

# Python Programming-An Introduction

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Introduction
- 2 IDLE - An Interpreter for Python
- 3 Python Strings
- 4 Relational Operators
- 5 Logical Operators
- 6 Bitwise Operators
- 7 Variables and Assignment Statements
- 8 Keywords
- 9 Script Mode
- 10 Summary

# Introduction

- Python is a general purpose, high level and interpreted programming language.
- Python is an interactive programming language.
- Simple syntax, Easy to read and write
- Python was developed by **Guido Van Rossum** in 1991 at the National Research Institute for Mathematics and Computer Science in the Netherlands.

# Introduction (Cont.)

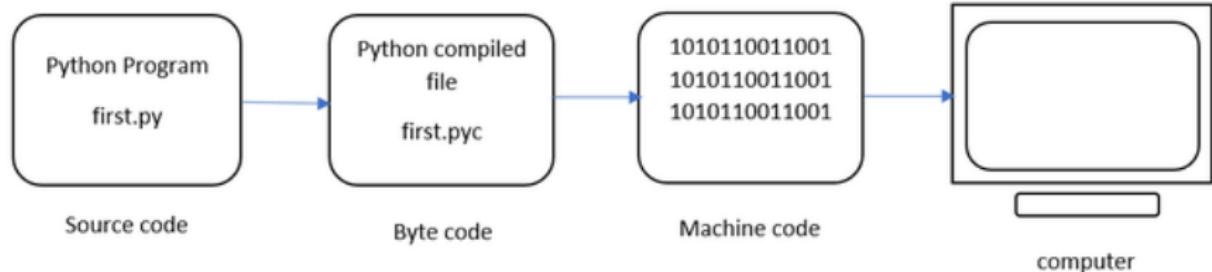


Figure 1: Steps of execution of a python program

# Execution of a Python Program

- Normally, when we compile a python program, we cannot see .pyc file produced by python compiler and the machine code generated by Python Virtual Machine (PVM). This is done internally in the memory and the output is finally visible. For example, if our python program name is first.py, we can use python compiler to compile it as:

**python first.py**

- In the preceding statement, *python* is the command for calling the python compiler. The compiler should convert the first.py file into its byte code equivalent file first.pyc. Instead of doing this, the compiler directly displays the output or result. To separately create .pyc file from the source code, we can use the following command:

**python -m py\_compile first.py**

- In order to interpret the .pyc file using PVM, the python compiler can be called using the following command:

**python first.cpython-38.pyc**

# IDLE - An Interpreter for Python

- IDLE stands for Integrated Development and Learning Environment.
- Python IDLE comprises **Python Shell** (An Interactive Interpreter) and **Python Editor** (Allows us to work in Script Mode).
- While using Python shell, we just need to type Python code at the `>>>` prompt and press Enter key. For Example:  
`>>> print('Hello World')`  
Hello World
- Python shell may also be used as a calculator. For Example:

```
>>> 18+5  
23  
>>> 27//5  
5  
>>> 27.0//5  
5.0  
>>> 27%5  
2
```

# IDLE - An Interpreter for Python

- We evaluate the foregoing expressions in Python shell. For Example:

```
>>> 3**2  
9  
>>> 6/3/2      #(6/3)/2  
1.0  
>>> 2**3**2      #2** (3**2)  
512
```

- Left associative operators: +, -, \*, /, //, %
- Right associative operators: \*\*
- Precedence of Arithmetic operators are:

( ) (parentheses)	decreasing order
** (exponentiation)	
- (negation)	
/ (division) // (integer division) * (multiplication) % (modulus)	
+ (addition) - (subtraction)	



# IDLE - An Interpreter for Python

- While the parentheses have the highest precedence, addition and subtraction are at the lowest level.
- Python complains when it encounters a wrongly formed expression. For Example

```
>>> 7+3(4+5)
```

Traceback (most recent call last):

```
File "<pyshell#17>", line 1, in <module>
```

```
7+3(4+5)
```

TypeError: 'int' object is not callable

- Similarly, Division by zero is a meaningless operation. For Example

```
>>> 7/0
```

Traceback (most recent call last):

```
File "<pyshell#18>", line 1, in <module>
```

```
7/0
```

ZeroDivisionError: division by zero

# Python Strings

- A string is a sequence of characters.
- We can enclose a sequence of characters between single, double, or triple quotes to specify.
- A string enclosed in single quotes may include double quotes marks and vice versa.
- A string enclosed in triple quotes (also known as docstring, i.e. documentation string) may include both single and double quote marks and may extend over several lines. For Example

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

# Python Strings

- Escape sequence \ n marks a new line.
- Use of + as the concatenation operator.
- The operator \* (multiplication) is used to repeat a string a specified number of times.
- For Example:

```
>>> 'hello' + '!!'
```

```
'hello !!'
```

```
>>> 'how' + 'are' + 'you?'
```

```
'how are you?'
```

```
>>> 'hello' * 5
```

```
'hellohellohellohellohello'
```

# Relational Operators

- Relational Operators are used for comparing two expressions and yield True or False.
- Arithmetic operators have higher precedence than the relational operators.

==	(equal to)
<	(less than)
>	(greater than)
< =	(less than or equal to)
> =	(greater than or equal to)
! =	(not equal to)

- The relational operators are:
- A relational operator applied on expressions as:  
expression <*comparisonoperator*> expression
- For Example:  
`>>> 23 != 23  
False`
- ASCII values of characters are used for string comparison.
- Python 3 does not allow string values to be compared with numeric values.

# Logical Operators

- Logical operators not, and, and or are applied to logical operands True and False, also called Boolean values, and yield either true or False.
- The operator not is unary, whereas other two are binary operator, those are as described below:

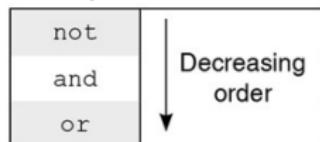
not	
True	False
False	True

and	True	False
True	True	False
False	False	False

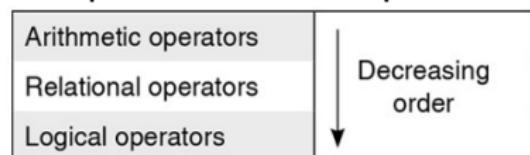
or	True	False
True	True	True
False	True	False

# Logical Operators

- The precedence of logical operators are:



- The precedence of operators are:



- For Example: The expression  $(10 < 5)$  and  $((5/0) < 10)$  yields False, the expression  $(10 > 5)$  and  $((5/0) < 10)$  yields an error.

# Bitwise Operators

- These operators operate on integers interpreted as strings of binary digits 0 and 1, also called bits, as:

Bitwise operator	Description	Example
Bitwise AND $x \& y$	A bit in $x \& y$ is 1 if the corresponding bit in each of $x$ and $y$ is 1, and 0 otherwise.	$10 \& 6$ yields 2: 10: 00001010 6: 00000110 ----- 2: 00000010
Bitwise OR $x   y$	A bit in $x   y$ is 1 if at least one of the corresponding bits in $x$ and $y$ is 1, and 0 otherwise.	$10   6$ yields 14: 10: 00001010 6: 00000110 ----- 14: 00001110
Bitwise Complement $\sim x$	A bit in $\sim x$ is 1 if and only if the corresponding bit in $x$ is 0. The result obtained from this operation is $x - 1$ .	$\sim 11$ yields -12: 11: 00001011 ----- -11: 11110100 Note that 11110100 is the two's complement of 12. 12: 00001100
Bitwise Exclusive OR $x ^ y$	A bit in $x ^ y$ is 1 if exactly one of the corresponding bits in $x$ and $y$ is 1, and 0 otherwise.	$10 ^ 6$ yields 12: 10: 00001010 6: 00000110 ----- 12: 00001100
Left Shift $x << y$	Bits in the binary representation of $x$ are shifted left by $y$ places. Rightmost $y$ bits of the result are filled with zeros. The result obtained from this operation is $x \times 2^y$ .	$5 << 2$ yields 20: 5: 00000101 ----- 20: 00010100
Right Shift $x >> y$	Bits in the binary representation of $x$ are shifted right by $y$ places. Leftmost $y$ bits of the result are filled with sign bit. The result obtained from this operation is $x / 2^y$ .	$5 >> 2$ will yield 1 as the result as shown below: 5: 00000101 ----- 1: 00000001

# Bitwise Operators

- The precedence of operators are as:

( )	
**	
+x, -x, ~x	
/, //, *, %	
+, -	
<<, >>	
&	
^	
==, <, >, <=, >=, !=	
not	
and	
or	

Decreasing  
order



# Variables and Assignment Statements

- Variables provide a means to name values so that they can be used and manipulated later on. For Example:

```
>>> english = 57
```

When this statement is executed, Python associates the variable `english` with value 57.

- In Python style (often called Pythonic style or Pythonic way), variables are called names, and an assignment is called an association or a building.

```
>>> english
```

```
57
```

```
>>> maths = 64
```

```
>>> total = english + maths
```

```
121
```

# Variables and Assignment Statements (Contd..)

- Syntax for assignment statement:  
variable = expression
- Rules for naming variables:
  1. A variable must begin with a letter or \_ (Underscore character).
  2. A variable may contain any number of letters, digits, or underscore characters. No other character apart from these is allowed.
- Always use meaningful variables names.
- Follow a consistent style in choosing variables.
- Examples of not valid variables as:

total_no.	#use of dot (.)
1st_number	#begins with a digit
AmountIn\$	#use of dollar symbol (\$)
Total Amount	#Presence of blank between two words

## Variables and Assignment Statements (Contd..)

- Python is case-sensitive. For example, age and Age are not same, different.
- More than one variable may refer to the same object.

```
>>> a = 5
```

```
>>> b = a
```

```
>>> b
```

```
>>> a = 7
```

```
>>> a
```

- The shorthand notation works for all binary mathematical operators.  $a = a \langle operator \rangle b$  is equivalent to  $a \langle operator \rangle = b$ . For example:  $a = a + b$  is equivalent to  $a += b$ .
  - Multiple assignments in a single statement as:
- ```
>>> a, b, c = 'how', 'are', 'you?'
```
- Assigning same value to multiple variables in a single statement as:
- ```
>>> a = b = 0
```

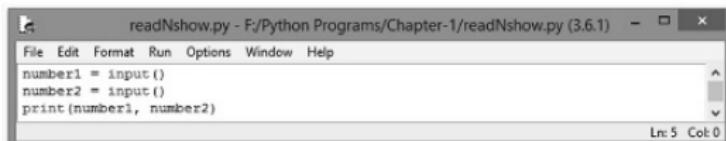
# Keywords

- Keywords are the reserved words that are already defined by the Python for specific uses.
- Keywords cannot be used for any other purposes.
- Keywords cannot be used for naming objects.
- List of Python keywords are as:

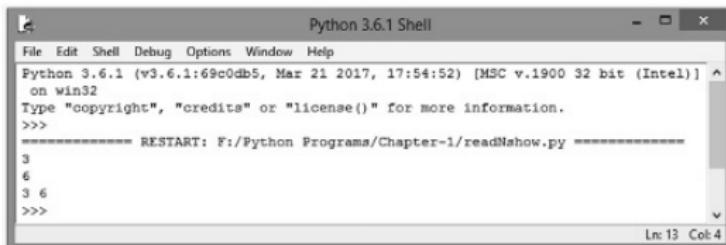
False	class	finally	is	return	None
continue	for	lambda	try	True	def
from	nonlocal	while	and	del	global
not	with	as	if	or	yield
assert	else	import	pass	break	except
in	raise	elif			

# Script Mode

- When we exit an IDLE session, and start another IDLE session, we must redo all computations. This is not convenient mode of operation for most of the computational tasks.
- Python provides another way of working called script mode.
- In script mode, instructions are written in a file.
- A script should have extension .py or .pyw.
- For Example:



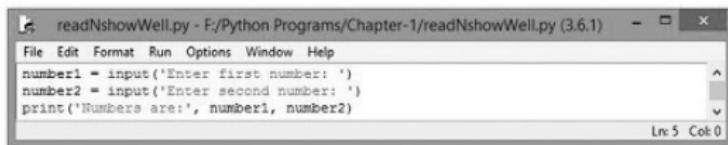
```
File Edit Format Run Options Window Help
number1 = input()
number2 = input()
print(number1, number2)
Ln: 5 Col: 0
```



```
File Edit Shell Debug Options Window Help
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] ^_ 
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: F:/Python Programs/Chapter-1/readNshow.py =====_
3
6
>>>
Ln: 13 Col: 4
```

## Script Mode (Contd..)

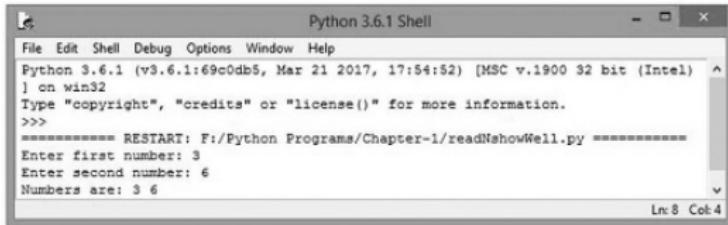
- We can modify the previous example as we may not remember that the program requires two numbers as the input.
- A good programming practice is to display to the user what data to be entered.
- For Example:



A screenshot of a Windows-style code editor window titled "readNshowWell.py - F:/Python Programs/Chapter-1/readNshowWell.py (3.6.1)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code area contains the following Python script:

```
number1 = input('Enter first number: ')
number2 = input('Enter second number: ')
print('Numbers are:', number1, number2)
```

The status bar at the bottom right shows "Ln: 5 Col: 0".



A screenshot of a Windows-style terminal window titled "Python 3.6.1 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The title bar also displays "Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32". The shell prompt shows the Python version and build information, followed by "Type "copyright", "credits" or "license()" for more information." The command line starts with ">>>". The output shows the script being run and its execution:

```
=====
RESTART: F:/Python Programs/Chapter-1/readNshowWell.py =====
Enter first number: 3
Enter second number: 6
Numbers are: 3 6
```

The status bar at the bottom right shows "Ln: 8 Col: 4".

# Summary

- Python interpreter executes one expression (command) at a time.
- A string is a sequence of characters. To specify a string, we may use single, double, or triple quotes.
- While evaluating a boolean expression involving *and* operator, the second sub-expression is evaluated only if the first sub-expression yields True.
- While evaluating an expression involving *or* operator, the second sub-expression is evaluated only if the first sub-expression yields False.
- A variable is a name that refers to a value. We may also say that a variable associates a name with data object such as number, character, string or Boolean.
- A variable name must begin with a letter or \_.
- Python is case-sensitive.
- Python programming can be done in an interactive and script mode.

# References



Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You  
Any Questions?

# Functions

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Introduction
- 2 Built-in Functions
- 3 Function Definition and Call
- 4 Importing User-Defined Module
- 5 Assert Statement

# Introduction

- Simple statements can be put together in the form of functions to do useful tasks.
- To solve a problem, divide it into simpler sub-problems.
- The solutions of the sub-programs are then integrated to form the final program.
- This approach to problem solving is called step-wise refinement method or **modular approach**.
- Functions are generally of two types. Such as:
  1. Built-in functions
  2. User-defined functions

# Built-in Functions

- Built-in Functions are predefined functions that are already available in Python.
- The function ***input*** enables us to accept an input string from the user without evaluating its value.
- The function *input* continues to read input text from the user until it encounters a newline. For Example:

```
>>> name = input('Enter a Name:')
```

Enter a Name: Alok

```
>>> name
```

'Alok'

Here, the string 'Enter a Name:' specified within the parentheses is called an argument.

- The function ***eval*** is used to evaluate the value of a string. For Example:

```
>>> eval('15')
```

15

```
>>> eval('15 + 10')
```

25

## Built-in Functions (Cont.)

- The value returned by a function may be used as an argument for another function in a nested manner. This is called **composition**.

For Example:

```
>>> n1 = eval(input('Enter a Number:'))
```

Enter a Name: 234

```
>>> n1
```

234

- We can print multiple values in a single call to print function, where expressions are separated by comma. For Example:

```
>>> print(2, 567, 234)
```

2 567 234

```
>>> name = 'Raman'
```

```
>>> print('hello',name, '2+2 =', 2+2)
```

hello Raman 2+2 = 4

## Built-in Functions (Cont.)

- We can use python escape sequences such as \n (newline), \t (tab), \a (bell), \b (backspace), \f (form feed), and \r (carriage return). For Example:

```
>>> print('hello',name, '\n 2+2 =', 2+2)
hello Raman
2+2 = 4
```

- Python function type tells us the type of a value. For Example:

```
>>> print(type(12), type(12.5), type('hello'), type(int))
<class'int'>, <class'float'>, <class'str'>, <class'type'>
```

- The round function rounds a number up to specific number of decimal places. For Example:

```
>>> print(round(89.625,2), round(89.625), round(89.625,0)
89.62 90 90.0
```

## Built-in Functions (Cont.)

- The input function considers all inputs as strings. Hence, type conversion is required. For Example:

```
>>> str(123)
```

```
'123'
```

```
>>> float(123)
```

```
123.0
```

```
>>> int(123.0)
```

```
123
```

```
>>> str(123.45)
```

```
'123.45'
```

```
>>> float('123.45')
```

```
123.45
```

```
>>> int('123.45') //String incompatible for conversion
```

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

```
int('123.45')
```

```
ValueError: invalid literal for int() with base 10: '123.45'
```

## Built-in Functions (Cont.)

- The functions *max* and *min* are used to find maximum and minimum values respectively; can also operate on string values.
- The integer and floating point values are compatible for comparison; whereas numeric values cannot be compared with string values. For Example:

```
>>> max(59,80,95.6,95.2)
```

```
95.6
```

```
>>> min(59,80,95.6,95.2)
```

```
59
```

```
>>> max('hello', 'how', 'are', 'you')
```

```
'you'
```

```
>>> min('hello', 'how', 'are', 'you', 'Sir')
```

```
'Sir'
```

- The function *pow(a,b)* computes a to the power b.

```
>>> a = pow(2,3)
```

```
>>> a
```

```
8
```

## Built-in Functions (Cont.)

- The function random is used to generate a random number in the range [0,1). Python module random contains this function and needs to be imported for using it.
- Let us agree that player A will play the first turn if the generated falls in the range [0,0.5), otherwise player B will play as:

```
if random.random() < 0.5:  
    print('Player A plays the first turn.')  
else:  
    print('Player B plays the first turn.')
```

- The math module provides some functions for mathematical computations.
- In order to obtain these functions available for use in script, we need to import the math module as:  

```
import math
```
- Name of the module, followed by the separator dot, should precede function name. The math module also defines a constant math.pi having value 3.141592653589793.

# Built-in Functions (Cont.)

- Functions of math module are:

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
floor(x)	Returns the largest integer less than or equal to x.
fabs(x)	Returns the absolute value of x.
exp(x)	Returns the value of expression $e^{**}x$ .
log(x,b)	Returns the $\log(x)$ to the base b. In the case of absence of the second argument, the logarithmic value of x to the base e is returned.
log10(x)	Returns $\log(x)$ to the base 10. This is equivalent to specifying <code>math.log(x,10)</code> .
pow(x,y)	Returns x raised to the power of y, i.e., $x^{**}y$ .
sqrt(x)	Returns the square root of x.
cos(x)	Returns the cosine of x radians.
sin(x)	Returns the sine of x radians.
tan(x)	Returns the tangent of x radians.

# Built-in Functions (Cont.)

- Functions of math module are (Cont.):

Function	Description
acos(x)	Returns the inverse cosine of x in radians.
asin(x)	Returns the inverse sine of x in radians.
atan(x)	Returns the inverse tangent of x radians.
degrees(x)	Returns a value in degree equivalent of input value x(in radians).
radians(x)	Returns a value in radian equivalent of input value x(in degrees).

- For Example:

```
>>> import math  
>>> math.ceil(3.4)  
4  
>>> math.floor(3.7)  
3
```

## Built-in Functions (Cont.)

- If we want to see the complete list the complete list of built-in functions, we can use the built-in function dir as  
`dir(__builtins__)`
- We can get the help on a function as:

```
>>> import math  
>>> help(math.cos)
```

Help on built-in function cos in module math:

`cos(...)`

`cos(x)`

Return the cosine of x (measured in radians).

# Function Definition and Call

- The syntax for function definition as:

```
def function_name (comma_separated_list_of_parameters):  
    statements
```

- For Example, We can print a triangle, a blank line and a square as:

```
def main():  
    # To print a triangle  
    print('*')  
    print(' **')  
    print(' ****')  
    print('*****')  
    # To print a blank line  
    print()  
    # To print a square  
    print('* * * *')  
    print('* * * *')  
    print('* * * *')  
    print('* * * *')
```

- Python code following colon must be intended, i.e., shifted right.

## Function Definition and Call (Cont.)

- Having developed the function main in the script picture, we would like to execute it.
- To do this, we need to invoke the function main in the following two steps.
  - In Run menu, click on the option Run Module.
  - Using Python shell, invoke (call) the function main by executing the following command:  
`>>> main()`
- We can eliminate the need to call function main explicitly from the shell, by including in the script picture, the following call to function main:  
`if '__name__' == '__main__':  
 main()`
- Python module has a built-in variable called \_\_name\_\_ containing the name of the module. When the module itself is being run as the script, this variable \_\_name\_\_ is assigned the string '\_\_main\_\_' designating it to be a \_\_main\_\_ module.

# Function Definition and Call (Cont.)

- Program to print a triangle followed by a square as:

```
def main():
    # To print a triangle
    print('*')
    print(' **')
    print(' ***')
    print(' ****')
    print(' *****')
    # To print a blank line
    print()
    # To print a square
    print('* * * *')
    print('* * * *')
    print('* * * *')
    print('* * * *')
    if '__name__' == '__main__':
        main()
```

Function main serves as a caller function for the callee or called functions triangle and square.

# Function Definition and Call (Cont.)

- A function definition has no effect unless it is invoked.

```
def triangle():
    # To print a triangle
    print(' *')
    print(' ***')
    print(' *****')
    print('*****')
def main():
    # To print a triangle
    print('Triangle')
if '__name__' == '__main__':
    main()
```

The script shows that function triangle is not being invoked.

## Function Definition and Call (Cont.)

- **return** statement returns the value of expression following the return keyword. In the absence of the return statement, default value None is returned.
- Arguments: Variables or Expressions whose values are passed to called function.
- Parameters: Variables or Expressions in function definition which receives value when the function is invoked.
- Arguments must appear in the same order as that of parameters.
- For Example, computing area of a rectangle:

```
def areaRectangle(length, breadth):
```

```
    ""
```

Objective: To compute the area of rectangle

Input Parameters: length, breadth- numerical value

Return Value: area - numerical value

```
    ""
```

```
        area = length * breadth
```

```
        return area
```

# Function Definition and Call (Cont.)

## Fruitful Functions vs void Functions

- A function that returns a value is often called a fruitful function.
- A function that does not return a value is often called a void function.

## Function Help

- Recall that function help can be used to provide a description of built-in functions.
- Function help can also be used to provide description of the function defined by user.
- Function help retrieves first multi-line comment from the function definition

# Function Definition and Call (Cont.)

## Default Parameter Values

- The function parameters may be assigned initial values also called **default values**.
- Function call uses default value, if value for a parameter is not provided.
- Non-default arguments should not follow default arguments in a function definition. For Example:

```
def areaRectangle(length, breadth=1):
```

```
    ""
```

Purpose : To compute area of rectangle

Input Parameters :

length - int

breadth (default = 1) - int

Return Value: area - int

```
    ""
```

```
area = length * breadth
```

```
return area
```

```
>>> areaRectangle(5)
```

```
5
```

```
>>> areaRectangle(5,2)
```

```
10
```

# Function Definition and Call (Cont.)

## Keyword Arguments

- Python allows us to specify arguments in an arbitrary order in a function call, by including the parameter names along with arguments.
- The syntax for keyword arguments is:  
**parameter\_name = value**
- indeed, in situations involving a large number of parameters, several of which may have default values, keyword arguments can be of great help. For Example:

```
>>> def f(a=2, b=3, c=4, d=5, e=6, f=7, g=8, h=9):  
    return a+b+c+d+e+f+g+h
```

```
>>> f(c=10, g=20)
```

62

# Importing User-Defined Module

- To access a function from a user-defined module (also known as program or script that may comprise functions, classes, and variables), we need to import it from that module.
- To ensure that the module is accessible to the script, we are currently working on, we append to the system's path, the path to the folder containing the module.
- The syntax for importing a module:  
***import name-of-the-module***

# Assert Statement

- The assert statement is used for error checking.
- For Example, When we are going for the calculation of average of marks, be sure that inputs provided by the user are in the correct range.
- For this purpose, we make use of an assert statement that has the following form:  
***assert condition***
- If the condition specified in an assert statement fails to hold, Python responds with an assertion error.

# References



Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

# Control Structures

## Lecture 3

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Introduction
- 2 if Conditional Statement
- 3 Iteration (for and while statements)

# Introduction

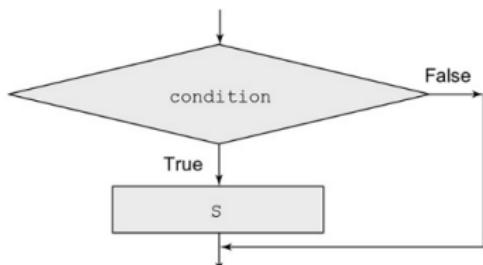
- Control structures are used for non-sequential and repetitive execution of instructions.
- Control statements (also called control structures) are used to control the flow of program execution by allowing non-sequential or repetitive execution of instructions.
- Python supports ***if, for, and while*** control structures.
- In a control statement, Python code following the colon (:) is indented.

