Agenda

22 January 2025 07:11

In Built Functions
In Built modules
Intro to Functions

In Built Functions

22 January 2025 07:23

→ 2 types of functions

(i) in-built

(ii) user defined

(i) in-built ✓

→ Those fns already created by python developers

Ex:- len()
min()
max()
id()
type()
print()
input()

(ii) user-defined

will see in other slide (intro to functions)

module

22 January 2025 07:23

These are python files which will have functions, classes, variables.

Those are divided into 2 types

(i) In Built

(ii) user-defined module

(i) In-Built module :-

-> Those are python files developed by python developers So, that we can re-use them based on our requirement

Ex:- math
random
os
datetime

math.py (module)

filename Extension

class
fun
variables

x = 10

l = [1, 2, 3, 4]

user-validation.py

class
functions
variables

import user-validation

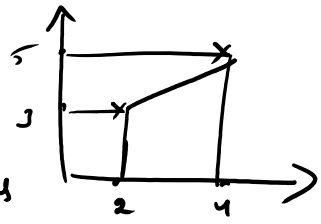
user-validation.

Intro to Functions(user-defined)

22 January 2025 07:23

$P_1 = (2,3)$, $P_2 = (4,5)$

dist(P_1, P_2)



def dist(P1, P2): — name; parameters (or inputs); not mandatory

Keyword """ """ → doc-string

Numbers, logic

return

not mandatory: name

input values: $P_1 = (2,3)$ (x_1, y_1) $P_2 = (4,5)$ (x_2, y_2)

$\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$

```
def dist(P1,P2):
    d = math.sqrt((P2[0] - P1[0])**2 + (P2[1] - P1[1])**2)
    return d
```

} function definition

```
d = dist((2,3),(4,5))
print(d)
```

} → function calling

(2,3) → P1, (4,5) → P2

More on Functions

23 January 2025 07:33

Recap
Parameters vs Arguments
Types of Arguments
*args, **kwargs
Order of defining and passing the params/arguments

Recap:-

2 types of functions
(i) In-built (python developers)
(ii) user-defined (users has to create his own functions)

2 types of modules
(i) in-built (python developers)
(ii) user-defined (users can create their own modules)

Advantages:-
1) reusability

Ex:- distance user-defined

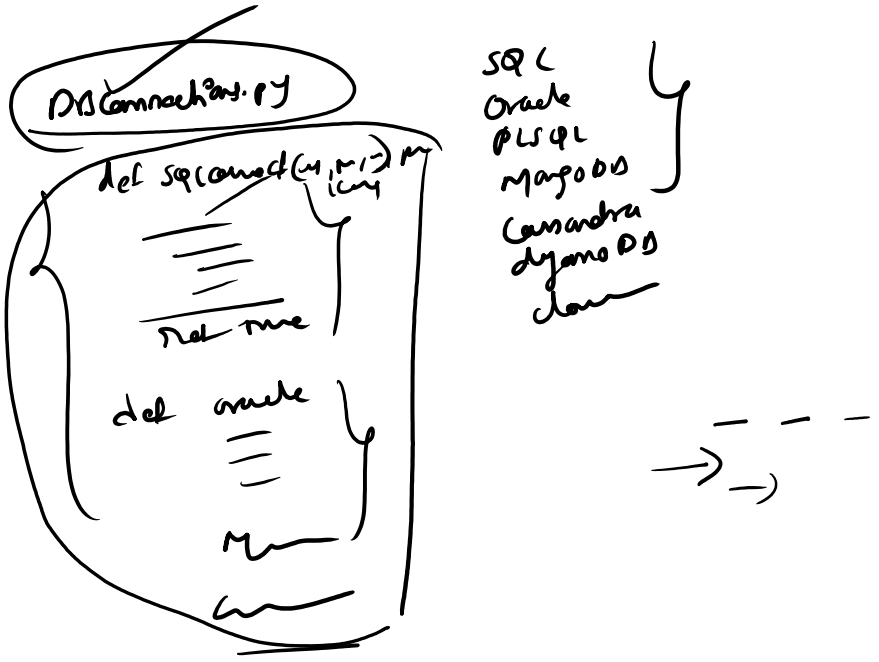
```
def dist(p1, p2):
    ...
    return output
x = dist((2,3),(3,4))
print(x)
```

geometry.py

```
import geometry
geometry.
```

```
def angle(p1, p2):
    ...
    return output
x = angle((2,3),(3,4))
print(x)
def hamdg_dist(p1,p2):
    ...
    return output
x = hamdg_dist((2,3),(3,4))
print(x)
```

