

# Python Programming-An Introduction

# 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

- 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.

# IDLE - An Interpreter for Python

- IDLE stands for “*Integrated Development and Learning Environment*”.
- Python IDLE comprises:
  - 1 Python Shell: An Interactive Interpreter.
  - 2 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
```

# IDLE - An Interpreter for Python

- 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 can 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

Table 1: Precedence of Arithmetic Operators.

( ) : 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

# IDLE - An Interpreter for Python

```
>>> 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.
- 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.
  - Result can be: **True** or **False**.
- Arithmetic operators have higher precedence than the relational operators.

Table 2: Relational Operators.

<code>==</code>	equal to
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt; =</code>	less than or equal to
<code>&gt; =</code>	greater than or equal to
<code>! =</code>	not equal to

# Relational Operators

- A relational operator applied on expressions as:  
**expression ( comparison operator ) 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.
  - Produces either *True* or *False*.
- The operator *not* is unary, whereas other two are binary operator, those are as described below:

Table 3: *not* operator

True	False
False	True

Table 4: *and* operator

	True	False
True	True	False
False	False	False

Table 5: *or* operator

	True	False
True	True	True

# Logical Operators

False	True	False
-------	------	-------

# Logical Operators

Table 6: Precedence of Logical Operators.

not	Decreasing Order ↓
and	
or	

Table 7: Precedence of Operators.

Arithmetic Operators	Decreasing Order ↓
Relational Operators	
Logical Operators	

- 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.

Table 8: Boolean Operators.

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 : 0 0 0 0 1 0 1 0 6 : 0 0 0 0 0 1 1 0 ----- 2 : 0 0 0 0 0 0 1 0
Bitwise OR $x   y$	A bit in $x   y$ is 1, if at least one of the corresponding bit in $x$ and $y$ is 1, and 0 otherwise.	10   6 yields 14: 10 : 0 0 0 0 1 0 1 0 6 : 0 0 0 0 0 1 1 0 ----- 14 : 0 0 0 0 1 1 1 0
Bitwise Compliment $\sim x$	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: $\sim 11$ : 0 0 0 0 1 0 1 1 ----- $\sim 11$ : 1 1 1 1 0 1 0 0 Note that 1 1 1 1 0 1 0 0 is the two's compliment of 12: 0 0 0 0 1 1 0 0

# Bitwise Operators

Table 9: Boolean Operators.

Bitwise Operator	Description	Example
Bitwise Exclusive OR $x \wedge y$	A bit in $x \wedge y$ is 1, if exactly one of the corresponding bits in $x$ and $y$ is 1, and 0 otherwise.	$10 \wedge 6$ yields 12: 10 : 0 0 0 0 1 0 1 0 6 : 0 0 0 0 0 1 1 0 ----- 12 : 0 0 0 0 1 1 0 0
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 $x * 2^y$ .	$5 << 2$ yields 20: 5 : 0 0 0 0 0 1 0 1 ----- 20 : 0 0 0 1 0 1 0 0
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 bits. The result obtained from this operation is $x / 2^y$ .	$5 >> 2$ yields 1: 5 : 0 0 0 0 0 1 0 1 ----- 1 : 0 0 0 0 0 0 0 1

# Bitwise Operators

Table 10: Precedence of Operators.

( )	Decreasing Order ↓
**	
+X, -X, ~X	
/, //, *, %	
+, -	
<<, >>	
&	
^	
==, < , >, <=, >=, !=	
not	
and	
or	

# 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.  $a = a \langle\text{operator}\rangle b$  is equivalent to  $a \langle\text{operator}\rangle = b$ . For example:  $a = a + b$  is equivalent to  $a += b$ .
- Multiple assignments in a single statement as:
- Assigning same value to multiple variables in a single statement as:

```
>>> a, b, c = 'how', 'are', 'you?'
```

```
>>> 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.

Table 11: List of Python Keywords.

False	class	finally	is	return	None
continue	for	lambda	try	True	def
from	nonlocal	while	adn	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.

```
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
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.

The screenshot shows a 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 indicates "Ln: 5 Col: 0".

The screenshot shows a window titled "Python 3.6.1 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell displays the Python version and build information, followed by a prompt and the script's output:

```
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/readNshowWell.py =====
Enter first number: 3
Enter second number: 6
Numbers are: 3 6
```

The status bar at the bottom right indicates "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

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

Thank You  
Any Questions?