# if Conditional Statement

- *if* statement allows non-sequential execution depending upon whether the condition is satisfied.
- The general form of *if* conditional statement is as follows:  
*if* ⟨condition⟩ :

*Sequence S of statements to be executed*

Here, condition is a Boolean expression which is evaluated at the beginning of the if statement. If condition evaluates to True, then the sequence S of statements is executed, and the control is transferred to the statement following if statement. However, if condition evaluates to False, then the sequence S of statements is ignored, and the control is immediately transferred to the statement.



# if Conditional Statement

- For Example (Marks to be updated to passMarks based on a condition):

```
def moderate(marks, passMarks):
    """
    Objective to moderate result by maximum 1 or 2 marks to
    achieve passMarks
    Input parameters:
        marks - int
        passMarks - int
    Return Value: marks - int
    """
    if marks == passMarks-1 or marks == passMarks-2:
        marks=passMarks
    return marks

def main():
    """
    Objective: To moderate marks if a student just misses
    pass marks
    Input Parameter: None
    Return Value: None
    """
    passMarks=40
    marks = input('Enter marks:')
    intmarks = int(marks)
    moderatedMarks = moderate(intmarks,passMarks)
    print('Moderated marks:',moderatedMarks)

if __name__=='__main__':
    main()
```

# if Conditional Statement

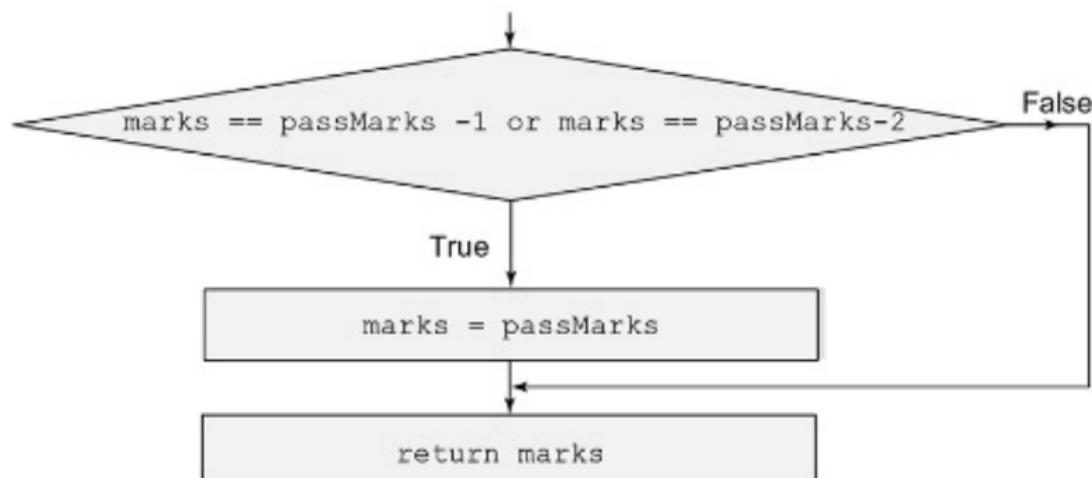
- For Example:

```
>>>
```

Enter marks: 38

Moderated marks: 40

- The flowchart of function moderate is as:



# if Conditional Statement

- The general form of if-else statement is as follows:

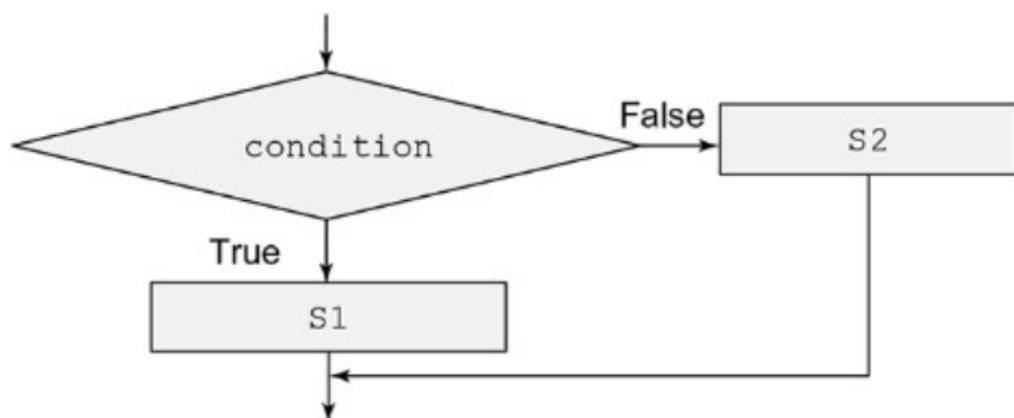
if <condition> :

    Sequence S1 of statements to be executed

else:

    Sequence S2 of statements to be executed

Here, condition is a Boolean expression. If condition evaluates to True, then the sequence S1 of statements gets executed, otherwise, the sequence S2 of statements gets executed.



# if Conditional Statement

- For Example (Program to authenticate user and allow access to system using if else statement):

```
def authenticateUser(password):
    """
    Objective to authenticate user and allow access to system
    Input parameter : password - string
    Return Value: message - string
    """
    if password == 'magic':
        message='Login Successful!!\n Welcome to system.'
    if password != 'magic':
        message=' Password mismatch!!\n'
    return message

def main():
    """
    Objective: To authenticate user
    Input Parameter: None
    Return Value: None
    """
    print(' \t LOGIN SYSTEM ')
    print('=====')
    password = input('Enter Password:')
    intmarks = int(marks)
    message = authenticateUser(password)
    print(message)

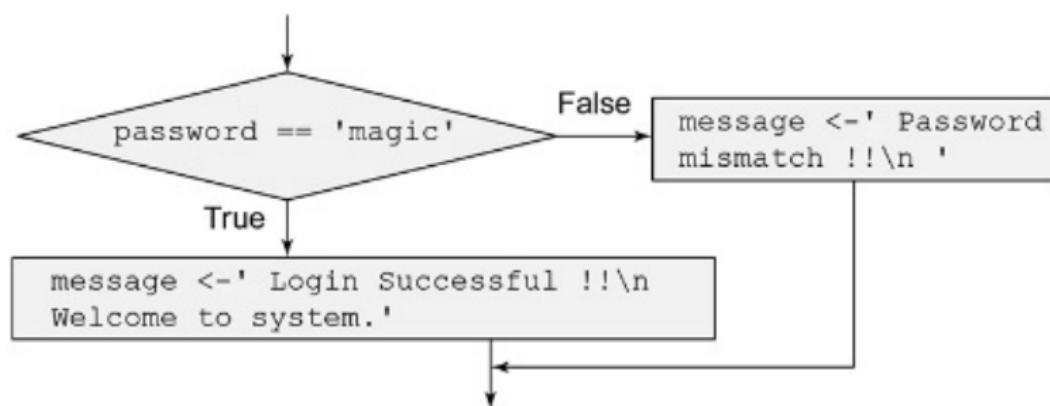
if __name__=='__main__':
    main()
```

# if Conditional Statement

- For Example:

```
>>>
if password = 'magic':
    message = 'Login Successful !! \n'
else:
    message = 'Password mismatch !! \n'
```

- The flowchart of if-else statement in the function authenticate User is as:



# if Conditional Statement

- The general form of if-elif-else statement is as follows:

```
if <condition> :  
    Sequence S1 of statements to be executed  
elif <condition2> :  
    Sequence S2 of statements to be executed  
elif <condition3> :  
    Sequence S1 of statements to be executed  
...  
else:  
    Sequence Sn of statements to be executed
```

The clauses elif and else of if control structure are optional. When a control structure is specified within another control structure, it is called nesting of control structures.

# Iteration (for and while statements)

- The process of repetitive execution of statement or a sequence of statements is called a loop.
- Execution of a sequence of statements in a loop is known as an iteration of the loop.
- The control statement *for* is used when we want to execute a sequence of statements a fixed number of times.
- The general form of for statement is as follows:  
for variable in sequence :

Sequence S of Statements

Here, variable refers to the control variable. The sequence S of statements is executed for each value in sequence.

- The function call `range(1, n+1)` generates a sequence of numbers from 1 to n. In general, **range(start, end, increment)** produces a sequence of numbers from start up to end (but not including end) in steps of increment. If third argument is not specified, increment is assumed to be 1.

# Iteration (for and while statements)

- For Example (Program to find sum of first n positive integers):

```
def summation(n):
    """
    Objective: To find sum of first n positive integers
    Input Parameter : n - numeric value
    Return Value: total - numeric value
    """
    total = 0
    for count in range(1,n+1):
        total += count
    return total

def main():
    """
    Objective: To find sum of first n positive integers based on
    user input
    Input Parameter: None
    Return Value: None
    """
    n = int(input('Enter number of terms:'))
    total = summation(n)
    print('Sum of first', n, 'positive integers:', total )

if __name__=='__main__':
    main()
```

# Iteration (for and while statements)

- A while loop is used for iteratively executing a sequence of statements again and again on the basis of a test-condition.
- The general form of while loop is as follows:

```
while <test – condition> :  
    Sequence S of statements
```

Here, test-condition is a Boolean expression which is evaluated at the beginning of the loop. If the test-condition evaluates to True, the control flows through the Sequence S of statements (i.e, the body of the loop), otherwise the control moves to the statement immediately following the while loop. On execution of the body of the loop, the test-condition is evaluated again, and the process of evaluating the test-condition and executing the body of the loop is continued until the test-condition evaluates to False.

# Iteration (for and while statements)

- For Example (Program to compute sum of numbers):

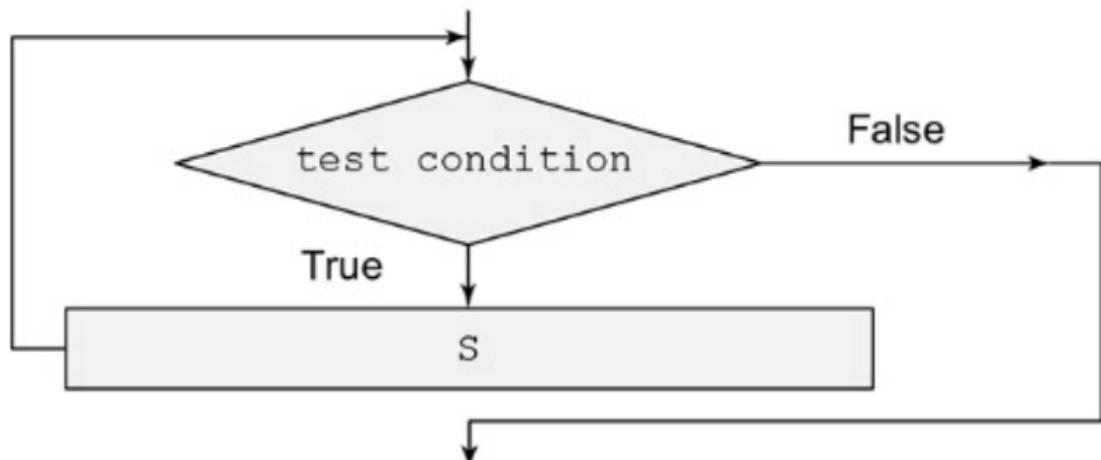
```
def main():
    """
        Objective: To compute sum of numbers entered by user until
        user provides with null string as the input
        Input Parameter: None
        Return Value: None
    """

    total=0
    number = input('Enter a number')
    while number != '':
        total += int(number)
        number = input('Enter a number: ')
    print('Sum of all input numbers is', total)

if __name__=='__main__':
    main()
```

# Iteration (for and while statements)

- The flow diagram of While structure:



# Iteration (for and while statements)

## while statement vs for statement

- Let us consider the code of for statement:

```
for count in range(1, n+1):  
    total += count
```

- We can re-write the above statement in while statement:

```
count = 1  
while count < n+1:  
    total += count  
    count += 1
```

- N.T. - For computing the sum of first few natural numbers, the use of for loop is more elegant and easy as compared to the while loop.

# Iteration (for and while statements)

Example: To print some pictures

- Program to print right triangle and inverted triangle:

```
def rightTriangle(nRows):
```

```
    """
```

Objective: To print right triangle

Input Parameter: nRows - integer value

Return Value: None

```
    """
```

```
    for i in range(1, nRows + 1)
```

```
        print('*' * i)
```

```
def invertedTriangle(nRows):
```

```
    """
```

Objective: To print inverted triangle

Input Parameter: nRows - integer value

Return Value: None

```
    """
```

```
nSpaces = 0
```

```
nStars = 2 * nRows - 1
```

```
for i in range(1, nRows+1):
```

```
    print(' ' * nSpaces + '*' * nStars)
```

```
    nStars -= 2
```

```
    nSpaces += 1
```

# Iteration (for and while statements)

Example: To print some pictures [Cont.]

```
def main():
```

```
    """
```

Objective: To print right triangle or inverted triangle

depending on user's choice

Input Parameter: None

Return Value: None

```
    """
```

```
choice = int(input('Enter 1 for right triangle.\n'+ 'Enter 2 for inverted triangle.\n'))
```

```
assert choice == 1 or choice == 2
```

```
nRows = int(input('Enter no. of rows'))
```

```
if choice == 1:
```

```
    rightTriangle(nRows)
```

```
else:
```

```
    invertedTriangle(nRows)
```

```
if __name__ == '__main__':
```

```
    main()
```

Output: (a) Right Triangle

```
*
**
***
*****
*****
*****
```

(b) Inverted Triangle

```
*****
 ****
  ***
   *
    *
```

# Iteration (for and while statements)

## Examples of nested control structures

<pre>for(...):     #Sequence of statements S1     for(...):         #Sequence of statements S2         #Sequence of statements S3</pre>	<pre>while (...):     #Sequence of statements S1     while(...):         #Sequence of statements S2         #Sequence of statements S3</pre>
<pre>for(...):     #Sequence of statements S1     while(...):         #Sequence of statements S2         #Sequence of statements S3</pre>	<pre>while (...):     #Sequence of statements S1     for(...):         #Sequence of statements S2         #Sequence of statements S3</pre>
<pre>for(...):     #Sequence of statements S1     for(...):         #Sequence of statements S2         while(...):             #Sequence of statements S3             #Sequence of statements S4             #Sequence of statements S5</pre>	<pre>while (...):     #Sequence of statements S1     for(...):         #Sequence of statements S2         while(...):             #Sequence of statements S3             #Sequence of statements S4             #Sequence of statements S5</pre>

## **break, continue and pass statements**

- The break statement is used for exiting from the loop to the statement following the body of the loop.
- The continue statement is used to transfer the control to next iteration of the loop without executing rest of the body of loop.
- The pass statement lets the program go through this piece of code without performing any action.
- The else clause can be used with for and while loop. Statements specified in else clause will be executed on smooth termination of the loop. However, if the loop is terminated using break statement, then the else clause is skipped.

# References

- [1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

# Debugging

## Lecture 4

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

1 Introduction

2 Testing

3 Debugging

# Introduction

- When a program fails to yield the desirable result, we say that it contains a bug.
- The bug could be an error such as division by zero, invalid type conversion, using a variable not defined, wrong initialization, or some other unintended operation being performed.
- The process of discovering the bugs and making the program bug-free is called debugging.
- debugging: making the program error free.

- Program testing aims to expose the presence of bugs in the programs.
- To find the bugs in a program, we test it on various inputs.
- We correct the errors found in this process and feel confident that the program will run smoothly.

# Testing

- For Example (Finding Maximum of Three Numbers):

```
def max3(n1, n2, n3):
    """
    Objective: To find maximum of three numbers
    Input Parameters: n1,n2,n3 - numeric values
    Return Value: maxNumber - numeric value
    """
    maxNumber=0
    if n1 < n2 :
        if n1 > n3:
            maxNumber = n1
    elif n2 > n3:
        maxNumber=n2
    else:
        maxNumber = n3
    return maxNumber

def main():
    """
    Objective: To find maximum of three numbers
    Input Parameter: None
    Return Value: None
    """
    n1 = int(input('Enter first number: '))
    n2 = int(input('Enter second number: '))
    n1 = int(input('Enter third number: '))
    maximum = max3(n1,n2,n3)
    print('Maximum number is:',maximum)

if __name__=='__main__':
    main()
```

# Testing

- The function max3 is intended to return the maximum of three numbers. Since the input numbers can be either positive or negative, we may have test cases containing various combinations of positive and negative numbers.
- We test this script on various permutations of the inputs: 10, 20, 30, namely:

1. 30, 20, 10	4. 20, 30, 10
2. 30, 10, 20	5. 10, 30, 20
3. 20, 10, 30	6. 10, 20, 30

- It so turns out that the result obtained is correct for all input permutations, except for the permutations 20, 10, and 30:

```
>>>
```

```
Enter first number: 20
```

```
Enter second number: 10
```

```
Enter third number: 30
```

```
Maximum number is 0
```

- Thus, a bug persists in the program that needs to be detected and removed.

# Debugging

- There are various Python debugging tools such as pdb, pudb, pydb, and pydbgr. In this section, we will discuss Python's built-in debugger pdb, which is an interactive tool that helps in examining the code execution step by step.
- Debugging allows us to stop the execution of a program at a chosen instruction called a break point, and evaluate the various expressions in the current context.
- In order to debug a program in Python shell, we need to import the module pdb as well as the script to be debugged.

```
>>> import pdb  
>>> import max3  
>>> pdb.run('max3.main()')  
> <string >(1) <module>()->None
```

- Python debugger uses the prompt (Pdb) to indicate that the program is executing in the debugging mode.

# Debugging

- Another way to execute a program in debugging mode is by including the following two lines in the script:

```
import pdb  
pdb.set_trace()
```

- Note that including above two lines at the beginning of the program for Finding maximum of three numbers. It will increase every line number by two. By invoking the function set\_trace() at the very beginning, the program would run in debugging mode right from the first instruction to be executed.
- However, since we know that the bug exists in the function max3, we may invoke set\_trace() within the function body.

# Debugging

- Let us have a look at debugging commands:

Command	Explanation
<b>h or help</b>	it lists all the available commands.
<b>h commandName</b>	it prints the description about a command.
<b>w or where</b>	Prints the stack trace (sequence of function calls currently in execution, most recent function call being at the beginning). Also shows the statement to be executed next.
<b>u or up</b>	Moves to the stack frame one level up in the stack trace
<b>d or down</b>	Moves to the stack frame one level down in the stack trace
<b>s or step</b>	Executes the current statement and moves the control to either next statement, or the function being invoked in the current statement.
<b>n or next</b>	Executes the current statement and moves the control to the next statement.unlike step command, if a function is being invoked in the current statement, the invoked function gets executed completely.
<b>r or return</b>	Continue execution until the current function returns.
<b>p expression print(expression)</b>	Prints the value of the specified expression in the current context.

# Debugging

Command	Explanation
j (jump) lineno	Jumps to the given line number for the next statement to be executed.
l or list	List 11 lines in the vicinity of the current statement
b or break [[file : ]] line[func[,cond]]	Sets the breakpoint at the specified line. Name of the file is one of the optional arguments. When the argument <b>func</b> is used, the breakpoint is set at the first executable statement of the specified function. The second argument may be used to denote a condition which must evaluate to <b>True</b> for setting the breakpoint.
tbreak [[file : ]] line[func[,cond]]	Similar to <b>break</b> command. However, the break point being set is automatically removed once it is reached
cl or clear [[file : ]] line[func[,cond]]	Clears the specified breakpoint. In the absence of an argument, clears all the breakpoints.
c or continue	Continue execution until the breakpoint is reached
a or args	Prints the argument list of the current function along with their values.
q or quit	Quits from the python debugger

# Debugging

- Execution of Finding Maximum of Three Numbers Program in debugging for the inputs 20,10,30 is shown below:

```
> f:\pythoncode \ch04 \max3.py<module>()
-> def max3(n1,n2,n3):
```

**(pdb) h s**  
s(step)

Execute the current line, stop at the first possible  
occasion (either in a function that is called or in the current function)

**(pdb)s**

```
> f:\pythoncode \ch04 \max3.py<module>()
-> def main():
```

**(pdb)s**

```
> f:\pythoncode \ch04 \max3.py<module>()
->if __name__=='__main__':
```

**(pdb)s**

```
> f:\pythoncode \ch04 \max3.py <module>()
->main()
```

**(pdb)s**

```
-call-> f:\pythoncode \ch04 \max3.py main()
->def main()
```

# Debugging

**(pdb)s**

```
> f:\pythoncode \ch04 \max3.py main()
n1= int(input('Enter first number: '))
```

**(pdb) n**

```
Enter first number: 20
```

```
> f:\pythoncode \ch04 \max3.py main()
n1= int(input('Enter second number: '))
```

**(pdb) n**

```
Enter second number: 10
```

```
> f:\pythoncode \ch04 \max3.py main()
n1= int(input('Enter third number: '))
```

**(pdb) n**

```
Enter third number: 30
```

```
> f:\pythoncode \ch04 \max3.py main()
->maximum = max3(n1,n2,n3)
```

**(pdb) p (n1, n2, n3)**

```
(20,10,30)
```

# Debugging

(pdb) n

```
> f:\pythoncode \ch04 \max3.py main()
-> print('Maximum number is',maximum)
```

(pdb) n

Maximum number is 0

—Return—

```
> f:\pythoncode \ch04 \max3.py main() -< None
-> print('Maximum number is',maximum)
```

(pdb) s

—Return—

```
> f:\pythoncode \ch04 \max3.py <module>()-> None
-> main()
```

(pdb) q

Traceback (most recent call last):

```
File "F:/pythoncode/ch04/max3.py",27 line 32, in <module >
main()
```

```
bdb.BdbQuit
```

# References

- [1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

# Scope

## Lecture 5

**Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University)  
Bhubaneswar, Odisha, India**



# Contents

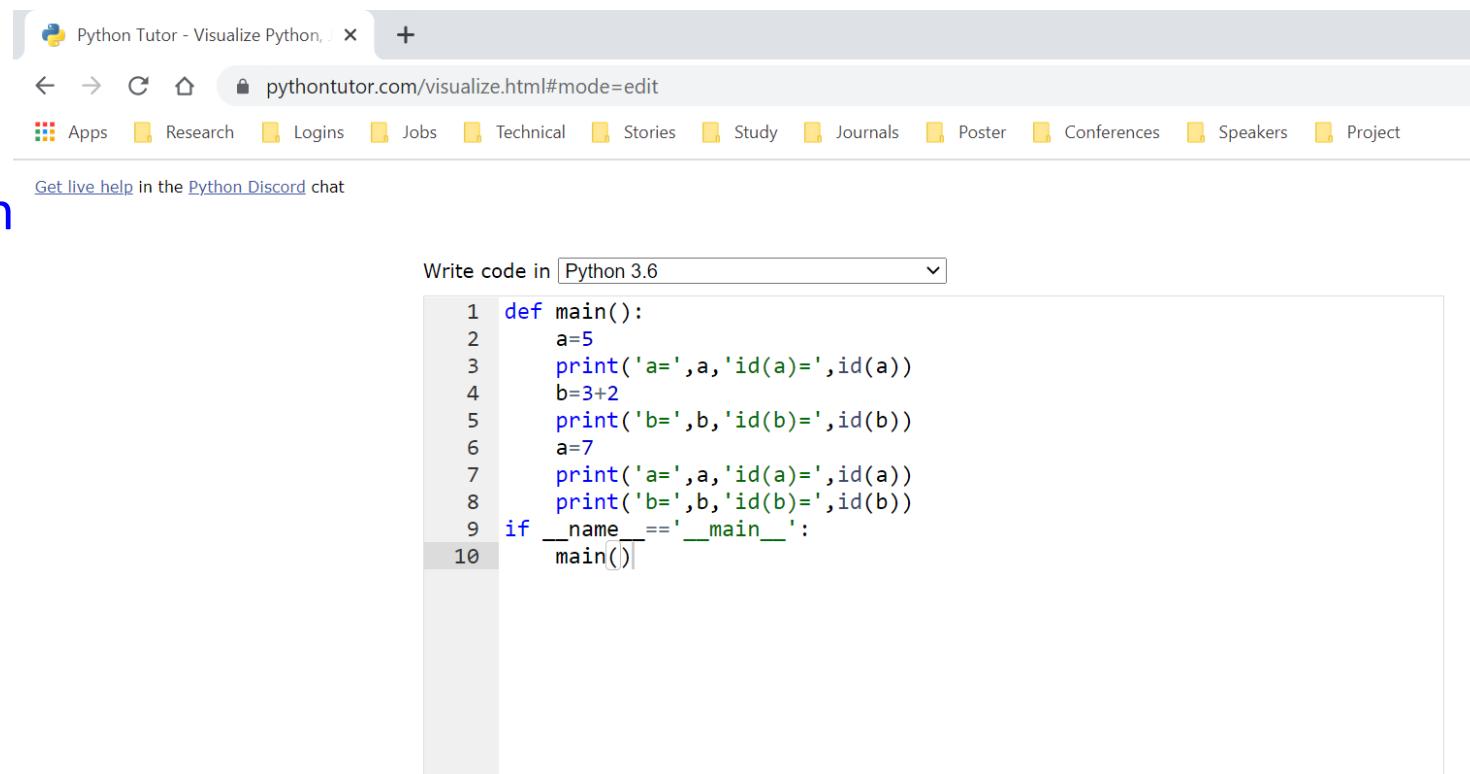
- Visualization in Python tutor
- Objects and Object IDs
- Namespaces
- Scope

\*Python Programming: A Modular Approach by Sheetal Taneja and Kumar Naveen, Pearson Education India, Inc., 2017



# Visualization of Example-I in Python tutor

- Each object in Python is assigned a unique identifier that can be accessed using the function `id`
- Visit the following website [www.pythontutor.com](http://www.pythontutor.com)
- Click on **Edit this code**
- Now write the code
- Click on **Visualize Execution**



The screenshot shows the Python Tutor interface. At the top, there's a browser-like header with a Python logo, the title "Python Tutor - Visualize Python", and a "mode=edit" URL. Below the header is a navigation bar with links like "Apps", "Research", "Logins", etc. A message "Get live help in the Python Discord chat" is visible. The main area has a "Write code in" dropdown set to "Python 3.6". The code editor contains the following Python code:

```
1 def main():
2     a=5
3     print('a=',a,'id(a)=',id(a))
4     b=3+2
5     print('b=',b,'id(b)=',id(b))
6     a=7
7     print('a=',a,'id(a)=',id(a))
8     print('b=',b,'id(b)=',id(b))
9     if __name__ == '__main__':
10        main()
```

At the bottom, there are two buttons: "Visualize Execution" and "Live Programming Mode". Below the buttons are three dropdown menus: "hide exited frames [default]", "inline primitives, don't nest objects [default]", and "draw pointers as arrows [default]". There's also a link "Create test cases".



