

MODULE I

1.1 THE WAY OF THE PROGRAM

- ✓ Programs are generally written to solve the real-time arithmetic/logical problems.
- ✓ Nowadays, computational devices like personal computer, laptop, and cell phones are embedded with operating system, memory and processing unit. Using such devices one can write a program in the language (which a computer can understand) of one's choice to solve various types of problems.
- ✓ Humans are tend get bored by doing computational tasks multiple times. Hence, the computer can act as a personal assistant for people for doing their job!! To make a computer to solve the required problem, one has to feed the proper program to it. Hence, one should know how to write a program!!
- ✓ There are many programming languages that suit several situations. The programmer must be able to choose the suitable programming language for solving the required problem based on the factors like computational ability of the device, data structures that are supported in the language, complexity involved in implementing the algorithm in that language etc.

❖ Creativity and Motivation

- ✓ When a person starts programming, he himself will be both the programmer and the end-user. Because, he will be learning to solve the problems.
- ✓ But, later, he may become a proficient programmer. A programmer should have logical thinking ability to solve a given problem. He/she should be creative in analyzing the given problems, finding the possible solutions, optimizing the resources available and delivering the best possible results to the end-user.
- ✓ Motivation behind programming may be a job-requirement and such other prospects. But the programmer should follow certain ethics in delivering the best possible output to his/her clients.
- ✓ The responsibilities of a programmer include developing a feasible, user-friendly software with very less or no hassles.
- ✓ The user is expected to have only the abstract knowledge about the working of software, but not the implementation details. Hence, the programmer should strive hard towards developing most effective software.

1.2 COMPUTER HARDWARE ARCHITECTURE

To understand the art programming, it is better to know the basic architecture of computer hardware.

The computer system involves some of the important parts as shown in Figure 1.1, these parts are as explained below:

Central Processing Unit (CPU): It performs basic arithmetic, logical, control and I/O operations specified by the program instructions. CPU will perform the given tasks with a tremendous speed. Hence, the good programmer has to keep the CPU busy by providing enough tasks to it.

Main Memory: It is the storage area to which the CPU has a direct access. Usually, the programs stored in the secondary storage are brought into main memory before the execution. The processor (CPU) will pick a job from the main memory and performs the tasks. Usually, information stored in the main memory will be vanished when the computer is turned-off.

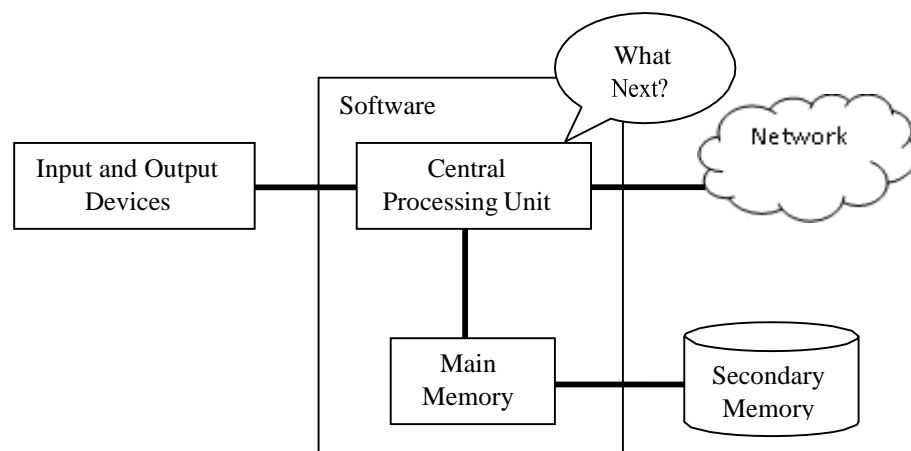


Figure 1.1 Computer Hardware Architecture

Secondary Memory: The secondary memory is the permanent storage of computer. Usually, the size of secondary memory will be considerably larger than that of main memory. Hard disk, USB drive etc can be considered as secondary memory storage.

I/O Devices: These are the medium of communication between the user and the computer. Keyboard, mouse, monitor, printer etc. are the examples of I/O devices.

Network Connection: Nowadays, most of the computers are connected to network and hence they can communicate with other computers in a network. Retrieving the

information from other computers via network will be slower compared to accessing the secondary memory. Moreover, network is not reliable always due to problem in connection.

The programmer has to use above resources sensibly to solve the problem.

Usually, a programmer will be communicating with CPU by telling it “what to do next”. The usage of main memory, secondary memory, I/O devices also can be controlled by the programmer.

To communicate with the CPU for solving a specific problem, one has to write a set of instructions. Such a set of instructions is called as a program.

1.3 UNDERSTANDING PROGRAMMING

A programmer must have skills to look at the data/information available about a problem, analyze it and then to build a program to solve the problem. The skills to be possessed by a good programmer includes –

Thorough knowledge of programming language: One needs to know the vocabulary and grammar (technically known as syntax) of the programming language. This will help in constructing proper instructions in the program.

Skill of implementing an idea: A programmer should be like a „story teller“. That is, he must be capable of conveying something effectively. He/she must be able to solve the problem by designing suitable algorithm and implementing it. And, the program must provide appropriate output as expected.

Thus, the art of programming requires the knowledge about the problem’s requirement and the strength/weakness of the programming language chosen for the implementation. It is always advisable to choose appropriate programming language that can cater the complexity of the problem to be solved.

1.4 Words and Sentences

Every programming language has its own constructs to form syntax of the language.

Basic constructs of a programming language includes set of characters and keywords that it supports.

The keywords have special meaning in any language and they are intended for doing specific task. Python has a finite set of keywords as given in Table below.

Table 1: Keywords in Python

and	as	assert	Break	class	continue
def	del	elif	Else	except	False
finally	for	from	Global	if	import
In	is	lambda	None	nonlocal	not
Or	pass	raise	Return	True	try
while	with	Yield			

programmer may use *variables* to store the values in a program.

Unlike many other programming languages, a variable in Python need not be declared before its use.

1.5 PYTHON EDITORS AND INSTALLING PYTHON

Before getting into details of the programming language Python, it is better to learn how to install the software.

Python is freely downloadable from the internet. There are multiple IDEs (Integrated Development Environment) available for working with Python. Some of them are PyCharm, LiClipse, IDLE etc.

When you install Python, the IDLE editor will be available automatically. Apart from all these editors, Python program can be run on command prompt also.

One has to install suitable IDE depending on their need and the Operating System they are using.

Because, there are separate set of editors (IDE) available for different OS like Window, UNIX, Ubuntu, Solaris, Mac, etc. The basic Python can be downloaded from the link:

<https://www.python.org/downloads/>

Python has rich set of libraries for various purposes like large-scale data processing, predictive analytics, scientific computing etc. Based on one's need, the required packages can be downloaded. But, there is a free open source distribution *Anaconda*, which simplifies package management and deployment.

Hence, it is suggested for the readers to install *Anaconda* from the below given link, rather than just installing a simple Python.

<https://anaconda.org/anaconda/python>

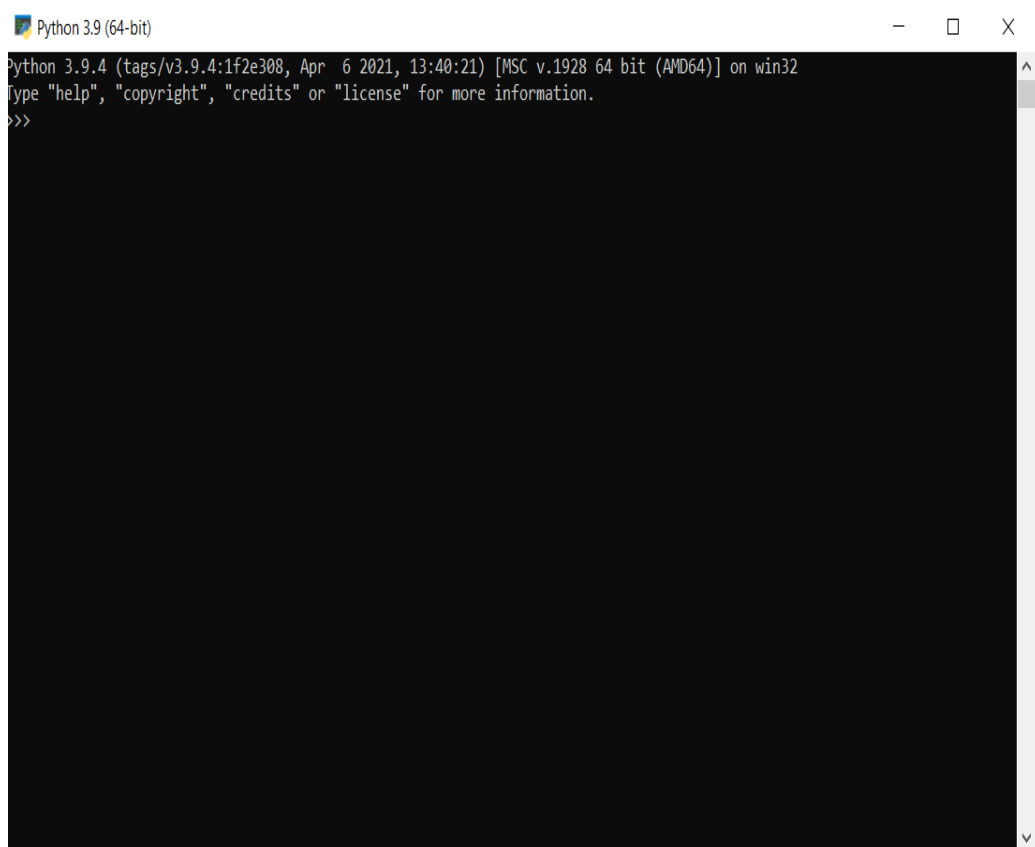
Successful installation of *anaconda* provides you Python in a command prompt, the default editor IDLE and also a browser-based interactive computing environment known as *jupyter notebook*.

❖ Conversing with Python

Once Python is installed, one can go ahead with working with Python.

Use the IDE of your choice for doing programs in Python.

After installing Python (or Anaconda distribution), if you just type „python“ in the command prompt, you will get the message as shown in Figure 1.2.



```
Python 3.9 (64-bit)
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr 6 2021, 13:40:21) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 1.2: Python initialization in command prompt

The prompt `>>>` (usually called as *chevron*) indicates the system is ready to take Python instructions.

If you would like to use the default IDE of Python, that is, the IDLE, then you can just run IDLE and you will get the editor as shown in Figure 1.3.

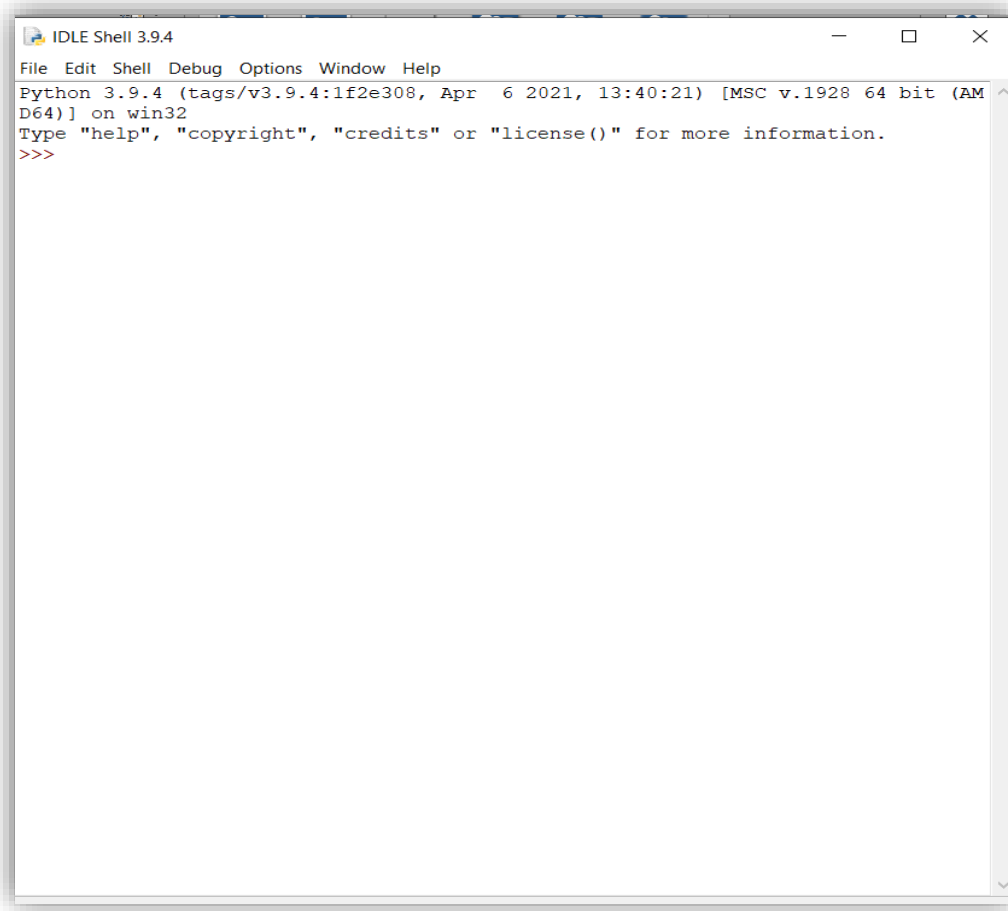


Figure 1.3: Python IDLE editor

After understanding the basics of few editors of Python, let us start our communication with Python, by saying *Hello World*. The Python uses *print()* function for displaying the contents. Consider the following code –

```
>>> print("Hello World")           #type this and press
enter key                           #output displayed
Hello World                          #prompt returns again
>>>
```

Here, after typing the first line of code and pressing the enter key, we could able to get the output of that line immediately. Then the prompt (>>>) is returned on the screen.

This indicates, Python is ready to take next instruction as input for processing.

Once we are done with the program, we can close or terminate Python by giving *quit()* command as shown –

```
>>> quit()                          #Python terminates
```

❖ Terminology: Interpreter and Compiler

All digital computers can understand only the machine language written in terms of zeros and ones. But, for the programmer, it is difficult to code in machine language.

Hence, we generally use high level programming languages like Java, C++, PHP, Perl, JavaScript etc.

Python is also one of the high level programming languages. The programs written in high level languages are then translated to machine level instruction so as to be executed by CPU.

How this translation behaves depending on the type of translators viz. *compilers* and *interpreters*.

A compiler translates the source code of high-level programming language into machine level language. For this purpose, the source code must be a complete program stored in a file (with extension, say, .java, .c, .cpp etc). The compiler generates executable files (usually with extensions

.exe, .dll etc) that are in machine language. Later, these executable files are executed to give the output of the program.

On the other hand, interpreter performs the instructions directly, without requiring them to be pre-compiled. Interpreter parses (syntactic analysis) the source code and interprets it immediately. Hence, every line of code can generate the output immediately, and the source code as a complete set, need not be stored in a file. That is why, in the previous section, the usage of single line `print("Hello World")` could be able to generate the output immediately.

Consider an example of adding two numbers –

```
>>> x=10
>>> y=20
>>> z= x+y
>>> print(z)
30
```

Here, x, y and z are variables storing respective values. As each line of code above is processed immediately after the line, the variables are storing the given values.

Observe that, though each line is treated independently, the knowledge (or information) gained in the previous line will be retained by Python and hence, the further lines can make use of previously used variables.

Thus, each line that we write at the Python prompt are logically related, though they look independent.

NOTE that, Python do not require variable declaration (unlike in C, C++, Java etc) before its use. One can use any valid variable name for storing the values. Depending on the type (like number, string etc)of the value being assigned, the type and behavior of the variable name is judged by Python.

❖ Writing a Program

As Python is interpreted language, one can keep typing every line of code one after the other (and immediately getting the output of each line) as shown in previous section.

But, in real-time scenario, typing a big program is not a good idea. It is not easy to logically debug such lines.

Hence, Python programs can be stored in a file with extension *.py* and then can be run using python command.

Programs written within a file are obviously reusable and can be run whenever we want. Also, they are transferrable from one machine to other machine via pen-drive, CD etc.

❖ What is a Program?

A program is a sequence of instructions intended to do some task.

For example, if we need to count the number of occurrences of each word in a text document, we can write a program to do so.

Writing a program will make the task easier compared to manually counting the words in a document.

Moreover, most of the times, the program is a generic solution. Hence, the same program may be used to count the frequency of words in another file.

The person who does not know anything about the programming also can run this program to count the words.

Programming languages like Python will act as an intermediary between the computer and the programmer. The end-user can request the programmer to write a program to solve one's problem.

1.6 THE BUILDING BLOCKS OF PROGRAMS

There are certain low-level conceptual structures to construct a program in any programming language.

They are called as building-blocks of a program and listed below –

- **Input:** Every program may take some inputs from outside. The input may be through keyboard, mouse, disk-file etc. or even through some sensors like microphone, GPS etc.
- **Output:** Purpose of a program itself is to find the solution to a problem. Hence, every program must generate at least one output. Output may be displayed on a monitor or can be stored in file. Output of a program may even be a music/voice message.
- **Sequential Execution:** In general, the instructions in the program are sequentially executed from the top.
- **Conditional Execution:** In some situations, a set of instructions have to be executed based on the truth-value of a variable or expression. Then conditional constructs (like *if*) have to be used. If the condition is true, one set of instructions will be executed and if the condition is false, the true-block is skipped.
- **Repeated Execution:** Some of the problems require a set of instructions to be repeated multiple times. Such statements can be written with the help of looping structures like *for*, *while* etc.
- **Reuse:** When we write the programs for general-purpose utility tasks, it is better to write them with a separate name, so that they can be used multiple times whenever/wherever required. This is possible with the help of *functions*.

The art of programming involves thorough understanding of the above constructs and using them legibly.

❖ What Could Possibly Go Wrong?

It is obvious that one can do mistakes while writing a program. The possible mistakes are categorized as below –

- **Syntax Errors:** The statements which are not following the grammar (or syntax) of the programming language are tend to result in syntax errors. Python is a case-sensitive language. Hence, there is a chance that a beginner may do some syntactical mistakes while writing a program. The lines involving such mistakes are encountered

by the Python when you run the program and the errors are thrown by specifying possible reasons for the error. The programmer has to correct them and then proceed further.

- **Runtime Errors:** Usually called as *exceptions*. It may occur due to wrong input (like trying to divide a number by zero), problem in database connectivity etc. When a runtime error occurs, the program throws some error, which may not be understood by the normal user. And he/she may not understand how to overcome such errors. Hence, suspicious lines of code have to be treated by the programmer himself by the procedure known as *exception handling*. Python provides mechanism for handling various possible exceptions like *ArithmeticError*, *FloatingpointError*, *EOFError*, *MemoryError* etc

- **Semantic Errors:** A semantic error may happen due to wrong use of variables, wrong operations or in wrong order. For example, trying to modify un-initialized variable etc.
 - **Logical Errors:** Logical error occurs due to poor understanding of the problem. Syntactically, the program will be correct. But, it may not give the expected output. For example, you are intended to find $a\%b$, but, by mistake you have typed a/b . Then it is a logical error.

1.7 VARIABLES, EXPRESSIONS AND STATEMENTS

After understanding some important concepts about programming and programming languages, we will now move on to learn Python as a programming language with its syntax and constructs.

❖ Values and Types

A *value* is one of the basic things a program works with.

It may be like 2, 10.5, “Hello” etc.

Each value in Python has a type. Type of 2 is integer; type of 10.5 is floating point number; “Hello” is string etc.

The type of a value can be checked using *type* function as shown below –

