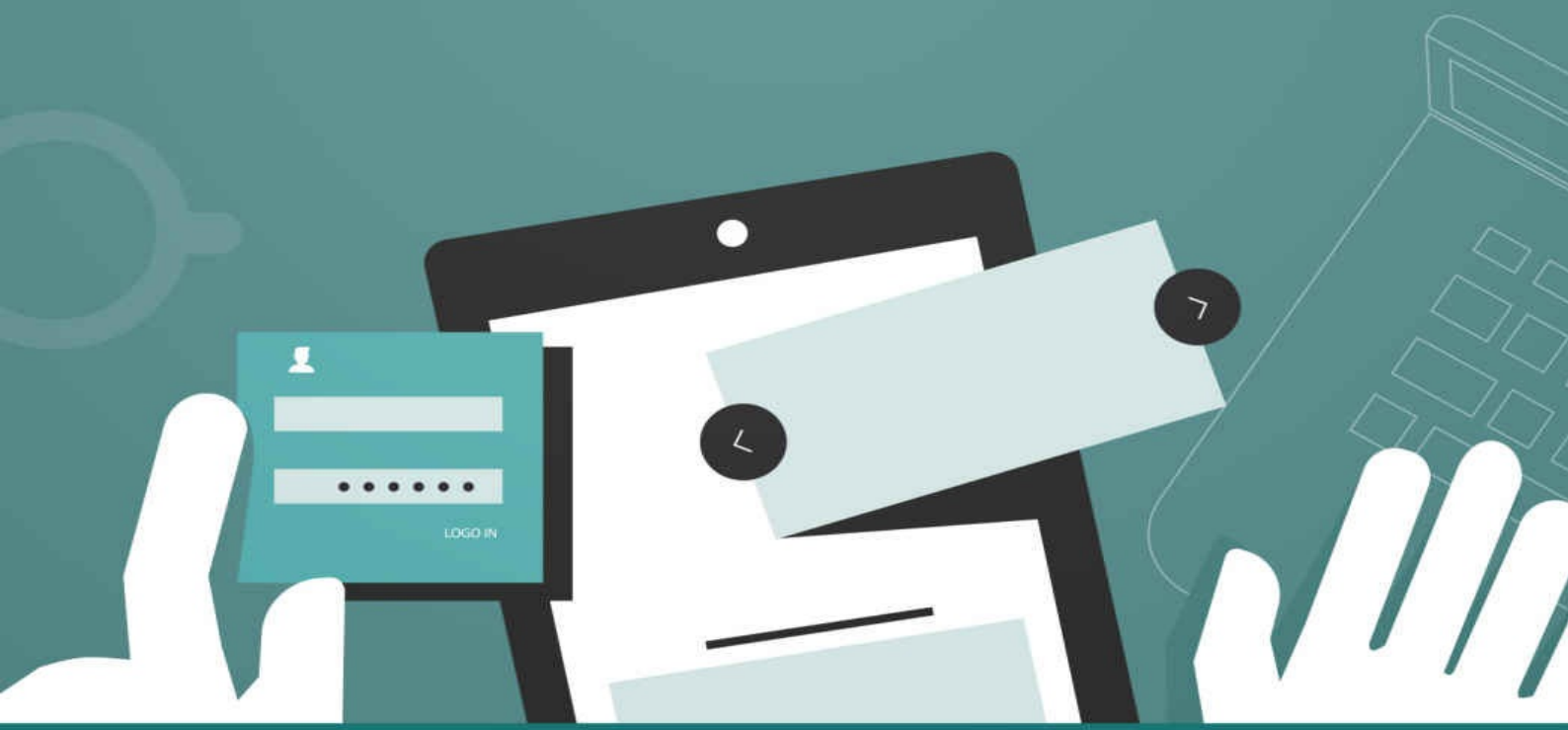
# INTRODUCTION TO PROGRAMMING WITH PYTHON

## Python Programming for Beginners




VIVIAN BAILEY

# Introduction to Programming with Python:

*Vivian Bailey*

# Disclaimer

## Further Information:

Limit Of Liability/Disclaimer Of Warranty: The Publisher And The Author Make No Representations Or Warranties With Respect To The Accuracy Or Completeness Of The Contents Of This Work And Specifically Disclaim All Warranties, Including Without Limitation Warranties Of Fitness For A Particular Purpose. No Warranty May Be Created Or Extended By Sales Or Promotional Materials. The Advice And Strategies Contained Herein May Not Be Suitable For Every Situation. This Work Is Sold With The Understanding That The Publisher Is Not Engaged In Rendering Legal, Accounting, Or Other Professional Services. If Professional Assistance Is Required, The Services Of A Competent Professional Person Should Be Sought. Neither The Publisher Nor The Author Shall Be Liable For Damages Arising Herefrom. The Fact That An Organization Or Website Is Referred To In This Work As A Citation And/Or A Potential Source Of Further Information Does Not Mean That The Author Or The Publisher Endorses The Information The Organization Or Website May Provide Or Recommendations It May Make. Further, Readers Should Be Aware That Internet Websites Listed In This Work May Have Changed Or Disappeared Between When This Work Was Written And When It Is Read.

# Table of Contents

# Why This Book?

The adage “pen is mightier than sword” is best served by programming languages. You literally type and communicate solutions to problems on a computer, and it comes to life.

This book introduces Python, the simplest and currently one of the most powerful Object Oriented Programming languages on the planet. The book shows how it is similar to learning a new dialect of English, one meant for the computing devices you hold in your hands.

It will introduce you to the basics of Python language: its fundamental building blocks, their syntax, their basic uses, and how they are related to one another in creating larger codes.

This book is meant for novices and beginners. Hence, it uses simple real life examples, offers short codes with explanations, and continuously links back to earlier chapters and builds on them.

# Chapter 1. Why You Should Learn Programming

The rise of smart devices, wireless connectivity, and increased user mobility in offices made computers an integral part of our lifestyles. As our reliance on smart devices and various apps to complete a diverse range of activities increases, learning to code is becoming akin to learning a second language.

Normally, we learn second languages to increase our reach, overcome geographic and cultural barriers, and to be able to communicate with people from diverse cultures. Smart devices have brought down barriers across different cultures, and have become one of our most trusted companions. However, we are not able to effectively communicate and instruct our smart companions to perform the actions that we desire.

Learning to effectively communicate with computers and smart devices, and being able to instruct them to perform different tasks is fast becoming something more than an added skill — it is becoming part of what it means to be human in the age of Digital Natives.

# 7 Reasons for Learning to Program

Learn to program and you will be able to experience the following seven, awe-inspiring realities of being a Digital Native.

## *It's the only time when writing can literally solve problems*

The adage "pen is mightier than sword" is best served by programming languages. You literally type and communicate solutions to a problem on a computer, and it comes to life.

It's amazing to realize how powerful this form of communication is. Most of smartphone users have visited App Stores and seen a range of apps, each solving a unique problem for us. Knowing a programming language means that you can break a problem into smaller problems and create programs for yourself and others.  So many problems, no matter how trivial can now be resolved simply by writing them out in a programming language.

## *Programming transforms your computer and smartphone into a powerhouse*

Humans landed on the moon relying on computing power at least a 1000 times less powerful than the computing power of our smartphones.

By 2016, over 2 billion people will be using smartphones across the globe. Just imagine what this processing power can be put to use for. Imagine what you can create and develop on your PC for use with your smartphones: personalized apps, games, tools, and more.

The possibilities are endless.

## *Programming simplifies tasks by automating them*

We live in a world where efficiency is the hallmark of everything. Nobody wants to perform tedious and repetitive tasks if you can pay for a solution that can automate it. This ranges from office tasks to personal ones.

By learning to code, you can create simple scripts that simplify tasks. Think of generating macros in Microsoft Office (Excel and Word) or scripts that simplify tasks you have to perform regularly at work e.g. creating folders for projects, completing forms, creating shortcuts, adding more features to existing software, etc.

## *It develops your ability for abstract and disciplined thinking*

Instructing computers requires that give them step-by-step instructions that link pre-determined inputs to expected outcomes. To communicate to the computer, you have to write in an explicit language, from problem to solution. This requires breaking down abstract problems into simpler ones, and then organizing them into inputs and outputs.

Programming teaches you to organize thoughts and break down a problem into constituents, identifying external factors and triggers, and mapping a step-by-step route for computing the inputs into outputs i.e. into simpler chunks of programmable code.

This trains your mind to think clearly and systematically, filtering out information that has no bearings on the solution, and finding the simplest solution to the problem.

### *Future jobs will require software skill*

Currently, programming may be a skill to boast your chances of entry into the market, however as organizations focus on increasing interdepartmental collaboration, they begin to prefer people with inter-disciplinary skill sets. In the long run, being able to program will become a skill needed to survive in the technology market.

