Chapter 24
Error and Exception Handling

24.1 Introduction

This chapter considers exception and error handling and how it is implemented in Python. You are introduced to the object model of exception handling, to the throwing and catching of exceptions, and how to define new exceptions and exception-specific constructs.

24.2 Errors and Exceptions

When something goes wrong in a computer program someone needs to know about it. One way of informing other parts of a program (and potentially those running a program) is by generating an error object and propagating that through the code until either something *handles* the error and sorts thing out or the point at which the program is entered is found.

If the Error propagates out of the program, then the user who ran the program needs to know that something has gone wrong. They are notified of a problem via a short report on the Error that occurred and a stack trace of where that error can be found.

You may have already seen these yourself when writing your own programs. For example, the following screen dump illustrates a programming *error* where someone has tried to concatenate a string and a number together using the '+'

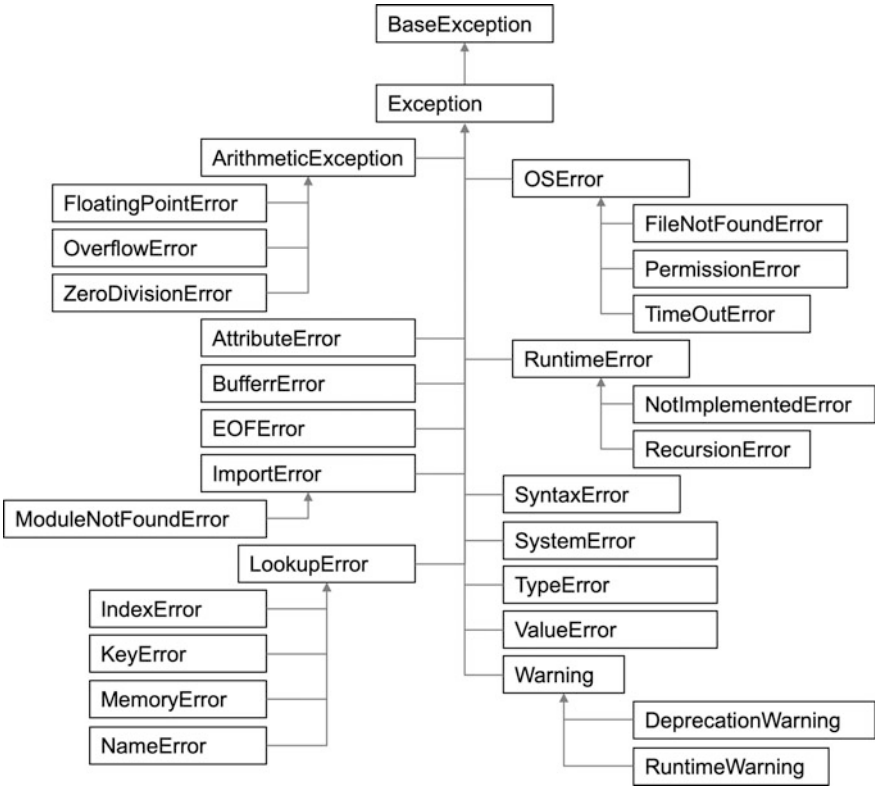
J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_24

operator. This *error* has propagated out of the program and a stack trace of the code that was called is presented (in this case the function print used the `__str__()` method of the class `Person`). Note the line numbers are included which helps with *debugging* the problem.

In Python the terms Error and Exception are used inter-changeably; although from a style point of view Exceptions might be used to represent issues with operations such as arithmetic exceptions and errors might be associated with functional issues such as a file not being found.

24.3 What Is an Exception?

In Python, everything is a type of object, including integers, strings, booleans and indeed Exceptions and Errors. In Python the Exception/Error types are defined in a class hierarchy with the root of this hierarchy being the `BaseException` type. All built-in errors and exceptions eventually extend from the `BaseException` type. It has a subclass `Exception` which is the root of all user defined exceptions (as well as many built-in exceptions). In turn `ArithmeticException` is the base class for all built-in exceptions associated with arithmetic errors.



