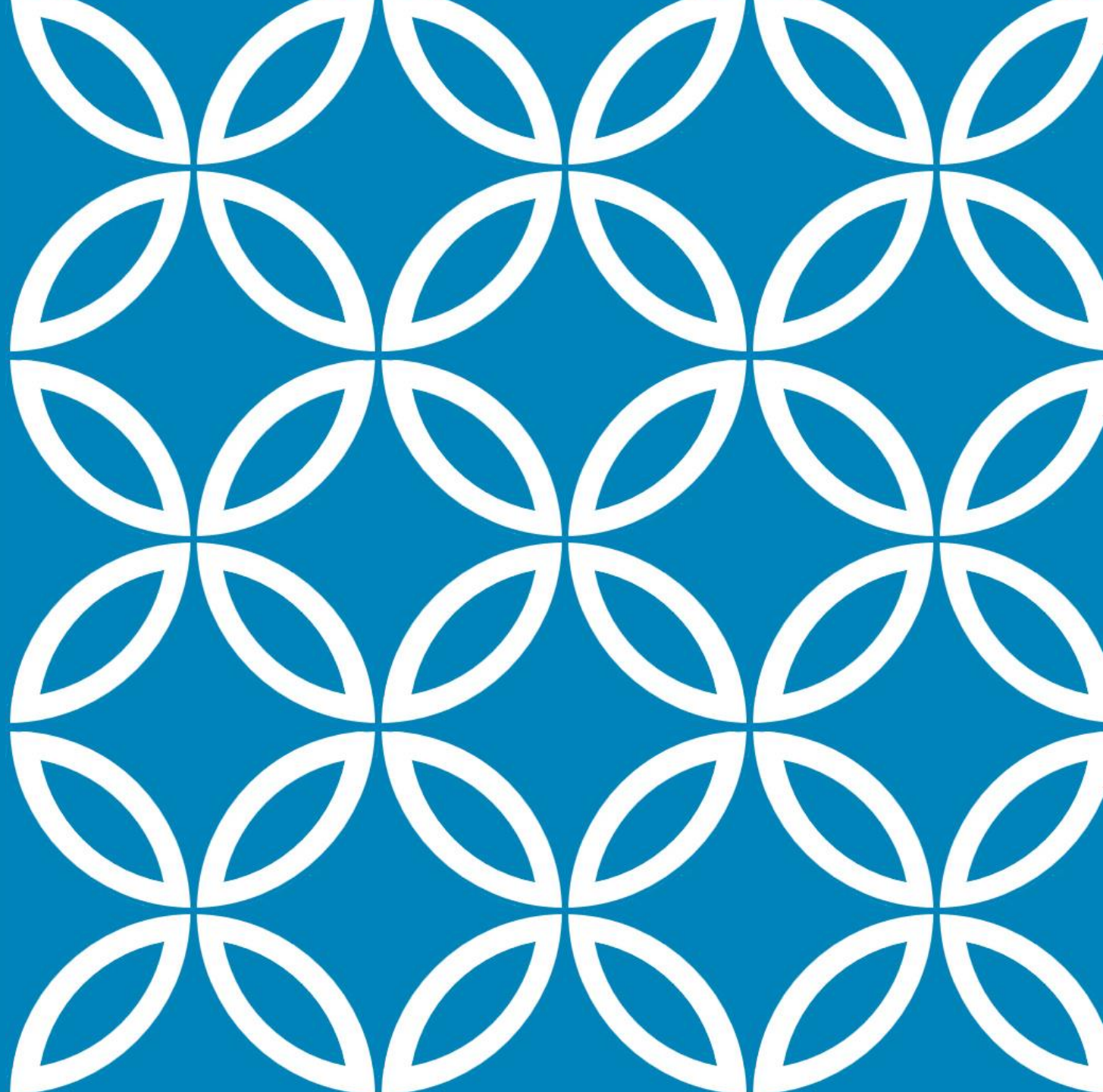


FUNCTIONS & RECURSION



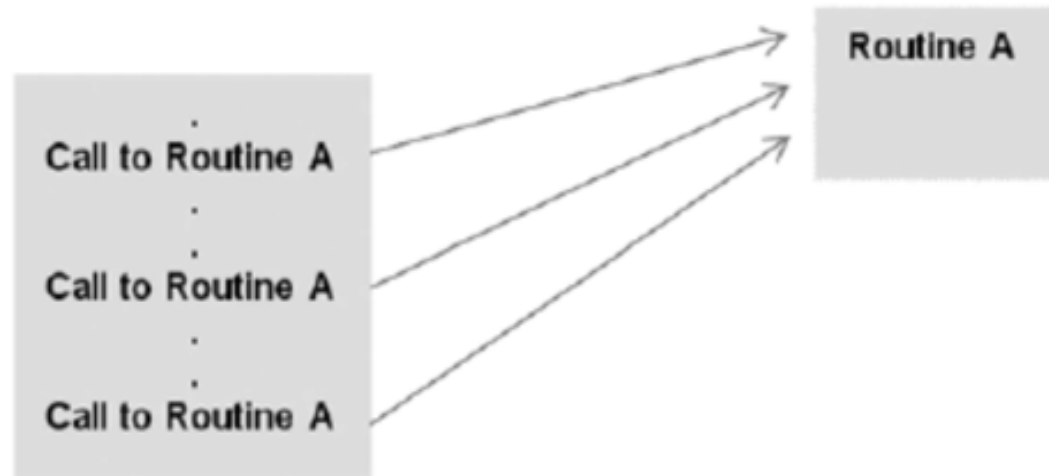
PROGRAM ROUTINES

Most of the computer programs that solve real world problems are much bigger and complex than the programs presented in the first few chapters. The problem is one of complexity. Some smart phones, for example, contain over 10 million lines of code. Imagine the effort needed to develop and debug software of that size. It certainly cannot be implemented by any one person, it takes a team of programmers to develop such a project.

In order to manage the complexity of a large problem, it is broken down into smaller sub problems. Then, each sub problem can be focused on and solved separately. In programming, we do the same thing. **Programs are divided into manageable pieces called program routines (or simply routines).** Doing so is a form of abstraction in which a more general, less detailed view of a system can be achieved. In addition, program routines provide the opportunity for code reuse, so that systems do not have to be created from “scratch.” Routines, therefore, are a fundamental building block in software development.

FUNCTION ROUTINES

A routine is a named group of instructions performing some task. A routine can be invoked (called) as many times as needed in a given program, as shown in Figure. When a routine terminates, execution automatically returns to the point from which it was called. Such routines may be predefined in the programming language, or designed and implemented by the programmer. A function is Python's version of a program routine. Some functions are designed to return a value, while others are designed for other purposes.



FUNCTIONS

A function is a collection of statements grouped together that performs an operation. Functions can be used to define reusable code and organize and simplify code. Suppose that you need to find the sum of integers from 1 to 10, 20 to 37, and 35 to 49. If you create a program to add these three sets of numbers, your code might look like as given in first figure. You may have observed that the code for computing these sums is very similar, except that the starting and ending integers are different. Wouldn't it be nice to be able to write commonly used code once and then reuse it? You can do this by defining a function, which enables you to create reusable code. For example, the preceding code can be simplified by using functions, as used in second figure .

FUNCTIONS

```
# Without function
sum = 0
for i in range(1, 11):
    sum += i
print("Sum from 1 to 10 is", sum)
sum = 0
for i in range(20, 38):
    sum += i
print("Sum from 20 to 37 is", sum)
sum = 0
for i in range(35, 50):
    sum += i
print("Sum from 35 to 49 is", sum)
```

```
#With function
def sum(i1, i2):
    result = 0
    for i in range(i1, i2 + 1):
        result += i
    return result

def main():
    print("Sum from 1 to 10 is", )
    print("Sum from 20 to 37 is", sum(20, 37))
    print("Sum from 35 to 49 is", sum(35, 49))
main()    # Call the main function
```

