21CSC203P -
Advanced Programming Practice
Unit-4 :
PYTHONIC PROGRAMMING PARADIGM

Outline of the course

Functional Programming Paradigm:
Concepts,
PureFunction and Built-in Higher-Order Functions;
Logic Programming Paradigm:
Structures, Logic, and Control
Parallel Programming Paradigm:
Shared and Distributed memory
Multi-Processing – Ipython
Network Programming Paradigm:
Socket; Socket Types;
Creation and Configuration of Sockets in TCP / UDP – Client / Server Model

Introduction

- Functional programming is a programming paradigm in which it is tried to bind each and everything in pure mathematical functions. It is a declarative type of programming style that focuses on what to solve rather than how to solve.
- Functional programming paradigm is based on lambda calculus.
- Instead of statements, functional programming makes use of expressions. Unlike a statement, which is executed to assign variables, evaluation of an expression produces a value.
- Functional programming is a declarative paradigm because it relies on expressions and declarations rather than statements. Unlike procedures that depend on a local or global state, value outputs in FP depend only on the arguments passed to the function.
- Functional programming consists only of PURE functions.

•In functional programming, control flow is expressed by combining function calls, rather than by assigning values to variables.

Example:

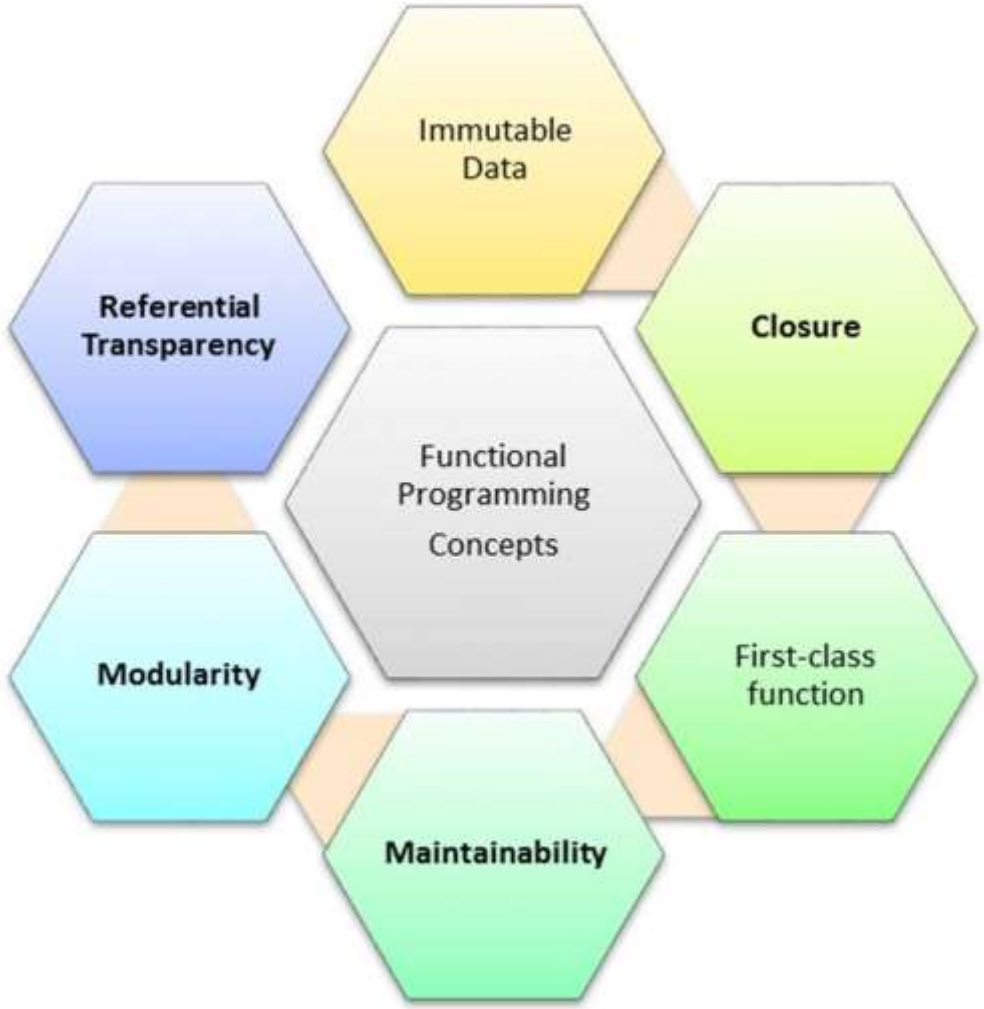
```
sorted(p.name.upper() for p in people if len(p.name) > 5)
```

Characteristics of Functional Programming

- Functional programming method focuses on results, not the process
- Emphasis is on what is to be computed
- Data is immutable
- Functional programming Decompose the problem into 'functions
- It is built on the concept of mathematical functions which uses conditional expressions and recursion to do perform the calculation
- It does not support iteration like loop statements and conditional statements like If-Else
- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports higher-order functions and lazy evaluation features.
- Functional programming languages don’t support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.

Concepts of FP

- Pure functions
- Recursion
- Referential transparency
- Functions are First-Class and can be Higher-Order
- Immutability



Functional Programming vs Procedure Programming

```
#Functional Programming

num = 1
def function_to_add_one(num):
    num += 1
    return num

print("Num is :",function_to_add_one(num)) #global num =1
print("Num is :",function_to_add_one(num)) #global num =1
print("Num is :",function_to_add_one(num)) #global num =1
```

```
Num is : 2
Num is : 2
Num is : 2
```

```
#Procedural  Programming
'''The basic rules for global keyword in Python are:
When we create a variable inside a function, it's local by default.
When we define a variable outside of a function, it's global by default. ...
We use global keyword to read and write a global variable inside a function'''

num = 1
def procedure_to_add_one():
    global num
    num += 1
    return num

procedure_to_add_one()
procedure_to_add_one()
procedure_to_add_one()
```

1. Pure functions

- Pure functions always return the same results when given the same inputs. Consequently, they have no side effects.

- Properties of Pure functions are:
  - First, they always produce the same output for same arguments irrespective of anything else.
  - Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.
- A simple example would be a function to receive a collection of numbers and expect it to increment each element of this collection.
- We receive the numbers collection, use map with the ***inc*** function to increment each number, and return a new list of incremented numbers.

Example:

```
def inc(x):
        return x+1
list=[8,3,7,5,2,6]
x=map(inc,list)   #print(list)
print(x)
```

```
var z = 15;

function add(x, y) {
    return x + y;
}
```

Note : Function involving Reading files,
using global data, random numbers are impure functions

Note: if a function relies on the global variable or class member's data, then it is not pure. And in such cases, the return value of that function is not entirely dependent on the list of arguments received as input and can also have side effects.

A side effect is a change in the state of an application that is observable outside the called function other than its return value

2. Recursion

- In the functional programming paradigm, there is no for and while loops. Instead, functional programming languages rely on recursion for iteration. Recursion is implemented using recursive functions, which repetitively call themselves until the base case is reached.

```
function sumRange(start, end, acc) {
    if (start > end)
        return acc;
    return sumRange(start + 1, end, acc + start)
}
```

- The above code performs recursion task as the loop by calling itself with a new start and a new accumulator.

Referential transparency

- An expression is said to be referentially transparent if it can be replaced with its corresponding value without changing the program's behaviour. As a result, evaluating a referentially transparent function gives the same value for fixed arguments. If a function consistently yields the same result for the same input, it is referentially transparent.
- Functional programs don’t have any assignment statements. For storing additional values in a program developed using functional programming, new variables must be defined. State of a variable in such a program is constant at any moment in time
  - pure function + immutable date = referential transparency

```
int add(int a, int b)
 {
 return a + b
 }
int mult(int a, int b)
 {
 return a * b;
 }
int x = add(2, mult(3, 4));
```

- Let’s implement a square function:
  - This (pure) function will always have the same output, given the same input.
  - Passing “2” as a parameter of the square function will always returns 4. So now we can replace the (square 2) with 4.

Immutability

- In functional programming you cannot modify a variable after it has been initialized. You can create new variables and this helps to maintain state throughout the runtime of a program.

```
var x = 1;
x = x + 1;
```

- In imperative programming, this means “take the current value of x, add 1 and put the result back into x.” In functional programming, however, x = x + 1 is illegal. That’s because there are technically no variables in functional programming.
- Using immutable data structures, you can make single or multi-valued changes by copying the variables and calculating new values,
- Since FP doesn’t depend on shared states, all data in functional code must be immutable, or incapable of changing

Functions are First-Class and can be Higher-Order

- A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.
- Higher-order functions are functions that take at least one first-class function as a parameter
- Examples:
  - name_lengths = map(len, ["Bob", "Rob", "Bobby"])
- Higher Order functions are map, reduce, filter

Functional style of getting a sum of a list:

```
new_lst = [1, 2, 3, 4]
def sum_list(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        return lst[0] + sum_list(lst[1:])
print(sum_list(new_lst))
```

 # or the pure functional way in python using higher order function

```
import functools
print(functools.reduce(lambda x, y: x + y, new_lst))
```

Functions are First-Class and can be Higher-Order

Map

map() can run the function on each item and insert the return values into the new collection. Example:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])

import random

names = ['Seth', 'Ann', 'Morganna']
team_names = map(lambda x: random.choice(['A Team','B Team']),names)

print names

// ['A Team', 'B Team', 'B Team']
```

reduce()

- reduce() is another higher order function for performing iterations. It takes functions and collections of items, and then it returns the value of combining the items

Example:

```
sum = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print sum        // 10
```

Functions are First-Class and can be Higher-Order ..cont

filter()

- filter function expects a true or false value to determine if the element should or should not be included in the result collection. Basically, if the call-back expression is true, the filter function will include the element in the result collection. Otherwise, it will not.

Example:

```
def f(x):
    return x%2 != 0 and x%3 ==0
filter(f, range(2,25))
```

Functions are First-Class and can be Higher-Order ..cont

```
# Example of using map, filter, and reduce in Python
data = [1, 2, 3, 4, 5]

# Using the map to apply a function to each element
# Lambda function returns the square of x
result1 = map(lambda x: x * 2, data)  # Result: [2, 4, 6, 8, 10]

# Using the filter to filter elements based on a condition
# Lambda function returns True for an even number
result2 = filter(lambda x: x % 2 == 0, data)  # Result: [2, 4]

# Using reduce to aggregate elements
# Lambda function returns product of x and y
from functools import reduce
result3 = reduce(lambda x, y: x * y, data)  # Result: 120
```

```
lambda arguments : expression
```

Functional Programming – Non Strict Evaluation

- In programming language theory, lazy evaluation, or call-by-need[1] is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations
- Allows Functions having variables that have not yet been computed

In Python, the logical expression operators and, or, and if-then-else are all non-strict. We sometimes call them short-circuit operators because they don't need to evaluate all arguments to determine the resulting value.

The following command snippet shows the and operator's non-strict feature:

```
>>> 0 and print("right")
0
>>> True and print("right")
Right
```

When we execute the preceding command snippet, the left-hand side of the and operator is equivalent to False; the right-hand side is not evaluated. When the left-hand side is equivalent to True, the right-hand side is evaluated

Functional Programming – lambda calculus

Lambda expressions in Python and other programming languages have their roots in lambda calculus. Lambda calculus can encode any computation. Functional languages get their origin in mathematical logic and lambda calculus

In Python, we use the lambda keyword to declare an anonymous function, which is why we refer to them as "lambda functions". An anonymous function refers to a function declared with no name.

when you want to pass a function as an argument to higher-order functions, that is, functions that take other functions as their arguments

Characteristics of Python lambda functions:

- A lambda function can take any number of arguments, but they contain only a single expression. An expression is a piece of code executed by the lambda function, which may or may not return any value.
- Lambda functions can be used to return function objects.
- Syntactically, lambda functions are restricted to only a single expression.

Syntax:

lambda argument(s): expression

Example:

```
remainder = lambda num: num % 2
print(remainder(5))
```

```
[(lambda x: x*x)(x) for x in [2,6,9,3,6,4,8]]
```

Functional Programming – Closure

Basically, the method of binding data to a function without actually passing them as parameters is called closure. It is a function object that remembers values in enclosing scopes even if they are not present in memory.

Example:.

```
def counter(start=0, step=1):
    x = [start]
    def _inc():
        x[0] += step
        return x[0]
    return _inc
c1 = counter()
c2 = counter(100, -10)
c1()
//1
c2()
90
```

Pure Functions in Python

- If a function uses an object from a higher scope or random numbers, communicates with files and so on, it might be impure

```
def multiply_2_pure(numbers):
    new_numbers = []
    for n in numbers:
        new_numbers.append(n * 2)
    return new_numbers

original_numbers = [1, 3, 5, 10]
changed_numbers = multiply_2_pure(original_numbers)
print(original_numbers) # [1, 3, 5, 10]
print(changed_numbers)  # [2, 6, 10, 20]
```

```
[1, 3, 5, 10]
[2, 6, 10, 20]
```

```
# Example1 : Impure Function
A = 5
def impure_sum(b): # A is out side function ,it has side effect
    return b + A

impure_sum(8)
```

```
13
```

```
# Example2 : Pure Function

def pure_sum(a, b):    #a and b inside function
    return a + b
print(impure_sum(6))
```

```
11
```

Built-in Higher Order Functions

Map

- The map function allows us to apply a function to every element in an iterable object

Filter

- The filter function tests every element in an iterable object with a function that returns either True or False, only keeping those which evaluates to True.

Combining map and filter

- As each function returns an iterator, and they both accept iterable objects, we can use them together for some really expressive data manipulations!

List Comprehensions

- A popular Python feature that appears prominently in Functional Programming Languages is list comprehensions. Like the map and filter functions, list comprehensions allow us to modify data in a concise, expressive way.

Anonymous Function

- In Python, anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Characteristics of Python lambda functions:

- A lambda function can take any number of arguments, but they contain only a single expression. An expression is a piece of code executed by the lambda function, which may or may not return any value.
- Lambda functions can be used to return function objects.
- Syntactically, lambda functions are restricted to only a single expression.

Syntax of Lambda Function in python

lambda arguments: expression

Example:

```
double = lambda x: x * 2
print(double(5))
# Output: 10
```

```
product = lambda x, y : x * y
print(product(2, 3))
```

Note: you want to pass a function as an argument to higher-order functions, that is, functions that take other functions as their arguments.

map() Function

Example Map with lambda

```
tup= (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)
newtuple = tuple(map(lambda x: x+3 , tup))
print(newtuple)
```

//with multiple iterables

```
list_a = [1, 2, 3]
list_b = [10, 20, 30]
 map(lambda x, y: x + y, list_a, list_b)
```

Example with Map

```
from math import sqrt
map(sqrt, [1, 4, 9, 16])
```

[1.0, 2.0, 3.0, 4.0]

```
map(str.lower, ['A', 'b', 'C'])
```

['a', 'b', 'c']

```
#splitting the input and convert to int using map
print(list(map(int, input.split(‘ ‘)))
```

```
numbers_list = [2, 6, 8, 10, 11, 4, 12, 7, 13, 17, 0, 3, 21]

mapped_list = list(map(lambda num: num % 2, numbers_list))

print(mapped_list)
```

map() Function

- map() function is a type of higher-order. As mentioned earlier, this function takes another function as a parameter along with a sequence of iterables and returns an output after applying the function to each iterable present in the sequence.

Syntax:

map(function, iterables)

Example without Map

```
my_pets = ['alfred', 'tabitha', 'william', 'arla']
uppered_pets = []
for pet in my_pets:
    pet_=pet.upper()
    uppered_pets.append(pet_)
print(uppered_pets)
```

Example with Map

```
my_pets = ['alfred', 'tabitha', 'william', 'arla']
uppered_pets=list(map(str.upper,my_pets)) print(uppered_pets)
//map with multiple list as input
circle_areas = [3.56773, 5.57668, 4.00914, 56.24241, 9.01344, 32.00013]
result = list(map(round, circle_areas, range(1,7)))
print(result)
```

filter() Function

- filter extracts each element in the sequence for which the function returns True.
- filter(), first of all, requires the function to return boolean values (true or false) and then passes each element in the iterable through the function, "filtering" away those that are false

Syntax:

filter(func, iterable)

The following points are to be noted regarding filter():

- Unlike map(), only one iterable is required.
- The func argument is required to return a boolean type. If it doesn't, filter simply returns the iterable passed to it. Also, as only one iterable is required, it's implicit that func must only take one argument.
- filter passes each element in the iterable through func and returns only the ones that evaluate to true. I mean, it's right there in the name -- a "filter".

Example:

```
def isOdd(x): return x % 2 == 1

filter(isOdd, [1, 2, 3, 4])

# output --->  [1, 3]
```

filter() Function

Example:

Python 3

scores = [66, 90, 68, 59, 76, 60, 88, 74, 81, 65]

def is_A_student(score):

    return score > 75

over_75 = list(filter(is_A_student, scores))

print(over_75)

```
'''
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)
for x in adults:
  print(x)
```

```
# Python 3
dromes = ("demigod", "rewire", "madam", "freer",
"anutforajaroftuna", "kiosk")

palindromes = list(filter(lambda word: word == word[::-1],
dromes))

print(palindromes)
```

```
#function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False
# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function
filtered = filter(fun, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

reduce() Function

- reduce, combines the elements of the sequence together, using a binary function. In addition to the function and the list, it also takes an initial value that initializes the reduction, and that ends up being the return value if the list is empty.
- The “reduce” function will transform a given list into a single value by applying a given function continuously to all the elements. It basically keeps operating on pairs of elements until there are no more elements left.
- reduce applies a function of two arguments cumulatively to the elements of an iterable, optionally starting with an initial argument

Syntax:

reduce(func, iterable[, initial])

Example:

```
reduce(lambda s,x: s+str(x), [1, 2, 3, 4], '')
#output '1234‘
my_list = [3,8,4,9,5]
reduce(lambda a, b: a * b, my_list)
#output 4320 ( 3*8*4*9*5)
```

```
from functools import reduce
y = filter(lambda x: (x>=3), (1,2,3,4))
print(list(y))


reduce(lambda a,b: a+b,[23,21,45,98])


nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18, 16,
        29, 21, 60, 27, 62, 59, 86, 56]
sum = reduce(lambda x, y : x + y, nums) / len(nums)
```

map(), filter() and reduce() Function

Using filter() within map():

```
c = map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4)))

print(list(c))
```

Using map() within filter():

```
c = filter(lambda x: (x>=3),map(lambda x:x+x, (1,2,3,4))) #lambda x: (x>=3)

print(list(c))
```

Using map() and filter() within reduce():

```
d = reduce(lambda x,y: x+y,map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4))))

print(d)
```

map(), filter() and reduce() Function

```
from functools import reduce
# Use map to print the square of each numbers rounded# to two decimal places
my_floats = [4.35, 6.09, 3.25, 9.77, 2.16, 8.88, 4.59]
# Use filter to print only the names that are less than or equal to seven letters
my_names = ["olumide", "akinremi", "josiah", "temidayo", "omoseun"]
# Use reduce to print the product of these numbers
my_numbers = [4, 6, 9, 23, 5]
map_result = list(map(lambda x: round(x ** 2, 3), my_floats))
filter_result = list(filter(lambda name: len(name) <= 7, my_names))
reduce_result = reduce(lambda num1, num2: num1 * num2, my_numbers)
print(map_result)
print(filter_result)
print(reduce_result)
```

Examples

```
#print the sum of even numbers from the user input
import functools
l = input("Enter the list of numbers separated by space").split(" ")
l = map(int,l)
def even_fil(x):
    flag  = False
    if x%2==0:
        flag=True
    return flag
even = list(filter(even_fil,l))
print(even)


s = functools.reduce(lambda x,y:x+y,even)
```

Examples

```
//Closure
def twice(x):
    def square():
        return x*x
    return square()*square()
def quad(x):
    return twice(x)
print(quad(3))
```

Examples

```
//partial to make some of the parameters as fixed

import functools

def s_total(a,b,c):
    return a*b+c    #a=5,b=15, c=from list

s = functools.partial(s_total,5,15)

l= [13,54,76,89,10]
for i in l:
    print(s(i))
```

Function vs Procedure

S.No	Functional Paradigms	Procedural Paradigm
1	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Derived from structured programming, based on the concept of modular programming or the *procedure call*
2	Main traits are Lambda alculus, compositionality, formula, recursion, referential transparency	Main traits are Local variables, sequence, selection, iteration, and modularization
3	**Functional** programming focuses on **expressions**	**Procedural** programming focuses on **statements**
4	Often recursive. Always returns the same output for a given input.	The output of a routine does not always have a direct correlation with the input.
5	Order of evaluation is usually undefined.	Everything is done in a specific order.
6	Must be stateless. i.e. No operation can have side effects.	Execution of a routine may have side effects.
7	Good fit for parallel execution, Tends to emphasize a divide and conquer approach.	Tends to emphasize implementing solutions in a linear fashion.

