

Python

Python is a high-level, interpreted, interactive and object-oriented, scripting language.

Python is designed to be highly readable which uses English keywords frequently where as other languages use punctuation and it has fewer syntactical constructions than other languages.

Scripting language is a form of programming language that is usually interpreted rather than compiled. Conventional programs like c,c++ are converted permanently into executable files before they are run. In contrast, programs in scripting language are interpreted one command at a time.

1. Python is Interpreted:

This means that it is processed at runtime by the interpreter and you do not need to compile your program before executing it.

2. Python is Interactive:

This means that you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

3. Python is Object-Oriented:

This means that Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

4. Python is Beginner's Language:

Python is a great language for the beginner programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

1.Introduction

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, ALGOL-68, SmallTalk, and Unix shell and other scripting languages.

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python features

Python's feature highlights include:

- **Easy-to-learn:** Python has relatively few keywords, simple structure and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time.
- **Easy-to-maintain:** Python's success is that its source code is fairly easy-to-maintain.
- **A broad standard library:** One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows and Macintosh.
- **Interactive Mode:** Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Expandable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh and the X Window system of Unix.
- **Scalable:**
 - Python provides a better structure and support for large programs than shell scripting.
 - Apart from the above-mentioned features, Python has a big list of good features, few are listed below:
 - Support for functional and structured programming methods as well as OOP.
 - It can be used as a scripting language or can be compiled to byte-code for building large applications.
 - Very high-level dynamic data types and supports dynamic type checking.
 - Supports automatic garbage collection.
 - It can be easily integrated with C, C++, ActiveX, CORBA and Java.

Available online interpreters

Refer these links for online interpreters to execute python code,

Link 1: <http://codepad.org/>

Link 2: <http://www.pythontutor.com/visualize.html#mode=edit>

Link 3: <http://www.codeskulptor.org/>

A sample screen shot from pythontutor.com explaining execution of the program.

Command line Interpreter

Python has command line interpreter that helps in executing python commands by directly entering into python without writing a script.

Example:

If we type 1+2 the output would be 3.

The command line starts with >>> symbol as shown in examples,

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

```
4
```

```
>>> 50 - 5*6
```

```
20
```

```
>>> (50 - 5*6) / 4.0
```

```
5.0
```

```
>>> 8 / 5 # division always returns a floating point number
```

```
1.6
```

```
>>> width = 20
```

```
>>> height = 5 * 9
```

```
>>> width * height
```

```
900
```

Python on desktop

Refer to this link for downloading Python interpreter on your desktop.

<https://store.enthought.com/downloads/>

2. My first program in python

Objective:

- Simple program in Python
- Commenting in Python
- Quotations in python
- Lines and Indentation
- Multi-Line Statements

- [2.1 Simple program in Python](#)

- [2.2 Practise Problems](#)

2.1. Simple program in Python

Let us start with an example to understand basic program of python

```
print('Hello, world')
```

Here, print keyword is used to provide output on the screen.

The output is : Hello, world

The above program can also be written as shown below

```
print 'Hello, world'
```

The output of the above program is : Hello, world

Note: 1. It is optional to use parenthesis with print statement. In both conditions output will be same.

It is mandatory from python 3.x versions.

2. We can also end a statement with semicolon and it is optional again.

```
Ex: print 'Hello, world';
```

The semicolon (;) allows multiple statements on the single line

```
Ex: print 'hello'; print 'world';
```

Here is a sample script of python,

Sample script:

```
#!/usr/bin/python
#Printing on to the terminal
print "Hi this is my first python script"
```

To execute the sample script type python<space>filename.py

Execution: python sample.py

Commenting in Python

Single line comment

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them.

Example:

```
# Here you should write comments
```

```
print 'Hello, Python'
```

```
#This line is skipped by interpreter because it is after hash sign
```

Multi line comment

“""" or """ are used to write multi line comments if they are not inside a string literal.

Example 1:

```
"""  
  
This is a multi line  
comment  
  
"""  
  
print 'Hello, world'
```

Example 2:

```
"""  
  
This is a multi line  
comment  
  
"""  
  
print 'Hello, world'
```

Quotations in python

Python accepts single ('), double (") and triple ('' or ''') quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes can be used to span the string across multiple lines.

For example, all the following are legal

```
word = 'word'  
  
sentence = "This is a sentence."  
  
paragraph = """This is a paragraph. It is  
made up of multiple lines and sentences."""(Single quotes can also be used in place of  
double quotes).  
  
quote="Hi" #The string literal is 'Hi'
```

We can also write the same literal as shown below:

```
quote='\Hi\'
```

Note: Single and double quotes have no difference while using as a literal.

Line Indentation

One of the first cautions programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented with same amount of spaces

Block 1:

```
if True:
    print "True"
else:
    print "False"
```

However, the second block in this example will generate an error:

Block 2:

```
if True:
    print "Answer"
    print "True"
else:
    print "Answer"
    print "False"
```

Thus, in Python all the continuous lines indented with similar number of spaces would form a block.

Note: Use 4 spaces for indentation as a good programming practice.

Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

```
total = first_one + \
        second_two + \
        third_three
```

Statements contained within the [], {} or () brackets do not need to use the line continuation character.

For example:

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Problem statement

Write a program to get the following output?

```
$  
$$$  
$
```

Solution

The output would be as shown below,

```
print '$'  
print '$$$'  
print '$'
```


2.2. Practise Problems

1. Write a program to to print a sample text as shown,

This is Python

and hope you are a programmer.

Who wants to learn me?

2. Write a program to get the following output?

w w

 w w w

 w

 w w w

w w

The first and last lines have tab as space between them.

3.How to write multiple statements on a single line?

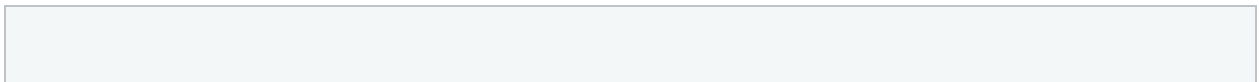
4.Write a program to provide following output

hello 'hi' bye 'why'

5.Write a program to show documented comments in the Program.

Please attempt these above problems.

Click [here](#) to download the solutions.



3.How user interact with the application? - I/O operations

Objective

- Input operations
- Output operations

- **3.1** Input operations

3.2 Output Operations

3.3 Practise Problems

3.1. Input operations

Python has two functions designed for accepting data directly from the user.

They are,

1.raw_input()

2.input()

1. raw_input()

raw_input() asks the user for a string of data (ended with a newline), and simply returns the string. It can also take an argument, which is displayed as a prompt before the user enters the data.

Example:

```
print (raw_input('Enter input'))
```

prints out Enter input <user input data here>

In order to assign the string data to a variable string1 you would type

```
string1=raw_input('Enter input')
```

After user inputs data e.g.hello. The value of string1 is hello

2. input()

input() uses raw_input to read a string of data, and then attempts to evaluate it as if it were a Python program, and then returns the value that results.

So entering 10 takes as an integer. But for raw_input() it is a string.

lambda expression:

Used to provide complex expressions to the input

```
lambda x:x+x,range(10)
```

Here expression converts x into x+x for all values in the range.

To provide complex expressions the input from user can be

```
map(lambda x: x*x, range(10))
```

Note: No inputted statement can span more than one line.

The difference between input() and raw_input() is shown in the example,

```
#!/usr/bin/python
```

```
x=raw_input('enter value')
```

```
print type(x)
```

Now,

if input x is 20 then the type(x) is a string.

```
#!/usr/bin/python
```

```
x=input('enter value')
```

```
print type(x)
```

Now,

if input x is 20 then the type(x) is an integer

Lets go through another example:

#For raw_input:

```
#!/usr/bin/python
```

```
str = raw_input("Enter your input: ");
```

```
print "Received input is : ", str
```

Execution:

Enter your input: Hello Python

Received input is : Hello Python

#For input:

```
#!/usr/bin/python
```

```
str = input("Enter your input: "); #complex program can be given as input.The input() function evaluates it.
```

```
print "Received input is : ", str
```

Execution:

Enter your input: [x*5 for x in range(2,10,2)]

Recieved input is : [10, 20, 30, 40]

3.2. Output Operations

1. `print()`

The basic way to do output is the print statement.

```
print 'Hello, world'
```

To print multiple things on the same line separated by spaces, use commas between them, like this:

```
print 'Hello,', 'World'
```

This will print out the following:

```
Hello, World
```

While neither string contain a space, a space was added by the print statement because of the comma between the two objects. Arbitrary data types can be printed this way:

```
print 1,2,(10+5j),-0.999
```

This will output the following:

```
1 2 (10+5j) -0.999
```

Objects can be printed on the same line without needing to be on the same line if one puts a comma at the end of a print statement:

```
for i in range(10):
```

```
    print i,
```

This will output the following:

```
0 1 2 3 4 5 6 7 8 9
```

To end the printed line with a newline, add a print statement without any objects.

```
for i in range(10):
```

```
    print i,
```

```
print
```

```
for i in range(10,20):
```

```
    print i,
```

This will output the following:

```
0 1 2 3 4 5 6 7 8 9
```

```
10 11 12 13 14 15 16 17 18 19
```

If the bare print statement were not present, the above output would look like:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Omitting newlines

To avoid adding spaces and newlines between objects' output with subsequent print statements, you can do one of the following:

Concatenation: Concatenate the string representations of each object, then later print the whole thing at once.

```
print str(1)+str(2)+str(10+5j)+str(-0.999)
```

This will output the following:

```
12(10+5j)-0.999
```

Write function: You can make a shorthand for sys.stdout.write and use that for output.

```
import sys  
  
write = sys.stdout.write  
  
write('20')  
  
write('05')
```

This will output the following:

```
2005
```

You may need sys.stdout.flush() to get that text on the screen quickly.

Examples:

Examples of output with Python 2.x:

```
print "Hello"
```

```
print "Hello", "world"
```

Separates the two words with a space.

```
print "Hello", 34
```

Prints elements of various data types, separating them by a space.

```
print "Hello " + 34
```

Throws an error as a result of trying to concatenate a string and an integer.

```
print "Hello " + str(34)
```

Uses "+" to concatenate strings, after converting a number to a string.

```
print "Hello",
```

Prints "Hello " without a newline, with a space at the end.

```
sys.stdout.write("Hello")
```

Prints "Hello" without a newline. Doing "import sys" is a prerequisite. Needs a subsequent "sys.stdout.flush()" in order to display immediately on the user's screen.

```
sys.stdout.write("Hello\n")
```

Prints "Hello" with a newline.

```
print >> sys.stderr, "An error occurred."
```

Prints to standard error stream.

```
sys.stderr.write("Hello\n")
```

Prints to standard error stream.

Examples of output with Python 3.x:

```
from __future__ import print_function
```

Ensures Python 2.6 and later Python 2.x can use Python 3.x print function.

```
print ("Hello", "world")
```

Prints the two words separated with a space. Notice the surrounding brackets, unused in Python 2.x.

```
print ("Hello world", end="")
```

Prints without the ending newline.

```
print ("Hello", "world", sep="-")
```

Prints the two words separated with a hyphen.

Problem statement

Write a program to convert Celsius input into Fahrenheit scale.

Solution

```
#!/usr/bin/python
```

```
x=float(input('enter Celsius input'))
```

```
y=x*9.0/5.0 + 32
```

```
print ("The Fahrenheit value is:",y)
```


3.3. Practise Problems

1. Write a program to print absolute value of a given number.
2. Write a program to find the type of the given input.
3. Write a program to take input and convert it into cube at input() function only and print the output.
4. Write a program to check whether given input is a prime number.
5. Write a program that prints output in the same line each value separated by -.

like 1-4-5-g-6-8.0 e.t.c.,

Please attempt these above problems.

Click [here](#) to download the solutions.

4. Basic Data Types

Objective

- Basic Data types
- Variables and Literals
- More on strings
- Type conversions

- [4.1 Basic Data types](#)

[4.2 Variables and Literals](#)[4.3 More on strings](#)[4.4 Type conversions](#)[4.5 Practise Problems](#)

4.1. Basic Data types

Type represents the kind of value and determines how the value can be used. All data values in Python are encapsulated in relevant object classes. Everything in Python is an object and every object has an identity, a type and a value.

To determine a variable's type in Python you can use the `type()` function.

1.Boolean

In Python programming language, the Boolean datatype is a primitive datatype having one of two values: True or False.

Keyword `bool` is used for boolean data type.

Examples of usage of boolean data type:

```
1.print bool(True)
2.print bool(False)
3.print bool("a")
4.print bool("")
5.print bool(' ') #space is literal here
6.print bool(0)
7.print bool()
8.print bool(3)
9.print bool(None)
10.print bool(3.0)
```

Ans:

```
1.True
2.False
3.True
4.False
5.True
6.False
7.False
8.True
```

9.False

10.True

2.Number

In Python programming language, we have integer numbers, floating point numbers, and complex numbers.

Here are some examples of these numbers,

integer: 103,-103

Keyword int is used for integers.

Long integer: 5678L (Long integers end with L)

Keyword long is used for long integer data type.

float: 103.0

Keyword float is used for float data type.

Complex : 3+4j

Keyword complex is used for complex data type.

3.String

Strings in programming are simply text, either individual characters, words, phrases, or complete sentences.

Keyword str is used for string data type

Example:

```
string="Python"
```

4. None

There is another special data type - None. Basically, this data type means non existent, not known or empty.

4.2. Variables and Literals

Variable:

Variable is a location in the memory where the value can be stored and the value can be change in time depending on the conditions or information passed to the program.

Literal:

Literal is a value or information provided to the variable.

Ex: In expression `x=3`

`x` is a variable, `3` is a literal.

Important points about variable assignment and its usage:

1. Declaration of variables is not required in Python. If there is need of a variable, you think of a name and start using it as a variable.

