UNIT – II

3.1 Values and types

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'.

DATA TYPES

Python has six basic data types which are as follows:

1. **Numeric**
2. **String**
3. **List**
4. **Tuple**
5. **Dictionary**
6. **Boolean**

3.1.1. Numeric

Numeric data can be broadly divided into integers and real numbers (ie. Fractional numbers). Integers can be positive or negative. Python does not have any upper bound on the size of integers. The real numbers or fractional numbers are called **floating point** numbers. Such floating point numbers contain a decimal and a fractional part.

Example:

```
>>>num2                          # integer number
>>>num2=2.5                      # real number (float)
>>>num1
2                #output
>>>num2
2.5              #output
>>>
```

In python 2,

```
>>>5/2        #division operator
2
```

The result becomes a floating number when either the numerator or denominator is a floating number. When both the numerator and denominator are floating numbers, the result is again a floating

```
>>>5.0/2
2.5
```

In python 3

```
>>>5/2
2.5
```

3.1.2. String

- **Single quotes or double quotes are used to represent strings. A string in python can be a series or sequence of a alphabets, numerals and special characters .**

>>>s=”hello”

>>>s

hello

- **These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so-called because it contains a “string” of letters.**
- **If you are not sure what type a value has, the interpreter can tell you.**

>>> type('Hello, World!')

<type 'str'>

>>> type(17)

<type 'int'>

strings belong to the type **str** and integers belong to the type **int**.

Less obviously, numbers with a decimal point belong to a type called **float**, because these numbers are represented in a format called **floating-point**.

>>> type(3.2)

<type 'float'>

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

>>> type('17')

<type 'str'>

>>> type('3.2')

<type 'str'>

>>> 1,000,000

(1, 0, 0)

Python interprets 1,000,000 as a comma separated sequence of integers.

This is the first example we have seen of a semantic error:

the code runs without producing an error message, but it doesn't do the "right" thing.

String operations

- There are several operators such as slice operator ([]) and [:] , concatenation operator (+), repetition operator (*),etc.
- The slicing is used to take out a subset of the string, concatenation is used to combine two or more than two strings and repetition is used to repeat the same string several times.

Example:

>>> a=”hello”

>>>b=”world”

>>>c=a+b

>>>c

“helloworld”

>>>a*3

“hellohellohello”

Python provides slice operators ([] and :) to extract substring from the string.

In python, the indexing of the characters starts from 0; therefore, the index value of the first character is 0.

>>>string[start : end <:step>] # step is optional

>>> sample_string=”hello”

>>>sample_string[1] # display 1st index element

‘e’ #output

>>>sample_string[0:2] #display 0 to 1st index element

“he” #output

>>>sample_string=”helloworld”

>>>sample_string[1:8:2] #display all the alternate characters between index 1 to 8 ie. 1,3,5,7

“elwr”

3.1.3. List

- A list can contain same type of items. Alternatively, a list can also contain different types of items.
- A list is an ordered and indexable sequence.
- To declare list in python, we need to separate the items using commas and enclose them within square brackets ([]).
- The list is somewhat similar to the array in c Language.
- But, array contains only the same type of items while a list can contain different types of items.

Example:

```
>>>a=[1,2,3,4]
>>>b=[“hai”,hello”]
>>>c=[1,”hai”,2.3]
>>>a
[1,2,3,4]
>>>b
[‘hai’,’hello’]
>>>c
[1,’hai’,2.3]

>>>first =[1,”two”,3.0,”four”]
>>>second=[“five”,6]
>>> first
[1,’two’,3.0,’four’]
>>> first+second                    #concatenate first and second list
[1,’two’,3.0,’four’,’five’,6]
>>>second*3                        #repeat seond list 3 times
[‘five’6,’five’,6,’five’,6]
>>>first[0:2]                      #display sublist
[1,’two’]
```

3.1.4. Tuple

- A tuple is also used to store sequence of items.
- Like a list, a tuple consists of items separated by commas.
- However, tuple are enclosed within parentheses rather than within square brackets.

Difference between list and tupe.

