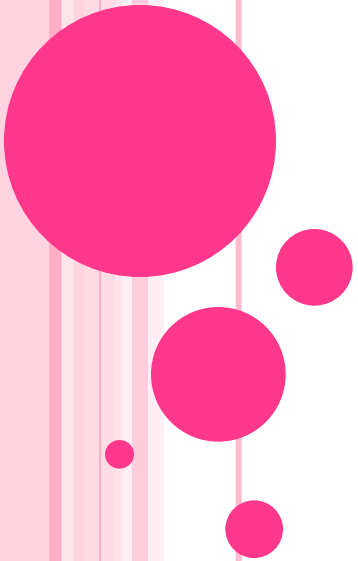
CHAPTER 2
FUNCTIONS, SCOPING AND ABSTRACTION



FUNCTIONS

- A function is similar to a program that contains a group of statements that are intended to perform a specific task.
- When there are several tasks to be performed , the programmer will write several functions.
- There are several 'built in' functions in Python to perform various tasks.

FUNCTIONS

- Example to display output, Python has print() function.
- To calculate square root there is sqrt() function and to calculate power value there is power() function.
- Similar to these functions, a programmer can also create his own functions called as ‘user defined functions’.

ADVANTAGES OF FUNCTIONS

- Once a function is written, it can be reused as and when required. So, functions are also called as reusable code.
  - Because of this reusability, the programmer can avoid code redundancy. Possible to avoid writing the same code again and again.
- Functions provide modularity for programming.
  - A module represents a part of the program.
  - A programmer divides the main task into smaller sub tasks called modules.

ADVANTAGES OF FUNCTIONS

- To represent each module, the programmer will develop a separate function.
- Then these functions are called from a main program to accomplish the complete task.
- Code maintenance will become easy because of functions.
- When a new feature has to be added to the existing software, a new function can be written and integrated into the software.
- When a particular feature is no more needed by the user, the corresponding function can be deleted or put into comments.

ADVANTAGES OF FUNCTIONS

- When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software. Thus, code debugging will become easy.
- Use of functions will reduce the length of the program.

DEFINING A FUNCTION

- Define a function using the keyword ***def*** followed by function name.
- After the function name, write parentheses ( ) which contain parameters.
- Function definition
  - def functionname(para1, ppara2,…):
    - “””function docstring”””
    - function statements

- Example:
  - def sum(a+b):
    - “””function finds sum of two numbers”””
    - c=(a+b)
    - print(c)

DEFINING A FUNCTION

- Function to add two values as:
  - `def sum(a,b):`
- 'def' represents starting of function definition.
- 'sum' is the name of the function.
- Parentheses ( ) are compulsory which denote it is a function and not a variable.
- We wrote two variables 'a' and 'b' which are called parameters.
- A parameter is a variable that receives data from outside the function.
- The function receives two variables from outside and stored in variables 'a' and 'b'.

DEFINING A FUNCTION

- After parentheses colon(:) represents beginning of the function body.
- Write a string as the first statement in the function body.
- This string is called “docstring” which gives information about the function.
- Docstrings are written inside triple double quotes or triple single quotes.
- Docstring is not compulsory all the time.
- After writing docstring , then write the logic of the function.
- The statements should be written with proper indentation.

CALLING A FUNCTION

- A function can run only when we call the function.
- To call the function pass the necessary values to the function in the parentheses as:
  - sum(10, 15)
- 'sum' function is called passing two values 10 and 15.
- When this statement is executed , the Python interpreter jumps to the function and copies the values 10 and 15 in to the parameters 'a' and 'b'.
- The values passed to the function are called 'arguments'.

CALLING A FUNCTION

```
1 def sum(a,b):
2     c=a+b
3     print('Sum=',c)
4 sum(10,15)
5 sum(1.5,10.5)
```

```
In [2]: %run "c:\users\mdhanl~1\appdata\local\temp\tmpegojbm.py"
Sum= 25
Sum= 12.0
```

CALLING A FUNCTION

- Once a function is written it is used again and again whenever required.
- So, functions are called reusable code.
- Integer type data is passed to the function when called for first time and float type data is passed for second time.
- During run time ‘a’ and ‘b’ may assume any type of data may be int or float or may be strings.
- This is called dynamic typing.

RETURNING RESULTS FROM A FUNCTION

- Return the result or output from the function using a 'return' statement in the body of the function.
- Example:
  - return c
  - return 100
  - return lst
  - return x,y,c
- When a function does not return any result, need not write the return statement in the body of the function.

RETURNING RESULTS FROM A FUNCTION

```
1 def sum(a,b):
2     c=a+b
3     return c
4 x=sum(10,15)
5 print('Sum is:',x)
6 y=sum(1.5,10.5)
7 print('Sum is:',y)
```

```
In [3]: %run "c:\users\mdhanl~1\appdata\local\temp\tmpoleubr.py"
Sum is: 25
Sum is: 12.0
```

RETURNING RESULTS FROM A FUNCTION

- Function to test whether a number is even or odd:

```
1 def even_odd(num):
2     if num%2==0:
3         print(num,"is even")
4     else:
5         print(num,"is odd")
6 even_odd(6)
7 even_odd(13)
```

```
In [4]: %run "c:\users\mdhanl~1\appdata\local\temp\tmpklhngj.py"
6 is even
13 is odd
```

FUNCTIONS AND SCOPING

FUNCTIONS AND SCOPING

- **Function Definitions:**
  - **Syntax:**
    - **def *name of function (list of formal parameters):***
      - ***body of function***
  - *Example, we could define the function max by the code:*
    - **def max(x, y):**
    - **if x > y:**
    - **return x**
    - **else:**
    - **return y**

FUNCTIONS AND SCOPING

- ***def*** is a reserved word that tells Python that a function is about to be defined.
- The function name (max in this example) is simply a name that is used to refer to the function.
- The sequence of names within the parameters following the function name (x,y in this example) are the **formal parameters** of the function.
- When the function is used, the formal parameters are bound to the **actual parameters** (referred as arguments) of the **function invocation** (referred as function call).

FUNCTIONS AND SCOPING

- Example, the invocation:
  - maxVal(3,4) binds x to 3 and y to 4.
- The function body is any piece of code.
- A special statement, **return** that can be used only within the body of a function.
- A function call is an expression, and like all expressions it has a value.
- That value is the value returned by the invoked function.
- Example maxVal(3,4)*maxVal(3,2) is 12 becoz the first invocation of maxVal is int 4 and second returns the int 3.
- The execution of a return statement terminates an invocation of the function.

FUNCTIONS AND SCOPING

- **Steps involved when a function is called:**
  1. The expressions that make up the actual parameters are evaluated, and the formal parameters of the function are bound to the resulting values. Example, the invocation maxVal(3+4, z) will bind the formal parameter x to 7 and the formal parameter y to whatever the value the variable z has when the invocation is evaluated.
  2. The point of execution moves from the point of invocation to the first statement in the body of the function.

FUNCTIONS AND SCOPING

- 3. The code in the body of the function is executed until either a return statement is encountered, in which case the value of the function invocation, or there are no more statements to execute, in which case the function returns the value None.
- 4. The value of the invocation is the returned value.
- 5. The point of execution is transferred back to the code immediately following the invocation.

