

- 1 About Python
 - 1.1 Python Overview
 - 1.2 History of Python
 - 1.3 Python Features
 - 1.4 Python Environment
 - 1.5 Getting Python
 - 1.6 Case Studies
 - 1.7 Exercises
- 2 The context of Software Developments
 - 2.1 Install Python
 - 2.1.1 Unix & Linux Installation
 - 2.1.2 Windows Installation
 - 2.1.3 Macintosh Installation
 - 2.1.4 Setting up PATH
 - 2.1.5 Setting path at Unix/Linux
 - 2.1.6 Setting path at Windows
 - 2.1.7 Python Environment Variables
 - 2.2 Running python programs
 - 2.2.1 Interactive Interpreter
 - 2.2.2 Script from the Command-line
 - 2.2.3 Integrated Development Environment
 - 2.3 Python Basic Syntax
 - 2.4 First Steps Towards Programming
 - 2.5 Interactive Mode Programming
 - 2.6 Script Mode Programming
 - 2.7 Case Studies
 - 2.8 Exercises
- 3 Python Simple Basics
 - 3.1 Python Identifiers
 - 3.2 Reserved Words
 - 3.3 Lines and Indentation
 - 3.4 Multi-Line Statements
 - 3.5 Quotation in Python
 - 3.6 Comments in Python
 - 3.7 Using Blank Lines
 - 3.8 Waiting for the User
 - 3.9 Multiple Statements on a Single Line
 - 3.10 Multiple Statement Groups as Suites

- 3.11 Command-Line Arguments
- 3.12 Accessing Command-Line Arguments
 - Example
- 3.13 Parsing Command-Line Arguments
- 3.14 getopt.getopt method
- 3.15 exception getopt.GetoptError
 - Example
- 3.16 Case Studies
- 3.17 Exercises
- 4 Python Variable Types Values and Identifiers
 - 4.1 Assigning Values to Variables
 - 4.2 Multiple Assignment
 - 4.3 Standard Data Types
 - 4.4 Data Type Conversion
 - 4.4 Python Numbers
 - 4.4.1 Number Type Conversion
 - 4.4.2 Floating-point Numbers
 - 4.4.3 Mathematical Functions
 - 4.4.4 Syntax
 - 4.4.5 Parameters
 - 4.4.6 Return Value
 - Example
 - Description
 - 4.4.7 Mathematical Constants
 - 4.5 Python Strings
 - 4.5.1 Accessing Values in Strings
 - 4.5.2 Integer and String Values
 - 4.5.3 Control Codes within Strings
 - 4.5.4 Controlling the print Function
 - 4.5.5 String Formatting
 - 4.5.6 Multi-line Strings
 - 4.5.7 Updating Strings
 - 4.5.8 Escape Characters
 - 4.5.9 String Special Operators
 - 4.5.10 String Formatting Operator
 - 4.5.11 Triple Quotes
 - 4.5.12 Raw String
 - 4.5.13 Unicode String
 - 4.5.14 Built-in String Methods
 - Description
 - Syntax
 - Parameters

Return Value Example

- 4.6 User Input
- 4.7 Variables and Assignment
- 4.8 Identifiers
- 4.9 Case Studies
- 4.10 Exercises

5 Expressions and Arithmetic

- 5.1 Expressions
- 5.2 Mixed Type Expressions
- 5.3 Operator Precedence and Associativity
- 5.4 Formatting Expressions
- 5.5 Python Basic Operators
 - 5.5.1 Python Arithmetic Operators
Example
 - 5.5.2 Python Comparison Operators
Example
 - 5.5.3 Python Assignment Operators
Example
 - 5.5.4 Python Bitwise Operators
Example
 - 5.5.5 Python Logical Operators
Example
 - 5.5.6 Python Membership Operators
Example
 - 5.5.7 Python Identity Operators
Example
 - 5.5.8 Python Operators Precedence
Example
- 5.6 Case Studies
- 5.7 Exercises

6 More Control Flow Tools

- 6.1 The simple If statements
 - Syntax
 - Flow Diagram
 - Example
- 6.2 if...else statements
 - Syntax
 - Flow Diagram
 - Example
- 6.3 The elif Statement

Example
nested if statements
Syntax
Example

6.4 The while Statement

Syntax
Flow Diagram
Example

6.5 Definite Loops vs. Indefinite Loops

Syntax
Flow Diagram
Example

6.6 The for Statement

Syntax
Flow Diagram
Example

6.7 Nested Loops

Syntax
Flow Diagram
Example

6.8 Abnormal Loop Termination

The break statement
Syntax
Flow Diagram
Example

6.9 The continue statement

Syntax
Flow Diagram
Example

6.10 while/else and for/else

Syntax
Flow Diagram
Example

6.11 Infinite Loops

6.12 Iteration Examples

6.13 Multi-way Decision Statements

6.14 Multi-way Versus Sequential Conditionals

6.15 Conditional Expressions

6.16 Errors in Conditional Statements

6.17 Logic Complexity

6.18 Computing Square Root

6.19 Drawing a Tree

6.20 Printing Prime Numbers

6.21 Case Studies

6.22 Exercises

- 7 Using Functions
 - 7.1 Introduction to Functions
 - 7.2 Functions and Modules
 - 7.3 The Built-in Functions
 - 7.4 Standard Mathematical Functions
 - 7.5 time Functions
 - 7.5 Random Numbers
 - 7.6 System-specific Functions
 - 7.7 The eval and exec Functions
 - 7.7 Turtle Graphics
 - 7.8 Other Techniques for Importing Functions and Modules
 - 7.9 Writing Functions
 - 7.9.1 Function Basics
 - 7.9.2 Parameter Passing
 - 7.9.3 Documenting Functions
 - 7.9.4 Function Examples
 - 7.9.5 Better Organized Prime Generator
 - 7.9.6 Command Interpreter
 - 7.9.7 Restricted Input
 - 7.9.8 Better Die Rolling Simulator
 - 7.9.9 Tree Drawing Function
 - 7.9.10 Floating-point Equality
 - 7.9.11 Refactoring to Eliminate Code Duplication
 - 7.9.12 Custom Functions vs. Standard Functions
 - 7.10 More on Functions
 - 7.10.1 Global Variables
 - 7.10.2 Default Parameters
 - 7.10.4 Introduction to Recursion
 - 7.10.5 Making Functions Reusable
 - 7.10.6 Functions as Data
 - 7.10.7 Separating Concerns with Pluggable Modules
 - 7.10.8 Lambda Expressions
 - 7.10.9 Generators
 - 7.10.10 Local Function Definitions
 - 7.10.11 Decorators
 - 7.10.12 Partial Application
 - 7.11 Case Studies
 - 7.12 Exercises
- 8 Python Objects

- 8.1 Python Objects
- 8.2 Standard Types
- 8.3 Other Built-in Types
- 8.4 Internal Types
- 8.5 Standard Type Operators
- 8.6 Standard Type Built-in Functions
- 8.7 Categorizing the Standard Types
- 8.8 Unsupported Types
 - 8.9 Case Studies
 - 8.10 Exercises
- 9 Sequences: Strings, Lists, and Tuples
 - 9.1 Sequences
 - 9.1.1 Strings
 - 9.1.2 Strings and Operators
 - 9.1.3 String-only Operators
 - 9.1.4 String Built-in Methods
 - 9.1.5 Special Features of Strings
 - 9.1.6 Related Modules
 - 9.1.7 Summary of String Highlights
 - 9.2 Lists
 - 9.2.1 Operators
 - 9.2.2 Built-in Functions
 - 9.2.3 List Type Built-in Methods
 - 9.2.4 Special Features of Lists
 - 9.3 Tuples
 - 9.3.1 Tuple Operators and Built-in Functions
 - 9.3.2 Special Features of Tuples
 - 9.3.3 Related Modules
 - 9.3.4 *Shallow and Deep Copies
 - 9.4 Case Studies
 - 9.5 Exercises
- 10. Introduction to Dictionaries
 - 10.1 Operators
 - 10.2 Built-in Functions
 - 10.3 Built-in Methods
 - 10.4 Dictionary Keys
 - 10.5 Exercises
 - 10.6 Case Studies
 - 10.7 Exercises
- 11. Errors And Exceptions
 - 11.1 What Are Exceptions?

- 11.2 Exceptions in Python
- 11.3 Detecting and Handling Exceptions
- 11.4 *Exceptions as Strings
- 11.5 *Exceptions as Classes
- 11.5 Raising Exceptions
- 11.6 Assertions
- 11.7 Standard Exceptions
- 11.8 *Creating Exceptions
- 11.9 Why Exceptions (Now)?
- 11.10 Why Exceptions at All?
- 11.11 Exceptions and the sys Module
- 11.12 Case Studies
- 11.13 Exercises
- 12. Modules
 - 12.1 What are Modules?
 - 12.2 Modules and Files
 - 12.3 Namespaces
 - 12.4 Importing Modules
 - 12.5 Importing Module Attributes
 - 12.6 Module Built-in Functions
 - 12.7 Packages
 - 12.8 Other Features of Modules
 - 12.9 Case Studies
 - 12.10 Exercises
- 13 Classes and OOP
 - 13.1 Introduction
 - 13.2 Object-oriented Programming
 - 13.3 Classes
 - 13.4 Class Attributes
 - 13.5 Instances
 - 13.6 Core Python Programming
 - 13.7 Instance Attributes
 - 13.8 Binding and Method Invocation
 - 13.9 Composition
 - 13.10 Subclassing and Derivation
 - 13.11 Inheritance
 - 13.12 Built-in Functions for Classes, Instances, and Other Objects
 - 13.13 Type vs. Classes/Instances
 - 13.14 Customizing Classes with Special Methods
 - 13.15 Privacy
 - 13.16 Delegation

- 13.17 Related Modules and Documentation
 - 13.18 Case Studies
 - 13.19 Exercises
- 14. Files and Input/Output
 - 14.1 File Objects
 - 14.2 File Built-in Function [open()]
 - 14.3 File Built-in Methods
 - 14.4 File Built-in Attributes
 - 14.5 Standard Files
 - 14.6 Command-line Arguments
 - 14.7 File System
 - 14.8 File Execution
 - 14.9 Persistent Storage Modules
 - 14.10 Related Modules
 - 14.11 Case Studies
 - 14.12 Exercises
- 15. March Towards Implementation
 - 15.1 Example Project 1
 - 15.2 Example Project 2