S.NO	List	tuple
1	list items are enclosed within square brackets []	tuple items are enclosed within parentheses()
2	lists are mutable	tuples are immutables
3.	Items are stored in list can be modified.	tuples are read only list. I.e., once the items are stored, the tuple cannot be modified.

Example:

```
>>>a=(1,2,3,4)
>>>b=(‘hai’,’hello’)
>>>c=(1,’hai’,2.5)
>>>a
(1,2,3,4)
>>>b
(‘hai’,’hello’)
>>>c
(1,’hai’,2.5’)

>>>first[0]=”one”
>>>third[0]=”seven”
```

Syntax error will display because the item cannot be modified in a tuple but the same is not the case with.

3.1.5. Dictionary

- A dictionary is an unordered collection of key-value pairs.
- Keys and values can be of any type in a dictionary.
- Items in a dictionary are enclosed within curly-braces {} and separated by the commas (,).
- A colon (:) is used to separate key from value.
- A key inside the square bracket is used for accessing the dictionary items.
- It is same as the hash table type.
- The order of elements in a dictionary is undefined.

But, we can iterate over the following.

1. The keys
2. The values
3. The items (key-value pairs) in a dictionary.

Example:

```
>>>dict1={1:”first”,”second”:2}
>>>dict1[3]={3:”third”}

>>>dict1
{1:”first”,”second”:2,3:”third”}

>>>dict1.keys()
[1,”second”3]

>>>dict1.values()
[“first”,2,”third”]
```

3.1.6.Boolean

- Mostly data is stored in the form of alphanumeric.
- But sometimes we need to store the data in the form of ‘yes’ or ‘no’.
- In terms of programming language, yes is similar to True and no is similar to False.
- This True and False data is known as Boolean data.
- The data type which stores this Boolean data are known as Boolean Data types.

Example:

```
>>>a=True
 <type  ‘bool’>
>>>x=False
>>>type(x)
<type ‘bool’>
```

3.1.7. Sets

The lists and dictionaries are known as sequence or order collection of data.

A set is an unordered collection of data.

A set does not contain any duplicate values or elements.

Some Operations

- Union
- Intersection
- Difference
- Symmetric Difference

Union: Union operation performed on two sets returns all the elements from both the sets. It is performed by using | operator.

Intersection: Intersection operation performed on two sets returns all the elements which are common or in both the sets. It is performed by using & operator.

Difference: Difference operation performed on two sets set1 and set 2 returns the elements which are present on set1 but not in set2. It is performed by using **-** operator.

Symmetric difference: Symmetric difference operation performed on two sets set1 and set 2 returns the elements which are present in either set1 or set2 but not in both. It is performed by using ^ operator.

Example:

```
>>>s=set1([1,2,3,4,5,6])
>>>print  s
set([1,2,3,4,5,6])
>>>set1=set([1, 2, 4, 1, 2, 8, 5, 4])
>>>set2=set([1, 9, 3, 2,5])
>>>print set2
set([1, 2,3, 5,9])
>>>print  set1
Set([1,2,4,5,8])

>>>union=set1 | set2
>>>union
set([1,2,3,4,5,8,9])

>>>intersection= set1 & set2
>>>intersection
set([1,2,5])

>>>diff=set1 – set2
>>>diff
set([8,4])

>>>symm_diff= set1 ^ set2
>>> print symm_diff
set([3,4,8,9])
```

3.2 Variables

- A variable is a name that refers to a value.
- A variable holds a value that may change.
- The process of writing the variable name is called declaring the variable.
- In Python, variables do not need to declare explicitly in order to reserve memory spaces as in other programming languages.
- When we initialize the variable in Python, Python interpreter automatically does the declaration process.
- An **assignment statement** creates new variables and gives them values.

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

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.

3.2.1. Initializing a variable

The general format of assignment statement is as follows

Variable=expression

The equal sign (=) is known as Assignment Operator.

An expression is any value, text or arithmetic expression, whereas variable is the name of the variable.

The value of the expression will be stored in the variable.

