Python Programming

Department of AI and DS

Course Outcome

CO1: Understand the fundamental concepts of computers and basic blocks of programming.

CO2: Apply control statements, decision making statements to solve the given problems.

CO3: Usage of complex data types in python to develop on application.

C04: Practice problem solving using functions and string operations.

C05: Solve the real - world problems using file handling operations, modules, packages and

error handling methods.

Modules

Module 1: Introduction to Programming

Module 2: Python Statements

Module 3: Python Complex Data types

Module 4: Functions and Strings

Module 5: Files, Modules and Package

Computer Programming

PROGRAM & PROGRAMMING

- A program is a set of logically related instructions that is arranged in a sequence that directs the computer in solving a problem.
- The process of writing a program is called programming.
- Software is a collection of computer programs and related data that provides the instructions for telling a computer what to do and how to do it.
- Computer software can be broadly classified into two categories :

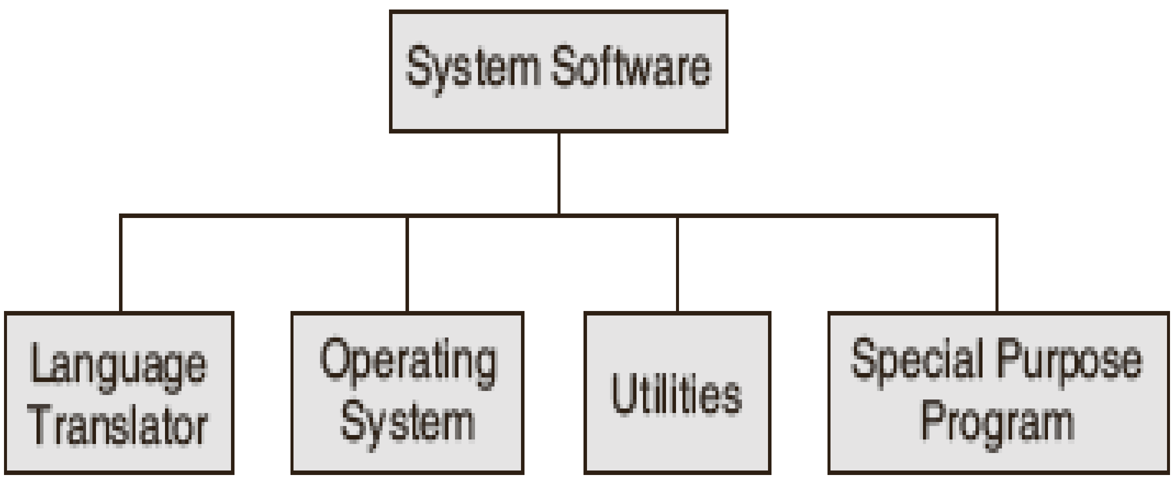
  (a) system software

  &

  (b) application software

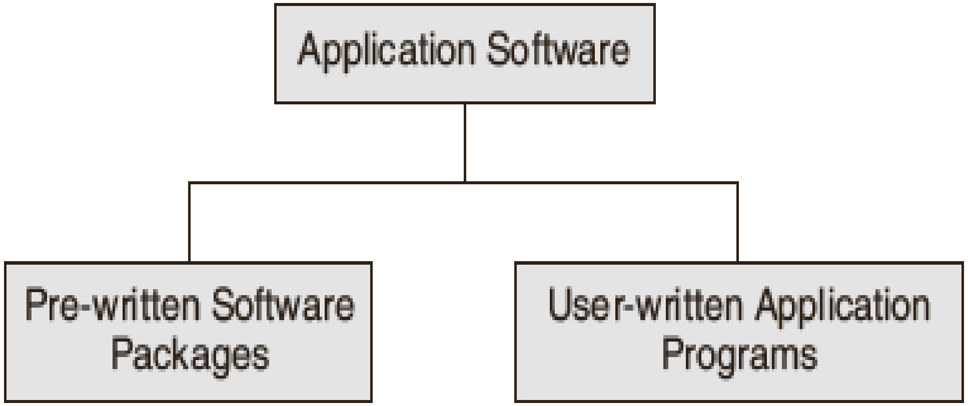
SYSTEM SOFTWARE

- System software is a collection of programs that interfaces with the hardware.
- Categories of system software :



APPLICATION SOFTWARE

- Application software is written to enable the computer to solve a specific data processing task.
- Categories of application software :



PROGRAMMING LANGUAGE

- A programming language is composed of a set of instructions in a language understandable to the programmer and recognizable by a computer.

- Programming languages can be classified as

  (a) High-level language - BASIC, COBOL & FORTRAN(application programs).

  (b) Middle level language - C (application & system programs).

  (c) Low level language – assembly language (system programs).

COMPILER

- For executing a program written in a high-level language, it must be first translated into a form the machine can understand. This is done by a software called the *compiler.*
- The compiling process consists of two steps:

  a . The analysis of the source program and

  b . The synthesis of the object program in the machine language of the specified machine.
- Compiler action :

INTERPRETER

- During the process of translation There is another type of software that also does translation. This is called an interpreter.
- Differences between compiler and interpreter :

Compiler	Interpreter
Scans the entire program before translating it into machine code.	Translates and executes the program line by line.
Converts the entire program to machine code and executes program only when all the syntax errors are removed.	The interpreter executes one line at a time, after checking and correcting its syntax errors and then converting it to machine code.
Slow in debugging or removal of mistakes from a program.	Good for fast debugging.
Program execution time is less.	Program execution time is more.

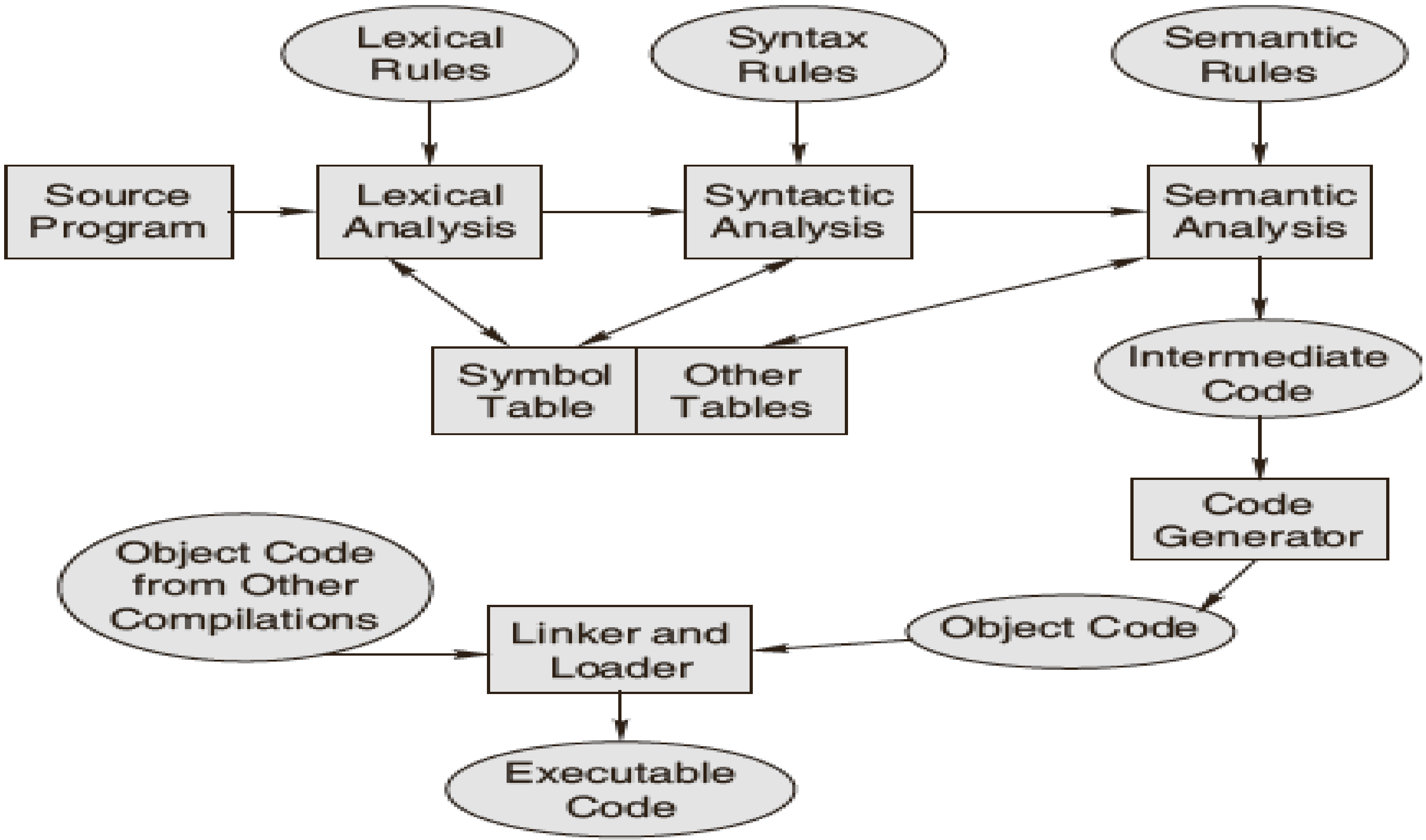
COMPILING & EXECUTING HIGH LEVEL LANGUAGE

- The compiling process consists of two steps: the analysis of the source program and the synthesis of the object program in the machine language of the specified machine.

- The analysis phase uses the precise description of the source programming language.

- A source language is described using (a) *lexical rules, (b)syntax rules, and (c)semantic* rules.

THE PROCESS OF COMPILATION



EXECUTION STEPS OF A PROGRAM

- Steps :

1. ***Translation of the program resulting in the object*** program.
2. ***Linking of the translated program with other object*** programs needed for execution, thereby resulting in a binary program.
3. ***Relocation of the program to execute from the specific*** memory area allocated to it.
4. ***Loading of the program in the memory for the purpose*** of execution.

LINKER

- Linking resolves symbolic references between object programs. It makes object programs known to each other.
- Linking makes the addresses of programs known to each other so that transfer of control from one subprogram to another or a main program takes place during execution.
- In FORTRAN/COBOL , all program units are translated separately.

RELOCATION

- Relocation is more than simply moving a program from one area to another in the main memory.

- Relocation means adjustment of all address-dependent locations, such as address constant, to correspond to the allocated space, which means simple modification of the object program so that it can be loaded at an address different from the location originally specified.

