SMM692
Introduction to Programming in Python

Contents

List of Figures

List of Tables

Preface

...

Chapter 1

Organization of the Module

...

Chapter 2

Getting Started with Python

2.1 Installing Python

...

2.2 How Python Runs Programs

...

2.3 How We Run Python Programs

...

2.4 Managing Python Environments

...

Chapter 3

Python Language Fundamentals

3.1 Variables

...

3.2 Objects

...

3.3 References

...

Chapter 4

Python Objects

In this chapter, we pursue the following learning objectives:

- ....

What is a Python object?

In essence, Python objects are pieces of data. Mark Lutz, the author of the popular book Learning Python[1], points out

> *in Python we do things with stuff. "Things" take the form of operations like addition and concatenation, and "stuff" refers to the objects on which we perform those operations.*

Built-in and ad-hoc objects

In Python, there are two families of objects: built-in objects provided by the Python language itself and ad-hoc objects — called classes — we can create to accomplish specific goals.

Why do built-in Python objects matter?

Typically, we do not need to create ad-hoc objects. Python provides us with diverse built-in objects that make our job easier:

- built-in objects make coding efficient and easy. For example, using the string object, we can represent and manipulate a piece of text — e.g., a newspaper article — without loading any module
- built-in objects are flexible. For example, we can deploy built-in objects to create a class
- built-in objects have been created and refined over time by a large community of expert developers. Hence, they are often more efficient than ad-hoc objects (unless the creator of the ad-hoc object really knows her business!)

The core built-in Python objects

Table 4.1 illustrates the types of built-in Python objects. For example, Numbers and strings objects are used to represent numeric and textual data respectively. Lists and dictionaries are — likely as not — the two most popular data structures in Python. Lists are ordered collections of other objects such (any type!!). Dictionaries are pairs of keys (e.g., a product identifier) and objects (e.g., the price of the product). No worries: we will go through each built-in type in the following sections of this document. Caveat: in the interest of logical coherence, the various built-in types will not be presented in the order adopted Table 4.1.

Table 4.1
Built-In Objects in Python

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	‘spam’, “Bob’s”, b‘a\x01c’, u‘sp\xc4m’
Lists	[1, [2, ‘three’], 4.5], list(range(10))
Dictionaries	{‘food’: ‘spam’, ‘taste’: ‘yum’}, dict(hours=10)
Tuples	(1, ‘spam’, 4, ‘U’), tuple(‘spam’), namedtuple
Files	open(‘eggs.txt’), open(r‘C:\ham.bin’, ‘wb’)
Sets	set(‘abc’), {‘a’, ‘b’, ‘c’}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation types	Compiled code, stack tracebacks

4.1 Number Type Fundamentals

Types of ‘number’ objects

Example 4.1, “Doing stuff with numbers,” highlights the two most popular ‘number’ instances in Python: integers and floating-point numbers. Integers are whole numbers such as 0, 4, or -12. Floating-point numbers are the representation of real numbers such as 0.5, 3.1415, or -1.6e-19. However, floating points in Python do not have — in general — the same value as the real number they represent.[2] It is worth noticing that any single number with a period ‘.’ is considered a floating point in Python. Also, Example 4.1 shows that the multiplication of an integer by a floating point yields a floating point. That happens because Python first converts operands up to the type of the most complicated operand.

Example 4.1 — doing 'stuff' with numbers

```
# integer addition
>>> 1 + 1
2

# floating-point multiplication
>>> 10 * 0.5
5.0

# 3 to the power 100
>>> 3 ** 100
515377520732011331036461129765621272702107522001

```

Besides integers and floating points

Besides integers and floating points numbers, Python includes fixed-precision, rational numbers, Booleans, and sets instances — see Table 4.2.

TABLE 4.2
Number Type Objects in Python

Literal	Interpretation
1234, -24, 0, 99999999999999	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals in 3.X
0177, 0o177, 0x9ff, 0b101010	Octal, octal, hex, and binary literals in 2.X
3+4j, 3.0+4.0j, 3J	Complex number literals
set('spam'), {1, 2, 3, 4}	Sets: 2.X and 3.X construction forms
Decimal('1.0'), Fraction(1, 3)	Decimal and fraction extension types
bool(X), True, False	Boolean type and constants

Basic arithmetic operations in Python

Numbers in Python support the usual mathematical operations:

- '+' addition
- '-' → subtraction
- '*' → multiplication
- '/' → floating point division
- '//' → integer division
- '%' → modulus (remainder)
- '**' → exponentiation

To use these operations, it is sufficient to launch a Python or IPython session without any modules loaded (see Example 4.1).

Advanced mathematical operations

Besides the mathematical operations shown above, there are many modules shipped with Python that carry out advanced/specific numerical analysis. For example, the `math` module provides access to the mathematical functions defined by the C standard.[3] Table 4.3 reports a sample of these functions. To use them `math`, we have to import the module as shown in Example 4.2. Another popular module shipped with Python is `random`, implementing pseudo-random number generators for various distributions (see the lower section of Example 2).

