Errors and Exceptions

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 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 function **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.

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. 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.

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.

Handling 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. Note that a user-generated interruption is signalled by raising the **KeyboardInterrupt** exception.

Syntax:

```
try:
   # write some code
   # that might throw exception
except <ExceptionType>:
   # Exception handler, alert the user
```

Example

```
>>> 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.

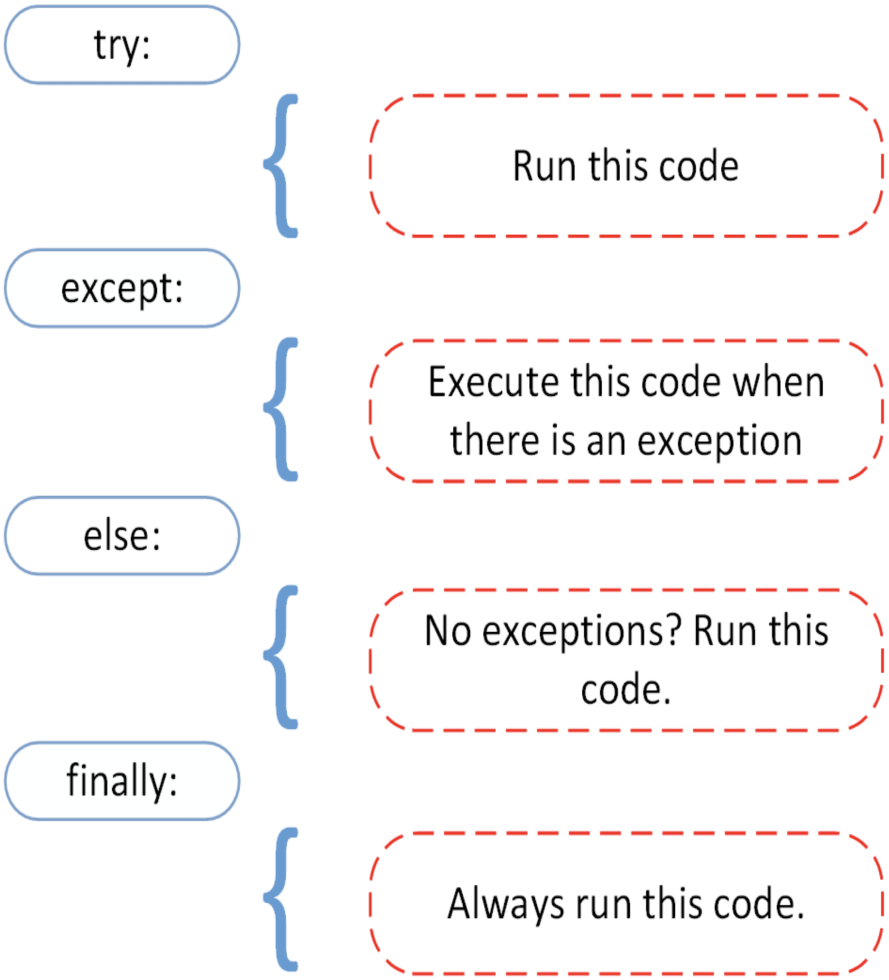
- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try 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 try statement can have more than once except clause, It can also have optional else and/or finally statement.



Note: Statements under the else clause run only when no exception is raised.

Note: Statements in finally block will run every time no matter exception occurs or not.

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>

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 Exception).

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**

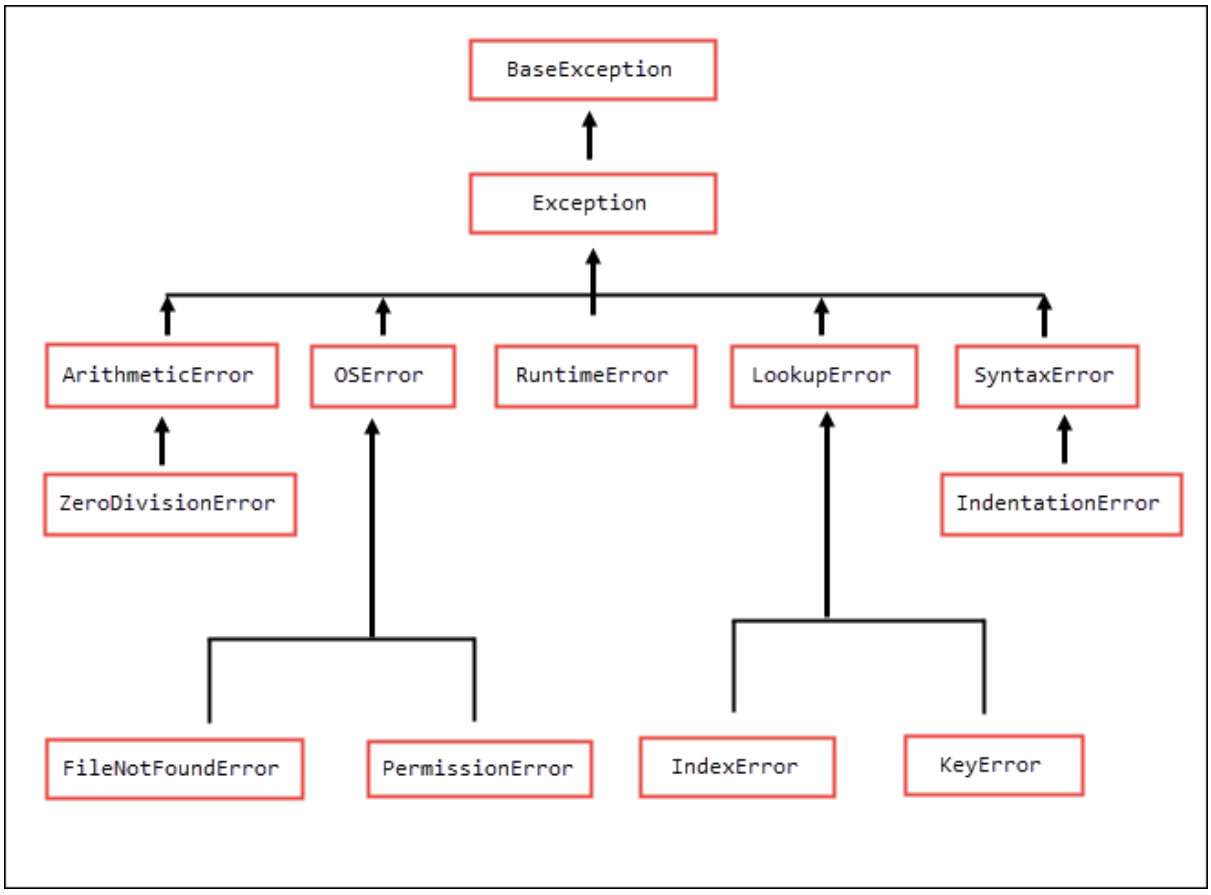
An exception flew by!

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

NameError: HiThere

Exception Classes Hierarchy



User-defined Exceptions

Programs may name their own exceptions by creating a new exception class.

Exceptions should typically be derived from the **Exception** class, either directly or indirectly.

A common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message
```

