

SRM Institute of Science and Technology
School of Computing

Advanced Programming Practice-18CSC207J

Structured Programming

Structured Programming Paradigm

Unit-I (15 Session)

Session 1-5 cover the following topics:-

- Structured Programming Paradigm
 - Programming Language Theorem
 - Böhm-Jacopini theorem
 - Sequence, Selection, Decision, Iteration and Recursion
 - Other Languages : C/C++/Java /C# /Ruby
 - Demo: Structured Programming in Python
 - Lab 1: Structured Programming
-
- **TextBook:** Shalom, Elad. A Review of Programming Paradigms Throughout the History: With a Suggestion Toward a Future Approach, Kindle Edition

Structured Programming

- Structured programming is a paradigm that is based on improving **clarity and quality of programs** by using subroutines, block structures and loops (for and while) **and discouraging the use of goto statement.**
1. History
 2. Overview
 3. Component
 4. Representations in different languages(C,C++,Java,python)

1. History

The Böhm-Jacopini theorem, also called structured program theorem, stated that working out a function is possible by combining subprograms in only three manners:

- Executing one subprogram, and the other subprogram (sequence) .
- Executing one of two subprograms according to the value of a Boolean expression (selection) .
- Executing a subprogram until a Boolean expression is true (iteration)
- Some of the languages that initially used structured approach are ALGOL, Pascal, PL/I and Ada.
- By the end of the 20th century, concepts of structured programming were widely applied so programming languages that originally lacked structure now have it (FORTRAN, COBOL and BASIC). Now, it is possible to do structured programming in any programming language (Java, C++, Python ...).

2. Overview

Structured programming was defined as a method used to minimize complexity that uses:

1. Top-down analysis for problem solving :-

Top-down analysis includes solving the problem and providing instructions for every step. When developing a solution is complicated, the right approach is to divide a large problem into several smaller problems and tasks.

2. Modularization for program structure and organization :

Modular programming is a method of organizing the instructions of a program. Large programs are divided into smaller sections called modules, subroutines, or subprograms. Each subroutine is in charge of a specific job.

3. Structured code for the individual modules:

Structured coding relates to division of modules into set of instructions organized within control structures. A control structure defines the order in which a set of instructions are executed. The statements within a specific control structure are executed:

- **sequentially** – denotes in which order the controls are executed. They are executed one after the other in the exact order they are listed in the code.
- **conditionally** – allows choosing which set of controls is executed. Based on the needed condition, one set of commands is chosen while others aren't executed.
- **repetitively** – allows the same controls to be performed over and over again. Repetition stops when it meets certain terms or performs a defined number of iterations.

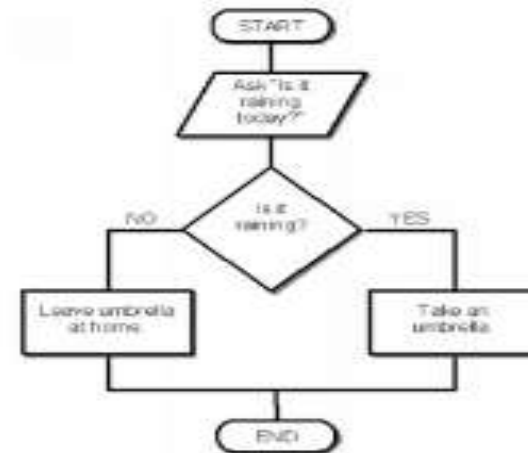
3.Component

- 3.1 **Structograms** - graphical representation of structured programming(Flowchart).
- 3.2 **Subroutine** - Subroutine is a sequence of program instructions packaged as a unit so that together they perform a specific task.
- 3.3 **Block**- Block is a section of code grouped together and it consists of one or more declarations and statements.(ex: {})
- 3.4 **Indentation**- Another typical characteristic of structured programming is indent style applied to a block to display program structure. In most programming languages, indentation is not a requirement **but when used, code is easier to read and follow.**
- 3.5. **Control structure** – sequence, selection and iteration

Component

- Structograms or Nassi–Shneiderman -graphical representation of structured programming.
- Structograms can be compared to flowcharts.
- Nassi–Shneiderman diagrams have no representation for a goto statement.

Ex:-



- **Structograms use the following diagrams:**

1. process blocks - Process blocks represent the simplest actions and don't require analysis. Actions are performed block by block.

Standard Process Block:

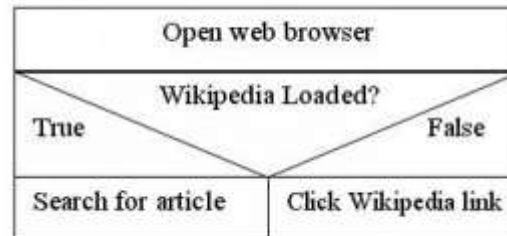


Example|



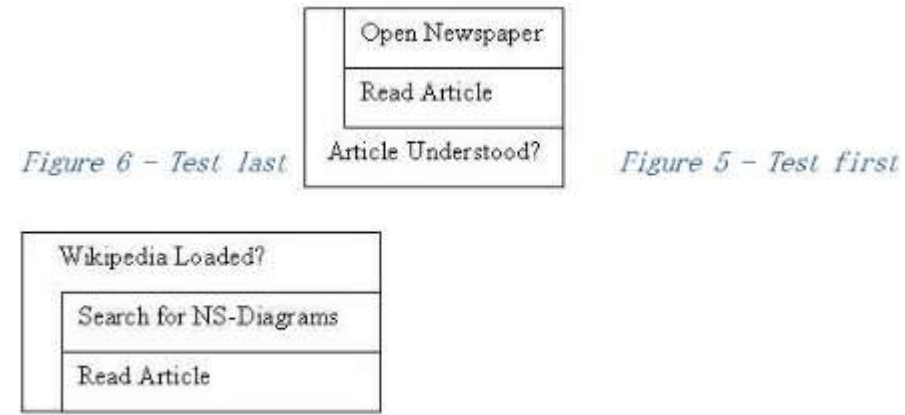
2.branching blocks

Branching blocks are of two types – True/False or Yes/No block and multiple branching block.



3. Testing loops

Testing loops allow the program to repeat one or many processes until a condition is fulfilled. There are two types of testing loops – test first and test last blocks – and the order in which the steps are performed is what makes them different.



