CS 61A Spring 2023

Object-Oriented Programming

Discussion 7: March 8, 2023

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.

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:

    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'

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

['Elle']

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

3

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

7

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

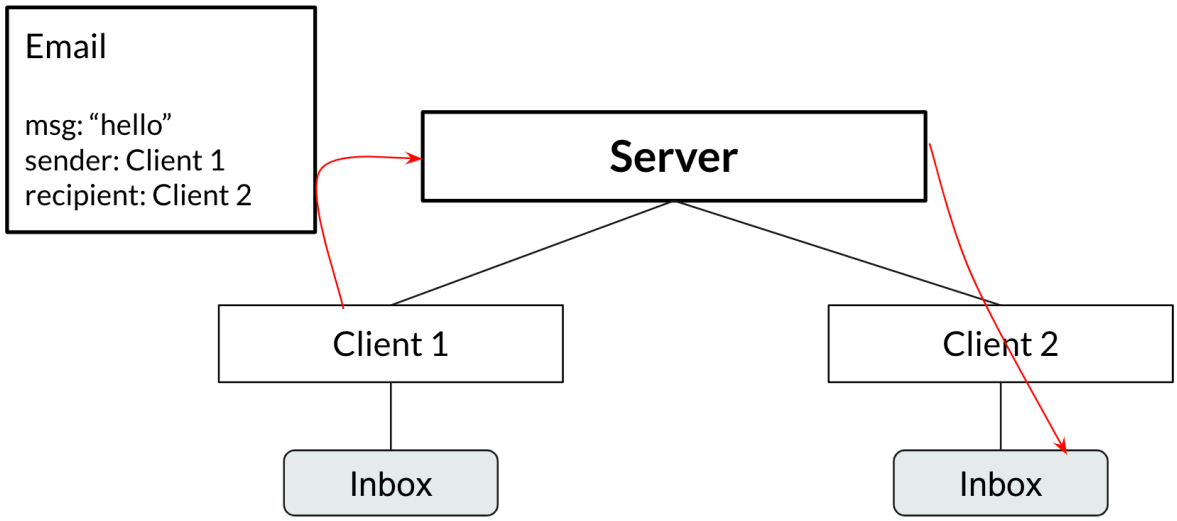
3

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

To solve this problem, we'll split the section into two halves (students on the left and students on the right):

- Everyone will implement the `Email` class together
- The first half (left) will implement the `Server` class
- The other half (right) will implement the `Client` class

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 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
```

```
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)
```

Q3: 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 `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, using the doctests as a reference for the behavior of a `Keyboard`.

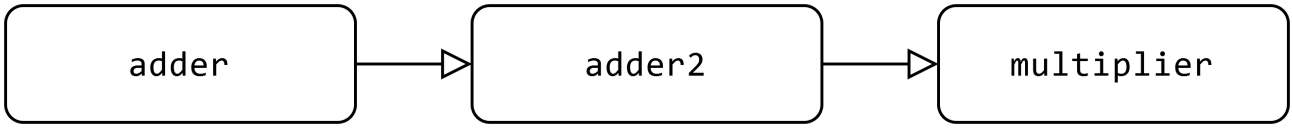
Hint: You can iterate over *args as if it were a list.

```
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
```

Q4: Relay

In a Math Olympiad style relay, team members solve questions while sitting in a line. Each team member's answer is calculated based on the answer from the team member sitting in front of them.

For example, suppose we have three team members, `adder`, `adder2`, and `multiplier`, with `adder` sitting at the very front, `adder2` in the middle, and `multiplier` at the end. When we call the `relay_calculate` method from `multiplier`, we first apply the `adder` operation to the input `x`. Then, the answer from `adder` is passed into the `adder2` operation. Finally, the answer from `adder2` is passed into the `multiplier` operation. The answer from `multiplier` is our final answer.



Relay example

Additionally, each team member has a `relay_history` method, which uses the fact that each team member has an instance variable `history`. `relay_history` returns a list of the answers given by each team member, and this is updated each time we call `relay_calculate`.

Here are some examples of how the `TeamMember` class should behave:

```
>>> adder = TeamMember(lambda x: x + 1) # team member at front
>>> adder2 = TeamMember(lambda x: x + 2, adder) # team member 2
>>> multiplier = TeamMember(lambda x: x * 5, adder2) # team member 3
>>> adder.relay_history() # relay history starts off as empty
[]
>>> adder.relay_calculate(5) # 5 + 1
6
>>> adder2.relay_calculate(5) # (5 + 1) + 2
8
>>> multiplier.relay_calculate(5) # (((5 + 1) + 2) * 5)
40
>>> multiplier.relay_history() # history of answers from the most recent relay multiplier
    participated in
[6, 8, 40]
>>> adder.relay_history()
[6]
>>> multiplier.relay_calculate(4) # (((4 + 1) + 2) * 5)
35
>>> multiplier.relay_history()
[5, 7, 35]
>>> adder.relay_history() # adder participated most recently in multiplier.
   relay_calculate(4), where it gave the answer 5
[5]
>>> adder.relay_calculate(1)
2
>>> adder.relay_history() # adder participated most recently in adder.relay_calculate(1),
    where it gave the answer 2
[2]
>>> multiplier.relay_history() # but the most relay multiplier participated in is still
   multiplier.relay_calculate(4)
[5, 7, 35]
```

Fill in the definitions below to complete the implementation of the TeamMember class!

```
class TeamMember:
    def __init__(self, operation, prev_member=None):
        """
        A TeamMember object is instantiated by taking in an `operation`
        and a TeamMember object `prev_member`, which is the team member
        who "sits in front of" this current team member. A TeamMember also
        tracks a `history` list, which contains the answers given by
        each individual team member.
        """
        self.history = []
        self.operation = operation
        self.prev_member = prev_member
    def relay_calculate(self, x):
        """
        The relay_calculate method takes in a number `x` and performs a
        relay by passing in `x` to the first team member's `operation`.
        Then, that answer is passed to the next member's operation, etc. until
        we get to the current TeamMember, in which case we return the
        final answer, `result`.
        """
        if not self.prev_member:
            result = self.operation(x)
            self.history = [result]
        else:
            prev_result = self.prev_member.relay_calculate(x)
            result = self.operation(prev_result)
            self.history = self.prev_member.history + [result]
        return result
    def relay_history(self):
        """
        Returns a list of the answers given by each team member in the
        most recent relay the current TeamMember has participated in.
        """
        return self.history
```

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:
    def __init__(self, name, owner):
        self.name = name
        self.owner = owner
    @classmethod
    def robo_factory(cls, owner):
        return cls("RoboDog", owner)

    # With other previously defined methods not written out
```

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

Note that with the call above, we don't have to explicitly pass in the `Dog` class as the `cls` argument, since Python implicitly does that for us. We only have to pass in a value for `owner`. When the body of the `Dog.robo_factory` is run, the line `cls("RoboDog", owner)` is equivalent to `Dog("RoboDog", owner)` (since `cls` is bound to the `Dog` class), which creates the new `Dog` instance.

Q5: Own A Cat

Now implement the `cat_creator` method below, which takes in a string `owner` and creates a `Cat` named "[**owner**]'s Cat", where [**owner**] is replaced with the name in the `owner` string.

> **Hint:** To place an apostrophe within a string, the entire string must be surrounded in double-quotes (i.e. `"DeNero's Dog"`)

```
class Cat:
    def __init__(self, name, owner, lives=9):
        self.is_alive = True
        self.name = name
        self.owner = owner
        self.lives = lives

    def talk(self):
        return self.name + ' says meow!'

    @classmethod
    def cat_creator(cls, owner):
        """
        Returns a new instance of a Cat.

        This instance's name is "[owner]'s Cat", with
        [owner] being the name of its owner.

        >>> cat1 = Cat.cat_creator("Bryce")
        >>> isinstance(cat1, Cat)
        True
        >>> cat1.owner
        'Bryce'
        >>> cat1.name
        "Bryce's Cat"
        >>> cat2 = Cat.cat_creator("Tyler")
        >>> cat2.owner
        'Tyler'
        >>> cat2.name
        "Tyler's Cat"
        """
        name = owner + "'s Cat"
        return cls(name, owner)
```