```
>>> type("hello")
      <class 'str'>
>>> type(3)
      <class 'int'>
>>> type(10.5)
      <class 'float'>
>>> type("15")
      <class 'str'>
```

In the above four examples, one can make out various types *str*, *int* and *float*.

Observe the 4th example – it clearly indicates that whatever enclosed within a double quote is a string.

❖ Variables

A variable is a named-literal which helps to store a value in the program.

Variables may take value that can be modified wherever required in the program.

Note that, in Python, a variable need not be declared with a specific type before its usage.

Whenever we want a variable, just use it. The type of it will be decided by the value assigned to it.

A value can be assigned to a variable using *assignment operator* (=).

Consider the example given below–

```
>>> x=10
>>> print(x)
      10                                #output
>>> type(x)
      <class 'int'>                     #type of x is integer
>>> y="hi"
>>> print(y)
      hi                                #output
>>> type(y)
      <class 'str'>                     #type of y is string
```

It is observed from above examples that the value assigned to variable determines the type of that variable.

❖ Variable Names and Keywords

It is a good programming practice to name the variable such that its name indicates its purpose in the program.

There are certain rules to be followed while naming a variable –

- Variable name must not be a keyword
- They can contain alphabets (lowercase and uppercase) and numbers, but should not start with a number.
- It may contain a special character underscore(_), which is usually used to combine variables with two words like *my_salary*, *student_name* etc. No other special characters like @, \$ etc. are allowed.
- Variable names can start with an underscore character, but we generally avoid it.
- As Python is case-sensitive, variable name *sum* is different from *SUM*, *Sum* etc.

Examples:

```
>>> 3a=5                                #starting with a number
SyntaxError: invalid syntax
>>> a$=10                                #contains $
SyntaxError: invalid syntax
>>> if=15                                #if is a keyword
SyntaxError: invalid syntax
```

❖ Statements

A *statement* is a small unit of code that can be executed by the Python interpreter. It indicates some action to be carried out. In fact, a program is a sequence of such statements.

Two kinds of statements are: print being an expression statement and assignment statement

Following are the examples of statements –

```
>>> print("hello")                       #printing statement
hello
>>> x=5                                   #assignment statement
>>> print(x)                              #printing statement
5
```

❖ Operators and Operands

Special symbols used to indicate specific tasks are called as *operators*.

An operator may work on single operand (unary operator) or two operands (binary operator).

There are several types of operators like arithmetic operators, relational operators, logical operators etc, in Python.

Arithmetic Operators are used to perform basic operations as listed in Table:

Table2: listing Arithmetic Operators

Operator	Meaning	Example
+	Addition	Sum= a+b
-	Subtraction	Diff= a-b
*	Multiplication	Pro= a*b
/	Division	Q = a/b X = 5/3 (X will get a value 1.666666667)
//	Floor Division – returns only integral part after division	F = a//b X= 5//3 (X will get a value 1)
%	Modulus remainder after division	R = a %b (Remainder after dividing a by b)
**	Exponent	E = x** y (means x to the powder of y)

Relational or Comparison Operators are used to check the relationship (like less than, greater than etc) between two operands. These operators return a Boolean value – either *True* or *False*.

Assignment Operators: Apart from simple assignment operator = which is used for assigning values to variables, Python provides compound assignment operators.

For example,

`x=x+y` can be written as `x+=y`

Now, += is compound assignment operator. Similarly, one can use most of the arithmetic and bitwise operators (only binary operators, but not unary) like *, /, %, //, &, ^ etc. as compound assignment operators.

For example,

```
>>> x=3
>>> y=5
>>> x+=y          #x=x+y

>>> print (x)
8

>>> y//=2        #y=y//2

>>> print(y)
2                #only integer part will be printed
```

NOTE:

Python has a special feature – one can assign values of different types to multiple variables in a single statement.

For example,

```
>>> x, y, st=3, 4.2, "Hello"
>>> print("x= ", x, " y= ",y, " st= ", st)
x=3          y=4.2      st=Hello
```

Python supports bitwise operators like &(AND), |(OR), ~(NOT), ^(XOR), >>(right shift) and <<(left shift). These operators will operate on every bit of the operands. Working procedure of these operators is same as that in other languages like C and C++.

There are some special operators in Python viz. *Identity operator* (is and is not) and *membership operator* (in and not in). These will be discussed in further Modules.

❖ Expressions

A combination of values, variables and operators is known as expression.

Following are few examples of expression –

```
x=5
y=x+10
z= x-y*3
```

The Python interpreter evaluates simple expressions and gives results even without *print()*.

For example,

```
>>> 5
5           #displayed as it is
>>> 1+2
3           #displayed the sum
```

But, such expressions do not have any impact when written into Python script file.

1.8 Order of Operations

When an expression contains more than one operator, the evaluation of operators depends on the *precedence of operators*.

The Python operators follow the precedence rule (which can be remembered as *PEMDAS*) as given below –

Parenthesis have the highest precedence in any expression. The operations within parenthesis will be evaluated first. For example, in the expression $(a+b)*c$, the addition has to be done first and then the sum is multiplied with c .

Exponentiation has the 2nd precedence. But, it is right associative. That is, if there are two exponentiation operations continuously, it will be evaluated from right to left (unlike most of other operators which are evaluated from left to right).

For example,

```
>>>print(2**3)      #It is 23 8
>>>print(2**3**2) #It is 512 i.e., 232
```

Multiplication and Division are the next priority. Out of these two operations, whichever comes first in the expression is evaluated.

```
>>> print(5*2/4) #multiplication and then division
2.5
>>> print(5/4*2) #division and then multiplication
2.5
```

Addition and Subtraction are the least priority. Out of these two operations, whichever appears first in the expression is evaluated i.e., they are evaluated from left to right

❖ String Operations

String concatenation can be done using + operator as shown below –

```
>>> x="32"
>>> y="45"
>>> print(x+y)
3245
```

Observe the output: here, the value of y (a string “45”, but not a number 45) is placed just in front of value of x(a string “32”). Hence the result would be “3245” and its type would be *string*.

NOTE: One can use single quotes to enclose a string value, instead of double quotes.

❖ Asking the User for Input

Python uses the built-in function *input()* to read the data from the keyboard.

When this function is invoked, the user-input is expected. The input is read till the user presses enter- key.

For example:

```
>>> str1=input()
Hello how are you?           #user input
>>> print("String is ",str1)
String is Hello how are you?#printing str1
```

When *input()* function is used, the cursor will be blinking to receive the data.

For a better understanding, it is better to have a prompt message for the user informing what needs to be entered as input.

The *input()* function itself can be used to do so, as shown below –

```
>>> str1=input("Enter a string: ")
Enter a string: Hello
>>> print("You have entered:",str1)
You have entered: Hello
```

One can use new-line character \n in the function *input()* to make the cursor to appear in the nextline of prompt message –


```
>>> str1=input("Enter a string:\n")
Enter a string:
Hello                               #cursor is pushed here
```

The key-board input received using *input()* function is always treated as a string type.

If you need an integer, you need to convert it using the function *int()*.

Observe the following example –

```
>>> x=input("Enter x:")
Enter x:10                           #x takes the value "10", but not
10
>>> type(x)                           #So, type of x would be str
<class 'str'>
>>> x=int(input("Enter x:"))          #use int()
Enter x:10
>>> type(x)                           #Now, type of x is int
<class 'int'>
```

A function *float()* is used to convert a valid value enclosed within quotes into float number as shown below –

```
>>> f=input("Enter a float value:") Enter a float
value: 3.5
>>> type(f)
<class 'str'> #f is actually a string "3.5"
>>> f=float(f) #converting "3.5" into float value 3.5
>>> type(f)
<class 'float'>
```

A function *chr()* is used to convert an integer input into equivalent ASCII character.

```
>>> a=int(input("Enter an integer:"))
Enter an integer:65
>>> ch=chr(a)
>>> print("Character Equivalent of ", a, "is ",ch)
Character Equivalent of 65 is A
```

There are several such other utility functions in Python, which will be discussed later.

1.9 Comments

It is a good programming practice to add comments to the program wherever required.

This will help someone to understand the logic of the program. Comment may be in a single line or spread into multiple lines. A single-line comment in Python starts with the symbol #. Multiline comments are enclosed within a pair of 3-single quotes.

```
Ex1. #This is a single-line comment
```

```
Ex2.     ''' This is a
           Multi line
           Comment'''
```

Python (and all programming languages) ignores the text written as comment lines.

They are only for the programmer's (or any reader's) reference.

❖ Choosing Mnemonic Variable Names

Choosing an appropriate name for variables in the program is always at stake.

Consider the following examples –

```
Ex1.     a=10000
           b=0.3*a
           c=a+b
           print(c)  #output is 13000
```

```
Ex2.     basic=10000
           da=0.3*basic
           gross_sal=basic+da
           print("Gross Sal = ",gross_sal) #output is 13000
```

One can observe that both of these two examples are performing same task.

But, compared to Ex1, the variables in Ex2 are indicating what is being calculated.

That is, variable names in Ex2 are indicating the purpose for which they are being used in the program. Such variable names are known as *mnemonic variable names*. The word

mnemonic means *memory aid*. The mnemonic variables are created to help the programmer to remember the purpose for which they have been created.

Python can understand the set of reserved words (or keywords), and hence it flashes an error when such words are used as variable names by the programmer.

Moreover, most of the Python editors have a mechanism to show keywords in a different color. Hence, programmer can easily make out the keyword immediately when he/she types that word.

❖ Debugging

Some of the common errors a beginner programmer may make are syntax errors.

Though Python flashes the error with a message, sometimes it may become hard to understand the cause of errors. Some of the examples are given here –

Ex1: >>> avg sal=10000
 SyntaxError: invalid syntax
 Here, there is a space between the terms avg and sal, which is not allowed.

Ex2: >>>m=09
 SyntaxError: invalid token
 Python does not allow preceding zeros for numeric values.

Ex3. >>> basic=2000
 >>> da=0.3*Basic
 NameError: name 'Basic' is not defined
 As Python is case sensitive, basic is different from Basic.

As shown in above examples, the syntax errors will be alerted by Python. But, programmer is responsible for logical errors or semantic errors. Because, if the program does not yield into expected output, it is due to mistake done by the programmer, about which Python is unaware of.

One can observe from previous few examples that when a runtime error occurs, it displays a term *Traceback* followed by few indications about errors.

A *traceback* is a stack trace from the point of error-occurrence down to the call-sequence till the point of call.

This is helpful when we start using functions and when there is a sequence of multiple function calls from one to other.

Then, traceback will help the programmer to identify the exact position where the error occurred.

Most useful part of error message in traceback are –

What kind of error it is

Where it occurred

Compared to runtime errors, syntax errors are easy to find, most of the times. But, *whitespace* errors in syntax are quite tricky because spaces and tabs are invisible.

For example –

```
>>> x=10
>>>   y=15
      SyntaxError: unexpected indent
```

The error here is because of additional space given before `y`. As Python has a different meaning (separate block of code) for indentation, one cannot give extra spaces as shown above.

In general, error messages indicate where the problem has occurred. But, the actual error may be before that point, or even in previous line of code.

1.10 FUNCTIONS

Functions are the building blocks of any programming language.

A sequence of instructions intended to perform a specific independent task is known as a *function*.

In this section, we will discuss various types of built-in functions, user-defined functions, applications/uses of functions etc.

❖ Function Calls

A function is a named sequence of instructions for performing a task.

When we define a function we will give a valid name to it, and then specify the instructions for performing required task.

Later, whenever we want to do that task, a function is *called* by its name.

Consider an example:

```
>>> type(15)
<class 'int'>
```

Here *type* is a function name, 15 is the argument to a function and <class 'int'> is the result of the function.

Usually, a function *takes* zero or more arguments and *returns* the result.

❖ Built-in Functions

Python provides a rich set of built-in functions for doing various tasks.

The programmer/user need not know the internal working of these functions; instead, they need to know only the purpose of such functions.

Some of the built in functions are given below –

- ❖ **max()**: This function is used to find maximum value among the arguments. It can be used for numeric values or even to strings.

```
>>>max(10, 20, 14, 12)      #maximum of 4 integers
20
>>>max("hello world")
'w'                        #character having maximum ASCII code
>>>max(3.5, -2.1, 4.8, 15.3, 0.2)
15.3                       #maximum of 5 floating point values
```

- ❖ **min()**: As the name suggests, it is used to find minimum of arguments.

```
>>>min(10, 20, 14, 12)    #minimum of 4 integers
```

```

10
>>>min("hello world")
' '           #space has least ASCII code here
>>>min(3.5, -2.1, 4.8, 15.3, 0.2)
-2.1         #minimum of 5 floating point values

```

- ❖ **len():** This function takes a single argument and finds its length. The argument can be a string, list, tuple etc.

```

>>>len("hello how are you?")
18

```

There are many other built-in functions available in Python. They are discussed in further Modules, wherever they are relevant.

1.11 TYPE CONVERSION FUNCTIONS

As we have seen earlier (while discussing *input()* function), the type of the variable/value can be converted using functions *int()*, *float()*, *str()*.

Python provides built-in functions that convert values from one type to another

Consider following few examples –

```

>>>int('20')      #integer enclosed within single quotes
                  #converted to integer type
>>>int("20")     #integer enclosed within double quotes
20
>>>int("hello")  #actual string cannot be converted to int

```

Traceback (most recent call last):

```

File "<pyshell#23>", line
1, in <module>
int("hello") ValueError:
invalid literal for int()
with base 10: 'hello'

```

```
>>>int(3.8)    #float value being converted to integer
3              #round-off will not happen, fraction is ignored

>>>int(-5.6)
-5

>>>float('3.5')    #float enclosed within single quotes
3.5              #converted to float type
>>>float(42)        #integer is converted to float
42.0

>>>str(4.5)        #float converted to string
'4.5'

>>>str(21)         #integer converted to string
'21'
```

❖ Random Numbers

Most of the programs that we write are *deterministic*.

That is, the input (or range of inputs) to the program is pre-defined and the output of the program is one of the expected values.

But, for some of the real-time applications in science and technology, we need randomly generated output. This will help in simulating certain scenario.

Random number generation has important applications in games, noise detection in electronic communication, statistical sampling theory, cryptography, political and business prediction etc. These applications require the program to be *nondeterministic*.

There are several algorithms to generate random numbers. But, as making a program completely *nondeterministic* is difficult and may lead to several other consequences, we generate *pseudo-random numbers*.

That is, the type (integer, float etc) and range (between 0 and 1, between 1 and 100 etc) of the random numbers are decided by the programmer, but the actual numbers are unknown.

Moreover, the algorithm to generate the random number is also known to the programmer. Thus, the random numbers are generated using deterministic computation and hence, they are known as pseudo-random numbers!!

Python has a module *random* for the generation of random numbers. One has to *import* this module in the program. The function used is also *random()*.

By default, this function generates a random number between 0.0 and 1.0 (excluding 1.0).

For example–

```
import random                #module random is imported
print(random.random())      #random() function is
invoked
0.7430852580883088         #a random number generated

print(random.random())
0.5287778188896328        #one more random number
```

Importing a module creates an object.

Using this object, one can access various functions and/or variables defined in that module. Functions are invoked using a dot operator.

There are several other functions in the module *random* apart from the function *random()*. (Do not get confused with module name and function name. Observe the parentheses while referring a function name).

Few are discussed hereunder:

❖ **randint():** It takes two arguments *low* and *high* and returns a random integer between these two arguments (both *low* and *high* are inclusive).

For example,

```
>>>random.randint(2,20)
14                #integer between 2 and 20 generated
>>> random.randint(2,20) 10
```

❖ **choice():** This function takes a sequence (a *list* type in Python) of numbers as an argument and returns one of these numbers as a random number. For example,

```
>>> t=[1,2, -3, 45, 12, 7, 31, 22] #create a list t
```



```
>>> random.choice(t)           #t is argument to choice()
12                              #one of the elements in t

>>> random.choice(t)
1                               #one of the elements in t
```

Various other functions available in *random* module can be used to generate random numbers following several probability distributions like Gaussian, Triangular, Uniform, Exponential, Weibull, Normal etc.

1.12 MATH FUNCTIONS

Python provides a rich set of mathematical functions through the module *math*. To use these functions, the *math* module has to be imported in the code. Some of the important functions available in *math* are given hereunder

- ❖ **sqrt():** This function takes one numeric argument and finds the square root of that argument.

```
>>> math.sqrt(34)                #integer argument
5.830951894845301

>>> math.sqrt(21.5)             #floating point argument
4.636809247747852
```

- ❖ **pi:** The constant value *pi* can be used directly whenever we require.

```
>>> print (math.pi)
3.141592653589793
```

- ❖ **log10():** This function is used to find logarithm of the given argument, to the base 10.

```
>>> math.log10(2)
0.3010299956639812
```

- ❖ **log():** This is used to compute natural logarithm (base e) of a given number.

```
>>> math.log(2)
0.6931471805599453
```

- ❖ **sin():** As the name suggests, it is used to find *sine* value of a given argument. Note that, the argument must be in radians (not degrees). One can convert the number of degrees into radians by multiplying $\pi/180$ as shown below –

```
>>>math.sin(90*math.pi/180)           #sin(90) is 1
1.0
```

- ❖ **cos():** Used to find *cosine* value –

```
>>>math.cos(45*math.pi/180)
0.7071067811865476
```

- ❖ **tan():** Function to find tangent of an angle, given as argument.

```
>>> math.tan(45*math.pi/180)
0.9999999999999999
```

- ❖ **pow():** This function takes two arguments x and y , then finds x to the power of y .

```
>>> math.pow(3,4) 81.0
```

1.13 COMPOSITION

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them. One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

```
>>> minutes = hours * 60          # right
>>> hours * 60 = minutes          # wrong!
```

SyntaxError: can't assign to operator

1.14 ADDING NEW FUNCTIONS (USER-DEFINED FUNCTIONS)

Python facilitates programmer to define his/her own functions.

The function written once can be used wherever and whenever required.

The syntax of user-defined function would be –

```
def fname(arg_list):
    statement_1
    statement_2
    ...
    ...
    ...
    ...
    ...
    Statement_n
    return value
```

Here *def* is a keyword indicating it as a function definition.

Fname is any valid name given to the function

arg_list is list of arguments taken by a function. These are treated as inputs to the function from the position of function call. There may be zero or more arguments to a function.

statements are the list of instructions to perform required task.

return is a keyword used to return the output *value*. This statement is optional

The first line in the function *def* fname(arg_list) is known as *function header/definition*. The remaining lines constitute *function body*.

The function header is terminated by a colon and the function body must be indented. To come out of the function, indentation must be terminated.

Unlike few other programming languages like C, C++ etc, there is no *main()* function or specific location where a user-defined function has to be called.

The programmer has to invoke (call) the function wherever required.

Consider a simple example of user-defined function –

```
def myfun():
    print("Hello")
    print("Inside the function")

print("Example of function")
myfun()
print("Example over")
```

Observe indentation. Statements outside the function without indentation.

myfun() is called here.

The output of above program would be –

```
Example of function
Hello
Inside the function
Example over
```

The function definition creates an object of type *function*.

In the above example, *myfun* is internally an object.

This can be verified by using the statement –

```
>>> print(myfun)                # myfun without parenthesis
<function myfun at 0x0219BFA8>
>>> type(myfun)                  # myfun without parenthesis
```

```
<class 'function'>
```

Here, the first output indicates that myfun is an object which is being stored at the memory address 0x0219BFA8 (0x indicates octal number).

The second output clearly shows myfun is of type function.

(NOTE: In fact, in Python every type is in the form of class. Hence, when we apply *type* on any variable/object, it displays respective class name. The detailed study of classes will be done in Module 4.)

❖ Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks

like this:

```
def print_lyrics():  
    print("Jhony Jhony.")  
    print("Yes Pappa.")  
def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()  
    repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.

1.15 FLOW OF EXECUTION

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don't run until the function is called. A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

The *flow of execution* of every program is sequential from top to bottom, a function can be invoked only after defining it.

