UNIT II DATA, EXPRESSIONS, STATEMENTS

Python interpreter and interactive mode; values and types: int, float, Booleans, strings, and lists; variables, expressions, statements, tuple assignment, precedence of operators, comments; modules and functions, function definition and use, flow of execution, parameters and arguments; Illustrative programs: exchange the values of two variables, circulate the values of n variables, distance between two points.

2.1 INTRODUCTION

Guido van Rossum (Figure 1-26) is the creator of the Python programming language, first released in the early 1990s. Its name comes from a 1970s British comedy sketch television show called Monty Python's Flying Circus . The development environment IDLE provided with Python (discussed below) comes from the name of a member of the comic group.Python has a simple syntax. Python programs are clear and easy to read. At the same time, Python provides powerful programming features,and is widely used. Companies and organizations that use Python include YouTube, Google, Yahoo,and NASA.

Python is a powerful high-level, object-oriented programming language which is simple easy-to-use syntax, making it the perfect language for someone trying to learn computer programming for the first time.

2.1.1 What is Python (Programming)? - The Basics

Python is a general-purpose language. It has wide range of applications from Web development (like: Django and Bottle), scientific and mathematical computing (Orange, SymPy, NumPy) to desktop graphical user Interfaces (Pygame, Panda3D).

The syntax of the language is clean and length of the code is relatively short. It's fun to work in Python because it allows you to think about the problem rather than focusing on the syntax.

2.1.2 Features of Python Programming

1. **A simple language which is easier to learn**
   Python has a very simple and elegant syntax. It's much easier to read and write Python programs compared to other languages like: C++, Java, C#. Python makes programming fun and allows you to focus on the solution rather than syntax.
2. **Free and open-source**:
   You can freely use and distribute Python, even for commercial use. Not only can you use and distribute softwares written in it, you can even make changes to the Python's source code. Python has a large community constantly improving it in each iteration.

3. **Portability**

   You can move Python programs from one platform to another, and run it without any changes. It runs seamlessly on almost all platforms including Windows, Mac OS X and Linux.
4. **Extensible and Embeddable:**

   A high-level, interpreted language Unlike C/C++, you don't have to worry about daunting tasks like memory management, garbage collection and so on. Likewise, when you run Python code, it automatically converts your code to the language your computer understands. You don't need to worry about any lower-level operations.
5. **Large standard libraries to solve common tasks**

   Python has a number of standard libraries which makes life of a programmer much easier since you don't have to write all the code yourself. For example: Need to connect MySQL database on a Web server? You can use MySQLdb library using import MySQLdb. Standard libraries in Python are well tested and used by hundreds of people. So you can be sure that it won't break your application.
6. **Object-oriented**

   Everything in Python is an object. Object oriented programming (OOP) helps you solve a complex problem intuitively. With OOP, you are able to divide these complex problems into smaller sets by creating objects.

2.1.3 Applications of Python

- ✓ **Web Applications**

  You can create scalable Web Apps using frameworks and CMS (Content Management System) that are built on Python. Some of the popular platforms for creating Web Apps are: Django, Flask, Pyramid, Plone, Django CMS.Sites like Mozilla, Reddit, Instagram and PBS are written in Python.
- ✓ **Scientific and Numeric Computing**

  There are numerous libraries available in Python for scientific and numeric computing. There are libraries like: SciPy and NumPy that are used in general purpose computing. And, there are specific libraries like: EarthPy for earth science, AstroPy for Astronomy and so on.Also, the language is heavily used in machine learning, data mining and deep learning.
- ✓ **Creating software Prototypes**

  Python is slow compared to compiled languages like C++ and Java. It might not be a good choice if resources are limited and efficiency is a must. However, Python is a great language for creating prototypes. For example: You can use Pygame (library for creating games) to create your game's prototype first. If you like the prototype, you can use language like C++ to create the actual game.

✓ **Good Language to Teach Programming**

Python is used by many companies to teach programming to kids and newbies.It is a good language with a lot of features and capabilities. Yet, it's one of the easiest language to learn because of its simple easy-to-use syntax.

2.1.4 Reasons to Choose Python as First Language

1. **Simple Elegant Syntax**

   Programming in Python is fun. It's easier to understand and write Python code. **Why?** The syntax feels natural. Take this source code for an example:

   ```
   a = 2 b = 3
   sum = a +
   b
   print(sum)
   ```

   Even if you have never programmed before, you can easily guess that this program adds two numbers and prints it.

2. **Not overly strict**

   You don't need to define the type of a variable in Python. Also, it's not necessary to add semicolon at the end of the statement. Python enforces you to follow good practices (like proper indentation). These small things can make learning much easier for beginners.

3. **Expressiveness of the language**

   Python allows you to write programs having greater functionality with fewer lines of code. Here's a link to the source code of Tic-tac-toe game with a graphical interface and a smart computer opponent in less than 500 lines of code. This is just an example. You will be amazed how much you can do with Python once you learn the basics.

4. **Great Community and Support**

   Python has a large supporting community. There are numerous active forums online which can be handy if you are stuck. Some of them are:

   a. Learn Python subreddit
   b. Google Forum for Python
   c. Python Questions - Stack Overflow

Run Python on Your Operating System

You will find the easiest way to run Python on your computer (Windows, Mac OS X or Linux) in this section.

