

# Chapter 5. Numbers and Expressions

This chapter begins our in-depth tour of the Python language. In Python, data takes the form of *objects*—either built-in objects that Python provides or objects we create using Python tools and other languages like C. In fact, objects are the basis of every Python program you will ever write. Because they are the most fundamental notion in Python programming, objects are also our first focus in this book.

In the preceding chapter, we took a quick first pass over Python’s core object types. Although essential terms were introduced in that chapter, we avoided covering too many specifics in the interest of space. Here, we’ll begin a more careful second look at object concepts, to fill in details we glossed over earlier. Let’s get started by exploring our first category: Python’s numeric objects and operations.

## Numeric Object Basics

Most of Python’s numeric support is fairly typical and will probably seem familiar if you’ve used almost any other programming language in the past. They can be used to keep track of your bank balance, the distance to Mars, the number of visitors to your website, and just about any other numeric quantity.

In Python, numbers are not really a single object type, but a category of similar types. Python supports the usual numeric types (integers and floating points), as well as literals for creating numbers and expressions for processing them. In addition, Python provides more advanced numeric programming support and objects for more advanced needs. A fairly complete inventory of Python’s numeric toolbox includes:

- Integer and floating-point objects
- Complex number objects
- Decimal fixed-precision objects
- Fraction rational number objects
- Set objects and operations

- Boolean and bitwise operations
- Built-in modules, such as `math`, `cmath`, `random`, and `statistics`
- Third-party add-ons, including vectors, visualization, plotting, and extended precision

Because the object types in this list’s first bullet item tend to see the most action in Python code, this chapter starts with basic numbers and fundamentals, then moves on to explore other types on this list, which serve specialized roles. We’ll also study *sets* here, which have both numeric and collection qualities, but are generally considered more the former than the latter. Before we jump into code, though, the next few sections get us started with a brief overview of how we write and process numbers in our scripts.

## Numeric Literals

Among its basic object types, Python provides *integers*, which are positive and negative whole numbers, and *floating-point* numbers, which are numbers with a fractional part (sometimes called *floats* for verbal economy). Python also allows us to write integers using hexadecimal, octal, and binary literals; offers a complex number type; and allows integers to have unlimited *precision* —they can grow to have as many digits as your memory space allows.

[Table 5-1](#) shows what Python’s numeric types look like when written out in a program as literals or constructor-function calls.

Table 5-1. Numeric literals and constructors

Literal	Interpretation
<code>1234</code> , <code>-24</code> , <code>0</code> , <code>9_999_999_999_999</code>	Integers (unlimited size)
<code>1.23</code> , <code>1.</code> , <code>3.14e-10</code> , <code>4E210</code> , <code>4.0e+210</code>	Floating-point numbers
<code>0o177</code> , <code>0x9ff</code> , <code>0b101010</code>	Octal, hex, and binary literals
<code>3+4j</code> , <code>3.0+4.0j</code> , <code>3J</code>	Complex number literals
<code>set('hack')</code> , <code>{1, 2, 3, 4}</code>	Sets: constructors and literals
<code>Decimal('1.0')</code> , <code>Fraction(1, 3)</code>	Decimal and fraction extension types
<code>bool(X)</code> , <code>True</code> , <code>False</code>	Boolean type and constants

In general, Python's numeric type literals are straightforward to write, but a few coding concepts are worth highlighting up front:

### *Integer and floating-point literals*

Integers are written as strings of decimal digits. As noted, they have precision (number of digits) limited only by your device's available memory. You can easily compute 2 raised to the power 1,000,000, though with 300K digits, it may take some time to print (and can't be converted to a print string today by default, per [Chapter 4](#)).

Floating-point numbers have a decimal point and/or an optional signed exponent introduced by an `e` or `E` and followed by an optional sign. If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when the object is used in an expression. As you'll learn, mixing a floating-point number with an integer does floating-point math too, after converting the integer up.

### *Hexadecimal, octal, and binary literals*

Integers may be coded in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2), the last three of which are common in some programming domains. Hexadecimals start with a leading `0x` or `0X`, followed by a string of hexadecimal digits (`0 – 9` and `A – F`). Hex digits may be coded in lowercase or uppercase. Octal literals start with a leading `0o` or `0O` (zero and lowercase or uppercase letter `o`), followed by a string of octal digits (`0 – 7`). Binary literals begin with a leading `0b` or `0B`, followed by binary digits (`0 – 1`).

All of these literals produce integer objects in program code; they are just alternative syntaxes for specifying values. The built-in calls `hex(I)`, `oct(I)`, and `bin(I)` convert an integer to its representation string in these three bases, and `int(str, base)` converts a runtime string to an integer per a given base.

### *Complex numbers*

Though used more rarely, Python complex literals are written as `realpart+imaginarypart`, where the `imaginarypart` is terminated with a `j` or `J`. The `realpart` is technically optional, so the `imaginarypart` may appear on its own. Internally, complex

numbers are implemented as pairs of floating-point numbers, but all numeric operations perform complex math when applied to complex numbers. Complex numbers may also be created with the `complex(real, imag)` built-in call.

### *Coding other numeric types*

As you'll see later in this chapter, there are additional numeric types near the end of [Table 5-1](#) that serve more advanced or specialized roles. You create some of these by calling functions in imported modules (e.g., decimals and fractions), others have literal syntax all their own (e.g., sets), and Booleans are a kind of specialized integer.

## Built-in Numeric Tools

Besides the built-in number literals and construction calls shown in [Table 5-1](#), Python provides a set of tools for processing number objects:

### *Expression operators*

`+`, `-`, `*`, `/`, `>>`, `**`, `&`, `%`, etc.

### *Built-in mathematical functions*

`pow`, `abs`, `round`, `int`, `hex`, `bin`, etc.

### *Utility modules*

`random`, `math`, `statistics`, etc.

You'll meet all of these as we go along.

Although numbers are primarily processed with expressions, built-ins, and modules, they also have a handful of type-specific *methods* today, which you'll meet in this chapter. Floating-point numbers, for example, have an `as_integer_ratio` method useful for the fraction number type, and an `is_integer` method to test if the number is an integer. Integer attributes include a `bit_length` method that gives the number of bits necessary to represent the object's value, and as part collection and part number, *sets* support both expressions and methods too.

Since expressions are the most essential tool for most number types, though, let's turn to them next.

# Python Expression Operators

The most fundamental tool that processes numbers is the *expression*: a combination of numbers (or other objects) and operators that computes a value when executed by Python. In Python, you write expressions using the usual mathematical notation and operator symbols. For instance, to add two numbers `X` and `Y` you would say `X + Y`, which tells Python to apply the `+` operator to the values named by `X` and `Y`. The result of the expression is the sum of `X` and `Y`, another number object.

[Table 5-2](#) lists all the operator expressions available in Python, abstractly.

Many are self-explanatory; for instance, the usual mathematical operators (`+`, `-`, `*`, `/`, and so on) are supported. A few will be familiar if you've used other languages in the past: `%` computes a division remainder, `<<` performs a bitwise left-shift, `&` computes a bitwise AND result, and so on. Others are more Python specific, and not all are numeric in nature: for example, the `is` operator tests object identity (i.e., same address in memory, a strict form of equality), and `lambda` creates unnamed functions.

Table 5-2. Python expression operators, by increasing precedence (binding)

Operators	Description
<code>yield x, yield from x</code>	Generator function <code>send</code> protocol
<code>x := y</code>	Assignment expression
<code>lambda args: expression</code>	Anonymous function generation
<code>n</code>	
<code>x if y else z</code>	Ternary selection ( <code>x</code> is evaluated only if <code>y</code> is true)
<code>x or y</code>	Logical OR ( <code>y</code> is evaluated only if <code>x</code> is false)
<code>x and y</code>	Logical AND ( <code>y</code> is evaluated only if <code>x</code> is true)
<code>not x</code>	Logical negation
<code>x in y, x not in y</code>	Membership (iterables)
<code>x is y, x is not y</code>	Object identity tests
<code>x &lt; y, x &lt;= y, x &gt; y, x &gt;= y</code>	Magnitude comparison, set subset and superset
<code>x == y, x != y</code>	Value equality operators
<code>x   y</code>	Bitwise OR, set union, dictionary merge
<code>x ^ y</code>	Bitwise XOR, set symmetric difference
<code>x &amp; y</code>	Bitwise AND, set intersection
<code>x &lt;&lt; y, x &gt;&gt; y</code>	Shift <code>x</code> left or right by <code>y</code> bits
<code>x + y</code>	Addition, concatenation
<code>x - y</code>	Subtraction, set difference
<code>x * y</code>	Multiplication, repetition
<code>x % y</code>	Remainder, format
<code>x / y, x // y</code>	Division: true and floor
<code>x @ y</code>	Matrix multiplication (unused by Python)
<code>-x, +x, ~x</code>	Negation, identity
	Bitwise NOT (inversion)
<code>x ** y</code>	Power (exponentiation)

Operators	Description
<code>await x</code>	Await expression (async functions)
<code>x[i]</code>	Indexing (sequence, mapping, others)
<code>x[i:j:k]</code>	Slicing
<code>x(...)</code>	Call (function, method, class, other callable)
<code>x.attr</code>	Attribute reference
<code>(...)</code>	Tuple, expression, generator expression
<code>[...]</code>	List, list comprehension
<code>{...}</code>	Dictionary, set, dictionary and set comprehensions

While [Table 5-2](#) works as a reference, some of its operators won’t make sense until you’ve seen them in action, and some are more subtle than the table may imply. For instance:

- Parentheses are required for `yield` if it’s not alone on the right side of an assignment statement, as well as the `:=` named-assignment operator if it’s used in some contexts.
- Comparison operators compare all parts of collections automatically and may be chained as a shorthand and potential optimization (e.g., `X < Y < Z` produces the same result as `X < Y` and `Y < Z`).
- Python defines an `@` operator meant for matrix multiplication but does not provide an implementation for it; unless you code one in a class or use a library that does, this operator does *nothing*.
- The parentheses used for tuples, expressions, and generators may sometimes be omitted; when omitted for tuples, the *comma* separating its items acts like a lowest-precedence operator if not otherwise significant.
- Some operators, like `yield`, `lambda`, and `await`, have to do with larger topics that have little to do with numbers and can safely be ignored at this early point in your Python career.