Usage of function name before its definition will generate error. Observe the following code:

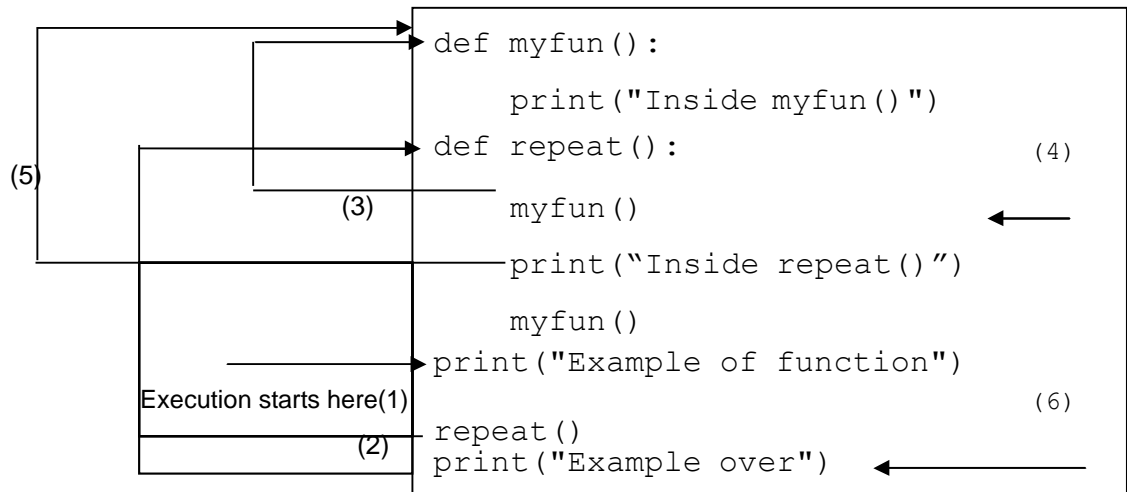
```
>>>print("Example of function")
>>>myfun()      #function call before definition
    print("Example over")

>>>def myfun():    #function definition is here
    print("Hello")
>>>print("Inside the function")
```

The above code would generate error
saying `NameError: name
'myfun' is not defined`

Functions are meant for code-reusability. That is, a set of instructions written as a function need not be repeated. Instead, they can be called multiple times whenever required.

Consider the enhanced version of previous program as below –



The output is –

```

Example of function
Inside myfun()
Inside repeat()
Inside myfun()
Example over

```

Observe the output of the program to understand the flow of execution of the program.

Initially, we have two function definitions myfun() and repeat() one after the other. But, functions are not executed unless they are called (or invoked). Hence, the first line to execute in the above program is –

```
print("Example of function")
```

Then, there is a function call repeat(). So, the program control jumps to this function. Inside repeat(), there is a call for myfun().

Now, program control jumps to myfun() and executes the statements inside and returns back to repeat() function. The statement print("Inside repeat()") is executed.

Once again there is a call for myfun()function and hence, program control jumps there. The function myfun() is executed and returns to repeat().

As there are no more statements in repeat(), the control returns to the original position of its call. Now there is a statement print("Example over")to execute, and program is terminated.

1.16 PARAMETERS AND ARGUMENTS

In the previous section, we have seen simple example of a user-defined function, where the function was without any argument.

But, a function may take arguments as an input from the calling function.

Consider an example of a function which takes a single argument as below –

```
def test(var):  
    print("Inside test()")  
    print("Argument is ",var)  
  
print("Example of function with arguments")  
x="hello"  
test(x)  
y=20  
test(y)  
print("Over!!")
```

The output would be –

```
Example of function with arguments  
Inside test()  
Argument is hello  
Inside test()  
Argument is 20 Over!!
```

In the above program, var is called as *parameter* and x and y are called as *arguments*.

The argument is being passed when a function test() is invoked. The parameter receives the argument as an input and statements inside the function are executed.

As Python variables are not of specific data types in general, one can pass any type of value to the function as an argument.

Python has a special feature of applying multiplication operation on arguments while passing them to a function. Consider the modified version of above program –

```
>>>def test(var):
    print("Inside test()")
    print("Argument is ",var)

>>>print("Example of function with arguments")

x="hello"
>>>test(x*3)

>>>y=20
>>>test(y*3)

>>>print("Over!!")
```

The output would be –

```
Example of function with arguments
    Inside test()
    Argument is hellohellohello           #observe repetition
    Inside test()
    Argument is 60                         #observe multiplication
    Over!!
```

One can observe that, when the argument is of type *string*, then multiplication indicates that string is repeated 3 times.

Whereas, when the argument is of numeric type (here, integer), then the value of that argument is literally multiplied by 3.

1.17 VARIABLES AND PARAMETERS ARE LOCAL

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Twinkle Twinkle '
>>> line2 = 'Little Star.'
>>> cat_twice(line1, line2)
Twinkle Twinkle Little Star.
Twinkle Twinkle Little Star.
```

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

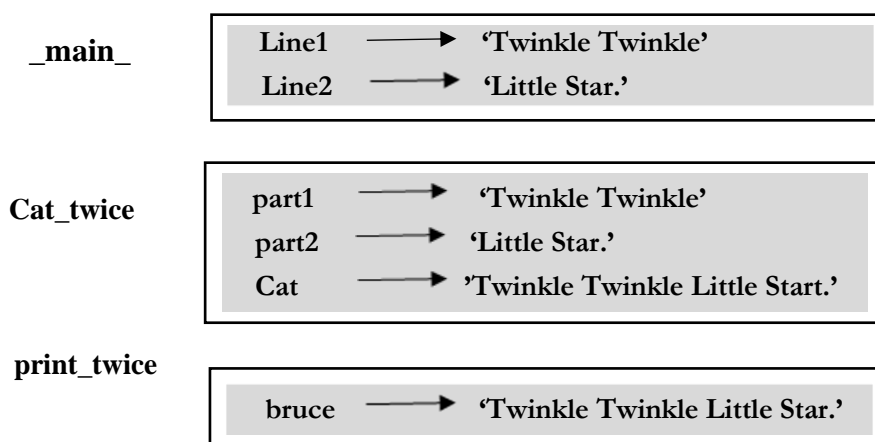


Figure 1.4: Stack diagram.

```
print(cat)
```

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

```
Parameters are also local.
```

```
For example, outside print_twice, there is no such thing as
bruce
```

1.18 STACK DIAGRAMS

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to. Each function is represented by a **frame**.

A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 1.4. The frames are arranged in a stack that indicates which function called which, and so

on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`. Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.

If an error occurs during a function call, Python prints the name of the function, the name of the function that called it, and the name of the function that called *that*, all the way back to `__main__`. For example, if you try to access `cat` from within `print_twice`, you get a `NameError`:

```
Traceback (innermost last):
File "test.py", line 13, in __main__
  cat_twice(line1, line2)
File "test.py", line 5, in cat_twice
  print_twice(cat)
File "test.py", line 9, in print_twice
  print(cat)
NameError: name 'cat' is not defined
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

1.19 FRUITFUL FUNCTIONS AND VOID FUNCTIONS

A function that performs some task, but do not return any value to the calling function is known as ***void function***. The examples of user-defined functions considered till now are void functions.

The function which returns some result to the calling function after performing a task is known as *fruitful function*. The built-in functions like mathematical functions, random number generating functions etc. that have been considered earlier are examples for fruitful functions.

One can write a user-defined function so as to return a value to the calling function as shown in the following example –

```
def sum(a,b):  
    return a+b  
  
x=int(input("Enter a number:"))  
y=int(input("Enter another number:"))  
  
s=sum(x,y)  
print("Sum of two numbers:",s)
```

The sample output would be –

Enter a number:3

Enter another number:4

Sum of two numbers: 7

In the above example, The function sum() take two arguments and returns their sum to the receiving variable s.

When a function returns something and if it is not received using a LHS variable, then the return value will not be available.

For instance, in the above example if we just use the statement sum(x,y) instead of s=sum(x,y), then the value returned from the function is of no use.

On the other hand, if we use a variable at LHS while calling void functions, it will receive None. For example,

```
p= test(var)           #function used in previous example  
  
print(p)
```

Now, the value of p would be printed as None. Note that, None is not a string, instead it is of typeclass 'NoneType'. This type of object indicates *no value*.

❖ Why Functions?

Functions are essential part of programming because of following reasons –

Creating a new function gives the programmer an opportunity to name a group of statements, which makes the program easier to read, understand, and debug.

Functions can make a program smaller by eliminating repetitive code. If any modification is required, it can be done only at one place.

Dividing a long program into functions allows the programmer to debug the independent functions separately and then combine all functions to get the solution of original problem.

Well-designed functions are often useful for many programs. The functions written once for a specific purpose can be re-used in any other program.

✚ FOR THE CURIOUS MINDS (Something Beyond The Syllabus)

❖ *Special parameters of print() – sep and end :*

Consider an example of printing two values using *print()* as below –

```
>>> x=10
>>> y=20
>>> print(x,y)
10 20          #space is added between two values
```

Observe that the two values are separated by a space without mentioning anything specific. This is possible because of the existence of an argument *sep* in the *print()* function whose default value is white space. This argument makes sure that various values to be printed are separated by a space for a better representation of output.

The programmer has a liberty in Python to give any other character (or string) as a separator by explicitly mentioning it in *print()* as shown below –

```
>>> print("03", "06", "2021", sep='-')
03-06-2021
```

We can observe that the values have been separated by hyphen, which is given as a value

for the argument *sep*. Consider one more example –

```
>>> college="BGSIT"
>>> address="BGNAGARA"
>>> print(college, address, sep='@')
BGSIT@BGNAGARA
```

If you want to deliberately suppress any separator, then the value of *sep* can be set with empty string as shown below –

```
>>> print("Hello", "World", sep='')
HelloWorld
```

You might have observed that in Python program, the *print()* adds a new line after printing the data. In a Python script file, if you have two statements like –

```
print("Hello")
print("World")
```

then, the output would be

```
Hello World
```

This may be quite unusual for those who have experienced programming languages like C, C++ etc. In these languages, one has to specifically insert a new-line character (`\n`) to get the output in different lines. But, in Python without programmer's intervention, a new line will be inserted. This is possible because, the *print()* function in Python has one more special argument *end* whose default value itself is new-line. Again, the default value of this argument can be changed by the programmer as shown below (Run these lines using a script file, but not in the terminal/command prompt) –

```
print("Hello", end= „@“)
print("World")
```

The output would be –

```
Hello@World
```

❖ Formatting the output:

There are various ways of formatting the output and displaying the variables with a required number of space-width in Python. We will discuss few of them with the help of examples.

Ex1: When multiple variables have to be displayed embedded within a string, the *format()*

function is useful as shown below –

```
>>> x=10

>>> y=20

>>> print("x={0}, y={1}".format(x,y))
x=10, y=20
```

While using *format()* the arguments of *print()* must be numbered as 0, 1, 2, 3, etc. and they must be provided inside the *format()* in the same order.

Ex2: The *format()* function can be used to specify the width of the variable (the number of spaces that the variable should occupy in the output) as well. Consider below given example which displays a number, its square and its cube.

```
for x in range(1,5):
    print("{0:1d} {1:3d} {2:4d}".format(x,x**2, x**3))
```

Output:

```
1   1   1
2   4   8
3   9  27
4  16  64
```

Here, 1d, 3d and 4d indicates 1-digit space, 2-digit space etc. on the output screen.

Ex3: One can use % symbol to have required number of spaces for a variable. This will be useful in printing floating point numbers.

```
>>> x=19/3

>>> print(x)
```

```
6.3333333333333333 #observe number of digits after dot
>>> print("%.3f"%(x)) #only 3 places after decimal point 6.333
>>> x=20/3
>>> y=13/7
>>> print("x= ",x, "y=",y) #observe actual digits
x=6.666666666666667 y= 1.8571428571428572
>>> print("x=%0.4f, y=%0.2f"%(x,y))
x=6.6667, y=1.86 #observe rounding off digits
```


MODULE 2

2.1 CONDITIONAL EXECUTION

In general, the statements in a program will be executed sequentially. But, sometimes we need a set of statements to be executed based on some conditions. Such situations are discussed in this section.

2.1.1 Floor division and modulus

The floor division operator, `//`, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, rounding down:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

To get the remainder, you could subtract off one hour in minutes:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

An alternative is to use the modulus operator, `%`, which divides two numbers and returns the remainder.

```
>>> remainder = minutes % 60
>>> remainder
45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`. Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly `x % 100` yields the last

two digits. If you are using Python 2, division works differently. The division operator, /, performs floor division if both operands are integers, and floating-point division if either operand is a float.

2.1.2 Boolean Expressions

A *Boolean Expression* is an expression which results in *True* or *False*. The *True* and *False*

are special values that belong to class **bool**. Check the following –

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Boolean expression may be as below –

```
>>> 10==12
False
>>> x=10
>>> y=10
>>> x==y
True
```

Various comparison operations are shown in Table 2.1.

Table 2.1 Relational (Comparison) Operators

Operator	Meaning	Example
>	Greater than	a>b
<	Less than	a=	Greater than or equal to	a>=b
<=	Less than or equal to	a<=b
==	Comparison	a==b
!=	Not equal to	a !=b
is	Is same as	a is b
is not	Is not same as	a is not b

Few Examples:

```
>>> a=10
>>> b=20
>>> x= a>b
>>> print(x)
False
>>> print(a==b)
False
>>> print("a<b is ", a<b)
a<b is True
>>> print("a!=b is", a!=b)
a!=b is True
```

```
>>> 10 is 20
False
>>> 10 is 10
True
```

NOTE: For a first look, the operators `==` and `is` look same. Similarly, the operators `!=` and `is not` look the same. But, the operators `==` and `!=` does the **equality test**. That is, they will compare the values stored in the variables. Whereas, the operators `is` and `is not` does the **identity test**. That is, they will compare whether two objects are same. Usually, two objects are same when their memory locations are same. This concept will be more clear when we take up classes and objects in Python.

2.1.3 Logical Operators

There are 3 logical operators in Python as shown in Table 1.4. (NOTE that symbols like `&&`, `||` are not used in Python for representing logical operators)

Table 2.2 Logical Operators

Operator	Meaning	Example
<code>and</code>	Returns true, if both operands are true	<code>a and b</code>
<code>or</code>	Returns true, if any one of two operands is true	<code>a or b</code>
<code>not</code>	Return true, if the operand is false (it is a unary operator)	<code>not a</code>

NOTE:

1. Logical operators treat the operands as Boolean (True or False).
2. Python treats any non-zero number as True and zero as False.
3. While using `and` operator, if the first operand is False, then the second operand is not evaluated by Python. Because *False and'ed* with anything is False.
4. In case of `or` operator, if the first operand is True, the second operand is not evaluated. Because *True or'ed* with anything is True.

Example 1 (with Boolean Operands):

```
>>> x= True
>>> y= False
>>> print('x and y is', x and y)
x and y is False
>>> print('x or y is', x or y)
x or y is True
>>> print('Complement of x is ', not x)
Complement of x is False
```

Example 2 (With numeric Operands):

```
>>> a=-3
>>> b=10
>>> print(a and b)           #and operation
```

```

10      #a is true, hence b is evaluated and printed

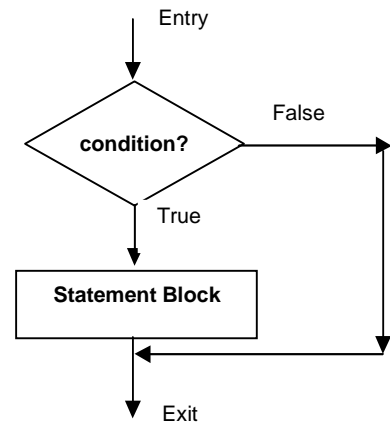
>>> print(a or b)      #or operation
-3      #a is true, hence b is not evaluated
>>> print(0 and 5)    #0 is false, so printed
0
    
```

2.1.4 Conditional Execution

The basic level of conditional execution can be achieved in Python by using *if* statement. The syntax and flowcharts are as below –

```

if condition:
    Statement block
    
```



Observe the colon symbol after *condition*. When the *condition* is true, the *Statement block* will be executed. Otherwise, it is skipped. A set (block) of statements to be executed under *if* is decided by the indentation (tab space) given.

Consider an example –

```

>>> x=10
>>> if x<40:
    print("Fail") #observe indentation after if
    
```

Fail #output

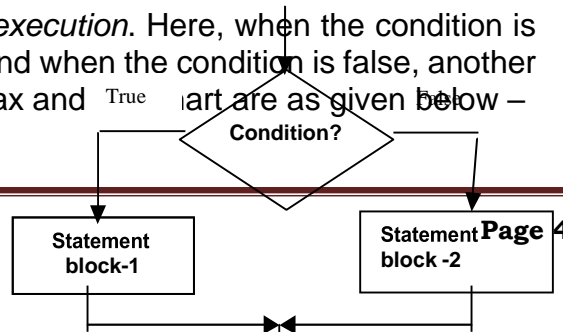
Usually, the *if* conditions have a statement block. In any case, the programmer feels to do nothing when the condition is true, the statement block can be skipped by just typing **pass** statement as shown below –

```

>>> if x<0:
    pass #do nothing when x is negative
    
```

2.1.5 Alternative Execution

A second form of *if* statement is *alternative execution*. Here, when the condition is true, one set of statements will be executed and when the condition is false, another set of statements will be executed. The syntax and flowchart are as given below –



```

if condition:
    Statement block -1
else:
    Statement block -2
    
```

As the *condition* will be either true or false, only one among *Statement block-1* and *Statement block-2* will be get executed. Thesetwo alternatives are known as **branches**.

Example:

```

x=int(input("Enter x:"))
if x%2==0:
    print("x is even")
else:
    print("x is odd")
    
```

Sample output:

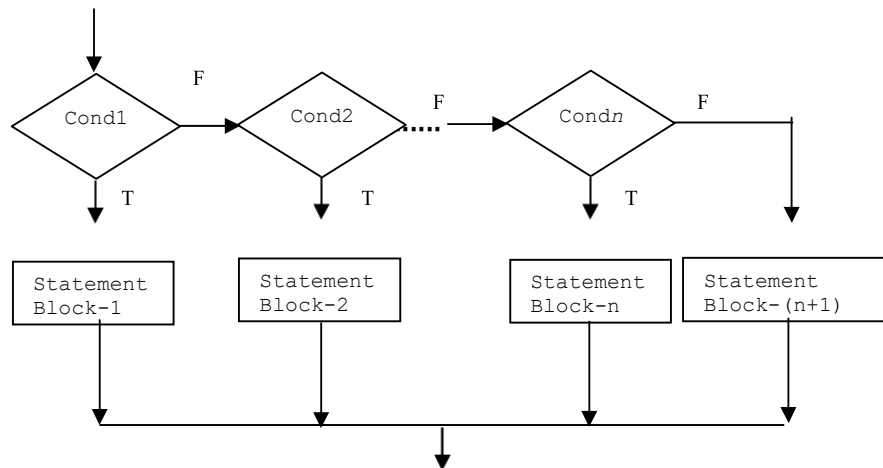
Enter x: 13x is odd

2.1.6 Chained Conditionals

Some of the programs require more than one possibility to be checked for executing a setof statements. That means, we may have more than one branch. This is solved with the help of *chained conditionals*. The syntax and flowchart is given below –

```

if condition1:
    Statement Block-1
elif condition2:
    Statement Block-2
    |
    |
    |
elif condition_n:
    Statement Block-n
else:
    Statement Block-(n+1)
    
```



The conditions are checked one by one sequentially. If any condition is satisfied, the respective statement block will be executed and further conditions are not checked. Note that, the last *e/else* block is not necessary always.

Example:

```
marks=float(input("Enter marks:"))
if marks >= 80:
    print("First Class with Distinction")
elif marks >= 60 and marks < 80:
    print("First Class")
elif marks >= 50 and marks < 60:
    print("Second Class")
elif marks >= 35 and marks < 50:
    print("Third Class")
else:
    print("Fail")
```

Sample Output:

```
Enter marks: 78
First Class
```

2.1.7 Nested Conditionals

The conditional statements can be nested. That is, one set of conditional statements can be nested inside the other. It can be done in multiple ways depending on programmer's requirements. Examples are given below –

Ex1.

