Intro

This morning I'd like to talk a little bit about iterators and generators. We will start out with similarities and differences, then we will see how to draw them in environment diagrams, and we will finish with some examples. Happy learning!

Iterators

1. Access

First things first: how do we get an iterator object? This is a bit confusing, so we'll start at the top with iterables. By now you are familiar with iterables — things like lists, tuples, strings, etc. We can index them. (Yes, all of them, not just lists!)
But, a lot of the time we just find ourselves writing `lst[0]`, `lst[1]`, `lst[2]`, and so on. We just want to get the elements of the sequence in order. It would be a lot more convenient if there was some object that we could tell, "Okay, give me the next element." Then we would not have to keep track of the indices.
This is where iterators come in. We provide an iterable, and then the iterator object will feed us the elements one by one, in order, just like we asked. To provide an iterable, we just use the `iter` method. In summary:

Iterables		*Iterators*
* Lists		* iter(*list*)
* Tuples	$\xrightarrow{\texttt{iter(\_\_)}}$	* iter(*tuple*)
* Strings		* iter(*string*)
* etc.		* range(*integer*)
access via indexing		access using `next(__)`

Also note that, technically, the word "iterable" is also a blanket term for iterables, iterators, and generators. That's because all of these things are *iterable*, meaning that we are *able* to *iterate* over them, using a `for` loop.

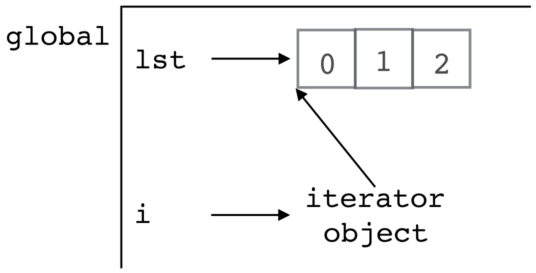
2. Environment Diagrams

Environment diagrams are the single best tool I know of for learning CS. (Yes, better than computers. Actually.) They are already worth a significant fraction of your grade, thanks to WWPD and similar questions which appear consistently on exams. But even more importantly, environment diagrams are how you work your way through confusing bugs or edge cases you don't understand. This is why it's *so vital* that you know how to represent everything in environment diagrams. So, let's talk about drawing iterators in environment diagrams, yay!

(Example 1 on next page.)

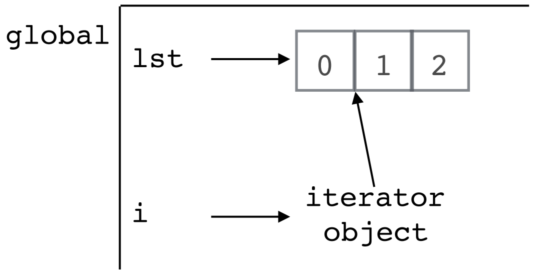
Example 1: Iterators and Mutability
Iterators have one job: to keep track of an index in an iterable.

```
lst = [0, 1, 2]
i = iter(lst)
```



In this environment diagram, `lst` is a list, and `i` is an iterator over that list. In the environment diagram, we simply represent `i` as an `iterator object` holding a pointer to the very beginning of `lst`.

```
lst = [0, 1, 2]
i = iter(lst)
next(i)
```



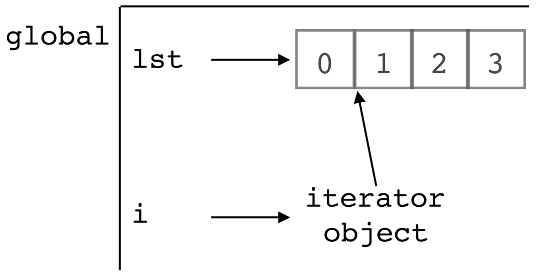
When we call `next(i)`, the number `0` gets returned and we move the pointer in `i` so that it points to the next element in `lst`.

