Iterators, Generators, and a Light Intro to OOP Solutions

Computer Science Mentors 61A

March 10 – March 14, 2025

1 Intro to OOP

Object oriented programming is a programming paradigm that organizes relationships within data into **objects** and **classes**. In object oriented programming, each object is an **instance** of some particular class. For example, we can write a `Car` class that acts as a template for cars in general:

```
class Car:
    wheels = 4
    def __init__(self):
        self.gas = 100

    def drive(self):
        self.gas -= 10
        print("Current gas level:", self.gas)

my_car = Car()
```

To represent an individual car, we can then initialize a new instance of `Car` as `my_car` by "calling" the class. Doing so will automatically construct a new object of type `Car`, pass it into the `__init__` method (also called the **constructor**), and then return it. Often, the `__init__` method will initialize an object's **instance attributes**, variables specific to one object instead of all objects in its class. In this case, the `__init__` method initially sets the `gas` instance attribute of each car to 100. It is important to note, however, that you can also manually set object-specific attributes outside of the `__init__` method through variable declaration and methods.

Classes can also have **class attributes**, which are variables shared by all instances of a class. In the above example, `wheels` is shared by all instances of the `Car` class. In other words, all cars have 4 wheels.

Functions within classes are known as methods. **Instance methods** are special functions that act on the instances of a class. We've already seen the `__init__` method. We can call instance methods by using the dot notation we use for instance attributes:

```
>>> my_car.drive()
Current gas level: 90
```

In instance methods, `self` is the instance from which the method was called. We don't have to explicitly pass in `self` because, when we call an instance method from an instance, the instance is automatically

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

passed into the first parameter of the method by Python. That is, `my_car.drive()` is exactly equivalent to the following:

```
>>> Car.drive(my_car)
Current gas level: 80
```

1. What would Python display in the land of Oz?

```
class Elphaba(object):
    magic = 'Defying Gravity'

    def __init__(self, magic):
        self.magic = magic

    def spell(self):
        return type(self).magic + ' and ' + self.magic

class Glinda(Elphaba):
    magic = 'Popular'

    def __init__(self, magic):
        Elphaba.__init__(self, 'good ' + magic)

elphaba = Elphaba('Wicked')
```

(a) `>>> Elphaba.magic`

`'Defying Gravity'`

(b) `>>> elphaba.magic`

`'Wicked'`

(c) `>>> elphaba.spell()`

`'Defying Gravity and Wicked'`

(d) `>>> Elphaba.spell()`

`Error`

(e) `>>> Elphaba.spell(elphaba)`

`'Defying Gravity and Wicked'`

(f)
```
>>> glinda = Glinda('Power')
>>> Glinda.magic
```

`'Popular'`

(g) `>>> glinda.magic`

`'good Power'`

(h) `>>> glinda.spell()`

`'Popular and good Power'`

2. **(H)OOP**
   Given the following code, what will Python output for the following prompts?

```
class Baller:
    all_players = []
    def __init__(self, name, has_ball = False):
       self.name = name
       self.has_ball = has_ball
       Baller.all_players.append(self)

    def pass_ball(self, other_player):
       if self.has_ball:
          self.has_ball = False
          other_player.has_ball = True
          return True
       else:
          return False

class BallHog(Baller):
    def pass_ball(self, other_player):
       return False

>>> richard = Baller('Richard', True)
>>> albert = BallHog('Albert')
>>> len(Baller.all_players)
```

2

```
>>> Baller.name
```

Error

```
>>> len(albert.all_players)
```

2

```
>>> richard.pass_ball()
```

Error

```
>>> richard.pass_ball(albert)
```

True

```
>>> richard.pass_ball(albert)
```

False

```
>>> BallHog.pass_ball(albert, richard)
```

False

```
>>> albert.pass_ball(richard)
```

False

```
>>> albert.pass_ball(albert, richard)
```

Error

2 Iterators & Generators

On a conceptual level, **iterables** are simply objects whose elements can be iterated over. Think of an iterable as anything you can use in a `for` loop, such as ranges, lists, strings, or dictionaries.

On a technical level, iterables are a bit more complicated. An **iterator** is an object on which you can (repeatedly) call `next`, which will return the next element of a sequence. For example, if `it` is an iterator representing the sequence $1, 2, 3$, then we could do the following:

```
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
StopIteration
```

`StopIteration` is an exception that is raised when an iterator has no more elements to produce; it's how we know we've reached the end of an iterator. Iterators that will never produce a `StopIteration` exception are called *infinite*.

Under this regime, an iterable is formally defined as an object that can be turned into an iterator by passing it into the `iter` function. When you iterate over an iterable, Python first uses `iter` to create an iterator from the iterable and then iterates over the iterator. The simple `for` loop syntax abstracts away this fact. f There are a few useful functions that act on iterables that are particularly useful:

- `map(f, it)`: Returns an iterator that produces each element of `it` with the function `f` applied to it.
- `filter(pred, it)`: Returns an iterator that includes only the elements of `it` where the predicate function `pred` returns true.
- `reduce(f, it, init)`: Reduces `it` to a single value by repeatedly calling the two-argument function `f` on the elements of `it`: `reduce(add, [1, 2, 3])` → `6`. Optionally, an initializer may be provided: `reduce(add, [1], 5)` → `6`.

Generators, which are a specific type of iterator, are created using the traditional function definition syntax in Python (`def`) with the body of the function containing one or more `yield` statements. When a generator function (a function that has `yield` in the body) is called, it returns a generator object; the body of the function is not executed. Only when we call `next` on the generator object is the body executed until we hit a `yield` statement. The `yield` statement yields the value and pauses the function. `yield from` is another way to yield values. When we `yield from` another iterable, it yields each element from that other iterable one at a time.