```
marks=float(input("Enter marks:"))
if marks>=60:
    if marks<70:
        print("First Class")
    else:
        print("Distinction")
```

Sample Output:

```
Enter marks:68First Class
```

Here, the outer condition `marks>=60` is checked first. If it is true, then there are two branches for the inner conditional. If the outer condition is false, the above code does nothing.

Ex2.

```
gender=input("Enter gender:")
age=int(input("Enter age:"))

if gender == "M" :if age >= 21:
    print("Boy, Eligible for Marriage")
else:
    print("Boy, Not Eligible for Marriage")
elif gender == "F":
    if age >= 18:
        print("Girl, Eligible for Marriage")
else:
    print("Girl, Not Eligible for Marriage")
```

Sample Output:

```
Enter gender: FEnter age: 17
Girl, Not Eligible for Marriage
```

NOTE: Nested conditionals make the code difficult to read, even though there are proper indentations. Hence, it is advised to use logical operators like *and* to simplify the nested conditionals. For example, the outer and inner conditions in **Ex1** above can be joined as -

```
if marks>=60 and marks<70:#do something
```

2.1.8 Catching Exceptions using try and except

As discussed in Section 1.1.11, there is a chance of runtime error while doing some program. One of the possible reasons is wrong input. For example, consider the following code segment –

```
a=int(input("Enter a:"))
b=int(input("Enter b:"))c=a/b
print(c)
```

When you run the above code, one of the possible situations would be –

```
Enter a:12Enter b:0
Traceback (most recent call last):
  File "C:\Users\Manojkumar\Python\Python39\p1.py",
line 154,in <module>
    c=a/b
ZeroDivisionError: division by zero
```

For the end-user, such type of system-generated error messages is difficult to handle. So the code which is prone to runtime error must be executed conditionally within *try* block. The *try* block contains the statements involving suspicious code and the *except* block contains the possible remedy (or instructions to user informing what went wrong and what could be the way to get out of it). If something goes wrong with the statements inside *try* block, the *except* block will be executed. Otherwise, the except-block will be skipped. Consider the example –

```
a=int(input("Enter a:"))
b=int(input("Enter b:"))try:
    c=a/b print(c)
except:
    print("Division by zero is not possible")
```

Output:

```
Enter a:12Enter b:0
Division by zero is not possible
```

Handling an exception using *try* is called as **catching** an exception. In general, catching an exception gives the programmer to fix the probable problem, or to try again or at least to end the program gracefully.

2.1.9 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

If *n* is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs *n* and then calls a function named `countdown`—itself—passing *n*-1 as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

The execution of `countdown` begins with *n*=3, and since *n* is greater than 0, it outputs the value 3, and then calls itself...

The execution of `countdown` begins with *n*=2, and since *n* is greater than 0, it outputs the value 2, and then calls itself...

The execution of `countdown` begins with *n*=1, and since *n* is greater than 0, it outputs the value 1, and then calls itself...

The execution of `countdown` begins with *n*=0, and since *n* is not greater than 0, it outputs the word, “Blastoff!” and then returns.

The `countdown` that got *n*=1 returns.

The `countdown` that got *n*=2 returns.

The `countdown` that got *n*=3 returns.

And then you’re back in `__main__`. So, the total output looks like this:

```
3
2
1
```


Blastoff!

A function that calls itself is **recursive**; the process of executing it is called **recursion**. As another example, we can write a function that prints a string n times.

```
def print_n(s, n):  
    if n <= 0:  
        return  
    print(s)  
    print_n(s, n-1)
```

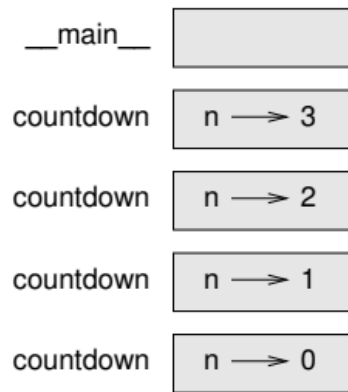


Figure 2.1: Stack Diagram of Recursion

If $n \leq 0$ the **return statement** exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

The rest of the function is similar to `countdown`: it displays `s` and then calls itself to display $n - 1$ additional times. So the number of lines of output is $1 + (n - 1)$, which adds up to n .

For simple examples like this, it is probably easier to use a for loop. But we will see examples later that are hard to write with a for loop and easy to write with recursion, so it is good to start early.

2.2 ITERATION

Iteration is a processing repeating some task. In a real time programming, we require a set of statements to be repeated certain number of times and/or till a condition is met. Every programming language provides certain constructs to achieve the repetition of tasks. In this section, we will discuss various such looping structures.

2.2.2 The *while* Statement

The *while* loop has the syntax as below –

```
while condition:
    statement_1
    statement_2
    .....
    statement_n

statements_after_while
```

Here, **while** is a keyword. The `condition` is evaluated first. Till its value remains true, the `statement_1` to `statement_n` will be executed. When the `condition` becomes false, the loop is terminated and statements after the loop will be executed. Consider an example –

```
n=1
while n<=5:
    print(n)    #observe indentation
    n=n+1

print("over")
```

The output of above code segment would be –

```
1
2
3
4
5
over
```

In the above example, a variable `n` is initialized to 1. Then the condition `n<=5` is being checked. As the condition is true, the block of code containing print statement (`print(n)`) and increment statement (`n=n+1`) are executed. After these two lines, condition is checked again. The procedure continues till condition becomes false, that is when `n` becomes 6. Now, the while-loop is terminated and next statement after the loop will be executed. Thus, in this example, the loop is *iterated* for 5 times.

Note that, a variable `n` is initialized before starting the loop and it is incremented

inside the loop. Such a variable that changes its value for every iteration and controls the total execution of the loop is called as **iteration variable** or **counter variable**. If the count variable is not updated properly within the loop, then the loop may not terminate and keeps executing infinitely.

2.2.3 Infinite Loops, *break* and *continue*

A loop may execute infinite number of times when the condition is never going to become false. For example,

```
n=1
while True:
    print(n) n=n+1
```

Here, the condition specified for the loop is the constant `True`, which will never get terminated. Sometimes, the condition is given such a way that it will never become false and hence by restricting the program control to go out of the loop. This situation may happen either due to wrong condition or due to not updating the counter variable.

In some situations, we deliberately want to come out of the loop even before the normal termination of the loop. For this purpose **break** statement is used. The following example depicts the usage of *break*. Here, the values are taken from keyboard until a negative number is entered. Once the input is found to be negative, the loop terminates.

```
while True:
    x=int(input("Enter a number:"))if x>= 0:
        print("You have entered ",x)else:
        print("You have entered a negative number!!")
        break          #terminates the loop
```

Sample output:

```
Enter a number:23 You have entered 23Enter a number:12
You have entered 12Enter a number:45 You have entered 45
Enter a number:0 You have entered 0 Enter a number:-2
You have entered a negative number!!
```

In the above example, we have used the constant `True` as condition for while-loop, which will never become false. So, there was a possibility of infinite loop. This has been avoided by using *break* statement with a condition. The condition is kept inside the loop such a way that, if the user input is a negative number, the loop terminates. This indicates that, the loop may terminate with just one iteration (if user gives negative number for the very first time) or it may take thousands of iteration (if user keeps on giving only positive numbers as input). Hence, the number of iterations here is unpredictable. But, we are making sure that it will not be an infinite-loop, instead, the user has control on the loop.

Sometimes, programmer would like to move to next iteration by skipping few

statements in the loop, based on some condition. For this purpose *continue* statement is used. For example, we would like to find the sum of 5 even numbers taken as input from the keyboard. The logic is –

- Read a number from the keyboard
- If that number is odd, without doing anything else, just move to next iteration forreading another number
- If the number is even, add it to *sum* and increment the accumulator variable.
- When accumulator crosses 5, stop the programThe program for the above

task can be written as –

```
sum=0 count=0 while True:
    x=int(input("Enter a number:"))
    if x%2 !=0:
        continue
    else:
        sum+=x
        count+=1

    if count==5:
        break

print("Sum= ", sum)
```

Sample Output:

```
Enter a number:13
Enter a number:12
Enter a number:4
Enter a number:5
Enter a number:-3
Enter a number:8
Enter a number:7
Enter a number:16
Enter a number:6
Sum= 46
```

2.2.4 Definite Loops using *for*

The *while* loop iterates till the condition is met and hence, the number of iterations are usually unknown prior to the loop. Hence, it is sometimes called as *indefinite loop*. When we know total number of times the set of statements to be executed,

```

for var in list/sequence:
    statement_1
    statement_2
    .....
    statement_n

statements_after_for

```

for loop will be used. This is called as a *definite loop*. The for-loop iterates over a set of numbers, a set of words, lines in a file etc. The syntax of for-loop would be –

Here, *for* and *in* are keywords
list/sequence is a set of elements on which the loop is iterated. That is, the loop will be executed till there is an element in *list/sequence*
statements constitutes body of the loop

Ex: In the below given example, a *list* `names` containing three strings has been created. Then the counter variable `x` in the *for*-loop iterates over this *list*. The variable `x` takes the elements in `names` one by one and the body of the loop is executed.

```

names=["Rama", "Shyama", "Bhama"]
for x in names:
    print(x)

```

The output would be –

```

Rama Shyama Bhama

```

NOTE: In Python, list is an important data type. It can take a sequence of elements of different types. It can take values as a comma separated sequence enclosed within square brackets. Elements in the list can be extracted using index (just similar to extracting array elements in C/C++ language). Various operations like indexing, slicing, merging, addition and deletion of elements etc. can be applied on lists. The details discussion on Lists will be done in Module 3.

The *for* loop can be used to print (or extract) all the characters in a string as shown below –

```

for i in "Hello": print(i, end='\t')

```

Output:

```

H   e   l   l   o

```

When we have a fixed set of numbers to iterate in a *for* loop, we can use a function ***range()***. The function *range()* takes the following format –

```
range(start, end, steps)
```

The `start` and `end` indicates starting and ending values in the sequence, where `end` is excluded in the sequence (That is, sequence is up to `end-1`). The default value of `start` is 0. The argument `steps` indicates the increment/decrement in the values of sequence with the default value as 1. Hence, the argument `steps` is optional. Let us consider few examples on usage of *range()* function.

Ex1. Printing the values from 0 to 4 –

```
for i in range(5): print(i, end= '\t')
```

Output:

```
0    1    2    3    4
```

Here, 0 is the default starting value. The statement `range(5)` is same as `range(0,5)` and `range(0,5,1)`.

Ex2. Printing the values from 5 to 1 –

```
for i in range(5,0,-1):print(i, end= '\t')
```

Output:

```
5    4    3    2    1
```

The function `range(5,0,-1)` indicates that the sequence of values are 5 to 0(excluded) insteps of -1 (downwards).

Ex3. Printing only even numbers less than 10 –

```
for i in range(0,10,2):print(i, end= '\t')
```

Output:

```
0    2    4    6    8
```

2.2.5 Loop Patterns

The *while*-loop and *for*-loop are usually used to go through a list of items or the contents of a file and to check maximum or minimum data value. These loops are generally constructed by the following procedure –

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

The construction of these loop patterns are demonstrated in the following examples.

Counting and Summing Loops: One can use the *for* loop for counting number of items in the list as shown –

```
count = 0
for i in [4, -2, 41, 34, 25]:
    count = count + 1
    print("Count:", count)
```

Here, the variable `count` is initialized before the loop. Though the counter variable `i` is not being used inside the body of the loop, it controls the number of iterations. The variable `count` is incremented in every iteration, and at the end of the loop the total number of elements in the list is stored in it.

One more loop similar to the above is finding the sum of elements in the list –

```
total = 0
for x in [4, -2, 41, 34, 25]:
    total = total + x
    print("Total:", total)
```

Here, the variable `total` is called as **accumulator** because in every iteration, it accumulates the sum of elements. In each iteration, this variable contains *running total of values so far*.

NOTE: In practice, both of the counting and summing loops are not necessary, because there are built-in functions `len()` and `sum()` for the same tasks respectively.

Maximum and Minimum Loops: To find maximum element in the list, the following code can be used –

```
big = None
print('Before Loop:', big)
for x in [12, 0, 21, -3]:
    if big is None or x > big :
        big = x
    print('Iteration Variable:', x, 'Big:', big)

print('Biggest:', big)
```

Output:

```
Before Loop: None
Iteration Variable: 12    Big: 12
Iteration Variable: 0    Big: 12
Iteration Variable: 21    Big: 21
```

```
Iteration Variable: -3   Big: 21
Biggest: 21
```

Here, we initialize the variable `big` to `None`. It is a special constant indicating empty. Hence, we cannot use relational operator `==` while comparing it with `big`. Instead, the `is` operator must be used. In every iteration, the counter variable `x` is compared with previous value of `big`. If `x > big`, then `x` is assigned to `big`.

Similarly, one can have a loop for finding smallest of elements in the list as given below –

```
small = None
print('Before Loop:', small)
for x in [12, 0, 21,-3]:
    if small is None or x < small :
        small = x
    print('Iteration Variable:', x, 'Small:', small)

print('Smallest:', small)
```

Output:

```
Before Loop: None
Iteration Variable: 12 Small: 12
Iteration Variable: 0 Small: 0
Iteration Variable: 21 Small: 0
Iteration Variable: -3 Small: -3
Smallest: -3
```

NOTE: In Python, there are built-in functions `max()` and `min()` to compute maximum and minimum values among. Hence, the above two loops need not be written by the programmer explicitly. The inbuilt function `min()` has the following code in Python –

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

2.3 STRINGS

A string is a sequence of characters, enclosed either within a pair of single quotes or double quotes. Each character of a string corresponds to an index number, starting with zero as shown below –

```
S= "Hello World"
```

character	H	e	l	l	o		w	o	r	l	d
-----------	---	---	---	---	---	--	---	---	---	---	---

index	0	1	2	3	4	5	6	7	8	9	10
-------	---	---	---	---	---	---	---	---	---	---	----

The characters of a string can be accessed using index enclosed within square brackets. For example,

```
>>> word1="Hello"
>>> word2='hi'
>>> x=word1[1]          #2nd character of word1 is extracted
>>> print(x) e
>>> y=word2[0]         #1st character of word1 is extracted
>>> print(y) h
```

Python supports negative indexing of string starting from the end of the string as shown below –

S= "Hello World"

character	H	e	l	l	o		w	o	r	l	D
Negative index	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

The characters can be extracted using negative index also. For example,

```
>>> var="Hello"
>>> print(var[-1]) o
>>> print(var[-4]) e
```

Whenever the string is too big to remember last positive index, one can use negative index to extract characters at the end of string.

2.3.2 Getting Length of a String using *len()*

The *len()* function can be used to get length of a string.

```
>>> var="Hello"
>>> ln=len(var)
>>> print(ln)
5
```

The index for string varies from 0 to length-1. Trying to use the index value beyond this range generates error.

```
>>> var="Hello"
>>> ln=len(var)
>>> ch=var[ln]
IndexError: string index out of range
```

2.3.3 Traversal through String with a Loop

Extracting every character of a string one at a time and then performing some action on that character is known as *traversal*. A string can be traversed either using *while* loop or using *for* loop in different ways. Few of such methods is shown here –

- **Using for loop:**

```
st="Hello"
for i in st:
    print(i, end='\t')
```

Output:

```
H   e   l   l   o
```

In the above example, the *for* loop is iterated from first to last character of the string *st*. That is, in every iteration, the counter variable *i* takes the values as H, e, l, l and o. The loop terminates when no character is left in *st*.

- **Using while loop:**

```
st="Hello"
i=0
while i<len(st):
    print(st[i], end='\t')
    i+=1
```

Output:

```
H   e   l   l   o
```

In this example, the variable *i* is initialized to 0 and it is iterated till the length of the string. In every iteration, the value of *i* is incremented by 1 and the character in a string is extracted using *i* as index.

2.3.4 String Slices

A segment or a portion of a string is called as *slice*. Only a required number of characters can be extracted from a string using colon (:) symbol. The basic syntax for slicing a string would be –

```
st[i:j:k]
```

This will extract character from *i*th character of *st* till (*j*-1)th character in steps of *k*. If first index *i* is not present, it means that slice should start from the beginning of the string. If thesecond index *j* is not mentioned, it indicates the slice should be till the end of the string. The third parameter *k*, also known as **stride**, is used to indicate number of steps to be incremented after extracting first character. The default value of stride is 1.

Consider following examples along with their outputs to understand string slicing.

```
st="Hello World"           #refer this string for all examples
```

```
➤ print("st[:] is", st[:])      #output
Hello World
```

As both index values are not given, it assumed to be a full string.