LOADER

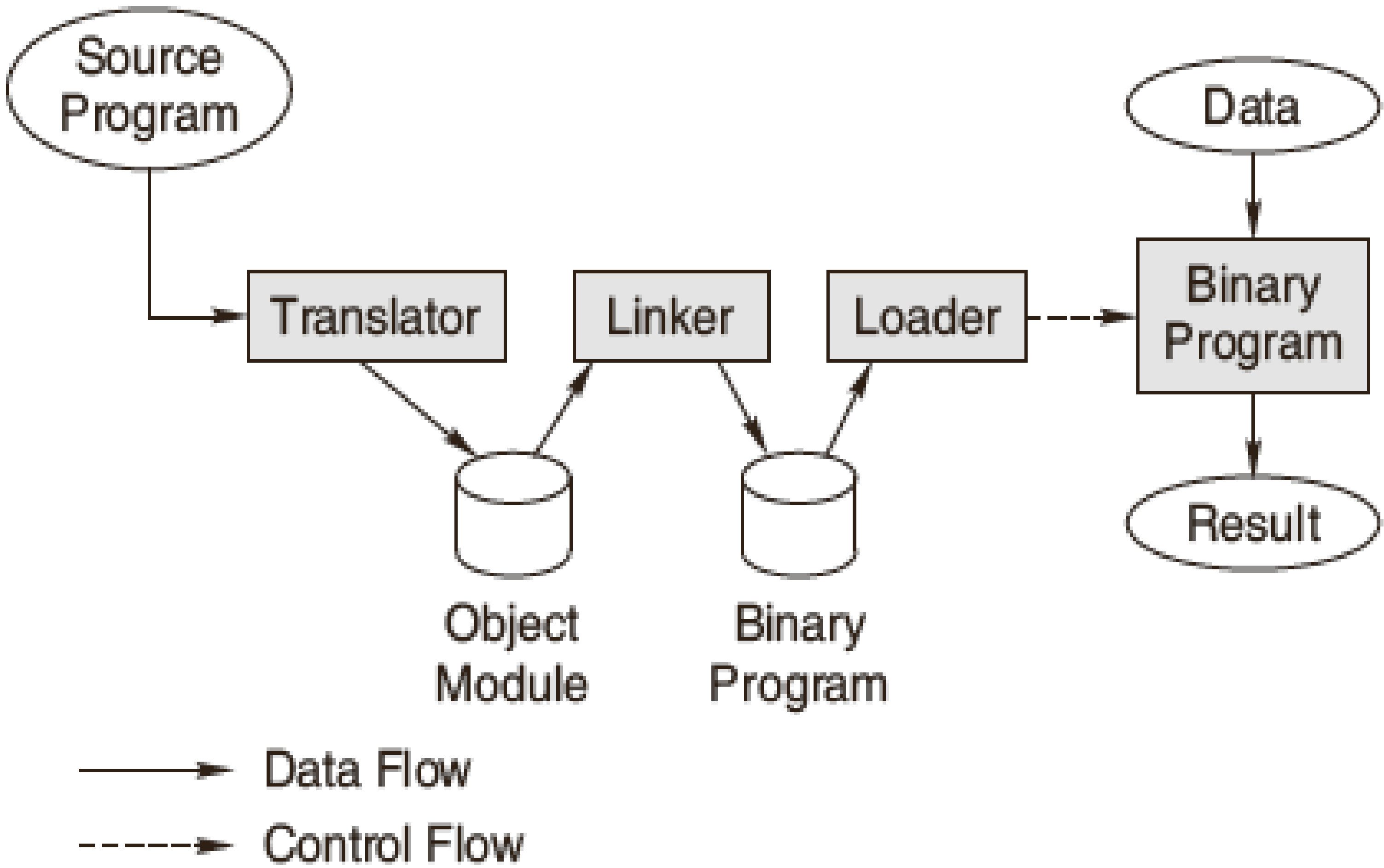
- Loading means physically placing the machine instructions and data into main memory, also known as primary storage area.
- The functions performed by the loader are :

a. Assignment of load-time storage area to the program
b. Loading of program into assigned area
c. Relocation of program to execute properly from its load time storage area
d. Linking of programs with one another

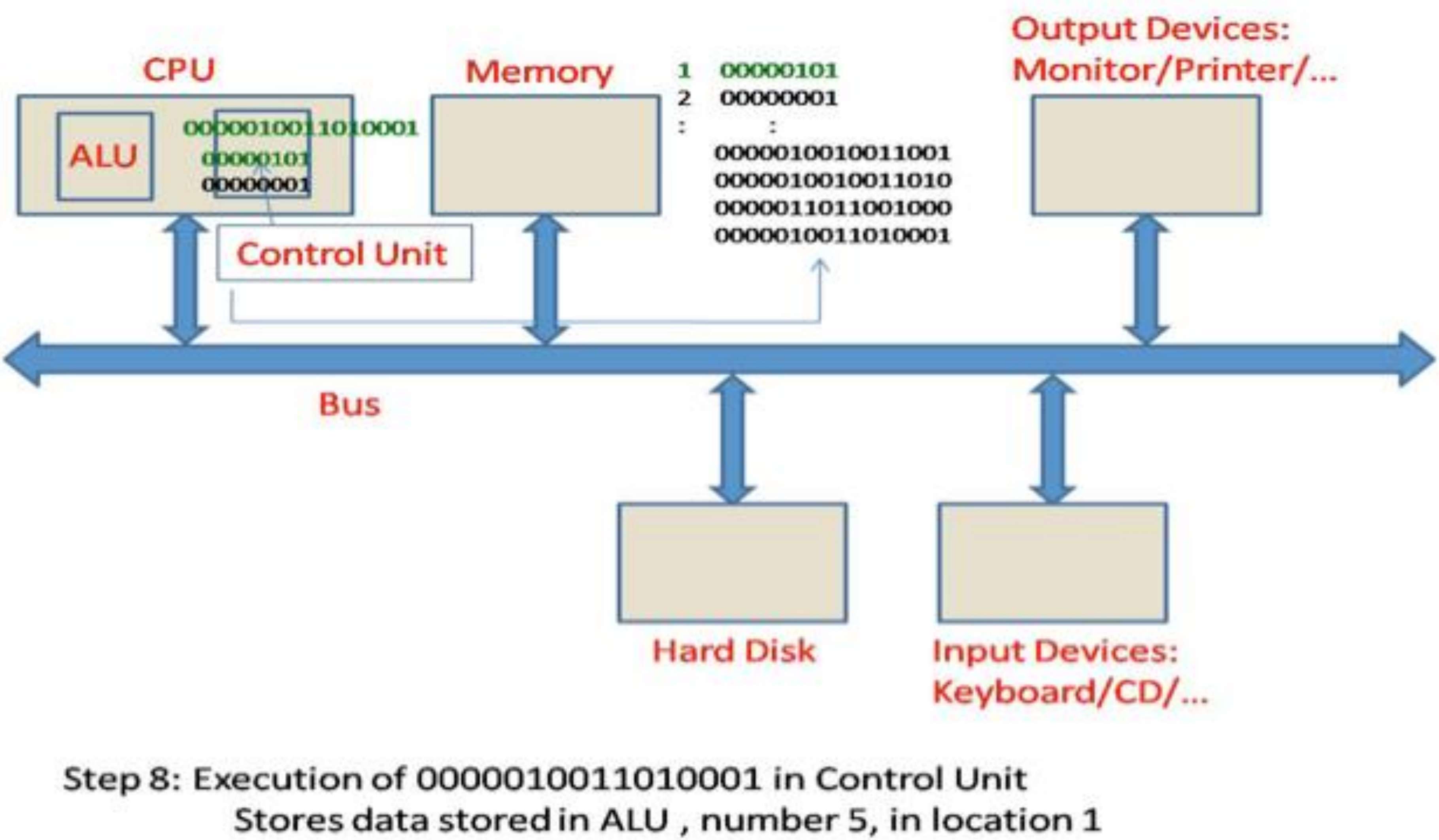
PROGRAM EXECUTION

- When a program is compiled and linked, each instruction and each item of data is assigned an address.
- At execution time, the CPU finds instructions and data from these addresses.
- The program counter, is a CPU register that holds the address of the next instruction to be executed in a program.
- The CPU has random access capability to any and all words of the memory, no matter what their addresses.

