भारतीय प्रौद्योगिकी संस्थान तिरुपति

TIRUPATI

Python Functions

Panchatcharam Mariappan

Assistant Professor

Department of Mathematics and Statistics,
IIT Tirupati

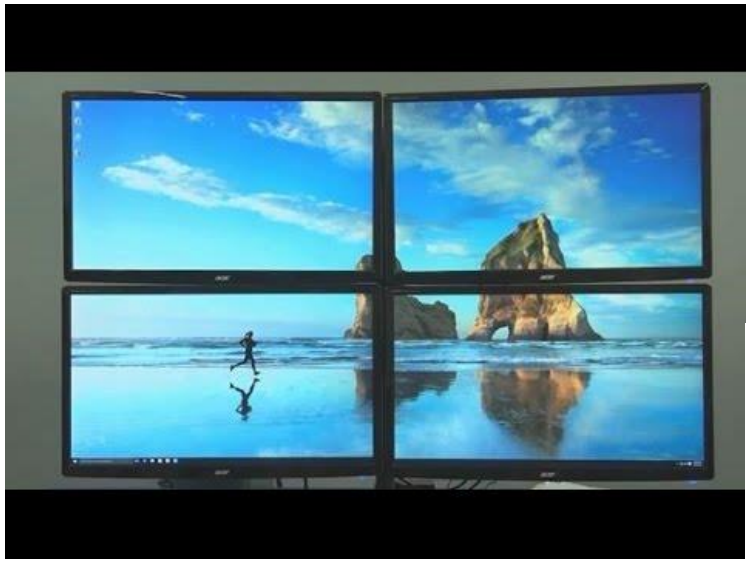
Good Programmer

- ✓ **Introduce Functions**
- ✓ **Decompose the problem into smaller portions**
- ✓ **Introduce abstraction**

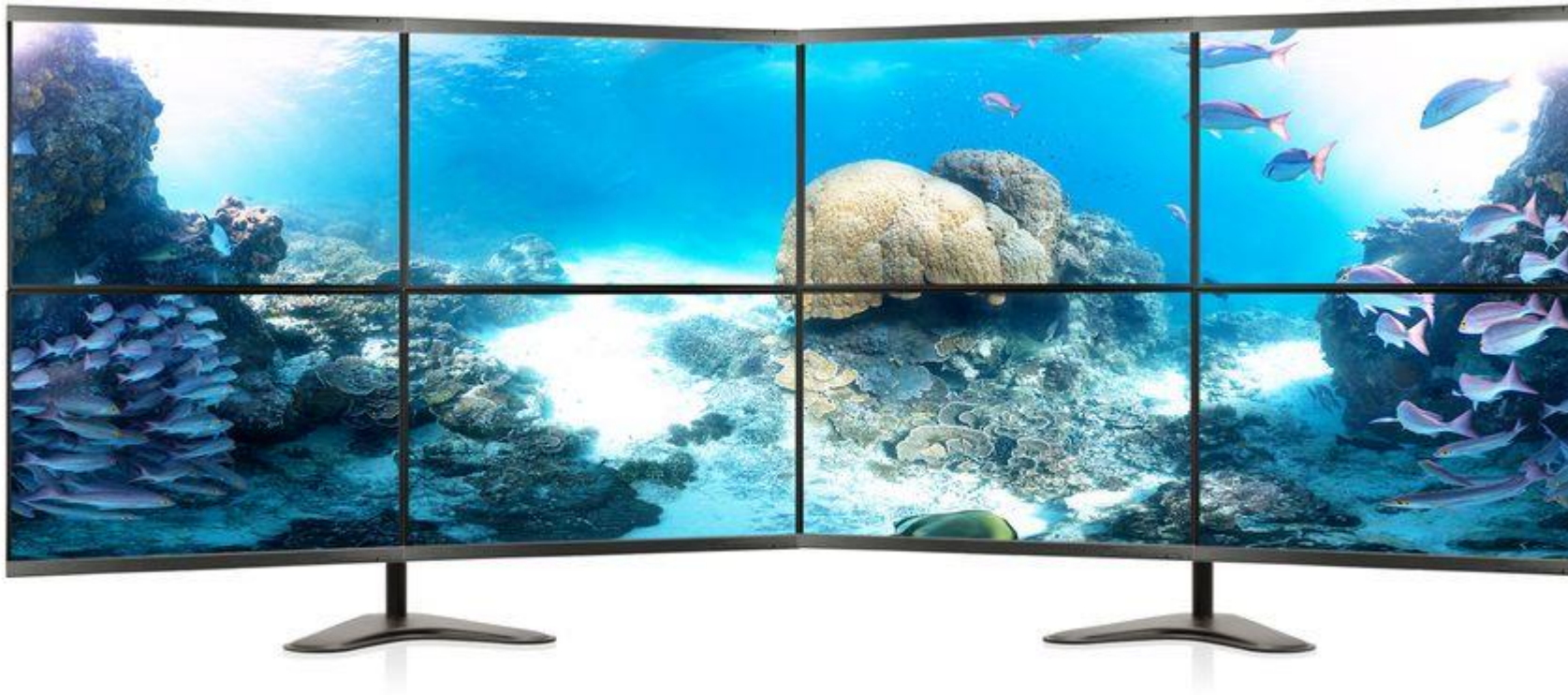
- ✗ **More code is not necessarily a good thing**
- ✗ **Keeping everything in a single file**
- ✗ **Writing 1000 lines of code**

- **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**



Decomposition

- **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**

Abstraction

- **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**
  - ✓ **High Coding details**

Functions

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

FUNCTIONS

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
```

Functions

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

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

Variable Scope

- ✓ **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**

Formal parameter

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

Actual parameter

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

Variable Scope

```
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			

No Return or None Return

```
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 a()

Print(func_a()
Print(7+func(3))
print(func_c(fun_a)
```

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

Global Scope		Func_a Scope	
Func_a	Some Code		
Func_b	Some Code		
Func_c	Some Code		
	None		

Functions Arguments

- ✓ **Positional Arguments**
- ✓ **Keyword Arguments**

```
def printplacementpositional(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 printplacementkeyword(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
printplacementpositional(x,y,z)
printplacementkeyword(Age=y,Name=y,Maks=z)
```

Default Arguments

```
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
```

*args and **kwargs

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

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

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

placement(name=x,Age=y,Grade=z)
placement(sno="firstcolumn",name="secondcolumn",quiz="thirdcolumn")
placement(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 Function

- ✓ **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)
```

Map Function

- ✓ 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)*(T-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)]
  - ✓ [func(func(s1,s2),s3),s4,…,sn)]
  - ✓ [func(func(func(s1,s2),s3),s4),s5,…,sn)]
  - ✓ …..

Syntax: `reduce(function, seq)`

```
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

Efficient Recursion with Dictionaries

- ✓ **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 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
```

```
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}
counterfibeff=0
print(fibeff(41,d))
print("Recursive Functions Called",counterfibeff)
print("Total MicroSeconds for FibEff= {}".format((datetime.now()-start).microseconds))
print("Total Seconds for FibEff= {}".format((datetime.now()-start).seconds))
```

DECORATORS

Decorators

- ✓ The most beautiful and most powerful design
- ✓ More complicated to get into
- ✓ Use of decorators is easy, but writing decorators is complicated
  - ✓ Experience in Functional Programming is must

1. Function Decorators
2. Class Decorators

Preliminaries

```
def nextguess(x):
    return x+1
successor=nextguess
print(successor(20))
Print(nextguess(15))
```

Functions Inside Functions

```
def temperature(t,T):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32
    def farenheit2celsius(x):
        return (float(5)/9)*(x-32)
    if T=='CtoF':
        return "The Temperature  " + str(celsius2fahrenheit(t)) + " degrees F!"
    if T=='FtoC':
        return "The Temperature  " + str(farenheit2celsius(t)) + " degrees C!"


print(temperature(100,'CtoF'))
print(temperature(100,'FtoC'))
```

Functions as Parameters

```
def Function1():
    print('I am indise the Function 1')
def Function2(Function1):
    print('I am inside the Function 2')
    Function1()

Function2(Function1)
```

Functions as Parameters

```
def polynomial(*coeffs):
    def polynomialeval(x):
        result=0
        for index,coeff in enumerate(coeffs[::-1]):
            result+=coeff*x*index
        return result
    return polynomialeval

p0=polynomial(2)
p1=polynomial(1,4)
p2=polynomial(1,2,4)
p4=polynomial(-1,3,4,8,9)
x=1
print(p0(x),p1(x),p2(x),p4(x))
```

$$p_0(x) = 2$$
$$p_1(x) = 4x + 1$$
$$p_2(x) = 4x^2 + 2x + 1$$
$$p_4(x) = 9x^4 + 8x^3 + 4x^2 + 3x - 1$$
$$p_n(x) = \sum_{i=0}^{n} a_i x^i$$

Horners Method

```
def polynomialHorner(*coeffs):
    def polynomialeval(x):
        result=coeffs[0]
        for i in range(1,len(coeffs)):
            result=result*x+coeffs[i]
        return result
    return polynomialeval

p0=polynomialHorner(2)
p1=polynomialHorner(1,4)
p2=polynomialHorner(1,2,4)
p3=polynomialHorner(-1,3,4,8,9)
x=1
print(p1(x),p2(x),p3(x),p0(x))
```

$$p_0(x) = 2$$
$$p_1(x) = 4x + 1$$
$$p_2(x) = 4x^2 + 2x + 1$$
$$p_4(x) = 9x^4 + 8x^3 + 4x^2 + 3x - 1$$
$$p_n(x) = \sum_{i=0}^{n} a_i x^i$$
$$\Big(\dots\big((a_n x + a_{n-1})x + a_{n-2}\big)\Big)x \dots + a_1)x + a_0$$

Simple Decorators

```
#Simple Decorator
def mydecorator(func):
    def mywrapper(x):
        print("Before Calling "+func.__name__)
        func(x)
        print("After Calling "+func.__name__)
    return mywrapper

def mynewfun(x):
    print("My New Function called with an argument",x)

mynewfun("Placement")
print("Let us simple decorate")
mynewfun=mydecorator(mynewfun)
print("Let us call after decorator")
mynewfun("IITTP")
```

Simple Decorators

```
#Simple Decorator
def mydecorator(func):
    def mywrapper(x):
        print("Before Calling "+func.__name__)
        func(x)
        print("After Calling "+func.__name__)
    return mywrapper

def mynewfunc(x):
    print("My New Function called with an argument",x)

mynewfunc("Placement")
print("Let us do simple decorate")
mynewfunc=mydecorator(mynewfunc)
print("Let us call after decorator")
mynewfunc("IITTP")
```

Simple Decorators

```
def div(a,b):
    return a/b

def intellidiv(func):
    def wrapper(a,b):
        if a<b:
            a,b=b,a
        return func(a,b)
    return wrapper

div=intellidiv(div)
print(div(2,4))
```

Simple Decorators

Warning

1. Decoration is usually not permitted in the way we did
2. `mynewfunc=mydecorator(mynewfunc)` is easy to grasp
   - `mynewfunc` exists in the same program in two versions
     - ❑ Before and after decorations

Usual Syntax

Proper Decoration

It occurs in the line before the function header

@ is followed by the decorator function name

Instead of writing the statement

```
mynewfunc=mydecorator(mynewfunc)
```

We can write it as follows

```
@mydecorator
def mynewfunc(x):
    print("My New Function called with an argument",x)

mynewfunc(" IITTP ")
```

Decorators

```
@intellidiv
def div2(a,b):
    return a/b

print("Output from div2",div2(2,4))
print("Output from div2",div2(4,2))
```

```
@mydecorator
def myanotherfunc(x):
    print('Hey, I am a new function Here,',x)

myanotherfunc("I got Success")
```

Decorators with Third Party Functions

```
from math import sin, cos
def mydecorator(func):
    def mywrapper(x):
        print("Before Calling "+func.__name__)
        print(func(x))
        print("After Calling "+func.__name__)
    return mywrapper

sin=mydecorator(sin)
cos=mydecorator(cos)
sin(3.415)
cos(3.415)
```

Warning

1. You can't use with @ sign here

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????**

@dispatch decorator

```
from multipledispatch import dispatch
@dispatch(int,int)
def add(a,b):
    return a+b
@dispatch(int,float)
def add(a,b):
    return a+b
@dispatch(int,int,int)
def add(a,b,c):
    return a+b+c

s=add(3,2,4)
print(s)
s=add(3,4)
print(s)
s=add(3,4.4)
print(s)
```

https://github.com/wesselb/plum
https://pypi.org/project/multipledispatch/

End of Python Lab