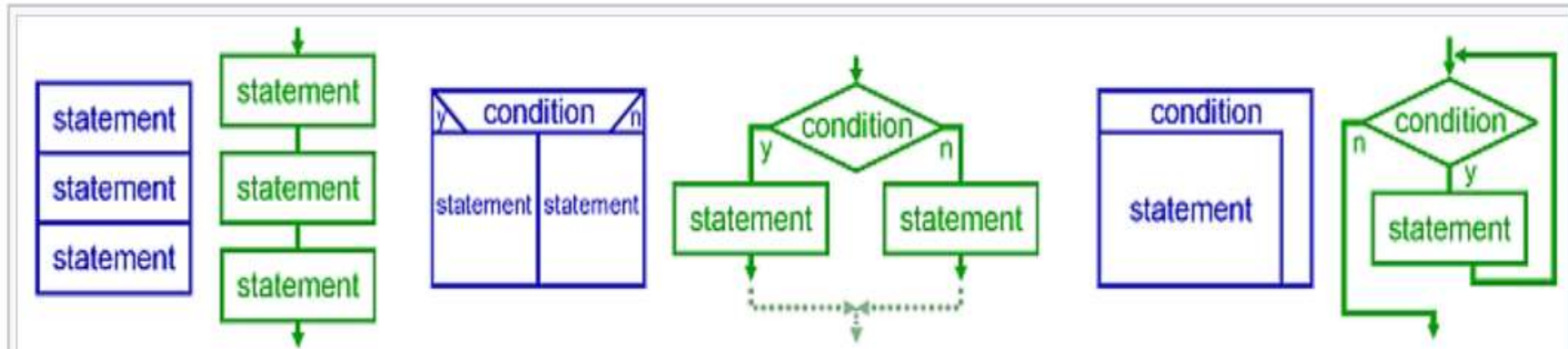
Advantages of structured programming are:

- Programs are more easily and more quickly written.
- Programs have greater reliability.
- Programs require less time to debug and test.
- Programs are easier to maintain.

Control structure – sequence, selection ,iteration and recursion.
(example for Control structure)

Recursion:

Recursion"; a statement is executed by repeatedly calling itself until termination conditions are met. While similar in practice to iterative loops, recursive loops may be more computationally efficient, and are implemented differently as a cascading stack.



Graphical representation of the three basic patterns — sequence, selection, and repetition

Control Structure - DECISION MAKING (PYTHON)

- Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

if statement :

It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

if condition:

 # Statements to execute if

 # condition is true

Example :

```
i = 10
```

```
if (i > 15):
```

```
    print ("10 is less than 15")
```

```
print ("I am Not in if")
```

- **if..else statements:**

We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

if (condition):

 # Executes this block if

 # condition is true

else:

 # Executes this block if

 # condition is false

Example :

i = 20;

if (i < 15):

 print ("i is smaller than 15")

 print ("i'm in if Block")

else:

 print ("i is greater than 15")

 print ("i'm in else Block")

 print ("i'm not in if and not in else Block")

Output:

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

- **nested if statements**

Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1):
```

```
    # Executes when condition1 is true
```

```
    if (condition2):
```

```
        # Executes when condition2 is true
```

```
    # if Block is end here
```

```
# if Block is end here
```


Example : Nested if else

```
i = 10
if (i == 10):
    # First if statement
    if (i < 15):
        print ("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if (i < 12):
        print ("i is smaller than 12 too")
    else:
        print ("i is greater than 15")
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

if-elif-else ladder

- Here, a user can decide **among multiple options**. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then Syntax:-

Syntax:-

if (condition):

 statement

elif (condition):

 statement

else:

- statement the final else statement will be executed.

example

```
i = 20
```

```
if (i == 10):
```

```
    print ("i is 10")
```

```
elif (i == 15):
```

```
    print ("i is 15")
```

```
elif (i == 20):
```

```
    print ("i is 20")
```

```
else:
```

```
    print ("i is not present")
```

CONDITION	SYNTAX
SIMPLE IF	if test expression: statement(s)
IF....ELSE	if test expression: Body of if else: Body of else
IF...ELIF...ELSE	if test expression: Body of if elif test expression: Body of elif else: Body of else
NESTED IF	if test expression: if test expression: Body of if else: Body of else else: Body of else

Conditional Expression

A conditional expression evaluates an expression based on a condition.

Conditional expression is expressed using **if** and **else** combined with expression

Syntax:

expression if Boolean-expression else expression

Example:

Biggest of two numbers

num1 = 23

num2 = 15

big = num1 **if** num1 > num2 **else** num2

print (“ the biggest number is “ , big)

Even or odd

print (“ num is even “ if num % 2 == 0 else “ num is odd “)

Iteration – Loops

- Python has two primitive loop commands:
- while loops
- for loops

The while Loop

- With the while loop we can execute a set of statements as long as a condition is true

Example

- Print i as long as i is less than 6:
- `i = 1`
 `while i < 6:`
 `print(i)`
 `i += 1`

```
C:\Users\My Name>python demo_while.py
1
2
3
4
5
```

Note: remember to increment i, or else the loop will continue forever.

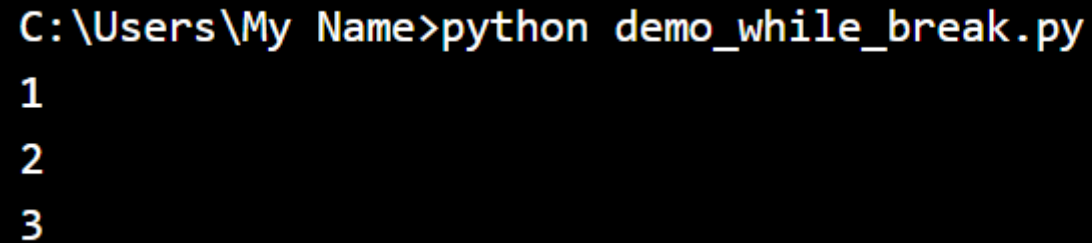
The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```



```
C:\Users\My Name>python demo_while_break.py
1
2
3
```

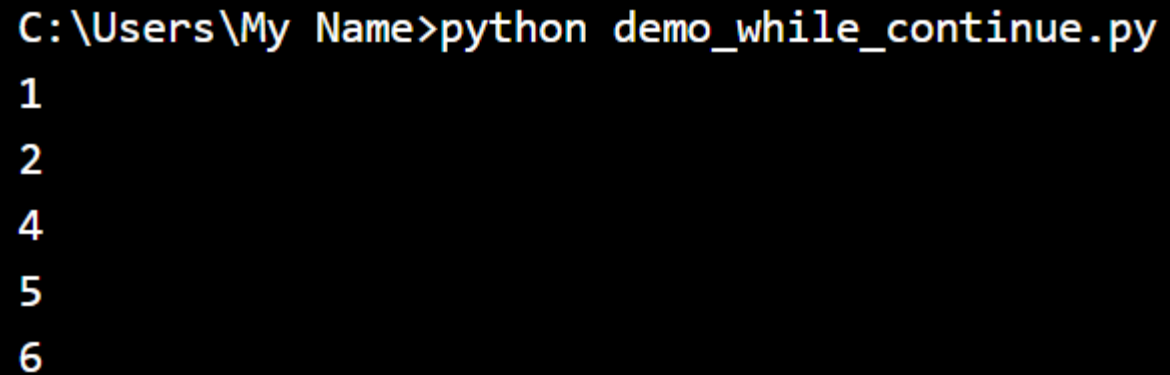

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```



```
C:\Users\My Name>python demo_while_continue.py
1
2
4
5
6
```