- **Install and Run Python in Mac OS X**
- **Install and Run Python in Linux (Ubuntu)**

Install and Run Python in Windows

1. Go to Download Python page on the official site and click **Download Python 3.6.0** (You may see different version name).
2. When the download is completed, double-click the file and follow the instructions to install it.When Python is installed, a program called IDLE is also installed along with it. It provides graphical user interface to work with Python.
3. Open IDLE, copy the following code below and press enter.
4. print("Hello, World!")
5. To create a file in IDLE, go to **File > New Window** (Shortcut: **Ctrl+N**).
6. Write Python code (you can copy the code below for now) and save (Shortcut: **Ctrl+S**) with **.py** file extension like: hello.py or your-first-program.py

```
print("Hello, World!")
```

7. Go to **Run > Run module** (Shortcut: **F5**) and you can see the output. Congratulations, you've successfully run your first Python program.

2.1.5 The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language ➢ There are two ways to use the interpreter:

  - **Interactive mode**

    In interactive mode, we type Python programs and the interpreter displays the result:

    ```
    >>> 1 + 1
    2
    ```

    The chevron, >>>, is the **prompt** the interpreter uses to indicate that it is ready.

  - **Script mode**.

    Alternatively, we can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**.

    Python scripts have names that end with .py.

2.1.6 First Python Program

Often, a program called "Hello, World!" is used to introduce a new programming language to beginners. A "Hello, World!" is a simple program that outputs "Hello, World!".

However, Python is one of the easiest language to learn, and creating "Hello, World!" program is as simple as writing print("Hello, World!"). So, we are going to write a different program.

Program to Add Two Numbers

Script.py # Add
two numbers num1
= 3 num2 = 5
sum = num1+num2 print(sum)

How this program works?

Line 1: # Add two numbers
Any line starting with # in Python programming is a comment.
Comments are used in programming to describe the purpose of the code. This helps you as well as other programmers to understand the intent of the code. Comments are completely ignored by compilers and interpreters.
Line 2: num1 = 3
Here, num1 is a variable. You can store a value in a variable. Here, 3 is stored in this variable.
Line 3: num2 = 5
Similarly, 5 is stored in num2 variable.
Line 4: sum = num1+num2
The variables num1 and num2 are added using **+** operator. The result of addition is then stored in another variable sum.
Line 5: print(sum)
The print() function prints the output to the screen. In our case, it prints 8 on the screen.

Few Important Things to Remember

To represent a statement in Python, newline (enter) is used. The use of semicolon at the end of the statement is optional (unlike languages like C/C++, JavaScript, PHP). In fact, it's recommended to omit semicolon at the end of the statement in Python.

Instead of curly braces { }, indentations are used to represent a block.

```
  im_a_parent:
im_a_child:
im_a_grand_child
```

```
    im_another_child:
im_another_grand_child
```

2.2 VALUES & TYPES

- A **value** is one of the basic thing with which a program works like a letter or a number.

  Example:

  - 1, 2 (integer)
  - ‘I Love Black‘ (String)

- **Type** tells us what a type a value is, if we are not sure of it.
- We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class. **Example:**

```
 >>> type('Are you ready')
 <type 'str'>

 >>> type(132) <type
 'int'>

 >>> type(55.34)
 <type 'float'>

 >>>a = 5
>>>print(a, "is of type", type(a))

 >>>a = 2.0
 >>>print(a, "is of type", type(a))

 >>>a = 1+2j
 >>>print(a, "is complex number?", isinstance(1+2j,complex))
```

Data types:

- ✓ Every value in Python has a data type. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.
- ✓ There are various data types in Python. Some of the important types are listed below.
  - Numbers
    - Integers
    - Floats

    - Complex numbers
    - Booleans
- ➢ Lists
- ➢ String
- ➢ Tuple
- ➢ Dictionaries

2.2.1 Numbers

Python’s four number types are integers, floats, complex numbers,and Booleans:

- *Integers*—1, –3, 42, 355, 888888888888888, –7777777777
- *Floats*—3.0, 31e12, –6e-4
- *Complex numbers*—3 + 2j, –4- 2j, 4.2 + 6.3j
- *Booleans*—True, False

We can manipulate them using the arithmetic operators: + (addition), – (subtraction),* (multiplication), / (division), ** (exponentiation), and % (modulus).

Example: 1(Integers)

```
>>> x = 5 + 2 - 3 * 2
>>> x
1

>>> 5 / 2
2.5


>>> 5 // 2
2

>>> 5 % 2
1

>>> 2 ** 8
256

>>> 1000000001 ** 3
1000000003000000003000000001
```

✓ Division of integers with / results in a float , and division of integers with // results in truncation. Note that integers are of unlimited size. ✓ They will grow as large as you need them to.

Example 2: (float)

```
>>> x = 4.3 ** 2.4
>>> x
33.137847377716483

>>> 3.5e30 * 2.77e45
9.6950000000000002e+75

>>> 1000000001.0 ** 3
1.000000003e+27
```

Example 3 :(Complex numbers)

```
>>> (3+2j) ** (2+3j)
(0.68176651908903363-2.1207457766159625j)

>>> x = (3+2j) * (4+9j)
>>> x
(-6+35j)

>>> x.real
-6.0

>>> x.imag
35.0
```

- ✓ Complex numbers consist of both a real element and an imaginary element, suffixed with a j. In the preceding code, variable x is assigned to a complex number.
- ✓ We can obtain its “real” part using the attribute notation x.real.

<u>Several built-in functions can operate on numbers</u>.