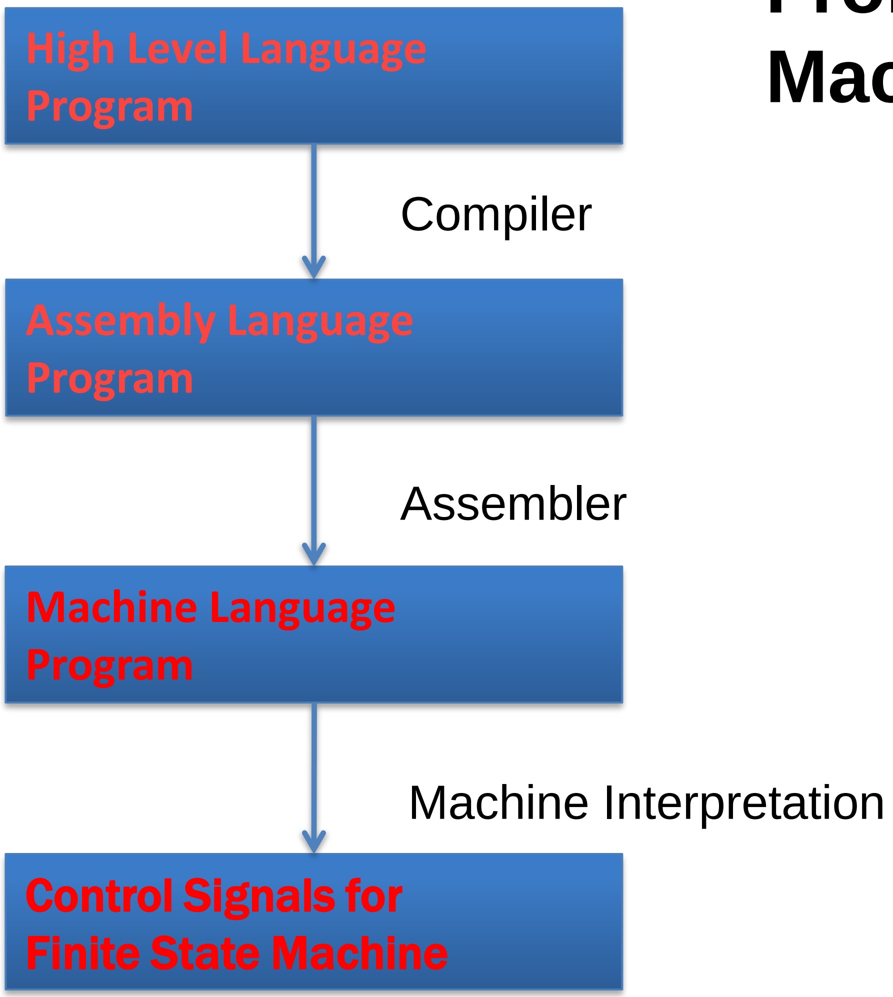
BLOCK DIAGRAM OF PROGRAM EXECUTION



LOGICAL ARRANGEMENT OF PARTS OF COMPUTER



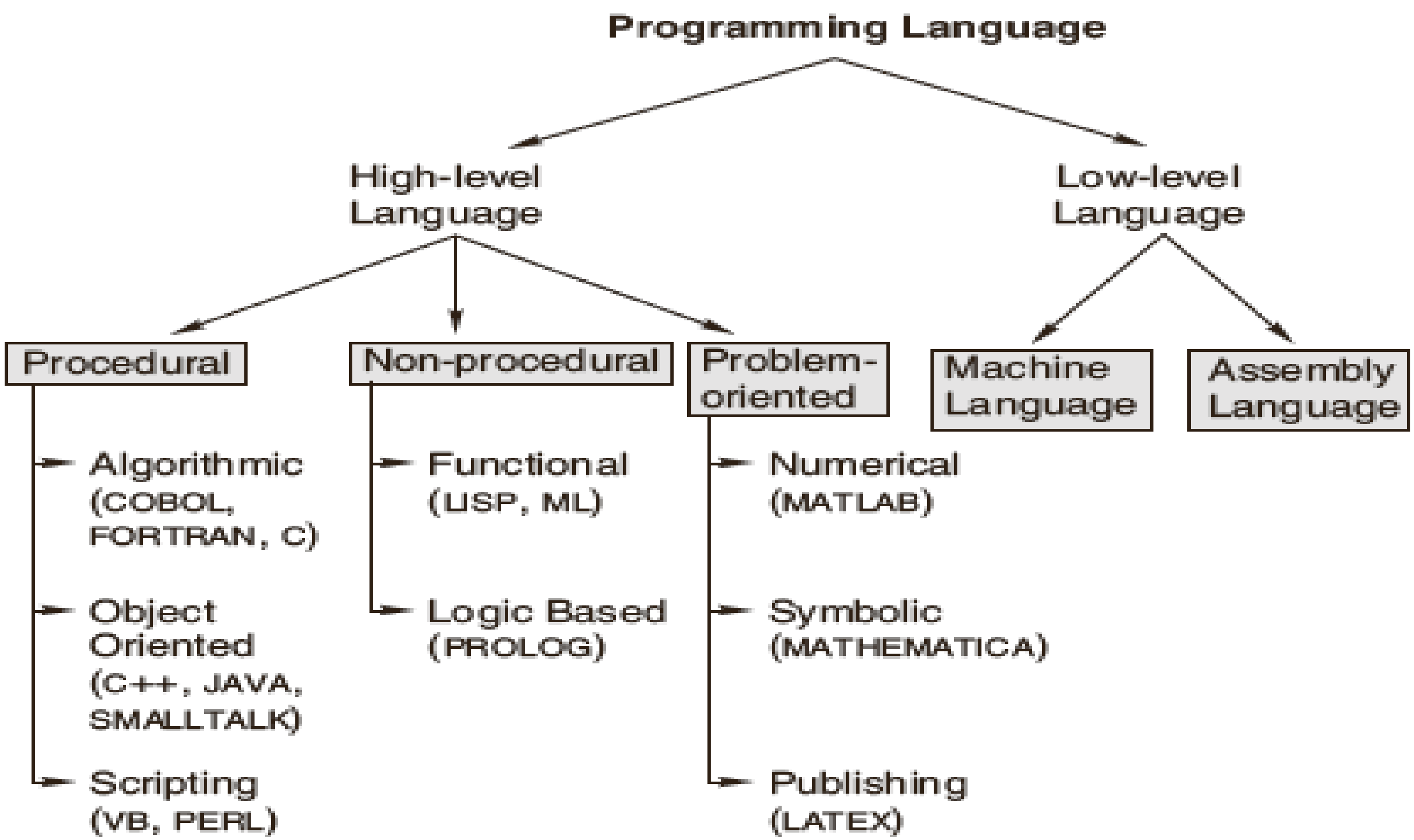
From Program To Machine Instructions



THIRD,FORTH & FIFTH GENERATION LANGUAGE

- Third generation programming language specifies how to perform a task using a large number of procedural instructions and is file oriented.
- Fourth generation programming language specifies what task has to be performed using fewer instructions and is database oriented.
- Fifth generation programming language resembles human speech and eliminates the need for the user or programmer to learn a specific vocabulary, grammar ,or syntax.

CLASSIFICATION OF PROGRAMMING LANGUAGES



Problem-Solving Approach to Programming:

- Programming is a tool to solve the problem

Algorithms as Paths Through "Problem Spaces"

Algorithm: a precise specification of a behaviour intended to solve a well-defined problem

Coding: the translation of an algorithm into a particular programming language

Program: a step-by-step execution plan that a computer can perform as a sequence of simple steps or instructions

STUCTURED PROGRAMMING

- Structured programming involves top–down analysis for program solving, modularization of program structure and organizing structured code for individual module.
- Top-down analysis breaks the whole problem into smaller logical tasks and defines the hierarchical link between the tasks.
- Modularization of program structure means making the small logical tasks into independent program modules that carries out the desired tasks.
- Structured coding is structured programming which consists of writing a program that produces a well organized module.

ALGORITHM

- An algorithm is ‘an effective procedure for solving a problem in a finite number of steps’.
- A well-designed algorithm has termination and correctness properties.
- The four common ways of representing an algorithm are the Step-form, Pseudo-code, Flowchart and Nassi-Schneiderman .
- algorithms show these three features:

a. Sequence (also known as process)

b. Decision (also known as selection)

c. Repetition (also known as iteration or looping)

VARIABLE & SUBROUTINE

- A variable, which has a name, is a container for a value that may vary during the execution of the program.
- A subroutine is a logical collection of instructions that is invoked from within a larger program to perform a specific task.
- The subroutine is relatively independent of the remaining statements of the program that invokes it & can be invoked several times from several places during a single execution.
- After completing the specific task, a subroutine returns to the point of invocation in the larger program.

PSEUDO CODE & FLOW CHART

- Like step-form, Pseudo-code is a written statement of an algorithm using a restricted and well-defined vocabulary.
- A flowchart comprises of a set of standard shaped boxes that are interconnected by flow lines to represent an algorithm.
- There should be a logical start and stop to the flowchart.
- The usual direction of the flow of a procedure or system is from left to right or top to bottom.
- The intersection of flow lines should be avoided.
- Flowcharts facilitate communication between programmers and users.

EXAMPLE: PSEUDO CODE

- **Problem:**
  - ➢ Write an algorithm to find out whether a given number is a prime number or not.
- **Solution:**
  - ➢ The algorithm for checking whether a given number is a prime number or not is as follows:

```
1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. IF N = 2 THEN
    PRINT "CO-PRIME" GOTO STEP 12
5. D ← 2
6. Q ← N/D (Integer division)
7. R ← N – Q*D
8. IF R = 0 THEN GOTO STEP 11
9. D ← D + 1
10. IF D <= N/2 THEN GOTO STEP 6
11. IF R = 0 THEN
     PRINT "NOT PRIME"
    ELSE
     PRINT "PRIME"
12. STOP
```

FLOW CHARTS : SYMBOLIC REPRESENTATION

- The START and STOP are represented by an ellipse-like figure :

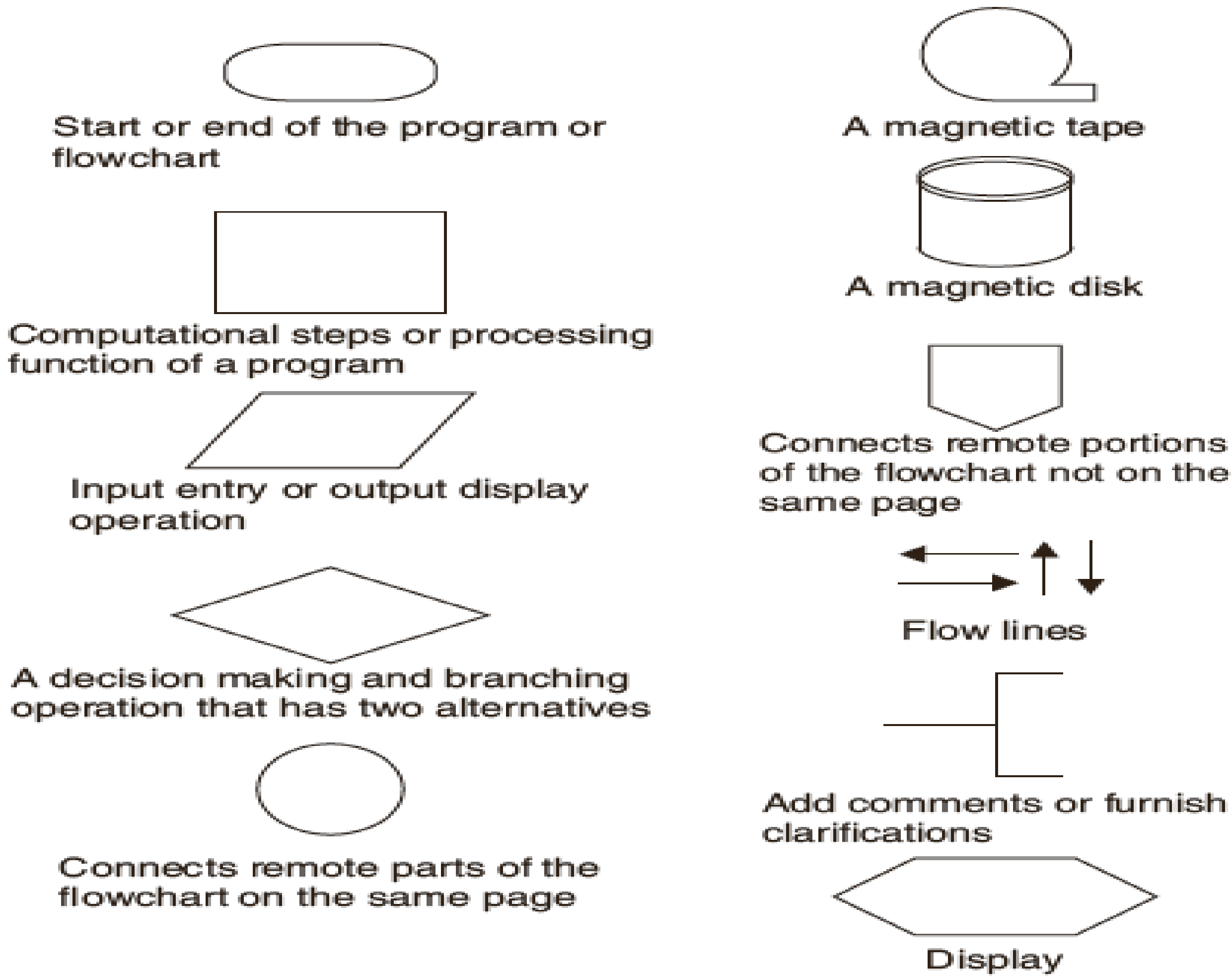
- Decisions construct by the rhombus-like figure :

- The processes by rectangles :

- Input / Output by parallelograms :

- Lines and arrows connect these blocks.

FLOW CHARTS:SYMBOLIC REPRESENTATION



FLOW-CHART ADVANTAGES

- *Communication*
- *Effective analysis*
- *Proper documentation*
- *Efficient coding*
- *Proper debugging*
- *Efficient program maintenance*

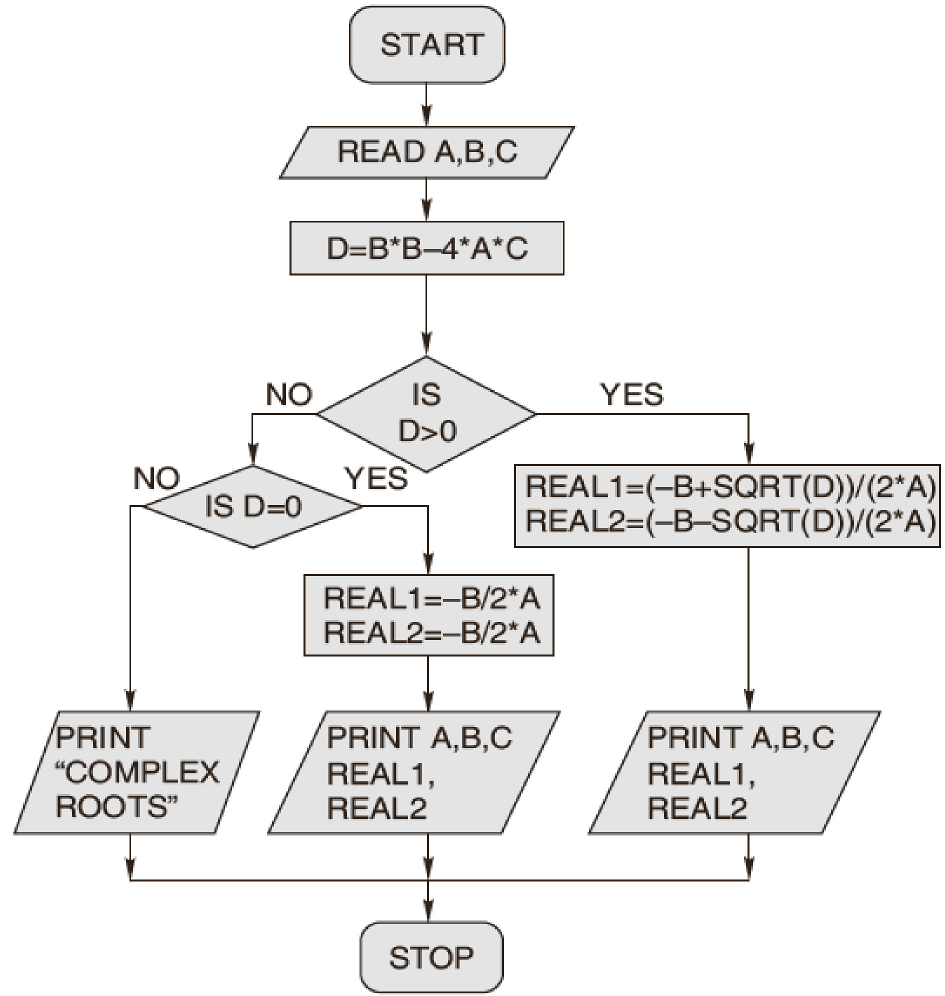
FLOW-CHARTS LIMITATIONS

- *Complex logic*
- *Alterations and modifications*
- *Reproduction*
- *Loss of objective*

FLOWCHART EXAMPLE

- **Problem:**

Draw a flowchart to find the roots of a quadratic equation.