The else Statement

- With the else statement we can run a block of code once when the condition no longer is true:
- Example
- Print a message once the condition is false:
- ```
i = 1
while i < 6:
 print(i)
 i += 1
else:
 print("i is no longer less than 6")
```

```
C:\Users\My Name>python demo_while_else.py
1
2
3
4
5
i is no longer less than 6
```

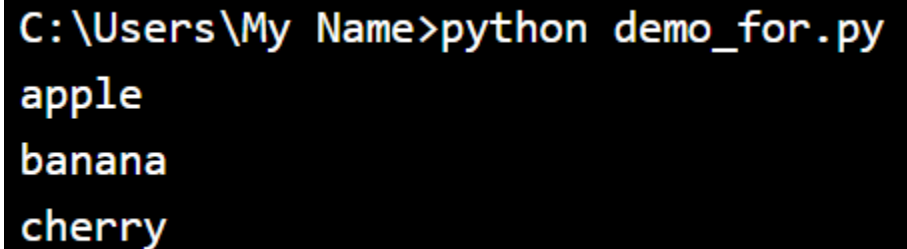
# For Loops

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

## Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
 print(x)
```

A terminal window with a black background and yellow text. The prompt is 'C:\Users\My Name>' followed by the command 'python demo\_for.py'. The output consists of three lines: 'apple', 'banana', and 'cherry'.

```
C:\Users\My Name>python demo_for.py
apple
banana
cherry
```

The for loop does not require an indexing variable to set beforehand

## Looping Through a String

- Even strings are iterable objects, they contain a sequence of characters:

### Example

- Loop through the letters in the word "banana":
- ```
for x in "banana":  
    print(x)
```

```
C:\Users\My Name>python demo_for_string.py  
b  
a  
n  
a  
n  
a
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when x is "banana":

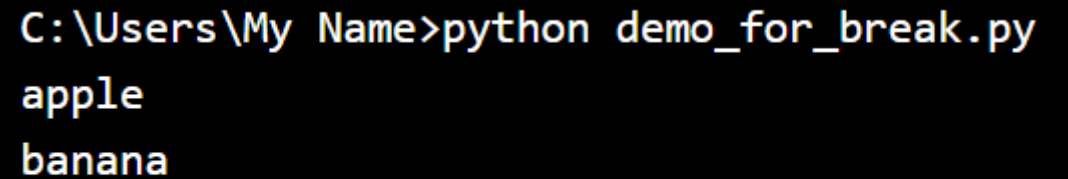
```
fruits = ["apple", "banana", "cherry"]
```

```
for x in fruits:
```

```
    print(x)
```

```
    if x == "banana":
```

```
        break
```



```
C:\Users\My Name>python demo_for_break.py  
apple  
banana
```

- **Example**

Exit the loop when x is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]
```

```
for x in fruits:
```

```
    if x == "banana":
```

```
        break
```

```
    print(x)
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

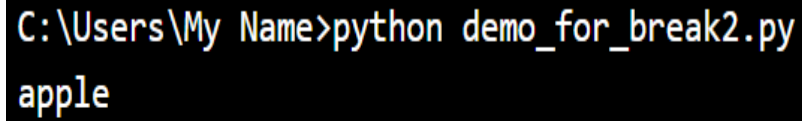
```
fruits = ["apple", "banana", "cherry"]
```

```
for x in fruits:
```

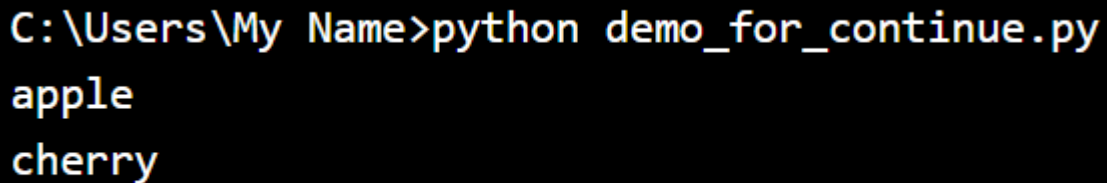
```
    if x == "banana":
```

```
        continue
```

```
    print(x)
```

A terminal window with a black background and white text. The prompt is 'C:\Users\My Name>' followed by the command 'python demo_for_break2.py'. The output is 'apple' on the next line.

```
C:\Users\My Name>python demo_for_break2.py
apple
```

A terminal window with a black background and white text. The prompt is 'C:\Users\My Name>' followed by the command 'python demo_for_continue.py'. The output is 'apple' on the first line and 'cherry' on the second line, with 'banana' being skipped.

```
C:\Users\My Name>python demo_for_continue.py
apple
cherry
```

The range() Function

- To loop through a set of code a specified number of times, we can use the range() function,
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.
- **Example**
- Using the range() function:
- **for x in range(6):**
 print(x)
- Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

```
C:\Users\My Name>python demo_for_range.py
0
1
2
3
4
5
```

- The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

- **Example**

Using the start parameter:

- **`for x in range(2, 6):`
 `print(x)`**

```
C:\Users\My Name>python demo_for_range2.py
2
3
4
5
```

- The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

- Example
- Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):  
    print(x)
```

```
C:\Users\My Name>python demo_for_range3.py  
2  
5  
8  
11  
14  
17  
20  
23  
26  
29
```

Else in For Loop

- The else keyword in a for loop specifies a block of code to be executed when the loop is finished:
- **Example**
- Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
```

```
    print(x)
```

```
else:
```

```
    print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

- The "inner loop" will be executed one time for each iteration of the "outer loop":
- **Example**
- Print each adjective for every fruit:
- `adj = ["red", "big", "tasty"]`
`fruits = ["apple", "banana", "cherry"]`

