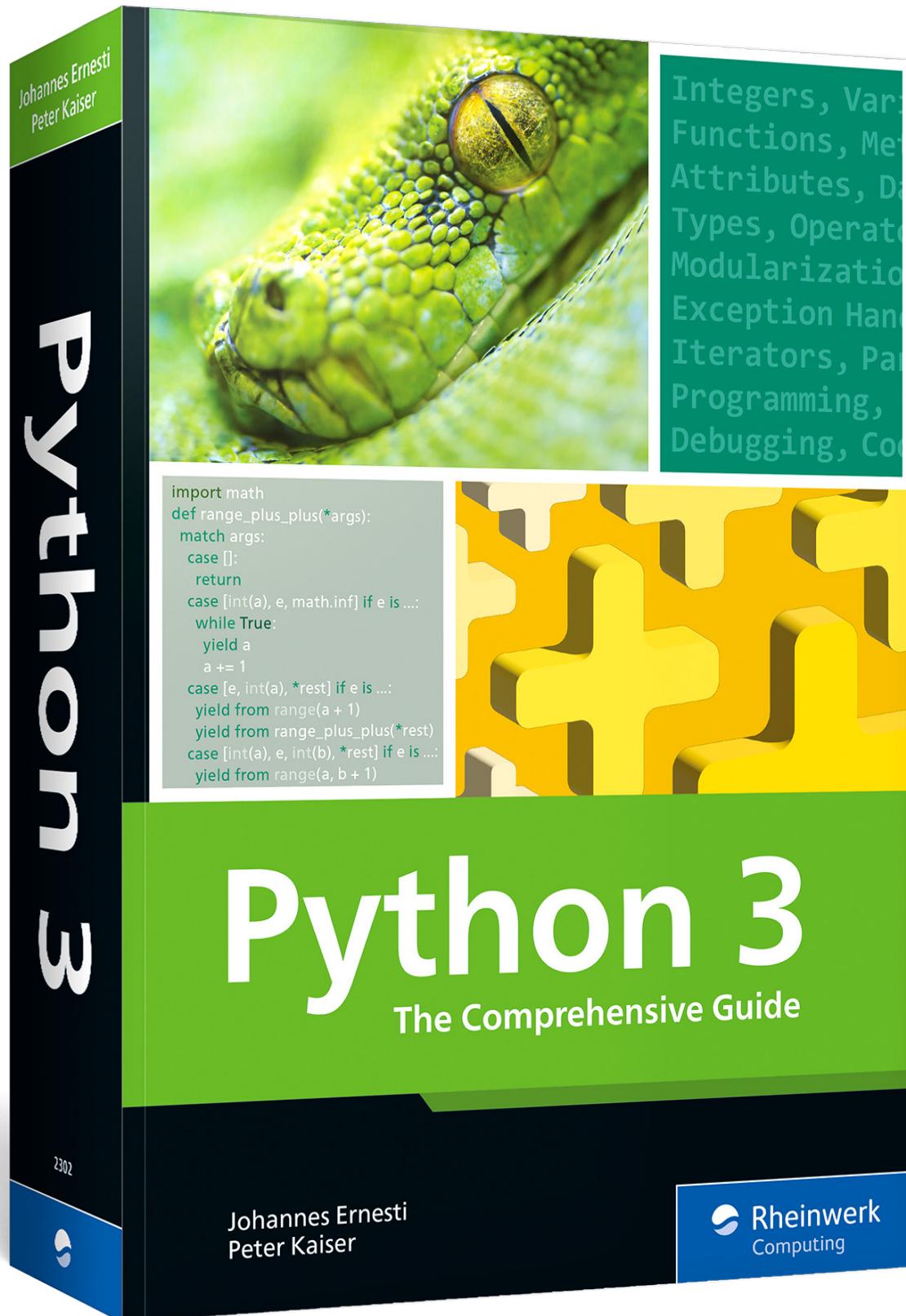


Build and deepen your coding knowledge  
from the top programming experts!

 Rheinwerk  
Computing



## Reading Sample

This chapter covers numeric data types, a major group of data types in the Python language. First, you'll learn about arithmetic and comparison operators, then see how to convert numeric data types using the `int`, `float`, `bool`, and `complex` functions. The chapter then goes into more detail on each of these functions, starting with `int` data type, which is used when working with integers. Next, you'll see how to use the `float` data type to store floats, then learn about Boolean values and using operators to create Boolean expressions. The chapter concludes with an overview of using the `complex` data type to store complex numbers.

 "Numeric Data Types"

 Contents

 Index

 The Author

Johannes Ernesti, Peter Kaiser

**Python 3: The Comprehensive Guide**

1036 pages, 2022, \$59.95

ISBN 978-1-4932-2302-2

 [www.rheinwerk-computing.com/5566](http://www.rheinwerk-computing.com/5566)

# Chapter 11

## Numeric Data Types

In this chapter, we'll describe numeric data types, the first major group of data types in Python. Table 11.1 lists all the data types belonging to this group and describes their purpose.

Data Type	Description	Mutability*	Section
<code>int</code>	Integers	Immutable	Section 11.4
<code>float</code>	Floats	Immutable	Section 11.5
<code>bool</code>	Boolean values	Immutable	Section 11.6
<code>complex</code>	Complex numbers	Immutable	Section 11.7

\* All numeric data types are immutable. This doesn't mean that there are no operators that change numbers, but rather that a new instance of the respective data type must be created after each change. So from the programmer's perspective, there is hardly any difference at first. For more details on the difference between mutable and immutable data types, see Chapter 7, Section 7.3.

**Table 11.1** Numeric Data Types

The numeric data types form a group because they're related thematically. This relatedness is also reflected in the fact that the numeric data types have many operators in common. In the following sections, we'll cover these common operators, and then we'll discuss the `int`, `float`, `bool`, and `complex` numeric data types in detail.

### 11.1 Arithmetic Operators

An *arithmetic operator* is considered an operator that performs an arithmetic calculation—for example, addition or multiplication. For all numeric data types, the arithmetic operators listed in Table 11.2 are defined.

Operator	Result
<code>x + y</code>	Sum of <code>x</code> and <code>y</code>
<code>x - y</code>	Difference of <code>x</code> and <code>y</code>

**Table 11.2** Common Operators of Numeric Data Types

Operator	Result
$x * y$	Product of $x$ and $y$
$x / y$	Quotient of $x$ and $y$
$x \% y$	Remainder when dividing $x$ by $y$ *
$+x$	Positive sign
$-x$	Negative sign
$x ** y$	$x$ to the power of $y$
$x // y$	Rounded quotient of $x$ and $y$ *

\* The % and // operators have no mathematical meaning for complex numbers and are therefore not defined for the complex data type.

Table 11.2 Common Operators of Numeric Data Types (Cont.)

**Note**

Two notes for readers who are already familiar with C or a related programming language:

First, there are no equivalents in Python for the increment and decrement operators, ++ and --, from C.

Second, the % and // operators can be described as follows:

- $x // y = \text{round}(x / y)$
- $x \% y = x - y * \text{round}(x / y)$

Python always rounds down, while C rounds up to zero. This difference occurs only if the operands have opposite signs.

**Augmented Assignments**

Besides these basic operators, there are a number of additional operators in Python. Often, for example, you want to calculate the total of  $x$  and  $y$  and store the result in  $x$ —that is, increase  $x$  by  $y$ . This requires the following statement with the operators shown previously:

```
x = x + y
```

For such cases, Python provides so-called *augmented assignments*, which can be regarded as an abbreviated form of the preceding statement. An overview of augmented assignments in Python is presented in Table 11.3.

Operator	Equivalent
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
$x /= y$	$x = x / y$
$x \% y$	$x = x \% y$
$x **= y$	$x = x ** y$
$x //= y$	$x = x // y$

Table 11.3 Common Operators of Numeric Data Types

It's important to note that you can use any arithmetic expression for  $y$  here, while  $x$  must be an expression that could also be used as the target of a normal assignment such as a symbolic name or an item of a list or dictionary.

**11.2 Comparison Operators**

A *comparison operator* is an operator that computes a truth value from two instances. Table 11.4 lists the comparison operators defined for numeric data types.

Operator	Result
$x == y$	True if $x$ and $y$ are equal
$x != y$	True if $x$ and $y$ are different
$x < y$	True if $x$ is less than $y$ *
$x <= y$	True if $x$ is less than or equal to $y$ *
$x > y$	True if $x$ is greater than $y$ *
$x >= y$	True if $x$ is greater than or equal to $y$ *

\* Because complex numbers can't be arranged in a meaningful way, the complex data type only allows for using the first two operators.

Table 11.4 Common Operators of Numeric Data Types

Each of these comparison operators returns a truth value as a result. Such a value is expected, for example, as a condition of an if statement. So the operators could be used as follows:

```
if x < 4:
    print("x is less than 4")
```

You can concatenate as many of the comparison operators as you like into a series. Strictly speaking, the preceding example is only a special case of this rule, as it has only two operands. The meaning of such a concatenation corresponds to the mathematical view and can be seen in the following example:

```
if 2 < x < 4:
    print("x is between 2 and 4")
```

We'll describe Boolean values in more detail in Section 11.6.

### 11.3 Conversion between Numeric Data Types

Numeric data types can be converted into each other using the `int`, `float`, `bool`, and `complex` built-in functions. Note that in the process, information can be lost depending on the transformation. As an example, let's consider some conversions in interactive mode:

```
>>> float(33)
33.0
>>> int(33.5)
33
>>> bool(12)
True
>>> complex(True)
(1+0j)
```

Instead of a concrete literal, a reference can also be used or a reference can be linked to the resulting value:

```
>>> var1 = 12.5
>>> int(var1)
12
>>> var2 = int(40.25)
>>> var2
40
```

#### Note

The `complex` data type assumes a special role in the conversions presented here as it can't be reduced to a single numerical value in a meaningful way. For this reason, a conversion such as `int(1+2j)` fails.

So much for the general introduction to numeric data types. The following sections will cover each data type in this group in more detail.

### 11.4 Integers: int

For working with integers, there is the data type `int` in Python. Unlike many other programming languages, this data type is not subject to any principal limits in its value range, which makes dealing with large integers in Python very convenient.<sup>1</sup>

We've already done a lot of work with integers, so using `int` doesn't really need any more demonstration. Nevertheless, for the sake of completeness, here is a small example:

```
>>> i = 1234
>>> i
1234
>>> p = int(5678)
>>> p
5678
```

Since the release of Python 3.6, an underscore can be used to group the digits of a literal:

```
>>> 1_000_000
1000000
>>> 1_0_0
100
```

Grouping doesn't change the numerical value of the literal, but it serves to increase the readability of numerical literals. Whether and how you group the digits is up to you.

#### 11.4.1 Numeral Systems

Integers can be written in Python in multiple *numeral systems*:<sup>2</sup>

- Numbers written without a special prefix, as in the previous example, are interpreted in the *decimal system* (base 10). Note that such a number must not be preceded by leading zeros:

```
v_dez = 1337
```

<sup>1</sup> In Python 2, there were still two data types for integers: `int` for the limited number space of 32 bits or 64 bits, and `long` with an unlimited value range.

<sup>2</sup> If you don't know what a numeral system is, you can easily skip this section.

- The prefix `0o` ("zero-o") indicates a number written in the *octal system* (base 8). Note that only digits from 0 to 7 are allowed here:

```
v_oct = 0o2471
```

The lowercase `o` in the prefix can also be replaced with a capital `O`. However, we recommend that you always use a lowercase `o` because the uppercase `O` is almost indistinguishable from the zero in many fonts.

- The next and far more common variant is the *hexadecimal system* (base 16), which is identified by the prefix `0x` or `0X` ("zero-x"). The number itself may be formed from the digits 0–9 and the letters A–F or a–f:

```
v_hex = 0x5A3F
```

- In addition to the hexadecimal system, the *dual system*, also referred to as *binary system* (base 2), is of decisive importance in computer science. Numbers in the dual system are introduced by the prefix `0b`, similar to the preceding literals:

```
v_bin = 0b1101
```

In the dual system, only the digits 0 and 1 may be used.

However, you may not want to limit yourself to these four numeral systems explicitly supported by Python; you may want to use a more exotic one. Of course, Python doesn't have a separate literal for every possible numeral system. Instead, you can use the following notation:

```
v_6 = int("54425", 6)
```

This is an alternative method of creating an instance of the `int` data type and providing it with an initial value. For this purpose, a string containing the desired initial value in the selected numeral system and the base of this numeral system as an integer are written in the brackets. Both values must be separated by a comma. In the example, a base 6 system was used.

Python supports numeral systems with a base from 2 to 36. If a numeral system requires more than 10 different digits to represent a number, the letters A to Z of the English alphabet are used in addition to the digits 0 to 9.

The `v_6` variable now has the value 7505 in the decimal system.

For all numeral system literals, the use of a negative sign is possible:

```
>>> -1234
-1234
>>> -0o777
-511
>>> -0xFF
-255
>>> -0b1010101
-85
```

Note that the numeral systems are only alternate notations of the same values. The `int` data type, for example, doesn't switch to a kind of hexadecimal mode as soon as it contains such a value; the numeral system is only of importance for assignments or outputs. By default, all numbers are output in the decimal system:

```
>>> v1 = 0xFF
>>> v2 = 0o777
>>> v1
255
>>> v2
511
```

We'll return to how numbers can be output in other numeral systems later, in Chapter 12, Section 12.5.3 in the context of strings.

## 11.4.2 Bit Operations

As already mentioned, the dual system or binary system is of great importance in computer science. For the `int` data type, some additional operators are therefore defined that explicitly refer to the binary representation of the number. Table 11.5 summarizes these *bit operators*.

Operator	Augmented Assignment	Result
<code>x &amp; y</code>	<code>x &amp;= y</code>	Bitwise AND of x and y (AND)
<code>x   y</code>	<code>x  = y</code>	Bitwise nonexclusive OR of x and y (OR)
<code>x ^ y</code>	<code>x ^= y</code>	Bitwise exclusive OR of x and y (XOR)
<code>~x</code>		Bitwise complement of x
<code>x &lt;&lt; n</code>	<code>x &lt;&lt;= n</code>	Bit shift by n places to the left
<code>x &gt;&gt; n</code>	<code>x &gt;&gt;= n</code>	Bit shift by n places to the right

Table 11.5 Bit Operators of the `int` Data Type

Because it may not be immediately clear what the individual operations do, we'll describe them in detail ahead.

### Bitwise AND

The *bitwise AND* of two numbers is formed by linking both numbers in their binary representation bit by bit. The resulting number has a 1 in its binary representation exactly where both of the respective bits of the operands are 1, and a 0 in all other places. Figure 11.1 illustrates this.

Binary		Decimal	
1	1	0	1
0	1	0	1
&			
0	0	1	1
		0	0
		1	0
		0	0
		1	1
			9

**Figure 11.1** Bitwise AND

Let's now try out in the interactive mode of Python whether the bitwise AND with the operands selected in the graphic actually returns the expected result:

```
>>> 107 & 25
9
>>> 0b1101011 & 0b11001
9
>>> bin(0b1101011 & 0b11001)
'0b1001'
```

In the example we use the built-in function `bin` (see Chapter 17, Section 17.14.5) to represent the result of the bitwise AND in the binary system.

### Bitwise OR

The *bitwise OR* of two numbers is formed by comparing both numbers in their binary representation bit by bit. The resulting number has a 1 in its binary representation exactly where at least one of the respective bits of the operands is 1. Figure 11.2 illustrates this.