Angle b/w perpendiculars = 90°



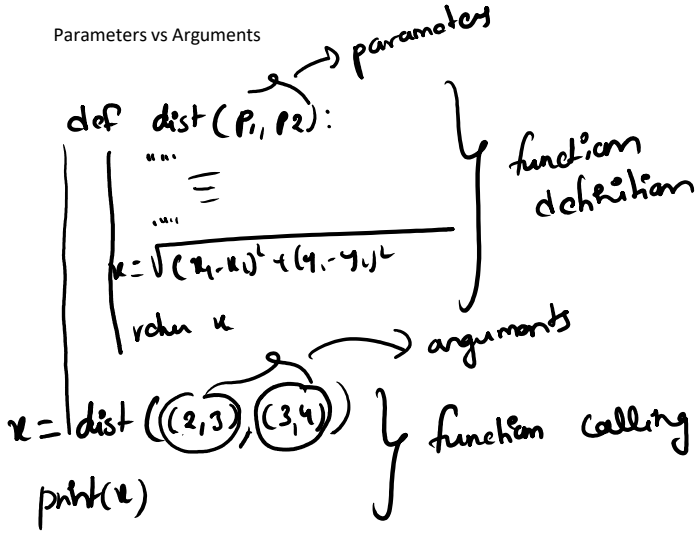
Syntax of a function :-

→ def <fn_name>( ): (# of parameters)

"""
"""
Body/logic ✓

return < > ✗

Parameters vs Arguments



Types of Arguments

(i) positional Arguments

(ii) Default Arguments

(i) positional Arguments

(ii) Default Arguments

(iii) Keyword Arguments

*args, **kwargs

*args :- it handles variable length of ↑ non keyword arguments

Note :- *args | *Rashes | *si

**kwargs :- it takes | handles variable length of key-word arguments

Note :- **kwargs | **rashes | **si

Order of defining and passing the params/arguments

-> positional arguments | params follows key word agumets
-> non-default follows default agumets

a=5, b, c *args **kwargs ✗

b, c, a=5, *args **kwargs

Advanced Concepts

24 January 2025 07:20

Recap
Variables Scope ✓
Nested Functions ✓
Functions are first class citizens ✓
HOF(Higher Order Functions) ✓
Recursive Functions ✓

Recap :-

1) params & Arguments ✓ ✓

2) Types of Arguments
   (i) positional arg ✓ order
   (ii) Default arg ✓
   (iii) Key word arg ✓ without order / position

3) *args & **kwargs ✓ ✓
   Var non-key Var key-word

4) order of arguments
   (a, b=5, *args, **kwargs)

Variables Scope

24 January 2025 07:24

How Functions are executed in memory ✓
Global and Local variables
Accessing Global from Local
Modifying Global from Local
global keyword

How Functions are executed in memory

-> for every fn a local space gets created

-> function memory/local space will be available until function Execution is completed. Once it is completed the allocated memory/local space for that function gets destroyed.