```
for x in adj:  
    for y in fruits:  
        print(x, y)
```

```
C:\Users\My Name>python demo_for_nested.py  
red apple  
red banana  
red cherry  
big apple  
big banana  
big cherry  
tasty apple  
tasty banana  
tasty cherry
```

Note:

What is meant by structured language?

- C is called a **structured** programming **language** because to solve a large problem, C programming **language** divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions

Examples of **Structured Programming** language are C, C+, C++, C#, Java, PERL, Ruby, PHP, ALGOL, Pascal, PL/I and Ada

What is unstructured programming language?

- An **unstructured** program is a procedural program – the statements are executed in sequence as written. But this type of **programming** uses the goto statement. A goto statement allows control to be passed to any other place in the program. ... This means that it is often difficult to understand the logic of such a program.

Examples of unstructured Programming language are JOSS, FOCAL, MUMPS, TELCOMP, COBOL

Procedural Programming Paradigm

Session 6 – 10 covers the following Topics:-

- Procedural Programming Paradigm
- Routines, Subroutines, functions
- Using Functions in Python
- logical view, control flow of procedural programming in various aspects
- Other languages: Bliss, ChucK, Matlab
- Demo: creating routines and subroutines using functions in Python
- Lab 2: Procedural Programming

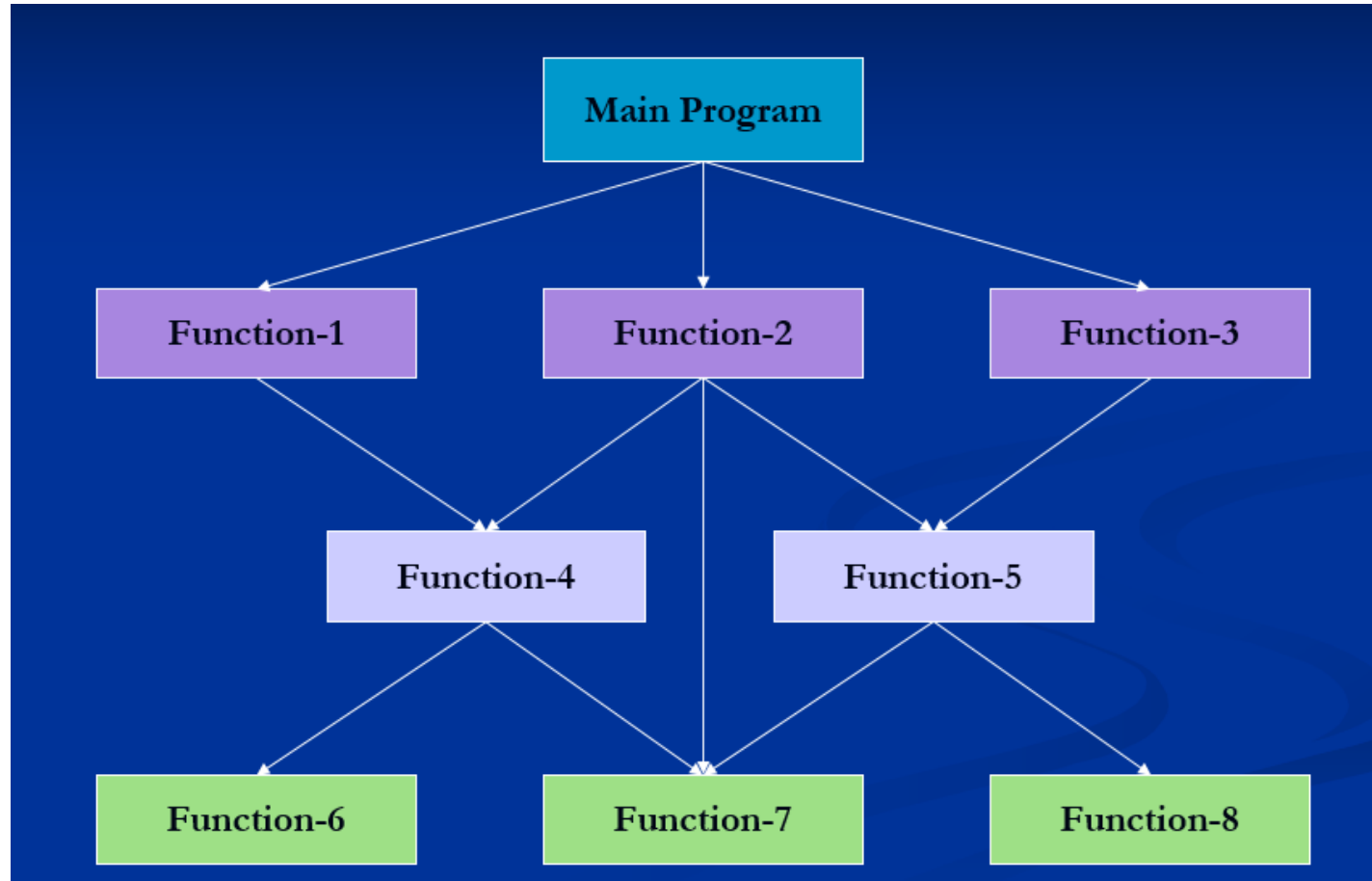
Text Book: Shalom, Elad. A Review of Programming Paradigms Throughout the History: With a Suggestion Toward a Future Approach, Kindle Edition

Procedure Oriented Programming(POP):-

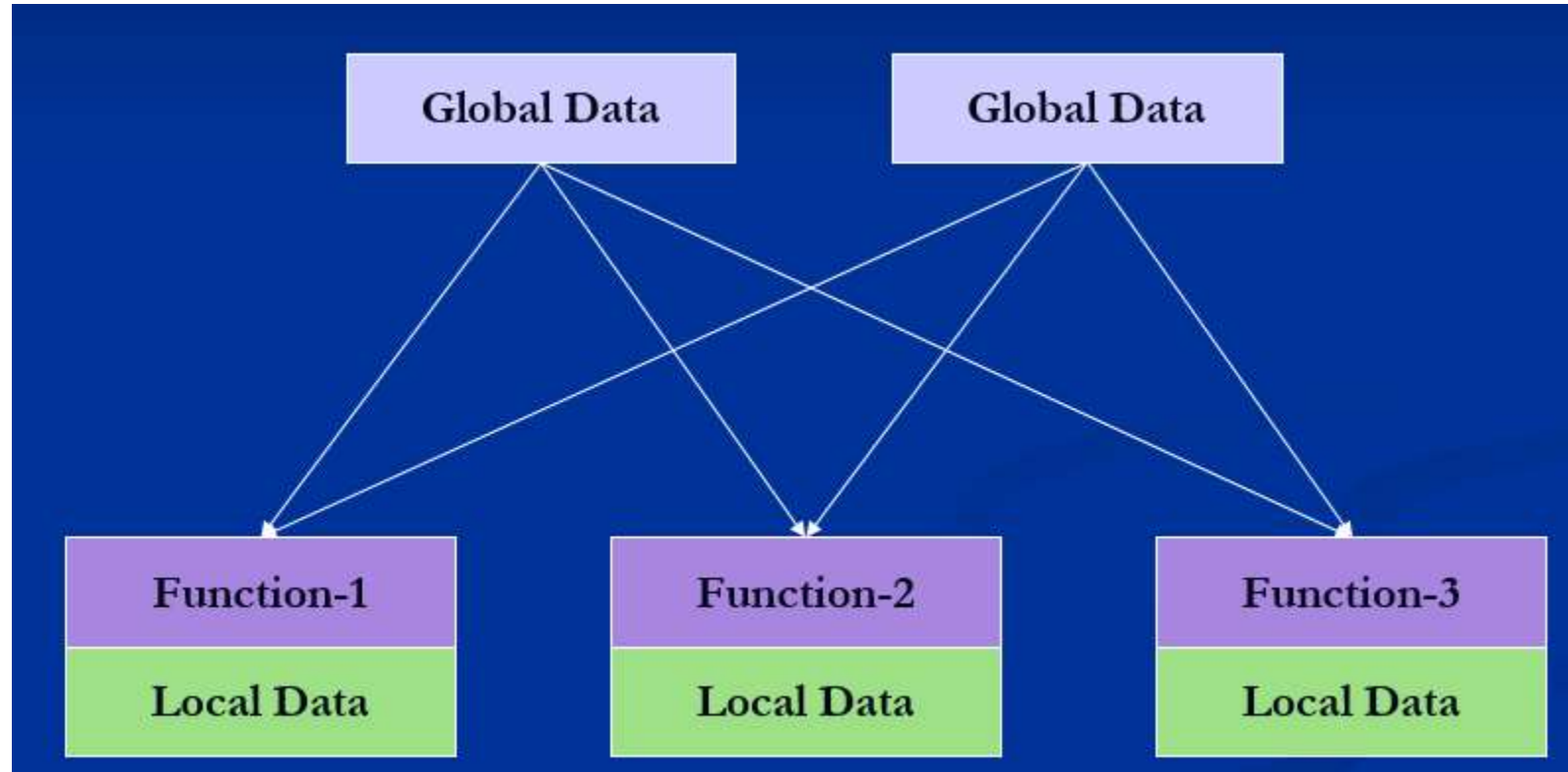
- High level languages such as COBOL, FORTRAN and C, is commonly known as procedure oriented programming(POP). In the procedure oriented programming, program is divided into sub programs or modules and then assembled to form a complete program. These modules are called functions.
- The problem is viewed as a sequence of things to be done.
- The primary focus is on functions.
- Procedure-oriented programming basically consists of writing a list of instructions for the computer to follow and organizing these instructions into groups known as functions.

- In a multi-function program, many important data items are placed as global so that they may be accessed by all functions. Each function may have its own local data. If a function made any changes to global data, these changes will reflect in other functions. Global data are more unsafe to an accidental change by a function. In a large program it is very difficult to identify what data is used by which function.
- This approach does not model real world problems. This is because functions are action-oriented and do not really correspond to the elements of the problem.

Typical structure of procedure-oriented program



Relationship of data and functions in procedural programming



Characteristics of Procedure-Oriented Programming

- Emphasis is on doing things.
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Logical view and Control flow of POP (routine, subroutine and function)

- **Procedural programming** is a programming paradigm, derived from structured programming, based on the concept of the *procedure call*. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.
- **procedural languages** generally use reserved words that act on blocks, such as if, while, and for, to implement control flow, whereas non-structured imperative languages use goto statements and branch tables for the same purpose.

Note:

Subroutine:-

- Subroutines; callable units such as procedures, functions, methods, or subprograms are used to allow a sequence to be referred to by a single statement.

Function in python

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.
- There are 2 types of function

Built-in function ex. `print()`

User defined function -User can create their own functions.

Defining a Function

- Function blocks begin with the keyword **def** followed by the function name and parentheses (()).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

Syntax:

- `def functionname(parameters):`
 `"function_docstring"`
 `function_suite`
 `return [expression]`

Example:

`#function definition`

