Chapter 6 - Errors

In this chapter you will learn how to deal with errors in your code. You will learn different types of errors, and how you can catch specific exceptions.

Types of errors

Errors in Python can be categorized into two types:

1. Compile time errors – errors that occur when you ask Python to run the application. Before the program can be run, the source code must be compiled into the machine code. If the conversion can not perfomed, Python will inform you that your application can not be run before the error is fixed. The most common errors of this type are syntax errors – for example, if you don’t end an **if statement** with the **colon**. Here is an example:

```
x = int(input('Enter a number: '))


if x%2 == 0
   print('You have entered an even number.')
else:
   print ('You have entered an odd number.')
```

The code above checks if the number the user enters is an odd or an even number. However, notice how the **if statement** is missing the **colon** (**:**) at the end of the line. Because of it, the program won’t run and the interpreter will even inform us what the problem is:

```
C:Python34Scripts>python error.py
File "error.py", line 3
if x%2 == 0
^
SyntaxError: invalid syntax
```

2. Runtime errors – errors that occur after the code has been compiled and the program is running. The error of this type will cause your program to behave unexpectedly or even crash. An example of an runtime error is the division by zero. Consider the following example:

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))
z = x/y


print (x,'divided by',y,'equals: ',z)
```

The program above runs fine until the user enters **0** as the second number:

```
>>>
```

```
Enter a number: 9
Enter a number: 2
9.0 divided by 2.0 equals: 4.5
>>>
Enter a number: 11
Enter a number: 3
11.0 divided by 3.0 equals: 3.6666666666666665
>>>
Enter a number: 5
Enter a number: 0
Traceback (most recent call last):
File "C:/Python34/Scripts/error1.py", line 3, in <module>
z = x/y
ZeroDivisionError: float division by zero
>>>
```

Syntax and logical errors

Two types of errors can occur in Python:

1. Syntax errors – usually the easiest to spot, syntax errors occur when you make a typo. Not ending an **if** statement with the colon is an example of an syntax error, as is misspelling a Python keyword (e.g. using **whille** instead of **while**). Syntax error usually appear at compile time and are reported by the interpreter. Here is an example of a syntax error:

```
x = int(input('Enter a number: '))


whille x%2 == 0:
   print('You have entered an even number.')
else:
   print ('You have entered an odd number.')
```

Notice that the keyword **whille** is misspelled. If we try to run the program, we will get the following error:

```
C:Python34Scripts>python error.py
  File "error.py", line 3
    whille x%2 == 0:
           ^
SyntaxError: invalid syntax
```

2. Logical errors – also called **semantic errors**, logical errors cause the program to behave incorrectly, but they do not usually crash the program. Unlike a program with syntax errors, a program with logic errors can be run, but it does not operate as intended. Consider the following example of an logical error:

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))

z = x+y/2
print ('The average of the two numbers you have entered is:',z)
```

The example above should calcuate the average of the two numbers the user enters. But, because of the order of operations in arithmetic (the division is evaluated before addition) the program will not give the right answer:

```
>>>
Enter a number: 3
Enter a number: 4
The average of the two numbers you have entered is: 5.0
>>>
```

To rectify this problem, we will simply add the parentheses: **z = (x+y)/2**

Now we will get the right result:

```
>>>
Enter a number: 3
Enter a number: 4
The average of the two numbers you have entered is: 3.5
>>>
```

The try…except statements

To handle errors (also known as exceptions) in Python, you can use the **try…except** statements. These statements tell Python what to do when an exception is encountered. This act of detecting and processing an exception is called **exception handling**. The syntax of the **try…except** statements is:

```
try:
    statements # statements that can raise exceptions
except:
    statements # statements that will be executed to handle exceptions
```

If an exception occurs, a **try** block code execution is stopped and an except block code will be executed. If no exception occurs inside the **try** block, the statements inside the **except** block will not be executed.

Consider the following example:

```
age=int(input('Enter your age: '))


if age <= 21:
    print('You are not allowed to enter, you are too young.')
else:
    print('Welcome, you are old enough.')
```

The example above asks the user to enter his age. It then checks to see if the user is old enough (older than 21). The code runs fine, as long as the user is entering only numberic values. However, consider what happens when the user enters a string value:

```
>>>
Enter your age: 13
You are not allowed to enter, you are too young.
>>> RESTART
>>>
Enter your age: 22
Welcome, you are old enough.
>>> RESTART
```

```
>>>
Enter your age: a
Traceback (most recent call last):
  File "C:/Python34/Scripts/exceptions.py", line 2, in <module>
    age=int(input('Enter your age: '))
ValueError: invalid literal for int() with base 10: 'a'
>>>
```

Because a numeric value is expected, the program crashed when the user entered a non-numeric value. We can use the **try…except** statements to recify this:

```
try:
    age=int(input('Enter your age: '))
except:
    print ('You have entered an invalid value.')
```

Now, the code inside the **try** block has its exceptions handled. If the user enters a non-numeric value, the statement inside the **except** block will be executed:

```
>>>
Enter your age: a
You have entered an invalid value.
>>>
```

In the example above the **except** clause catches all the exceptions that can occur, which is not considered a good programming practice. The **except** clause can have a specific exception associated, which we will describe in the following lessons.

