Introduction

Python

Python

- Combines the features of **C** and **JAVA**
- It offers elegant style of developing programs like C
- It offers classes and objects like Java

Features of Python

- Simple
  - More clarity
  - Less stress on reading and understanding the syntax
- Easy to learn
  - Uses very few keywords
  - Very simple structure, resembles C
- Open source
- High level language
- Dynamically typed
  - Type of the variable is not declared statically

Features of Python...

- Platform Independent
  - Python compiler generates byte code
  - PVM interprets the byte code
- Portable
- Procedure and Object oriented language
- Interpreted
- Extensible
- Embeddable
- Huge Library
- Scripting Language
- Database Connectivity
  - Provides interfaces to DB like Oracle, Sybase or MySql

Execution of a Python Program

- Example:
  - x.py → python_compiler → x.pyc → PVM → Machine_Code
  - python -m py_compile x.py
  - python   x.cpython-34.py
  - python -m dis add.py

Memory Management in Python

- In C or C++, allocation and deallocation of memory will be done manually
    - malloc(), calloc(), realloc() or free()
- In python, it is done at run time automatically
- Memory Manager inside the PVM takes care of allocating memory for all objects in Python.
- All objects are stored in **Heap**

Garbage Collection in Python

- Garbage collector is a module in Python that is useful to delete objects from memory which are not used in the program.
- The module that represents the GC is gc.
- It will keep track of how many times the object is referenced.
    - If it is referenced 0 times, then **gc** will remove object from memory.

C Vs Python

C	Python
Procedure Oriented language	Object Oriented language
Faster	Slower
Compulsory to declare the data types of variables	Data Types are not required
Type discipline is static and weak	Dynamic and strong
Pointers concept present	No pointers concept
No exception handling facility	Exception handling facility is robust
Do-while is present	Absent
Has switch statement	No Switch

C Vs Python

C	Python
Manually allocate the memory	Automatic
Absence of GC	GC is present
Supports Single and multi dimensional arrays	Supports only single dimension
Array should be positive	Can be Positive or negative
Array bounds checking is not present	Present
Indentation is not necessary	Strictly needed
Every statement is terminated by ;	No semicolon

Chapter-2

Data Types

Comments

Comments

Single Line Comments

- Starts with # symbol
- Comments are non-executable statements

```
#To find sum of two numbers
a = 10 #Store 10 into variable 'a'
```

Comments

Multi Line Comments

- Version-1

```
#To find sum of two numbers
#This is multi-line comments
#One more commented line
```

- Version-2

```
"""
This is first line
This second line
Finally comes third
"""
```

- Version-3

```
'''
This is first line
This second line
Finally comes third
'''
```

Docstrings

Multi Line Comments

- Python supports only single line commenting
- Strings enclosed within ''' … ''' or """ … """, if not assigned to any variable, they are removed from memory by the **GC**
- Also called as Documentation Strings **OR** docstrings
- Useful to create API file

```
Command to Create the html file
-------------------------------
py -m pydoc -w 1_Docstrings

-m: Module
-w: To create the html file
```

How python sees variables

Data-Types

None Type

- **None** data-type represents an object that does not contain any value
- In Java, it is called as **NULL** Object
- In Python, it is called as **NONE** Object
- In boolean expression, **NONE** data-type represents ‘**False**’
- Example:
  - a = “”

Data-Types

Numeric Type

- int
  - No limit for the size of an int datatype
  - Can store very large numbers conveniently
  - Only limited by the memory of the system
  - Example:
    - a = 20

Data-**T**ypes

Numeric **T**ype

- float
  - Example-1:
    - A = 56.78
  - Example-2:
    - B = 22.55e3 ⇔ B = 22.55 x 10^3

Data-Types

Numeric Type

- Complex
  - Written in the form **a + bj** OR **a + bJ**
  - a and b may be ints or floats
  - Example:
    - c = 1 + 5j
    - c = -1 - 4.4j

Representation

Binary, **O**ctal, **H**exadecimal

- Binary
  - Prefixed with **0b** OR **0B**
    - 0b11001100
    - 0B10101100
- Octal
  - Prefixed with 0o OR 0O
    - 0o134
    - 0O345
- Hexadecimal
  - Prefixed with 0x OR 0X
    - **0x**AB
    - **0X**ab

Conversion
Explicit

- Coercion / type conversions
  - Example-1:

```
x = 15.56
int(x) #Will convert into int and display 15
```

  - Example-2:

```
x = 15
float(x) #Will convert into float and display 15.0
```

Conversion

Explicit

- Coercion / type conversions
  - ○ Example-3:

```
a = 15.56
complex(a) #Will convert into complex and display (15.56 + 0j)
```

  - ○ Example-4:

```
a = 15
b = 3
complex(a, b) #Will convert into complex and display (15 + 3j)
```

Conversion

Explicit

- Coercion / type conversions
  - Example-5: To convert string into integer
  - Syntax: int(string, base)

```
str = “1c2”
n = int(str, 16)
print(n)
```

  - Other functions are
    - bin(): To convert int to binary
    - oct(): To convert oct to binary
    - hex(): To convert hex to binary

bool Data-Type

- Two bool values
  - True: Internally represented as 1
  - False: Internally represented as 0

- Blank string “” also represented as False

- Example-1:

```
a = 10
b = 20
if ( a < b):
      print(“Hello”)
```

bool Data-Type

- Example-2:

```
a = 10 > 5
print(a)  #Prints True

a = 5 > 10
print(a)  #Prints False
```

- Example-3:

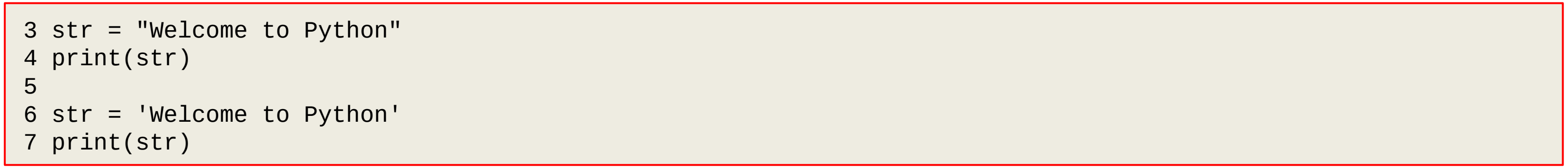
```
print(True + True)   #Prints 2


print(True + False)  #Prints 1
```

Sequences

Sequences

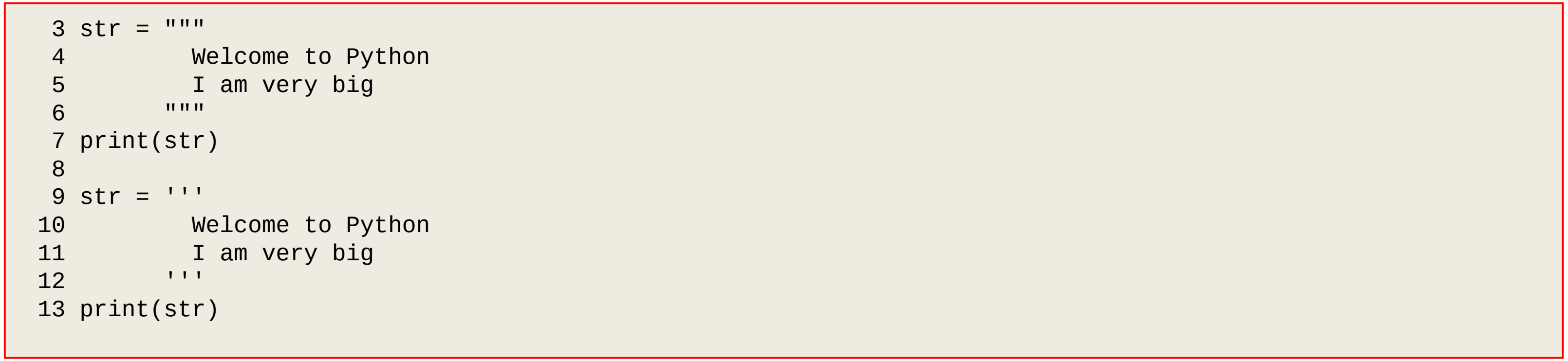
str

- **str** represents the string data-type
- Example-1:



```
str = "Welcome to Python"
print(str)

str = 'Welcome to Python'
print(str)
```

- Example-2:

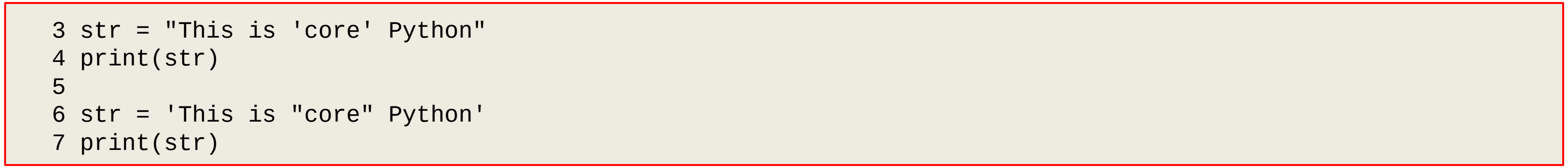


```
str = """
        Welcome to Python
        I am very big
      """
print(str)

str = '''
        Welcome to Python
        I am very big
      '''
print(str)
```

Sequences

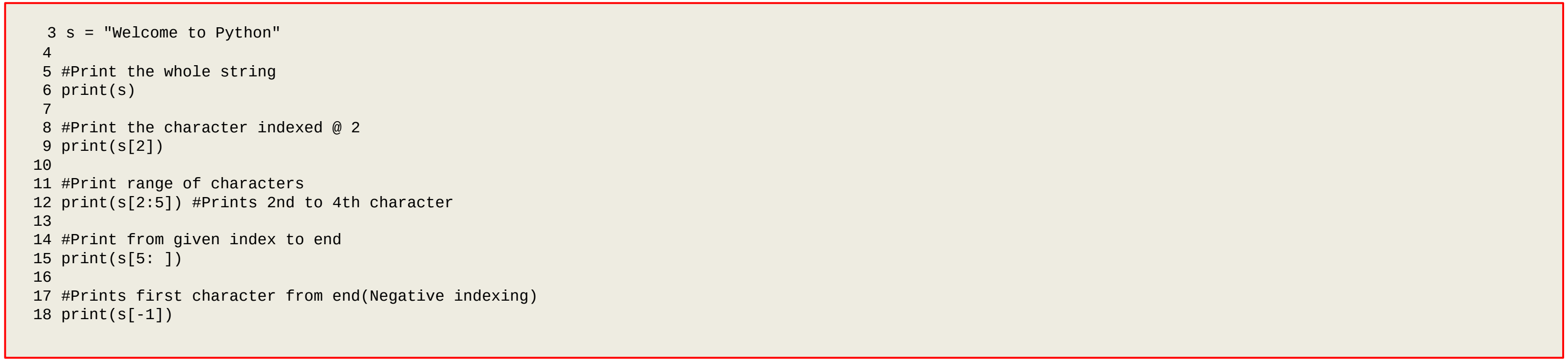
str

- Example-3:



```
str = "This is 'core' Python"
print(str)

str = 'This is "core" Python'
print(str)
```

- Example-4:

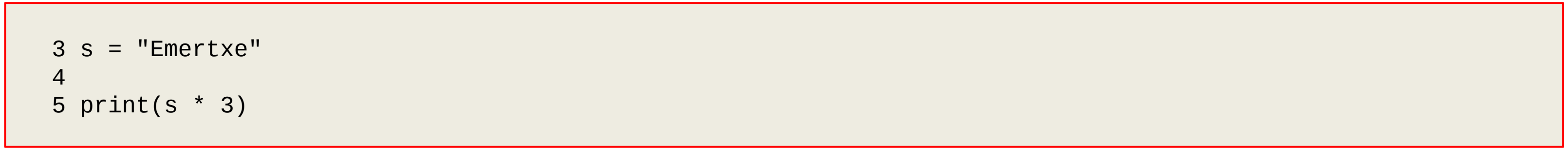


```
s = "Welcome to Python"

#Print the whole string
print(s)

#Print the character indexed @ 2
print(s[2])

#Print range of characters
print(s[2:5]) #Prints 2nd to 4th character

#Print from given index to end
print(s[5: ])

#Prints first character from end(Negative indexing)
print(s[-1])
```

Sequences

str

- Example-5:



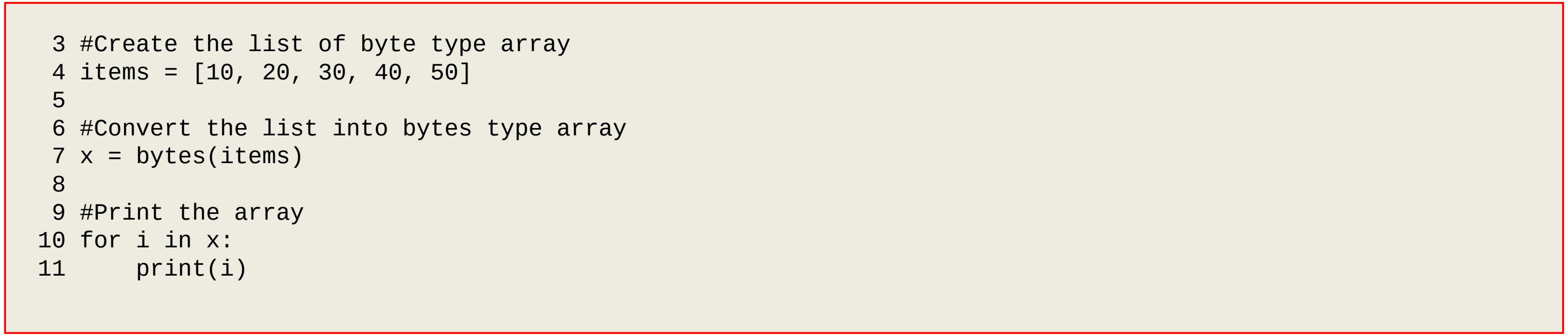
```
s = "Emertxe"

print(s * 3)
```

bytes Data-types

Sequences

bytes

- **bytes** represents a group of byte numbers
- A **byte** is any positive number between 0 and 255(Inclusive)
- Example-1:

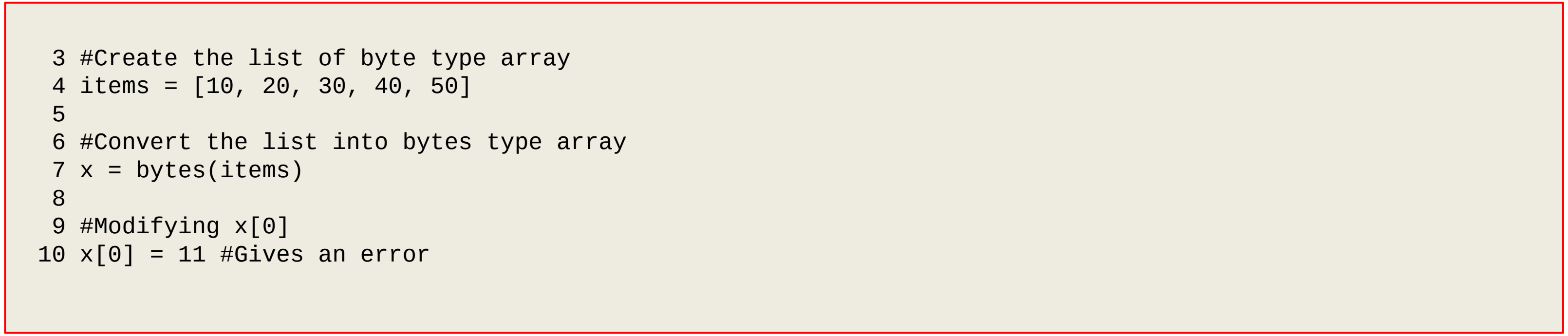


```
#Create the list of byte type array
items = [10, 20, 30, 40, 50]

#Convert the list into bytes type array
x = bytes(items)

#Print the array
for i in x:
    print(i)
```

Sequences

bytes

- Modifying any item in the **byte** type is not possible
- Example-2:



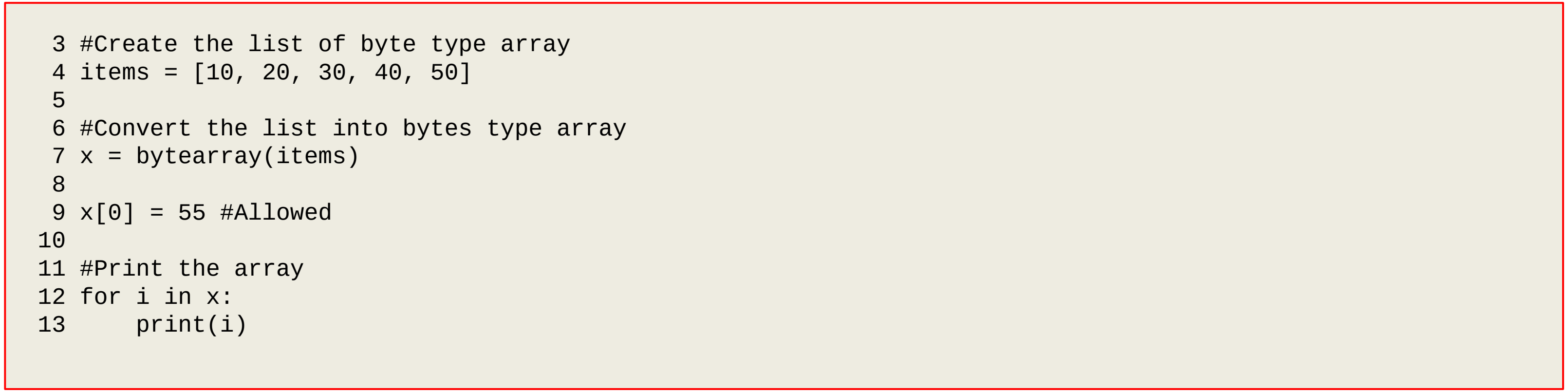
```
#Create the list of byte type array
items = [10, 20, 30, 40, 50]

#Convert the list into bytes type array
x = bytes(items)

#Modifying x[0]
x[0] = 11 #Gives an error
```

bytearray Data-type

Sequences

bytearray

- **bytearray** is similar to **bytes**
- Difference is items in **bytearray** is **modifiable**
- Example-1:



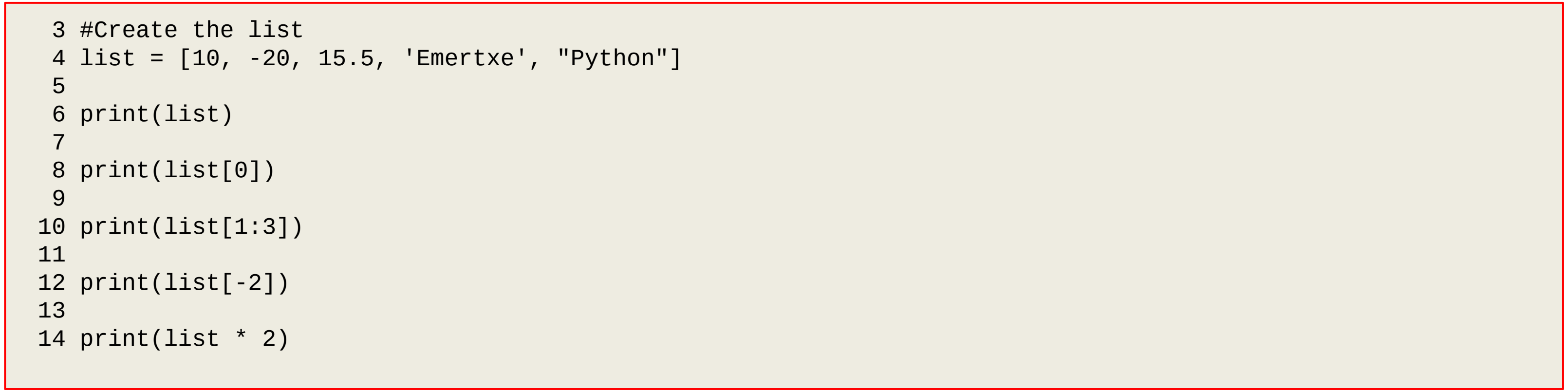
```
#Create the list of byte type array
items = [10, 20, 30, 40, 50]

#Convert the list into bytes type array
x = bytearray(items)

x[0] = 55 #Allowed

#Print the array
for i in x:
    print(i)
```

list Data-type

Sequences

list

- **list** is similar to array, but contains items of different data-types
- **list** can grow dynamically at run-time, but arrays cannot
- Example-1:

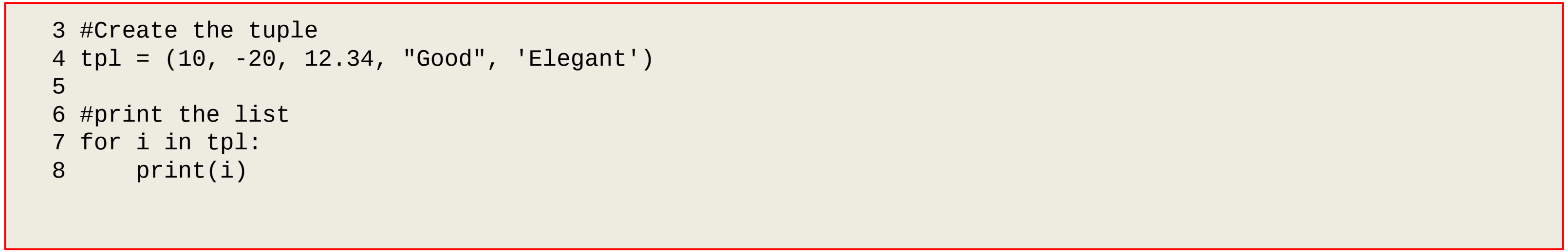


```
#Create the list
list = [10, -20, 15.5, 'Emertxe', "Python"]

print(list)

print(list[0])

print(list[1:3])

print(list[-2])

print(list * 2)
```

tuple Data-type

Sequences
tuple

- **tuple** is similar to **list,** but items cannot be modified
- **tuple** is read-only **list**
- **tuple** are enclosed within ()
- Example-1:



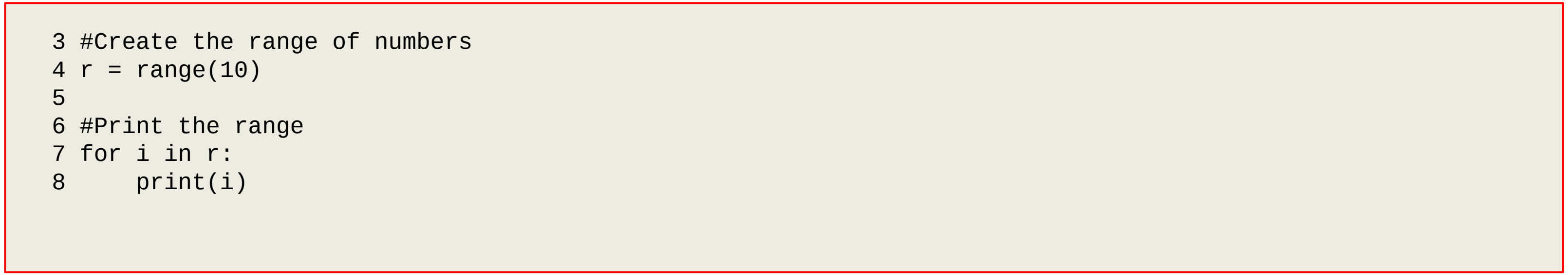
```
#Create the tuple
tpl = (10, -20, 12.34, "Good", 'Elegant')

#print the list
for i in tpl:
    print(i)
```

range Data-type

Sequences

range

- **range** represents sequence of numbers
- Numbers in **range** are not modifiable
- Example-1:

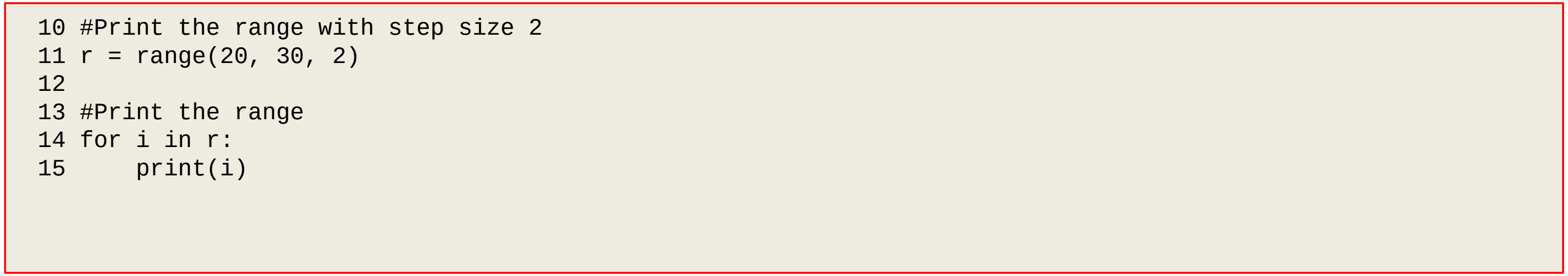


```
#Create the range of numbers
r = range(10)

#Print the range
for i in r:
    print(i)
```

Sequences

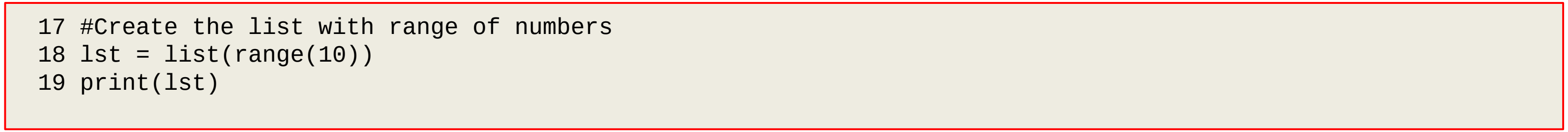
range

- Example-2:



```
#Print the range with step size 2
r = range(20, 30, 2)

#Print the range
for i in r:
    print(i)
```

- Example-3:



```
#Create the list with range of numbers
lst = list(range(10))
print(lst)
```

Sets

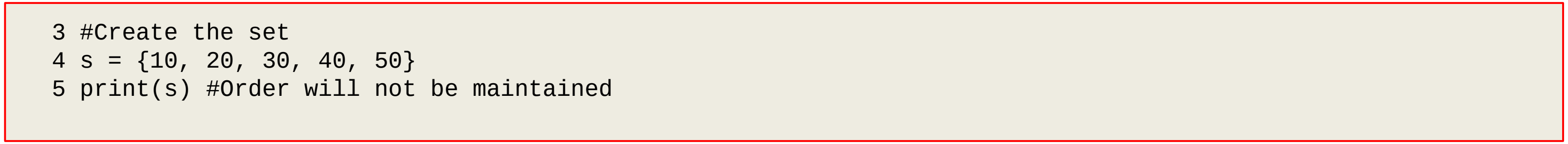
Sets

- **Set** is an unordered collection of elements
- Elements may not appear in the same order as they are entered into the set
- **Set** does not accept duplicate items
- Types
  - set datatype
  - frozenset datatype

