PES UNIVERSITY

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science
and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit - 3: Functions – Recursion

Prof. Sowmya Shree P, Prof. Sindhu R Pai

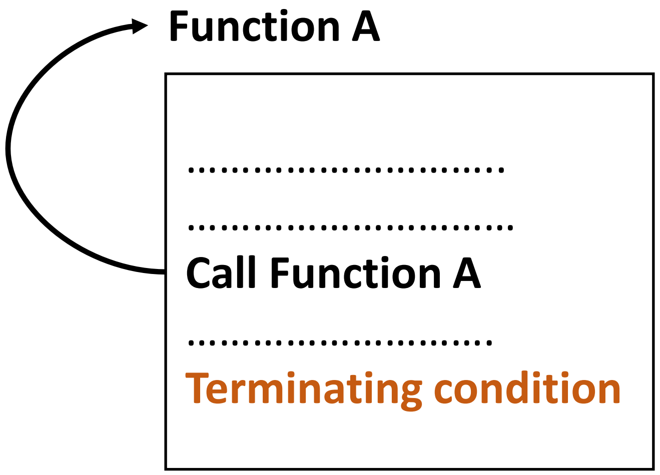
Department of Computer Science and Engineering

Recursion : Function calling itself

- There are many problems for which the solution can be expressed in terms of the problem itself.

- Computational problem solving via the use of recursion is a powerful problem-solving approach.

- Ex: Factorial of n can be expressed as n times factorial of (n – 1) if n > 0 and factorial of 0 can be expressed as 1.



Function A

…………………………

…………………………

Call Function A

…………………………

Terminating condition

Recursive Function Definition

Recursion : Function calling itself

Need for Recursion

- We can reduce the length of our code and make it easier to read and write.
- Solve complex problems by breaking them down to simpler ones.

Recursion : Function calling itself

- **Criteria for Recursion:**

  1. Recursive function should have a terminating/stopping condition

  2. Each call to recursive function must be towards terminating condition

Recursion : Function calling itself

- **Characteristics of Recursive function:**

1. There must be at least one base case whose solution is known without further recursive breakdown.

2. Problems that are not a base case are broken down into subproblems and work towards a base case.

3. There is a way to derive the solution of the original problem from the solutions of the recursively solved subproblems.

Recursion : Example: Computation of factorial of a number

factorial(0)=1
factorial(1)=1*1
factorial(2)=2*1
factorial(3)=3*2*1
factorial(4)=4*3*2*1
.
.
.

$$\text{factorial}(n)=n*\underbrace{(n-1)*(n-2)*\ldots\ldots\ldots\ldots\ldots\ldots*1}$$

factorial(n)=n * factorial(n-1)

The complete definition of the factorial function is,

$$\textbf{factorial(n)=}\begin{cases}\textbf{1} & \textbf{if n=0}\\ \textbf{n*factorial(n-1)} & \textbf{otherwise}\end{cases}$$

How Recursion Works?

- Recursion is implemented using stack because activation records are to be stored in LIFO order (last in first out).
- An activation record of a function call contains arguments, return address and local variables of the function.
- Stack is a linear data structure in which elements are inserted to the top and deleted from the top.

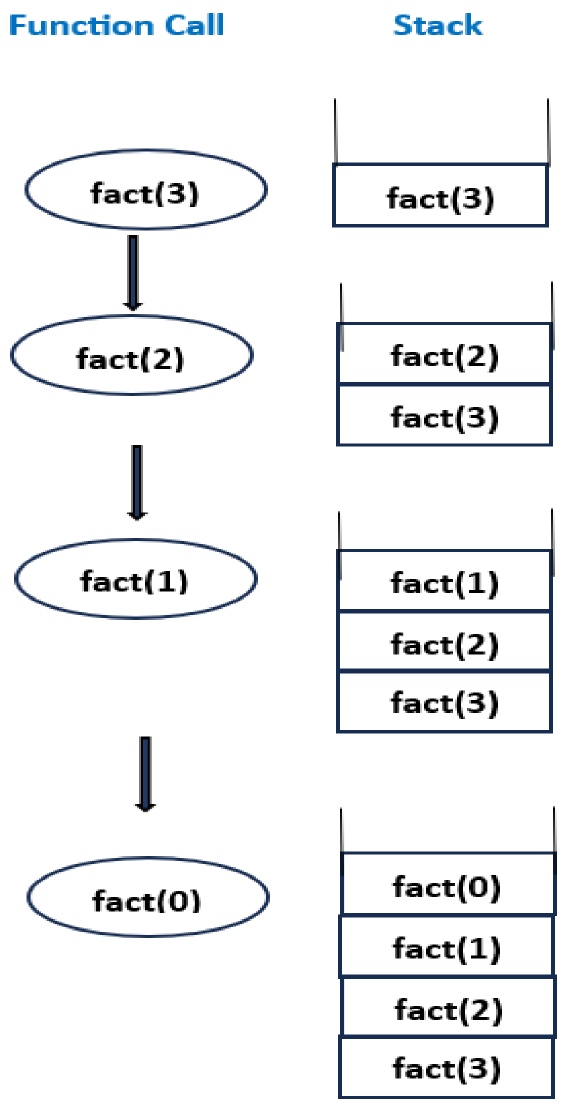
Implementation of Recursion using Stack

- Consider the example, Computing factorial of a number using recursion

```
def fact(n): #Recursive Function
        if n == 0 : #terminating condition
                res = 1
        else:
                res = n * fact(n - 1)
        return res
```

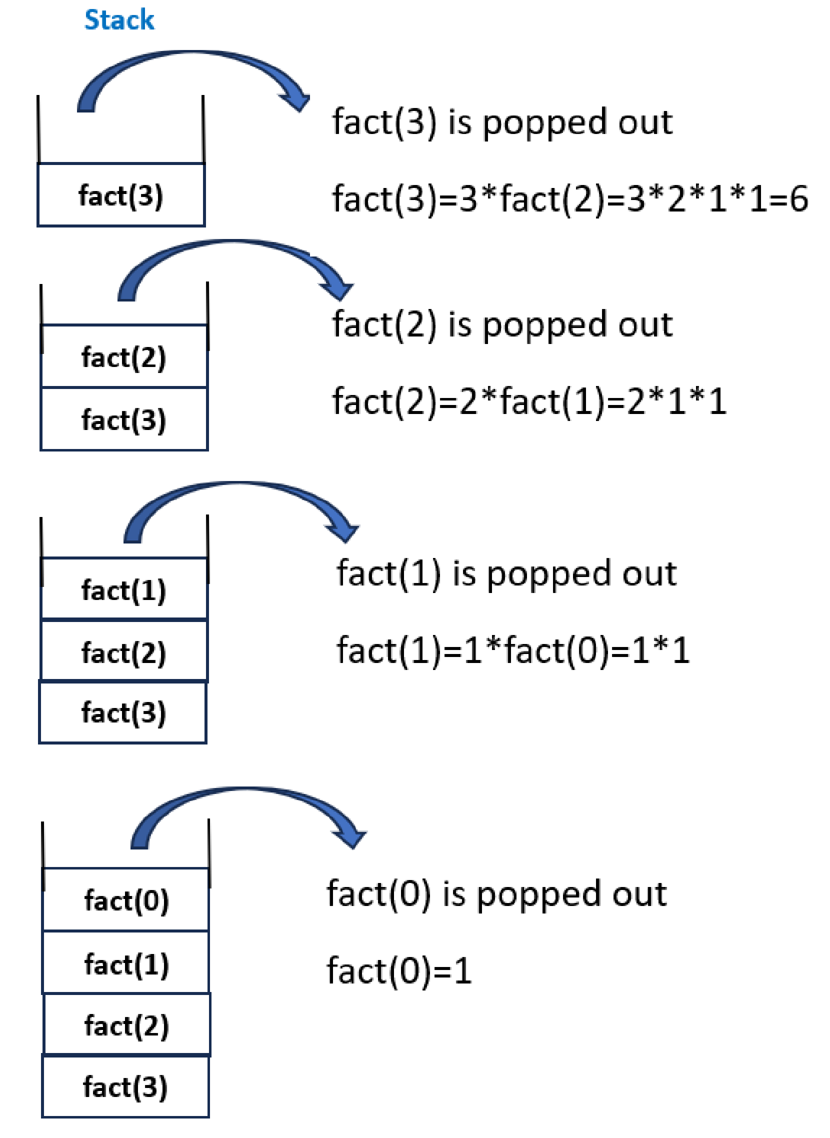
- Suppose we want to compute ***fact(3)*** the above function gets executed in the manner of Stack.

Functions - Recursion



- When the function call is made recursively, the activation record for each call will be placed on the top of the stack.
- Initially, fact(3) is called which recursively calls fact(2), fact(1), fact(0) and activation record gets inserted to top of the stack.

Functions - Recursion



Stack

fact(3)

fact(3) is popped out

fact(3)=3*fact(2)=3*2*1*1=6

fact(2)
fact(3)

fact(2) is popped out

fact(2)=2*fact(1)=2*1*1

fact(1)
fact(2)
fact(3)

fact(1) is popped out

fact(1)=1*fact(0)=1*1

fact(0)
fact(1)
fact(2)
fact(3)

fact(0) is popped out

fact(0)=1

- For n=0 i.e. base case(Termination Condition), the function call stops and fact(0) is popped out from the top of the stack.
- It returns 1, then the recursion backtracks and solve the pending function calls are popped out of the stack and fact(3) is computed.

Stack Memory Allocation

- In Python, function calls and the references are stored in **stack memory.**
- Allocation happens on contiguous blocks of memory – referred as ***Function call Stack***.
- The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack.
- Any local memory assignments such as variable initializations inside the particular functions are stored temporarily on the function call stack, where it is deleted once the function returns.
- This allocation onto a contiguous block of memory is handled by the compiler using predefined routines.

Example 1– Compute factorial of a number

```
def fact(n): #Recursive Function
        if n == 0 : #terminating condition
                res = 1
        else:
                res = n * fact(n - 1)
        return res
print(fact(5))
print(fact(0))
```

Output:

```
120
1
```

Example 2 – Compute GCD of two numbers

```
def gcd(m, n): #Recursive Function
        if m == n : #terminating condition
                res = m
        elif m > n :
                res = gcd(m -n, n)
        else:
                res = gcd(m, n - m)
        return res
print(“GCD : ", gcd(65, 91))
```

Output:

```
GCD : 13
```

Example 3 – Generate Fibonacci Series upto n_terms

```
def fib(n):    #Recursive Function
        if n <= 1: #terminating condition
            return n
        else:
            return(fib(n-1) + fib(n-2))

n_terms=int(input("Enter the number of terms for Fibonacci Series\n"))

for i in range(n_terms):
      print(fib(i))
```

Example 3 – Generate Fibonacci Series upto n_terms

Output:

```
Enter the number of terms for Fibonacci Series
5
0
1
1
2
3
```

Example 4 – Solving Tower of Hanoi Puzzle

```
def TowerOfHanoi(n , src, aux, dest): #Recursive Function
    if n==1: #terminating condition
        print ("Move disk 1 from ",src,"to ",dest)
        return
    TowerOfHanoi(n-1, src, dest, aux)
    print ("Move disk",n,"from ",src,"to ",dest)
    TowerOfHanoi(n-1, aux, src, dest)

n=int(input("Enter number of disks\n"))
print("A-Source  B-Auxiliary  C-Destination\n")
TowerOfHanoi(n,'A','B','C')
```

Example 4 – Solving Tower of Hanoi Puzzle

Output:

```
Enter number of disks
3
A-Source  B-Auxiliary  C-Destination

Move disk 1 from  A to  C
Move disk 2 from  A to  B
Move disk 1 from  C to  B
Move disk 3 from  A to  C
Move disk 1 from  B to  A
Move disk 2 from  B to  C
Move disk 1 from  A to  C
```

Example 5 – To add two numbers using recursion

```
def add(x, y): #Recursive Function
    if(y == 0):    #terminating condition
        return x
    return add(x, y - 1) + 1

print("Sum =", add(10, 20))
```

Output:

```
Sum = 30
```

Example 6 – To subtract two numbers using recursion

```
def subtract(x, y): #Recursive Function
    if(y == 0):          #terminating condition
        return x
    return subtract(x-1, y-1)

print("Result =", subtract(10, 20))
```

Output:

```
Result = -10
```

Example 7 – To multiply two numbers using recursion

```
def product(a,b): #Recursive Function
    if(a<b):
        return product(b,a)
    elif(b!=0):
        return(a+product(a,b-1))
    else:                 #Stopping point
        return 0

print("Product =",product(10,20))
```

Output:

```
Product = 200
```

Example 8 – To divide two numbers using recursion

```
def divide(x, y):   #Recursive Function
    if(x < y):            #terminating condition
        return 0
    else:
        return 1 + divide(x - y, y)

print("Result:", divide(20, 5))
```

Output:

```
Result: 4
```

Recursion : Function calling itself

Advantages of Recursion

- Recursive functions make the code look clean and elegant
- A complex task can be broken down into simpler sub-problems using recursion
- Sequence generation is easier with recursion than using some nested iteration

Recursion : Function calling itself

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time
- Recursive functions are hard to debug.

Recursion vs. Iteration

Recursion	Iteration
Function calls itself	Set of program statements executed repeatedly
Implemented using Function calls	Implemented using Loops
Termination condition is defined within the recursive function	Termination condition is defined in the definition of the loop
Leads to infinite recursion, if does not meet termination condition	Leads to infinite loop, if the condition in the loop never becomes false
It is slower than iteration	It is faster than recursion
Uses more memory than iteration	Uses less memory compared to recursion

Note: ***when a problem can be solved both recursively and iteratively with similar programming effort, it is generally best to use an iterative approach .***

Recursion : Summary

Recursion

- is a process in which a function calls itself directly or indirectly
- Using recursive algorithm, certain problems can be solved easily
- Additional care should be taken while designing recursive function otherwise it may lead to infinite calls

THANK YOU

Team Python - 2022

Department of Computer Science and Engineering

Contact Email ID's:

sindhurpai@pes.edu

sowmyashree@pes.edu

PES UNIVERSITY

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science
and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit - 3: Functions – Callback

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science and Engineering

Function: Callback

- A callback function is a function that is passed to another function as an argument.

- **It can be implemented by**
  Passing one function as an argument to another function

Function: Callback

Need for Callback in Functions

- Used in event-driven programming, where a function is called in response to a specific event or action, such as a button press or the completion of a network request.
- Also used in functional programming, where a function is passed as an argument to another function to be used as a "hook" for performing specific operations.
- Helps to separate functions’ functionality and make code more reusable and modular.

Function: Callback

Example 1 (using built-in function):

```
s=["Hello", "Welcome", "to", "python", "world"]
print(sorted(s))   #The list is sorted based on the ASCII values only.
print(sorted(s, key=str.upper))  #Sort the list based on only the uppercase form of each letter
```

Output

```
['Hello', 'Welcome', 'python', 'to', 'world’]
['Hello', 'python', 'to', 'Welcome', 'world’]
```

- **Illustration**

We are calling the str.upper function inside the sorted function. So, str.upper function is the callback function.

Function: Callback

Example 2 (using user-defined function):

```
def multiply(x):
   return num_list[0]*num_list[1]

def compute(func,x):
   return func(x)

num_list=[2,3]
product=compute(multiply,num_list)
print("Multiplication=",product)
```

Illustration

compute(multiply,num_list) – the caller function with 2 arguments,

1) a function,multiply and 2) a list,num_list

Here, multiply is the callback function.

Output

```
Multiplication= 6
```

Functions - Callback

Example 3 : Multiple Callback functions

```python
def function(func_list, x, y):
   print("Inside function")
   for func in func_list:
      func(x,y)

def add(x,y):
   z = x+y
   print('Sum =',z)

def divide(x,y):
   z = x/y
   print('Quotient =',z)

cb_list=[add, divide]
function(cb_list, 10, 5)
```

Output

```
Inside function
Sum = 15
Quotient = 2.0
```

Function: Callback

Advantages:

- Calling function(outer function) can call the callback function as many times as it required to complete the specified task.
- Calling function can pass appropriate parameters according to the task to the called functions. This allows information hiding.