The try…except…else statements

You can include an **else** clause when catching exceptions with a **try** statement. The statements inside the **else** block will be executed only if the code inside the **try** block doesn’t generate an exception. Here is the syntax:

```
try:
    statements # statements that can raise exceptions
except:
    statements # statements that will be executed to handle exceptions
```

```
else:
    statements # statements that will be executed if there is no exception
```

Here is an example:

```
try:
    age=int(input('Enter your age: '))
except:
    print ('You have entered an invalid value.')
else:
    if age <= 21:
        print('You are not allowed to enter, you are too young.')
    else:
        print('Welcome, you are old enough.')
```

The output:

```
>>>
Enter your age: a
You have entered an invalid value.
>>> RESTART
>>>
Enter your age: 25
Welcome, you are old enough.
>>>RESTART
>>>
Enter your age: 13
You are not allowed to enter, you are too young.
>>>
```

As you can see from the output above, if the user enters a non-numeric value (in this case the letter **a**) the statement inside the **except** code block will be executed. If the user enters a numeric value, the statements inside the **else** code block will be executed.

The try…except…finally statements

You can use the **finally** clause instead of the **else** clause with a **try** statement. The difference is that the statements inside the **finally** block will always be executed, regardless whether an exception occurrs in the **try**block. Finally clauses are also called clean-up or termination clauses, because they are usually used when your program crashes and you want to perform tasks such as closing the files or logging off the user. Here is the syntax:

```
try:
   statements # statements that can raise exceptions
except:
   statements # statements that will be executed to handle exceptions
finally:
   statements # statements that will always be executed
```

Here is an example:

```
try:
age=int(input('Enter your age: '))
except:
print ('You have entered an invalid value.')
finally:
print ('There may or may not have been an exception.')
```

The output:

```
>>>
Enter your age: 55
There may or may not have been an exception.
>>> RESTART
>>>
Enter your age: a
You have entered an invalid value.
There may or may not have been an exception.
>>>
```

Note that the **print** statement inside the **finally** code block was executed regardless of whether the exception occured or not.

Catch specific exceptions

We've already mentioned that catching all exceptions with the **except** clause and handling every case in the same way is not considered to be a good programming practice. It is recommended to specify the exact exceptions that the **except** clause will catch. For example, to catch an exception that occurs when the user enters a non-numerical value instead of a number, we can catch only the built-in **ValueError** exception that will handle such event. Here is an example:

```
try:
    age=int(input('Enter your age: '))
except ValueError:
    print('Invalid value entered.')
else:
    if age >= 21:
        print('Welcome, you are old enough.')
    else:
        print('Go away, you are too young.')
```

The code above will ask the user for his age. If the user enters a number, the program will evaluate whether the user is old enough. If the user enters a non-numeric value, the **Invalid value entered** message will be printed:

```
>>>
Enter your age: 5
Go away, you are too young.
>>>RESTART
>>>
Enter your age: 22
Welcome, you are old enough.
>>>RESTART
>>>
Enter your age: a
Invalid value entered.
>>>
```

In the output above, you can see that the statements inside the **else** clause were executed when the user entered a number. However, when the user entered a non-numeric value (the letter **a**), the **ValueError** exception occured and the **print** statement inside the except **ValueError** clause was executed.

Note that the **except ValueError** clause will catch only exceptions that occur when the user enters a non-numeric value. If another exception occur, such as the **KeyboardInterrupt** exception (raised when the user hits **Ctrl+c**), the **except ValueError** block would not handle it. You can, however, specify multiple except clauses to handle multiple exceptions, like in this example:

```
try:
   age=int(input('Enter your age: '))
except ValueError:
   print('Invalid value entered.')
except KeyboardInterrupt:
   print('You have interrupted the program.')
else:
   if age >= 21:
      print('Welcome, you are old enough.')
   else:
      print('Go away, you are too young.')
```

The output:

```
>>>
Enter your age: 5
Go away, you are too young.
>>> RESTART
>>>
Enter your age: 22
Welcome, you are old enough.
>>>RESTART
>>>
Enter your age: a
Invalid value entered.
>>>
>>>
```

```
Enter your age:
You have interrupted the program.
>>>
```

You can also handle multiple exceptions with a single **except** clause. We can simple rewrite our program like this:

```
try:
    age=int(input('Enter your age: '))
except (ValueError, KeyboardInterrupt):
    print('There was an exception.')
else:
    if age >= 21:
        print('Welcome, you are old enough.')
    else:
        print('Go away, you are too young.')
```

Raise exception

You can manually throw (raise) an exception in Python with the keyword **raise**. This is usually done for the purpose of error-checking. Consider the following example:

```
try:
    raise ValueError
except ValueError:
    print('There was an exception.')
```

The code above demonstrates how to raise an exception. Here is the output:

```
>>>
There was an exception.
>>>
```

You can use the **raise** keyword to signal that the situation is exceptional to the normal flow. For example:

```
x = 5
if x < 10:
    raise ValueError('x should not be less than 10!')
```

Notice how you can write the error message with more information inside the parentheses. The example above gives the following output (by default, the interpreter will print a traceback and the error message):

```
>>>
Traceback (most recent call last):
  File "C:/Python34/Scripts/raise1.py", line 3, in <module>
    raise ValueError('x should not be less than 10!')
ValueError: x should not be less than 10!
>>>
```