Example:

>>> message =“Python Programming”

>>> n = 12

>>> pi = 3.1415926535897932

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

We can also assign different types of values to the same variable.

Example:

```
>>>a=50
>>>a
50
>>>a=’fifty’
>>>a
‘fifty’
```

3.3.1. Identifiers

A Python identifier is the name given to a variable, function, class, module or other object.

An identifier can begin with an alphabet (A-Z or a – z), or an underscore (_) and can include any number of letters, digits, or underscores.

Spaces are not allowed.

Python will not accept @, $ and % as identifiers.

Python is a case sensitive language.

Thus ***A*** and ***a*** both are different identifiers.

In Python, class name always start with capital letters.

Rules:

1. A variable name must start with a letter or the underscore character.
2. A variable name cannot start with a number.
3. A variable name can only contain alpha-numeric character and underscore (A-Z,a-z,0-9,_)
4. Variable names are case sensitive ie A not equal to a.

valid	Invalid
Sum_odd	Sum odd
a1	1a
myname	My name
N123	1N23%

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

3.3.2. Keywords

The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python keywords:

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

3.5 Operators and operands

Operators are special symbols that represent computations like addition and multiplication.

The values the operator is applied to are called **operands**.

Based on functionality, operators are categories into following seven types.

1. Arithmetic operators
2. Comparison operators
3. Assignment operators
4. Logical operators
5. Bitwise operators
6. Membership operators
7. Identity operators

3.5.1. Arithmetic operators

These operators are used to perform arithmetic operators such as addition, subtraction, multiplication and division.

operator	Description	Example
+	Addition operator toad two operands	10+20=30
-	Subtraction operator to subtract two operands	10-20=-10
*	Multiplication operator to multiply two operands	10*20=200
/	Division operator to divide left hand operator by right hand operator	5/2=2.5
**	Exponential operator to calculate remainder	5**2=25
%	Modulus operator to find remainder	5%2=1
//	floor division operator to find the quotient and remove the fractional part	5//2=2

3.5.2. Comparison Operator

These operators are used to compare values. Comparison operators are also called relational operators. The results of these operators is always Boolean value, ie, either true or false.

Operator	Description	Example
==	operator to check whether two operands are equal	10==20,false
!=	operator to check whether two operands are not equal	10!=20, true
<	operator to check whether first operand is less than second operand	10<20,true
>	operator to check whether first operand is greater than second operand	5>2,true
<=	operator to check whether first operand is less than or equal to second operand	5<=2, false
>=	operator to check whether first operand is greater than or equal to second operand	5>=2,true

3.5.3. Assignment operator

This operator is used to store the right hand side operand in the left hand side operand.

Operator	Description	Example
=	Assignment operator	A=9
+=	Add the right side operand with left side operand and store the result in left side	a+=b ➔a=a+b
-=	Subtract the right side operand from left side operand and store the result in left side	a-=b➔ a=a-b
*=	Multiply the right side operand with left side operand and store the result in left side	a*=b➔a=a*b
/=	Divide left side operand by right side operand and store the result in left side	a/=b➔ a=a/b
%=	Add the right side operand with left side operand and store the result in left side operand	a%b➔a=a%b
**=	Find the exponential and store the result in left side operand	a**=b ➔a=a**b
//=	Find the floor division and store the result in left side operand	a//=b=➔a=a//b

3.5.4. Bitwise Operator

These operators perform bit level operation on operand.

Let us take two operands x=10 and y=12.

Binary format of x and y are x=1010 and y=1100.

Operator	Description	Example
&	Bitwise AND	x&y, results 1000
|	Bitwise OR	x&y, results 1110
^	Bitwise XOR	x^y, results 0110
~	Bitwise inverse	~x,results0101
<<	Left shift	x<<2,results 101000
>>	Right shift	x>>2,results 0010

Example:

```
>>>x=10                #10= 0000  1010
>>>y=12                #12= 0000  1100
>>>z=0
>>>z=x&y
>>>print z
8                                  #8= 0000  1000
>>>z=x|y
>>>print z
14                                 #14= 0000  1110
>>>z=x^y
>>>print z
6                                  #8= 0000  0110
>>>z=~x
>>>print z
-11                                #-11= 1111  0101
>>>z=x<<2
>>>print z
40                                 #40= 0010  1000
>>>z=x>>2
>>>print z
2                                  #2= 0000  0010
```

3.5.5. Logical Operator

These operators are used to check two or more conditions. The resultant of these operators is Boolean value.