The above diagram illustrates the class hierarchy for some of the common types of errors and exceptions.

When an exception occurs, this is known as *raising* an exception and when it is passed to code to handle this is known as *throwing* an exception. These are terms that will become more obvious as this chapter progresses.

24.4 What Is Exception Handling?

An exception moves the flow of control from one place to another. In most situations, this is because a problem occurs which cannot be handled locally but that can be handled in another part of the system.

The problem is usually some sort of error (such as dividing by zero), although it can be any problem (for example, identifying that the postcode specified with an address does not match). The purpose of an exception, therefore, is to handle an error condition when it happens at run time.

It is worth considering why you should wish to handle an exception; after all the system does not allow an error to go unnoticed. For example, if we try to divide by zero, then the system generates an error for you. This may mean that the user has

entered an incorrect value, and we do not want users to be presented with a dialog suggesting that they enter the system debugger. We can therefore use exceptions to force the user to correct the mistake and rerun the calculation.

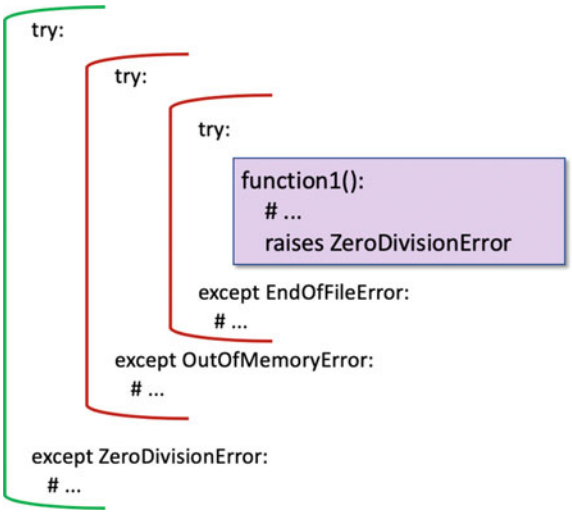
The following table illustrates terminology typically used with exception/error handling in Python.

Exception	An error which is generated at runtime
Raising an exception	Generating a new exception
Throwing an exception	Triggering a generated exception
Handling an exception	Processing code that deals with the error
Handler	The code that deals with the error (referred to as the catch block)
Signal	A particular type of exception (such as *out of bounds* or *divide by zero*)

Different types of error produce different types of exception. For example, if the error is caused by dividing an integer by zero, then the exception is a *arithmetic* exception. The type of exception is identified by objects and can be caught and processed by exception handlers. Each handler can deal with exceptions associated with its class of error or exception (and its subclasses).

An exception is instantiated when it is raised. The system searches back up the execution stack (the set of functions or methods that have been invoked in reverse order) until it finds a handler which can deal with the exception. The associated handler then processes the exception. This may involve performing some remedial action or terminating the current execution in a controlled manner. In some cases, it may be possible to restart executing the code.

As a handler can only deal with an exception of a specified class (or subclass), an exception may pass through a number of handler blocks before it finds one that can process it.



The above figure illustrates a situation in which a divide by zero exception called `ZeroDivisionError` is raised. This exception is passed up the execution stack where it encounters an exception handler defined for an End of File exception. This handler cannot handle the `ZeroDivisionError` and so it is passed further up the execution stack. It then encounters a handler for an out of memory exception. Again, it cannot deal with a `ZeroDivisionError` and the exception is passed further up the execution stack until it finds a handler defined for the `ZeroDivisionError`. This handler then processes the exception.

24.5 Handling an Exception

You can catch an exception by implementing the `try—except` construct. This construct is broken into three parts:

- `try` block. The `try` block indicates the code which is to be monitored for the exceptions listed in the except expressions.
- `except` clause. You can use an optional `except` clause to indicate what to do when certain classes of exception/error occur (e.g. resolve the problem or generate a warning message). There can be any number of except clauses in sequence checking for different types of error/exceptions.
- `else` clause. This is an optional clause which will be run if and only if no exception was thrown in the `try` block. It is useful for code that must be executed if the `try` clause does not raise an exception.
- `finally` clause. The optional `finally` clause runs after the `try` block exits (whether or not this is due to an exception being raised). You can use it to clean up any resources, close files, etc.