TOP DOWN DEVELOPMENT STEP

- The top-down development phase plans out the way the solution has to be done by breaking it into smaller modules and establishing a logical connection among them.
- ***Stepwise refinement :***

a. Work out each and every detail for each small piece of manageable solution procedure.

b. Decompose any solution procedure into further smaller pieces and iterate until the desired level of detail is achieved.

CONT.

c. Group processes together which have some commonality.

d. Group variables together which have some appropriate commonality.

e. Test each small procedure for its detail and correctness and its interfacing with the other small procedures.

CONVERSION

- **Specification for Converting Algorithms into Programs:**

The general procedure to convert an algorithm into a program is to code the algorithm using a suitable programming language, check the program code by employing the desk-check method and finally evaluate and modify the program, if needed.

Object Oriented Programming

Programming model based on the concept of objects and classes. In this model, programmers define the functions that can be applicable to the data structures and their data type. Object-oriented programming turns data structure into an object, including both data and functions. It encourages the reusing of these objects in the same and other programmes as well.

Concept of OOPS

- Object and object instantiation
- Class methods
- Inheritance in Python Class
- Encapsulation
- Polymorphism
- Data abstraction

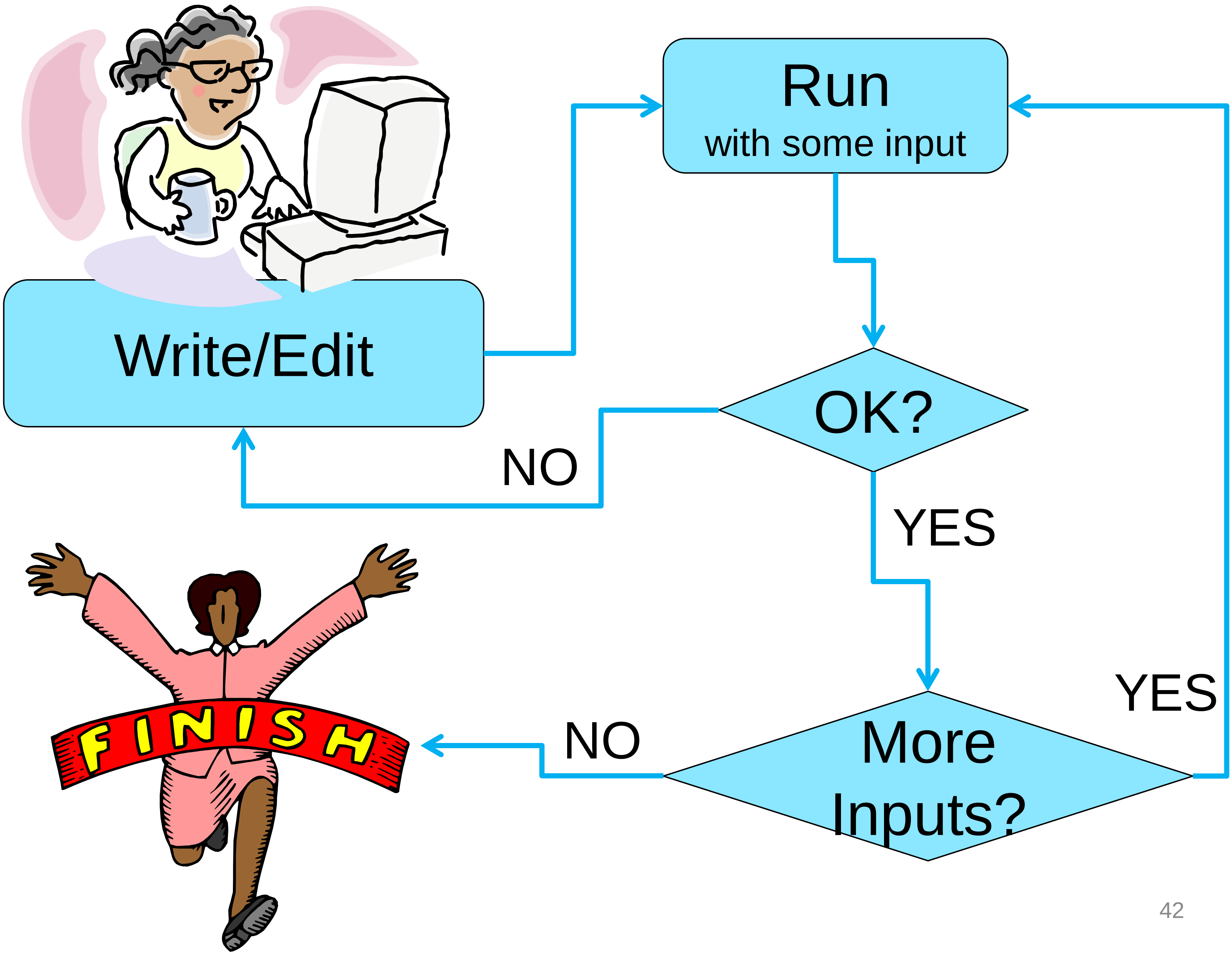
Introduction to Python

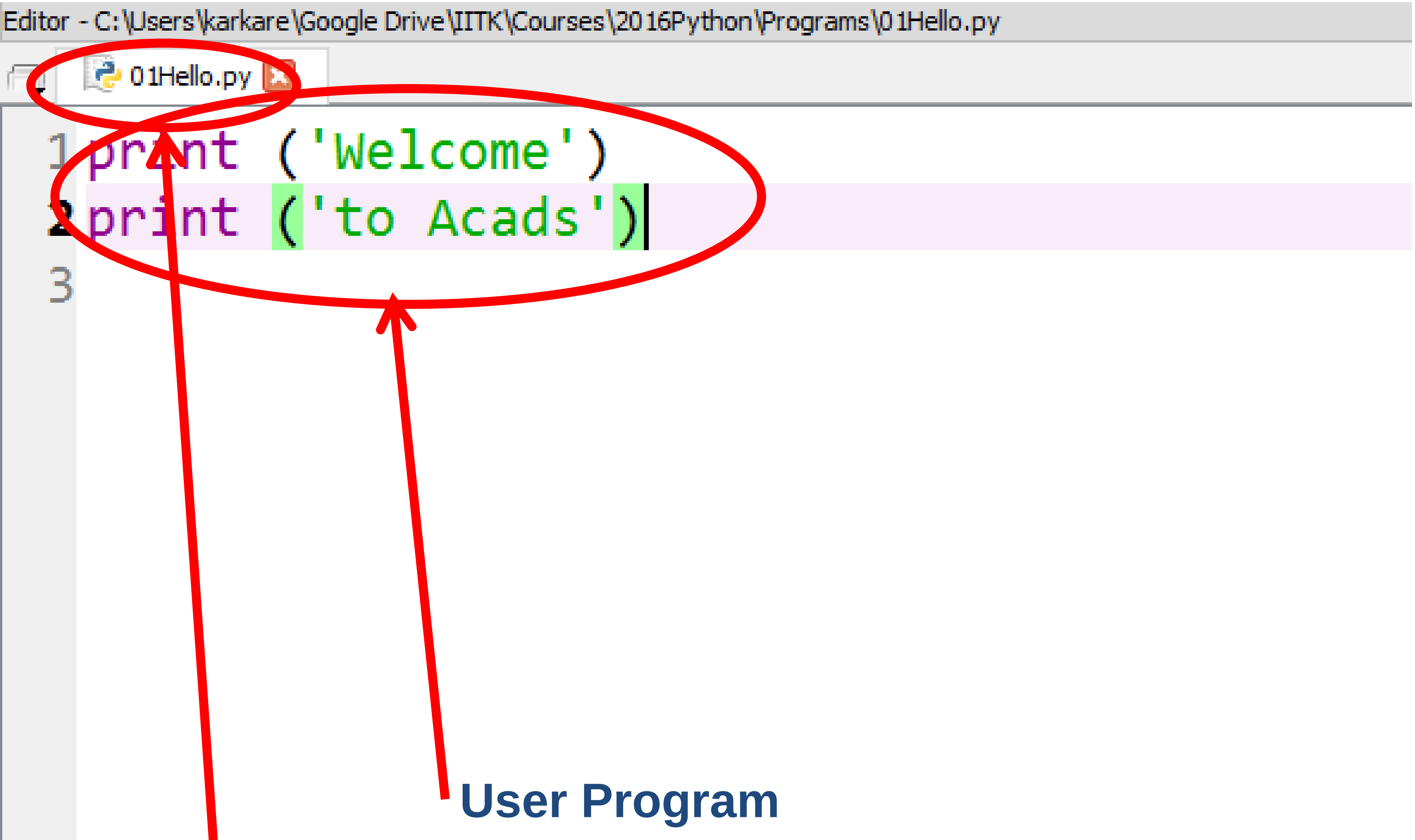
Python

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
- Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together.

Python

- Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.
- Python supports modules and packages, which encourages program modularity and code reuse.
- Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed


Write/Edit
Run
with some input
OK?
NO
YES
More
Inputs?
YES
NO
FINISH



User Program

Filename, preferred extension is py

```
IN[1]:
   Welcome
   to Acads
IN[2]:
```

Python Shell Prompt

Outputs

Python Shell is Interactive

Interacting with Python Programs

- Python program communicates its results to user using print
- Most useful programs require information from users
  - Name and age for a travel reservation system
- Python 3 uses input to read user input as a string (str)

input

- Take as argument a string to print as a prompt
- Returns the user typed value as a string
  - details of how to process user string later

```
IN[1]: age =      input('How old are you?')

IN[2]:

IN[3]:
```

Elements of Python

- A Python program is a sequence of **definitions** and **commands (statements)**
- Commands manipulate **objects**
- Each object is associated with a **Type**
- **Type:**
  - A set of values
  - A set of operations on these values
- **Expressions**: An operation (combination of objects and **operators**)

Types in Python

- **int**
  - Bounded integers, e.g. 732 or -5
- **float**
  - Real numbers, e.g. 3.14 or 2.0
- **long**
  - Long integers with unlimited precision
- **str**
  - Strings, e.g. ‘hello’ or ‘C’

Types in Python

- **Scalar**
  - Indivisible objects that do not have internal structure
  - **int** (signed integers), **float** (floating point), **bool** (Boolean), ***NoneType***
    - NoneType is a special type with a single value
    - The value is called **None**
- **Non-Scalar**
  - Objects having internal structure
  - **str** (strings)

Example of Types

```
In [14]: type(500)
Out[14]: int
```

Type Conversion (Type Cast)

- Conversion of value of one type to other
- We are used to int ↔ float conversion in Math
  - Integer 3 is treated as float 3.0 when a real number is expected
  - Float 3.6 is truncated as 3, or rounded off as 4 for integer contexts
- Type names are used as type converter functions

Type Conversion Examples

```
In [20]: int(2.5)
Out[20]: 2

In [21]: int(2.3)
Out[21]: 2

In [22]: int(3.9)
Out[22]: 3
```