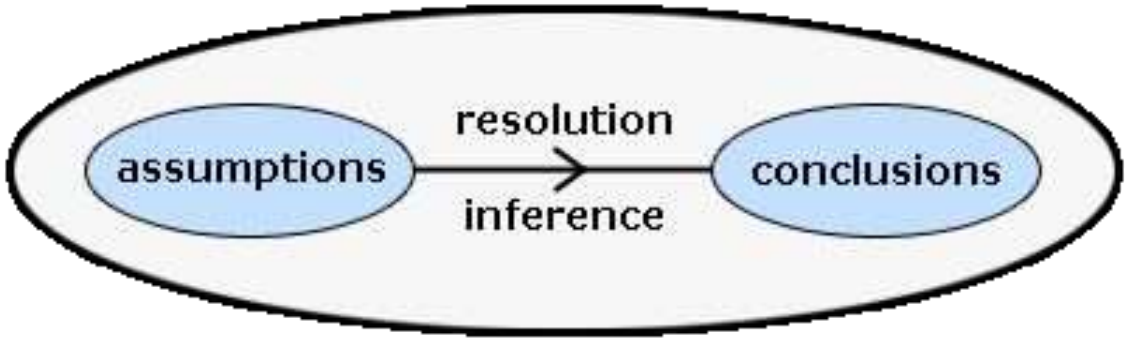
Function vs Object Oriented

S.No	Functional Paradigms	Object Oriented Paradigm
1	FP uses Immutable data.	OOP uses Mutable data.
2	Follows Declarative Programming based Model.	Follows Imperative Programming Model.
3	What it focuses is on: "What you are doing. in the programme."	What it focuses is on "How you are doing your programming."
4	Supports Parallel Programming.	No supports for Parallel Programming.
5	Its functions have no-side effects.	Method can produce many side effects.
6	Flow Control is performed using function calls & function calls with recursion.	Flow control process is conducted using loops and conditional statements.
7	Execution order of statements is not very important.	Execution order of statements is important.
8	Supports both "Abstraction over Data" and "Abstraction over Behavior."	Supports only "Abstraction over Data".

Logical Programming Paradigm

Logical Programming Paradigm

Logic programming is a paradigm where computation arises from proof search in a logic according to a fixed, predictable strategy. A logic is a language. It has syntax and semantics. A logic is a language. It has syntax and semantics. More than a language, it has inference rules.



Syntax:

this is the rules about how to form formulas; this is usually the easy part of a logic.

Semantics:

About the meaning carried by the formulas, mainly in terms of logical consequences.

Inference rules:

Inference rules describe correct ways to derive conclusions

Logical Programming Paradigm

Logic :

A Logic program is a set of predicates.

Predicates :

Define relations between their arguments. Logically, a Logic program states what holds. Each predicate has a name, and zero or more arguments. The predicate name is a atom. Each argument is an arbitrary Logic term. A predicate is defined by a collection of clauses.

Clause :

A clause is either a rule or a fact. The clauses that constitute a predicate denote logical alternatives: If any clause is true, then the whole predicate is true.

Logical Programming Paradigm

Fact

A fact must start with a predicate (which is an atom). The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be atoms (in this case, these atoms are treated as constants), numbers, variables or lists.

Facts are axioms; relations between terms that are assumed to be true.

Example facts:

+big('bear')

+big('elephant')

+small('cat')

+brown('bear')

+black('cat')

+grey('elephant')

Consider the 3 fact saying ‘cat’ is a smallest animal and fact 6 saying the elephant is grey in color

Logical Programming Paradigm

Rule

- Rules are theorems that allow new inferences to be made.

•Describe Relationships Using other Relationships.

Example

Two people are sisters if they are both female and have the same parents.

Gives a definition of one relationship given other relationships.

Both must be females.

Both must have the same parents.

If two people satisfy these rules, then they are sisters (according to our simplified relationship)

dark(X)<=black(X)

dark(X)<=brown(X)

Consider rule 1 saying the animal color is black its consider to be dark color animal

Logical Programming Paradigm

Queries

A query is a statement starting with a predicate and followed by its arguments, some of which are variables. Similar to goals, the predicate of a valid query must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the query must be the same as that appears in the consulted program.

```
print(pyDatalog.ask('father_of(X,jess)'))
```

Output:

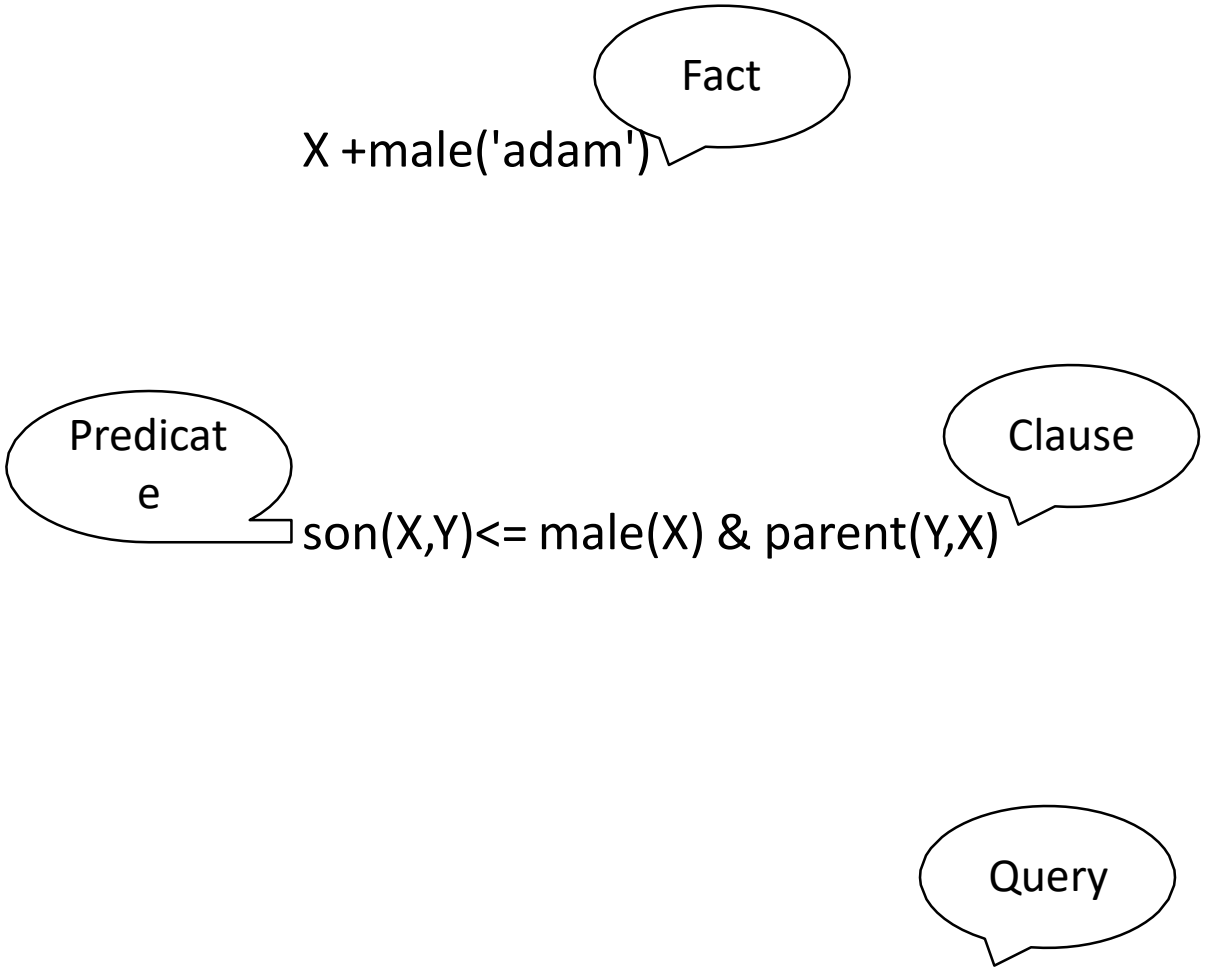
```
{('jack',)}
X
```

```
print(father_of(X,'jess'))
```