For example we can write a program as shown,

```
x=2
```

```
print x
```

prints 2 as output

2. Value of a variable may change during program execution and even the type can be changed. For example, you can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable.

3. keywords cannot be used as variable names.

List of keywords in python:

```
and    del    from    not    while  as    elif    global  or    with  assert  else    if    pass
yield  break   except  import  print  class  exec    in     raise  continue finally is    return  def
for    lambda  try
```

Python is a dynamic language. It changes during time. The list of keywords may change in the future.

To check the keywords of python version that you are working on use this statement,

```
print keyword.kwlist
```

`keyword.iskeyword(s)` returns true if `s` is a keyword.

Sample example to explain about first two points,

```
i = 42          # data type is implicitly set to integer
i = 42 + 0.11   # data type is changed to float
i = "string"    # and now it will be a string
```

Consider the assignment shown below,

```
x=3
y=x
y=2
```

The first assignment is unproblematic: Python chooses a memory location for x and saves the integer value 3. The second assignment is more worthwhile:

Intuitively, you may assume that Python will find another location for the variable y and will copy the value of 3 in this place. But Python goes his own way, which differs from our intuition and the ways of C and C++. As both variables will have the same value after the assignment, Python lets y point to the memory location of x.

The critical question arises in the next line of code. Y will be set to the integer value

What will happen to the value of x? C programmers will assume, that x will be changed to 2 as well, because we said before, that y "points" to the location of x. But this is not a C-pointer. Because x and y will not share the same value anymore, y gets his or her own memory location, containing 2 and x sticks to 3, as can be seen in the animated graphics on the right side.

But what we said before can't be determined by typing in those three lines of code. But how can we prove it? The identity function `id()` can be used for this purpose. Every instance (object or variable) has an identity, i.e. an integer which is unique within the script or program, i.e. other objects have different identities.

So, let's have a look at our previous example and how the identities will change:

```
#!/usr/bin/python
x = 3
print id(x)
157379912
y=x
```

```
print id(y)
```

```
157379912
```

```
y=2
```

```
print id(y)
```

```
157379924
```

You can see the change in id for y=2.

Note: id() is an operator used to find the location of variable.

4.3. More on strings

Special characters in strings

The backslash (\) character is used to introduce a special character. See the following table.

Escape sequence	Meaning
-----------------	---------

<code>\n</code>	Newline
-----------------	---------

<code>\t</code>	Horizontal Tab
-----------------	----------------

<code>\\</code>	Backslash
-----------------	-----------

<code>\'</code>	Single Quote
-----------------	--------------

<code>\"</code>	Double Quote
-----------------	--------------

String indices

Strings are arrays of characters and elements of an array can be accessed using indexing. Indices start with 0 from left side and -1 when start from right side.

Consider, `string1="PYTHON TUTORIAL"`

Following are the statements to access single character from various positions:

```
print(string1[0])    Ans:P
```

```
print(string1[-15])    Ans:P
```

```
print(string1[14])    Ans:L
```

```
print(string1[-1])    Ans:L
```

```
print(string1[4])    Ans:O
```

```
print(string1[15])    Ans:IndexError: string index out of range
```

'in' operator in Strings

The 'in' operator is used to check whether a character or a substring is present in a string or not. The expression returns a Boolean value.

Consider, string1="PYTHON TUTORIAL"

See the following statements:

'K' in string1 Ans:False

'P' in string1 Ans:True

'THO' in string1 Ans:True

'tho' in string1 Ans:False

String slicing

To cut a substring from a string is called string slicing. Here two indices are used separated by a colon (:). A slice 3:7 means indices characters of 3rd, 4th, 5th and 6th positions. The second integer index i.e. 7 is not included. You can use negative indices for slicing.

Consider, string1="PYTHON TUTORIAL"

See the following statements:

print(string1[0:5]) Ans:PYTHO

print(string1[3:7]) Ans:HON

print(string1[-4:-1]) Ans:RIA

4.4. Type conversions

Sometimes it's necessary to perform conversions between the built-in types. To convert between types you simply use the type name as a function. In addition, several built-in functions are supplied to perform special kind of conversions. All of these functions return a new object representing the converted value.

Function	Description
<code>int(x)</code>	Converts <code>x</code> to an integer.
<code>long(x)</code>	Converts <code>x</code> to a long integer.
<code>float(x)</code>	Converts <code>x</code> to a floating-point number.
<code>complex(real [, imag])</code>	Creates a complex number.
<code>str(x)</code>	Converts object <code>x</code> to a string representation.
<code>repr(x)</code>	Converts object <code>x</code> to an expression string.
<code>eval(str)</code>	Evaluates a string and returns an object.

Problem statement

```
#!/usr/bin/python
string1="hello friend welcome to python learning"
print string1[2:4]
print string1[-4:-1]
print string1[-1:-4]
print string1[:12]
print string1[12:]
```

output of above code will be?

Solution

The output would be as follows,

ll

nin

hello friend

welcome to python learning

4.5. Practise Problems

```
1.  a=4  
    b='string'  
    print a+b
```

what is the error in the above program? Please correct the error.

2. Find out the output and its type of the following statements.

```
int(5.0)  
float(5)  
str(7)  
int("")  
str(7.0)  
float("5.0")  
int("five")
```

3. Write a program to check a character in a string.

Note: Take a string of your wish.

4. Consider you have 22 apples,20 bananas,30 carrots. Create variables for each fruit and print the output as follows,

I have 22 apples 20 bananas 30 carrots.

Note:In place of number use the variable.

5. Answer the following questions.

- a) Which function used for the address of a variable?
- b) Which function used to know whether a given variable is keyword?
- c) Which function used to convert object into an expression string?
- d) Which function used to find maximum size that integer can take?

Please attempt these above problems.

Click [here](#) to download the solutions.

5.Operators supported for Data manipulation and Comparison

Objective

- Basic operators
- Operator precedence

5.1 Basic operators

5.2 Operator precedence

5.3 Practise Problems

5.1. Basic operators

What is an operator?

Simple answer can be given using expression $2 + 3$ is equal to 5. Here, 2 and 3 are called operands and + is called operator. Python language supports the following types of operators.

1. Arithmetic operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	$a + b$ will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	$a - b$ will give -10
*	Multiplication - Multiplies values on either side of the operator	$a * b$ will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	$b \% a$ will give 0
**	Exponent - Performs exponential (power) calculation on operators	$a ** b$ will give 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	$9 // 2$ is equal to 4 and $9.0 // 2.0$ is equal to 4.0

Example:

Try the following example to understand all the arithmetic operators available in Python programming language:

```
#!/usr/bin/python
```

```
a = 21
```

```
b = 10
```

```
c = 0
```

```
c = a + b
```

```
print "Line 1 - Value of c is ", c
```

```
c = a - b
```

```
print "Line 2 - Value of c is ", c
```

```
c = a * b
print "Line 3 - Value of c is ", c

c = a / b
print "Line 4 - Value of c is ", c

c = a % b
print "Line 5 - Value of c is ", c

a = 2
b = 3
c = a**b
print "Line 6 - Value of c is ", c

a = 10
b = 5
c = a//b
print "Line 7 - Value of c is ", c
```

When you execute the above program, it produces the following result:

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 8
Line 7 - Value of c is 2
```

2. Relational operators

Relational (comparison) operators always return a boolean result that indicates whether some relationship holds between their operands. Most relational operators are symbols (== != < > <= >=)The table below lists the relational operators and their descriptions.

Assume variable a holds 10 and variable b holds 20, then:

Example:

Try following example to understand all the relational operators available in Python programming language:

```
#!/usr/bin/python
```

```
a = 21
```

```
b = 10
```

```
c = 0
```

```
if ( a == b ):
```

```
    print "Line 1 - a is equal to b"
```

```
else:
```

```
    print "Line 1 - a is not equal to b"
```

```
if ( a != b ):
```

```
    print "Line 2 - a is not equal to b"
```

```
else:
```

```
    print "Line 2 - a is equal to b"
```

```
if ( a <> b ):
```

```
    print "Line 3 - a is not equal to b"
```

```
else:
```

```
    print "Line 3 - a is equal to b"
```

```
if ( a < b ):
```

```
    print "Line 4 - a is less than b"
```

```
else:
```

```
    print "Line 4 - a is not less than b"
```

```
if ( a > b ):
```

```
    print "Line 5 - a is greater than b"
```

```
else:
```

```
    print "Line 5 - a is not greater than b"
```

```
a = 5;
```

```
b = 20;
```

```
if ( a <= b ):
```

```
    print "Line 6 - a is either less than or equal to b"
```

else:

print "Line 6 - a is neither less than nor equal to b"

if (b >= a):

print "Line 7 - b is either greater than or equal to b"

else:

print "Line 7 - b is neither greater than nor equal to b"

When you execute the above program it produces the following result:

Line 1 - a is not equal to b

Line 2 - a is not equal to b

Line 3 - a is not equal to b

Line 4 - a is not less than b

Line 5 - a is greater than b

Line 6 - a is either less than or equal to b

Line 7 - b is either greater than or equal to b

3. Assignment operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assign value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	Floor Division and assigns a value, Performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Example:

Try following example to understand all the assignment operators available in Python programming language:

```
#!/usr/bin/python
```

```
a = 21
```

```
b = 10
```

```
c = 0
```

```
c = a + b
```

```
print "Line 1 - Value of c is ", c
```

```
c += a
```

```
print "Line 2 - Value of c is ", c
```

```
c *= a
```

```
print "Line 3 - Value of c is ", c
```

```
c /= a
```

```
print "Line 4 - Value of c is ", c
```

```

c = 2
c %= a
print "Line 5 - Value of c is ", c

c **= a
print "Line 6 - Value of c is ", c

c //= a
print "Line 7 - Value of c is ", c

```

When you execute the above program, it produces the following result:

```

Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864

```

4. Bitwise operators

Bitwise operator works on bits and perform bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows:

```

a = 0011 1100
b = 0000 1101
-----
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a  = 1100 0011

```

There are following Bitwise operators supported by Python language:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(a b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 111

Example:

Try following example to understand all the bitwise operators available in Python programming language:

```
#!/usr/bin/python

a = 60      # 60 = 0011 1100
b = 13      # 13 = 0000 1101
c = 0

c = a & b;   # 12 = 0000 1100
print "Line 1 - Value of c is ", c

c = a | b;   # 61 = 0011 1101
print "Line 2 - Value of c is ", c

c = a ^ b;   # 49 = 0011 0001
print "Line 3 - Value of c is ", c

c = ~a;      # -61 = 1100 0011
print "Line 4 - Value of c is ", c

c = a << 2;   # 240 = 1111 0000
print "Line 5 - Value of c is ", c
```

```
c = a >> 2;    # 15 = 0000 1111
print "Line 6 - Value of c is ", c
```

When you execute the above program it produces the following result:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

5. Logical operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.

Example:

Try the following example to understand all the logical operators available in Python programming language:

```
#!/usr/bin/python
a = 10
b = 20
c = 0

if ( a and b ):
    print "Line 1 - a and b are true"
else:
    print "Line 1 - Either a is not true or b is not true"
```

```
if ( a or b ):
    print "Line 2 - Either a is true or b is true or both are true"
else:
    print "Line 2 - Neither a is true nor b is true"
```

```
a = 0
if ( a and b ):
    print "Line 3 - a and b are true"
else:
    print "Line 3 - Either a is not true or b is not true"
```

```
if ( a or b ):
    print "Line 4 - Either a is true or b is true or both are true"
else:
    print "Line 4 - Neither a is true nor b is true"
```

```
if not( a and b ):
    print "Line 5 - Either a is not true or b is not true"
else:
    print "Line 5 - a and b are true"
```

When you execute the above program it produces the following result:

```
Line 1 - a and b are true
Line 2 - Either a is true or b is true or both are true
Line 3 - Either a is not true or b is not true
Line 4 - Either a is true or b is true or both are true
Line 5 - Either a is not true or b is not true
```

6. Membership operators

In addition to the operators discussed previously, Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators explained below:

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Example:

Try following example to understand all the membership operators available in Python programming language:

```
#!/usr/bin/python
a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
    print "Line 1 - a is available in the given list"
else:
    print "Line 1 - a is not available in the given list"

if ( b not in list ):
    print "Line 2 - b is not available in the given list"
else:
    print "Line 2 - b is available in the given list"

a = 2
if ( a in list ):
    print "Line 3 - a is available in the given list"
else:
    print "Line 3 - a is not available in the given list"
```

When you execute the above program it produces the following result:

Line 1 - a is not available in the given list

Line 2 - b is not available in the given list

Line 3 - a is available in the given list

7. Identity operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Example:

Try following example to understand all the identity operators available in Python programming language:

```
#!/usr/bin/python
```

```
a = 20
```

```
b = 20
```

```
if ( a is b ):
```

```
    print "Line 1 - a and b have same identity"
```

```
else:
```

```
    print "Line 1 - a and b do not have same identity"
```

```
if ( id(a) == id(b) ):
```

```
    print "Line 2 - a and b have same identity"
```

```
else:
```

```
    print "Line 2 - a and b do not have same identity"
```

```
b = 30
```

```
if ( a is b ):
```

```
    print "Line 3 - a and b have same identity"
```

```
else:
```

```
    print "Line 3 - a and b do not have same identity"
```

```
if ( a is not b ):
```

```
    print "Line 4 - a and b do not have same identity"
```

else:

```
    print "Line 4 - a and b have same identity"
```

When you execute the above program it produces the following result:

Line 1 - a and b have same identity

Line 2 - a and b have same identity

Line 3 - a and b do not have same identity

Line 4 - a and b do not have same identity

5.2. Operator precedence

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are + @ and - @)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ 	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Example:

Try the following example to understand operator precedence available in Python programming language :