Binary		Decimal	
1	1	0	1
0	0	1	1
1	1	1	1
		0	1
		1	1
			123

**Figure 11.2** Bitwise Nonexclusive OR

We'll now try out in the interactive mode of Python whether the bitwise OR with the operands selected in the graphic actually returns the expected result:

```
>>> 107 | 25
123
>>> 0b1101011 | 0b11001
123
```

```
>>> bin(0b1101011 | 0b11001)
'0b1111011'
```

In the example we use the built-in function `bin` (see Chapter 17, Section 17.14.5) to represent the result of the bitwise OR in the binary system.

### Bitwise Exclusive OR

The *bitwise exclusive OR* (also *exclusive OR*) of two numbers is formed by comparing both numbers in their binary representation bit by bit. The resulting number has a 1 in its binary representation exactly where the respective bits of the operands differ from each other, and a 0 where they are the same. This is shown in Figure 11.3.

Binary		Decimal	
1	1	0	1
0	0	1	1
^			
1	1	1	0
		0	0
		1	1
		0	0
		1	0
			114

**Figure 11.3** Bitwise Exclusive OR

In the next step, we'll try out in the interactive mode of Python whether the bitwise exclusive OR with the operands selected in the graphic actually returns the expected result:

```
>>> 107 ^ 25
114
>>> 0b1101011 ^ 0b11001
114
>>> bin(0b1101011 ^ 0b11001)
'0b1110010'
```

In the example we use the built-in function `bin` (see Chapter 17, Section 17.14.5) to represent the result of the bitwise exclusive OR in the binary system.

### Bitwise Complement

The *bitwise complement* forms the so-called one's complement of a dual number, which corresponds to the negation of all occurring bits. In Python, this isn't possible at the bit level because an integer is unlimited in length and the complement must always be formed in a closed number space. For this reason, the actual bit operation becomes an arithmetic operation and is defined as follows:<sup>3</sup>

<sup>3</sup> This makes sense because the so-called *two's complement* is used to represent negative numbers in closed number spaces. This is obtained by adding 1 to the ones' complement. So:  $-x = \text{two's complement of } x = \sim x + 1$ . From this follows:  $\sim x = -x - 1$ .

```
~x = -x - 1
```

In the interactive mode, the functionality of the bitwise complement can be tested experimentally:

```
>>> ~9
-10
>>> ~0b1001
-10
>>> bin(~0b1001)
'-0b1010'
```

In the example we use the built-in function `bin` (see Chapter 17, Section 17.14.5) to represent the result of the bitwise complement in the binary system.

### Bit Shift

The *bit shift* is used to shift the bit sequence in the binary representation of the first operand to the left or right by the number of places provided by the second operand. Any gaps that occur on the right-hand side are filled with zeros, and the sign of the first operand is retained. Figure 11.4 and Figure 11.5 illustrate a shift of two places to the left and to the right, respectively.

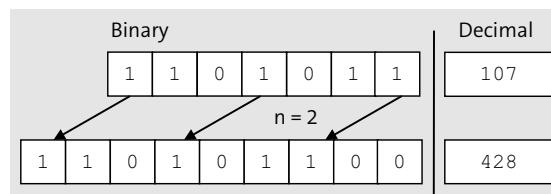


Figure 11.4 Bit Shift by Two Places to the Left

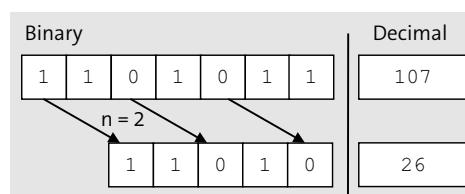


Figure 11.5 Bit Shift by Two Places to the Right

The gaps that occur in the bit representation on the right- or left-hand side are filled with zeros.

The bit shift is implemented arithmetically in Python, similar to the complement operator. A shift by  $x$  places to the right corresponds to an integer division by  $2^x$ . A shift by  $x$  places to the left corresponds to a multiplication by  $2^x$ .

For the bitwise shifts too, we can follow the examples shown in the graphics in the interactive mode:

```
>>> 107 << 2
428
>>> 107 >> 2
26
>>> bin(0b1101011 << 2)
'0b110101100'
>>> bin(0b1101011 >> 2)
'0b11010'
```

In the example we use the built-in function `bin` (see Chapter 17, Section 17.14.5) to represent the result of the bit shifts in the binary system.

### 11.4.3 Methods

The `int` data type has a method that refers to the binary representation of the integer. The `bit_length` method calculates the number of digits needed for the binary representation of the number:

```
>>> (36).bit_length()
6
>>> (4345).bit_length()
13
```

The binary representation of 36 is 100100, and that of 4345 is 100001111001. Thus, the two numbers require 6 and 13 digits, respectively, for their binary representation.

#### Note

Note that the parentheses around the number literals are required for integers; otherwise there could be ambiguities with regard to the syntax for floats.

### 11.5 Floats: float

We mentioned floats briefly earlier. Now we'd like to describe them in greater detail. To store a float with limited precision,<sup>4</sup> you can use the `float` data type.

As discussed previously, the literal for a float in the simplest case looks like this:

```
v = 3.141
```

<sup>4</sup> See Section 11.5.2 for further remarks on the precision.

The parts before and after the period can be omitted if they have the value 0:

```
>>> -3.  
-3.0  
>>> .001  
0.001
```

Note here that the period is an essential element of a float literal and as such must not be omitted.

Since Python 3.6, an underscore can also be used to group the digits of a float literal:

```
>>> 3.000_000_1  
3.0000001
```

### 11.5.1 Exponential Notation

Python also supports a notation that allows you to use the exponential notation:

```
v = 3.141e-12
```

A lowercase or uppercase e separates the *mantissa* (3.141) from the *exponent* (-12). Translated into mathematical notation, this corresponds to the value  $3.141 \cdot 10^{-12}$ . Note that both the mantissa and the exponent must be specified in the decimal system. As no other numeral systems are supported, it's safe to use leading zeros:

```
v = 03.141e-0012
```

### 11.5.2 Precision

You may have just experimented a bit with floats and encountered a supposed error in the interpreter:

```
>>> 1.1 + 2.2  
3.3000000000000003
```

Real numbers can't be stored with infinite precision in the float data type, but are instead approximated.

Technically savvy people and those switching from other programming languages will be interested to know that float instances in Python are IEEE-754 floats with double precision. The float data type in Python is thus comparable to the double data type in C, C++, and Java.

If you want to explicitly use single precision floats, you can draw on the float32 data type of the third-party NumPy library (see Chapter 43).

### 11.5.3 Infinite and Not a Number

The precision of float is limited. This also implies that there must be both an upper and lower limit for this data type. And indeed, floats that exceed a certain limit in size can no longer be represented in Python. If the limit is exceeded, the number is stored as inf<sup>5</sup> or as -inf if the number has fallen below the lower limit. So there is no error, and it's still possible to compare a number that's too high with others:

```
>>> 3.0e999  
inf  
>>> -3.0e999  
-inf  
>>> 3.0e999 < 12.0  
False  
>>> 3.0e999 > 12.0  
True  
>>> 3.0e999 == 3.0e999999999999  
True
```

Although it's possible to compare two infinitely large floats with each other, you can only use them to a limited extent for calculations. Let's look at the following example:

```
>>> 3.0e999 + 1.5e999999  
inf  
>>> 3.0e999 - 1.5e999999  
nan  
>>> 3.0e999 * 1.5e999999  
inf  
>>> 3.0e999 / 1.5e999999  
nan  
>>> 5 / 1e9999  
0.0
```

Two infinite floats can be easily added or multiplied. The result in both cases is again inf. However, there's a problem when trying to subtract or divide two such numbers. Because these arithmetic operations don't make sense, they result in nan. The nan status is comparable to inf, but it means "not a number"—that is, not calculable.

Note that neither inf nor nan is a constant you could use yourself in a Python program. Instead, you can create float instances with the values inf and nan as follows:

```
>>> float("inf")  
inf  
>>> float("nan")  
nan
```

<sup>5</sup> Here, inf stands for *infinity*.

```
>>> float("inf") / float("inf")
nan
```

## 11.6 Boolean Values: bool

An instance of the `bool`<sup>6</sup> data type can have only two different values: `true` or `false`—or, to stay within the Python syntax, `True` or `False`. For this reason, it's absurd to categorize `bool` as a numeric data type at first glance. As is common usage in many programming languages, in Python, `True` is regarded as similar to `1` and `False` as similar to `0`, so that Boolean values can be calculated in the same way as, for example, integers. The names `True` and `False` are constants that can be used in the source code. Note especially that the constants start with an uppercase letter:

```
v1 = True
v2 = False
```

### 11.6.1 Logical Operators

One or more Boolean values can be combined into a Boolean expression using certain operators. When evaluated, such an expression results again in a Boolean value—that is, `True` or `False`. Before it gets too theoretical, Table 11.6 shows the so-called logical operators.<sup>7</sup> We'll follow that with further explanations and concrete examples.

Operator	Result
<code>not x</code>	Logical negation of <code>x</code>
<code>x and y</code>	Logical AND between <code>x</code> and <code>y</code>
<code>x or y</code>	Logical (nonexclusive) OR between <code>x</code> and <code>y</code>

Table 11.6 Logical Operators of the `bool` Data Type

#### Logical Negation

The *logical negation* of a Boolean value can be quickly explained: the corresponding `not` operator turns `True` into `False` and `False` into `True`. In a concrete example, this would look as follows:

```
if not x:
    print("x is False")
else:
    print("x is True")
```

<sup>6</sup> The name `bool` goes back to the British mathematician and logician George Boole (1815–1864).

<sup>7</sup> Note that there's a difference between logical operators, which refer to Boolean values, and binary operators, which refer to the binary representation of a number.

#### Logical AND

The *logical AND* between two truth values only returns `True` if both operands are already `True`. Table 11.7 lists all possible cases.

x	y	x and y
True	True	True
False	True	False
True	False	False
False	False	False

Table 11.7 Possible Cases of Logical AND

In a concrete example, the application of logical AND would look as follows:

```
if x and y:
    print("x and y are True")
```

#### Logical OR

The *logical OR* between two truth values results in a true statement if and only if at least one of the two operands is true. Accordingly, it's a nonexclusive OR. An operator for a logical exclusive OR doesn't exist in Python.<sup>8</sup> Table 11.8 lists all possible cases.

x	y	x or y
True	True	True
False	True	True
True	False	True
False	False	False

Table 11.8 Possible Cases of Logical OR

A logical OR could be implemented as follows:

```
if x or y:
    print("x or y is True")
```

Of course, you can combine all these operators and use them in a complex expression. This could look something like this:

```
if x and y or ((y and z) and not x):
    print("Holy cow")
```

<sup>8</sup> A logical exclusive OR between `x` and `y` can be modeled using `(x or y)` and `not (x and y)`.

At this point, we don't want to discuss this expression in further detail. Suffice to say that the use of parentheses has the expected effect—namely, that expressions in parentheses are evaluated first. Table 11.9 shows the truth value of the expression, as a function of the three parameters *x*, *y*, and *z*.

<b>x</b>	<b>y</b>	<b>z</b>	<b>x and y or ((y and z) and not x)</b>
True	True	True	True
False	True	True	True
True	False	True	False
True	True	False	True
False	False	True	False
False	True	False	False
True	False	False	False
False	False	False	False

**Table 11.9** Possible Results of the Expression

### The Combination of Logical Operators and Comparison Operators

At the beginning of the section on numeric data types, we introduced some comparison operators that yield a truth statement as a Boolean value. The following example shows that they can be used quite normally in combination with the logical operators:

```
if x > y or (y > z and x != 0):
    print("My goodness")
```

In this case, *x*, *y*, and *z* must be instances of comparable types, such as `int`, `float`, or `bool`.

### 11.6.2 Truth Values of Non-Boolean Data Types

Instances of any basic data type can be converted to a Boolean value using the built-in `bool` function:

```
>>> bool([1,2,3])
True
>>> bool("")
False
>>> bool(-7)
True
```

This is a useful property because an instance of the basic data types can often be in one of two stages: "empty" and "nonempty." For example, it often happens that you want to test whether a string contains letters or not. Because a string can be converted to a Boolean value, such a test is made very easy by logical operators:

```
>>> not ""
True
>>> not "abc"
False
```

By using a logical operator, the operand is automatically interpreted as a truth value.

For each basic data type, a specific value is defined as `False`. All other values are `True`. Table 11.10 lists the corresponding `False` value for each data type. Some of the data types haven't been introduced yet, but you shouldn't worry about that at this point.

Basic Data Type	False Value	Description
<code>NoneType</code>	<code>None</code>	The <code>None</code> value
<b>Numeric Data Types</b>		
<code>int</code>	<code>0</code>	The numeric value zero
<code>float</code>	<code>0.0</code>	The numeric value zero
<code>bool</code>	<code>False</code>	The boolean value <code>False</code>
<code>complex</code>	<code>0 + 0j</code>	The numeric value zero
<b>Sequential Data Types</b>		
<code>str</code>	<code>" "</code>	An empty string
<code>list</code>	<code>[]</code>	An empty list
<code>tuple</code>	<code>()</code>	An empty tuple
<b>Associative Data Types</b>		
<code>dict</code>	<code>{}</code>	An empty dictionary
<b>Quantities</b>		
<code>set</code>	<code>set()</code>	An empty set
<code>frozenset</code>	<code>frozenset()</code>	An empty set

**Table 11.10** Truth Values of the Basic Data Types

All other values result in `True`.

### 11.6.3 Evaluating Logical Operators

Python evaluates logical expressions basically from left to right—so in the following example, first `a`, then `b`:

```
if a or b:
    print("a or b are True")
```

However, it isn't guaranteed that every part of the expression will actually be evaluated. For optimization reasons, Python immediately terminates the evaluation of the expression when the result has been obtained. So, in the preceding example, if `a` already has the value `True`, the value of `b` is of no further concern; `b` would then no longer be evaluated. The following example demonstrates this behavior, which is referred to as *lazy evaluation*:

```
>>> a = True
>>> if a or print("Lazy "):
...     print("Evaluation")
...
Evaluation
```

Although the `print` function is called in the condition of the `if` statement, this screen output is never performed because the value of the condition is already certain after the evaluation of `a`. This detail seems unimportant, but it can lead to errors that are hard to find, especially in the context of functions with side effects.<sup>9</sup>

In Section 11.6.1, we mentioned that a Boolean expression always yields a Boolean value when evaluated. This is not quite correct because here too, the interpreter's way of working has been optimized in a way you should be aware of. This is clearly illustrated by the following example from the interactive mode:

```
>>> 0 or 1
1
```

From what we have discussed so far, the result of the expression should be `True`, which is not the case. Instead, Python returns the first operand here with the truth value `True`. In many cases this does not make a difference, because the returned value is automatically converted to the truth value `True` without any problem.

The evaluation of the two operators `or` and `and` works as follows: The logical OR (`or`) takes the value of the first operand that has the truth value `True`, or—if there is no such operand—the value of the last operand. The logical AND (`and`) takes the value of the first operand that has the truth value `False`, or—if there is no such value—the value of the last operand.

<sup>9</sup> See Chapter 17, Section 17.10.

However, these details also have an entertaining value:

```
>>> "Python" or "Java"
'Python'
```

### 11.7 Complex Numbers: complex

Surprisingly, there is a data type for storing complex numbers among the basic data types of Python. In many programming languages, complex numbers would be more of a side note in the standard library or left out altogether. If you aren't familiar with complex numbers, you can safely skip this section. It doesn't cover anything that would be required for further learning Python.