- ```
print("st[0:5] is ", st[0:5]) #output is
Hello
```

Starting from 0<sup>th</sup> index to 4<sup>th</sup> index (5 is exclusive), characters will be printed.
- ```
print("st[0:5:1] is", st[0:5:1]) #output is  
Hello
```

This code also prints characters from 0th to 4th index in the steps of 1. Comparing this example with previous example, we can make out that when the stride value is 1, it is optional to mention.
- ```
print("st[3:8] is ", st[3:8]) #output is
lo Wo
```

Starting from 3<sup>rd</sup> index to 7<sup>th</sup> index (8 is exclusive), characters will be printed.
- ```
print("st[7:] is ", st[7:]) #output is  
orld
```

Starting from 7th index to till the end of string, characters will be printed.
- ```
print(st[::2]) #outputs
HloWrD
```

This example uses stride value as 2. So, starting from first character, every alternative character (char+2) will be printed.
- ```
print("st[4:4] is ", st[4:4]) #gives empty string
```

Here, `st[4:4]` indicates, slicing should start from 4th character and end with (4- 1)=3rd character, which is not possible. Hence the output would be an empty string.
- ```
print(st[3:8:2]) #output is
l o
```

Starting from 3<sup>rd</sup> character, till 7<sup>th</sup> character, every alternative index is considered.
- ```
print(st[1:8:3]) #output is  
eoo
```

Starting from index 1, till 7th index, every 3rd character is extracted here.
- ```
print(st[-4:-1]) #output is
orl
```

Refer the diagram of negative indexing given earlier. Excluding the -1st character, all characters at the indices -4, -3 and -2 will be displayed. Observe the role of stride with default value 1 here. That is, it is computed as  $-4+1=-3$ ,  $-3+1=-2$  etc.

➤ `print(st[-1:])`                      **#output is**  
d

Here, starting index is -1, ending index is not mentioned (means, it takes the index 10) and the stride is default value 1. So, we are trying to print characters from -1 (which is the last character of negative indexing) till 10<sup>th</sup> character (which is also the last character in positive indexing) in incremental order of 1. Hence, we will get only last character as output.

➤ `print(st[:-1])`                      **#output is**  
Hello Worl

Here, starting index is default value 0 and ending is -1 (corresponds to last character in negative indexing). But, in slicing, as last index is excluded always, -1<sup>st</sup> character is omitted and considered only up to -2<sup>nd</sup> character.

➤ `print(st[::])`                      **#outputs**

Hello World

Here, two colons have used as if stride will be present. But, as we haven't mentioned stride its default value 1 is assumed. Hence this will be a full string.

➤ `print(st[::-1])`                      **#outputs**

dlroW olleH

This example shows the power of slicing in Python. Just with proper slicing, we could be able to **reverse the string**. Here, the meaning is *a full string to be extracted in the order of -1*. Hence, the string is printed in the reverse order.

➤ `print(st[::-2])`                      **#output is**  
drWolH

Here, the string is printed in the reverse order in steps of -2. That is, every alternative character in the reverse order is printed. Compare this with example (6) given above.

By the above set of examples, one can understand the power of string slicing and of Python script. The slicing is a powerful tool of Python which makes many tasks simple pertaining to data types like strings, Lists, Tuple, Dictionary etc. (Other types will be discussed in later Modules)

### 2.3.5 Strings are Immutable

The objects of string class are immutable. That is, once the strings are created (or initialized), they cannot be modified. No character in the string can be edited/deleted/added. Instead, one can create a new string using an existing string by imposing any modification required.

Try to attempt following assignment –

```
>>> st= "Hello World"
>>> st[3]='t'
TypeError: 'str' object does not support item assignment
```

Here, we are trying to change the 4<sup>th</sup> character (index 3 means, 4<sup>th</sup> character as the first index is 0) to *t*. The error message clearly states that an assignment of new *item* (a string) is not possible on string object. So, to achieve our requirement, we can create a new string using slices of existing string as below –

```
>>> st= "Hello World"
>>> st1= st[:3]+ 't' + st[4:]
>>> print(st1)
Helto World #l is replaced by t in new
string st1
```

### 2.3.6 Looping and Counting

Using loops on strings, we can count the frequency of occurrence of a character within another string. The following program demonstrates such a pattern on computation called as a **counter**. Initially, we accept one string and one character (single letter). Our aim to find the total number of times the character has appeared in string. A variable *count* is initialized to zero, and incremented each time a character is found. The program is given below –

```
def countChar(st, ch):

 count=0
 for i in st:
 if i==ch:
 count+=1
 return count

st=input("Enter a string:")
ch=input("Enter a character to be counted:")
c=countChar(st, ch)
print("{0} appeared {1} times in {2}".format(ch, c, st))
```

#### Sample Output:

```
Enter a string: hello how are you?
Enter a character to be counted: h
h appeared 2 times in hello how are you?
```

### 2.3.7 The *in* Operator

The *in* operator of Python is a Boolean operator which takes two string operands. It returns True, if the first operand appears in second operand, otherwise returns False. For example,

```
>>> 'el' in 'hello' #el is found in helloTrue
>>> 'x' in 'hello' #x is not found in helloFalse
```

### 2.3.8 String Comparison

Basic comparison operators like < (less than), > (greater than), == (equals) etc. can be applied on string objects. Such comparison results in a Boolean value True or False. Internally, such comparison happens using ASCII codes of respective characters. Consider following examples –

```
Ex1. st= "hello"
 if st== 'hello':
 print('same')
```

Output is same. As the value contained in `st` and `hello` both are same, the equality results in True.

```
Ex2. st= "hello"
 if st<= 'Hello':
 print('lesser')
 else:
 print('greater')
```

Output is greater. The ASCII value of h is greater than ASCII value of H. Hence, `hello` is greater than `Hello`.

**NOTE:** A programmer must know ASCII values of some of the basic characters. Here are few –

|           |            |
|-----------|------------|
| A – Z     | : 65 – 90  |
| a – z     | : 97 – 122 |
| 0 – 9     | : 48 – 57  |
| Space     | 32         |
| Enter Key | 13         |

### 2.3.9 String Methods

String is basically a *class* in Python. When we create a string in our program, an *object* of that class will be created. A class is a collection of member variables and member methods(or functions). When we create an object of a particular class, the

object can use all the members (both variables and methods) of that class. Python provides a rich set of built-in classes for various purposes. Each class is enriched with a useful set of utility functions and variables that can be used by a Programmer. A programmer can create a class based on his/her requirement, which are known as user-defined classes.

The built-in set of members of any class can be accessed using the dot operator as shown–

```
objName.memberMethod(arguments)
```

The dot operator always binds the member name with the respective object name. This is very essential because, there is a chance that more than one class has members with same name. To avoid that conflict, almost all Object oriented languages have been designed with this common syntax of using dot operator. (Detailed discussion on classes and objects will be done in later Modules.)

Python provides a function (or method) **dir** to list all the variables and methods of a particular class object. Observe the following statements –

```
>>> s="hello" #string object is created with the name s
>>> type(s) #checking type of s
<class 'str'> #s is object of type class str
>>> dir(s) #display all methods and variables of
object s

['__add__', '__class__', '__contains__', '__delattr__', '
dir__', '__doc__', '__eq__', '__format__', '__ge__', '
getattribute__', '__getitem__', '__getnewargs__', '__gt__', '
hash__', '__init__', '__init_subclass__', '__iter__', '__le
', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod
', '__rmul__', '__setattr__', '__sizeof__', '__str__', '
subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

**Students need not remember the above list !!**

Note that, the above set of variables and methods are common for any object

of string class that we create. Each built-in method has a predefined set of arguments and return type. To know the usage, working and behavior of any built-in method, one can use the command **help**. For example, if we would like to know what is the purpose of `islower()` function (refer above list to check its existence!!), how it behaves etc, we can use the statement –

```
>>> help(str.islower)
Help on method_descriptor:

islower(...)
 S.islower() -> bool

 Return True if all cased characters in S are
 lowercase and there is at least one cased character in S,
 False otherwise.
```

This is built-in help-service provided by Python. Observe the `className.memberName` format while using **help**.

The methods are usually called using the object name. This is known as **method invocation**. We say that a method is invoked using an object.

Now, we will discuss some of the important methods of string class.

- **capitalize(s)** : This function takes one string argument `s` and returns a capitalized version of that string. That is, the first character of `s` is converted to upper case, and all other characters to lowercase. Observe the examples given below –

```
Ex1. >>> s="hello"
 >>> s1=str.capitalize(s)
 >>> print(s1)
 Hello #1st character is changed to
 uppercasse
```

```
Ex2. >>> s="hello World"
 >>> s1=str.capitalize(s)
 >>> print(s1)
 Hello world
```

Observe in Ex2 that the first character is converted to uppercase, and an in-between uppercase letter `W` of the original string is converted to lowercase.



- **s.upper():** This function returns a copy of a string *s* to uppercase. As strings are immutable, the original string *s* will remain same.

```
>>> st= "hello"
>>> st1=st.upper()
>>> print(st1)
 'HELLO'
>>> print(st) #no change in original string'hello'
```

- **s.lower():** This method is used to convert a string *s* to lowercase. It returns a copy of original string after conversion, and original string is intact.

```
>>> st='HELLO'
>>> st1=st.lower()
>>> print(st1)hello
>>> print(st) #no change in original string
 HELLO
```

- **s.find(s1) :** The `find()` function is used to search for a substring *s1* in the string *s*. If found, the index position of first occurrence of *s1* in *s*, is returned. If *s1* is not found in *s*, then -1 is returned.

```
>>> st='hello'
>>> i=st.find('l')
>>> print(i) #output is 2
>>> i=st.find('lo')
>>> print(i) #output is 3
>>> print(st.find('x')) #output is -1
```

The `find()` function can take one more form with two additional arguments viz. start and end positions for search.

```
>>> st="calender of Feb. cal of march"
>>> i= st.find('cal')
>>> print(i) #output is 0
```

Here, the substring `'cal'` is found in the very first position of *st*, hence the result is 0.

```
>>> i=st.find('cal',10,20)
>>> print(i) #output is 17
```

Here, the substring `cal` is searched in the string *st* between 10<sup>th</sup> and 20<sup>th</sup> position and hence the result is 17.

```
>>> i=st.find('cal',10,15)
>>> print(i) #ouput is -1
```

In this example, the substring 'cal' has not appeared between 10<sup>th</sup> and 15<sup>th</sup> character of st. Hence, the result is -1.

- **s.strip():** Returns a copy of string s by removing leading and trailing white spaces.

```
>>> st=" hello world "
>>> st1 = st.strip()
>>> print(st1)
 hello world
```

The *strip()* function can be used with an argument *chars*, so that specified *chars* are removed from beginning or ending of s as shown below –

```
>>> st="###Hello###"
>>> st1=st.strip('#')
>>> print(st1) #all hash symbols are removed
 Hello
```

We can give more than one character for removal as shown below –

```
>>> st="Hello world"
>>> st.strip("Hld")ello wor
```

- **S.startswith(prefix, start, end):** This function has 3 arguments of which *start* and *end* are option. This function returns True if S starts with the specified *prefix*, False otherwise.

```
>>> st="hello world"
>>> st.startswith("he") #returns True
```

When *start* argument is provided, the search begins from that position and returns True or False based on search result.

```
>>> st="hello world"
>>> st.startswith("w",6) #True because w is at 6th
position
```

When both *start* and *end* arguments are given, search begins at *start* and ends at *end*.

```
>>> st="xyz abc pqr ab mn gh"
>>> st.startswith("pqr ab mn",8,12) #returns False
>>> st.startswith("pqr ab mn",8,18) #returns True
```

The *startswith()* function requires case of the alphabet to match. So, when we are not sure about the case of the argument, we can convert it to either upper case or lowercase and then use *startswith()* function as below –

```
>>> st="Hello"
>>> st.startswith("he") #returns False
>>> st.lower().startswith("he") #returns True
```

- **S.count(s1, start, end):** The `count()` function takes three arguments – *string*, *starting position* and *ending position*. This function returns the number of non-overlapping occurrences of substring s1 in string S in the range of *start* and *end*.

```
>>> st="hello how are you? how about you?"
>>> st.count('h') #output is 3
>>> st.count('how') #output is 2
>>> st.count('how', 3, 10) #output is 1 because of range
given
```

**There are many more built-in methods for string class. Students are advised to explore more for further study.**

### 2.3.10 Parsing Strings

Sometimes, we may want to search for a substring matching certain criteria. For example, finding domain names from email-IDs in the list of messages is a useful task in some projects. Consider a string below and we are interested in extracting only the domain name.

```
"From chetanahegde@ieee.org Wed Feb 21 09:14:16 2018"
```

Now, our aim is to extract only *ieee.org*, which is the domain name. We can think of logic as–

- Identify the position of @, because all domain names in email IDs will be after the symbol @
- Identify a white space which appears after @ symbol, because that will be the end of domain name.
- Extract the substring between @ and white-space.

The concept of string slicing and *find()* function will be useful here. Consider the code given below –

```
st="From chetanahegde@ieee.org Wed Feb 21 09:14:16 2018"
atpos=st.find('@') #finds the position of @

print('Position of @ is', atpos)

spacePos=st.find(' ', atpos) #position of white-space
after @print('Position of space after @ is', spacePos)
```

```
host=st[atpos+1:spacePos] #slicing from @ till white-
spaceprint(host)
```

Execute above program to get the output as *ieee.org*. One can apply this logic in a loop, when our string contains series of email IDs, and we may want to extract all those mail IDs.

### 2.3.11 Format Operator

The format operator, % allows us to construct strings, replacing parts of the strings with the data stored in variables. The first operand is the format string, which contains one or more *format sequences* that specify how the second operand is formatted. The result is a string.

```
>>> sum=20
>>> '%d' %sum
 '20' #string '20', but not integer 20
```

Note that, when applied on both integer operands, the % symbol acts as a modulus operator. When the first operand is a string, then it is a format operator. Consider few examples illustrating usage of format operator.

**Ex1.** >>> "The sum value %d is originally integer"%sum'The sum  
value 20 is originally integer`

**Ex2.** >>> '%d %f %s'%(3,0.5, 'hello')  
 '3 0.500000 hello`

**Ex3.** >>> '%d %g %s'%(3,0.5, 'hello')  
 '3 0.5 hello`

**Ex4.** >>> '%d'% 'hello'  
TypeError: %d format: a number is required, not str

**Ex5.** >>> '%d %d %d'%(2,5)  
TypeError: not enough arguments for format string

## MODULE – 3

# LISTS AND DICTIONARIES

### 3.1 LISTS

A list is an ordered sequence of values. It is a data structure in Python. The values inside the lists can be of any type (like integer, float, strings, lists, tuples, dictionaries etc) and are called as *elements* or *items*. The elements of lists are enclosed within square brackets. For example,

```
ls1=[10,-4, 25, 13]
ls2=["Tiger", "Lion", "Cheetah"]
```

Here, ls1 is a list containing four integers, and ls2 is a list containing three strings. A list need not contain data of same type. We can have mixed type of elements in list. For example,

```
ls3=[3.5, 'Tiger', 10, [3,4]]
```

Here, ls3 contains a float, a string, an integer and a list. This illustrates that a list can be nested as well.

An empty list can be created any of the following ways –

```
>>> ls =[]
>>> type(ls)
<class 'list'>
or
>>> ls =list()
>>> type(ls)
<class 'list'>
```

In fact, list() is the name of a method (special type of method called as constructor – which will be discussed in Module 4) of the class *list*. Hence, a new list can be created using this function by passing arguments to it as shown below –

```
>>> ls2=list([3,4,1])
>>> print(ls2)
[3, 4, 1]
```

### 3.1.1 List Operations

Python allows to use operators + and \* on lists. The operator + uses two list objects and returns concatenation of those two lists. Whereas \* operator take one list object and one integer value, say n, and returns a list by repeating itself for n times.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6,7]
>>> print(ls1+ls2) #concatenation using +
[1, 2, 3, 5, 6, 7]

>>> ls1=[1,2,3]
>>> print(ls1*3) #repetition using *
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> [0]*4 #repetition using *
[0, 0, 0, 0]
```

### 3.1.2 Traversing a List

A list can be traversed using *for* loop. If we need to use each element in the list, we can use the *for* loop and *in* operator as below –

```
>>> ls=[34, 'hi', [2,3],-5]
>>> for item in ls:
 print(item)
```

```
34
hi
Hello
-5
```

List elements can be accessed with the combination of *range()* and *len()* functions as well –

```
ls=[1,2,3,4]
for i in range(len(ls)):
 ls[i]=ls[i]**2

print(ls) #output is [1, 4, 9, 16]
```

Here, we wanted to do modification in the elements of list. Hence, referring indices is suitable than referring elements directly. The *len()* returns total number of elements in the list (here it is 4). Then *range()* function makes the loop to range from 0 to 3 (i.e. 4-1). Then, for every index, we are updating the list elements (replacing original value by its square).

### 3.1.3 List Slices

Similar to strings, the slicing can be applied on lists as well. Consider a list `t` given below, and a series of examples following based on this object.

```
t=['a','b','c','d','e']
```

- Extracting full list without using any index, but only a slicing operator

```
>>> print(t[:])
['a', 'b', 'c', 'd', 'e']
```

- Extracting elements from 2<sup>nd</sup> position –

```
>>> print(t[1:])
['b', 'c', 'd', 'e']
```

- Extracting first three elements –

```
>>> print(t[:3])
['a', 'b', 'c']
```

- Selecting some middle elements –

```
>>> print(t[2:4])
['c', 'd']
```

- Using negative indexing –

```
>>> print(t[:-2])
['a', 'b', 'c']
```

- **Reversing a list** using negative value for stride –

```
>>> print(t[::-1])
['e', 'd', 'c', 'b', 'a']
```

- Modifying (reassignment) only required set of values –

```
>>> t[1:3]=['p','q']
>>> print(t)
['a', 'p', 'q', 'd', 'e']
```

Thus, slicing can make many tasks simple.

### 3.1.4 List Methods

There are several built-in methods in *list* class for various purposes. Here, we will discuss some of them.

- **append():** This method is used to add a new element at the end of a list.

```
>>> ls=[1,2,3]
>>> ls.append('hi')
```

```
>>> ls.append(10)
>>> print(ls)
[1, 2, 3, 'hi', 10]
```

- **extend():** This method takes a list as an argument and all the elements in this list are added at the end of invoking list.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.extend(ls1)
>>> print(ls2)
[5, 6, 1, 2, 3]
```

Now, in the above example, the list ls1 is unaltered.

- **sort():** This method is used to sort the contents of the list. By default, the function will sort the items in ascending order.

```
>>> ls=[3,10,5, 16,-2]
>>> ls.sort()
>>> print(ls)
[-2, 3, 5, 10, 16]
```

When we want a list to be sorted in descending order, we need to set the argument as shown –

```
>>> ls.sort(reverse=True)
>>> print(ls)
[16, 10, 5, 3, -2]
```

- **reverse():** This method can be used to reverse the given list.

```
>>> ls=[4,3,1,6]
>>> ls.reverse()
>>> print(ls)
[6, 1, 3, 4]
```

- **count():** This method is used to count number of occurrences of a particular value within list.

```
>>> ls=[1,2,5,2,1,3,2,10]
>>> ls.count(2)
3 #the item 2 has appeared 3 times
in ls
```

- **clear():** This method removes all the elements in the list and makes the list empty.

```
>>> ls=[1,2,3]
>>> ls.clear()
>>> print(ls)
[]
```

- **insert():** Used to insert a value before a specified index of the list.



```
>>> ls=[3,5,10]
>>> ls.insert(1,"hi")
>>> print(ls)
 [3, 'hi', 5, 10]
```

- **index():** This method is used to get the index position of a particular value in the list.

```
>>> ls=[4, 2, 10, 5, 3, 2, 6]
>>> ls.index(2)
 1
```

Here, the number 2 is found at the index position 1. Note that, this function will give index of only the first occurrence of a specified value. The same function can be used with two more arguments *start* and *end* to specify a range within which the search should take place.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6]
>>> ls.index(2)
 2
>>> ls.index(2,3,7)
 6
```

If the value is not present in the list, it throws ValueError.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6]
>>> ls.index(53)
 ValueError: 53 is not in list
```

#### ***Few important points about List Methods:***

1. There is a difference between *append()* and *extend()* methods. The former adds the argument as it is, whereas the latter enhances the existing list. To understand this, observe the following example –

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.append(ls1)
>>> print(ls2)
 [5, 6, [1, 2, 3]]
```

Here, the argument *ls1* for the *append()* function is treated as one item, and made as an inner list to *ls2*. On the other hand, if we replace *append()* by *extend()* then the result would be –

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.extend(ls1)
>>> print(ls2)
 [5, 6, 1, 2, 3]
```

2. The **sort()** function can be applied only when the list contains elements of compatible types. But, if a list is a mix non-compatible types like integers and string, the comparison cannot be done. Hence, Python will throw `TypeError`. For example,

```
>>> ls=[34, 'hi', -5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

Similarly, when a list contains integers and sub-list, it will be an error.

```
>>> ls=[34,[2,3],5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'list' and 'int'
```

Integers and floats are compatible and relational operations can be performed on them. Hence, we can sort a list containing such items.

```
>>> ls=[3, 4.5, 2]
>>> ls.sort()
>>> print(ls)
[2, 3, 4.5]
```

3. The **sort()** function uses one important argument **keys**. When a list is containing tuples, it will be useful. We will discuss tuples later in this Module.
4. Most of the list methods like *append()*, *extend()*, *sort()*, *reverse()* etc. modify the list object internally and return `None`.

```
>>> ls=[2,3]
>>> ls1=ls.append(5)
>>> print(ls)
[2,3,5]
>>>
print (ls1)
None
```

### 3.1.5 Deleting Elements

Elements can be deleted from a list in different ways. Python provides few built-in methods for removing elements as given below –

- **pop():** This method deletes the last element in the list, by default.

```
>>> ls=[3,6,-2,8,10]
>>> x=ls.pop() #10 is removed from list and stored in x
>>> print(ls)
[3, 6, -2, 8]
```

```
>>>print(x)
10
```

When an element at a particular index position has to be deleted, then we can give that position as argument to *pop()* function.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1) #item at index 1 is popped
>>> print(t)
 ['a', 'c']
>>> print(x)
 b
```

- **remove():** When we don't know the index, but know the value to be removed, then this function can be used.

```
>>> ls=[5,8, -12,34,2]
>>> ls.remove(34)
>>> print(ls)
 [5, 8, -12, 2]
```

Note that, this function will remove only the first occurrence of the specified value, but not all occurrences.

```
>>> ls=[5,8, -12, 34, 2, 6, 34]
>>> ls.remove(34)
>>> print(ls)
 [5, 8, -12, 2, 6, 34]
```

Unlike *pop()* function, the *remove()* function will not return the value that has been deleted.

