

**Department of Computer Science & Applications**  
**Study Material – 2022(Odd Semester)**

Sub:Open source computing  
Staff I/c: K. Aravindhan M.Sc.,

Paper Code: 21PCS  
Date:

Class: II M.Sc CS  
Head:

---

**UNIT –I**

Python: Introduction – Numbers – Strings – Variables – Lists – Tuples – Dictionaries – Sets – Comparison.

---

**Topic 1: Introduction**

**Possible Question: Discuss about python?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

**Python** is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This **tutorial** gives enough understanding on **Python programming** language.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

**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 copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

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 features include –

- **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
- **A broad standard library** – Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** – Python has support for an interactive mode which allows 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.

- **Extendable** – 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 –

- It supports 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.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

## Topic 2:Numbers

### Possible Question:Explain Numbers?

### Possible Marks: 5 or 10 Marks

#### Outcomes:

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example –

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is –

```
del var1[,var2[,var3[ ...,varN]]]]
```

You can delete a single object or multiple objects by using the **del** statement. For example –

```
del var
del var_a, var_b
```

Python supports four different numerical types –

- **int (signed integers)** – They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
- **long (long integers )** – Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
- **float (floating point real values)** – Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 ( $2.5e2 = 2.5 \times 10^2 = 250$ ).
- **complex (complex numbers)** – are of the form  $a + bJ$ , where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

#### Examples

Here are some examples of numbers

int	long	float	complex
10	51924361L	0.0	3.14j

100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEL	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

### Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type **int(x)** to convert x to a plain integer.
- Type **long(x)** to convert x to a long integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

### Mathematical Functions

Python includes following functions that perform mathematical calculations.

Sr.No.	Function & Returns ( description )
1	<u>abs(x)</u> The absolute value of x: the (positive) distance between x and zero.
2	<u>ceil(x)</u> The ceiling of x: the smallest integer not less than x
3	<u>cmp(x, y)</u> -1 if x < y, 0 if x == y, or 1 if x > y
4	<u>exp(x)</u> The exponential of x: e <sup>x</sup>
5	<u>fabs(x)</u> The absolute value of x.
6	<u>floor(x)</u> The floor of x: the largest integer not greater than x
7	<u>log(x)</u> The natural logarithm of x, for x > 0
8	<u>log10(x)</u>

	The base-10 logarithm of x for $x > 0$ .
9	<u>max(x1, x2,...)</u> The largest of its arguments: the value closest to positive infinity
10	<u>min(x1, x2,...)</u> The smallest of its arguments: the value closest to negative infinity
11	<u>modf(x)</u> The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
12	<u>pow(x, y)</u> The value of $x^{**}y$ .
13	<u>round(x [,n])</u> x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
14	<u>sqrt(x)</u> The square root of x for $x > 0$

### Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

Sr.No.	Function & Description
1	<u>choice(seq)</u> A random item from a list, tuple, or string.
2	<u>randrange ([start,] stop [,step])</u> A randomly selected element from range(start, stop, step)
3	<u>random()</u> A random float r, such that 0 is less than or equal to r and r is less than 1
4	<u>seed([x])</u> Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
5	<u>shuffle(lst)</u> Randomizes the items of a list in place. Returns None.
6	<u>uniform(x, y)</u> A random float r, such that x is less than or equal to r and r is less than y

### Trigonometric Functions

Python includes following functions that perform trigonometric calculations.

Sr.No.	Function & Description
1	<u>acos(x)</u> Return the arc cosine of x, in radians.
2	<u>asin(x)</u> Return the arc sine of x, in radians.

3	<u>atan(x)</u> Return the arc tangent of x, in radians.
4	<u>atan2(y, x)</u> Return atan(y / x), in radians.
5	<u>cos(x)</u> Return the cosine of x radians.
6	<u>hypot(x, y)</u> Return the Euclidean norm, sqrt(x*x + y*y).
7	<u>sin(x)</u> Return the sine of x radians.
8	<u>tan(x)</u> Return the tangent of x radians.
9	<u>degrees(x)</u> Converts angle x from radians to degrees.
10	<u>radians(x)</u> Converts angle x from degrees to radians.

### Mathematical Constants

The module also defines two mathematical constants –

Sr.No.	Constants & Description
1	<b>pi</b> The mathematical constant pi.
2	<b>e</b> The mathematical constant e.

---

### Topic 3: Strings

#### Possible Question: Explain Strings?

**Possible Marks: 5 or 10 Marks**

#### Outcomes:

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example –

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

#### Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example –

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"
```

```
print"var1[0]: ", var1[0]
print"var2[1:5]: ", var2[1:5]
```

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

```
var1[0]: H
var2[1:5]: ytho
```

### Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example –

```
#!/usr/bin/python

var1 ='Hello World!'
print"Updated String :- ", var1[:6]+'Python'
```

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

```
Updated String :- Hello Python
```

### Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0-7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0-9, a-f, or A-F

## String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give - HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[ : ]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n' prints \n
%	Format - Performs String formatting	See at next section

## String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example –

```
#!/usr/bin/python  
  
print"My name is %s and weight is %d kg!"%('Zara',21)
```

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

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % –

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer

%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table –

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

### Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINES, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

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

```
para_str="""this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""
printpara_str
```



When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINES occur either with an explicit carriage return at the end of a line or its escape code (\n) – this is a long string that is made up of several lines and non-printable characters such as TAB ( ) and they will show up that way when displayed. NEWLINES within the string, whether explicitly given like this within the brackets [ ], or just a NEWLINE within the variable assignment will also show up. Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it –

```
#!/usr/bin/python

print'C:\\nowhere'
```

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

```
C:\\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'** as follows –

```
#!/usr/bin/python

printr'C:\\nowhere'
```

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

```
C:\\nowhere
```

### Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following –

```
#!/usr/bin/python

printu'Hello, world!'
```

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

```
Hello, world!
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

### Built-in String Methods

Python includes the following built-in methods to manipulate strings –

Sr.No.	Methods with Description
1	<u>capitalize()</u> Capitalizes first letter of string
2	<u>center(width, fillchar)</u> Returns a space-padded string with the original string centered to a total of width columns.
3	<u>count(str, beg= 0,end=len(string))</u> Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	<u>decode(encoding='UTF-8',errors='strict')</u>

	Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	<u>encode(encoding='UTF-8',errors='strict')</u> Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	<u>endswith(suffix, beg=0, end=len(string))</u> Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	<u>expandtabs(tabsize=8)</u> Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	<u>find(str, beg=0 end=len(string))</u> Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	<u>index(str, beg=0, end=len(string))</u> Same as find(), but raises an exception if str not found.
10	<u>isalnum()</u> Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	<u>isalpha()</u> Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	<u>isdigit()</u> Returns true if string contains only digits and false otherwise.
13	<u>islower()</u> Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	<u>isnumeric()</u> Returns true if a unicode string contains only numeric characters and false otherwise.
15	<u>isspace()</u> Returns true if string contains only whitespace characters and false otherwise.
16	<u>istitle()</u> Returns true if string is properly "titlecased" and false otherwise.
17	<u>isupper()</u> Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	<u>join(seq)</u> Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	<u>len(string)</u>

	Returns the length of the string
20	<u>ljust(width[, fillchar])</u> Returns a space-padded string with the original string left-justified to a total of width columns.
21	<u>lower()</u> Converts all uppercase letters in string to lowercase.
22	<u>lstrip()</u> Removes all leading whitespace in string.
23	<u>maketrans()</u> Returns a translation table to be used in translate function.
24	<u>max(str)</u> Returns the max alphabetical character from the string str.
25	<u>min(str)</u> Returns the min alphabetical character from the string str.
26	<u>replace(old, new [, max])</u> Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	<u>rfind(str, beg=0, end=len(string))</u> Same as find(), but search backwards in string.
28	<u>rindex( str, beg=0, end=len(string))</u> Same as index(), but search backwards in string.
29	<u>rjust(width[, fillchar])</u> Returns a space-padded string with the original string right-justified to a total of width columns.
30	<u>rstrip()</u> Removes all trailing whitespace of string.
31	<u>split(str="", num=string.count(str))</u> Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	<u>splitlines( num=string.count('\n'))</u> Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	<u>startswith(str, beg=0, end=len(string))</u> Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	<u>strip([chars])</u> Performs both lstrip() and rstrip() on string.
35	<u>swapcase()</u> Inverts case for all letters in string.

36	<u>title()</u> Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	<u>translate(table, deletechars="")</u> Translates string according to translation table str(256 chars), removing those in the del string.
38	<u>upper()</u> Converts lowercase letters in string to uppercase.
39	<u>zfill (width)</u> Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	<u>isdecimal()</u> Returns true if a unicode string contains only decimal characters and false otherwise.

#### Topic 4: Variables

##### Possible Question: Describe Variables?

##### Possible Marks: 5 or 10 Marks

##### Outcomes:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

##### Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example –

```
#!/usr/bin/python

counter =100# An integer assignment
miles =1000.0# A floating point
name  ="John"# A string

print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result –

```
100
1000.0
John
```

##### Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example –

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example –

```
a,b,c = 1,2,"john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

### Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

### Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example –

```
var1 = 1  
var2 = 10
```

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is –

```
del var1[,var2[,var3[ ...,varN]]]]
```

You can delete a single object or multiple objects by using the del statement. For example –

```
del var  
del var_a, var_b
```

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAE1	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by  $x + yj$ , where  $x$  and  $y$  are the real numbers and  $j$  is the imaginary unit.

### Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator. For example –

```
#!/usr/bin/python

str='Hello World!'

print str      # Prints complete string
printstr[0]# Prints first character of the string
printstr[2:5]# Prints characters starting from 3rd to 5th
printstr[2:]# Prints string starting from 3rd character
print str *2# Prints string two times
print str +"TEST"# Prints concatenated string
```

This will produce the following result –

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

### Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator. For example –

```
#!/usr/bin/python

list=['abcd',786,2.23,'john',70.2]
tinylist=[123,'john']

print list      # Prints complete list
printlist[0]# Prints first element of the list
printlist[1:3]# Prints elements starting from 2nd till 3rd
printlist[2:]# Prints elements starting from 3rd element
printtinylist*2# Prints list two times
print list +tinylist# Prints concatenated lists
```

This produce the following result –

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

## Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as **read-only** lists. For example –

```
#!/usr/bin/python

tuple=('abcd',786,2.23,'john',70.2)
tinytuple=(123,'john')

print tuple           # Prints the complete tuple
printtuple[0]# Prints first element of the tuple
printtuple[1:3]# Prints elements of the tuple starting from 2nd till 3rd
printtuple[2:]# Prints elements of the tuple starting from 3rd element
printtinytuple*2# Prints the contents of the tuple twice
print tuple +tinytuple# Prints concatenated tuples
```

This produce the following result –

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```
#!/usr/bin/python

tuple=('abcd',786,2.23,'john',70.2)
list=['abcd',786,2.23,'john',70.2]
tuple[2]=1000# Invalid syntax with tuple
list[2]=1000# Valid syntax with list
```

## Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ( { } ) and values can be assigned and accessed using square braces ( [ ] ). For example –

```
#!/usr/bin/python

dict={ }
dict['one']="This is one"
dict[2]="This is two"

tinydict={'name':'john','code':6734,'dept':'sales'}

printdict['one']# Prints value for 'one' key
printdict[2]# Prints value for 2 key
printtinydict# Prints complete dictionary
printtinydict.keys()# Prints all the keys
printtinydict.values()# Prints all the values
```

This produce the following result –

This is one

This is two

```
{'dept': 'sales', 'code': 6734, 'name': 'john'}
```

```
['dept', 'code', 'name']
```

```
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

### Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	<b>int(x [,base])</b> Converts x to an integer. base specifies the base if x is a string.
2	<b>long(x [,base] )</b> Converts x to a long integer. base specifies the base if x is a string.
3	<b>float(x)</b> Converts x to a floating-point number.
4	<b>complex(real [,imag])</b> Creates a complex number.
5	<b>str(x)</b> Converts object x to a string representation.
6	<b>repr(x)</b> Converts object x to an expression string.
7	<b>eval(str)</b> Evaluates a string and returns an object.
8	<b>tuple(s)</b> Converts s to a tuple.
9	<b>list(s)</b> Converts s to a list.
10	<b>set(s)</b> Converts s to a set.
11	<b>dict(d)</b> Creates a dictionary. d must be a sequence of (key,value) tuples.
12	<b>frozenset(s)</b> Converts s to a frozen set.
13	<b>chr(x)</b> Converts an integer to a character.
14	<b>unichr(x)</b> Converts an integer to a Unicode character.



15	<b>ord(x)</b> Converts a single character to its integer value.
16	<b>hex(x)</b> Converts an integer to a hexadecimal string.
17	<b>oct(x)</b> Converts an integer to an octal string.

## Topic 5:Lists

### Possible Question: Describe Lists?

### Possible Marks: 5 or 10 Marks

#### Outcomes:

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

#### Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

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

#### Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
#!/usr/bin/python

list1=['physics','chemistry',1997,2000];
list2=[1,2,3,4,5,6,7];
print"list1[0]: ", list1[0]
print"list2[1:5]: ", list2[1:5]
```

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

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

#### Updating Lists

You 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. For example –

```
#!/usr/bin/python

list=['physics','chemistry',1997,2000];
print"Value available at index 2 : "
printlist[2]
list[2]=2001;
```

```
print"New value available at index 2 : "  
printlist[2]
```

**Note** – append() method is discussed in subsequent section.

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

```
Value available at index 2 :
```

```
1997
```

```
New value available at index 2 :
```

```
2001
```

### Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example –

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

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

```
print list1
```

```
del list1[2];
```

```
print"After deleting value at index 2 : "
```

```
print list1
```

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

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

```
After deleting value at index 2 :
```

```
['physics', 'chemistry', 2000]
```

**Note** – remove() method is discussed in subsequent section.

### 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 in the prior

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

### Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

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

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

### Built-in List Functions & Methods

Python includes the following list functions –

Sr.No.	Function with Description
1	<u>cmp(list1, list2)</u> Compares elements of both lists.
2	<u>len(list)</u> Gives the total length of the list.
3	<u>max(list)</u> Returns item from the list with max value.
4	<u>min(list)</u> Returns item from the list with min value.
5	<u>list(seq)</u> Converts a tuple into list.

Python includes following list methods

Sr.No.	Methods with Description
1	<u>list.append(obj)</u> Appends object obj to list
2	<u>list.count(obj)</u> Returns count of how many times obj occurs in list
3	<u>list.extend(seq)</u> Appends the contents of seq to list
4	<u>list.index(obj)</u> Returns the lowest index in list that obj appears
5	<u>list.insert(index, obj)</u> Inserts object obj into list at offset index
6	<u>list.pop(obj=list[-1])</u> Removes and returns last object or obj from list
7	<u>list.remove(obj)</u> Removes object obj from list
8	<u>list.reverse()</u> Reverses objects of list in place
9	<u>list.sort([func])</u> Sorts objects of list, use compare func if given

## Topic 6:Tuples

**Possible Question: ExplainTuples?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5 );  
tup3 = "a", "b", "c", "d";
```

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 –

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

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

### Accessing Values in 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. For example –

```
#!/usr/bin/python  
  
tup1=('physics','chemistry',1997,2000);  
tup2=(1,2,3,4,5,6,7);  
print"tup1[0]: ", tup1[0];  
print"tup2[1:5]: ", tup2[1:5];
```

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

```
tup1[0]: physics  
tup2[1:5]: [2, 3, 4, 5]
```

### Updating 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 as the following example demonstrates –

```
#!/usr/bin/python  
  
tup1 =(12,34.56);  
tup2 =('abc','xyz');  
  
# Following action is not valid for tuples  
# tup1[0] = 100;  
  
# So let's create a new tuple as follows  
tup3 = tup1 + tup2;  
print tup3;
```

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

```
(12, 34.56, 'abc', 'xyz')
```

### Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example –

```
#!/usr/bin/python  
  
tup =('physics','chemistry',1997,2000);  
print tup;  
del tup;  
print"After deleting tup : ";  
print tup;
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more –

```
('physics', 'chemistry', 1997, 2000)  
After deleting tup :
```

```
Traceback (most recent call last):
File "test.py", line 9, in <module>
print tup;
NameError: name 'tup' is not defined
```

### Basic Tuples 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.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior

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

### Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input –

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

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

### No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples –

```
#!/usr/bin/python

print'abc',-4.24e93,18+6.6j,'xyz';
x, y =1,2;
print"Value of x , y : ",x,y;
```

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

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

### Built-in Tuple Functions

Python includes the following tuple functions –

Sr.No.	Function with Description
1	<u>cmp(tuple1, tuple2)</u> Compares elements of both tuples.
2	<u>len(tuple)</u> Gives the total length of the tuple.

3	<u>max(tuple)</u> Returns item from the tuple with max value.
4	<u>min(tuple)</u> Returns item from the tuple with min value.
5	<u>tuple(seq)</u> Converts a list into tuple.

## Topic7:Dictionaries

### Possible Question: Explain Dictionaries?

### Possible Marks: 5 or 10 Marks

#### Outcomes:

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary 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.

#### Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –

```
#!/usr/bin/python

dict={'Name':'Zara','Age':7,'Class':'First'}
print"dict['Name']: ",dict['Name']
print"dict['Age']: ",dict['Age']
```

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

```
dict['Name']: Zara
dict['Age']: 7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows –

```
#!/usr/bin/python

dict={'Name':'Zara','Age':7,'Class':'First'}
print"dict['Alice']: ",dict['Alice']
```

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

```
dict['Alice']:
Traceback (most recent call last):
File "test.py", line 4, in <module>
print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

#### Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

```
#!/usr/bin/python

dict={'Name':'Zara','Age':7,'Class':'First'}
dict['Age']=8;# update existing entry
dict['School']="DPS School";# Add new entry

print"dict['Age']: ",dict['Age']
```

```
print"dict['School']: ",dict['School']
```

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

```
dict['Age']: 8
```

```
dict['School']: DPS School
```

### Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example –

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

```
dict={'Name':'Zara','Age':7,'Class':'First'}  
deldict['Name'];# remove entry with key 'Name'  
dict.clear();# remove all entries in dict  
deldict;# delete entire dictionary
```

```
print"dict['Age']: ",dict['Age']
```

```
print"dict['School']: ",dict['School']
```

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more –

```
dict['Age']:
```

```
Traceback (most recent call last):
```

```
File "test.py", line 8, in <module>
```

```
print "dict['Age']: ", dict['Age'];
```

```
TypeError: 'type' object is unsubscriptable
```

**Note** – del() method is discussed in subsequent section.

### Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys –

(a) More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example –

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

```
dict={'Name':'Zara','Age':7,'Name':'Manni'}  
print"dict['Name']: ",dict['Name']
```

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

```
dict['Name']: Manni
```

(b) Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example –

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

```
dict={['Name']:'Zara','Age':7}  
print"dict['Name']: ",dict['Name']
```

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

```
Traceback (most recent call last):
```

```
File "test.py", line 3, in <module>
```

```
dict = {'Name': 'Zara', 'Age': 7};
```

```
TypeError: unhashable type: 'list'
```

### Built-in Dictionary Functions & Methods

Python includes the following dictionary functions –

Sr.No.	Function with Description
--------	---------------------------

1	<u>cmp(dict1, dict2)</u> Compares elements of both dict.
2	<u>len(dict)</u> Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	<u>str(dict)</u> Produces a printable string representation of a dictionary
4	<u>type(variable)</u> Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods –

Sr.No.	Methods with Description
1	<u>dict.clear()</u> Removes all elements of dictionary <i>dict</i>
2	<u>dict.copy()</u> Returns a shallow copy of dictionary <i>dict</i>
3	<u>dict.fromkeys()</u> Create a new dictionary with keys from seq and values <i>set</i> to <i>value</i> .
4	<u>dict.get(key, default=None)</u> For <i>key</i> key, returns value or default if key not in dictionary
5	<u>dict.has_key(key)</u> Returns <i>true</i> if key in dictionary <i>dict</i> , <i>false</i> otherwise
6	<u>dict.items()</u> Returns a list of <i>dict</i> 's (key, value) tuple pairs
7	<u>dict.keys()</u> Returns list of dictionary <i>dict</i> 's keys
8	<u>dict.setdefault(key, default=None)</u> Similar to <i>get()</i> , but will set <i>dict[key]=default</i> if <i>key</i> is not already in <i>dict</i>
9	<u>dict.update(dict2)</u> Adds dictionary <i>dict2</i> 's key-values pairs to <i>dict</i>
10	<u>dict.values()</u> Returns list of dictionary <i>dict</i> 's values

## Topic 8:Sets

**Possible Question: ExplainSets?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.



A set is a collection which is *unordered*, *unchangeable\**, and *unindexed*.  
Sets are written with curly brackets.

Create a Set:

```
thisset = {"apple", "banana", "cherry"}  
print(thisset)
```

### Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

### Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

### Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

### Duplicates Not Allowed

Sets cannot have two items with the same value.

Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}  
print(thisset)
```

### Get the Length of a Set

To determine how many items a set has, use the `len()` function.

*Example*

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}  
print(len(thisset))
```

### Set Items - Data Types

Set items can be of any data type:

*Example*

String, int and boolean data types:

```
set1 = {"apple", "banana", "cherry"}  
set2 = {1, 5, 7, 9, 3}  
set3 = {True, False, False}
```

A set can contain different data types:

*Example*

A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

### type()

From Python's perspective, sets are defined as objects with the data type 'set':

```
<class 'set'>
```

*Example*

What is the data type of a set?

```
myset = {"apple", "banana", "cherry"}  
print(type(myset))
```

### The set() Constructor

It is also possible to use the `set()` constructor to make a set.