There are also the library module cmath (which contains functions for complex numbers) and the library module math (which contains functions for the other three types): Built In Function:

```
>>> round (3.49)
3

Library module function: >>>
import math
>>> math.ceil(3.49)
4
```

- ✓ Built-in functions are always available and are called using a standard function calling syntax. In the preceding code, round is called with a float as its input argument.

- ✓ The functions in library modules are made available using the import statement. In second example the math library module is imported, and it's ceil function is called using attribute notation:
  module. *function*(*arguments*)

Example 4: (Booleans)

```
>>> x = False
>>> x
False

>>> not x
True

>>> y = True * 2
>>> y
2
```

Other than their representation as True and False, Booleans behave like the numbers 1 (True) and 0 (False).

2.2.2 Lists

Python has a powerful built-in list type:

```
[ ]
[1]
[1, 2, 3, 4, 5, 6, 7, 8, 12]
[1, "two", 3L, 4.0, ["a", "b"], (5, 6)]
```

A list can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number. A list can be indexed from its front or back.

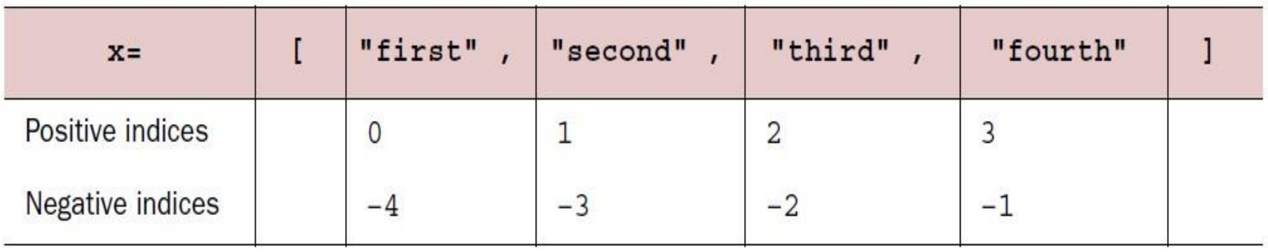
We can also refer to a sub segment, or *slice*, of a list using slice notation:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'
>>> x[2]
'third'
>>> x[-1]
'fourth'
>>> x[-2]
'third'
>>> x[1:-1]
['second', 'third']
```

```
>>> x[0:3]
['first', 'second', 'third']
>>> x[-2:-1]
['third']
>>> x[:3]
['first', 'second', 'third']
>>> x[-2:]
['third', 'fourth']
```

- ✓ Index from the front using positive indices (starting with 0 as the first element).
- ✓ Index from the back using negative indices (starting with -1 as the last element).
- ✓ Obtain a slice using [m:n] , where m is the inclusive starting point and n is the exclusive ending point (see table 3.1). An [:n] slice r starts at its beginning, and an [m:] slice goes to a list's end.

List indices



x=	[	"first" ,	"second" ,	"third" ,	"fourth"	]
Positive indices		0	1	2	3	
Negative indices		-4	-3	-2	-1	

You can use this notation to add, remove, and replace elements in a list or to obtain an element or a new list that is a slice from it:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[1] = "two"
>>> x[8:9] = []
>>> x
[1, 'two', 3, 4, 5, 6, 7, 8]
>>> x[5:7] = [6.0, 6.5, 7.0]
>>> x
[1, 'two', 3, 4, 5, 6.0, 6.5, 7.0, 8]
>>> x[5:]
[6.0, 6.5, 7.0, 8]
```

- ✓ The size of the list increases or decreases if the new slice is bigger or smaller than the slice it's replacing.
- ✓ Some built-in functions (len, max, and min), some operators (in, +,and *), the *del* statement, and the list methods (append, count, extend, index, insert, pop, remove, reverse, and sort) will operate on lists:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(x)
9
>>> [-1, 0] + x
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.reverse()
>>> x
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- ✓ The operators + and * each create a new list, leaving the original unchanged. A list's methods are called using attribute notation on the list itself: x.*method*(*arguments*).
- ✓ A number of these operations repeat functionality that can be performed with slice notation, but they improve code readability.

2.2.3 String

- ✓ String processing is one of Python's strengths. There are many options for delimiting strings:

```
"A string in double quotes can contain 'single quote' characters."
'A string in single quotes can contain "double quote" characters.'
'''\This string starts with a tab and ends with a newline character.\n'''
"""This is a triple double quoted string, the only kind that can contain
real newlines."""
```

- ✓ Strings can be delimited by single (' '), double (" "), triple single (''' '''), or triple double (""" """) quotations and can contain tab (\t) and newline (\n) characters.
- ✓ Strings are also immutable. The operators and functions that work with them return new strings derived from the original. The operators (in, +, and *) and built-in functions (len, max, and min) operate on strings as they do on lists and tuples.
- ✓ Index and slice notation works the same for obtaining elements or slices but can't be used to add, remove, or replace elements.
- ✓ Strings have several methods to work with their contents, and the re library module also contains functions for working with strings:

```
>>> x = "live and let \t \tlive"
>>> x.split()
['live', 'and', 'let', 'live']
>>> x.replace(" let \t \tlive", "enjoy life")
'live and enjoy life'
>>> import re
>>> regexpr = re.compile(r"[\t ]+")
>>> regexpr.sub(" ", x)
'live and let live'
```

The re module provides regular expression functionality. It provides more sophisticated pattern extraction and replacement capability than the string module.
The print function outputs strings. Other Python data types can be easily converted to strings and formatted:

```
>>> e = 2.718
>>> x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
>>> print("The constant e is:", e, "and the list x is:", x)
  The constant e is: 2.718 and the list x is: [1, 'two', 3, 4.0,['a', 'b'], (5, 6)]
>>> print("the value of %s is: %.2f" % ("e", e)) the
value of e is: 2.72
```

Objects are automatically converted to string representations for printing. The % operator provides a formatting capability similar to that of C’s sprintf.

