



Python Notes 1-5 units

BCA (University of Madras)



Scan to open on Studocu

CORE - I PROBLEM SOLVING USING PYTHON
I YEAR / I SEM
SUBJECT CODE : _____

UNIT – I

Introduction: The essence of computational problem solving – Limits of computational problem solving-Computer algorithms-Computer Hardware-Computer Software-The process of computational problem solving-Python programming language - Literals - Variables and Identifiers - Operators - Expressions and Data types.

UNIT - II

Control Structures: Boolean Expressions - Selection Control - If Statement- Indentation in Python- Multi-Way Selection -- Iterative Control- While Statement- Infinite loops- Definite vs. Indefinite Loops- Boolean Flags and Indefinite Loops. Lists: List Structures - Lists in Python - Iterating over lists in Python.

UNIT - III

Functions: Program Routines- Defining Functions- More on Functions: Calling Value-Returning Functions- Calling Non-Value-Returning Functions- Parameter Passing - Keyword Arguments in Python - Default Arguments in Python-Variable Scope.

UNIT - IV

Objects and their use: Software Objects - Turtle Graphics – Turtle attributes-Modular Design: Modules - Top-Down Design - Python Modules - Text Files: Opening, reading and writing text files - String Processing - Exception Handling.

UNIT - V

Dictionaries and Sets: Dictionary type in Python - Set Data type. Object Oriented Programming using Python: Encapsulation - Inheritance – Polymorphism. Recursion: Recursive Functions.

TEXT BOOK:

1. Charles Dierbach, “Introduction to Computer Science using Python - A computational Problem solving Focus”, Wiley India Edition, 2015.

REFERENCE BOOKS:

1. Mark Lutz, “Learning Python Powerful Object Oriented Programming”, O’reilly Media 2018, 5th Edition.
2. Timothy A. Budd, “Exploring Python”, Tata McGraw Hill Education Private Limited 2011, 1st Edition.
3. Allen Downey, Jeffrey Elkner, Chris Meyers, “How to think like a computer scientist: learning with Python”, 2012.
4. Sheetal Taneja & Naveen kumar, “Python Programming a Modular approach – A Modular approach with Graphics, Database, Mobile and Web applications”, Pearson, 2017.
5. Ch Satyanarayana M Radhika Mani, B N Jagadesh, “Python programming”, Universities Press 2018.

WEB REFERENCES

- <http://interactivepython.org/courselib/static/pythonds>
- <http://www.ibiblio.org/g2swap/byteofpython/read/>
- <http://www.diveintopython3.net/>
- <http://greenteapress.com/wp/think-python-2e/>
- NPTEL & MOOC courses titled Python programming
- http://spoken-tutorial.org/tutorial-search/?search_foss=Python&search_language=English
- <http://docs.python.org/3/tutorial/index.html>

UNIT - /

UNIT - I

S.No.	Topic	Page No.
1.	Define computational problem solving.	
2.	Define computation.	
3.	Define algorithm.	
4.	Define essence of computational problem solving.	
5.	Explain limits of computational problem solving.	
6.	Explain algorithm.	
7.	Define hardware.	
8.	Explain digital computing.	
9.	Explain computer components.	
10.	Explain operating system.	
11.	Define software.	
12.	Define syntax.	
13.	Define semantics.	
14.	Define compiler.	
15.	Define interpreter.	
16.	Differentiate syntax and semantic errors	
17.	Define program debugging.	
18.	Explain computational problem solving	
19.	Explain about python.	
20.	Define basic I/O operations	
21.	Explain literals.	
22.	Explain variables.	
23.	Explain identifiers	
24.	Explain operators.	
25.	Explain expressions.	
26.	Explain data types.	

I – CHAPTER: INTRODUCTION FUNDAMENTAL

1.1 Computer Science

- Computer science is fundamentally about is **computational problem solving** —that is, solving problems by the use of computation.
- There are various areas of study in computer science including software engineering (the design and implementation of large software systems), database management, computer networks, computer graphics, computer simulation, data mining, information security, programming language design, systems programming, computer architecture, human–computer interaction, robotics, and artificial intelligence, among others.

Computation

- One characterization of computation is given by the notion of an **algorithm**.

Algorithm

- An algorithm to be a series of steps that can be systematically followed for producing the answer to a certain type of problem.

The Essence of Computational Problem Solving

To solve a problem computationally, two things are needed:

- Representation
- Algorithm

A **representation** that captures all the relevant aspects of the problem, and an **algorithm** that solves the problem by use of the representation.

Limits of Computational Problem Solving

1.2 Computer Algorithms

Algorithms

An **algorithm** is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time.

Algorithms and Computers: A Perfect Match

Most require the use of computers either because they would require too much time for a person to apply, or involve so much detail as to make human error likely. Because computers can execute instructions very quickly and reliably without error, algorithms and computers are a perfect match.

EX:

To determine the day of the week for a given **month**, **day**, and **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits + floor(year_digits / 4)**
4. If **century_digits** equals 18, then add 2 to **value**, else if **century_digits** equals 20, then add 6 to **value**.
5. If the **month** is equal to January and **year** is not a leap year, then add 1 to **value**, else,
if the **month** is equal to February and the **year** is a leap year, then add 3 to **value**; if not a leap year, then add 4 to **value**, else,
if the **month** is equal to March or November, then add 4 to **value**, else,
if the **month** is equal to May, then add 2 to **value**, else,
if the **month** is equal to June, then add 5 to **value**, else,
if the **month** is equal to August, then add 3 to **value**, else,
if the **month** is equal to October, then add 1 to **value**, else,
if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to **(value + day) mod 7**.
7. If **value** is equal to 1, then the day of the week is Sunday; else
if **value** is equal to 2, day of the week is Monday; else
if **value** is equal to 3, day of the week is Tuesday; else
if **value** is equal to 4, day of the week is Wednesday; else
if **value** is equal to 5, day of the week is Thursday; else
if **value** is equal to 6, day of the week is Friday; else
if **value** is equal to 0, day of the week is Saturday

1.3 Computer Hardware

Computer hardware comprises the physical part of a computer system. It includes the all-important components of the central processing unit (CPU) and main memory. It also includes peripheral components such as a keyboard, monitor, mouse, and printer.

1.3.1 Digital Computing: It's All about Switches

- It is essential that computer hardware be reliable and error free. The key to developing reliable systems is to keep the design as simple as possible.

- In digital computing, all information is represented as a series of digits. We are used to representing numbers using base 10 with digits 0–9.
- All information within a computer system is represented using only two digits, 0 and 1, called **binary representation**.
- Computer hardware, therefore, is based on the use of simple electronic “on/off” switches called **transistors** that switch at very high speed.
- **Integrated circuits** (“chips”), the building blocks of computer hardware, are comprised of millions or even billions of transistors.

1.3.2 The Binary Number System

For representing numbers, any base (radix) can be used. For example, in base 2, there are two possible digits (0, 1), in which each column value is a power of 2.

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	0	0	1	1
$0 + 64 + 32 + 0 + 0 + 0 + 2 + 1 = 99$							

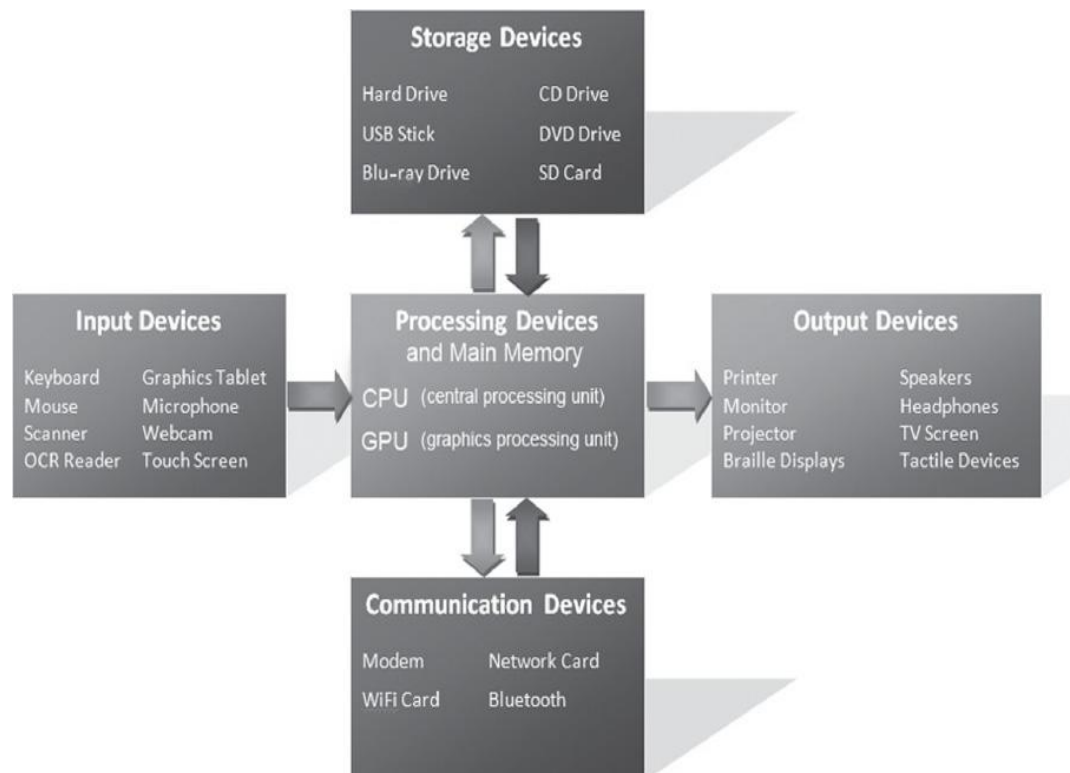
The term **bit** stands for **bi** nary digit. Therefore, every bit has the value 0 or 1. A **byte** is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits.

99/2	=	49,	with remainder 1	
49/2	=	24,	with remainder 1	
24/2	=	12,	with remainder 0	
12/2	=	6,	with remainder 0	
6/2	=	3,	with remainder 0	
3/2	=	1,	with remainder 1	
1/2	=	0,	with remainder 1	

1.3.3 Fundamental Hardware Components

- **Central Processing Unit (CPU)** is the “brain” of a computer system, containing digital logic circuitry able to interpret and execute instructions.
- **Main memory** is where currently executing programs reside, which the CPU can directly and very quickly access. Main memory is volatile; that is, the contents are lost when the power is turned off.

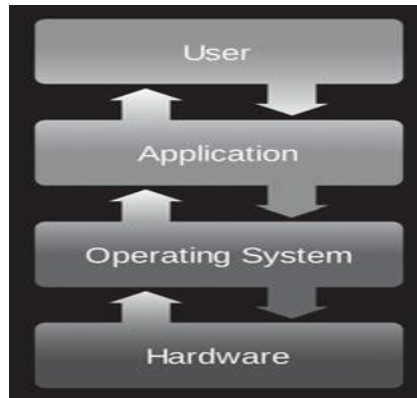
- **Secondary memory** is nonvolatile, and therefore provides long-term storage of programs and data. This kind of storage, for example, can be magnetic (hard drive), optical (CD or DVD), or nonvolatile flash memory (such as in a USB drive).
- **Input/output devices** include anything that allows for input (such as the mouse and keyboard) or output (such as a monitor or printer).
- **Buses** transfer data between components within a computer system, such as between the CPU and main memory.



Fundamental Hardware Components

1.3.4 Operating Systems—Bridging Software and Hardware

- An **operating system** is software that has the job of managing and interacting with the hardware resources of a computer. Because an operating system is intrinsic to the operation of a computer, it is referred to as **system software**.
- An operating system acts as the “middle man” between the hardware and executing application programs.



1.3.5 Limits of Integrated Circuits Technology: Moore's Law

Moore's Law states that the number of transistors that can be placed on a single silicon chip doubles roughly every two years.

1.4 Computer Software

Computer software is a set of program instructions, including related data and documentation that can be executed by computer. They can be classified in two,

- System Software
- Application Software

System software is intrinsic to a computer system; application software fulfils users' needs, such as a photo-editing program.

1.4.1 Syntax, Semantics, and Program Translation

Syntax

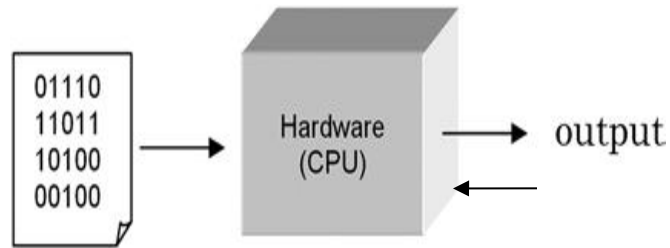
The **syntax** of a language is a set of characters and the acceptable arrangements (sequences) of those characters.

Semantics

The **semantics** of a language is the meaning associated with each syntactically correct sequence of characters.

Program Translation

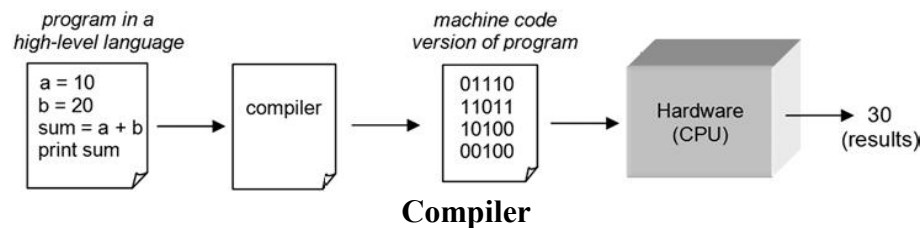
- A central processing unit (CPU) is designed to interpret and execute a specific set of instructions represented in binary form (i.e., 1s and 0s) called **machine code**. Only programs in machine code can be executed by a CPU.



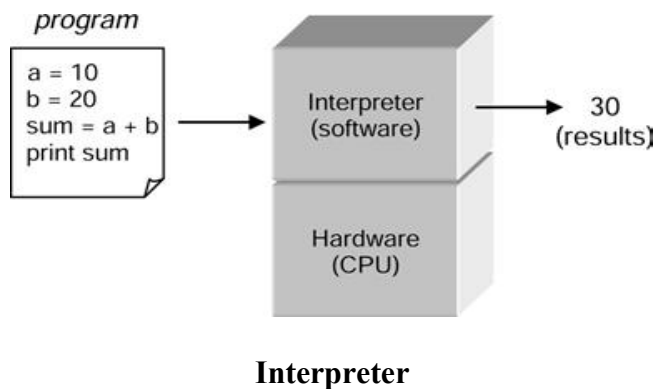
- Writing programs at this “low level” is tedious and error-prone. Therefore, most programs are written in a “high-level” programming language such as Python. Since the instructions of such programs are not in machine code that a CPU can execute, a translator program must be used. There are two fundamental types of translators.

- Compiler
- Interpreter

A **compiler** translates programs directly into machine code to be executed by the CPU.



An **interpreter**, which executes program instructions in place of (“running on top of”) the CPU. An interpreter can immediately execute instructions as they are entered. This is referred to as interactive mode.



Program Debugging: Syntax Errors vs. Semantic Errors

- **Program debugging** is the process of finding and correcting errors (“**bugs**”) in a computer program. Programming errors are inevitable during program development.
- **Syntax errors** are caused by invalid syntax (for example, entering `prnt` instead of `print`).

- **Semantic errors** (generally called **logic errors**) are errors in program logic. Such errors cannot be automatically detected, since translators cannot understand the intent of a given computation. For example, if a program computed the average of three numbers as follows,

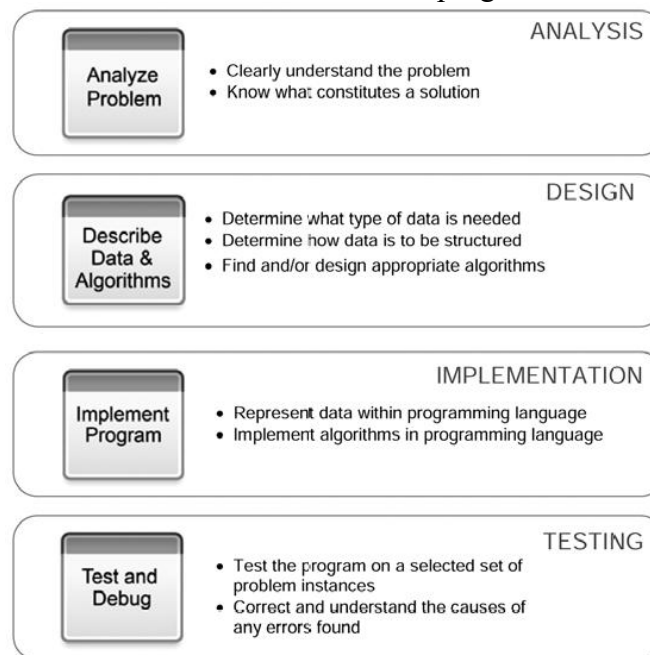
$$(\text{num1 } 1 \text{ num2 } 1 \text{ num3}) / 2.0$$

1.4.3 Procedural vs. Object-Oriented Programming

Programming languages fall into a number of programming paradigms. The two major programming paradigms in use today are procedural (imperative) programming and object-oriented programming. Each provides a different way of thinking about computation. While most programming languages only support one paradigm, Python supports both procedural and object-oriented programming.

1.5 COMPUTATIONAL PROBLEM SOLVING

It is a process, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested.



Process of Computational Problem Solving

1.5.1 Problem Analysis

Understanding the Problem

Once a problem is clearly understood, the fundamental computational issues for solving it can be determined. For each of the problems discussed earlier, the representation is straightforward.

For example the Man, Cabbage, Goat, Wolf (MCGW) problem, a brute-force algorithmic approach of trying all possible solutions works very well, since there are a small number of actions that can be taken at each step, and only a relatively small number of steps for reaching a solution.

Knowing What Constitutes a Solution

Besides clearly understanding a computational problem, one must know what constitutes a solution. For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a program may be stated as finding,

- A solution
- An approximate solution
- A best solution
- All solutions

For example the MCGW problem, there are an infinite number of solutions since the man could pointlessly row back and forth across the river an arbitrary number of times.

1.5.2 Program Design

Describing the Data Needed

For the Man, Cabbage, Goat, Wolf problem, a list can be used to represent the correct location (east and west) of the man, cabbage, goat, and wolf as discussed earlier, reproduced below,

Man	Cabbage	Goat	Wolf
[W,	E,	W,	E]

Describing the Needed Algorithms

When solving a computational problem, either suitable existing algorithms may be found or new algorithms must be developed. For the MCGW problem, there are standard search algorithms that can be used.

1.5.3 Program Implementation

Since we are programming in Python, the implementation needs to be expressed in a syntactically correct and appropriate way, using the instructions and features available in Python.

1.5.4 Program Testing

Software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the program is complete, and when the program needs to be updated.

1.6 The Python Programming Language

1.6.1 About Python

Guido van Rossum is the creator of the Python programming language, first released in the early 1990s. Its name comes from a 1970s British comedy sketch television show called Monty Python's Flying Circus. The development environment IDLE provided with Python comes from the name of a member of the comic group.

Python has a simple syntax. Python programs are clear and easy to read. At the same time, Python provides powerful programming features, and is widely used. Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA. Python is well supported and freely available at www.python.org.

1.6.2 The IDLE Python Development Environment

IDLE is an **integrated development environment (IDE)**. An IDE is a bundled set of software tools for program development. This typically includes an **editor** for creating and modifying programs, a **translator** for executing programs, and a **program debugger**. A debugger provides a means of taking control of the execution of a program to aid in finding program errors. Python is most commonly translated by use of an interpreter. The window that provides this interaction is referred to as the **Python shell**. Interacting with the shell is much like using a calculator.

1.6.3 The Python Standard Library

The **Python Standard Library** is a collection of built-in modules, each providing specific functionality beyond what is included in the “core” part of Python.

1.6.4 A Bit of Python

Variables

One of the most fundamental concepts in programming is that of a variable. (Variables are discussed in detail in Chapters 2.) A simple description of a variable is “a name that is assigned to a value,”

EX:

$n = 5$ (variable n is assigned the value 5)

Some Basic Arithmetic Operators

The common arithmetic operators in Python are

- +(addition)
- -(subtraction)
- *(multiplication)
- /(division)

➤ ** (exponentiation)

EX:

10 + 20, 25 - 15, 5 * 10, 20 / 10, 2 ** 4 (2 to the 4th power)

Basic Input and Output

Input

The programs that we will write request and get information from the user.

EX:

```
nam=input('What is your name?: ')
```

Characters within quotes are called strings. This particular use of a string, for requesting input from the user, is called a prompt. The input function displays the string on the screen to prompt the user for input.

EX:

What is your name? : Krishna

Output

The print function is used to display information on the screen in Python. This may be used to display a message.

EX:

```
>>> print('Welcome to My First Program!')
```

II - CHAPTER: DATA AND EXPRESSIONS

2.1 Literals

2.1.1 What Is a Literal?

A **literal** is a sequence of one or more characters that stands for itself.

2.1.2 Numeric Literals

A **numeric literal** is a literal containing only the digits 0–9, a sign character (1 or 2) and a possible decimal point. If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or “float” (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10). Commas are never used in numeric literals.

EX:

5+9

Limits of Range in Floating-Point Representation

There is no limit to the size of an integer that can be represented in Python. Floating-point values, however, have both a limited range and a limited precision. Python uses a double-precision standard format (IEEE 754) providing a range of 10^{-308} to 10^{308} with 16 to 17 digits of precision. To denote such a range of values, floating-points can be represented in scientific notation.

EX:

9.0045602e + 5	(9.0045602 * 10^5 , 8 digits of precision)
1.006249505236801e8	(1.006249505236801 * 10^8 , 16 digits of precision)
4.239e216	(4.239 * 10^{-16} , 4 digits of precision)

Arithmetic overflow

A condition that occurs when a calculated result is too large in magnitude (size) to be represented.

EX:

```
>>> 1.5e200 * 2.0e210
```

Arithmetic underflow

A condition that occurs when a calculated result is too small in magnitude to be represented.

EX:

```
>>> 1.0e2300 / 1.0e100
```

Built-in format Function

The built-in **format** function can be used to produce a numeric string of a given floating-point value rounded to a specific number of decimal places.

EX:

```
>>>12/5    2.4                                >>>format(12/5, '.2f')    2.40
>>>5/7      0.7142857142857143                >>>format(5/7, '.2f')    0.71
```

2.1.3 String Literals

➤ **String literals**, or “**strings**,” represent a sequence of characters. In Python, string literals may be delimited (surrounded) by a matching pair of either single (') or double (") quotes. Strings must be contained all on one line.

EX:

```
>>> print('Welcome to Python!')
```

➤ String may contain zero or more characters, including letters, digits, special characters, and blanks. A string consisting of only a pair of matching quotes (with nothing in between) is called the **empty string**, which is different from a string containing only blank characters.

EX:

```
>>>print('') "Empty string"
>>>print(' ') "Blank string"
```

The Representation of Character Values

Unicode is capable of representing over 4 billion different characters, enough to represent the characters of all languages, past and present. Python's (default) character encoding uses **UTF-8**, an eight-bit encoding that is part of the Unicode standard.

Space	00100000	32	A	01000001	65
!	00100001	33	B	01000010	66
"	00100010	34	C	01000011	67
#	00100011	35	.		
.			.		
.			Z	01011010	90
0	00110000	48	a	01100001	97
1	00110001	49	b	01100010	98
2	00110010	50	c	01100011	99
.			.		
.			.		
9	00111001	57	z	01111010	122

2.1.4 Control Characters

Control characters are special characters that are not displayed on the screen. Rather, they control the display of output (among other things). They are represented by a combination of characters called an escape print sequence. '\n', represents the newline control character, used to begin a new screen line.

EX:

```
('Hello\n Jennifer Smith')
```

2.1.5 String Formatting

The format function can be used to control how strings are displayed.

Syntax:

format (value, format_specifier)

Where value is the value to be displayed and format_specifier can contain a combination of formatting options.

EX:

format('Hello', '< 20')	→ 'Hello '	Left justified
format('Hello', '> 20')	→ ' Hello'	Right justified
format('Hello', '^20')	→ ' Hello '	Centre justified
format(' ', '30')	→ ' '	Blank space
format('!', '! < 30')	→ '.....!.....'	Printing character

2.1.6 Implicit and Explicit Line Joining

Sometimes a program line may be too long to fit in the Python-recommended maximum length of 79 characters.

Implicit Line Joining

There are certain delimiting characters that allow a logical program line to span more than on physical line. This includes matching parentheses, square brackets, curly braces, and triple quotes.

EX:

```
print('Name:', student_name, 'Address:', student_address,  
      'Number of Credits:', total_credits, 'GPA:', current_gpa)
```

Explicit Line Joining

Program lines may be explicitly joined by use of the backslash (\) character.

EX:

```
numsecs_1900_dob 5 ((year_birth 2 1900) * avg_numsecs_year) 1 \
((month_birth 2 1) * avg_numsecs_month) 1 \
(day_birth * numsecs_day)
```

2.2 Variables and Identifiers

2.2.1 What Is a Variable?

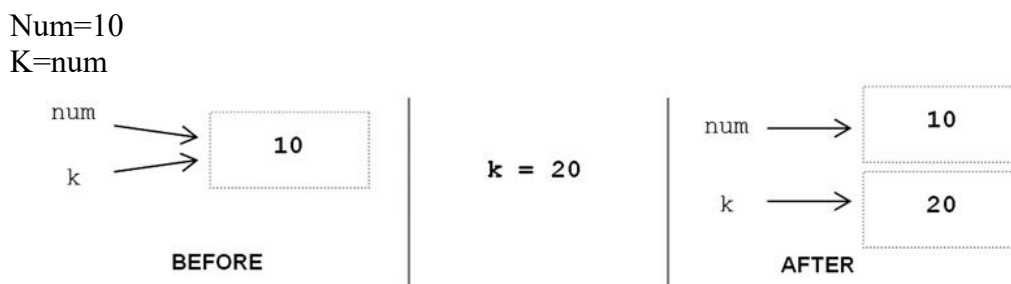
A **variable** is a name (identifier) that is associated with a value. A variable can be assigned different values during a program's execution. Variables are assigned values by use of the **assignment operator**, "=".

EX:

```
num =10, num =num+1
```

An **immutable value** is a value that cannot be changed.

EX:



2.2.2 Variable Assignment and Keyboard Input

The value can come from the user by use of the input function.

EX:

```
>>>name =input('What is your first name?')
What is your first name? Krishna
```

If the user hit return without entering any value, name would be assigned to the empty string ("). All input is returned by the input function as a string type. Python provides built-in **type conversion Functions**.

EX:

```
>>>int(input('Enter no'))
```

2.2.3 What Is an Identifier?

An **identifier** is a sequence of one or more characters used to provide a name for a given program element. Python is case sensitive. Identifiers may contain letters and digits, but cannot begin with a digit. The underscore character, `_`, is also allowed to aid in the readability of long identifier names. It should not be used as the first character, however, as identifiers beginning with an underscore have special meaning in Python. Spaces are not allowed as part of an identifier.

EX:

`StdName, add2, e_id`

2.2.4 Keywords and Other Predefined Identifiers in Python

A **keyword** is an identifier that has predefined meaning in a programming language. Therefore, keywords cannot be used as “regular” identifiers.

EX:

<code>and</code>	<code>as</code>	<code>assert</code>	<code>break</code>	<code>class</code>	<code>continue</code>	<code>def</code>
<code>del</code>	<code>elif</code>	<code>else</code>	<code>except</code>	<code>finally</code>	<code>for</code>	<code>from</code>
<code>global</code>	<code>if</code>	<code>import</code>	<code>in</code>	<code>is</code>	<code>lambda</code>	<code>nonlocal</code>
<code>not</code>	<code>or</code>	<code>pass</code>	<code>raise</code>	<code>return</code>	<code>try</code>	<code>while</code>
<code>with</code>	<code>yield</code>	<code>false</code>	<code>none</code>	<code>true</code>		

2.3 Operators

2.3.1 What Is an Operator?

An **operator** is a symbol that represents an operation that may be performed on one or more operands. An **operand** is a value that a given operator is applied. A **unary operator** operates on only one operand. A **binary operator** operates on two operands, as with the addition operator.

2.3.2 Arithmetic Operators

Python provides the arithmetic operators are

Arithmetic Operators		Example	Result
<code>-x</code>	negation	<code>-10</code>	<code>-10</code>
<code>x + y</code>	addition	<code>10 + 25</code>	<code>35</code>
<code>x - y</code>	subtraction	<code>10 - 25</code>	<code>-15</code>
<code>x * y</code>	multiplication	<code>10 * 5</code>	<code>50</code>
<code>x / y</code>	division	<code>25 / 10</code>	<code>2.5</code>
<code>x // y</code>	truncating div	<code>25 // 10</code>	<code>2</code>
		<code>25 // 10.0</code>	<code>2.0</code>
<code>x % y</code>	modulus	<code>25 % 10</code>	<code>5</code>
<code>x ** y</code>	exponentiation	<code>10 ** 2</code>	<code>100</code>

The **division operator**, `/`, produces “true division” regardless of its operand types. The **truncating division operator**, `//`, produces either an integer or float truncated result based on

the type of operands applied to. The **modulus operator** (%) gives the remainder of the division of its operands.

2.4 Expressions and Data Types

2.4.1 What Is an Expression?

An **expression** is a combination of symbols (or single symbol) that evaluates to a value. A **sub expression** is any expression that is part of a larger expression.

EX:

$$4 + (3 * k)$$

2.4.2 Operator Precedence

Operator precedence is the relative order that operators are applied in the evaluation of expressions, defined by a given **operator precedence table**.

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right

EX:

$$\triangleright 10 + 20 < 20 + 30 \rightarrow 30 < 50 \rightarrow \text{True}$$

$$\triangleright 10 < 20 \text{ and } 30 < 20 \text{ or } 30 < 40 \rightarrow \text{True and False or True} \\ \rightarrow \text{False or True} \rightarrow \text{True}$$

2.4.3 Operator Associativity

Operator associativity is the order that operators are applied when having the same level of precedence, specific to each operator.

EX:

$$(2+3) + 4 \rightarrow 9$$

2.4.4 What Is a Data Type?

A **data type** is a set of values, and a set of operators that may be applied to those values. Integers, floats, and strings are part of a set of predefined data types in Python called the **built-in types**. In **static typing**, a variable is declared as a certain type before it is used, and can only be assigned values of that type. Python, however, uses dynamic typing. In **dynamic typing**, the data type of a variable depends only on the type of value that the variable is currently holding.

2.4.5 Mixed-Type Expressions

A **mixed-type expression** is an expression containing operands of different type. Operands of mixed-type expressions therefore must be converted to a common type. Values can be converted in one of two ways—by implicit (automatic) conversion, called coercion, or by explicit type conversion.

Coercion vs. Type Conversion

- **Coercion** is automatically performed on mixed-type expressions only if the operands can be safely converted, that is, if no loss of information will result.
- **Type conversion** is the explicit conversion of operands to a specific type. Type conversion can be applied even if loss of information results.

UNIT - II

UNIT - II

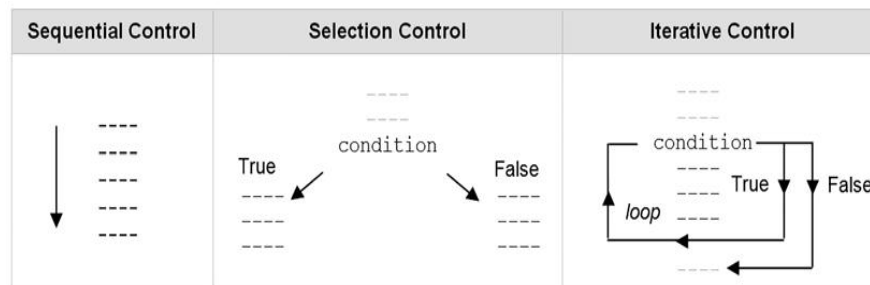
S.No	Topic	Page No.
1.	What is a control structure?	
2.	Explain boolean expressions	
3.	Explain selection control?	
4.	Explain iterative control?	
5.	Define lists in python.	
6.	Define list operations.	
7.	Define list traversal.	
8.	Explain lists in python.	
9.	Explain iterating over lists (sequences) in python	

III - CHAPTER: CONTROL STRUCTURES

3.1 What Is a Control Structure?

Control is the order that instructions are executed in a program. A **control statement** is a statement that determines the control flow of a set of instructions. A **control structure** is a set of instructions and the control statements controlling their execution. There are three fundamental forms of control that programming languages provide,

- Sequential control
 - Selection control
 - Iterative control
- **Sequential control** is an implicit form of control in which instructions are executed in the order that they are written.
- **Selection control** is provided by a control statement that selectively executes instructions
- **Iterative control** is provided by an iterative control statement that repeatedly executes instructions.



3.2 Boolean Expressions (Conditions)

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python. A **Boolean expression** is an expression that evaluates to a Boolean value.

3.2.1 Relational Operators

The relational operators in Python perform the usual comparison operations. Relational expressions are a type of Boolean expression, since they evaluate to a Boolean result.

Relational Operators	Example	Result
<code>==</code> equal	<code>10 == 10</code>	True
<code>!=</code> not equal	<code>10 != 10</code>	False
<code><</code> less than	<code>10 < 20</code>	True
<code>></code> greater than	<code>'Alan' > 'Brenda'</code>	False
<code><=</code> less than or equal to	<code>10 <= 10</code>	True
<code>>=</code> greater than or equal to	<code>'A' >= 'D'</code>	False

3.2.2 Membership Operators

Python provides a convenient pair of **membership operators**. These operators can be used to easily determine if a particular value occurs within a specified list of values.

Membership Operators	Examples	Result
<code>in</code>	<code>10 in (10, 20, 30)</code>	True
	<code>red in ('red', 'green', 'blue')</code>	True
<code>not in</code>	<code>10 not in (10, 20, 30)</code>	False

3.2.3 Boolean Operators

George Boole, in the mid-1800s, developed what we now call Boolean algebra. Boolean algebra contains a set of **Boolean** (logical) **operators**, denoted by `and`, `or`, and `not` in Python. these logical operators can be used to construct arbitrarily complex Boolean expressions.

x	y	x and y	x or y	not x
False	False	False	False	True
True	False	False	True	False
False	True	False	True	
True	True	True	True	

EX:

```
>>>(n>10) and (n<15)
>>>(n==1)or(n==0)
>>>not(n=0)
```

3.2.4 Operator Precedence and Boolean Expressions

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right
<, >, <=, >=, !=, == (relational operators)	left-to-right
not	left-to-right
and	left-to-right
or	left-to-right

EX:

$(10 < 20 \text{ and } 30 < 20) \text{ or } (30 < 40)$

3.2.5 Short-Circuit (Lazy) Evaluation

In **short-circuit (lazy) evaluation**, the second operand of Boolean operators and or is not evaluated if the value of the Boolean expression can be determined from the first operand alone.

EX:

if $n \neq 0$:
if $1/n < \text{tolerance}$

3.2.6 Logically Equivalent Boolean Expressions

In numerical algebra, there are arithmetically equivalent expressions of different form.

Logically Equivalent Boolean Expressions		
$x < y$	is equivalent to	$\text{not}(x \geq y)$
$x \leq y$	is equivalent to	$\text{not}(x > y)$
$x == y$	is equivalent to	$\text{not}(x \neq y)$
$x \neq y$	is equivalent to	$\text{not}(x == y)$
$\text{not}(x \text{ and } y)$	is equivalent to	$(\text{not } x) \text{ or } (\text{not } y)$
$\text{not}(x \text{ or } y)$	is equivalent to	$(\text{not } x) \text{ and } (\text{not } y)$

3.3 Selection Control

A **selection control statement** is a control statement providing selective execution of instructions. A selection control structure is a given set of instructions and the selection control statement(s) controlling their execution.

3.3.1 If Statement

An **if statement** is a selection control statement based on the value of a given Boolean expression. A statement that contains other statements, such as the if statement, is called a **compound statement**.

if statement	Example use
<pre>if condition: statements else: statements</pre>	<pre>if grade >= 70: print('passing grade') else: print('failing grade') if grade == 100: print('perfect score!')</pre>

EX:

```
1 # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3 # Display program welcome
4 print('This program will convert temperatures (Fahrenheit/Celsius)')
5 print('Enter (F) to convert Fahrenheit to Celsius')
6 print('Enter (C) to convert Celsius to Fahrenheit')
7
8 # Get temperature to convert
9 which = input('Enter selection: ')
10 temp = int(input('Enter temperature to convert: '))
11
12 # Determine temperature conversion needed and display results
13 if which == 'F':
14     converted_temp = (temp - 32) * 5/9
15     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
16 else:
17     converted_temp = (9/5 * temp) + 32
18     print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')
```

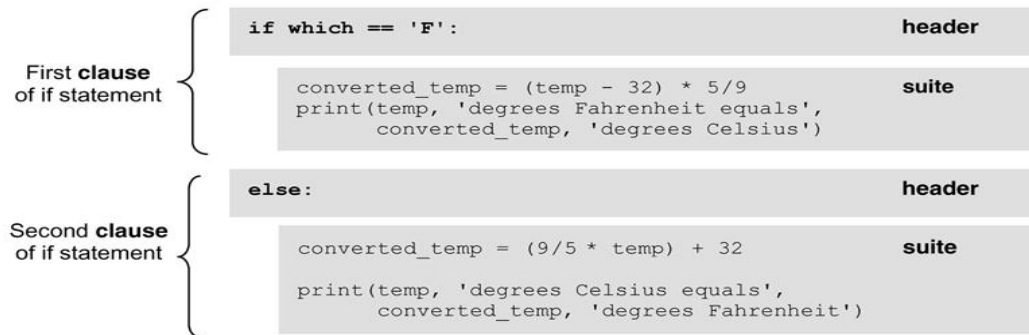
3.3.2 Indentation in Python

➤ In Python, however, indentation is used to associate and group statements. A **header** in Python is a specific keyword followed by a colon. Headers that are part of the same compound statement must be indented the same amount—otherwise, a syntax error will result.

➤ The set of statements following a header in Python is called a **suite** (commonly called a **block**). The statements of a given suite must all be indented the same amount. A header and its

associated suite are together referred to as a **clause**. A compound statement in Python may consist of one or more clauses.

EX:



Valid indentation		Invalid indentation	
(a) <pre>if condition: statement statement else: statement statement</pre>	(b) <pre>if condition: statement statement else: statement statement</pre>	(c) <pre>if condition: statement statement else: statement statement</pre>	(d) <pre>if condition: statement statement else: statement statement</pre>

3.3.3 Multi-Way Selection

Nested if Statements

If statements can be nested in Python, resulting in **multi-way selection**.

Nested if statements	Example use
<pre> if condition: statements else: if condition: statements else: if condition: statements etc. </pre>	<pre> if grade >= 90: print('Grade of A') else: if grade >= 80: print('Grade of B') else: if grade >= 70: print('Grade of C') else: if grade >= 60: print('Grade of D') else: print('Grade of F') </pre>

EX:

```

credits 5 45
if credits .5 90:
    print('Senior')
else:
    if credits .5 60:
        print('Junior')
    else:
        if credits .5 30:
            print('Sophomore')
        else:
            if credits .5 1:
                print('Freshman')
            else:
                print('* No Earned Credits *')

```

The elif Header in Python

Python has another header called elif (“else-if”) that provides multi-way selection in a single if statement. A final else clause may be used for “catch-all” situations.

EX:

```

if grade >= 90:
    print('Grade of A')
elif grade >= 80:
    print('Grade of B')
elif grade >= 70:
    print('Grade of C')
elif grade >= 60:
    print('Grade of D')
else:
    print('Grade of F')

```

3.4 Iterative Control

An **iterative control statement** is a control statement providing the repeated execution of a set of instructions. An iterative control structure is a set of instructions and the iterative control Statement controlling their execution.

3.4.1 While Statement

A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression (condition). As long as the condition of a while statement is true, the statements within the loop are (re)executed. Once the condition becomes false, the iteration terminates and control continues with the first statement after the while loop.

while statement	Example use
<pre> while condition: suite </pre>	<pre> sum = 0 current = 1 n = int(input('Enter value: ')) while current <= n: sum = sum + current current = current + 1 </pre>

3.4.2 Input Error Checking

The while statement is well suited for input error checking in a program.

```

1 # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3 # Display program welcome
4 print('This program will convert temperatures (Fahrenheit/Celsius)')
5 print('Enter (F) to convert Fahrenheit to Celsius')
6 print('Enter (C) to convert Celsius to Fahrenheit')
7
8 # Get temperature to convert
9 which = input('Enter selection: ')
10
11 while which != 'F' and which != 'C':
12     which = input("Please enter 'F' or 'C': ")
13
14 temp = int(input('Enter temperature to convert: '))
15
16 # Determine temperature conversion needed and display results
17 if which == 'F':
18     converted_temp = format((temp - 32) * 5/9, '.1f')
19     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
20 else:
21     converted_temp = format((9/5 * temp) + 32, '.1f')
22     print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')

```

3.4.3 Infinite loops

An **infinite loop** is an iterative control structure that never terminates (or eventually terminates with a system error). Infinite loops are generally the result of programming errors. Such infinite loops can cause a program to “hang,” that is, to be unresponsive to the user. In such cases, the program must be terminated by use of some special keyboard input (such as ctrl-C) to interrupt the execution.

EX:

```

# add up first n integers
sum = 0
current = 1

n = int(input('Enter value: '))

while current <= n:
    sum = sum + current

```

3.4.4 Definite vs. Indefinite Loops

A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. Although it is not known what the value of n will be until the input statement is executed, its value is known by the time the while loop is reached. Thus, it will execute “n times.”

EX:

```

sum = 0
current = 1
n = input('Enter value: ')
while current <= n:

```

```
sum = sum + current
current = current + 1
```

An **indefinite loop** is a program loop in which the number of times that the loop will iterate cannot be determined before the loop is executed.

EX:

```
which = input("Enter selection: ")
while which != 'F' and which != 'C':
    which = input("Please enter 'F' or 'C': ")
```

Boolean Flags and Indefinite Loops

A single Boolean variable used as the condition of a given control statement is called a **Boolean flag**.

```
1 # Oil Change Notification Program
2
3 # display program welcome
4 print('This program will determine if your car is in need of an oil change')
5
6 # init
7 miles_between_oil_change = 7500 # num miles between oil changes
8 miles_warning = 500 # how soon to warn of needed oil change
9 valid_entries = False
10
11 # get mileage of last oil change and current mileage and display
12 while not valid_entries:
13     mileage_last_oilchange = int(input('Enter mileage of last oil change: '))
14     current_mileage = int(input('Enter current mileage: '))
15
16     if current_mileage < mileage_last_oilchange:
17         print('Invalid entry - current mileage entered is less than')
18         print('mileage entered of last oil change')
19     else:
20         miles_traveled = current_mileage - mileage_last_oilchange
21         valid_entries = True
22
23 if miles_traveled >= miles_oil_change:
24     print('You are due for an oil change')
25 elif miles_traveled >= miles_oil_change - miles_warning:
26     print('You will soon be due for an oil change')
27 else:
28     print('You are not in immediate need of an oil change')
```

IV - CHAPTER:LIST

4.1 List Structures

4.1.1 What Is a List?

A **list** is a linear data structure, meaning that its elements have a linear ordering. That is, there is a first element, a second element, and so on. In which each item in the list is identified by its index in which each item in the list is identified by its index value.

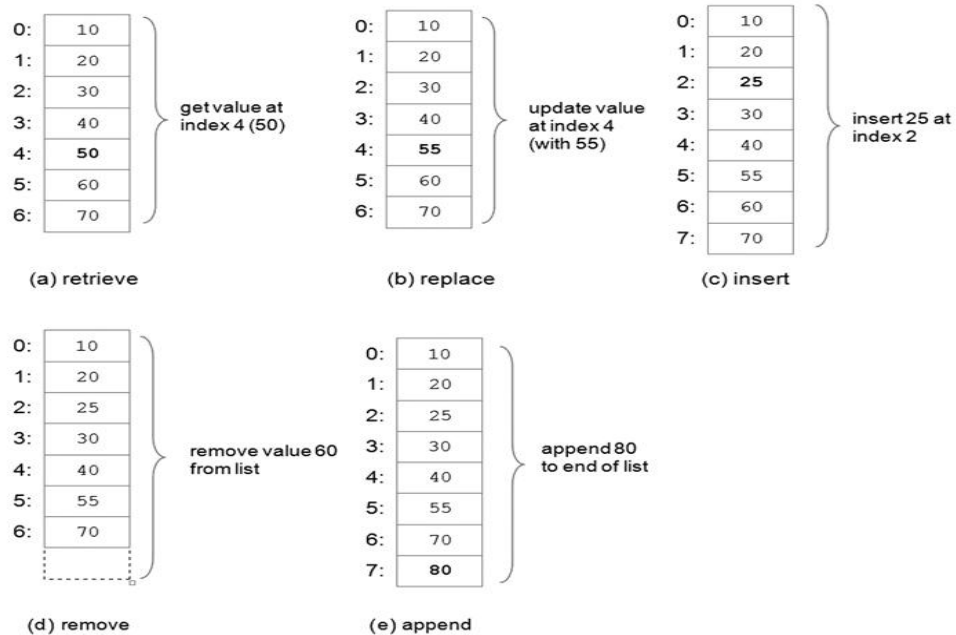
EX:

0:	68.8
1:	70.2
2:	67.2
3:	71.8
4:	73.2
5:	75.6
6:	74.0

4.1.2 Common List Operations

Operations commonly performed on lists include retrieve, update, insert, delete (remove) and append.

EX:



4.1.3 List Traversal

A **list traversal** is a means of accessing, one-by-one, the elements of a list.

EX:

Adding up all values in the list					Searching for the value 50 in the list				
				sum					Find
0:	10	←	+10	10	0:	10	←	50?	no
1:	20	←	+20	30	1:	20	←	50?	no
2:	30	←	+30	60	2:	30	←	50?	no
3:	40	←	+40	100	3:	40	←	50?	no
4:	50	←	+50	150	4:	50	←	50?	yes
5:	60	←	+60	210	5:	60			
6:	70	←	+70	280	6:	70			

4.2 Lists (Sequences) in Python

4.2.1 Python List Type

➤ A **list** in Python is a mutable, linear data structure of variable length, allowing mixed-type elements. Mutable means that the contents of the list may be altered. Lists in Python use zero based indexing. Thus, all lists have index values 0 ... n-1, where n is the number of elements in the list. Lists are denoted by a comma-separated list of elements within square brackets.

- An **empty list** is denoted by an empty pair of square brackets, []. Elements of a list are accessed by using an index value within square brackets.

EX:

```
lst = [1, 2, 3]
lst[0] → 1 access of first element
lst[1] → 2 access of second element
lst[2] → 3 access of third element
```

Operation	fruit = ['banana', 'apple', 'cherry']	
Replace	fruit[2] = 'coconut'	['banana', 'apple', 'coconut']
Delete	del fruit[1]	['banana', 'cherry']
Insert	fruit.insert(2, 'pear')	['banana', 'apple', 'pear', 'cherry']
Append	fruit.append('peach')	['banana', 'apple', 'cherry', 'peach']
Sort	fruit.sort()	['apple', 'banana', 'cherry']
Reverse	fruit.reverse()	['cherry', 'banana', 'apple']

4.2.2 Tuples

A **tuple** is an immutable linear data structure. Thus, in contrast to lists, once a tuple is defined, it cannot be altered. Otherwise, tuples and lists are essentially the same. To distinguish tuples from lists, tuples are denoted by parentheses instead of square brackets.

EX:

student 5 ('John Smith', 48, 'Computer Science', 3.42)

Another difference between tuples and lists is that tuples of one element must include a comma following the element. Otherwise, the parenthesized element will not be made into a tuple.

An **empty tuple** is represented by a set of empty parentheses, (). (We shall later see the usefulness of the empty tuple.) The elements of tuples are accessed the same as lists, with square brackets. Thus, delete, update, insert, and append operations are not defined on tuples.

4.2.3 Sequences

A **sequence** in Python is a linearly ordered set of elements accessed by an index number. Lists, tuples, and strings are all sequences. Strings, like tuples, are immutable ; therefore, they cannot be altered.

Operation		String s = 'hello' w = 'l'	Tuple s = (1,2,3,4) w = (5,6)	List s = [1,2,3,4] w = [5,6]
Length	len(s)	5	4	4
Select	s[0]	'h'	1	1
Slice	s[1:4]	'ell'	(2, 3, 4)	[2, 3, 4]
	s[1:]	'ello'	(2, 3, 4)	[2, 3, 4]
Count	s.count('e')	1	0	0
	s.count(4)	error	1	1
Index	s.index('e')	1	--	--
	s.index(3)	--	2	2
Membership	'h' in s	True	False	False
Concatenation	s + w	'hello!'	(1, 2, 3, 4, 5, 6)	[1, 2, 3, 4, 5, 6]
Minimum Value	min(s)	'e'	1	1
Maximum Value	max(s)	'o'	4	4
Sum	sum(s)	error	10	10

4.2.4 Nested Lists

Lists and tuples can be nested within each other to construct arbitrarily complex data structures.

EX:

```
class_grades 5=[ [85, 91, 89], [78, 81, 86], [62, 75, 77], ...]
```

```
class_grades[0][0] → [85, 91, 89][0] → 85
```

4.3 Iterating Over Lists (Sequences) in Python

4.3.1 For Loops

A **for statement** is an iterative control statement that iterates once for each element in a specified sequence of elements. Thus, for loops are used to construct definite loops.

for statement	Example use
<pre>for k in sequence: suite</pre>	<pre>nums = [10, 20, 30, 40, 50, 60] for k in nums: print(k)</pre>

Variable k is referred to as a **loop variable**.

EX:

```
for ch in 'Hello':
    print(ch)
```

4.3.2 The Built-in range Function

Python provides a built-in **range function** that can be used for generating a sequence of integers that a for loop can iterate over. The values in the generated sequence include the starting value, up to but not including the ending value.

EX:

```
sum = 0
for k in range(1, 11):
    sum = sum + k
```

This for loop adds up the integer values 1–10. By default, the range function generates a sequence of consecutive integers. A “step” value can be provided, however. Range (0, 11, 2) produces the sequence [0, 2, 4, 6, 8, 10], with a step value of 2. A sequence can also be generated “backwards” when given a negative step value. For example, range (10, 0, -2) produces the sequence [10, 8, 6, 4, 2, 0].

4.3.3 Iterating Over List Elements vs. List Index Values

An **index variable** is a variable whose changing value is used to access elements of an indexed data structure. However, there are times when the loop variable must iterate over the index values of a list instead.

Loop variable iterating over the elements of a sequence	Loop variable iterating over the index values of a sequence
<pre>nums = [10, 20, 30, 40, 50, 60] for k in nums: sum = sum + k</pre>	<pre>nums = [10, 20, 30, 40, 50, 60] for k in range(len(nums)): sum = sum + nums[k]</pre>

4.3.4 While Loops and Lists (Sequences)

For situations in which a sequence is to be traversed while a given condition is true, a while loop is the appropriate control structure to use.

EX:

```
k = 0
item_to_find = 40
found_item = False

while k < len(nums) and not found_item:
    if nums[k] == item_to_find:
        found_item = True
    else:
        k = k + 1

if found_item:
    print('item found')
else:
    print('item not found')
```

UNIT - IV

UNIT – III

S.No	Topic	Page No.
1.	What is a function routine?	
2.	Explain defining function in python?	
3.	Explain calling value-returning functions?	
4.	Explain calling non-value-returning functions?	
5.	Explain parameter passing?	
6.	Explain keyword arguments in python?	
7.	Define default arguments in python?	
8.	Explain variable scope?	

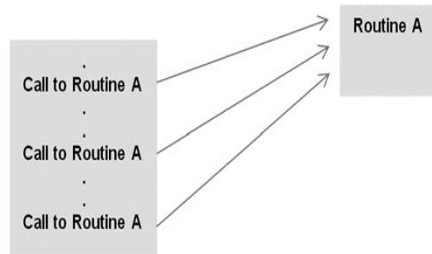
V- CHAPTER: FUNCTIONS

5.1 Program Routines

5.1.1 What Is a Function Routine?

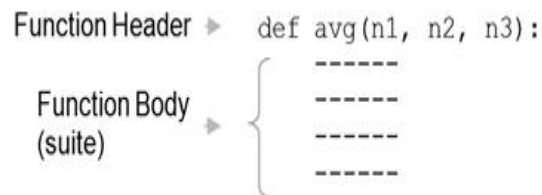
A program **routine** is a named group of instructions that accomplishes some task. A routine may be **invoked** (called) as many times as needed in a given program. A **function** is Python's version of a program routine.

EX:



5.1.2 Defining Functions

The elements of a function definition are



A function header starts with the keyword `def`, followed by an identifier (`avg`), which is the function's name. The function name is followed by a comma-separated (possibly empty) list of identifiers (`n1, n2, n3`) called **formal parameters**, or simply "parameters." Following the parameter list is a colon (`:`). Following the function header is the body of the function, a suite (program block) containing the function's instructions. As with all suites, the statements must be indented at the same level, relative to the function header.

Functions are generally defined at the top of a program. However, every function must be defined before it is called.

The number of items in a parameter list indicates the number of values that must be passed to the function, called **actual arguments** (or simply "arguments"), such as the variables `num1`, `num2`, and `num3`.

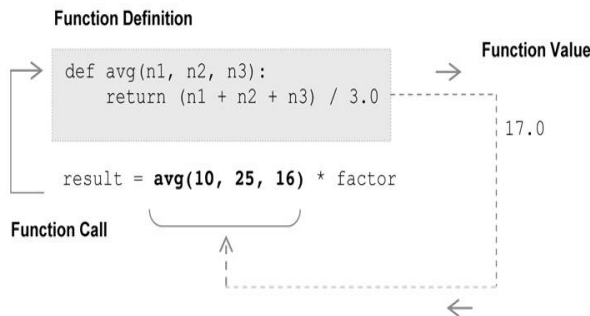
EX:

```
>>>num1=10
>>>num2=20
>>>num3=30
>>>avg(num1,num2,num3)
```

Value-Returning Functions

A **value-returning function** in Python is a program routine called for its return value, and is therefore similar to a mathematical function.

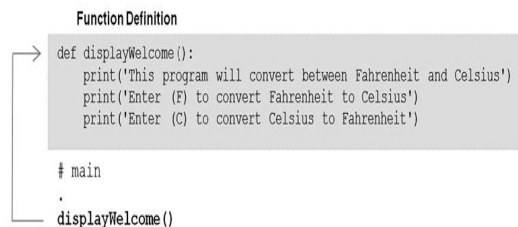
EX:



Non-Value-Returning Functions

A **non-value-returning function** is called not for a returned value, but for its side effects. A **side effect** is an action other than returning a function value, such as displaying output on the screen. There is a fundamental difference in the way that value-returning and non-value-returning functions are called. A call to a value-returning function is an expression, as for the call to function avg: result 5 **avg(10, 25, 16) * factor**. When non-value-returning functions are called, however, the function call is a statement. Since such functions do not have a return value, it is incorrect to use a call to a non-value-returning function as an expression.

EX:



5.2 More on Functions

5.2.1 Calling Value-Returning Functions

Function calls to value-returning functions can be used anywhere that a function's return value is appropriate.

EX:

function definition

```
def maxmin(num_list):  
    return (max(num_list), min(num_list))
```

function use

```
weekly_temps = [45, 30, 52, 58, 62, 48, 49]
```

5.2.2 Calling Non-Value-Returning Functions

Non-value-returning functions are called for their side effects, and not for a returned function value. Thus, such function calls are statements, and therefore can be used anywhere that an executable statement is allowed.

EX:

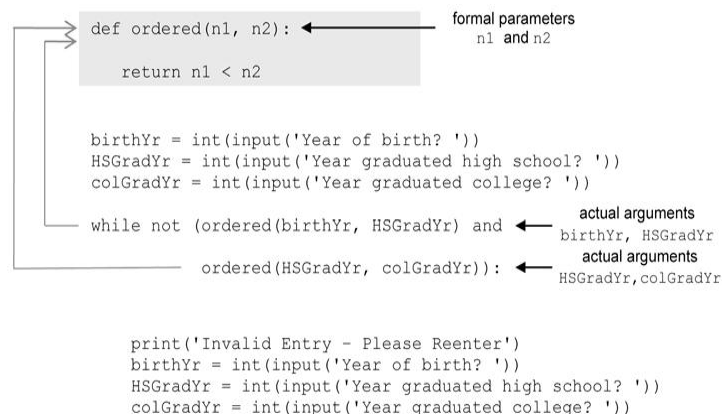
```
welcome_displayed = displayWelcome()
```

5.2.3 Parameter Passing

Actual Arguments vs. Formal Parameters

Parameter passing is the process of passing arguments to a function. As we have seen, actual arguments are the values passed to a function's formal parameters to be operated on.

EX:



Note that the correspondence of actual arguments and formal parameters is determined by the order of the arguments passed, and not their names.

EX:



```
def ordered(n1, n2):  
    return n1 < n2  
  
num1 = int(input('Enter your age: '))  
num2 = int(input('Enter your brother's age: '))  
  
if ordered(num1, num2):  
    print('He is your older brother')  
else:  
    if ordered(num2, num1):  
        print('He is your younger brother')  
    else:  
        print('Are you twins?')
```

Mutable vs. Immutable Arguments There is an issue related to parameter passing that we have yet to address. We know that when a function is called, the current values of the arguments passed become the initial values of their corresponding formal parameters.

5.2.4 Keyword Arguments in Python

A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list Python provides the option of calling any function by the use of keyword arguments.

A **keyword argument** is an argument that is specified by parameter name, rather than as a positional argument as shown below (note that keyword arguments, by convention, do not have a space before or after the equal sign).

5.2.5 Default Arguments in Python

A **default argument** is an argument that can be optionally provided in a given function call. When not provided, the corresponding parameter provides a default value.

5.2.6 Variable Scope

Local Scope and Local Variables

A **local variable** is a variable that is only accessible from within a given function. Such variables are said to have **local scope**.

EX:

```
def func1():
    n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)

>>> func2()
n in func2 before call to func1 = 20
n in func1 = 10
n in func2 after call to func1 = 20
```

Global Variables and Global Scope

A **global variable** is a variable that is defined outside of any function definition. Such variables are said to have **global scope**.

EX:

```
max = 100 # global variable

def func1(count):
    if count < max:
        .

def func2(count):
    for i in range(1, max):
        .
```

global variable max accessed

global variable max accessed

UNIT - IV

UNIT – IV

S.NO	Topic	Page No.
1.	What is an object?	
2.	Explain object references in python?	
3.	Define turtle graphics.	
4.	Explain creating a turtle graphics window?	
5.	Explain the “default” turtle?	
6.	Explain fundamental turtle attributes and behavior?	
7.	Explain additional turtle attributes?	
8.	Explain creating multiple turtles?	
9.	Explain module?	
10.	Explain module specification?	
11.	Explain top-down design?	
12.	What is a python module?	
13.	Explain modules and namespaces?	
14.	Explain importing modules?	
15.	Explain module loading and execution?	
16.	Explain local, global, and built-in namespaces in python?	
17.	Explain a programmer-defined stack module?	
18.	What is text file?	
19.	Explain opening the text file?	
20.	Explain reading the text file?	
21.	Explain writing the text file?	
22.	Explain string processing?	
23.	Define string processing.	
24.	Define string traversal.	
25.	Explain string operations?	
26.	Explain string methods?	
27.	Define exception handling.	
28.	What is exception?	
29.	Explain the propagation of raised exceptions?	
30.	Explain catching and handling exceptions?	
31.	Explain exception handling and user input?	
32.	Explain exception handling and file processing?	

VI-CHAPTER: OBJECT AND THEIR USE

6.1 Software Objects

6.1.1 What Is an Object?

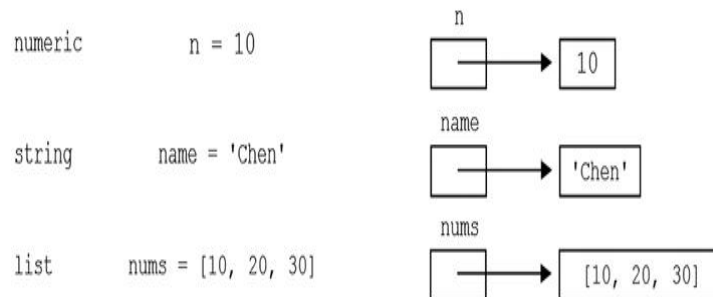
An **object** contains a set of attributes, stored in a set of **instance variables**, and a set of functions called **methods** that provide its behavior. an **object** contains a set of attributes, stored in a set of **instance variables** , and a set of functions called **methods** that provide its behavior. All list objects contain the same set of methods. The period is referred to as the dot operator, used to select a member of a given object.

6.1.2 Object References

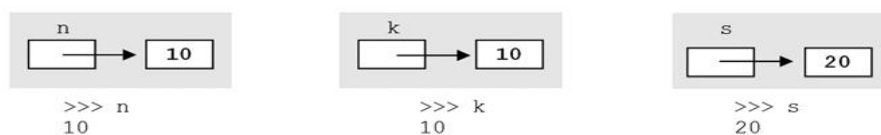
References in Python

In Python, objects are represented as a reference to an object in memory, A **reference** is a value that references, or “points to,” the location of another entity. Thus, when a new object in Python is created, two entities are stored—the object, and a variable holding a reference to the object. All access to the object is through the reference value.

EX:



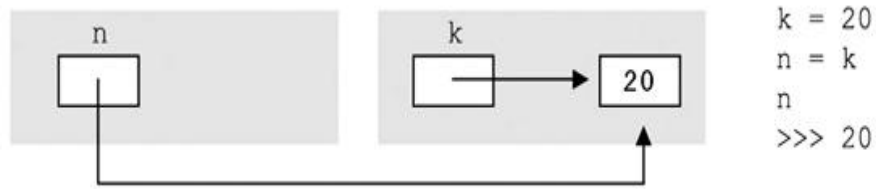
The value that a reference points to is called the **dereferenced value** .



We can get the reference value of a variable (that is, the location in which the corresponding object is stored) by use of **built-in function id**.

```
>>> id(n)      >>> id(k)      >>> id(s)
505498136      505498136      505498296
```

The Assignment of References



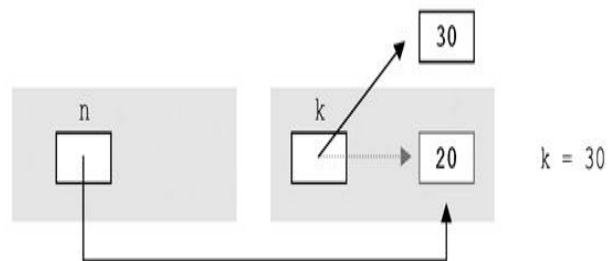
When variable `n` is assigned to `k`, it is the reference value of `k` that is assigned, not the dereferenced value 20. This can be determined by use of the built-in `id` function, as demonstrated below.

```
>>> id(k)
505498136
```

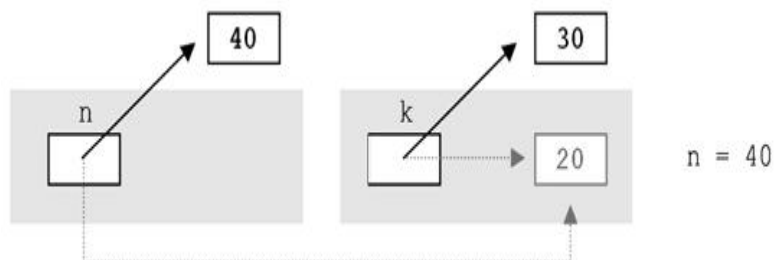
```
>>> id(k) == id(n)
True
```

```
>>> id(n)
505498136
```

```
>>> n is k
True
```



Memory Deallocation and Garbage Collection



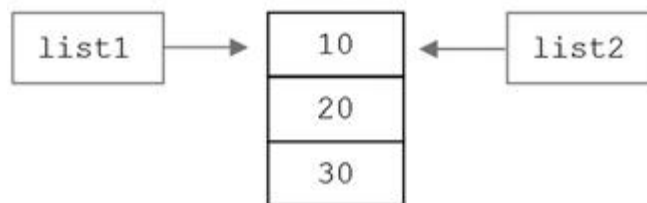
To **deallocate** a memory location means to change its status from “currently in use” to “available for reuse.” In Python, memory deallocation is automatically performed by a process called garbage collection. **Garbage collection** is a method of automatically determining which

locations in memory are no longer in use and deallocating them. The garbage collection process is ongoing during the execution of a Python program.

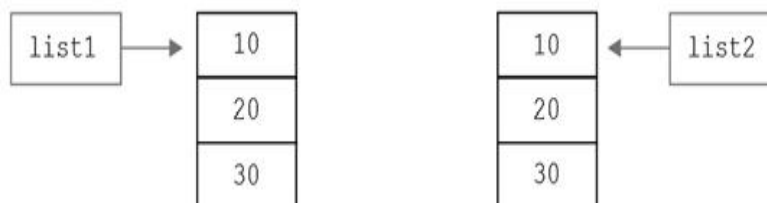
List Assignment and Copying

When a variable is assigned to another variable referencing a list, each variable ends up referring to the same instance of the list in memory.

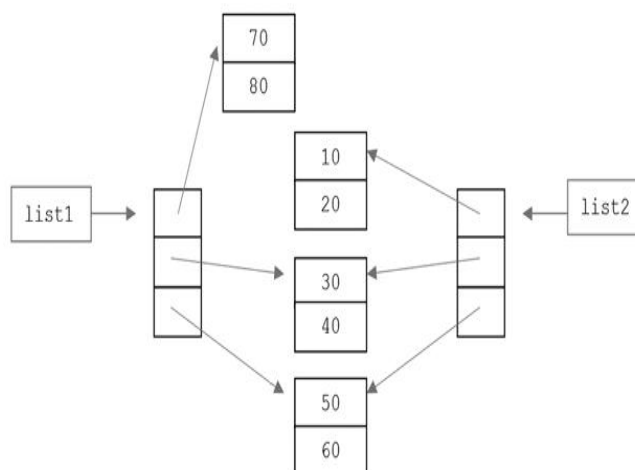
EX:



list() is referred to as a list constructor .



Although copies were made of the top-level list structures, the elements within each list were not copied. This is referred to as a **shallow copy**. Thus, if a top-level element of one list is reassigned, for example list1 [0] 5 [70, 80], the other list would remain unchanged,



A **deep copy** operation of a list (structure) makes a copy of the complete structure, including sublists. (Since immutable types cannot be altered, immutable parts of the structure may not be copied.) Such an operation can be performed with the deep copy method of the copy module.

6.2 Turtle Graphics

When a variable is assigned to another variable referencing a list, each variable ends up referring to the same instance of the list in memory.

6.2.1 Creating a Turtle Graphics Window

The first step in the use of turtle graphics is the creation of a turtle graphics window (a turtle screen).

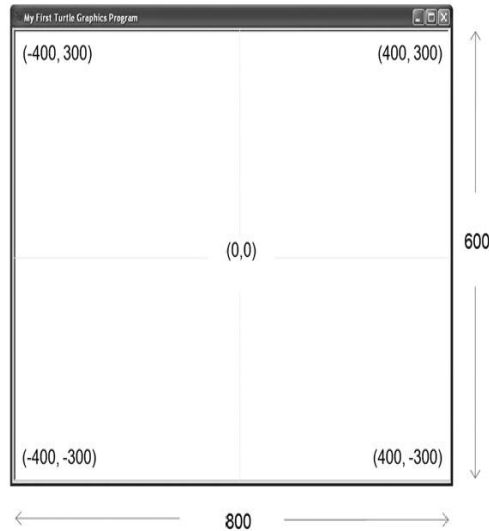
```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

Assuming that the `import turtle` form of import is used, each of the turtle graphics methods must be called in the form `turtle.methodname`. The first method called, `setup`, creates a graphics window of the specified size (in pixels). In this case, a window of size 800 pixels wide by 600 pixels high is created. The center point of the window is at coordinate (0,0). Thus, x-coordinate values to the right of the center point are positive values, and those to the left are negative values. Similarly, y-coordinate values above the center point are positive values, and those below are negative values. The top-left, top-right, bottom-left, and bottom-right coordinates for a window of size (800, 600).



The background color of the turtle window can be changed from the default white background color. This is done using method bgcolor.

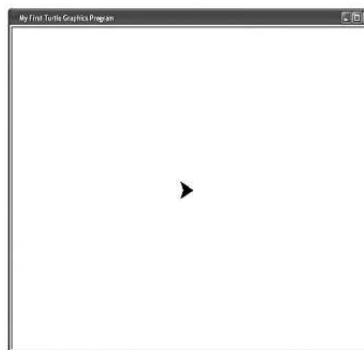
```
window ==turtle.Screen()  
window.bgcolor('blue')
```

6.2.2 The “Default” Turtle

A “turtle” is an entity in a turtle graphics window that can be controlled in various ways. Like the graphics window, turtles are objects. A “default” turtle is created when the setup method is called. The reference to this turtle object can be obtained by.

```
the_turtle = turtle.getturtle()
```

A call to getturtle returns the reference to the default turtle and causes it to appear on the screen. The initial position of all turtles is the center of the screen at coordinate (0,0).

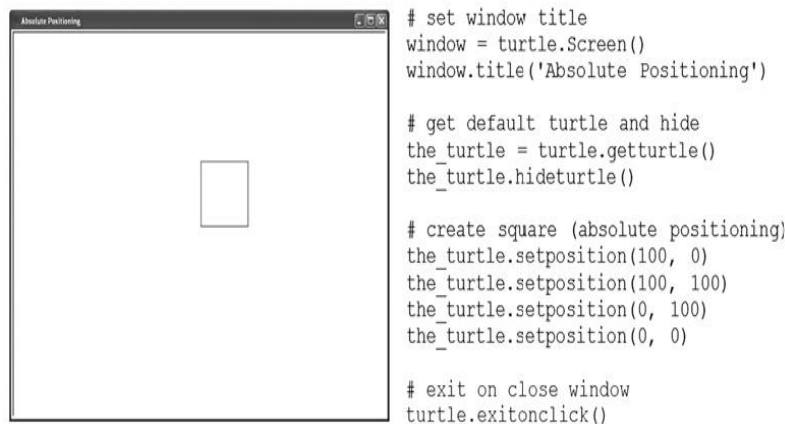


The default turtle shape is an arrowhead. (The size of the turtle shape was enlarged from its default size for clarity.) A turtle's shape can be set to basic geometric shapes, or even made from a provided image file.

6.2.3 Fundamental Turtle Attributes and Behavior

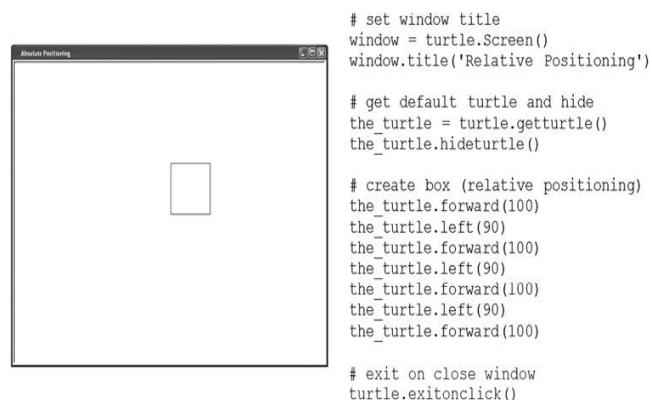
Absolute Positioning

Method `position` returns a turtle's current position. For newly created turtles, this returns the tuple `(0, 0)`. A turtle's position can be changed using absolute positioning by moving the turtle to a specific x,y coordinate location by use of method `setposition`.



Turtle Heading and Relative Positioning

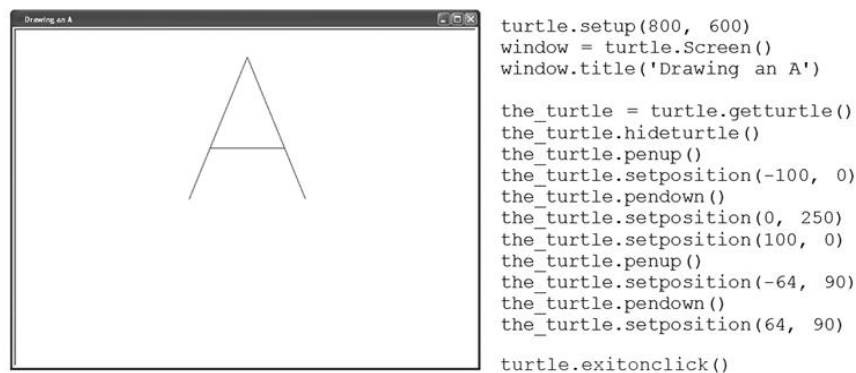
A turtle's position can also be changed through relative positioning. In this case, the location that turtle moves to is determined by its second fundamental attribute, its heading. A newly created turtle's heading is to the right, at 0 degrees. A turtle with heading 90 degrees moves up; with heading 180 degrees moves left; and with a heading 270 degrees moves down. A turtle's heading can be changed by turning the turtle a given number of degrees left, `left(90)`, or right, `right(90)`. The `forward` method moves a turtle in the direction that it is currently heading.



Methods left and right change a turtle's heading relative to its current heading. A turtle's heading can also be set to a specific heading by use of method setheading: the_turtle.setheading(90). In addition, method heading can be used to determine a turtle's current heading.

Pen Attributes

The pen attribute of a turtle object is related to its drawing capabilities. The most fundamental of these attributes is whether the pen is currently “up” or “down,” controlled by methods penup() and pendown(). When the pen attribute value is “up,” the turtle can be moved to another location without lines being drawn. This is especially needed when drawing graphical images with disconnected segments.



The pen size of a turtle determines the width of the lines drawn when the pen attribute is “down.” The pensize method is used to control this: the_turtle.pensize(5). The width is given in pixels, and is limited only by the size of the turtle screen.



6.2.4 Additional Turtle Attributes

Turtle Visibility

A turtle's visibility can be controlled by use of methods hideturtle() and showturtle() (in which an invisible turtle can still draw on the screen).

Turtle Size

The size of a turtle shape can be controlled with methods `resizemode` and `turtlesize`.

```
# set to allow user to change turtle size
the_turtle.resizemode('user')

# set a new turtle size
the_turtle.turtlesize(3, 3)
```

Turtle Shape

There are a number of ways that a turtle's shape (and fill color) may be defined to something other than the default shape (the arrowhead) and fill color (black). First, a turtle may be assigned one of the following provided shapes: 'arrow', 'turtle', 'circle', 'square', 'triangle', and 'classic' (the default arrowhead shape).

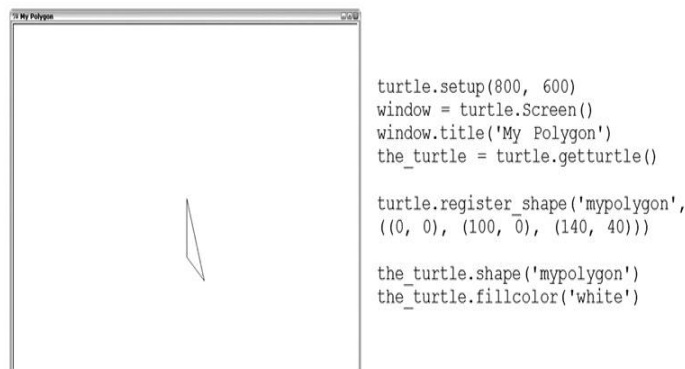


The shape and fill colors are set by use of the `shape` and `fillcolor` methods,

```
the_turtle.shape('circle')  
the_turtle.fillcolor('white')
```

New shapes may be created and registered with (added to) the turtle screen's shape dictionary.

One way of creating a new is shape by providing a set of coordinates denoting a polygon.



Turtle Speed

A turtle's speed can be set to a range of speed values from 0 to 10, with a "normal" speed being around 6. To set the speed of the turtle, the `speed` method is used, `the_turtle.speed(6)`. The following speed values can be set using a descriptive rather than a numeric value,

10: 'fast' 6: 'normal' 3: 'slow' 1: 'slowest' 0: 'fastest'

Thus, a normal speed can also be set by the `_turtle.speed('normal')`. When using the turtle for line drawing only, the turtle will move more quickly if it is made invisible (by use of the `hideturtle` method).

6.2.5 Creating Multiple Turtles

Any number of turtle objects can be created by use of method `Turtle()`.

```
turtle1 = turtle.Turtle()  
turtle2 = turtle.Turtle()  
etc.
```

By storing turtle objects in a list, any number of turtles may be maintained,

```
turtles = []  
turtles.append(turtle.Turtle())  
turtles.append(turtle.Turtle())  
etc.
```

VII-CHAPTER: MODULAR DESIGN

7.1 Modules

7.1.1 What Is a Module?

The term “**module**” refers to the design and/or implementation of specific functionality to be incorporated into a program. The use of modules has a number of advantages.

EX:

- **SOFTWARE DESIGN**
 - provides a means for the development of well-designed programs
- **SOFTWARE DEVELOPMENT**
 - provides a natural means of dividing up programming tasks
 - provides a means for the reuse of program code
- **SOFTWARE TESTING**
 - provides a means of separately testing parts of a program
 - provides a means of integrating parts of a program during testing
- **SOFTWARE MODIFICATION AND MAINTENANCE**
 - facilitates the modification of specific program functionalities

Modular design allows large programs to be broken down into manageable size parts, in which each part (module) provides a clearly specified capability. It aids the software development process by providing an effective way of separating programming tasks among various individuals or teams. It allows modules to be individually developed and tested, and eventually integrated as a part of a complete system. Finally, modular design facilitates program modification since the code responsible for a given aspect of the software is contained within specific modules, and not distributed throughout the program.

7.1.2 Module Specification

Every module needs to provide a specification of how it is to be used. This is referred to as the module's **interface**. Any program code making use of a particular module is referred to as a **client** of the module. A module's specification should be sufficiently clear and complete so that its clients can effectively utilize it.

```
def numPrimes(start, end):

    """ Returns the number of primes between start and end. """
```

The function's specification is provided by the line immediately following the function header, called a docstring in Python. A **docstring** is a string literal denoted by triple quotes given as the first line of certain program elements. The docstring of a particular program element can be displayed by use of the `__doc__` extension,... `print(numPrimes.__doc__)` Returns the number of primes between start and end.

This provides a convenient way for discovering how to use a particular function without having to look at the function definition itself. Some software development tools also make use of docstrings.

Let's consider how complete this specification is for this function. At first look it may seem sufficient. it is not clear what will happen if the function is called with a first argument (start) greater than the second (end). Thus, a more complete specification is needed.

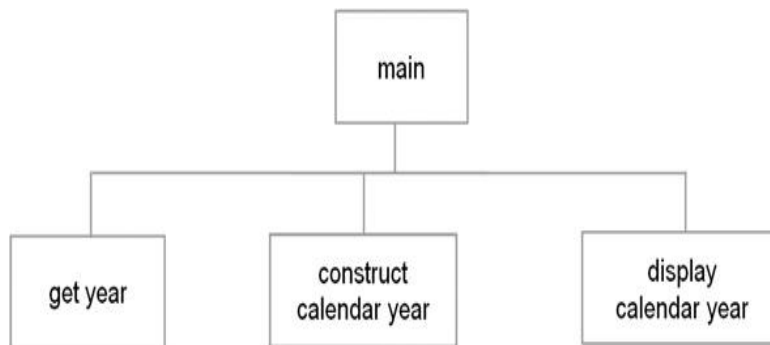
```
def numprimes(start, end):
    """Returns the number of primes between start and end, inclusive.
    Returns -1 if start is greater than end.
    """
```

7.2 Top-Down Design

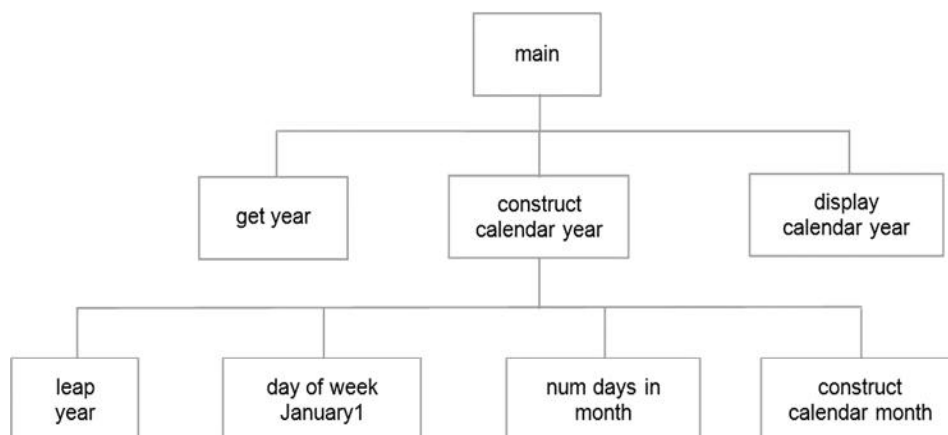
Top-down design is an approach for deriving a modular design in which the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps.

7.2.1 Developing a Modular Design of the Calendar Year Program

The three overall steps of the program are getting the requested year from the user, creating the calendar year structure, and displaying the year.



The goal of top-down design is that each module provides clearly defined functionality, which collectively provides all of the required functionality of the program. Module construct calendar year, on the other hand, is where most of the work is done, and is therefore further broken down.



7.2.2 Specification of the Calendar Year Program Modules

This stage of the design provides sufficient detail from which to implement the program. The main module provides the overall construction of the program.

```
def getYear():
    """ Returns an integer value between 1800-2099, inclusive, or -1. """

def leapYear(year):
    """ Returns True if provided year a leap year, otherwise returns False. """

def dayOfWeekJan1(year, leap_year):
    """ Returns the day of the week for January 1 of the provided year.
        year must be between 1800 and 2099. leap_year must be True if
        year a leap year, and False otherwise.
    """

def numDaysInMonth(month_num, leap_year):
    """ Returns the number of days in a given month.
        month_num must be in the range 1-12, inclusive.
        leap_year must be True if month in a leap year, otherwise False.
    """

def constructCalMonth(month_num, first_day_of_month, num_days_in_month):
    """ Returns a formatted calendar month for display on the screen.
        month_num must be in the range 1-12, inclusive.
        first_day_of_month must be in the range 0-6 (1-Sun, 2-Mon, ..., 0-Sat).
        Returns a list of strings of the form,
        [month_name, week1, week2, week3, week4, week5, week6].
    """

def constructCalYear(year):
    """ Returns a formatted calendar year for display on the screen.
        year must be in the range 1800-2099, inclusive.
        Returns a list of list of strings of the form,
        [year, month1, month2, month3, ..., month12].
    """

def displayCalendar(calendar_year):
    """ Displays the provided calendar_year on the screen three months across.
        Provided calendar year should be in the form of a list of twelve sublists, in which each
        sublist is of the form [monthname, week1, week2, ..., ]
    """

# main
terminate = False

print('This program will display a calendar year for a given year')

while not terminate:
    year = getYear()

    if year == -1:
        terminate = True
    else:
        calendar_year = constructCalYear(year)
        displayCalendar(calendar_year)
```

Calls module getYear to get the year from the user, calls module constructCalYear to construct the year, and finally calls module displayCalendar to display the calendar year. The only detail that the main module is concerned with is allowing the user to keep displaying another calendar year, or enter -1 to terminate the program. This is controlled by Boolean variable terminate.

The first module called, `getYear`, returns the integer value entered by the user. The module's specification indicates that it returns an integer value between 1800 and 2099, inclusive; or 21 (if the user decides to terminate the program). Therefore, it is the responsibility of the module to ensure that no other value is returned. This relieves the main module of having to check for bad input.

The next module that the main module uses is module `constructCalYear`. This module returns a list of twelve sublists, one for each month, in which each sublist begins with the name of the month (as a string value), followed by each of the weeks in the month, each week formatted as a single string. The module's specification indicates that it is to be passed a year between 1800 and 2099, inclusive. Therefore, any year given to it that is outside that range violates its condition for use, and therefore the results are not guaranteed.

The last module called from the main module is module `displayCalendar`. It is given a constructed calendar year, as constructed by module `constructCalMonth`.

7.3 Python Modules

7.3.1 What Is a Python Module?

A **Python module** is a file containing Python definitions and statements. The Python Standard Library contains a set of predefined **standard (built-in) modules**.

7.3.2 Modules and Namespaces

A **namespace** is a container that provides a named context for a set of identifiers. Namespaces enable programs to avoid potential name clashes by associating each identifier with the namespace from which it originates. In software development, a **name clash** is when two otherwise distinct entities with the same name become part of the same scope.

In Python, each module has its own namespace. This includes the names of all items in the module, including functions and global variables—variables defined within the module and outside the scope of any of its functions.

```
# module somemodule
```

```
def func1(n):  
    return n * 10
```

```
def func2(n1, n2):  
    return n1 * n2
```

```
# ---- main
```

```
from somemodule import *
```

```
# NAMESPACES AND FUNCTION USE
```

```
def func2(n):  
    return n * n          # definition of func2 masks imported func2
```

```
print(func1(8))           # outputs 80 (func1 of somemodule called)  
print(somemodule.func1(8)) # NameError: name 'somemodule' is not defined
```

```
print(func2(5))           # outputs 25 (func2 of MAIN called)  
print(func2(3, 8))        # TypeError: func2() takes exactly 1 argument
```

```
print(somemodule.func2(3, 8)) # NameError: name 'somemodule' is not  
                             defined
```

7.3.3 Importing Modules

In Python, the **main module** of any program is the first (“top-level”) module executed. When working interactively in the Python shell, the Python interpreter functions as the main module, containing the global namespace. The namespace is reset every time the interpreter is started (or when selecting Shell → Restart Shell).

The “import modulename ” Form of Import

When using the import modulename form of import, the namespace of the imported module becomes available to, but not part of, the importing module. Identifiers of the imported module, therefore, must be fully qualified (prefixed with the module’s name) when accessed. Using this form of import prevents any possibility of a name clash.

The “from-import” Form of Import

With the from-import form of import, imported identifiers become part of the importing module’s namespace. Because of the possibility of name clashes, import modulename is the preferred form of import in Python. Python also provides an alternate import statement of the form **from** modulename **import** something where something can be a list of identifiers, a single renamed identifier, or an asterisk, as shown below,

- (a) **from** modulename **import** func1, func2
- (b) **from** modulename **import** func1 **as** new_func1
- (c) **from** modulename **import** *

EX:

```
# module somemodule
```

```
def func1(n):  
    return n * 10
```

```
def func2(n1, n2):  
    return n1 * n2
```

```
# ---- main
```

```
from somemodule import *
```

```
# NAMESPACES AND FUNCTION USE
```

```
def func2(n):  
    return n * n          # definition of func2 masks imported func2
```

```
print(func1(8))           # outputs 80 (func1 of somemodule called)  
print(somemodule.func1(8)) # NameError: name 'somemodule' is not defined
```

```
print(func2(5))           # outputs 25 (func2 of MAIN called)  
print(func2(3, 8))        # TypeError: func2() takes exactly 1 argument
```

```
print(somemodule.func2(3, 8)) # NameError: name 'somemodule' is not  
                             defined
```

Module Private Variables

In Python, all the variables in a module are “public,” with the convention that variables beginning with an two underscores are intended to be private.

7.3.4 Module Loading and Execution

Each imported module of a Python program needs to be located and loaded into memory. Python first searches for modules in the current directory. If the module is not found, it searches the directories specified in the PYTHONPATH environment variable. If the module is still not found (or PYTHONPATH is not defined), a Python installation-specific path is searched (e.g., C:\Python32\Lib). If the program still does not find the module, an error (ImportError exception) is reported. For our purposes, all of the modules of a program will be kept in the same directory.

However, if you wish to develop a module made available to other programs, then the module can be saved in your own Python modules directory specified in the PYTHONPATH, or stored in the particular Python installation Lib directory. When a module is loaded, a compiled version of the module with file extension .pyc is automatically produced. Then, the next time that the module is imported, the compiled .pyc file is loaded, rather than the .py file, to save the time of recompiling. A new compiled version of a module is automatically produced whenever the

compiled version is out of date with the source code version of the module when loading, based on the dates that the files were created/modified.

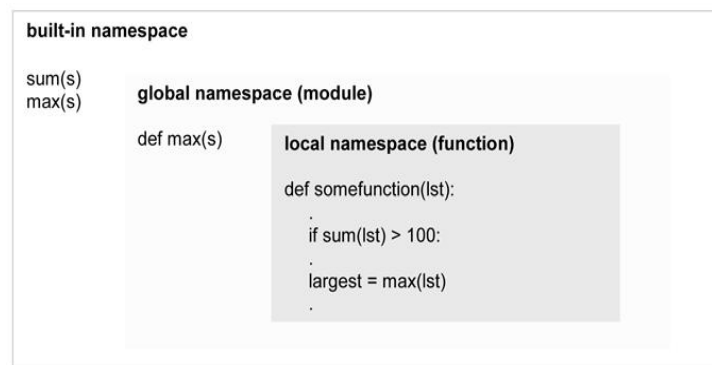
Built-in Function dir()

Built-in function dir() is very useful for monitoring the items in the namespace of the main module for programs executing in the Python shell.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
The following shows the namespace after importing and defining
variables,
>>> import random
>>> n = 10
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'n', 'random']
Selecting Shell → Restart Shell (Ctrl-F6) in the shell resets the namespace,
(after Restart Shell selected)
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

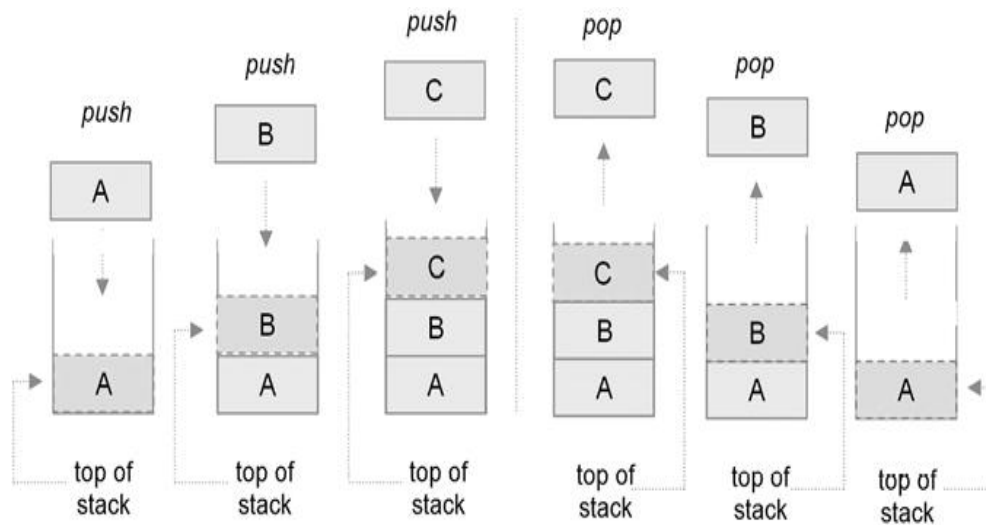
7.3.5 Local, Global, and Built-in Namespaces in Python

The **built-in namespace** contains the names of all the built-in functions, constants, and so on, in Python. The **global namespace** contains the identifiers of the currently executing module. And the **local namespace** is the namespace of the currently executing function (if any).



7.3.6 A Programmer-Defined Stack Module

A **stack** is a very useful mechanism in computer science. Stacks are used to temporarily store and retrieve data. They have the property that the last item placed on the stack is the first to be retrieved. This is referred to as LIFO—“last in, first out.” A stack can be viewed as a list that can be accessed only at one end,



```

1  # stack Module
2
3  def getStack():
4
5      """Creates and returns an empty stack."""
6
7      return []
8
9  def isEmpty(s):
10
11      """Returns True if stack empty, otherwise returns False. """
12
13      if s == []:
14          return True
15      else:
16          return False
17
18  def top(s):
19
20      """Returns value of the top item of stack, if stack not empty.
21         Otherwise, returns None.
22      """
23
24      if isEmpty(s):
25          return None
26      else:
27          return s[len(s) - 1]
28
29  def push(s, item):
30
31      """Pushes item on the top of stack. """
32
33      s.append(item)
34
35  def pop(s):
36
37      """Returns top of stack if stack not empty. Otherwise, returns None."""
38
39      if isEmpty(s):
40          return None
41      else:
42          item = s[len(s) - 1]
43          del s[len(s) - 1]
44          return item

```

```
def getYear():
```

```
    """ Returns an integer value between 1800-2099, inclusive, or -1. """
```

```
def leapYear(year):
```

```
    """ Returns True if provided year a leap year, otherwise returns False. """
```

```
def dayOfWeekJan1(year, leap_year):
```

```
    """ Returns the day of the week for January 1 of the provided year.
```

```
    year must be between 1800 and 2099. leap_year must be True if  
    year a leap year, and False otherwise.
```

```
    """
```

```
def numDaysInMonth(month_num, leap_year):
```

```
    """ Returns the number of days in a given month.
```

```
    month_num must be in the range 1-12, inclusive.
```

```
    leap_year must be True if month in a leap year, otherwise False.
```

```
    """
```

```
def constructCalMonth(month_num, first_day_of_month, num_days_in_month):
```

```
    """ Returns a formatted calendar month for display on the screen.
```

```
    month_num must be in the range 1-12, inclusive.
```

```
    first_day_of_month must be in the range 0-6 (1-Sun, 2-Mon, ..., 0-Sat).
```

```
    Returns a list of strings of the form,
```

```
    [month_name, week1, week2, week3, week4, week5, week6].
```

```
    """
```

```
def constructCalYear(year):
```

```
    """ Returns a formatted calendar year for display on the screen.
```

```
    year must be in the range 1800-2099, inclusive.
```

```
    Returns a list of list of strings of the form,
```

```
    [year, month1, month2, month3, ..., month12].
```

```
    """
```

```
def displayCalendar(calendar_year):
```

```
    """ Displays the provided calendar_year on the screen three months across.
```

```
    Provided calendar year should be in the form of a list of twelve sublists, in which each  
    sublist is of the form [monthname, week1, week2, ..., ]
```

```
    """
```

```
# main
terminate = False

print('This program will display a calendar year for a given year')

while not terminate:
    year = getYear()

    if year == -1:
        terminate = True
    else:
        calendar_year = constructCalYear(year)
        displayCalendar(calendar_year)
```

VIII- CHAPTER: TEXT FILES

8.1 What Is a Text File?

A **text file** is a file containing characters, structured as lines of text. A **binary file** is a file that is formatted in a way that only a computer program can read.

8.2 Using Text Files

Fundamental operations of all types of files include opening a file, reading from a file, writing to a file, and closing a file.

8.2.1 Opening Text Files

All files must first be opened before they can be read from or written to. In Python, when a file is (successfully) opened, a file object is created that provides methods for accessing the file.

Opening for Reading

To open a file for reading, the built-in open function is used as shown,

```
input_file = open('myfile.txt','r')
```

The first argument is the file name to be opened, 'myfile.txt'. The second argument, 'r', indicates that the file is to be opened for reading. (The second argument is optional when opening a file.

Opening for Writing

To open a file for writing, the open function is used as shown below,

```
output_file=open('mynewfile.txt','w')
```

Note that, in this case, 'w' is used to indicate that the file is to be opened for writing. If the file already exists, it will be overwritten (starting with the first line of the file). When using a second argument of 'a', the output will be appended to an existing file instead. It is important to close a file that is written to, otherwise the tail end of the file may not be written to the file (discussed below),

```
output_file.close()
```

When opening files for writing, there is not much chance of an I/O error occurring. The provided file name does not need to exist since it is being created (or overwritten). Thus, the only error that may occur is if the file system (such as the hard disk) is full.

8.2.2 Reading Text Files

The readline method returns as a string the next line of a text file, including the end-of-line character,\n. When the end-of-file is reached, it returns an empty string as demonstrated in the while loop.

Text File myfile.txt <pre>Line One\n Line Two\n Line Three\n</pre>	<pre>input_file = \ open('myfile.txt','r') empty_str = '' line = input_file.readline() while line != empty_str: print(line) line = input_file.readline() input_file.close()</pre>	Screen Output <pre>Line One Line Two Line Three</pre>
---	--	--

It is also possible to read the lines of a file by use of the for statement,

```
input_file = \
open('myfile.txt','r')
or line in input_file:
```

Using a for statement, all lines of the file will be read one by one. Using a while loop, however, lines can be read until a given value is found, for example. Finally, note the blank lines in the screen output. Since read_line returns the newline character, and print adds a newline character, two newline characters are output for each line displayed, resulting in a skipped line after each.

8.2.3 Writing Text Files

The write method is used to write strings to a file,

Text File myfile.txt <pre>Line One\n Line Two\n Line Three\n</pre>	<pre>empty_str = '' input_file = open('myfile.txt','r') output_file = open('myfile_copy.txt','w') line = input_file.readline() while line != empty_str: output_file.write(line) line = input_file.readline() output_file.close()</pre>	Text File myfile_copy.txt <pre>line one line two line three</pre>
---	---	--

This code copies the contents of the input file, 'myfile.txt', line by line to the output file, 'myfile_copy.txt'. In contrast to print when writing to the screen, the write method does not add a newline character to the output string. Thus, a newline character will be output only if it is part of the string being written. In this case, each line read contains a newline character. Finally, when writing to a file, data is first placed in an area of memory called a buffer. Only when the buffer becomes full is the data actually written to the file. (This makes reading and writing files more efficient.) Since the last lines written may not completely fill the buffer, the last buffer's

worth of data may not be written. The `close ()` method flushes the buffer to force the buffer to be written to the file.

8.3 String Processing

String processing refers to the operations performed on strings that allow them to be accessed, analyzed, and updated.

8.3.1 String Traversal

The characters in a string can be easily traversed, without the use of an explicit index variable, using the `for chr in string` form of the `for` statement.

8.3.2 String-Applicable Sequence Operations

All string operations that “modify” a string return a new string that is a modified version of the original string .

Sequences Operations Applicable to Strings			
Length	<code>len(str)</code>	Membership	<code>'h' in s</code>
Select	<code>s[index_val]</code>	Concatenation	<code>s + w</code>
Slice	<code>s[start:end]</code>	Minimum Value	<code>min(s)</code>
Count	<code>s.count(char)</code>	Maximum Value	<code>max(s)</code>
Index	<code>s.index(char)</code>	Comparison	<code>s == w</code>

8.3.3 String Methods

Python provides a number of methods specific to strings, in addition to the general sequence operations. The `find`, `replace`, and `strip` methods in Python can be used to search and produce modified strings.

Checking the Contents of a String			
<code>str.isalpha()</code>	Returns True if <i>str</i> contains only letters.	<code>s = 'Hello'</code>	<code>s.isalpha()</code> → True
		<code>s = 'Hello!'</code>	<code>s.isalpha()</code> → False
<code>str.isdigit()</code>	Returns True if <i>str</i> contains only digits.	<code>s = '124'</code>	<code>s.isdigit()</code> → True
		<code>s = '124A'</code>	<code>s.isdigit()</code> → False
<code>str.islower()</code> <code>str.isupper()</code>	Returns True if <i>str</i> contains only lower (upper) case letters.	<code>s = 'hello'</code>	<code>s.islower()</code> → True
		<code>s = 'Hello'</code>	<code>s.isupper()</code> → False
<code>str.lower()</code> <code>str.upper()</code>	Return lower (upper) case version of <i>str</i> .	<code>s = 'Hello!'</code>	<code>s.lower()</code> → 'hello!'
		<code>s = 'hello!'</code>	<code>s.upper()</code> → 'HELLO!'
Searching the Contents of a String			
<code>str.find(w)</code>	Returns the index of the first occurrence of <i>w</i> in <i>str</i> . Returns -1 if not found.	<code>s = 'Hello!'</code>	<code>s.find('l')</code> → 2
		<code>s = 'Goodbye'</code>	<code>s.find('l')</code> → -1
Replacing the Contents of a String			
<code>str.replace(w,t)</code>	Returns a copy of <i>str</i> with all occurrences of <i>w</i> replaced with <i>t</i> .	<code>s = 'Hello!'</code>	<code>s.replace('H', 'J')</code> → 'Jello'
		<code>s = 'Hello'</code>	<code>s.replace('ll', 'r')</code> → 'Hero'
Removing the Contents of a String			
<code>str.strip(w)</code>	Returns a copy of <i>str</i> with all leading and trailing characters that appear in <i>w</i> removed.	<code>s = ' Hello! '</code> <code>s = 'Hello\n'</code>	<code>s.strip(' !')</code> → 'Hello' <code>s.strip('\n')</code> → 'Hello'
Splitting a String			
<code>str.split(w)</code>	Returns a list containing all strings in <i>str</i> delimited by <i>w</i> .	<code>s = 'Lu, Chao'</code>	<code>s.split(',')</code> → ['Lu', 'Chao']

8.4 Exception Handling

Various error messages can occur when executing Python programs. Such errors are called exceptions. Exceptions can be “caught” and “handled” by a program, however, to either correct the error and continue execution, or terminate the program gracefully.

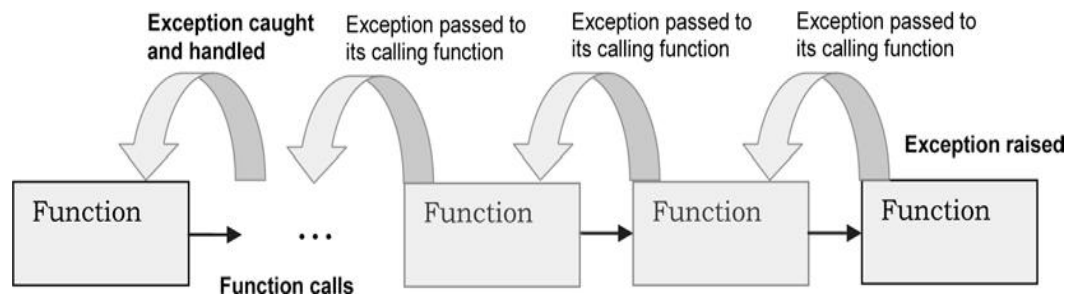
8.4.1 What Is an Exception?

An **exception** is a value (object) that is raised (“thrown”) signaling that an unexpected, or “exceptional,” situation has occurred. Python contains a predefined set of exceptions referred to as **standard exceptions**.

<code>ImportError</code>	raised when an import (or <code>from...import</code>) statement fails
<code>IndexError</code>	raised when a sequence index is out of range
<code>NameError</code>	raised when a local or global name is not found
<code>TypeError</code>	raised when an operation or function is applied to an object of inappropriate type
<code>ValueError</code>	raised when a built-in operation or function is applied to an appropriate type, but of inappropriate value
<code>IOError</code>	raised when an input/output operation fails (e.g., “file not found”)

8.4.2 The Propagation of Raised Exceptions

An exception is either handled by the client code, or automatically propagated back to the client’s calling code, and so on, until handled. If an exception is thrown all the way back to the main module (and not handled), the program terminates displaying the details of the exception.



8.4.3 Catching and Handling Exceptions

Exceptions are caught and handled in Python by use of a **try block** and **exception handler**.

```
import math

num = int(input('Enter number to compute factorial of: '))
try:
    print(math.factorial(num))
except ValueError:
    print('Cannot compute the factorial of negative numbers')
```

```
Enter number to compute factorial of: -5
Cannot compute the factorial of negative numbers
```

The block of code surrounded by try and except headers. The suite following the except header is referred to as an **exception handler**. This exception header “catches” exceptions of type ValueError. The exception handler in this case simply outputs an error message to the user before terminating the program.

Rather than terminating the program, the user is prompted again for input so that the program can continue executing.

```
import math

num = int(input('Enter number to compute factorial of: '))
valid_input = False

while not valid_input:
    try:
        result = math.factorial(num)
        print(result)
        valid_input = True
    except ValueError:
        print('Cannot compute factorial of negative numbers')
        num = int(input('Please re-enter: '))
```

```
Enter number to compute factorial of: -5
Cannot compute the factorial of negative numbers
Please re-enter: 5
120
```

8.4.4 Exception Handling and User Input

Programmer-defined functions may raise exceptions in addition to the exceptions raised by the built-in functions of Python.

```

valid = False

while not valid:
    try:
        month = int(input('Enter current month (1-12): '))

        while month < 1 or month > 12:
            print('Invalid Input - Must be in the range 1-12')
            month = int(input('Enter current month (1-12): '))
        valid = True
    except ValueError:
        print('Invalid Month Value')

```

```

def getMonth():
    month = int(input('Enter current month (1-12): '))

    if month < 1 or month > 12:
        raise ValueError('Invalid Month Value')

    return month

```

```

valid = False

while not valid:
    try:
        month = getMonth()
        valid = True
    except ValueError as err_mesg:
        print(err_mesg, '\n')

```

8.4.4 Exception Handling and File Processing

IOError exceptions raised as a result of a file open error can be caught and handled.

```

file_name = input('Enter file name: ')
empty_str = ''

input_file_opened = False

while not input_file_opened:
    try:
        input_file = open(file_name, 'r')
        input_file_opened = True

        line = input_file.readline()

        while line != empty_str:
            print(line.strip('\n'))
            line = input_file.readline()
    except IOError:
        print('File Open Error\n')
        file_name = input('Enter file name: ')

```

Variable `file_name` stores the file name entered by the user. Variable `input_file_opened` is initialized to `False`. The while loop continues to iterate as long as `input_file_opened` is `False`. Because of the try block within the loop, every time that `open(file_name, 'r')` raises an exception, the remaining lines in the try block are skipped, and the exception handler following the except header is executed. Only when the call to `open` does not throw an exception do all the

instructions in the try block get executed, with the program continuing after the while loop. A similar, but much less likely exception can be raised when opening a file for writing and the file system (hard disk, for example) we want to write to is full. Note that when reading from a text file, the readline method does not raise any exceptions. When the end of file is reached, readline returns an empty string rather than throwing an exception.

UNIT - V

Unit – V

S.No.	Topic	Page No.
1.	Explain dictionary in python.	
2.	Explain set data type in python.	
3.	Define class.	
4.	Define oops.	
5.	Define message passing.	
6.	Explain encapsulation in python.	
7.	Explain inheritance in python.	
8.	Explain polymorphism in python.	
9.	Explain recursive function in python.	

IX - CHAPTER: DICTIONARIES AND SETS

9.1 Dictionary Type in Python

9.1.1 What Is a Dictionary?

A **dictionary** in Python is a mutable, associative data structure of variable length denoted by the use of curly braces.

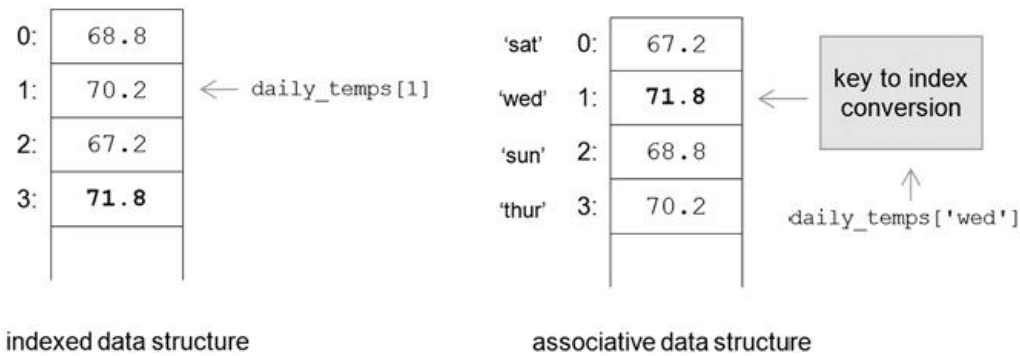
EX:

```
daily_temps = {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8, 'thur': 73.2, 'fri': 75.6, 'sat': 74.0}
```

Each temperature has associated with it a unique key value ('sun', 'mon', etc.). Strings are often used as key values. The syntax for accessing an element of a dictionary is the same as for accessing elements of sequence types, except that a key value is used within the square brackets instead of an index value: `daily_temps['sun']`.

A comparison of accessing indexed data structures vs. associative data structures

EX:



Although the elements of the associative data structure are physically ordered, the ordering is irrelevant to the way that the structure is utilized. The location that an element is stored in and retrieved from within an associative data structure depends only on its key value, thus there is no logical first element, second element, and so forth. The specific location that a value is stored is determined by a particular method of converting key values into index values called hashing.

Although strings are often used as key values, any immutable type may be used as well, such as a tuple.

EX:

```
temps = { ('Jan', 2, 2004): 34.8, ('Apr', 14, 2001): 74.6,
          ('Mar', 6, 2000): 68.2, ('Dec', 21, 2002): 28.8,
          ('Nov', 30, 2003): 61.0, ('Nov', 30, 2003): 61.0,
          ('Apr', 14, 2001): 74.6, ('Jan', 2, 2004): 34.8,
          ('Dec', 21, 2002): 28.8, ('Mar', 6, 2000): 68.2 }
```

The operations related to the dynamic creation and updating of dictionaries

Operation	Results
<code>dict()</code>	Creates a new, empty dictionary
<code>dict(s)</code>	Creates a new dictionary with key values and their associated values from sequence <code>s</code> , for example, <code>fruit_prices = dict(fruit_data)</code> where <code>fruit_data</code> is (possibly read from a file): <code>[['apples', .66], ..., ['bananas', .49]]</code>
<code>len(d)</code>	Length (num of key/value pairs) of dictionary <code>d</code> .
<code>d[key] = value</code>	Sets the associated value for <code>key</code> to <code>value</code> , used to either add a new key/value pair, or replace the value of an existing key/value pair.
<code>del d[key]</code>	Remove key and associated value from dictionary <code>d</code> .
<code>key in d</code>	True if key value <code>key</code> exists in dictionary <code>d</code> , otherwise returns False.

9.2 Set Data Type

9.2.1 The Set Data Type in Python

A **set** is a mutable data type with non duplicate, unordered values, providing the usual mathematical set operations

Set operator	Set A = {1,2,3} Set B = {3,4,5,6}		
membership	1 in A	True	True if 1 is a member of set
add	A.add(4)	{1,2,3,4}	Adds new member to set
remove	A.remove(2)	{1,3}	Removes member from set
union	A B	{1,2,3,4,5,6}	Set of elements in either set A or set B
intersection	A & B	{3}	Set of elements in both set A and set B
difference	A - B	{1,2}	Set of elements in set A, but not set B
symmetric difference	A ^ B	{1,2,4,5,6}	Set of elements in set A or set B, but not both
size	len(A)	3	Number of elements in set (general sequence operation)

EX:

```
>>>fruit = {'apple', 'banana', 'pear', 'peach'}
>>> fruit
{'pear', 'banana', 'peach', 'apple'}
>>>'apple' in fruit
True
```

Note that the items in the set are not displayed in the order that they were defined. Sets, like dictionaries, do not maintain a logical ordering. The order that items are stored is determined by Python, and not by the order in which they were provided. Therefore, it is invalid and makes no sense to access an element of a set by index value.

The add and remove methods allow sets to be dynamically altered during program execution, as shown below,

```
>>> fruit.add('pineapple')
>>> fruit
{'pineapple', 'pear', 'banana', 'peach', 'apple'}
```

To define an initially empty set, or to initialize a set to the values of a particular sequence, the set constructor is used,

EX:

```
>>> set1 = set()    >>> vegs = ['peas', 'corn']    >>> vowels = 'aeiou'
>>> len(set1)    >>> set(vegs)    >>> set(vowels)
0    {'corn', 'peas'}    {'a', 'i', 'e', 'u', 'o'}
```

Note that set(), and not empty braces are not used to create an empty set, since that notation is

used to create an empty dictionary. Because sets do not have duplicate elements, adding an already existing item to a set results in no change to the set.

Finally, there are two set types in Python—the mutable set type, and the immutable frozenset type. Methods add and remove are not allowed on sets of frozenset type. Thus, all its members are declared when it is defined,

EX:

```
>>> apple_colors = frozenset(['red', 'yellow', 'green'])
```

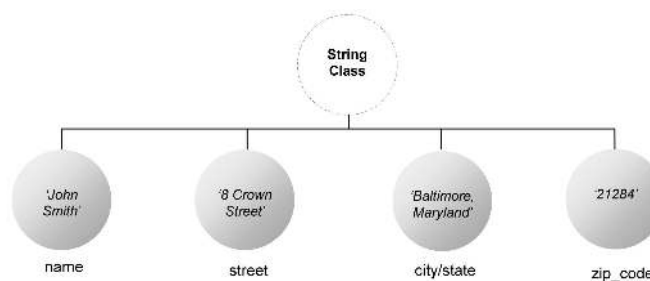
X – CHAPTER : OBJECT-ORIENTED PROGRAMMING

10.1 What Is Object-Oriented Programming?

10.1.1 What Is a Class?

A **class** specifies the set of instance variables and methods that are “bundled together” for defining a type of object. A class, therefore, is a “cookie cutter” that can be used to make as many object instances of that type object as needed.

EX:



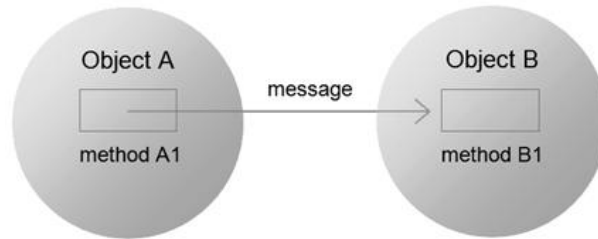
10.1.2 Three Fundamental Features of Object-Oriented Programming

Object-oriented programming languages, such as Python, provide three fundamental features that support object-oriented programming— encapsulation , inheritance , and polymorphism .

Message Passing

These support a paradigm shift in software development from the focus on variables and passing of variables to functions in procedural programming, to the focus on objects and message passing between them.

EX:

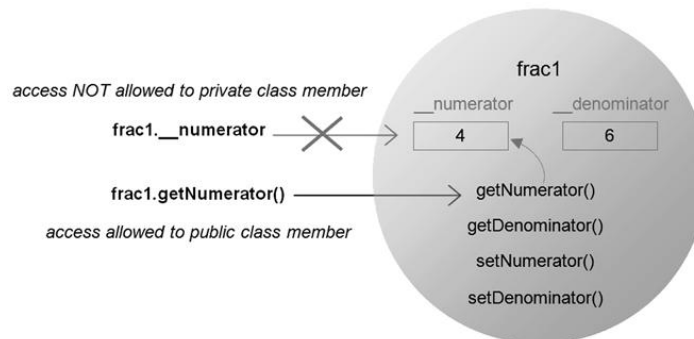


10.2 Encapsulation

10.2.1 What Is Encapsulation?

Encapsulation is a means of bundling together instance variables and methods to form a given type (class). Selected members of a class can be made inaccessible (“hidden”) from its clients, referred to as information hiding . Information hiding is a form of abstraction.

EX:



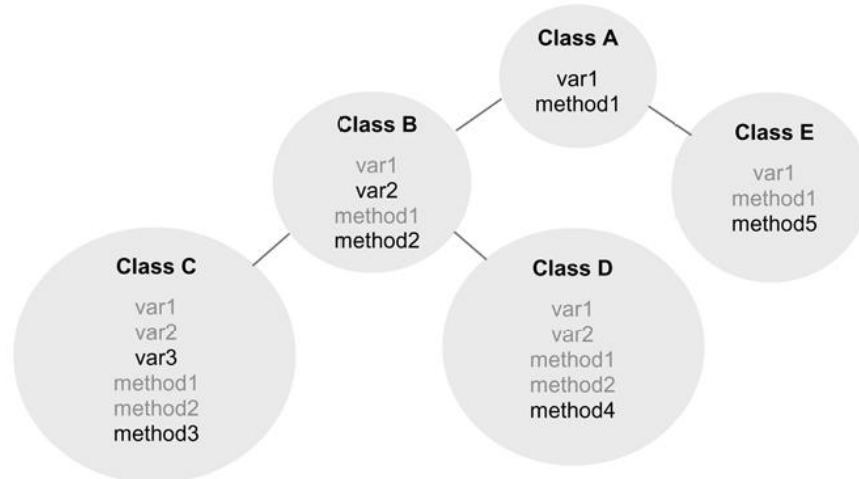
These methods are referred to as **getters** and **setters** since their purpose is to get (return) and set (assign) private instance variables of a class. Restricting access to instance variables via getter and setter methods allows the methods to control what values are assigned (such as not allowing an assignment of 0 to the denominator), and how they are represented when retrieved. Thus, the instance variables of a class are generally made private, and the methods of the class generally made public.

10.3 Inheritance

10.3.1 What Is Inheritance?

Inheritance , in object-oriented programming, is the ability of a class to inherit members of another class as part of its own definition. The inheriting class is called a **subclass** (also “derived class” or “child class”), and the class inherited from is called the **superclass** (also “base class” or “parent class”). Superclasses may themselves inherit from other classes, resulting in a hierarchy of classes.

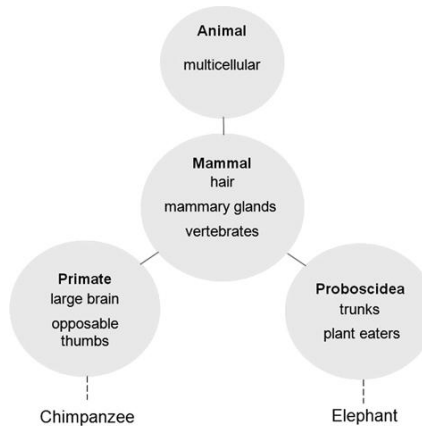
EX:



Class A is the superclass of all the classes in the figure. Thus, subclasses B and E each inherit variable `var1` and method `method1` from Class A. In addition, Class B defines variable `var2` and method `method2`, and Class E defines `method5`. Since Class C is a subclass of Class B, it inherits everything in Class A and Class B, adding `var3` and `method3` in its own definition. And since Class D is also a subclass of Class B, it inherits everything in Class A and Class B, also defining `method4`.

10.3.2 Subtypes

A **subtype** is something that can be substituted for and behave as its parent type (and its parent's parent type, etc.)



10.3.3 Defining Subclasses in Python

Class Names of the Built-In Types in Python

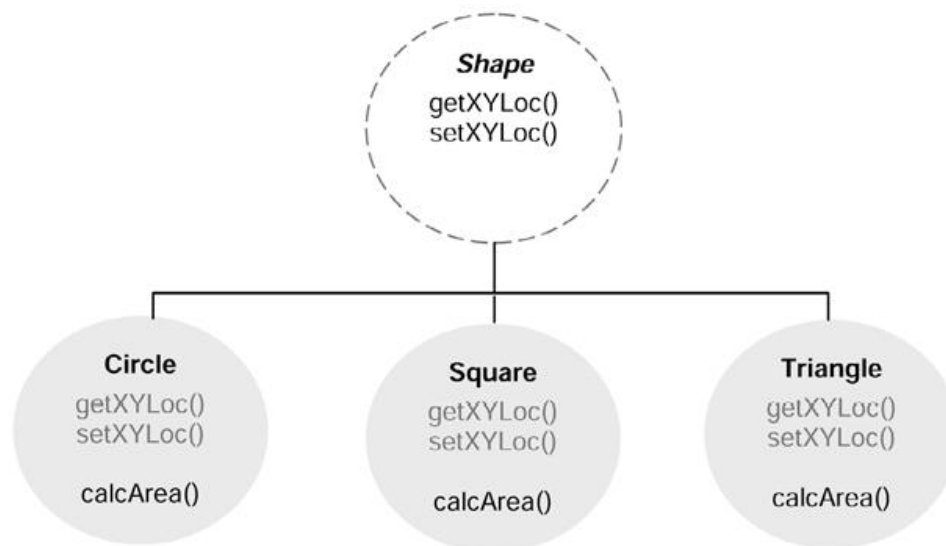
There exists a class definition for each of the built-in types. To determine the type (class name) of a particular value (object) in Python, the built-in function `type` can be used.

10.4 Polymorphism

10.4.1 What Is Polymorphism?

The word polymorphism derives from Greek meaning “something that takes many forms.” In object oriented programming, **polymorphism** allows objects of different types, each with their own specific behaviors, to be treated as the same general type.

EX:



```
class Shape:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def getXyLoc(self):
        return (self.__x, self.__y)

    def setXyLoc(self, x, y):
        self.__x = x
        self.__y = y

    def calcArea(self):
        raise NotImplementedError("Method calcArea not implemented")
```

Subclasses of the Shape class must implement the calcArea method, otherwise a Not-ImplementedError exception is raised. A class in which one or more methods are unimplemented (or only implemented to raise an exception) is called an **abstract class**.

```
class Circle(Shape):
```

```
    def __init__(self, x, y, r):  
        Shape.__init__(self, x, y)  
        self.__radius = r
```

```
    def calcArea(self):  
        return math.pi * self.__radius ** 2
```

```
class Square(Shape):
```

```
    def __init__(self, x, y, s):  
        Shape.__init__(self, x, y)  
        self.__side = s
```

```
    def calcArea(self):  
        return self.__side ** 2
```

```
class Triangle(Shape):
```

```
    def __init__(self, x, y, s):  
        Shape.__init__(self, x, y)  
        self.__side = s
```

```
    def calcArea(self):  
        return (self.__side ** 2) * math.sqrt(3) / 4.0
```

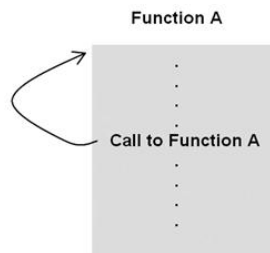
XI - CHAPTER – RECURSION

11.1 Recursive Functions

11.1.1 What Is a Recursive Function?

A **recursive function** is often defined as “a function that calls itself.”

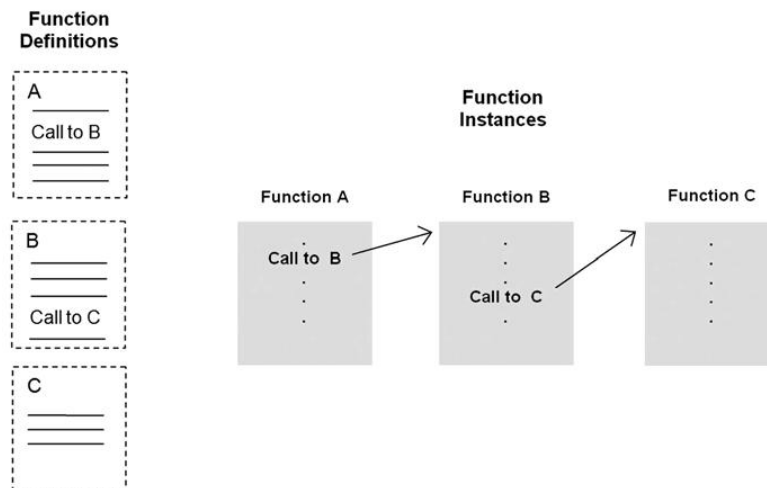
EX:



The notion of a self-referential function is inherently confusing. There are two types of entities related to any function however—the function definition , and any current execution instances. What is meant by the phrase “a function that calls itself ” is a function execution instance that calls another execution instance of the same function. A function definition is a “cookie cutter” from which any number of execution instances can be created. Every time a call

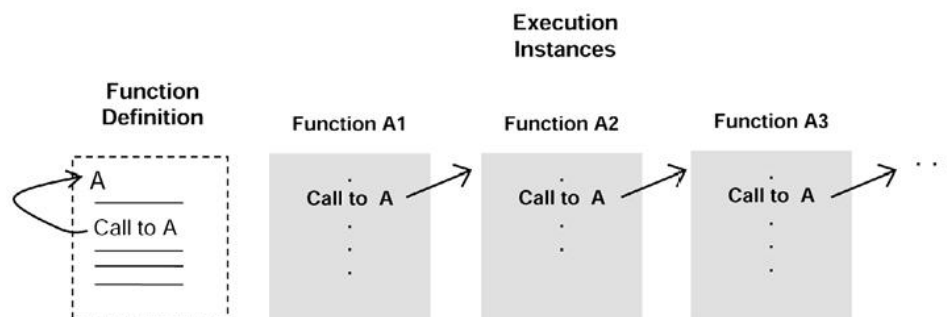
to a function is made, another execution instance of the function is created. Thus, while there is only one definition for any function, there can be any number of execution instances.

General Function Calls



First, an execution instance from the definition of function A is created and begins executing. When the call to function B is reached, the execution instance of function A is suspended while an execution instance of function B is created and begins executing. In turn, when the function call to function C is reached, function B suspends execution while an execution instance of function C is created and begins executing. This calling and suspending of executing function instances could (theoretically) continue indefinitely. However, in this case, function C does not make a call to any other function. Thus, it simply executes until termination, returning control to the function that called it, function B. Function B then continues its execution until terminating, returning control to the function that called it, function A. Finally, function A completes and terminates, returning control to wherever it was called from.

Recursive Function Execution Instances



Note that the execution of a series of recursive function instances is similar to the execution of a series of non-recursive instances, except that the execution instances are “clones” of each other (that is, of the same function definition). Thus, since all instances are identical, the

function calls occur in exactly the same place in each. Clearly, if the definition of a recursive function were written so that the function calls itself unconditionally, then every execution instance would unconditionally call another execution instance, ad infinitum. Such a nonterminating sequence of calls is referred to as **infinite recursion**, similar to the notion of an infinite loop. Therefore, properly designed recursive functions always conditionally call another execution instance so that eventually the chain of function calls terminates.

11.1.2 The Factorial Function

The Recursive Definition of the Factorial Function

The factorial function is an often-used example of the use of recursion. The computation of the factorial of 4 is given as,

$$\text{factorial}(4) = 4 * 3 * 2 * 1 = 24$$

In general, the computation of the factorial of any (positive, nonzero) integer n is,

$$\text{factorial}(n) = n * (n - 1) * (n - 2) * \dots 1$$

The one exception is the factorial of 0, defined to be 1. Note that if we apply this definition to the factorial of $n - 1$, we get $\text{factorial}(n - 1) = (n - 1)(n - 2) \dots 1$. Therefore, the factorial of n can be defined as n times the factorial of $n - 1$,

$$\text{factorial}(n) = n * (n - 1) * (n - 2) * \dots 1$$

$$\text{factorial}(n - 1)$$

Thus, the complete definition of the factorial function is,

$$\text{factorial}(n) = 1, \text{ if } n = 0$$

$$= n * \text{factorial}(n - 1), \text{ otherwise}$$

This definition of the factorial function is clearly defined in terms of itself, referred to as a **recursive definition**. The part of the definition “ $\text{factorial}(n) = 1$, if $n = 0$ ” is referred to as the **base case**. The base case of a recursive definition is what terminates the repeated application of the definition (and thus the repeated function calls when executed).

Consider what would happen if the base case for the factorial function were not part of the definition,

$$\text{factorial}(n) = n * \text{factorial}(n - 1), \text{ for all } n$$

Applying this definition, the computation of the factorial of 4 would be,

$$\text{factorial}(4) = 4 * 3 * 2 * 1 * 0 * (-1) * (-2) * (-3) * (-4) * \dots$$

The factorial of any (positive) number would be 0, since 0 would always be part of the product of values. Thus, not only is this an incorrect definition of the factorial function, if we implemented this definition as a recursive function, it would never terminate (and thus never produce a result). Suppose, on the other hand, the base case for the definition of the factorial function is given, but that $n * \text{factorial}(n - 1)$ were given as $n * \text{factorial}(n + 1)$ instead,

$$\begin{aligned} \text{factorial}(n) &= 1, \text{ if } n = 0 \\ &= n * \text{factorial}(n - 1), \text{ otherwise} \end{aligned}$$