TABLE 4.3
A Sample of Functions Provided by the `math` Module

Function name	Expression
`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	e^x
`math.log(x)`	lnx
`math.log(x, b)`	$log_b(x)$
`math.log10(x)`	$log_{10}(x)$
`math.sin(x)`	$sin(x)$
`math.cos(x)`	$cos(x)$
`math.tan(x)`	$tan(x)$
`math.asin(x)`	$arcsin(x)$
`math.acos(x)`	$arccos(x)$
`math.atan(x)`	$arctan(x)$
`math.sinh(x)`	$sinh(x)$
`math.cosh(x)`	$cosh(x)$
`math.tanh(x)`	$tanh(x)$
`math.asinh(x)`	$arsinh(x)$
`math.acosh(x)`	$arcosh(x)$
`math.atanh(x)`	$artanh(x)$
`math.hypot(x, y)`	The Euclidean norm, $\sqrt{x^2 + y^2}$
`math.factorial(x)`	$x!$
`math.erf(x)`	The error function at x
`math.gamma(x)`	The gamma function at x, $\omega(x)$
`math.degrees(x)`	Converts x from radians to degrees
`math.radians(x)`	Converts x from degrees to radians

Example 4.2 — advanced mathematical operations with the modules shipped with Python

```
1  # import the math module
2  >>> import math
3
4  # base-y log of x
5  >>> math.log(12, 8)
6  1.1949875002403856
7
8  # base-10 log of x
9  >>> math.log10(12)
10 1.0791812460476249
11
12 # import the random module
13 >>> import random
14
15 # a draw from a normal distribution with mean = 0 and standard deviation = 1
16 >>> random.normalvariate(0, 1)
17 -0.136017752991189
18
19 # trigonometric functions
20 >>> math.cos(0)
21 1.0
22
23 >>> math.sin(0)
24 0.0
25
26 >>> math.tan(0)
27 0.0
28
29 # an expression containing a factorial product
30 >>> math.factorial(4) - 4 * 3 * 2 * 1
31 0
```

Operator precedence

As shown in Example 4.2, line 30, Python expressions can string together multiple operators. So, how does Python know which operation to perform first? The answer to this question lies in operator precedence. When you write an expression with more than one operator, Python groups its parts according to what are called precedence rules,[4] and this grouping determines the order in which the expression's parts are computed. Table 4.4 reports the precedence hierarchy concerning the most common operators. Note that operators lower in the table have higher precedence. Parentheses can be used to create sub-expressions that override operator precedence rules.

Table 4.4
Operator Precedence Hierarchy
(Ascending Order)

Operator	Description
`x + y`	Addition, concatenation
`x - y`	Subtraction, set difference
`x * y`	Multiplication, repetition
`x % y`	Remainder, format;
`x / y`, `x // y`	Division: true and floor
`-x`, `+x`	Negation, identity
`~x`	Bitwise NOT (inversion)
`x ** y`	Power (exponentiation)

Technical and scientific computation with Python

Python is at the center of a rich ecosystem of modules for technical and scientific computation. In the following chapter, the attention will revolve around two of the most prominent modules: NumPy and SciPy. In a nutshell, NumPy offers the infrastructure for the efficient manipulation of (potentially massive) data structures, while SciPy implements many algorithms across the fields of statistics, linear algebra, optimization, calculus, signal processing, image processing, and others. Another core module in the technical and scientific domain is SimPy, a library for symbolic mathematics. Note that none of these three modules are shipped with Python and should be installed with the package manager of your choice (e.g., `conda`).

Variables and Basic Expressions

Variables are simply names—created by you or Python—that are used to keep track of information in your program. In Python:

- Variables are created when they are first assigned values
- Variables are replaced with their values when used in expressions
- Variables must be assigned before they can be used in expressions
- Variables refer to objects and are never declared ahead of time

As Example 4.3 shows, the assignment of `x = 2` causes the variable `x` to come into existence 'automatically.' From that point, we can use the variables in the context of expressions such as the ones displayed in lines 8, 12, 16, and 20, or to create new variables like in line 24.

Example 4.3 — expressions involving arithmetic operations

```
1
2 # let us assign the variables 'x' and 'y' to two number objects
3 >>> x = 2
4
5 >>> y = 4.0
6
7 # subtracting an integer from variable 'x'
8 >>> x - 1
9 1
10
11 # dividing the variable 'y' by an integer
12 >>> y / 73
13 0.0547945205479452
14
15 # integer-dividing the variable 'y' by an integer
16 >>> y // 73
17 0.0
18
19 # getting a linear combination of 'x' and 'y'
20 >>> 3 * x - 5 * y
21 -14.0
22
23 # assigning the variable 'z' to the linear combination of 'x' and 'y'
24 >>> z = 3 * x - 5 * y
```

Displaying number objects

Example 4.3 includes some expressions whose result is not passed to a new variable (e.g., lines 8, 12, 16, 20). In those cases, the IPython session displays the outcome of the expression 'as is' (e.g., 0.0547945205479452). However, a number with more than three or four decimals may not suit the table or report we have to prepare. Python has powerful string formatting capabilities to display number objects in a readable and nice manner. Table 4.5 illustrates various number formatting options with concrete cases. Format strings contain 'replacement fields' surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. Example 4.4 presents a fully-fledged number formatting case. First, we assign the variable `a` to a floating-point number (line 2). Then, we pass the formatting option {`:.2f`} over the variable `a` using the Python built-in function `format`.

Table 4.5
Number Formatting Options in Python

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	Format float 2 decimal places
3.1415926	{:+.2f}	+3.14	Format float 2 decimal places with sign
-1	{:+.2f}	-1.00	Format float 2 decimal places with sign
2.71828	{:.0f}	3	Format float with no decimal places
5	{:0>2d}	05	Pad number with zeros (left padding, width 2)
5	{:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1,000,000	Number format with comma separator
0.25	{:.2%}	25.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
13	{:10d}	13	Right aligned (default, width 10)
13	{:<10d}	13	Left aligned (width 10)
13	{:^10d}	13	Center aligned (width 10)

Example 4.4 — number formatting in Python

```
# assign the variable 'a' to a floating-point number
>>> a = 0.67544908755

# displaying 'a' with the first two decimals only
>>> "{:.2f}".format(a)
"0.68"

# displaying 'a' with the first three decimals only
>>> "{:.3f}".format(a)
"0.675"
```

How do I compare number objects?

