⇒ Agenda → continue the discussion of funcⁿ,
passes by value & references

Recursion

functions

def fun(a, b): → $a = a*2$, $b = b-1$

do something → we do some changes on the parameters

return something

x = 10

y = 20

fun(___ , ___)

Q:- Given 2 numbers in ; write a program to swap them.

$\boxed{\begin{matrix} a = 10 \\ b = 20 \end{matrix}}$ & $\boxed{\text{Swap the}}$ $\longrightarrow$ $\begin{matrix} a = 20 \\ b = 10 \end{matrix}$

$\boxed{\begin{matrix} temp = a \\ a = b \\ b = temp \end{matrix}}$

temp ← 10
a ← 20

def swap(x, y):

swap(a, b)

→ Does this work for lists also ??

→ When we pass a list, inside a func^n, then it is passed by reference (means actual original list is passed) So whatever change you will do persists.

But in case of int, str, bool, float ..... they a passed by value (means a copy of them is passed) So whatever changes you will do, they won't persist outside the func^n local copy of a is passed

```
def fun(a1):
    a1 = 20

a = 5
print(a)
fun(a)
print(a)
```

diff run

a = 5 → global(a) → int

print(a) → 5

diff memory loc

```
def gun(p):
    p.append(20)

li = [1, 2, 3, 4]
print(li)        → [1, 2, 3, 4]
gun(li)
print(li)        → [1, 2, 3, 4, 20]
```

their args are

[1, 2, 3, 4, 20] → in memory

lists are passed by reference

Q→

```
def update(x):
    x += "1"

y = "10"
print(y)
update(y)
print(y)
```

pass by value → int, str, bool, float ....

pass by reference → lists

Recursion

$y = f(x)$ — y: o/p, x: input

$y' = g(x)$ — y': o/p, x: i/p

$f(g(x))$ → composite func^n → calling another func^n inside a fun

$y = f(x)$ $\qquad x = f(x')$

$f(f(x'))$ $\qquad x' = f(x'')$

$\downarrow$

$f(f(f(x'')))$

$\vdots$

special composite funcⁿ which calls itself inside it only but with a different parameters.

→ Mathematically, Recursion is the process when we have special composite func^n, where the composition is made such that func^n calls itself inside it only with or without the same parameter.

when it will be infinite?

$x = f(x)$

$f(f(x))$ - - - - -

In programming recursion is defined as a tool where a function solving a bigger problem, calls itself consider it to solve another problem, till the time we reach to a stage where we have the smallest already solved problem with an extra memory buffer/space

→

Principal of mathematical induction

↳ Prove that sum of first n natural no. is $\frac{n \times (n+1)}{2}$

using PMI

↳ for $n=1$, we already know that sum is $\frac{1 \times (1+1)}{2} = 1$

So formula works. → Base Case → Smallest

subproblem for which we already know ans.

→ assume that it works for $n = K$ → assumption

→ proove that if it works for $n = K$, then it also works

for $n = K+1$ → self work   for any K

if → $K+1$
↳ first get K
if → (K)
get $(K-1)$
if $(K-1)$
↳ $K-2$
- - - - if (2)
↳ got (1)

