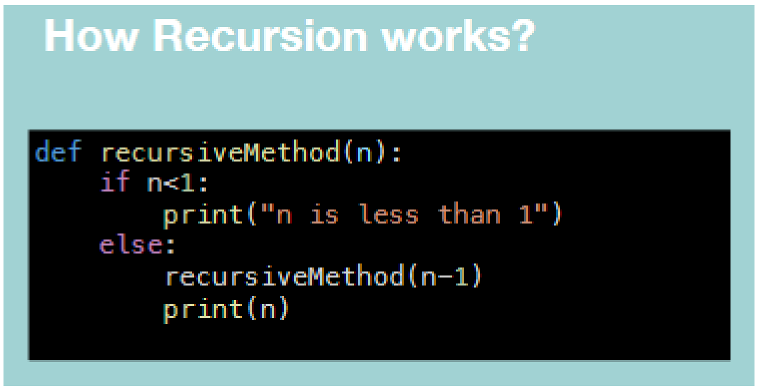
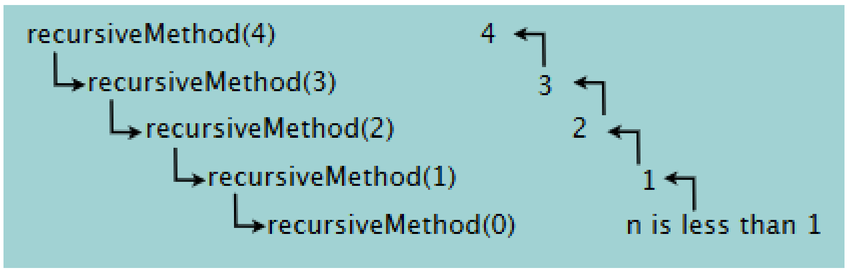
Data Structure

# Now Will See the How Recursive method is Stored in Stack Memory.


How Recursion works?

```
def recursiveMethod(n):
    if n<1:
        print("n is less than 1")
    else:
        recursiveMethod(n-1)
        print(n)
```

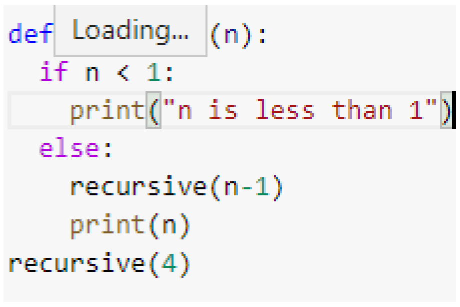
1st



2nd

Note: See the above image where it states about how Recursion Works.

- one function is there ie; recursiveMethod() user is given input 4 so the flow of the recursive function is the same as in the above image.



```
def Loading... (n):
  if n < 1:
    print("n is less than 1")
  else:
    recursive(n-1)
    print(n)
recursive(4)
```

```
n is less than 1
1
2
3
4
```

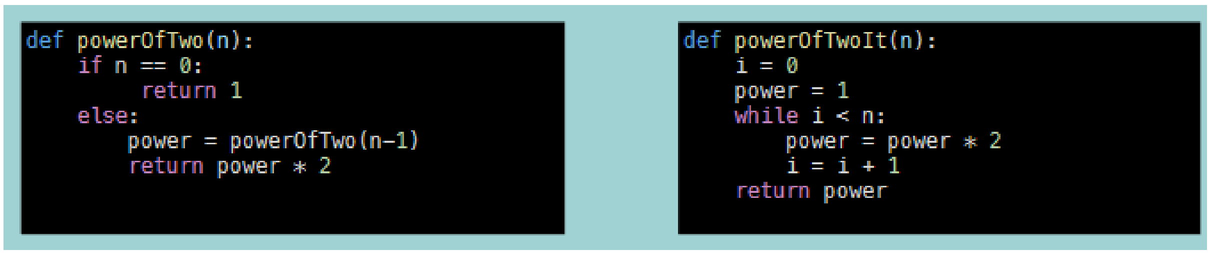
3rd

- see the 2nd image based on the LIFO (Stack) as we can see the last method recursiveMethod(0) called. so lastmethod will pop out first. ie; n is less than 1 and so on.

Note: we understood that stack memory is used by the system for managing the recursive calls.

- So every time Recursive method calls itself, the system stores it in the stack for coming back because there are execution (print) statement left after calling itself.
- This means that system somehow remembers the point where it should stops, and call to function with different parameter. based on the condition.

Recursive vs Iterative Solutions



```
def powerOfTwo(n):
    if n == 0:
         return 1
    else:
        power = powerOfTwo(n-1)
        return power * 2
```

```
def powerOfTwoIt(n):
    i = 0
    power = 1
    while i < n:
        power = power * 2
        i = i + 1
    return power
```

- Here we can see the two functions are given one based on the Recursion, and another based on the iterative traditional method of looping concept.

- in the Recursion function, as we can see the above image we have a one condition to stop further execution (Infinite Loop).
- if the condition is not satisfied then it will execute the else block and return the power of 2.
- Conditional statement decides the termination of Recursion.
- Here in Recursive function, infinite Recursion can leads the system crash.
- Recursion repeatedly invokes (triggering) the mechanism consequently (accordingly) as per method calls.
- so conclusion it can be Expensive for both processor time and memory Space. as we can discuss in previous section where it recursive function call stored the function in stack memory.
- That means if the algorithm resources depth of N (N is number of times so it directly depends on the number so till it will execute based on the Number. so it uses at least O(N) memory.

# in other side in Iterative solution

- we have created a two variable which is i and power, until i will less than 1 this while loop will execute till that time.
- As per analyzing the two function, Recursion code is easy to write. compare to Iterative one.
- but here variable value of i will decide the termination of execution.
- here in iterative function, infinite iteration consume CPU cycles ie; (CPU usage).
- where in iterative function while executive it will not store any instance while executing

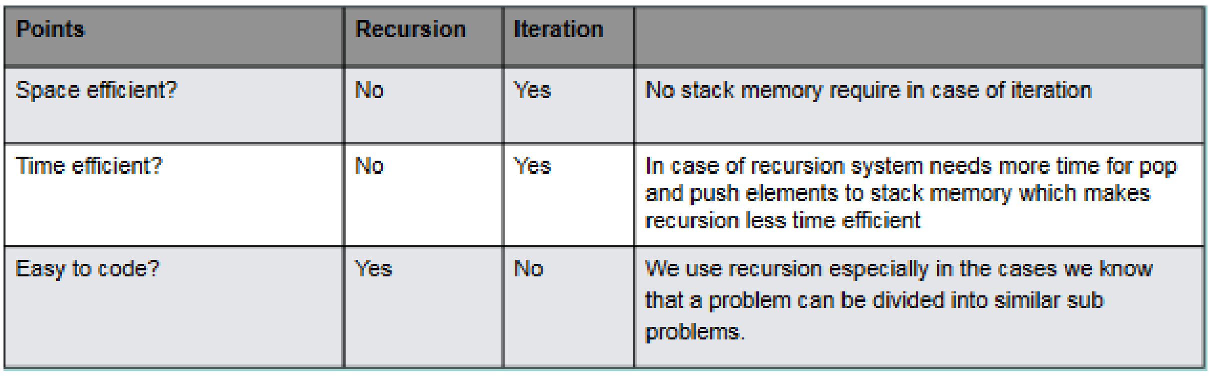
Note: So for this reason, its better to implement the recursive Algorithm iteratively. (which based on the loop system.)

# so the question is does iteration look always better than Recursion.?

- so the answer is NO because each iteration logic having their own Advantage,

- We use the Recursion especially in the case of when the bigger problem is divided in similar problems.
- And when we deal with tree and graph the uses of recursion is more on that concept.

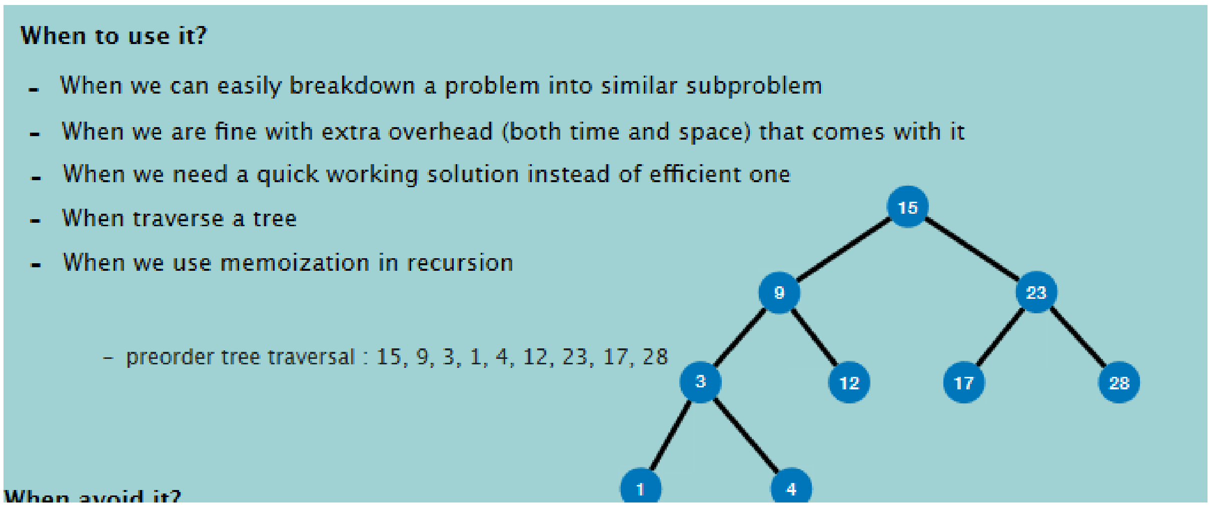
Differences between Recursion Vs Iteration.