FUNCTION CALL

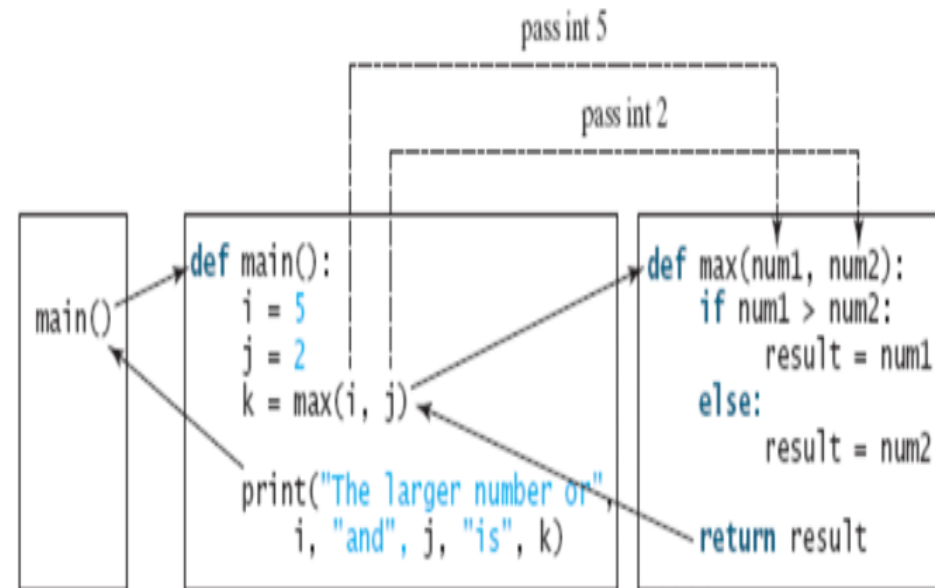
maxpro.py - C:\Users\dipen\AppData\Local\Programs\Python\Python36-32\maxpro.py (3.6.1)

File Edit Format Run Options Window Help

```
# return the max of two numbers
def max(num1, num2):
    if num1 > num2:
        return num1
    else:
        return num2

def main():
    i=5
    j=2
    k=max(i, j)
    print("the larger number of", i, "and", j, "is", k)
```

main()



TYPES OF FUNCTIONS

*functionwithoutreturn.py - C:/Users/dipen/AppData/Local/Programs/Python/Python36

File Edit Format Run Options Window Help

```
# Print grade for the score
```

```
def printGrade(score):
```

```
    if score >= 90.0:
```

```
        print('A')
```

```
    elif score >= 80.0:
```

```
        print('B')
```

```
    elif score >= 70.0:
```

```
        print('C')
```

```
    elif score >= 60.0:
```

```
        print('D')
```

```
    else:
```

```
        print('F')
```

```
def main():
```

```
    score = eval(input("Enter a score: "))
```

```
    print("The grade is ", end = " ")
```

```
    printGrade(score)
```

```
main()
```

```
# Call the main function
```

functionwithreturn.py - C:/Users/dipen/AppData/Local/Programs,

File Edit Format Run Options Window Help

```
# return grade for the score
```

```
def getGrade(score):
```

```
    if score >= 90.0:
```

```
        return 'A'
```

```
    elif score >= 80.0:
```

```
        return 'B'
```

```
    elif score >= 70.0:
```

```
        return 'C'
```

```
    elif score >= 60.0:
```

```
        return 'D'
```

```
    else:
```

```
        return 'F'
```

```
def main():
```


```
    score = eval(input("Enter a score: "))
```

```
    print("The grade is ", getGrade(score))
```

```
main()
```

```
# Call the main function
```


DIFFERENT TYPES OF ARGUMENTS

 *defaultargument.py - C:/Users/dipen/AppData/Local/Programs/Python/Python36-32/defaultargument.py (3.6.1)*

File Edit Format Run Options Window Help

```
def printArea(width = 1, height = 2 ):
    area = width * height
    print("width:", width, "\theight:", height, "\tarea:", area)

printArea()                # Default arguments width = 1 and height = 2
printArea(4,2.5)          # Positional arguments width = 4 and height = 2.5
printArea(height = 5, width = 3) # Keyword arguments width
printArea(width = 1.2 )   # Default height = 2
printArea(height = 6.2)  # Default width = 1
```


PRACTICE QUESTION 1

```
def Display(Name,age):  
    print("Name =",Name,"age =",age)
```

```
Display("John",25)
```

```
Display(40,"Sachin")
```

(a) Name = John age = 25

error

(b) error

(c) Name = John age = 25

Name = 40 age = Sachin

PRACTICE QUESTION 2

```
def Display(Name,age):  
    print("Name =",Name,"age =",age)  
  
Display("John")
```

(a) Name = John age = 0

(b) Name = John age = age

(c) Name = john

(d) error

PRACTICE QUESTION 3

```
def Display(num1,num2):
```

```
    print(num1,num2)
```

```
Display(40,num2=10)
```

(a) 40 10

(b) error

PRACTICE QUESTION 4

```
def Display(num1,num2):
```

```
    print(num1,num2)
```

```
Display(num2=10,40)
```

(a) 40 10

(b) 10 40

(c) error

VARIABLE LENGTH ARGUMENTS

Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition.

