Exceptions

WE'LL COVER THE FOLLOWING

Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python itself will raise them in a lot of different circumstances. You'll see them repeatedly throughout this book.

What is an exception? Usually it's an error, an indication that something went wrong. (Not all exceptions are errors, but never mind that for now.) Some programming languages encourage the use of error return codes, which you *check*. Python encourages the use of exceptions, which you *handle*.

When an error occurs in the Python Shell, it prints out some details about the exception and how it happened, and that's that. This is called an *unhandled* exception. When the exception was raised, there was no code to explicitly notice it and deal with it, so it bubbled its way back up to the top level of the Python Shell, which spits out some debugging information and calls it a day. In the shell, that's no big deal, but if that happened while your actual Python program was running, the entire program would come to a screeching halt if nothing handles the exception. Maybe that's what you want, maybe it isn't.

> *Unlike Java, Python functions don't declare which exceptions they might raise. It's up to you to determine what possible exceptions you need to catch.*

An exception doesn't need to result in a complete program crash, though. Exceptions can be *handled*. Sometimes an exception is really because you have a bug in your code (like accessing a variable that doesn't exist), but

sometimes an exception is something you can anticipate. If you're opening a file, it might not exist. If you're importing a module, it might not be installed.

If you're connecting to a database, it might be unavailable, or you might not have the correct security credentials to access it. If you know a line of code may raise an exception, you should handle the exception using a `try...except` block.

> *Python uses **try...except** blocks to handle exceptions, and the **raise** statement to generate them. Java and c++ use **try...catch** blocks to handle exceptions, and the **throw** statement to generate them.*

The `approximate_size()` function raises exceptions in two different cases: if the given `size` is larger than the function is designed to handle, or if it's less than zero.

```
if size < 0:
    raise ValueError('number must be non-negative')
```

The syntax for raising an exception is simple enough. Use the `raise` statement, followed by the exception name, and an optional human-readable string for debugging purposes. The syntax is reminiscent of calling a function. (In reality, exceptions are implemented as classes, and this `raise` statement is actually creating an instance of the `ValueError` class and passing the string `'number must be non-negative'` to its initialization method. But we're getting ahead of ourselves!)

> *You don't need to handle an exception in the function that raises it. If one function doesn't handle it, the exception is passed to the calling function, then that function's calling function, and so on "up the stack." If the exception is never handled, your program will crash, Python will print a "traceback" to standard error, and that's the end of that. Again, maybe that's what you want; it depends on what your program does.*

Catching Import Errors

One of Python's built-in exceptions is `ImportError`, which is raised when you

try to import a module and fail. This can happen for a variety of reasons, but the simplest case is when the module doesn't exist in your import search path.

You can use this to include optional features in your program. For example, the `chardet` library provides character encoding auto-detection. Perhaps your program wants to use this library `if it exists`, but continue gracefully if the user hasn't installed it. You can do this with a `try..except` block.

```
try:
  import chardet
except ImportError:
  chardet = None
```

Later, you can check for the presence of the `chardet` module with a simple `if` statement:

```
if chardet:
  # do something
else:
  # continue anyway
```

Another common use of the `ImportError` exception is when two modules implement a common `API`, but one is more desirable than the other. (Maybe it's faster, or it uses less memory.) You can try to import one module but fall back to a different module if the first import fails. For example, the XML chapter talks about two modules that implement a common `API`, called the `ElementTree API`. The first, `lxml`, is a third-party module that you need to download and install yourself. The second, `xml.etree.ElementTree`, is slower but is part of the Python 3 standard library.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

By the end of this `try..except` block, you have imported `some` module and named it `etree`. Since both modules implement a common `API`, the rest of your code doesn't need to keep checking which module got imported. And since the module that *did get* imported is always called `etree`, the rest of your code doesn't need to be littered with `if` statements to call differently-named modules.