Note that float to int conversion is truncation, not rounding off

```
In [23]: float(3)
Out[23]: 3.0

In [24]: int('73')
Out[24]: 73

In [25]: int('Acads')
Traceback (most recent call last):

  File "<ipython-input-25-90ec37205222>", line 1, in <module>
    int('Acads')

ValueError: invalid literal for int() with base 10: 'Acads'
```

```
In [26]: str(3.14)
Out[26]: '3.14'

In [27]: str(26000)
Out[27]: '26000'
```

Type Conversion and Input

```
In [11]: age = input('How old are you? ')

How old are you? 35

In [12]: print ('In 5 years, your age will be', age + 5)
```

```
In [13]: print ('In 5 years, your age will be', int(age) + 5)
In 5 years, your age will be 40
```

Operators

Arithmetic	+	-	*	//	/	%	**	
Comparison	==	!=	>	<	>=	<=		
Assignment	=	+=	-=	*=	//=	/=	%=	**=
Logical	and	or	not					
Bitwise	&	|	^	~	>>	<<		
Membership	in	not in						
Identity	is	is not						

Variables

- A name associated with an object
- Assignment used for binding

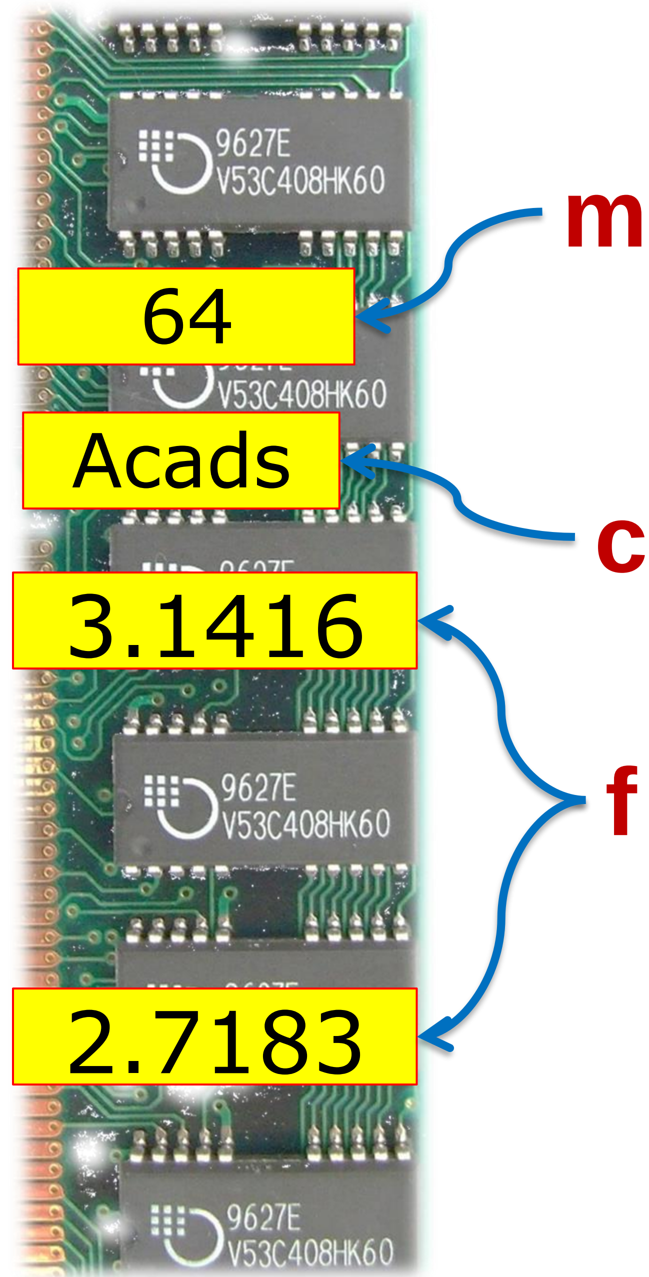
m = 64;

c = ‘Acads’;

f = 3.1416;

- Variables can change their bindings

f = 2.7183;



Assignment Statement

- A simple assignment statement

Variable = Expression;

- Computes the value (object) of the expression on the right hand side expression (RHS)
- Associates the name (variable) on the left hand side (LHS) with the RHS value
- = is known as the assignment operator.

Multiple Assignments

- Python allows multiple assignments

```
x, y = 10, 20
```

Binds x to 10 and y to 20

- Evaluation of multiple assignment statement:
  - All the expressions on the RHS of the = are first evaluated **before any binding happens**.
  - Values of the expressions are bound to the corresponding variable on the LHS.

```
x, y = 10, 20
x, y = y+1, x+1
```

x is bound to 21 and y to 11 at the end of the program

Programming using Python

Operators and Expressions

Binary Operations

Op	Meaning	Example	Remarks
+	Addition	9+2 is 11	
		9.1+2.0 is 11.1	
-	Subtraction	9-2 is 7	
		9.1-2.0 is 7.1	
*	Multiplication	9*2 is 18	
		9.1*2.0 is 18.2	
/	Division	9/2 is 4.25	In Python3
		9.1/2.0 is 4.55	Real div.
//	Integer Division	9//2 is 4	
%	Remainder	9%2 is 1	

The // operator

- Also referred to as “integer division”
- Result is a whole integer (floor of real division)
  - But the type need not be int
  - the integral part of the real division
  - rounded towards minus infinity ($-\infty$)
- Examples

9//4 is 2	**(-1)//2 is -1**	**(-1)//(-2) is 0**
1//2 is 0	**1//(-2) is -1**	**9//4.5 is 2.0**

The % operator

- The remainder operator % returns the remainder of the result of dividing its first operand by its second.

9%4 is 1	(-1)%2 is 1	(-1)//(-2) is 0
9%4.5 is 0.0	1%(-2) is 1	1%0.6 is 0.4

Ideally: x == (x//y)*y + x %y

Conditional Statements

- In daily routine
  - If it is very hot, I will skip exercise.
  - If there is a quiz tomorrow, I will first study and then sleep. Otherwise I will sleep now.
  - If I have to buy coffee, I will go left. Else I will go straight.

if-else statement

- Compare two integers and print the min.

```
if  x < y:
        print (x)
else:
        print (y)
print ('is the minimum')
```

1. Check if x is less than y.
2. If so, print x
3. Otherwise, print y.

Indentation

- Indentation is **important** in Python
  - grouping of statement (block of statements)
  - no explicit brackets, e.g. { }, to group statements

```
x,y = 6,10

if x < y:
    print (x)
else:
    print (y)
    print        e min’)
```

skipped

Run the program

6

10

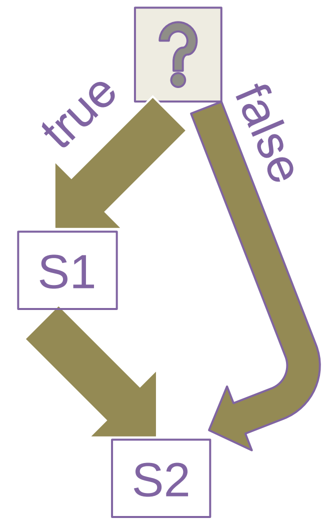
Output

6

if statement (no else!)

- General form of the if statement

```
if boolean-expr :
      S1
S2
```

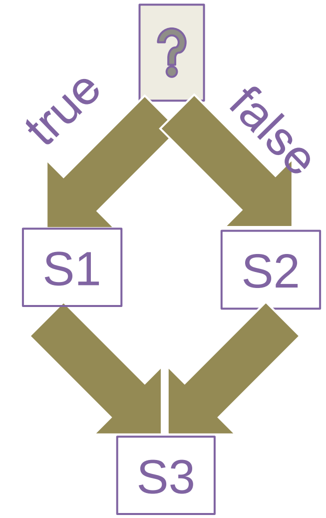


- Execution of if statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to the S2.
  - If expression evaluates to **false**, then control moves to the S2 directly.

if-else statement

- General form of the if-else statement

```
if boolean-expr :
        S1
else:
        S2
S3
```



- Execution of if-else statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to S3.
  - If expression evaluates to **false**, then S2 is executed and then control moves to S3.
  - S1/S2 can be **blocks** of statements!

Nested if, if-else

```
if a <= b:
       if a <= c:
           …
       else:
           …
else:
       if b <= c) :
           …
       else:
           …
```

Elif

- A special kind of nesting is the chain of if-else-if-else-… statements
- Can be written elegantly using if-elif-..-else

```
if cond1:
        s1
else:
    if cond2:
        s2
    else:
        if cond3:
            s3
        else:
            …
```

```
if cond1:
        s1
elif cond2:
    s2
elif cond3:
    s3
elif  …
else
    last-block-of-stmt
```

Summary of if, if-else

- if-else, nested if's, elif.
- Multiple ways to solve a problem
  - issues of readability, maintainability
  - and efficiency

Short-circuit Evaluation

- Do not evaluate the second operand of binary short-circuit logical operator if the result can be deduced from the first operand
  - Also applies to nested logical operators

true false false true

not((2>5) and (3/0 > 1)) or (4/0 < 2)

Evaluates to true

3 Factors for Expr Evaluation

- **Precedence**
  - Applied to two different class of operators
  - + and *, - and *, and and or, ...
- **Associativity**
  - Applied to operators of same class
  - * and *, + and -, * and /, ...
- **Order**
  - Precedence and associativity **identify the operands** for each operator
  - **Not which operand is evaluated first**
  - Python evaluates expressions from left to right
  - While evaluating an assignment, the right-hand side is evaluated before the left-hand side.

Caution about Using Floats

- Representation of *real numbers* in a computer can not be exact
  - Computers have limited memory to store data
  - *Between any two distinct real numbers, there are infinitely many real numbers.*
- On a typical machine running Python, there are 53 bits of precision available for a Python float

Caution about Using Floats

- The value stored internally for the decimal number 0.1 is the binary fraction

0.000110011001100110011001100110011001100110011001100110011010

- Equivalent to decimal value

0.1000000000000000055511151231257827021181583404541015625

- Approximation is similar to decimal approximation 1/3 = 0.333333333...
- No matter how many digits you use, you have an approximation

Comparing Floats

- Because of the approximations, comparison of floats is not exact.
- **Solution?**
- Instead of

