

# **Python Functions, Recursion, Import, File Processing**

Panchatcharam M

- ✓ **Introduce Functions**
  - ✓ **Decompose the problem into smaller portions**
  - ✓ **Introduce abstraction**
- 
- ✗ **More code is not necessarily a good thing**
  - ✗ **Keeping everything in a single file**
  - ✗ **Writing 1000 lines of code**

- ❖ **Will you ask a single person to organize everything?**
  - 1. Food Committee**
  - 2. Cultural Committee**
  - 3. Finance Committee**
  - 4. Technical Committee**
  - 5. Website Development Committee**
  - 6. ...**
  - 7. ....**

# *Multi TV –Single Picture*

❖ **Different devices work together for a single job**



- ❖ **Divide the program or code into modules**
  - ✓ **It should be self-contained**
  - ✓ **Break up the codes into pieces**
  - ✓ **It should be reusable**
  - ✓ **Helps you to organize the code**
  - ✓ **Coherence**

❖ **Functions/Modules/Procedures/Methods**

❖ **Classes**

- ❖ **Take a Bank Details or Your Mobile Phone or PC**
  - ✓ **It is not necessary that everyone should know everything about your account**
  - ✓ **Manager/Administrator has a role**
  - ✓ **Cashier/User has a role**
- ✓ **Think: A piece of code as black box**
  - ✓ **Cannot See**
  - ✓ **Do not need to see**
  - ✓ **Do not want to see**
  - ✓ **Hide Coding details**

- ✓ **Functions:** Write reusable pieces of code
  - ✓ Should not run until they are called or invoked
- ✓ **Characteristics:**
  - ✓ **Name**
  - ✓ **Parameters ( $\geq 0$ )**
  - ✓ **Docstring (Optional)**
  - ✓ **Body**
  - ✓ **Return (None or something)**

# FUNCTIONS



- **`def` creates a function and assigns it a name**
- **`return` sends a result back to the caller**
- **Arguments are passed by assignment**
- **Arguments and return types are not declared**

```
def name(arg1, arg2, ...):  
    """documentation""" # optional doc string  
    statements or Body  
    return expression    # from function  
  
def product(x,y):  
    return x*y  
  
product(2,3)  
6
```

```
def gcd(a, b):  
    "greatest common divisor"  
    while a != 0:  
        a, b = b%a, a        # parallel assignment  
    return b
```

$a = q_0b + r_0$   
 $b = q_1r_0 + r_1$

```
>>> gcd.__doc__  
'greatest common divisor'  
>>> gcd(12, 20)  
4
```

$b = q_0a + r_0$   
 $a = q_1r_0 + r_1$   
 $r_0 = q_2r_1 + r_2$

- ✓ **Formal Parameter** gets bound the value of actual parameter when function is called
- ✓ **New Scope/Frame/Environment** created when enter a function
- ✓ **Scope: Mapping of names to objects**

```
def func(a):  
    a = a + 1  
    print('Value of a', a)  
    return a
```

*Formal parameter*

```
a = 10  
b = func(a)
```

*Actual parameter*

```
def func(a):  
    a = a + 1  
    print('Value of a', a)  
    return a
```

```
a = 10
```

```
b = func(a)
```

Global Scope		Func Scope	
Func	Some Code	a	10
a	10		
b	11		

```
def func(a):  
    a = a + 1  
    print('Value of a', a)
```

```
a = 10  
b=func(a)  
print(b)
```

# Function as Arguments

```
def func_a():  
    print('Inside func_a')  
  
def func_c(y):  
    print('Inside func_c')  
    return func_a()  
  
print(func_a())  
print(7+func_c(3))  
print(func_c(func_a))
```

```
def func_b(x):  
    print('Inside func_b')  
    return x
```