### *Because it's not hard to learn*

If you can comprehend English language, learning to program is simply a matter of learning the logic behind programming *words*. Over the years, programming languages have become more intuitive, utilizing commands that clearly state their purpose and hence allow you to nearly construct sentences "If-Then", "Else", "Print", "Switch", are just the most commonly used commands in programming language.

All you have to do is learn the grammar, and with practice you will be able to write those short essays that identify the problem, its components and inputs, and the outputs for each input.

### *programming has great Earning potential*

As you get better at programming, you can easily leverage your problem solving skills to develop software for sale. This includes working as a freelancer — building apps, developing custom software, creating and designing websites, or helping clients complete tasks using existing software and platforms (Microsoft Excel and Word, Flash, Java, etc.).

## In Conclusion— Prepare to learn

Programming is an add-on to your current understanding of the English language. Think of it as a new dialect, one that your computer and smart device understands, one that has a lot of benefits in the modern world, is easy to learn and one that will soon become a must have for digital natives.

# Chapter 2. Why Python is the Perfect Language

In an age where machines are becoming highly intelligent and are continuously simplifying human-machine interactions, it only makes sense to use an intelligent programming language.

Existing statically typed languages such as Pascal, C, or other subsets of C++ and Java introduce verbose syntax that cloud the actual process of problem decomposition and design of data types that have to be programmed.

Hence, for someone who does not know how to code, these languages seem unnatural. Furthermore, the additional complexity of the syntax only slows down the pace of ingraining the methodology of programming.

This is where Python brings in the power of simple and consistent syntax, backed by a large standard library with real problems.

Notwithstanding the rise in its popularity in American colleges as reported by the Association for Computing Machinery, Python has effectively replaced Java as the first language of choice for budding programmers.

So what is making Python the perfect language after two decades of being in the field?

# #1— Minimal Setup

Installing and running Python is extremely simple. All you have to do is download the file and either run it through the Terminal program (for Mac), or the PowerShell program (Windows), and Lo and behold! You’re running Python.

## #2 — Writing a program is akin to writing in English!

Python is a universal language. This means that its syntax and coding lingo is exceptionally simplified and easy.

Python is the closest thing to writing a logical argument in English. The commands are simple and the additional baggage of grammar (programming syntax such as brackets, colons, quotes, etc) is minimal. You grasp how to systematically break down a problem into simpler steps, and you can easily code it in Python.

In a phrase: a great first experience for any beginner.

The standard first hands-on programming experience that all beginners go through is to print something on the screen using coding. This is normally the words “Hello World”, among others.

Python makes it literally a matter of writing a sentence

```
print(“Hello World”)
```

Compare this to the following (Java):

```
public class HelloWorld {
    public static void main(String[] args) {
    System.out.println(“Hello, World”);
    }
}
```

Furthermore, when you move ahead into the course, other examples of simplified programming will leap out, including simple reading/writing of information, string processing, GUI’s, website code, Databases, websites, etc.

Its simplicity and ease of programming is the reason why it has been adopted by so many people and for such a wide range of tasks. This is further made enjoyable by the fact that a very large community of enthusiastic developers is always a buzz to aid new comers become initiated into the fold.

## #3 — Python is easy to read

Python is designed to be an easy read.

“Readability” of a code may and seem unimportant right now, but it becomes the defining factor when the code becomes larger and more complex.

Readability is very important, because unlike a course book, the chapters (or chunks of code) are not always arranged to work in a step-by-step or linear order. At times coding blocks in the far recesses of the code need to be called earlier, or later, multiple times throughout the code, and so on.

Ease of readability is important for sharing a code between a team, for troubleshooting a program, and for making changes to it. In traditional languages, reading becomes increasingly difficult as because of all the non-English grammatical syntax.

Python uses indentation to give structure to the code, and though programmers who are migrating from a different language to Python may be heard complaining about the lack (and near absence) of braces, it is one of the key benefits of Python: it simply de-clutters the work environment!

Think of indentations like the headings in your word processing software (H1, H2, H3, H4, H5, and so on). In coding the headings are normally curly brackets (“{}”) that have to be added at the start and end of the paragraph.

As a result, two types of communication are happening simultaneously: braces to tell the machine about the program, and the indentations to tell the reader (another programmer) what the program is about.

Now imagine NOT having to add brackets and indentation and follow a legend you have created earlier to keep your program readable. Python does this by offering a single structure to denote a program: making it easier for humans and computers alike to read it with ease.

## #4 — You do not have to compile the program to run it

When you're learning something new, mistakes are bound to happen. How those mistakes are shown to you often plays a crucial role in how motivated you are to try again, or how affected you are by the mistake that you have made. With Python, all errors are identified at *run time* instead of showing a failure to compile error. This makes it easier to identify and fix mistakes immediately.

This is incredibly useful when you will be designing a complex algorithm where changes are made continuously. Where other programming languages will waste time taking you through the compile-run-debug cycle, Python simply shows the result (or the problem) in the same interpreter.

Hence, you can make innumerable changes to a piece of code and execute it in real-time in the interpreter. This boosts the learning process as you can consciously make errors, see its impact, and troubleshoot the program. This dramatically reduces the development cycle, and becomes especially useful for rapid prototyping of your code.

# #5 —Python is Open

Python is an open on two accounts:

1) Open source as programming language
2) Built using Open objects

As an open source platform, its liberal distribution license allows the language to be used for coding programs/apps that can seamlessly be integrated as an extension of other propriety languages.

In terms of coding architecture, Python is great at introspection because the code is based on discrete chunks of programs (known as objects). We will get technical later, but for the moment “being open” means that it will be really hard for you to write dirty code or sidestep proper coding methods to solve problems.

Hence, Python forces you to write better code from the beginning, and this proves very helpful during debugging.

# Chapter 3. Getting Started: Pre-reqs

There are only two prerequisites for getting started with installing and using Python:

- Reasonable knowledge of using a computer and internet.
- A compelling desire for learning a new language for communicating with computers.

# Installing Python on Windows

Download the latest version of Python from the official Website. [LINK]

**NOTE:** Two versions of the language are available, Python 2.x and Python 3.x The difference, in all its simplicity is that *Python 2.x is legacy* i.e. it is the one that has been installed on most devices and hence is the current norm for Python, whereas, *Python 3.x is the present and future of the language.*

The Windows version is downloaded as an MSI package, and can be installed with a double-click on the open shell file.

Allow the wizard to install Python to the default directory:

- For Python 3.x it will be **C:\Python3x\ —** (x being replaced by the version you've downloaded, the latest being 2.7)

**NOTE**: The different folders mean that you can install multiple versions of Python on the same system without causing any conflicts.

However, a single interpreter acts as the default application for Python file types.

That's it.

You can use Python; however, it is recommended that you install the libraries and tools described next before working on writing your code. In particular, Setuptools is a must install as it allows you to easily use other third-party Python libraries.

## *Setuptools + Pip*

From all the *Setuptools*, this is the most crucial third-party Python software because it extends the capabilities of the packaging and installation facilities (offered in the distutils present in the standard library).

Once the Setuptools is added to your Python system and directory, all Python compliant third-party software can be added using a single command. Furthermore, with the Setuptools installed, you can add a network installation capability to the software that you create.

Python 3, generally, comes equipped with PIP. You can download the latest version of the Setuptools for Windows HERE.

Now install PIP, a replacement for the Python command easy_install and which introduces un-installation of different packages.

## *Virtual Environments*

Next, you have to install a Virtual Environment, *virtualenv,* a tool to keep each project

self-contained and separate from one another. Think of it this way: when you write a code, it leverages the Python work environment to reduce its size (e.g. using some pre-made functions from the Python library). Hence, when it is saved, only the essential information needed for the saved file to be reopened and run in the Python work environment.

Now you create another code and save it.

Now the new project may use different library objects from Python, however, unless a mechanism exists to keep their reliance separate from one another, one program can easily conflict with the other.

A virtual environment separates all dependencies required by different projects in separate places. Hence, each program can safely run in its own safe haven.

Download it from HERE.

## The second Prerequisite

- A strong desire to learn how to write computer programs.

If you don't know how to program and are new of Python, then understand that as with any new language, it takes time, practice, and perseverance in face of mistakes and small failures. Hence, you must have a strong desire to learn the language to succeed.

### *Strong desire to learn*

Python requires focused effort in learning the basics, learning a new method for logically breaking down a problem into constituent parts, learning a new programming methodology (Python is an object oriented programming language), and practicing coding behavior in a new coding environment.

Don't expect a miracle to happen.