```
#!/usr/bin/python
```

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

```
e = (a + b) * c / d      #( 30 * 15 ) / 5
```

```
print "Value of (a + b) * c / d is ", e
```

```
e = ((a + b) * c) / d    # (30 * 15) / 5
```

```
print "Value of ((a + b) * c) / d is ", e
```

```
e = (a + b) * (c / d);   # (30) * (15/5)
```

```
print "Value of (a + b) * (c / d) is ", e
```

```
e = a + (b * c) / d;    # 20 + (150/5)
```

```
print "Value of a + (b * c) / d is ", e
```

When you execute the above program, it produces the following result:

Value of $(a + b) * c / d$ is 90

Value of $((a + b) * c) / d$ is 90

Value of $(a + b) * (c / d)$ is 90

Value of $a + (b * c) / d$ is 50

Problem Statement

Consider values a,b,c,d,e .Now write a program to display output of $(a*b+c)*e+d$.

Take sample values for a,b,c,d,e.

Solution

```
a=4
```

```
b=5
```

```
c=4
```

```
d=2
```

```
e=6
```

```
print ((a*b+c)*e+d)
```

The output of the above code is 146.

5.3. Practise Problems

1. Consider $a=4, b=6, c=10$. Now write a program to have all the arithmetic operations(+, -, *, /, %, **, //) between them. For ex: $a+b+c$ and also find the output.
2. Consider $a=27, b=63$. Now write a program that has all bitwise operations and also find the output.
3. Consider $a=7, b=5$. Now, check whether a is greater than b if so increment it by 1.
4. Consider $a=5$ What is the output of the following $a \ll 3, a \gg 3, a \gg 2$ and $a \ll 2$.
5. A man has taken a debt of 200 R.S. Write a program to find simple interest after 2 years with rate of interest=3% and what is the Interest?

Please attempt these above problems.

Click [here](#) to download the solutions.

6. Controlling Program Flow - Conditional Statements

Objective

- If statement
- if...else statement
- if...elif...else statement
- Nested if

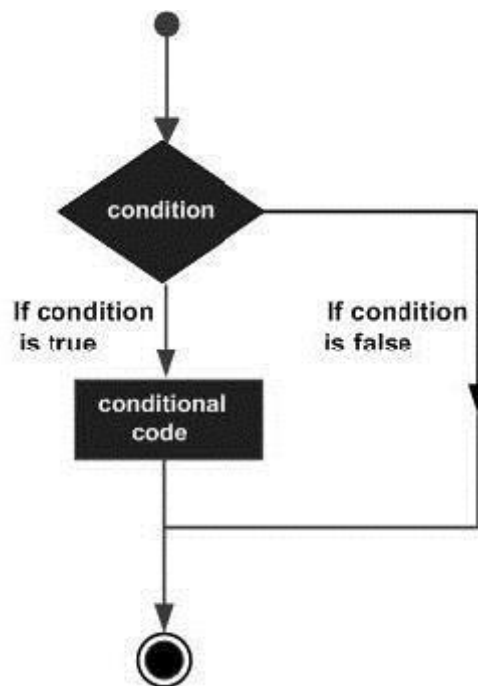
[6.1 Decision making structure](#)

[6.2 Practise Problems](#)

6.1. Decision making structure

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



Python programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

Python programming language provides following types of decision making statements.

1. If statement

The if statement of Python is similar to that of other languages. The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax:

The syntax of an if statement in Python programming language is:

if expression:

statement(s)

If the boolean expression evaluates to true, then the block of statement(s) inside the if statement will be executed. If boolean expression evaluates to false, then the first set of code after the end of the if statement(s) will be executed.

Python programming language assumes any non-zero and non-null values as true and if it is either zero or null, then it is assumed as false value.

Example:

```
#!/usr/bin/python
var1 = 100
if var1:
    print 1
    print var1

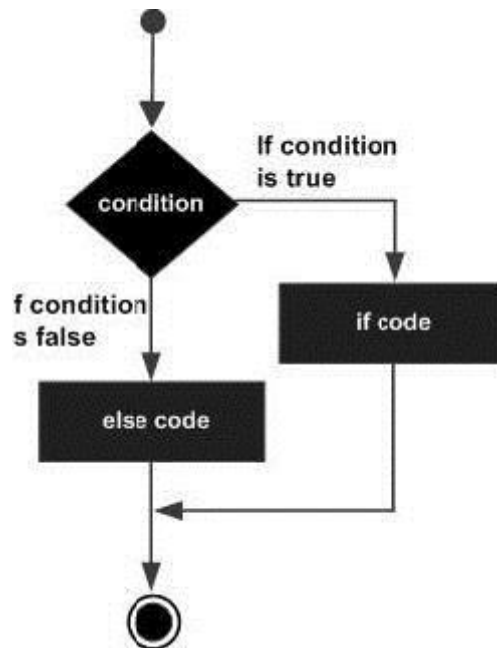
var2 = 0
if var2:
    print 2
    print var2
print "Good bye!"
```

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

```
1
100
Good bye!
```

2. If...else statement

An else statement can be combined with an if statement. An else statement



contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a false value.

The else statement is an optional statement and there could be at most only one else statement following if.

Syntax:

The syntax of the if...else statement is:

if expression:

statement(s)

else:

statement(s)

Example:

```
#!/usr/bin/python
```

```
var1 = 100
```

```
if var1:
```

```
    print "1 - Got a true expression value"
```

```
    print var1
```

```
else:
```

```
    print "1 - Got a false expression value"
```

```
    print var1
```

```
var2 = 0

if var2:
    print "2 - Got a true expression value"
    print var2
else:
    print "2 - Got a false expression value"
    print var2

print "Good bye!"
```

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

```
1 - Got a true expression value
100
2 - Got a false expression value
0
Good bye!
```

3. If...elif...else statement

The elif statement allows you to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true.

Like the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

The syntax of the if...elif statement is:

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```


Note: Core Python does not provide switch or case statements as in other languages, but we can use if...elif...statements to simulate switch case as follows:

Example:

```
#!/usr/bin/python
var = 100
if var == 200:
    print "1 - Got a true expression value"
    print var
elif var == 150:
    print "2 - Got a true expression value"
    print var
elif var == 100:
    print "3 - Got a true expression value"
    print var
else:
    print "4 - Got a false expression value"
    print var

print "Good bye!"
```

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

```
3 - Got a true expression value
100
Good bye!
```

4. Nested if condition

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct.

In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

Syntax:

The syntax of the nested if...elif...else construct may be:

```
if expression1:
    statement(s)
```

```
if expression2:
    statement(s)
elif expression3:
    statement(s)
else
    statement(s)
elif expression4:
    statement(s)
else:
    statement(s)
```

Example:

```
#!/usr/bin/python
var = 100
if var < 200:
    print "Expression value is less than 200"
    if var == 150:
        print "Which is 150"
    elif var == 100:
        print "Which is 100"
    elif var == 50:
        print "Which is 50"
elif var < 50:
    print "Expression value is less than 50"
else:
    print "Could not find true expression"
print "Good bye!"
```

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

Expression value is less than 200

Which is 100

Good bye!

Problem statement

Create a program with python that calculate the cost for shipping.

Solution

```
#!/usr/bin/python

total = int(raw_input('What is the total amount for your online shopping?'))

country = raw_input('Shipping within the US or Canada?')

if country == "US":
    if total <= 50:
        print "Shipping Costs $6.00"
    elif total <= 100:
        print "Shipping Costs $9.00"
    elif total <= 150:
        print "Shipping Costs $12.00"
    else:
        print "FREE"
if country == "Canada":
    if total <= 50:
        print "Shipping Costs $8.00"
    elif total <= 100:
        print "Shipping Costs $12.00"
    elif total <= 150:
        print "Shipping Costs $15.00"
    else:
        print "FREE"
```

6.2. Practise Problems

CONTROLLING PROGRAM FLOW – CONDITIONAL STATEMENTS

1. Write a program to find maximum of three numbers.
2. Write a program to check whether input provided is odd or even.
3. Write a program to compare two numbers provided as input.
4. Write a program to find absolute value of a given number.
5. Write a program to find whether given number is a prime number.

Please attempt these above problems.

Click [here](#) to download the solutions.

7.How to execute a block of statements repetitively – Loop Structures

Objective

- While loop
- for loop
- else statement used with loops
- break statement
- continue statement
- pass statement

7.1 Loop statement

7.2 Loop control statements

7.3 Problem statement

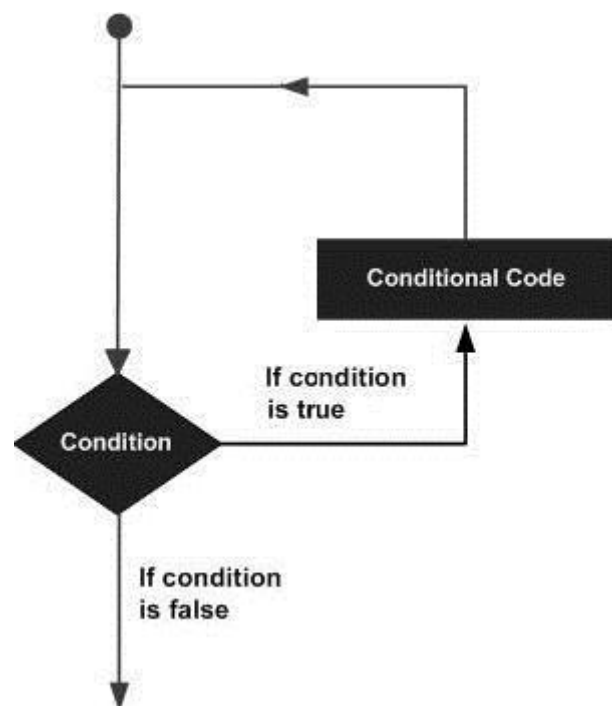
7.4 Practise Problems

7.1. Loop statement

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Python programming language provides following types of loops to handle looping requirements:

1. While loop
2. for loop
3. Loop control statements

1. while loop

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true. Unlike the for loop, there is no built-in end counter. It is up to the programmer to make sure that the given condition stops being True at some point, or else the loop will become an infinite loop.

Syntax:

The syntax of a while loop in Python programming language is:

while expression:

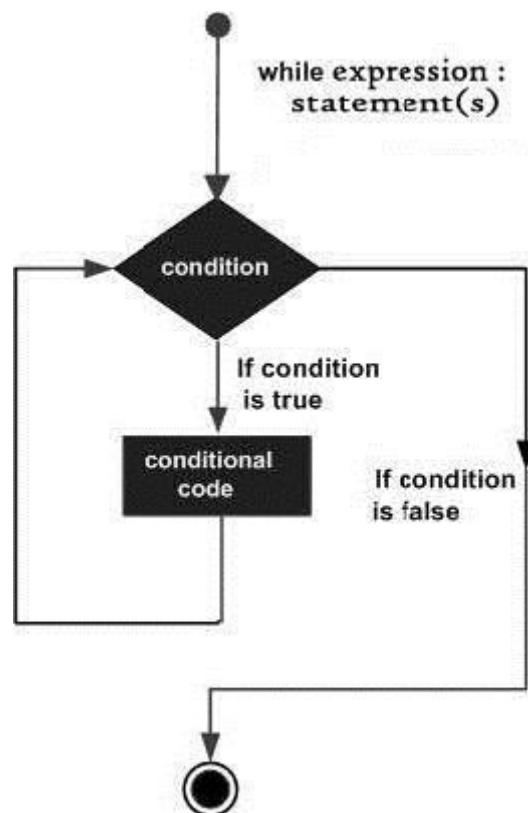
 statement(s)

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram:



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
#!/usr/bin/python  
a = 0  
while a < 5:  
    a += 1 # Same as a = a + 1  
    print (a)
```

And here is the output:

```
1  
2  
3  
4  
5
```

2. for loop

The for loop in Python has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

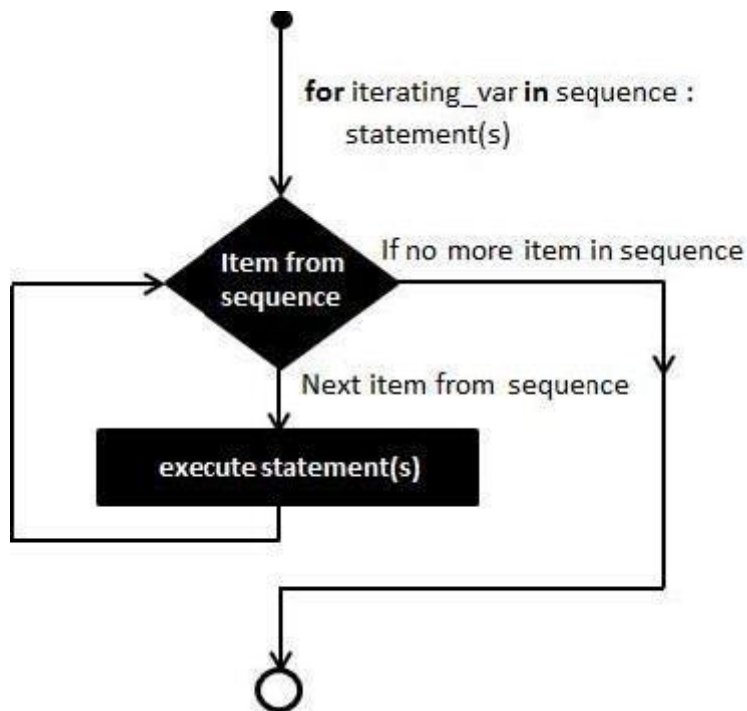
The syntax of a for loop look is as follows:

```
for iterating_var in sequence:
```

```
    statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable `iterating_var`. Next, the statements block is executed. Each item in the list is assigned to `iterating_var`, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram:



Example:

```
#!/usr/bin/python
```

```
list = [2,4,6,8]
```

```
sum = 0
```

```
for num in list:
```

```
    sum = sum + num
```

```
print ("The sum is: %d" % sum)
```

with the output simply being:

The sum is: 20

For / Range Loops:

The `range()` function creates an arithmetic progression as a list. The for loop can use range to loop from a beginning integer to an ending integer. The default increment or step value is 1, but you can set this to any positive or negative integer. Here is the syntax:

```
range([start,]stop[,step])
```

Example:

Here is some code to print out the first 9 numbers of the Fibonacci series:

```
#!/usr/bin/python
```

```
a = 1
```

```
b = 1
for c in range(1,10):
    print (a)
    n = a + b
    a = b
    b = n
print ("")
```

with the surprising output :

```
1
1
2
3
5
8
13
21
34
```

Everything that can be done with for loops can also be done with while loops but for loops give an easy way to go through all the elements in a list or to do something a certain number of times.

Else statement used with loops

Python supports to have an else statement associated with a loop statement.

If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

Example1: else with for loop

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
#!/usr/bin/python
for num in range(10,20): #to iterate between 10 to 20
```

```

for i in range(2,num): #to iterate on the factors of the number
    if num%i == 0:      #to determine the first factor
        j=num/i        #to calculate the second factor
        print '%d equals %d * %d' % (num,i,j)
        break #to move to the next number, the #first FOR
else:                  # else part of the loop
    print num, 'is a prime number'

```

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

```

10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number

```

Example2: else with while loop

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```

#!/usr/bin/python
count = 0
while count < 5:
    print count, " is less than 5"
    count = count + 1
else:
    print count, " is not less than 5"

```

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

```

0 is less than 5

```

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5

5 is not less than 5

7.2. Loop control statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements

1. break statement
2. continue
3. pass

1.break statement

The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for a break statement in Python is as follows:

```
break
```

Example:

```
#!/usr/bin/python
for letter in 'Python':
    if letter == 'h':
        break
    print 'Current Letter :', letter
```

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

Current Letter : P

Current Letter : y

Current Letter : t

2. continue statement

The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and for loops.

Syntax:

The syntax for a continue statement in Python is as follows:

```
continue
```

Example:

```
#!/usr/bin/python
for letter in 'Python':
    if letter == 'h':
        continue
    print 'Current Letter :', letter
```

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

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

3. pass statement

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet for ex. in stubs.

The syntax for a pass statement in Python is as follows:

```
pass
```

Example:

```
#!/usr/bin/python
for letter in 'Python':
```

```
if letter == 'h':  
    pass  
    print 'This is pass block'  
print 'Current Letter :', letter  
  
print "Good bye!"
```

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

```
Current Letter : P  
Current Letter : y  
Current Letter : t  
This is pass blocks  
Current Letter : h  
Current Letter : o  
Current Letter : n
```

7.3. Problem statement

Write a code to print Fibonacci series

Solution

```
#!/usr/bin/python

# Program to display the Fibonacci sequence up to nth term where n is provided by the user

# take input from the user
nterms = int(input("How many terms? "))

# first two terms
n1 = 0
n2 = 1
count = 2

# check if the number of terms is valid
if nterms <= 0:
    print("Plese enter a positive integer")
elif nterms == 1:
    print("Fibonacci sequence:")
    print(n1)
else:
    print ("Fibonacci sequence:")
    print (n1, ", ", n2, end=" , ")

    while count < nterms:
        nth = n1 + n2

        if count !=(nterms-1):
            print(nth,end=" , ")# update values
        else:
            print (nth)

        n1 = n2
        n2 = nth
        count += 1
```

Let us take input as 10 ,The output would be as follows,

How many terms? 10

Fibonacci sequence:

0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34

7.4. Practise Problems

1. Write a program to print even numbers less than 100
2. Write a program to print prime numbers less than 100 using for else statement
3. Write a program to print output as following

^ ^ ^ ^ ^

^ ^ ^ ^

^ ^ ^

^ ^

^

^ ^

^ ^ ^

^ ^ ^ ^

^ ^ ^ ^ ^

4. Write a program to find whether a given number is palindrome or not
5. Write a program to display multiplication table for a given number

Ex: number is 12

Then $12*1=12$

$12*2=24$

and so on.

Please attempt these above problems.

Click [here](#) to download the solutions.

8.Operations on String Variables

Objective

- Inbuilt string methods
- String formatting operations

8.1 Inbuilt String Methods

8.2 String Formatting Operations

8.3 Problem Statement

8.4 Practise Problems

8.1. Inbuilt String Methods

Python's built-in string methods are incredibly powerful. As the name implies, each of the following methods are available through class string. Every string object is an instance of that class and has these methods available.

1. **str.capitalize()**

Returns a copy of the string with its first character capitalized and the rest lowercased.

Example:

```
#!/usr/bin/python
a="string"
print a.capitalize()
```

Output:

String

2. **str.upper()**

Returns a copy of string with all capital letters.

Example:

```
#!/usr/bin/python
a="string"
print a.upper()
```

Output:

STRING

3. **str.lower()**

Returns a copy of string with all small letters.

Example:

```
#!/usr/bin/python
a="String"
print a.lower()
```

Output:

string

4. str.count(sub[, start[, end]])

Returns the number of non-overlapping occurrences of substring sub in the range [start, end].

Optional arguments start and end are interpreted as in string slice notation.

Example:

```
#!/usr/bin/python
a="String string String string String"
print a.count("String")
```

Output:

3

5. str.index(sub[, start[, end]])

Returns the lowest index in the string where substring sub is found, such that sub is contained in the slice s[start:end]. Optional arguments start and end are interpreted as in slice notation.

Example:

```
#!/usr/bin/python
a="String string String string String"
print (a.index("String",3))
```

Output:

14

6. str.endswith(suffix[, start[, end]])

Returns True if the string ends with the specified suffix, otherwise return False.

suffix can also be a tuple of suffixes to look for. With optional start, test beginning at that position. With optional end, stop comparing at that position.

Example 1:

```
#!/usr/bin/python
a="String string String string String"
print (a.endswith("String"))
```

Output:

True

Example 2:

```
#!/usr/bin/python  
a="String string String string String"  
print (a.endswith("String",0,27))
```

Output:

False

7. str.expandtabs([tabsize])

Returns a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every tabsize characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

Example:

```
#!/usr/bin/python  
a="String\tstring\tString\tstring\tString"  
print ("1. " +a)  
print ("2. " +a.expandtabs(1))
```

Output:

1. String string String string String
2. String string String string String

8. str.find(sub[, start[, end]])

Returns the lowest index in the string where substring sub is found, such that sub is contained in the slice s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found.

Example:

```
#!/usr/bin/python  
a="String string String string String"  
print (a.find("string"))
```

Output:

7

9. str.isalnum()

Returns true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

Example:

```
#!/usr/bin/python  
a="String string String string String3"  
b="StringstringStringstringString3"  
print (a.isalnum())  
print (b.isalnum())
```

Output:

False

True

10. str.isalpha()

Returns true if all characters in the string are alphabetic and there is at least one character, false otherwise.

Example:

```
#!/usr/bin/python  
a="StringstringStringstringString3"  
b="StringstringStringstringString"  
print (a.isalpha())  
print (b.isalpha())
```

Output:

False

True

11. str.isdigit()

Returns true if all characters in the string are digits and there is at least one character, false otherwise.

Example:

```
#!/usr/bin/python
a="StringstringStringstringString3"
b="33434"
print (a.isdigit())
print (b.isdigit())
```

Output:

```
False
True
```

12. str.title()

Returns a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

Example:

```
#!/usr/bin/python
a="string string string string string"
print (a.title())
```

Output:

```
String String String String String
```

13. str.islower()

Returns true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Example:

```
#!/usr/bin/python
a="string String string string string"
print (a.islower())
```


Output:

False

14. str.isspace()

Returns true if there are only whitespace characters in the string and there is at least one character, false otherwise.

Example:

```
#!/usr/bin/python  
b=" "  
print (b.isspace())
```

Output:

True

15. str.istitle()

Returns true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

Example:

```
#!/usr/bin/python  
a="string String string string string"  
print (a.istitle())
```

Output:

False

16. str.isupper()

Returns true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

Example:

```
#!/usr/bin/python  
a="string String string string string"  
print (a.isupper())
```

Output:

False

17. str.partition(sep)

Split the string at the first occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple(will be discussed in further course)containing the string itself, followed by two empty strings.

Example:

```
#!/usr/bin/python
a="string String string string string"
print (a.partition(" "))
```

Output:

('string', ' ', 'String string string string')

18. str.replace(old, new[, count])

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

Example:

```
#!/usr/bin/python
a="string String1 string2 string3 string"
print (a.replace("string","wing"))
```

Output:

wing String1 wing2 wing3 wing

Example:

```
#!/usr/bin/python
a="string String1 string2 string3 string"
print (a.replace("string","wing",1))
```

Output:

wing String1 string2 string3 string

19. str.strip([chars])

Returns a copy of the string with the leading and trailing characters removed. The chars argument is a

string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing white space.

Example:

```
#!/usr/bin/python  
  
a=" strip "  
  
print (a.strip())
```

Output:

strip

20. str.split([sep[, maxsplit]])

Returns a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

If sep is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',') returns ['1', '', '2']). The sep argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Splitting an empty string with a specified separator returns [""].

If sep is not specified or is None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a None separator returns [].

For example, ' 1 2 3 '.split() returns ['1', '2', '3'], and ' 1 2 3 '.split(None, 1) returns ['1', '2 3 '].

21. str.startswith(prefix[, start[, end]])

Returns True if string starts with the prefix, otherwise return False. prefix can also be a tuple of prefixes to look for. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.

Example 1:

```
#!/usr/bin/python  
  
a="String string String string String"  
  
print (a.startswith("String"))
```

Output:

True

22. str.swapcase()

Returns a copy of the string with uppercase characters converted to lowercase and vice versa.

Example:

```
#!/usr/bin/python  
  
a="string String string string"  
  
print (a.swapcase())
```

Output:

STRING sTRING STRING STRING

23. str.translate(table[, deletechars])

Returns a copy of the string where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

Example:

```
#!/usr/bin/python  
  
a="read this text"  
  
print a.translate(None,'aeiou')
```

Output:

rd ths txt

8.2. String Formatting Operations

1. % Operator

Directive	Interactive Session
<code>%s</code> Represents a value as a string	<pre>>>> "%s" % list ['hi', 1, 1.0, 1L] >>> "list equals %s" % list list equals ['hi', 1, 1.0, 1L]</pre>
<code>%i</code> Integer	<pre>>>> "i = %i" % (5) 'i = 5' >>> "i = %3i" % (5) 'i = 5'</pre>
<code>%d</code> Decimal integer	<pre>>>> "d = %d" % 5 'd = 5' >>> "%3d" % (3) ' 3'</pre>
<code>%x</code> Hexadecimal integer	<pre>>>> "%x" % (0xff) 'ff' >>> "%x" % (255)</pre>
<code>%o</code> Octal integer	<pre>>>> "%o" % (255) 377 >>> "%o" % (0377) 377</pre>
<code>%u</code> Unsigned integer	<pre>>>> print "%u" % -2000 2147481648 >>> print "%u" % 2000 2000</pre>
<code>%e</code> Float exponent	<pre>>>> print "%e" % (300000000L) 3.000000e+007 >>> "%5.2e" % (3000000000L)</pre>
<code>%f</code> Float	<pre>>>> "check = %1.2f" % (3000) 'check = 3000.00' >>> "payment = \$%1.2f" % 3000 'payment = \$3000.00'</pre>
<code>%g</code> Float exponent	<pre>>>> "%3.3g" % 100 '100.' >>> "%3.3g" % 10000000000000L '10.e11' >>> "%g" % 100 '100.'</pre>
<code>%c</code> ASCII character	<pre>>>> "%c" % (97) 'a' >>> "%c" % 97</pre>

Example1:

```
#!/usr/bin/python
print "%i, %f, %e" % (1000, 1000, 1000)
```

Output:

```
1000, 1000.000000, 1.000000e+03
```

Example2:

```
pwd='secret'
uid='student'
print "%s is not a good password for %s" % (pwd, uid)
```

Output:

secret is not a good password for student

2. format() operator

format() operator supports different operations as shown below:

2.1 Accessing arguments by position

Let us understand this with an example,

```
print 'hello {0}, {1}, {2}'.format('a', 'b', 'c')
```

It prints : hello a,b,c

Here, we are assigning a,b and c to the string that we want to format.

Let us take another example,

```
print 'hello {2}, {1}, {0}'.format('a', 'b', 'c')
```

Output would be hello c,b,a as we changed the positions of indices in the string.

Note: Argument indices can be repeated.

2.2 Accessing arguments by name

The formatting can also be done with name as index.

Example:

```
#!/usr/bin/python
print 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
```

Results the output as shown,

Coordinates: 37.24N, -115.81W

2.3 Accessing arguments' attributes

For example `c=3-5j`

```
print ('The complex number {0} is formed from the real part {0.real} and the imaginary part {0.imag}.').format(c)
```

We can access the attributes of complex number i.e., real and imaginary part as shown in above example.

8.3. Problem Statement

Write a program to find whether given string is a palindrome

Solution

```
# Program to check if a string
#!/usr/bin/python
# is palindrome or not
# take input from the user
my_str = input("Enter a string: ")
# make it suitable for caseless comparison
my_str = my_str.lower()
# reverse the string
rev_str = reversed(my_str)
# check if the string is equal to its reverse
if list(my_str) == list(rev_str):
    print("It is palindrome")
else:
    print("It is not palindrome")
```

Let us give input as, aibia then output would be It is a palindrome.

8.4. Practise Problems

1. Write a program to compare two strings the program should not consider case(means case insensitive). The output would be "Both are same " if they are same otherwise "Both are not same".
2. Consider,a string x="This is python programmer learning python". Write a program to get sub string "python"and check how many times the string is repeated.
3. Write a program to trim the leading and trialing spaces in a string taken from user as input.
4. Write a program that adds and multiplies two complex numbers and shows output on screen as a complex number and also real and imaginary numbers separately.
5. Write a program to reverse a string without using reversed() inbuilt function.

Please attempt these above problems.

Click [here](#) to download the solutions.

9. Additional Data Storage Objects

Objective

- Lists
- Tuples
- Sets
- Dictionaries

9.1 Lists

9.2 Tuples

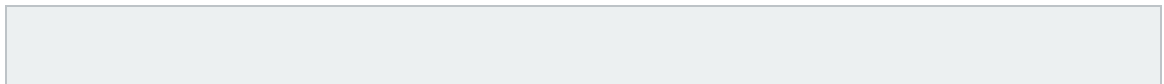
9.3 Differences between list and tuple

9.4 Dictionaries

9.5 Sets

9.6 Problem statement

9.7 Practise Problems



9.1. Lists

Overview

The list type is a container that holds a number of other objects, in a given order. The list type implements the sequence protocol, and also allows you to add and remove objects from the sequence.

Creating Lists

To create a list, put a number of expressions in square brackets:

```
L = [ ]
```

```
L = [expression, ...]
```

Example:

```
#!/usr/bin/python
```

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

```
list1 = ['physics', 'chemistry', 1997, 2000]
```

```
list2 = [1, 2, 3, 4, 5 ]
```

```
list3 = ["a", "b", "c", "d"]
```

```
listinlist=[1,2,[3,4,5],4,5]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Lists

Lists implement the standard sequence interface; `len(L)` returns the number of items in the list, `L[i]` returns the item at index `i` (the first item has index 0), and `L[i:j]` returns a new list, containing the objects between `i` and `j`.

