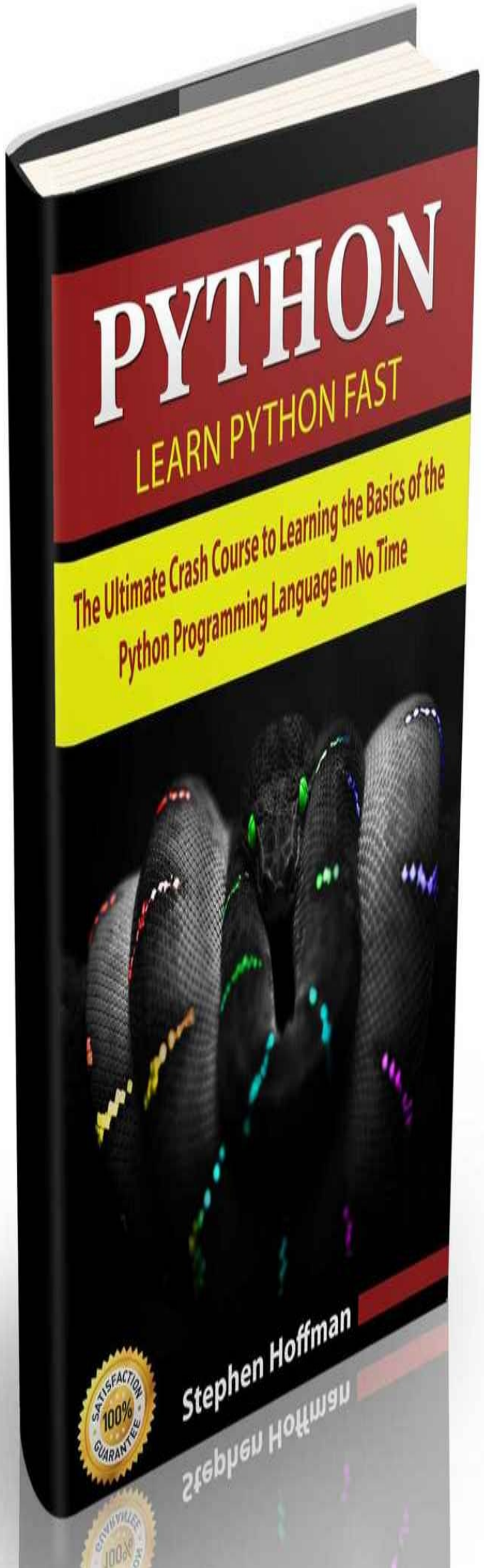
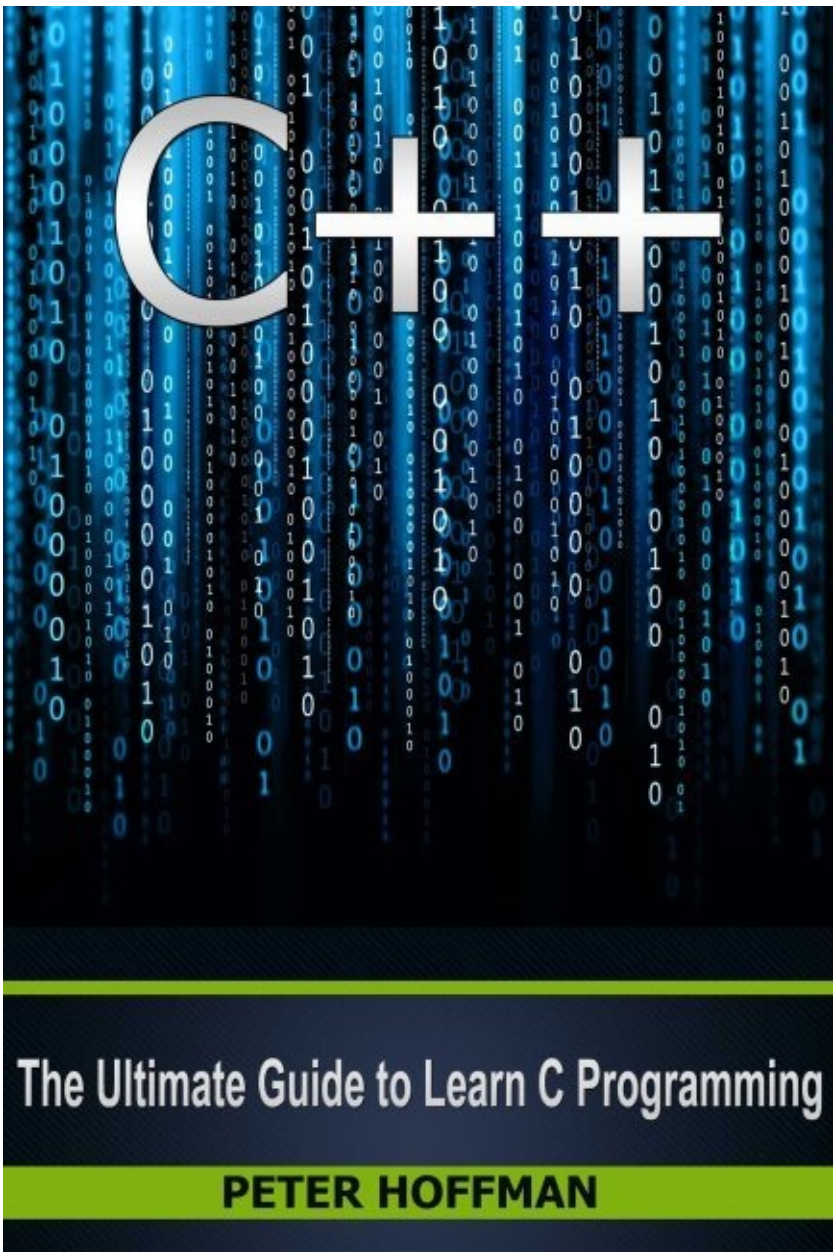

C++
The Ultimate Guide to Learn C Programming
PETER HOFFMAN


PYTHON
LEARN PYTHON FAST
The Ultimate Crash Course to Learning the Basics of the Python Programming Language In No Time
SATISFACTION 100% GUARANTEE
Stephen Hoffman

C++

The Ultimate Crash Course to Learning the Basics of C++ and the Python Programming Language (c plus plus, how to program, python programming, python language)


C++
The Ultimate Guide to Learn C Programming
PETER HOFFMAN

C++

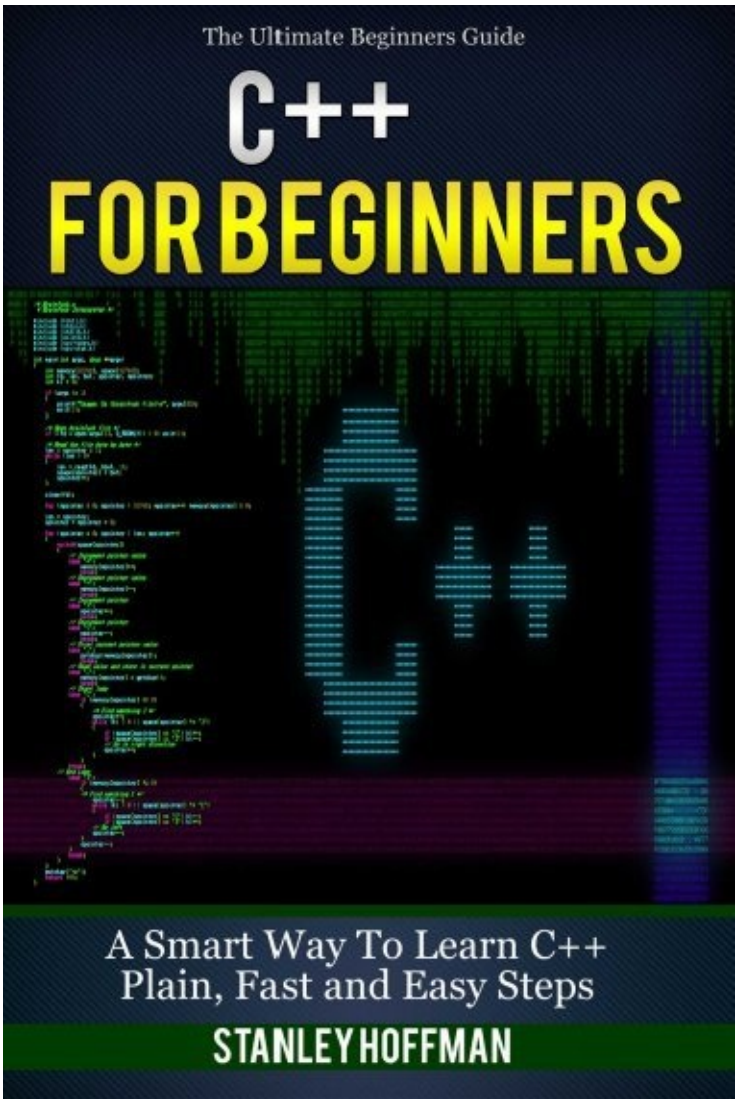
The Ultimate Guide to Learn C Programming (c plus plus, C++ for beginners, programming computer, how to program)

PETER HOFFMAN

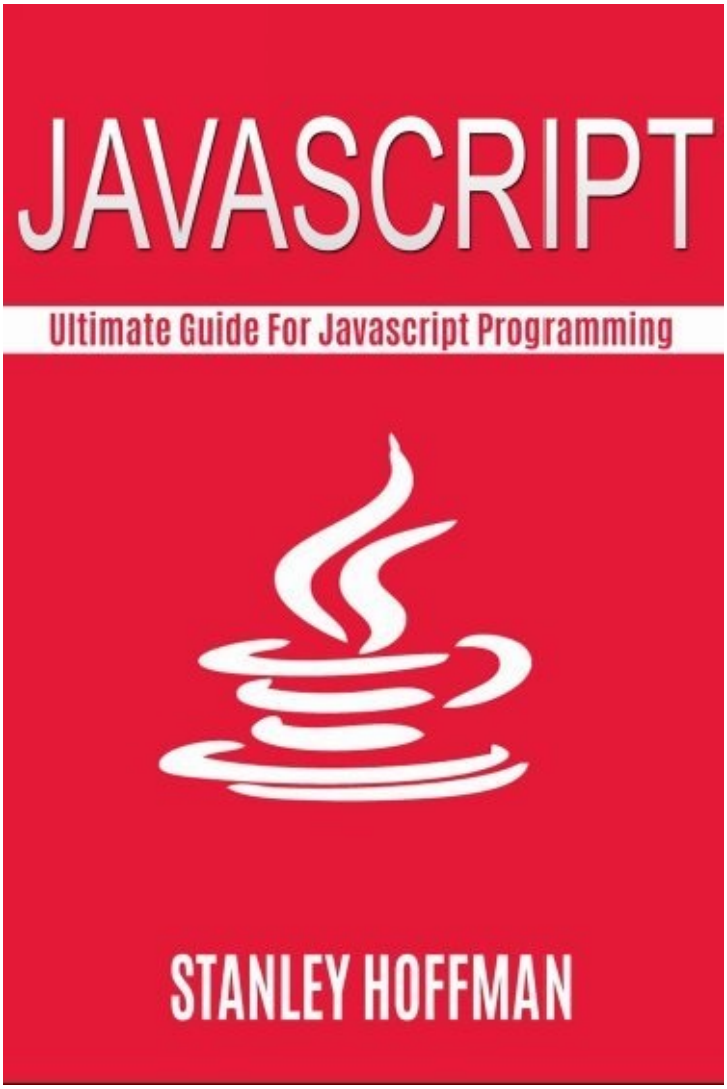
CONTENTS

I think next books will also be interesting for you:

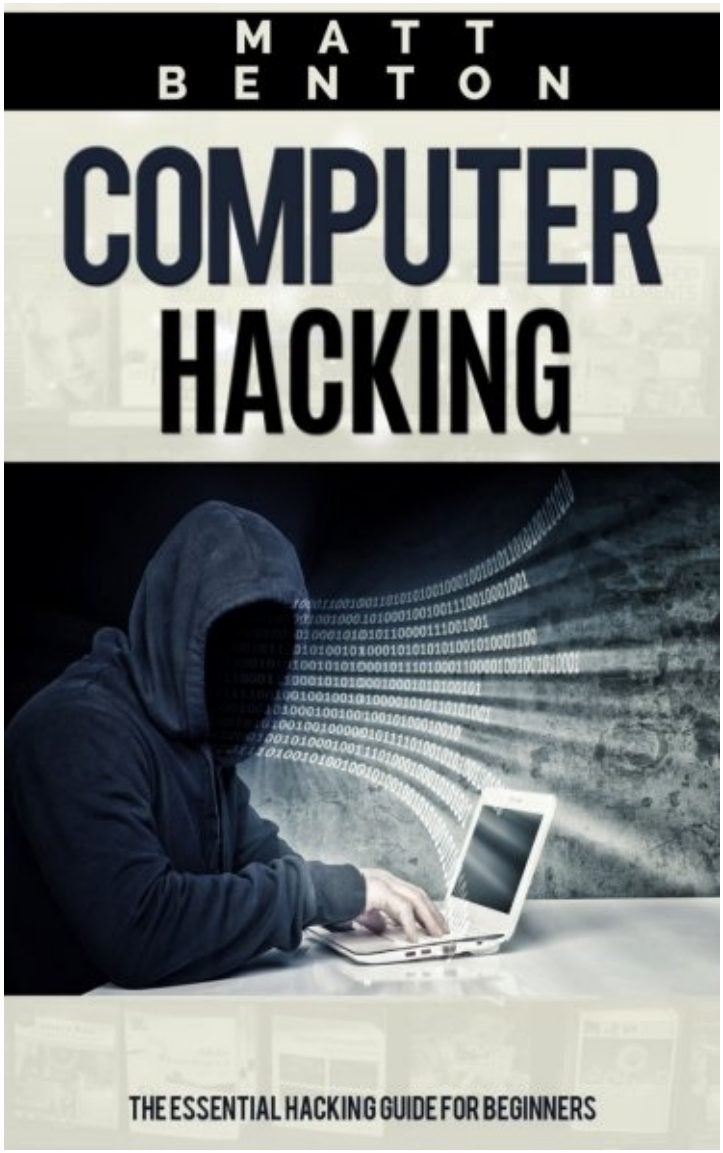
C++



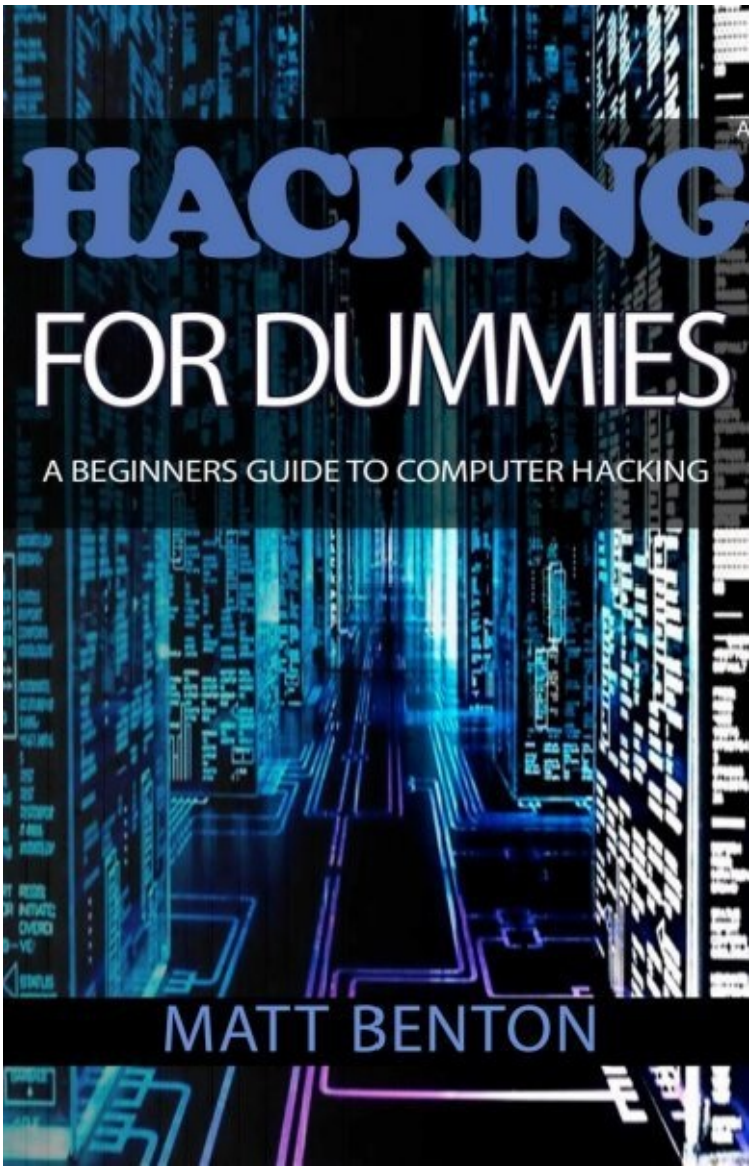
Javascript


JAVASCRIPT
Ultimate Guide For Javascript Programming
STANLEY HOFFMAN

Computer Hacking



[Hacking for Dummies]



Introduction

A C++ program is simply a text file which contains a sequence of commands using specific C++ text. We can call this text file the *source file*, and it will normally end in a .CPP extension (just as an audio file ends in .MP3 or a PDF document ends in .PDF).

The purpose of writing a C++ program is to put together a specific sequence of C++ commands which will be converted by the computer into *machine-language* which will perform whatever task that we want done. This conversion process is called *compiling* which is performed by the compiler. In addition, the code that we write must be written to include both setup and tear down instructions in a process which is called *linking*. Together, the process of compiling and linking is called *building*, which builds our code into a machine-executable .EXE file of which the computer can run as a program that it understands.

In order to write a program in C++, you will need only two things – an editor to write the code and build the .CPP file, and a compiler which will convert the code (the source file) into a machine-executable file which will do the tasks that we have programmed it to do (such as open a file, play a video, print text on the screen, and so on).

This book will use examples for you to follow along with using Microsoft Visual Studio – the perfect companion to both the beginner and more advanced C++ programmer.

Chapter 1 – WRITING YOUR FIRST C++ PROGRAM

Before we write our first program in C++, you are going to need to download the software called Microsoft Visual Studio, which allows you to write, compile and execute commands and programs written in C++ code language. It is also an excellent program which can be used to debug and edit your program should it need changes or if something goes wrong.

As your first program, we are going to explore the basic “Hello World” program which is a must for everybody who is wanting to learn how to write and code C++. The Hello World program will demonstrate the absolute bare essentials that nearly every program written in C++ will share. It does not matter how simple, or how complex a program is, it will still require the following few lines of very basic code.

The first step you need to do, is open Microsoft Visual Studio and create a new project. We will call this project “Hello World” or “First Program” if you prefer. Don’t worry if you do not understand the following lines of text just yet, as we will have a closer look at them in a few moments and explore them deeper in the chapters which are to follow.

Firstly, you should write the following lines in Microsoft Visual Studio:

```
#include <iostream>
using namespace std;
```

These first lines are basically what we call a “pre-processor directive”, or in other words, letting the code to follow know that we are going to start a program. The letters “i” and “o” in the word <iostream> stand for both input and output, which is required in every C++ program that is written. The final line, “using namespace std”, means that we are telling the computer that we will be using the standard library (a library in this case meaning the type of commands we are using and writing). In other words, we are tell the compiler (Microsoft Visual Studio), to basically “include” the library called “iostream”. This library is what holds all of our functions that we use, but more on that in a minute.

The next lines that you should write are:

```
int main () {
```

These two lines tell the program to start. It basically means that we are showing the “start” of the program to the compiler. The { is very important here, and is known as a “Scope Bracket”. It is important, and will become very important as we continue to write this program and any others that exist, because every single line of code that you write must end in a ; or { / }.

The next lines that you should write are:

```
cout <<”Hello World”<<endl;
```

This line is very simple. The word “cout” is a function which we are using from the iostream library, and simply works as a print statement. In other words, by using the cout function, you are telling the program that you would like to print (display on screen) the words “Hello World”. Feel free to change these words to whatever you want. The line ends in “endl;”, which is where we are basically telling the program that we are going to be writing another line after this one. It is basically the same as pressing the Enter key on your computer keyboard to write another line.

The second last line of code should be:

return 0;

This is where we are telling our compiler to “return” back to the command prompt (which we will discuss later in more depth) if the program ran smoothly without any problems.

Finally, the last line of code should be written as:

} //end main

The } is basically the closing bracket. As we discussed above, all commands should end in a Scope Bracket which tells the compiler that the command is finished (for that line, or for the whole program). The last part, “//end main” is a function that we call a “comment”. These are what we use to keep track of what we are using and writing in the program. In the case where we are writing a very long program, this final statement is basically the closing bracket to the function that we started with, int main(), which will keep the whole program together as one entity.

Now, our program should look like this:

```
#include <iostream>
using namespace std;
int main () {
    cout <<”Hello World”<<endl;

    return 0;
} //end main
```

Now when we save our program and then run it, it will display the words “Hello World”. Try it again, except this time I want you to write something else and see what happens.

In the next chapter we will start to look at some of the major features and functions of a program.

Chapter 2 – EXPANDING YOUR PROGRAM: VARIABLES

The most important part of C++ code is a *variable*. You can look at these as if they were small storage boxes where we can store things for later use by the program – especially numbers. The term and use of these variables comes from mathematics. You should be familiar with something like this:

x = 1

This means that x is equal to the number 1. The statement is also saying that wherever this letter x appears, it will be worth the number 1. In essence, the letter x is *holding* or *storing* the number 1 until it needs to be used in a statement or algorithm. Variables work in exactly the same way in C++, except for the fact that we will always include a scope bracket at the end to imply and tell the program that it is the end of a function. For example:

x = 1;

From this point on, wherever we use the letter x in the program, it will be recognised as the number 1. But how would be declare variables? Generally we are not going to just use one variable when it would be much easier to just write the number. Lets look at an example from algebra that you might see in a more advanced level of mathematics:

(x + 2) = y / 2

x + 4 = y

solve for x and y

If you are any good at algebra you may have already solved the problem. However, simply inputting this information into C++ is not enough for the program to understand what this all means. It will read it only as gibberish and it will make no sense at all to the computer. This is why we must *declare* our variables to the program before we can use it, so that the computer knows exactly what we are trying to do. So we would declare our variables by:

```
int x;
x = 10;

int y;
y = 5;
```

As we have seen in the previous chapter, the statement “int” basically means “start”. Here, we are saying that x equals 10 and y equals 5. If we were then to write the code again, the program would be able to solve the problem. We can use these variables anywhere that we like in the program, but we must always declare to the system what they mean before we can use them. And it is not only numbers that we can store in the variables. We could also use the following:

```
x = 1
x = 2.3
x = “this is a sentence”
```

To give you a further example of how variables work, consider the following. I tell you to remember the number 5 and the number 2. You have just stored two values in your mind. I then tell you to add the number 1 to the first number which I said. Now, in your memory, you should have the number 6 (5+1) and the number 2. I then ask you to subtract the first number from the second and then we get the answer of 4.

This basically explains how C++ will store variables and then use them to solve problems or statements. When actually writing it in code, it would look like this:

```
a = 5;
b = 2;
a = a + 1;
result = a – b;
```

This is a very simple statement which you could have probably solved faster than writing it, but just remember that your computer has the potential to calculate billions of these problems every second, so you should be able to see the bigger picture of what each of these variables can do for you and larger programs.

But what if we wanted to write a small program which we could practice declaring and using variables? There are two ways which we could write this. We can write in a single statement such as:

```
int a, b, c;
```

Which means that a,b,c are variables with the type “int”, which means the same as:

```
int a;
int b;
int c;
```

It depends on whether we want to type more than one line, but both represent the same statement. To use both in an example, and to see our mathematical problem we just solved using variables, lets have a look at what it would look like in a small program on the next page.

```
// operating with variables

#include <iostream>
using namespace std;
```

```
int main ()
{
    // declaring variables:
    int a, b;
    int result;

    // process:
    a = 5;
    b = 2;
    a = a + 1;
    result = a – b;

    // print out the result:
    cout << result;

    // terminate the program:
    return 0;
}
```

When we run the program, we will see that it will display the answer for us. Very simple and you should be able to interpret what each of it means using what we have discussed so far with our “Hello World” program and the variables we have used. However, we do have something new here, and that is the use of “//” followed by some text. The first “// operating with variables” is simply a title for our program, and each of the following // are simply sub-headings explaining what we are doing at that point of the program. It makes it a lot easier, especially if you are dealing with a very large amount of code, to go back through the lines and find what you are looking for instead of having to try and decipher each line.

There is also an easier way to write the code for the example that we just used. This second method is called the “initialization” of a variable, and there are three ways that we can do it. To *initialize* a variable, basically means that we can declare its value at the same time that we declare the variable. The three ways that we can do this is by writing one of the following:

```
int x = 0;
```

```
int x (0);

int x {0};
```

It does not matter which method that you use because they all do the same thing. The best part however, is that you can use all three types at once if you chose to. For example:

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
    int a=5;
    int b(3);
    int c{2};
    int result;

    a = a + b;
    result = a – c;
    cout << result;

    return 0;
}
```

As you can see, the program will do exactly the same thing that we wrote it to do before, except for the fact that now we have declared the value of the variables as we have declared the variables themselves.

Chapter 3 – OPERATORS

Now that we are familiar in using variables, we can expand on our program and start using what are called “operators”. Operators are basically symbols which tell the compiler (Microsoft Visual Studio), to use mathematics or logic to solve the problems that we have written in code. There are 6 types of operators which C++ will most commonly use. These are:

1. Arithmetic Operators.
2. Relational Operators.
3. Logical Operators.
4. Bitwise Operators.
5. Assignment Operators.
6. Misc Operators.

We will look at only the main and most important types of operators and define what each of the most commonly used symbol in each mean and what they do.

ARITHMETIC OPERATORS

Let’s imagine that we have declared that A equals 10 and B equals 20. We could use the following arithmetic operators to do the following with those variables:

OPERATOR	WHAT IT DOES	EXAMPLE OF USE
+	Adds together	A + B = 30
-	Subtracts #2 from #1	A – B = -10
*	Multiplies the numbers	A * B = 200
/	Divides the numbers	B / A = 2
%	Gives a percentage	B % A = 0
++	Increases integer by one	A++ = 11
—	Decreases integer by one	A— = 9

RELATIONAL OPERATORS

Using the same variables as above, we could also use:

OPERATOR	WHAT IT DOES	EXAMPLE OS USE
!=	If equal, statement is true	(A != B) is true
>	Is left # higher than right?	(A > B) is not true
<	Is left # lower than right?	(A < B) is true

>=	Is left # greater or = to right?	(A >= B) is not true
<=	Is left # lower or = to right?	(A <= B) is true
(A == B)	If not equal, not true	(A == B) is not true

These are the main two types of operators which are used 90% of the time, and it is important that you get to know what each of them mean before you start looking into the more advanced and complex types of operators.

Chapter 4 – CONDITIONS and FUNCTIONS

Conditionals are what gives the program choice to choose what it would like to use. This doesn't mean that the program can think for itself and make decisions like a human being, but rather it makes a choice based only on the options of which you write for it to choose from. The makes programs written in C++ a huge amount of versatility, and stops you from designing a program which only keeps doing the same thing over and over again. Imagine if our "Hello World" program did nothing but keep saying "Hello World"? Or if our variable program did nothing but give us the answer "4" all of the time? These choices are smaller programs within our larger program which we call "functions". What that means is that in order for a program to complete its task, it must run through a series of smaller tasks and choices (conditions) in order to output a result. To put simply, everything that we tell the program to do as a whole is a function.

To look at Conditions, to give our program choice we will use a syntax like the following:

```
if (and then we include our condition here)
{
    insert our program here for the program to work out
}
```

This is basically saying, “If this happens, then do this”. You should be familiar with the basic terms of IF, THEN, AND, OR which is basic mathematics and problem solving skills. However, it is within the brackets where we put our condition, where the program can choose whether or whether not it is going to execute that condition. For example, if we have the three variables of x,y and z, we will first need to declare their values to the program. However, to start including some operators into the equation, lets say that the variable “x” is greater than the variable “y”.

Our code would look like this:

```
int x,y,z;

cin>>x;
cin>>y;

    if(x>y)
    {
            z = x+y;
            cout<<z;
    {
```

Once we declare the values of the variables of x and y, the program will determine if “x” is indeed greater than “y”. If the program finds this to be true, then the undeclared variable “z” will be declared as the sum of x and y and then give us the answer. However, if the program finds that x is not greater than our new variable of z, nothing will happen.

This is where we give our program choice of what it can do when faced with

different scenarios. In this example, we must now tell the program what to do if x does not equal a number greater than the variable z.

Look at the following:

```
int x,y,z;

cin>>x;
cin>>y;

   if(x>y)
            {
            z = x+y;
            cout<<z;
            {
   else if (x<y)
            {
            z=x-y;
            cout<<z;
            }
```

Now we have told the problem that if x does not equal more than z, to go to the next option which tells us to give the answer of the number of the result of x minus y. You can add as many of these conditions as you like which will allow you to create a very versatile program which can be applied in many ways. But, using our example, what if variable x is not more or less than variable y? Nothing would happen. We would still want to program to do *something*, so we could include:

```
//Full Working Program
#include <iostream>
using namespace std;
int main()
{
int x,y,z;
```

```
cin>>x;
cin>>y;

    if(x>y)
                {
                z=x+y;
                cout<<z;
                }
    else if (x<y)
                {
                z=x-y;
                cout<<z;
                }
    else
                {
                cout<<"x is not greater or less than y so they must be equal"<<end1;
                }
return 0;
}
```

Now we have created a program in which if x is not greater or less than the variable y, then the "if" and "else if" conditions will not be executed. The program now has the choice to execute the third condition of "else", since the first two conditions can not be met. We can also use many different types of operators in these strings of conditions. Refer to the previous chapter on Operators to see those.

Chapter 5 – LOOPS

We have looked at basic programming, variables, operators and conditions. Now we must have a look at loops. Loops are a very important part of programming, as they will give our program the ability to keep running the same lines of code as many times as you tell it to. You may think that doing the same task over and over again is pointless and useless, but the meaning behind loops will become very clear to you as we look at the following chapter.

Let us look at the following lines of code from a program.

```
#include <iostream>

using namespace std;

int main()
{
    cout<<”1”<<end1;
    cout<<”2”<<end1;
    cout<<”3”<<end1;
    cout<<”4”<<end1;
```

```
    cout<<”5”<<end1;
    cout<<”6”<<end1;
    cout<<”7”<<end1;
    cout<<”8”<<end1;
    cout<<”9”<<end1;
    cout<<”10”<<end1;

return 0;
}
```

As you can probably see, based on what we have learnt so far, is that this simple program will display for us the numbers from 1 through to 10. You can probably guess that if you were to have to include a lot of these lines in your program, the task would become very boring and monotonous. Can you imagine if you had top design a program which would display the numbers from 1 through to 1000? Luckily though, there is a solution to make our job a whole lot easier. We can create a syntax which will turn the above example into one simple line of code. This syntax will be a loop, and will look like this:

```
for (initialization; condition; increment/decrement)
```

This is basically saying, “for the following, do this”. The initialization is the start of the program (as you should know by now), the condition is whatever choice we give to the program to execute, and lastly, the increment and decrement will represent by how much we would like to add or subtract from our starting number as the program and loop continues to run.

The following is the same example as above, except this time we have included a loop.

```
#include <ostream>

using namespace std;

int main()
```

```
{

int x;

    for (x=0; x<=10; x++)
            {
                    cout<<x;
            }

return 0;
}
```

This program will execute exactly the same as the example above to print numbers, except in this one, we did not need to specify every single number between 1 and 10 for the program to display.

What makes this new set of code more effective, is that we can see that we have used “for (x=0; x<=10; x++)” which is our (initialization; condition; increment/decrement) statement that we looked at above. Here, we have declared x to equal 0 when the program starts, and the program will end when x equals less than or equal to 10. The increment/decrement of “x++” means that the value of x will increase by the number 1 at the end of every loop.

To explain it further, the condition for the loop to run is that the program asks itself, does x equal 10 or less than 10? If yes, 1 is added to x. The program will then ask itself, x is now worth 1, is that equal to or less than 10? Yes, so I will add another 1. Eventually the program will say to itself, “x equals 10, is this equal to or less than 10? Yes it is. I will not make another loop and will now end the program”. This is basically how loops work. The program will have a defined amount of conditions that will keep being checked upon, and if those conditions are passed or met, there will be another loop or repeat of the sequence.

What we have looked at is called a “for” loop. There are two other types of loops that we will also look at. The first new loop that we will look at is called a “while” loop. A while loop will allow your program to be more versatile in some circumstances, and the basic syntax looks something like this:

```
while (condition)
{
    statements
}
```

This basically says, “while these conditions are true or false, I will run the loops which have been set out for me in the statements that have been made”.

What would our “for” loop program look like with “while” conditions instead?

```
//Full Working Program

#include <iostream>

using namespace std;

int main()
{

int x=0;

    while (x<=10)
    {
            cout <<x<<end1;
            x++;
    }

return 0;
}
```

Our program will perform the same task as before, except this time it will use a “while” instead of “for” conditions for its loops. In this example though, we have declared the value of x to be 0 at the start of the program instead of writing it in our

statement section. This is because a while loop is only focused on the conditions that have been set for it, and the incrementing of the number between loops. We could also simplify the while loop by writing the following:

```
while (x<=10)
{
    cout<<x++<<end1;
}
```

This is saying that while x equals less than or is equal to the number 10, it will add one number. The only difference here is that x has been declared at the start of the program.

The third type of loop that we can look at is called a “do while” loop. It will perform the same task, but the condition is placed at the end of the loop, so the loop will not be checked to see if it should be performed until it runs for the very first time. For example:

```
//Full Working Program
#include <iostream>
using namespace std;

int main()
{
int x=0;
    do
    {
                cout<<x<<end1;
                x++;
    }while(x<=10);

return 0;
}
```

This is saying that 1 will be added to x while x equals less than or equal to 10. Did you notice the scope bracket (semicolon) at the end of the while loop? This means that we are closing this line off as a function for the program to execute.

Chapter 6 – ARRAYS

Arrays are very similar to variables in that they allow us to store information for later use. Arrays allow us to store a collection of data under a single name, which means that we have a greater level of control in organizing our data which becomes very useful especially dealing with large programs with thousands of lines of code.

To give you an example of how a set of data could be stored in an array, imagine that you wanted to design a program which would display the age of 5 people. You would need to declare the name of the variable and the value of the variable, which would look something like this:

```
int age1=15;
int age2=24;
int age3=54;
int age4=33;
int age5=120;
```

We are now faced with the same problem as before in the previous chapter where we wanted to print the numbers from 1 through to 10. Imagine if you had to design a program to display a list of the ages of 500 people? Luckily though, just as we saw in the previous chapter, there is a way to display this information without having to manually type in every single line of code.

The syntax for an array looks like this:

datatype name [index];

This means that we want to list a series of stored values much like a variable. In order to create an array which will store something similar, and in this example the ages of our 5 people, we would write:

```
int ages [5];
```

This means that we have a list of 5 things which the program knows only to be something called “ages” at this point. Just like declaring a value for the letter “x”, we must also declare values for this new variable we have called “ages”. We can do this by:

```
int ages [5];

ages[0]=15;
ages[1]=24;
ages[2]=54;
ages[3]=33;
ages[4]=120;
```

Although I mentioned that there was a much easier way to write this index, the above looks exactly the same. The reason for this is that I want to show you how we declare this new datatype of ages. When we start the program, we have told it that when it starts there is an array of [5] items. The reason that our list of declarations start at 0, is because this is how computers read a list of numbers. A 0 will always mean first.

However, if we want to write the above in a much more simpler way, we can write the following:

```
int ages [5];

ages = {15,24,54,33,120};
```

This is telling the program the same as above, in that 0 equals 15, 1 equals 24, and so on. What does this look like if we were to write it out completely in code?

```
#include <iostream>
using namespace std;

int main()
{
    int ages [5];
    ages={15,24,54,33,120};
    cout<<ages<<end1;

return 0;
}
```

Can you spot the error in the above code? Most would think that we could write this program the same way as our previous examples, in that we define a set of variables, and then have the program list those when it is executed. However, this is not the case when dealing with arrays. The problem in this code is that we have ended the program with “cout<<ages<<end1;” after we have defined the array. The problem with this is that C++ cannot print a whole array, since you must tell your compiler (Microsoft Visual Studio) what array to display. For example, if we wanted to display the age 120, we would need to tell it to display age [4]. Then we would need to list every variable as a list like as follows:

```
int main()
{
    int ages [5];
    ages={15,24,54,33,120};
    cout<<ages[0]<<end1;
    cout<<ages[1]<<end1;
    cout<<ages[2]<<end1;
    cout<<ages[3]<<end1;
    cout<<ages[4]<<end1;

return 0;
}
```

But I thought that you said that there was a away to store a whole set of integers into an array? There is, and the way that we do that is to declare a variable, say “k” and declare to the program that it is storing integers to be used.

For example:

```
//Full working program
#include <iostream>

using namespace std;

int main ()
{
    int k;
    int ages [5];
    ages = {15,24,54,33,120};

            for (k=0; k<5; k++)
            {
                    cout<<ages[k]<<end1;
            }

return 0:
}
```

Notice how we have defined that “k” is a variable and was declared to hold the entire string above? Basically, “k” was declared as 0 which will execute the array as long as “k” remains less than 5 (not equal to in this case if you notice the different operator used). The program is basically saying when it will display the ages until the datatype “age” reaches 5, which is then when the program will stop displaying the ages.

Chapter 7 – POINTERS

Pointers are something that we have not yet looked at in our previous examples. The reason that they have been left for now is for the fact that it can be sometimes very difficult to understand exactly what they are for. This does not mean that you do not need to learn about how they work and how to use them because they are still a very important part of the structure of more complex programs.

Let's start looking at them by looking at a syntax of how we would declare a pointer:

```
double *p;
```

In this statement, we are saying that we have a pointer called "p" which can have an an address or location of floating point values. The word "double" is our new datatype, and the reason that there is an * before p is because this is how we declare a pointer.

In code, we would declare an address which a pointer points to by writing the following:

```
p=&x;
```

The letter "p" will now store the location of "x" (which is a variable in this case). In simpler terms, p is pointing to x. We use "&" before the variable in order to assign it an address (but not a value). P will point to x, but x will not have a value. To give x a value, we will need to make changes to p, and can write code to do so like this:

```
*p=25.3;
```

This now means that x equals 25.3 by making p a pointer, and by pointing p to x. Since p holds x, x is now valued at 25.3 (whatever value p is). If we were to write a complete program which uses a pointer and assigns it a value, we could write the following:

```
//Full working program
#include <iostream>
#include <stdlib.h>

using namespace std;

int main()
{
double x;
double *p;
p=&x;
*p=25.3
    cout<<x<<end1;

system (“pause”);
return 0;
}
```

This program points p at x and tells the program that x is valued at 25.3. However, this does not really seem that pointers are very useful. We could simply just use variables and define them (and perhaps use some conditional loops) in order to make our program work. What they are useful for though, is that they give you the ability to manipulate the information and code simply by pointing a variable at another variable. This is where they do become useful.

Let’s say for example that you have written several thousand lines of code. Now, imagine if all of a sudden you need to change the value of 50 of those variables within the code? It would take a very long time to find all of those values within the lines of code, and would perhaps even mean that you would have to write the code all over. If you accidentally change something in the process when you should not have, the program may be damaged.

Luckily, this is where pointers actually have a useful and important function. We can use pointers to change the value of variables which already exist in the program, or we can point and declare the variables at the start of writing the program. Pointers

allow us to manipulate a function, whereas this is impossible with a normal variable.

This is all that you need to know about pointers at a beginner level.

Chapter 8 – DYNAMIC MEMORY

Everything, even lines of code, need storage space to be stored and to be executed. For example, when you are copying a song or a photo from your computer to your phone or vice versa, there will be a program which will ask the phone or the computer if there is enough space before the file is moved. This is because if there is not enough storage space to store the new file, it is pointless even trying to move it. The program will determine before the process is executed if there is enough storage space for the operation to be executed.

However, there are cases when the program will not know how much memory or storage is needed until the program is executed, and this amount of memory can constantly change. This is why when you write any C++ program, you need to tell it to allocate a certain amount of memory or storage to accommodate for this (and this can sometimes be unknown). This second type of memory is what we will be focusing on here, since it is easy to determine memory needs of storage before a task is executed.

Every time the program is executed, the allocated memory needs can be referred to as the “new” operator. But, as soon as the memory allocation is no longer needed, we refer to “delete” operator. The syntax for this looks like:

```
new datatype;
```

This basically means that we are defining a new operator, followed by whatever it is that we define afterwards. As an example, we can write some code where we will use a pointer to allocate memory as soon as the program is executed:

```
double* pvalue = NULL;
pvalue = new double;
```

However, these lines of code will not tell us if memory has been successfully allocated – it is only a command to allocate. In that case, to have the program tell us that memory has been successfully allocated, we would write:

```
double* pvalue = NULL;
if( !(pvalue = new double ))
{
    cout << “Error: out of memory.” <<end1;
    exit(1);
}
```

The program will now say to us, “Error: out of memory”, should there not be enough available memory to allocate to the program. Once we reach a point in our program where the expected memory requirements are deemed to not require allocated memory, we delete operator. For example:

```
delete pvalue;
```

Simple as that. Now, let's have a look at what a program would like like that first allocated memory using a new operator, executed a pointer to the allocation of that memory, and then de-allocated that memory with delete operator:

```
#include <iostream>
using namespace std;

int main ()
{
    double* pvalue = NULL;
    pvalue = new double;

    *pvalue = 25677.99;
    cout << "value of pvalue : " <<*pvalue << endl;

    delete pvalue;

    return 0;
}
```

Running this code would show us:

```
Value of pvalue: 25678
```

It is quite simple to input new and delete operators, and definitely much easier than the other things that we have learnt in this book so far. But let's have a further look at other ways we can allocate memory to our program. For example, let's assume that we want to make a program that will display to us a line of 100 characters. Based on the examples above, the code would look like this:

```
char* pvalue = NULL;
pvalue = new char[100];
```

This is saying that we declare our pointer to be NULL, and that we have executed a new request for memory to be allocated for our 100 characters. Once we do no longer need this memory to be allocated, we insert a delete operator:

```
delete [] pvalue;
```

Now we have told our program to delete our array which we have pointed to in the previous lines of code. But, what if we wanted to allocate memory for multiple arrays at the same time? We do not need to start a new operator for each one. Instead, we can write the following:

```
double ** pvalue = NULL;
pvalue = new double [4] [5];
```

Here, we first declare that our pointer is valued as NULL, and then we tell our program that we want to allocated memory for an array with the dimensions of 4x5. Deleting is the same as before:

```
delete [] pvalue;
```

Chapter 9 – CLASSES AND OBJECTS

Now that we have learnt the basics of programming in C++, it is time to have a look

at something slightly more advanced. Classes are part of what we call “object orientated” programming. This allows us to use programs in everyday life by using objects. Objects can be defined as anything that we use in everyday life – such as cars, computers, mobile phones, and so on. As we should have learnt by now, a program is made up of very small programs within it, just the same as a car is made up of very small parts which work together in order for the car to operate. Some of these components can be changed, like turning the lights on and off in the car, whereas others cannot be changed, such as the central computer of the car. These can be differentiated by what we call “private” and “public” variables, or in other words, what you are allowed to change, and what you are not (but we will look at that in a moment).

Classes exist on a slightly more advanced level when learning C++ because they are more involved. Learning them however is a great reward, as it will allow us to create and develop programs which we can apply for use in everyday life. They also allow use to reuse code and apply it to different scenarios. For example, every player in a football team with have a name, age and an income, but each of these will be different for every player. In this case, the code would create the attribute of class which is simply the name, age and income for each player.

To write this as an example, our code for our player information would look like this:

```
Class Player
{
    string name;
    string age;
    double income;
};
```

Our player Class now has structure. From now on, every time that we declare a value to each of the variables of the individual player, we will be creating a new record, or a new object that exists within Class Player. Every object will consist of a name, age and income. In order to declare the variables of each Player type, we use the same method as we would declaring any other variable:

```
Player player1;
Player player2;
Player player 3;
```

Once we have created a Class as seen in the first example, we can use it like any other example of data type that we have previously used. However, we want to also be able to limit what the program is able to access and manipulate within these class structures. Some objects or classes must remain the same at all times (private class), where some can be changed (public). For example, the class which contains only name, age and income must remain private, as this is a constant. On the other hand, the variables that we declare for each individual is public, because that information is not constant.

To see this in code, we would write the following:

```
Class Player
{
private:
    string name;
    string age;
    double income;

public:
    Player();
};
```

Executing the above lines of code would prohibit the program from being able to make changes to the private class of name, age and income, but would allow it to make changes to the public class of individual player information. We use ":" after declaring private and public, because they tell the program that everything afterwards is considered that class type.

When we actually input and manipulate public data, we use what are called "constructors" and "destructors". Every time that we declare a class object, a constructor is executed to perform whatever it has been programmed to do. To initiate a constructor, we would write the following code:

```
Player();
Player(string,int,double);
```

This means that we have told the constructor to use our three parameters of name, age and income. We also have a Player class with no value on the top line, which gives the constructor the option to assign or not to assign parameters to the class type. But, we still need to tell the constructors what to do, so we would write:

```
Player()
{
    name = “”;
    age = 0;
    income = 0;
}

Player(string n, int a, double s)
{
    name = n;
    age = a;
    income = i;
}
```

The constructor that is executed first has nothing to do apart for setting everything to the value of 0. However, the constructor which is executed second must use our three parameters of name, age and income. Then, all that is left for us to do is to actually declare each Player as required.

```
Player player1;
Player player2(“David”,33,95.20);
```

Much more simple to understand when looking at what a constructor does when we see it in code. Basically, they create a class or an entry of information based on what we give it. It does the work for us to a degree. However, we also need to look at destructors and what they achieve within our program.

A destructor is the opposite to a constructor in that they will remove instead of create data. The syntax used to execute a destructor is ~ followed by the name of your class. For example:

~Player();

And that is the basics of classes and objects within C++ programming. There are of course a lot more which we could look at, but that scope goes beyond the realm of the beginner level.

Conclusion

In summary, by reading this book you should now have the required basic skills required to understand the basics of the C++ programming language, and should now be able to program your own simply C++ programs with ease. Just remember that although on the outside every program looks complex and filled with many lines of code, all that it is simply is a bunch of smaller programs which make up a whole. Just focus on working on those smaller programs one at a time and you will be fine.

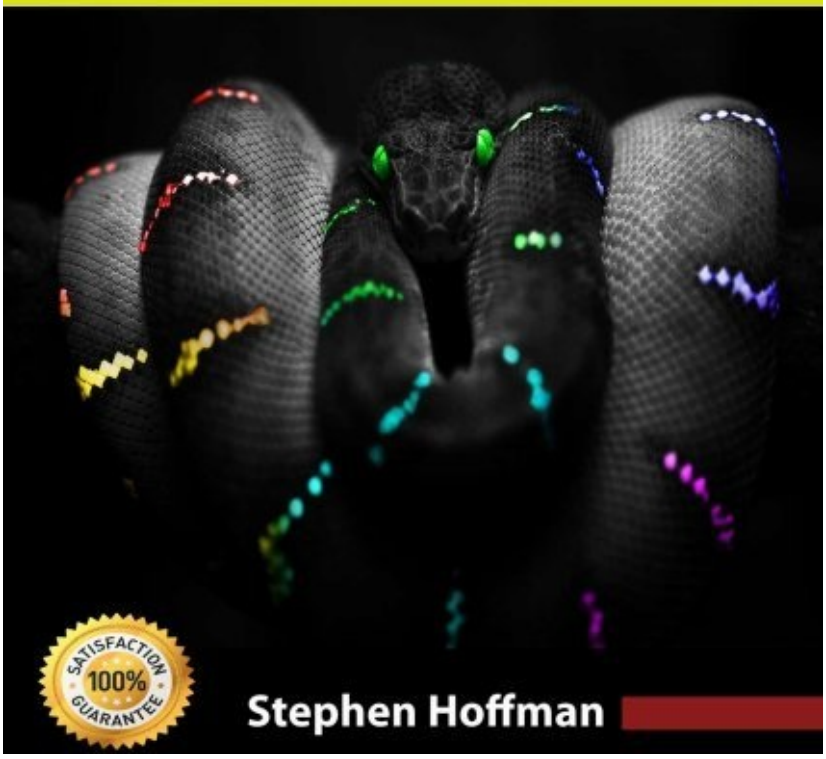
This book has also shown you the many different types of code that we can use when making a program in C++, and this knowledge will allow you to be able to impress your family and friends with simple programming skills, or perhaps allow you to make basic programs to use in your every day life. You should also now have the basics required to move on to more advanced topics on your way to become an advanced or professional C++ programmer.

Use this book as a guide or as a reference as much as possible on your journey through the wonderful and exciting world of computer programming, and always remember that C++ is easy to learn once you know the basics.

PYTHON

LEARN PYTHON FAST

The Ultimate Crash Course to Learning the Basics of the Python Programming Language In No Time



Stephen Hoffman

Python

Learn Python FAST - The Ultimate Crash Course to Learning the Basics of the Python Programming Language In No Time

STEPHEN HOFFMAN

CONTENTS

Introduction

Python can be used for a wide variety of projects, and there's a reason many programmers will reach for Python before they reach for any other programming language. Python can be used for web applications, automating tasks on systems, finding colors in images, and so many other different programming tasks!

Python is a general-purpose programming language that was created in the late 80's, and it was named after the infamous Monty Python. It's used by thousands of programmers to do anything from testing microchips at Intel to building video games with a PyGame library. It's a small language that resembles the English language and has hundreds of third-party libraries that already exist.

There are two main reasons as to why programmers will choose Python over many different programming languages.

The first reason is readability. Because it's so close to the English language, using words such as 'not' and 'in' make it so that you can read a script out loud and not feel like you're reading some long-forgotten language. This is also helped by the strict punctuation rules in Python that mean you don't have curly braces all over the code.

In addition, Python has some set rules, known as PEP 8, that tell Python developers how to format their code. This means you will always know where to put new lines and every code that you pick up from someone else will look similar and be just as easy as yours to read. That makes for some very easy collaboration between developers!

The second reason is that there are preexisting libraries for almost anything you want to do in Python. The language has been around for over twenty years, so there are many programmers out there who have mastered the language and know how to write what you want. So if you find yourself in doubt, try to find someone else who has already done it!

So the main reasons why you should learn Python are that it’s simple and it’s easy for you to collaborate with others in order to hone your skills. So let’s get started!

Chapter One – Setting up Python

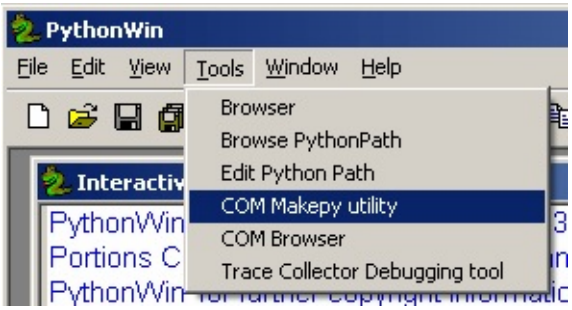
When it comes to programming in python, the first thing you should know is that python is already installed on many of the operating systems. Most operating systems, other than Windows, already come with Python installed by default. If you want to check to be sure it's installed on your computer, open the command line by running the terminal program, and type in 'python –V'.

If you see version information, then Python is already installed on your computer. However, if you get the 'bash: python: command not found' prompt, then there is a possibility that it's not installed on yours.

Follow the instructions for downloading and installing ActivePython on your computer.

Installation

The first step is to go to the Python website, www.activestate.com/Products/ActivePython, this will take you to a website where you can choose the operating system you have. If you have a Windows version that's older than Windows XP, you'll need to install the Windows Installer 2.0 before you continue.



Once you find the installer you need, double click on it and go through the installation steps. After you've installed it, go to your Start-Programs-ActiveState ActivePython (versions #)-PythonWin IDE and you'll see a little script telling you what version you have and what operating system it's installed on.

Interpreter

Once you start the interactive mode with Python, it'll prompt you for the next command. This is shown with three greater than symbols, >>>. If you want to run python interpreter interactively, then you need to type in 'python' in the terminal.

It'll look something like this:

How to Run Programs

So once Python is all set up and you know how to use the interpreter, how are you going to run programs in python? The simplest way to run a program is to type ‘python helloworld.py’. If your executable is set on a .py, then the file can be run by name without having to type in the python first.

Now, set the execute bit with a ‘chmod’ command, such as this:

- $ chmod +x hello.py

Now you’re able to run your program as ./hello.py.

Text Editor

Before you’re able to write a python program in a source file, you need an editor to write that source file. When it comes to programming python, there are many editors you can choose from. Just pick one that will work with your platform. The editor you choose is going to depend on your experience with computers, what you have to do, and what platform you need in order to do it.

If you’re a beginner, choose something like Notepad ++ or TextEdit.

Beginning Writing

Writing and running a python program is just a text file that is being edited by you directly. In the command line or your terminal, you can run any number of programs you want, just as you did with the aforementioned program of ‘python helloworld.py’.

When the command line prompt comes up, just hit the up-arrow key in order to recall the commands that were typed previously, that way it’s easy to run a previous command without retyping it.

To try out the editor you have, edit the helloworld.py program. Instead of using the word ‘Hello’, use the word ‘Greetings’. Save the edits and run the program again to see its output. Then try adding a ‘print yay!’ just below the print that exists with the same indentation.

Run the program and see what the edits look like.

Congratulations! You just edited your first Python program! Now that you know how to edit a program let’s take a look at variables you can use in your programs.

Chapter Two – Variables

Variables the characters you're able to use in Python. You can use any letter, every number, and every special character, but you cannot start with a number. White spaces and the signs with the special meanings like + and – are not allowed.

Either capital or lowercase letters are allowed, but most prefer to use lowercase letters in order to keep it all uniform. Variables names are case sensitive, and python is typed dynamically, so that means you don't need to declare what type every variable is. In python, the variables are a storage placeholder for texts and numbers, so it has to have a name in order to find it again.

The variable is assigned with an equal sign, followed by the value of your variable. There are a few reserved words in python that you cannot use for your variable. It's also important to know that variables can be changed later on if you need to.

For example, you can use these variables for numbers and create this formula:

- foo = 10
- bar = foo + 10

Here are a few different variable types to get you started.

# integer	X = 123
# long integer	X = 123L
# double float	X = 3.14
# string	X = "hello"
# list	X = [0,1,2]

# tuple	X = (0,1,2)
# file	X = open(‘hello.py’, ‘r’)

You’re also able to assign a single value to several variables at the same time with multiple assignments. Variable a, b, and c are allocated to the identical memory location with the value of 1; a = b = c = 1.

For example:

Now that you know how to use variables let’s take a look at the interpreter in more detail.

Chapter Three – Interpreter

The interpret is usually installed at the /usr/local/bin/python location on machines where it's available. Putting /usr/local/bin in the Unix shell's search path is going to make it able to start it by typing the command: python to the shell.

When you begin the python interactive mode, it will prompt for the following command using three greater than symbols, >>>. It will then print a welcome message that states its version number and the copyright notice before it prints the first prompt.

Interactively

When used interactively, it's a good idea to have some standard comments executed each time the interpreter is begun. You can do this when you set an environment variable called PYTHONSTARTUP in the title of a file that contains start-up commands. This is like the .profile feature of the Unix shells. To do this, add this line to the .bashrc file:

>> export PYTHONSTARTUP=$HOME/.pythonstartup

Create or change the .pythonstartup file and place the python code in it, like this:

```
import os
os.system('ls -l')
```

To exit your interactive prompt, hit Ctrl+D if you're using a Linux machine.

Chapter Four – Importance of Comments

The comments in python are strictly for the person reading the code, whether it's you or someone else. It's a good way to communicate with groups of people what the code is supposed to do so that they're able to collaborate better. However, there are two ways you can add comments to python that are universally accepted in the programming industry.

A single line comment is going to start with the # symbol and is terminated by the end of the line. Python will ignore all text that comes after the # symbol to the end of that line because it views that as not part of the command.

Comments that will span more than a single line is achieved by inserting a mutli-line string with "''" as the delimiter on either end. They are meant as documentation for anyone who is reading the code.

Let's take a look at two examples so that you can see what I'm talking about.

So now you know how to collaborate with others and just keep your code organized with comments for yourself, let's take a look at python docstrings.

Chapter Five - Python Docstrings

Docstrings, or python documentation strings, provide a nice way to associate documentation with python functions, modules, methods, and classes. An object's docstring is made up of a string constant as the first statement in the definition of the object.

It's specified in the source code that's used to record a specific piece of code. Unlike regular source code comments, the docstring should be a description of what the function does rather than how. All functions need to have a docstring.

This lets the program inspect the comments during run time, for example, as an interactive help system or as metadata. You can use the _doc_ attribute on objects to access the docstring.

What It Should Look Like

The docstringline is going to begin with a capital letter and end with a period. The first line is going to be a short description. Don't write the name of your object. If there's more than one line in the documentation string, the second one should be blank so that it visually separates the summary from the remainder of the description. The rest of thelinesshould be one or more paragraphs that describe the object's calling conventions and anything else that's important.

An example of a docstring is as follows:

And this is what it would like when it was printed.

Declaration

This python file shows you a declaration of a docstring in a python source file.

```
"""
Assuming this is file mymodule.py, then this string, being the
first statement in the file, will become the "mymodule" module's
docstring when the file is imported.
"""

class MyClass(object):
    """The class's docstring"""

    def my_method(self):
        """The method's docstring"""

def my_function():
    """The function's docstring"""
```

Accessing the Docstring

The following session will show you how the docstring can be accessed:

```
>>> import mymodule
>>> help(mymodule)
```

Assuming this is the file mymodule.py, then the string that is the primary declaration in the file will be the mymoduleelement docstring when the file is introduced.

```
>>> help(mymodule.MyClass)
The class's docstring

>>> help(mymodule.MyClass.my_method)
The method's docstring

>>> help(mymodule.my_function)
The function's docstring
```

So now that you know what a docstring is let’s talk more about some keywords in python that you need to know.

Chapter Six - Keywords in Python

Keywords in python are reserved words that are not able to be used ordinary identifiers. They have to be spelled exactly as you see them written. If you want to print the keyword list in python, then use the following commands.

Definitions of Keywords

print

- print to console

while

- controls the movement of the platform

for

- repeat over parts of a collection in the order they appear

break

- break the loop cycle

continue

- made to interrupt the present cycle without getting out of the whole cycle. A new cycle will start.

If

- Used to figure out which statement is going to be executed

Elif

- Is and abbreviation of else if; if the first test evaluates to false, then it will continue to the next one.

Else

- This means the command is optional. The statement following the else word is executed unless the condition is true.

Is

- Looks for object identity

Not

- Disproves a Boolean value

and

- all circumstances in a boolean expression must happen

or

- at the minimum one condition must happen

import

- bring in other elements into a Python script

as

- gives a module a different alias

from

- for introducing an exact variable, class or a function from an element

def

- makes a new user-defined function

return

- gets out of the function and yields a value

lambda

- makes a new unspecified function

global

- accesses variables defined exterior to functions

try

- states exemption handlers

except

- holds the exception and completes codes

finally

- is constantly completed in the end and is needed to clean up resources

raise

- make a user-defined exclusion

del

- deletes objects

pass

- does nothing

assert

- used for restoring purposes

class

- used to create new user defined objects

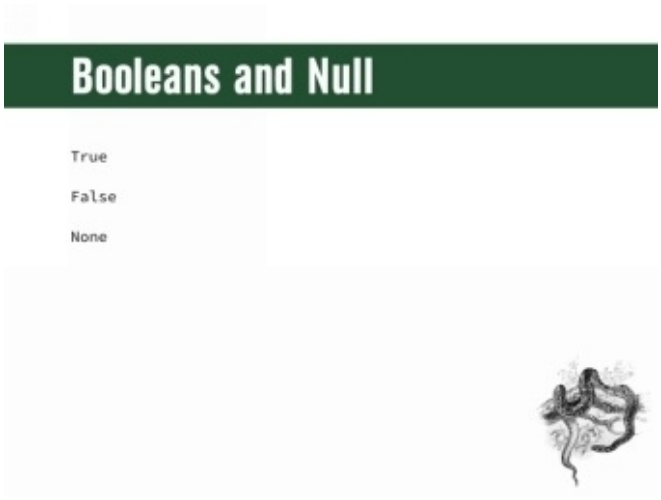
exec

- executes Python code dynamically

yield

- is used with generators

Chapter Seven - Booleans, True or False in Python



These values are the two continuous objects of true and false. They're used to symbolize truth values, which are other values that can be considered true or false. In a numeric context, like if they're used as an argument to a mathematic operator, they will behave like the integers zero and one respectively.

The preexisting function bool() is able to be used to create any value to a Boolean if the value is able to be interpreted as a truthvalue. They're written as false and true.

Boolean Strings

A string can be tested for truth value. The return type is going to be in Boolean value or True or False. So let's take a look at an example. First, create a new variable and assign it a value.

```
numberone_string = "Hello World"
numberone_string.isalnum()                  #check if all char are numbers
numberone_string.isalpha()                  #check if all char in the string are alphabetic
numberone_string.isdigit()                  #test if string has digits
numberone_string.istitle()                  #test to see if the string has title words
numberone_string.isupper()                  # test to see if the string has upper case letters
numberone_string.islower()                  #test to see if the string has lower case letters
numberone_string.isspace()                  #test to see if the string has spaces
numberone_string.endswith('d')          #test if the string ends with a 'd'
numberone_string.startswith('H')          #test if the string begins with 'H'
```

To see what the return value would be, print it out.

numberone_string=”Hello World”

```
print numberone_string.isalnum()          #false
print numberone_string.isalpha()          #false
print numberone_string.isdigit()                  #false
print numberone_string.istitle()                  #true
print numberone_string.isupper()          #false
print numberone_string.islower()          #false
print numberone_string.isspace()          #false
print numberone_string.endswith(‘d’)          #true
print numberone_string.startswith(‘H’)          #true
```

Logical Operator Boolean

These values tend to be respond to logical operators and/or

```
>>>True and False
    False
>>>True and True
    True
>>>False and True
    False
>>>False or True
    True
>>>False or False
    False
```

Remember the built-in type of Boolean can have only one or two possible answers: True or False.

Now that you know a little more about Booleans and how to use them let’s talk about python operators.

Chapter Eight - Python Operators

There are a few different types of operators. There's arithmetic, comparison, and logical operators. Let's take a look at each one in a little more detail.

Arithmetic Operators

Python has the modulus -, +,*, /, % and ** operators. Assume that the variable 'a' has a value of ten and variable 'b' has a value of twenty. Therefore:

- +
    - a+b=30
- -
    - a-b=-10
- *
    - a*b=200
- /
    - b/a=2
- %
    - b%a=0
- **
    - a**b=10^{20}
- //
    - 9//2=4

Comparison Operators

The plain comparison operators like ==, <, >=, and the others are utilized on all different manner of values. Strings, numbers, mappings, and sequences are all able to be compared. The following bullets will show you a list of comparison operators.

- < is less than
- == equal
- <= less than or equal to
- >= greater than or equal to
- > greater than
- <> not equal
- != not equal

Logical Operators

The logical operators and or also give back a Boolean value when they're used in a decision structure. There are three logical operators, or, and, and not.

For example, x>0 and x<10 is only true if x is greater than 0 and less than 10.

Operator Description:

- and: logical AND
- or: logical OR
- not: logical NOT

Now that you know about operators let's talk more about using math in python.

Chapter Nine - Using Math in Python

The Python language has the Python interpreter, which is a simple development environment known as IDLE, tools, libraries, and documentation. It’s preinstalled on almost all Linux machines and Mac systems, but it can be an older version.

In order to begin using the calculator in the Python interpreter, type in the following script:

Counting With Variables

To put in values in the variables to count the area of a rectangle, try this code out:

Counter

Counters are needed tools in programming in order to increase or decrease a value every time it is run. The following code is a counter:

Counting with a While Loop

And here's an example of when it's useful to use a counter.

The above program will count from zero to four. Between the world while and the colon, there's an expression that is going to be true at first and then become false. As long as that expression is true, the following code is going to run. The code that has to be run has to be indented. The last declaration is a counter that adds one to the value for every time the loop runs.

Multiplication Table

To make a multiplication table in Python, use the following code:

```
table = 8
start = 1
max = 10
print "-" * 20
print "The table of 8"
print "-" * 20
i = start
while i <= max:
  result = i * table
  print i, " * ", table, " =" , result
  i = i + 1
print "-" * 20
print "Done counting..."
print "-" * 20
```

The output for this code is going to be as follows:

>>Output:

The table of 8

1*8=8

2*8=16

3*8=24

4*8=32

5*8=40

6*8=48

7*8=56

8*8=64

8*9=72

8*10=80

Done counting.

And that’s how you use math in Python. Now let’s look at exception handling in Python in the following chapter.

Chapter Ten - Exception Handling in Python

Before you get into what you can do with exceptions, you first have to know what they are. An exception is an error that occurs while a program is running. When that error happens, Python will generate an exception that can be used in order to avoid crashing the program.

Exceptions are convenient for handling special conditions and errors in programs. When you believe you have a code that is able to produce an error, then you can put in an exception handle. You can raise this exception on your own by with the raise exception declaration. Rising an exception will break the current code execution and return the exception to the previous code until it can be handled.

Here are some common errors you might see in your program

- IOError – the file is unable to be opened.
- ImportError – Python is not able to find the module.
- ValueError – this is brought up when a built-in operation or function obtains an argument that might have the right type but not the right value.
- KeyboardInterupt – This is brought up when the user will hit the interrupt key on the keyboard, this is usually the Cntrl – C or Del keys.
- EOFError – this comes up when one of the built-in functions, input() or raw_input() gets to an end-of-file conditions or EOF without scanning the data.

Set Up Exception Handling Blocks

In order to use exception handling in Python, first you have to have a catch-all except clause. The words ‘try’ and ‘except’ are Python keywords that programmers use in order to catch exceptions.

For example:

try – except [exception-name] (insert exception) blocks

The code in the try clause is going to be executed statement by statement. If an exception happens, the rest of the block is going to be skipped and the except clause is going to be run.

```
Try:
        some statements here
Except:
        exception handling
```

So let’s see an example of this.

```
try:
    print 1/0
except zerodividionerror:
    print “You can’t divide by zero.”
```

How does it work?

Error handling is handled through the use of exceptions that are made in try blocks and handled in except blocks. If your code encounters an error, a try block code execution is going to stop and transfer down to the except block.

In addition to using the except block after the try block, you're also able to use the finally block. The code in this block is going to be executed whether or not the exception happens.

Let's take a look at some code to see what will happen when you do not use error handling in a program.

Code Example

This program is going to ask the user to input a number between one and ten and then print that number.

number = int(raw_input("Enter number between 1-10"))

print "you entered number", number

This program will work just fine as long as the user enters the number, but what happens if they put in something else, like a string?

Enter a number between 1-10

hello

You can see that the program will throw an error when the string is entered.

Traceback (most recent call last):

```
File "enter_number.py", line 1, in
    number = int(raw_input("Enter number between 1-10\n"))
valueerror: invalid literal for int() with base 10: 'hello'
```

Thevalueerror is a type of exception. Now let's see how you can utilize exemption handling to repair the aforementioned program.

import sys

print "Let's fix the preceding code with exception handling"

try:

    number = int(raw_input("Enter a number between 1-10 \n"))

except valueerror:

```
    print “Only numbers are accepted.”
sys.exit()

print “you entered number \n”, number
```

If you now start the program and input a string in lieu of amounts, you can see that you’ll get a different response. So now, let’s run the program to be sure it worked. The response should be:

Enter number between 1-10

hello

Only numbers are accepted.

Try … Except … Else Clause

The else clause in a try, except statement has to follow all except clauses and is used for code that has to be executed if the try clause will not raise an exception. For example:

```
try:
    data = something_that_might_go_wrong
except IOError:
    handling_the_exception_error
else:
    trying_a_different_exception_handling
```

Exceptions in an else clause will not be handled by the preceding exception clauses. Be sure that the else clause is run before the final block.

Try … Finally Clause

Thefinallyclause is an optional clause. It's used to define clean-up actions that have to be executed under all circumstances. For example:

```
try:
    raise KeyboardInterrupt
finally:
    print 'Goodbye!'
…
    Goodbye!
    KeyboardInterrupt
```

A final clause will be executed before leaving the try statement, regardless of whether an exception has happened.

Remember that if you don't specify the exception type on the exception line, it's going to catch all exceptions. This is not a good thing because it means the program will ignore any unexpected errors, as well as errors the exception block is able to handle.

Now that you know about exceptions in Python let's take a look at Strings Built-In Methods in the following chapter.

Chapter Eleven - Strings Built-In Methods

This chapter is just going to contain a list of built-in methods you can use for each string. Let's first make a string with the value of 'Hello'.

It'll look a lot like this:

string = 'Hello'

In order to manipulate this string, take a look at some of the following built-in methods and their explanations.

- string.upper()
    - This makes all the letters in the string uppercase, so if the string were 'hello jane', it would come out as 'Hello Jane'.
- string.lower()
    - This will make all the letters in your string lowercase, so if the string reads 'Hello Jane', it will come out as 'hello jane' in the program.
- string.capitalize()
    - This will capitalize only the first letter. Therefore, if the string was 'hello jane', it would read as 'Hello jane' in the program.
- string.title()
    - This would capitalize the first letter of the words. So if the string were 'hello jane' in the program, it would read 'Hello Jane'.
- string.swapcase()
    - This string will convert all uppercase letters to lowercase and vice versa, so the following string, 'hello jane' would read 'HELLO JANE' in the program.
- string.strip()
    - This string removes all white space from the string. So if the string was 'Hello Jane', it would read 'HelloJane' in the program.

- string.lstrip()
    - This string removes all whitespace from the left. Therefore, if the string were ‘ Hello Jane ‘ with space before hello and one after Jane, then the code would spit out ‘HelloJane’.
- string.rstrip()
    - This string will remove all whitespace from the right. Therefore, if the same line of code were to read ‘Hello Jane ‘ with a space after Jane, the space after Jane would be removed first to read ‘HelloJane’.
- string.split()
    - This command will split words. So if the code read ‘HelloJane’, then it’ll be split into ‘Hello Jane’.
- string.split(‘,’)
    - This means the code will split words by a comma, so the following string ‘Hello Jane’ would come out ‘Hello, Jane’.
- string.count(‘l’)
    - This string will count the amount of times l, or any other letter you choose to put into the string, occur in the string. So the input would be ‘Hello Jane’ and the program would tell you 2.
- string.find(Ja)
    - This string will find the word ‘Ja’ in the string.
- string.index(“Ja”)
    - This will find the letters Ja in the string.
- “:”.join(string)
    - This will add a : between every character. So ‘Hello Jane’ would come out ‘H:e:l:l:o:J:a:n:e’
- ““.join(string)
    - This will add white space between every character, so ‘Hello Jane’ would read ‘H e l l o J a n e’
- len(string)
    - This will find the length of the string in characters, so if the string is ‘Hello Jane’ then the program would say ten.

So now that you know some of the basic strings that are built-in to the Python code, let’s take a look at list examples in the following chapter.

Chapter Twelve – Lists

Lists are created with square brackets, [], and the elements that are between those brackets are what the program will use. The elements in a list do not have to be the same type, ergo, you can have letters, numbers, and strings between just one set of brackets.

Let’s take a look at an example.

So what does this list do? The script is going to create a list, labeled (list1), with the values of 1, 2, 3, 5, 8, 2, 5, 2. The while loop is going to print each part of the list. Each part of the list list1 is going to be obtained by the index or the letter in the square brackets.

The len function will be used to go through the entire length of the list. And the final line tells the program to increase each variable by one for every time the loop runs.

So the output for this program is going to be: 1 2 3 5 8 2 5.2

Let’s take a look at another example.

This example is going to count the average value of the elements that are in the list.

```
list1 = [1,2,3,5,8,2,5.2]
total = 0
i = 0
while i < len(list1):
    total = total + list1[i]
    i = i + 1


average = total / len(list1)
print average
```

The output for this will be:

#Output >> 3.74285714286

As you can see, lists are pretty easy to create and manipulate. Let’s take a look at using dictionaries in Python.

Chapter Thirteen - How to Use Dictionaries in Python

You can use the curly brackets, {}, to make a dictionary in Python. You can use the square brackets, [], to index the dictionary. Separate the key and the value with some colons and with commas between the pairs. Keys have to be quoted just like with lists, and you can print out the dictionary by printing the reference to the dictionary.

A dictionary will map a set of objects (keys) to another set of objects (values). A python dictionary will map unique keys to values. Dictionaries are changeable or mutable in python. The values the keys point to are able to be any Python value. They are unordered, so the order the keys are added does not reflect with the order they might be reported back as.

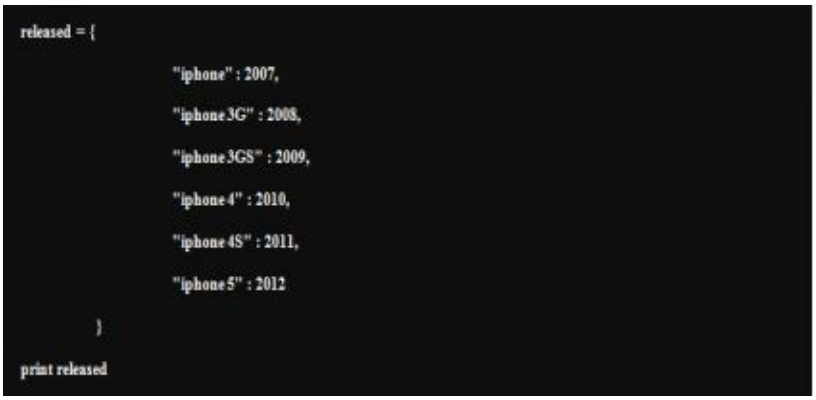
Let’s take a look at how to create a new dictionary in python.

In order to make a dictionary, you first have to start with an empty one. Use:

>>>mydict={}#

Example 1 – Creating a New Dictionary

This creates a dictionary that has six key-value pairs to begin with, where iPhone* is the key and the years the values.



The output for this would be:

```
>>Output

{'iphone 3G': 2008, 'iphone 4S': 2011, 'iphone 3GS': 2009, '

        iphone': 2007, 'iphone 5': 2012, 'iphone 4': 2010}
```

Now let's add a value to your dictionary.


```
>>Output

{'iphone 5S': 2013, 'iphone 3G': 2008, 'iphone 4S': 2011, 'iphone 3GS': 2009,

'iphone': 2007, 'iphone 5': 2012, 'iphone 4': 2010}
```

Now that you know how to do that let’s remove a key and the value from the code using the del operator.

```
del released["iphone"]
print released
```

The output would be:

```
>>output
{'iphone 3G': 2008, 'iphone 4S': 2011, 'iphone 3GS': 2009, 'iphone 5': 2012,
'iphone 4': 2010}
```

Now let’s check the length of the function by using the len() function.

```
print len(released)
```

Now let’s test out the dictionary! Check to see if a key exists in a given dictionary by using an operator such as this:

```
>>> my_dict = {'a' : 'one', 'b' : 'two'}
>>> 'a' in my_dict
True
>>> 'b' in my_dict
True
>>> 'c' in my_dict
False
```

or this if you have a loop:

```
for item in released:
  if "iphone 5" in released:
    print "Key found"
    break
  else:
    print "No keys found"
```

or this if you need a value of a specified key:

```
print released.get("iphone 3G", "none")
```

or this if you need to print all key with a for loop:

```
print "-" * 10
print "iphone releases so far: "
print "-" * 10
for release in released:
  print release
```

or this if you want to print all the key and values:

```
for key,val in released.items():
  print key, "=>", val
```

or this if you need to get only the keys from your dictionary:

```
phones = released.keys()
print phones
```

Now that you know how to do those, why not look at how to print the values?

```
print "Values:\n",
for year in released:
  releases= released[year]
  print releases
```

Here’s how to sort your dictionary:

```
for key, value in sorted(released.items()):
  print key, value
```

And that’s how you use a dictionary in Python!

Conclusion

There’s a lot that you can do with Python code. Many developers will use it to do something as simple as sorting some inventory to something complex like making a video game.

The important part about Python is that there are user groups all over the globes, known as PUGs, who do major conferences on continents across the globe so that Python users can interact with one another. Many of these conferences allow children to attend, who will learn how to use Python on their new Raspberry Pi’s they’re given at the conference.

Overall, the community surrounding the language of Python is very positive and upbeat. They are a group of people who want to make the world a better place with their knowledge. So never be afraid to reach out to forums and discussion groups to figure out how to write the code you need.

Thank you for reading. I hope you enjoy it. I ask you to leave your honest feedback.