and then sends it to the recipient client through the server) and receive an email (which places an email into the client's inbox).

Emails will only be sent/received within the same server, so clients will always use the server they're registered in to send emails to other clients that are registered in the same rerver.

An example flow: A `Client` object (Client 1) composes an `Email` object with message `"hello"` with recipient Client 2, which the `Server` routes to Client 2's inbox.



Email example

Fill in the definitions below to finish the implementation!

```
class Email:
    """
    Every email object has 3 instance attributes: the
    message, the sender name, and the recipient name.
    >>> email = Email('hello', 'Alice', 'Bob')
    >>> email.msg
    'hello'
    >>> email.sender_name
    'Alice'
    >>> email.recipient_name
    'Bob'
    """
    def __init__(self, msg, sender_name, recipient_name):
        self.msg = msg
        self.sender_name = sender_name
        self.recipient_name = recipient_name
```

```
class Client:
    """
    Every Client has three instance attributes: name (which is
    used for addressing emails to the client), server
    (which is used to send emails out to other clients), and
    inbox (a list of all emails the client has received).

    >>> s = Server()
    >>> a = Client(s, 'Alice')
    >>> b = Client(s, 'Bob')
    >>> a.compose('Hello, World!', 'Bob')
    >>> b.inbox[0].msg
    'Hello, World!'
    >>> a.compose('CS 61A Rocks!', 'Bob')
    >>> len(b.inbox)
    2
    >>> b.inbox[1].msg
    'CS 61A Rocks!'
    """
    def __init__(self, server, name):
        self.inbox = []
        self.server = server
        self.name = name
        self.server.register_client(self, self.name)

    def compose(self, msg, recipient_name):
        """Send an email with the given message msg to the given recipient client."""
        email = Email(msg, self.name, recipient_name)
        self.server.send(email)

    def receive(self, email):
        """Take an email and add it to the inbox of this client."""
        self.inbox.append(email)
```

```
class Server:
    """
    Each Server has one instance attribute: clients (which
    is a dictionary that associates client names with
    client objects).
    """
    def __init__(self):
        self.clients = {}

    def send(self, email):
        """
        Take an email and put it in the inbox of the client
        it is addressed to.
        """
        client = self.clients[email.recipient_name]
        client.receive(email)

    def register_client(self, client, client_name):
        """
        Takes a client object and client_name and adds them
        to the clients instance attribute.
        """
        self.clients[client_name] = client
```

Q3: Keyboard

Below is the definition of a `Button` class, which represents a button on a keyboard. It has three attributes: `pos` (numerical position of the button on the keyboard), `key` (the letter of the button), and `times_pressed` (the number of times the button is pressed).

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

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 `int`s that represent the position on the `Keyboard`, and the values will be the respective `Button`. Fill out the methods in the `Keyboard` class according to each description.

Important: Utilize the doctests as a reference for the behavior of a `Keyboard` instance.

- **Hint:** You can iterate over `*args` as if it were a list.

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

class Keyboard:
    """A Keyboard stores an arbitrary number of Buttons in a dictionary.
    Each dictionary key is a Button's position, and each dictionary
    value is the corresponding Button.
    >>> b1, b2 = Button(5, "H"), Button(7, "I")
    >>> k = Keyboard(b1, b2)
    >>> k.buttons[5].key
    'H'
    >>> k.press(7)
    'I'
    >>> k.press(0) # No button at this position
    ''
    >>> k.typing([5, 7])
    'HI'
    >>> k.typing([7, 5])
    '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, pos):
        """Takes in a position of the button pressed, and
        returns that button's output."""
        if pos in self.buttons.keys():
            b = self.buttons[pos]
            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

Recall that a *subclass* (child class) by default inherits all of the methods and class attributes of its *superclass* (parent class). The subclass can override methods and class attributes by redefining them. `super()` can be used to access the methods and class attributes of the parent class.

Q4: That's inheritance, init?

