# The Quick Python overview

### *This lab covers*

- [Surveying Python]
- [Using built-in data types]
- [Controlling program flow]
- [Creating modules]
- [Using object-oriented programming]

The purpose of this lab is to give you a basic feeling for the syntax, semantics, capabilities, and philosophy of the Python language. It has been designed to provide you an initial perspective or conceptual framework on which you'll be able to add details as you encounter them in the rest of the book.

### Built-in data types

Python has several built-in data types, from scalars such as numbers and Booleans to more complex structures such as lists, dictionaries, and files.

The following examples use integers:

```
>>> x = 5 + 2 - 3 * 2
>>> x
1
>>> 5 / 2
2.5                                  1
>>> 5 // 2
2                                    2
>>> 5 % 2
1
>>> 2 ** 8
256
>>> 1000000001 ** 3
1000000003000000003000000001           3
```

Division of integers with [/] ***1*** results in a float (new in Python 3.x), and division of integers with [//] ***2*** results in truncation. Note that integers are of unlimited size ***3***; they grow as large as you need them to, limited only by the memory available.

These examples work with floats, which are based on the doubles in C:

```
>>> x = 4.3 ** 2.4
>>> x
33.13784737771648
>>> 3.5e30 * 2.77e45
9.695e+75
>>> 1000000001.0 ** 3
1.000000003e+27
```

These examples use complex numbers:

```
>>> (3+2j) ** (2+3j)
(0.6817665190890336-2.1207457766159625j)
>>> x = (3+2j) * (4+9j)
```

```
>>> x                          1
(-6+35j)
>>> x.real
-6.0
>>> x.imag
35.0
```

Complex numbers consist of both a real element and an imaginary element, suffixed with [j]. In the preceding code, variable [x] is assigned to a complex number ***1***. You can obtain its "real" part by using the attribute notation [x.real] and obtain the "imaginary" part with [x.imag].

Several built-in functions can operate on numbers. There are also the library module [cmath] (which contains functions for complex numbers) and the library module [math] (which contains functions for the other three types):

```
>>> round(3.49)               1
3
>>> import math
>>> math.ceil(3.49)           2
4
```

Built-in functions are always available and are called by using a standard function-calling syntax. In the preceding code, [round] is called with a float as its input argument ***1***.

The functions in library modules are made available via the [import] statement. At ***2***, the [math] library module is imported, and its [ceil] function is called using attribute notation: *[module.function]*[()*[arguments]*[)].

The following examples use Booleans:

```
>>> x = False
>>> x
False
>>> not x
True
>>> y = True * 2           1
>>> y
2
```

Other than their representation as [True] and [False], Booleans behave like the numbers 1 (True) and 0 (False) ***1***.

**Lists**

Python has a powerful built-in list type:

```
[]
[1]
[1, 2, 3, 4, 5, 6, 7, 8, 12]
[1, "two", 3, 4.0, ["a", "b"], (5,6)]         1
```

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 ***1***.

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:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]                                          1
'first'                                           1
>>> x[2]                                          1
'third'
>>> x[-1]                                         2
'fourth'                                          2
>>> x[-2]                                         2
'third'                                           2
>>> x[1:-1]                                       2
['second', 'third']                               3
>>> x[0:3]                                        3
['first', 'second', 'third']                      3
>>> x[-2:-1]                                      3
['third']                                         3
>>> x[:3]                                         3
['first', 'second', 'third']                      4
>>> x[-2:]                                        4
['third', 'fourth']                               4
```

Index from the front ***1*** using positive indices (starting with 0 as the first element). Index from the back ***2*** using negative indices (starting with -1 as the last element). Obtain a slice using [[m:n]] ***3***, where [m] is the inclusive starting point and [n] is the exclusive ending point (see [table 3.1]). An [[:n]] slice ***4*** starts at its beginning, and an [[m:]] slice goes to a list's end.

You can use this notation to add, remove, and replace elements in a list or to obtain an element or a new list that's a slice from it:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[1] = "two"
>>> x[8:9] = []
>>> x
[1, 'two', 3, 4, 5, 6, 7, 8]
>>> x[5:7] = [6.0, 6.5, 7.0]                    1
>>> x
[1, 'two', 3, 4, 5, 6.0, 6.5, 7.0, 8]
>>> x[5:]
[6.0, 6.5, 7.0, 8]
```

The size of the list increases or decreases if the new slice is bigger or smaller than the slice it's replacing ***1***.