Operator	Description	Example
and	Logical and	x and y = If x is true and y is true then result is true otherwise result is false
or	Logical or	x or y = If any one true then result is true otherwise result is false
not	Logical not	not x= if x is ture then result is false otherwise result is true

Example:

```
>>>x=True
>>>y=False
>>>print (x and y)
False
>>>print(x or y)
True
>>>print(not x)
True
```

3.5.6. Membership Operators

These operators are used to check an item or an element that is part of a string, a list or a tuple.

A membership operator reduces the effort of searching an element in the list.

Operator	Description	Example
in	Return true, it item is in list or in sequence. Return false, if item is not in list or in sequence	x in y, results true
not in	Return false, it item is in list or in sequence. Return true, if item is not in list or in sequence	x not in y, results false

Example:

```
>>>x=10
>>>y=12
>>>list=[21,13,10,17]
>>>if (x in list):
        print “x  is present in list”
    Else
        print “ x is not in list”
```

Output

x is present in list

```
>>>if (y not in list):
        print “y  is not present in list”
    Else
        print “ y is present  in list”
```

Output

y is not present in list

3.5.7. Identity operators

These operators are used to check whether both operands are same or not.

operator	Description
is	Return true, if the operands a re same. Otherwise return false
is not	Return false, if the operands are same. Otherwise return true

Example:

```
>>>x=12
>>>y=12
>>>if  (x is y):
                    print “x is same as y”
      else:
                    print “x is not same as y”
```

Output

x is same as y

```
>>>x=12
>>>y=12
>>>if  (x is not y):
                    print “x is not same as y”
      else:
                    print “x is same as y”
```

Output

x is same as y

3.5.8. Precedence of Operators

The following table shows the precedence from lower to higher.

Operator	Description
NOT, OR,AND	Logical operators
in, not in	Membership operator
is, not is	Identity operator
=,%=,/=,//=,-=,+=,*=,**=	Assignment operator
<>,==,!=	Equality comparison operator
<=,<.>,>=	Comparison operators
^,|	Bitwise operators
&	Bitwise AND operator
<<,>>	Bitwise left shift and right shift
+,-	Addition and subtraction operator
*,/,%,//	Multiplication, division, modulus and floor division
**	Exponential operator

3.6. Expressions and Statements

- An **expression** is a combination of values, variables, and operators.
- The following are all legal expressions (assuming that the variable x has been assigned a value):

```
17
x
x + 17
```

- A **statement** is a unit of code that the Python interpreter can execute.
- Two kinds of statement: print and assignment.
- Technically an expression is also a statement.
- The important difference is that an expression has a value; a statement does not.

3.7. Boolean Expression

A Boolean expression may have only one of two values: True or False.

Example:

```
>>>5==5
True
>>>5==6
False
>>>True
True
>>>False
False
```

3.11 Comments

Single line comments

- It is a good idea to add notes to your programs to explain in natural language what the program is doing.
- These notes are called **comments**, and they start with the # symbol:
- **# compute the percentage of the hour that has elapsed**
- percentage = (minute * 100) / 60
- Everything from **the # to the end of the line is ignored—it has no effect** on the program.
- Comments are most useful when they document non-obvious features of the code.

Multiline Comments

To add a multiline comment, you could insert'#' for each line or not quite as intended. Since the Python will ignore string literals that are not assigned to a variable, you can add a multi string **(triple quotes)** in your code and place your comment inside it.

Example

```
“““
Hello
Hai
Good moring
”””
```

Functions

4.1. Introduction

Functions are self-contained programs that perform some particular tasks.

Once a function is created by the programmer for a specific task, this function can be called anytime to perform that task.

