

The Quick Python Overview

Chapter 3

This Chapter Covers

- Surveying Python
- Using built-in data types
- Controlling program flow
- Creating modules
- Using Object-Oriented programming

Python Synopsis

- Python has several built-in data types, such as integers, floats, complex numbers, strings, lists, tuples, dictionaries, and file objects.
- These data types can be manipulated using language operators, built-in functions, library functions, or a data type's own methods.
- Programmers can also define their own classes and instantiate their own class instances.
- These class instances can be manipulated by programmer-defined methods, as well as the language operators and built-in functions for which the programmer has defined the appropriate special method attributes.

Python Synopsis

- Python provides conditional and iterative control flow through an if-elif-else construct along with while and for loops. It allows function definition with flexible argument-passing options.
- Exceptions (errors) can be raised by using the raise statement, and they can be caught and handled by using the try-except-else-finally construct.
- Variables (or identifiers) don't have to be declared and can refer to any built-in data type, user-defined object, function, or module.

Built-in Data Types

- Numbers
- Lists
- Tuples
- Strings
- Dictionaries
- Sets
- File Objects

```
9  # define a variable age and assign value 17
10 age = 17
11
12 # print age to output
13 print(age)
```

Numbers

- Python's four number types are integers, floats, complex numbers, and Booleans:
 - Integers—1, -3, 42, 355, 888888888888888888, -77777777777 (integers aren't limited in size except by available memory)
 - Floats—3.0, 31e12, -6e-4
 - Complex numbers—3 + 2j, -4- 2j, 4.2 + 6.3j
 - Booleans—True, False

```
1  # define a variable age and assign value 17
2  age = 17
3
4  # print age to output
5  print(age)
6
7  # numbers
8  print(5 + 2 - 3 * 2)
9  print(5 / 2)      # floating 2.5
10 print(5 / 2.0)    # also floating 2.5
11 print(5 // 2)     # integer result 2
12 print(3000000000) # a large number
13 print(3000000000 * 3)
14 print(3000000000 * 3.0)
15 print(2.0e-8)     # scientific
```

Numbers

```
14 # lists and slicing
15 x = ["first", "second", "third", "fourth"]
16
17 # display all values in list
18 print(x)
19
20 # display the first value in list
21 print(x[0])
22
23 #slicing
24 print(x[1:-1])
25 print(x[0:3])
26 print(x[-2:-1])
27 print(x[:3])
28 print(x[-2:])
```

Lists

- A list can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number.
- A list can be indexed from its front or back. You can also refer to a subsegment, or slice, of a list by using slice notation.


```
1 # A list can be converted to a tuple by using the built-in function tuple
2 x = [1, 2, 3, 4]
3 print(x)
4 print(tuple(x))
5
6 # Conversely, a tuple can be converted to a list by using the built-in function list
7 x = (1, 2, 3, 4)
8 print(x)
9 print(list(x))
```

Tuples

- Tuples are similar to lists but are immutable - that is, they can't be modified after they've been created.
- A list can be converted to a tuple by using the built-in function tuple and vice-versa using the built-in function list.

```
1 # strings
2 str1 = "A string in double quotes can contain 'single quote' characters."
3 str2 = 'A string in single quotes can contain "double quote" characters.'
4 str3 = '''\tA string which starts with a tab; ends with a newline character.\n'''
5 str4 = """This is a triple double quoted string, the only kind that can
6 contain real newlines."""
7
8 print(str1)
9 print(str2)
10 print(str3)
11 print(str4)
```

Strings

- String processing is one of Python's strengths.
- Strings can be delimited by single (' '), double (" "), triple single (''' '''), or triple double (""" """) quotations and can contain tab (\t) and newline (\n) characters.
- Strings are also immutable.

```
13 # define string s
14 x = "live and let \t \tlive"
15 print(x)
16
17 # split
18 y = x.split()
19 print(y)
20
21 # replace
22 z = x.replace(" let \t \tlive", "enjoy life")
23 print(z)
24
25 import re
26 regexpr = re.compile(r"[\t ]+")
27 w = regexpr.sub(" ", x)
28 print(w)
```

Strings

- Strings have several methods to work with their contents, and the re library module also contains functions for working with strings.

```
30 # formatted output
31 e = 2.718
32 x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
33 print("The constant e is:", e, "and the list x is:", x)
34
35 print("the value of %s is: %.2f" % ("e", e))
```

- The print function outputs strings. Other Python data types can be easily converted to strings and formatted.