This book will defer to Python’s manuals for other minutiae, but you’ll see most of the operators in [Table 5-2](#) in action later. First, though, we need to take a quick look at the ways these operators may be combined in expressions.

---

**NOTE**

*Meet the Python no-ops:* The @ character is used by Python to introduce function decorators (covered later in this book), but not as an expression operator—despite its being specified as such. In the latter role, @ joins **Ellipsis** (...) and *type hinting* as tools defined but wholly unused by Python itself. As if you didn’t have enough to learn with the real stuff!

---

## Mixed Operators: Precedence

As in most languages, in Python, you code more complex expressions by stringing together the operator expressions in [Table 5-2](#). For instance, the sum of two multiplications might be written as a mix of variables and operators:

A \* B + C \* D

Which raises the question: how does Python know which operation to perform first? The answer to this question lies in *operator precedence*. When you write an expression with more than one operator, Python groups its parts according to what are called *precedence rules*, and this grouping determines the order in which the expression’s parts are computed. To denote this, [Table 5-2](#) is ordered by operator precedence:

- Operators *lower* in the table have higher precedence, and so bind more *tightly* in mixed expressions. Put another way, operators *higher* in the table have lower precedence and bind less tightly than those below them.
- Operators in the *same row* in the table generally group from *left to right* when combined (except for exponentiation, which groups right to left, and comparisons, which chain left to right).

So, for example, if you write X + Y \* Z, Python evaluates the multiplication first (Y \* Z) then adds that result to X, because \* has higher precedence (is lower in the table) than +. Similarly, in this section’s original example, both multiplications (A \* B and C \* D) will happen before their results are added because + is above \*.

## Parentheses Group Subexpressions

You can largely forget about precedence rules if you’re careful to group parts of expressions with parentheses. When you enclose subexpressions in parentheses, you override Python’s precedence rules; Python always evaluates expressions in parentheses *first* before using their results in the enclosing expressions.

For instance, instead of coding `X + Y * Z`, you could write one of the following to force Python to evaluate the expression in either desired order:

```
(X + Y) * Z  
X + (Y * Z)
```

In the first case, `+` is applied to `X` and `Y` first, because this subexpression is wrapped in parentheses. In the second case, the `*` is performed first (just as if there were no parentheses at all). Generally speaking, adding parentheses in large expressions is a good idea—it not only forces the evaluation order you want, but also aids readability.

## Mixed Types Are Converted Up

Besides mixing operators in expressions, you can also mix numeric types. For instance, you can add an integer to a floating-point number:

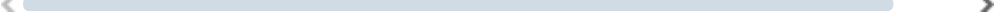
```
40 + 3.14
```

But this leads to another question: what type is the result—integer or floating point? The answer is simple, especially if you’ve used almost any other language before: in mixed-type numeric expressions, *operands* (the parts of the expression that aren’t operators) are first converted *up* to the type of the most complicated operand, and then the math is performed on same-type operands. The result is that of the up-converted operands.

For this, Python ranks the complexity of numeric types like so: integers are simpler than floating-point numbers, which are simpler than complex numbers. So, when an integer is mixed with a floating point, as in the preceding example, the integer is converted up to a floating-point value first,

and floating-point math yields the floating-point result. See for yourself in your local Python REPL:

```
>>> 40 + 3.14      # Integer to float, float math/ress  
43.14
```



Similarly, any mixed-type expression where one operand is a complex number results in the other operand being converted up to a complex number, and the expression yields a complex result. Conversions also run in equality and magnitude comparisons: `3 == 3.0` is true, but `3 > 3.0` is not.

You can force the issue by calling built-in functions to convert types manually:

```
>>> int(3.1415)    # Truncates float to integer  
3  
>>> float(3)      # Converts integer to float  
3.0
```

However, you won't usually need to do this: because Python automatically converts up to the more complex type within an expression, the results are normally what you want.

While automatic conversions are run for both numeric and comparison operators, keep in mind that they apply only when mixing *numeric* objects (e.g., an integer and a float) in an expression. In general, Python does *not* convert across any other type boundaries automatically. Adding a string of digits to an integer, for example, results in an error, unless you manually convert one or the other; watch for an example and rationale when we explore strings in [Chapter 7](#). Equality tests do work on mixed types (e.g., a string is never equal to any integer), but magnitude comparisons do not.

## Preview: Operator Overloading and Polymorphism

Although we're focusing on built-in numbers right now, all Python operators may be *overloaded* (i.e., implemented) by Python classes and C extension types to work on objects you create. For instance, you'll see later that objects coded with classes may be added or concatenated with `x+y` expressions, indexed with `x[i]` expressions, and so on.

Furthermore, Python itself automatically overloads some operators, such that they perform different actions depending on the type of built-in objects being processed. For example, the `+` operator performs addition when applied to numbers but performs concatenation when applied to sequence objects like strings and lists. In fact, `+` can mean anything at all when applied to objects you define with classes.

As we saw in the prior chapter, this property is usually called *polymorphism*—a term indicating that the meaning of an operation depends on the type of the objects being processed. We'll revisit this concept when we explore functions in [Chapter 16](#), because it becomes a much more obvious feature in that context.

# Numbers in Action

On to the code! Probably the best way to understand numeric objects and expressions is to see them in action, so with all the preceding basics in hand, let's start up the interactive command line and try some simple but illustrative operations (be sure to see [Chapter 3](#) for pointers if you need help starting a REPL).

# Variables and Basic Expressions

First of all, let's do the math to demo some basics. In the following interaction, we first assign two variables (`a` and `b`) to integers so we can use them later in a larger expression. *Variables* are simply names—created by you or Python—that are used to keep track of information in your program. We'll say more about this in the next chapter, but in Python:

- Variables are created when they are first assigned values.
  - Variables are replaced with their values when used in expressions.
  - Variables must be assigned before they can be used in expressions.
  - Variables refer to objects and need not be declared ahead of time.

In other words, these assignments cause the variables `a` and `b` to spring into existence automatically:

```
$ python3 # Fire up a REPL  
>>> a = 3 # Name created: no need to c
```

```
>>> b = 4
```

This code also uses *comments*. Recall from [Chapter 3](#) that in Python code, text after a `#` mark and continuing to the end of the line is considered to be a comment and is ignored by Python. Comments are one way to write human-readable documentation for your code, and an important part of programming. They describe aspects of the code, salient or subtle, as an aid for others (and you, six months down the road!). In the next part of the book, you'll also meet a related feature—*documentation strings*—that attaches docs to objects so it's available after your code is loaded.

Again, though, because code you type interactively is temporary, you won't normally write comments in this context. If you're working along, this means you don't need to type any of the comment text from the `#` through to the end of the line in a REPL; it's not a required part of the statements we're running this way.

Now, let's use our new integer objects in expressions. At this point, the values of `a` and `b` are still `3` and `4`, respectively. Variables like these are replaced with their values whenever they're used inside an expression, and the expression results are echoed back immediately and automatically when we're working interactively:

```
>>> a + 1, a - 1      # Addition (3 + 1), subtract  
(4, 2)  
>>> b * 3, b / 2      # Multiplication (4 * 3), di  
(12, 2.0)  
>>> a % 2, b ** 2      # Modulus (remainder), power  
(1, 16)  
>>> 2 + 4.0, 2.0 ** b    # Mixed-type conversions  
(6.0, 16.0)
```

Per [Chapter 3](#), the results being echoed back here are *tuples* of two values because the lines typed at the prompt contain two expressions separated by commas; that's why the results are displayed in parentheses. More importantly, these expressions work because the variables `a` and `b` within them have been assigned values. If you use a different variable that has *not yet been assigned*, Python reports an error rather than filling in some default value:

```
>>> c * 2  
NameError: name 'c' is not defined
```

As also previewed in [Chapter 3](#), you don’t need to predeclare variables in Python, but they must have been assigned at least once before you can use them. In practice, this means you have to initialize counters to zero before you can add to them, initialize lists to an empty list before you can append to them, and so on.