*Example*

Using the `set()` constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets  
print(thisset)
```

### Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.

- **Dictionary** is a collection which is ordered\*\* and changeable. No duplicate members.

## Topic 9: Comparison

### Possible Question: Explain Comparison?

Possible Marks: 5 or 10 Marks

#### Outcomes:

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

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

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

#### Example

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

```
#!/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

```

### Model Questions

#### Objective

1. Who developed Python Programming Language?

- a) Wick van Rossum                      b) Rasmus Lerdorf      c) Guido van Rossum d) NieneStom

Answer: c

2. Which type of Programming does Python support?

- a) object-oriented programming    b) structured programming  
c) functional programming    d) all of the mentioned

Answer: d

3. Is Python case sensitive when dealing with identifiers?

- a) no                      b) yes                      c) machine dependent d) none of the mentioned

Answer: a

4. Which of the following is the correct extension of the Python file?

- a) .python                      b) .pl                      c) .py                      d) .p

Answer: c

5. Is Python code compiled or interpreted?

- a) Python code is both compiled and interpreted

b) Python code is neither compiled nor interpreted

c) Python code is only compiled

d) Python code is only interpreted

Answer: b

6. All keywords in Python are in \_\_\_\_\_

a) Capitalized

b) lower case

c) UPPER CASEd) None of the mentioned

Answer: d

7. What will be the value of the following Python expression?

$4 + 3 \% 5$

a) 7

b) 2

c) 4

d) 1

Answer: a

8. Which of the following is used to define a block of code in Python language?

a) Indentation

b) Key

c) Brackets

d) All of the mentioned

Answer: a

9. Which keyword is used for function in Python language?

a) Function

b) Def

c) Fun

d) Define

Answer: b

10. Which of the following character is used to give single-line comments in Python?

a) //

b) #

c) !

d) /\*

Answer: b

### Subjective

1. Explain overview of Python?
2. Write short notes on Numbers?
3. Explain about Strings?
4. Discuss about Variables?
5. Explain about Lists?
6. Write short notes on Tuples?
7. Discuss about Dictionaries?
8. Explain about Sets?
9. Explain about Comparison operator?

---

**Unit – I – END**

**Signature of the staff  
with date**



# Sri Ganesh College of Arts & Science – Salem- 14.

Department of Computer Science & Applications

Study Material – 2022(Odd Semester)

Sub: Open source computing

Paper Code: 21PCS

Class: II M.Sc CS

Staff I/c: K. Aravindhan M.Sc.,

Date:

Head:

---

## UNIT – II

Code Structures: if, elif, and else – Repeat with while – Iterate with for – Comprehensions – Functions – Generators – Decorators – Namespaces and Scope – Handle Errors with try and except – User Exceptions. Modules, Packages, and Programs: Standalone Programs – Command-Line Arguments – Modules and the import Statement – The Python Standard Library. Objects and Classes: Define a Class with class – Inheritance – Override a Method – Add a Method – Get Help from Parent with super – In self Defense – Get and Set Attribute Values with Properties – Name Mangling for Privacy – Method Types – Duck Typing – Special Methods – Composition

---

### Topic 1: Code Structures

**Possible Question: Explain detail about Code Structures of if ,elif and else?**

**Possible Marks: 5 or 10 Marks**

#### Outcomes:

Decision making is required when we want to execute a code only if a certain condition is satisfied.

The if...elif...else statement is used in Python for decision making.

#### Python if Statement Syntax

if test expression:

statement(s)

Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.

If the test expression is False, the statement(s) is not executed.

In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end.

Python interprets non-zero values as True. None and 0 are interpreted as False.

Python if Statement Flowchart

Flowchart of if in Python programming

Example: Python if Statement

```
# If the number is positive, we print an appropriate message
```

```
num = 3
```

```
if num > 0:
```

```
print(num, "is a positive number.")
```

```
print("This is always printed.")
```

```
num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

When you run the program, the output will be:

```
██████████
██████████
██████████
```

In the above example, `num > 0` is the test expression.

The body of `if` is executed only if this evaluates to `True`.

When the variable `num` is equal to 3, test expression is true and statements inside the body of `if` are executed.

If the variable `num` is equal to -1, test expression is false and statements inside the body of `if` are skipped.

The `print()` statement falls outside of the `if` block (unindented). Hence, it is executed regardless of the test expression.

### **Python if...else Statement**

#### **Syntax of if...else**

```
if test expression:
```

```
    Body of if
```

```
else:
```

```
    Body of else
```

The `if..else` statement evaluates test expression and will execute the body of `if` only when the test condition is `True`.

If the condition is `False`, the body of `else` is executed. Indentation is used to separate the blocks.

Python `if..else` Flowchart

Flowchart of `if...else` statement in Python

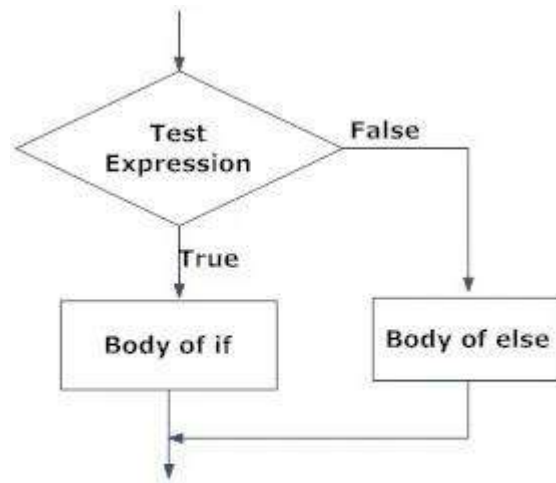


Fig: Operation of if...else statement

Example of if...else

# Program checks if the number is positive or negative

# And displays an appropriate message

```

num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
  
```

### Output

██████████

In the above example, when num is equal to 3, the test expression is true and the body of if is executed and the body of else is skipped.

If num is equal to -5, the test expression is false and the body of else is executed and the body of if is skipped.

If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.

### Python if...elif...else Statement

#### Syntax of if...elif...else

if test expression:

    Body of if

elif test expression:

Body of elif

else:

Body of else

The elif is short for else if. It allows us to check for multiple expressions.

If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, the body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

Flowchart of if...elif...else

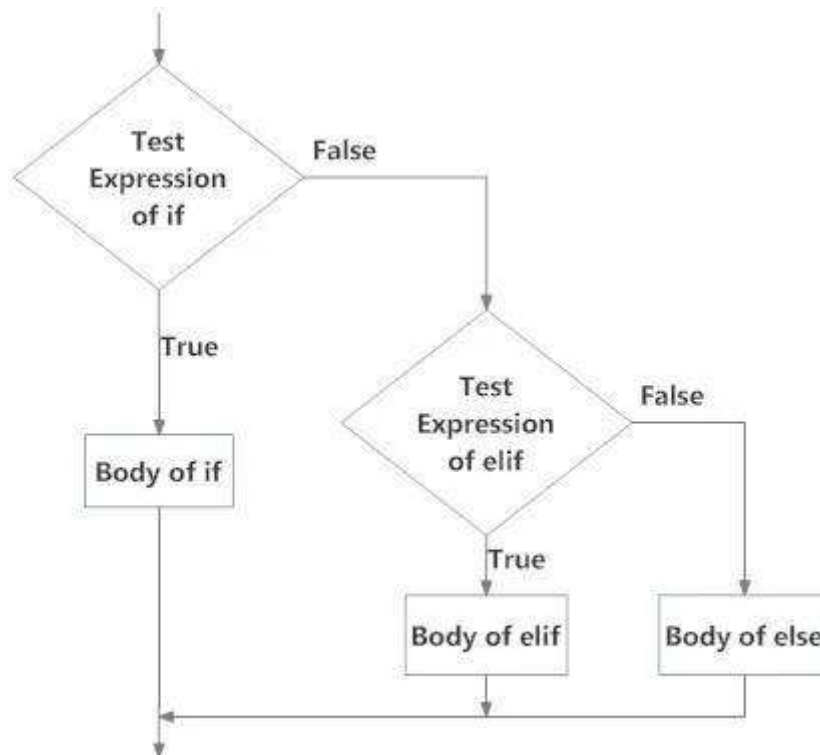


Fig: Operation of if...elif...else statement

Example of if...elif...else

"In this program,  
we check if the number is positive or  
negative or zero and  
display an appropriate message"

```
num = 3.4
```

```
# Try these two variations as well:
```

```
# num = 0
```

```
# num = -4.5
```

```
if num > 0:
```



```
print("Positive number")
elif num == 0:
    print("Zero")
else:
print("Negative number")
```

When variable num is positive, Positive number is printed.

If num is equal to 0, Zero is printed.

If num is negative, Negative number is printed.

### **Python Nested if statements**

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

### **Python Nested if Example**

"In this program, we input a number  
check if the number is positive or  
negative or zero and display  
an appropriate message  
This time we use nested if statement"

```
num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
print("Positive number")
else:
print("Negative number")
```

Output 1

Enter a number: 5

Positive number

Output 2

Enter a number: -1

Negative number

Output 3

Enter a number: 0

## Topic 2: Repeat with while

**Possible Question: Explain detail about Repeat with while statement?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know the number of times to iterate beforehand.

### Syntax of while Loop in Python

```
while test_expression:  
    Body of while
```

In the while loop, test expression is checked first. The body of the loop is entered only if the `test_expression` evaluates to `True`. After one iteration, the test expression is checked again. This process continues until the `test_expression` evaluates to `False`.

In Python, the body of the while loop is determined through indentation.

The body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as `True`. `None` and `0` are interpreted as `False`.

### Flowchart of while Loop

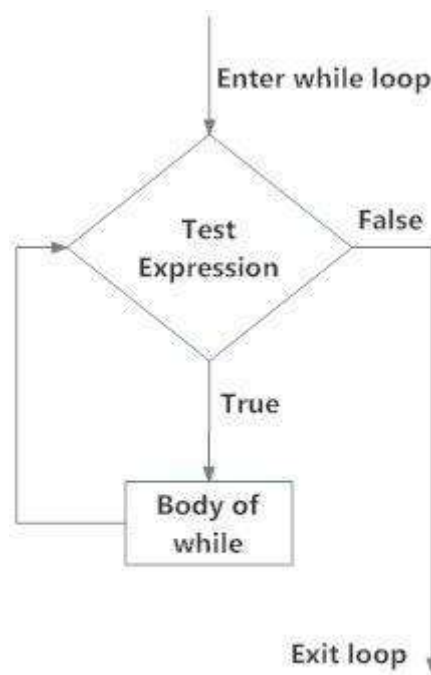


Fig: operation of while loop

## Example: Python while Loop

```
# Program to add natural
# numbers up to
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i + 1 # update counter

# print the sum
print("The sum is", sum)
```

When you run the program, the output will be:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be `True` as long as our counter variable `i` is less than or equal to `n` (10 in our program).

We need to increase the value of the counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never-ending loop).

Finally, the result is displayed.

## While loop with else

Same as with for loops, while loops can also have an optional `else` block.

The `else` part is executed if the condition in the while loop evaluates to `False`.

The while loop can be terminated with a break statement. In such cases, the `else` part is ignored.

Hence, a while loop's `else` part runs if no break occurs and the condition is false.

Here is an example to illustrate this.

```
''' Example to illustrate
the use of else statement
with the while loop '''
```

```
counter = 0

while counter <3:
print("Inside loop")
    counter = counter + 1
else:
print("Inside else")
Run Code
```

## Output

```
Inside loop
Inside loop
Inside loop
Inside else
```

Here, we use a counter variable to print the string `Inside loop` three times.