Comparisons are used frequently to create control flows, a topic we will discuss later in this chapter. Normal comparisons in Python regard two number objects and return a Boolean result. Chained comparisons concern three or more objects, and, like normal comparisons, yield a Boolean result. Example 4.5 provides a sample of normal comparisons (between lines 1 and 15) and chained comparisons (between lines 21 and 30). As evident in the example, comparisons can regard both numbers and variables assigned to numbers. Chained comparisons can take the form of a range test (see line 21), a joined, 'AND' test of the truth of multiple expressions (see line 25) or a disjoined, 'OR' test of the truth of multiple expressions (see line 29).

Example 4.5 — comparing numeric objects

```
# less than
>>> 3 < 2
False

# greater than or equal
>>> 1 <= 2
True

# equal
>>> 2 == 2
True

# not equal
>>> 4 != 4
False

# range test
>>> x = 3
>>> y = 5
>>> z = 4
>>> x < y < z
False

# joined test
>>> x < y and y > z
True

# disjoined test
>>> x < y or y < z
True
```

4.2 String Type Fundamentals

What is a string?

A Python string is a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions. Strictly speaking, strings are *immutable sequences* of one-character strings; other, more general sequence types include lists and tuples, covered later.

How do we use strings?

Strings are used to record words, contents of text files loaded into memory, Internet addresses, Python source code, and so on. Strings can also be used to hold the raw bytes used for media files and network transfers, and both the encoded and decoded forms of non-ASCII Unicode text used in internationalized programs.

Is `abc` a Python string?

Nope. Python strings are enclosed in single quotes ('…') *or* double quotes ("…") with the same result. Hence, "`abc`" can be Python string, while `abc` cannot. `abc` can be a variable name, though.

String indexing and slicing

The fact that strings are immutable sequences affects how we manipulate textual data in Python. In Example 4.6, we fetch the individual elements of `S`, a variable assigned to "`Python 3.X`." As per the built-in function `len`, `S` contains six unitary strings. That means that each element in `S` is associated with a position in the numerical progression $\{0, 1, 2, 3, 4, 5\}$. Now, you may be surprised to see the first element of the list is 0 instead of 1. The reason is that Python is a zero-based indexed programming language: the first element of a series has index 0, while the last element has index `len(obj) - 1`. Fetching the individual elements of a string, such as `S`, requires passing the desired index between brackets, as shown in line 9 (where we get the first unitary string, namely, "`P`".), line 13 (where we get the last unitary string, namely, "`X`"), and line 21 (where we get the unitary string with index 3, i.e., the fourth unitary string appearing in `S`, "`h`"). Note that line 17 is an alternative indexing strategy to the one presented in line 13: it is possible to retrieve the last unitary string by counting 'backward'; that is, getting the first element starting from the right-hand side of the string, which equates to index `-1`. In lines 26 and 30, we exploit the indices of `S` to retrieve multiple unitary strings in a row. What we pass among brackets is not a single index. Instead, we specify a range of indices `i:j`. It is wort noticing that, in Python, the element associated with the lower bound index `i` is returned, whereas the element associated with the upper bound index `j` is not. In line 26, we fetch the unitary strings between index `2` — equating to third unitary string of `S` — and index `5` excluded — namely, the fifth unitary string of `S`. In line 30, we adopt the 'backward' approach to retrieve the unitary string with index `-3` — the third string counting from the right-hand side of `S` — as well as any other unitary strings following index `-3`. To do that, we leave the upper bound index blank.

Example 4.6 — Python strings as sequences

```
# let us assign the string "Python 3.X" to the variable S
>>> S = "Python 3.X"

# check the length of S
>>> len(S)
6

# access the first unitary string in the sequence behind S
>>> S[0]
"P"

# access the last unitary string in the sequence behind S
>>> S[len(S)-1]
"X"

# or, equivalently
>>> S[-1]
"X"

# access the i-th, e.g., 3rd, unitary string in the sequence behind S
>>> S[3]
"h"

# access the unitary strings between the i-th and j-th positions in the
# sequence behind S
>>> S[2:5]
"tho"

# access the unitary strings following the i-th position in the sequence
# behind S
>>> S[-3:]
"3.X"
```

Common string literals and operators

Example 4.6 deals with string indexing and slicing, two of the many operations we can carry out on strings. Table 4.6 reports a sample of common string literals and operators. The first two lines of Table 4.6 remind us that single and double quotes are equivalent when it comes to assigning a variable to a string object. However, we must refrain from mixing and matching single and double quotes. In other words, a string object requires the leading and trailing quotes are of the same type (i.e., double-double or single-single). In the interest of consistency, it is a good idea to make a policy choice, such as "in my Python code, I use double quotes only", and to stick with that throughout the various lines of the script. I prefer using double quotes because the single quote symbol is relatively popular in natural language (consider for example the Saxon genitive). As shown in the third line of Table 4.6 the single quote is treated as a unitary string insofar as double quotes are used to delimit the string object. Should the string object be delimited by single quotes, we should tell Python not to treat the single quote symbol after `m` as a Python special character but as a unitary string. To do that, we use the escape symbol \ as shown in the fourth line of Table 4.6. Table A.1 provides several escaping examples.