This language construct may at first seem confusing, however once you have worked with it for a while you will find it less daunting.

As an example, consider the following function which divides a number by zero; this will raise the `ZeroDivisionError` when it is run for any number:

```
def runcalc(x):
    x / 0
```

If we now call this function, we will get the error trackback in the standard output:

```
runcalc(6)
```

This is shown below:

```
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3.7 /Users/Shared/workspaces/pycharm/python
Traceback (most recent call last):
  File "/Users/Shared/workspaces/pycharm/pythonintro/exceptions/exceptions.py", line 67, in <module>
    main()
  File "/Users/Shared/workspaces/pycharm/pythonintro/exceptions/exceptions.py", line 39, in main
    runcalc(0)
  File "/Users/Shared/workspaces/pycharm/pythonintro/exceptions/exceptions.py", line 2, in runcalc
    x / 0
ZeroDivisionError: division by zero
```

However, we can handle this by wrapping the call to `runcalc` within a `try` statement and providing an `except` clause. The syntax for a `try` statement with an `except` clause is:

```
try:
    <code to monitor>
except <type of exception to monitor for>:
    <code to call if exception is found>
```

A concrete example of this is given below for a `try` statement that will be used to monitor a call to `runcalc`:

```
try:
    runcalc(6)
except ZeroDivisionError:
    print('oops')
```

which now results in the string 'oops' being printed out. This is because when `runcalc` is called the '`/`' operator throws the `ZeroDivisionError` which is passed back to the calling code which has an `except` clause specifying this type of exception. This *catches* the exception and runs the associated code block which in this case prints out the string 'oops'.

In fact, we don't have to be as precise as this; the `except` clause can be given a class of exception to look for and it will match any exception that is of that type or is an instance of a subclass of the exception. We therefore can also write:

```
try:
    runcalc(6)
except Exception:
    print('oops')
```

The `Exception` class is a grandparent of the `ZeroDivisionError` thus any ZeroDivisionError object is also a type of `Exception` and thus the `except` block matches the exception passed. This means that you can write one `except` clause and that clause can handle a whole range of exceptions.

However, if you don't want to have a common block of code handling your exceptions, you can define different behaviours for different types of exception. This is done by having a series of `except` clauses; each monitoring a different type of exception:

```
try:
    runcalc(6)
except ZeroDivisionError:
    print('oops')
except IndexError:
    print('arrgh')
except FileNotFoundError:
    print('huh!')
except Exception:
    print('Duh!')
```

In this case the first `except` monitors for a `ZeroDivisionError` but the other `excepts` monitor for other types of exception. Note that the `except Exception` is the last `except` clause in the list as `ZeroDivisionError`, `IndexError` and `FileNotFoundError` are all eventual subclasses of `Exception` and thus this clause would catch any of these types of exception. As only one `except` clause is allowed to run; if this `except` handler came fist the other `except` handers would never ever be run.

24.5.1 Accessing the Exception Object

It is possible to gain access to the exception object being caught by the `except` clause using the `as` keyword. This follows the exception type being monitored and can be used to bind the exception object to a variable, for example:

```
try:
    runcalc(6)
except ZeroDivisionError as exp:
    print(exp)
    print('oops')
```

Which produces:

```
division by zero
oops
```

If there are multiple `except` clauses, each `except` clause can decide whether to bind the exception object to a variable or not (and each variable can have a different name):

```
try:
    runcalc(6)
except ZeroDivisionError as exp:
    print(exp)
    print('oops')
except IndexError as e:
    print(e)
    print('arrgh')
except FileNotFoundError:
    print('huh!')
except Exception as exception:
    print(exception)
    print('Duh!')
```

In the above example three of the four `except` clauses bind the exception to a variable (each with a different name—although they could all have the same name) but one, the `FileNotFoundError` except clause does not bind the exception to a variable.

