Errors and Exceptions

By now you've probably seen your fair share of Python errors. For example:

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

This `IndexError` is a Python *Exception*, which is a way of signaling behavior that is exceptional including errors. Most modern programming languages support exceptions—they allow you to structure your programs so that code to both check and deal with errors is logically distinct from your actual control flow. This makes code much more readable. Here is how exception-handling in Python works. We embed code in a `try`/`except` block where, if an exception is raised, the flow immediately jumps to the `except` clause. If the type of exception matches, then the block is entered. Control flow then returns to the code after the `except` block. For example, running the follow code

```
l = list(range(10))
try:
  l[10]
except IndexError as ie:
  print("Caught an IndexError: {} -- moving on".format(ie))

print(l[0])
```

produces

```
Caught an IndexError: list index out of range -- moving on
0
```

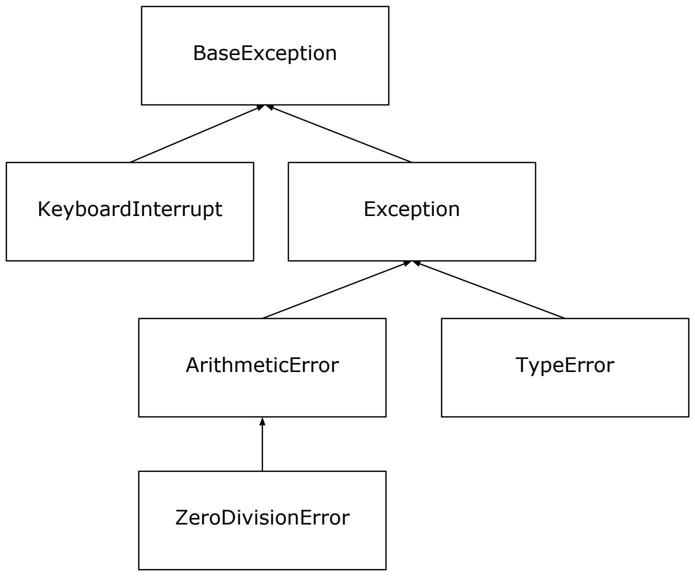
You can use the class hierarchy to catch some types of errors and let others through. For example, above we would only catch exceptions of type `IndexError`—if we executed the following code

```
l = list(range(10))
try:
  l.push(5)
except IndexError as ie:
  print("Caught an IndexError: {} -- moving on".format(ie))
```

then the exception would not be caught

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'list' object has no attribute 'push'
```

Errors are actually a good thing: when it comes to exceptions, the rule of thumb is to *catch only what you can handle*. Consider the portion of the Exception class hierarchy and code example shown below:



```
def int_fraction(num, denom):
    try:
        return num // denom
    except Exception as e:
        print("Can't divide by zero")
        return None
```

The only exceptional behavior that we can reason about here is dividing-by-zero. However, our except clause catches all exception classes that inherit from the general `Exception` class. As a result, we do not properly handle the following code:

```
>>> int_fraction(3, 'a'):
Can't divide by zero
None
```

What we really want is to receive the following error so we know to go back and fix our code:

```
Traceback (most recent call last):
   File ``<stdin>'', line 1, in <module>
TypeError: unsupported operand type(s) for //: 'int' and 'str'
```

We would simply change our except clause to catch the specific `ZeroDivisionError` class.

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

To cause an error, you simply `raise` the name of a class that is derived from `BaseException`. In the next section on iterators, we'll see how the built-in `StopIteration` class is used to signal the end of iteration.

Review of Iterators

A Python object is *iterable* if it supports the `iter` function—that is, it has the magic method `__iter__` defined—and returns an iterator. An *iterator* is something that

- supports the `next` function—that is, the magic 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* or with *generators*. For example, suppose we want an iterator that generates all squares below a certain threshold. We could define the following `squares` class.

```
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()
```

Some specific points:

- We use the optional parameter `threshold=None` to allow for infinite generation. This convention of setting the value to `None` is common in Python.
- The `__iter__` method returns `self`
- The `_below_threshold` method makes use of a *short-circuited* logical `or` operator. Short-circuited means that the expressions are evaluated left-to-right and if the whole expression can be inferred without evaluating any more expressions, then evaluation is complete. In this case, if `self._threshold is None` then the right-hand side is never evaluated, which is good because you can't compare an integer to `None`.
- The `__next__` method raises a `StopIteration` using the `raise` syntax.

Example: Even Squares

Imagine that you wanted to create an iterator that returned squares that were even. One way to do this is to create a new `even_squares` class that *inherits* from `squares`. Without any new methods, the `even_squares` class inherits the behavior of `squares` as is. However, when `next` is called, we only want even squares returned. To do this, we *override* the the `__next__` method so that it calls the next method of its *superclass* until it reaches an even square.

```
class EvenSquares(Squares):

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