# Functions

## Lecture 2

# Contents

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

- 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')
```

# Built-in Functions (Cont.)

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'
```

## Built-in Functions (Cont.)

- 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
<code>ceil(x)</code>	Returns the smallest integer greater than or equal to x.
<code>floor(x)</code>	Returns the largest integer less than or equal to x.
<code>fabs(x)</code>	Returns the absolute value of x.
<code>exp(x)</code>	Returns the value of expression $e^{**}x$ .
<code>log(x,b)</code>	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.
<code>log10(x)</code>	Returns $\log(x)$ to the base 10. This is equivalent to specifying <code>math.log(x,10)</code> .
<code>pow(x,y)</code>	Returns x raised to the power of y, i.e., $x^{**}y$ .
<code>sqrt(x)</code>	Returns the square root of x.
<code>cos(x)</code>	Returns the cosine of x radians.
<code>sin(x)</code>	Returns the sine of x radians.
<code>tan(x)</code>	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):  
    statement_1  
    statement_2  
    statement_3  
    .....  
    statement_n
```

- “*function name*” should not be a python keyword.
- Python code following colon must be intended, i.e., shifted right.

# Function Definition and Call

- Example: Program to print a triangle, a blank line and a square:  
**(picture.py)**

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

    # To print a blank line
    print()

    # To print a square
    print('* * * *')
    print('* * * *')
    print('* * * *')
    print('* * * *')
```

## Function Definition and Call (Cont.)

- Executing a python file, for e.g., **picture.py**
- To do this, we need to invoke the function `main()` in the following two steps.

- 1 In Run menu, click on the option Run Module.
- 2 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.py**, 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('*****')

    # To print a blank line
    print()

    # To print a square
    print('* * * *')
    print('* * * *')
    print('* * * *')
    print('* * * *')

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

## Function Definition and Call (Cont.)

- Invoking a function from main() function.

```
def triangle():
    # To print a triangle
    print('*')
    print('***')
    print('*****')
    print('*****')

def main():
    # To print a triangle
    triangle()

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

- Function main serves as a caller function for the callee or called function triangle.

## 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.

## Function Definition and Call (Cont.)

- Example: Program to compute 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
```

## 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

## 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.

## Function Definition and Call (Cont.)

- Default Parameter Values: 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.

# Command Line Arguments

- Steps to execute the python script from the command line interface.
  - Open the directory (say F:\PythonCode \Ch02) containing the file “area.py”.
  - Open the command prompt window using an option in context menu.
  - Execute the command : **python area.py**

```
F:\PythonCode \Ch02>python area.py
Enter the following values for rectangle:
Length : integer value : 20
Breadth : integer value : 10
Area of rectangle is 200
End of program
```

# Command Line Arguments

- Command line arguments can be used to accept the input from the user.

```
F:\PythonCode\Ch02>python area.py 20 10  
Area of rectangle is 200  
End of program
```

- When the above command is executed, the arguments including the name of python program (“area.py”) and the following inputs (20 and 10) gets stored in the list “`sys.argv`”.
- Name of the script is the first argument and accessed as `argv[0]`.
- Similarly, 20 and 10 can be accessed as `argv[1]` and `argv[2]`, respectively

# References

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

# Control Structures

## Lecture 3

# Contents

- 1 Introduction
- 2 Conditional Statement: if
- 3 Conditional Statement: if-else
- 4 Iteration (for and while statements)
  - Iteration: for loop
  - Iteration: while statement

# 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.

## Conditional Statement: if

- *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.

# Conditional Statement: if

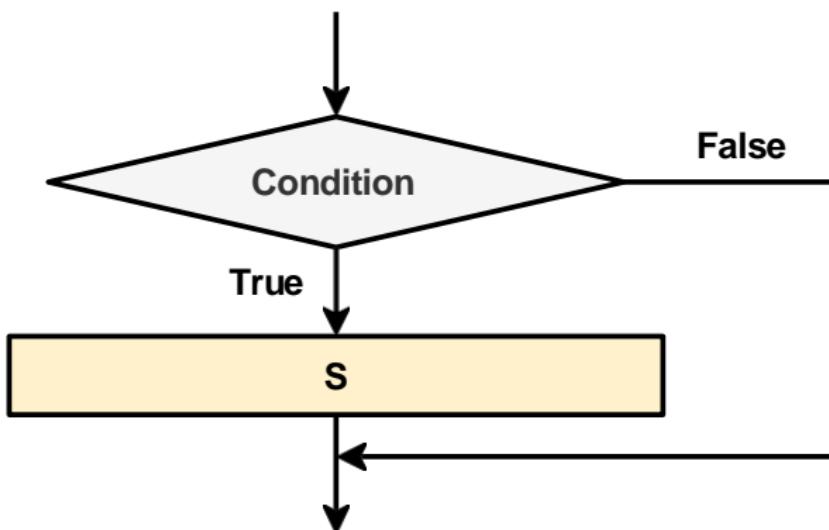


Figure 1: Flow diagram for *if* statement.

# Conditional Statement: if

- 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()
```

## Conditional Statement: if

# Conditional Statement: if

- For Example:

```
>>> Enter marks: 38
```

```
Moderated marks: 40
```

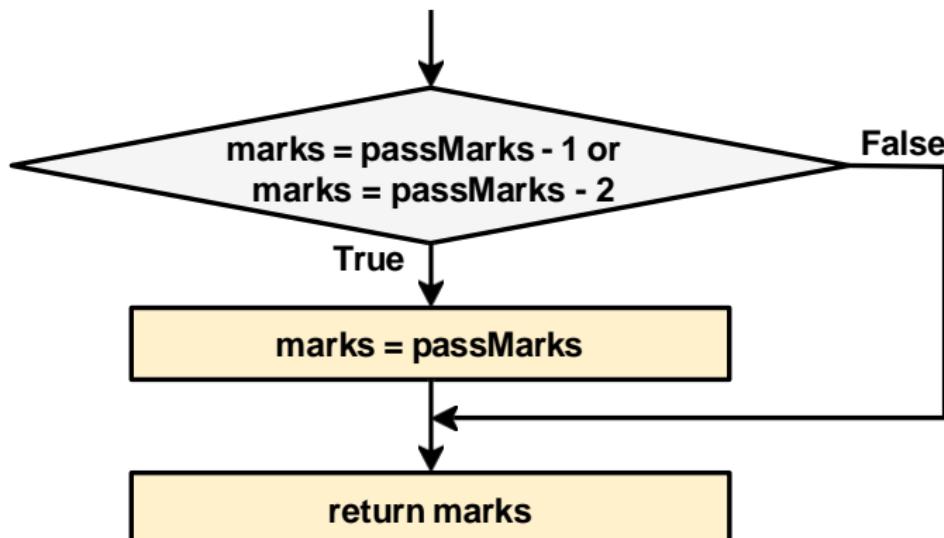


Figure 2: Flowchart of function **moderate**.

## Conditional Statement: if-else

- 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.

# Conditional Statement: if-else

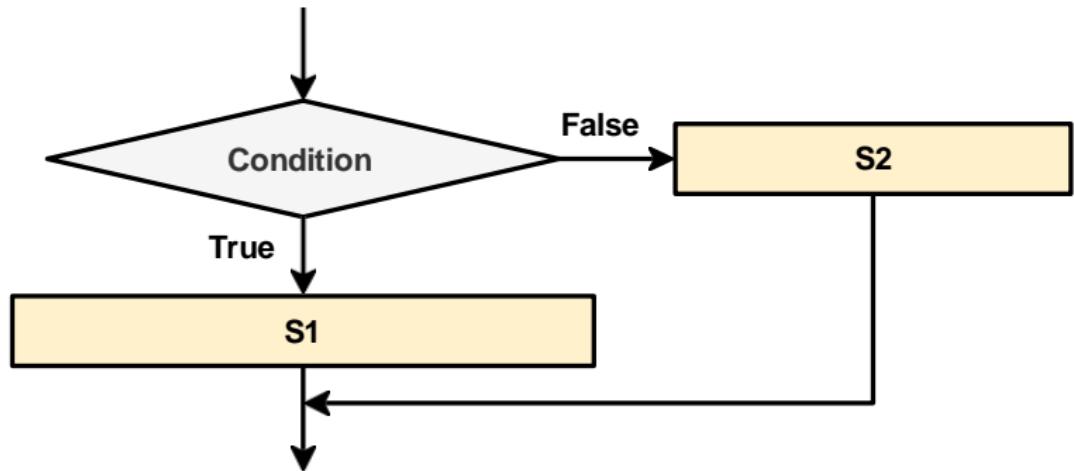


Figure 3: Flow diagram of *if-else* structure.

# Conditional Statement: if-else

- 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.'
    else:
        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:')
    message = authenticateUser(password)
    print(message)

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

## Conditional Statement: if-else

# Conditional Statement: if-else

- For Example:

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

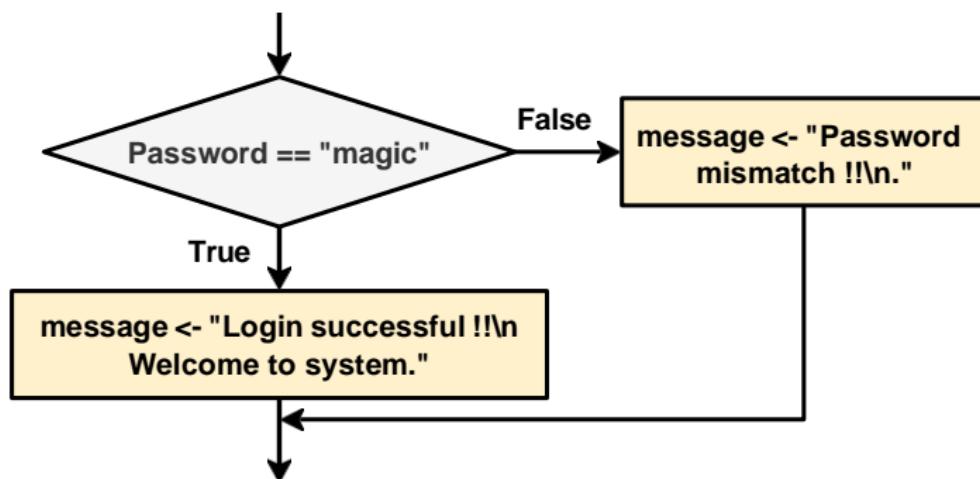


Figure 4: Flowchart of *if-else* statement in the function authenticateUser

## Conditional Statement: if-elif-else

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

**if** *(condition)* :

    Sequence  $S_1$  of statements to be executed

**elif** *(condition2)* :

    Sequence  $S_2$  of statements to be executed

**elif** *(condition3)* :

    Sequence  $S_3$  of statements to be executed

...

**else:**

    Sequence  $S_n$  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**.
- Loops eliminates repetitive writing of the same sequence of statements.
- Execution of a sequence of statements in a loop is known as an iteration of the loop.
- Commonly employed commands for executing in a loop are:
  - ① **for**: This is used when we want to execute a sequence of statements a fixed number of times.
  - ② **while**: This is used when the number of times to execute a sequence of statements is not known in advance.

- 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 loop

- 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: while statement

- 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: while statement

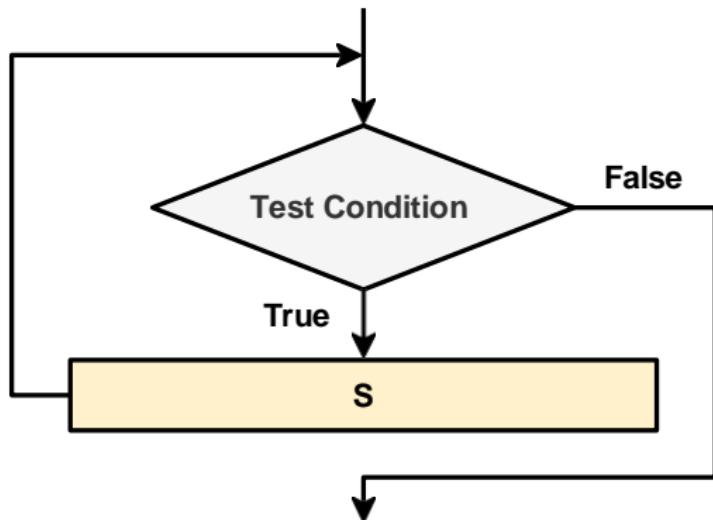
- For Example (Program to compute sum of numbers):