Sets

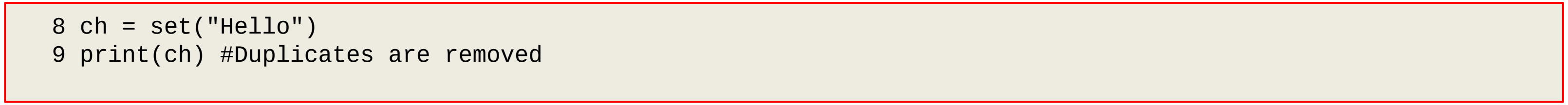
set

- Example-1:



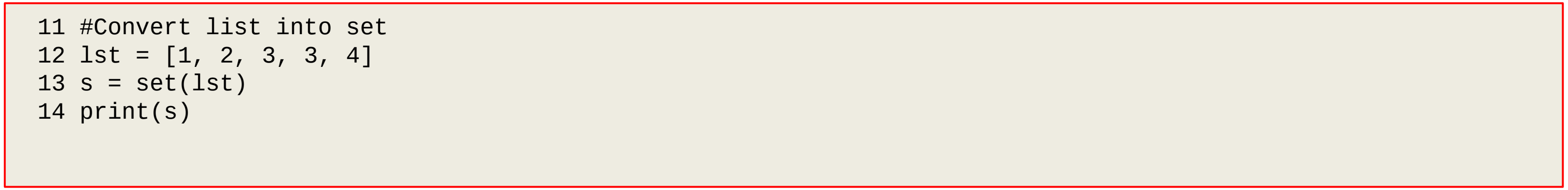
```
#Create the set
s = {10, 20, 30, 40, 50}
print(s) #Order will not be maintained
```

- Example-2:



```
ch = set("Hello")
print(ch) #Duplicates are removed
```

- Example-3:



```
#Convert list into set
lst = [1, 2, 3, 3, 4]
s = set(lst)
print(s)
```

Sets

set

- Example-5:

```
#Convert list into set
lst = [1, 2, 3, 3, 4]
s = set(lst)
print(s)
```

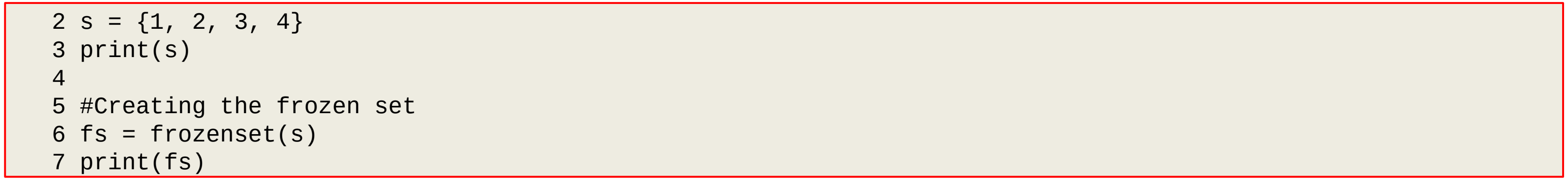
- Example-6:

```
#Addition of items into the array
s.update([50, 60])
print(s)

#Remove the item 50
s.remove(50)
print(s)
```

Sets

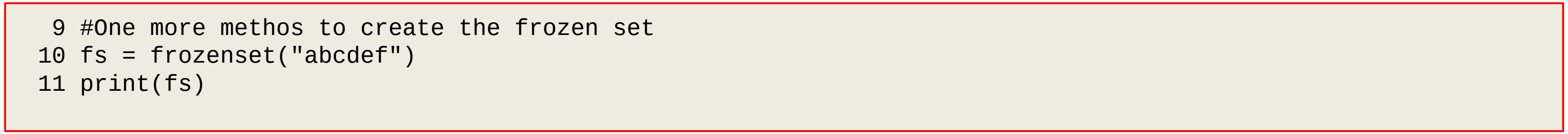
frozenset

- Similar to that of **set,** but cannot modify any item
- Example-1:



```
s = {1, 2, 3, 4}
print(s)

#Creating the frozen set
fs = frozenset(s)
print(fs)
```

- Example-2:

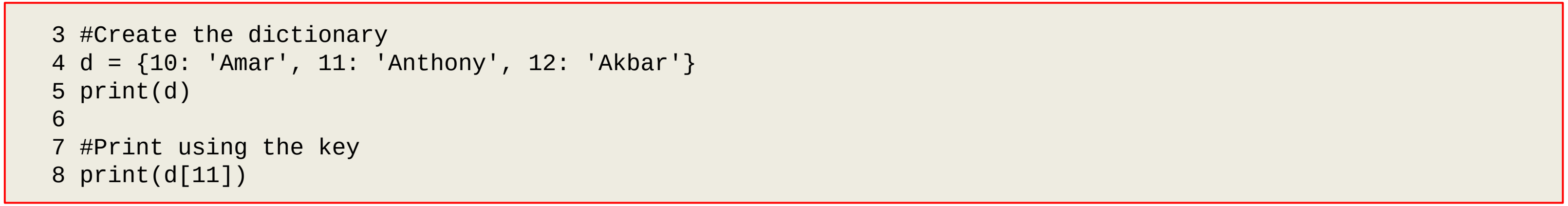


```
#One more methos to create the frozen set
fs = frozenset("abcdef")
print(fs)
```

Mapping Types

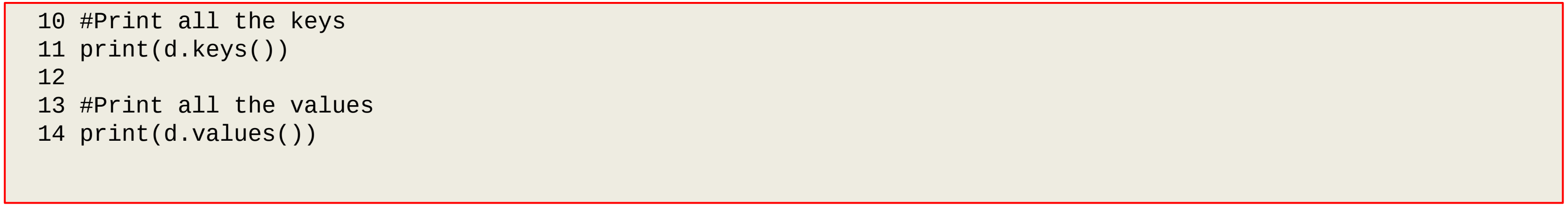
Mapping

- **Map** represents group of items in the form of **key: value** pair
- dict data-type is an example for map
- Example-1:



```
#Create the dictionary
d = {10: 'Amar', 11: 'Anthony', 12: 'Akbar'}
print(d)

#Print using the key
print(d[11])
```

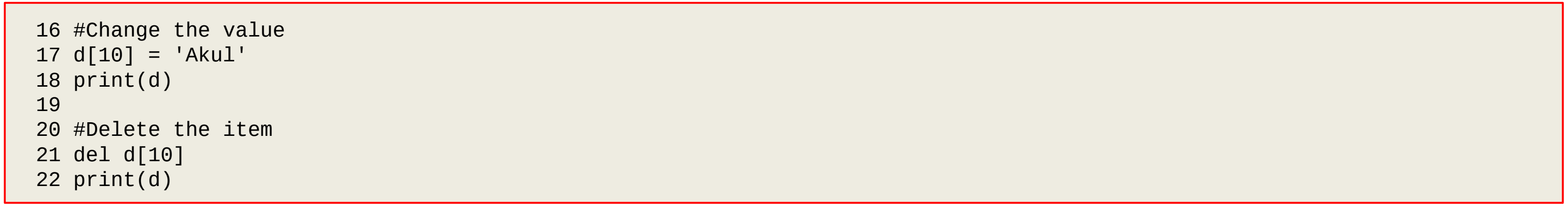
- Example-2:



```
#Print all the keys
print(d.keys())

#Print all the values
print(d.values())
```

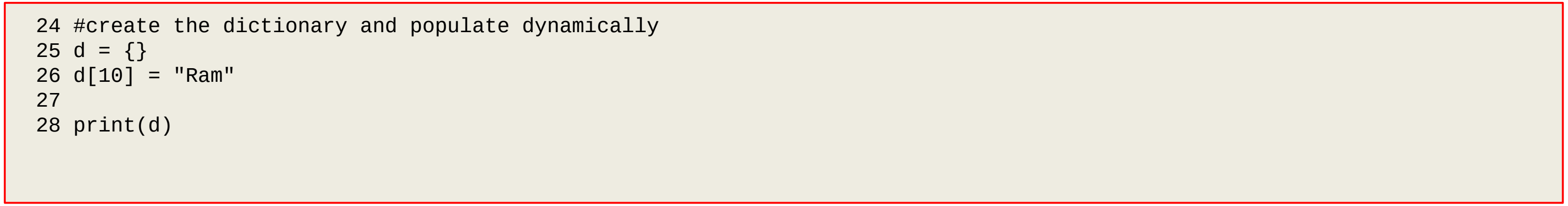
Mapping

- Example-3:



```
#Change the value
d[10] = 'Akul'
print(d)

#Delete the item
del d[10]
print(d)
```

- Example-4:

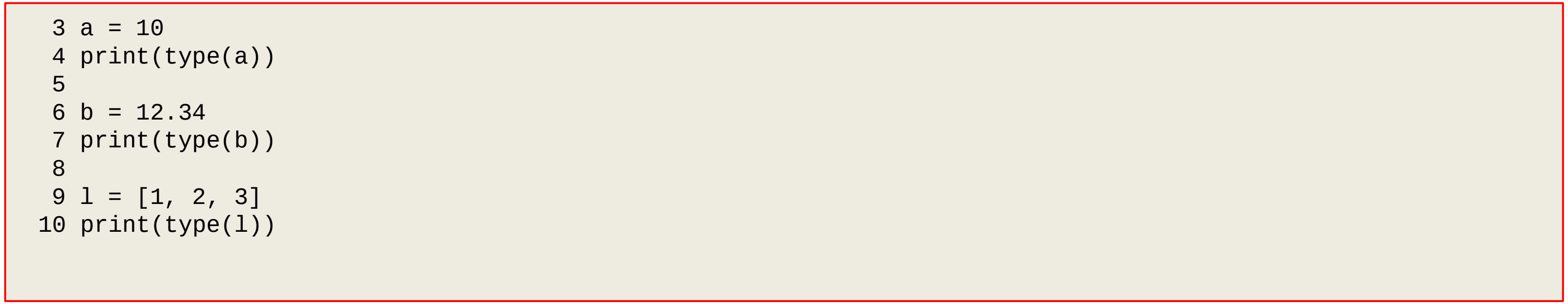


```
#create the dictionary and populate dynamically
d = {}
d[10] = "Ram"

print(d)
```

Determining the Datatype

Determining Datatype of a Variable

- type()
- Example-1:



```
a = 10
print(type(a))

b = 12.34
print(type(b))

l = [1, 2, 3]
print(type(l))
```

Operators

Team Emertxe

Arithmetic

OPERATORS

Arithmetic

Operator	Example	Result
+	a + b	18
-	a - b	8
*	a * b	65
/	a / b	2.6
%	a % b	3
**	a ** b	371293
//	a // b	2

```
The results are obtained for the values of:
a = 13
b = 5
```

OPERATORS
Assignment

Operators
=
+=
-+
*+
/=
%=
**=
//=

Example-1:

```
a = b = 1
```

Example-2:

```
a = 1; b = 1
```

Example-3:

```
a, b = 1, 2
```

```
Python does not have ++ AND -- operators
```

OPERATORS

Unary Minus

Example-1:

```
n = 10
print(-n)
```

Example-2:

```
num = -10
num = -num
print(num)
```

OPERATORS

Relational

Operator	Example	Result
>	a > b	False
>=	a >= b	False
<	a < b	True
<=	a <= b	True
==	a == b	False
!=	a != b	True

```
The results are obtained for the values of:
a = 1
b = 2
```

OPERATORS

Relational: Chaining

Example-1:

```
x = 15
print(10< x < 20)
```

Example-2:

```
print(1 < 2 < 3 < 4)
```

OPERATORS

Logical

If a = 100, b = 200

Operator	Example	Result
and	a and b	2
or	a or b	1
not	not a	False

Example-1:

```
if (a < b and b < c):
     print("Yes")
else:
     print("No")
```

Example-2:

```
if (a > b or b < c):
     print("Yes")
else:
     print("No")
```

Short Circuit evaluation implies to Logical Operators

OPERATORS

Boolean

If a = True, b = False

Operator	Example	Result
and	a and b	False
or	a or b	True
not	not a	False

Example-1:

```
print(a and b)
print(a or b)
print(not a)
```

OPERATORS
Bitwise

If a = 10(0000 1010), b = 11(0000 1011)

Operator	Example	Result
~	~a	1111 0101(**-11**)
&	a & b	0000 1010(**10**)
|	a | b	0000 1011(**11**)
^	a ^ b	0000 0001(**1**)
<<	a << 2	0010 1000(**40**)
>>	a >> 2	0000 0010(**2**)

In case of >> shifting, it preserves the sign of the number.

OPERATORS
Membership

Operator	Description
in	Returns True, if an item **is found** in the specified sequence
not in	Returns True, if an item **is not found** in the specified sequence

Example-1:

```
names = ["Ram", "Hari", "Thomas"]

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

Example-2:

```
postal = {"Delhi": 110001, "Chennai": 600001, "Bangalore": 560001}

for city in postal:
    print(city, postal[city])
```

OPERATORS

Identity

- Use to comapre the memory locations of two objects
- id(): Is used to get the memory location ID

Example-1:

```
a = 25
b = 25
if (a is b): #This compares only the locations
    print("a and b are same")
```

Operator	Description
is	Returns True, if ID of two objects **are same**
is not	Returns True, if ID of two objects **are not same**

OPERATORS

Identity

- To compare two objects, use ‘==’ operator

Example-1:

```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]

if (a == b):
    print("Objects are same")
else:
    print("Objects are not same")
```

OPERATORS

Precedence & Associativity

Operator	Name
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or tuple display, list display, dictionary display, set display
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
**	Exponentiation
+, -, ~	Positive, negative, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+, -	Addition, Subraction
<<, >>	Bitwise Left, Right shift
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not	Boolean not
and	Boolean and
or	Boolean or
if-else	Conditional Expression
lambda	Lambda Expression

All operators follow, Left - Right associativity, except ** which follows Right - Left

Mathematical Functions

Example-1:

```
import math
x = math.sqrt(16)
```

Example-2:

```
import math as m
x = m.sqrt(16)
```

Example-3:

```
from math import sqrt
x = sqrt(16)
```

Example-4:

```
from math import sqrt, factorial
x = sqrt(16)
y = factorial(5)
```

THANK YOU

Standard Input & Output

Team Emertxe

Output Statements

Output Statements

Print()

- print(), when called simply throws the cursor to the next line
- Means, a blank line will be displayed

Output Statements

Print(“string”)

Example	Output
print()	Prints the '\n' character
print("Hello")	Hello
print('Hello')	Hello
print("Hello \nWorld")	Hello World
print("Hello \tWorld")	Hello World
print("Hello \\nWorld")	Hello \nWorld
print(3 * 'Hello')	HelloHelloHello
print("Hello"+"World")	HelloWorld
print("Hello","World")	Hello World

Output Statements

Print(variable list)

Example	Output
a, b = 1, 2 print(a, b)	1 2
print(a, b, sep=",")	1,2
print(a, b, sep=':')	1:2
print(a, b, sep='---')	1---2

print("Hello", end="")
print("World")
	HelloWorld

print("Hello", end="\t")
print("World")
	Hello World

Output Statements

Print(object)

- Objects like list, tuples or dictionaries can be displayed

Example	Output

`lst = [10, 'A', "Hai"]`
`print(lst)`
	`[10, 'A', 'Hai']`

`d = {10: "Ram", 20: "Amar"}`
`print(d)`
	`{10: 'Ram', 20: 'Amar'}`

Output Statements

Print(“string”, variable list)

Example	Output

`a = 2`
`print(a, ": Even Number")`
 	
`print("You typed", a, "as Input")`
	`2 : Even Number`
	`You typed 2 as Input`

Output Statements

Print(formatted string)

```
Syntax: print("formatted string" % (varaible list))
```

Example	Output
a = 10 print("The value of a: %i" % a)	The value of a: 10
a, b = 10, 20 print("a: %d\tb: %d" % (a, b))	a: 10 b: 20

name = "Ram"
print("Hai %s" % name)
print("Hai (%20s)" % name)
print("Hai (%-20s)" % name)
	Hai Ram
	Hai (Ram)
	Hai (Ram)

print("%c" % name[2])	m
print("%s" % name[0:2])	Ra

num = 123.345727
print("Num: %f" % num)
print("Num: %8.2f" % num)
	Num: 123.345727
	Num: 123.35

Output Statements

Print(formatted string)

```
Syntax: print("formatted string" % (varaible list))
```

Example	Output

a, b, c = 1, 2, 3
print("First= {0}". format(a))
print("First= {0}, Second= {1}". format(a, b))
print("First= {one}, Second= {two}". format(one=a, two=b))
print("First= {}, Second= {}". format(a, b))
	First= 1
	First= 1, Second= 2
	First= 1, Second= 2
	First= 1, Second= 2

name, salary = "Ram", 123.45
print("Hello {0}, your salary: {1}". format(name, salary))
print("Hello {n}, your salary: {s}". format(n=name, s=salary))
print("Hello {:s}, your salary: {:.2f}". format(name, salary))
print("Hello %s, your salary: %.2f" % (name, salary))
	Hello Ram, your salary: 123.45
	Hello Ram, your salary: 123.45
	Hello Ram, your salary: 123.45
	Hello Ram, your salary: 123.45

ΣMERTXE

Input Statements

Input Statements

Input()

Example

`str = input()`
`print(str)`

`str = input("Enter the name: ")`
`print(str)`

`a = int(input("Enter the number: "))`
`print(a)`

`b = float(input("Enter the float number: "))`
`print(b)`

Command Line Arguments

CLA

Example

```
#To display CLA

import sys

#Get the no. of CLA
n = len(sys.argv)

#Get the arguments
args = sys.argv

#Print the 'n'
print("No. Of CLA: ", n)

#print the arguments in one shot
print(args)

#Print the arguments one by one
for i in args:
    print(i)
```

CLA

Parsing CLA

- argparse module is useful to develop user-friendly programs
- This module automatically generates help and usage messages
- May also display appropriate error messages

CLA

Parsing **CLA: S**teps

- Step-1: Import argparse module

```
import argparse
```

- Step-2: Create an Object of ArgumentParser

```
parser = argparse.ArgumentParser(description="This program displays square of two numbers")
```

- Step-2a: If programmer does not want to display description, then above step can be skipped

```
parser = argparse.ArgumentParser()
```

- Step-3: Add the arguments to the parser

```
parser.add_argument("num", type=int, help="Enter only int number.")
```

- Step-4: Retrieve the arguments

```
args = parser.parse_args()
```

- Step-5: Access the arguments

```
args.num
```

THANK YOU

Control Statements

Team Emertxe

Single **C**ontrol **S**tatements

If Statements

If-Else

- Syntax

```
if condition:
    statements
else:
    statements
```

- Example

```
if num % 2:
    print("ODD")
else:
    print("EVEN")
```

If Statements

If-Elif-Else

- Syntax

```
if condition1:
    statements
elif condition2:
    statements
else
    statements
```

- Example

```
if num == 1:
    print("You entered 1")
elif num == 2:
    print("You entered 2")
else:
    print("You entered 3")
```

Multiple **C**ontrol **S**tatements

Multiple Statements

While

- Syntax

```
while condition:
     statements
```

- Example

```
i = 1
while i <= 10:
     print(i)
     i = i + 1
```

Multiple Statements

For

- Syntax

```
for var in sequence:
    statements
```

- Example-v1.1

```
str = "Hello"
for ch in str:
    print(ch, end='')
```

- Example-v1.2

```
n = len(str)
for i in range(n):
    print(str[i])
```

Else **S**uite

Multiple Statements

For with Else suite

- Syntax

```
for var in sequence:
    statement / statements
else:
    statement / statements
```

- Example

```
for i in range(5):
    print(i)
else:
    print("Over")
```

Multiple Statements
While with Else suite

- Syntax

```
while condition:
    statement / statements
else:
    statement / statements
```

- Example

```
i = 0
while i < 5
    print(i)
    i += 1
else:
    print("Over")
```

While searching an elemnt in the sequence is not found,
else will be the best option to display the item not found

Misc **S**tatements

Misc Statements

Break

- Example

```
x = 10
while x >= 1:
    print("x = ", x)
    x -= 1
    if x == 5:
        break
```

Misc Statements
Continue

- Example

```
x = 10
while x >= 1:
    if x == 5:
        x -= 1
        continue
    print("x = ", x)
    x -= 1
```

Misc Statements

Pass

- Example-1

```
x = 0
while x < 10:
    x += 1
    if x == 5:
        pass
    print(x)
```

Misc Statements

Pass

- Example-2: Program to retrieve only the negative numbers from the list

```
num = [1, 2, 3, -4, -5, -6, 7, 8]
for i in num:
    if (i > 0):
        pass
    else:
        print(i)
```

Pass does nothing

Misc Statements

Assert

- Syntax:

```
assert expression, message
```

- Example-1

```
num = int(input("Enter the number greater than zero: "))
assert num > 0, "Wrong input"
print("Num: ", num)
```

Misc Statements

Assert: Try Except

- Example-1

```
num = int(input("Enter the number greater than zero: "))
try:
    assert num > 0, "Wrong input"
    print("Num: ", num)
except AssertionError:
    print("You entered wrong input")
    print("Enter positive number")
```

Misc Statements

Return

- Example-1: To add two numbers & return the result

```
def sum(a, b):
    return a + b

res = sum(5, 10)
print(res)
```

THANK YOU

Array

Team Emertxe

ΣMERTXE

Single **D**imensional **A**rrays

Single Dimensional Arrays

Creating an Array

Syntax	array_name = array(type_code, [elements])
Example-1	a = array(‘i’, [4, 6, 2, 9])
Example-2	a = array(‘d’, [1.5, -2.2, 3, 5.75])

Single Dimensional Arrays

Creating an Array

Typecode	C Type	Sizes
‘b’	signed integer	1
‘B’	unsigned integer	1
‘i’	signed integer	2
‘I’	unsigned integer	2
‘l’	signed integer	4
‘L’	unsigned integer	4
‘f’	floating point	4
‘d’	double precision floating point	8
‘u’	unicode character	2

Single Dimensional Arrays

Importing an Array Module

`import array`	`a = array.array(‘i’, [4, 6, 2, 9])`
`import array as ar`	`a = ar.array(‘i’, [4, 6, 2, 9])`
`from array import *`	`a = array(‘i’, [4, 6, 2, 9])`

Importing an Array Module

Example-1

```
import array

#Create an array
a = array.array("i", [1, 2, 3, 4])

#print the items of an array
print("Items are: ")
for i in a:
    print(i)
```

Importing an Array Module

Example-2

```
from array import *

#Create an array
a = array("i", [1, 2, 3, 4])

#print the items of an array
print("Items are: ")
for i in a:
    print(i)
```

Importing an Array Module

Example-3

```
from array import *


#Create an array
a = array('u', ['a', 'b', 'c', 'd'])
#Here, 'u' stands for unicode character



#print the items of an array
print("Items are: ")
for ch in a:
    print(ch)
```

Importing an Array Module

Example-4

```
from array import *

#Create first array
a = array('i', [1, 2, 3, 4])

#From first array create second
b = array(a.typecode, (i for i in a))

#print the second array items
print("Items are: ")
for i in b:
    print(i)

#From first array create third
c = array(a.typecode, (i * 3 for i in a))

#print the second array items
print("Items are: ")
for i in c:
    print(i)
```

Indexing & Slicing on Array

Example-1: Indexing

```
#To retrieve the items of an array using array index

from array import *

#Create an array
a = array('i', [1, 2, 3, 4])

#Get the length of the array
n = len(a)

#print the Items
for i in range(n):
    print(a[i], end=' ')
```

Indexing & Slicing on Array

Example-2: Indexing

```
#To retrieve the items of an array using array index using while loop

from array import *

#Create an array
a = array('i', [1, 2, 3, 4])

#Get the length of the array
n = len(a)

#print the Items
i = 0
while i < n:
    print(a[i], end=' ')
    i += 1
```

Indexing & Slicing on Array

Slicing

Syntax	arrayname[start: stop: stride]
Example	arr[1: 4: 1] Prints items from index 1 to 3 with the step size of 1

Indexing & Slicing on Array

Example-3: Slicing

```
#Create an array
x = array('i', [10, 20, 30, 40, 50, 60])
```

```
#Create array y with Items from 1st to 3rd from x
y = x[1: 4]
print(y)
```

```
#Create array y with Items from 0th till the last Item in x
y = x[0: ]
print(y)
```

```
#Create array y with Items from 0th till the 3rd Item in x
y = x[: 4]
print(y)
```

```
#Create array y with last 4 Items in x
y = x[-4: ]
print(y)
```

```
#Stride 2 means, after 0th Item, retrieve every 2nd Item from x
y = x[0: 7: 2]
print(y)
```

```
#To display range of items without storing in an array
for i in x[2: 5]:
    print(i)
```

Indexing & Slicing on Array

Example-4: Slicing

```
#To retrieve the items of an array using array index using for loop

from array import *

#Create an array
a = array('i', [1, 2, 3, 4])

#Display elements from 2nd to 4th only
for i in a[2: 5]:
     print(i)
```

Processing the Array

Method	Description
a.append(x)	Adds an element x at the end of the existing array a
a.count(x)	Returns the numbers of occurrences of x in the array a
a.extend(x)	Appends x at the end of the array a. ‘x’ can be another array or an iterable object
a.index(x)	Returns the position number of the first occurrence of x in the array. Raises ‘ValueError’ if not found
a.insert(i, x)	Inserts x in the position i in the array

Processing the Array

Method	Description
a.pop(x)	Removes the item x from the arry a and returns it
a.pop()	Removes last item from the array a
a.remove(x)	Removes the first occurrence of x in the array a. Raises ‘ValueError’ if not found
a.reverse()	Reverse the order of elements in the array a
a.tolist()	Converts the array ‘a’ into a list

Processing the Array

Examples

```
from array import *
#Create an array
a = array('i', [1, 2, 3, 4, 5])
print(a)

#Append 6 to an array
a.append(6)
print(a)

#Insert 11 at position 1
a.insert(1, 11)
print(a)

#Remove 11 from the array
a.remove(11)
print(a)

#Remove last item using pop()
item = a.pop()
print(a)
print("Item pop: ", item)
```

Processing the Array

Exercises

1. To store student's marks into an array and find total marks and percentage of marks

2. Implement Bubble sort

3. To search for the position of an item in an array using sequential search

4. To search for the position of an element in an array using index() method

Single **D**imensional **A**rrays
Numpy

Single Dimensional Arrays

Importing an **numpy**

`import numpy`	`a = numpy.array([4, 6, 2, 9])`
`import numpy as np`	`a = np.array([4, 6, 2, 9])`
`from numpy import *`	`a = array([4, 6, 2, 9])`

Single Dimensional Arrays

Creating an Array: **numpy-array()**

Example-1: To create an array of **int** datatype

```
a = array([10, 20, 30, 40, 50], int)
```

Example-2: To create an array of **float** datatype

```
a = array([10.1, 20.2, 30.3, 40.4, 50.5], float)
```

Example-3: To create an array of **float** datatype without specifying the float datatype