Each function is given a name, using which we call it.

A function may or may not return a value.

There are many built-in functions

dir(), len(),abs(),etc.

Users can also built their own functions, which are called user-defined functions.

Advantages

There are many advantages of using functions.

1. They reduce duplication of code in a program.
2. They break the large complex problems into small parts.
3. They help in improving the clarity of code.
4. A piece of code can be reused as many times as we want with the help of functions.

4.2 Function calls

- A **function** is a named sequence of statements that performs a computation.
- When you define a function, you specify the name and the sequence of statements.
- Later, you can “call” the function by name.

We have already seen one example of a **function call**:

```
>>> type(32)
<type 'int'>
```

- The name of the function is type.
- The expression in parentheses is called the **argument** of the function.
- The result is the type of the argument.
- It is common to say that a function “takes” an argument and “returns” a result.
- The result is called the **return value**.

4.3 Type conversion functions

Python provides built-in functions that convert values from one type to another.
The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, str converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4. Type coercion

Type conversion discussed above is known as explicit type conversion.

There is another kind of type conversion in Python language, known as implicit conversion.

Implicit conversion is also known as type coercion and is automatically done by the interpreter.

Type coercion is a process through which the python interpreter automatically converts a Value of one type into a value of another type according to the requirement.

Example:

Suppose we want to calculate an elapsed fraction of an hour.

The expression min/60 does integer division.

Solution 1: **Type Conversion**

```
>>>min=59
>>>float(min)/60
0.98333333333
```

Solution 2: Type Coercion

```
>>>min=59
>>>min/60.0
0.98333333333
```

4.5 Math functions

Python has a math module that provides most of the familiar mathematical functions.

A **module** is a file that contains a collection of related functions.

Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a **module object** named math.

If you print the module object, you get some information about it:

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

The module object contains the functions and variables defined in the module.

To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

4.6. Date and Time

Built-in module time and calendar through which we can handle date and time in several ways.

We can use these modules to get the current time and date.

In order to use the time module, we need to import it into our program first.

Similarly, we need to import calendar module for date.

Example:

Getting current date and time

>>>import time

>>> localtime=time.location(time.time())

>>>print " Local current time:",local time

Output

Local current time: time.struct_time(tm_year=2020, tm+month=10, tm_mday=6, tm_hour=18,tm_min=10,tm_sec=30,tm_wday=1,tm_yday=152,tm_isdst= 0)

This function returns a time-tuple with nine item. If we want we can change it.

Getting formatted date and time

```
>>> import time
>>>localtime=time.asctime(time.localtime(time.time())
>>>print  localtime
```

Output

Tue Oct 6 14:29:43 2020

asctime () function is used to get a readable format of date and time.

Getting calendar for a year

```
>>> import calendar
>>>c=calendar.month (2020,10)
```

Output

October 2020

```
Mo Tu We Th Fr Sa Su
          1    2    3    4
 5    6    7   8   9  10 11
12  13  14  15  16 17  18
19  20  21  22  23 24  25
26  27  28  29  30 31
```

4.7 dir() function

dir () function takes an object as argument.

It returns a list of strings which are names of members of that object.

If object is module, it will list sub-modules, functions provided by, variables, constants, etc.

Example

>>>**dir(math)**

>>>list1=dir(math)

>>>print list1

Output

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

4.8. help() function

help() function is a built-in function which is used to invoke the help system.

It takes an object as an argument.

It gives all detailed information about that object like if it's a module, then it will tell you about the sub-modules, functions, variables and constants in details

Example:

```
import math
help(math.sin)
```

output

```
Help on built-in function sin in module math:
sin(x, /)
    Return the sine of x (measured in radians).
```

4.10. User Defined Functions (Adding new functions)

- A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called.
- The block of function starts with a keyword **def** after which the function name is written followed by parentheses.
- The keyword def indicates that this is a function definition.

The rules for function names are the same as for variable names:

letters, numbers and some punctuation marks are legal, but the first character can't be a number.

