CS 61A DISCUSSION 6

ITERATORS AND GENERATORS

THE BIG PICTURE FOR TODAY

ITERATORS ARE OBJECTS THAT SWEEP OVER A COLLECTION OF ITEMS IN A SPECIFIC ORDER. THIS HAPPENS VIA REPEATED APPLICATION OF THE next METHOD, WHICH IS DEFINED ON ALL ITERATORS.

CS 61A Guru

TOPIC #1

ITERATORS

—-

ITERATORS

Iterators step through a collection, item by item, via `next`
Iterables "are" the collection, and provide **iterators** via `iter`

- If you have an iter**able**, you can get an iter**ator** over it by calling `iter`. Then you can observe all of its elements by repeatedly calling `next` on the iter**ator**.
- Be warned: iterators are single-use only! (Once an iterator has gone through all the elements of a finite-length iterable, it's done. Calling `next` on it will give `StopIteration` errors forever.)

—-

IN SUMMARY

`iter(iterable)` –> iterator

`next(iterator)` –> value, or a `StopIteration` error

—-

BUILTIN FUNCTIONALITY

Lots of builtin functions take or produce iterators!

- `map(function, iterable)`
  - Returns an iterator over mapped elements in the iterable.
- `filter(function, iterable)`
  - Returns an iterator over filtered elements from the iterable.
- `zip(*iterables)`
  - Returns an iterator over aggregations of elements from each of the iterables.

—-

EXAMPLE: CREATING AN ITERABLE

To create an iterable, you could write a class that implements `__iter__` as a generator function.

```
class Primes:
    def __init__(self, n):
        self.n = n # upper limit
    def __iter__(self):
        P = [True for i in range(2, self.n + 1)]
        for i in range(2, self.n + 1):
            if not P[i - 2]: continue
            yield i
            if i > sqrt(self.n): continue
            for j in [i*i + k*i for k in range((self.n - i*i)//i + 1)]:
                P[j - 2] = False

list(Primes(30)) → [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

—-

FOR-LOOPS: EXPOSED

Behind the scenes, for-loops really just create iterators using `iter` and then call `next` a bunch of times.

```
for x in <expr>:
    <do stuff>
```

– is equivalent to –

```
iterator = iter(<expr>)
try:
    while True:
        n = next(iterator)
        <do stuff>
except StopIteration:
    pass
```

—-

EXAMPLE: IMPLEMENTING THE LIST CONSTRUCTOR

Let's implement the `list` function. Here we have the function specification:

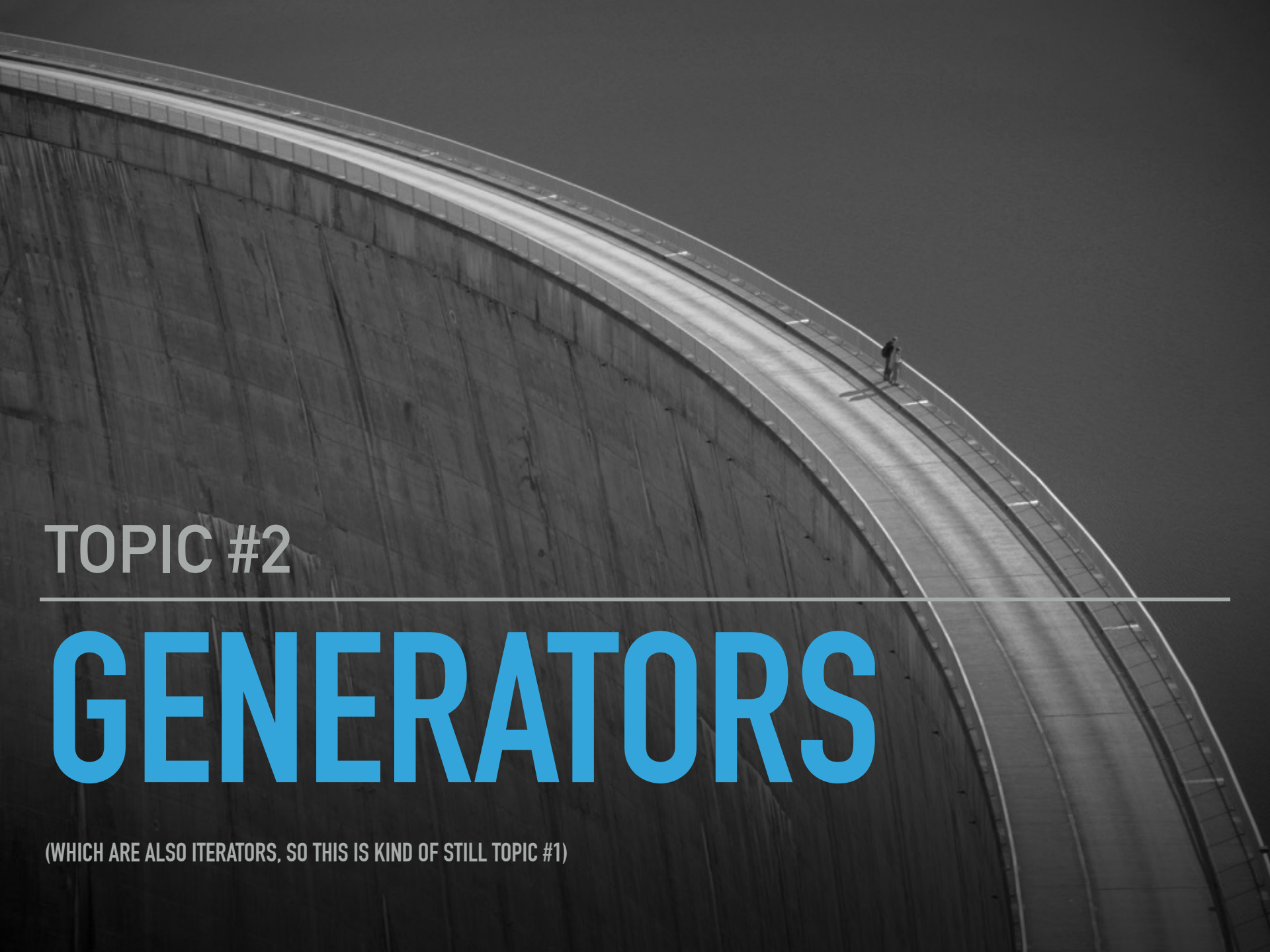
```
def list(iterable):
    """Creates a list.