Complex numbers consist of a real part and an imaginary part. The imaginary part is a real number multiplied by the imaginary unit  $j$ .<sup>10</sup> The imaginary unit  $j$  is defined as the solution of the following equation:

$$j^2 = -1$$

In the following example, we assign the name `v` to a complex number:

```
v = 4j
```

If you specify only an imaginary part, as in the example, the real part is automatically assumed to be 0. To determine the real part, it is added to the imaginary part. The following two notations are equivalent:

```
v1 = 3 + 4j
v2 = 4j + 3
```

Instead of a lowercase `j`, you can also use an uppercase `J` as a literal for the imaginary part of a complex number. It's entirely up to your preferences which of the two options you want to use.

Both the real and imaginary parts can be any real number. The following notation is therefore also correct:

```
v3 = 3.4 + 4e2j
```

At the beginning of the section on numeric data types, we already indicated that complex numbers differ from the other numeric data types. Since no mathematical ordering is defined for complex numbers, instances of the `complex` data type can only be checked for equality or inequality. The set of comparison operators is thus limited to `==` and `!=`.

<sup>10</sup> The symbol of the imaginary unit, which is actually common in mathematics, is  $i$ . Python adheres to the notations of electrical engineering here.

Furthermore, both the % modulo operator and the // operator for integer division have no mathematical sense and are therefore not available in combination with complex numbers.

The `complex` data type has two attributes that make it easier to use it. For example, it can happen that you want to make calculations only with the real part or only with the imaginary part of the stored number. To isolate one of the two parts, a `complex` instance provides the attributes listed in Table 11.11.

Attribute	Description
<code>x.real</code>	Real part of <code>x</code> as a float
<code>x.imag</code>	Imaginary part of <code>x</code> as a float

**Table 11.11** Attributes of the complex Data Type

These can be used as shown in the following example:

```
>>> c = 23 + 4j
>>> c.real
23.0
>>> c.imag
4.0
```

In addition to its two attributes, the `complex` data type has a method, which is explained in Table 11.12 as an example of a reference to a complex number called `x`.

Method	Description
<code>x.conjugate()</code>	Returns the complex number conjugated to <code>x</code>

**Table 11.12** Methods of the complex Data Type

The following example demonstrates the use of the `conjugate` method:

```
>>> c = 23 + 4j
>>> c.conjugate()
(23-4j)
```

The result of `conjugate` is again a complex number and therefore also has the `conjugate` method:

```
>>> c = 23 + 4j
>>> c2 = c.conjugate()
>>> c2
(23-4j)
```

```
>>> c3 = c2.conjugate()
>>> c3
(23+4j)
```

Conjugating a complex number is a self-inverse operation. This means that the result of a double conjugation is again the initial number.

# Contents

<b>1</b>	<b>Introduction</b>	33
1.1	Why Did We Write This Book? .....	33
1.2	What Does This Book Provide? .....	34
1.3	Structure of the Book .....	34
1.4	How Should You Read This Book? .....	35
1.5	Sample Programs .....	36
1.6	Preface To the First English Edition (2022) .....	36
1.7	Acknowledgments .....	37
<b>2</b>	<b>The Python Programming Language</b>	39
2.1	History, Concepts, and Areas of Application .....	39
2.1.1	History and Origin .....	39
2.1.2	Basic Concepts .....	40
2.1.3	Possible Areas of Use and Strengths .....	41
2.1.4	Examples of Use .....	42
2.2	Installing Python .....	42
2.2.1	Installing Anaconda on Windows .....	43
2.2.2	Installing Anaconda on Linux .....	43
2.2.3	Installing Anaconda on macOS .....	44
2.3	Installing Third-Party Modules .....	45
2.4	Using Python .....	45

## Part I Getting Started with Python

<b>3</b>	<b>Getting Started with the Interactive Mode</b>	49
3.1	Integers .....	49
3.2	Floats .....	51
3.3	Character Strings .....	51

<b>3.4</b>	<b>Lists</b>	52
<b>3.5</b>	<b>Dictionaries</b>	53
<b>3.6</b>	<b>Variables</b>	54
3.6.1	The Special Meaning of the Underscore	54
3.6.2	Identifiers	55
<b>3.7</b>	<b>Logical Expressions</b>	55
<b>3.8</b>	<b>Functions and Methods</b>	57
3.8.1	Functions	57
3.8.2	Methods	58
<b>3.9</b>	<b>Screen Outputs</b>	59
<b>3.10</b>	<b>Modules</b>	60

## 4 The Path to the First Program

<b>4.1</b>	<b>Typing, Compiling, and Testing</b>	63
4.1.1	Windows	63
4.1.2	Linux and macOS	64
4.1.3	Shebang	65
4.1.4	Internal Processes	65
<b>4.2</b>	<b>Basic Structure of a Python Program</b>	66
4.2.1	Wrapping Long Lines	68
4.2.2	Joining Multiple Lines	69

<b>4.3</b>	<b>The First Program</b>	70
4.3.1	Initialization	71
4.3.2	Loop Header	71
4.3.3	Loop Body	71
4.3.4	Screen Output	72

<b>4.4</b>	<b>Comments</b>	72
<b>4.5</b>	<b>In Case of Error</b>	72

## 5 Control Structures

<b>5.1</b>	<b>Conditionals</b>	75
5.1.1	The if Statement	75
5.1.2	Conditional Expressions	78

<b>5.2</b>	<b>Loops</b>	79
5.2.1	The while Loop	79
5.2.2	Termination of a Loop	80
5.2.3	Detecting a Loop Break	81
5.2.4	Aborting the Current Iteration	82
5.2.5	The for Loop	84
5.2.6	The for Loop as a Counting Loop	85
<b>5.3</b>	<b>The pass Statement</b>	87
<b>5.4</b>	<b>Assignment Expressions</b>	87
5.4.1	The Guessing Numbers Game with Assignment Expressions	89

## 6 Files

<b>6.1</b>	<b>Data Streams</b>	91
<b>6.2</b>	<b>Reading Data from a File</b>	92
6.2.1	Opening and Closing a File	92
6.2.2	The with Statement	93
6.2.3	Reading the File Content	94
<b>6.3</b>	<b>Writing Data to a File</b>	96
<b>6.4</b>	<b>Generating the File Object</b>	97
6.4.1	The Built-In open Function	97
6.4.2	Attributes and Methods of a File Object	99
6.4.3	Changing the Write/Read Position	100

## 7 The Data Model

<b>7.1</b>	<b>The Structure of Instances</b>	105
7.1.1	Data Type	106
7.1.2	Value	107
7.1.3	Identity	108
<b>7.2</b>	<b>Deleting References</b>	109
<b>7.3</b>	<b>Mutable versus Immutable Data Types</b>	111

---

<b>8 Functions, Methods, and Attributes</b>	115
<b>8.1 Parameters of Functions and Methods</b>	115
8.1.1 Positional Parameters	116
8.1.2 Keyword Arguments	116
8.1.3 Optional Parameters	117
8.1.4 Keyword-Only Parameters	117
<b>8.2 Attributes</b>	118
<b>9 Sources of Information on Python</b>	119
<b>9.1 The Built-In Help Function</b>	119
<b>9.2 The Online Documentation</b>	120
<b>9.3 PEPs</b>	120
<b>Part II Data Types</b>	
<b>10 Basic Data Types: An Overview</b>	125
<b>10.1 Nothingness: NoneType</b>	126
<b>10.2 Operators</b>	127
10.2.1 Operator Precedence	127
10.2.2 Evaluation Order	129
10.2.3 Concatenating Comparisons	129
<b>11 Numeric Data Types</b>	131
<b>11.1 Arithmetic Operators</b>	131
<b>11.2 Comparison Operators</b>	133
<b>11.3 Conversion between Numeric Data Types</b>	134
<b>11.4 Integers: int</b>	135
11.4.1 Numeral Systems	135

---

<b>11.4.2 Bit Operations</b>	137
<b>11.4.3 Methods</b>	141
<b>11.5 Floats: float</b>	141
<b>11.5.1 Exponential Notation</b>	142
<b>11.5.2 Precision</b>	142
<b>11.5.3 Infinite and Not a Number</b>	143
<b>11.6 Boolean Values: bool</b>	144
<b>11.6.1 Logical Operators</b>	144
<b>11.6.2 Truth Values of Non-Boolean Data Types</b>	146
<b>11.6.3 Evaluating Logical Operators</b>	148
<b>11.7 Complex Numbers: complex</b>	149
<b>12 Sequential Data Types</b>	153
<b>12.1 The Difference between Text and Binary Data</b>	153
<b>12.2 Operations on Instances of Sequential Data Types</b>	154
12.2.1 Checking for Elements	155
12.2.2 Concatenation	157
12.2.3 Repetition	158
12.2.4 Indexing	159
12.2.5 Slicing	160
12.2.6 Length of a Sequence	164
12.2.7 The Smallest and the Largest Element	164
12.2.8 Searching for an Element	165
12.2.9 Counting Elements	166
<b>12.3 The list Data Type</b>	166
12.3.1 Changing a Value within the List: Assignment via []	167
12.3.2 Replacing Sublists and Inserting New Elements: Assignment via []	167
12.3.3 Deleting Elements and Sublists: del in Combination with []	168
12.3.4 Methods of list Instances	169
12.3.5 Sorting Lists: s.sort([key, reverse])	171
12.3.6 Side Effects	174
12.3.7 List Comprehensions	177
<b>12.4 Immutable Lists: tuple</b>	179
12.4.1 Packing and Unpacking	179
12.4.2 Immutable Doesn't Necessarily Mean Unchangeable!	181

<b>12.5 Strings: str, bytes, bytearray .....</b>	182
12.5.1 Control Characters .....	185
12.5.2 String Methods .....	187
12.5.3 Formatting Strings .....	196
12.5.4 Character Sets and Special Characters .....	207

## 13 Mappings and Sets

---

<b>13.1 Dictionary: dict .....</b>	215
13.1.1 Creating a Dictionary .....	215
13.1.2 Keys and Values .....	216
13.1.3 Iteration .....	218
13.1.4 Operators .....	219
13.1.5 Methods .....	221
13.1.6 Dict Comprehensions .....	227
<b>13.2 Sets: set and frozenset .....</b>	227
13.2.1 Creating a Set .....	228
13.2.2 Iteration .....	229
13.2.3 Operators .....	230
13.2.4 Methods .....	235
13.2.5 Mutable Sets: set .....	236
13.2.6 Immutable Sets: frozenset .....	237

## 14 Collections

---

<b>14.1 Chained Dictionaries .....</b>	239
<b>14.2 Counting Frequencies .....</b>	240
<b>14.3 Dictionaries with Default Values .....</b>	242
<b>14.4 Doubly Linked Lists .....</b>	243
<b>14.5 Named Tuples .....</b>	245

## 15 Date and Time

---

<b>15.1 Elementary Time Functions—time .....</b>	247
15.1.1 The struct_time Data Type .....	248
15.1.2 Constants .....	249
15.1.3 Functions .....	250
<b>15.2 Object-Oriented Date Management: datetime .....</b>	254
15.2.1 datetime.date .....	255
15.2.2 datetime.time .....	256
15.2.3 datetime.datetime .....	257
15.2.4 datetime.timedelta .....	259
15.2.5 Operations for datetime.datetime and datetime.date .....	262
<b>15.3 Time Zones: zoneinfo .....</b>	263
15.3.1 The IANA Time Zone Database .....	263
15.3.2 Specifying the Time in Local Time Zones .....	265
15.3.3 Calculating with Time Indications in Local Time Zones .....	265

## 16 Enumerations and Flags

---

<b>16.1 Enumeration Types: enum .....</b>	269
<b>16.2 Enumeration Types for Bit Patterns: flag .....</b>	271
<b>16.3 Integer Enumeration Types: IntEnum .....</b>	272

## Part III Advanced Programming Techniques

## 17 Functions

---

<b>17.1 Defining a Function .....</b>	278
<b>17.2 Return Values .....</b>	280
<b>17.3 Function Objects .....</b>	282
<b>17.4 Optional Parameters .....</b>	282
<b>17.5 Keyword Arguments .....</b>	283
<b>17.6 Arbitrarily Many Parameters .....</b>	284

---

<b>17.7 Keyword-Only Parameters .....</b>	286
<b>17.8 Positional-Only Parameters .....</b>	287
<b>17.9 Unpacking When Calling a Function .....</b>	288
<b>17.10 Side Effects .....</b>	290
<b>17.11 Namespaces .....</b>	293
17.11.1 Accessing Global Variables: global .....	293
17.11.2 Accessing the Global Namespace .....	294
17.11.3 Local Functions .....	295
17.11.4 Accessing Parent Namespaces: nonlocal .....	296
17.11.5 Unbound Local Variables: A Stumbling Block .....	297
<b>17.12 Anonymous Functions .....</b>	299
<b>17.13 Recursion .....</b>	300
<b>17.14 Built-In Functions .....</b>	300
17.14.1 abs(x) .....	304
17.14.2 all(iterable) .....	304
17.14.3 any(iterable) .....	305
17.14.4 ascii(object) .....	305
17.14.5 bin(x) .....	305
17.14.6 bool([x]) .....	306
17.14.7 bytearray([source, encoding, errors]) .....	306
17.14.8 bytes([source, encoding, errors]) .....	307
17.14.9 chr(i) .....	307
17.14.10 complex([real, imag]) .....	307
17.14.11 dict([source]) .....	308
17.14.12 divmod(a, b) .....	309
17.14.13 enumerate(iterable, [start]) .....	309
17.14.14 eval(expression, [globals, locals]) .....	309
17.14.15 exec(object, [globals, locals]) .....	310
17.14.16 filter(function, iterable) .....	310
17.14.17 float([x]) .....	311
17.14.18 format(value, [format_spec]) .....	311
17.14.19 frozenset([iterable]) .....	311
17.14.20 globals() .....	312
17.14.21 hash(object) .....	312
17.14.22 help([object]) .....	313
17.14.23 hex(x) .....	313
17.14.24 id(object) .....	313
17.14.25 input([prompt]) .....	314
17.14.26 int([x, base]) .....	314

17.14.27 len(s) .....	315
17.14.28 list([sequence]) .....	315
17.14.29 locals() .....	315
17.14.30 map(function, [*iterable]) .....	316
17.14.31 max(iterable, {default, key}), max(arg1, arg2, [*args], {key}) .....	317
17.14.32 min(iterable, {default, key}), min(arg1, arg2, [*args], {key}) .....	318
17.14.33 oct(x) .....	318
17.14.34 ord(c) .....	318
17.14.35 pow(x, y, [z]) .....	318
17.14.36 print([*objects], {sep, end, file, flush}) .....	318
17.14.37 range([start], stop, [step]) .....	319
17.14.38 repr(object) .....	320
17.14.39 reversed(sequence) .....	320
17.14.40 round(x, [n]) .....	321
17.14.41 set([iterable]) .....	321
17.14.42 sorted(iterable, [key, reverse]) .....	321
17.14.43 str([object, encoding, errors]) .....	322
17.14.44 sum(iterable, [start]) .....	323
17.14.45 tuple([iterable]) .....	323
17.14.46 type(object) .....	323
17.14.47 zip([*iterables], {strict}) .....	324

