

1.About Python

1.1 Python Overview

- Open source
- General purpose
- Easy to interface with C/objC/Java/Fortran/C++
- Great interactive environment

Website and downloads: www.python.org

Documentation: www.python.org/doc/

1.2 History of Python

Innovative languages are mostly a product of either, a large well-funded research project or,frustration out of the lack of tools that were needed at the time to accomplish mundane or time taking tasks, where most of them could be automated could be automated.

- Conceived in the late 1980s
- Work on python began in 1989 by Guido van Rossum, at CWI in the Netherlands
- Made as a successor to ABC capable of exception handling and interfacing with the Amoeba operating system
- Released for public distribution in February 1991 labeled version 0.9.0,
- Already present at python version 0.9.0 in development were core features classes with inheritance, exception handling, functions, and the core data types of “list”, “dict” and “str” and so on.

1.3 Features of Python

- High level
- Object oriented
- Scalable
- Extensible
- Portable
- Easy to learn
- Easy to read
- Easy to maintain
- Robust
- Effective as a rapid prototyping tool
- A memory manager
- Interpreted and byte compiled

1.4 Python Environment

Before we start writing our programs in python , it's important to know how to set up a Python environment. Python is available on a wide Number of platforms. Open a terminal window and type "python" to check if its already installed and which version you have if it is already installed.

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, et al.)
- Win 9x/NT/2000 (Windows 32-bit systems)
- Macintosh (PPC, 68K)
- OS/2
- DOS (multiple versions)
- Windows 3.x
- PalmOS
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python is also ported to Java and .Net VM's

1.5 Getting Python

For the most up-to-date and current source code, binaries, documentation, news, etc., check either the main Python language site or the PythonLabs Web site:

`http://www.python.org` (community home page)

`http://www.pythonlabs.com` (commercial home page)

If you do not have access to the Internet readily available, all three versions (source code and binaries) are available on the CD-ROM in the back of the book. The CD-ROM also features the complete online documentation sets viewable via offline browsing or as archive files which can be installed on hard disk. All of the code samples in the book are there as well as the Online Resources appendix section (featured as the Python "hotlist").

1.6 Case studies

Exercise

2. Context of Software Developments

A computer program, from one perspective, is a sequence of instructions that dictate the flow of electrical impulses within a computer system. These impulses affect the computer's memory and interact with the display screen, keyboard, and mouse in such a way as to produce the "magic" that permits humans to perform useful tasks, solve high-level problems, and play games. One program allows a computer to assume the role of a financial calculator, while another transforms the machine into a worthy chess opponent. Note the two extremes here:

- at the lower, more concrete level electrical impulses alter the internal state of the computer, while
- at the higher, more abstract level computer users accomplish real-world work or derive actual pleasure.

So well is the higher-level illusion achieved that most computer users are oblivious to the lower-level activity (the machinery under the hood, so to speak). Surprisingly, perhaps, most programmers today write software at this higher, more abstract level also. An accomplished computer programmer can develop sophisticated software with little or no interest or knowledge of the actual computer system upon which it runs. Powerful software construction tools hide the lower-level details from programmers, allowing them to solve problems in higher-level terms.

The concepts of computer programming are logical and mathematical in nature. In theory, computer programs can be developed without the use of a computer. Programmers can discuss the viability of a program and reason about its correctness and efficiency by examining abstract symbols that correspond to the features of real-world programming languages but appear in no real-world programming language. While such exercises can be very valuable, in practice computer programmers are not isolated from their machines. Software is written to be used on real computer systems. Computing professionals known as software engineers develop software to drive particular systems. These systems are defined by their underlying hardware and operating system. Developers use concrete tools like compilers, debuggers, and profilers.

2.1 Software

A computer program is an example of computer software. One can refer to a program as a piece of software as if it were a tangible object, but software is actually quite intangible. It is stored on a medium. A hard drive, a CD, a DVD, and a USB pen drive are all examples of media upon which software can reside. The CD is not the software; the software is a pattern on the CD. In order to be used, software must be stored in the computer's memory. Typically computer programs are loaded into memory from a medium like the computer's hard disk. An electromagnetic pattern representing the program is stored on the computer's hard drive. This pattern of electronic symbols

must be transferred to the computer's memory before the program can be executed. The program may have been installed on the hard disk from a CD or from the Internet. In any case, the essence that was transferred from medium to medium was a pattern of electronic symbols that direct the work of the computer system.

These patterns of electronic symbols are best represented as a sequence of zeroes and ones, digits from the binary (base 2) number system. An example of a binary program sequence is

```
10001011011000010001000001001110
```

To the underlying computer hardware, specifically the processor, a zero here and three ones there might mean that certain electrical signals should be sent to the graphics device so that it makes a certain part of the display screen red. Unfortunately, only a minuscule number of people in the world would be able to produce, by hand, the complete sequence of zeroes and ones that represent the program Microsoft Word for an Intel-based computer running the Windows 7 operating system. Further, almost none of those who could produce the binary sequence would claim to enjoy the task.

The Word program for older Mac OS X computers using a PowerPC processor works similarly to the Windows version and indeed is produced by the same company, but the program is expressed in a completely different sequence of zeroes and ones! The Intel Core 2 Duo processor in the Windows machine accepts a completely different binary language than the PowerPC processor in the Mac. We say the processors have their own machine language.

2.2 Development Tools

If very few humans can (or want) to speak the machine language of the computers' processors and software is expressed in this language, how has so much software been developed over the years?

Software can be represented by printed words and symbols that are easier for humans to manage than binary sequences. Tools exist that automatically convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like Python allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Some examples of the more popular of the hundreds of higher-level programming languages that have been devised over the past 60 years include FORTRAN, COBOL, Lisp, Haskell, C, Perl, C++, Java, and C#. Most programmers today, especially those concerned with high-level applications, usually do not worry about the details of underlying hardware platform and its machine language.

One might think that ideally such a conversion tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called compilers that translate one computer language into another have been around for 60 years, but natural language processing is still an active area of artificial intelligence research. Natural languages, as they are used

by most humans, are inherently ambiguous. To understand properly all but a very limited subset of a natural language, a human (or artificially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today's software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any program that can be solved by a computer.

Consider the following program fragment written in the Python programming language:

```
subtotal = 25  
tax = 3  
total = subtotal + tax
```

These three lines do not make up a complete Python program; they are merely a piece of a program. The statements in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, subtotal, tax, and total, called variables, are used to hold information. Mathematicians have used variables for hundreds of years before the first digital computer was built. In programming, a variable represents a value stored in the computer's memory. Familiar operators (= and +) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the Python language, not machine language, it cannot be executed directly on any processor. A program called an interpreter translates the Python code into machine code when a user runs the program.

The higher-level language code is called source code. The interpreted machine language code is called the target code. The interpreter translates the source code into the target machine language.

The beauty of higher-level languages is this: the same Python source code can execute on different target platforms. The target platform must have a Python interpreter available, but multiple Python interpreters are available for all the major computing platforms. The human programmer therefore is free to think about writing the solution to the problem in Python, not in a specific machine language.

Programmers have a variety of tools available to enhance the software development process. Some common tools include:

Editors. An editor allows the programmer to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colors to highlight language features. The syntax of a language refers to the way pieces of the language are arranged to make well-formed sentences. To illustrate, the sentence

The tall boy runs quickly to the door.

uses proper English syntax. By comparison, the sentence

Boy the tall runs door to quickly the.
is not correct syntactically. It uses the same words as the original sentence, but their arrangement does not follow the rules of English.

Similarly, programming languages have strict syntax rules that must be followed to create well formed programs. Only well-formed programs are acceptable and can be compiled and executed. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors before the program is compiled.

Compilers. A compiler translates the source code to target code. The target code may be the machine language for a particular platform or embedded device. The target code could be another source language; for example, the earliest C++ compiler translated C++ into C, another higher-level language. The resulting C code was then processed by a C compiler to produce an executable program. (C++ compilers today translate C++ directly into machine language.)

Interpreters. An interpreter is like a compiler, in that it translates higher-level source code into machine language. It works differently, however. While a compiler produces an executable program that may run many times with no additional translation needed, an interpreter translates source code statements into machine language as the program runs. A compiled program does not need to be recompiled to run, but an interpreted program must be interpreted each time it is executed. In general, compiled programs execute more quickly than interpreted programs because the translation activity occurs only once. Interpreted programs, on the other hand, can run as is on any platform with an appropriate interpreter; they do not need to be recompiled to run on a different platform. Python, for example, is used mainly as an interpreted language, but compilers for it are available. Interpreted languages are better suited for dynamic, explorative development which many people feel is ideal for beginning programmers.

Debuggers. A debugger allows programmers to simultaneously run a program and see which source code line is currently being executed. The values of variables and other program elements can be watched to see if their values change as expected. Debuggers are valuable for locating errors (also called bugs) and repairing programs that contain errors. (See Section 3.4 for more information about programming errors.)

Profilers. A profiler is used to evaluate a program's performance. It indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Profilers also can be used for testing purposes to ensure all the code in a program is actually being used somewhere during testing. This is known as coverage. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing. The main purpose of profiling is to find the parts of a program that can be improved to make the program run faster.

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Examples of commercial IDEs include Microsoft's Visual Studio 2010, the Eclipse Foundation's Eclipse IDE, and Apple's XCode. IDLE is a very

simple IDE for Python.

3. Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python. If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

3.1 Unix & Linux Installation

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to www.python.org/download/
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the *Modules/Setup* file if you want to customize some options.
- **run** `./configure` script
- **make**
- **make install**

This will install python in a standard location `/usr/local/bin` and its libraries are installed in `/usr/local/lib/pythonXX` where XX is the version of Python that you are using.

3.2 Windows Installation

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to www.python.org/download/
- Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you are going to install.
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Just save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file by double-clicking it in Windows Explorer. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you're ready to roll!

3.3 Macintosh Installation

Recent Macs come with Python installed, but it may be several years out of date. See www.python.org/download/mac/ for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available."

Jack Jansen maintains it and you can have full access to the entire documentation at his Web site - **Jack Jansen**

Website : <http://www.cwi.nl/~jack/macpython.html>

3.4 Setting up a path

Programs and other executable files can live in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables. The path is stored in an environment variable, which is a named string maintained by the operating system. These variables contain information available to the command shell and other programs. The **path** variable is named **PATH** in Unix or **Path** in Windows (Unix is case-sensitive; Windows is not). In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

3.4.1 Setting up a path at Unix/Linux

To add the Python directory to the path for a particular session in Unix:

- **In the csh shell:** type

`setenv PATH "$PATH:/usr/local/bin/python"` and press Enter.

- **In the bash shell (Linux):** type

`export PATH="$PATH:/usr/local/bin/python"` and press Enter.

- **In the sh or ksh shell:** type

`PATH="$PATH:/usr/local/bin/python"` and press Enter.

Note: `/usr/local/bin/python` is the path of the Python directory

3.4.2 Setting up a path at Windows

To add the Python directory to the path for a particular session in Windows:

- **At the command prompt :** type

`path %path%;C:\Python` and press Enter.

Note: `C:\Python` is the path of the Python directory

3.5 Python Environment Variables

Here are important environment variables, which can be recognized by Python:

Variable	Description
PYTHONPATH	Has a role similar to PATH. This variable tells the Python interpreter where to locate the module files you import into a program. PYTHONPATH should include the Python source library directory and the directories containing your Python source code. PYTHONPATH is sometimes preset by the Python installer.
PYTHONSTARTUP	Contains the path of an initialization file containing Python source code that is executed every time you start the interpreter (similar to the Unix .profile or .login file). This file, often named .pythonrc.py in Unix, usually contains commands that load utilities or modify PYTHONPATH.
PYTHONCASEOK	Used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.
PYTHONHOME	An alternative module search path. It's usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

4. Running Python Programs

There are three different ways to start Python:

4.1 Interactive Interpreter

You can enter **python** and start coding right away in the interactive interpreter by starting it from the command line. You can do this from Unix, DOS or any other system, which provides you a command-line interpreter or shell window.

```
$python # Unix/Linux
or
python% # Unix/Linux
or
C:>python # Windows/DOS
```

Here is the list of all the available command line options:

Option	Description
-d	provide debug output
-O	generate optimized bytecode (resulting in .pyo files)
-S	do not run import site to look for Python paths on startup
-v	verbose output (detailed trace on import statements)
-X	disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6
-c cmd	run Python script sent in as cmd string
file	run Python script from given file

4.2 Script from the command line

A Python script can be executed at command line by invoking the interpreter on your application, as in the following:

```
$python script.py # Unix/Linux
or
python% script.py# Unix/Linux
or
C:>python script.py# Windows/DOS
```

Note: Be sure the file permission mode allows execution.

4.3 Integrated development Environment(IDE)

You can run Python from a graphical user interface (GUI) environment as well. All you need is a GUI application on your system that supports Python.

- **Unix:** IDLE is the very first Unix IDE for Python.
- **Windows:** PythonWin is the first Windows interface for Python and is an IDE with a GUI.
- **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

Before proceeding to next chapter, make sure your environment is properly set up and working perfectly fine. If you are not able to set up the environment properly, then you can take help from your system admin.

All the examples given in subsequent chapters have been executed with Python 2.4.3 version available on CentOS flavor of Linux.

5. Python Basic Syntax

Python is an interpreted language which means it is executed sequentially from the first line to the last line, and of course control flow can be changed by control flow statements.