Output:

```
jack
X
```

Anatomy Logical Programming Paradigm



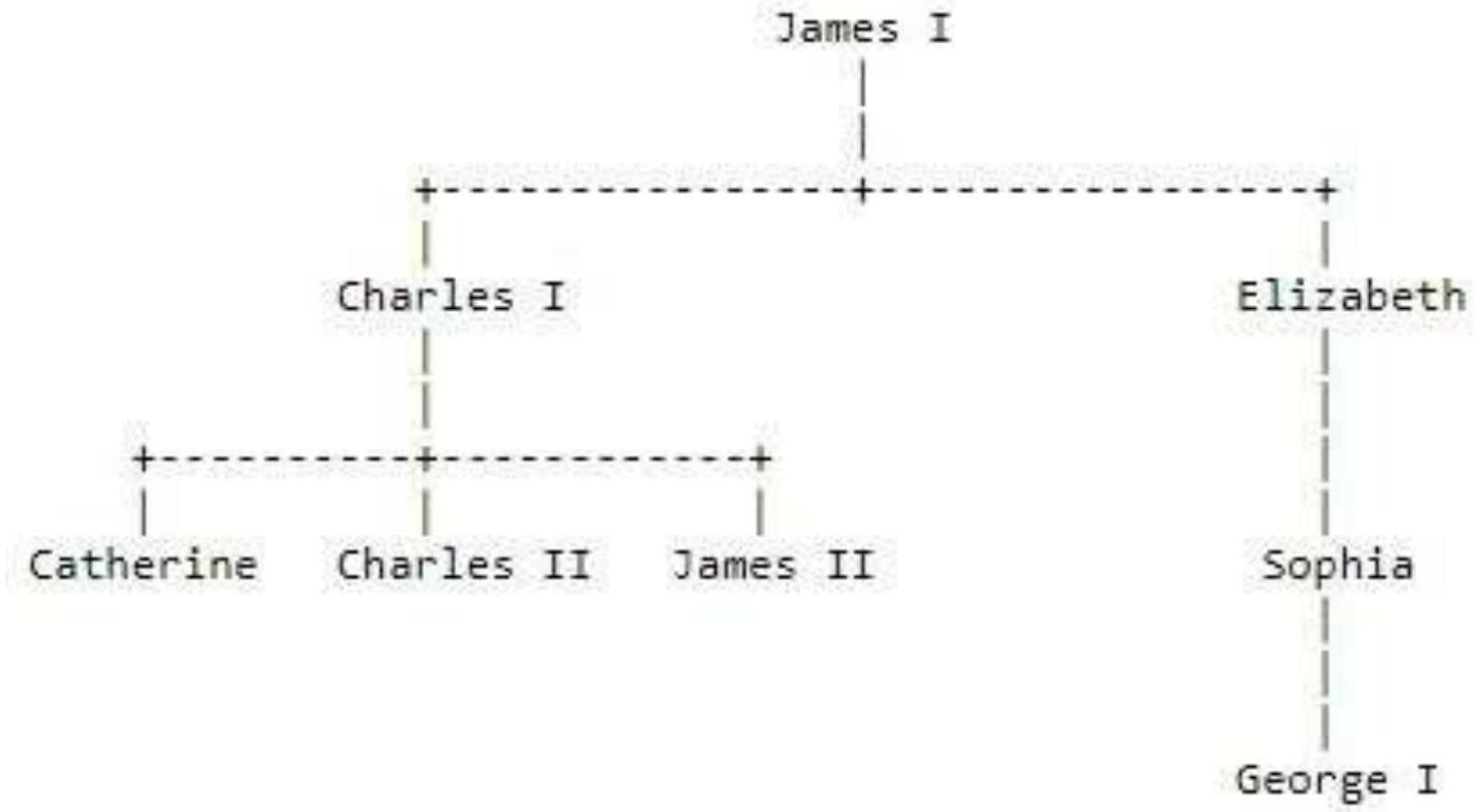
Logical Programming Paradigm

```
from pyDatalog import pyDatalog

pyDatalog.create_atoms('parent,male,female,son,daughter,X,Y,Z')

+male('adam')

+female('anne')

+female('barney')

+male('james')

+parent('barney','adam')

+parent('james','anne')


#The first rule is read as follows: for all X and Y, X is the son of Y if there exists X and Y such that Y is the parent of X and X is male.

#The second rule is read as follows: for all X and Y, X is the daughter of Y if there exists X and Y such that Y is the parent of X and X is female.

son(X,Y)<= male(X) & parent(Y,X)

daughter(X,Y)<= parent(Y,X) & female(X)

print(pyDatalog.ask('son(adam,Y)'))

print(pyDatalog.ask('daughter(anne,Y)'))

print(son('adam',X))
```

Logical Programming Paradigm

```
pyDatalog.create_terms('factorial, N')

factorial[N] = N*factorial[N-1]

factorial[1] = 1

print(factorial[3]==N)
```

Logical Programming Paradigm



Logical Programming Paradigm

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, works_in, department_size, manager, indirect_manager, count_of_indirect_reports') #
Mary works in Production
+works_in('Mary', 'Production')
+works_in('Sam',  'Marketing')
+works_in('John', 'Production')
+works_in('John', 'Marketing')
+(manager['Mary'] == 'John')
+(manager['Sam']  == 'Mary')
+(manager['Tom']  == 'Mary')

indirect_manager(X,Y) <= (manager[X] == Y)
print(works_in(X,  'Marketing'))
indirect_manager(X,Y) <= (manager[X] == Z) & indirect_manager(Z,Y)
print(indirect_manager('Sam',X))
```

Logical Programming Paradigm

Lucy is a Professor

Danny is a Professor

James is a Lecturer

All professors are Dean

Write a Query to reterive all deans?

Soln

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,professor,lecturer, dean')
+professor('lucy')
+professor('danny')
+lecturer('james')
dean(X)<=professor(X)
print(dean(X))
```

Logical Programming Paradigm

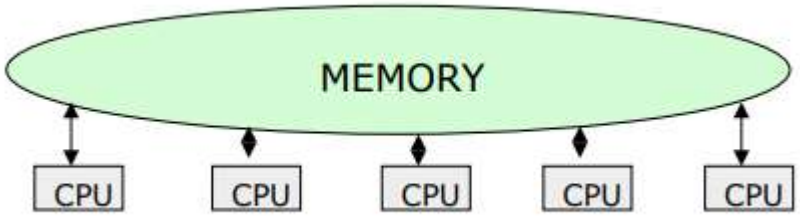
```
likes(john, susie).                 /* John likes Susie */
likes(X, susie).                    /* Everyone likes Susie */
likes(john, Y).                     /* John likes everybody */
likes(john, Y), likes(Y, john).     /* John likes everybody and everybody likes John */
likes(john, susie); likes(john,mary). /* John likes Susie or John likes Mary */
not(likes(john,pizza)).             /* John does not like pizza */
likes(john,susie) :- likes(john,mary)./* John likes Susie if John likes Mary.


rules
friends(X,Y) :- likes(X,Y),likes(Y,X).          /* X and Y are friends if they like each other */
hates(X,Y) :- not(likes(X,Y)).                  /* X hates Y if X does not like Y. */
enemies(X,Y) :- not(likes(X,Y)),not(likes(Y,X)).  /* X and Y are enemies if they don't like each other */
```

Parallel Programming Paradigm

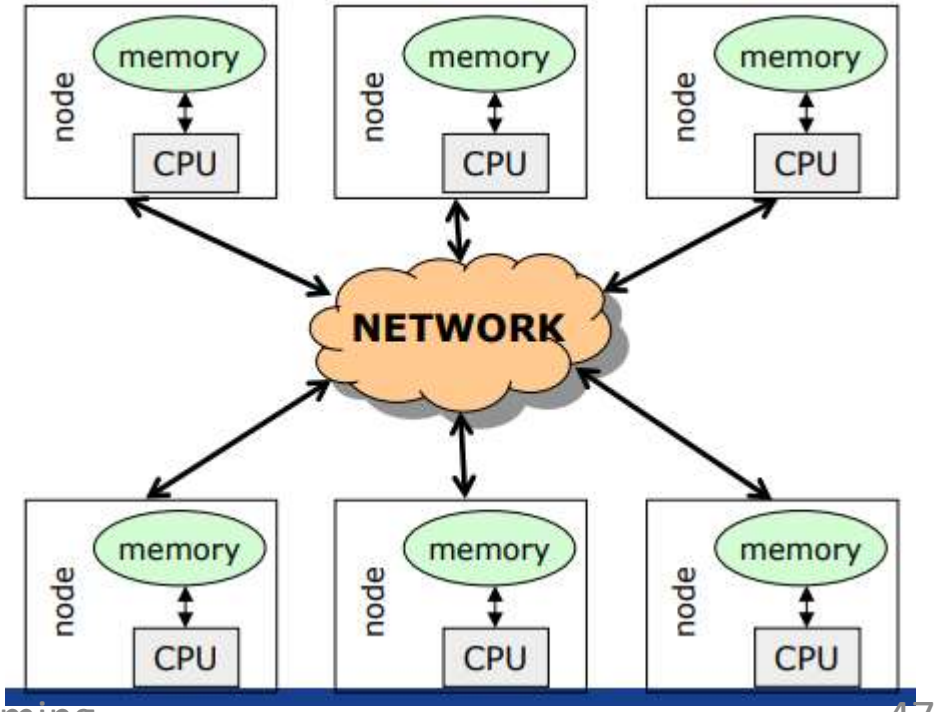
Introduction

- A system is said to be parallel if it can support two or more actions executing simultaneously i.e., multiple actions are simultaneously executed in parallel systems.
- The evolution of parallel processing, even if slow, gave rise to a considerable variety of programming paradigms.
- Parallelism Types:
  - Explicit Parallelism
  - Implicit Parallelism

Shared memory Architecture



Message Passing Architecture



Explicit parallelism

- Explicit Parallelism is characterized by the presence of explicit constructs in the programming language, aimed at describing (to a certain degree of detail) the way in which the parallel computation will take place.
- A wide range of solutions exists within this framework. One extreme is represented by the ``ancient'' use of basic, low level mechanisms to deal with parallelism--like fork/join primitives, semaphores, etc--eventually added to existing programming languages. Although this allows the highest degree of flexibility (any form of parallel control can be implemented in terms of the basic low level primitives gif), it leaves the additional layer of complexity completely on the shoulders of the programmer, making his task extremely complicate.