You won't learn it by simply reaching the end of this book.

As with any language (spoken or coded), practice and experience matter a lot. Simply reading and memorizing syntax will not bring a miraculous transformation and make you an apt programmer in the field.

Unless you are willed by a true desire to learn the new language, learning to program can become boring, mistakes will compound as a de-motivating factor, and you will not ingrain a new analytic process of breaking down problems and writing solutions.

### *Practice, more practice, and even more practice*

All languages are learned by repetition and experimentation.

You have to learn how to put together phrases and create meaningful sentences out of them. With Python, it is about using the syntax and commands to create chunks of code that can perform an action (take input, compute, display an output, etc.), and do so efficiently.

This book will offer you the basics of the language, but you must will yourself to put your brain in high gear and write A LOT of code for different problems. Only then will the information that is shared in this book will stick with you.

Think of the things you have become good at over the years: talking, surfing, gaming, playing music, general knowledge about fashion, writing good essays, humor, and more. All of them needed to be practiced again and again to gain sufficient command and confidence in you abilities.

Programming with Python is quite similar to that experience.

# Chapter 4. Variables, Expressions, Statement, Strings

Remember those English grammar lessons? How “each sentence” is made of a subject and an object, has nouns, pronouns, auxiliary verbs, and adjectives, and the like?

Just remember how words and auxiliary verbs were used to create sentences, and how these sentences are used to communicate more complex thoughts.

This is what we are going to do here.

Python (and most other programming languages) are made of some universal basic components. These include (in order of increasing complexity) variables, expressions, statements, and strings.

The names are self-explanatory. Let’s see them in detail.

# VARIABLE

A variable is the “x” you tried to find in most of your math classes. In programming it is a *data type* capable of containing changeable values.

The ability to manipulate variable in a programming language is one of the most powerful features in the coding world.

Hence, we must begin by understanding “values” in Python.

## *VALUES AND DATA TYPES*

Values are categorized according to the type of data they can handle. For example, “Hello, Programming!”, “2”, and “2.3” are all considered values in Python, except each belongs to a different category or different *data type*.

**NOTE:** *Data type* donates the different types of data that can be used e.g. characters, numbers, float point (decimal numbers), etc.

The numeric 2 is an *integer* because it contains a whole number, whereas the “Hello, Programming!” is a *string* because it contains a string of letters, and “2.3” is a *float* because it has decimal number. Your Python interpreter is able to identify and use strings as a single value enclosed within the quotation marks.

If in confusion about the type, let Python tell you the data type of your value with the *type* () command:

```
>>> type (“Hello, Programming!”)
<type ‘str’>
>>> type (13)
<type ‘int’>
```

Note that the strings are always contained within quotes “”. Hence, if you were to put numeric values within these quotes, they automatically become strings.

```
>>> type (“13”)
<type ‘str’>
```

Furthermore, when writing strings, avoid using commas to separate tens, hundreds, and thousands. For example, if you want to write two hundred thousand as 2, 00,000, Python would react as follows:

```
>>> print (2,000,000)
2 0 0
```

Unexpected right?

That’s because commas are interpreted by Python as separators. Hence, you see three values: 2, 0, and 0.

**NOTE:** The syntax for print is print (x) with ‘x’ being replaced by strings (“string”) or normal numeric value (2)

This is different from Python 2, where the syntax is simply: print x, print 2, or print “this”

Other types include *float* (for decimal places)

>>> type (13.1)

<type ‘float’>

>>> type (“13.1”)

<type ‘str’>

**NOTE:** You can use both, single (‘’) and double (“”) quotes for writing strings in Python. AND, you can use double quotes within single quotes. For example: (‘He said, “Lo and behold!” ‘)

Let’s start with variables.

## *VARIABLES*

Remember how basic equations were created in math?

**Question:** If Sam bought two eBooks for $3, how much will it cost to purchase 5 eBooks? We normally began with supposing “x” as the cost of 5 eBooks.

Once supposed, “x” can be used anywhere in the solution and anyone can easily figure (interpret) that “x” is referring to the “Cost of 5 eBooks” defined in the beginning.

In programming, a variable is the name that refers to a changeable value — one that can be changed and automatically updated at will.

Like math, variables have to be declared/created (supposed) and then assigned a value. In python, the value of a variable is assigned using the **assignment statement:**

```
>>> message = “I’m programming with Python!”
>>> x = 2015
>>> yr = 365.25
```

Notice how each type of data is assigned in a similar manner, starting with a string, an integer, and a floating point number. The most important thing is the equal sign “=” that separates the *name of the variable* (left hand side) from the *value of the variable* (right hand side). The equal sign “=” is known as the **assignment operator**.

The left-hand, right-hand divide is a rule. Deviation is not permitted. If you reverse the order, it becomes meaningless for the interpreter.

Hence, the following makes no sense.

```
>>> “I’m programming with Python!”= message
>>> 2015 = x
>>> 365.25 = yr
```

## *Types of Variables*

Variables also have types. In the example above, the message is a string, ‘x’ is an integer, and ‘yr’ is a float.

You can verify the type of the variable using type().

```
>>> type(message)
<type ‘str’>

>>> type(x)
<type ‘int’>

>>> type(yr)
<type ‘float’>
```

Note how the type of the variable corresponds to the type of the value it contains.

## *Naming Your Variables, The Illegalities*

Certain naming conventions are used when naming variables. These are simply standards that are used to standardize how code is written and to keep it clean. This gains importance as your code gets longer and your variables become more descriptive.

Additionally, certain python rules limit how your variable can be assigned. Your variable must:

1. Always begin with an alphabet/letter
2. Never use illegal characters ($, &, *, etc.)
3. Never use a Python *keyword*

Therefore, the following will result in syntax errors:

```
>>> 14numbers = “What are those fourteen numbers?”
SyntaxError: invalid syntax

>>> Increase$ = 985
SyntaxError: invalid syntax

>>> class = “Python Basics “
SyntaxError: invalid syntax
```

Now “class” is a Python keyword, which means it is part of its rules and structure.

Python has thirty-one such keywords:

| | | | | | |
|---|---|---|---|---|---|
| raise | while | return | try | with | print |
| is | or | lambda | not | pass | |

| | | | | |
|---|---|---|---|---|
| for | if | from | global | import |
| del | except | elif | else | exec |
| as | class | assert | break | continue |
| yield | and | def | finally | in |

Keep this table handy, and if the interpreter complains about syntax error, check your variable naming!

# STATEMENTS

Like normal language, a statement is an instruction. In Python, you can execute primarily two kinds of statements: print and assignment . The former produces a value while the later does nothing.

When you write a statement, the interpreter executes it and tries to display results:

```
print (3)
t = 17
print (t)
```

The above code will produce the output:

```
3
17
```

The assignment statement used for creating a variable produces no output.

# Expressions

As in language, an expression is a complete thought, with a premise and a conclusion. In python, an expression has inputs, a defined/expected use of those inputs, and an expected way of either directly displaying an output or computing it to display an output.

Generally, meaningful expressions are a combination of variables, values, statements, and operators that forces the interpreter to evaluate it and display results:

```
>>> 1 + 1
2
```

A value and a variable itself are simple expressions in Python:

```
>>> 2015
2015
>>> t
17
```

However, these expressions simply print/display the expression. Evaluating an expression is not similar to simply printing a value.

Take the example:

```
>>> message = “Writing expressions?”
>>> message
“Writing expressions?”
```

Notice how the quotation marks are displayed in the output. However, when using print() , only the contents of the string are displayed:

```
>>> print (message)
Writing expressions?
```

This is because the print ()  statement fetches the value of the variable (the string).

Now, an expression all by itself is a legal statement. You can write as many as you want, but unless you use certain operators and commands to communicate the purpose of the expression, NO OUTPUT will be displayed. For instance:

```
2015
365.25
“Python, Expressions!”
13 + 1
```

No output will be displayed.

This is where *operators* and *operands* come into play.

## *Operators and operands*

A code is supposed to perform certain computations on the expressions to generate predictable outputs. Computations are performed using **operators** — special symbols

which you may think of as short forms for different commands.

The value that the operator uses is called an **operand**. Here are some examples:

```
1+2+3
t-20
t*30+minutes
t/60
(1+2+3)* (6-2)
```

How the symbols have been used +, -, /, * (for multiplication) are the same way they are used in mathematics. Now, when a variable is used with operator(s) to create an expression, the interpreter fetches and computes the value assigned to that variable.

However, there is a caveat. Though addition, multiplication, subtraction, and exponentiation (represented by the double asterisk operator **) have a single use and generate an expected result, division does not.