Points	Recursion	Iteration	
Space efficient?	No	Yes	No stack memory require in case of iteration
Time efficient?	No	Yes	In case of recursion system needs more time for pop and push elements to stack memory which makes recursion less time efficient
Easy to code?	Yes	No	We use recursion especially in the cases we know that a problem can be divided into similar sub problems.

When To Use Or Avoid a Recursion.

- Iteration performs better than Recursion in terms of time and space complexity. (its consuming the memory space)


15
9
23
3
12
17
28
1
4

When to use it?

- When we can easily breakdown a problem into similar subproblem
- When we are fine with extra overhead (both time and space) that comes with it
- When we need a quick working solution instead of efficient one
- When traverse a tree
- When we use memoization in recursion

  - preorder tree traversal : 15, 9, 3, 1, 4, 12, 23, 17, 28

When avoid it?

- The second point describes about as we already saw recursion take a space (Memory Space) and time(Time Complexity). So if your System is ready/able to handle the both complexity then you can use Recursion.

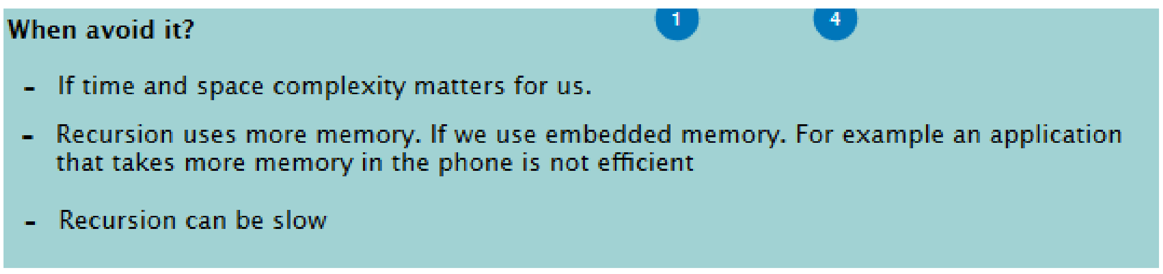
Note: for Example suppose if your developing a mobile Application Which should run on low-memory devices as well. So in this Cases, Recursion is not Advisable.

Note : suppose if you are developing an algorithm for a critical system. which should must be so fast, like Air Bag system in the car or gun bullets shots, where a fraction of speed matters a lot. so in this case avoid Recursion.

Note: Refer the third point, as we can see the writing code using Recursion is much easier. ie; Solving the mathematical problems like Fibonacci or factorial so, in this case we can use Recursion as an easy solution.

Note: Tree is a collection of an object, that are linked to one another. Recursion is very beneficial when we use pre-order traversal where we used Recursion. from the above image take a reference traversal flow.

When To Avoid a Recursion.



When avoid it?

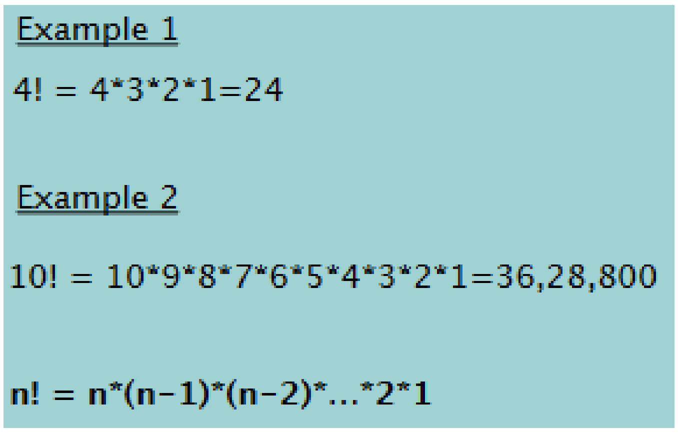
- If time and space complexity matters for us.
- Recursion uses more memory. If we use embedded memory. For example an application that takes more memory in the phone is not efficient
- Recursion can be slow

- The first point tells about we already saw when we use Recursion, its took time and space (for every execution the instance were stored into Stack) so its takes time and space while executing.

- Recursion using more memory because the Recursive function has to add the into the stack with each Recursive call and keep storing the values until function is not finished.
- Recursion can slow means for every call value has to be stored into the stack so its time consuming process and also its based on stack (LIFO).

Recursion Writes In Optimise Way.

# Write a Factorial code.



Example 1

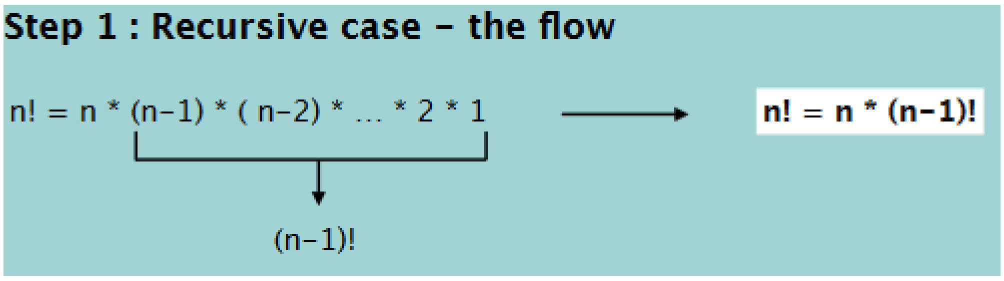
$4! = 4*3*2*1=24$

Example 2

$10! = 10*9*8*7*6*5*4*3*2*1=36,28,800$

$\mathbf{n! = n*(n-1)*(n-2)*\dots*2*1}$

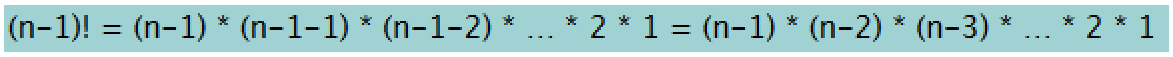
Note: Factorial of 0! is 1.



Step 1 : Recursive case – the flow

$$n! = n * \underbrace{(n-1) * (n-2) * \dots * 2 * 1}_{(n-1)!} \longrightarrow \boxed{\mathbf{n! = n * (n-1)!}}$$

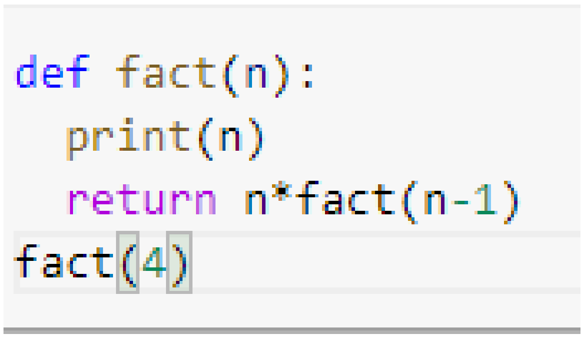
- Calculation be like the above image.



$(n-1)! = (n-1) * (n-1-1) * (n-1-2) * \dots * 2 * 1 = (n-1) * (n-2) * (n-3) * \dots * 2 * 1$

- So above the image express the expression of (n-1)! Factorial.

Note: So how we can express this term with code point of view that we can see the below.



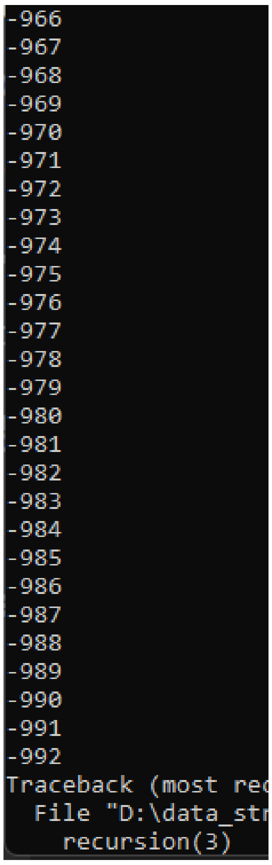
```python
def fact(n):
  print(n)
  return n*fact(n-1)
fact(4)
```

Note: See the above image this is the Recursion Function, so is this enough for execute the Recursion. so answer is No, because some bug in this code, while executing you got to know.

- So python will throw the Error , RecursionError is that maximum recursion depth exceeded while calling a Python object.

