Discussion 6: Object-Oriented Programming, Iterators and Generators

This is an online worksheet that you can work on during discussions and tutorials. Your work is not graded and you do not need to submit anything.

OOP

In a previous lecture, you were introduced to the programming paradigm known as **Object-Oriented Programming (OOP)**. OOP 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. So, a student `Angela` would be an instance of the class `Student`.

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**. An example would be the `num_slip_days_allowed` attribute; the number of slip days that students can use during the semester is not a property of any given student but rather of all of them.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are said to be **methods**. In this case, these actions would be bound 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 attribute**: a data attribute of an object, shared by all instances of a class
- **method**: an action (function) that all instances of a class may perform

Questions

Q1: OOP WWPD - Student

Below we have defined the classes `Professor` and `Student`, implementing some of what was described above. Remember that we pass the `self` argument implicitly to instance methods when using dot-notation.

```
class Student:
    num_students = 0 # this is a class attribute
    def __init__(self, name, staff):
        self.name = name # this is an instance attribute
        self.understanding = 0
        Student.num_students += 1
        print("There are now", Student.num_students, "students")
        staff.add_student(self)

    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
```

What will the following lines output?

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

```
There are now 1 students
```

```
>>> 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
```

```
There are now 2 students
```

```python
>>> x
```

```
Vivian
```

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

```
['Elle']
```

Q2: (Tutorial) Email

We would like to write three different classes (`Server` , `Client` , and `Email`) to simulate a system for sending and receiving email. Fill in the definitions below to finish the implementation!

Important: We suggest that you approach this problem by first filling out the `Email` class, then the `register_client` method of `Server` , the `Client` class, and lastly the `send` method of the `Server` class.

```

    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.
        """
        "*** YOUR CODE HERE ***"
        self.clients[email.recipient_name].receive(email)

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


class Client:
    """Every Client has 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()
```

```python
        >>> 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 = []
        "*** YOUR CODE HERE ***"
        self.server = server
        self.name = name
        # 绑定对应的client
        server.register_client(self, name)

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

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

```

Inheritance

Python classes can implement a useful abstraction technique known as **inheritance**. To illustrate this concept, consider the following `Dog` and `Cat` classes.

```
class Dog():
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")

class Cat():
    def __init__(self, name, owner, lives=9):
        self.is_alive = True
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")
```

Notice that because dogs and cats share a lot of similar qualities, there is a lot of repeated code! 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 redefine **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):
        print(self.name + ' says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class **is a** (no relation to the Python `is` operator) more specific version of the other, e.g. a dog **is a** pet. Because `Dog` inherits from `Pet`, we didn't have to redefine `__init__` or `eat`. However, since we want `Dog` to `talk` in a way that is unique to dogs, we did **override** the `talk` method.

We can use the `super()` function to refer to a class's superclass. For example, calling `super()` with the class definition of `Dog` allows us to refer to the `Pet` class.

Here's an example of an alternate equivalent definition of `Dog` that uses `super()` to explicitly call the `__init__` method of the parent class:

```python
class Dog(Pet):
    def __init__(self, name, owner):
        super().__init__(name, owner)
        # this is equivalent to calling Pet.__init__(self, name, owner)
    def talk(self):
        print(self.name + ' says woof!')
```

Keep in mind that creating the `__init__` function shown above is actually not necessary, because creating a `Dog` instance will automatically call the `_init__` method of `Pet`. Normally when defining an `__init__` method in a subclass, we take some additional action to calling `super().__init__`. For example, we could add a new instance variable like the following:

```python
def __init__(self, name, owner, has_floppy_ears):
    super().__init__(name, owner)
    self.has_floppy_ears = has_floppy_ears
```

Questions

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. We have included the `Pet` class as well for your convenience.

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

```
        print(self.name)

class Cat(Pet):
    def __init__(self, name, owner, lives=9):
        "*** YOUR CODE HERE ***"
        self.name = name
        self.owner = owner
        self.lives = lives

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

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
        "*** YOUR CODE HERE ***"
        print(f"{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 out
        that the cat has no more lives to lose.
        """
        "*** YOUR CODE HERE ***"
        if self.lives == 0:
            print("the cat has no more lives to lose!")
```

```
            self.lives -= 1
            if self.lives == 0:
                self.if_alive = False

```

Q4: (Tutorial) NoisyCat

More cats! Fill in this implemention 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?
        # It's not necessary,because it just differtent in Talk.
        "*** YOUR CODE HERE ***"
    def talk(self):
        """Talks twice as much as a regular cat.