TABLE 4.6
Sample of String Literals and Operators

Literal/operation	Interpretation
`S = ""`	Empty string
`S = ‘’`	Single quotes, same as double quotes
`S = "spam’s"`	Single quote as a string
`S = ‘spam\’s’`	Escape symbol
`length(S)`	Length
`S[i]`	Index
`S[i:j]`	Slice
`S1 + S2`	Concatenate
`S * 3`	Repeat S n times (e.g., three times)
`"text".join(strlist)`	Join multiple strings on a character (e.g., “text”)
`"{}".format()`	String formatting expression
`S.strip()`	Remove white spaces
`S.replace("pa", "xx")`	Replacement
`S.split(",")`	Split on a character (e.g., “,”)
`S.lower()`	Case conversion — to lower case
`S.upper()`	Case conversion — to upper case
`S.find("text")`	Search substring (e.g., ”text”)
`S.isdigit()`	Test if the string is a digit
`S.endswith("spam")`	End test
`S.startswith("spam")`	Start test
`S = """...multiline..."""`	Triple-quoted block strings

String manipulation tasks

Example 4.7 presents a sample of miscellaneous string manipulation tasks. In lines 6 and 9, we check the length of the variables `S1` and `S2`. In line 13, we display five repetitions of `S1`. In line 17, we use the algebraic operator “+” to concatenate `S1` and `S2`. In line 21, we expand on the previous input by separating `S1` and `S2` by a white-space. In line 25, we carry out the same task as line 17 — however, we rely on the built-in `join` function to join `S1` and `S2` with whitespace. The argument taken by `join` is a Python `list`, the subject of paragraph 5.3. In line 29, `S1` and `S2` are joined with a custom string object, namely, `" Vs.  "`. Finally, in line 33, we use the built-in `format` function (see also Example 4.4) to display a string object including `S1` and `S2`. For a comprehensive list of string methods, see Table A.2.

Example 4.7 — miscellaneous string manipulation tasks

```
# let us assign S1 and S2 to two strings
>>> S1 = "Python 3.X"
>>> S2 = "Julia"

# check the length of S1 and S2
>>> len(S1)
10

>>> len(S2)
5

# display the S1 repeated five times
>>> S1 * 5
"Python 3.XPython 3.XPython 3.XPython 3.XPython 3.X"

# display the concatenation of S1 and S2
>>> S1 + S2
"Python 3.XJulia"

# display the concatenation of S1, whitespace, and S2
>>> S1 + " " + S2
"Python 3.X Julia"

# display the outcome of joining S1 and S2 with a whitespace
>>> " ".join([S1, S2])
"Python 3.X Julia"

# display the outcome of joining S1 and S2 with an arbitrary string object
>>> " Vs. ".join([S1, S2])
"Python 3.X Vs. Julia"

# string formatting
>>> "Both {} and {} have outstanding ML modules".format(S1, S2)
"Both Python 3.X and Julia have outstanding ML modules"
```

String editing

Example 4.8 illustrates some string editing tasks. In line 5, we use lstrip — a variation of the built-in function strip — that returns a copy of the string with leading characters removed. In line 9, we use the built-in replace to return a copy of the string with all occurrences of substring `old` (first argument taken by the function) replaced by `new` (second argument taken by the function). Finally, in line 17, we use the built-in function `lower` to return a copy of the string with all the cased characters converted to lowercase.

Example 4.8 — miscellaneous string editing tasks

```
1 # let us assign S to a string object
2 >>> S = "Both Python 3.X and Julia have outstanding ML modules"
3
4 # strip target leading characters
5 >>> S.lstrip("Both ")
6 "Python 3.X and Julia have outstanding ML modules"
7
8 # replace target characters
9 >>> S.replace("Python 3.X", "R")
10 "Both R and Julia have outstanding ML modules"
11
12 # split string on target characters
13 >>> S.split(" and ")
14 ["Both Python 3.X", "Julia have outstanding ML modules"]
15
16 # make the string lower case
17 >>> S.lower()
18 "both python 3.x and julia have outstanding ml modules"
```

Testing and searching strings

Example 4.9 presents a series of string test and search tasks. The built-in function `find` (see lines 5 and 9) returns the lowest index in the string where substring sub is found within the slice `S[start:end]` or -1 if substring is not found. The built-in function `isdigit` return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Finally, the built-in function endswith returns `True` if the string ends with the specified suffix, otherwise returns `False`.

Example 4.9 — miscellaneous string test and search tasks

```
1 # let us assign S to a string object
2 >>> S = "The first version of Python was released in 1991"
3
4 # search for "Python" in S
5 >>> S.find("Python")
6 21
7
8 # search for "Julia" in S
9 >>> S.find("Julia")
10 -1
11
12 # slice the string the get Python's release year information
13 >>> SS = S[-4:]
14
15 # display SS
16 >>> SS
17 "1991"
18
19 # test if all characters in SS are digits
20 >>> SS.isdigit()
21 True
22
23 # test if all characters in SS are digits
24 >>> SS.isdigit()
25 True
26
27 # test if all characters in SS are digits
28 >>> SS.isdigit()
29 True
30
31 # test if S ends with "1991" / SS
32 >>> S.endswith(SS)
33 True
```

Multiline string printing

In the previous examples, we came across the built-in function `print`. Such a function can be used to print both number- and string-type objects. Sometimes, what we want to print fits into a single line. In other circumstances, we are interested in visualizing rich data, which can span multiple lines. Example 4.9 how to print objects across multiple lines with the triple-quoted block string (see line). As evident from the Python code included in lines 8-13, any line between triple-quotes is considered part of the same string object.

Example 4.9 — multiline string printing

```
1  # single-line print
2  >>> print("Hello world!")
3  Hello world!
4  
5  # multi-line print
6  >>> print(
7  ... """
8  ... =======================================================
9  ... COL A        | COL B       | ...         | COL K
10 ... -------------------------------------------------------
11 ... Sheldon      | Cooper      | ...         | bazinga.com
12 ... -------------------------------------------------------
13 ... NOTES: this table has fake data
14 ... """
15 ... )
16 
17 =======================================================
18 COL A        | COL B       | ...         | COL K
19 -------------------------------------------------------
20 Sheldon      | Cooper      | ...         | bazinga.com
21 -------------------------------------------------------
22 NOTES: this table has fake data
```

4.3 List and Dictionaries

What is a list?