THANK YOU

Department of Computer Science and Engineering

Contact Email ID's:

sindhurpai@pes.edu

sowmyashree@pes.edu

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai
Department of Computer Science and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit – 3: Functions - Closure

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science and Engineering

Functions - Closure

A closure is a nested function which has access to a free variable from an enclosing function that has finished its execution.

Three characteristics of a Python closure are:

- It is a nested function
- It has access to a free variable in outer scope
- It is returned from the enclosing function

A free variable - a variable that is not bound in the local scope.

Closures with immutable variables such as numbers and strings - use the nonlocal keyword.

Functions - Closure

1. Example:

```
def  outer(msg): # This is the outer enclosing function
   def inner():   # This is the nested function
      print(msg)
   return inner  # returns the nested function

# Now let's try calling this function.
different = outer(“This is an example of closure")
different () #refers to inner()
```

Output:

```
This is an example of closure
```

Functions - Closure

2. Example:

```
def f1(): #outer function
        def f2(): #inner function
                print ("Hello")
                print ("world")
        print('f2=',id(f2))
        return f2
c=f1() #refers to f2()
c()
print('c=',id(c)) #id of f2() and c() are same
```

Output:

```
f2= 2908823806840
Hello
world
c= 2908823806840
```

Functions - Closure

3. Example:

```
def division(y): #outer function
    def divide(x): #inner function
        return x/y
    return divide

d1=division(2) #refers to divide()
d2=division(3) #refers to divide()

print(d1(20))
print(d2(96))
```

Output:

```
10.0
32.0
```

4. **Example**:

```
def f1(): #outer function
    def f2(): #inner function
        print ("Hello")
        print ("world")
    return f2
c=f1() #refers to f2()
del f1
c() #still works
```

Output:

```
Hello
world
```

Functions - Closure

5. **Example**:

```
def outer(msg):
    text = msg        #text is having the scope of outer function
    def inner():
        print(text)    #using non-local variable text
    return inner      #return inner function


func = outer('Hello')
func()
```

Output:

```
Hello
```

Functions - Closure

6. **Example**:

```
def outerfunc(x):
        def innerfunc():
                print(x)
        return innerfunc

myfunc=outerfunc(7)
myfunc() #refers to innerfunc()
del outerfunc
myfunc() #still refers to innerfunc() retaining the value of enclosing scope of x
```

Output:

```
7
7
```

We are assigning the function outerfunc() to the variable myfunc. Even if we delete outerfunc() from the memory, the function outerfunc() can be called, using the referred variable myfunc.

7. **Example**:

```
def f1():          #outer function
    text="Python"
    def f2():      #inner function
        nonlocal text
        text="Hi"
        print(text)
    print(text)
    return f2

f = f1()
f()
```

Output:

```
Python
Hi
```

Note: In Python Closures, the inner function may access non-local variable but can’t modify it.

Functions - Closure

8. **Example**:

```
def f1():    #outer function
   x=0
   def f2(): #inner function
      nonlocal x     # x - that belongs to scope of outer function is made non-local
      x=x+1
      return x
   return f2

func = f1()
retval = func()
print ("x=", retval)
retval = func()
print ("x=", retval)
```

Output:

```
x= 1
x= 2
```

Function Closure vs. Nested function

- Not all nested functions are closures.
- For a nested function to be a closure, the following conditions need to be satisfied:

  1. The inner function has access to the non-local variables or local variables of the outer function.
  2. The outer function must return the inner function.

Function Closure vs. Nested function

Example: Nested Function but not Closure (When msg is passed to inner(), msg ends up belonging to inner() function's local scope. So, the 1st condition is not satisfied)

```
def  outer(msg):           # This is the outer enclosing function
    def inner(m=msg):    # This is the nested function
        print(m,"World")
    return inner              # returns the nested function

different = outer(msg="Hello")
different()                          #refers to inner()
```

Output

```
Hello World
```

Function Closure: Summary

- A function object that remembers values in enclosing scopes even when the variable goes out of scope.

- Python closures help avoiding the usage of global values and provide some form of data hiding.

  They are used in Python decorators.

PES UNIVERSITY

THANK YOU

Department of Computer Science and Engineering

Contact Email ID’s:

sindhurpai@pes.edu

sowmyashree@pes.edu

PES UNIVERSITY

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai
Department of Computer
Science
and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit - 3: Functions – Decorators

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science and Engineering

Functions - Decorators

- A powerful and useful tool in Python since it allows programmers to modify the behavior of function or class.

- Decorators **wrap a function and modify its behavior in one or the other way, without changing the source code** of the function being decorated.

- In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Function Decorators are used

- when we need to change the behavior of a function without modifying the function itself.
  Eg: logging, test performance, verify permissions and so on.

- when we need to run the same code on multiple functions. This avoids writing duplicating code.

Functions - Decorators

1. **Example**:

```
def func_decorator(func):
    def inner_func():
        print("Hello, before the function is called")
        func()
        print("Hello, after the function is called")
    return inner_func

def func_hello():
  print("Inside Hello function")

hello = func_decorator(func_hello)
hello()
```

Output:

```
Hello, before the function is called
Inside Hello function
Hello, after the function is called
```

Functions - Decorators

- ***func_decorator*** is the decorator function, accepts another function as an argument and "decorates it“.

- ***func_hello*** is an ordinary function that we need to decorate.

- ***inner_func*** is the wrapper function, that is actually decorating the ***func_hello*** function. In this example, all it does is print a simple statement before and after ***func_hello***.

The function decorator in the above example can also be implemented in other way. (See Example 2)
By using @ symbol

2. **Example (Same as Exampe1 with different format of Decorator):**

```
def func_decorator(func):
    def inner_func():
        print("Hello, before the function is called")
        func()
        print("Hello, after the function is called")
    return inner_func

@func_decorator
def func_hello():
    print("Inside Hello function")

func_hello()
```

Output:

```
Hello, before the function is called
Inside Hello function
Hello, after the function is called
```

3. **Example**

```
import math
def calculate(f):              #decorator function
          def inner1(*args):     #*args is variable length argument
                print("Decorator")
                f(*args)         # this is being decorated by decorator
                print("**************")
          return inner1

@calculate
def factorial(num):    #factorial() getting decorated
          print(math.factorial(num))
```

3. **Example (contd…)**

```
@calculate
def squareroot(num):     #squareroot() getting decorated
                    print(math.sqrt(num))

@calculate
def maximum(*num):     #maximum() getting decorated
                    print(max(num[0],num[1],num[2]))

factorial(5)                    #calls decorated factorial()
squareroot(16)                  #calls decorated sqrt1()
maximum(23,9,78)                #calls decorated maximum()
```

Functions - Decorators

Output:

```
Decorator
120
**************
Decorator
4.0
**************
Decorator
78
**************
```

Functions - Decorators

4. **Example**:

```
import math
def compute(func):          #decorator function
        def inner(a,b):
            print(“Computing hypotenuse")
            func(a,b)         # this is being decorated by decorator
            print("*****************")
        return inner

@compute
def hypotenuse(a, b):        # hypotenuse() is getting decorated
    h=math.sqrt(a*a+b*b)
    print(h)

hypotenuse(3,4)              #calls decorated hypotenuse
```

Functions - Decorators

Output:

```
Computing hypotenuse
5.0
*****************
```

Chaining Decorators - Decorating a function with multiple decorators.

```
def decorator_x(func):
   def inner_func():
      print("X"*20)      #Printing X 20 times
      func()
      print("X"*20)      #Printing X 20 times
   return inner_func

def decorator_y(func):
   def inner_func():
      print("Y"*20)      #Printing Y 20 times
      func()
      print("Y"*20)      #Printing Y 20 times
   return inner_func
```

Functions - Decorators

```
def func_hello():
  print("Hello")

hello = decorator_y(decorator_x(func_hello))       #Chaining Decorators
hello()
```

Output:

```
YYYYYYYYYYYYYYYYYYYY
XXXXXXXXXXXXXXXXXXXX
Hello
XXXXXXXXXXXXXXXXXXXX
YYYYYYYYYYYYYYYYYYYY
```

Functions - Decorators

Above Example can be implemented with different format of Decorators.

```
def decorator_x(func):
   def inner_func():
      print("X"*20)      #Printing X 20 times
      func()
      print("X"*20)      #Printing X 20 times
   return inner_func

def decorator_y(func):
   def inner_func():
      print("Y"*20)      #Printing Y 20 times
      func()
      print("Y"*20)      #Printing Y 20 times
   return inner_func
```

Functions - Decorators

```
@decorator_y         #Chaining Decorators
@decorator_x
def func_hello():
  print("Hello")

func_hello()
```

Output:

```
YYYYYYYYYYYYYYYYYYYY
XXXXXXXXXXXXXXXXXXXX
Hello
XXXXXXXXXXXXXXXXXXXX
YYYYYYYYYYYYYYYYYYYY
```

Functions Decorators: Summary

- A Decorator is just a function that takes another function as an argument and extends its behavior without explicitly modifying it.

- Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

- Using decorators, we can extend the features of different functions in a common way.

THANK YOU

Team Python - 2022

Department of Computer Science and Engineering

Contact Email ID's:

sindhurpai@pes.edu

sowmyashree@pes.edu

PES UNIVERSITY

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science
and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit - 3: Functions – Generators

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science and Engineering

Functions - Generators

- A generator is a function that **returns an iterator that produces a sequence of values** when iterated over.

- A way to create to declare a function that behaves like an iterator, providing a faster and easier way to create iterators.

- Does not return a single value, instead, it returns an iterator object with a sequence of values.

- A *yield* statement is used rather than *return* statement.

- If the body of a def contains *yield*, the function automatically becomes a Python generator function.

Functions - Generators

- **Syntax**

```
def generator_function_name(arg):
                  ………………………
                    ………………………
                     ………………………
                     yield statement
```

- When the generator function is called, it does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

Functions - Generators

Example 1: Simple generator function that will yield three integers (using for loop)

```
# Generator function
def generator_func():
    yield 1
    yield 2
    yield 3

# Code to check above generator function
for value in generator_func():
    print(value)
```

Output:

```
1
2
3
```

Generator Object

- Python Generator functions return a generator object that is iterable (used as an Iterator).

- Generator objects are used
  - by calling the next method of the generator object

    or
  - using the generator object in a “for” loop.

Functions - Generators

Example 2: Simple generator function that will yield three integers (using next() function)

```
# Generator function
def generator_func():
    yield 10
    yield 20
    yield 30

#obj is a generator object
obj=generator_func()

# Iterating over the generator object using next
print(next(obj))
print(next(obj))
print(next(obj))
```

Output:

```
10
20
30
```

Generator Expression

- Generator expression is another way of writing the generator function.

- Similar to List comprehension technique but instead of storing the elements in a list in memory, it creates generator objects.

- **Syntax**:

  (expression for element in iterable)

Generator Expression - Example

```
#Generator Expression
generator_exp=(i**2 for i in range(5) if i%2==0)

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

Output:

```
0
4
16
```

Pipelining Generators

Multiple generators can be used to pipeline a series of operations

Example: Compute the sum of squares of numbers in the Fibonacci series

```
# Generator function - fibonacci_numbers
def fibonacci_numbers(nums):
    x,y=0,1
    for i in range(nums):
        x,y=y,x+y
        yield x
# Generator function - square
def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci_numbers(3))))
```

Output:

```
6
```

Function Generators: yield vs. return

yield	return
Returns a value and pauses the execution while maintaining the internal states	Returns a value and terminates the execution of the function
Used to convert a regular Python function into a generator	Used to return the result to the caller statement
Used when the generator returns an intermediate result to the caller	Used when a function is ready to send a value
Code written after yield statement execute in next function call	Code written after return statement wont execute
It can run multiple times	It only runs single time

Note: We can't include return inside generator function. If we include, it will terminate the function.

Function Generators: Summary

- Python generator functions allow to declare a function that behaves like an iterator, making it a faster, cleaner and easier way to create an iterator.

- Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

- The simplification of code is a result of generator function and generator expression support provided by Python.

THANK YOU

Department of Computer Science and Engineering

Contact Email ID's:

sindhurpai@pes.edu

sowmyashree@pes.edu

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai
Department of Computer Science and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit – 3: Graphical User Interface with Tkinter package

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science and Engineering

Why do we need GUI?

- A user with no computer knowledge can literally start learning about the machine because of GUI as it provides scope for users to explore and provides discoverability.

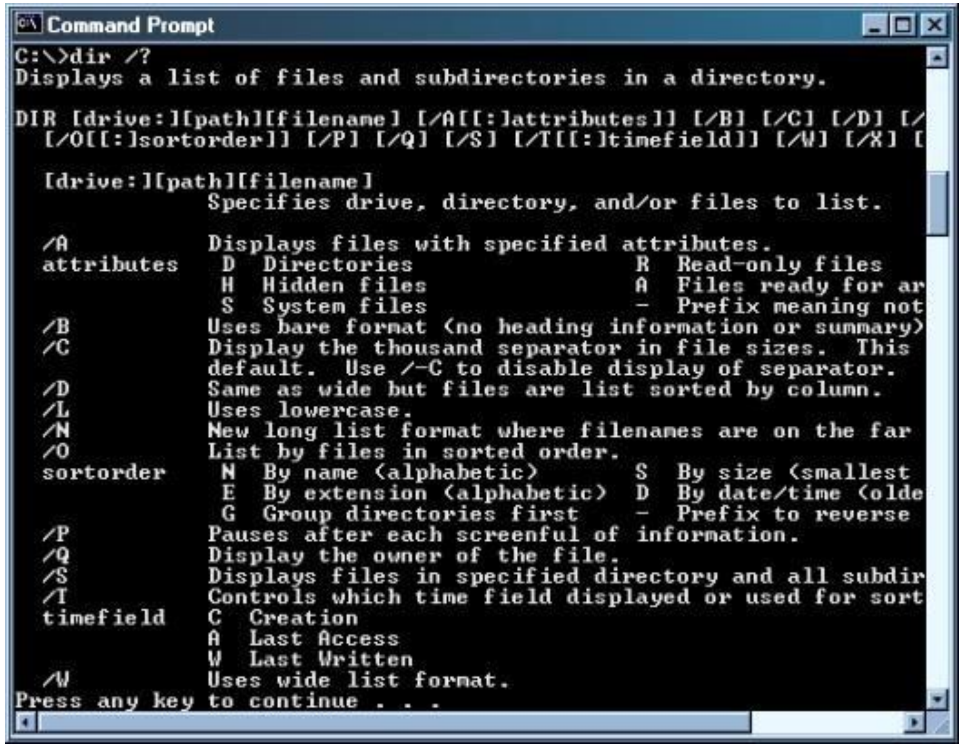
- For example, a user starts using a computer with no Interface, then he/she has to provide commands to the machine to execute each task. In a way, the user must have some kind of programming knowledge.

- In real time, you consider the system used in Retail store for billing purpose for command line interface and voting portal for GUI interface.

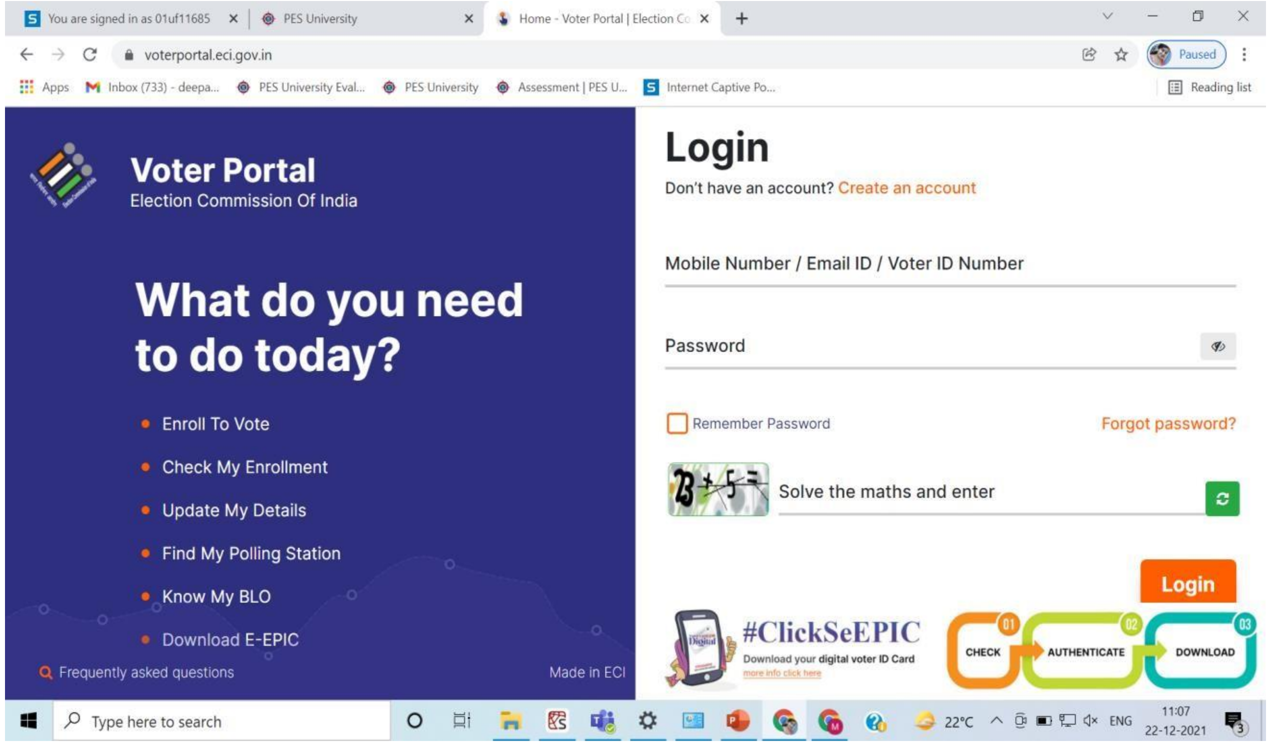
PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

GUI - Tkinter

Observe the below screenshot, to get clear picture of Command Line Interface and Graphical User Interface.