- **del:** This is an operator to be used when more than one item to be deleted at a time. Here also, we will not get the items deleted.

```
>>> ls=[3,6,-2,8,1]
>>> del ls[2] #item at index 2 is deleted
>>> print(ls)
 [3, 6, 8, 1]

>>> ls=[3,6,-2,8,1]
>>> del ls[1:4] #deleting all elements from index 1 to 3
>>> print(ls)
 [3, 1]
```

#### Deleting all odd indexed elements of a list -

```
>>> t=['a', 'b', 'c', 'd', 'e']
>>> del t[1::2]
>>> print(t)
 ['a', 'c', 'e']
```

### 3.1.6 Lists are Mutable

The elements in the list can be accessed using a numeric index within square-brackets. It is similar to extracting characters in a string.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[1])
hi
>>> print(ls[2])
[2, 3]
```

Observe here that, the inner list is treated as a single element by outer list. If we would like to access the elements within inner list, we need to use double-indexing as shown below –

```
>>> print(ls[2][0])
2
>>> print(ls[2][1])
3
```

Note that, the indexing for inner-list again starts from 0. Thus, when we are using double-indexing, the first index indicates position of inner list inside outer list, and the second index means the position particular value within inner list.

Unlike strings, lists are mutable. That is, using indexing, we can modify any value within list. In the following example, the 3<sup>rd</sup> element (i.e. index is 2) is being modified –

```
>>> ls=[34, 'hi', [2,3],-5]
>>> ls[2]='Hello'
>>> print(ls)
[34, 'hi', 'Hello', -5]
```

The list can be thought of as a relationship between indices and elements. This relationship is called as a **mapping**. That is, each index maps to one of the elements in a list.

The index for extracting list elements has following properties –

- Any integer expression can be an index.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[2*1])
'Hello'
```
- Attempt to access a non-existing index will throw an IndexError.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[4])
IndexError: list index out of range
```
- A negative indexing counts from backwards.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[-1])
-5
>>> print(ls[-3])
hi
```

The **in** operator applied on lists will results in a Boolean value.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> 34 in ls
True
>>> -2 in ls
False
```

### 3.1.7 Lists and Functions

The utility functions like **max()**, **min()**, **sum()**, **len()** etc. can be used on lists. Hence most of the operations will be easy without the usage of loops.

```
>>> ls=[3,12,5,26, 32,1,4]
>>> max(ls) # prints 32
>>> min(ls) # prints 1
>>> sum(ls) # prints 83
>>> len(ls) # prints 7

>>> avg=sum(ls)/len(ls)
>>> print(avg)
11.857142857142858
```

When we need to read the data from the user and to compute sum and average of those numbers, we can write the code as below –

```
ls= list()
while (True):
 x= input('Enter a number: ')
 if x== 'done':
 break

 x= float(x)
 ls.append(x)

average = sum(ls) / len(ls)
print('Average:', average)
```

In the above program, we initially create an empty list. Then, we are taking an infinite *while*-loop. As every input from the keyboard will be in the form of a string, we need to convert x into float type and then append it to a list. When the keyboard input is a string 'done', then the loop is going to get terminated. After the loop, we will find the average of those numbers with the help of built-in functions *sum()* and *len()*.

### 3.1.8 Lists and Strings

Though both lists and strings are sequences, they are not same. In fact, a list of characters is not same as string. To convert a string into a list, we use a method **list()** as below –

```
>>> s="hello"
>>> ls=list(s)
>>> print(ls)
['h', 'e', 'l', 'l', 'o']
```

The method **list()** breaks a string into individual letters and constructs a list. If we want a list of words from a sentence, we can use the following code –

```
>>> s="Hello how are you?"
>>> ls=s.split()
>>> print(ls)
['Hello', 'how', 'are', 'you?']
```

Note that, when no argument is provided, the **split()** function takes the delimiter as white space. If we need a specific delimiter for splitting the lines, we can use as shown in following example –

```
>>> dt="20/03/2018"
>>> ls=dt.split('/')
>>> print(ls)
['20', '03', '2018']
```

There is a method **join()** which behaves opposite to **split()** function. It takes a list of strings as argument, and joins all the strings into a single string based on the delimiter provided. For example –

```
>>> ls=["Hello", "how", "are", "you"]
>>> d=' '
>>>
>>> d.join(ls)
'Hello how
are you'
```

Here, we have taken delimiter d as white space. Apart from space, anything can be taken as delimiter. When we don't need any delimiter, use empty string as delimiter.

### 3.1.9 Parsing Lines

In many situations, we would like to read a file and extract only the lines containing required pattern. This is known as **parsing**. As an illustration, let us assume that there is a log file containing details of email

communication between employees of an organization. For all received mails, the file contains lines as –

From [ManojkumarSB@bgsit.ac.in](mailto:ManojkumarSB@bgsit.ac.in) Sat Jul 03 09:14:16 2021

From [sbmanojkumar@bgsit.ac.in](mailto:sbmanojkumar@bgsit.ac.in) Sun Jul 4 06:12:51 2021

.....

Apart from such lines, the log file also contains mail-contents, to-whom the mail has been sent etc. Now, if we are interested in extracting only the days of incoming mails, then we can go for parsing. That is, we are interested in knowing on which of the days, the mails have been received. The code would be –

```
fhand = open('logFile.txt')
for line in fhand:
 line = line.rstrip()
 if not line.startswith('From '):
 continue
 words =
 line.split()
 print(words[2])
```

Obviously, all received mails starts from the word From. Hence, we search for only such lines and then split them into words. Observe that, the first word in the line would be From, second word would be email-ID and the 3<sup>rd</sup> word would be day of a week. Hence, we will extract words[2] which is 3<sup>rd</sup> word.

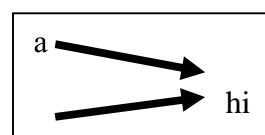
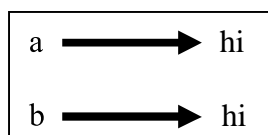
### 3.1.10 Objects and Values

Whenever we assign two variables with same value, the question arises – whether both the variables are referring to same object, or to different objects. This is important aspect to know, because in Python everything is a class object. There is nothing like elementary datatype.

Consider a situation –

```
a= "hi"
b= "hi"
```

Now, the question is whether both a and b refer to the **same string**. There are two possible states –



In the first situation, a and b are two different objects, but containing same value. The modification in one object is nothing to do with the other. Whereas, in the second case, both a and b are referring to the same object. That is, a is an **alias name** for b and vice-versa. In other

words, these two are referring to same memory location.

To check whether two variables are referring to same object or not, we can use **is** operator.

```
>>> a= "hi"
>>> b= "hi"
>>> a is b #result is True
>>> a==b #result is True
```

When two variables are referring to same object, they are called as **identical objects**. When two variables are referring to different objects, but contain a same value, they are known as **equivalent objects**. For example,

```
>>> s1=input("Enter a string:")#assume you entered hello
>>> s2= input("Enter a string:")#assume you entered hello

>>> s1 is s2 #check s1 and s2 are identicalFalse
>>> s1 == s2 #check s1 and s2 are equivalentTrue
```

Here **s1** and **s2** are equivalent, but not identical.

If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

String literals are **interned** by default. That is, when two string literals are created in the program with a same value, they are going to refer same object. But, string variables read from the key-board will not have this behavior, because their values are depending on the user's choice.

Lists are not interned. Hence, we can see following result –

```
>>> ls1=[1,2,3]
>>> ls2=[1,2,3]
>>> ls1 is ls2 #output is False
>>> ls1 == ls2 #output is True
```

### 3.1.11 Aliasing

When an object is assigned to other using assignment operator, both of them will refer to same object in the memory. The association of a variable with an object is called as **reference**.

```
>>> ls1=[1,2,3]
>>> ls2= ls1
```

### 3.1.12 List Arguments

When a list is passed to a function as an argument, then function receives reference to this list. Hence, if the list is modified within a function, the caller will get the modified version. Consider an example –



```
def del_front(t):
 del t[0]

ls = ['a', 'b', 'c']
del_front(ls)
print(ls) # output is ['b', 'c']
```

Now, ls2 is said to be **reference** of ls1. In other words, there are two references to the same object in the memory.

An object with more than one reference has more than one name, hence we say that object is **aliased**. If the aliased object is mutable, changes made in one alias will reflect the other.

```
>>> ls2[1]= 34
>>> print(ls1) #output is [1, 34, 3]
```

Strings are safe in this regards, as they are immutable.

Here, the argument ls and the parameter t both are aliases to same object.

One should understand the operations that will modify the list and the operations that create a new list. For example, the **append()** function modifies the list, whereas the + operator creates a new list.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1) #output is [1 2 3]
>>> print(t2) #prints None

>>> t3 = t1 + [5]
>>> print(t3) #output is [1 2 3 5]
>>> t2 is t3 #output is False
```

Here, after applying *append()* on t1 object, the t1 itself has been modified and t2 is not going to get anything. But, when + operator is applied, t1 remains same but t3 will get the updated result.

The programmer should understand such differences when he/she creates a function intending to modify a list. For example, the following function has no effect on the original list –

```
def test(t):
 t=t[1:]

ls=[1,2,3]
test(ls)
print(ls) #prints [1, 2, 3]
```

One can write a return statement after slicing as below –

```
def test(t):
 return t[1:]

ls=[1,2,3]
ls1=test(ls)
print(ls1) #prints [2, 3]
print(ls) #prints [1, 2, 3]
```

In the above example also, the original list is not modified, because a return statement always creates a new object and is assigned to LHS variable at the position of function call.

## 3.2 DICTIONARIES

A dictionary is a collection of unordered set of **key:value** pairs, with the requirement that keys are unique in one dictionary. Unlike lists and strings where elements are accessed using index values (which are integers), the values in dictionary are accessed using keys. A key in dictionary can be any immutable type like strings, numbers and tuples. (The tuple can be made as a key for dictionary, only if that tuple consist of string/number/ sub-tuples). As lists are mutable – that is, can be modified using index assignments, slicing, or using methods like *append()*, *extend()* etc, they cannot be a key for dictionary.

One can think of a dictionary as a mapping between set of indices (which are actually keys) and a set of values. Each key maps to a value.

An empty dictionary can be created using two ways –

```
d= {}
```

OR

```
d=dict()
```

To add items to dictionary, we can use square brackets as –

```
>>> d={}
>>> d["Mango"]="Fruit"
>>> d["Banana"]="Fruit"
>>> d["Cucumber"]="Veg"
>>> print(d)
{'Mango': 'Fruit', 'Banana': 'Fruit', 'Cucumber':
'Veg'}
```

To initialize a dictionary at the time of creation itself, one can use the code like –

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135}
>>> print(tel_dir)
 {'Tom': 3491, 'Jerry': 8135}

>>> tel_dir['Donald']=4793
>>> print(tel_dir)
 {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793}
```

**NOTE** that the order of elements in dictionary is unpredictable. That is, in the above example, don't assume that 'Tom': 3491 is first item, 'Jerry': 8135 is second item etc. As dictionary members are not indexed over integers, the order of elements inside it may vary. However, using a *key*, we can extract its associated value as shown below –

```
>>> print(tel_dir['Jerry'])
 8135
```

Here, the key 'Jerry' maps with the value 8135, hence it doesn't matter where exactly it is inside the dictionary.

If a particular key is not there in the dictionary and if we try to access such key, then the *KeyError* is generated.

```
>>> print(tel_dir['Mickey'])
 KeyError: 'Mickey'
```

The **len()** function on dictionary object gives the number of key-value pairs in that object.

```
>>> print(tel_dir)
 {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793}
>>> len(tel_dir)
 3
```

The **in** operator can be used to check whether any **key** (not value) appears in the dictionary object.

```
>>> 'Mickey' in tel_dir #output is False
>>> 'Jerry' in tel_dir #output is True
>>> 3491 in tel_dir #output is False
```

We observe from above example that the value 3491 is associated with the key 'Tom' in tel\_dir. But, the **in** operator returns False.

The dictionary object has a method **values()** which will **return a list** of all the values associated with keys within a dictionary. If we would like to check whether a particular value exist in a dictionary, we can make use of it as shown below –

```
>>> 3491 in tel_dir.values() #output is True
```

The **in** operator behaves differently in case of lists and dictionaries as explained hereunder–

- When **in** operator is used to search a value in a list, then *linear search* algorithm is used internally. That is, each element in the list is checked one by one sequentially. This is considered to be expensive in the view of total time taken to process. Because, if there are 1000 items in the list, and if the element in the list which we are search for is in the last position (or if it does not exists), then before yielding result of search (True or False), we would have done 1000 comparisons. In other words, linear search requires  $n$  number of comparisons for the input size of  $n$  elements. Time complexity of the linear search algorithm is  $O(n)$ .
- The keys in dictionaries of Python are basically **hashable** elements. The concept of **hashing** is applied to store (or maintain) the keys of dictionaries. Normally hashing techniques have the time complexity as  $O(\log n)$  for basic operations like insertion, deletion and searching. Hence, the **in** operator applied on keys of dictionaries works better compared to that on lists. (Hashing technique is explained at the end of this Section, for curious readers)

### 3.2.1 Dictionary as a Collection of Counters

Assume that we need to count the frequency of alphabets in a given string. There are different methods to do it –

- Create 26 variables to represent each alphabet. Traverse the given string and increment the corresponding counter when an alphabet is found.
- Create a list with 26 elements (all are zero in the beginning) representing alphabets. Traverse the given string and increment corresponding indexed position in the list when an alphabet is found.
- Create a dictionary with characters as keys and counters as values. When we find a character for the first time, we add the item to dictionary. Next time onwards, we increment the value of existing item.

Each of the above methods will perform same task, but the logic of implementation will be different. Here, we will see the implementation

using dictionary.

```
s=input("Enter a string:") #read a string
d=dict() #create empty dictionary

for ch in s: #traverse through stringif
 ch not in d: #if new character found
 d[ch]=1 #initialize counter to 1
 else: #otherwise, increment counter
 d[ch]+=1

print(d) #display the dictionary
```

The sample output would be –

```
Enter a string: Hello World
{'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'W': 1, 'r': 1, 'd': 1}
```

It can be observed from the output that, a dictionary is created here with characters as keys and frequencies as values. **Note** that, here we have computed **histogram** of counters.

Dictionary in Python has a method called as **get()**, which takes key and a default value as two arguments. If key is found in the dictionary, then the **get()** function returns corresponding value, otherwise it returns default value. For example,

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
>>> print(tel_dir.get('Jerry',0))
 8135
>>> print(tel_dir.get('Donald',0))
 0
```

In the above example, when the **get()** function is taking 'Jerry' as argument, it returned corresponding value, as 'Jerry' is found in tel\_dir. Whereas, when **get()** is used with 'Donald' as key, the default value 0 (which is provided by us) is returned.

The function **get()** can be used effectively for calculating frequency of alphabets in a string. Here is the modified version of the program –

```
s=input("Enter a
string:") d=dict()

for ch in s:
 d[ch]=d.get(ch,0)+1
```

```
print(d)
```

In the above program, for every character *ch* in a given string, we will try to retrieve a value. When the *ch* is found in *d*, its value is retrieved, 1 is added to it, and restored. If *ch* is not found, 0 is taken as default and then 1 is added to it.

### 3.2.2 Looping and Dictionaries

When a *for*-loop is applied on dictionaries, it will iterate over the keys of dictionary. If we want to print key and values separately, we need to use the statements as shown –

```
tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
for k in tel_dir:
 print(k, tel_dir[k])
```

Output would be –

```
Tom 3491
Jerry 8135
Mickey 1253
```

Note that, while accessing items from dictionary, the keys may not be in order. If we want to print the keys in alphabetical order, then we need to make a list of the keys, and then sort that list. We can do so using **keys()** method of dictionary and **sort()** method of lists. Consider the following code –

```
tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
ls=list(tel_dir.keys())
print("The list of keys:",ls)
ls.sort()
print("Dictionary elements in alphabetical order:")
for k in ls:
 print(k, tel_dir[k])
```

The output would be –

```
The list of keys: ['Tom', 'Jerry', 'Mickey']
Dictionary elements in alphabetical order:
Jerry 8135
Mickey 1253
Tom 3491
```

**Note:** The key-value pair from dictionary can be together accessed with the help of a method **items()** as shown –

```
>>> d={'Tom':3412, 'Jerry':6781, 'Mickey':1294}
>>> for k,v in d.items():
 print(k,v)
```

Output:

```
Tom 3412
Jerry 6781
Mickey 1294
```

The usage of comma-separated list `k,v` here is internally a tuple (another data structure in Python, which will be discussed later).

### 3.2.3 Reverse lookup

Given a dictionary `d` and a key `k`, it is easy to find the corresponding value `v = d[k]`. This operation is called a **lookup**. But what if you have `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search. Here is a function that takes a value and returns the first key that maps to that value:

```
def reverse_lookup(d, v):
 for k in d:
 if d[k] == v:
 return k
 raise LookupError()
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, `raise`. The **raise statement** causes an exception; in this case it causes a `LookupError`, which is a built-in exception used to indicate that a lookup operation failed. If we get to the end of the loop, that means `v` doesn't appear in the dictionary as a value, so we raise an exception. Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
And an unsuccessful one:
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 5, in reverse_lookup
LookupError
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message. The raise statement can take a detailed error message as an optional argument. For example:

```
>>> raise LookupError('value does not appear in the
dictionary')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
```

LookupError: value does not appear in the dictionary

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

### 3.2.4 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters. Here is a function that inverts a dictionary:

```
def invert_dict(d):
 inverse = dict()
 for key in d:
 val = d[key]
 if val not in inverse:
 inverse[val] = [key]
 else:
 inverse[val].append(key)
 return inverse
```

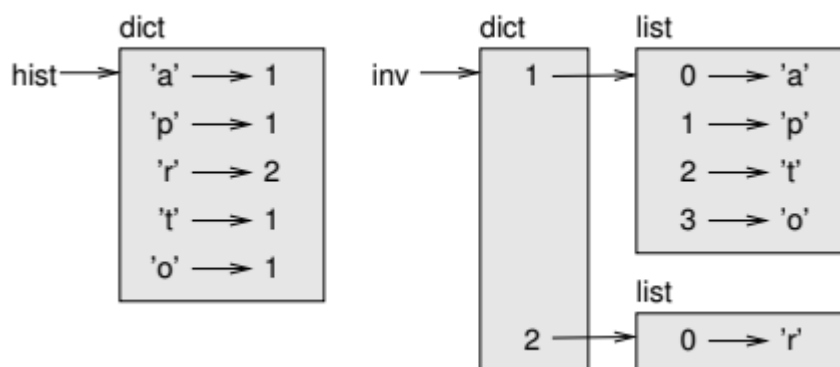


Figure 3.1: State diagram.



Each time through the loop, `key` gets a key from `d` and `val` gets the corresponding value. If `val` is not in `inverse`, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise, we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

Figure 3.1 is a state diagram showing `hist` and `inverse`. A dictionary is represented as a box with the type `dict` above it and the key-value pairs inside. If the values are integers, floats or strings, I draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple. Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**.

A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

### 3.2.5 Dictionaries and Files

A dictionary can be used to count the frequency of words in a file. Consider a file *myfile.txt* consisting of following text –

```
hello, how are you?
I am doing fine.
How about you?
```

Now, we need to count the frequency of each of the word in this file. So, we need to take an outer loop for iterating over entire file, and an inner loop for traversing each line in a file. Then in every line, we count the occurrence of a word, as we did before for a character. The program is given as below –

```
fname=input("Enter file name:")
try:
 fhand=open
n(fname)
except:
 print("File cannot be opened")
 exit()

d=dict()

for line in fhand:
 for word in line.split():
 d[word]=d.get(word,0)+1

print(d)
```

The output of this program when the input file is *myfile.txt* would be –

```
Enter file name: myfile.txt
{'hello,': 1, 'how': 1, 'are': 1, 'you?': 2, 'I': 1,
'am': 1, 'doing': 1, 'fine.': 1, 'How': 1, 'about': 1}
```

Few points to be observed in the above output –

- The punctuation marks like comma, full point, question mark etc. are also considered as a part of word and stored in the dictionary. This means, when a particular word appears in a file with and without punctuation mark, then there will be multiple entries of that word.
- The word ‘how’ and ‘How’ are treated as separate words in the above example because of uppercase and lowercase letters.

While solving problems on text analysis, machine learning, data analysis etc. such kinds of treatment of words lead to unexpected results. So, we need to be careful in parsing the text and we should try to eliminate punctuation marks, ignoring the case etc. The procedure is discussed in the next section.

### 3.2.6 Advanced Text Parsing

As discussed in the previous section, during text parsing, our aim is to eliminate punctuation marks as a part of word. The *string* module of

Python provides a list of all punctuation marks as shown –

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

The *str* class has a method *maketrans()* which returns a translation table usable for another method *translate()*. Consider the following syntax to understand it more clearly –

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

The above statement replaces the characters in *fromstr* with the character in the same position in *tostr* and delete all characters that are in *deletestr*. The *fromstr* and *tostr* can be empty strings and the *deletestr* parameter can be omitted.

Using these functions, we will re-write the program for finding frequency of words in a file.