```
def my_function():  
    print("Hello from a function")
```

`# To call a function, use the function name followed by parenthesis:`
my_function()

Function Arguments

You can call a function by using the following types of formal arguments –

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

- Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function `printme()`, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

Function definition is here

```
def printme( str ):
```

```
    "This prints a passed string into this function"
```

```
    print str
```

```
    return;
```

Now you can call printme function

```
printme()
```

When the above code is executed, it produces the following result –

Traceback (most recent call last):

```
File "test.py", line 11, in <module>
```

```
    printme();
```

TypeError: printme() takes exactly 1 argument (0 given)

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways –

Ex2:-

Ex1: # Function definition is here

- def printme(str):
- "This prints a passed string into this function"
- print str
- return;

Now you can call printme function

- printme(str = "My string")
- **When the above code is executed, it produces the following result –**

O/p -> My string

```
# Function definition is here
def printinfo( name, age ):
    "This prints a passed info into this function"
    print "Name: ", name
    print "Age ", age
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result –

```
Name: miki
Age 50
```

Note that the order of parameters does not matter

Default arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, **it prints default age if it is not passed** –

Function definition is here

```
def printinfo( name, age = 35 ):  
    "This prints a passed info into this function"  
    print "Name: ", name  
    print "Age ", age  
    return;
```

Now you can call printinfo function

```
printinfo( age=50, name="miki" )  
printinfo( name="miki" )
```

When the above code is executed, it produces the following result –

```
Name:  miki  
Age   50  
Name:  miki  
Age   35
```

Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.
- **Syntax for a function with non-keyword variable arguments is this –**

```
def functionname([formal_args,] *var_args_tuple ):
```

```
    "function_docstring"
```

```
    function_suite
```

```
    return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example –

Function definition is here

```
def printinfo( arg1, *vartuple ):  
    "This prints a variable passed arguments"  
    print "Output is: "  
    print arg1  
    for var in vartuple:  
        print var  
    return;
```

Now you can call printinfo function

printinfo(10) When the above code is executed, it produces the following result –

Output is:

10

Output is: **printinfo(70, 60, 50)**

70

60

50

The Anonymous Functions

- These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

- **The syntax of lambda functions contains only a single statement, which is as follows –**

lambda [arg1 [,arg2,.....argn]]:expression

- Following is the example to show how lambda form of function works –

Function definition is here

```
sum = lambda arg1, arg2: arg1 + arg2;
```

Now you can call sum as a function

```
print "Value of total : ", sum( 10, 20 )
```

```
print "Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result –

O/p

Value of total : 30

Value of total : 40

The return Statement

The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

All the above examples are not returning any value. You can return a value from a function as follows –

Function definition is here

```
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2
    print "Inside the function : ", total
    return total;
```

Now you can call sum function

```
total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result –

Inside the function : 30

Outside the function : 30

- **Scope of Variables**
- All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python –
 - **Global variables**
 - **Local variables**
- **Global vs. Local variables**
- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example –


