Chapter 5

Exceptions

Exceptions are execution errors or situations where the program cannot obtain a valid or expected value. In most cases, they are indicating that something is going wrong with the execution of the program. For example, this situation happens when the input data type does not match with the expected by our program, or any other runtime errors. These conditions produce the unexpected termination of the execution. Python represents exceptions as objects.

5.1 Exception Types

In Python, all exceptions inherit from the `BaseException` class. Below, we see examples of the most common Python exceptions.

```
1  # 0.py
2
3  # SyntaxError exception: print is valid Python2.x, but incorrect in Python3.x
4  print 'Hello World'
```

```
  File "0.py", line 4
    print 'Hello World'
                      ^
SyntaxError: Missing parentheses in call to 'print'
```

```
1  # 1.py
2
3  # In this Python version print is a function
4  # and requires params inside the brackets
```

```
5 print('Hello World')
```

```
Hello World
```

```
1 # 2.py
2
3 # NameError exception: for data input in Python2.x, but incorrect in
4 # Python3.x
5 a = raw_input('Enter a number: ')
6 print(a)
```

```
Traceback (most recent call last):
  File "2.py", line 4, in <module>
    a = raw_input('Enter a number: ')
NameError: name 'raw_input' is not defined
```

```
1 # 3.py
2
3 # ZeroDivisionError exception: division by zero
4 x = 5.0 / 0
```

```
Traceback (most recent call last):
  File "4.py", line 5, in <module>
    x = 5.0 / 0
ZeroDivisionError: float division by zero
```

```
1 # 4.py
2
3 # IndexError exception: index out of range. A typical error that ocurrs
4 # when we try to access to an element of a list with an index that exceeds
5 # its size.
6 # Lists in Python have indexes from 0 to len(list_)-1
7
8 age = [36, 23, 12]
9 print(age[3])
```

```
Traceback (most recent call last):
  File "5.py", line 9, in <module>
```

```
    print(age[3])
IndexError: list index out of range
```

```
# 5.py

# TypeError exception: erroneus data type handling.
# A typical example is trying to concatenate a list
# with a variable that is not a list

age = [36, 23, 12]
print(age + 2)
```

```
Traceback (most recent call last):
  File "6.py", line 8, in <module>
    print(age + 2)
TypeError: can only concatenate list (not "int") to list
```

```
# 6.py

# Correct data type handling.
# To concatenate a list to another object, the latter has to be a list too

age = [36, 23, 12]
print(age + [2])
```

```
[36, 23, 12, 2]
```

```
# 7.py

# AttributeError exception: incorrect use of methods of a class or data type.
# In this example the class Car has only defined the method move, but the
# program tries execute the method stop() that doesn't exist


class Car:
    def __init__(self, doors=4):
        self.doors = doors
```

```
11
12      def mover(self):
13          print('avanzando')
14
15  chevi = Car()
16  chevi.stop()
```

```
Traceback (most recent call last):
  File "8.py", line 16, in <module>
    chevi.stop()
AttributeError: 'Car' object has no attribute 'stop'
```

```
1  # 8.py
2
3  # KeyError exception: incorrect use of key in dictionaries.
4  # In this example the program ask for an item associated with a key that
5  # doesn't appears in the dictionary
6
7  book = {'author': 'Bob Doe', 'pages': 'a lot'}
8  print(book['editorial'])
```

```
Traceback (most recent call last):
  File "9.py", line 8, in <module>
    print(book['editorial'])
KeyError: 'editorial'
```

5.2 Raising exceptions

We trigger exceptions in a program, or within a class or function using the statement `raise`. We can also add optionally a descriptive message to be shown when the exception is raised:

```
1  # 9.py
2
3
4  class Operations:
5
6      @staticmethod
```

```
7       def divide(num, den):
8           if den == 0:
9               # Here we generate the exception and we include
10              # information about its meaning.
11              raise ZeroDivisionError('Denominator is 0')
12          return float(num) / float(den)
13
14
15  print(Operations().divide(3, 4))
16  print(Operations().divide(3, 0))
```

```
0.75
Traceback (most recent call last):
  File "10.py", line 16, in <module>
    print(Operations().divide(3, 0))
  File "10.py", line 11, in divide
    raise ZeroDivisionError('Denominator is 0')
ZeroDivisionError: Denominator is 0
```

