Chapter 9: Dealing with Errors

What we will learn:

- ✓ How to identify errors
- ✓ Categorising different types of error
- ✓ How to fix different errors
- ✓ Example of errors

What you need to know before:

- ✓ Writing simple programs

Keywor

Runtime errors Syntax Semantics Recursive Exceptions Errors

9.1 Exceptions and Errors

There are different types of error that can occur when we are writing programs. In this chapter, we will discuss them and examine possible solutions.

Types of Errors

Type of Error	Meaning	Example

Syntax	A syntax error occurs when the programmer uses a token (letter, symbol and operator) somewhere that is not defined in the grammar of the language.
	
	Programming languages have rules, similar to other forms of communication. The interpreter takes the source code and parses the tokens, then sends this parsed byte code to the evaluator to get its meaning.
		`if age<18`
		`  print("Child")`
		
		Here the colon is missing after the condition (`age<18`)

Runtime	A runtime error occurs during execution of a program. The programmer normally thinks about these but can do nothing about them. They are often dealt with by the exception handler.	Program running out of memory Faulty hardware

Semantic or logical	Semantic refers to the meaning of things. These errors happen when there are errors with the logic of the program, which gives unexpected results. The interpreter will not raise an exception or give an error message for semantic errors, but the user will not get the expected result.
	
	In evaluating $\frac{15}{2\pi}$ in Python we would type `15/2*math.pi`
	As division and multiplication have the same precedence, we would get 23.56, instead of the correct answer 2.38. In order to get the correct answer we would have to use brackets; therefore, we would type `15/(2*math.pi)`
		For $\frac{12}{5\pi}$
		
		`12/5*math.pi`
		incorrect
		
		`12/(5*math.pi)`
		correct

Table 7

In general, the term ***exception*** is used to describe when something goes wrong in a program; this include errors caused by the programmer and external factors, such as hardware failure. Some programming languages make explicit differentiation between an error caused by a programmer and an exception. In Python, the difference is more subtle in that both errors and exceptions are handled by the BaseException class. Subclasses are then derived which are either errors or exceptions.

We will now turn our attention to dealing with these kinds of error and then to exceptions.

9.2 Syntax Error

Syntax errors are quite easy to fix. The interpreter will raise this error where it notices that the error occurred. This is often at the exact location or sometimes in the previous line. For example, if a token such as a closed bracket or a colon is missing from the previous line, the syntax error is normally noticed and highlighted in the following line. The code snippet below demonstrates this.

```
>>> print("I am leaving out the closed bracket here"
      but when I run the program the error is in the wrong place

SyntaxError: invalid syntax
>>>
```

We will also get a syntax error if we group two operators together, for example:

```
>>> 5 + 4 +*
SyntaxError: invalid syntax
>>>
```

This will generate a syntax error, I'll use Backus–Naur Form to explain why we get a syntax error for the example above.

The grammar of almost all programming languages is described in **Backus–Naur Form (BNF)**. BNF was developed by John Backus in the 1950s; he was very influential in developing the FORTRAN programming language. The purpose of BNF is to develop a concise way of describing a programming language. The structure of BNF is:

<non-terminal> → replacement

The replacement can be replaced with zero or more non-terminal or terminal.
Terminals terminates the statement; once it is used it cannot be replaced.

The grammar for mathematical statements can be expressed by the following notation:

expression → expression operator expression
expression → number
*operator → +, -, *, /, %, ...*
number → 0, 1, 2, 3, 4, ...

You will notice that there are two lines for defining the expression. Together we call this a recursive definition. A recursive definition is where we define something in terms of itself, but there is also a known definition called the base case. This qualifies the expression as a recursive definition as the expression can be replaced by an expression and also by number, which is a terminal. This is a very powerful concept in computer science and is used to define and break down many large problems.

How do we use this grammar?
Suppose we want to generate the expression **12 + 3 * 15**.
We can use the following steps:

1. Start out with an expression	*Expression*
2. Replace with expression → expression	***expression operator expression***
3. Replace expression → number	***number*** *operator expression*
4. Replace number →12	***12*** *operator expression*
5. Replace operator → +	*12* ***+*** *expression*
6. Replace expression → expression operator expression	*12 +* ***expression operator expression***
7. Replace expression → number	*12 +* ***number*** *operator expression*
8. Replace number → 3	*12 +* ***3*** *operator expression*
9. Replace operator → *	*12 + 3* ******* *expression*
10. Replace expression → number	*12 + 3 ** ***number***
11. Replace number → 15	*12 + 3 ** ***15***

The replaced is highlighted on the right.

Now let us try to generate **5 + 4 +***.

1. Start out with an expression	*Expression*
2. Replace with expression → expression	***expression operator expression***
3. Replace expression → number	***number*** *operator expression*
4. Replace number →5	***5*** *operator expression*
5. Replace operator → +	*5* ***+*** *expression*
6. Replace expression → exp op exp	*5 +* ***expression operator expression***
7. Replace expression → number	*5 +* ***number*** *operator expression*
8. Replace number → 4	*5 +* ***4*** *operator expression*
9. Replace operator → +	*5 + 4* ***+*** *expression*
10. Error we cannot replace expression → operator	***5 + 4 + SyntaxError***

It turns out that it is impossible to generate **5 + 4 + *** using the grammar defined above, and therefore the interpreter will generate a syntax error when to try to parse the expression.