On the fourth iteration, the condition in `while` becomes `False`. Hence, the `else` part is executed.

---

## Topic 3: Iterate with for

**Possible Question: Explain Iterate with for statement?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

### Syntax of for Loop

```
for val in sequence:
    loop body
```

Here, `val` is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

### Flowchart of for Loop

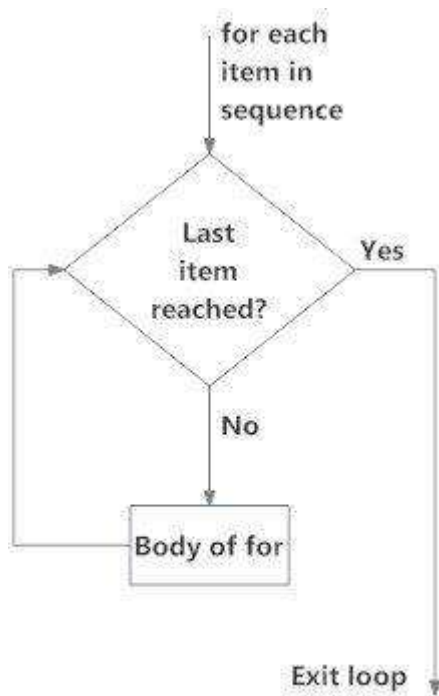


Fig: operation of for loop

### Example: Python for Loop

# Program to find the sum of all numbers stored in a list

# List of numbers

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
```

# variable to store the sum

```
sum = 0
```

# iterate over the list

```
for val in numbers:
```

```
    sum = sum+val
```

```
print("The sum is", sum)
```

[Run Code](#)

When you run the program, the output will be:

The sum is 48

### The range() function

We can generate a sequence of numbers using `range()` function. `range(10)` will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as `range(start, stop, step_size)`. `step_size` defaults to 1 if not provided.

The `range` object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports `in`, `len` and `getitem` operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function `list()`.

The following example will clarify this.

```
print(range(10))  
  
print(list(range(10)))  
  
print(list(range(2, 8)))  
  
print(list(range(2, 20, 3)))  
Run Code
```

### Output

```
range(0, 10)  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
[2, 3, 4, 5, 6, 7]  
[2, 5, 8, 11, 14, 17]
```

We can use the `range()` function in `for` loops to iterate through a sequence of numbers. It can be combined with the `len()` function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing  
  
genre = ['pop', 'rock', 'jazz']  
  
# iterate over the list using index  
for i in range(len(genre)):  
    print("I like", genre[i])  
Run Code
```

### Output

```
I like pop  
I like rock  
I like jazz
```

### for loop with else

A `for` loop can have an optional `else` block as well. The `else` part is executed if the items in the sequence used in for loop exhausts.

The `break` keyword can be used to stop a for loop. In such cases, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

```
digits = [0, 1, 5]
```

```
foriin digits:
```

```
print(i)
```

```
else:
```

```
print("No items left.")
```

When you run the program, the output will be:

```
0
```

```
1
```

```
5
```

```
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the `else` and prints `No items left.`

This `for...else` statement can be used with the `break` keyword to run the `else` block only when the `break` keyword was not executed. Let's take an

#### **example:**

```
# program to display student's marks from record
```

```
student_name = 'Soyuj'
```

```
marks = {'James': 90, 'Jules': 55, 'Arthur': 77}
```

```
for student in marks:
```

```
if student == student_name:
```

```
print(marks[student])
```

```
break
```

```
else:
```

```
print('No entry with that name found.')
```

#### **Output**

```
No entry with that name found.
```

---

## **Topic 4: Comprehensions**

**Possible Question: Describe about Comprehensions?**

**Possible Marks: 5 or 10 Marks**

## Outcomes:

We can create new sequences using a given python sequence. This is called comprehension. It basically a way of writing a concise code block to generate a sequence which can be a list, dictionary, set or a generator by using another sequence. It may involve multiple steps of conversion between different types of sequences.

### List Comprehension

In this method, we create a new list by manipulating the values of an existing list. In the below example we take a list and create a new list by adding 3 to each element of the given list.

Example

```
given_list=[x for x in range(5)]
print(given_list)

new_list=[var+3 for var in given_list]

print(new_list)
```

Output

Running the above code gives us the following result –

```
[0, 1, 2, 3, 4]
[3, 4, 5, 6, 7]
```

### Dictionary Comprehensions

Similar to the above we can take in a list and create a dictionary from it.

Example

```
given_list=[x for x in range(5)]
print(given_list)

#new_list = [var+3 for var in given_list]
new_dict={ var:var+3 for var in given_list}

print(new_dict)
```

Output

Running the above code gives us the following result –

```
[0, 1, 2, 3, 4]
{0: 3, 1: 4, 2: 5, 3: 6, 4: 7}
```

We can also take in two lists and create a new dictionary out of it.

Example

```
list1 =[x for x in range(5)]
list2=['Mon','Tue','Wed','Thu','Fri']
print(list1)
print(list2)

new_dict={key:value for (key,value) in zip(list1, list2)}

print(new_dict)
```

Output

Running the above code gives us the following result –

```
[0, 1, 2, 3, 4]
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
{0: 'Mon', 1: 'Tue', 2: 'Wed', 3: 'Thu', 4: 'Fri'}
```

### Set Comprehension

We can take a similar approach as above and create new set from existing set or list. In the below example we create a new set by adding 3 to the elements of the existing set.

Example

```
given_set={x for x in range(5)}
print(given_set)
```



```
new_set={var+3forvaringiven_set}
```

```
print(new_set)
```

#### Output

Running the above code gives us the following result –

```
{0, 1, 2, 3, 4}
```

```
{3, 4, 5, 6, 7}
```

### Generator comprehension

New generators can be created from the existing list. These generators are memory efficient as they allocate memory as the items are generated instead of allocating it at the beginning.

#### Example

```
given_list=[x for x inrange(5)]
```

```
print(given_list)
```

```
new_set=(var+3forvaringiven_list)
```

```
for var1 innew_set:
```

```
    print(var1,end=" ")
```

#### Output

Running the above code gives us the following result –

```
[0, 1, 2, 3, 4]
```

```
3 4 5 6 7
```

---

## Topic 5: Functions

### Possible Question: Describe about Functions?

### Possible Marks: 5 or 10 Marks

#### Outcomes:

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions*.

#### 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 behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

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

### 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 printme() function –

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

```
# Function definition is here
```

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

```
# Now you can call printme function
```

```
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

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

```
I'm first call to user defined function!
```

```
Again second call to the same function
```

### Pass by reference vs value

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. For 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.

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

```
# 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]

### Function Arguments

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

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

#### 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 gives a syntax error as follows –

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

```
# Function definition is here
```

```
def printme( str ):
```

```
    "This prints a passed string into this function"
```

```
    print str
```

```
    return;
```

```
# Now you can call printme function
```

```
printme()
```

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

Traceback (most recent call last):

File "test.py", line 11, in <module>

```
printme();
```

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

### **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
```

```
# Function definition is here
```

```
def printme( str ):
```

```
    "This prints a passed string into this function"
```

```
    print str
```

```
    return;
```

```
# Now you can call printme function
```

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

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

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

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

```
# Function definition is here
```

```
def printinfo( name, age ):
```

```
    "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" )
```

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

```
Name: miki
```

```
Age 50
```

### **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. The following example gives an idea on default arguments, it prints default age if it is not passed –

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

```
# Function definition is here
```

```
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
```

### **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.

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 holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example –

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

```
# Function definition is here
```

```
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
```

### **The Anonymous Functions**

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression

- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

### Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows –

```
lambda [arg1 [,arg2,... argn]]:expression
```

Following is the example to show how *lambda* form of function works –

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

```
# Function definition is here
```

```
sum = lambda arg1, arg2: arg1 + arg2;
```

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

```
print "Value of total : ", sum( 10, 20 )
```

```
print "Value of total : ", sum( 20, 20 )
```

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

```
Value of total : 30
```

```
Value of total : 40
```

### The return Statement

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`.

All the above examples are not returning any value. You can return a value from a function as follows –

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

```
# 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 –

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

```
total = 0; # This is global variable.  
# Function definition is here  
def sum( arg1, arg2 ):  
    # Add both the parameters and return them."  
    total = arg1 + arg2; # Here total is local variable.  
    print "Inside the function local total : ", total  
    return total;
```

```
# Now you can call sum function  
sum( 10, 20 );  
print "Outside the function global total : ", total
```

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

```
Inside the function local total : 30  
Outside the function global total : 0
```

---

## Topic 6:Generators

**Possible Question: Explain in detail about Generators?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Generators have been an important part of python ever since they were introduced with PEP 255.

Generator in python are special routine that can be used to control the iteration behaviour of a loop. A generator is similar to a function returning an array. A generator has parameter, which we can call and it generates a sequence of numbers. But unlike functions, which return a whole array, a generator yields one value at a time which requires less memory.

Any python function with a keyword “yield” may be called as generator. A normal python function starts execution from first line and continues until we get a return statement or an exception or end of the function however, any of the local variables created during the function scope are destroyed and not accessible further. While in case of generator when it encounters a yield keyword the state of the function is frozen and all the variables are stored in memory until the generator is called again.

We can use generator in accordance with an iterator or can be explicitly called using the “next” keyword.

**Generally generators in Python:**

- Defined with the def keyword
- Use the yield keyword
- May contain several yield keywords.
- Returns an iterator.

**Generators with Iterators**

```

defgenerator_thr_iter():
    yield'xyz'
    yield246
    yield40.50
foriingenerator_thr_iter():
    print(i)

```

Output

```

xyz
246
40.5

```

### Generator using next

```

defgenerator_thr_iter():
    yield'xyz'
    yield246
    yield40.50
>>> g =generator_thr_iter()
>>>g._next_()
'xyz'
>>>g._next_()
246
>>>g._next_()
40.5
>>>g.__next__()
Traceback(most recent call last):
File"<pyshell#39>", line 1,in<module>
g._next_()
StopIteration

```

We can think of generators as the one returning multiple items one by one instead of all at once and the generator function is paused until the next item is requested.

### Program to print square of numbers from 1 to n

Consider we want to calculate the square of number from 1 to n, where n is really big number, such that creating a list of numbers up to 'n' would occupy the entire system memory space.

Without generator, our approach will be something like -

```

>>> n=200000000000
>>>number_list=range(1, n+1)
>>>foriinnumber_list:
    print(i*i)

```

Above approach will consume lot of system memory. Better approach would be, is to iterate over the numbers without ever creating the list of numbers so that the system memory isn't occupied. Here comes the use of generators.

Our generator program for the same would be -