A Python `list` is an *ordered*, *mutable* array of objects. A list is constructed by specifying the objects, separated by commas, between square brackets, `[]`.

Why do we use lists?

Lists are just places to collect other objects so you can treat them as groups.

What type of objects can we include in a list?

Lists can contain any sort of object: numbers, strings, and even other lists. See Example 4.10.

Example 4.10 — sample lists with different items

```
# an empty list
>>> L = []

# a list with an integer, a float, and a string
>>> L = [2, -3.56, "XyZ"]

# a list with an integer and a list
>>> L = [4, ["abc", 8.98]]
```

List indexing

We can retrieve one or more list component objects via indexing. That is possible because list items are ordered by their position (similarly to strings). Since Python is a zero-based indexed programming language, to fetch the first item of a list we have to call the index `0` (see Example 4.11, line 5). A list nested in another list can be fetched by using multiple indices (line 13). The first index refers to the outer list, while any subsequent index refers to an inner list. In our case, we have two indices, one for each list; that is, `L` and its sub-list `["abc", 8.98]`.

Example 4.11 — list indexing and slicing

```
1  # the list
2  >>> L = [4, ["abc", 8.98]]
3
4  # get the first item of L
5  >>> L[0]
6  4
7
8   # get the second element of L
9  >>> L[1]
10 ["abc", 8.98]
11
12 # get the first item of L's second item
13 >>> L[1][0]
14 "abc"
```

List mutability

Lists are mutable objects, which may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more. Example 4.12 illustrates some snippets to change a list's items. In the first part of the example, we change the items using indexing (line 5) and slicing (line 10). In the second half, we use Python's `del` statement to delete the items using indexing (line 15) and slicing (line 20).

Example 4.12 — changing and deleting list items in place

```
# the list
>>> L = ["Leonard", "Penny", "Sheldon"]

# change the second item of L via indexing
>>> L[1] = "Raj"
>>> print(L)
["Leonard", "Raj", "Sheldon"]

# change multiple items of L via slicing
>>> L[0:2] = ["Amy", "Howard"]
>>> print(L)
["Amy", "Howard", "Sheldon"]

# delete the first item of L via indexing and using the 'del' statement
>>> del L[0]
>>> print(L)
["Howard", "Sheldon"]

# delete multiple items of L via slicing and using the 'del' statement
>>> del L[0:2]
>>> print(L)
[]
```

List manipulation with built-in methods

Python offers many methods to manipulate and test list objects. Table 4.7 reports some of the most popular methods along with synopses. The first three methods — `.append()`, `.insert()`, and `.extend()` — expand an existing list. The fourth method, `.index` test for the presence of an item in an existing list. Note this method raises a `ValueError` if there is no such item.The remaining methods produce in-place changes in an existing list's items.

TABLE 4.7
Popular list methods

Method	Synopsis
`L.append(X)`	Append an item to an existing list
`L.insert(i, X)`	Append an item to an exisintg list in position i
`L.extend([X0, X1, X2])`	Extend an existing list with the items from another list
`L.index(X)`	Get the index of the first instance of the argument in an existing list
`L.count(X)`	Get the cardinality of an item in an existing list
`L.sort()`	Sort the items in an existing list
`L.reverse()`	Reverse the order of the items in an existing list
`L.copy()`	Get a copy of an existing list
`L.pop(i)`	Remove the item at the given position in the list, and return it
`L.remove(X)`	Remove the first instance of an item in an existing list
`L.clear()`	Remove all items in an existing list

Expanding an existing list

Both the `.append()` and `.extend()` methods can be used to expand an existing list as per Example 4.13. However, they accomplish different goals and should not be confused: `.append()` adds a new item (of any type) to the end of the list (see line 6); `.extend()` extend the list by appending all the items from another iterable (e.g., another list, see line 11).

Example 4.13 — methods for expanding an existing list

```
1 # create two lists
2 >>> L1 = ["Leonard", "Penny", "Sheldon"]
3 >>> L2 = ["Howard", "Raj", "Amy", "Bernadette"]
4
5 # expand an existing list with .append()
6 >>> L2.append("Priya")
7 >>> print(L2)
8 ["Howard", "Raj", "Amy", "Bernadette", "Priya"]
9
10 # concatenate L1 and L2 with .extend()
11 >>> L1.extend(L2)
12 >>> print(L1)
13 ["Leonard", "Penny", "Sheldon", "Howard", "Raj", "Amy", "Bernadette", "Priya"]
```

In-place change of an existing list's items

One of the most common list manipulation task consists of changing the order of an item's list. As shown in Example 4.14, it is possible to use the `.reverse()` method to reverse the elements of the list in place (see line 5), while sorting an item's list can be carried out with the `.sort()` method.

Example 4.14 — methods for changing list items in place

```
1 # create a list
2 >>> L = ["Howard", "Raj", "Amy", "Bernadette", "Priya"]
3
4 # reverse the list's item positions
5 >>> L.reverse()
6 >>> print(L)
7 ["Priya", "Bernadette", "Amy", "Raj", "Howard"]
8
9 # sort the list's items
10 >>> L.sort()
11 >>> print(L)
12 ["Amy", "Bernadette", "Howard", "Priya", "Raj"]
```

4.4 Dictionaries

What is a dictionary?

Along with lists, dictionaries are one of the most flexible built-in data types in Python. If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by *key*, instead of by *positional offset*.

Why do we use dictionaries?

Dictionaries take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions.

What type of objects can we include in a dictionary?

Like lists, dictionaries can contain objects of any type, and they support nesting to any depth(they can contain lists, other dictionaries, and so on). Each key can have just one associated value, but that value can be a collection of multiple objects if needed, and a given value can be stored under any number of keys.