```
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: while statement

- The flow diagram of While structure:



# Iteration: for vs 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

222

```
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)
ame__='__main__':
main()
```

**Output: (a) Right Triangle**

### (b) Inverted Triangle

\* \* \* \*

\* \* \* \* \*

# Iteration (for and while statements)

## Examples of nested control structures

for(...): #Sequence of statements S1 for(...): #Sequence of statements S2 #Sequence of statements S3	while (...): #Sequence of statements S1 while(...): #Sequence of statements S2 #Sequence of statements S3
for(...): #Sequence of statements S1 while(...): #Sequence of statements S2 #Sequence of statements S3	while (...): #Sequence of statements S1 for(...): #Sequence of statements S2 #Sequence of statements S3
for(...): #Sequence of statements S1 for(...): #Sequence of statements S2 while(...): #Sequence of statements S3 #Sequence of statements S4 #Sequence of statements S5	while (...): #Sequence of statements S1 for(...): #Sequence of statements S2 while(...): #Sequence of statements S3 #Sequence of statements S4 #Sequence of statements S5

## 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

# Contents

1 Introduction

2 Testing

3 Debugging

- 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.

## 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

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

# Contents

1 Introduction

2 del operator

3 Namespaces

4 Scope

# Introduction: 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 and write the code.
- Click on Visualize Execution.

The screenshot shows the Python Tutor interface. At the top, there's a navigation bar with links for Apps, Research, Logins, Jobs, Technical, Stories, Study, Journals, Poster, Conferences, Speakers, and Project. Below the navigation bar, a message says "Get live help in the Python Discord chat". The main area has a "Write code in" dropdown set to "Python 3.6" containing the following 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".

# Visualization of Example-I in Python tutor (Continue...)

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

The screenshot shows the Python Tutor interface. On the left, a code editor displays Python 3.6 code with comments about known limitations. The code defines a main function that prints variable values and their memory addresses. A red arrow points to the first line of the main function. Below the code is an 'Edit this code' button. At the bottom, there are navigation buttons for 'First', 'Prev', 'Next', and 'Last'. The status bar indicates 'Step 1 of 12'. On the right, a large 'Print output' box is visible, with 'Frames' and 'Objects' tabs below it. The browser address bar shows 'pythontutor.com/visualize.html#mode=display'. The page title is 'Python Tutor - Visualize Python'.

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

line that just executed  
next line to execute

<< First < Prev Next >> Last >>

Step 1 of 12

Customize visualization

Print output (drag lower right corner to resize)

Frames Objects

4 / 35

# Visualization of Example-I in Python tutor (Continue...)

- 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 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()

Edit this code
```

Annotations in the screenshot:

- A green arrow points to line 11, "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 be executed.
- The "Global frame" section shows the variable "main" pointing to the function "function main()".
- The "Frames" and "Objects" sections are also visible.

At the bottom, there are navigation buttons: << First, < Prev, Next >, Last >>, and Step 2 of 12. The page is identified as 5 / 35.

# Visualization of Example-I in Python tutor (Continue...)



# Visualization of Example-I in Python tutor (Continue...)

- 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.

Python Tutor - Visualize Python. x +

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

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

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

line that just executed

next line to execute

Print output (drag lower right corner to resize)

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

Frames Objects

Global frame

main

function main()

main

a 5

# Visualization of Example-I in Python tutor (Continue...)



# Visualization of Example-I in Python tutor (Continue...)

- 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.

Python Tutor - Visualize Python. x +

pythonutor.com/visualize.html#mode=display

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

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

line that just executed  
next line to execute

<< First < Prev Next >> Last >>

Step 9 of 12

Customize visualization

Print output (drag lower right corner to resize)

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

Frames Objects

Global frame