Example:

```
list1 = ['india','australia','south africa','west indies']
```

```
print list1[0],"has brighter chances to win the world cup"
```

```
print "But may face competition from",list1[2]
```

Output of the program is:

```
india has brighter chances to win the world cup
```

```
But may face competition from south africa
```

Looping over Lists

The for-in statement makes it easy to loop over the items in a list:

Example:

```
#!/usr/bin/python
list1 = ['physics', 'chemistry', 1997, 2000]
for element in list1:
    print element
```

Output:

```
physics
chemistry
1997
2000
```

If you need only the index, use range and len:

To understand this here is an example,

```
#!/usr/bin/python
list1 = ['physics', 'chemistry', 1997, 2000]
for index in range(len(list1)):
    print index
```

Output is:

```
0 1 2 3
```

Modifying Lists

We can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. Following is a simple example:

Example:

```
#!/usr/bin/python
List1=['a','b',1,'c']
List1[2]='d'
print List1
```

Output of the program is:

```
['a', 'b', 'd', 'c']
```

We can also delete an element from a list by using del operator.

Example:

```
#!/usr/bin/python
```

```
List1=['a','b',1,'c']
```

```
del List1[2]
```

```
print List1
```

Output:

```
['a', 'b', 'c']
```

Basic List operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings.

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

Built-in functions

Python includes the following list functions:

1.cmp(list1, list2)

Compares elements of both lists.

2.len(list)

Gives the total length of the list.

3.max(list)

Returns item from the list with max value.

4.min(list)

Returns item from the list with min value.

5.list(seq)

Converts a tuple into list.

Built-in methods

Python includes following list methods

1.list.append(obj)

Appends object obj to list

2.list.count(obj)

Returns count of how many times obj occurs in list

3.list.extend(seq)

Appends the contents of seq to list

4.list.index(obj)

Returns the lowest index in list that obj appears

5.list.insert(index, obj)

Inserts object obj into list at offset index

6.list.pop(obj=list[-1])

Removes and returns last object or obj from list

7.list.remove(obj)

Removes object the obj from list

8.list.reverse()

Reverses the objects of list in place

9.list.sort()

Sorts the objects of list

9.2. Tuples

Overview

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The only difference is that tuples can't be changed i.e., tuples are immutable and tuples use parentheses and lists use square brackets.

Creating tuples

Creating a tuple is as simple as putting different comma-separated values and optionally you can put these comma-separated values between parentheses also.

For example:

```
tup1 = ('p', 'c', 19, 20)
```

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

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

```
tup4 = 'a','b','c','d'
```

The empty tuple is written as two parentheses containing nothing:

```
tup1 = ()
```

To write a tuple containing a single value you have to include a comma, even though there is only one value to differentiate it with other data types:

```
tup1 = (50,)
```

Accessing tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. Following is a simple example:

```
#!/usr/bin/python
```

```
t=(1,2,3)
```

```
print t[0]
```

```
print t+(t[0],)
```

Output would be:

```
1
```

```
(1, 2, 3, 1)
```

Modifying tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples.

Example:

```
#!/usr/bin/python
t=(1,2,3,4)
t[0]=10      #not a valid operation on tuple
t2=t+t       #This is possible
and t2 would be (1,2,3,4,1,2,3,4)
```

Removing individual tuple elements is not possible. But, using del operator it is possible to delete whole tuple object.

Example:

```
#!/usr/bin/python
t=(103,6527,10)
del t
print t
```

Output:

Name Error: name 't' is not defined.

Basic tuple operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

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

Built-in tuple functions

Python includes following tuple functions

1.cmp(tuple1, tuple2)

Compares elements of both tuples.

2.len(tuple)

Gives the total length of the tuple.

3.max(tuple)

Returns item from the tuple with max value.

4.min(tuple)

Returns item from the tuple with min value.

5.tuple(seq)

Converts a list into tuple

9.3. Differences between list and tuple

1. Size

```
a = tuple(range(1000))
```

```
b = list(range(1000))
```

```
a.__sizeof__() # 8024
```

```
b.__sizeof__() # 9088
```

Due to the smaller size of a tuple operation with it a bit faster but not that much to mention about until you have a huge amount of elements.

2. Permitted operations

```
b = [1,2]
```

```
b[0] = 3    # [3, 2]
```

```
a = (1,2)
```

```
a[0] = 3    # Error
```

that also mean that you can't delete element or sort tuple.

At the same time you could add new element to both list and tuple with the only difference that you will change id of the tuple by adding element

```
a = (1,2)
```

```
b = [1,2]
```

```
id(a)      # 140230916716520
```

```
id(b)      # 748527696
```

```
a += (3,)  # (1, 2, 3)
```

```
b += [3]   # [1, 2, 3]
```

```
id(a)      # 140230916878160
```

```
id(b)      # 748527696
```

Usage

3. Appending to list and tuple

```
a = (1,2)
```

```
b = [1,2]
```

```
a.append(3)  #Error(Tuple has no attribute appened.
```

```
b.append(3)  #[1,2,3]
```

4. You can't use list as a dictionary identifier

```
a = (1,2)
```

```
b = [1,2]
```

```
c = {a: 1}  # OK
```

```
c = {b: 1}  # Error
```

9.4. Dictionaries

A dictionary is mutable and is another container type that can store any number of Python objects, including other container types. Dictionaries consist of pairs (called items) of keys and their corresponding values.

Python dictionaries are also known as associative arrays or hash tables. The general notation of a dictionary is as follows:

```
diction = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

The things on left side of ":" are keys and right side are values.

Note:

Keys of a particular dictionary are unique while values may not be.

The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Operations on dictionaries

Consider below notation as an example,

```
d={1:'one',2:'two',3:'three',4:'four'}
```

Now to extract keys from dictionary Keys() operator can be used as shown below

```
d.keys()
```

Then it results output as shown

```
[1,2,3,4]
```

In the same way to extract values from dictionary values() can be used as shown

```
d.values()
```

Then it results output as shown

```
['one','two','three','four']
```

1. Adding new elements to a dictionary

To add new element to a dictionary assign a value by providing a key as shown

```
d[5]='five'
```

now,d contains five pairs of keys and values.

The elements of d are as follows:

```
{1:'one',2:'two',3:'three',4:'four',5:'five'}
```

2. Deleting a key from a dictionary

del operator is used to delete a key and corresponding value from the dictionary

example:

```
d={1:'one',2:'two',3:'three',4:'four',5:'five'}
```

```
del d[2]
```

```
print d
```

Output:

```
{1: 'one', 3: 'three', 4: 'four', 5: 'five'}
```

3. has_key() operator

We can check the existence of a key by using has_key operator

example:

```
#!/usr/bin/python
```

```
d={1:'one',2:'two',3:'three',4:'four',5:'five'}
```

```
print d.has_key(1)
```

```
print d.has_key(6)
```

Output would be as shown,

```
True
```

```
False
```

We can also use condition as shown below to check whether key exists,

```
1 in d      #True as 1 is a key contained in d
```

```
6 in d      #False as 6 is not available in keys of dictionary d
```

4. Copying a dictionary

A dictionary can be copied with method copy()

Example:

```
#!/usr/bin/python
```

```
d={1:'one',2:'two',3:'three',4:'four',5:'five'}
```

```
w=d.copy()
```

```
print w
```

Output would be,

```
{1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}
```

5. Merging two dictionaries

What about concatenating dictionaries, like we did with lists? There is something similar for dictionaries: the update method `update()` merges the keys and values of one dictionary into another, overwriting values of the same key:

Example:

```
#!/usr/bin/python
```

```
d={1:'one',2:'two',3:'three',4:'four',5:'five'}
```

```
w={2:'to',8:'eight'}
```

```
d.update(w)
```

```
print d
```

Output would be,

```
{1: 'one', 2: 'to', 3: 'three', 4: 'four', 5: 'five', 8: 'eight'}
```

6. Clearing contents of dictionary

The content of a dictionary can be cleared with the method `clear()`. The dictionary is not deleted, but set to an empty dictionary:

Example:

```
#!/usr/bin/python
```

```
d={1:'one',2:'two',3:'three',4:'four',5:'five'}
```

```
d.clear()
```

```
print d
```

Output would be

```
{}
```

7. Connection between Lists and Dictionaries

If we have a dictionary

```
{"list": "List", "dictionary": "dict", "function": "Func"}
```

we could turn this into a list with two-tuples:

```
[("list","List"), ("dictionary","dict"), ("function","Func")]
```

Example:

```
#!/usr/bin/python
```

```
d={1:'one',2:'two',3:'three',4:'four',5:'five'}
```

```
print d.items()
```

Output would be,

```
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four'), (5, 'five')]
```

9.5. Sets

A set is a dictionary with no values. It has only unique keys. Its syntax is similar to that for a dictionary. But we omit the values, and can use complex, powerful set logic.

1. Creating a set

General syntax for set creation is as follows:

```
set1={'a','b','c'}
```

We can also create a set using set() function by providing parameters as list.

Example:

```
set1=set([1,2])
```

Note : Use set() operator for creating empty set ,variable with empty {} will be considered as a dictionary.

2. Modifying sets

This example explains about the usage of sets.

```
#!/usr/bin/python
```

```
set1 = set()      # A new empty set
```

```
set1.add("cat")
```

```
''' Adds a single member
```

```
(We can't add several members using add operator)
```

```
'''
```

```
set1.update(["dog", "mouse"]) # Adds several members
```

```
if "cat" in set1:      # Membership test
```

```
    set1.remove("cat") #removes string literal "cat"
```

```
#set1.remove("elephant") - throws an error as there is no "elephant" in set list
```

```
print set1
```

```
for item in set1:      # Iteration on sets
```

```
    print item
```

```
print "Item count:", len(set1) # size of set1 is printed as output
```

```
len(set1) == 0      # Tests if set is empty
```

```
set1 = set(["cat", "dog"]) # Initialize set from a list
```

```
set2 = set(["dog", "mouse"])
```



```
set3 = set1 & set2          # Intersection of sets
set31=set1.intersection(set2) # same as above
set4 = set1 | set2          # Union of sets
set41=set1.union(set2)      #same as above
set5 = set1 - set3          # Set difference
set6 = set1.difference(set3) #output would be same as set5
set1 <= set2  # Subset test(returns true if set1 is a subset of set2)
set1.issubset(set2) #same as set1<=set2
set1 >= set2 # Superset test(returns true if set1 is a super of set2)
set1.issuperset(set2) #same as set1>=set2
set8 = set1.copy()          # set1 is copied to set8
set8.clear()                # Clears all the elements of set8
```

9.6. Problem statement

Consider the following,

```
animals = { 'a':['art'], 'b': ['balloon'], 'c': ['coat'],'d':['den','dog','deer']}
```

write a function to find the key that has maximum values Ex: maxani(animals) output is 'd'.

Solution

```
#!/usr/bin/python
```

```
animals = { 'a': ['art'], 'b': ['balloon'], 'c': ['coat'], 'd': ['den','dog','deer']}
```

```
def biggest(aDict):
```

```
    result = None
```

```
    biggestValue = 0
```

```
    for key in aDict.keys():
```

```
        if len(aDict[key]) >= biggestValue:
```

```
            result = key
```

```
            biggestValue = len(aDict[key])
```

```
    return result
```

```
print (biggest(animals))
```

9.7. Practise Problems

1. Consider the dictionary mentioned below:

```
dic={'a':'act','b':'battle','c':'cart','d':'diction'}
```

Write a program to print values of dic in different lines.

2. Write a program to find all the subsets of the set $x=\{1,2,3,4\}$

3. Consider a tuple, $x=('s','y','e')$ and a dictionary, $d={'a':'act','b':'battle','y':'cart','s':'diction'}$.

Write a program to find number of tuple values that are keys in dictionary.

4. Write a program to find number of keys in a dictionary.

5. Write a program to find a value in a given list.

Please attempt these above problems.

Click [here](#) to download the solutions.

10.Modularity and Code reusability – Functions

Objective

- Defining a function
- Calling a function
- Pass by reference
- Function arguments
- Return statement
- Scope of variables
- Recursion

10.1 Functions in Python

10.2 Pass by reference

10.3 Function arguments

10.4 Return statement and Scope of Variables

10.5 Problem statement

10.6 Practise Problems

10.1. Functions in Python

A function is a piece of code in a program. The function performs a specific task. The advantages of using functions are:

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

Functions in Python are first-class citizens. It means that functions have equal status with other objects in Python. Functions can be assigned to variables, stored in collections or passed as arguments. This brings additional flexibility to the language.

There are two basic types of functions. Built-in functions and user defined ones. The built-in functions are part of the Python language. Examples are: `dir()`, `len()` or `abs()`. The user defined functions are functions created with the `def` keyword.

Defining a function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

Function blocks begin with the keyword `def` followed by the function name and parentheses `()`.

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement - the documentation string of the function or docstring.

The code block within every function starts with a colon `:` and is indented.

The statement `return [expression]` exits a function, optionally passing back an expression to the caller.

A return statement with no arguments is the same as `return None`.

Syntax:

```
def functionname( parameters ):
    "function_docstring"
```

```
function_suite  
    return [expression]
```

By default, parameters have a positional behaviour and you need to inform them in the same order that they were defined.

Example:

here is a function that prints the words "hello" on screen, and then returns the number '1234' to the main program:

```
#!/usr/bin/python  
  
def hello():  
    print "hello"  
    return 1234
```

Calling a function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call hello() function:

Below is the function

```
def hello():  
    print "hello"  
    return 1234
```

And here is the function being used

```
print hello()
```

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

```
hello  
1234
```

Accessing python documented comments

We can access the documented comments by using `__doc__` attribute as shown below,

```
def func(param):  
    """This is a documented function
```

```
with some comments.
```

```
'''
```

```
pass
```

```
print func.__doc__
```

Output:

This is a documented function

with some comments.

10.2. Pass by reference

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function, whereas pass by value will not reflect back.

Python is “pass-by-object-reference”.

Example:

```
#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

    "This changes a passed list into this function"

    mylist.append([1,2,3,4]);

    print "Values inside the function: ", mylist

    return


# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object.

So, this would produce the following result:

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition is here

def changeme( mylist ):

    "This changes a passed list into this function"

    mylist = [1,2,3,4]; # This would assign new reference in mylist

    print "Values inside the function: ", mylist

    return
```



```
# Now you can call changeme function  
mylist = [10,20,30];  
changeme( mylist );  
print "Values outside the function: ", mylist
```

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result:

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

10.3. Function arguments

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

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

1. Required arguments:

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

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

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

printme();
```

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

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

2. Keyword arguments:

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

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

```
#!/usr/bin/python

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

printme( str = "My string");
```

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

My string

Following example gives more clear picture. Note, here order of the parameter does not matter.

```
#!/usr/bin/python

def printinfo( name, age ):

    "This prints a passed info into this function"

    print "Name: ", name;

    print "Age ", age;

    return;

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

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

Name: miki

Age 50

3. Default arguments:

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. Following example gives an idea on default arguments, it would print default age if it is not passed.

```
#!/usr/bin/python

def printinfo( name, age = 35 ):

    "This prints a passed info into this function"

    print "Name: ", name;

    print "Age ", age;

    return;

# Now you can call printinfo function

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

printinfo( name="miki" );
```

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

Name: miki

Age 50

Name: miki

Age 35

4. Variable-length arguments:

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

The general syntax for a function with non-keyword variable arguments is this:

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An asterisk (*) is placed before the variable name that will hold the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Following is a simple example:

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

# Now you can call printinfo function
printinfo( 10 );
printinfo( 70, 60, 50 );
```

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

Output is:

10

Output is:

70

60

50

10.4. Return statement and Scope of Variables

Return statement

A function is created to do a specific task. Often there is a result from such a task. The return keyword is used to return values from a function. A function may or may not return a value. If a function does not have a return keyword, it will send a None value.

All the above examples are not returning any value, but if you like you can return a value from a function as follows:

```
# Function definition is here
```

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

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

```
    total = arg1 + arg2
```

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

```
    return total;
```

```
# Now you can call sum function
```

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

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

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

```
Inside the function : 30
```

```
Outside the function : 30
```

Scope of variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

- Global variables
- Local variables

Global vs. Local variables:

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example:

```
name = "Jack"

def f():
    name = "Robert"
    print "Within function", name
print "Outside function", name
f()
```

A variable defined in a function body has a local scope. It is valid only within the body of the function.

Output will be:

Outside function Jack

Within function Robert

By default, we can get the contents of a global variable inside the body of a function. But if we want to change a global variable in a function, we must use the global keyword.

Recursion

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of recursive function to find the factorial of an integer. Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is $1*2*3*4*5*6 = 720$.

```
#!/usr/bin/python

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

```
num = int(raw_input("Enter a number: "))
```

```
if num >= 1:
```

```
    print("The factorial of ", num, "is", fact(num))
```

output will be:

Enter a number: 4

The factorial of 4 is 24 .

10.5. Problem statement

Write a function to sort the given list using selection sort.

Solution

```
#!/usr/bin/python
def selSort(L):
    for i in range(len(L) - 1):
        minIndx = i
        minVal = L[i]
        j = i+1
        while j < len(L):
            if minVal > L[j]:
                minIndx = j
                minVal = L[j]
            j += 1
        if minIndx != i:
            temp = L[i]
            L[i] = L[minIndx]
            L[minIndx] = temp
    return L
print selSort([4,2,1,3])
```

The output of the above code is as follows [1,2,3,4]

10.6. Practise Problems

1. Write a function for finding square of a number.
2. Write a function that prints hello before the input provided by user for ex:func('siddu') should print output as Hello siddu.
3. Write a function whichType() so that it should print following outputs.

whichType(4)

This is int

whichType(4.0)

This is float

whichType('hi')

This is string

4. Write a merge sort function to sort a given list.
5. Write a function to convert seconds into minutes For ex:convertMin(100) should give output as
1 minute 40 seconds

Please attempt these above problems.

Click [here](#) to download the solutions.

11.Importing Blocks and Code Modules

Objective

- Import statement
- from...import statement
- from...import * statement

11.1 Import Statement

11.2 Problem statement

11.3 Practise Problems



11.1. Import Statement

The basic purpose of the import statement is to:

- Identify one of the following items
- An ordinary module
- A package
- A module within a package
- An attribute of a module or package

Bind information about this external item to a variable local to the current module.

Code in the current module will then use this local-module variable to interact with the external item

Python provides at least three different ways to import modules. You can use the import statement, the from statement, or the built-in `__import__` function.

These are:

- `import x` -- imports the module X, and creates a reference to that module in the current namespace. x may be a module or package
- `from x import b` – imports the module X, and creates references in the current namespace to the given objects. Or in other words, you can now use a in your program. x may be a module or package; b can be a contained module or object (class, function etc.)
- `From x import *` - imports the module X, and creates references in the current namespace to all public objects defined by that module
- Finally, `X = __import__('X')` works like `import X`, with the difference that you

1) pass the module name as a string,

and 2) explicitly assign it to a variable in your current namespace.

Import statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax:

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module `hello.py`, you need to put the following command at the top of the script:

Here's an example of a simple module, hello..py

```
#!/usr/bin/python

def print_func( par ):

    print "Hello : ", par

    return
```

now if we want to use this in another program then we can just write as below:

```
# Import module support

import support

# Now you can call defined function that module as follows

support.print_func("Sarah")
```

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

Hello : Sarah

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

Example:

```
vi mod.py

#!/usr/bin/python

def func():

    print "This is sample function"

vi mod1.py

#!/usr/bin/python

#Importing the module mod.py

import mod

mod.func()
```

After execution of mod1.py as python mod1.py the output will be:

This is sample function

From...import statement

Python's from statement lets you import specific attributes from a module into the current namespace.

The from...import has the following syntax:

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function fibonacci from the module fib, use the following statement:

```
from fib import fibonacci
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

Example:

```
vi mod2.py
#!/usr/bin/python
from mod import func
func()
```

After execution of mod2.py as python mod2.py the output will be:

This is sample function

From...import * statement

It is also possible to import all names from a module into the current namespace by using the following import statement:

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace.

Example:

```
vi mod3.py
from mod import *
func()
```

It will import every function from module into the current namespace.

dir() function:

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module.

Following is a simple example:

```
#!/usr/bin/python
# Import built-in module math
```

```
import math
content = dir(math)
print content;
```

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

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Here, the special string variable `__name__` is the module's name, and `__file__` is the filename from which the module was loaded.

11.2. Problem statement

Find whether random function generated is in range of 25 or 50 or 75 or 100

Solution

```
#!/usr/bin/python

import random

i=random.randrange(0,100) #random number generated will be from 0 to 100.

print " random number is "

print i

if i<25:

    print " This value is less than 25"

elif i<50:

    print " This value is less than 50 greater than 25"

elif i<75:

    print " This value is less than 75 grater than 50"

else:

    print " This value is less than 100 greater than 75"
```


11.3. Practise Problems

1. Write a function to find the area of square.Import the module area and find volume of the cube with the area.(height shouldn't be defined).
2. Write a program with three functions add,subtract and multiply.Import add and multiply into a program which displays results of those functions.
3. Generate a random value from x to y.Here take x and y from user.

Please attempt these above problems.

Click [here](#) to download the solutions.

12.File Handling Operations

Objective

- Opening and Closing Files
- Reading and Writing Files
- Renaming and Deleting Files
- Directories in Python

12.1 Opening and Closing Files

12.2 Reading and Writing Files

12.3 Renaming and Deleting Files

12.4 Directories in Python

12.5 Problem statement

12.6 Practise Problems

12.1. Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to play with actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do your most of the file manipulation using a file object.

1. The open Function

Before you can read or write a file, you have to open it using Python's built-in `open()` function. This function creates a file object, which would be utilized to call other support methods associated with it.

SYNTAX:

```
file object = open(file_name [, access_mode][, buffering])
```

Here is parameters' detail,

file_name: The `file_name` argument is a string value that contains the name of the file that you want to access.

access_mode: The `access_mode` determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

buffering: If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default(default behaviour).

Here is a list of the different modes of opening a file:

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

2. The file object attributes

Once a file is opened and you have one file object, you can get various information related to that file.

Here is a list of all attributes related to file object:

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

EXAMPLE:

```
#!/usr/bin/python

# Open a file

fo = open("foo.txt", "wb")

print "Name of the file: ", fo.name

print "Closed or not : ", fo.closed

print "Opening mode : ", fo.mode

print "Soft space flag : ", fo.softspace
```

This would produce the following result:

```
Name of the file: foo.txt

Closed or not : False

Opening mode : wb

Soft space flag : 0
```

3. The close() Method

The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

SYNTAX:

```
fileObject.close();
```

EXAMPLE:

```
# Open a file

fo = open("foo.txt", "wb")

print "Name of the file: ", fo.name


# Close opened file

fo.close()
```

This would produce the following result:

```
Name of the file: foo.txt
```

12.2. Reading and Writing Files

The file object provides a set of access methods to make our lives easier. We would see how to use `read()` and `write()` methods to read and write files.

1. The `write()` Method

The `write()` method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The `write()` method does not add a newline character (`'\n'`) to the end of the string:

SYNTAX:

```
fileObject.write(string);
```

Here, passed parameter is the content to be written into the opened file.

EXAMPLE:

```
#!/usr/bin/python
# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\n Yeah its great!!\n");

# Close opened file
fo.close()
```

The above method would create `foo.txt` file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

Python is a great language.

Yeah its great!!

2. The `read()` Method

The `read()` method reads a string from an open file. It is important to note that Python strings can have binary data and not just text.

SYNTAX:

```
fileObject.read([count]);
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts

reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

EXAMPLE:

Let's take a file foo.txt, which we have created above.

```
#!/usr/bin/python
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opened file
fo.close()
```

This would produce the following result:

Read String is : Python is

3. File Positions

The tell() method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The seek(offset[, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

If from is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

EXAMPLE:

Let's take a file foo.txt, which we have created above.

```
#!/usr/bin/python
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
```

```
# Check current position
position = fo.tell();
print "Current file position : ", position

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str

# Close opened file
fo.close()
```

This would produce the following result:

Read String is : Python is

Current file position : 10

Again read String is : Python is

12.3. Renaming and Deleting Files

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

1. The rename() Method

The rename() method takes two arguments, the current file name and the new file name.

SYNTAX:

```
os.rename(current_file_name, new_file_name)
```

EXAMPLE:

Following is the example to rename an existing file test1.txt:

```
import os

# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

2. The remove() Method

You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.

SYNTAX:

```
os.remove(file_name)
```

EXAMPLE:

Following is the example to delete an existing file test2.txt:

```
import os

# Delete file test2.txt
os.remove("text2.txt")
```

12.4. Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The `os` module has several methods that help you create, remove and change directories.

1. The `mkdir()` Method

You can use the `mkdir()` method of the `os` module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

SYNTAX:

```
os.mkdir("newdir")
```

EXAMPLE:

Following is the example to create a directory `test` in the current directory:

```
import os

# Create a directory "test"
os.mkdir("test")
```

2. The `chdir()` Method

You can use the `chdir()` method to change the current directory. The `chdir()` method takes an argument, which is the name of the directory that you want to make the current directory.

