Discussion 7: Iterators and Generators, Object-Oriented Programming

This is an online worksheet that you can work on during discussions. Your work is not graded and you do not need to submit anything. The last section of most worksheets is Exam Prep, which will typically only be taught by your TA if you are in an Exam Prep section. You are of course more than welcome to work on Exam Prep problems on your own.

Iterators

An iterable is any object that can be iterated through, or gone through one element at a time. One construct that we've used to iterate through an iterable is a for loop:

```
for elem in iterable:
    # do something
```

`for` loops work on any object that is *iterable*. We previously described it as working with any sequence -- all sequences are iterable, but there are other objects that are also iterable! We define an **iterable** as an object on which calling the built-in function `iter` function returns an *iterator*. An iterator is another type of object that allows us to iterate through an iterable by keeping track of which element is next in the sequence.

To illustrate this, consider the following block of code, which does the exact same thing as a the for statement above:

```
iterator = iter(iterable)
try:
    while True:
        elem = next(iterator)
        # do something
except StopIteration:
    pass
```

Here's a breakdown of what's happening:

- First, the built-in `iter` function is called on the iterable to create a corresponding *iterator*.
- To get the next element in the sequence, the built-in `next` function is called on this iterator.
- When `next` is called but there are no elements left in the iterator, a `StopIteration` error is raised. In the for loop construct, this exception is caught and execution can continue.

Calling `iter` on an iterable multiple times returns a new iterator each time with distinct states (otherwise, you'd never be able to iterate through a iterable more than once). You can also call `iter` on the iterator itself, which will just return the same iterator without changing its state. However, note that you cannot call `next` directly on an iterable.

Let's see the `iter` and `next` functions in action with an iterable we're already familiar with -- a list.

```
>>> lst = [1, 2, 3, 4]
>>> next(lst)            # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> list_iter
<list_iterator object ...>
>>> next(list_iter)      # Calling next on an iterator
1
>>> next(list_iter)      # Calling next on the same iterator
2
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
3
>>> list_iter2 = iter(lst)
>>> next(list_iter2)     # Second iterator has new state
1
>>> next(list_iter)      # First iterator is unaffected by second iterator
4
>>> next(list_iter)      # No elements left!
StopIteration
>>> lst                  # Original iterable is unaffected
[1, 2, 3, 4]
```

Since you can call `iter` on iterators, this tells us that that they are also iterables! Note that while all iterators are iterables, the converse is not true - that is, not all iterables are iterators. You can use iterators wherever you can use iterables, but note that since iterators keep their state, they're only good to iterate through an iterable once:

```
>>> list_iter = iter([4, 3, 2, 1])
>>> for e in list_iter:
...     print(e)
4
3
2
1
>>> for e in list_iter:
...     print(e)
```

> **Analogy**: An iterable is like a book (one can flip through the pages) and an iterator for a book would be a bookmark (saves the position and can locate the next page). Calling `iter` on a book gives you a new bookmark independent of other bookmarks, but calling `iter` on a bookmark gives you the bookmark itself, without changing its position at all. Calling `next` on the bookmark moves it to the next page, but does not change the pages in the book. Calling `next` on the book wouldn't make sense semantically. We can also have multiple bookmarks, all independent of each other.

Iterable Uses

We know that lists are one type of built-in iterable objects. You may have also encountered the `range(start, end)` function, which creates an iterable of ascending integers from start (inclusive) to end (exclusive).

```
>>> for x in range(2, 6):
...     print(x)
...
2
3
4
5
```

Ranges are useful for many things, including performing some operations for a particular number of iterations or iterating through the indices of a list.

There are also some built-in functions that take in iterables and return useful results:

- `map(f, iterable)` - Creates an iterable over `f(x) for x in iterable`. In some cases, computing a list of the values in this iterable will give us the same result as [`func(x) for x in iterable`]. However, it's important to keep in mind that iterators can potentially have infinite values because they are evaluated lazily, while lists cannot have infinite elements.
- `filter(f, iterable)` - Creates iterator over `x` for each `x` in `iterable` if `f(x)`
- `zip(iterables*)` - Creates an iterable over co-indexed tuples with elements from each of the `iterables`
- `reversed(iterable)` - Creates iterator over all the elements in the input iterable in reverse order
- `list(iterable)` - Creates a list containing all the elements in the input `iterable`
- `tuple(iterable)` - Creates a tuple containing all the elements in the input `iterable`
- `sorted(iterable)` - Creates a sorted list containing all the elements in the input `iterable`
- `reduce(f, iterable)` - Must be imported with `functools`. Apply function of two arguments `f` cumulatively to the items of `iterable`, from left to right, so as to reduce the sequence to a single value.

Questions

Q1: Iterators WWPD

What would Python display?