Let's say the following expression is used:

```
>>> t = 49
>>> t/60
0.8166
```

But what if wanted to perform an integer division where the answer is always rounded off to the closest integer value?

Let's say:

```
>>> t=121
```

This is where "//" is used to indicate our intention

```
>>> t/60
>>>t//60
2.0166
2
```

Another way of overcoming is to assign data types to the answer (a float), but we will come to that later.

### *Order Of Operations*

You can use multiple operators in a single expression; however their evaluation is depended on the **rules of precedence.**

Python follows the same rule as is applied for mathematical operators. Remember PEMDAS to memorize the order :

- **Parentheses** — Expressions in the parenthesis are always evaluated first. Hence, `3*(4-1)` is 9. Parenthesis are also a great way for making it easier to read

the expression such as `(t*100)/60` .

- **Exponentiation** — (**) has the next highest precedence. Hence, `3**1+1` is 4 and not 9, whereas `4*1**3` is 4 and not 64.
- **Multiplication and Division** — Both have the same precedence, albeit higher than addition and subtraction
- **Addition and Subtraction** — Also have the same precedence, albeit lower than multiplication and division.

What if multiplication and division come in the same expression?

For example: `t*100/60`

Operators that have the same precedence are evaluated from their appearance from left to right. Hence, in the expression `t*100/60` , multiplication happens first, followed by division.

## Operations and strings

Generally, you cannot perform any mathematical operations on strings. This includes instances where the strings look like numbers, or only have numbers in them. The following example shows an illegal mathematical operation on a set of strings:

```
message-1 “Word”/2015 message*“Now” “2012”+3
```

Though *mathematical operations* cannot be performed algebraically (or as we are accustomed to) on strings, some operators can work with them, but in a different manner.

For example, the “+” operator when used with strings performs *concatenation* instead of addition. It means that the two strings (or operands) are *linked* end-to-end i.e. they are attached one after the other:

```
Subject = “Python”
Object = “ Programming Language”
Aux_Verb = “ is a”
Adj = “ simple”
print (Subject+ Aux_Verb + Adj +Object)
```

The output of this code will be program Python is a simple Programming Language . The spaces before the words in the string are part of string, and necessary to produce the grammatically correct spacing between the concatenated strings.

The multiplication operator (*) also works on strings. Its function is to perform repetitions on that string. So, for example, the operations:

```
‘ Why?’*3
```

Will produce the results ‘Why? Why? Why?’

In multiplication’s (*) case, one of the operands must be an integer while the other a string.

# Chapter 5. Functions

Let’s delve into purposeful programming where the program will demand user input to execute a command or to perform a task.

This is achieved using *functions*.

Let’s take a real life example say you want to surprise a friend on their birthday. You inform other friends, gather a group, and *tell them* how the party will go. For example, you decide on a signal at which they will turn on the lights, blow the trumpets, or sing the birthday song. In this scenario, you have created a basic plan of action and a signal (an input) and how everyone (individually) will react to it.

You are not expecting to tell everyone what to do *again* when you give them the signal, BECAUSE you have already defined different functions and associated signals that will activate them.

In Python, *functions* are smaller independent pieces of code in your program. They allow you to divide and order your code, and hence make it compact, more organized, and readable. Once created, a function can be used at any time, and any number of times *anywhere* in your program.

# Using a function

Python has a library full of pre-made functions for various common tasks. You can use them by simply calling them, and giving them an input.

“Calling” a function simply means using the function name (and not copying the whole code again), and giving the function an input it can compute.

Think of it like calling the movers or the letterman. They know their function, and if you give them the right input (i.e. things to be moved or a letter/parcel), then you would not have to explain it to them what to do with that input. They will complete the job once you give them the right input.

In Python, calling a function is simple. Define a parameter name and its working parameters:

```
function_name(parameters)
```

Simple, right?

Let’s understand the details of the syntax shown above:

- **Function_Name** identifies which function it is you want to use. You will figure it out yourself, as in the previous analogy, if you want to move something you’d call the movers and a letterman if it’s a small parcel/letter. As an example, a basic function is the raw_input function. We’ll use it soon.
- **Parameters** are the values or the inputs that you pass to the function. This value will inform the function what and how it should use it. For example, if you have a function that added a value to a number after multiplying it by 3 then the parameters will be 1) the number it should multiply by 3, and 2) the number it should add to this result. So if you put 5 and 2 in the parameters for this function then it will first multiply (5 x3) and then add 2 (15+2), giving you 17

## *Parameters and Returned Values*

Now, functions normally work in the background and will normally not show the result of the computation they have performed on the parameters you have given them.

Returning the value or result of its computation is important to know where the program is, or if it is generating the expected results. What if you have given it a letter or an illegal symbol ($,%, #) instead of a number?

So how does a function show what it does, or has done on the terminal?

Let’s see how the computer sees the function. It does not see the name of the function (or in case of a variable, the variable’s name), rather, it only sees the final value that the function or the variable has stored in it.

Hence, in case of the function, the computer sees the final, end result.

Let’s take the example of a function multiply()  that multiplies any number by 7. So, its input is any number you can think of and place it in the brackets, the function multiplies it

by 7.

NOTE: multiply() is not a pre-made function in Python.  You can't use it straight out as mentioned here.

So if you typed this:

```
t = multiply(60)
```

It will appear to the computer as this:

```
t = 420
```

The function ran itself, and returned a number to the main program, using the parameters we had given.

Let's try a real function, **input** .

**NOTE**: Previous versions of Python (<Python 3) used raw_input instead.

The following function asks the user to enter a value that it turns into a string of text. For example:

```
# the first line makes "t" equal to the value that you type in on the screen
t = input("Type something, and I shall display it on the screen again:")
# The next line will print, or return the value that you have entered in 't'. It will show the result of the function
print (t)
```

The result is simply the content that you have entered on the screen after the program is run.

**NOTE:** ‘#’ is used for writing comments in the code. Any line that begins with “#” is not computed by the Python interpreter.

Think of commenting as your in-program documentation, informing or reminding you about what each chunk of code/function does.

It becomes crucial as your code becomes larger and more complex. Practice it from the beginning .

So, if you type in “Really?” in the above program when asked to do so, the computer sees the function as:

```
t = “Really?”
print (“Really?”)
```

Remember, that to the computer, variable is just a stored value. It does not appear as ‘t’. So is the case with functions. They don’t appear as the whole code, only the value they contain in them.

### *Defining Your Own Functions*

You can create your own functions using the ‘ def ’ operator.

Think of a function, and what it will do. Now name it, and use ‘ def ’ as follows:

```
def function_name(parameter_1,parameter_2):
#notice the colon ":" at the end of the line? #it is an important part of the syntax. Always
#add it at the end of line that starts with #'def'
    {this is the code in the function}
    {more code}
    return {value to return to the main program}
```

Recall that to the computer, the variable in the function looks like the value that is stored inside it and that functions are similar and only the value that they return is seen by the computer.

It does not matter how large your function’s code is. Only the value matters. Because the function is a separate, self-contained program, it doesn’t see any of the variables in your main program.

Why the stress on what the computer sees?

Here is an example to show the importance. The following function ‘written ()’ will print the word “Functions are independent of the main program” on the screen, followed by returning a number “54321” *to* the main program.

```
#creating the function
def written():
    print (” Functions are independent of the main program “)
    return 54321
# how to use the function
written()
```

The last line of code is calling the function. When you type that, the function's ***values*** are displayed on the screen and not the complete function:

```
Functions are independent of the main program
54321
```

So what happened?

As soon as the def written()  was run, Python created a function and labeled it 'written'

When the written()  was run, Python executed the function 'written' that it had previously created. Running the function meant executing all lines of code within in.

The function first printed "Functions are independent of the main program" and then returned the number ' 54321 ' back to the main program.

As a result, the main program only sees the line as 'print 54321' and so displays '54321' in the next line. Your function did not know that the string "Functions are independent of the main program" had been printed already, all it saw was that it received a new value '54321' and printed that onscreen.

Functions can also receive arguments i.e. you can pass variables to them. For example:

```
def a_function_with_args(username, message):
    print ("Welcome, %s , I am your new function. I wish you %s"%(username, message))
```

Functions can also return values.

Functions may return a value to the caller, using the keyword — 'return' as used in the previous example. Another example:

```
def add_both_numbers(a, b):
    return a + b
```

# Chapter 6. Conditionals, Flow Control, Decisions

Useful programs are able to make decisions based on the inputs. This is achieved by giving them the ability to change the behavior or *flow* of the program based on the inputs. This is done using conditional statements.

The *flow* of the code is the order of the code you have written. Recall the birthday example. You may have planned that the hallway lights will go up first, followed by the friends on the stairways singing the songs, someone coming with the cake, etc.