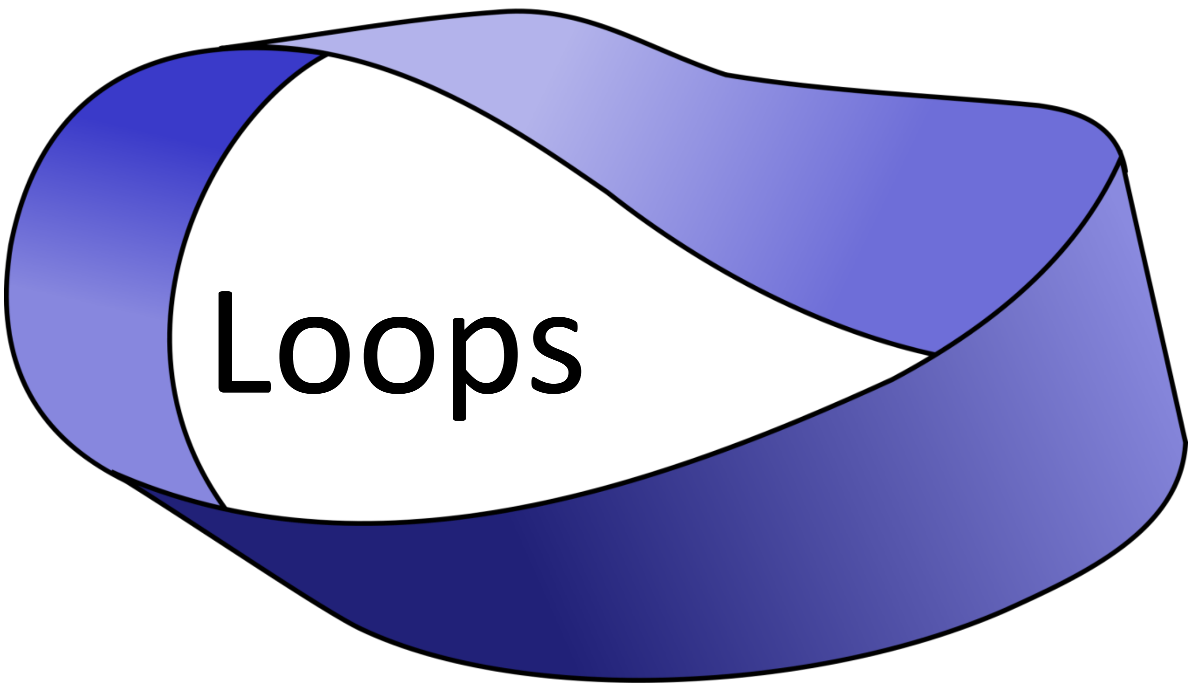
x == y

use

abs(x-y) <= epsilon

where epsilon is a suitably chosen small value

Programming using Python



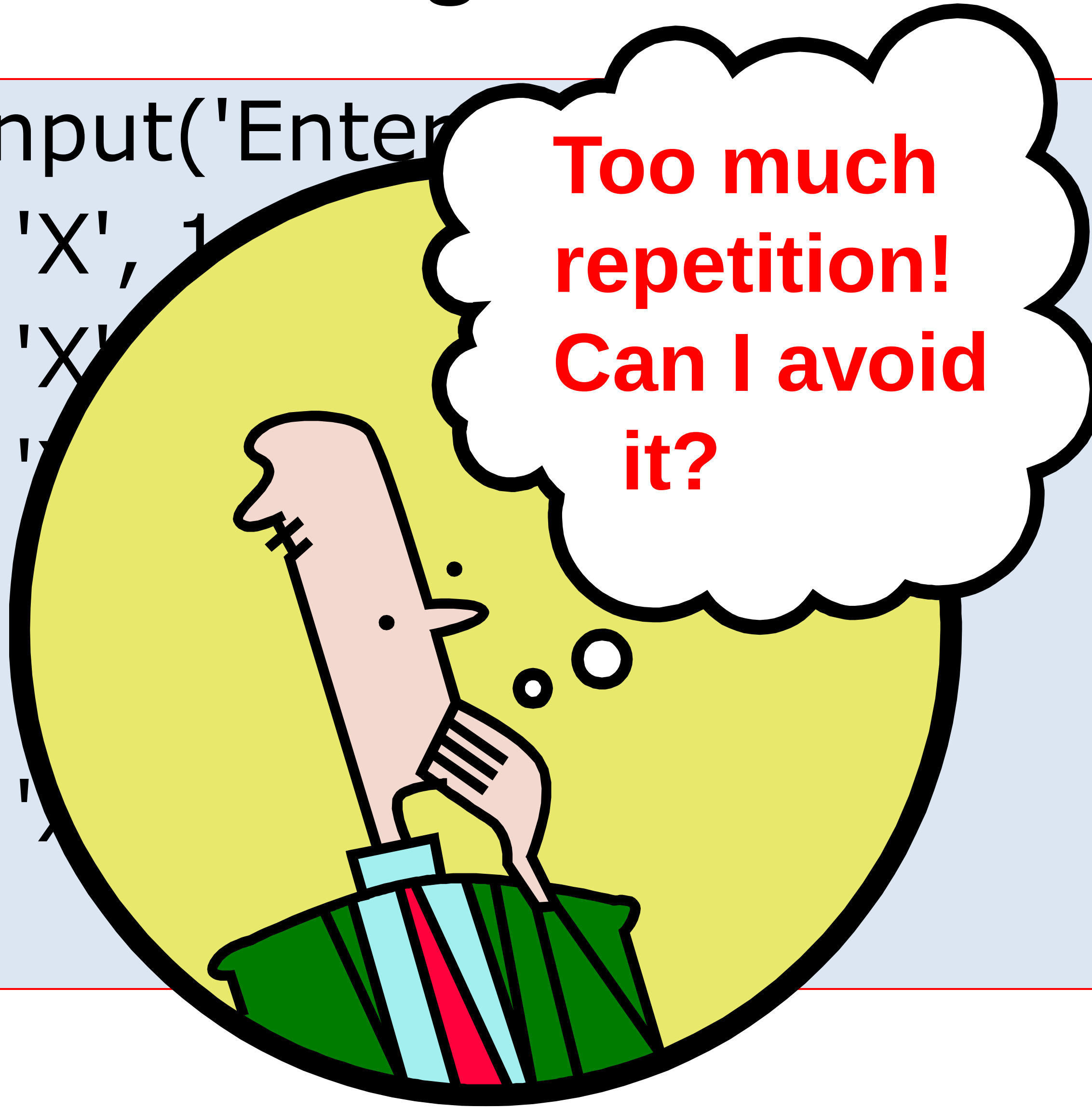
Loops

Printing Multiplication Table

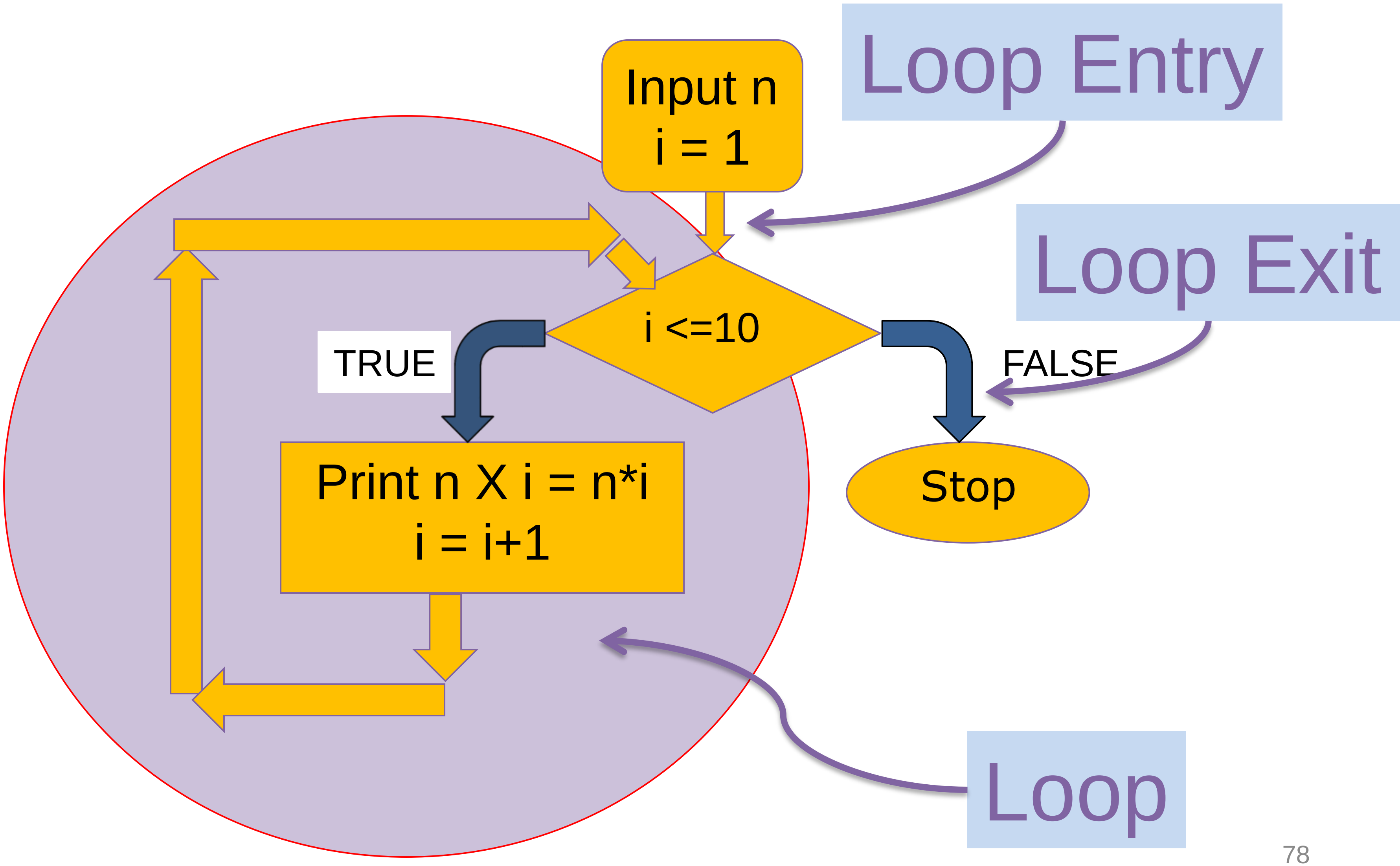
5	X	1	=	5
5	X	2	=	10
5	X	3	=	15
5	X	4	=	20
5	X	5	=	25
5	X	6	=	30
5	X	7	=	35
5	X	8	=	40
5	X	9	=	45
5	X	10	=	50

Program...

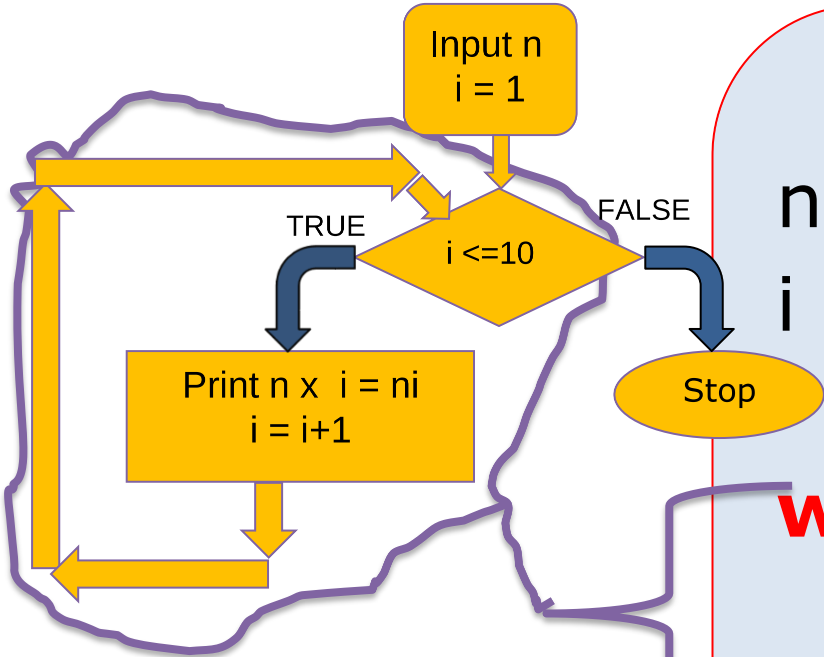
```
n = int(input('Ente
print (n, 'X', 1
print (n, 'X'
print (n, '
print (n,
print (n,
print (n, '
....
```