Global Scope		Func_a Scope	
Func_a			
Func_b			
Func_c			

- ✓ Positional Arguments
- ✓ Keyword Arguments

```
def functionpositional(x,y,z): #Positional Arguments
    print("I am inside the Positional")
    print("My Name is ",x)
    print("My Age is ",y)
    print("My Marks is ",z)
```

```
def functionkeyword(Name=None, Age=None, Marks=None): #Keyword
Arguments
```

```
    print("Another Definition")
    print("My Name is ",Name)
    print("My Age is ",Age)
    print("My Marks is ",Marks)
```

```
x,y,z="IITTP",6,4.8
functionpositional(x,y,z)
functionkeyword(Age=y, Name=y, Maks=z)
```

```
def func(a, b, c=10, d=100):  
    print(a, b, c, d)
```

```
>>> func(1, 2)
```

```
1 2 10 100
```

```
>>> func(1, 2, 3, 4)
```

```
1 2 3 4
```



- ✓ **Non-keyword Arguments with variable length**
- ✓ **Keyword Arguments with variable length**

```
def studentdetails(name,*data):  
    print(name)  
    print(data)  
    print(type(data))  
studentdetails("placement",28,'A',True)
```

```
def studentdetailskey(**kwargs):  
    print(kwargs["name"])  
    print(type(kwargs))  
    for keys,vals in kwargs.items():  
        print(keys,vals)
```

```
studentdetailskey(name=x,Age=y,Grade=z)  
studentdetailskey(sno="firstcolumn",name="secondcolumn",quiz="third  
column")  
studentdetailskey(sno="firstcolumn",quiz="thirdcolumn")
```

# *Lambda Function/Expression*

- ✓ **Lambda operator or lambda function or lambda expression is a way to create small anonymous function**
- ✓ **Function without name**
- ✓ **Throw away function**
- ✓ **Major uses Filter, Map, Reduce**

**Lambda argument list: lambda expression**

```
double = lambda x: x*x  
double(4)  
16
```

```
def secretkey(n):  
    return lambda a: a*n  
mykey=secretkey(4)  
print(mykey(11))
```

- ✓ Filter out all elements of a list or dictionary
- ✓ Example find all even numbers from a list

Syntax: `filter(function, list)`

```
numbers=[0,1,3,4,6,10,39,30,28,5,7]
even=list(filter(lambda x: x%2==0, numbers))
print(even)
```

```
phoneos=['Android','iOS','Ubuntu','Windows','Android','Android','Ubuntu']
even=list(filter(lambda x: x=='Android', phoneos))
print(even)
```

- ✓ It maps to a new list
- ✓ Applies the function to all the elements of the sequence

Syntax: `map(function, seq)`

```
numbers=[0,1,3,4,6,10,39,30,28,5,7]
even=list(map(lambda x: x*2, numbers))
print(even)
```

```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)
```

```
temperature=[0,1,3,4,6,10,39,30,28,5,7]
F=list(map(fahrenheit, temperature))
C=list(map(celsius, temperature))
```

```
temperature=[0,1,3,4,6,10,39,30,28,5,7]
F=list(map(lambda x: ((float(9)/5)*x + 32), temperature))
C=list(map(lambda x: (float(5)/9)*(x-32), temperature))
```

# Reduce Function

- ✓ **sum():** This function computes the sum of an array
- ✓ Instead of if you want to operate on two different parameters, then func can be used
- ✓ First define two elements of the list: func(s1,s2)
- ✓ reduce on a list = [s1,s1,...,sn] will work as follows
  - ✓ [func(s1,s2),s3,...,sn] Syntax: reduce (function, seq)
  - ✓ [func(func(s1,s2),s3),s4,...,sn]
  - ✓ [func(func(func(s1,s2),s3),s4),s5,...,sn]
  - ✓ .....

```
numbers=[0,1,3,4,6,10,39,30,28,5,7]
mysum= reduce (lambda x,y: x+y, numbers)
print(mysum)
```

```
numbers=[0,1,3,4,6,10,39,30,28,5,7]
mysub=reduce (lambda x,y: x*y, numbers)
print(mysub)
```

