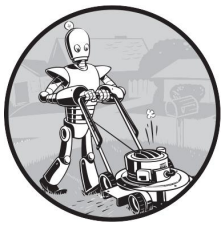
1

PYTHON BASICS



The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that; you just need to learn enough to write some handy little programs.

To accomplish this, however, you'll have to master some programming concepts. Like a wizard in training, you might think these concepts seem tedious, but with some practice, they'll enable you to command your computer like a magic wand and perform incredible feats.

This chapter has a few examples that encourage you to enter code into the *interactive shell*, also called the *read-evaluate-print-loop (REPL)*, which lets you run (or *execute*) Python instructions one at a time and instantly shows you the results. Using the interactive shell is great for learning what basic Python instructions do, so give it a try as you follow along. You'll remember the things you do much better than the things you only read.

Entering Expressions into the Interactive Shell

You can run the interactive shell by launching the Mu editor. This book's introduction provides setup instructions for downloading and installing it. On Windows, open the Start menu, enter **Mu**, and open the Mu app. On macOS, open your Applications folder and double-click **Mu**. Click the **New** button and save an empty file as *blank.py*. When you run this blank file by clicking the **Run** button or pressing F5, it will open the interactive shell, which will open as a new pane at the bottom of the Mu editor's window. You should see a >>> prompt in the interactive shell.

You can also run the interactive shell from the command line Terminal (on macOS and Linux) or Windows Terminal (on Windows, where you could also use the older Command Prompt application). After opening these command line windows, enter **`python`** (on Windows) or **`python3`** (on macOS and Linux). You'll see the same >>> prompt for the interactive shell. If you want to run a program, run `python` or `python3` followed by the name of the program's *.py* file, such as `python blank.py`. Be sure you don't run `python` on macOS's Terminal, as this may launch the older, backward-incompatible Python 2.7 version on certain versions of macOS. You may even see a message saying `WARNING: Python 2.7 is not recommended`. Exit the 2.7 interactive shell and run `python3` instead.

Enter **`2 + 2`** at the prompt to have Python do some simple math. The Mu window should now look like this:

```
>>> 2 + 2
4
>>>
```

In Python, `2 + 2` is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as `2`) and *operators* (such as +), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, `4`. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

The Mu editor has a REPL button that shows an interactive shell with a prompt that looks like `In [1]:`. The popular Jupyter Notebook editor uses this kind of interactive shell. You can use this interactive shell the same way as the normal Python interactive shell with the >>> prompt. REPLs are not unique to Python; many programming languages also offer REPLs so that you can experiment with their code.

ERRORS ARE OKAY!

The best thing about computers is that they carry out the exact instructions you give them. This is also the worst thing about computers. Computers can't use common sense to figure out what you intended to do. Programs will crash if they contain code the computer can't understand, which will cause Python to show an error message. Error messages don't damage your computer, though, so don't be afraid to make mistakes. A *crash* just means the program unexpectedly stopped running.

Get used to seeing error messages, because you'll constantly encounter them (even if you have decades of programming experience). Error messages are often vague and not meant to be immediately understood by beginners. If you want to know more about an error, you can search for the exact error message text online for more information. Note that if you are using the Mu editor, the keyboard shortcut for copying highlighted text in the interactive shell pane to the clipboard is CTRL-SHIFT-C, while the shortcut for copying text in the main file editor pane is the standard CTRL-C. You can also check out the resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition* to see a list of common Python error messages and their meanings.

Programming involves some math operations you might not be familiar with:

- Exponentiation (or *to the power of*) is multiplying a number by itself repeatedly, just like multiplication is adding a number to itself repeatedly. For example, *two to the power of four* (or *two to the fourth power*), written as or 2^4 or `2 ** 4`, is the number two multiplied by itself four times: $2^4 = 2 \times 2 \times 2 \times 2 = 16$.
- Modular arithmetic is similar to the remainder result of division. For example, `14 % 4` evaluates to `2` because 14 divided by 4 is 3 with remainder 2. Even though Python's modulo operator is `%`, modular arithmetic has nothing to do with percentages.
- Integer division is the same as regular division except the result is rounded down. For example, `25 / 8` is `3.125` but `25 // 8` is `3`, and `29 / 10` is `2.9` but `29 // 10` is `2`.

You can use plenty of other operators in Python expressions too. For example, Table 1-1 lists all the math operators in Python.