# Example-I (continued)

- After clicking on **Visualize Execution**, a <Print output> box will appear at the right top

The screenshot shows the Python Tutor interface. At the top, there's a browser-like header with a Python logo icon, the text "Python Tutor - Visualize Python", a close button, and a plus sign for tabs. Below that is a navigation bar with icons for back, forward, refresh, and home, followed by a URL "pythontutor.com/visualize.html#mode=display". Underneath are several category links: Apps, Research, Logins, Jobs, Technical, Stories, Study, Journals, Poster, Conferences, Speakers, and Project. A link to "Get live help in the Python Discord chat" is also present.

The main area displays Python code in a syntax-highlighted editor:

```
Python 3.6
(known limitations)
→ 1 def main():
    2     a=5
    3     print('a=',a,'id(a)=',id(a))
    4     b=3+2
    5     print('b=',b,'id(b)=',id(b))
    6     a=7
    7     print('a=',a,'id(a)=',id(a))
    8     print('b=',b,'id(b)=',id(b))
    9
   10 if __name__=='__main__':
      11     main()
```

Below the code is a link "[Edit this code](#)". To the left of the code, there are two small explanatory arrows: a green arrow pointing right labeled "line that just executed" and a red arrow pointing right labeled "next line to execute".

At the bottom of the code area is a horizontal scrollbar with a slider and buttons for "<< First", "< Prev", "Next >", and "Last >". The text "Step 1 of 12" is centered below the scrollbar. At the very bottom, there's a link "[Customize visualization](#)".

To the right of the code editor is a large, empty rectangular box titled "Print output (drag lower right corner to resize)". Below this box are two buttons: "Frames" and "Objects".



# Example-I (continued)

- To start visualization, we click <Next>. On encountering the **main** function definition, the global frame lists the identifier **main** as shown in the Figure
- The **red arrow** marks the next line to be executed, and the **green arrow** marks the line just executed

The screenshot shows the Python Tutor interface visualizing the following Python code:

```
Python 3.6
(known limitations)

1 def main():
2     a=5
3     print('a=',a,'id(a)=',id(a))
4     b=3+2
5     print('b=',b,'id(b)=',id(b))
6     a=7
7     print('a=',a,'id(a)=',id(a))
8     print('b=',b,'id(b)=',id(b))
9
10 if __name__=='__main__':
11     main()
```

The code is annotated with arrows indicating the state of execution:

- A green arrow points to line 1, `def main():`, indicating it is the line just executed.
- A red arrow points to line 10, `if __name__=='__main__':`, indicating it is the next line to execute.

To the right of the code, there is a visualization of the Global frame:

- Frames**: Shows the **Global frame** containing the **main** function.
- Objects**: Shows the **function main()**.

At the bottom of the interface, there are navigation buttons: << First, < Prev, Next >, Last >>, and Step 2 of 12. There is also a link to [Customize visualization](#).



# Example-I (continued)

- Now clicking <Next>, it executes the **if** statement. Clicking <Next> again executes the call to the function **main** and the visualizer shows the frame for the **main** function
- Clicking <Next>, it moves the next line pointer to **line 2**, and its execution shows the creation of **int** object **5** having name **a**. Later, clicking <Next> shows the output of the execution of **line 3** in <Print output> box

The screenshot shows the Python Tutor interface visualizing the execution of a Python script. The code is as follows:

```
Python 3.6
(known limitations)

1 def main():
2     a=5
3     print('a=',a,'id(a)=',id(a))
4     b=3+2
5     print('b=',b,'id(b)=',id(b))
6     a=7
7     print('a=',a,'id(a)=',id(a))
8     print('b=',b,'id(b)=',id(b))
9
10 if __name__=='__main__':
11     main()

Edit this code
```

Execution details:

- Line 3 is highlighted in green, indicating it is the "line that just executed".
- Line 4 is highlighted in red, indicating it is the "next line to execute".

Output and State:

- Print output**: Shows the output of the printed statements.
- Frames**: Shows the global frame and the local frame for the **main()** function. The **main** frame contains the variable **a** with value **5**.
- Objects**: Shows the created objects: **a** (an int object with id 140426606966816) and **b** (an int object with id 140426606966817).

Navigation and Status:

- Buttons at the bottom: << First, < Prev, Next >, Last >>.
- Status bar: Step 7 of 12.
- Customization link: Customize visualization.



# Example-I (continued)

- Next click executes **line 4** and the name **b** is mapped to the **int** object **5** created earlier
- Further click executes **line 5** and the output appears in the **<Print output>** box

The screenshot shows the Python Tutor interface visualizing the execution of a Python script. The code is as follows:

```
Python 3.6
(known limitations)

1 def main():
2     a=5
3     print('a=',a,'id(a)=',id(a))
4     b=3+2
5     print('b=',b,'id(b)=',id(b))
6     a=7
7     print('a=',a,'id(a)=',id(a))
8     print('b=',b,'id(b)=',id(b))
9
10 if __name__=='__main__':
11     main()

Edit this code
```

Execution details:

- Line 4 is highlighted with a green arrow indicating it is the next line to execute.
- Line 5 has just been executed, indicated by a red arrow pointing to it.
- Output from line 5 is shown in the "Print output" box:

```
a= 5 id(a)= 140426606966816
b= 5 id(b)= 140426606966816
```

Variable state:

- Frames**: Global frame contains a reference to the **main** function.
- Objects**: The **main** function object contains two variables: **a** (value 5) and **b** (value 5).

Legend:

- Green arrow: line that just executed
- Red arrow: next line to execute

Navigation:

- << First
- < Prev
- Next >
- Last >>

Customize visualization

Step 9 of 12

# Example-I (continued)

- Clicking <Next> executes **line 6**, resulting in creation of a new **int** object **7** and its mapping to **a**
- Further clicks execute **lines 7** and **8** and the output appears in the <Print output> box
- Next click shows return value **None** associated with the function **main** as it does not return any value

The screenshot shows the Python Tutor interface visualizing the execution of a Python script. The code is as follows:

```
Python 3.6
(known limitations)

1 def main():
2     a=5
3     print('a=',a,'id(a)=',id(a))
4     b=3+2
5     print('b=',b,'id(b)=',id(b))
6     a=7
7     print('a=',a,'id(a)=',id(a))
8     print('b=',b,'id(b)=',id(b))
9
10 if __name__=='__main__':
11     main()
```

Annotations indicate the current step:

- A green arrow points to line 3, labeled "line that just executed".
- A red arrow points to line 8, labeled "next line to execute".

The "Print output" panel shows the following output:

```
a= 5 id(a)= 140426606966816
b= 5 id(b)= 140426606966816
a= 7 id(a)= 140426606966880
b= 5 id(b)= 140426606966816
```

The "Frames" panel shows the Global frame with the function `main()` and its local variables `a` and `b`:

```
Global frame
main | ↗ function main()
      a | 7
      b | 5
      Return value None
```

The "Objects" panel shows the current state of the objects:

main	a	7	
	b	5	
Return value			None



# Explanation of execution for Example-I

- Recall that when this program script is executed, Python makes a note of the definition of the function main in the global frame
- Next, on encountering the **if** statement, Python checks whether the name of the current module is **\_\_main\_\_**
- This being true, the expression **\_\_ name\_\_ == '\_\_main\_\_'** evaluates as True, and the function main gets invoked
- Next, the statements in the main function are executed in a sequence
- Execution of **line 2** creates an **int** object **5** and assigns it the name **a**. This object has **a** unique object id but can have multiple names as the execution of the script proceeds.
- For example, execution of the statement

**b = 3 + 2**

in **line 4** does not generate a new object, but only associates the name **b** to the **int** object **5** created earlier

- Now, **a** and **b** have the same object id. However, execution of **line 6** creates an **int** object **7** and associates it with the name **a**
- The name **b** continues to be associated with int object **5** created earlier.



# Object ID for different data types

- The general principle is expressed by saying that Python caches or interns small integer objects (typically, up to 100) for future use. The same may not hold for other forms of data



IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 1  
Type "help", "copyright", "credits" or "license()"  
>>> print(id(2.4))  
2228033462928  
>>> print(id(2.4))  
2228033463120  
>>> print(id(2.4))  
2228004056944  
>>> print(id(2.4))  
2228033463120  
>>> print(id(2.4))  
2228004056944  
>>> print(id(2.4))  
2228033462928  
>>>
```

- Note that the first three instructions create new objects
- However, subsequent instructions sometimes used the objects created earlier



# del operator

- It makes a name (i.e. the association between the name and the object) undefined



IDLE Shell 3.10.0

```
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 1
Type "help", "copyright", "credits" or "license()"
>>> a = 5
>>> b = 5
>>> print(id(a), id(b))
1535573229936 1535573229936
>>> del a
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined
>>> print(b)
5
>>> del b
>>> print(b)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print(b)
NameError: name 'b' is not defined
>>>
```

- The first statement creates an **int** object **5** and binds it to name **a**
- The second statement does not create a new object
- It binds the same object to name **b** and thus creates another reference to **int** object **a**



# Visualization of `del` operator in Python tutor

- The first statement creates an `int` object `5` and binds it to name `a`
- The second statement does not create a new object
- It binds the same object to name `b` and thus creates another reference to `int` object `a`

The screenshot shows the Python Tutor interface. On the left, a code editor displays the following Python script:

```
Python 3.6
(known limitations)

1 a=5
2 b=5
3 print(id(a),id(b))
4 print(a)
5 del a
6 print(b)
7 del b
8 print(b)
```

Below the code editor are two status indicators: a green arrow pointing right labeled "line that just executed" and a red arrow pointing right labeled "next line to execute". At the bottom of the code editor are navigation buttons: << First, < Prev, Next >, and Last >>. The status bar at the bottom indicates "Step 3 of 8".

To the right of the code editor is a "Print output" panel with a resize handle in the bottom-right corner. Below it are two tabs: "Frames" and "Objects". The "Objects" tab is selected, showing the "Global frame" with the following table:

b	5
a	5



# Visualization of `del` operator (continued)

- Python keeps a count of the number of references to an object
  - When the statement `del a` is executed, it reduces the reference count of `int` object **5** from 2 to 1 and removes the binding of name `a` to `int` object **5**
- Thus, an attempt to access `a` now yields an error

The screenshot shows the Python Tutor interface visualizing the state of memory after executing the code below.

```
Python 3.6
(known limitations)

1 a=5
2 b=5
3 print(id(a),id(b))
4 print(a)
5 del a
6 print(b)
7 del b
8 print(b)
```

The code is step 6 of 8. The current line being executed is `5 del a`, indicated by a green arrow. The next line to execute is `6 print(b)`, indicated by a red arrow. The output window shows the results of previous print statements: `140366912186400 140366912186400` and `5`. The objects frame shows a global frame containing an object `b` with value `5`.

Print output (drag lower right corner to resize)  
140366912186400 140366912186400  
5

Frames Objects

Global frame  
b | 5

Edit this code

line that just executed  
next line to execute

<< First < Prev Next > >> Step 6 of 8

Customize visualization



# Visualization of `del` operator (continued)

- When the statement `del b` is executed, it reduces the reference count of `int` object 5 from 1 to 0, and removes the binding of name `b` to `int` object 5
- Thus, an attempt to access `b` now yields an error

The screenshot shows the Python Tutor interface. On the left, a code editor displays the following Python script:

```
Python 3.6
(known limitations)

1 a=5
2 b=5
3 print(id(a),id(b))
4 print(a)
5 del a
6 print(b)
7 del b
8 print(b)
```

Below the code editor are two status indicators: a green arrow pointing right labeled "line that just executed" and a red arrow pointing right labeled "next line to execute". At the bottom of the code editor are navigation buttons: "<< First", "< Prev", "Next >", and "Last >". The status bar at the bottom indicates "Step 8 of 8".

To the right of the code editor is a "Print output" window containing the following text:

```
Print output (drag lower right corner to resize)
140366912186400 140366912186400
5
5
```

Below the output window are two tabs: "Frames" and "Objects".

At the very bottom of the page is a footer with the text "Customize visualization".



# Visualization of Example-II in Python tutor

- Write a program for calculating the percentage of a student in a subject

The screenshot shows the Python Tutor interface. At the top, there's a browser-like header with a Python logo icon, the text "Python Tutor - Visualize Python, Jupyter Notebook", a search bar containing "pythontutor.com/visualize.html#mode=display", and a "+" button. Below the header are navigation icons (back, forward, home) and a link to "Get live help in the Python Discord chat". A horizontal menu bar includes "Apps", "Research", "Logins", "Jobs", "Technical", "Stories", "Study", "Journals", "Poster", and "Conference".

The main area displays Python code for calculating a student's percentage. The code is presented in a step-by-step visual format:

```
Python 3.6
(known limitations)

→ 1 def percent(marks,maxMarks):
    2     percentage = (marks/maxMarks)*100
    3     return percentage
    4
    5 def main():
    6     # To compute percentage
    7     maxMarks = float(input('Enter total marks: '))
    8     marks = float(input('Enter marks obtained: '))
    9     percentage = percent(marks,maxMarks)
   10    print('Percentage is: ',percentage)
   11
   12 if __name__=='__main__':
   13    main()
```

Below the code, there's an "Edit this code" button, a legend for execution markers (green arrow for executed line, red arrow for next line), and a scroll bar. At the bottom, there are navigation buttons: "<< First", "< Prev", "Next >", and "Last >". The status bar at the bottom center says "Step 1 of 5".



# Visualization of Example-II (continued)

- We are about to execute the first of the five steps in the script
- These five steps are as follows:
  - Definition of function **percent (marks , maxMarks)**
  - Definition of function **main ()**
  - **if** statement
  - Invoking the function **main ()**
  - Execution of function **main ()**



# Visualization of Example-II (continued)

- On clicking <Next>, step 1 is executed and we see the function **percent** in the global frame

Python Tutor - Visualize Python, x +

← → C ⌂ pythonutor.com/visualize.html#mode=display

Apps Research Logins Jobs Technical Stories Study Journals Poster Conferences Speakers Prc

Python 3.6 (known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13    main()
```

Frames Objects

Global frame

function percent(marks, maxMarks)

Edit this code

line that just executed

next line to execute

<< First < Prev Next > >> Step 2 of 5

Customize visualization



# Visualization of Example-II (continued)

- On clicking <Next>, step 1 is executed and we see the function **percent** in the global frame
- On execution of next step, the function **main** is also shown in the global frame

The screenshot shows the Python Tutor interface with the following details:

- Header:** Python Tutor - Visualize Python, pythontutor.com/visualize.html#mode=display
- Toolbar:** Apps, Research, Logins, Jobs, Technical, Stories, Study, Journals, Poster, Conferences, Speakers, Prc
- Code Area:** Python 3.6 (known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13    main()
```
- Execution Status:** Step 2 of 5, showing the first line of the main() function has just been executed.
- Global Frame View:** A sidebar titled "Frames" and "Objects" shows the "Global frame" containing the variable "percent". An arrow points from the "percent" entry to its definition in the code.
- Legend:** green arrow for "line that just executed", red arrow for "next line to execute".
- Navigation:** << First, < Prev, Next >, Last >>
- Footer:** Customize visualization



# Visualization of Example-II (continued)

- When step 3 is executed, the condition `if __name__=='__main__'` evaluates as `True`
- Thus, in step 4 the function `main` is invoked

Python Tutor - Visualize Python, x + ← → C ⌂ pythontutor.com/visualize.html#mode=display Apps Research Logins Jobs Technical Stories Study Journals Poster Conferences Speakers Pro

Python 3.6 (known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13    main()
```

Frames Objects

Global frame

- percent → function `percent(marks, maxMarks)`
- main → function `main()`

Edit this code

line that just executed  
next line to execute

<< First < Prev Next > Last >>

Step 3 of 5

[Customize visualization](#)

SIKSHA 'O' ANUSANDHAN  
(DEEMED TO BE UNIVERSITY)

# Visualization of Example-II (continued)

- Next click executes the call to function **main** and the visualizer shows the function **main** among frames

Python Tutor - Visualize Python... x +

← → ⌂ ⌂ pythontutor.com/visualize.html#mode=display

Apps Research Logins Jobs Technical Stories Study Journals Poster Conferences Speakers Prc

Get live help in the [Python Discord](#) chat

Python 3.6  
([known limitations](#))

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13     main()
```

[Edit this code](#)

■ line that just executed  
► next line to execute

<< First < Prev Next > >> Step 5 of 5

Customize visualization

Frames Objects

Global frame

percent ↗ function percent(marks, maxMarks)

main ↗ function main()

main



# Visualization of Example-II (continued)

- Next click yields a message that prompts the user to enter total marks in <Input Box> and hit the **Submit** button
- This input message along with the values entered as input are also shown in <Print output>
- At this stage, we enter **500** as total marks, and click **Submit**

The screenshot shows the Python Tutor interface. On the left, the Python code is displayed:

```
Python 3.6
(known limitations)

1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13     main()
```

Annotations indicate the current line being executed (line 7) and the next line to execute (line 8). Below the code, there is an "Edit this code" link and a legend for line colors.

On the right, the "Frames" and "Objects" panel shows the state of variables:

- Global frame**:
  - `percent` points to the `percent(marks, maxMarks)` function definition.
  - `main` points to the `main()` function definition.
- main**:
  - `maxMarks` is bound to the user input value.
  - `marks` is bound to the user input value.
  - `percentage` is bound to the result of the `percent` function call.

At the bottom, there is an "Enter user input:" field containing "Enter total marks: 500" and a "Submit" button.

# Visualization of Example-II (continued)

- Now the execution is resulting in creation of an instance of **float** object **500.0**
- This object is named as **maxMarks**
- The next click executes again prompting for marks obtained (say, **450**)

Python Tutor - Visualize Python, [x](#) [+](#)

[←](#) [→](#) [C](#) [Home](#) [🔒](#) [pythontutor.com/visualize.html#mode=display](#)

[Apps](#) [Research](#) [Logins](#) [Jobs](#) [Technical](#) [Stories](#) [Study](#) [Journals](#) [Poster](#) [Conferences](#) [Speakers](#) [Project](#)

Print output (drag lower right corner to resize)  
Enter total marks: 500

Frames Objects

Global frame

```
percent → function percent(marks, maxMarks)
main → function main()
```

main

maxMarks 500.0

Python 3.6  
(known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13    main()
```

[Edit this code](#)

→ line that just executed  
→ next line to execute

<< First < Prev Next > >>

Enter user input:  
Enter marks obtained: 450

[Customize visualization](#)



# Visualization of Example-II (continued)

- As before, an instance of **float** object **450.0** is created, and named as **marks**
- The control moves the next line

Python Tutor - Visualize Python, [x](#) [+](#)

[←](#) [→](#) [C](#) [Home](#) [pythontutor.com/visualize.html#mode=display](#)

[Apps](#) [Research](#) [Logins](#) [Jobs](#) [Technical](#) [Stories](#) [Study](#) [Journals](#) [Poster](#) [Conferences](#) [Speakers](#) [Project](#)

Print output (drag lower right corner to resize)

Enter total marks: 500

Frames Objects

Global frame

```
graph TD; percent --> function_percent["function percent(marks, maxMarks)"]; main --> function_main["function main()"];
```

percent

main

main

maxMarks 500.0

Python 3.6  
(known limitations)

```
1 def percent(marks,maxMarks):  
2     percentage = (marks/maxMarks)*100  
3     return percentage  
4  
5 def main():  
6     # To compute percentage  
7     maxMarks = float(input('Enter total marks: '))  
8     marks = float(input('Enter marks obtained: '))  
9     percentage = percent(marks,maxMarks)  
10    print('Percentage is: ',percentage)  
11  
12 if __name__=='__main__':  
13    main()
```

[Edit this code](#)

line that just executed  
next line to execute

<< First < Prev Next > >> Last

Enter user input:

Enter marks obtained:  Submit

Customize visualization

SIKSHA 'O' ANUSANDHAN  
(DEEMED TO BE UNIVERSITY)

# Visualization of Example-II (continued)

- Clicking <Next> executes the call to the function **percent** and the visualizer shows the function **percent** among frames
- Note that the parameters **marks** and **maxMarks** are mapped to **float** objects created earlier

The screenshot shows the Python Tutor interface visualizing the execution of a Python script. The script defines a `percent` function and a `main` function. The `percent` function takes `marks` and `maxMarks` as parameters and returns their percentage. The `main` function prompts the user for total marks and obtained marks, then prints the percentage using the `percent` function. The visualizer shows the execution flow, the current line being executed (line 2), and the state of variables in the Global frame and the `main` frame.

Python 3.6 (known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13     main()
```

Edit this code

line that just executed  
next line to execute

<< First < Prev Next > >> Step 10 of 14

Customize visualization

Print output (drag lower right corner to resize)  
Enter total marks: 500  
Enter marks obtained: 450

Frames Objects

Global frame

percent → function percent(marks, maxMarks)  
main → function main()

main

maxMarks 500.0  
marks 450.0

percent

marks 450.0  
maxMarks 500.0



# Visualization of Example-II (continued)

- On the next click, Python creates a return value i.e. the float object (to be returned to the `main` function)  
**90.0** which was created earlier

Python Tutor - Visualize Python, x +

← → C Home 🔒 pythontutor.com/visualize.html#mode=display

Apps Research Logins Jobs Technical Stories Study Journals Poster Conferences Speakers Project

Python 3.6 (known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13     main()
```

Edit this code

line that just executed  
next line to execute

<< First < Prev Next > >> Step 11 of 14

Customize visualization

Print output (drag lower right corner to resize)

Enter total marks: 500  
Enter marks obtained: 450

Frames Objects

Global frame

- percent → function `percent(marks, maxMarks)`
- main → function `main()`

main

maxMarks	500.0
marks	450.0

percent

marks	450.0
maxMarks	500.0
percentage	90.0



# Visualization of Example-II (continued)

- The next click returns the value **90.0** from the function **percent** by associating it with the variable **percentage** of function **main**
- Note that the variable **percentage** of the function **main** now maps to the **float** object **90.0**

Python Tutor - Visualize Python, [x](#) [+](#)

[←](#) [→](#) [C](#) [Home](#) [pythontutor.com/visualize.html#mode=display](#)

[Apps](#) [Research](#) [Logins](#) [Jobs](#) [Technical](#) [Stories](#) [Study](#) [Journals](#) [Poster](#) [Conferences](#) [Speakers](#) [Project](#)

Python 3.6  
(known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13    main()
```

Print output (drag lower right corner to resize)  
Enter total marks: 500  
Enter marks obtained: 450

Frames Objects

Global frame

```
percent → function percent(marks, maxMarks)
main → function main()
```

main

```
maxMarks 500.0
marks 450.0
```

percent

```
marks 450.0
maxMarks 500.0
percentage 90.0
Return value 90.0
```

[Edit this code](#)

Green arrow: line that just executed  
Red arrow: next line to execute

[<< First](#) [< Prev](#) [Next >](#) [Last >>](#)

Step 12 of 14

[Customize visualization](#)

# Visualization of Example-II

- The next two clicks shows return value **None** associated with the function **main()** as it does not return any value

Python Tutor - Visualize Python, x +

← → C ⌂ pythontutor.com/visualize.html#mode=display

Apps Research Logins Jobs Technical Stories Study Journals Poster Conferences Speakers Project

Python 3.6 (known limitations)

```
1 def percent(marks,maxMarks):
2     percentage = (marks/maxMarks)*100
3     return percentage
4
5 def main():
6     # To compute percentage
7     maxMarks = float(input('Enter total marks: '))
8     marks = float(input('Enter marks obtained: '))
9     percentage = percent(marks,maxMarks)
10    print('Percentage is: ',percentage)
11
12 if __name__=='__main__':
13     main()
```

Print output (drag lower right corner to resize)

```
Enter total marks: 500
Enter marks obtained: 450
Percentage is: 90.0
```

Frames Objects

Global frame

```
percent → function percent(marks, maxMarks)
main → function main()
```

main

maxMarks	500.0
marks	450.0
percentage	90.0
Return value	None

Edit this code

Line that just executed  
next line to execute

<< First < Prev Next > >> Step 14 of 14

Customize visualization



# Namespaces

- As the term suggests, **a namespace** is a space that holds some names
- A **namespace** defines a mapping of names to the associated objects
- In Python, a module, class, or function defines a namespace
- Names that appear in global frame (usually outside of the definition of classes, functions, and objects) are called **global names** and collectively they define the namespace called **global namespace**
- The names introduced in a class or function are said to be **local** to it.
- The region in a script in which a name is accessible is called its **scope**
- Thus, **the scope of a name is resolved in the context of the namespace** in which it is defined



# An example

- For example, each of the functions **f1** and **f2** defined in a script may have the name **x**
- The variable **x** defined in function **f1** may refer to an object of type different from that of the object associated with variable **x** in the function **f2**
- A namespace maps names to objects
- Being an object-oriented language, definitions of functions and classes are also examples of objects



# Scope

- The scope rules for names in Python are often summarized as **LEGB rule**
- LEGB stands for **local, enclosing, global, and built in**
- All names defined within the body of a function are **local** to it
- **Function parameters** are also considered local
- If a name is not locally found, Python **recursively searches** for its definition in an enclosing scope
- Names defined in the Python script but usually outside of any function or class definition are called **global**
- Python defines some built-in names such as **len** and **abs**, which can be accessed from anywhere in a program



# Examples

- Example1: Note that as the variable **a** has global scope, it is accessible in function **f**

```
1  a = 4
2  def f():
3      print('global a: ', a)
4  f()
```

- Example 2: Note that the name **a** introduced in line 3 in the function **f** is local to it and has associated value **5**. Thus defining the value of name **a** in the function **f**, does not affect the value of the global name **a**.

