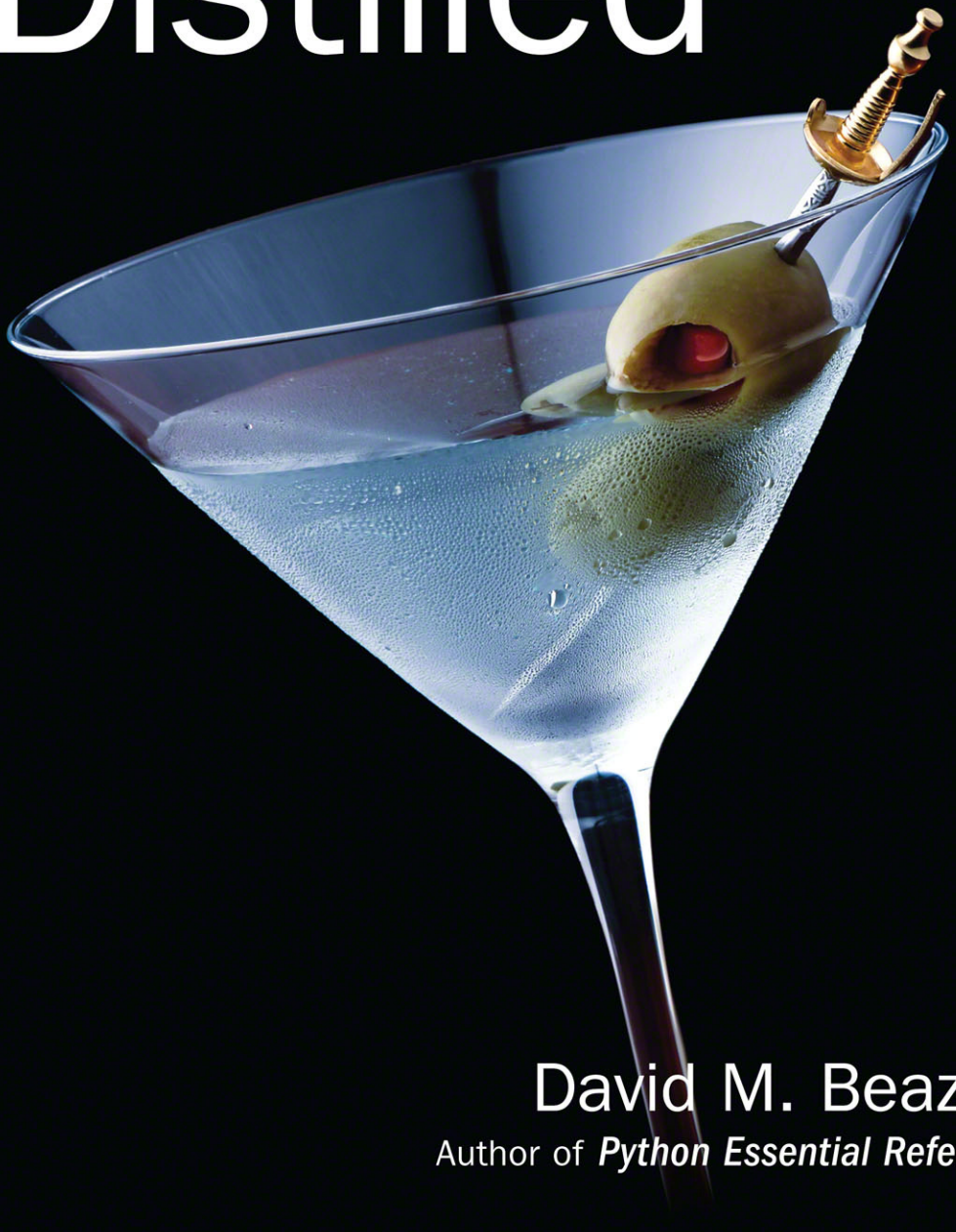
Python Distilled



David M. Beazley
Author of *Python Essential Reference*

Python Distilled

This page intentionally left blank

Python Distilled

David M. Beazley

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021943288

Cover illustration by EHStockphoto/Shutterstock

ISBN-13: 978-0-13-417327-6
ISBN-10: 0-13-417327-9

ScoutAutomatedPrintCode

Contents

Preface xiii

1 Python Basics 1

1.1 Running Python 1
1.2 Python Programs 2
1.3 Primitives, Variables, and Expressions 3
1.4 Arithmetic Operators 5
1.5 Conditionals and Control Flow 7
1.6 Text Strings 9
1.7 File Input and Output 12
1.8 Lists 13
1.9 Tuples 15
1.10 Sets 17
1.11 Dictionaries 18
1.12 Iteration and Looping 21
1.13 Functions 22
1.14 Exceptions 24
1.15 Program Termination 26
1.16 Objects and Classes 26
1.17 Modules 30
1.18 Script Writing 32
1.19 Packages 33
1.20 Structuring an Application 34
1.21 Managing Third-Party Packages 35
1.22 Python: It Fits Your Brain 36

2 Operators, Expressions, and Data Manipulation 37

2.1 Literals 37
2.2 Expressions and Locations 38
2.3 Standard Operators 39
2.4 In-Place Assignment 41
2.5 Object Comparison 42
2.6 Ordered Comparison Operators 42

2.7 Boolean Expressions and Truth Values 43
2.8 Conditional Expressions 44
2.9 Operations Involving Iterables 45
2.10 Operations on Sequences 47
2.11 Operations on Mutable Sequences 49
2.12 Operations on Sets 50
2.13 Operations on Mappings 51
2.14 List, Set, and Dictionary Comprehensions 52
2.15 Generator Expressions 54
2.16 The Attribute (.) Operator 56
2.17 The Function Call ( ) Operator 56
2.18 Order of Evaluation 56
2.19 Final Words: The Secret Life of Data 58

3 Program Structure and Control Flow 59

3.1 Program Structure and Execution 59
3.2 Conditional Execution 59
3.3 Loops and Iteration 60
3.4 Exceptions 64
3.4.1 The Exception Hierarchy 67
3.4.2 Exceptions and Control Flow 68
3.4.3 Defining New Exceptions 69
3.4.4 Chained Exceptions 70
3.4.5 Exception Tracebacks 73