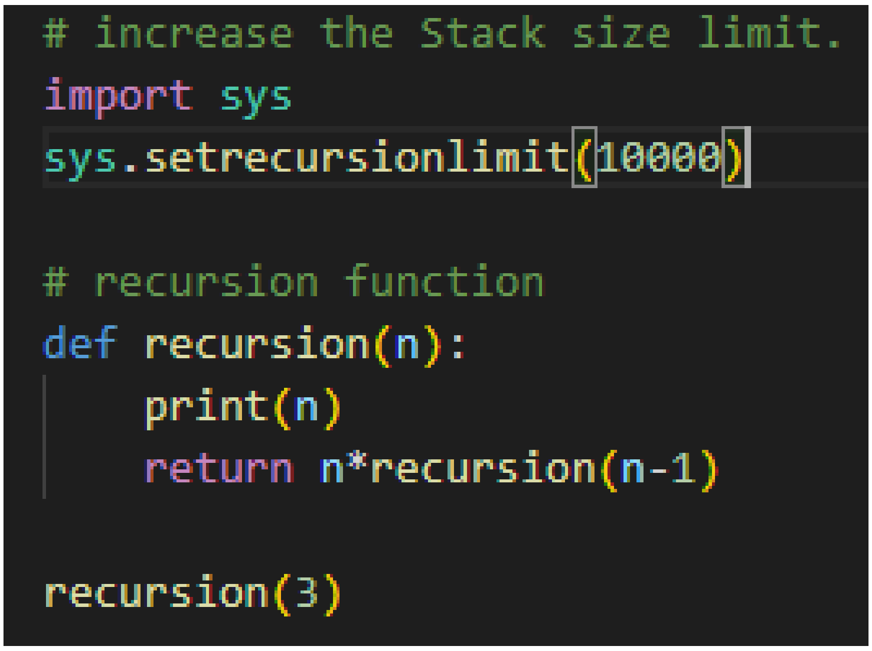
```
RecursionError                            Traceback (most recent call last)
<ipython-input-2-73926926907f> in <module>
      2   print(n)
      3   return n*fact(n-1)
----> 4 fact(4)

                         1 frames
... last 1 frames repeated, from the frame below ...

<ipython-input-2-73926926907f> in fact(n)
      1 def fact(n):
      2   print(n)
----> 3   return n*fact(n-1)
      4 fact(4)

RecursionError: maximum recursion depth exceeded while calling a Python object
```

- It means the number of object limitation in Stack memory is exceed, stack not able to increase a size itself.



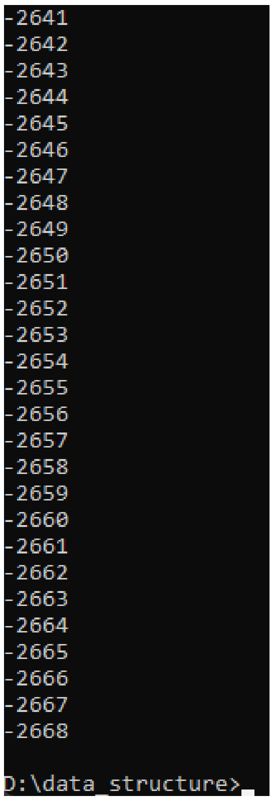
```
-966
-967
-968
-969
-970
-971
-972
-973
-974
-975
-976
-977
-978
-979
-980
-981
-982
-983
-984
-985
-986
-987
-988
-989
-990
-991
-992
Traceback (most rec
  File "D:\data_str
    recursion(3)
```

Note: We can increase the size limit of stack manually through code.



```python
# increase the Stack size limit.
import sys
sys.setrecursionlimit(10000)

# recursion function
def recursion(n):
    print(n)
    return n*recursion(n-1)

recursion(3)
```

Note: After Increasing the size limit.

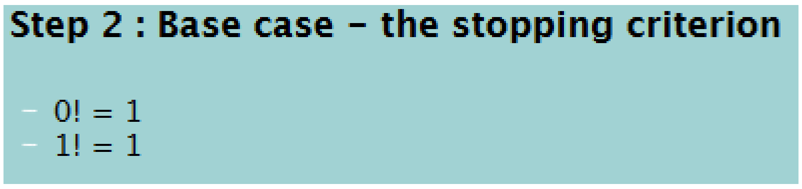


```
-2641
-2642
-2643
-2644
-2645
-2646
-2647
-2648
-2649
-2650
-2651
-2652
-2653
-2654
-2655
-2656
-2657
-2658
-2659
-2660
-2661
-2662
-2663
-2664
-2665
-2666
-2667
-2668

D:\data_structure>
```

# So How to prevent this situation, how to use Recursion in Efficient way, while excecuting not need to increase the memory of Stack.

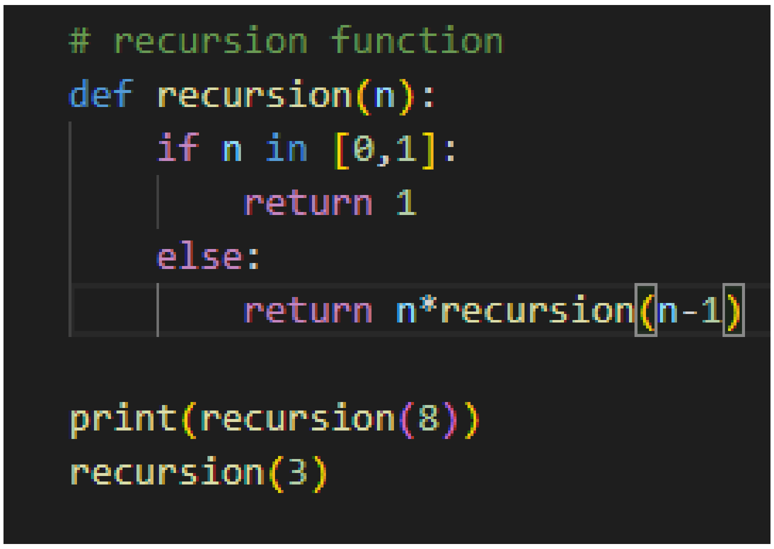
Note: So here function calling itself continuously. so how we can prevent from this situation.

- As we can know Already, Factorial of 0 it returns 1 and Factorial of 1 will be 1 so we have to add the base condition.



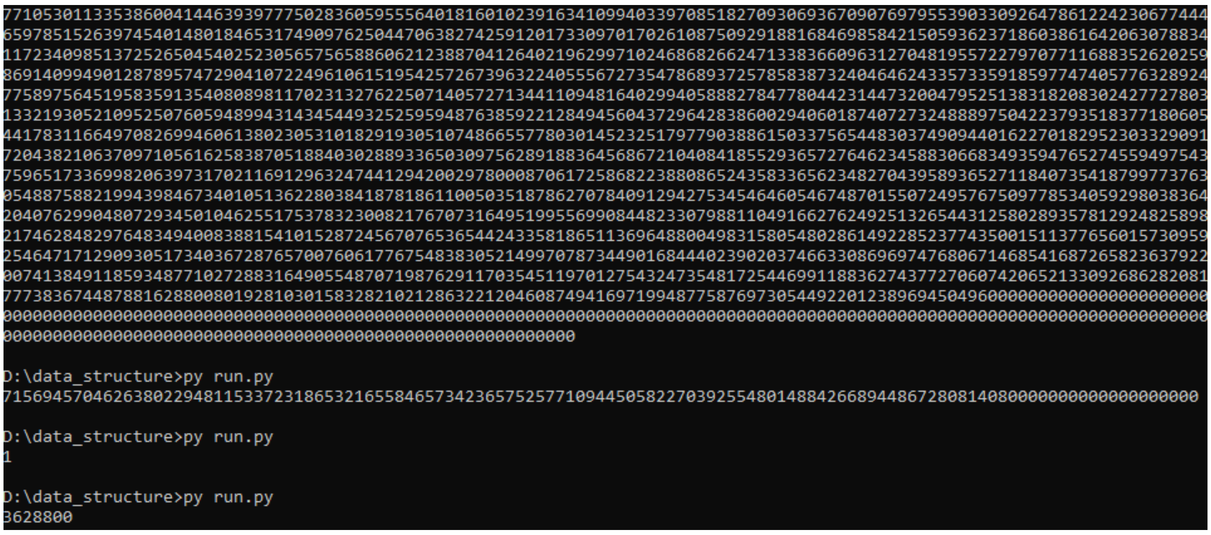
Step 2 : Base case – the stopping criterion

- 0! = 1
- 1! = 1



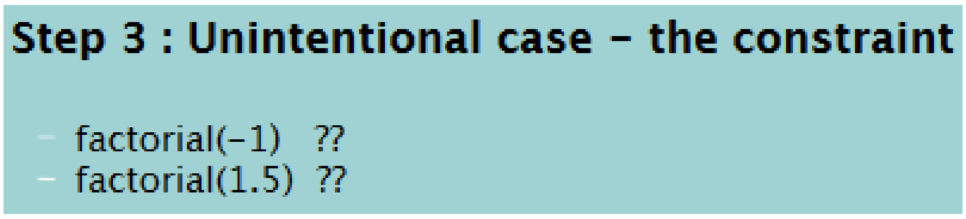
```python
# recursion function
def recursion(n):
    if n in [0,1]:
        return 1
    else:
        return n*recursion(n-1)

print(recursion(8))
recursion(3)
```

