SMM692
Introduction to Programming in Python

Contents

Chapter 1

Organization of the Module

...

Chapter 2

Getting Started with Python

...

Chapter 3

Managing Python Environments

...

Chapter 4

Collaborative and Versioning Tools

...

Chapter 5

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 I 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 I.

TABLE I
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

5.1 Number Type Fundamentals

Types of 'number' objects

Example 5.1, "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 5.1 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 5.1 — doing 'stuff' with numbers

```
# integer addition
In [1]: 1 + 1
Out[1]: 2

# floating-point multiplication
In [2]: 10 * 0.5
Out[2]: 5.0

# 3 to the power 100
In [3]: 3 ** 100
Out[3]: 515377520732011331036461129765621272702107522001

```

Besides integers and floating points

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

TABLE II
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 5.1).

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 III reports a sample of these functions. To use them `math`, we have to import the module as shown in Example 5.2. Another popular module shipped with Python is `random`, implementing pseudo-random number generators for various distributions (see the lower section of Example 2).

TABLE III
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 5.2 — advanced mathematical operations with the modules shipped with Python

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

Operator precedence

As shown in Example 5.2, 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 IV 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 IV
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 5.3 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 5.3 — expressions involving arithmetic operations

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

Displaying number objects

Example 5.3 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 V 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 5.4 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 V
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 5.4 — number formatting in Python

```
1  # assign the variable 'a' to a floating-point number
2  In [1]: a = 0.67544908755
3
4  # displaying 'a' with the first two decimals only
5  In [2]: "{:.2f}".format(a)
6  Out[2]: "0.68"
7
8  # displaying 'a' with the first three decimals only
9  In [3]: "{:.3f}".format(a)
10 Out[3]: "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 5.5 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 5.4 — comparing numeric objects

```
# less than
In [1]: 3 < 2
Out[1]: False

# greater than or equal
In [2]: 1 <= 2
Out[2]: True

# equal
In [3]: 2 == 2
Out[3]: True

# not equal
In [4]: 4 != 4
Out[4]: False

# range test
In [5]: x = 3
In [6]: y = 5
In [7]: z = 4
In [8]: x < y < z
Out[8]: False

# joined test
In [9]: x < y and y > z
Out[9]: True

# disjoined test
In [10]: x < y or y < z
Out[10]: True
```

5.2 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 5.5, 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 5.5 — Python strings as sequences

```
# let us assign the string "Python 3.X" to the variable S
In [1]: S = "Python 3.X"

# check the length of S
In [2]: len(S)
Out[2]: 6

# access the first unitary string in the sequence behind S
In [3]: S[0]
Out[3]: "P"

# access the last unitary string in the sequence behind S
In [4]: S[len(S)-1]
Out[4]: "X"

# or, equivalently
In [5]: S[-1]
Out[5]: "X"

# access the i-th, e.g., 3rd, unitary string in the sequence behind S
In [6]: S[3]
Out[6]: "h"

# access the unitary strings between the i-th and j-th positions in the
# sequence behind S
In [7]: S[2:5]
Out[7]: "tho"

# access the unitary strings following the i-th position in the sequence
# behind S
In [8]: S[-3:]
Out[8]: "3.X"
```

Common string literals and operators

Example 5.5 deals with string indexing and slicing, two of the many operations we can carry out on strings. Table VI reports a sample of common string literals and operators. The first two lines of Table VI remind us that single and double quotes are equivalent when it comes to assign 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 all throughout the various lines of the script. I prefer using double quotes for the reason that is evident the third line of the

TABLE VI
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 = """...multiline..."""`	Triple-quoted block strings
`S1 + S2`	Concatenate
`S * 3`	Repeat S n times (e.g., three times)
`S[i]`	Index
`S[i:j]`	Slice
`length(S)`	Length
`"{}".format()`	String formatting expression
`S.find("pa")`	Search
`S.strip()`	Remove white spaces
`S.replace("pa", "xx")`	Replacement
`S.split(",")`	Split on a character (e.g., “,”)
`S.isdigit()`	Test if the string is a digit
`S.lower()`	Case conversion — to lower case
`S.upper()`	Case conversion — to upper case
`S.endswith("spam")`	End test
`"-".join(strlist)`	Join multiple strings on a character (e.g., “-”)

5.3 List and Dictionaries

...

5.4 Tuples, Files, and Everything Else

...

5.5 Python Statements

...

5.6 If Test

...

5.7 While and For Loops

...

5.8 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

Chapter 6

Technical and Scientific Computation with NumPy and SciPy

...

Chapter 7

Data Management with Pandas and Dask

...

Chapter 8

Coda

...