Now that we see how to draw iterators in environment diagrams, let's take a look at a confusing edge case:

```
lst = [0, 1, 2]
i = iter(lst)
next(i)
lst.append(3)
next(i)
next(i)
next(i)
```

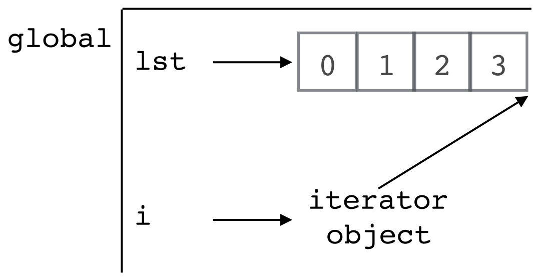
Will the last call to `next` cause an error, or will it return 3? Let's draw the environment diagram. Remember that `append` is a mutable function, so `lst.append(3)` does not make a new `lst`. Rather, it *changes* `lst` so that `3` is at the end. So the first 4 lines should look like this:

```
lst = [0, 1, 2]
i = iter(lst)
next(i)
lst.append(3)
```



Now the `iterator object` has no way to know that 3 was not originally in `lst`. So if we call `next(i)` a few more times, we get 3 and not an error. At the very end, the environment diagram looks like this:

```
lst = [0, 1, 2]
i = iter(lst)
next(i)
lst.append(3)
next(i)
next(i)
next(i)
```



Check that this makes sense to you, before you keep reading.

All right, now let's consider a similar but different piece of code. What will happen — the same thing, or something different?

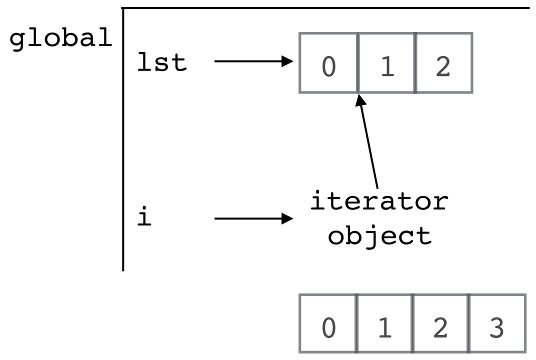
```
lst = [0, 1, 2]
i = iter(lst)
next(i)
lst = lst + [3]
next(i)
next(i)
next(i)
```

The difference here is that we have `lst = lst + [3]` instead of `lst.append(3)`. Sometimes environment diagrams can get tricky, especially when we are assigning or reassigning variables. Even though it may be tedious, *please* use the following process *whenever* you see an equals sign. I have taught it to several students and they all got 100% or very near to 100% on the midterm 2 WWPD.

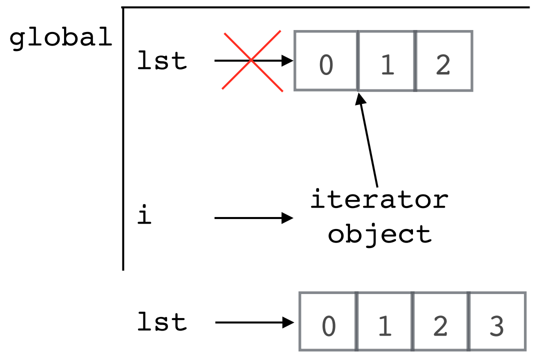
1. Do not look at the left side of the equals sign.
2. Evaluate the right side of the equals sign. Draw the result somewhere with enough space.
3. Now look at the left side of the equals sign, and bind that value to what you just drew.

Continue to the next page to see what it looks like when we apply the process above to the line `lst = lst + [3]`.

Ignore the left hand side of the equals sign for now. Instead let's just evaluate `lst + [3]`. This is drawn below the environment diagram, and nothing is bound to it yet.



This is important. Note that the variable `lst` is *not* bound to the list `[0,1,2]`. Rather, `lst` is just a *pointer* to the list `[0,1,2]`. So, when we reassign `lst`, this only overwrites that *pointer*. It does not overwrite the actual list `[0,1,2]`. The result is the environment diagram to the right.



Note that nothing has changed the list `[0,1,2]`, so the `iterator object` is still the same as before we reassigned the variable `lst`. When we call `next(i)` it will return `1`, and then `2`, and if we call `next(i)` again we will get an error. `next(i)` will never return 3.

Generators

1. Access

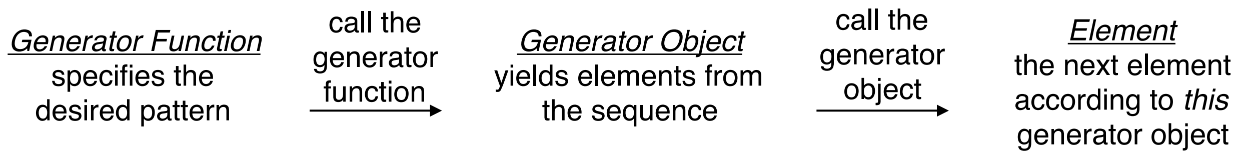
Sometimes we want the elements of a sequence, one by one, but we don't already have that sequence in a list, tuple, etc. That means we can't use an iterator to get the elements we want. Instead, we use something called a generator. Generators only compute a value once we request it.