Strings

Dictionaries

- Python's built-in dictionary data type provides associative array functionality implemented by using hash tables.
- The built-in len function returns the number of key-value pairs in a dictionary.
- The del statement can be used to delete a key-value pair.
- As is the case for lists, several dictionary methods (clear, copy, get, items, keys, update, and values) are available
- Keys must be of an immutable type B, including numbers, strings, and tuples.
- Values can be any kind of object, including mutable types such as lists and dictionaries.

```
1  # dictionaries
2  x = {1: "one", 2: "two"}
3  print(x)
4
5  # add key-value pair
6  x["first"] = "one"
7  print(x)
8
9  # get all keys
10 keys = list(x.keys())
11 print(keys)
12
13 # get value
14 print(x[1])
15
16 # optional user-defined value if key-value pair not found
17 print(x.get(1, "not available"))
18 print(x.get(4, "not available"))
```

Dictionaries

Sets

```
1  # sets
2  x = set([1, 2, 3, 1, 3, 5])
3  print(x)
4
5  # in operator
6  print(1 in x)
7  print(4 in x)
```

- A set in Python is an unordered collection of objects, used in situations where membership and uniqueness in the set are the main things you need to know about that object.
- Sets behave as collections of dictionary keys without any associated values.

```
# open file for reading and writing
```

```
f = open("myfile", "w")
```

```
# write into file
```

```
f.write("First line with necessary newline character\n")
```

```
f.write("Second line to write to the file\n")
```

```
f.close()
```

```
# open file for reading only
```

```
f = open("myfile", "r")
```

```
line1 = f.readline()
```

```
line2 = f.readline()
```

```
f.close()
```

```
print(line1, line2)
```

File Objects

A file is accessed through a Python file object.

Control Flow Structures

- Python has a full range of structures to control code execution and program flow, including common branching and looping structures.
- Python has several ways of expressing Boolean values; the Boolean constant False, 0, the Python nil value None, and empty values (for example, the empty list [] or empty string "") are all taken as False.
- The Boolean constant True and everything else is considered True.
- The comparison operators (<, <=, ==, >, >=, !=, is, is not, in, not in) and the logical operators (and, not, or), which all return True or False.

```
1  x = 5
2
3  ✓ if x < 5:
4      |     y = -1
5      |     z = 5
6  ✓ elif x > 5:
7      |     y = 1
8      |     z = 11
9  ✓ else:
10     |     y = 0
11     |     z = 10
12
13  print(x, y, z)
```

The if-elif-else Statement

- The block of code after the first True condition (of an if or an elif) is executed.
- If none of the conditions is True, the block of code after the else is executed.
- The elif and else clauses are optional B, and there can be any number of elif clauses.
- No explicit delimiters, such as brackets or braces, are necessary.
- All these statements must be at the same level of indentation.

```
1  x = 5
2
3  ✓ if x < 5:
4      |     y = -1
5      |     z = 5
6  ✓ elif x > 5:
7      |     y = 1
8      |     z = 11
9  ✓ else:
10     |     y = 0
11     |     z = 10
```

The if-elif-else Statement

- The block of code after the first True condition (of an if or an elif) is executed.
- If none of the conditions is True, the block of code after the else is executed.
- The elif and else clauses are optional B, and there can be any number of elif clauses.
- No explicit delimiters, such as brackets or braces, are necessary.
- All these statements must be at the same level of indentation.

```
1  u, v, x, y = 0, 0, 100, 30
2
3  while x > y:
4      u = u + y
5      x = x - y
6      if x < y + 2:
7          v = v + x
8          x = 0
9      else:
10         v = v + y + 2
11         x = x - y - 2
12 print(u, v)
```

The while Loop

- The while loop is executed as long as the condition (which here is $x > y$) is True.

The for Loop

```
1 item_list = [3, "string1", 23, 14.0, "string2", 49, 64, 70]
2
3 for x in item_list:
4     if not isinstance(x, int):
5         continue
6     if not x % 7:
7         print("found an integer divisible by seven: %d" % x)
8         break
```

- The for loop is simple but powerful because it's possible to iterate over any iterable type, such as a list or tuple.
- Unlike in many languages, Python's for loop iterates over each of the items in a sequence (for example, a list or tuple), making it more of a foreach loop.