To reduce or eliminate syntax errors:

- Ensure that operators are surrounded by operands (Python uses what is called an infixed notation and therefore, the operator must be in the middle of the operands.
- Ensure that all opening brackets (, {, [have a matching closing bracket], }.).
- Ensure that all opening quotation marks (" or ') have a matching closing quotation mark (" or ').
- Ensure that you include a colon (:) at the end of your IF statements or other control structures.
- Ensure that you are using the correct operator, for example, = means assignment and == is equal to.
- Ensure that you are not trying to use a keyword as a variable name; this will generate a syntax error.
- Ensure that you are using the correct indentation. Python is indent sensitive; typically we indent after a colon (:) and return to the original alignment afterwards.

9.3 Runtime Errors

Runtime errors are more difficult to spot, largely because the program will run without reporting an error to the user which gives the impression that everything is working well. Runtime errors can happen in a few ways:

- The program runs without an output. If this is the case, then check to ensure that there is an output statement in the code, unless the code is a module that will be used for utilities.
- If the program hangs and become inactive. This normally happens when the program enters an indeterminate loop, without statements to make the condition false. This was mentioned when I introduced the while loop. If this happens, the program will run forever. I'll use an example to illustrate this.

```
number = 1
while number<10:
    print("The value of number is ", number)
    # notice that there is no statement to make the condition false
```

  Notice that in the body of the while loop there is no statement that will change the value of number, and therefore the number will always be less than 10, and hence the condition will always be true. If this code snippet was in your code, the error would be clear as there is a print statement in the body of the while loop that will be printing "The value of number is 1" on your screen. Therefore, we could identify where the error is. Now if there was no print statement, for example, the while loop was updating a database or traversing a list for each iteration, then this would become more difficult to identify.

- Runtime errors can happen when we have a recursive call that does not "bottom out". We use the term "bottom out" to refer to the situation when either there is no base case or the function will never get to the base case. Let us think about the recursive definition for expression in BNF that was mentioned in the syntax error section. We defined expression as:
  expression → expression operator expression
  expression → number

  If the second statement were absent (*expression → number*), then it would not be possible to get to a situation where we have expressions in all terminals (numbers and operators).

  Let us use a recursive definition to define a function called `power_number()` that takes two values as input, namely a base and an index.

```
def power_number(base, index):
    if index ==0:
        return 1
    else:
        return base * power_number(base, index-1)
```

  The base condition in this situation is to return 1 whenever the index is equal to 0. This is consistent with mathematics, in that any number raised to the power of zero is equal to 1. Now if this base condition was absent or impossible to get to, then after running for some time the interpreter will return a runtime error, stating that maximum recursion depth is exceeded in comparison.

```
def power_number(base, index):
    if index ==15:
        return 1
    else:
        return base * power_number(base, index-1)


print(power_number(4,12))
```

In the example above, the index will never be equal to 15, as the index starts at 12 from our print statement and decreases in value.

```
ors and Exception/errors.py", line 23, in power_number
    return base * power_number(base, index-1)
  File "C:/Users/Devz/Downloads/Python Book/Python for Key stage 4/Chapter 9 Err
ors and Exception/errors.py", line 23, in power_number
    return base * power_number(base, index-1)
  File "C:/Users/Devz/Downloads/Python Book/Python for Key stage 4/Chapter 9 Err
ors and Exception/errors.py", line 20, in power_number
    if index ==15:
RuntimeError: maximum recursion depth exceeded in comparison
>>>
>>>
```

Whenever you suspect that a recursive definition is not getting to the base case, it is worth using a trace table to go through a simple case, or using print statements to display the value of the variables at strategic locations in the function call.

9.4 Semantic or Logical Error

Semantic refers to the meaning of a language or logic. Semantic errors are even more difficult to spot than runtime errors. Similar to runtime errors, the interpreter will not give you any messages or warnings. The program will even run without hanging, or giving a maximum recursion depth message. Instead, the output will not be the one that you are expecting.

Fortunately there are strategies available to the programmer to identify and correct semantic errors.

- Inserting `print()` statement at strategic locations in your code to see the values of variables is quite helpful.
- The use of trace tables to record the value of variables can be also helpful.
- Sometimes it is worth breaking down complex expressions into simpler statements, rather than having large complex statements.

  If you define a function called `larger()` that takes two numbers and returns the larger of the two. It is possible to use *`larger(num1, larger(num2,num3)`*, to return the larger of three numbers. That is equivalent to:

  ```
  temp_num = larger(num1, num2)
  larger(temp_num, num3)
  ```

  This may seem trivial, but it turns out that it is less likely to make mistakes on simpler expressions than on complex ones.