Here are two slightly larger expressions to illustrate operator grouping and more about conversions:

```
>>> b / 2 + a          # Same as ((4 / 2) + 3)  
5.0  
>>> b / (2 + a)      # Same as (4 / (2 + 3))  
0.8
```

In the *first* expression, there are no parentheses, so Python automatically groups the components according to its precedence rules—because `/` is lower in [Table 5-2](#) than `+`, it binds more tightly and so is evaluated first. The result is as if the expression had been organized with parentheses as shown in the comment to the right of the code. In the *second* expression, parentheses are added around the `+` part to force Python to evaluate it first (i.e., before the `/`).

Also, notice that all the numbers are integers in each of these examples.

Python’s `/` performs *true* division, which always retains fractional remainders and gives a floating-point result—which is in turn reflected in the result of the whole expression. You can force a fractional result by coding `2.0` instead of `2` but don’t have to. You can also opt to use *floor* division by coding these examples with `//` instead of `/`, and Python will discard decimal digits in the result; because results reflect the types of operands, you’ll get back truncated floating-point for floats:

```
>>> a, b          # Same, original values  
(3, 4)  
  
>>> b // 2 + a    # Floor division: integer  
5  
>>> b // (2 + a)  # Truncates fraction (for pc
```

0

```
>>> b // 2.0 + a          # Auto-conversions: floating
5.0
>>> b // (2.0 + a)
0.0
```

You'll learn more about division later in this section.

## Numeric Display Formats

Once you start playing with Python numbers in earnest, the results of some expressions may look a bit odd the first time you see them:

```
>>> 1.1 + 2.2          # What's up with the 3 at t
3.3000000000000003
>>> print(1.1 + 2.2)    # Same for prints
3.3000000000000003
```

The full story behind this odd result has to do with the limitations of floating-point hardware and its inability to exactly represent some values in a limited number of bits. Python floating-point numbers map to the underling chips on your device and are only as accurate as those chips allow—a physical constraint that can be addressed with add-ons that extend floating-point precision, as well as techniques discussed in [“Floating-point equality”](#).

Because computer architecture is well beyond this book's scope, though, we'll finesse this by saying that your computer's floating-point hardware is doing the best it can, and neither it nor Python is in error here. In fact, this is partly a *display* issue—Python's floating-point display logic tries to be intelligent and usually shows fewer decimal digits, but occasionally cannot. Our earlier examples gave fewer digits automatically, and you can always force the issue in programs with string formatting:

```
>>> num = 1.1 + 2.2
>>> num                      # Auto-echoes (and pri
3.3000000000000003

>>> '%e' % num                # String-formatting ex
'3.300000e+00'
```

```
>>> '%.1f' % num                      # Alternative floating
      '3.3'

>>> f'{num:e}', f'{num:.1f}'        # F-strings (see also
('3.300000e+00', '3.3')
```

The last three tests here employ flexible string formatting, which we will explore in full in the upcoming chapter on strings ([Chapter 7](#)). Its results are strings that are typically, but not always, printed to displays or reports.

---

#### DISPLAY FORMATS: STR AND REPR

Although it's not yet obvious in this chapter, the default output format of interactive echoes and `print` technically correspond to the built-in `repr` and `str` functions, respectively:

```
>>> repr('hack')                      # Used by echoes: as-code fc
"'hack'"
>>> str('hack')                       # Used by print: user-friendly
'hack'
```



Both of these convert arbitrary objects to their string representations: `repr` (and the default interactive echo) produces results that look as though they were code; `str` (and the `print` operation) converts to a typically more user-friendly format if available. Some objects have both—a `str` for general use, and a `repr` with extra details. This notion will resurface when we explore both strings and operator overloading in classes.

Besides providing print strings for arbitrary objects, the `str` built-in is also the name of the string object type, which can be called with an encoding name to decode a Unicode string from a byte string as an alternative to the `bytes.decode` method introduced briefly in [Chapter 4](#). You'll learn more about this advanced `str` role in [Chapter 37](#).

---

## Comparison Operators

So far, we've been dealing with standard numeric operations (e.g., addition and multiplication), but numbers, like all Python objects, can also be compared. Normal comparisons work for numbers exactly as you'd expect—

they compare the relative magnitudes of their operands and return a Boolean result, which we would normally test and take action on in a larger statement and program (e.g., see the intro to `if` in [Chapter 4](#)):

```
>>> 1 < 2                      # Less than (magnitude)
True
>>> 2.0 >= 1                  # Greater than or equal: mixed type
True
>>> 2.0 == 2.0                # Equal value
True
>>> 2.0 != 2.0                # Not equal value
False
```

Notice again how mixed types are allowed in numeric expressions (only); in the second test here, Python compares values in terms of the more complex type, float.

## Chained comparisons

Interestingly, Python also allows us to *chain* multiple comparisons together to perform range tests. Chained comparisons are a sort of shorthand for larger Boolean expressions. In short, Python lets us string together magnitude comparison tests to code chained comparisons such as range tests. The expression `(A < B < C)`, for instance, tests whether `B` is between `A` and `C`, noninclusively; it is equivalent to the Boolean test `(A < B and B < C)` but is easier on the eyes (and the keyboard). For example, assume the following assignments:

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

The following two expressions have identical effects, but the first is shorter to type, and it may run slightly faster since Python needs to evaluate `Y` only once—and it may matter if `Y` is a call to a complicated function:

```
>>> X < Y < Z                  # Chained comparisons: range
True
```

```
>>> X < Y and Y < Z
```

```
True
```

The same equivalence holds for false results, and arbitrary chain lengths are allowed:

```
>>> X < Y > Z
```

```
False
```

```
>>> X < Y and Y > Z
```

```
False
```

```
>>> 1 < 2 < 3.0 < 4
```

```
True
```

```
>>> 1 > 2 > 3.0 > 4
```

```
False
```

You can use other comparisons in chained tests, but the resulting expressions can become nonintuitive unless you evaluate them the way Python does. The first of the following, for instance, is false just because 1 is not equal to 2:

```
>>> 1 == 2 < 3          # Same as: (1 == 2) and (2 < 3)
False
>>> True is False is True    # Same as: (True is False) and (False is True)
False
```

Python does not compare the `1 == 2` expression's `False` result to 3—this would technically mean the same as `0 < 3`, which would be `True` (you'll learn more about the `True` and `False` objects later in this chapter and explore the rarely used `is` identity operator in the next).

## Floating-point equality

One last note here before we move on: chaining aside, numeric comparisons are based on magnitudes, which are generally simple—though *floating-point* numbers may not always work as you'd expect, and may require conversions or other massaging to be compared meaningfully:

```
>>> 1.1 + 2.2 == 3.3          # Shouldn't this be 1?
False
```

```
>>> 1.1 + 2.2                      # Close to 3.3, but r
3.3000000000000003
```

This is related to the earlier coverage of numeric display formats and stems from the fact that floating-point numbers cannot represent some values exactly due to their limited number of bits—a fundamental issue in numeric programming not unique to Python. To accommodate this imprecision in equality tests, either truncate, round, use floors, or, as of Python 3.5, import and call the `math` standard-library module’s `isclose`, which is true if values are within a tolerance of each other (there’s more on `math` and floors ahead, and more on `isclose` in Python’s manuals):

```
>>> int(1.1 + 2.2) == int(3.3)      # OK if convert:
True
>>> round(1.1 + 2.2, 1) == round(3.3, 1)
True
>>> import math                      # Import modules
>>> math.isclose(1.1 + 2.2, 3.3)     # Within default-
True
```

We’ll revisit this later in this chapter when we meet *decimals* and *fractions*, which can also address such limitations. First, though, let’s continue our tour of Python’s core numeric operations, with a deeper look at division.

## Division Operators

Python has two division operators introduced earlier, as well as one that’s strongly related. Here’s the whole gang:

$X / Y$

Called *true* division, this always keeps remainders in floating-point results, regardless of types.

$X // Y$

Called *floor* division, this always truncates fractional remainders down to their floor, regardless of types, and its result type depends on the types of its operands.

$X \% Y$

Called *modulus*, this returns a division’s remainder, with a result type that varies per operand types. This also does formatting when used on

strings, per [Chapter 7](#).