How do we create a dictionary?

Example 4.15 shows two different ways to create a dictionary. A dictionary can be created by including key-value pairs among braces (see line 2). In the example, there are three keys, associated with Marvel characters, and as many values, which can be thought as the characters' position in an ideal power rank. A column separates a key and its associated value. The second way to create a dictionary is based on Python's builtin `dict`, mapping key onto values, and `zip`, which iterates over two elements in parallel. Specifically, `zip` creates the one-to-one correspondence between keys (characters) and values (characaters' power) that is passed as the argument of `dict`. We will analyze the topic of iterations extensively in sections 4.10 and 4.11.

Example 4.15 — initializing a new dictionary object

```
1 # method 1
2 >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4 # method 2
5 >>> CHARACTERS = ["Captain Marvel", "Living Tribunal", "One-Above-All"]
6 >>> RANK = [3, 2, 1]
7 >>> D = dict(zip(CHARACTERS, RANK))
8 >>> print(D)
9 {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
```

Accessing a dictionary's values

Dictionaries' items cannot be accessed via positional offsets — like lists. Instead, we fetch the individual items by using the dictionary keys as shown in Example 4.16 (see line 5). The reference key is passed among brackets. When the dictionary at hand contains nested dictionaries (see line 9), it is possible to concatenate multiple queries, namely, sequences of keys between brackets (see line 21).

Example 4.16 — fetching dictionary items

```
1  # the dictionary
2  >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4  # let's fetch Captain Marvel's position in the Marvel characters' power rank
5  >>> D["Captain Marvel"]
6  3
7
8  # a dictionary of dictionaries
9  >>> D = {
10         "Dr. Strange": {
11                 "first_appearance": 1963,
12                 "created_by": "Lee & Ditko"
13                 },
14         "Iron Man": {
15                 "first_appearance": 1963,
16                 "created_by": "Lee, Lieber, Heck & Kirby"
17                 },
18     }
19
20 # let us fetch the creator of Dr. Strange
21 >>> D["Dr. Strange"]["created_by"]
22 "Lee & Ditko"
```

Are dictionaries mutable?

Dictionsries, like lists, are mutable. Thus, we can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry. The del statement works here, too; it deletes the entry associated with the key specified as an index (see Example 4.17).

Example 4.17 — dictionary mutability examples

```
# the dictionary
>>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}

# let us change the power rank for Captain Marvel
>>> D["Captain Marvel"] = 12
>>> print(D)
{"Captain Marvel": 12, "Living Tribunal": 2, "One-Above-All": 1}

# let us eliminate the character Living Tribunal
>>> del D["Living Tribunal"]
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1}

# let us add a further character
>>> D["Wanda Maximoff"] = 4
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1, "Wanda Maximoff": 4}
```

Dictionary manipulation with built-in methods

Like for lists, Python offers many `methods to manipulate dictionary objects`. Table 4.8 reports some of the most common methods along with synopses. The first three methods, `.keys()` `.values()` `.items()`, get the constitutive elements of dictionaries: keys, values, and key-value pairs respectively. The fourth method, `.get(key, default?)` gets the value for a specific key. The fifth method, `.update()`, updates the value for a specific key. Like `.update()`, `.popitem()`, `.pop()`, and `d.clear()` alter the information of a dictionary in place. The first removes the value of a certain key; the second removes the item (a key-value pair) for a certain key; the latter delete all dictionary items. Finally, `.copy()` creates a shallow copy of an existing dictionary.

Table 4.8
Popular dictionary methods

Method	Synopsis
`D.keys()`	Get all dictionary keys
`D.values()`	Get all dictionary values
`D.items()`	Get all dictionary key-value pairs as tuples
`D.get(key, default?)`	Query a dictionary element by key
`D.update(D2)`	Update a dictionary key's value
`D.popitem()`	Remove the value corresponding to a certain key
`D.pop(key, default?)`	Remove the item at the given position in the list,
`D.clear()`	Delete all dictionary items
`D.copy()`	Copy the target dictionary

How do we access the information in a dictionary?

Example 4.18 shows how to use builtin methods to carry out three fundamental tasks: accessing dictionary keys (see line 5), values (see line 9), and items (i.e., key-value pairs, see line 13). It is worth noticing that the three methods illustrated in the example yield specific dictionary objects such as `dict_keys`, `dict_values`, and `dict_items`. Translating one of these dictionary objects into a list — if needed — is straightforward (see line 17).

Example 4.18 — accessing the information included in a dictionary

```
1  # the dictionary
2  >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4  # get the keys
5  >>> D.keys()
6  dict_keys(["Captain Marvel", "Living Tribunal", "One-Above-All"])
7
8  # get the values
9  >>> D.values()
10 dict_values([3, 2, 1])
11
12 # get the items
13 >>> D.items()
14 dict_items([("Captain Marvel", 3), ("Living Tribunal", 2), ("One-Above-All", 1)])
15
16 # get the keys as a list
17 >>> list(D.keys())
18 ["Captain Marvel", "Living Tribunal", "One-Above-All"]
```

4.5 Tuples

What is a tuple?

Tuples are sequences of immutable Python objects. They are similar to lists, but they are immutable. Tuples are created by enclosing a comma-separated list of values in parentheses.

Tuples are immutable!

Tuples are immutable, which means that once they are created, they cannot be changed!!

Why do we use tuples?

Tuples are useful for storing data that is not to be changed, such as the coordinates of a point in a two-dimensional space. In general, we use tuples any time information integrity is a concern — in other words when we want to make sure the information included in an object will not change because of another reference somewhere in our program.

How do we create a tuple?

Python objects, separated by a comma, must be included between parentheses (see Example 4.19, line 2).