```
1  a = 4.2
2  def f():
3      a = 5
4      print('local a: ', a)
5  f()
6  print('global a: ', a)
```



# Examples (continued)

- Example 3: In this example, during execution of function **g**, when the variable **a** is to be accessed, Python looks for it in the local scope of function **g**, as it is not defined in the body of function **g**, it is searched in the next enclosing scope, i.e., the scope of function **f**, where the variable **a** is indeed defined, and therefore the value **5** of the variable **a** in function **f** gets bound to the occurrence of variable **a** in the function **g**.

```
1  a = 6
2  def f():
3      a = 5
4      def g():
5          b = a
6          print('inside function g, b: ', b)
7      g()
8  f()
```

- Example 4: In this example, the variable **a** is defined in the body of inner function **g**. When we attempt to access the name **a** in line 5 of the outer function **f**, Python looks for its definition first inside the body of function **f**, as there is no definition of **a** in the function **f**, it looks for definition of **a** in the next available enclosing scope, which is the global scope. Again, there is no definition of the name **a** in global name space, and hence the error message is printed.

```
1  def f():
2      def g():
3          a = 5
4      g()
5      print('in outer function g, a =', a)
6  f()
```



# Conclusions

- Python visualizer is an online tool for visualizing the execution of Python code.
- A namespace defines a mapping of names to the associated objects.
- Names that appear in global frame outside of the definition of classes, functions, and objects are called global names, and collectively they define the namespace called global namespace.
- The names introduced in a class, or function are said to be local to it.
- The region in a script in which a name is accessible is called its scope.
- The scope rules for names in Python are often summarized as LEGB rule
- If a name is not locally defined, Python recursively searches for its definition in an enclosing scope.
- Python defines some built-in names such as len and abs which can be accessed from anywhere in a program.



# Strings

## Lecture 6

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Introduction
- 2 Slicing
- 3 Membership
- 4 Built-in Functions on Strings
- 5 String Processing Examples

# Introduction

- A string is a sequence of characters represented using single, double, or triple quotes.  
    >>> message = 'Hello Gita'
- Triple quotes are typically used for strings that span multiple lines.
- *len* function finds the length of a string  
    >>> len(message)  
    10

## Introduction (Cont.)

Individual characters within a string are accessed using a technique known as indexing.

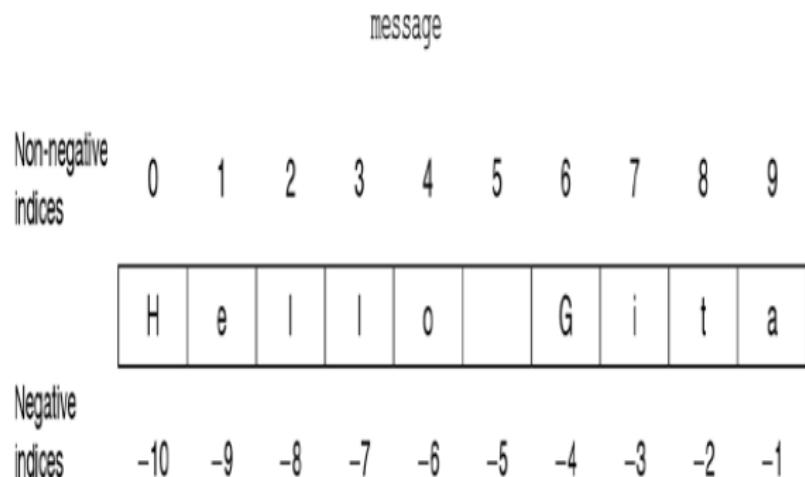


Figure 1: Indexing of variable **message**

## Introduction (Cont.)

- An index is specified within the square brackets to access individual characters in a string via indices, for example:

```
>>> message[6]
```

```
'G'
```

```
>>> index = len(message) - 1
```

```
>>> message[index]
```

```
'a'
```

- The negative indices range from  $-(\text{length of the string})$  to  $-1$ .
- The entire range of valid indices would be  $\{-10, -9, \dots, -1, 0, 1, \dots, 9\}$ .

```
>>> message[-1]
```

```
'a'
```

```
>>> message[-index]
```

```
'e'
```

## Introduction (Cont.)

```
>>> message[15]
Traceback (most recent call last):
File "<pyshell#17>", line 1, in <module>
message[15]
IndexError: string index out of range
```

Strings in Python are immutable.

```
>>> message[6] = 'S'
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
message[6] = 'S'
TypeError: 'str' object does not support item assignment
```

## Introduction (Cont.)

```
>>> 'Computer' + ' Science'  
'Computer Science'  
>>> 'Hi'*3  
'HiHiHi'
```

`max()` and `min()` are used to find maximum and minimum respectively of values.

```
>>> max('AZ', 'C', 'BD', 'BT')  
'C'  
>>> min('BD', 'AZ', 'C')  
'AZ'  
>>> max('hello', 'How', 'Are', 'You', 'sir')  
'sir'
```

# Slicing

- To retrieve a sub-string, also called a slice, from a string.

```
>>> message = 'Hello Sita'
```

```
>>> message[0:5]
```

```
'Hello'
```

```
>>> message[-10:-5]
```

```
'Hello'
```

- Python assumes 0 as the default value of start, and length of the string as the default value of end.

```
>>> message[:5]
```

```
'Hello'
```

```
>>> message[5:]
```

```
' Sita'
```

```
>>> message[:]
```

```
"Hello Sita"
```

## Slicing (Cont.)

```
>>> message[:5] + message[5:]  
'Hello Sita'  
>>> message[:15]  
'Hello Sita'  
>>> message[15:]  
"  
>>> message[:15] + message[15:]  
'Hello Sita'  
>>> message[6:None]  
'Sita'
```

# Membership

- Python allows us to check for membership of the individual characters or sub-strings in strings using *in* operator.
- The expression s in str1 yields True or False depending on whether s is a sub-string of str1, for example:

```
>>> 'h' in 'hello'
```

True

```
>>> 'ell' in 'hello'
```

True

```
>>> 'h' in 'Hello'
```

False

- For loop can be used to iterate over each element in a sequence.

# Built-in Functions on Strings (Cont.)

Functions	Explanation
s.count(str1)	counts number of times string str1 occurs in the string s
s.find(str1)	Returns index of the first occurrence of the string str1 in string s, and returns -1 if str1 is not present in string s
s.rfind(str1)	Returns index of the last occurrence of string str1 in string s, and returns -1 if str1 is not present in string s
s.capitalize(str1)	Returns a string that has first letter of the string s in uppercase and rest of the letters in lowercase

# Built-in Functions on Strings (Cont.)

s.title()	returns a string that has first letter of every word in the string s in uppercase and rest of the letters in the lowercase
s.lower()	returns a string that has all uppercase letters in string s converted into corresponding lower-case letters
s.upper()	returns a string that has all lowercase letters in string s converted into corresponding upper-case letters
s.swapcase()	returns a string that has all lowercase letters in string s converted into uppercase letters and <u>vice-versa</u>

# Built-in Functions on Strings (Cont.)

s.islower()	returns True if all alphabets in string s (comprising atleast one alphabet) are in lowercase, else returns False
s.isupper()	returns True if all alphabets in string s (comprising atleast one alphabet) are in uppercase, else returns False
s.istitle()	returns True if string s is in the title case, i.e., only first letter of each word is capitalized and the string s contains atleast one alphabet, and returns False otherwise
s.replace(str1, str2)	returns a string that has every occurrence of string str1 in s replaced by string str2

# Built-in Functions on Strings (Cont.)

s.strip()	returns a string that has whitespaces in s removed from the begining and the end
s.lstrip()	returns a string that has whitespaces in s removed from the begining
s.rstrip()	returns a string that has whitespaces in s removed from the end
s.split(delimiter)	returns a list formed by splitting the string s into substrings. The delimiter is used to mark the split points
s.partition (delimiter)	partitions the string s into three parts based on delimiter

# Built-in Functions on Strings (Cont.)

s.join(sequence)	returns a string comprising elements of the sequence separated by delimiter s
s.isspace()	returns true if all characters in string s comprise whitespace characters only, i.e., ' ', '\n', and '\t' else False
s.isalpha()	returns true if all characters in string s comprise alphabets only, and False otherwise
s.isdigit()	returns true if all characters in string s comprise digits only, and false otherwise
s.isalnum()	returns true if all characters in string s comprise alphabets and digits only, and false otherwise

# Built-in Functions on Strings (Cont.)

s.startswith(str1)	returns true if string s starts with string str1, and false otherwise
s.endswith(str1)	returns true if string s ends with string str1, and false otherwise
s.encode(encoding)	returns s in an encoded form, based on the given encoding scheme
s.decode(encoding)	returns the decoded string s, based on the <u>encoding scheme</u>

Table 1: String Functions

## Built-in Functions on Strings (Cont.)

```
>>> 'Encyclopedia'.count('c')
2
>>> colors = 'green, red, blue, red, red, green'
colors.find('red')
7
>>> colors.rfind('red')
23
>>> colors.find('orange')
-1
>>> 'python IS a Language'.capitalize()
'Python is a language'
>>> 'python IS a PROGRAMMING Language'.title()
'Python Is A Programming Language'
```

## Built-in Functions on Strings (Cont.)

```
>>> emailId1 = 'geek@gmail.com'  
>>> emailId2 = 'Geek@gmail.com'  
>>> emailId1 == emailId2  
False  
>>> emailId1.lower() == emailId2.lower()  
True  
>>> emailId1.upper() == emailId2.upper()  
True  
>>> 'pYTHON IS PROGRAMMING LANGUAGE'.swapcase()  
'Python is programming language'  
>>> 'python'.islower()  
True  
>>> 'Python".isupper()  
False
```

# Built-in Functions on Strings (Cont.)

```
>>> 'Basic Python Programming'.istitle()
True
>>> 'Basic PYTHON Programming'.istitle()
False
>>> '123'.istitle()
False
>>> 'Book 123'.istitle()
True
>>> message = 'Amey my friend, Amey my guide'
>>> message.replace('Amey', 'Vihan')
'Vihan my friend, Vihan my guide'
```

## Built-in Functions on Strings (Cont.)

```
>>> 'Hello How are you! '.lstrip()
'Hello How are you! '
>>> 'Hello How are you! '.rstrip()
' Hello How are you!'
>>> 'Hello How are you! '.strip()
'Hello How are you!'
>>> colors = 'Red, Green, Blue, Orange, Yellow, Cyan'
>>> colors.split(',')
['Red', ' Green', ' Blue', ' Orange', ' Yellow', ' Cyan']
>>> colors.split()
['Red', 'Green', 'Blue', 'Orange', 'Yellow', 'Cyan']
>>> 'Hello. How are you?'.partition('.')
('Hello', '.', ' How are you?')
```

# Built-in Functions on Strings (Cont.)

```
>>> ' '.join(['I', ' am', ' ok'])
'I > am > ok'
>>> ''.join(['I', 'am', 'ok'])
'I am ok'
>>> '''.join("''I', ' am', ' ok'''')
'''I''', '''a''', '''m''', '''o''', '''k''''
>>> name = input('Enter your name : ')
>>> Enter your name : Nikhil
>>> name.isalpha()
True
>>> name = input('Enter your name : ')
>>> Enter your name : Nikhil Kumar
>>> name.isalpha()
False
```

## Built-in Functions on Strings (Cont.)

```
>>> mobileN = input('Enter mobile no : ')
Enter mobile no : 1234567890
>>> mobileN.isdigit() and len(mobileN) == 10
True
>>> ' '.isspace()
True
>>> password = input('Enter password : ')
Enter password : Kailash107Ganga
>>> password.isalnum()
True
>>> password = input('Enter password : ')
Enter password : Kailash 107 Ganga
>>> password.isalnum()
False
>>> name = 'Ankita Narain Talwar'
>>> name.endswith('Talwar')
True
```

## Built-in Functions on Strings (Cont.)

Note that the original string S remains unchanged in each case.

# Counting the Number of Matching Characters in a Pair of Strings

```
def nMatchedChar(str1, str2):  
    """
```

Objective: to count number of occurrences of characters in str1 that are also in str2

Input parameters: str1,str2-string

Return value: count-numeric

```
"""
```

```
temp1 = str1.lower()
```

```
temp2 = str2.lower()
```

```
count=0
```

```
for ch1 in temp1:
```

```
    #search for ch1 in temp2
```

```
    for ch2 in temp2:
```

```
        if ch1==ch2:
```

```
            count+=1
```

```
return count
```

# Counting the Number of Matching Characters in a Pair of Strings (Cont.)

```
>>> name1 = 'Ram Rahim'  
>>> name2 = 'SAMARTH RAHI'  
>>> nMatchedChar(name1, name2)  
16
```

# Counting the Number of Common Characters in a Pair of Strings

””

Objective: to count number of occurrences of characters in two strings

Input parameters: str1,str2-string

Return value: count-numeric

””

# Counting the Number of Common Characters in a Pair of Strings

```
def nCommonChar(str1, str2):
    temp1 = str1.lower()
    temp2 = str2.lower()
    count=0
    for i in range(len(temp1)):
        ch1=temp1[i]
        if not(ch1 in temp1[:i]):
            #if the character has not been encountered earlier
            for ch2 in temp2:
                if ch1==ch2:
                    count+=1
                    break
    return count
```

# Reversing a String

```
def reverse(str1):
    """
    Objective: to reverse a string
    Input parameters: str1-string
    Return value: reverseStr-reverse of str1- string
    """
    reverseStr = ""
    for i in range(len(str1)):
        reverseStr=str1[i] + reverseStr
    return reverseStr
```

# Reversing a String (Cont.)

using recursion

```
def reverse(str1):
    """
    Objective: to reverse a string
    Input parameters: str1-string
    Return value: reverse of str1- string
    """
    if str1 == "":
        return str1
    else:
        return reverse(str1[1:]) + str1[0]
```

# Pattern Matching

- **Alphabet ( $\Sigma$ ):** An alphabet is a non-empty set of symbols
- **String:** A string is a finite sequence of symbols chosen from the alphabet. An empty string " containing no symbols is called null string and is often denoted by  $\lambda$  or  $\epsilon$ . The length of a string is defined as the number of symbols in the string. The length of the null string is defined to be zero.
- **Language:** It is the set of strings (words) defined over the alphabet that conforms to some predefined pattern, or rule(s).
- A regular language is described by a regular expression
- regular expressions: used to define regular languages

## Pattern Matching (Cont.)

- Each symbol of the alphabet defines a regular language, comprising the symbol itself.
- If  $r$  is regular expression,  $L(r)$  denotes the language described by  $r$ .
- $\lambda$  or  $\epsilon$  is a regular expression that denotes the language comprising the null string only.
- The language containing no word, not even  $\lambda$  is called null or empty language  $\{\}$ , and is denoted by the regular expression  $\phi$ .
- If  $r$  and  $s$  are regular expressions,  $r|s$  is also a regular expression and denotes the language  $L(r|s) = L(r) \cup L(s)$ .
- If  $r$  is a regular expression, so is  $(r)$ , denoting the same language, i.e.  $L(r) = L((r))$ . Parentheses are used to enforce precedence of operators in the regular expressions.
- The regular expression  $rs$  denotes the language  $L(rs) = L(r) L(s)$
- Concatenation has higher precedence than the union operator

## Pattern Matching (Cont.)

- Concatenation is not commutative
- The regular expression  $(0|1)1$  denotes the language  $L((0|1)1) = L((0|1))L(1) = \{0,1\} \{1\} = \{01,11\}$ .
- If  $r$  is a regular expression,  $r^*$  is also a regular expression.  $*$  denotes **zero or more** occurrences of the preceding pattern  $r$ .
- $0^*$  defined over alphabet  $\Sigma = \{0, 1\}$  defines the language,  $L = \{\lambda, 0, 00, 000, 0000, \dots\}$
- Given a pattern  $r$ ,

$$r^+$$

denotes the language comprising **one or more** occurrences of strings that match the pattern  $r$ .

- For dealing with a regular expression in Python, we need to import the module `re`, which contains functions for handling the regular expressions

# Pattern Matching (Cont.)

Symbols used in regular expression	Meaning
*	zero or more occurrences of the preceding pattern
.	exactly one arbitrary character excluding new-line
?	zero or one occurrence of the preceding pattern
+	one or more occurrences of the preceding pattern
{m}	exactly m occurrences of the preceding pattern

## Pattern Matching (Cont.)

{m,n}	at least m and at most n occurrences of preceding term. In absence of n, there is no upper bound and in the absence of m, the lower bound is assumed to be zero
{list-of-char}	a single character from list of characters enclosed between []
[.]	matches dot (not an arbitrary character)
[a-z]	a single character in the range a to z
[A-Z]	a single character in the range A to Z

# Pattern Matching (Cont.)

[0-9]

[^...]

^

\$

r1|r2

A single digit in the range 0 to 9  
when ^ occurs at the beginning of a list symbols enclosed between [], it denotes a single character not in the list

matches beginning of the string

matches end of the string or just before the newline character at the end of the string

regular expression r1 or r2

# Pattern Matching (Cont.)

()	Groups pattern elements
\d	any digit
\D	any non-digit character
\s	whitespace character
\S	Non-whitespace character
\w	any alphanumeric character including _
\W	<u>any non-alphanumeric character excluding _</u>

Table 2: symbols used in regular expressions

# Pattern Matching (Cont.)

regular expression	set of matched patterns
python	{python}
{p P}ython	{python, Python}
a*	{ $\lambda$ , a, aa, aaa, ...}
a+	{, a, aa, aaa,...}
a?	{ $\lambda$ , a}
[aeiou]	{a,e,i,o,u}
[ab]?	{ $\lambda$ , a, b}
[ab]*	{ $\lambda$ , a, b, aa, ab, bb, aa, aab, ...}

## Pattern Matching (Cont.)

regular expression	set of matched patterns
\d	{0,1,2,3,4,5,6,7,8,9}
\d{2}	{00,01,02,03,...,99}
\d{2,3}	{00,01,02,...99,000,001,002,...,999}
\D	{a,b,...,z,A,B,...,Z, *, \$, !, ....}
\w	{a,b,...,z,A,B,...,Z,0,1,2,...,9}
\s	{space, tab, newline, carriage return}
[^a-b]	the set comprising characters other than 0 and 1
(a b)(c)\$	{ac,bc}, there should be no character after c in the string that matches the pattern

## Pattern Matching (Cont.)

$^a(0 1)^*$	$\{a, a0, a1, a00, a01, \dots\}$ , a should be first in the string which pattern matches
$a\^b$	$\{a^*b\}$ , when the backslash character precedes a character with a special meaning, the special meaning of the character is ignored. Example, although the regular expression $a^*b$ defines a pattern comprising zero or more occurrences of a, followed by b, the pattern $a\^b$ defines the string ' $a^*b$ '

Table 3: Examples of regular expressions and the corresponding languages

## Pattern Matching (Cont.)

- The function **search** of this module is used for matching a regular expression in the given string.
- It looks for the first location of a match in the given string.
- If the search is successful, it returns the **matchObject** instance matching the regular expression pattern, otherwise it returns **None**.
- The function **group** of matchObject instance returns the substring that matches the regular expression.

## Pattern Matching (Cont.)

Regular Expression	Example
Python	<pre>&gt;&gt;&gt; string1 = 'Welcome to python shell' &gt;&gt;&gt; match = re.search('python', string1) &gt;&gt;&gt; match.group() 'python'</pre>
{p P}ython	<pre>&gt;&gt;&gt; string1 = 'Welcome to Python shell' &gt;&gt;&gt; match = re.search('{p P}ython', string1) &gt;&gt;&gt; match.group() 'Python'</pre>
Shel*	<pre>&gt;&gt;&gt; string1 = 'Python shell' &gt;&gt;&gt; match = re.search('Shel*', string1) &gt;&gt;&gt; match.group() 'Shell'</pre>

## Pattern Matching (Cont.)

Shel?

```
>>> string1 = 'Python shell'  
>>> match = re.search('Shel?', string1)  
>>> match.group()  
'Shel'
```

Shel{1,2}

```
>>> string1 = 'Python shelllll'  
>>> match = re.search('Shel{1,2}', string1)  
>>> match.group()  
'Shell'
```

.....

```
>>> string1 = 'Python shell'  
>>> if re.search('.....', string1):  
    print('String length is greater than or  
equal to 5')  
String length is greater than or equal to 5
```

## Pattern Matching (Cont.)

^Python	>>> string1 = 'Python is a powerful language' >>> if(re.search('^Python', string1)) print('String starts with python')  'String starts with python'
^power	>>> string1 = 'Python is a powerful language' >>> if(re.search('^power', string1)) print('String starts with power') else: print('String does not start with power')  String does not start with power
powerful\$	>>> string1 = 'Python is a powerful language' >>> if(re.search('powerful\$', string1)) print('String ends with powerful') else: print('String does not ends with powerful')  String does not ends with powerful

## Pattern Matching (Cont.)

language\$	>>> string1 = 'Python is a powerful language' >>> if(re.search('language\$', string1)) print('String ends with language') else: print('String does not ends with lan- guage') String ends with language
\d\d\d\d\d	>>> string1 = 'Roll number is 23456' >>> match=re.search('\d\d\d\d\d', string1) >>> match.group() '23456'
\d{5}	>>> string1 = 'Roll number is 23456' >>> match=re.search('\d{5}', string1) >>> match.group() '23456'

## Pattern Matching (Cont.)

-[0-9]+\.+[0-9]+	<pre>&gt;&gt;&gt; string1 = 'Decrease in price is -45.89' &gt;&gt;&gt; match = re.search('-[0-9]+\.+[0-9]+', string1) match.group() '-45.89'</pre>
\w*	<pre>&gt;&gt;&gt; string1 = 'Python Shell' &gt;&gt;&gt; match = re.search('\w*', string1) match.group() 'Python'</pre>
\w*\s\w*	<pre>&gt;&gt;&gt; string1 = 'We used Python Shell' &gt;&gt;&gt; match = re.search('\w*\s\w*', string1) match.group() 'We used'</pre>
*	<pre>&gt;&gt;&gt; string1 = 'I use **Python**, do you?' &gt;&gt;&gt; match = re.search('.*', string1) match.group() 'I use **Python**, do you?'</pre>

## Pattern Matching (Cont.)

(a(b|c))\*

```
>>> string1 = 'abac12ccaab'  
>>> match = re.search('(a(b|c))*', string1)  
match.group()  
'abac'
```

(a(b|c))\*\d{1,2}c\*

```
>>> string1 = 'abac12ccaab'  
>>> match = re.search('(a(b|c))*\d{1,2}c*',  
string1)  
match.group()  
'abac12cc'
```

\w\*\d\d.\*b\$

```
>>> string1 = 'abac12ccaab'  
>>> match = re.search('\w*\d\d.*b$\s\w*',  
string1)  
match.group()  
'abac12ccaab'
```

## Pattern Matching (Cont.)

(a(b c))*	<pre>&gt;&gt;&gt; string1 = 'abac12ccaab' &gt;&gt;&gt; match = re.search('(a(b c))*', string1) match.group() 'abac'</pre>
(a(b c))+	<pre>&gt;&gt;&gt; string1 = 'abac12ccaab' &gt;&gt;&gt; match = re.search('(a(b c))+', string1) match.group() 'abac'</pre>

Table 4: Python examples of regular expressions

## Pattern Matching (Cont.)

To find email ids (pranav.gupta@cs.iitd.ac.in) from a string:

a sequence of alphanumeric characters	denoted by [a-zA-Z0-9]+
a repeating(0 or more times) sequence of dots followed by alphanumeric characters	denoted by (\.[a-zA-Z0-9]+)*
@	denoted by @
sequence of alphabetic characters	denoted by [a-zA-Z]+
repeating (1 or more times) sequence of dot followed by alphabetic characters	denoted by (\.[a-zA-Z]+)+

An email id may be represented using the regular expression  
 $[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*@[a-zA-Z]+(\.[a-zA-Z]+)+$ .

## Pattern Matching (Cont.)

- Generally, a regular expression is preceded with r to denote a raw string.
- Use of a raw string as a regular expression avoids any confusion with some characters that have special meaning in regular expressions.

```
>>> match = re.search(r'[a-zA-Z]+(\.[a-zA-Z]+)*@[a-zA-Z]+\(\.[a-zA-Z]+\)+',  
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,  
raman@gmail.com')
```

```
>>> match.group()
```

```
'ram@gmail'
```

## Pattern Matching (Cont.)

- **finditer:** retrieving all substrings matching a regular expression.

```
>>> for i in re.finditer(r'[a-zA-Z]+(\.[a-zA-Z]+)*@[a-zA-Z]+(\.[a-zA-Z]+)+',  
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,  
raman@gmail.com'):  
    print(i.group())
```

```
'ram@gmail.com'  
'pranav.gupta@cs.iitd.ac.in'  
'nik@yahoo.com'  
'raman@gmail.com'
```

## Pattern Matching (Cont.)

- searching for words ending with **ing** in a string

```
>>> for i in re.finditer(r'[A-Za-z]+ing', 'Walking down the road, he was  
thinking about the coming years.'): 
```

```
>>> print(i.group()) 
```

Walking

thinking

coming

- **findall:** retrieve a list of all the substrings matching a regular expression

```
>>> for i in re.findall(r'[A-Za-z]+ing', 'Walking down the road, he was  
thinking about the coming years.'): 
```

```
['Walking', 'thinking', 'coming'] 
```

## Pattern Matching (Cont.)