Printing Multiplication Table



Printing Multiplication Table

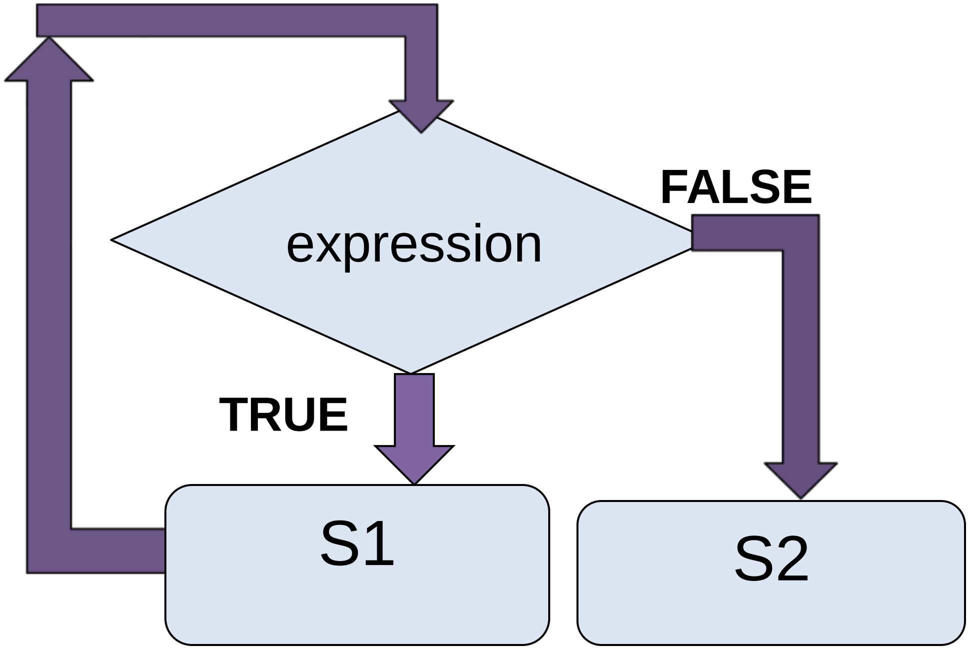



```
n = int(input('n=? '))
i = 1

while (i <= 10) :
    print (n ,'X', i, '=', n*i)
    i = i + 1
print ('done‘)
```

While Statement

```
while (expression):
        S1
S2
```



1. Evaluate expression
2. If TRUE then
   a) execute statement1
   b) goto step 1.
3. If FALSE then execute statement2.

For Loop

- Print the sum of the reciprocals of the first 100 natural numbers.

```
rsum=0.0# the reciprocal sum

# the for loop
for i in range(1,101):
    rsum = rsum + 1.0/i
print ('sum is', rsum)
```

For loop in Python

- General form

```
for variable in sequence:
    stmt
```

range

- range(s, e, d)
  - generates the list:

    [s, s+d, s+2*d, ..., s+k*d]

    where s+k*d < e <= s+(k+1)*d
- range(s, e) is equivalent to range(s, e, 1)
- range(e) is equivalent to range(0, e)

Exercise: What if d is negative? Use python interpreter to find out.

Continue and Update Expr

- Make sure continue does not bypass update-expression for while loops

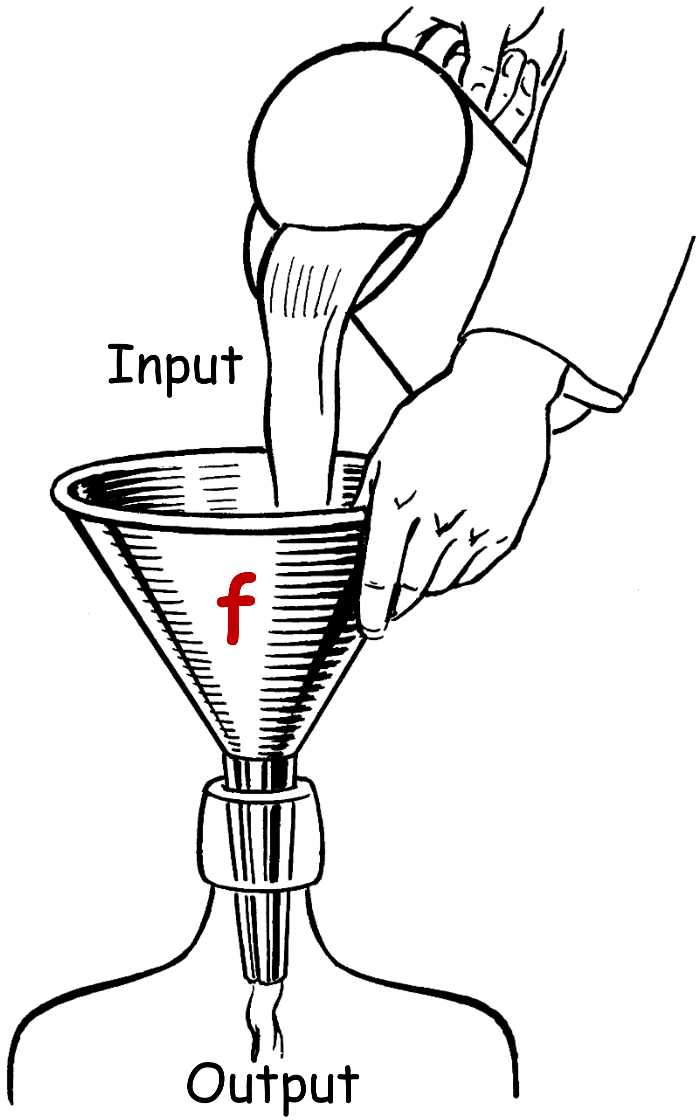
```
# print all odd numbers < 10
i = 1
while i <= 10:
    if i%2==0: # even
        continue
    print (i, end=' ')
    i = i+1
```

i is not incremented when even number encountered. Infinite loop!!

Programming using Python

f(unctions)

Parts of a function



```
def max(a, b):
   '''return maximum among a and b'''
   if (a > b):
      return a
   else:
      return b

x = max(6, 4)
```

keyword

Function Name

2 arguments a and b (formal args)

Body of thefunction, indented w.r.t the def keyword

Call to the function. Actual args are 6 and 4.

Documentation comment (**docstring**), type help <function-name> on prompt to get help for the function

```
def max (a, b):
   “‘‘return maximum among a and b’’’
   if (a > b):
       return a
  else:
       return b
```

```
In[3] : help(max)
Help on function max in module __main__:


max(a, b)
   return maximum among a and b
```

Keyword Arguments

```
def printName(first, last, initials) :
    if initials:
        print (first[0] + '. ' + last[0] + '.')
    else:
        print (first, last)
```

Note use of [0] to get the first character of a string. More on this later.

Call	Output
printName('Acads', 'Institute', False)	Acads Institute

Keyword Arguments

- Parameter passing where formal is bound to actual using formal's name
- Can mix keyword and non-keyword arguments
  - All non-keyword arguments precede keyword arguments in the call
  - Non-keyword arguments are matched by position (order is important)
  - Order of keyword arguments is not important

Default Values

```
def printName(first, last, initials=False) :
    if initials:
        print (first[0] + '. ' + last[0] + '.')
    else:
        print (first, last)
```

Note the use of "default" value

Call	Output
printName('Acads', 'Institute')	Acads Institute

Default Values

- Allows user to call a function with fewer arguments
- Useful when some argument has a fixed value for most of the calls
- All arguments with default values must be at the end of argument list
  - non-default argument can not follow default argument

Globals

- Globals allow functions to communicate with each other indirectly
  - Without parameter passing/return value
- Convenient when two seemingly “far-apart” functions want to share data
  - No *direct* caller/callee relation
- If a function has to update a global, it must re-declare the global variable with global keyword.

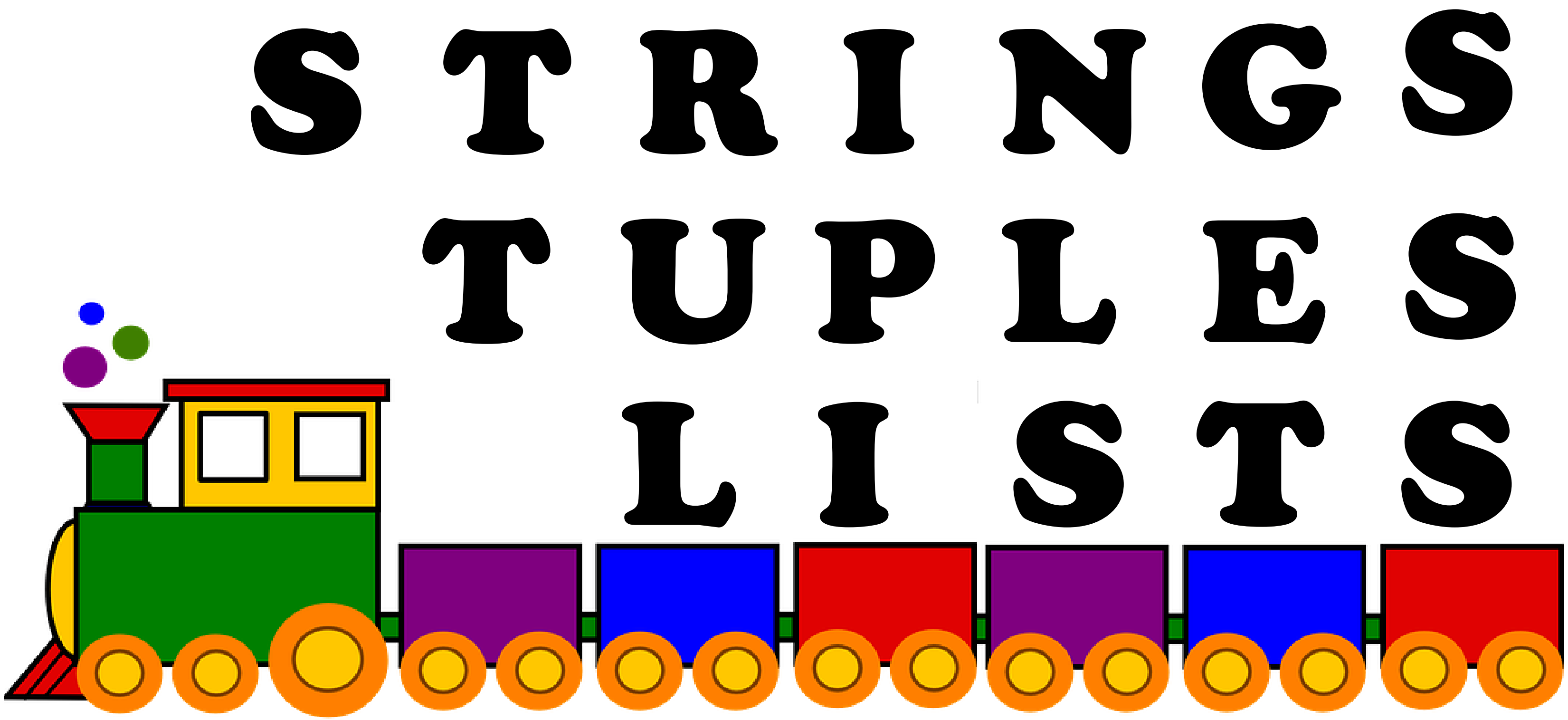
Globals

```
PI = 3.14
def perimeter(r):
    return 2 * PI * r
def area(r):
    return PI * r * r
def update_pi():
    global PI
    PI = 3.14159
```

```
>>> print(area (100))
31400.0
>>> print(perimeter(10))
62.800000000000004
>>> update_pi()
>>> print(area(100))
31415.999999999996
>>> print(perimeter(10))
62.832
```

defines PI to be of float type with value 3.14. PI can be used across functions. Any change to PI in update_pi will be visible to all due to the use of global.