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
```

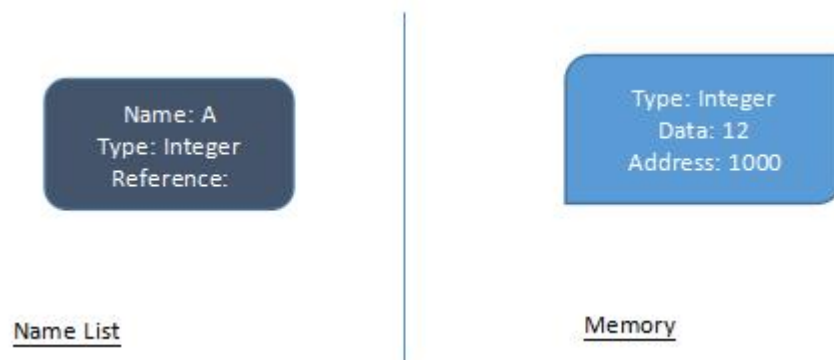
1. First an integer object 12 is created and it is stored in memory.

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.

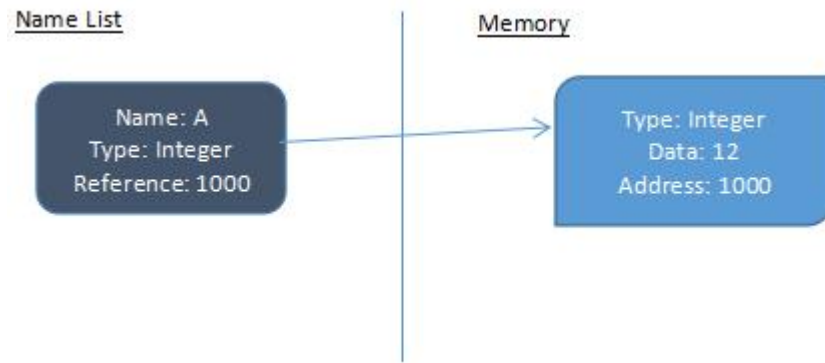
```
In [1]: A = 12
```

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

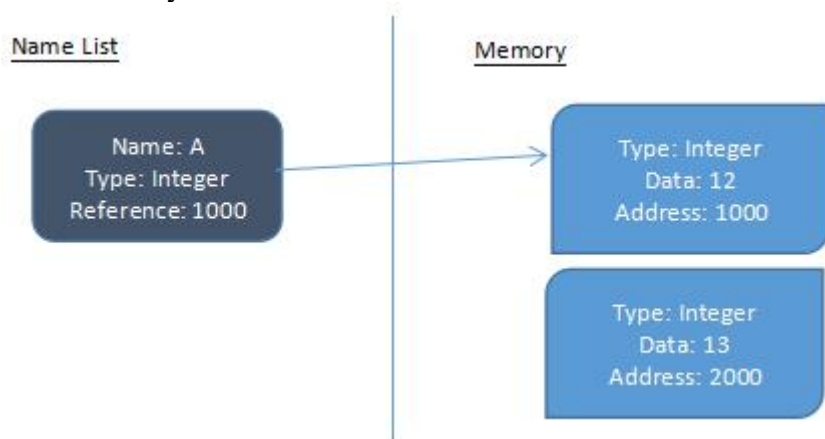
```
In [3]: A
Out[3]: 13
```

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

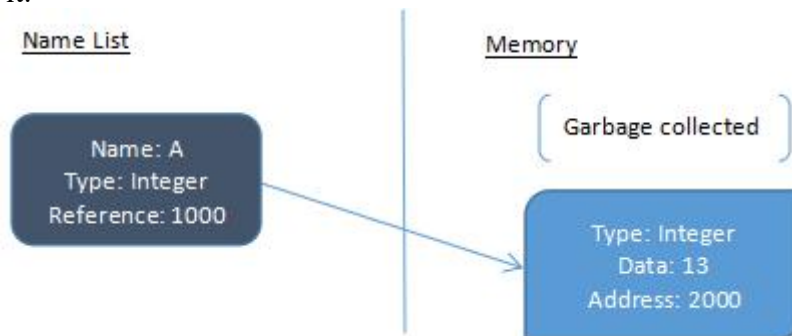
1. The reference of 'A' is looked up.
2. The value of the integer object at that reference is retrieved.



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

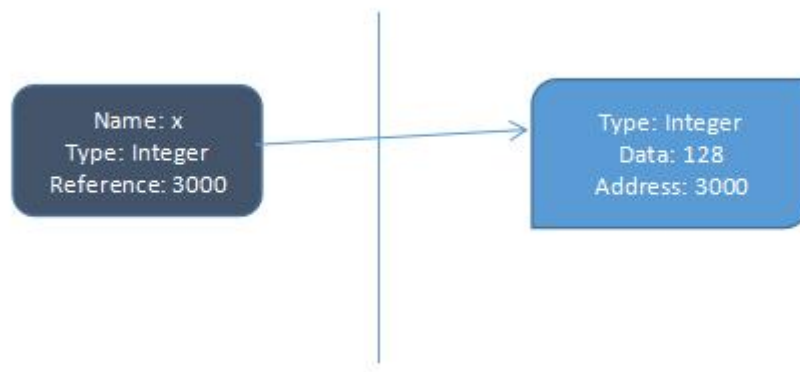


4. The variable 'A' is changed to point to this new integer object.while the previous integer object '12' is deleted by the garbage collector as there are no variables referring to it.

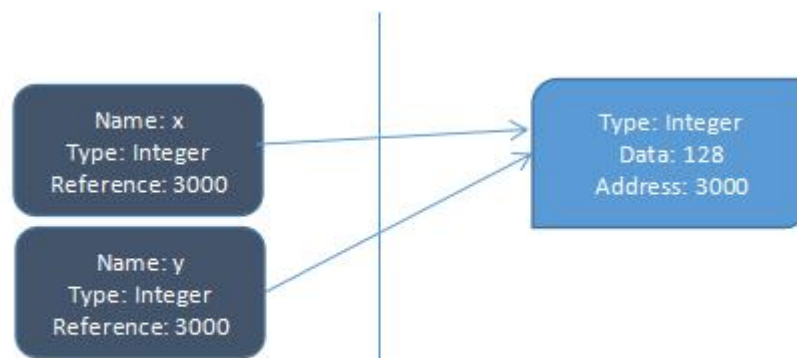


So for python's basic data types assignments behaves just as you expect them to. Take another example.

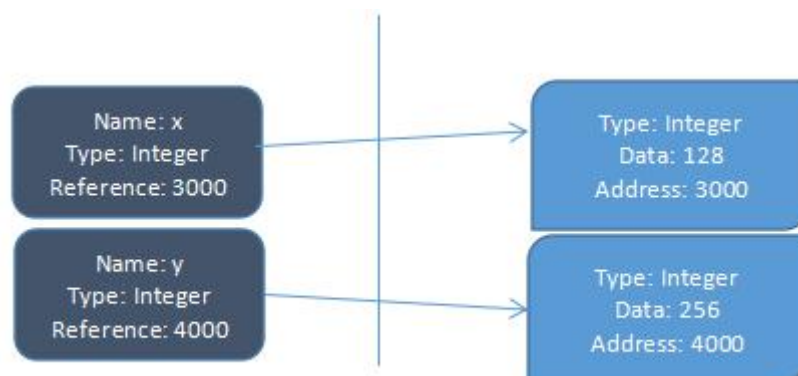
```
In [1]: x = 128
```



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



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



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

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

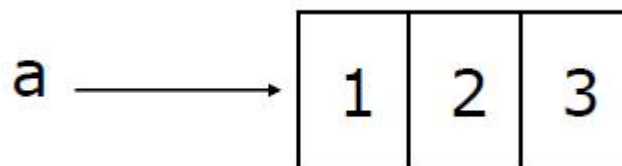
These data types that we've worked with till now are immutable(the values of the objects can't be changed). Assignments in python works differently when its done with mutable objects.

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.

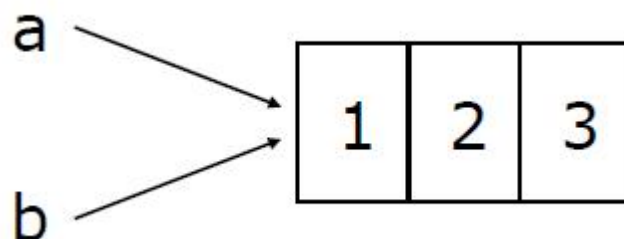
```
In [1]: a = [1,2,3]
In [2]: b = a
In [3]: b.append(4)
In [4]: a
Out[4]: [1, 2, 3, 4]
```

This is what happens when the above code is executed.

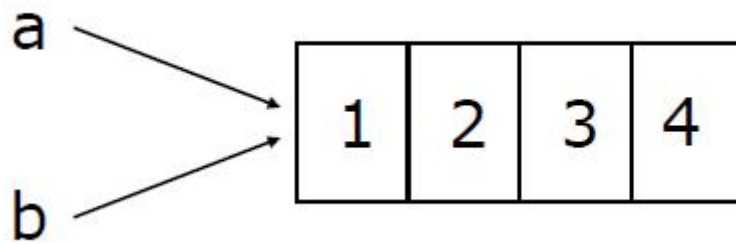
```
In [1]: a = [1,2,3]
```



```
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 a new element at any location in the list. The insert method takes two parameters, first one is the location in the list to which 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.

6.7 Exercises

- Will the following lines of code print the same thing? Explain why or why not.

```
x = 6
print(6)
print("6")
```
- Will the following lines of code print the same thing? Explain why or why not.

```
x = 7
print(x)
print("x")
```
- What is the largest floating-point value available on your system?
- What is the smallest floating-point value available on your system?
- What happens if you attempt to use a variable within a program, and that variable has not been assigned a value?
- What is wrong with the following statement that attempts to assign the value ten to variable x?

```
10 = x
```
- Once a variable has been properly assigned can its value be changed?
- In Python can you assign more than one variable in a single statement?
- Classify each of the following as either a legal or illegal Python identifier:
 - fred
 - if
 - 2x
 - 4
 - sum_total
 - sumTotal

- (g) sum-total
- (h) sum total
- (i) sumtotal
- (j) While
- (k) x2
- (l) Private
- (m) public
- (n) \$16
- (o) xTwo
- (p) _static
- (q) _4
- (r) _____
- (s) 10%
- (t) a27834
- (u) wilma's

10. What can you do if a variable name you would like to use is the same as a reserved word?
11. How is the value 2.45×10^{-5} expressed as a Python literal?
12. How is the value 0.0000000000000000000000000000449 expressed as a Python literal?
13. How is the value 569923412000000000000000000000000000000000000000 expressed as a Python literal?
14. Can a Python programmer do anything to ensure that a variable's value can never be changed after its initial assignment?
15. Is "i" a string literal or variable?

9. 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.

9.1 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.

9.2 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.

9.3 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

```

9.4 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

```

9.5 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

```

9.6 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

```

9.7 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
```

9.8 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
```

9.9 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
```

9.10 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
```


9.11 Exercises

1. Is the literal 4 a valid Python expression?
2. Is the variable x a valid Python expression?
3. Is $x + 4$ a valid Python expression?
4. What affect does the unary + operator have when applied to a numeric expression?
5. Sort the following binary operators in order of high to low precedence: +, -, *, //, /, %, =.

6. Given the following assignment:

x = 2

Indicate what each of the following Python statements would print.

- (a) `print("x")`
- (b) `print('x')`
- (c) `print(x)`
- (d) `print("x + 1")`
- (e) `print('x' + 1)`
- (f) `print(x + 1)`

7. Given the following assignments:

i1 = 2

i2 = 5

i3 = -3

d1 = 2.0

d2 = 5.0

d3 = -0.5;

Evaluate each of the following Python expressions.

- (a) `i1 + i2`
- (b) `i1 / i2`
- (c) `i1 // i2`
- (d) `i2 / i1`
- (e) `i2 // i1`
- (f) `i1 * i3`
- (g) `d1 + d2`
- (h) `d1 / d2`
- (i) `d2 / d1`
- (j) `d3 * d1`
- (k) `d1 + i2`
- (l) `i1 / d2`
- (m) `d2 / i1`
- (n) `i2 / d1`
- (o) `i1/i2*d1`
- (p) `d1*i1/i2`
- (q) `d1/d2*i1`
- (r) `i1*d1/d2`

- (s) $i2/i1*d1$
- (t) $d1*i2/i1$
- (u) $d2/d1*i1$
- (v) $i1*d2/d1$

8. What is printed by the following statement:

```
#print(5/3)
```

9. Given the following assignments:

```
i1 = 2
i2 = 5
i3 = -3
d1 = 2.0
d2 = 5.0
d3 = -0.5
```

Evaluate each of the following Python expressions.

- (a) $i1 + (i2 * i3)$
- (b) $i1 * (i2 + i3)$
- (c) $i1 / (i2 + i3)$
- (d) $i1 // (i2 + i3)$
- (e) $i1 / i2 + i3$
- (f) $i1 // i2 + i3$
- (g) $3 + 4 + 5 / 3$
- (h) $3 + 4 + 5 // 3$
- (i) $(3 + 4 + 5) / 3$
- (j) $(3 + 4 + 5) // 3$
- (k) $d1 + (d2 * d3)$
- (l) $d1 + d2 * d3$
- (m) $d1 / d2 - d3$
- (n) $d1 / (d2 - d3)$
- (o) $d1 + d2 + d3 / 3$
- (p) $(d1 + d2 + d3) / 3$
- (q) $d1 + d2 + (d3 / 3)$
- (r) $3 * (d1 + d2) * (d1 - d3)$

10. What symbol signifies the beginning of a comment in Python?

11. How do Python comments end?

12. Which is better, too many comments or too few comments?

13. What is the purpose of comments?

14. Why is human readability such an important consideration?

15. Consider the following program which contains some errors. You may assume that the comments within the program accurately describe the program's intended behavior.

```
# Get two numbers from the user
n1, n2 = eval(input()) # 1
# Compute sum of the two numbers
print(n1 + n2) # 2
# Compute average of the two numbers
print(n1+n2/2) # 3
# Assign some variables
d1 = d2 = 0 # 4
# Compute a quotient
print(n1/d1) # 5
# Compute a product
n1*n2 = d1 # 6
# Print result
print(d1) # 7
```

For each line listed in the comments, indicate whether or not an interpreter error, run-time exception, or logic error is present. Not all lines contain an error.

16. Write the shortest way to express each of the following statements.

- (a) $x = x + 1$
- (b) $x = x / 2$
- (c) $x = x - 1$
- (d) $x = x + y$
- (e) $x = x - (y + 7)$
- (f) $x = 2 * x$
- (g) $\text{number_of_closed_cases} = \text{number_of_closed_cases} + 2 * \text{ncc}$

17. What is printed by the following code fragment?

```
x1 = 2
x2 = 2
x1 += 1
x2 -= 1
print(x1)
print(x2)
```

Why does the output appear as it does?

18. Consider the following program that attempts to compute the circumference of a circle given the radius entered by the user. Given a circle's radius, r , the circle's circumference, C is given by the formula:

$$C = 2\pi r$$

```
r = 0
PI = 3.14159
# Formula for the area of a circle given its radius
```

```
C = 2*PI*r
# Get the radius from the user
r = eval(input("Please enter the circle's radius: "))
# Print the circumference
print("Circumference is", C)
```

- (a) The program does not produce the intended result. Why?
- (b) How can it be repaired so that it works correctly?

19. Write a Python program that ...

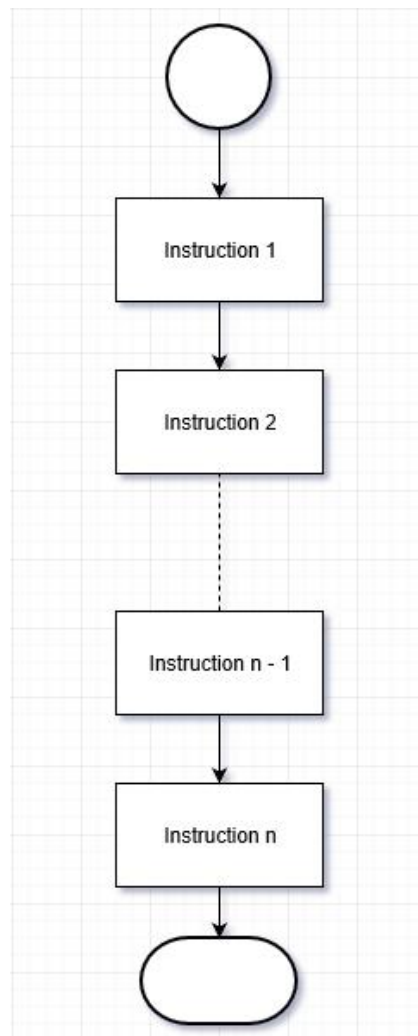
20. Write a Python program that ...

10. Python's flow control tools

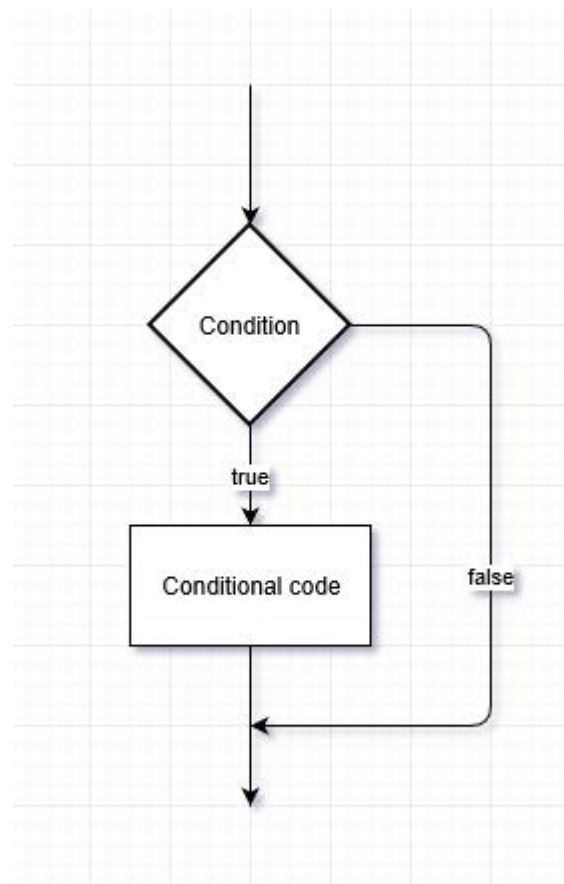
Most of the code examples that were discussed prior to this chapter followed a sequential execution of statements, from start to finish (from line 1 to that last line of code) except some examples that had to involve the “if” and “if-else” for the purpose of explanation of the content being discussed. 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).

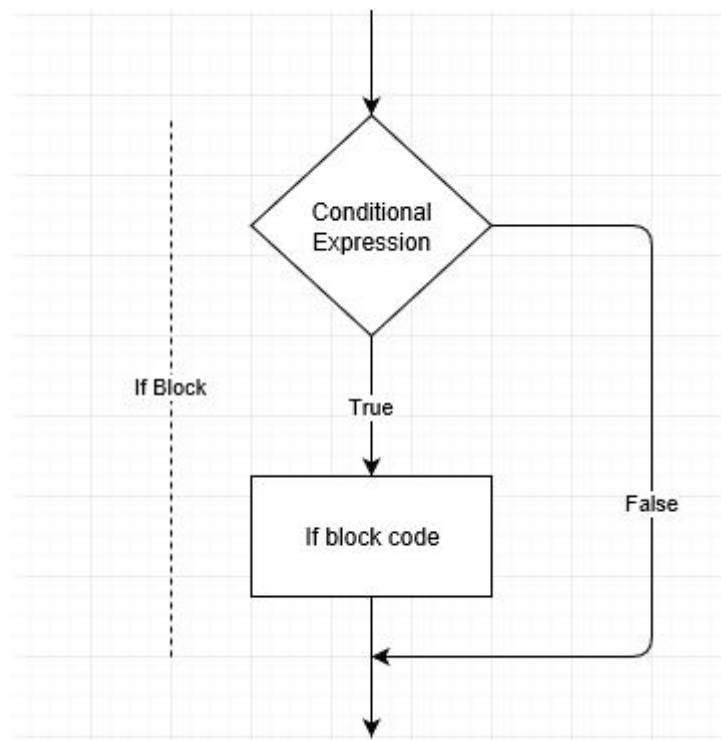
Decision making is needed in cases where a block of code should be executed only when certain condition is true, for these kind of cases that require decision making we can use “if...elif...else” statements.

10.1 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.

```

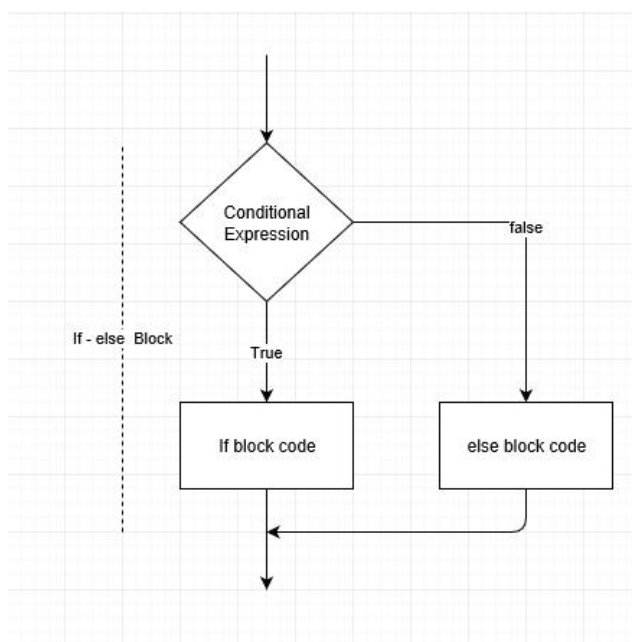
10.2 if-else statements

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
```

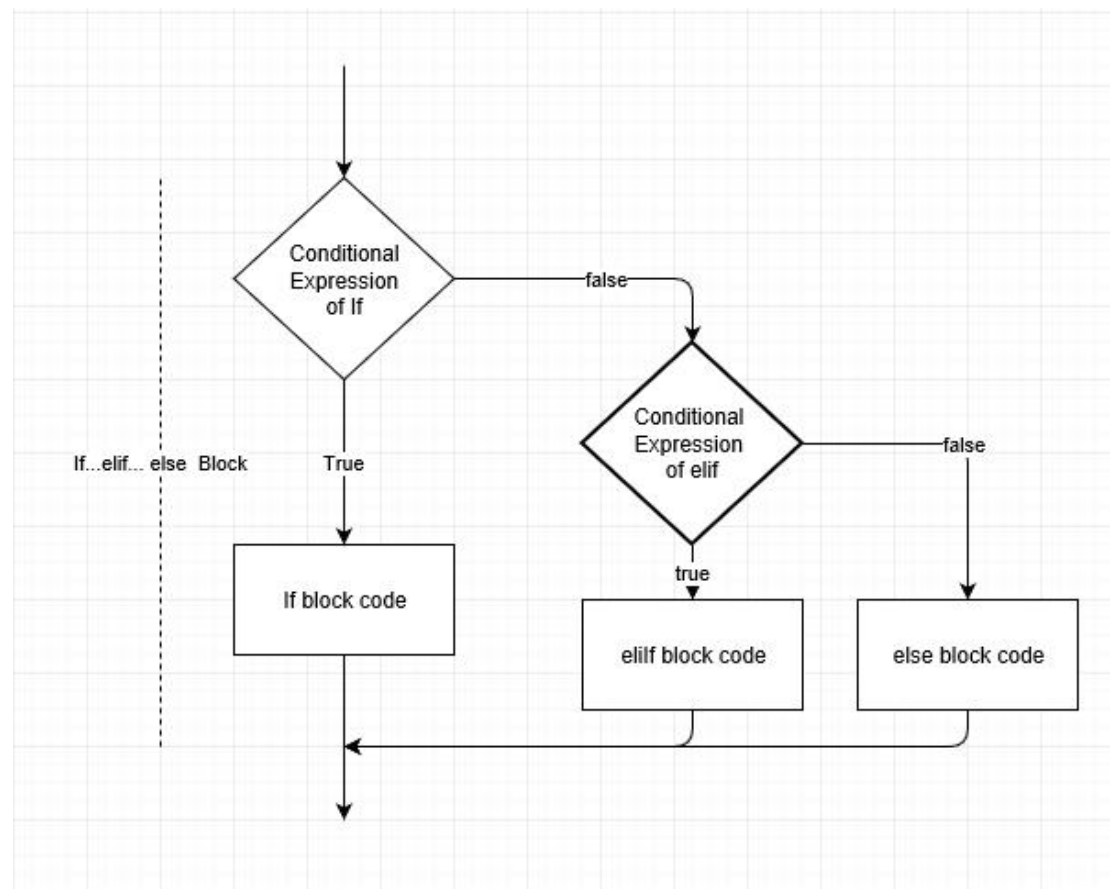
In the above program the condition of the “if statement”(num > 0) is true so the “if block” is executed and the “else” block code is skipped. If the value of num is changed to “0” the condition is still evaluated to be true so again the “if” block is executed instead. If the value of num is changed to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

10.3 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

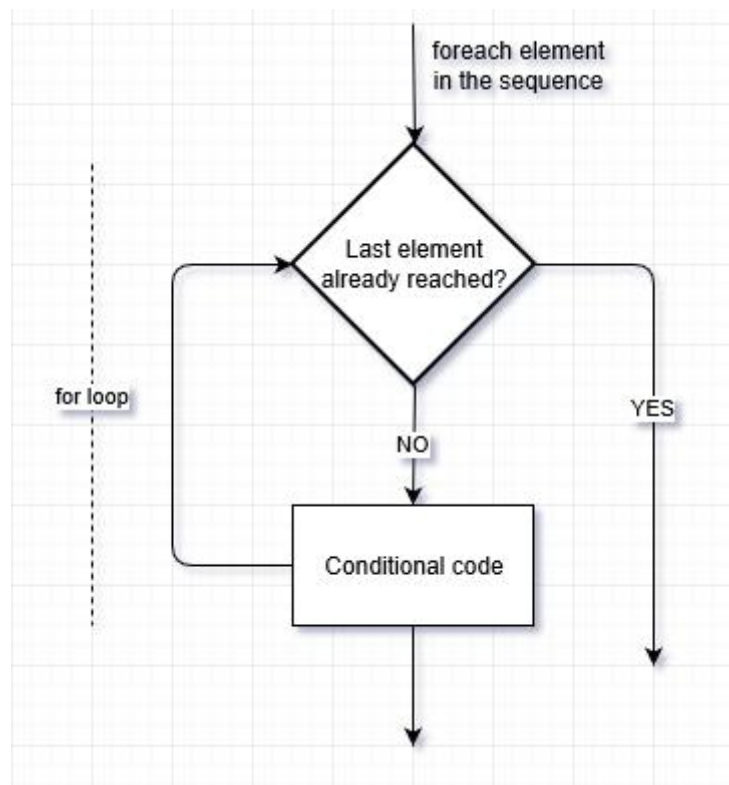
- for loop
- While loop

10.4 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
```

10.4.1 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

```

10.4.2 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!

```

10.5 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.

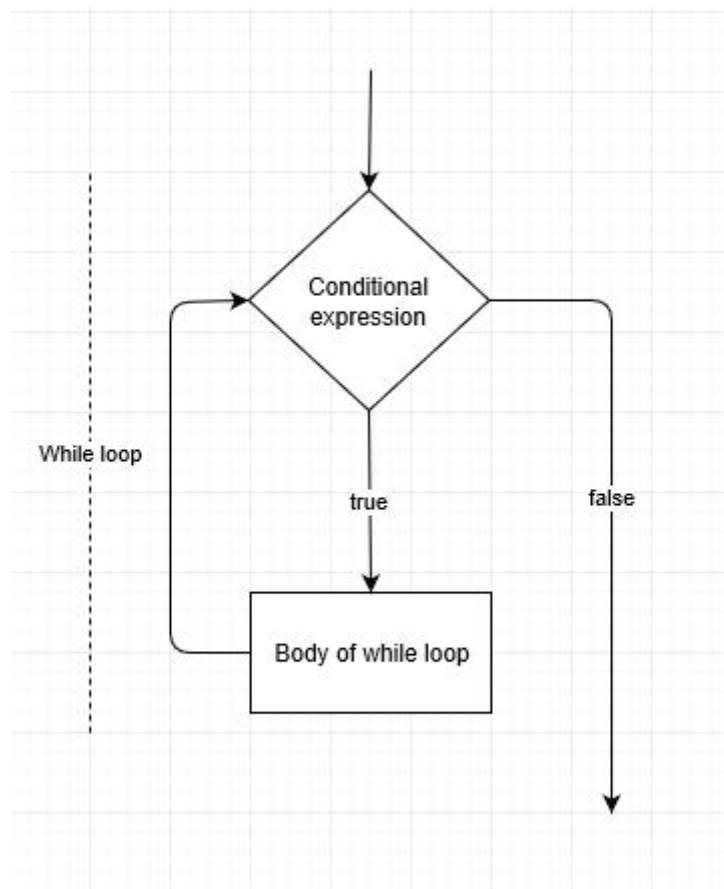
While <Boolean expression>:

<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
```

10.5.1 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
```

10.5.2 Infinite loops

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).

Task Manager			
File Options View			
Processes Performance App history Startup Users Details Services			
Name	Status	39% CPU	51% Memory
Apps (6)			
> Microsoft Edge (11)		0%	3,915.9 MB
> Python (8)		32.9%	112.4 MB
> Snipping Tool		0.4%	3.2 MB
> Task Manager		0.2%	21.2 MB
> Windows Explorer		0.3%	29.5 MB
> WPS Writer (32 bit)		0.7%	124.7 MB
Background processes (49)			
<div> <div>^</div> <div>Fewer details</div> <div>End task</div> </div>			

Lets take a look at some more examples of infinite loops with while loops.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Mar  2 15:41:08 2019
4
5 @author: Mohamed Khalid
6 """
7
8 #=====
9 # Examples of loops that terminate and infinite loops
10 #=====
11
12 # Assignments
13 var1 = 1
14 var2 = 2
15 var3 = 3
16
17 #----- Loop is never executed
18 while (0 + 1 - var1):
19     print(var1)
20     var1 += 1
21
22 #----- Terminating Loop
23
24 while (var1 <= 10):
25     print(var1)
26     var1 += 1

```

```

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

```

10.6 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.

10.6.1 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

```

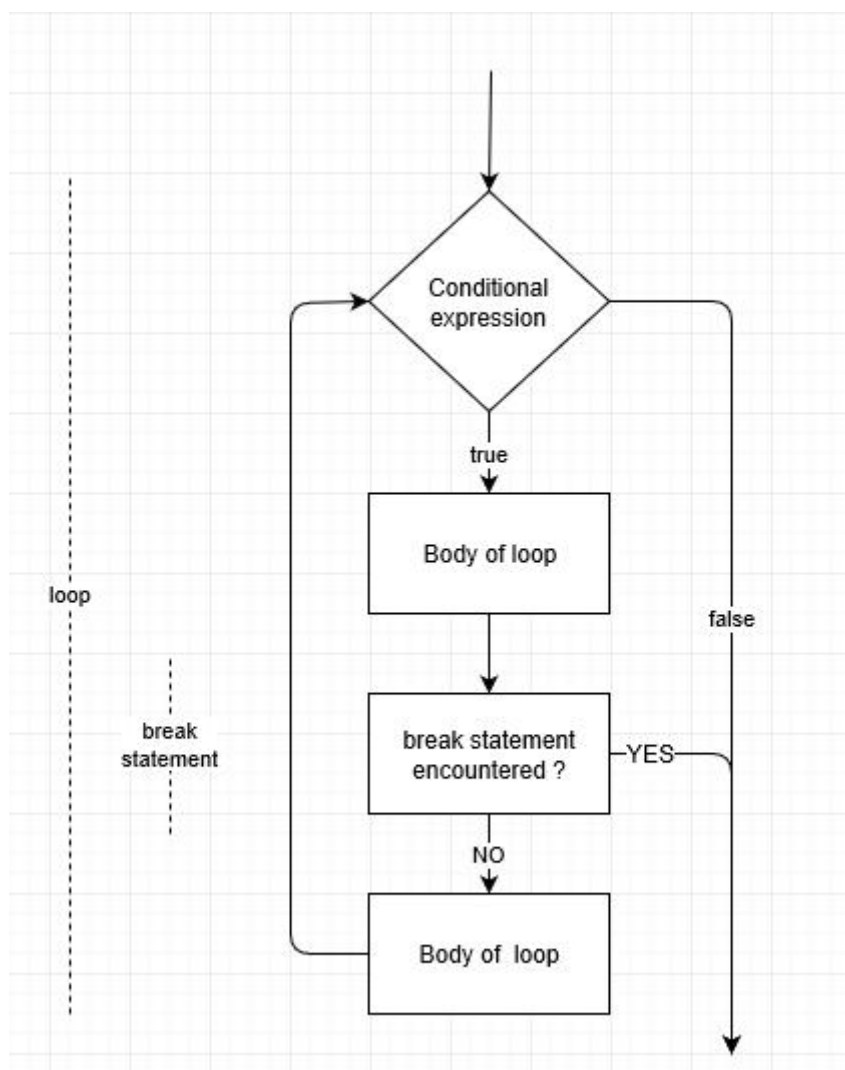


The general code given below illustrates what the `break` statement does when it is encountered in a “while loop”.

```
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

```

10.6.2 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”.

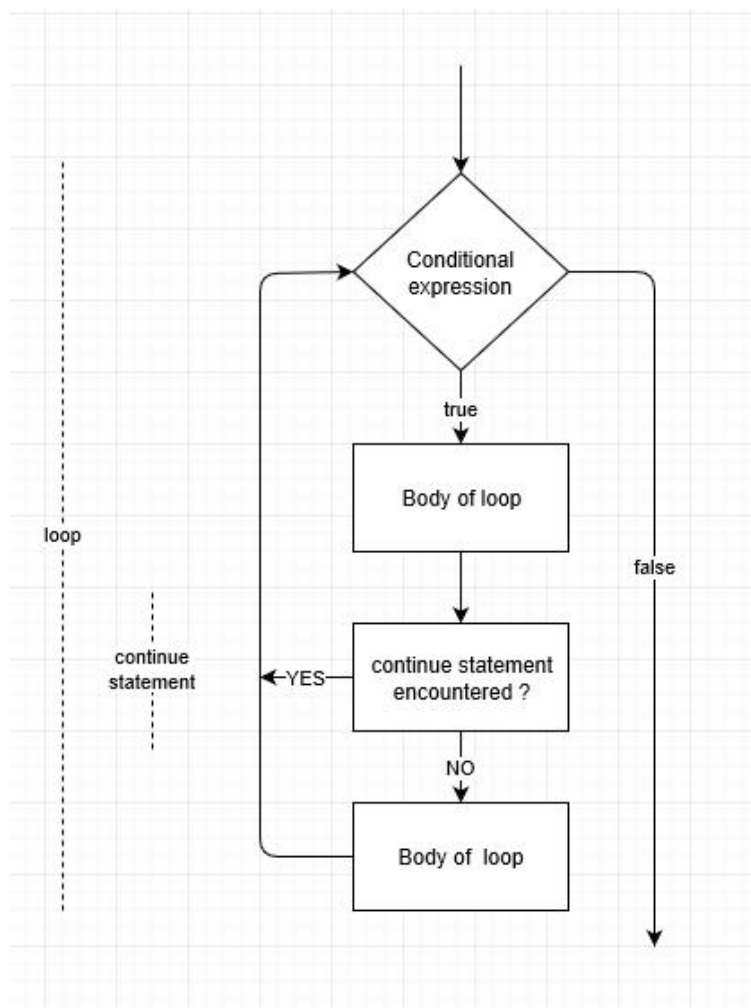
```

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

10.7 Exercises

1. What possible values can a Boolean expression have?
2. Where does the term Boolean originate?
3. What is an integer equivalent to True in Python?
4. What is the integer equivalent to False in Python?
5. Is the value -16 interpreted as True or False?

6. Given the following definitions:

x, y, z = 3, 5, 7

evaluate the following Boolean expressions:

- (a) `x == 3`
- (b) `x < y`
- (c) `x >= y`
- (d) `x <= y`
- (e) `x != y - 2`
- (f) `x < 10`
- (g) `x >= 0 and x < 10`
- (h) `x < 0 and x < 10`
- (i) `x >= 0 and x < 2`
- (j) `x < 0 or x < 10`
- (k) `x > 0 or x < 10`
- (l) `x < 0 or x > 10`

7. Given the following definitions:

b1, b2, b3, b4 = true, false, x == 3, y < 3

evaluate the following Boolean expressions:

- (a) `b3`
- (b) `b4`
- (c) `not b1`
- (d) `not b2`
- (e) `not b3`
- (f) `not b4`
- (g) `b1 and b2`
- (h) `b1 or b2`
- (i) `b1 and b3`
- (j) `b1 or b3`
- (k) `b1 and b4`
- (l) `b1 or b4`
- (m) `b2 and b3`
- (n) `b2 or b3`
- (o) `b1 and b2 or b3`
- (p) `b1 or b2 and b3`
- (q) `b1 and b2 and b3`

- (r) $b_1 \text{ or } b_2 \text{ or } b_3$
- (s) $\text{not } b_1 \text{ and } b_2 \text{ and } b_3$
- (t) $\text{not } b_1 \text{ or } b_2 \text{ or } b_3$
- (u) $\text{not } (b_1 \text{ and } b_2 \text{ and } b_3)$
- (v) $\text{not } (b_1 \text{ or } b_2 \text{ or } b_3)$
- (w) $\text{not } b_1 \text{ and not } b_2 \text{ and not } b_3$
- (x) $\text{not } b_1 \text{ or not } b_2 \text{ or not } b_3$
- (y) $\text{not } (\text{not } b_1 \text{ and not } b_2 \text{ and not } b_3)$
- (z) $\text{not } (\text{not } b_1 \text{ or not } b_2 \text{ or not } b_3)$

8. Express the following Boolean expressions in simpler form; that is, use fewer operators. x is an integer.
 - (a) $\text{not } (x == 2)$
 - (b) $x < 2 \text{ or } x == 2$
 - (c) $\text{not } (x < y)$
 - (d) $\text{not } (x <= y)$
 - (e) $x < 10 \text{ and } x > 20$
 - (f) $x > 10 \text{ or } x < 20$
 - (g) $x != 0$
 - (h) $x == 0$
9. What is the simplest tautology?
10. What is the simplest contradiction?
11. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print "OK;" otherwise, do not print anything.
12. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print "OK;" otherwise, print "Out of range."
13. Write a Python program that allows a user to type in an English day of the week (Sunday, Monday, etc.). The program should print the Spanish equivalent, if possible.
14. Consider the following Python code fragment:

```
# i, j, and k are numbers
if i < j:
    if j < k:
        i = j
    else:
        j = k
    else:
        if j > k:
            j = i
        else:
            i = k
print("i =", i, " j =", j, " k =", k)
```

What will the code print if the variables i, j, and k have the following values?

- (a) i is 3, j is 5, and k is 7
- (b) i is 3, j is 7, and k is 5
- (c) i is 5, j is 3, and k is 7
- (d) i is 5, j is 7, and k is 3
- (e) i is 7, j is 3, and k is 5
- (f) i is 7, j is 5, and k is 3

15. Consider the following Python program that prints one line of text:

```
val = eval(input())
if val < 10:
    if val != 5:
        print("wow ", end="")
    else:
        val += 1
    else:
        if val == 17:
            val += 10
        else:
            print("whoa ", end="")
            print(val)
```

What will the program print if the user provides the following input?

- (a) 3
- (b) 21
- (c) 5
- (d) 17
- (e) -5

16. Write a Python program that requests five integer values from the user. It then prints the maximum and minimum values entered. If the user enters the values 3, 2, 5, 0, and 1, the program would indicate that 5 is the maximum and 0 is the minimum. Your program should handle ties properly; for example, if the user enters 2, 4 2, 3 and 3, the program should report 2 as the minimum and 4 as maximum.

17. Write a Python program that requests five integer values from the user. It then prints one of two things: if any of the values entered are duplicates, it prints "DUPLICATES"; otherwise, it prints "ALL UNIQUE".

18. Write a Python program that ...

19. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    print('*', end='')
    a += 1
print()
```

20. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    print('*', end='')
    print()
```

21. How many asterisks does the following code fragment print?

```
a = 0
while a > 100:
    print('*', end='')
    a += 1
    print()
```

22. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    b = 0;
    while b < 55:
        print('*', end='')
        b += 1
    print()
    a += 1
```

23. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    if a % 5 == 0:

        += 1
    print()
```

24. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    b = 0
    while b < 40:
        if (a + b) % 2 == 0:
            print('*', end='')
            b += 1
        print()
        a += 1
```

25. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
    b = 0
    while b < 100:
        c = 0
```

```
while c < 100:  
    print('*', end='')  
    c++;  
    b += 1  
    a += 1  
    print()
```

26. How many asterisks does the following code fragment print?

```
for a in range(100):  
    print('*', end='')  
    print()
```

27. How many asterisks does the following code fragment print?

```
for a in range(20, 100, 5):  
    print('*', end='')  
    print()
```

28. How many asterisks does the following code fragment print?

```
for a in range(100, 0, -2):  
    print('*', end='')  
    print()
```

29. How many asterisks does the following code fragment print?

```
for a in range(1, 1):  
    print('*', end='')  
    print()
```

30. How many asterisks does the following code fragment print?

```
for a in range(-100, 100):  
    print('*', end='')  
    print()
```

11. Functions in python

Function syntax has already been introduced in the previous chapters, we have already worked with some of functions from python's standard library like `print()`, `list()`, `tuple()`, `len()`, `id()` and some other functions. The python library has several functions that can be used for common programming tasks.

Functions in mathematics are of the form $f(x) = y$. Here function 'f' when given the input 'x' performs operations on 'x' and gives 'y' as the output. An example of function a definition in mathematics is

$$f(x) = 3x + 4$$

From this function definition we can compute the values of $f(x)$ for the different values of 'x' as below.

$$f(3) = 13$$

$$f(5) = 19$$

$$f(0) = 4$$

Functions in python is a block of code that is given a name(like the name of above function is 'f'). This block of code can be called any where in the program, when called control is shifted to the body of the called function. The above function 'f' can be written in python as follows.

```
In [1]: def f(x):return 3 * x + 4
```

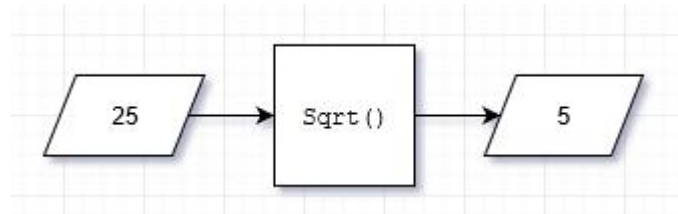
```
In [2]: f(3)
Out[2]: 13
```

```
In [3]: f(5)
Out[3]: 19
```

```
In [4]: f(0)
Out[4]: 4
```

One example of a function in python's standard library is the square root function, this function is named `sqrt`. The square root function takes numeric as parameter and returns a float value, like square root of "25" is '5'.

Functions should be thought of as tools that perform a certain operation, so it's important to know "what a function does" than "how it does it". Thinking of functions this way allows the programmer to see functions as a black box that performs the desired operation. In the case of the square root function it allows the programmer to focus on the task at hand rather than focusing on the algorithm to calculate the square root of a number, like for example to make a program that return the square root of a number provided by the user, the `sqrt()` function can be used to avoid the complex logic of computing the square root of a number. The code example below returns the square root of a number provided by the user by using the `sqrt()` function instead of implementing the logic for square root calculation.



```
1 from math import sqrt
2
3 # ask user for input
4 num = eval(input("Enter number: "))
5
6 # Compute the square root with sqrt()
7 # sqrt() is responsible for square root calculation,
8 # allowing the programmer to focus on what's important
9 root = sqrt(num);
10
11 # print the result
12 print("Square root of", num, "=", root)
13
```

Output for the above program is below.

```
Enter number: 25
Square root of 25 = 5.0
```

Here `sqrt(num)` is function invocation or a function call (a call to the `sqrt()` function). A function provides a service to its caller (invoker of the service). The above code that we have to compute square root of a number is the “calling code” or the “client code” to the `sqrt()` function which acts like a service that can be invoked.

```
root = sqrt(num)
```

The above statement invokes `sqrt()` passing it the value of `num`. The expression `sqrt(num)` is the square root of the value of the variable “num”.

Unlike the small collection of functions (like `type`, `str`, `int`, `range`, `id`) that are always available in python programs, the `sqrt()` function is available in the “math” module and can not be used unless the math module is imported, hence the purpose of the “import” statement (the first line of code).

```
from math import sqrt
```

The above code imports the `sqrt()` function from the “math” module. A module is a collection of code that can be imported and used in other python programs. The “math” module is very rich in the functions it provides to perform math operations, it provides functions for trigonometric, logarithmic and several other math operations.

A function has a name followed by a set of parentheses which contain the information that the function needs to perform its task.

```
Sqrt( num ) # sqrt needs a parameter to return the square root of
```

num is the value that the `sqrt()` function operates on. The values that are passed to a function are called the parameters or arguments of the function. Functions after they perform the computations it was defined to perform can return a value back to the caller(client invoking it) like in the case of the `sqrt()` function which returns the square root of the parameter passed as a float type to its invoker. This value that is returned to the client can be used by the client just like any other value.

```
In [3]: print(sqrt(81))
9.0
```

The `print()` function above can use the value returned by the `sqrt()` function.

```
In [4]: y = sqrt(144)
```

```
In [5]: y
Out[5]: 12.0
```

Like in the above assignment statement, the value returned by a function can be assigned to a name. When the client code attempts to pass an argument to the function it's invoking that has a type that is different then what is expected by the function, then the interpreter raises an error.

```
In [6]: sqrt('12')
Traceback (most recent call last):

  File "<ipython-input-6-b61ce7c9721f>", line 1, in <module>
    sqrt('12')

TypeError: must be real number, not str
```

A function can be called any number of times anywhere in the program. Remember to be the client of a function, the function is like a labeled black box, the client knows what the function being called does and is not concerned with how the task is performed. All functions can be treated like black boxes. A service that a function provides can be used without being concerned of the internal details.

The only way a behaviour of a function can be altered is through the functions parameters(the arguments that are passed to a function can be different). When immutable types(int, float, string, tuple) are passed as arguments to a function, a copy of the objects are passed instead of a reference to the original objects, therefore immutable objects when passed to a function as an argument can not be changed by the called function. In the case that the arguments passed are mutable objects a reference to those objects are passed instead of a new copy of those objects, therefore mutable objects when passed to a function as an argument can be changed by the invoked function.

As we already seen before functions can take more than one parameter like the `range()` function that can take one, two or three parameters.

From the view of the client/caller, a function has three parts. Lets discuss each of them in brief.

Name

A function's name is the handle that is used to identify the code to be executed when it is invoked. The same rules that apply to variable names apply to function names. Function names, like variable names are also identifiers.

Parameters

Functions are called with zero or more parameters. The types of each argument passed in the invocation of a function must be the same as the types of parameters in the function definition. If more or less parameters are passed than what is expected the interpreter raises an error. Some examples of these two kinds of errors are illustrated in the shell execution below.

```
In [1]: from math import sqrt

In [2]: sqrt("123")
Traceback (most recent call last):

  File "<ipython-input-2-9c20d16c4ba7>", line 1, in <module>
    sqrt("123")
TypeError: must be real number, not str
```

```
In [3]:

In [3]: sqrt(625,6,3)
Traceback (most recent call last):

  File "<ipython-input-3-5521980cd00a>", line 1, in <module>
    sqrt(625,6,3)
TypeError: sqrt() takes exactly one argument (3 given)
```

```
In [4]:

In [4]: sqrt()
Traceback (most recent call last):

  File "<ipython-input-4-a69d70ec25ee>", line 1, in <module>
    sqrt()
TypeError: sqrt() takes exactly one argument (0 given)
```

Return type

A function always returns a value to its caller. The client code should be compatible with the type of the object returned by called function. The return type and the parameter types are not related. Even in the case that the function does not specifically return any value, a special "none" value is return to the client.

There are functions that take no parameter like the `random()` function. The `random()` function returns a random floating point value and it belongs to the “random” module. Again if the arguments passed does not match with the number of parameters and the type of each parameter in the function definition then an error is raised by the interpreter. You can see in the below code that if a parameter is passed to the `random()` function an error is raised.

```
In [1]: from random import random

In [2]: random() # Takes no paramters
Out[2]: 0.26771707004169076

In [3]: random(10, 20) # error raised when passed arguments
Traceback (most recent call last):

  File "<ipython-input-3-a4f2f2b8f180>", line 1, in <module>
    random(10, 20) # error raised when passed arguments
TypeError: random() takes no arguments (2 given)
```

One of the ways that functions can be categorized is as,

- **Functions without side effects:** These types of functions just perform the required task and return the result to the client caller and that is all it does, it makes no other changes like printing to the display, changing the variables of the caller, etc. This type of functions always return a result(not none) to the invoking client
- **Functions with side effects:** These types of variables perform the required task which may also require to change it’s environment like printing to the display or changing the value of the caller, etc. This type of functions may sometimes return a special value “none” to the invoking client.

11.1 Standard mathematical functions

The “math” module provides almost all of the functionality of a scientific a scientific calculator. Some of he function of the “math” module a re listed below.

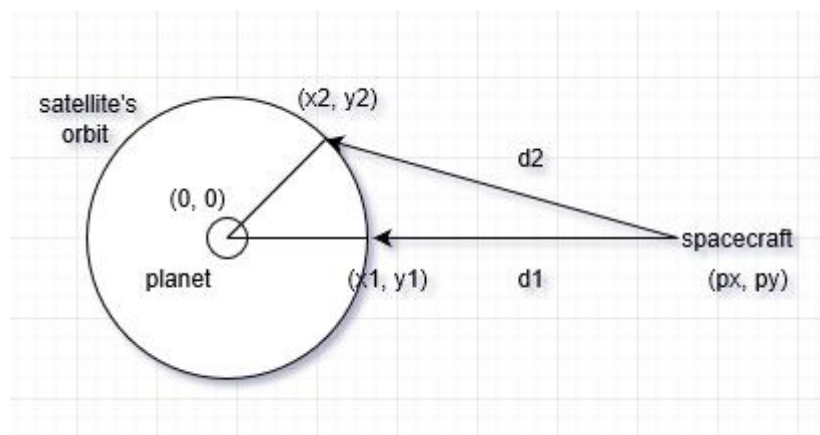
Function	description
<code>sqrt</code>	Computes the square root of a number.
<code>exp</code>	Computes e raised a power
<code>log</code>	Computes the natural logarithm of a number.
<code>log10</code>	Computes the common logarithm of a number.
<code>cos</code>	Computes the cosine of a value specified in radians: $\cos(x) = \cos x$; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent

pow	Raises one number to a power of another.
degrees	Converts a value in radians to degrees.
radians	Converts a value in degrees to radians.
fabs	Computes the absolute value of a number.

The parameters passed to the function by the calling client code is called the actual parameters(arguments) and the parameters that are specified in the function definition are the formal parameters. When a function is invoked the first actual parameter is assigned to the first formal parameter, then the second actual parameter is assigned to the second formal parameter, so the assignment of actual parameters to formal parameters happens from left to right. The order of passing the assignment matters, for example calling the `pow()` function as `pow(10, 2)` returns 100 but calling it as `pow(2, 10)` returns 1024.

Lets try to solve a trigonometry problem with the functions provided by the math module. The problem is as follows.

Suppose that there is a spacecraft at some distance from a plane. A satellite orbits this planet, the problem is to find how much farther a way the satellite would be away from the spacecraft if the the satellite moves 10 degrees along it's orbital path. The spacecraft is on the plane of the satellites orbit.



Let the center of the planet be the origin(0, 0) of the coordinate system, this is also the center of the satellites orbit. In this problem the spacecraft is located at the point (p_x, p_y) , the satellite is initially at point (x_1, y_1) then moves to point (x_2, y_2) . Here the the points (x_1, y_1) and (p_x, p_y) are given and the problem is to find the the difference between d_1 and d_2 . This problem can be solved in two steps as follows.

Step 1

Calculate the point (x_2, y_2) . For any given point (x_1, y_1) a rotation of theta degrees gives a new point (x_2, y_2) , where

$$\begin{aligned} x_2 &= x_1 \cos \theta - y_1 \sin \theta \\ y_2 &= x_1 \sin \theta + y_1 \cos \theta \end{aligned}$$

Step 2

The distance d_1 between the two points (p_x, p_y) and (x_1, y_1) is given by,

$$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

The same way, distance d_2 between the two points (p_x, p_y) and (x_2, y_2) is given by,

$$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

The python program below solves the orbital distance problem for the spacecraft at location (100, 0) and user provided starting location for the satellite. You can see that the standard math problem was used in the program to come at the solution.

```
1 #=====
2 # Python program for the orbital distance problem
3 #=====
4 from math import sqrt, cos, sin, pi
5
6 # The orbiting point is (x,y)
7 # A fixed point is always (100, 0),
8 # (p_x, p_y). Change them as necessary.
9 p_x = 100
10 p_y = 0
11
12 # number of radians in 10 degrees
13 radians = 10 * pi/180
14
15 # The cosine and sine of 10 degrees
16 cos10 = cos(radians)
17 sin10 = sin(radians)
18
19 # ask the satellites starting point from the user
20 x1, y1 = eval(input("Enter the satellites initial coordinates (x1,y1):"))
21
22 # calculate the initial distance d1
23 d1 = sqrt((p_x - x1)**2 + (p_y - y1)**2)
24
25 # calculate the new location (x2, y2) of the satellite after movement
26 x2 = x1*cos10 - y1*sin10
27
28 y2 = x1*sin10 + y1*cos10
29
30 # Compute the new distance d2
31 d2 = sqrt((p_x - x2)**2 + (p_y - y2)**2)
32
33 # Print the difference in the distances
34 print("Difference in the distances: %.5f" % (d2 - d1))
35
```

The output when the above program is executed for the starting location (10, 10) is,

```
Enter the satellites initial coordinates (x1,y1):10, 10
Difference in the distances: 2.06192
```

11.2 Time functions

Python's "time" module provides a lot of functions that involve time. In this section we will take a look at two of the functions that are provided by this module: the `clock()` and `sleep()` functions.

The `clock` function allows measuring time in seconds and it can be used anywhere in a program allowing the measure of time at any part of a program. The `clock` function works differently based on the operating system that it is being executed on. On Unix-like operating systems like Linux and Mac OS, the `clock()` function returns the number of seconds elapsed from the start of the program's execution. On Microsoft Windows the `clock()` function returns the number of seconds elapsed from the previous call to the `clock()` function. The return value is the number of seconds elapsed as a floating point number. The difference between the returned values of two `clock` function calls can be used to find the time period between the two calls.

The below program returns the amount of time it took the user to respond.

```
1 #=====
2 # This program prints the time it took the user to respond in seconds
3 #=====
4 from time import clock # import the clock method
5
6 print("User respond! : ")
7 startTime = clock()
8 input()
9 endTime = clock()
10 print("It took you %f seconds to respond " % (endTime - startTime))
11
```

An output a very slow user might get is,

```
User respond! :
Hello
It took you 18.599143 seconds to respond
```

Here is another program that displays the amount of time it took to add the first ten million natural numbers.

```
1 #=====
2 # the Time it takes to add the first ten million natural numbers
3 #=====
4 from time import clock # import the clock method
5
6 sum = 0 # initialize sum as 0
7
8 print("started adding")
9 startTime = clock()
10
11 for i in range(10000000):
12     sum += i
13
14 endTime = clock()
15 print("It took %f seconds " % (endTime - startTime))
16
```

The output of this program is given below.

```
started adding
It took 1.363418 seconds
```

The sleep function can be used to suspend a program for a period of time in seconds. The number of seconds to suspend the program for is passed as an argument to the sleep function. The below program counts down to zero from three with one second intervals.

```
1 #=====
2 # Three second count down timer!
3 #=====
4 from time import sleep # import the sleep method
5
6 timer = int(3) # initialize sum as 0
7
8 print("count down starts... %i" %timer)
9
10 while timer > 0:
11     sleep(1)
12     timer -= 1
13     print(timer)
14
15 sleep(1)
16
```

The proves to be very useful when in comes to controlling the speed in graphical animations. This program when ran produces the following output.

```
count down starts... 3
2
1
0
```

11.3 Random functions

The package random provides some functions that deal with randomness. Applications that require the behaviour of randomness need these kind of functions. Almost all random number generating algorithms produce “pseudo random” numbers which means they are not truly random. pseudo random number generators when repeated long enough produce results that start repeating the exact same sequence of numbers. This not a serious problem as the length of this sequence is long enough to be considered safe for almost all practical applications.

Pythons “random” module consists of a number of functions that enable the use of pseudo random numbers in programs. Few of the commonly used functions of this module are described below.

Function	Description
random()	Returns a pseudo random floating-point number x in the range $0 \leq x < 1$

<code>randrange()</code>	Returns a pseudo random integer value within a specified range.
<code>seed()</code>	Sets the random number seed.

The `seed()` sets the value from which the sequence of the pseudo random numbers are generated. Every time the functions `random()` and `randrange()` are called they return the next value in the sequence of pseudo random values. The program below, prints ten random numbers in the range of one to ten.

```

1 from random import randrange, seed
2
3 seed(10)
4
5 for i in range(10):
6     print(i+1,"",randrange(1, 10))
7

```

This program produces the output shown below.

```

1 ) 1
2 ) 7
3 ) 8
4 ) 1
5 ) 4
6 ) 8
7 ) 8
8 ) 5
9 ) 3
10 ) 1

```

The function `seed()` sets the “seed number” that is used to generate the pseudo random number sequence and as there is a dependency if the seed value is changed then then sequence of pseudo random random numbers will also change. This dependency is in one way a problem because if the above program is run again it produces the exact same sequence of random numbers. This is illustrated in the example program shown below.

```

1 from random import randrange, seed
2
3 seed(10) # seed is 10
4
5 for i in range(3):
6     print(i+1,"",randrange(1, 10))
7     print("-----")
8
9 seed(26) # Now change the value of the seed
10
11 for i in range(3):
12     print(i+1,"",randrange(1, 10))
13     print("-----")
14
15 seed(10) # when the seed is ten again the same sequence gets generated
16

```

```

17 for i in range(3):
18     print(i+1,"",randrange(1, 10))
19 print("-----")
20

```

From this program you can see that the same sequence of pseudo random numbers are generated for the same value of the seed. Having constant value of the seed isn't a safe way to generate random numbers, but it can be useful while testing a program. When the value of the seed is not specifically mentioned, then the system time determines the seed. This allows the pseudo random numbers generated to have a more random behaviour. The out put of the above program are shown below.

```

1 ) 1
2 ) 7
3 ) 8
-----
1 ) 4
2 ) 4
3 ) 7
-----
1 ) 1
2 ) 7
3 ) 8
-----

```

Now lets write a program that simulates the rolling of a six sided die. A die has six sides labeled one to six with one of the sides that can land on top when the die is rolled. We want to user to provide information on the number of dice to be thrown. The random number generated should lie between zero and six, so the randrange(1, 6) will be used for this, and it will be called for each die thrown. Lastly this entire setup should be placed in a loop that continue indefinitely until the user provides the number zero as input.

```

1 from random import randrange
2
3 # skiping the seed setup
4
5 sides = 6
6
7 while 1:
8     diceCount = int(input("Enter the number of dice to throw :"))
9     if diceCount == 0:
10         break
11     for i in range(1, diceCount + 1):
12         dieValue = randrange(1,sides + 1)
13         print("%i " % (dieValue), end = " " )
14

```

An output for the above program is given below.

```

Enter the number of dice to throw :2
1 3
Enter the number of dice to throw :5
5 5 5 6 4
Enter the number of dice to throw :10
2 5 6 1 6 5 1 4 4 6
Enter the number of dice to throw :0

```


11.4 Why write your own functions

All the programs that we have seen till now were very small and come nowhere near the size of practical applications. When the programs size increase the complexity of the program increases as well and it starts to get difficult to manage this complexity. Programs like the ones we have gone through are “monolithic code”, the program execution flows from start to finish, these types of programs are like one big block of code, when the size of a monolithic program grows its complexity starts to get out of hand. The solution to this is to divide the code into several almost independent smaller pieces with very specific roles. This division of a program's complexity into smaller pieces can be done using functions. A program can be made up of multiple functions, each one having the task of addressing a part of the problem and when used together solves the problem as a whole.

One big and complex monolithic code is avoided for several reasons.

- **Difficulty writing correctly:** When writing any statement, the details of the entire code must be considered. This becomes unmanageable for programmers beyond a certain point of complexity.
- **Difficulty debugging:** The larger a code block is, the harder it is to find the source of an error or an exception in that block of code. The effect of an erroneous statement might not be known until later in the program a correct statement uses the result of the erroneous statement.
- **Difficulty in modification:** Before a code sequence can be modified to achieve a different result, this code sequence needs to be understood, which becomes a very heavy task as the size of code sequence increases.

Our code can be made more manageable by dividing our code into parts and rewriting these parts of our code as functions, calling them in the program when and as needed. Using this divide and conquer strategy, a complicated code block can now be broken down into simpler code blocks each being handled by a function. The programmer can now accomplish the same original task by delegating tasks to functions, this way results in more manageable code blocks that are easy to read, write, debug and modify. Besides these advantages of structuring the code, functions provide several other advantages like reusing same code block for the same kind of task (reusability). Functions that provide similar functionality can be bundled together into reusable parts.

Although it's time saving to use library functions, it is sometimes necessary to write our own functions for custom behaviour. Fortunately python allows programmers to create their own custom functions and use them.

If the purpose of the custom functions are general enough and written in a proper manner then this function can even be used in other programs. In the following sections we'll take the first steps in writing functions in python.

11.5 Functions basics

Every python function has

1. **Function definition:** A functions definition describes every detail about that function including how many parameters and what type of parameters it takes, what the function does and also what the function returns.
2. **Function invocation(call):** After a function is defined, it can be called from anywhere in the program by passing the appropriate parameters to the function. A function can have only one definition but it can be invoked any number of times.

A function definition has three parts, they are listed below.

- **Name:** The name of a function is what is used to identify the block of code that this function contains. Function names are also identifiers, so they follow all the rules of python identifiers. One of the best ways to name a function is to give it the name of the task it is designed to do.
- **Parameters:** The parameters that the function accepts from callers is specified in the function definition. They appear with a set of parentheses() in a comma(,) separated list. The list of parameters can be left empty if the functions definition does not specify parameters to passed, these types of function do not require information from the caller to perform the task it was designed to carry out.
- **Body:** The block of indented statements that follow after the functions header is the functions body. This is what gets executed when the function call is made. If the function is required to produce a result and return it to the client, the `return` statement can be used to return the result back to the caller.

The syntax of a function definition is given below.

```
def <function_name>(<parameter list>):  
    """<docstring>"""  
    <statement(s)>
```

The above function definition syntax has the following the components.

- `def` is the keyword that signifies the start of the header of a function
- `<function_name>` is the unique name that identifies the the body of the function
- `<parameter list>` is used to pass arguments to the function. They are optional.
- `“:”` signifies the end of the function header

- `<"""docstring""">` is the documentation string that can be used to describe the purpose and functionality of the function
- `<statement(s)>` are the valid python statements that make up the body of the function. All the statements in the function body have the same indentation usually four spaces(“ ”) area used.
- A `return` statement is optional and it is used to return the result back to the caller if any

The example code below illustrates a simple function definition and calling this defined function.

```
1 def greet(name):
2     """This function prints a hello and doesnt return anything"""
3     print("Hello " + name)
4
5 greet("khalid")
6
```

In the above code example a function “greet” is defined that takes a single string(name) as a parameter and prints a specified string onto the prompt. Then the defined function is immediately called after it’s definition, this executes the body of the greet function with the passed arguments.

11.5.1 Docstring

The first line after the function header optionally has a string called the “docstring” that is sort for “document string”. The “docstring” is used for explaining what the function does.

If you have already programmed in other languages you would be aware how programmers tend to forget what their code does, so it’s always a best practice to document your code. Even if you may not use the documentation to refresh your memory on what the function does, it can help others who will be working with your code.

The docstring of a function can be accessed with “`.__doc__`” after a function name. So if the following code is entered into the python shell it returns,

```
In [3]: print (greet.__doc__)
This function prints a hello and doesnt return anything
```

11.5.2 The return statement

The return statement for functions is like how the break statement is for loops in a way that they both takes the control out of the block they were encountered at. The return statement exits the function, passes the control back to the place it was called from the calling code additionally the return statement contains followed by an expression list that is evaluated and returned back to the caller.

The syntax of the return statement is given below.

```
return <expression_list>
```

Here the expression in the `return` statement is evaluated and returned back to the client. In the case that there is no expression provided or the `return` statement itself is skipped then a special object `none` is returned back to the calling client.

For example try the following code in the python shell. Print the value that is return by the `greet` function.

```
In [4]: print(greet("Earth"))
Hello Earth
None
```

In the above call to `print`, which is passed a call to the `greet` function as the parameter. First the `greet` method is called which prints to the prompt then the `greet` method returns a “none” object back to the `print` function, the `print` function then prints the “none” object

Lets take a look at an example that illustrates the `return` statement. The program below defines and uses a function that returns the absolute value of a number that’s passed to it. You can see how the `return` evaluates the expression given to it and returns the result as well as the control back to the caller. Remember that an expression is any combination of values, variables, operators, function calls. Expressions always return an object. If there is no `return` statement in a function the execution of the function ends when the last statement of the functions code block has been executed and the “none” object as well as the control is returned back to the caller.

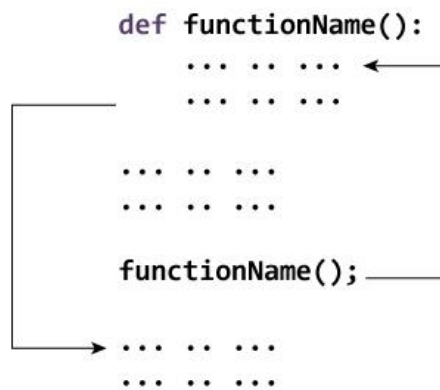
```
1 #=====
2 # Evaluate absolute value of a number
3 #=====
4 def absolute_value(num):
5     """This function returns the absolute
6     value of the number passed as an argument"""
7
8     if num >= 0:
9         return num
10    else:
11        return (-num)
12
13
14 userInput = eval(input("Enter a number: ")) # evaluate users input
15
16 print("It's absolute value is " + str(absolute_value(userInput)))
17
```

The output for the above program for the value “-56” is shown below.

```
Enter a number: -56
It's absolute value is 56
```

11.5.3 Flow of control at function call and return

The below figure illustrates the control flow when a function call is made and when the the execution of the functions code block completes.



When a function call is made the control shifts to the body of the called function, the only two ways that control can exit from a function is by the return statement or by executing the last statement in the body of the function.

11.5.4 The scope and lifetime of variables

Scope of a variable in a program is the part of the program that can recognize and use the object referred to by that variable. Parameters of a function are considered to be local to that function(local variables) and so they are in the scope of that functions code block but outside the scope of the rest of the program. A functions local variables can't be used by the parts of the program outside that functions code block hence they have local scope.

Lifetime of a variable is the period through which the variable exists in the memory. The lifetime of variables inside functions(i.e local variables) is as long as the function executes. After the function completes it's execution all it's variables are deleted and their associated objects are garbage collected, and because of this functions don't have any memory between two function calls, they don't know what the value of local variable was in the previous call.

The below example code illustrates the scope of a variable inside a function

```

1 def printLocalx():
2     x = 10      # Local variable, scope lines 2 - 3
3     print("This is printLocalx() function's value of x :",x)
4     print("This is printLocalx() function's value of y :",y)
5     # Local variable x gets deleted
6
7 x = 20         # Global variable, scope lines 1 - 8
8 y = 40         # Global variable, scope lines 1 - 8
9 printLocalx()
10 print("Value outside the function:",x)
11

```

The above program produces the below output when executed. Observe that even though the the global variable `x` is in the scope of `printLocalx()` function, the function prefers the value of it's local variable when there is a match.

```

This is printLocalx() function's value of x : 10
This is printLocalx() function's value of y : 40
Value outside the function: 20

```

Even though function `printLocalx()` changes the value of the the variable `x`, it does not affect the global variable `x`. This is because variable `x` inside the `printLocalx()` function is different from the one outside of it, so remember that they are two different variables with different scopes.

In Python the `global` keyword allows modification variables that are outside the current scope. The can be used to create global variables and modify it in the local context.

Things to remember when using the global keyword.

1. A variable is by default a local to a function when it is declared inside the function
2. A variable is by default a global variable when its declared outside a function. It makes no difference using the `global` keyword on variables defined outside functions.
3. The `global` keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the `global` keyword.

```

1 x = 20 # x is a global variable
2
3 def printx():
4     print(x) # Since there is no local variable x, the global variable x
5             # can be accessed
6
7 printx()
8

```

This example code produces the below output.

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```

1 x = 20 # x is a global variable
2
3 def printx():
4     x = x + 4 # try using the value of the global variable
5     print(x)
6
7 printx()
8

```

When the above program is ran it raises the error below, because the value of a global variable is being modified in the function. Without using the `global` keyword it is only possible to read the value of a global variable inside a function.

```

File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4

```

```

UnboundLocalError: local variable 'x' referenced before assignment

```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a `global` keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```

1 x = 20 # x is a global variable
2
3 def printx():
4     global x # Now the global variable x can be modified
5     x = x + 4 # No error raised
6     print("This is the value of x inside printx: %i" %x)
7
8 printx()
9 print("This is the value of x outside printx: %i" %x)
10

```

The output of the above program is given below. Notice that the value of the variable 'x' is the same inside and outside the `printx()` function. This is because variable 'x' inside the `printx()` function now refers to the same object as the variable 'x' outside this function.

```

This is the value of x inside printx: 24
This is the value of x outside printx: 24

```

More on the global and local scopes will be discussed in the following sections.

11.5.5 Arguments

Functions may take any number of parameters (No upper or lower bound) to perform the task that it was designed for. Before calling a user defined function, this function must be defined and its function definition must specify the parameters it needs. When a function is called by a client code, the client caller must provide the same number of arguments as specified in the function definition. If there is any mismatch in the number of arguments passed then what was expected then the python interpreter raises an error. The examples below executed in the python shell show the errors raised when there are issues with the arguments passed to a function call

```
In [1]: def square(x): return x * x # Takes 1 integer and returns it's square

In [2]: square(9, 8 , 0) # Number of arguments don't match
Traceback (most recent call last):

  File "<ipython-input-2-50d9f47ac714>", line 1, in <module>
    square(9, 8 , 0) # Number of arguments don't match

TypeError: square() takes 1 positional argument but 3 were given

In [3]:

In [3]: square(9) # A valid call
Out[3]: 81

In [4]: square("Can I be passed?") # Valid call, error raised from function body
Traceback (most recent call last):

  File "<ipython-input-4-6f0de8c10498>", line 1, in <module>
    square("Can I be passed?") # Valid call, error raised from function body

  File "<ipython-input-1-b50f9e4cba73>", line 1, in square
    def square(x): return x * x # Takes 1 integer and returns it's square

TypeError: can't multiply sequence by non-int of type 'str'
```

Observe that the number of arguments to pass are compulsory but the types of the arguments are not. Although passing an argument of a different type than what is expected would most likely end up raising an error.

Python also allows its users to define functions that take a variable number of parameters. Functions that take a variable number of parameters can be defined with the following types of arguments. We'll discuss each of them in the following section.

1. Default arguments
2. Keyword arguments
3. Arbitrary arguments

Default and non-default arguments

Python allows to assign default values for function parameters. A default value for a parameter can be given with the “=” operator. When a parameter is assigned a default value, its argument can be skipped in the function call in which case the default value provided in the function definition gets assigned instead. In the example below we create our own range function that in turn calls python’s standard range function. This example illustrates python’s default arguments.

```
1 def My_range_func(stop, start = 0, step = 1):
2     """
3     This function intrun calls
4     the standard range function
5     and returns it's result as
6     a list
7     """
8
9     li = list(range(start, stop, step))
10    return li
11
```

Now when this program is executed, try making calls to this function with different parameters from the python shell. Observe that the function call can be made with one, two or three parameters, this is because the second and third arguments are default arguments and when their values are not provided they assume their default values. Here it is necessary to pass a value for parameter `stop` and failing to do so would raise an error. For illustration there are some function calls to the `My_range_func()` given below. These function calls where made from the python shell.

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

```
In [2]: My_range_func(20, -3, 2)
Out[2]: [-3, -1, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
In [3]: My_range_func(12, 4)
Out[3]: [4, 5, 6, 7, 8, 9, 10, 11]
```

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

Here the parameter `start` does not have a default value and is mandatory to provide in a call. Default values 0 and 1 are provided for parameters `start` and `step` and so they’re value can be skipped in a function call but if provided they overwrite the default values. Any number of parameters in a function can have a default value but it must be made sure that all default values are pushed to the right of the parameter list. This is needed as without it there would be difficulty differentiating between default and compulsory arguments in a function call. To ensure this the python interpreter

raises an error when it encounters a function header that doesn't have all default parameters pushed to the right. An example is given below.

```
1 def My_range_func(start = 0, stop, step = 1): # This raises an error
```

SyntaxError: non-default argument follows default argument

Keyword and positional arguments

When a function is called by passing arguments to it, the arguments are assigned to the parameters based on its position in the argument list which is from left to right. For instance in the above function `My_range_func()` when `(20, -3, 2)` is passed as the argument list, the assignment occurs in the following order.

1. 20 gets assigned to variable `stop`
2. -3 gets assigned to variable `start`
3. 2 gets assigned to variable `step`

These arguments are positional arguments as the position of the arguments matter in the function call. Python allows re-ordering of the arguments that are passed if the arguments are passed as keyword arguments. When function call is made with keyword arguments the arguments can be passed in any order. The general syntax of argument list when passing arguments as keyword arguments is given below.

`<function_name>(<ParameterA> = <value>, <ParameterB> = <value>,...)`

The following are calls to the previously defined `My_range_func()` function and they are all valid.

```
In [2]: My_range_func(start = 3, step = 1, stop = 8)
Out[2]: [3, 4, 5, 6, 7]
```

```
In [3]: My_range_func(step = 3, stop = 10)
Out[3]: [0, 3, 6, 9]
```

```
In [4]: My_range_func(step = 3, stop = 10, start = 5)
Out[4]: [5, 8]
```

A function call can be made with a mixture of positional and keyword arguments but in this case the keyword arguments must be pushed to the right side of the argument list which means there should not be any positional arguments that follow keyword arguments in the argument list of a function call. The example below shows a valid and invalid instance of making function calls with combination of both positional and keyword arguments.

```
In [5]: My_range_func(10, step = 3, start = 5)
Out[5]: [5, 8]
```

```
In [6]: My_range_func(step = 3, 20, start = 5)
File "<ipython-input-6-ccd9242036e2>", line 1
    My_range_func(step = 3, 20, start = 5)
```

SyntaxError: positional argument follows keyword argument

Arbitrary arguments

There are many cases where the number of arguments that will be passed during a function call is unknown. An example of this case is a function that simply prints every argument it receives, but the number of arguments it is passed is arbitrary. An arbitrary parameter can be specified in the function definition by adding a '*' before the parameter which says that this parameter is a tuple, this tuple will contain the arbitrary number of arguments that will be passed. Let's take a look at an example that illustrates arbitrary arguments in python.

```
1 def sayHiTo(*people):
2     """This function says hi
3     to everyone in the people tuple."""
4
5     # names is a tuple that contains all the arguments
6     for person in people:
7         print("Hi " + person + "!")
8
9 sayHiTo("khalid", "guido von rossum")
10
```

The above program defines a function that takes an arbitrary number of arguments that is available to the function through the "people" tuple. "people" tuple is a tuple that consists of all the arbitrary arguments that are passed to the function when it's called. "Hi <person name>!" is printed for each element of the "people" tuple. The output of the above program is given below.

```
Hi khalid!
Hi guido von rossum!
```

11.5.6 Recursive functions

We already know that functions in python can call other functions, which means there also can be a case where a function calls itself. These kind of functions are called as "recursive functions" as they are defined in terms of themselves. A very common example that is used to explain recursive functions is the factorial program (program that returns the factorial of a given number). The factorial of a number is the product of all integers from one to itself.

For example factorial of five represented as "5!" is equal to "1 * 2 * 3 * 4 * 5". Let's take a look at an example that defines a recursive function to return the factorial of a given number.

```

1 #=====
2 # Python program to calculates the factorial of a number
3 #=====
4
5 def factorial(x):
6     """This is a recursive function
7     that calculates the factorial of
8     an integer"""
9
10    if x == 1: # recursive calls terminates when this condition is met
11        return 1
12    else:
13        return (x * factorial(x-1)) # The function calls itself
14
15 while 1:
16     num = eval(input("Enter an number: "))
17     if num == -1:
18         break
19     print("The factorial of", num, "is", factorial(num))
20

```

In this example the factorial() function is recursive as it makes a call to itself in the function body. Whenever the factorial function is called, it recursively calls itself decrementing the number passed as an argument each time until the terminating condition is met that is “x == 1” The results are continually evaluated from the last call to the first and the result of each function call is returned to it’s client caller until the initial call is reached. The a possible output of the above program is given below.

```

Enter an number: 1
The factorial of 1 is 1

Enter an number: 2
The factorial of 2 is 2

Enter an number: 3
The factorial of 3 is 6

Enter an number: 4
The factorial of 4 is 24

Enter an number: 5
The factorial of 5 is 120

Enter an number: -1

```

The following illustration shows the method of evaluation when the factorial() function is called with 5 passed as the argument.

```

factorial(4)           # initial call with 4
4 * factorial(3)       # second call with 3
4 * 3 * factorial(2)   # third call with 2
4 * 3 * 2 * factorial(1) # fourth call with 1
4 * 3 * 2 * 1         # fourth call returns 1
4 * 3 * 2             # third call returns 2 * 1 = 2

```

4 * 6

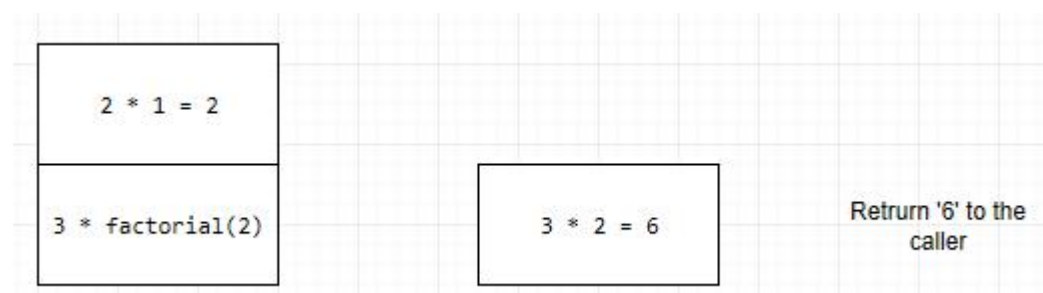
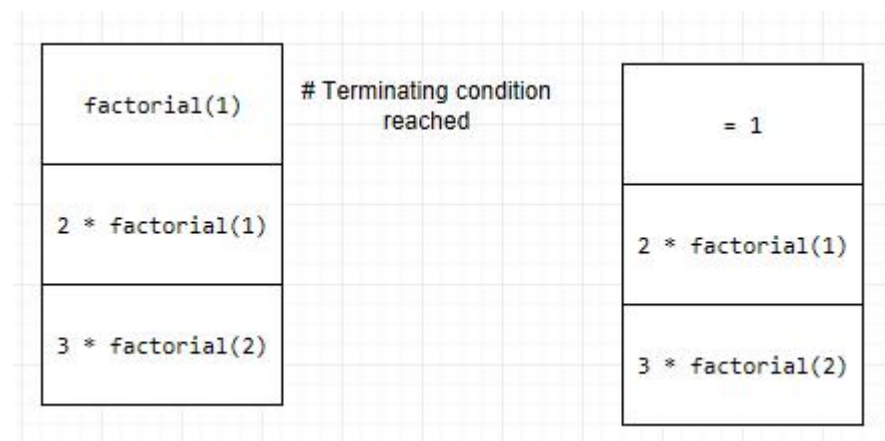
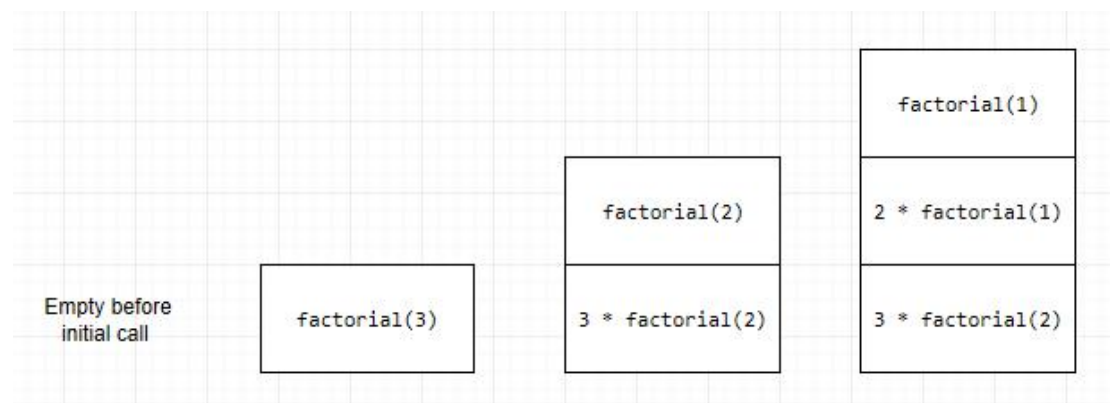
second call returns $3 * 2 = 6$

24

return from initial call $4 * 6 = 24$

24 is the value returned by factorial(4)

Recursive function calls can be easily understood in terms of pushing and popping elements from a stack, where each call to a recursive function is pushed to the top of the stack. After the last insertion to the stack has been made the elements of the stack are popped one by one till all the elements have been popped. Before popping an element at the top of the stack the function is evaluated and the if there is a any value returned by that function it is returned to it's caller in the next top of the stack. This order of evaluation continues till there are no more elements in the stack and the if any value is returned by the last element of the stack it is returned to the initial caller of the recursive function. Lets try to visualize this with the function call factorial(3) .



Advantages of Recursion:

- The use of recursive functions make the code more elegant and readable.
- Recursive functions can be used to break a complex problem down to simpler problems of a repetitive kind.
- Generating a sequence is easier using recursion than using nested iteration.

Disadvantages of Recursion:

- Sometimes the logic of recursive code can be hard to follow or understand.
- Recursive calls take up more memory than it's iterative version.
- Harder to debug

11.5.7 Lambda functions

In this section we'll take a look at what lambda functions are in python and some examples using lambda functions.

Python allows creating functions without a name called lambda functions. Like normal functions are defined using the `def` keyword lambda functions are defined using the `lambda` keyword. Lambda functions are also called as “anonymous functions”. The syntax of lambda functions in python are given below.

`Lambda <arguments>: <Expression>`

Lambda functions are allowed to have any number of arguments but can have only one expression as it's body. When a lambda function is called the expression is evaluated and returned. Lambda functions are used whenever function objects are needed. Let's take a look at an example that defines and uses a lambda function.

```
1 # Program that defines and uses a lambda function that
2 # returns the cube of a number
3
4 cube = lambda a: a * a * a # Defining the lambda function
5
6 num = eval(input("Enter a number: "))
7 print(cube(num))          # using the lambda function
8
```

In the above code `lambda a: a * a * a` where “a” is the argument and “a * a * a” is the expression that is evaluated and returned. This lambda function has no name that's why its called an anonymous function. The identifier `cube` returns the function object that is assigned to it, so it can be called a normal function.

The statement

```
cube = lambda a: a * a * a
```

Has the same meaning as the below function definition

```
def cube(a):
```



```
return a * a * a
```

Lambda functions are usually used when a nameless function is needed for a short time. They are generally used as an argument to functions that take other functions as parameters (higher order functions). Let's go through two examples that show the practical use of lambda functions in python.

Using lambda functions in the `filter()` function

The `filter()` function in python takes a function and a list for parameters. The provided function is called for all the elements of the provided list and a new list is returned with all the elements that returned true when passed to the provided function.

The example below shows the call to the “filter” function to filter out all the zero's from the provided list and returns a new list object with only non zero elements in the provided list.

```
1 #=====
2 # Program to filter out zero's from a list
3 #=====
4 my_list = [10, 0, 7, 76, 0, 0, 39, 87]
5
6 li = list(filter(lambda a: a != 0 , my_list))
7
8 print(li)
9
```

The above program prints the output given below when executed.

```
[10, 7, 76, 39, 87]
```

Using lambda functions in the `map()` function

python's `map()` function takes in a function object and a list object as its arguments and returns a list object containing all the values returned by the provided function when it's called on every element in the provided list.

The example code below makes a call to the `map()` function to return the square of all the elements in the provided list.

```
1 #=====
2 # Program to square all the elements of a given list
3 #=====
4 my_list = [1, 2, 4, 8, 16, 32, 64, 128]
5
6
7 li = list(map(lambda x: x * x , my_list))
8
9 print(li)
10
```

The output of this program is shown below.

```
[1, 4, 16, 64, 256, 1024, 4096, 16384]
```

11.5.8 Global, local and non local variables

Lifetime and scope of variables has already been introduced in the previous chapters, now let's take a look at global, local and non local variables in detail.

Global variables

The parts of a python program that exists at the surface level i.e outside of any internal code blocks like “if block”, “while block”, “function definitions”, etc are said to be the global scope and any variables that are created at that level are called global variables. Variables declared in the global scope can be accessed from any part of the program as long as they are alive. In this example, ‘x’ is declared in the global scope so it's a global variable and can be accessed from any part of the program this ability of global variable x is shown by defining a function that uses the value of ‘x’ without it having an ‘x’ as parameter nor as a variable declared inside the function.

```
1 #-----Global scope
2 x = 10      # Global variable
3 y = 20      # Global variable
4 z = 30      # Global variable
5 #-----Global scope
6
7 def foo(): # foo's definition
8     #-----foo's local scope starts
9     a = 0    # foo's local variable
10    print(a, x, y, z, b) # foo can access global variables
11    #-----foo's local scope ends
12
13 #-----Global scope
14 b = 40      # Global variable
15 foo()       # Calling foo
16 #-----Global scope
17
```

This program prints the output below. Note that the function foo can read the global variables.

```
0 10 20 30 40
```

Take a look at an interesting case where the python interpreter raises an error because it misunderstands what is actually meant.


```

1 #-----Global scope
2 x = 10      # Global variable
3 #-----Global scope
4
5 def foo(): # foo's definition
6     #-----foo's local scope starts
7     x = x + 40      # Trying to change the value of x -----!
8     a = 0          # foo's local variable
9     print(a, x) # foo can access global variables
10    #-----foo's local scope ends
11
12 #-----Global scope
13 foo()          # Calling foo
14 #-----Global scope
15

```

When the above code is executed the interpreter raises an error because, in the normal case when an assignment statement is encountered by the interpreter in a local scope, the name on the right hand side of the assignment is considered to be local variable of that local scope. So when the interpreter encounters the statement “`x = x + 40`”, it checks the left hand side, creates a new name ‘x’ in the local scope of the function “foo” then checks the expression on the right hand side of the assignment statement tries to access the non-existent object associated to the local variable ‘x’. It raises an error because ‘x’ is referenced in an expression before it’s assigned a value. When the above code is executed, it gives the following output.

```

File "C:/Users/khalid/Documents/ML/Bucca/global scope 2.py", line 7, in foo
    x = x + 40      # Trying to change the value of x -----!
UnboundLocalError: local variable 'x' referenced before assignment

```

To solve this problem of not being able to modify a global variable, python provides a global variable which will be discussed in the sections that follow.

Local Variables

A variable that is declared inside a local scope is a local variable, a variable local to the scope that it was created in. A local variable exists only in the scope that it was defined in and can not be accessed outside of that scope. Local variables are created just like global variables the only difference being that local variables are declared inside functions or are arguments of a function call and global variables are declared at the global scope of a python program. The example below creates a local variable inside function “foo”.

```

1 def foo(): # foo's definition
2     #-----foo's local scope starts
3     a = "I'm a local variable in function foo!" # foo's local variable
4     print(a)
5     #-----foo's local scope ends
6
7 foo()

```

The output prints the output given below.

Im a local variable in function foo!

In the example below the variable 'a' that is local to the function "foo" is accessed outside of the scope of the variable which results in an error.

```
1 def foo(): # foo's definition
2     #-----foo's local scope starts
3     a = 0      # foo's local variable
4     print(a)
5     #-----foo's local scope ends
6
7 #-----Global scope
8
9 print(a) #-----> a is not defined in this scope!
10
11 #-----Global scope
```

The following error is raised when the above program is executed as the variable 'a' is defined in the local scope of function "foo" and is available for use only there, it can't be accessed by the upper levels. This program raises the error shown below.

```
File "C:/Users/khalid/Documents/ML/Bucca/local scope1.py", line 9, in <module>
    print(a) #-----> a is not defined in this scope!
```

```
NameError: name 'a' is not defined
```

Global and local variables with the same name

When a variable is declared in the local scope of a function that has the same name as a variable in the global scope, which variable gets accessed? In such cases the python prefers the local variable over the global variable, so when a variable name in the local scope is accessed that has the same name as a variable in the global scope the object associated to that name in the local scope is returned. The code example below shows an instance of this case.

```
1 #-----Global scope
2 a = 10      # Global variable
3 #-----Global scope
4
5 def foo():
6     #-----foo's local scope starts
7     a = 0      # foo's local variable
8     return a    # The local variable a is accessed (a = 0)
9     #-----foo's local scope ends
10
11 #-----Global scope
12 print("The value of 'a' in foo is ",foo())
13 print("The value of 'a' in global scope is ", a)
14 #-----Global scope
```

In the above code, the same name 'a' is used both in the global scope as well as in the local scope of function "foo". When the values of both variables are printed we get different values as the local variables preferred over global if there is a name match

```
The value of 'a' in foo is 0
The value of 'a' in global scope is 10
```

Lets look back at the previous example where we try to change the value of a global variable inside a function. This causes an error to be raised by the python interpreter, and it's already been mentioned that the solution python provides for this is the "global" keyword

In Python the `global` keyword allows modification variables that are outside the current scope. The can be used to create global variables and modify it in the local context.

Here are some things to remember while using the `global` keyword.

1. A variable is by default a local to a function when it is declared inside the function
2. A variable is by default a global variable when its declared outside a function. It makes no difference using the `global` keyword on variables defined outside functions.
3. The `global` keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the `global` keyword.

```
1 x = 20 # x is a global variable
2
3 def printx():
4     print(x) # Since there is no local variable x, the global variable x
5               # can be accessed
6
7 printx()
8
```

This example code produces the below output.

```
20
```

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```
1 x = 20 # x is a global variable
2
3 def printx():
4     x = x + 4 # try using the value of the global variable
5     print(x)
6
7 printx()
8
```

When the above program is ran it raises the error below,because the value of a global variable is being modified in the function.without using the global keyword it is only possible to read the value of a global variable inside a function

```
File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4
```

```
UnboundLocalError: local variable 'x' referenced before assignment
```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a global keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first to be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```
1 x = 20 # x is a global variable
2
3 def printx():
4     global x # Now the global variable x can be modified
5     x = x + 4 # No error raised
6     print("This is the value of x inside printx: %i" %x)
7
8 printx()
9 print("This is the value of x outside printx: %i" %x)
10
```

The output of the above program is given below. notice that the value of the variable 'x' is the same inside and outside the `printx()` function. This is because variable 'x' inside the `printx()` function now refers to the same object as the variable 'x' outside this function.

```
This is the value of x inside printx: 24
This is the value of x outside printx: 24
```

The `global` keyword can also be used in nested functions, let's take a look at an example of using the `global` keyword in a nested function.

```
1 a = 10
2 print("a in the global scope before calling foo: ", a)
3
4 def foo():
5     a = 40
6     print("a in local scope before calling nested_foo: ", a)
7
8     def nested_foo():
9         global a
10        a = 20
11
12    nested_foo()
13    print("a in local scope after calling nested_foo: ", a)
14
15 foo()
16 print("a in the global scope before calling foo: ", a)
```

When the global keyword is used on the variable 'a' in the `nested_foo()` function, any change that this function makes to the variable 'a' in its scope affects the variable at the global scope. The output of the above program is given below.

```
a in the global scope before calling foo: 10
a in local scope before calling nested_foo: 40
a in local scope after calling nested_foo: 40
a in the global scope before calling foo: 20
```

Non local variables

Non local variables are used in nested functions where the local scope is not defined, so the variable can't be in either the local nor the global scope.

The keyword `nonlocal` is used to declare a non local variable in python. let's take a look at an example that explains non local variables in python.

```
1 def foo():
2     #-----foo's local scope starts
3     a = 20          # foo's local variable
4     print("a in foo: ", a)
5     #-----foo's local scope ends
6     def nested_foo():
7         nonlocal a  # Now 'a' is a non local variable
8         a = a + 20
9         print("a in nested_foo: ", a)
10    nested_foo()
11    print("a in foo: ", a)

12 #-----Global scope
13 foo()
14 #-----Global scope
```

The output of the above program is given below.

```
a in foo: 20
a in nested_foo: 40
a in foo: 40
```

Here the function `nested_foo()` is defined in the local scope of the function `foo()`. The `nonlocal` keyword is used to declare 'a' to be a non local variable. Observe that when a nonlocal variable is changed, this change also reflects on the local variable 'a' of function `foo()`. This is because the variable 'a' which is in the local scope of function `foo()` is now changed by function `nested_foo()`.

11.6 Exercises

1. Suppose you need to compute the square root of a number in a Python program. Would it be a good idea to write the code to perform the square root calculation? Why or why not?

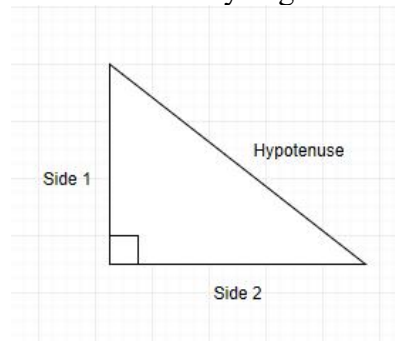
2. Which of the following values could be produced by the call `random.randrange(0, 100)` function?

4.5 34 -1 100 0 99

3. Classify each of the following expressions as legal or illegal. Each expression represents a call to a standard Python library function.

- (a) `math.sqrt(4.5)`
- (b) `math.sqrt(4.5, 3.1)`
- (c) `random.rand(4)`
- (d) `random.seed()`
- (e) `random.seed(-1)`

4. From geometry: Write a computer program that, given the lengths of the two sides of a right triangle adjacent to the right angle, computes the length of the hypotenuse of the triangle. (See Figure ??.) If you are unsure how to solve the problem mathematically, do a web search for the Pythagorean theorem.



5. Write a guessing game program in which the computer chooses at random an integer in the range 1. . . 100. The user's goal is to guess the number in the least number of tries. For each incorrect guess the user provides, the computer provides feedback whether the user's number is too high or too low.
6. Extend Problem 5 by keeping track of the number of guesses the user needed to get the correct answer. Report the number of guesses at the end of the game.
7. Extend Problem 6 by measuring how much time it takes for the user to guess the correct answer. Report the time and number of guesses at the end of the game.
8. Is the following a legal Python program?

```
def proc(x):  
    return x + 2  
def proc(n):  
    return 2*n + 1  
def main():  
    x = proc(5)  
    main()
```

9. Is the following a legal Python program?


```
def proc(x):  
    return x + 2  
def main():  
    x = proc(5)  
    y = proc(4)  
    main()
```

10. Is the following a legal Python program?

```
def proc(x):  
    print(x + 2)  
def main():  
    x = proc(5)  
    main()
```

11. Is the following a legal Python program?

```
def proc(x):  
    print(x + 2)  
def main():  
    proc(5)  
    main()
```

12. Is the following a legal Python program?

```
def proc(x, y):  
    return 2*x + y*y  
def main():  
    print(proc(5, 4))  
    main()
```

13. Is the following a legal Python program?

```
def proc(x, y):  
    return 2*x + y*y  
def main():  
    print(proc(5))  
    main()
```

14. Is the following a legal Python program?

```
def proc(x):  
    return 2*x  
def main():  
    print(proc(5, 4))  
    main()
```

15. Is the following a legal Python program?

```
def proc(x):
    print(2*x*x)

def main():
    proc(5)
    main()
```

16. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
def proc(x):
    x = 2*x*x
def main():
    num = 10
    proc(num)
    print(num)
    main()
```

17. Is the following program legal since the variable x is used in two different places (proc and main)? Why or why not?

```
def proc(x):
    return 2*x*x
def main():
    x = 10
    print(proc(x))
    main()
```

18. Is the following program legal since the actual parameter has a different name from the formal parameter (y vs. x)? Why or why not?

```
def proc(x):
    return 2*x*x
def main():
    y = 10
    print(proc(y))
    main()
```

19. Complete the following distance function that computes the distance between two geometric points (x1,y1) and (x2,y2):

```
def distance(x1, y1, x2, y2):
    ...
```

Test it with several points to convince yourself that is correct.

20. What happens if a client passes too many parameters to a function?

12. Python Modules

Modules in python allow a logical organization of code. When code gets too large to manage, as already discussed it's best break this code into pieces that can interact with each other, each piece having it's own independent functionality. These pieces generally have some attribute that have some relation to each other, this could be that they belong to the same class of member variables and member methods or that they belong to the same set of similar but independent functions. It would very advantageous if this code could be shared and reused in other python programs by the same or a different programmer. For this python gives modules the ability to bring in code that's already been written and this is an important way that python provides code reusability. This method of including the code in other modules in another program is called "importing". In a single sentence a module can be defined as "self contained and organized pieces of python code that can be shared".

In python modules are in the form of a python file i.e with a ".py" extension, so any file with a ".py" extension and has python code is considered python module. Modules can be used to break down large and unmanageable code into a more structured and organized set of pieces that can be reused later in other python files.

An example of the use of function's is that all your most used functions can be defined in a python file and imported in any other python file without the need to copy all the function definitions. Let's take a look at an example that creates a python module that defines a function that returns the square of provided number.

```
1 #=====
2 # Module "MyModule.py"
3 #=====
4
5 def square(x):
6     """This function returns
7     the square of the given number"""
8
9     result = x * x
10    return result
```

This module named "MyModule" has a function "square" defined inside it, this function takes an numeric value as it's parameter and returns it's square. A module can be imported into another module or to the python shell, allowing them to use the attributes(classes and functions) that are defined inside the module. To do this python provides the `import` keyword. Try importing our previously defined module in to the python shell with the following statement(make sure the shell runs in the same directory that the module exists in).

```
In [1]: import MyModule
```

This statement doesn't add the names of the module attributes(functions and classes) to the current symbol table directly, it only adds the name "MyModule" to it, so the function defined in that module can't directly be invoked. It can however be invoked using the module name(`MyModule`) and the dot operator(`.`) like in the example below.

```
In [1]: import MyModule    # Importing MyModule to the shell

In [2]: # Now MyModule is in the current symbol table

In [3]: square(10)        # Doesn't work, "square" isn't in symbol table
Traceback (most recent call last):
```

```
File "<ipython-input-3-8d5d868e200c>", line 1, in <module>
    square(10)    # Doesn't work, "square" isn't in symbol table
```

```
NameError: name 'square' is not defined
```

```
In [4]:
```

```
In [4]: MyModule.square(20) # Access with module name and dot operator
Out[4]: 400
```

Just like the module that we just made python provides several standard modules that are already available to use. The full list of python standard module and its description is available in the documentation at “python.org” or you can take this link docs.python.org/3/py-modindex.html.

As the standard modules are just modules that are already written, the method to import them is the same as importing user defined modules. Let’s discuss the different methods to import python modules.

The import statement

Modules can be imported with the `import` keyword, after importing the attributes that were defined in the function can be accessed by the module that it was imported into with the module’s name and the dot operator(`.`) as described in the previous section. Let’s take an example of importing one of python’s standard modules that we have already dealt with before.

```
1 import math
2
3 num = eval(input("Enter a number: "))
4 num_root = math.sqrt(num) # Using the "sqrt()" function
5                           # defined in the "math" module
6
7 print("The square root of ", num, " is ", num_root)
```

The output of this program when it is executed is given below.

```
Enter a number: 12000
The square root of 12000 is 109.54451150103323
```

import by renaming

When a module is imported with the `import` statement, the module’s name becomes the handle to access the attributes that are defined in the module. Python allows renaming the handle used to access the contents of a module by renaming the module handle at the import statement. This can be done with the `as` keyword that python provides. An example of it’s use is shown below.

```
1 import math as m
2
3 num = eval(input("Enter a number: "))
4
5 num_root = m.sqrt(num) # Observe that the module handle is 'm'
6 num_root = math.sqrt(num) # math is not recognised
7
8 print("The square root of ", num, " is ", num_root)
```

In this program the handle for the “math” module that was imported is now ‘m’ so the name “math ” no longer recognized in our scope, making `math.sqrt()` an invalid call and `m.sqrt()` a valid call.

The from..import statement

Python allows importing specific names from a module instead of instead of importing the entire module. This can be done with the `from..import` statement. The general syntax for the `from..import` statement is given below.

```
from <module_name> import <name>
```