```
>>> message = 'Python:Python is an interactive language. It is  
developed by Guido Van Rossum'  
>>> words = re.findall('\w+', message)  
>>> len(words)  
13  
>>> re.findall(r'([a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*@[a-zA-Z]+\(\.[a-zA-Z]+\)+)',  
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,  
raman@gmail.com')  
[('ram@gmail.com', '', '.com'), ('pranav.gupta@cs.iitd.ac.in', '.gupta',  
.in'), ('nik@yahoo.com', '', '.com'), ('raman@gmail.com', '', '.com')]
```

## Pattern Matching (Cont.)

- regular expression to extract all the single line comments

#.\*

Example:

```
>>> pythonCode = ""  
""""
```

Python code to add two numbers.

""""

```
a = 5 #number1
```

```
b = 5 #number2
```

Compute addition of two numbers

```
c = a + b
```

""

```
>>> for i in re.finditer(r'(.*)', pythonCode):
```

```
>>> print(i.group())
```

#number1

#number2

#Compute addition of two numbers

## Pattern Matching (Cont.)

- regular expression to extract all the multi-line comments

```
""".*?"""
```

**Note:** a dot in a regular expression includes any character except end of line. However, if we include re.DOTALL as the third argument in the function **finditer**, dot matches newline character also. For example:

```
>>> for i in re.finditer(r'""".*?"""', pythonCode, re.DOTALL):  
>>> print(i.group())
```

```
"""
```

Python code to add two numbers.

```
"""
```

## Pattern Matching (Cont.)

```
>>> re.split(r',', 'Mira, Ronit, Vivek')
['Mira', 'Ronit', 'Vivek']
```

re.split (): returns a list of the substrings delimited by the regular expression provided

```
>>> re.split(r',|\n', '"Mira,Rohit,Vivek
Aiysha,Renuka,Robin
Sneha,Ravi"')
['Mira', 'Rohit', 'Vivek', 'Aiysha', 'Renuka', 'Robin', 'Sneha', 'Ravi']
```

# References

- [1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You  
Any Questions?

# Mutable and Immutable Objects

## Lecture 7

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

1 Introduction

2 Lists

3 Sets

4 Tuples

5 Dictionary

# Introduction

- In Python, ‘mutable’ is the ability of objects to change their values.
- In Python, if the value of an object cannot be changed over time, then it is known as immutable.
- List, Set and Dictionary are mutable.
- Strings and Tuples are immutable.
- Elementary forms of data such as numeric and Boolean are called scalar data types.
- Several applications require more complex forms of data, for example, the name of a person, coordinates of a point, a set of objects, or a list of personal records of individuals.

- A list is an ordered sequence of values. It is a non-scalar type.
- Values stored in a list can be of any type such as string, integer, float, or list, for example, a list may be used to store the names of subjects:  
`>>> subjects=['Hindi', 'English', 'Maths', 'History']`
- The elements of a list are enclosed in square brackets, separated by commas.
- Elements of a list are arranged in a sequence beginning index 0, just like characters in a string.

## Lists (Cont.)

- We show the representation of the list subjects as in PythonTutor.



- To understand the lists better, let us invoke the function id that returns object identifier for the list object subjects:

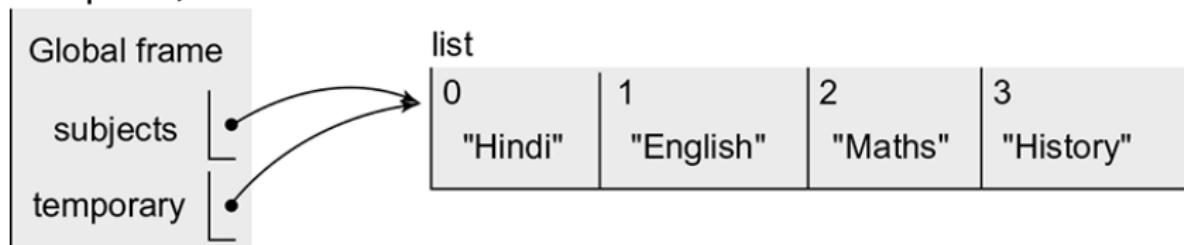
```
>>> id(subjects)  
57135752
```

- Different variables may refer to the same list object

```
>>> temporary = subjects  
>>> id(temporary)  
57135752
```

## Lists (Cont.)

- Each of the names subjects and temporary is associated with the same list object having object id 57135752.
- PythonTutor representation of the lists subjects and temporary, at this point, is shown below:



- This method of accessing an object by different names is known as aliasing.

## Lists (Cont.)

- As each of the two names temporary and subjects refers to the same list, on modifying the component temporary[0] of the list temporary, the change is reflected in subjects[0] as shown below:

```
>>> temporary[0] = 'Sanskrit'  
>>> print(temporary)  
['Sanskrit', 'English', 'Math', 'History']  
>>> print(subjects)  
['Sanskrit', 'English', 'Math', 'History']  
>>> print(id(subjects), id(temporary))  
57135752 57135752
```

- **Two-dimensional list: list of lists**
- we create a list of subjects and their corresponding subject codes. For this purpose, we represent each pair of subject and subject code as a list, and form a list of such lists:

```
>>> subjectCodes = [['Sanskrit', 43], ['English', 85], ['Maths', 65],  
['History', 36]]
```

- A list of lists such as subjectCodes, each of whose elements itself is a list, is called a two-dimensional list.
- Thus, subjectCodes[1] being a list, its components may be accessed as subjectCodes[1][0] and subjectCodes[1][1]:

```
>>> subjectCodes[1]
```

```
['English', 85]
```

```
>>> print(subjectCodes[1][0], subjectCodes[1][1])
```

```
English 85
```

## Lists (Cont.)

- Often times, we need to take a list as an input from the user. For this purpose, we use the function **input** for taking the input from the user, and subsequently apply the function **eval** for transforming the raw string to a list:

```
>>> details = eval(input('Enter details of Megha: '))
Enter details of Megha: ['Megha Verma', 'C-55, Raj Nagar,Pitam
Pura, Delhi - 110034', 9876543210]
>>> details
['Megha Verma', 'C-55, Raj Nagar,Pitam Pura, Delhi - 110034',
9876543210]
```

## Lists (Cont.)

- we describe some functions and operations on lists with the help of the following lists:

```
>>> list1 = ['Red', 'Green']  
>>> list2 = [10, 20, 30]
```

## Lists (Cont.)

Operation	Example
Multiplication Operator *	<pre>&gt;&gt;&gt; list2 * 2</pre> <code>[10, 20, 30, 10, 20, 30]</code>
Concatenation Operator +	<pre>&gt;&gt;&gt; list1=list1 + ['Blue']</pre> <pre>&gt;&gt;&gt; list1</pre> <code>['Red', 'Green', 'Blue']</code>
Length Operator len	<pre>&gt;&gt;&gt; len(list1)</pre> 3
Indexing	<pre>&gt;&gt;&gt; list2[-1]</pre> 30
Slicing Syntax: start:end:inc	<pre>&gt;&gt;&gt; list2[0:2]</pre> <code>[10, 20]</code> <pre>&gt;&gt;&gt; list2[0:3:2]</pre> <code>[10, 30]</code>
Function min	<pre>&gt;&gt;&gt; min(list2)</pre> 10

## Lists (Cont.)

Operation	Example
Function <b>max</b>	<code>&gt;&gt;&gt; max(list1)</code> <code>'Red'</code>
Function <b>sum</b> (Not defined on strings)	<code>&gt;&gt;&gt; sum(list2)</code> <code>60</code>
Membership operator <b>in</b>	<code>&gt;&gt;&gt; 40 in list2</code> <code>False</code>

Table 1: Summary of operations that can be applied on lists

## Lists (Cont.)

- The membership operator **in** may be used in a for loop for sequentially iterating over each element in the list, for example: iterating over the elements of a list

```
>>> students = ['Ram', 'Shyam', 'Gita', 'Sita']
>>> for name in students:
    print(name)
```

Ram

Shyam

Gita

Sita

## Lists (Cont.)

- We list some important built-in functions that can be applied to lists.
- Many of these functions such as **append**, **reverse**, **sort**, **extend**, **pop**, **remove**, and **insert** modify the list.
- Functions such as **count** and **index** do not modify the list.
- **dir(list)** outputs the list including all the functions that can be applied to objects of the type **list**.

## Lists (Cont.)

Function	Explanation
L.append(e)	Inserts the element e at the end of list L.
L.extend(L2)	Inserts the item in the sequence L2 at the end of elements of the list L.
L.remove(e)	Removes the first occurrence of the element e from the list L
L.pop(i)	Returns the elements from the list L at index i, while removing it from the list.
L.count(e)	Returns count of occurrences of object e in the list L.
L.index(e)	Returns index of an object e, if present in list.
L.insert(i,e)	Inserts element e at index i in list.
L.sort()	Sorts the elements of list.
L.reverse()	Reverses the order of elements in the list.

Table 2: List functions

# Lists (List Comprehension)

- List comprehension provides a shorthand notation for creating lists.
- Suppose we want to create a list that contains cubes of numbers ranging from 1 to 10.
- To do this, we may create an empty list, and use a for-loop to append the elements to this list:

```
>>> cubes = []
>>> end = 10
>>>for i in range(1, end + 1):
    cubes.append(i ** 3)
>>> cubes
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

# Lists (List Comprehension)

- Alternatively, a simple one line statement can be used for achieving the same task.

```
>>> cubes = [x**3 for x in range(1, end + 1)]
```

# Lists (List as Arguments)

```
1 def listUpdate(a, i, value):  
2     """  
3         Objective: To change a value at a particular index in list  
4         Input Parameters:  
5             a - list  
6             i - index of the object in the list to be updated  
7             value - modified value at index i  
8         Return value: None  
9         """  
10        a[i] = value  
11    def main():  
12        """  
13            Objective: To change a value at a particular index in list  
14            Input Parameters: None  
15            Return value: None  
16            """  
17        lst = [10, 20, 30, [40, 50]]  
18        listUpdate(lst, 1, 15)  
19        print(lst)  
20    if __name__=='__main__':  
21        main()
```

## Lists (List as Arguments)

- The function `listUpdate` updates the list **a** by replacing the object `a[i]` by value.
- The main function invokes the function `listUpdate` with the arguments `lst`, `1`, and `15` corresponding to the formal parameters `a`, `i`, and `value`.
- As arguments are passed by reference, during execution of the function `listUpdate`, an access to the formal parameter **a** means access to the list **lst** created in the main function.
- Consequently, when we update the list **a** in the function `listUpdate`, it results in the corresponding update of the list **lst**.
- Thus, the value at index `1` of the list **lst** gets updated to the value `15`.

# Lists (Copying List Objects)

- assigning a list to another name does not create another copy of the list, instead it creates another reference

```
>>> list1 = [10, 20, [30, 40]]
```

```
>>> list2 = list1
```

- As the names list1 and list2 refer to the same list object, any changes made in the list will relate to both the names list1 and list2, for example:

```
>>> list1[1] = 22
```

```
>>> list1
```

```
[10, 22, [30, 40]]
```

```
>>> list2
```

```
[10, 22, [30, 40]]
```

## Lists (Copying List Objects)

- To create another instance of the list object having different storage, we need to import the copy module and invoke the function **copy.copy()**:

```
>>> import copy  
>>> list1 = [10, 20, [30, 40]]  
>>> list3 = copy.copy(list1)
```

- the copy function creates a new copy list3 of the list1. Consequently, on modifying list1[1], list3[1] remains unaltered:

```
>>> list1[1] = 25  
>>> list1  
[10, 25, [30, 40]]  
>>> list3  
[10, 20, [30, 40]]
```

## Lists (Copying List Objects)

- However, the copy function creates a shallow copy i.e. it does not create copies of the nested objects. Thus, the two lists share the same nested objects.
- list1[2] and list3[2] refer to the same nested list [30, 40].
- let us modify list1[2][0] in sub-list list1[2] of list list1 to value 35.

```
>>> list1[2][0] = 35
```

```
>>> list1
```

```
[10, 25, [35, 40]]
```

```
>>> list3
```

```
[10, 20, [35, 40]]
```

## Lists (Copying List Objects)

- To create a copy of a list object so that the nested objects (at all levels) get copied to new objects, we use the function `deepcopy` of the `copy` module:
- **deepcopy()**: To create a copy of a list including copies of the nested objects at all level

```
>>> import copy  
>>> list1 = [10, 20, [30, 40]]  
>>> list4 = copy.deepcopy(list1)  
>>> list1[2][0] = 35  
>>> list1  
[10, 20, [35, 40]]  
>>> list4  
[10, 20, [30, 40]]
```

# Lambda Expression

- A lambda expression consists of the lambda keyword followed by a comma separated list of arguments and the expression to be evaluated using the list of arguments in the following format:

- **lambda arguments: expression**

- Example: lambda expression to compute the cube of a number

```
>>> cube = lambda x: x ** 3
```

```
>>> cube(3)
```

27

- Example: lambda expression to compute sum of cubes of two numbers

```
>>> sum2Cubes = lambda x, y: x**3 + y**3
```

```
>>> sum2Cubes(2, 3)
```

35

# map, reduce, and filter Operations on a Sequence

- Python provides several built-in functions based on expressions, which work faster than loop-based user defined code.
- The function map is used for transforming every value in a given sequence by applying a function to it.
- It takes two input arguments: the iterable object (i.e. object which can be iterated upon) to be processed and the function to be applied, and returns the map object obtained by applying the function to the list as follows:
- **result = map(function, iterable object)**
- The function to be applied may have been defined already, or it may be defined using a lambda expression which returns a **function** object.

# map, reduce, and filter Operations on a Sequence

- Mapping every value in the sequence to its cube

```
>>> lst = [1, 2, 3, 4, 5]
>>> list(map(lambda x: x ** 3, lst))
[1, 8, 27, 64, 125]
>>> list(map(cube, lst))
[1, 8, 27, 64, 125]
```

- We may also use any system defined or user-defined function as an argument of the map function, for example:

```
>>> list(map(abs, [-1, 2, -3, 4, 5]))
[1, 2, 3, 4, 5]
```

# map, reduce, and filter Operations on a Sequence

- Suppose we wish to compute the sum of cubes of all elements in a list.
- Adding elements of the list is a repetitive procedure of adding two elements at a time.
- The function `reduce`, available in `functools` module, may be used for this purpose. It takes two arguments: a function, and a iterable object, and applies the function on the iterable object to produce a single value:

```
>>> lstCubes = list(map(lambda x: x ** 3, lst))
>>> import functools
>>> sumCubes = functools.reduce(lambda x, y: x + y, lstCubes)
>>> sumCubes
225
```

# map, reduce, and filter Operations on a Sequence

- To compute the sum of cubes of only those elements in the list `IstCubes` that are even.
- Thus, we need to **filter** the even elements from the list `IstCubes`
- Python provides the function **filter** that takes a function and a iterable object as the input parameters and returns only those elements from the iterable object for which function returns True.
- Since the function **filter** returns a **filter** object, we have used the function **list** to convert it to list.

# map, reduce, and filter Operations on a Sequence

```
>>> evenCubes = list(filter(lambda x: x%2== 0, lstCubes))
>>> evenCubes
[8, 64]
>>> sumEvenCubes = functools.reduce(lambda x, y: x + y,
evenCubes)
>>> sumEvenCubes
72
```

- Alternatively, the sum of odd cubes may be computed as follows:  

```
>>> sumEvenCubes = functools.reduce(lambda x, y: x + y,
filter(lambda x: x%2 == 0, lstCubes))
>>> sumEvenCubes
72
```
- The functions map, reduce, and filter are often used to exploit the parallelism of the system to distribute computation to several processors.

# Sets

- A comma separated unordered sequence of values enclosed within curly braces is called **Set**.
- Function **set()** is used to convert a sequence to a set.

```
>>> vowels = set('aeiou')
```

```
>>> vowels
```

```
{'e', 'a', 'o', 'u', 'i'}
```

```
>>> vehicles = set(['Bike', 'Car', 'Bicycle', 'Scooter'])
```

```
>>> vehicles
```

```
{'Scooter', 'Car', 'Bike', 'Bicycle'}
```

```
>>> digits = set((0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
```

```
>>> digits
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

# Sets

- Elements of a set must be immutable.
- Set type does not support indexing, slicing, + operator, and \* operator.
- We can iterate over elements of the set using **in** operator  

```
>>> for v in vowels:  
    print(v, end = ' ')  
e a o u i
```
- The functions **min**, **max**, **sum**, and **len** work for sets in the same manner as defined for lists.  

```
>>> len(vehicles)  
4
```
- The membership operator **in** checks whether an object is in the set.  

```
>>> 'Bike' in vehicles  
True
```

# Sets (Functions)

- List of some important built-in functions that can be applied to sets.

Function	Description
S.add(e)	Adds the elements to the set S, if not present already.
S1.update(L1)	Add the items in object L1 to the set S1, if not already present.
S.remove(e)	Removes the element e from set S.
S.pop()	Removes an element from the set S.
S.clear()	Removes all elements from the set S.
S.copy()	Creates a copy of the set S.
S1.union(S2)	Returns union of the Sets S1 and S2.
S1.intersection(S2)	Returns a set containing common elements of sets S1 and S2.

# Sets (Functions)

- List of some important built-in functions that can be applied to sets.

Function	Description
<code>S1.difference(S2)</code>	Returns a set containing elements in set S1 but not in set S2.
<code>S1.symmetric_difference(S2)</code>	Returns a set containing elements that are in one of the two sets S1 and S2, but not in both.

Table 3: Set Functions

- The operators `<=`, `==`, and `>=` may be used to check whether a given set is a subset, equal, or superset of another set.

# Sets (Set Comprehension)

- Set comprehension is applied to find common factors
- A common factor cannot exceed smaller of the two numbers, say  $n_1$  and  $n_2$ .
- To build a set of common factors, we make use of comprehensions. For each number  $i$  in  $\text{range}(1, \min(n_1, n_2) + 1)$ , we include it in the set if it is a factor of each of  $n_1$  and  $n_2$ .
- Thus, our code comprises just one line:

```
1 commonFactors = {i for i in range(1,min(n1+1, n2+1)) if n1%  
    i == 0 and n2%i ==0}
```

# Tuples

- A **tuple** is a non-scalar type defined in Python. Just like a list, a tuple is an ordered sequence of objects.
- However, unlike **lists**, **tuples** are immutable, i.e. elements of a **tuple** cannot be overwritten.
- A tuple may be specified by enclosing in the parentheses, the elements of the tuple (possibly of heterogeneous types), separated by commas, for example, the tuple t1 comprises five objects:

```
>>> t1 = (4, 6, [2, 8], 'abc', {3,4})  
>>> type(t1)  
<class'tuple'>
```

# Tuples

- If a tuple comprises a single element, the element should be followed by a comma to distinguish a tuple from a parenthesized expression, for example:

```
>>> (2,)  
(2,)
```

- A tuple having a single element is also known as singleton tuple.
- Another notation for tuples is just to list the elements of a tuple, separated by commas:

```
>>> 2, 4, 6  
(2, 4, 6)
```

- Elements of a tuple may be mutable

```
>>> t1 = (1, 2, [3, 4])  
>>> t1[2][1] = 5  
>>> t1  
(1, 2, [3, 5])
```

# Tuple Operations

- we summarize the operations on tuples and use the following tuples t1 and t2 for the purpose of illustration:

```
>>> t1 = ('Monday', 'Tuesday')
```

```
>>> t2 = (10, 20, 30)
```

Operation	Example
Multiplication Operator *	>>> t1 * 2 ('Monday', 'Tuesday', 'Monday', 'Tuesday')
Concatenation Operator +	>>> t3=t1 + ('Wednesday',) >>> t3 ('Monday', 'Tuesday', 'Wednesday')
Length Operator len	>>> len(t1) 2
Indexing	>>> t2[-2] 20

# Tuple Operations

Operation	Example
Slicing Syntax: start:end:inc	<code>&gt;&gt;&gt; t1[1:2] ('Tuesday', )</code>
Function <b>min</b>	<code>&gt;&gt;&gt; min(t2) 10</code>
Function <b>max</b>	<code>&gt;&gt;&gt; max(t2) 30</code>
Function <b>sum</b> (not defined on strings)	<code>&gt;&gt;&gt; sum(t2) 60</code>
Membership operator <b>in</b>	<code>&gt;&gt;&gt; 'Friday' in t1 False</code>

Table 4: Summary of operations that can be applied on tuples

# Functions Tuple and Zip

- The function tuple can be used to convert a sequence to a tuple, for example:

```
>>> vowels = 'aeiou'  
>>> tuple(vowels)  
('a', 'e', 'i', 'o', 'u')
```

- The function zip is used to produces a zip object (iterable object), whose ith element is a tuple containing ith element from each iterable object passed as argument to the zip function.
- We have applied list function to convert the zip object to a list of tuples. For example,

```
>>> colors = ('red', 'yellow', 'orange')  
>>> fruits = ['cherry', 'banana', 'orange']  
>>> fruitColor = list(zip(colors, fruits))  
>>> fruitColor  
[('red', 'cherry'), ('yellow', 'banana'), ('orange', 'orange')]
```

# Functions count and index

- The function **count** is used to find the number of occurrences of a value in a tuple, for example:

```
>>> age = (20, 18, 17, 19, 18, 18)  
>>> age.count(18)
```

- The function **index** is used to find the index of the first occurrence of a particular element in a tuple, for example:

```
>>> age.index(18)  
1
```

Function	Explanation
T.count(e)	Returns count of occurrences of e in Tuple T
T.index(e)	Returns index of first occurrences of e in Tuple T

Table 5: Tuple Functions

# Dictionary

- Unlike lists, tuples, and strings, a **dictionary** is an unordered sequence of key-value pairs.
- Indices in a **dictionary** can be of any immutable type and are called keys.
- Beginning with an empty dictionary, we create a dictionary of month\_number-month\_name pairs as follows:

```
>>> month = {}  
>>> month[1] = 'Jan'  
>>> month[2] = 'Feb'  
>>> month[3] = 'Mar'  
>>> month[4] = 'Apr'  
>>> month  
{1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr'}  
>>> type(month)  
<class'dict'>
```

# Dictionary

```
>>> price = {'tomato':40, 'cucumber':30, 'potato':20,  
'cauliflower':70, 'cabbage':50, 'lettuce':40, 'raddish':30, 'carrot':20,  
'peas':80}  
>>> price['potato']  
20  
>>> price['carrot']  
20  
>>> price.keys()  
dict_keys(['tomato', 'cucumber', 'potato', 'cauliflower', 'cabbage',  
'lettuce', 'raddish', 'carrot', 'peas'])  
>>> price.values()  
dict_values([40, 30, 20, 70, 50, 40, 30, 20, 80])  
>>> price.items()  
dict_items([('tomato', 40), ('cucumber', 30), ('potato', 20),  
('cauliflower', 70), ('cabbage', 50), ('lettuce', 40), ('raddish', 30),  
('carrot', 20), ('peas', 80)])
```

# Dictionary

- The search in a dictionary is based on the key. Therefore, in a dictionary, the keys are required to be unique.
- As keys in a dictionary are immutable, lists cannot be used as keys.
- Keys in a dictionary may be of heterogeneous types, for example:  

```
>>> counting = {1:'one', 'one':1, 2:'two', 'two':2}
```

# Dictionary Operations

- some operations that can be applied to a dictionary and illustrate these operations using a dictionary of digit-name pairs:  
`digits = {0:'Zero', 1:'One', 2:'Two', 3:'Three', 4:'Four', 5:'Five',  
6:'Six', 7:'Seven', 8:'Eight', 9:'Nine'}`

Operation	Examples
Length operator <b>len</b> (number of key-value pairs in dictionary)	<code>&gt;&gt;&gt; len(digits)</code> 10
Indexing	<code>&gt;&gt;&gt; digits[1]</code> 'One'
Function <b>min</b>	<code>&gt;&gt;&gt; min(digits)</code> 0
Function <b>max</b>	<code>&gt;&gt;&gt; max(digits)</code> 9
Function <b>sum</b> (assuming keys are compatible for addition)	<code>&gt;&gt;&gt; sum(digits)</code> 45

# Dictionary Operations

Operation	Examples
Membership operator <b>in</b>	<pre>&gt;&gt;&gt; 5 in digits</pre> <pre>True</pre> <pre>&gt;&gt;&gt; 'Five' in digits</pre> <pre>False</pre>

Table 6: Summary of operations that can be applied on dictionaries

- Consider a dictionary named as winter:  

```
>>> winter = {11:'November', 12: 'December', 1:'January',  
2:'February'}
```
- Membership operation **in**, and functions **min**, **max** and **sum** apply only to the keys in a dictionary.

# Dictionary Operations

- Thus, applying these operations on the dictionary winter is equivalent to applying them on winter.keys() as shown below:  
    >>> 2 in winter, min(winter), max(winter), sum(winter)  
    (True, 1, 12, 26)  
    >>> 2 in winter.keys(), min(winter.keys()), max(winter.keys()),  
        sum(winter.keys())  
    (True, 1, 12, 26)
- We may remove a key-value pair from a dictionary using del operator, for example:  
    >>> del winter[11]  
    >>> winter  
    {12: 'December', 1: 'January', 2: 'February'}
- To delete a dictionary, we use del operator:  
    >>> del winter

# Dictionary Functions

Function	Explanation
D.items()	Returns an object comprising of tuples of <b>key-value</b> pairs present in dictionary D
D.keys()	Returns an object comprising of all keys of dictionary D
D.values()	Returns an object comprising of all values of dictionary D
D.clear()	Removes all <b>key-value</b> pairs from dictionary D
D.get(key, default)	For the specified <b>key</b> , the function returns the associated value. Returns the <b>default</b> value in the case <b>key</b> is not present in the dictionary D
D.copy()	Creates a shallow copy of dictionary D
D1.update(D2)	Adds the <b>key-value</b> pairs of dictionary D2 to dictionary D1

Table 7: Dictionary Functions

# Inverted Dictionary

- Suppose we are maintaining a dictionary of words and their meanings of the form **word:meaning**. For simplicity, we assume that each word has a single meaning.
- There might be a few words that may have shared meaning. Given this dictionary, we wish to find synonyms of a word, i.e. given a word, we wish to find the list of words that have the same meaning.
- For this purpose, we would like to build an inverted dictionary **invDict** of **meaning:list-of-words**.