2.2.4 Tuple

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, it looks like lists. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values.

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

It is common to enclose tuples in parentheses.

```
>>> t = ('a', 'b', 'c',’d’, 'e')
```

2.2.5 Dictionary

Dictionary is one of the compound data type like strings, list and tuple. Every element in a dictionary is the **key-value pair. EX:**
>>> empty={} >>> a={1:’apple’,2:’ball’}

2.3 VARIABLES

- ✓ A variable is a location in memory used to store some data (value).
- ✓ A variable is a name that refers to a value.
- ✓ They are given unique names to differentiate between different memory locations. The rules for writing a variable name is same as the rules for writing identifiers in Python.
- ✓ We don't need to declare a variable before using it. In Python, we simply assign a value to a variable and it will exist. We don't even have to declare the type of the variable. This is handled internally according to the type of value we assign to the variable.
- ✓ An **assignment statement** creates new variables and gives them values:

Example

```
>>> message = ‘I Like Icecreams’ >>> n =
17
>>> pi = 3.1415926535897932
```

The first assigns a string to a new variable named message; second gives the integer 17 to n; third assigns the (approximate) value of π to pi.

Rules for Variable names

- Name should be meaningful.
- Variable names can be arbitrarily long.
- They can contain both letters and numbers, but they have to begin with a letter.
- It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter
- The underscore character, _, can appear in a name. It is often used in names with multiple words, such as my_name .
- If you give a variable an illegal name, you get a syntax error

2.3.1 Python Keywords

- ✓ Keywords are the reserved words in Python.
- ✓ We cannot use a keyword as variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- ✓ In Python, keywords are case sensitive.
- ✓ There are 33 keywords in Python 3.3. This number can vary slightly in course of time.
- ✓ All the keywords except True, False and None are in lowercase and they must be written as it is.

  The list of all the keywords is given below.

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

2.3.2 Python Identifiers

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers. >>> global = 1

```
 File "<interactive input>", line 1
global = 1
      ^
SyntaxError: invalid syntax
```

4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
>>> a@ = 0
 File "<interactive input>", line 1
a@ = 0
  ^
SyntaxError: invalid syntax
```

5. Identifier can be of any length.

Variable assignment

We use the assignment operator (=) to assign values to a variable. Any type of value can be assigned to any valid variable.

```
a = 5 b =
3.2 c =
"Hello"
```

Here, we have three assignment statements. 5 is an integer assigned to the variable a. Similarly, 3.2 is a floating point number and "Hello" is a string (sequence of characters) assigned to the variables b and c respectively.

Multiple assignments

In Python, multiple assignments can be made in a single statement as follows: a,
b, c = 5, 3.2, "Hello"

If we want to assign the same value to multiple variables at once, we can do this as x
= y = z = "same"

This assigns the "same" string to all the three variables.

2.4 EXPRESSIONS

- ✓ An expression is a combination of values, variables, and operators.
- ✓ A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