The following demos the two *division* operators at work:

```
>>> 10 / 4          # True div: keeps remainder alw
2.5
>>> 10 / 4.0        # Same for floats
2.5
>>> 10 // 4          # Floor div: drops remainder alw
2
>>> 10 // 4.0        # Same for floats, but type var
2.0
```

Notice that the object type of the result for `//` is dependent on its operand's types: if either is a float, the result is a float; otherwise, it is an integer. If you want to ensure an integer result, simply wrap the expression in `int` to convert:

```
>>> int(10 // 4.0)
2
```

The related *modulus* returns the remainder of division with a type to match operands (useful when your code needs to know how much is “left over” after a `//`), and the `divmod` built-in function gives both parts when needed:

```
>>> 10 % 3, 10 % 3.0    # Remainder of division: (3 * 3)
(1, 1.0)
>>> divmod(10, 3)       # Both parts of division in a tuple
(3, 1)
```

## Floor versus truncation

One subtlety here: the `//` operator is informally called *truncating* division, but it's more accurate to refer to it as *floor* division—it truncates the result down to its floor, which means the closest whole number below the true result. The net effect is to round down, not strictly truncate, and this matters for negatives. You can see the difference for yourself with the Python `math` module (as you've learned, modules must be imported before you can use their contents):

```
>>> import math
>>> math.floor(2.5)                      # Closest number below v
2
>>> math.floor(-2.5)                     # But not truncation for
-3
>>> math.trunc(2.5)                      # Truncate fractional pa
2
>>> math.trunc(-2.5)                     # And is truncation for r
-2
```



When running division operators, you only really truncate for positive results, since truncation is then the same as floor; for negatives, it's a floor result.

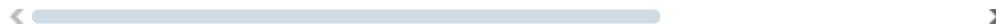
Really, they are both floor, but floor just happens to be the same as truncation for positives (*cut-and-pasters*: if minus signs morph to Unicode dashes in this book, replace them with simple ASCII “-” hyphens to run; Python requires the latter for the minus sign, and tools are notorious for botching this):

```
>>> 5 / 2, 5 / -2                      # True division keeps ren
(2.5, -2.5)

>>> 5 // 2, 5 // -2                     # Truncates to floor: rou
(2, -3)

>>> 5 / 2.0, 5 / -2.0                  # Ditto for floats
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0                # Though result is float
(2.0, -3.0)
```



If you really want truncation toward zero regardless of sign, you can always run a true division result through `math.trunc` (as demoed earlier, the `round` built-in has related functionality and the `int` built-in has the same effect, and neither requires an import):

```
>>> import math
>>> 5 / -2                            # Keep remainder
-2.5
>>> 5 // -2                           # Floor below result
-3
```

```
>>> math.trunc(5 / -2)          # Truncate instead of flc  
-2
```

So why the fuss over truncation? This won't be obvious until you graduate to writing larger Python programs later in this book, but it's an essential tool in some use cases. Watch for a prime-number `while` loop example in [Chapter 13](#) and a corresponding exercise at the end of [Part IV](#) that wholly rely on the truncating behavior of `//`.

## Integer Precision

Python division may come in multiple flavors, but it's still fairly standard as programming languages go. Here's something a bit more unusual. As mentioned earlier, Python integers support unlimited size:

```
>>> 99999999999999999999999999999999 + 1  
100000000000000000000000000000000000000000
```

Unlimited-precision integers are a convenient built-in tool. For instance, you can use them to count your country's national debt in Python without numeric-value overflow (which is, of course, more impressive and resource intensive in some locales than others). More universally, a 2 raised to the power 269 isn't particularly large, but nearly breaches this page's width limits, and much larger numbers work sans the safeguards against DOS attacks noted in [Chapter 4](#):

```
>>> 2 ** 269  
9485687950320942729098935091911713411339877143809275006  
  
>>> x = 2 ** 1000000  
>>> x  
ValueError: Exceeds the limit (4300 digits) for integer  
use sys.set_int_max_str_digits() to increase the limit
```

Because Python must do extra work to support the extended precision, integer math is usually substantially slower than normal when numbers grow large. However, if you need the precision, the fact that it's built in for you to use will likely outweigh its performance penalty.

## Complex Numbers

Although less commonly used than the types we've been exploring thus far, complex numbers are a distinct core object type in Python. They are typically used in engineering and science applications. If you know what they are, you know why they are useful; if not, consider this section optional reading (until they appear in code you must reuse).

Complex numbers are represented as two floating-point numbers—the real and imaginary parts—and you code them by adding a `j` or `J` suffix to the imaginary part. We can also write complex numbers with a nonzero real part by adding the two parts with a `+`. For example, the complex number with a real part of `2` and an imaginary part of `-3` is written `2 + -3j`. Here are some examples of complex math at work:

```
>>> 1j * 1j
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
```

Complex numbers also allow us to extract their parts as attributes (via attributes `real` and `imag`), support all the usual mathematical expressions, and may be processed with tools in the standard `cmath` module (the complex analogue of the standard `math` module). Because complex numbers are rare in most programming domains, though, we'll skip the rest of this story here. Check Python's language reference manual for additional details.

## Hex, Octal, and Binary

As previewed near the start of this chapter, Python integers can be coded in hexadecimal, octal, and binary notation, in addition to the normal base-10 decimal coding we've been using so far. The first three of these may at first seem foreign to 10-fingered beings, but some programmers find them convenient alternatives for specifying values, especially when their mapping to bytes and bits is important. We detailed coding rules before; let's try these out live.

As noted earlier, these other-base *literals* are simply an alternative syntax for specifying the value of an integer object. For example, the following literals produce normal integers with the specified values. In memory, an integer's value is the same, regardless of the base we use to specify it in our code:

```
>>> 0x01, 0x10, 0xFF          # Hex Literals: base 16
(1, 16, 255)
>>> 0o1, 0o20, 0o377        # Octal Literals: base
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111 # Binary Literals: base
(1, 16, 255)
```

Here, the hex value `0xFF`, the octal value `0o377`, and the binary value `0b11111111` are all decimal 255. The F digits in the hex value, for example, each mean 15 in decimal and a 4-bit 1111 in binary, and reflect powers of 16. Thus, the hex value `0xFF` and others convert to decimal values as follows:

```
>>> 0xFF, (15 * (16 ** 1)) + (15 * (16 ** 0))      # How
(255, 255)
>>> 0x2F, (2 * (16 ** 1)) + (15 * (16 ** 0))
(47, 47)
>>> 0xF, 0b1111, (1*(2**3) + 1*(2**2) + 1*(2**1) + 1*(2
(15, 15, 15)
```

Python prints integer values in decimal (base 10) by default, but it also provides built-in functions that convert integers to other bases' digit strings formatted per Python-literal syntax—useful when programs or users expect to see values in a given base:

```
>>> oct(64), hex(64), bin(64)          # Numbers=>
('0o100', '0x40', '0b1000000')
```

The `oct` function converts decimal to octal, `hex` to hexadecimal, and `bin` to binary—all as strings. To go the other way, the built-in `int` function converts a string of digits to an integer, and an optional second argument lets you specify the numeric base—useful for numbers read from files as strings instead of coded in scripts:

```
>>> 64, 0o100, 0x40, 0b1000000          # Digits=>r
(64, 64, 64, 64)

>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)    # Literal j
(64, 64, 64)
```

The `eval` function can also be used to convert digit strings to numbers, because it treats strings as though they were Python code. Therefore, it has a similar effect, but usually runs more *slowly*—it actually compiles and runs the string as a piece of a program, and it assumes the string being run comes from a *trusted source*—a clever user might be able to submit a string that deletes files on your machine. In other words, be sparing and careful with this call:

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')      # Numbers=>r
(64, 64, 64, 64)
```

Finally, you can also convert integers to base-specific strings with any of Python’s three *string-formatting* tools, though you’ll have to take this partly on faith until we reach strings’ full coverage in [Chapter 7](#):

```
>>> '%o, %x, %#X' % (64, 255, 255)          # Numbers=>r
'100, ff, 0xFF'

>>> '{:o}, {:b}, {:x}, {:#X}'.format(64, 64, 255, 255)
'100, 1000000, ff, 0xFF'

>>> f'{64:o}, {64:b}, {255:x}, {255:#X}'      # The newest
'100, 1000000, ff, 0xFF'
```

In this code, `o`, `b`, and `x` format as octal, binary, and hex, respectively, and `#X` adds a base prefix and uses uppercase. As an aside, you can avoid the repeated inputs in each of these three formatting tools (but you’re probably starting to see why picking just one is generally a good idea—and why feature redundancy is generally a bad idea!):

```
>>> '%(i)o, %(j)x, %(j)#X' % dict(i=64, j=255)
'100, ff, 0xFF'
>>> '{0:o}, {0:b}, {1:x}, {1:#X}'.format(64, 255)
'100, 1000000, ff, 0xFF'
>>> f'{(i:=64):o}, {(i):b}, {((i:=255)):x}, {(i):#X}'
'100, 1000000, ff, 0xFF'
```

Before we move on, keep in mind that other-base literals and converters support *arbitrarily* large integers too. The following, for instance, creates an integer in hex and displays it in decimal and octal and binary with converters:

Speaking of binary digits, the next section takes us on a tour of tools that process numbers' individual bits.

# Bitwise Operations

Besides the normal numeric operations (addition, subtraction, and so on), Python supports most of the numeric expressions available in the C language. This includes operators that treat integers as strings of *binary bits* and can come in handy if your Python code must deal with things like network packets, serial ports, or packed binary data produced by or intended for a C program.

We can't dwell on the fundamentals of Boolean math here—again, those who must use it probably already know how it works, and others can often postpone the topic altogether—but the basics are straightforward. For instance, here are some of Python's bitwise expression operators at work performing bitwise shift and Boolean operations on integers:

```
>>> x = 1                      # 1 decimal is 0001 in bits  
>>> x << 2                    # Shift left 2 bits: 0100
```

```

4
>>> x | 3                      # Bitwise OR (either bit=1): 6
3
>>> x & 3                      # Bitwise AND (both bits=1): 6
1

```

In the first expression, a binary 1 (in base 2, `0001`) is shifted left two slots to create a binary 4 (`0100`). The last two operations perform a binary OR to combine bits (`0001|0011 = 0011`) and a binary AND to select common bits (`0001&0011 = 0001`). Such bit-masking operations allow us to encode and extract multiple flags and other values within a single integer.

This is one area where the binary and hexadecimal number support in Python become especially useful—they allow us to code and inspect numbers by bit-strings:

```

>>> X = 0b0001                  # Binary literals
>>> X << 2                   # Shift left
4
>>> bin(X << 2)             # Binary digits string
'0b100'

>>> bin(X | 0b0011)           # Bitwise OR: either
'0b11'
>>> bin(X & 0b11)             # Bitwise AND: both
'0b1'

```

This is also true for values that begin life as hex literals, or undergo base conversions:

```

>>> X = 0xFF                    # Hex literals
>>> bin(X)
'0b11111111'
>>> X ^ 0b10101010            # Bitwise XOR: either but not
85
>>> bin(X ^ 0b10101010)
'0b1010101'

>>> int('01010101', 2)        # Digits=>number: string to int
85

```

```
>>> hex(85)                      # Number=>digits: Hex digit str
'0x55'
```

Also in this department, Python integers come with a `bit_length` method, which allows you to query the number of bits required to represent the number's value in binary. Sharp-eyed readers might point out that you can often achieve the same effect by subtracting 2 from the length of the `bin` string using the `len` built-in function we first used in [Chapter 4](#) (to account for the leading “0b”), though its temporary string result may make it less efficient:

```
>>> X = 99
>>> bin(X), X.bit_length(), len(bin(X)) - 2
('0b1100011', 7, 7)
>>> bin(256), (256).bit_length(), len(bin(256)) - 2
('0b100000000', 9, 9)
```

We won't go into much more detail on such “bit twiddling” here. It's supported if you need it, but bitwise operations are often not as important in a high-level language such as Python as they are in a low-level language such as C. As a rule of thumb, if you find yourself wanting to flip bits in Python, you should think about which language you're really coding. As you'll see in upcoming chapters, Python's lists, dictionaries, and the like provide richer ways to encode information than bit strings, especially when your data's audience includes readers of the human variety.

## Underscore Separators in Numbers

If you're finding it hard to read the longer digit strings in this chapter, there's some good news: as of 3.6, numeric literals in Python can be coded with embedded underscores (“\_”) to group digits for easier viewing. These underscores don't modify number values; Python simply discards them after reading your code. They do, however, work on all the numbers and bases we've met, including complex-number parts and floating-point decimal digits, and can enhance the readability of numeric literals in your scripts. Here's how they look—with before on the left and after on the right:

```
>>> 999999999999 == 9_999_999_999_999
True
>>> 0xFFFFFFFF == 0xFF_FF_FF_FF
```

```
True
>>> 0o777777777777 == 0o777_777_777_777
True
>>> 0b1111111111111111 == 0b1111_1111_1111_1111
True
>>> 3.141592653589793 == 3.141_592_653_589_793
True
>>> 123456789.123456789 == 123_456_789.123_456_789
True
```

While this is a useful feature, you should keep in mind that it's just skin deep. For example, numbers lose their underscores once read. You can add back comma and underscore separators with the *string-formatting* method and f-string we'll explore in [Chapter 7](#), but this is mostly just for display, and the originals are lost:

```
>>> x = 9_999_998                      # Your number with "_"s
>>> x                                # But dropped when read:
9999998
>>> x + 1                            # Ditto for derived comp
9999999

>>> f'{x:,} and {x:_}'      # Formatting adds separa
'9,999,998 and 9_999_998'
```

Moreover, Python doesn't do any sort of sanity checks on underscores, except for disallowing leading, trailing, and multiple-appearance uses. The underscores are really just digit "spacers" that can be used—and misused—arbitrarily:

```
>>> 99_9                          # No position-error chec
999
>>> 1_23_456_7890                  # Hmm...
1234567890

>>> _9
NameError: name '_9' is not defined. Did you mean: '_'?
>>> 9_
SyntaxError: invalid decimal literal
>>> 9_9_9
SyntaxError: invalid decimal literal
```

```
>>> 9_9_9                      # Syntax oddities check
999
>>> hex(0xf_ff_fff_f_f)        # And Python won't retain
'0xffffffff'
```

Also bear in mind that *commas* cannot be used in numeric literals you code—despite their similarity, underscores are essentially ignored as documentation, but commas are taken to be *tuple* item separators if erroneously used:

```
>>> 12_345_678                  # Underscores are ignored
12345678
>>> 12,345,678                 # But commas mean a tuple
(12, 345, 678)
```

Underscores can enhance readability to be sure, but they are largely cosmetic and apply only to large numeric literals in your code—because typical Python programs *compute* most numbers rather than *hardcoding* them, this seems likely to be uncommon in practice. A more promising use case is *input*—string-to-number converters allow underscores too:

```
>>> int('1_234_567')           # Works in text read from file
1234567
>>> eval('1_234_567')          # But does raw-data reading
1234567
>>> float('1_2_34.567_8_90')
1234.56789
```

But it's difficult to justify underscores on data alone, given that scripts could simply strip underscores themselves. Like all such tools, use when it makes sense (and don't be shocked if this crops up in unfair interview questions!).

## Other Built-in Numeric Tools

In addition to its core object types, Python also provides both built-in *functions* and standard-library *modules* for numeric processing. The `pow` and `abs` built-in functions, for instance, compute powers and absolute values, respectively. Here's a brief roundup of common tools in the built-in `math` module (which contains most of the tools in the C language's math library), along with a few numeric built-in functions:

```
>>> import math  
>>> math.pi, math.e # Con  
(3.141592653589793, 2.718281828459045)  
  
>>> math.sin(2 * math.pi / 180) # Sin  
0.03489949670250097  
  
>>> math.sqrt(144), math.sqrt(2) # Squ  
(12.0, 1.4142135623730951)  
  
>>> pow(2, 4), 2 ** 4, 2.0 ** 4.0 # Expr  
(16, 16, 16.0)  
  
>>> abs(-62.0), sum((1, 2, 3, 4)) # Abs  
(62.0, 10)  
  
>>> min(3, 1, 2, 4), max(3, 1, 2, 4) # Mir  
(1, 4)
```

The `sum` function shown here works on a sequence (really, *iterable*) of numbers, and `min` and `max` accept either a collection or individual arguments. There are also multiple ways to drop the decimal digits of floating-point numbers, both for calculations and displays; some of these are richer than previously shown:

```
>>> math.floor(2.567), math.floor(-2.567) #  
(2, -3)  
  
>>> math.trunc(2.567), math.trunc(-2.567) #  
(2, -2)  
  
>>> int(2.567), int(-2.567) #  
(2, -2)  
  
>>> round(2.567), round(2.567, 2), round(2567, -3) #  
(3, 2.57, 3000)  
  
>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567) # F  
('2.6', '2.57')
```

As shown earlier, the last of these produces strings that we would usually print and supports a variety of formatting options. String formatting is still subtly different, though: `round` rounds and drops decimal digits but still produces a number in memory, whereas string formatting produces a string, not a number:

```
>>> (1 / 3.0), round(1 / 3.0, 2), f'{(1 / 3.0):.2f}'  
(0.3333333333333333, 0.33, '0.33')
```

Interestingly, there are three ways to compute *square roots* in Python: using a module function, an expression, or a built-in function (if you’re interested in performance, we will revisit these in an exercise and its solution at the end of [Part IV](#), to see which runs quicker):

```
>>> import math  
>>> math.sqrt(144)                      # Module  
12.0  
>>> 144 ** .5                          # Expression  
12.0  
>>> pow(144, .5)                      # Built-in  
12.0
```

Notice that standard-library modules such as `math` must be imported, but built-in functions such as `abs` and `round` are always available without imports. This is because modules are external components, but built-in functions live in an implied namespace that Python automatically searches to find names used in your program. This namespace simply corresponds to the standard-library module called `builtins`, and there is much more about name resolution in the function and module parts of this book; for now, when you hear “module,” think “import.”

The standard library’s `statistics` and `random` modules must be imported as well. Both modules provide an array of tools; `statistics` supports operations commonly found on calculators, and `random` enables tasks such as picking a random number between 0 and 1 and selecting a random integer between two numbers:

```
>>> import statistics  
>>> statistics.mean([1, 2, 4, 5, 7])      # Average,
```

```
3.8
>>> statistics.median([1, 2, 4, 5, 7])           # And a whole lot more
4

>>> import random
>>> random.random()
0.5566014960423105
>>> random.random()                           # Random floats, integers
0.051308506597373515

>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
9
```

The `random` module can also *choose* an item at random from a sequence, and *shuffle* a list of items randomly:

```
< ━━━━━━ >
>>> random.choice(['Pizza', 'Tacos', 'Tikka', 'Lasagna']
'Lasagna'
>>> random.choice(['Pizza', 'Tacos', 'Tikka', 'Lasagna']
'Lasagna'

>>> suits = ['hearts', 'clubs', 'diamonds', 'spades']
>>> random.shuffle(suits)
>>> suits
['spades', 'hearts', 'diamonds', 'clubs']
>>> random.shuffle(suits)
>>> suits
['clubs', 'diamonds', 'hearts', 'spades']
```

Though we'd need additional code to make this more tangible here, the `random` module can be useful for shuffling cards in games, picking images at random in a slideshow GUI, performing statistical simulations, and much more. We'll deploy it again later in this book (e.g., in [Chapter 20](#)'s permutations case study), but for more details, consult Python's library manual.

# Other Numeric Objects

So far in this chapter, we've been using Python's core numeric types—integer, floating point, and complex. These will suffice for most of the number crunching that many programmers will ever need to do. Python comes with a handful of more exotic numeric types, though, that merit a brief look here.

## Decimal Objects

First up, is Python's special-purpose numeric object known formally as `Decimal` (and informally as `decimal`). Syntactically, decimals are created by calling a function within an imported standard-library module, rather than running a literal expression. Functionally, decimals are like floating-point numbers, but they have a fixed and configurable number of decimal digits. Hence, decimals are *fixed-precision* floating-point values.

For example, with decimals, we can have a floating-point value that always retains just two decimal digits. Furthermore, we can specify how to round or truncate the extra decimal digits beyond the object's cutoff. Although it generally incurs a performance penalty compared to normal floating point, `decimal` is well suited to representing fixed-precision quantities like sums of money and can achieve better numeric accuracy in some contexts.

### Decimal basics

As we learned when we explored comparisons, floating-point math is less than exact because of the limited space used to store values. For instance, the following should yield zero, but it does not. The result is close to zero, but there are not enough bits to be precise here:

```
>>> 0.1 + 0.1 + 0.1 - 0.3 # Almost zero
5.551115123125783e-17
```

Using `print` for the user-friendly display format doesn't help here, because the hardware related to floating-point math is inherently limited in terms of accuracy (a.k.a. *precision*). With decimals, however, the result can be dead-on:

```
>>> from decimal import Decimal  
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -  
Decimal('0.0')
```

As shown here, we can make decimal objects by calling the `Decimal` constructor function in the `decimal` module and passing in strings that have the desired number of decimal digits for the resulting object (using the `str` function to convert floating-point values to strings if needed). When decimals of different precision are mixed in expressions, Python converts up to the largest number of decimal digits automatically:

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.1000'  
Decimal('0.0000')
```

It's also possible to create a decimal object from a floating-point object, with either a call to `Decimal.from_float` or by passing floating-point numbers directly:

```
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(  
Decimal('2.775557561565156540423631668E-17')
```

The conversion is exact but can yield a large default number of digits, unless they are fixed per the next section.

## Setting decimal precision

Other tools in the `decimal` module can be used to set the precision of all decimal numbers, arrange error handling, and more. For instance, a context object in this module allows for specifying precision (number of decimal digits) and rounding modes (down, ceiling, etc.). The precision is applied globally for all decimals created by the caller:

```
>>> import decimal  
>>> decimal.Decimal(1) / decimal.Decimal(7)  
Decimal('0.1428571428571428571428571429')  
  
>>> decimal.getcontext().prec = 4
```

```
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')

>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)
Decimal('1.110E-17')
```

Technically, significance is determined by digits input, and precision is applied on math operations. Although more subtle than we can explore in this brief overview, this property can make decimals useful as the basis for some monetary applications and may sometimes serve as an alternative to manual rounding and string formatting.

Because use of the decimal type is relatively rare in practice, though, this book will defer to Python’s interactive `help` function and standard-library manuals for more details. And because decimals address some of the same floating-point accuracy issues as the fraction type, let’s move on to the next section to see how the two compare.

## Fraction Objects

Python’s standard-library `fractions` module implements a *rational number* object. It essentially keeps both a numerator and a denominator explicitly, so as to avoid some of the inaccuracies and limitations of floating-point math. Like decimals, fractions do not map as closely to computer hardware as floating-point numbers. This means their performance may not be as good, but it also allows them to provide extra utility in a standard tool where useful.

### Fraction basics

`Fraction` is a functional cousin to the `Decimal` fixed-precision object of the prior section, as both can be used to address the floating-point object’s numerical inaccuracies. It’s also used in similar ways—like `Decimal`, `Fraction` resides in a module; import its constructor and pass in a numerator and a denominator to make one (among other schemes). The following interaction shows how:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)                      # Numerator,
>>> y = Fraction(4, 6)                      # Simplified
>>> x
```

```
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

Once created, `Fraction`s can be used in mathematical expressions as usual:

```
>>> x + y
Fraction(1, 1)
>>> x - y
Fraction(-1, 3) # Results are exact
>>> x * y
Fraction(2, 9)
```

`Fraction` objects can also be created from floating-point number strings, much like decimals:

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)

>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

## Numeric accuracy in fractions and decimals

Fraction math is different from floating-point-type math, which is constrained by the underlying limitations of floating-point hardware. To compare, here are the same operations run with floating-point objects, and notes on their limited accuracy—they may display fewer digits in recent Pythons than they used to, but they still aren’t exact values in memory:

```
>>> a = 1 / 3 # Only as accurate
>>> b = 4 / 6 # Can lose precision
>>> a
0.3333333333333333
>>> b
0.6666666666666666
```

```
>>> a + b  
1.0  
>>> a - b  
-0.3333333333333333  
>>> a * b  
0.2222222222222222
```

This floating-point limitation is especially apparent for values that cannot be represented accurately given their limited number of bits in memory. Both `Fraction` and `Decimal` provide ways to get exact results, albeit at the cost of some lost speed and added code verbosity. For instance, in the following example (repeated from the prior section), floating-point numbers do not accurately give the zero answer expected, but both of the other types do:

```
>>> 0.1 + 0.1 + 0.1 - 0.3          # This should be zero  
5.551115123125783e-17  
  
>>> from fractions import Fraction  
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10)  
Fraction(0, 1)  
  
>>> from decimal import Decimal  
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -  
Decimal('0.0')
```

Moreover, fractions and decimals both allow more intuitive and accurate results than floating points sometimes can, in different ways—by using rational representation and by limiting precision:

```
>>> 1 / 3          # Normal floating-point  
0.3333333333333333  
  
>>> Fraction(1, 3)          # Numeric accuracy,  
Fraction(1, 3)  
  
>>> import decimal  
>>> decimal.getcontext().prec = 2  
>>> Decimal(1) / Decimal(3)  
Decimal('0.33')
```

In fact, fractions both retain accuracy and automatically simplify results.

Continuing the preceding interaction:

```
>>> (1 / 3) + (6 / 12)
0.8333333333333333

>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)

>>> decimal.Decimal(1 / 3) + decimal.Decimal(6 / 12)
Decimal('0.83')
```

To support conversions, floating-point objects have an `as_integer_ratio` method noted earlier that yields numerator and denominator; fractions have a `from_float` method; and `float` accepts a `Fraction` as an argument. Because `Fraction` is also a lesser-used utility, though, we’re going to stop short here too; for more details on `Fraction`, experiment further on your own and consult Python’s documentation.

## Set Objects

In addition to all the numeric objects we’ve explored, Python has built-in support for *sets*—an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory. Sets straddle the fence between collections and math but lean far enough on the latter side to warrant coverage in this chapter.

By definition, an item appears only once in a set, no matter how many times it is added. Accordingly, sets have a variety of applications, especially in numeric and database-focused work. On the other hand, because sets are collections of other objects, they share some behavior with objects such as lists and dictionaries previewed in [Chapter 4](#). For example, sets are iterable, can grow and shrink on demand, and may contain a variety of object types.

Still, because sets are unordered and do not map keys to values, they are neither sequence nor mapping types; they are a type category unto themselves. Moreover, because sets also tend to be used much less often than pervasive objects like lists and dictionaries, a brief look should suffice for most readers; let’s get started here with the usual REPL tour.