3.4.6 Exception Handling Advice 73
3.5 Context Managers and the `with` Statement 75
3.6 Assertions and `__debug__` 77
3.7 Final Words 78

4 Objects, Types, and Protocols 79
4.1 Essential Concepts 79
4.2 Object Identity and Type 80
4.3 Reference Counting and Garbage Collection 81
4.4 References and Copies 83
4.5 Object Representation and Printing 84
4.6 First-Class Objects 85
4.7 Using None for Optional or Missing Data 87
4.8 Object Protocols and Data Abstraction 87
4.9 Object Protocol 89
4.10 Number Protocol 90
4.11 Comparison Protocol 92
4.12 Conversion Protocols 94
4.13 Container Protocol 95
4.14 Iteration Protocol 97
4.15 Attribute Protocol 98
4.16 Function Protocol 98
4.17 Context Manager Protocol 99
4.18 Final Words: On Being Pythonic 99

5 Functions 101
5.1 Function Definitions 101
5.2 Default Arguments 101
5.3 Variadic Arguments 102
5.4 Keyword Arguments 103
5.5 Variadic Keyword Arguments 104
5.6 Functions Accepting All Inputs 104
5.7 Positional-Only Arguments 105
5.8 Names, Documentation Strings, and Type Hints 106
5.9 Function Application and Parameter Passing 107
5.10 Return Values 109
5.11 Error Handling 110
5.12 Scoping Rules 111
5.13 Recursion 114

5.14 The `lambda` Expression 114
5.15 Higher-Order Functions 115
5.16 Argument Passing in Callback Functions 118
5.17 Returning Results from Callbacks 121
5.18 Decorators 124
5.19 Map, Filter, and Reduce 127
5.20 Function Introspection, Attributes, and Signatures 129
5.21 Environment Inspection 131
5.22 Dynamic Code Execution and Creation 133
5.23 Asynchronous Functions and `await` 135
5.24 Final Words: Thoughts on Functions and Composition 137

6 Generators 139

6.1 Generators and `yield` 139
6.2 Restartable Generators 142
6.3 Generator Delegation 142
6.4 Using Generators in Practice 144
6.5 Enhanced Generators and `yield` Expressions 146
6.6 Applications of Enhanced Generators 148
6.7 Generators and the Bridge to Awaiting 151
6.8 Final Words: A Brief History of Generators and Looking Forward 152