Note :- Global & local space

Global and Local variables

-> variables which are present in global space called global variables

-> Variables " " " " local " " local variables

-> from local space we can access global variables

Modifying Global from Local

-> Cannot modify the global variable from function local space

global keyword

```
def func1(z):
    print(x+1)

x=5
func1(x)
print(x)
```

global space
func1
x = 5

o/p
6
5

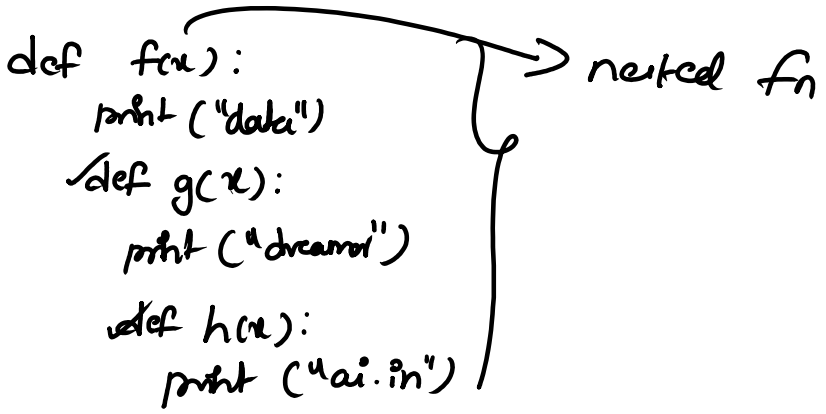
Def :- using global keyword we can modify the global variables from local space

Nested Functions

24 January 2025 07:25

→ A function in which is having other functions are called nested fns

Ex:-



Functions are 1st class citizens

24 January 2025 07:25

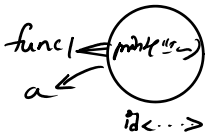
type
Id
Aliasing/re-assigning
storing
Deleting
Returning a function
Functions passing as an input arguments

Benefits of functions
. Code modularity
Each function performs a specific task, making the code easier to understand and maintain
. Code re-usability
Functions allow you to write code once and reuse it multiple times by simply calling the function.
. Improved Debugging
Debugging becomes simpler since errors can be isolated to specific functions without affecting the rest of the program.

Aliasing | re-assigning

```
def func1():
    print("I am inside a func1")

a = func1
a()
id(a) == id(func1)
```

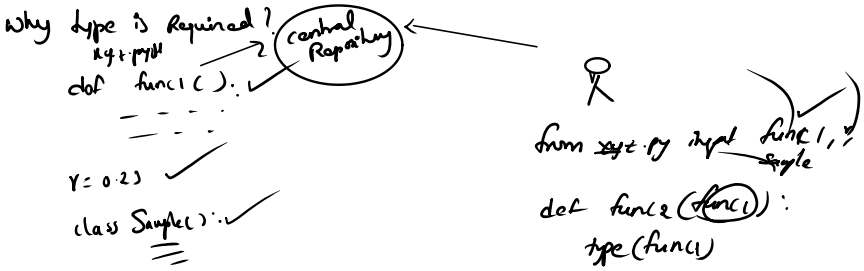


Why the name "Functions are called 1st class citizens"?

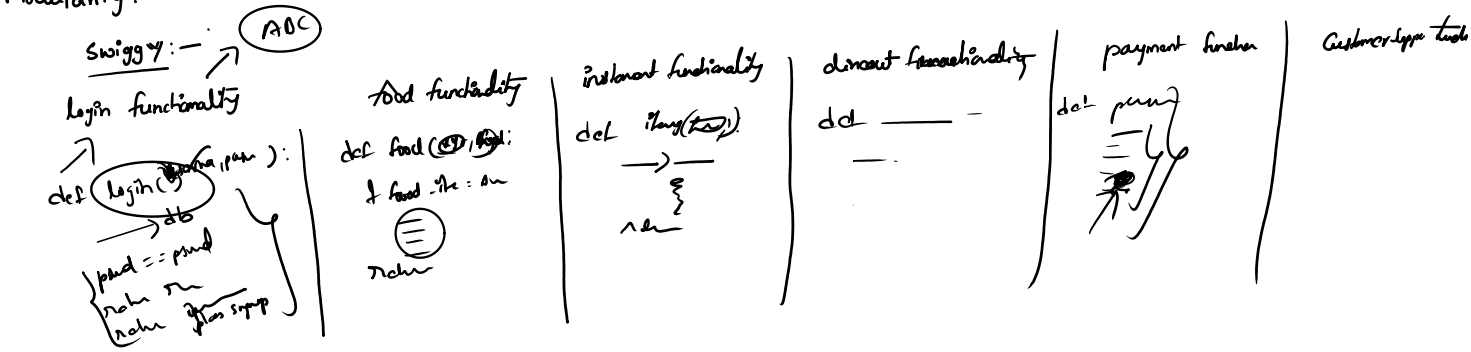
list
-> type() ✓
-> id() ✓
-> storing ✓
-> delete ✓

∴ functions are nothing but a datatype in python

Why type is Required?



