

Exceptions

Chapter 14

This chapter covers

- Understanding exceptions
- Handling exceptions in Python
- Using the with keyword

General philosophy of errors and exception handling

- SOLUTION 1: DON'T HANDLE THE PROBLEM
 - The simplest way to handle this disk-space problem is to assume that there'll always be adequate disk space for whatever files you write and that you needn't worry about it.
 - Unfortunately, this option seems to be the most commonly used.
- SOLUTION 2: ALL FUNCTIONS RETURN SUCCESS/FAILURE STATUS
 - There are numerous ways to do this, but a typical method is to have
 - each function or procedure return a status value that indicates whether that function or procedure call executed successfully.
- SOLUTION 3: THE EXCEPTION MECHANISM
 - The code checks for errors on each attempted file write and passes an error status message back up to the calling procedure if an error is detected.

A more formal definition of exceptions

- The act of generating an exception is called **raising** or **throwing** an exception.
- The act of responding to an exception is called **catching** an exception, and the code that handles an exception is called **exception-handling** code or just an **exception handler**.

Handling different types of exceptions

- Depending on exactly what event causes an exception, a program may need to take different actions.
- An exception raised when disk space is exhausted needs to be handled quite differently from an exception that's raised if you run out of memory, and both of these exceptions are completely different from an exception that arises when a divide-by-zero error occurs.
- One way to handle these different types of exceptions is to globally record an error message indicating the cause of the exception, and have all exception handlers examine this error message and take appropriate action. In practice, a different method has proved to be much more flexible.

Exceptions in Python

- An exception is an object generated automatically by Python functions with a raise statement.
- After the object is generated, the raise statement, which raises an exception, causes execution of the Python program to proceed in a manner different from what would normally occur.
- Instead of proceeding with the next statement after the raise or whatever generated the exception, the current call chain is searched for a handler that can handle the generated exception.
- If such a handler is found, it's invoked and may access the exception object for more information.
- If no suitable exception handler is found, the program aborts with an error message.

Types of Python exceptions

- The Python exception set is hierarchically structured, as reflected by the indentation in this list of exceptions.
- Each type of exception is a Python class, which inherits from its parent exception type.
- This hierarchy is deliberate: Most exceptions inherit from Exception, and it's strongly recommended that any user-defined exceptions also subclass Exception, not BaseException.

```
BaseException
  SystemExit
  KeyboardInterrupt
  GeneratorExit
  Exception
    StopIteration
    ArithmeticError
      FloatingPointError
      OverflowError
      ZeroDivisionError
    AssertionError
    AttributeError
    BufferError
    EOFError
    ImportError
      ModuleNotFoundError
    LookupError
      IndexError
      KeyError
    MemoryError
    NameError
      UnboundLocalError
    OSError
      BlockingIOError
      ChildProcessError
      ConnectionError
        BrokenPipeError
        ConnectionAbortedError
        ConnectionRefusedError
        ConnectionResetError
      FileExistsError
      FileNotFoundError
      InterruptedError
      IsADirectoryError
      NotADirectoryError
      PermissionError
      ProcessLookupError
```

Raising exceptions

- Error-checking code built into Python detects that the second input line requests an element at a list index that doesn't exist and raises an `IndexError` exception.

```
>>> alist = [1, 2, 3]
>>> element = alist[7]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```


Catching and handling exceptions

- By defining appropriate exception handlers, you can ensure that commonly encountered exceptional circumstances don't cause the program to fail; perhaps they display an error message to the user or do something else, perhaps even fix the problem, but they don't crash the program.

```
try:
    body
except exception_type1 as var1:
    exception_code1
except exception_type2 as var2:
    exception_code2
.
.
.
except:
    default_exception_code
else:
    else_body
finally:
    finally_body
```

Where to use exceptions

- Exceptions are natural choices for handling almost any error condition.
- It's an unfortunate fact that error handling is often added when the rest of the program is largely complete, but exceptions are particularly good at intelligibly managing this sort of after-the-fact error-handling code (or, more optimistically, when you're adding more error handling after the fact).
- Exceptions are also highly useful in circumstances where a large amount of processing may need to be discarded after it becomes obvious that a computational branch in your program has become untenable.

Context managers using the with keyword

- Some situations, such as reading files, follow a predictable pattern with a set beginning and end.
- In the case of reading from a file, quite often the file needs to be open only one time: while data is being read.
- Then the file can be closed.
- In terms of exceptions, you can code this kind of file access like this:

```
try:
    infile = open(filename)
    data = infile.read()
finally:
    infile.close()
```

Context managers using the with keyword

- Python 3 offers a more generic way of handling situations like this: context managers.
- Context managers wrap a block and manage requirements on entry and departure from the block and are marked by the with keyword.
- File objects are context managers, and you can use that capability to read files:

```
with open(filename) as infile:  
    data = infile.read()
```

Lab

Custom Exceptions

Summary

- Python's exception-handling mechanism and exception classes provide a rich system to handle runtime errors in your code.
- By using try, except, else, and finally blocks, and by selecting and even creating the types of exceptions caught, you can have very fine-grained control over how exceptions are handled and ignored.
- Python's philosophy is that errors shouldn't pass silently unless they're explicitly silenced.
- Python exception types are organized in a hierarchy because exceptions, like all objects in Python, are based on classes.