7 Classes and Object-Oriented Programming 153

7.1 Objects 153
7.2 The `class` Statement 154
7.3 Instances 155
7.4 Attribute Access 156
7.5 Scoping Rules 158

7.6 Operator Overloading and Protocols 159
7.7 Inheritance 160
7.8 Avoiding Inheritance via Composition 163
7.9 Avoiding Inheritance via Functions 166
7.10 Dynamic Binding and Duck Typing 167
7.11 The Danger of Inheriting from Built-in Types 167
7.12 Class Variables and Methods 169
7.13 Static Methods 173
7.14 A Word about Design Patterns 176
7.15 Data Encapsulation and Private Attributes 176
7.16 Type Hinting 179
7.17 Properties 180
7.18 Types, Interfaces, and Abstract Base Classes 183
7.19 Multiple Inheritance, Interfaces, and Mixins 187
7.20 Type-Based Dispatch 193
7.21 Class Decorators 194
7.22 Supervised Inheritance 197
7.23 The Object Life Cycle and Memory Management 199
7.24 Weak References 204
7.25 Internal Object Representation and Attribute Binding 206
7.26 Proxies, Wrappers, and Delegation 208
7.27 Reducing Memory Use with `__slots__` 210
7.28 Descriptors 211
7.29 Class Definition Process 215
7.30 Dynamic Class Creation 216
7.31 Metaclasses 217
7.32 Built-in Objects for Instances and Classes 222
7.33 Final Words: Keep It Simple 223

8 Modules and Packages 225
8.1 Modules and the `import` Statement 225

8.2 Module Caching 227
8.3 Importing Selected Names from a Module 228
8.4 Circular Imports 230
8.5 Module Reloading and Unloading 232
8.6 Module Compilation 233
8.7 The Module Search Path 234
8.8 Execution as the Main Program 234
8.9 Packages 235
8.10 Imports Within a Package 237
8.11 Running a Package Submodule as a Script 238
8.12 Controlling the Package Namespace 239
8.13 Controlling Package Exports 240
8.14 Package Data 241
8.15 Module Objects 242
8.16 Deploying Python Packages 243
8.17 The Penultimate Word: Start with a Package 244
8.18 The Final Word: Keep It Simple 245

9 Input and Output 247
9.1 Data Representation 247
9.2 Text Encoding and Decoding 248
9.3 Text and Byte Formatting 250
9.4 Reading Command-Line Options 254
9.5 Environment Variables 256
9.6 Files and File Objects 256
9.6.1 Filenames 257
9.6.2 File Modes 258
9.6.3 I/O Buffering 258
9.6.4 Text Mode Encoding 259
9.6.5 Text-Mode Line Handling 260
9.7 I/O Abstraction Layers 260
9.7.1 File Methods 261

9.8 Standard Input, Output, and Error 263
9.9 Directories 264
9.10 The `print()` function 265
9.11 Generating Output 265
9.12 Consuming Input 266
9.13 Object Serialization 268
9.14 Blocking Operations and Concurrency 269
9.14.1 Nonblocking I/O 270
9.14.2 I/O Polling 271
9.14.3 Threads 271
9.14.4 Concurrent Execution with `asyncio` 272
9.15 Standard Library Modules 273
9.15.1 `asyncio` Module 273
9.15.2 `binascii` Module 274
9.15.3 `cgi` Module 275
9.15.4 `configparser` Module 276
9.15.5 `csv` Module 276
9.15.6 `errno` Module 277
9.15.7 `fcntl` Module 278
9.15.8 `hashlib` Module 278
9.15.9 `http` Package 279
9.15.10 `io` Module 279
9.15.11 `json` Module 280
9.15.12 `logging` Module 280
9.15.13 `os` Module 281
9.15.14 `os.path` Module 281
9.15.15 `pathlib` Module 282
9.15.16 `re` Module 283
9.15.17 `shutil` Module 284
9.15.18 `select` Module 284
9.15.19 `smtplib` Module 285
9.15.20 `socket` Module 286
9.15.21 `struct` Module 288
9.15.22 `subprocess` Module 288
9.15.23 `tempfile` Module 289
9.15.24 `textwrap` Module 290