Some built-in functions ([len], [max], and [min]), some operators ([in], [+], and [*]), the [del] statement, and the list methods ([append], [count], [extend], [index], [insert], [pop], [remove], [reverse], and [sort]) operate on lists:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(x)
9
>>> [-1, 0] + x                                 1
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.reverse()                                 2
>>> x
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

The operators [+] and [*] each create a new list, leaving the original unchanged ***1***. A list's methods are called by using attribute notation on the list itself: [x.]*[method]* [()]*[arguments]*[)] ***2***.

Some of these operations repeat functionality that can be performed with slice notation, but they improve code readability.

**Tuples**

An important purpose of tuples is for use as keys for dictionaries. They're also more efficient to use when you don't need modifiability.

```
()
(1,)
(1, 2, 3, 4, 5, 6, 7, 8, 12)
```

A list can be converted to a tuple by using the built-in function [tuple]:

```
>>> x = [1, 2, 3, 4]
>>> tuple(x)
(1, 2, 3, 4)
```

Conversely, a tuple can be converted to a list by using the built-in function [list]:

```
>>> x = (1, 2, 3, 4)
>>> list(x)
[1, 2, 3, 4]
```

**Strings**

String processing is one of Python's strengths. There are many options for delimiting strings:

```
"A string in double quotes can contain 'single quote' characters."
'A string in single quotes can contain "double quote" characters.'
'''\tA string which starts with a tab; ends with a newline character.\n'''
"""This is a triple double quoted string, the only kind that can
    contain real newlines."""
```

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. The operators and functions that work with them return new strings derived from the original. The operators ([in], [+], and [*]) and built-in functions ([len], [max], and [min]) operate on strings as they do on lists and tuples. Index and slice notation works the same way for obtaining elements or slices but can't be used to add, remove, or replace elements.

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

```
>>> x = "live and     let \t   \tlive"
>>> x.split()
['live', 'and', 'let', 'live']
>>> x.replace("    let \t   \tlive", "enjoy life")
'live and enjoy life'
>>> import re                                    1
>>> regexpr = re.compile(r"[\t ]+")
```

```
>>> regexpr.sub(" ", x)
'live and let live'
```

The [re] module ***1*** provides regular-expression functionality. It provides more sophisticated pattern extraction and replacement capabilities than the [string] module.

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

```
>>> e = 2.718
>>> x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
>>> print("The constant e is:", e, "and the list x is:", x)          1
The constant e is: 2.718 and the list x is: [1, 'two', 3, 4.0,
['a', 'b'], (5, 6)]
>>> print("the value of %s is: %.2f" % ("e", e))                    2
the value of e is: 2.72
```

Objects are automatically converted to string representations for printing ***1***. The [%] operator ***2*** provides formatting capability similar to that of C's [sprintf].

**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.

```
>>> x = {1: "one", 2: "two"}
>>> x["first"] = "one"
>>> x[("Delorme", "Ryan", 1995)] = (1, 2, 3)
>>> list(x.keys())
['first', 2, 1, ('Delorme', 'Ryan', 1995)]
>>> x[1]
'one'
>>> x.get(1, "not available")
'one'
>>> x.get(4, "not available")
'not available'
```

Keys must be of an immutable type ***2***, including numbers, strings, and tuples. Values can be any kind of object, including mutable types such as lists and dictionaries. If you try to access the value of a key that isn't in the dictionary, a [KeyError] exception is raised. To avoid this error, the dictionary method [get] ***3*** optionally returns a user-definable value when a key isn't in a dictionary.

**Sets**

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:

```
>>> x = set([1, 2, 3, 1, 3, 5])        1
>>> x
{1, 2, 3, 5}                           2
>>> 1 in x                             3
True
>>> 4 in x                             3
```

```
False
>>>
```

You can create a set by using [set] on a sequence, like a list ***1***. When a sequence is made into a set, duplicates are removed ***2***. The [in] keyword ***3*** is used to check for membership of an object in a set.

#### File objects

A file is accessed through a Python file object:

```
>>> f = open("myfile", "w")                                    1
>>> f.write("First line with necessary newline character\n")
44
>>> f.write("Second line to write to the file\n")
33
>>> f.close()
>>> f = open("myfile", "r")                                    2
>>> line1 = f.readline()
>>> line2 = f.readline()
>>> f.close()
>>> print(line1, line2)
First line with necessary newline character
Second line to write to the file
>>> import os                                                  3
>>> print(os.getcwd())
c:\My Documents\test
>>> os.chdir(os.path.join("c:\\", "My Documents", "images"))   4
>>> filename = os.path.join("c:\\", "My Documents",
"test", "myfile")                                              5
>>> print(filename)
c:\My Documents\test\myfile
>>> f = open(filename, "r")
>>> print(f.readline())
First line with necessary newline character
>>> f.close()
```

The [open] statement ***1*** creates a file object. Here, the file myfile in the current working directory is being opened in write (["w"]) mode. After writing two lines to it and closing it ***2***, you open the same file again, this time in read (["r"]) mode. The [os] module ***3*** provides several functions for moving around the filesystem and working with the pathnames of files and directories. Here, you move to another directory ***4***. But by referring to the file by an absolute pathname ***5***, you're still able to access it.

