**Errors and Exceptions**

The parser repeats the offending line and displays a little `arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the keyword print, since a colon (":") is missing before it. File name and line number are printed so you know where to look in case the input came from a script:

>>> while 1 print 'Hello world'

  File "<stdin>", line 1

while 1 print 'Hello world'

                      ^SyntaxError: invalid syntax

**Exceptions**

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

>>>

>>> 10 \* (1/0)

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

>>> 4 + spam\*3

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

NameError: name 'spam' is not defined

>>> '2' + 2

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: Can't convert 'int' object to str implicitly

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError, and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

**Handling Exceptions**

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signaled by raising the KeyboardInterrupt exception.

>>>

>>> while True:

...    try:

...        x = int(input("Please enter a number: "))

...        break

...    except ValueError:

...        print("Oops!  That was no valid number.  Try again...")

...

The try statement works as follows.

First, the try clause (the statement(s) between the try and except keywords) is executed.

If no exception occurs, the except clause is skipped and execution of the try statement is finished.

If an exception occurs during the execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.

If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

... except (RuntimeError, TypeError, NameError):

...     pass

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

class B(Exception):

    pass

class C(B):

    pass

class D(C):

    pass

for cls in [B, C, D]:

    try:

        raise cls()

    except D:

        print("D")

    except C:

        print("C")

    except B:

        print("B")

Note that if the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

import sys

try:

    f = open('myfile.txt')

    s = f.readline()

    i = int(s.strip())

except OSError as err:

    print("OS error: {0}".format(err))

except ValueError:

    print( "Could not convert data to an integer.")

except:

    print("Unexpected error:", sys.exc\_info()[0])

    raise

The try … except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

for arg in sys.argv[1:]:

    try:

        f = open(arg, 'r')

    except OSError:

        print('cannot open', arg)

    else:

        print(arg, 'has', len(f.readlines()), 'lines')

        f.close()

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try … except statement.

When an exception occurs, it may have an associated value, also known as the exception’s argument. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines \_\_str\_\_() so the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

>>>

>>> try:

...    raise Exception('spam', 'eggs')

... except Exception as inst:

...    print(type(inst))    # the exception instance

...    print(inst.args)     # arguments stored in .args

...    print(inst)          # \_\_str\_\_ allows args to be printed directly,

...                         # but may be overridden in exception subclasses

...    x, y = inst.args     # unpack args

...    print('x =', x)

...    print('y =', y)

...

<class 'Exception'>

('spam', 'eggs')

('spam', 'eggs')

x = spam

y = eggs

If an exception has arguments, they are printed as the last part (‘detail’) of the message for unhandled exceptions.

Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

>>>

>>> def this\_fails():

...    x = 1/0

...

>>> try:

...    this\_fails()

... except ZeroDivisionError as err:

...    print('Handling run-time error:', err)

...

Handling run-time error: division by zero