    >>> list(range(4))
    [0, 1, 2, 3]
    >>> list(iter(range(4)))
    [0, 1, 2, 3]
    """
    # YOUR CODE HERE
```

—-

EXAMPLE: IMPLEMENTING THE LIST CONSTRUCTOR

Let's implement the `list` function. Here we have the function specification:

```
def list(iterable):
    iterator = iter(iterable)
    result = []
    try:
        while True:
            result.append(next(iterator))
    except StopIteration:
        return result
```



TOPIC #2

GENERATORS

(WHICH ARE ALSO ITERATORS, SO THIS IS KIND OF STILL TOPIC #1)

—-

WTF IS A GENERATOR?

Generator functions are functions containing `yield` statements.

- These functions return **generators** when called.

Generators are **iterators** obtained by calling a generator function.

- Every time you call `next` on a generator, it goes through its associated function body until it hits a `yield` - at which point it "yields" the specified value. The state of the function w.r.t. this generator is saved, so whenever `next` is called again on the generator, execution of the function body will continue from where it left off.

(To create a fresh iterator, just call the generator function again.)

EXAMPLE: A GENERATOR OVER THE NATURAL NUMBERS

```
def gen_naturals():
    curr = 0
    while True:
        yield curr
        curr += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

—-

EXERCISE: GENERATING UNORDERED SUBSETS OF A LIST

Write a generator function that goes through 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, ..., `n`], where `n` is the number of times `next` was previously called.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...       print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
    # YOUR-CODE-HERE
```

—-

EXERCISE: GENERATING UNORDERED SUBSETS OF A LIST

```
def generate_subsets():
    # YOUR-CODE-HERE
```

Thought process:

- Uh…
- Okay, well it's a generator function so we're going to have to yield stuff
- Looking at the doctests, it seems as if we always want the positive integers to be in order. We're just splitting them up
  - `[[]]`
  - `[[], [1]]`
  - `[[], [1], [2], [1, 2]]`
- What do you notice? Each successive yield is just everything from before, and also everything from before with the latest value of `n` tacked onto the end

—-

EXERCISE: GENERATING UNORDERED SUBSETS OF A LIST

```
def generate_subsets():
    # YOUR-CODE-HERE
```

In other words,

- `n = 0:`
  `[[]]`
- `n = 1:`
  `[[], [1]]`, aka `[[]` and `[] + [1]]`
- `n = 2:`
  `[[], [1], [2], [1, 2]]`, aka `[[], [1]` and `[] + [2], [1] + [2]]`
- `n = 3:`
  `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`, aka
  `[...` and `[] + [3], [1] + [3], [2] + [3], [1, 2] + [3]]`

EXERCISE: GENERATING UNORDERED SUBSETS OF A LIST

```
def generate_subsets():
    n, subsets = 1, [[]]
    while True:
        yield subsets
        subsets += [s + [n] for s in subsets]
        n += 1
```