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

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

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

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

Generators

We can create our own custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**. Generator functions have `yield` statements within the body of the function instead of `return` statements. Calling a generator function will return a generator object and will *not* execute the body of the function.

For example, let's consider the following generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

Calling `countdown(k)` will return a generator object that counts down from `k` to 0. Since generators are iterators, we can call `iter` on the resulting object, which will simply return the same object. Note that the body is not executed at this point; nothing is printed and no numbers are output.

```
>>> c = countdown(5)
>>> c
<generator object countdown ...>
>>> c is iter(c)
True
```

So how is the counting done? Again, since generators are iterators, we call `next` on them to get the next element! The first time `next` is called, execution begins at the first line of the function body and continues until the `yield` statement is reached. The result of evaluating the expression in the `yield` statement is returned. The following interactive session continues from the one above.

```
>>> next(c)
Beginning countdown!
5
```

Unlike functions we've seen before in this course, generator functions can remember their state. On any consecutive calls to `next`, execution picks up from the line after the `yield` statement that was previously executed. Like the first call to `next`, execution will continue until the next `yield` statement is reached. Note that because of this, `Beginning countdown!` doesn't get printed again.

```
>>> next(c)
4
>>> next(c)
3
```

The next 3 calls to `next` will continue to yield consecutive descending integers until 0. On the following call, a `StopIteration` error will be raised because there are no more values to yield (i.e. the end of the function body was reached before hitting a `yield` statement).

```
>>> next(c)
2
>>> next(c)
1
>>> next(c)
0
>>> next(c)
Blastoff!
StopIteration
```

Separate calls to `countdown` will create distinct generator objects with their own state. Usually, generators shouldn't restart. If you'd like to reset the sequence, create another generator object by calling the generator function again.

```
>>> c1, c2 = countdown(5), countdown(5)
>>> c1 is c2
False
>>> next(c1)
5
>>> next(c2)
5
```

Here is a summary of the above:

- A *generator function* has a `yield` statement and returns a *generator object*.
- Calling the `iter` function on a generator object returns the same object without modifying its current state.
- The body of a generator function is not evaluated until `next` is called on a resulting generator object. Calling the `next` function on a generator object computes and returns the next object in its sequence. If the sequence is exhausted, `StopIteration` is raised.
- A generator "remembers" its state for the next `next` call. Therefore,
    - the first `next` call works like this:
        1. Enter the function and run until the line with `yield`.
        2. Return the value in the `yield` statement, but remember the state of the function for future `next` calls.
    - And subsequent `next` calls work like this:
        1. Re-enter the function, start at **the line after the** `yield` **statement that was previously executed**, and run until the next `yield` statement.
        2. Return the value in the `yield` statement, but remember the state of the function for future `next` calls.
- Calling a generator function returns a brand new generator object (like calling `iter` on an iterable object).
- A generator should not restart unless it's defined that way. To start over from the first element in a generator, just call the generator function again.

Another useful tool for generators is the `yield from` statement (introduced in Python 3.3). `yield from` will yield all values from an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2, 3, 4])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
4
>>> next(g)
StopIteration
```

Questions

Q2: Filter-Iter

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

```
def filter_iter(iterable, fn):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even)) # a list of the values yielded from the call to
    [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 ***"


```

Q3: 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 ***"


```

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

Q4: 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)
```

```
>>> 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]
```

Q5: 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.

```python
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):
        "*** YOUR CODE HERE ***"

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.
        """
        "*** YOUR CODE HERE ***"

    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 ***"

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()
    >>> 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 ***"

    def compose(self, msg, recipient_name):
```

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()` within the class definition of `Dog` allows us to access the same object but as if it were an instance of its superclass, in this case `Pet`. This is a little bit of a simplification, and if you're interested you can read more at https://docs.python.org/3/library/functions.html#super.

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

```
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:

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

Questions

Q6: Inheritance Review: That's a Constructor, __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):
        _________________________________________
        self.colors = ['orange', 'black', 'white']
```

```
super.__init__()
```

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

```
Butterfly.__init__()
```

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

Some butterflies like the Owl Butterfly (https://en.wikipedia.org/wiki/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(______________):
    def __init__(self, mimic_animal):
        _______________.__init__()
        ______________ = mimic_animal
```

What expression completes the first blank?

What expression completes the second blank?

What expression completes the third blank?

Q7: 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`.

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

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

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
        "*** YOUR CODE HERE ***"

    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 ***"

```

Q8: NoisyCat

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`! If you'd like to test your code, feel free to copy over your solution to the `Cat` class above.

