Chapter 1

Introduction

Python is both a useful all-purpose programming language and an excellent tool for introducing the concepts of programming. In this chapter we'll see how to install and run Python, and also how to use the programming environment that comes with Python. You might just skim the last few sections of this chapter at the start and then come back to them after you have studied the material in chapters 2 through 4. Much of the advice found here will seem more relevant after you know a bit about programming.

1.1 About Python and Other Programming Languages

Although a host of science fiction movies would have it otherwise, computers don't think, they just follow simple instructions. Computers are not even sophisticated enough to understand human languages, they need their own ultra-simple languages. In the end, the instructions that computers can follow are things like "Take the numbers from this memory location and that memory location, add them together, and put result into this location." The first programming languages in the 1950's were expressed in terms very similar to these. Because it is difficult for people to think in such basic terms, computer scientists have gradually learned how to create languages that humans can easily work with, but that can be translated into the simple instructions that can be followed by a computer. These are called *programming languages*. Thousands of programming languages have been created, but a relatively small number are in common use today. In these notes we will work with one language, called *Python* that was invented in 1989 by the Dutch programmer Guido van Rossum. Unfortunately, people don't always get along, and in 2008 Python bifurcated into two similar languages, Python2 and Python3 because the language developers couldn't agree on what were mainly minor differences of syntax. In these notes we will use Python3 and we will usually refer to the language simply as *Python*.

Just to give you a sense of the variety of programming languages that are in use, here are programs from different languages that all print to the computer screen the result of multiplying 3 and 4:

```
#include <stdio.h>
int main(void)
{
        int x, y;
        x = 3;
        y = 4;
        printf("%d\n", x*y);
        return 0;
}
```

Program 1.1.1: The C programming language

```
#include <iostream>

using namespace std;
int main()
{
        int x, y;
        x = 3;
        y=4;
        cout << x*y << endl;
        return 0;
}
```

Program 1.1.2: The C++ programming language

```
program Times
        integer x
        integer y
        x = 3
        y = 4
        print *, x*y
end program Times
```

Program 1.1.3: Fortran

```
public class Times {
   public static void main(String[] args) {
        int x, y;
        x = 3;
        y = 4;
        System.out.println(x*y);
   }
}
```

Program 1.1.4: Java

Here this program is in Python:

```
x = 3
y = 4
print( x*y )
```

Program 1.1.5: Python

Notice how much simpler this program is in Python. There are two things that Python does differently than most programming languages:

- Most languages group statements together with some type of bracket or connective terminology. The "{" and "}" symbols you see in some of these programs are examples of this. Python uses instead the way the program is laid out on the page. This makes Python programs easier to read.

- Most languages require the programmer to say what kind of data will be stored in variables. In the programs above "x" and "y" are variables – names attached to memory locations that will be used to store data. All of the examples except the one in Python have statements that say that these are "int" or "integer" variables. The Python system deduces this information from the program itself; it does not require the user to specify it.

In general, Python has less verbiage around a program and lets the programmer concentrate on the actual instructions in the code. This helps to make Python programs are easier to read, and, as we will emphasize many times in these notes, readable programs are more likely to be correct.

There are two types of programming languages. In both types the program needs to be run through a software system, because almost no one wants to program in a language that the computer can execute directly. *Compiled* languages can be converted into machine code. Programs in such languages are run through a *compiler*, a program that converts the compilable program into machine code. This produces an *executable* program – a program that runs directly on the machine. Most of the programs you have ever run, such as Microsoft Word or Mozilla Firefox, are compiled. The compiler is needed to produce the executable program, after that the executable stands on its own and can even be copied and transferred to another system; the compile is no longer needed. *Interpreted* programs are not converted to machine code; instead, they are run through a system called an *interpreter* that executes the program one statement at a time in the same way a computer would if it could understand the language the program was written in. An interpreter needs to be used every time the program is run. Python is interpreted, though there are ways to produce standalone Python programs that can be run without an interpreter. In general compiled programs run more efficiently (faster, using less memory) than interpreted programs, though machines have become so fast and memory so cheap that for many applications this is not an important distinction.

Understanding programs

Learning to read programs is an important part of learning to program. You can't read a program the way you read a book. You should read programs the same way a computer does. Every instruction does one of two things – it either changes the computer's memory, or it alters the sequence in which the statements are executed. Consider the following portion of a program:

```
x = 5
if x < 10:
    print( "small" )
else:
    print( "big" )
```

The first statement, x=5, changes the data stored in variable x (a memory location) to 5. The next line, **if** x < 10: looks into the computer's memory for the value stored in variable x (we just set that value in the previous line, but the system doesn't remember this from one statement to the next. The system doesn't *understand* the program, it just follows it one instruction at a time.) If the value in x is less than 10, the system executes the next line, which is **print**("small"); if the value in x is not less than 10, the system executes the line following **else**:, which is **print**("big"). Of course, you can do this in your head if the program only has a few lines, but we will get to more sophisticated programs quickly. When you are reading them, keep a piece of paper that represents the computer's memory. Each time a variable is given a value, write it down on your paper. Walk through the program from the starting point (you'll see where that is in section 2.1) until you run out of instructions and you will be executing the program the same way the computer does. This might seem tedious, and it is. Computers are not "smart", they are just fast and persistent. They don't get bored. Computers do sophisticated things by doing many, many simple instructions very, very quickly. In this course you will learn to write these instructions in ways that will reliably get the computer to do the things you want.