A function contains a group of statements and performs a specific task.

U-3 FUNCTION STRINGS PYTHON DATA STRUCTURE

III	**Function:** Parts of A Function , Execution of A Function , Keyword and Default Arguments ,Scope Rules. **Strings :** Length of the string and perform Concatenation and Repeat operations in it. Indexing and Slicing of Strings. **Python Data Structure :** Tuples , Unpacking Sequences , Lists , Mutable Sequences , List Comprehension , Sets , Dictionaries **Higher Order Functions:** Treat functions as first class Objects , Lambda Expressions

FUNCTION: Parts of a Function*

```
a = int(input("Enter first number"))
b = int(input("Enter second number"))
c = a+b
print("sum is", c)
```

Someprogram.py

Now inside my someprogram.py if want to repeat this 4 line code 3 times, then I have to write these 4 line 3 times.

Here is where function concept comes in.

We will give these 4 line code a name and place it inside a block.

We define a function using def keyword.

```
def sum():   <- colon
    a = int(input("Enter first number"))
    b = int(input("Enter second number"))
    c = a+b
    print("sum is", c)
```

indented Function body

Now whenever I want to use these 4 line of code I will write only the name of function eg. sum and write ( ) after it. parenthesis

i.e. sum().

As no parameters* are defined inside function, when we simply write sum() it means we called function sum, due to which 4 line of code will be executed.

Execution of a function

Parts of a Function*

1. Function definition
2. Function body
3. Function Call
4. return (optional)

So my final program eg. someprogram.py looks like this:

someprogram.py

```
sum()
sum()
sum()
```

*a parameter is a variable that receives data from outside into a function.

abhi.c

FOUR ways to define a function

① Takes Nothing, Returns Nothing

```
def add():
    print("Enter two numbers")
    a = int(input())
    b = int(input())
    S = a+b
    print("sum is", S)
add()
```

② Takes Something, Returns Nothing

```
def add(a, b):
    S = a+b
    print("Sum is", S)
add(10,20)
```

parameter

Values 10 and 20 will be copied into parameters a and b respectively when statement add(10,20) will be executed.

③ Takes Nothing, Returns Something

```
def add(a, b):
    S = a+b
    return S
x = add(10,20)
print("sum is", x)
```

④ Takes Something, Returns Something

```
def add():
    print("Enter two numbers")
    a = int(input())
    b = int(input())
    S = a+b
    return S
x = add()
print("Sum is", x)
```

Key word Arguments: can be passed out of order. Python interpreter uses keywords to match the values passed with the parameters used in the function definition.

```
def display(i, a, S):
    print(i, a, S)
display(2, 3, 'abc')      #ok
display(a=3, i=2, S='abc')   #ok
display(s='abc', a=3, i=2)   #ok
display(s='abc', i=2, a=3)   #ok
```

Default arguments: assume a default value, if we do not pass the value for that argument during the call.

```
def fun(a, b=100, c=3.14):
    return a+b+c
x = fun(10)   # passes 10 to a, b is taken as 100, c as 3.14
y = fun(20,50) # passes 20, 50 to a, b. c is taken as 3.14
z = fun(30, 70, 6.27) # passes 30, 70, 6.27 to a, b, c
```

Scope Rules: When we declare a variable inside a function, it becomes a local variable.
A local variable is a variable whose scope is limited only to that function where it is created.
That mean the local variable is available only in that function and not outside of that function.

```
def myFunction():
    a = 1          # this is local variable
    a = a+1        # increment it
    print(a)       # displays 2
myFunction()
print(a)  # error, not available as a is local variable.
```

when a variable is declared above a function, it becomes global variable. Such variables are available to all the functions written after it.

```
a = 1    # this is global variable
def myFunction():
    b = 2    # local var
    print(a)
    print(b)
myFunction()
print(a)  # available as a is global variable
print(b)  # error, not available
```

Strings: A string represents a group of characters. In Python, the str datatype represents a string.
Length of the string: len() is a built-in function that returns the number of characters present in string.

```
s = 'Python'
n = len(s)
print(n)  # 6
```

length of string

Concatenation: operator is denoted by '+'. It joins or concatenates the strings.

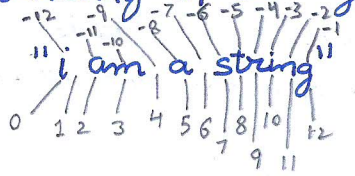
```
s1 = 'Python'
s2 = 'Programming'
s3 = s1+s2
print(s3)  # PythonProgramming
```

Repetition operator: denoted by '*'

```
s = 'Python'
print(s*2)  # PythonPython
```

abhi.c

Indexing of strings: is used to obtain individual characters.



Strings can be enclosed in single quotes '...' or double quotes "..."

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one.

← Right to Left (-ve index) (counting starts with -1 index)

→ Left to Right (+ve index) (counting starts with 0 index)

print(a[0]) gives value i

print(a[-1]) gives value g (last character)

print(a[-12]) also gives value i

print(a[12]) also gives value g (last character)

Slicing of strings: allows you to obtain substring.

means piece of string

format of slicing is

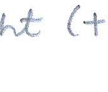
stringname [begin : end : step]

all 3 values are optional (but : is not optional)

+ L to R

− R to L

end value is always excluded in the result




```
>>> word = 'Python'
>>> word[-1:]      # last character
'n'
>>> word[-2:]      # characters from the second-last (included) to the end.
'on'
```

character from last to the end

```
>>>word[:2]       # character from the beginning to position 2 (excluded)
'Py'
>>>word[4:]       # characters from position 4 (included) to the end
'on'
>>> word[1:3]
'yt'
>>> word[-1:-4:-2]
'nh'
>>> word[-1:4:-2]
'n'
```

whenever we travel (R to L), we have to mention step as negative, even when the step is -1

```
>>> word[1:4]==word[1:4:1]      # 'yth'
True
>>> word[:]==word[::]           # 'Python'
True
```

this means that by default step size is 1

Reversing a string

```
>>> word[::-1]
'nohtyP'
>>> word[len(word)::-1]
'nohtyP'
```

Python Data Structure :

Tuples: A tuple consist of a number of values seperated by commas.

eg. `>>> t = 123, 'abc', 690`
`>>> t`
`(123, 'abc', 690)`
`>>> t[0]`
`123`
`>>> u = t, (1, 2, 3, 4)` — nested tuple
`>>> u`
`((123, 'abc', 690), (1, 2, 3, 4))`

On output, tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; tuples may be input with or without surrounding parentheses

Tuples are immutable:

`>>> t[0] = 9`
TypeError

A tuple can contain mutable objects, such as list.

eg. `>>> a = ([4, 5], 7)`
`>>> type(a)`
`<class 'tuple'>`

Tuple functions : ① len(t) Returns number of elements in the tuple
eg. here number of elements is 3.

Tuple methods : at.count(x) Returns how many times the element x is found in at

eg. `>>> at = 1, 2, 3, 2, 2, 2`
`>>> at.count(2)`
`4`

`>>> at.index(2)` Returns the first occurence of the element
`1` eg. 2 is the tuple . ie 2 occurs at index 1.

Sorted function a = (2, 54, 7, 9, 1)

`>>> sorted(a)` Sort the elements of the tuple into ascending order list
`[1, 2, 7, 9, 54]`

`>>> sorted(a, reverse=True)` sort in reverse order.
`[54, 9, 7, 2, 1]`

`>>> a`
`(2, 54, 7, 9, 1)`

a remains as it is, because a is immutable.

abhi.c

Unpacking Sequences :

Suppose a function is expecting positional arguments also called required arguments. Positional argument must be passed in correct positional order. and the arguments to be passed are in a list or tuple. In this case we need to unpack the list or tuple using the * operator before passing it to the function.

```
def printIt(a,b,c,d):
    print(a,b,c,d)
lst= [10, 20, 30, 50]
tpl= ('A', 'B', 'C', 'E')
print(*lst)
print(*tpl)
```

↓

```
10 20 30 50
A B C E
```

↳ ie if a function expects an int, float, string to be passed to it then:

```
fun(10, 3.14159, 'absolute')   # correct call
fun( 3.14, 10, 'absolute')     # incorrect
```

In positional arguments (required arguements), number of arguments must match with the number of arguments received.

Tuple (packing unpacking)

```
a=1
b=2
c=3
t= (a,b,c)
```

packing ie we packed these three variables and made a single thing called tuple t.

---

eg. Unpacking

```
t= (10, 20, 30)
a,b,c = t
```

```
>>>a
10
>>>b
20
>>>c
30
```

ie I want to unpack three values from t and assign to a, b, c (UNPACKING)

Lists : are comma-seperated values (items) between square brackets.
Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25, 36]
>>> squares
[1, 4, 9, 16, 25, 36]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced.

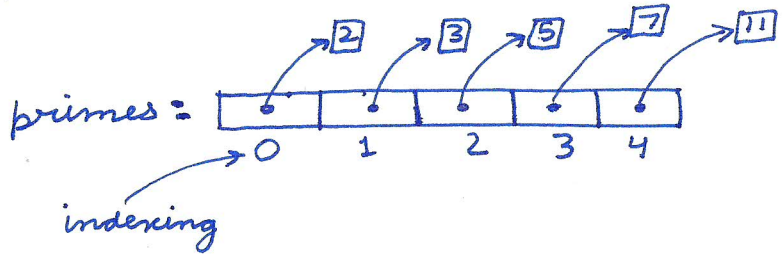
```
>>> squares[0]      # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]    # slicing returns a new list
[9, 16, 25]
>>> squares + [49, 64, 81, 100]    #Concatenation operation
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

strings are immutable .but <u>lists are mutable</u> (ie contents in list can be edited after its creation)

```
>>> cubes = [1, 8, 27, 65, 125]
>>> cubes = [1, 8, 27, 4**2, 125]
cubes
[1, 8, 27, 64, 125]
```

Diagram of list

primes = [2, 3, 5, 7, 11]



important methods in lists:

insert() append() extend()
remove() pop() sort()
clear() reverse() index

Mutable sequences : Lists are mutable sequences.

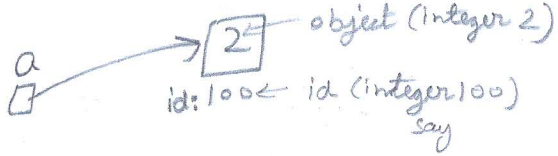
The types that can be indexed is known as sequence types (eg. list, tuple, range, string)

Mutable sequence can change their content after creation.

i.e. mutable objects can change their value but keep their id() *

id(object) return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime.

We can think of id as the address of the object in memory




```
>>> l = [2]
>>> id(l)
1274001
>>> l.append(3)
>>> l
[2, 3]
>>> id(l)
1274001
```

that's why list is a mutable object *

Mutable objects have (eg list, set, dict)

accessors
these method
(dont change the state of object)

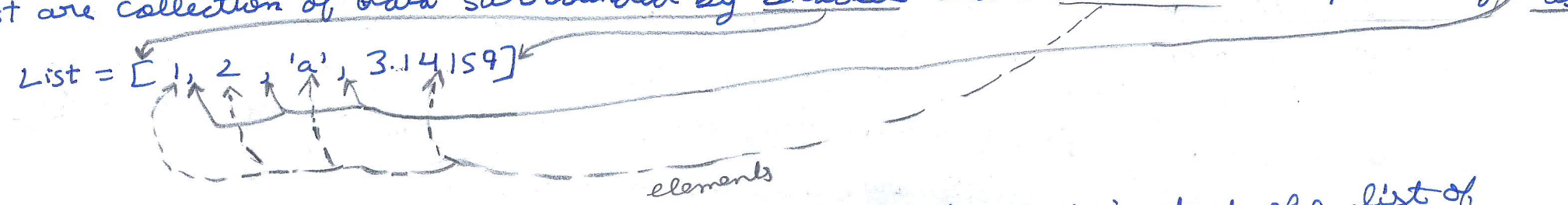
mutators
these method
(also called update methods)
(changes the state of object)
eg. list class has sort method

Immutable objects have (eg. tuple, str, int)

accessor
these method
(dont change the state of object)

List Comprehension: offers an easy way of creating lists.

List are collection of data surrounded by brackets and the elements are seperated by comma



List comprehension is also surrounded by brackets, but instead of a list of data inside you enter an expression followed by for loops and if clauses.

[expr for val in collection]

generates the elements in the list

```
>>> v = [1, 2, 3]
>>> w = [4*x for x in v]
>>> w
[4, 8, 12]
```

we can do scalar multiplication by list comprehension

Scalar multiplication (ie each element got multiplied with 4)

[expr for val in collection if <test>]

[expr for val in collection if <test1> and <test2>]

```
>>> squares = [i**2 for i in range(1, 6)]
>>> a
[0, 1, 4, 9, 16]

>>> words = ["Available", "Botany", "Absolute", "Ginger"]
>>> awords = [t for t in words if t.startswith("A")]
>>> awords
["Available", "Absolute"]
```

abhi

Sets: A set is an unordered collection with no duplicate elements.

Basic uses include membership testing and eliminating duplicate entries. Set objects support mathematical operations like union, intersection, difference, symmetric difference. Curly braces {} or the set() function can be used to create sets. To create an empty set you have to use set(), not {} because {} creates an empty dictionary (a data structure).

```
>>> basket = {'apple', 'orange', 'apple', 'banana'}
>>> basket
{'orange', 'banana', 'apple'}
>>> 'orange' in basket
True
```

```
>>> f = set()
>>> type(f)
<class 'set'>
```

```
>>> set1 = {1, 5, 4, 3, 6, 7, 10}
>>> set2 = {10, 3, 7, 12, 15}
>>> set1 | set2
{1, 3, 4, 5, 6, 7, 10, 12, 15}
>>>
```

```
>>> set1 & set2
{10, 3, 7}
```

```
>>> set1 - set2
{1, 4, 5, 6}
```

```
>>> set1 ^ set2
{1, 4, 5, 6, 12, 15}
```

abhi.c

Dictionaries : Unlike sequences which are indexed by a range of numbers, dictionaries are indexed by keys. (keys should be of immutable type). Strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples. If a tuple contains any mutable object (eg. list, dict etc) either directly or indirectly, it cannot be used as a key.

```
>>> a = { 'A' : 2, 'B' : 1, 'C' : 3 }
>>> a ['B']
1
>>> 'c' in a
True
>>> 'D' not in a
True
```

dict() constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict ( [ ('sape', 413), ('gui', 412), ('jac', 40)])
{'sape': 413, 'gui': 412, 'jac': 40}
```

quote symbol

To avoid writing quote symbol when key is a simple string so to ease this process we write like this

```
>>> dict ( sape = 413, gui = 412, jac = 40)
{'sape': 413, 'gui': 412, 'jac': 40}
```

Note here we are using concept of keyword arguments.

```
>>> sales = {'apple': 2, 'orange': 3, 'grapes': 4}
>>> sales.items()
dict_items ([('apple', 2), ('orange', 3), ('grapes', 4)])
>>> sales.keys()
dict_keys(['apple', 'orange', 'grapes'])
>>> sales.values()
dict_values ([2, 3, 4])
```

Higher Order Functions: Treat functions as first class objects.

In Python, functions are considered as first class objects, which means that we can use functions as perfect objects. When we create a function, the Python interpreter internally creates an object. (which supports the fact that in Python everything is an object!)

Since functions are objects, which means we can pass a function to another function just like we pass an object (or value) to a function. Also, it is possible to return a function from another function. (This is similar to returning an object (or value) from a function.

(↑ like 2 is object (of int class), like 3.6 is object (of float class))

Therefore,
- It is possible to assign a function to a variable. (call ← this will refer to the value returned from the function definition.)
- It is possible to define one function inside another function
- It is possible to pass a function as parameter to another function
- It is possible that a function can return another function.

```
def show(s):
    return 'Hello' +s

x= show('abc')    # assign function to a variable x
print(x)
```

o/p: Hello S

```
def d():
    def message():
        return 'How r you'
    return message

fun= d()
print(fun())
```

o/p: How r you

This will return message() function out of d() function. The returned message() function can be referenced with a new name 'fun'

```
def d(fun):
    return 'Hi ' + fun()
def message():
    return 'how r you'
print(d(message))
```

we pass message() function a parameter to d() function. ie parameter fun will now refer to d() function.

```
def display(str):
    def message():
        return 'hello'
    result = message() + str
    return result
print(display('abc'))
```

o/p: Hello abc

abhi.c

Lambda Expressions: allow us to define a function anonymously.
It is an expression, not a statement.
Lambda Expression have implicit return statement
(you dont write return keword, because returning is automatic.)

Format:

lambda argument1, argument2, ... : expression

(: colon represents the beginning of function that contains expression x * x.)

lambda — Keyword lambda

This represents that an anonymous function (a function without a name) is created.

eg:
```
f= lambda x: x**
v= f(9)
```
v refers to the square of 9 ie 81

```
# Sum of two numbers
f= lambda x,y: x+y
result= f(6.3, 9)
print('sum= ', result)
```

o/p Sum = 15.3

abhi.c