Function Definition

```
1  # syntax
2  # def name(param1, param2,...):
3  #     body
4
5  # function that will return the factorial
6  def fact(n):
7      """ Return the factorial of the given number """
8      r = 1
9      while n > 0:
10         r = r * n
11         n = n - 1
12     return r
13
14 print(fact(4))
```

- Functions are defined by using the def statement.
- The return statement is what a function uses to return a value.
- This value can be of any type. If no return statement is encountered, Python's None value is returned.
- Function arguments can be entered either by position or by name (keyword).

Exceptions

```
1 class EmptyFileError(Exception):
2     pass
3 filenames = ["myfile1", "nonExistent", "emptyFile", "myfile2"]
4 for file in filenames:
5     try:
6         f = open(file, 'r')
7         line = f.readline()
8         if line == "":
9             f.close()
10            raise EmptyFileError("%s: is empty" % file)
11    except IOError as error:
12        print("%s: could not be opened: %s" % (file, error.strerror))
13    except EmptyFileError as error:
14        print(error)
15    else:
16        print("%s: %s" % (file, f.readline()))
17    finally:
18        print("Done processing", file)
```

- Exceptions (errors) can be caught and handled by using the try-except-else-finally compound statement.
- This statement can also catch and handle exceptions you define and raise yourself.
- Any exception that isn't caught causes the program to exit.

```
1 filename = "myfile.txt"
2 ✓ with open(filename, "r") as f:
3   ✓   for line in f:
4       print(f)
```

Context handling using the with keyword

- A more streamlined way of encapsulating the try-except-finally pattern is to use the with keyword and a context manager.
- One benefit of context managers is that they may (and usually do) have default cleanup actions defined, which always execute whether an exception occurs.

Module creation

- It's easy to create your own modules, which can be imported and used in the same way as Python's built-in library modules.
- The example in this listing is a simple module with one function that prompts the user to enter a filename and determines the number of times that words occur in this file.

```
1  """wo module. Contains function: words_occur()"""
2  # interface functions
3  def words_occur():
4      """words_occur() - count the occurrences of words in a file."""
5      # Prompt user for the name of the file to use.
6      file_name = input("Enter the name of the file: ")
7      # Open the file, read it and store its words in a list.
8      f = open(file_name, 'r')
9      word_list = f.read().split()
10     f.close()
11     # Count the number of occurrences of each word in the file.
12     occurs_dict = {}
13     for word in word_list:
14         # increment the occurrences count for this word
15         occurs_dict[word] = occurs_dict.get(word, 0) + 1
16     # Print out the results.
17     print("File %s has %d words (%d are unique)" \
18           % (file_name, len(word_list), len(occurs_dict)))
19     print(occurs_dict)
20
21 if __name__ == '__main__':
22     words_occur()
```

Object-oriented programming

- Python provides full support for OOP.
- Listing is an example that might be the start of a simple shapes module for a drawing program.
- Classes are defined by using the class keyword.
- The instance initializer method (constructor) for a class is always called `__init__`
- Methods, like functions, are defined by using the `def` keyword.

```
1  """sh module. Contains classes Shape, Square and Circle"""
2  class Shape:
3      """Shape class: has method move"""
4      def __init__(self, x, y):
5          self.x = x
6          self.y = y
7      def move(self, deltaX, deltaY):
8          self.x = self.x + deltaX
9          self.y = self.y + deltaY
10 class Square(Shape):
11     """Square Class: inherits from Shape"""
12     def __init__(self, side=1, x=0, y=0):
13         Shape.__init__(self, x, y)
14         self.side = side
15 class Circle(Shape):
16     """Circle Class: inherits from Shape and has method area"""
17     pi = 3.14159
18     def __init__(self, r=1, x=0, y=0):
19         Shape.__init__(self, x, y)
20         self.radius = r
21     def area(self):
22         """Circle area method: returns the area of the circle"""
23         return self.radius * self.radius * self.pi
24     def __str__(self):
25         return "Circle of radius %s at coordinates (%d, %d)"
26         % (self.radius, self.x, self.y)
```



Summary



This chapter is a rapid and very high-level overview of Python; the following chapters provide more detail. This chapter ends the book's overview of Python.



You may find it valuable to return to this chapter and work through the appropriate examples as a review after you read about the features covered in subsequent chapters.



If this chapter was mostly a review for you, or if you'd like to learn more about only a few features, feel free to jump around, using the index or table of contents.



You should have a solid understanding of the Python features in this chapter before skipping ahead to part 4.