```
total = 0; # This is global variable.
```

Function definition is here

```
def sum( arg1, arg2 ):
```

```
    # Add both the parameters and return them."
```

```
    total = arg1 + arg2; # Here total is local variable.
```

```
    print "Inside the function local total : ", total
```

```
    return total;
```

Now you can call sum function

```
sum( 10, 20 );
```

```
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result –

Inside the function local total : 30

Outside the function global total : 0

3. Object-Oriented Programming

Session 11-15 covers the following Topics:-

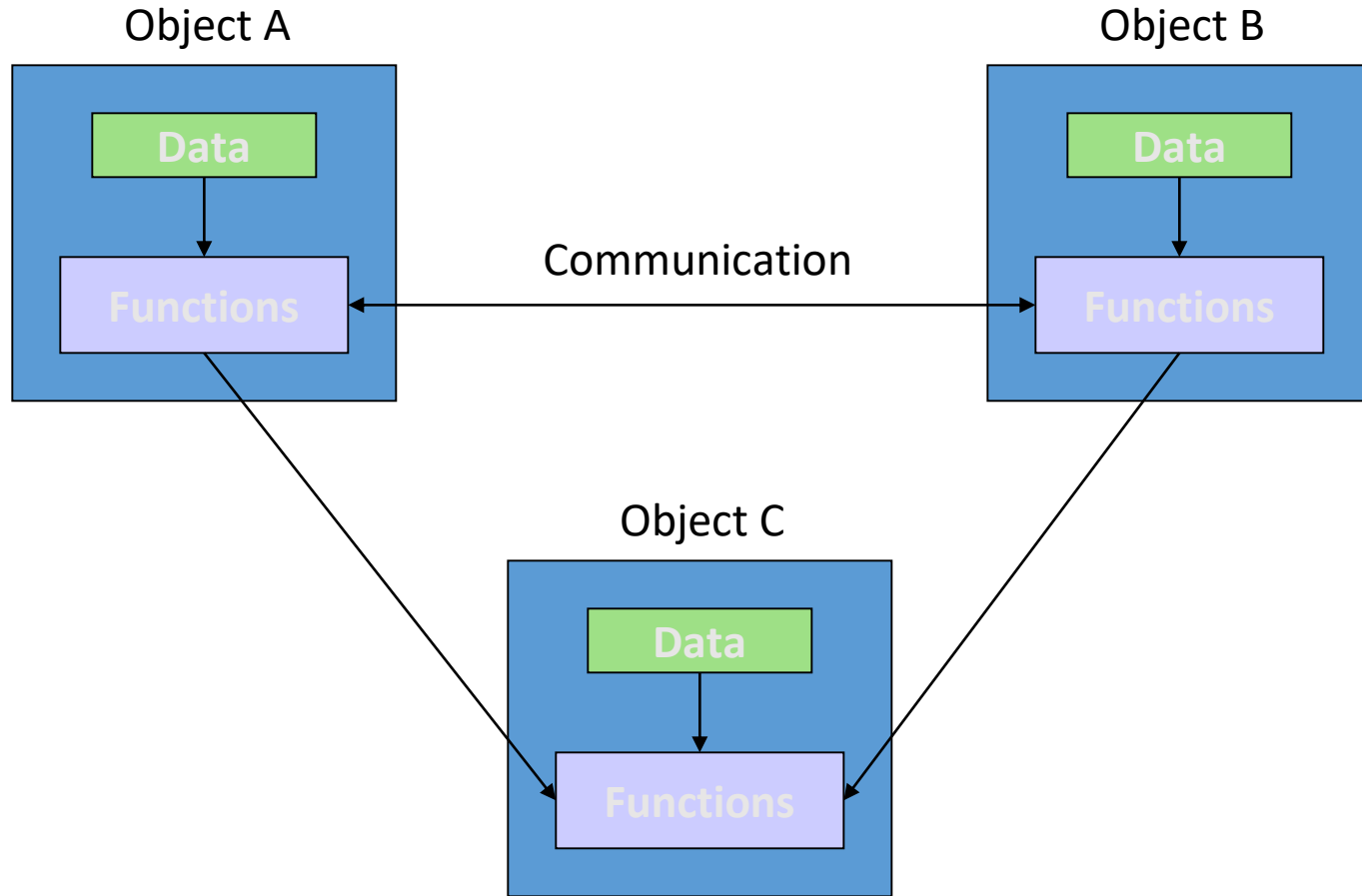
- Object Oriented Programming Paradigm
- Class, Objects, Instances, Methods
- Encapsulation, Data Abstraction
- Polymorphism, Inheritance
- Constructor, Destructor
- Example Languages: BETA, Cecil, Lava.
- Demo: OOP in Python
- **Lab 3: Object Oriented Programming**

TextBook: Shalom, Elad. A Review of Programming Paradigms Throughout the History:
With a Suggestion Toward a Future Approach

Object-Oriented Programming

- OOP treat data as a critical element in the program development and does not allow it to flow freely around the system.
- It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.
- OOP allows decomposition of a problem into a number of entities called objects and then build data functions around these objects.
- The data of an object can be accessed only by the functions associated with that object.
- Functions of one object can access the functions of another objects

Organization of data and functions in OOP



Characteristics of Object-Oriented Programming

- Emphasis is on data rather than procedure.
- Programs are divided into objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and can not be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be added easily whenever necessary.
- Follows bottom-up approach in program design.

Object-Oriented Programming

- **Definition:**

It is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand. *Thus the object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data.*

Basic Concepts of Object-Oriented Programming

- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Basic Concepts of OOP

continue ...