# RECURSION

- ✓ **Recursion**
  - ✓ **Base Case**
  - ✓ **Non Base case**

```
def fact(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        return n*fact(n-1)
```

```
def fib(n):  
    if n==1 or n==2:  
        return 1  
    else:  
        return fib(n-1)+fib(n-2)
```

# Efficient Recursion with Dictionaries

```
def fib(n):  
    global counterfib  
    counterfib+=1  
    if n==1 or n==2:  
        return 1  
    else:  
        return fib(n-1)+fib(n-2)
```

```
def fibeff(n,d):  
    global counterfibeff  
    counterfibeff+=1  
    if n in d:  
        return d[n]  
    else:  
        result=fibeff(n-1,d)+fibeff(n-2,d)  
        d[n]=result  
    return result
```



# Efficient Recursion with Dictionaries

```
def fib(n):  
    global counterfib  
    counterfib+=1  
    if n==1 or n==2:  
        return 1  
    else:  
        return fib(n-1)+fib(n-2)
```

```
def fibeff(n,d):  
    global counterfibeffect  
    counterfibeffect+=1  
    if n in d:  
        return d[n]  
    else:  
        result=fibeff(n-1,d)+fibeff(n-2,d)  
        d[n]=result  
        return result
```

```
from _datetime import datetime  
start=datetime.now()  
counterfib=0  
print(fib(41))  
print("Recursive Functions Called",counterfib)  
print("Total MicroSeconds for Fib= {}".format((datetime.now()-start).microseconds))  
print("Total Seconds for Fib= {}".format((datetime.now()-start).seconds))  
start=datetime.now()  
d={1:1,2:1}  
counterfibeffect=0  
print(fibeff(41,d))  
print("Recursive Functions Called",counterfibeffect)  
print("Total MicroSeconds for FibEff= {}".format((datetime.now()-start).microseconds))  
print("Total Seconds for FibEff= {}".format((datetime.now()-start).seconds))
```

CAN WE OVERLOAD?

NOT EXACTLY

# Function Overloading

- ✓ An assignment of more than one behaviour to a particular function
- ✓ For example adding two integers, adding to double, adding two strings in C++
- ✓ Python does not support it by default

```
def add(a,b):  
    return a+b  
  
def add(a,b,c):  
    return a+b+c  
  
s=add(3,2,4)  
print(s)
```

```
def add(a,b):  
    return a+b  
  
def add(a,b,c):  
    return a+b+c  
  
s=add(3,2)  
print(s)
```

```
def add(a,b,c):  
    return a+b+c  
  
def add(a,b):  
    return a+b  
  
s=add(3,2)  
print(s)
```

```
def add(a,b,c):  
    return a+b+c  
  
def add(a,b):  
    return a+b  
  
s=add(3,2,4)  
print(s)
```

# Inefficient Hackaround

```
def add(*args):  
    if type(args[0])==int or type(args[0])==float:  
        res=0  
    if type(args[0])==str:  
        res=0  
    for x in args:  
        res=res+x  
    return res
```

```
s=add(3,2,4)  
print(s)  
s=add(3,4)  
print(s)
```

✓ Why?????

**IMPORT**

- Modules: File containing Python definitions and Statements
- We can import module definition from one file to another

***import <module\_name>***

Example:

```
import math
import numpy
import random
```

- ❑ A module can contain variables
- ❑ A module can contain function
- ❑ To access the variables and functions use dot operator

***module.variables***  
***module.function***

Example:

```
math.sin(30)
```

```
math.pi
```

```
random.randint(100)
```

- ❑ Further module can contain
  - ❑ Executable statements with Function definitions
  - ❑ Its own private symbol table : Global symbol for all functions of the module
- ❑ Module Search Path
  1. Current directory
  2. List of directories specified in PYTHONPATH



- When we specify *import <module\_name>*
  - You can access each of its function as
    - *module.function()*
- When you have modified a few functions in module
  - You should restart your interpreter
  - Alternatively, you can reload the module
  - E.g.: `import importlib`
  - `importlib.reload(module_name)`