$5! = 5 \times 4!$

factorial → Let's say $f(n)$ is a funcⁿ

which returns $n!$

$f(5)$

for $n=1$, we know, $f(1) = 1$ or $f(0) = 1$ Base Case

assume $f(n)$ works for $n=k$ $f(k)$ → correct value

cases

$n = k+1$

$f(k+1) = (k+1) \times f(k)$ → correct

$f(5) = 5 \times f(4)$

$\hookrightarrow 4 \times f(3)$

$\hookrightarrow 3 \times f(2)$

$\hookrightarrow 2 \times f(1)$

0th 1st 2nd 3rd 4th 5th 6th 7th 8th - - - - - -

→ fibonacci ?? 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

any i^{th} term is the sum of previous 2 terms

Let say $\boxed{f(n)}$ → is a func that calcs n^{th} fibonacci

→ Base Case → $f(0) = 0$, $f(1) = 1$

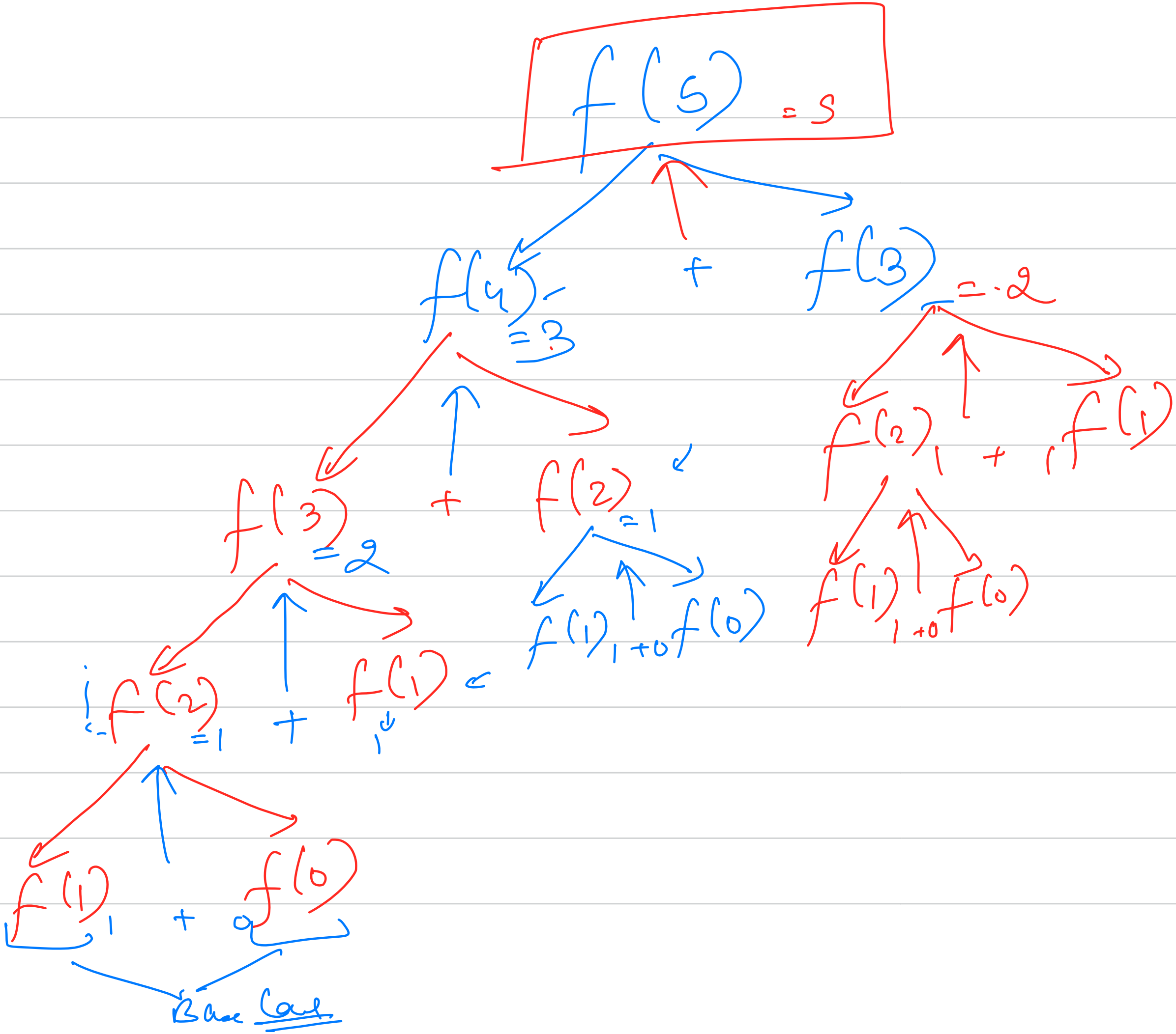
assume
→ for $n = K$ $f(K)$ correctly calcs K^{th} fibonacci
for $n = K-1$ $f(K-1)$ correctly calc $(K-1)^{th}$ fib] correct

→ self work proove for $n = K+1$

$$f(K+1) = f(K) + f(K-1)$$

80

$$f(n) = \underbrace{f(n-1)}_{\text{correctly}} + \underbrace{f(n-2)}_{\text{correctly}}$$

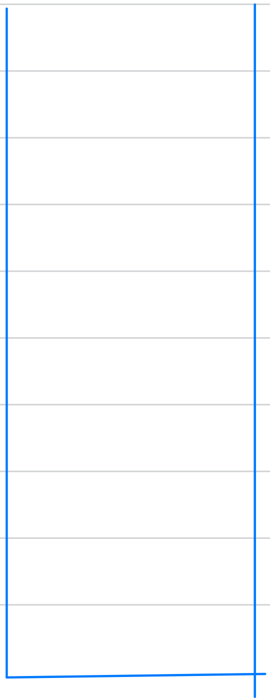


f(5) = 5
f(4) = 3
+
f(3) = 2
f(3) = 2
+
f(2) = 1
f(2) = 1
+
f(1)
f(2) = 1
+
f(1)
f(1) + f(0)
f(1) + f(0)
f(1) + f(0)
Base Case