```
C:\>dir /?
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] [/A[[:]attributes]] [/B] [/C] [/D] [/
  [/O[[:]sortorder]] [/P] [/Q] [/S] [/T[[:]timefield]] [/W] [/X] [

  [drive:][path][filename]
              Specifies drive, directory, and/or files to list.

  /A          Displays files with specified attributes.
  attributes   D  Directories                R  Read-only files
               H  Hidden files               A  Files ready for ar
               S  System files               -  Prefix meaning not
  /B          Uses bare format (no heading information or summary)
  /C          Display the thousand separator in file sizes.  This
              default.  Use /-C to disable display of separator.
  /D          Same as wide but files are list sorted by column.
  /L          Uses lowercase.
  /N          New long list format where filenames are on the far
  /O          List by files in sorted order.
  sortorder    N  By name (alphabetic)       S  By size (smallest
               E  By extension (alphabetic)  D  By date/time (olde
               G  Group directories first    -  Prefix to reverse
  /P          Pauses after each screenful of information.
  /Q          Display the owner of the file.
  /S          Displays files in specified directory and all subdir
  /T          Controls which time field displayed or used for sort
  timefield   C  Creation
              A  Last Access
              W  Last Written
  /W          Uses wide list format.
Press any key to continue . . .
```



Voter Portal
Election Commission Of India

What do you need to do today?

- Enroll To Vote
- Check My Enrollment
- Update My Details
- Find My Polling Station
- Know My BLO
- Download E-EPIC

Frequently asked questions

Made in ECI

Login

Don't have an account? Create an account

Mobile Number / Email ID / Voter ID Number

Password

□ Remember Password

Forgot password?

Solve the maths and enter

Login

#ClickSeEPIC

Download your digital voter ID Card

Popular Python GUI frameworks

1. Tkinter
2. Qt for Python: PySide2 / Qt5
3. PySimpleGUI
4. PyGUI
5. Kivy
6. wxPython
7. Libavg
8. PyForms
9. Wax
10. PyGTK

Tkinter

- Built into the Python standard library

- It’s cross-platform, so the same code works on Windows, macOS, and Linux

- Lightweight and relatively easy to use compared to other frameworks

Tkinter is used

1. To create Windows and Dialog boxes
2. To build a GUI for Desktop Applications
3. To add a GUI to Command-Line Program
4. To create custom Widgets
5. In Prototyping a GUI

GUI - Tkinter

Let us learn how to create a GUI application in Python using Tkinter.

Import Tkinter package:

```
import tkinter
```

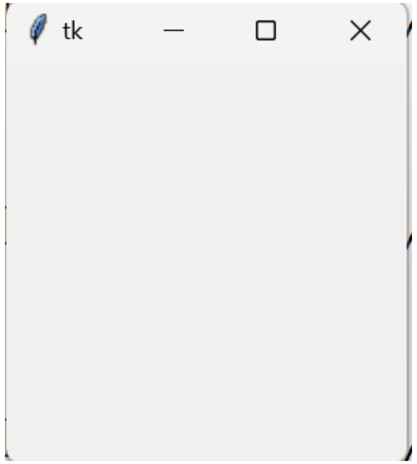
To create a Window

Step 1: Import tkinter package

Step 2: root=tkinter.Tk()

Step 3: root.mainloop()

Output



```
import tkinter
root = tkinter.Tk()        #creates window
root.mainloop()            #loops continuously until we close the  window
```

mainloop()

- A function that continuously loops and displays the window till we close it or an action closes the window.
- It will loop forever, waiting for events from the user, until the user exits the program (either by closing the window, or by terminating the program with a keyboard interrupt in the console)
- All windows that are created, work on this concept of constant looping to keep track of the interactions of the user with the Interface.
- It can track the movements of the mouse on the window because it constantly loops and has knowledge of where the mouse pointer is on the window at every frame.

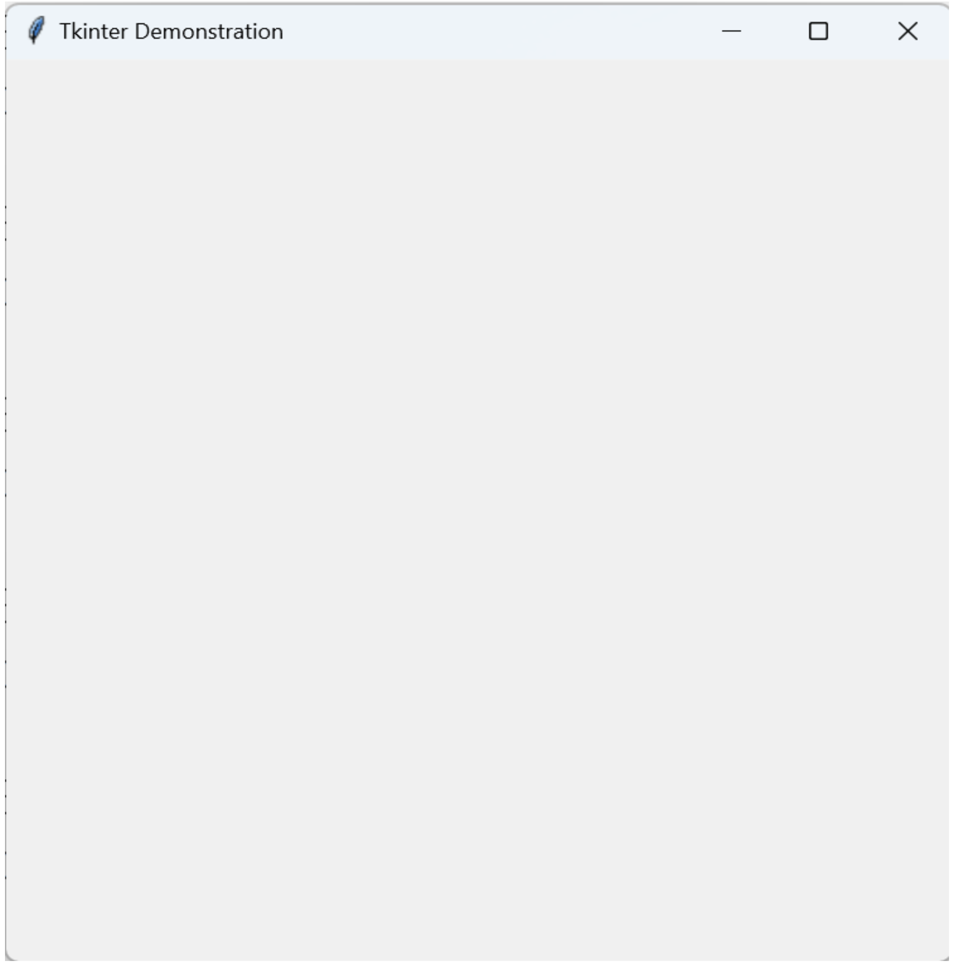
Adding title and geometry to the Window

root.title(Title Name)
root.geometry(Dimension in widthxheight)

Example:

```
import tkinter
root = tkinter.Tk()   #creates window
root.title("Tkinter Demonstration")   #Title
root.geometry('500x500’)  #Dimension
root.mainloop()
```

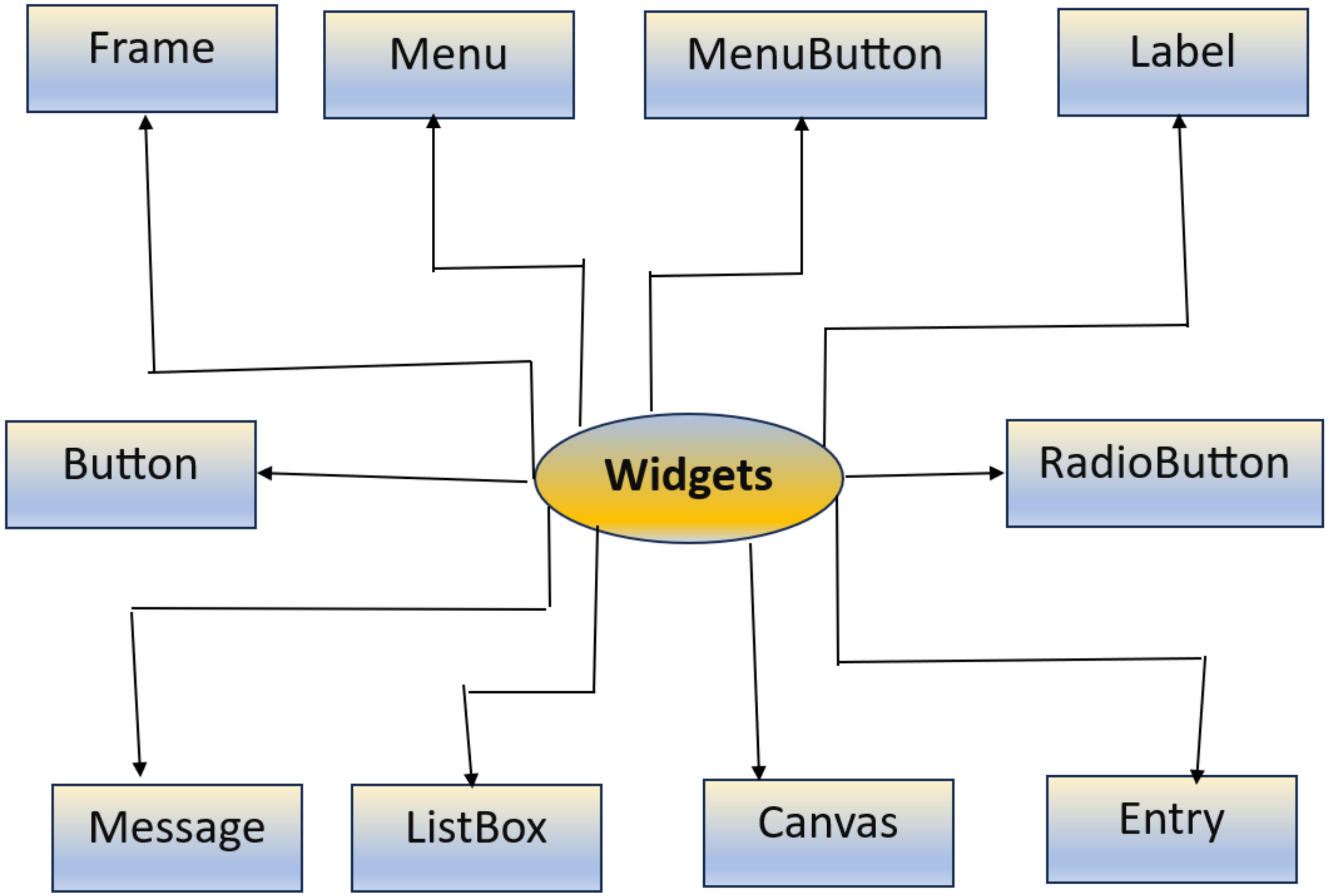
Output



Widgets

- After creating window, we need to add elements to make it more interactive.
- Each element in Tkinter is Widget.
- In Tkinter , Widgets are objects.
- Each separate widget is a Python object.
- When creating a widget, we must pass its parent as a parameter to the widget creation function.
- Except “root” window, which is the top-level window that will contain everything else and it does not have a parent.

GUI - Tkinter


Frame
Menu
MenuButton
Label
Button
Widgets
RadioButton
Message
ListBox
Canvas
Entry

GUI - Tkinter

Widget Name	Description
Button	To add a button to the application
Canvas	To draw a complex layout and pictures (like graphics, text, etc.)
CheckButton	To display a number of options as checkboxes
Entry	To display a single-line text field that accepts values from the user
Frame	To group and organize other widgets
Label	To Provide a single-line caption, can contain images also.
Listbox	To provide a user with a list of options
Menu	Creates all kinds of Menus required in the application
Menubutton	To display the menu items to the user

GUI - Tkinter

Widget Name	Description
Message	Displays a message box to the user
Radiobutton	Number of options to be displayed as radio buttons
Scale	A graphical slider that allows to select values from the scale
Scrollbar	To scroll the window up and down
Text	A multi-line text field to the user where users enter or edit the text and it is different from Entry
Toplevel	Used to provide a separate window container
Spinbox	An entry to the "Entry widget" in which value can be input just by selecting a fixed value of numbers
PanedWindow	A container widget that is mainly used to handle different panes
MessageBox	Used to display messages in desktop applications

Widgets

- Steps to add widget to the Window
  1. Create widget
  2. Add it to the Window

- Creating a new widget doesn't mean that it will appear on the screen. To display it, we need to call a special method: either **grid, pack**, or **place**.

1. **pack()** - packs widgets in rows or columns
2. **grid()** - puts the widgets in a 2-dimensional table. The master widget is split into a number of rows and columns, and each "cell" in the resulting table can hold a widget.
3. **place()** - explicitly set the position and size of a window, either in absolute terms, or relative to another window.

Button Widget - To add a button to the application

- **Syntax**

  w=Button(parent,options)
- parent – parent window
- options – to change look of the buttons, written as comma-separated

Button widget options

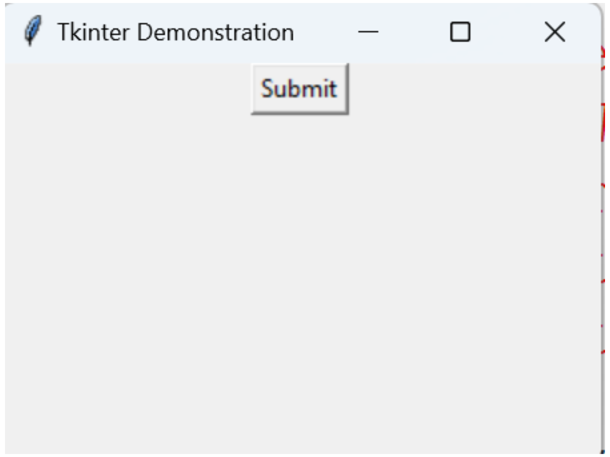
activebackground – background of button when the mouse hovers the button
activeforeground – represents the font color when the mouse hovers the button
bd – width of the border
bg – background color of button
fg – foreground colot of button
height – height of button
justify – with 3 values, LEFT, RIGHT, CENTER
underline – underline the text of button
width – width of the button

Button Widget

Example1

```
from tkinter import *
win =Tk()
win.title("Tkinter Demonstration")
win.geometry('300x200')
b=Button(win, text='Submit')
b.pack()
win.mainloop()
```

Output



Button Widget

Example2