- So benefit of using the Base condition, we can prevent through RecursionError.



```
771053011335386004144639397775028360595556401816010239163410994033970851827093069367090769795539033092647861224230677444
659785152639745401480184653174909762504470638274259120173309701702610875092918816846985842150593623718603861642063078834
117234098513725265045402523056575658860621238870412640219629971024686826624713383660963127048195572279707711688352620259
869140994901287895747290410722496106151954257267396322405556727354786893725785838732404646243357335918597747405776328924
775897564519583591354080898117023132762250714057271344110948164029940588827847780442314473200479525138318208302427727803
133219305210952507605948994314345449325259594876385922128494560437296428386002940601874072732488897504223793518377180605
441783116649708269946061380230531018291930510748665577803014523251797790388615033756544830374909440162270182952303329091
720438210637097105616258387051884030288933650309756289188364568672104084185529365727646234588306683493594765274559497543
759651733699820639731702116912963247441294200297800087061725868223880865243583365623482704395893652711840735418799773763
054887588219943984673401051362280384187818611005035187862707840912942753454646054674870155072495767509778534059298038364
204076299048072934501046255175378323008217670731649519955699084482330798811049166276249251326544312580289357812924825898
217462848297648349400838815410152872456707653654424335818651136964880049831580548028614922852377435001511377656015730959
254647171290930517340367287657007606177675483830521499707873449016844402390203746633086969747680671468541687265823637922
007413849118593487710272883164905548707198762911703545119701275432473548172544699118836274377270607420652133092686282081
777383674487881628800801928103015832821021286322120460874941697199487758769730544922012389694504960000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000

D:\data_structure>py run.py
71569457046263802294811533723186532165584657342365752577109445058227039255480148842668944867280814080000000000000000000

D:\data_structure>py run.py
1

D:\data_structure>py run.py
3628800
```

Note: The most important thing to consider while using the Recursion, make sure that Function Stops for every possible Argument. So here We have to see in which case it does not stop. So for that we have to use the Third Steps.

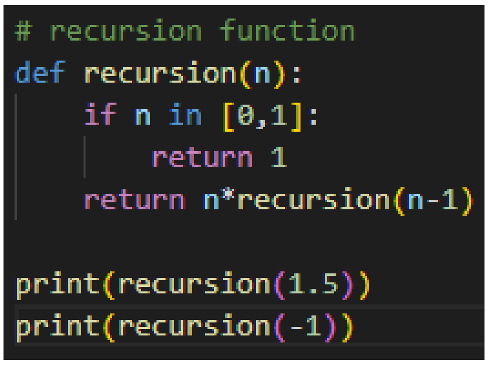


Step 3 : Unintentional case – the constraint

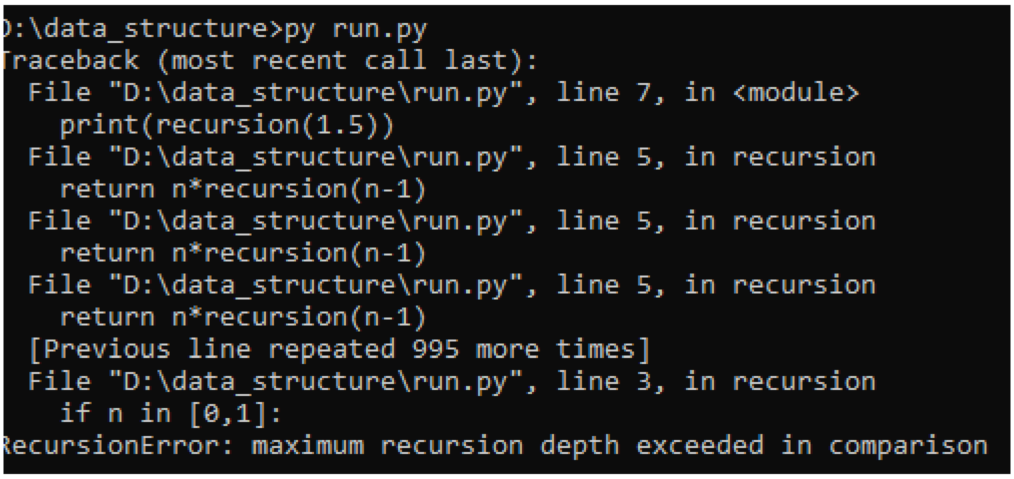
- factorial(-1) ??
- factorial(1.5) ??

Note: As we can see already as of now we only tried with positive integers, so we used the base condition that only satisfied with positive integers. what about negative or decimal integers?

- For positive Integers, we added the base condition but for negetive integers or decimal number we did not added any condition for them.



```
# recursion function
def recursion(n):
    if n in [0,1]:
        return 1
    return n*recursion(n-1)

print(recursion(1.5))
print(recursion(-1))
```

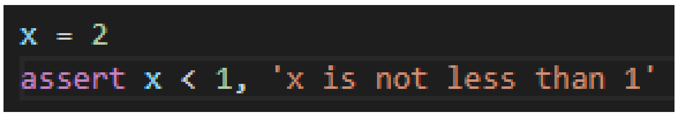


```
):\data_structure>py run.py
Traceback (most recent call last):
  File "D:\data_structure\run.py", line 7, in <module>
    print(recursion(1.5))
  File "D:\data_structure\run.py", line 5, in recursion
    return n*recursion(n-1)
  File "D:\data_structure\run.py", line 5, in recursion
    return n*recursion(n-1)
  File "D:\data_structure\run.py", line 5, in recursion
    return n*recursion(n-1)
  [Previous line repeated 995 more times]
  File "D:\data_structure\run.py", line 3, in recursion
    if n in [0,1]:
RecursionError: maximum recursion depth exceeded in comparison
```

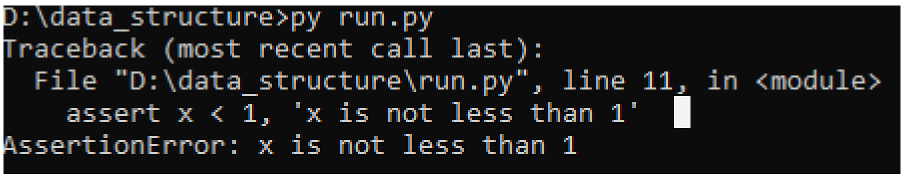
Note: For negetive or decimal integers you will get the same error as per above image, because condition not handled for the these integers except positive integers.

# So How We Can Prevent From This Error and How to Handle.

- So in python, there is one simple method used that is Assertion Method, The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError than can be written by ourselves.



```
x = 2
assert x < 1, 'x is not less than 1'
```

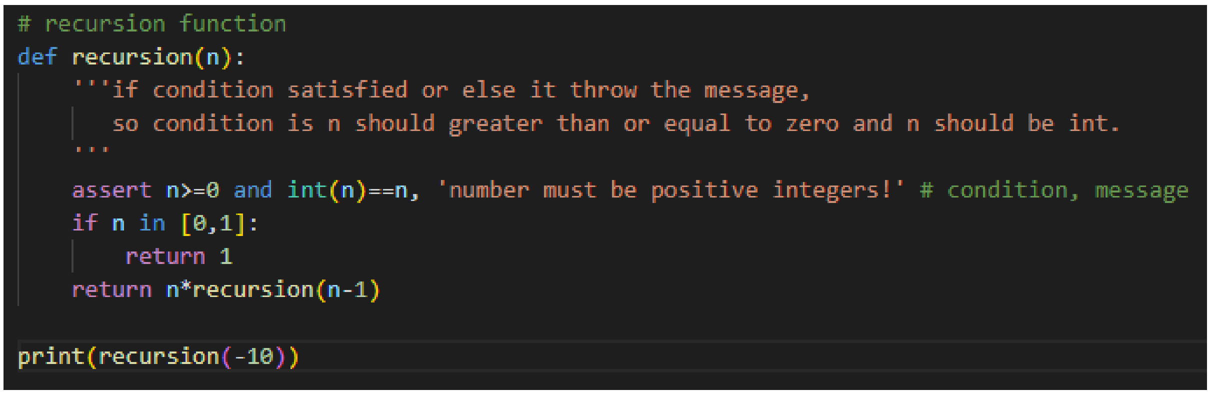


```
D:\data_structure>py run.py
Traceback (most recent call last):
  File "D:\data_structure\run.py", line 11, in <module>
    assert x < 1, 'x is not less than 1'
AssertionError: x is not less than 1
```

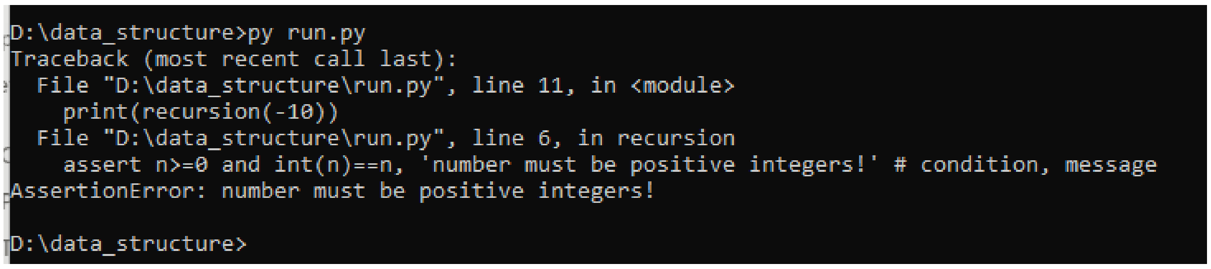
The `assert` keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError than can be written by ourselves.