Modularity :-

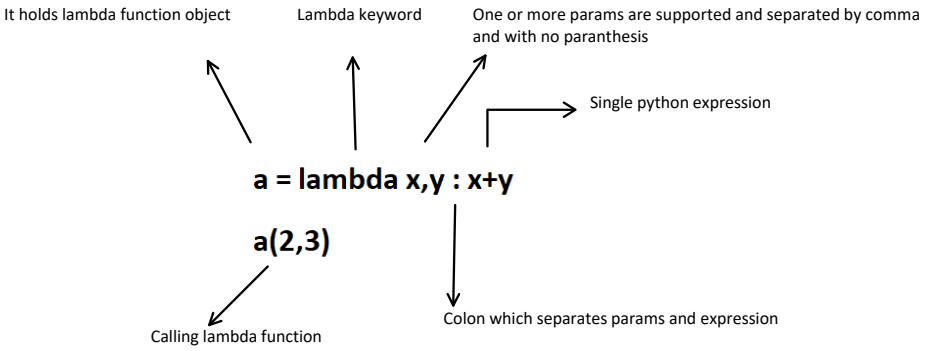


Lambda & HOF

24 January 2025 07:27

Lambda
HOF(Higher Order Functions):
- Map
- Reduce
- Filter

Lambda:



Difference b/w lambda and normal function
- . No name(i.e anonymous)
- . Lambda will not return anything (but it returns function itself)
- . Lambda is written in single line
- . Not reusable

Note: Then why use lambda function
. It is used with HOF

Note: If lambda is assigned to a reference variable then it can be re-used or can be called in the same file or in other file as well.

xyz.py

a = lambda x : x*2

abc.py

from xyz import x

a(4)

#16

HOF(Higher Order Functions):

A **higher-order function** is a function that meets at least one of the following conditions:
1. Any function that takes **another function as an argument.**
2. Any function that **returns a function.**

Python offers inbuilt HOF functions:

map(function, iterable): Applies a function to all elements in an iterable.

filter(function, iterable): Filters elements based on a condition defined by a function.

from functools import reduce
reduce(function, iterable) : Reduces an iterable to a single value by applying a function.

1. Any function that takes **another function as an argument.**

```
def func2():
    ≡
def func1(z):        # HOF
    ≡
print(func1(func2))
```

z = func2

2. Any function that **returns a function.**

```
def func2():
    ≡
def func1():         # HOF
    ≡
    return func2
```

z = func1

z() => func2()

Q) finding the squares from a list

① a = map(lambda x: x**2, [1,2,3,4])

returns map object

```
for i in a:
    print(i)
```

1
4
9
16

```
for i in a:
    lambda(i)
```

x = 1 => 1
x = 2 => 4
x = 3 => 9
x = 4 => 16

② print(list(map(lambda x: x**2, [1,2,3,4])))

[1,4,9,16]

---

filter

a = filter(lambda x: x>2, [1,2,3,4])

```
for i in a:
    print(i)
```

[False, False, True, True]

[3,4]

```
def lambda
    for x in [1,2,3,4]:
        if x>2:
            return True
        Else
            return False
```

---

reduce:-

```
a = reduce(lambda x, y : x+y, [1,2,3,4])
print(a)
```

x y
3 3
x y

x = 1, y = 2 → x+y = 3

x = 3, y = 3 → x+y = 6

x = 6, y = 4 → x+y = 10

Problem:

```
logs = [
    "2025-01-27 10:15:30 0.2 ERROR 500 Internal Server Error",
    "2025-01-27 10:16:10 0.3 ERROR 404 Not Found",
    "2025-01-27 10:17:45 0.5 ERROR 503 Service Unavailable",
    "2025-01-27 10:18:00 0.7 INFO Service Started"
]
```

o/p [{"timestamp": "2025-01-27 10:15:30", "duration": 0.2, "Error_code": 500, "Error_name": "internal server Error"},
{"timestamp": "2025-01-27 10:16:10", "duration": 0.3,
{ },
{ },
]

2025-01-2210:15:30
2025-01-27 10:15:30

x.split(" ")[5:] = ["internal", "Server", "Error"] => "internal Server Error"

" ".join(x.split(" ")[5:])

sk = " "
x = "internal"
x = "internal server"
x = "internal server Error"

x = "internal_Server_Error"

Recursive Functions(Diving Deep)

24 January 2025 07:28

A **recursive function** is a function that calls itself in its definition. It is a useful approach for solving problems that can be broken down into smaller, similar sub-problems. Each recursive call reduces the size of the problem until it reaches a base case, which stops the recursion.