- Semantic errors often arise from the use of incorrect operators, for example, using the < sign instead of the >.
- It is often worth using brackets to explicitly force the correct order of operation, from our example in the table above:

  In evaluating $\frac{15}{2\pi}$ in Python we would type `15/2*math.pi`

  As division and multiplication have the same precedence, we would get 23.56, instead of the correct answer 2.38. In order to get the correct answer we would have to use brackets, therefore, we would type `15/ (2*math.pi)`

9.5 Dealing with Errors

Even with the best intentions, there will be situations in which errors occur in a program that were not foreseen. Many programming languages provide error handing facilities for such situations. In Python, all exceptions are instances of the BaseException class and are implemented using the try statement. The debugger tool in Python is called the Python debugger and is imported using the `import pdb`. There are two versions of the try statement, namely try-except and try-finally.

9.5.1 The Try Statement

Syntax:

```
try:
<block_to_execute>

except <exception>:
      <block_to_execute>
```

`<block_to_execute>` is the originally intended code.
`except` must include at least one exception, but can include as many as you need.

Example:

```
try:
    x=int(input("Enter a number to divide: "))
    y=int(input("Enter a number to divide by: "))
    print(x/y)
except ZeroDivisionError:
          print("Cannot divide by zero: ")
          y=1
```

This will catch the exception then print a more meaningful statement and continue with the execution of the program.

In the example above, the interpreter will try to execute all the statements in the try block. If the user enters zero for the y value, the ZeroDivisionError exception will be raised. Instead of halting the program, the interpreter will catch the exception in the “except” clause and recover from the unstable state by reassigning y to 1. We could have included other exceptions in the “except” clause, and the interpreter would search until it finds a matching exception to enter into the “except” clause. If no matching exception is found, the program will halt/stop and print the exception to the screen.

The example below shows how we can catch an ImportError (import a file/module that does not exist).

```
>>> import unknown_module
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    import unknown_module
ImportError: No module named 'unknown_module'
```

This is the import statement without the exception handler. Note that this would stop the running program.

```
>>> try:
        import unknow_module
except ImportError:
        print("Sorry module not found")


Sorry module not found
>>>
```

With the exception handler, the program will recover and continue to run.

9.5.2 The Try Statement with Multiple Exceptions

There are situations in which more than one exception can occur in a particular code. You have the option of writing multiple except clauses, where you check each exception and resolve it, or you can use one except clause where the interpreter will check through a list of exceptions. It turns out that if there is one way to deal with all exceptions, then the latter is a better option, but if each exception requires a separate solution then having multiple except clauses is the better solution.

If we are writing a program and we want the user to enter a number and then carry out some mathematical operation on this number, we will use the int() function to convert the data entered to an integer. There are two possible exceptions that can occur in this situation, namely:

TypeError: *when an operation or function is attempted that is invalid for the specified data type.*
ValueError: *values have the valid type of arguments, but the arguments have invalid values specified.*

The code below will catch both the ValueError and TypeError exception and will print the associated message. In the situation where the course of action is dependent on the type of error, then the solution below is applicable.

```
try:
    x=int(input("Enter a number: "))
except ValueError:
    print("You entered an incorrect value: ")
except TypeError:
    print("The data type is invalid ")


print("Continuing after the input ...")
```

9.5.3 The Optional Else Clause

The optional else clause is used when you only want to execute statements if no exceptions were raised in the try clause.

```
try:
    x=int(input("Enter a number: "))
except (ValueError, TypeError):
    print("Something went wrong")
else:
    print("Nothing went wrong")
```

Notice that if the user's input is valid, then Python will execute the optional else clause. Notice also that in the code snippet above, we have changed from using multiple except clauses to a single one with multiple exceptions (ValueError and TypeError).

9.5.4 The Try-Finally Statement

If we have multiple statements in the try block and one of the statements raises an exception before the end of the try block, the other statements in the try block will be executed. There are situations in which you want the entire try block to be executed, even if an exception was raised. It turns out that the try-finally statement will resolve this.

Syntax:

```
try:
    <block_to_execute>

finally:
    <finally_block>
```

<block_to_execute> is the originally intended code.
finally must block that; it will be executed regardless of any exception.

Example:

If we are writing a program that opens a file and do some calculations then close the file. It is a good idea to raise an exception after attempting to open the file. If an error occurs while trying to open the file, then the IOError exception is raised. Consider the code below.

```
try:
    test_file = open("example_file.txt", 'w')
    test_file.write("Some operations")
except IOError:
    print("File don't exist ...")
    test_file.close()
```

If we encounter an error while trying to write to the file, the IOError exception will be raised; the "except" clause will handle the exception and close the file. If no exception is raised, then the file will remain open, which is not good for our program. It turns out that the "try-finally" can help to solve this problem. Whenever we use the "try-finally" statement if an exception is raised, then execution is passed to the finally clause and then to the exception handler in the "except" section.

The code snippet below gives this solution.

```
try:
    test_file = open("example_file.txt", 'w')
    try:
        test_file.write("Some operations")
    finally:
        test_file.close()
except IOError:
    print("File don't exist ...")
```

Notice that if the write statement fails then the file will be closed.