Implicit Parallelism

- Allows programmers to write their programs without any concern about the exploitation of parallelism. Exploitation of parallelism is instead automatically performed by the compiler and/or the runtime system. In this way the parallelism is transparent to the programmer maintaining the complexity of software development at the same level of standard sequential programming.
- Extracting parallelism implicitly is not an easy task. For imperative programming languages, the complexity of the problem is almost prohibitively and allows positive results only for restricted sets of applications (e.g., applications which perform intensive operations on arrays.
- Declarative Programming languages, and in particular Functional and Logic languages, are characterized by a very high level of abstraction, allowing the programmer to focus on what the problem is and leaving implicit many details of how the problem should be solved.
- Declarative languages have opened new doors to automatic exploitation of parallelism. Their focusing on a high level description of the problem and their mathematical nature turned into positive properties for implicit exploitation of parallelism.

Methods for parallelism

There are many methods of programming parallel computers. Two of the most common are message passing and data parallel.

1. Message Passing - the user makes calls to libraries to explicitly share information between processors.
2. Data Parallel - data partitioning determines parallelism
3. Shared Memory - multiple processes sharing common memory space
4. Remote Memory Operation - set of processes in which a process can access the memory of another process without its participation
5. Threads - a single process having multiple (concurrent) execution paths
6. Combined Models - composed of two or more of the above.

Methods for parallelism

Message Passing:

- Each Processor has direct access only to its local memory
- Processors are connected via high-speed interconnect
- Data exchange is done via explicit processor-to-processor communication i.e processes communicate by sending and receiving messages : send/receive messages
- Data transfer requires cooperative operations to be performed by each process (a send operation must have matching receive)

Data Parallel:

- Each process works on a different part of the same data structure
- Processors have direct access to global memory and I/O through bus or fast switching network
- Each processor also has its own memory (cache)
- Data structures are shared in global address space
- Concurrent access to shared memory must be coordinate
- All message passing is done invisibly to the programmer

Steps in Parallelism

- Independently from the specific paradigm considered, in order to execute a program which exploits parallelism, the programming language must supply the means to:
  - Identify parallelism, by recognizing the components of the program execution that will be (potentially) performed by different processors;
  - Start and stop parallel executions;
  - Coordinate the parallel executions (e.g., specify and implement interactions between concurrent components).

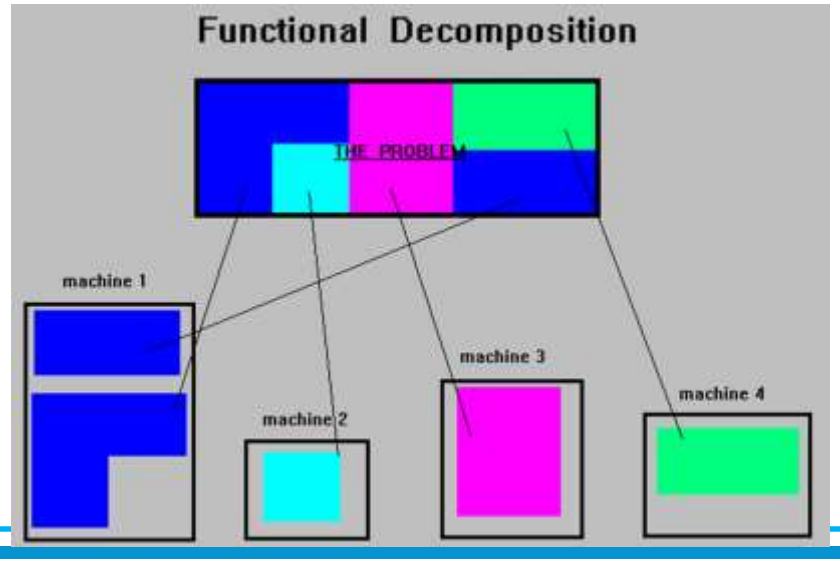
Ways for Parallelism

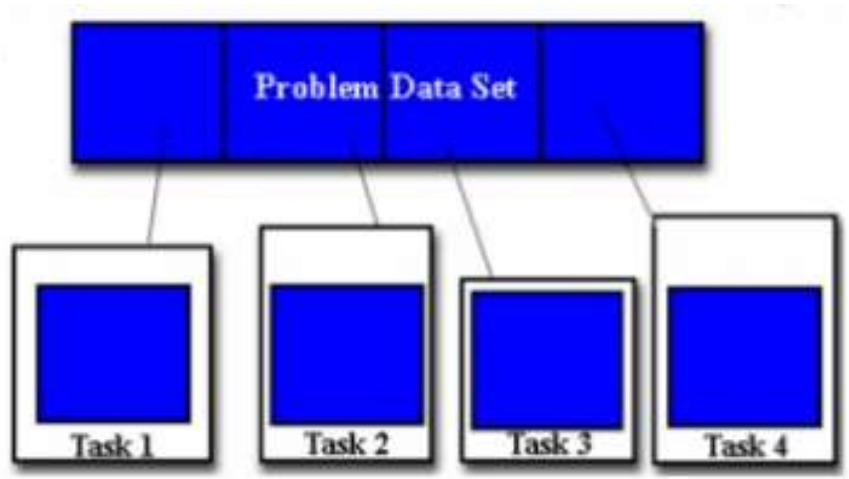
Functional Decomposition (Functional Parallelism)

- Decomposing the problem into different tasks which can be distributed to multiple processors for simultaneous execution
- Good to use when there is not static structure or fixed determination of number of calculations to be performed

Domain Decomposition (Data Parallelism)

- Partitioning the problem's data domain and distributing portions to multiple processors for simultaneous execution
- Good to use for problems where:
- data is static (factoring and solving large matrix or finite difference calculations)
- dynamic data structure tied to single entity where entity can be subsetted (large multi-body problems)
- domain is fixed but computation within various regions of the domain is dynamic (fluid vortices models)





Parallel Programming Paradigm

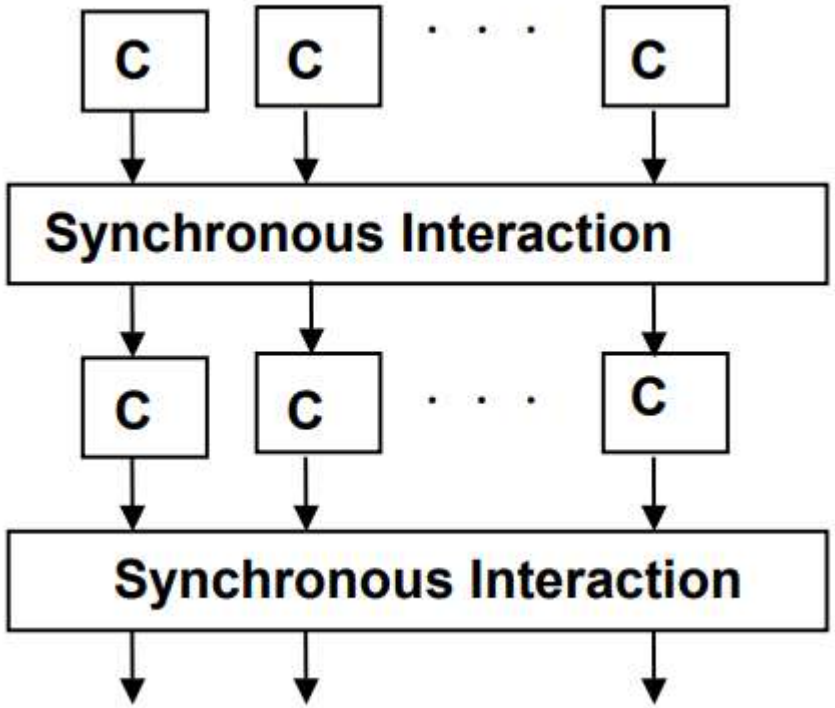
- Phase parallel
- Divide and conquer
- Pipeline
- Process farm
- Work pool

Note:

- The parallel program consists of number of super steps, and each super step has two phases : computation phase and interaction phase

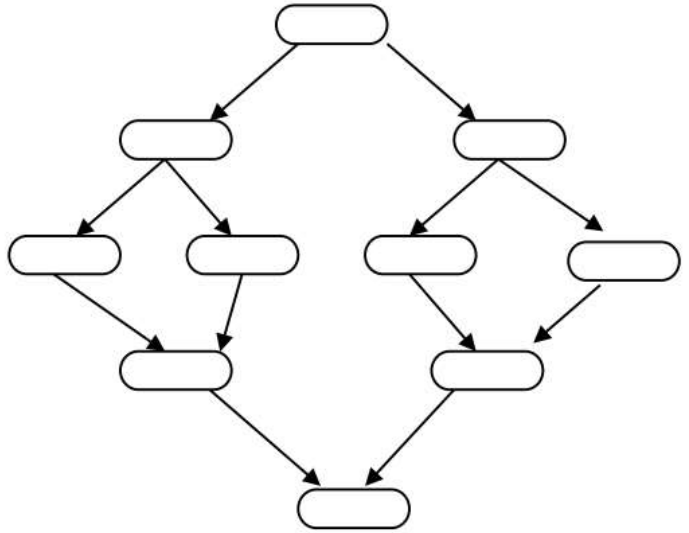
Phase Parallel Model

- The phase-parallel model offers a paradigm that is widely used in parallel programming.
- The parallel program consists of a number of supersteps, and each has two phases.
  - In a computation phase, multiple processes each perform an independent computation C.
  - In the subsequent interaction phase, the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication.
  - Then next superstep is executed.