```
"*** YOUR CODE HERE ***"
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?
        "*** 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 ***"


```

Exam Prep

Q9: Apply That Again

This problem was taken from the Spring 2015 final exam (https://inst.eecs.berkeley.edu//~cs61a/sp15/assets/pdfs/61a-sp15-final.pdf#page=7).

NOTE: We will introduce this problem in section and give you time to work on it then. If you'd like to solve it then, don't look ahead!

Difficulty: ★

Implement `amplify`, a generator function that takes a one-argument function `f` and a starting value `x`. The element at index *k* that it yields (starting at 0) is the result of applying `f` *k* times to `x`. It terminates whenever the next value it would yield is a false value, such as `0`, `""`, `[]`, `False`, etc.

```
def amplify(f, x):
    """Yield the longest sequence x, f(x), f(f(x)), ... that are all true values

    >>> list(amplify(lambda s: s[1:], 'boxes'))
    ['boxes', 'oxes', 'xes', 'es', 's']
    >>> list(amplify(lambda x: x//2-1, 14))
    [14, 6, 2]
    """
    "*** YOUR CODE HERE ***"


```

Q10: Fibonacci Generator

Difficulty: ★★

Construct the generator function `fib_gen`, which when called returns a generator that yields elements of the Fibonacci sequence in order. **Hint:** consider using the `zip` function.

IMPORTANT: As a warm-up, try solving this problem iteratively without using the template. Then try to find a recursive solution using the template (you may add an extra line or two, but no extra structure is necessary). We recommend running your code in a local interpreter, as sometimes the online interpreter has bugs with recursive generator functions.

```
def fib_gen():
    """
    >>> fg = fib_gen()
    >>> for _ in range(7):
    ...     print(next(fg))
    0
    1
    1
    2
    3
    5
    8
    """
    yield from __________________________________
    a = __________________________________________
    ______________________________________________
    for x, y in __________________________________:
        ___________________________________________


```

Q11: Cucumber - Fall 2015 Final Q7

Difficulty: ★★

Cucumber is a card game. Cards are positive integers (no suits). Players are numbered from 0 up to `players` (0, 1, 2, 3 in a 4-player game).

In each `Round`, the players each `play` one card, starting with the `starter` and in ascending order (player 0 follows player 3 in a 4-player game). If the `card` played is as high or higher than the `highest` card played so far, that player takes control. The winner is the last player who took control after every player has played once.

Implement `Round` so that `Game` behaves as described in the doctests below.

> *Hint*: Here is an example of a try-catch with an `AssertionError`
>
> ```
> >> try:
> ...   assert False, 'oh no!'
> ... except AssertionError as e:
> ...   print(e)
> ...
> oh no!
> ```

EDIT: The first two lines in the play function should be:

```
assert ________________________________, f'The round is over, player {who}'
assert _______________________________, f'It is not your turn, player {who}'
```

```
class Game:
    """Play a round and return all winners so far. Cards is a list of pairs.
    Each (who, card) pair in cards indicates who plays and what card they play.
    >>> g = Game()
    >>> g.play_round(3, [(3, 4), (0, 8), (1, 8), (2, 5)])
    >>> g.winners
    [1]
    >>> g.play_round(1, [(3, 5), (1, 4), (2, 5), (0, 8), (3, 7), (0, 6), (1, 7)])
    It is not your turn, player 3
    It is not your turn, player 0
    The round is over, player 1
    >>> g.winners
    [1, 3]
    >>> g.play_round(3, [(3, 7), (2, 5), (0, 9)]) # Round is never completed
    It is not your turn, player 2
    >>> g.winners
    [1, 3]
    """
    def __init__(self):
        self.winners = []

    def play_round(self, starter, cards):
        r = Round(starter)
        for who, card in cards:
            try:
                r.play(who, card)
            except AssertionError as e:
                print(e)
        if r.winner != None:
            self.winners.append(r.winner)

class Round:
    players = 4

    def __init__(self, starter):
```

Q12: Partition Generator

Difficulty: ★★★

Construct the generator function `partition_gen`, which takes in a number `n` and returns an *n-partition iterator*. An *n-partition iterator* yields partitions of `n`, where a partition of `n` is a list of integers whose sum is `n`. The iterator should only return unique partitions; the order of numbers within a partition and the order in which partitions are returned does not matter.

Important: The skeleton code is only a suggestion; feel free to add or remove lines as you see fit.

```
def partition_gen(n):
    """
    >>> for partition in partition_gen(4): # note: order doesn't matter
    ...     print(partition)
    [4]
    [3, 1]
    [2, 2]
    [2, 1, 1]
    [1, 1, 1, 1]
    """
    def yield_helper(j, k):
        if j == 0:
            ____________________________________________
        elif ____________________________________________:
            for small_part in ________________________________:
                yield ____________________________________________
            yield ________________________________________
    yield from yield_helper(n, n)

```