9.15.25 `threading` Module 291
9.15.26 `time` Module 293
9.15.27 `urllib` Package 293
9.15.28 `unicodedata` Module 294
9.15.29 `xml` Package 295
9.16 Final Words 296

10 Built-in Functions and Standard Library 297
10.1 Built-in Functions 297
10.2 Built-in Exceptions 314
10.2.1 Exception Base Classes 314
10.2.2 Exception Attributes 314
10.2.3 Predefined Exception Classes 315
10.3 Standard Library 318
10.3.1 `collections` Module 318
10.3.2 `datetime` Module 318
10.3.3 `itertools` Module 318
10.3.4 `inspect` Module 318
10.3.5 `math` Module 318
10.3.6 `os` Module 319
10.3.7 `random` Module 319
10.3.8 `re` Module 319
10.3.9 `shutil` Module 319
10.3.10 `statistics` Module 319
10.3.11 `sys` Module 319
10.3.12 `time` Module 319
10.3.13 `turtle` Module 319
10.3.14 `unittest` Module 319
10.4 Final Words: Use the Built-Ins 320

Index 321

Preface

More than 20 years have passed since I authored the *Python Essential Reference*. At that time, Python was a much smaller language and it came with a useful set of batteries in its standard library. It was something that could mostly fit in your brain. The *Essential Reference* reflected that era. It was a small book that you could take with you to write some Python code on a desert island or inside a secret vault. Over the three subsequent revisions, the *Essential Reference* stuck with this vision of being a compact but complete language reference—because if you're going to code in Python on vacation, why not use all of it?

Today, more than a decade since the publication of the last edition, the Python world is much different. No longer a niche language, Python has become one of the most popular programming languages in the world. Python programmers also have a wealth of information at their fingertips in the form of advanced editors, IDEs, notebooks, web pages, and more. In fact, there's probably little need to consult a reference book when almost any reference material you might want can be conjured to appear before your eyes with the touch of a few keys.

If anything, the ease of information retrieval and the scale of the Python universe presents a different kind of challenge. If you're just starting to learn or need to solve a new problem, it can be overwhelming to know where to begin. It can also be difficult to separate the features of various tools from the core language itself. These kinds of problems are the rationale for this book.

Python Distilled is a book about programming in Python. It's not trying to document everything that's possible or has been done in Python. Its focus is on presenting a modern yet curated (or distilled) core of the language. It has been informed by my years of teaching Python to scientists, engineers, and software professionals. However, it's also a product of writing software libraries, pushing the edges of what makes Python tick, and finding out what's most useful.

For the most part, the book focuses on Python programming itself. This includes abstraction techniques, program structure, data, functions, objects, modules, and so forth—topics that will well serve programmers working on Python projects of any size. Pure reference material that can be easily obtained via an IDE (such as lists of functions, names of commands, arguments, etc.) is generally omitted. I've also made a conscious choice to not describe the fast-changing world of Python tooling—editors, IDEs, deployment, and related matters.

Perhaps controversially, I don't generally focus on language features related to large-scale software project management. Python is sometimes used for big and serious things—comprised of millions upon millions of lines of code. Such applications require specialized tooling, design, and features. They also involve committees, and meetings, and decisions to be made about very important matters. All this is too much for this small book. But

perhaps the honest answer is that I don't use Python to write such applications—and neither should you. At least not as a hobby.

In writing a book, there is always a cut-off for the ever-evolving language features. This book was written during the era of Python 3.9. As such, it does not include some of the major additions planned for later releases—for example, structural pattern matching. That's a topic for a different time and place.

Last, but not least, I think it's important that programming remains fun. I hope that my book will not only help you become a productive Python programmer but also capture some of the magic that has inspired people to use Python for exploring the stars, flying helicopters on Mars, and spraying squirrels with a water cannon in the backyard.

Acknowledgments

I'd like to thank the technical reviewers, Shawn Brown, Sophie Tabac, and Pete Fein, for their helpful comments. I'd also like to thank my long-time editor Debra Williams Cauley for her work on this and past projects. The many students who have taken my classes have had a major if indirect impact on the topics covered in this book. Last, but not least, I'd like to thank Paula, Thomas, and Lewis for their support and love.

About the Author

David Beazley is the author of the *Python Essential Reference, Fourth Edition* (Addison-Wesley, 2010) and *Python Cookbook, Third Edition* (O'Reilly, 2013). He currently teaches advanced computer science courses through his company Dabeaz LLC (`www.dabeaz.com`). He's been using, writing, speaking, and teaching about Python since 1996.