## Sets in action

First off, there are two ways to make sets—by call and literal. The *literal* uses the same “{}” braces as dictionaries but simply enumerates items (there is no key) and allows you to initialize the set with individual objects. The *call* to `set` accepts an existing sequence (or other iterable) of items to add to the new set and is required to make an empty set ( `{}` is reserved for an empty dictionary). When sets are printed, they prefer the literal form, except when empty:

```
>>> x = set('abcde')                      # Make a set by
>>> y = {99, 'b', 'y', 'd', 1.2}          # Make a set by

>>> x
{'d', 'c', 'e', 'a', 'b'}                  # Order is scr
>>> y
{1.2, 99, 'y', 'd', 'b'}
```

Notice that sets don’t maintain insertion order, unlike the keys in a dictionary (see [Chapter 4](#)’s introduction). This is by definition—sets are just groups of items—but the lack of positional ordering means that sequence operations won’t work on sets. Per the following, empty sets require a call, a `*` in a literal unpacks items, both call and `*` accept any iterable, and sets always filter out duplicate entries; again, this is just how sets work in Python and elsewhere:

```
>>> z = set()                            # Make empty se
>>> z
set()

>>> z = set([1.2, 'a', 3, 1.2, 'a'])    # Any sequence
>>> z
{'a', 1.2, 3}

>>> {1, *'abc', *[1, 2, 3]}            # Literal star
{1, 2, 3, 'c', 'b', 'a'}
```

Once you have sets, expression operators invoke set operations. Here are the most common in action; to run any of these on plain sequences like strings and lists, you must first create a set of their items:

```

>>> x = set('abcd')
>>> y = set('bdxy')

>>> x - y                                # Difference: in x,
{'a', 'c'}

>>> x | y                                # Union: in either
{'y', 'd', 'x', 'c', 'a', 'b'}

>>> x & y                                # Intersection: in
{'d', 'b'}

>>> x ^ y                                # Symmetric difference
{'y', 'x', 'c', 'a'}

>>> x < y, x > y                         # Superset, subset
(False, False)

```

An exception: the `in` set membership test expression is also defined to work on all other collection types, where it also performs membership (or a search, if you prefer to think in procedural terms). Hence, we do not need to convert things like strings and lists to sets to run this test:

```

>>> 'd' in x                                # Membership
True

>>> 'd' in 'code', 2 in [1, 2, 3]          # But
(True, True)

```

In addition to expressions, the set object provides *methods* that correspond to these operations and more, and that support set changes. For instance, the set `add` method inserts one item, `update` is an in-place union, and `remove` deletes an item by value (per the prior chapter, run a `dir` call on any set instance or the `set` type name to see all the available methods). Assuming `x` and `y` are still as they were in the prior interaction:

```

>>> z = x.intersection(y)                  # Same
>>> z
{'d', 'b'}

>>> z.add('HACK')                        # Insert

```

```
>>> z
{'HACK', 'd', 'b'}
>>> z.update(set(['X', 'Y']))                                     # Mer
>>> z
{'X', 'HACK', 'd', 'b', 'Y'}
>>> z.remove('b')                                                 # Del
>>> z
{'X', 'HACK', 'd', 'Y'}
```

Sets also are *iterable* (i.e., they support the iteration protocol introduced in the prior chapter) and hence can also be used in operations such as `len`, `for` loops, and list comprehensions. Because they are unordered, though, they don't support sequence operations like indexing, slicing, or concatenation:

```
<----->
>>> for item in set('abc'):
    print(item * 3)                                              # See

aaa
ccc
bbb
>>> {'a', 'b', 'c'} + {'d'}
TypeError: unsupported operand type(s) for +: 'set' and
```

Finally, although the set expressions shown earlier generally require two sets, their method-based counterparts can often work with *any iterable* as well—and may run faster because of it (though speed guesses are perilous in Python):

```
>>> S = set([1, 2, 3])
>>> S | set([3, 4])                                         # Expressions require
{1, 2, 3, 4}
>>> S | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and
```

```
>>> S.union([3, 4])                                         # But their methods
{1, 2, 3, 4}
>>> S.intersection((1, 3, 5))
{1, 3}
>>> S.issubset(range(-5, 5))                                # Subset of range -5
True
>>> S = set([1, 2, 3])
```

```
>>> S.intersection_update((1, 2, 5))
>>> S
{1, 2}
>>>
>>> S |= {1, 2, 4}
>>> S
{1, 2, 4}
```

For more details on set operations, see Python’s library manual. Among topics skipped here, sets also support in-place changes with additional methods, as well as assignment operators we’ll study later (e.g., `&=` and `|=`). Although sets can be coded manually in Python with other types like lists and dictionaries (and often were in the past), Python’s built-in sets use efficient algorithms and implementation techniques to provide quick and standard operations.

## Immutable constraints and frozen sets

Sets are powerful and flexible objects, but they do have one constraint that you should keep in mind—largely because of their implementation, sets can contain only *immutable* object types (Python refers to this as “hashable,” which is close enough to “immutable” to use the latter here). Hence, lists and dictionaries cannot be embedded in sets, but tuples of other immutables can if you need to store compound values. Tuples compare by full values when used in sets:

```
>>> S = {1.23}

>>> S.add([1, 2, 3])                      # Only immutable
TypeError: unhashable type: 'list'
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'

>>> S.add((1, 2, 3))                     # No List or dict
{1.23, (1, 2, 3)}

>>> S | {(4, 5, 6), (1, 2, 3)}        # Union: same as
{1.23, (4, 5, 6), (1, 2, 3)}

>>> (1, 2, 3) in S                      # Membership: by
True
```

```
>>> (1, 4, 3) in S
```

```
False
```

Tuples in a set, for instance, might be used to represent dates, records, IP addresses, and so on (more on tuples later in this part of the book). Sets may also contain modules, type objects, and more. Sets themselves are mutable too, and so cannot be nested in other sets directly; if you need to store a set inside another set, the `frozenset` built-in call works just like `set` but creates an immutable set that cannot change and thus can be embedded in other sets:

```
>>> S.add(frozenset('app'))  
>>> S  
{1.23, (1, 2, 3), frozenset({'a', 'p'})}
```

## Set comprehensions

In addition to literals and calls, sets can also be made by running comprehension expressions, previewed briefly in [Chapter 4](#). Comprehensions also work for lists, dictionaries, and generators, and behave largely the same in all. For sets, comprehensions are coded in curly braces. When run, they perform a loop that collects the result of an expression on each iteration; a loop variable gives access to the current iteration value for use in the collection expression. The result is a new set with all the normal set behavior.

For example:

```
>>> {x ** 2 for x in [1, 2, 3, 4]} # Make a new  
{16, 1, 4, 9}
```

In this expression, the loop is coded on the right, and the collection expression is coded on the left (`x ** 2`). As for list comprehensions, we get back pretty much what this expression says: “Give me a new set containing X squared, for every X in a list.” Comprehensions can also iterate across other kinds of objects, such as strings; the first of the following examples also illustrates the comprehension-based way to make a set from an existing iterable:

```
>>> {x for x in 'py3X'} # Same as: s  
{'p', 'X', '3', 'y'}
```

```
>>> {c * 4 for c in 'py3X'}                      # Set of col
{'yyyy', '3333', 'XXXX', 'pppp'}
>>> {c * 4 for c in 'py3X' + 'py2X'}            # Expressior
{'yyyy', '3333', 'XXXX', '2222', 'pppp'}

>>> S = {c * 4 for c in 'py3X'}                  # ALL set of
>>> S | {'zzzz', 'XXXX'}
{'yyyy', '3333', 'XXXX', 'pppp', 'zzzz'}
>>> S & {'zzzz', 'XXXX'}
{'XXXX'}
```

Because the rest of the comprehensions story relies upon underlying concepts we're not yet prepared to tackle, we'll postpone further details until later in this book. In [Chapter 8](#), you'll meet first cousins, the list and dictionary comprehension, and you'll learn much more about all comprehensions—set, list, dictionary, and generator—later on, especially in Chapters [14](#) and [20](#). As you'll find, all comprehensions support additional syntax not shown here, including nested loops and `if` tests, which can be challenging before you've had a chance to study larger statements.