For example, imagine I have a generator over Fibonacci numbers called `fib`. When I call `fib()` the first time, it yields `0`. When I call `fib()` again, it yields `1`. The *n*th time I call `fib()`, I will get the *n*th Fibonacci number.

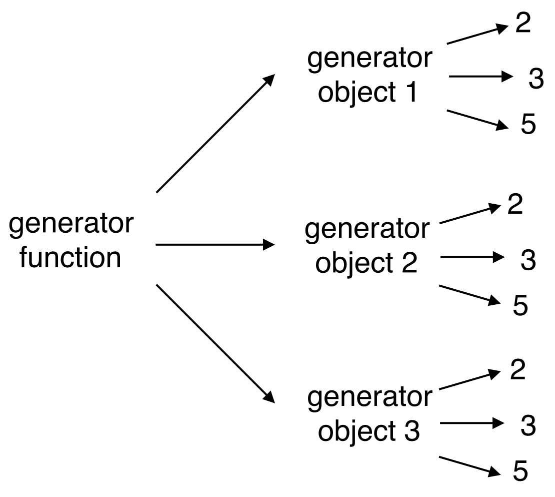
So, how do I get the generator object I want? I have to specify the sequence it should give me. This makes sense, since it couldn't give me the next number in the sequence if it didn't know what the pattern was. To specify a pattern, I have to write a function.

Continue on the next page.

The function that specifies the pattern I want is called a *generator function*. When I call the generator function, it does *not* give me the first number in the pattern. Instead, it gives me a generator object. The generator object is what yields the pattern I want. To summarize:


Generator Function
specifies the desired pattern
call the generator function
Generator Object
yields elements from the sequence
call the generator object
Element
the next element according to *this* generator object

This means that I can call a generator function multiple times, and then I can call each of the resulting generator objects multiple times.



Make sure you understand everything above, before proceeding.

2. Environment Diagrams
All right, so now we know what generators are. How do we draw them? We will explore this question in the following examples.

Example 2: Fibonacci Generator
Let's say I want to get all the Fibonacci numbers, one by one. I can't use an iterator since there are infinite Fibonacci numbers, and iterators are only finite. Instead we can use a generator.

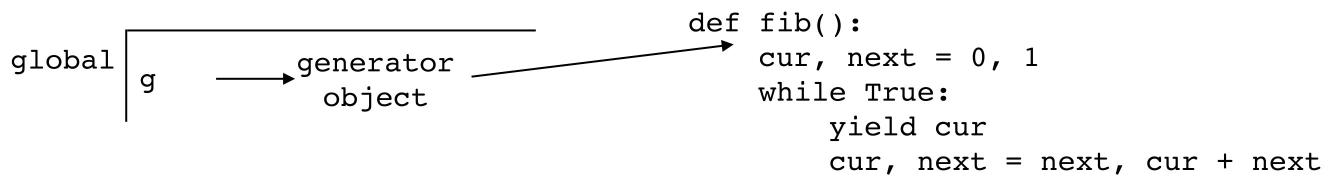
Remember how iterators just have to keep track of an index in an iterable? Well, generators are a really similar idea. The difference is this: instead of keeping track of an index in an iterable, we will be keeping tack of a line of code in a function. See the next page to see what it looks like.

Take a look at this code. First we write a Fibonacci generator function, and then we assign the variable `g` to a particular Fibonacci generator object. Don't worry about how to write this function yet — for now we will focus on being able to understand it.

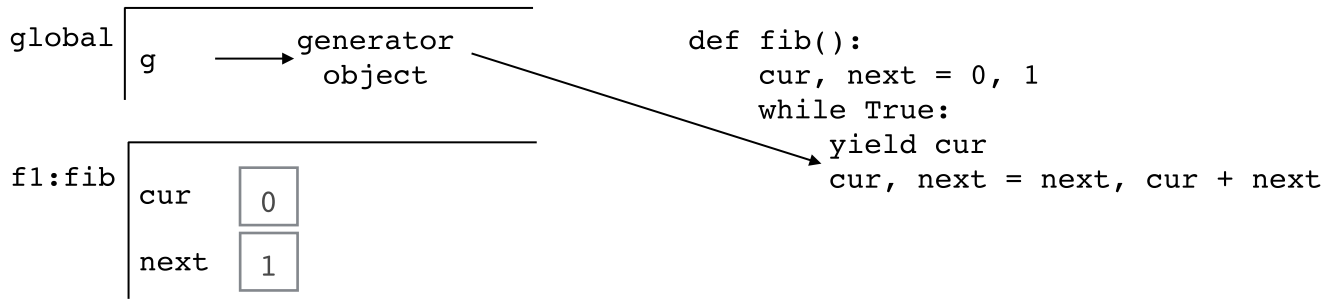
```
def fib():
    cur, next = 0, 1
    while True:
        yield cur
        cur, next = next, cur + next

g = fib()
```