Syntax

```
assert {condition}, {message}
```



```python
# recursion function
def recursion(n):
    '''if condition satisfied or else it throw the message,
       so condition is n should greater than or equal to zero and n should be int.
    '''
    assert n>=0 and int(n)==n, 'number must be positive integers!' # condition, message
    if n in [0,1]:
        return 1
    return n*recursion(n-1)

print(recursion(-10))
```

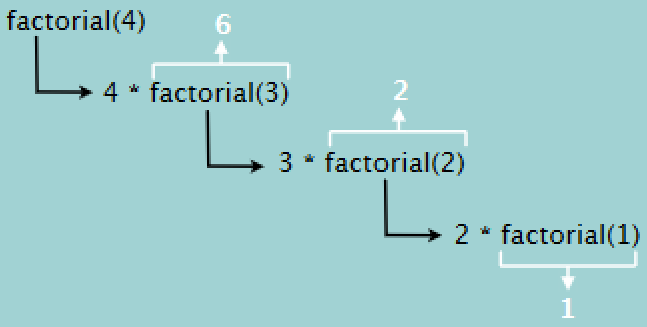


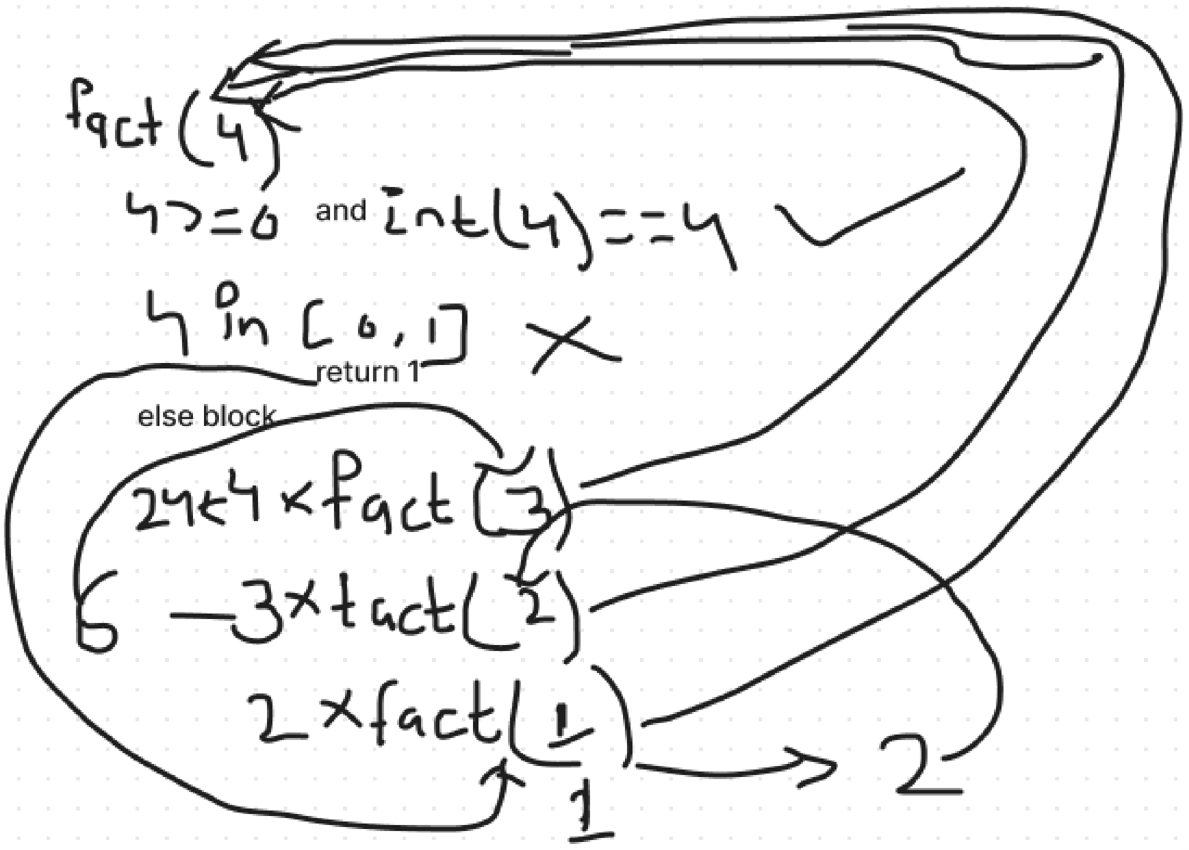
```
D:\data_structure>py run.py
Traceback (most recent call last):
  File "D:\data_structure\run.py", line 11, in <module>
    print(recursion(-10))
  File "D:\data_structure\run.py", line 6, in recursion
    assert n>=0 and int(n)==n, 'number must be positive integers!' # condition, message
AssertionError: number must be positive integers!

D:\data_structure>
```

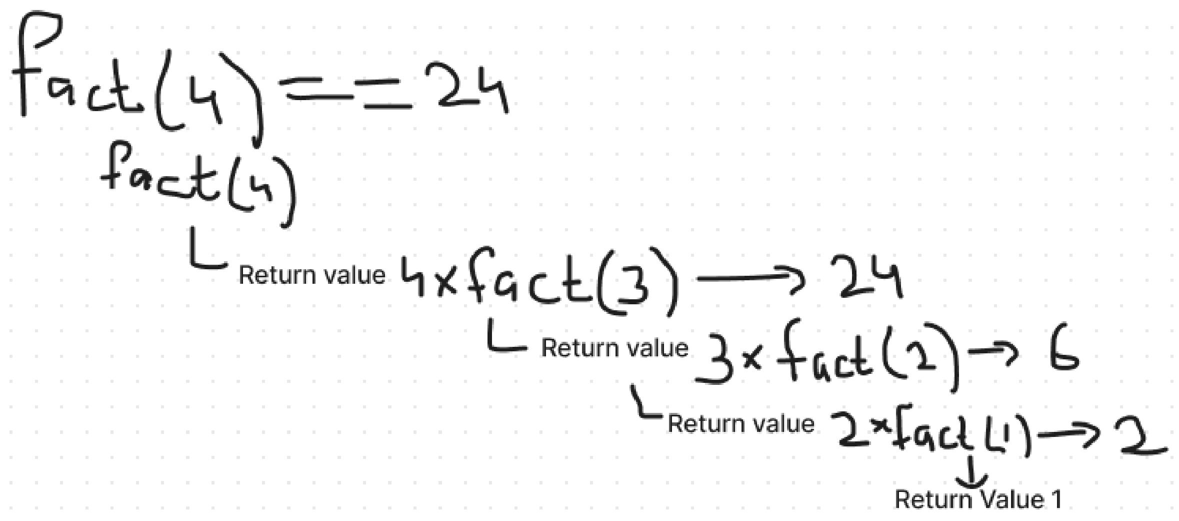
How to write recursion in 3 steps?

```
def factorial(n):
    assert n >= 0 and int(n) == n, 'The number must be positive integer only!'
    if n in [0,1]:
        return 1
    else:
        return n * factorial(n-1)
```

factorial(4) = 24



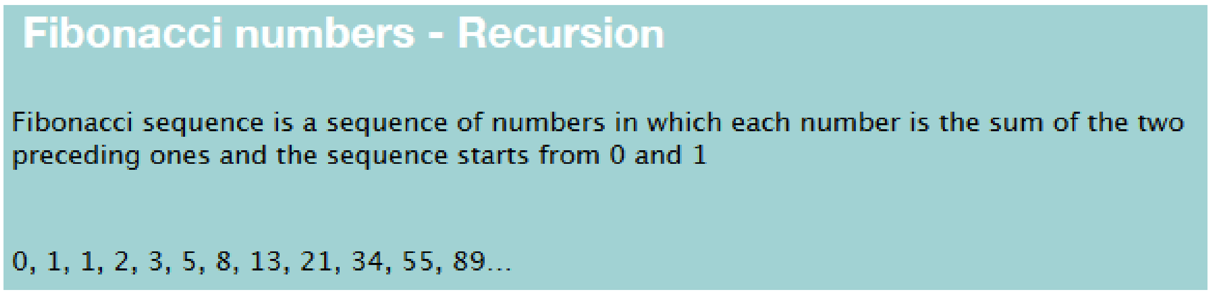



fact(4) == 24
fact(4)
Return value 4×fact(3) → 24
Return value 3×fact(2) → 6
Return value 2×fact(1) → 2
Return Value 1

Note : Return value it means that the previous function Return, Function call another Function.

How to Find Fibonnaci Numbers Using Recursion.

Note: Before starting the Fibonnaci manipulation first we should aware of what exact is Fibonnaci Numbers.