```
>>> Enter word meaning dictionary: {'dubious':'doubtful',
    'hilarious':'amusing', 'suspicious':'doubtful', 'comical':'amusing',
    'hello':'hi'}
```

Inverted Dictionary:

```
{'doubtful': ['dubious', 'suspicious'], 'amusing': ['hilarious',
    'comical']}
```

# Inverted Dictionary Code

```
1 def buildInvDict(dict1):
2     """
3         Objective: To construct inverted dictionary
4         Input parameter: dict1: dictionary
5         Return value: invDict: dictionary
6     """
7     invDict = {}
8     for key,values in dict1.items():
9         if value in invDict:
10             invDict[value].append(key)
11         else:
12             invDict[value] = [key]
13     invDict={x:invDict[x] for x in invDict if len(invDict[x])>1}
14
15     return invDict
```

# Inverted Dictionary Code

```
1 def main():
2     """
3         Objective: To find inverted dictionary
4         Input parameters: None
5         Return value: None
6         """
7     wordMeaning = eval(input('Enter word meaning dictionary:'))
8     meaningWord = buildInvDict(wordMeaning)
9     print('Inverted dictionary:\n', meaningWord)
10
11 # Statements to initiate the call to main function.
12
13 if __name__=='__main__':
14     main()
```

# References



Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You  
Any Questions?

# Recursion

## Lecture 8

Department of Computer Science and engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India



# Table Of Contents:

1

Introduction

2

Recursion on Numbers

3

Recursion on Strings

4

Recursion on Lists

5

Shallow copy and Deep copy

6

Conclusion

7

References

# Introduction

- Recursion is the process of describing the computation in a function in terms of function itself.
- Recursive function comprises of two parts:
  - ① Base part:- The solution of the simplest version of the problem, often termed stopping or termination criterion.
  - ② Inductive part:- The recursive part of the problem that reduces the complexity of the problem by redefining the problem in terms of simpler version(s) of the original problem itself.

## Recursion on Numbers(Example I)

- Recursion can be used to find the factorial of a given number
- Following algorithm can be used:

$$n! = \begin{cases} 1 & \text{if } n == 0 \text{ or } n == 1 \text{ (Base part)} \\ n * (n - 1)! & \text{if } n > 1 \text{ (Inductive part)} \end{cases}$$

# Recursion on Numbers(Example I Continued)

Following program finds the factorial of a given number:

```
def factorial(n):
    """
    Objective: To compute factorial of a positive integer
    Input parameter: n—numeric value
    Return value: factorial of n— numeric value
    """

    assert n>= 0
    if n==0 or n==1:
        return 1
    else:
        return n*factorial(n-1)

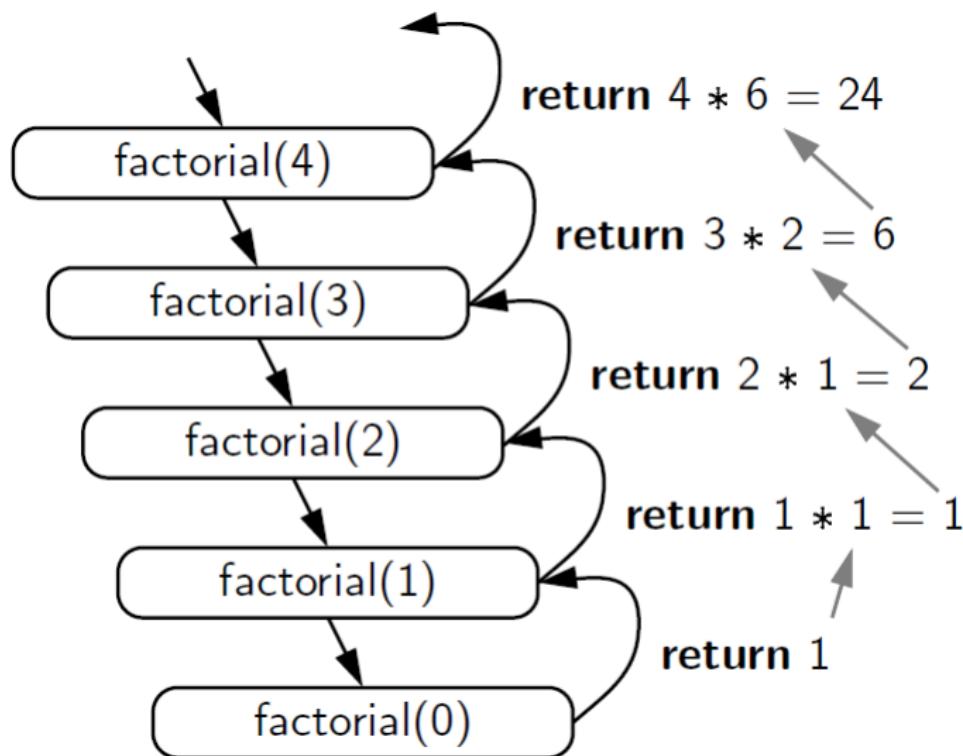
def main():
    """
    Objective: To compute factorial of a number provided by the user
    Input parameter: None
    Return value: None
    """

    n = int(input('Enter the number: '))
    result = factorial(n)
    print('Factorial of ',n,' is ',result)

if __name__=='__main__':
    main()
```

## Recursion on Numbers(Example I Continued)

The following is the recursive tree to get the factorial of positive integer 4:



## Recursion on Numbers(Example II)

- Sum of the first n natural numbers can be found recursively.
- Following algorithm can be used:

$$sum(n) = \begin{cases} 1 & \text{if } n == 1 \text{ (Base part)} \\ n + sum(n - 1) & \text{if } n > 1 \text{ (Inductive part)} \end{cases}$$

## Recursion on Numbers(Example II Continued)

Following program finds the sum of n positive integers number:

```
def sum(n):
    if n==1:
        return 1
    else:
        return n+sum(n-1)

def main():
    n = int(input('Enter the number: '))
    print('Sum of first n natural numbers is ', sum(n))

if __name__=='__main__':
    main()
```

## Recursion on Numbers(Example III)

- Recursion can be used to find the reverse of a given number
- Following algorithm can be used:

`rev = 0`

$$\text{reverse}(n, \text{rev}) = \begin{cases} 0 & \text{if } n == 0 \text{ (Base part)} \\ \text{reverse}(n//10, \text{rev} * 10 + r) & \text{if } n > 1 \text{ (Inductive part)} \\ \text{where, } r = n \% 10 \end{cases}$$

## Recursion on Numbers(Example III Continued)

Following program finds the reverse of a given number:

```
def reverse(n, rev):
    if n==0:
        return rev
    else:
        r = n%10
        rev = rev*10+r
        return reverse(n//10, rev)

def main():
    rev = 0
    n = int(input('Enter the number: '))
    print('reverse of ',n,' is ',reverse(n,rev))

if __name__=='__main__':
    main()
```

## Recursion on Numbers(Example IV)

- GCD of two given numbers x and y can be obtained using recursion
- Following algorithm can be used:

$$GCD(x, y) = \begin{cases} x & \text{if } y == 0 \text{ (Base part)} \\ GCD(y, x \% y) & \text{otherwise (Inductive part).} \end{cases}$$

## Recursion on Numbers(Example IV Continued)

Following program finds the GCD of two given numbers x and y:

```
def GCD(x,y):  
    if y==0:  
        return x  
    else:  
        r = x%y  
        x = y  
        y = r  
        return GCD(x,y)  
  
def main():  
    x = int(input('Enter x:'))  
    y = int(input('Enter y:'))  
    print('GCD of ',x,' and ',y,' is ',GCD(x,y))  
  
if __name__=='__main__':  
    main()
```

## Recursion on Numbers(Example V)

- nth term of the Fibonacci series can be obtained using binary recursion.
- Following algorithm can be used:

$$fib(n) = \begin{cases} 0 & \text{if } n == 1 \text{ (Base part)} \\ 1 & \text{if } n == 2 \text{ (Base part)} \\ fib(n - 1) + fib(n - 2) & \text{if } n > 2 \text{ (Inductive part).} \end{cases}$$

## Recursion on Numbers(Example V Continued)

Following program gives the nth term of the Fibonacci sequence:

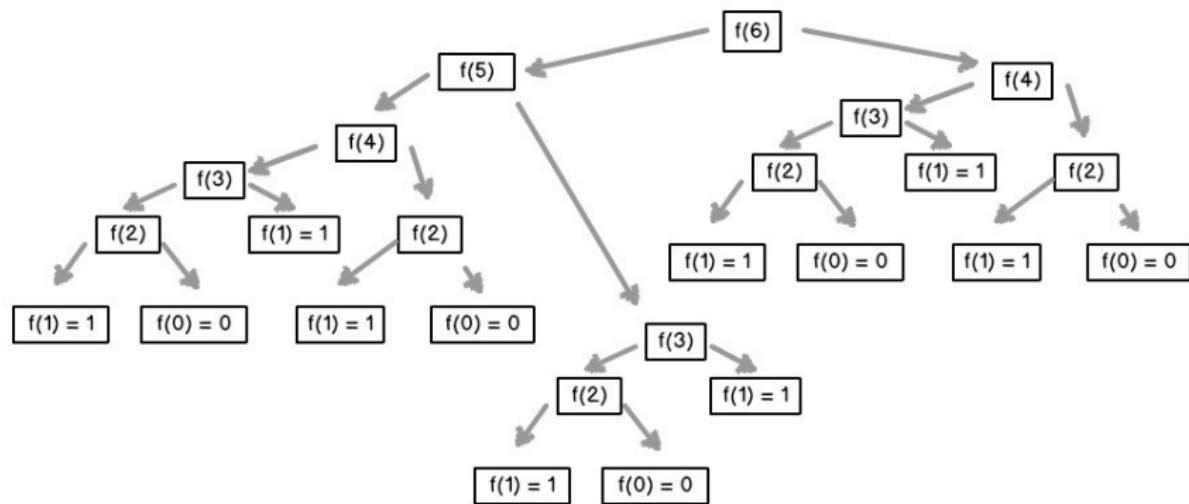
```
def fib(n):
    if n==1:
        return 0
    elif n==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)

def main():
    n = int(input('Enter a number: '))
    print('The fibonacci number at place',n,'is',fib(n))

if __name__=='__main__':
    main()
```

## Recursion on Numbers(Example V Continued)

The following is the recursive tree of the Fibonacci sequence:



## Recursion on Strings(Example I)

- Recursion can be used to find the length of a string
- Following algorithm can be used:

$$\text{length}(\text{str1}) = \begin{cases} 0 & \text{if } \text{str} == "" \text{ (Base part)} \\ 1 + \text{length}(\text{str1}[1:]) & \text{otherwise (Inductive part).} \end{cases}$$

## Recursion on Strings(Example I continued)

Following program gives the length of a given string:

```
def length(str1):  
    '''
```

*Objective: To determine the length of input string*

*Input parameter: str1 – string*

*Return value: numeric*

```
'''
```

```
if str1 == '':  
    return 0  
else:  
    return 1 + length(str1[1:])
```

```
def main():  
    '''
```

*Objective: To determine length of string*

*Input parameter: None*

*Return value: None*

```
'''
```

```
str1 = input('Enter the string: ')  
result = length(str1)  
print('Length of string', str1, 'is', result)
```

```
if __name__ == '__main__':  
    main()
```

## Recursion on Strings(Example II)

- Reverse of a string can be found recursively
- Following algorithm can be used:

$$\text{reverse}(\text{str1}) = \begin{cases} 0 & \text{if } \text{str} == "" \text{ (Base part)} \\ \text{str1}[-1] + \text{reverse}(\text{str}[:-1]) & \text{otherwise (Inductive part).} \end{cases}$$

## Recursion on Strings(Example II conitnued)

Following program finds the reverse of a given string:

```
def reverse(str1):
    if str1=='':
        return str1
    else:
        return str1[-1]+reverse(str1[:-1])

def main():
    str1 = input('Enter a string: ')
    print('Reverse of ',str1,' is ',reverse(str1))

if __name__=='__main__':
    main()
```

## Recursion on Strings(Example III)

- Recursion can be used to check whether a given string is a palindrome or not
- Following algorithm can be used:

$$\text{isPalindrome}(\text{str1}) = \begin{cases} \text{True} & \text{if } \text{str} == \text{" (Base part)} \\ \text{False} & \text{if } \text{str1}[0] != \text{str1}[-1] \text{ " (Base part)} \\ \text{isPalindrome}(\text{str1}[1:-1]) & \text{otherwise (Inductive part).} \end{cases}$$

## Recursion on Strings(Example III continued)

Following program checks whether a given string is a palindrome or not

```
def isPalindrome(str1):
    if str1 == '':
        return True
    else:
        return (str1[0]==str1[-1] and isPalindrome(str1[1:-1]))

def main():
    str1 = input('Enter the string : ')

    if isPalindrome(str1):
        print('String is a palindrome.')
    else:
        print('String is not a palindrome.')

if __name__=='__main__':
    main()
```

## Recursion on Lists(Example I)

- Recursion can be used to find the sum of elements of a list.
- Following algorithm can be used:

$$\text{sumls}(\text{ls}, n) = \begin{cases} 0 & \text{if } n == 0 \text{ (Base part)} \\ \text{ls}[n-1] + \text{sumls}(\text{ls}, n-1) & \text{otherwise (Inductive part).} \end{cases}$$

## Recursion on Lists(Example I continued)

Following program finds the sum of the elements of the given list:

```
def sumls(ls ,n ):  
    if n==0:  
        return 0  
    else:  
        return ls [n-1]+sumls(ls ,n-1)  
  
def main ():  
    ls = eval(input('Enter list elements: '))  
    n = len(ls )  
    print('Sum of elements of list is ',sumls(ls ,n ))  
  
if __name__=='__main__':  
    main()
```

## Recursion on Lists(Example II)

- Given a list of lists, the nesting of lists may occur up to any arbitrary level.
- Flattening of a list means to create a list of all data values in the given list.
- The data values in the flattened list appear in the left to right order of their appearance in the original list, ignoring the nested structure of the list.
- Thus the list  $[1,[2,[3,4]]]$  can be flattened to obtain  $[1,2,3,4]$ .
- Following algorithm can be used:  
For every element in list 1  
if i is not a list  
append it to list 2  
otherwise flatten the list i

## Recursion on Lists(Example II continued)

Following program finds the reverse:

```
def flatten(ls1, ls2 = []):
    for element in ls1:
        if type(element) != list:
            ls2.append(element)
        else:
            flatten(element, ls2)
    return ls2

def main():
    ls1 = eval(input('Enter the list: '))
    result = flatten(ls1)
    print('Flattened List:', result)

if __name__ == '__main__':
    main()
```

## Shallow copy and Deep copy

- Simply assigning a list object to another name does not create a copy of the list; instead, both the names refer to the same list object.
- So when a change is made in any of the list it appears in both the lists.

# Shallow copy and Deep copy

## Shallow copy

- A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original.
- The copying process does not recurse and therefore won't create copies of the child objects themselves
- It means that any changes made to a copy of object do reflect in the original object.

# Shallow copy and Deep copy

```
## Making shallow copy
```

```
import copy
```

```
ls1 = [[1,2,3],[4,5,6],[7,8,9]]
```

```
ls2 = copy.copy(ls1)
```

```
print('ls1=' , ls1)
```

```
print('ls2=' , ls2)
```

```
ls2[0] = [0,0,0]
```

```
print('ls1=' , ls1)
```

```
print('ls2=' , ls2)
```

```
ls2[1][2] = 0
```

```
print('ls1=' , ls1)
```

```
print('ls2=' , ls2)
```

# Shallow copy and Deep copy

## Deep copy

- A deep copy means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original
- In case of deep copy, a copy of object is copied in other object.
- It means that any changes made to a copy of object do not reflect in the original object.

# Shallow copy and Deep copy

### Making deep copy

```
import copy
```

```
ls1 = [[1,2,3],[4,5,6],[7,8,9]]
```

```
ls2 = copy.deepcopy(ls1)
```

```
print('ls1=' ,ls1)
```

```
print('ls2=' ,ls2)
```

```
ls2[0] = [0,0,0]
```

```
print('ls1=' ,ls1)
```

```
print('ls2=' ,ls2)
```

```
ls2[1][2] = 0
```

```
print('ls1=' ,ls1)
```

```
print('ls2=' ,ls2)
```

# Conclusion

- Python allows a function to call itself, possibly with new parameter values. This technique is called recursion.
- Recursion allows us to break a large task down to smaller tasks by repeatedly calling itself.
- Every recursive function must have at least two cases: the inductive case and the base case.
- A recursive function requires a base case to stop execution, and the call to itself which gradually leads to the function to the base case.
- The inductive case is the more general case of the problem we are trying to solve.
- Recursion is not hard to implement in the right circumstances. It's important to make sure that the algorithm terminates, especially in cases where data corruption may occur.

# References

- [1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, Pearson Education India, Inc., 2017.

# FILES AND EXCEPTIONS

## Lecture 9

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Introduction
- 2 File Handling
- 3 Writing Structures to a File
- 4 Errors and Exceptions
- 5 Handling Exceptions using try... except
- 6 File Processing Example

# Introduction

- Programs that we have developed so far take data from the user in an interactive manner.
- Such data remain in memory only during the lifetime of the program.
- Often we want to store data permanently, in the form of files that usually reside on disks, so that it is available as and when required.
- By default, Python provides a standard input file and a standard output file to deal with the transitory data.

## Introduction (Cont.)

- The standard input file is read from the keyboard, and the standard output file is displayed on the screen.
- Apart from these standard input/output files, we can also create files on disks that store data permanently for subsequent use.
- Files provides a means to store data permanently.
- A file is a stream of bytes, comprising data of interest.
- Before performing a read or write operation in a file, we need to open the file.

# File Handling

- Built-in function **open()** is used to open a file.
- This function **open()** takes the name of the file as the first argument and the mode for accessing the file as The second argument.
- A file may be opened in any of the three modes: r (read), w (write), and a (append).
- Read mode is used when an existing file is to be read.
- Write mode is used when a file is to be accessed for writing data in it.

## File Handling (Cont.)

- append mode allows us to write into a file by appending contents at the end of the specified file.
- While opening a file in the read mode, if the specified file does not exist, it would lead to an error.
- while opening a file in write mode, if the specified file does not exist, Python will create a new file.
- while opening a file in write mode, if the specified file already exists, this file gets overwritten
- while opening a file in append mode, if the specified file does not exist, a new file is created.

## File Handling (Cont.)

- The absence of the second argument in the open function sets it to default value 'r' (read mode).
- A file may be opened using the following syntax  
 $f = \text{open}(\text{file\_name}, \text{access\_mode})$   
`>>> f = open( 'PYTHON', 'w' )`
- By default, the system creates a file PYTHON in the default working directory.
- we use write() function to write some text in the file PYTHON  
`>>> f.write('failure is a part of success')`

## File Handling (Cont.)

- Since the file is opened in write mode, Python disallows read operation on it by flagging an error as shown below:

```
>>> f.read()
```

Traceback (most recent call last):

```
  File "<pyshell#2>", line 1, in <module>
      f.read()
```

```
io.UnsupportedOperation: not readable
```

- When a file is no longer required for reading or writing, it should be closed by invoking the function close as shown below:

```
>>> f.close()
```

## File Handling (Cont.)

- The function close also saves a file, which was opened for writing.
- To read the contents of the file f, we open the file again, but this time in read mode

```
>>> f = open('PYTHON', 'r')
>>> f.read()          OR    >>> print(f.read())
>>> f.close()
```

- The f.read() reads the entire contents of the file but The f.read(4) reads only 4 bytes of contents of the file.
- Consider following programs (a) and (b)

# File Handling (Cont.)

(a). `f=open('PYTHON','w')`

```
f.write('failure is a part of success')
```

```
f = open('PYTHON', 'r')
```

```
print(f.read(4))
```

```
f.close()
```

(b). `f=open('PYTHON','w')`

```
f.write('failure is a part of success')
```

```
f = open('PYTHON', 'r')
```

```
print(f.read())
```

```
f.close()
```

- The output of (a) is **fail** and output of program (b) is **failure is a part of success**

## File Handling (Cont.)

- We use the function tell() to know the current position while reading the file object f.

```
>>> f.tell()
```

- we use function readline() to read one line at a time from the file.

```
>>> f.readline()
```

- Note that the readline function reads a stream of bytes beginning the current position until a newline character is encountered.

- We use function seek() to reach desired position in a file

```
>>> f.seek(0)
```

0

- The function seek returns the new absolute position.

## File Handling (Cont.)

- The function `readlines()` reads and returns all the remaining lines of the file in the form of a list.

```
>>> f.readlines()
```

- Just like the `readlines` function discussed above, the function `writelines` takes a list of lines to be written in the file as an argument

```
>>> f.writelines()
```

## File Handling (Cont.)

- The output of following program is  
**we either choose the pain of discipline**  
**or**  
**the pain of regret**

```
f = open('PYTHON', 'w')
description =[ 'we either choose the pain of discipline \n', 'or\n'
'the pain of regret\n']
f.writelines(description)
f.close()
f = open('PYTHON', 'r')
print(f.read())
f.close()
```

## File Handling (Cont.)

- Suppose if we wish to copy the contents of a text file, say, PYTHON in another file PythonCopy. For this purpose, we open the source file PYTHON in read mode and the output file PythonCopy (yet to be created) in write mode, read the data from the file PYTHON, and write it into the file PythonCopy as follows:

```
>>> f1 = open('PYTHON', 'r')
>>> f2 = open('PythonCopy', 'w')
>>> data = f1.read()
>>> f2.write(data)
>>> f1.close()
>>> f2.close()
```

## File Handling (Cont.)

- Note that if an application requires the file to be opened in both read and write mode, 'r+' mode can be used while opening it.
- If we wish to put the new content at the end of previously existing contents in a file, we need to open the file in append ('a') mode as shown below:

```
>>> f = open('PYTHON', 'a')
>>> f.write('simplicity is the ultimate sophistication')
>>> f.write('You think you can or you can't, you are right')
>>> f.close()
>>> f = open('PYTHON', 'r')
>>> f.read()
>>> f.close()
```

## File Handling (Cont.)

- To ensure that the contents written to a file have been saved on the disk, it must be closed. While we are still working on a file, its contents may be saved anytime using the flush function:

```
>>> f=open('file2','w')  
>>> f.write("Believe in yourself")  
>>> f.flush()
```

# Writing Structures to a File

- To write a structure such as a list or a dictionary to a file and read it subsequently, we use the Python module pickle as follows:

```
import pickle  
def main():  
    ""
```

Objective: To write and read a list and a dictionary to and from a file

Input Parameter: None

Return Value: None

```
""
```

```
f=open('file3','wb')  
pickle.dump(['hello','world'],f)  
pickle.dump(1:'one', 2:'two',f)  
f.close()
```

## Writing Structures to a File (cont.)

