Lecture 24: Exceptions and Iterators
Python alerts us of an extraordinary event by throwing an *Exception*

```python
>>> l = list(range(10))
>>> l[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- An *IndexError* is a type of exception
- All exceptions are classes that inherit from the *BaseException* class
We can separate our code’s normal control flow from error handling using try and except:

```
1 l = list(range(10))
2 try:
3     l[10]
4 except IndexError as ie:
5     print("Caught an IndexError: {} -- moving on".format(ie))
6 print(l[0])
```

produces:

Caught an IndexError: list index out of range -- moving on
0
But only *catch what you can handle* by catching the most specific exception class(es)

```python
def int_fraction(num, denom):
    try:
        return num // denom
    except Exception as e:
        print("Can't divide by zero —— returning 0")
    return 0
```

- This code catches and handles a ZeroDivisionError properly
- But other exception classes also inherit from Exception
But only *catch what you can handle* by catching the most specific exception class(es)

```python
def int_fraction(num, denom):
    try:
        return num // denom
    except Exception as e:
        print("Can’t divide by zero —— returning 0")
    return 0
```

- This code catches and handles a ZeroDivisionError properly
- But other exception classes also inherit from Exception

```python
>>> int_fraction(3, 'a'):
Can’t divide by zero —— returning 0
0
```

We mistakenly handle a TypeError as if it were a ZeroDivisionError
To throw an exception, raise the name of a class that is derived from BaseException

```python
def __next__(self):
    if self._has_more_items():
        return self._next_item()
    else:
        raise StopIteration()
```

- Iterators depend on exceptions to indicate they are out of items
Recall that something is *iterable* if it supports the `iter` function—that is the method `__iter__` is defined—and returns an iterator. An *iterator* is something that

- supports the `next` function—that is, the method `__next__` is defined;
- throws a `StopIteration` when the iterator is empty; and
- returns itself under an `iter` call.

Iterators may be defined using *classes* (this lecture) or with *generators* (next lecture).
An Iterator for Squares

class Squares:
    def __init__(self, threshold=None):
        self._state = 1
        self._threshold = threshold

    def _below_threshold(self):
        return self._threshold is None or self._state**2 < self._threshold

    def __iter__(self):
        return self

    def __next__(self):
        if self._below_threshold():
            sq = self._state**2
            self._state += 1
            return sq
        else:
            raise StopIteration()
An Iterator for Even Squares

class EvenSquares(Squares):
    def __next__(self):
        sq = super().__next__()
        while (sq % 2 != 0):
            sq = super().__next__()
        return sq