```
17 x
x + 17
```

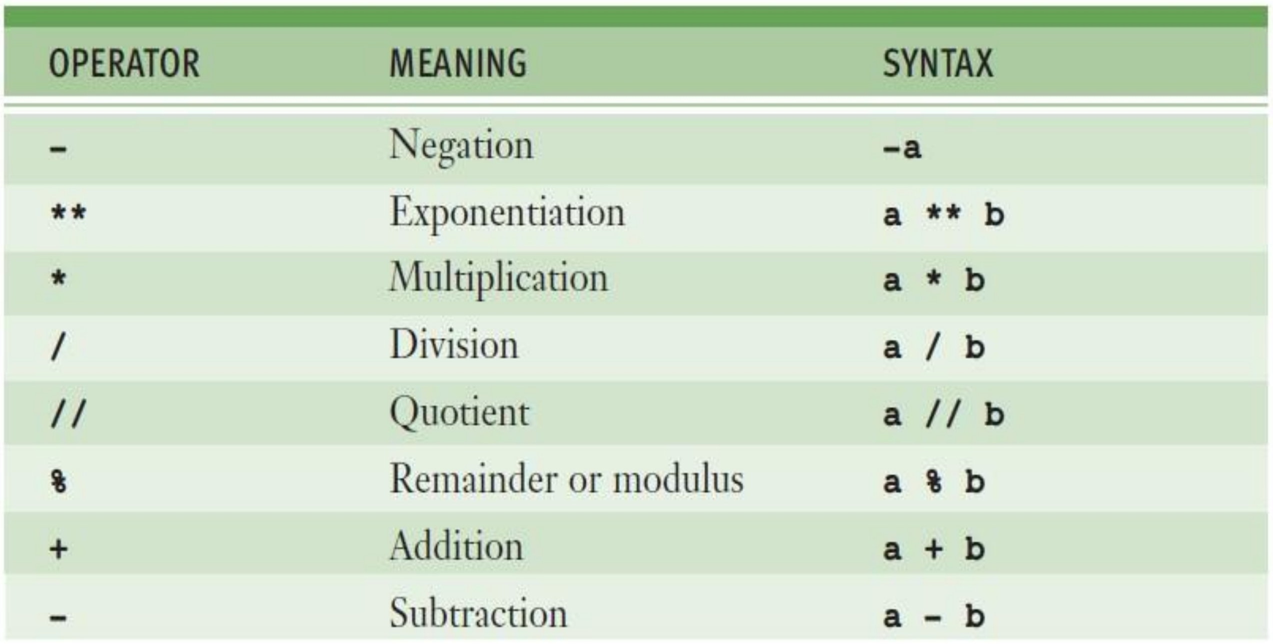
As we have seen, a literal evaluates to itself, whereas a variable reference evaluates to the variable's current value.

Expressions provide an easy way to perform operations on data values to produce other data values. When entered at the Python shell prompt, an expression's operands are evaluated, and its operator is then applied to these values to compute the value of the expression.

2.4.1 Arithmetic Expressions

An **arithmetic expression** consists of operands and operators combined in a manner that is already familiar to you from learning algebra. Table 2.6 shows several arithmetic operators and gives examples of how you might use them in Python code.

Arithmetic operators:

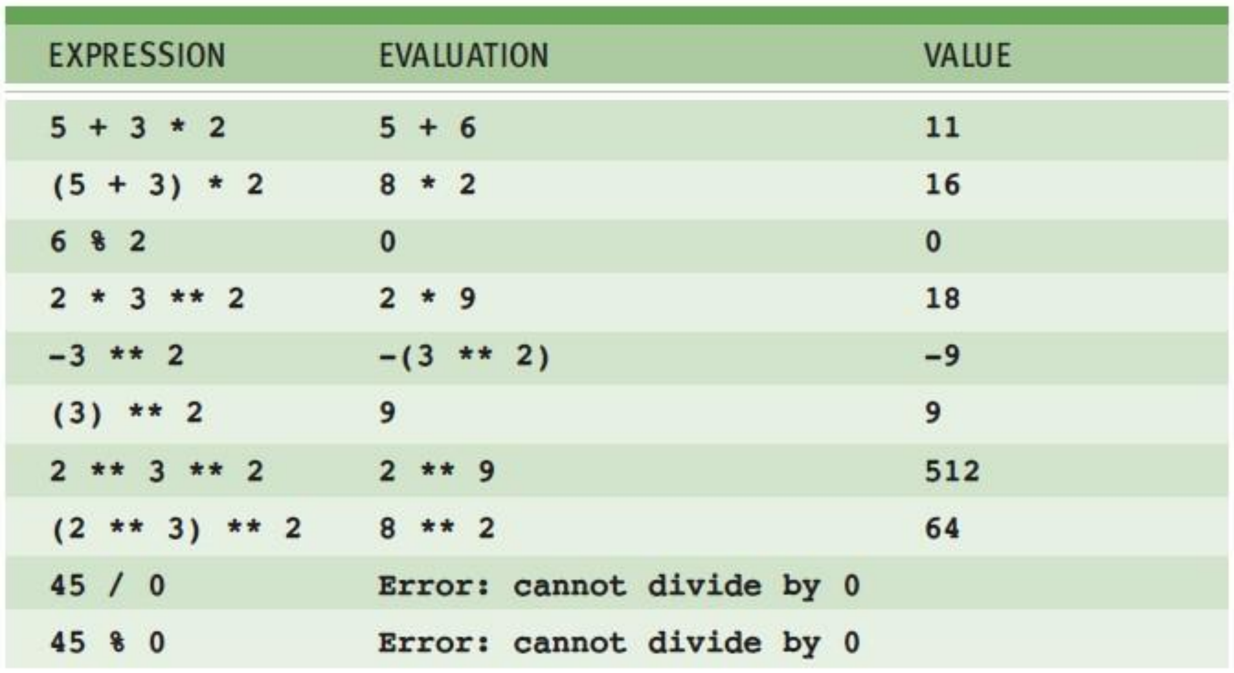


OPERATOR	MEANING	SYNTAX
-	Negation	-a
**	Exponentiation	a ** b
*	Multiplication	a * b
/	Division	a / b
//	Quotient	a // b
%	Remainder or modulus	a % b
+	Addition	a + b
-	Subtraction	a - b

- ✓ However, in Python, we must indicate multiplication explicitly, using the multiplication operator (*****), like this: **a * b**.

- ✓ Binary operators are placed between their operands (**a * b**, for example), whereas unary operators are placed before their operands (**-a**, for example).
- ✓ The **precedence rules** you learned in algebra apply during the evaluation of arithmetic expressions in Python:
  - ➢ Exponentiation has the highest precedence and is evaluated first.
  - ➢ Unary negation is evaluated next, before multiplication, division, and remainder.
  - ➢ Multiplication, both types of division, and remainder are evaluated before addition and subtraction.
  - ➢ Addition and subtraction are evaluated before assignment.
  - ➢ With two exceptions, operations of equal precedence are **left associative**, so they are evaluated from left to right. Exponentiation and assignment operations are **right associative**, so consecutive instances of these are evaluated from right to left.
  - ➢ We can use parentheses to change the order of evaluation.

Some arithmetic expressions and their values



EXPRESSION	EVALUATION	VALUE
5 + 3 * 2	5 + 6	11
(5 + 3) * 2	8 * 2	16
6 % 2	0	0
2 * 3 ** 2	2 * 9	18
-3 ** 2	-(3 ** 2)	-9
(3) ** 2	9	9
2 ** 3 ** 2	2 ** 9	512
(2 ** 3) ** 2	8 ** 2	64
45 / 0	Error: cannot divide by 0	
45 % 0	Error: cannot divide by 0	

- ✓ The last two lines of Table attempts to divide by 0, which result in an error. These expressions are good illustrations of the difference between syntax and semantics.
- ✓ **Syntax** is the set of rules for constructing well-formed expressions or sentences in a language.
- ✓ **Semantics** is the set of rules that allow an agent to interpret the meaning of those expressions or sentences.
- ✓ A computer generates a **syntax error** when an expression or sentence is not well formed.

- ✓ A **semantic error** is detected when the action that an expression describes cannot be carried out, even though that expression is syntactically correct. Although the expressions 45 / 0 and 45 % 0 are syntactically correct, they are meaningless, because a computing agent cannot carry them out.

2.4.2 Mixed-Mode Arithmetic and Type Conversions

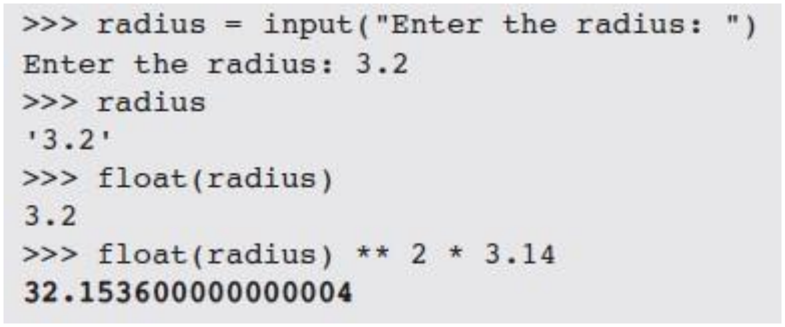
- ✓ When working with a handheld calculator, we do not give much thought to the fact that we intermix integers and floating-point numbers. Performing calculations involving both integers and floating-point numbers is called **mixed-mode arithmetic**.
- ✓ For instance, if a circle has radius 3, we compute the area as follows:

```
>>>3.14*3**2
28.26
```

- ✓ Python has different operators for quotient and exact division.
  For instance,

  3 // 2 * 5.0 yields **1 * 5.0**, which yields **5.0**
  Whereas
  3 / 2 * 5 yields **1.5 * 5**, which yields **7.5**

- ✓ In general, when you want the most precise results, you should use exact division.
- ✓ You must use a **type conversion function** when working with the input of numbers.
- ✓ A type conversion function is a function with the same name as the data type to which it converts. Because the **input** function returns a string as its value, you must use the function **int** or **float** to convert the string to a number before performing arithmetic, as in the following example:



```
>>> radius = input("Enter the radius: ")
Enter the radius: 3.2
>>> radius
'3.2'
>>> float(radius)
3.2
>>> float(radius) ** 2 * 3.14
32.153600000000004
```

Type conversion functions

CONVERSION FUNCTION	EXAMPLE USE	VALUE RETURNED
`int(<a number or a string>)`	`int(3.77)`	`3`
	`int("33")`	`33`
`float(<a number or a string>)`	`float(22)`	`22.0`
`str(<any value>)`	`str(99)`	`'99'`

- ✓ The **int** function converts a **float** to an **int** by truncation, not by rounding to the nearest whole number.
- ✓ Truncation simply chops off the number's fractional part.
- ✓ The **round** function rounds a **float** to the nearest **int** as in the next example:

```
>>> int(6.75)
6
>>> round(6.75)
7
```

- ✓ Another use of type conversion occurs in the construction of strings from numbers and other strings. For instance, assume that the variable **profit** refers
to a floating-point number that represents an amount of money in dollars and cents.
- ✓ Suppose that, to build a string that represents this value for output, we need to concatenate the **$** symbol to the value of **profit**.
- ✓ However, Python does not allow the use of the **+** operator with a string and a number:

```
>>> profit = 1000.55
>>> print('$' + profit)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

- ✓ To solve this problem, we use the str function to convert the value of profit to a string and then concatenate this string to the $ symbol, as follows:

```
>>> print('$' + str(profit))
$1000.55
```

- ✓ Python is a **strongly typed programming language**. The interpreter checks data types of all operands before operators are applied to those operands. If the type of an operand is not appropriate, the interpreter halts execution with an error message. This error checking prevents a program from attempting to do something that it cannot do.

2.5 STATEMENTS

- ✓ A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statement: print and assignment.
- ✓ The important difference is that an expression has a value; a statement does not.
- ✓ Instructions that a Python interpreter can execute are called statements. For example, a = 1 is an assignment statement. if statement, for statement, while statement etc. are other kinds of statements which will be discussed later.

2.5.1 Multi-line statement

- ✓ In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\).
- ✓ For example:

```
a = 1 + 2 + 3 + \
        4 + 5 + 6 + \
        7 + 8 + 9
```

- ✓ This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +                4 + 5 + 6 +
       7 + 8 + 9)
```

- ✓ Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
colors = ['red',
          'blue',
         'green']
```

- ✓ We could also put multiple statements in a single line using semicolons, as follows a = 1; b = 2; c = 3

2.5.2 Python Indentation

- ✓ Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.
- ✓ A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.
- ✓ Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

Ex.py

```
for i in range(1,11):
        print(i)
    if i == 5:
            break
```

The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent.

Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
    if True:
    print('Hello')
    a = 5
and if True: print('Hello'); a =
    5
```

both are valid and do the same thing. But the former style is clearer. Incorrect indentation will result into IndentationError.

2.5.3 Input, Output and Import

- ✓ Python provides numerous built-in functions that are readily available to us at the Python prompt.
- ✓ Some of the functions like input() and print() are widely used for standard input and output operations respectively. Let us see the output section first.
- ✓ Python Output Using print() function
- ✓ We use the print() function to output data to the standard output device (screen).
- ✓ We can also output data to a file, but this will be discussed later. An example use is given below.

Ex.py print('This sentence is output to the screen') # Output:

```
This sentence is output to the screen a = 5
print('The value of a is', a) # Output: The
value of a is 5
```

- In the second print() statement, we can notice that a space was added between the string and the value of variable a.This is by default, but we can change it.
- The actual syntax of the print() function is print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False) Here, objects is the value(s) to be printed.
- The sep separator is used between the values. It defaults into a space character.
- After all values are printed, end is printed. It defaults into a new line.
- The file is the object where the values are printed and its default value is sys.stdout (screen). Here are an example to illustrate this.

Ex.py print(1,2,3,4) # Output: 1

```
2 3 4
```

```
print(1,2,3,4,sep='*')
# Output: 1*2*3*4

print(1,2,3,4,sep='#',end='&')
# Output: 1#2#3#4&
```
Output formatting

- ✓ Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is {}'.format(x,y))
```

- ✓ The value of x is 5 and y is 10

Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

<u>Ex.py</u> print('I love {0} and {1}'.format('bread','butter'))

```
# Output: I love bread and butter

print('I love {1} and {0}'.format('bread','butter'))
# Output: I love butter and bread
```

- ✓ We can even use keyword arguments to format the string.

```
>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))
Hello John, Goodmorning
```

- ✓ We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Python Input

- Other than direct assignment to variables,it is also possible to get input from the user. In Python, we have the input() function to allow this. The syntax for input() is

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

- Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

- This same operation can be performed using the eval() function. But it takes it further. It can evaluate even expressions, provided the input is a string

```
>>> int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2+3'
>>> eval('2+3')
5
```

Python Import

- ✓ When our program grows bigger, it is a good idea to break it into different modules.
- ✓ A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py.
- ✓ Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.
- ✓ For example, we can import the math module by typing in import math.

Ex.py
```
import math
print(math.pi)
```

- ✓ Now all the definitions inside math module are available in our scope. We can also import some specific attributes and functions only, using the from keyword. For example:

```
>>> from math import pi
>>> pi
3.141592653589793
```

2.6 PRECEDENCE OF OPERATORS

- ✓ When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence.
- ✓ For mathematical operators, Python follows mathematical convention.
- ✓ The acronym PEMDAS is a useful way to remember the rules:
    - Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first,2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.
    - Exponentiation has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.
    - Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.
    - Operators with the same precedence are evaluated from left to right (except exponentiation).
    - So in the expression degrees / 2 * pi, the division happens first and the result is multiplied by pi. To divide by 2p, you can use parentheses or write degrees / 2 / pi.

2.7 COMMENTS

- ✓ It is a good idea to add notes to your programs to explain in natural language what the program is doing.

Single Line Comment:

- ✓ These notes are called comments, and they start with the # symbol:

```
# compute the percentage
percentage = (minute * 100) / 60
```

- ✓ In this case, the comment appears on a line by itself. ✓ You can also put comments at the end of a line:

```
v = 5 # assign 5 to v
```

- ✓ Everything from the # to the end of the line is ignored—it has no effect on the program.
- ✓ It is reasonable to assume that the reader can figure out what the code does; it is much more useful to explain why.
- ✓ Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out.

Multi-line comments

- ✓ If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

```
#This is a long comment
#and it extends
#to multiple lines
```

- ✓ Another way of doing this is to use triple quotes, either ''' or """. These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well.

Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a perfect
example of multi-line
comments"""
```

Docstring in Python

- ✓ Docstring is short for documentation string.
- ✓ It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring.
- ✓ Triple quotes are used while writing docstrings.

```
For example: def double(num):
   """Function to double the
value"""     return 2*num
```

- ✓ Docstring is available to us as the attribute __doc__ of the function. Issue the following code in shell once you run the above program.

```
>>> print(double.__doc__)

Function to double the value
```

2.8 MODUES & FUNCTIONS

2.8.1 Definition & Use

- ✓ A **module** is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

- ✓ Within a module, the module's name (as a string) is available as the value of the global variable __name__.
- ✓ For instance, use your favorite text editor to create a file called **fibo.py** in the current directory with the following contents:

fibo.py # Fibonacci numbers module

```
def fib(n):    # write Fibonacci series up to
n     a, b = 0, 1     while b < n:
        print(b, end=' ')
a, b = b, a+b     print()

def fib2(n):   # return Fibonacci series up to
n     result = []     a, b = 0, 1     while b < n:
result.append(b)          a, b = b, a+b     return
result
```

- ✓ Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

- ✓ This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

- ✓ If you intend to use a **function** often you can assign it to a local name:

```
              >>> fib = fibo.fib
              >>> fib(500)
      1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- ✓ In Python, function is a group of related statements that perform a specific task.
- ✓ Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- ✓ Furthermore, it avoids repetition and makes code reusable.

2.8.2 Syntax of Function

```
def function_name(parameters):

    """docstring"""

statement(s)
```

Above shown is a function definition which consists of following components.