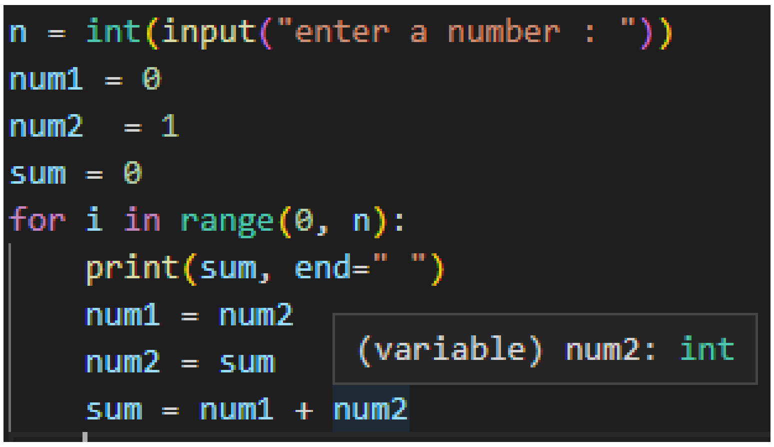


Fibonacci numbers - Recursion

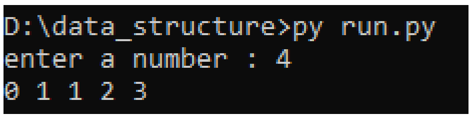
Fibonacci sequence is a sequence of numbers in which each number is the sum of the two preceding ones and the sequence starts from 0 and 1

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Note : We can Solve the fibonacci series by Loops , or Recursion also, so here first we gone a see through Loops.



```
n = int(input("enter a number : "))
num1 = 0
num2  = 1
sum = 0
for i in range(0, n):
    print(sum, end=" ")
    num1 = num2
    num2 = sum
    sum = num1 + num2
```



```
D:\data_structure>py run.py
enter a number : 4
0 1 1 2 3
```

n = int (input) 4

x = 0

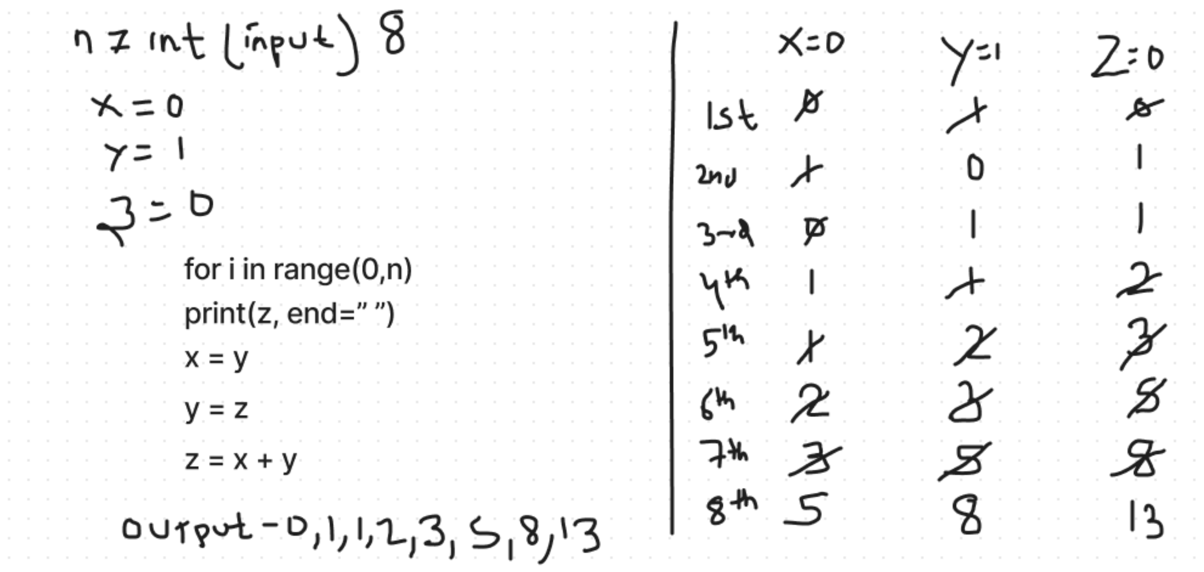
y = 1

z = 0

```
for i in range(0,n) ~~0~~ ~~1~~ ~~2~~ ~~3~~ (n-1)
    print(z, end=" ")
    x = y
    y = z
    z = x + y
```

output - 0,1,1,2

	X=0	Y=1	Z=0
1st	~~0~~	~~1~~	~~0~~
2nd	~~1~~	0	~~1~~
3rd	~~0~~	1	~~1~~
4th	1	1	2



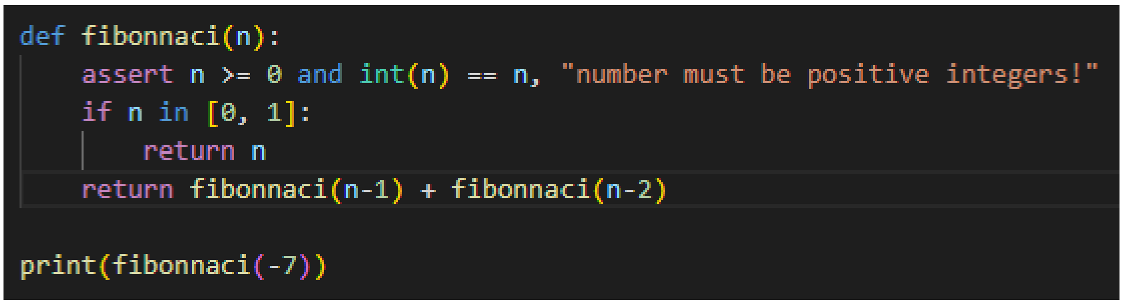
n = int (input) 8
x = 0
y = 1
z = 0

```
for i in range(0,n)
print(z, end=" ")
x = y
y = z
z = x + y
```

output - 0,1,1,2,3,5,8,13

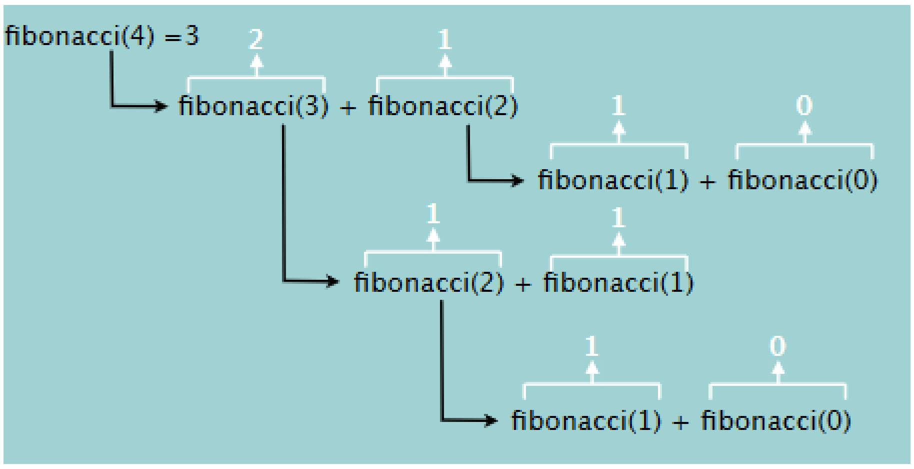
	x=0	y=1	z=0
1st	~~0~~	~~1~~	~~0~~
2nd	~~1~~	0	1
3rd	~~0~~	1	1
4th	1	~~1~~	2
5th	~~1~~	~~2~~	~~3~~
6th	~~2~~	~~3~~	~~5~~
7th	~~3~~	~~5~~	~~8~~
8th	5	8	13

Note : Using Recursion Write a Fibonnaci Series.



```python
def fibonnaci(n):
    assert n >= 0 and int(n) == n, "number must be positive integers!"
    if n in [0, 1]:
        return n
    return fibonnaci(n-1) + fibonnaci(n-2)

print(fibonnaci(-7))
```

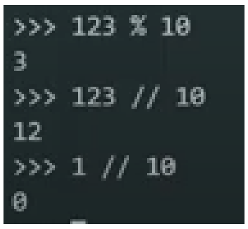
- Working Flow of behind the code.


fibonacci(4) =3
2
1
fibonacci(3) + fibonacci(2)
1
0
fibonacci(1) + fibonacci(0)
1
1
fibonacci(2) + fibonacci(1)
1
0
fibonacci(1) + fibonacci(0)

Question Practice

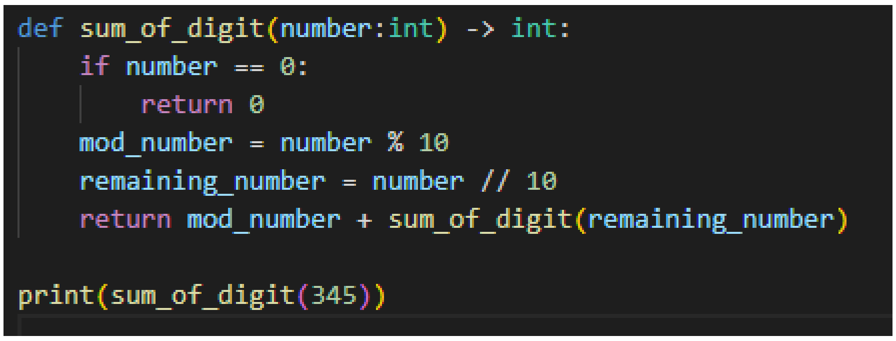
Sum of Digits

1. How to find the sum of digit of a positive integer number using Recursion.

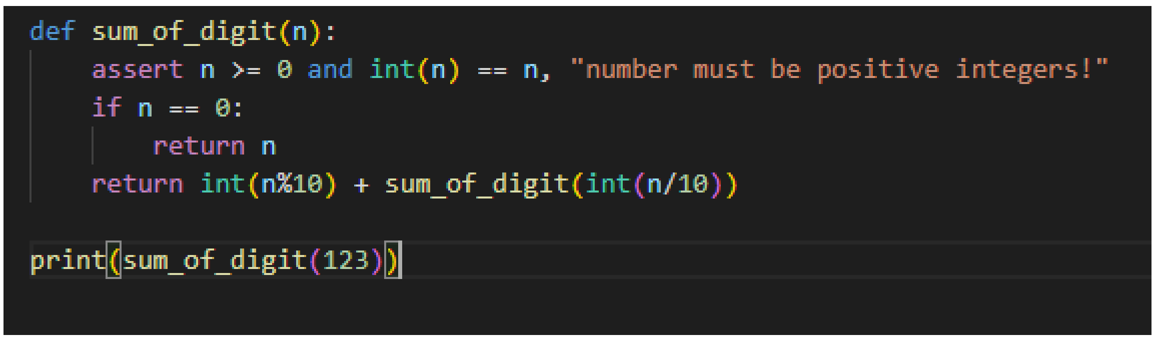


```
>>> 123 % 10
3
>>> 123 // 10
12
>>> 1 // 10
0
```