When an exception occurs inside a function contained in another one, this exception interrupts all the functions that include it and the main program. Besides the existent exceptions, it is possible to raise exceptions with customized messages:

```
1   # 10.py
2
3
4   class Circle:
5
6       def __init__(self, center):
7           if not isinstance(center, tuple):
8               raise Exception('Center has to be a tuple')
9               print('This line is not printed')
10
11          self.center = center
12
13      def __repr__(self):
14          return 'The center is {0}'.format(self.center)
```

```
15
16
17 c1 = Circle((2, 3))
18 print(c1)
19
20 c2 = Circle([2, 3])
21 print(c2)
```

```
The center is (2, 3)
Traceback (most recent call last):
  File "11.py", line 20, in <module>
    c2 = Circle([2, 3])
  File "11.py", line 8, in __init__
    raise Exception('Center has to be a tuple')
Exception: Center has to be a tuple
```

5.3 Exception handling

Every time an exception occurs, it is possible to handle it with the statements `try` and `except`. When a block of instructions defined inside the *try* statement triggers an exception, and the `except` processed it using the instructions inside of it. Afterward, the program continues its execution normally and does not stop or crash. The block of instructions inside the `except` defines how the program behaves depending on the exception type. In the following example, we observe that the program does not crash, even though occurs an invalid operation inside the `try` block:

```
1  # 11.py
2
3
4  class Operations:
5
6      @staticmethod
7      def divide(num, den):
8          # This method will raise an exception when denominator be 0
9          return float(num) / float(den)
10
11
12 try:
```

```
    # Here we manage the exceptions during the runtime of the function.
    # The first case will return an output and the second case will yield and
    # error beacuse denominator is 0. The output in this wont be printed.

    print('First case: {}'.format(Operations().divide(4, 5)))
    print('Second case: {}'.format(Operations().divide(4, 0)))

except ZeroDivisionError as err:
    print('Error: {}'.format(err))
```

```
First case: 0.8
Error: float division by zero
```

We can handle separately different types of exceptions by adding more specific exception blocks (ex: `ZeroDivisionError`, `TypeError`, `KeyError`, etc.), each one *catches* the exceptions according to the exception type that occurred:

```
# 12.py


class Operations:

    @staticmethod
    def divide(num, den):
        # Check if the input parameters are a valid type
        if not (isinstance(num, int) and isinstance(den, int)):
            raise TypeError('Invalid input type.')

        # Check the numerator and denominator are greater than 0
        if num < 0 or den < 0:
            # The message inside brackets will show once the
            # exception has been handled.
            raise Exception('Negative values. Check the input parameters')

        return float(num) / float(den)


```

```
# In this code section we manage the runtime exception using try and except
# sentences.

# First example, using float values
try:
    print('First case: {}'.format(Operations().divide(4.5, 3)))

except (ZeroDivisionError, TypeError) as err:
    # This block works with the already defined exception types
    print('Error: {}'.format(err))

except Exception as err:
    # This block only handles type Exception exceptions
    print('Error: {}'.format(err))


# Second example, using negative values
try:
    print('Second case: {}'.format(Operations().divide(-5, 3)))

except (ZeroDivisionError, TypeError) as err:
    # This block works with the already defined exception types
    print('Error: {}'.format(err))

except Exception as err:
    # This block only handles type Exception exceptions
    print('Error: {}'.format(err))
```

```
Error: Invalid input type.
Error: Negative values. Check the input parameters
```

