Course code	PROBLEM SOLVING AND PROGRAMMING	L	T	P	J	C
CSE1001		0	0	6	0	3
Pre-requisite	NIL	Syllabus version				
		v1.0				

Course Objectives:

1. To develop broad understanding of computers, programming languages and their generations
2. Introduce the essential skills for a logical thinking for problem solving
3. To gain expertise in essential skills in programming for problem solving using computer

Expected Course Outcome:

1. Understand the working principle of a computer and identify the purpose of a computer programming language.
2. Learn various problem solving approaches and ability to identify an appropriate approach to solve the problem
3. Differentiate the programming Language constructs appropriately to solve any problem
4. Solve various engineering problems using different data structures
5. Able to modulate the given problem using structural approach of programming
6. Efficiently handle data using flat files to process and store data for the given problem

Student Learning Outcomes (SLO): 1, 12, 14

1. Having an ability to apply mathematics and science in engineering applications
12. Having adaptive thinking and adaptability
14. Having an ability to design and conduct experiments, as well as to analyze and interpret data

List of Challenging Experiments (Indicative)

1	Steps in Problem Solving Drawing flowchart using yEd tool/Raptor Tool	4 Hours
2	Introduction to Python, Demo on IDE, Keywords, Identifiers, I/O Statements	4 Hours
3	Simple Program to display Hello world in Python	4 Hours
4	Operators and Expressions in Python	4 Hours
5	Algorithmic Approach 1: Sequential	4 Hours
6	Algorithmic Approach 2: Selection (if, elif, if.. else, nested if else)	4 Hours
7	Algorithmic Approach 3: Iteration (while and for)	6 Hours
8	Strings and its Operations	6 Hours
9	Regular Expressions	6 Hours
10	List and its operations	6 Hours
11	Dictionaries: operations	6 Hours
12	Tuples and its operations	6 Hours
13	Set and its operations	6 Hours
14	Functions, Recursions	6 Hours
15	Sorting Techniques (Bubble/Selection/Insertion)	6 Hours
16	Searching Techniques : Sequential Search and Binary Search	6 Hours
17	Files and its Operations	6 Hours
	Total hours:	**90 hours**

Text Book(s)

1. John V. Guttag., 2016. Introduction to computation and programming using python: with applications to understanding data. PHI Publisher.

Reference Books

1. Charles Severance.2016.Python for everybody: exploring data in Python 3, Charles Severance.
2. Charles Dierbach.2013.Introduction to computer science using python: a computational problem-solving focus. Wiley Publishers.

Mode of Evaluation: **PAT / CAT / FAT**

Recommended by Board of Studies | 04-04-2014

Introduction to Python

Need of programming Languages

- Computers can execute tasks very rapidly and assist humans
- Programming languages – Helps for communication between human and machines.
- They can handle a greater amount of input data.
- But they cannot design a strategy to solve problems for you

Python – Why?

- Simple syntax
- Programs are clear and easy to read
- Has powerful programming features
- Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA.
- Python is well supported and freely available at www.python.org.

Python – Why?

- Guido van Rossum – Creator
- Released in the early 1990s.
- Its name comes from a 1970s British comedy sketch television show called *Monty Python's Flying Circus.*

Python….

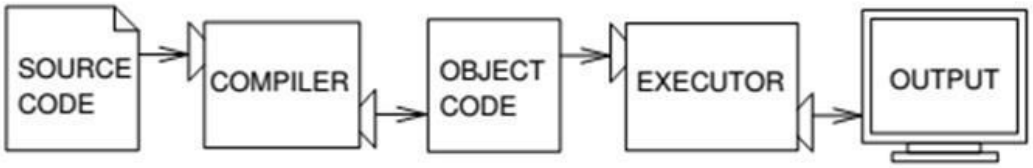
- **High-level language**; other high-level languages you might have heard of are C, C++, Perl, and Java.
- There are also **low-level languages**, sometimes referred to as "machine languages" or "assembly languages."
- Computers can only run programs written in low-level languages.
- So programs written in a high-level language have to be processed before they can run.

Python….

- Two kinds of program translator to convert from high-level languages into low-level languages:
  - **Interpreters**
  - **Compilers**.
- An interpreter processes the program by reading it line by line

Python….

- Compiler translates completely a high level program to low level it completely before the program starts running.
- High-level program is called **source code**
- Translated program is called the **object code** or the **executable**.



A compiler translates source code into object code, which is run by a hardware executor.

Python….

- Python is considered an interpreted language because Python programs are executed by an interpreter.
- There are two ways to use the interpreter:

  interactive mode and **script mode**.

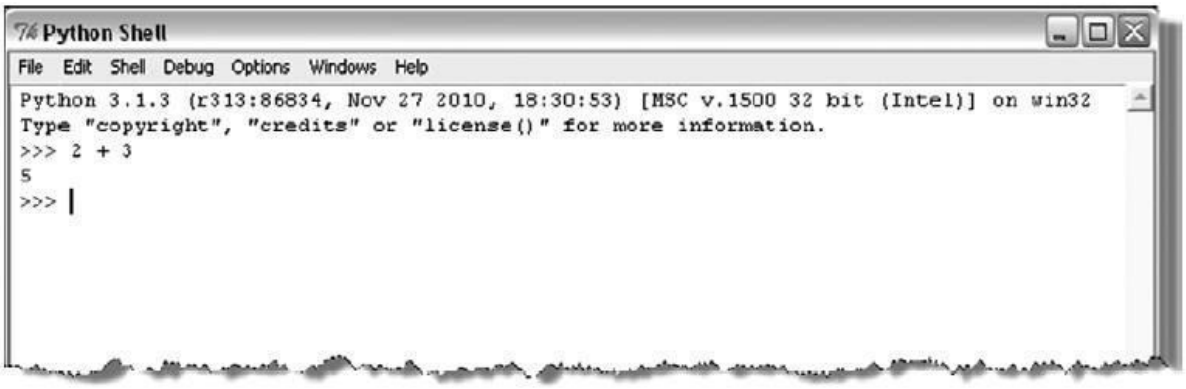
Python….Interactive mode

In interactive mode, you type Python programs and the interpreter displays the result:

```
>>> 1 + 1
2
```

The shell prompt, >>>, is the **prompt** the interpreter uses to indicate that it is ready. If you type 1 + 1, the interpreter replies 2.

Python …. Shell



Python…. Script Mode

- Can also store code in a file and use the interpreter to execute the contents of the file, which is called a **script**.
- Python scripts have names that end with .py.
- Interactive mode is convenient for testing small pieces of code because you can type and execute them immediately.
- But for anything more than a few lines, should save your code as a script so you can modify and execute it in future.

Python....Script Mode …

File name : first.py

```
print(4+3)
print(4-3)
print(4>3)
print("hello World")
```

```
C:\Python34\python.exe C:/Users/sathisbsk/first.py
7
1
True
hello World

Process finished with exit code 0
```


Problem

- Little Bob loves chocolate, and he goes to a store with Rs. *N* in his pocket. The price of each chocolate is Rs. *C*. The store offers a discount: for every *M* wrappers he gives to the store, he gets one chocolate for free. This offer is available only once. How many chocolates does Bob get to eat?

PAC For Chocolate Problem

Input	Processing	Output
Amount in hand, N Price of one chocolate, C Number of wrappers for a free chocolate, M	Number of Chocolates P = Quotient of N / C Free chocolate F = Quotient of P/M	Total number of chocolates got by Bob

1A-14

Pseudocode

- READ N and C
- COMPUTE num_of_chocolates as N/C
- CALCULATE returning_wrapper as number of chocolates/m
- TRUNCATE decimal part of returning_wrapper
- COMPUTE Chocolates_recieved as num_of_chocolates + returning_wrapper
- PRINT Chocolates_recieved

Knowledge Required

- Following knowledge is required in Python to write a code to solve the above problem
- Read input from user
- Data types in Python
- Perform arithmetic calculations
- Write output

What is an Identifier?

- An **identifier** is a sequence of one or more characters used to name a given program element.
- In Python, an identifier may contain letters and digits, but cannot begin with a digit.
- Special underscore character can also be used
- Example : line, salary, emp1, emp_salary

Rules for Identifier

- Python is *case sensitive*, thus, Line is different from line.
- Identifiers may contain letters and digits, but cannot begin with a digit.
- The underscore character, _, is also allowed to aid in the readability of long identifier names. It should not be used as the *first* character
- Spaces are not allowed as part of an identifier.

Identifier Naming

Valid Identifiers	Invalid Identifiers	Reason Invalid
totalSales	'totalSales'	quotes not allowed
totalsales	total sales	spaces not allowed
salesFor2010	2010Sales	cannot begin with a digit
sales_for_2010	_2010Sales	should not begin with an underscore

Keywords

- A **keyword** is an identifier that has pre-defined meaning in a programming language.
- Therefore, keywords cannot be used as “regular” identifiers. Doing so will result in a syntax error, as demonstrated in the attempted assignment to keyword and below,

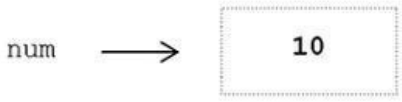
```
>>> and = 10
SyntaxError: invalid syntax
```


Keywords in Python

and	as	assert	break	class	continue	def
del	elif	else	except	finally	for	from
global	if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try	while
with	yield	false	none	true		

Variables and Identifiers

- A **variable** is a name (identifier) that is associated with a value.
- A simple description of a variable is “a name that is assigned to a value,”



Variables are assigned values by use of the **assignment operator**,
num = 10
num = num + 1

Comments

- Meant as documentation for anyone reading the code
- Single-line comments begin with the hash character ("#") and are terminated by the end of line.
- Python ignores all text that comes after # to end of line
- Comments spanning more than one line are achieved by inserting a multi-line string (with """ as the delimiter one each end) that is not used in assignment or otherwise evaluated, but sits in between other statements.
- *#This is also a comment in Python*
- """ This is an example of a multiline comment that spans multiple lines ... """”

Literals

- A **literal** is a sequence of one or more characters that stands for itself.

Numeric literal

- A **numeric literal** is a literal containing only the digits 0–9, an optional sign character (1 or 2), and a possible decimal point. (The letter *e* is also used in exponential notation).
- If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or "**float**" (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10).
- *Commas are never used in numeric literals*

Numeric Literals in Python

Numeric Literals							
integer values	floating-point values					incorrect	
5	5.	5.0	5.125	0.0005	5000.125	5,000.125	
2500	2500.		2500.0	2500.125		2,500	2,500.125
+2500	+2500.		+2500.0	+2500.125		+2,500	+2,500.125
-2500	-2500.		-2500.0	-2500.125		-2,500	-2,500.125

Since numeric literals without a provided sign character denote positive values, an explicit positive sign character is rarely used.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> 1024
???

>>> -1024
???

>>> .1024
???

>>> 1,024
???

>>> 0.1024
???

>>> 1,024.46
???
```

String Literals

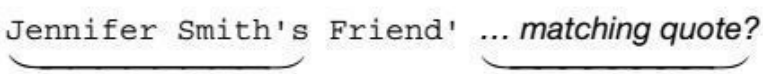
- **String literals**, or "**strings**," represent a sequence of characters,
  'Hello' 'Smith, John' "Baltimore, Maryland 21210"
- In Python, string literals may be surrounded by a matching pair of either single (') or double (") quotes.
- >>> print('Welcome to Python!')
  >>>Welcome to Python!

String Literal Values

`'A'`	- a string consisting of a single character
`'jsmith16@mycollege.edu'`	- a string containing non-letter characters
`"Jennifer Smith's Friend"`	- a string containing a single quote character
`' '`	- a string containing a single blank character
`''`	- the empty string

Note

If this string were delimited with single quotes, the apostrophe (single quote) would be considered the matching closing quote of the opening quote, leaving the last final quote unmatched,



Thus, Python allows the use of more than one type of quote for such situations. (The convention used in the text will be to use single quotes for delimiting strings, and only use double quotes when needed.)

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> print('Hello')
???
>>> print('Hello")
???
>>> print('Let's Go')
???
>>> print("Hello")
???
>>> print("Let's Go!')
???
>>> print("Let's go!")
???
```

Control Characters

- **S**pecial characters that are not displayed on the screen. Rather, they *control* the display of output
- Do not have a corresponding keyboard character
- Therefore, they are represented by a combination of characters called an *escape sequence*.
- The backslash (\) serves as the escape character in Python.
- For example, the escape sequence '\n', represents the *newline control character*, used to begin a new screen line

```
print('Hello\nJennifer Smith')
```

which is displayed as follows,

```
Hello
Jennifer Smith
```


LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> print('Hello World')
???

>>> print('Hello World\n')
???

>>> print('Hello World\n\n')
???

>>> print('\nHello World')
???

>>> print('Hello\nWorld')
???

>>> print('Hello\n\nWorld')
???

>>> print(1, '\n', 2, '\n', 3)
???

>>> print('\n', 1, '\n', 2, '\n', 3)
???
```

Data Types

- Python's data types are built in the core of the language
- They are easy to use and straightforward.
- Data types supported by Python
  - **Boolean values**
  - **None**
  - **Numbers**
  - **Strings**
  - **Tuples**
  - **Lists**
  - **Sets**

Boolean values

- Primitive datatype having one of two values: True or False
- some common values that are considered to be True or False

Boolean values

print bool(True)	True
print bool(False)	False
print bool("text")	True
print bool("")	False
print bool(' ')	True
print bool(0)	False
print bool()	False
print bool(3)	True
print bool(None)	False

None

- Special data type - None
- Basically, the data type means non existent, not known or empty
- Can be used to check for emptiness

Numbers

Types of numbers supported by Python:

- Integers
- floating point numbers
- complex numbers
- Fractional numbers

Integers

- Integers have no fractional part in the number
- Integer type automatically provides extra precision for large numbers like this when needed.
- >>> a = 10
- >>> b = a

Binary, Octal and Hex Literals

- 0b1, 0b10000, 0b11111111 # Binary literals: base 2, digits 0-1
- 0o1, 0o20, 0o377
- # Octal literals: base 8, digits 0-7
- 0x01, 0x10, 0xFF # Hex literals: base 16, digits 0-9/A-F
- (1, 16, 255)

Conversion between different bases

- Provides built-in functions that allow you to convert integers to other bases' digit strings
- oct(64), hex(64), bin(64)
- # Numbers=>digit strings ('0o100', '0x40', '0b1000000')
- These literals can produce arbitrarily long integers

Numbers can be very long

- >>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF
- >>> X
  5192296858534827628530496329220095
- >>> oct(X)
  '0o1777777777777777777777777777777777'
- >>> bin(X)
  '0b11111111111111111111111111111111111111111111111111111111111111'
- ...and so on... 11111'

Floating Point Numbers

- Number with fractional part
- >>> 3.1415 * 2
- >>>6.283

Floating Point Numbers

Table 5-1. Numeric literals and constructors

Literal	Interpretation
1234, -24, 0, 99999999999999	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals in 3.X
0177, 0o177, 0x9ff, 0b101010	Octal, octal, hex, and binary literals in 2.X
3+4j, 3.0+4.0j, 3J	Complex number literals
set('spam'), {1, 2, 3, 4}	Sets: 2.X and 3.X construction forms
Decimal('1.0'), Fraction(1, 3)	Decimal and fraction extension types
bool(X), True, False	Boolean type and constants

Arithmetic overflow

- a condition that occurs when a calculated result is too large in magnitude (size) to be represented,
  >>>1.5e200 * 2.0e210
  >>> inf
  This results in the special value inf (“infi nity”) rather than the arithmetically correct result 3.0e410, indicating that arithmetic overflow has occurred.

Arithmetic underflow

- a condition that occurs when a calculated result is too small in magnitude to be represented,
  >>>1.0e-300 / 1.0e100
  >>>0.0
- This results in 0.0 rather than the arithmetically correct result 1.0e-400, indicating that arithmetic underflow has occurred.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> 1.2e200 * 2.4e100
???

>>> 1.2e200 * 2.4e200
???

>>> 1.2e200 / 2.4e100
???

>>> 1.2e-200 / 2.4e200
???
```

Arithmetic overflow occurs when a calculated result is too large in magnitude to be represente
Arithmetic underflow occurs when a calculated result is too small in magnitude to be represer

Repeated Print

```
>>>print('a'*15)
# prints 'a' fifteen times

>>>print('\n'*15)
# prints new line character fifteen times
```

Complex Numbers

- A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.
- **complex(x)** to convert x to a complex number with real part x and imaginary part zero
- **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y.
- x and y are numeric expressions

Complex Numbers

```
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
```

- A = 1+2j; B=3+2j
- # Mutpile statements can be given in same line using semicolon
- C = A+B; print(C)

Complex Numbers

prints real part of the number

- print(A.real)

prints imaginary part of the number

- print(A.imag)

Can do operations with part of complex number

- print(A.imag+3)

Input and output function

Input function : input

Basic_pay = input('Enter the Basic Pay: ')

Output function : print

print('Hello world!')

print('Net Salary', salary)

By Default…

- Input function reads all values as strings, to convert them to integers and float, use the function int() and float()

Type conversion…

```
line = input('How many credits do you have?')
num_credits = int(line)
line = input('What is your grade point average?')
gpa = float(line)
```

Here, the entered number of credits, say '24', is converted to the equivalent integer value, 24, before being assigned to variable num_credits. For input of the gpa, the entered value, say '3.2', is converted to the equivalent floating-point value, 3.2. Note that the program lines above could be combined as follows,

```
num_credits = int(input('How many credits do you have? '))
gpa = float(input('What is your grade point average? '))
```

Assignment Statement

Statement	Type
spam = 'Spam'	Basic form
spam, ham = 'yum', 'YUM'	Tuple assignment (positional)
[spam, ham] = ['yum', 'YUM']	List assignment (positional)
a, b, c, d = 'spam'	Sequence assignment, generalized
a, *b = 'spam'	Extended sequence unpacking (Python 3.X)
spam = ham = 'lunch'	Multiple-target assignment
a += 42	Augmented assignment (equivalent to a = a + 42)

Range

```
>>>a,b,c = range(1,4)
>>>a
1
>>>b
2
```

```
>>> S = "spam" >>> S += "SPAM"  # Implied concatenation
 >>> S
'spamSPAM'
```

Assignment is more powerful in Python

```
>>> nudge = 1
>>> wink  = 2
>>> nudge, wink = wink, nudge
# Tuples: swaps values
# Like T = nudge; nudge = wink; wink = T
>>> nudge, wink
(2, 1)
```


Operators and Expressions in Python

Basic Arithmetic operators in Python

Command	Name	Example	Output
+	Addition	4 + 5	9
-	Subtraction	8 - 5	3
*	Multiplication	4 * 5	20
/	True Division	19 / 3	6.3333
//	Integer Division	19//3	6
%	Remainder (modulo)	19 % 3	1
**	Exponent	2 ** 4	16

Order of Operations

Remember that thing called order of operations that they taught in maths? Well, it applies in Python, too. Here it is, if you need reminding:

1.parentheses ()

2.exponents **

3.multiplication *, division \, and remainder %

4.addition + and subtraction -

Order of Operations

Operator	Operation	Precedence
()	parentheses	0
**	exponentiation	1
*	multiplication	2
/	division	2
//	int division	2
%	remainder	2
+	addition	3
-	subtraction	3

- The computer scans the expression from left to right,
- first clearing parentheses,
- second, evaluating exponentiations from left to right in the order they are encountered
- third, evaluating *, /, //, % from left to right in the order they are encountered,
- fourth, evaluating +, - from left to right in the order they are encountered

2**3+2*(2+3)

- 18

Example 1 – Order of operations

```
>>> 1 + 2 * 3
 7
 >>> (1 + 2) * 3
 9
```

- In the first example, the computer calculates 2 * 3 first, then adds 1 to it. This is because multiplication has the higher priority (at 3) and addition is below that (at a lowly 4).
- In the second example, the computer calculates 1 + 2 first, then multiplies it by 3. This is because parentheses have the higher priority (at 1), and addition comes in later than that.

Example 2 – Order of operations

Also remember that the math is calculated from left to right, *unless* you put in parentheses. The innermost parentheses are calculated first. Watch these examples:

```
>>> 4 - 40 - 3
-39
>>> 4 - (40 - 3)
 -33
```

- In the first example, 4 - 40 is calculated, then - 3 is done.
- In the second example, 40 - 3 is calculated, then it is subtracted from 4.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> num = input('Enter number: ')
Enter number: 5
???
```

```
>>> num = input('Enter name: ')
Enter name: John
???
```

```
>>> num = int(input('Enter number: '))
Enter number: 5
???
```

```
>>> num = int(input('Enter name: '))
Enter name: John
???
```

Quotation in Python

- Python accepts single ('), double (") and triple ('" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes are used to span the string across multiple lines. For example, all the following are legal −
- word = 'word'
- sentence = "This is a sentence."
- paragraph = """This is a paragraph. It is made up of multiple lines and sentences."""

Built-in format Function

- Because floating-point values may contain an arbitrary number of decimal places, the built-in **format** function can be used to produce a numeric string version of the value containing a specific number of decimal places.

```
>>> 12/5
2.4
>>> format(12/5, '.2f')
'2.40'
```

```
>>> 5/7
0.7142857142857143
>>> format(5/7, '.2f')
'0.71'
```

- In these examples, *format specifier* '.2f' rounds the result to two decimal places of accuracy in the string produced.

- For very large (or very small) values 'e' can be used as a format specifier,

```
>>> format(2 ** 100, '.6e')
'1.267651e+30'
```


LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> format(11/12, '.2f')
???
>>> format(11/12, '.2e')
???
>>> format(11/12, '.3f')
???
>>> format(11/12, '.3e')
???
```

Python is a Dynamic Type language

- Same variable can be associated with values of different type during program execution, as indicated below.
- It's also very dynamic as it rarely uses what it knows to limit variable usage

```
var = 12          integer
var = 12.45       float
var = 'Hello'     string
```


LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> num = 10
>>> num
???
>>> id(num)
???

>>> num = 20
>>> num
???
>>> id(num)
???

>>> k = num
>>> k
???
>>> id(k)
???
>>> id(num)
???
```

```
>>> k = 30
>>> k
???
>>> num
???
>>> id(k)
???
>>> id(num)
???

>>> k = k + 1
>>> k
???
>>> id(num)
???
>>> id(k)
???
```

Bitwise Operations

- This includes operators that treat integers as strings of binary bits, and can come in handy if your Python code must deal with things like network packets, serial ports, or packed binary data
- >>> x = 1 # 1 decimal is 0001 in bits
  >>> x << 2 # Shift left 2 bits: 0100
- 4

- >>> x | 2 # Bitwise OR (either bit=1): 0011
- 3
- >>> x & 1 # Bitwise AND (both bits=1): 0001
  1
- In the first expression, a binary 1 (in base 2, 0001) is shifted left two slots to create a binary 4 (0100).
- The last two operations perform a binary OR to combine bits (0001| 0010 = 0011) and a binary AND to select common bits (0001&0001 = 0001).

- To print in binary format use bin function:
- >>> X = 0b0001 # Binary literals
- >>> X << 2 # Shift left 4
- >>> bin(X << 2) # Binary digits string '0b100‘
- >>> bin(X | 0b010) # Bitwise OR: either '0b11'
- >>> bin(X & 0b1) # Bitwise AND: both '0b0'

Logical Operators

Assume a = 10 and b = 20

Operator	Description	Example
and	If both the operands are true then condition becomes true.	(a and b) is true.
Or	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not	Used to reverse the logical state of its operand.	Not(a and b) is false.

Python is a Strongly Typed language

- interpreter keeps track of all variable types
- Check type compatability while expressions are evaluated
- >>> 2+3 # right
- >>>”two”+1 # Wrong!!

Table 11-2. Augmented assignment statements

X += Y	X &= Y	X -= Y	X |= Y
X *= Y	X ^= Y	X /= Y	X >>= Y
X %= Y	X <<= Y	X **= Y	X //= Y

Python Program for Bob Problem

```
n = float(input("Enter amount in hand"))
c = float(input("Enter price of one chocolate"))
m = int(input("Enter num of wrapper for free chocolate"))
#compute number of chocolates bought
p = n//c
#compute number of free chocolates
f = p//m
print("Number of chocolates",int(p+f))
```


Problem -1

ABC company Ltd. is interested to computerize the pay calculation of their employee in the form of Basic Pay, Dearness Allowance (DA) and House Rent Allowance (HRA). DA and HRA are calculated as certain % of Basic pay(For example, DA is 80% of Basic Pay, and HRA is 30% of Basic pay). They have the deduction in the salary as PF which is 12% of Basic pay. Propose a computerized solution for the above said problem.

Input : Basic Pay

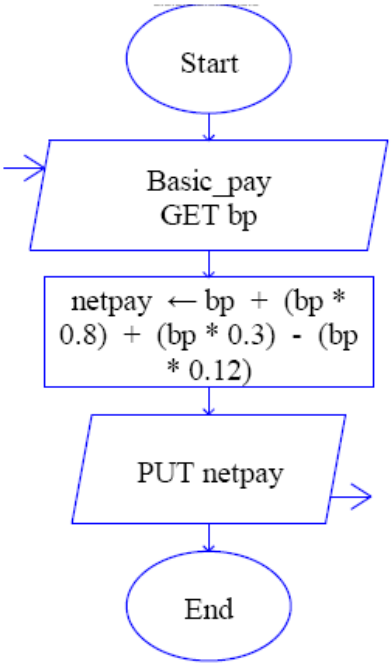
Process : Calculate Salary

(Basic Pay + (Basic Pay * 0.8) + (Basic Pay * 0.3 - (Basic Pay * 0.12)

----------allowances ---------- --- deductions----

Output : Salary

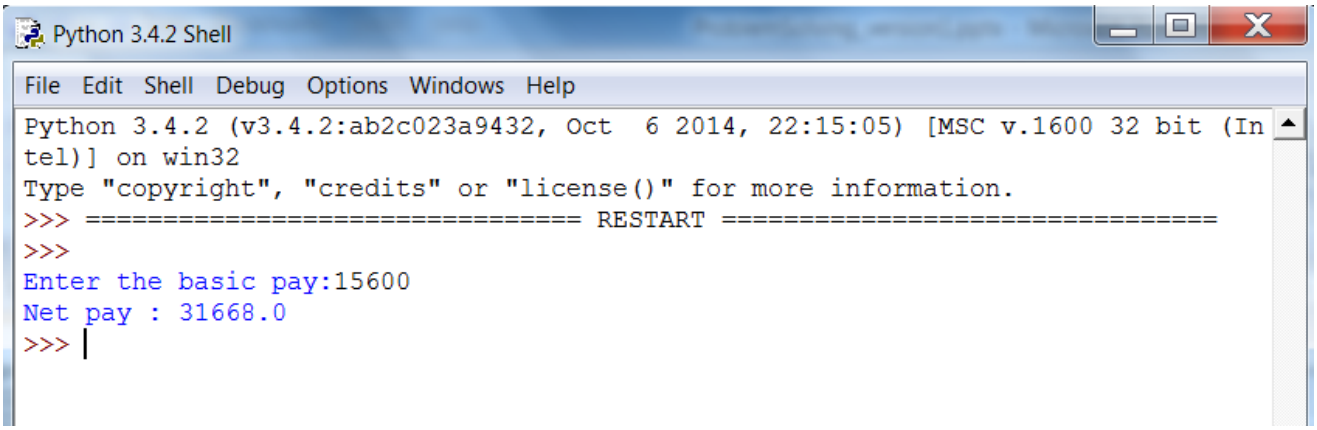
Flow chart



Python code

```
#Enter the basic pay
bp=float (input('Enter the basic pay:'))
# net pay calucluation
netpay =bp + (bp*0.8) + (bp*0.3) - (bp*0.12)
# display net salary
print ('Net pay :',netpay)
```

Output



Python features….. lambda operator

The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name.