Sample python code

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Feb 16 19:00:02 2019
4
5 @author: khalid
6 """
7 x = 37 ** 2 - 23          # A comment.
8 y = 'Hi'                 # Another one.
9 z = 3.14
10 if z == 3.14 or y == 'Hi!':
11     x = x + 1
12     y = y + ' World'     # Concatenating strings.
13 print(x)
14 print(y)
```

5.1 First Python Program

There are two ways you can take to run your python code.

1. Interactive mode programming: Using the python shell to run single a statement at a time
2. Script mode programming: Running your python script as a whole.

Interactive Mode Programming: (Using the Python shell)

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

```
(base) C:\Users\gmkha>python
Python 3.6.3 |Anaconda custom (64-bit)| (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Type the following text to the right of the Python prompt and press the Enter key:

```
>>> print("Hi World!")
```

this will produce following result:

```
Hi World!
```

Script Mode Programming:

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. All python files will have extension **.py**. So put the following source code in a test.py file.

```
print "Hello, Python!";
```

Here, I assumed that you have Python interpreter set in PATH variable. Now, try to run this program as follows:

```
$ python test.py
```

This will produce the following result:

```
Hello, Python!
```

Let's try another way to execute a Python script. Below is the modified test.py file:

```
#!/usr/bin/python  
print "Hello, Python!";
```

Here, I assumed that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows:

```
$ chmod +x test.py # This is to make file executable  
$ ./test.py
```

This will produce the following result:

```
Hello, Python!
```

5.2 Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, \$ and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are following identifier naming convention for Python:

- Class names start with an uppercase letter and all other identifiers with a lowercase letter.
- Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

5.3 Reserved words

The following list shows the reserved words in Python. These reserved words may not be used as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

And	Exec	Not
Assert	Finally	Or
Break	For	Pass
Class	From	Pass
Continue	Global	Raise
Def	If	Return
Del	Import	Try
Elif	In	While
Else	Is	With
Except	Lambda	Yield

5.4 Lines and Indentation

One of the first caveats programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

```
if True:
    print "True"
else:
    print "False"
```

However, the second block in this example will generate an error:

```
if True:
    print "Answer"
    print "True"
else:
    print "Answer"
    print "False"
```

Thus, in Python all the continuous lines indented with similar number of spaces would form a block. Following is the example having various statement blocks:

Note: Don't try to understand logic or different functions used. Just make sure you understood various blocks even if they are without braces.

```
#!/usr/bin/python

import sys

try:
    # open file stream
    file = open(file_name, "w")
except IOError:
    print "There was an error writing to", file_name
    sys.exit()

print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
    file_text = raw_input("Enter text: ")
    if file_text == file_finish:
        # close the file
        file.close
        break
    file.write(file_text)
file.write("\n")
file.close()

file_name = raw_input("Enter filename: ")

if len(file_name) == 0:
    print "Next time please enter something"
    sys.exit()
try:
    file = open(file_name, "r")
except IOError:
    print "There was an error reading file"
```



```
sys.exit()
file_text = file.read()
file.close()
print file_text
```

5.5 Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {} or () brackets do not need to use the line continuation character. For example:

```
days = ['Monday', 'Tuesday', 'Wednesday',
         'Thursday', 'Friday']
```

5.6 Quotation in python

Python accepts single ('), double (") and triple (""" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes can be used to span the string across multiple lines. For example, all the following are legal:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

5.7 Comments in python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python
# First comment
print "Hello, Python!"; # second comment
```

This will produce the following result:

```
Hello, Python!
```

A comment may be on the same line after a statement or expression:

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows:

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

5.8 Using blank Lines

A line containing only white-space, possibly with a comment, is known as a blank line and Python totally ignores it. In an interactive interpreter session, you must enter an empty physical line to terminate a multi-line statement.

5.9 Waiting for the users

The following line of the program displays the prompt, Press the enter key to exit and waits for the user to press the Enter key:

```
#!/usr/bin/python
raw_input("\n\nPress the enter key to exit.")
```

Here, "\n\n" are being used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

5.10 Multiple statements on a single line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

5.11 Multiple statement groups as suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as `if`, `while`, `def`, and `class`, are those which require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (`:`) and are followed by one or more lines, which make up the suite. For example:

```
if expression :  
    suite  
elif expression :  
    suite  
else :  
    suite
```

5.12 Command line arguments

You may have seen, for instance, that many programs can be run so that they provide you with some basic information about how they should be run, Python lets you do this with `-h`:

```
$ python -h  
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...  
Options and arguments (and corresponding environment  
variables):  
-c cmd : program passed in as string (terminates option list)  
-d : debug output from parser (also PYTHONDEBUG=x)  
-E : ignore environment variables (such as PYTHONPATH)  
-h : print this help message and exit  
[ etc. ]
```

You can also program your script in such a way that it should accept various options.

5.12.1 Accessing command line arguments

Python provides a **getopt** module that helps you parse command-line options and arguments.

```
$ python test.py arg1 arg2 arg3
```

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purpose:

- `sys.argv` is the list of command-line arguments.

- `len(sys.argv)` is the number of command-line arguments.

Here `sys.argv[0]` is the program ie. script name.

Example:

Consider a script `myprog.py`

```
#!/usr/bin/python
import sys
print('Number of arguments ', len(sys.argv), 'arguments.')
print('Argument List ', str(sys.argv))
```

When the above script is run as follows

```
$ python myprog.py arg1 arg2 arg3
```

Output you would get is

```
Number of arguments  4 arguments.
Argument List  ['myprog.py', 'arg1', 'arg2', 'arg3']
```

Note that the script's name is the first argument, and its also considered when counting the number of arguments

5.12.2 Parsing command line arguments

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command-line argument parsing. This tutorial would discuss about one method and one exception, which are sufficient for your programming requirements.

getopt.getopt() method:

This method parses command-line options and parameter list. Following is simple syntax for this method:

```
getopt.getopt(args, options[, long_options])
```

Here is the detail of the parameters:

- **args**: This is the argument list to be parsed.
- **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options**: This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.

This method returns value consisting of two elements: the first is a list of (**option, value**) pairs. The second is the list of program arguments left after the option list was stripped.

Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

Exception getopt.GetoptError :

This is exception gets raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option.

Example:

```
def main():
    host = None
    port = None
    try:
        opts, args = getopt.getopt(sys.argv[1:], "h:p:")
    except getopt.GetoptError as err:
        print str(err)
        usage()
        sys.exit(2)
    for o, a in opts:
        if o == "-h":
            host = a
        elif o == "-p":
```

```
        port = a
    else: assert False, "unhandled option"
x = PTECMPResolution(host, port)
x.send()
```

6. Python variable types, values and Identifiers

Variables in python are reserved spaces in memory that can hold some value, so whenever a variable is created some space is reserved in memory to hold the value of that variable. Every variable in python is a specific data type which tells the python interpreter how much memory should be reserved for the variable and what type of values the variable can hold.

In python variables don't need to be declared to assign values to them, so values can be assigned to variable names without explicit declaration of that variable beforehand, the declaration happens when you assign values to them.

6.1 Assigning values to variables

Assignments in python creates references not copies. Variables hold references to objects and do not hold the object itself. Names don't have intrinsic types objects have types, the type of reference is determined by the type of object assigned to the name. Remember the words "Name", "Variable" and "Identifier" refer to the same thing.

A name is created the first time it appears on the left hand side of an assignment. The code below shows some examples.

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Feb 20 15:42:18 2019
4
5 @author: khalid
6 """
7
8 x = 3 + 12 / 2
9 y = 'Im being assigned to variable y'
10
11 a = 1989
12 b = 3.14
13 c = 'Guido van Rossum'
14
15 print(x)
16 print(y)
17
18 print(a)
19 print(b)
20 print(c)
```

This python code when executed gives the following result.

```
9.0
Im being assigned to variable y
1989
3.14
Guido van Rossum

In [2]:
```

The garbage collector deletes reference after any names that are bound to the reference have passed out of scope. When the reference is deleted the object being referenced is deleted.

In the case where a non existent name(a variable that doesn't exist yet) is accessed an error is raised. Take a look at the example.

```
In [1]: y
Traceback (most recent call last):

  File "<ipython-input-1-009520053b00>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

If a name is accessed after assignment it returns the value the name was last assigned. Take a look at the example given below.

```
In [1]: y = 1
In [2]: y = 1989
In [3]: y
Out[3]: 1989
In [4]: |
```

6.2 Multiple assignment

Python allows assigning a single value to several names(variables) simultaneously. For example:

```
In [1]: x = y = z = 1
In [2]: x
Out[2]: 1
In [3]: y
Out[3]: 1
In [4]: z
Out[4]: 1
```

Memory is reserved for the integer object with the value 1, and all three variables refer to the same object. Python also allows assigning multiple objects to multiple variables like the example below.


```
In [1]: a, b, c, d = 1, 112, 5.0542, 'Multiple assignments!'

In [2]: a
Out[2]: 1

In [3]: b
Out[3]: 112

In [4]: c
Out[4]: 5.0542

In [5]: d
Out[5]: 'Multiple assignments!'
```

Here two integer objects are assigned to the names 'a' and 'b' , a float object is assigned to the name 'c' and a string object is assigned to the name 'd'.

6.3 Naming rules

Names are case sensitive they can contain letters, numbers and underscores but they cant start with a number.

Examples of valid name instances:

```
Var, VAR, Var1223, V123Bob, c020b
```

Examples of invalid instances:

```
1Var, 2a00
```

Reserved words cant be used as a variable name, so the following cant be used as names.

```
and, assert, break, class, continue, def, del, elif,
else, except, exec, finally, for, from, global, if,
import, in, is, lambda, not, or, pass, print, raise,
return, try, while
```

6.4 Understanding reference semantics

This may be different from the language that you used to work with but in python an assignment operation changes references instead of making a new copy of the object. This can be better understood with an example.

```
In [1]: x = 10
```

```
In [2]: y = 20
```

```
In [3]: x  
Out[3]: 10
```

```
In [4]: y  
Out[4]: 20
```

```
In [5]: x = y
```

```
In [6]: x  
Out[6]: 20
```

Here `x = y` doesn't make a copy of the object `y` references, instead it makes `x` reference the object(integer 20) that `y` refers to. This can prove to be very useful but we have to be careful for example take a look at the code below.

```
In [1]: x = [1, 2, 3] # x now references the list [1, 2, 3]
```

```
In [2]: y = x          # y now references what x references
```

```
In [3]: x.append(4)    # this changes the list x references
```

```
In [4]: print(y)      # if we print what y references you find that y also changed!  
[1, 2, 3, 4]
```

If this confusing it's Important to first understand what happens when an assignment operation(like `x = 3`) is performed.

There is a lot that happens in an assignment. For example this is what happens when the below assignment expression is executed

```
A = 12
```

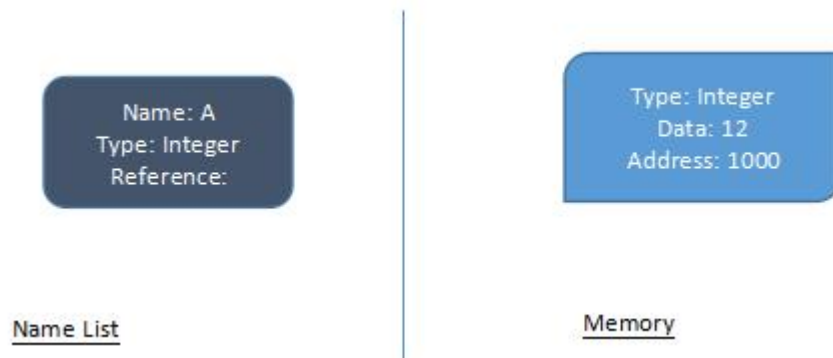
1. First an integer object 12 is created and it is stored in memory.

Name List

Memory

Type: Integer
Data: 12
Address: 1000

2. A name 'A' is created.



3. A reference to the memory location where the integer object 12 is stored at is now assigned to the name 'A'



The data 12 that was just created is of type integer. In python the data types integer, float, string and tuple are immutable(which means that object cant be changed). This does not mean that we cant change the value of the variable 'A', it can still be reassigned to another integer object i.e what 'A' refers to can be changed. For example we could still increment 'A' even though its data type is immutable.

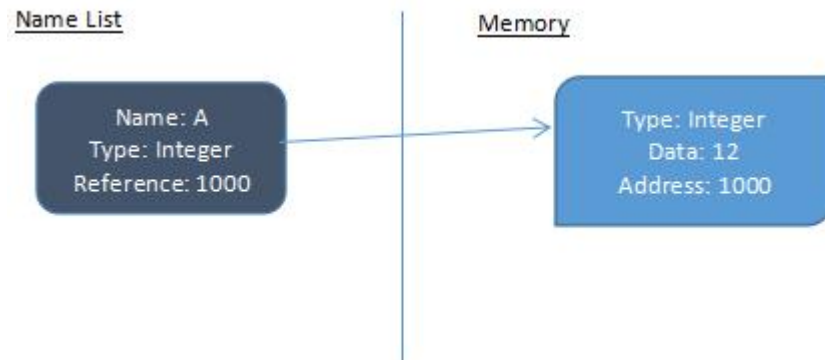
```
In [1]: A = 12
```