- ```
f=open('file3','rb')
value1=pickle.load(f)
value2=pickle.load(f)
print(value1,value2)
f.close()
if __name__=='__main__':
    main()
```
- the output of above program is:  
['hello', 'world'] {1: 'one', 2: 'two'}
- dump(): to convert a structure to byte stream and write it to a file
- load(): to read a byte stream from a file and convert it back to the original structure

# Errors and Exceptions

- Errors occur when something goes wrong.
- The errors in Python programming may be categorized as syntax errors and exceptions.
- A syntax error occurs when a rule of Python grammar is violated.
- some Example of syntax error :

```
>>> print('Hello)  
>>> for i in range(0, 10)  
indentation error
```

- In contrast to a syntax error, an exception occurs because of some mistake in the program that the system cannot detect before executing the code as there is nothing wrong in terms of the syntax, but leads to a situation during the program execution that the system cannot handle.

# Errors and Exceptions(Cont.)

- Some common Exceptions:

## 1. Name error

```
>>> marks = Input('Enter your marks')
>>> print(price)
```

## 2. TypeError

```
>>> 'sum of 2 and 3 is ' + 5
```

## 3. ValueError

```
>>> int('Hello')
```

## 4. ZeroDivisionError

```
>>> 78/(2+3-5)
```

## 5. OSError

```
>>> f = open('passwordFile.txt')
```

## 6. IndexError

```
>>> colors = ['red', 'green', 'blue']
>>> colors[4]
>>> colors[2][4]
```

## Errors and Exceptions(Cont.)

- We have noticed that whenever an exception occurs, a Traceback object is displayed which includes error name, its description, and the point of occurrence of the error such as line number.
- Remember:
  - syntax error: violation of Python grammar rule
  - exceptions: errors that occur at execution time

# Handling Exceptions using try...except

- Exceptions that would result in abrupt termination of program execution are known as unhandled exceptions.
- To prevent a program from terminating abruptly when an exception is raised, We use the try...except clause.
- try block comprises statements that have the potential to raise an exception
- except block describes the action to be taken when an exception is raised.
- We can also specify a finally block in the try...except clause, which is executed irrespective of whether an exception is raised.
- try block: statements having potential to raise an exception
- except block: action to be performed when exception is raised
- finally block: executed irrespective of whether an exception is raised

# Errors and Exceptions(Cont.)

- def main():

""

Objective: To illustrate the use of raise and finally clauses

Input Parameter: None

Return Value : None

""

marks=110

try:

    if marks< 0 or marks> 100:

        raise ValueError('Marks out of range')

except:

    pass

finally:

    print('bye')

    print('program continues after handling exception')

if \_\_name\_\_ == '\_\_main\_\_':

    main()

## Errors and Exceptions(Cont.)

- The OUTPUT of above Program is:  
bye  
program continues after handling exception

# Errors and Exceptions(Cont.)

- def main():  
    ""  
        Objective: To open a file for reading  
        Input Parameter : None  
        Return Value: None  
    ""  
        try:  
            f=open('Temporary\_file', 'r')  
        except IOError:  
            print('Problem with Input Output.....')  
            print('Program continues smoothly beyond try...except block')  
        if \_\_name\_\_ == '\_\_main\_\_':  
            main()

## Errors and Exceptions(Cont.)

- The OUTPUT of above Program is:  
Problem with Input Output.....  
Program continues smoothly beyond try...except block

## Errors and Exceptions(Cont.)

- import sys  
def main():  
    ""

Objective: To compute price per unit weight of an item

Input Parameter : None

Return Value: None

""

```
price=input('enter price of item purchased: ')
weight=input('Enter weight of item purchased: ')
try:
```

```
    if price==": price=None
```

```
    try:
```

```
        price=float(price)
```

```
    except ValueError:
```

```
        print('Invalid inputs: ValueError')
```

```
    if weight==": weight=None
```

```
    try:
```

# Errors and Exceptions(Cont.)

- try:  
    weight=float(weight)  
except ValueError:  
    print('Invalid inputs: ValueError')  
assert price>= 0 and weight>= 0  
result=price/weight  
except TypeError:  
    print('Invalid inputs: ValueError')  
except ZeroDivisionError:  
    print('Invalid inputs: ZeroDivisionError')  
except:  
    print(str(sys.exc\_info()))  
else:  
    print('Price per unit weight: ', result)  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

## Errors and Exceptions(Cont.)

- The OUTPUT of above Program is:
- enter price of item purchased: 20  
Enter weight of item purchased: 0  
Invalid inputs: ZeroDivisionError
- enter price of item purchased: -20  
Enter weight of item purchased: 0  
(*< class'AssertionError' >, AssertionError(), < tracebackobject at 0x000002B6FA61E400 >*)
- enter price of item purchased: -20  
Enter weight of item purchased: 10  
(*< class'AssertionError' >, AssertionError(), < tracebackobject at 0x000002B6FA60CA80 >*)
- enter price of item purchased: 20  
Enter weight of item purchased:  
Invalid inputs: ValueError

## File Processing Example

- We are given a file named **studentMarks**. This file contains the student data that includes roll number (rollNo), name (name), and marks (marks) for each student. The data about each student is stored in a separate line and the individual pieces of information rollNo, name, and marks are separated by commas. As shown below:

4001,Nitin Negi,75

4002,Kishalaya Sen,98

4003,Kunal Dua,80

4004,Prashant Sharma,60

4005,Saurav Sharma,88

- We define addPerCent as the percentage of maxMarks that should be added to the marks obtained to get the moderated marks, subject to the upper limit of maxMarks. To carry out the moderation, we prompt the user to enter the moderation percentage (addPerCent) and produce another file **moderatedMarks** containing moderated marks of the students.



# File Processing Example

- We describe this task in the form of a pseudocode:
  1. Open file studentMarks in read mode while checking for any errors.
  2. Open file moderateMarks in write mode while checking for any error.
  3. Read one line of input(line1) from studentMarks.
  4. while(line != ""):
    - Retrieve the values of Roll No., name, and marks from the line1 while checking for any errors.
    - Compute moderated marks and write one line of output in the file moderateMarks
    - Read one line of input(line1) from studentMarks.
- The complete script is given below:

# File Processing Example

- The complete script is given below:
- When we execute the following program and enter 3 as value of addPercent, the system outputs the contents of the file moderatedMarks as follows:

4001,Nitin Negi,78.0

4002,Kishalaya Sen,100

4003,Kunal Dua,83.0

4004,Prashant Sharma,63.0

4005,Saurav Sharma,91.0

# File Processing Example

```
>>> import sys
>>> def computemoderatemarks(file1,file2,addpercent):
>>>     try:
>>>         fin=open(file1,'r')
>>>         fout=open(file2,'w')
>>>     except IOError:
>>>         print('problem in opening the file');sys.exit()
>>>     line1=fin.readline()
>>>     while(line1!=""):
>>>         slist=line1.split(',')
>>>         try:
>>>             rollno=int(slist[0])
>>>             name=slist[1]
>>>             marks=int(slist[2])
>>>         except IndexError:
>>>             print('undefined index');sys.exit()
```

# File Processing Example

```
>>>         except (ValueError):
>>>             print('Unsuccessful conversion to int');sys.exit()

>>>         maxmarks=100
>>>
moderemarks=marks+((addpercent*maxmarks)/100)
>>>         if moderemarks>100:
>>>             moderemarks=100
>>>
fout.write(str(rollno)+','+name+','+str(moderemarks)+'\n')
>>>         line1=fin.readline()
>>>         fin.close()
>>>         fout.close()
```

# File Processing Example

```
>>> def main():
>>>     import sys
>>>     sys.path.append('F:\pythoncode \ch9')
>>>     file1=input('enter name of file containing marks: ')
>>>     file2=input('enter output file for moderated marks: ')
>>>     addpercent=int(input('enter moderation percentage: '))
>>>     computemoderatemarks(file1,file2,addpercent)
>>> if __name__ == '__main__':
>>>     main()
```

## File Processing Example)

In our next program, we wish to compute monthly wages to be paid to employees in an organization. The input data has been provided to us in two files named empMaster and empMonthly. The first file empMaster contains permanent data about employees (also called master data) that include employee id (emplID), employee name (empName), and hourly wages (hrlyWages) as follows:

1001,Vinay Kumar,30

1002,Rohit Sen,35

1003,Vinita Sharma,28

1004,Bijoy Dutta,35

## File Processing Example)

The second file empMonthly contains monthly information (often called transaction data) about employees. It stores two pieces of information about each employee, namely, employee id (tEmpID) and the number of hours worked (hrsWorked) as follows:

1001,245

1002,0

1003,0

1004,240 The pseudocode is given below

## File Processing Example)

1. Open files **empMaster** and **empmonthly** in read mode while checking for any errors.
2. Open file **monthlyWages** in write mode while checking for any errors.
3. Read one line of input(line1) from **empMaster**
4. while (line1 != ""):
  - (a) Retrieve value of empID and hrlyWages from the line1 while checking for any errors.
  - (b) Read one line of inputs(line2) from enpMonthly.
  - (c) Retrieve the value of tEmpID and hrsWorked from line2 while checking for any errors.
  - (c) check that empID in empMaster and tEmpID in the empMonthly match
  - (d) compute monthly wages and write one line of output in file **monthlyWages**
  - (d) Read one line of input(line1) from empMaster.

## File Processing Example)

The complete script is given below On the execution of the following program program, the system creates an output file monthlyWages with the following content:

1001,7350

1002,0

1003,0

1004,8400

## File Processing Example)

```
>>> import sys
>>> def generatesalary(file1,file2,file3):
>>>     try:
>>>         fmaster=open(file1,'r')
>>>         ftrans=open(file2,'r')
>>>         fwages=open(file3,'w')
>>>     except IOError:
>>>         print('problem with opening the file');sys.exit()
>>>     line1=fmaster.readline()
>>>     while((line1)!=""):
>>>         slist1=line1.split(',')
```

## File Processing Example)

```
>>>     try:  
>>>         empID=int(slist[0])  
>>>         hrlywages=int(slist1[2])  
>>>     except IndexError:  
>>>         print('undefined index');sys.exit()  
>>>     except (ValueError, TypeError):  
>>>         print('unsuccesssful convertion to int'); sys.exit()  
>>>     line2=fTrans.readlines()  
>>>     sList2=line2.split(',')
```

## File Processing Example)

```
>>>     try:  
>>>         tEmpid=int(slist2[0])  
>>>         hrsworked=int(slist2[1])  
>>>     except IndexError:  
>>>         print('undefined index');sys.exit()  
>>>     except (ValueError,TypeError):  
>>>         print('unsuccessful conversion to int');sys.exit()  
>>>     if emplId==tEmplId:  
>>>         fwages.write(str(emplId)+','+  
        .           str(hrlywages*hrsworked)+'\n')  
>>>     line1=fmaster.readline()  
>>> fmaster.close()  
>>> ftrans.close()  
>>> fwages.close()
```

## File Processing Example)

```
>>> def main():
>>>     import sys
>>>     sys.path.append('F:09')
>>>     file1=input('enter name of file containing hourly rate: ')
>>>     file2=input('enter name of file containing hours worked: ')
>>>     file3=input('enter output file for salary generation : ')
>>>     generatesalary(file1,file2,file3)
>>> if __name__ == '__main__':
>>>     main()
```

# Conclusion

- A file is a stream of bytes, comprising data of interest. Built-in function `open()` is used for opening a file. It returns a file object. This function takes the name of the file as the first argument. The second argument indicates mode for accessing the file.
- A file may be opened in any of three modes: `r` (read), `w` (write), and `a` (append). Read mode is used when an existing file is to be read. Write mode is used when a file is to be accessed for writing data in it. Append mode allows one to write into a file by appending contents at the end. If the specified file does not exist, a file is created.
- The absence of the second argument in `open` function sets it to default value '`r`' (read mode).

# Conclusion

- When an existing file is opened in write mode, previous contents of the file get erased.
- The functions `read` and `write` are used for reading from and writing into the file, respectively. To use `read/write` function, we use the following notation: name of the file object, followed by the dot operator (`.`), and followed by the name of the function.
- Function `close` is used for closing the file. The function `close` also saves a file, which was opened for writing. Once a file is closed, it cannot be read or written any further unless it is opened again and an attempt to access the file results in an I/O (input/output) error.
- Function `tell` yields current position in the file.

# Conclusion

- The readline function reads a stream of bytes beginning the current position until a newline character is encountered. Read operation on a file whose all the contents have been read will return a null string.
- Function seek() is used for accessing the file from a particular position.
- Function readlines() returns all the remaining lines of the file in the form of a list of strings. Each string terminates with a newline character.
- Function writelines takes a list of lines to be written to the file as an argument.

# Conclusion

- Pickling refers to the process of converting a structure to a byte stream before writing to the file. The reverse process of converting a stream of bytes into a structure is known as unpickling.
- Python module pickle is used for writing an object such as list or dictionary to a file and reading it subsequently. Function dump of the pickle module performs pickling. Function load of the pickle module performs unpickling.
- A syntax error occurs when a rule of Python grammar is violated.

# Conclusion

- The exception occurs because of some mistake in the program that the system cannot detect before executing the code as there is nothing wrong in terms of the syntax, but leads to a situation during the program execution that the system cannot handle.
- These errors disrupt the flow of the program at a run-time by terminating the execution at the point of occurrence of the error.
- NameError exception occurs whenever a name specified in the statement is not found globally.
- TypeError exception occurs when an operation in an expression is incompatible with the type of operands.
- ValueError exception occurs whenever an invalid argument is used in a function call.
- ZeroDivisionError exception occurs when we try to perform numeric division in which denominator happens to be zero.
- IOError exception occurs whenever there is any error related to input or output.

# Conclusion

- IndexError exception occurs whenever we try to access an index out of the valid range.
- try...except clause is used for handling the exception. Whereas a try block comprises statements, which have the potential to raise an exception, except block describes the action to be taken when exception is raised. In the except clause, we may specify a list of exceptions and the action to be taken on occurrence of each exception.
- finally block is associated with try...except clause and is executed irrespective of whether an exception is raised.
- Details of the exception raised by Python can be accessed from the object: sys.exc\_info().

# References



Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

# Any question ?

## Any question ?

# Lecture:10 Classes I

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Classes I
- 2 Classes and Objects
- 3 Person-An example of class
  - Visualisation
  - Constructor and Destructor
  - Destructor
- 4 Class as Abstract Datatype
- 5 Date Class

# CLASSES I

- Class is a template that provides logical grouping of data and methods that operate on them.
- In OOP we define/ create objects( instances of class).
- Objects used are derived using classes.
- Related code and data are packaged together using classes.
- We can have user defined classes and there are in-built classes also.
- Data and attributes with a class are collectively known as class attributes.
- We can assign instances to objects of class.

# Classes and Objects

- We have various in-built class in python like str, int, float etc.
- 'Raman','CBSE' are instances of class str that are assigned to variable name and board.
- Various methods of class can act on objects of class.
- Method lower() can be used on object to return all the lowercase letters of string.
- methods are invoked by the object name followed by dot and the attribute name.
- Also, we can invoke class methods by class name followed by dot operator, method name and object name passed as an argument for method.

# Person-An example of class

## Syntax of defining class

```
class ClassName:  
    classbody
```

```
class Person:  
    count=0  
  
def __init__(self, name, DOB, address):  
    self.name=name  
    self.DOB=DOB  
    self.address=address  
    Person.count+=1
```

## About Person

```
def getName(self):  
    return self.name
```

```
def getDOB(self):  
    return self.DOB
```

```
def getAddress(self):  
    return self.address
```

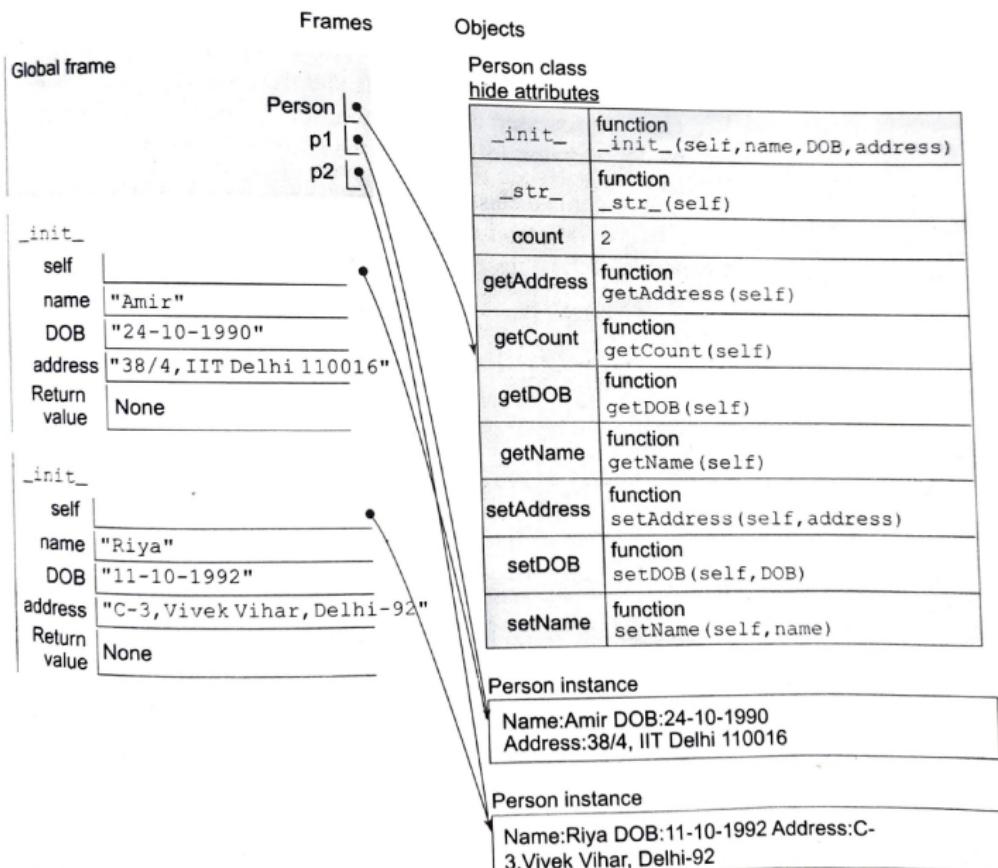
```
def setName(self,name):  
    self.name=name
```

```
def setDOB(self,DOB):  
    self.DOB=DOB
```

# Create Class

```
def setAddress(self,address):  
    self.address=address  
  
def getCount(self):  
    return Person.count  
  
def __str__(self):  
    return 'Name:' +self.name+':'+str(self.DOB)+':'+self.address
```

# Visualisation



# Constructor

- Constructors are generally used for instantiating an object.
- In Python the `__init__()` method is called the constructor and is always called when an object is created.
- The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created.

# Destructor

## Default Constructor

- The default constructor is a simple constructor which doesn't accept any arguments.
- Its definition has only one argument which is a reference to the instance being constructed.

## Parameterized constructor

- Constructor with parameters is known as parameterized constructor.
- The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

# Destructor

- We use **del** statement to remove no more required objects.
- Execution of del *reduces the reference count by one*.
- `__del__` method is invoked when count is zero.
- `del` is actually called the destructor because everytime class object reduces by one and so we reduce the count by one.

# Destructor Cont.

```
def __del__(self):  
    """
```

*Objective: To be invoked on deletion  
of an instance of the class person.*

*Input parameter:*

*self(implicit parameter)-object of  
type Person*

*Return Value:None*

```
"""
```

```
print('Deleted_!!')
```

```
Person.count-=1
```

# Class as Abstract Datatype

- Python introduces an abstract data type i.e., not having any concrete existence.
- Objects of a class can be created, modified and destroyed.
- Class defined can be saved as independent module that can be imported for use in any module.
- Class attributes can also be imported after changing the directory using sys module.

## Cont.

```
import sys
import os
sys.path.append('F:\Pythoncode\Ch10')
from person import person
def main():
    print('*****dir(Person):\n', dir(Person))
    print('*****Person.__doc__:\n', Person.__doc__)
    print('*****Person.__module__:\n', Person.__module__)

p1=Person('Amir','24-10-1991','38/4,IIT Delhi 110016')
print('*****Person.count:\n', Person.count)
print('*****p1.getcount():\n', p1.getcount())
print('*****p1.__doc__:\n', p1.__doc__)
print('*****p1.__module__:\n', p1.__module__)
print('*****p1:\n', p1)
print('*****dir(p1):\n', dir(p1))
```

# Cont.

```
p2=Person('Riya','11-10-1992','C-3,『Vivek Vihar,『Delhi-92」)
print('*****Person.count:\n',Person.count)
print('*****p2.getcount():\n',p2.getcount())
print('*****p1.getcount():\n',p1.getcount())
print('*****p2.__doc__:\n',p2.__doc__)
print('*****p2.__module__:\n',p2.__module__)
print('*****p2:\n',p2)
print('*****dir(p2):\n',dir(p2))

print('\n*****_id:Person.__doc__:\n',id(Person.__doc__))
print('*****_id:Person.__module__:\n',id(Person.__module__))
print('*****_id:p1.__doc__:\n',id(p1.__doc__))
print('*****_id:p1.__module__:\n',id(p1.__module__))
print('*****_id:p2.__doc__:\n',id(p2.__doc__))
print('*****_id:p2.__module__:\n',id(p2.__module__))
print('*****_id:dir(Person):\n',id(dir(Person)))
print('*****_id:dir(p1):\n',id(dir(p1)))
print('*****_id:dir(p2):\n',id(dir(p2)))
```

## Cont.

```
print('*****Person.__dict__\n',Person.__dict__)
print('*****p1.__dict__\n',p1.__dict__)
print('*****p2.__dict__\n',p2.__dict__)

if __name__=='__main__':
    main()
```

# Date Class

- Today and defaultDate are two objects of class MyDate.
- Arguments passed while invoking constructors may be invalid.
- method checkDay is defined in order to check the arguments whether valid or not.
- print statement of above file returns the output in the string format.

## Date Class Contd.

```
import sys
class MyDate:
    def __init__(self, day=1, month=1, year=2000):
        if not(type(day)==int and type(month)==int \\
               and type(year)==int):
            print('Invalid_data_provided_for_date')
            sys.exit()
        if month>0 and month <=12:
            self.month=month
        else:
            print('Invalid_value_for_month')
            sys.exit()
        if year>1900:
            self.year=year
        else:
            print('Invalid_value_for_year.Year_\\\
                  should_be_greater_than_1900.')
            sys.exit()
        self.day=self.checkday(day)
```

# Date Class Contd.

```
def checkday(self, day):
    if self.year%400==0 or \
       (self.year%100!=0 and self.year%4==0):
        currentYear=[31,29,31,30,31,30,31,31,30,31,30,31]
    else:
        currentYear=[31,28,31,30,31,30,31,31,30,31,30,31]
    if (day>0 and day<=currentYear[self.month-1]):
        return day
    else:
        print('Invalid_value_for_day')
        sys.exit()
def __str__(self):
    if self.day<=9:
        day='0'+str(self.day)
    else:
        day=str(self.day)
    if self.month<=9:
        month='0'+str(self.month)
    else:
        month=str(self.month)
    return day+'-'+month+'-'+str(self.year)
```

# Date Class Contd.

```
def main():
    today=MyDate(3,9,2014)
    print(today)
    defaultDate=MyDate()
    print(defaultDate)
if __name__=='__main__':
    main()
```

Thank you  
Any questions?

# Classes II

## Chapter 11

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Introduction
- 2 Polymorphism
  - Operator Overloading
  - Function Overloading
- 3 Encapsulation, Data Hiding, and data Abstraction
- 4 Modifier and Accessor Methods
- 5 Static Method
- 6 Adding Methods Dynamically
- 7 Composition
- 8 Inheritance
  - Single Inheritance
  - Hierarchical Inheritance
  - Multiple Inheritance
  - Abstract Methods
  - Attribute resolution order for Inheritance
- 9 Built-In functions for classes

# Introduction

- The classes lie at the heart of a programming methodology called object-oriented paradigm or Object-oriented Programming (OOP).
- It includes several concepts which are listed below.
  1. Polymorphism
  2. Encapsulation
  3. Data Hiding
  4. Data Abstraction
  5. Inheritance

# Polymorphism

## Definition

A method/operator may be applied to objects of different types (classes). This feature of object oriented programming is called polymorphism.

## Example

- `len()`: `len` function operates on various types of objects such as `str`, `list`, and `tuple`.

# Operator Overloading

- Use of the same syntactic operator for objects of different classes (types) is called **operator overloading**.
- Python also provides special methods such as `__eq__`, `__lt__`, `__le__`, `__gt__`, and `__ge__` for overloading comparison operators `==`, `<`, `<=`, `>`, and `>=` respectively.
- Overload the special method `__add__` by defining an implementation of addition for objects of type Point given in example below.

# Operator Overloading Cont..

- Output of given code is:(5,7)

```
1 '''Python Program to perform addition of two points in
2     coordinate plane using binary + operator overloading.'''
3
4 class Point:
5     def init (self, x, y):
6         self.x = x
7         self.y = y
8     ''' adding two Point objects'''
9     def add (self, other):
10        x = self.x + other.x
11        y = self.y + other.y
12        return Point(x,y)
13
14 point1 = Point(3,6)
15 point2 = Point(2,1)
16 print(point1 + point2)
```

# Function Overloading

- **Function overloading** provides the ability of writing different functions having the same name, but with a different number of parameters, possibly of the various types.
- Example: We may like to define two functions by the name area to compute the area of a circle or rectangle:

```
1 def area(radius):  
2     """Computes the area of circle"""  
3     areaCirc = 3.14 * radius * radius  
4     return areaCirc  
5 def area(length, breadth):  
6     """Computes the area of rectangle"""  
7     areaRect = length * breadth  
8     return areaRect
```

## Function Overloading Cont...

- Python does not support function overloading. Indeed, Python cannot distinguish between parameters based on their type as the type of a parameter is inferred implicitly when a function is invoked with actual arguments.
- Thus, whereas invoking the function area with a single argument results in error, invoking it with two arguments yields the expected output.
- In python function overloading can be achieved by indirect implementation. An example is shown below where if the function area is invoked with two arguments, it computes the area of the rectangle, and if with only one argument, it computes the area of the circle.

# Function Overloading Cont...