How Does a Recursive Function Work?

A recursive function has two main parts:

1. **Base Case**: This is the stopping point. It tells the function when to stop calling itself to prevent infinite loop.
2. **Recursive Case**: This is where the function breaks the problem into smaller and similar pieces and calls itself with those smaller pieces.

Recap

Functions
→ 2 types — in-built, user def

Modules
→ 2 types — in built, user def

Intro to user-def:- Syntax

More on functions
params → (posn, key, def)
*args, **kwargs

Advanced:-
global space
func local space
global keyword

Nested functions
functions are called 1st class citizen
type, id, ( ), del,
pass fn as arg,
return fn

Lambda & Hof (map, reduce, filter)

Ex:- factorial of number:-

```
def fact(n):
    result = 1
    for i in range(1,n):
        result = result * i
    return result
```

} definition

```
0 = fact(5)
print(0) # 120
```

Ex:- factorial of number using Recursion

Base Case :-
Recursive Case:-

Recursive Cases

5! = 5 x 4!
= 5 x 4 x 3!
= 5 x 4 x 3 x 2!
= 5 x 4 x 3 x 2 x 1!

Base Case

= 5 × 4 × 3 × 2 × 1 !

=

Code :-

```
def fact(n):                          # Recursive fn
    if n == 1:                        # -> base case
        return 1                      # fn definition
    return n * fact(n-1)              # Recursive case
o = fact(5)                           # fn calling
print(o)
# 120
```

n = 5 4 3 2 1

5 × fact(4)

5 × 4 × fact(3)

5 × 4 × 3 × fact(2)

5 × 4 × 3 × 2 × fact(1)

return 5 × 4 × 3 × 2 × 1

Ex: num = 12345

Sum of digits = 15

Recursive Case

10) 12345 (1234
  12340

      5

10) 1234 (123
  1230

     4

10) 123 (12
  120

    3

10) 12 (1
  10
  --
   2

10) 1 (0
  0
  -
  1

5 + 4 + 3 + 2 + 1

Base Case

if quotient == 0 :
    return 0

∴ n % 10 = Remainder

n // 10 = Quotient

```
def sum_of_digits(n):                 # Recursive fn
    if n == 0:                        # -> base cond
        return 0
    return (n % 10) + sum_of_digits(n // 10)   # recursive case
o = sum_of_digits(12345)              # fn calling
print(o)
# 15
```

n = 12345 1234

return 5 + sum_of_dig(1234)

return 5 + 4 + sum_of_dig(123)

return 5 + 4 + 3 + sum_of_dig(12)

return 5 + 4 + 3 + 2 + sum_of_dig(1)

return 5 + 4 + 3 + 2 + 1 + sum_of_dig(0)

return (5 + 4 + 3 + 2 + 1 + 0)

return 15

Ex: 3 fibbonacci of a number

```
def fibb(n):                          # Recursive fn
    if n == 0:                        # base cases
        return 0
    if n == 1:
        return 1
    return fibb(n-1) + fibb(n-2)      # recursive case
```

n = 3 2 1 0

return fibb(2) + fibb(1)

return fibb(2) + 1

return fibb(1) + fibb(0) + 1

return fibb(1) + 0 + 1

return 1 + 0 + 1

n = 5

ans = fibb(4) + fibb(3)

= fibb(4) + fibb(2) + fibb(1)