## 18 Modules and Packages

---

<b>18.1 Importing Global Modules .....</b>	326
<b>18.2 Local Modules .....</b>	328
18.2.1 Name Conflicts .....	329
18.2.2 Module-Internal References .....	330
18.2.3 Executing Modules .....	330
<b>18.3 Packages .....</b>	331
18.3.1 Importing All Modules of a Package .....	333
18.3.2 Namespace Packages .....	333
18.3.3 Relative Import Statements .....	334
<b>18.4 The importlib Package .....</b>	335
18.4.1 Importing Modules and Packages .....	335
18.4.2 Changing the Import Behavior .....	335
<b>18.5 Planned Language Elements .....</b>	338

---

<b>19 Object-Oriented Programming</b>	341
<b>19.1 Example: A Non-Object-Oriented Account</b>	341
19.1.1 Creating a New Account	342
19.1.2 Transferring Money	342
19.1.3 Depositing and Withdrawing Money	343
19.1.4 Viewing the Account Balance	344
<b>19.2 Classes</b>	346
19.2.1 Defining Methods	347
19.2.2 The Constructor	348
19.2.3 Attributes	349
19.2.4 Example: An Object-Oriented Account	349
<b>19.3 Inheritance</b>	351
19.3.1 A Simple Example	352
19.3.2 Overriding Methods	353
19.3.3 Example: Checking Account with Daily Turnover	355
19.3.4 Outlook	363
<b>19.4 Multiple Inheritance</b>	363
<b>19.5 Property Attributes</b>	365
19.5.1 Setters and Getters	365
19.5.2 Defining Property Attributes	366
<b>19.6 Static Methods</b>	367
<b>19.7 Class Methods</b>	369
<b>19.8 Class Attributes</b>	370
<b>19.9 Built-in Functions for Object-Oriented Programming</b>	370
19.9.1 Functions for Managing the Attributes of an Instance	371
19.9.2 Functions for Information about the Class Hierarchy	372
<b>19.10 Inheriting Built-In Data Types</b>	373
<b>19.11 Magic Methods and Magic Attributes</b>	375
19.11.1 General Magic Methods	375
19.11.2 Overloading Operators	382
19.11.3 Emulating Data Types: Duck Typing	388
<b>19.12 Data Classes</b>	393
19.12.1 Tuples and Lists	393
19.12.2 Dictionaries	394
19.12.3 Named Tuples	394
19.12.4 Mutable Data Classes	395

---

19.12.5 Immutable Data Classes	396
19.12.6 Default Values in Data Classes	396

---

## 20 Exception Handling

<b>20.1 Exceptions</b>	399
20.1.1 Built-In Exceptions	400
20.1.2 Raising an Exception	401
20.1.3 Handling an Exception	401
20.1.4 Custom Exceptions	406
20.1.5 Re-Raising an Exception	408
20.1.6 Exception Chaining	410
<b>20.2 Assertions</b>	411
<b>20.3 Warnings</b>	412

---

## 21 Generators and Iterators

<b>21.1 Generators</b>	415
21.1.1 Subgenerators	418
21.1.2 Generator Expressions	421
<b>21.2 Iterators</b>	422
21.2.1 The Iterator Protocol	422
21.2.2 Example: The Fibonacci Sequence	423
21.2.3 Example: The Golden Ratio	424
21.2.4 A Generator for the Implementation of <code>__iter__</code>	424
21.2.5 Using Iterators	425
21.2.6 Multiple Iterators for the Same Instance	428
21.2.7 Disadvantages of Iterators Compared to Direct Access via Indexes	430
21.2.8 Alternative Definition for Iterable Objects	430
21.2.9 Function Iterators	431
<b>21.3 Special Generators: <code>itertools</code></b>	432
21.3.1 <code>accumulate(iterable, [func])</code>	433
21.3.2 <code>chain([*iterables])</code>	433
21.3.3 <code>combinations(iterable, r)</code>	434
21.3.4 <code>combinations_with_replacement(iterable, r)</code>	434
21.3.5 <code>compress(data, selectors)</code>	435
21.3.6 <code>count([start, step])</code>	435

21.3.7	cycle(iterable) .....	436
21.3.8	dropwhile(predicate, iterable) .....	436
21.3.9	filterfalse(predicate, iterable) .....	436
21.3.10	groupby(iterable, [key]) .....	437
21.3.11	islice(iterable, [start], stop, [step]) .....	437
21.3.12	permutations(iterable, [r]) .....	438
21.3.13	product([*iterables], [repeat]) .....	438
21.3.14	repeat(object, [times]) .....	439
21.3.15	starmap(function, iterable) .....	439
21.3.16	takewhile(predicate, iterable) .....	439
21.3.17	tee(iterator, [n]) .....	439
21.3.18	zip_longest([*iterables], {fillvalue}) .....	440

## 22 Context Manager

22.1	The with Statement .....	441
22.1.1	__enter__(self) .....	443
22.1.2	__exit__(self, exc_type, exc_value, traceback) .....	444
22.2	Helper Functions for with Contexts: contextlib .....	444
22.2.1	Dynamically Assembled Context Combinations - ExitStack .....	444
22.2.2	Suppressing Certain Exception Types .....	445
22.2.3	Redirecting the Standard Output Stream .....	445
22.2.4	Optional Contexts .....	446
22.2.5	Simple Functions as Context Manager .....	447

## 23 Decorators

23.1	Function Decorators .....	449
23.1.1	Decorating Functions and Methods .....	451
23.1.2	Name and Docstring after Applying a Decorator .....	451
23.1.3	Nested Decorators .....	452
23.1.4	Example: A Cache Decorator .....	452
23.2	Class Decorators .....	454
23.3	The functools Module .....	455
23.3.1	Simplifying Function Interfaces .....	455
23.3.2	Simplifying Method Interfaces .....	457

23.3.3	Caches .....	457
23.3.4	Completing Orderings of Custom Classes .....	459
23.3.5	Overloading Functions .....	459

## 24 Annotations for Static Type Checking

24.1	Annotations .....	464
24.1.1	Annotating Functions and Methods .....	465
24.1.2	Annotating Variables and Attributes .....	466
24.1.3	Accessing Annotations at Runtime .....	468
24.1.4	When are Annotations Evaluated? .....	470
24.2	Type Hints: The typing Module .....	471
24.2.1	Valid Type Hints .....	472
24.2.2	Container Types .....	472
24.2.3	Abstract Container Types .....	473
24.2.4	Type Aliases .....	474
24.2.5	Type Unions and Optional Values .....	474
24.2.6	Type Variables .....	475
24.3	Static Type Checking in Python: mypy .....	476
24.3.1	Installation .....	476
24.3.2	Example .....	477

## 25 Structural Pattern Matching

25.1	The match Statement .....	479
25.2	Pattern Types in the case Statement .....	480
25.2.1	Literal and Value Patterns .....	481
25.2.2	OR Pattern .....	481
25.2.3	Patterns with Type Checking .....	482
25.2.4	Specifying Conditions for Matches .....	483
25.2.5	Grouping Subpatterns .....	483
25.2.6	Capture and Wildcard Patterns .....	484
25.2.7	Sequence Patterns .....	486
25.2.8	Mapping Patterns .....	488
25.2.9	Patterns for Objects and Their Attribute Values .....	490

## Part IV The Standard Library

<b>26 Mathematics</b>	497
<b>26.1 Mathematical Functions: math, cmath</b>	497
26.1.1 General Mathematical Functions	498
26.1.2 Exponential and Logarithm Functions	501
26.1.3 Trigonometric and Hyperbolic Functions	501
26.1.4 Distances and Norms	502
26.1.5 Converting Angles	502
26.1.6 Representations of Complex Numbers	502
<b>26.2 Random Number Generator: random</b>	503
26.2.1 Saving and Loading the Random State	504
26.2.2 Generating Random Integers	504
26.2.3 Generating Random Floats	505
26.2.4 Random Operations on Sequences	505
26.2.5 SystemRandom([seed])	507
<b>26.3 Statistical Calculations: statistics</b>	507
<b>26.4 Intuitive Decimal Numbers: decimal</b>	509
26.4.1 Using the Data Type	509
26.4.2 Nonnumeric Values	512
26.4.3 The Context Object	513
<b>26.5 Hash Functions: hashlib</b>	514
26.5.1 Using the Module	516
26.5.2 Other Hash Algorithms	517
26.5.3 Comparing Large Files	517
26.5.4 Passwords	518
<b>27 Screen Outputs and Logging</b>	521
<b>27.1 Formatted Output of Complex Objects: pprint</b>	521
<b>27.2 Log Files: logging</b>	523
27.2.1 Customizing the Message Format	525
27.2.2 Logging Handlers	527

## 28 Regular Expressions

<b>28.1 Syntax of Regular Expressions</b>	529
28.1.1 Any Character	530
28.1.2 Character Classes	530
28.1.3 Quantifiers	531
28.1.4 Predefined Character Classes	533
28.1.5 Other Special Characters	534
28.1.6 Nongreedy Quantifiers	535
28.1.7 Groups	536
28.1.8 Alternatives	536
28.1.9 Extensions	537
<b>28.2 Using the re Module</b>	539
28.2.1 Searching	540
28.2.2 Matching	540
28.2.3 Splitting a String	541
28.2.4 Replacing Parts of a String	541
28.2.5 Replacing Problem Characters	542
28.2.6 Compiling a Regular Expression	542
28.2.7 Flags	543
28.2.8 The Match Object	544
<b>28.3 A Simple Sample Program: Searching</b>	546
<b>28.4 A More Complex Sample Program: Matching</b>	547
<b>28.5 Comments in Regular Expressions</b>	550

## 29 Interface to Operating System and Runtime Environment

<b>29.1 Operating System Functionality: os</b>	553
29.1.1 environ	554
29.1.2 getpid()	554
29.1.3 cpu_count()	554
29.1.4 system(cmd)	554
29.1.5 popen(command, [mode, buffering])	555
<b>29.2 Accessing the Runtime Environment: sys</b>	555
29.2.1 Command Line Parameters	556
29.2.2 Default Paths	556
29.2.3 Standard Input/Output Streams	556

29.2.4	Exiting the Program .....	556
29.2.5	Details of the Python Version .....	557
29.2.6	Operating System Details .....	557
29.2.7	Hooks .....	559
<b>29.3</b>	<b>Command Line Parameters: argparse</b> .....	<b>561</b>
29.3.1	Calculator: A Simple Example .....	562
29.3.2	A More Complex Example .....	566
<b>30</b>	<b>File System</b>	<b>569</b>
<b>30.1</b>	<b>Accessing the File System: os</b> .....	<b>569</b>
30.1.1	access(path, mode) .....	570
30.1.2	chmod(path, mode) .....	571
30.1.3	listdir([path]) .....	571
30.1.4	mkdir(path, [mode]) and makedirs(path, [mode]) .....	572
30.1.5	remove(path) .....	572
30.1.6	removedirs(path) .....	572
30.1.7	rename(src, dst) and renames(old, new) .....	573
30.1.8	walk(top, [topdown, onerror]) .....	573
<b>30.2</b>	<b>File Paths: os.path</b> .....	<b>575</b>
30.2.1	abspath(path) .....	576
30.2.2	basename(path) .....	577
30.2.3	commonprefix(list) .....	577
30.2.4	dirname(path) .....	577
30.2.5	join(path, *paths) .....	578
30.2.6	normcase(path) .....	578
30.2.7	split(path) .....	578
30.2.8	splitdrive(path) .....	579
30.2.9	splitext(path) .....	579
<b>30.3</b>	<b>Accessing the File System: shutil</b> .....	<b>579</b>
30.3.1	Directory and File Operations .....	581
30.3.2	Archive Operations .....	582
<b>30.4</b>	<b>Temporary Files: tempfile</b> .....	<b>585</b>
30.4.1	TemporaryFile([mode, [bufsize, suffix, prefix, dir]]) .....	585
30.4.2	tempfile.TemporaryDirectory([suffix, prefix, dir]) .....	586

<b>31</b>	<b>Parallel Programming</b>	<b>587</b>
<b>31.1</b>	<b>Processes, Multitasking, and Threads</b> .....	<b>587</b>
31.1.1	The Lightweights among the Processes: Threads .....	588
31.1.2	Threads or Processes? .....	590
31.1.3	Cooperative Multitasking: A Third Way .....	590
<b>31.2</b>	<b>Python's Interfaces for Parallelization</b> .....	<b>591</b>
<b>31.3</b>	<b>The Abstract Interface: concurrent.futures</b> .....	<b>592</b>
31.3.1	An Example with a futures.ThreadPoolExecutor .....	593
31.3.2	Executor Instances as Context Managers .....	595
31.3.3	Using futures.ProcessPoolExecutor .....	595
31.3.4	Managing the Tasks of an Executor .....	596
<b>31.4</b>	<b>The Flexible Interface: threading and multiprocessing</b> .....	<b>602</b>
31.4.1	Threads in Python: threading .....	603
31.4.2	Processes in Python: multiprocessing .....	611
<b>31.5</b>	<b>Cooperative Multitasking</b> .....	<b>613</b>
31.5.1	Cooperative Functions: Coroutines .....	613
31.5.2	Awaitable Objects .....	614
31.5.3	The Cooperation of Coroutines: Tasks .....	615
31.5.4	A Cooperative Web Crawler .....	618
31.5.5	Blocking Operations in Coroutines .....	625
31.5.6	Other Asynchronous Language Features .....	627
<b>31.6</b>	<b>Conclusion: Which Interface Is the Right One?</b> .....	<b>629</b>
31.6.1	Is Cooperative Multitasking an Option? .....	629
31.6.2	Abstraction or Flexibility? .....	630
31.6.3	Threads or Processes? .....	630
<b>32</b>	<b>Data Storage</b>	<b>631</b>
<b>32.1</b>	<b>XML</b> .....	<b>631</b>
32.1.1	ElementTree .....	633
32.1.2	Simple API for XML .....	640
<b>32.2</b>	<b>Databases</b> .....	<b>643</b>
32.2.1	The Built-In Database in Python: sqlite3 .....	646
<b>32.3</b>	<b>Compressed Files and Archives</b> .....	<b>661</b>
32.3.1	gzip.open(filename, [mode, compresslevel]) .....	661
32.3.2	Other Modules for Accessing Compressed Data .....	662

<b>32.4 Serializing Instances: pickle .....</b>	662
32.4.1 Functional Interface .....	663
32.4.2 Object-Oriented Interface .....	665

<b>32.5 The JSON Data Exchange Format: json .....</b>	665
---	-----

<b>32.6 The CSV Table Format: csv .....</b>	667
32.6.1 Reading Data from a CSV File with reader Objects .....	668
32.6.2 Using Custom Dialects: Dialect Objects .....	670

<b>33 Network Communication .....</b>	673
---------------------------------------	-----

<b>33.1 Socket API .....</b>	674
33.1.1 Client-Server Systems .....	675
33.1.2 UDP .....	677
33.1.3 TCP .....	678
33.1.4 Blocking and Nonblocking Sockets .....	680
33.1.5 Creating a Socket .....	681
33.1.6 The Socket Class .....	682
33.1.7 Network Byte Order .....	685
33.1.8 Multiplexing Servers: selectors .....	686
33.1.9 Object-Oriented Server Development: socketserver .....	688

<b>33.2 XML-RPC .....</b>	690
33.2.1 The Server .....	691
33.2.2 The Client .....	694
33.2.3 Multicall .....	696
33.2.4 Limitations .....	697

<b>34 Accessing Resources on the Internet .....</b>	701
---	-----

<b>34.1 Protocols .....</b>	701
34.1.1 Hypertext Transfer Protocol .....	701
34.1.2 File Transfer Protocol .....	701

<b>34.2 Solutions .....</b>	702
34.2.1 Outdated Solutions for Python 2 .....	702
34.2.2 Solutions in the Standard Library .....	702
34.2.3 Third-Party Solutions .....	702