```

defnum_generator(n):
    num=1
    whileTrue:
        yieldnum
        ifnum== n:
            return
        else:
            num+=1
forinum_generator(200000000000):
    print(i*i)

```

So in above approach, when the for loop is first initialised the num\_generator is called and the value of n = 200000000000 is stored in memory and num=1 is initialised and is entered into while loop which loops forever. Then the yield num is encountered, at this time the while loop



is frozen and all the local variables are stored in memory. Since num=1, yield num is returned to the for loop and is assigned to I, where 1(i\*i) is printed and the next call to num\_generator is made.

Now the execution starts from the point where it has frozen previously, so it executes the line num == n (1 == 200000000000), which is false so num +=1 is executed which comes to num = 2 and the while loop is executed once again and the process continues.

Finally while loop is executed till n=200000000000, when 200000000000 is yielded then the next line 'num == n'(200000000000 == 200000000000) is executed, since it is true the return statement is executed.

So when generator executes a return statement or encounters exception or reached end of the generator the "StopIteration" exception is raised and the for loop iteration stops at the moment. So above we are able to print square of number upto 200000000000 without ever creating a big list of numbers which would have occupied large system memory.

Consider above scenario, we could use generators in our daily programming practice to create more efficient program.>

---

## Topic 7: Decorators

**Possible Question: Explain detail about Decorators?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

Python developers can extend and modify the behavior of a callable functions, methods or classes without permanently modifying the callable itself by using decorators. In short we can say they are callable objects which are used to modify functions or classes.

Function decorators are functions which accepts function references as arguments and adds a wrapper around them and returns the function with the wrapper as a new function.

Let's understand function decorator by an example:

#### Code1

```
@decorator
def func(arg):
    return "value"
```

Above code is same as:

#### Code2

```
def func(arg):
    return "value"
func = decorator(func)
```

So from above, we can see a decorator is simply another function which takes a function as an argument and returns one.

Decorators basically "decore" or "wrap" another function and let you execute code before and after the wrapped function runs as explained in below example:

```
defour_decorator(func):
    deffunction_wrapper(x):
        print("Before calling "+func._name_)
        func(x)
        print("After calling "+func._name_)
    returnfunction_wrapper

def foo(x):
    print("Hi, foo has been called with "+ str(x))
```

```
print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo =our_decorator(foo)

print("We call foo after decoration:")
foo(90)
```

## Output

```
We call foo before decoration:
Hi, foo has been called with Hi
We now decorate foo with f:
We call foo after decoration:
Before calling foo
Hi, foo has been called with 90
After calling foo
```

If you think above is little bit complex, let write the simplest possible decorator:

```
def null_decorator(func):
    return func
```

Above `null_decorator` is a callable(function), it takes another callable as its input and it returns the same input callable without modifying it.

Let's extend our above simplest decorator by decorating (or wrap) another function.

```
def null_decorator(func):
    return func
def greet():
    return "Hello, Python!"

greet = null_decorator(greet)
>>>greet()
'Hello, Python!'
```

Above we have defined a `greet` function and then immediately decorated it by running it through the `null_decorator` function.

Much simpler way to writing above python decorative program (instead of explicitly calling `null_decorator` on `greet` and then reassigning the `greet` variable) is to use python `@syntax` for decorating a function in one step:

```
@null_decorator
def greet():
    return "Hello, Python!"

>>>greet()
```

```
'Hello, Python!'
```

---

## Topic 8:Namespaces and Scope

**Possible Question: Explain detail about Namespaces and Scope?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement `global VarName` tells Python that `VarName` is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable `Money` in the global namespace. Within the function `Money`, we assign `Money` a value, therefore Python assumes `Money` as a local variable. However, we accessed the value of the local variable `Money` before setting it, so an `UnboundLocalError` is the result. Uncommenting the global statement fixes the problem.

```
#!/usr/bin/python
Money=2000
defAddMoney():
# Uncomment the following line to fix the code:
# global Money
Money=Money+1
printMoney
AddMoney()
printMoney
```

---

## Topic 9:Handle Errors with try and except

**Possible Question: Explain Handle Errors with try and except?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

To use exception handling in python, we first need to catch the all except clauses.

Python provides, “try” and “except” keywords to catch exceptions. The “try” block code will be executed statement by statement. However, if an exception occurs, the remaining “try” code will not be executed and the except clause will be executed.

```
try:
    some_statements_here
except:
    exception_handling
```

Let's see above syntax with a very simple example –

```
try:
    print("Hello, World!")
except:
    print("This is an error message!")
```

### **Output**

Hello, World!

Above is a very simple example, let's understand the above concept with another example –

```
import sys
List=['abc',0,2,4]
for item inList:
    try:
        print("The List Item is", item)
        r =1/int(item)
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print('\n')
        print("Next Item from the List is: ")
        print()
print("The reciprocal of",item,"is",r)
```

### **Output**

The List Item is abc

Oops! <class 'ValueError'>occured.

Next Item from the List is:

The List Item is 0

Oops! <class 'ZeroDivisionError'>occured.

Next Item from the List is:

The List Item is 2

The reciprocal of 2 is 0.5

In the above program, the loops run until we get (as user input) an integer that has a valid reciprocal. The code which causes an exception to raise is placed within the try block.

In case some exception occurs, it will be caught by the except block. We can test the above program with different exception errors. Below are some of the common exception errors –

- **IOError**  
Raised in case we cannot open the file.
- **ImportError**  
Raised in case module is missing.
- **ValueError**  
It happened whenever we pass the argument with the correct type but an inappropriate value of a built-in operator or function.
- **KeyboardInterrupt**  
Whenever the user hits the interrupt key (generally control-c)
- **EOFError**  
Exception raised when the built-in functions hit an end-of-file condition (EOF) without reading any data.

---

## Topic 10:User Exceptions

### Possible Question: Describe about User Exceptions?

Possible Marks: 5 or 10 Marks

#### Outcomes:

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

#### Handling an exception

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 –

```
try:
You do your operations here;
.....
except ExceptionI:
If there is ExceptionI, then execute this block.
except ExceptionII:
If there is ExceptionII, 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

This example opens a file, writes 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!!")
exceptIOError:
print"Error: can't find file or read data"
else:
```

```
print"Written content in the file successfully"
fh.close()
```

This produces the following result –

```
Written content in the file successfully
```

Example

This example tries to open a file where you do not have write permission, so it raises an exception –

```
#!/usr/bin/python

try:
fh=open("testfile","r")
fh.write("This is my test file for exception handling!!")
exceptIOError:
print"Error: can't find file or read data"
else:
print"Written content in the file successfully"
```

This produces 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.

### 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.
```

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.
.....
```

You cannot 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 –

```
#!/usr/bin/python

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()
exceptIOError:
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;
.....
exceptExceptionType,Argument:
You can printvalueofArgument here...
```

If you write 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 receives 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 produces the following result –

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

## Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

### 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 –

```
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 the except clause as follows –

```
try:
    BusinessLogic 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 subclassed 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 the exception as follows –

```
try:
    raise Networkerror("Bad hostname")
except Networkerror, e:
    print e.args
```

---

## Topic 11: Modules, Packages, and Programs

**Possible Question: Describe about Modules, Packages, and Programs?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, *support.py*

```
def print_func(par):
    print "Hello : ", par
    return
```

### The 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 *support.py*, you need to put the following command at the top of the script –

```
#!/usr/bin/python

# Import module support
import support
```

```
# Now you can call defined function that module as follows
support.print_func("Zara")
```

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

```
Hello : Zara
```

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.

### The `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.

### The `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; however, this statement should be used sparingly.

### Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences –

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable `PYTHONPATH`.
- If all else fails, Python checks the default path. On UNIX, this default path is normally `/usr/local/lib/python/`.

The module search path is stored in the system module `sys` as the `sys.path` variable. The `sys.path` variable contains the current directory, `PYTHONPATH`, and the installation-dependent default.

### The `PYTHONPATH` Variable

The `PYTHONPATH` is an environment variable, consisting of a list of directories. The syntax of `PYTHONPATH` is the same as that of the shell variable `PATH`.

Here is a typical `PYTHONPATH` from a Windows system –

```
set PYTHONPATH = c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system –

```
set PYTHONPATH = /usr/local/lib/python
```

## Namespaces and Scoping

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the `global` statement.

The statement `global VarName` tells Python that `VarName` is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable `Money` in the global namespace. Within the function `Money`, we assign `Money` a value, therefore Python assumes `Money` as a local variable. However, we accessed the value of the local variable `Money` before setting it, so an `UnboundLocalError` is the result. Uncommenting the global statement fixes the problem.

```
#!/usr/bin/python

Money=2000
defAddMoney():
# Uncomment the following line to fix the code:
# global Money
Money=Money+1

printMoney
AddMoney()
printMoney
```

## The `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.

### The `globals()` and `locals()` Functions

The `globals()` and `locals()` functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If `locals()` is called from within a function, it will return all the names that can be accessed locally from that function.

If `globals()` is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the `keys()` function.

### The `reload()` Function

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the `reload()` function. The `reload()` function imports a previously imported module again. The syntax of the `reload()` function is this –

```
reload(module_name)
```

Here, `module_name` is the name of the module you want to reload and not the string containing the module name. For example, to reload `hello` module, do the following –

```
reload(hello)
```

### Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file `Pots.py` available in `Phone` directory. This file has following line of source code –

```
#!/usr/bin/python  
  
defPots():  
    print"I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above –

- `Phone/Isdn.py` file having function `Isdn()`
- `Phone/G3.py` file having function `G3()`

Now, create one more file `__init__.py` in *Phone* directory –

- `Phone/__init__.py`

To make all of your functions available when you've imported *Phone*, you need to put explicit import statements in `__init__.py` as follows –

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to `__init__.py`, you have all of these classes available when you import the *Phone* package.

```
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

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

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

---

## Topic 12: Standalone Programs

**Possible Question: Explain in detail about Standalone Programs?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

With `BuildApplet` you can build a standalone Python application that works like any other Mac application: you can double-click it, run it while the Python interpreter is running other scripts, drop files on it, etc. It is, however, still dependent on the whole Python installation on your machine: the PythonCore engine, the plugin modules and the various Lib folders.

In some cases you may want to create a true application, for instance because you want to send it off to people who may not have Python installed on their machine, or because you the application is important and you do not want changes in your Python installation like new versions to influence it.

### The easy way

The easiest way to create an application from a Python script is simply by dropping it on the `BuildApplication` applet in the main Python folder. `BuildApplication` has a similar interface as `BuildApplet`: you drop a script on it and it will process it, along with an optional `.rsrc` file.

What `BuildApplication` does, however, is very different. It parses your script, recursively looking for all modules you use, bundles the compiled code for all these modules in `PYC`

resources, adds the executable machine code for the PythonCore engine, any dynamically loaded modules you use and a main program, combines all this into a single file and adds a few preference resources (which you can inspect with EditPythonPrefs, incidentally) to isolate the new program from the existing Python installation.

Usually you do not need to worry about all this, but occasionally you may have to exercise some control over the process, for instance because your program imports modules that don't exist (which can happen if your script is multi-platform and those modules will never be used on the Mac). See the section on [directives](#) below for details. If you get strange error messages about missing modules it may also be worthwhile to run `macfreeze` in report mode on your program, see below.

### **Doing it the hard way**

With the `macfreeze` script, for which `BuildApplication` is a simple wrapper, you can go a step further and create CodeWarrior projects and sourcefiles which can then be used to build your final application. While `BuildApplication` is good enough for 90% of the use cases there are situations where you need `macfreeze` itself, mainly if you want to embed your frozen Python script into an existing C application, or when you need the extra bit of speed: the resulting application will start up a bit quicker than one generated with `BuildApplication`.