```
1 def area(a, b = None):
2     """
3         Objective: To compute area of a circle or rectangle
4                 depending on the number of parameters.
5         Inputs:    a: radius, in the case of a circle side1, in case
6                 of rectangle
7                 b: None, in the case of a circle side2, in the
8                     case
9                     of rectangle
10                    Output: area of circle or rectangle as applicable"""
11                    """Computes the area of circle"""
12                    if b == None:
13                        areaCirc = 3.14 * a * a
14                        return areaCirc
15                    """Computes the area of rectangle"""
16                    else:
17                        areaRect = a * b
18                        return areaRect
```

# Encapsulation, Data Hiding, and data Abstraction

- **Encapsulation** enables us to group together related data and its associated functions under one name.
- Classes provide an **abstraction** where essential features of the real world are represented in the form of interfaces, hiding the lower-level complexities of implementation details, i.e.

**Abstraction:** representing essential features of the real world, hiding lower level details.

- Technique of restricting access to private members from outside the class is known as **name mangling**.
- The restriction on the use of private variables from outside the scope of the class may be bypassed using the syntax:

*< instance > .. < className >< attributeName >*

## Modifier and Accessor Methods

- The methods in the class definition that modify the value of one or more arguments are known as **modifiers**.
- The methods in the class definition that can only access(not modify) the value of one or more arguments are known as **accessors**.

## Static Method

- An **instance method** typically defines operations on the data members of an instance of a class.
- **Static methods** are used for modifying class data members and do not require passing object as the first parameter.
- To tell Python that a method is a static method, the function decorator **@staticmethod** precedes the method definition.

## Adding Methods Dynamically

- Python allows us to add methods dynamically to a class using the syntax:

```
< className > . < newMethodName > = <  
existingFunctionName >
```

- If we were to associate a method with a particular instance of a class, we need to import the function MethodType from the module types. Subsequently, we use the following syntax to add a method to an instance of a class:

```
< instance > . < newMethodName > = MethodType(<  
existingFunctionName >, < instance >)
```

## Composition

The process of using objects of the other classes as attribute values is called object **composition**.

# Inheritance

- **Inheritance** is an important feature of object oriented programming that imparts ability to a class to inherit properties and behavior of another class.
- Class which inherits properties is called **drived**, sub, or child class and class from which properties are inherited is called **base**, super, or parent class.
- **Object** class is the base class of all classes.
- Example: Person is base class of employee. The notion of inheritance allows us to express the parent–child relationship among the classes as shown in figure (1).

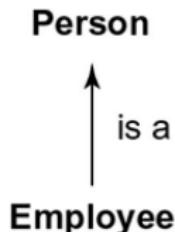


Figure 1: Parent and child class

# Types of Inheritance

- There are different types of inheritance based on how base classes and derived classes are defined. These types are given below:
  1. Single Inheritance
  2. Multilevel Inheritance
  3. Hierarchical Inheritance
  4. Multiple Inheritance

# Single Inheritance

- When inheritance involves a derived class that derives its properties from a single base class as shown in figure (1) is known as **Single Inheritance**.
- Syntax:**

```
class derived_class_name (base_class_name):  
    body of derived class
```

- Example:**
- ```
class Employee (Person):  
    body of derived class
```

- All the data and method attributes defined in the base class become available to the derived class.
- When an object of a derived class makes a call to a method that is also defined in the base class, Python invokes the method of derived class. Thus, the derived class methods **override** the base class methods, and this process is known as method **overriding**.

# Single Inheritance Cont...

- Using **super function** for accessing a method of a base class:  
`super(derived_class_name, self).`  
`__init__(parameters of base class)`  
or  
`super().__init__(parameters of base class)`
- **Scope Rule:**  
To access an instance attribute, Python will first look for it in the instance's namespace, then in the class namespace, and then in the superclass namespace recursively going up in the hierarchy.
- **\_\_bases\_\_**: used to find base class(es) of a class.  
Example:  
`>>> str.__bases__`  
`(<class 'object'>,)`
- **\_\_name\_\_**: used to retrieve the name of a class  
Example:  
`>>> 'Python'.__class__.__name__`  
`'str'`

# Hierarchical Inheritance

- Derived class is like any other class and may serve as base class for another class derived from it and so on.
- Each derived class may define its new attributes. Thus, inheritance allows us to create a class hierarchy, also called type hierarchy.
- When inheritance involves more than one level of hierarchy such as  $A -> B -> C$ , it is called **multilevel inheritance**, and inheritance involving multiple classes deriving from single base class is known as **hierarchical inheritance** as shown in figure (2).
- Hierarchical inheritance may be combined with multiple and multilevel inheritance, to yield hybrid inheritance.

# Hierarchical Inheritance Cont...

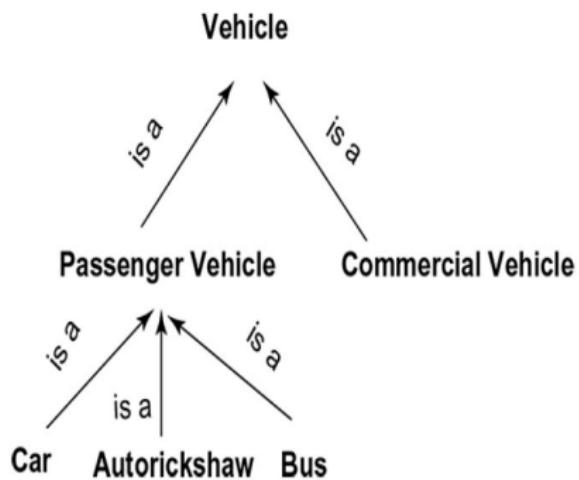


Figure 2: Example of inheritance hierarchy

# Multiple Inheritance

- Subclass derives its attributes from two or more classes then such inheritance is called **multiple inheritance** as shown in figure (3).

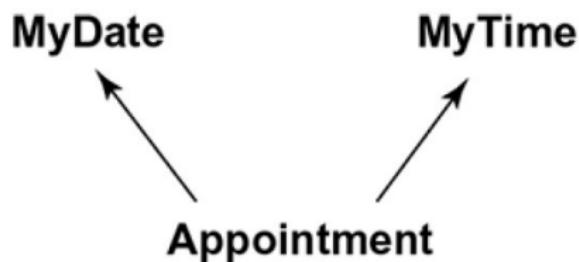


Figure 3: Example of Multiple Inheritance

# Abstract Methods

- An **abstract method** in a base class identifies the functionality that should be implemented by all its subclasses.
- The implementation of an abstract method would differ from one subclass to another, often the method body comprises just a **pass** statement.
- A class containing abstract methods is called **abstract class**.
- To use Python **Abstract Base Classes** (ABCs), one needs to import **ABCMeta** and **abstractmethod** from the abc module.
- To indicate that a method is abstract it is by preceding its definition by **@abstractmethod (function decorator)**.
- The definition of an abstract class begins with **\_\_metaclass\_\_ = ABCMeta**.
- Example:  
Shape is base class of rectangle and circle and area is abstract method because area of rectangle and circle is different.

# Abstract Methods Cont...

```
1 from abc import ABCMeta, abstractmethod
2 class Shape:
3     metaclass = ABCMeta
4     def __init__(self, shapetype):
5         self.shapetype = shapetype
6     @abstractmethod
7     def area(self):
8         pass
9
10 class rectangle(Shape):
11     def __init__(self, length, breadth):
12         Shape.__init__(self, 'rectangle')
13         self.length = length
14         self.breadth = breadth
15     def area(self):
16         return self.length * self.breadth
```

# Abstract Methods Cont...

- Output of following code is:

Area of rectangle with length = 30 and breadth = 15 is: 450

Area of circle with radius = 5 is: 78.5

```
1 class circle(Shape):
2     pi = 3.14
3     def __init__(self, radius):
4         Shape.__init__(self,'circle')
5         self.radius = radius
6     def area(self):
7         return round(circle.pi*(slef.radius**2),2)
8
9 r = rectangle(30,15)
10 area_rectangle = r.area()
11 print("Area of rectangle with length = 30 and breadth = 15
12     is: ",area_rectangle)
13 c = circle(5)
14 area_circle = c.area()
15 print("Area of circle with radius = 5 is: ",area_circle)
```

# Attribute resolution order for Inheritance

- A new-style class inherits from the built-in class object.
- In the new style classes, to access an attribute of an object, Python typically looks for it in the namespace of the object itself, then in the namespace of the class, then in the namespace of the superclasses in the order in which they are derived and so on.
- For determining the order in which methods of classes will be accessed, Python provides the attribute `__mro__` which stands for **method resolution order**.
- `__mro__` returns a list ordered by the classes in which search for methods is to take place.

## Attribute resolution order for Inheritance Cont...

- The newStyleClasses, class C inherits from classes B1 and B2, which further inherit from the class A. It is important to point out that the class C inherits from classes B1 and B2 in the order B1 and B2. **The order of resolution** would be the object c1 of class C, class C, class B1, class B2, class A, class object. The output of this example is 50.

```
1 class A:  
2     test = 20  
3  
4 class B1(A):  
5     pass  
6  
7 class B2(A):  
8     test = 50  
9  
10 class C(B1,B2):  
11     def str (self):  
12         return str(self.test)  
13 c1 = C()  
14 print(c1)
```



# Built-In functions for classes

- To find whether a class is a subclass of another class, one may use the function **issubclass** that takes two arguments: sub and super:

`issubclass(sub, super)`

The function `issubclass` returns True if sub is the subclass of class super, and False otherwise.

- To find whether the instance obj is an object of class class1, we use the function **isinstance**:

`isinstance(obj, class1)`

This function returns True if either obj is an instance of class class1 or it is an instance of a subclass of class class1, and False otherwise.

## Built-In functions for classes Cont...

- To find whether an instance obj contains an attribute attr, we use the function **hasattr**:

```
hasattr(obj, attr)
```

This function returns True if instance obj contains an attribute attr, and False otherwise.

- The functions **getattr** and **setattr** may be used to retrieve and set respectively, the value val of an attribute attr of an instance obj, as illustrated below:

```
getattr(obj, attr)
```

```
setattr(obj, attr, val)
```

- The function **delattr** used to delete an attribute attr of instance obj as follows:

```
delattr(obj, attr)
```

# References

- [1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You  
Any Questions?

# List Manipulation

## Lecture 12

Department of Computer Science and Engineering, ITER  
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



# Contents

- 1 Introduction
- 2 Selection sort
- 3 Bubble sort
- 4 Insertion sort
- 5 Linear Search
- 6 Binary Search
- 7 Merge Sort
- 8 Quick Sort

# Introduction

- In several applications, we need to sort data or search for some item of interest in the given data.
- Searching is more efficient when data is sorted.
- The process of arranging data in ascending/descending order is called **sorting**.
- The attribute or property of data that forms the basis of sorting is called the **key**.

# Selection Sort

- To begin with, we find the lexicographically smallest value in the list and interchange this entry in the list with the first value in the list.
- Next, we find the entry with the smallest value out of the remaining entries, and interchange it with the second value in the list, and proceed in this manner.
- If there are  $n$  values, only  $n-1$  values need to be placed in order because when  $(n-1)$  values have been put in order, then the  $n$ th value would automatically be in order.
- This technique of sorting is called **Selection sort**.

## Selection sort (Contd.)

In the first iteration ( $i=0$ )

- We find the smallest name in the list.
- To do this, we begin with the name at index  $j=0$ .
- We keep track of the index of the smallest name examined so far using the variable *minIndex*.
- When  $j=0$ , *minIndex* is equal to 0

$j=0$	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5
minIndex						

## Selection sort (Contd.)

- Compare value at index  $j=1$  with the value at index  $minIndex$ . Since 'Sanvi' is smaller than 'Vijaya',  $minIndex$  will be set equal to 1.

$j=1$	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5
	$minIndex$					

- Now we compare the next value which is at index  $j=2$  with the value at  $minIndex$ .
- Since 'Ruby' is smaller than 'Sanvi', we will set  $minIndex$  to 2.

$j=2$	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5
	$minIndex$					

## Selection sort (Contd.)

- When  $j=3$ , 'Zafar' will be compared with 'Ruby' (value at  $minIndex$  2).
- Since 'Zafar' is greater than 'Ruby', the value of  $minIndex$  will remain unchanged.

$j=3$	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

$minIndex$

- When  $j=4$ , 'Maya' will be compared with 'Ruby' (value at  $minIndex$  2), and  $minIndex$  will be set to 4 since 'Maya' is smaller than 'Ruby'.

$j=4$	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

$minIndex$

## Selection Sort (Contd.)

- Finally, when  $j=5$ , 'Anya' is found to be smaller than 'Maya' (value at  $\text{minIndex}$  4).
- Thus, after scanning the entire list,  $\text{minIndex}$  will have value 5.

$j=5$	'Vijaya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Anya'
	0	1	2	3	4	5

$\text{minIndex}$

- Now, we need to exchange this smallest name entry 'Anya' at index 5 ( $\text{minIndex}$ ) with the name at index  $i=0$ .
- Modified data **at the end of first iteration** looks like:

'Anya'	'Sanvi'	'Ruby'	'Zafar'	'Maya'	'Vijaya'
0	1	2	3	4	5

# Selection Sort Code

```
1 '''Selection Sort'''
2 def selectionSort(lst):
3     for i in range(0,len(lst)-1):
4         minIndex=i
5         for j in range(i+1,len(lst)):
6             if lst[j]<lst[minIndex]:
7                 minIndex=j
8             if minIndex!=i:
9                 lst[minIndex],lst[i]=lst[i],lst[minIndex]
10 def main():
11     lst=eval(input('Enter a list :'))
12     print('Sorted list :')
13     selectionSort(lst)
14     print(lst)
15 if __name__=='__main__':
16     main()
```

## Selection Sort (Contd.)

- Since a list (like other objects) is passed by reference in Python, and the function only modifies the components of the list, **there is no need to return the sorted list.**
- The function selectionSort works equally well on numeric and string data.

# Bubble Sort

- In each pass, we compare values in adjacent positions and interchange them, if they are out of order.
- To begin with, we have a list of  $n$  values.
- The  $n$ th value and  $(n-1)$ th value are compared and interchanged if the  $n$ th value is smaller than  $(n-1)$ th value, then  $(n-1)$ th value and  $(n-2)$ th value are compared and interchanged if found out of order, and so on.

## Bubble Sort (Contd.)

- In the first pass, the smallest value will move to the front of the list at index 0; on subsequent passes, it will be ignored.
- In the second iteration, we need to sort the remaining list of  $n-1$  values excluding the value at index 0.
- After  $n-1$  iterations, the list will be completely sorted, and the algorithm will halt.
- We start at index 5 and keep on comparing adjacent locations in order to move the smallest name to the beginning of the list.

The steps performed in first iteration are as follows:

j=5	'Vijaya'	'Sanvi'	"Ruby"	"Zafar"	'Maya'	'Anya'
	0	1	2	3	4	5
j=4	'Vijaya'	'Sanvi'	"Ruby"	'Zafar'	'Anya'	'Maya'
	0	1	2	3	4	5
j=3	'Vijaya'	'Sanvi'	"Ruby"	'Anya'	'Zafar'	'Maya'
	0	1	2	3	4	5
j=2	'Vijaya'	"Sanvi"	"Anya"	"Ruby"	"Zafar"	'Maya'
	0	1	2	3	4	5
j=1	'Vijaya'	'Anya'	'Sanvi'	"Ruby"	"Zafar"	'Maya'
	0	1	2	3	4	5

# Bubble Sort Code

```
1 def bubbleSort(lst):
2     n=len(lst)-1
3     for i in range(0,n):
4         for j in range(n,i,-1):
5             if lst[j]<lst[j-1]:
6                 lst[j],lst[j-1]=lst[j-1],lst[j]
7 def main():
8     lst=eval(input('Enter the list :'))
9     print('Sorted list :')
10    bubbleSort(lst)
11    print(lst)
12 if __name__=='__main__':
13     main()
```

# Bubble Sort (Modified)

- If not even a single pair of data is swapped in an iteration, we conclude that the list has already been sorted.
- The control should break out of the loop, instead of unnecessarily proceeding with the remaining iterations.
- So, we modify the Bubble sort code to include this possibility and reduce the time.

# Bubble Sort Code

```
1 def bubbleSort(lst):
2     n=len(lst)-1
3     for i in range(0,n):
4         swap=False
5         for j in range(n,i,-1):
6             if lst[j]<lst[j-1]:
7                 swap=True
8                 lst[j],lst[j-1]=lst[j-1],lst[j]
9             if swap==False:
10                 break
11
12 def main():
13     lst=eval(input('Enter the list :'))
14     print('Sorted list :')
15     bubbleSort(lst)
16     print(lst)
17 if __name__=='__main__':
18     main()
```

# Insertion Sort

- The list is logically divided into two parts.
- The left part is the sorted part, the right part is the unsorted part comprising the elements yet to be arranged in sorted order.
- In each iteration, we increase the length of the sorted part by one in the following manner:
  - Insert the first element from the unsorted part into the sorted part at the correct position.
  - To find the correct position of the value to be inserted, we compare it with values in the sorted part (starting from the rightmost value in the sorted part) and shift each value to the right by one position, until the correct position is found.

## Insertion Sort (Contd.)

- Examine the list below:

'Anya'	'Sanvi'	'Vijaya'	'Zafar'	'Maya'	'Ruby'
0	1	2	3	4	5

- In this example, the left sorted part (grey portion) comprises elements at indices 0, 1, 2, 3
- The right unsorted part (white portion) comprises the elements at indices 4 and 5.

## Insertion Sort (Contd.)

- We need to insert element  $lst[4]$  at the correct position.
- As 'Maya' is less than 'Zafar', we shift 'Zafar' one position right, but before we do so, we must save 'Maya' in a temporary variable, say, temp.
- The modified list is

temp



'Maya'

Step 1	'Anya'	'Sanvi'	'Vijaya'	'Zafar'	'Zafar'	'Ruby'
	0	1	2	3	4	5

## Insertion Sort (Contd.)

- Next, we compare the value of the variable temp, i.e. 'Maya' with  $lst[2]$ , i.e., 'Vijaya'.
- As 'Maya' is less than 'Vijaya', we shift 'Vijaya' one position right.
- The modified list is

Step 2	'Anya'	'Sanvi'	'Vijaya'	'Vijaya'	'Zafar'	'Ruby'
	0	1	2	3	4	5

## Insertion Sort (Contd.)

- Next, we compare the value of the variable temp, i.e. 'Maya' with  $lst[1]$ , i.e. 'Sanvi'.
- As 'Maya' is less than 'Sanvi', we shift 'Sanvi' one position right.
- The modified list is

Step 3	'Anya'	'Sanvi'	'Sanvi'	'Vijaya'	'Zafar'	'Ruby'
	0	1	2	3	4	5

## Insertion Sort (Contd.)

- Next, we compare, 'Maya' with  $Ist[0]$ , i.e. 'Anya'.
- As 'Maya' is greater than 'Anya', we do not need to shift 'Anya', and 'Maya' can now be placed at index 1.
- The modified list is

'Anya'	'Maya'	'Sanvi'	'Vijaya'	'Zafar'	'Ruby'
0	1	2	3	4	5

- Now 'Maya' has been placed at its correct position.

# Insertion Sort Code

```
1 def insertionSort(lst):
2     for i in range(1,len(lst)):
3         temp=lst[i]
4         j=i-1
5         while j>=0 and lst[j]>temp:
6             lst[j+1]=lst[j]
7             j=j-1
8         lst[j+1]=temp
9 def main():
10    lst=eval(input('Enter a list :'))
11    print('Sorted list :')
12    insertionSort(lst)
13    print(lst)
14
15 if __name__=='__main__':
16     main()
```

# Searching

- We will see how to find out whether a data value appears in a list.
- For example, given a list of names of students in a class, we wish to find whether a particular name appears in the list.
- We'll discuss two methods of searching here, **Linear search** and **Binary search**

# Linear search

- In this method, we scan the list from the beginning till the required data value is found or the list is exhausted.
- This searching technique is known as **linear search** as the search process is sequential.

# Linear search Code

```
1 def linearSearch(lst,searchValue):
2     for i in range(0,len(lst)):
3         if lst[i]==searchValue:
4             return i
5     return None
6 def main():
7     lst=eval(input('Enter a list :'))
8     searchVal=eval(input('Enter the value to be searched :'))
9     searchResult=linearSearch(lst,searchVal)
10    print(searchVal,'found at index',searchResult)
11 if __name__=='__main__':
12     main()
```

# Binary search

- If the list to be searched is already sorted, we can use a faster method, called **binary search**.
- First, we examine the middle name of the list, if it matches the name that we are looking for, we stop.
- If we do not find the name at the middle position, we will only have to search on the left or right side of the middle name depending on whether the name at the middle position is greater or smaller than the name we are looking for.
- To see how fast binary search works, we only have to observe that every time search fails, the search interval is reduced to half. Thus, in the worst case, we would require no more than  $(\log N) + 1$  steps

# Binary search Code

```
1 def binarySearch(lst,searchValue):
2     low,high=0,len(lst)-1
3     while low<=high:
4         mid=int((low+high)/2)
5         if lst[mid]==searchValue:
6             return mid
7         elif searchValue<lst[mid]:
8             high=mid-1
9         else:
10            low=mid+1
11    return None
12 def isSorted(lst):
13     for i in range(1,len(lst)):
14         if lst[i]<lst[i-1]:
15             return False
16     return True
```

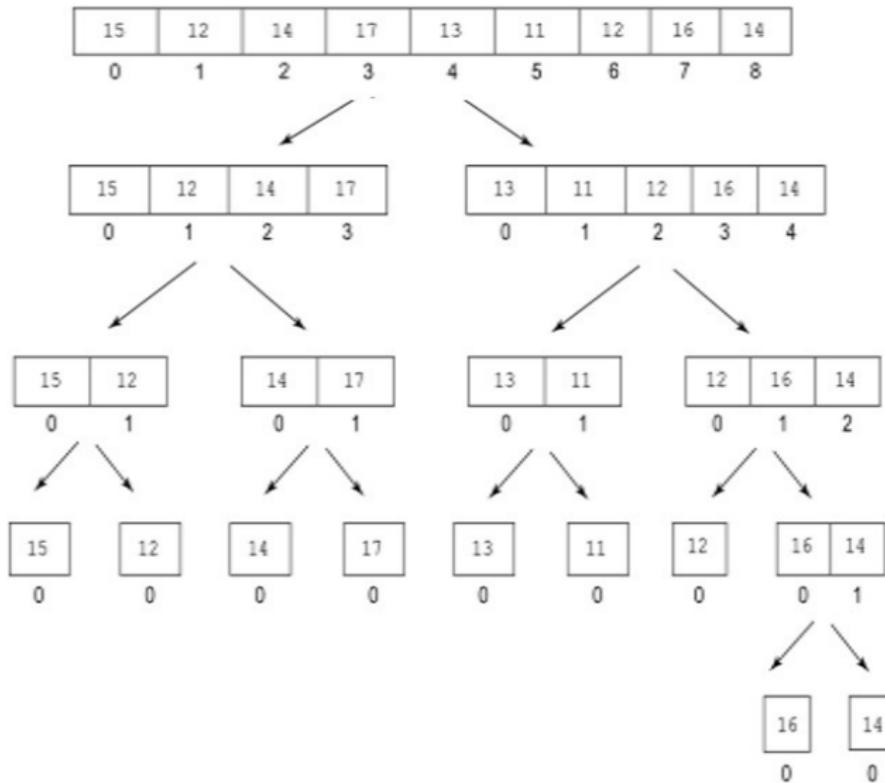
## Binary search Code(Contd.)

```
1 def main():
2     lst=eval(input('Enter a sorted list (ascending order) :'))
3     if not(isSorted(lst)):
4         print('Given list is not sorted')
5     else:
6         searchVal=eval(input('Enter the value to be searched :'))
7         print(searchVal,'found at index',binarySearch(lst,
8             searchVal))
9 if __name__=='__main__':
10    main()
```

# Merge Sort

- The merge sort algorithm is based on the divide and conquer strategy that takes a list as an input and keeps on dividing it into two lists until a list of length 1 is obtained.
- It conquers the sorting problem by merging pairs of length one lists to obtain lists of length two, merging pairs of lists of length two to obtain lists of length four, and so on until the final sorted list of the size of the original list is obtained.

# Test Case



# Merge Sort Code

```
1 def merge(lst1, lst2):
2     sortedList=[]
3     while len(lst1)!=0 and len(lst2)!=0:
4         if lst1[0]<lst2[0]:
5             sortedList.append(lst1[0])
6             lst1.remove(lst1[0])
7         else:
8             sortedList.append(lst2[0])
9             lst2.remove(lst2[0])
10    if len(lst1)==0:
11        sortedList+=lst2
12    else:
13        sortedList+=lst1
14    return sortedList
```

## Merge Sort Code(Contd.)

```
1 def mergeSort(lst):
2     if len(lst)==0 or len(lst)==1:
3         return lst
4     else:
5         mid=len(lst)//2
6         lst1=mergeSort(lst[:mid])
7         lst2=mergeSort(lst[mid:])
8         return merge(lst1,lst2)
```

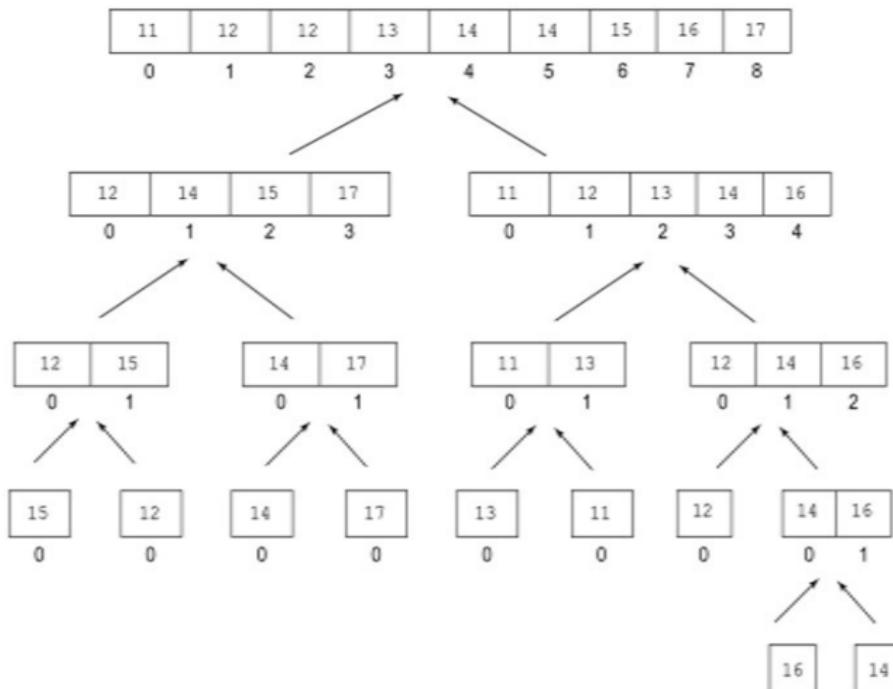
# Quick Sort

- **Quick sort** is more efficient in terms of the space requirement as it does not need extra space for sorting.
- It works by choosing a value called **pivot element**, using which we split the list to be sorted in two parts:
  - The first part comprising elements smaller than or equal to the pivot element
  - The second part comprising elements greater than the pivot element
- The two parts are further sorted using quick sort in a similar manner until the partitions of length 1 or 0 are left.

# Quick Sort Code

```
1 def partition(lst,start,end):
2     pivotElement=lst[end]
3     i=start-1
4     for j in range(start,end):
5         if lst[j]<=pivotElement:
6             i+=1
7             lst[i],lst[j]=lst[j],lst[i]
8     lst[i+1],lst[end]=lst[end],lst[i+1]
9     return i+1
10 def quickSort(lst,start=0,end=None):
11     if end==None:
12         end=len(lst)-1
13     if start<end:
14         splitPoint=partition(lst,start,end)
15         quickSort(lst,start,splitPoint-1)
16         quickSort(lst,splitPoint+1,end)
17     return lst
```

# Test Case



# References

- [1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You  
Any Questions?