```
function main()
```

main

main

a	5
b	5

9 / 35

# Visualization of Example-I in Python tutor (Continue...)

- 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 with the following details:

**Code Area:**

```
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
```

**Print output:**

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

**Frames and Objects:**

- Global frame:** Contains the variable `main`.
- function main():** Contains the local variables `a` (value 7) and `b` (value 5).

**Tooltips:**

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

**Bottom Navigation:**

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

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

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

10 / 35

## Visualization of Example-I in Python tutor (Continue...)

- 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

$$b = 3 + 2 \tag{1}$$

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.

## Visualization of Example-I in Python tutor (Continue...)

- 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  
=>>>
```

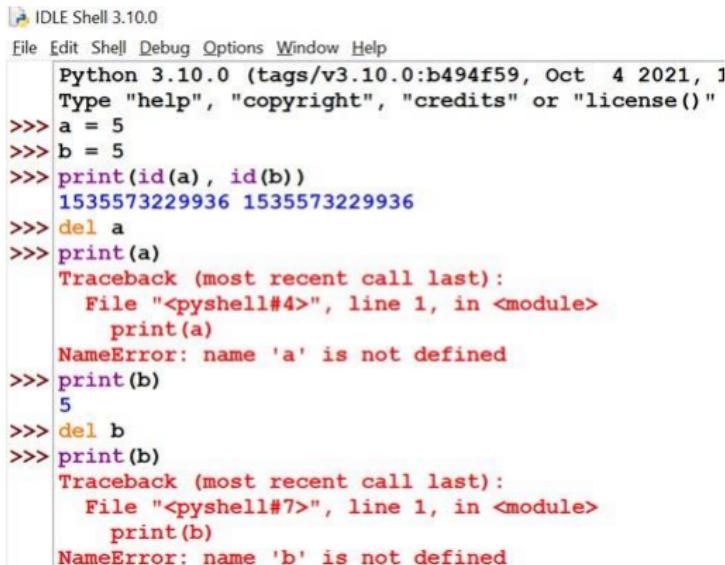
- Note that the first three instructions create new objects.

# Object ID for different data types

- 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.

# del operator

- 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 with the URL [pythontutor.com/visualize.html#mode=display](http://pythontutor.com/visualize.html#mode=display). The code being run is:

```
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 status bar indicates "line that just executed" (green arrow) and "next line to execute" (red arrow). The output window shows the results of the code execution:

Print output (drag lower right corner to resize)

Frames	Objects
Global frame	
b   5	a   5

Below the code editor, there are navigation buttons: << First, < Prev, Next >, Last >>. The current step is Step 3 of 8. At the bottom, there are links for "Customize visualization" and a set of navigation icons.

# 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 for Python 3.6. On the left, a code editor displays the following script:

```
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)
```

Line 5, `del a`, is highlighted with a green arrow pointing to it, indicating it is the next line to execute. Line 6, `print(b)`, is highlighted with a red arrow pointing to it, indicating it is the line that just executed. Below the code editor are two buttons: "Edit this code" and a progress bar labeled "Step 6 of 8".

On the right, the "Print output" panel shows the results of the code execution:

```
140366912186400 140366912186400
5
```

Below the output panel are two tabs: "Frames" and "Objects". The "Objects" tab is selected, showing a table with one row:

Global frame
b   5

At the bottom right of the interface are navigation icons for search and refresh.

# 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 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, Speaker

**Code Area:** 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)
```

**Output Area:** Print output (drag lower right corner to resize)

```
140366912186400 140366912186400
5
5
```

**Execution Indicators:**

- Line 7: green arrow pointing to `del b`, labeled "line that just executed".
- Line 8: red arrow pointing to `print(b)`, labeled "next line to execute".

**Navigation:** << First, < Prev, Next >, Last >>

Step 8 of 8

[Customize visualization](#)

**Page-Footer:** 19 / 35

# 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 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, Conference
- Text Area:** Python 3.6 (known limitations)  
The code is:

```
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()
```
- Buttons:** Frames, Objects
- Execution Status:** line that just executed (green arrow), next line to execute (red arrow)
- Navigation:** << First, < Prev, Next >, Last >>
- Progress:** Step 1 of 5
- Bottom:** Customize visualization, Help, 20 / 35

## 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

pythontutor.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 percent

function percent(marks, maxMarks)

Edit this code

line that just executed

next line to execute

<< First < Prev Next >> Last >>

Step 2 of 5

Customize visualization

Navigation icons: back, forward, search, etc.

22 / 35

# 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:

**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 Indicators:**

- Line 1: green arrow pointing left, labeled "line that just executed".
- Line 5: red arrow pointing right, labeled "next line to execute".

**Frames and Objects Panel:**

Frames	Objects
Global frame	function percent(marks, maxMarks)

A blue box highlights the "percent" entry in the Global frame column, with a red arrow pointing from it to the corresponding function definition in the code area.