<b>34.3 The Easy Way: requests .....</b>	703
34.3.1 Simple Requests via GET and POST .....	703
34.3.2 Web APIs .....	704

<b>34.4 URLs: urllib .....</b>	705
--------------------------------	-----

34.4.1 Accessing Remote Resources: urllib.request .....	706
34.4.2 Reading and Processing URLs: urllib.parse .....	710

<b>34.5 FTP: ftplib .....</b>	713
-------------------------------	-----

34.5.1 Connecting to an FTP Server .....	714
34.5.2 Executing FTP commands .....	715
34.5.3 Working with Files and Directories .....	715
34.5.4 Transferring Files .....	716

<b>35 Email .....</b>	721
-----------------------	-----

<b>35.1 SMTP: smtplib .....</b>	721
35.1.1 SMTP([host, port, local_hostname, timeout, source_address]) .....	722
35.1.2 Establishing and Terminating a Connection .....	722
35.1.3 Sending an Email .....	723
35.1.4 Example .....	724

<b>35.2 POP3: poplib .....</b>	724
35.2.1 POP3(host, [port, timeout]) .....	725
35.2.2 Establishing and Terminating a Connection .....	725
35.2.3 Listing Existing Emails .....	726
35.2.4 Retrieving and Deleting Emails .....	727
35.2.5 Example .....	727

<b>35.3 IMAP4: imaplib .....</b>	728
35.3.1 IMAP4([host, port, timeout]) .....	729
35.3.2 Establishing and Terminating a Connection .....	730
35.3.3 Finding and Selecting a Mailbox .....	730
35.3.4 Operations with Mailboxes .....	731
35.3.5 Searching Emails .....	731
35.3.6 Retrieving Emails .....	732
35.3.7 Example .....	733

<b>35.4 Creating Complex Emails: email .....</b>	734
35.4.1 Creating a Simple Email .....	734
35.4.2 Creating an Email with Attachments .....	735
35.4.3 Reading an Email .....	737

---

<b>36 Debugging and Quality Assurance</b>	739
<b>36.1 The Debugger</b>	739
<b>36.2 Automated Testing</b>	741
36.2.1 Test Cases in Docstrings: doctest	742
36.2.2 Unit Tests: unittest	746
<b>36.3 Analyzing the Runtime Performance</b>	749
36.3.1 Runtime Measurement: timeit	749
36.3.2 Profiling: cProfile	752
36.3.3 Tracing: trace	756
<b>37 Documentation</b>	759
<b>37.1 Docstrings</b>	759
<b>37.2 Automatically Generated Documentation: pydoc</b>	761

## Part V Advanced Topics

---

<b>38 Distributing Python Projects</b>	765
<b>38.1 A History of Distributions in Python</b>	765
38.1.1 The Classic Approach: distutils	766
38.1.2 The New Standard: setuptools	766
38.1.3 The Package Index: PyPI	766
<b>38.2 Creating Distributions: setuptools</b>	767
38.2.1 Installation	767
38.2.2 Writing the Module	767
38.2.3 The Installation Script	768
38.2.4 Creating a Source Distribution	773
38.2.5 Creating a Binary Distribution	773
38.2.6 Installing Distributions	774
<b>38.3 Creating EXE files: cx_Freeze</b>	775
38.3.1 Installation	775
38.3.2 Usage	775

---

<b>38.4 Package Manager</b>	776
38.4.1 The Python Package Manager: pip	777
38.4.2 The conda Package Manager	778
<b>38.5 Localizing Programs: gettext</b>	781
38.5.1 Example of Using gettext	781
38.5.2 Creating the Language Compilation	783
<b>39 Virtual Environments</b>	785
<b>39.1 Using Virtual Environments: venv</b>	786
39.1.1 Activating a Virtual Environment	786
39.1.2 Working in a Virtual Environment	786
39.1.3 Deactivating a Virtual Environment	787
<b>39.2 Virtual Environments in Anaconda</b>	787
<b>40 Alternative Interpreters and Compilers</b>	789

---

<b>40.1 Just-in-Time Compilation: PyPy</b>	789
40.1.1 Installation and Use	789
40.1.2 Example	790
<b>40.2 Numba</b>	790
40.2.1 Installation	791
40.2.2 Example	791
<b>40.3 Connecting to C and C++: Cython</b>	793
40.3.1 Installation	793
40.3.2 The Functionality of Cython	794
40.3.3 Compiling a Cython Program	794
40.3.4 A Cython Program with Static Typing	796
40.3.5 Using a C Library	797
<b>40.4 The Interactive Python Shell: IPython</b>	799
40.4.1 Installation	799
40.4.2 The Interactive Shell	799
40.4.3 The Jupyter Notebook	802

---

<b>41 Graphical User Interfaces</b>	805
<b>  41.1 Toolkits</b>	805
41.1.1 Tkinter (Tk)	805
41.1.2 PyGObject (GTK)	806
41.1.3 Qt for Python (Qt)	806
41.1.4 wxPython (wxWidgets)	807
<b>  41.2 Introduction to tkinter</b>	807
41.2.1 A Simple Example	807
41.2.2 Control Variables	810
41.2.3 The Packer	811
41.2.4 Events	815
41.2.5 Widgets	821
41.2.6 Drawings: The Canvas Widget	839
41.2.7 Other Modules	846
<b>  41.3 Introduction to PySide6</b>	850
41.3.1 Installation	850
41.3.2 Basic Concepts of Qt	850
41.3.3 Development Process	852
<b>  41.4 Signals and Slots</b>	859
<b>  41.5 Important Widgets</b>	861
41.5.1 QCheckBox	862
41.5.2 QComboBox	862
41.5.3 QDateEdit, QDateTimeEdit, and QDateTimeEdit	863
41.5.4 QDialog	863
41.5.5 QLineEdit	864
41.5.6 QListWidget and QListView	864
41.5.7 QProgressBar	865
41.5.8 QPushButton	865
41.5.9 QRadioButton	865
41.5.10 QSlider and QDial	866
41.5.11 QTextEdit	866
41.5.12 QWidget	867
<b>  41.6 Drawing Functionality</b>	868
41.6.1 Tools	868
41.6.2 Coordinate System	870
41.6.3 Simple Shapes	871
41.6.4 Images	873
41.6.5 Text	874
41.6.6 Eye Candy	876

---

<b>41.7 Model-View Architecture</b>	879
41.7.1 Sample Project: An Address Book	880
41.7.2 Selecting Entries	888
41.7.3 Editing Entries	890
<b>42 Python as a Server-Side Programming Language on the Web: An Introduction to Django</b>	893
<b>  42.1 Concepts and Features of Django</b>	894
<b>  42.2 Installing Django</b>	895
<b>  42.3 Creating a New Django Project</b>	896
42.3.1 The Development Web Server	897
42.3.2 Configuring the Project	898
<b>  42.4 Creating an Application</b>	900
42.4.1 Importing the Application into the Project	901
42.4.2 Defining a Model	901
42.4.3 Relationships between Models	902
42.4.4 Transferring the Model to the Database	903
42.4.5 The Model API	904
42.4.6 The Project Gets a Face	909
42.4.7 Django's Template System	915
42.4.8 Processing Form Data	926
42.4.9 Django's Admin Control Panel	930
<b>43 Scientific Computing and Data Science</b>	935
<b>  43.1 Installation</b>	936
<b>  43.2 The Model Program</b>	936
43.2.1 Importing numpy, scipy, and matplotlib	937
43.2.2 Vectorization and the numpy.ndarray Data Type	938
43.2.3 Visualizing Data Using matplotlib.pyplot	942
<b>  43.3 Overview of the numpy and scipy Modules</b>	944
43.3.1 Overview of the numpy.ndarray Data Type	944
43.3.2 Overview of scipy	952
<b>  43.4 An Introduction to Data Analysis with pandas</b>	953
43.4.1 The DataFrame Object	954

---

43.4.2 Selective Data Access .....	955
43.4.3 Deleting Rows and Columns .....	961
43.4.4 Inserting Rows and Columns .....	961
43.4.5 Logical Expressions on Data Records .....	962
43.4.6 Manipulating Data Records .....	963
43.4.7 Input and Output .....	965
43.4.8 Visualization .....	966

---

## 44 Inside Knowledge 969

44.1 Opening URLs in the Default Browser: <code>webbrowser</code> .....	969
44.2 Interpreting Binary Data: <code>struct</code> .....	969
44.3 Hidden Password Entry .....	971
44.3.1 <code>getpass([prompt, stream])</code> .....	971
44.3.2 <code>getpass.getuser()</code> .....	972
44.4 Command Line Interpreter .....	972
44.5 File Interface for Strings: <code>io.StringIO</code> .....	975
44.6 Generators as Consumers .....	976
44.6.1 A Decorator for Consuming Generator Functions .....	978
44.6.2 Triggering Exceptions in a Generator .....	978
44.6.3 A Pipeline as a Chain of Consuming Generator Functions .....	979
44.7 Copying Instances: <code>copy</code> .....	981
44.8 Image Processing: <code>Pillow</code> .....	984
44.8.1 Installation .....	984
44.8.2 Loading and Saving Image Files .....	984
44.8.3 Accessing Individual Pixels .....	985
44.8.4 Manipulating Images .....	986
44.8.5 Interoperability .....	992

---

## 45 From Python 2 to Python 3 993

45.1 The Main Differences .....	996
45.1.1 Input/Output .....	996
45.1.2 Iterators .....	997
45.1.3 Strings .....	998
45.1.4 Integers .....	999

---

45.1.5 Exception Handling .....	999
45.1.6 Standard Library .....	1000
<b>45.2 Automatic Conversion .....</b>	<b>1001</b>

---

## Appendices 1005

<b>A Appendix .....</b>	<b>1005</b>
<b>B The Authors .....</b>	<b>1017</b>

Index .....	1019
-------------	------

# Index

-	.....	234, 386, 387
^	.....	139, 234, 386, 387
_	.....	54, 485
abs	__	388
add	__	386, 391
aenter	__	628
aexit	__	628
aiter	__	628
and	__	386
anext	__	628
annotations	__	469
bool	__	376
builtins	__	330
bytes	__	376
call	__	376, 378, 453
complex	__	376, 389
contains	__	391
debug	__	412
del	__	376, 377
delattr	__	379
delitem	__	390
dict	__	379
divmod	__	386
doc	__	760
enter	__	390, 443, 627
eq	__	385
exit	__	390, 443, 627
file	__	330
float	__	376, 389
floordiv	__	386
future	__	338
ge	__	385
getattr	__	379
getattribute	__	379, 380
getitem	__	390
gt	__	385
hash	__	376, 378
iadd	__	388, 391
inand	__	388
ifloordiv	__	388
ilshift	__	388
imatmul	__	388
imod	__	388
imul	__	388, 391
index	__	376
init	__	376
init	.py	331, 333, 767
int	__	389
invert	__	388
ior	__	388
ipow	__	388
irshift	__	388
isub	__	388
iter	__	391, 422
itruediv	__	388
ixor	__	388
le	__	385
len	__	390
lshift	__	386
lt	__	385
main	__	330
match_args	__	492
matmul	__	386
mod	__	386
mul	__	386, 391
name	__	330
ne	__	385
neg	__	388
next	__	422
or	__	386
pos	__	388
pow	__	386
radd	__	387, 391
rand	__	387
rdivmod	__	387
repr	__	376
rfloordiv	__	387
rlshift	__	387
rmatmul	__	387
rmod	__	387
rmul	__	387, 391
ror	__	387
round	__	376, 389
rpow	__	387
rrshift	__	387
rshift	__	386
rsub	__	387
rtruediv	__	387
rxor	__	387
setattr	__	379, 380
setitem	__	390
slots	__	379, 381
sub	__	386
truediv	__	386
xor	__	386
:=		87, 117, 491

... .....	487	Archive (Cont.)
) .....	487	XZ ..... 584
[...] .....	117	ZIP ..... 579, 584
[] .....	117, 486	Arcsine ..... 501
{...} .....	117	Arctangent ..... 501
{} .....	117, 488	argparse ..... 561
@ .....	384, 386, 387, 938	Argument ..... 116, 278
* .....	288, 386, 387	<i>keyword</i> ..... 116, 283
** .....	132, 289, 386, 387	<i>keyword-only</i> ..... 117
/ .....	386, 387	<i>optional</i> ..... 117
// .....	50, 132, 386, 387	<i>positional</i> ..... 116, 283
\ .....	68, 69	Argument (command) ..... 561
\u .....	211	Arithmetic expression ..... 127
\x .....	209	Arithmetic mean ..... 508
& .....	137, 233, 386, 387	Arithmetic operator ..... 131
% .....	132, 198, 386, 387	as ..... 327, 404, 484
+ .....	386, 387	ASCII ..... 195, 208, 212
< .....	232	ascii (function) ..... 305
<< .....	386, 387	ASGI ..... 897
> .....	232	assert ..... 411
>> .....	386, 387	Assignment ..... 54
.....	138, 221, 232, 386, 387	<i>augmented</i> ..... 132, 387
~ .....	139	Assignment expression ..... 87
\$ .....	43, 944	async ..... 592, 614, 619, 627
2to3 .....	1001	async def ..... 614
<b>A</b>		
ABC .....	473	async for ..... 628, 629
ABC (programming language) .....	39	async with ..... 619, 627
abs .....	304	Asynchronous comprehension ..... 629
Abstract base class .....	473	Asynchronous generator ..... 628
Access, random .....	633	Asynchronous iterator ..... 628
Admin control panel .....	930	asyncio ..... 592, 629
aiofiles .....	618	Attribute ..... 118, 345, 349, 632
aiohttp .....	618	<i>class attribute</i> ..... 369
all .....	304	<i>magic attribute</i> ..... 375
Alpha blending .....	877	<i>property attribute</i> ..... 366
Anaconda .....	42	Augmented assignment ..... 132, 387
Anaconda Navigator .....	787	Automated testing ..... 741
Anaconda PowerShell .....	43	Average ..... 508
Anaconda Prompt .....	43	await ..... 592, 614
and .....	56	Awaitable object ..... 614
Annotation .....	463, 464	
Anonymous function .....	299	
Antialiasing (Qt) .....	878	
any .....	305	
API .....	704	
Arccosine .....	501	
Archive .....	579	
<i>BZ2</i> ..... 584		
TAR ..... 579, 584		
<b>B</b>		
Backslash .....	68	
Base class .....	351	
BaseException .....	400	
Basic data type		
<i>bool</i> ..... 144		
<i>bytearray</i> ..... 182		
<i>bytes</i> ..... 182		
<i>complex</i> ..... 149		
<i>dict</i> ..... 53, 215, 241, 242		

