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.

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.

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.
Iterators, Generators, Object-Oriented Programming

- **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.

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

Error

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

6

>>> next(lst_iter)

1

>>> next(iter(lst))

6

>>> [x for x in lst_iter]

['a']
```
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.

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:

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

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

```python
>>> 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
```
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 \([1, 2, \ldots, n]\), where \(n\) is the number of previous calls to `next`.

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

subsets = [[]]

n = 1

while True:
    yield subsets
    subsets = subsets + [s + [n] for s in subsets]
    n += 1
```

We start with a base list of subsets. To get the next sequence of subsets, we need two things:

- All current subsets will continue to be valid subsets in the future.
- We take all the subsets we currently have, and add the next number. These are also valid subsets.
Implement `sum_paths_gen`, which takes in a tree `t` 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.

```python
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 ______________________

def sum_paths_gen(t):
    if is_leaf(t):
        yield label(t)
    for b in branches(t):
        for s in sum_paths_gen(b):
            yield s + label(t)
```

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

```python
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
```

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

```python
>>> 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

>>> x

'Vivian'

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

['Elle']
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.

```python
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):
        self.msg = msg
        self.sender_name = sender_name
        self.recipient_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."
        ""

        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
```
Iterators, Generators, Object-Oriented Programming

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 = []
        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)
4 Inheritance

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

```python
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:

```python
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.

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

```python
class Cat(Pet):
    def \_init\_(self, name, owner, lives=9):
        Pet._init_(self, name, owner)
        self.lives = lives

    def talk(self):
        """Print out a cat's greeting."
        print(self.name + ' says meow!')

    def lose\_life(self):
        """Decrement 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."
        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 :(")
```

Video walkthrough

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

```python
class ___________________:  # Fill me in!

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

    Cat.talk(self)
    Cat.talk(self)
```

Video walkthrough

---

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*
4.3 (Summer 2013 Final) What would Python display?

```python
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()
2
>>> B.f()
Error (missing self argument)
>>> x.g(x, 1)
4
>>> y.g(x, 2)
8
```

Video walkthrough

---

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