SYNTAX:

```
os.chdir("newdir")
```

EXAMPLE:

Following is the example to go into `/home/newdir` directory:

```
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

3. The `getcwd()` Method

The `getcwd()` method displays the current working directory.

SYNTAX:

```
os.getcwd()
```

EXAMPLE:

Following is the example to give current directory:

```
import os

# This would give location of the current directory
os.getcwd()
```

4. The rmdir() Method

The rmdir() method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

SYNTAX:

```
os.rmdir('dirname')
```

EXAMPLE:

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

```
#!/usr/bin/python

import os

# This would remove "/tmp/test" directory.
os.rmdir( "/tmp/test" )
```

12.5. Problem statement

Write a program to read from a file and write to a file simultaneously.

Solution

```
#!/usr/bin/python
fobj_in = open("file1.txt")
fobj_out = open("file2.txt", "w")
i = 1
for line in fobj_in:
    print(line.rstrip())
    fobj_out.write(str(i) + ": " + line)
    i = i + 1
fobj_in.close()
fobj_out.close()
```

12.6. Practise Problems

1. Write a program to open a file and append the content to the file and close the file.
2. Write a program to display the content of a file on the screen.
3. Write a program that creates a directory and adds files to that directory.
4. Write a program to Create a directory with name pyt and rename it to python and delete it.
5. Write a program to display the content of a file except the first 10 bytes of the file.

Note: Create a sample file on desktop and have some content on it.

Please attempt these above problems.

Click [here](#) to download the solutions.

13.Handling runtime errors - Exception Handling

Objective

- Exception
- Handling exceptions
- Raising exceptions
- User – defined exceptions

13.1Exception

13.2Raising exceptions and User defined Exceptions

13.3Problem statement

13.4Practise Problems

13.1. Exception

What is Exception?

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
```

Traceback (most recent call last):

File "<stdin>", line 1, in ?

ZeroDivisionError: integer division or modulo by zero

```
>>> 4 + spam*3
```

Traceback (most recent call last):

File "<stdin>", line 1, in ?

NameError: name 'spam' is not defined

```
>>> '2' + 2
```

Traceback (most recent call last):

File "<stdin>", line 1, in ?

TypeError: cannot concatenate 'str' and 'int' objects

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Built-in Exceptions lists the built-in exceptions and their meanings.

Handling exceptions

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

SYNTAX:

Here is simple syntax of try....except...else blocks:

```
#!/usr/bin/python
```

```
try:
```

```
    You do your operations here;
```

```
    .....
```

```
except Exception1:
```

```
    If there is Exception1, then execute this block.
```

```
except Exception2:
```

```
    If there is Exception2, then execute this block.
```

```
    .....
```

```
else:
```

```
    If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax:

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

EXAMPLE:

Here is simple example, which opens a file and writes the content in the file and comes out gracefully because there is no problem at all:

```
#!/usr/bin/python
```

```
try:
```

```
    fh = open("testfile", "w")
```

```
    fh.write("This is my test file for exception handling!!")
```



```
except IOError:
```

```
    print "Error: can't find file or read data"
```

```
else:
```

```
    print "Written content in the file successfully"
```

```
    fh.close()
```

This will produce the following result:

Written content in the file successfully.

EXAMPLE:

Here is one more simple example, which tries to open a file where you do not have permission to write in the file, so it raises an exception:

```
#!/usr/bin/python
```

```
try:
```

```
    fh = open("testfile", "r")
```

```
    fh.write("This is my test file for exception handling!!")
```

```
except IOError:
```

```
    print "Error: can't find file or read data"
```

```
else:
```

```
    print "Written content in the file successfully:"
```

This will produce the following result:

Error: can't find file or read data

The except clause with no exceptions:

You can also use the except statement with no exceptions defined as follows:

```
try:
```

```
    You do your operations here;
```

```
    .....
```

```
except:
```

```
    If there is any exception, then execute this block.
```

```
    .....
```

```
else:
```

```
    If there is no exception then execute this block.
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

Example:

```
# import module sys to get the type of exception
#!/usr/bin/python
import sys

while True:
    try:
        x = int(input("Enter an integer: "))
        r = 1/x
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Please try again.")
        print()

print("The reciprocal of",x,"is",r)
```

Here is a sample run of this program.

Enter an integer: 1.3

Oops! <class 'ValueError'> occured.

Please try again.

Enter an integer: 0

Oops! <class 'ZeroDivisionError'> occured.

Please try again.

Enter an integer: 2

The reciprocal of 2 is 0.5

In this program, we loop until the user enters an integer that has a valid reciprocal. The portion that can cause exception is placed inside try block. If no exception occurs, except block is skipped and normal flow continues. But if any exception occurs, it is caught by the except block. Here, we print the

name of the exception using `ex_info()` function inside `sys` module and ask the user to try again. We can see that the values 'a' and '1.3' causes `ValueError` and '0' causes `ZeroDivisionError`.

The except clause with multiple exceptions:

You can also use the same `except` statement to handle multiple exceptions as follows:

try:

 You do your operations here;

except(Exception1[, Exception2[,...ExceptionN]]):

 If there is any exception from the given exception list,
 then execute this block.

else:

 If there is no exception then execute this block.

Example:

```
#!/usr/bin/python
```

try:

```
    f = open('integers.txt')
```

```
    s = f.readline()
```

```
    i = int(s.strip())
```

except (IOError, ValueError):

```
    print("An I/O error or a ValueError occurred")
```

except:

```
    print("An unexpected error occurred")
```

```
    raise
```

The try-finally clause:

You can use a `finally:` block along with a `try:` block. The `finally` block is a place to put any code that must execute, whether the `try`-block raised an exception or not.

The syntax of the `try-finally` statement is this:

try:

 You do your operations here;

 Due to any exception, this may be skipped.

finally:

 This would always be executed.

Note that you can provide except clause(s), or a finally clause, but not both. You can not use else clause as well along with a finally clause.

EXAMPLE:

```
#!/usr/bin/python
```

try:

```
    fh = open("testfile", "w")
```

```
    fh.write("This is my test file for exception handling!!")
```

finally:

```
    print "Error: can't find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result:

```
Error: can't find file or read data
```

Same example can be written more cleanly as follows:

try:

```
    fh = open("testfile", "w")
```

try:

```
    fh.write("This is my test file for exception handling!!")
```

finally:

```
    print "Going to close the file"
```

```
    fh.close()
```

except IOError:

```
    print "Error: can't find file or read data"
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

Argument of an Exception:

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows:

try:

 You do your operations here;

except ExceptionType, Argument:

 You can print value of Argument here...

If you are writing the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable will receive the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

EXAMPLE:

Following is an example for a single exception:

```
#!/usr/bin/python
```

```
# Define a function here.
```

```
def temp_convert(var):
```

```
    try:
```

```
        return int(var)
```

```
    except ValueError, Argument:
```

```
        print "The argument does not contain numbers\n", Argument
```

```
# Call above function here.
```

```
temp_convert("xyz");
```

This would produce the following result:

The argument does not contain numbers

invalid literal for int() with base 10: 'xyz'

13.2. Raising exceptions and User defined Exceptions

Raising exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement.

SYNTAX:

```
raise [Exception [, args [, traceback]]]
```

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

EXAMPLE: An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows:

```
#!/usr/bin/python

def functionName( level ):

    if level < 1:

        raise "Invalid level!", level

        # The code below to this would not be executed

        # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write our except clause as follows:

```
try:

    Business Logic here...

except "Invalid level!":

    Exception handling here...

else:

    Rest of the code here...
```

User-defined exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to RuntimeError.

Here, a class is created that is sub classed from RuntimeError.

This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

```
class Networkerror(RuntimeError):
```

```
    def __init__(self, arg):
```

```
        self.args = arg
```

So once you defined above class, you can raise your exception as follows:

```
try:
```

```
    raise Networkerror("Bad hostname")
```

```
except Networkerror,e:
```

```
    print e.args
```


13.3. Problem statement

Write a program to find reciprocal of x and it shouldn't show any error for any input value.

Solution

```
#!/usr/bin/python
```

```
try:
```

```
    x = float(raw_input("Your number: "))
```

```
    inverse = 1.0 / x
```

```
    print inverse
```

```
except ValueError:
```

```
    print "You should have given either an int or a float"
```

```
except ZeroDivisionError:
```

```
    print "Infinity"
```

13.4. Practise Problems

1. Write a program to find the square root of a given number without any errors displayed for wrong values.
2. Write a program that takes only integer value and throws error for any other value.
3. Try to write into a file without opening and raise an IO error showing file is not opened.
4. Additionally,for problem statement 3,Raise an IO error if file is not closed.
5. Write a program that raises an error when there is negative input.

Please attempt these above problems.

Click [here](#) to download the solutions.

14.Object Oriented Programming in Python

Objective

- Classes and objects
- Methods
- Principles of object orientation

14.1Classes and objects

14.2Methods

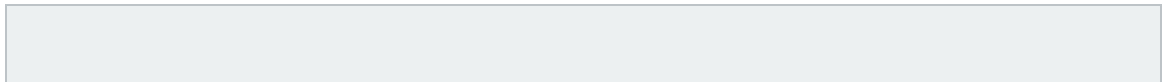
14.3Python memory management

14.4Principles of object orientation

14.5Problem statement

14.6Practise Problems

14.7Reference Links



14.1. Classes and objects

Object-oriented programming (OOP) is a programming paradigm that uses objects and their interactions to design applications and computer programs.

Before going into Object-oriented programming, let us discuss about programming paradigms available.

There are three widely used programming paradigms. They are Procedural programming, functional programming and object-oriented programming. Python supports both procedural and object-oriented programming. There is some limited support for functional programming too.

What is a class?

A class is used in object-oriented programming to describe one or more objects. It serves as a template for creating, or instantiating, specific objects (That have a particular behaviour) within a program. While each object is created from a single class, one class can be used to instantiate multiple objects.

What is an object?

An object is instance of a class. Objects are the data abstraction that encapsulate internal abstraction.

Let us understand about classes and objects with a simple example,

Consider Animal as a class then the objects of this class would be dog, cat, rat etc., That is collection of particular behaviour is a class and the physical object that shows this behaviour is an object.

Class definition and object instantiation

Class definition syntax:

```
class Subclass([superclass]):
```

```
[attributes and methods]
```

If there is no super class mention object in place of super class

Example;

```
class Customer(object):
```

```
    statements
```

```
    ....
```

```
    ....
```

As usual indentation should be followed.

Note: class name should start with capital letter.

Object instantiation syntax:

```
object = class()
```

Example:

```
customer1=Customer()
```

```
customer2=Customer()
```

The class `Customer(object)` line does not create a new customer. That is, just because we've defined a `Customer` doesn't mean we've created one. we've merely outlined the blueprint to create a `Customer` object. `Customer` objects are created as shown in the example of object instantiation.

14.2. Methods

A function defined in a class is called a "method". Methods have access to all the data contained on the instance of the object. they can access and modify anything previously set on self (discussed in 1.2.2.4). Because they use self, they require an instance of the class in order to be used. For this reason, they're often referred to as "instance methods".

Constructor `__init__`

The `__init__` method is run as soon as an object of a class is instantiated. Its aim is to initialize the object. While creating an instance we need some values for internal data.

Example of class with `__init__` method

```
#!/usr/bin/python
```

```
class Coordinate(object):
```

```
    def __init__(self,x,y):
```

```
        self.x=x
```

```
        self.y=y
```

The “.” operator is used to access an attribute of an object. So the `__init__` method above is defining two attributes for new Coordinate object x and y.

```
Now,c=Coordinate(3,4)
```

```
print c.x,c.y
```

Prints 3 4 as output

Invoking attributes and methods

Syntax to invoke attribute:

```
object.attribute
```

Syntax to invoke method:

```
object.method()
```

User defined methods

User defined methods are same as function definition but must be defined inside a class.

Example of class with methods:

```
#!/usr/bin/python
```

```

class Rectangle:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def area(self):
        return self.x * self.y
    def perimeter(self):
        return 2 * self.x + 2 * self.y

```

This example has methods area and perimeter that are used to calculate area and perimeter of an object of the class Rectangle.

self

Each method in a class definition begins with a reference to the instance object. It is by convention named self i.e., methods class access data via self.

For example above(Rectangle class),

instantiate object

```
r=Rectangle(4,5)
```

```
r.area() #area() method uses x and y attributes using self
```

“Self” in Python works as a variable of function but it won't invoke data.

Form and object for a class

Class includes two members form and object.

The following example can explain the difference between form and object for a class.

