Lecture 24: Exceptions and Iterators

Exceptions

Python alerts us of an extraordinary event by throwing an *Exception*

```
>>> 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

Exceptions

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
7  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)

```
1  def int_fraction(num, denom):
2      try:
3          return num // denom
4      except Exception as e:
5          print("Can't divide by zero -- returning 0")
6          return 0
7
```

- 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)

```
1  def int_fraction(num, denom):
2      try:
3          return num // denom
4      except Exception as e:
5          print("Can't divide by zero -- returning 0")
6          return 0
7
```

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

```
>>> int_fraction(3, 'a'):
Can't divide by zero -- retuning 0
0
```

We mistakenly handle a `TypeError` as if it were a `ZeroDivisionError`

Exceptions

To throw an exception, `raise` the name of a class that is derived from `BaseException`

```
1  def __next__(self):
2      if self._has_more_items():
3          return self._next_item()
4      else:
5          raise StopIteration()
6
```

- Iterators depend on exeptions to indicate they are out of items

Iterators

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

```
1  class Squares:
2
3      def __init__(self, threshold=None):
4          self._state = 1
5          self._threshold = threshold
6
7      def _below_threshold(self):
8          return self._threshold is None or self._state**2 < self._threshold
9
10     def __iter__(self):
11         return self
12
13     def __next__(self):
14         if self._below_threshold():
15             sq = self._state**2
16             self._state += 1
17             return sq
18         else:
19             raise StopIteration()
```

An Iterator for Even Squares

```
1  class EvenSquares(Squares):
2
3      def __next__(self):
4          sq = super().__next__()
5          while (sq % 2 != 0):
6              sq = super().__next__()
7          return sq
```