**Navigation and Status:**

<< First < Prev Next >> Last >>

Step 2 of 5

Customize visualization

23 / 35

# 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 ⌂ 🔍 pythonutor.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

19 / 35

# 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

Frames Objects

Global frame → function percent(marks, maxMarks)

percent → function main()

main

<< First < Prev Next >> Last >>

Step 5 of 5

20 / 35

```
graph LR
    GlobalFrame[Global frame] --- percentObject["function percent(marks, maxMarks)"]
    GlobalFrame --- mainObject["function main()"]
    percentObject --- percentFunc[percent]
    mainObject --- mainFunc[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.

Python Tutor - Visualize Python +

pythontutor.com/visualize.html#mode=display

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

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

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

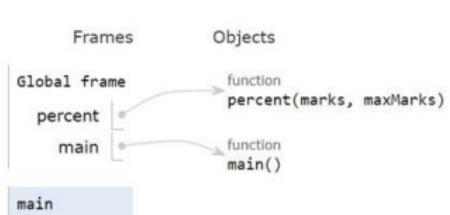
Frames Objects

Global frame

percent → function percent(marks, maxMarks)

main → function main()

main



## Visualization of Example-II (continued)



# 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.
- Then next click executes again prompting for marks obtained (say, 450).

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

**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()
```

**Output Area:**

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

**Frames and Objects:**

- Global frame
  - percent → function percent(marks, maxMarks)
  - main → function main()
- main
  - maxMarks | 500.0

**Execution Indicators:**

- line that just executed (green arrow pointing to line 7)
- next line to execute (red arrow pointing to line 8)

**Navigation and Input:**

<< First < Prev Next >> Last >>

Enter user input:  
Enter marks obtained: 450 Submit

23 / 35

# 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... +

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 >> Last >>

Enter user input:

Enter marks obtained: 450 Submit

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

Customize visualization

24 / 35

# 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 with the following details:

**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()

Edit this code
```

**Print Output:**

```
Enter total marks: 500
Enter marks obtained: 450
```

**Frames and Objects:**

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

**Execution Control:**

- line that just executed
- next line to execute

**Navigation:**

<< First < Prev Next > Last >>

Step 10 of 14

[Customize visualization](#)

# 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

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

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 the function main.
- Note that the variable percentage of the function main now maps to the float object 90.0.

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

[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)

```
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 12 of 14

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

Customize visualization

27 / 35

# Visualization of Example-II (continued)

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

Python Tutor - Visualize Python. × +

pythonutor.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

line that just executed

next line to execute

<< First < Prev Next > Last >>

Step 14 of 14

Customize visualization

Navigation icons: back, forward, search, etc.

28 / 35

# 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.

- 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()
```

# Examples (continued)

- 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.

# References

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

# Strings

## Lecture 6

# 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.
- 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.

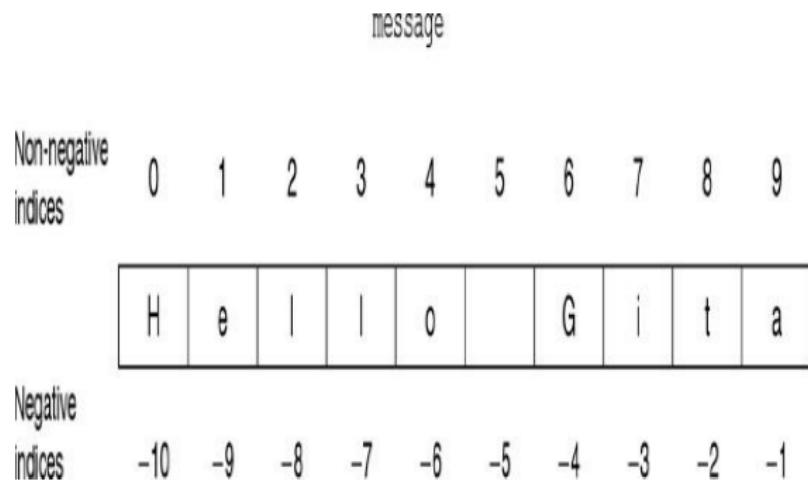


Figure 1: Indexing of variable **message**

## 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[:]
```

# Slicing

"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'
```

- 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()	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 lowercase letters
s.upper()	returns a string that has all lowercase letters in string s converted into corresponding uppercase letters
s.swapcase()	returns a string that has all lowercase letters in string s converted into uppercase letters and vice-versa

## 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 comprisedigits 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.)

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

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()
```

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

False

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

```
> > > mobileN = input('Enter mobile no : ')
Enter mobile no : 1234567890
> > > mobileN.isdigit() and len(mobileN) == 10
True
> > > '\n\t'.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'
```

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

```
>>> 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]	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
r1 r2	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	any non-alphanumeric character excluding _

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.)

$^a(a)(0 1)^*$	{a,a0,a1,a00,a01,...}, 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;</pre>

## Pattern Matching (Cont.)

```
match.group()  
'Shell'
```

## 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')

## Pattern Matching (Cont.)

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) >>>

## Pattern Matching (Cont.)