- When a very large application is developed with multiple submodules, you can't import all submodules at one go.
- ***from <module\_name> import \****
  - It does not import all submodules from a package
    - E.g: `from scipy import *` does not import `scipy.integrate`
  - Make sure, you have imported required packages properly
    - E.g: `from scipy.integrate import *`
    - `import scipy.integrate`

- ***from <module\_name> import <name(s)>***
  - You can also integrate a few specific module
    - **E.g:** `from scipy.integrate import quadrature, trapz`

- ***from <module\_name> import <name(s)> as <alt\_name>***
  - You can import a module and given alternative name
  - E.g: `import numpy as np`
  - `import matplotlib as mpl`
  - `import matplotlib.pyplot as plt`

<https://realpython.com/python-modules-packages/#python-packages>

# FILE PROCESSING

- Many programs handle data, which often comes from files.
- Opening a file

```
variableName = open ("filename", "mode")
```

Example:

```
file = open ("bankaccount.txt")
```

## Open Reading (default)

```
file = open("bankaccount.txt", 'r')
```

## Open for writing or overwriting

```
file = open("bankaccount.txt", 'w')
```

## Open for appending

```
file = open("bankaccount.txt", 'a')
```

## Open in binary mode

```
file = open("bankaccount.txt", 'rb')
```

```
file = open("bankaccount.txt", 'wb')
```

- Closing a file: First you should open the file in any one mode

***variableName.close()***

Example:

```
file = open("bankaccount.txt")  
file.close()
```



- Reading the entire contents of a file:

```
variableName = open ("filename") .read (size=-1)
```

**Or**

```
variableName = open ("filename")
```

```
variableName .read ()
```

Example:

```
file_text = open ("bankaccount.txt") .read ()
```

- Reading a file line-by-line:

```
variableName = open ("filename") .readline (size=-1)
```

*or*

```
variableName = open ("filename")
```

```
variableName .readline (size=-1)
```

Example:

```
file_text = open ("bankaccount.txt") .readline ()
```

- Reading a file line-by-line:

```
variableName = open("filename").readlines()  
for line in open("filename").readlines():  
    statements
```

Example:

```
count = 0  
for line in open("bankaccount.txt").readlines():  
    count = count + 1  
print("The file contains", count, "lines.")
```

## *File Processing: reading lines*

```
count = 0
with open("bankaccount.txt") as reader:
    for line in reader.readlines():
        count = count + 1
        print(line,end=' ')
print("The file contains", count, "lines.")
```

```
count = 0
with open("bankaccount.txt") as reader:
    for line in reader:
        count = count + 1
        print(line,end=' ')
print("The file contains", count, "lines.")
```

# *File Processing: Reading a file*

```
count = 0
for line in open("bankaccount.txt"):
    count = count + 1
    print(line)
print("The file contains", count, "lines.")
```

- Reading a file line-by-line:

```
variableName = open ("filename", 'w' ).write (string)
```

**or**

```
variableName = open ("filename", 'w' )
```

```
variableName .write (string)
```

Example:

```
Filewrite = open ("bankaccount.tx", 'w' )
```

```
Filewrite.write ("Account No: 2442, \n
```

```
Account Name : SB")
```

```
Filewrite.close ()
```

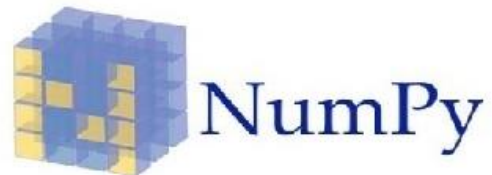
## *File Processing: reading lines*

```
count = 0
with open("bankaccount.txt", 'w') as writer:
    for line in reader.readlines():
        count = count + 1
        print(line, end=' ')
print("The file contains", count, "lines.")
```

```
count = 0
with open("bankaccount.txt") as reader:
    for line in reader:
        count = count + 1
        print(line, end=' ')
print("The file contains", count, "lines.")
```



# End of Python Functions



IP[y]:  
IPython



pandas  
 $y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