If we do not use any particular exception name after the `Except` statement, it caught any exception triggered in the `try`. The `try` and `except` blocks can be complemented by the `else` and `finally` statements. The `else` block execute the instructions inside of it only in case no exception happened. The `finally` statement always executes the instructions defined in its block. This statement is commonly used to trigger cleaning actions, such as closing a file, database connections, etc. The following code shows an example:

```
# 13.py


class Operations:

    @staticmethod
    def divide(num, den):
        if not (isinstance(num, int) and isinstance(den, int)):
            raise TypeError('Invalid input type')

        if num < 0 or den < 0:
            raise Exception('Negative input values')

        return float(num) / float(den)


# The complete Try/Except structure

try:
    # Check if we can excecute this operation
    resultado = Operations.divide(10, 0)

except (ZeroDivisionError, TypeError):
    # This block works with the already defined exception types
    print('Check the input values. '
          'They aren\'t ints or the denominator is 0')

except Exception:
    # This block only handles type Exception exceptions
    print('All given values are negative')

else:
    # When we do no have errors, the program excecute this lines
    print('Everything is ok. They were no errors')

```

```
36 finally:
37     print('Rebember to ALWAYS use this structure to handle your runtime'
38           'errors')
```

```
Check the input values. They aren't ints or the denominator is 0
Rebember to ALWAYS use this structure to handle your runtimeerrors
```

5.4 Creating customized exceptions

In Python, there are three main types of exceptions: `SystemExit`, `KeyboardInterrupt` and `Exception`. All of the them inherits from `BaseException`. All the other exceptions suchs as the exceptions generated by errors iherits from the `Exceptions` class, as is shown in Figure 5.1:

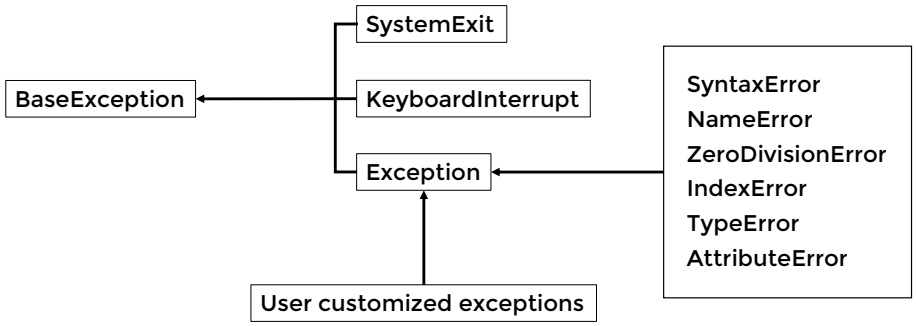

Figure 5.1: Diagram of exceptions hierarchy. All exceptions descended from a the general exception `BaseException`. We can create our new exceptions inheriting from the base class `Exception`.

The previous diagram explains the reason why using only the `Exception` statement without specifying exceptions catches any error. All of them are subclasses of `Exception`.

```
1 # 14.py
2
3
4 class Operations:
5
6     @staticmethod
7     def divide(num, den):
8         if not (isinstance(num, int) and isinstance(den, int)):
9             raise TypeError('Invalid input type')
```

```
10
11          if num < 0 or den < 0:
12              raise Exception('Negative input values')
13
14          return float(num) / float(den)
15
16
17  # In this section we handle the excetions
18  try:
19      print(Operations().divide(4, 0))
20
21  except Exception as err:
22      # This block works for all exception types.
23      print('Error: {}'.format(err))
24      print('Check the input')
```

```
Error: float division by zero
Check the input
```