For example, if the programmer is writing a function to add two numbers, he can write:

```
add(a, b)
```

But, the user who is using this function may want to use this function to find sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as:

```
add(10, 15, 20)
```

Then the `add()` function will fail and error will be displayed. If the programmer wants to develop a function that can accept 'n' arguments, that is also possible in Python..

VARIABLE LENGTH ARGUMENTS

For this purpose, a variable length argument is used in the function definition.

A variable length argument is an argument that can accept any number of values.

The variable length argument is written with a ' * ' symbol before it in the function definition as:

```
def add(farg, *args):
```

Here, 'farg' is the formal argument and '*args' represents variable length argument. We can pass 1 or more values to this '*args' and it will store them all in a tuple. A tuple is like a list where a group of elements can be stored. In Program 19, we are showing how to use variable length argument

RETURNING MULTIPLE VALUES

The Python return statement can return multiple values. Python allows a function to return multiple values. Figure below defines a function that takes two numbers and returns them in ascending order.

```
returnmultiple.py - C:/Users/dipen/AppData/Local/Pr  
File Edit Format Run Options Window Help  
def sort(number1, number2):  
    if number1 < number2:  
        return number1, number2  
  
    else:  
        return number2, number1  
  
n1, n2 = sort (3,2)  
print("n1 is", n1)  
print("n2 is", n2)  
|
```

The sort function returns two values. When it is invoked, you need to pass the returned values in a simultaneous assignment.

(a) n1 is 3

n2 is 2

(b) n1 is 2

n2 is 3

(c) n1 is 3

n2 is 3

(d) n1 is 2

n2 is 2

PRACTICE QUESTION 1

A GAME OF LUCK

A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.

PRACTICE QUESTION 2

Write a program to print calendar for a specific month in a year

```
Enter full year (e.g., 2001): 2011 ↵
Enter month as number between 1 and 12: 9 ↵

September 2011
-----
Sun Mon Tue Wed Thu Fri Sat
    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
```

LOCAL VS GLOBAL VARIABLES

```
def Demo():
```

```
    q = 10 #Local variable q
```

```
    print("The value of Local variable q:",q)
```

```
Demo()
```

```
print("The value of Local variable q:",q)
```

```
p = 20 #global variable p
```

```
def Demo():
```

```
    q = 10 #Local variable q
```

```
    print("The value of Local variable q:",q)
```

```
    print("The value of Global Variable p:",p)
```

```
Demo()
```

```
print("The value of Global Variable p:",p)
```

```
print("The value of Local variable q:",q)
```

GLOBAL KEYWORD

```
p = 20 #global variable p
```

```
def Demo():
```

```
    q = 10 #Local variable q
```

```
        print("The value of Local variable q:",q)
```

```
        print("The value of Global Variable p:",p)
```

```
Demo()
```

```
print("The value of Global Variable p:",p)
```

```
print("The value of Local variable q:",q)
```

```
p = 20 #global variable p
```

```
def Demo():
```

```
    global q
```

```
    q = 10 #Local variable q
```

```
        print("The value of Local variable q:",q)
```

```
        print("The value of Global Variable p:",p)
```

```
Demo()
```

```
print("The value of Global Variable p:",p)
```

```
print("The value of Local variable q:",q)
```

LAMBDA FUNCTION

- Lambda functions are named after the Greek letter λ (lambda).
- These are also known as anonymous functions.
- Such kind of functions are not bound to a name.
- They only have a code to execute that which is associated with them.
- The basic syntax for a lambda function is:

Name = lambda(variables): Code

LAMBDA FUNCTION

```
def func(x):  
    return x*x*x
```

```
print(func(3))
```

```
cube = lambda x: x*x*x    #Define lambda  
function
```

```
print(cube(3))           #Call lambda function
```