```
a = array([10, 20, 30.3, 40, 50])
```

Note: If one item in the array is of float type, then Python interpreter converts remaining items into the float datatype

Example-4: To create an array of **char** datatype

```
a = array([‘a’, ‘b’, ‘c’, ‘d’])
```

Note: No need to specify explicitly the char datatype

Single Dimensional Arrays

Creating an Array: **numpy-array()**

Program-1: To create an array of **char** datatype

```
from numpy import *

a = array(['a', 'b', 'c', 'd'])
print(a)
```

Program-2: To create an array of **str** datatype

```
from numpy import *

a = array(['abc', 'bcd', 'cde', 'def'], dtype=str)
print(a)
```

Single Dimensional Arrays

Creating an Array: **numpy-array()**

Program-3: To create an array from another array using numpy

```
from numpy import *

a = array([1, 2, 3, 4, 5])
print(a)

#Create another array using array() method
b = array(a)
print(a)

#Create another array by just copy
c = a
print(a)
```

Single Dimensional Arrays

Creating an Array: numpy-linspace()

Syntax	linspace(start, stop, n)
Example	a = linspace(0, 10, 5)
Description	Create an array ‘a’ with starting element 0 and ending 10. This range is divide into 5 equal parts Hence, items are 0, 2.5, 5, 7.5, 10

Program-1: To create an array with 5 equal points using linspace

```
from numpy import *

#Divide 0 to 10 into 5 parts and take those points in the array
a = linspace(0, 10, 5)
print(a)
```

Single Dimensional Arrays

Creating an Array: numpy-logspace()

Syntax	logspace(start, stop, n)
Example	a = logspace(1, 4, 5)
Description	Create an array ‘a’ with starting element 10^1 and ending 10^4. This range is divide into 5 equal parts Hence, items are 10. 56.23413252 316.22776602 1778.27941004 10000.

Program-1: To create an array with 5 equal points using logspace

```
from numpy import *

#Divide the range 10^1 to 10^4 into 5 equal parts
a = logspace(1, 4, 5)
print(a)
```

Single Dimensional Arrays

Creating an Array: numpy-arange()

Syntax	arange(start, stop, stepsize)	
Example-1	arange(10)	Produces items from 0 – 9
Example-2	arange(5, 10)	Produces items from 5 – 9
Example-3	arange(1, 10, 3)	Produces items from 1, 4, 7
Example-4	arange(10, 1, -1)	Produces items from [10 9 8 7 6 5 4 3 2]
Example-5	arange(0, 10, 1.5)	Produces [0. 1.5 3. 4.5 6. 7.5 9.]

Program-1: To create an array with even number upto 10

```
from numpy import *

a = arange(2, 11, 2)
print(a)
```

Single Dimensional Arrays

Creating Array: numpy-zeros() & ones()

Syntax	zeros(n, datatype)
	ones(n, datatype)	

Example-1	zeros(5)	Produces items [0. 0. 0. 0. 0.] Default datatype is float
Example-2	zeros(5, int)	Produces items [0 0 0 0 0]
Example-3	ones(5, float)	Produces items [1. 1. 1. 1. 1.]

Program-1: To create an array using zeros() and ones()

```
from numpy import *

a = zeros(5, int)
print(a)

b = ones(5) #Default datatype is float
print(b)
```

Single Dimensional Arrays

Vectorized Operations

Example-1	`a = array([10, 20 30.5, -40])`
	`a = a + 5 #Adds 5 to each item of an array`

Example-2	`a1 = array([10, 20 30.5, -40])`
	`a2 = array([1, 2, 3, 4])`
	`a3 = a1 + a2 #Adds each item of a1 and a2`

Importance of vectorized operations

1. Operations are faster
   - Adding two arrays in the form a + b is faster than taking corresponding items of both arrays and then adding them.

2. Syntactically clearer
   - Writing a + b is clearer than using the loops

3. Provides compact code

Single Dimensional Arrays

Mathematical Operations

sin(a)	Calculates sine value of each item in the array a
arcsin(a)	Calculates sine inverse value of each item in the array a
log(a)	Calculates natural log value of each item in the array a
abs(a)	Calculates absolute value of each item in the array a
sqrt(a)	Calculates square root value of each item in the array a
power(a, n)	Calculates a ^ n
exp(a)	Calculates exponential value of each item in the array a
sum(a)	Calculates sum of each item in the array a
prod(a)	Calculates product of each item in the array a
min(a)	Returns min value in the array a
max(a)	Returns max value in the array a

Single Dimensional Arrays

Comparing Arrays

- Relational operators are used to compare arrays of same size
- These operators compares corresponding items of the arrays and return another array with Boolean values

Program-1: To compare two arrays and display the resultant Boolean type array

```
from numpy import *

a = array([1, 2, 3])
b = array([3, 2, 3])

c = a == b
print(c)

c = a > b
print(c)

c = a <= b
print(c)
```

Single Dimensional Arrays

Comparing Arrays

- any(): Used to determine if any one item of the array is True
- all(): Used to determine if all items of the array are True

Program-2: To know the effects of any() and all()

```
from numpy import *

a = array([1, 2, 3])
b = array([3, 2, 3])

c = a > b
print(c)

print("any(): ", any(c))
print("all(): ", all(c))

if (any(a > b)):
    print("a contains one item greater than those of b")
```

Single Dimensional Arrays

Comparing Arrays

- logical_and(), logical_or() and logical_not() are useful to get the Boolean array as a
- result of comparing the compound condition

Program-3: To understand the usage of logical functions

```
from numpy import *

a = array([1, 2, 3])
b = array([3, 2, 3])

c = logical_and(a > 0, a < 4)
print(c)
```

Single Dimensional Arrays

Comparing Arrays

- where(): used to create a new array based on whether a given condition is True or False
- Syntax: a = where(condition, exp1, exp2)
  - If condition is True, the exp1 is evaluated, the result is stored in array
  - a, else exp2 will be evaluated

Program-4: To understand the usage of where function

```
from numpy import *

a = array([1, 2, 3], int)

c = where(a % 2 == 0, a, 0)
print(c)
```

Single Dimensional Arrays

Comparing Arrays

- where(): used to create a new array based on whether a given condition is True or False
- Syntax: a = where(condition, exp1, exp2)
    - If condition is True, the exp1 is evaluated, the result is stored in array
    - a, else exp2 will be evaluated

Exercise-1: To retrieve the biggest item after comparing two arrays using where()

Single Dimensional Arrays

Comparing Arrays

- nonzero(): used to know the positions of items which are non-zero
  - Returns an array that contains the indices of the items of the array which are non-zero
- Syntax: a = nonzero(array)

Program-5: To retrieve non zero items from an array

```
from numpy import *

a = array([1, 2, 0, -1, 0, 6], int)

c = nonzero(a)

#Display the indices
for i in c:
    print(i)


#Display the items
print(a[c])
```

Single Dimensional Arrays

Aliasing Arrays

- ‘Aliasing means not copying’. Means another name to the existing object

Program-1: To understand the effect of aliasing

```
from numpy import *

a = arange(1, 6)
b = a
print(a)
print(b)

#Modify 0th Item
b[0] = 99
print(a)
print(b)
```

Single Dimensional Arrays

Viewing & Copying

- view(): To create the duplicate array
- Also called as ‘shallow copying’

Program-1: To understand the view()

```
from numpy import *

a = arange(1, 6)
b = a.view() #Creates new array
print(a)
print(b)

#Modify 0th Item
b[0] = 99
print(a)
print(b)
```

Single Dimensional Arrays

Viewing & Copying

- copy(): To create the copy the original array
- Also called as ‘deep copying’

Program-1: To understand the view()

```
from numpy import *

a = arange(1, 6)
b = a.copy() #Creates new array
print(a)
print(b)

#Modify 0th Item
b[0] = 99
print(a)
print(b)
```

Multi **D**imensional **A**rrays **N**umpy

Multi Dimensional Arrays

Creating an Array

Example-1: To create an 2D array with 2 rows and 3 cols

```
a = array([[1, 2, 3],
           [4, 5, 6]]
```

Example-2: To create an 3D array with 2-2D arrays with each 2 rows and 3 cols

```
a = array([[[1, 2, 3],[4, 5, 6]]
           [[1, 1, 1], [1, 0, 1]]]
```

Multi Dimensional Arrays

Attributes of an Array: *The ndim*

- The ‘ndim’ attribute represents the number of dimensions or axes of an array
- The number of dimensions are also called as ‘rank’

Example-1: To understand the usage of the ndim attribute

```
a = array([1, 2, 3])

print(a.ndim)
```

Example-2: To understand the usage of the ndim attribute

```
a = array([[[1, 2, 3],[4, 5, 6]]
           [[1, 1, 1], [1, 0, 1]]]]

print(a.ndim)
```

Multi Dimensional Arrays

Attributes of an Array: *The shape*

- The ‘shape’ attribute gives the shape of an array
- The shape is a tuple listing the number of elements along each dimensions

Example-1: To understand the usage of the ‘shape’ attribute

```
a = array([1, 2, 3])

print(a.shape)
```

```
Outputs: (5, )
```

Example-2: To understand the usage of the ‘shape’ attribute

```
a = array([[1, 2, 3],[4, 5, 6]])

print(a.shape)
```

```
Outputs: (2, 3)
```

Example-3: To ‘shape’ attribute also changes the rows and cols

```
a = array([[1, 2, 3],[4, 5, 6]])

a.shape = (3, 2)

print(a)
```

```
Outputs:

[[1 2]
 [3 4]
 [5 6]]
```

Multi Dimensional Arrays

Attributes of an Array: *The size*

- The ‘size’ attribute gives the total number of items in an array

Example-1: To understand the usage of the ‘size’ attribute

```
a = array([1, 2, 3])

print(a.size)
```

Outputs: 5

Example-2: To understand the usage of the ‘size’ attribute

```
a = array([[1, 2, 3],[4, 5, 6]])

print(a.size)
```

Outputs: 6

Multi Dimensional Arrays

Attributes of an Array: *The itemsize*

- The 'itemsize' attribute gives the memory size of an array element in bytes

Example-1: To understand the usage of the 'itemsize' attribute

```
a = array([1, 2, 3, 4, 5])

print(a.itemsize)
```

```
Outputs: 4
```

Example-2: To understand the usage of the 'size' attribute

```
a = array([1.1, 2.3])

print(a.itemsize)
```

```
Outputs: 8
```

Multi Dimensional Arrays

Attributes of an Array: *The dtype*

- The ‘dtype’ attribute gives the datatype of the elements in the array

Example-1: To understand the usage of the ‘dtype’ attribute

```
a = array([1, 2, 3, 4, 5])

print(a.dtype)
```

```
Outputs: int32
```

Example-2: To understand the usage of the ‘dtype’ attribute

```
a = array([1.1, 2.3])

print(a.dtype)
```

```
Outputs: float64
```

Multi Dimensional Arrays

Attributes of an Array: *The nbytes*

- The 'nbytes' attribute gives the total number of bytes occupied by an array

Example-1: To understand the usage of the 'nbytes' attribute

```
a = array([1, 2, 3, 4, 5])

print(a.nbytes)
```

```
Outputs: 20
```

Example-2: To understand the usage of the 'nbytes' attribute

```
a = array([1.1, 2.3])

print(a.nbytes)
```

```
Outputs: 16
```

Multi Dimensional Arrays

Methods of an Array: *The reshape()*

- The ‘reshape’ method is useful to change the shape of an array

Example-1: To understand the usage of the ‘reshape’ method

```
a = arange(10)

#Change the shape as 2 Rows, 5 Cols
a = a.reshape(2, 5)

print(a)
```

```
Outputs:

[[0 1 2 3 4]
 [5 6 7 8 9]]
```

Example-2: To understand the usage of the ‘reshape’ method

```
#Change the shape to 5 rows, 2 cols
a = a.reshape(5, 2)

print(a)
```

```
Outputs:

[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

Multi Dimensional Arrays

Methods of an Array: *The flatten()*

- The ‘flatten’ method is useful to return copy of an array collapsed into ine dimension

Example-1: To understand the usage of the ‘flatten’ method

```
#flatten() method
a = array([[1, 2], [3, 4]])
print(a)

#Change to 1D array
a = a.flatten()
print(a)
```

Outputs:

```
[1 2 3 4]
```

Multi Dimensional Arrays

Methods of creating an 2D-Array

- Using `array()` function
- Using `ones()` and `zeroes()` functions
- Uisng `eye()` function
- Using `reshape()` function

Multi Dimensional Arrays

Creation of an 2D-Array: *array()*

Example-1:

```
a = array([[1, 2], [3, 4]])
print(a)
```

Outputs:

```
[[1, 2],
[3, 4]]
```

Multi Dimensional Arrays

Creation of an 2D-Array: ones*() & zeros()*

Syntax	zeros((r, c), dtype)
	
	ones((r, c), dtype)	

Example-1	a = ones((3, 4), float)	Produces items [[1. 1. 1. 1.] [1. 1. 1. 1.] [1. 1. 1. 1.]]
Example-2	b = zeros((3, 4), int)	Produces items [[0 0 0 0] [0 0 0 0] [0 0 0 0]]

Multi Dimensional Arrays

Creation of an 2D-Array: *The eye()*

- The eye() function creates 2D array and fills the items in the diagonal with 1's

Syntax	eye(n, dtype=datatype)	
Description	– Creates 'n' rows & 'n' cols – Default datatype is float	
Example-1	a = eye(3)	– Creates 3 rows and 3 cols [[1. 0. 0.] [0. 1. 0.] [0. 0. 1.]]

Multi Dimensional Arrays

Creation of an 2D-Array: *The reshape()*

- Used to convert 1D into 2D or nD arrays

Syntax	reshape(arrayname, (n, r, c))	
Description	arrayname – Represents the name of the array whose elements to be converted n – Numbers of arrays in the resultant array r, c – Number of rows & cols respectively	

Example-1	a = array([1, 2, 3, 4, 5, 6])
	b = reshape(a, (2, 3))
	print(b)
		Outputs:
		[[1 2 3]
		[4 5 6]]

Multi **D**imensional **A**rrays

Creation of an 2D-**A**rray: *The reshape()*

- Used to convert 1D into 2D or nD arrays

Syntax	reshape(arrayname, (n, r, c))	
Description	arrayname – Represents the name of the array whose elements to be converted n – Numbers of arrays in the resultant array r, c – Number of rows & cols respectively	

Example-2	a = arange(12)
	b = reshape(a, (2, 3, 2))
	print(b)
		Outputs:
		[[0 1]
		[2 3]
		[4 5]]
		
		[[6 7]
		[8 9]
		[10 11]]

Multi Dimensional Arrays

Indexing of an 2D-Array

Program-1: To understand indexing of 2D arrays

```
from numpy import *

#Create an 2D array with 3 rows, 3 cols
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

#Display only rows
for i in range(len(a)):
    print(a[i])

#display item by item
for i in range(len(a)):
    for j in range(len(a[i])):
        print(a[i][j], end=' ')
```

Multi Dimensional Arrays

Slicing of an 2D-Array

```
#Create an array
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
a = reshape(a, (3, 3))
print(a)
```

```
Produces:

[[1 2 3]
[4 5 6]
[7 8 9]]
```

```
a[:, :]
a[:]
a[: :]
```

```
Produces:

[[1 2 3]
[4 5 6]
[7 8 9]]
```

```
#Display 0th row
a[0, :]
```

```
#Display 0th col
a[:, 0]
```

```
#To get 0th row, 0th col item
a[0:1, 0:1]
```

Matrices **in N**umpy

Matrices in Numpy

Syntax	matrix-name = matrix(2D Array or String)	

Example-1	a = [[1, 2, 3], [4, 5, 6]]
	a = matrix(a)
	print(a)
		Outputs:
		[[1 2 3]
		[4 5 6]]

Example-2	a = matrix([[1, 2, 3], [4, 5, 6]])	Outputs: [[1 2 3] [4 5 6]]

Example-3	a = ‘1 2; 3 4; 5 6’
	b = matrix(a)
		[[1 2]
		[3 4]
		[5 6]]

Matrices in Numpy

Getting Diagonal Items

Function	diagonal(matrix)	

Example-1	#Create 3 x 3 matrix
	a = matrix("1 2 3; 4 5 6; 7 8 9")
	
	#Find the diagonal items
	d = diagonal(a)
	print(d)
		Outputs:
		[1 5 9]

Matrices in Numpy

Finding Max and Min Items

Function	max()
	min()	

Example-1	#Create 3 x 3 matrix
	a = matrix("1 2 3; 4 5 6; 7 8 9")
	
	#Print Max + Min Items
	big = a.max()
	small = a.min()
	print(big, small)
		Outputs:
		
		9 1

ΣMERTXE

Matrices in Numpy

Exercise

1. To find sum, average of elements in 2D array

2. To sort the Matrix row wise and column wise

3. To find the transpose of the matrix

4. To accept two matrices and find thier sum

5. To accept two matrices and find their product

Note: Read the matrices from the user and make the program user friendly

THANK YOU

Strings

Team Emertxe

Strings **A**nd **C**haracters

Strings And Characters

Creating Strings

Example-1	s = 'Welcome to Python'
Example-2	s = "Welcome to Python"
Example-3	s = """ Welcome to Python """
Example-4	s = ''' Welcome to Python '''
Example-5	s = "Welcome to 'Core' Python"
Example-6	s = 'Welcome to "Core" Python'
Example-7	s = "Welcome to\tCore\nPython"
Example-8	s = r"Welcome to\tCore\nPython"

Strings And Characters

Length of a String

len()	Is used to find the length of the string

Example	str = "Core Python"
	n = len(str)
	print("Len: ", n)

Strings And Characters

Indexing the Strings

- Both positive and Negative indexing is possible in Python

```
str = "Core Python"
```

```
#Method-1: Access each character using while loop
n = len(str)


i = 0
while i < n:
    print(str[i], end=' ')
    i += 1
```

```
#Method-2: Using for loop

for i in str:
    print(i, end=' ')
```

```
#Method-3: Using slicing operator

for i in str[::]:
    print(i, end='')
print()
```

```
#Method-4: Using slicing operator

#Take sthe step size as -1
for i in str[: : -1]:
    print(i, end='')
```

Strings And Characters

Slicing the Strings

str = "Core Python"		
1	str[: :]	Prints all
2	str[0: 9: 1]	Access the string from 0th to 8th element
3	str[0: 9: 2]	Access the string in the step size of 2
4	str[2: 3: 1]	Access the string from 2nd to 3rd Character
5	str[: : 2]	Access the entire string in the step size of 2
6	str[: 4:]	Access the string from 0th to 3rd location in steps of 1
7	str[-4: -1:]	Access from str[-4] to str[-2] from left to right
8	str[-6: :]	Access from -6 till the end of the string
9	str[-1: -4: -1]	When stepsize is negative, then the items are counted from right to left
10	str[-1: : -1]	Retrieve items from str[-1] till the first element from right to left

Strings And Characters

Repeating the Strings

- The repetition operator * is used for repeating the strings

Example-1	`str = "Core Python"`
	`print(str * 2)`

Example-2	`print(str[5: 7] * 2)`

Strings And Characters

Concatenation of Strings

- `+ is used as a concatenation operator`

Example-1

```
s1 = "Core"
s2 = "Python"
s3 = s1 + s2
```

Strings And Characters

Membership Operator

- We can check, if a string or a character is a member of another string or not using 'in' or 'not in' operator

- 'in' or 'not in' makes case sensitive comaprisons

Example-1

```
str = input("Enter the first string: ")
sub = input("Enter the second string: ")

if sub in str:
    print(sub+" is found in main string")
else:
    print(sub+" is not found in main string")
```

Strings And Characters
Removing Spaces

str = " Ram Ravi "	

lstrip()	#Removes spaces from the left side
	print(str.lstrip())

rstrip()	#Removes spaces from the right side
	print(str.rstrip())

strip()	#Removes spaces from the both sides
	print(str.strip())

Strings And Characters
Finding the Sub-Strings

- Methods useful for finding the strings in the main string
  - – find()
  - – rfind()
  - – index()
  - – rindex()
  - find(), index() will search for the sub-string from the begining
  - rfind(), rindex() will search for the sub-string from the end
  - find(): Returns -1, if sub-string is not found
  - index(): Returns 'ValueError' if the sub-string is not found

Strings And Characters

Finding the Sub-Strings

Syntax

```
mainstring.find(substring, beg, end)
```

Example

```
str = input("Enter the main string:")
sub = input("Enter the sub string:")

#Search for the sub-string
n = str.find(sub, 0, len(str))

if n == -1:
    print("Sub string not found")
else:
    print("Sub string found @: ", n + 1)
```

Strings And Characters
Finding the Sub-Strings

Syntax

```
mainstring.index(substring, beg, end)
```

Example

```
str = input("Enter the main string:")
sub = input("Enter the sub string:")

#Search for the sub-string
try:
    #Search for the sub-string
    n = str.index(sub, 0, len(str))
except ValueError:
    print("Sub string not found")
else:
    print("Sub string found @: ", n + 1)
```

Strings And Characters

Finding the Sub-Strings: Exercise

1 To display all positions of a sub-string in a given main string

Strings And Characters

Counting Sub-Strings in a String

count()	To count the number of occurrences of a sub-string in a main string
Syntax	stringname.count(substring, beg, end)

Example-1	str = “New Delhi”
	n = str.count(‘Delhi’)

Example-2	str = “New Delhi”
	n = str.count(‘e’, 0, 3)

Example-3	str = “New Delhi”
	n = str.count(‘e’, 0, len(str))

Strings And Characters

Strings are Immutable

- Immutable object is an object whose content cannot be changed

Immutable	Numbers, Strings, Tuples
Mutable	Lists, Sets, Dictionaries

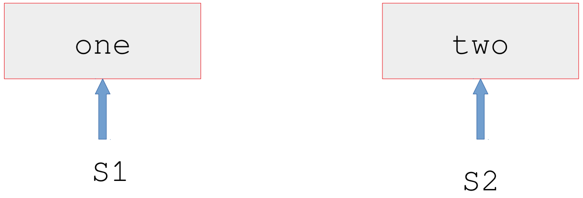
Reasons: Why strings are made immutable in Python	
Performance	Takes less time to allocate the memory for the Immutable objects, since their memory size is fixed
Security	Any attempt to modify the string will lead to the creation of new object in memory and hence ID changes which can be tracked easily

Strings And Characters

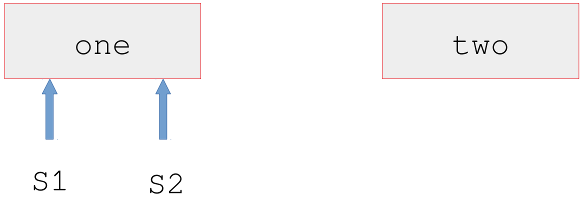
Strings are Immutable

- Immutable object is an object whose content cannot be changed

- Example:
  - s1 = “one”
  - s2 = “two”



  - S2 = s1



Strings And Characters

Replacing String with another String

replace()	To replace the sub-string with another sub-string
Syntax	stringname.replace(old, new)

Example	str = "Ram is good boy"
	str1 = str.replace("good", "handsome")
	print(str1)

Strings And Characters

Splitting And Joining Strings

split()	- Used to brake the strings - Pieces are returned as a list
Syntax	stringname.split(‘character’)
Example	str = "one,two,three" lst = str.split(',')

join()	- Groups into one sring
Syntax	separator.join(str) - separator: Represents the character to be used between two strings - str: Represents tuple or list of strings

Example	str = ("one", "two", "three")
	str1 = "-".join(str)

Strings And Characters

Changing the Case of the Strings

Methods	upper()
	lower()
	swapcase()
	title()	

str = "Python is the future"		
upper()	print(str.upper())	PYTHON IS THE FUTURE
lower()	print(str.lower())	python is the future
swapcase()	print(str.swapcase())	pYTHON IS THE FUTURE
title()	print(str.title())	Python Is The Future

Strings And Characters

Check: Starting & Ending of Strings

Methods	startswith()
	endswith()	

str = "This is a Python"		
startswith()	print(str.startswith("This"))	True
endswith()	print(str.endswith("This"))	False

Strings And Characters

String Testing Methods

isalnum()	Returns True, if all characters in the string are alphanumeric(A – Z, a – z, 0 – 9) and there is atleast one character
isalpha()	Returns True, if the string has atleast one character and all characters are alphabets(A – Z, a – z)
isdigit()	Returns True if the string contains only numeric digits(0-9) and False otherwise
islower()	Returns True if the string contains at least one letter and all characters are in lower case; otherwise it returns False
isupper()	Returns True if the string contains at least one letter and all characters are in upper case; otherwise it returns False
istitle()	Returns True if each word of the string starts with a capital letter and there at least one character in the string; otherwise it returns False
isspace()	Returns True if the string contains only spaces; otherwise, it returns False

Strings And Characters

Formatting the strings

format()	Presenting the string in the clearly understandable manner
Syntax	"format string with replacement fields". format(values)

```
id = 10
name = "Ram"
sal = 19000.45
```

```
print("{}, {}, {}". format(id, name, sal))
```

```
print("{}-{}-{}". format(id, name, sal))
```

```
print("ID: {0}\tName: {1}\tSal: {2}\n". format(id, name, sal))
```

```
print("ID: {2}\tName: {0}\tSal: {1}\n". format(id, name, sal))
```

```
print("ID: {two}\tName: {zero}\tSal: {one}\n". format(zero=id, one=name, two=sal))
```

```
print("ID: {:d}\tName: {:s}\tSal: {:10.2f}\n". format(id, name, sal))
```

Strings And Characters

Formatting the strings

format()	Presenting the string in the clearly understandable manner
Syntax	"format string with replacement fields". format(values)
n = 5000	
print("{:*>15d}". format(num))	
print("{:*^15d}". format(num))	

Strings And Characters

Exercise

1. To know the type of character entered by the user

2. To sort the strings in alphabetical order

3. To search for the position for a string in agiven group of strings

4. To find the number of words in a given strings

5. To insert the sub-string into a main string in a particular position

THANK YOU

Functions

Team Emertxe

EMERTXE

Function **vs M**ethod

Function vs Method

- A function can be written individually in a Python
- Function is called using its name
- A function within the class is called “Method”
- Method is called in two ways,
  - objectname.methodname()
  - Classname.methodname()

Function & Method are same except thier placement and the way they are called

Defining & **C**alling a Function

Defining & Calling

Syntax

```
def function_name(para1, para2, para3,...)
     """ docstring """
     statements
```

Example

```
def sum(a, b):
    """ This function finds sum of two numbers """
    c = a + b
    print('Sum= ', c)

#call the function
sum(10, 15)
```

Returning **V**alue/s **F**rom a Function

Returning a Value

Example	Description
return c	Returns c from the function
return 100	Returns constant from a function
return lst	Return thelist that contains values
return z, y, z	Returns more than one value

Returning a Value

Example

```
# A function to add two numbers
def sum(a, b):
    """ This function finds sum of two numbers """
    c = a + b
    return c # return result

#call the function
x = sum(10, 15)
print('The Sum is: ', x)

y = sum(1.5, 10.75)
print('The Sum is: ', y)
```

Returning ‘M’ Values

Example

```
# A function that returns two results
def sum_sub(a, b):
    """ this function returns results of
    addition and subtraction of a, b """
    c = a + b
    d = a - b
    return c, d

# get the results from sum_sub() function
x, y = sum_sub(10, 5)

# display the results
print("Result of addition: ", x)
print("Result of subtraction: ", y)
```

