ITERATORS, GENERATORS, AND A LIGHT INTRO TO OOP Meta

COMPUTER SCIENCE MENTORS 61A

October 14 – October 18, 2024

Recommended Timeline

- Introduction to OOP:
  - OOP Mini Lecture: 10 minutes
  - Foobar: 8 minutes
  - Hoops: 8 minutes
- Iterators & Generators:
  - Iterators & Generators Mini Lecture: 7 minutes
  - In Order: 5 minutes
  - Foo: 8 minutes
  - All Sums: 10 minutes (a question that combines tree recursion and generators)
  - Skip Machine: 16 minutes (worth to go though, in content's humble opinion, but will take some time – also includes some OOP concepts, so if you didn't go over that, you might want to in case you want to cover this section)
  - Distinct pairs: 5-7 minutes (question adopted from Final Fall-23)

Please remember that these times don't add up to 50 minutes because you're not expected to cover every question—especially since this week's worksheet is a bit longer! Think of it as a problem bank that you can use to shape your section based on what your students need. Before and during section, try to decide which questions would be the most helpful and how you want to budget your time. And feel free to ask your students directly what they'd like to focus on!

As a tip, try planning out how long you'll spend on each topic (like 25 minutes on OOP and 25 minutes on iterators and generators) and do your best to stick to it. That way, you can cover a variety of topics while making sure your students get the most out of it!

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 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
```

Something I like to emphasize with my students is that you can *only* access class and instance attributes using dot notation from an instance. That is, you can never just write `__init__` or `wheels`; you *must* use dot notation to access these attributes. The reason that students are confused by this is that the rules of variable scope in classes are different from those in functions. They often feel like because they are "inside" the class they should be able to access all of these variables without dot notation. I think it's often useful to dispel this notion by emphasizing that the rules are different and that it's essentially the objects and classes that "hold on" to their instance variables. But you should be careful when giving an explanation like this to not confuse your students more.

This overview is not meant to be a first exposure resource for your students, since there are so many ins and outs of OOP. It is likely that you will need to walk through some of the concepts in a more intuitive way than they are presented here.

1. What would Python display?

```
class Foo(object):
    x = 'bam'

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

    def baz(self):
        return type(self).x + self.x

class Bar(Foo):
    x = 'boom'

    def __init__(self, x):
        Foo.__init__(self, 'er' + x)

foo = Foo('boo')
```

(a) `>>> Foo.x`

`'bam'`

(b) `>>> foo.x`

`'boo'`

(c) `>>> foo.baz()`

`'bamboo'`

(d) `>>> Foo.baz()`

`Error`

(e) `>>> Foo.baz(foo)`

`'bamboo'`

(f)
```
>>> bar = Bar('ang')
>>> Bar.x
```

`'boom'`

(g) `>>> bar.x`

`'erang'`

(h) `>>> bar.baz()`

`'boomerang'`

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`.

Technically, `map` and `filter` are not functions but classes, but that is not a distinction we need to make.

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()
```

Something to really emphasize here is the difference between regular function execution and generator function execution. When you call a generator function, you do not begin executing the function body! You only begin executing the function body when **next** is called on the generator object. You then pause when you hit a `yield` statement. I like to tell my students that this is an "abuse of notation": they're coopting function syntax to do something completely different from what a function normally does.

Another thing I like to emphasize is that it is impossible to go "backward" with iterators and generators. After all, we only have a **next**, not a `prev`!

You might find it advantageous to go over some of the examples more in depth.

You may or may not find it useful to present students with an example of how iteration works behind the scene:

```
for x in "Hello":
    print(x)
```

```
it = iter("Hello")
while True:
    try:
        x = next(it)
        print(x)
    except StopIteration:
        pass
```

It's possible this may confuse some students, so be cautious if you attempt to use this or a similar example. In particular, students may be confused by the infinite looping and the **try** and **except** blocks. While error handling isn't something super important in CS 61A, they should be able to use it specifically for dealing with iterators, so it might be a good idea to go over this a bit with your students.

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])
   ```

Teaching Tips

- Trees are meant to be implemented recursively, and this should be emphasized to students.
- What is the base case of the problem? With trees it is typically the leaf, and it works out in this case, where there is only one item to yield.
- Draw out an example of a tree (maybe the doctest). What do we expect the recursive call on each of the branches to return (note that trees either have 0 or 2 branches)?
- After seeing what the recursive calls do, figure out how you combine the label, the left tree recursive call, and the right tree recursive call to get the desired result. Yielding the left recursive call's values, then the label, and then the right recursive call will give the in-order traversal.

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
```

Teaching Tips

- Emphasize heavily the fact that when generators are called, they return a generator object. They do NOT start executing their function body until after `next` is called! (So what does that first line return? A generator object!)
- Remind students that generator objects are independent from one another; if you create a new one from calling the same function again, it starts from the beginning again. Each generator on its own, however, remembers where it stopped after the previous `next` call, so that it can resume the next time you call `next`.
- What happens when there are no more `yield` statements, like in the second `call` on `foo_gen`? The generator has reached the end of all possible values to iterate over, and so it returns a StopIteration error.
- If you stick a generator object inside a for loop (or a list, for that matter), it will go all the way through from start to finish, outputting each yield value after another.
  - Careful, however: 'start' doesn't necessarily mean the very first lines of the function or the first yield call; if you feed in a generator on which you've already called `next`, its "start" will be where it last left off.

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
```

Teaching Tips

- This is a classic tree recursion problem but now in generator form!
- A tree diagram of how the list splits is a good visualization to draw
- Students may have trouble with this because the order in which they're dealing with the recursive case is a bit different than usual.
- If students are struggling to understand the problem, start from the base case of an empty list and work your way up with the sums of a length-1 list, length-2, etc.
- As always, the recursive leap of faith is helpful in understanding what `all_sums(lst[1:])` returns.
- Even though this is a generator problem, we iterate over the call in a for loop so we can treat the function like it returns a list!

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)
```