1

Python Basics

This chapter gives an overview of the core of the Python language. It covers variables, data types, expressions, control flow, functions, classes, and input/output. The chapter concludes with a discussion of modules, script writing, packages, and a few tips on organizing larger programs. This chapter is not trying to provide comprehensive coverage of every feature, nor does it concern itself with all of the tooling that might surround a larger Python project. However, experienced programmers should be able to extrapolate from the material here to write more advanced programs. Newcomers are encouraged to try the examples in a simple environment, such as a terminal window and a basic text editor.

1.1 Running Python

Python programs are executed by an interpreter. There are many different environments in which the Python interpreter might run—an IDE, a browser, or a terminal window. However, underneath all that, the core of the interpreter is a text-based application that can be started by typing `python` in a command shell such as `bash`. Since Python 2 and Python 3 might both be installed on the same machine, you might need to type `python2` or `python3` to pick a version. This book assumes Python 3.8 or newer.

When the interpreter starts, a prompt appears where you can type programs into a so-called "read-evaluation-print loop" (or REPL). For example, in the following output, the interpreter displays its copyright message and presents the user with the `>>>` prompt, at which the user types a familiar "Hello World" program:

```
Python 3.8.0 (default, Feb  3 2019, 05:53:21)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World')
Hello World
>>>
```

Certain environments may display a different prompt. The following output is from `ipython` (an alternate shell for Python):

```
Python 3.8.0 (default, Feb 4, 2019, 07:39:16)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print('Hello World')
Hello World

In [2]:
```

Regardless of the exact form of output you see, the underlying principle is the same. You type a command, it runs, and you immediately see the output.

Python's interactive mode is one of its most useful features because you can type any valid statement and immediately see the result. This is useful for debugging and experimentation. Many people, including the author, use interactive Python as their desktop calculator. For example:

```
>>> 6000 + 4523.50 + 134.25
10657.75
>>> _ + 8192.75
18850.5
>>>
```

When you use Python interactively, the variable _ holds the result of the last operation. This is useful if you want to use that result in subsequent statements. This variable only gets defined when working interactively, so don't use it in saved programs.

You can exit the interactive interpreter by typing `quit()` or the EOF (end of file) character. On UNIX, EOF is Ctrl+D; on Windows, it's Ctrl+Z.

1.2 Python Programs

If you want to create a program that you can run repeatedly, put statements in a text file. For example:

```
# hello.py
print('Hello World')
```

Python source files are UTF-8-encoded text files that normally have a `.py` suffix. The # character denotes a comment that extends to the end of the line. International (Unicode) characters can be freely used in the source code as long as you use the UTF-8 encoding (this is the default in most editors, but it never hurts to check your editor settings if you're unsure).

To execute the `hello.py` file, provide the filename to the interpreter as follows:

```
shell % python3 hello.py
Hello World
shell %
```

It is common to use `#!` to specify the interpreter on the first line of a program, like this:

```
#!/usr/bin/env python3
print('Hello World')
```

On UNIX, if you give this file execute permissions (for example, by `chmod +x hello.py`), you can run the program by typing `hello.py` into your shell.

On Windows, you can double-click on a `.py` file or type the name of the program into the Run command on the Windows Start menu to launch it. The `#!` line, if given, is used to pick the interpreter version (Python 2 versus 3). Execution of a program might take place in a console window that disappears immediately after the program completes—often before you can read its output. For debugging, it's better to run the program within a Python development environment.

The interpreter runs statements in order until it reaches the end of the input file. At that point, the program terminates and Python exits.

1.3 Primitives, Variables, and Expressions

Python provides a collection of primitive types such as integers, floats, and strings:

```
42            # int
4.2           # float
'forty-two'   # str
True          # bool
```

A variable is a name that refers to a value. A value represents an object of some type:

```
x = 42
```

Sometimes you might see a type explicitly attached to a name. For example:

```
x: int = 42
```

The type is merely a hint to improve code readability. It can be used by third-party code-checking tools. Otherwise, it is completely ignored. It does not prevent you from assigning a different kind of value later.