= fibb(4) + fibb(2) + 1

= fibb(4) + fibb(1) + fibb(0) + 1

= fibb(4) + 1 + 0 + 1

return fibb(n-1) + fibb(n-2) } recursive case

-> 0 = fibb(3) } fn calls
main(0)
# 2

return ← 1+0+1

return 2

= fibb(4) + 1 + 0 + 1

= fibb(3) + fibb(2) + 1 + 0 + 1
2 ← 1 ←
= fibb(2) + fibb(1) + 1 + 0 + 1

= fibb(2) + 1 + 0 + 1 + 0 + 1
←

= fibb(2) + fibb(1) + 1 + 0 + 1 + 0 + 1

= fibb(1) + 1 + 1 + 0 + 1 + 0 + 1

= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1

= 5

fibb(0) = 0

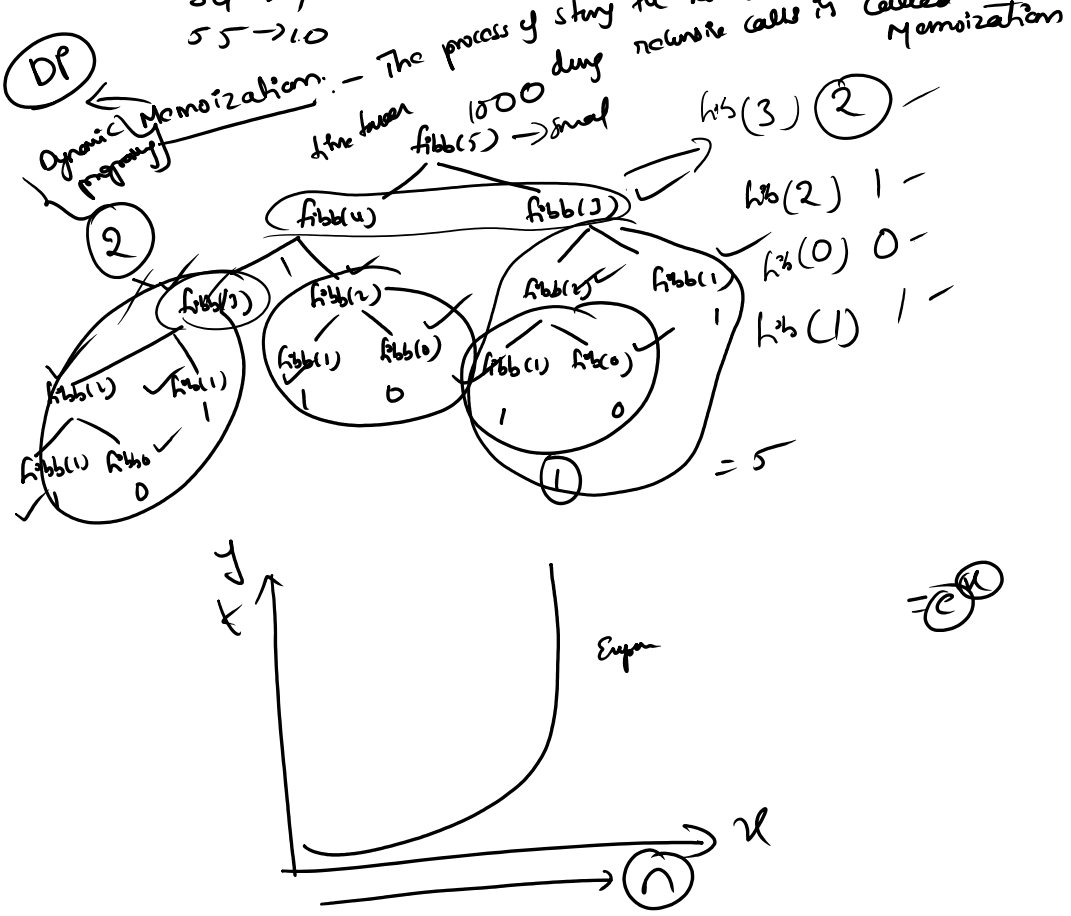
fibb(1) = 1

fibb(2) = 1

fibb(3) = 2

fibb(10) = 55

{ 1 —
1 →
2 → 3
3 → 4
5 → 5
8 → 6
13 → 7
21 → 8
34 → 9
55 → 10



Memoization - The process of storing intermediate results while performing recursion

Disadvantages of recursion:
- It makes resource intensive - It repetitively calculates the same operation
- As input n increases the time taken is exponential

We can solve above problems using memoization.