```
import tkinter
from tkinter import *
from tkinter import messagebox

win = Tk()
win.title("Tkinter Button Widget Demonstration")
win.geometry('300x200')

def click():
   messagebox.showinfo("Message", "Green Button clicked")

a=Button(win, text="yellow", activeforeground="yellow", activebackground="orange", pady=10)
```

Button Widget

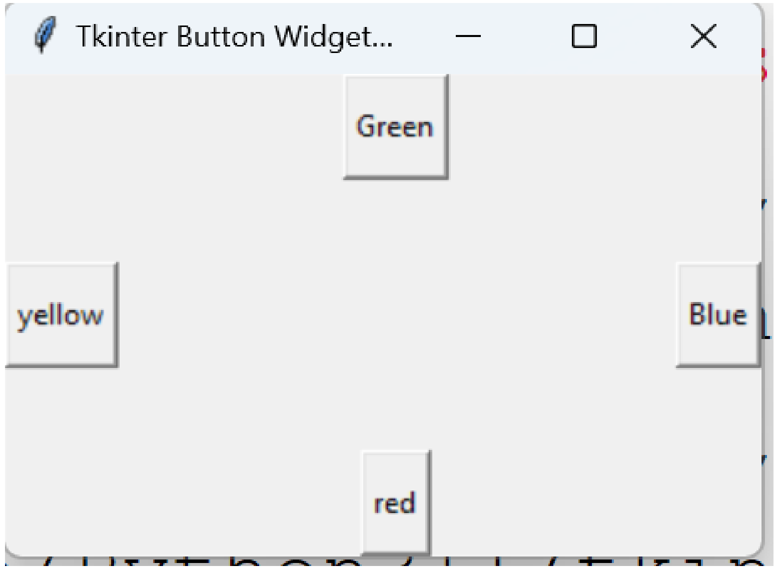
Example2 (Contd…)

```
b=Button(win, text="Blue", activeforeground="blue", activebackground="orange", pady=10)
# adding click function to the below button
c=Button(win, text="Green", command=click, activeforeground = "green",
activebackground="orange", pady=10)
d=Button(win, text="red", activeforeground="red", activebackground="orange", pady=10)
a.pack(side=LEFT)
b.pack(side=RIGHT)
c.pack(side=TOP)
d.pack(side=BOTTOM)
win.mainloop()
```

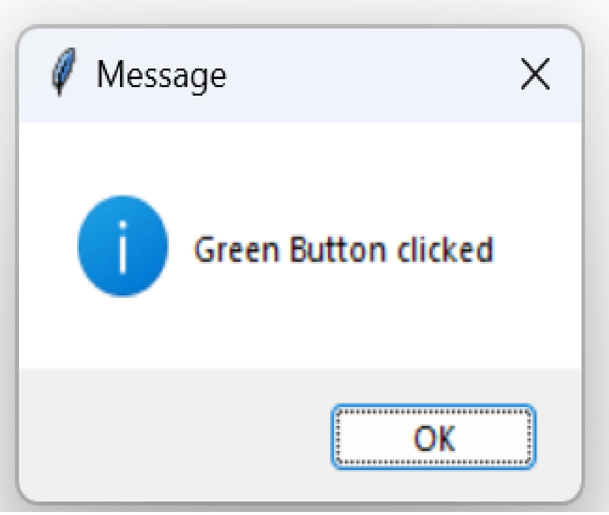
Button Widget

Example2 (Contd...)

Output



After clicking Green button, Messagebox appears.



Green Button clicked

Canvas Widget - used to draw anything on the application window

- **Syntax**

  w=Canvas(parent,option=value)
- parent – parent window
- option – to change layout of the canvas, written as comma-separated-Key-values.

Canvas widget options

bd – width of the border

bg – background color

cursor – to use arrow, dot, or circle

height – height of canvas

xscrollcommand – horizontal scrollbar

yscrollcommand – vertical scrollbar

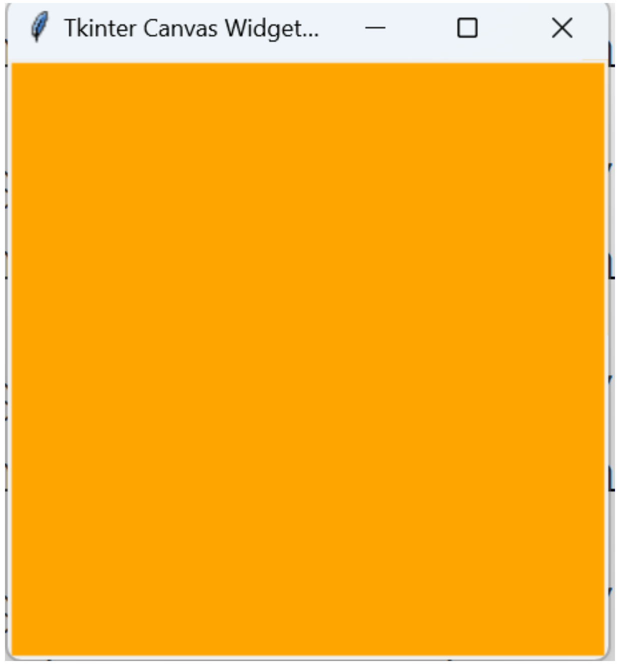
confine – non-scrollable outside the scroll region

Canvas Widget

Example1

```
from tkinter import *

win=Tk()
win.title("Tkinter Canvas Widget Demonstration")
win.geometry("300x300")

#creating canvas
cv=Canvas(win, bg = "orange", height = "300")

cv.pack()

win.mainloop()
```

Output



Canvas Widget

Example2

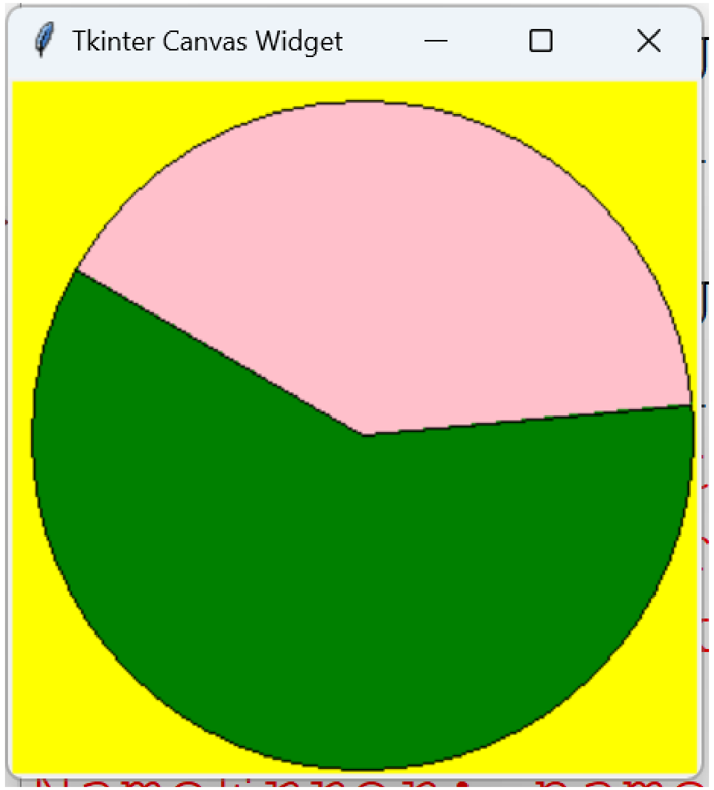
```
import tkinter

win=tkinter.Tk()
win.title("Tkinter Canvas Widget")

# creating canvas
cv=tkinter.Canvas(win, bg="yellow", height=300, width=300)

# drawing two arcs
coord = 10, 10, 300, 300
arc1=cv.create_arc(coord, start=0, extent=150, fill="pink")
arc2=cv.create_arc(coord, start=150, extent=215, fill="green")
```

Canvas Widget

Example2 (Contd…)

```
# adding canvas to window and display it
cv.pack()
win.mainloop()
```

Output



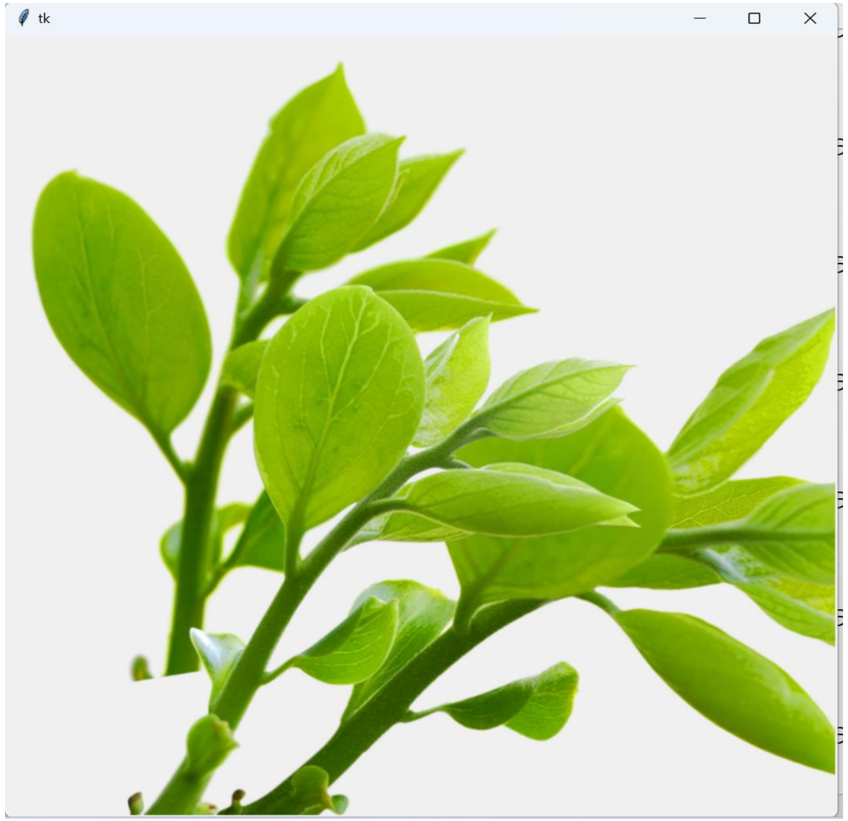
Canvas Widget

Example3

```
from tkinter import *

win=Tk()

cv=Canvas(win, height=700, width=700)
filename=PhotoImage(file="nature.png")

image=cv.create_image(20, 20, anchor=NW, image=filename)

cv.pack()
win.mainloop()
```

Canvas Widget

Example3 - Output



Checkbutton Widget - button to select from multiple options

- **Syntax**

  w= Checkbutton(parent,option=value)
- parent – parent window
- option – to configure checkbutton, written as comma-separated-Key-value pair.

Checkbutton widget options

bd – width of the border
bg – background color of button
bitmap – to display image in the button
command – function to be called on checking the button
height – height of widget
image – display generic image on the button
justify – with 3 values, LEFT, RIGHT, CENTER
padx – space to leave to the left and right of the checkbutton and text. Default value is 1 pixel
pady – space to leave to the above and below the checkbutton and text. Default value is 1 pixel

Checkbutton Widget

- **Functions**

1. deselect(): to turn off the checkbutton
2. flash(): The checkbutton is flashed between the active and normal colors.
3. invoke(): invoke the method associated with the checkbutton.
4. select(): to turn on the checkbutton.
5. toggle(): to toggle between the different Checkbuttons.

Checkbutton Widget

Example

```
from tkinter import *

win=Tk()
win.geometry("300x300")

w=Label(win, text ='Select Your Hobbies:', fg="Blue",font = "100")
w.pack()

Checkbutton1 = IntVar() # holds integer data passed to the checkbutton widget
Checkbutton2 = IntVar()
Checkbutton3 = IntVar()
```

Checkbutton Widget

Example (Contd...)

```
cb1=Checkbutton(win, text="Painting",variable = Checkbutton1,
                                                onvalue = 1,
                                               offvalue = 0,
                                               height = 2,
                                               width = 10)

cb2=Checkbutton(win, text = "Dancing", variable = Checkbutton2,
                                               onvalue = 1,
                                               offvalue = 0,
                                               height = 2,
                                               width = 10)
```

Checkbutton Widget

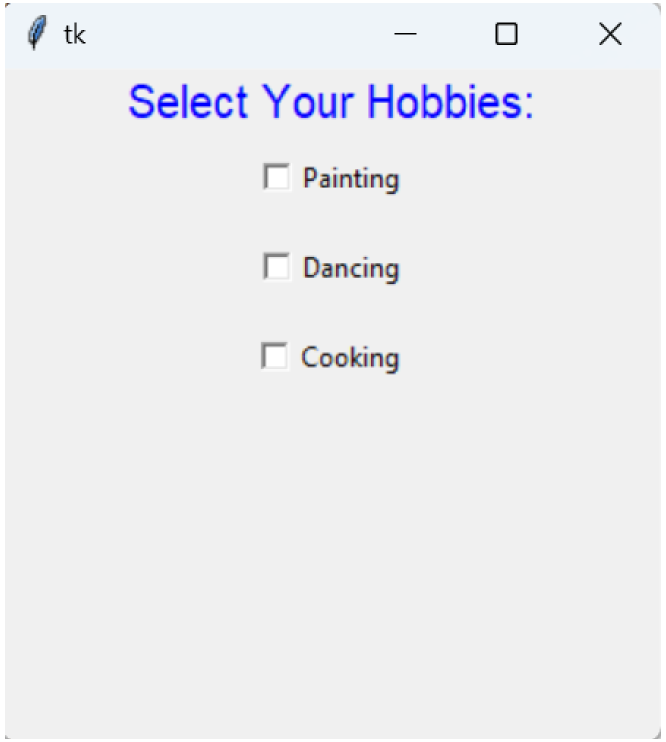
Example (Contd…)

```
cb3=Checkbutton(win, text = "Cooking", variable = Checkbutton3,
                                        onvalue = 1,
                                        offvalue = 0,
                                        height = 2,
                                        width = 10)


cb1.pack()
cb2.pack()
cb3.pack()

mainloop()
```

Checkbutton Widget

Example (Contd...)

Output



Select Your Hobbies:

□ Painting

□ Dancing

□ Cooking

Label Widget - to provide a message about the other widgets

- **Syntax**

  w= Label(parent,options)
- parent – parent window
- option – to configure the text, written as comma-separated-Key-value pair.

Label widget options

anchor – to control the position of widget
bg – background color of widget
bitmap – to set the bitmap equals to the graphical object
cursor – type of cursor to show when the mouse is moved over the label
height – height of widget
image – indicates the image that is shown as label
justify – with 3 values, LEFT, RIGHT, CENTER
padx – Horizontal padding of text. Default value is 1.
pady – Vertical padding of text. Default value is 1.

Label Widget

Example