- **Objects**

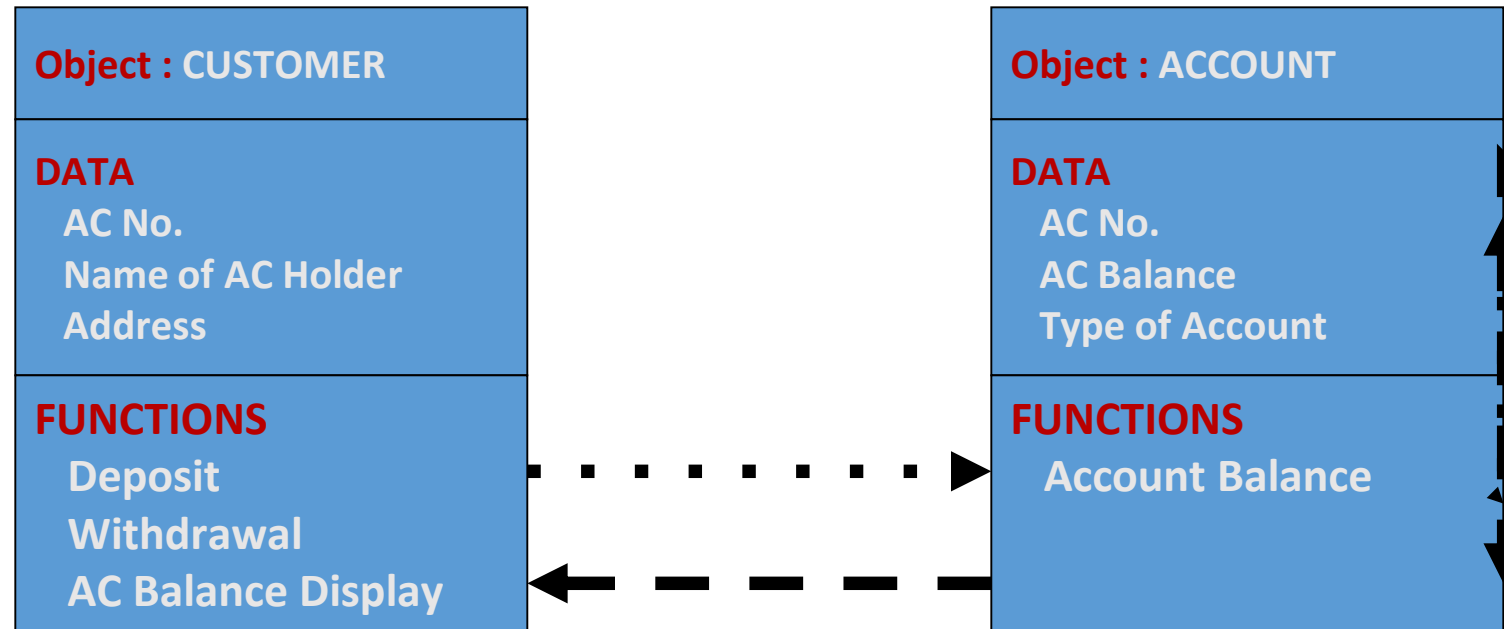
Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, etc. Objects take up space in the memory and have an associated address like a structure in C.

When a program is executed, the objects interact by sending messages to one another.

Basic Concepts of OOP

continue ...

- Objects



Basic Concepts of OOP

continue ...

- **Classes**

Classes are user-defined data types.

The entire set of data and code of an object can be made a user-defined data type with the help of a class. Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created.

A class is a collection of objects of similar type.

Basic Concepts of OOP

continue ...

- Classes

If fruit has been defined as a class, then the statement

```
f r u i t  m a n g o ;
```

will create an object **mango** belonging to the class **fruit**.

Basic Concepts of OOP

continue ...

- **Data Abstraction and Encapsulation**
 - o The wrapping up of data and functions into a single unit is known as encapsulation.
 - o The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
 - o These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.

Basic Concepts of OOP

continue ...

- Data Abstraction and Encapsulation

The attributes wrapped in the classes are called data members and the functions that operate on these data are called methods or member functions.

Since the classes use the concept of data abstraction, they are known as Abstracted Data Types (ADT).

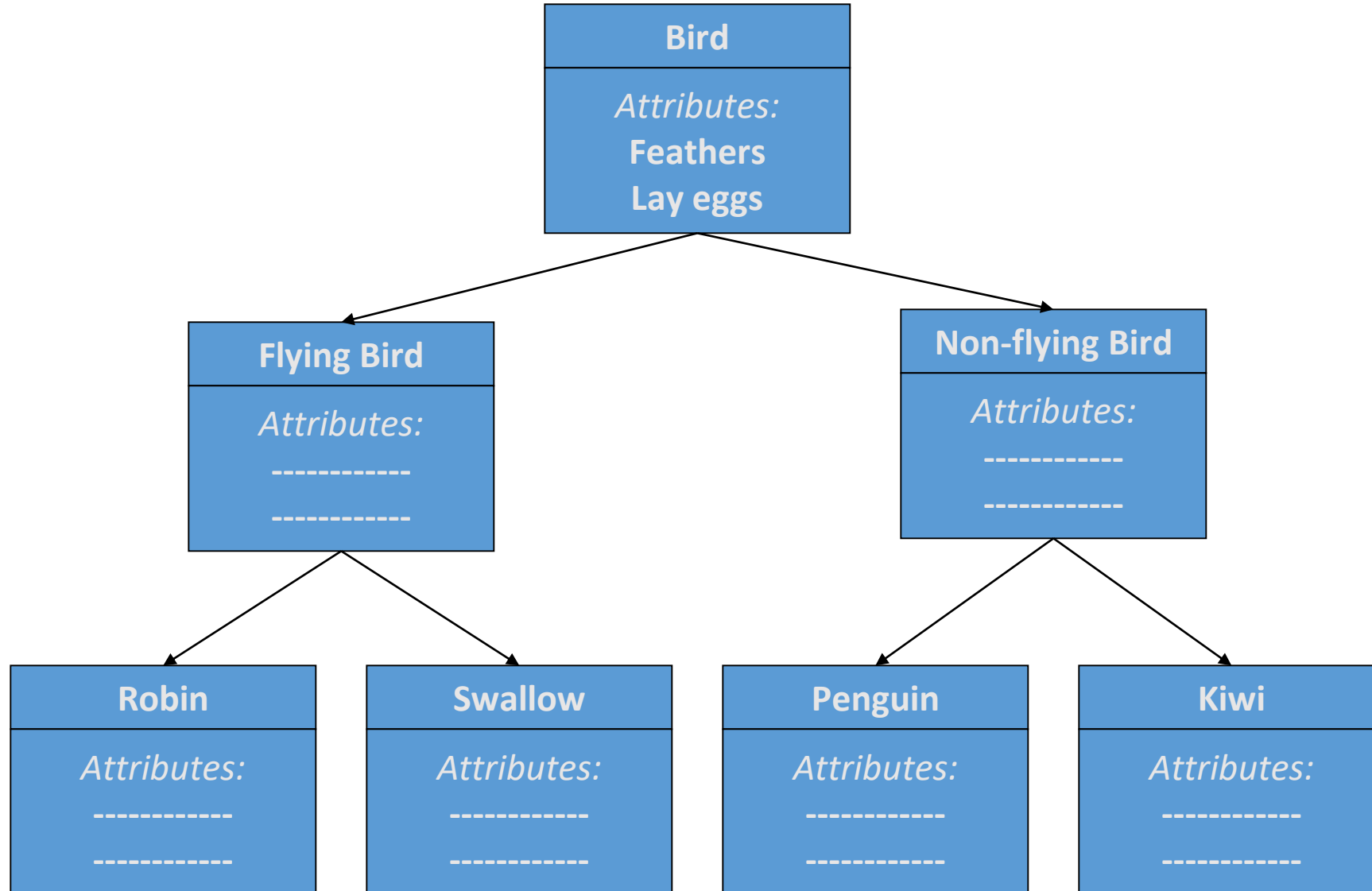
Basic Concepts of OOP

continue ...

- Inheritance

- o Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- o It supports the concept of hierarchical classification.
- o Each derived class shares common characteristics with the class from which it is derived.

Property Inheritance



Basic Concepts of OOP

continue ...