But different inputs (or reactions by your friends) the flow of the party might have to change.

In coding, normally, when the code is executed, the interpreter reads it line by line, moving from the top to the bottom. However, at times you may need to execute a function that begins 40 lines ahead. The most effective way of increasing efficiency of the program is to *break the flow* and immediately bring the interpreter to the function and execute it.

When there is a need to break the code and execute a different block of code altogether to complete a task, *conditionals* come in.

Conditionals can be divided into *basic conditions* and the *If* conditions, including:

- If-Statement
- If-Else Statement

Let’s take them one by one.

## *Basic Conditions*

These include all the arithmetic comparisons, such as less than (<), greater than (>), equal (=). For example:

```
3<4
```

```
3>2
```

```
t=5
```

```
t>7
```

All of the above have Boolean results i.e. they are either true or false. These conditions can be used to control the flow of the program using conditional statements.

However, not all arithmetic decision are transferred exactly to Python:

| | | |
|---|---|---|
| Greater than | > | > |
| Less than | < | < |
| Greater than or equal | ≥ | >= |

| | | |
|---|---|---|
| Less than or equal | ≤ | <= |
| Equals | = | == |
| Not equal | ≠ | != |

## *Control Flow with The If-Statement*

We are writing a code for a moving company that offers flat rates for all items *less than* 60kg, whereas anything *greater than* that is charged $20 extra. Our code will used with their weighing machine to inform the customer of the extra charges. Here's an example code:

```
weight = float (input ("Please enter the weight of your luggage (in Kg):"))
if weight > 60:
        print ("This exceeds our 60 Kg flat-rate limit. An additional $20 is applied for heavier luggage like this.")
print ("Thank you for your business.")
```

Notice the indentation before the first print command. This indentation tells Python what to do when the statement weigh > 60 is true or false:

- When the weight is *greater than* 60 kg, it prints additional charges, and
- When it is not true (weight<60), it thanks the customer.

The syntax for the if-statement is:

```
if condition:
    Indented code/statement
# the colon is part of the syntax.
```

What if you want to execute separate codes for both, True and False conditions? If-else statement is used for that as well using the if-else statement.

## *Control Flow with If-Else Statement*

The general syntax for if-else statements in Python is:

```
if condition:
    Indented code if condition is True
else:
    Indented code if condition is False
```

The indented blocks can have any number of further statements inside them. For example:

```
time = float (input ('How long did you run?'))
if time > 20:
```

```
    print (‘ Good work! Let’s continue maintaining our health’)
else:
    print (‘Good effort. Let’s try to hit our target tomorrow!’)
print (‘Exercise for Healthy Living’)
```

The if-else statement is limited to testing a single condition and control the flow using only two results (True/False). Multiple tests can be included using the if-elifstatement .

## *Controlling Flow Using If-Elif Statements*

Why not use the if-else statement repeatedly in case of multiple tests? Let’s take the example of grade assigner:

```
def GradeAssign (score):
    if score >= 85:
    assign = ‘A’
    else:   # if not A then maybe B, C, D or F?
    if score >= 75:
    assign = ‘B’
    else:  # if not A then maybe C, D or F?

    if score >= 65:
    assign = ‘C’
    else:    # grade must D or F
    if score >= 60:
    assign = ‘D’
    else:
    assign = ‘F’
    return assign
```

Notice the number of indentations.

As your code grows longer, this will become problematic. See how if-elif transforms it:

```
def GradeAssign(score):
    if score >= 85:
    letter = ‘A’
    elif score >= 75:
    letter = ‘B’
    elif score >= 65:
    letter = ‘C’
    elif score >= 60:
    letter = ‘D’
```

```
    else:
    letter = ‘F’
    return letter
```

### *Nested Statements*

Your code can have statements within statements as well:

```
if t>60:
    Indented statement #1
else:
        if t<30:
    Another indented statement #2
        else:
    Indented statement # 3
```

As your program grows, nesting statements will increase the decision-making ability of your code.

# Chapter 7. Iterations

Programs are used to automate and simplify problems. This requires repeating tasks, e.g. asking for input multiple times when a table needs to be filled. This is called an iteration of the same task. They require the interpreter to execute the same block of code multiple times.

However, this is not possible with how the interpreter normally works i.e. *sequentially* by moving from one block of code to the next.

This is where loop statements come into play.

They allow us to execute the same block of code as many as is needed to complete the task.

There are two main loops: the While Loop and the For Loop .

Both loops rely on updating variables. So far we have only created and used variables, without changing and updating the original value of the variable during execution. Here's an example of updating variables:

```
a = 1
b = a+2
b = 2*b
a = b-a
print (a, b)
```

The program is executed step-by-step, top to bottom. Here's how the variables get updated at each line:

| Line | Value of 'a' | Value of 'b' | Details |
|---|---|---|---|
| 1 | 1 | - | |
| 2 | 1 | 3 | a =1, so a+2 = 1+2 =3 |
| 3 | 1 | 6 | b = 2*b. Now b=3, so 2*b= 2x3=6 |
| 4 | 5 | 6 | b-a = 6-1 =5 |
| 5 | 5 | 6 | print: 5 6 |

The interpreter always executes the commands in a sequential order, updating any variables that are changed during the process.

The loop statements break this sequential flow, allowing us to access functions defined anywhere in the whole code.

**Loop Control Statements**

When should the loop start? How long should it continue, and when should it terminate?

This is achieved using ***loop control statements***.

Without them, the interpreter will either not execute the loop or if it does will never get out of it. Let's see how it works practically.

**List Types and Range Functions**

To use the for-loop, it is important to understand *list* type. Lists are ordered sequence of data. Some examples include:

```
['white', 'red', 'green']
```

```
[1, 2, 3, 4, 5, 6]
```

```
['word', 34, 'another word', -4, 'last']
```

```
[] # completely empty list
```

The basic for-loop relies on the list type.

### *The for-loop*

This loop iterates (repeats) a function or set of commands for the fixed number of parameters you give it. This is different from the While-Loop, which runs as long as a condition is true.

For example:

- If it is raining outside, take the rain coat.
- While the sun shines, the monks continue working tirelessly on the field

Hence, unlike the while-loop, the for-loop runs once.

The syntax of the for loop is

```
for variable/function in sequence/list:
        indented statements to loop
```

Here's an example:

```
for count in [1, 2, 3]:
  print(count)
  print('Go' * count)
```

The above for-loop has the for-statement, the variable which it has to work on, and the list of parameters it has to work with.

The loop has three parameters and a variable. When it is executed, the two lines of code

are repeated three times, once for each list parameter.

| Line | Value in ‘count’ | Details |
|---|---|---|
| 1 | 1 | Takes first element from list |
| 2 | 1 | print 1 |
| 3 | 1 | Apply operator ‘Go * 1 equal ‘Go’; prints Go |
| 1 | 2 | Update value to next element |
| 2 | 2 | print 2 |
| 3 | 2 | ‘Go * 2 equals ‘GoGo’; prints Go Go; |
| 1 | 3 | change count to the next element in the list |
| 2 | 3 | print 3 |
| 3 | 3 | ‘Go’ * 3 is ‘GoGoGo’; prints GoGoGo; |
| | | List complete, done. |

In this case, the for-loop performs two actions: updating the variable in place of ‘item’, and running the indented block of code after. The above example used the variable within the loop as well.

A simpler loop can also be created without using a list. This is where *Range* (a pre-made function in Python) can be used.

```
for x in range(7):
    print(‘For-Loop’)
```

Notice that the variable x is not used inside the body of the for-loop. You can choose the number of times the loop must iterate or repeat itself.

```
# a program for specified repetitions
n = int(input(‘How many times do you want to repeat this?: ‘))
for x in range(n):
    print(‘Repeating…’)
```

On execution, the program will print the statement ‘Repeating…’ *n* number of times i.e. the number you enter in the beginning.

### *The While Loop*

It is a Boolean loop that repeats the same set of command until the original condition is False.

The syntax for the while loop in Python programming language is:

```
while expression:
    indented statement(s)
```

As long as the expression (its Boolean condition) holds true, the indented block of code will be executed again and again. The moment the condition becomes false, the program moves out the loop and executes the next line outside of it.

Example

```
n = 0
while (n< 5):
    print (‘I have counted till: ‘, n)
    n = n+ 1
print (“Good bye!”)
```

The above while-loop holds true for as long as the *updated* ‘n’ has a value less than 5. The moment it reaches 5 the program will move to the first line outside of the loop (the un-indented line).

```
Good bye!
```

How about a loop that can never end?

An infinite loop where the false condition does not exist? These loops are important for keeping the program running and seeking continuous input from the user. If an infinite loop was not running, any smart device or PC would shut down after loading (completing loading) its operating system.