Table 1-1: Math Operators

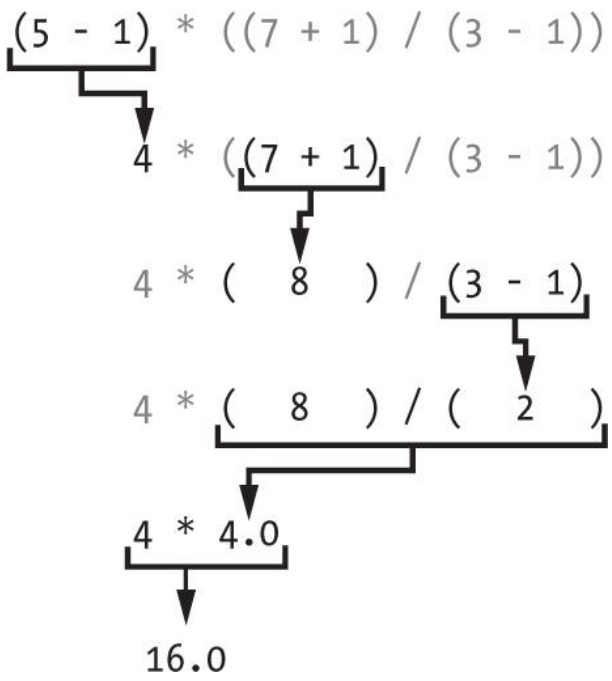
Operator	Operation	Example	Evaluates to ...
`**`	Exponentiation	`2 ** 3`	`8`
`%`	Modulus/remainder	`22 % 8`	`6`
`//`	Integer division	`22 // 8`	`2`
`/`	Division	`22 / 8`	`2.75`
`*`	Multiplication	`3 * 5`	`15`
`-`	Subtraction	`5 - 2`	`3`
`+`	Addition	`2 + 2`	`4`

The *order of operations* (also called *precedence*) of Python math operators is similar to that of mathematics. The `**` operator is evaluated first; the `*`, `/`, `//`, and `%` operators are evaluated next, from left to right; and the + and − operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to. Whitespace in between the operators and values doesn't matter in Python, except for the indentation at the beginning of the line. But the *convention*, or unofficial rule, is to have a single space in between operators and values. Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2    +          2
4
```

```
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it. Python will keep evaluating parts of the expression until it becomes a single value:



Description

These rules for gether to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you enter a bad Python instruction, Python won't be able to understand it and will display a `SyntaxError` error message, as shown here:

```
>>> 5 +
  File "<python-input-0>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<python-input-0>", line 1
```

```
    42 + 5 + * 2
               ^
SyntaxError: invalid syntax
```

You can always test whether an instruction works by entering it into the interactive shell. Don't worry about breaking the computer; the worst that could happen is that Python responds with an error message. Professional software developers get error messages all the time while writing code.

The Integer, Floating-Point, and String Data Types

Remember that expressions are just values combined with operators, and they always evaluate to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most common data types in Python are listed in Table 1-2. The values `-2` and `30`, for example, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers. Numbers with a decimal point, such as `3.14`, are called *floating-point numbers* (or *floats*). Note that even though the value `42` is an integer, the value `42.0` would be a floating-point number. Programmers often use *number* to refer to ints and floats collectively, although *number* itself is not a Python data type.

One subtle detail about Python is that any math performed using an int and a float results in a float, not an int. While `3 + 4` evaluates to the integer `7`, the expression `3 + 4.0` evaluates to the floating-point number `7.0`. Any division between two integers with the `/` division operator results in a float as well. For example, `16 / 4` evaluates to `4.0` and not `4`. Most of the time, this information doesn't matter for your program, but knowing it will explain why your numbers may suddenly gain a decimal point.

Table 1-2: Common Data Types

Data type	Examples
Integer (int)	`-2`, `-1`, `0`, `1`, `2`, `3`, `4`, `5`
Floating-point number (float)	`-1.25`, `-1.0`, `-0.5`, `0.0`, `0.5`, `1.0`, `1.25`
String (str)	`'a'`, `'aa'`, `'aaa'`, `'Hello!'`, `'11 cats'`, `'5'`

Python programs can also have text values called *strings*, or *strs* (pronounced “stirs”). Always surround your string in single-quote (`'`) characters (as in `'Hello'` or `'Goodbye cruel world!'`) so that Python knows where the string begins and ends. You can even have a string with no characters in it, `''`, called a *blank string* or an *empty string*. Strings are explained in greater detail in Chapter 8.

You may see the error message `SyntaxError: unterminated string literal`, as in this example:

```
>>> 'Hello, world!
SyntaxError: unterminated string literal
(detected at line 1)
```

This error means you probably forgot the final single-quote character at the end of the string.