```
from tkinter import *

win=Tk()

win.geometry("400x250")


username=Label(win, text = "Username").place(x = 30,y = 50)

password=Label(win, text = "Password").place(x = 30, y = 90)
```

Label Widget

Example

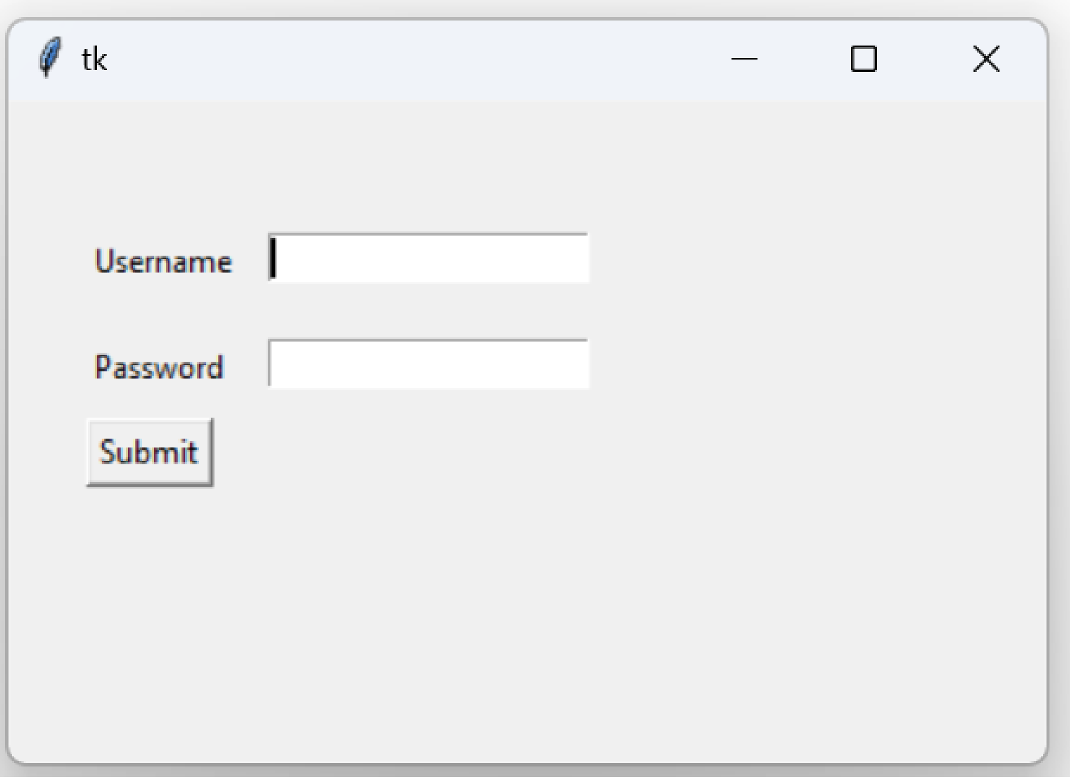
```
submitbutton=Button(win, text = "Submit",activebackground = "red", activeforeground =
"blue").place(x = 30, y = 120)

e1=Entry(win,width = 20).place(x = 100, y = 50)
e2=Entry(win, width = 20).place(x = 100, y = 90)

win.mainloop()
```

GUI - Tkinter

Label Widget

Example

Output



Entry Widget - to enter or display single line of text

- **Syntax**
  w= Entry(parent,options)
- parent – parent window
- option – to configure the entry, written as comma-separated values.

Entry widget options

bg – background color of widget

font – font used for the text

fg – color to render the text

relief – default value, relief=FLAT. Other styles are : SUNKEN, RIGID, RAISED, GROOVE

show –to show the text while making an entry, Eg: for Password set Show="*"

textvariable – to retrieve the current text from your entry widget

Entry Widget

- **Functions**

1. get(): Returns the entry’s current text as a string
2. delete(): Deletes characters from the widget
3. insert(index,name): Inserts string ‘name’ before the character at the given index

Entry Widget

Example

```
from tkinter import *

win=Tk()

win.geometry("400x250")

name=Label(win, text = "Name").place(x = 30,y = 50)
email=Label(win, text = "Email").place(x = 30, y = 90)
password=Label(win, text = "Password").place(x = 30, y = 130)
```

Entry Widget

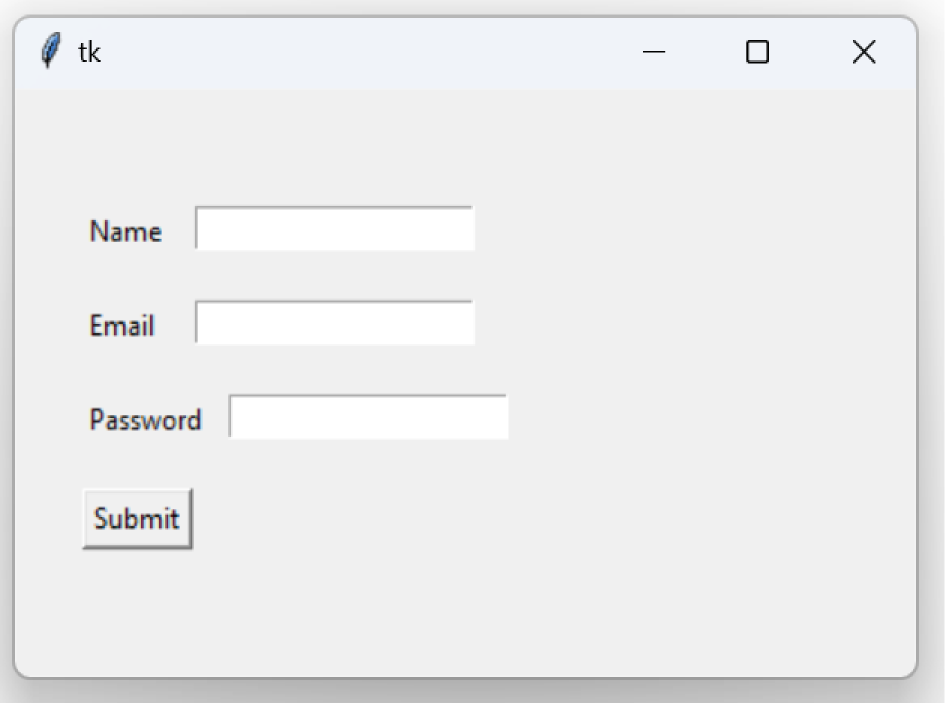
Example (Contd…)

```
submitbtn=Button(win, text = "Submit",activebackground = "red", activeforeground = "blue").place(x = 30, y = 170)

entry1=Entry(win).place(x = 80, y = 50)
entry2=Entry(win).place(x = 80, y = 90)
entry3=Entry(win).place(x = 95, y = 130)

win.mainloop()
```

Entry Widget

Example (Contd…)

Output



Dialogs in Tkinter

- A window which is used to "talk" to the application
- Used to input data, modify data, change the application settings etc.
- Communication between a user and a computer program

Tkinter Message Box Dialog

- Provide messages to the user of the application
- Message consists of text and image data
- Located in tkMessagebox module
- By using the message box library several Information is displayed, such as Error, Warning, Cancellation etc.

Message Box

- **Syntax**

  messagebox.function_name(Title, Message, [,options])

- function_name – Name of the function we want to use
- Title – Message box’s Title
- Message – Message to be shown on the dialog
- options – to Configure the options

function_name

Name of the function	Significance
showinfo()	To display some important information
showwarning()	To display some type of Warning
showerror()	To display some Error Message
askquestion()	To display a dialog box that asks with two options YES or NO
askokcancel()	To display a dialog box that asks with two options OK or CANCEL
askretrycancel()	To display a dialog box that asks with two options RETRY or CANCEL
askyesnocancel()	To display a dialog box that asks with three options YES or NO or CANCEL

Messagebox – askquestion()

Example1

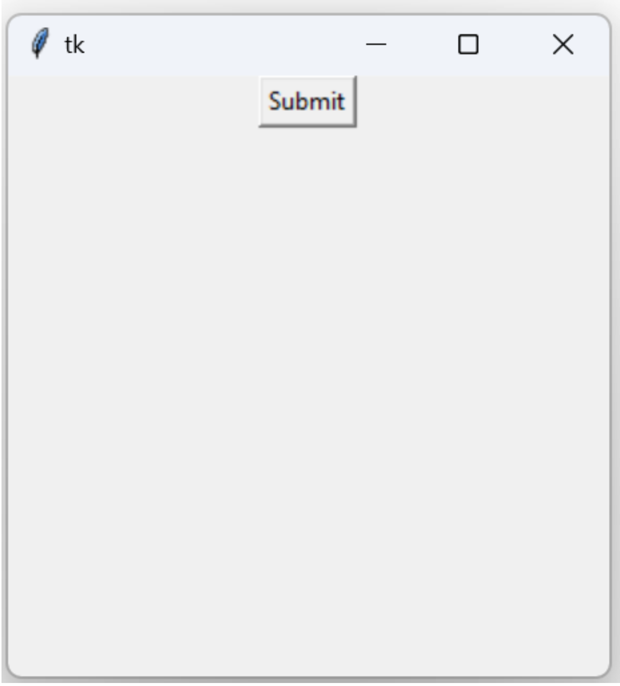
```
from tkinter import *
from tkinter import messagebox

win=Tk()

# function to use the askquestion() function
def Submit():
    messagebox.askquestion("Form", "Do you want to Submit")

win.geometry("300x300")
```

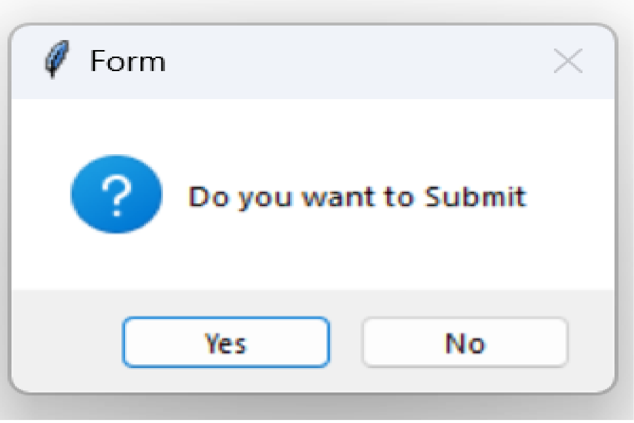
Example1 (Contd…)

```
# creating Submit Button
b=Button(win, text = "Submit", command = Submit)
b.pack()

win.mainloop()
```

Example1 (Contd...)

Output





Do you want to Submit

After clicking Submit button in the 1st window, message box is displayed.

Frame widget in Tkinter

- A frame - rectangular region on the screen.
- Used to implement complex widgets.
- Organize a group of widgets.
- **Syntax**

```
w=frame(parent,options)
```

- parent – parent window
- options – to configure frames, written as comma-separated-Key-value pair.

Frame widget options

bg – background color displayed behind the label and indicator
bd – border size, default is 2 pixels
cursor – to change the mouse cursor pattern
height – vertical dimension of new frame
highlightcolor – color of focus highlight when the frame has focus
highlightthickness – color the focus when the frame does not have the focus
highlightbackground – thickness of focus highlight
relief – type of the border of the frame. default =FLAT
width – width of the frame

Frame widget

Example1

```
from tkinter import *

win = Tk()
win.geometry("300x150")

w=Label(win, text ='Frame Demonstration', font = "50")
w.pack()

frame=Frame(win)
frame.pack()
```

Example1 (Contd...)

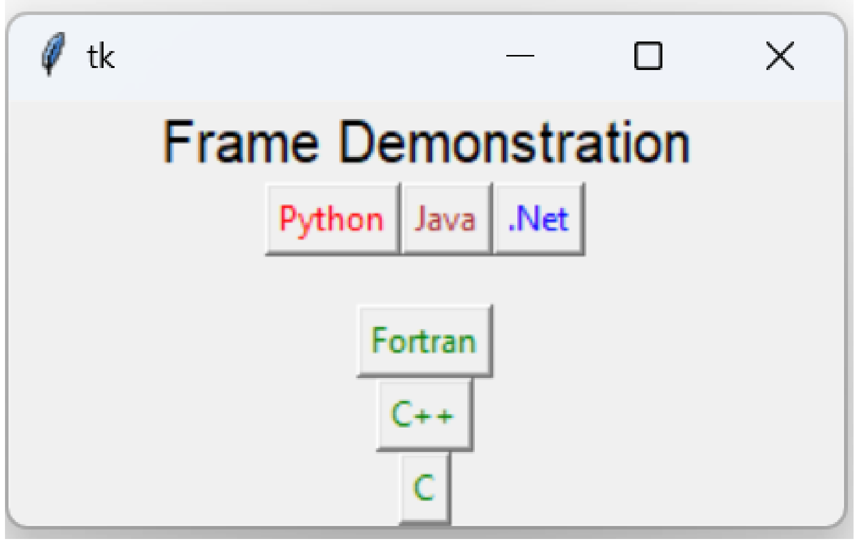
```
bottomframe=Frame(win)
bottomframe.pack( side = BOTTOM )

b1= Button(frame, text ="Python", fg ="red")
b1.pack( side = LEFT)

b2 = Button(frame, text ="Java", fg ="brown")
b2.pack( side = LEFT )

b3 = Button(frame, text =".Net", fg ="blue")
b3.pack( side = LEFT )
```

Example1 (Contd…)

```
b4 = Button(bottomframe, text ="C", fg ="green")
b4.pack( side = BOTTOM)

b5 = Button(bottomframe, text ="C++", fg ="green")
b5.pack( side = BOTTOM)

b6 = Button(bottomframe, text ="Fortran", fg ="green")
b6.pack( side = BOTTOM)

win.mainloop()
```

Example1 (Contd...)

Output



Frame widget –Nested Frames

- A frame within another frame
- Steps to create Nested Frames
  1. Create normal Tkinter window
  2. Create 1st Frame
  3. Create 2nd Frame
  4. Take the 1st frame as parent for 2nd Frame
  5. Execute code
- Syntax

  frame(parent)

Frame widget – Nested Frames

Example 4

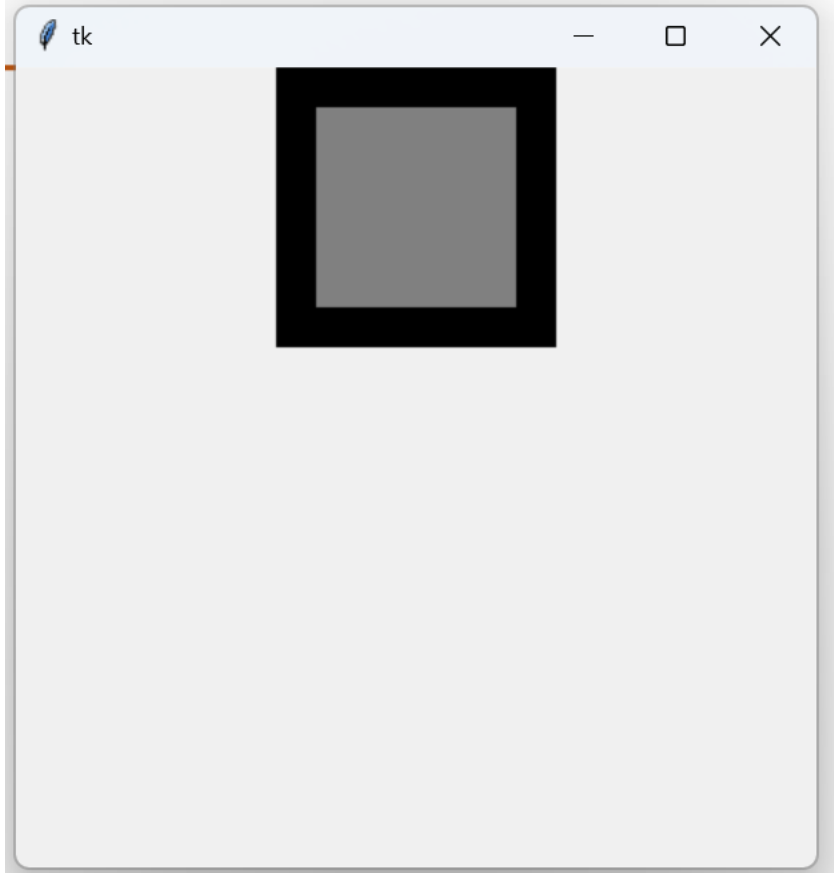
```
from tkinter import *

win=Tk()
win.geometry("400x400")