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

Assignment

1. How many except statements can a try-except block have?

a) zero

b) one

c) more than one

d) more than zero

Answer: d

Explanation: There has to be at least one except statement.

2. When will the else part of try-except-else be executed?

a) always

b) when an exception occurs

c) when no exception occurs

d) when an exception occurs in to except block

Answer: c

Explanation: The else part is executed when no exception occurs.

3. Is the following code valid?

```
try:
    # Do something
except:
    # Do something
finally:
    # Do something
```

a) no, there is no such thing as finally

b) no, finally cannot be used with except

c) no, finally must come before except
d) yes
Answer: b
Explanation: Refer documentation.

4. Is the following code valid?

```
try:
    # Do something
except:
    # Do something
else:
    # Do something
```

a) no, there is no such thing as else
b) no, else cannot be used with except
c) no, else must come before except
d) yes
Answer: d
Explanation: Refer documentation.

5. Can one block of except statements handle multiple exception?
a) yes, like except TypeError, SyntaxError [,…].
b) yes, like except [TypeError, SyntaxError].
c) no
d) none of the mentioned
Answer: a
Explanation: Each type of exception can be specified directly. There is no need to put it in a list.

6. When is the finally block executed?

a) when there is no exception
b) when there is an exception
c) only if some condition that has been specified is satisfied
d) always
Answer: d
Explanation: The finally block is always executed

7. What is the output of the following code?

```
def foo():
    try:
        return 1
    finally:
        return 2
k = foo()
print(k)
```

a) 1
b) 2
c) 3
d) error, there is more than one return statement in a single try-finally block
Answer: b
Explanation: The finally block is executed even there is a return statement in the try block.

8. What is the output of the following code?

```
def foo():
    try:
        print(1)
    finally:
        print(2)
foo()
```

a) 1 2
b) 1
c) 2
d) none of the mentioned
Answer: a
Explanation: No error occurs in the try block so 1 is printed. Then the finally block is executed and 2 is printed.

9. What is the output of the following?

```
try:
    if '1' != 1:
        raise "someError"
    else:
        print("someError has not occured")
except "someError":
    print ("someError has occured")
```

a) someError has occured
b) someError has not occured
c) invalid code
d) none of the mentioned
Answer: c
Explanation: A new exception class must inherit from a BaseException. There is no such inheritance here.

10. What happens when '1' == 1 is executed?
a) we get a True
b) we get a False
c) an TypeError occurs
d) a ValueError occurs

Answer: b
Explanation: It simply evaluates to False and does not raise any exception.

11. Which of the following is not a standard exception in Python?
a) NameError
b) IOError
c) AssignmentError
d) ValueError

Answer: c
Explanation: NameError, IOError and ValueError are standard exceptions in Python whereas Assignment error is not a standard exception in Python.

12. Syntax errors are also known as parsing errors ?

A)

13. An exception is a object ?

A)

14. Which of the following blocks will be executed whether an exception is thrown or not?

a) except
b) else
c) finally
d) assert

15) What is the output of the following program?

```
data = 50

try:

   data = data/0

except ZeroDivisionError:

   print('Cannot divide by 0 ', end = '')

else:

   print('Division successful ', end = '')

 try:

   data = data/5

except:

   print('Inside except block ', end = '')

else:

   print('NNK', end = '')
```

a) Cannot divide by 0 NNK

b) Cannot divide by 0

c) Cannot divide by 0 Inside except block NNK

d) Cannot divide by 0 Inside except block

Ans. (a)

Explanation: The else block of code is executed only when there occurs no exception in try block.