How do we access the information in a tuple?

By positional offsets, like lists (see Example 4.19, lines 5 and 9).

Example 4.19 — creating and accessing a tuple

```
1  # the tuple
2  >>> T = ("Captain Marvel", 3)
3
4  # access a tuple element
5  >>> T[0]
6  "Captain Marvel"
7
8  # access a tuple element
9  >>> T[1]
10 3
```

Can we convert a tuple into a list?

Yes, we can. To do that, we pass the tuple as the argument of `list` (see Example 4.20).

Example 4.20 — tuple conversion

```
>>> T = ("Captain Marvel", 3)

# from a tuple to a list
>>> L = list(T)
>>> print(L)
["Captain Marvel", 3]

# amend L's items
>>> L[1] = 4

# get back to a tuple
>>> T = tuple(L)
>>> print(T)
("Captain Marvel", 4)
```

Tuples with the `collections` module

`collections` is a module that is shipped with Python and provides data containers that are alternative to Python's general purpose built-in containers, i.e., `dict`, `list`, `set`, and `tuple`. One of these containers can be created with the function `namedtuple` (see Example 4.21), which allows annotating the tuple items with names. In line 2, we import the function `namedtuple` from the `collections` module. In line 5, we create an ad hoc class that best represents the structure of our sample data, concerning Marvel characters' names and the year in which they first appeared in the comic series. The first argument taken by the function is customary and regards the name of the class we are about to create. The second argument is a list with the names of the attributes included in our data structure. In line 8, we use the newly created class `Rec` to create a tuple, which is eventually printed as per line 11.

Example 4.21 — creating an annotated tuple with the collection module

```
1  # import the namedtuple function from the module collection
2  >>> from collections import namedtuple
3
4  # create an ad hoc class object 'Rec' that fits our data structure
5  >>> Rec = namedtuple("Rec", ["character", "first_appearance"])
6
7  # use the generated class "Rec"
8  >>> IRONMAN = Rec("Iron Man", 1963)
9
10 # A named-tuple record
11 >>> IRONMAN
12 Rec(character="Iron Man", first_appearance=1963)
```

4.6 Sets

What is a set?

A `set` is an *unordered* collection of *unique* and *immutable* objects.

What does it mean that sets are unordered collections?

By design, `set` is a data structure with *undefined element ordering* (see Example 4.22 — the outcome included in line 6 does not follow any particular order).

What does it mean that sets have unique items?

By definition, an item appears only once in a set, no matter how many times it is added (see Example 4.22, line 2 Vs. line 7).

Example 4.22 — creating a set

```
# create a list
>>> L = ["a", "a", "b", "c", "c"]

# get a set from L
>>> S = set(L)
>>> print(S)
{"b", "a", "c"}
```

Why do we use sets?

Sets made this way support common mathematical set operations (see Example 4.23). Hence, they have a variety of applications, especially in numeric and database-focused work.

Example 4.23 — set operations

```
# create two sets
>>> X = set(["a", "b", "c"])
>>> Y = set(["c", "d", "e"])

# set difference
>>> X - X
set()
>>> X - Y
{"a", "b"}

# union
>>> X | Y
{"a", "b", "c", "d", "e"}

# intersection
>>> X & Y
{"c"}

# superset
>>> X > Y
False

# subset
>>> X < Y
False
```

4.7 Files

How do the files in our OS relate with Python?

Our Python program may involve input and/or output operations. In other words, we may want to read data from a file stored in our machine and/or write the outcome of our analysis to a file. The built-in function `open` creates a Python file object, which serves as a link to a file residing on your machine. As Lutz notes:

> *"Compared to the types you've seen so far, file objects are somewhat unusual. They are considered a core type because they are created by a built-in function, but they're not numbers, sequences, or mappings, and they don't respond to expression operators; they export only methods for common file-processing tasks" (page 282)*

How do we open a file?

We open a pipe to a file using the built-in function `open`. The output of the function is a `file` object. Example 4.24 illustrates the functioning of `open`. In the first part of the snippet, we create a `file` object to read the data included in the existing file `my_file.txt`.[5] At least, we have to pass one argument to `open`: the path pointing to the file. A second optional argument is `mode`, which specifies the mode in which the file is opened to source. It defaults to `r`, which means open for reading in text mode. Other common values are `w` for writing (truncating the file if it already exists), `x` for exclusive creation, and `a` for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). If encoding is not specified the encoding used is platform-dependent. Specifically, `locale.getpreferredencoding(False)` is called to get the current locale encoding.[6]

Example 4.24 — data input with open

```
# create a pipe to a file
>>> file = open(file="my_file.txt", mode="r")

# calling "file" yields the attributes of the file object
>>> file
<_io.TextIOWrapper name="my_file.txt" mode="r" encoding="UTF-8">

# let us source the data
>>> data = file.read()
>>> print(data)
Hi there
```

Example 4.25 — data output with open

```
# the strings (data) to save permanently to a file
FIRSTLAW = "A robot may not injure a human being or, through inaction, "\
           "allow a human being to come to harm."
SECONDLAW = "A robot must obey the orders given it by human beings except "\
            "where such orders would conflict with the First Law."
THIRDLAW = "A robot must protect its own existence as long as such "\
           "protection does not conflict with the First or Second Law."
# create a pipe to a file
>>> file = open(file="my_file.txt", mode="r")
```

Table 4.9
Popular file methods

Method	Description
`close()`	Closes the file
`detach()`	Returns the separated raw stream from the buffer
`fileno()`	Returns a number that represents the stream as per the OS' perspective
`flush()`	Flushes the internal buffer
`isatty()`	Returns whether the file stream is interactive or not
`read()`	Returns the file content
`readable()`	Returns whether the file stream can be read or not
`readline()`	Returns one line from the file
`readlines()`	Returns a list of lines from the file
`seek()`	Change the file position
`seekable()`	Returns whether the file allows us to change the file position
`tell()`	Returns the current file position
`truncate()`	Resizes the file to a specified size
`writable()`	Returns whether the file can be written to or not
`write()`	Writes the specified string to the file
`writelines()`	Writes a list of strings to the file

4.8 Python Statements and Syntax

...