LAMBDA FUNCTIONS

ANONYMOUS FUNCTIONS OR LAMBDAS

- Parameters provide something called **lambda abstraction.**
- A function without a name is called ‘anonymous function’.
- So far the functions we wrote were defined using the keyword ‘def’.
- But anonymous functions are not defined using ‘def ’.
- They are defined using the keyword lambda and hence they are called ‘ ***Lambda functions***’.

ANONYMOUS FUNCTIONS OR LAMBDAS

- Take a normal function that returns square of a given value:
  - **def square(x):**
    - **return x*x**
- The same function can be written as anonymous function as:
  - **lambda x: x*x**
- Observe the keyword ‘lambda’.
- This represents an anonymous function is being created.
- After that we have written an argument of the function i.e ‘x’.

ANONYMOUS FUNCTIONS OR LAMBDAS

- Then colon (:) represents the beginning of the function that contains an expression x*x.
- **Syntax of lambda function:**
  - **lambda argument_list : expression**
- If a function returns some value, we assign that value to a variable as:
  - **y = square(5)**
- But, lambda functions return a function and hence they should be assigned to a function as:
  - **f = lambda x : x*x**
- 'f' is the function name to which the lambda expression is assigned.
- If we call the function f() as:
  - **value = f(5)**
- Now 'value' contains the square value of 5 i.e. 25.

ANONYMOUS FUNCTIONS OR LAMBDAS

- **Python program to create a lambda function that returns a square value of a given number.**
- # lambda function to calculate square value
- f=lambda x:x*x #write lambda function
- value=f(5) #call lambda function
- print('Square of 5 = ',value) #display result

- **Output:**
  - **%run "D:/Python Programs/square.py"**
  - **Square of 5 = 25**

ANONYMOUS FUNCTIONS OR LAMBDAS

- **A lambda function to calculate the sum of two numbers.**
- **# lambda function to calculate sum of two numbers**
- f = lambda x , y: x+y
- result = f(2,12)
- print('Sum = ' , result)

- **Output:**
- **%run "D:/Python Programs/summ.py"**
- **Sum = 14**

ANONYMOUS FUNCTIONS OR LAMBDAS

- **A lambda function to find the bigger number in two given numbers.**
- # lambda function that returns bigger number
- max = lambda x, y : x if x>y else y
- a , b = input('Enter two numbers:').split(',')
- a=int(a)
- b=int(b)
- print('Bigger number = ', max(a,b))
- **Output:**
  - %run "D:/Python Programs/bigg.py"
  - Enter two numbers:10,20
  - Bigger number = 20

LAMBDAS WITH FILTER() FUNCTION

- **Using Lambdas with filter() Function:**
  - The filter() function is useful to filter out the elements of a sequence depending on the result of a function.
  - Supply a function and a sequence to the filter() function as:
    - **filter(function, sequence)**
  - 'function' represents a function name that may either return True or False.
  - Sequence represents a list, string or tuple.
  - The 'function' is applied to every element in the 'sequence' and when the function returns True , the element is extracted otherwise it is ignored.

LAMBDAS WITH FILTER() FUNCTION

- **Python program using filter() to filter out even numbers from a list.**

```
def is_even(x):
    if x%2==0:
        return True
    else:
        return False
lst=[10,25,45,46,71,99]
lst1=list(filter(is_even,lst))
print(lst1)
```

```
In [2]: %run "D:/Python Programs/filtereven.py"
[10, 46]
```

LAMBDAS WITH FILTER() FUNCTION

- **Passing Lambda function to filter() function.**

```
lst=[10,23,45,46,70,99]
lst1=list(filter(lambda x:(x%2 == 0), lst))
print(lst1)
```

```
In [3]: %run "D:/Python Programs/filtereven1.py"
[10, 46, 70]
```

LAMBDAS WITH MAP() FUNCTION

- **Using Lambdas with map() Function:**
  - The map() function is similar to filter() function but it acts on each element of the sequence and changes the elements.
  - Format of map() function:
    - **map(function,sequence)**
  - The 'function' performs a specified operation on all the elements of the sequence and the modified elements are returned which can be stored in another sequence.

LAMBDAS WITH MAP() FUNCTION

- **Python program to find squares of elements in a list.**

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

lst=[1,2,3,4,5]
lst1=list(map(squares,lst))
print(lst1)
```

```
In [4]: %run "D:/Python Programs/mapsquare.py"
[1, 4, 9, 16, 25]
```

LAMBDAS WITH MAP() FUNCTION

- A lambda function that returns squares of elements in a list.

```
lst=[1,2,3,4,5]
lst1=list(map(lambda x:x*x,lst))
print(lst1)
```

```
In [6]: %run "D:/Python Programs/mapsquare1.py"
[1, 4, 9, 16, 25]
```

LAMBDAS WITH MAP() FUNCTION

- **A lambda function that returns multiplication of elements in a list.**

```
lst1=[1,2,3,4,5]
lst2=[10,20,30,40,50]
lst3=list(map(lambda x,y:x*y,lst1,lst2))
print(lst3)
```

```
In [1]: %run "D:/Python Programs/map2.py"
[10, 40, 90, 160, 250]
```

LAMBDAS WITH REDUCE() FUNCTION

- **Using Lambdas with reduce() function:**
  - The reduce() function reduces a sequence of elements to a single value by processing the elements according to a function supplied.
  - The format of reduce() function:
    - reduce(function,sequence)
- **Lamda function to calculate products of elements of a list.**

```
from functools import reduce
lst=[1,2,3,4,5]
result=reduce(lambda x,y:x*y,lst)
print(result)
```

```
In [6]: %run "D:/Python Programs/reduce.py"
120
```

LAMBDAS WITH REDUCE() FUNCTION

- **Lamda function to calculate sum of numbers from 1 to 50 using reduce() function.**

```
from functools import reduce
sum=reduce(lambda a,b:a+b,range(1,51))
print(sum)
```

```
In [7]: %run "D:/Python Programs/reduce1.py"
1275
```

EXAMPLE PROGRAMS

- **Python program to calculate factorial values of numbers:**

```
1 def fact(n):
2     prod = 1
3     while n>=1:
4         prod*=n
5         n-=1
6     return prod
7
8 for i in range(1,11):
9     print('Factorial of {} is {}'.format(i, fact(i)))
```

```
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Factorial of 6 is 720
Factorial of 7 is 5040
Factorial of 8 is 40320
Factorial of 9 is 362880
Factorial of 10 is 3628800
```

EXAMPLE PROGRAMS

- **Python program to check whether a given number is prime or not:**

```
 1 def prime(n):
 2     x=1
 3     for i in range(2, n):
 4         if n%i == 0:
 5             x=0
 6             break
 7         else:
 8             x=1
 9     return x
10 num = int(input('Enter a number: '))
11 result = prime(num)
12 if result == 1:
13     print(num, ' is prime')
14 else:
15     print(num, ' is not prime')
```

```
In [9]: %run "D:/Python Programs/prime.py"

Enter a number: 4
4  is not prime

In [10]: %run "D:/Python Programs/prime.py"

Enter a number: 5
5  is prime
```

EXAMPLE PROGRAMS

- **Python program to check whether a given number is armstrong or not:**

```
# Python Program For Armstrong Number using Functions

def Armstrong_Number(Number):
        # Initializing Sum and Number of Digits
        Sum = 0
        Times = 0

        # Calculating Number of individual digits
        Temp = Number
        while Temp > 0:
                Times = Times + 1
                Temp = Temp // 10

