3.3 Exceptions

Programmers must be always mindful of possible errors that may arise in their programs. Examples abound: a function may not receive arguments that it is designed to accept, a necessary resource may be missing, or a connection across a network may be lost. When designing a program, one must anticipate the exceptional circumstances that may arise and take appropriate measures to handle them.

There is no single correct approach to handling errors in a program. Programs designed to provide some persistent service like a web server should be robust to errors, logging them for later consideration but continuing to service new requests as long as possible. On the other hand, the Python interpreter handles errors by terminating immediately and printing an error message, so that programmers can address issues as soon as they arise. In any case, programmers must make conscious choices about how their programs should react to exceptional conditions.

Exceptions, the topic of this section, provides a general mechanism for adding error-handling logic to programs. *Raising an exception* is a technique for interrupting the normal flow of execution in a program, signaling that some exceptional circumstance has arisen, and returning directly to an enclosing part of the program that was designated to react to that circumstance. The Python interpreter raises an exception each time it detects an error in an expression or statement. Users can also raise exceptions with `raise` and `assert` statements.

Raising exceptions. An exception is a object instance with a class that inherits, either directly or indirectly, from the `BaseException` class. The `assert` statement introduced in Chapter 1 raises an exception with the class `AssertionError`. In general, any exception instance can be raised with the `raise` statement. The general form of raise statements are described in the Python docs. The most common use of `raise` constructs an exception instance and raises it.

```
>>> raise Exception('An error occurred')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: an error occurred
```

When an exception is raised, no further statements in the current block of code are executed. Unless the exception is *handled* (described below), the interpreter will return directly to the interactive read-eval-print loop, or terminate entirely if Python was started with a file argument. In addition, the interpreter will print a *stack backtrace*, which is a structured block of text that describes the nested set of active function calls in the branch of execution in which the exception was raised. In the example above, the file name `<stdin>` indicates that the exception was raised by the user in an interactive session, rather than from code in a file.

Handling exceptions. An exception can be handled by an enclosing `try` statement. A `try` statement consists of multiple clauses; the first begins with `try` and the rest begin with `except`:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

The `<try suite>` is always executed immediately when the `try` statement is executed. Suites of the `except` clauses are only executed when an exception is raised during the course of executing the `<try suite>`. Each `except` clause specifies the particular class of exception to handle. For instance, if the `<exception class>` is `AssertionError`, then any instance of a class inheriting from `AssertionError` that is raised during the course of executing the `<try suite>` will be handled by the following `<except suite>`. Within the `<except suite>`, the identifier `<name>` is bound to the exception object that was raised, but this binding does not persist beyond the `<except suite>`.

For example, we can handle a `ZeroDivisionError` exception using a `try` statement that binds the name `x` to 0 when the exception is raised.

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0
handling a <class 'ZeroDivisionError'>
>>> x
0
```

A `try` statement will handle exceptions that occur within the body of a function that is applied (either directly or indirectly) within the `<try suite>`. When an exception is raised, control jumps directly to the body of the `<except suite>` of the most recent `try` statement that handles that type of exception.

```
>>> def invert(x):
        result = 1/x  # Raises a ZeroDivisionError if x is 0
        print('Never printed if x is 0')
```

```
        return result

>>> def invert_safe(x):
        try:
            return invert(x)
        except ZeroDivisionError as e:
            return str(e)

>>> invert_safe(2)
Never printed if x is 0
0.5
>>> invert_safe(0)
'division by zero'
```

This example illustrates that the `print` expression in `invert` is never evaluated, and instead control is transferred to the suite of the `except` clause in `invert_safe`. Coercing the `ZeroDivisionError` `e` to a string gives the human-interpretable string returned by `invert_safe`: `'division by zero'`.

3.3.1 Exception Objects

Exception objects themselves can have attributes, such as the error message stated in an `assert` statement and information about where in the course of execution the exception was raised. User-defined exception classes can have additional attributes.

In Chapter 1, we implemented Newton's method to find the zeros of arbitrary functions. The following example defines an exception class that returns the best guess discovered in the course of iterative improvement whenever a `ValueError` occurs. A math domain error (a type of `ValueError`) is raised when `sqrt` is applied to a negative number. This exception is handled by raising an `IterImproveError` that stores the most recent guess from Newton's method as an attribute.

First, we define a new class that inherits from `Exception`.

```
>>> class IterImproveError(Exception):
        def __init__(self, last_guess):
            self.last_guess = last_guess
```

Next, we define a version of `improve`, our generic iterative improvement algorithm. This version handles any `ValueError` by raising an `IterImproveError` that stores the most recent guess. As before, `improve` takes as arguments two functions, each of which takes a single numerical argument. The `update` function returns new guesses, while the `done` function returns a boolean indicating that improvement has converged to a correct value.

```
>>> def improve(update, done, guess=1, max_updates=1000):
        k = 0
        try:
            while not done(guess) and k < max_updates:
                guess = update(guess)
                k = k + 1
            return guess
        except ValueError:
            raise IterImproveError(guess)
```

Finally, we define `find_zero`, which returns the result of `improve` applied to a Newton update function returned by `newton_update`, which is defined in Chapter 1 and requires no changes for this example. This version of `find_zero` handles an `IterImproveError` by returning its last guess.

```
>>> def find_zero(f, guess=1):
        def done(x):
            return f(x) == 0
        try:
            return improve(newton_update(f), done, guess)
        except IterImproveError as e:
            return e.last_guess
```

Consider applying `find_zero` to find the zero of the function $2x^2 + \sqrt{x}$. This function has a zero at 0, but evaluating it on any negative number will raise a `ValueError`. Our Chapter 1 implementation of Newton's Method would raise that error and fail to return any guess of the zero. Our revised implementation returns the last guess found before the error.

```
>>> from math import sqrt
>>> find_zero(lambda x: 2*x*x + sqrt(x))
-0.030211203830201594
```

Although this approximation is still far from the correct answer of 0, some applications would prefer this coarse approximation to a `ValueError`.

Exceptions are another technique that help us as programs to separate the concerns of our program into modular parts. In this example, Python's exception mechanism allowed us to separate the logic for iterative improvement, which appears unchanged in the suite of the `try` clause, from the logic for handling errors, which appears in `except` clauses. We will also find that exceptions are a useful feature when implementing interpreters in Python.

Continue: 3.4 Interpreters for Languages with Combination