Chapter 1

Getting Started

This chapter will get you up and running with Python, from downloading it to writing simple programs.

1.1 Installing Python

Go to www.python.org and download the latest version of Python (version 3.5 as of this writing). It should be painless to install. If you have a Mac or Linux, you may already have Python on your computer, though it may be an older version. If it is version 2.7 or earlier, then you should install the latest version, as many of the programs in this book will not work correctly on older versions.

1.2 IDLE

IDLE is a simple integrated development environment (IDE) that comes with Python. It's a program that allows you to type in your programs and run them. There are other IDEs for Python, but for now I would suggest sticking with IDLE as it is simple to use. You can find IDLE in the Python 3.4 folder on your computer.

When you first start IDLE, it starts up in the shell, which is an interactive window where you can type in Python code and see the output in the same window. I often use the shell in place of my calculator or to try out small pieces of code. But most of the time you will want to open up a new window and type the program in there.

Note At least on Windows, if you click on a Python file on your desktop, your system will run the program, but not show the code, which is probably not what you want. Instead, if you right-click on the file, there should be an option called `Edit with Idle`. To edit an existing Python file,

either do that or start up IDLE and open the file through the `File` menu.

Keyboard shortcuts The following keystrokes work in IDLE and can really speed up your work.

Keystroke	Result
`CTRL+C`	Copy selected text
`CTRL+X`	Cut selected text
`CTRL+V`	Paste
`CTRL+Z`	Undo the last keystroke or group of keystrokes
`CTRL+SHIFT+Z`	Redo the last keystroke or group of keystrokes
`F5`	Run module

1.3 A first program

Start IDLE and open up a new window (choose `New Window` under the `File` Menu). Type in the following program.

```
temp = eval(input('Enter a temperature in Celsius: '))
print('In Fahrenheit, that is', 9/5*temp+32)
```

Then, under the `Run` menu, choose `Run Module` (or press F5). IDLE will ask you to save the file, and you should do so. Be sure to append `.py` to the filename as IDLE will not automatically append it. This will tell IDLE to use colors to make your program easier to read.

Once you've saved the program, it will run in the shell window. The program will ask you for a temperature. Type in 20 and press enter. The program's output looks something like this:

```
Enter a temperature in Celsius: 20
In Fahrenheit, that is 68.0
```

Let's examine how the program does what it does. The first line asks the user to enter a temperature. The **`input`** function's job is to ask the user to type something in and to capture what the user types. The part in quotes is the prompt that the user sees. It is called a *string* and it will appear to the program's user exactly as it appears in the code itself. The **`eval`** function is something we use here, but it won't be clear exactly why until later. So for now, just remember that we use it when we're getting numerical input.

We need to give a name to the value that the user enters so that the program can remember it and use it in the second line. The name we use is `temp` and we use the equals sign to assign the user's value to `temp`.

The second line uses the **`print`** function to print out the conversion. The part in quotes is another string and will appear to your program's user exactly as it appears in quotes here. The second

argument to the **print** function is the calculation. Python will do the calculation and print out the numerical result.

This program may seem too short and simple to be of much use, but there are many websites that have little utilities that do similar conversions, and their code is not much more complicated than the code here.

A second program Here is a program that computes the average of two numbers that the user enters:

```
num1 = eval(input('Enter the first number: '))
num2 = eval(input('Enter the second number: '))
print('The average of the numbers you entered is', (num1+num2)/2)
```

For this program we need to get two numbers from the user. There are ways to do that in one line, but for now we'll keep things simple. We get the numbers one at a time and give each number its own name. The only other thing to note is the parentheses in the average calculation. This is because of the order of operations. All multiplications and divisions are performed before any additions and subtractions, so we have to use parentheses to get Python to do the addition first.

1.4 Typing things in

Case Case matters. To Python, `print`, `Print`, and `PRINT` are all different things. For now, stick with lowercase as most Python statements are in lowercase.

Spaces Spaces matter at the beginning of lines, but not elsewhere. For example, the code below will not work.

```
temp = eval(input('Enter a temperature in Celsius: '))
  print('In Fahrenheit, that is', 9/5*temp+32)
```

Python uses indentation of lines for things we'll learn about soon. On the other hand, spaces in most other places don't matter. For instance, the following lines have the same effect:

```
print('Hello world!')
print ('Hello world!')
print( 'Hello world!' )
```

Basically, computers will only do what you tell them, and they often take things very literally. Python itself totally relies on things like the placement of commas and parentheses so it knows what's what. It is not very good at figuring out what you mean, so you have to be precise. It will be very frustrating at first, trying to get all of the parentheses and commas in the right places, but after a while it will become more natural. Still, even after you've programmed for a long time, you will still miss something. Fortunately, the Python interpreter is pretty good about helping you find your mistakes.