String Concatenation and Replication

The meaning of an operator may change based on the data types of the values next to it. For example, + is the addition operator when it operates on two integers or floating-point values. However, when + is used to combine two string values, it joins the strings as the *string concatenation* operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the + operator on a string and an integer value, Python won’t know how to handle this and will display an error message:

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
    'Alice' + 42
```

```
TypeError: can only concatenate str (not
"int") to str
```

The error message `can only concatenate str (not "int") to str` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string because Python cannot do this automatically. (I'll explain how to convert between data types in "Dissecting the Program" on page 14, where we talk about the `str()`, `int()`, and `float()` functions.)

The `*` operator multiplies two integer or floating-point values. But when the `*` operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original string a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

The `*` operator can only be used with two numeric values (for multiplication), or one string value and one integer value (for string replication). Otherwise, Python will just display an error message, such as the following:

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int
of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
```

```
<module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int
of type 'float'
```

It makes sense that Python wouldn't understand these expressions: you can't multiply two words, and it's hard to replicate an arbitrary string a fractional number of times.

Expressions, data types, and operators may seem abstract to you right now, but as you learn more about these concepts, you'll be able to create increasingly sophisticated programs that do math on data pulled from spreadsheets, websites, the output of other programs, and other places.

Storing Values in Variables

A *variable* is like a box in the computer's memory where you can store a single value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

You'll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, a variable named `spam` will have the integer value `42` stored in it.

You can think of a variable as a labeled box that a value is placed in, but Chapter 6 explains how a name tag attached to the value might be a better metaphor. Both are shown in Figure 1-1.

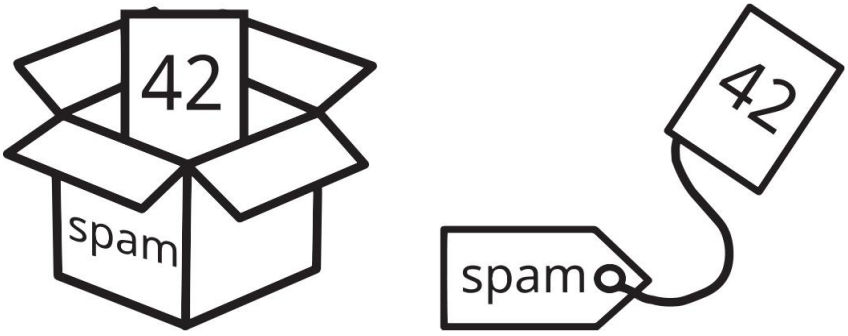


Figure 1-1: The code `spam = 42` is like telling the program, "The variable `spam` now has the integer value 42 in it."