```
>>> ftoc =lambda f: (f-32)*5.0/9
>>> ftoc(104)
```

Different patterns in Algorithm

Sequential

- **Sequential structure executes the program in the order in which they appear in the program**

Selectional (conditional-branching)

- **Selection structure control the flow of statement execution based on some condition**

Iterational (Loops)

- **Iterational structures are used when part of the program is to be executed several times**

Sequential Pattern

Example1: Find the average runs scored by a batsman in 4 matches

Algorithm:

Step 1: Start

Step 2: Input 4 scores say runs1,runs2,runs3 and runs4

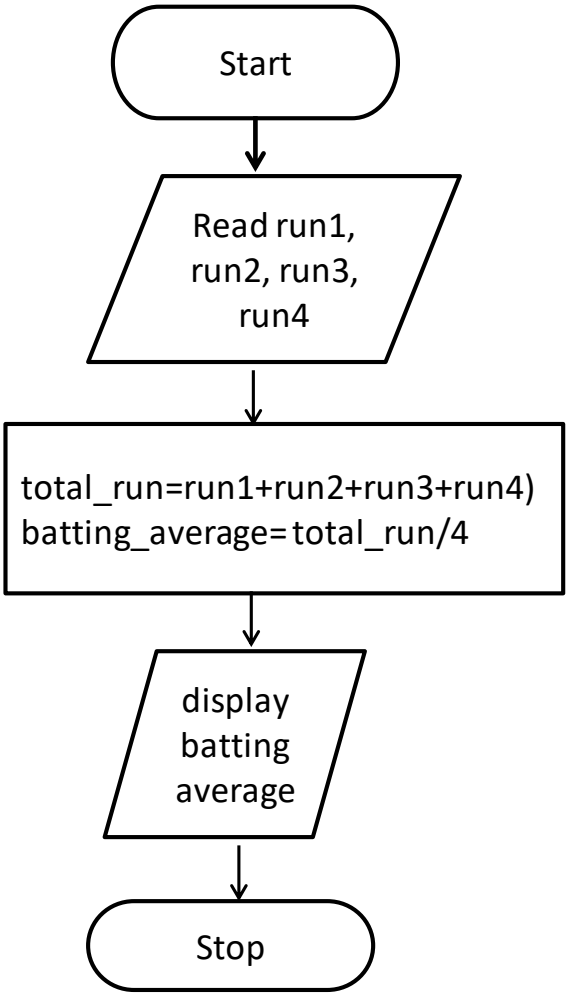
Step 3: Accumulate runs1,runs2,run3,and runs4 and store it in the variable called total_runs

Step 4: Divide total_runs by 4 and find the average

Step 5: Display the average

Step 6: Stop

Flowchart



Pseudo code:

Begin

read run1,run2,run3 and run4

compute total_run= run1+run2+run3+run4

compute batting_average= total_run/4

display batting_average

end

Batting Average

```
print("Enter four scores")
run1 = int(input())
run2 = int(input())
run3 = int(input())
run4 = int(input())
total_run=(run1+run2+run3+run4)
batting_average= total_run/4
print('batting_average is' ,batting_average)
```

Area of a circle

Step 1 : Start

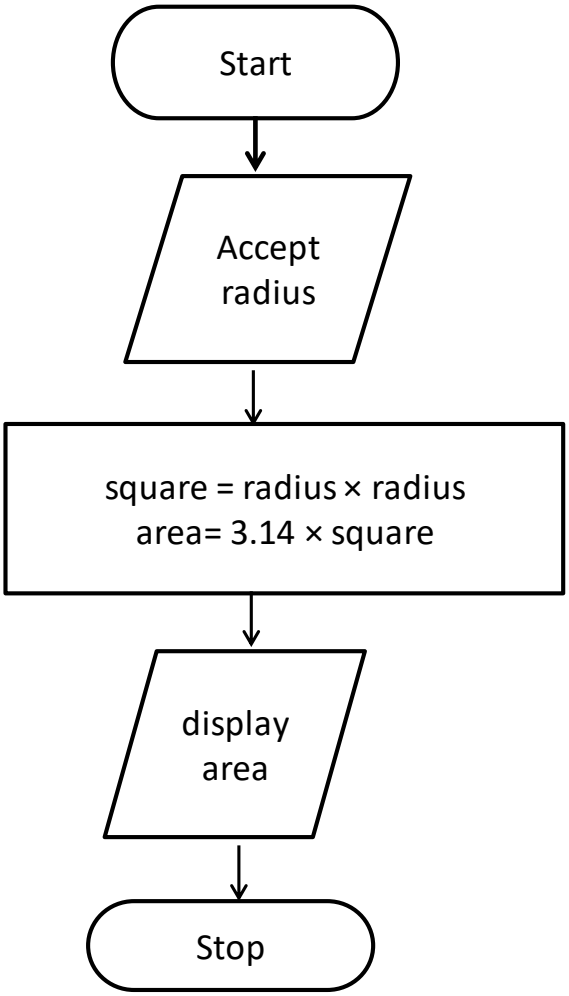
Step 2: Get the input for RADIUS

Step 3 : Find the square of RADIUS and store it in SQUARE

Step 4 : Multiply SQUARE with 3.14 and store the result in AREA

Step 5: Stop

Flowchart



Pseudo code:

begin

accept radius

compute square = radius * radius

compute area = pi * square

display area

end

Area of a circle

```
import math
print("Enter radius")
radius=float(input())
area = math.pi*radius*radius
print("area of circle is ", area)
```

Exercise

An university is setting up a new lab at their premises. Design an algorithm and write Python code to determine the approximate cost to be spent for setting up the lab. Cost for setting the lab is sum of cost of computers, cost of furnitures and labour cost. Use the following formulae for solving the problem:

Cost of computer = cost of one computer * number of computers

Cost of furniture = Number of tables * cost of one table + number of chairs * cost of one chair

Labour cost = number of hours worked * wages per hour

Budget for Lab

Input	Processing	Output
cost of one computer, number of computers, number of tables, cost of one table, number of chairs, cost of one chair, number of hours worked, wages per hour	Budget = Cost of computers + cost of furniture + labour cost Cost of computer = cost of one computer * number of computers Cost of furniture = Number of tables * cost of one table + number of chairs * cost of one chair Labour cost = number of hours worked * wages per hour	Budget for Lab

Python Program

```
print("Enter cost of one computer")
cost_Computer = float(input())
print("Enter num of computers")
num_Computer = int(input())
print("Enter cost of one table")
cost_Table = float(input())
print("Enter num of tables")
num_Tables = int(input())
print("Enter cost of one chair")
cost_Chair = float(input())
print("Enter num of chairs")
num_Chairs = int(input())
print("Enter wage for one hour")
wages_Per_Hr = float(input())
print("Enter num of hours")
num_Hrs = int(input())
```

Python Program

```
cost_Of_Computers = cost_Computer* num_Computer
cost_Of_Furnitures = num_Tables * cost_Table +\
                     cost_Chair*num_Chairs
wages = wages_Per_Hr * num_Hrs
budget = cost_Of_Computers + cost_Of_Furnitures + wages
#format for two decimal places
print ("Budget for Lab ",format(budget,'.2f'))
```

Browsing Problem

Given the number of hours and minutes browsed, write a program to calculate bill for Internet Browsing in a browsing center. The conditions are given below.

(a) 1 Hour Rs.50

(b) 1 minute Re. 1

(c) Rs. 200 for five hours

Boundary condition: User can only browse for a maximum of 7 hours

Check boundary conditions

Browsing Program

Input	Processing	Output
Number of hours and minutes browsed	Check number of hours browsed, if it is greater than 5 then add Rs 200 to amount for five hours and subtract 5 from hours Add Rs for each hour and Re 1 for each minute Basic process involved: Multiplication and addition	Amount to be paid

Pseudocode

```
READ hours and minutes
SET amount = 0
IF hours >=5 then
    CALCULATE amount as amount + 200
    COMPUTE hours as hours – 5
END IF
COMPUTE amount as amount + hours * 50
COMPUTE amount  as amount + minutes * 1
PRINT amount
```

Test Cases

Input

Hours = 6

Minutes = 21

Output

Amount = 271

Processing Involved

Amount = 200 for first five hours

50 for sixth hour

21 for each minute

Test Cases

Input

Hours = 8

Minutes = 21

Output

Invalid input

Processing Involved

Boundary conditions are violated

Already Know

- To read values from user
- Write arithmetic expressions in Python
- Print values in a formatted way

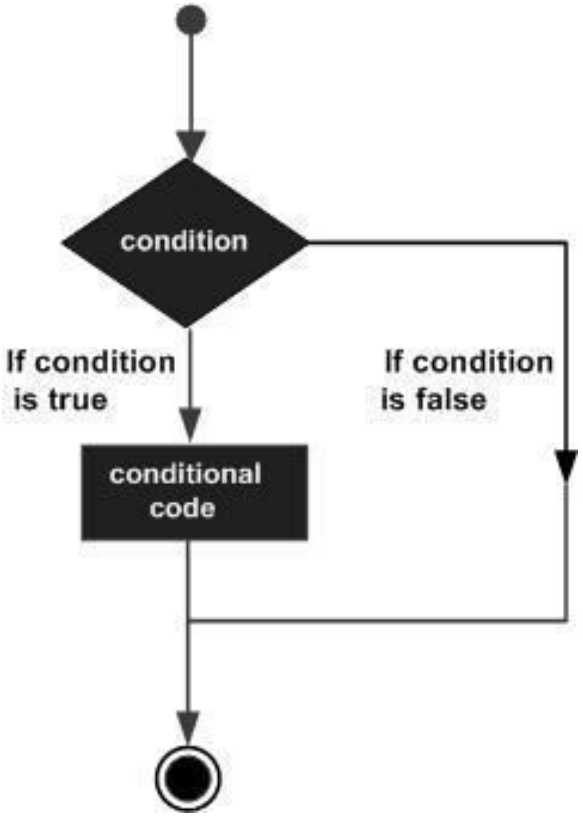
Yet to learn

- Check a condition

Selection pattern

- A **selection control statement** is a control statement providing selective execution of instructions.

Control flow of decision making



If Statement

- An **if statement** is a selection control statement based on the value of a given Boolean expression.
  The if statement in Python is

If statement	Example use

If condition:
  statements
else:
  statements
	If grade >=70:
	  print('pass')
	else:
	Print('fail')

If Statement

If statement	Example use

If condition:
  statements
else:
  statements
	If grade >=70:
	  print('pass')
	else:
	Print('fail')

Indentation in Python

- One fairly unique aspect of Python is that the amount of indentation of each program line is significant.
- In Python indentation is used to associate and group statements

Valid indentation

(a)
```
if condition:
    statement
    statement
else:
    statement
    statement
```

(b)
```
if condition:
    statement
    statement
else:
    statement
    statement
```

Invalid indentation

(c)
```
if condition:
   statement
   statement
else:
    statement
    statement
```

(d)
```
if condition:
   statement
   statement
else:
    statement
     statement
```

Nested if Statements

- There are often times when selection among more than two sets of statements (suites) is needed.
- For such situations, if statements can be nested, resulting in **multi-way selection**.

Nested if statements

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

            etc.
```

Example use

```
if grade >= 90:
    print('Grade of A')
else:
    if grade >= 80:
        print('Grade of B')
    else:
        if grade >= 70:
            print('Grade of C')
        else:
            if grade >= 60:
                print('Grade of D')
            else:
                print('Grade of F')
```

Else if Ladder

```
if grade >= 90:
    print('Grade of A')
elif grade >= 80:
    print('Grade of B')
elif grade >= 70:
    print('Grade of C')
elif grade >= 60:
    print('Grade of D')
else:
    print('Grade of F')
```

Multiple Conditions

- Multiple conditions can be check in a ‘if’ statement using logical operators ‘and’ and ‘or’.
- Python code to print ‘excellent’ if mark1 and mark2 is greater than or equal to 90, print ‘good’ if mark1 or mark2 is greater than or equal to 90, print ‘need to improve’ if both mark1 and mark2 are lesser than 90

```
if mark1>=90 and mark2 >= 90:
    print('excellent')
if mark1>=90 or mark2 >= 90:
    print('good')
else:
    print('needs to improve')
```

Browsing Program

```
print("enter num of hours")
hour = int(input())
print("enter num of minutes")
min = int(input())
if(hour>7):
    print("Invalid input")
elif hour>=5:
    amount = 200
    hour = hour - 5
    amount = amount+hour*50+min
    print(amount)
```

Eligibility for Scholarship

Government of India has decided to give scholarship for students who are first graduates in family and have scored average > 98 in math, physics and chemistry. Design an algorithm and write a Python program to check if a student is eligible for scholarship

Boundary Conditions: All marks should be >0

Scholarship Program

Input	Processing	Output
Read first graduate, physcis, chemistry and maths marks	Compute total = phy mark + che mark + math mark Average = total/3 Check if the student is first graduate and average >=98	Print either candidate qualified for Scholarship or candidate not qualified for Scholarship

Algorithm

Step 1 : Start

Step 2: Read first graduate, physcis,chemistry and maths marks

Step 3: If anyone of the mark is less than 0 then print 'invalid input' and terminate execution

Step 3 : Accumulate all the marks and store it in Total

Step 4 : Divide Total by 3 and store it in Average

Step 5 : If student is first graduate Average score is greater than or equal to 98 then print candidate qualified for Scholarship

Else

Print candidate not qualified for scholarship

Stop 6: Stop

Test Cases

Input

First graduate = 1 Phy mark = 98, Che mark = 99, math mark = 98

Output

candidate qualified for Scholarship

Processing Involved

Total = 295

Average = 98.33

Student is first graduate and average > 98

Test Cases

Input

First graduate = 0 Phy mark = 98, Che mark = 99, math mark = 98

Output

candidate is not qualified for Scholarship

Processing Involved

Total = 295

Average = 98.33

Student is not first graduate but average > 98

Test Cases

Input

First graduate = 1 Phy mark = 98, Che mark = 99, math mark = 90

Output

candidate is not qualified for Scholarship

Processing Involved

Total = 287

Average = 95.67

Student is first graduate but average < 98

```
print('Is first graduate(1 for yes and 0 for no')
first = int(input())
print('Enter Physics Marks')
phy_mark = float(input())
print('Enter Chemistry Marks')
che_mark=float(input())
print('Enter Math Marks')
mat_mark=float(input())
total_mark= phy_mark+che_mark+mat_mark
```

```
if(phy_mark <0 or che_mark <0 or mat_mark<0):
    print('Invalid input')
else:
    average = total_mark/3
    if first==1 and average >= 98 :
        print('candidate qualified for  Scholarship')
    else:
        print('candidate not qualified for  Scholarship')
```

Algorithm for Largest of Three numbers

Step1: Start

Step2: Read value of a, b and c

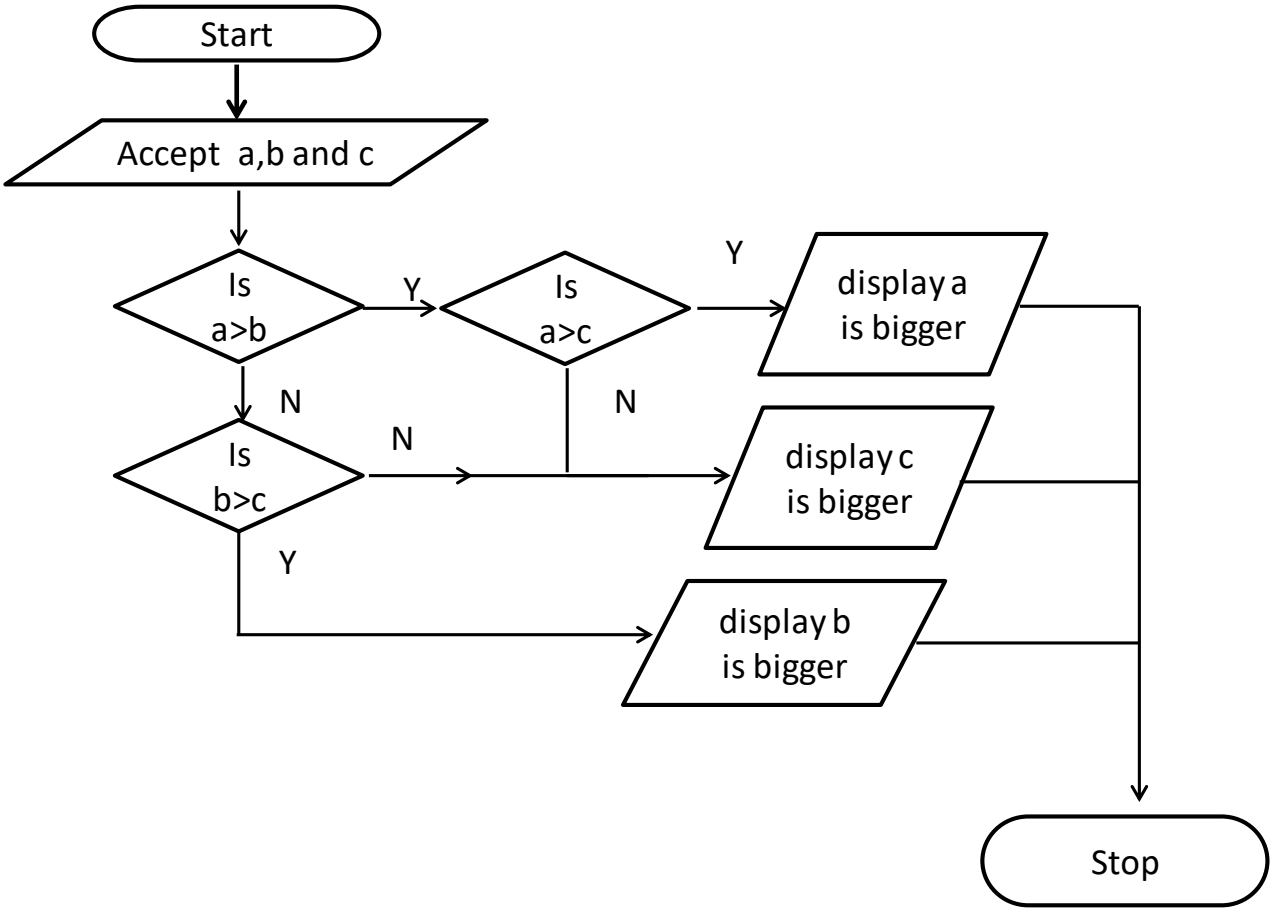
Step3: If a is greater than b then

compare a with c and if a is bigger then say a is biggest else say c is biggest

else Compare b with c , if b is greater than c say b is biggest else c is biggest

Step 5: Stop

Flowchart



Test Cases

Input

a = 12, b = 13, c = 14

Output

c is greatest

Processing Involved

B is greater than a but c is greater than b

Test Cases

Input

a = 13, b = 12, c = 14

Output

c is greatest

Processing Involved

a is greater than b but c is greater than a

Test Cases

Input

a = 13, b = 2, c = 4

Output

a is greatest

Processing Involved

a is greater than b and a is greater than c

Test Cases

Input

a = 3, b = 12, c = 4

Output

b is greatest

Processing Involved

b is greater than a and b is greater than c

```
a = int(input())
b = int(input())
c = int(input())
if a>b:
    if a>c:
        print ('a is greatest')
    else:
        print ('c is greatest')
else:
    if b>c:
        print ('b is greatest')
    else:
        print ('c is greatest')
```

The if/else Ternary Expression

Consider the following statement, which sets A to either Y or Z, based on the truth value of X:

if X:

A = Y

else:

A = Z

new expression format that allows us to say the same thing in one expression:

- A = Y if X else Z

```
>>> A = 't' if 'spam' else 'f'
>>> A
't'
>>> A = 't' if '' else 'f'
>>> A
'f'
```

Exercise Problem

1. Write a python code to check whether a given number of odd or even?
2. Write a python code to check whether a given year is leap year or not?
3. Write a python code in finding the roots of a quadratic equation?
4. Write a python program to segregate student based on their CGPA. The details are as follows:

<=9 CGPA <=10 - outstanding
<=8 CGPA <9 - excellent
<=7 CGPA <8 - good
<=6 CGPA <7 - average
<=5 CGPA <6 - better
CGPA<5 - poor

Class Average

- Given marks secured in CSE1001 by the students in a class, design an algorithm and write a Python code to determine the class average. Print only two decimal digits in average

Class Average

Input	Processing	Output
Number of students in class, mark scored by each student	Determine total of marks secured by students Find average of marks	Class average of marks

Average marks scored by ‘N’ number of Students

Step 1: Start
Step 2 : Read Number Of Students
Step 3 : Initialize counter as 0
Step 4 : Input mark
Step 5 : Add the mark with total
Step 6 : Increment the counter by 1
Step 7: repeat Step 4 to Step 6 until counter less than number of students
Step 7: Divide the total by number of students and store it in average
Step 8: Display the average
Step 9: Stop

Test Cases

Input

5

90 85 70 50 60

Output

71.00

Processing Involved

Already Know

- To read values from user
- To check if a condition is satisfied
- Print characters

Yet to learn

- Repeatedly execute a set of statements

Need of iterative control

Repeated execution of set of statements

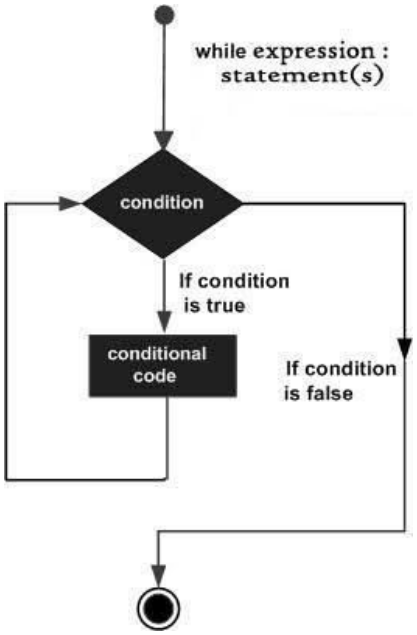
- An **iterative control statement** is a control statement providing repeated execution of a set of instructions
- Because of their repeated execution, iterative control structures are commonly referred to as “loops.”

While statement

- Repeatedly executes a set of statements based on a provided Boolean expression (condition).
- All iterative control needed in a program can be achieved by use of the while statement.

Syntax of While in Python

```
while test:                    # Loop test
    statements                 # Loop body
else:                          # Optional else
    statements
# Run if didn't exit loop with break
```



Example use

Sum of first 'n' numbers

```
sum =0
current =1
n=3
while  current  <= n:
    sum=sum + current
    current = current + 1
```

Iteration	sum	current	current <= 3	sum = sum + current	current = current + 1
1	0	1	True	sum = 0 + 1 (1)	current = 1 + 1 (2)
2	1	2	True	sum = 1 + 2 (3)	current = 2 + 1 (3)
3	3	3	True	sum = 3 + 3 (6)	current = 3 + 1 (4)
4	6	4	False	loop termination	

Print values from 0 to 9 in a line

```
a=0; b=10
while a < b:            # One way to code counter loops
        print(a, end=' ')
        a += 1                  # Or, a = a + 1
```

Output:

0 1 2 3 4 5 6 7 8 9

Include end=' ' in print statement to suppress default move to new line

Break, continue, pass, and the Loop else

- break Jumps out of the closest enclosing loop
- continue Jumps to the top of the closest enclosing loop
- pass Does nothing at all: it’s an empty statement placeholder
- Loop else block Runs
- if and only if the loop is exited normally (i.e., without hitting a break)

Break statement

- while True:

```
        name = input('Enter name:')
        if name == 'stop': break
       age  = input('Enter age: ')
        print('Hello', name, '=>', int(age) ** 2)
```

Output:

```
 Enter name:bob
Enter age: 40
Hello bob => 1600
```

Pass statement

- Infinite loop
- while True: pass

  # Type Ctrl-C to stop me!

Print all even numbers less than 10 and greater than or equal to 0

```
x = 10
while x:
        x = x-1                         # Or, x -= 1
        if x % 2 != 0: continue         # Odd? -- skip print
        print(x, end=' ')
```

Class Average

```
count =0
total = 0
n=int(input('enter how many mark you want to read: '))
while count < n:
    mark=int(input('enter mark :'))
    if mark<0:
            print ("mark should be greater than 0, terminates.
            break
    total = total + mark
    count = count + 1
else:
    average=total/n
    print("average mark is" , format(average,"0.2f"))
```

Pattern Generation

- Your teacher has given you the task to draw the structure of a staircase. Being an expert programmer, you decided to make a program for the same. You are given the height of the staircase. Given the height of the staircase, write a program to print a staircase as shown in the example. For example, Staircase of height 6:

  #
  ##
  ###
  ####
  #####
  ######

Boundary Conditions: height >0

Pattern Generation

Input	Processing	Output
Staircase height	Create steps one by one To create a step print character equal to length of step	Pattern

Pseudocode

```
READ staircase_height
if staircase_height > 0
x = 1
Repeat
y = 1
Repeat
    print #
    y = y + 1
Until y <= x
x = x + 1
Until x <= staircase_height
End if
Else
Print “Invalid input”
```

Test Cases

Input

3

Output

```
#
# #
# # #
```

Processing Involved

Print step by step

Test Cases

Input

-1

Output

Invalid input

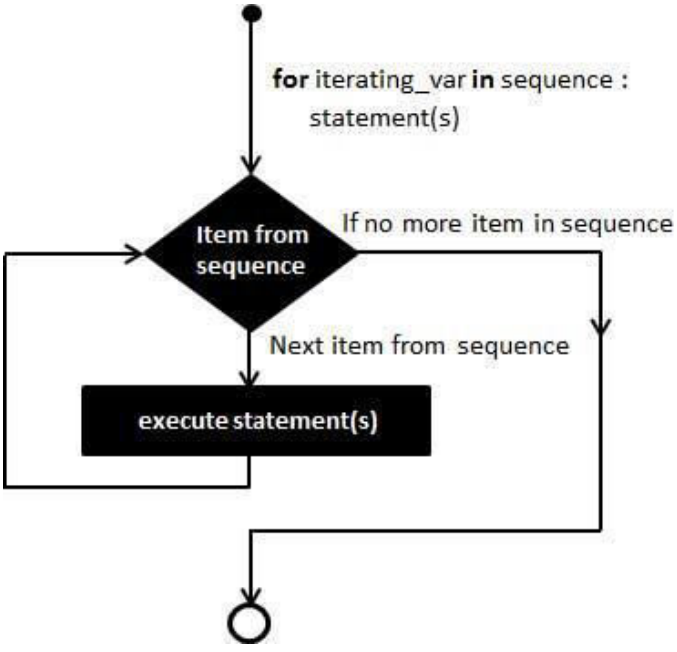
Processing Involved

Boundary condition check fails

For iteration

- In while loop, we cannot predict how many times the loop will repeat
- The number of iterations depends on the input or until the conditional expression remains true

Control flow of for statement



Syntax of for Statement

```
for target in object:
    # Assign object items to target
    statements
    if test: break          # Exit loop now, skip else
    if test: continue       # Go to top of loop now
else:    statements         # If we didn't hit a 'break'
```

For and Strings

for iterating_var in sequence or range:

    statement(s)

Example:

```
for letter in 'Python':
    print ('Current Letter :', letter)
```

For and Strings

When the above code is executed:

```
Current Letter : P
 Current Letter : y
 Current Letter : t
 Current Letter : h
 Current Letter : o
 Current Letter : n
```

For and Range

```
for n in range(1, 6):
    print(n)
```

When the above code is executed:

```
1
2
3
4
5
```

range function call

Syntax - range(begin,end,step)

where

Begin - first value in the range; if omitted, then default value is 0

end - one past the last value in the range; end value may not be omitted

Step - amount to increment or decrement; if this parameter is omitted, it defaults to 1 and counts up by ones

begin, end, and step must all be integer values;

floating-point values and other types are not allowed

Example for Range

range(10) → 0,1,2,3,4,5,6,7,8,9

range(1, 10) → 1,2,3,4,5,6,7,8,9

range(1, 10, 2) → 1,3,5,7,9

range(10, 0, -1) → 10,9,8,7,6,5,4,3,2,1

range(10, 0, -2) → 10,8,6,4,2

range(2, 11, 2) → 2,4,6,8,10

range(-5, 5) → −5,−4,−3,−2,−1,0,1,2,3,4

range(1, 2) → 1

range(1, 1) → (empty)

range(1, -1) → (empty)

range(1, -1, -1) → 1,0

range(0) → (empty)

Print Even Numbers Using Range

```
>>> for i in range(2,10,2):
        print(i)
```

Output:

2

4

6

8

```
print("Enter number of steps")
n = int(input())
for i in range(0,n):
    for j in range(0,i+1):
        print('#',end = ' ')
    print()
```

Exercise Problem

1. Write a program that read a group ‘g’ of five numbers and another number ‘n’ and print a number in ‘g’ if it is a factor for a given number n?
2. Write a program to find the factorial of a number n?
3. Write a menu driven program which get user choice to perform add/sub/mul/div with the obtained two input?
4. Write a program to display few odd multiples of a odd number n ?

Exercise Problem

5. The Head Librarian at a library wants you to make a program that calculates the fine for returning the book after the return date. You are given the actual and the expected return dates. Calculate the fine as follows:

   a. If the book is returned on or before the expected return date, no fine will be charged, in other words fine is 0.

   b. If the book is returned in the same month as the expected return date, Fine = 15 Rupees × Number of late days

   c. If the book is not returned in the same month but in the same year as the expected return date, Fine = 500 Rupees × Number of late months

   d. If the book is not returned in the same year, the fine is fixed at 10000 Rupees.

Strings in Python

Check Validity of a PAN

In any of the country's official documents, the PAN number is listed as follows

<alphabet>< alphabet> < alphabet > < alphabet >

< alphabet > <digit><digit><digit><digit>< alphabet >

Your task is to figure out if the PAN number is valid or not. A valid PAN number will have all its letters in uppercase and digits in the same order as listed above.