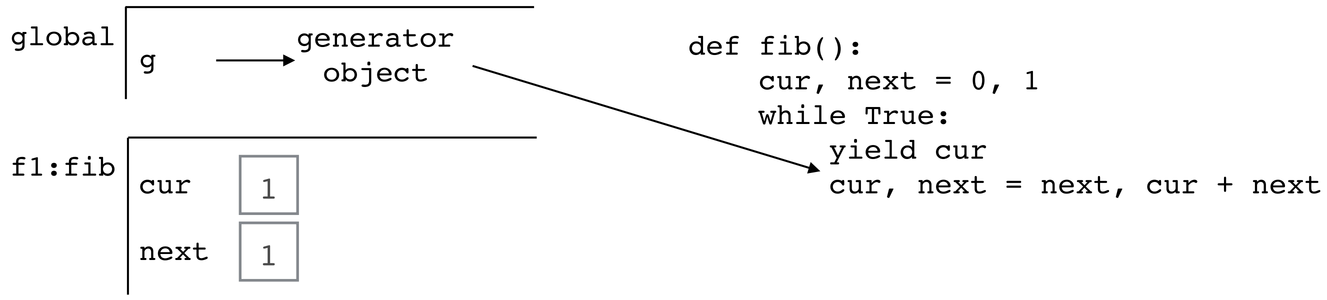
This corresponds to the environment diagram below:



At the beginning, the `generator object` points right under the function signature. When we call `g()`, the function executes until right after the first `yield`. The call to `g` will yield `0` and the environment diagram will look like this:



If we call `g()` again, then we will resume from where we left off in `fib`. The next line to execute is the very bottom one, where `cur` and `next` both get reassigned to `1`. Then we re-enter the `while` loop and `yield cur`. The `generator object`'s pointer leaves off at this `yield`, which is only coincidentally the same one we left off on before. Afterwards, the environment diagram looks like this:



Make sure you understand this diagram before moving to the next page.

Example 3: Generator of Generators
Now that we know how to draw generator objects in environment diagrams, let's talk a little bit about how to write generator functions. The very first thing you want to do is think about *how many* things you should be yielding. Typically we want to `yield` a lot of things — maybe hundreds or even infinite — but we don't want to have to write that many `yield` statements. So let's think about looping. We have seen 3 methods in this class: recursion, `while`, and `for`. The latter two are mostly interchangeable, except for the behavior of `while True`, so we will consider these 3 methods instead:

1. recursion
2. `while True`
3. `for`

Let's consider them one by one. Think about implementing a `fib` generator function recursively. We might try something like this:

```
def fib(n):
    if n <= 0:
        yield n
    yield fib(n-1) + fib(n-2)
```

There is a problem, though. Recall that a *generator function* does *not* return any elements of the sequence we want. Instead it returns a *generator object*, and that yields the elements we want. Look at the recursive calls above. Since `fib` is a generator function, these recursive calls will *not* return numbers. Instead they will return new *generator objects*, and we can't add generator objects! So this will error. In general, this problem makes recursion a bad idea for writing generator functions.

The two remaining options are `while True` and `for`. They are both handy, and suited to different tasks:
* If you want to yield infinitely many elements, then use `while True`.
* If you want to yield finitely many elements, then use `for`.

$\longleftarrow$ This might be something you want on your cheat sheet ;-)

Let's put it to use. In the extra lab questions, there was a function called `remainders_generator(m)`. Instead of yielding numbers or strings like most generator functions we will see in 61A, this generator function yields *generator objects*. (Meta, right?) Go on to the next page to see some doctests.

```
x = remainders_generator(3)
 g1 = x()           g2 = x()             g3 = x()          x()
 g1() –> 0          g2() –> 1            g3() –> 2         StopIteration
 g1() –> 3          g2() –> 4            g3() –> 5
 g1() –> 6          g2() –> 7            g3() –> 8
 g1() –> 9          g2() –> 10           g3() –> 11
 ...                ...                  ...
```

How do we even started on something like this? As always, our very first step should be to decide between using `while True` or using `for`. Refer back to the previous page if you need to. Ask yourself, should `remainders_generator(m)` yield infinitely many things, or finitely many things? Looking at the above doctests, we see it should yield finitely many things — specifically, `m` things. It will yield `m` generator objects. That means we want a `for` loop.