Functions are First Class Objects

Functions

First Class Objects

- Functions are considered as first class objects
- When function is defined, python interpreter internally creates an Object
- Noteworthy:
  - It is possible to assign a function to a variable
  - It is possible to define one function inside another function
  - It is possible to pass a function as parameter to a another function
  - It is possible that a function can return another function

Pass by Object **R**eferences

Functions

Pass by Object References

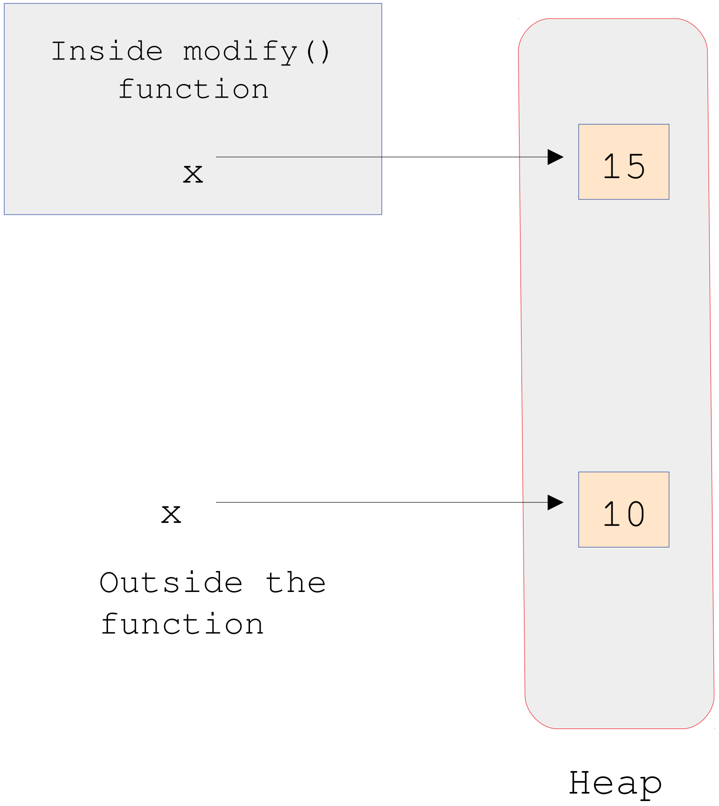
- The values are sent to the function by means of Object References
- Objects are created on heap memory at run time
- Location of the object can be obtained by using id( ) function

Functions

Pass by Object References

- Example-1: To pass an integer to a function and modify it

```
# passing an integer to a function
def modify(x):
    """ reassign a value to the variable """
    x = 15
    print(x, id(x))

# call mofify() and pass x
x = 10
modify(x)
print(x, id(x))
```

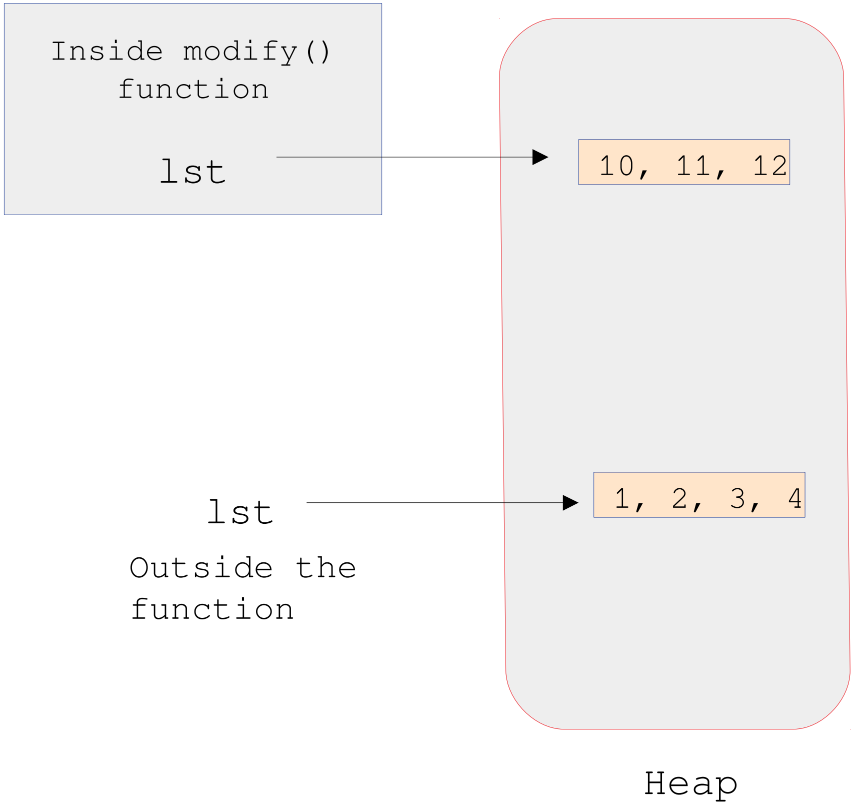


Functions

Pass by Object References

- Example-1: To pass a list to a function and modify it

```
# passing an list to a function
def modify(lst):
    """ to create a new list """
    lst = [10, 11, 12]
    print(lst, id(lst))

# call mofify() and pass lst
lst = [1, 2, 3, 4]
modify(lst)
print(lst, id(lst))
```

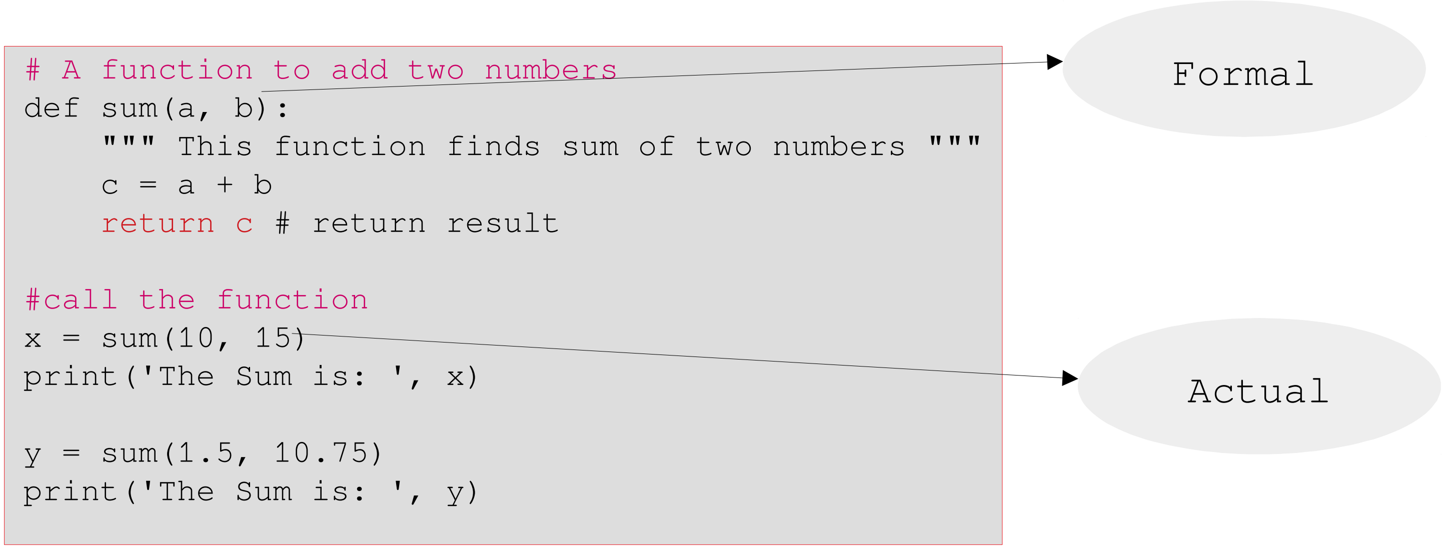


If altogether a new object inside a function is created, then it will not be available outside the function

Formal and **A**ctual **A**rguments

Functions

Formal and Actual Arguments



```
# A function to add two numbers
def sum(a, b):
    """ This function finds sum of two numbers """
    c = a + b
    return c # return result

#call the function
x = sum(10, 15)
print('The Sum is: ', x)

y = sum(1.5, 10.75)
print('The Sum is: ', y)
```

Formal

Actual

Functions

Formal and Actual Arguments

- Types: Actual Arguments
  - Positional
  - Keyword
  - Default
  - Variable length

Actual Arguments
Positional Arguments

- Arguments are passed to a function in correct positional order

```
# positional arguments demo
def attach(s1, s2):
    """ to joins1 and s2 and display total string """
    s3 = s1 + s2
    print('Total string: ' + s3)

# call attach() and pass 2 strings
attach('New','York') # positional arguments
```

Actual Arguments
Keyword Arguments

- Keyword Arguments are arguments that identify the parameters by their names

```
# key word arguments demo
def grocery(item, price):
    """ to display the given arguments """
    print('Item = %s' % item)
    print('Price = %.2f' % price)

# call grocerry() and pass two arguments
grocery(item='sugar', price = 50.75) #keyword arguments
grocery(price = 88.00, item = 'oil') #keyword arguments
```

Actual Arguments
Variable Length Arguments-1

- An argument that can accept any number of arguments
- Syntax: `def function_name(farg, *args)`
  - – farg: Formal argument
  - – *args: Can take 1 or more arguments
- *args will be stored as tuple

```
# variable length arguments demo
def add(farg, *args): # *args can take 1 or more values
    """ to add given numbers """
    print('Formal arguments= ', farg)

    sum = 0
    for i in args:
        sum += i
    print('Sum of all numbers= ', (farg + sum))

# call add() and pass arguments
add(5, 10)
add(5, 10, 20, 30)
```

Actual Arguments
Variable Length Arguments-2

- An argument that can accept any number of values provided in the format of keys and values
- Syntax: `def function_name(farg, **kwargs)`
  - – farg: Formal argument
  - – **kwargs:
    - – Called as keyword variable
    - – Internally represents dictionary object
- **kwargs will be stored as dictionary

```
# keyword variable argument demo
def display(farg, **kwargs): # **kwargs can take 0 or more values
    """ to add given values """
    print('Formal arguments= ', farg)

    for x, y in kwargs.items(): # items() will give pair of items
        print('key = {}, value = {}'.format(x, y))

# pass 1 formal argument and 2 keyword arguments
display(5, rno = 10)
print()
#pass 1 formal argument and 4 keyword arguments
display(5, rno = 10, name = 'Prakesh')
```

Local and **G**lobal **V**ariables

Local & Global Vars

The Global Keyword

- The global variable can be accessed inside the function using the global keyword
  - – global var

```
# accesing the global variable from inside a function
a = 1 # this is global variable
def myfunction():
    global a # this is global variable
    print('global a =', a) # display global variable
    a = 2 # modify global variable value
    print('modified a =', a) # display new value

myfunction()
print('global a =', a) # display modified value
```

Local & Global Vars

The Global Keyword

- Syntax to get a copy of the global variable inside the function and work on it
  - – globals()[“global_var_name”]

```
#same name for global and local variable
a = 1 # this is global variable:
def myfunction():
    a = 2 # a is local var
    x = globals()['a'] # get global var into x
    print('global var a =', x) # display global variable
    print('local a =', a) # display new value

myfunction()
print('global a =', a)
```

Passing Group of Items to a Function

Passing

The Group Of Items

- To pass the group of items to a function, accept them as a list and then pass it.

Example-1

```
# a function to find total and average
def calculate(lst):
    """ to find total and average """
    n = len(lst)
    sum = 0
    for i in lst:
        sum += i
        avg = sum / n
    return sum, avg

# take a group of integers from keyboard
print('Enter numbers seperated by space: ')
lst = [int(x) for x in input().split()]

#call calculate() and pass the list
x, y = calculate(lst)
print('Total: ', x)
print('Average: ', y)
```

Recursive **F**unction

Recursions

- A function calling itself is called as Recursions

Example-1

```
# resursive function to calculate factorial
def factorial(n):
    """ to find factorial of n """
    if n == 0:
        result = 1
    else:
        result = n * factorial(n - 1)
    return result

# find factorial values for first 10 numbers
for i in range(1, 11):
    print('Factorial of {} is {}'.format(i, factorial(i)))
```

Anonymous Functions Or **Lambdas**

Lambdas

- A function without name is called ‘Anonymous functions’
- Anonymous functions are not defined using the ‘def’ keyword
- Defined using the keyword ‘lambda’, hence called as lambda function
- Example

Normal Function	Anonymous Function

`def square(x):`
`    return x * x`
	`f = lambda x: x * x`
	
	`#f = function name`

`#Calling the function`
 	
`square(x)`
	`#Calling the function`
	
	`value = f(5)`

- Syntax

```
lambda argument_list: expression
```

Lambdas

Example

- Example

```
# lambda function to calculate square value

f = lambda x: x*x # write lambda function

value = f(5) # call lambda func

print('Square of 5 = ', value) # display result
```

Lambdas

using lambdas with filter()

- A filter() is useful to filter out the elements of a sequence depending on the result of a function
- Syntax: filter(function, sequence)
- Example

```
def is_even(x):
    if x % 2 == 0:
        return True
    else:
        return False
```

- filter (is_even, lst)

  - is_even function acts on every element on the lst

Lambdas

using lambdas with filter()

- Example

```
# a normal function that returns
# even numbers from a list
def is_even(x):
    if x % 2 == 0:
        return True
    else:
        return False

# let us take a list of numbers
lst = [10, 23, 45, 46, 70, 99]

# call filter() eith is_even() and list
lstl = list(filter(is_even, lst))
print(lstl)
```

```
# lambda function that returns even numbers from list
lst = [10, 23, 45, 46, 70, 99]
lstl = list(filter(lambda x: (x % 2 == 0), lst))

print(lstl)
```

Lambdas

using lambdas with map()

- A map() is similar to filter(), but it acts on each element of the sequence and changes the items
- Syntax: `map(function, sequence)`
- Example

Normal Function	Lambdas

#map() function that gives squares
def squares(x):
    return x * x
 	
#let us take a lis of numbers
lst = [1, 2, 3, 4, 5]
 	
# call map() with square()s and lst 	
lstl = list(map(squares, lst))
print(lstl)
	# lambda that returns squares
	lst = [1, 2, 3, 4, 5]
	lstl = list(map(lambda x: x * x, lst))
	print(lstl)

Writing using the lambdas will be more elegant

Lambdas

using lambdas with reduce()

- A reduce() reduces a sequence of elements to a single value by processing the elements according to the function supplied
- `Syntax: reduce(function, sequence)`
- Example

Lambdas

```
# lambda that returns products of elements of a list

from functools import *

lst = [1, 2, 3, 4, 5]

result = reduce(lambda x, y: x * y, lst)

print(result)
```

`import functools, since reduce() belongs to functools`

Lambdas

using lambdas with reduce(): Exercise

Problem

To calculate sum of numbers from 1 to 50 using reduce() & lambda functions

import functools, since reduce() belongs to functools

Function **D**ecorators

Function Decorators

- A decorator is a function that accepts a function as parameter and returns a function
- A decorator takes the result of a function, modifies and returns it

Function Decorators

Steps to Create Decorators

- STEP-1: Define the decorator

```
def decor(fun):
```

- STEP-2: Define the function inside the decorator

```
def decor(fun):
     def inner():
          value = fun()
          return value + 2
     return inner
```

- STEP-3: Define one function

```
def num():
     return 10
```

- STEP-4: Call the decorator

```
res = decor(num)
```

Function Decorators

Complete Program

```
# a decorator that increments the value of a function by 2
def decor(fun):     #this is decorator func
    def inner():    #this is inner func that modifies
        value = fun()
        return value + 2
    return inner    # return inner function

# take a function to which decorator should be applied
def num():
    return 10

#call decorator func and pass me
result_fun = decor(num)     # result_fun represents ''inner function
print(result_fun())         # call result_fun and display
```

Function Decorators

@ decor

- To apply the decorator to a function

```
@decor
def num():
     return 10
```

  - It means decor() is applied to process or decorate the result of the num() function
  - No need to call decorator explicitly and pass the function name
  - @ is useful to call the decorator function internally

Function Decorators

@ decor: Example

```
# a decorator that increments the value of a function by 2
def decor(fun):      #this is decorator func
    def inner():     #this is inner func that modifies
        value = fun()
        return value + 2
    return inner     # return inner function

# take a function to which decorator should be applied
@decor #apply decor to the below function
def num():
    return 10

#call num() function and display its result
print(num())
```

Function Decorators

@ decor: More than one decorator

```
# a decorator that increments the value of a function by 2
def decor(fun):      #this is decorator func
    def inner():     #inner func that modifies
        value = fun()
        return value + 2
    return inner     # return inner function
```

```
# a decorator that doubles the value of a function
def decor1(fun):          #this is decorator func
    def inner():          #Inner func that modifies
        value = fun()
        return value * 2
    return inner          # return inner function
```

```
# take a function to which decorator should be applied
@decor
@decor1
def num():
    return 10
```

```
#call num() function and apply decor1 and then decor
print(num())
```

Without using @, decorators can be called

Function **G**enerators

Function Generators

- Generator: Function that returns sequence of values
- It is written like ordinary function but it uses ‘yield’ statement

```
# generator that returns sequence from x and y
def mygen(x, y):
    while x <= y:
        yield x
        x += 1

# fill generator object with 5 and 10
g = mygen(5, 10)

# display all numbers in the generator
for i in g:
    print(i, end=' ')
```

To retrieve element by element from a generator object, use **next()** function

Creating Our Own Modules in Python

Creating
Own Modules in Python

- A module represents a group of

1. Classes
2. Methods
3. Functions
4. Variables

- Modules can be reused
- Types:
  - Built-in: sys, io, time ...
  - User-defined

Creating

Own Modules in Python: Example

employee.py

```
# to calculate dearness allowance
def da(basic):
    """ da is 80% of basic salary """
    da = basic * 80 / 100
    return da
```

```
# to calculate house rent allowance
def hra(basic):
    """ hra is 15% of basic salary """
    hra = basic * 15 / 100
    return hra
```

```
# to calculate provident fund amount
def pf(basic):
    """ pf is 12% of basic salary """
    pf = basic * 12 / 100
    return pf
```

```
# to calculate income tax
def itax(gross):
    """ tax is calculated
        at 10% on gross """
    tax = gross * 0.1
    return tax
```

usage.py

```
from employee import *


# calculate gross salary of employee by taking
basic
basic= float(input('Enter basic salary: '))


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


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

The Special Variable __name__

The Special Variable

__name__

- It is internally created, when program is executed
- Stores information regarding whether the program is executed as an individual program or as a module
- When a program is executed directly, it stores __main__
- When a program is executed as a module, the python interpreter stores module name

The Special Variable

__name__ : Example-1

```
#python program to display message. save this as one.py
def display():
    print('Hello Python')


if __name__ == '__main__':
    display()   # call display func
    print('This code is run as a program')
else:
    print('This code is run as a module')
```

The Special Variable

__name__ : Example-2

```
# in this program one.py is imported as a module. save this as two.py

import one

one.display() # call module one's display function.
```

THANK YOU

List And Tuples

Team Emertxe

List

List

Introduction

- Used for storing different types of data unlike arrays

Example-1	student = [10, "Amar", 'M', 50, 55, 57, 67, 47]
Example-2	e_list = [] #Empty List

- Indexing + Slicing can be applied on list

Example-1	print(student[1])	Gives "Amar"
Example-2	print(student[0: 3: 1])	Prints [10, "Amar", 'M']
Example-3	student[::]	Print all elements

List

Examples

Example-1	#Create list with integer numbers
	num = [10, 20, 30, 40, 50]
	print(num)
	print("num[0]: %d\tnum[2]: %d\n" % (num[0], num[2]))

Example-2	#Create list with strings
	names = ["Ram", "Amar", "Thomas"]
	print(names)
	print("names[0]: %s\tnames[2]: %s\n" % (names[0], names[2]))

Example-3	#Create list with different dtypes
	x = [10, 20, 1.5, 6.7, "Ram", 'M']
	print(x)
	print("x[0]: %d\tx[2]: %f\tx[4]: %s\tx[5]: %c\n" %(x[0], x[2], x[4], x[5]))

List

Creating list using range()

Example

```
#Create list
num = list(range(4, 9, 2))
print(num)
```

List

Updating list

1	Creation	lst = list(range(1, 5))
		print(lst)
			[1, 2, 3, 4]

2	append	lst.append(9)
		print(lst)
			[1, 2, 3, 4, 9]

3	Update-1	lst[1] = 8
		print(lst)
			[1, 8, 3, 4, 9]

4	Update-2	lst[1: 3] = 10, 11 print(lst)	[1, 10, 11, 4, 9]
5	delete	del lst[1] print(lst)	[1, 11, 4, 9]

6	remove	lst.remove(11)
		print(lst)
			[1, 4, 9]

7	reverse	lst.reverse()
		print(lst)
			[9, 4, 1]

List

Concatenation of Two List

'+' operator is used to join two list

Example

```
x = [10, 20, 30]
y = [5, 6, 7]
print(x + y)
```

List

Repetition of List

'*' is used to repeat the list 'n' times	

Example	x = [10, 20, 30]
	print(x * 2)

List

Membership of List

'in' and 'not in' operators are used to check, whether an element belongs to the list or not		

Example	x = [1, 2, 3, 4, 5]
	a = 3
	print(a in x)
		Returns True, if the item is found in the list

Example	x = [1, 2, 3, 4, 5]
	a = 7
	print(a not in x)
		Returns True, if the item is not found in the list

List

Aliasing And Cloning Lists

Aliasing: Giving new name for the existing list	

Example	x = [10, 20, 30, 40]
	y = x
	Note: No separate memory will be allocated for y

Cloning / Copy: Making a copy	

Example	x = [10, 20, 30, 40]
	y = x[:] <=> y = x.copy()
	x[1] = 99
	print(x)
	print(y)
	Note: Changes made in one list will not reflect other

List

Exercise

1. To find the maximum & minimum item in a list of items

2. Implement Bubble sort

3. To know how many times an element occurred in the list

4. To create employee list and search for the particular employee

List

To find the common items

```
#To find the common item in two lists

l1 = ["Thomas", "Richard", "Purdie", "Chris"]
l2 = ["Ram", "Amar", "Anthony", "Richard"]

#Covert them into sets
s1 = set(l1)
s2 = set(l2)

#Filter intersection of two sets
s3 = s1.intersection(s2)

#Convert back into the list
common = list(s3)

print(common)
```

List

Nested List

```
#To create a list with another list as element

list = [10, 20, 30, [80, 90]]

print(list)
```

List

List Comprehensions

- List comprehensions represent creation of new lists from an iterable object(list, set,
- tuple, dictionary or range) that satisfies a given condition

Example-1: Create a list with squares of integers from 1 to 10

```
#Version-1
squares = []
for x in range(1, 11):
    squares.append(x ** 2)

print(squares)
```

```
#Version-2
squares = []
squares = [x ** 2 for x in range(1, 11)]
print(squares)
```

List

List Comprehensions

- List comprehensions represent creation of new lists from an iterable object(list, set,
- tuple, dictionary or range) that satisfies a given condition

```
Example-2: Get squares of integers from 1 to 10 and take only the even numbers from the
result

even_squares = [x ** 2 for x in range(1, 11) if x % 2 == 0]

print(even_squares)
```

List

List Comprehensions

- List comprehensions represent creation of new lists from an iterable object(list, set,
- tuple, dictionary or range) that satisfies a given condition

```
Example-3: #Adding the elements of two list one by one
```

```
#Example-1
x = [10, 20, 30]
y = [1, 2, 3, 4]

lst = []

#Version-1
for i in x:
    for j in y:
        lst.append(i + j)

#Version-2
lst = [i + j for i in x for j in y]
```

```
#Example-2
lst = [i + j for i in "ABC" for j in "DE"]
print(lst)
```

Tuple

Tuple

Introduction

- A tuple is similar to list but it is immutable

Tuple

Creating Tuples

To create empty tuple

```
tup1 = ()
```

Tuple with one item

```
tup1 = (10, )
```

Tuple with different dtypes

```
tup3 = (10, 20, 1.1, 2.3, "Ram", 'M')
```

Tuple with no braces

```
t4 = 10, 20, 30, 40
```

Create tuple from the list

```
list = [10, 1.2, "Ram", 'M']
t5 = tuple(list)
```

Create tuple from range

```
t6 = tuple(range(4, 10, 2))
```

Tuple

Accessing Tuples

- Accessing items in the tuple can be done by indexing or slicing method, similar to that of list

Tuple

Basic Operations On Tuples

```
s = (10, "Ram", 10, 20, 30, 40, 50)
```

```
To find the length of the tuple

print(len(s))
```

```
Repetition operator

fee = (25.000, ) * 4
print(fee)
```

```
Concatenate the tuples using *

ns = s + fee
print(ns)
```

```
Membership

name = "Ram"
print(name in s)
```

```
Repetition

t1 = (1, 2, 3)
t2 = t1 * 3
print(t2)
```

Tuple

Functions To Process Tuples

len()	len(tpl)	Returns the number of elements in the tuple
min()	min(tpl)	Returns the smallest element in the tuple
max()	max()	Returns the biggest element in the tuple
count()	tpl.count(x)	Returns how many times the element ‘x’ is found in the tuple
index()	tpl.index(x)	Returns the first occurrence of the element ‘x’ in tpl. Raises ValueError if ‘x’ is not found in the tuple
sorted()	sorted(tpl)	Sorts the elements of the tuple into ascending order. sorted(tpl, reverse=True) will sort in reverse order

Tuple

Exercise

1. To accept elements in the form of a a tuple and display thier sum and average

2. To find the first occurrence of an element in a tuple

3. To sort a tuple with nested tuples

4. To insert a new item into a tuple at a specified location

5. To modify or replace an existing item of a tuple with new item

6. To delete an element from a particular position in the tuple

THANK YOU

Dictionaries

Team Emertxe

Dictionaries

Introduction

Group of items arranged in the form of key-value pair

Example

```
d = {"Name": "Ram", "ID": 102, "Salary": 10000}
```

Program

```
#Print the entire dictionary
print(d)

#Print only the keys
print("Keys in dic: ", d.keys())

#Print only values
print("Values: ", d.values())

#Print both keys and value pairs as tuples
print(d.items())
```

Dictionaries

Operations

```
d = {"Name": "Ram", "ID": 102, "Salary": 10000}
```

1. To get the no. of pairs in the Dictionary	`n = len(d)`
2. To modify the existing value	`d[salary] = 15000`
3. To insert new key:value pair	`d["Dept"] = "Finance"`
4. To delete the key:value pair	`del d["ID"]`
5. To check whether the key is present in dictionary	`"Dept" in d` - Returns True, if it is present

6. We can use any datatype fro values, but keys should obey the rules

R1: Keys should be unique

```
        Ex: emp = {10: "Ram", 20: "Ravi", 10: "Rahim"}
        - Old value will be overwritten,
        emp = {10: "Rahim", 20: "Ravi"}
```

R2: Keys should be immutable type. Use numbers, strings or tuples

        If mutable keys are used, will get 'TypeError'

Dictionaries

Methods

clear()	d.clear()	Removes all key-value pairs from the d
copy()	d1 = d.copy()	Copies all items from ‘d’ into a new dictionary ‘d1’
fromkeys()	d.fromkeyss(s, [,v])	Create a new dictionary with keys from sequence ‘s’ and values all set to ‘v’
get()	d.get(k, [,v])	Returns the value associated with key ‘k’. If key is not found, it returns ‘v’
items()	d.items()	Returns an object that contains key-value pairs of ‘d’. The pairs are stored as tuples in the object
keys()	d.keys()	Returns a sequence of keys from the dictionary ‘d’
values()	d.values()	Returns a sequence of values from the dictionary ‘d’
update()	d.update(x)	Adds all elements from dictionary ‘x’ to ‘d’
pop()	d.pop(k, [,v])	Removes the key ‘k’ and its value.

Dictionaries

Programs

```
To create the dictionary with employee details
```

```
d = {"Name": "Ram", "ID": 1023, "Salary": 10000}

#Print the entire dictionary
print(d)

#Print only the keys
print("Keys in dic: ", d.keys())

#Print only values
print("Values: ", d.values())

#Print both keys and value pairs as tuples
print(d.items())
```

Dictionaries

Programs

```
#To create a dictionary from the keyboard and display the items


x = {}

print("Enter 'n' value: ", end='')
n = int(input())

for i in range(n):
    print("Enter the key: ", end='')
    k = input()
    print("Enter the value: ", end='')
    v = int(input())
    x.update({k: v})

print(x)
```

Dictionaries

Using for loop with Dictionaries

Method-1

```
for k in colors:
    print(k)
```

Method-2

```
for k in colors:
    print(colors[k])
```

Method-3

```
for k, v in colors.items():
    print("key = {}\nValue = {}". format(k, v))
```

Dictionaries

Sorting Dictionaries: Exercise

```
To sort the elements of a dictionary based on akey or value
```

Dictionaries

Converting Lists into Dictionary

```
Two step procedure
    - zip()
    - dict()
```

```
#To convert list into dictionary


countries = ["India", "USA"]

cities = ["New Delhi", "Washington"]


#Make a dictionary

z = zip(countries, cities)

d = dict(z)


print(d)
```

Dictionaries

Converting strings into dictionary

```
str = "Ram=23,Ganesh=20"

#Create the empty list
lst = []

for x in str.split(','):
    y = x.split('=')
    lst.append(y)

#Convert into dictionary
d = dict(lst)

print(d)
```

Dictionaries

Passing dictionary to function

By specifying the name of the dictionary as the parameter, we can pass the dictionary to the function.

Example

```
d = {10: "Ram"}

display(d)
```

Dictionaries

Ordered Dictionaries

```
from collections import OrderedDict
```

Example

```
d = {10: "Ram"}

display(d)
```

Program:

```
#To create the ordered dictionary
from collections import OrderedDict

#Create empty dictionary
d = OrderedDict()

d[10] = 'A'
d[11] = 'B'
d[12] = 'C'
d[13] = 'D'

print(d)
```

THANK YOU

Classes **A**nd **O**bjects

Team **E**mertxe

ΣMERTXE

Creation of **C**lass

Creation of Class

General Format

- Class is a model or plan to create the objects
- Class contains,
  - Attributes: Represented by variables
  - Actions   : Performed on methods
- Syntax of defining the class,

Syntax

```
class Classname(object):
        """docstrings"""

        Attributes

        def __init__(self):
        def method1():
        def method2():
```

Example

```
class Student:
        """The below block defines attributes"""
        def __init__(self):
            self.name = "Ram"
            self.age = 21
            self.marks = 89.75

        """The below block defines a method"""
        def putdata(self):
            print("Name: ", self.name)
            print("Age: ", self.age)
            print("Marks: ", self.marks)
```

Creation of Class

Program

```
#To define the Student calss and create an Object to it.