PAC For Chocolate Problem

Input	Processing	Output
PAN number	Take each character and check if alphabets and digits are appropriately placed	Print Valid or Invalid

Pseudocode

```
READ PAN
If length of PAN is not ten then print “Invalid” and exit
FOR x=0 to5
        if PAN[x] is not a character THEN
                PRINT ‘invalid’
                BREAK;
        END IF
END FOR
FOR x=5 to 9
        if PAN[x] is not a digit THEN
                PRINT ‘invalid’
                BREAK;
        END IF
END FOR
IF PAN[9] is not a character THEN
        PRINT ‘invalid’
                END IF
PRINT ‘valid’
```

Test Case 1

abcde1234r

Valid

Test Case 2

abcde12345

Invalid

Test Case 3

abcd01234r

Invalid

Strings

- Immutable sequence of characters
- A string literal uses quotes
- 'Hello' or "Hello“ or ‘”Hello”’
- For strings, + means “concatenate”
- When a string contains numbers, it is still a string

String Operations

Operation	Interpretation
`S = ''`	Empty string
`S = "spam's"`	Double quotes, same as single
`S = 's\np\ta\x00m'`	Escape sequences
`S = """...multiline..."""`	Triple-quoted block strings
`S = r'\temp\spam'`	Raw strings (no escapes)
`B = b'sp\xc4m'`	Byte strings in 2.6, 2.7, and 3.X (Chapter 4, Chapter 37)
`U = u'sp\u00c4m'`	Unicode strings in 2.X and 3.3+ (Chapter 4, Chapter 37)
`S1 + S2`	Concatenate, repeat
`S * 3`	
`S[i]`	Index, slice, length
`S[i:j]`	

String Operations

`len(S)`	
`"a %s parrot" % kind`	String formatting expression
`"a {0} parrot".format(kind)`	String formatting method in 2.6, 2.7, and 3.X
`S.find('pa')`	String methods (see ahead for all 43): search,
`S.rstrip()`	remove whitespace,
`S.replace('pa', 'xx')`	replacement,
`S.split(',')`	split on delimiter,

String Operations

Operation	Interpretation
`S.isdigit()`	content test,
`S.lower()`	case conversion,
`S.endswith('spam')`	end test,
`'spam'.join(strlist)`	delimiter join,

Example Strings

- Single quotes: 'spa"m'
- Double quotes: "spa'm"
- Triple quotes: '''... spam ...''', """... spam ..."""
- Escape sequences: "s\tp\na\0m“
- Raw strings: r"C:\new\test.spm"

Escape Sequences

- Represent Special Characters

```
>>> s = 'a\nb\tc'
>>> print(s)
a
b	c
>>> len(s)
5
```

Escape Sequences

Table 7-2. String backslash characters

Escape	Meaning
newline	Ignored (continuation line)
\\	Backslash (stores one \)
\'	Single quote (stores ')
\"	Double quote (stores ")
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline (linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\x*hh*	Character with hex value *hh* (exactly 2 digits)
ooo	Character with octal value *ooo* (up to 3 digits)
\0	Null: binary 0 character (doesn't end string)

Length of a String

```
>>> s = ‘bobby’
>>> len(s)
5
>>> print(s)
a b c
```

Backslash in Strings

- if Python does not recognize the character after a \ as being a valid escape code, it simply keeps the backslash in the resulting string:

```
>>> x = "C:\py\code"
# Keeps \ literally (and displays it as \\)
>>> x
'C:\\py\\code'
>>> len(x)
10
```

Backslash in Strings

```
>>> x = "C:\py\code"
>>> x
'C:\\py\\code'
>>> len(x)
10
```

Check this

```
s = "C:\new\text.dat“
>>>s
print(s)
s1 = r"C:\new\text.dat“
>>>s1
print(s1)
s2 = "C:\\new\\text.dat“
print(s2)
>>>s2
```

Opening a File

- myfile = open('C:\new\text.dat', 'w') - Error
- myfile = open(r'C:\new\text.dat', 'w')
- Alternatively two backslashes may be used
- myfile = open('C:\\new\\text.dat', 'w')

```
>>> path = r'C:\new\text.dat'
>>> print(path)                    # User-friendly format
C:\new\text.dat
>>> len(path)
15
```

Basic Operations

```
>>> 'Ni!' * 4
'Ni!Ni!Ni!Ni!‘
>>> print('-' * 80)                # 80 dashes, the easy way
>>> myjob = "hacker"
>>> for c in myjob:
        print(c, end=' ')
h a c k e r
```

Basic Operations

```
>>> for c in ‘myjob’:
        print(c, end=' ')
h a c k e r
```

Using ‘in’ Operator in Strings

```
>>> "k" in myjob                    # Found
True
>>> "z" in myjob                    # Not found
False
>>> 'spam' in 'abcspamdef'
# Substring search, no position returned
True
```

Counting

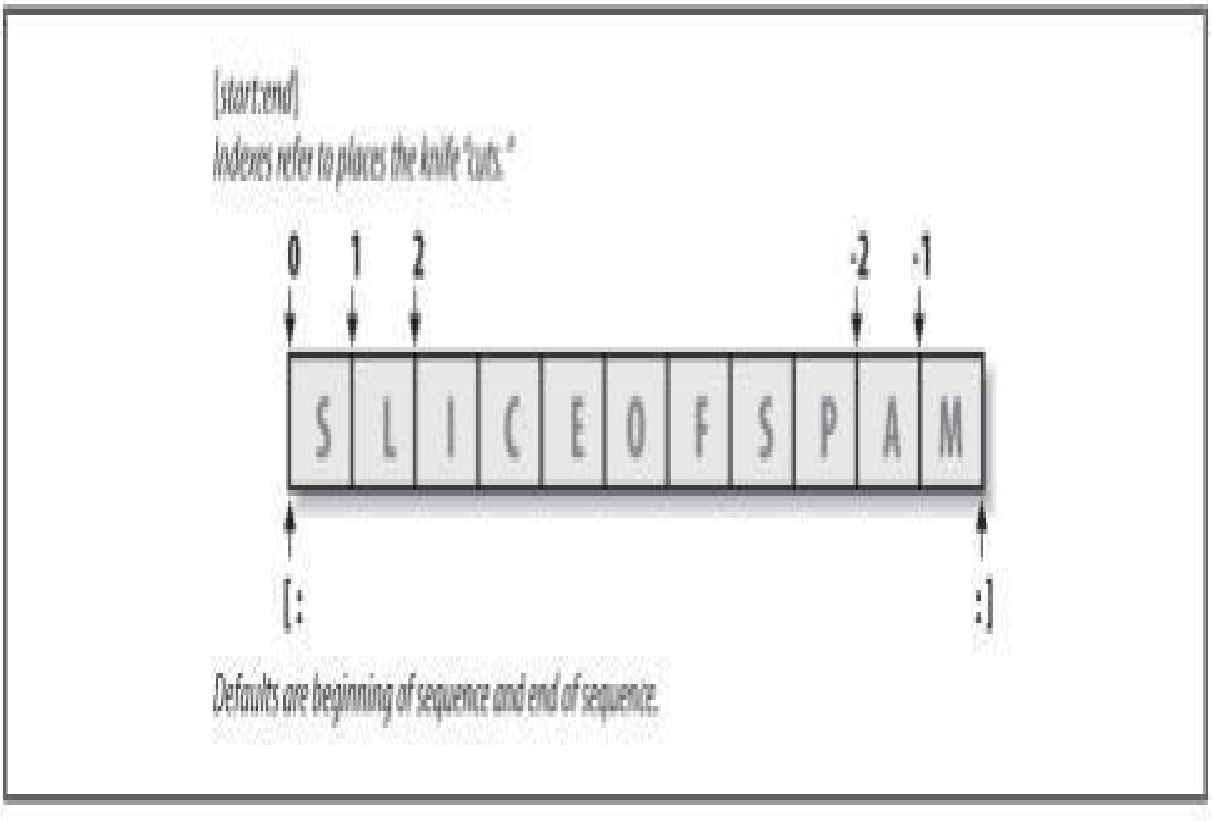
- Count the number of ‘a’

Counting

- Count the number of ‘a’
- Example
- *word = 'Btechallbranches'*
- *count = 0*
- *for letter in word :*
- *if letter == 'a' :*
- *count = count + 1*
- *print(count)*

Indexing and Slicing

- >>> S = 'spam'
- Last character in the string has index -1 and the one before it has index -2 and so on



Indexing

- Take one letter from a word at a time
- Use square bracket and give the index of the letter to be extracted
- Indexing can be done either from front or from end
- >>> S[0], S[−2]
- ('s', 'a')

Slicing

- Take a part of a word
- Square bracket with two arguments with a colon
- First value indicates the starting position of the slice and second value indicates the stop position of the slice
- Character at the stop position is not included in the slice
- >>> S[1:3]
- 'pa’

Slicing

- If the second number is beyond the end of the string, it stops at the end
- If we leave off the first or last number of the slice, it is assumed to be beginning or end of the string respectively
- s = ‘spam’
- >>>s[:3]
- ‘spa’
- >>>s[1:]
- ‘pam’

Properties of Slicing

- S[1:3] - fetches items at offsets 1 up to but not including 3.
- S[1:] - fetches items at offset 1 through the end
- S[:3] - fetches items at offset 0 up to but not including 3
- S[:−1] - fetches items at offset 0 up to but not including last item
- S[:] - fetches items at offsets 0 through the end—making a top-level copy of S

Extended slicing

- X[I:J:K] - means “extract all the items in X, from offset I through J−1, by K.”
- Third limit, K, defaults to +1
- If you specify an explicit value it is used to skip items
- Extraction is reversed when negative value is given for K-1
- Each time K-1 items are skipped

```
ation.
>>> str1 = '0123456789'
>>> str1[:]
'0123456789'
>>> str1[::2]
'02468'
>>> str1[::3]
'0369'
>>> str1[1::2]
'13579'
>>> str1[1:6:2]
'135'
>>>
```

Extended slicing Examples

```
>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]              # Skipping items
'bdfhj'
>>> S[::2]
'acegikmo‘
>>> S = 'hello'
>>> S[::−1]                # Reversing items
'olleh'
```

Extended slicing Examples

```
>>> S = 'hello'
>>> S[::−1]                # Reversing items
'olleh'
```

String Conversion Tools

- >>> "42" + 1
- TypeError: Can't convert 'int' object to str implicitly
- >>> int("42"), str(42) # Convert from/to string
- (42, '42')
- int("42") + 1
- 43
- >>> "42" + str(1)
- '421'

Character code Conversions

- ord () - Convert a single character to its underlying integer code (e.g., its ASCII byte value)— this value is used to represent the corresponding character in memory.
- >>> ord('s')
- 115

Character code Conversions

- chr () – Does inverse of ord
- >>> chr(115)
- 's'

Character code Conversions - Example

- >>> S = '5'
- >>> S = chr(ord(S) + 1)
- >>> S
- '6‘
- >>> S = chr(ord(S) + 1)
- >>> S
- '7'

Character code Conversions - Example

- >>> ord('5') - ord('0')
- 5
- >>> int('1101', 2) # Convert binary to integer
- 13
- >>> bin(13) # Convert integer to binary
- '0b1101'

Concatenation

- >>>S1 = ‘Welcome’
- >>>S2 = ‘Python’
- >>>S3 = S1 + S2
- >>>S3
- 'WelcomePython'

Changing Strings

- String - “immutable sequence”
- Immutable - you cannot change a string in place
- >>> S = 'spam'
- >>> S[0] = 'x' # Raises an error!
- TypeError: 'str' object does not support item assignment

Changing Strings

- >>> S = S + 'SPAM!'
- # To change a string, make a new one
- >>> S
- 'spamSPAM!'
- >>> S = S[:4] + 'Burger' + S[−1]
- >>> S
- 'spamBurger!'

Replace

```
>>> S = 'splot'
>>> S = S.replace('pl', 'pamal')
>>> S
'spamalot'
```

Formatting Strings

```
>>> 'That is %d %s bird!' % (1, 'dead')
That is 1 dead bird!
>>> 'That is {0} {1} bird!'.format(1, 'dead')
'That is 1 dead bird!'
```

String Library

- Python has a number of string functions which are in the string library
- These functions do not modify the original string, instead they return a new string that has been altered

String Library

- >>> greet = 'Hello Arun'
- >>> zap = greet.lower()
- >>> print (zap)
- hello arun
- >>> print (‘Hi There'.lower())
- hi there

Searching a String

- find() - function to search for a string within another
- find() - finds the first occurrence of the substring
- If the substring is not found, find() returns -1
- >>> name = 'pradeepkumar‘
- >>> pos = name.find('de')
- >>> print (pos)
- 3

Searching a String

- >>> aa = “fruit”.find('z')
- >>> print (aa)
- -1
- >>> name = 'pradeepkumar‘
- >>> pos = name.find('de‘,5,8)
- >>>pos
- -1

Other Common String Methods in Action

- >>> line = "The knights who say Ni! “
- >>> line.rstrip()
- 'The knights who say Ni!'
- >>> line.upper()
- 'THE KNIGHTS WHO SAY NI!‘

```
>>> s.find('e',3,4)
-1
>>> s.find('t',3,6)
-1
>>> s='Hello    \t'
>>> s
'Hello    \t'
>>> s.rstrip()
'Hello'
>>>
```

Other Common String Methods in Action

- >>> line.isalpha()
- False
- >>> line.endswith('Ni!')
- True
- >>> line.startswith('The')
- True

Other Common String Methods in Action

- length and slicing operations can be used to mimic endswith:
- >>> line = 'The knights who say Ni!\n'
- >>> line.find('Ni') != -1
- True
- >>> 'Ni' in line
- True

Other Common String Methods in Action

- >>> sub = 'Ni!\n'
- >>> line.endswith(sub) # End test via method call or slice
- True

```
pan = input("enter pan")
invalid = False
if len(pan)!=10:
    invalid = True
else:
    for i in range(0,5):
        if not pan[i].isalpha():
            invalid = True
            break
    for i in range(5,9):
        if not pan[i].isdigit():
            invalid = True
            break
if not pan[9].isalpha():
            invalid = True
if invalid == True:
    print ("Invalid")
```

To check if all letters in a String are in Uppercase

isupper() function is used to check if all

letters in a string are in upper case

Examples

```
>>> 'a'.isupper()
False
>>> 'A'.isupper()
True
>>> 'AB'.isupper()
True
>>> 'ABc'.isupper()
False
>>>
```

To check if all letters in a String are in Lowercase

islower() function is used to check if all letters in a string are in lower case

Examples

```
>>> 'a'.islower()
True
>>> 'aB'.islower()
False
```

To check if a sentence is in Title case

istitle() function is used to check if all letters in a string are in title case

Examples

```
>>> 'Apple Is A Tree'.istitle()
True


>>> 'Apple Is A tree'.istitle()
False
```

Looping Through Strings

Using a while statement and an iteration variable, and the len function, we can construct a loop to look at each of the letters in a string individually.

```
fruit = 'banana'
index = 0
while index < len(fruit) :
   letter = fruit[index]
   print (index, letter)
   index = index + 1
```

For loop

A definite loop using a for statement is much more elegant

The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'
for letter in fruit :
   print (letter)
```

Looping and Counting

This is a simple loop that loops through each letter in a string and counts the number of times the loop encounters the 'a' character.

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print (count)
```

String Comparison

```
word=input()
if word == 'banana':
    print ( 'All right, bananas')
```

Strings

String Data Type

- A string is a sequence of characters
- A string literal uses quotes 'Hello' or “Hello”
- For strings, + means “concatenate”
- When a string contains numbers, it is still a string
- We can convert numbers in a string into a number using int()

```
>>> str1 = "Hello”
>>> str2 = 'there'
>>> bob = str1 + str2
>>> print bob
Hellothere
>>> str3 = '123'
>>> str3 = str3 + 1
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>TypeError: cannot
concatenate 'str' and 'int' objects
>>> x = int(str3) + 1
>>> print (x)
124
>>>
```

Reading and Converting

- Data entered from keyboard is by default String
- input numbers must be converted from strings

```
>>> msg = input('Enter:')
Enter:Hello
>>> print (msg)
Hello
>>> number = input('Enter:')
Enter:100
>>> x = number – 10
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>TypeError: unsupported
operand type(s) for -: 'str' and 'int'
>>> x = int(number) – 10
>>> print( x)
90
```

Looking Inside Strings

- We can get at any single character in a string using an index specified in square brackets
- The index value must be an integer and starts at zero
- The index value can be an expression that is computed

b	a	n	a	n	a
0	1	2	3	4	5

```
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> print (letter)
a
>>> n = 3
>>> w = fruit[n - 1]
>>> print (w)
n
```

A Character Too Far

- You will get a python error if you attempt to index beyond the end of a string.
- So be careful when constructing index values and slices

```
>>> zot = 'abc'
>>> print zot[5]
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>IndexError: string index
out of range
>>>
```

Strings Have Length

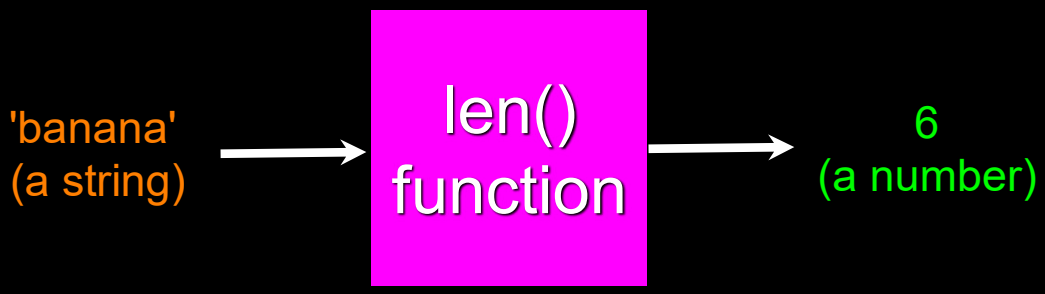
- There is a built-in function len that gives us the length of a string


```
>>> fruit = 'banana'
>>> print (len(fruit))
6
```

Len Function

```
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print (x)
6
```

A function is some stored code that we use. A function takes some input and produces an output.

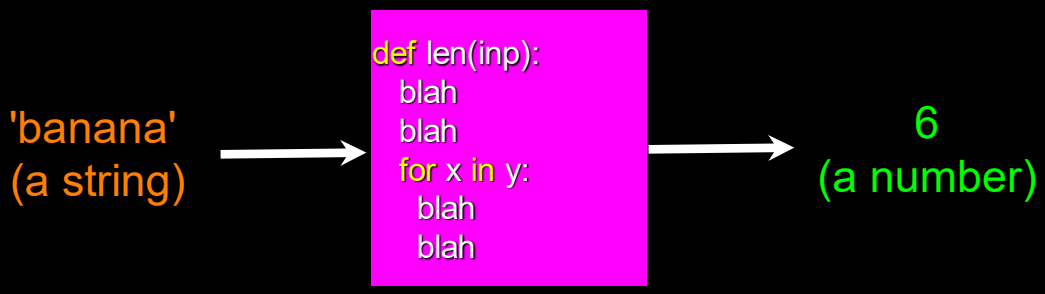


Guido wrote this code

Len Function

```
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print x
6
```

A function is some stored code that we use. A function takes some input and produces an output.



Looping Through Strings

- Using a while statement and an iteration variable, and the len function, we can construct a loop to look at each of the letters in a string individually

```
fruit = 'banana'
index = 0
while index < len(fruit) :
   letter = fruit[index]
    print (index, letter)
    index = index + 1
```

```
0 b
1 a
2 n
3 a
4 n
5 a
```

Looping Through Strings

- A definite loop using a for statement is much more elegant
- The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'
for letter in fruit :
   print (letter)
```

```
b
a
n
a
n
a
```

Looping Through Strings

- A definite loop using a for statement is much more elegant
- The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'
for letter in fruit :
   print letter


index = 0
while index < len(fruit) :
    letter = fruit[index]
    print (letter)
    index = index + 1
```

```
b
a
n
a
n
a
```

Looping and Counting

- This is a simple loop that loops through each letter in a string and counts the number of times the loop encounters the 'a' character.

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print (count)
```

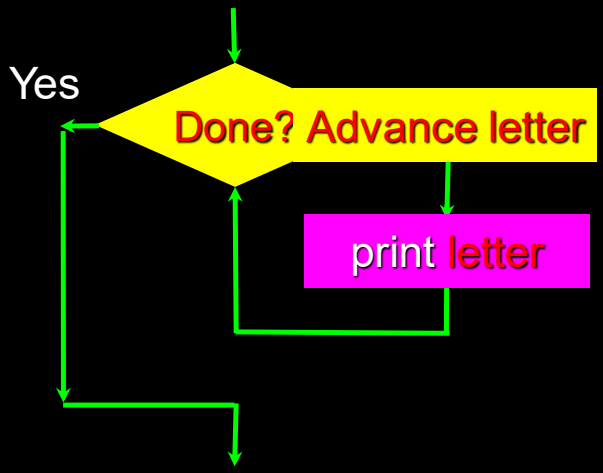
Looking deeper into in

- The iteration variable "iterates" though the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence

Six-character string

Iteration variable

```
for letter in 'banana' :
    print (letter)
```



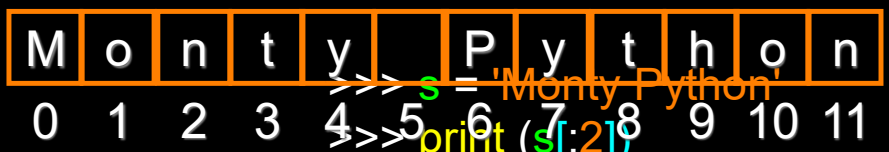

```
for letter in 'banana' :
    print (letter)
```

The iteration variable "iterates" though the string and the block (body) of code is executed once for each value in the sequence

Slicing Strings

- We can also look at any continuous section of a string using a colon operator
- The second number is one beyond the end of the slice - “up to but not including”
- If the second number is beyond the end of the string, it stops at the end

```
>>> s = 'Monty Python'
>>> print (s[0:4])
Mont
>>> print ( s[6:7])
P
>>> print ( s[6:20])
Python
```



- If we leave off the first number or the last number of the slice, it is assumed to be the beginning or end of the string respectively

```
>>> s = 'Monty Python'
>>>print (s[:2])
Mo
>>> print (s[8:])
Thon
>>> print (s[:])
Monty Python
>>> print( s[:-6])
Monty
>>> print (s[-4])
t
>>>print (s[-3:])
'hon'
```

String Concatenation

- When the + operator is applied to strings, it means "concatenation"

```
>>> a = 'Hello'
>>> b = a + 'There'
>>> print b
HelloThere
>>> c = a + '  ' + 'There'
>>> print c
Hello There
>>>
```

Using in as an Operator

- The in keyword can also be used to check to see if one string is "in" another string
- The in expression is a logical expression and returns True or False and can be used in an if statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit :
...     print 'Found it!'
...
Found it!
>>>
```

String Comparison

```
if word == 'banana':
    print ( 'All right, bananas.‘)

if word < 'banana':
    print ( 'Your word,' + word + ', comes before banana.’)
elif word > 'banana':
    print ( 'Your word,' + word + ', comes after banana.’)
else:
    print ('All right, bananas.‘)
```

String Library

- Python has a number of string functions which are in the string library
- These functions are already *built into* every string - we invoke them by appending the function to the string variable
- These functions do not modify the original string, instead they return a new string that has been altered

```
>>> greet = 'Hello Bob‘
>>> zap = greet.lower()
>>> print (zap)
hello bob
>>> print (greet)
Hello Bob
>>> print ('Hi There'.lower())
hi there
>>>
```

```
>>> stuff = 'Hello world'
>>> type(stuff)<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

http://docs.python.org/lib/string-methods.html

`str.`**`replace`**(*old*, *new*[, *count*])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.`**`rfind`**(*sub*[, *start*[, *end*]])

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within s[start,end]. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.`**`rindex`**(*sub*[, *start*[, *end*]])

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

`str.`**`rjust`**(*width*[, *fillchar*])

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if *width* is less than `len(s)`.

http://docs.python.org/lib/string-methods.html

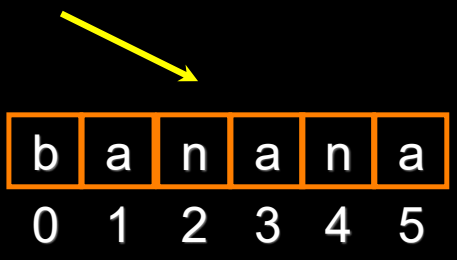
String Library

str.capitalize()
str.center(width[, fillchar])
str.endswith(suffix[, start[, end]])
str.find(sub[, start[, end]])
str.lstrip([chars])
str.replace(old, new[, count])
str.lower()
str.rstrip([chars])
str.strip([chars])
str.upper()

http://docs.python.org/lib/string-methods.html

Searching a String

- We use the find() function to search for a substring within another string
- find() finds the first occurance of the substring
- If the substring is not found, find() returns -1
- Remember that string position starts at zero




```
>>> fruit = 'banana'
>>> pos = fruit.find('na')
>>> print pos
2
>>> aa = fruit.find('z')
>>> print aa
-1
```

Making everything UPPER CASE

- You can make a copy of a string in lower case or upper case
- Often when we are searching for a string using find() - we first convert the string to lower case so we can search a string regardless of case

```
>>> greet = 'Hello Bob'
>>> nnn = greet.upper()
>>> print (nnn)
HELLO BOB
>>> www = greet.lower()
>>> print (www)
hello bob
>>>
```

Search and Replace

- The replace() function is like a "search and replace" operation in a word processor
- It replaces all occurrences of the search string with the replacement string

```
>>> greet = 'Hello Bob'
>>> nstr = greet.replace('Bob','Jane')
>>> print (nstr)
Hello Jane
>>> nstr = greet.replace('o','X')
>>> print (nstr)
HellX BXb
>>>
```

Stripping Whitespace

- Sometimes we want to take a string and remove whitespace at the beginning and/or end
- lstrip() and rstrip() to the left and right only
- strip() Removes both begin and ending whitespace

```
>>> greet = '   Hello Bob  '
>>> greet.lstrip()
'Hello Bob  '
>>> greet.rstrip()
'   Hello Bob'
>>> greet.strip()
'Hello Bob'
>>>
```

Prefixes

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Summary

- String type
- Read/Convert
- Indexing strings []
- Slicing strings [2:4]
- Looping through strings with for and while
- Concatenating strings with +
- String operations

Regular Expressions

Problem

Write a Python code to check if the given mobile number is valid or not. The conditions to be satisfied for a mobile number are:

a) Number of characters must be 10

b) All characters must be digits and must not begin with a ‘0’

Validity of Mobile Number

Input	Processing	Output
A string representing a mobile number	Take character by character and check if it valid	Print valid or invalid

Test Case 1

- abc8967891
- Invalid
- Alphabets are not allowed

Test Case 2

- 440446845
- Invalid
- Only 9 digits

Test Case 3

- 0440446845
- Invalid
- Should not begin with a zero

Test Case 4

- 8440446845
- Valid
- All conditions statisfied

Python code to check validity of mobile number (Long Code)

```
import sys
number = input()
if len(number)!=10:
    print ('invalid')
    sys.exit(0)
if number[0]==‘0’:
     print ('invalid')
    sys.exit(0)
for chr in number:
    if chr.isalpha():
        print ('invalid')
        break
else:
    print('Valid‘)
```

- Manipulating text or data is a big thing
- If I were running an e-mail archiving company, and you, as one of my customers, requested all of the e-mail that you sent and received last February, for example, it would be nice if I could set a computer program to collate and forward that information to you, rather than having a human being read through your e-mail and process your request manually.

- Another example request might be to look for a subject line like “ILOVEYOU,” indicating a virus-infected message, and remove those e-mail messages from your personal archive.
- So this demands the question of how we can program machines with the ability to look for patterns in text.
- Regular expressions provide such an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality.
- Python supports regexes through the standard library re module

- regexes are strings containing text and special characters that describe a pattern with which to recognize multiple strings.
- Regexs without special characters

Regex Pattern	String(s) Matched
foo	foo
Python	Python
abc123	abc123

- These are simple expressions that match a single string
- Power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition

Special Symbols and Characters

Notation	Description	Example Regex
Symbols		
literal	Match literal string value *literal*	`foo`
re1|*re2*	Match regular expressions *re1* or *re2*	*foo*|*bar*
.	Match *any character* (except \n)	`b.b`
^	Match *start of string*	`^Dear`
$	Match *end of string*	`/bin/*sh$`
*	Match *0 or more* occurrences of preceding regex	`[A-Za-z0-9]*`
+	Match *1 or more* occurrences of preceding regex	`[a-z]+\.com`
?	Match *0 or 1* occurrence(s) of preceding regex	`goo?`

Special Symbols and Characters

{*N*}	Match *N* occurrences of preceding regex	`[0-9]{3}`
{*M*,*N*}	Match from *M* to *N* occurrences of preceding regex	`[0-9]{5,9}`
[...]	Match any single character from *character* class	`[aeiou]`
[..*x*-*y*..]	Match any single character in the *range from* x to y	`[0-9],[A-Za-z]`

Special Symbols and Characters

Symbols		
[^...]	*Do not match* any character from character class, including any ranges, if present	[^aeiou], [^A-Za-z0-9_]

Matching Any Single Character (.)

- dot or period (.) symbol (letter, number, whitespace (not including "\n"), printable, non-printable, or a symbol) matches any single character except for \n
- To specify a dot character explicitly, you must escape its functionality with a backslash, as in "\."

Regex Pattern	Strings Matched
f.o	Any character between "f" and "o"; for example, fao, f9o, f#o, etc.
..	Any pair of characters
.end	Any character before the string end

```
import re
if re.match("f.o","fooo"):
    print("Matched")
else:
    print("Not matched")
```

Output:

Prints matched

Since it searches only for the pattern 'f.o' in the string

```
import re
if re.match("f.o$","fooo"):
    print("Matched")
else:
    print("Not matched")
```

Check that the entire string starts with ‘f’, ends with ‘o’ and contain one letter in between

```
import re
if re.match("..","fooo"):
    print("Matched")
else:
    print("Not matched")
```

Matched

Two dots matches any pair of characters.

```
import re
if re.match("..$","foo"):
    print("Matched")
else:
    print("Not matched")
```

Not matched

Including a ‘$’ at the end will match only strings of length 2

```
import re
if re.match(".end","bend"):
    print("Matched")
else:
    print("Not matched")
```

Matched

The expression used in the example, matches any character for ‘.’

