Example:

```
def calculator(n1,n2,opr): # function parameters are local variables

    def add():
        return n1+n2
    def sub():
        return n1-n2
    def multiply():
        return n1*n2
    def div():
        return n1/n2

    if opr=='+':
       return add()
    elif opr=='-':
        return sub()
    elif opr=='*':
        return multiply()
    elif opr=='/':
        return div()


def main():
    num1=int(input("Enter first number "))
    num2=int(input("Enter second number "))
    opr=input("Enter Operator ")
    result=calculator(num1,num2,opr)
    print(f'result of {num1}{opr}{num2}={result}')


main()
```

Output:

```
Enter first number 10
Enter second number 5
Enter Operator +
result of 10+5=15

Enter first number 5
Enter second number 2
Enter Operator *
result of 5*2=10
```

Decorator function or Decorator

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. A decorator is a special function, which input as one function and return modified function/updated function or transformed function. Decorator is used to transform one function to another function.

Examples of predefined decorators:
@staticmethod, @abstractmethod, @classmethod

There are two steps in working with decorators

1. Creating decorator
2. Assigning to decorator to function or applying decorator to function

Basic step for creating decorator

1. Define a function which receive input as another function
2. Inside function write another function, which transform function to another function

3. Return inner function/transformed function

Assigning decorator to function

```
@decoratorname
```

Example:

```
def decorator1(f): # Outer function
    def fun2(): # Inner function
        f()
        print("transformed function")
    return fun2


@decorator1
def fun1():
    print("This is function1")



# Interanal concept

#f2=decorator1(fun1)
#f2()

fun1()
```

Output:

```
This is function1
transformed function
```

Example:

```
def box(f):
    def display_box():
        print("*"*40)
        f()
        print("*"*40)
    return display_box

@box
def display():
    print("PYTHON LANGUAGE")

@box
def print_data():
    stud={'naresh':'python',
          'suresh':'java',
          'ramesh':'oracle'}
    for name,course in stud.items():
        print(f'{name}-->{course}')



display()
print_data()
```

Output

```
****************************************
PYTHON LANGUAGE
****************************************
****************************************
naresh-->python
suresh-->java
ramesh-->oracle
****************************************
```

Why use Python decorators?
Decorators give you the ability to modify the behavior of functions without altering their source code, providing a concise and flexible way to enhance and extend their functionality.

Example

```
def smart_div(f):
    def new_div(n1,n2):
        if n2==0:
            return 0
        else:
            return f(n1,n2)
    return new_div

@smart_div
def div(n1,n2):
    n3=n1/n2
    return n3

def main():
    num1=int(input("Enter first number "))
    num2=int(input("Enter second number "))
    num3=div(num1,num2)
    print(f'result is {num3}')

main()
```

Output

```
Enter first number 5
Enter second number 0
```

```
result is 0

Enter first number 4
Enter second number 2
result is 2.0
```

decorator chaining

Chaining decorators means applying more than one decorator inside a function.

Example:

```
def b(f):
   def y():
      print("b decorator")
      f()
   return y


def a(f):
   def x():
      print("a decorator")
      f()
   return x

@b
@a
def fun1():
   print("inside fun1")

fun1()
```

Output:

```
b decorator
a decorator
inside fun1
```