        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
        "*** YOUR CODE HERE ***"
        super().talk()
        super().talk()
```

Iterators

An **iterable** is a data type which contains a collection of values which can be processed one by one sequentially. Some examples of iterables we've seen include lists, tuples, strings, and dictionaries. In general, any object that can be iterated over in a `for` loop can be considered an iterable.

While an iterable contains values that can be iterated over, we need another type of object called an **iterator** to actually retrieve values contained in an iterable. Calling the `iter` function on an iterable will create an iterator over that iterable. Each iterator keeps track of its position within the iterable. Calling the `next` function on an iterator will give the current value in the iterable and move the iterator's position to the next value.

In this way, the relationship between an iterable and an iterator is analogous to the relationship between a book and a bookmark - an iterable contains the data that is being iterated over, and an iterator keeps track of your position within that data.

Once an iterator has returned all the values in an iterable, subsequent calls to `next` on that iterable will result in a `StopIteration` exception. In order to be able to access the values in the iterable a second time, you would have to create a second iterator. Check out the example below:

```
>>> a = [1, 2]
>>> a_iter = iter(a)
>>> next(a_iter)
1
>>> next(a_iter)
2
>>> next(a_iter)
StopIteration
```

Iterables can be used in for loops and as arguments to functions that require a sequence (e.g. `map` and `zip`). For example:

```
>>> for n in range(2):
...     print(n)
...
0
1
```

This works because the for loop implicitly creates an iterator using the `__iter__` method. Python then repeatedly calls `next` repeatedly on the iterator, until it raises `StopIteration`. In other words, the loop above is (basically) equivalent to:

```
range_iterator = iter(range(2))
is_done = False
while not is_done:
    try:
        val = next(range_iterator)
        print(val)
    except StopIteration:
        is_done = True
```

One important application of iterables and iterators is the `for` loop. We've seen how we can use `for` loops to iterate over iterables like lists and dictionaries.

This only works because the `for` loop implicitly creates an iterator using the built-in `iter` function. Python then calls `next` repeatedly on the iterator, until it raises `StopIteration`.

Most iterators are also iterables - that is, calling `iter` on them will return an iterator. This means that we can use them inside `for` loops. However, calling `iter` on most iterators will not create a new iterator - instead, it will simply return the same iterator.

We can also iterate over iterables in a list comprehension or pass in an iterable to the built-in function `list` in order to put the items of an iterable into a list.

In addition to the sequences we've learned, Python has some built-in ways to create iterables and iterators. Here are a few useful ones:

- `range(start, end)` returns an iterable containing numbers from start to end-1. If `start` is not provided, it defaults to 0. Check out the docs (https://docs.python.org/3/library/stdtypes.html#range) for more details.
- `map(f, iterable)` returns a new iterator containing the values resulting from applying `f` to each value in `iterable`. Check out the docs (https://docs.python.org/3/library/functions.html#map) for more details and other uses of `map`, such as passing in multiple iterables.
- `filter(f, iterable)` returns a new iterator containing only the values in `iterable` for which `f(value)` returns `True`. Check out the docs (https://docs.python.org/3/library/functions.html#filter) for more details.

Questions

Q5: Iterators WWPD

What would Python display?

```
>>> s = [[1, 2]]
>>> i = iter(s)
>>> j = iter(next(i))
>>> next(j)
```

1

```
>>> s.append(3)
>>> next(i)
```

3

```
>>> next(j)
```

2

```
>>> next(i)
```

StopIteration

Generators

A **generator function** is a special kind of Python function that uses a **yield** statement instead of a **return** statement to report values. When a generator function is called, it returns a generator object, which is a type of iterator. Below, you can see a function that returns an iterator over the natural numbers.

```
>>> def gen_naturals():
...     current = 0
...     while True:
...         yield current
...         current += 1
>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

The `yield` statement is similar to a `return` statement. However, while a `return` statement closes the current frame after the function exits, a `yield` statement causes the frame to be saved until the next time `next` is called, which allows the generator to automatically keep track of the iteration state.

Once `next` is called again, execution resumes where it last stopped and continues until the next `yield` statement or the end of the function. A generator function can have multiple `yield` statements.

Including a `yield` statement in a function automatically tells Python that this function will create a generator. When we call the function, it returns a generator object instead of executing the body. When the generator's `next` method is called, the body is executed until the next `yield` statement is executed.

When `yield from` is called on an iterator, it will `yield` every value from that iterator. It's similar to doing the following:

```
for x in an_iterator:
    yield x
```

Questions

Q6: Filter-Iter

Implement a generator function called `filter_iter(iterable, fn)` that only yields elements of `iterable` for which `fn` returns True.

```
>>> list(filter_iter(range(5), is_even)) # a list of the values yielded from the call to filter_it
[0, 2, 4]
>>> all_odd = (2*y-1 for y in range(5))
>>> list(filter_iter(all_odd, is_even))
[]
>>> naturals = (n for n in range(1, 100))
>>> s = filter_iter(naturals, is_even)
>>> next(s)
2
>>> next(s)
4
"""
"*** YOUR CODE HERE ***"

for x in iterable:
    if fn(x):
        yield x

```

Q7: (Tutorial) Merge

Write a generator function `merge` that takes in two infinite generators `a` and `b` that are in increasing order without duplicates and returns a generator that has all the elements of both generators, in increasing order, without duplicates.

```
def merge(a, b):
    """
    >>> def sequence(start, step):
    ...     while True:
    ...         yield start
    ...         start += step
    >>> a = sequence(2, 3) # 2, 5, 8, 11, 14, ...
    >>> b = sequence(3, 2) # 3, 5, 7, 9, 11, 13, 15, ...
    >>> result = merge(a, b) # 2, 3, 5, 7, 8, 9, 11, 13, 14, 15
    >>> [next(result) for _ in range(10)]
    [2, 3, 5, 7, 8, 9, 11, 13, 14, 15]
    """
    "*** YOUR CODE HERE ***"
    first_a, first_b = next(a), next(b)
    while True:
```