```
import re
if re.match(".end","bends"):
    print("Matched")
else:
    print("Not matched")
```

Prints Matched

```
import re
if re.match(".end$","bends"):
    print("Matched")
else:
    print("Not matched“)
```

Prints Not matched - $ check for end of string

Matching from the Beginning or End of Strings or Word Boundaries (^, $)

^ - Match beginning of string

$ - Match End of string

Regex Pattern	Strings Matched
^From	Any string that starts with From
/bin/tcsh$	Any string that ends with /bin/tcsh
^Subject: hi$	Any string consisting solely of the string Subject: hi

if you wanted to match any string that ended with a dollar sign, one possible regex solution would be the pattern .*$

Register number validity

Check whether the given register number of a VIT student is valid or not.

Example register number – 15bec1032

Register number is valid if it has two digits

Followed by three letters

Followed by four digits

Denoting Ranges (-) and Negation (^)

- brackets also support ranges of characters
- A hyphen between a pair of symbols enclosed in brackets is used to indicate a range of characters;
- For example A–Z, a–z, or 0–9 for uppercase letters, lowercase letters, and numeric digits, respectively

Regex Pattern	Strings Matched
`z.[0-9]`	"z" followed by any character then followed by a single digit
`[r-u][env-y][us]`	"r," "s," "t," or "u" followed by "e," "n," "v," "w," "x," or "y" followed by "u" or "s"
`[^aeiou]`	A non-vowel character (Exercise: why do we say "non-vowels" rather than "consonants"?)
`[^\t\n]`	Not a TAB or \n
`["-a]`	In an ASCII system, all characters that fall between "" and "a," that is, between ordinals 34 and 97

Multiple Occurrence/Repetition Using Closure Operators (*, +, ?, {})

- special symbols *, +, and ?, all of which can be used to match single, multiple, or no occurrences of string patterns
- Asterisk or star operator (*) - match zero or more occurrences of the regex immediately to its left
- Plus operator (+) - Match one or more occurrences of a regex

- Question mark operator (?) - match exactly 0 or 1 occurrences of a regex.
- There are also brace operators ({}) with either a single value or a comma-separated pair of values. These indicate a match of exactly N occurrences (for {N}) or a range of occurrences; for example, {M, N} will match from M to N occurrences

Code to check the validity of register number

```
import re
register= input()
if re.match("^[1-9][0-9][a-zA-Z][a-zA-Z][a-zA-Z][0-9][0-9][0-9][0-9]$",register):
    print("Matched")
else:
    print("Not matched")
```

^ - denote begin (Meaning is different when we put this symbol inside the square bracket)

$ - denote end

Regex Pattern	Strings Matched
`[dn]ot?`	"d" or "n," followed by an "o" and, at most, one "t" after that; thus, `do`, `no`, `dot`, `not`.
`0?[1-9]`	Any numeric digit, possibly prepended with a "0." For example, the set of numeric representations of the months January to September, whether single or double-digits.
`[0-9]{15,16}`	Fifteen or sixteen digits (for example, credit card numbers.

Refined Code to check the validity of register number

{n} – indicate that the pattern before the braces should occur n times

```
import re
register= input()

if re.match("^[1-9][0-9][a-zA-Z]{3}[0-9]{4}$",register):
    print("Matched")
else:
    print("Not matched")
```

Check validity of Mobile Number (Shorter Code)

```
import re
number = input()
if re.match(‘[^0][0-9]{9}',number):
    print('valid')
else:
    print('invalid')
```

Bug: Will also accept a843338320

Check validity of Mobile Number (Shorter Code)

```
import re
number = input()
if re.match(‘[1-9][0-9]{9}',number):
    print('valid')
else:
    print('invalid')
```

Check validity of PAN card number with RE

```
import re
pan=input()
if len(pan) < 10 or len(pan) > 10 :
    print ("PAN Number should be 10 characters")
    exit




elif re.search("[^a-zA-Z0-9]",pan):
    print ("No symbols allowed, only
    alphanumerics")
    exit
elif re.search("[0-9]",pan[0:5]):
    print ("Invalid - 1")
    exit
```

```
elif re.search("[A-Za-z]",pan[5:9]):
    print ("Invalid - 2")
    exit
elif re.search("[0-9]",pan[-1]):
    print ("Invalid - 3")
    exit
else:
    print ("Your card "+ pan + " is valid“)
```

Python read all input as string

In some cases it is necessary to check if the value entered is an integer

We can check it using regular expressions

Rules for an integer

optionally begin with a negative sign include ^ symbol

first digit must be a number other than zero

may be followed zero to any number of digits

string must end with it so add $ symbol

```
import re
register= input()
#optionally begin with a negative sign include ^
    symbol
#first digit must be a number other than zero
# may be followed zero to any number of digits
# string must end with it so add $ symbol




if re.match("^\-?[1-9][0-9]*$",register):
    #'\' is added in front of '-' to overcome its default
    meaning in REs
    print("Matched")
else:
    print("Not matched")
```

Rules for an integer or a floating point value

optionally begin with a negative sign include ^ symbol

first digit must be a number other than zero

may be followed zero to any number of digits

string must end with it so add $ symbol

Optionally followed by a '.'

Followed by zero or more digits

String ends here

```
import re
register= input()
if re.match("^\-?[1-9][0-9]*\.?[0-9]*$",register):
    # '.' can occur zero or one time followed by a digit occurred zero to infinite number of times
    print("Matched")
else:
    print("Not matched")
```

Problem

A farmer with a fox, a goose, and a sack of corn needs to cross a river. Now he is on the east side of the river and wants to go to west side. The farmer has a rowboat, but there is room for only the farmer and one of his three items. Unfortunately, both the fox and the goose are hungry. The fox cannot be left alone with the goose, or the fox will eat the goose. Likewise, the goose cannot be left alone with the sack of corn, or the goose will eat the corn. Given a sequence of moves find if all the three items fox, goose and corn are safe. The input sequence indicate the item carried by the farmer along with him in the boat. 'F' – Fox, 'C' – Corn, 'G' – Goose, N-Nothing. As he is now on the eastern side the first move is to west and direction alternates for each step.

Pseudocode

```
READ items_carried
SET east as G, C, F
SET west as empty
SET from_Dir = east and to_Dir = west
FOR each item in items_carried
        IF from_Dir == east THEN
                remove item from east and add to west
                IF east or west contains 'C' and 'G' or 'G' and 'F' THEN
                        PRINT 'NOT SAFE'
                        BREAK
        ELSE
                remove item from west and add to east
                IF east or west contains 'C' and 'G' or 'G' and 'F' THEN
                        PRINT 'NOT SAFE'
                        BREAK
        END IF
IF from_Dir == east THEN
        SET from_Dir = west
        SET to_Dir = east
ELSE
        SET from_Dir = east
        SET to_Dir = west
END IF
END FOR
PRINT 'SAFE'
```

While going for a Shopping ….!!!???

Imagine you have the scores in “Python Programming” for 100 students. If you want to find the average score in Python…?

Simple Statistics

- Many programs deal with large collections of similar information.
  - Words in a document
  - Students in a course
  - Data from an experiment
  - Customers of a business
  - Graphics objects drawn on the screen
  - Cards in a deck

List

Examples

- Apple, Banana, Berry, Mango
- Football, Basketball, Throwball, Tennis, Hockey
- Sunrise, Sugar, Cheese, Butter, Pickle, Soap, Washing Powder, Oil….
- Agra, Delhi, Kashmir, Jaipur, Kolkata…

Introduction

- Contains multiple values that are logically related
- List is a type of mutable sequence in Python
- Each element of a list is assigned a number – index / position
- Can do indexing, slicing, adding, multiplying, and checking for membership
- Built-in functions for finding length of a sequence and for finding its largest and smallest elements

What is a List?

- Most flexible data type in Python
- Comma-separated items can be collected in square brackets
- Good thing is..
  - THE ITEMS IN THE LIST NEED NOT BE OF SAME TYPE

Creating a list

- Creating an EMPTY list

  listname = []

Example:

L1 = []

MyList = []

Books = []

- Creating a list with items

listname = [item1, item2, ….]

Example:

Temp = [100, 99.8, 103, 102]

S = [‘15BIT0001’, ‘Achu’, 99.9]

L2 = [1, 2, 3, 4, 5, 6, 7]

Course = [‘Python’, ‘C’, ‘C++’, ‘Java’]

Accessing Values

- Using index or indices

```
>>>L1 = [1, 2, 3, 4, 5, 6]
>>>print (L1[3]) #indexing
>>>4
>>>print (L1[2:5]) #slicing
>>>[3, 4, 5]
```

Updating Elements

- Update an element in list using index

```
>>>L1 = [1, 2, 3, 4, 5, 6]
>>>L1[2] = 111
>>>L1
    [1, 2, 111, 4, 5, 6]
```

Deleting Elements

- Delete an element in list using index

```
>>>L1 = [1, 2, 3, 4, 5, 6]
>>>del (L1[4])
>>>L1
    [1, 2, 111, 4, 6]
```

Basic Operations in List

- >>> len([1, 2, 3]) # Length
  3
- >>> [1, 2, 3] + [4, 5, 6] # Concatenation
  [1, 2, 3, 4, 5, 6]
- >>> ['Ni!'] * 4 # Repetition
  ['Ni!', 'Ni!', 'Ni!', 'Ni!']

Basic Operations in List

- >>> str([1, 2]) + "34" # Same as "[1, 2]" + "34"
  '[1, 2]34'
- >>> [1, 2] + list("34")
  # Same as [1, 2] + ["3", "4"]
  [1, 2, '3', '4‘]

List Iteration

- >>> 3 in [1, 2, 3] # Membership
  True
- >>> for x in [1, 2, 3]:
  print(x, end=' ')

Iteration (2.X uses: print x,) ... 1 2 3

List Comprehensions

>>> res = [c * 4 for c in 'SPAM']

List comprehensions

>>> res

['SSSS', 'PPPP', 'AAAA', 'MMMM']

- expression is functionally equivalent to a for loop that builds up a list of results manually
- list comprehensions are simpler to code and likely faster to run today:

List Comprehensions

```
# List comprehension equivalent ...
>>> res = []
>>> for c in 'SPAM':
        res.append(c * 4)
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

Map

- map built-in function applies a function to items in a sequence and collects all the results in a new list
- >>> list(map(abs, [−1, −2, 0, 1, 2]))
- # Map a function across a sequence

  [1, 2, 0, 1, 2]

Indexing, Slicing

```
>>> L = ['spam', 'Spam', 'SPAM!']

>>> L[2]              # Offsets start at zero

'SPAM!'

>>> L[−2]             # Negative: count from the right

'Spam'

>>> L[1:]             # Slicing fetches sections

['Spam', 'SPAM!']
```

Matrixes

- a basic 3 × 3 two-dimensional list-based array:
  >>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
- With one index, you get an entire row (really, a nested sublist), and with two, you get an item within the row:

```
>>> matrix[1]

[4, 5, 6]
```

Matrixes

```
>>> matrix[1][1]
5
>>> matrix[2][0]
7
>>> matrix = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]
>>> matrix[1][1]
5
```

Insertion, Deletion and Replacement

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5]        # Replacement/insertion
>>> L
[1, 4, 5, 3]
>>> L[1:1] = [6, 7]    # Insertion (replace nothing)
>>> L
[1, 6, 7, 4, 5, 3]
>>> L[1:2] = []              # Deletion (insert nothing)
>>> L
[1, 7, 4, 5, 3]
```

Insertion, Deletion and Replacement

```
>>> L = [1]

>>> L[:0] = [2, 3, 4]

# Insert all at :0, an empty slice at front

>>> L

[2, 3, 4, 1]

>>> L[len(L):] = [5, 6, 7]

 # Insert all at len(L):, an empty slice at end

>>> L

[2, 3, 4, 1, 5, 6, 7]
```

List method calls

```
>>> L = ['eat', 'more', 'SPAM!']

>>> L.append('please')

# Append method call: add item at end

>>> L

['eat', 'more', 'SPAM!', 'please']

>>> L.sort()                    # Sort list items ('S' < 'e')

>>> L

['SPAM!', 'eat', 'more', 'please']
```

More on Sorting Lists

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                          # Sort with mixed case
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)             # Normalize to lowercase
>>> L
['abc', 'ABD', 'aBe']
```

More on Sorting Lists

```
>>> L.sort(key=str.lower, reverse=True)
# Change sort order
>>> L ['aBe', 'ABD', 'abc']
```

Other common list methods

```
>>> L = [1, 2]
>>> L.extend([3, 4, 5])
# Add many items at end (like in-place +)
>>> L [1, 2, 3, 4, 5]
>>> L.pop()
# Delete and return last item
5
```

Other common list methods

```
>>> L
[1, 2, 3, 4]
>>> L.reverse()                    # In-place reversal method
>>> L
 [4, 3, 2, 1]
>>> list(reversed(L))   # Reversal built-in with a result
    (iterator)
[1, 2, 3, 4]
```

Other common list methods

```
>>> L = ['spam', 'eggs', 'ham']
>>> L.index('eggs')       # Index of an object (search/find)
1
>>> L.insert(1, 'toast') # Insert at position
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove('eggs')         # Delete by value
>>> L
['spam', 'toast', 'ham']
```

Other common list methods

```
>>> L.pop(1)                  # Delete by position 'toast'
>>> L
['spam', 'ham']
>>> L.count('spam')          # Number of occurrences 1
1
```

Other common list methods

```
>>> L = ['spam', 'eggs', 'ham', 'toast']
>>> del L[0]                    # Delete one item
>>> L
['eggs', 'ham', 'toast']
>>> del L[1:]                   # Delete an entire section
>>> L                           # Same as L[1:] = []
['eggs‘]
```

Other common list methods

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

Hence…

- A list is a sequence of items stored as a single object.
- Items in a list can be accessed by indexing, and sub lists can be accessed by slicing.
- Lists are mutable; individual items or entire slices can be replaced through assignment statements.
- Lists support a number of convenient and frequently used methods.
- Lists will grow and shrink as needed.

Strings and Lists

```
>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
>>> L[3] = 'x'                    # Works for lists, not strings
>>> L[4] = 'x‘
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
>>> S = “ “.join(L)    #uses ‘’ for joining elements of list
>>> S
'spaxxy‘
```

Strings and Lists

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast‘
# uses ‘SPAM’ for joining elements of list
>>> line = 'aaa bbb  ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

Statistics using List

Find the mean, standard deviation and median of a set of numbers

Table 8-1. Common list literals and operations

Operation	Interpretation
`L = []`	An empty list
`L = [123, 'abc', 1.23, {}]`	Four items: indexes 0..3
`L = ['Bob', 40.0, ['dev', 'mgr']]`	Nested sublists
`L = list('spam')`	List of an iterable's items, list of successive integers
`L = list(range(-4, 4))`	
`L[i]`	Index, index of index, slice, length
`L[i][j]`	
`L[i:j]`	
`len(L)`	
`L1 + L2`	Concatenate, repeat

Operation	Interpretation
`L * 3`	
`for x in L: print(x)`	Iteration, membership
`3 in L`	
`L.append(4)`	Methods: growing
`L.extend([5,6,7])`	
`L.insert(i, X)`	
`L.index(X)`	Methods: searching
`L.count(X)`	
`L.sort()`	Methods: sorting, reversing,
`L.reverse()`	copying (3.3+), clearing (3.3+)
`L.copy()`	
`L.clear()`	
`L.pop(i)`	Methods, statements: shrinking
`L.remove(X)`	
`del L[i]`	
`del L[i:j]`	
`L[i:j] = []`	
`L[i] = 3`	Index assignment, slice assignment
`L[i:j] = [4,5,6]`	
`L = [x**2 for x in range(5)]`	List comprehensions and maps (Chapter 4, Chapter 14, Chapter 20)
`list(map(ord, 'spam'))`	

Exercise 1

Given a positions of coins of player1 and player2 in a 3X3 Tic Tac Toc board, write a program to determine if the board position is a solution and if so identify the winner of the game. In a Tic Tac Toc problem, if the coins in a row or column or along a diagonal is of the same player then he has won the game. Assume that player1 uses ‘1’ as his coin and player2 uses '2' as his coin. '0' in the board represent empty cell.

Exercise 2

- In a supermarket there are two sections S1 and S2. The sales details of $item_1$ to $item_n$ of section1 and $item_1$ to $item_p$ of section2 are maintained in a sorted order. Write a program to merge the elements of the two sorted lists to form the consolidated list.

Exercise 3

- Watson gives Sherlock an list of *N numbers*. Then he asks him to determine if there exists an element in the list such that the sum of the elements on its left is equal to the sum of the elements on its right. If there are no elements to the left/right, then the sum is considered to be zero.

Exercise 4

- Sunny and Johnny together have *M* dollars they want to spend on ice cream. The parlor offers *N* flavors, and they want to choose two flavors so that they end up spending the whole amount.
- You are given the cost of these flavors. The cost of the *ith* flavor is denoted by *ci*. You have to display the indices of the two flavors whose sum is *M*.

Exercise 5

- Given a list of integer values, find the fraction of count of positive numbers, negative numbers and zeroes to the total numbers. Print the value of the fractions correct to 3 decimal places.

Exercise 6

- Given *N* integers, count the number of pairs of integers whose difference is *K*.

Problem

Write a kids play program that prints the capital of a country given the name of the country.

PAC For Quiz Problem

Input	Processing	Output
A set of question/ answer pairs and a question	Map each question to the corresponding answer. Find the answer for the given question	Answer for the question

Pseudocode

```
READ num_of_countries
FOR i=0 to num_of_countries
        READ name_of_country
        READ capital_of_country
        MAP name_of_country to capital_of_country
END FOR
READ country_asked
GET capital for country_asked
PRINT capital
```


Already we know

- To read values from user
- Print values
- We have to yet know to
  - Map a pair of values

Python provides Dictionaries to Map a pair of values

Introduction to Dictionaries

- Pair of items
- Each pair has key and value
- Keys should be unique
- Key and value are separated by :
- Each pair is separated by ,

Example:

dict = {‘Alice’ : 1234, ‘Bob’ : 1235}

Properties of Dictionaries

- unordered mutable collections;
- items are stored and fetched by key,
- Accessed by key, not offset position
- Unordered collections of arbitrary objects
- Variable-length, heterogeneous, and arbitrarily nestable

Creating a Dictionary

- Creating an EMPTY dictionary

  dictname = {}

Example:

Dict1 = {}

MyDict = {}

Books = {}

- Creating a dictionary with items

dictname = {key1:val1, key2:val2, ….}

Example:

MyDict = { 1 : 'Chocolate', 2 : 'Icecream'}

MyCourse = {'MS' : 'Python', 'IT' : 'C', 'CSE' : 'C++', 'MCA' : 'Java'}

MyCircle = {'Hubby':9486028245, 'Mom':9486301601}

Accessing Values

- Using keys within square brackets

>>> print (MyDict[1])

'Chocholate'

>>> print (MyCourse['CSE'])

'C++'

Updating Elements

- update by adding a new item (key-value) pair
- modify an existing entry

```
>>>MyDict[1] = 'Pizza'
>>>MyCourse['MCA'] = 'UML'
```


Deleting Elements

- remove an element in a dictionary using the key

```
>>>del MyCourse['IT']
```

- remove all the elements

```
>>>MyCourse.clear()
```

- delete the dictionary

```
>>>del MyCourse
```

Basic Operations

```
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> len(D) # Number of entries in dictionary 3
>>> 'ham' in D # Key membership test
True
>>> list(D.keys())  # Create a new list of D's keys
['eggs', 'spam', 'ham']
```


Basic Operations

```
>>> list(D.values()) [3, 2, 1]
>>> list(D.items()) [('eggs', 3), ('spam', 2), ('ham', 1)]
>>> D.get('spam')            # A key that is there
2
>>> print(D.get('toast'))    # A key that is missing
None
```

Update Method

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
>>> D2 = {'toast':4, 'muffin':5}
>>> D.update(D2)
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
#unordered
```


Pop Method

Delete and return value for a given key

```
>>> D = {'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2,
   'ham': 1}
>>> D.pop('muffin')
5
>>> D.pop('toast‘)
4
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
```

List vs Dictionary

```
>>>  L = []
>>> L[99] = 'spam'
Traceback (most recent call last):  File "<stdin>", line 1, in ? IndexError: list assignment index out of range
>>>D = {}
>>> D[99] = 'spam'
>>> D {99: 'spam'}
```


Nesting in dictionaries

```
>>> jobs = []
>>> jobs.append('developer')
>>> jobs.append(‘manager‘)
```

Nesting in dictionaries

```
rec = {}
>>> rec['name'] = 'Bob'
>>> rec['age']  = 40.5
>>> rec['job']  = jobs
```


Nesting in dictionaries

```
>>> rec
{'name': 'Bob', 'age': 40.5, 'job': ['developer',
   'manager']}
```

Nesting in dictionaries

```
>>> rec['name']
'Bob'
>>> rec['job']
['developer', 'manager']
>>> rec['job'][1]
'manager’
```


Other Ways to Make Dictionaries

```
D = {'name': 'Bob', 'age': 40}
D = {}                          # Assign by keys dynamically
D['name'] = 'Bob'
D['age']  = 40
# Creating a dictionary by assignment
dict(name='Bob', age=40)

# Creating dictionary with tuples form
dict([('name', 'Bob'), ('age', 40)])
```

Comprehensions in Dictionaries

```
>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}

>>> D

{'b': 2, 'c': 3, 'a': 1}

>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}

# Or: range(1, 5)
```


Comprehensions in Dictionaries

```
>>> D

{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'SPAM'}

>>> D

{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}
```

Comprehensions in Dictionaries

```
>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS',
   'HAM']}
>>> D
{'eggs': 'EGGS!', 'spam': 'SPAM!', 'ham': 'HAM!'}
```


Initializing Dictionaries

```
# Initialize dict from keys
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)
>>> D {'b': 0, 'c': 0, 'a': 0}
# Same, but with a comprehension
>>> D = {k:0 for k in ['a', 'b', 'c']}
>>> D {'b': 0, 'c': 0, 'a': 0}
```

Initializing Dictionaries

Comprehension

```
>>> D = {k: None for k in 'spam'}

>>> D {'s': None, 'p': None, 'a': None, 'm': None}
```


Dictionary methods

- `<dict>.items()`
  - displays the items in the dictionary (pair of keys and values)
- `<dict>.keys()`
  - display the keys in the dictionary
- `<dict>.values()`
  - displays the values in the dictionary
- `<dict>.pop()`
  - removes the last item from the dictionary
- `<dict2> = <dict1>.copy()`
  - copies the items from dict1 to dict2
- `<dict>.clear()`
  - removes all the items from the dictionary

Other methods

- `str(dict)`
  - produces printable string representation of a dictionary
- `len(dict)`
  - returns the number of items in the dictionary

Dictionaries can replace elif ladder

print ({1:'one',2:'two',3:'three',4:'four',5:'five'} [choice])

if choice = 3 then the code prints three

Tuples

Problem

A hospital has received a set of lab reports. Totally five tests are conducted in the lab and the report is prepared. The report consist of name of the test and the value observed for that particular patient. Given the details of a test made for a patient, write an algorithm and the subsequent Python program to print if the test result is normal or not normal by referring the values in Table 1. Since the value is sensitive, provide a mechanism so that the values do not get altered.

Name of the Test	Minimum Value	Maximum Value
Test1	20	30
Test2	35.5	40
Test3	12	15
Test4	120	150
Test5	80	120

PAC For Lab Test Problem

Input	Processing	Output
Test name and a pair of values indicating the minimum and the maximum value for the five tests Name of the test and the value observed	Check if the observed value is in the range of permissible values for the test and print 'Normal' or 'Abnormal'	Print 'Normal' or 'Abnormal'

Pseudocode LabTest Problem

```
FOR i =0 to 5
        READ test_Name_i
        READ minimum_i
        READ maximum_i
        Map test_Name_i to minimum_i and maximum_i
READ test_Name_Chk
READ observed_Value
END_FOR
```

```
IF observed_Value>min of test_Name_Chk
      and observed_Value <max of test_Name_Chk
            THEN
                  PRINT 'Normal'
             ELSE
                  PRINT 'Abnormal'
END IF
```

Store the values such that it is not getting modified

We Already Know

- To read values
- Map a value to another - Dictionary
- Print Values
- Form a pair of values – List – But the values can be changed
- Yet to learn about pairing values that cannot be modified

Tuples

Sequence of immutable Python objects

Tuples cannot be changed like lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values.

Optionally you can put these comma-separated values between parentheses also.

For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";
```

empty tuple −

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma −

```
a = (50)                # an integer

tup1 = (50,);           # tuple containing an integer
```

tuple indices start at 0

```
print ("tup1[0]: ", tup1[0]) # print physics
print ("tup2[1:5]: ", tup2[1:5]) # print (2,3,4,5)
```

Tuples in Action

```
>>> (1, 2) + (3, 4)   # Concatenation
(1, 2, 3, 4)

>>> (1, 2) * 4              # Repetition
(1, 2, 1, 2, 1, 2, 1, 2)

>>> T = (1, 2, 3, 4)        # Indexing, slicing

>>> T[0], T[1:3]
(1, (2, 3))
```

Sorted method in Tuples

```
>>> tmp = ['aa', 'bb', 'cc', 'dd']

>>> T = tuple(tmp)  # Make a tuple from the list's items

>>> T

('aa', 'bb', 'cc', 'dd')

>>> sorted(T)

['aa', 'bb', 'cc', 'dd‘]
```

List comprehensions can also be used with tuples.

The following, for example, makes a list from a tuple, adding 20 to each item along the way:

```
>>> T = (1, 2, 3, 4, 5)

>>> L = [x + 20 for x in T]
```

Equivalent to:

```
>>>L = []
>>>for x in T:
        L.append(x+20)
>>> L
[21, 22, 23, 24, 25]
```

Index method can be used to find the position of particular value in the tuple.

```
>>> T = (1, 2, 3, 2, 4, 2)
>>> T.index(2)          # Offset of first appearance of 2
1
>>> T.index(2, 2)      # Offset of appearance after offset 2
3
>>> T.count(2)          # How many 2s are there?
3
```

Nested Tuples

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'spam'                    # fails: can't change
tuple itself TypeError: object doesn't support item
assignment
```

```
>>> T[1][0] = 'spam'
# Works: can change mutables inside
>>> T
(1, ['spam', 3], 4)
```

```
>>> bob = ('Bob', 40.5, ['dev', 'mgr'])
# Tuple record
>>> bob
('Bob', 40.5, ['dev', 'mgr'])
>>> bob[0], bob[2]                    # Access by position
('Bob', ['dev', 'mgr'])
```

```
# Prepares a Dictionary record from tuple

>>> bob = dict(name='Bob', age=40.5, jobs=['dev',
'mgr'])

>>> bob
{'jobs': ['dev', 'mgr'], 'name': 'Bob', 'age': 40.5}

>>> bob['name'], bob['jobs']          # Access by key
('Bob', ['dev', 'mgr'])
```

Dictionary to Tuple

We can convert parts of dictionary to a tuple if needed:

```
>>> tuple(bob.values())              # Values to tuple
(['dev', 'mgr'], 'Bob', 40.5)
>>> list(bob.items())                # Items to list of tuples
[('jobs', ['dev', 'mgr']), ('name', 'Bob'), ('age', 40.5)]
```

Using Tuples

Immutable which means you cannot update or change the values of tuple elements

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
tup1[0] = 100;
```

You are able to take portions of existing tuples to create new tuples as the following example demonstrates

```
tup3 = tup1 + tup2;
print (tup3)
```

Delete Tuple Elements

Removing individual tuple elements is not possible

But possible to remove an entire tuple

```
tup = ('physics', 'chemistry', 1997, 2000);
print (tup)
del tup;
print ("After deleting tup : ")
print (tup)
```

Error

Basic Tuples Operations

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing

If L = ('spam', 'Spam', 'SPAM!')

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in Tuple Functions

```
tuple1, tuple2 = (123, 'xyz'), (456, 'abc')

len(tuple1)
2
```

When we have numerical tuple:

```
t1 = (1,2,3,7,4)

max(t1) #prints 7

min(t1) #prints 1
```

Converts a list into a tuple

```
tuple(seq)

t2=tuple([2,4])

>>> t2
(2, 4)
```

Sets

Problem:

An University has published the results of the term end examination conducted in April. List of failures in physics, mathematics, chemistry and computer science is available. Write a program to find the number of failures in the examination. This includes the count of failures in one or more subjects

PAC For University Result Problem

Input	Processing	Output
Read the register number of failures in Maths, Physics, Chemistry and Computer Science	Create a list of register numbers who have failed in one or more subjects Count the count of failures	Print Count

Pseudocode

```
READ maths_failure, physics_failure, chemistry_failure and cs_failure
Let failure be empty
FOR each item in maths_failure
        ADD item to failure
FOR each item in physics_failure
        IF item is not in failure THEN
                ADD item to failure
        END IF
FOR each item in chemistry_failure
        IF item is not in failure THEN
                ADD item to failure
        END IF
FOR each item in cs_failure
        IF item is not in failure THEN
                ADD item to failure
        END IF
SET count = 0
FOR each item in failure
        count = count + 1
PRINT count
```

Sets

an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory

Set is mutable

No duplicates

Sets are iterable, can grow and shrink on demand, and may contain a variety of object types

Does not support indexing

```
x = {1, 2, 3, 4}

y = {'apple','ball','cat'}

x1 = set('spam')  # Prepare set from a string
print (x1)
```

```
{'s', 'a', 'p', 'm'}
```

```
x1.add('alot')    # Add an element to the set
print (x1)
```

```
{'s', 'a', 'p', 'alot', 'm'}
```

Set Operations

Let S1 = {1, 2, 3, 4}

Union (|)

```
S2 = {1, 5, 3, 6} | S1
print(S2)          # prints {1, 2, 3, 4, 5, 6}
```

Intersection (&)

```
S2 = S1 & {1, 3}
print(S2) # prints {1, 3}
```

Difference (-)

```
S2 = S1 - {1, 3, 4}
print(S2) # prints {2}
```

Super set (>)

```
S2 = S1 > {1, 3}
print(S2) # prints True
```

Empty sets must be created with the set built-in, and print the same way

```
S2 = S1 - {1, 2, 3, 4}
print(S2) # prints set() – Empty set
```

Empty curly braces represent empty dictionary but not set

In interactive mode – type({}) gives

```
<class 'dict'>
```

```
>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}

>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> {1, 2, 3} | set([3, 4]) #Convert list to set and work
{1,2,3,4}
```

```
>>> {1, 2, 3}.union([3, 4]) #if you use union it will not give you the error
{1,2,3,4}

>>> {1, 2, 3}.union({3, 4})
{1,2,3,4}
```

Immutable constraints and frozen sets

Can only contain immutable (a.k.a. “hashable”) object types

lists and dictionaries cannot be embedded in sets, but tuples can if you need to store compound values.

Tuples compare by their full values when used in set operations:

```
>>> S
{1.23}

>>> S.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

```
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'
```

Works for tuples:

```
>>> S.add((1, 2, 3))
>>> S
{1.23, (1, 2, 3)}
```

```
>>> S | {(4, 5, 6), (1, 2, 3)}
{1.23, (4, 5, 6), (1, 2, 3)}
```

```
>>> (1, 2, 3) in S # Check for tuple as a whole
True
```

```
>>> (1, 4, 3) in S
False
```

clear()

All elements will removed from a set.

```
>>> cities = {"Stuttgart", "Konstanz", "Freiburg"}
>>> cities.clear()
>>> cities
set() # empty
>>>
```

Copy

Creates a shallow copy, which is returned.

```
>>> more_cities = {"Winterthur","Schaffhausen","St. Gallen"}
>>> cities_backup = more_cities.copy()
>>> more_cities.clear()
>>> cities_backup  # copied value is still available
{'St. Gallen', 'Winterthur', 'Schaffhausen‘}
```

Just in case, you might think, an assignment might be enough:

```
>>> more_cities = {"Winterthur","Schaffhausen","St. Gallen"}
>>> cities_backup = more_cities #creates alias name
>>> more_cities.clear()
>>> cities_backup
set()
>>>
```

The assignment "cities_backup = more_cities" just creates a pointer, i.e. another name, to the same data structure.

difference_update()

removes all elements of another set from this set. x.difference_update() is the same as "x = x - y"

```
>>> x = {"a","b","c","d","e"}
>>> y = {"b","c"}
>>> x.difference_update(y)
>>> x
{'a', 'e', 'd'}
```

discard(el)

el will be removed from the set, if it is contained in the set and nothing will be done otherwise

```
>>> x = {"a","b","c","d","e"}
>>> x.discard("a")

>>> x
{'c', 'b', 'e', 'd'}

>>> x.discard("z")

>>> x
{'c', 'b', 'e', 'd'}
```

remove(el)

works like discard(), but if el is not a member of the set, a KeyError will be raised.