```
In [2]: A = A + 1
```

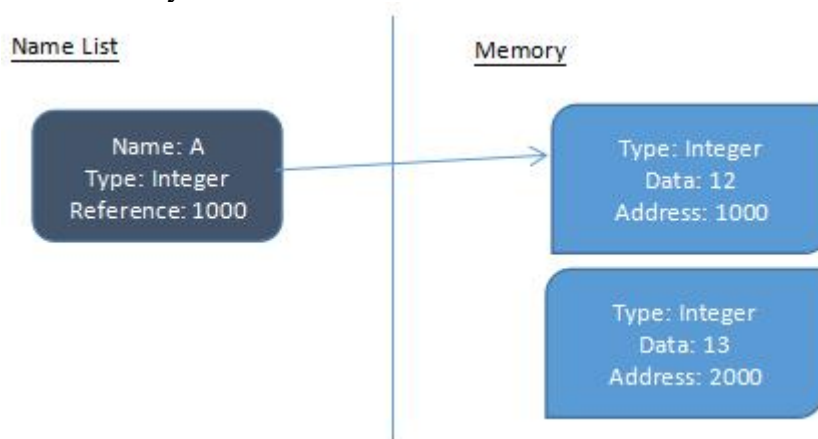
```
In [3]: A
Out[3]: 13
```

Lets look at what's really happening when 'A' is incremented($A = A + 1$).

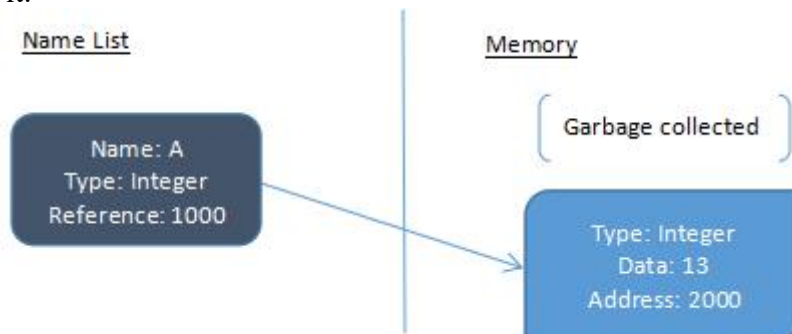
1. The reference of 'A' is looked up.
2. The value of the integer object at that reference is retrieved.



3. The $12 + 1$ increment operation is performed and the result integer object 13 is stored in a new memory location

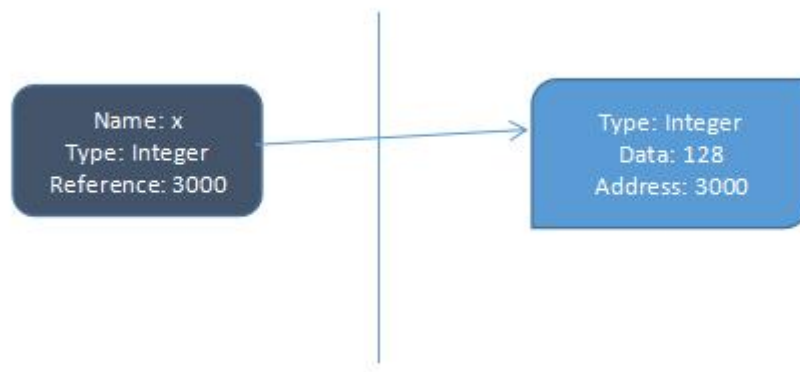


4. The variable 'A' is changed to point to this new integer object.while the previous integer object '12' is deleted by the garbage collector as there are no variables referring to it.

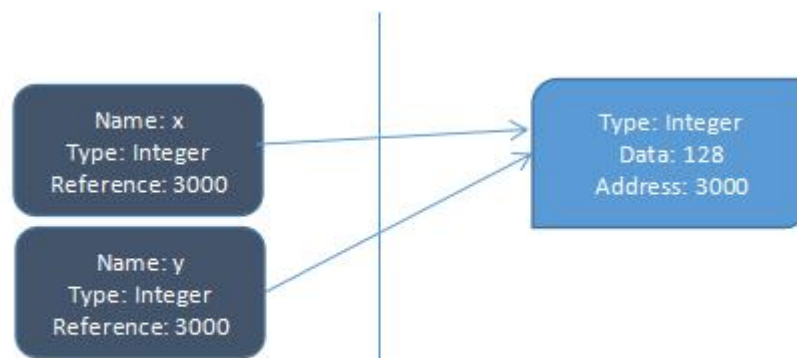


So for python's basic data types assignments behaves just as you expect them to. Take another example.

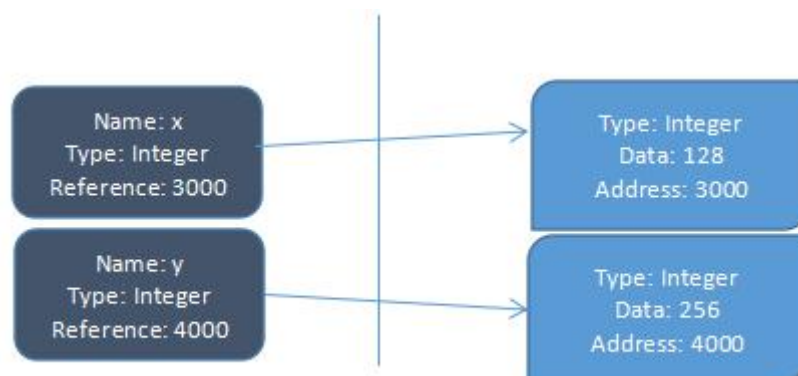
```
In [1]: x = 128
```



```
In [2]: y = x
```



```
In [3]: y = 256
```



```
In [4]: print(y)
256
```

```
In [5]: print(x)
128
```

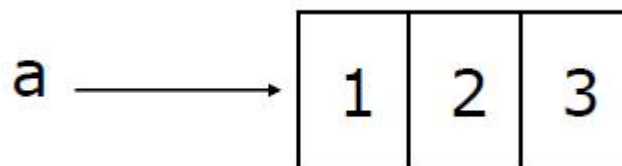
These data types that we've worked with till now are immutable(the values of the objects cant be changed). Assignments in python works differently when its done with mutable objects.

Data types like lists, dictionaries, user defined types are mutable, when mutable data is changed the change happens in place. That means they are not copied to a new memory location every time they are changed. If you type "`y = x`" then change the value of '`y`' the value of '`x`'. Remember a new copy is not created to make a change, the change is performed on the same object(unlike for immutable objects) so all variables referring to this object has its value changed as well. We'll go through an example on this just to make it clear.

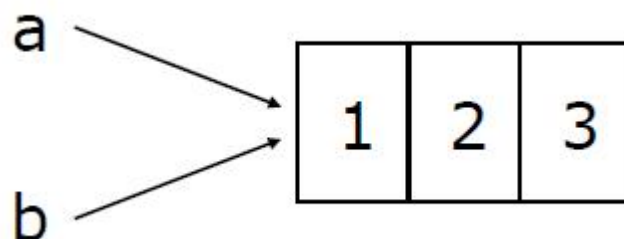
```
In [1]: a = [1,2,3]
In [2]: b = a
In [3]: b.append(4)
In [4]: a
Out[4]: [1, 2, 3, 4]
```

This is what happens when the above code is executed.

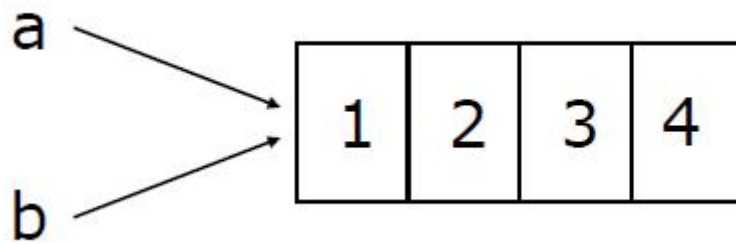
```
In [1]: a = [1,2,3]
```



```
In [2]: b = a
```



```
In [3]: b.append(4)
```



Lets look at each of python standard data types briefly one by one.

6.5 Python's standard data types

If you have experience in other programming languages you would already know that the value that can be stored in variables are of different types. Having different data types is of course a requirement of programmers in general. Each data type defines the different operations possible and it's storage mechanism.

Python has five standard data types:

- Integers
- Strings
- List
- Tuple
- Dictionary

6.5.1 Python Integer(Number)

Integer objects in python are immutable,so the value of the objects can't be changed so a new object is created every time a change is made to the variable

Number objects are created when a value is assigned to them.

```
In [1]: var1 = 10
```

```
In [2]: var2 = 20
```

```
In [3]: var1
```

```
Out[3]: 10
```

```
In [4]: var2
```

```
Out[4]: 20
```

These objects can be deleted using the 'del' keyword. The syntax for the 'del' command is as follows.

```
del var1[,var2[,var3[....,varN]]]
```

With 'del' objects can be deleted one at a time or many at a time. In the following code three variables(a, b, c) are created then 'a' and 'c' are deleted using 'del' command

```
In [1]: a = 10
In [2]: b = 20
In [3]: c = 30
In [4]: del a, c
In [5]: a
Traceback (most recent call last):
  File "<ipython-input-5-60b725f10c9c>", line 1, in <module>
    a
NameError: name 'a' is not defined
In [6]:
In [6]: b
Out[6]: 20
```

There are four different numerical types in Python:

1. `int` : These are standard signed integers. There is no upper limit on the number of bits that an `int` object can take up so they can take as much as the memory can offer. This can be tested out by running the following script

[illegible]

Output:

[illegible]

In python 3 there is only one int type but in python 2.7 there are two int types, one is 'int' which is of 32 bit length and the other is 'long int' which is the same as the Python 3.x int type(it can store arbitrarily large numbers).

When the following code is run on python 3.x and python 2.7 outputs in each would be as follows

[illegible]

Output in python 3.x :

```
<class 'int'>
<class 'int'>
```

Output in python 2.7 :

```
<type 'int'>
<type 'long'>
```

2. float (floating point real values): floats can be used to represent real numbers with a whole number part and a fractional part.
3. complex (complex numbers): represented by the formula $a + bJ$, where a and b are real numbers(float), and J is the square root of -1 (the result of which is an imaginary number). Complex numbers aren't that often used by python programmers.

Examples of each numeric type:

int	Long int	float	complex
1242	5193488661L	0.0	3.14j
-0009743	-0x19323L	-21.9	45.j
-0x3344	0162L	42.3+e18	4e+26J
1	-05236456727123L	80.2-E12	-.6985+0J
0x958	-47223598762L	-92.	4.76e-7j

Remember 'long int' is only in python 2.7 which is the same as 'int' in python 3.

6.5.2 Sequence types(Lists, Tuples and Strings)

Among the standard types in python there are three sequence types Lists, Tuples and Strings.

List:

A List is a simple ordered sequence of items, these items can be of different types and can also include other sequence types. Lists are mutable so to the value of list objects can be changed in place with out having to create a copy of the object that has the changed value.

List is the most versatile of the collection types in python. They contain a sequence of items separated by commas(,) and enclosed within square brackets([]) This might seem similar to arrays in c, but here items of the list can be of different types.

A list variable in python is created when a list object is assigned to a name as follows.

```
In [1]: myFirstList = ['xyz', 344, 40.14, 73]
In [2]: mySecondList = [23, 'Second List', (90, 30, 'ab')]
In [3]: myFirstList
Out[3]: ['xyz', 344, 40.14, 73]
In [4]: mySecondList
Out[4]: [23, 'Second List', (90, 30, 'ab')]
```

List are defined using square brackets and commas. Individual members of a list can be accessed using the square bracket notation, the same way arrays are accessed in c as follows.

```
In [6]: myFirstList[0]
Out[6]: 'xyz'
In [7]: myFirstList[1]
Out[7]: 344
In [8]: myFirstList[2]
Out[8]: 40.14
In [9]: myFirstList[3]
Out[9]: 73
```

Tuple:

A Tuple is a simple ordered sequence of items, these items can be of different types and can also include sequence types. They are immutable so to change the content of a tuple object a new object with the changed value is created and the tuple variable is made to point to this new object.

Python tuples are similar to lists except that unlike lists they are enclosed within parentheses and tuple objects are immutable. Tuples can be thought of as read only lists. Creating a tuple variable is similar to the creation of a list variable but with parentheses instead of square brackets.

```
In [1]: myFirstTuple = (245, 'xyz', 3.14, (2,3), '1989')
```

```
In [2]: myFirstTuple
```

```
Out[2]: (245, 'xyz', 3.14, (2, 3), '1989')
```

And just like lists, individual members of a tuple can be accessed using the square bracket notation, the same way arrays are accessed in c .

```
In [3]: myFirstTuple[0]
```

```
Out[3]: 245
```

```
In [4]: myFirstTuple[1]
```

```
Out[4]: 'xyz'
```

```
In [5]: myFirstTuple[2]
```

```
Out[5]: 3.14
```

```
In [6]: myFirstTuple[3]
```

```
Out[6]: (2, 3)
```

```
In [7]: myFirstTuple[4]
```

```
Out[7]: '1989'
```

String:

Strings in python are a continuous set of characters that are enclosed within quotation marks. Python allows using either single quotes(') or double quotes("").Strings are immutable like tuples, so the value of a string object cant be changed in place to change the value of a string object a new object is created with the changed value. Declaration of a string and accessing individual elements of a string are the same as in lists and tuples. Examples of string declaration and access are given below.

```
In [1]: myFirstString = 'Im a string'
```

```
In [2]: myFirstString
```

```
Out[2]: 'Im a string'
```

```
In [3]: myFirstString[0]
```

```
Out[3]: 'I'
```

```
In [4]: myFirstString[1]
```

```
Out[4]: 'm'
```

```
In [5]: myFirstString[2]
```

```
Out[5]: ' '
```

```
In [6]: myFirstString[10]
```

```
Out[6]: 'g'
```

6.5.3 Operations on sequence types

The operations shown here can be applied to all sequence types. Examples are shown for clarity on how they work. Most examples show the operation performed only on one or two of the sequence types but remember that these operations can be performed on all of the sequence types.

Positive and Negative index look up:

Positive index look up counts from the left and starts the count from 0.

```
In [1]: x = [1, 45.6, "qwerty"]

In [2]: x[1] # Positive index look up the second element
Out[2]: 45.6

In [3]: y = "Jupyter notebooks!"

In [4]: y[0] # positive index look up the first element
Out[4]: 'J'
```

Negative index look up counts from the left and starts the count from -1.

```
In [1]: x = (1, 45.6, "qwerty")

In [2]: y = ("Guido von Rossum", 1989)

In [3]: z = 'xyz'

In [4]: x[-1]
Out[4]: 'qwerty'

In [5]: y[-2]
Out[5]: 'Guido von Rossum'

In [6]: z[-3]
Out[6]: 'x'
```

When an element that is out of range is accessed the python interpreter gives an index error, an example is shown below.

```
In [1]: li = [0,1,2,3,4]

In [2]: li
Out[2]: [0, 1, 2, 3, 4]

In [3]: li[5]
Traceback (most recent call last):

  File "<ipython-input-3-5376a9d4d9a4>", line 1, in <module>
    li[5]

IndexError: list index out of range
```

Slicing:

Slicing can be used to return a copy subset of the sequence typed object. The general form of a slicing operation is as follows

```
<Variable Name>[x : y]
```

Where 'x' and 'y' are the first and second indices of the slicing operation. The container is copied from index 'x' and the copying is discontinued from index y. Lets go through some examples so its made clear how the slicing operation actually works. We'll work with the list 'li' given below.

```
var = (245, 'xyz', 3.14, (2,3), '1989')
```

The below slicing operation on 'var' returns a copy of a subset of 'var'. Copying starts from the first index(1) and discontinues from the second index(4).

```
In [2]: var[1 : 4]
Out[2]: ('xyz', 3.14, (2, 3))
```

Negative indices can also be used when slicing like below.