1. Keyword def marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Example

```
def greet(name):

    """This function greets to

the person passed in as

     parameter"""

     print("Hello, " + name + ". Good morning!")
```

Function Call

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
Hello, Paul. Good morning!
```

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax

```
return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

Example:

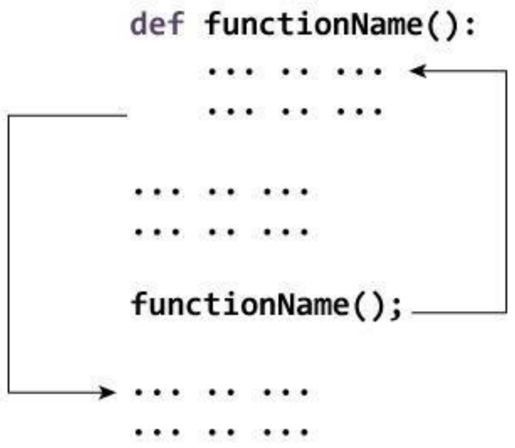
```
>>> print(greet("May")) Hello,
May. Good morning! None
```

Here, None is the returned value.

Example

```
def absolute_value(num):
        """This function returns the
absolute        value of the entered
number"""       if num >= 0:
                return num
        else:
        return -num
# Output: 2
print(absolute_value(2))
# Output: 4
print(absolute_value(-4))
```

2.8.3 How Function works in Python?



2.8.4 Fruitful functions and void functions

- ✓ Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them fruitful functions. Other functions, like print_twice, perform an action but don't return a value. They are called void functions.
- ✓ When you call a fruitful function, you almost always want to do something with the result;
- ✓ For example, you might assign it to a variable or use it as part of an expression: x = math.cos(radians) golden = (math.sqrt(5) + 1) / 2
- ✓ When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.2360679774997898
```

- ✓ But in a script, if you call a fruitful function all by itself, the return value is lost forever! math.sqrt(5)
- ✓ This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.
- ✓ **Void functions** might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result None
```

- ✓ The value None is not the same as the string 'None'. It is a special value that has its own type:

```
>>> print type(None)
<type 'NoneType'>
```

Import

- ✓ Python provides two ways to import modules; we have already seen one:

```
>>> import math
>>> print math
```

```
<module 'math' (built-in)>
>>> print math.pi
3.14159265359
```

- ✓ If you import math, you get a module object named math. The module object contains constants like pi and functions like sin and exp.
- ✓ But if you try to access pi directly, you get an error.

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

- ✓ As an alternative, you can import an object from a module like this:

```
>>> from math import pi
```

- ✓ Now you can access pi directly, without dot notation.

```
>>> print pi
3.14159265359
```

- ✓ Or we can use the star operator to import everything from the module:

```
>>> from math import *
>>> cos(pi) -1.0
```

- ✓ The advantage of importing everything from the math module is that your code can be more concise.
- ✓ The disadvantage is that there might be conflicts between names defined in different modules, or between a name from a module and one of your variables.

2.8.5 Flow of Execution

- ✓ In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution.
- ✓ Execution always begins at the first statement of the program. Statements are executed 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 are not executed 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, executes all 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 execute the statements in another function. But while executing that new function, the program might have to execute 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.

2.8.6 Parameters and Arguments

- ✓ Some of the built-in functions we have seen require arguments.
- ✓ For example, when you call math.sin you pass a number as an argument. Some functions take more than one argument:

  math.pow takes two, the base and the exponent.
- ✓ Inside the function, the arguments are assigned to variables called parameters. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
print bruce print
bruce
```

- ✓ This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter (whatever it is) twice.
- ✓ This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

- ✓ The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_twice:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

- ✓ The argument is evaluated before the function is called, so in the examples the expressions 'Spam '*4 and math.cos(math.pi) are only evaluated once. ✓ You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a
bee.' >>> print_twice(michael)
Eric, the half a bee. Eric, the half a
bee.
```

- ✓ The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn't matter what the value was called back home (in the caller); here in print_twice, we call everybody bruce.

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 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2) Bing
tiddle tiddle bang.
Bing tiddle tiddle bang.
```

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

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

2.9 ILLUSTRATIVE PROGRAMS

2.9.1 Python program to swap two variable

```
# To take input from the user

# x = input('Enter value of x: ')

# y = input('Enter value of y: ')

x = 5 y = 10

# create a temporary variable and swap the values

temp = x x = y y = temp print('The value of x after

swapping: {}'.format(x)) print('The value of y after

swapping: {}'.format(y))
```

2.9.2 Program to find distance between two points

```
import math p1=[2,4] p2=[3,6] distance=math.sqrt(((p2[0]-
p1[0])**2)+((p2[1]-p1[1])**2)) print("Distance between
two points:%d"%distance)
```

<u>Output:</u>

```
 Distance between two points:2
```

2.9.3 Circulate the values of n variables def

```
rotate(list,n):
    new=list[n:]+list[:n]
    return new example=[1,2,3,4,5]
print("Original list:",example)
a=rotate(example,1) print("List
rotated clockwise by 1:",a)
a=rotate(example,2) print("List
rotated clockwise by 1:",a)
a=rotate(example,-2) print("List
rotated clockwise by 1:",a) Output:
```

```
Original list: [1, 2, 3, 4, 5]
List rotated clockwise by 1: [2, 3, 4, 5, 1]
List rotated clockwise by 1: [3, 4, 5, 1, 2]
List rotated clockwise by 1: [4, 5, 1, 2, 3]
```