A loop becomes infinite loop if a condition never becomes FALSE.

```
n = 1
while n == 1 :  # This is the official secret for constructing an infinite loop
        a = input(“Enter a value:”)
        print (“You entered: “, a)
print “Good bye!”
```

# Chapter 8. Lists, Dictionaries, Tuples

Variables allow us to store information that can be changed anytime. However, they store a single piece of information at a time: a value, a string, etc.

What if we need to store a list of information that will not change over time?

For example, the contact information of your family members, the names of the months of the year, or a phone book where you have multiple information ( the name of the contact and related phone numbers.

This is where Lists, Tuples, and Dictionaries come in. Let’s briefly discuss each so you know the difference, before using them in a code.

| | |
|---|---|
| List | As the name implies, it is a list of values. The values in a list are counted from zero onwards (the first value is numbered zero, the second 1st, and so on). List lets values to be removed and added at will. |
| Tuples | They are similar to lists except their values cannot be modified. Once created, the values remain static for the rest of the program. Again, the values are numbered for reference, starting with zero. |
| Dictionaries | Like a normal dictionary, it allows you to create an index of words where each word has a unique definition. In Python, the word is called the ‘key’ whereas its definition is called its ‘value’. Like a dictionary, none of the words/keys are numbered. The values in the dictionary can be created, removed, and modified. |

Let’s start with the unchanging tuples.

## Using Tuples

Tuples are easy to create. Name your tuple and list the values that it will carry. Here's a tuple for carrying the months of the year:

```
months = ('Jan', 'Feb, 'Mar', 'Apr', \ 'May',' Jun', 'Jul', 'Aug, 'Sept, 'Oct, \ 'Nov,' 'Dec')
```

Syntactically, a tuple is a comma-separated sequence of values. The parenthesis and the space after the comma are simply a convention, and not necessary for creating a tuple. Furthermore, notice the '\' at the end of each line? It carries the line to the next line, making big lines more readable.

Once created, Python creates a numbered index to organize the values in a tuple. Starting from zero, the values are indexed in the order you entered them in the tuple. The above tuple becomes:

| Index | Value |
| --- | --- |
| 0 | January |
| 1 | Feb |
| 2 | Mar |
| 3 | Apr |
| 4 | May |
| 5 | Jun |
| 6 | Jul |
| 7 | Aug |
| 8 | Sep |
| 9 | Oct |
| 10 | Nov |
| 11 | Dec |

So if you were to call the tuple ‘month’, you will use the index to call it:

```
>>>month [2]
Mar
```

Additionally, Python has a very powerful tuple assignment feature that allows us to assign value to variables on the left with values on the right. Hence, for the ‘months’ we have created earlier, we can further assign values to each of the value in the list:

```
>>>(research, submit outline, discussion, study, seminars, presentation, field trip, submit paper, panel discussion, final presentation, semester ends) = months
```

The only requirement is that the number of variables on the left must equal the number of elements declared in the tuple.

A simple way of looking at this assignment is to to think of it as tuple packing/unpacking. When packed, the values on the left are packed together:

```
>>>  months = (‘Jan’, ‘Feb, ‘Mar’, ‘Apr’, \ ‘May’,’ Jun’, ‘Jul’, ‘Aug, ‘Sept, ‘Oct, \ ‘Nov,’ ‘Dec’)
```

In tuple unpacking, the *values* on the right (the names of the months) are unpacked into *variables/names/categories* on the right:

```
>>>  months = (‘Jan’, ‘Feb, ‘Mar’, ‘Apr’, \ ‘May’,’ Jun’, ‘Jul’, ‘Aug, ‘Sept, ‘Oct, \ ‘Nov,’ ‘Dec’)
>>>(research, submit outline, discussion, study, seminars, presentation, field trip, submit paper, panel discussion, final presentation, semester ends) = months
>>> research
Jan
>>> seminars
May
>>> submit outline
Feb
```

Another powerful use of tuple is when you have to swap the values of two variables. Normally, you would have to use a temporary variable for swapping:

```
#Swap ‘b’ with ‘a’
temp = b
b = a
a = temp
```

A tuple resolves this in a single line:

```
(b, a) = (a, b)
```

Simple. The right hand side is a tuple of values while the left hand side is a tuple of variables. Naturally, the number of values much always matches the number of variables:

```
>>> (a, b, c, d) = (4, 3, 2)
```

```
ValueError: need more than 3 values to unpack
```

# Using Lists

Lists are similar to tuples: they store a range of values and are defined in a similar fashion. However, unlike tuples, you can modify them. This makes them the normal choice when it comes to storing lists. Here's an example of a list:

```
team =[ 'Sam', 'Michel', 'Azazel', 'Harrison']
```

Notice that the only difference in syntax is the use of *square brackets* instead of parentheses. Like tuples, the spaces after the comma are a standard practice for increasing readability.

Recalling the values stored in the list are also similar to calling values in a tuple:

```
print team [3]
```

```
Harrison
```

Like tuples, you can also call values from a range within the list. For example, team [1:3] will recall 3rd and 4th members of the team.

The important thing with lists is its ability to allow change. This is crucial when you are building databases that store values (e.g. a grocery store's inventory will have changing values of the stock and need to be updated regularly).

Let's say you induct another member into your team, how will you add him/her?

Values can be added using the 'append() ' function. The syntax of the append function is of the form:

```
list_name.append(value-to-add)
```

Hence, for a new member, Gabriel:

```
team.append('Gabriel')
```

And done! The new member's name is added after the last value stored in the list ('Harrison').

How do you remove an item from a list? Suppose Michel is not getting along with Azazel and Gabriel, and is lowering the moral of the team, etc.

To delete a value from the list, you use 'del'. Recall how Python indexes the lists, beginning from zero and onwards. So Michel is the second value on the list, making its index numbering '1'.

```
#Removing Michel from the list
```

```
team =[ 'Sam', 'Michel', 'Azazel', 'Harrison', Gabriel]
```

```
del team[1]
```

You can delete a range from the list by assigning an empty list to them:

team [1:3] = []

Now the team the last three names removed from it.

What if you wanted to add a new team member right after Michel? Normally, append[ ] adds the new value at the end of the list.

You simply tell after which member the new member should be placed. This is called slicing the list:

```
>>>team =[ ‘Sam’, ‘Michel’, ‘Azazel’, ‘Harrison’]
>>>team [1:1] = [‘Gabriel’]
team =[ ‘Sam’, ‘Michel’, ‘Gabriel’, ‘Azazel’, ‘Harrison’]
```

### *METHODS THAT CAN BE USED WITH LISTS*

.append is just one of the several methods that are extensively used with creating lists. Other methods include the following:

**.insert** it is used for posting a new entry at the specified index number. For example, want to add numbers to your list of team:

```
>> team.insert(1, 3320)
```

This inserts the number 3320 at position 1, shifting the other values up i.e. 2nd becomes 3rd and so on.

If you want to repeat the list within itself, then you will simply use **.extend** in it.

```
>>> team.extend ([‘Sam’, ‘Michel’, ‘Azazel’, ‘Harrison’])
>>> mylist
[‘Sam’, ‘Michel’, ‘Azazel’, ‘Harrison’, ‘Sam’, ‘Michel’, ‘Azazel’, ‘Harrison’]
If you want to find the index number of any value in the list use .index
>>> team.index(3)
‘Harrison’
Reverse the whole list using .reverse
>>> team.reverse()
[‘Harrison’, ‘Azazel’, ‘Michel’, ‘Sam’]
Remove the a repetitive item or the first use of any item using .remove
>>> team.remove(‘Sam’)
[‘Michel’, ‘Azazel’, ‘Harrison’]
Or in case you have a number list, and you want to sort them in ascending order, use .sort
>>> numlist.sort()
[1, 3, 3, 3, 7, 9, 10, 10, 24]
```

# Using Dictionaries

Previously we have created lists of names of a team and a tuple with variable and value assignment. However, in both of them, the *value* of the indexed variable can only be called by giving the index number for that value.

What if you want to create a database, or a small phonebook, that gives you the details of a variable when you *enter its name* instead of the index number? Lists and tuples cannot give you the required accessibility.

Dictionaries can.

Recall that dictionaries have keys and values. In a phone book, you have names of the people and their contact details. See any similarities?

Creating a dictionary is similar to making a list or a tuple, except a slight difference in its brackets.

- (Tuples) use parenthesis
- [Lists] use square brackets
- {Dictionaries} use curly braces.