An expression is a combination of primitives, names, and operators that produces a value:

```
2 + 3 * 4       # -> 14
```

The following program uses variables and expressions to perform a compound-interest calculation:

```
# interest.py

principal = 1000        # Initial amount
rate = 0.05             # Interest rate
numyears = 5            # Number of years
year = 1
while year <= numyears:
    principal = principal * (1 + rate)
    print(year, principal)
    year += 1
```

When executed, it produces the following output:

```
1 1050.0
2 1102.5
3 1157.625
4 1215.5062500000001
5 1276.2815625000003
```

The `while` statement tests the conditional expression that immediately follows. If the tested condition is true, the body of the `while` statement executes. The condition is then retested and the body executed again until the condition becomes false. The body of the loop is denoted by indentation. Thus, the three statements following `while` in `interest.py` execute on each iteration. Python doesn't specify the amount of required indentation, as long as it's consistent within a block. It is most common to use four spaces per indentation level.

One problem with the `interest.py` program is that the output isn't very pretty. To make it better, you could right-align the columns and limit the precision of principal to two digits. Change the `print()` function to use a so-called *f-string* like this:

```
print(f'{year:>3d} {principal:0.2f}')
```

In the f-string, variable names and expressions can be evaluated by enclosing them in curly braces. Optionally, each substitution can have a formatting specifier attached to it. `'>3d'` means a three-digit decimal number, right aligned. `'0.2f'` means a floating-point number with two decimal places of accuracy. More information about these formatting codes can be found in Chapter 9.

Now the output of the program looks like this:

```
  1 1050.00
  2 1102.50
  3 1157.62
  4 1215.51
  5 1276.28
```

1.4 Arithmetic Operators

Python has a standard set of mathematical operators, shown in Table 1.1. These operators have the same meaning they do in most other programming languages.

Table 1.1 Arithmetic Operators

Operation	Description
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x // y`	Truncating division
`x ** y`	Power (x to the y power)
`x % y`	Modulo (x mod y). Remainder.
`–x`	Unary minus
`+x`	Unary plus

The division operator (`/`) produces a floating-point number when applied to integers. Therefore, `7/4` is `1.75`. The truncating division operator `//`, also known as floor division, truncates the result to an integer and works with both integers and floating-point numbers. The modulo operator returns the remainder of the division `x // y`. For example, `7 % 4` is `3`. For floating-point numbers, the modulo operator returns the floating-point remainder of `x // y`, which is `x – (x // y) * y`.

In addition, the built-in functions in Table 1.2 provide a few more commonly used numerical operations.

Table 1.2 Common Mathematic Functions

Function	Description
`abs(x)`	Absolute value
`divmod(x,y)`	Returns `(x // y, x % y)`
`pow(x,y [,modulo])`	Returns `(x ** y) % modulo`
`round(x,[n])`	Rounds to the nearest multiple of 10 to the nth power.

The `round()` function implements "banker's rounding." If the value being rounded is equally close to two multiples, it is rounded to the nearest even multiple (for example, 0.5 is rounded to 0.0, and 1.5 is rounded to 2.0).

Integers provide a few additional operators to support bit manipulation, shown in Table 1.3.

Table 1.3 Bit Manipulation Operators

Operation	Description
`x << y`	Left shift
`x >> y`	Right shift
`x & y`	Bitwise and
`x | y`	Bitwise or
`x ^ y`	Bitwise xor (exclusive or)
`~x`	Bitwise negation

One would commonly use these with binary integers. For example:

```
a = 0b11001001
mask = 0b11110000
x = (a & mask) >> 4   # x = 0b1100 (12)
```

In this example, `0b11001001` is how you write an integer value in binary. You could have written it as decimal `201` or hexadecimal `0xc9`, but if you're fiddling with bits, binary makes it easier to visualize what you're doing.

The semantics of the bitwise operators assumes that the integers use a two's complement binary representation and that the sign bit is infinitely extended to the left. Some care is required if you are working with raw bit patterns that are intended to map to native integers on the hardware. This is because Python does not truncate the bits or allow values to overflow—instead, the result will grow arbitrarily large in magnitude. It's up to you to make sure the result is properly sized or truncated if needed.

To compare numbers, use the comparison operators in Table 1.4.

Table 1.4 Comparison Operators

Operation	Description
`x == y`	Equal to
`x != y`	Not equal to
`x < y`	Less than
`x > y`	Greater than
`x >= y`	Greater than or equal to
`x <= y`	Less than or equal to

The result of a comparison is a Boolean value `True` or `False`.