For example, enter the following into the interactive shell:

```
❶ >>> spam = 40
>>> spam
40
>>> eggs = 2
❷ >>> spam + eggs
42
>>> spam + eggs + spam
82
❸ >>> spam = spam + 2
>>> spam
42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to `42` instead of `40` at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Just like the box in Figure 1-1, the `spam` variable in Figure 1-2 stores `'Hello'` until you replace the string with `'Goodbye'`.

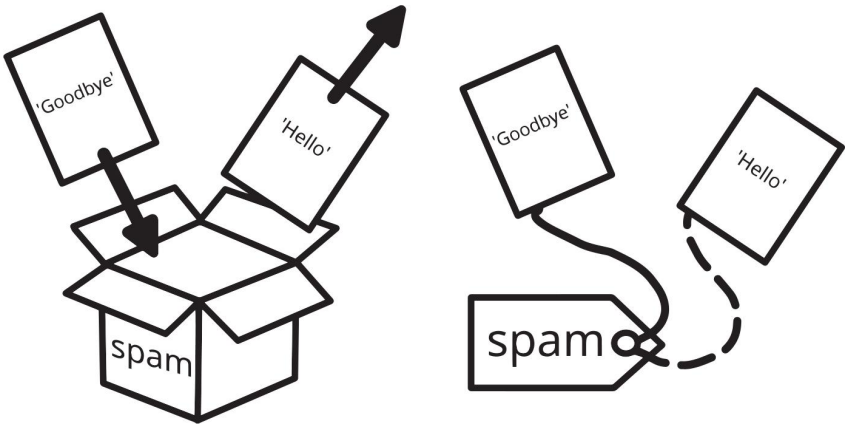


Figure 1-2: When a new value is assigned to a variable, the old one is forgotten.

You can also think of overwriting a variable as reassigning the name tag to a new value.

Variable Names

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You'd never find anything! Most of this book's examples (and Python's documentation) use generic variable names like `spam`, `eggs`, and `bacon`, which come from the Monty Python "Spam" sketch. But in your programs, descriptive names will help make your code more readable.

Though you can name your variables almost anything, Python does have some naming restrictions. Your variable name must obey the following four rules:

- It can't have spaces.
- It can use only letters, numbers, and the underscore (_) character.
- It can't begin with a number.
- It can't be a Python keyword, such as `if`, `for`, `return`, or other keywords you'll learn in this book.

Table 1-3 shows examples of legal variable names.

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
`current_balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`account4`	`4account` (can't begin with a number)
`_42`	`42` (can begin with an underscore but not a number)
`TOTAL_SUM`	

Valid variable names	Invalid variable names
	`TOTAL_$UM` (special characters like `$` are not allowed)
`hello`	`'hello'` (special characters like ' are not allowed)

Variable names are case-sensitive, meaning that `spam`, `SPAM`, `Spam`, and `sPaM` are four different variables. It is a Python convention to start your variables with a lowercase letter: `spam` instead of `Spam`.

CODE STYLE OPINIONS AND PEP 8

Previous editions of this book used *camelCase* instead of underscores to separate words in variable names; that is, variables `lookedLikeThis` instead of `looking_like_this`. The latter form is called *snake_case* because the underscores between words look like little snakes (while the uppercase letters in camelCase look like the humps on a camel). Some experienced programmers may point out that the official Python code style document, PEP 8, says that underscores should be used. I unapologetically prefer camelCase and point to the "A Foolish Consistency Is the Hobgoblin of Little Minds" section in PEP 8 itself as my defense:

> *Consistency with the style guide is important. But most importantly: know when to be inconsistent—sometimes the style guide just doesn't apply. When in doubt, use your best judgment.*

The computer doesn't care which style you use, and PEP 8 is not a stone tablet of irrefutable commandments. It doesn't matter which style you use as long as you use the same style consistently. To prove this, I've rewritten the code in this book to use snake_case because it truly doesn't matter either way.

Your First Program

While the interactive shell is good for running Python instructions one at a time, to write entire Python programs you'll enter the instructions into the file editor. The *file editor* is similar to text editors such as Notepad and TextMate, but it has some features specifically for entering source code. To open a new file in Mu, click the **New** button on the top row.

The tab that appears should contain a cursor awaiting your input, but it's different from the interactive shell, which runs Python instructions as soon as you press ENTER. The file editor lets you enter

many instructions, save the file, and run the program. Here's how you can tell the difference between the two:

- The interactive shell will always be the one with the >>> or `In [1]:` prompt.
- The file editor won't have the >>> or `In [1]:` prompt.

Now it's time to create your first program! When the file editor window opens, enter the following into it:

```
# This program says hello and asks for my
name.

print('Hello, world!')
print('What is your name?')  # Ask for their
name.
my_name = input('>')
print('It is good to meet you, ' + my_name)
print('The length of your name is:')
print(len(my_name))
print('What is your age?')  # Ask for their
age.
my_age = input('>')
print('You will be ' + str(int(my_age) + 1) +
' in a year.')
```

Once you've entered your source code, save it so that you won't have to retype it each time you start Mu. Click **Save**, enter **hello.py** in the File Name field, and then click **Save**.

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit Mu, you won't lose the code. As a shortcut, you can press CTRL-S on Windows and Linux or ⌘-S on macOS to save your file.

Once you've saved, let's run our program. Press the F5 key or click the **Run** button. Enter your name when your program asks for it. The program's output in the interactive shell should look something like this:

```
Hello, world!
What is your name?
```

```
>Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
>4
You will be 5 in a year.
>>>
```

When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.) The Mu editor displays the >>> interactive shell prompt after the program terminated, in case you'd like to enter some further Python code.

You can close the file editor by clicking the **X** on the file's tab, just like closing a browser tab. To reload a saved program, click **Load** from the menu. Do that now, and in the window that appears, choose **hello.py** and click the **Open** button. Your previously saved *hello.py* program should open in the file editor window.

You can view the step-by-step execution of a program using the Python Tutor visualization tool at *http://pythontutor.com*. Click the forward button to move through each step of the program's execution. You'll be able to see how the variables' values and the output change.

Dissecting the Program

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a *comment*:

```
# This program says hello and asks for my
name.
```

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program isn't working. You can remove the # later when you are ready to put the line back in.

Python also ignores the blank line after the comment. You can add as many blank lines to your program as you want. This spacing can make your code easier to read, like paragraphs in a book.

The print() Function

The `print()` function displays the string value inside its parentheses on the screen:

```
print('Hello, world!')
print('What is your name?')  # Ask for their
name.
```

The line `print('Hello, world!')` means "Print out the text in the string `'Hello, world!'`." When Python executes this line, you say that Python is *calling* the `print()` function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument.* Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value's text.

NOTE

You can also use this function to display a blank line on the screen; call `print()` with nothing in between the parentheses.

When you write a function name, the opening and closing parentheses at the end identify it as the name of a function. This is why in this book, you'll see `print()` rather than `print`. It's a standard convention to have no spaces in between the function name and the opening parentheses, even though Python doesn't require this. Chapter 3 describes functions in more detail.

The input() Function

The `input()` function waits for the user to type some text on the keyboard and press ENTER:

```
my_name = input('>')
```

This function call evaluates to a string identical to the user's text, and the rest of the code assigns the `my_name` variable to this string value. The `'>'` string passed to the function causes the > prompt to appear, which serves as an indicator to the user that they are expected to enter something. Your programs don't have to pass a string to the `input()` function; if you call `input()`, the program will wait for the user's text without displaying any prompt.

THE > AND >>> PROMPTS

You can pass any string to `input()` to change the prompt that appears when your program runs. Calling `input('>')` puts an angle bracket > on the screen as the prompt. This is different from the `>>>` prompt that appears in the Python interactive shell. In this book, the `>>>` prompt indicates a response to the Python interactive shell and the > prompt indicates a response to a program running the `input('>')` call. My choice of `'>'` was arbitrary; you can use any prompt you want or no prompt at all.

You can think of the `input()` function call as an expression that evaluates to whatever string the user typed. If the user entered `'Al'`, the assignment statement would effectively be `my_name = 'Al'`.

If you call `input()` and see an error message, like `NameError: name 'Al' is not defined`, the problem is that you're running the code with Python 2 instead of Python 3.

The Greeting Message

The following call to `print()` contains the expression `'It is good to meet you, ' + my_name` between the parentheses:

```
print('It is good to meet you, ' + my_name)
```

Remember that expressions can always evaluate to a single value. If `'Al'` is the value stored in `my_name`, then this expression evaluates to `'It is good to meet you, Al'`. This single string value is then passed to `print()`, which prints it on the screen.

The len() Function

You can pass the `len()` function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string:

```
print('The length of your name is:')
print(len(my_name))
```

Enter the following into the interactive shell to try this:

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

Just like in those examples, `len(my_name)` evaluates to an integer. We say that the `len()` function call *returns* or *outputs* this integer value, and the value is the function call's *return value*. It is then passed to `print()` to be displayed on the screen. The `print()` function allows you to pass it either integer values or string values, but notice the error that shows up when you enter the following into the interactive shell:

```
 >>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: can only concatenate str (not "int") to str
```

The `print()` function isn't causing that error; rather, it's the expression you tried to pass to `print()`. You'll get the same error message if you type the expression into the interactive shell on its own:

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
    'I am ' + 29 + ' years old.'
TypeError: can only concatenate str (not
"int") to str
```

Python gives an error because the + operator can be used only to add two numbers together or to concatenate two strings. You can't add an integer to a string because this is not allowed in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

The str(), int(), and float() Functions

If you want to concatenate an integer such as `29` with a string to pass to `print()`, you'll need to get the value `'29'`, which is the string form of `29`. The `str()` function can be passed an integer value and will return a string value version of the integer, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

Because `str(29)` evaluates to `'29'`, the expression `'I am ' + str(29) + ' years old.'` evaluates to `'I am ' + '29' + ' years old.'`, which in turn evaluates to `'I am 29 years old.'` This is the string value that is passed to the `print()` function.

The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions, and watch what happens:

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

The previous examples call the `str()`, `int()`, and `float()` functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.

The `str()` function is handy when you have an integer or float that you want to concatenate to a string. The `int()` function is also helpful if you have a number as a string value that you want to use in some mathematics. For example, the `input()` function always returns a string, even if the user enters a number. Enter **`spam = input()`** into the interactive shell, then enter **`101`** when it waits for your text:

```
>>> spam = input()
101
>>> spam
'101'
```

The value stored inside `spam` isn't the integer `101` but the string `'101'`. If you want to do math using the value in `spam`, use the `int()` function to get its integer form and then store this as the variable's new value. If `spam` is the string `'101'`, then the expression

`int(spam)` will evaluate to the integer value `101`, and the assignment statement `spam = int(spam)` will be equivalent to `spam = 101`:

```
>>> spam = int(spam)
>>> spam
101
```

Now you should be able to treat the `spam` variable as an integer instead of a string:

```
>>> spam * 10 / 5
202.0
```

Note that if you pass a value to `int()` that it cannot evaluate as an integer, Python will display an error message:

```
>>> int('99.99')
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
    int('99.99')
ValueError: invalid literal for int() with
base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
    int('twelve')
ValueError: invalid literal for int() with
base 10: 'twelve'
```

The `int()` function is also useful if you need to round a floating-point number down:

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

You used the `int()` and `str()` functions in the last three lines of your program to get a value of the appropriate data type for the code:

```
print('What is your age?')  # Ask for their
age.
my_age = input('>')
print('You will be ' + str(int(my_age) + 1) +
' in a year.')
```

The `my_age` variable contains the value returned from `input()`. Because the `input()` function always returns a string (even if the user entered a number), you can use the `int(my_age)` code to return an integer value of the string in `my_age`. This integer value is then added to `1` in the expression `int(my_age) + 1`.