Several other input/output capabilities are available. You can use the built-in [input] function to prompt and obtain a string from the user. The [sys] library module allows access to [stdin], [stdout], and [stderr]. The [struct] library module provides support for reading and writing files that were generated by, or are to be used by, C programs. The Pickle library module delivers data persistence through the ability to easily read and write the Python data types to and from files.

### Control flow structures

Python has a full range of structures to control code execution and program flow, including common branching and looping structures.

#### Boolean values and expressions

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].

You can create comparison expressions by using the comparison operators ([<], [<=], [==], [>], [>=], [!=], [is], [is not], [in], [not in]) and the logical operators ([and], [not], [or]), which all return [True] or [False].

**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:

```
x = 5
if x < 5:
    y = -1
    z = 5
elif x > 5:          1
    y = 1            1
    z = 11           1
else:
    y = 0            2
    z = 10           2
print(x, y, z)
```

The [elif] and [else] clauses are optional ***1***, and there can be any number of [elif] clauses. Python uses indentation to delimit blocks ***2***. No explicit delimiters, such as brackets or braces, are necessary. Each block consists of one or more statements separated by newlines. All these statements must be at the same level of indentation. The output in the example would be [5 0 10.]

**The while loop**

The [while] loop is executed as long as the condition (which here is [x > y]) is [True]:

```
u, v, x, y = 0, 0, 100, 30            1
while x > y:                          2
    u = u + y                         2
    x = x - y                         2
    if x < y + 2:                     2
        v = v + x                     2
        x = 0                         2
    else:                             2
        v = v + y + 2                 2
        x = x - y - 2                 2
print(u, v)
```

This is a shorthand notation. Here, [u] and [v] are assigned a value of 0, [x] is set to 100, and [y] obtains a value of 30 ***1***. This is the loop block ***2***. It's possible for a loop to contain [break] (which ends the loop) and [continue] statements (which abort the current iteration of the loop). The output would be [60 40.]

**The for loop**

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. The following loop finds the first occurrence of an integer that's divisible by 7:

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

[x] is sequentially assigned each value in the list ***1***. If [x] isn't an integer, the rest of this iteration is aborted by the [continue] statement ***2***. Flow control continues with [x] set to the next item from the list. After the first appropriate integer is found, the loop is ended by the [break] statement ***3***. The output would be

```
found an integer divisible by seven: 49
```

**Function definition**

Python provides flexible mechanisms for passing arguments to functions:

```
>>> def funct1(x, y, z):                          1
...     value = x + 2*y + z**2
...     if value > 0:
...         return x + 2*y + z**2                  2
...     else:
...         return 0
...
>>> u, v = 3, 4
>>> funct1(u, v, 2)
15
>>> funct1(u, z=v, y=2)                           3
23
>>> def funct2(x, y=1, z=1):                      4
...     return x + 2 * y + z ** 2
...
>>> funct2(3, z=4)
21
>>> def funct3(x, y=1, z=1, *tup):                 5
...     print((x, y, z) + tup)
...
>>> funct3(2)
(2, 1, 1)
>>> funct3(1, 2, 3, 4, 5, 6, 7, 8, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> def funct4(x, y=1, z=1, **kwargs):             6
...     print(x, y, z, kwargs)
>>> funct4(1, 2, m=5, n=9, z=3)
1 2 3 {'n': 9, 'm': 5}
```

Functions are defined by using the [def] statement ***1***. The [return] statement ***2*** 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). Here, [z] and [y] are entered by name ***3***. Function parameters can be defined with defaults that are used if a function call leaves them out ***4***. A special parameter can be defined that collects all extra positional arguments in a function call into a tuple ***5***. Likewise, a special parameter can be defined that collects all extra keyword arguments in a function call into a dictionary ***6***.

**Exceptions**

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. This listing shows basic exception handling.

**Listing: File exception.py**

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

**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. Python defines context managers for things like file access, and it's possible for the developer to define custom context managers. One benefit of context managers is that they may (and usually do) have default clean-up actions defined, which always execute whether or not an exception occurs.