        # Finding Armstrong Number
        Temp = Number
        for n in range(1, Temp + 1):
                Reminder = Temp % 10
                Sum = Sum + (Reminder ** Times)
                Temp //= 10
        return Sum
#End of Function

#User Input
Number = int(input("\nPlease Enter the Number to Check for Armstrong: "))

if (Number == Armstrong_Number(Number)):
    print("\n %d is Armstrong Number.\n" %Number)
else:
    print("\n %d is Not a Armstrong Number.\n" %Number)
```

EXAMPLE PROGRAMS

- **Python program to check whether a given number is armstrong or not:**

```
In [1]: %run "D:/Python Programs/armstrong.py"

Please Enter the Number to Check for Armstrong: 153

 153 is Armstrong Number.
```

```
In [3]: %run "D:/Python Programs/armstrong.py"

Please Enter the Number to Check for Armstrong: 100

 100 is Not a Armstrong Number.
```

FORMAL AND ACTUAL ARGUMENTS

FORMAL AND ACTUAL ARGUMENTS

- When a function is defined, it may have some arguments.
- These parameters are useful to receive values from outside of the function.
- They are called **'formal arguments'.**
- When we call the function, we should pass data or values to the function.
- These values are called **'actual arguments'.**

FORMAL AND ACTUAL ARGUMENTS

- def sum(a,b)
  - c=a+b
  - print(c)
- x=10, y=15
- sum(x,y)

- Here a,b are the formal arguments
- x,y are actual arguments

FORMAL AND ACTUAL ARGUMENTS

- **POSITIONAL ARGUMENTS:**
  - These arguments passed to a function in correct positional order.
  - The number of arguments and their positions in the function definition should match exactly with the number and position of the argument in the function call.
  - Example, take a function definition with two arguments as:
    - **def attach(s1,s2)**

FORMAL AND ACTUAL ARGUMENTS

- **Example:**
  - attach('New','York')
- It will print:
  - **NewYork**
- If we try to pass more than or less than 2 strings, there will be an error.
- Example:
  - attach('New','York','City')
  - **Then there will be an error displayed.**

FORMAL AND ACTUAL ARGUMENTS

- Python program to understand the positional arguments of a function:

```
def attach(s1,s2):
    s3=s1+s2
    print('Total string:  '+s3)
attach('New','York')
```

```
In [12]: %run "D:\Python Programs\positional.py"
Total string:  NewYork
```

FORMAL AND ACTUAL ARGUMENTS

- **Keyword Arguments:**
  - Keyword arguments are arguments that identify the parameters by their names.
  - Example, the definition of a function that displays grocery item and price can be written as:
    - **def grocery(item,price)**
  - At the time of calling this function, we have to pass two values and we can mention which value is for what.
  - Example:
    - **grocery(item='Sugar', price=50.75)**
  - We are mentioning a keyword 'item' and its value and then another keyword 'price' and its value.
  - These keywords are nothing but the parameter names which receive these values.

FORMAL AND ACTUAL ARGUMENTS

- It is possible to change the order of the argument as:
  - grocery(price=87.00, item='oil')
- Even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value.

```
def grocery(item,price):
    print('Item=%s' % item)
    print('Price=%.2f' % price)
grocery(item='Rice',price=30.00)
grocery(price=30.00,item='Oil')
```

```
In [14]: %run "D:\Python Programs\keyarg.py"
Item=Rice
Price=30.00
Item=Oil
Price=30.00
```

FORMAL AND ACTUAL ARGUMENTS

- **Default Arguments:**
  - Mention some default value for the function parameters in the definition.
  - Example:
    - **def grocery(item. Price=40.00):**
  - The first argument is 'item' whose default value is not mentioned.
  - The second argument is 'price' and its default value is mentioned as 40.00.
  - At the time of calling the function, if we do not pass the value then the default value of 40.00 is taken.
  - If we mention the 'price' value then that mentioned value is utilized.
  - A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

FORMAL AND ACTUAL ARGUMENTS

- Python program to understand the use of default arguments in a function.

```
def grocery(item, price=40.00):
    print('Item = %s' % item)
    print('Price = %.2f' % price)
grocery(item='Sugar', price=50.75)
grocery(item='Sugar')
```

```
In [16]: %run "D:/Python Programs/defaultt.py"
Item = Sugar
Price = 50.75
Item = Sugar
Price = 40.00
```

FORMAL AND ACTUAL ARGUMENTS

- **Variable Length Arguments:**
  - Programmer does not know how many values a function may receive.
  - It is not possible to decide how many arguments to be given in the function definition.
  - Writing a function to add two numbers:
    - **add(a,b)**
  - But the user who is using this function may want to use this function to find sum of three numbers.
  - The user may provide 3 arguments to this function as:
    - **add(10,20,30)**
  - Then the add() function will fail and error will be displayed.

FORMAL AND ACTUAL ARGUMENTS

- To accept ‘n’ arguments a variable length argument is used in function definition.
- A variable length argument is an argument that can accept any number of values.
- The variable length argument is written with a ‘*’ symbol before it in the function definition as:
  - def add(farg, *args)
- ‘farg’ is the formal argument and ‘args’ represents variable length argument.
- Pass 1 or more values to this ‘args’ and it will store them all in a tuple.
- A tuple is like a list where a group of elements are stored.

FORMAL AND ACTUAL ARGUMENTS

- **Python program to show variable length argument and its use.**
- **def add(farg, *args)**
  - **print('Formal argument= ', farg)**
  - **sum=0**
  - **for i in args:**
    - **Sum+=i**
  - **print('Sum of all numbers= ',(farg+sum))**
- **add(5,10)**
- **add(5,10,15,20)**
- **Output:**
  - **Formal argument= 5**
  - **Sum of all numbers= 15**
  - **Formal argument= 5**
  - **Sum of all numbers= 50**

KEYWORD ARGUMENTS AND DEFAULT VALUES

KEYWORD ARGUMENTS & DEFAULT VALUES

- Keyword Arguments and Default Values:
  - In Python, there are two ways that formal parameters get bound to actual parameters.
  - The most common method, which is the only one we have used thus far, is called **positional**—the first formal parameter is bound to the first actual parameter, the second formal to the second actual, etc.
  - Python also supports what it calls **keyword arguments**, in which formals are bound to actuals using the name of the formal parameter.

KEYWORD ARGUMENTS & DEFAULT VALUES

- Consider the function:

```
def printName(firstName, lastName, reverse):
    if reverse:
        print lastName + ', ' + firstName
    else:
        print firstName, lastName
```

Figure 4.2 Function that prints a name

- The function printName assumes that firstName and lastName are strings and that reverse is a Boolean.
- If reverse == True, it prints lastName, firstName, otherwise it prints firstName lastName.

KEYWORD ARGUMENTS & DEFAULT VALUES

- Each of the following is an equivalent invocation of printName:
  - printName(‘Core', 'Python', False)
  - printName(‘Core', 'Python', False)
  - printName(‘Core', 'Python', reverse = False)
  - printName(‘Core', lastName = 'Python', reverse = False)
  - printName(lastName=‘Python', firstName=‘Core', reverse=False)
- Though the keyword arguments can appear in any order in the list of actual parameters, it is not legal to follow a keyword argument with a non-keyword argument.
- Therefore, an error message would be produced by
  - printName(‘Core', lastName = 'Python', False)

KEYWORD ARGUMENTS & DEFAULT VALUES

- Keyword arguments are commonly used in conjunction with **default parameter values**.
- For example, write
  - def printName(firstName, lastName, reverse = False):
  - if reverse:
  - print lastName + ', ' + firstName
  - else:
  - print firstName, lastName
- Default values allow programmers to call a function with fewer than the specified number of arguments.

KEYWORD ARGUMENTS & DEFAULT VALUES

- **Example:**
  - printName('Core', 'Python')
  - printName('Core', 'Python', True)
  - printName('Core', 'Python', reverse = True)
- It will print
  - Core Python
  - Python, Core
  - Python, Core
- The last two invocations of printName are semantically equivalent.

SCOPING

- Look at another small example,
  - def f(x): #name x used as formal parameter
  - y = 1
  - x = x + y
  - print 'x =', x
  - return x
  - x = 3
  - y = 2
  - z = f(x) #value of x used as actual parameter
  - print 'z =', z
  - print 'x =', x
  - print 'y =', y

FUNCTIONS AND SCOPING

- **Output:**
  - **x = 4**
  - **z = 4**
  - **x = 3**
  - **y = 2**
- At the call of f, the formal parameter x is locally bound to the value of the actual parameter x. It is important to note that though the actual and formal parameters have the same name, they are not the same variable.
- Each function defines a new **name space**, also called a **scope**.
- The formal parameter x and the **local variable** y that are used in f exist only within the scope of the definition of f.

FUNCTIONS AND SCOPING

- The assignment statement x = x + y within the function body binds the local name x to the object 4.
- The assignments in f have no effect at all on the bindings of the names x and y that exist outside the scope of f.
- At top level, i.e., the level of the shell, a **symbol table** keeps track of all names defined at that level and their current bindings.
- When a function is called, a new symbol table called a **stack frame** is created.
- This table keeps track of all names defined within the function including the formal parameters and their current bindings.

FUNCTIONS AND SCOPING

- If a function is called from within the function body, yet another stack frame is created.
- When the function completes, its stack frame goes away.
- In Python, one can always determine the scope of a name by looking at the program text.
- This is called **static** or **lexical scoping**.

RECURSION

- A function that calls itself is known as 'Recursive Function'.
- Example, we can write the factorial of 3 as :
  - factorial(3) = 3*factorial(2)
  - factorial(2) = 2*factorial(1)
  - factorial(1) = 2*factorial(0)
- The results will be as:
  - factorial(3) = 3*factorial(2)
  - = 3*2*factorial(1)
  - =3*2*1*factorial(0)
  - =6
- From the above statements we can write the formula to calculate factorial of any number 'n' as:
  - **factorial(n) = n * factorial(n-1)**

RECURSION

```
def factorial(n):
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    return result
for i in range(1,12):
    print('Factorial of {} is {}'.format(i,factorial(i)))
```

```
In [4]: %run "D:/Python Programs/factrecursive.py"
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Factorial of 6 is 720
Factorial of 7 is 5040
Factorial of 8 is 40320
Factorial of 9 is 362880
Factorial of 10 is 3628800
Factorial of 11 is 39916800
```

GLOBAL VARIABLES

GLOBAL VARIABLES

- When we declare a variable inside a function, it becomes a local variable.
- A local variable is a variable whose scope is limited only to that function where it is created.
- A local variable is available only in that function and not outside of that function.
- **Example:**

```
def myfunction():
    a=1
    a+=1
    print(a)
myfunction()
print(a)
```

GLOBAL VARIABLES

- **Output:**

```
In [2]: %run -i "D:/Python Programs/globe1.py"
2

In [3]: ---------------------------------------------------------------------
NameError                                 Traceback
D:/Python Programs/globe1.py in <module>()
      4     print(a)
      5 myfunction()
----> 6 print(a)

NameError: name 'a' is not defined
```

- The variable ‘a’ is declared inside the myfunction().
- Once we come out of the function, the variable ‘a’ is removed from the memory and it is not available.

GLOBAL VARIABLES

- When a variable is declared above a function, it becomes global variable.
- Such variables are available to all the functions which are written after it.

```
a=1  # global variable
def myfunction():
    b=2
    print('a= ',a) # display global variable
    print('b= ',b) # display local variable
myfunction()
print(a)   #available
print(b)   #error not available
```

GLOBAL VARIABLES

- **Output:**

```
In [3]: %run "D:/Python Programs/globe2.py"
a=  1
b=  2
1
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
D:\Python Programs\globe2.py in <module>()
      6 myfunction()
      7 print(a)
----> 8 print(b)

NameError: name 'b' is not defined
```

GLOBAL VARIABLES

- Sometimes, the global variable and local variable may have the same name.
- In that case the function by default refers to the local variable and ignores the global variable.
- So, the global variable is not accessible inside the function, but outside of it, it is accessible.

```
a=1  #global var
def myfunction():
    a=2  # local var
    print('a= ',a)
myfunction()
print('a= ',a)
```

```
In [4]: %run "D:/Python Programs/glolocal.py"
a=  2
a=  1
```

GLOBAL VARIABLES

- When a programmer wants to use the global variable inside a function, he can use the keyword ‘global’ before the variable in the beginning of the function body as:
  - global a
  - In this way, the global variable is made available to the function and the programmer can work with it as he wishes.

GLOBAL VARIABLES

- Python Program to access global variable inside a function and modify it

```
a=1 #this is global var
def myfunction():
    global a
    print('global a =',a) # display global var
    a=2 #modify global var value
    print('modified a =',a) #display new value
myfunction()
print('global a =',a) #display mofdified value
```

```
In [5]: %run "D:/Python Programs/globekey.py"
global a = 1
modified a = 2
global a = 2
```

GLOBAL VARIABLES

- When the global variable name and local variable names are same, the programmer will face difficulty to differentiate between them inside a function.
- Example there is a global variable ‘a’ with some value declared above the function.
- A local variable with the same name ‘a’ with some other value inside the function.
- Consider the following code:
  - a=1 #global var
  - def myfunction():
    - a = 2 # local var
- If the programmer wants to work with global variable, how is it possible?

GLOBAL VARIABLES

- If the ‘global’ keyword is used, the he can access only global variable and the local variable is no more available.
- The globals() function will solve this problem.
- This is a built in function which returns a table of current global variables in the form of a dictionary.
- Using this function we can refer to the global variable ‘a’ as: globals()[‘a’].
- This value can be assigned to another variable, say ‘x’ and the programmer can work with that value.

GLOBAL VARIABLES

- Python program to get a copy of global variable into a function and work with it.

```
a=1
def myfunction():
    a=2
    x=globals()['a']
    print('global a= ',x)
    print('local a= ',a)
myfunction()
print('global a= ',a)
```

```
In [6]: %run "D:/Python Programs/globalss.py"
global a=  1
local a=  2
global a=  1
```

MODULES

MODULES

- A module represents a group of classes, methods, functions and variables.
- While we are developing software, there may be several classes, methods and functions.
- First group them depending on their relationship into various modules and later use these modules in other programs.
- When a module is developed, it can be reused in any program that needs that module.
- Python has several built-in modules like sys, io, time etc.
- We van also create our own modules and use them whenever we need them.
- Once a module is created, any programmer in the project team can use that module .

MODULES

- Modules will make software development easy and faster.
- We will create our own module by the name ‘employee’ and store the functions da(), hra(), pf() and itax() in that module.