1.5 Getting input

The **input** function is a simple way for your program to get information from people using your program. Here is an example:

```
name = input('Enter your name: ')
print('Hello, ', name)
```

The basic structure is

variable name = **input**(*message to user*)

The above works for getting text from the user. To get numbers from the user to use in calculations, we need to do something extra. Here is an example:

```
num = eval(input('Enter a number: '))
print('Your number squared:', num*num)
```

The **eval** function converts the text entered by the user into a number. One nice feature of this is you can enter expressions, like `3*12+5`, and **eval** will compute them for you.

Note If you run your program and nothing seems to be happening, try pressing enter. There is a bit of a glitch in IDLE that occasionally happens with **input** statements.

1.6 Printing

Here is a simple example:

```
print('Hi there')
```

The **print** function requires parenthesis around its arguments. In the program above, its only argument is the string `'Hi there'`. Anything inside quotes will (with a few exceptions) be printed exactly as it appears. In the following, the first statement will output `3+4`, while the second will output `7`.

```
print('3+4')
print(3+4)
```

To print several things at once, separate them by commas. Python will automatically insert spaces between them. Below is an example and the output it produces.

```
print('The value of 3+4 is', 3+4)
print('A', 1, 'XYZ', 2)
```

```
The value of 3+4 is 7
A 1 XYZ 2
```

Optional arguments

There are two optional arguments to the **print** function. They are not overly important at this stage of the game, so you can safely skip over this section, but they are useful for making your output look nice.

sep Python will insert a space between each of the arguments of the print function. There is an optional argument called `sep`, short for separator, that you can use to change that space to something else. For example, using `sep=':'` would separate the arguments by a colon and `sep='##'` would separate the arguments by two pound signs.

One particularly useful possibility is to have nothing inside the quotes, as in `sep=''`. This says to put no separation between the arguments. Here is an example where `sep` is useful for getting the output to look nice:

```
print ('The value of 3+4 is', 3+4, '.')
print ('The value of 3+4 is ', 3+4, '.', sep='')
```

```
The value of 3+4 is 7 .
The value of 3+4 is 7.
```

end The print function will automatically advance to the next line. For instance, the following will print on two lines:

```
print('On the first line')
print('On the second line')
```

```
On the first line
On the second line
```

There is an optional argument called `end` that you can use to keep the print function from advancing to the next line. Here is an example:

```
print('On the first line', end='')
print('On the second line')
```

```
On the first lineOn the second line
```

Of course, this could be accomplished better with a single print, but we will see later that there are interesting uses for the `end` argument.

1.7 Variables

Looking back at our first program, we see the use of a variable called `temp`:

```
temp = eval(input('Enter a temperature in Celsius: '))
print('In Fahrenheit, that is', 9/5*temp+32)
```

One of the major purposes of a variable is to remember a value from one part of a program so that it can be used in another part of the program. In the case above, the variable `temp` stores the value that the user enters so that we can do a calculation with it in the next line.

In the example below, we perform a calculation and need to use the result of the calculation in several places in the program. If we save the result of the calculation in a variable, then we only need to do the calculation once. This also helps to make the program more readable.

```
temp = eval(input('Enter a temperature in Celsius: '))
f_temp = 9/5*temp+32
print('In Fahrenheit, that is', f_temp)
if f_temp > 212:
    print('That temperature is above the boiling point.')
if f_temp < 32:
    print('That temperature is below the freezing point.')
```

We haven't discussed if statements yet, but they do exactly what you think they do.

A second example Here is another example with variables. Before reading on, try to figure out what the values of `x` and `y` will be after the code is executed.

```
x=3
y=4
z=x+y
z=z+1
x=y
y=5
```

After these four lines of code are executed, `x` is 4, `y` is 5 and `z` is 8. One way to understand something like this is to take it one line at a time. This is an especially useful technique for trying to understand more complicated chunks of code. Here is a description of what happens in the code above:

1. `x` starts with the value 3 and `y` starts with the value 4.
2. In line 3, a variable `z` is created to equal `x+y`, which is 7.
3. Then the value of `z` is changed to equal one more than it currently equals, changing it from 7 to 8.
4. Next, `x` is changed to the current value of `y`, which is 4.
5. Finally, `y` is changed to 5. Note that this does not affect `x`.
6. So at the end, `x` is 4, `y` is 5, and `z` is 8.

Variable names

There are just a couple of rules to follow when naming your variables.

- Variable names can contain letters, numbers, and the underscore.
- Variable names *cannot* contain spaces.
- Variable names *cannot* start with a number.
- Case matters—for instance, `temp` and `Temp` are different.

It helps make your program more understandable if you choose names that are descriptive, but not so long that they clutter up your program.

1.8 Exercises

1. Print a box like the one below.

```
**********
**********
**********
**********
```

2. Print a box like the one below.

```
**********
*        *
*        *
**********
```

3. Print a triangle like the one below.

```
*
**
***
****
```

4. Write a program that computes and prints the result of $\frac{512-282}{47 \cdot 48 + 5}$. It is roughly .1017.

5. Ask the user to enter a number. Print out the square of the number, but use the `sep` optional argument to print it out in a full sentence that ends in a period. Sample output is shown below.

```
Enter a number: 5
The square of 5 is 25.
```

6. Ask the user to enter a number x. Use the `sep` optional argument to print out x, $2x$, $3x$, $4x$, and $5x$, each separated by three dashes, like below.

```
Enter a number: 7
7---14---21---28---35
```

7. Write a program that asks the user for a weight in kilograms and converts it to pounds. There are 2.2 pounds in a kilogram.

8. Write a program that asks the user to enter three numbers (use three separate input statements). Create variables called `total` and `average` that hold the sum and average of the three numbers and print out the values of `total` and `average`.

9. A lot of cell phones have tip calculators. Write one. Ask the user for the price of the meal and the percent tip they want to leave. Then print both the tip amount and the total bill with the tip included.

Chapter 2

For loops

Probably the most powerful thing about computers is that they can repeat things over and over very quickly. There are several ways to repeat things in Python, the most common of which is the for loop.

2.1 Examples

Example 1 The following program will print `Hello` ten times:

```
for i in range(10):
    print('Hello')
```

The structure of a for loop is as follows:

> `for` *variable name* `in range(` *number of times to repeat* `):`
> *statements to be repeated*

The syntax is important here. The word `for` must be in lowercase, the first line must end with a colon, and the statements to be repeated *must* be indented. Indentation is used to tell Python which statements will be repeated.

Example 2 The program below asks the user for a number and prints its square, then asks for another number and prints its square, etc. It does this three times and then prints that the loop is done.

```
for i in range(3):
    num = eval(input('Enter a number: '))
    print ('The square of your number is', num*num)
print('The loop is now done.')
```

```
Enter a number: 3
The square of your number is 9
Enter a number: 5
The square of your number is 25
Enter a number: 23
The square of your number is 529
The loop is now done.
```

Since the second and third lines are indented, Python knows that these are the statements to be repeated. The fourth line is not indented, so it is not part of the loop and only gets executed once, after the loop has completed.

Looking at the above example, we see where the term *for loop* comes from: we can picture the execution of the code as starting at the `for` statement, proceeding to the second and third lines, then looping back up to the `for` statement.

Example 3 The program below will print A, then B, then it will alternate C's and D's five times and then finish with the letter E once.

```
print('A')
print('B')
for i in range(5):
    print('C')
    print('D')
print('E')
```

The first two print statements get executed once, printing an A followed by a B. Next, the C's and D's alternate five times. Note that we don't get five C's followed by five D's. The way the loop works is we print a C, then a D, then loop back to the start of the loop and print a C and another D, etc. Once the program is done looping with the C's and D's, it prints one E.

Example 4 If we wanted the above program to print five C's followed by five D's, instead of alternating C's and D's, we could do the following:

```
print('A')
print('B')
for i in range(5):
    print('C')
for i in range(5):
    print('D')
print('E')
```

2.2 The loop variable

There is one part of a for loop that is a little tricky, and that is the loop variable. In the example below, the loop variable is the variable `i`. The output of this program will be the numbers 0, 1, . . . , 99, each printed on its own line.

```
for i in range(100):
    print(i)
```

When the loop first starts, Python sets the variable `i` to 0. Each time we loop back up, Python increases the value of `i` by 1. The program loops 100 times, each time increasing the value of `i` by 1, until we have looped 100 times. At this point the value of `i` is 99.

You may be wondering why `i` starts with 0 instead of 1. Well, there doesn't seem to be any really good reason why other than that starting at 0 was useful in the early days of computing and it has stuck with us. In fact most things in computer programming start at 0 instead of 1. This does take some getting used to.

Since the loop variable, `i`, gets increased by 1 each time through the loop, it can be used to keep track of where we are in the looping process. Consider the example below:

```
for i in range(3):
    print(i+1, '-- Hello')
```

```
1 -- Hello
2 -- Hello
3 -- Hello
```

Names There's nothing too special about the name `i` for our variable. The programs below will have the exact same result.

```
for i in range(100):
    print(i)
```

```
for wacky_name in range(100):
    print(wacky_name)
```

It's a convention in programming to use the letters i, j, and k for loop variables, unless there's a good reason to give the variable a more descriptive name.

2.3 The `range` function

The value we put in the **`range`** function determines how many times we will loop. The way **`range`** works is it produces a list of numbers from zero to the value minus one. For instance, **`range`**`(5)` produces five values: 0, 1, 2, 3, and 4.

If we want the list of values to start at a value other than 0, we can do that by specifying the starting value. The statement `range(1,5)` will produce the list 1, 2, 3, 4. This brings up one quirk of the `range` function—it stops one short of where we think it should. If we wanted the list to contain the numbers 1 through 5 (including 5), then we would have to do `range(1,6)`.

Another thing we can do is to get the list of values to go up by more than one at a time. To do this, we can specify an optional step as the third argument. The statement `range(1,10,2)` will step through the list by twos, producing 1, 3, 5, 7, 9.

To get the list of values to go backwards, we can use a step of -1. For instance, `range(5,1,-1)` will produce the values 5, 4, 3, 2, in that order. (Note that the `range` function stops one short of the ending value 1). Here are a few more examples:

Statement	Values generated
`range(10)`	`0,1,2,3,4,5,6,7,8,9`
`range(1,10)`	`1,2,3,4,5,6,7,8,9`
`range(3,7)`	`3,4,5,6`
`range(2,15,3)`	`2,5,8,11,14`
`range(9,2,-1)`	`9,8,7,6,5,4,3`

Here is an example program that counts down from 5 and then prints a message.

```
for i in range(5,0,-1):
    print(i, end=' ')
print('Blast off!!')
```

```
5 4 3 2 1 Blast off!!!
```

The `end=' '` just keeps everything on the same line.

2.4 A Trickier Example

Let's look at a problem where we will make use of the loop variable. The program below prints a rectangle of stars that is 4 rows tall and 6 rows wide.

```
for i in range(4):
    print('*'*6)
```

The rectangle produced by this code is shown below on the left. The code `'*'*6` is something we'll cover in Section 6.2; it just repeats the asterisk character six times.

```
******
******
******
******
```

```
*
**
***
****
```

Suppose we want to make a triangle instead. We can accomplish this with a very small change to the rectangle program. Looking at the program, we can see that the for loop will repeat the **print** statement four times, making the shape four rows tall. It's the `6` that will need to change.

The key is to change the `6` to `i+1`. Each time through the loop the program will now print `i+1` stars instead of 6 stars. The loop counter variable `i` runs through the values 0, 1, 2, and 3. Using it allows us to vary the number of stars. Here is triangle program:

```
for i in range(4):
    print('*'*(i+1))
```

2.5 Exercises

1. Write a program that prints your name 100 times.
2. Write a program to fill the screen horizontally and vertically with your name. [Hint: add the option `end=''` into the **print** function to fill the screen horizontally.]
3. Write a program that outputs 100 lines, numbered 1 to 100, each with your name on it. The output should look like the output below.

```
1 Your name
2 Your name
3 Your name
4 Your name
...
100 Your name
```

4. Write a program that prints out a list of the integers from 1 to 20 and their squares. The output should look like this:

```
1 --- 1
2 --- 4
3 --- 9
...
20 --- 400
```

5. Write a program that uses a for loop to print the numbers 8, 11, 14, 17, 20, . . . , 83, 86, 89.
6. Write a program that uses a for loop to print the numbers 100, 98, 96, . . . , 4, 2.
7. Write a program that uses exactly four for loops to print the sequence of letters below.

```
AAAAAAAAAABBBBBBBCDCDCDCDEFFFFFFG
```

8. Write a program that asks the user for their name and how many times to print it. The program should print out the user's name the specified number of times.

9. The Fibonacci numbers are the sequence below, where the first two numbers are 1, and each number thereafter is the sum of the two preceding numbers. Write a program that asks the user how many Fibonacci numbers to print and then prints that many.

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

10. Use a for loop to print a box like the one below. Allow the user to specify how wide and how high the box should be. [Hint: `print('*'*10)` prints ten asterisks.]

```
********************
********************
********************
********************
```

11. Use a for loop to print a box like the one below. Allow the user to specify how wide and how high the box should be.

```
********************
*                  *
*                  *
********************
```

12. Use a for loop to print a triangle like the one below. Allow the user to specify how high the triangle should be.

```
*
**
***
****
```

13. Use a for loop to print an upside down triangle like the one below. Allow the user to specify how high the triangle should be.

```
****
***
**
*
```

14. Use for loops to print a diamond like the one below. Allow the user to specify how high the diamond should be.

```
    *
   ***
  *****
*******
  *****
   ***
    *
```

15. Write a program that prints a giant letter A like the one below. Allow the user to specify how large the letter should be.