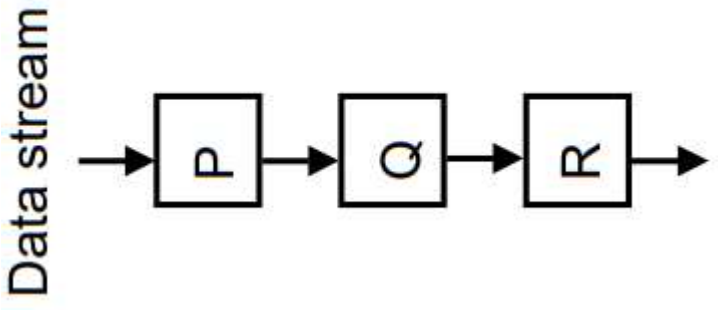
Divide and Conquer & Pipeline model

- A parent process divides its workload into several smaller pieces and assigns them to a number of child processes.
- The child processes then compute their workload in parallel and the results are merged by the parent.
- The dividing and the merging procedures are done recursively.
- This paradigm is very natural for computations such as quick sort.



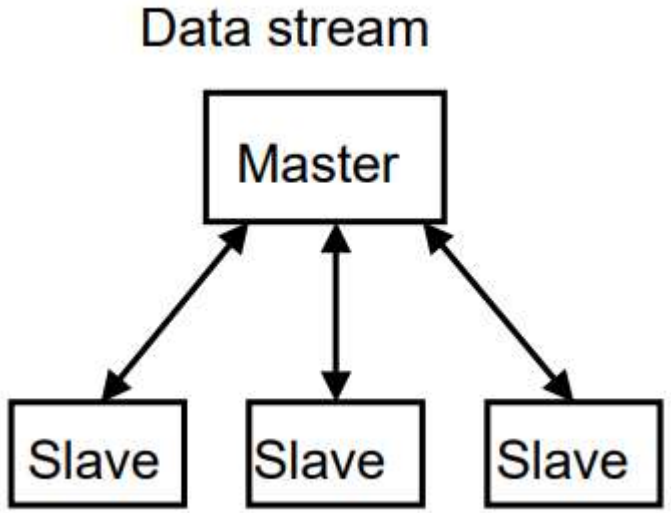
Pipeline

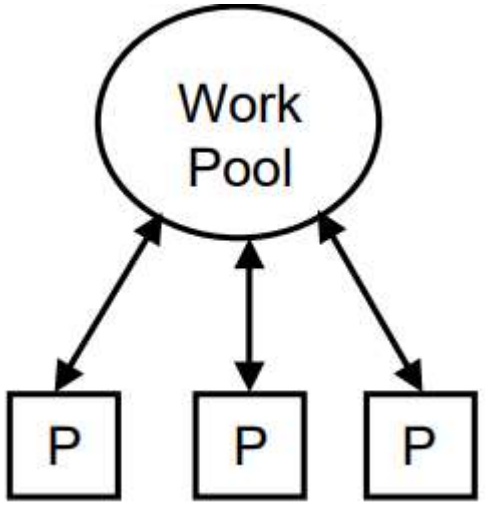
- In pipeline paradigm, a number of processes form a virtual pipeline.
- A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion.



Process Farm & Work Pool Model

- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- This is a very simple paradigm, where the coordination is done by the master.
- This paradigm is often used in a shared variable model.
- A pool of works is realized in a global data structure.
- A number of processes are created. Initially, there may be just one piece of work in the pool.
- Any free process fetches a piece of work from the pool and executes it, producing zero, one, or more new work pieces put into the pool. The parallel program ends when the work pool becomes empty.
- This paradigm facilitates load balancing, as the workload is dynamically allocated to free processes.





Parallel Program using Python

- A thread is basically an independent flow of execution. A single process can consist of multiple threads. Each thread in a program performs a particular task. For Example, when you are playing a game say FIFA on your PC, the game as a whole is a single process, but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc.
- Threading is that it allows a user to run different parts of the program in a concurrent manner and make the design of the program simpler.
- Multithreading in Python can be achieved by importing the threading module.

Example:

```
import threading
from threading import *
```

Parallel program using Threads in Python

```
# simplest way to use a Thread is to instantiate it with a target function and call start() to let it begin working.
from threading import Thread,current_thread
print(current_thread().getName())
def mt():
    print("Child Thread")
    for i in range(11,20):
        print(i*2)
def disp():
   for i in range(10):
        print(i*2)
child=Thread(target=mt)
child.start()
disp()
print("Executing thread name :",current_thread().getName())
```

```
from threading import Thread,current_thread
class mythread(Thread):
    def run(self):
        for x in range(7):
            print("Hi from child")
a = mythread()
a.start()
a.join()
print("Bye from",current_thread().getName())
```

Parallel program using Process in Python

```
import multiprocessing
def worker(num):
    print('Worker:', num)
    for i in range(num):
        print(i)
    return

jobs = []
for i in range(1,5):
    p = multiprocessing.Process(target=worker, args=(i+10,))
    jobs.append(p)
    p.start()
```

Concurrent Programming Paradigm

- Computing systems model the world, and the world contains actors that execute independently of, but communicate with, each other. In modelling the world, many (possibly) parallel executions have to be composed and coordinated, and that's where the study of concurrency comes in.
- There are two common models for concurrent programming: shared memory and message passing.
  - **Shared memory.** In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.
  - **Message passing.** In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling

Issues Concurrent Programming Paradigm

Concurrent programming is programming with multiple tasks. The major issues of concurrent programming are:

- Sharing computational resources between the tasks;
- Interaction of the tasks.

Objects shared by multiple tasks have to be safe for concurrent access. Such objects are called protected. Tasks accessing such an object interact with each other indirectly through the object.

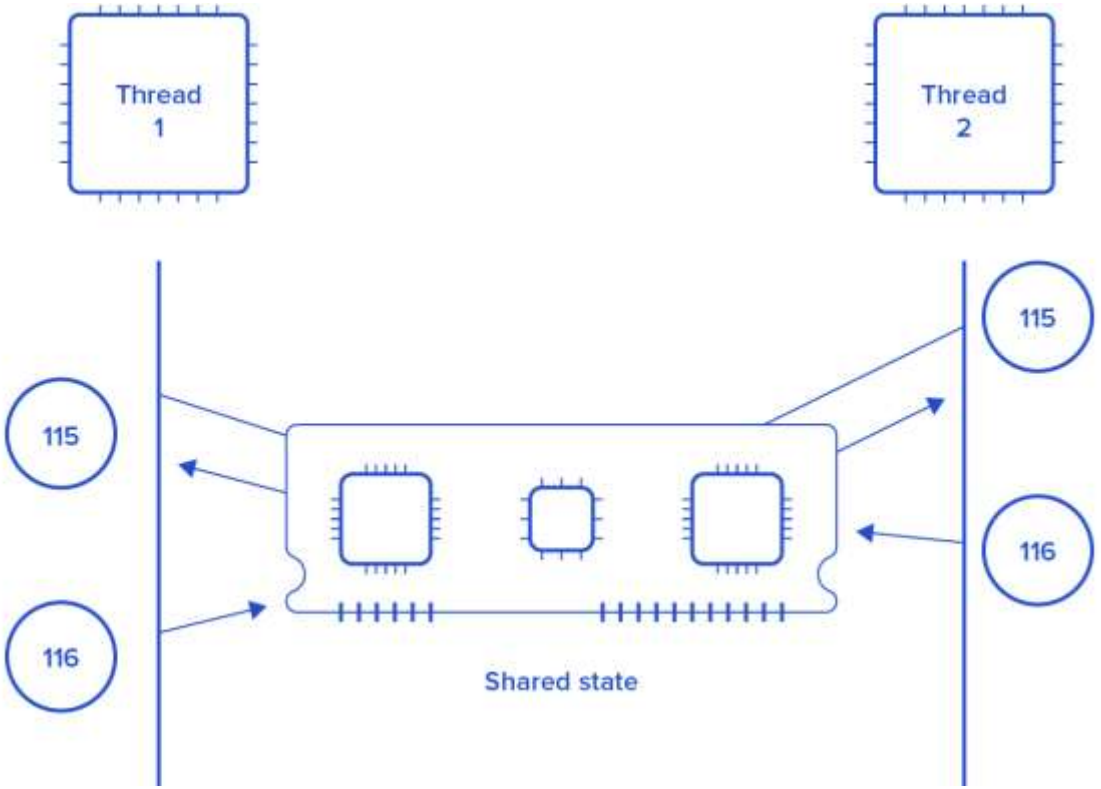
An access to the protected object can be:

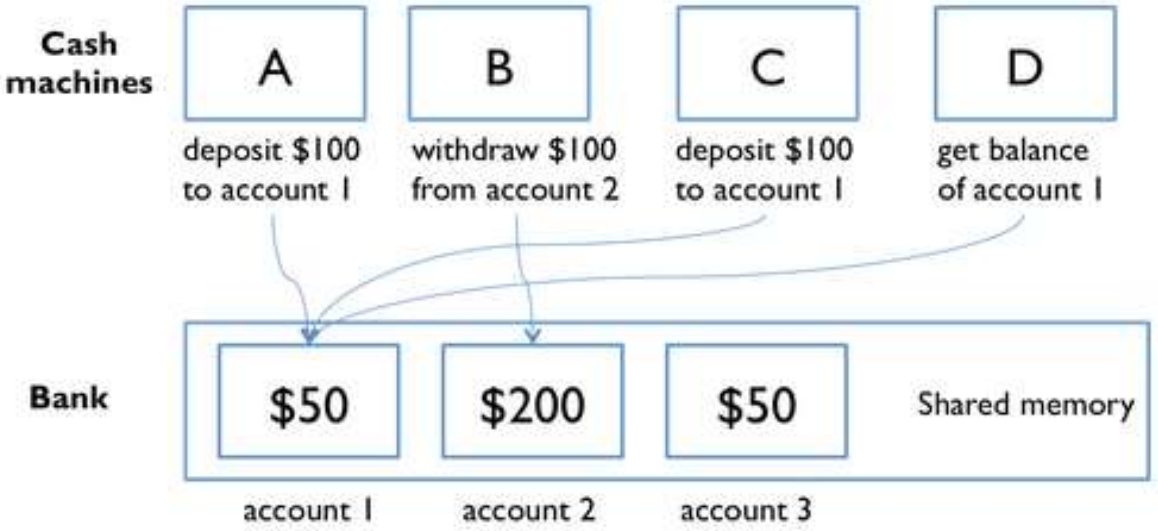
- Lock-free, when the task accessing the object is not blocked for a considerable time;
- Blocking, otherwise.