- can't use a keyword as the name of a function.
- should avoid having a variable and a function with the same name.
- We can also give some input parameters or arguments to a function by placing them within these parentheses.
- The parameter can also be defined within these parentheses.
- The empty parentheses after the name indicate that this function doesn't take any arguments.

The first line of the function definition is called the **header**; the rest is called the **body**.

The header has to end with a colon and the body has to be indented.

By convention, the indentation is always four spaces.

The body can contain any number of statements.

The block of statements always start with a colon(:).

After writing the code statements, the block is ended with a return statement whose syntax is return [expression].

A function may or may not return a value.

If you want to return more than one value, separate the values using commas.

The default return value is NONE.

Syntax

def functionname(parameters):

“function _docstring”

Statement(s)

return [expression]

_docstring does not affect the program execution because it is a string literal which is used to document a specific part of the code.

Example:

```
>>> def print_lines():
        print (“ Hello Python”)
        print (“Welcome to Python Programming”)
        ……
```

Defining a function creates a variable with the same name.

```
>>>print print_lines
<function print_lines at 0x0294D970>
>>>type(print_line)
<type ‘function’>
```

The value of print_lines is a **function object**, which has type 'function'.

The syntax for calling the new function is the same as for built-in functions:

```
 >>> print print_lines
Hello Python
Welcome to Python Programming
```

Once you have defined a function, you can use it inside another function. Example

```
>>>def new_print():
        print_lines()
       print_lines()
>>>new_print()
```

Output

```
Hello Python
Welcome to Python Programming
Hello Python
Welcome to Python Programming
```

4.11 Definitions and uses

```
>>>def new_print():
        print_lines()
        print_lines()

>>> def print_lines():
            print (“ Hello Python”)
            print (“Welcome to Python programming”)
>>>new_print()
```

Output
Hello Python
Welcome to Python Programming
Hello Python
Welcome to Python Programming

4.12 Flow of execution

- **flow of execution :** The order in which statements are executed.
- 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.
- 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!
- 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.

4.13 Parameters and Arguments

math.sin → 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**.

At the time of function definition, we have to define some parameters if that function requires some be passed at the time of calling.

Here is an example of a user-defined function that takes an argument:

Example:

```
>>>def print_lines(line):
        print (line)
        print (line)
```

In this function we have defined a variable line which is a parameter.

Now, when the function is called, it prints the value of the parameter line twice.

```
>>> print_lines("Hello")
Hello
Hello
```

```
>>>print_lines(15)
15
15
>>print_lines(math.pi)
3.1415965359
3.1415965359
```

The print_lines() function works with any type of arguments that can be displayed.

There are four types of formal arguments using which a function can be called which are as follows.

1. **Required arguments.**
2. **Keyword arguments**
3. **Default arguments**
4. **Variable-length arguments**

Required arguments

When function defined with arguments, then at the time of calling the function the user must pass arguments in correct positional order and no. of arguments.

The no. of arguments should match the no. of arguments defined.

Example:

```
>>>def print_lines(str):
        print (str)
        return;
```

When we defined one parameter str to the function print-lines(). Hence, at the time of calling, we have to pass exactly one argument to the function.

Otherwise it will provide an error.

#function calling with one argument

```
>>>print_lines(“Hello”)
```

Output

Hello

#function calling with no argument

>>>print_lines()

Traceback (most recent call last):

File “<pyshell#18>,line 1,in <module>

print_lines()

TypeError: print_lines() takes exactly 1 argument (0 given)

There is an error because we did not pass any argument to the function print_lines(), while according to the function definition, the function print_lines() must take exactly one argument.

Keyword arguments

In keyword arguments, the caller recognizes the arguments by the parameter's names.

This type of argument can also be skipped or can also be out of order.

Example:

#function definition

```
>>>def print_lines():
                print (str)
                return;
```

 # function calling

```
>>>print_lines(str="Hello Python")
```

Output

Hello Python

Example2:

#function definition

```
>>>def print_lines(name,age):
                print (“Name:”,name)
                print (“Age:”, age)
                return
```

#function calling

```
 >>>print_lines(age=18,name=”anu”)
```

Output

Name:anu

Age:18

Default arguments

In default arguments, we can assign a value to a parameter at the time of function definition.