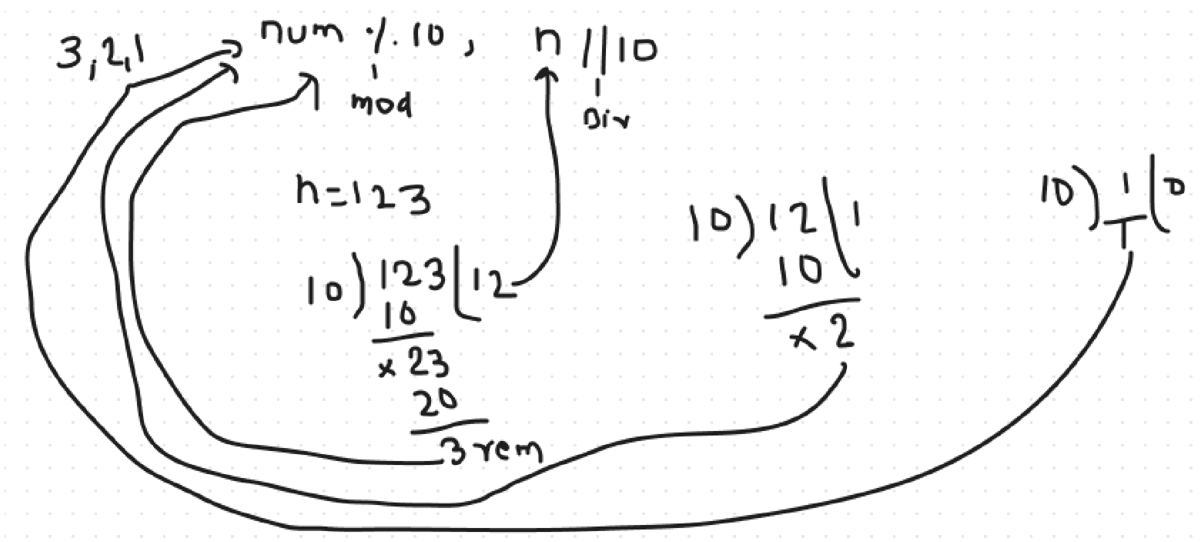
ie; 3 + 1 + 2 = 6 (Sum of Digit) using Recursion. 0 is ensuring that when will Recursion will Stop (Base Condition).



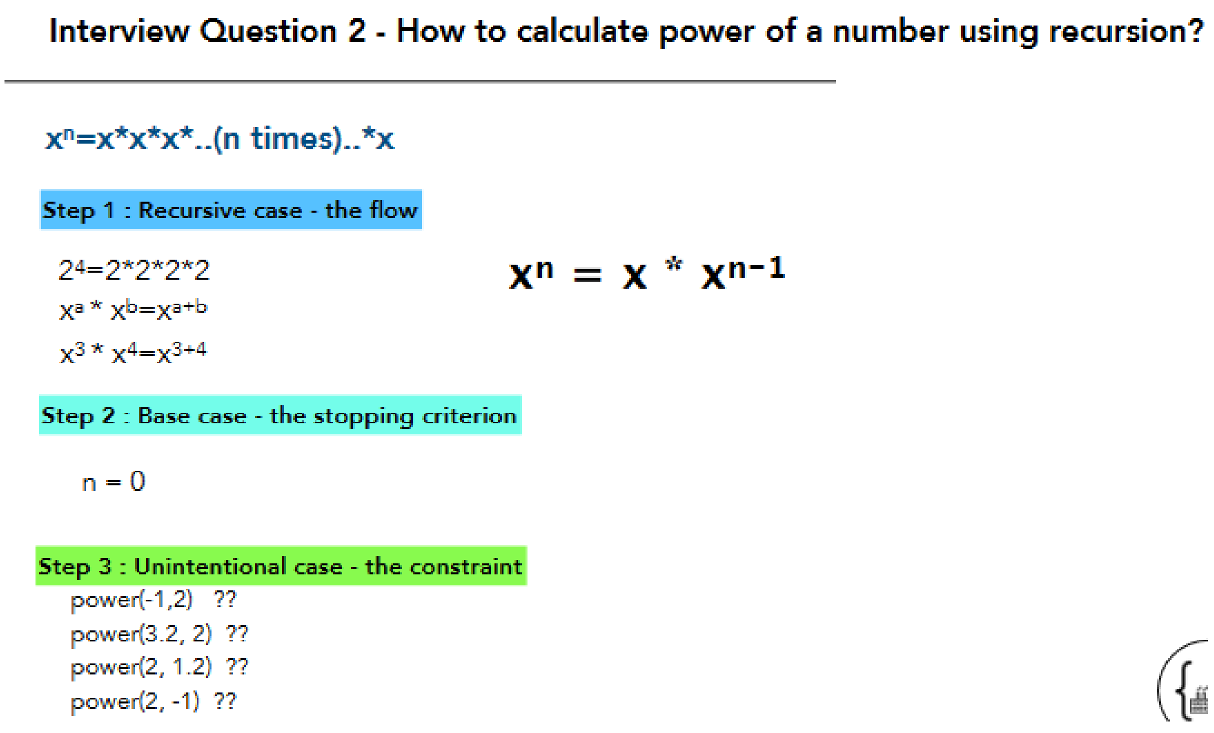
```python
def sum_of_digit(number:int) -> int:
    if number == 0:
        return 0
    mod_number = number % 10
    remaining_number = number // 10
    return mod_number + sum_of_digit(remaining_number)

print(sum_of_digit(345))
```



```python
def sum_of_digit(n):
    assert n >= 0 and int(n) == n, "number must be positive integers!"
    if n == 0:
        return n
    return int(n%10) + sum_of_digit(int(n/10))

print(sum_of_digit(123))
```



How to calculate the power of Number using Recursion.



Interview Question 2 - How to calculate power of a number using recursion?

$x^n = x*x*x*..(n \text{ times})..*x$

Step 1 : Recursive case - the flow

$2^4=2*2*2*2$

$x^a * x^b = x^{a+b}$

$x^3 * x^4 = x^{3+4}$

$$x^n = x * x^{n-1}$$

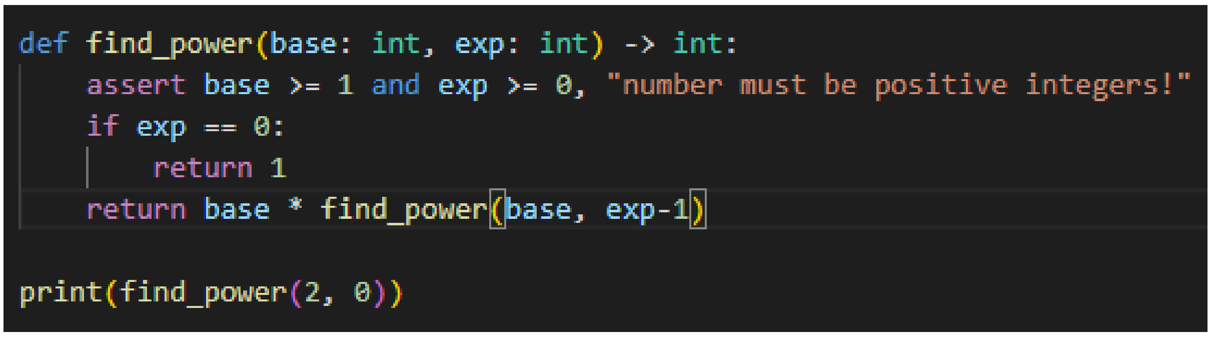
Step 2 : Base case - the stopping criterion

$n = 0$

Step 3 : Unintentional case - the constraint

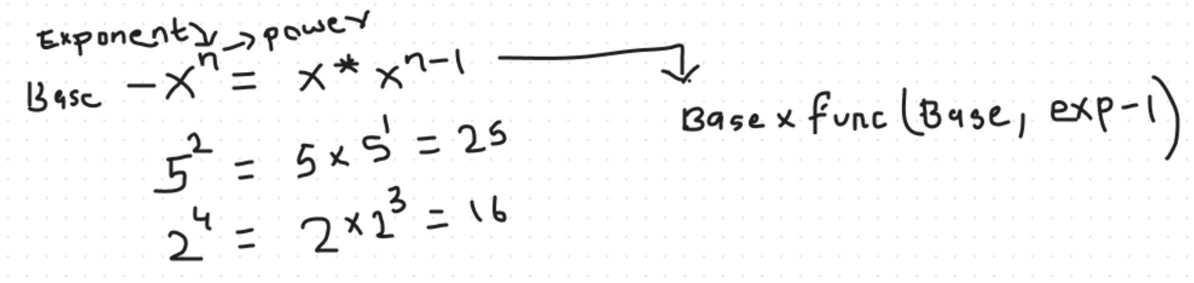
power(-1,2) ??
power(3.2, 2) ??
power(2, 1.2) ??
power(2, -1) ??