Blocking objects can be used for task synchronization. To the examples of such objects belong:

- Events;
- Mutexes and semaphores;
- Waitable timers;
- Queues

Issues Concurrent Programming Paradigm





Race Condition

```
import threading
x = 0     # A shared value
COUNT = 100

def incr():
    global x
    for i in range(COUNT):
        x += 1
        print(x)

def decr():
    global x
    for i in range(COUNT):
        x -= 1
        print(x)
```

```
t1 = threading.Thread(target=incr)
t2 = threading.Thread(target=decr)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
```

Synchronization in Python

Locks:

Locks are perhaps the simplest synchronization primitives in Python. A Lock has only two states — locked and unlocked (surprise). It is created in the unlocked state and has two principal methods — acquire() and release(). The acquire() method locks the Lock and blocks execution until the release() method in some other co-routine sets it to unlocked.

R-Locks:

R-Lock class is a version of simple locking that only blocks if the lock is held by another thread. While simple locks will block if the same thread attempts to acquire the same lock twice, a re-entrant lock only blocks if another thread currently holds the lock.

Semaphore:

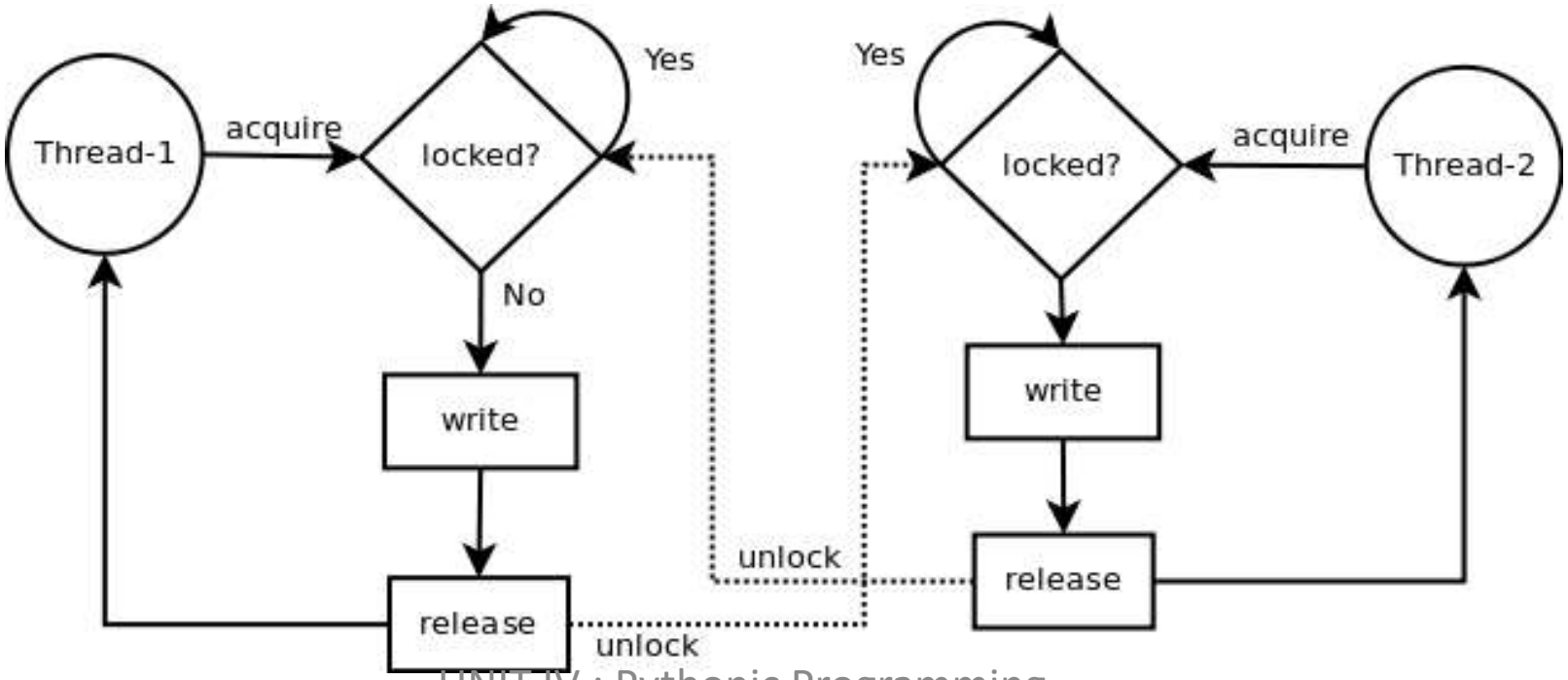
A semaphore has an internal counter rather than a lock flag, and it only blocks if more than a given number of threads have attempted to hold the semaphore. Depending on how the semaphore is initialized, this allows multiple threads to access the same code section simultaneously.

LOCK in python

Synchronization using LOCK

Locks have 2 states: locked and unlocked. 2 methods are used to manipulate them: acquire() and release(). Those are the rules:

1. if the state is unlocked: a call to acquire() changes the state to locked.
2. if the state is locked: a call to acquire() blocks until another thread calls release().
3. if the state is unlocked: a call to release() raises a RuntimeError exception.
4. if the state is locked: a call to release() changes the state to unlocked().



Synchronization in Python using Lock

```
import threading
x = 0     # A shared value
COUNT = 100
lock = threading.Lock()


def incr():
    global x
    lock.acquire()
    print("thread locked for increment cur x=",x)
    for i in range(COUNT):
        x += 1
        print(x)
    lock.release()
    print("thread release from increment cur x=",x)
def decr():
    global x
    lock.acquire()
    print("thread locked for decrement cur x=",x)
    for i in range(COUNT):
        x -= 1
        print(x)
    lock.release()
    print("thread release from decrement cur x=",x)
t1 = threading.Thread(target=incr)
t2 = threading.Thread(target=decr)
t1.start()
t2.start()
t1.join()
t2.join()
```

Synchronization in Python using RLock

```
import threading

class Foo(object):
    lock = threading.RLock()
    def __init__(self):
        self.x = 0
    def add(self,n):
        with Foo.lock:
            self.x += n
    def incr(self):
        with Foo.lock:
            self.add(1)
    def decr(self):
        with Foo.lock:
            self.add(-1)


def adder(f,count):
    while count > 0:
        f.incr()
        count -= 1

def subber(f,count):
    while count > 0:
        f.decr()
        count -= 1

# Create some threads and make sure it works
COUNT = 10
f = Foo()
t1 = threading.Thread(target=adder,args=(f,COUNT))
t2 = threading.Thread(target=subber,args=(f,COUNT))
t1.start()
t2.start()
t1.join()
t2.join()
print(f.x)
```

Synchronization in Python using Semaphore

```
import threading
import time


done = threading.Semaphore(0)
item = None


def producer():
    global item
    print "I'm the producer and I produce data."
    print "Producer is going to sleep."
    time.sleep(10)
    item = "Hello"
    print "Producer is alive. Signaling the consumer."
    done.release()
```

```
def consumer():
    print "I'm a consumer and I wait for data."
    print "Consumer is waiting."
    done.acquire()
    print "Consumer got", item


t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)
t1.start()
t2.start()
```

Synchronization in Python using event

```
import threading
import time
item = None
# A semaphore to indicate that an item is available
available = threading.Semaphore(0)
# An event to indicate that processing is complete
completed = threading.Event()
# A worker thread
def worker():
    while True:
        available.acquire()
        print "worker: processing", item
        time.sleep(5)
        print "worker: done"
        completed.set()
# A producer thread
def producer():
    global item
    for x in range(5):
        completed.clear()      # Clear the event
        item = x               # Set the item
        print "producer: produced an item"
        available.release()    # Signal on the semaphore
        completed.wait()
        print "producer: item was processed"
t1 = threading.Thread(target=producer)
t1.start()
t2 = threading.Thread(target=worker)
t2.setDaemon(True)
t2.start()
```

Producer and Consumer problem using thread

```
import threading,time,Queue
items = Queue.Queue()
# A producer thread
def producer():
    print "I'm the producer"
    for i in range(30):
        items.put(i)
        time.sleep(1)
# A consumer thread
def consumer():
    print "I'm a consumer", threading.currentThread().name
    while True:
        x = items.get()
        print threading.currentThread().name,"got", x
        time.sleep(5)
# Launch a bunch of consumers
cons = [threading.Thread(target=consumer)
        for i in range(10)]
for c in cons:
    c.setDaemon(True)
    c.start()
# Run the producer
producer()
```

Producer and Consumer problem using thread

```
import threading
import time
# A list of items that are being produced.  Note: it is actually
# more efficient to use a collections.deque() object for this.
items = []
# A condition variable for items
items_cv = threading.Condition()
def producer():
   print "I'm the producer"
   for i in range(30):
      with items_cv:         # Always must acquire the lock first
         items.append(i)     # Add an item to the list
         items_cv.notify()   # Send a notification signal
      time.sleep(1)
def consumer():
   print "I'm a consumer", threading.currentThread().name
   while True:
      with items_cv:          # Must always acquire the lock
         while not items:     # Check if there are any items
            items_cv.wait()  # If not, we have to sleep
         x = items.pop(0)     # Pop an item off
      print threading.currentThread().name,"got", x
      time.sleep(5)
cons = [threading.Thread(target=consumer)
      for i in range(10)]
for c in cons:
   c.setDaemon(True)
   c.start()
producer()
```

Network Programming Paradigm

Introduction

The Network paradigm involves thinking of computing in terms of a client, who is essentially in need of some type of information, and a server, who has lots of information and is just waiting to hand it out. Typically, a client will connect to a server and query for certain information. The server will go off and find the information and then return it to the client.