```
#!/usr/bin/python
```

Class A:

```
i=123    #It is a form
```

```
def __init__(self):
```

```
    self.i=12345
```

```
print A.i    #It is a form
```

```
print A().i  #It is an object
```

Output would be

123

12345

Destructor

Destructor method is used to delete an object created by initializing constructor.

It is called when the instance is about to be destroyed.

The following example shows a class with constructor and destructor:

```
#!/usr/bin/python
```

```
class Greeting:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def __del__(self):
```

```
        print "Destructor started"
```

```
    def SayHello(self):
```

```
        print "Hello", self.name
```

`__del__` is a method used for class destructor.

Let us create an object

```
x1=Greeting("Hari")
```

```
x2=x1    #Another object x2 is created
```

```
>>>del x1
```

```
>>>del x2
```

Destructor started

Here,If you observe destructor is called only after reference count of objects reaches zero.

Note: special methods (like ex: `__init__` , `__del__`) start and end with two underscores.

14.3. Python memory management

Python's memory allocation and deallocation method is automatic.

The user does not have to preallocate or deallocate memory by hand as one has to when using dynamic memory allocation in languages such as C or C++.

Python uses two strategies for memory allocation reference counting and garbage collection.

Prior to Python version 2.0, the Python interpreter only used reference counting for memory management.

Reference counting works by counting the number of times an object is referenced by other objects in the system.

When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero the object is deallocated.

Reference counting is extremely efficient but it does have some caveats.

One such caveat is that it cannot handle reference cycles. A reference cycle is when there is no way to reach an object but its reference count is still greater than zero.

The easiest way to create a reference cycle is to create an object which refers to itself as in the example below:

```
def make_cycle():  
    l = []  
    l.append(l)
```

```
make_cycle()
```

Because `make_cycle()` creates an object `l` which refers to itself, the object `l` will not automatically be freed when the function returns.

This will cause the memory that `l` is using to be held onto until the Python garbage collector is invoked.

Automatic Garbage Collection of Cycles

Because reference cycles take computational work to discover, garbage collection must be a scheduled activity. Python schedules garbage collection based upon a threshold of object allocations

and object deallocations. When the number of allocations minus the number of deallocations are greater than the threshold number, the garbage collector is run. One can inspect the threshold for new objects (objects in Python known as generation 0 objects) by loading the gc module and asking for garbage collection thresholds:

```
import gc
print "Garbage collection thresholds: %r" % gc.get_threshold()
```

```
Garbage collection thresholds: (700, 10, 10)
```

Here we can see that the default threshold on the above system is 700. This means when the number of allocations vs. the number of deallocations is greater than 700 the automatic garbage collector will run.

Automatic garbage collection will not run if your Python device is running out of memory; instead your application will throw exceptions, which must be handled or your application crashes. This is aggravated by the fact that the automatic garbage collection places high weight upon the NUMBER of free objects, not on how large they are. Thus any portion of your code which frees up large blocks of memory is a good candidate for running manual garbage collection.

Manual Garbage Collection

For some programs, especially long running server applications or embedded applications running on a Digi Device automatic garbage collection may not be sufficient. Although an application should be written to be as free of reference cycles as possible, it is a good idea to have a strategy for how to deal with them. Invoking the garbage collector manually during opportune times of program execution can be a good idea on how to handle memory being consumed by reference cycles.

The garbage collection can be invoked manually in the following way:

```
import gc
gc.collect()
```

gc.collect() returns the number of objects it has collected and deallocated. You can print this information in the following way:

```
import gc
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)
```

If we create a few cycles, we can see manual collection work:

```
import sys, gc

def make_cycle():
    l = { }
    l[0] = l

def main():
    collected = gc.collect()
    print "Garbage collector: collected %d objects." % (collected)
    print "Creating cycles..."
    for i in range(10):
        make_cycle()
    collected = gc.collect()
    print "Garbage collector: collected %d objects." % (collected)

if __name__ == "__main__":
    ret = main()
    sys.exit(ret)
```

In general there are two recommended strategies for performing manual garbage collection: time-based and event-based garbage collection. Time-based garbage collection is simple: the garbage collector is called on a fixed time interval. Event-based garbage collection calls the garbage collector on an event. For example, when a user disconnects from the application or when the application is known to enter an idle state.

Recommendations

Which garbage collection technique is correct for an application? It depends. The garbage collector should be invoked as often as necessary to collect cyclic references without affecting vital application performance. Garbage collection should be a part of your Python application design process.

1) Do not run garbage collection too freely, as it can take considerable time to evaluate every memory object within a large system. For example, one team having memory issues tried calling `gc.collect()`

between every step of a complex start-up process, increasing the boot time by 20 times (2000%).

Running it more than a few times per day - without specific design reasons is likely a waste of device resources.

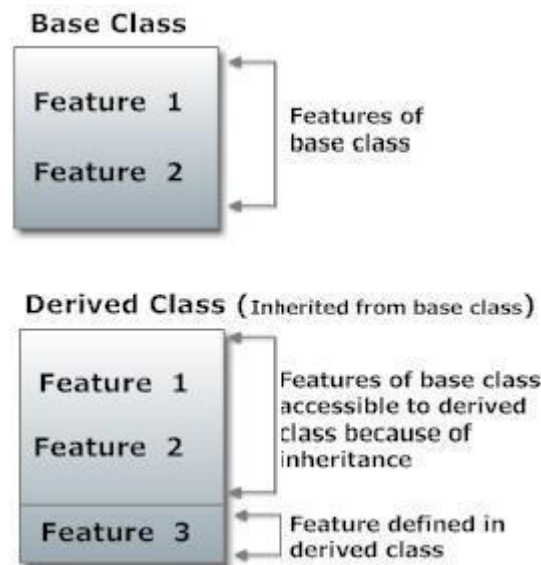
2) Run manual garbage collection after your application has completed start up and moves into steady-state operation. This frees potentially huge blocks of memory used to open and parse file, to build and modify object lists, and even code modules never to be used again. For example, one application reading XML configuration files was consuming about 1.5MB of temporary memory during the process. Without manual garbage collection, there is no way to predict when that 1.5MB of memory will be returned to the python memory pools for reuse.

3) Consider manually running garbage collection either before or after timing-critical sections of code to prevent garbage collection from disturbing the timing. As example, an irrigation application might sit idle for 10 minutes, then evaluate the status of all field devices and make adjustments. Since delays during system adjustment might affect field device battery life, it makes sense to manually run garbage collection as the gateway is entering the idle period after the adjustment process - or run it every sixth or tenth idle period. This insures that garbage collection won't be triggered automatically during the next timing-sensitive period.

14.4. Principles of object orientation

1. Inheritance

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class. Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.



Inheritance Syntax:

```
class Derivedclass(Baseclass):
```

```
...
```

```
...
```

Example:

Consider a class Shape as shown below,

```
#!/usr/bin/python
```

```
class Shape:
```

```
    def __init__(self,x,y):
```

```
        self.x = x
```

```
        self.y = y
```

```
        self.description = "This shape has not been described yet"
```

```
        self.author = "Nobody has claimed to make this shape yet"
```

```
    def area(self):
```

```

        return self.x * self.y
    def perimeter(self):
        return 2 * self.x + 2 * self.y
    def describe(self,text):
        self.description = text
    def authorName(self,text):
        self.author = text
    def scaleSize(self,scale):
        self.x = self.x * scale
        self.y = self.y * scale

```

The below class is inherited from Square class

```

class Square(Shape):
    def __init__(self,x):
        self.x = x
        self.y = x

```

Now, The class Square can use all the behaviours and attributes from the class Shape until and unless they are not accessible (will be discussed in encapsulation).

Use super() method in sub class to call a method from Parent class.

Example:

```

#!/usr/bin/python
class Dataset(object):
    def __init__(self, data=None):
        self.data = data
class MRIDataset(Dataset):
    def __init__(self, data=None, parameters=None):
        # here has the same effect as calling
        # Dataset.__init__(self)
        super(MRIDataset, self).__init__(data)

```

```
self.parameters = parameters
mri_data = MRIDataset(data=[1,2,3])
```

2. Multiple inheritance

Multiple inheritance is possible in Python. A class can be derived from more than one base classes.

The syntax for multiple inheritance is similar to single inheritance.

Syntax:

```
class Base1:
    ...
class Base2:
    ...
class MultiDerived(Base1, Base2):
    ...
```

Here MultiDerived class uses features of Base2 and Base1. Base1 methods are checked first and then Base2 functions follow.

3. Multilevel Inheritance

On the other hand, we can inherit from a derived class. This is also called multilevel inheritance.

Multilevel inheritance can be of any depth in Python.

An example is given below,

```
#!/usr/bin/python
class Base:
    pass
class Derived1(Base):
    pass
class Derived2(Derived1):
    pass
```

Here Derived2 can use features of Base and Derived1 along with features of Derived2.

4. Encapsulation

Encapsulation is the packing of data and functions into a single component.

The features of encapsulation are supported using classes in most object-oriented programming languages, although other alternatives also exist.

It allows selective hiding of properties and methods in an object by building an impenetrable wall to protect the code from accidental corruption.

It is a language mechanism for restricting access to some of the object's components. A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

The accessibility of data is done by providing access specifiers. The access specifiers we have in programming language are public,private,protected.

In python programming language everything we write is public that means every class can access the variables/methods as they are public.

To make the accessibility hidden from the classes other than it is defined we should make it as a private variable/method. To restrict its access to specified classes we make them as protected.

The access specifiers' syntax(check the comments) is explained with following examples,

```
#!/usr/bin/python
class Person:
    def __init__(self):
        self.a='hari'          #public variable
        self.__b='siddartha'    #private variable
        self._c='hyd'          #protected variable
```

Now let us check how they are accessible with below example,

```
#!/usr/bin/python
class Person:
    def __init__(self):
        self.a='hari'          #public variable
        self.__b='siddartha'    #private variable
        self._c='hyd'          #protected variable
    def printName(self):
        print self.a
        print self.__b
        print self._c
```



```
P=Person()
```

```
P.a
```

```
P.b
```

```
P.c
```

```
P.__b
```

```
P._c
```

Check what happens with above code.

Here you can't access P.b,P.c,P.__b.

To access private variable the syntax to be followed is `_ClassName__variable` or

`_ClassName__function()`. So, we can't access them as `object.__variable` or `object.__function()`.

5. Polymorphism

Another important attribute of an object-oriented programming language is polymorphism: the ability to use the same syntax for objects or methods of different types. (Strictly speaking, this is ad-hoc polymorphism.) For example, in Python, the square bracket operator is used to perform indexing of various sequence types (`list[3]`, `dict["foo"]`); polymorphism allows us to define our own types, as classes, that emulate built-in Python types like sequences and which therefore can use e.g. square brackets for indexing.

This example describes about polymorphism,

```
#!/usr/bin/python
```

```
class Animal:
```

```
    def Name(self):
```

```
        pass
```

```
    def Sleep(self):
```

```
        print 'sleep'
```

```
    def MakeNoise(self):
```

```
        pass
```

```
class Dog(Animal):
```

```
    def Name(self):
```

```
        print 'I am a dog'
```

```
    def MakeNoise(self):
```

```
        print 'Woof'
```

```
class Cat(Animal):
    def Name(self):
        print 'I am cat'
    def MakeNoise(self):
        print 'Meow'
class Lion(Animal):
    def Name(self):
        print 'I am a lion'
    def MakeNoise(self):
        print 'Roar'
class TestAnimals:
    def PrintName(self,animal):
        animal.Name()
    def GotoSleep(self,animal):
        animal.Sleep()
    def MakeNoise(self,animal):
        animal.MakeNoise()
```

```
TestAnimals=TestAnimals()
dog=Dog()
cat=Cat()
lion=Lion()
TestAnimals.PrintName(dog)
TestAnimals.GotoSleep(dog)
TestAnimals.MakeNoise(dog)
TestAnimals.PrintName(cat)
TestAnimals.GotoSleep(cat)
TestAnimals.MakeNoise(cat)
TestAnimals.PrintName(lion)
TestAnimals.GotoSleep(lion)
TestAnimals.MakeNoise(lion)
```

As you can see same methods are repeated in different classes, It is called method overloading.

The output of above program is,

I am a dog

sleep

Woof

I am cat

sleep

Meow

I am a lion

sleep

Roar

Suppose you've created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will not work as desired.

You could, however, define the `__add__` method in your class to perform vector addition and then the plus operator would behave as per expectation. This is called as operator overloading.

EXAMPLE:

```
#!/usr/bin/python
```

```
class Vector:
```

```
    def __init__(self, a, b):
```

```
        self.a = a
```

```
        self.b = b
```

```
    def __str__(self):
```

```
        return 'Vector (%d, %d)' % (self.a, self.b)
```

```
    def __add__(self, other):
```

```
        return Vector(self.a + other.a, self.b + other.b)
```

```
v1 = Vector(2,10)
```

```
v2 = Vector(5,-2)
```

```
print v1 + v2
```

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

Vector(7,8)

14.5. Problem statement

Write a program of account class which has the functionalities like checking, withdrawing, depositing and transferring the balance using object oriented approach.

Note: The credit line (the balance less than zero) of the account is 1500.

Solution

```
#!/usr/bin/python
```

```
class Account(object):
```

```
    def __init__(self, holder, number, balance, credit_line=1500):
```

```
        self.Holder = holder
```

```
        self.Number = number
```

```
        self.Balance = balance
```

```
        self.CreditLine = credit_line
```

```
    def deposit(self, amount):
```

```
        self.Balance = amount
```

```
    def withdraw(self, amount):
```

```
        if((self.Balance) - amount < -(self.CreditLine)):
```

```
            # coverage insufficient
```

```
            return False
```

```
        else:
```

```
            self.Balance -= amount
```

```
            return True
```

```
    def balance(self):
```

```
        return self.Balance
```

```
    def transfer(self, target, amount):
```

```
        if((self.Balance - amount) < -(self.CreditLine)):
```

```
            # coverage insufficient
```

```
            return False
```

```
        else:
```

```
            self.Balance -= amount
```

```
            target.Balance += amount
```

```
            return True
```

Now check the working of the code with two sample accounts as shown below,

```
obj=Account('siddu',1032,17000)
```

```
print (obj.balance())
```

```
obj2=Account('sid',1435,20000)
```

```
print (obj2.balance())
```

```
obj2.transfer(obj,2000)
```

```
print (obj.balance())
```

```
print (obj2.balance())
```

The answer would be,

17000

20000

19000

18000

14.6. Practise Problems

1. Define a class Book which takes parameters title,author,pages and write methods to display the title,author and no.of pages.
2. Create a destructor for the above program for deleting the book instance of the class.
3. Write a program to find area of circle,rectangle,square.Take a generic class Shape and inherit the classes Circle,Rectangle,Square.
4. Write a Class program with Cordinate class to locate two coordinates x,y(attributes) and add two coordinates by creating instances.
5. Write a Class program that takes numerator and denominator of rational number and also have methods add,sub,mul,div that do basic arithmetic operations on two rational numbers.

Please attempt these above problems.

Click [here](#) to download the solutions.

14.7. Reference Links

Reference Links

<http://www.w3resource.com/python/python-data-type.php>

<http://www.informit.com/articles/article.aspx?p=459269&seqNum=7>

<http://www.pythonforbeginners.com/basics/keywords-in-python>

<http://www.python-course.eu/input.php>

http://www.tutorialspoint.com/python/python_strings.htm

<http://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/singh.pdf>

http://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection