

CS 1032: Programming Fundamentals

Part A: Introduction to Python Programming

BSc Engineering - Semester 1 (2019 Intake)

January – May 2020

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

Compiled by:

Sanath Jayasena, Shehan Perera and Thirasara Ariyaratna.

© Dept of Computer Science & Engineering, University of Moratuwa

Acknowledgements:

All help and support provided to make this Course Note possible are acknowledged.

Sections of this note were based on:

1. Python documentation at <https://docs.python.org/3/index.html>
2. Python Tutorial at <http://www.tutorialspoint.com/>
3. Python Tutorial at <https://docs.python.org/3/tutorial/index.html>
4. Introduction to Computing Using Python: An Application Development Focus by Ljubomir Perkovic.
5. “Introduction to C Programming” Course Notes (2013) from the Dept of Computer Science & Engineering, compiled by Sanath Jayasena, Dilum Bandara, Shehan Perera and others.
6. Early version of this document and feedback received on it. Contributors include Adeesha Wijayasiri, Sachini Herath, Darshana Priyasad and Sanduni Prasadi.

Date of this Part A version: Saturday, 04 January 2020

Note: Feedback is very much appreciated for improvement of this document. Please email at sanath@cse.mrt.ac.lk.

Table of Contents

1 - Background.....	1
1.1 Programs and Programming	1
1.1.1 Program Development	1
1.1.2 Define the Problem	1
1.1.3 Outline the Solution	2
1.1.4 Develop an Algorithm.....	2
1.1.5 Test the Algorithm for Correctness.....	3
1.1.6 Code the Algorithm.....	4
1.1.7 Ensure Program Has No Syntax Errors.....	4
1.1.8 Compile to Generate Translated Code (Optional)	4
1.1.9 Run the Program	4
1.1.10 Test and Debug the Program.....	4
1.1.11 Document and Maintain the Program	4
1.2 Program Translation and Execution	4
1.2.1 Compilation and Compilers	5
1.2.2 Interpretation and Interpreters.....	5
1.2.3 Combination of Compilation and Interpretation	6
1.3 Programming Languages.....	6
1.4 The C Programming Language	7
1.4.1 Steps in Developing a Program in C.....	7
1.4.2 An Example C Program	8
1.4.3 Compiling and Running a C Program.....	9
1.4.4 C Keywords and Tokens	9
2 - Introduction to Python.....	11
2.1 Overview of Python.....	11
2.2 How to Use Python.....	11
2.2.1 Interactive Mode	11
2.2.2 Scripting Mode	11
2.3 Basic Data Types in Python	12
2.4 Python Objects	12
2.5 Mutability of Objects.....	14
2.6 Arithmetic and Algebraic Expressions.....	14
2.7 Boolean Expressions and Operators.....	16

2.8 Variables and Assignments.....	17
2.9 Python Programs.....	18
2.10 Input, Output and Files.....	19
2.10.1 The <code>input()</code> Function.....	19
2.10.2 Reading Input from a File	19
2.10.3 Writing to a File	20
2.11 Python Keywords	20
3 - Operators and Expressions in Python	21
3.1 Types of Operators	21
3.2 Python Arithmetic Operators	21
3.3 Python Comparison Operators.....	21
3.4 Python Assignment Operators	22
3.5 Python Bitwise Operators	22
3.6 Python Logical Operators	23
3.7 Python Membership Operators	23
3.8 Python Identity Operators.....	23
3.9 Operator Precedence in Python.....	23
4 - “Selection” Control Structures in Python	24
4.1 Control Flow	24
4.2 “ <code>if</code> ” Structure	25
4.3 “ <code>if...else</code> ” Structure.....	26
4.4 Multi-way Selection with the “ <code>elif</code> ” Keyword.....	27
4.5 Nested Selection Structures	28
4.6 Exception Handling	28
5 - Loop Control Structures	30
5.1 The <code>for</code> Loop	30
5.2 The <code>while</code> Loop.....	31
5.3 Using <code>else</code> with Loops	32
5.4 Nested Loops	33
5.5 The <code>break</code> Keyword	34

5.6 The <i>continue</i> Keyword.....	34
5.7 The <i>pass</i> Keyword	35
6 - Lists.....	36
6.1 Sequences	36
6.2 Python Lists.....	36
6.3 Accessing Values in Lists.....	36
6.4 Updating Lists	36
6.5 Deleting List Elements	36
6.6 Other List Operations	37
6.7 Indexing and Slicing.....	37
7 - Numbers	38
7.1 Introduction	38
7.2 Examples	38
7.3 Number Type Conversion	39
8 - Strings.....	40
8.1 Introduction	40
8.2 Accessing Elements and Slicing.....	40
8.3 Can We Update Strings?	41
8.4 Escape Characters.....	42
8.5 String Special Operators.....	42
8.6 String Formatting Operator %	43
9 - Functions	45
9.1 Defining a Function.....	45
9.2 Calling a Function	46
9.3 Parameter Passing.....	46
9.4 Function Arguments	47
9.4.1 Required Arguments	48
9.4.2 Keyword Arguments.....	48
9.4.3 Default Arguments.....	49
9.4.4 Variable-Length Arguments	49
9.5 The <i>return</i> Statement	50

9.6 Scope of Variables	50
9.7 Global vs. Local Variables	50
10 - Handling Files	52
10.1 Printing to the Screen	52
10.2 Reading Keyboard Input	52
10.3 Opening and Closing of Files.....	52
10.3.1 The open Function.....	53
10.3.2 The close Method	54
10.4 Reading and Writing Files.....	55
10.4.1 File Reading	55
10.4.2 File Writing	56
10.5 File Positions	56
10.6 File and Directory Handling with the "os" Module	57
Appendix A - Library Functions	58

1 - Background

1.1 Programs and Programming

A (software) program is an ordered sequence of instructions that the hardware of a computer can execute. It performs a particular computational task in a step-by-step manner. A program is like a recipe in which a set of instructions tells a cook how to make a particular dish. It describes the ingredients (the *data*) and the sequence of steps (the *process*) on what to do with those ingredients.

A computer program is expressed using a *programming language*. *Computer programming* is the art of developing computer programs. A person who develops a program using a programming language is a *programmer*. Generally, he/she will first develop an *algorithm* to solve the given *problem* and then convert the algorithm into a program. An algorithm is a sequence of a finite number of well-defined steps for accomplishing some task. The programmer should also test the program to see whether it is working as expected and take corrective actions if not.

1.1.1 Program Development

Developing a program generally involves the following sequence of steps:

1. Define the problem
2. Outline the solution
3. Develop an algorithm
4. Test the algorithm for correctness
5. Code the algorithm in a programming language to obtain a program
6. Ensure program has no syntax errors
7. Compile to generate translated code (optional)
8. Run the program
9. Test and debug the program
10. Document and maintain the program

Most of these steps are common to any problem-solving task. Program development can take several hours, days, weeks, months or years, depending on the complexity and scale of the problem. After development, users will use the program. While in use the program needs to be maintained. The maintenance phase will continue for several months or several years. Therefore, program development is not a onetime task; it is a cycle where some of the above steps are performed again and again. The steps are discussed in detail next.

1.1.2 Define the Problem

First, the problem should be clearly defined. The problem can be divided into three components:

- Outputs – what do you want to have?
- Inputs – what do you have?
- Processing – how do you go from inputs to outputs?

Programmers should clearly understand “the inputs”, “what is expected as output(s)” and “how to process inputs to generate necessary outputs”. Consider an example where we need to calculate and output the circumference and area of a circle when the radius is given as an input.

- Outputs – circumference (c) and area (a) of a circle
- Input – the radius (r) of the circle
- Processing – need to find out how to produce the outputs from the input

1.1.3 Outline the Solution

The programmer should next outline the processing required to solve the problem. This involves identifying the following.

- the major computational tasks and any subtasks
- the major variables and data structures (to store data; e.g., inputs, intermediate results)
- the control structures (e.g., sequence, selection, repetition) that can be used

Consider the above example. The processing required is to compute $c = 2\pi r$ and $a = \pi r^2$.

In order to calculate the circumference:

- Variables – radius (r), circumference (c)
- Computation – $c = 2\pi r$

In order to calculate the area:

- Variables – radius (r), area (a)
- Computation – $a = \pi r^2$

In this example the processing does not require repetitive computations.

1.1.4 Develop an Algorithm

The next step is to develop an algorithm based on the solution outline of previous step. The algorithm should consist of a precise and clear sequence of steps that will solve the problem. Note that more than one algorithm could exist for a given problem. Developing an algorithm is perhaps the most important step in the process of developing a program. For a complex problem, developing an algorithm is generally more difficult than obtaining a computer program by coding an algorithm.

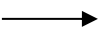
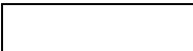
Pseudocode (a structured form of the English language) can be used to express an algorithm. A suitable algorithm in pseudocode for our example would be as follows.

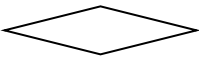
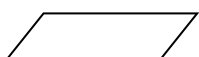
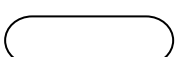
- | | | |
|------|----|---|
| Step | 1. | Start |
| | 2. | Input r |
| | 3. | Compute circumference
$c \leftarrow 2 * \text{PI} * r$ |
| | 4. | Compute area
$a \leftarrow \text{PI} * r * r$ |
| | 5. | Output c and a |
| | 6. | Stop |

In the above, 'PI' is a constant that represents the value of π and '*' is the multiplication operator.

Another tool that can be used to express algorithms is *flowcharts*. A flowchart is an easy to understand pictorial representation of an algorithm. In flowcharts different steps in the algorithm are represented by differently shaped boxes and the flow of the algorithm is indicated by arrows. Table 1.1 shows a commonly used set of flowchart symbols.

Table 1.1: Some Commonly Used Flowchart Symbols

Symbol	Description
	Flow is used to connect different symbols and indicates the direction of flow of the algorithm.
	Process symbol can be used for a single step or an entire sub process.

	Decision indicates a choice and selection of a path by the algorithm based on a certain criterion.
	Input / Output represents input or output data to the algorithm.
	Start / Stop represents the start / end of the algorithm.

The flowchart in Figure 1.1(a) shows the flowchart representation of the algorithm for our example above. The flowchart in Figure 1.1(b) shows an algorithm to output the summation of the first 100 natural numbers.

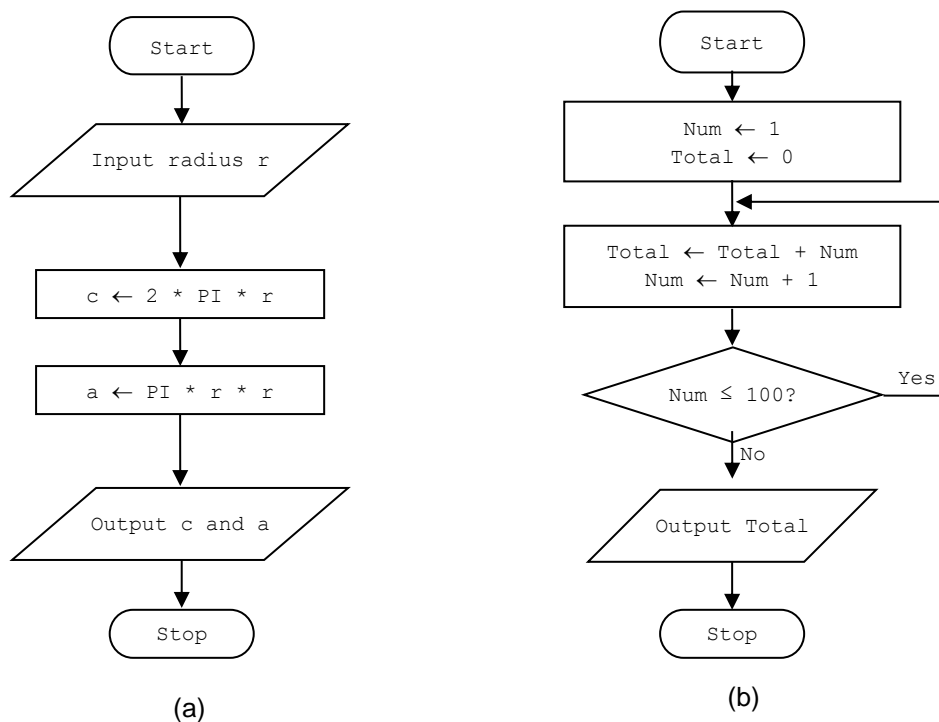


Figure 1.1: Example algorithms expressed using flowcharts

The algorithm shown in Figure 1.1(b) can be expressed in pseudo-code form as follows.

- Step 1. Start
2. $\text{Num} \leftarrow 1$
 $\text{Total} \leftarrow 0$
3. $\text{Total} \leftarrow \text{Total} + \text{Num}$
 $\text{Num} \leftarrow \text{Num} + 1$
4. If $\text{Num} \leq 100$ go to Step 3 // implied: else continue to Step 5
5. Output Total
6. Stop

Among different forms of pseudocode, the above is simple and easy to understand for an average person. There are other formats used by practitioners in the field due to their expressive power and succinctness.

1.1.5 Test the Algorithm for Correctness

The programmer must make sure that the algorithm is correct. The objective is to identify any errors early, so that correction is easy. Test data should be applied to each step, to check whether the algorithm actually does

what it is supposed to do. Up to this step, the work can be done using a pen and paper (although one may use software tools to develop and test complex algorithms).

1.1.6 Code the Algorithm

After all the considerations have been met and the algorithm is checked, the algorithm is coded using a suitable programming language. The result is a computer program. We code this by editing (typing-in) on a computer using a program (software) such as a simple text editor or a more sophisticated integrated development environment (IDE).

1.1.7 Ensure Program Has No Syntax Errors

When an algorithm is coded in a programming language, we have to ensure that the code obeys and does not violate the syntax (grammar) rules of the language. This means, the program we just created must not have syntax errors. Instead of tedious and error prone manual checking (by humans), we can check for syntax errors automatically (much easily, quickly and accurately) using programs that are meant for that. Examples of such programs include language translators such as compilers and interpreters. We will keep editing our program to correct syntax errors reported by the compiler/interpreter until there is none. This process of editing-checking can take several iterations.

1.1.8 Compile to Generate Translated Code (Optional)

As an optional step, depending on the programming language and the language implementation being used, we may have to use a compiler (a language translator program) to generate translated code. The compiler will translate our program into machine language and create an executable program, which will be stored on disk. This optional step will be combined with the previous step of checking for syntax errors, because the compiler will translate our program only if there are no syntax errors. This step will not be necessary if we are using an interpreter. (We will discuss compilers and interpreters in Section 1.2).

1.1.9 Run the Program

How we run (execute) our program on the computer will depend on the programming language and the language implementation being used. If the optional step above was completed, then the executable program generated after compiling can be executed. If not, we will run our program using the language interpreter. There are several possible outcomes when we run a program; for example, it may appear that the program is not running, it may terminate unexpectedly, it may produce unexpected or incorrect output, it may give correct output.

1.1.10 Test and Debug the Program

Just because our program runs, it does not necessarily mean our program is correct. We have to test the program adequately, using many input test data and in different environments, whether it behaves as expected. During the process, we may detect *runtime errors* (e.g., unexpected termination) or *logic errors* (errors due to mistakes in computations and/or the algorithm) in the program. If errors are encountered, we have to determine the causes (bugs) and go back to the relevant step in the process (e.g., algorithm development, coding of the algorithm) to correct them and repeat the process. This cycle continues until we are satisfied with the testing. Removing bugs in a program is called *debugging*. There are software tools and techniques that can help programmers with testing and debugging of programs.

1.1.11 Document and Maintain the Program

Programs are usually not static; they evolve (get revised), due to numerous reasons. A common reason is the change of requirements, which is equivalent to a change in the problem that the program is expected to solve. Another reason is bugs being found out later, after it is used for some time. The program should be maintained (revised or evolved) as a result. For this purpose, it is very important that the algorithm and code are well documented for future reference.

1.2 Program Translation and Execution

Programs are coded today in human-readable languages called *high-level languages*. Programs coded in such high-level languages are referred to as *source programs* or *source code*. A program is developed to solve a computational problem by executing it on the computer. However, the computer understands only 1's and 0's

(referred to as the *machine language*¹ or *machine code*). Therefore, we need to translate (convert) a human-readable, high-level language program into machine-language instructions for execution. There are two main ways this translation is achieved, depending on the *language implementation*:

- Compilation (by a compiler)
- Interpretation (by an interpreter)

1.2.1 Compilation and Compilers

In a *compiled-implementation* of a high-level language, there is a *compiler* that will translate a source program directly into executable machine code that is specific to the target machine (CPU) and operating system (see Figure 1.2). A compiler usually first produces an intermediate binary form called *object code* from source code. Object code will call on building block functions which are kept in a library of object code modules. These separate object codes can be combined by a *linker* during a process called *linking* that produces the final executable code. Here, for simplicity, we can consider compiling includes linking also (i.e., the compiler does the work of the linker).

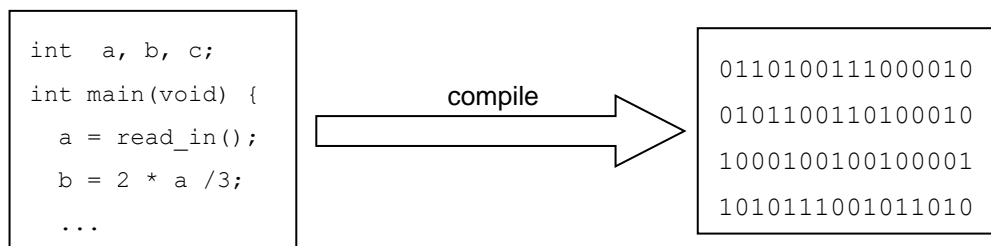


Figure 1.2: Compiling a high-level language program into machine-language program

Executable machine code produced by a compiler can be saved on disk as a file (referred to as the *executable file*) and executed repeatedly as and when necessary (see Figure 1.3). Since the source program is already compiled, no compilation is needed again unless the source program is modified. The saving on the translation time is an advantage gained from compiling. Programming languages such as FORTRAN, COBOL, C, C++, Pascal and Java have compiled-implementations.

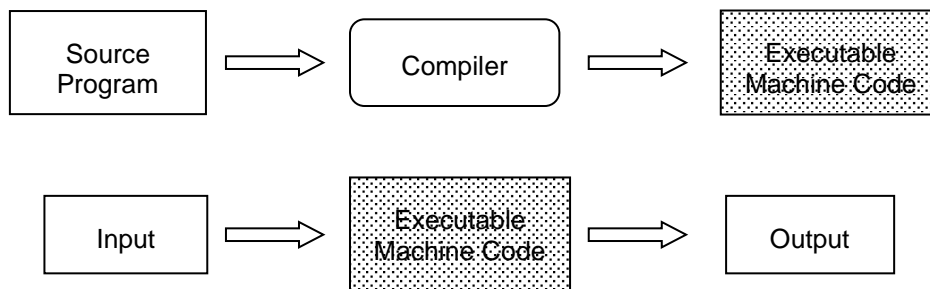


Figure 1.3: Compiling and executing a program

1.2.2 Interpretation and Interpreters

In an *interpreted-implementation* of a high-level language, a program referred to as an *interpreter* reads the original source code, translates it into machine-code and then executes (generally, line-by-line or one statement at a time). The interpreter is specifically implemented for the target machine. Translated machine-code of the full original source program is neither produced nor saved on disk, unlike in compiled-implementations. So, we can consider interpretation as translating a source program and executing it at the same time (see Figure 1.4). Note that interpretation described this way is a simplified view.

¹ Machine language is the native language of the computer (i.e. what is understood by the CPU hardware).

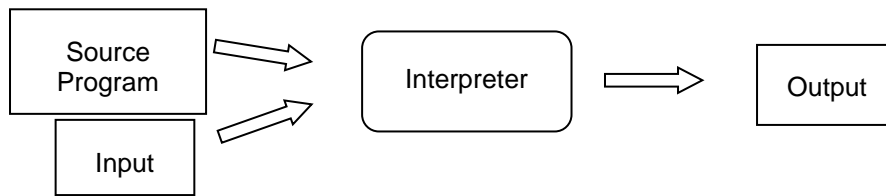


Figure 1.4: Interpretation of a program

Every time the program runs, the interpreter translates high-level language statements into a machine-code format and executes them. In contrast, a compiler translates the whole program from source code just once. Thus, the interpretation process (that requires both translating and executing a program, every time) can be slower than directly executing a previously compiled machine code. Languages like BASIC, Perl, Smalltalk, Lisp, Python, MATLAB, Ruby have interpreted-implementations.

1.2.3 Combination of Compilation and Interpretation

In some modern language implementations, the original source code is first converted into some intermediate code (e.g., *bytecode*) which is then converted into machine code at the time of execution. The former is a compilation, and the latter is an interpretation; so, the whole process is a combination of compilation and interpretation. Java and Python languages have such implementations.

In Java, the Java source code is compiled once into intermediate representation, the Java bytecode, and saved on disk. In Python, the source code is compiled into bytecode each time before execution or each time a change in the source is detected before execution. In both Java and Python, the bytecode is then interpreted by an interpreter that is referred to as a “virtual machine”.

1.3 Programming Languages

Modern programming languages were invented to make programming easier. They became popular because they are much easier to handle than machine language. Many programming languages are designed to be both high-level and general purpose.

A language is high-level if it is independent of the underlying hardware of a computer. It is general purpose if it can be applied to a wide range of situations. There are so many programming languages and some of these languages are general purpose while others are suitable for specific classes of applications. Languages such as C, C++, Java, C#, Python and Visual Basic can be used to develop a variety of applications. On the other hand, FORTRAN was initially developed for numerical computing, Lisp for artificial intelligence and Simula for simulation.

In the early days, computers were programmed using machine language instructions that the hardware understood. Machine language belongs to the *first generation* of programming languages. These programs were hardly human readable sequences of 0's and 1's; therefore developing, understanding and modifying them were difficult.

Later programs were written in more human-readable *assembly language*. Assembly language belongs to the *second generation* of programming languages. Each assembly language instruction directly maps into a machine language instruction. Assembly language programs were automatically translated into machine language by a program called an *assembler*. Writing and understanding assembly language programs were easier than machine language programs. However even the assembly language programs tend to be lengthy and tedious to write. Programs developed in assembly language runs only on a specific type of computer hardware. Further, programmers were required to have a sound knowledge about *computer architecture*².

In the *third generation* (also referred as 3GL) more human-readable *high-level* languages were introduced. They are closer to English than the previous languages. These languages allowed programmers to ignore the details of the hardware. The programs written using these languages were *portable*³ to more than one type of computer hardware. A compiler or an interpreter was used to translate the high-level code to machine code.

² Computer architecture describes the internal organization of a computer, the instruction set, instruction format, use of registers, memory management, communicating with I/O devices, etc.

³ Portable programs can be compiled and executed on machines with different hardware configurations.

Languages such as FORTRAN, COBOL, C and Pascal belong to the third generation. Source code of the programs written in these languages is much shorter than programs written in 2GL languages (i.e. a single high-level language instruction maps into multiple machine language instructions). *Fourth generation* (4GL) languages allow users to specify what they require without having to specify each step of how it should be done. Software development using 4GL was expected to be easier and faster than with languages of previous generations. Python, Perl and Ruby are considered by many as 4GLs. Another example is SQL (Structured Query Language) commonly used to access databases. However, programs developed in 4GL may not utilize resources optimally (i.e., may consume large amount of processing power and memory). They are generally slower than programs in languages of previous generations. Support for development of Graphical User Interfaces (GUIs) and report generation are considered as a 4GL characteristic.

Under CS1032, our objective is for students to learn fundamentals of programming. Our approach to achieve that in this document is to first give a brief introduction to the C language (in Section 1.4) and then to cover programming in the Python language in detail Chapter 2 onwards.

1.4 The C Programming Language

“C” is one of the most widely used languages in the world. C was introduced by Dennis Ritchie in 1972. Since C was developed along with the UNIX operating system it is strongly associated with UNIX.

Over a few decades, C has gained wide-spread use among programmers. Languages such as C++, Java and C# have branched away from C by adding object-orientation and GUI features. Today C compilers are available for a number of operating systems including all flavours of UNIX, Linux, MS-DOS, Windows and Apple Mac.

With a rich set of library functions, C can be used to write any complex program. C is well suited to develop commercial applications, complex engineering/scientific applications and system software since it incorporates features of both high-level languages and assembly level languages. C programmers have power and flexibility to make their C programs fast and efficient with respect to resource usage. C programs are fairly portable; i.e., with little or no modification and compiling, they can be executed on different hardware. The syntax and coding style of C is simple and well structured. Due to this reason most of the modern languages such as C++, Java and C# inherit C coding style.

The power of C comes at the cost of it being strongly-typed, having a strict syntax and requiring the programmer to manage dynamic memory (relative to a language like Perl or Python). As a result, learning the C language as well as developing a C program for a computational problem can take more time than with a language like Perl or Python. Further, if we compare a C program and a Python program developed to solve the same problem, the C program will generally be much longer (i.e., will have more number of lines) than the Python program.

1.4.1 Steps in Developing a Program in C

A programmer uses a text editor to create and modify files containing the C source code. A file containing source code is called a *source file* (C source files are given the extension .c). After a C source file has been created, the programmer must invoke the C compiler to compile the source code. If the compiler finds no errors in the source code it produces a file containing the machine code (this file is referred to as the *executable file*).

The translation of C source code into executable machine code is a three-stage process; preprocessing, compiling and linking as shown in Figure 1.5.

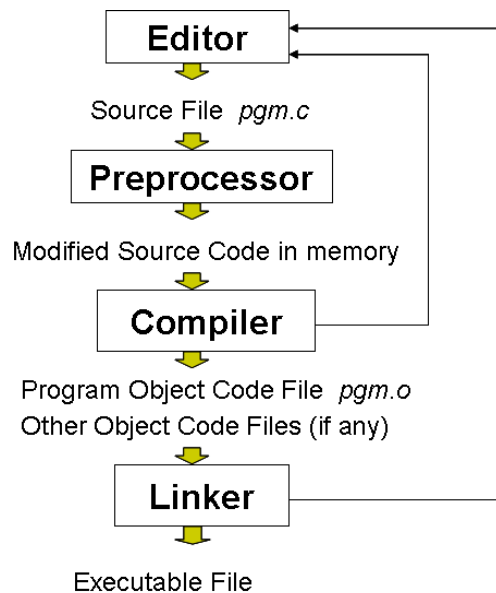


Figure 1.5: Translation of a C program

Preprocessing is performed by a component in the compiler called the *preprocessor*. It modifies the source code (in memory) according to preprocessor directives (example: `#define`) embedded in the source code. It also strips comments and unnecessary white spaces from the source code. The preprocessor does not modify the source code stored on disk, everything is done on the copy loaded into the memory.

Compilation really happens then on the code produced by the preprocessor. The *compiler* translates the preprocessor-modified source code into object code (machine code). While trying to do so it may encounter syntax errors. If such errors are found, they will be notified to the programmer and compilation will stop. If the compiler finds any non-standard codes or conditions which are suspicious but legitimate those will be notified to the programmer as warnings and compilation will continue. A correct program should not have any compilation errors or warnings.

Linking is the final step and the *linker* combines the program object code with other required object codes (e.g., run-time libraries, other libraries, or object files that the programmer has created) to produce the executable file. Finally, it saves the executable code as a file on the disk. If any linker errors are encountered the executable file will not be generated.

1.4.2 An Example C Program

The following code is a C program which displays the message “Hello, World!” on the screen.

```

/* Program-1.1 - My First C Program */
#include <stdio.h>
int main()
{
    printf("Hello, World!");
    return 0;
}

```

The heart of this program is the function `printf`, which actually does the work. A C program is built from functions like `printf` that execute different tasks. The functions must be used in a block of lines which starts with the function name `main()`. The block of lines in the `main()` function is marked by braces `{}`. The `main()` is a special function that is required in every C program.

The keyword `int` before the function `main()` indicates that `main()` function *returns* an integer value. Most functions return some value and sometimes this return value indicates the success or the failure of a function. Because function `main()` returns an integer value, there must be a statement that indicates what this value is. The statement “`return 0;`” does that; in this case it returns zero (conventionally 0 is returned to indicate success and a non-zero value an error or failure).

A line starting with characters `/*` and ending with characters `*/` is a *comment*; e.g., the first line in the above program. Comments are used by programmers to add remarks and explanations within the program. It is always a good practice to comment your code whenever possible. Comments are useful for program maintenance. Most programs need to be modified after some time. In such cases comments will make a programmer's life easy in understanding the source code and the reasons for writing them. Compiler ignores all the comments and they do not have any effect on the executable program. Comments are meant to be read by humans.

Comments are of two types; *single line* comments and *block* comments. Single line comments start with two slashes `/**` and all the text until the end of the line is considered a comment. Block comments start with characters `/*` and end with characters `*/`. Any text between those characters is considered a block of comments.

Lines that start with a pound (`#`) symbol are called *directives* for the preprocessor. A directive is not a part of the actual program; it is used as a command to the preprocessor to direct the translation of the program. The directive `#include <stdio.h>` appears in all programs as it refers to the *standard input output header* file (`stdio.h`). Here, the header file `stdio.h` includes information about the `printf()` function. When using more than one directive, each must appear on a separate line.

A header file includes data types, macros, function prototypes, inline functions and other common declarations. A header file does not include any implementation of the functions declared. The C preprocessor is used to insert the function definitions into the source files. The actual library file which contains the function implementation is linked at link time. There are prewritten libraries of functions such as `printf()` to help us. Some of these functions are very complex and long. Use of prewritten functions makes the programmers life easier and they also allow faster and error free development (since they are used and tested by many programmers).

The function `printf()` is embedded into a statement whose end is marked by a semicolon (`;`). Semicolon indicates the end of a statement to the compiler.

The C language is case sensitive and all C programs are generally written in lowercase letters. Some of the special words may be written in uppercase letters.

1.4.3 Compiling and Running a C Program

Use a text editor in Linux and type the above program. Save it as `HelloWorld.c` (all C source files are saved with the extension `.c`). Then use the shell and go to the directory where you have saved your source file. Use the command `gcc HelloWorld.c` to compile your program. If there are no errors, the code will be compiled, and an executable file will be created in the same directory with the default file name `a.out`. To execute this, at the prompt type `./a.out` (i.e., to execute the `a.out` file in current directory). When you execute your program, it should display "Hello World!" on the screen, as shown below.

```
$ gcc HelloWorld.c
$ ./a.out
Hello World!$
```

However, you will see the shell prompt (`$`) appearing just after the message "Hello World!" You can avoid this by adding the *new line* character (`"\n"`) to the end of the message. Then your modified program should look like the following:

```
/* Program-1.1 - My First C Program, version 2 */
#include <stdio.h>
int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

1.4.4 C Keywords and Tokens

The smallest element identified by the compiler in a source file is called a *token*. It may be a single character or a sequence of characters that form a single unit. Tokens can be classified as *keywords*, *literals*, *identifiers*,

operators, etc. Literals can be further classified as numeric literals, character literals and string literals (or numeric constants, character constants and string constants).

Language specific tokens used by a programming language are called *keywords*. Keywords are also called as *reserved words*. They are defined as a part of the programming language, they have special meaning in a program and therefore cannot be used arbitrarily, say as a name of a variable or a function. Any user defined literals or identifiers should not conflict with keywords or compiler directives. Table 1.2 lists keywords in the C language.

Table 1.2: The C language keywords

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Literals are factual data represented in a language. Numeric constants are an uninterrupted sequence of digits (possibly containing a period). Examples: 123, 10000 and 99.99. Character constants represents a single character and it is surrounded by single quotation marks (‘ and ’). Examples: ‘a’, ‘A’, ‘\$’ and ‘4’. A sequence of characters surrounded by double quotation marks (“ and ”) is called a *string* constant. Examples: “4”, “I like ice cream.”.

Identifiers are also referred to as names. A user defined valid identifier is composed of a letter followed by a sequence of letters, digits or underscore (_) symbols. An identifier must begin with a letter or underscore and the rest can be letters, digits or underscores. Keywords cannot be used as identifiers. Identifiers are case sensitive; therefore, the identifier `abc` is different from `ABC` or `Abc`.

C identifiers can be very long and so it allows descriptive names like “number_of_students”. While defining identifiers programmers should follow a naming standard or convention for better readability of the program. One such standard is the use of underscore symbol (_) to combine two words (example: `sub_total`).

Operators are used with operands to build expressions. For example, “4+5” is an expression containing two operands (4 and 5) and one operator (+ symbol). C supports a large number of mathematical and logical operators such as +, -, *, /, %, ^, &, &&, |, ||,...

The C language also uses several other characters such as semicolon (;), colon (:), comma (,), brackets ([]), braces ({}), and parentheses (()).

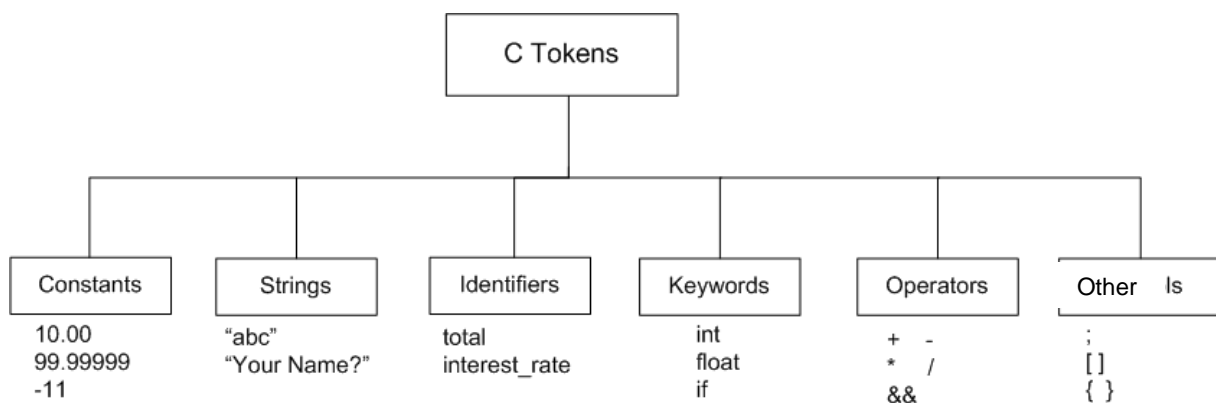


Figure 1.6: C Tokens

2 - Introduction to Python

2.1 Overview of Python

The Python programming language was developed in the late 1980s by Dutch programmer Guido van Rossum. The language was not named after the large snake species but rather after the BBC comedy series *Monty Python's Flying Circus*. Guido van Rossum happens to be a fan. Just like the Linux OS, Python eventually became an open source software development project. However, Guido van Rossum still has a central role in deciding how the language is going to evolve. To cement that role, he has been given the title of “*Benevolent Dictator for Life*” by the Python community.

Python is a versatile and easy-to-use language that was specifically designed to make programs readable and easy to develop. Python also has a rich library making it possible to build sophisticated applications using relatively simple-looking, small amount of code. For these reasons, Python has become a popular application development language.

Important Note: This document is based on Python version 3.6.

2.2 How to Use Python

We use the Python language by working with the Python interpreter. Two common ways the Python interpreter can be used are the *interactive mode* and the *scripting mode*, which are described below.

2.2.1 Interactive Mode

We first invoke (start) the Python interpreter and then work with it interactively. That is, we give the interpreter Python commands, one at a time. To start the Python interpreter in interactive mode, type the command `python` on the command-line, at the shell prompt on Linux, as shown below.

```
$ python
Python 3.6.6 (default, Aug 12 2018, 20:37:26)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

When Python commands are read from the keyboard, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (`>>>`); for continuation lines it prompts with the secondary prompt, by default three dots (...). The interpreter prints a welcome message stating its version number and a copyright notice before printing the primary prompt.

After that, you can type commands and statements at the primary prompt. Some examples:

```
>>> 1 + 5
6
>>> 2 * 5
10
>>> print("Hello World!\n")
Hello World!

>>>
```

The interactive mode makes it easy to experiment with features of the language, or to test functions during program development. It is also a handy desk calculator. We will use the interactive mode to introduce and demonstrate Python language features in this document, when it is more appropriate.

2.2.2 Scripting Mode

The scripting mode is also called the "normal mode" (or the "programming mode") and is non-interactive; in this we give the Python interpreter a text file containing a *Python program* (also called a *Python script*) as input, on the command-line, as follows:

```
$ python myprog.py
```

Here “`myprog.py`” is the name of the text file that contains the Python program. Python source files are given the filename extension “`.py`”.

A Python program can consist of several lines of Python code. A programmer uses a text editor to create and

modify files containing the Python source code. We will use this scripting mode more towards the later parts of this document and use the interactive mode more in the initial sections to discuss the basics.

2.3 Basic Data Types in Python

Let us see a few examples in the interactive mode to gain ideas about basic data types in Python.

```
>>> 2 * 5
10
>>> 1 / 2
0.5
```

There are three basic numeric types in Python: plain integers with unlimited precision (`int`), floating point numbers (`float`), and complex numbers (`complex`). In addition, Booleans (`bool`) are a subtype of integers. '`int`' objects are integers. '`float`' objects are floating-point values ("real", non-integer numbers that can have a fractional part).

Unlike in previous versions of Python and other programming languages, division of an integer by an integer using '/' will result in a "float" value. For example, `4/2` will return `2.0`.

```
>>> 4 / 2
2.0
>>> 4.0/2
2.0
>>> 7. /2
3.5
>>> 'aaa'
'aaa'
>>> len('aaa')
3
>>> len('aaa') + len("tttt")
7
>>> len('aaa') + len("tttt") + 1
8
>>> 'aaa' + "tttt"
'aaatttt'
>>> 'aaa' + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    'aaa'+5
TypeError: must be str, not int
```

In Python, '`str`' objects are character strings, which are normally enclosed within either single quotes or double quotes. That is, '`aaa`' and "`aaa`" are the same. A string can also be triple quoted, either with three single quotes, as '`' 'aaa' '`', or three double quotes, as `"""aaa"""`.

The error message in the last line of the interactive session above says that a string and an integer cannot be joined.

You can check the type of values using the *built-in function* `type()`:

```
>>> type(1)
<class 'int'>
>>> type('1')
<class 'str'>
>>> type(1.0)
<class 'float'>
```

The Python interpreter has several functions built into it, such as `len()` and `type()` above, that are always available.

2.4 Python Objects

So far, we have seen values (integers, floats and strings). We can associate a name to a value and access the value through the associated name:

```
>>> a = 3
>>> a
3
```

Here we have assigned (associated) the *name* "a" to an integer *object* whose value is 3. An object name is also referred to as a *variable*. In the second line above, the interpreter displays the value (3) of the object named "a". Here is another example:

```
>>> myVar = 'one sentence'
>>> myVar
'one sentence'
```

Objects are Python's abstraction for data. Data in a program are represented by objects or as relations between objects.

Every object has the following attributes:

- *An identity*: This means *an address in memory*. This never changes within a single execution. We can check the identity of an object with the built-in function `id()`. Note that a single object may have more than one name (i.e., it may be associated with more than one variable); for e.g., if both the variables "a" and "b" refer to the same object, then the identity of "a" and "b" must be equal. Alternatively, we can use the "**is**" operator to check if both "a" and "b" are the same (i.e., whether they both refer to the same object).
- *A type*: Use the built-in function `type()` to check this. Possible operations and values depend on the type of an object.
- *A value*: The value of some object types can change (*mutable*) whereas other types cannot (*immutable*). We will further discuss this later.

Here is an example.

```
>>> b = 57
>>> c = b
>>> c
57
>>> id(b)
1661170464
>>> id(c)
1661170464
>>> b is c
True
```

We cannot use arbitrary strings as object names (variables). The Python *variable naming rules* are:

- Must begin with a letter (a - z, A - Z) or underscore (_).
- Other characters can be letters, numbers or _ only.
- Names are case sensitive.
- Reserved words cannot be used as a variable name.

Even when a variable name is "legal" (i.e., follows the rules), it might not be a "good" name. Here are some generally accepted conventions for designing good names:

- A name should be meaningful: "price" is better than "p".
- For a multiple-word name, use either the underscore as the delimiter (e.g., `temp_var` and `interest_rate`) or use camelCase capitalization (e.g., `tempVar`, `TempVar`, `interestRate` or `InterestRate`); pick one style and use it consistently throughout your program.
- Shorter meaningful names are better than longer ones.

Can you explain the following?

```
>>> lstring = 'one string'
SyntaxError: invalid syntax
>>> myVar = 'one sentence'
```

```

>>> myvar
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    myvar
NameError: name 'myvar' is not defined
>>> a = 2
>>> a
2
>>> a * 5
10

```

2.5 Mutability of Objects

When the value of an object can change, it is *mutable*; if not, then the object is *immutable*. In Python, mutability depends on the *type*, as follows:

- Immutable types: all numeric types (numbers), strings, tuples, range, bytes, Unicode, frozenset
- Mutable types: lists, dictionaries, bytearray, set, classes, class instances

Here are a few examples with integer objects and string objects.

```

>>> b = 57
>>> c = b
>>> c
57
>>> id(b)
1661170464
>>> id(c)
1661170464
>>> b is c
True
>>> b = 60
>>> b
60
>>> c
57
>>> id(b)
1661170560
>>> id(c)
1661170464
>>> b is c
False

```

```

>>> p = 'abcxyz'
>>> q = p
>>> q
'abcxyz'
>>> p is q
True
>>> id(p), id(q)
(2491577354984, 2491577354984)
>>> q[2]
'c'
>>> q = '01234'
>>> id(p), id(q)
(2491577354984, 2491577354760)
>>> q[2] = 'W'
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in
<module>
    q[2] = 'W'
TypeError: 'str' object does not support
item assignment

```

2.6 Arithmetic and Algebraic Expressions

Arithmetic expressions evaluate to a numeric value, whether of type int or float or other number types that Python supports. We can enter an arithmetic expression, such as $3 + 7$, and hit the Enter key on the keyboard to view the result of evaluating that expression, as seen below:

```

>>> 3 + 7
10
>>> 3 * 2
6
>>> 5 / 2
2.5
>>> 4.0 / 2
2.0

```

In the first two expressions, integers are added or multiplied, and the result is an integer, which is what you expect. In the third expression, an integer is divided by another and the result is a float. In Python version 3, division of an integer by an integer returns the actual value in the form a floating point number. In the last expression, where the float value 4.0 is divided by 2 and the result shown is 2.0.

Values without the decimal point are of type integer, or simply `int`. Values with a decimal point alone (such as `4.`) or with a fractional part (such as `4.32`) are of type floating point, or simply `float`. Let us continue evaluating a few more expressions of both types:

```
>>> 2 * 3 + 1
7
>>> (3 + 1) * 3
12
>>> 4.321 / 3 + 10
11.440333333333333
>>> 4.321 / (3 + 10)
0.3323846153846154
```

Multiple operators are used in these expressions, which raises the question: In what order should the operations be evaluated? The standard algebra precedence rules apply in Python: *multiplication and division take precedence over addition and subtraction* and, just as in algebra, parentheses are used when we want to explicitly specify the order in which operations should take place. If all else fails, expressions are evaluated left- to-right.

All the expressions we have evaluated so far are plain arithmetic expressions involving number values (of type `int` or type `float`), arithmetic operators (such as `+`, `-`, `/`, and `*`), and parentheses. When you hit the Enter key, the Python interpreter will read the expression and evaluate it.

The two most frequently used types of number values, `int` and `float`, have different properties. For example, when two `int` values are added, subtracted, or multiplied, the result is an `int` value. If at least one `float` value appears in such an expression, however, the result is always a `float` value.

The exponentiation operator `**` can be used to compute x^y using the expression `x**y`:

```
>>> 2**3
8
>>> 2**4
16
```

To obtain the *integer quotient* and the *integer remainder* from an integer division operation, operators `//` and `%`, respectively, are used. The operator `//` performs *floor division* (also called *integer division*), which gives the integer quotient of the division; that is, the expression `a//b` returns the integer quotient of the operation where integer `a` is divided by integer `b`. The `%` operator in expression `a%b` returns the remainder of the operation where integer `a` is divided by integer `b`. For example:

```
>>> 14/3
4.666666666666667
>>> 14//3
4
>>> 14%3
2
```

The second `(14//3)` expression evaluates to 4 because 4 is the integer part (quotient) of the division. In the third expression, `14 % 3` evaluates to 2 because 2 is the remainder when 14 is divided by 3.

The Python interpreter has several functions built into it that are always available. For example, the Python function `abs()` can be used to compute the absolute value of a number.

```
>>> abs(-4)
4
>>> abs(4)
4
>>> abs(-3.2)
3.2
```

Other built-in functions available in Python include `min()` and `max()`, which return the minimum or maximum, respectively, of the input values, as seen below.

```
>>> min(6, -2)
-2
>>> max(6, -2)
```

```

6
>>> min(2, -4, 6, -2)
-4
>>> max(12, 26.5, 3.5)
26.5

```

Practice Problem 2.1

Write Python expressions corresponding to the following statements:

- The sum of the first 5 positive integers
 - The average age of Geetha (age 23), Kasun (age 19), and Fathima (age 31)
 - The number of times 73 goes into 403
 - The remainder when 403 is divided by 73
 - 2 to the 10th power
 - The absolute value of difference between the heights of Geetha (54 inches) and Kasun (57 inches)
 - The lowest price among the following prices: Rs. 34.99, Rs. 29.95, and Rs. 31.50
-

2.7 Boolean Expressions and Operators

Expressions other than arithmetic expressions are also common. For example, the expression $2 < 3$ does not evaluate to a number; it evaluates to either *True* or *False* (*True* in this case), which are Boolean values. (*True* and *False* are Python built-in constants of `bool` type).

Comparison operators (such as $<$ or $>$) are commonly used in Boolean expressions. For example:

```

>>> 2 < 3
True
>>> 3 < 2
False
>>> type(True)
<class 'bool'>
>>> 5 - 1 > 2 + 1
True

```

The last expression illustrates that arithmetic/algebraic expressions on either side of a comparison operator are evaluated before the comparison is made. As we will see later, *arithmetic operators take precedence over comparison operators*. For example, in $5 - 1 > 2 + 1$, the operations $-$ and $+$ are performed first, and then the comparison is made between the resulting values.

In order to check equality between values, the comparison operator `==` is used. Note that this operator has two `=` symbols, not one. Then, `!=` is the not equal operator. Here are some examples:

```

>>> 3 == 3
True
>>> 3 + 5 == 4 + 4
True
>>> 3 == 5 - 3
False

```

There are a few other logical comparison operators:

```

>>> 3 <= 4
True
>>> 3 >= 4
False
>>> 3 != 4
True

```

The Boolean expression $3 \leq 4$ uses the `<=` operator to test whether the expression on the left (3) is less than or equal to the expression of the right (4). The Boolean expression evaluates to *True*. The `>=` operator is used

to test whether the operand on the left is greater than or equal to the operand on the right. The expression `3 != 4` uses the `!=` (not equal) operator to test whether the expressions on the left and right evaluate to different values.

Practice Problem 2.2

Translate the following statements into Python Boolean expressions and evaluate them:

- (a). The sum of 2 and 2 is less than 4.
- (b). The value of `7 // 3` is equal to `1 + 1`.
- (c). The sum of 3 squared and 4 squared is equal to 25.
- (d). The sum of 2, 4, and 6 is greater than 12.
- (e). 1, 387 is divisible by 19.
- (f). 31 is even.
- (g). The lowest price among Rs. 34.99, Rs. 29.95, and Rs.31.50 is less than Rs.30.00.

Boolean expressions can be combined using Boolean operators `and`, `or`, and `not` to form larger Boolean expressions. The `and` operator applied to two Boolean expressions will evaluate to `True` if both expressions evaluate to `True`; if either expression evaluates to `False`, then it will evaluate to `False`:

```
>>> 2 < 3 and 4 > 5
False
>>> 2 < 3 and True
True
```

Both expressions illustrate that comparison operators are evaluated before Boolean operators. This is because comparison operators take precedence over Boolean operators, as we will see later.

The `or` operator applied to two Boolean expressions evaluates to `False` only when both expressions are false. If either one is true or if both are true, then it evaluates to `True`.

```
>>> 3 < 4 or 4 < 3
True
>>> 3 < 2 or 2 < 1
False
```

The `not` operator is a unary Boolean operator, which means that it is applied to a single Boolean expression. It evaluates to `False` if the expression is true or to `True` if the expression is false.

```
>>> not (3 < 4)
False
```

2.8 Variables and Assignments

As we already know, it is useful to assign names to objects, and we call those names *variables*. For example, the object of integer type with value 3 may be assigned to variable `x` as `x = 3`. The variable `x` can be thought of as a name that enables us to retrieve this object later. In order to retrieve it, we just need to evaluate `x` in an expression.

```
>>> x = 4
```

The statement `x = 4` is called an *assignment statement*. The general format of an assignment statement is:

```
<variable> = <expression>
```

An expression we refer to as `<expression>` lies on the right-hand side of the assignment operator `=`; it can be an algebraic, Boolean, or other kind of expression. On the left-hand side is a variable referred to as `<variable>`. The assignment statement assigns to the `<variable>` the value that `<expression>` evaluates to. In the last example, `x` is assigned value 4. With the assignment statement, we also *define* the variable.

A defined variable (i.e., a variable with a value assigned), can be used in a Python expression:

```
>>> x
4
```

When Python evaluates an expression containing a variable, it will evaluate the variable to its assigned value and then perform the operations in the expression:

```
>>> 4 * x
16
```

An expression involving variables may appear on the right side of an assignment statement, such as:

```
>>> counter = 4 * x
```

In statement `counter = 4 * x`, `x` is first evaluated to 4, then the expression `4 * 4` is evaluated to 16, and then 16 gets assigned to variable `counter`:

```
>>> counter
16
```

So far, we have defined two variable names: `x` with value 4 and `counter` with value 16. What about, say, the value of variable `z` that has not yet been defined (not assigned a value)? Let's see:

```
>>> z
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    z
NameError: name 'z' is not defined
```

We get an error message, saying that there is a “NameError” because `z` is not defined.

We can use the “`del`” statement to delete a defined (an existing) variable, which will remove (undefine) the variable from the system. Obviously, an undefined variable cannot be deleted.

```
>>> del x
>>> x
Traceback (most recent call last):
  File "<pyshell#152>", line 1, in <module>
    x
NameError: name 'x' is not defined
```

In summary, if a variable has either not been assigned a value or been deleted, it does not exist. When Python tries to evaluate an undefined name, a “NameError” occurs.

Practice Problem 2.3

Write Python statements for the following actions and execute them:

- (a). Assign integer value 3 to variable `a`.
- (b). Assign 4 to variable `b`.
- (c). Assign to variable `c` the value of expression `a * a + b * b`.

2.9 Python Programs

We explained in Section 2.2 on how to use a Python program with the Python interpreter in the *scripting mode*. A Python program is a sequence of Python statements, spread over several lines. Let us try our first Python program. Type the following exactly as shown using a text editor and save the file as `prog-2-1.py`.

```
#!/usr/bin/python
# Program-2.1
msg="Let's learn Python"
print("Hello, World!")
print(msg)
```

Next run the Python interpreter with the file as input and observe the output.


```
$ python prog-2-1.py
Hello, World!
Let's learn Python
```

2.10 Input, Output and Files

Here we briefly introduce how to perform input, output and file operations in Python. A more detailed discussion is in Chapter 10.

The simplest way to produce output is using the `print()` built-in function, as we have seen already. This converts the expressions you pass into a string and writes the result to standard output as follows:

```
>>> print("Let's learn Python")
Let's learn Python
```

Python provides the built-in function `input()` to read a line of text from standard input, which by default comes from the keyboard.

2.10.1 The `input()` Function

In the `input([prompt])` built-in function, if the optional `prompt` argument is present, it is written to standard output (screen) without a trailing newline. The function then reads a line from standard input (keyboard), converts it to a string (stripping a trailing newline), and returns that. In Python 2.7 the `raw_input([prompt])` function does the same thing.

```
#!/usr/bin/python
# Program-2.2
string = input("Enter your input: ")
print("Received input is:", string)
```

This would prompt you to enter any string and it would display same string on the screen. When we type "Hello Python!", its output is like this:

```
Enter your input: Hello Python
Received input is: Hello Python
```

The `eval([prompt])` function can evaluate valid Python expressions given through `input([prompt])`. Here is a program that uses it.

```
#!/usr/bin/python
# Program-2.3
string = input("Enter your input: ")
print("Received input is:", eval(string))
```

This would produce the following result against the entered input:

```
Enter your input: [x*5 for x in range(2,10,2)]
Received input is: [10, 20, 30, 40]
```

2.10.2 Reading Input from a File

The following program will open the file named as "pythonlearn.txt" and print its content on the screen (i.e., the standard output). For this program to work, the file "pythonlearn.txt" must exist in the same directory with the Python program file.

```
#!/usr/bin/python
# Program-2.4
# Open a file
fo = open("pythonlearn.txt", "r")
string = fo.read();
print("Read String is: ", string)
# Close opened file
fo.close()
```

2.10.3 Writing to a File

The following program will write the given sentence to the “firstwrite.txt” file.

```
#!/usr/bin/python
# Program-2.5
# Open a file
fo = open("firstwrite.txt", "w")
fo.write("Python is a great language.\nYeah its great!!\n");
# Close opened file
fo.close()
```

2.11 Python Keywords

Shown below are the keywords (reserved words) of the Python language.

and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield
await	else	import	pass	False
break	except	in	raise	True
class	finally	is	return	None

The above should not be used arbitrarily in programs, e.g., as names of variables. There are also several words that have special meanings in Python, for e.g., True, False, None.

3 - Operators and Expressions in Python

3.1 Types of Operators

In the expression $4 + 5$, 4 and 5 are operands and + is the operator. Python language supports the following types of operators.

- Arithmetic operators
- Comparison (i.e., relational) operators
- Assignment operators
- Bitwise operators
- Logical operators
- Membership operators
- Identity operators

Let's look at these types one by one.

3.2 Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	$a + b$ will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	$a - b$ will give -10
*	Multiplication - Multiplies values on either side of the operator	$a * b$ will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2.0
%	Modulus - Divides left hand operand by right hand operand and returns remainder	$b \% a$ will give 0
**	Exponent - Performs exponential (power) calculation on operators	$a ** b$ will give 10 to the power 20
//	Floor (or integer) division - Division such that the fractional part of the result is removed, and only the integer part remains.	$9 // 2$ is equal to 4 and $9.0 // 2.0$ is equal to 4.0

3.3 Python Comparison Operators

Operator	Description	Example
==	Checks if the value of two operands are equal; if yes then condition becomes true.	$(a == b)$ is not true.
!=	Checks if the value of two operands are not equal; if values are not equal then condition becomes true.	$(a != b)$ is true.
<>	Checks if the value of two operands are not equal; if values are not equal then condition becomes true. This is similar to the != operator.	$(a < > b)$ is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	$(a > b)$ is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	$(a < b)$ is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	$(a >= b)$ is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	$(a <= b)$ is true.

3.4 Python Assignment Operators

Operator	Description	Example
=	Simple assignment operator, assigns values from right side operands to left side operand	c = a + b will assign value of a + b into c
+=	Add AND assignment operator, it adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, it subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, it multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, it divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, it takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	Floor Division and assigns a value, performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

3.5 Python Bitwise Operators

Bitwise operators work on bits; that is, they perform a bit-by-bit operation on operands. Assume a = 60 and b = 13. The following show their binary representation and 4 bitwise operations on them.

```

a  = 0011  1100
b  = 0000  1101
-----
a&b = 0000  1100
a|b = 0011  1101
a^b = 0011  0001
~a  = 1100  0011

```

The following bitwise operators are supported by Python.

Operator	Description	Example
&	Binary AND Operator: sets a bit to 1 in the result if the corresponding bit is 1 in both operands.	(a & b) will give 12 which is 0000 1100
	Binary OR Operator: sets a bit to 1 in the result if the corresponding bit is 1 in either operands.	(a b) will give 61 which is 0011 1101
^	Binary XOR Operator: sets a bit to 1 in the result if the corresponding bit is 1 in exactly one operand.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator: is unary and has the effect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement
<<	Binary Left Shift Operator: The left operand's value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator: The left operand's value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

3.6 Python Logical Operators

The following logical operators are supported. Assume variable a holds 10 and variable b holds 20.

Operator	Description	Example
and	Logical AND operator. If both the operands are true then condition becomes true.	(a and b) is true.
or	Logical OR Operator. If any of the two operands is true (non zero) then condition becomes true.	(a or b) is true.
not	Logical NOT Operator. Reverses the logical state of its operand. If an expression is true then Logical NOT of that is false.	not(a and b) is false.

3.7 Python Membership Operators

Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators explained below:

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

3.8 Python Identity Operators

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

3.9 Operator Precedence in Python

The following table lists all operators we discussed in this Chapter, from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~, +, -	Complement, unary plus and minus (method names for the last two are +@ and -@)
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^,	Bitwise exclusive 'OR' and regular 'OR'
<=, <, >, >=	Comparison operators
<>, ==, !=	Equality operators
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators
is, is not	Identity operators
in, not in	Membership operators
not, or, and	Logical operators

4 - “Selection” Control Structures in Python

4.1 Control Flow

In a program, *control flow* (or *flow of control*) refers to the order in which individual statements of the program are executed. Similarly, control flow in an algorithm is the order in which individual steps of the algorithm are executed.

So far, we have considered *sequential control flow*, i.e., statements getting executed from top to bottom, in the order they appear in the program. The sequential flow of control is the default behavior. However, we often need to alter this flow when we write programs, because the problems we can solve with sequential control flow alone are limited to simple (or, as one might say, trivial) problems. In other words, there are many problems that cannot be solved with the sequential control flow alone.

Many problems that we encounter are complex enough that they require programs with enhanced control flows. For this, most programming languages provide at least three *control structures* for altering the default sequential flow. These control structures are known as *selection*, *loop*, and *subprogram*. Together with the default sequential flow, we have four control structures for specifying the control flow as shown in Figure 4.1.

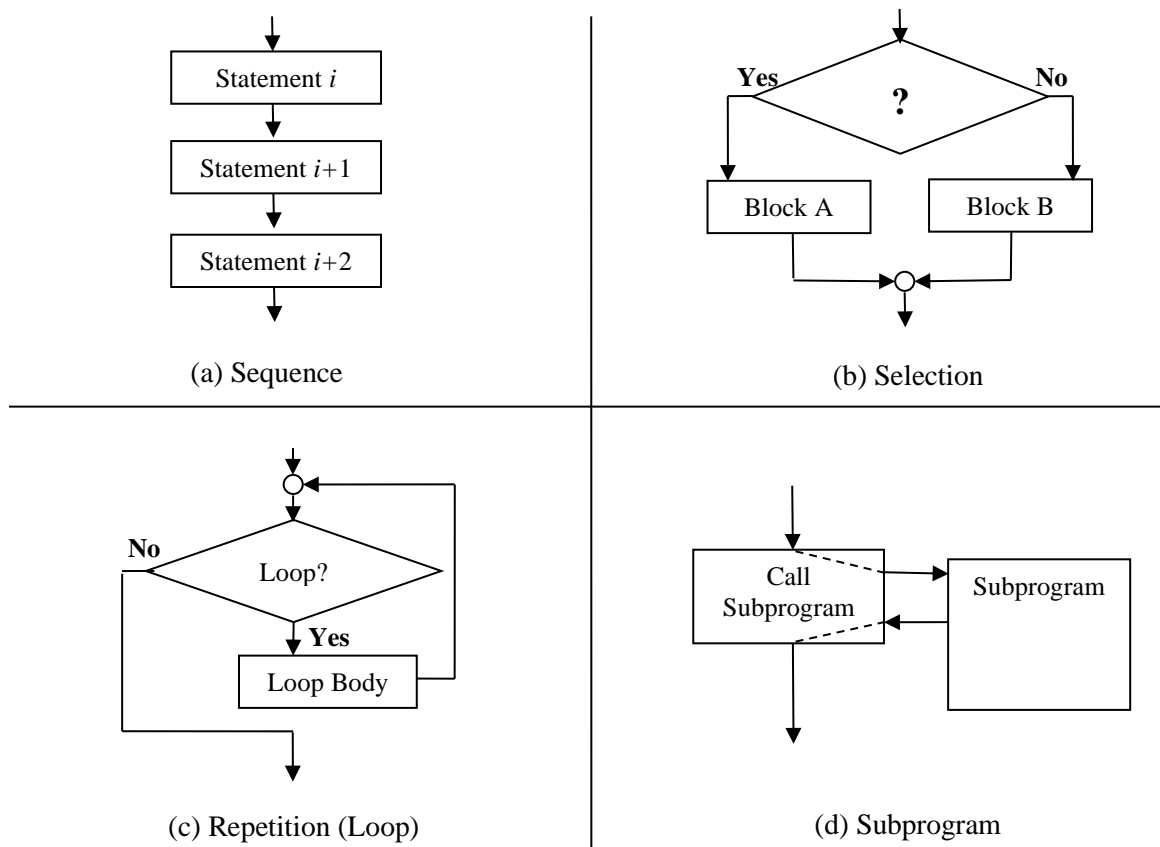


Figure 4.1: Four control structures used in programs

We will discuss the selection control structures in Python in the rest of this Chapter and the repetition (loop) and subprogram control structures in later Chapters.

When using the selection control structure, we specify one or more conditions to be evaluated (or tested) by the program; this is shown by the decision element with a question mark (“?”) in Figure 4.1(b). Then we specify one block of statements (Block A) to be selected for execution if the condition is determined to be true, and another block of statements (Block B) to be selected for execution if the condition is determined to be false. Note that only one of the blocks, either A or B, will get executed. Further, each of the blocks A and

B can be empty, a single statement or a complex program segment containing a combination of all four control structures.

4.2 “if” Structure

The `if` structure in Python is similar to that of other languages. It contains an expression followed by a set of statements to be executed if the expression is evaluated as true.

Syntax: The syntax of an `if` structure in Python:

```
if expression:
    statement_1
    statement_2
    ...
    statement_n
```

It starts with the keyword `if`, followed by an expression that represents a condition, followed by a colon (“:”), followed finally by a block of statements (also referred to as the *if-body*). With reference to Figure 4.1(b), the *if-body* corresponds to block A whereas block B is empty.

The *if-body* is *indented*. *Indentation* is Python’s way of grouping statements. In interactive mode, you have to type a tab or space(s) for each indented line. In a Python program, indentation can be done with a text editor; decent text editors have an auto-indent facility. When a *compound statement* like an `if`-structure is entered interactively, it must be followed by a blank line to indicate completion (since the Interpreter cannot guess when you have typed the last line). Note that *each line within the body must be indented by the same amount*.

The *if-body* is executed if the expression (also called a *conditional expression*) is evaluated as “true”. In Python, like in C, any non-zero integer value is true; zero is false. The expression may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false.

Consider the following example:

```
if (marks > 50):
    print("You passed the exam!")
```

If the value of the variable “marks” is greater than 50, the print statement is executed and the string “You passed the exam!” is displayed on the screen; otherwise the print statement is skipped, and nothing is displayed. Program 4.1 illustrates the use of this in a Python program.

```
# Program-4.1
marks = int(input('Enter your marks: '))
if (marks > 50):
    print("You passed the exam")
```

Note that even if the expression “marks > 50” existed without the surrounding parenthesis, it still would be valid Python syntax. We can use parenthesis for grouping of expressions and for clarity.

Executing Program-4.1 with different inputs will get the following results.

Case 1: Enter marks: **73**

You passed the exam!

Case 2: Enter marks: **34**

Case 3: Enter marks: **50**

In the cases 2 and 3, nothing is displayed as the print statement is not executed.

Example 4.1: Develop a program which accepts a number (an amount of money to be paid by a customer in rupees) entered from the keyboard. If the amount is greater than or equal to 1,000 rupees, a 5% discount is given to the customer. Then display the final amount that the customer has to pay.

Let us see how to do this. First the program needs to read the input number and check whether it is greater than or equal to 1000; if it is the case a 5% discount should be given. Then the final amount needs to be displayed. We can compute the discount as the body of an if-structure. Program-4.2 is an *attempt* to implement this.

```
# Program-4.2 (this contains a bug!)
amount = float(input('Enter your amount: '))
if (amount >= 1000):
    discount = amount * 0.05
    final_amount = amount - discount
print('your amount is', '%.2f' % amount)
print('your final amount is', '%.2f' % final_amount)
```

Executing Program-4.2 with 1250.25 as the keyboard input will display the following:

```
Enter your amount: 1250.25
your amount is 1250.25
your final amount is 1187.74
```

Now test the program with a few other inputs. *Detect a bug in the program and correct it.* □

4.3 “if...else” Structure

To implement in Python the selection control structure shown in Figure 4.1(b) with both blocks A and B specified, the `else` keyword can be combined with the `if` keyword. The `else` keyword is followed by the code that gets executed if the `if`-body does not get executed (i.e., conditional expression is not evaluated to true).

The `else` part is optional and there could be at most one `else` part following an `if` part. Further, an `else` part cannot exist alone; it must be paired with an `if` part.

Syntax: The syntax of an `if...else` structure in Python:

```
if expression:
    statement(s)
else:
    statement(s)
```

Note that a colon (“:”) is required after the keyword `else`. It is followed by a block of statements referred to as the *else-body*. With reference to Figure 4.1(b), the `if`-body corresponds to block A and the `else`-body corresponds to block B (or, vice versa). Just like the `if`-body, the `else`-body is also indented in a program.

The `if`-body is executed if the expression is evaluated to true and the `else`-body is executed otherwise.

Example 4.2: Modify the program for the Example 4.1 so that it displays the message “No discount...” if the amount is less than 1,000.

We can use an `else` part for the added requirement. Program-4.3 below implements this. □

```
# Program-4.3
amount = float(input('Enter your amount: '))
final_amount = 0
if (amount >= 1000):
    discount = amount * 0.05
    final_amount = amount - discount
else:
    print('No discount')
```



```

    final_amount = amount
    print('your amount is', '%.2f' % amount)
    print('your final amount is', '%.2f' % final_amount)

```

Example 4.3: A car increases its velocity from $u \text{ ms}^{-1}$ to $v \text{ ms}^{-1}$ within t seconds. Write a program to compute and display the acceleration, assuming u , v and t are integers and given as input.

The relationship between a , u , v and t can be given as $v = u + at$. Therefore, acceleration can be found by the equation $a = \frac{v-u}{t}$. Note that users can input any values to the program. Therefore to find the correct acceleration t has to be non-zero and positive. So our program should make sure it only accepts correct inputs. The following Program-4.4 computes the acceleration given u , v and t . □

```

# Program-4.4
u = int(input("Enter u (m/s):"))
v = int(input("Enter v (m/s):"))
t = int(input("Enter t (s):"))
if t >= 0:
    a = float((v-u))/t
    print("acceleration is: %.2f m/s^2" % a)
else:
    print("Incorrect time")

```

Exercise 4.1: Modify Program-4.1 to display the message “You failed!”, if marks ≤ 50 . □

4.4 Multi-way Selection with the “**elif**” Keyword

The `elif` keyword (meaning “else-if”) allows us to implement multi-way selection, going beyond the two-way selection in the `if-else` structure. This means, we can select one block of code for execution from among many (> 2). For this, we need to specify multiple conditional expressions for truth value and execute a block of code as soon as the corresponding expression evaluates to true.

An `elif` part is optional and there can be an arbitrary number of `elif` parts following an `if` part.

Syntax: The syntax of an `if...elif` structure in Python:

```

if expression_1:
    statement(s)
elif expression_2:
    statement(s)
elif expression_3:
    statement(s)
...
else:
    statement(s)

```

Just like the `if-body` and the `else-body`, an `elif-body` is also indented.

If the “`expression_1`” is evaluated to true, the `if-body` is executed and rest (the corresponding `elif` and `else` parts, if any) will be skipped. Otherwise, the “`expression_2`” is evaluated; if it is true, then the corresponding `elif-body` is executed and rest (the remaining `elif` and `else` parts, if any) will be skipped. Otherwise the “`expression_3`” is evaluated and so on. If none of the expressions is true, the `else-body` is executed.

The `if...elif` structure is a substitute for the “switch-case” structure in some other languages such as C.

4.5 Nested Selection Structures

There may be a situation where we want to test for one or more additional conditions after testing one condition. In such a situation, we can use a nested selection structure, implemented in Python using nested `if...elif...else` constructs. Nested means having one `if...elif...else` construct inside another.

Syntax: The syntax of an example nested `if...elif...else` structure in Python:

```
if expression_1:
    statement(s)
    if expression_2: -----
        statement(s)
    elif expression_3: -----
        statement(s)
    else:
        statement(s) -----
elif expression_4:
    statement(s)
else:
    statement(s)
```

} inner **if...elif...else** structure

Notice how critical the level of indentation in representing nested constructs. In the above example, the inner `if...elif...else` structure is at the same level of indentation with the other code that forms the `if`-body of the outer `if...elif...else` structure.

4.6 Exception Handling

There are two distinguishable kinds of errors that you may have experienced so far: *syntax errors* and *exceptions*. Syntax errors, also known as parsing errors, are due to violation of syntax (grammar) rules of the Python language, as we stated before.

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* (also called *run-time errors* in some contexts). Exceptions can be handled in Python programs. Most exceptions are not handled by programs, however, and result in error messages. Here are three examples:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    10 * (1/0)
ZeroDivisionError: division by zero

>>> 4 + rate*3
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    4 + rate * 3
NameError: name 'rate' is not defined

>>> '3' + 4
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    '3'+ 4
TypeError: must be str, not int
```

The last line of the error message in each example above indicates what happened. Exceptions come in different types, and the *exception type* is printed as part of the message: the types in the three examples are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions. Standard exception names are built-in identifiers (not reserved keywords).

An exception is an event that disrupts the normal flow of the program execution. In general, when Python encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.

Python makes it possible to handle unexpected errors in our Python programs. This is important as a debugging tool and to improve the quality of our programs.

If you have some code segment that may raise an exception, you can make your program to handle (“catch”) it by placing the suspicious code segment in a `try...except` construct. This construct has a **try:** clause followed by one or more **except:** clauses. Each `except` clause has a block of code which handles one exception as elegantly as possible. The `try...except` construct has an optional **else** clause, which, when present, must follow all `except` clauses. It is useful for code that must be executed if the `try` clause does not raise an exception.

Here is the structure of a simple `try....except...else` construct:

```
try:
    Do some operation that may raise an exception;
    .....
except Exception_1:
    If there is Exception_1, then execute this block.
    .....
except Exception_2:
    If there is Exception_2, then execute this block.
    .....
else:
    If there is no exception then execute this block.
    .....
```

Here is a simple example, which opens a file to write some content in it.

```
#!/usr/bin/python
# Program-4.5
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print("Error: can't open file or write data.")
else:
    print("Finished writing the file successfully.")
    fh.close()
```

In addition to using `except` clauses and an `else` clause in a `try` construct, you can also use the **finally** clause. Note that the `finally` clause is *optional* and intended to define clean-up actions that must be executed under all circumstances. A `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed “on the way out” when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. The `finally` clause is useful for releasing external resources (e.g., files, network connections). The following is an example that demonstrated the use of the `finally` clause.

```
#!/usr/bin/python
# Program-4.6
num1 = int(input("Input num1: "))
num2 = int(input("Input num2: "))
try:
    result = num1 / num2
except ZeroDivisionError:
    print("division by zero!")
else:
    print("result is", result)
finally:
    print("executing finally clause")
```

5 - Loop Control Structures

Python provides the two loop structures: the **for** loop and the **while** loop. We can also have nested loops, i.e., a loop inside a loop. These are discussed in this Chapter.

5.1 The **for** Loop

The **for** loop construct is used to repeat a statement or block of statements a specified number of times. The **for** loop can also iterate over the items of any sequence (a list or a string), in the order that they appear in the sequence.

Syntax: The syntax of the **for** loop is as follows:

```
for iterating_var in sequence:
    statements(s)
```

The block of statements executed repeatedly is called the *loop body*. The loop body is *indented*.

If the sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable `iterating_var` and the loop body is executed. This concludes one iteration of the loop. Next the second iteration of the loop body is executed after the second item is assigned to the iterating variable `iterating_var`. Like this, the loop body is executed repeatedly, with a unique item in the list assigned to `iterating_var` in each iteration, until the entire sequence is exhausted.

Here is an example.

```
#!/usr/bin/python
# Program-5.1
for counter in range(1, 6):
    print("This is a loop: counter = ", counter)
```

Execution of program-5.1 displays the following.

```
This is a loop: counter = 1
This is a loop: counter = 2
This is a loop: counter = 3
This is a loop: counter = 4
This is a loop: counter = 5
```

The `range()` function: If we do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions. Implementation of `range()` is as either `range(stop)` or `range(start, stop[, step])`. See how it is useful in program-5.1. Here are four interactive examples.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, 0,1,2,...,9, i.e., the legal indices for items of a sequence of length 10 (first example). It is possible to let the range start at another number (second example above), or to specify a different increment (even negative), which is called the *step*., with a third argument for the `range()` function, as seen in the last two examples above.

To iterate over the indices of a *list* or *sequence* (which will be covered in the next Chapter) in a **for** loop, you can combine `range()` and `len()` functions as follows:

```
>>> a = ['Mali', 'had', 'a', 'little', 'puppy']
```

```
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mali
1 had
2 a
3 little
4 puppy
```

Exercise 5.1: Write a program to display the integers from 100 to 200.

Example 5.1: Write a program to compute the sum of all even numbers up to 100, inclusive.

For this problem, the first even number is 0 and the last even number is 100. By adding 2 to an even number the next even number can be found. Therefore we can have a loop with a counter incremented by 2 in each iteration. Program-5.2 is an implementation of this. □

```
#!/usr/bin/python
# Program-5.2
sum_total = 0
for counter in range(0,101,2):
    sum_total += counter
print('total = ', sum_total)
```

Exercise 5.2 : Modify Program-5.2 such that it computes the sum of all the odd numbers up to 100.

Exercise 5.3 : Write a program to compute the sum of all integers form 1 to 100.

Exercise 5.4 : Write a program to compute the factorial of any given positive integer < 100.

Example 5.2 : Write a program to compute the sum of all integers between any given two numbers.

In this program both inputs should be given from the keyboard. Therefore at the time of program development both initial value and the final value are not known. □

```
#!/usr/bin/python
# Program-5.3
num1 = int(input("Enter First Number:")) #read Number1
num2 = int(input("Enter Second Number:")) #read Number2
sum = 0
for N in range( num1,num2+1) :
    sum = sum + N
print("The sum is:",sum)
```

Exercise 5.5 : Can the loop body of a for loop never get executed? Explain using program-5.3.

5.2 The while Loop

A while loop in Python repeatedly executes the loop body as long as a given condition is true. The condition is specified by an expression.

Syntax: The syntax of a while loop is:

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

The block of statements executed repeatedly is the *loop body*, which is *indented*, as in the `for` loop.

The condition to execute the loop body is considered true if the expression is true or it is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop body.

Note that, in Python, all statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Note that the *while* loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Consider the following example:

```
#!/usr/bin/python
# Program-5.4
num = int(input("Enter Number of time to repeat: "))
while (num != 0) :
    print("Hello World!")
    num = num - 1
```

Execution of Program-5.4 with 5 as the input will display the following.

```
Enter number of times to repeat: 5
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

5.3 Using *else* with Loops

Python allows to use an *else* clause with loops, as follows.

- If the *else* clause is used with a *for* loop, the *else* clause is executed when the loop has exhausted iterating the list.
- If the *else* clause is used with a *while* loop, the *else* clause is executed when the loop condition becomes false.

Note however that the above does not happen when the loop is terminated by a *break* statement (to be discussed later). That is, a loop's *else* clause is executed when no *break* occurs.

The following example illustrates the combination of an *else* clause with a *while* loop that prints a number as long as it is less than 5. When the loop condition becomes false, the *else* clause gets executed.

```
#!/usr/bin/python
# Program-5.5
count = 0
while count < 5:
    print(count, " is less than 5")
    count = count + 1
else:
    print(count, " is not less than 5")
```

When the above code is executed, it produces the following output.

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
```

5 is not less than 5

5.4 Nested Loops

We can use one loop inside another loop. We will show a few examples to illustrate the concept.

Syntax: The syntax for a **nested for-while loop** combination is as follows:

```
for iterating_var in sequence:
    while expression:
        statement(s)
    statements(s)
```

Syntax: The syntax for a **nested while-for loop** combination is as follows:

```
while expression:
    for iterating_var in sequence:
        statements(s)
    statement(s)
```

Note that you can put any type of loop inside any other type of loop. For example a `for` loop can be inside a `for` loop.

Example 5.2 : The following program uses a nested loop to find the prime numbers from 2 to 100.

```
#!/usr/bin/python
# Program-5.6
i = 2
while (i < 100):
    j = 2
    while (j <= (i/j)):
        if not(i%j): break
        j = j + 1
    if (j > i/j) : print(i, " is prime")
    i = i + 1
print("Good bye!")
```

Example 5.3 : Write a Python program to display the following pattern:

```
$$$$$$$$$$
$$$$$$$$$$
$$$$$$$$$$
$$$$$$$$$$
$$$$$$$$$$
```

There are 10 “\$” symbols in a single row (10 columns) and there are 5 rows. This can be implemented by having two loops, one nested inside another. The inner loop will print 10 individual “\$” symbols within a row, on a line. The outer loop can advance to the next line. Program-5.7 displays the above symbol pattern. To print the value of the next iteration in the same row, we have to initialize `end=""` where we do not need a space between printed strings or `end=' '` where we need a space between printed elements.

```
#!/usr/bin/python
# Program-5.7
for x in range (0, 5):
    for y in range (0, 10):
```

```

        print('$',end='')
print('\n',end='')

```

Exercise 5.5 – Write a Python program using two loops (nested) to display the following pattern:

```

*
**
***
****
*****
*****

```

5.5 The *break* Keyword

The `break` keyword is used inside a loop and is used for terminating the current iteration of the loop body immediately; i.e., to break out of the smallest enclosing `for` or `while` loop. The control will be transferred to the first statement following the loop body. If you are inside the inner loop of a nested loop, then the `break` statement inside that inner loop transfers the control to the immediate outer loop. The `break` statement can be used to terminate an infinite loop or to force a loop to end before its normal termination.

Consider the following example.

```

#!/usr/bin/python
# Program-5.8
n = 10;
for var in range(0, n):
    print("Hello World!")
    if (var == 5):
        print("Countdown Aborted")
        break;

```

Under normal circumstances (without the `break`) the Program-5.8 will display the “Hello World!” 10 times. During the first 5 iterations of the `for` loop, the program executes normally displaying the string “Hello World!”. Then at the beginning of the sixth iteration variable “`var`” becomes “5”. Therefore the `if` clause which evaluates whether “`var == 5`” becomes “`true`”; as a result the “Countdown Aborted” string gets displayed and then the `break` statements is executed. At this point the loop will terminate because of the `break` keyword.

5.6 The *continue* Keyword

The `continue` keyword inside a loop causes the program to skip the rest of the loop body in the current iteration, causing it to continue with the next iteration of the loop.

Here is an example.

```

# Program-5.9
for i in range(-5,6):
    if i == 0 :
        continue
    print("5 divided by ", i, " is: ", (5.0/i))

```

In program-5.9, 5 is divided by all the integers from -5 to +5. However we want to avoid dividing by 0. In Program-5.9, when “`i`” is 0 (when the `if` condition is `TRUE`) the `continue` keyword is used to skip the rest of the loop body in that iteration; this will skip the `print` statement which has the division by 0.

5.7 The *pass* Keyword

The `pass` statement inside a loop is used when a statement is required syntactically but you do not want any command or code to execute. This is mainly used when we have a loop or a function that is not implemented yet. We cannot have an empty body in Python. So we use `pass` statement to construct a body that does nothing.

Here is an example.

```
# Program-5.10
fruits = ['Apple', 'Orange', 'Grapes', 'Pineapple', 'Banana', 'Mango',
'Starfruit']
for fruit in fruits:
    pass
```

6 - Lists

6.1 Sequences

The most basic data structure in Python is the *sequence*. Sequences are *compound* data types, and they are used to group other values together. Each element of a sequence is assigned a number - its position or *index*. The first index is zero, the second index is one, and so forth.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has many built-in functions to be used with sequence types, for e.g., for finding the length of a sequence and for finding its largest and smallest elements.

Python has six built-in types of sequences; the most common one is *lists*, which we will discuss now.

6.2 Python Lists

The list is the most versatile data-type available in Python which can be written as a list of comma-separated values (items) between square brackets. Items in a list need not all have the same type. Creating a list is as simple as listing different comma-separated values between square brackets. For example:

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5]
list3 = ["a", "b", "c", "d", 'pqr', 12.345]
```

List indices start at 0, and lists can be sliced, concatenated and so on.

6.3 Accessing Values in Lists

To access values in lists, use square brackets for slicing along with the index or indices to obtain value available at that index. The following is a simple example:

```
#!/usr/bin/python
# Program-6.1
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5, 6, 7 ]
print("list1[0]: ", list1[0])
print("list2[1:5]: ", list2[1:5])
```

When the above code is executed, it produces the following result:

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

6.4 Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator. You can add to elements in a list with the `append()` function. For example:

```
#!/usr/bin/python
# Program-6.2
list = ['physics', 'chemistry', 1997, 2000];
print("Value at index 2 : ", list[2])
list[2] = 2001;
print("New value at index 2 : ", list[2])
```

When the above code is executed, it produces the following result:

```
Value at index 2 : 1997
New value at index 2 : 2001
```

6.5 Deleting List Elements

To remove a list element, you can use either the `del` statement if you know exactly which element(s) you are deleting. The `remove()` method of a list object can also be used.

The following is an example.

```
#!/usr/bin/python
# Program-6.3
list1 = ['physics', 'chemistry', 1997, 2000]
print(list1)
del list1[2]
print("After deleting value at index 2 : ", list1)
```

When the above code is executed, it produces the following output:

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 : ['physics', 'chemistry', 2000]
```

6.6 Other List Operations

Lists respond to the + and * operators (much like strings), where “+” means concatenation and “*” means repetition, and the result is a new list. In fact, lists respond to all general sequence operations. Table 6.1 shows some example list operations.

Table 6.1 : Some example list operations

Python Expression	Result	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]:	1 2 3	Iteration

6.7 Indexing and Slicing

Because lists are sequences, indexing and slicing work the same way for lists. Assume L=['spam', 'Spam', 'SPAM!'] and then see if you can understand Table 6.2.

Table 6.2 : List indexing and slicing

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Exercise 6.1 - Write a program that can find all the even numbers in a list of numbers and then print them.

Exercise 6.2 - Write a program that will multiply two lists and generate a 3rd list. You need to also print all three lists.

7 - Numbers

7.1 Introduction

Number data types store numeric values. They are *immutable*, which means once created their value never changes. This means, changing the value of a numeric data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example:

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the `del` statement. The syntax of the `del` statement is:

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

You can delete a single object or multiple objects by using the `del` statement. For example:

```
del var
del var_a, var_b
```

Python supports four different numerical types:

- `int` (signed integers): often called just integers or ints, are positive or negative whole numbers with no decimal point. There is no limit for the length of integer literals apart from what can be stored in available memory
- `float` (floating point, real values) : or floats, represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 ($2.5e2 = 2.5 \times 10^2 = 250$).
- `complex` (complex numbers) : are of the form $a + bJ$, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). a is the real part of the number, and b is the imaginary part. Complex numbers are not used much in Python programming.

7.2 Examples

Here are some examples:

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

A complex number consists of an ordered pair of real floating-point numbers denoted by $a + bj$, where a is the real part and b is the imaginary part of the complex number.

7.3 Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, we need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type `int(x)` to convert `x` to a plain integer.
- Type `float(x)` to convert `x` to a floating-point number.
- Type `complex(x)` to convert `x` to a complex number with real part `x` and imaginary part zero.
- Type `complex(x, y)` to convert `x` and `y` to a complex number with real part `x` and imaginary part `y`. `x` and `y` are numeric expressions.

8 - Strings

8.1 Introduction

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes ' ' the same as double quotes " ".

Creating strings is as simple as assigning a value to a variable. For example:

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Strings can be concatenated (glued together) with the + operator, and repeated with *. This is another way to create new strings. For example:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Two string literals next to each other are automatically concatenated. The first line above could also have been written word = 'Help' 'A'; this only works with two literals, not with arbitrary string expressions.

8.2 Accessing Elements and Slicing

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. Individual elements can be accessed with an index. Substrings can be specified with the *slice notation*: two indices separated by a colon.

Some examples:

```
>>> word = 'Help' + 'A'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Slice indices have useful defaults: an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

Indices may be negative numbers, to start counting from the right. Continuing from previous example:

```
>>> word[-1]      # The last character
'A'
>>> word[-2]      # The last-but-one character
'p'
>>> word[-2:]      # The last two characters
'pA'
>>> word[:-2]      # Everything except the last two characters
'Hel'
```

The following is an example program with strings.

```
# Program-8.1
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"
print("var1[0]: ", var1[0])
```

```
print("var2[1:5]: ", var2[1:5])
```

When the above code is executed, it produces the following output.

```
var1[0]:  H
var2[1:5]:  ytho
```

8.3 Can We Update Strings?

Python strings are *immutable*, i.e., *cannot be changed*. Assigning to an indexed position in a string results in an error, as shown below.

```
>>> word = 'Help' + 'A'
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support slice assignment
```

However, creating a new string with the combined content is easy and efficient, as shown below.

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Strings are immutable does not mean we cannot assign a new (separate) string “xyz” to an existing variable which is now assigned a string “pqr”. That means, we can “update” an existing variable, which now has a string value, by (re)assigning it another string. The new value can be related to its previous value or to a completely different string altogether.

The following explains this.

```
>>> str1 = "pqr"
>>> str2 = str1
>>> str2
'pqr'
>>> str1[0]="x"

Traceback (most recent call last):
  File "<pyshell#110>", line 1, in <module>
    str1[0]="x"
TypeError: 'str' object does not support item assignment
>>> str1
'pqr'
>>> str1="xyz"
>>> str1
'xyz'
>>> str2
'pqr'
>>> a = str2
>>> a
'pqr'
>>> id(a)
33770768
>>> id(str2)
33770768
>>> id(str1)
33261640
```

As shown above, we can use the built-in function `id()` to check if two variables refer to the same thing. The `id()` function returns the identity of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for an object during its lifetime. Two objects may have the same `id()` value, because it is the address of an object in memory.

8.4 Escape Characters

Table 8.1 shows a list of escape or non-printable characters that can be represented with backslash notation. An escape character gets interpreted in single quoted as well as double quoted strings.

Table 8.1 : Escape Characters

Backslash notation	Hexadecimal character	Description
<code>\a</code>	0x07	Bell or alert
<code>\b</code>	0x08	Backspace
<code>\cx</code>		Control-x
<code>\C-x</code>		Control-x
<code>\e</code>	0x1b	Escape
<code>\f</code>	0x0c	Formfeed
<code>\M-\C-x</code>		Meta-Control-x
<code>\n</code>	0x0a	Newline
<code>\nnn</code>		Octal notation, where n is in the range 0-7
<code>\r</code>	0x0d	Carriage return
<code>\s</code>	0x20	Space
<code>\t</code>	0x09	Tab
<code>\v</code>	0x0b	Vertical tab
<code>\x</code>		Character x
<code>\xnn</code>		Hexadecimal notation, where n is in the range 0-9, a-f, or A-F

8.5 String Special Operators

Assume variable `a` holds 'Hello' and variable `b` holds 'Python', then:

Operator	Description	Example
<code>+</code>	Concatenation - Adds values on either side of the operator	<code>a + b</code> will give <code>HelloPython</code>
<code>*</code>	Repetition - Creates new strings, concatenating multiple copies of the same string	<code>a*2</code> will give <code>HelloHello</code>
<code>[]</code>	Slice - Gives the character from the given index	<code>a[1]</code> will give <code>e</code>
<code>[:]</code>	Range Slice - Gives the characters from the given range	<code>a[1:4]</code> will give <code>ell</code>
<code>in</code>	Membership - Returns true if a character exists in the given string	<code>H in a</code> will give <code>1</code>

not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print (r'\n') prints \n and print (R'\n') prints \n
%	Format - Performs String formatting	See at next section

8.6 String Formatting Operator %

One of Python's most useful features is the string format operator % (which we have used before). This operator is unique to strings. The following is an example.

```
>>> print("My name is %s and I am %d years old!" % ('Kumar', 20))
My name is Kumar and I am 20 years old!
```

Here is the list of complete set of symbols which can be used along with %:

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following.

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number

#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

9 - Functions

A function is a block of organized, reusable code that is used to perform a single task. Functions are the *subprogram* control structure in Python (see Section 4.1). Functions provide better modularity for our programs and a high degree of code reuse.

Figure 9.1 shows the flow of control when a function is called and returns.

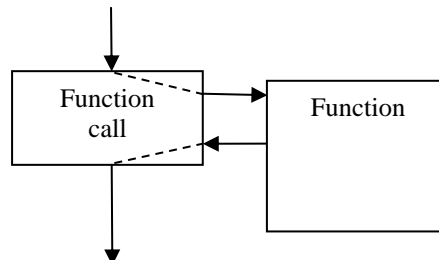


Figure 9.1: Flow of control in a function call and return

As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions which are called *user-defined functions*.

9.1 Defining a Function

We define functions to provide required functionality. Here are simple rules on functions in Python.

- Function definition begins with the keyword `def`, followed by the function name and parentheses.
- Any input parameters (or arguments) should be placed within these parentheses. You can also define parameters inside these parentheses.
- A colon (`:`) follows the closing parenthesis, on the same line.
- The *body* of a function starts in the line below and is indented.
- An optional string called the *docstring* within quotes documenting the function can be the first item in the body of the function.
- The statement `return [expression]` exits a function, passing the control back to the caller and optionally passing back an expression to the caller. A `return` statement with no arguments is the same as `return None`. If nothing is to be returned, the `return` statement may be omitted.

Syntax:

```
def functionname( parameters ):  
    "function_docstring"  
    function_suite  
    return [expression]
```

By default, parameters have a positional behavior; thus when invoking (calling) the function you need to list them in the same order that they were defined.

Here is an example Python function that takes a string as input parameter and prints it on screen.

```
def printme( string ):  
    # "This prints a passed string into this function"  
    print(string)  
    return
```

9.2 Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the function is defined, you can execute it by calling it from your (main) program, another function or directly from the Python prompt. In the following example, we call the `printme()` function. Note that in this example, the `return` is optional (the program will work even without the `return`).

```
#!/usr/bin/python
# Program-9.1

# Function definition is here
def printme( string ):
    # "This prints a passed string into this function"
    print(string)
    return

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result:

```
I'm first call to user defined function!
Again second call to the same function
```

Example 9.1 : Write a program to compute the circumference and area of a circle given its radius. Implement computation of circumference and area as separate functions.

```
#!/usr/bin/python
# Program-9.2

pi = 3.141

def area(r):
    return (pi*r*r)

def circumference(r):
    return (2*pi*r)

radius = 0.0
radius = float(input("Enter radius: "))
print("Area : ", area(radius))
print("Circumference : ", circumference(radius))
```

9.3 Parameter Passing

In Python, we deal with objects. *Variables* in Python can be more properly called *names* of objects. An *assignment* is the *reference (binding)* of a *name* to an *object*. Each binding has a *scope* that defines its visibility, usually a *block* or *function* in which the name originates. (Scope is discussed in more details later in sections 9.6 and 9.7).

All function parameters (arguments) in the Python language are *passed-by-object-reference*. It means if you change what a parameter refers to within a function, the change can reflect back in the calling function if the reference (binding) was to a mutable object. If we call `func(x)`, we are merely creating a binding within the scope of `func` to the object that the argument `x` is bound to when the function is called. If `x` refers to a mutable object and `func` changes its value, then these changes can be visible outside of the scope of the function.

For example:

```
#!/usr/bin/python
```

```
# Program-9.3

# Function definition is here
def changeme( inlist ):
    # "This changes a passed list into this function"
    inlist[0] = 5555
    print("Values inside the function: ", inlist)
    return

# Now you can call changeme function
mylist = [10,20,30]
print("Values before calling the function: ", mylist)
changeme( mylist )
print("Values after and outside the function: ", mylist)
```

The name `inlist` is a reference (is bound to) in the scope of the function `changeme` to the list object referenced by name `mylist` in the program scope. We change the list object (lists are mutable in Python) in the function; it will be visible outside the function. The following output will result:

```
Values before calling the function:  [10, 20, 30]
Values inside the function:  [5555, 20, 30]
Values after and outside the function:  [5555, 20, 30]
```

The following example demonstrates another point. In the function call, the argument `mylist` is the name bound to a list object; it is being passed-by-object-reference to the function (therefore the list object is bound to the name `inlist` within the function scope at the time of function call).

```
#!/usr/bin/python
# Program-9.4

# Function definition is here
def changeme( inlist ):
    # "This changes a passed list into this function"
    inlist = [1,2,3,4] # This binds inlist to a new list object
    print("Values inside the function: ", inlist)
    return

# Now you can call changeme function
mylist = [10,20,30];
print("Values before calling the function: ", mylist)
changeme( mylist )
print("Values outside the function: ", mylist)
```

Changing `inlist` within the function does not affect `mylist`. The assignment `inlist = [1,2,3,4]` binds the name `inlist` to a new (different) list object and that does not change the original list object bound to `mylist` outside the function. The program would produce the following output.

```
Values before calling the function:  [10, 20, 30]
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

9.4 Function Arguments

We can call a function by using the following types of formal arguments.

- Required arguments
- Keyword arguments
- Default arguments

- Variable-length arguments

9.4.1 Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call a function with required arguments, you definitely need to pass those arguments; otherwise it would cause a syntax error. Here is an example:

```
#!/usr/bin/python
# Program-9.5

# Function definition is here
def printme( string ):
    # "This prints a passed string into this function"
    print(string)
    return

# Now you can call printme function
printme()
```

When the above code is executed, we get the following syntax error.

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

9.4.2 Keyword Arguments

When you use keyword arguments in a function call, the caller identifies the arguments by the argument name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the `printme()` function above in the following ways:

```
#!/usr/bin/python
# Program-9.6
# Function definition is here
def printme( string ):
    # "This prints a passed string into this function"
    print(string)
    return

# Now you can call printme function
printme( string = "My string")
```

When the above code is executed, it produces the following result:

```
My string
```

The following is another example. Note, that the order of the arguments does not matter.

```
# Program-9.7
#!/usr/bin/python
# Function definition is here
def printinfo( name, age ):
    # "This prints a passed info into this function"
    print("Name: ", name)
    print("Age ", age)
    return

# Now you can call printinfo function
printinfo( age=20, name="Kumari" )
```

When the above code is executed, it produces the following result:

```
Name: Kumari
Age 20
```

9.4.3 Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments; it would print default age if a value for age is not passed.

```
#!/usr/bin/python
# Program-9.8

# Function definition is here
def printinfo( name, age = 25 ):
    # "This prints a passed info into this function"
    print("Name: ", name)
    print("Age ", age)
    return

# Now you can call printinfo function
printinfo( age=20, name="Kumari" );
printinfo( name="Kumari" );
```

When the above code is executed, it produces the following result:

```
Name: Kumari
Age 20
Name: Kumari
Age 25
```

9.4.4 Variable-Length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax: The general syntax for a function with non-keyword variable-length arguments is:

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An *asterisk* (*) is placed before the variable name that will hold the values of all non-keyword variable arguments. This *tuple* remains empty if no additional arguments are specified during the function call. The following is an example.

```
#!/usr/bin/python
# Program-9.9

# Function definition is here
def printinfo( arg1, *vartuple ):
    # "This prints a variable passed arguments"
    print("Output is: ")
    print(arg1)
    for var in vartuple:
        print(var)
    return

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result:

```
Output is:
10
Output is:
70
60
50
```

9.5 The *return* Statement

The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

None of the examples so far return any value, but if you like you can return a value from a function as shown in the following example.

```
#!/usr/bin/python
# Program-9.10

# Function definition is here
def sum( arg1, arg2 ):
    # "Add both the parameters and return them."
    total = arg1 + arg2
    print("Inside the function : ", total)
    return total

# Now you can call sum function
total = sum( 10, 20 )
print("Outside the function : ", total)
```

When the above code is executed, it produces the following result:

```
Inside the function : 30
Outside the function : 30
```

Exercise **9.1** What would be the output of the following code segment: 10 , 100 or 110 ?

```
def passValue( a ):
    a=100
    return

x=10
passValue(x)
print(x)
```

9.6 Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you declare a variable.

The *scope of a variable* determines the portion of the program where you can access the particular variable. There are two basic scopes of variables in Python:

- Global variables
- Local variables

9.7 Global vs. Local Variables

Variables that are defined inside a function body (*local variables*) have a local scope, and those defined outside (*global variables*) have a global scope.

You can write to (assign a value to) a global variable in a function by declaring it as `global` in the function. Python assumes that any name that is *assigned to*, anywhere within a function, is *local* to that function unless explicitly told otherwise using the keyword `global`. If the function is only reading from a name, and the name doesn't exist locally, it will try to look up the name in any containing scopes (e.g. the global scope).

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example:


```
#!/usr/bin/python
# Program-9.11

total = 0    # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2    # Here total is a local variable.
    print("Inside the function local total : ", total)
    return total

# Now you can call sum function
sum( 10, 20 )
print("Outside the function global total : ", total)
```

When the above code is executed, it produces the following output.

```
Inside the function local total : 30
Outside the function global total : 0
```

In the above function `sum()`, if there was a line “global total” before the line “total = arg1 + arg2”, then the program output will have 30 in both cases.

Exercise 9.2 : Predict the output of each `print` statement in Program-9.12.

```
#!/usr/bin/python
# Program-9.12

a = 5
def swap1(x,y):
    z=x
    x=y
    y=z
    return None
def swap2(x):
    global a
    z=x
    x=a
    a=z
    return None
b =10
c =15
print("Value before 1st function a = ", a," b = ",b," c = ",c)
swap1(b,c)
print("Value after 1st function a = ", a," b = ",b," c = ",c)
swap2(b)
print("Value after 2nd function a = ", a," b = ",b," c = ",c)
print("Test")
```

Exercise 9.2 Write a function that can find and return all the even numbers in a list of numbers and return it as a list. For example, if `[2, 4, 6, 5, 3, 8, 1]` is the input, the output will be `[2, 4, 6, 8]`.

10 - Handling Files

The programs that were developed up to now were only able to read data from the keyboard and write to the screen (except for a couple of programs in Section 2.9 which briefly introduced file handling). Using files you can save your output data permanently and retrieve them later. This chapter discusses how to do this in Python.

A file is a collection of data elements or can be considered a set of related *records* such as student information, marks obtained in an exam, employee salaries, etc. Each record is a collection of related items called *fields*, such as “student name”, “date of birth”, “subjects registered”, etc. The common operations associated with a file are:

- Read from a file (input)
- Write to a file (output)
- Append to a file (write to the end of a file)
- Update a file (modifying any location within the file)

10.1 Printing to the Screen

The simplest way to produce output is using the `print` statement where you can pass zero or more expressions separated by commas. This converts the expressions you pass into a string and writes the result to standard output. Here is an example:

```
#!/usr/bin/python
print("Python is really a great language,", "isn't it?");
```

This would produce the following result on your standard output (display screen):

```
Python is really a great language, isn't it?
```

10.2 Reading Keyboard Input

Python 3.6 provides a built-in function to read a line of text from standard input, which by default comes from the keyboard. This function is:

- `input`

The `input([prompt])` function reads one line from standard input and returns it as a string (removing the trailing newline). Here is an example:

```
#!/usr/bin/python
string = input("Enter your input: ")
print("Received input is : ", string)
```

This would produce the following result against the entered input.

```
Enter your input: 10
Received input is : 10
```

10.3 Opening and Closing of Files

Until now, you have been reading from standard input and writing to the standard output. Now, we will see how to play with actual data files.

Python provides basic functions to manipulate files. You can perform most file operations using *file objects*.

10.3.1 The open Function

Before you can read or write a file, you have to open it using Python's built-in `open()` function. This function creates a *file object*, which would be utilized to call *methods* (functions that perform operations on the file object).

Syntax: The syntax for calling the `open()` in Python:

```
file_object = open(file_name [, access_mode][, buffering])
```

The parameters (arguments) used with the `open()` function are as follows:

- **file_name:** A string value that contains the name of the file that you want to access.
- **access_mode:** This determines the mode in which the file has to be opened, for e.g., read, write or append. A complete list of possible mode values are given below in a table. This is an optional parameter and the default file access mode is read (`r`).
- **buffering:** If set to 0, no buffering will take place. If the value is 1, line buffering will be performed while accessing a file. If the value is an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior). This is an optional parameter.

A list of different modes of opening a file is given below.

Mode	Description
<code>r</code>	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
<code>rb</code>	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
<code>r+</code>	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
<code>rb+</code>	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
<code>w</code>	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
<code>wb</code>	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
<code>w+</code>	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
<code>wb+</code>	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
<code>a</code>	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
<code>ab</code>	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Attributes of a *file object*:

Once a file is opened and you have a *file object* associated with that file, you can get related information (attributes) of that file. Here is a list of attributes of file objects.

Attribute	Description
f.closed	Returns True if the file f is closed, False otherwise.
f.mode	Returns the access mode with which file f was opened.
f.name	Returns the name of the file f.

Here is an example program that uses these (it requires a file named "foo.txt" to exist).

```
#!/usr/bin/python
# Program-10.1

# Open a file
fo = open("foo.txt", "wb")
print("Name of the file: ", fo.name)
print("File closed: ", fo.closed)
print("Opening mode : ", fo.mode)
```

The above program would produce the following result.

```
Name of the file:  foo.txt
File closed:  False
Opening mode :  wb
```

10.3.2 The `close` Method

The `f.close()` method of a *file object* f flushes any unwritten data and closes the file object, after which no more writing can be done.

Python automatically closes a file when its reference object is reassigned to another file. It is a good practice to explicitly close a file using the `close()` method.

Syntax: The syntax for using the `close()` method of a file object f in Python:

```
f.close()
```

Here is an example.

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print("Name of the file: ", fo.name)

# Close the file
fo.close()
```

This would produce the following result:

```
Name of the file:  foo.txt
```

10.4 Reading and Writing Files

The *file* object provides methods to read and write files, as described below.

10.4.1 File Reading

The `f.read()` method reads some quantity of data from an open file object `f` and returns it as a string. It is important to note that Python strings can have binary data as well, not only text.

Syntax: The syntax for using the `read()` method of a file object `f` in Python:

```
f.read([count])
```

Here, the optional argument `count` is the maximum number of bytes to be read. This method starts reading from the beginning of the file and at most `count` bytes are read and returned. If `count` is omitted or negative, the entire contents of the file will be read and returned.

The `f.readline()` method reads a single line from a file `f`; a newline character (`'\n'`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. If `f.readline()` returns an empty string, the end of the file `f` has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline. As in the `f.read()` method, this method too has an optional argument `count`, which sets a maximum byte count (including the trailing newline) to read; as a result an incomplete line may be returned.

For reading lines from a file, we can loop over the file object. This is memory efficient, fast, and leads to simple code. Here is an example in which a file is read and its content output in a loop, a line in each iteration.

```
#!/usr/bin/python
# Program-10.2

fo = open("foo.txt", "r")
for line in fo:
    print(line,end='')

# Close the file
fo.close()
```

It is a good practice to use the **with** keyword when dealing with file objects. This has the advantage that the file is properly closed when the `with` block finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks.

The following example program-10.3 shows how the previous program-10.2 can be re-written using the `with` keyword.

```
#!/usr/bin/python
# Program-10.3

with open("foo.txt", "r") as fo:
    for line in fo:
        print(line)
```

Note that an explicit `fo.close()` is not needed in the above, as the file is automatically closed when the `with` block is exited.

10.4.2 File Writing

The `f.write()` method writes a string to an open file `f`. There is no return value (i.e., it returns `None`). Due to buffering, the string may not actually show up in the file until the `f.flush()` or `f.close()` method is called. The `f.write()` method does not add a newline character (`\n`) to the end of the string.

Syntax: The syntax for using the `write()` method of a file object `f` in Python:

```
f.write(string)
```

Here, the argument 'string' is the string content to be written into the opened file `f`.

Here is an example.

```
#!/usr/bin/python
# Program-10.4
#
fo = open("foobar.txt", "w")
fo.write( "Python is a great language.\nYeah I love it!\n")
fo.close()
```

The above would create the `foobar.txt` file, write the given content in that file and finally close that file. If you subsequently open this file, you would see the following content.

```
Python is a great language.
Yeah I love it!
```

10.5 File Positions

The `f.tell()` method tells us the current position within the file `f`; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The `f.seek(offset[, from])` method changes the current file position. The `offset` argument indicates the amount of change by number of bytes. The optional `from` argument specifies the reference position from where the position is to be changed.

If `from` is set to 0, the beginning of the file is used as the reference position; 1 means use the current position as the reference position; 2 means the end of the file is taken as the reference position.

In the following example Program-10.5, the file `foobar.txt` created in Program-10.4 above is used.

```
#!/usr/bin/python
# Program-10.5

# Open file
fo = open("foobar.txt", "r+")
string = fo.read(10)
print("Read string is : ", string)

# Check current position
position = fo.tell()
print("Current file position : ", position)

# Position pointer at the beginning once again
position = fo.seek(0, 0)
string = fo.read(10)
print("Again, read string is : ", string)
# Close file
fo.close()
```

This would produce the following output:

```
Read string is : Python is
Current file position : 10
Again, read string is : Python is
```

10.6 File and Directory Handling with the "os" Module

The `os` module in Python provides many functions that help you perform file and directory processing operations, among others.

To use this module you need to import it first and then call the specific functions.

Here is an example program that uses the functions in the `os` module to rename a file, remove (delete) a file and create a directory.

```
#!/usr/bin/python
# Program-10.6

import os

# Ensure a text file "test1.txt" exists to make this work

# Rename file "test1.txt" as "test2.txt"
os.rename( "test1.txt", "test2.txt" )

# Delete file "test2.txt"
os.remove("test2.txt")

# Create a directory "test"
os.mkdir("test")
```

Appendix A - Library Functions

The following is a list of commonly used functions. For a more complete list reader should refer to the manual (or help) of the version of Python compiler that is being used.

List Functions

Python includes the following list functions:

SN	Function with Description
1	<u>cmp(list1, list2)</u> Compares elements of both lists.
2	<u>len(list)</u> Gives the total length of the list.
3	<u>max(list)</u> Returns item from the list with max value.
4	<u>min(list)</u> Returns item from the list with min value.
5	<u>list(seq)</u> Converts a tuple into list.

Python includes the following list methods

SN	Methods with Description
1	<u>list.append(obj)</u> Appends object obj to list
2	<u>list.count(obj)</u> Returns count of how many times obj occurs in list
3	<u>list.extend(seq)</u> Appends the contents of seq to list
4	<u>list.index(obj)</u> Returns the lowest index in list that obj appears
5	<u>list.insert(index, obj)</u> Inserts object obj into list at offset index
6	<u>list.pop(obj=list[-1])</u> Removes and returns last object or obj from list
7	<u>list.remove(obj)</u> Removes object obj from list
8	<u>list.reverse()</u> Reverses objects of list in place
9	<u>list.sort([func])</u> Sorts objects of list, use compare func if given

String Functions

Python includes the following built-in functions to manipulate strings:

SN	Functions with Description
1	<u>capitalize()</u> Capitalizes first letter of string
2	<u>center(width, fillchar)</u> Returns a space-padded string with the original string centered to a total of width columns
3	<u>count(str, beg= 0,end=len(string))</u> Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given
4	<u>decode(encoding='UTF-8',errors='strict')</u> Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	<u>encode(encoding='UTF-8',errors='strict')</u> Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	<u>endswith(suffix, beg=0, end=len(string))</u> Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise
7	<u>expandtabs(tabsize=8)</u> Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided
8	<u>find(str, beg=0 end=len(string))</u> Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise
9	<u>index(str, beg=0, end=len(string))</u> Same as find(), but raises an exception if str not found
10	<u>isalnum()</u> Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise
11	<u>isalpha()</u> Returns true if string has at least 1 character and all characters are alphabetic and false otherwise
12	<u>isdigit()</u> Returns true if string contains only digits and false otherwise
13	<u>islower()</u> Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise
14	<u>isnumeric()</u> Returns true if a unicode string contains only numeric characters and false otherwise
15	<u>isspace()</u> Returns true if string contains only whitespace characters and false otherwise
16	<u>istitle()</u> Returns true if string is properly "titlecased" and false otherwise

17	<u>isupper()</u> Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise
18	<u>join(seq)</u> Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string
19	<u>len(string)</u> Returns the length of the string
20	<u>ljust(width[, fillchar])</u> Returns a space-padded string with the original string left-justified to a total of width columns
21	<u>lower()</u> Converts all uppercase letters in string to lowercase
22	<u>lstrip()</u> Removes all leading whitespace in string
23	<u>maketrans()</u> Returns a translation table to be used in translate function.
24	<u>max(str)</u> Returns the max alphabetical character from the string str
25	<u>min(str)</u> Returns the min alphabetical character from the string str
26	<u>replace(old, new [, max])</u> Replaces all occurrences of old in string with new or at most max occurrences if max given
27	<u>rfind(str, beg=0, end=len(string))</u> Same as find(), but search backwards in string
28	<u>rindex(str, beg=0, end=len(string))</u> Same as index(), but search backwards in string
29	<u>rjust(width[, fillchar])</u> Returns a space-padded string with the original string right-justified to a total of width columns.
30	<u>rstrip()</u> Removes all trailing whitespace of string
31	<u>split(str="", num=string.count(str))</u> Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given
32	<u>splitlines(num=string.count('\n'))</u> Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed
33	<u>startswith(str, beg=0, end=len(string))</u> Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise
34	<u>strip([chars])</u> Performs both lstrip() and rstrip() on string

35	<u>swapcase()</u> Inverts case for all letters in string
36	<u>title()</u> Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase
37	<u>translate(table, deletechars="")</u> Translates string according to translation table str(256 chars), removing those in the del string
38	<u>upper()</u> Converts lowercase letters in string to uppercase
39	<u>zfill (width)</u> Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero)
40	<u>isdecimal()</u> Returns true if a unicode string contains only decimal characters and false otherwise

Mathematical Functions (from the `math` module)

The prefix "`math.`" should be inserted to each of the following. Note that `abs()` is a built-in Python function which is always available and not listed here.

Function	Returns (description)
<u>ceil(x)</u>	The ceiling of x: the smallest integer not less than x
<u>cmp(x, y)</u>	-1 if $x < y$, 0 if $x == y$, or 1 if $x > y$
<u>exp(x)</u>	The exponential of x: e^x
<u>fabs(x)</u>	The absolute value of x.
<u>floor(x)</u>	The floor of x: the largest integer not greater than x
<u>log(x)</u>	The natural logarithm of x, for $x > 0$
<u>log10(x)</u>	The base-10 logarithm of x for $x > 0$.
<u>max(x1, x2,...)</u>	The largest of its arguments: the value closest to positive infinity
<u>min(x1, x2,...)</u>	The smallest of its arguments: the value closest to negative infinity
<u>modf(x)</u>	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
<u>pow(x, y)</u>	The value of $x^{**}y$.
<u>round(x [,n])</u>	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: <code>round(0.5)</code> is 1.0 and <code>round(-0.5)</code> is -1.0.
<u>sqrt(x)</u>	The square root of x for $x > 0$

Random Number Functions (from `random` module)

Python's `random` module includes following functions that are commonly used. The prefix "`random.`" should be inserted to each of the following.

Function	Description
<u>choice(seq)</u>	A random item from a list, tuple, or string.
<u>randrange([start,] stop [,step])</u>	A randomly selected element from <code>range(start, stop, step)</code>
<u>random()</u>	A random float <code>r</code> , such that 0 is less than or equal to <code>r</code> and <code>r</code> is less than 1
<u>seed([x])</u>	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns <code>None</code> .
<u>shuffle(lst)</u>	Randomizes the items of a list in place. Returns <code>None</code> .
<u>uniform(x, y)</u>	A random float <code>r</code> , such that <code>x</code> is less than or equal to <code>r</code> and <code>r</code> is less than <code>y</code>

Trigonometric Functions (from `math` module)

Python includes following functions that perform trigonometric calculations. The prefix "`math.`" should be inserted to each of the following.

Function	Description
<u>acos(x)</u>	Return the arc cosine of <code>x</code> , in radians.
<u>asin(x)</u>	Return the arc sine of <code>x</code> , in radians.
<u>atan(x)</u>	Return the arc tangent of <code>x</code> , in radians.
<u>atan2(y, x)</u>	Return <code>atan(y / x)</code> , in radians.
<u>cos(x)</u>	Return the cosine of <code>x</code> radians.
<u>hypot(x, y)</u>	Return the Euclidean norm, <code>sqrt(x*x + y*y)</code> .
<u>sin(x)</u>	Return the sine of <code>x</code> radians.
<u>tan(x)</u>	Return the tangent of <code>x</code> radians.
<u>degrees(x)</u>	Converts angle <code>x</code> from radians to degrees.
<u>radians(x)</u>	Converts angle <code>x</code> from degrees to radians.

Mathematical Constants (from `math` module)

The `math` module also defines two mathematical constants:

Constants	Description
<code>math.pi</code>	The mathematical constant <code>pi=3.141592...</code> , to available precision.
<code>math.e</code>	The mathematical constant <code>e= 2.718281...</code> , to available precision.

----- END OF PART A -----