```
>>> x = {"a","b","c","d","e"}
>>> x.remove("a")
>>> x
{'c', 'b', 'e', 'd'}

>>> x.remove("z")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
```

isdisjoint()

This method returns True if two sets have a null intersection

issubset()

x.issubset(y) returns True, if x is a subset of y

"<=" is an abbreviation for "Subset of" and ">=" for "superset of"

"<" is used to check if a set is a proper subset of a set

issuperset()

x.issuperset(y) returns True, if x is a superset of y. ">=" - abbreviation for "issuperset of“

">" - to check if a set is a proper superset of a set

```
>>> x = {"a","b","c","d","e"}
>>> y = {"c","d"}
>>> x.issuperset(y)
True
```

```
>>> x > y
True
>>> x >= y
True
>>> x >= x
True
>>> x > x
False
>>> x.issuperset(x)
True
```

```
>>> x = {"a","b","c","d","e"}
>>> y = {"c","d"}
>>> x.issubset(y)
False
>>> y.issubset(x)
True
```

```
>>> x < y
False
>>> y < x # y is a proper subset of x
True
>>> x < x # a set is not a proper subset of oneself.
False
>>> x <= x
True
```

pop()

pop() removes and returns an arbitrary set element.

The method raises a KeyError if the set is empty

```
>>> x = {"a","b","c","d","e"}
>>> x.pop()
'a'
>>> x.pop()
'c'
```

Sets themselves are mutable too, and so cannot be nested in other sets directly;

if you need to store a set inside another set, the frozenset built-in call works just like set but creates an immutable set that cannot change and thus can be embedded in other sets

To create frozenset:

```
cities = frozenset(["Frankfurt", "Basel","Freiburg"])

cities.add("Strasbourg")   #cannot modify

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Set comprehensions

run a loop and collect the result of an expression on each iteration

result is a new set you create by running the code, with all the normal set behavior

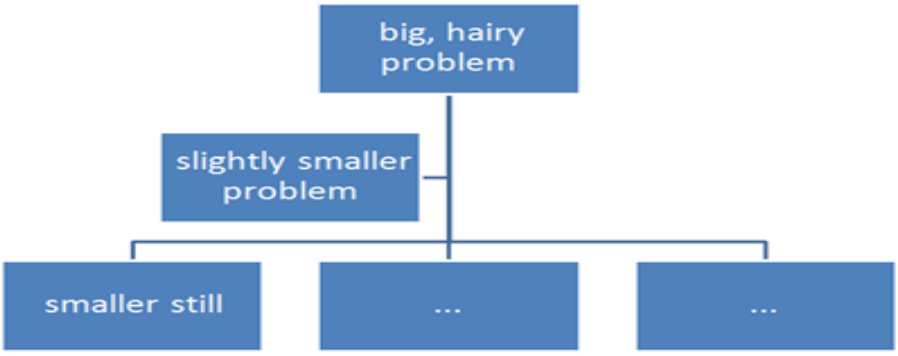
```
>>> {x ** 2 for x in [1, 2, 3, 4]}
{16, 1, 4, 9}

>>> {x for x in 'spam'}
{'m', 's', 'p', 'a'}
```

```
>>> S = {c * 4 for c in 'spam'}

>>> print(S)
{'pppp','aaaa','ssss', 'mmmm'}

>>>> S = {c * 4 for c in 'spamham'}
{'pppp','aaaa','ssss', 'mmmm','hhhh'}

>>>S | {'mmmm', 'xxxx'}
{'pppp', 'xxxx', 'mmmm', 'aaaa', 'ssss'}

>>> S & {'mmmm', 'xxxx'}
{'mmmm'}
```

```
math = set()
phy = set()
che = set()
cs = set()
m_N = int(input())
for i in range(0,m_N):
    val =input()
    math = math|{val}
m_P = int(input())
for i in range(0,m_P):
    val = input()
    phy = phy|{val}
m_C = int(input())
for i in range(0,m_C):
    val = input()
    che = che|{val}
```

```
m_CS = int(input())
for i in range(0,m_CS):
      val = input()
      cs = cs|{val}
failure = math|phy|che|cs
print(len(failure))
```

Functions



Triangle Formation Problem

Given three points, write an algorithm and the subsequent Python code to check if they can form a triangle. Three points can form a triangle, if they do not fall in a straight line and length of a side of triangle is less than the sum of length of other two sides of the triangle.

Triangle Formation Problem

For example, the points (5,10), (20,10) and (15,15) can form a triangle as they do not fall in a straight line and length of any side is less than sum of the length of the other two sides

Pseudocode for Triangle Formation

Read the three points

If the three points fall on a straight line then print "No Triangle" and break

Otherwise find length of all three sides

If length of one side is greater than the sum of length of the other two sides then print "Triangle" and print "No Triangle" otherwise

Pseudocode for Fall in Straight Line

```
input : X and Y coordinates of  the three points
IF (pt1.x==pt2.x==pt3.x) THEN
        RETURN true
ELIF (pt1.y==pt2.y==pt3.y) THEN
        RETURN true
ELSE
        RETURN false
```

Pseudocode for Distance between Two Points (Length of a side in a triangle)

input : X and Y coordinates of the two points

Distance = sqrt((pt1.x-pt2.x)**2 – (pt1.y-pt2.y)**2)

Return distance

Pseudocode for Checking Length Constraint

input : Length of three sides l1, l2, and l3

if l1<l2+l3 or l2<l1+l3 or l3<l1+l2:

return false

else:

return true

Sub Problems

- Next 8 slides recall the concept of dividing a problem to sub problems

Bigger Problems

- If you are asked to find a solution to a major problem, it can sometimes be very difficult to deal with the complete problem all at the same time.
- For example building a car is a major problem and no-one knows how to make every single part of a car.
- A number of different people are involved in building a car, each responsible for their own bit of the car's manufacture.
- The problem of making the car is thus broken down into smaller manageable tasks.
- Each task can then be further broken down until we are left with a number of step-by-step sets of instructions in a limited number of steps.
- The instructions for each step are exact and precise.

Top Down Design

- Top Down Design uses the same method to break a programming problem down into manageable steps.
- First of all we break the problem down into smaller steps and then produce a Top Down Design for each step.
- In this way sub-problems are produced which can be refined into manageable steps.

Top Down Design for Real Life Problem

PROBLEM: To repair a puncture on a bike wheel.

ALGORITHM:

1. remove the tyre
2. repair the puncture
3. replace the tyre

Step 1: Refinement:

1. Remove the tyre

1.1 turn bike upside down

1.2 lever off one side of the tyre

1.3 remove the tube from inside the tyre

Step 2: Refinement:

2. Repair the puncture Refinement:

2.1 find the position of the hole in the tube

2.2 clean the area around the hole

2.3 apply glue and patch

Step 3: Refinement:

3. Replace the tyre Refinement:

3.1 push tube back inside tyre

3.2 replace tyre back onto wheel

3.3 blow up tyre

3.4 turn bike correct way up

Still more Refinement:

Sometimes refinements may be required to some of the sub-problems, for example if we cannot find the hole in the tube, the following refinement can be made to 2.1:-

Still more Refinement:

Step 2.1: Refinement

2.1 Find the position of the hole in the tube

2.1.1 WHILE hole cannot be found

2.1.2 Dip tube in water

2.1.3 END WHILE

Python Functions

- A function has a name that is used when we need for the task to be executed.
- Asking that the task be executed is referred to as "calling" the function.
- Some functions need one or more pieces of input when they are called. Others do not.
- Some functions give back a value; others do not. If a function gives back a value, this is referred to as "returning" the value.

Why Functions?

- To reduce code duplication and increase program modularity.
- A software cannot be implemented by any one person, it takes a team of programmers to develop such a project.

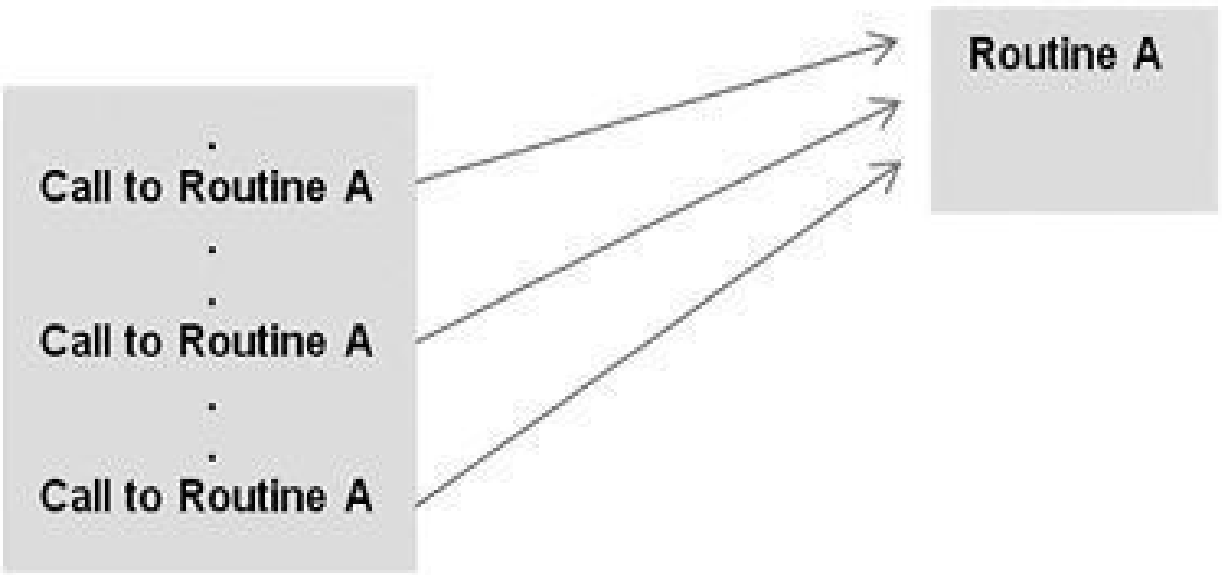
Term	Number of Lines of Code (LOC)	Equivalent Storage
KLOC	1,000	Application programs
MLOC	1,000,000	Operating systems / smart phones
GLOC	1,000,000,000	Number of lines of code in existence for various programming languages

Functions Contd…

- In order to manage the complexity of a large problem, it is broken down into smaller sub problems. Then, each sub problem can be focused on and solved separately.
- Program routines provide the opportunity for code reuse, so that systems do not have to be created from "scratch."

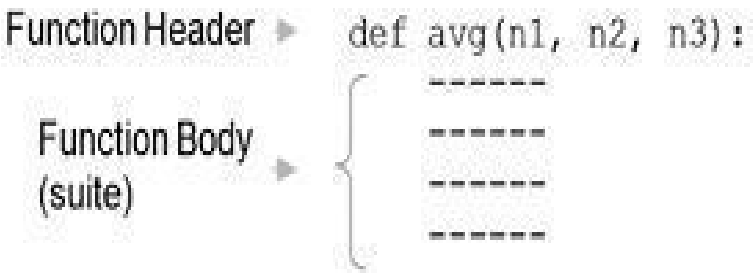
What Is a Function Routine?

- A **function or routine** is a named group of instructions performing some task.
- A routine can be **invoked** (*called*) as many times as needed in a given program
- When a routine terminates, execution automatically returns to the point from which it was called.



Defining Functions

- Functions may be designed as per user's requirement.
- The elements of a function definition are given



Defining Functions Contd…

- The number of items in a parameter list indicates the number of values that must be passed to the function, called **actual arguments** (or simply "arguments"), such as the variables num1, num2,and num3 below.

```
>>> num1 = 10
>>> num2 = 25
>>> num3 = 16

>>> avg(num1,num2,num3)
```

- *Every function must be defined before it is called*.

Parameters

- **Actual parameters**, or simply "arguments," - values passed to functions to be operated on.
- **Formal parameters**, or simply the "placeholder" names for the arguments passed.
- Actual parameters are matched with formal parameters by following the assignment rules

Assignment Statements Recap

Table 11-1. Assignment statement forms

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.X)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams + 42`)

Assignment Statements Recap

```
>>> [a, b, c] = (1, 2, 3)        # Assign tuple of values to list of names
>>> a, c
(1, 3)
>>> (a, b, c) = "ABC"
# Assign string of characters to tuple
>>> a, c
('A', 'C')
```

Assignment Statements Recap

```
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> print(a, b, c, d)
1 2 3 4
>>> a, b = seq
ValueError: too many values to unpack
(expected 2)
```

Assignment Statements Recap

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

Assignment Statements Recap

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

Assignment Statements Recap

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

Assignment Statements Recap

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])
>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm‘)
```

Assignment Statements Recap

```
>>> a, *b, c = range(4)
>>> a, b, c
(0, [1, 2], 3)
```

Example Functions

```
def printer(message):
        print('Hello ' + message)

def adder(a, b=1, *c):
        return a + b + c[0]
```

Example Functions

```
>>> def times(x, y):
        # Create and assign function ...
        return x * y
        # Body executed when called ...
```

When Python reaches and runs this def, it creates a new function object that packages the function's code and assigns the object to the name times

Calls

```
>>> times(2, 4)
# Arguments in parentheses prints 8
>>> x = times(3.14, 4)     # Save the result object
>>> x
12.56
>>> times('Ni', 4)         # Functions are "typeless"
'NiNiNiNi'
```

Example function

```
def intersect(seq1, seq2):
        res = []                # Start empty
        for x in seq1:          # Scan seq1
                if x in seq2:           # Common item?
                res.append(x)       # Add to end
return res
```

Example function

```
>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2)          # Strings
['S', 'A', 'M']
```

Equivalent Comprehension

```
>>> [x for x in s1 if x in s2]
['S', 'A', 'M']
```

Works for list also:

```
>>> x = intersect([1, 2, 3], (1, 4))      # Mixed types
>>> x                                     # Saved result object
[1]
```

Def Statements

creates a function object and assigns it to a name

def is a true executable statement: when it runs, it creates a new function object and assigns it to a name

Because it's a statement, a def can appear anywhere a statement can—even nested in other statements

Def Statements

```
if test:
        def func():          # Define func this way
else:
        def func():          # Or else this way      ... .
 func()
# Call the version selected and built
```

Function definition in selection statement Example

```
a = 4
if a%2==0:
        def func():
                print ('even')
else:
        def func():
                print('odd')
 func()
```

Output:

even

Function definition in selection statement Example

```
a = 4
if a%2==0:
        def func():
                print ('even')
else:
        def func():
                print('odd')
```

Function definition in selection statement Example

```
# error no function print1 is defined
 func()
```

Error in only condition satisfied item is found.
Otherwise code execute normal

Output:

```
even
```

Function definition in selection statement Example

```
a = 5
if a%2==0:
        def func():

                print ('even')
else:
        def func():
                print1('odd') # error no function print1 is defined
 func()
```

Output:

```
error
```

Function Call through variable

```
def one():
    print('one')

def two():
    print('two')

def three():
    print('three')
```

Function Call through variable

```
a = 3
 if a == 1:
    call_Func=one
elif a == 2:
    call_Func=two
else:
    call_Func=three
call_Func()
```

Scope of Variables

- Enclosing module is a global scope
- Global scope spans a single file only
- Assigned names are local unless declared global or nonlocal
- Each call to a function creates a new local scope

Name Resolution: The LEGB Rule

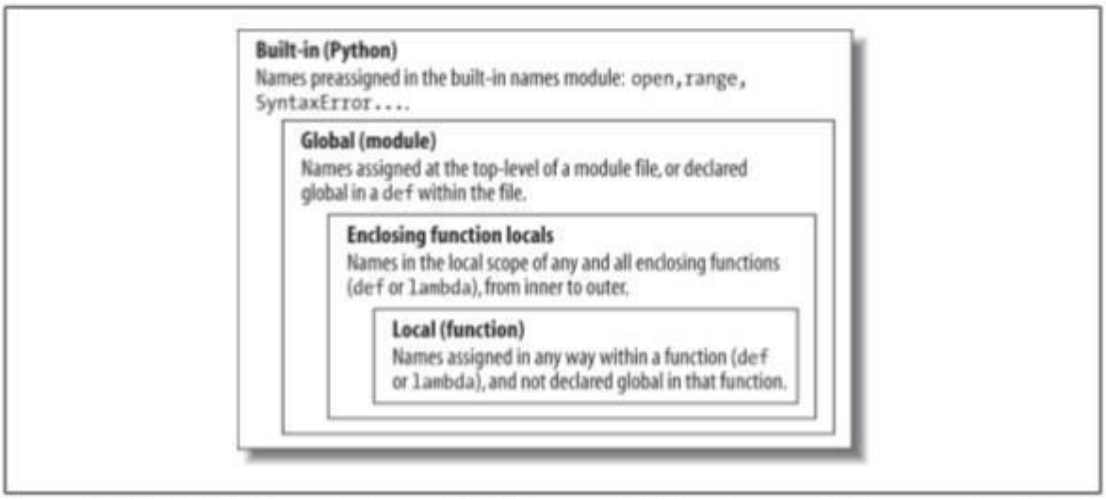


Figure 17-1. The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned usually determines its scope. In Python 3.X, nonlocal declarations can also force names to be mapped to enclosing function scopes, whether assigned or not.

Scope Example

```
# Global scope
X = 99
# X and func assigned in module: global
def func(Y):
        # Y and Z assigned in function: locals
        # Local scope
        Z = X + Y        # X is a global
        print (Z)
func(1)             # func in module: result=100
```

Scope Example

- Global names: X, func
- Local names: Y, Z

Scope Example

```
X = 88                  # Global X
def func():
        X = 99
# Local X: hides global
func()
print(X)                # Prints 88: unchanged
```

Accessing Global Variables

```
X = 88                  # Global X
def func():
   global X
   X = 99               # Global X: outside def
func()
print(X)                # Prints 99
```

Accessing Global Variables

```
y, z = 1, 2                    # Global variables in module
def all_global():
    global x                   # Declare globals assigned
    x = y + z
# No need to declare y, z: LEGB rule
```

Nested Functions

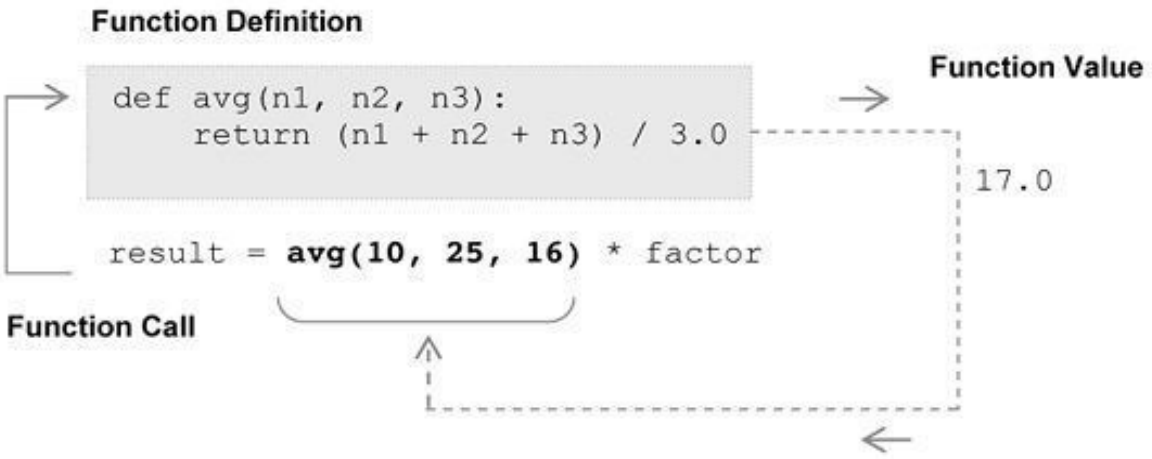
```
X = 99       # Global scope name: not used
def f1():
        X = 88          # Enclosing def local
        def f2():
                print(X)
        f2()
f1()                # Prints 88: enclosing def local
```

Return Functions

```
def f1():
    X = 88
    def f2():
        print(X)
    return f2          # Return f2 but don't call it
action = f1()          # Make, return function
action()               # Call it now: prints 88
```

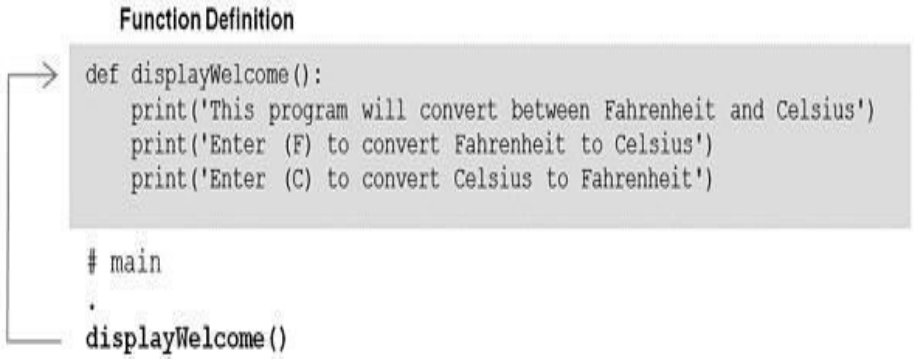
Value-Returning Functions

- Program routine called for its return value, and is therefore similar to a mathematical function.
- Function avg takes three arguments (n1, n2, and n3) and returns the average of the three.
- The *function call* avg(10, 25, 16), therefore, is an expression that evaluates to the returned function value.
- This is indicated in the function's *return statement* of the form return *expr*, where *expr* may be any expression.



Non-Value-Returning Functions

- A **non-value-returning function** is called not for a returned value, but for its *side effects*.
- A **side effect** is an action other than returning a function value, such as displaying output on the screen.



- In this example, function display Welcome is called only for the side-effect of the screen output produced.

Returning Multiple Values

- >>> def multiple(x, y):
        x = 2   # Changes local names only
        y = [3, 4]
    return x, y

Return multiple new values in a tuple

Returning Multiple Values

```
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)
# Assign results to caller's names
>>> X, L
(2, [3, 4])
```

```
x=int(input())
y=int(input())
pt1=(x,y)
x=int(input())
y=int(input())
pt2=(x,y)
x=int(input())
y=int(input())
No = False
pt3=(x,y)
if fall_St_line(pt1,pt2,pt3):
    print('No Triangle')
    No = True
else:
    dist = calc_Distance(pt1,pt2,pt3)
    if dist_Check(dist):
        print('No Triangle')
        No = True
    if not No:
        print('Triangle')
```

```
import math
def fall_St_line(pt1,pt2,pt3):
    #if x - coordinate of all points or
    # y-coordinate of all points are equal
    # then the points lie on the same straight line
    if (pt1[0]==pt2[0]==pt3[0])or(pt1[1]==pt2[1]==pt3[1]):
        return True
    else:
        return False
```

```
def calc_Distance(pt1,pt2,pt3):
    #distance between pt1 and pt2
    d1 = math.sqrt((pt1[0]-pt2[0])**2+(pt1[1]-pt2[1])**2)
    #distance between pt2 and pt3
    d2 = math.sqrt((pt2[0]-pt3[0])**2+(pt2[1]-pt3[1])**2)
    #distance between pt1 and pt3
    d3 = math.sqrt((pt1[0]-pt3[0])**2+(pt1[1]-pt3[1])**2)
    return (d1,d2,d3)
```

```
def dist_Check(dist):
    if dist[0]>(dist[1]+dist[2]):
        return True
    elif dist[1]>(dist[0]+dist[2]):
        return True
    elif dist[2]>(dist[1]+dist[0]):
        return True
    else:
        return False
```

Exercises

- Compute area of circle using function protypes.
- Compute Simple interest for given principle(P), number of years(N) and rate of interest(R). If R value is not given then consider R value as 10.5%.

Recursion to find GCD

```
def findgcd(x,y):
    while(x!=y):
        if(x>y):
            return findgcd(x-y,y)
        else:
            return findgcd(x,y-x)

    return x;


x=int(input("enter first number"))
y=int(input("enter second number"))
z=findgcd(x,y)
print(z)
```

Recursion to find Fibonacci series:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

```
n=int(input())
for i in range(0,n):
  print(fib(i))
```

Recursion to find factiorial

```
def fact(n):
  if n==1:
    return (1)
  else:
    return (n * fact(n-1))

n=int(input())
res=fact(n)
print(res)
```


Sorting

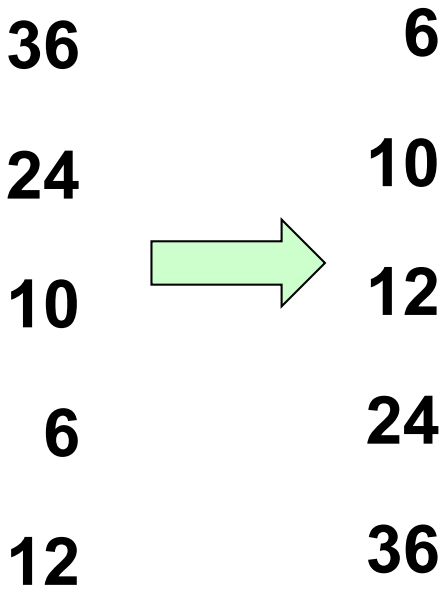
Session – 31 to 32

Problem

- Results of VIT entrance exam has been released. Given the details of the students such as name, address and marks scored in entrance, write a program to sort the student details so that it will be convenient to call for counselling.

Sorting means . . .

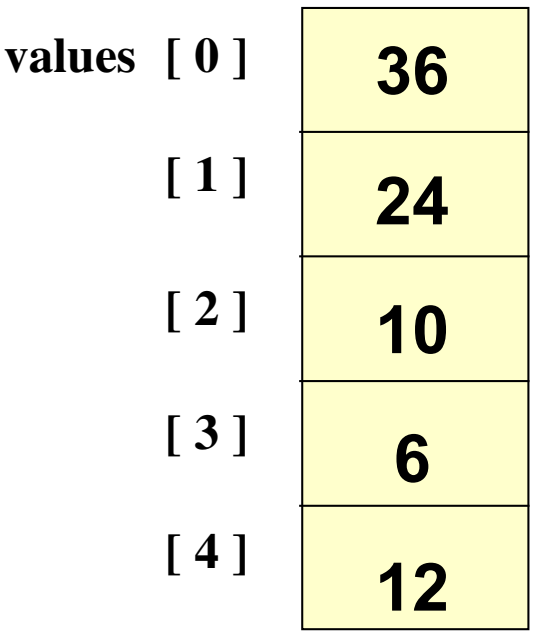
- Sorting rearranges the elements into either ascending or descending order within the array (we'll use ascending order).



Sorting also means…

- There are several sorting algorithms available like bubble sort, selection sort, insertion sort, quick sort, merge sort, radix sort etc.
- Sorting operation is performed in many applications to provide the output in desired order.
- For example listing all the product in the increasing order of their names or decreasing order of supplier names
- Searching will be easier in a sorted collection of elements
- List containing exam scores sorted from lowest to highest or vice versa
- We study Bubble Sorting, Selection Sorting and Insertion Sorting in this lab course

Bubble Sort



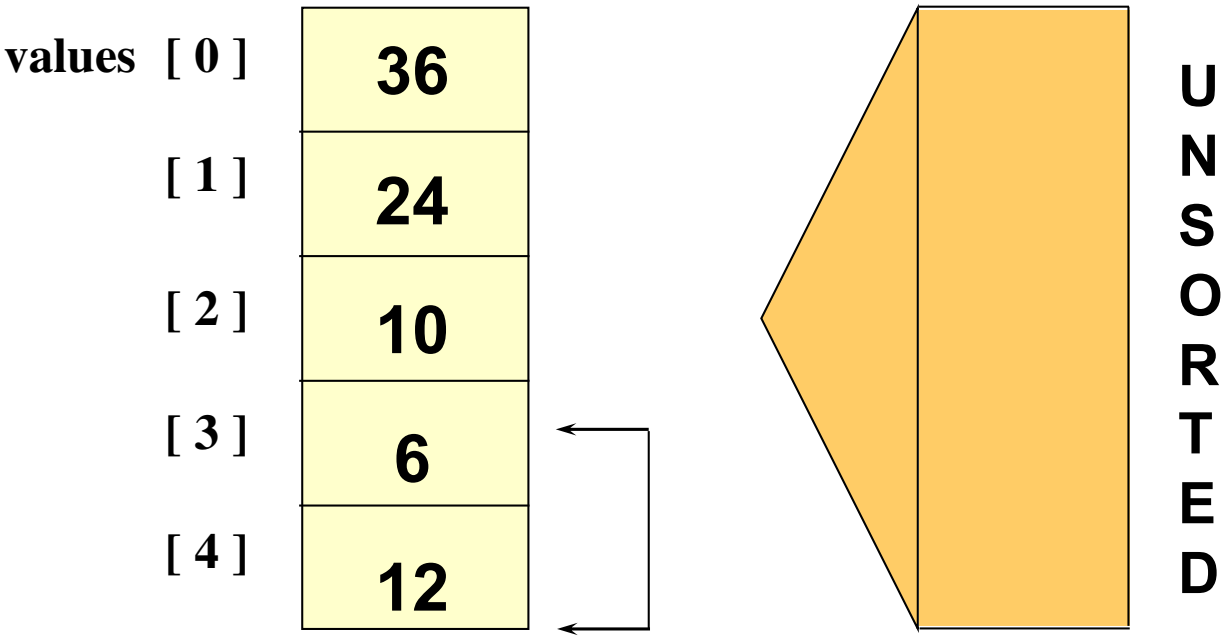
Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to "bubble up" to its correct place in the array.

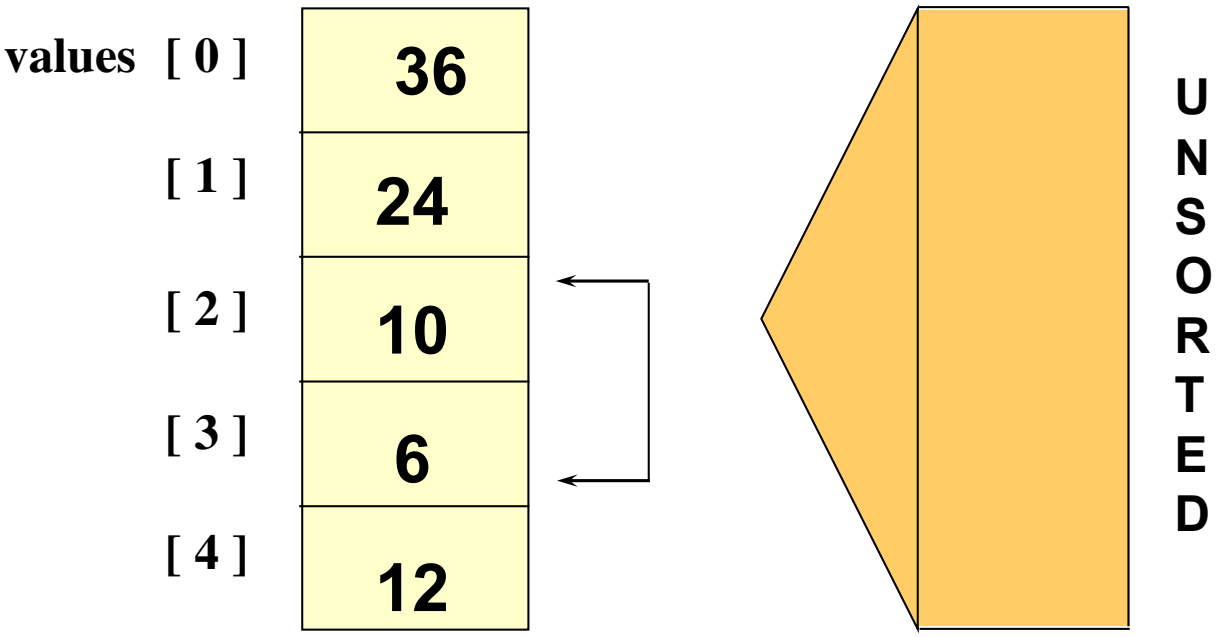
Bubble Sort Pseudo Code