```
In [3]: var[1 : -2]
Out[3]: ('xyz', 3.14)
```

The first index can be left blank to return a copy of a subset of the that starts copying from the first element of the container.

```
In [5]: var[ : -2]
Out[5]: (245, 'xyz', 3.14)
```

The second index can be left blank to start copying from the first index and continue copying till the last element.

```
In [6]: var[0 : ]
Out[6]: (245, 'xyz', 3.14, (2, 3), '1989')

In [7]: var[4 : ]
Out[7]: ('1989',)
```

To return a copy of the entire container both indices can be left blank

```
In [8]: var[ : ]
Out[8]: (245, 'xyz', 3.14, (2, 3), '1989')
```

From the above examples it could be understood that for any sequence typed variable say 'A'

$$A[:] = A[0 :] \neq A[0 : -1]$$

Since lists are mutable sequence types, a change in the objects value happens in place without creating a new copy of the object, but when the slicing operator is used a new object is created and returned. Take a look at the example given below.

```
1 list1 = [1,2,3,4]
2
3 list2 = list1      # 2 names referring to the same object
4                    # When one name is changed the other is changed
5
6 list2 = list1[ : ]  # name list2 refers to a new object which is a
7                    # copy of a subset list1
8
```

The “in” operator:

The “in” operator can be used as a Boolean test to check whether some value is inside a container object.

```
In [1]: list1 = [1, 2, 'a', 'b', ([3,4])]
In [2]: 1 in list1
Out[2]: True

In [3]: "b" in list1
Out[3]: True

In [4]: ([3,4]) in list1
Out[4]: True

In [5]: 'b' not in list1
Out[5]: False

In [6]: 't' not in list1
Out[6]: True
```

For strings the “in” operator can be used to test whether the given sub string exists in the string like in the examples given below.

```
In [1]: myString = "Python is general purpose!"
In [2]: '!' in myString
Out[2]: True

In [3]: 'Python' in myString
Out[3]: True

In [4]: 'Java' in myString
Out[4]: False

In [5]: "Java" not in myString
Out[5]: True
```

The '+' operator:

This operator is used to concatenate two tuples, strings or lists together the result obtained after use is the new object obtained after concatenation. Concatenation can not be performed with different container types. Here are some examples of the use of the '+' operator with tuples, lists and strings.

With tuples:

```
In [1]: t1 = (1, 2, 3)

In [2]: t1 + (10, 20)
Out[2]: (1, 2, 3, 10, 20)

In [3]: (3, 2, 1) + t1
Out[3]: (3, 2, 1, 1, 2, 3)

In [4]: t1 + t1
Out[4]: (1, 2, 3, 1, 2, 3)

In [5]: (10, 40) + (100, 200)
Out[5]: (10, 40, 100, 200)

In [6]: t1 + t1 + t1
Out[6]: (1, 2, 3, 1, 2, 3, 1, 2, 3)

In [8]: t1 + (10000,) + t1 + (10, 20)
Out[8]: (1, 2, 3, 10000, 1, 2, 3, 10, 20)
```

With lists:(A new object is returned as the result)

```
In [1]: l1 = [1, 2, 3]

In [2]: l1 + [4, 5] + [6,] + [1000, 2000]
Out[2]: [1, 2, 3, 4, 5, 6, 1000, 2000]

In [3]: l1 + l1
Out[3]: [1, 2, 3, 1, 2, 3]

In [4]: [1, 2] + [3, 4]
Out[4]: [1, 2, 3, 4]
```

With strings:

```
In [1]: str1 = "Python is" + ' ' + 'Number 3' + " " + 'on TIOBE index'

In [2]: str1
Out[2]: 'Python is Number 3 on TIOBE index'

In [3]: str1 + "!!!!"
Out[3]: 'Python is Number 3 on TIOBE index!!!!'

In [5]: print("Hello" + " " + 'World!!')
Hello World!!
```


The '*' operator:

Produces a new tuple, list or string that has the original value repeated a number of times, the number of repeats is given as an operand to this operator and the other operand is the sequence itself. Lets take a look at some examples.

```
In [1]: t1 = (1, 2, 3)

In [2]: t1 * 2
Out[2]: (1, 2, 3, 1, 2, 3)

In [3]: (1,) * 5
Out[3]: (1, 1, 1, 1, 1)

In [4]: t1 = (1, 2) * 3

In [5]: t1
Out[5]: (1, 2, 1, 2, 1, 2)

In [6]: l1 = ['a', 'b', 'c']

In [7]: l1
Out[7]: ['a', 'b', 'c']

In [8]: l1 * 2
Out[8]: ['a', 'b', 'c', 'a', 'b', 'c']

In [9]: [2, 4] * 0
Out[9]: []

In [10]: "Python!" * 4
Out[10]: 'Python!Python!Python!Python!'

In [11]: str1 = "Anaconda! " * 2

In [12]: str1
Out[12]: 'Anaconda! Anaconda! '

In [13]: str1 * 0
Out[13]: ''
```

6.5.4 Lists and Tuples (Mutability vs immutability)

Tuple is an immutable type so the value/content of a tuple object can't be changed the only way to change the value of the a tuple variable is to create a new tuple object with the changed value and assign the new reference to the tuple variable. The value/content of a list can be changed in place as they are mutable, but when a value of a mutable object that is referred to by more than one variable is changed the value is changed for all variables referring to that object. Take a look at the example below.


```
In [1]: t1 = (1, 'xyz', 3.14, [5, 6, 7])

In [2]: t1[2] = 5.6
Traceback (most recent call last):

  File "<ipython-input-2-a8c083c13568>", line 1, in <module>
    t1[2] = 5.6

TypeError: 'tuple' object does not support item assignment
```

Tuples can't be changed but instead make a fresh tuple and assign the reference of it to an already existing name like below.

```
In [3]: t1 = (1, 'xyz', 5.6, [5, 6, 7])

In [4]: t1
Out[4]: (1, 'xyz', 5.6, [5, 6, 7])
```

Unlike tuples, lists are mutable and so they can be changed in place without having to change the reference to a new object

```
In [1]: li = ['xyz', 45, 5.6, 12]

In [2]: li[0] = 10

In [3]: li
Out[3]: [10, 45, 5.6, 12]
```

Like in the above example lists can be changed in place. Name “li” still points to the same memory reference when the changed is done. The mutability of lists means that they aren't as fast as tuples.

Since lists are mutable there are some operations that are applicable only to lists and not tuples. lets go through those operations one by one. The list class has some in built methods that can be used to change the value of the list object or return a value based on the current state of the list object. This is also our first exposure to python's method syntax.

append() :

The list class has an append method that can be used to insert a new list element at the end of the list. In the example below a new name “li” is assigned a reference to a list object, then the append method of the list class is called to add the element passed as an argument to method at the end of the list object.

```
In [1]: li = [1, 2, 3, 4]
```

```
In [2]: li  
Out[2]: [1, 2, 3, 4]
```

```
In [3]: li.append(5)
```

```
In [4]: li  
Out[4]: [1, 2, 3, 4, 5]
```

insert() :

The list class has an insert method that can be used to insert an new element at any location the list. The insert method takes two parameter's, first one is the location in the list to which the the new element is to be inserted and the second is the element itself. In the below example a new element is inserted at location 2 in the list.

```
In [1]: li = [1, 2, 3, 4]
```

```
In [2]: li  
Out[2]: [1, 2, 3, 4]
```

```
In [3]: li.insert(2, '128')
```

```
In [4]: li  
Out[4]: [1, 2, '128', 3, 4]
```

extend() :

Some times instead of appending single elements at a time we may want to append a list of elements to a list, the list class has a method to do that, the extend method. The extend method takes a list as an argument and appends that list to the list object to which it was called upon. An example is given below.

```
In [1]: li = [1, 2, 3, 4]
```

```
In [2]: li  
Out[2]: [1, 2, 3, 4]
```

```
In [3]: li.extend([5, 6, 7, 8])
```

```
In [4]: li  
Out[4]: [1, 2, 3, 4, 5, 6, 7, 8]
```

It may look like the extend method does exactly what the '+' operator does for sequence types but the difference is that the '+' operator creates a new list with a new memory reference, and extend operates on the list object it's called upon in place.

Extend method takes a list as an argument and append method takes a single object as an argument. Take a look at the example below for more clarity.

```

In [1]: list1 = [1, 2, 3]
In [2]: list2 = [1, 2, 3]
In [3]: list1.extend([1, 2, 3])
In [4]: list1
Out[4]: [1, 2, 3, 1, 2, 3]
In [5]: list2.append([1, 2, 3])
In [6]: list2
Out[6]: [1, 2, 3, [1, 2, 3]]

```

index() :

List class has an index method that returns the index of the first occurrence of the value passed as an argument to the method. If the value passed does not exist in the list then a “value error” is raised by the python interpreter. Examples are shown below.

```

In [1]: li = [1, 2, 3, 4]
In [2]: li.index(1)
Out[2]: 0
In [3]: li.index(4)
Out[3]: 3
In [4]: li.index(12)
Traceback (most recent call last):
  File "<ipython-input-4-8b5712dca681>", line 1, in <module>
    li.index(12)
ValueError: 12 is not in list

```

count() :

When a value is passed as an argument to the count method it returns the number of occurrences of that value in the list object on which it was called upon.

```

In [1]: li = [1, 2, 2, 3, 3, 3]
In [2]: li.count(1)
Out[2]: 1
In [3]: li.count(2)
Out[3]: 2
In [4]: li.count(3)
Out[4]: 3
In [5]: li.count(4)
Out[5]: 0

```

remove() :

The remove method of the list class can be used to remove the first occurrence of the value passed as the argument to the method. examples are shown below.

```
In [1]: li = ['z', 'y', 'x']
In [2]: li.extend(['z', 'y', 'x'])
In [3]: li.extend(['z', 'y', 'x'])
In [4]: li
Out[4]: ['z', 'y', 'x', 'z', 'y', 'x', 'z', 'y', 'x']
In [5]: li.remove('x')
In [6]: li
Out[6]: ['z', 'y', 'z', 'y', 'x', 'z', 'y', 'x']
In [7]: li.remove('x')
In [8]: li
Out[8]: ['z', 'y', 'z', 'y', 'z', 'y', 'x']
In [9]: li.remove('x')
In [10]: li
Out[10]: ['z', 'y', 'z', 'y', 'z', 'y']
```

reverse() :

When the reverse method of the list class is called on a list object it reverses the list object in place in the memory. In the below example when the reverse method is called for a list object, the order of its content are reversed.

```
In [1]: li = [1, 2, 3, 4, 5]
In [2]: li
Out[2]: [1, 2, 3, 4, 5]
In [3]: li.reverse()
In [4]: li
Out[4]: [5, 4, 3, 2, 1]
```

sort() :

The sort method of the list class can be used to sort the contents of the list in place in memory

```

In [1]: list1 = [5, 4, 3, 2, 1]

In [2]: list1.sort() # Sorts the list in ascending order

In [3]: list1
Out[3]: [1, 2, 3, 4, 5]

In [4]: list2 = ['E', 'D', 'C', 'B', 'A']

In [5]: list2.sort()

In [6]: list2
Out[6]: ['A', 'B', 'C', 'D', 'E']

```

The sort function of the list class can be passed a user defined function with it's own approach to sorting a list.

```
li.sort(userDefinedFunction()) # sort in place using user-defined comparison
```

Although lists are slower than tuples, they are much more powerful than tuples and are the most versatile types in python. Lists unlike tuples are mutable, they can be modified and they have a lot of operations that can be performed on them that make them very handy while programming. Tuples are immutable, they can't be modified and they have much lesser features than lists, although they are faster than lists.

list() and tuple() :

Python provides the list method and tuple method to convert between lists and tuples. The list method takes a tuple and returns a new instance of a list object. The tuple method takes a list and returns a new instance of a tuple object. Some examples are shown below.

```

In [1]: var1 = (523, 'abc', 3.141592, (2,3), 'XY!@3')

In [2]: type(var1)
Out[2]: tuple

In [3]: var2 = list(var1)

In [4]: var2
Out[4]: [523, 'abc', 3.141592, (2, 3), 'XY!@3']

In [5]: type(var2)
Out[5]: list

In [6]: var3 = tuple(var2)

In [7]: var3
Out[7]: (523, 'abc', 3.141592, (2, 3), 'XY!@3')

In [8]: type(var3)
Out[8]: tuple

```

6.5.5 Dictionary

Python dictionaries are like a hash table, They store a mapping between key's and value's. They consist of key value pairs, Where the key can be any of python's immutable types, although they are usually numbers or strings. Values on the other hand can be any python object. Dictionaries are enclosed within curly braces({ }) and values can be assigned and accessed using square braces([]). A single dictionary can store values of different types.

Dictionaries have many operations that can be performed on them. It's possible to define, view, look up and delete the key value pairs in a dictionary object.

Declaration :

Dictionaries are enclosed in curly braces({ }), each key value pair is separated by commas(,) and a colon(:) between the key and value of each key and value pair. The keys can be any immutable type in python and values can be any standard or user defined type in python. Dictionary variables are declared when a dictionary object reference is assigned to a name as in the example below.

```
In [1]: dict1 = {'User_Name': 'Larry Page', 'password': 1973}
```

Accessing Values :

Dictionary values can be accessed using the keys enclosed in square braces like how they are accessed in tuples and lists except in the case of dictionaries the subscript is the key. Some examples are shown below.

```
In [2]: dict1["User_Name"]  
Out[2]: 'Larry Page'
```

```
In [3]: dict1["password"]  
Out[3]: 1973
```

Python doesn't support look ups with the values if a key that doesn't exist in the dictionary is looked up a "key error" is raised.

```
In [4]: dict1[1973]  
Traceback (most recent call last):  
  
  File "<ipython-input-4-29298a58646a>", line 1, in <module>  
    dict1[1973]  
KeyError: 1973
```

Changing values :

Dictionaries are mutable types so the values of the dictionary objects are changed in place in memory. A value corresponding to a dictionary key can be changed as follows

```
In [5]: dict1["User_Name"] = "Sergey Brin"
In [7]: dict1["User_Name"]
Out[7]: 'Sergey Brin'
In [8]: dict1
Out[8]: {'User_Name': 'Sergey Brin', 'password': 1973}
```

Inserting a new key-value pair :

A new key-value pair can be inserted into the dictionary by assigning a value to a dictionary key that doesn't exist yet. An example is shown below.

```
In [9]: dict1["id"] = 100001
In [10]: dict1
Out[10]: {'User_Name': 'Sergey Brin', 'id': 100001, 'password': 1973}
```

del command :

The `del` command is used to remove a single key value pair from the dictionary object, only the specified key value pair is deleted the rest of the dictionary is left intact. An example of using the `del` command is shown below.

```
In [11]: del dict1["id"] # Removes only one
In [12]: dict1
Out[12]: {'User_Name': 'Sergey Brin', 'password': 1973}
```

clear() :

The dictionary class has a “clear” method which when called for a dictionary object deletes all the key-value pairs in that dictionary object. It is the same as applying the `del` command to all keys in the dictionary

```
In [13]: dict1
Out[13]: {'User_Name': 'Sergey Brin', 'password': 1973}
In [14]: dict1.clear()
In [15]: dict1
Out[15]: {}
```

Let's discuss some of the other methods that the python dictionary class offers.

keys() :

The “keys” method of python's dictionary class when called for a method returns a list containing all the keys of that dictionary object. An example is shown below.


```
In [16]: dict1 = {'User_Name': 'Larry Page', 'password': 1973}

In [17]: dict1
Out[17]: {'User_Name': 'Larry Page', 'password': 1973}

In [18]: dict1.keys()
Out[18]: dict_keys(['User_Name', 'password'])
```

values() :

The “values” method of the dictionary class when called for a dictionary object returns a list containing all the values in the dictionary.

```
In [19]: dict1.values()
Out[19]: dict_values(['Larry Page', 1973])
```

items() :

Python’s dictionary method has a items class which when called on an object returns a list of all the key value pairs as tuples. An example is shown below.

```
In [20]: dict1.items()
Out[20]: dict_items([('User_Name', 'Larry Page'), ('password', 1973)])
```

6.6 Data type conversion

There may come situations where it’s need to convert between python’s built in data type, to convert between built in data types the type nae can be used as a function. There are several built in functions to perform a conversion from one type to another type, these functions return a fresh object in memory with the converted value.

Function	Description
int(x [,base])	Converts x to an integer. base specifies the base if x is a string.
long(x [,base])	Converts x to a long integer. base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.

frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

In the next chapter we'll look into the basic python operator's.

7. Statements, Expressions and operators

In this chapter we'll get understand the what a statement, expression and an operator in python is. We'll discuss the different basic operators in python and the operator precedence in python. Let's get started.

7.1 Statements and Expressions

Statements are instructions that the python interpreter could execute. So far only the assignment, del and method invocation statement has been discussed but there are other statements like “for” statements, “if” statements, “while” statements, “import” statements and others.

Expressions, they are a combination of variables, values, function calls and operators. Expressions in python need to be evaluated before they can be used. When an expression is the right hand side of an assignment statement the expression is first evaluated before it is assigned to name. When an expression is passed as the parameter to python's “print” function the expressions is first evaluated then the result is displayed. In the below examples the expressions are first evaluated then are used.

```
In [1]: print(78 + 22 ** 1)
100
```

```
In [2]: print(len("Eclipse!"))
8
```

Here len() is a built in python function that returns the length of the string passed as an argument. In the above code “78 + 2 ** 1” and “len(“Eclipse!”)” are both expressions they are evaluated before they are used by the print function.

Evaluating an expression results in a value, and that's the reason why expressions can be on the right hand side of an assignment statement. A value or variable or a function call that return a value are all simple expressions. Evaluating a variable gives the value the variable refers to. It's important to remember that expressions in python returns a value. Take a look at the code below.

```
1 y = 3.1415
2 x = len("Spyder!")
3 print(x)
4 print(y)
```

We can see one of the differences between expressions and assignments when the above code is run in the python shell.

```
In [1]: y = 3.1415
```

```
In [2]: x = len("Spyder!")
```

```
In [3]: print(x)
7
```

```
In [4]: print(y)
3.1415
```

```
In [5]: x
Out[5]: 7

In [6]: y
Out[6]: 3.1415

In [7]: x + y
Out[7]: 10.1415
```

We can see that for assignments only the prompt is returned and no value. That's because assignment statements do not return any values. The assignment statement “`y = 3.1415`” doesn't return any value so only the prompt is returned, but the result of the simple expression “`3.1415`” returns a value (i.e. `3.1415`) which creates a new float object in memory and assigns its reference to ‘`y`’. Assignment statements are simply executed and don't return any value.

When the print statement is called on ‘`y`’ we can see the value that ‘`y`’ is referring to as the print function returns a value, also when ‘`y`’ is entered into the shell by itself ‘`y`’ is evaluated and returned.

7.2 What are operators and operands

Operands are the tokens in python that represent some kind of computation and the computation happens between the operand or operands that are given to the operator to work on.

The following are some legal expressions using arithmetic operators.

```
1 # Assignments
2 x = 10
3 y = 20
4 z = 30
5
6 # Valid expressions using arithmetic operators
7
8 25 + 22
9 x - 1
10 y * 60 + z
11 x / 60
12 5 ** 2
13 (5 + 13) * (55 + 77)
```

The operators that python supports are listed below.

- Arithmetic operators
- Relational operators (comparison operators)
- Assignment operators
- Logical operators
- Bitwise operators
- Membership operators

- Identity operators

We'll go through each of these operators one by one.

7.3 Arithmetic operators

The seven arithmetic operators supported by python are listed and described below briefly. We'll then go through some examples on these arithmetic operators.

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponent - Performs exponential (power) calculation on operators
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.

Now lets look at some example code that demonstrates how the arithmetic operators work in python.

```
1 #=====
2 # Assignments
3 #=====
4 x = 100
5 y = 200
6 z = 500
7
8 #=====
9 # Using arithmetic operators
10 #=====
11 z = x + y
12 print("Line 1 - Value of z is ", z)
13 z = x - y
14 print("Line 2 - Value of z is ", z)
15 z = x * y
16 print("Line 3 - Value of z is ", z)
17 z = x / y
18 print("Line 4 - Value of z is ", z)
19 z = x % y
20 print("Line 5 - Value of z is ", z)
```

```

21 x = 2
22 y = 3
23 z = x**y
24 print("Line 6 - Value of z is ", z)
25 x = 10
26 y = 5
27 z = x//y
28 print("Line 7 - Value of z is ", z)
29

```

The above code gives the output given below.

```

Line 1 - Value of z is 300
Line 2 - Value of z is -100
Line 3 - Value of z is 20000
Line 4 - Value of z is 0.5
Line 5 - Value of z is 100
Line 6 - Value of z is 8
Line 7 - Value of z is 2

```

7.4 Relational operators

The six relational operators that python supports are listed and described in the table below.

Operators	Description
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

The example code below demonstrates the use of python's relational operators. As relational operators give a Boolean result of either true or false, if-else statements are used to demonstrate the behaviour of relational operators in python .

```

1 #=====
2 # Assignments
3 #=====
4 x = 10
5 y = 5
6 z = 0
7

```

```

8 #=====
9 # Using relational operators
10 #=====
11 if( x == y ):
12     print("Line 1 - x is equal to y")
13 else:
14     print("Line 1 - x is not equal to y")
15 if( x != y ):
16     print("Line 2 - x is not equal to y")
17 else:
18     print("Line 2 - x is equal to y")
19 if( x < y ):
20     print("Line 3 - x is less than y")
21 else:
22     print("Line 3 - x is not less than y")
23 if( x > y ):
24     print("Line 4 - x is greater than y")
25 else:
26     print("Line 4 - x is not greater than y")
27 x = 5;
28 y = 20;
29 if( x <= y ):
30     print("Line 5 - x is either less than or equal to y")
31 else:
32     print("Line 5 - x is neither less than nor equal to y")
33 if( y >= x ):
34     print("Line 6 - y is either greater than or equal to x")
35 else:
36     print("Line 6 - y is neither greater than nor equal to x")
37

```

The above python code is ran it gives the output shown below.

```

Line 1 - x is not equal to y
Line 2 - x is not equal to y
Line 3 - x is not less than y
Line 4 - x is greater than y
Line 5 - x is either less than or equal to y
Line 6 - y is either greater than or equal to x

```

7.5 Relational operators

Python's most basic assignment operator is '=' which assigns a reference to the value on the right hand of the operator to the variable name on the left hand side of the operator. Python also has several other assignment operators that perform an arithmetic operation before the assignment is made. The nine different assignment operators in python are listed and described below.

Operator	Description
=	Simple assignment operator, Assigns values from right side operands to left side operand

+=	Add AND assignment operator, It adds right operand to the left operand and assigns the result to left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assigns the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assigns the result to left operand
/=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assigns the result to left operand
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assigns value to the left operand
//=	Floor Division and assigns a value, Performs floor division on operators and assigns value to the left operand

Example code is given below that explains the working of the different assignment operators in python.

```

1 #=====
2 # Assignments
3 #=====
4 x = 10
5 y = 5
6 z = 0
7
8 #=====
9 # Using assignment operators
10 #=====
11 z = x + y
12 print("Line 1 - Value of z is ", z)
13 z += x
14 print("Line 2 - Value of z is ", z)
15 z *= x
16 print("Line 3 - Value of z is ", z)
17 z /= x
18 print("Line 4 - Value of z is ", z)
19 z = 2
20 z %= x
21 print("Line 5 - Value of z is ", z)
22 z **= x
23 print("Line 6 - Value of z is ", z)
24 z //= x
25 print("Line 7 - Value of z is ", z)

```

The above code gives the below output when executed.

```

Line 1 - Value of z is 15
Line 2 - Value of z is 25
Line 3 - Value of z is 250
Line 4 - Value of z is 25.0
Line 5 - Value of z is 2
Line 6 - Value of z is 1024
Line 7 - Value of z is 102

```

7.6 Bitwise operators

Python's bitwise operators are applied to the binary form of the values that python's numeric data types can hold.

Bitwise operators are performed bit by bit. For example when a bitwise operation is performed on “x = 10” and “y = 20”, the binary equivalent of the integer objects 10 and 20 are used instead of the decimal values(10, 20) that were assigned to ‘x’ and ‘y’, so the operands to the bitwise operation are the binary values “1010”(10) and “10100”(20). Operation is performed bit by bit.

X = 63 (base - 10) = 0011 1100 (base - 2)

Y = 13 (base - 10) = 0000 1101 (base - 2)

X & y = 0000 1100

X & y = 0011 1101

X & y = 0011 0001

The six bitwise operators that python supports are listed and briefly described in the below table.

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

The following example code makes use of each of bitwise operations that python supports.


```

1 #=====
2 # Assignments
3 #=====
4
5 #-----In Decimal-----In Binary-----
6 #-----
7 x =      60          # 0011 1100
8 y =      13          # 0000 1101
9 z =       0
10
11 #=====
12 # Using assignment operators
13 #=====
14 z = x & y;          # 12 = 0000 1100
15 print("Line 1 - Value of z is ", z)
16
17 z = x | y;          # 61 = 0011 1101
18 print("Line 2 - Value of z is ", z)
19
20 z = x ^ y;          # 49 = 0011 0001
21 print("Line 3 - Value of z is ", z)
22
23 z = ~x;             # -61 = 1100 0011
24 print("Line 4 - Value of z is ", z)
25
26 z = x << 2;          # 240 = 1111 0000
27 print("Line 5 - Value of z is ", z)
28
29 z = x >> 2;          # 15 = 0000 1111
30 print("Line 6 - Value of z is ", z)

```

When the above python script is executed it displays the following output.

```

Line 1 - Value of z is 12
Line 2 - Value of z is 61
Line 3 - Value of z is 49
Line 4 - Value of z is -61
Line 5 - Value of z is 240
Line 6 - Value of z is 15

```

7.7 Logical operators

The logical operators and, or and not are supported by python these logical operators are described below. Unlike languages like c, logical operators are written in plain in English.

Operators	Description
and	Called logical AND operator. If both the operands are true, then the condition becomes true.
or	Called logical OR Operator. If any of the two operands are non zero, then the condition becomes true.
not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

Any non zero numerical value is considered true and the numerical value zero(0) is considered false. The example coed below works with python's logical operators.

```
1 #=====
2 # Assignments
3 #=====
4 x = 5
5 y = 10
6
7 #=====
8 # Using Logical operators
9 #=====
10 if( x and y ):
11     print("Line 1 - x and y are true")
12 else:
13     print("Line 1 - Either x is not true or y is not true")
14
15 if( x or y ):
16     print("Line 2 - Either x is true or y is true or both are true")
17 else:
18     print("Line 2 - Neither x is true nor y is true")
19
20 x = 0
21
22 if( x and y ):
23     print("Line 3 - x and y are true")
24 else:
25     print("Line 3 - Either x is not true or y is not true")
26
27 if( x or y ):
28     print("Line 4 - Either x is true or y is true or both are true")
29 else:
30     print("Line 4 - Neither x is true nor y is true")
31
32 if not( x and y ):
33     print("Line 5 - Either x is not true or y is not true")
34 else:
35     print("Line 5 - x and y are true")
36
```

The output when the above code is executed is provided below.

```
Line 1 - x and y are true
Line 2 - Either x is true or y is true or both are true
Line 3 - Either x is not true or y is not true
Line 4 - Either x is true or y is true or both are true
Line 5 - Either x is not true or y is not true
```

7.8 Membership operators

Python's two membership operators enable testing whether a given element exists in a sequence (lists, tuples or strings). The two membership operators that python support are listed and described in the table below.

Operators	Description
<code>in</code>	Evaluates to true if it finds a variable in the specified sequence and false otherwise.
<code>not in</code>	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

The example code below may give a better understanding on what the python membership operators do.

```
1 #=====
2 # Assignments
3 #=====
4 x = 6
5 y = 7
6 li = [1, 2, 3, 4, 5 ];
7
8 #=====
9 # Using membership operators
10 #=====
11 if ( x in li ):
12     print("Line 1 - x is available in the given list")
13 else:
14     print("Line 1 - x is not available in the given list")
15
16 if ( y not in li ):
17     print("Line 2 - y is not available in the given list")
18 else:
19     print("Line 2 - y is available in the given list")
20
21 x = 4
22
23 if ( x in li ):
24     print("Line 3 - x is available in the given list")
25 else:
26     print("Line 3 - x is not available in the given list")
27
```

The above code gives the below output.

```
Line 1 - x is not available in the given list
Line 2 - y is not available in the given list
Line 3 - x is available in the given list
```

7.9 Identity operators

Python's identity operators can be used to check whether two names points to/references the same object in memory. There are two identity operators supported by python, they are described in the following table.

Operators	Description
<code>is</code>	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
<code>is not</code>	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

The code below demonstrates the use of identity operators in python. When you try this out in your python shell you can see how the python interpreter assigns objects to variables.

```
1 #=====
2 # Assignments
3 #=====
4 x = 10
5 y = 10
6
7 #=====
8 # Using Identity operators
9 #=====
10 if ( x is y ):
11     print("Line 1 - x and y have same identity")
12 else:
13     print("Line 1 - x and y do not have same identity")
14
15 if ( id(x) == id(x) ):
16     print("Line 2 - x and y have same identity")
17 else:
18     print("Line 2 - x and y do not have same identity")
19
20 y = 30
21
22 if ( x is y ):
23     print("Line 3 - x and y have same identity")
24 else:
25     print("Line 3 - x and y do not have same identity")
26
27 if ( x is not y ):
28     print("Line 4 - x and y do not have same identity")
29 else:
30     print("Line 4 - x and y have same identity")
31
```

The `id()` method takes a variable as parameter and returns the identity of the object that name references. An identity of an object is constant and is retained by the object as long as it is alive. The output of the of the above code is given below.

```
Line 1 - x and y have same identity
Line 2 - x and y have same identity
Line 3 - x and y do not have same identity
Line 4 - x and y do not have same identity
```

7.10 Precedence and associativity of python operators

It's already been discussed that an expression in python and most other programming languages are a combination of values, variables, function calls and operators. Lets take some instances of simple expressions in python.

```
1 # Assignments
2 var1 = 10
3 var2 = 20
4 var3 = "Ipython"
5
6 # Expressions with a single or no operator
7 var1
8 var2
9 var1 + var2
10 var1 % 3
11 len(var3) ** var1
12 1 + 4.67
```

Remember a valid expression always returns a value and an expression can not be used until its evaluated. In the above code every expression has either one or no operator so evaluation is very straight forward but in the cases where multiple operators are involved in the expression what is the order of evaluation to be followed? Valid expressions of this case are given in the code below.

```
1 # Assignments
2 var1 = 10
3 var2 = 20
4 var3 = "Ipython"
5
6 # Expressions with multiple operators
7 var1 + var2 ** (2 % 7) // var1
8 var1 % 3 ** (var1 + var2)
9 len(var3) ** var1 % 100000
10 1 + 4.67 * 5 + (len(var3) * id(var2))
11
```

Python specifies the order of precedence of operators and associativity(in the case that there are more than one operator with the same precedence level). For example example multiplication has a higher order than subtraction.

```
In [1]: 41 - 4 * 10
Out[1]: 1
```

But we can change this order of evaluation by using parentheses(()). parentheses have a higher precedence level.

```
In [2]: (41 - 4) * 10
Out[2]: 370
```

Python operator precedence rule (from highest precedence to lowest)

Operator	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, %, //	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, in, not in, is, is not,	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Python operator associativity

In the above table there are certain precedence levels that have multiple operators, this means that all those operators have the same level of precedence. When there are multiple operators of the same precedence in an expression the appropriate associativity rule is followed. For almost all of python's operators the associativity rule is left to right. Take a look at the example below.

[illegible]

The exponent operator (`**`) has right to left associativity in python, so if there are multiple exponent operators they are evaluated from right to left after operators of higher precedence have already been evaluated.

```
1 #=====
2 # Right to left associativity of ** exponent operator
3 #=====
4
5 # Output: 512
6 print(2 ** 3 ** 2)
7
8 # Output: 64
9 print((2 ** 3) ** 2) # parantheses has the highest precedence
10
```

Non associative operators(Assignment and comparison operators)

There are some python operators that do not follow any associativity rule, the assignment and comparison operators. There are some special rules followed when multiple operators of this type are used in a sequence.

Comparison operators:

Take a conditional expression “`x < y < z`” this expression is not equivalent to “`(x < y) < z`” or “`x < (y < z)`”, but instead is equivalent to “`x < y and y < z`” and this is evaluated from left to right.

Assignment operators:

Assignments of the form “`x = y = z`” are valid and are evaluated from right to left, but assignments of the form “`x = y += z`” are not considered to be a valid syntax.

```
1 # Initializing x, y, z
2
3 x = 1, y = 1, z = 1
4
5 # The below assignment
6 # expression is invalid
7 # (Non-associative operators)
8 # SyntaxError: invalid syntax
9
10 x = y += 2
```

So when the python interpreter encounters such syntax it raises a syntax error.

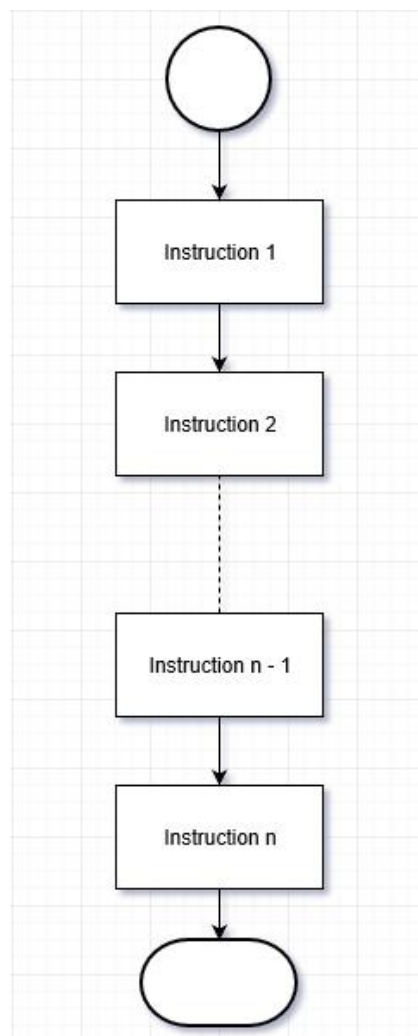
```
File "C:/Users/khalid/Documents/ML/Bucca/IdentityOperators.py", line 10
    x = y += 2
SyntaxError: invalid syntax
```

8. Python's flow control tools

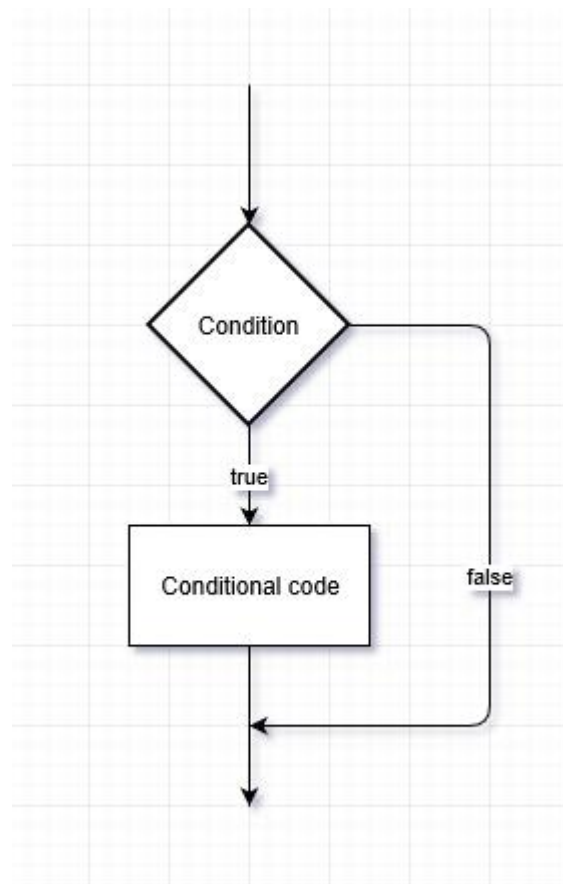
Most of the code examples that were discussed prior to this chapter followed a sequential execution of statements, from start to finish (from line 1 to that last line of code) except some examples that had to involve the “if” and “if-else” for the purpose of explanation of the content being discussed. We’ll now go through each of the tools that Python offers to change the normal sequential flow of control that Python programs follow.

These control flow statements can also be referred to as “branching statements”. There are two types of branching statements in Python: “conditional branching” and “unconditional branching”. Conditional branching statements change the control flow of programs based on the result of a conditional expression, while unconditional branching statements change the control flow of the program when they are encountered and regardless of any condition. Unconditional branching instructions are almost always used along with conditional branching statements.

The sequential control of a Python program is shown below in the flow diagram.



The following flowchart portrays the general decision making structure in almost all programming languages.



In python the numerical value “0” and “null” are considered to be “false” any value that is not “0” or “null” are considered to be “true”. The types of decision making statements in python are listed and briefly described below, we’ll go into details in the coming sections.

Statement	Description
if statements	An if statement consists of a conditional expression followed by one or more statements.
if-else statements	An if statement can be followed by an optional else statement, which executes when the conditional expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).

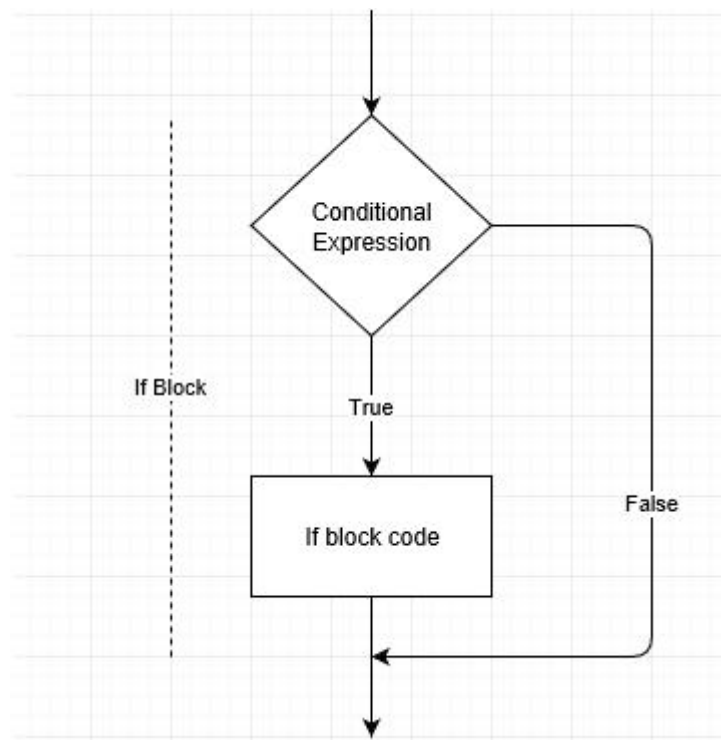
Decision making is needed in cases where a block of code should be executed only when certain condition is true, for these kind of cases that require decision making we can use “if...elif...else” statements.

8.1 if statements

The “if statement” has a conditional expression, when the result of the conditional expression is true the body of the “if statement” is executed, in the case that it is false body of the “if statement” is skipped. The general syntax of the if statement is given below.

```
if <Boolean Expression> :  
    <if block Statement(s)>
```

In python the body of the “if statement” is specified by indentation. The body starts with an indentation and the first line without the indentation marks the end and is outside the body of the “if block”. The flowchart of python’s “if statement” is shown below.



The below example code demonstrates how the “if statement” works in python.

```
1 #=====
2 # Demonstrating the if statement.
3 # If the number is positive, then print the appropriate message.
4 #=====
5 num = 3
6 if num > 0:                                # if statements condition
7     print(num, "is a positive number.")    # indentation marks the if block
8     print("still in if block")             # indentation marks the if block
9 print("outside the if block.")             # outside the if block
10
```

```

11 num = -1
12
13 if num > 0:                # if statements condition
14     print(num, "is a positive number.") # indentation marks the if block
15 print("outside the if block.") # outside the if block
16

```

The body of the “if statement” is executed when the condition “num > 0” is evaluated to be true. If the condition is evaluated to be “true” the body of the “if statement”(marked by the indentation) gets executed, else it is skipped. In the above example code the condition of the first “if statement” is true so it’s body is executed and the condition of the second “if statement” is false so it’s body is skipped. The output when the above program is executed is given below.

```

3 is a positive number.
still in if block
outside the if block.
outside the if block.

```

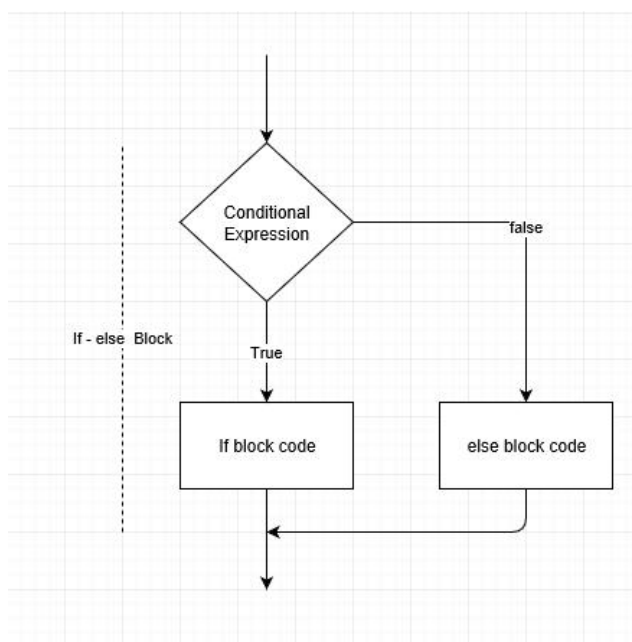
8.2 if-else statements

The “if-else statement” has a conditional expression, when the result of the conditional expression is true the body of the “if” is executed, in the case that it is false body of the “else” is executed. The general syntax of the if-else statement is given below.

```

if <Boolean Expression>:
    <if block Statement(s)>
else:
    <else block Statement(s)>

```



In the given flow chart of the flow chart of the “if-else statement” we can see that when the Boolean expression is evaluated to be false the the body of the else block is executed instead.

The example code below shows an instance of using the “if-else statement”, take a look for more clarity.

```
1 #=====
2 # Demonstrating the if-else statement.
3 # If the number is positive, then print the appropriate message.
4 #=====
5
6 num = 3
7
8 # try these two variations your self
9 # num = -12
10 # num = 0
11
12 if num >= 0:
13     print("num is either Positive or Zero")
14 else:
15     print("num is a Negative number")
16
```

The output of the above program is given below, and try the above two variations program yourself.

```
num is either Positive or Zero
```

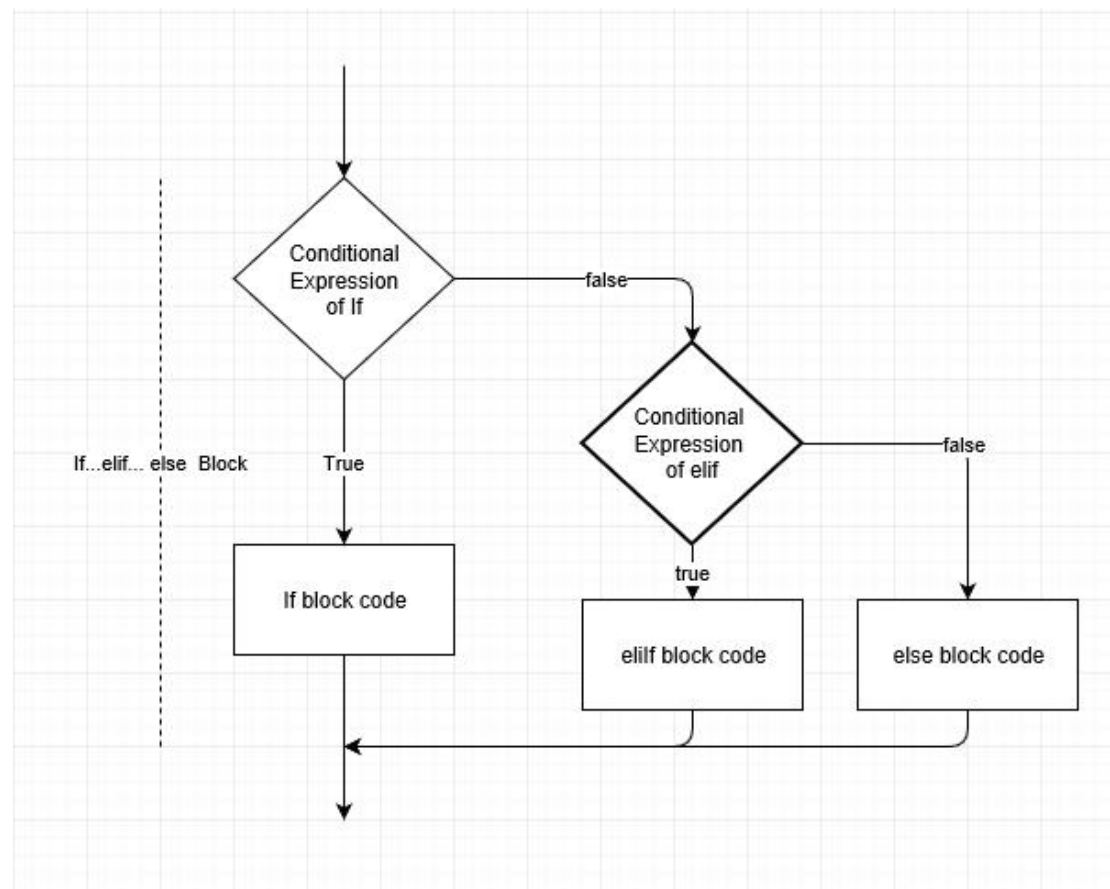
In the above program the condition of the “if statement”(num > 0) is true so the “if block” is executed and the “else” block code is skipped. If the value of num is changed to “0” the condition is still evaluated to be true so again the “if” block is executed instead. If the value of num is changed to “-12” the “else” block is executed instead.

8.3 if..elif..else statements

The general syntax of “if..elif..else” is given below. First the Boolean expression of “if” is evaluated, if it is evaluated to be false then the Boolean expression of elif is evaluated, if it is evaluated to be true then the body of that “elif” is executed if it is false then it’s body is skipped and control goes to the next “elif” or “else”. It is allowed to have more than one “elif” in the “if..elif..else” chain.

```
if <Boolean Expression>:
    <if block Statement(s)>
elif <Boolean Expression>:
    <elif block Statement(s)>
else:
    <else block Statement(s)>
```

The flowchart of the if..elif..else chain is shown below. Remember there can be more than one “elif” in the “if..elif..else” chain.



Lets take a look at a code example to better understand how the if..elif..else decision chain works in python.

```
1 #=====
2 # Demonstrating the if..elif..else Decision chain.
3 # Print the appropriate message for the different values of num
4 #=====
5
6 num = 3
7
8 # try these two variations your self
9 # num = 0
10 # num = -20
11
12 if num > 0:
13     print("Positive number")
14 elif num == 0:
15     print("Zero")
16 else:
17     print("Negative number")
18
```

In the above program if “num” is positive then “positive number” Since the Boolean expression of “if” evaluates to be true, the if block code is executed, after execution the control goes out of the current “if..elif..else” decision chain. Try the above two variations yourself. The output of the above program is given below.

Positive number

Python permits nesting any number “if..elif..else” chains within “if..elif..else” decision chains. The only way to differentiate one chain from the other is by the indentation and so in very complex cases gets confusing to read, This is better avoided to maintain the readability of the program. Lets take a look at a code example that uses “nested if statements”.

```
1 #=====
2 # Demonstrating the nested if statements.
3 # Print the appropriate message for the different values of num
4 #=====
5
6 num = int(input("Enter a number: ")) # input from user
7                                     # convert to type int
8
9 if num >= 0:
10     if num == 0:
11         print("You entered a Zero!")
12     else:
13         print("You entered a Positive number!")
14 else:
15     print("You entered a Negative number!")
16
```

The output of the above program for each possible case is given below. In the above python code the input function waits for the user to enter a value in the python shell and returns this value as a string. In this program it is expected that the user enters a numerical value, this value entered by the user is returned by the input function as a string which is converted into an int type using the int() method.

```
In [1]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/Bucca')
```

```
Enter a number: 12
You entered a Positive number!
```

```
In [2]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/Bucca')
```

```
Enter a number: -1
You entered a Negative number!
```

```
In [3]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/Bucca')
```

```
Enter a number: 0
You entered a Zero!
```

There are some statements in python that allow iterating the execution over a certain part of the code until some condition is met, these types of statements are called looping statements. Python provides two looping statements

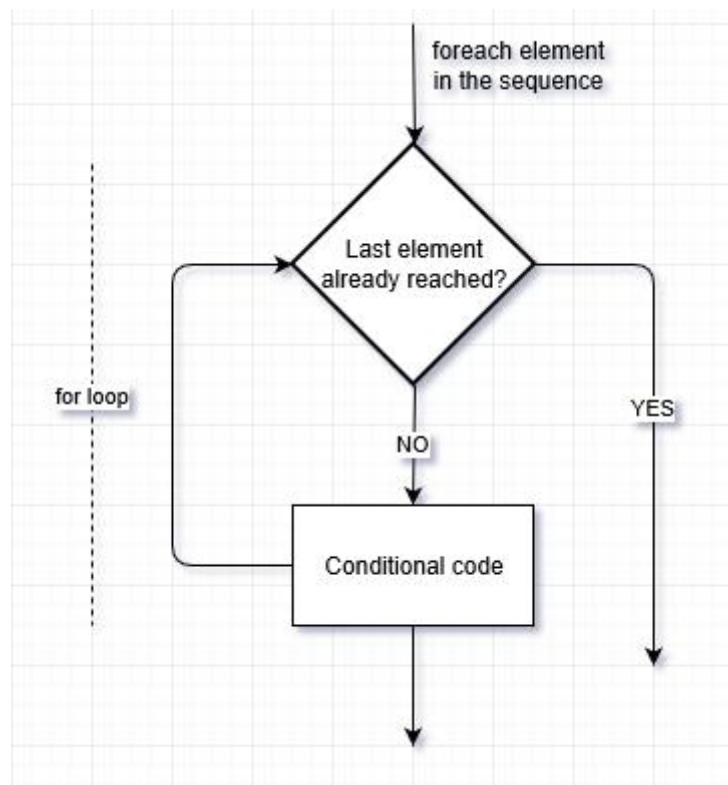
- for loop
- While loop

8.4 Python's for loop

For loops are very handy they are used to iterate over all the elements of a sequence typed object (lists, tuples, strings) or an object that is iterated through. They are similar to the “foreach statement” which is available in languages like C#. The body of the for loop is executed once for each element of the container object. The general syntax of python's for loop is given below.

```
for <variable> in <iterable typed object>:  
    <Statements in body of for loop>
```

Here the body of the for loop is identified by indentation. <variable> can be used in the body of the for loop and it holds the value of the element in the sequence typed object for each iteration. The loop continues until the last element in the sequence is reached. The flow chart of the for loop is shown below.



Lets look at a program that uses a “for loop” in it. In the example below a for loop is used to add the elements of a list and print the output. “numbers” is a a hard coded list of integers and variable “sum” is initialized to 0. In every iteration the current element of the sequence of that iteration is added to the variable “sum”.

```
1 #=====
2 # Program to find the sum of all numbers in a list
3 #=====
4
5 # List of integers
6 numlist = [10, 20, 30, 40, 50, 60, 70, 80, 90]
7
8 # variable to store the sum of elements
9 sum = 0
10
11 # iterate over the list
12 for num in numlist:
13     sum = sum + num
14
15 # Output: The sum is 450
16 print("The sum of all elements is ", sum)
17
```

The output of the above code is given below.

```
The sum of all elements is 450
```

8.4.1 The range() method

Python provides a range method that can be used to generate a list of a sequence of numbers. For example if “range(10)” is called it can generate a list of numbers from 0 to 10.

```
In [1]: list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The number to start from, the number to end at and the step size can be specified in the range function by passing them as parameters as “range(start, stop, step size)”. The step size has a default value of “1” if not specified. The start index defaults to “0” if not specified. The range function can be called in the following ways.

- range(stop)
- range(start, stop)
- range(start, stop, step size)

Here start is the number to start from, stop is the number to stop at(not generated by range function). In python 3.7 the range function returns an object of type “range”, which can be converted to a list using the list() method. The following examples can clarify more on the range() function.


```

In [1]: range(10)
Out[1]: range(0, 10)

In [2]: range(5,10)
Out[2]: range(5, 10)

In [3]: li = list(range(5,10))

In [4]: li
Out[4]: [5, 6, 7, 8, 9]

In [5]: list(range(-5, 5, 2))
Out[5]: [-5, -3, -1, 1, 3]

In [6]: list(range(0, 10, 2))
Out[6]: [0, 2, 4, 6, 8]

```

The `range()` function is being discussed with “for loops” because the `range()` function is mostly used with for loops. Then can be used to generate the sequence of items that the “for loop” iterates over. In the example below the `range()` function is combined with the `len()` is used in a for loop to iterate through a list of items using index.

```

1 #=====
2 # iterate through a list using indexing
3 #=====
4 Titans = ['Apple', 'Alphabet', 'Microsoft']
5
6 # iterate through the list using index
7 for i in range(len(Titans)):
8     print(Titans[i])          # Accessing the i-th element of List Titans
9

```

Here `len(Titans)` returns “3” which is passed as a parameter to the `range()` function. The `range()` method returns an range object “`range(0, 3)`” which can be iterated through by the for loop. This range can be used as the index for the “for loop”. The above code generates the below output.

```

Apple
Alphabet
Microsoft

```

8.4.2 for / else statement

The “for loop” can have an optional “else” block, which is executed after all the elements in the sequence have been iterated through. In the case that a “break” statement was encountered during the execution of the “for loop” which causes an abrupt exit from the loop the “else” part is ignored and so the else part of the “for loop” is executed if there is no abrupt exit from the loop(if a break statement is not encountered while in the loop). An example that involves the “else” part of the “for loop” is given below.

```

1 #=====
2 # for / else loops example
3 #=====
4 temp = list(range(10))
5
6 for i in temp:
7     print(i)
8 else:
9     print("Done!") # Executed after all iterations through
10                   # the list are completed.
11

```

The output of the above program is given below.

```

0
1
2
3
4
5
6
7
8
9
Done!

```

8.5 Python's while loop

Python provides flow control tool, “while” statement which is similar to the “for” statement. The while loop in python allows looping/iterating over a block of code as long as its conditional expression(condition) evaluates to be true. The difference between this loop and the “for loop” is that the for loop iterates over the elements in a sequence, the while loop on the other hand continues looping as long as its condition is true, so the while loop is mostly used in cases where the number of times to loop is unknown. The syntax of the while loop is given below.

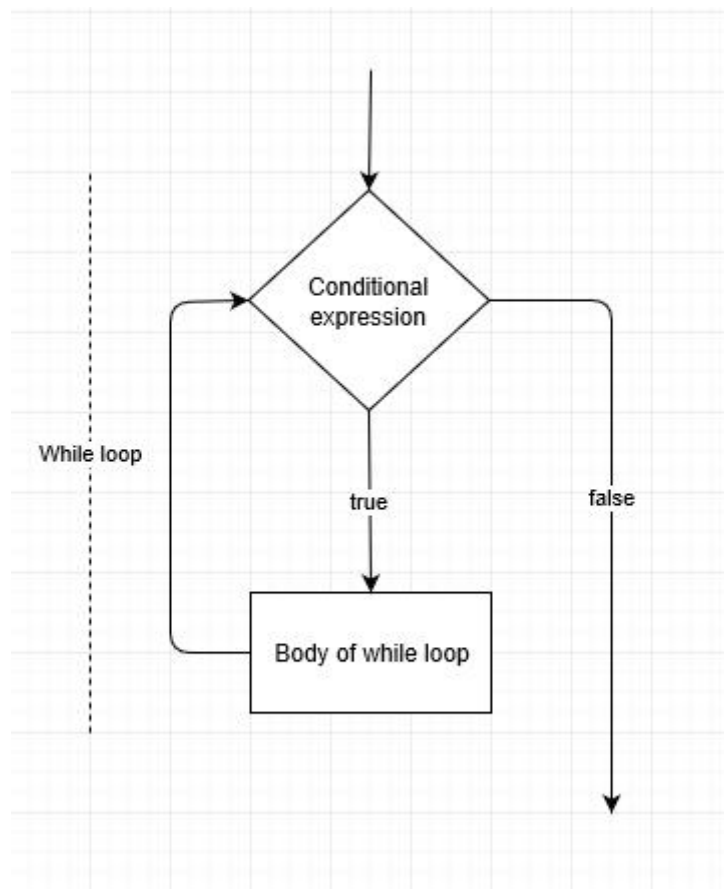
While <Boolean expression>:

<while block code>

Here as long as the <boolean expression> is evaluated to be true the <while block code> is executed. The step by step execution of the while loop is shown below.