```
match.group()  
'23456'
```

## Pattern Matching (Cont.)

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

## Pattern Matching (Cont.)

```
>>> match.group()  
'I use **Python**, do you?'
```

## 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()

## Pattern Matching (Cont.)

'abac12ccaab'

## Pattern Matching (Cont.)

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

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-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'):  
    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

```
〉〉〉 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-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')  
[('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

Centre for Data Science, ITER  
Siksha ‘O’ Anusandhan (Deemed to be University),  
Bhubaneswar, Odisha, India.

# Contents

1 Introduction

2 Lists

- Two-dimensional Lists
- Lists as User-Input
- Lists Operations
- Lists Built-in Functions
- List Comprehension
- Lambda Expression

3 Sets

4 Tuples

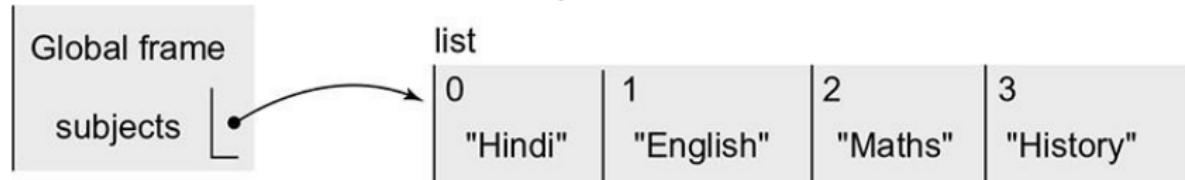
5 Dictionary

- **Mutable:** It is the ability of objects to change their values.
  - Example: List, Set and Dictionary.
- **Immutable:** If an object cannot be changed its values over time.
  - Example: Strings and Tuples.
- Elementary forms of data such as numeric and Boolean are called scalar data types.
- Several applications require more complex forms of data. For example;
  - 1 The name of a person
  - 2 A coordinates of a point
  - 3 A set of objects, or
  - 4 A list of personal records of individuals.

- A list is an ordered sequence of values. It is a non-scalar type.
- A list can store values of heterogeneous type:
  - 1 string
  - 2 integer
  - 3 float
  - 4 list
- **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.)

- Representation of the list “**subjects**” as in PythonTutor.



- Function `id()` returns object identifier for the list object **subjects**:

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

```
57135752
```

- Different variables may refer to the same list object

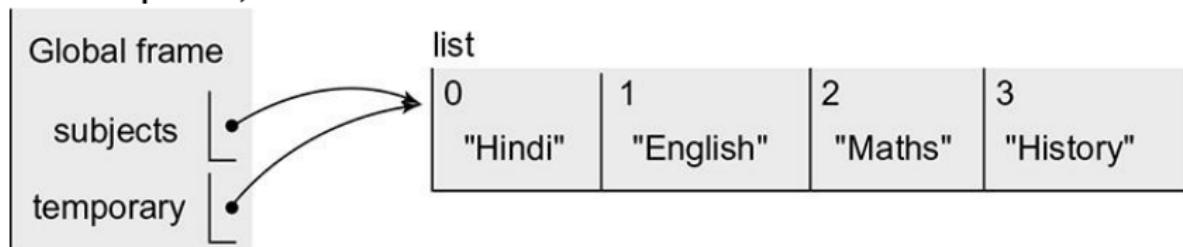
```
>>> temporary = subjects
```

```
>>> 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.)

- Modifying the component ***temporary*[0]** of the list ***temporary***, the change is also 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 Lists

- Two-dimensional list: List of lists
- Example: A list of subjects and their corresponding subject codes.

```
>>> 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 as User-Input

- 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:
- For example:

```
>>> 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 Operations

- Consider the following lists:

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

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

## Lists Operations (Cont.)

- Consider the following lists:

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

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

## Lists Operations (Cont.)

- The membership operator “in” may be used in a “for” loop for sequentially iterating over each element in the list.
- **Example 1:**

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

Ram

Shyam

Gita

Sita

## Lists Operations (Cont.)

- Example 2:

```
>>> markList = [78, 32, 89, 99, 56]
>>> count = 0
>>> for marks in markList:
    if(marks >= 75):
        count += 1
>>> count
3
```

# Lists Built-in Functions

- Many of these functions are used to modify the list. Example:

- 1 **append(e)**: Insert the element “e” at the end of list L.

Example:

```
>>> a = [10, 20, 30, 40]
>>> a.append(35)
>>> a
[10, 20, 30, 40, 35]
```

- 2 **extend(L2)**: Insert the item in sequence L2 at the end of list L.

Example:

```
>>> a = [10, 20, 30]
>>> a.extend([35, 40])
>>> a
[10, 20, 30, 35, 40]
>>> a.extend('abc')
>>> a
[10, 20, 30, 35, 40, 'a', 'b', 'c']
```

# Lists Built-in Functions

- Many of these functions are used to modify the list. Example:

- 1 **pop(i)**: Removes and return the element from the list L at index “i”.