```
import string

fname=input("Enter
file name:")try:
 fhand=open(fname)
except:
 print("File cannot be opened")
 exit()

d=dict()
for line in
 fhand:
 line=line.r
 strip()
 line=line.translate(line.maketrans('',' ',string.punctuati
on)) line=line.lower()

 for word in
 line.split():
 d[word]=d.get(wor
d,0)+1

print(d)
```

Now, the output would be –

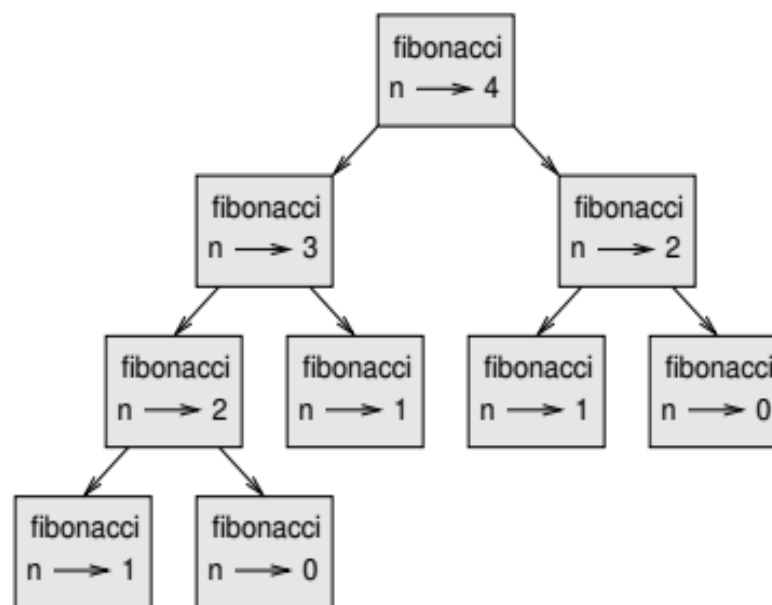
```
Enter file name:myfile.txt
{'hello': 1, 'how': 2, 'are': 1, 'you': 2, 'i': 1,
'am': 1, 'doing': 1, 'fine': 1, 'about': 1}
```

Comparing the output of this modified program with the previous one, we can make out that all the punctuation marks are not considered for parsing and also the case of the alphabets are ignored.

### 3.2.7 Memos

If you played with the fibonacci function from Section 6.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases quickly.

To understand why, consider Figure 3.2, which shows the **call graph** for fibonacci with  $n=4$ :



**Figure 3.2: Call graph**

A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fibonacci with  $n=4$  calls fibonacci with  $n=3$  and  $n=2$ . In turn, fibonacci with  $n=3$  calls fibonacci with  $n=2$  and  $n=1$ . And so on. Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger. One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**. Here is a “memoized” version of fibonacci:

```

known = {0:0, 1:1}
def fibonacci(n):
 if n in known:

```

```
return known[n]
res = fibonacci(n-1) + fibonacci(n-2)
known[n] = res
return res
```

known is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1. Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it. If you run this version of fibonacci and compare it with the original, you will find that it is much faster

### 3.2.8 Debugging

When we are working with big datasets (like file containing thousands of pages), it is difficult to debug by printing and checking the data by hand. So, we can follow any of the following procedures for easy debugging of the large datasets –

- **Scale down the input:** If possible, reduce the size of the dataset. For example if the program reads a text file, start with just first 10 lines or with the smallest example you can find. You can either edit the files themselves, or modify the program so it reads only the first n lines. If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you correct the errors.
- **Check summaries and types:** Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers. A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.
- **Write self-checks:** Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a **sanity check** because it detects results that are “completely illogical”. Another kind of check compares the results of two different computations to see if they are consistent. This is called a **consistency check**.
- **Pretty print the output:** Formatting debugging output can make it easier to spot an error.

## ***Hashing Technique (For curious minds – Only for understanding, not for Exams!!)***

Hashing is a way of representing dictionaries (Not a Python data structure Dictionary!!). Dictionary is an abstract data type with a set of operations searching, insertion and deletion defined on its elements. The elements of dictionary can be numeric or characters or most of the times, records. Usually, a record consists of several fields; each may be of different data types. For example, student record may contain student id, name, gender, marks etc. Every record is usually identified by some **key**. Hashing technique is very useful in database management, because it is considered to be very efficient searching technique.

Here we will consider the implementation of a dictionary of  $n$  records with keys  $k_1, k_2 \dots k_n$ . Hashing is based on the idea of distributing keys among a one-dimensional array

$H[0 \dots m-1]$ , called **hash table**.

For each key, a value is computed using a predefined function called **hash function**. This function assigns an integer, called **hash address**, between 0 to  $m-1$  to each key. Based on the hash address, the keys will be distributed in a hash table.

For example, if the keys  $k_1, k_2, \dots, k_n$  are integers, then a hash function can be  $h(K) = K \bmod m$ .

Let us take keys as 65, 78, 22, 30, 47, 89. And let hash function be,  
 $h(k) = k \% 10$ .

Then the hash addresses may be any value from 0 to 9. For each key, hash address will be computed as –

$$h(65) = 65 \% 10 = 5$$

$$h(78) = 78 \% 10 = 8$$

$$h(22) = 22 \% 10 = 2$$

$$h(30) = 30 \% 10 = 0$$

$$h(47) = 47 \% 10 = 7$$

$$h(89) = 89 \% 10 = 9$$

Now, each of these keys can be hashed into a hash table as –

|    |   |    |   |   |   |    |   |    |    |    |
|----|---|----|---|---|---|----|---|----|----|----|
| 0  | 1 | 2  | 3 | 4 | 5 | 6  | 7 | 8  | 9  |    |
| 30 |   | 22 |   |   |   | 65 |   | 47 | 78 | 89 |

In general, a hash function should satisfy the following requirements:

- A hash function needs to distribute keys among the cells of hash table as evenly as possible.
- A hash function has to be easy to compute.

**Hash Collisions:** Let us have  $n$  keys and the hash table is of size  $m$  such that  $m < n$ . As each key will have an address with any value between 0 to  $m-1$ , it is obvious that more than one key will have same hash address. That is, two or more keys need to be hashed into the same cell of hash table. This situation is called as **hash collision**.

In the worst case, all the keys may be hashed into same cell of hash table. But, we can avoid this by choosing proper size of hash table and hash function. Anyway, every hashing scheme must have a mechanism for resolving hash collision. There are two methods for hash collision resolution, viz.

- Open hashing
- closed hashing

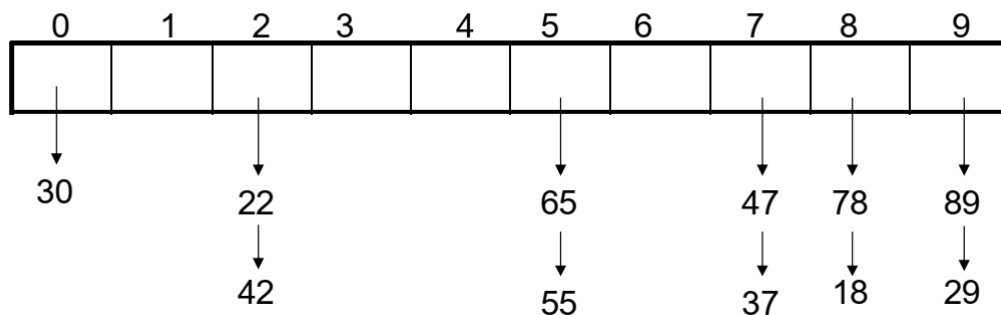
**Open Hashing (or Separate Chaining):** In open hashing, keys are stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell. For example, consider the elements

65, 78, 22, 30, 47, 89, 55, 42, 18, 29, 37.

If we take the hash function as  $h(k) = k \% 10$ , then the hash addresses will be

$$\begin{array}{ll} -h(65) = 65 \% 10 = 5 & h(78) = 78 \% 10 = 8 \\ h(22) = 22 \% 10 = 2 & h(30) = 30 \% 10 = 0 \\ h(47) = 47 \% 10 = 7 & h(89) = 89 \% 10 = 9 \\ h(55) = 55 \% 10 = 5 & h(42) = 42 \% 10 = 2 \\ h(18) = 18 \% 10 = 8 & h(29) = 29 \% 10 = 9 \\ h(37) = 37 \% 10 = 7 & \end{array}$$

The hash table would be –



### Operations on Hashing:

- **Searching:** Now, if we want to search for the key element in a hash

table, we need to find the hash address of that key using same hash function. Using the obtained hash address, we need to search the linked list by tracing it, till either the key is found or list gets exhausted.

- **Insertion:** Insertion of new element to hash table is also done in similar manner. Hash key is obtained for new element and is inserted at the end of the list for that particular cell.
- **Deletion:** Deletion of element is done by searching that element and then deleting it from a linked list.

**Closed Hashing (or Open Addressing):** In this technique, all keys are stored in the hash table itself without using linked lists. Different methods can be used to resolve hash collisions. The simplest technique is **linear probing**.

This method suggests to check the next cell from where the collision occurs. If that cell is empty, the key is hashed there. Otherwise, we will continue checking for the empty cell in a circular manner. Thus, in this technique, the hash table size must be at least as large as the total number of keys. That is, if we have  $n$  elements to be hashed, then the size of hashtable should be greater or equal to  $n$ .

Example: Consider the elements 65, 78, 18, 22, 30, 89, 37, 55, 42  
 Let us take the hash function as  $h(k) = k \% 10$ , then the hash addresses will be –

|                        |                        |
|------------------------|------------------------|
| $h(65) = 65 \% 10 = 5$ | $h(78) = 78 \% 10 = 8$ |
| $h(18) = 18 \% 10 = 8$ | $h(22) = 22 \% 10 = 2$ |
| $h(30) = 30 \% 10 = 0$ | $h(89) = 89 \% 10 = 9$ |
| $h(37) = 37 \% 10 = 7$ | $h(55) = 55 \% 10 = 5$ |
| $h(42) = 42 \% 10 = 2$ |                        |

Since there are 9 elements in the list, our hash table should at least be of size 9. Here we are taking the size as 10.

Now, hashing is done as below –

|    |    |    |    |   |    |    |    |    |    |
|----|----|----|----|---|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4 | 5  | 6  | 7  | 8  | 9  |
| 30 | 89 | 22 | 42 |   | 65 | 55 | 37 | 78 | 18 |

**Drawbacks:**

- Searching may become like a linear search and hence not efficient.



## MODULE 4

### TUPLES & FILES

#### Tuples are immutable

- A tuple<sup>1</sup> is a sequence of values much like a list.
- The values stored in a tuple can be any type, and they are indexed by integers.
- The important difference is that tuples are *immutable*.
- Tuples are also *comparable* and *hashable* so we can sort lists of them and use tuples as key values in Python dictionaries.
- Syntactically, a tuple is a comma-separated list of values

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
```

```
>>> type(t1)
```

```
<type 'tuple'>
```

Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
```

```
>>> type(t2)
```

```
<type 'str'>
```

Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
```

```
>>> print(t)
```

```
()
```

If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

## Comparing tuples

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called *DSU* for **Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence, **Sort** the list of tuples using the Python built-in sort, and **Undecorate** by extracting the sorted elements of the sequence.

[DSU]

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
txt = 'but soft what light in yonder window breaks'
```

```
words = txt.split()
```

```
t = list()
```

```
for word in words:
```

```
 t.append((len(word), word))
```

```
t.sort(reverse=True)
```

```
res = list()
```

```
for length, word in t:
```

```
 res.append(word)
```

```
print(res)
```

- The first loop builds a list of tuples, where each tuple is a word preceded by its length. sort compares the first element, length, first, and only considers the second element to break ties.
- The keyword argument reverse=True tells sort to go in decreasing order.
- The second loop traverses the list of tuples and builds a list of words in descending order of length.
- The four-character words are sorted in *reverse* alphabetical order, so “what” appears before “soft” in the following list.

The output of the program is as follows:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Of course the line loses much of its poetic impact when turned into a Python list and sorted in descending word length order.

## Tuple assignment

- One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement.
- This allows you to assign more than one variable at a time when the left side is a sequence.
- In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

```
>>> m = ['have', 'fun']
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:

```
>>> m = ['have', 'fun']
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
>>> m = ['have', 'fun']
>>> (x, y) = m
```

```
>>> x
'have'
>>> y
'fun'
>>>
```

A particularly clever application of tuple assignment allows us to *swap* the values of two variables in a single statement:

```
>>> a, b = b, a
```

- Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side.
- All the expressions on the right side are evaluated before any of the assignments.
- The number of variables on the left and the number of values on the right must be the same:

```
>>> a, b = 1, 2, 3
```

**ValueError:** too many values to unpack

More generally, the right side can be any kind of sequence (string, list, or tuple).

For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

## Tuples as return values

- A function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.

- For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute  $x/y$  and then  $x\%y$ .
- It is better to compute them both at the same time.
- The built-in function `divmod` takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)
```

```
>>> t
```

```
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
```

```
>>> quot
```

```
2
```

```
>>> rem
```

```
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
```

```
 return min(t), max(t)
```

`max` and `min` are built-in functions that find the largest and smallest elements of a sequence.

`min_max` computes both and returns a tuple of two values.

### Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with **\*** **gathers** arguments into a tuple. For example, `printall` takes any number of arguments and prints them:

```
def printall(*args):
```

```
 print(args)
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

```
>>> printall(1, 2.0, '3')
```

```
(1, 2.0, '3')
```

The complement of gather is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the \* operator.

For example, divmod takes exactly two arguments; it doesn't work with a tuple:

```
>>> t = (7, 3)
```

```
>>> divmod(t)
```

```
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
>>> divmod(*t)
```

```
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, max and min can take any number of arguments:

```
>>> max(1, 2, 3)
```

```
3
```

But sum does not.

```
>>> sum(1, 2, 3)
```

```
TypeError: sum expected at most 2 arguments, got 3
```

As an exercise, write a function called sumall that takes any number of arguments and returns their sum.

## Lists and tuples

zip is a built-in function that takes two or more sequences and returns a list of tuples where each tuple contains one element from each sequence.

The name of the function refers to a zipper, which joins and interleaves two rows of teeth.

This example zips a string and a list:

```
>>> s = 'abc'
```

```
>>> t = [0, 1, 2]
```

```
>>> zip(s, t)
```

```
<zip object at 0x7f7d0a9e7c48>
```

The result is a **zip object** that knows how to iterate through the pairs. The most common use of zip is in a for loop:

```
>>> for pair in zip(s, t):
```

```
... print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

A zip object is a kind of **iterator**, which is any object that iterates through a sequence.

Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

If you want to use list operators and methods, you can use a zip object to make a list:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one.

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a for loop to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
 print(number, letter)
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letter and number. The output of this loop is:

```
0 a
1 b
2 c
```

If you combine zip, for and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, `has_match` takes two sequences, `t1` and `t2`, and returns True if there is an index `i` such that `t1[i] == t2[i]`:



```
def has_match(t1, t2):
 for x, y in zip(t1, t2):
 if x == y:
 return True
 return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

```
for index, element in enumerate('abc'):
 print(index, element)
```

The result from `enumerate` is an `enumerate` object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence. In this example, the output is

```
0 a
```

```
1 b
```

```
2 c
```

Again.

## Dictionaries and tuples

Dictionaries have a method called `items` that returns a sequence of tuples, where each tuple is a key-value pair.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

The result is a `dict_items` object, which is an iterator that iterates the key-value pairs.

You can use it in a for loop like this:

```
>>> for key, value in d.items():
... print(key, value)
...
c 2
```

```
a 0
```

```
b 1
```

As you should expect from a dictionary, the items are in no particular order.

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
```

```
>>> d = dict(t)
```

```
>>> d
```

```
{'a': 0, 'c': 2, 'b': 1}
```

Combining dict with zip yields a concise way to create a dictionary:

```
>>> d = dict(zip('abc', range(3)))
```

```
>>> d
```

```
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method update also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

- It is common to use tuples as keys in dictionaries (primarily because you can't use lists).
- For example, a telephone directory might map from last-name, first-name pairs to telephone numbers.
- Assuming that we have defined last, first and number, we could write:

```
directory[last, first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

for last, first in directory:

```
print(first, last, directory[last,first])
```

- This loop traverses the keys in directory, which are tuples.
- It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.
- There are two ways to represent tuples in a state diagram.
- The more detailed version shows the indices and elements just as they appear in a list.
- For example, the tuple ('Cleese', 'John') would appear as in Figure a.
- But in a larger diagram you might want to leave out the details.