When you start `Mac:Tools:macfreeze:macfreeze.py` you are asked for the script file, and you can select which type of freeze to do. The first time you should always choose *report only*, which will produce a listing of modules and where they are included from in the console window. `Macfreeze` actually parses all modules, so it may crash in the process. If it does try again with a higher debug value, this should show you where it crashes.

### **Directives**

For more elaborate programs you will often see that freeze includes modules you don't need (because they are for a different platform, for instance) or that it cannot find all your modules (because you modify `sys.path` early in your initialization). It is possible to include directives to tell `macfreeze` to add items to the search path and include or exclude certain modules. All your directives should be in the main script file.

Directives have the following form:

```
# macfreeze: command argument
```

The trigger `macfreeze:` must be spelled exactly like that, but the whitespace can be any combination of spaces and tabs. `Macfreeze` understands the following directives:

`path`

Prepend a folder to `sys.path`. The argument is a pathname, which should probably be relative (starting with a colon) and is interpreted relative to the folder where the script lives.

`include`

Include a module. The module can either be given by filename or by module name, in which case it is looked up through the normal method.

`exclude`

Exclude a module. The module must be given by `modulename`. Even when freeze deems the module necessary it will not be included in the application.

`optional`

Include a module if it can be found, but don't complain if it can't.

There is actually a fourth way that `macfreeze` can operate: it can be used to generate only the resource file containing the compiled PYC resources. This may be useful if you have embedded Python in your own application. The resource file generated is the same as for the CodeWarrior generation process.

## Freezing with CodeWarrior

To freeze with CodeWarrior you need CodeWarrior, obviously, and a full source distribution of Python. You select the Code warrior source and project option. You specify an output folder, which is by default the name of your script with .py removed and build. prepended. If the output folder does not exist yet it is created, and a template project file and bundle resource file are deposited there. Next, a source file mac freeze config.c is created which includes all builtin modules your script uses, and a resource file frozen modules.rsrc which contains the PYC resources for all your Python modules.

The project expects to live in a folder one level below the Python root folder, so the next thing you should do is move the build folder there. It is a good idea to leave an alias with the same name in the original location: when you run freeze again it will regenerate the frozen modules.rsrc file but not the project and bundle files. This is probably what you want: if you modify your python sources you have to re-freeze, but you may have changed the project and bundle files, so you don't want to regenerate them.

An alternative is to leave the build folder where it is, but then you have to adapt the search path in the project.

The project is set up to include all the standard builtin modules, but the CW linker is smart enough to exclude any object code that isn't referenced. Still, it may be worthwhile to remove any sources for modules that you are sure are not used to cut back on compilation time. You may also want to examine the various resource files (for Tcl/Tk, for instance): the loader has no way to know that these aren't used.

You may also need to add sourcefiles if your script uses non-standard builtin modules, like anything from the Extensions folder.

The frozenbundle.rsrc resource file contains the bundle information. It is almost identical to the bundle file used for applets, with the exception that it sets the sys.path initialization to \$(APPLICATION) only. This means that all modules will only be looked for in PYC resources in your application.

---

## Topic 13: Command-Line Arguments

**Possible Question: Explain in detail about Command-Line Arguments?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Python provides a **getopt** module that helps you parse command-line options and arguments.

```
$ python test.py arg1 arg2 arg3
```

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes –

- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.

Here **sys.argv[0]** is the program ie. script name.

Example

Consider the following script test.py –

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

```
import sys

print'Number of arguments:',len(sys.argv),'arguments.'
print'Argument List:', str(sys.argv)
```

Now run above script as follows –

```
$ python test.py arg1 arg2 arg3
```

This produce following result –

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

**NOTE** – As mentioned above, first argument is always script name and it is also being counted in number of arguments.

## Parsing Command-Line Arguments

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

### getopt.getopt method

This method parses command line options and parameter list. Following is simple syntax for this method –

```
getopt.getopt(args, options, [long_options])
```

Here is the detail of the parameters –

- **args** – This is the argument list to be parsed.
- **options** – This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long\_options** – This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
- This method returns value consisting of two elements: the first is a list of (**option, value**) pairs. The second is the list of program arguments left after the option list was stripped.
- Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

### Exception getopt.GetoptError

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option.

#### Example

Consider we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows –

```
usage: test.py -i<inputfile> -o <outputfile>
```

Here is the following script to test.py –

```
#!/usr/bin/python

import sys,getopt
```



```

def main(argv):
inputfile=""
outputfile=""
try:
    opts,args=getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
exceptgetopt.GetoptError:
print'test.py -i<inputfile> -o <outputfile>'
sys.exit(2)
for opt,argin opts:
if opt =='-h':
print'test.py -i<inputfile> -o <outputfile>'
sys.exit()
elif opt in("-i","--ifile"):
inputfile=arg
elif opt in("-o","--ofile"):
outputfile=arg
print'Input file is "',inputfile
print'Output file is "',outputfile

if __name__=="_main_":
    main(sys.argv[1:])

```

Now, run above script as follows –

```

$ test.py -h
usage: test.py -i<inputfile>-o <outputfile>

$ test.py -i BMP -o
usage: test.py -i<inputfile>-o <outputfile>

$ test.py -iinputfile
Input file is" inputfile
Output file is "

```

---

## Topic 14: Modules and the import Statement

**Possible Question: Explain detail about Modules and the import Statement?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

A module is basically a file which has many lines of python code that can be referred or used by other python programs. A big python program should be organized to keep different parts of the program in different modules. That helps in all aspects like debugging, enhancements and packaging the program efficiently. To use a module in any python program we should first import it to the new program. All the functions, methods etc. from this module then will be available to the new program.

### With import statement

Let's create a file named profit.py which contains program for a specific calculation as shown below.

Example

```

defgetprofit(cp,sp):
    result =((sp-cp)/cp)*100
    return result

```

Next we want to use the above function in another python program. We can then use the import function in the new program to refer to this module and its function named getprofit.

### Example

```
import profit

perc=profit.getprofit(350,500)
print(perc)
```

### Output

Running the above code gives us the following result –

```
42.857142857142854
```

### With From Module Import

We can also import only a specific method from a module instead of the entire module. For that we use the from Module import statement as shown below. In the below example we import the value of pi from math module to be used in some calculation in the program.

### Example

```
from math import pi

x =30*pi
print(x)
```

### Output

Running the above code gives us the following result –

```
94.24777960769379
```

### Investigating modules

If we want to know the location of various inbuilt modules we can use the sys module to find out. Similarly to know the various function available in a module we can use the dir method as shown below.

### Example

```
import sys
import math

print(sys.path)
print(dir(math))
```

### Output

Running the above code gives us the following result –

```
['',
'C:\\Windows\\system32\\python38.zip',
'C:\\Python38\\DLLs',
'C:\\Python38\\lib',
'C:\\Python38',
'C:\\Python38\\lib\\site-packages']

['.....log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod',.....]
```

---

## Topic 15: The Python Standard Library

**Possible Question: Explain detail about The Python Standard Library?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

Python has a standard library, which includes a wide variety of routines that help you code and reuse these codes easily. A **Module** is a Python file that has definitions of variables and a set of related routines or functions.

Some of the core **Modules** provided by the Python library are as follows –

- **Built-in Functions and Exceptions** – Python imports both these modules when it starts up and makes their content available for all programs. The built-in

module defines built-in functions like **len**, **int**, **range**, while the exceptions module defines all built-in exceptions.

- **Operating System Interface Modules** – The OS module makes available, the functions that enables performing OS level operations through scripts.
- **Type Support Modules** – Type support modules include string module-to implement, commonly used string operations; math module provides mathematical operations etc.
- **Regular Expressions** – Regular Expressions are string patterns written in a specific syntax, which can be used to match or extract strings or substrings. The re module provides Regex support for Python.
- **Language Support Modules** – The sys module gives you access to various interpreter variables, such as the module search path, and the interpreter version. The operator module provides functional equivalents to many built-in operators. The copy module allows you to copy objects. Finally, the gc module gives you more control over the garbage collector facilities in python 2.0.

### About JSON

The JavaScript Object Notation (JSON) is a data-interchange format. Though many programming languages support JSON, it is especially useful for JavaScript-based apps, including websites and browser extensions. JSON can represent numbers, Booleans, strings, null, arrays (ordered sequences of values), and objects (string-value mappings) made up of these values (or of other arrays and objects).

---

## Topic 16: Objects and Classes

**Possible Question: Explain Objects and Classes?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

To understand the need for creating a class in Python let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

### Class Definition Syntax:

```
class ClassName:  
    # Statement
```

### Object Definition Syntax:

```
obj = ClassName()  
print(obj.attr)
```

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

### Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.:  
Myclass.Myattribute

### Defining a class

```
# Python3 program to
# demonstrate defining
# a class
```

```
class Dog:
    pass
```

In the above example, the class keyword indicates that you are creating a class followed by the name of the class (Dog in this case).

### Class Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of :

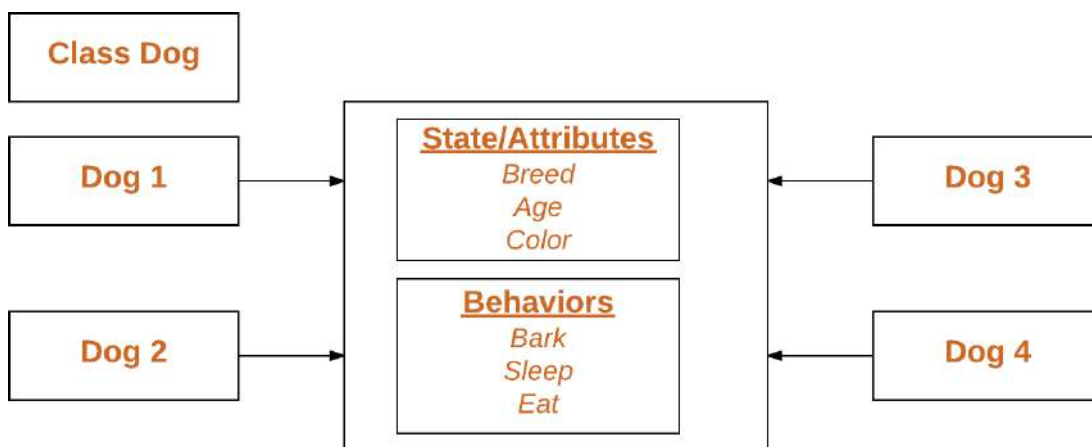
- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.



### Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

**Example:**



### Declaring an object

```
# Python3 program to
```

```
# demonstrate instantiating
# a class
```

```
classDog:
```

```
    # A simple class
    # attribute
    attr1 ="mammal"
    attr2 ="dog"
```

```
    # A sample method
    deffun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)
```

```
# Driver code
# Object instantiation
Rodger =Dog()
```

```
# Accessing class attributes
# and method through objects
print(Rodger.attr1)
Rodger.fun()
```

### **Output:**

```
mammal
I'm a mammal
I'm a dog
```

In the above example, an object is created which is basically a dog named Rodger. This class only has two class attributes that tell us that Rodger is a dog and a mammal.