1. The conditional expression is evaluated.
2. Executed the body of the while loop if the conditional expression is evaluated to be true.
3. Repeat steps 1 and 2 until the conditional expression is evaluated to be false.

Remember that any value other than “0” and “null” are considered to be true. The flowchart of the “while loop” is given below.



The conditional expression in the “while loop” usually contains one or many variables that is changed every time in the “while” block. This is important because without this change the “while loop” may end up looping without any end as it’s condition is always true, so its important to have one or more variables in the conditional expression of the “while loop” that approaches the terminating condition. Lets look at an example that uses the “while loop”.

```
1 #=====
2 # This program adds the first "n" natural numbers.
3 #=====
4
5 n = int(input("Enter the value of n:"))
6
7 # initialize sum and counter variables
8 i = 1
9 sum = 0
10
11 while i <= n:
12     sum = sum + i
13     i = i+1    # update counter
14
15 # print the sum of natural numbers.
16 print("The sum is", sum)
17
```

The above python script adds the first ‘n’ natural numbers(1, 2, 3, ...), where “n” is a user provided input to add until. Here ‘i’ is the counter variable, it’s value is incremented in the body of the loop so the next natural number can be added in the next iteration of the loop. The conditional expression of the while loop returns false when the value of ‘i’ equals “n + 1”, this is when the control leaves the while loop. Some of the possible outputs for the above program are given below.

When “n = 2” the output of the above program is given below.

```
Enter the value of n:2
The sum is 3
```

When “n = 3” the output of the above program is

```
Enter the value of n:3
The sum is 6
```

When “n = 10” the output of the above program is

```
Enter the value of n:10
The sum is 55
```

When “n = 50” the output of the above program is

```
Enter the value of n:50
The sum is 1275
```

8.5.1 while / else statement

The just like the “for loop” the “while loop” can have an optional “else” block, which is executed after all the elements in the sequence have been iterated through. In the case that a “break” statement was encountered during the execution of the “while loop” which causes an abrupt exit from the loop the “else” part is ignored and so the else part of the “while loop” is executed if there is no abrupt exit from the loop.

An example that uses the “else” part of the “while loop’ is given below.

```
1 #=====
2 # This program illustrates the else statement with a while loop
3 #=====
4
5 i = 0 # Initiallizing the counter of the while loop
6
7 while i < 5:
8     print(i, " Inside the loop block")
9     i = i + 1
10 else:
11     print(i, " Inside the else block")
12
```

In the above python script the conditional expression for the while loop “i < 5” is true until “i = 5” which is when the else block code is executed

The output for the above code example is given below.

```
0 ) Inside the loop block
1 ) Inside the loop block
2 ) Inside the loop block
3 ) Inside the loop block
4 ) Inside the loop block
5 ) Inside the else block
```

8.5.2 Infinite loops while

As we have already seen the “while loops” conditional expression commonly contain one or more “counter variables” that have their value changed on every iteration through the loop this allows the conditional expression to tend towards the terminating condition which when reached the control exits from the loop.

In the previously discussed example program for counting first n natural numbers the “while” block has a statement that increments the value of ‘ i ’, this is very important (and sometimes forgotten) because if missed the loop executes forever with an end, this is called an infinite loop. Try the “counting the first n natural numbers” program yourself without the statement to increment the value of the counter variable.

```
1 #=====
2 # This program adds the first "n" natural numbers
3 #=====
4
5 n = int(input("Enter the value of n:"))
6
7 # initialize sum and counter variables
8 i = 1
9 sum = 0
10
11 while i <= n:
12     sum = sum + i
13     #i = i + 1    # update counter skipped
14
15 # print the sum of natural numbers.
16 print("The sum is", sum)
17
```

When the above program is executed, it prints the output given below.

```
Enter the value of n:5
```

As you can see that after the value of ‘ n ’ is provided, the sum doesn’t get printed, this is because the control is still in the while loop which is an infinite loop. It’s possible to see that the program is still in execution by checking the “task manager”(ctrl + shift + esc on windows).


```

27
28 #----- Terminating Loop
29 while (var2 >= -10):
30     print(var2)
31     var1 -= 5
32
33 #----- Infinite Loop
34 while (var3 != 10):
35     print(var3)
36     var3 += 6
37
38 #----- Infinite Loop
39 while (1):
40     print(var1)
41

```

8.6 Break and Continue statements

Python provides two statements for abnormal termination of loops, the `break` and `continue` statements. Break and continue statements in python are used to change the control flow in a loop. Loop are used to iterate over a certain portion of the code, break statement is used to abruptly exit the loop entirely and continue is used to skip a specific iteration of the loop. Let's discuss more details about the break and continue statement.

8.6.1 The break statement

The `break` statement when encountered terminates the loop that contains it. When the `break` statement is encountered by the the python interpreter passes the control to the statement immediately after the body of the loop.

In the case of nested loops(loops embedded in other loops) the the `break` statement when encountered, terminates the loop that it was encountered at. As the `break` statement changes the control flow regardless of any condition it is an unconditional branching statement. The syntax of the break statement is given below.


`break`

The general code given below illustrates what the `break` statement does when it is encountered in a “for loop”.

```


for var in sequence:
    # codes inside for loop
    if condition:
        break
    # codes inside for loop
# codes outside for loop

```

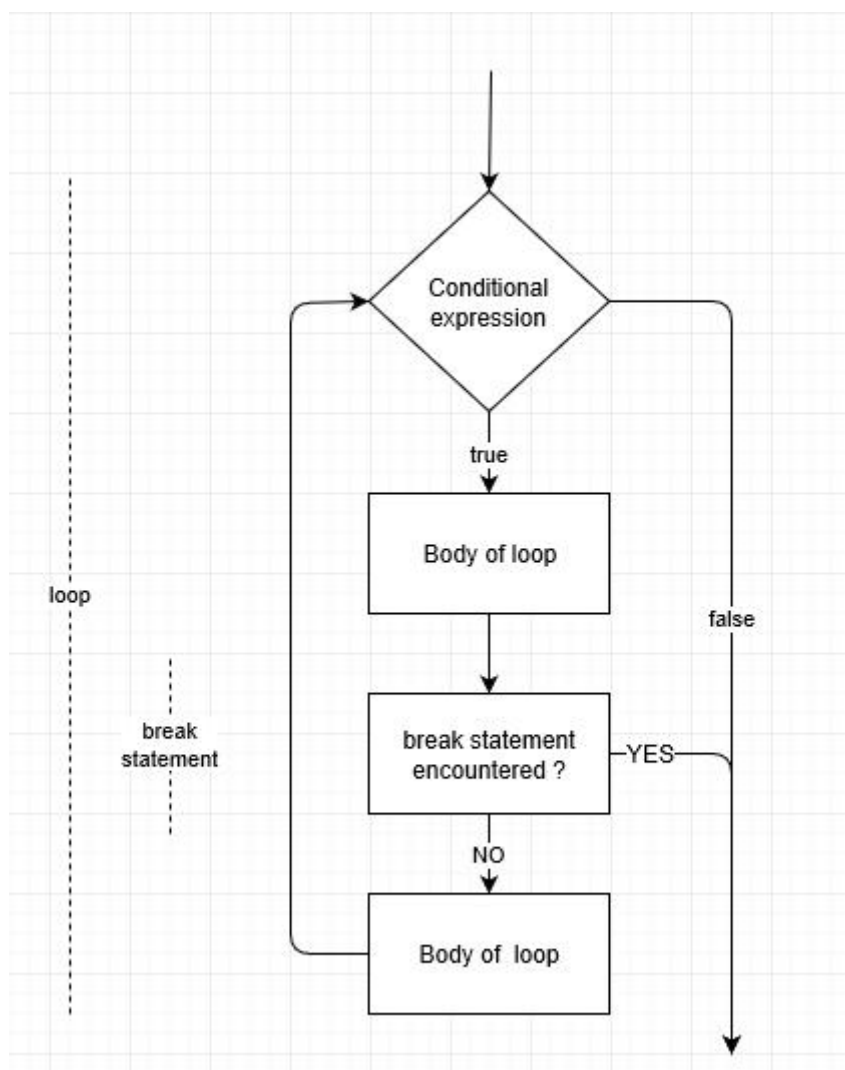


The general code given below illustrates what the `break` statement does when it is encountered in a “while loop”.

```
while test expression:
    # codes inside while loop
    if condition:
        break
    # codes inside while loop
# codes outside while loop
```



The flow chart of the `break` statement is shown below.



Lets look at an example that uses the `break` statement in a loop. In this python program the loop is abruptly terminated when the a specific letter is found.


```

1 #=====
2 # Print all charecters of a string untill the first "i" is found
3 #=====
4
5 userString = input("Enter a string : ")
6
7 for val in userString:
8     if val == "i":
9         print("Loop terminated")
10        break
11    print(val)
12 else:
13     print("The letter i was not found in your string")
14

```

In the above python script a “for loop” is used to iterate through each character of the the user provided string. On each iteration the current character is checked if it is the letter ‘i’ in the case it is not, then this character is printed, if the character is the letter ‘i’ then the python interpreter encounters the `break` statement which causes the control to exit the loop and passes the control outside the “for loop”(“else” is not executed)

The output of the above program is given below.

```

Enter a string : Any thing can be made with python
A
n
y

t
h
Loop terminated

```

8.6.2 The continue statement

The `continue` statement when encountered abruptly terminates the current iteration loop that contains it. When the `continue` statement is encountered by the the python interpreter passes the control to the statement immediately after the body of the loop.

In the case of nested loops(loops embedded in other loops) the the `continue` statement when encountered, terminates the current iteration of the loop that it was encountered at. As the `continue` statement changes the control flow regardless of any condition it is an unconditional branching statement. The syntax of the `continue` statement is given below

```
continue
```

The general code given below illustrates what the `continue` statement does when it is encountered in a “for loop”.

```

for var in sequence:
    # codes inside for loop
    if condition:
        continue
    # codes inside for loop

# codes outside for loop

```

The general code given below illustrates what the `continue` statement does when it is encountered in a “while loop”.

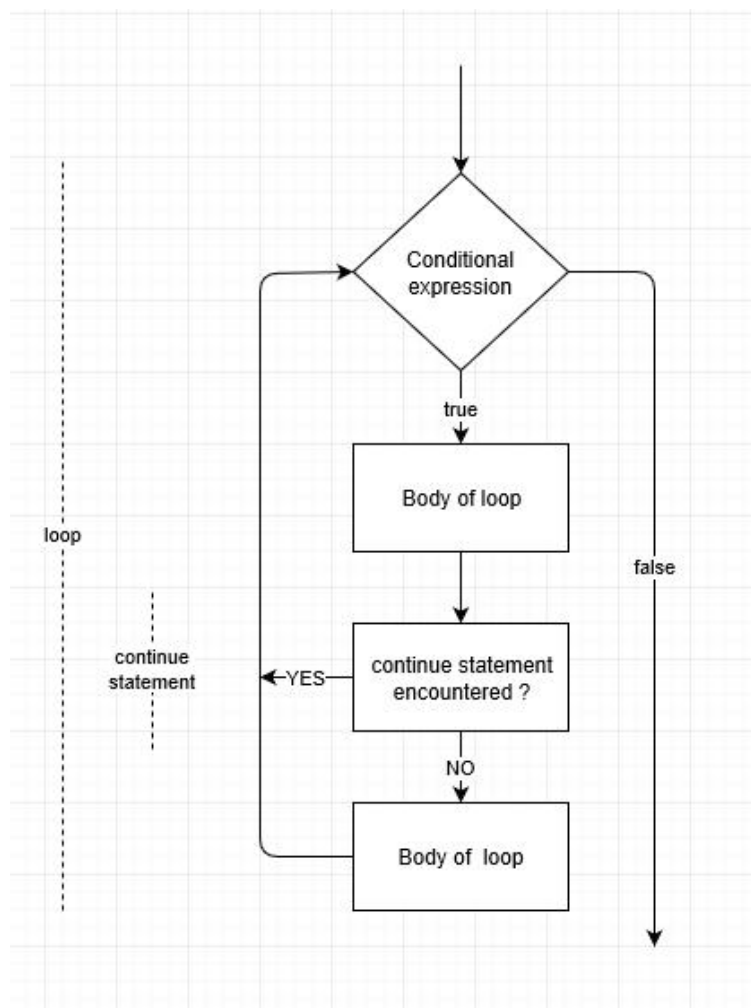
```

while test expression:
    # codes inside while loop
    if condition:
        continue
    # codes inside while loop

# codes outside while loop

```

The flowchart of the `continue` statement is shown below.



Lets look at an example that uses the `continue` statement in a loop. In this python program the an iteration that contains a specific letter is skipped.

```
1 #=====
2 # Print all charecters of a given string except i's
3 #=====
4
5 userString = input("Enter a string : ")
6
7 for val in userString:
8     if val == "i":
9         print("An \"i\" was skipped here.")
10        continue
11    print(val)
12 else:
13    print("...Done")
14
```

In the above python script a “for loop” is used to iterate through each character of the the user provided string. On each iteration the current character is checked if it is the letter ‘i’ in the case it is not, then this character is printed, if the character is the letter ‘i’ then the python interpreter encounters the `continue` statement which causes the control to exit the current iteration of the loop and passes the control to the next iteration of that loop.

```
P
y
t
h
o
n

An "i" was skipped here.
s

o
p
e
n

s
o
u
r
c
e
!
...Done
```