TEXT AND NUMBER EQUIVALENCE

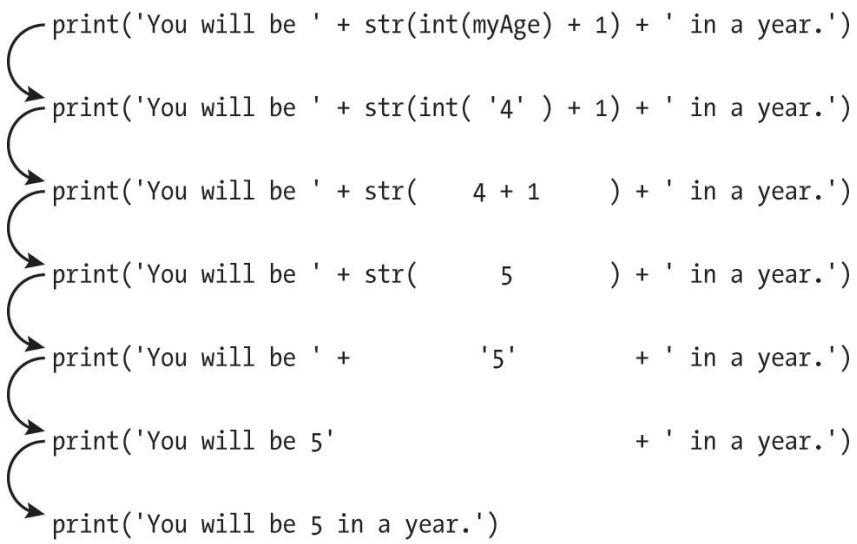
Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point:

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python makes this distinction because strings are text, while integers and floats are numbers.

The result of this addition is passed to the `str()` function: `str(int(my _age) + 1)`. The string value returned is then concatenated with the strings `'You will be '` and `' in a year.'` to evaluate to one large string value. This large string is finally passed to `print()` to be displayed on the screen.

Let's say the user enters the string `'4'` for `my_age`. The evaluation steps would look something like the following:



Description

The string `'4'` is converted to an integer, so you can add `1` to it. The result is `5`. The `str()` function converts the result back to a string, so you can concatenate it with the second string, `'in a year.'`, to create the final message.

The type() Function

Integer, floating-point, and string aren't the only data types in Python. As you continue to learn about programming, you may come across values of other data types. You can always pass these to the `type()` function to determine what type they are. For example, enter the following into the interactive shell:

```
>>> type(42)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('forty two')
<class 'str'>
>>> name = 'Zophie'
>>> type(name)  # The name variable has a
```

```
value of the string type.
<class 'str'>
>>> type(len(name))  # The len() function
returns integer values.
<class 'int'>
```

Not only can you pass any value to `type()`, but (as with any function call) you can also pass it any variable or expression to determine the data type of the value that it evaluates to. The `type()` function itself returns values, but the angle brackets mean they are not syntactically valid Python code; you cannot run code like `spam = <class 'str'>`.

The round() and abs() Functions

Let's learn about two more Python functions that, like the `len()` function, take an argument and return a value. The `round()` function accepts a float value and returns the nearest integer. Enter the following into the interactive shell:

```
>>> round(3.14)
3
>>> round(7.7)
8
>>> round(-2.2)
-2
```

The `round()` function also accepts an optional second argument specifying how many decimal places it should round. Enter the following into the interactive shell:

```
>>> round(3.14, 1)
3.1
>>> round(7.7777, 3)
7.778
```

The behavior for rounding half numbers is a bit odd. The function call `round(3.5)` rounds up to `4`, while `round(2.5)` rounds down

to 2. For halfway numbers that end with .5, the number is rounded to the nearest even integer. This is called *banker's rounding*.

The abs() function returns the absolute value of the number argument. In mathematics, this is defined as the distance from 0, but I find it easier to think of it as the positive form of the number. Enter the following into the interactive shell:

```
>>> abs(25)
25
>>> abs(-25)
25
>>> abs(-3.14)
3.14
>>> abs(0)
0
```

Python comes with several different functions that you'll learn about in this book. This section demonstrates how you can experiment with them in the interactive shell to see how they behave with different inputs. This is a common technique for practicing the new code that you learn.