```
>>> a = [10, 20, 30, 10, 50, 20, 60, 20, 30, 55]
```

```
>>> a.pop(3)
```

```
10
```

```
>>> a.pop(3)
```

```
50
```

```
>>> a
```

```
[10, 20, 30, 20, 60, 20, 30, 55]
```

- 2 **remove(e)**: Removes the first occurrence of the element e from the list.

```
>>> a.remove(20)
```

```
>>> a
```

```
[10, 30, 20, 60, 20, 30, 55]
```

- 3 **del**: Removes a subsequence of elements (*start:end:increment*) from the list.

```
>>> del a[2:6:2]
```

```
>>> a
```

```
[10, 30, 60, 30, 55]
```

# Lists Built-in Functions: Case Study

- ```
>>> names = ['Ram', 'Shyam', 'Sita', 'Gita', 'Sita']
>>> rollNums = [1, 2, 3, 4, 5]
```
- Remove a student entry “Shyam” from the lists:  

```
>>> rollNums.pop(names.index('Shyam'))
2
>>> names.remove('Shyam')
>>> print(names, rollNums)
['Ram', 'Sita', 'Gita', 'Sita'] [1, 3, 4, 5]
```

# Lists Built-in Functions

- Many of these functions are used to modify the list. Example:

- 1 **reverse()**: Reverses the order of elements in the list.
- 2 **sort()**: Sorts the elements of list.

**Example 1:**

```
>>> names = ['Ram', 'Sita', 'Gita', 'Anya']
>>> names.sort()
>>> names
['Anya', 'Gita', 'Ram', 'Sita']
```

**Example 2:**

```
>>> names.sort(reverse = True)
>>> names
['Sita', 'Ram', 'Gita', 'Anya']
```

- 3 **insert(i, e)**: Inserts element **e** at index **i** in list.

```
>>> names = ['Ram', 'Sita', 'Gita', 'Sita']
>>> names.insert(2, 'Shyam')
>>> names
['Ram', 'Sita', 'Shyam', 'Gita', 'Sita']
```

# Lists Built-in Functions

- Functions such as **count** and **index** do not modify the list.
  - ① **count(e)**: Returns count of occurrences of object e in the list L.

```
>>> a = [10, 30, 20, 60, 20, 30, 55, 20]
a.count(20)
3
```
  - ② **index(e)**: Returns index of an object e, if present in list.
- **dir(list)** outputs the list including all the functions that can be applied to objects of the type **list**.

# List Comprehension

- List comprehension provides a shorthand notation for creating lists.
- **Example:** Create a list that contains cubes of numbers ranging from 1 to 10.

```
>>> 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]
```

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

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

## List Comprehension (Cont.)

- Example: Create a list comprising of student whose height exceed or equal a threshold.

```
>>> lst = [['Rama', 160],  
           ['Anya', 159],  
           ['Mira', 158],  
           ['Sona', 161]]  
>>> threshold = 160  
>>> tall = [x for x in lst if x[1] >= threshold]  
>>> tall  
[['Rama', 160], ['Sona', 161]]
```

# 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     a[i] = value  
10  
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     lst = [10, 20, 30, [40, 50]]  
17     listUpdate(lst, 1, 15)  
18     print(lst)  
19  
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

- Lambda expression is used for defining a short and anonymous functions.
- Syntax for lambda expression:  
 $\text{lambda argument(s) : expression}$
- Lambda expression can take any number of arguments, but only a one expression.

- Example 1: lambda expression to compute the cube of a number

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

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

```
27
```

- Example 2: 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 lstCubes that are even.
- Thus, we need to **filter** the even elements from the list lstCubes
- 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.

- 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                                                       |
|----------------------------------|-------------------------------------------------------------------|
| <code>S.add(e)</code>            | Adds the elements to the set S, if not present already.           |
| <code>S1.update(L1)</code>       | Add the items in object L1 to the set S1, if not already present. |
| <code>S.remove(e)</code>         | Removes the element e from set S.                                 |
| <code>S.pop()</code>             | Removes an element from the set S.                                |
| <code>S.clear()</code>           | Removes all elements from the set S.                              |
| <code>S.copy()</code>            | Creates a copy of the set S.                                      |
| <code>S1.union(S2)</code>        | Returns union of the Sets S1 and S2.                              |
| <code>S1.intersection(S2)</code> | 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 1: 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<br>Operator * | >>> t1 * 2<br>('Monday', 'Tuesday', 'Monday', 'Tuesday')                   |
| Concatenation<br>Operator +  | >>> t3=t1 + ('Wednesday',)<br>>>> t3<br>('Monday', 'Tuesday', 'Wednesday') |
| Length<br>Operator len       | >>> len(t1)<br>2                                                           |
| Indexing                     | >>> t2[-2]<br>20                                                           |

# Tuple Operations

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

Table 2: 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 3: 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)])
```

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

# Dictionary Operations

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

| Operation                           | Examples                                                                                                                       |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| Membership operator <code>in</code> | <code>&gt;&gt;&gt; 5 in digits</code><br><code>True</code><br><code>&gt;&gt;&gt; 'Five' in digits</code><br><code>False</code> |

- 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

Table 5: 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                                                                                                    |

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

# References

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

Thank You  
Any Questions?