8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
File "<stdin>", line 1
while True print('Hello world')
^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function "print()", since a colon ("':'") is missing just before it.

The file name ("<stdin>" in our example) and line number are printed so you know where to look in case the input came from a file.

8.2. 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>
10 * (1/0)
~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
4 + spam*3
^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
'2' + 2
~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

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 occurred, 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.

Built-in Exceptions lists the built-in exceptions and their meanings.

8.3. 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 signalled 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 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/except block.

* 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 an error message.

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 matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around --- an *except clause* listing a derived class does not match instances of its base classes). 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.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except clause* may specify a variable after the exception name. The variable is bound to the exception instance which typically has an "args" attribute that stores the arguments. For convenience, builtin exception types define "__str__()" to print all the arguments without explicitly accessing ".args".

```
>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
```

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

The exception's "__str__()" output is printed as the last part ('detail') of the message for unhandled exceptions.

"BaseException" is the common base class of all exceptions. One of its subclasses, "Exception", is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of "Exception" are not typically handled, because they are used to indicate that the program should terminate. They include "SystemExit" which is raised by "sys.exit()" and "KeyboardInterrupt" which is raised when a user wishes to interrupt the program.

"Exception" can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling "Exception" is to print or log the exception and then re-raise it (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:", err)
except ValueError:
print("Could not convert data to an integer.")
except Exception as err:
print(f"Unexpected {err=}, {type(err)=}")
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.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that 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
```

8.4. Raising Exceptions
=======================

The "raise" statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
raise NameError('HiThere')
NameError: HiThere
```

The sole argument to "raise" indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from "BaseException", such as "Exception" or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the "raise" statement allows you to re-raise the exception:

```
>>> try:
... raise NameError('HiThere')
... except NameError:
... print('An exception flew by!')
... raise
```

```
...
An exception flew by!
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
raise NameError('HiThere')
NameError: HiThere
```

8.5. Exception Chaining
======================

If an unhandled exception occurs inside an "except" section, it will have the exception being handled attached to it and included in the error message:

```
>>> try:
... open("database.sqlite")
... except OSError:
... raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
open("database.sqlite")
~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
File "<stdin>", line 4, in <module>
raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the "raise" statement allows an optional "from" clause:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
>>> def func():
... raise ConnectionError
...
>>> try:
... func()
... except ConnectionError as exc:
... raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
func()
~~~~^^
File "<stdin>", line 2, in func
ConnectionError
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
File "<stdin>", line 4, in <module>
raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

It also allows disabling automatic exception chaining using the "from None" idiom:

```
>>> try:
... open('database.sqlite')
... except OSError:
... raise RuntimeError from None
...
Traceback (most recent call last):
File "<stdin>", line 4, in <module>
raise RuntimeError from None
RuntimeError
```

For more information about chaining mechanics, see Built-in Exceptions.

8.6. User-defined Exceptions
============================

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the "Exception" class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in "Error", similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

8.7. Defining Clean-up Actions
==============================

The "try" statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
... raise KeyboardInterrupt
... finally:
... print('Goodbye, world!')
...
```

```
Goodbye, world!
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
raise KeyboardInterrupt
KeyboardInterrupt
```

If a "finally" clause is present, the "finally" clause will execute as the last task before the "try" statement completes. The "finally" clause runs whether or not the "try" statement produces an exception. The following points discuss more complex cases when an exception occurs:

* If an exception occurs during execution of the "try" clause, the exception may be handled by an "except" clause. If the exception is not handled by an "except" clause, the exception is re-raised after the "finally" clause has been executed.

* An exception could occur during execution of an "except" or "else" clause. Again, the exception is re-raised after the "finally" clause has been executed.

* If the "finally" clause executes a "break", "continue" or "return" statement, exceptions are not re-raised. This can be confusing and is therefore discouraged. From version 3.14 the compiler emits a "SyntaxWarning" for it (see **PEP 765**).

* If the "try" statement reaches a "break", "continue" or "return" statement, the "finally" clause will execute just prior to the "break", "continue" or "return" statement's execution.

* If a "finally" clause includes a "return" statement, the returned value will be the one from the "finally" clause's "return" statement, not the value from the "try" clause's "return" statement. This can be confusing and is therefore discouraged. From version 3.14 the compiler emits a "SyntaxWarning" for it (see **PEP 765**).

For example:

```
>>> def bool_return():
... try:
... return True
... finally:
... return False
...
>>> bool_return()
False
```

A more complicated example:

```
>>> def divide(x, y):
... try:
... result = x / y
... except ZeroDivisionError:
... print("division by zero!")
... else:
... print("result is", result)
```

```
... finally:
... print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
divide("2", "1")
~~~~~~^^^^^^^^^^
File "<stdin>", line 3, in divide
result = x / y
~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the "finally" clause is executed in any event. The "TypeError" raised by dividing two strings is not handled by the "except" clause and therefore re-raised after the "finally" clause has been executed.

In real world applications, the "finally" clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

8.8. Predefined Clean-up Actions
================================

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The "with" statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
for line in f:
print(line, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

8.9. Raising and Handling Multiple Unrelated Exceptions
===

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin "ExceptionGroup" wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
... excs = [OSError('error 1'), SystemError('error 2')]
... raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| f()
| ~^^
| File "<stdin>", line 3, in f
| raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+-+---------------- 1 ----------------
| OSError: error 1
+---------------- 2 ----------------
| SystemError: error 2
+------------------------------------
>>> try:
... f()
... except Exception as e:
... print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

By using "except*" instead of "except", we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each "except*" clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be reraised.

```
>>> def f():
... raise ExceptionGroup(
... "group1",
... [
... OSError(1),
... SystemError(2),
... ExceptionGroup(
... "group2",
```

```
... [
... OSError(3),
... RecursionError(4)
... ]
... )
... ]
... )
...
>>> try:
... f()
... except* OSError as e:
... print("There were OSErrors")
... except* SystemError as e:
... print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 2, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
| raise ExceptionGroup(
| ...<12 lines>...
| )
| ExceptionGroup: group1 (1 sub-exception)
+-+---------------- 1 ----------------
| ExceptionGroup: group2 (1 sub-exception)
+-+---------------- 1 ----------------
| RecursionError: 4
+------------------------------------
>>>
```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```
>>> excs = []
... for test in tests:
... try:
... test.run()
... except Exception as e:
... excs.append(e)
...
>>> if excs:
... raise ExceptionGroup("Test Failures", excs)
...
```

8.10. Enriching Exceptions with Notes
=====================================

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method "add_note(note)" that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

```
>>> try:
... raise TypeError('bad type')
... except Exception as e:
... e.add_note('Add some information')
... e.add_note('Add some more information')
... raise
...
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred.

```
>>> def f():
... raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
... try:
... f()
... except Exception as e:
... e.add_note(f'Happened in Iteration {i+1}')
... excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ExceptionGroup: We have some problems (3 sub-exceptions)
+-+---------------- 1 ----------------
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
| raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 1
+---------------- 2 ----------------
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
```

```
| raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 2
+---------------- 3 ----------------
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
| raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 3
+------------------------------------
>>>
```