The `and`, `or`, and `not` operators (not to be confused with the bit-manipulation operators above) can form more complex Boolean expressions. The behavior of these operators is as shown in Table 1.5.

A value is considered false if it is literally `False`, `None`, numerically zero, or empty. Otherwise, it's considered true.

Table 1.5 Logical Operators

Operator	Description
`x or y`	If x is false, return y; otherwise, return x.
`x and y`	If x is false, return x; otherwise, return y.
`not x`	If x is false, return `True`; otherwise, return `False`.

It is common to write an expression that updates a value. For example:

```
x = x + 1
y = y * n
```

For these, you can write the following shortened operation instead:

```
x += 1
y *= n
```

This shortened form of update can be used with any of the `+`, `-`, `*`, `**`, `/`, `//`, `%`, `&`, `|`, `^`, `<<`, `>>` operators. Python does not have increment (`++`) or decrement (`--`) operators found in some other languages.

1.5 Conditionals and Control Flow

The `while`, `if` and `else` statements are used for looping and conditional code execution. Here's an example:

```
if a < b:
    print('Computer says Yes')
else:
    print('Computer says No')
```

The bodies of the `if` and `else` clauses are denoted by indentation. The `else` clause is optional. To create an empty clause, use the `pass` statement, as follows:

```
if a < b:
    pass      # Do nothing
else:
    print('Computer says No')
```

To handle multiple-test cases, use the `elif` statement:

```
if suffix == '.htm':
    content = 'text/html'
elif suffix == '.jpg':
    content = 'image/jpeg'
```

```
elif suffix == '.png':
    content = 'image/png'
else:
    raise RuntimeError(f'Unknown content type {suffix!r}')
```

If you are assigning a value in combination with a test, use a conditional expression:

```
maxval = a if a > b else b
```

This is the same as the longer:

```
if a > b:
    maxval = a
else:
    maxval = b
```

Sometimes, you may see the assignment of a variable and a conditional combined together using the `:=` operator. This is known as an assignment expression (or more colloquially as the "walrus operator" because `:=` looks like a walrus tipped over on its side—presumably playing dead). For example:

```
x = 0
while (x := x + 1) < 10:   # Prints 1, 2, 3, ..., 9
    print(x)
```

The parentheses used to surround an assignment expression are always required.

The `break` statement can be used to abort a loop early. It only applies to the innermost loop. For example:

```
x = 0
while x < 10:
    if x == 5:
        break  # Stops the loop. Moves to Done below
    print(x)
    x += 1

print('Done')
```

The `continue` statement skips the rest of the loop body and goes back to the top of the loop. For example:

```
x = 0
while x < 10:
    x += 1
    if x == 5:
       continue  # Skips the print(x). Goes back to loop start.
    print(x)

print('Done')
```

1.6 Text Strings

To define a string literal, enclose it in single, double, or triple quotes as follows:

```
a = 'Hello World'
b = "Python is groovy"
c = '''Computer says no.'''
d = """Computer still says no."""
```

The same type of quote used to start a string must be used to terminate it. Triple-quoted strings capture all the text until the terminating triple quote—as opposed to single- and double-quoted strings which must be specified on one logical line. Triple-quoted strings are useful when the contents of a string literal span multiple lines of text:

```
print('''Content-type: text/html

<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.
''')
```

Immediately adjacent string literals are concatenated into a single string. Thus, the above example could also be written as:

```
print(
'Content-type: text/html\n'
'\n'
'<h1> Hello World </h1>\n'
'Clock <a href="http://www.python.org">here</a>\n'
)
```

If the opening quotation mark of a string is prefaced by an `f`, escaped expressions within a string are evaluated. For example, in earlier examples, the following statement was used to output values of a calculation:

```
print(f'{year:>3d} {principal:0.2f}')
```

Although this is only using simple variable names, any valid expression can appear. For example:

```
base_year = 2020
...
print(f'{base_year + year:>4d} {principal:0.2f}')
```

As an alternative to f-strings, the `format()` method and `%` operator are also sometimes used to format strings. For example:

```
print('{0:>3d} {1:0.2f}'.format(year, principal))
print('%3d %0.2f' % (year, principal))
```