Here's an example of a database for money owed to each member your business team:

```
#Initial business funds:
Logbook = {'Sam Kim': 4000, 'Michel Sanderson': 4300, \
Stark Garret': 5120, ' Azazel Forest': 3230, 'Harrison Snow': 6300 }
```

Notice the syntax:

```
Key: Value
```

Here is how the keys are used to *look up* the corresponding value, just like in a dictionary:

```
>>> print(logbook["Azazel Forest"])
3230
```

## *Dictionary operations*

You can add new key:value pairs in the dictionary as well as remove and update existing dictionary entries.

## *Adding new Entries to Dictionaries*

To add new entries in your existing dictionary, you simply define them as follows:

```
#Adding Gabriel to the logbook:
logbook['Gabriel Sky'] = 7300
The above states that the key : value = 'Gabriel Sky' : 7300
```

## *Deleting Entries*

Now what if you want to delete some entries? This is done exactly how it was done for lists. Now let's say Michel has been paid in full, and he has resigned. You want to delete his account permanently. Just like with the lists, you use 'del':

```
del phonebook['Michel Sanderson']
```

The 'del' operator will delete any variable, entry, or function in a list or a dictionary. Another example is of a small inventory in a grocery store. The dictionary contains the names of various fruits and their availability (number in stock):

```
>>> inventory = {"apples": 350, "bananas": 230, "Mangos": 100, "Peaches": 250}
```

```
>>> print(inventory)
```

Now what if Mangos go out of stock? We have two options: of deleting the key:value or simply change the value of the key:

```
#deleting the value
```

```
>>> del inventory["mangos"]
```

```
>>> print (inventory)
```

```
{"apples": 350, "bananas": 230, "Peaches": 250}
```

In case the store is receiving more stock, the we need the option to simply update the value with a new value:

```
   >>> inventory["Mangos"] = 0
```

```
   >>> print(inventory)
```

```
{"Mangos": 0, "apples": 350, "bananas": 230, "Peaches": 250}
```

Let's say we have a new shipment for Mango within the hour, and it will add 150 additional Mangos to the inventory. This can be handled like this:

```
>>> inventory ["Mangos"] += 150
```

```
>>> print (inventory)
```

```
{"Mangos": 150, "apples": 350, "bananas": 230, "Peaches": 250}
```

Tuples, lists, and dictionaries play an important role in writing simpler and more powerful codes in Python. They become even more important when we start programming using and interfacing objects.

# Chapter 9. Object Oriented Programming

Historically, a code has mostly been viewed as a logical procedure where data is taken as input, processed, and used to produce output data. Consequently, the challenge of programming was seen how to create logic for using the data and not defining the data.

Object-Oriented Programming (OOP) is a programming language model that relies on building code using objects and data rather than “actions” and “logic”. It takes the view that what we should really care about (and which programmers always have) are objects that we can manipulate instead of the sequential logic needed to manipulate them.

Object-oriented programming has its roots in the 1960s, however it became the dominant paradigm of programming as the complexity and size of the software began to increase rapidly, giving way to more powerful and complex systems.

Examples range from electronic devices (who properties and attributes can be named), to human beings (described in terms of their attitudes and properties), all the way to the apps we use on our smartphones.

## *Getting Started with OOP*

The first step in applying OOP (which you must because Python is an object oriented programming language) is to identify the *objects* that are needed to build the code, and how they relate to each other, and which you want to manipulate.

Though it is called data modeling, think of it like building a strategy or a blueprint for creating something.

In Python, once you have identified and object, a generalized class of objects in its name is created. This class defines the kind/type of data this object contains and any logic sequences which it can manipulate.

The logic sequence used for each class is known as its method, whereas the interfaces through which the objects communicate with one another are called messages.

## *Benefits of using OOP*

This framework offers newer ways of programming in Python. Important benefits include:

Data classes allows a programmer greater flexibility and creativity in creating new types of data for use with the program. Even if certain data types are not available in Python, you can easily create them as a separate object and use them like functions (crudely speaking) inside your code.

The data class makes it possible to easily define subclasses of data objects that can either share all the properties of the main class characteristics or some of it. This property is called inheritance where the subclass inherits properties and attributes. This ability of the OOP framework allows more robust analysis of data, allows more accurate coding, and significantly reduces development.

OOP allows data hiding which prevents accidental sharing of data across the code. Since each class has a predefined set of attributes and data properties that it must be concerned with, any instances of this object where used throughout the code will only use its specified data, hence avoiding any instances of data corruption.

Classes once defined are easily re-useable. The object once created becomes global, in the send that it can be used by both, the program for which it was created and by other OOP codes using the same machine/network.

# Chapter 10. Classes, Objects, Inheritance

Programmers are supposed to be lazy. They have to find the most efficient way of doing things, saving time for more coding or to do what they will.

The purpose of programming is to simply and automate actions and hence avoid repeating writing the same code again. Recall using functions. Once you have created a function, you can call it and re-use it anywhere in the code and the interpreter will know what you are asking for.

However, functions have limitations. They use data but cannot store and hold information. Every time they are called, they start afresh. Now at times certain functions and variables are related to one another very closely, and whenever the function is run, those values need to be fetched and computed alongside new data.

But what if we need a function to generate multiple outputs and values instead of just one output? What you need here is the ability to group multiple functions and associated variables in one place so that they can easily interact with one another.

Take the example of your team.

All of them have joined a golf course and bought new golf clubs.

Your program will store different variables regarding the clubs (their shaft length, weight, forgiveness, spin, etc.), and you want to keep track of the impact each property experiences over time (weakened shaft, increased frustration, scratches, etc.). Suppose you make functions for each team member, what if they decide to have more than one club?

Will you write a whole chunk of code for each different golf club?

Given that most of the clubs share common features, the ideal thing to do would be to create a basic category, or an ideal category that defines all the attributes of the golf club. Hence, whenever you create a new club, all you have to do is specify the new or changed attributes and a new item will be created in the database.

This is where classes and objects come into play. They allow you to create small independent ‘communities’ where functions and variables can interact together, can easily be modified as needed, and remain unchanged (and unaffected by other code).

We start building such objects by first creating classes.

## Creating a Class

A class is a blueprint, an idea of something. It does not exist as a usable function, rather it *describes* how to make something. This blueprint can be used to create a lot of objects.

You can create a class using a class operator. The general syntax for a class is:

```
class name_of_my_class:
    [statement 1]
    [statement 2]
    [etc]
```

Here’s an example

```
class new_shape:
    def __init__(self, x, y):
    self.x = x
    self.y = y
    shape_details = “This is an undescribed shape”
    creator = “No one lays claim to creating this shape “
    def creatorName(self,text):
    self.creator = text
    def detail(self,text):
    self.shape_details = text
def perimeter(self):
    return 2 * self.x + 2 * self.y
def area(self):
    return self.x * self.y
    def scaleSize(self,scale):
    self.x = self.x * scale
    self.y = self.y * scale
```

The above is a vague description of a shape (square or rectangle).

The description is defined using the variables you have used for the shape , and using the functions you have defined the operations the new class ‘new_shape” can be use for. But you have not created any actual shape. It is simply a description that needs values to define how it would look (the height ‘x’ and width ‘y’), and which will define its properties (area and perimeter).

Wondering what ‘self’ is all about? Recall that you have not created an actual shape yet. ‘self’ is how things are referred to in the class from within itself. It is a parameter that does not pass any value to the function. As a result, the interpreter does not run the code when a

class is defined. It considered as a practice in making new functions and passing variables.

All the functions/variables that are created on the first indentation level (the line of code indented right after) are automatically put into self. If a parameter or a function inside the class has to be used within that class, then the name of the function must be proceeded with a self-dot (**self.**) e.g. self.x as used in the previous code.

### *Using a Class*

We can create a class. So, now the question is: how do you use the magical blueprint to create an actual shape?

We use the function _init_ to create an instance of new_shape.

Assuming that the previous code has been run, lets create a shape:

rectangle = new_shape(80,40)

This is where __init__ function comes into play. We have already created a class (a blueprint normally called an instance for the class). We did this by giving it:

- A name (new_shape)
- Values in brackets, which are passed pass to the __init__ function.

Using the parameters given to the init functions, the init function generates an instance of that class, assigning it the name rectangle

Now, this new instance of our new_shape class has become a self-contained collection of functions and variables (we will discuss inheritance later). Earlier we were using **self.** to access the variables and functions defined in that class because we were accessing it from within itself. Now, we will use the name we have assigned to its tangible form (rectangle). We will access the variables and functions in the class from outside.

Once the above code has run, we can access the attributes of our class to create a shape with the following code:

```
#Calculating the perimeter of the rectangle:
print rectangle.perimeter()
#Calculating the area of the rectangle:
print rectangle.area()
#details about the rectangle
rectangle.describe(“A rectangle that is twice as long as it is wide. “)
#Scaling the rectangle to half its size, i.e. making it 50% smaller
rectangle.scaleSize(0.5)
```

Notice how the assigned name of the new object is being used when the class is being used outside of itself. Whereas, when it was being used from within itself the **self.** operator was being used.

When used from outside of itself, we can easily change the value of the variables inside the class and access its functions. Think of the class like a factory that assembles new products when given the right input. When using classes, we are not limited to a single instance. We can have multiple of them. For instance, could create another object named thin, long, and sturdy rectangles:

thinrectangle = new_shape(100,5)

longrectangel =new_shape (100,20)

sturdyrectangle = new_shape(100,90)

Even when we have created three instances, all of which relied on the variables and functions from the same class, ALL OF THEM are completely independent of one another and can be used countless times throughout the program.

## *Appreciating the Geek Talk*

The Object-oriented-programming is a framework with its specific set of words used to describe a programming action. You should know the basic lingo to avoid any confusion in company of another programmer. Here are some basic words:

- Describing a class means defining it (similar to functions, albeit more detailed)
- Encapsulation is the grouping of similar functions and variables under a single class
- Class itself can be used in two instances, to describe the chunk of code that defines a class and the instance where the class is used to create a new object.
- A class is also known as a ‘data structure’. It can hold data and has the methods to process data provided to it
- The attribute of a class are the variables inside them
- A method is the function you have defined inside the class
- A class itself is an object, and in the same category of things such as dictionaries, variables, lists, etc.

# INHERITANCE

We've talked about inheritance in functions and earlier in the same chapter. What is it and why is it important? Let's first recap what we have done so far in terms of functions, variables, and creating classes.

Earlier you saw how we can group a diverse range of variable and functions together. This allows the data and the processing code to be present in the same spot, making it easier to read the code as well to execute it across the code from a single spot. Now, the attributes we granted to the class (variables) and the methods it can use to process the data (the functions), allows us to create innumerable instances of that class without writing new code for every new object that we create.

What if we want to cater an anomaly in our database? Imagine if we have a shape with additional features, ones that does share common features with the original class but has additional features that the original class cannot process?

Creating a new code to cater differences is not the right way to handle it. This is where we make a child class that inherits all the properties of the ideal class while adding its own new features to it.

This is where inheritance comes into play, and which Python makes exceptionally easy to implement.

How does it work?

We define a new class, using the existing class as its 'parent' class. Consequently, the child class takes everything from the parent class while allowing us to add new features and attributes and methods to it.

Let's build on the new_shape class:

```
Here's an example
class new_shape:
    def __init__(self, x, y):
    self.x = x
    self.y = y
    shape_details = "This is an undescribed shape"
    creator = "No one lays claim to creating this shape "
    def creatorName(self,text):
    self.creator = text
    def detail(self,text):
    self.shape_details = text
def perimeter(self):
```

```
    return 2 * self.x + 2 * self.y
def area(self):
    return self.x * self.y
    def scaleSize(self,scale):
    self.x = self.x * scale
    self.y = self.y * scale
```

Now square is also considered a form of the rectangle, however its width and length are equal to one another. Now if we want to define a new class using the existing new_shape class, the code would look like this:

```
class square(new_shape):
    def __init__(self,x):
    self.x = x
    self.y = x
```

This is quite similar to how we defined the new_shape class, except that we have used the parent class as the set of parameters that need to be inherited. Notice how easy it was to cater this new object without having to redefine and write its code separately.

We changed only what needed to be changed, borrowing the rest. We have merely redefined __init__ function of new_shape so that the height and width (x and y) become the same. However, the new variables defined in the child class will never overwrite the ones present in new_shape.

Now, square itself has become a new class and we can use it to create another class as well! You see a pattern here? Its like a lego puzzle, you create objects and you use them as bricks to build better and more complex objects and programs.

Let’s create a double square, so that two squares are created side by side:

```
class 2square(Square):
    def __init__(self,y):
    self.y = y
    self.x = 2 * y
    def perimeter(self):
    return 2 * self.x + 3 * self.y
```

Notice that this time we have an additional method (a function) in the code as well. We have redefined the perimeter function because the one that square has inherited form new-shape cannot cater to the needs of this new **class**.

If you create an *instance of* 2square, your double squares will all have the same attributes and properties defined by the 2square class and not the square or new_shape.

### *Pointers and Dictionaries of Classes*

Previously, we had seen how variable swapping works, e.g. var1=var2 would swap the left hand side variable with the value stored in the right hand side variable.

The same does not hold true when it comes to creating class instances. When you write instance1=instance2, what is actually happening is that the first class is pointing to the class on the right. “Pointing” means that both the *names of instances* refer to the same class instance, and that the same class instance can be used by either name.

This brings us to dictionaries of classes.

Building on how pointers work, Python lets us easily assign instance of class to an entry in a list or dictionary as well. But why do it? What’s the benefit?

It allows us to create virtually unlimited number of class instances to run from our program. Here’s an example of using pointers to create dictionary instances. Assuming the original definition of new_shape, and that the square and 2sqaure classes have been run:

```
#Create a dictionary:
dictionary = {}
#Next, create some instances of classes in the dictionary:
dictionary[“2square 1”] = 2square(5)
dictionary[“long rectangle”] = new_shape(100,30)
```

Now you can use them like normal classes!

```
dictionary[“2square 1”].creatorName(“Python Coder”)
print dictionary[“2square 1”].author
```

```
print dictionary[“long rectangle”].area()
```

Now, you have replaced the previous name we had created for our creation with an arguably better name by creating a new dictionary entry.

# Chapter 11. Debugging

Codes remind us of the importance of paying attention in English grammar lessons, and how our overreliance on auto-correct features in word processing software has made us susceptible to overlooking small mistakes — a comma here, a colon there, and a misspelled variable there, among others.

Mistakes, weather small or large can pull the brakes on a program. As a result, your code will not run at all, display errors, or give obnoxious results. This is where you have to troubleshoot the program, locate the problem, and correct it.

In programming lingo, that’s called debugging your program.

Small mistakes such as wrong syntax or declaring non-existing variables, or more fundamental mistakes such as insufficient understanding of the problem (and hence the objects that need to be created and their interaction), and incorrect flow of data can make the code dysfunctional.

Given that only you can best understand your code, you will often find yourself in a position where you alone are right person to save your code. However, decades of testing have given us a few tips and tricks for debugging your code.

However, before we move into the tips, it is important that you download your first debugger.

# Download a Debugger

Why?

Python already ships in with its built-in debugger.

Yes, Python comes equipped with a debugger, sporting a basic command line interface and with various features and options. However, it lacks certain features that will simplify debugging for beginners, including stack traces and differentiating coloring for different types of code.

So, here's a better option for a debugging tool.

Download Pudb 2015.3, a full screen console based program debugger. It has a modern GUI-based environment, and allows you to debug your code right where you write it: on the terminal.

# Chapter 12. Next up

This eBook is meant to introduce you to the basics of Python language: its fundamental building blocks, their syntax, their basic uses, and how they are related to one another in creating larger programs. However, like any other language, the only way to improve and excel at it is to consciously try to use it to solve problems.

Having introduced the basic building blocks of the language, it’s now like chess: you know the components, you know the basic rules, and you know the moves (both legal and illegal). Reading more about it will not help you on the board. Only practice will.

So what should you do next?

Python is a language for communicating with computing devices, and most of what you will be writing will never be visible on the front end. I mean, how many apps have you used, or games have you played where you know the code line you’re on right now?

Understand that you have only scratched the surface. As you start building code with the basics, you will come across advanced codes that you can build on. Continue practicing and sharing your questions with a blossoming online community of Python codes and fans.

Python is one of the most advanced and one of the simplest OOP languages on the planet. Be patient, and master the basics before moving towards the more advanced features of the language.

# About The Author

Inspired by legendary female coder, Augusta Ada, Vivian has been in the software engineering since she was in college at the University of Sydney, Australia. While studying at UofS, Vivian took the path most females would not study. She enrolled intending to major in Architecture, but after 2 semesters she began to design websites for local charities. Her love of the web took off and switched major to Computer Science and Information.

Since graduating in 2012, Vivian as has gone on to do a wide variety of freelance projects and sub-contract work in mobile and web development. She often volunteers in workshops and training programs to help those interested in learning all about software development.

Vivian travels the world and works from her laptop most of the time. She is interested in helping others learn how to program and start new careers in the web industry.

# One Last Thing…

If you enjoyed this book or found it useful I'd be very grateful if you'd post a short review on Amazon. Your support really does make a difference and I read all the reviews personally so I can get your feedback and make this book even better.

Thanks again for your support!