RECURSION

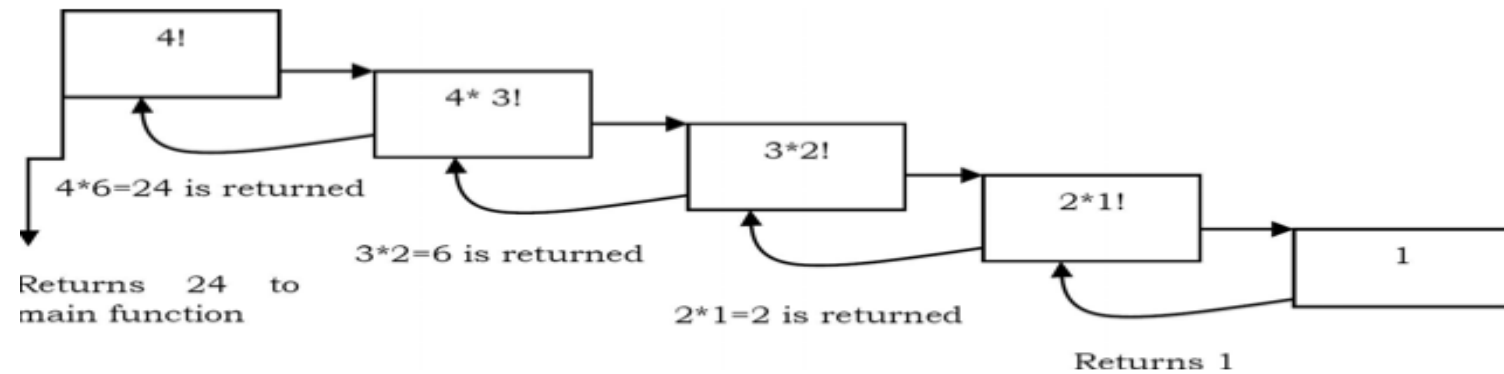
Recursion is a useful technique borrowed from mathematics.

Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted.

Recursion is most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search, and traversal problems often have simple recursive solutions.

RECURSION

```
// calculates factorial of a positive integer
def factorial(n):
    if n == 0: return 1
    return n*factorial(n-1)
```



NORMAL RECURSION VS TAIL RECURSION (FACTORIAL OF A NUMBER)

Normal Recursion

```
*rec.py - C:\Users\Dipen\AppData\Local\Programs\Python\Python37-32\rec.py (3.7.4)*
File Edit Format Run Options Window Help
def fact(n):
    if n==1:
        return 1
    else:
        return n*fact(n-1)
def main():
    num = eval(input("enter the number"))
    x=fact(num)
    print(x)
main()
```

Tail Recursion

```
tr.py - C:\Users\Dipen\AppData\Local\Programs\Python\Python37-32\tr.py (3.7.4)
File Edit Format Run Options Window Help
def fact(n,result):
    if n==1:
        return result
    else:
        return fact(n-1,n*result)
def main():
    num = eval(input("enter the number"))
    x=fact(num,1)
    print(x)
main()
```

HOW TO CONVERT A NON TAIL RECURSIVE FUNCTION INTO A TAIL RECURSIVE FUNCTION?

A non tail recursion function can be converted to a tail recursive function by adding one or more auxiliary parameters .

For example, result is added as an auxiliary parameter in the definition of function **fact** in the previous example.

THE FUNCTION `SUM_POSITIVE_NUMBERS` SHOULD RETURN THE SUM OF ALL POSITIVE NUMBERS BETWEEN THE NUMBER `N` RECEIVED AND 1. FOR EXAMPLE, WHEN `N` IS 3 IT SHOULD RETURN $1+2+3=6$, AND WHEN `N` IS 5 IT SHOULD RETURN $1+2+3+4+5=15$. FILL IN THE GAPS TO MAKE THIS WORK:

```
def sum_positive_numbers(n):
```

```
    # The base case is n being smaller than 1
```

```
    if n < 1:
```

```
        return _____
```

```
    # The recursive case is adding this number to the sum of the numbers smaller than this one.
```

```
    return _____ + sum_positive_numbers(____)
```

```
print(sum_positive_numbers(3))      # Should be 6
```

```
print(sum_positive_numbers(5))      # Should be 15
```