```
def bubbleSort(lyst):
    n = len(lyst)
    while n > 1:                        # Do n - 1 bubbles
        i = 1                           # Start each bubble
        while i < n:
            if lyst[i] < lyst[i - 1]:   # Exchange if needed
                swap(lyst, i, i - 1)
            i += 1
        n -= 1
```

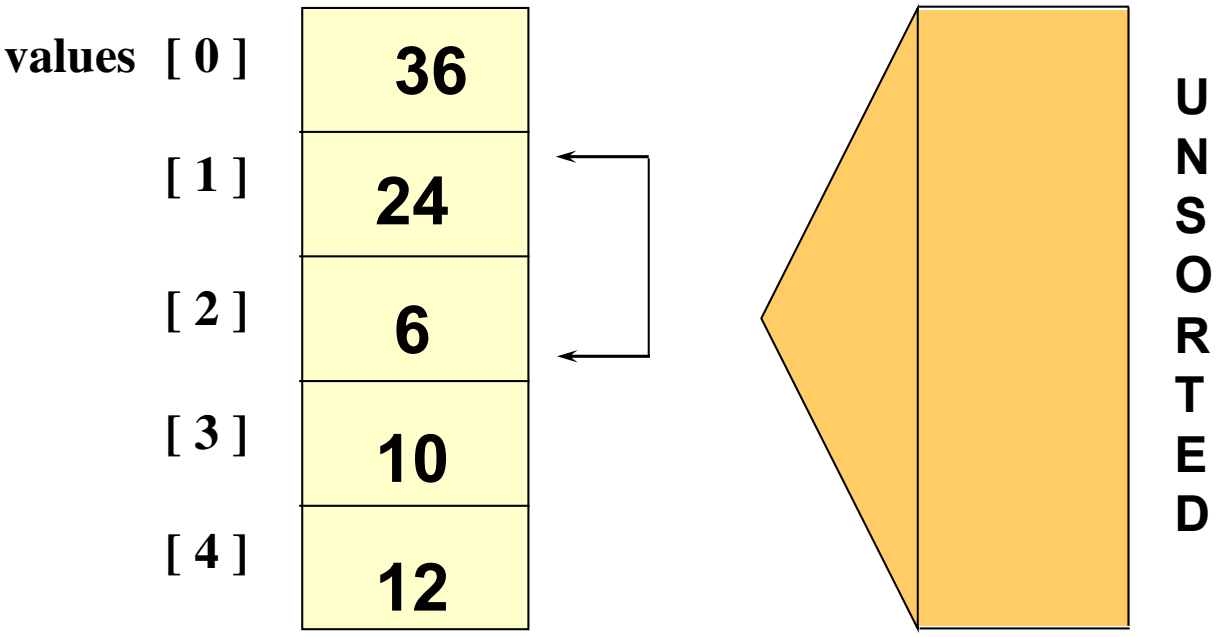

Bubble Sort: Pass One



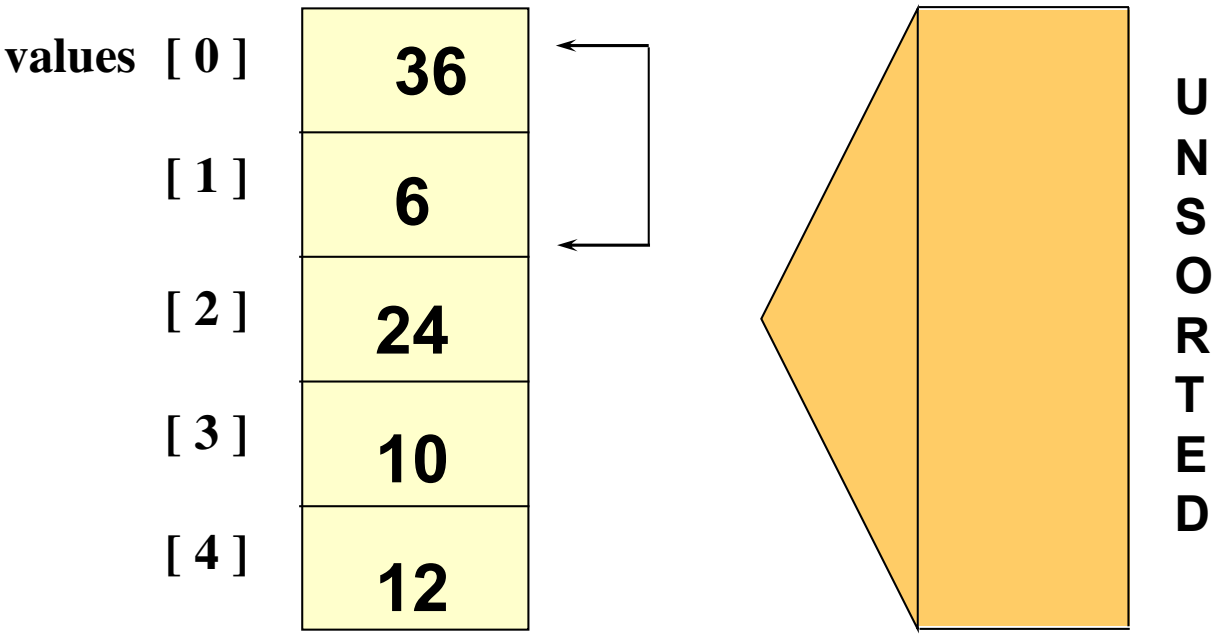
Bubble Sort: Pass One



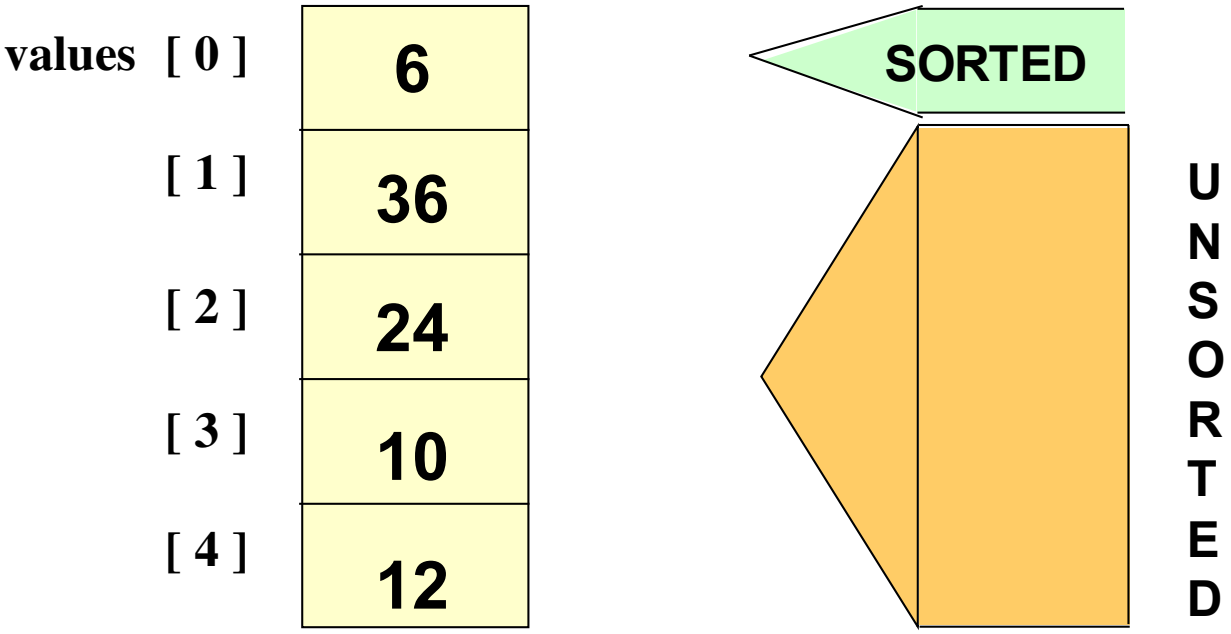
Bubble Sort: Pass One



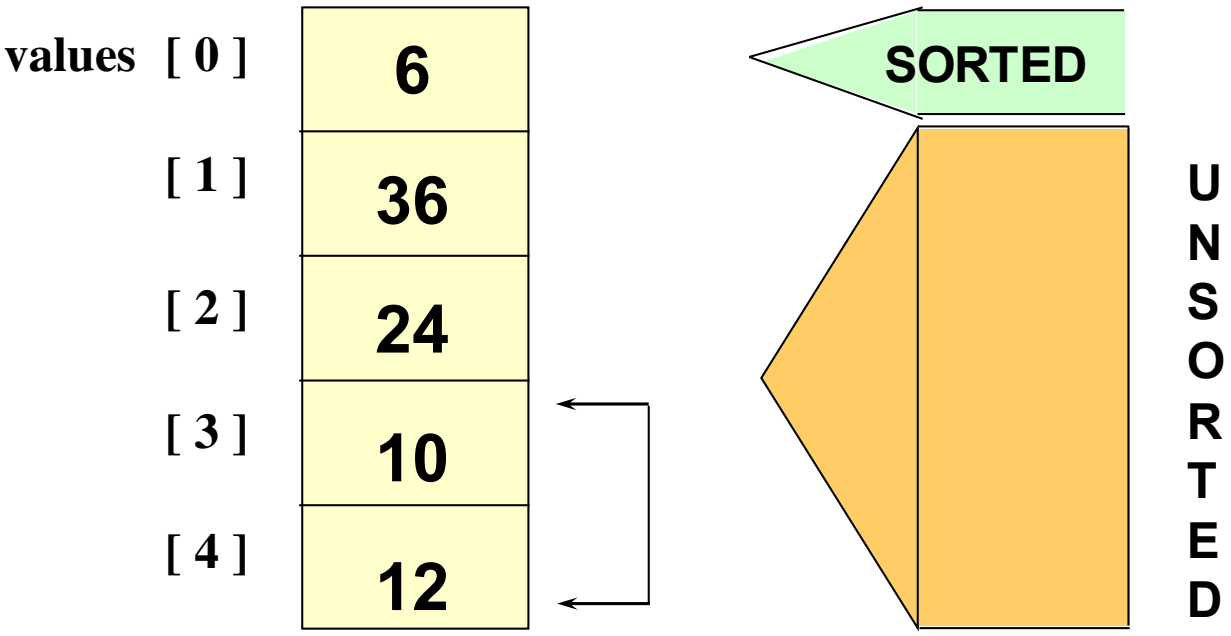
Bubble Sort: Pass One



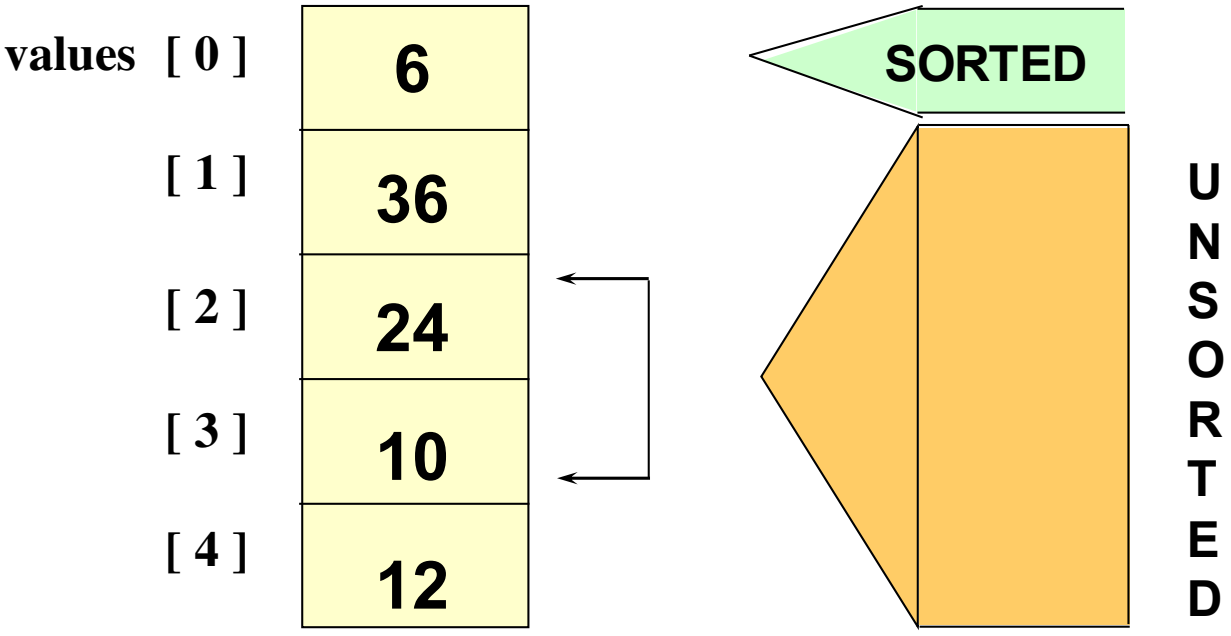
Bubble Sort: End Pass One



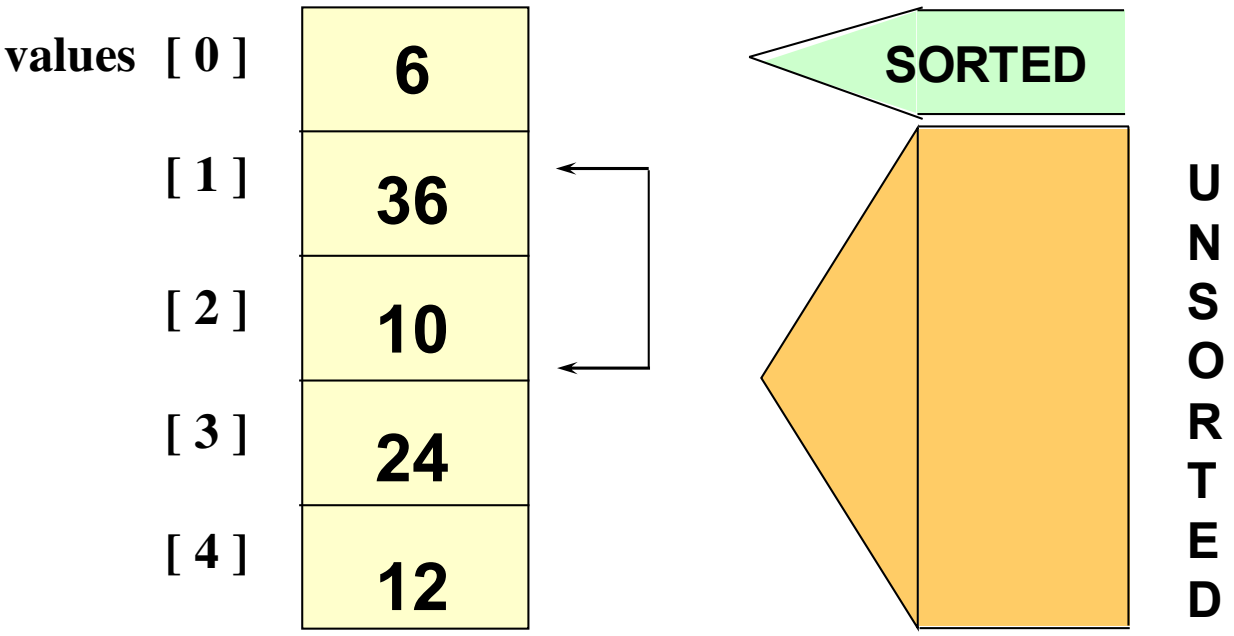
Bubble Sort: Pass Two



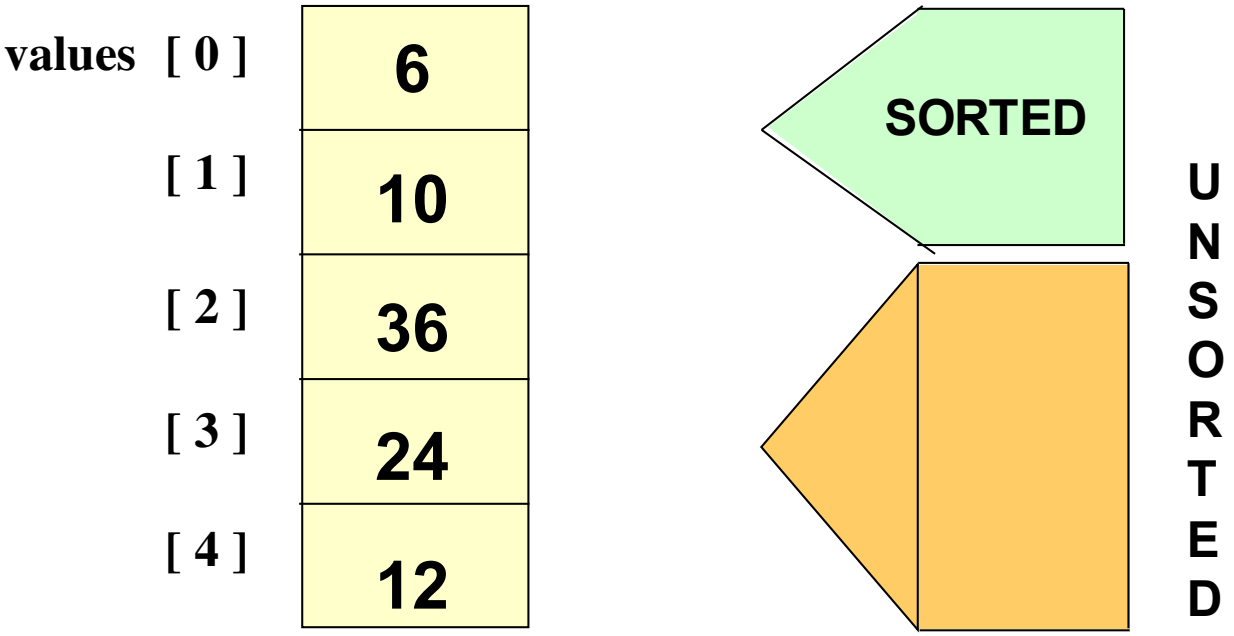
Bubble Sort: Pass Two



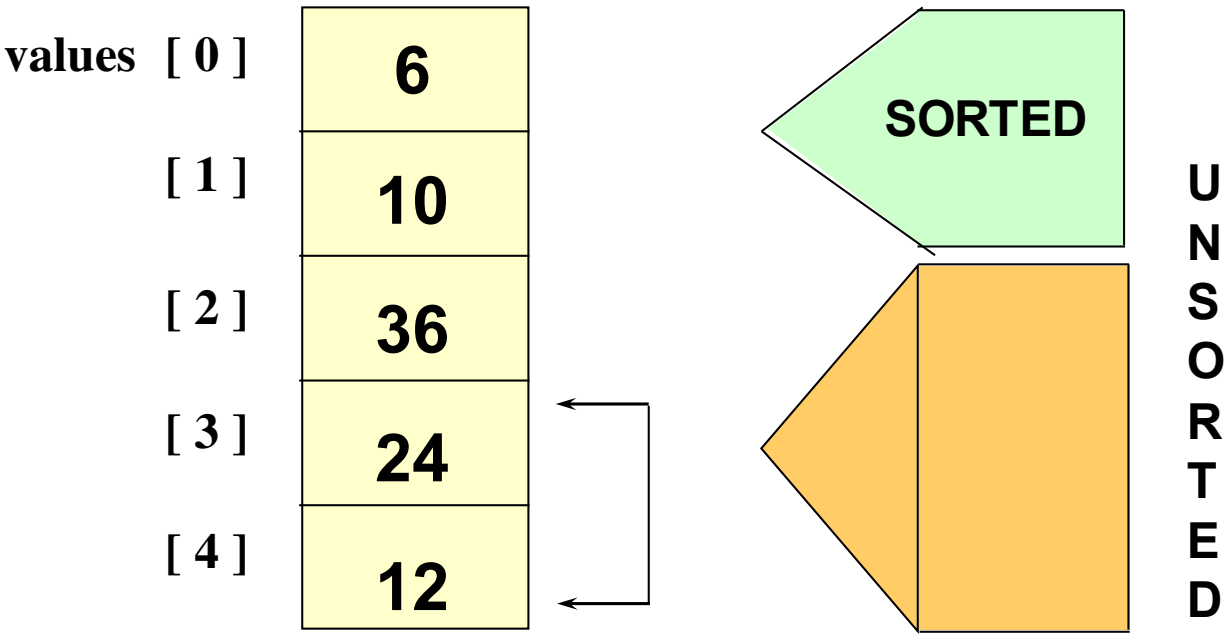
Bubble Sort: Pass Two



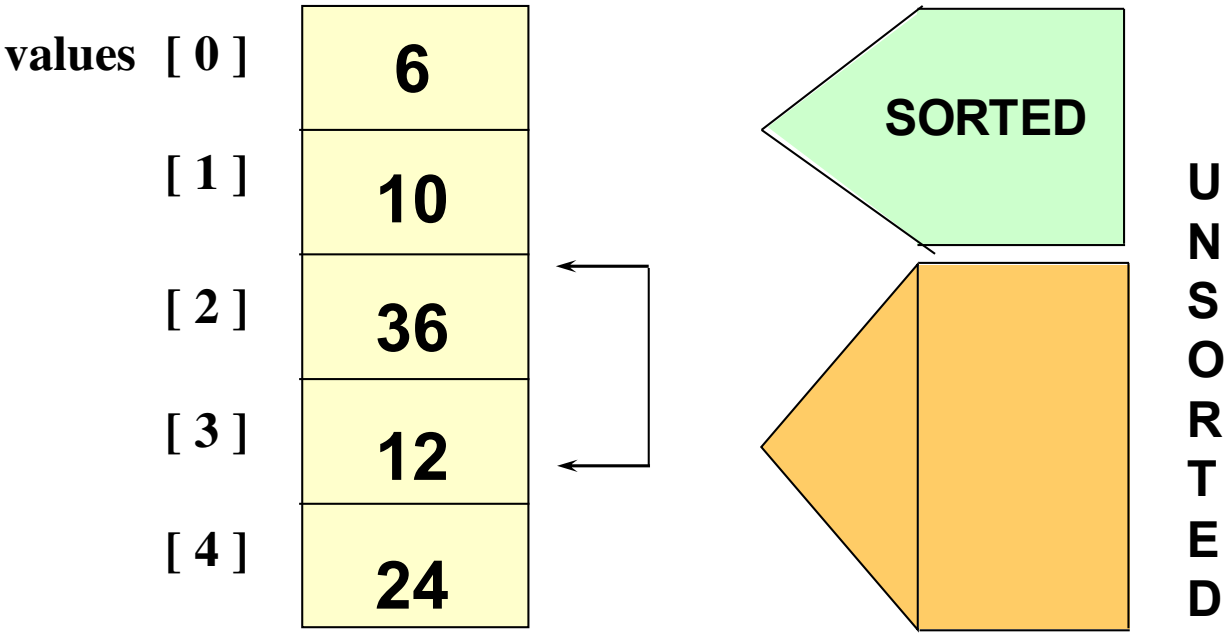
Bubble Sort: End Pass Two



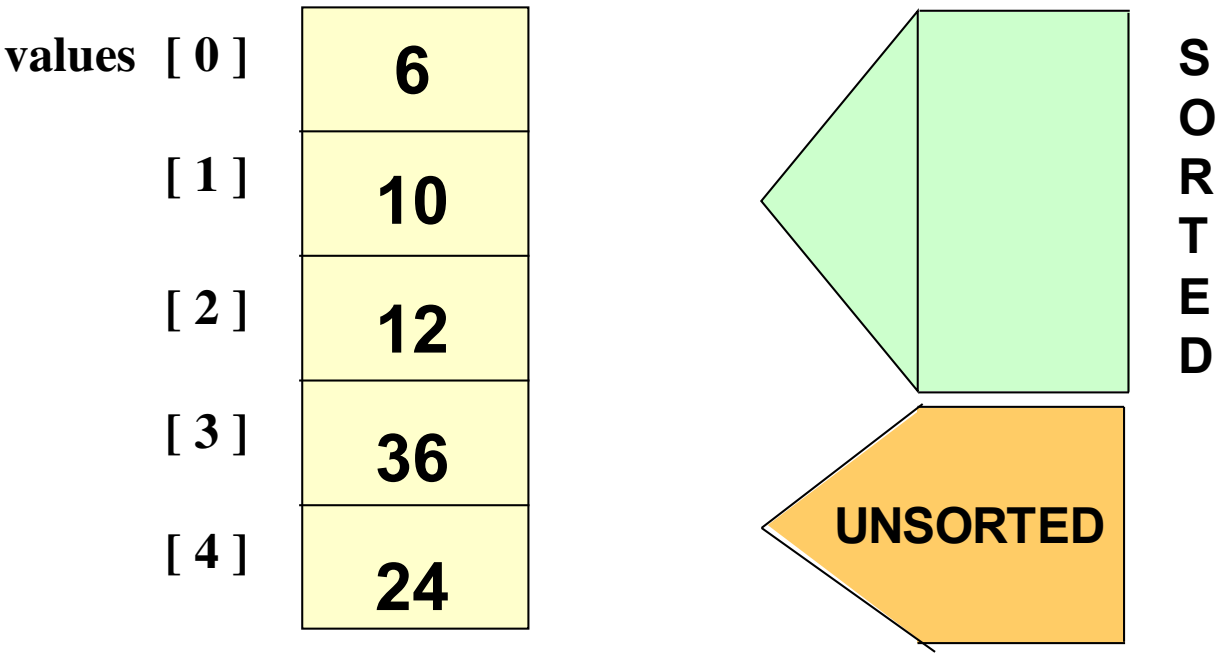
Bubble Sort: Pass Three



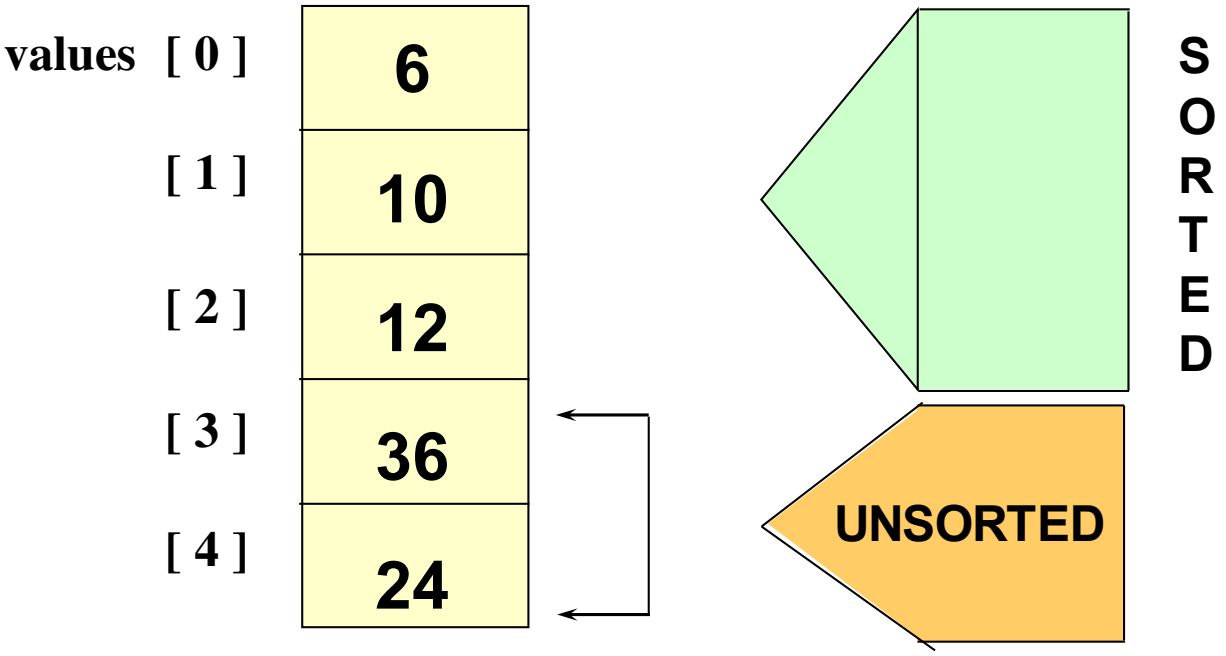
Bubble Sort: Pass Three



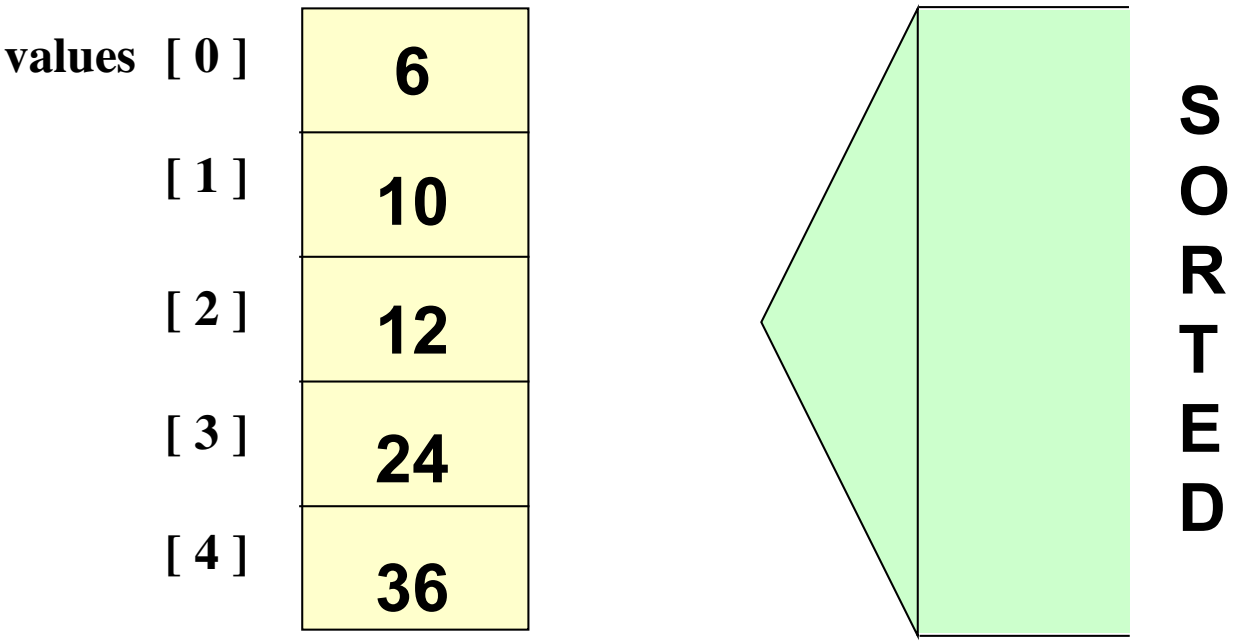
Bubble Sort: End Pass Three



Bubble Sort: Pass Four



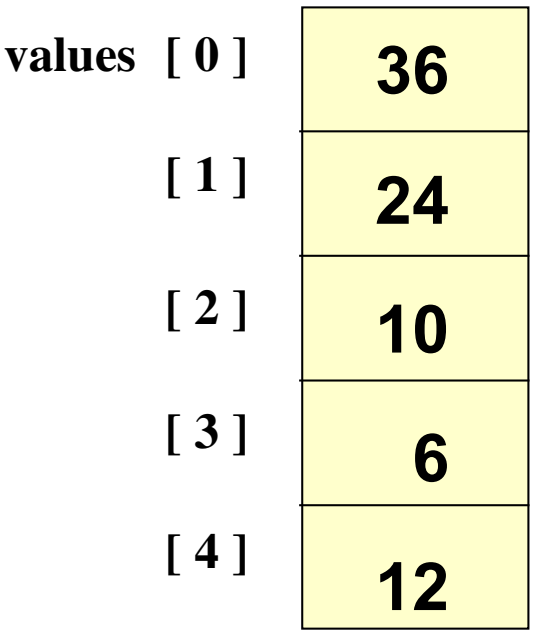
Bubble Sort: End Pass Four



A sample Python code for Bubble Sort