How Computers Store Data with Binary Numbers

That's enough Python code for now. At this point, you might think that programming seems almost magical. How does the computer know to transform 2 + 2 into 4? The answer is too complicated for this book, but I can explain part of what's going on behind the scenes by discussing what binary numbers (numbers that have only the digits 1 and 0) have to do with computing.

Hacking in movies often involves streams of 1s and 0s flowing across the screen. This looks mysterious and impressive, but what do these 1s and 0s actually mean? The answer is that binary is the simplest number system, and it can be implemented with inexpensive components for computer hardware. Binary, also called the *base-2 number system*, can represent all of the same numbers that our more familiar base-10 *decimal number system* can. Decimal has 10 digits, 0 through 9. Table 1-4 shows the first 27 integers in decimal and binary.

Table 1-4: Equivalent Decimal and Binary Numbers

Decimal	Binary	Decimal	Binary	Decimal	Binary
0	0	9	1001	18	10010
1	1	10	1010	19	10011
2	10	11	1011	20	10100
3	11	12	1100	21	10101
4	100	13	1101	22	10110
5	101	14	1110	23	10111
6	110	15	1111	24	11000
7	111	16	10000	25	11001
8	1000	17	10001	26	11010

Think of these number systems as a mechanical odometer, like in Figure 1-3. When you reach the last digit, they each reset to 0 while incrementing the value of the next digit over. In decimal, the last digit is 9, and in binary, the last digit is 1. That's why the decimal number after 9 is 10 and the decimal number after 999 is 1000. Similarly, the binary number after 1 is 10 and the binary number after 111 is 1000. However, *10* in binary doesn't represent the same quantity as *ten* in decimal; rather, it represents *two*. And *1000* in binary doesn't mean *one thousand* in decimal, but rather *eight*. You can view an interactive binary and decimal odometer at https://inventwithpython.com/odometer.

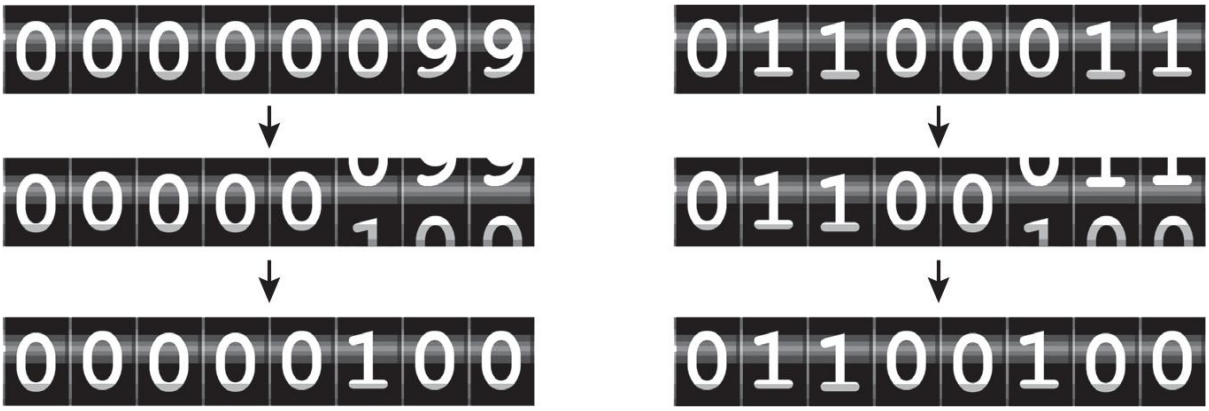


Figure 1-3: A mechanical odometer in decimal (left) and in binary (right)

Representing binary numbers with computer hardware is simpler than representing decimal numbers because there are only two states to represent. For example, Blu-ray discs and DVDs have smooth *lands* and indented *pits* etched on their surface that will or won't reflect the disc player's laser, respectively. Circuits can have electric current flowing through them or no electric current. These various hardware standards all have ways of representing two different states. On the other hand, it'd be expensive to create high-quality electronic components that are

sensitive enough to detect the difference between 10 different voltage levels with reliable accuracy. It's more economical to use simple components, and two binary states are as simple as you can get.

These binary digits are called *bits* for short. A single bit can represent two numbers, and 8 bits (or 1 *byte*) can represent 2^8, or 256, numbers, ranging from 0 to 255 in decimal or 0 to 11111111 in binary. This is similar to how a single decimal digit can represent 10 numbers (0 to 9), while an eight-digit decimal number can represent 10^8 or 100,000,000 numbers (0 to 99,999,999). Files on your computer are measured in how many bytes they take up:

- A kilobyte (KB) is 2^{10} or 1,024 bytes.
- A megabyte (MB) is 2^{20} or 1,048,576 bytes (or 1,024KB).
- A gigabyte (GB) is 2^{30} or 1,073,741,824 bytes (or 1,024MB).
- A terabyte (TB) is 2^{40} or 1,099,511,627,776 bytes (or 1,024GB).