FILL IN THE BLANKS TO MAKE THE `IS_POWER_OF` FUNCTION RETURN WHETHER THE NUMBER IS A POWER OF THE GIVEN BASE. NOTE: BASE IS ASSUMED TO BE A POSITIVE NUMBER. TIP: FOR FUNCTIONS THAT RETURN A BOOLEAN VALUE, YOU CAN RETURN THE RESULT OF A COMPARISON.

```
def is_power_of(number, base):
```

```
    if number < base:
```

```
        # Base case: when number is smaller than base.
```

```
        # If number is equal to 1, it's a power (base**0).
```

```
        return _____
```

```
    return is_power_of(__, __) # Recursive case: keep dividing number by base.
```

```
print(is_power_of(8,2)) # Should be True
```

```
print(is_power_of(64,4)) # Should be True
```

```
print(is_power_of(70,10)) # Should be False
```

FIBONACCI SERIES:0 1 1 2 3 5 8 13 21

ENTER THE TERM NUMBER(FOR EXAMPLE TERM NUMBER OF 0 IS 1,2 IS 4.....) IN OUTPUT WE WILL HAVE THE NUMBER AT THAT LOCATION

```
fibrec.py - C:/Users/Dipen/AppData/Local/Programs/Python/Python37-32/fibrec.py (3.7.4)
File Edit Format Run Options Window Help
def fib_norm(n1):
    if n1==1:
        return 0
    elif n1==2:
        return 1
    else:
        return fib_norm(n1-1)+fib_norm(n1-2)

def main():
    n= eval(input("enter the term no. "))
    term = fib_norm(n)
    print("Fibonacci term is",term)
main()
```

NORMAL RECURSION VS TAIL RECURSION (FIBONACCI FUNCTION)

NORMAL RECURSION

```
fibrec.py - C:/Users/Dipen/AppData/Local/Programs/Python/Python37-32/fibrec.py (3.7.4)
File Edit Format Run Options Window Help
def fib_norm(n1):
    if n1==1:
        return 0
    elif n1==2:
        return 1
    else:
        return fib_norm(n1-1)+fib_norm(n1-2)

def main():
    n= eval(input("enter the term no. "))
    term = fib_norm(n)
    print("Fibonnaci term is",term)
main()
```

TAIL RECURSION

```
fibtailrec.py - C:/Users/Dipen/AppData/Local/Programs/Python/Python37-32/fibtailrec.py (3.7.4)
File Edit Format Run Options Window Help
def fib_tail(n1,next,result):
    if n1==1:
        return result
    else:
        return fib_tail(n1-1, next+result,next)

def main():
    n= eval(input("enter the term no. "))
    term = fib_tail(n,1,0)
    print("Fibonnaci term is",term)
main()
```


RECURSION VS ITERATION

A recursive function generally takes more time to execute than an equivalent iterative approach. This is because the multiple function calls are relatively time-consuming. In contrast, while and for loops execute very efficiently. Thus, when a problem can be solved both recursively and iteratively with similar programming effort, it is generally best to use an iterative approach.

Recursion	Iteration
<ul style="list-style-type: none">• Terminates when a base case is reached.	<ul style="list-style-type: none">• Terminates when a condition is proven to be false.
<ul style="list-style-type: none">• Each recursive call requires extra space on the stack frame(memory)	<ul style="list-style-type: none">• Each iteration does not require extra space.
<ul style="list-style-type: none">• If we get infinite recursion, the program may run out of memory and gives stack overflow.	<ul style="list-style-type: none">• An infinite loop could forever since there is no extra memory being created.
<ul style="list-style-type: none">• Solutions to some problems are easier to formulate recursively.	<ul style="list-style-type: none">• Iterative solutions to a problem may not always be as obvious as a recursive solution.

IMPORTANT POINTS ABOUT RECURSION

Recursive algorithms have two types of cases, recursive cases and base cases.

Every recursive function case must terminate at a base case.

Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].

A recursive algorithm can be implemented without recursive function calls using a stack, but it's usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.

For some problems, there are no obvious iterative algorithms.

Some problems are best suited for recursive solutions while others are not.

EXAMPLE ALGORITHMS OF RECURSION

Fibonacci Series, Factorial Finding

Merge Sort, Quick Sort

Binary Search

Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder

Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]

Dynamic Programming Examples

Divide and Conquer Algorithms

Towers of Hanoi

Backtracking