### **The self**

- Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method that takes no arguments, then we still have to have one argument.
- This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` – this is all the special self is about.

### **\_\_init\_\_ method**

The `__init__` method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

- Python3

```
# A Sample class with init method
```

```
classPerson:
```

```
    # init method or constructor
```

```

def __init__(self, name):
    self.name =name

# Sample Method
def say_hi(self):
    print('Hello, my name is', self.name)

```

```

p =Person('Nikhil')
p.say_hi()

```

### **Output:**

Hello, my name is Nikhil

### **Class and Instance Variables**

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Defining instance variables using a constructor.

```

# Python3 program to show that the variables with a value
# assigned in the class declaration, are class variables and
# variables inside methods and constructors are instance
# variables.

```

```

# Class for Dog

```

```

class Dog:

```

```

    # Class Variable
    animal ='dog'

```

```

    # The init method or constructor
    def __init__(self, breed, color):

```

```

        # Instance Variable
        self.breed=breed
        self.color=color

```

```

# Objects of Dog class
Rodger =Dog("Pug", "brown")
Buzo=Dog("Bulldog", "black")

```

```

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

```

```

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)

```

```
print('Color: ', Buzo.color)

# Class variables can be accessed using class
# name also
print("\nAccessing class variable using class name")
print(Dog.animal)
```

**Output:**

Rodger details:  
Rodger is a dog  
Breed: Pug  
Color: brown

Buzo details:  
Buzo is a dog  
Breed: Bulldog  
Color: black

Accessing class variable using class name  
dog  
Defining instance variables using the normal method.

```
# Python3 program to show that we can create
# instance variables inside methods
```

```
# Class for Dog
```

```
classDog:
```

```
    # Class Variable
    animal ='dog'
```

```
    # The init method or constructor
    def __init__(self, breed):
```

```
        # Instance Variable
        self.breed=breed
```

```
    # Adds an instance variable
    def setColor(self, color):
        self.color=color
```

```
    # Retrieves instance variable
    def getColor(self):
        returnself.color
```

```
# Driver Code
Rodger =Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

**Output:** brown

---

## Topic 17: Define a Class with class

**Possible Question: Describe about Define a Class with class?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Python is a completely object-oriented language. You have been working with classes and objects right from the beginning of these tutorials. Every element in a Python program is an object of a class. A number, string, list, dictionary, etc., used in a program is an object of a corresponding built-in class. You can retrieve the class name of variables or objects using the `type()` method, as shown below.

Example: Python Built-in Classes

```
>>> num=20
>>> type(num)
<class 'int'>
>>> s="Python"
>>> type(s)
<class 'str'>
```

### Defining a Class

A class in Python can be defined using the `class` keyword.

```
class <ClassName>:
```

```
    <statement1>
```

```
    <statement2>
```

```
    .
```

```
    .
```

```
    <statementN>
```

As per the syntax above, a class is defined using the `class` keyword followed by the class name and `:` operator after the class name, which allows you to continue in the next indented line to define class members. The followings are class members.

1. Class Attributes
2. Constructor
3. Instance Attributes
4. Properties
5. Class Methods

A class can also be defined without any members. The following example defines an empty class using the `pass` keyword.

Example: Define Python Class

```
class Student:
```

```
    pass
```

Class instantiation uses function notation. To create an object of the class, just call a class like a parameterless function that returns a new object of the class, as shown below.

Example: Creating an Object of a Class

```
std = Student()
```

Above, `Student()` returns an object of the `Student` class, which is assigned to a local variable `std`. The `Student` class is an empty class because it does not contain any members.



## Class Attributes

Class attributes are the variables defined directly in the class that are shared by all objects of the class. Class attributes can be accessed using the class name as well as using the objects.

Example: Define Python Class

```
class Student:
    schoolName = 'XYZ School'
```

Above, the `schoolName` is a class attribute defined inside a class. The value of the `schoolName` will remain the same for all the objects unless modified explicitly.

Example: Define Python Class

```
>>> Student.schoolName
'XYZ School'
>>> std = Student()
>>> std.schoolName
'XYZ School'
```

As you can see, a class attribute is accessed by `Student.schoolName` as well as `std.schoolName`. Changing the value of class attribute using the class name would change it across all instances. However, changing class attribute value using instance will not reflect to other instances or class.

Example: Define Python Class

```
>>> Student.schoolName = 'ABC School' # change attribute value using class name
>>> std = Student()
>>> std.schoolName
'ABC School' # value changed for all instances
>>> std.schoolName = 'My School' # changing instance's attribute
>>> std.schoolName
'My School'
>>> Student.schoolName # instance level change not reflectd to class attribute
'ABC School'
>>> std2 = Student()
>>> std2.schoolName
'ABC School'
```

The following example demonstrates the use of class attribute `count`.

Example: Student.py

```
class Student:
    count = 0
    def __init__(self):
        Student.count += 1
```

In the above example, `count` is an attribute in the Student class. Whenever a new object is created, the value of `count` is incremented by 1. You can now access the `count` attribute after creating the objects, as shown below.

Example:

```
>>> std1=Student()
>>> Student.count
1
>>> std2 = Student()
>>> Student.count
2
```

## Constructor

In Python, the constructor method is invoked automatically whenever a new object of a class is instantiated, same as constructors in C# or Java. The constructor must have a special name `__init__()` and a special parameter called `self`.

The first parameter of each method in a class must be the `self`, which refers to the calling object. However, you can give any name to the first parameter, not necessarily `self`.

The following example defines a constructor.

Example: Constructor

```
class Student:
    def __init__(self): # constructor method
        print('Constructor invoked')
```

Now, whenever you create an object of the `Student` class, the `__init__()` constructor method will be called, as shown below.

Example: Constructor Call on Creating Object

```
>>>s1 = Student()
Constructor invoked
>>>s2 = Student()
Constructor invoked
```

The constructor in Python is used to define the attributes of an instance and assign values to them.

## Instance Attributes

Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

The following example defines instance attributes `name` and `age` in the constructor.

Example: Instance Attributes

```
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self): # constructor
        self.name = " " # instance attribute
        self.age = 0 # instance attribute
```

An instance attribute can be accessed using dot notation: `[instance name].[attribute name]`, as shown below.

Example:

```
>>> std = Student()
>>> std.name
"
>>> std.age
0
```

You can set the value of attributes using the dot notation, as shown below.

Example:

```
>>> std = Student()
>>> std.name = "Bill" # assign value to instance attribute
>>> std.age=25      # assign value to instance attribute
>>> std.name       # access instance attribute value
Bill
>>> std.age        # access value to instance attribute
25
```

You can specify the values of instance attributes through the constructor. The following constructor includes the name and age parameters, other than the self parameter.

Example: Setting Attribute Values

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Now, you can specify the values while creating an instance, as shown below.

Example: Passing Instance Attribute Values in Constructor

Copy

```
>>> std = Student('Bill',25)
>>> std.name
'Bill'
>>> std.age
25
```

You don't have to specify the value of the self parameter. It will be assigned internally in Python.

You can also set default values to the instance attributes. The following code sets the default values of the constructor parameters. So, if the values are not provided when creating an object, the values will be assigned later.

Example: Setting Default Values of Attributes

Copy

```
class Student:
    def __init__(self, name="Guest", age=25)
        self.name=name
        self.age=age
```

Now, you can create an object with default values, as shown below.

Example: Instance Attribute Default Value

```
>>> std = Student()
>>> std.name
'Guest'
>>> std.age
25
```

Visit class attributes vs instance attributes in Python for more information.

### **Class Properties**

In Python, a property in the class can be defined using the `property()` function.

The `property()` method in Python provides an interface to instance attributes. It encapsulates instance attributes and provides a property, same as Java and C#.

The `property()` method takes the `get`, `set` and `delete` methods as arguments and returns an object of the property class.

The following example demonstrates how to create a property in Python using the `property()` function.

Example: `property()`

```
class Student:
    def __init__(self):
        self._name=""
    def setname(self, name):
        print('setname() called')
        self._name=name
    def getname(self):
        print('getname() called')
        return self._name
    name=property(getname, setname)
```

In the above example, `property(getname, setname)` returns the property object and assigns it to `name`. Thus, the `name` property hides the private instance attribute `__name`. The `name` property is accessed directly, but internally it will invoke the `getname()` or `setname()` method, as shown below.

Example: `property()`

```
>>> std = Student()
>>> std.name="Steve"
setname() called
>>> std.name
getname() called
'Steve'
```

It is recommended to use the `property` decorator instead of the `property()` method.

### **Class Methods**

You can define as many methods as you want in a class using the `def` keyword. Each method must have the first parameter, generally named as `self`, which refers to the calling instance.

Example: Class Method

```
class Student:
    def displayInfo(self): # class method
        print('Student Information')
```

`Self` is just a conventional name for the first argument of a method in the class. A method defined as `mymethod(self, a, b)` should be called as `x.mymethod(a, b)` for the object `x` of the class.

The above class method can be called as a normal function, as shown below.

Example: Class Method

```
>>> std = Student()
>>> std.displayInfo()
'Student Information'
```

The first parameter of the method need not be named `self`. You can give any name that refers to the instance of the calling method. The following `displayInfo()` method names the first parameter as `obj` instead of `self` and that works perfectly fine.

Example: Class Method

```
class Student:
    def displayInfo(obj): # class method
        print('Student Information')
```

Defining a method in the class without the `self` parameter would raise an exception when calling a method.

Example: Class Method

```
class Student:
    def displayInfo(): # method without self parameter
        print('Student Information')
>>> std = Student()
>>> std.displayInfo()
Traceback (most recent call last):
std.displayInfo()
TypeError: displayInfo() takes 0 positional arguments but 1 was given
```

The method can access instance attributes using the `self` parameter.

Example: Class Method

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def displayInfo(self): # class method
        print('Student Name: ', self.name, ', Age: ', self.age)
```

You can now invoke the method, as shown below.

Example: Calling a Method

```
>>> std = Student('Steve', 25)
>>> std.displayInfo()
Student Name: Steve , Age: 25
Deleting Attribute, Object, Class
```

You can delete attributes, objects, or the class itself, using the `del` keyword, as shown below.

Example: Delete Attribute, Object, Class

```
>>> std = Student('Steve', 25)
>>> del std.name # deleting attribute
```

```

>>> std.name
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module>
std.name
AttributeError: 'Student' object has no attribute 'name'
>>> del std # deleting object
>>> std.name
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module>
std.name
NameError: name 'std' is not defined
>>> del Student # deleting class
>>> std = Student('Steve', 25)
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module>
std = Student()
NameError: name 'Student' is not defined

```

---

## Topic 18: Inheritance

**Possible Question: Describe about Inheritance?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

#### *Syntax*

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```

class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite
#!/usr/bin/python

class Parent:    # define parent class
    parentAttr = 100
    def __init__(self):
        print "Calling parent constructor"

    def parentMethod(self):
        print 'Calling parent method'

    def setAttr(self, attr):
        Parent.parentAttr = attr

    def getAttr(self):
        print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
    def __init__(self):
        print "Calling child constructor"

```

```

def childMethod(self):
    print 'Calling child method'

c = Child()      # instance of child
c.childMethod() # child calls its method
c.parentMethod() # calls parent's method
c.setAttr(200)  # again call parent's method
c.getAttr()     # again call parent's method

```

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

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

Similar way, you can drive a class from multiple parent classes as follows –

```
class A: # define your class A
```

.....