Programming with Python



Strings

- Strings in Python have type str
- They represent sequence of characters
  - Python does not have a type corresponding to character.
- Strings are enclosed in single quotes(') or double quotes(“)
  - Both are equivalent
- Backslash (\) is used to escape quotes and special characters

Strings

```
>>> name='intro to python'
>>> descr='acad\'s first course'
```

- More readable when print is used

```
>>> print descr
acad's first course
```

Length of a String

- len function gives the length of a string

```
>>> name='intro to python'
>>> empty=''
>>> single='a'
```

\n is a **single** character: the special character representing newline

Concatenate and Repeat

- In Python, + and * operations have special meaning when operating on strings
  - + is used for concatenation of (two) strings
  - * is used to repeat a string, an int number of time
  - Function/Operator Overloading

Concatenate and Repeat

```
>>> details = name + ', ' + descr
>>> details
"intro to python, acad's first course"
```

Indexing

- Strings can be indexed
- First character has index 0

```
>>> name='Acads'
```

Indexing

- Negative indices start counting from the right
- Negatives indices start from -1
- -1 means last, -2 second last, ...

```
>>> name='Acads'
>>> name[-1]
's'
>>> name[-5]
'A'
>>> name[-2]
'd'
```

Indexing

- Using an index that is too large or too small results in "index out of range" error

Slicing

- To obtain a substring
- s[start:end] means substring of s starting at index start and ending at index end-1
- s[0:len(s)] is same as s
- Both start and end are optional
  - If start is omitted, it defaults to 0
  - If end is omitted, it defaults to the length of string
- s[:] is same as s[0:len(s)], that is same as s

Slicing

```
>>> name='Acads'
>>> name[0:3]
```

More Slicing

```
>>> name='Acads'
>>> name[-4:-1]
'cad'
>>> name[-4:]
'cads'
>>> name[-4:4]
'cad'
```

Understanding Indices for slicing

A	c	a	d	s	
0	1	2	3	4	5
-5	-4	-3	-2	-1	

Out of Range Slicing

A	c	a	d	s
0	1	2	3	4
-5	-4	-3	-2	-1

- Out of range indices are ignored for slicing
- when start and end have the same sign, if start >=end, empty slice is returned

Why?

Tuples

- A tuple consists of a number of values separated by commas

```
>>> t = 'intro to python', 'amey karkare', 101
```

- Empty and Singleton Tuples

Nested Tuples

- Tuples can be nested

- Note that course tuple is copied into student.
  - Changing course does not affect student

Length of a Tuple

- len function gives the length of a tuple

```
>>> course = 'Python', 'Amey', 101
>>> student = 'Prasanna', 34, course
>>> empty = ()
>>> singleton = 1,
>>> len(empty)
0
>>> len(singleton)
1
>>> len(course)
3
>>> len(student)
3
```

.

More Operations on Tuples

- Tuples can be concatenated, repeated, indexed and sliced

```
>>> 2*course1
('Python', 'Amey', 101, 'Python', 'Amey', 101)
```

Unpacking Sequences

- Strings and Tuples are examples of sequences
  - Indexing, slicing, concatenation, repetition operations applicable on sequences
- Sequence Unpacking operation can be applied to sequences to get the components
  - *Multiple assignment* statement
  - LHS and RHS must have equal length

Unpacking Sequences

```
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
>>> name,roll,regdcourse=student
>>> name
```

Lists

- Ordered sequence of values
- Written as a sequence of comma-separated values between square brackets
- Values can be of different types
  – usually the items all have the same type

```
>>> lst = [1,2,3,4,5]
>>> lst
[1, 2, 3, 4, 5]
>>> type(lst)
<type 'list'>
```

Lists

- List is also a sequence type
  - Sequence operations are applicable

Lists

- List is also a sequence type
  - Sequence operations are applicable

```
>>> [0] + fib # Concatenation
```

More Operations on Lists

- L.append(x)
- L.extend(seq)
- L.insert(i, x)
- L.remove(x)
- L.pop(i)
- L.pop()
- L.index(x)
- L.count(x)
- L.**sort**()
- L.**reverse**()

x is any value, seq is a sequence value (list, string, tuple, …),
i is an integer value

Mutable and Immutable Types

- Tuples and List types look very similar
- However, there is one major difference: Lists are **mutable**
  - Contents of a list can be modified
- Tuples and Strings are **immutable**
  - Contents can not be modified

Summary of Sequences

Operation	Meaning
seq[i]	i-th element of the sequence
len(seq)	Length of the sequence
seq1 + seq2	Concatenate the two sequences
num*seq seq*num	Repeat seq num times
seq[start:end]	slice starting from **start**, and ending at **end-1**
e in seq	True if e is present is seq, False otherwise
e not in seq	True if e is not present is seq, False otherwise
for e in seq	Iterate over all elements in seq (e is bound to one element per iteration)

Sequence types include String, Tuple and List.
Lists are mutable, Tuple and Strings immutable.

Programming with Python

Sets and Dictionaries

Sets

- An unordered collection with no duplicate elements
- Supports
  - membership testing
  - eliminating duplicate entries
  - Set operations: union, intersection, difference, and symmetric difference.

Sets

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'o
range', 'banana']
>>> fruits = set(basket)
```

Create a set from a sequence

Set Operations

```
>>> A=set('acads')
>>> B=set('institute')
>>> A
{'a', 's', 'c', 'd'}
>>> B
{'e', 'i', 'n', 's', 'u', 't'}
```

Dictionaries

- Unordered set of *key:value* pairs,
- Keys have to be unique and immutable
- Key:value pairs enclosed inside curly braces {...}
- Empty dictionary is created by writing {}
- Dictionaries are mutable
  - add new key:value pairs,
  - change the pairing
  - delete a key (and associated value)

Operations on Dictionaries

Operation	Meaning
len(d)	Number of key:value pairs in d
d.keys()	List containing the keys in d
d.values()	List containing the values in d
k in d	True if key k is in d
d[k]	Value associated with key k in d
d.get(k, v)	If k is present in d, then d[k] else v
d[k] = v	Map the value v to key k in d (replace d[k] if present)
del d[k]	Remove key k (and associated value) from d
for k in d	Iterate over the keys in d

Operations on Dictionaries

```
>>> capital = {'India':'New Delhi', 'USA':'Washington DC', 'France':'Paris', 'Sri Lanka':'Colombo'}
```

Dictionary Construction

- The dict constructor: builds dictionaries directly from *sequences of key-value pairs*

```
>>> airports=dict([('Mumbai', 'BOM'), ('Delhi', 'Del
'),('Chennai', 'MAA'), ('Kolkata', 'CCU')])
>>> airports
{'Kolkata': 'CCU', 'Chennai': 'MAA', 'Delhi': 'Del',
'Mumbai': 'BOM'}
```

Programming with Python

File I/O

File I/O

- Files are persistent storage
- Allow data to be stored beyond program lifetime
- The basic operations on files are
  - open, close, read, write
- Python treat files as sequence of lines
  - sequence operations work for the data read from files

File I/O: open and close

open(filename, mode)

- While opening a file, you need to supply
  - The name of the file, including the path
  - The mode in which you want to open a file
  - Common modes are r (read), w (write), a (append)
- Mode is optional, defaults to r
- open(..) returns a file object
- close() on the file object closes the file
  - finishes any buffered operations

File I/O: Example

```
>>> players = open('tennis_players', 'w')
>>>
>>>
>>>
>>>
>>>
>>> players.close() # done with writing
```

- Do some writing
- How to do it?
  - see the next few slides

File I/O: read, write and append

- Reading from an open file returns the contents of the file
  - as **sequence** of lines in the program
- Writing to a file
  - IMPORTANT: If opened with mode 'w', **clears** the existing contents of the file
  - Use append mode ('a') to preserve the contents
  - Writing happens at the end

File I/O: Examples

```
>>> players = open('tennis_players', 'w')
```

```
>>> players.close() # done with writing
```

File I/O: Examples

```
>>> print(players)
```

```
>>> pn = n.read() # read all players
```

File I/O: Examples

```
>>> n = open('tennis_players', 'r')
>>> c = open('tennis_countries', 'r')
```

of for ... in

File I/O: Examples

Programming using Python

Modules and Packages

Modules

- As program gets longer, need to organize them for easier access and easier maintenance.
- Reuse same functions across programs without copying its definition into each program.
- Python allows putting definitions in a file
  - use them in a script or in an interactive instance of the interpreter
- Such a file is called a *module*
  - definitions from a module can be *imported* into other modules or into the *main* module

Modules

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module's name is available in the global variable __name__.

Modules Example

fib.py - C:\

fib.py - C:\Users\karkare\Google Drive\IITK\Courses\2016Python\Programs\fib.py (2.7.12)

File Edit Format Run Options Window Help

```python
# Module for fibonacci numbers

def fib_rec(n):
    '''recursive fibonacci'''
    if (n <= 1):
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

Modules Example

```
def fib_rec(n):
    '''recursive fibonacci'''
    if (n <= 1):
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)

def fib_iter(n):
    '''iterative fibonacci'''
    cur, nxt = 0, 1
    for k in range(n):
        cur, nxt = nxt, cur+nxt
    return cur

def fib_upto(n):
    '''given n, return list of fibonacci
    numbers <= n'''
    cur, nxt = 0, 1
    lst = []
    while (cur < n):
        lst.append(cur)
        cur, nxt = nxt, cur+nxt
    return lst
```

```
>>> import fib
>>> fib.fib_upto(5)
[0, 1, 1, 2, 3]
>>> fib.fib_rec(10)
55
>>> fib.fib_iter(20)
6765
>>> fib.__name__
'fib'
```

Within a module, the module's name is available as the value of the global variable `__name__`.

Importing Specific Functions

- To import specific functions from a module

- This brings only the imported functions in the current symbol table
  - No need of modulename. (absence of `fib.` in the example)

Importing ALL Functions

- To import *all* functions from a module, in the current symbol table

```
>>> from fib import *
>>> fib_upto(6)
[0, 1, 1, 2, 3, 5]
>>> fib_iter(8)
21
```

- This imports all names except those beginning with an underscore (_).

`__main__` in Modules

- When you run a module on the command line with

  `python fib.py <arguments>`

  the code in the module will be executed, just as if you imported it, but with the `__name__` set to "__main__".
- By adding this code at the end of your module

```
if __name__ == "__main__":
    ... # Some code here
```

  you can make the file usable as a script as well as an importable module

`__main__` in Modules

```
if __name__ == "__main__":
    import sys
    print (fib_iter(int(sys.argv[1])))
```

- This code parses the command line only if the module is executed as the “main” file:

```
$ python fib.py 10
55
```

- If the module is imported, the code is not run:

```
>>> import fib
>>>
```

Package

- A Python package is a collection of Python modules.
- Another level of *organization.*
- *Packages* are a way of structuring Python's module namespace by using *dotted module names*.
  - The module name A.B designates a submodule named B in a package named A.
  - The use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

A **sound** Package

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

https://docs.python.org/3/tutorial/modules.html

A **sound** Package

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

What are these files with funny names?

https://docs.python.org/3/tutorial/modules.html

`__init.py__`

- The `__init__.py` files are required to make Python treat directories containing the file as packages.
- This prevents directories with a common name, such as `string`, unintentionally hiding valid modules that occur later on the module search path.
- `__init__.py` can just be an empty file
- It can also execute initialization code for the package

Importing Modules from Packages

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

https://docs.python.org/3/tutorial/modules.html

Importing Modules from Packages

```
import sound.effects.echo
```

- Loads the submodule `sound.effects.echo`
- It must be referenced with its full name:

```
sound.effects.echo.echofilter(
    input, output,
    delay=0.7, atten=4
)
```

Importing Modules from Packages

```
from sound.effects import echo
```

- This also loads the submodule `echo`
- Makes it available without package prefix
- It can be used as:

```
echo.echofilter(
    input, output,
    delay=0.7, atten=4
)
```

Importing Modules from Packages

```
from sound.effects.echo import echofilter
```

- This loads the submodule `echo`, but this makes its function `echofilter()` directly available.

```
echofilter(input, output,
              delay=0.7, atten=4)
```

Popular Packages

- pandas, numpy, scipy, matplotlib, …
- Provide a lot of useful functions

Reference

- John V. Guttag, "Introduction to Computation & Programming using Python", 2nd Edition, The MIT Press. 2016
- Allen B. Downey, "Think Python: How to Think Like a Computer Scientist". 2nd edition, Updated for Python 3. Shroff O'Reilly Publishers, 2016.
- Guido van Rossum and Fred L. Drake Jr., “An Introduction to Python — Revised and updated for Python 3.2". Network Theory Ltd., 2011.

Acknowledgement

Image Source from Websites and Python Tutorials

Content Source from Books/ Website/ Handout