CS 61A Iterators, Generators, Object-Oriented Programming

Summer 2020 — Discussion 8: July 21, 2020

1 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.

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

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. 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`.

The code to the right shows how we can mimic the behavior of `for` loops using `while` loops.

```
counts = [1, 2, 3]

for i in counts:
   print(i)

# equivalent to following pseudocode
# items = iter(counts)
# while True
#   if next(items) errors
#     exit the loop
#   i = the value that returned
#   print(i)
```

Note that 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.

- **map**(f, iterable) returns a new iterator containing the values resulting from applying f to each value in iterable.
- **filter**(f, iterable) returns a new iterator containing only the values in iterable for which f(value) returns True.

Questions

1.1 What would Python display? If a StopIteration Exception occurs, write StopIteration, and if another error occurs, write Error.

```
>>> lst = [6, 1, "a"]
>>> next(lst)
```

```
>>> lst_iter = iter(lst)
>>> next(lst_iter)
```

```
>>> next(lst_iter)
```

```
>>> next(iter(lst))
```

```
>>> [x for x in lst_iter]
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

2 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.* To the right, 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
```

The example to the right demonstrates how to use generators to output natural numbers.

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Questions

2.1 Write a generator function `generate_subsets` that returns all subsets of the positive integers from 1 to `n`. Each call to this generator's **`next`** method will return a list of subsets of the set `[1, 2, ..., n]`, where `n` is the number of previous calls to `next`.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
```

2.2 Implement `sum_paths_gen`, which takes in a tree `t` and and returns a generator which yields the sum of all the nodes from a path from the root of a tree to a leaf.

You may yield the sums in any order.

```
def sum_paths_gen(t):
    """
    >>> t1 = tree(5)
    >>> next(sum_paths_gen(t1))
    5
    >>> t2 = tree(1, [tree(2, [tree(3), tree(4)]), tree(9)])
    >>> sorted(sum_paths_gen(t2))
    [6, 7, 10]
    """

    if ___________________________:

        yield ____________________

    for __________________________:

        for __________________________:

            yield ____________________
```

3 Object Oriented Programming

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 attributes**. Every student has these attributes, but their values differ from student to student. An attribute that is shared among all instances of `Student` is known as a **class attribute**. An example would be the `students` attribute; the number of students that exist 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 attribute:** a property of an object, specific to an instance
- **class attribute:** a property of an object, shared by all instances of a class
- **method:** an action (function) that all instances of a class may perform

Questions

3.1 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. There are more questions on the next page.

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

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

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

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

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

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

```
>>> x
```

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

3.2 We now want to write three different classes, `Server`, `Client`, and `Email` to simulate email. Fill in the definitions below to finish the implementation! There are more methods to fill out on the next page.

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

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







class Server:
    """Each Server has an 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.
        """






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

```
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).
    """
    def __init__(self, server, name):
        self.inbox = []







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







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

4 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 **superclass** 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* 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.

Questions

4.1 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` to set a cat's `name` and `owner`.

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




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

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas 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.
        """
```

4.2 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`!

```
class ____________________: # Fill me in!

    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?



    def talk(self):
        """Talks twice as much as a regular cat.

        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
```

4.3 (Summer 2013 Final) What would Python display?

```
class A:
    def f(self):
        return 2
    def g(self, obj, x):
        if x == 0:
            return A.f(obj)
        return obj.f() + self.g(self, x - 1)

class B(A):
    def f(self):
        return 4

>>> x, y = A(), B()
>>> x.f()


>>> B.f()


>>> x.g(x, 1)


>>> y.g(x, 2)
```