This value is considered the default value to that parameter.

If we do not provide a value to the parameter at the time of calling, it will not produce error.

Instead it will pick the value and use it.

Example :

#function definition

```
>>>def print_lines(name,age=20)
                    print “Name:”,name
                    print “Age:”, age
                    return
```

#function calling

```
 >>>print_lines(age=18,name=”anu”)
```

Output

Name:anu

Age:18

#function calling

>>>print_lines(name=”anu”)

Output

Name:anu

Age:20

we have assigned 20 to the argument age.

It is treated as the default argument.

In the first function call, we provide the vale 18 to the age argument.

Hence the Python interpreter takes the value provided by us and does not use the default value.

In the second function call, we don’t provide the value for age.

So, Python interpreter takes default value 20 to the age and does not provide any error message.

Variable-length arguments

Suppose we are required to process a function with more number of arguments than we specified in the function definition.

These types of arguments are known as variable-length arguments.

These names of these arguments are not specified in the function definition.

Instead we use an asterisk (*) before the name of the variable which holds the value for all non-keyword variable-length arguments.

Syntax

>>>def functionname([formal parameter] *var_args_tuple)

"function doc_string"

function_body

return [expression]

Example:

#function definition

```
>>>def print_info(arg1, *vartuple):
                    print (“Result is”)
                    print (arg1)
                    for var in vartuple:
                              print (var)
                    return
```

#function call

```
>>>print_info(10)
```

Output

10

```
>>>print_info(90,40,50)
```

Output

90

40

50

Python program to exchange two values

```
# Python swap program.
x = input('Enter value of x: ')
y = input('Enter value of y: ')
# 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))
```

OUTPUT

Enter value of x: 2

Enter value of y: 3

The value of x after swapping: 3

The value of y after swapping: 2

Python program to circulating n variables

```
# Circulate the values of n variables
n = int(input("Enter number of values : "))
list1 = []
for val in range(0,n,1):
    ele = int(input("Enter integer : "))
    list1.append(ele)
print("Circulating the elements of list ", list1)
for val in range(0,n,1):
    ele = list1.pop(0)
    list1.append(ele)
    print(list1)
```

OUTPUT

```
Enter number of values : 6
Enter integer : 2
Enter integer : 3
Enter integer : 4
Enter integer : 5
Enter integer : 6
Enter integer : 7
Circulating the elements of list  [2, 3, 4, 5, 6, 7]
[3, 4, 5, 6, 7, 2]
[4, 5, 6, 7, 2, 3]
[5, 6, 7, 2, 3, 4]
[6, 7, 2, 3, 4, 5]
[7, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]
```

Python program to find the distance between two points

```
import math
def calculatedistance(x1,y1,x2,y2):
  dist = math.sqrt((x2-x1)**2+(y2-y1)**2)
  return dist

# main program
x1=int(input("Enter first point x1 value:"))
x2=int(input("Enter second point x2 value:"))
y1=int(input("enter first point y1:"))
y2=int(input("Enter second point y2 value:"))

result= calculatedistance(x1,y1,x2,y2)
print ("Distance between two point =", result)
```

OUTPUT

Enter first point x1 value:20

Enter second point x2 value:30

enter first point y1:10

Enter second point y2 value:20

Distance between two point = 14.142135623730951

References

1. Allen B Downey. “Think Python: How to Think like a Computer Scientist”, Green Tea Press,version 2.0.17.
2. E. Balagurusamy , “Introduction to Computing and Problem Solving Using Python”,Mc Graw hill education.
3. Guido van Rossum and Fred L. Drake Jr, “An Introduction to Python – Revised and updated for Python 3.2”, Network Theory Ltd., 2011.
4. John V Guttag, “Introduction to Computation and Programming Using Python”, Revised and expanded Edition, MIT Press , 2013
5. www.w3school.com
6. https://www.programiz.com/python-programming.
7. https://www.tutorialspoint.com