Basic data type (Cont.)		
<i>float</i> ..... 51, 141		Built-in function (Cont.)
<i>frozenset</i> ..... 215, 227, 237		<i>enumerate</i> ..... 309
<i>int</i> ..... 49, 135		<i>eval</i> ..... 309
<i>list</i> ..... 52, 166		<i>exec</i> ..... 310
<i>NoneType</i> ..... 126		<i>filter</i> ..... 178, 310
<i>set</i> ..... 215, 227, 236		<i>float</i> ..... 311
<i>str</i> ..... 51, 182		<i>format</i> ..... 311
<i>tuple</i> ..... 179		<i>frozenset</i> ..... 311
Batteries included .....	35	<i>getattr</i> ..... 370
Bezier curve (Qt) .....	878	<i>globals</i> ..... 312
Big endian .....	559	<i>hasattr</i> ..... 370
bin .....	305	<i>hash</i> ..... 312
Binary distribution .....	765, 773	<i>help</i> ..... 119, 313
Binary operator .....	386	<i>hex</i> ..... 313
Binary system .....	136	<i>id</i> ..... 108, 313
Bit operator .....	137	<i>input</i> ..... 314
<i>bit shift</i> ..... 140		<i>int</i> ..... 314
<i>bitwise AND</i> ..... 137		<i>isinstance</i> ..... 371
<i>bitwise complement</i> ..... 139		<i>issubclass</i> ..... 371
<i>bitwise exclusive OR</i> ..... 139		<i>iter</i> ..... 422
<i>bitwise OR</i> ..... 138		<i>len</i> ..... 164, 219, 232, 315
Bitmap .....	101	<i>list</i> ..... 315
Block comment .....	72	<i>locals</i> ..... 315
Bodiless tag (XML) .....	632	<i>map</i> ..... 178, 316
bool .....	144, 146, 306	<i>max</i> ..... 115, 164, 317
Boolean expression .....	55, 144	<i>min</i> ..... 164, 318
Boolean operator .....	57	<i>oct</i> ..... 318
Boolean value .....	144	<i>open</i> ..... 92, 97, 625
break .....	80	<i>ord</i> ..... 212, 318
Breakpoint .....	740	<i>pow</i> ..... 318
breakpoint (function) .....	301	<i>print</i> ..... 59, 318
Brush (Qt) .....	870	<i>property</i> ..... 366
Bubble sort .....	794	<i>range</i> ..... 85, 277, 319
Bug .....	739	<i>repr</i> ..... 320
Built-in exceptions .....	400	<i>reversed</i> ..... 320
Built-in function .....	58, 115, 300, 1007	<i>round</i> ..... 321
<i>abs</i> ..... 304		<i>set</i> ..... 321
<i>all</i> ..... 304		<i>setattr</i> ..... 370
<i>any</i> ..... 305		<i>sorted</i> ..... 321
<i>ascii</i> ..... 305		<i>staticmethod</i> ..... 368, 450
<i>bin</i> ..... 305		<i>str</i> ..... 322
<i>bool</i> ..... 146, 306		<i>sum</i> ..... 323
<i>breakpoint</i> ..... 301		<i>tuple</i> ..... 323
<i>bytearray</i> ..... 306		<i>type</i> ..... 106, 125, 323
<i>bytes</i> ..... 307		<i>zip</i> ..... 324
<i>chr</i> ..... 212, 307		Built-in module ..... 325
<i>classmethod</i> ..... 369		Busy waiting ..... 686
<i>complex</i> ..... 307		Button (tkinter) ..... 823
<i>delattr</i> ..... 370		Byte code ..... 40, 65
<i>dict</i> ..... 308		Byte order ..... 685
<i>divmod</i> ..... 309		bytarray ..... 182, 306

bytes ..... 182, 307  
BZ2 ..... 584

---

**C**

C/C++ ..... 790, 793  
Cache ..... 452  
Cache (for function) ..... 457  
Call by Reference ..... 290  
Call by Sharing ..... 291  
Call by Value ..... 290  
Callstack ..... 408  
Canvas (tkinter) ..... 839  
Capture pattern ..... 484  
Cartesian coordinates ..... 502, 503  
Cartesian product ..... 438  
case ..... 480  
Case-sensitive ..... 55  
cdef ..... 796, 798  
ChainMap (dictionaries) ..... 239  
Character class ..... 530, 533  
Character literal ..... 529  
Character set ..... 208  
Character string ..... 51  
Checkbox (Qt) ..... 862  
Checkbutton (tkinter) ..... 824  
Children (DOM) ..... 634  
chr ..... 212, 307  
cimport ..... 797  
Class ..... 346, 347  
  attribute ..... 349  
  base class ..... 351  
  constructor ..... 348  
  data class ..... 393  
  instance ..... 346  
  method ..... 347  
Class attribute ..... 369  
Class decorator ..... 454  
Class method ..... 369  
classmethod (function) ..... 369  
Client ..... 675, 677  
Client-server system ..... 675  
cmath ..... 497  
cmd ..... 972  
Code point ..... 210  
Codepage ..... 208  
coding (file header) ..... 214  
collections ..... 239  
  ChainMap ..... 239  
  counter ..... 241, 242  
  defaultdict ..... 242  
  deque ..... 244

collections (Cont.)  
  *namedtuple* ..... 245  
Column index ..... 955, 956  
Combination ..... 434  
Combobox (Qt) ..... 862  
Command line interpreter ..... 972  
Command line parameters ..... 556, 561  
Command prompt ..... 43, 63, 561  
Comment ..... 72  
Communication socket ..... 676  
Comparison ..... 55  
Comparison operator ..... 133, 384  
Compiler ..... 40, 65, 793  
  *just-in-time* ..... 66  
Complement ..... 139  
complex ..... 149, 307  
Complex number ..... 149, 502  
  *conjugated* ..... 150  
  *imaginary part* ..... 149  
  *real part* ..... 149  
Complex plane ..... 502  
Comprehension  
  *asynchronous* ..... 629  
  *dict* ..... 216, 227  
  *generator expression* ..... 421  
  *list* ..... 177  
Concatenation (of sequences) ..... 157  
concurrent.futures ..... 591, 630  
Conditional ..... 75, 479  
Conditional expression ..... 78  
Connection object ..... 684  
Connection socket ..... 675  
Console ..... 561  
Console application ..... 63  
Constructor ..... 348  
  *factory function* ..... 368  
Consumer (generator) ..... 976  
Consumer (queue) ..... 620  
Container ..... 239, 390  
Context manager ..... 389, 442  
  *asynchronous* ..... 619, 627  
Context object ..... 441  
contextlib ..... 444  
continue ..... 82  
Control (GUI) ..... 805  
Control character ..... 185  
Control structure ..... 75  
  *conditional* ..... 75  
  *conditional expression* ..... 78  
  loop ..... 79  
  *structural pattern matching* ..... 479  
Control variable (tkinter) ..... 810

Convex polygon ..... 844  
Cooperative multitasking ..... 591, 613, 629  
Coordinate system (Qt) ..... 870  
Coordinate system (tkinter) ..... 839  
Coordinated Universal Time (UTC) .... 248, 264  
copy ..... 981  
Coroutine ..... 613  
Cosine ..... 501  
Counter ..... 241, 242  
Counting loop ..... 85  
Coverage analysis ..... 756  
cProfile ..... 752  
CPU ..... 554  
CPython ..... 66, 590, 601, 789, 798  
CriticalSection ..... 605, 690  
CSV ..... 667, 966  
  *dialect* ..... 668  
CUDA ..... 790  
cx\_Freeze ..... 775  
Cython ..... 793

---

**D**

Data class ..... 393, 455  
Data science ..... 953  
Data stream ..... 91  
Data type ..... 106, 125  
  *container* ..... 239  
  *conversion* ..... 134  
  *immutable* ..... 111, 181  
  *mapping* ..... 215  
  *mutable* ..... 111  
  *numeric* ..... 131  
  *sequential* ..... 153  
  *set* ..... 215  
Database ..... 643  
  *cursor* ..... 647  
  *join* ..... 653  
  *query* ..... 644  
  *transaction* ..... 649  
Date ..... 247  
Date edit (Qt) ..... 863  
datetime ..... 254  
Daylight Saving Time (DST) ..... 265  
Deadlock ..... 610  
Debugging ..... 739  
  *breakpoint* ..... 740  
  *post mortem* ..... 741  
decimal (module) ..... 509  
Decimal system ..... 135  
decode ..... 209

Decorator ..... 449  
  *nested* ..... 452  
  *of a class* ..... 454  
  *of a function* ..... 449, 451  
  *of a method* ..... 451  
def ..... 279, 614  
Default path ..... 556  
defaultdict ..... 242  
del ..... 110, 168, 220  
delattr ..... 370  
Delegate (Qt) ..... 883  
DeprecationWarning ..... 412  
deque (data type) ..... 244  
Deserialize ..... 662  
Development environment (IDE) ..... 45, 1014  
Development web server (Django) ..... 897  
Dialog (Qt) ..... 858, 863  
  *modal* ..... 863  
  *nonmodal* ..... 863  
dict ..... 53, 215, 308  
Dict comprehension ..... 216, 227  
Dictionary ..... 53, 215, 241, 242  
  *chained* ..... 239  
Difference set ..... 234  
  *symmetric* ..... 234  
Distribution ..... 765  
distutils ..... 766, 767  
divmod ..... 309, 386, 387  
Django ..... 894  
  *application* ..... 894, 900  
  *field lookup* ..... 908  
  *migration* ..... 903  
  *path* ..... 911  
  *project* ..... 894  
  *view* ..... 910  
Docstring ..... 742, 759  
doctest ..... 742  
Document Object Model (XML) → DOM (XML)  
Documentation ..... 119, 759  
Dollar sign ..... 43, 944  
DOM (XML) ..... 633  
  *child* ..... 634  
  *children* ..... 634  
  *node* ..... 633  
  *Root* ..... 634  
  *siblings* ..... 634  
double ..... 142  
Drawing (Qt) ..... 868  
Drawing (tkinter) ..... 839  
DRY principle (Django) ..... 895  
Dual system ..... 136, 137  
Duck typing ..... 388, 463, 794

<b>E</b>	
Egg	766
ElementTree (XML)	633
elif	76
Ellipsis	487
else	77, 79, 81, 404
Email	721
<i>header</i>	734
email (module)	734
Emoji	211
encode	209
Encoding	98
Encoding declaration	214
End of file (EOF)	91
Entry widget (tkinter)	826
Enum	481
enum	269, 270
enumerate (function)	309
Enumeration	269
<i>alias</i>	271
<i>flag</i>	271
<i>integer</i>	272
Escape sequence	185, 209, 534
<i>\N</i>	211
<i>\u</i>	211
<i>\x</i>	209
eval	309
Event (Qt)	859
Event (tkinter)	815
Event handler (Qt)	859
Event handler (tkinter)	815
Excel	966
except	402
Exception	399, 999, 1011
<i>BaseException</i>	400
<i>built-in</i>	400
<i>chaining</i>	410
<i>handling</i>	401
<i>raise</i>	401
<i>re-raise</i>	408
exec	310
Exit code	556
Exponent	142, 499
Exponential function	501
Exponential notation	142
Expression	
<i>arithmetic</i>	127
<i>Boolean</i>	55, 144
<i>logical</i>	55, 144
<i>self-documenting</i>	206
Extension	767, 794

<b>F</b>	
Factorial	83
Factory function	368
False	55, 144
Fibonacci sequence	423
Field lookup (Django)	908
File	92
<i>temporary</i>	585
File access rights	571
File descriptor	99
File dialog (tkinter)	847
File object	92, 99, 707
File path	575
File system	569
File Transfer Protocol	
<i>see FTP</i>	713
File-like object	555
filter	178
Filter (Django)	918
filter (function)	310
finally	404
Finder (importlib)	336
Fire and Forget	617
First in, first out (FIFO)	620
Flag (enum)	272
Flag (RegExp)	543
float	51, 141, 311
Font (tkinter)	848
for	84, 177, 628, 629
format	311
Frequency distribution	240
from	327, 333
from/cimport	797
frozenset	215, 227, 237, 311
f-string	198, 206
FTP	701, 713
<i>control channel</i>	714
<i>data channel</i>	714
<i>mode</i>	714
ftplib	702, 713
Function	57, 115, 277
<i>anonymous</i>	299
<i>argument</i>	278
<i>body</i>	279
<i>built-in</i>	300
<i>call</i>	57, 278
<i>definition</i>	279
<i>hyperbolic</i>	501
<i>interface</i>	279
<i>keyword argument</i>	283
<i>keyword-only parameter</i>	286

<b>Function (Cont.)</b>	
<i>local</i>	295
<i>name</i>	279
<i>namespace</i>	293
<i>optional parameter</i>	282
<i>overloading</i>	459
<i>parameter</i>	278
<i>positional-only parameter</i>	287
<i>recursive</i>	300
<i>return value</i>	278, 279
<i>trigonometric</i>	501
Function annotations	463, 465
Function call	57, 278
Function decorator	449, 451
Function iterator	431
Function name	279
Function object	282
Function parameters	58
functools	455
Future import	339

<b>H</b>	
Harmonic mean	508
hasattr	370
Hash	179
hash (function)	312
Hash collision	515
Hash function	514
Hash randomization	230
Hash value	217, 312, 514
hashable	379
hashlib	514
HDF5	966
Help	119
<i>interactive</i>	119
help (function)	119, 313
hex	313
Hexadecimal system	136
History function	.49
Hook (function)	559
HTML	546, 966
HTTP	701
HTTPS	701
Hyperbolic cosine	501
Hyperbolic function	501
Hyperbolic sine	501
Hyperbolic tangent	501
Hypotenuse	502

<b>G</b>	
Garbage collection	110
Gaussian distribution	505
Generator	416, 976
<i>asynchronous</i>	628
<i>consuming</i>	976
<i>subgenerator</i>	418
Generator expression	421
Generics	472
Geometric mean	508
GET (HTTP)	701, 703, 927
getattr	370
getpass	971
Getter method	365
gettext	781
<i>language compilation</i>	783
GIL	590, 798
Git	1015
Global	295
Global Interpreter Lock -> see GIL	590
Global module	325
Global namespace	293
Global reference	293
Global variable	589
globals	312
GNU gettext API	781
Golden ratio	423
Gradient (Qt)	876
Graphical user interface -> see GUI	805
GTK	806
IANA time zone database	263
id	108, 313
IDE	1014
<i>LiClipse</i>	1015
<i>PyCharm</i>	1014
<i>PyDev</i>	1015
<i>Spyder</i>	1016
<i>Visual Studio Code</i>	1015
Identifiers	55
Identity (of an instance)	108

Identity comparison (of instances) ..... 109  
 IDLE ..... 45, 739  
 IEEE-754 ..... 142  
 if ..... 76, 79, 178, 483  
 Image processing ..... 984  
 Images (Qt) ..... 873  
 Imaginary part ..... 149  
 IMAP4 ..... 728  
     *mailbox* ..... 729  
 imaplib ..... 728  
 Immutable ..... 111, 125, 181  
 Immutable data type ..... 111  
 Import  
     *absolute* ..... 334  
     *relative* ..... 334  
 import ..... 333  
 import statement ..... 60, 326, 338  
 Importer ..... 335  
 importlib ..... 335  
     *finder* ..... 336  
     *loader* ..... 337  
 in ..... 155, 177, 221, 232  
 in place ..... 158  
 Indentation ..... 67  
 Index (in a sequence) ..... 159  
 IndexError ..... 160  
 inf ..... 143, 499, 512  
 Infinite ..... 143  
 Inheritance ..... 351  
     *multiple inheritance* ..... 363  
 input ..... 314  
 Installation script ..... 768  
 Instance ..... 58, 103, 346  
     *data type* ..... 106  
     *identity* ..... 108  
     *value* ..... 107  
 Instantiation ..... 58, 346  
 int ..... 49, 135, 314  
 Integer ..... 49, 135, 999  
 Integer division ..... 50  
 Integrated Development Environment → IDE  
 IntEnum ..... 272  
 Interactive help ..... 119  
 Interactive mode ..... 45, 49  
     *history function* ..... 49  
 Interface ..... 116, 279, 455  
 Internationalization ..... 781  
 Interpreter ..... 40, 66  
     *CPython* ..... 66, 590, 601, 789, 798  
     *PyPy* ..... 789  
 Intersection (of sets) ..... 233  
 Inverse hyperbolic cosine ..... 501  
 Inverse hyperbolic sine ..... 501  
 Inverse hyperbolic tangent ..... 502  
 io.StringIO ..... 975  
 IP address ..... 674  
 IPv6 ..... 681  
 IPython ..... 799  
     *Notebook* ..... 802  
 is ..... 109, 127  
 isinstance ..... 371  
 issubclass ..... 371  
 iter ..... 422  
 Iterable object ..... 84, 422  
     *Cartesian product* ..... 438  
     *chain* ..... 433  
     *combination* ..... 434  
     *group* ..... 437  
     *partial sum* ..... 433  
     *permutation* ..... 438  
     *repeat* ..... 439  
 Iterator ..... 422, 997  
     *asynchronous* ..... 628  
 Iterator protocol ..... 84, 422  
 itertools ..... 432