```
def da(basic):
    da=basic*80/100
    return da

def hra(basic):
    hra=basic*15/100
    return hra

def pf(basic):
    pf=basic*12/100
    return pf

def itax(gross):
    tax=gross*0.1
    return tax
```

MODULES

- The module name is ‘employee.py’.
- This module contains 4 functions which can be used in any program by importing this module.
- To import the module write:
  - **import employee**
- Refer the functions by adding the module name as:
  - employee.da(),employee.hra(),employee.pf(), employee.itax().
- This is little bit cumbersome and we use another type of import statement:
  - **from employee import ***
- We can refer all the functions using the module name simply as da(), hra(), pf() and itax().

MODULES

- Python program where we are using the 'employee' module and calculating the gross and net salaries of an employee.

```
from employee import *

basic = float(input('Enter basic salary: '))

gross=basic+da(basic)+hra(basic)
print('Your gross salary: {:10.2f}' . format(gross))

net=gross-pf(basic)-itax(gross)
print('Your net salary: {:10.2f}' . format(net))
```

```
In [9]: %run "D:/Python Programs/empfun.py"

Enter basic salary: 15000
Your gross salary:   29250.00
Your net salary:   24525.00
```

MODULES

- A **module** is a .py file containing Python definitions and statements.
- We could create, for example, a file circle.py containing

```
pi=3.14159
def area(radius):
    return(pi*(radius**2))

def circumference(radius):
    return 2*pi*radius

def sphereSurface(radius):
    return 4.0*area(radius)

def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

MODULES

- A program gets access to a module through an import statement.
- So, for example, the code:

```
import circle
pi=3
print(pi)
print (circle.pi)
print (circle.area(3))
print (circle.circumference(3))
print (circle.sphereSurface(3))
```

- will print

```
In [13]: %run "D:/Python Programs/circlefun.py"
3
3.14159
28.27431
18.849539999999998
113.09724
```

MODULES

- Modules are typically stored in individual files. Each module has its own private symbol table.
- Within circle.py we access objects (e.g., pi and area) in the usual way.
- Executing import M creates a binding for module M in the scope in which the importation occurs.
- Therefore, in the importing context we use dot notation to indicate that we are referring to a name defined in the imported module.
- Outside of circle.py, the references pi and circle.pi can refer to different objects.

FILES

FILES

- Every computer system uses **files** to save things from one computation to the next.
- Python provides many facilities for creating and accessing files.
- Each operating system (e.g., Windows and MAC OS) comes with its own file system for creating and accessing files.
- **Python achieves operating-system independence by accessing files through something called a file handle.**
- The code
  - **nameHandle = open('kids', 'w')**
- instructs the operating system to create a file with the name kids, and return a file handle for that file.

FILES

- The argument 'w' to open indicates that the file is to be opened for writing.
- The following code opens a file, uses the **write** method to write two lines, and then closes the file.
- It is important to remember to close the file when the program is finished using it.
- Otherwise there is a risk that some or all of the writes may not be saved.
  - **nameHandle = open('kids', 'w')**
  - **for i in range(2):**
  - **name = raw_input('Enter name: ')**
  - **nameHandle.write(name + '\n')**
  - **nameHandle.close()**

FILES

- In a string, the character “\” is an escape character used to indicate that the next character should be treated in a special way.
- In this example, the string '\n' indicates a new line character.
- We can now open the file for **reading** (using the argument 'r'), and print its contents.
- Since Python treats a file as a sequence of lines, we can use a for statement to iterate over the file’s contents.
  - **nameHandle = open('kids', 'r')**
  - **for line in nameHandle:**
  - **print line**
  - **nameHandle.close()**

FILES

- If we had typed in the names David and Andrea, this will print
  - David
  - Andrea
- The extra line between David and Andrea is there because print starts a new line each time it encounters the '\n' at the end of each line in the file.
- We could have avoided printing that by writing print line[:-1].

FILES

- Now consider
  - **nameHandle = open('kids', 'w')**
  - **nameHandle.write('Michael\n')**
  - **nameHandle.write('Mark\n')**
  - **nameHandle.close()**
  - **nameHandle = open('kids', 'r')**
  - **for line in nameHandle:**
  - **print line[:-1]**
  - **nameHandle.close()**
- It will print
  - **Michael**
  - **Mark**

FILES

- Notice that we have overwritten the previous contents of the file kids.
- If we don't want to do that we can open the file for **appending** (instead of writing) by using the argument 'a'.
- For example, if we now run the code
  - **nameHandle = open('kids', 'a')**
  - **nameHandle.write('David\n')**
  - **nameHandle.write('Andrea\n')**
  - **nameHandle.close()**
  - **nameHandle = open('kids', 'r')**
  - **for line in nameHandle:**
  - **print line[:-1]**
  - **nameHandle.close()**

FILES

- It will print
  - **Michael**
  - **Mark**
  - **David**
  - **Andrea**

FILES

Some of the common operations on files are summarized in Figure 4.11.

`open(fn, 'w')` fn is a string representing a file name. Creates a file for writing and returns a file handle.
`open(fn, 'r')` fn is a string representing a file name. Opens an existing file for reading and returns a file handle.
`open(fn, 'a')` fn is a string representing a file name. Opens an existing file for appending and returns a file handle.
`fh.read()` returns a string containing the contents of the file associated with the file handle fh.
`fh.readline()` returns the next line in the file associated with the file handle fh.
`fh.readlines()` returns a list each element of which is one line of the file associated with the file handle fh.
`fh.write(s)` write the string s to the end of the file associated with the file handle fh.
`fh.writeLines(S)` S is a sequence of strings. Writes each element of S to the file associated with the file handle fh.
`fh.close()` closes the file associated with the file handle fh.

Figure 4.11 Common functions for accessing files

Types of files in python

- There are two types of files:
  - Text Files
  - Binary Files
- Text file stores data in the form of characters.
- Binary file store entire data in the form of bytes that is group of 8 bits each.
- When the data is retrieved form the binary file, the programmer can retrieve the data as bytes.
- Binary files can be used to store text, image, audio and video.

OPENING A FILE

- open() function to open a file.
- **Filehandler=open(“filename”,”openmode”,”buffering”);**
- filename – name on which the data is stored.
- Use any name to reflect the data.
- Several opening modes:
- **w,**
- **r,**
- **a,**
- **w+,**
- **r+,**
- **a+,**
- **x**

OPENING A FILE

File Open mode	Description
w	Write data into file. If any data is present in the file, it would be deleted and present data will be stored.
r	To read data from the file. The file pointer is positioned at the beginning of the file.
a	Append data to the file. Adding at the end of existing data. File pointer is placed at the end of the file. If the file does not exist , it will create a new file for writing data.
w+	To write and read data of a file. The previous data in the file will be deleted.
r+	To read and write data into a file. The previous data in the file will not be deleted. File pointer is placed at beginning of file.
a+	Append and read data of a file. File pointer will be at the end of the file if the file exists. If the file does not exist, it creates a new file for reading and writing.
x	Open file in exclusive creation mode. File creation fails if the file already exists.