The following generators all represent the sequence $1, 2, 3$:

```
def a():
    yield 1
    yield 2
    yield 3
```

```
def b():
    for x in range(1, 4):
        yield x
```

```
def c():
    yield from b()
```

1. Define a generator function `in_order`, which takes in a tree `t`; assume that `t` and each of its subtrees have either 0 or 2 branches only. Fill in `in_order` to yield the labels of `t` "in order"; that is, for each node, the labels of the left branch should precede the parent label, which should precede the labels of the right branch. You can think of "in order" traversal as reading the tree like you would a book.

```
def in_order(t):
    """
    >>> t = tree(0, [tree(1), tree(2, [tree(3), tree(4)])])
    >>> list(in_order(t))
    [1, 0, 3, 2, 4] # 1 goes first because it's the leftmost node
    """
```

```
def in_order(t):
    if is_leaf(t):
        yield label(t)
    else:
        yield from in_order(branches(t)[0])
        yield label(t)
        yield from in_order(branches(t)[1])
```

2. Given the following code block, what is output by the lines that follow?

```
def foo():
    a = 0
    if a == 0:
        print("Hello")
        yield a
        print("World")

>>> foo()
```

```
<generator object>
```

```
>>> foo_gen = foo()
>>> next(foo_gen)
```

```
Hello
0
```

```
>>> next(foo_gen)
```

```
World
StopIteration
```

```
>>> for i in foo():
...   print(i)
```

```
Hello
0
World
```

```
>>> a = iter(filter(lambda x: x % 2, map(lambda x: x - 1, range(10))))
>>> next(a)
```

```
-1
```

```
>>> reduce(lambda x, y: x + y, a)
```

```
16
```

3. Define `all_sums`, a generator that iterates through all the possible sums of elements from `lst`. (Repeat sums are permitted.)

```
def all_sums(lst):
    """
    >>> list(all_sums([]))
    [0]
    >>> list(all_sums([1, 2]))
    [3, 2, 1, 0]
    >>> list(all_sums([1, 2, 3]))
    [6, 5, 4, 3, 3, 2, 1, 0]
    >>> list(all_sums([1, 2, 7]))
    [10, 9, 8, 7, 3, 2, 1, 0]
    """

    if len(lst) == 0:
        yield 0
    else:
        for sum_rest in all_sums(lst[1:]):
            yield sum_rest + lst[0]
            yield sum_rest
```

4. What Would Python Display?

```
class SkipMachine:
    skip = 1
    def __init__(self, n=2):
        self.skip = n + SkipMachine.skip

    def generate(self):
        current = SkipMachine.skip
        while True:
            yield current
            current += self.skip
            SkipMachine.skip += 1

p = SkipMachine()
twos = p.generate()
SkipMachine.skip += 1
twos2 = p.generate()
threes = SkipMachine(3).generate()
```

(a) **next**(twos)

2

(b) **next**(threes)

2

(c) **next**(twos)

5

(d) **next**(twos)

8

(e) **next**(threes)

7

(f) **next**(twos2)

5

5. (Exam Level: Final Fall-23) Implement `unequal_pairs`, a generator function that yields all **non-empty** pairings of a list `s` in which no pair contains two equal elements.

```
def distinct_pairs(s):
    """
  Yield all non-empty pairings from the list s, where each pair consists
     of two distinct elements.

    >>> sorted(distinct_pairs([4, 2, 2, 4, 4, 1, 1]))  # Four different
       pairings!
    [[(2, 4)], [(4, 1)], [(4, 2)], [(4, 2), (4, 1)]]
    >>> max(unequal_pairs([4, 2, 2, 4, 5, 4, 4, 1, 5, 5, 6]), key=len)  #
       The longest pairing
    [[(4, 2), (4, 5), (4, 1), (5, 6)]]
    """
    if len(s) >= 2:

        yield from _____________________   # (1)

        if _____________________:          # (2)

            pair = (s[0], s[1])

            _____________________          # (3)

            for rest in distinct_pairs(s[3:]):  # Note: [0, 1][3:]
                                                # evaluates to []

                yield _____________________  # (4)
```

```
def distinct_pairs(s):

    if len(s) >= 2:
        yield from  distinct_pairs(s[1:])   # (1)
        if  s[0] != s[1]:                   # (2)
            pair = (s[0], s[1])
             yield [pair]                   # (3)
            for rest in distinct_pairs(s[3:]):  # Note: [0, 1][3:]
                                                # evaluates to []
                yield [pair] + rest]    # (4)
```

6. (Exam Level: Final Spring-25) Taylor Swift is performing at **The Eras Tour**, and she wants to make sure her fans get to hear their favorite songs! However, her setlist is **finite**, so after she runs through all her songs, she **starts over from the beginning** to keep the show going.

   Implement `eras_cycle`, a generator function that takes a **non-empty finite list** `songs` (representing her setlist) and a **positive integer** `k`. It yields `k` songs from `songs`, restarting from the first song each time the end is reached. It is **okay to mutate** `songs`.

```
def eras_cycle(songs, k):
    """
    Yield k songs from a non-empty finite list, restarting from the
       beginning when needed.

    >>> t = eras_cycle(["Cruel Summer", "Lover", "Karma"], 10)
    >>> [next(t), next(t), next(t), next(t)]
    ['Cruel Summer', 'Lover', 'Karma', 'Cruel Summer']
    >>> list(t)  # 6 of the 10 songs remain
    ['Lover', 'Karma', 'Cruel Summer', 'Lover', 'Karma', 'Cruel Summer']
    """
    for song in ________:  # (d)
        yield ________  # (e)
    ________  # (f)
```

```
def eras_cycle(songs, k):
    n = len(songs)
    for i in range(k):  # Loop for k items
        yield songs[i % n]  # Restart when reaching the end
```