**J**

JIT → Just-in-time compiler ..... 790  
 Join (SQL) ..... 653  
 JSON ..... 665, 704, 966  
 Jupyter Notebook ..... 802  
 JupyterLab ..... 802  
 Just-in-time compiler ..... 66, 790  
     *Numba* ..... 790  
     *PyPy* ..... 789

**K**

Keras ..... 42  
 Key-value pair ..... 215  
 Keyword ..... 55, 1005  
     *and* ..... 56  
     *as* ..... 327, 484  
     *assert* ..... 411  
     *break* ..... 80  
     *case* ..... 480  
     *class* ..... 347  
     *continue* ..... 82  
     *def* ..... 279  
     *del* ..... 110, 168, 220  
     *elif* ..... 76  
     *else* ..... 77, 79, 81, 404  
     *except* ..... 402

Keyword (Cont.)  
     *False* ..... 55, 144  
     *finally* ..... 404  
     *for* ..... 84, 177  
     *from* ..... 327, 333  
     *global* ..... 295  
     *if* ..... 76, 79, 178, 483  
     *import* ..... 326, 333, 338  
     *in* ..... 155, 177, 221, 232  
     *is* ..... 109, 127  
     *lambda* ..... 299  
     *match* ..... 480  
     *None* ..... 126  
     *nonlocal* ..... 296  
     *not* ..... 56, 144, 221  
     *not in* ..... 156, 232  
     *or* ..... 57  
     *pass* ..... 87  
     *raise* ..... 401  
     *return* ..... 280  
     *True* ..... 55, 144  
     *try* ..... 402  
     *while* ..... 79  
     *with* ..... 441  
     *yield* ..... 416, 976  
 Keyword argument ..... 116, 283  
 Keyword-only parameter ..... 117

**L**

Label (tkinter) ..... 827  
 LabelFrame (tkinter) ..... 828  
 lambda ..... 299  
 Language compilation ..... 783  
 Language element, planned ..... 338  
 LaTeX ..... 944  
 Layout (Qt) ..... 851  
 Layout (tkinter) ..... 811  
 Lazy evaluation ..... 79, 148  
 Leap second ..... 249  
 len ..... 164, 219, 232, 315  
 Library ..... 325  
 LiClipse (IDE) ..... 1015  
 Lightweight process ..... 589  
 Line comment ..... 72  
 Line edit (Qt) ..... 864  
 List ..... 52  
     *doubly linked* ..... 243  
     *side effect* ..... 175  
 list (data type) ..... 52, 166, 315  
 List comprehension ..... 177  
 Listbox (tkinter) ..... 829  
 Listen mode ..... 685  
 QListWidget (Qt) ..... 864  
 Literal ..... 49  
 Literal pattern ..... 481  
 Little endian ..... 559  
 Loader (importlib) ..... 337  
 loc ..... 957  
 Local function ..... 295  
 Local module ..... 325, 328  
 Local namespace ..... 293  
 Local reference ..... 293  
 Local time ..... 248  
 Localization ..... 781  
 locals ..... 315  
 Lock object ..... 605  
 Log file ..... 523  
 Logarithm function ..... 501  
 Logging ..... 523  
 Logging handler ..... 527  
 Logical expression ..... 144  
 Logical operator ..... 144  
     *logical AND* ..... 145  
     *logical negation* ..... 144  
     *logical OR* ..... 145  
 Logical expression ..... 55  
 long ..... 135  
 Loop ..... 79  
     *asynchronous* ..... 628  
     *body* ..... 79  
     *break* ..... 80  
     *continue* ..... 82  
     *counting loop* ..... 85  
     *else* ..... 81  
     *for* ..... 84  
     *while* ..... 79  
 Loose coupling (Django) ..... 894  
 Lower median ..... 508

**M**

Magic attribute ..... 375  
     *\_annotations\_* ..... 469  
     *\_dict\_* ..... 379  
     *\_doc\_* ..... 760  
     *\_match\_args\_* ..... 492  
     *\_slots\_* ..... 379, 381  
 Magic line (program header) ..... 65  
 Magic method ..... 375  
     *\_abs\_* ..... 388  
     *\_add\_* ..... 386, 391  
     *\_and\_* ..... 386  
     *\_bytes\_* ..... 376

Magic method (Cont.)	
<code>_call_</code>	376, 378, 453
<code>_complex_</code>	376, 389
<code>_contains_</code>	391
<code>_del_</code>	376, 377
<code>_delattr_</code>	379
<code>_delitem_</code>	390
<code>_div_</code>	386
<code>_divmod_</code>	386
<code>_enter_</code>	390, 443
<code>_eq_</code>	385
<code>_exit_</code>	390, 443
<code>_float_</code>	376, 389
<code>_floordiv_</code>	386
<code>_ge_</code>	385
<code>_getattr_</code>	379
<code>_getattribute_</code>	379, 380
<code>_getitem_</code>	390, 430
<code>_gt_</code>	385
<code>_hash_</code>	376, 378
<code>_iadd_</code>	388, 391
<code>_iand_</code>	388
<code>_idiv_</code>	388
<code>_ifloordiv_</code>	388
<code>_ilshift_</code>	388
<code>_imatmul_</code>	388
<code>_imod_</code>	388
<code>_imul_</code>	388, 391
<code>_index_</code>	376, 389
<code>_init_</code>	376
<code>_int_</code>	389
<code>_invert_</code>	388
<code>_ior_</code>	388
<code>_ipow_</code>	388
<code>_irshift_</code>	388
<code>_isub_</code>	388
<code>_iter_</code>	391, 422
<code>_ixor_</code>	388
<code>_le_</code>	385
<code>_len_</code>	390
<code>_lshift_</code>	386
<code>_lt_</code>	385
<code>_matmul_</code>	386
<code>_mod_</code>	386
<code>_mul_</code>	386, 391
<code>_ne_</code>	385
<code>_neg_</code>	388
<code>_next_</code>	422
<code>_nonzero_</code>	376
<code>_or_</code>	386
<code>_pos_</code>	388
<code>_pow_</code>	386

Magic method (Cont.)	
<code>_radd_</code>	387, 391
<code>_rand_</code>	387
<code>_rdiv_</code>	387
<code>_rdivmod_</code>	387
<code>_repr_</code>	376
<code>_rfloordiv_</code>	387
<code>_rlshift_</code>	387
<code>_rmatmul_</code>	387
<code>_rmod_</code>	387
<code>_rmul_</code>	387, 391
<code>_ror_</code>	387
<code>_round_</code>	376, 389
<code>_rpow_</code>	387
<code>_rrshift_</code>	387
<code>_rshift_</code>	386
<code>_rsub_</code>	387
<code>_rxor_</code>	387
<code>_setattr_</code>	379, 380
<code>_setitem_</code>	390
<code>_str_</code>	376
<code>_sub_</code>	386
<code>_xor_</code>	386
Mailbox	729
Main Event Loop (Qt)	858
Mantissa	142, 499
map (function)	178, 316
Mapping	215
Mapping patterns	488
match	480
Match object (RegExp)	544
Matching (RegExp)	529, 544, 547
math	497
MATLAB	935
matplotlib	42, 935, 942
max	115, 164, 317
MD5	516
Median	508
Member	345
Memory view	796
Menu (tkinter)	831
Menu bar (tkinter)	831
Menu button (tkinter)	833
Message box (tkinter)	848
Metaclass	346, 373
Method	58, 115, 345, 347
<code>class method</code>	369
<code>definition</code>	347
<code>getter method</code>	365
<code>magic method</code>	375
<code>overriding</code>	353
<code>setter method</code>	365

Method (Cont.)	
<code>static</code>	368
Microsoft Excel	966
Migration (Django)	903
MIME	734
min	164, 318
Modal dialog (Qt)	863
Modal value (statistics)	508
Mode	
<code>interactive</code>	45, 49
Mode (statistics)	508
Model (Django)	894, 901
Model API (Django)	904
Model class (Qt)	879
Model-view concept (Django)	894, 900, 901
Model-view concept (Qt)	851, 879
Modifier (tkinter)	816
Module	60, 325, 767
<code>built-in</code>	325
<code>executing</code>	330
<code>global</code>	325
<code>local</code>	325, 328
<code>name conflict</code>	329
ModuleNotFoundError	330
Modulo	499
Monty Python	39
Multicall	696
Multiple inheritance	363
Multiplexing server	675, 686
Multiprocessing	592, 611, 630
Multitasking	587
<code>cooperative</code>	591, 613, 629
<code>preemptive</code>	590
Mutable	111, 125
Mutable data type	111
mypy	476
N	
Name conflict	329
Named expression	89
namedtuple	245
Namespace	293, 326
<code>global</code>	293
<code>local</code>	293
Namespace package	333
NaN	143, 965
nan	143, 499, 512
ndarray (NumPy)	938, 944
Network byte order	685
Node (DOM)	633
nogil	798
Non-convex polygon	844
None	126
NoneType	126
nonlocal	296
Non-modal dialog (Qt)	863
Normal distribution	505
not	56, 144
Not a number (NaN)	143
not in	156, 221, 232
Notebook	802
NotImplemented	389
Numba	790
Number	
<code>complex</code>	149
<code>float</code>	51, 141
<code>integer</code>	49, 135, 999
Numerical system	135
<code>decimal system</code>	135
<code>dual system</code>	136, 137
<code>hexadecimal system</code>	136
<code>octal system</code>	136
Numeric data type	131
NumPy	42, 142, 791, 935, 953
<code>ndarray</code>	938, 944
O	
Object	341, 345
<code>awaitable</code>	614
<code>file-like</code>	555
<code>iterable</code>	422
oct	318
Octal system	136
Ones' complement	139
One-to-many relation	902
One-way coding	515
open	92, 97, 625
openpyxl	966
Operand	127
Operating system	557
Operator	50, 127
<code>arithmetic</code>	131
<code>binary</code>	386
<code>bit operator</code>	137
<code>Boolean</code>	57
<code>comparison operator</code>	133, 384
<code>logical</code>	144
<code>logical AND</code>	145
<code>logical negation</code>	144
<code>logical OR</code>	145
<code>overloading</code>	382
<code>relational operator</code>	55

Operator (Cont.)	
<i>unary</i>	388
Operator precedence	128, 1005
Option (command)	561
Optional parameter	117, 282
OptionMenu (tkinter)	834
or	57
ord	212, 318
Ordering	459
os	553, 569
os.path	575
OSI Model	673
<b>P</b>	
Package	331, 767
<i>_init_.py</i>	331, 333
<i>namespace package</i>	333
Package manager	776
Packer (tkinter)	808, 811
Packing (sequence)	179
Padding (tkinter)	814
Painter (Qt)	869
Painter path (Qt)	878
pandas	42, 935, 953
Parallel server	675
Parameter	58, 116, 278
<i>any number</i>	284
<i>keyword</i>	116, 283
<i>keyword-only</i>	117, 286
<i>optional</i>	117, 282
<i>positional</i>	116, 283
<i>positional-only</i>	287
<i>unpack</i>	288
Parent (DOM)	634
Parser (XML)	632
Partial sum	433
pass	87
Password	518, 971
Path	556, 569, 575
Pattern	
<i>capture pattern</i>	484
<i>literal pattern</i>	481
<i>mapping pattern</i>	488
<i>sequence pattern</i>	486
<i>type checking pattern</i>	482
Payload (HTTP)	704
PBKDF2	519
pd	954
Pen (Qt)	869
PEP	120
PEP 249	644
PEP 257 (Docstrings)	121
PEP 8 (Style Guide)	121
Permutation	438
PhD thesis	162
pickle	662
PIL	984
Pillow	984
pip	766, 777
Pipe	536
Planned language element	338
Platform independence	40
Polar coordinates	502, 503
Polygon	844, 873
<i>convex</i>	844
<i>non-convex</i>	844
POP3	724
poplib	724
Port (network)	675
Positional argument	283
Positional parameter	283
Positional-only parameter	287
POST (HTTP)	701, 703, 927
Postmortem debugger	741
pow	318
PowerShell	63, 561
pprint	60, 521
Precedence (operator)	128
Preemptive multitasking	590
Prime number	603
print	59, 318, 996
Procedure	277
Process	554, 587, 630
Processor	554
Producer (queue)	620
Profiler	752
Program file	63
Programming paradigm	40
Progress bar (Qt)	865
Prompt	68
Proper subset	232
property (function)	366
Property attribute	366
Protocol layer	673
Pseudorandom number	503
PSF (organization)	40
PSF license	40
Push button (Qt)	865
PyCharm (IDE)	464, 1014
PyDev (IDE)	1015
PyGObject	806
PyPI	766, 777
pyplot (matplotlib)	942

PyPy	66, 789
PyQt	806
PySide6	806, 850
Python 2	993
<i>conversion</i>	1001
Python API	40
Python Database API Specification	644
Python debugger (PDB)	301, 739
Python distribution	42
Python Enhancement Proposal -> see PEP	120
Python Imaging Library -> see PIL	984
Python package index -> see PyPI	766
Python package Manager -> see pip	777
Python shell	45
Python Software Foundation -> see PSF	40
Python version	557
Python website	42
PYTHONHASHSEED	230
PyTorch	42
<b>Q</b>	
QML (Qt)	852
qsort	798
Qt	806, 850
<i>alpha blending</i>	877
<i>anti-aliasing</i>	878
<i>Bezier curve</i>	878
<i>brush</i>	870
<i>checkbox</i>	862
<i>combobox</i>	862
<i>coordinate system</i>	870
<i>date edit</i>	863
<i>delegate</i>	883
<i>dialog</i>	858, 863
<i>drawing</i>	868
<i>drawing text</i>	874
<i>event</i>	859
<i>event handler</i>	859
<i>gradient</i>	876
<i>images</i>	873
<i>layout</i>	851
<i>line edit</i>	864
<i>list widget</i>	864
<i>main event loop</i>	858
<i>modal dialog</i>	863
<i>model class</i>	879
<i>model-view concept</i>	851, 879
<i>non-modal dialog</i>	863
<i>painter</i>	869
<i>painter path</i>	878
<i>pen</i>	869
Radio button (Qt)	865
Radio button (tkinter)	825
Rainbow table	519
raise	401
random	503
Random access	633
range	85, 277, 319
Rapid Prototyping	41
Raspberry Pi	41
Raw string	186
raw_input	997
re	529
Real part	149
Recursion	300
<i>depth</i>	300
Reference	103, 105
<i>global</i>	293
<i>local</i>	293
Reference count	110
Reference implementation	789
RegExp -> see Regular expression	529
Regular expression	529
<i>alternative</i>	536
<i>character class</i>	530, 533
<i>character literal</i>	529
<i>extension</i>	537

## Regular expression (Cont.)

*group* ..... 536  
*match object* ..... 544  
*matching* ..... 544, 547  
*quantifier* ..... 531, 535  
*searching* ..... 546  
*special characters* ..... 534  
*syntax* ..... 529  
Relational database ..... 644  
Relational operator ..... 55  
repr ..... 320  
Request handler ..... 688  
requests ..... 702, 703  
Reserved word ..... 55, 1005  
return ..... 280  
Return value ..... 58, 115, 278, 279  
reversed ..... 320  
Root (DOM) ..... 634  
round ..... 321  
Siblings (DOM) ..... 634  
Row index ..... 955, 956  
RPM ..... 773  
Runtime measurement ..... 749  
Runtime performance ..... 749

---

**S**

Salt ..... 230, 515  
SAX (XML) ..... 640  
scikit-learn ..... 42  
Scilab ..... 935  
SciPy ..... 42, 935, 952, 953  
Scrapy ..... 618  
Screen output ..... 59  
Scrollbar (tkinter) ..... 835  
Searching ..... 529, 546  
select ..... 686  
self ..... 347  
Self-documenting expression ..... 206  
Semicolon ..... 69  
Sequence Patterns ..... 486  
Sequence unpacking ..... 180  
Sequential data type ..... 153  
  *concatenation* ..... 157  
  *indexing* ..... 159, 165  
  *length* ..... 164  
  *maximum* ..... 164  
  *minimum* ..... 164  
  *slicing* ..... 161  
Serial server ..... 675  
Serialize ..... 662  
Server ..... 675  
  *multiplexing* ..... 675, 686

Server (Cont.)