# Frame 1
frame1=Frame(win,bg="black",width=500,height=300)
frame1.pack()
```

Example 4 (Contd…)

```
# Frame 2 is created within Frame 1
frame2=Frame(frame1,bg="Grey",width=100,height=100)
frame2.pack(pady=20,padx=20)

win.mainloop()
```

Example 4 (Contd...)

Output



GUI - Tkinter

Explore on:

- Tkinter Color Chooser Dialog - colorchooser – askcolor()
- Tkinter file dialog - filedialog – askopenfile()
- Frame widget – Change width
- Frame widget – Change Color

THANK YOU

Department of Computer Science and Engineering

Contact Email ID’s:

sindhurpai@pes.edu

sowmyashree@pes.edu

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai
Department of Computer Science
and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit - 3: Modules – Import Mechanisms

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science and Engineering

Modules – Import mechanisms

Module

- In Python, a module is a file that contains code.
- It can include functions, classes, variables or any runnable code.
- Basically, a module contains code to perform specific task.
- We can use modules to separate codes in separate files as per their functionality.

Modules – Import mechanisms

Advantages of Modules

- Reusability - makes the code reusable
- Modularity – Organizing the code into modules logically
- Separate scopes – separate namespace is defined by the module
- Grouping - Python modules help us to organize and group the content by using files and folders

Modules – Import mechanisms

Need for Modules

- While working on coding, We need to use many classes, variables, functions.
- If we include everything in a single file, the program may become large.
- To reduce size of the code, we can group together some similar functions, classes into a collection.
- That collection is nothing but modules in python.

Packages and Namespace

- Modules in Python can be grouped together in packages.
- Packages organize the code into logical groups and provide a namespace to the modules so that they don't conflict with modules with the same name in other packages.
- The import statement can also be used to import modules from a package, which allows to access the functions and classes defined in the package's modules.

Packages vs. Modules vs. Libraries

- Modules – contain several functions, variables, classes etc.
- Packages – contain several modules.

  - folder that contains various modules as files.
- Library - a collection of packages and modules used to access **built-in functionality**

Modules – Import mechanisms

Types of Modules

1. **Built-in Modules**
   - Python's standard library comes bundled with a large number of modules.
   - They are called built-in modules.

2. **User-defined Modules**
   - Any file with .py extension and containing Python code is basically a module.
   - It can contain definitions of one or more functions, variables, constants ,classes.

Modules – Import mechanisms

Built-in Modules

Name	Description
os	provides a unified interface to a number of operating system functions
string	contains a number of functions for string processing
re	regular expression functionalities
math	a number of mathematical operations
cmath	a number of mathematical operations for complex numbers
datetime	functions to deal with dates and the time
gc	an interface to the built-in garbage collector
asyncio	functionality required for asynchronous processing
collections	advanced Container datatypes
functools	Higher-order functions and operations on callable objects

Modules – Import mechanisms

Built-in Modules

Name	Description
operator	Functions on the standard operators
pickle	Convert Python objects to streams of bytes and back
socket	Low-level networking interface
sqlite3	A DB-API 2.0 implementation using SQLite 3.x
statistics	Mathematical statistics functions
typing	Support for type hints
venv	Creation of virtual environments
json	Encode and decode the JSON format
unittest	Unit testing framework for Python
random	Generate pseudo-random numbers

Modules – Import mechanisms

User-defined Modules

- Any file with .py extension and containing Python code is a module.
- It can contain definitions of one or more functions, variables, constants as well as classes.
- Any object from a module can be made available to interpreter or another Python script by using import statement.

Creating a Module

- Crete a file with .py extension
- **Example**: Creating a module(module1.py)

```
a=10
print("Welcome to Module-1")
def f1():
        print("in f1")
def f2():
        print("in f2")
def _f3():
        print("in f3")
```

module1.py

Import a Module

- import the functions, and classes defined in a module to another module
- When the interpreter encounters an import statement, it imports the module if the module is present. Otherwise, *ModuleNotFoundError* is thrown.
- Syntax - **import** *module_name*

  import – the keyword used to import the module

  module_name – name of the module to be imported

- To access the functions inside the module the dot(.) operator is used.

Import a Module

- **Example**: importing the **module1**(refer previous slide)

```
import module1 #All functions and variables of module1 are available

module1.f1()
module1.f2()
module1._f3()

print("value is", module1.a)
module1.a = module1.a + 2
print("value is", module1.a)
```

← **usingModule1.py**

Modules – Import mechanisms

Output

- Executing **usingModule1.py**

```
Welcome to Module-1
in f1
in f2
in f3
value is 10
value is 12
```

Modules – Import mechanisms

Import from a Module

- Import Specific Attributes from a module
- Syntax – **from** *module_name* **import** *specific_attributes*
- **Example 2**

```
def add(x, y):
   return (x+y)
def subtract(x, y):
   return (x-y)
def multiply(x, y):
   return (x*y)
def divide(x, y):
   return (x/y)          #Assuming 'y' is never zero
```

module2.py

Modules – Import mechanisms

Import from a Module

- <u>Example 2 (Contd…)</u>

```
from module2 import add,multiply
#importing only add and multiply functions from module2

print("Sum=",add(10,20))
print("Product=",multiply(25,10))
```

usingmodule2.py

<u>Output</u>

```
Sum= 30
Product= 250
```

Modules – Import mechanisms

Import all Names from a Module

- * symbol with the import statement is used to import all the names from a module.
- Syntax - **from** *module_name* **import** *
- **Example 3**

```
def add(x, y):
   return (x+y)
def subtract(x, y):
   return (x-y)
def multiply(x, y):
   return (x*y)
def divide(x, y):
   return (x/y)          #Assuming 'y' is never zero
```

module3.py

Modules – Import mechanisms

Import all Names from a Module

- **Example 3 (Contd…)**

```
from module3 import *
#importing all the functions from module2

print("Sum=",add(10,20))
print("Product=",multiply(25,10))
```

usingmodule3.py

Output

```
Sum= 30
Product= 250
```

Note: If we know exactly which attribute to import from the module, it is not recommended to use import *

Renaming/Aliasing Python Modules

- We can rename the module while importing it.
- Syntax – **import** *module_name* **as** *alias_name*
- **Example 4**

```
import math as mt  #Renaming math module as 'mt'

print(mt.factorial(6))
```

Output

```
720
```

Modules – Import mechanisms

Renaming/Aliasing Python Modules

- **Example 5**

```
GRAVITY=9.8
print("Illustration of Renaming a Module")
```

module5.py

```
import module5 as m5
print("***********************************")
print("Acceleration due to gravity on earth=",m5.GRAVITY,"m/s\u00b2")
```

usingmodule5.py

Output

```
Illustration of Renaming a Module
***********************************

Acceleration due to gravity on earth= 9.8 m/s²
```

Locating Python Modules

- Python modules are located by interpreter in following steps.
    - First, it will check for the built-in module.
    - If not built-in module, Search for the Module in the current directory
    - If not found in current directory, Python then searches each directory in the shell variable PYTHONPATH (An environment variable, consisting of a list of directories).
    - If that also fails python checks the sys.path (A built-in variable within the sys module. It contains a installation-dependent list of directories configured during Python installation).

Modules – Import mechanisms

Locating Python Modules

- **To get the Directories List**

```
# importing sys module
import sys
# importing sys.path
print(sys.path)
```

Output:

```
['', 'C:\\Users\\SOWMYA SHREE P\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\idlelib',
'C:\\Users\\SOWMYA SHREE P\\AppData\\Local\\Programs\\Python\\Python311\\python311.zip',
'C:\\Users\\SOWMYA SHREE P\\AppData\\Local\\Programs\\Python\\Python311\\DLLs',
'C:\\Users\\SOWMYA SHREE P\\AppData\\Local\\Programs\\Python\\Python311\\Lib',
'C:\\Users\\SOWMYA SHREE P\\AppData\\Local\\Programs\\Python\\Python311',
'C:\\Users\\SOWMYA SHREE P\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site-packages’]
```

This is a list of directories that the interpreter will search for the required module.

Modules – Import mechanisms

Sys.path.append()

- a built-in function of sys module that can be used with path variable to add a specific path for interpreter to search.

```
import sys
sys.path.append( '/path/to/module’)
```

Sys.path.insert()

- a built-in function of sys module that can be used to insert a path at a specific position in sys.path.

```
import sys
sys.path.insert(0, '/path/to/module’)
```

0 indicates that the path should be inserted at the beginning of sys.path .

-1 to insert a path at the end of sys.path.

__doc__ variable

- In Python, each object(Class, Fucntion, variable,..) can be documented using Docstrings.
- Docstrings can be accessed using the __doc__ attribute.

```
def add():
   '''Performing addition of two numbers.'''
   a=10
   b=7
   print(a+b)

print("Using __doc__:")
print(add.__doc__)

print("Using help:")
help(add)
```

Output

```
Using __doc__:
Performing addition of two numbers.
Using help:
Help on function add in module __main__:

add()
   Performing addition of two numbers.
```

__name__ variable and Modules

- It is a special variable in Python.
- If the source file is executed as the main program, the interpreter sets the __name__ variable to the value “__main__”.
- If this file is being imported from another module, __name__ will be set to the module’s name.

Modules – Import mechanisms

```
print (“file1 __name__ = %s" %__name__)

if __name__ == "__main__":
    print (“file1 is executed directly")
else:
    print (“file1 is imported")
```

file1.py

Output

```
file1 __name__ = __main__
file1 is executed directly
```

After executing file1.py

Modules – Import mechanisms

```
import file1

print ("file2 __name__ = %s" %__name__)

if __name__ == "__main__":
   print ("file2 is executed directly")
else:
   print ("file2 is imported")
```

file2.py

Output

```
file1 __name__ = file1
file1 is imported
file2 __name__ = __main__
file2 is executed directly
```

After executing file2.py

Modules - Summary

- Python Module is a python script file that can contain variables, functions, and classes.
- Python modules help us in organizing our code and then referencing them in other classes or python scripts.
- Modular Programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. These sub-tasks are Modules.

THANK YOU

Department of Computer Science and Engineering

Contact Email ID's:

sindhurpai@pes.edu

sowmyashree@pes.edu

PES
UNIVERSITY

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Prof. Sowmya Shree P, Prof. Sindhu R Pai
Department of Computer Science
and Engineering

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

Unit - 3: Testing – Pytest, Function testing with Doctest, pdb debugger commands

Prof. Sowmya Shree P, Prof. Sindhu R Pai

Department of Computer Science and Engineering

Pytest

- Pytest is a robust testing framework for Python.
- It allows users to write test codes using Python programming language.
- It helps to write tests from simple unit tests to complex functional tests.

Advantages of Pytest

1. Free and Open-Source
2. Simple syntax - very easy to start with
3. Run multiple tests in parallel, which reduces the execution time of the test suite
4. Automatically detect test file and test functions, if not mentioned explicitly
5. Allows to skip a subset of the tests during execution
6. Allows to run a subset of the entire test suite

Features of Pytest

1. Does not require API to use
2. Provides useful plugins
3. Can be written as a function or method
4. Gives useful failure information without the use of debuggers
5. Can be used to run doc tests and unit tests

Pytest – Environmental Setup

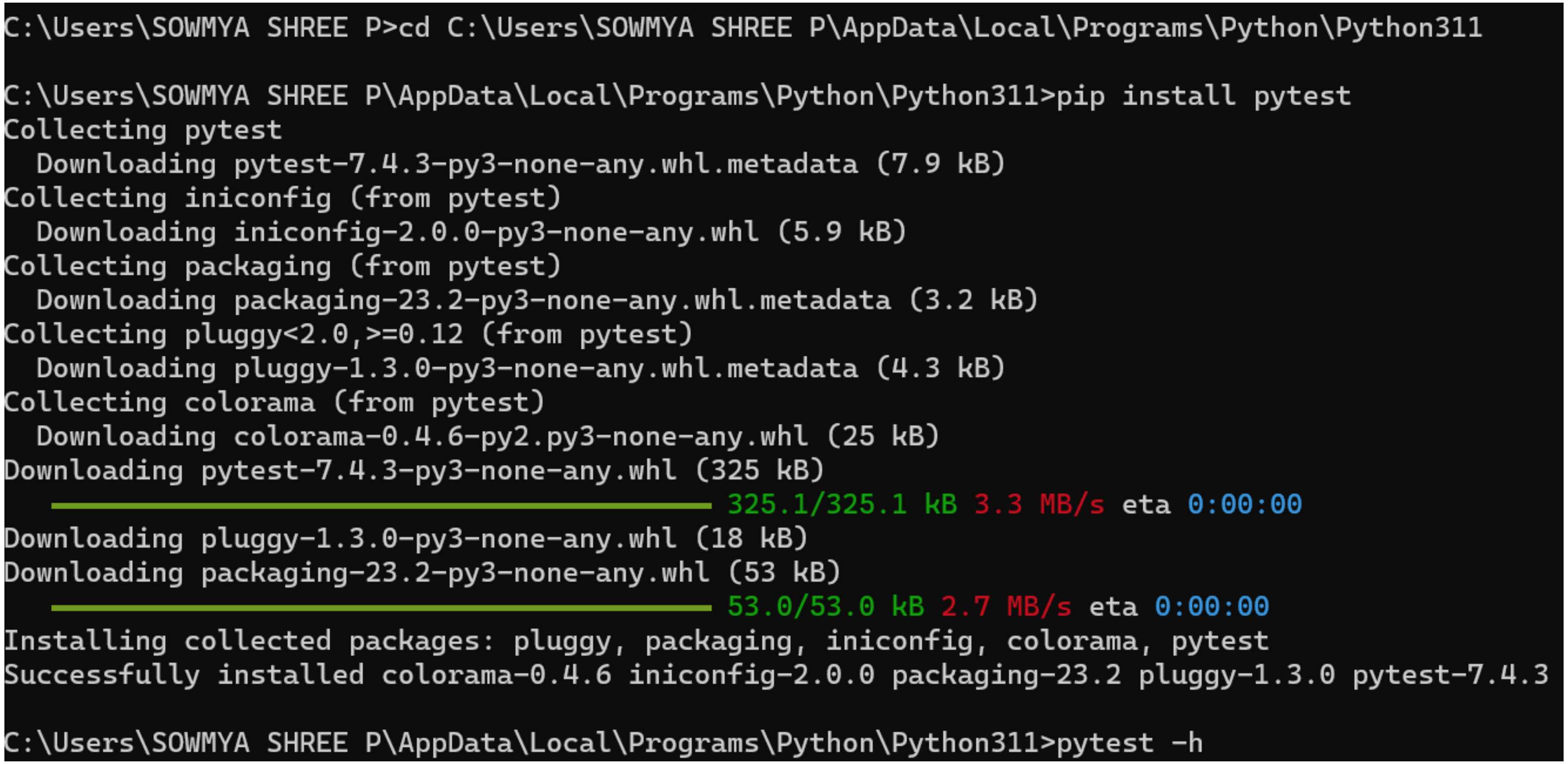
1. Open command prompt
2. Change directory to the location where Python is installed
3. Type command

   ```
   pip install pytest
   ```

4. Confirm the installation

   ```
   pytest -h
   ```

Pytest – Environmental Setup



```
C:\Users\SOWMYA SHREE P>cd C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311

C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311>pip install pytest
Collecting pytest
  Downloading pytest-7.4.3-py3-none-any.whl.metadata (7.9 kB)
Collecting iniconfig (from pytest)
  Downloading iniconfig-2.0.0-py3-none-any.whl (5.9 kB)
Collecting packaging (from pytest)
  Downloading packaging-23.2-py3-none-any.whl.metadata (3.2 kB)
Collecting pluggy<2.0,>=0.12 (from pytest)
  Downloading pluggy-1.3.0-py3-none-any.whl.metadata (4.3 kB)
Collecting colorama (from pytest)
  Downloading colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Downloading pytest-7.4.3-py3-none-any.whl (325 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 325.1/325.1 kB 3.3 MB/s eta 0:00:00
Downloading pluggy-1.3.0-py3-none-any.whl (18 kB)
Downloading packaging-23.2-py3-none-any.whl (53 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 53.0/53.0 kB 2.7 MB/s eta 0:00:00
Installing collected packages: pluggy, packaging, iniconfig, colorama, pytest
Successfully installed colorama-0.4.6 iniconfig-2.0.0 packaging-23.2 pluggy-1.3.0 pytest-7.4.3