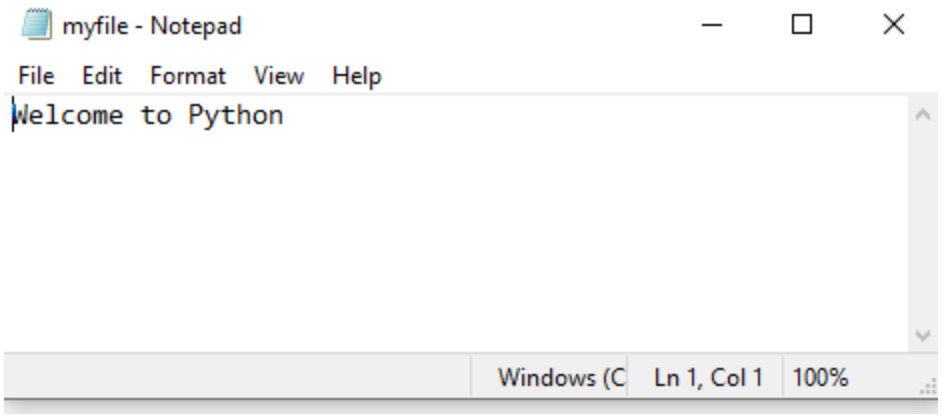
OPENING A FILE

- Attach ‘b’ to the read modes it represents modes for binary files.
- wb, rb, ab, w+b, r+b, a+b.
- Buffer represents a temporary block of memory.
- ‘buffering’ is an optional integer used to set the size of the buffer for the file.
- In binary mode pass 0 as buffering to inform not to use any buffering.
- In text mode use 1 for buffering to retrieve data from the file one line at a time.
- If no buffering integer is mentioned, then the default buffer size used is 4096 or 8192 bytes.

OPENING A FILE

- f = open(“myfile.txt”, “w”)
- ‘f’ is the file handler.
- It refers to the file with the name “myfile.txt” that is opened in “w” mode.
- We can write data into the file but cannot read data from this file.

```
f=open('E:\Academic Files\Subjects\Python\Python Programs\myfile.txt','w')
str=input('Enter text:')
f.write(str)
f.close()
```



OPENING A FILE

```
f=open('E:\Academic Files\Subjects\Python\Python Programs\myfile.txt','r')
str=f.read()
print(str)
f.close()
```

```
In [7]: %run "E:/Academic Files/Subjects/Python/Python Programs/readfile.py"
Welcome to Python
```

- str = f.read(n)
- 'n' represents the number of bytes to be read from the beginning of file.

WORKING WITH TEXT FILES CONTAINING STRINGS

- To store group of strings into a text file, use write() method inside a loop.
- Example to store strings into the file as long as the user does not type @ symbol.

```
f=open('E:\Academic
Files\Subjects\Python\Python
Programs\myfile.txt','w')
print('Enter text (@ at the end):')
while str != '@':
str=input()
if(str != '@'):
f.write(str+"\n")
f.close()
```

WORKING WITH TEXT FILES CONTAINING STRINGS

- Python Program to read all the strings from the text file and display them.

```
f=open('E:\Academic Files\Subjects\Python\Python Programs\myfile.txt','r')
print('The file contents are:')
str=f.read()
print(str)
f.close()
```

WORKING WITH TEXT FILES CONTAINING STRINGS

- Program to append data to the existing file and to display the data.
- f=open('myfile.txt','a+')
- Use write() method to append strings to the file.
- After writing the data without closing the file, we can read strings from the file.
- Place the file handler to the beginning of the file using seek() method
  - f.seek(offset, fromwhere)
  - 'offset' represents how many bytes to move.
  - 'fromwhere' -> fromwhich position to move.
  - **f.seek(10,0)** will position the handler at 10^{th} byte from beginning of the file.

WORKING WITH TEXT FILES CONTAINING STRINGS

- Program to append data to the existing file and to display the data.
- f=open('myfile.txt','a+')
- Use write() method to append strings to the file.
- After writing the data without closing the file, we can read strings from the file.
- Place the file handler to the beginning of the file using seek() method
  - f.seek(offset, fromwhere)
  - 'offset' represents how many bytes to move.
  - 'fromwhere' -> fromwhich position to move.
  - **f.seek(10,0)** will position the handler at 10^{th} byte from beginning of the file.

WORKING WITH TEXT FILES CONTAINING STRINGS

- Program to append data to the existing file and to display the data.

```
f=open('C:\DM\Python
Programs\myfile.txt','a+')
print('Enter text to append (@ the end):')
while str != '@':
str = raw_input()
if(str != '@'):
f.write(str+"\n")
f.seek(0,0)
print('The file contents are:')
str = f.read()
print(str)
f.close()
```

```
In [1]: %run "C:\DM\Python Programs\appendd.py"
Enter text to append (@ the end):

jgjr

@
The file contents are:
grrgrjgjr
```

KNOWING WHETHER A FILE EXISTS OR NOT

- The operating system module (os) has a sub module by the name 'path' that contains a method isfile().
- This method can be used to know whether a file that we are opening really exists or not.
- os.path.isfile(fname) gives True if the file exists otherwise False.
  - if os.path.isfile(fname):
    - f = open(fname,'r')
  - else:
    - print(fname+ 'does not exist')
    - sys,exit()

KNOWING WHETHER A FILE EXISTS OR NOT

```
1 import os,sys
2 fname=raw_input('Enter filename:')
3 if os.path.isfile(fname):
4     f = open(fname,'r')
5 else:
6     print(fname+ 'file does not exists')
7     sys.exit()
8 print('The file contents are:')
9 str= f.read()
10 print(str)
11 f.close()
```

```
In [19]: %run "C:/DM/Python Programs/fileexist.py"

Enter filename:C:\DM\Python Programs\myfile.txt
The file contents are:
This line is added
second line
```

KNOWING WHETHER A FILE EXISTS OR NOT

```
1 import os,sys
2 fname=raw_input('Enter filename:')
3 if os.path.isfile(fname):
4     f = open(fname,'r')
5 else:
6     print(fname+ 'file does not exists')
7     sys.exit()
8 cl=cw=cc=0
9 for line in f:
10    words=line.split()
11    cl+=1
12    cw+=len(words)
13    cc+=len(line)
14 print('No.of lines:', cl)
15 print('No.of words:', cw)
16 print('No.of characters:', cc)
17 f.close()
```

```
In [21]: %run "C:/DM/Python Programs/countt.py"

Enter filename:C:\DM\Python Programs\myfile.txt
('No.of lines:', 2)
('No.of words:', 6)
('No.of characters:', 31)
```

Working with binary files

- Binary files are used to read or write images, audio and video files.
- To open a binary file use ‘rb’ mode.
- ‘b’ is attached to ‘r’ to represent that it is a binary file.
- Use ‘wb’ mode to write bytes into a binary file.
- To read bytes from a binary file use read() method and to write bytes into a binary file use write() method.