- For example, a diagram of the telephone directory might appear as in Figure b.
- Here the tuples are shown using Python syntax as a graphical shorthand.
- The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

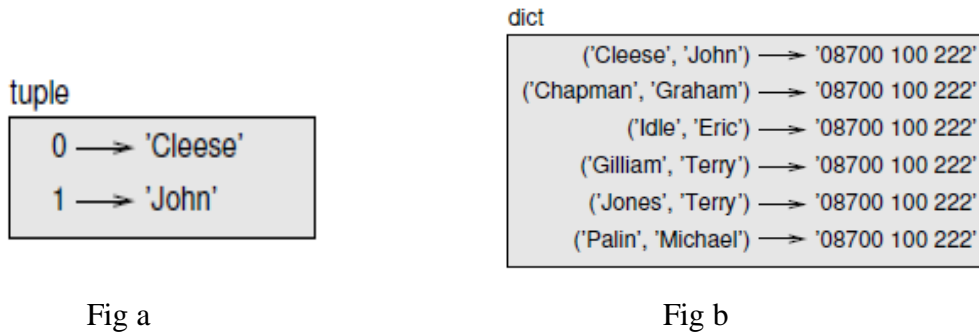


Figure: Stack diagram

## Multiple assignment with dictionaries

Combining items, tuple assignment, and for, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
for key, val in list(d.items()):
 print(val, key)
```

This loop has two *iteration variables* because items returns a list of tuples and key, val is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.

For each iteration through the loop, both key and value are advanced to the next key-value pair in the dictionary (still in hash order).

The output of this loop is:

```
10 a
22 c
1 b
```

Again, it is in hash key order (i.e., no particular order).

If we combine these two techniques, we can print out the contents of a dictionary sorted by the *value* stored in each key-value pair.

To do this, we first make a list of tuples where each tuple is (value, key). The items method would give us a list of (key, value) tuples, but this time we want to sort by value, not key. Once we have constructed the list with the value-key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
... l.append((val, key))
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

By carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

## Files

### Persistence

The CPU and memory are where our software works and runs. It is where all of the “thinking” happens.

But if you recall from our hardware architecture discussions, once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient fun exercises to learn Python.

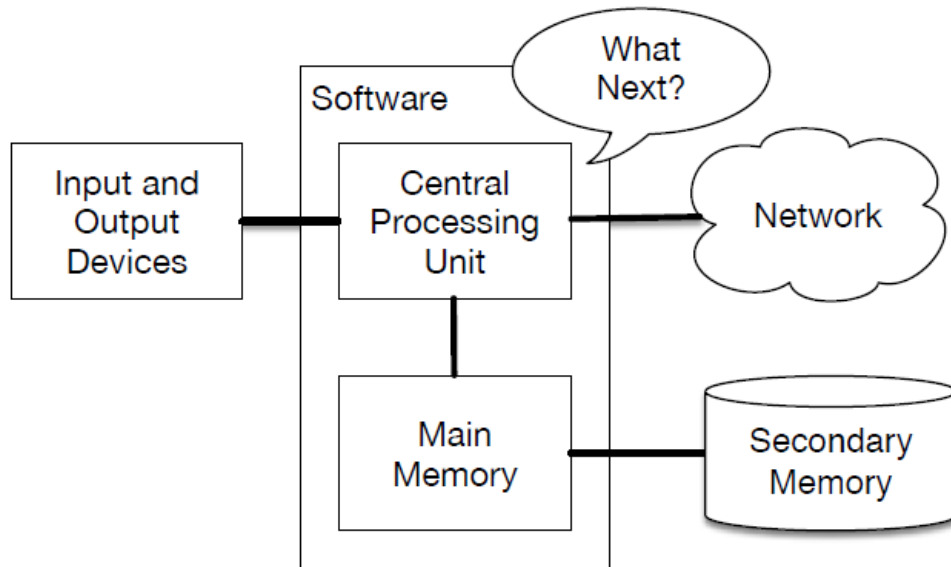


Figure: Secondary Memory

Secondary memory is not erased when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

We will primarily focus on reading and writing text files such as those we create in a text editor.

## Opening files

- When we want to read or write a file (say on your hard drive), we first must *open* the file. Opening the file communicates with your operating system, which knows where the data for each file is stored.
- When you open a file, you are asking the operating system to find the file by name and make sure the file exists.
- In this example, we open the file `mbox.txt`, which should be stored in the same folder that you are in when you start Python.

```
>>> fhand = open('mbox.txt')
```

```
>>> print(fhand)
```

```
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

- If the open is successful, the operating system returns us a *file handle*.
- The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data.
- You are given a handle if the requested file exists and you have the proper permissions to read the file.

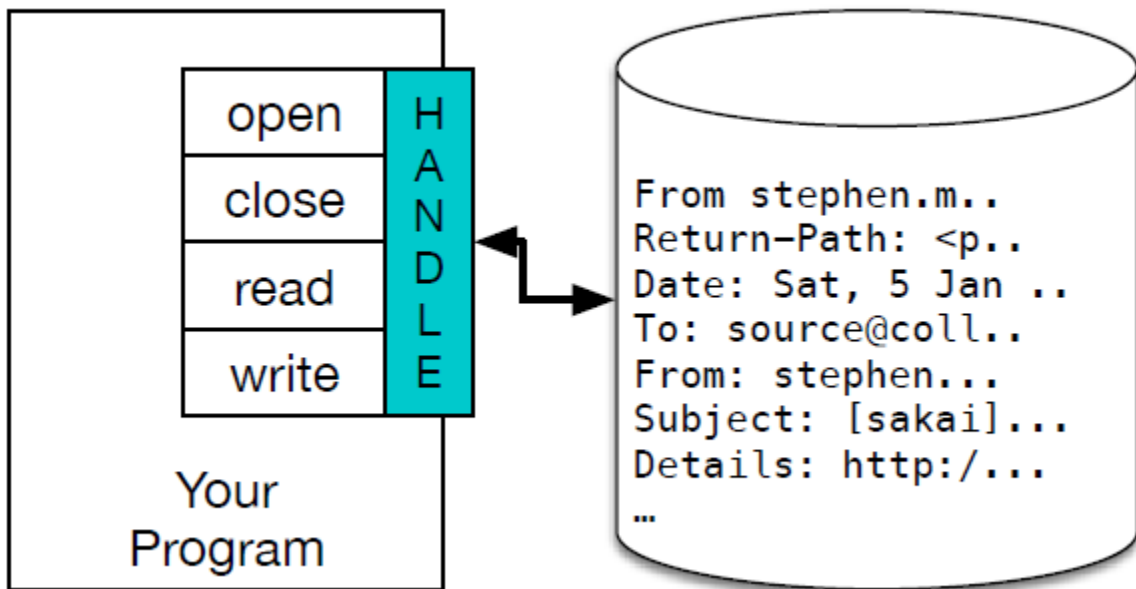


Figure 7.2: A File Handle

If the file does not exist, open will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'

## Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

To write a file, you have to open it with mode 'w' as a second parameter:

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

`open` returns a file object that provides methods for working with the file.

The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
```

```
>>> fout.write(line1)
```

```
24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call `write` again, it adds the new data to the end of the file.

```
>>> line2 = "the emblem of our land.\n"
```

```
>>> fout.write(line2)
```

```
24
```

When you are done writing, you should close the file.

```
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

## Format operator

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
```

```
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator.

But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
```

```
>>> '%d' % camels
```

```
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> I have spotted %d camels.' % camels
I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string.

Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

## Filenames and paths

Files are organized into **directories** (also called “folders”). Every running program has a “current directory”, which is the default directory for most operations.

For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories (“os” stands for “operating system”). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` stands for “current working directory”. The result in this example is `/home/dinsdale`, which is the home directory of a user named `dinsdale`.



A string like '/home/dinsdale' that identifies a file or directory is called a **path**.

A simple filename, like memo.txt is also considered a path, but it is a **relative path** because it relates to the current directory. If the current directory is /home/dinsdale, the filename memo.txt would refer to /home/dinsdale/memo.txt.

A path that begins with / does not depend on the current directory; it is called an **absolute path**.

To find the absolute path to a file, you can use os.path.abspath:

```
>>> os.path.abspath('memo.txt')
```

```
'/home/dinsdale/memo.txt'
```

os.path provides other functions for working with filenames and paths. For example,

os.path.exists checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
```

```
True
```

If it exists, os.path.isdir checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
```

```
False
```

```
>>> os.path.isdir('/home/dinsdale')
```

```
True
```

Similarly, os.path.isfile checks whether it's a file.

os.listdir returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
```

```
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
```

```
 for name in os.listdir(dirname):
```

```
 path = os.path.join(dirname, name)
```

```
 if os.path.isfile(path):
```

```
 print(path)
```

```
 else:
```

```
 walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

## Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

```
>>> fin = open('bad_file')
```

```
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
```

```
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
```

```
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

- To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if “Errno 21” is any indication, there are at least 21 things that can go wrong).
- It is better to go ahead and try—and deal with problems if they happen—which is exactly what the `try` statement does.
- The syntax is similar to an `if...else` statement:

`try:`

```
fin = open('bad_file')
```

`except:`

```
print('Something went wrong.')
```

- Python starts by executing the `try` clause.
- If all goes well, it skips the `except` clause and proceeds.
- If an exception occurs, it jumps out of the `try` clause and runs the `except` clause.
- Handling an exception with a `try` statement is called **catching** an exception.
- In this example, the `except` clause prints an error message that is not very helpful.
- In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

## Pickling

A limitation of dbm is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The pickle module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

`pickle.dumps` takes an object as a parameter and returns a string representation (dumps is short for “dump string”):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for pickle to interpret.

`pickle.loads` (“load string”) reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use pickle to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called `shelve`.

## Pipes

Most operating systems provide a command-line interface, also known as a **shell**.

Shells usually provide commands to navigate the file system and launch applications.

For example, in Unix you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a **pipe object**, which represents a running program.

For example, the Unix command `ls -l` normally displays the contents of the current directory in long format. You can launch `ls` with `os.popen1`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command.

The return value is an object that behaves like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print(stat)
```

None

The return value is the final status of the `ls` process;

None means that it ended normally (with no errors).

You can use a pipe to run `md5sum` from Python and get the result:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
>>> print(stat)
```

None

## Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
def linecount(filename):
 count = 0
 for line in open(filename):
 count += 1
 return count
print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7.

You can also import it like this:

```
>>> import wc
7
```

Now you have a module object `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

The module object provides `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom.

Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
 print(linecount('wc.py'))
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `'__main__'`; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

## MODULE 5

### Classes and objects, Inheritance

#### Programmer-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called Point that represents a point in two-dimensional space.

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, x and y.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

A programmer-defined type is also called a **class**. A class definition looks like this: `class Point:`  
`"""Represents a point in 2-D space."""`

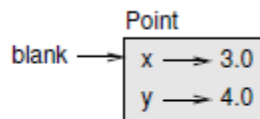


Figure : Object diagram.

The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named Point creates a **class object**.

```
>>> Point
<class '__main__.Point'>
```

Because Point is defined at the top level, its “full name” is `__main__.Point`.

The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to blank.

Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

Every object is an instance of some class, so “object” and “instance” are interchangeable.

But in this chapter I use “instance” to indicate that I am talking about a programmerdefined type.

### Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, “AT-trib-ute” is pronounced with emphasis on the first syllable, as opposed to “a-TRIB-ute”, which is a verb.

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**; see Figure

The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

You can read the value of an attribute using the same syntax:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

The expression `blank.x` means, “Go to the object `blank` refers to and get the value of `x`.” In the example, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>>> '%g, %g' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

You can pass an instance as an argument in the usual way. For example: `def print_point(p):`

```
print('%g, %g' % (p.x, p.y))
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

As an exercise, write a function called `distance_between_points` that takes two `Points` as arguments and returns the distance between them.

### Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles.

What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.



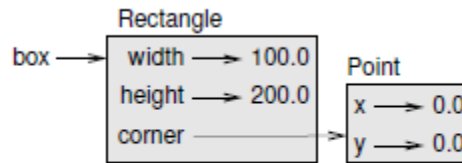


Figure : Object diagram.

Here is the class definition:

```

class Rectangle:
 """Represents a rectangle.
 attributes: width, height, corner.
 """

```

The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

```

The expression `box.corner.x` means, “Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`.”

### Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```

def find_center(rect):
 p = Point()
 p.x = rect.corner.x + rect.width/2
 p.y = rect.corner.y + rect.height/2
 return p

```

Here is an example that passes box as an argument and assigns the resulting Point to center:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

### Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a Rectangle object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
 rect.width += dwidth
 rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, `rect` is an alias for `box`, so when the function modifies `rect`, `box` changes.

As an exercise, write a function named `move_rectangle` that takes a Rectangle and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the x coordinate of corner and adding `dy` to the y coordinate of corner.

### Copying

Aliasing can make a program difficult to read because changes in one place might have

unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> import copy
>>> p2 = copy.copy(p1)
p1 and p2 contain the same data, but they are not the same Point.
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

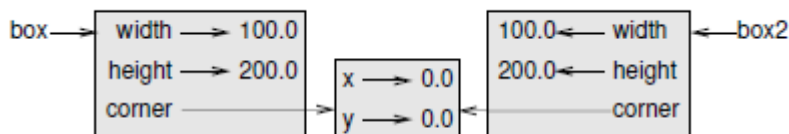


Figure : Object diagram.

The is operator indicates that p1 and p2 are not the same object, which is what we expected.

But you might have expected == to yield True because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle

object but not the embedded Point.

```
>>> box2 = copy.copy(box)
```

```
>>> box2 is box
```

```
False
```

```
>>> box2.corner is box.corner
```

```
True
```

For most applications, this is not what you want. In this example, invoking `grow_rectangle` on one of the Rectangles would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone. Fortunately, the `copy` module provides a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
```

```
>>> box3 is box
```

```
False
```

```
>>> box3.corner is box.corner
```

```
False
```

`box3` and `box` are completely separate objects.

## Inheritance

### Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit. It is not as obvious what type the attributes should be.

One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. In this context, “encode”

means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be “encryption”).

For example, this table shows the suits and the corresponding integer codes:

Spades 7! 3

Hearts 7! 2

Diamonds 7! 1

Clubs 7! 0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack 7! 11

Queen 7! 12

King 7! 13

I am using the 7! symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explicitly in the code.

The class definition for Card looks like this:

```
class Card:
 """Represents a standard playing card."""
 def __init__(self, suit=0, rank=2):
 self.suit = suit
 self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call Card with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

### **Class attributes**

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is

with lists of strings. We assign these lists to **class attributes**:

```
inside class Card:
suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
 '8', '9', '10', 'Jack', 'Queen', 'King']
def __str__(self):
 return '%s of %s' % (Card.rank_names[self.rank],
 Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object `Card`.

This term distinguishes them from variables like `suit` and `rank`, which are called **instance attributes** because they are associated with a particular instance.

Both kinds of attribute are accessed using dot notation. For example, in `__str__`, `self` is a `Card` object, and `self.rank` is its rank. Similarly, `Card` is a class object, and `Card.rank_names` is a list of strings associated with the class.

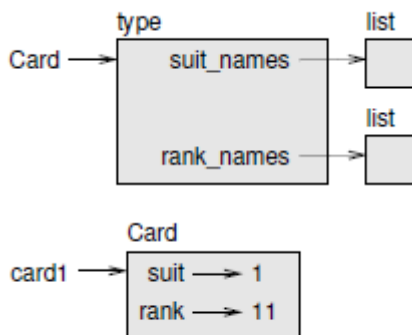


Figure : Object diagram.

Every card has its own suit and rank, but there is only one copy of `suit_names` and `rank_names`.

Putting it all together, the expression `Card.rank_names[self.rank]` means “use the attribute `rank` from the object `self` as an index into the list `rank_names` from the class `Card`, and select the appropriate string.”

The first element of `rank_names` is `None` because there is no card with rank zero. By including

None as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string '2', and so on. To avoid this tweak, we could have used a dictionary instead of a list.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

### Comparing cards

For built-in types, there are relational operators (<, >, ==, etc.) that compare values and determine

when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named `__lt__`, which stands for “less than”.

`__lt__` takes two parameters, `self` and `other`, and returns `True` if `self` is strictly less than `other`.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write `__lt__`:

```
inside class Card:
def __lt__(self, other):
 # check the suits
 if self.suit < other.suit: return True
 if self.suit > other.suit: return False
 # suits are the same... check ranks
 return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

```
inside class Card:
def __lt__(self, other):
 t1 = self.suit, self.rank
 t2 = other.suit, other.rank
 return t1 < t2
```

### Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for Deck. The init method creates the attribute cards and generates the standard set of fifty-two cards:

```
class Deck:
 def __init__(self):
 self.cards = []
 for suit in range(4):
 for rank in range(1, 14):
 card = Card(suit, rank)
 self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new Card with the current suit and rank, and appends it to self.cards.

### Printing the deck

Here is a \_\_str\_\_ method for Deck:

```
#inside class Deck:
def __str__(self):
 res = []

 for card in self.cards:
 res.append(str(card))
 return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method join. The built-in function str invokes the



`__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
```

```
>>> print(deck)
```

```
Ace of Clubs
```

```
2 of Clubs
```

```
3 of Clubs
```

```
...
```

```
10 of Spades
```

```
Jack of Spades
```

```
Queen of Spades
```

```
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

### **Add, remove, shuffle and sort**

To deal cards, we would like a method that removes a card from the deck and returns it.

The list method `pop` provides a convenient way to do that:

```
#inside class Deck:
```

```
def pop_card(self):
```

```
 return self.cards.pop()
```

Since `pop` removes the last card in the list, we are dealing from the bottom of the deck.

To add a card, we can use the list method `append`:

```
#inside class Deck:
```

```
def add_card(self, card):
```

```
 self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a **vener**. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case `add_card` is a “thin” method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

As another example, we can write a `Deck` method named `shuffle` using the function

shuffle from the random module:

```
inside class Deck:
def shuffle(self):
 random.shuffle(self.cards)
```

Don't forget to import random.

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance. To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

```
class Hand(Deck):
 """Represents a hand of playing cards."""
```

This definition indicates that Hand inherits from Deck; that means we can use methods like pop\_card and add\_card for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.

In this example, Hand inherits `__init__` from Deck, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize cards with an empty list.

If we provide an init method in the Hand class, it overrides the one in the Deck class:

```
inside class Hand:
def __init__(self, label=""):
 self.cards = []
 self.label = label
```

When you create a Hand, Python invokes this init method, not the one in Deck.

```
>>> hand = Hand('new hand')
```

```
>>> hand.cards
```

```
[]
```

```
>>> hand.label
```

```
'new hand'
```

The other methods are inherited from Deck, so we can use `pop_card` and `add_card` to deal a card:

```
>>> deck = Deck()
```

```
>>> card = deck.pop_card()
```

```
>>> hand.add_card(card)
```

```
>>> print(hand)
```

```
King of Spades
```

A natural next step is to encapsulate this code in a method called `move_cards`:

```
#inside class Deck:
```

```
def move_cards(self, hand, num):
```

```
 for i in range(num):
```

```
 hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a Hand object and the number of cards to deal. It modifies both `self` and `hand`, and returns `None`.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a Deck or a Hand, and `hand`, despite the name, can also be a Deck.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

### Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in, “a Rectangle has a Point.”
- One class might inherit from another. This relationship is called **IS-A**, as in, “a Hand is a kind of a Deck.”
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

A **class diagram** is a graphical representation of these relationships. For example, Figure shows the relationships between Card, Deck and Hand.

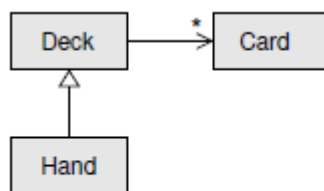


Figure : Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (\*) near the arrow head is a **multiplicity**; it indicates how many Cards a Deck has.

A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates

that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

A more detailed diagram might show that a Deck actually contains a list of Cards, but built-in types like list and dict are usually not included in class diagrams.

### **Data encapsulation**

The previous chapters demonstrate a development plan we might call “object-oriented design”. We identified objects we needed—like Point, Rectangle and Time—and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by **data encapsulation**.

Markov analysis, from Section 13.8, provides a good example. If you download my code from <http://thinkpython2.com/code/markov.py>, you’ll see that it uses two global variables—`suffix_map` and `prefix`—that are read and written from several functions.

```
suffix_map = { }
```

```
prefix = ()
```

Because these variables are global, we can only run one analysis at a time. If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here’s what that looks like:

```
class Markov:
```

```
 def __init__(self):
```

```
 self.suffix_map = { }
```

```
 self.prefix = ()
```

Next, we transform the functions into methods. For example, here’s `process_word`:

```
 def process_word(self, word, order=2):
```

```
 if len(self.prefix) < order:
```

```
self.prefix += (word,)
return

try:
self.suffix_map[self.prefix].append(word)
except KeyError:
if there is no entry for this prefix, make one
self.suffix_map[self.prefix] = [word]
self.prefix = shift(self.prefix, word)
```

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring (see Section 4.7).

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

As an exercise, download my Markov code from <http://thinkpython2.com/code/markov.py>, and follow the steps described above to encapsulate the global variables as attributes of a new class called Markov. Solution: <http://thinkpython2.com/code/Markov.py> (note the capital M).