```
def bubble(l):
    n = len(l)
    for i in range(0,n-1):
        for j in range(0,n-1):
            if l[j][2]<l[j+1][2]:
                l[j],l[j+1] = l[j+1],l[j]
    return l
n = int(input())
l = []
for i in range(0,n):
    name = input()
    addr = input()
    score = float(input())
    l.append((name,addr,score))
l = bubble(l)
print(l)
```

Selection Sort



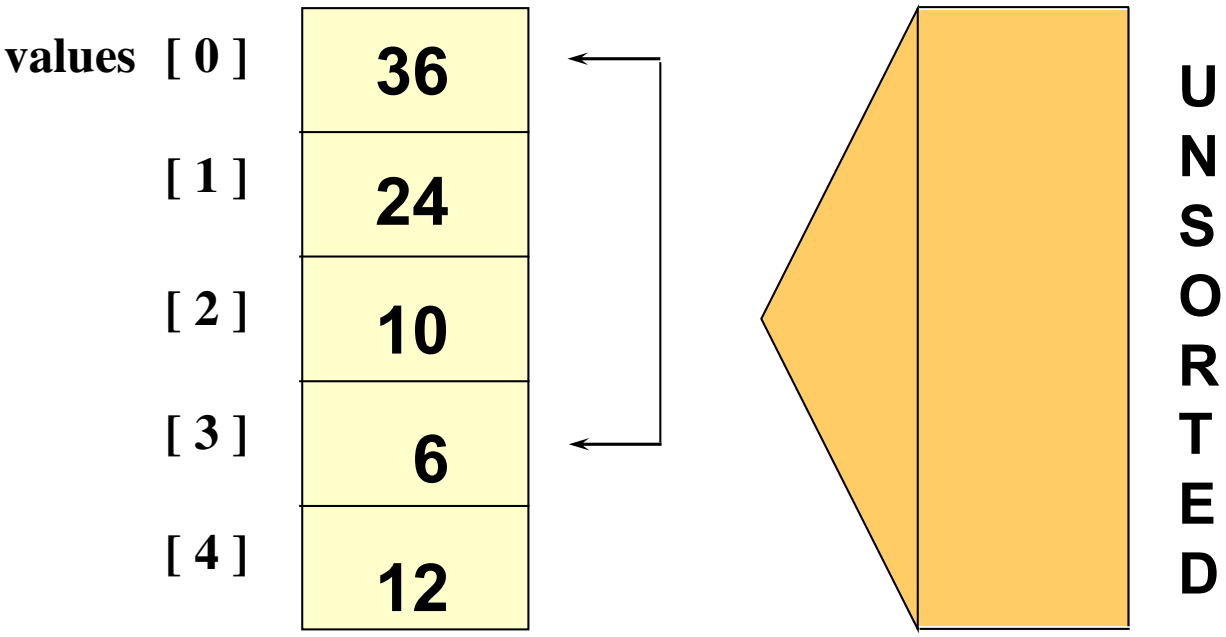
Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.

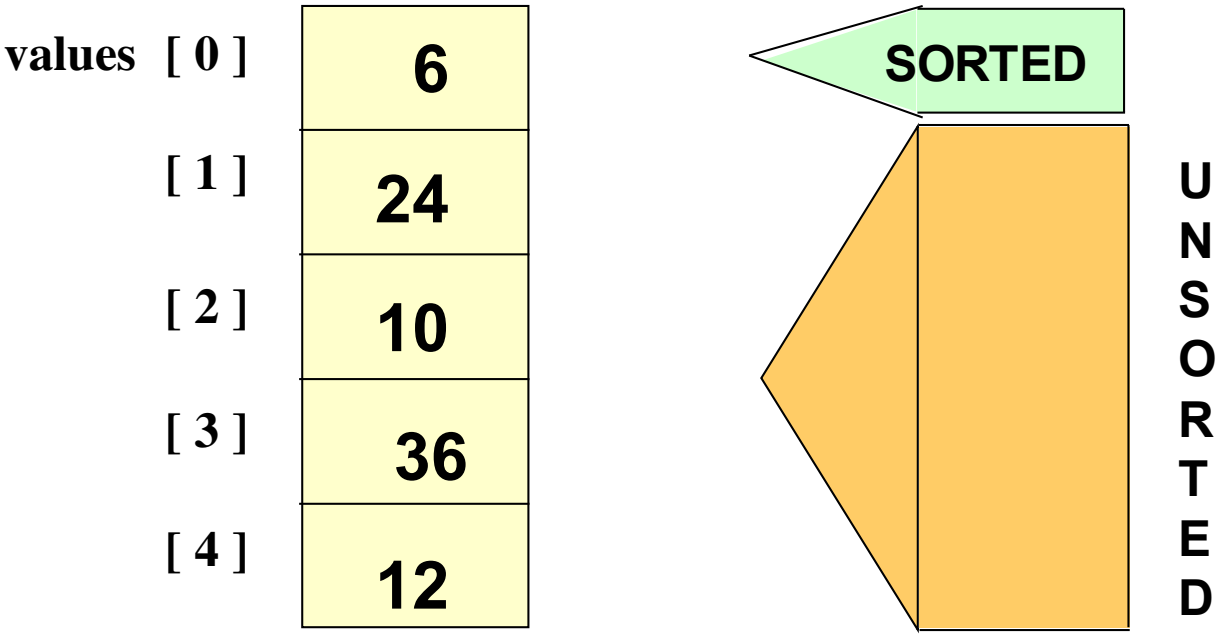
Selection Sort Pseudo Code

```
def selectionSort(lyst):
    i = 0
    while i < len(lyst) - 1:          # Do n - 1 searches
        minIndex = i                  # for the smallest
        j = i + 1
        while j < len(lyst):          # Start a search
            if lyst[j] < lyst[minIndex]:
                minIndex = j
            j += 1
        if minIndex != i:             # Exchange if needed
            swap(lyst, minIndex, i)
        i += 1
```

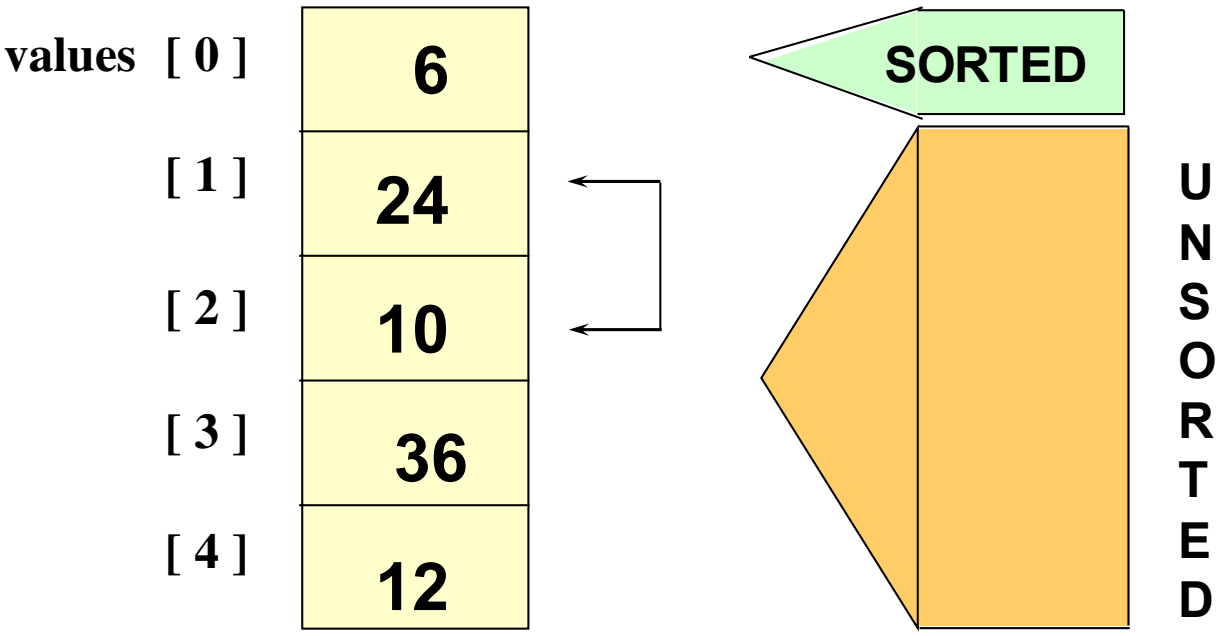
Selection Sort: Pass One



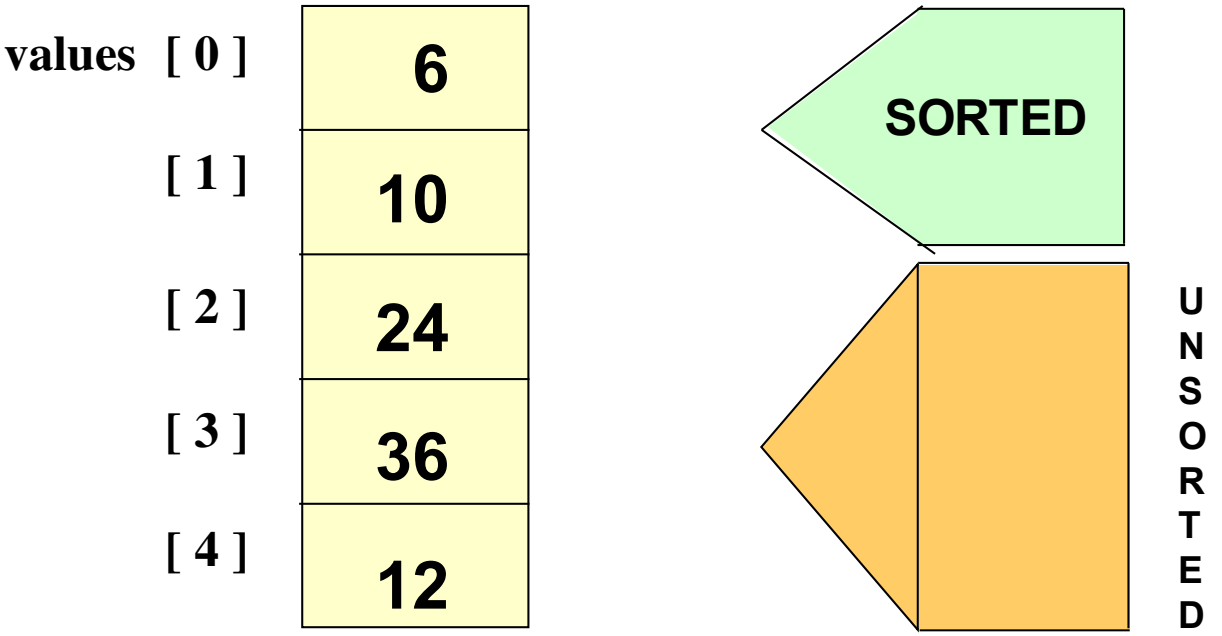
Selection Sort: End Pass One



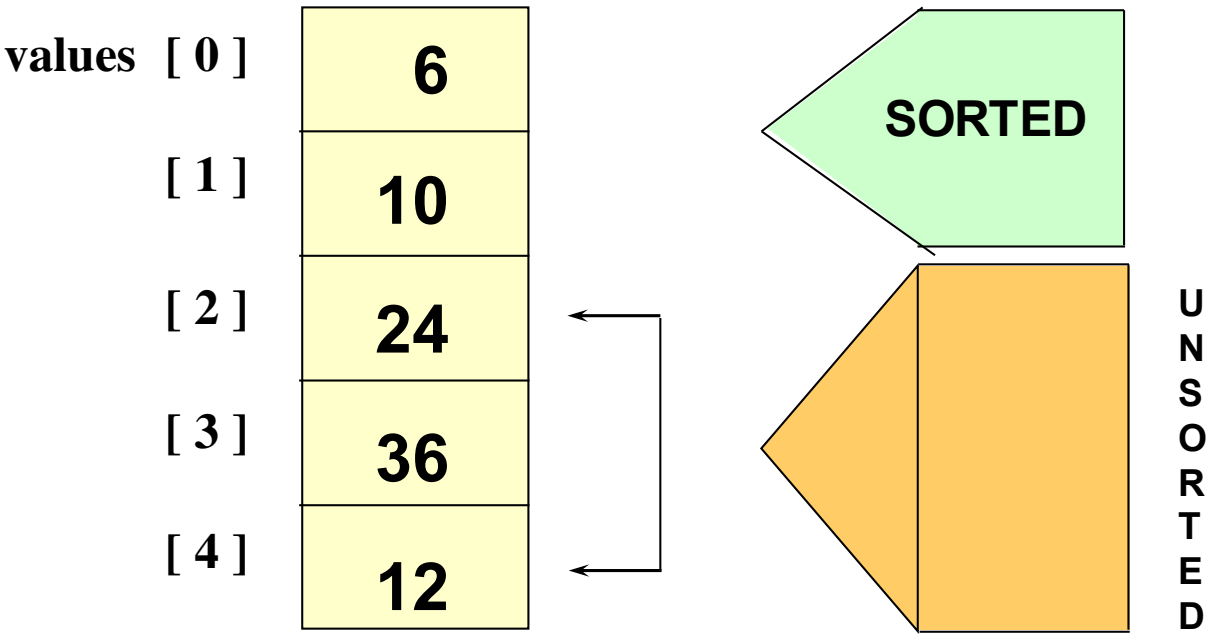
Selection Sort: Pass Two



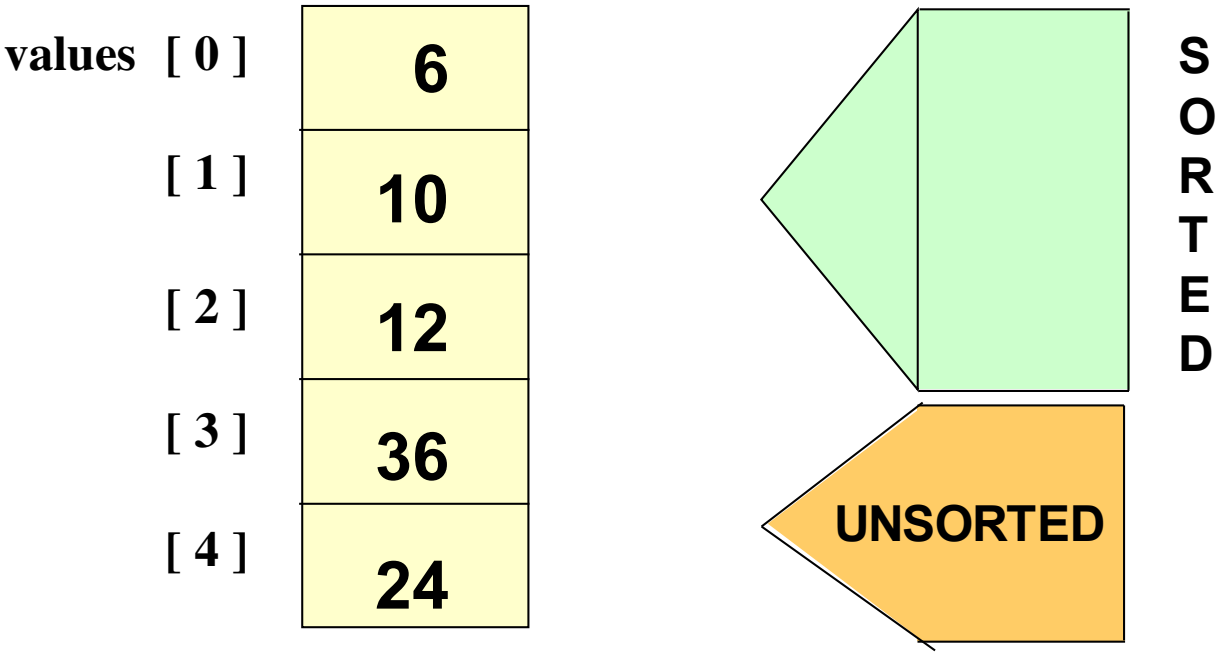
Selection Sort: End Pass Two



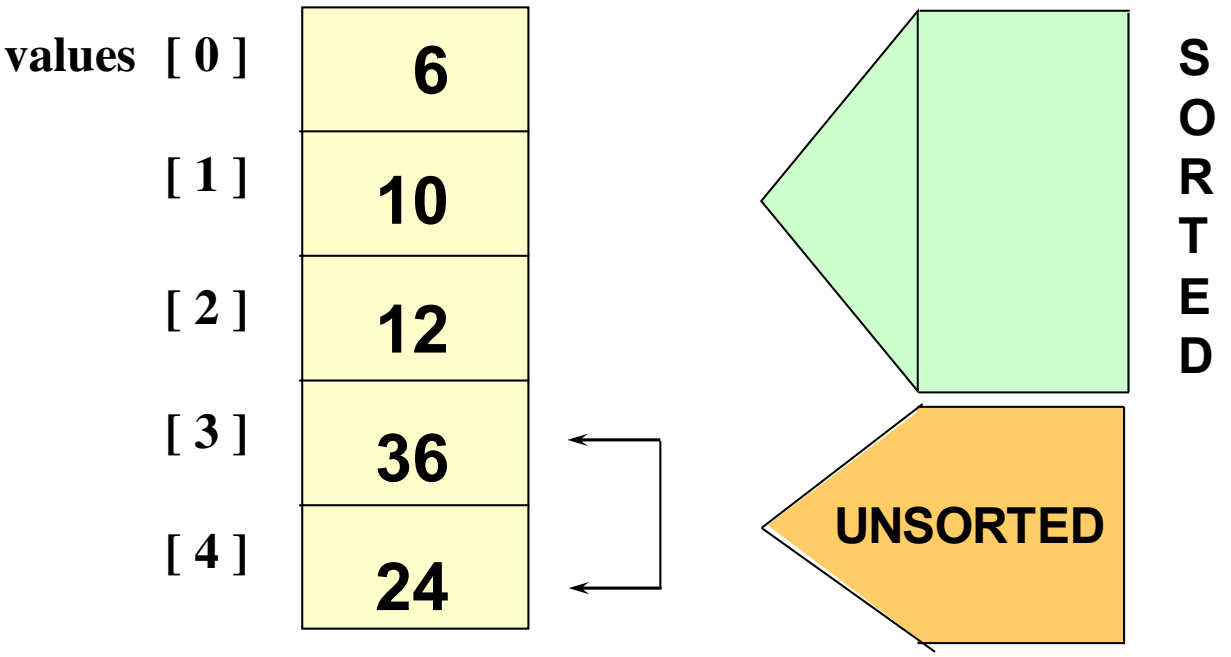
Selection Sort: Pass Three



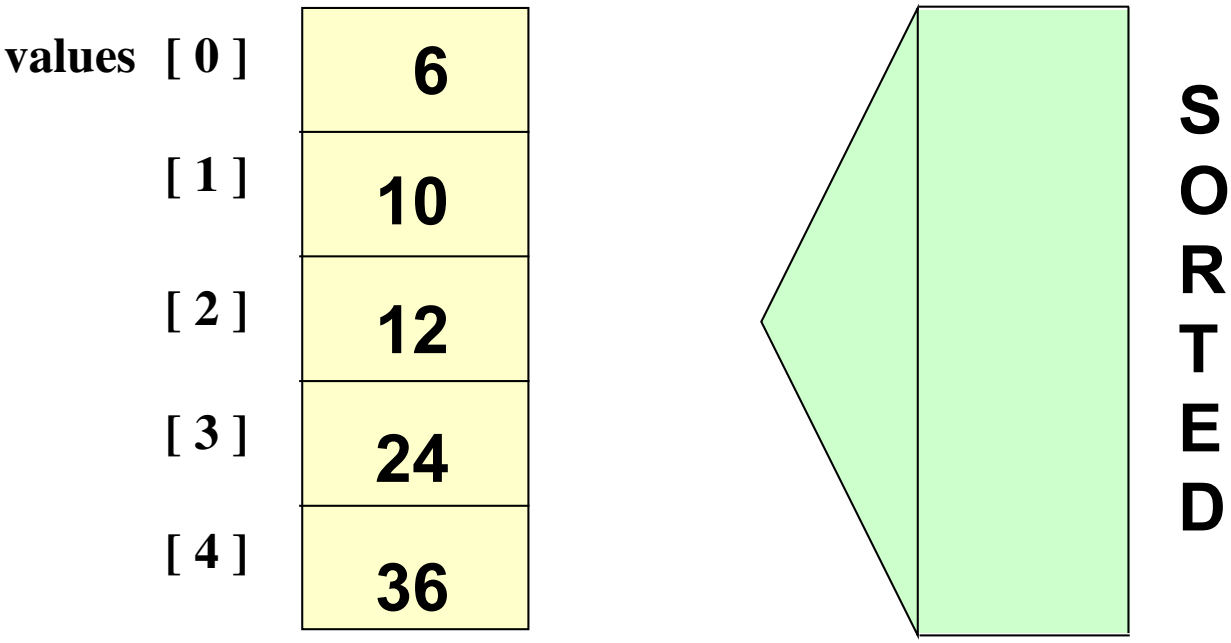
Selection Sort: End Pass Three



Selection Sort: Pass Four



Selection Sort: End Pass Four



Selection Sort: How many comparisons?

values	[0]	6
	[1]	10
	[2]	12
	[3]	24
	[4]	36

4 comparisons for **values[0]**

3 comparisons for **values[1]**

2 comparisons for **values[2]**

1 comparison for **values[3]**

= 4 + 3 + 2 + 1

A sample Python code for Selection Sort

```
def minIndex(i,l):
    minI = i
    for j in range(i+1,len(l)):
        if l[minI]>l[j]:
            minI = j
    return minI
def selection(l):
    n = len(l)
    for i in range(0,n):
        j = minIndex(i,l)
        if i!=j:
            l[i],l[j] = l[j],l[i]
    return l
```

A sample Python code for Selection Sort

```
n = int(input())
l = []
for i in range(0,n):
    e = int(input())
    l.append(e)
l = selection(l)
print(l)
```

Insertion Sort in Real Life

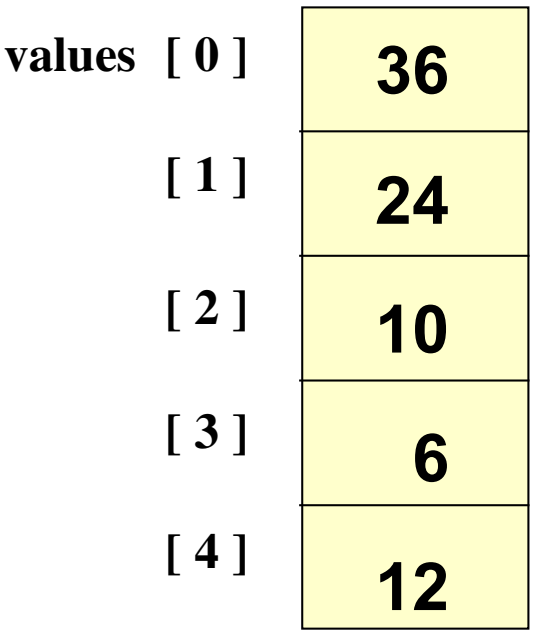
Have you ever seen a teacher alphabetizing a couple dozen papers?

She takes a paper from an unsorted collection and place into a sorted collection in order

While playing cards, to sort the cards in the hand

We extract a card, shift the remaining cards and insert the extracted card in correct place

Insertion Sort



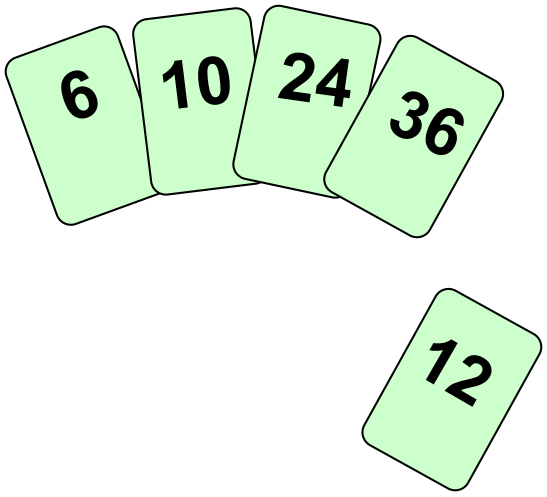
Insert, one by one, each unsorted array element into its proper place.

On each pass, this causes the number of already sorted elements to increase by one.

Insertion Sort Pseudo Code