#Class Definition
class Student:
    #Special method called constructor
    def __init__(self):
        self.name = "Ram"
        self.age = 21
        self.marks = 75.90

    #This is an instance method
    def putdata(self):
        print("Name: ", self.name)
        print("Age: ", self.age)
        print("Marks: ", self.marks)

#Create an instance to the student class
s = Student()

#Call the method using an Object
s.putdata()
```

The **Self** Variable

The **Self** Variable

- ‘Self’ is the default variable that contains the memory address of the instance of the current class

s1 = Student()	• s1 contains the memory address of the instance • This memory address is internally and by default passed to ‘self’ variable
Usage-1: def __init__(self):	• The ‘self’ variable is used as first parameter in the constructor
Usage-2: def putdata(self):	• The ‘self’ variable is used as first parameter in the instance methods

Constructor

Constructor

Constructor with **NO** parameter

- Constructors are used to create and initialize the 'Instance Variables'

Example

```
def __init__(self):
        self.name = "Ram"
        self.marks = 99
```

- Constructor will be called only once i.e at the time of creating the objects
- s = Student()

Constructor

Constructor with parameter

Example	def __init__(self, n = "", m = 0):
	        self.name = n
	        self.marks = m

Instance-1	s = Student() Will initialize the instance variables with default parameters
Instance-2	s = Student("Ram", 99) Will initialize the instance variables with parameters passed

Constructor

Program

```
#To create Student class with a constructor having more than one parameter

class Student:
    #Constructor definition
    def __init__(self, n = "", m = 0):
        self.name = n
        self.marks = m

    #Instance method
    def putdata(self):
        print("Name: ", self.name)
        print("Marks: ", self.marks)

#Constructor called without any parameters
s = Student()
s.putdata()

#Constructor called with parameters
s = Student("Ram", 99)
s.putdata()
```

Types of **V**ariables

Types Of Variables

- Instance variables
- Class / Static variables

Types Of Variables

Instance Variables

- Variables whose separate copy is created for every instance/object
- These are defined and init using the constructor with 'self' parameter
- Accessing the instance variables from outside the class,
  - instancename.variable

```
class Sample:
    def __init__(self):
        self.x = 10

    def modify(self):
        self.x += 1
```

```
#Create an objects
s1 = Sample()
s2 = Sample()

print("s1.x: ", s1.x)
print("s2.x: ", s2.x)

s1.modify()
print("s1.x: ", s1.x)
print("s2.x: ", s2.x)
```

Types Of Variables

Class Variables

- Single copy is created for all instances
- Accessing class vars are possible only by 'class methods'
- Accessing class vars from outside the class,
  - classname.variable

```
class Sample:
    #Define class var here
    x = 10

    @classmethod
    def modify(cls):
        cls.x += 1
```

```
#Create an objects
s1 = Sample()
s2 = Sample()

print("s1.x: ", s1.x)
print("s2.x: ", s2.x)

s1.modify()
print("s1.x: ", s1.x)
print("s2.x: ", s2.x)
```

Namespaces

Namespaces

Introduction

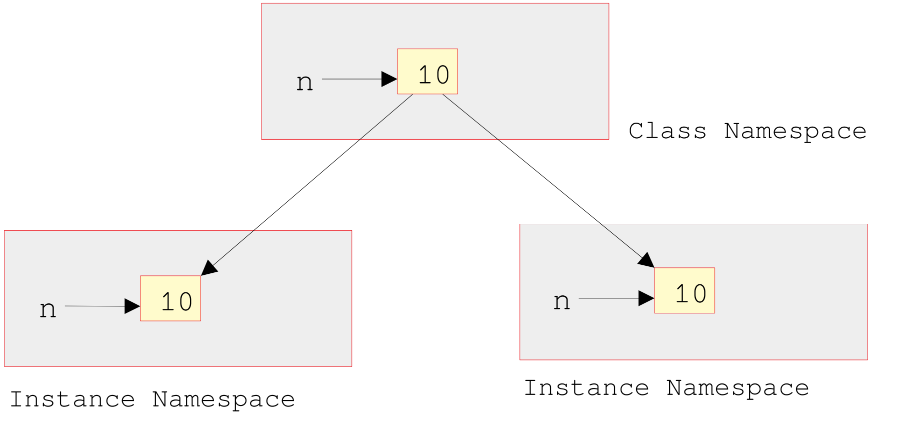
- Namespace represents the memory block where names are mapped/linked to objects
- Types:
  - Class namespace
    - – The names are mapped to class variables
  - Instance namespace
    - – The names are mapped to instance variables

Namespaces

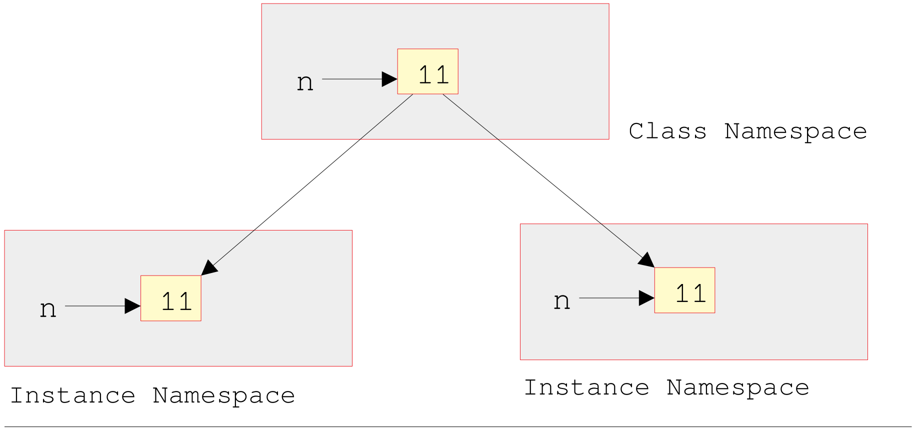
Class Namespace

```
#To understand class namespace

#Create the class
class Student:
    #Create class var
    n = 10

#Access class var in class namespace
print(Student.n)

#Modify in class namespace
Student.n += 1

#Access class var in class namespace
print(Student.n)

#Access class var in all instances
s1 = Student()
s2 = Student()

#Access class var in instance namespace
print("s1.n: ", s1.n)
print("s2.n: ", s2.n)
```

Before modifyng class variable ‘n’



After modifyng class variable ‘n’



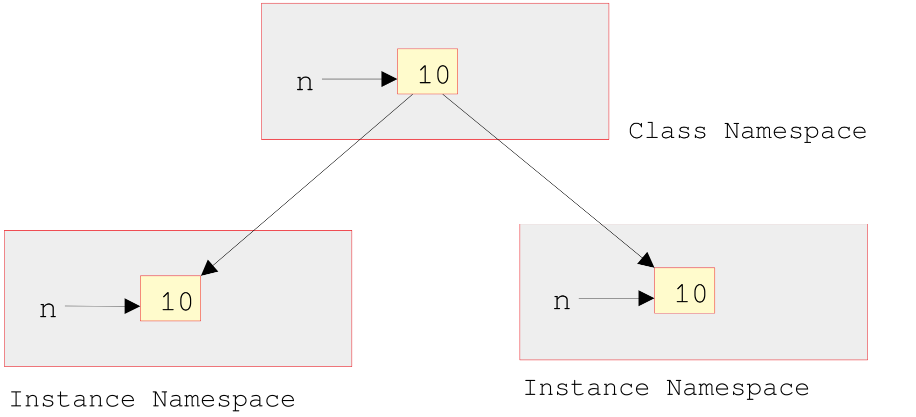
If class vars are modified in class namespace, then it reflects to all instances

Namespaces

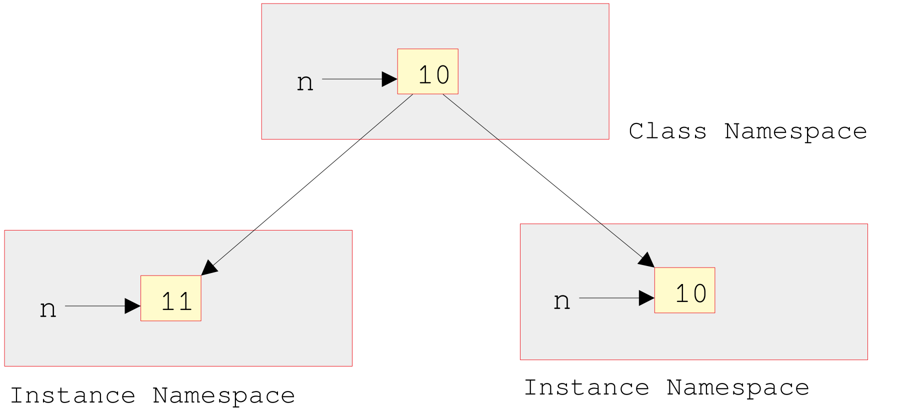
Instance Namespace

```
#To understand class namespace

#Create the class
class Student:
    #Create class var
    n = 10

s1 = Student()
s2 = Student()

#Modify the class var in instance namespace
s1.n += 1

#Access class var in instance namespace
print("s1.n: ", s1.n)
print("s2.n: ", s2.n)
```

Before modifyng class variable ‘n’



After modifyng class variable ‘n’



If class vars are modified in instance namespace, then it reflects only to that instance

Types of **M**ethods

Types of Methods

- Types:
  - Instance Methods
    - – Accessor
    - – Mutator
  - Class Methods
  - Static Methods

Types of Methods

Instance Methods

- Acts upon the instance variables of that class
- Invoked by instance_name.method_name()

```
#To understanf the instance methods

class Student:
    #Constructor definition
    def __init__(self, n = "", m = 0):
        self.name = n
        self.marks = m

    #Instance method
    def putdata(self):
        print("Name: ", self.name)
        print("Marks: ", self.marks)
```

```
#Constructor called without any parameters
s = Student()
s.putdata()

#Constructor called with parameters
s = Student("Ram", 99)
s.putdata()
```

Types of Methods

Instance Methods: Accessor + Mutator

Accessor	Mutator
• Methods just reads the instance variables, will not modify it • Generally written in the form: getXXXX() • Also called getter methods	• Not only reads the data but also modifies it • Generally wriiten in the form: setXXXX() • Also called setter methods

```
#To understand accessor and mutator

#Create the class
class Student:

    #Define mutator
    def setName(self, name):
        self.name = name

    #Define accessor
    def getName(self):
        self.name
```

```
#Create an objects
s = Student()

#Set the name
s.setName("Ram")

#Print the name
print("Name: ", s.getName())
```

Types of Methods

Class Methods

- This methods acts on class level
- Acts on class variables only
- Written using @classmethod decorator
- First param is 'cls', followed by any params
- Accessed by classname.method()

```
#To understand the class methods

class Bird:
    #Define the class var here
    wings = 2

    #Define the class method
    @classmethod
    def fly(cls, name):
        print("{} flies with {} wings" . format(name, cls.wings))

#Call
Bird.fly("Sparrow")
Bird.fly("Pigeon")
```

Types of Methods

Static Methods

- Needed, when the processing is at the class level but we need not involve the class or instances
- Examples:
  - Setting the environmental variables
  - Counting the number of instances of the class
- Static methods are written using the decorator @staticmethod
- Static methods are called in the form classname.method()

```
#To Understand static method

class Sample:
    #Define class vars
    n = 0

    #Define the constructor
    def __init__(self):
        Sample.n = Sample.n + 1

    #Define the static method
    @staticmethod
    def putdata():
        print("No. of instances created: ", Sample.n)
```

```
#Create 3 objects
s1 = Sample()
s2 = Sample()
s3 = Sample()

#Class static method
Sample.putdata()
```

Passing **M**embers

Passing Members

- It is possible to pass the members(attributes / methods) of one class to another
- Example:

```
e = Emp()
```

- After creating the instance, pass this to another class 'Myclass'
- `Myclass.mymethod(e)`
  - mymethod is static

Passing Members

Example

```
#To understand how members of one class can be passed to another
```

```
#Define the class
class Emp:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def putdata(self):
        print("Name: ", self.name)
        print("Salary: ", self.salary)
```

```
#Define another class
class Myclass:
    @staticmethod
    def mymethod(e):
        e.salary += 1000
        e.putdata()
```

```
#Create Object
e = Emp("Ram", 20000)

#Call static method of Myclass and pass e
Myclass.mymethod(e)
```

Passing Members

Exercise

1. To calculate the power value of a number with the help of a static method

Inner Class

Inner Class

Introduction

- Creating class B inside Class A is called nested class or Inner class
- Example:

  Person's Data like,

  - Name: Single value
  - Age: Single Value
  - DoB: Multiple values, hence separate class is needed

Inner Class

Program: Version-1

```
#To understand inner class

class Person:
    def __init__(self):
        self.name = "Ram"
        self.db = self.Dob()

    def display(self):
        print("Name: ", self.name)

    #Define an inner class
    class Dob:
        def __init__(self):
            self.dd = 10
            self.mm = 2
            self.yy = 2002

        def display(self):
            print("DoB: {}/{}/{}" . format(self.dd,
self.mm, self.yy))
```

```
#Creating Object
p = Person()
p.display()

#Create inner class object
i = p.db
i.display()
```

Inner Class

Program: Version-2

```
#To understand inner class

class Person:
    def __init__(self):
        self.name = "Ram"
        self.db = self.Dob()

    def display(self):
        print("Name: ", self.name)

    #Define an inner class
    class Dob:
        def __init__(self):
            self.dd = 10
            self.mm = 2
            self.yy = 2002

        def display(self):
            print("DoB: {}/{}/{}" . format(self.dd,
self.mm, self.yy))
```

```
#Creating Object
p = Person()
p.display()

#Create inner class object
i = Person().Dob()
i.display()
```

THANK YOU

Inheritance **A**nd **P**olymorphism

Team **E**mertxe

Significance of **I**nheritance

Significance Of Inheritance

Example-1: teacher.py

# A Python program to create Teacher class and store it into teacher.py module.

```
# This is Teacher class. save this code in teaccher.py file
class Teacher:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self, name):
        self.name = name

    def getname(self):
        return self.name

    def setaddress(self, address):
        self.address = address

    def getaddress(self):
        return self.address

    def setsalary(self, salary):
        self.salary = salary

    def getsalary(self):
        return self.salary
```

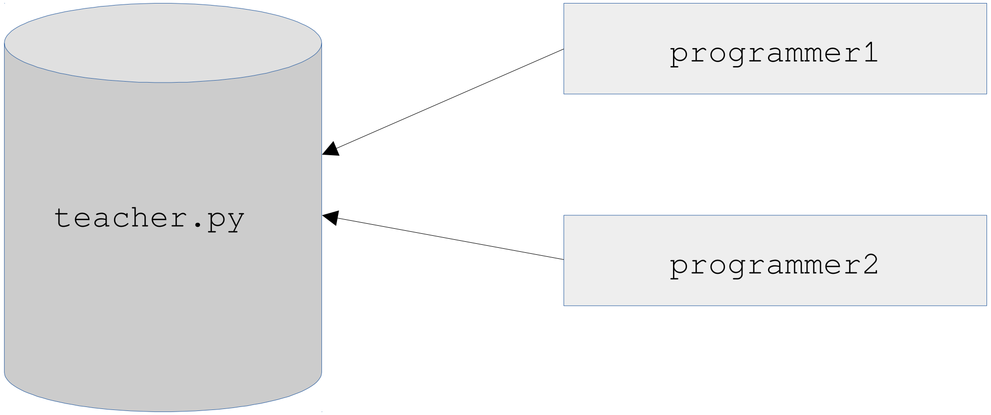
When the programmer wants to use this Teacher class that is available in teachers.py file, he can simply import this class into his program and use it

Significance Of Inheritance

Program

```
# using Teacher class from teacher important Teacher
from teacher import Teacher

# create instance
t = Teacher()

# store data into the instance
t.setid(10)
t.setname("Ram")
t.setaddress('HNO-10, Raj gardens, Delhi')
t.setsalary(25000.50)

# retrive data from instance and display
print('id= ', t.getid())
print('name= ', t.getname())
print('address= ', t.getaddress())
print('salary= ', t.getsalary())
```

Significance Of Inheritance



Significance Of Inheritance

```
Example-2: student.py

# A Python program to create sudent class and store it into student.py module
class Student:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self, name):
        self.name = name

    def getname(self):
        return self.name

    def setaddress(self, address):
        self.address = address

    def getaddress(self):
        return self.address

    def setmarks(self, marks):
        self.marks = marks

    def getmarks(self):
        return self.marks
```

Now, the second programmer who created this Student class and saved it as student.py can use it whenever he needs.

Significance Of Inheritance

Program

```
# using student class from student import student
from student import Student

# create instance
s = Student()

# store data into the instance
s.setid(100)
s.setname('Rakesh')
s.setaddress('HNO-22, Ameerpet, Hyderabad')
s.setmarks(970)

#Print the data
print("ID: ", s.getid())
print("Name: ", s.getname())
print("Address: ", s.getaddress())
print("Marks: ", s.getmarks())
```

Significance Of Inheritance

Comparision

```
class Teacher:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self, name):
        self.name = name

    def getname(self):
        return self.name

    def setaddress(self, address):
        self.address = address

    def getaddress(self):
        return self.address

    def setsalary(self, salary):
        self.salary = salary

    def getsalary(self):
        return self.salary
```

```
class Student:
    def setid(self, id):
        self.id = id

    def getid(self):
        return self.id

    def setname(self, name):
        self.name = name

    def getname(self):
        return self.name

    def setaddress(self, address):
        self.address = address

    def getaddress(self):
        return self.address

    def setmarks(self, marks):
        self.marks = marks

    def getmarks(self):
        return self.marks
```

By comparing both the codes, we can observe 75% of the code is common

Significance Of Inheritance

```
from teacher import Teacher

class Student(Teacher):
    def setmarks(self, marks):
        self.marks = marks

    def getmarks(self):
        return self.marks
```

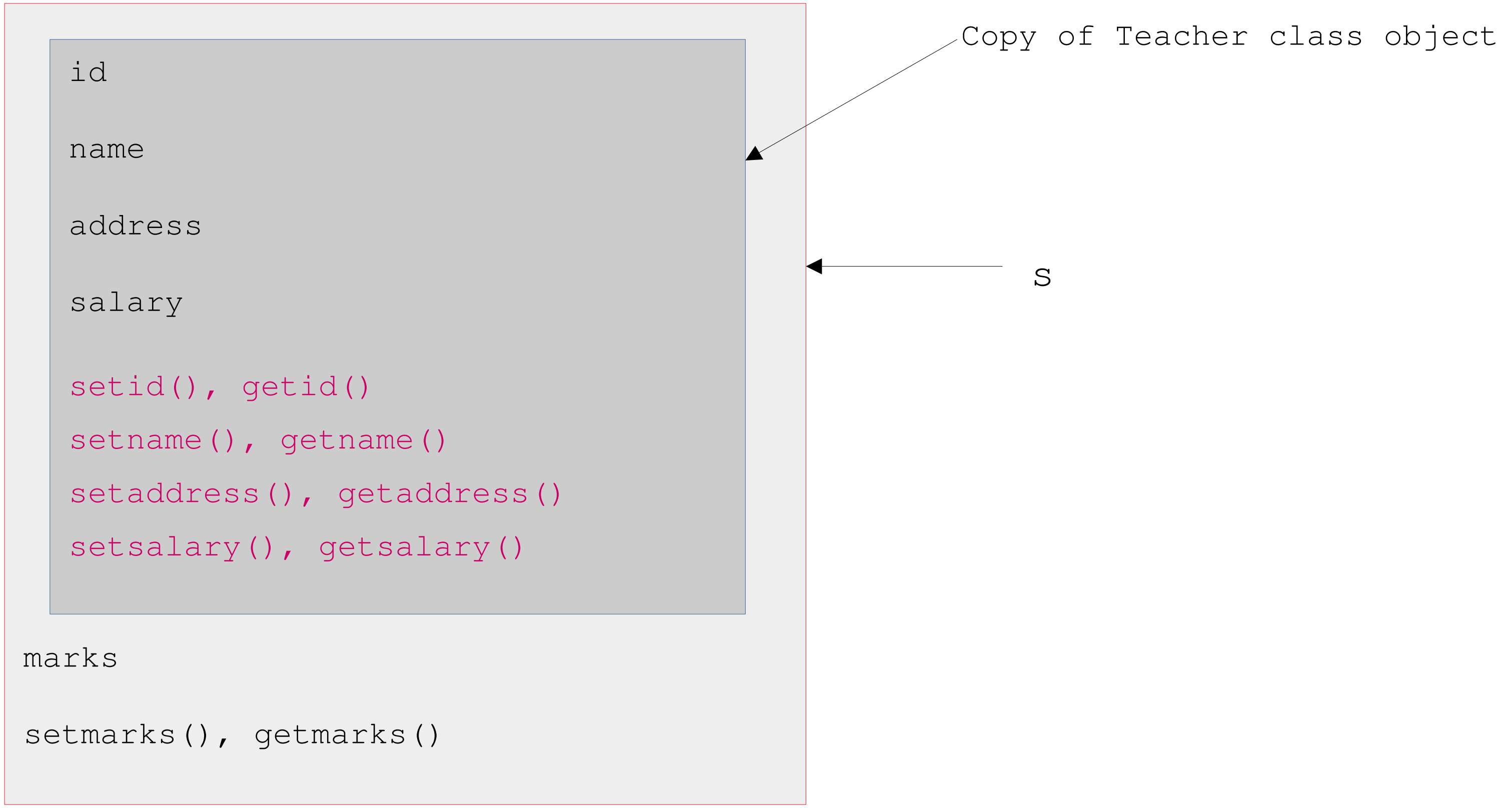
```
# create instance
s = Student()

# store data into the instance
s.setid(100)
s.setname('Rakesh')
s.setaddress('HNO-22, Ameerpet, Hyderabad')
s.setmarks(970)

#Print the data
print("ID: ", s.getid())
print("Name: ", s.getname())
print("Address: ", s.getaddress())
print("Marks: ", s.getmarks())
```

```
Syntax:
class Subclass(Baseclass):
```

Significance Of Inheritance

Advantages

- Smaller and easier to develop
- Productivity increases



id

name

address

salary

setid(), getid()

setname(), getname()

setaddress(), getaddress()

setsalary(), getsalary()

marks

setmarks(), getmarks()

Copy of Teacher class object

s

Student class Object

Inheritance

Definition

- Deriving the new classes from the existing classes such that the new classes inherit all the members of the existing classes is called Inheritance
- Syntax:

```
class Subclass(Baseclass):
```

Constructors in Inheritance

Constructors in Inheritance

Example

- Like variables & Methods, the constructors in the super class are also available in the sub-class

```
class Father:
        def __init__(self):
                self.property = 800000.00

        def display_property(self):
                print('Father\'s property= ',self.property)

class Son(Father):
        pass # we do not want to write anything in the sub class
```

```
#Create the instance
s = Son()
s.display_property()
```

Overriding Super Class Constructors and Methods

Overriding super class
Constructors + Methods

- Constructor Overriding
  - The sub-class *constructor* is replacing the super class constructor
- Method Overriding
  - The sub-class *method* is replacing the super class method

Example

```
# overriding the base class constructor and method in sub class
class Father:
        def __init__(self):
                self.property = 800000.00

        def display_property(self):
                print('Father\'s property= ', self.property)

class Son(Father):
        def __init__(self):
                self.property = 200000.00

        def display_property(self):
                print('child\'s property= ', self.property)

# create sub class instance and display father's property
s = Son()
s.display_property()
```

The **Super()** Method

The super() Method

- super() is a built-in method which is useful to call the super class constructor or Methods

Examples

```
#Call super class constructors
super().__init__()

#Call super class constructors and pass arguments
super().__init__(arguments)

#Call super class method
super().method()
```

The super() Method

Example

```
Example-1
# acceessing base class constructor in sub class

class Father:
        def __init__(self, property=0):
                self.property = property

        def display_property(self):
                print('Father\'s property= ', self.property)

class Son(Father):
        def __init__(self, property1=0, property=0):
                super().__init__(property)
                self.property1 = property1

        def display_property(self):
                print('Total property of child= ', self.property1 + self.property)

# create sub class instance and display father's property

s = Son(200000.00, 800000.00)
s.display_property()
```

The super() Method

Example

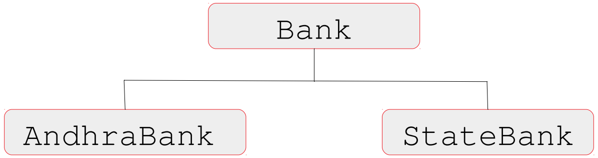
Example-2

```
# Accessing base class constructor and method in the sub class
class Square:
        def __init__(self, x):
                self.x = x

        def area(self):
                print('Area of square= ',self.x * self.x)

class Rectangle(Square):
        def __init__(self, x, y):
                super().__init__(x)
                self.y = y

        def area(self):
                super().area()
                print('Area of rectangle= ',self.x * self.y)

# find areas of square and rectangle
a, b = [float(x) for x in input("Enter two measurements: ").split()]
r = Rectangle(a,b)
r.area()
```

Types Of Inheritance

Types of Inheritance

Single



```
# A Python program showing single inhertiance in which two sub classes are derived from a
single base class.
```

```
# single inhertiance
class Bank(object):
        cash = 100000000

        @classmethod
        def available_cash(cls):
                print(cls.cash)
```

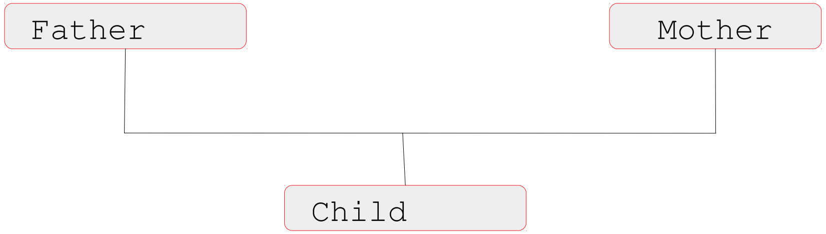
```
class StateBank(Bank):
        cash = 200000000

        @classmethod
        def available_cash(cls):
                print(cls.cash + Bank.cash)
```

```
class AndhraBank(Bank):
        pass
```

```
a = AndhraBank()
a.available_cash()

s = StateBank()
s.available_cash()
```

Types of Inheritance

Multiple



Syntax:

```
class Subclass(BaseClass1, BaseClass2, ...):
```

```
# A Python program to implement multiple inhertiance using two base classes
```

```
#multiple inheritance
class Father:
        def height(self):
                print('Height is 6.0 foot')
```

```
class Mother:
        def color(self):
                print('color is brown')
```

```
class child(Father, Mother):
        pass
```

```
c = child()
print('child\'s inherited qualities: ')
c.height()
c.color()
```

Multiple Inheritance

Problems in MI

```
# A Python program to prove that only one class constructor is available to sub class in multiple inheritance.
```

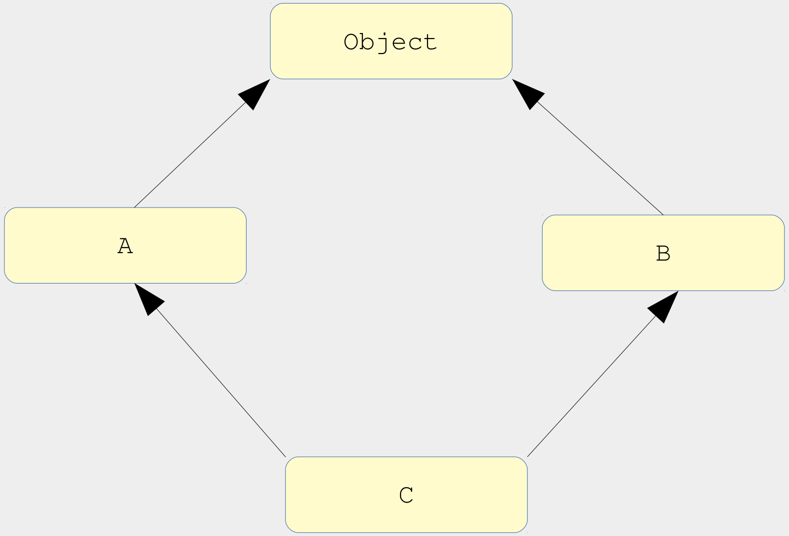
```
# when super classes have constructors

class A(object):
        def __init__(self):
                self.a = 'a'
                print(self.a)

class B(object):
        def __init__(self):
                self.b = 'b'
                print(self.b)

class C(A, B):
        def __init__(self):
                self.c = 'c'
                print(self.c)
                super().__init__()

# access the super class instance vars from C
o = C()    # o is object of class C
```

Multiple Inheritance

Solutions

```
#A Python program to access all the instance variables of both the base classes in multiple inheritance.
```

```
# when super classes have constructors - v2.0

class A(object):
        def __init__(self):
                self.a = 'a'
                print(self.a)
                super().__init__()

class B(object):
        def __init__(self):
                self.b = 'b'
                print(self.b)
                super().__init__()

class C(A,B):
        def __init__(self):
                self.c = 'c'
                print(self.c)
                super().__init__()

# access the super class instance vars from C

o = C()   # o is object class C
```



MRO(Method Resolution Operator)

MRO

- In Multiple Inheritance, any specified attribute or method is searched first in the current class. If not found, the search continues into parent classes in depth-first left to right fasion without searching for the same class twice

1. The first principle is to search for the sub classes before going for its base classes.

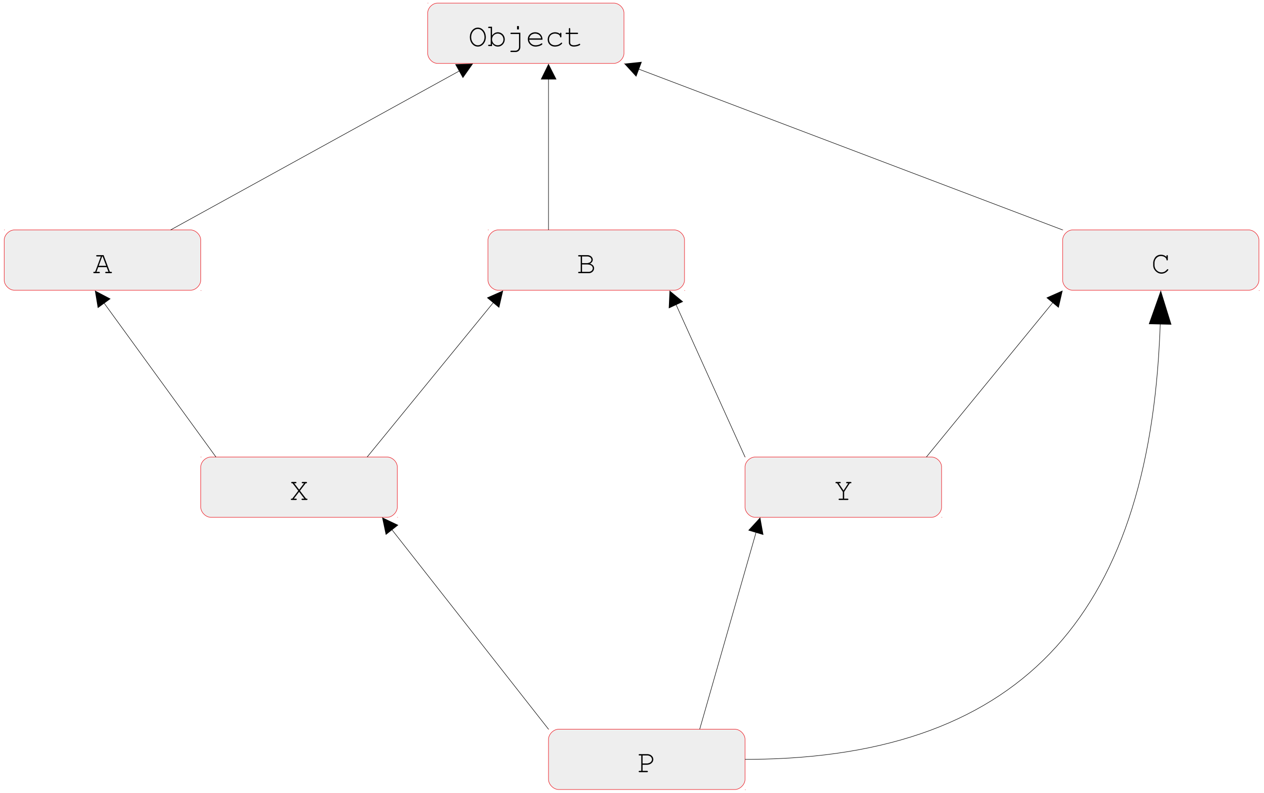
Thus if class B is inherited from A, it will search B first and then goes to A

2. The second principle is that when a class is inherited from several classes, it searches in the order from left to right in the base class.

Example: class C(A, B), then first it will search in A and then in B

3. The third principle is that it will not visit any class more than once. That means a class in the inheritance hierarchy is traversed only once exactly

MRO



MRO

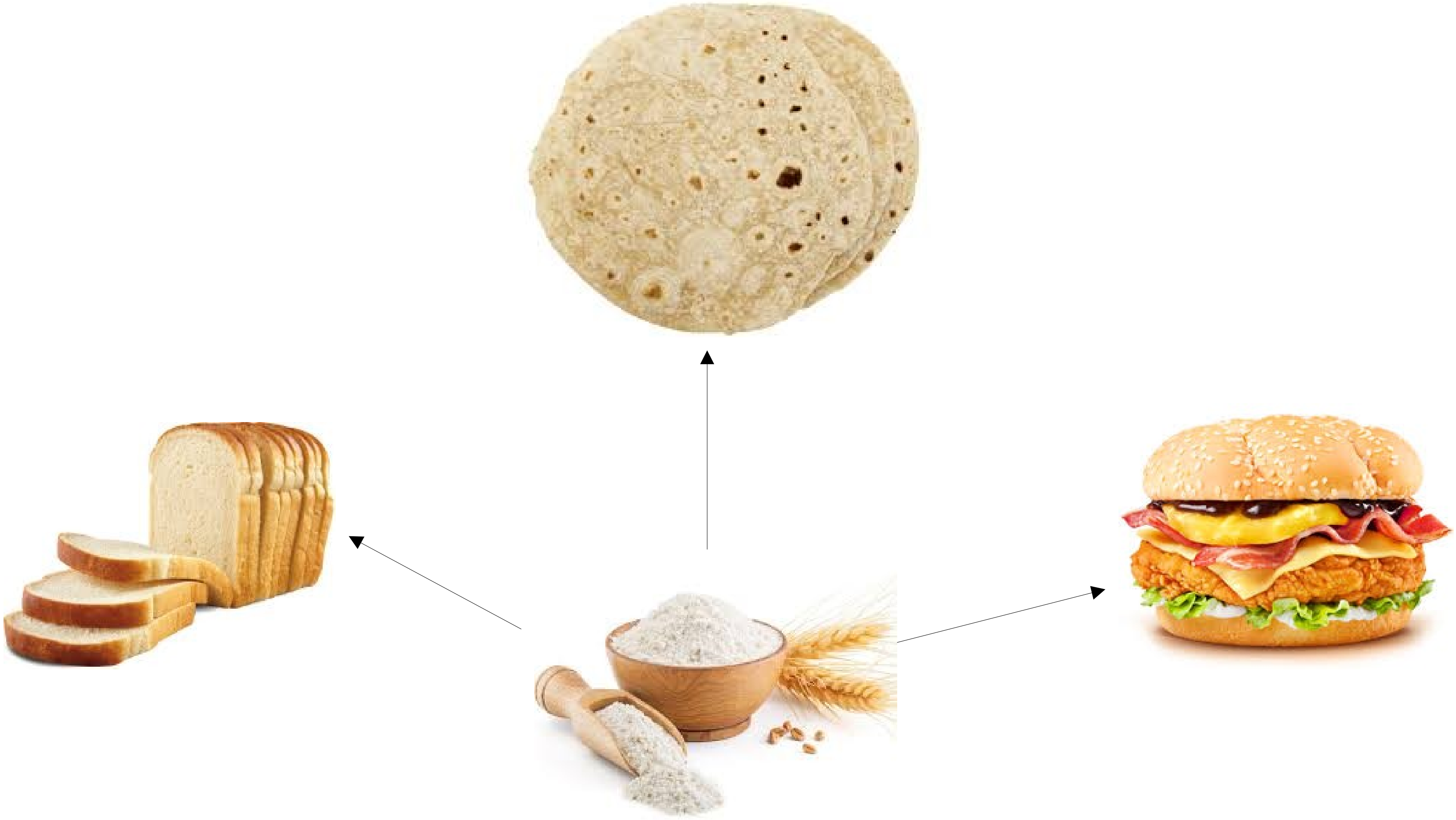
Program

```
# A Python program to understand the order of execution of methods in several base classes
according to MRO.

class A(object):
        def method(self):
                print('A class method')
                super().method()

class B(object):
        def method(self):
                print('B class method')
                super().method()
class C(object):
        def method(self):
                print('C class method')

class X(A, B):
        def method(self):
                print('X class method')
                super().method()
class Y(A, B):
        def method(self):
                print('Y class method')
                super().method()

class P(X,Y,C):
        def method(self):
                print('P class method')
                super().method()

P = P()
P.method()
```

P.mro(): Returns sequence of execution of classes

Polymorphism

Polymorphism



Polymorphism

Introduction

- Variable, Object or Method exhibits different behavior in different contexts called
- Polymorphism
- Python has built-in Polymorphism

Polymorphism

Duck Typing Philosophy

- Datatype of the variables is not explicitly declared
- type(): To check the type of variable or object

Example-1	x = 5
	print(type(x))
		<class ‘int’>

Example-2	x = “Hello”
	print(type(x))
		<class ‘str’>

Conclusion

1. Python’s type system is strong because every variable or object has a type that we can check with the type() function

2. Python’s type system is ‘dynamic’ since the type of a variable is not explicitly declared, but it changes with the content being stored

ΣMERTXE

Polymorphism

Duck Typing Philosophy: Program

```
# A Python program to invoke a method on an object without knowing the type (or class) of
the object.
```

```
# duck typing example

# Duck class contains talk() method
class Duck:
        def talk(self):
                print('Quack, quack!')

#Human class contains talk() method
class Human:
        def talk(self):
                print('Hello, hi!')

# this method accepts an object and calls talk() method
def call_talk(obj):
        obj.talk()

# call call_talk() method pass an object
# depending on type of object, talk() method is executed
x = Duck()

call_talk(x)
x = Human()
call_talk(x)
```

During runtime, if it is found that method does not belong to that object, there will be an error called ‘AttributeError’

Polymorphism

Attribute Error: Overcoming

```
# this method accepts an object and calls talk() method
def call_talk(obj):
        if hasattr(obj, 'talk'):
                obj.talk()
        elif hasattr(obj, 'bark'):
                obj.bark()
        else:
                print('Wrong object passed...')
```

During runtime, if it is found that method does not belong to that object, there will be an error called ‘AttributeError’

Operator **O**verloading

Operator Overloading

Example-1

```
# A Python program to use addition operator to act on different types of objects.
# overloading the + operator
# using + on integers to add them
print(10+15)

#using + on strings to concatenate them
s1 = "Red"
s2 = "Fort"
print(s1+s2)

#using + on lists to make a single list
a = [10, 20, 30]
b = [5, 15, -10]
print(a+b)
```

‘+’ operator is overloaded and thus exhibits polymorphism

Operator Overloading

Example-2

```
# Error
# using + operator on objects

class BookX:
        def __init__(self, pages):
                self.pages = pages

class BookY:
        def __init__(self, pages):
                self.pages = pages

b1 = BookX(100)
b2 = BookY(150)
print('Total pages = ', b1 + b2)
```

```
#Correction
# overloading + operator to act on objects

class BookX:
        def __init__(self, pages):
                self.pages = pages

        def __add__(self, other):
                return self.pages+other.pages

class BookY:
        def __init__(self, pages):
                self.pages = pages

b1 = BookX(100)
b2 = BookY(150)
print('Total pages= ', b1+b2)
```

```
def __add__(self, other):
```

Operator Overloading

Example-3

```python
#A Python program to overload greater than (>) operator to make it act on class objects.
# overloading > operator
class Ramayan:
    def __init__(self, pages):
        self.pages = pages

    def __gt__(self, other):
        return self.pages > other.pages

class Mahabharat:
    def __init__(self, pages):
        self.pages = pages

b1 = Ramayan(1000)
b2 = Mahabharat(1500)

if(b1 > b2):
    print('Ramayan has more pages')
else:
    print('Mahabharat has more pages')
```

```python
def __gt__(self, other):
```

Method **O**verloading

Operator Overloading

Example-1

```
# A Python program to show method overloading to find sum of two or three numbers.
# method overloading
class Myclass:
    def sum(self, a=None, b=None, c=None):
        if a!=None and b!=None and c!=None:
            print('Sum of three= ', a + b + c)
        elif a!=None and b!=None:
            print('Sum of two= ', a + b)
        else:
            print('Please enter two or three arguments')

# call sum() using object
m = Myclass()
m.sum(10, 15, 20)
m.sum(10.5, 25.55)
m.sum(100)
```

If a method is written such that it can perform more than one task, it is called method overloading

Method **O**verriding

Operator Overriding

Example-1

```
# A Python program to override the super class method in sub class.
# method overriding
import math
class Square:
    def area(self, x):
        print('Square area= %.4f' % (x * x))

class Circle(Square):
    def area(self, x):
        print('Circle area= %.4f' % (math.pi *x * x))

# call area() using sub class object
c = Circle()
c.area(15)
```

If a method written in sub class overrides the same method in super class, then it is called method overriding

Method overriding already discussed in Constructor & Method Overridings

THANK YOU

Abstract **C**lasses **A**nd **I**nterfaces

Team **E**mertxe

Introduction

Introduction

Example:

To understand that Myclass method is shared by all objects

```
class Myclass:
    def calculate(self, x):
        print("Square: ", x * x)
```

```
#All objects share same calculate() method
obj1 = Myclass()
obj1.calculate(2)


obj2 = Myclass()
obj2.calculate(3)


obj3 = Myclass()
obj3.calculate(4)
```

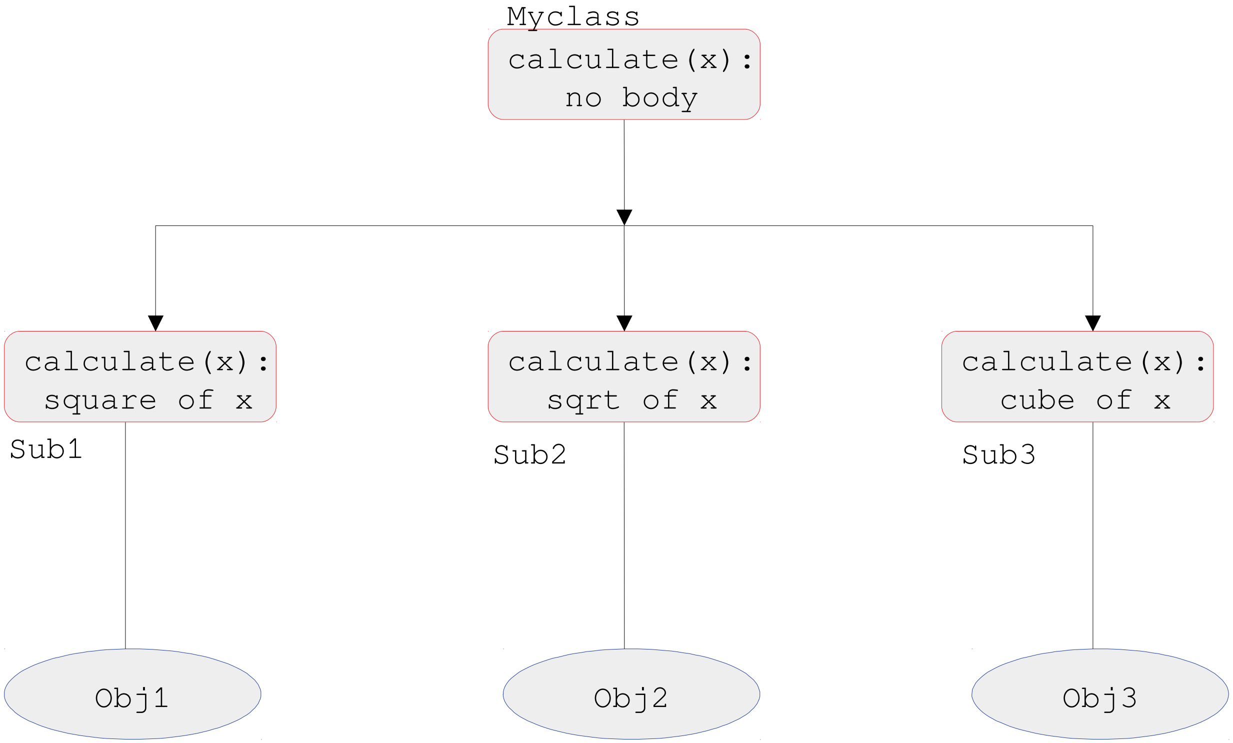
Question

- What If?
    - Object-1 wants to calculate square value
    - Object-2 wants to calculate square root
    - Object-3 wants to calculate Cube

Solution-1

- Define, three methods in the same class
  - calculate_square()
  - calculate_sqrt()
  - calculate_cube()
- Disadvantage:
  - All three methods are available to all the objects which is not advisable

Solution-2


Myclass
calculate(x):
no body
calculate(x):
square of x
Sub1
calculate(x):
sqrt of x
Sub2
calculate(x):
cube of x
Sub3
Obj1
Obj2
Obj3

Abstract Method and Class

Abstract Method & Class

- Abstract Method
  - – Is the method whose action is redefined in sub classes as per the requirements
- of the objects
  - – Use decorator @abstractmethod to mark it as abstract method
  - – Are written without body

- Abstract Class
  - – Is a class generally contains some abstract methods
  - – PVM cannot create objects to abstract class, since memory needed will not be
  - known in advance
  - – Since all abstract classes should be derived from the meta class ABC which belongs to abc(abstract base class) module, we need to import this module

  - – To import abstract class, use
  - – from abc import ABC, abstractmethod
  - OR
  - – from abc import *

Program-1

```
#To create abstract class and sub classes which implement the abstract method of the
abstract class
```

```
from abc import ABC, abstractmethod

class Myclass(ABC):
    @abstractmethod
    def calculate(self, x):
        pass
```

```
#Sub class-1
class Sub1(Myclass):
    def calculate(self, x):
        print("Square: ", x * x)
```

```
#Sub class-2
import math
class Sub2(Myclass):
    def calculate(self, x):
        print("Square root: ", math.sqrt(x))
```

```
#Sub class-3
class Sub3(Myclass):
    def calculate(self, x):
        print("Cube: ", x * x * x)
```

```
Obj1 = Sub1()
Obj1.calculate(2)

Obj2 = Sub2()
Obj2.calculate(16)

Obj3 = Sub3()
Obj2.calculate(3)
```

Example-2

- Maruthi, Santro, Benz are all objects of class Car

Registration no.	– All cars will have reg. no. – Create var for it
Fuel Tank	– All cars will have common fule tank – Action: Open, Fill, Close
Steering	– All cars will not have common steering say, Maruthi uses– Manual steering Santro uses – Power steering – So define this as an Abstract Method
Brakes	– Maruthi uses hydraulic brakes – Santro uses gas brakes – So define this as an Abstract Method

Program-2

```
#Define an absract class

from abc import *

class Car(ABC):
    def __init__(self, reg_no):
        self.reg_no = reg_no

    def opentank(self):
        print("Fill the fuel for car with reg_no: ", self.reg_no)

    @abstractmethod
    def steering(self):
        pass

    @abstractmethod
    def braking(self):
        pass
```

```
#Define the Maruthi class

from abstract import Car

class Maruthi(Car):
    def steering(self):
        print("Maruthi uses Manual steering")

    def braking(self):
        print("Maruthi uses hydraulic braking system")

#Create the objects
Obj = Maruthi(123)
Obj.opentank()
Obj.steering()
Obj.braking()
```

Interfaces

Interfaces

- Abstract classes contains both,
  - – Abstract methods
  - – Concrete Methods

- Interfaces is also an Abstract class, but contains only
  - – Abstract methods
- Plus point of Interface.
  - – Every sub-class may provide its own implementation for the abstract methods

Interfaces

Program-1

```
from abc import *
```

```
class Myclass(ABC):
    @abstractmethod
    def connect(self):
        pass

    @abstractmethod
    def disconnect(self):
        pass
```

```
#Sub-Class:1
class Oracle(Myclass):
    def connect(self):
        print("Connecting to oracle database...")

    def disconnect(self):
                   print("Disconnecting from oracle
database...")
```

```
#Sub-Class:2
class Sybase(Myclass):
    def connect(self):
        print("Connecting to sybase database...")

    def disconnect(self):
                   print("Disconnecting from sybase
database...")
```

```
#Define Database
class Database:

    str = input("Enter the database name: ")

    #Covert the string into the class name
    classname = globals()[str]

    #create an object
    x = classname()

    #Call methods
    x.connect()
    x.disconnect()
```

Interfaces

Program-2

```
from abc import *
```

```
class Myclass(ABC):
    @abstractmethod
    def putdata(self, text):
        pass

    @abstractmethod
    def disconnect(self):
        pass
```

```
#Sub-Class:1
class IBM(Myclass):
    def putdata(self, text):
        print(text)

    def disconnect(self):
        print("Disconnecting from IBM printer...")
```

```
#Sub-Class:2
class Epson(Myclass):
    def putdata(self, text):
        print(text)

    def disconnect(self):
        print("Disconnecting from Epson printer...")
```

```
#Define Printer
class Printer:

    str = input("Enter the printer name: ")

    #Covert the string into the class name
    classname = globals()[str]

    #create an object
    x = classname()

    #Call methods
    x.putdata("Sending to printer")
    x.disconnect()
```

THANK YOU

Exceptions

Team **E**mertxe

Introduction

Errors

- Categories of Errors
  - Compile-time
  - Runtime
  - Logical

Errors

Compile-Time

What?	These are syntactical errors found in the code, due to which program fails to compile
Example	Missing a colon in the statements llike if, while, for, def etc

Program

Output

```
x  = 1

if x == 1
    print("Colon missing")
```

```
py 1.0_compile_time_error.py
  File "1.0_compile_time_error.py", line 5
    if x == 1
            ^
SyntaxError: invalid syntax
```

```
x  = 1

#Indentation Error
if x == 1:
    print("Hai")
        print("Hello")
```

```
py 1.1_compile_time_error.py
  File "1.1_compile_time_error.py", line 8
    print("Hello")
    ^
IndentationError: unexpected indent
```

Errors

Runtime – 1

What?	When PVM cannot execute the byte code, it flags runtime error
Example	Insufficient memory to store something or inability of the PVM to execute some statement come under runtime errors

Program

```
def combine(a, b):
    print(a + b)

#Call the combine function
combine("Hai", 25)
```

Output

```
py 2.0_runtime_errors.py
Traceback (most recent call last):
  File "2.0_runtime_errors.py", line 7, in <module>
    combine("Hai", 25)
  File "2.0_runtime_errors.py", line 4, in combine
    print(a + b)
TypeError: can only concatenate str (not "int") to str
```

```
"""
Conclusion:

1. Compiler will not check the datatypes.
2.Type checking is done by PVM during run-time.
"""
```

Errors

Runtime - 2

What?	When PVM cannot execute the byte code, it flags runtime error
Example	Insufficient memory to store something or inability of the PVM to execute some statement come under runtime errors

Program

```
#Accessing the item beyond the array bounds

lst = ["A", "B", "C"]
print(lst[3])
```

Output

```
py 2.1_runtime_errors.py
Traceback (most recent call last):
  File "2.1_runtime_errors.py", line 5, in <module>
    print(lst[3])
IndexError: list index out of range
```

Errors

Logical-1

What?	These errors depicts flaws in the logic of the program
Example	Usage of wrong formulas

Program

```
def increment(sal):
    sal = sal * 15 / 100
    return sal

#Call the increment()
sal = increment(5000.00)
print("New Salary: %.2f" % sal)
```

Output

```
py 3.0_logical_errors.py
New Salary: 750.00
```

Errors

Logical-2

What?	These errors depicts flaws in the logic of the program
Example	Usage of wrong formulas

Program

```
#1. Open the file
f = open("myfile", "w")

#Accept a, b, store the result of a/b into the file
a, b = [int(x) for x in input("Enter two number: ").split()]
c = a / b

#Write the result into the file
f.write("Writing %d into myfile" % c)

#Close the file
f.close()
print("File closed")
```

Output

```
py 4_effect_of_exception.py

Enter two number: 10 0

Traceback (most recent call last):

  File "4_effect_of_exception.py", line 8, in <module>

    c = a / b

ZeroDivisionError: division by zero
```

Errors

Common

- When there is an error in a program, due to its sudden termination, the following things can be suspected
  - The important data in the files or databases used in the program may be lost
  - The software may be corrupted
  - The program abruptly terminates giving error message to the user making the user losing trust in the software

Exceptions

Introduction

- An exception is a runtime error which can be handled by the programmer
- The programmer can guess an error and he can do something to eliminate the harm caused by that error called an ‘Exception’

<table>
<tr><td colspan="2">BaseException</td></tr>
<tr><td colspan="2">Exception</td></tr>
<tr><td>StandardError</td><td>Warning</td></tr>
<tr><td>ArthmeticError
AssertionError
SyntaxError
TypeError
EOFError
RuntimeError
ImportError
NameError</td><td>DeprecationWarning
RuntimeWarning
ImportantWarning</td></tr>
</table>

Exceptions

Exception Handling

- The purpose of handling errors is to make program robust

Step-1	try:
	    statements
		#To handle the ZeroDivisionError Exception
		try:
		    f = open("myfile", "w")
		    a, b = [int(x) for x in input("Enter two numbers: ").split()]
		    c = a / b
		    f.write("Writing %d into myfile" % c)

Step-2	except exeptionname: statements	except ZeroDivisionError: print("Divide by Zero Error") print("Don't enter zero as input")

Step-3	finally:
	    statements
		finally:
		    f.close()
		    print("Myfile closed")

Exceptions

Program

```
#To handle the ZeroDivisionError Exception
```

```
#An Exception handling Example
try:
    f = open("myfile", "w")
    a, b = [int(x) for x in input("Enter two numbers: ").split()]
    c = a / b
    f.write("Writing %d into myfile" % c)

except ZeroDivisionError:
    print("Divide by Zero Error")
    print("Don't enter zero as input")

finally:
    f.close()
    print("Myfile closed")
```

Output:

```
py 5_exception_handling.py
Enter two numbers: 10 0
Divide by Zero Error
Don't enter zero as input
Myfile closed
```

Exceptions

Exception Handling Syntax

```
try:
    statements

except Exception1:
    handler1

except Exception2:
    handler2

else:
    statements

finally:
    statements
```

Exceptions

Exception Handling: Noteworthy

- A single try block can contain several except blocks.
- Multiple except blocks can be used to handle multiple exceptions.
- We cannot have except block without the try block.
- We can write try block without any except block.
- Else and finally are not compulsory.
- When there is no exception, else block is executed after the try block.
- Finally block is always executed.

Exceptions

Types: Program-1

```
#To handle the syntax error given by eval() function
```

```
#Example for Synatx error
try:
     date = eval(input("Enter the date: "))

except SyntaxError:
     print("Invalid Date")

else:
     print("You entered: ", date)
```

Output:

Run-1:

```
Enter the date: 5, 12, 2018
You entered:  (5, 12, 2018)
```

Run-2:

```
Enter the date: 5d, 12m, 2018y
Invalid Date
```

Exceptions

Types: Program-2

```
#To handle the IOError by open() function

#Example for IOError

try:
    name = input("Enter the filename: ")
    f = open(name, "r")

except IOError:
    print("File not found: ", name)

else:
    n = len(f.readlines())
    print(name, "has", n, "Lines")
    f.close()
```

If the entered file is not exists, it will raise an IOError

Exceptions

Types: Program-3

```
#Example for two exceptions

#A function to find the total and average of list elements
def avg(list):
      tot = 0
      for x in list:
            tot += x
      avg = tot / len(list)
      return tot.avg

#Call avg() and pass the list
try:
      t, a = avg([1, 2, 3, 4, 5, 'a'])
      #t, a = avg([]) #Will give ZeroDivisionError
      print("Total = {}, Average = {}". format(t, a))

except TypeError:
      print("Type Error: Pls provide the numbers")

except ZeroDivisionError:
      print("ZeroDivisionError, Pls do not give empty list")
```

```
Output:

Run-1:

Type Error: Pls provide the numbers

Run-2:

ZeroDivisionError, Pls do not give empty list
```

Exceptions

Except Block: Various formats

Format-1	except Exceptionclass:
Format-2	except Exceptionclass as obj:
Format-3	except (Exceptionclass1, Exceptionclass2, ...):
Format-4	except:

Exceptions

Types: Program-3A

```
#Example for two exceptions

#A function to find the total and average of list elements
def avg(list):
      tot = 0
      for x in list:
            tot += x
      avg = tot / len(list)
      return tot.avg

#Call avg() and pass the list
try:
      t, a = avg([1, 2, 3, 4, 5, 'a'])
      #t, a = avg([]) #Will give ZeroDivisionError
      print("Total = {}, Average = {}". format(t, a))

except (TypeError, ZeroDivisionError):
      print("Type Error / ZeroDivisionError”)
```

```
Output:

Run-1:

Type Error / ZeroDivisionError

Run-2:

Type Error / ZeroDivisionError
```

Exceptions

The assert Statement

- It is useful to ensure that a given condition is True, It is not True, it raises AssertionError.
- Syntax:

```
assert condition, message
```

Exceptions

The assert Statement: Programs

Program – 1

```
#Handling AssertionError
try:
    x = int(input("Enter the number between 5 and 10: "))
    assert x >= 5 and x <= 10
    print("The number entered: ", x)

except AssertionError:
    print("The condition is not fulfilled")
```

Program – 2

```
#Handling AssertionError
try:
    x = int(input("Enter the number between 5 and 10: "))
    assert x >= 5 and x <= 10, "Your input is INVALID"
    print("The number entered: ", x)

except AssertionError as Obj:
    print(Obj)
```

Exceptions

User-Defined Exceptions

Step-1

```
class MyException(Exception):
        def __init__(self, arg):
            self.msg = arg
```

Step-2

```
raise MyException("Message")
```

Step-3

```
try:
    #code
    except MyException as me:
        print(me)
```

Exceptions

User-Defined Exceptions: Program

```
#To create our own exceptions and raise it when needed
class MyException(Exception):
    def __init__(self, arg):
        self.msg = arg

def check(dict):
    for k, v in dict.items():
        print("Name = {:15s} Balance = {:10.2f}" . format(k, v)) if (v < 2000.00):
            raise MyException("Less Bal Amount" + k)

bank = {"Raj": 5000.00, "Vani": 8900.50, "Ajay": 1990.00}

try:
    check(bank)
except MyException as me:
    print(me)
```

THANK YOU

Files

Team Emertxe

Introduction

Introduction

- A file is an object on a computer that stores data, information, settings, or commands used with a computer program
- Advantages of files
  - – Data is stored permanently
  - – Updation becomes easy
  - – Data can be shared among various programs
  - – Huge amount of data can be stored

Files

Types

Text	Binary
Stores the data in the form of strings	Stores data in the form of bytes
Example: “Ram” is stored as 3 characters 890.45 is stored as 6 characters	Example: “Ram” is stored as 3 bytes 89000.45 is stored as 8 bytes
Examples: .txt, .c, .cpp	Examples: .jpg, .gif or .png

Files

Opening a file

Name	open()
Syntax	file_handler = open("file_name", "open_mode", "buffering") filename : Name of the file to be opened open_mode: Purpose of opening the file buffering: Used to stored the data temporarily
Opening Modes	
w	- To write the data - If file already exist, the data will be lost
r	- To read the data - The file pointer is positioned at the begining of the file
a	- To append data to the file - The file pointer is placed at the end of the file
w+	- To write and read data - The previous data will be deleted
r+	- To read and write - The previous data will not be deleted - The file pointer is placed at the begining of the file
a+	- To append and read data - The file pointer will be at the end of the file
x	- To open the file in exclusive creation mode - The file creation fails, if already file exist
Example	
f = open("myfile.txt", "w") Here, buffer is optional, if omitted 4096 / 8192 bytes will be considered.	

Files

Closing a file

Name	close()
Syntax	f.close()

Example	#Open the file
	f = open("myfile.txt", "w")
	
	#Read the string
	str = input("Enter the string: ")
	
	#Write the string into the file
	f.write(str)
	
	#Close the file
	f.close()

Files

Working with text files containing strings

```
To read the content from files,
     f.read()                : Reads all lines, displays line by line
     f.readlines()           : Displays all strings as elements in a list
     f.read().splitlines(): To suppress the "\n" in the list
```

Program

```
#To create a text file to store strings

#Open the file
f = open("myfile.txt", "r")

#Read the data from a file
str = f.read()  #Reads all data

#Display the data
print(str)

#Close the file
f.close()
```

Note:

```
"""
f.read(n): Will read 'n' bytes from the file
"""
```

Files

Working with text files containing strings

```
f.seek(offset, fromwhere)
      - offset          : No. of bytes to move
      - fromwhere       : Begining, Current, End
      - Example         : f.seek(10, 0), move file handler from Beg forward 10 bytes.

# Appending and then reading strings, Open the file for reading data
f = open('myfile.txt', 'a+')

print('Enter text to append(@ at end): ')
while str != '@':
    str = input()   # accept string into str

    # Write the string into file
    if (str != '@'):
        f.write(str+"\n")

# Put the file pointer to the beginning of the file
f.seek(0,0)

# Read strings from the file
print('The file cotents are: ')
str = f.read()
print(str)

# Closing the file
f.close()
```

Files

Knowing If file exists or not

```
Sample:
      if os.path.isfile(fname):
            f = open(fname, "r")
      else:
            print(fname + "Does not exist")
            sys.exit() #Terminate the program
```

```
# Checking if file exists and then reading data
import os, sys

# open the file for reading data
fname = input('Enter filename : ')

if os.path.isfile(fname):
    f = open(fname, 'r')
else:
    print(fname+' does not exist')
    sys.exit()

# Read strings from the file
print('The file contents are: ')
str = f.read()
print(str)

# Closing the file
f.close()
```

Files

Exercise

Problem- 1

To count number of lines, words and characters in a text file

Problem- 2

To copy an image from one file to another

Files

The with statement

1. Can be used while opening the file

2. It will take care of closing the file, without using close() explicitly

3. Syntax: with open("file_name", "openmode") as fileObj:

Program -1

```
# With statement to open a file
with open('sample.txt', 'w') as f:
    f.write('I am a learner\n')
    f.write('Python is attactive\n')
```

Program -2

```
# Using with statement to open a file
with open('sample.txt', 'r') as f:
    for line in f:
    print(line)
```

Files

The pickle + Unpickle

1. To store the data of different types, we need to create the class for it.

2. Pickle/Serialization:

   - Storing Object into a binary file in the form of bytes.
   - Done by a method dump() of pickle module
   - pickle.dump(object, file)

3. Unpickle/Deserialization

   - Process where byte stream is converted back into the object.
   - Object = pickle.load(file)

Files

The pickle: Program

```
# A python program to create an Emp class witg employee details as instance variables.
```

```
# Emp class - save this as Emp.py
class Emp:
    def_init_(self, id, name, sal):
    self.id = id
    self.name = name
    self.sal = sal

    def display(self):
        print("{:5d} {:20s} {:10.2f}".format(self.id, self.name,self.sal))
```

```
# pickle - store Emp class object into emp.dat file
import Emp, pickle

# Open emp.dat file as a binary file for writing
f = open('emp.dat', 'wb')
n = int(input('How many employees? '))

for i in range(n):
    id = int(input('Enter id: '))
    name = input('Enter name: ')
    sal = float(input('Enter salary: '))

for i in range(n):
    id = int(input('Enter id: '))
    name = input('Enter name: ')
    sal = float(input('Enter salary: '))

    # Create Emp class object
    e = Emp.Emp(id, name, sal)

    # Store the object e into the file f
    pickle.dump(e, f)

#close the file
f.close()
```

Files

The unpickle: Program

```
# A python program to create an Emp class witg employee details as instance variables.
```

```
# Emp class - save this as Emp.py
class Emp:
    def_init_(self, id, name, sal):
    self.id = id
    self.name = name
    self.sal = sal

    def display(self):
        print("{:5d} {:20s} {:10.2f}".format(self.id, self.name,self.sal))
```

```
# unpickle or object de-serialization
import Emp, pickle

# Open the file to read objects
f = open('emp.dat', 'rb')

print('Employees details: ')
while True:
    try:
        #Read object from file f
        obj = pickle.load(f)
        # Display the contents of employee obj
        obj.display()

    except EOFError:
        print('End of file reached....')
        break

#Close the file
f.close()
```

Random Binary File Access

using mmap

1. Using mmap, binary data can be viewed as strings

```
mm = mmap.mmap(f.fileno(), 0)
```

2. Reading the data using read() and readline()

```
print(mm.read())
print(mm.readline())
```

3. We can also retrieve the data using teh slicing operator

```
print(mm[5: ])
print(mm[5: 10])
```

4. To modify / replace the data

```
mm[5: 10] = str
```

5. To find the first occurrance of the string in the file

```
n = mm.find(name)
```

6. To convert name from string to binary string

```
name = name.encode()
```

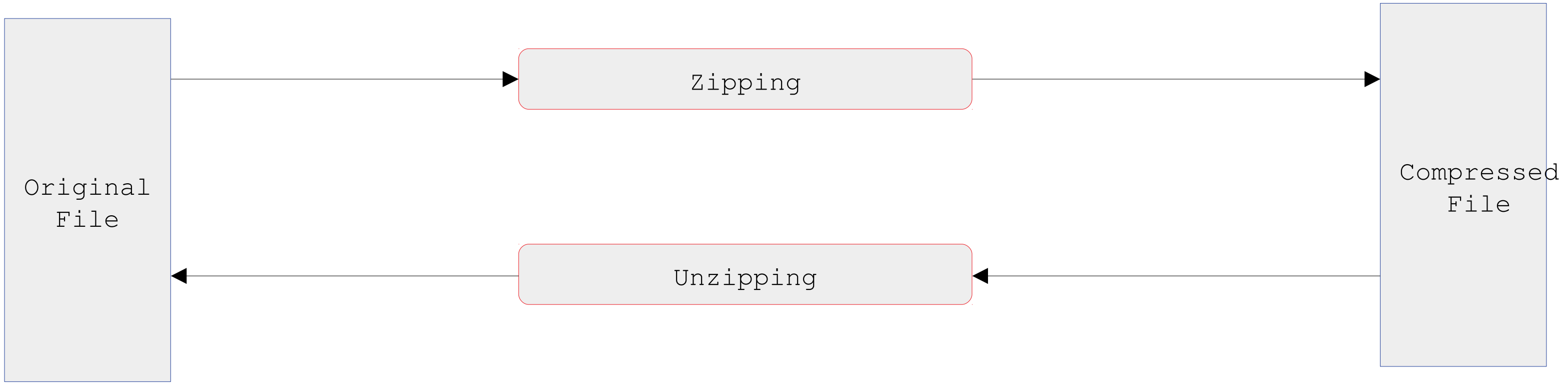
7. To convert bytes into a string

```
ph = ph.decode()
```

Demonstrate the code

Zip & Unzip

- Zip:
  - – The file contents are compressed and hence the size will be reduced
  - – The format of data will be changed making it unreadable



Zip & Unzip

Programs

```
# Zipping the contents of files
from zipfile import *

# create zip file
f = zipfile('test.zip', 'w', 'ZIP_DEFLATED')

# add some files. these are zipped
f.write('file1.txt')
f.write('file2.txt')
f.write('file3.txt')

# close the zip file
print('test.zip file created....')
f.close()
```

```
# A Python program to unzip the contents of the files
# that are available in a zip file.

# To view contents of zipped files
from zipfile import*

# open the zip file
z = Zipfile('test.zip', 'r')

# Extract all the file names which are int he zip file
z.extractall()
```

Working With Directories

Program-1

```
# A Python program to know the currently working directory.
```

```
import os

# get current working directory
current = os.getcwd()

print('Current sirectory= ', current)
```

Working With Directories

Program-2

```
# A Python program to create a sub directory and then sub-sun directory in the current
directory.
```

```
import os
# create a sub directory by the name mysub
os.mkdir('mysub')

# create a sub-sub directory by the same mysub2
os.mkdir('mysub/mysub2')
```

Working With Directories

Program-3

```
# A Python program to use the makedirs() function to create sub and sub-sub directories.
```

```
import os

# create sub and sub-sub directories
os.mkdirs('newsub/newsub2')
```

Working With Directories

Program-4

```
# A Python program to remove a sub directory that is inside another directory.
```

```
import os
# to remove newsub2 directory
os.rmdir('newsub/newsub2')
```

Working With Directories

Program-5

```
# A Python program to remove a group of directories in the path
```

```
import os
# to remove mysub3, mysub2 and then mysub.
os.removedirs('mysub/mysub2/mysub3')
```

Working With Directories

Program-6

```
# A Python program to rename a directory.
```

```
import os
# to rename enum as newenum
os.rename('enum', 'newenum')
```

Working With Directories

Program-7

```
# A Python program to display all contents of the current directory.
```

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

Running other programs

Program-7

The OS module has the system() method that is useful to run an executableprogram from our Python program		
Example-1	os.system(‘dir’)	Display contents of current working DIR
Example-2	os.system(‘python demo.py’)	Runs the demo.py code

THANK YOU

Regular Expressions

Team Emertxe

Regular **E**xpressions

Introduction

- RE is a string that contains special symbols and characters to find and extract the information
- Operations:
  - ✓ Search
  - ✓ Match
  - ✓ Find
  - ✓ Split
- Also called as *regex*
- Module: re
  - This module contains the methods like
    - compile()
    - search()
    - match()
    - findall()
    - split()...
  - import re

Regular Expressions

Steps

- Step-1: Compile the RE

```
prog = re.compile(r’m\w\w’)
```

- Step-2: Search the strings

```
str = “cat mat bat rat”

result = prog.search(str)
```

- Step-3: Display the result

```

print(result.group())
```

Regular Expressions

Example-1: search()

```
import re
str = 'man sun mop run'
result = re.search(r'm\w\w', str)
if result: #if result is not None
    print(result.group())
```

```
import re
str = 'man sun mop run'
prog = re.compile(r'm\w\w')
result = prog.search(str)
if result: #if result is not None
    print(result.group())
```

```
search(): Combination of compile and run

- Point: Returns only the first string matching the RE
```

Regular Expressions

Example-2: findall()

```
import re
str = 'man sun mop run'
result = re.findall(r'm\w\w', str)
print(result)
```

findall()

- Returns all the matching strings
- Returns in the form of the list

Regular Expressions

Example-3: match()

```
import re
str = 'man sun mop run'
result = re.match(r'm\w\w', str)
print(result.group())
```

```
match()
- Returns the string only if it is found in the begining of the string
- Returns None, if the string is not found
```

Regular Expressions

Example-4: match()

```
import re
str = 'sun man mop run'
result =  re.match(r'm\w\w', str)
print(result)
```

```
match()
- Returns None, since the string is not found
```

Regular Expressions

Example-5: split()

```
import re
str = 'This; is the: "Core" Python\'s Lecturer'
result = re.split(r'\w+', str)
print(result)
```

- split() - splits the RE
  - W : Split at non-alphanumeric character
  - + : Match 1 or more occurrences of characters

```
split()
– splits the string into pieces according to the given RE
```

Regular Expressions

Example-6: Find & Replace: sub()

```
import re
str = 'Kumbhmela will be conducted at Ahmedabad in India.'
res = re.sub(r'Ahmedabad', 'Allahabad', str)
print(res)
```

```
Syntax:
        sub(RE, new, old)
```

RE: Sequence Characters

RE: sequence characters

- Match only one character in the string

Character	Description
\d	Represents any digit(0 – 9)
\D	Represents any non-digit
\s	Represents white space Ex: \t\n\r\f\v
\S	Represents non-white space character
\w	Represents any alphanumeric(A-Z, a-z, 0-9)
\W	Represents non-alphanumeric\b
\b	Represents a space around words
\A	Matches only at start of the string
\Z	Matches only at end of the string

RE: sequence characters

Example-1:

To match all words starting with ‘a’

```
import re
str = 'an apple a day keeps the doctor away'
result = re.findall(r'a[\w]*', str)

# findall() returns a list, retrieve the elements from list
for word in result:
    print(word)
```

To match all words starting with ‘a’, not sub-words then RE will look like this

```
import re
str = 'an apple a day keeps the doctor away'
result = re.findall(r'\ba[\w]*\b', str)

# findall() returns a list, retrieve the elements from list
for word in result:
    print(word)
```

* Matches with 0 or more occurrences of the character

RE: sequence characters

Example-2:

To match all words starting with numeric digits

```
import re

str = 'The meeting will be conducted on 1st and 21st of every month'

result = re.findall(r'\d[\w]*', str)

#for word in result:

print(word)
```

* Matches with 0 or more occurrences of the character

RE: sequence characters

Example-3:

To retrieve all words having 5 characters

```
import re

str = 'one two three four five six seven 8 9 10'

result = re.findall(r'\b\w{5}\b', str)

print(result)
```

character	Description
\b	Matches only one space
\w	Matches any alpha numeric character
{5}	Repetition character

RE: sequence characters

Example-4: search()

To retrieve all words having 5 characters using search()

```
# search() will give the first matching word only.

import re

str = 'one two three four five six seven 8 9 10'

result = re.search(r'\b\w{5}', str)
```

character	Description
\b	Matches only one space
\w	Matches any alpha numeric character
{5}	Repetition character

RE: sequence characters

Example-5: findall()

To retrieve all words having 4 and above characters using findall()

```
import re
str = 'one two three four five six seven 8 9 10'
result = re.findall(r'\b\w{4,}\b', str)
print(result)
```

character	Description
\b	Matches only one space
\w	Matches any alpha numeric character
{4, }	Retrieve 4 or more characters

RE: sequence characters

Example-6: findall()

To retrieve all words having 3, 4, 5 characters using findall()

```
import re

str = 'one two three four five six seven 8 9 10'

result = re.findall(r'\b\w{3, 5}\b', str)

print(result)
```

character	Description
\b	Matches only one space
\w	Matches any alpha numeric character
{3, 5}	Retrieve 3, 4, 5 characters

RE: sequence characters

Example-7: findall()

To retrieve only single digit using findall()

```
import re

str = 'one two three four five six seven 8 9 10'

result = re.findall(r'\b\d\b', str)

print(result)
```

character	Description
\b	Matches only one space
\d	Matches only digit

RE: sequence characters

Example-7: findall()

To retrieve all words starts with ‘t’ from the end of the string

```
import re
str = 'one two three one two three'
result = re.findall(r't{\w}*\z', str)
print(result)
```

character	Description
\z	Matches from end of the string
\w	Matches any alpha numeric character
t	Starting character is ‘t’

RE: Quantifiers

RE: Quantifiers

- Characters which represents more than 1 character to be matched in the string

Character	Description
+	1 or more repetitions of the preceding regexp
*	0 or more repetitions of the preceding regexp
?	0 or 1 repetitions of the preceding regexp
{m}	Exactly m occurrences
{m, n}	From m to n. m defaults to 0 n defaults to infinity

RE: Quantifiers

Example-1:

To retrieve phone number of a person

```
import re
str = 'Tomy: 9706612345'
res = re.serach(r'\d+', str)
print(res.group())
```

character	Description
\d	Matches from any digit
+	1 or more repetitions of the preceding regexp

RE: Quantifiers

Example-2:

To retrieve only name

```
import re

str = 'Tomy: 9706612345'

res = re.serach(r'\D+', str)

print(res.group())
```

character	Description
\D	Matches from any non-digit
+	1 or more repetitions of the preceding regexp

RE: Quantifiers

Example-3:

To retrieve all words starting with “an” or “ak”

```
import re

str = 'anil akhil anant arun arati arundhati abhijit ankur'

res = re.findall(r'a[nk][\w]*', str)

print(res)
```

RE: Quantifiers

Example-4:

To retrieve DoB from a string

```
import re

str = 'Vijay 20 1-5-2001, Rohit 21 22-10-1990, Sita 22 15-09-2000'

res = re.findall(r'\d{2}-\d{2}-\d{4}', str)

print(res)
```

RE	Description
\d{2}-\d{2}-\d{4}	Retrieves only numeric digits in the format of 2digits-2digits-4digits

RE: Special Character

RE: Special Characters

Character	Description
\	Escape special character nature
.	Matches any character except new line
^	Matches begining of the string
$	Matches ending of a string
[...]	Denotes a set of possible characters Ex: [6b-d] matches any characters 6, b, c, d
[^...]	Matches every character except the ones inside brackets Ex: [^a-c6] matches any character except a, b, c or 6
(...)	Matches the RE inside the parentheses and the result can be captured
R | S	matches either regex R or regex S

RE: Special Characters

Example-1:

To search whether a given string is starting with ‘He’ or not

```
import re

str = "Hello World"

res = re.search(r"^He", str)

if res:

    print("String starts with 'He'")

else

    print("String does not start with 'He'")
```

RE	Description
“^He”	Search from the begining

RE: Special Characters

Example-2:

To search whether a given string is starting with ‘He’ or not from the end

```
import re

str = "Hello World"

res = re.search(r"World$", str)

if res:

    print("String ends with 'World'")

else

    print("String does not end with 'World'")
```

RE	Description
“World$”	Search from the end

RE: Special Characters

Example-3:

To search whether a given string is starting with ‘World’ or not from the end by ignoring the case

```
import re

str = "Hello World"

res = re.search(r"world$", str, re.IGNORECASE)

if res:

    print("String ends with 'world'")

else:

    print("String does not end with 'world'")
```

RE	Description
“World$”	Search from the end
re.IGNORECASE	Ignore the case

re.IGNORECASE

RE: Special Characters

Example-4:

To retrieve the timings am or pm

```
import re

str = 'The meeting may be at 8am or 9am or 4pm or 5pm.'

res = re.findall(r'\dam|\dpm', str)

print(res)
```

RE: On Files

RE: On Files

Example-1:

To retrieve the emails from the file

```
import re

# open file for reading

f = open('mails.txt', 'r')


# repeat for each line of the file

for line in f:

    res = re.findall(r'\s+@\S+', line)


# display if there ara some elements in result

if len(res)>0:

    print(res)

# close the file

f.close()
```

RE: On Files

Example-2:

To retrieve the data and write to another file

```
# Open the files

f1 = open('salaries.txt', 'r')

f1 = open('newfile.txt', 'w')


# repeat for each line of the file f1

for line in fi:

    res1 = re.search(r'\d{4}', line) # exptract id no from f1

    res2 = re.search(r'\d{4,}.\d{2}', line) # extract salary from f1

    print(res1.group(), res2.group()) # display them

    f2.write(res1.group()+"\t") # write id no into f2

    f2.write(res2.group()+"\n") # write salary into f2

# close the files

f1.close()

f2.close()
```

RE: On HTML Files

RE: On HTML Files

Example-1:

To retrieve info from the HTML file

Step-1:

import urllib.request	Import this module

f = urllib.request.urlopen(r'file:///path')
Ex:
f = urllib.request.urlopen(r'file:///~|Python\sample.html')	

urllib.request	Module name
urlopen	To open the html files
file:///	Protocol to open the local files
~|Python\sample.html	Under home DIR, under Python sub-DIR the sample.html file is present

RE: On HTML Files

Example-1:

Step-2: read and decode

text = f.read()	To read the file content
str = text.decode()	Since the HTML file contains the information in the byte strings

Step-3: Apply RE

```
r'<td>\w+</td>\s<td>(\w+)<\td>\s<td>(\d\d.\d\d)<\td>'
```

THANK YOU

Threads

Team **E**mertxe

Introduction

Creating Threads

Creating Threads

Introduction

- Python provides ‘Thread’ class of threading module to create the threads
- Various methods of creating the threads:
  - Method-1: Without using the class
  - Method-2: By creating a sub-class to Thread class
  - Method-3: Without creating a sub-class to Thread class

Creating Threads

Method-1: Without using class

- Step-1:
  - – Create a thread by creating an object class and pass the function name as target for the thread

Syntax	t = Thread(target = function_name, [args = (arg1, arg2, ...)])
target	Represents the function on which thread will act
args	Represents the tuple of arguments which are passed to the function

- Step-2:
  - – Start the thread by using start() method

```
t.start()
```

Creating Threads

Program-1: No arguments

Creating a thread without using a class

```
from threading import *

#Create a function
def display():
    print("Hello I am running")

#Create a thread and run the function 5 times
for i in range(5):
    #Create the thread and specify the function as its target
    t = Thread(target = display)

    #Run the thread
    t.start()
```

Output:

```
Hello I am running
Hello I am running
Hello I am running
Hello I am running
Hello I am running
```

Creating Threads

Program-2: With arguments

Creating a thread without using a class

#To pass arguments to a function and execute it using a thread

```
from threading import *

#Create a function
def display(str):
    print(str)

#Create a thread and run the function for 5 times
for i in range(5):
    t = Thread(target = display, args = ("Hello", ))
    t.start()
```

```
Output:

Hello
Hello
Hello
Hello
Hello
```

Creating Threads

Method-2: Creating Sub-class to Thread

- Step-1: Create a new class by inheriting the Thread class

Example	class MyThread(Thread):
MyThread	New Class
Thread	Base Class

- Step-2: Create an Object of MyThread class

```
t1 = MyThread()
```

- Step-3: Wait till the thread completes

```
t1.join()
```

Creating Threads:

Program-1: Creating Sub-class to Thread

Creating a thread by creating the sub-class to thread class

```
#Creating our own thread
from threading import Thread

#Create a class as sub class to Thread class
class MyThread(Thread):

    #Override the run() method of Thread class
    def run(self):
        for i in range(1, 6):
            print(i)

#Create an instance of MyThread class
t1 = MyThread()

#Start running the thread t1
t1.start()

#Wait till the thread completes its job
t1.join()
```

Output:

```
1
2
3
4
5
```

run() method will override the run() method in the Thread class

Creating Threads:

Program-2:

Creating a thread that access the instance variables of a class

```
#A thread that access the instance variables
from threading import *

#Create a class as sub class to Thread class
class MyThread(Thread):
    def __init__(self, str):
        Thread.__init__(self)
        self.str = str

    #Override the run() method of Thread class
    def run(self):
        print(self.str)

#Create an instance of MyThread class and pass the string
t1 = MyThread("Hello")

#Start running the thread t1
t1.start()

#Wait till the thread completes its job
t1.join()
```

Output:

```
Hello
```

Thread.__init__(self): Calls the constructor of the Thread class

Creating Threads

Method-3: Without creating sub-class to Thread class

- Step-1: Create an independent class
- Step-2: Create an Object of MyThread class

```
obj = MyThread('Hello')
```

- Step-3: Create a thread by creating an object to 'Thread' class

```
t1 = Thread(target = obj.display, args = (1, 2))
```

Creating Threads

Method-3: Without creating sub-class to Thread class: Program

Creating a thread without sub-class to thread class

```
from threading import *

#Create our own class
class MyThread:

    #A constructor
    def __init__(self, str):
        self.str = str

    #A Method
    def display(self, x, y):
        print(self.str)
        print("The args are: ", x, y)

#Create an instance to our class and store Hello string
Obj = MyThread("Hello")

#Create a thread to run display method of Obj
t1 = Thread(target = Obj.display, args = (1, 2))

#Run the thread
t1.start()
```

```
Output:

Hello
The args are:  1 2
```

Thread **C**lass **M**ethods

Single Tasking using a Thread

Single Tasking Thread

Introduction

- A thread can be employed to execute one task at a time
- Example:
    - Suppose there are three task executed by the thread one after one, then it is called single tasking

Problem: Preparation of the Tea

Task-1: Boil milk and tea powder for 5 mins

Task-2: Add sugar and boil for 3 mins

Task-3: Filter it and serve

```
#A method that performs 3 tasks one by one
     def prepareTea(self):
          self.task1()
          self.task2()
          self.task3()
```

Single Tasking Thread

Program

```
#Single tasking using a single thread
from threading import *
from time import *
```

```
#Create our own class
class MyThread:
      #A method that performs 3 tasks one by one
      def prepareTea(self):
            self.task1()
            self.task2()
            self.task3()

      def task1(self):
            print("Boil milk and tea powder for 5 mins...", end = '')
            sleep(5)
            print("Done")

      def task2(self):
            print("Add sugar and boil for 3 mins...", end = '')
            sleep(3)
            print("Done")

      def task3(self):
            print("Filter and serve...", end = '')
            print("Done")
```

```
#Create an instance to our class
obj = MyThread()

#Create a thread and run prepareTea method of Obj
t = Thread(target = obj.prepareTea)
t.start()
```

Multi Tasking using a Multiple Thread

Multi Tasking Threads

Program-1

```
#Multitasking using two threads
from threading import *
from time import *
```

```
#Create our own class
class Theatre:
      #Constructor that accepts a string
      def __init__(self, str):
            self.str = str

      #A method that repeats for 5 tickets
      def movieshow(self):
            for i in range(1, 6):
                  print(self.str, ":", i)
                  sleep(1)
```

```
#Create two instamces to Theatre class
obj1 = Theatre("Cut Ticket")
obj2 = Theatre("Show chair")

#Create two threads to run movieshow()
t1 = Thread(target = obj1.movieshow)
t2 = Thread(target = obj2.movieshow)

#Run the threads
t1.start()
t2.start()
```

```
Output:

Run-1:

Cut Ticket : 1
Show chair : 1
Cut Ticket : 2
Show chair : 2
Cut Ticket : 3
Show chair : 3
Cut Ticket : 4
Show chair : 4
Cut Ticket : 5
Show chair : 5

Run-2:

Cut Ticket : 1
Show chair : 1
Cut Ticket : 2
Show chair : 2
Show chair : 3
Cut Ticket : 3
Cut Ticket : 4
Show chair : 4
Cut Ticket : 5
Show chair : 5
```

Race Condition

Multi Tasking **T**hreads

Race-Condition

- Using more than one thread is called Multi-threading, used in multi-tasking
- Race-condition is a situation where threads are not acting in a expected sequence, leading to the unreliable output
- Race-condition can be avoided by ‘Thread Synchronization’

Multi Tasking Threads

Program-2

```
#Multitasking using two threads
from threading import *
from time import *
```

```
#Create our own class
class Railway:

	#Constrauctor that accepts no. of available berths
	def __init__(self, available):
		self.available = available

	#A method that reserves berth
	def reserve(self, wanted):

		#Display no. of available births
		print("Available no. of berths = ", self.available)

		#If available >= wanted, allot the berth
		if (self.available >= wanted):
			#Find the thread name
			name = current_thread().getName()

			#Display the berth is allotted for the person
			print("%d berths are alloted for %s" % (wanted, name))

			#Make time delay so that ticket is printed
			sleep(1.5)

			#Decrease the number of available berths
			self.available -= wanted

		else:

			#If avaible < wanted, then say sorry
			print("Sorry, no berths to allot")
```

```
#Create instance to railway class
#Specify only one berth is available
obj = Railway(1)

#Create two threads and specify 1 berth is needed
t1 = Thread(target = obj.reserve, args = (1, ))
t2 = Thread(target = obj.reserve, args = (1, ))

#Give names to the threads
t1.setName("First Person")
t2.setName("Second Person")

#Start running the threads
t1.start()
t2.start()
```

The output of the above code is not correct. Run multiple times & see the o/p

Thread Synchronization

Thread Synchronization

Introduction

Thread Synchronization OR Thread Safe	When a thread is already acting on an object, preventing any other thread from acting on the same object is called 'Thread Synchronization' OR 'Thread Safe'
Synchronized Object	The object on which the threads are synchronized is called synchronized object or Mutex(Mutually exclusive lock)
Techniques	1. Locks (Mutex) 2. Semaphores

Thread Synchronization

Mutex

1. Creating the lock

   ```
   l = Lock()
   ```

2. To lock the current object

   ```
   l.acquire()
   ```

3. To unlock or release the object

   ```
   l.release()
   ```

Thread Synchronization

Mutex: Program

```
#Create our own class
class Railway:

        #Constructor that accepts no. of available berths
        def __init__(self, available):
                self.available = available

                #Create a lock Object
                self.l = Lock()

        #A method that reserves berth
        def reserve(self, wanted):

                #lock the current object
                self.l.acquire()


                #Display no. of available births
                print("Available no. of berths = ", self.available)

                #If available >= wanted, allot the berth
                if (self.available >= wanted):
                        #Find the thread name
                        name = current_thread().getName()

                        #Display the berth is allotted for the person
                        print("%d berths are alloted for %s" % (wanted, name))

                        #Make time delay so that ticket is printed
                        sleep(1.5)

                        #Decrease the number of available berths
                        self.available -= wanted

                else:

                        #If avaible < wanted, then say sorry
                        print("Sorry, no berths to allot")

                #Task is completed, release the lock
                self.l.release()
```

```
#Create instance to railway class
#Specify only one berth is available
obj = Railway(1)


#Create two threads and specify 1 berth is needed
t1 = Thread(target = obj.reserve, args = (1, ))
t2 = Thread(target = obj.reserve, args = (1, ))


#Give names to the threads
t1.setName("First Person")
t2.setName("Second Person")


#Start running the threads
t1.start()
t2.start()
```

Thread Synchronization

Semaphore

Semaphore	Is an object that provides synchronization based on a counter
Creation	l = Semaphore(counter) #Counter value will be 1 by default

Usage	#Acquire the lock
	l.acquire()
	#Critical Section
	#Release the lock
	l.release()

Thread Synchronization

Mutex: Program

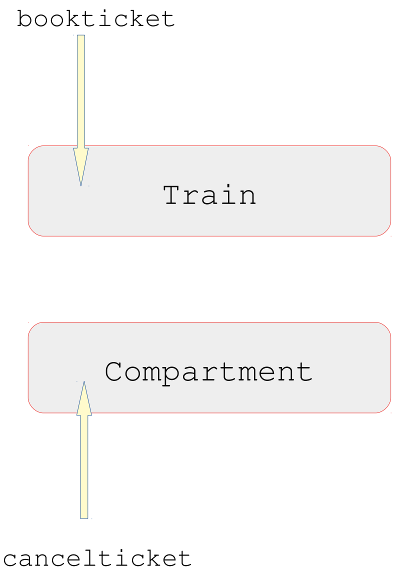
```
#Create our own class
class Railway:

       #Constrauctor that accepts no. of available berths
       def __init__(self, available):
              self.available = available

              #Create a lock Object
              self.l = Semaphore()

       #A method that reserves berth
       def reserve(self, wanted):

              #lock the current object
              self.l.acquire()


              #Display no. of available births
              print("Available no. of berths = ", self.available)

              #If available >= wanted, allot the berth
              if (self.available >= wanted):
                     #Find the thread name
                     name = current_thread().getName()

                     #Display the berth is allotted for the person
                     print("%d berths are alloted for %s" % (wanted, name))

                     #Make time delay so that ticket is printed
                     sleep(1.5)

                     #Decrease the number of available berths
                     self.available -= wanted

              else:

                     #If avaible < wanted, then say sorry
                     print("Sorry, no berths to allot")

              #Task is completed, release the lock
              self.l.release()
```

```
#Create instance to railway class
#Specify only one berth is available
obj = Railway(1)

#Create two threads and specify 1 berth is needed
t1 = Thread(target = obj.reserve, args = (1, ))
t2 = Thread(target = obj.reserve, args = (1, ))

#Give names to the threads
t1.setName("First Person")
t2.setName("Second Person")

#Start running the threads
t1.start()
t2.start()
```

Dead Locks

Dead Locks

Introduction



```
#Book Ticket thread
lock-1:
lock on train
    lock-2:
    lock on compartment

#Cancel Ticket thread
lock-2:
lock on compartment
    lock-1:
    lock on train
```

When a thread has locked an object and waiting for another object to be released by another thread, and the other thread is also waiting for the first thread to release the fisrt object, both threads will continue to wait forever. This condition is called Deadlock

Dead Locks

Program

```
#Dead lock of threads
from threading import *

#Take two locks
l1 = Lock()
l2 = Lock()
```

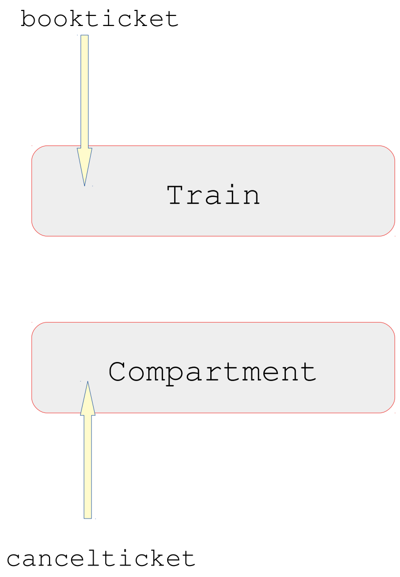
```
#Create a function for booking a ticket
def bookticket():
    l1.acquire()
    print("Bookticket locked train")
    print("Bookticket wants to lock on compartment")

    l2.acquire()
    print("Bookticket locked compartment")
    l2.release()
    l1.release()
    print("Booking ticket done...")
```

```
#Create a function for cancelling a ticket
def cancelticket():
    l2.acquire()
    print("Cancelticket locked compartment")
    print("Cancelticket wants to lock on train")

    l1.acquire()
    print("Cancelticket locked train")
    l1.release()
    l2.release()
    print("Cancellation of ticket is done...")
```

```
#Create two threads and run them
t1 = Thread(target = bookticket)
t2 = Thread(target = cancelticket)

t1.start()
t2.start()
```

Dead Locks

Avoiding



```
#Book Ticket thread
lock-1:
lock on train
    lock-2:
    lock on compartment

#Cancel Ticket thread
lock-1:
lock on compartment
    lock-2:
    lock on train
```

Dead Locks

Program: Avoiding Deadlocks

```
#Dead lock of threads
from threading import *

#Take two locks
l1 = Lock()
l2 = Lock()
```

```
#Create a function for booking a ticket
def bookticket():
    l1.acquire()
    print("Bookticket locked train")
    print("Bookticket wants to lock on compartment")

    l2.acquire()
    print("Bookticket locked compartment")
    l2.release()
    l1.release()
    print("Booking ticket done...")
```

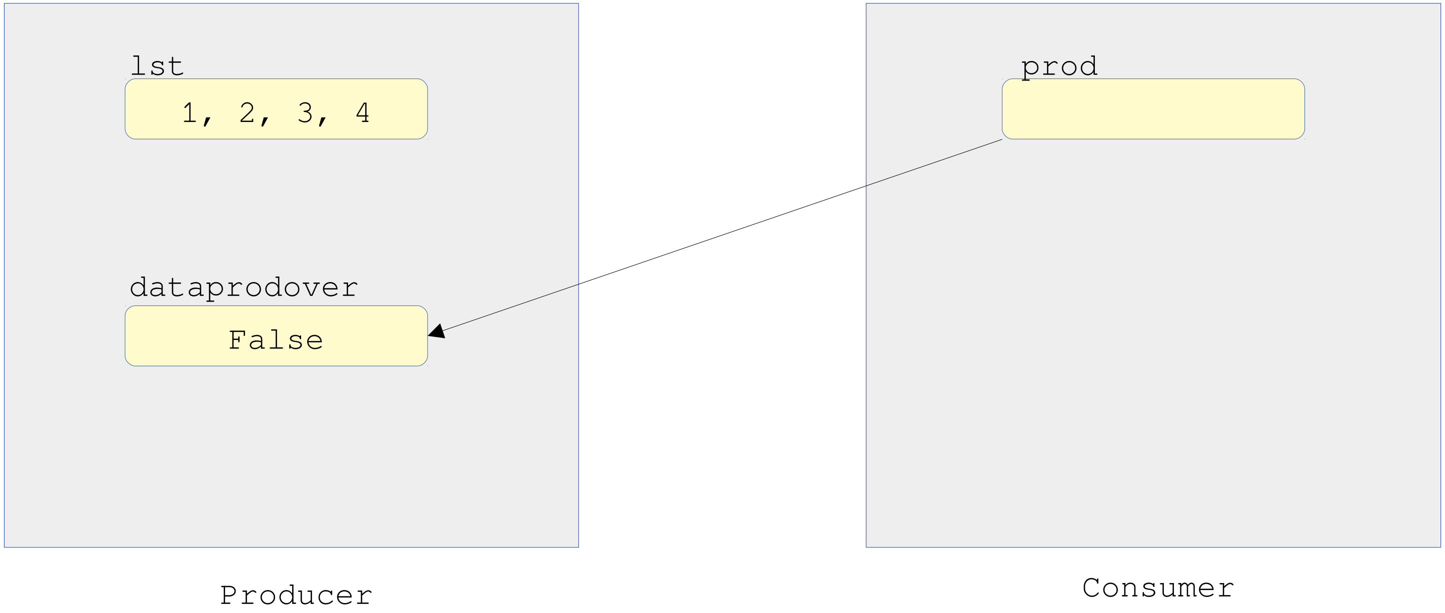
```
#Create a function for cancelling a ticket
def cancelticket():
    l1.acquire()
    print("Cancelticket locked compartment")
    print("Cancelticket wants to lock on train")

    l2.acquire()
    print("Cancelticket locked train")
    l2.release()
    l1.release()
    print("Cancellation of ticket is done...")
```

```
#Create two threads and run them
t1 = Thread(target = bookticket)
t2 = Thread(target = cancelticket)

t1.start()
t2.start()
```

Communication between Threads

Threads Communication

Introduction



Threads Communication

Program

```
from threading import *
from time import *
```

```
#Create producer class
class Producer:
    def __init__(self):
        self.lst = []
        self.dataprodover = False

    def produce(self):
        #create 1 to 10 items and add to the list
        for i in range(1, 11):
            self.lst.append(i)
            sleep(1)
            print("Item produced...")

        #Inform teh consumer that the data production is completed
        self.dataprodover = True
```

```
#Create the consumer class
class Consumer:
    def __init__(self, prod):
        self.prod = prod

    def consume(self):
        #sleep for 100ms a s long as dataprodover is False
        while self.prod.dataprodover == False:
            sleep(0.1)
        #Display the content of list when data production is over
        print(self.prod.lst)
```

```
#Create producer object
p = Producer()

#Create consumer object and pass producer object
c = Consumer(p)

#Create producer and consumer threads
t1 = Thread(target = p.produce)
t2 = Thread(target = c.consume)

#Run the threads
t1.start()
t2.start()
```

Threads Communication

Improving Efficiency

- Using notify() and wait()
- Using queue

Threads Communication

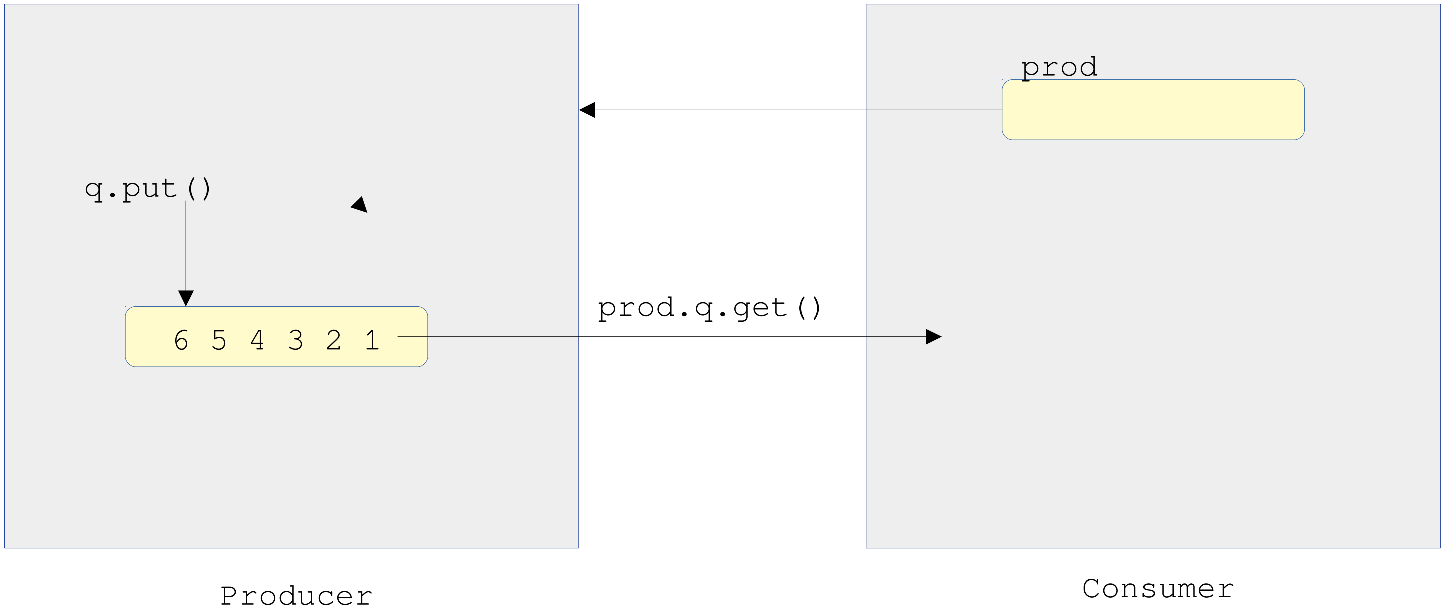
Improving Efficiency: notify(), wait()

```
#Create Producer Class
class Producer:
    def __init__(self):
        self.lst = []
        self.cv = Condition()

    def produce(self):
        #Lock the conditional object
        self.cv.acquire()

        #Create 1 to 10 items and add to the list
        for i in range(1, 11):
            self.lst.append(i)
            sleep(1)
            print("Item produced...")

        #Inform the consumer that production is completed
        self.cv.notify()

        #Release the lock
        self.cv.release()
```

```
#Create Consumer class
class Consumer:
    def __init__(self, prod):
        self.prod = prod

    def consume(self):
        #Get lock on condition object
        self.prod.cv.acquire()

        #Wait only for 0 seconds after the production
        self.prod.cv.wait(timeout = 0)

        #Release the lock
        self.prod.cv.release()

        #Display the contenst of list
        print(self.prod.lst)
```

Threads Communication

Improving Efficiency: Queues



Threads Communication

Improving Efficiency: Queues

```
#Create Producer class
class Producer:
    def __init__(self):
        self.q = Queue()

    def produce(self):
        #Create 1 to 10 items and add to the queue
        for i in range(1, 11):
            print("Producing item: ", i)
            self.q.put(i)
            sleep(1)
```

```
#Create Consumer class
class Consumer:
    def __init__(self, prod):
        self.prod = prod

    def consume(self):
        #Receive 1 to 10 items from the queue
        for i in range(1, 11):
            print("Receiving item: ", self.prod.q.get(i))
```

Daemon Threads

Daemon Threads

Introduction

- Sometimes, threads should be run continuosly in the memory
- Example
  - Internet Server
  - Garbage collector of Python program
- These threads are called Daemon Threads
- To make the thread as Daemon, make

```
d.daemon = True
```

Daemon Threads

Program

```
#To display numbers from 1 to 5 every second
def display():
    for i in range(5):
        print("Normal thread: ", end = '')
        print(i + 1)
        sleep(1)
```

```
#To display numbers from 1 to 5 every second
def display():
    for i in range(5):
        print("Normal thread: ", end = '')
        print(i + 1)
        sleep(1)
```

```
#Create a normal thread and attach it to display() and run it
t = Thread(target = display)
t.start()

#Create another thread and attach it to display_time()
d = Thread(target = display_time)

#make the thread daemon
d.daemon = True

#Run the daemon thread
d.start()
```

THANK YOU

Python2 Vs Python3

Team **E**mertxe

Division

Division

2.x	3.x
`print 5 / 2`	`print (5 / 2)`
Output 2	Output 2.5

Print

Print

2.x	3.x
print "Hello World"	print ("Hello World")
Output Hello World	Output Hello World

Unicode

Unicode

2.x	3.x

print(type('Hello'))
print(type(b'Hello'))
	print(type('Hello'))
	print(type(b'Hello'))

Output <type 'str'> <type 'str'>	Output <class 'str'> <class 'bytes'>

xrange

Xrange

2.x	3.x

`for x in xrange(1, 5):`
`    print(x)`
	`for x in xrange(1, 5):`
	`    print(x)`

Output
 	
1
2
3
4
	Output
	
	`Original exception was:`
	`Traceback (most recent call last):`
	`  File "1.py", line 1, in <module>`
	`    for x in xrange(1, 5):`
	`NameError: name 'xrange' is not defined`

Raising Exceptions

Raising Exceptions

2.x	3.x
print 'Python' raise IOError, "file error"	print ('Python') raise IOError("file error")
Output Traceback (most recent call last): File "1.py", line 2, in <module> raise IOError, "file error" IOError: file error	Output Original exception was: Traceback (most recent call last): File "1_3x.py", line 2, in <module> raise IOError("file error") OSError: file error

Raising Exceptions

2.x	3.x
print 'Python' try: Generate_Name_error except NameError, err: print err, '--> our error message'	print ('Python') try: Generate_Name_error except NameError as err: print (err, '--> our error message')
Output Python name 'Generate_Name_error' is not defined --> our error message	Output Python name 'Generate_Name_error' is not defined --> our error message

THANK YOU