C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311>pytest -h
```

Pytest – Example

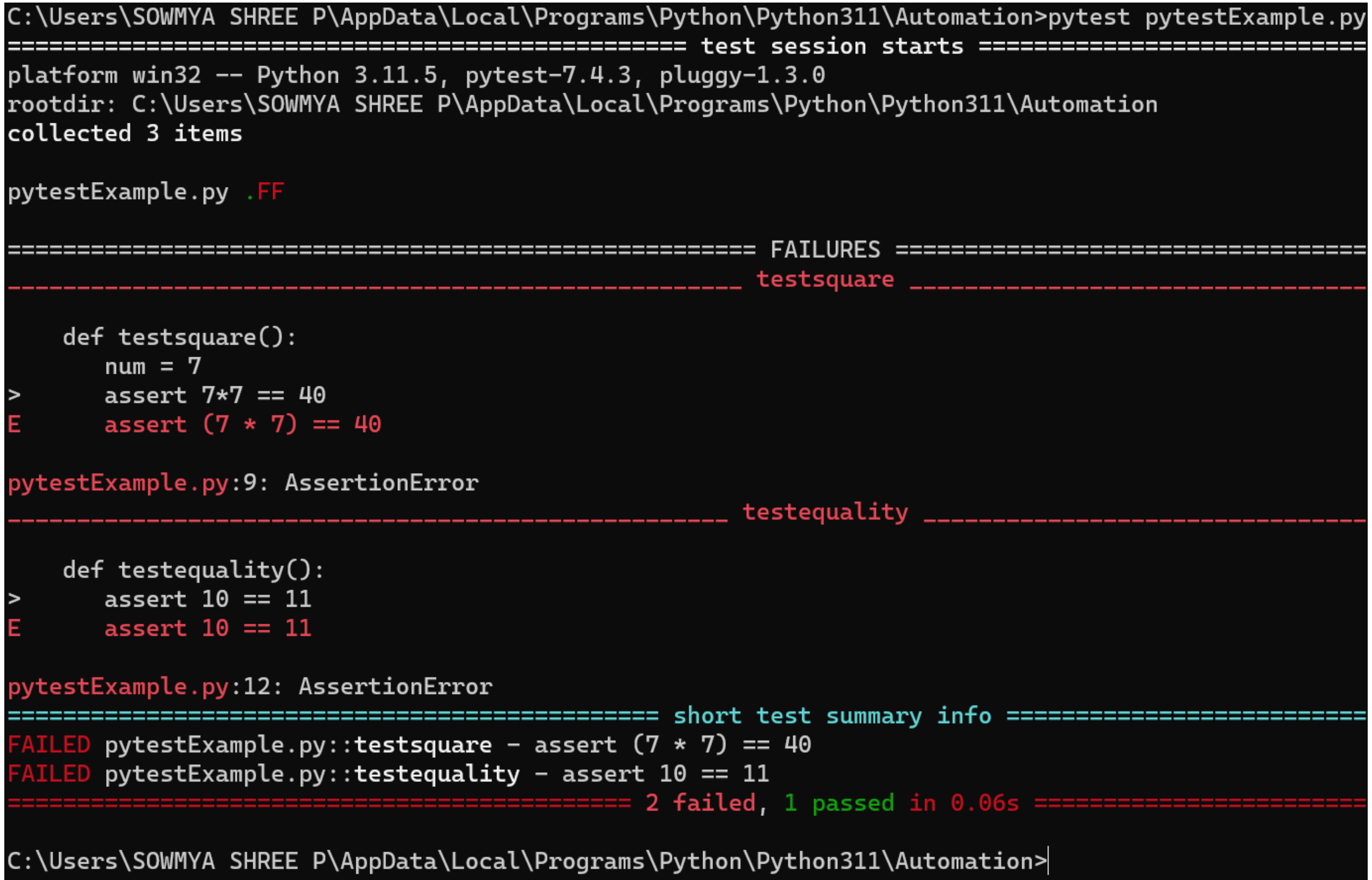
1. Create a new directory “automation” and navigate into the directory in the command line.
2. Create a file pytestExample.py

```
import math

def testsqrt():
   num = 25
   assert math.sqrt(num) == 5

def testsquare():
   num = 7
   assert 7*7 == 40

def testequality():
   assert 10 == 11
```

3. Run the file using the command

   pytest pytestExample.py

Pytest – Example (Output)



```
C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311\Automation>pytest pytestExample.py
================================================= test session starts =============================
platform win32 -- Python 3.11.5, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311\Automation
collected 3 items

pytestExample.py .FF

====================================================== FAILURES ===================================
_____________________________________________________ testsquare __________________________________

    def testsquare():
        num = 7
>       assert 7*7 == 40
E       assert (7 * 7) == 40

pytestExample.py:9: AssertionError
____________________________________________________ testequality _________________________________

    def testequality():
>       assert 10 == 11
E       assert 10 == 11

pytestExample.py:12: AssertionError
=============================================== short test summary info ===========================
FAILED pytestExample.py::testsquare - assert (7 * 7) == 40
FAILED pytestExample.py::testequality - assert 10 == 11
============================================= 2 failed, 1 passed in 0.06s =========================

C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311\Automation>
```

Function testing with Doctest

Doctest

- It is a module included in the Python programming language's standard library.

- It allows the easy generation of tests based on output from the standard Python interpreter shell.

- It finds patterns in the *docstring.*

- *Docstrings* - provides description of a class or a function to provide a better understanding of the code. Also, used for Testing purposes using doctest module.

Need for Doctest

- To check that a module’s docstrings are up-to-date (to ensure code still work as documented).
- To perform Regression Testing (Verifying the changes made to the code will not impact the existing functionalities of the Software).

Function testing with Doctest

Steps to write a function with doctest

1. Import the doctest module.
2. Write the function with docstring.
3. Inside the docstring, write the following two lines for testing the function.

   >>>function_name(args)

   Expected Output
4. Write the function logic(Coding).
5. Call the doctest.testmod(name= function_name, verbose=True)

If 'verbose' is set to False(default), output will be shown in case of failure only, not in the case of success.

Function testing with Doctest

Example1: Illustrating the testcase-pass

```
# import testmod for testing a function
from doctest import testmod

# define a function to test
def fact(n):
    '''
    >>> fact(5)
    120
    >>> fact(0)
    1
    '''
    if n==0:
        res=1
```

Lines for testing the function.

Function testing with Doctest

Example1 (Contd…)

```
    else:
        res=n*fact(n-1)
    return res

# call the testmod function
testmod(name ='fact', verbose = True)
```

Output:

```
Trying:
    fact(5)
Expecting:
    120
ok
Trying:
    fact(0)
Expecting:
    1
ok
1 items had no tests:
    fact
1 items passed all tests:
   2 tests in fact.fact
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Example2: Illustrating the testcase-failed

```
# import testmod for testing a function
from doctest import testmod

# define a function to test
def fact(n):
    '''
    >>> fact(5)
    120
    >>> fact(0)
    1
    '''
    if n==0:
        res=1
```

Lines for testing the function.

Function testing with Doctest

Example2 (Contd…)

```
    else:
        res=fact(n-1) #Wrong logic to compute factorial
    return res

# call the testmod function
testmod(name ='fact', verbose = True)
```

Output:

```
Failed example:
    fact(5)
Expected:
    120
Got:
    1
Trying:
    fact(0)
Expecting:
    1
ok
**********************************
**********************************
**
2 items had failures:
   2 of   2 in fact
   1 of   2 in fact.fact
4 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failure.
```

pdb debugger commands

pdb Module

- It is a module with a set of utilities for debugging of Python programs.
- pdb internally uses bdb (basic debugger) and cmd (command interpreters) modules.
- pdb runs purely in the command line.
- Pdb supports Setting breakpoints, Stepping through code, Source code listing, Viewing stack traces.

pdb debugger commands

pdb Module

- Pdb debugger can be invoked in two ways
  1. Command Line
     `python -m pdb fileName.py`

  2. Importing `pdb` module and call `pdb.set_trace()`

pdb debugger commands

- **Command Line**

1. Create a Python Script

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

print("Factorial of 5 =",fact(5))
```

pdbExample.py

2. Open Command Prompt

3. Change to the directory location where python is installed

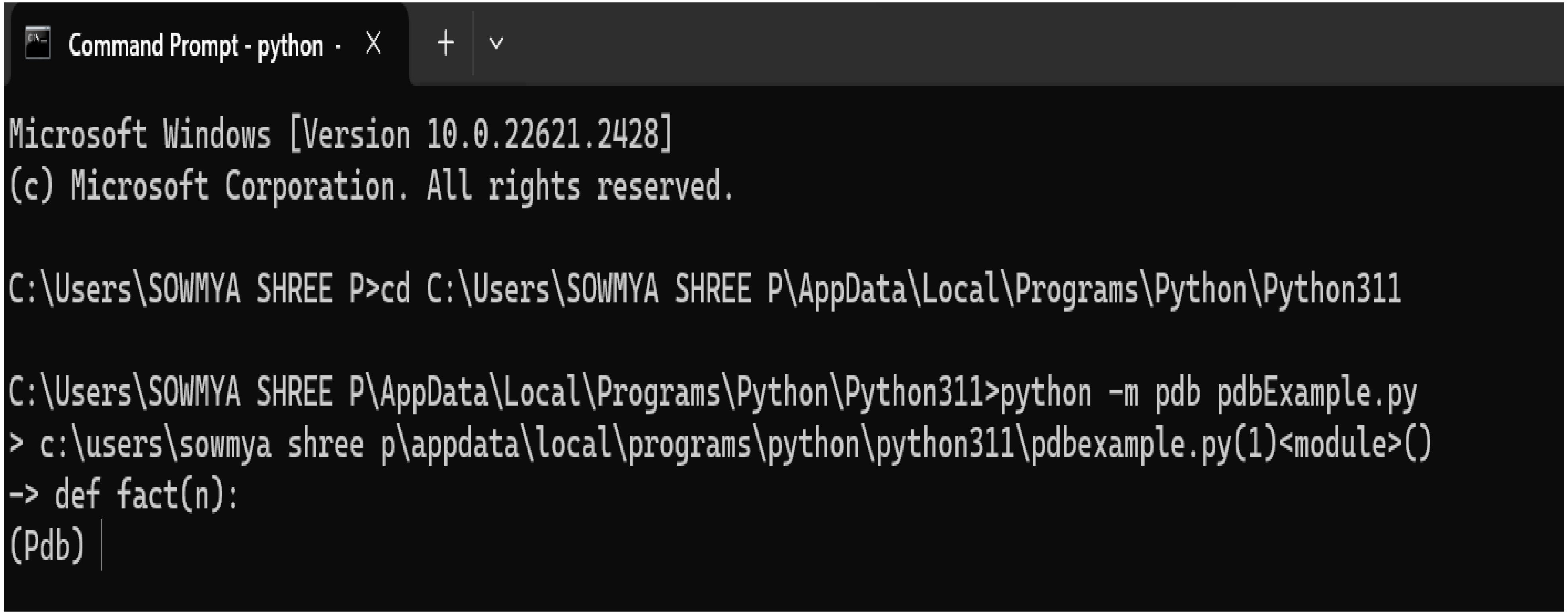
```
cd     C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311
```

3. Type the below command

```
python -m pdb pdbExample.py
```

PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

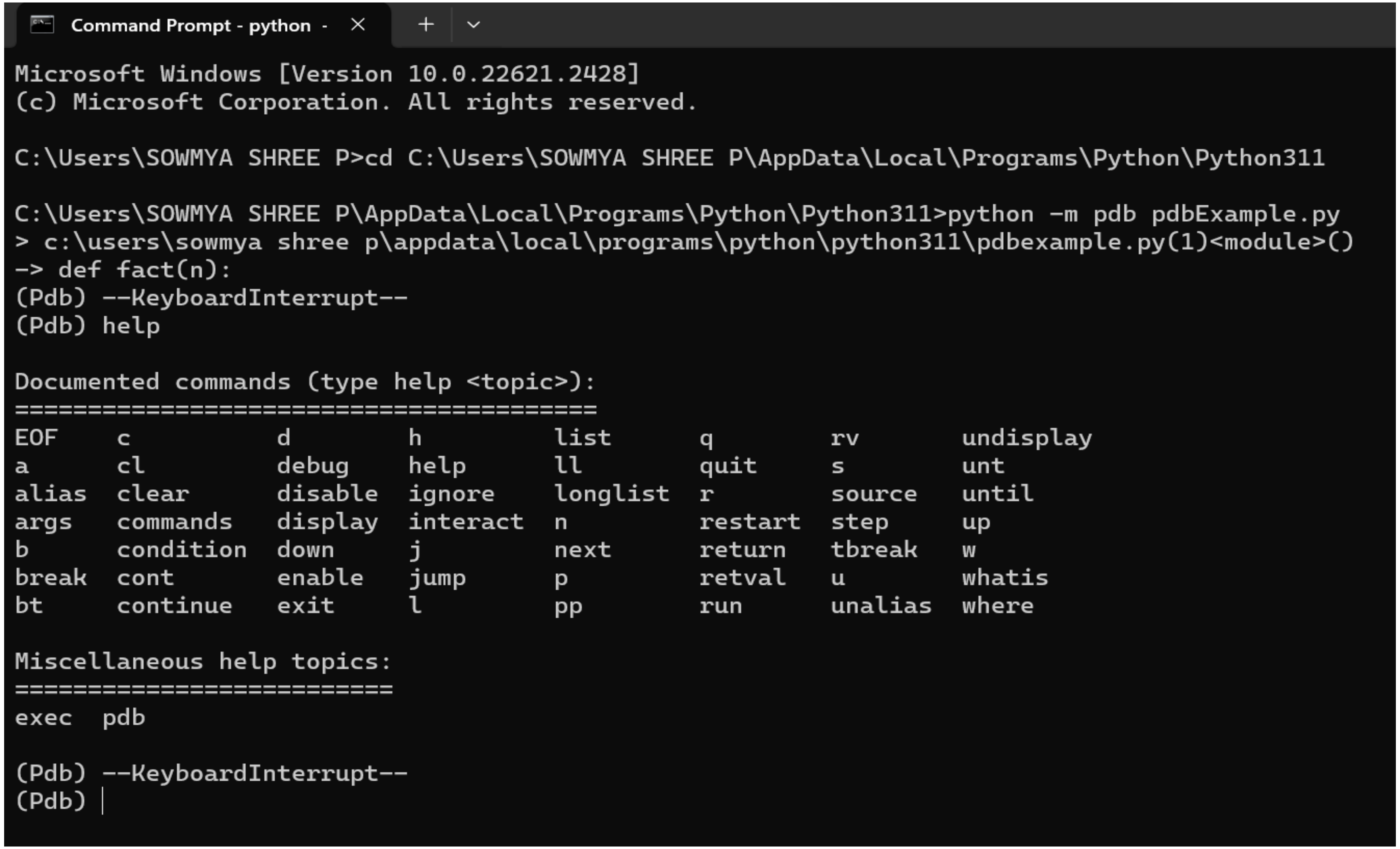
pdb debugger commands



```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\SOWMYA SHREE P>cd C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311

C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311>python -m pdb pdbExample.py
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(1)<module>()
-> def fact(n):
(Pdb)
```

Now, type help in front of the debugger prompt to know more about any command.

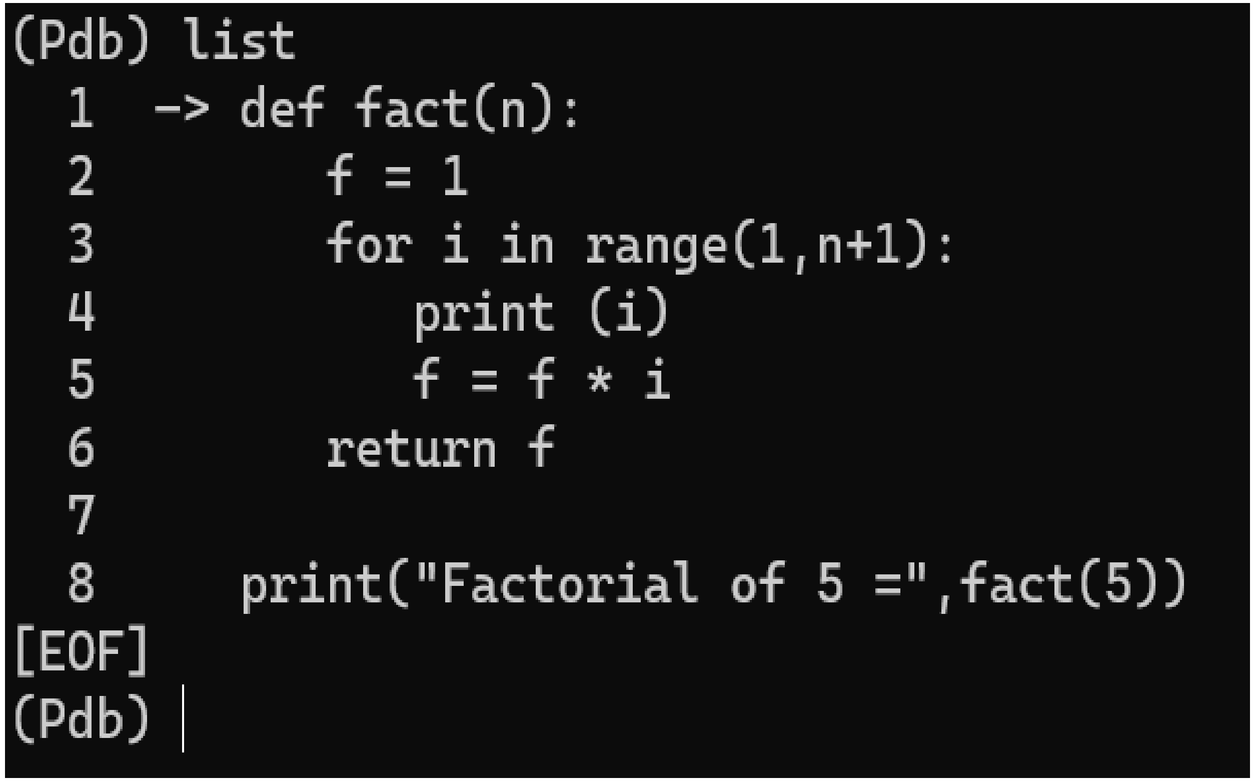
pdb debugger commands



```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\SOWMYA SHREE P>cd C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311

C:\Users\SOWMYA SHREE P\AppData\Local\Programs\Python\Python311>python -m pdb pdbExample.py
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(1)<module>()
-> def fact(n):
(Pdb) --KeyboardInterrupt--
(Pdb) help

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb) --KeyboardInterrupt--
(Pdb)
```

pdb debugger commands

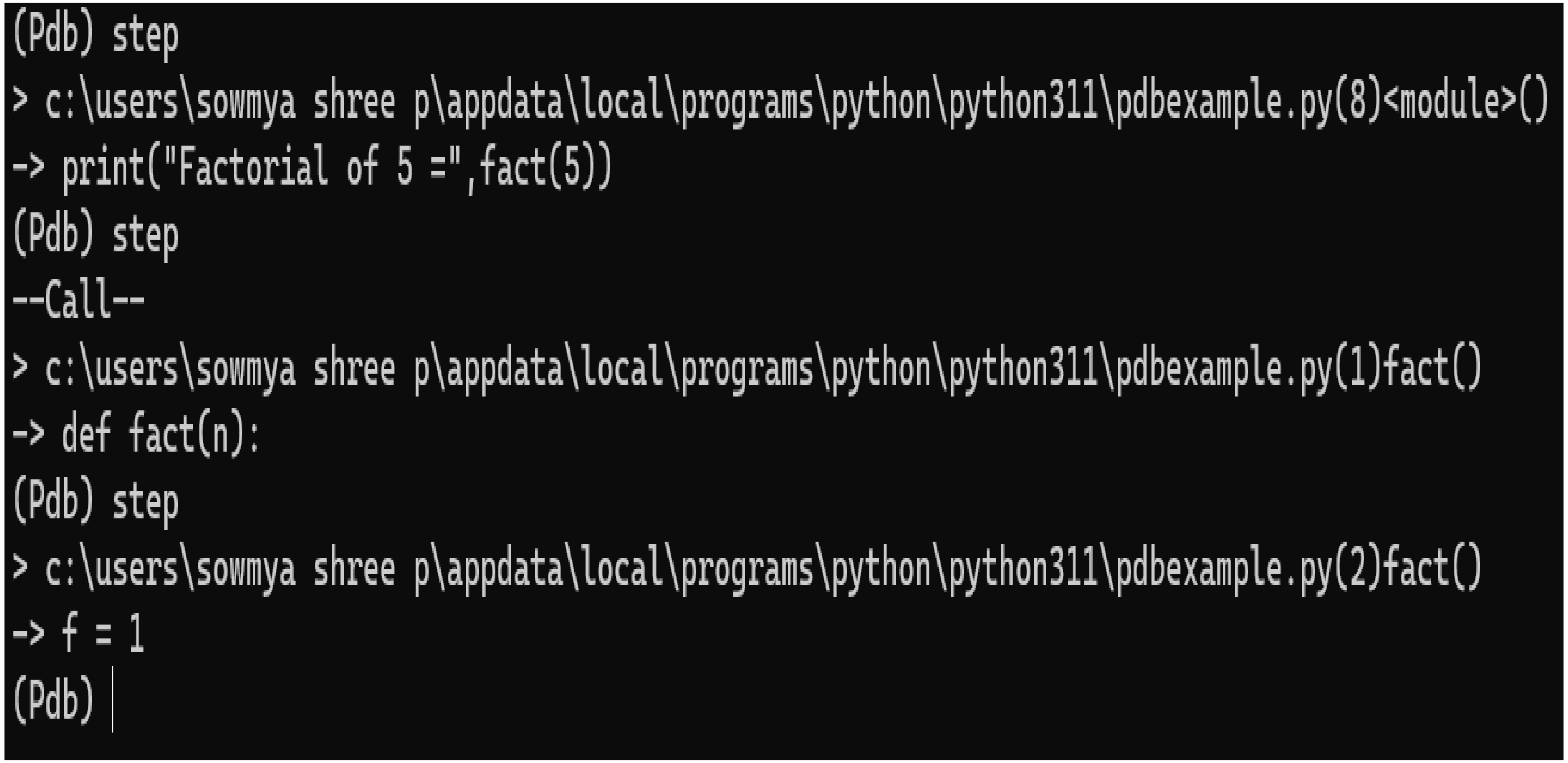
- list command - lists entire code with -> symbol to the left of a line at which program has halted.



```
(Pdb) list
  1  -> def fact(n):
  2          f = 1
  3          for i in range(1,n+1):
  4              print (i)
  5              f = f * i
  6          return f
  7
  8      print("Factorial of 5 =",fact(5))
[EOF]
(Pdb)
```

pdb debugger commands

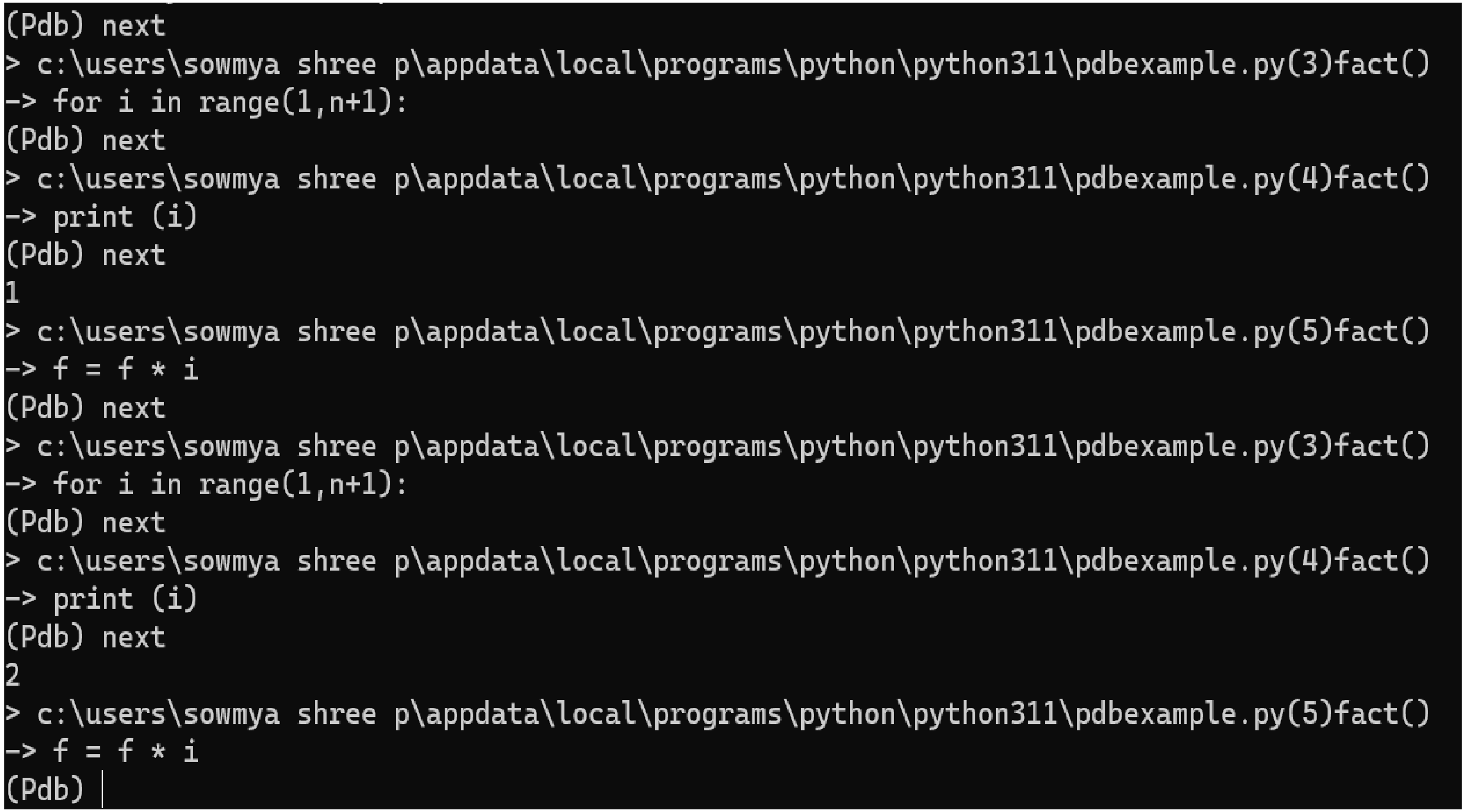
- step command – move line by line, will cause a program to stop within a function



```
(Pdb) step
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(8)<module>()
-> print("Factorial of 5 =",fact(5))
(Pdb) step
--Call--
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(1)fact()
-> def fact(n):
(Pdb) step
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(2)fact()
-> f = 1
(Pdb)
```

pdb debugger commands

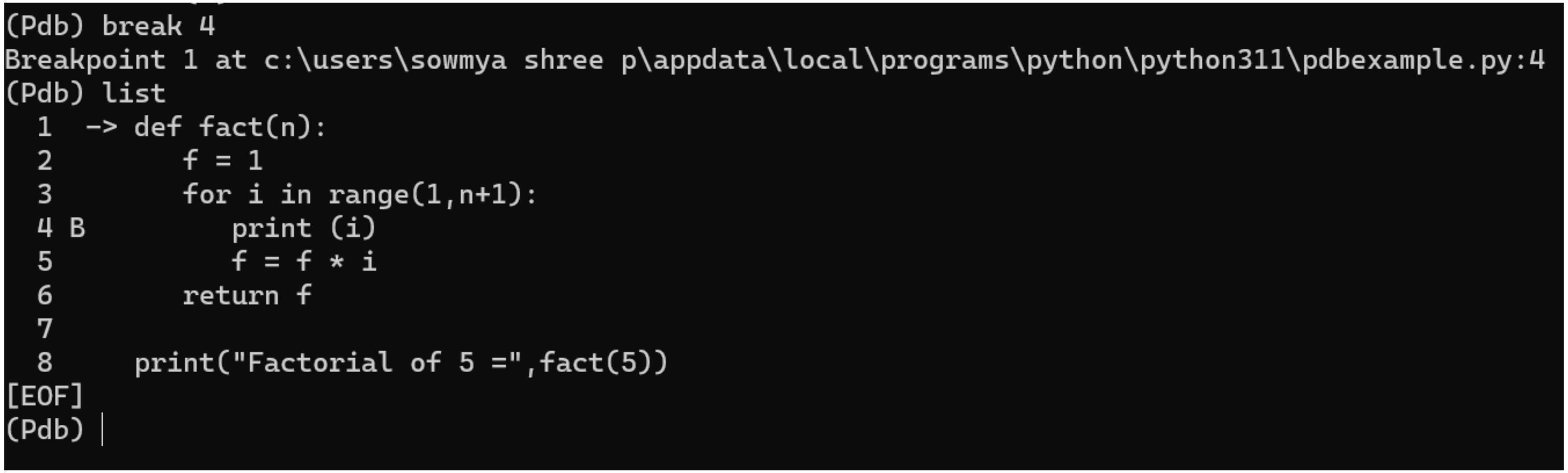
- next command - move line by line, executes a called function and stops after it.



```
(Pdb) next
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(3)fact()
-> for i in range(1,n+1):
(Pdb) next
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(4)fact()
-> print (i)
(Pdb) next
1
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(5)fact()
-> f = f * i
(Pdb) next
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(3)fact()
-> for i in range(1,n+1):
(Pdb) next
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(4)fact()
-> print (i)
(Pdb) next
2
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(5)fact()
-> f = f * i
(Pdb)
```

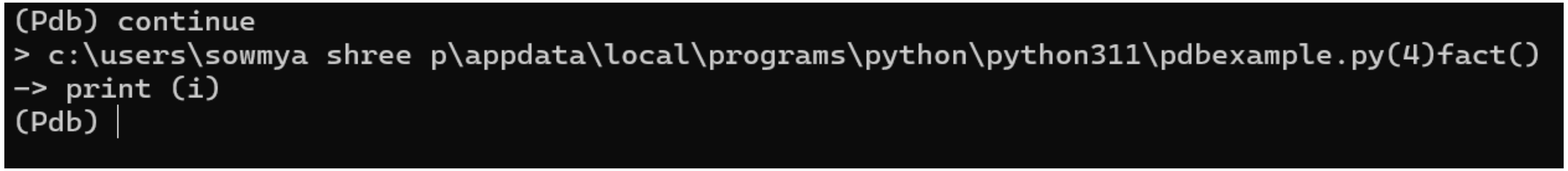
pdb debugger commands

- break command – set breakpoints within a program. Line number must be given.



```
(Pdb) break 4
Breakpoint 1 at c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py:4
(Pdb) list
  1  -> def fact(n):
  2         f = 1
  3         for i in range(1,n+1):
  4 B           print (i)
  5             f = f * i
  6         return f
  7
  8     print("Factorial of 5 =",fact(5))
[EOF]
(Pdb)
```

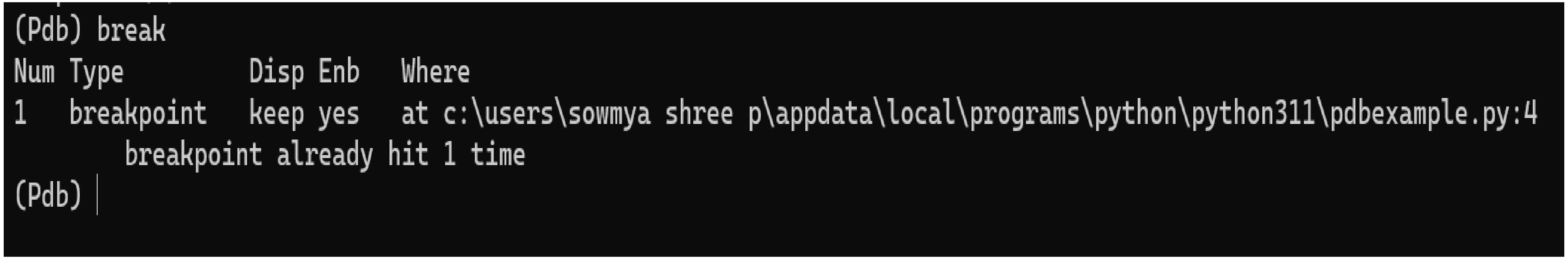
- continue command – program execution will proceed till it encounters a breakpoint



```
(Pdb) continue
> c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py(4)fact()
-> print (i)
(Pdb)
```

pdb debugger commands

- break command – Display all break points using break command without line number



```
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at c:\users\sowmya shree p\appdata\local\programs\python\python311\pdbexample.py:4
        breakpoint already hit 1 time
(Pdb)
```

pdb debugger commands

- **Using pdb.set_trace()**

The Pdb debugger can be used from within Python script also

1. import pdb

```
import pdb
def fact(n):
  f = 1
  for i in range(1,n+1):
    pdb.set_trace()
    print (i)
    f = f * i
  return f

print("Factorial of 5 =",fact(5))
```

pdbExample.py

2. Call set_trace function

The behavior of the debugger will be exactly the same as we find it in a command line environment.

Summary

- **pytest** – a tesing framework in Python, helps to write tests from simple unit tests to complex functional tests.

- **doctest** - a module that verifies whether the code work as intended. It allows generation of tests based on output from the standard Python interpreter shell.

- **pdb** - a module with a set of utilities for debugging of Python programs.

PES UNIVERSITY

THANK YOU

Department of Computer Science and Engineering

Contact Email ID's:

sindhurpai@pes.edu

sowmyashree@pes.edu