This listing shows opening and reading a file by using [with] and a context manager.

**Listing: File with.py**

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

Here, [with] establishes a context manager which wraps the [open] function and the block that follows. In this case, the context manager's predefined clean-up action closes the file, even if an exception occurs, so as long as the expression in the first line executes without raising an exception, the file is always closed. That code is equivalent to this code:

```
filename = "myfile.txt"
try:
    f = open(filename, "r")
    for line in f:
        print(f)
except Exception as e:
    raise e
```

```
finally:
    f.close()
```

### 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.

**Listing: File wo.py**

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

Documentation strings, or *docstrings*, are standard ways of documenting modules, functions, methods, and classes ***1***. Comments are anything beginning with a # character ***2***. [read] returns a string containing all the characters in a file ***3***, and [split] returns a list of the words of a string "split out" based on whitespace. You can use a [\] to break a long statement across multiple lines ***4***. This [if] statement allows the program to be run as a script by typing [python wo.py] at a command line ***5***.

If you place a file in one of the directories on the module search path, which can be found in [sys.path], it can be imported like any of the built-in library modules by using the [import] statement:

```
>>> import wo
>>> wo.words_occur()            1
```

This function is called ***1*** by using the same attribute syntax used for library module functions.

Note that if you change the file wo.py on disk, [import] won't bring your changes into the same interactive session. You use the [reload] function from the [imp] library in this situation:

```
>>> import imp
>>> imp.reload(wo)
<module 'wo'>
```

For larger projects, there is a generalization of the module concept called *packages*, which allows you to easily group modules in a directory or directory subtree and then import and hierarchically refer to them by using a *[package.subpackage.module]* syntax. This entails little more than creating a possibly empty initialization file for each package or subpackage.

### Object-oriented programming

**Listing: File sh.py**

```
"""sh module. Contains classes Shape, Square and Circle"""
class Shape:                                                    1
    """Shape class: has method move"""
    def __init__(self, x, y):                                   2
        self.x = x                                              3
        self.y = y                                              3
    def move(self, deltaX, deltaY):                             4
        self.x = self.x + deltaX
        self.y = self.y + deltaY
class Square(Shape):
    """Square Class:inherits from Shape"""
    def __init__(self, side=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.side = side
class Circle(Shape):                                            5
    """Circle Class: inherits from Shape and has method area"""
    pi = 3.14159                                                6
    def __init__(self, r=1, x=0, y=0):
        Shape.__init__(self, x, y)                              7
        self.radius = r
    def area(self):
        """Circle area method: returns the area of the circle."""
        return self.radius * self.radius * self.pi
    def __str__(self):                                          8
        return "Circle of radius %s at coordinates (%d, %d)"\
               % (self.radius, self.x, self.y)
```

Classes are defined by using the [class] keyword ***1***. The instance initializer method (constructor) for a class is always called [__init__] ***2***. Instance variables [x] and [y] are created and initialized here ***3***. Methods, like functions, are defined by using the [def] keyword ***4***. The first argument of any method is by convention called [self]. When the method is invoked, [self] is set to the instance that invoked the method. Class [Circle] inherits from class [Shape] ***5*** and is similar to, but not exactly like, a standard class variable ***6***. A class must, in its initializer, explicitly call the initializer of its base class ***7***. The [__str__] method is used by the [print] function ***8***. Other special method attributes permit operator overloading or are employed by built-in methods such as the length ([len]) function.

Importing this file makes these classes available:

```
>>> import sh
>>> c1 = sh.Circle()                            1
>>> c2 = sh.Circle(5, 15, 20)
>>> print(c1)
Circle of radius 1 at coordinates (0, 0)
>>> print(c2)                                   2
Circle of radius 5 at coordinates (15, 20)
>>> c2.area()
```

```
78.539749999999998
>>> c2.move(5,6)                                3
>>> print(c2)
Circle of radius 5 at coordinates (20, 26)
```

The initializer is implicitly called, and a circle instance is created ***1***. The [print] function implicitly uses the special [__str__] method ***2***. Here, you see that the [move] method of [Circle]'s parent class [Shape] is available ***3***. A method is called by using attribute syntax on the object instance: [object.method()]. The first ([self]) parameter is set implicitly.

### Summary

- [This lab is a rapid and very high-level overview of Python; the following chapters provide more detail. This lab ends the book's overview of Python.]
- [You may find it valuable to return to this lab and work through the appropriate examples as a review after you read about the features covered in subsequent chapters.]
- [If this lab 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.]