Let's say we want to create a class `Monarch` that inherits from another class, `Butterfly`. We've partially written an `__init__` method for `Monarch`. For each of the following options, state whether it would correctly complete the method so that every instance of `Monarch` has all of the instance attributes of a `Butterfly` instance. You may assume that a monarch butterfly has the default value of 2 wings.

```
class Butterfly():
    def __init__(self, wings=2):
        self.wings = wings

class Monarch(Butterfly):
    def __init__(self):
        super().__init__()
        self.colors = ['orange', 'black', 'white']
```

```
super.__init__()
```

No, because the `super` function must be called with parentheses.

```
super().__init__()
```

Yes.

```
Butterfly.__init__()
```

No, because we must explicitly pass in `self` as an argument.

```
Butterfly.__init__(self)
```

Yes.

Some butterflies like the Owl Butterfly have adaptations that allow them to mimic other animals with their wing patterns. Let's write a class for these `MimicButterflies`. In addition to all of the instance variables of a regular `Butterfly` instance, these should also have an instance variable `mimic_animal` describing the name of the animal they mimic. Fill in the blanks in the lines below to create this class.

```
class MimicButterfly(Butterfly):
    def __init__(self, mimic_animal):
        super().__init__()
        self.mimic_animal = mimic_animal
```

Q5: Cat

Below is the implementation of a `Pet` class. Each pet has three instance attributes (`is_alive`, `name`, and `owner`), as well as two class methods (`eat` and `talk`).

```
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)
```

Implement the `Cat` class, which inherits from the `Pet` class seen above. 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`.

Hint: The `__init__` method can be called at any point and used just like any other method.

```
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.")
    def revive(self):
        """Revives a cat from the dead. The cat should now have
        9 lives and is_alive should be true. Can only be called
        on a cat that is dead. If the cat isn't dead, print
        'This cat still has lives to lose.'
        """
        if not self.is_alive:
            self.__init__(self.name, self.owner)
        else:
            print('This cat still has lives to lose.')
```

Q6: NoisyCat

More cats! Fill in this implementation of a class called `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot: in fact, it talks 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()
```

Representation: Repr, Str

Recall that, for any given object `obj`:

- `repr(obj)` is used to get a formal representation of `obj`, which is displayed when `obj` is evaluated directly in the interpreter. The `__repr__` method of `obj` defines the output of `repr(obj)`.
- `str(obj)` is used to get a human-readable representation of `obj`, which is displayed when `print(obj)` is evaluated directly in the interpreter. The `__str__` method of `obj` defines the output of `str(obj)`.

Q7: WWPD: Repr-esentation

Note: This is not the typical way `repr` is used, nor is this way of writing `repr` recommended, this problem is mainly just to make sure you understand how `repr` and `str` work.

```
class Car:
    def __init__(self, color):
        self.color = color

    def __repr__(self):
         return self.color

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

class Garage:
    def __init__(self):
         print('Vroom!')
         self.cars = []

    def add_car(self, car):
         self.cars.append(car)

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

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

```
>>> Car('red')
```

red

```
>>> print(Car('red'))
```

redred

```
>>> repr(Car('blue'))
```

'blue'

```
>>> g = Garage()
```

Vroom!

```
>>> g.add_car(Car('red'))
>>> g.add_car(Car('blue'))
>>> g
```

2

redredblueblue

Q8: Cat Representation

Now let's implement the `__str__` and `__repr__` methods for the `Cat` class from earlier so that they exhibit the following behavior:

```
>>> cat = Cat("Felix", "Kevin")
>>> cat
Felix, 9 lives
>>> cat.lose_life()
>>> cat
Felix, 8 lives
>>> print(cat)
Felix
```

```
# (The rest of the Cat class is omitted here, but assume all methods from the Cat class
    above are implemented)
    def __repr__(self):
        return self.name + ", " + str(self.lives) + " lives"
    def __str__(self):
        return self.name
```

Appendix: Explanation of Material

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 `extension_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.

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`.

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 and displays the returned string **with the quotations removed**, while simply calling the object in interactive mode in the interpreter calls the `_repr__` method and displays the returned string **with the quotations removed**.

Here are some examples:

```
class Rational:

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

    def __str__(self):
        return str(self.numerator) + '/' + str(self.denominator)

    def __repr__(self):
        return 'Rational' + '(' + str(self.numerator) + ',' + str(self.denominator) + ')'

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