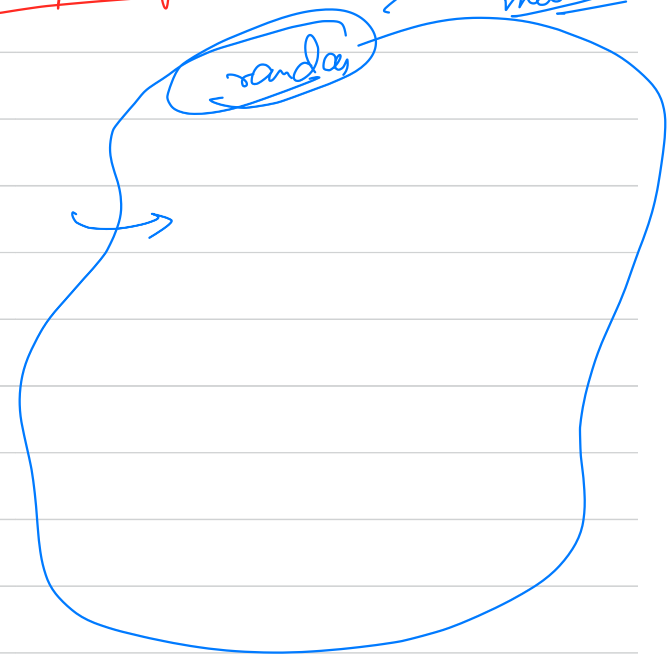
Memory has 2 parts (major)

There are some minor also

Call stack → linear memory storage



Heap (pool) — pool of mem



Call stack → whenever you call a function, then we add an entry to the top of stack. This entry is called as stack frame. (frames of func^n calls)

Q^n what is there in a frame??

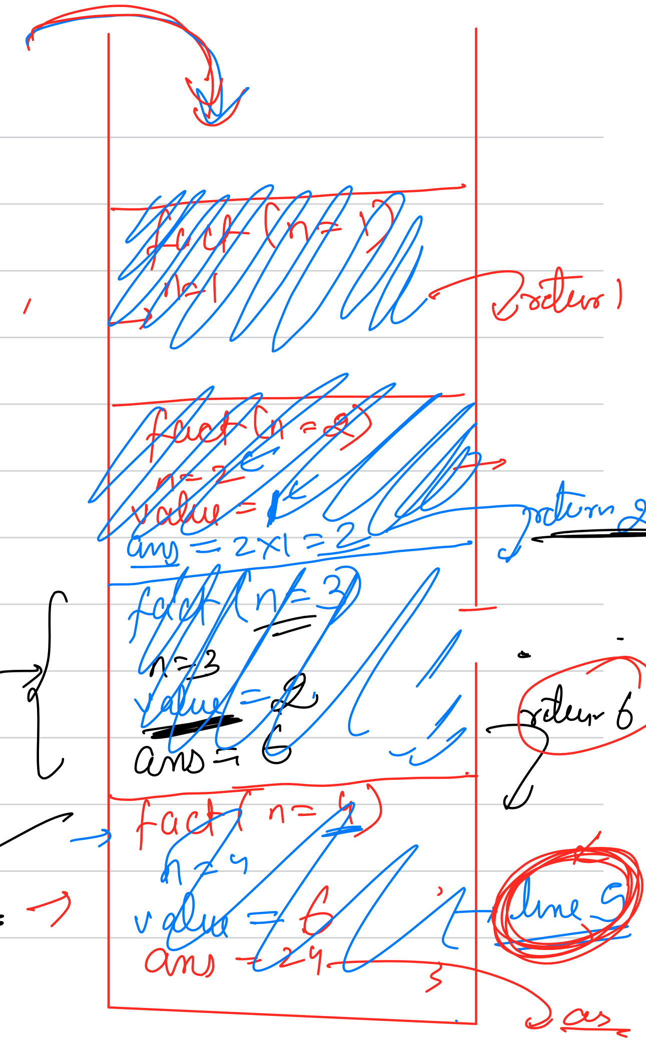
↳ all the local variables of a func^n are there in the frames.

```python
1  def fact(n):
2      if n == 1: # base case
3          return 1
4
5      value = fact(n - 1) # assume this works correctly
6      ans = n * value # self work ans -> f(n)
7      return ans
8
9
10 print(fact(4))
11
```

24

return → removes the entry of the frame from call stack & returns the value.

Call stack



loops are space optimized, Recursion is not

Some algorithms are really easy to implement via recursion.