The text of Shakespeare's *Romeo and Juliet* is about 135KB. A high-resolution photo is about 2MB to 5MB. A movie can be anywhere from 1GB to 50GB, depending on picture quality and movie length. However, hard drive and flash memory manufacturers blatantly lie about what these terms mean. For example, by calling a TB 1,000,000,000,000 bytes instead of 1,099,511,627,776 bytes, they can advertise a 9.09TB hard drive as 10TB.

The 1s and 0s of binary can represent not only any integer but also any form of data. Instead of 0 to 255, a byte can represent the numbers –128 to 127 using a system called *two's complement*. Fractional floating-point numbers can be represented in binary using a system called *IEEE-754*.

Text can be stored on computers as binary numbers by assigning each letter, punctuation mark, or symbol a unique number. A system for representing text as numbers is called an *encoding*. The most popular encoding for text is UTF-8. In UTF-8, a capital letter *A* is represented by the decimal number 65 (or 01000001 as an 8-bit binary number), a *?* (question mark) is represented by the number 63, and the numeral character *7* is represented by the number 55. The string `'Hello'` is stored as the numbers 72, 101, 108, 108, and 111. When stored in a computer, "Hello" appears as a stream of bits: 0100100001100101011011000110110001101111.

Wow! Just like in those hacker movies!

Engineers need to invent a way to encode each form of data as numbers. Photos and images can be broken up into a two-dimensional grid of colored squares called *pixels*. Each pixel can use three bytes to represent how much red, green, and blue color it contains. (Chapter 21 covers image data in more detail.) But for a short example, the numbers

255, 0, and 255 could represent a pixel with the maximum amount of red and blue but zero green, resulting in a purple pixel.

Sound is made up of waves of compressed air that reach our ears, which our brains interpret as audio sensation. We can graph the intensity and frequency of these waves over time. The numbers on this graph can then be converted to binary numbers and stored on a computer, which later control speakers to reproduce the sound. This is a simplification of how computer audio works, but describes how numbers can represent Beethoven's Symphony No. 5.

The data for several images combines with audio data to store videos. All forms of information can be encoded into binary numbers. There is, of course, a great deal more detail to it than this, but this is how 1s and 0s represent the wide variety of data in our information age.

Summary

You can compute expressions with a calculator or enter string concatenations with a word processor. You can even do string replication easily by copying and pasting text. But expressions, and their component values—operators, variables, and function calls—are the basic building blocks that make up programs. Once you know how to handle these elements, you will be able to instruct Python to operate on large amounts of data for you.

You'll find it helpful to remember the different types of operators (`+`, `-`, `*`, `/`, `//`, `%`, and `**` for math operations, and `+` and `*` for string operations) and the three data types (integers, floating-point numbers, and strings) introduced in this chapter.

I introduced a few different functions as well. The `print()` and `input()` functions handle simple text output (to the screen) and input (from the keyboard). The `len()` function takes a string and evaluates to an int of the number of characters in the string. The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, or floating-point number form of the value they are passed. The `round()` function returns the rounded integer, and the `abs()` function returns the absolute value of the arguments.

In the next chapter, you'll learn how to tell Python to make intelligent decisions about what code to run, what code to skip, and what code to repeat based on the values it has. This is known as *flow control*, and it allows you to write programs that make intelligent decisions.

Practice Questions

1. Which of the following are operators, and which are values?

```
*
'hello'
-88.8
-
/
+
5
```

2. Which of the following is a variable, and which is a string?

```
spam
'spam'
```

3. Name three data types.
4. What is an expression made up of? What do all expressions do?
5. This chapter introduced assignment statements, like `spam = 10`. What is the difference between an expression and a statement?
6. What does the variable `bacon` contain after the following code runs?

```
bacon = 20
bacon + 1
```

7. What should the following two expressions evaluate to?

```
'spam' + 'spamspam'
'spam' * 3
```

8. Why is `eggs` a valid variable name while `100` is invalid?
9. What three functions can be used to get the integer, floating-point number, or string version of a value?
10. Why does this expression cause an error? How can you fix it?

```
'I eat ' + 99 + ' burritos.'
```

Extra credit: Search online for the Python documentation for the `len()` function. It will be on a web page titled "Built-in Functions." Skim the list of other functions Python has, look up what the `bin()` and `hex()` functions do, and experiment with them in the interactive shell.