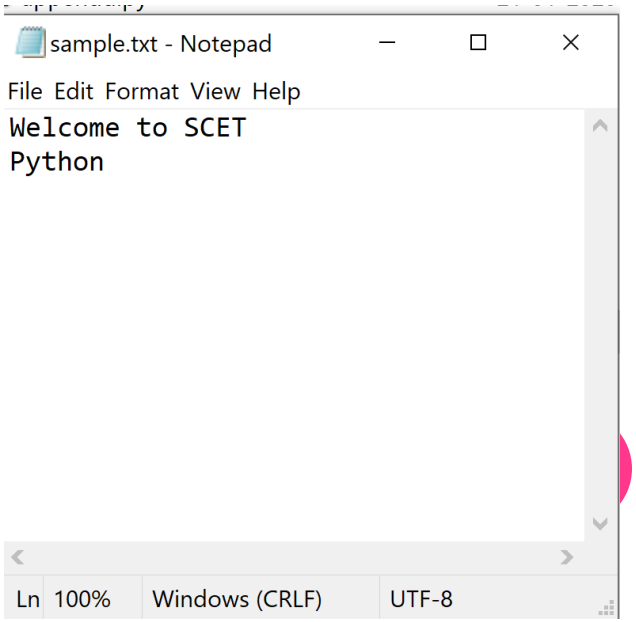
Working with binary files

```
1 f1=open('C:\\DM\\Python Programs\\cat.jpg','rb')
2 f2=open('C:\\DM\\Python Programs\\new.jpg','wb')
3 bytes=f1.read()
4 f2.write(bytes)
5 f1.close()
6 f2.close()
```

WITH STATEMENT

- The ‘with’ statement can be used while opening a file.
- Advantage is it will take are of closing a file which is opened by it.
- **Need not close the file explicitly.**
  - **with open(“filename”, “openmode”) as fileobject:**

```
1 with open('C:\DM\Python Programs\sample.txt','w') as f:
2     f.write('Welcome to SCET\n')
3     f.write('Python')
```



WITH STATEMENT

- Python program to use ‘with’ to open a file and read data from it.

```
1 with open('C:\DM\Python Programs\sample.txt','r') as f:
2     for line in f:
3         print(line)
```

```
In [28]: %run "C:/DM/Python Programs/withopen.py"
Welcome to SCET

Python
```

SEEK() AND TELL() METHODS

- To know the position of the file pointer, tell() method is used.
- It returns the current position of the file pointer from the beginning of the file.
  - **n=f.tell()**
- f- file handler
- n – integer that represents the byte position where the file pointer is positioned.
- To move the file pointer to another position, use seek() method.
  - **f.seek(offset, fromwhere)**
  - **Offset – How many bytes to move.**
  - **Fromwhere – Values can be 0,1,2**

SEEK() AND TELL() METHODS

- 0 – Beginning of file.
- 1 – Current position
- 2 – Ending of file.
- Default value of 'fromwhere' is 0 that is from beginning of file.
  - f.seek(10) will move file pointer to 11^{th} byte from the beginning of the file.
  - f.seek(-10,2) move the file pointer to the 9^{th} byte (-10+1) from the ending of the file.

SEEK() AND TELL() METHODS

```
with open('E:\\Academic
Files\\Subjects\\Python\\Python
Programs\\line.bin','r+b') as f:
f.write(b'Amazing Python')
f.seek(3)
print(f.read(2))
print(f.tell())
f.seek(-6,2)
print(f.read(1))
print(f.tell())
```

Random accessing of binary files

- Data in binary file is stored in the form of continuous bytes.
- Take a binary file having 1000 byte sof data.
- To access last 10 bytes no need to search the file byte by byte from the beginning.
- Use seek() method:
  - f.seek(990)
- Read the last 10 bytes using read() method:
  - f.read(10)
- **Directly going to any byte in the binary file is called random accessing.**

Random accessing of binary files

- Problem with binary files:
  - Accept data in the form of bytes or in binary format.
  - If we store string into a binary it will end up with an error.
- Reason:
  - We are trying to store strings into a binary file without converting them into binary format.
  - Solution: Convert the ordinary strings into binary format before they are stored into binary file.
  - Prefix character'b' before the string.
  - Use **encode() method to convert a string variable into binary format.**
  - Read a **string from binary file convert it into ordinary text using decode() method.**

Random accessing of binary files

- Consider a binary file and we are storing a groups of strings.
- The data of a binary file is viewed as a group of records with fixed length.
- Example: Take 20 bytes (or characters) as one record and store several such records into the file.
- Consider storing names of cities in cities.bin file.
- Record length = 20.
- If the city name entered by the user is less than 20 characters length, then remaining characters in the record will be filled with spaces.

Random accessing of binary files

- Example:
  - ln = len(city)
    - For example length of city is 5.
  - city = city + (20-ln)*' ' -> Add 15 spaces to end of city name.

```
1 reclen = 20
2
3 with open('C:\DM\Python Programs\cities.bin',"wb") as f:
4     n=int(raw_input('How many entreis?'))
5
6     for i in range(n):
7         city = raw_input('Enter city name:')
8         ln = len(city)
9         city = city + (reclen-ln)*' '
10
11        city = city.encode()
12        f.write(city)
13
```

Random accessing of binary files

- Python program to randomly access a record from a binary file.

```
1 reclen = 20
2
3 with open('C:\DM\Python Programs\cities.bin',"rb") as f:
4     n=int(raw_input('Enter record number:'))
5
6     f.seek(reclen * (n-1))
7
8     str = f.read(reclen)
9     print(str.decode())
10
```

Random accessing of binary files

- Python program to update or modify record in a binary file.
-

```
1  import os
2  reclen = 20
3  size = os.path.getsize('E:\Academic Files\Subjects\Python\Python Programs\cities.bin')
4  print('Size of file = {} bytes'.format(size))
5  n=int(size/reclen)
6  print('No.of records = {} '.format(n))
7  with open('E:\Academic Files\Subjects\Python\Python Programs\cities.bin','r+b') as f:
8      name  = input('Enter city name:')
9      name = name.encode()
10
11     newname = input('Enter new name:')
12     ln = len(newname)
13     newname = newname+(20-ln)* ''
14     newname = newname.encode()
```

Random accessing of binary files

- Python program to update or modify record in a binary file.

```
16      position = 0
17      found = False
18
19      for i in range(n):
20          f.seek(position)
21          str = f.read(20)
22          if name in str:
23              print('Updated record no:',(i+1))
24              found=True
25              f.seek(-20,1)
26              f.write(newname)
27          position+=reclen
28      if not found:
29          print('City not found')
```

Random accessing of binary files

- Python program to delete a specific record from a binary file.

```
1 import os
2 reclen = 20
3 size = os.path.getsize('E:\Academic Files\Subjects\Python\Python Programs\cities.bin')
4 print('Size of file = {} bytes'.format(size))
5 n=int(size/reclen)
6
7 f1 = open('E:\Academic Files\Subjects\Python\Python Programs\cities.bin','rb')
8 f2 = open('E:\\Academic Files\\Subjects\\Python\\Python Programs\\file2.bin','wb')
9 city= input('Enter city name to delete')
10
11 ln = len(city)
12 city = city+(reclen-ln)*' '
13
14 city = city.encode()
15
16 for i in range(n):
17     str = f1.read(reclen)
18     if(str!=city):
19         f2.write(str)
20 print('Record deleted..')
21
22 f1.close()
23 f2.close()
```

Random accessing of binary files using mmap

- *mmap* -> memory mapped file.
- A module in python that is useful to map or link to a binary file and manipulate the data of the file as we do with strings.
- Once a binary file is created with data, it can be viewed as strings and can be manipulated using mmap module.
  - **mm = mmap.mmap(f.fileno(),0)**
- It will map the currently opened file ‘f’ with file object ‘mm’.

Random accessing of binary files using mmap

- Arguments of mmap() method
  - f.fileno() is the handle to file object ‘f’.
  - ‘f’ -> Actual binary file that is being mapped.
  - Second argument zero(0) represents total size of the file should be considered for mapping.
- The entire file represented by the file object ‘f’ is mapped in memory to the object ‘mm’.
- Read the data from file using read or readline() methods as:
  - print(mm.read())
  - print(mm.readline())

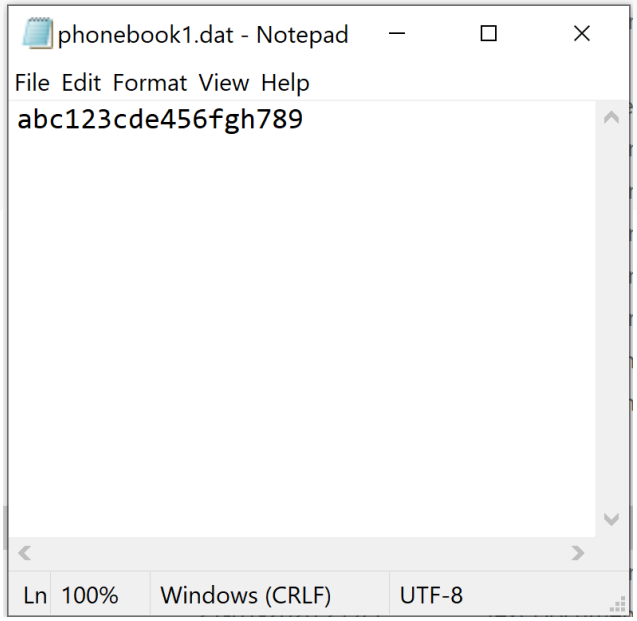
Random accessing of binary files using mmap

- Retrieve data from the file using slicing operator:
  - **print(mm[5:])**
  - **print(mm[5:10])**
- Modify the data of file using slicing:
  - **mm[5:10] = str**
- Use find() method that returns the first position of a string in file:
  - **n= mm.find(name)**
- seek() method to position the file pointer to any position:
  - **mm.seek(10,0)**

Random accessing of binary files using mmap

- Python program to create phonebook with names and phone numbers.

```
1 with open('C:\DM\Python Programs\phonebook1.dat','wb') as f:
2     n = int(raw_input('Enter no.of entries:'))
3
4     for i in range(n):
5         name = raw_input('Enter name:')
6         phone = raw_input('Enter phone:')
7         name=name.encode()
8         phone=phone.encode()
9         f.write(name+phone)
10
```



ZIPPING AND UNZIPPING FILES

- Software such as winzip provide zipping and unzipping of file data.
- The file contents are reduced and size will be reduced.
- The format of data will be changed making it unreadable.
- The algorithm finds out which bit pattern is most repeated in the original file and which bit pattern is least repeated.
- In Python the module zipfile contains ZipFile class that zip or unzip a file contents.

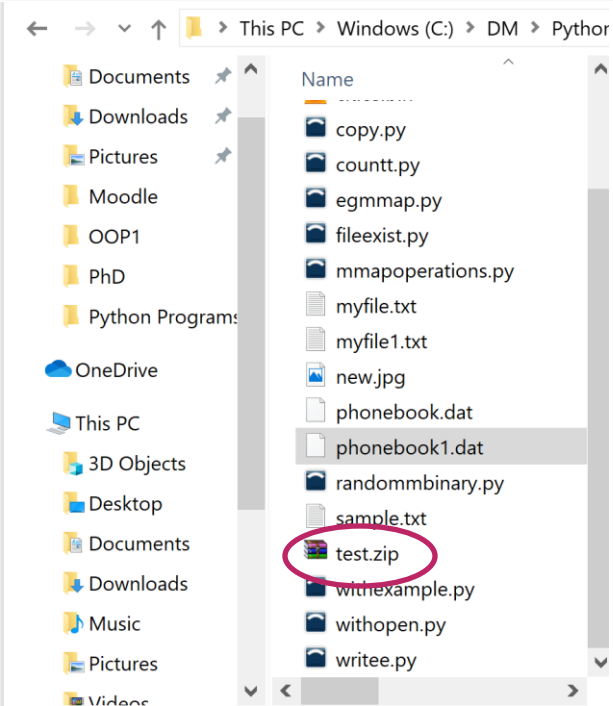
ZIPPING AND UNZIPPING FILES

- **f = ZipFile('test.zip','w',ZIP_DEFLATED)**

- 'f' is the ZipFile class object to which test.zip file name is passed.
- Next step is to add the filenames that are to be zipped using write() method as:
  - f.write('file1.txt')
  - f.write('file2.txt')
- These two .txt file will be compressed and stored in test.zip file.

ZIPPING AND UNZIPPING FILES

```
1 from zipfile import *
2 f = ZipFile('C:\\DM\Python Programs\\test.zip','w',ZIP_DEFLATED)
3
4 f.write('C:\DM\Python Programs\myfile.txt')
5 f.write('C:\DM\Python Programs\myfile1.txt')
6
7 print('test.zip created..')
8 f.close()
```



ZIPPING AND UNZIPPING FILES

- To unzip the contents of the compressed files and get back their original contents, use ZipFile class object in read mode:
  - z = ZipFile('testzip','r')
- Testzip is the filename that contains the compressed files.
- To extract all the files from the zip file object use extractall() method.
- extractall() method will extract all the files in current directory.
- To extract in another directory mention the directory path in extractall() method.

ZIPPING AND UNZIPPING FILES

- To unzip contents of the files that are available in a zip file.

```
from zipfile import *


z = ZipFile('C:\\DM\\Python
Programs\\test.zip','r')


z.extractall('C:\DM\Python
Programs')
```

Working with directories

Working with directories

- os module is used o perform simple operations on directories.
- getcwd() method is used to know the currently working directory.
- mkdir() method is used to create own directory in the present directory.
- Python program to know currently working directory:

```
import os
current = os.getcwd()
print('Current directory= ',current)
```

Working with directories

- Python program to create sub directory and sub-sub directory in the current directory.

```
import os
os.mkdir('C:\DM\Python Programs\mysub1')
os.mkdir('C:\DM\Python
Programs\mysub1\mysub2')
```

- Problem with mkdir() method:
  - Cannot create a sub directory unless the parent directory exists.
  - makedirs() method is used to recursively creates sub directories.

```
import os
os.makedirs('C:\DM\Python Programs\mysub2\mysub2')
```

Working with directories

- Python program to change to another directory.

```
import os
goto = os.chdir('C:\DM\Python Programs\mysub\mysub2')
current=os.getcwd()
print('Current directory=',current)
```

- To remove current directory rmdir() method.

```
import os
os.rmdir('C:\DM\Python Programs\mysub1\mysub2')
```

Working with directories

- Rename() method to give new name to an existing directory.
  - os.rename('oldname','newname')

- To know all the contents of current directory.
  - **os.walk()**
  - **os.walk(path,topdown=TRUE, onerror=None, followlinks=False)**
  - path- represents directory name. For current directory use[.]
  - Topdown is true then directory and its subdirectoies are traversed in top down manner.
  - Onerror represents what to do when an error is encountered

Working with directories

```
import os
for dirpath, dirnames, filenames in os.walk('.'):
print('Current path:',dirpath)
print('Directories:', dirnames)
print('Files:',filenames)
print()
```

Running other programs from python program

- Os module has system() method that is useful to run an executable program from our python program.
- System('string') represents any command or executable file name.

```
import os
os.system('dir *.py')
```