1st pic



```python
def find_power(base: int, exp: int) -> int:
    assert base >= 1 and exp >= 0, "number must be positive integers!"
    if exp == 0:
        return 1
    return base * find_power(base, exp-1)

print(find_power(2, 0))
```

2nd pic

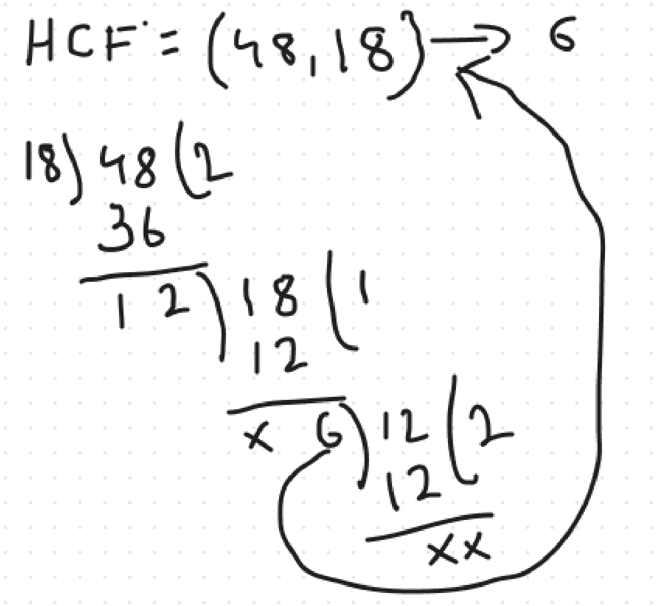


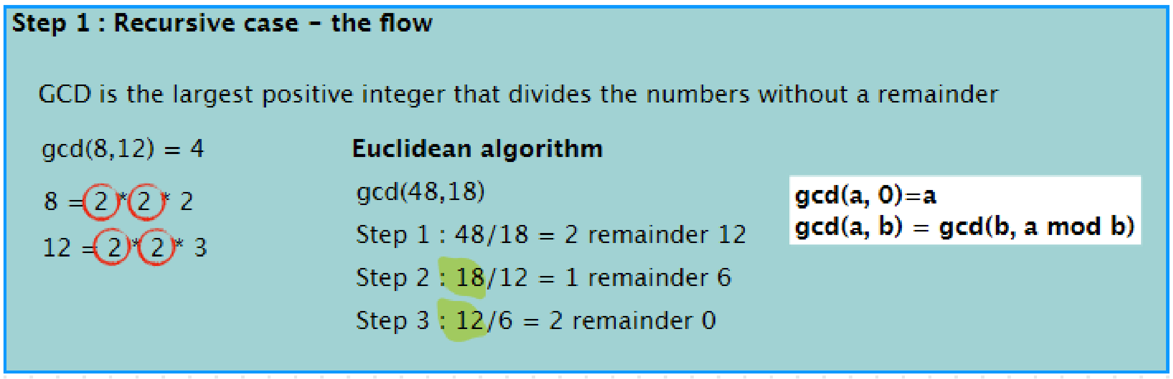
Exponent → power
Base
$-x^n = x * x^{n-1}$
$5^2 = 5 \times 5^1 = 25$
$2^4 = 2 \times 2^3 = 16$
Base x func (Base, exp-1)

Note: taking the above image reference (pic 1) so made the function as per expression.

ie; Base * function(Base, Exp -1)

How to find GCD (Greatest Common Divisor) of two numbers using Recursion. (Highest Common Factor) HCF.





Step 1 : Recursive case – the flow

GCD is the largest positive integer that divides the numbers without a remainder

gcd(8,12) = 4

8 = 2 * 2 * 2

12 = 2 * 2 * 3

Euclidean algorithm

gcd(48,18)

Step 1 : 48/18 = 2 remainder 12

Step 2 : 18/12 = 1 remainder 6

Step 3 : 12/6 = 2 remainder 0

gcd(a, 0)=a
gcd(a, b) = gcd(b, a mod b)

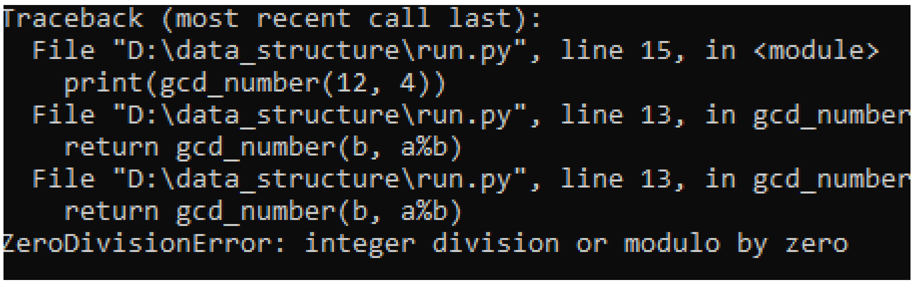
Note: As we can see in the above image every time next of gcd(a,b) a coming in the first position, (place of a), and remainder of a % b is goes to in place of b ie;

$$gcd(\underset{a}{48},\underset{b}{18})$$

$$gcd(b, a \bmod b) \Rightarrow gcd(18, 48 \% 18)$$

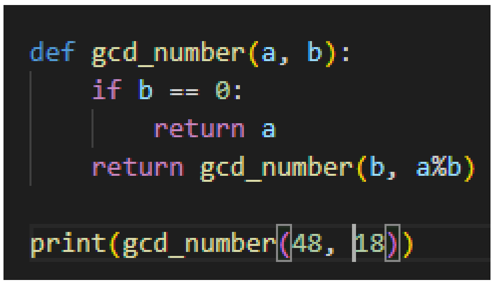
$$gcd(a,0) = a$$

$\hookrightarrow$ Base Condition

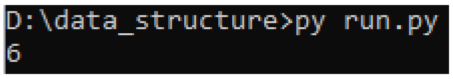


```
Traceback (most recent call last):
  File "D:\data_structure\run.py", line 15, in <module>
    print(gcd_number(12, 4))
  File "D:\data_structure\run.py", line 13, in gcd_number
    return gcd_number(b, a%b)
  File "D:\data_structure\run.py", line 13, in gcd_number
    return gcd_number(b, a%b)
ZeroDivisionError: integer division or modulo by zero
```

Note: if You want to prevent the ZeroDivisionError, so for that we have to use the base condition.

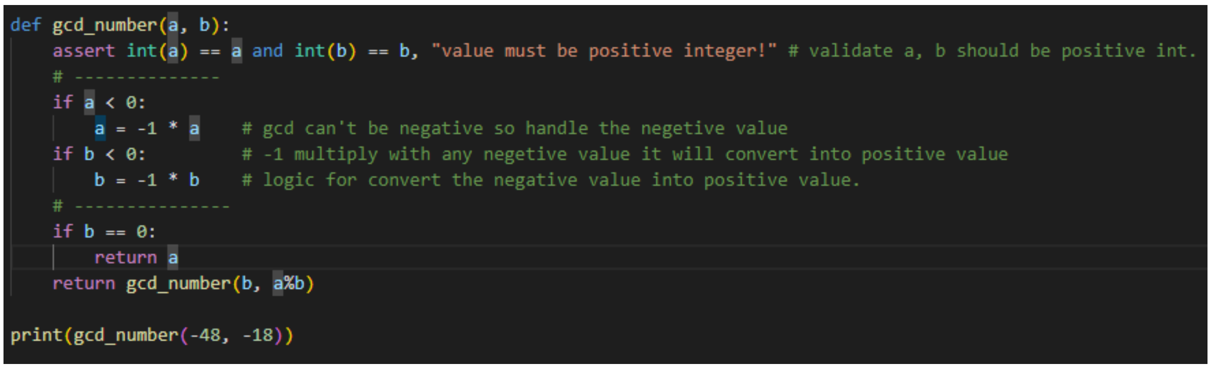


```python
def gcd_number(a, b):
    if b == 0:
        return a
    return gcd_number(b, a%b)

print(gcd_number(48, 18))
```



```
D:\data_structure>py run.py
6
```

- so as we can see the here gcd value is coming as per Expected, but still its not proper one still some of validation pending here. (-48, -18) → -6
- As we know the gcd cannot be negative, so in the case of negative value we have to use one more condition, which will be based on the negative case.



```python
def gcd_number(a, b):
    assert int(a) == a and int(b) == b, "value must be positive integer!" # validate a, b should be positive int.
    # --------------
    if a < 0:
        a = -1 * a    # gcd can't be negative so handle the negetive value
    if b < 0:         # -1 multiply with any negetive value it will convert into positive value
        b = -1 * b    # logic for convert the negative value into positive value.
    # ---------------
    if b == 0:
        return a
    return gcd_number(b, a%b)

print(gcd_number(-48, -18))
```