- Inheritance

- o Inheritance provides the idea of reusability.

- o We can add additional features to an existing class without modifying it.

- (By deriving new class from existing one. The new class will have the combined features of both the classes.)*

Basic Concepts of OOP

continue ...

- **Polymorphism** - *ability to take more than one form*
 - o An operation may exhibit different behaviours in different instances.
 - o The behaviour depends upon the types of data used in the operation.
 - o `add(3, 5)` gives 8
 - o `Add("hello", "-world")` gives "hello-world"

Basic Concepts of OOP

continue ...

- **Polymorphism** - *ability to take more than one form*
 - o The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
 - o << Insertion Operator
 - o << Left-shift bit-wise operator
 - o Using a single function name to perform different types of tasks is known as function overloading.
 - o add(3, 5) gives 8
 - o Add("hello", "-world") gives "hello-world"

Basic Concepts of OOP

continue ...

- **Dynamic Binding**

Binding refers to the linking of a procedure call to the code to be executed in response to the call.

Dynamic binding (late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time.

It is associated with polymorphism and inheritance.

Basic Concepts of OOP

continue ...

- **Message Passing**

- o An oop consists of a set of objects that communicate with each other.
- o Oop involves the following steps:
 - o Creating classes that define objects and their behaviour.
 - o Creating objects from class definitions.
 - o Establishing communication among objects.
- o Objects communicate with one another by sending and receiving information.

Basic Concepts of OOP

continue ...

- **Message Passing**

- o A message for an object is a request for execution of a procedure.
- o The receiving object will invoke a function and generates results.
- o Message passing involves specifying:
 - o The name of the Object.
 - o The name of the Function.
 - o The information to be send.

Benefits of OOP

- Inheritance – eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working module, no need of starting from the scratch.
- Data hiding helps the programmer to build secure programs that can not be invaded by code in other parts of the program.

Benefits of OOP

- Multiple instances of an objects can co-exists with out any interference. continue ...
- It is easy to partition the work in a project based on objects.
- Object-oriented system can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

constructor

- **What is constructor?**

A constructor is a special type of member function of a class which initializes objects of a class. Constructor is automatically called when object(instance of class) is created. It is special member function of the class because it does not have any return type.

How constructors are different from a normal member function?

- A constructor is different from normal functions in following ways:
 - Constructor has same name as the class itself
 - Constructors don't have input argument
 - Constructors don't have return type
 - A constructor is automatically called when an object is created.
 - It must be placed in public section of class.
 - If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

constructor

Constructor in C++

Default



Class_name()

Parameterized



Class_name(parameters)

Copy



Class_name(const Class_name old_object)



Destructor

- **What is a destructor?**

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

- The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

- **Syntax:**

- `~constructor-name();`

Destructor

- **Properties of Destructor:**
- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

Python Classes/Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

- To create a class, use the keyword class:

Example

- Create an object named p1, and print the value of x:

```
p1 = MyClass()  
print(p1.x)
```

Python Classes/Objects

The `__init__()` Function

- The examples above are classes and objects in their simplest form, and are not really useful in real life applications.
- To understand the meaning of classes we have to understand the built-in `__init__()` function.
- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

- Create a class named Person, use the `__init__()` function to assign values for name and age:

class Person:

```
def __init__(self, name, age):  
    self.name = name  
    self.age = age
```

```
p1 = Person("John", 36)
```

```
print(p1.name)  
print(p1.age)
```

Python Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

Create a Parent Class

- Any class can be a parent class, so the syntax is the same as creating any other class:

Example

- Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

#Use the Person class to create an object, and then execute the printname method:

```
x = Person("John", "Doe")
x.printname()
```

Python Inheritance

Create a Child Class

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

- Create a class named Student, which will inherit the properties and methods from the Person class:
- ```
class Student(Person):
 pass
```

**Note:** Use the pass keyword when you do not want to add any other properties or methods to the class.

## Example

- Use the Student class to create an object, and then execute the printname method:
- ```
x = Student("Mike", "Olsen")  
x.printname()
```

Python Inheritance

Add the `__init__()` Function

- So far we have created a child class that inherits the properties and methods from its parent.
- We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Example

- Add the `__init__()` function to the Student class:
- ```
class Student(Person):
 def __init__(self, fname, lname):
 #add properties etc.
```
- When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

**Note:** The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

- To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

## Example

- ```
class Student(Person):  
    def __init__(self, fname, lname):  
        Person.__init__(self, fname, lname)
```
- Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

5.Declarative paradigm

Unit- 2 (15 Session)

Session 6-10 covers the following Topics:-

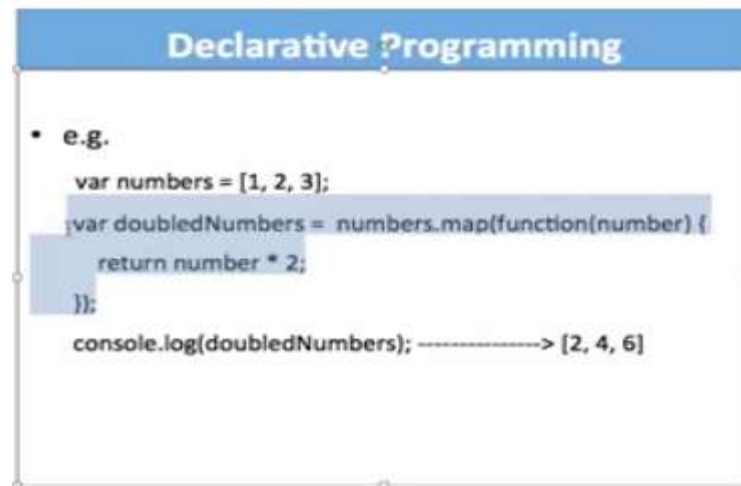
- Definition Declarative Paradigm
 - Sets of Declarative Statements
 - Object Attribute and Binding behavior
 - Creating Event without describing flow
 - Other languages: Prolog, Z3, LINQ, SQL
 - Demo: Declarative Programming in Python
 - Lab 5: Declarative Programming
-
- **TextBook:** Shalom, Elad. A Review of Programming Paradigms Throughout the History: With a Suggestion Toward a Future Approach, Kindle Edition

1. Declarative paradigm

- Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow.
- Logic, functional and domain-specific languages belong under declarative paradigms

Examples would be HTML, XML, CSS, SQL, Prolog, Haskell, F# and Lisp.

- Declarative code focuses on building logic of software without actually describing its flow. You are saying what without adding how. For example with HTML you use `` to tell browser to display an image and you don't care how it does that.



1.1 History

- The two main subparadigms of declarative programming are
functional Programming
&
logic programming.

Functional and logical programming languages are characterized by a declarative programming style.

In logical programming languages, programs consist of logical statements, and the program executes by searching for proofs of the statements.