## Why sets?

Set operations have a variety of common uses, some more practical than mathematical. For example, because items are stored only once in a set, sets can be used to *filter duplicates* out of other collections, albeit at the cost of original ordering because sets are unordered in general. Simply convert the collection to a set, and then convert it back:

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                         #
>>> L
[1, 2, 3, 4, 5]

>>> list(set(['yy', 'cc', 'aa', 'xx', 'dd', 'aa']))   #
['xx', 'cc', 'yy', 'aa', 'dd']
```

Sets can be used to *isolate differences* in lists, strings, and other iterable objects too—simply convert to sets and take the difference—though again the

unordered nature of sets means that the results may not match that of the originals:

```
>>> set([1, 3, 5, 7]) - set([1, 2, 4, 5, 6])      #
{3, 7}
>>> set('abcdefg') - set('abdghij')      #
{'c', 'e', 'f'}
>>> set('code') - set(['t', 'o', 'e'])      #
{'c', 'd'}
```

You can also use sets to perform *order-neutral equality* tests by converting to a set before the test, because order doesn't matter in a set. More formally, two sets are *equal* if and only if every element of each set is contained in the other—that is, each is a subset of the other, regardless of order. For instance, you might use this to compare the outputs of programs that should work the same but may generate results in different order. Sorting with Python's `sorted` built-in before testing has the same effect for equality; sets don't rely on an expensive sort, but also don't order items:

```
>>> L1, L2 = [1, 3, 5, 2, 4], [2, 5, 3, 4, 1]      #
>>> L1 == L2
False
>>> set(L1) == set(L2)      #
True
>>> sorted(L1) == sorted(L2)      #
True
>>> 'code' == 'edoc', set('code') == set('edoc'), sorted('code') == sorted('edoc')
(False, True, True)
```

Sets can also be used to keep track of where you've already been when traversing a graph or other *cyclic* structure. For example, the transitive module reloader and inheritance-tree lister examples we'll code in Chapters [25](#) and [31](#), respectively, must keep track of items visited to avoid loops, as [Chapter 19](#) discusses in the abstract. Using a list in this context is inefficient because searches require linear scans. Although recording states visited as keys in a dictionary is efficient, sets offer an alternative that's essentially equivalent and may be more intuitive.

Finally, sets are also convenient when you’re dealing with large data collections like database query results—the intersection of two sets contains objects common to both categories, and the union contains all items in either set. To illustrate, here’s a more tangible example of set operations at work, applied to people in a hypothetical company (like all examples in this book, any resemblance to the real world is purely coincidental!):

```
>>> engineers = {'pat', 'ann', 'bob', 'sue'}
>>> managers = {'sue', 'tom'}
```

```
>>> 'pat' in engineers                      # Is pat an engineer?
True
```

```
>>> engineers & managers                   # Who is both?
{'sue'}
```

```
>>> engineers | managers                  # All people involved?
{'ann', 'sue', 'pat', 'tom', 'bob'}
```

```
>>> engineers - managers                 # Engineers without managers?
{'pat', 'ann', 'bob'}
```

```
>>> managers - engineers                # Managers who aren't engineers?
{'tom'}
```

```
>>> engineers > managers              # Are all managers engineers?
False
```

```
>>> {'sue', 'bob'} < engineers          # Are both engineers?
True
```

```
>>> (managers | engineers) > managers    # All people involved?
True
```

```
>>> managers ^ engineers                # Who is in one or the other?
{'ann', 'pat', 'tom', 'bob'}
```



You can find more details on set operations in the Python library manual and some mathematical and database texts. Also stay tuned for [Chapter 8](#)’s revival of set operations we’ve seen here, in the context of dictionary view objects. Here, we have time for just one last numeric object type.

# Boolean Objects

Though somewhat gray, the Python Boolean type, `bool`, is arguably numeric in nature because its two values, `True` and `False`, are just customized versions of the integers 1 and 0 that print themselves differently. Python treats 1 and 0 as true and false like many programming languages, but its `True` and `False` makes Boolean roles more explicit. Although that's all some programmers may need to know, let's briefly reveal this type's forgery.

To represent truth values, Python has an explicit Boolean type called `bool`, from which the objects preassigned to built-in names `True` and `False` are made. That is, `True` and `False` are instances of `bool`, which is in turn just a subclass (in the object-oriented sense) of the built-in integer type `int`. `True` and `False` behave exactly like the integers 1 and 0, except that they have customized printing logic—they print themselves as the words `True` and `False`, instead of the digits `1` and `0`. `bool` accomplishes this by redefining `str` and `repr` string formats (introduced earlier in this chapter) for its two objects, and all logical tests yield `True` or `False` for their results.

Because of this customization, Boolean expressions typed at the interactive prompt print results as the words `True` and `False` instead of the less obvious `1` and `0`. In addition, Booleans make truth values more apparent in your code. For instance, an infinite loop can be coded as `while True:` instead of the less intuitive `while 1:`, and flags can be initialized more clearly with `flag = False`. We'll discuss these statements further in [Part III](#).

Again, though, for most practical purposes, you can treat `True` and `False` as though they are predefined variables set to integers `1` and `0`. This implementation can lead to curious results, though; because `True` is just the integer `1` with a custom display format, `True + 4` yields integer `5` in Python:

```
>>> type(True)                  # True is a bool
<class 'bool'>
>>> isinstance(True, int)      # As well as an int
True
>>> True == 1                  # Same value
True
```

```
>>> True is 1          # But a different object:  
False  
>>> True or False      # Same as: 1 or 0  
True  
>>> True + 4           # (Hmmm)  
5
```

Since you probably won’t come across an expression like the last of these in real Python code, you can safely ignore any of its deeper metaphysical implications. We’ll revisit Booleans in [Chapter 9](#) to define Python’s notion of truth, and again in [Chapter 12](#) to see how Boolean operators like `and` and `or` work.

## Numeric Extensions

Finally, although Python’s core numeric objects offer plenty of power for most applications, a large catalog of third-party open source extensions is available to address more focused numeric needs.

We surveyed tools in this domain in [Chapter 1](#)’s section “What Can I Do with Python?” In short, there is a now-common stack of tools for advanced numeric coding in Python today, including *NumPy*, *SciPy*, *pandas*, *matplotlib*, *Jupyter*, and more, and additional tools address subdomains like statistics, astronomy, and AI.

This toolkit is used by research organizations, financial entities, and aerospace groups around the world, and performs the sort of tasks formerly coded in languages like C++ or Fortran. Many who work in this field liken the combination of Python plus numeric extensions to a free, flexible, and powerful alternative to systems like MATLAB.

Though a popular and exciting domain, Python numeric programming is just one way to use the language (Python web development, for example, is similarly sized) and is easily rich enough to fill entire books by itself. Hence, this book doesn’t cover numeric extensions and focuses instead on teaching you the Python language that’s used in every domain. Once you’ve learned Python itself, you’ll find copious resources for add-ons both on the web and at book outlets near you when you’re ready to level up.

# Chapter Summary

This chapter has toured Python’s numeric object types and the operations we can apply to them. Along the way, we met the trusty integer and floating-point objects, as well as some more exotic and less commonly used types such as complex numbers, decimals, fractions, and sets. We also explored Python’s expression syntax, type conversions, bitwise operations, and various literal forms for coding numbers in scripts.

Later in this part of the book, we’ll continue our in-depth object tour by filling in details about the next object type—the string. In the next chapter, however, we’ll take some time to explore the mechanics of variable assignment in more detail than we have here. This turns out to be perhaps the most fundamental idea in Python, so make sure you check out the next chapter before moving on. First, though, it’s time to take the usual chapter quiz.

## Test Your Knowledge: Quiz

1. What is the value of the expression `2 * (3 + 4)` in Python, and why?
2. What is the value of the expression `2 * 3 + 4` in Python, and why?
3. What is the value of the expression `2 + 3 * 4` in Python, and why?
4. What tools can you use to find a number’s square root, as well as its square?
5. What is the type of the result of the expression `1 + 2.0 + 3`, and why?
6. How can you truncate and round a floating-point number?
7. How can you convert an integer to a floating-point number?
8. How would you display an integer in octal, hexadecimal, or binary notation?
9. How might you convert an octal, hexadecimal, or binary string to a plain integer?

## Test Your Knowledge: Answers

1. The value will be `14`, the result of  $2 * 7$ , because the parentheses force the addition to happen before the multiplication.
2. The value will be `10`, the result of  $6 + 4$ . Python’s operator precedence rules are applied in the absence of parentheses, and multiplication has

higher precedence than (i.e., happens before) addition, per [Table 5-2](#).

3. This expression yields `14`, the result of `2 + 12`, for the same precedence reasons as in the prior question.
4. Functions for obtaining the square root, as well as `pi`, tangents, and more, are available in the imported `math` module. To find a number's square root, import `math` and call `math.sqrt(N)`. To get a number's square, use either the exponent expression `X ** 2` or the built-in function `pow(X, 2)`. Either of these last two can also compute the square root when given a power of `0.5` (e.g., `X ** .5`).
5. The result will be a floating-point number: the integers are converted up to floating point, the most complex type in the expression, and floating-point math is used to evaluate it.
6. The `int(N)` and `math.trunc(N)` functions truncate, and the `round(N, digits)` function rounds. We can also compute the floor with `math.floor(N)` and round for display with string-formatting operations.
7. The `float(I)` function converts an integer to a floating point; mixing an integer with a floating point within an expression will result in a conversion as well. In some sense, Python / true division converts too—it always returns a floating-point result that includes the remainder, even if both operands are integers.
8. The `oct(I)`, `hex(I)`, and `bin(I)` built-in functions return the octal, hexadecimal, and binary string forms for an integer. All three flavors of string formatting (expression, method, and f-string) also provide targets for some such conversions.
9. The `int(S, base)` function can be used to convert from octal, hexadecimal, and binary digit strings to normal integers (pass in `8`, `16`, or `2` for the `base`). The `eval(S)` function can be used for this purpose too, but it's more expensive to run and can have security risks. To some extent, other-base literals like `0xFFFF` and `0b1111` in your code do this work too when read by Python. Note that integers are always stored in binary form in computer memory; these are just display string format conversions.