In the context of the Internet, clients are typically run on desktop or laptop computers attached to the Internet looking for information, whereas servers are typically run on larger computers with certain types of information available for the clients to retrieve. The Web itself is made up of a bunch of computers that act as Web servers; they have vast amounts of HTML pages and related data available for people to retrieve and browse. Web clients are used by those of us who connect to the Web servers and browse through the Web pages.

Network programming uses a particular type of network communication known as sockets. A socket is a software abstraction for an input or output medium of communication.

What is Socket?

- A socket is a software abstraction for an input or output medium of communication.
- Sockets allow communication between processes that lie on the same machine, or on different machines working in diverse environment and even across different continents.
- A socket is the most vital and fundamental entity. Sockets are the end-point of a two-way communication link.
- An endpoint is a combination of IP address and the port number.

For Client-Server communication,

  - Sockets are to be configured at the two ends to initiate a connection,
  - Listen for incoming messages
  - Send the responses at both ends
  - Establishing a bidirectional communication.

Socket Types

Datagram Socket

- A datagram is an independent, self-contained piece of information sent over a network whose arrival, arrival time, and content are not guaranteed. A datagram socket uses User Datagram Protocol (UDP) to facilitate the sending of datagrams (self-contained pieces of information) in an unreliable manner. Unreliable means that information sent via datagrams isn't guaranteed to make it to its destination.

Stream Socket:

- A stream socket, or connected socket, is a socket through which data can be transmitted continuously. A stream socket is more akin to a live network, in which the communication link is continuously active. A stream socket is a "connected" socket through which data is transferred continuously.

Socket in Python

sock_obj = socket.socket(socket_family, socket_type, protocol=0)

socket_family: - Defines family of protocols used as transport mechanism.

Either AF_UNIX, or

AF_INET (IP version 4 or IPv4).

socket_type: Defines the types of communication between the two end-points.

SOCK_STREAM (for connection-oriented protocols, e.g., TCP), or

SOCK_DGRAM (for connectionless protocols e.g. UDP).

protocol: We typically leave this field or set this field to zero.

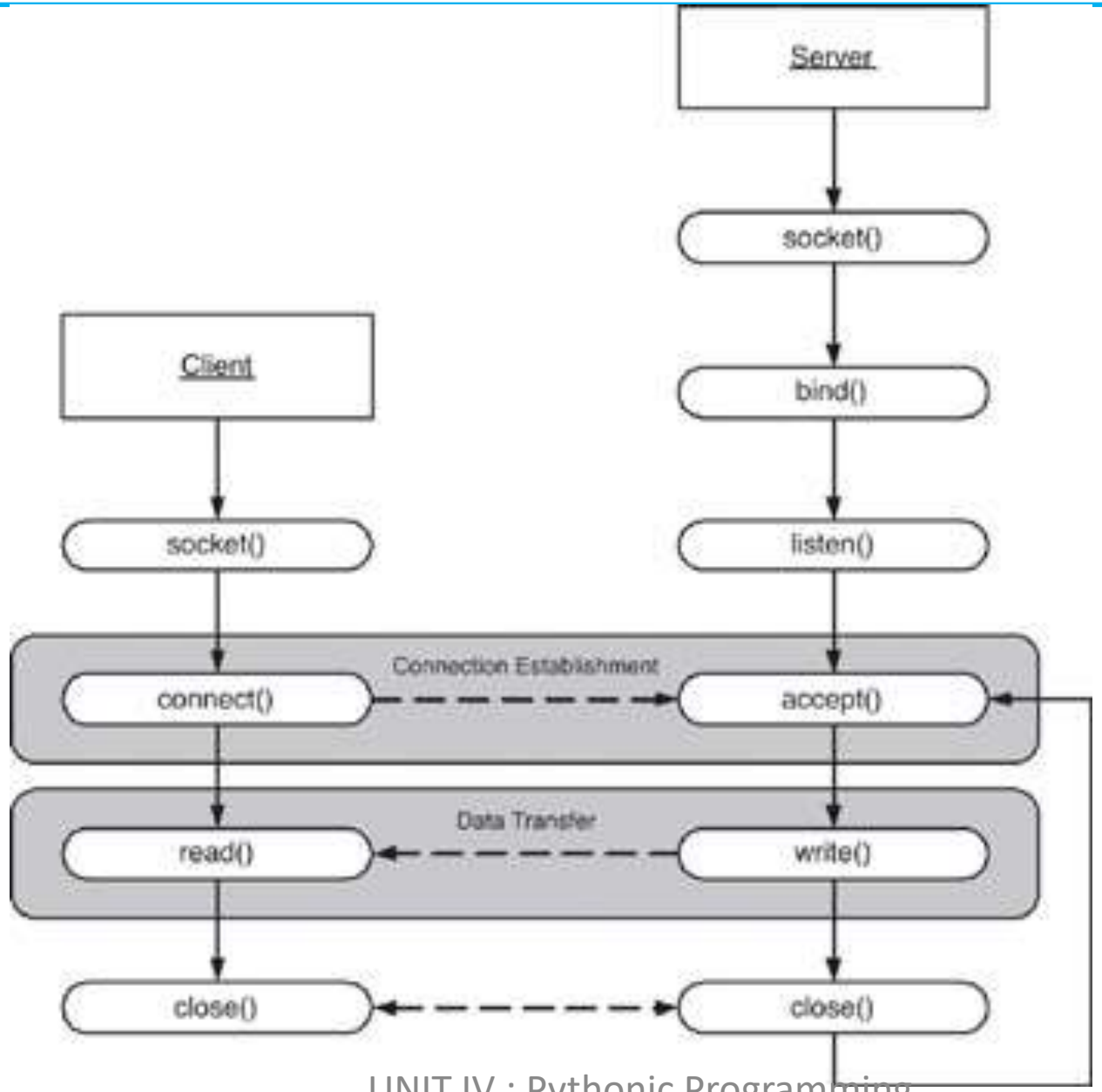
Example:

```
#Socket client example in python
import socket
#create an AF_INET, STREAM socket (TCP)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket Created'
```

Socket Creation

```
import socket
import sys
try:
        #create an AF_INET, STREAM socket (TCP)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, msg:
        print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
        sys.exit();

print 'Socket Created'
```

Client/server symmetry in Sockets applications



Socket in Python

To create a socket, we must use socket.socket() function available in the Python socket module, which has the general syntax as follows:

S = socket.socket(socket_family, socket_type, protocol=0)

socket_family: This is either AF_UNIX or AF_INET. We are only going to talk about INET sockets in this tutorial, as they account for at least 99% of the sockets in use.

socket_type: This is either SOCK_STREAM or SOCK_DGRAM.

Protocol: This is usually left out, defaulting to 0.

Client Socket Methods

Following are some client socket methods:

connect( ) : To connect to a remote socket at an address. An address format(host, port) pair is used for AF_INET address family.

Socket in Python

Server Socket Methods

bind( ): This method binds the socket to an address. The format of address depends on socket family mentioned above(AF_INET).

listen(backlog) : This method listens for the connection made to the socket. The backlog is the maximum number of queued connections that must be listened before rejecting the connection.

accept( ) : This method is used to accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair(conn, address) where conn is a new socket object which can be used to send and receive data on that connection, and address is the address bound to the socket on the other end of the connection.

General Socket in Python

sock_object.recv():

Use this method to receive messages at endpoints when the value of the protocol parameter is TCP.

sock_object.send():

Apply this method to send messages from endpoints in case the protocol is TCP.

sock_object.recvfrom():

Call this method to receive messages at endpoints if the protocol used is UDP.

sock_object.sendto():

Invoke this method to send messages from endpoints if the protocol parameter is UDP.

sock_object.gethostname():

This method returns hostname.

sock_object.close():

This method is used to close the socket. The remote endpoint will not receive data from this side.

Simple TCP Server

```
#!/usr/bin/python

#This is tcp_server.py script

import socket                           #line 1: Import socket module

s = socket.socket()                     #line 2: create a socket object
host = socket.gethostname()             #line 3: Get current machine name
port = 9999                             #line 4: Get port number for connection

s.bind((host,port))                     #line 5: bind with the address

print "Waiting for connection..."
s.listen(5)                             #line 6: listen for connections

while True:
    conn,addr = s.accept()              #line 7: connect and accept from client
    print 'Got Connection from', addr
    conn.send('Server Saying Hi')
    conn.close()                        #line 8: Close the connection
```

Simple TCP Client

```
#!/usr/bin/python

#This is tcp_client.py script

import socket

s = socket.socket()
host = socket.gethostname()          # Get current machine name
port = 9999                          # Client wants to connect to server's
                                     # port number 9999

s.connect((host,port))
print s.recv(1024)                   # 1024 is bufsize or max amount
                                     # of data to be received at once

s.close()
```

Simple UDP Server

```
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()                 # Host IP
udp_port = 12345                                # specified port to connect

#print type(sock) ============> 'type' can be used to see type
                # of any variable ('sock' here)

sock.bind((udp_host,udp_port))

while True:
    print "Waiting for client..."
    data,addr = sock.recvfrom(1024)          #receive data from client
    print "Received Messages:",data," from",addr
```

Simple UDP Client

```
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()      # Host IP
udp_port = 12345                     # specified port to connect

msg = "Hello Python!"
print "UDP target IP:", udp_host
print "UDP target Port:", udp_port

sock.sendto(msg,(udp_host,udp_port))        # Sending message to UDP server
```