*parallel* ..... 675  
  *serial* ..... 675  
Set ..... 215, 227  
  *difference* ..... 234  
  *intersection* ..... 233  
  *proper subset* ..... 232  
  *subset* ..... 232  
  *symmetric difference* ..... 234  
set ..... 215, 227, 236, 321  
setattr ..... 370  
Setter method ..... 365  
setup tools ..... 766  
SHA ..... 516  
Shebang ..... 65, 788  
Shell ..... 43, 561  
Shortcut function ..... 915  
shutil ..... 579  
Siblings (DOM) ..... 634  
Side effect ..... 114, 175, 290, 981  
Signal (Qt) ..... 851, 859  
Simple API for XML -> see SAX ..... 640  
Sine ..... 501  
site-packages ..... 325  
Sleeping thread ..... 588  
Slicing ..... 161, 957, 961  
Slider (Qt) ..... 866  
Slot (Qt) ..... 851, 859  
SMTP ..... 721  
smtplib ..... 721  
Socket

- blocking* ..... 680
- byte order* ..... 685
- communication socket* ..... 676
- connection object* ..... 684
- connection socket* ..... 675
- IPv6* ..... 681
- listen mode* ..... 685
- non-blocking* ..... 680

socketserver ..... 688  
sorted ..... 321  
Sorting method

- stable* ..... 173

Source code ..... 63  
Source distribution ..... 765, 773  
Sources of Information ..... 119  
Special characters ..... 208, 534  
Spinbox (tkinter) ..... 836  
Splitter (Qt) ..... 855  
Spyder (IDE) ..... 1016  
SQL ..... 644  
SQL Injection ..... 650

## SQLite

adaptation ..... 658  
conversion ..... 658  
sqlite3 ..... 646  
Stable sorting method ..... 173  
Standard deviation ..... 508  
Standard dialog (tkinter) ..... 847  
Standard library ..... 40, 60, 1000  
  *argparse* ..... 561  
  *asyncio* ..... 592  
  *cmath* ..... 497  
  *cmd* ..... 972  
  *collections* ..... 239  
  *concurrent.futures* ..... 591, 630  
  *contextlib* ..... 444  
  *copy* ..... 981  
  *cProfile* ..... 752  
  *csv* ..... 668  
  *datetime* ..... 254  
  *decimal* ..... 509  
  *distutils* ..... 766, 767  
  *doctest* ..... 742  
  *ElementTree (XML)* ..... 633  
  *email* ..... 734  
  *enum* ..... 270  
  *ftplib* ..... 702, 713  
  *functools* ..... 455  
  *getpass* ..... 971  
  *gettext* ..... 781  
  *gzip* ..... 661  
  *hashlib* ..... 514  
  *http* ..... 702  
  *imaplib* ..... 728  
  *importlib* ..... 335  
  *io.StringIO* ..... 975  
  *itertools* ..... 432  
  *logging* ..... 523  
  *math* ..... 497  
  *multiprocessing* ..... 592, 611, 630  
  *os* ..... 553, 575  
  *os.path* ..... 575  
  *pickle* ..... 662  
  *poplib* ..... 724  
  *pprint* ..... 60, 521  
  *random* ..... 503  
  *select* ..... 686  
  *shutil* ..... 579, 584  
  *smtplib* ..... 721  
  *socket* ..... 674  
  *socketserver* ..... 688  
  *sqlite3* ..... 646  
  *statistics* ..... 507

Standard library (Cont.)

*struct* ..... 969  
  *sys* ..... 555  
  *tempfile* ..... 585  
  *threading* ..... 592, 603, 630  
  *time* ..... 247  
  *timeit* ..... 749  
  *Tkinter* ..... 805  
  *tkinter* ..... 807  
  *trace* ..... 756  
  *typing* ..... 465, 469, 471  
  *unittest* ..... 746  
  *urllib* ..... 702  
  *urllib.parse* ..... 710  
  *urllib.request* ..... 706  
  *urllib2* ..... 702  
  *venv* ..... 786  
  *warnings* ..... 412  
  *webbrowser* ..... 969  
  *xml* ..... 631  
  *xmlrpc* ..... 690  
  *zoneinfo* ..... 263  
standard library ..... 325  
Statement

- body* ..... 67
- header* ..... 67

Static method ..... 368  
Static typing ..... 794, 796  
staticmethod ..... 450  
staticmethod (function) ..... 368  
statistics ..... 507  
stderr ..... 556  
stdin ..... 91, 556  
stdout ..... 91, 556  
str ..... 51, 182, 322, 998  
Stream ..... 91  
String ..... 51, 182, 998  
  *control character* ..... 185  
  *escape sequence* ..... 185, 209  
  *formatting* ..... 197  
  *line break* ..... 185  
  *raw string* ..... 186  
  *special characters* ..... 208  
  *whitespace* ..... 186

String formatting ..... 197  
StringIO ..... 975  
struct ..... 969  
Structural pattern matching ..... 479  
Structured Query Language --> see SQL ..... 644  
Subgenerator ..... 418  
Subset ..... 232  
Subversion (SVN) ..... 1015

sum .....	323
Symmetric difference set .....	234
Syntax .....	66
Syntax analysis .....	632
Syntax error .....	66
sys .....	555
<b>T</b>	
Tag (Django) .....	919
Tag (XML) .....	631
<b>bodiless</b> .....	632
Tangent .....	501
TAR .....	579, 584
Task .....	616
TCP .....	678
tempfile .....	585
Template (Django) .....	915
Template inheritance (Django) .....	921
Temporary file .....	585
TensorFlow .....	42
Terminator (iteration) .....	431
Test	
<i>automated</i> .....	741
Test-driven development .....	741
Testing	
<i>doctest</i> .....	742
<i>unittest</i> .....	746
Text edit (Qt) .....	866
Text widget (tkinter) .....	837
The Qt Company .....	806
Thread .....	589, 630
<i>sleeping</i> .....	588
threading .....	592, 603, 630
Time .....	247
time (module) .....	247
Time slice .....	588
Time zone .....	263
timeit .....	749
Timestamp .....	247
Tk .....	805
Tkinter .....	805
tkinter .....	807
button .....	823
canvas .....	839
checkbox .....	824
control variable .....	810
drawing .....	839
entry widget .....	826
event .....	815
event handler .....	815
font .....	848
tkinter (Cont.)	
label .....	827
LabelFrame .....	828
listbox .....	829
menu .....	831
menu bar .....	831
menu button .....	833
message box .....	848
modifier .....	816
OptionMenu .....	834
packer .....	808, 811
padding .....	814
radio button .....	825
scrollbar .....	835
spinbox .....	836
standard dialog .....	847
text widget .....	837
widget .....	821
Toolkit (GUI) .....	805
PyGObject .....	806
Qt .....	806, 850
Tkinter .....	805
wxPython .....	807
trace .....	756
Traceback .....	400, 560
Traceback object .....	444
Tracer .....	756
Transaction (database) .....	649
Transformation (Qt) .....	878
Transmission Control Protocol .....	678
Transparency (Qt) .....	877
Trigonometric function .....	501
Trolltech .....	806
True .....	55, 144
Truth value .....	55, 147
try .....	402
Tuple	
<i>named</i> .....	245
<i>unpacking</i> .....	180
tuple .....	179, 323
Tuple packing .....	179
Two's complement .....	139
type .....	106, 125, 323
Type alias .....	474
Type checking pattern .....	482
Type comment .....	463
Type hint .....	463, 471
Type union .....	474
Type variable .....	475
Typing	
<i>static</i> .....	794

typing .....	465, 469, 471
tzdata .....	264
<b>U</b>	
UDP .....	677
u-literal .....	186
Unary operator .....	388
Unbound local variable .....	298
UnboundLocalError .....	298
Underscore .....	54
unhashable .....	217, 228
Unicode .....	210, 212
unicode (function) .....	998
UnicodeDecodeError .....	213
Uniform distribution .....	505
Uniform Resource Locator -> see URL .....	705
Unit test .....	746
unittest (module) .....	746
Unix epoch .....	247
Unix timestamp .....	247
Unpacking .....	180, 216, 229
Upper median .....	508
URL .....	705, 706, 969
urllib .....	702
urllib.parse .....	710
urllib.request .....	706
urllib2 .....	702
urllib3 .....	702
UTC .....	248, 264
<b>V</b>	
Value .....	107
<i>Boolean</i> .....	144, 147
Value comparison .....	107
Variable .....	54, 55, 105
<i>global</i> .....	589
<i>unbound local</i> .....	298
Variance .....	508
Vectorization (numpy) .....	939
venv .....	786
View (Django) .....	894, 901, 910
View class (Qt) .....	879
Virtual environment .....	785
Virtual machine .....	66
virtualenv .....	787
Visual Studio Code (IDE) .....	1015
<b>W</b>	
Wallis product .....	596, 791
Walrus operator .....	89
Warning .....	412
warnings (module) .....	412
Web API .....	704
Web crawler .....	618
webbrowser .....	969
Wheel .....	766, 795
while .....	79
Whitespace .....	95, 186, 533
whl .....	766
Widget (GUI) .....	805
Widget (Qt) .....	852, 861, 867
Widget (tkinter) .....	821
Wildcard .....	485
Window (GUI) .....	805
Winter time .....	265
with .....	441
<i>asynchronous</i> .....	619, 627
Working directory .....	569
Wrapper function .....	449, 451
WSGI .....	897
wxPython .....	807
<b>X</b>	
XLSX .....	966
XML .....	631
<i>attribute</i> .....	632
<i>bodiless tag</i> .....	632
<i>declaration</i> .....	631
<i>Document Object Model</i> .....	633
<i>DOM</i> .....	633
<i>element</i> .....	631
<i>parser</i> .....	632
<i>path</i> .....	639
<i>SAX</i> .....	640
<i>tag</i> .....	631
XML-RPC .....	690
<i>multicall</i> .....	696
XZ .....	584
<b>Y</b>	
yield .....	416, 628, 976
yield from .....	418

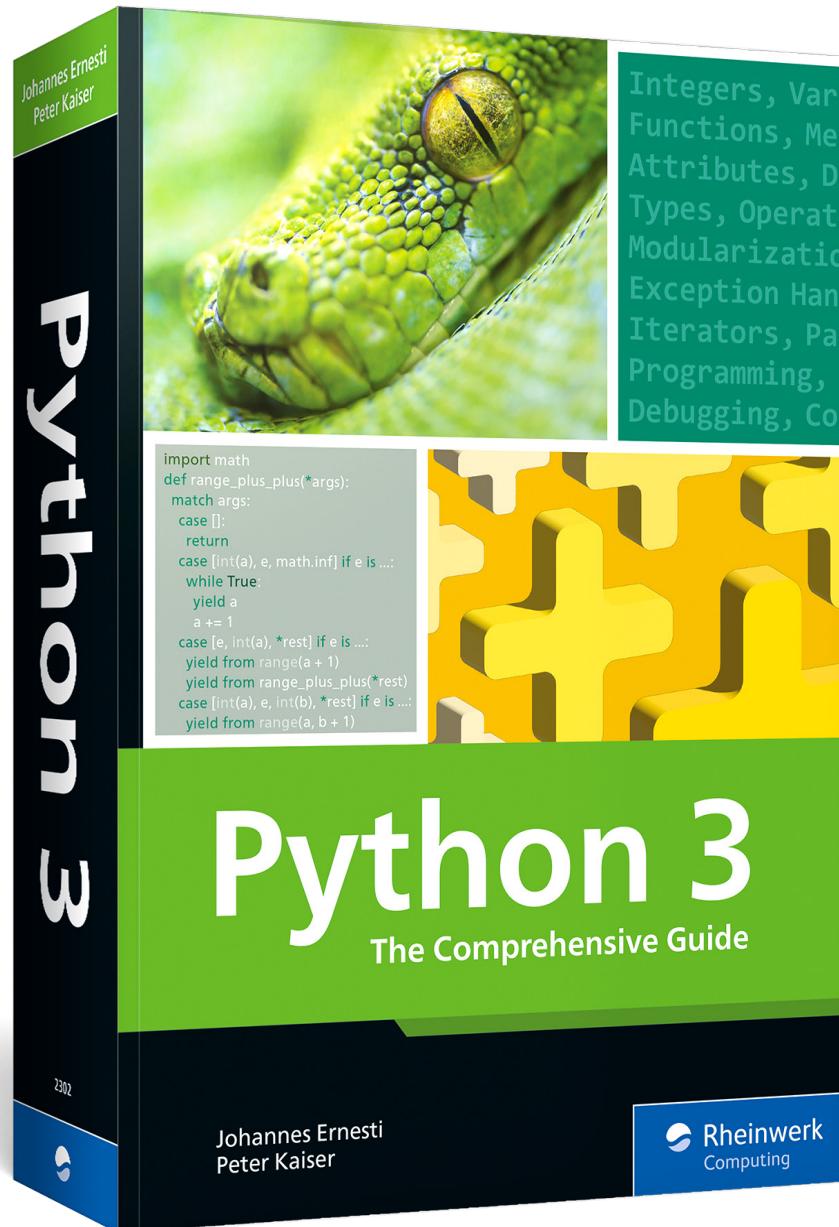
**Z**

---

- ZIP ..... 579, 584
- zip (function) ..... 324
- zlib ..... 661
- zoneinfo ..... 263

Build and deepen your coding knowledge  
from the top programming experts!

 Rheinwerk  
Computing



Johannes Ernesti, Peter Kaiser

## Python 3: The Comprehensive Guide

1036 pages, 2022, \$59.95  
ISBN 978-1-4932-2302-2

 [www.rheinwerk-computing.com/5566](http://www.rheinwerk-computing.com/5566)



**Johannes Ernesti** and **Peter Kaiser** received their doctorates in mathematics and computer science, respectively, from the Karlsruhe Institute of Technology (KIT).

They have been developing Python software professionally and privately for more than 20 years, currently as part of their research in neural machine translation at DeepL.

This book served as the basis for several trainings in companies and universities. At KIT, a Python lecture based on this book has been held annually since 2015.

*We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.*