24.5.2 Jumping to Exception Handlers

One of the interesting features of Exception handling in Python is that when an Error or an Exception is raised it is immediately *thrown* to the exception handlers (the `except` clauses). Any statements that follow the point at which the exception is raised are not run. This means that a function may be terminated early and further statements in the calling code may not be run.

As an example, consider the following code. This code defines a function `my_function()` that prints out a string, performs a division operation which will cause a `ZeroDivisionError` to be raised if the `y` value is Zero and then it has a further `print` statement. This function is called from within a `try` statement. Notice that there is a `print` statement each side of the call to `my_function()`. There is also a handler for the `ZeroDivisionError`.

```
def my_function(x, y):
    print('my_function in')
    result = x / y
    print('my_function out')
    return result


print('Starting')

try:
    print('Before my_function')
    my_function(6, 2)
    print('After my_function')
except ZeroDivisionError as exp:
    print('oops')

print('Done')
```

When we run this the output is

```
Starting
Before my_function
my_function in
my_function out
After my_function
Done
```

Which is what would probably be expected; we have run every statement with the exception of the `except` clause as the `ZeroDivisionError` was not raised.

If we now change the call to `my_function()` to pass in `6` and `0` we will raise the `ZeroDivisionError`.

```
print('Starting')

try:
    print('Before my_function')
    my_function(6, 0)
    print('After my_function')
except ZeroDivisionError as exp:
    print('oops')

print('Done'
```

Now the output is

```
Starting
Before my_function
my_function in
oops
Done
```

The difference is that the second `print` statement in `my_function()` has not been run; instead after print '`my_function in`' and then raising the error we have jumped straight to the `except` clause and run the `print` statement in the associated block of code.

This is partly why the term *throwing* is used with respect to error and exception handling; because the error or exception is raised in one place and thrown to the point where it is handled, or it is thrown out of the application if no except clause is found to handle the error/exception.

24.5.3 Catch Any Exception

It is also possible to specify an `except` clause that can be used to catch any type of error or exception, for example:

```
try:
    my_function(6, 0)
except IndexError as e:
    print(e)
except:
    print('Something went wrong')
```

This must be the last `except` clause as it omits the exception type and thus acts as a wildcard. It can be used to ensure that you get notified that an error did occur—although you do not know what type of error it actually was; therefore, use this feature with caution.

24.5.4 The Else Clause

The `try` statement also has an optional `else` clause. If this is present, then it must come after all `except` clauses. The `else` clause is executed if and only if no exceptions were raised. If any exception was raised the `else` clause will not be run. An example of the `else` clause is shown below:

```
try:
    my_function(6, 2)
except ZeroDivisionError as e:
    print(e)
else:
    print('Everything worked OK')
```

In this case the output is:

```
my_function in
my_function out
Everything worked OK
```

As you can see the `print` statement in the `else` clause has been executed, however if we change the `my_function()` call to pass in a zero as the second parameter (which will cause the function to raise a `ZeroDivisionError`), then the output is:

```
my_function in
division by zero
```

As you can see the `else` clause was not run but the `except` handler was executed.

24.5.5 The Finally Clause

An optional `finally` clause can also be provided with the `try` statement. This clause is the last clause in the statement and must come after any `except` classes as well as the `else` clause.

It is used for code that you want to run whether an exception occurred or not. For example, in the following code snippet:

```
try:
    my_function(6, 2)
except ZeroDivisionError as e:
    print(e)
else:
    print('Everything worked OK')
finally:
    print('Always runs')
```

The `try` block will run, if no error is raised then the `else` clause will be executed and last of all the `finally` code will run, we will therefore have as output:

```
my_function in
my_function out
Everything worked OK
Always runs
```

If however we pass in 6 and 0 to `my_function()`:

```
try:
    my_function(6, 0)
except ZeroDivisionError as e:
    print(e)
else:
    print('Everything worked OK')
finally:
    print('Always runs')
```

We will now raise an exception in `my_function()` which means that the `try` block will execute, then the `ZeroDivisionError` will be raised, it will be handled by the `except` clause and then the `finally` clause will run. The output is now:

```
my_function in
division by zero
Always runs
```

As you can see in both cases the `finally` clause is executed.