```
def insertionSort(lyst):
    i = 1
    while i < len(lyst):
        itemToInsert = lyst[i]
        j = i - 1
        while j >= 0:
            if itemToInsert < lyst[j]:
                lyst[j + 1] = lyst[j]
                j -= 1
            else:
                break
        lyst[j + 1] = itemToInsert
        i += 1
```

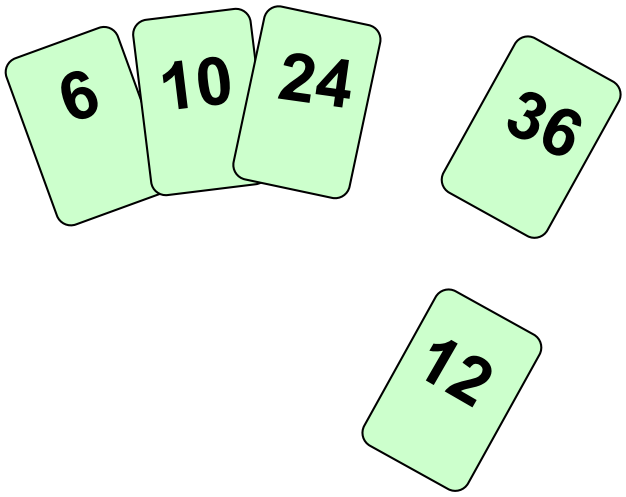
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

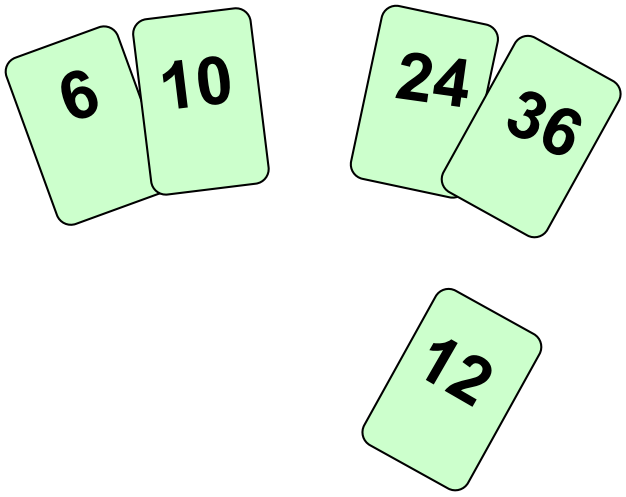
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

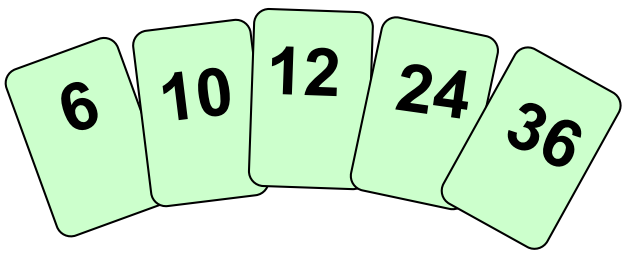
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

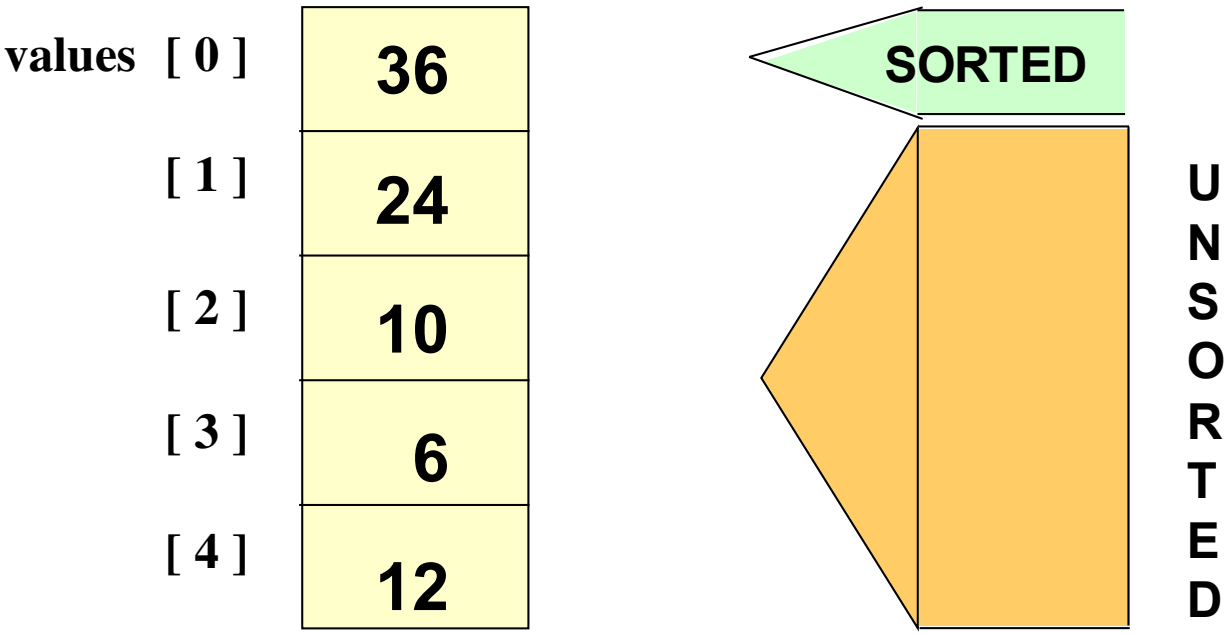
Insertion Sort



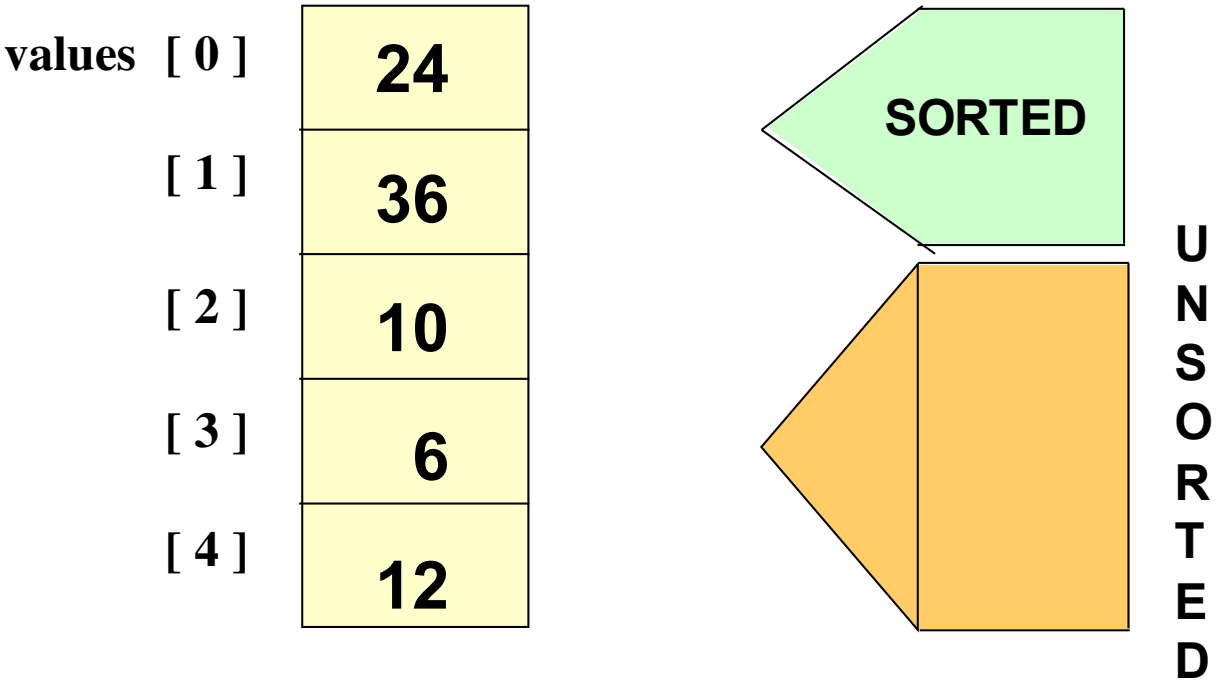
Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

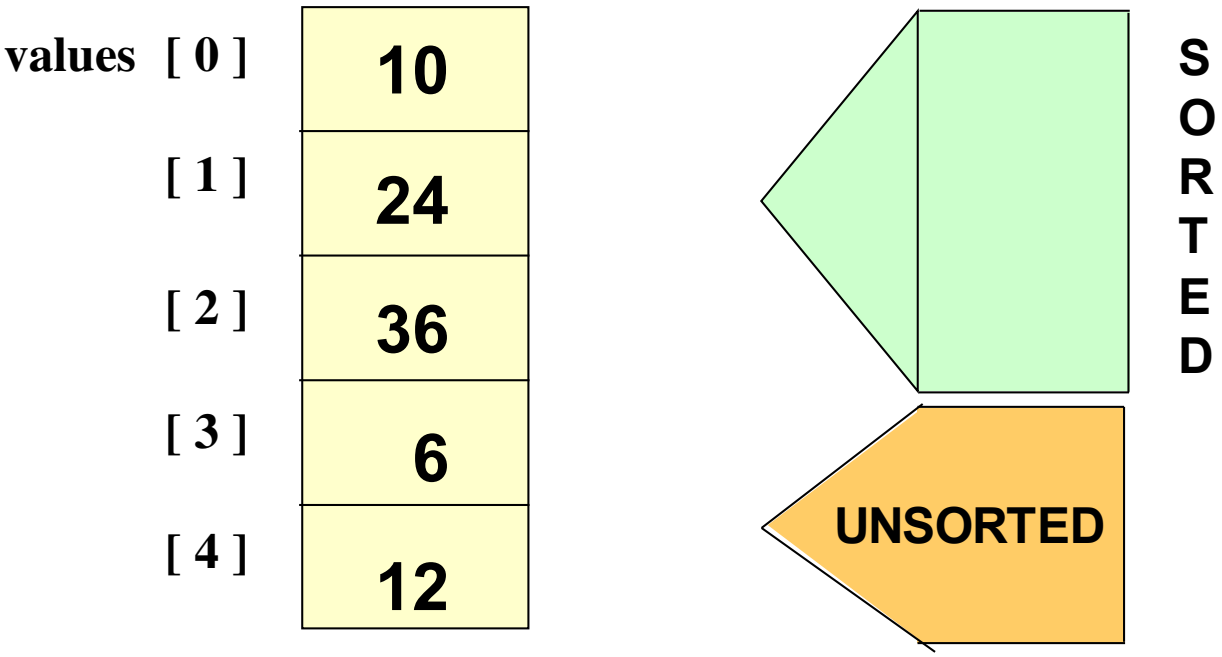
Insertion Sort: Pass One



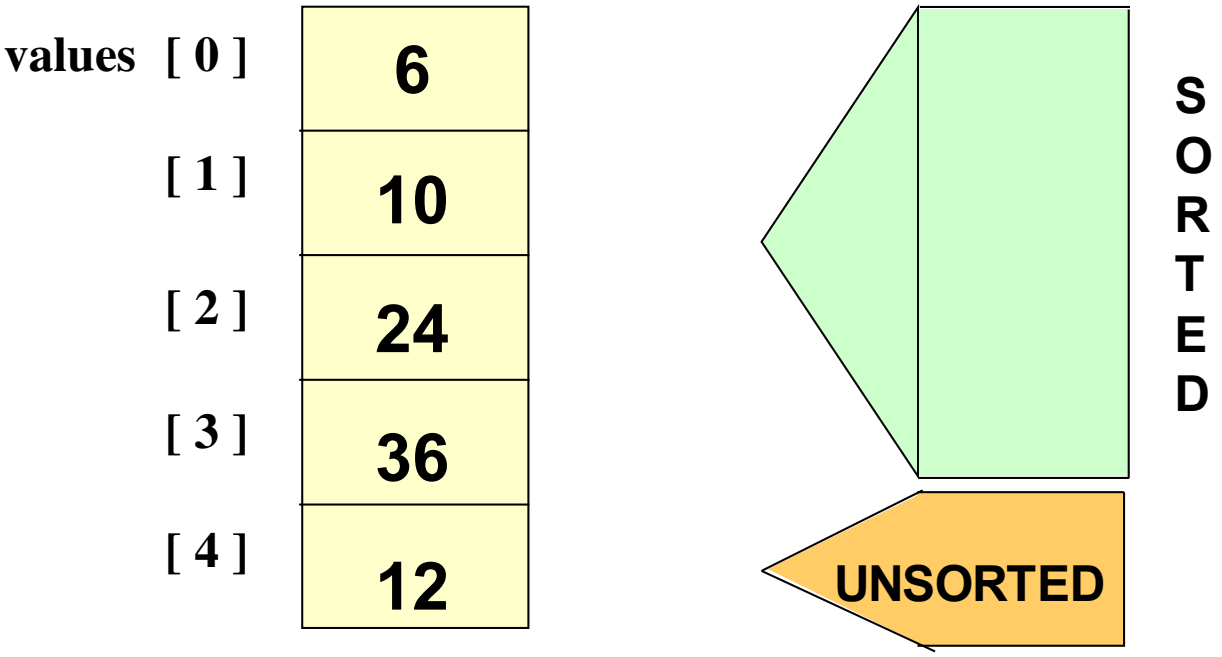
Insertion Sort: Pass Two



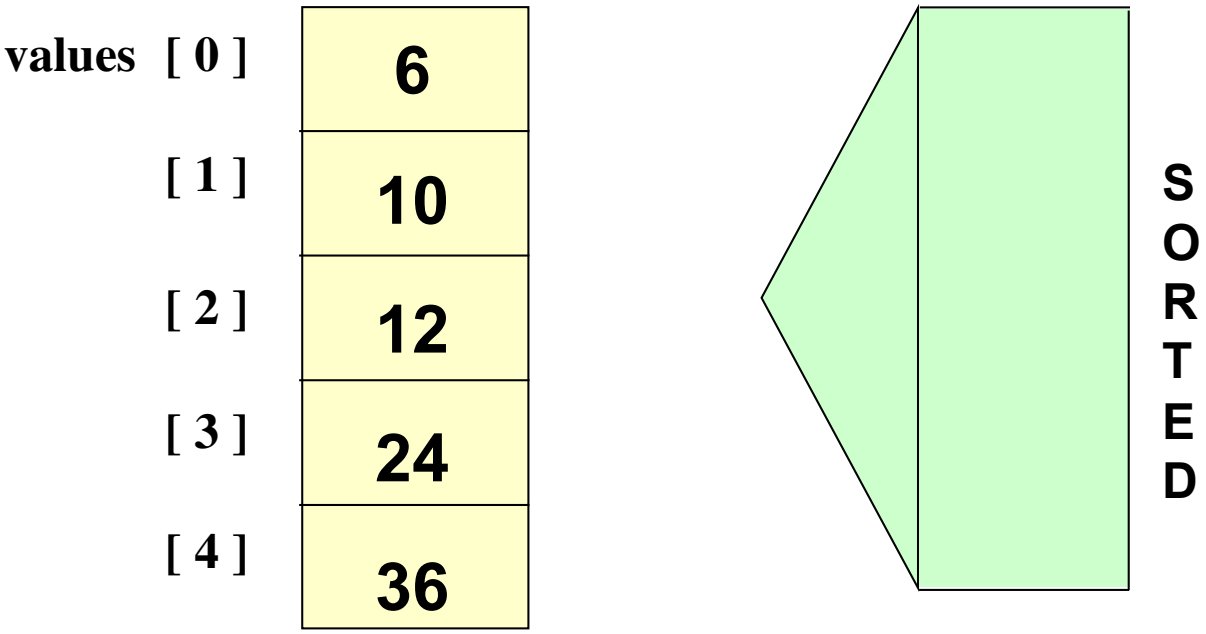
Insertion Sort: Pass Three



Insertion Sort: Pass Four



Insertion Sort: Pass Five



A sample Python code for Insertion Sort

```
def insertion_sort(items):
    """ Implementation of insertion sort """
    for i in range(1, len(items)):
        j = i
        while j > 0 and items[j] > items[j-1]:
            items[j], items[j-1] = items[j-1], items[j]
            j -= 1
```

Exercise 1

- Given a sorted list with an unsorted number *V* in the rightmost cell, can you write some simple code to *insert V* into the array so that it remains sorted? Print the array every time a value is shifted in the array until the array is fully sorted.
- *Guideline:* You can copy the value of *V* to a variable and consider its cell "empty". Since this leaves an extra cell empty on the right, you can shift everything over until *V* can be inserted.

Exercise 2

- Using the same approach as exercise 1, sort an entire unsorted array?
- *Guideline:* You already can place an element into a sorted array. How can you use that code to build up a sorted array, one element at a time? Note that in the first step, when you consider an element with just the first element - that is already "sorted" since there's nothing to its left that is smaller.
- In this challenge, don't print every time you move an element. Instead, print the array after each iteration of the insertion-sort, i.e., whenever the next element is placed at its correct position.
- Since the array composed of just the first element is already "sorted", begin printing from the second element and on.

Searching
Sequential & Binary Search

Problem

When the city planners developed your neighborhood, they accidentally numbered the houses wrong. As such, the addresses of the houses on your street are in a random order. How does the postman find your house using a linear search method?

Pseudocode

```
READ street_door_numbers and door_number_searched
FOR i =0 to length(street_door_numbers )
    IF street_door_numbers [i] == door_number_searched
    THEN
        give the post in the house
        break
END FOR
```

SEARCHING

- Searching is the algorithmic process of finding a particular item in a collection of items.
- ***A search typically answers either True or False as to whether the item is present.***

TYPES

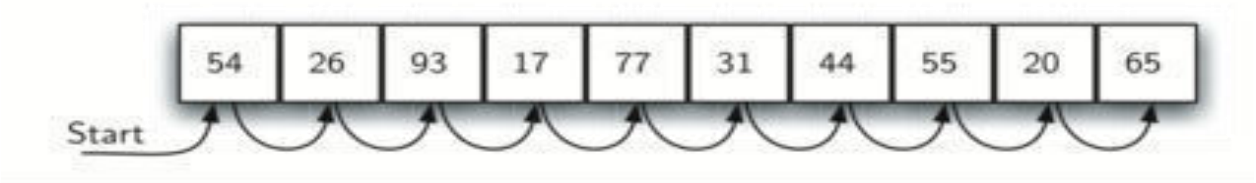
- SEQUENTIAL /LINEAR SEARCH
- BINARY SEARCH

SEQUENTIAL SEARCH

- When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship.
- Each data item is stored in a position relative to the others.
- In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence.
- This process gives rise to our first searching technique, the **sequential search**

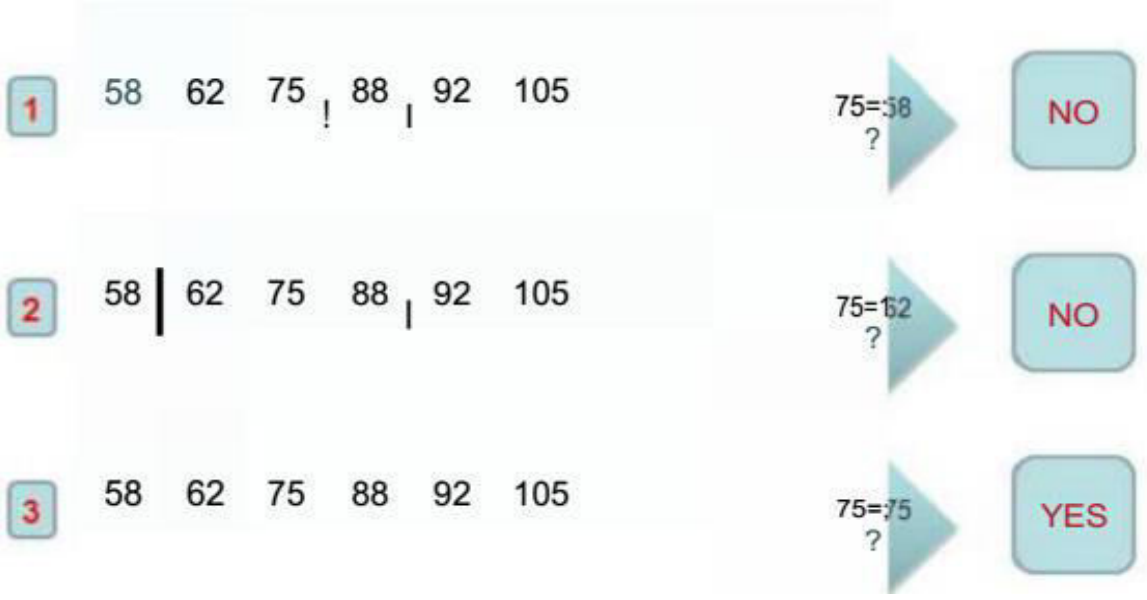
Sequential Search

Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.



EXAMPLE

List of elements 58,62,75,88,92,105
Element to be searched : 75



Linear Search Algorithm

```
procedure Linear Search (List of N elements, Search element S)
Begin
    for i=1 to N
        if (ith element of the list = S)
            return address or index of the ith element
        end if
    end for
            return Errors not found in the list
End
```

Python Implementation

- The function needs the list and the item we are looking for and returns a Boolean value as to whether it is present.

- The Boolean variable **found** is initialized to False and is assigned the value True if we discover the item in the list.

Python Implementation

```
def seqSearch(alist,item):
    found = False
    for i in range(0,len(alist)):
        if alist[i]==item:
            found = True
    return found
testlist=[0,1,2,8,13,17,19,32,42]
print(seqSearch(testlist,3))
print(seqSearch(testlist,13))
```

Exercise-Sequential Search

- You need a picture frame, so you walk down to the local photo store to examine their collection. They have all of their frames lined up against the wall. Apply the linear search algorithm to this problem, and describe how you would find the frame you wanted. Starting at the first frame, examine each frame along the wall (without skipping any) until you find the frame you want.

BINARY SEARCH

- In the sequential search, when we compare against the first item, there are at most n−1 more items to look through if the first item is not what we are looking for.

- Instead of searching the list in sequence; binary **search** will start by examining the middle item. ***If that item is the one we are searching for, we are done.***

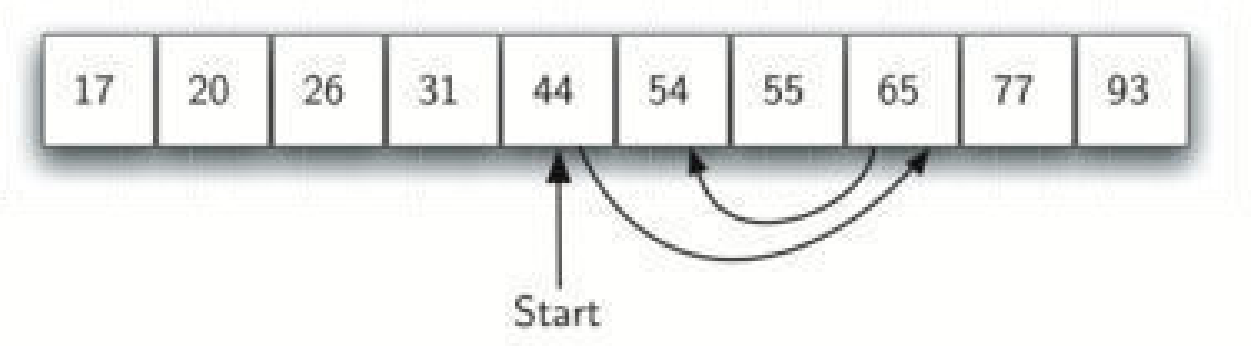
If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items.

BINARY SEARCH

Algorithm can quickly find the value 54

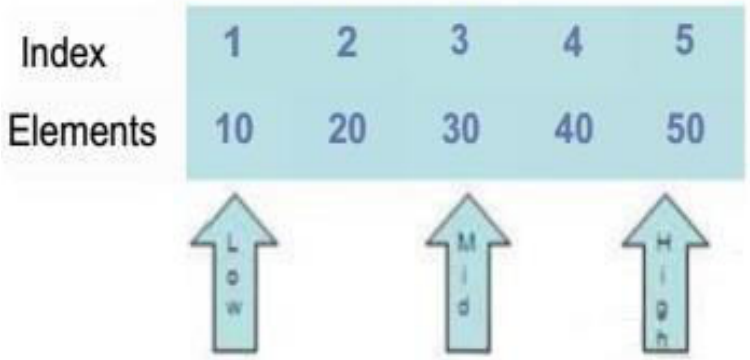
If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration.

The item, if it is in the list, must be in the upper half. We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space



Binary Search Example

Data 10.20,30,40,50 and Search element 40



Low = 1 and High =5
So Mid = (1+5)/2 = 3

Arr[Mid] = 30 < 40

Low = Mid +1 = 4

Mid = (Low + High) /2
Mid = 45 = 4

ArrfMid] = 40

The index to be returned is 4

Binary Search Algorithm

```
procedure BinarySearch(ArrayArrof N ele
Search element S)
Begin
        Low <- 1
        High <r N
        while (High>=Low)
            mid = (High + Low)/2
            if (Arr[mid] = S) then  return mid
            else
                if(Arr[mid] < S)then  low = mid +1
                else     high = mid -1
                end if
            end if
        end while
    End
```

```
def binarySearch(alist,item):
    first=0
    last=len(alist)-1
    found=False
    while first<=last and not found:
        midpoint = (first+last)//2

        if alist[midpoint] == item:
            found = True
        else:
            if item <alist[midpoint]:
                last=midpoint-1
            else:
                first=midpoint+1
    return found
testlist=[0,1,2,8,13,17,19,32,42]
print(binarySearch(testlist,3))
print(binarySearch(testlist,13))
```

Exercise-Binary Search

- Given a ordered list of student rank with the name of student. Your program will read the rank from the user and display the name of the student.

- An employee number is generated in ascending order whenever a new employee joins. Your program will read the employee number and display the dept of the employee

More Exercises

- T he element being searched for is not in an array of 100 elements. What is the maximum number of comparisons needed in a sequential search to determine that the element is not there if the ele ments are: (a) completely unsorted? (b) sorted in ascending order? (c) sorted in descending order?

- Consider the following array of sorted integers: 10, 15, 25, 30, 33, 34, 46, 55, 78, 84, 96, 99 Using binary search algorithm, search for 23. Show the sequence of array el e m e nts that are compared, and for each comparison, indicate the values of low and high

FILE HANDLING

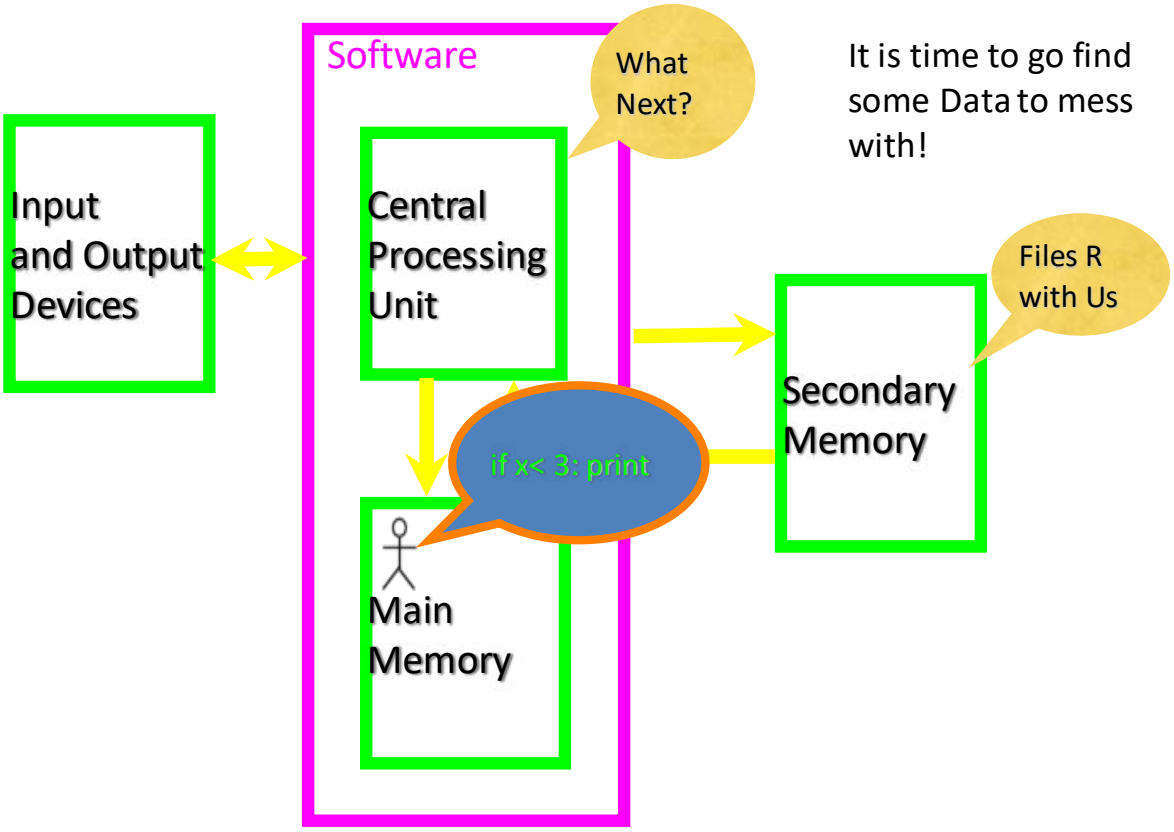
Problem

- Consider the following scenario

  -> You are asked to find the number of students who secured centum in mathematics in their examination. A total of 6 lakhs students appeared for the examinations and their results are available with us.

- The processing is typically supposed to be automatic and running on a computer. As data are most helpful when presented systematically and in fact educational to highlight their practicality.

```
Pseudocode
OPEN file
REPEAT
    READ each line of file
    SET count = 0
    PARSE the line
    IF maths_mark == 100 THEN
            COMPUTE count as count + 1
    END IF
until end of file is reached
PRINT count
```



Input / Output

- Input is any information provided to the program
  - Keyboard input
  - Mouse input
  - File input
  - Sensor input (microphone, camera, photo cell, etc.)
- Output is any information (or effect) that a program produces:
  - sounds, lights, pictures, text, motion, etc.
  - on a screen, in a file, on a disk or tape, etc.

Kinds of Input and Output

- What kinds of Input and Output have we knew
  - print (to the console)
  - input (from the keyboard)
- So far…
  - Input: keyboard input only
  - Output: graphical and text output transmitted to the computer screen

- Any other means of I/O?

Necessity of Files

- Small businesses accumulate various types of data, such as financial information related to revenues and expenses and data about employees, customers and vendors.
- Traditional file organization describes storing data in paper files, within folders and filing cabinets.
- Electronic file organization is a common alternative to paper filing; each system has its benefits and drawbacks.

What is a file?

- A file is some information or data which stays??? in the computer storage devices. We already know about different kinds of file, like music files, video files, text files, etc.

Introduction to file handling

- Files – Huge volume or Collection of data
- Types – Binary, Raw, Text, etc.
- Open any file before read/write.

Modes of opening a File:

- r – Reading only
- r+ - Both Read/Write
- w – Writing only
- w+ - Both Read/Write
- A – Appending
- a+ - Appending/Reading

File Object Attributes

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
```

Functions for file Handling

- The read functions contains different methods:
  - read() #return one big string
  - readline() #return one line at a time
  - readlines() #returns a list of lines

File Handling functions contd..

- This method writes a sequence of strings to the file.
  - write () #Used to write a fixed sequence of characters to a file
  - writelines() #writelines can write a list of strings.

Functions contd…

- The append function is used to append to the file instead of overwriting it.
- To append to an existing file, simply open the file in append mode ("a"):
- When you’re done with a file, use close() to close it and free up any system resources taken up by the open file.

To open a text file, use:

```
fh = open("hello.txt", "r")
```

To read a text file, use:

```
fh = open("hello.txt","r")
print (fh.read() )
```

To read one line at a time, use:

```
fh = open("hello".txt", "r")
print (fh.readline() )
```

To read a list of lines use:

```
fh = open("hello.txt.", "r")
print (fh.readlines() )
```

To write to a file, use:

```
fh = open("hello.txt","w")
fh.write("Hello World")
fh.close()
```

To write to a file, use:

```
fh = open("hello.txt", "w")
lines_of_text = ["a line of text", "another line of text", "a
third line"]
fh.writelines(lines_of_text)
fh.close()
```

To append to file, use:

```
fh = open("Hello.txt", "a")
write("Hello World again")
fh.close ()
```

Playing Randomly in files

- fileObject.tell() -> current position within a file
- fileObject.seek(offset [,from]) -> Move to new file position.
  - Argument offset is a byte count.
  - Optional argument whence defaults to 0 (offset from start of file, offset should be >= 0); other values are 1 (move relative to current position, positive or negative), and 2 (move relative to end of file, usually negative, although many platforms allow seeking beyond the end of a file)

Example for random seeking

```
f = open('workfile.txt', 'rb+')
f.write(b'0123456789abcdef')
f.seek(5)      # Go to the 6th byte in the file
print(f.read(1))
f.seek(-3, 2)  # Go to the 3rd byte before the end
print(f.read(1))
f.close()
```

Tips and Tricks makes it Easier…

- Number of characters in a file is same as the length of its contents.

```
def charcount(filename):
    return len(open(filename).read())
```

- Number of words in a file can be found by splitting the contents of the file.

```
def wordcount(filename):
    return len(open(filename).read().split())
```

- Number of lines in a file can be found from readlines method.

```
def linecount(filename):
    return len(open(filename).readlines())
```