# SMM692
# Introduction to Programming in Python

# Contents

# List of Figures

# List of Tables

# 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

...

## 2.5 Collaborative and Versioning Tools

...

# 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 ?? 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 ??.

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

Besides integers and floating points

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

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 **??** 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

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 **??** 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

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 **??** 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**

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

## 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

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 **??** reports a sample of common string literals and operators. The first two lines of Table **??** 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 **??** 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 **??**. Table **??** 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 **??**.

Example 4.7 — miscellaneous string manipulation tasks

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

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

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

## 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

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

List mutability

Lists may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more — they are mutable objects..

## 4.4 Tuples, Files, and Everything Else

...

## 4.5 Python Statements

...

## 4.6 If Test

...

## 4.7 While and For Loops

...

## 4.8 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 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' |