The `finally` clause can be very useful for general housekeeping type activities such as shutting down or closing any resources that your code might be using, even if an error has occurred.

24.6 Raising an Exception

An error or exception is raised using the keyword `raise`. The syntax of this is

```
raise <Exception/Error type to raise>()
```

For example:

```
def function_bang():
    print('function_bang in')
    raise ValueError('Bang!')
    print('function_bang')
```

In the above function the second statement in the function body will create a new instance of the `ValueError` class and then raise it so that it is thrown allowing it to be caught by any exception handlers that have been defined.

We can handle this exception by writing a `try` block with an `except` clause for the `ValueError` class. For example:

```
try:
    function_bang()
except ValueError as ve:
    print(ve)
```

This generates the output

```
function_bang in
Bang!
```

Note that if you just want to raise an exception without providing any constructor arguments, then you can just provide the name of the exception class to the raise keyword:

```
raise ValueError # short hand for raise ValueError()
```

You can also *re-raise* an error or an exception; this can be useful if you merely want to note that an error has occurred and then re throw it so that it can be handled further up in your application:

```
try:
    function_bang()
except ValueError:
    print('oops')
    raise
```

This will re raise the `ValueError` caught by the `except` clause. Note here we did not even bind it to a variable; however, we could have done this if required.

```
try:
    function_bang()
except ValueError as ve:
    print(ve)
    raise
```

24.7 Defining an Custom Exception

You can define your own Errors and Exceptions, which can give you more control over what happens in particular circumstances. To define an exception, you create a subclass of the `Exception` class or one of its subclasses.

For example, to define a `InvalidAgeException`, we can extend the `Exception` class and generate an appropriate message:

```
class InvalidAgeException(Exception):
    """ Valid Ages must be between 0 and 120 """
```

This class can be used to explicitly represent an issue when an age is set on a `Person` which is not within the acceptable age range.

We can use this with the class `Person` that we defined earlier in the book; this version of the `Person` class defined age as a property and attempted to validate that an appropriate age was being set:

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @property
    def age(self):
        """ The docstring for the age property """
        print('In age method')
        return self._age

    @age.setter
    def age(self, value):
        print('In set_age method(', value, ')')
        if isinstance(value, int) & (value > 0 & value < 120):
            self._age = value
        else:
            raise InvalidAgeException(value)

    @property
    def name(self):
        print('In name')
        return self._name

    @name.deleter
    def name(self):
        del self._name

    def __str__(self):
        return 'Person[' + str(self._name) + '] is ' +
self._age
```

Note that the age setter method now throws an `InvalidAgeException`, so if we write:

```
try:
    p = Person('Adam', 21)
    p.age = -1
except InvalidAgeException:
    print('In here')
```

We can capture the fact that an invalid `age` has been specified.

However, in the exception handler we don't know what the invalid age was. We can of course provide this information by including it in the data held by the `InvalidAgeException`.

If we now modify the class definition such that we provide an initialiser to allow parameters to be passed into the new instance of the `InvalidAgeException`:

```
class InvalidAgeException(Exception):
    """ Valid Ages must be between 0 and 120 """

    def __init__(self, value):
        self.value = value

    def __str__(self):
        return 'InvalidAgeException(' + str(self.value) + ')'
```

We have also defined a suitable `__str__()` method to convert the exception into a string for printing purposes.

We do of course need to update the setter to provide the value that has caused the problem:

```
@age.setter
def age(self, value):
    print('In set_age method(', value, ')')
    if isinstance(value,int) & (value > 0 & value < 120):
        self._age = value
    else:
        raise InvalidAgeException(value)
```

We can now write:

```
try:
    p = Person('Adam', 21)
    p.age = -1
except InvalidAgeException as e:
    print(e)
```

Now if the exception is raised a message will be printed out giving the actual value that caused the problem:

```
In set_age method( -1 )
InvalidAgeException(-1)
```

24.8 Chaining Exceptions

One final feature that can be useful when creating your own exceptions is to chain them to a generic underlying exception. This can be useful when a generic exception is raised, for example, by some library or by the Python system itself, and you want to convert it into a more meaningful application exception.

For example, let us say that we want to create an exception to represent a specific issue with the parameters passed to a function divide, but we don't want to use the generic `ZeroDivisionException`, instead we want to use our own `DivideByYWhenZeroException`. This new exception could be defined as

```
class DivideByYWhenZeroException(Exception):
    """ Sample Exception class"""
```

And we can use it in a function divide:

```
def divide(x, y):
    try:
        result = x /y
    except Exception as e:
        raise DivideByYWhenZeroException from e
```

We have used the `raise` and `from` keywords when we are instantiating the `DivideByYWhenZeroException`. This chains our exception to the original exception that indicates the underling problem.

We can now call the divide method as below:

```
def main():
    divide(6, 0)
```

This produces a Traceback as given below:

```
Traceback (most recent call last):
  File
"/Users/Shared/workspaces/pycharm/pythonintro/exceptions/exce
ptions.py", line 43, in divide
    result = x /y
ZeroDivisionError: division by zero
The above exception was the direct cause of the following
exception:
Traceback (most recent call last):
  File
"/Users/Shared/workspaces/pycharm/pythonintro/exceptions/exce
ptions.py", line 136, in <module>
    main()
  File
"/Users/Shared/workspaces/pycharm/pythonintro/exceptions/exce
ptions.py", line 79, in main
    divide(6, 0)
  File
"/Users/Shared/workspaces/pycharm/pythonintro/exceptions/exce
ptions.py", line 45, in divide
    raise DivideByYWhenZeroException from e
__main__.DivideByYWhenZeroException
```

As can be seen you get information about both the (application specific) `DivideByYWhenZeroException` and the original `ZeroDivisionError`—the two are linked together. This can be very useful when defining such application specific exceptions (but where the actual underlying exception must still be understood).

24.9 Online Resources

For more information on Pythons errors and exceptions sets:

- https://docs.python.org/3/library/exceptions.html The Standard Library documentation for built-in exceptions.
- https://docs.python.org/3/tutorial/errors.html The Standard Python documentation tutorial on errors and exceptions.
- https://www.tutorialspoint.com/python/python_exceptions.htm An alternative tutorial on Python exception handling.

24.10 Exercises

This exercise involves adding error handling support to the `CurrentAccount` class.

In the `CurrentAccount` class it should not be possible to withdraw or deposit a negative amount.

Define an exception/error class called `AmountError`. The `AmountError` should take the account involved and an error message as parameters.

Next update the `deposit()` and `withdraw()` methods on the `Account` and `CurrentAccount` class to raise an `AmountError` if the amount supplied is negative.

You should be able to test this using:

```
try:
    acc1.deposit(-1)
except AmountError as e:
    print(e)
```

This should result in the exception 'e' being printed out, for example:

```
AmountError (Cannot deposit negative amounts) on Account[123] -
John, current account = 21.17overdraft limit: -100.0
```

Next modify the class such that if an attempt is made to withdraw money which will take the balance below the over draft limit threshold an Error is raised.

The Error should be a `BalanceError` that you define yourself. The `BalanceError` exception should hold information on the account that generated the error.

Test your code by creating instances of `CurrentAccount` and taking the balance below the overdraft limit.

Write code that will use `try` and `except` blocks to catch the exception you have defined.

You should be able to add the following to your test application:

```
try:
    print('balance:', acc1.balance)
    acc1.withdraw(300.00)
    print('balance:', acc1.balance)
except BalanceError as e:
    print('Handling Exception')
    print(e)
```