4.9 If Test

...

4.10 While and For Loops

...

4.11 Iterations and Comprehensions

...

Notes

[1]Lutz, Mark. *Learning Python: Powerful object-oriented programming.* O'Reilly Media, Inc., 2013.

[2]Floating numbers are stored in binaries with an assigned level of precision that is typically equivalent to 15 or 16 decimals.

[3]As per the documentation of the Python programming language, `math` cannot be used with complex numbers.

[4]The official Python documentation has an extensive section on operator precedence rules in the section dedicated to syntax of expressions

[5]For the sake of simplicity, we assume the target file is located in the same directory as the Python script.

[6]Character encoding is the process of assigning numbers to graphical characters, especially the written characters of human language, allowing them to be stored, transmitted, and transformed using digital computers.

Chapter 5

Technical and Scientific Computation with NumPy and SciPy

...

Chapter 6

Data Management with Pandas and Dask

...

Chapter 7

Coda

...

Appendices

Appendix A

Cheat Sheets

TABLE A.1
Helpful Escapes

Escape	Meaning
`\\`	Backslash (stores one `\`)
`\'`	Single quotes escape (stores `'`)
`\"`	Double quotes escape (stores `"`)
`\a`	Bell
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

Table A.2
Comprehensive List of String Methods

Cases I	
s.capitalize()	Capitalize s # ’hello’ =>’Hello’
s.lower()	Lowercase s # ’HELLO’ =>’hello’
s.swapcase()	Swap cases of all characters in s # ’Hello’ =>”hELLO”
s.title()	Titlecase s # ’hello world’ =>’Hello World’
s.upper()	Uppercase s # ’hello’ =>’HELLO’
Sequence Operations I	
s2 in s	Return true if s contains s2
s + s2	Concat s and s2
len(s)	Length of s
min(s)	Smallest character of s
max(s)	Largest character of s
Sequence Operations II	
s2 not in s	Return true if s does not contain s2
s * integer	Return integer copies of s concatenated # ’hello’ =>’hellohellohello’
s[index]	Character at index of s
s[i:j:k]	Slice of s from i to j with step k
s.count(s2)	Count of s2 in s
Whitespace I	
s.center(width)	Center s with blank padding of width # ’hi’ =>’ hi ’
s.isspace()	Return true if s only contains whitespace characters
s.ljust(width)	Left justifiy s with total size of width # ’hello’ =>’hello ’
s.rjust(width)	Right justify s with total size of width # ’hello’ =>’ hello’
s.strip()	Remove leading and trailing whitespace from s # ’ hello ’ =>’hello’
Find / Replace I	
s.index(s2, i, j)	Index of first occurrence of s2 in s after index i and before index j
s.find(s2)	Find and return lowest index of s2 in s
s.index(s2)	Return lowest index of s2 in s (but raise ValueError if not found)
s.replace(s2, s3)	Replace s2 with s3 in s
s.replace(s2, s3, count)	Replace s2 with s3 in s at most count times
s.rfind(s2)	Return highest index of s2 in s
s.rindex(s2)	Return highest index of s2 in s (raise ValueError if not found)
Cases II	
s.casefold()	Casefold s (aggressive lowercasing for caseless matching) # ’ßorat’ =>’ssorat’
s.islower()	Return true if s is lowercase
s.istitle()	Return true if s is titlecased # ’Hello World’ =>true
s.isupper()	Return true if s is uppercase
Inspection I	
s.endswith(s2)	Return true if s ends with s2
s.isalnum()	Return true if s is alphanumeric
s.isalpha()	Return true if s is alphabetic
s.isdecimal()	Return true if s is decimal
s.isnumeric()	Return true if s is numeric
s.startswith(s2)	Return true is s starts with s2

Table A.2
(Cont'ed)

Splitting I	
s.join('123')	Return s joined by iterable '123' # 'hello' =>'1hello2hello3'
s.partition(sep)	Partition string at sep and return 3-tuple with part before, the sep itself, and part after # 'hello' =>('he', 'l', 'lo')
s.rpartition(sep)	Partition string at last occurrence of sep, return 3-tuple with part before, the sep, and part after # 'hello' =>('hel', 'l', 'o')
s.rsplit(sep, maxsplit)	Return list of s split by sep with rightmost maxsplits performed
s.split(sep, maxsplit)	Return list of s split by sep with leftmost maxsplits performed
s.splitlines()	Return a list of lines in s # 'hello\nworld' =>['hello', 'world']
Inspection II	
s[i:j]	Slice of s from i to j
s.endswith((s1, s2, s3))	Return true if s ends with any of string tuple s1, s2, and s3
s.isdigit()	Return true if s is digit
s.isidentifier()	Return true if s is a valid identifier
s.isprintable()	Return true is s is printable
Whitespace II	
s.center(width, pad)	Center s with padding pad of width # 'hi' =>'padpadhipadpad'
s.expandtabs(integer)	Replace all tabs with spaces of tabsize integer # 'hello\tworld' =>'hello world'
s.lstrip()	Remove leading whitespace from s # ' hello ' =>'hello '
s.rstrip()	Remove trailing whitespace from s # ' hello ' =>' hello'
s.zfill(width)	Left fill s with ASCII '0' digits with total length width # '42' =>'00042'

Appendix B

Collaborative and Versioning Tools

...