We start out like this:

```
def remainders_generator(m):
    ???
    for r in range(m):
        ???
```

Okay, next step. What are we yielding in the `for` loop? Generator objects. That means we need a generator function, so that we can call it to get the generator objects we need. We will have to define a generator function.

```
def remainders_generator(m):
    def remainders(???):
        ???
    for r in range(m):
        yield remainders(???)
```

This is the tricky part. Does `remainders` need any parameters? Look back at the doctests. `g1`, `g2`, and `g3` all yield different things. That means we can't use the same exact generator function to make them. We need a parameter so that `g1`, `g2`, and `g3` don't all use the same pattern. In the above function, `r` makes a good parameter since we know it is different each time we call `remainders`. The result is something like this:

```
def remainders_generator(m):
    def remainders(r):
        ???
    for r in range(m):
        yield remainders(r)
```

(Example 3 continued on the next page.)

Now we have finished writing `remainders_generator`, and we only have to write the generator function `remainders`. Again recall the first step in writing a generator function: choose between `while True` or `for`. Look at the doctests. `g1`, `g2`, and `g3` are all generator objects returned from a call to `remainders`. They yield infinitely many things, so use `while True`. Now our function looks like this:

```
def remainders_generator(m):
    def remainders(r):
        while True:
            ???
    for r in range(m):
        yield remainders(r)
```

Here's a tip that applies to generator functions, recursion, linked list problems, and pretty much every code-writing question in 61A. (Maybe make a note of it.) The simplest line in the doctest often tells you what your base case is. For reference, here are the simplest doctests for `g1`, `g2`, and `g3` from the previous page:

```
g1() –> 0
g2() –> 1
g3() –> 2
```

It looks like the first call to each generator object just yields the value of `r` that the generator object was created from. Now we have this:

```
def remainders_generator(m):
    def remainders(r):
        while True:
            yield r
            ???
    for r in range(m):
        yield remainders(r)
```

Of course, this is not complete. Now `g1` just yield `0` forever, `g2` will yield `1` forever, and `g3` will yield `2` forever. Since we are yielding `r` each time, think about how we have to modify `r` to get the correct value at every `yield` after the first one. In other words, what's the pattern? Recall from earlier, generator functions are just about writing down patterns. Here, we just increment by `m` to get each successive call:

```
g1: 0, 3, 6, 9
g2: 1, 4, 7, 10
g3: 2, 5, 8, 11
```

The complete function is on the next page.

```
def remainders_generator(m):
    def remainders(r):
        while True:
            yield r
            r += m
    for r in range(m):
        yield remainders(r)
```

Check to see that this makes sense to you, before moving on.

Now let's recap what we learned from that example:
* Check whether you want infinitely many things (`while True`), or finitely many things (`for`).
* Look at the simplest doctest to determine your base case.
* Frequently check your understanding with one of the doctests, as you write code.

Blurring the Line (bonus section)
There is nothing inherently special about iterators and generators in Python. We learn about them because they make a lot of tasks easier, but they are not *necessary* to solve those tasks. It is very important to recognize that the built in functions and data structures are not magic. In fact, you could implement them yourself. (Notably, you would not be able to implement the same syntax used for the built-ins, but this is just a parsing thing.)

Whenever I come across a new concept like generators, I always think about how I would implement it myself. This makes it clear that the built-ins are really nothing more than useful code the designers of Python courteously wrote so you wouldn't have to. For instance, consider the Fibonacci generator we wrote in *Example 2*. We can achieve the same functionality using nothing but a list and a higher order function.

Example 4: Fibonacci "Generator"
This task is more suited toward a lesson on higher order functions, so I will not go over the solution process here. However, I encourage you to give it a go before peeking at the answer. If you aren't able to immediately solve it, don't worry. Just make sure you understand why it works.

```
def fib():
    lst = [0,1]
    def yield_next():
        lst.append(lst[0] + lst[1])
        return lst.pop(0)
    return yield_next
```