To create customized exceptions, we need that the custom exception inherits from the `Exception` class. The following code shows an example:

```
1   # 15.py
2
3
4   class Exception1(Exception):
5       pass
6
7
8   class Exception2(Exception):
9
10      def __init__(self, a, b):
11          super().__init__("One of the values {0} or {1} is not integer"
12                           .format(a, b))
13
14
15  class Operations:
```

```
16
17      @staticmethod
18      def divide(num, den):
19          # In this example, we re-define exceptions that we used in the last
20          # examples.
21
22          if not (isinstance(num, int) and isinstance(den, int)):
23              raise Exception2(num, den)
24
25          if num < 0 or den < 0:
26              raise Exception1('Negative values\n')
27
28          return float(num) / float(den)
29
30
31  # This case raise the exception 1
32  try:
33      print(Operations().divide(4, -3))
34
35  except Exception1 as err:
36      # This block works for type one exception
37      print('Error: {}'.format(err))
38
39  except Excepcion2 as err:
40      # This block works for type two exception
41      print('Error: {}'.format(err))
42
43
44  # This case raise the exception 2
45  try:
46      print(Operations().divide(4.4, -3))
47
48  except Exception1 as err:
49      # This block works for type one exception
50      print('Error: {}'.format(err))
```

```
51
52 except Exception2 as err:
53     # This block works for type two exception
54     print('Error: {}'.format(err))
```

```
Error: Negative values

Error: One of the values 4.4 or -3 is not integer
```

Here we show another example:

```
1  # 16.py
2
3  class TransactionError(Exception):
4      def __init__(self, funds, expenses):
5          super().__init__("The money on your wallet is not enough to pay ${}"
6                           .format(expenses))
7          self.funds = funds
8          self.expenses = expenses
9
10     def excess(self):
11         return self.funds - self.expenses
12
13
14 class Wallet:
15     def __init__(self, money):
16         self.funds = money
17
18     def pay(self, expenses):
19         if self.funds - expenses < 0:
20             raise TransactionError(self.funds, expenses)
21         self.funds -= expenses
22
23 if __name__ == '__main__':
24     b = Wallet(1000)
25
26     try:
```

```
27          b.pay(1500)
28      except TransactionError as err:
29          print('Error: {}'.format(err))
30          print("There is an excess of expenses of ${}".format(err.excess()))
```

```
Error: The money on your wallet is not enough to pay $1500
There is an excess of expenses of $-500
```

Notes

Handling exceptions is another way of controlling the program's flow, similar to `if-else` sentences. It is recommended to use exceptions to control the errors in the program. We always can to create a sort of "error codes" for managing the returned values or results in different kind of operations. However, this solution makes the program and other modules hard to maintain. The number of error codes may grow at the same time the number of possible outputs we need to control. It makes our program impossible to be understood by any other programmers. A clearer example of the reason we need to handle exceptions is that in general, our program has to notify other applications that a particular error occurred. This kind of notifications would not be possible with the use of error codes. It is also critical that our code does not unexpectedly crash because during a crash the interpreter usually exposes in the output part of code that triggers the error. We have to avoid this situation if we correctly handle the exceptions.

5.5 Hands-On Activities

Activity 5

The Physics Club of your University *The Absolute Zeros* decided to program a calculator, whose main feature is that it handles letters and numbers as input. The person who was working on the project decided to change to the chemistry club *Cooler than Absolute Zero* and left the calculator unfinished. The calculator works well for some operations, but many times it triggers errors, and no one has been able to fix it. The president of the Physics Club has asked you to fix the code, such that it does not crash whenever is possible, handling the errors produced by user's input.

Consider that the president sent you a list of tested operations that work correctly: `tested_operations`; and a list with operations that do not work: `statements_for_testing`. We provide the file *AC05_0_provided_code.py* with the code. You cannot modify the code from line 87. Your job is to write the code to handle the exceptions and obtain the following output:

```
[ERROR] KeyError
```

```
Letter ‘g’ won’t be aggregated. It already exist in memory.
[ERROR] ZeroDivisionError
1 divided 0 is equal to infinite
[ERROR] KeyError
‘a’ doesn’t have any assigned values .It must be added before using it.
The operation a+2 wasn’t executed
9.81 plus 0 is equal to 9.81
88 divided by 2 is equal to 44.0
44.0 plus 0 is equal to 44.0
[ERROR] StopIteration
There’s one missing operator in 88/2+0+
[ERROR] ValueError
‘1=2’ cannot be parsed to float
[ERROR] The syntax ’1=2’ is incorrect. Read the manual for more information.
8.953 plus 1 is equal to 9.953
```