```
class B: # define your class B
```

.....

```
class C(A, B): # subclass of A and B
```

.....

You can use `issubclass()` or `isinstance()` functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of *Class*

---

## Topic 19: Override a Method

**Possible Question: Explain in detail about Override a Method?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

```

#!/usr/bin/python

class Parent:    # define parent class
    def myMethod(self):
        print 'Calling parent method'

class Child(Parent): # define child class
    def myMethod(self):
        print 'Calling child method'

c = Child()      # instance of child
c.myMethod()     # child calls overridden method

```

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

```
Calling child method
```

---

## Topic 20: Add a Method

**Possible Question: Explain detail about Add a Method?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

The normal way to add functionality (methods) to a class in Python is to define functions in the class body. There are many other ways to accomplish this that can be useful in different situations.

This is the traditional way

```
class A(object):
    def print_classname(self):
        print self.__class__.__name__
```

The method can also be defined outside the scope of the class. This allows the function “print\_classname” to be used as a standalone function and as a method of the class.

```
def print_classname(a):
    print a.__class__.__name__
```

```
class A(object):
    print_classname = print_classname
```

Or, equivalently

```
def print_classname(a):
    print a.__class__.__name__
```

```
class A(object):
    pass
```

```
setattr(A, "print_classname", print_classname)
```

Adding the method to an object of type “A” is also possible. However, you need to specify that the attribute “print\_classname” of the object is a method to make sure it will receive a reference to “self” as implicit first parameter when it is called.

```
from types import MethodType
```

```
def print_classname(a):
    print a.__class__.__name__
```

```
class A(object):
    pass
```

```
# this assigns the method to the instance a, but not to the class definition
a = A()
a.print_classname = MethodType(print_classname, a, A)
```

```
# this assigns the method to the class definition
A.print_classname = MethodType(print_classname, None, A)
```

Specific methods from another class can also be added (without inherit everything else) by adding the underlying function of the method. Otherwise the method will expect a reference to an instance of the original class as implicit first parameter.

```
class B(object):
```



```

def print_classname(self):
print self.__class__.__name__

# option 1
class A(object):
print_classname = B.print_classname.__func__

# option 2
class A(object):
pass

setattr(A, "print_classname", B.print_classname.__func__)

```

## Topic 21: Get Help from Parent with super

**Possible Question: Explain detail about Get Help from Parent with super?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

The `super()` builtin returns a proxy object (temporary object of the superclass) that allows us to access methods of the base class.

### Example

```

class Animal(object):
    def __init__(self, animal_type):
        print('Animal Type:', animal_type)

class Mammal(Animal):
    def __init__(self):
        # call superclass
        super().__init__('Mammal')

        print('Mammals give birth directly')

dog = Mammal()

# Output: Animal Type: Mammal
#         Mammals give birth directly
Run Code

```

### Use of super()

In Python, `super()` has two major use cases:

- Allows us to avoid using the base class name explicitly
- Working with Multiple Inheritance

### Example 1: super() with Single Inheritance

In the case of single inheritance, we use `super()` to refer to the base class.

```
class Mammal(object):
    def __init__(self, mammalName):
        print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):
    def __init__(self):
        print('Dog has four legs.')
        super().__init__('Dog')

d1 = Dog()
Run Code
```

### Output

```
Dog has four legs.
Dog is a warm-blooded animal.
```

Here, we called the `__init__()` method of the `Mammal` class (from the `Dog` class) using code `super().__init__('Dog')`

instead of

```
Mammal.__init__(self, 'Dog')
```

Since we do not need to specify the name of the base class when we call its members, we can easily change the base class name (if we need to).

```
# changing base class to CanidaeFamily
class Dog(CanidaeFamily):
    def __init__(self):
        print('Dog has four legs.')

# no need to change this
super().__init__('Dog')
```

The `super()` builtin returns a proxy object, a substitute object that can call methods of the base class via delegation. This is called indirection (ability to reference base object with `super()`)

Since the indirection is computed at the runtime, we can use different base classes at different times (if we need to).

### Example 2: `super()` with Multiple Inheritance

```
class Animal:
```

```

def __init__(self, Animal):
    print(Animal, 'is an animal. ');

class Mammal(Animal):
    def __init__(self, mammalName):
        print(mammalName, 'is a warm-blooded animal. ')
        super().__init__(mammalName)

class NonWingedMammal(Mammal):
    def __init__(self, NonWingedMammal):
        print(NonWingedMammal, "can't fly.")
        super().__init__(NonWingedMammal)

class NonMarineMammal(Mammal):
    def __init__(self, NonMarineMammal):
        print(NonMarineMammal, "can't swim.")
        super().__init__(NonMarineMammal)

class Dog(NonMarineMammal, NonWingedMammal):
    def __init__(self):
        print('Dog has 4 legs. ');
        super().__init__('Dog')

d = Dog()
print("")
bat = NonMarineMammal('Bat')

```

[Run Code](#)

## Output

```

Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.

Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.

```

## Method Resolution Order (MRO)

Method Resolution Order (MRO) is the order in which methods should be inherited in the presence of multiple inheritance. You can view the MRO by using the `__mro__` attribute.

```

>>> Dog.__mro__
(<class 'Dog'>,
 <class 'NonMarineMammal'>,

```

```
<class 'NonWingedMammal'>,
<class 'Mammal'>,
<class 'Animal'>,
<class 'object'>)
```

Here is how MRO works:

- A method in the derived class is always called before the method of the base class. In our example, Dog class is called before NonMarineMammal or NonWingedMammal. These two classes are called before Mammal, which is called before Animal, and Animal class is called before the object.
- If there are multiple parents like Dog(NonMarineMammal, NonWingedMammal), methods of NonMarineMammal is invoked first because it appears first.

---

## Topic 22: In self Defense

**Possible Question: Explain In self Defense?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words mysillyobject and abc instead of self:

```
class Person:
```

```
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age
```

```
    def myfunc(abc):
        print("Hello my name is " + abc.name)
```

```
p1 = Person("John", 36)
p1.myfunc()
```

---

## Topic 23: Get and Set Attribute Values with Properties

**Possible Question: Describe about Get and Set Attribute Values with Properties?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

For the purpose of data encapsulation, most object oriented languages use getters and setters method. This is because we want to hide the attributes of a object class from other classes so that no accidental modification of the data happens by methods in other classes.

As the name suggests, getters are the methods which help access the private attributes or get the value of the private attributes and setters are the methods which help change or set the value of private attributes.

### Accessing Private Attribute

Below we write code to create a class, initialize it and access its variables without creating any additional methods.

#### Example

```
class year_graduated:
    def __init__(self, year=0):
        self._year = year

# Instantiating the class
grad_obj = year_graduated()
#Printing the object
print(grad_obj)
#Printing the object attribute
print(grad_obj.year)
```

#### Output

Running the above code gives us the following result -

```
<__main__.year_graduated object at 0x00F2DD50>
0
```

While the first print statement gives us the details of the object created, the second print object gives us the default value of the private attribute.

### Using getters and setters

In the below examples we will make a class, initialize it and then add a getter and setter method to each of them. Then access the variables in these methods by instantiating the class and using these getter and setter methods. So you can hide your logic inside the setter method.

#### Example

```
class year_graduated:
    def __init__(self, year=0):
        self._year = year

    # getter method
    def get_year(self):
        return self._year

    # setter method
    def set_year(self, a):
        self._year = a

grad_obj = year_graduated()
```

```
# Before using setter
print(grad_obj.get_year())

# After using setter
grad_obj.set_year(2019)
print(grad_obj._year)
```

#### Output

Running the above code gives us the following result:

```
0
2019
```

### Making the Attributes Private

In the next example we see how to make the methods private so that the variables in it cannot be manipulated by external calling functions. They can only be manipulated by functions inside the class. They become private by prefixing them with two underscores.

#### Example

```
class year_graduated:
    def __init__(self, year=32):
        self.__year = year

    # make the getter method
    def get_year(self):
        return self.__year

    # make the setter method
    def set_year(self, a):
        self.__year = a

grad_obj = year_graduated()
print(grad_obj.__year)

# Before using setter
print(grad_obj.get_year())
#
## After using setter
grad_obj.set_year(2019)
print(grad_obj.__year)
```

#### Output

Running the above code gives us the following result:

```
32
AttributeError: 'year_graduated' object has no attribute '__year_graduated__year'
```

## Reading Values from Private Methods

No we can access the private attribute values by using the property method and without using the getter method.

Example

```
class year_graduated:
    def __init__(self, year=32):
        self._year = year

    @property
    def Aboutyear(self):
        return self._year

    @Aboutyear.setter
    def Aboutyear(self, a):
        self._year = a

grad_obj = year_graduated()
print(grad_obj._year)

grad_obj.year = 2018
print(grad_obj.year)
```

Output

Running the above code gives us the following result:

```
32
2018
```

---

## Topic 24: Name Mangling for Privacy

**Possible Question: Describe about Name Mangling for Privacy?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Python doesn't have the concept called **private variables**. But, most of the Python developers follow a naming convention to tell that a variable is not public and it's private.

We have to start a variable name with a **double underscore** to represent it as a private variable (not really). Example:- **one, two, etc...**

As we already said the variables whose names start with a double underscore are not private. We can still access. Let's see how to create private type variables and then we will see how to access them.

```
# creating a class
class Sample:
    def __init__(self, nv, pv):
        # normal variable
        self.nv = nv
```

```
# private variable(not really)
self._pv = pv
# creating an instance of the class Sample
sample = Sample('Normal variable', 'Private variable')
```

We have created a class and its instance. We have two variables one is normal and the other is private inside the `__init__` method. Now, try to access the variables. And see what happens.

Example

```
# creating a class
class Sample:
    def __init__(self, nv, pv):
        # normal variable
        self.nv = nv
        # private variable(not really)
        self._pv = pv
# creating an instance of the class Sample
sample = Sample('Normal variable', 'Private variable')
# accessing *nv*
print(sample.nv)
# accessing *_pv**
print(sample._pv)
```

Output

If you run the above code, then you will get the following output.

Normal variable

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-13-bc324b2d20ef> in <module>
    14
    15 # accessing *_pv**
--> 16 print(sample._pv)
AttributeError: 'Sample' object has no attribute '_pv'
```

The program displayed the `nv` variable without any errors. But we got **AttributeError** when we try to access the `_pv` variable.

Why do we get this error? Because there is no any attribute with the variable name `_pv`. Then what about `self._pv = pv` statement in the `init` method? We'll discuss this in a bit. First, let's see how to access the `_pv` variable.

We have access any class variable whose name startswith a **double underscore** as `_className_variableName_`. So, in or example it is `_Sample__pv_`. Now, access it using the `_Sample__pv_` name.

Example

```
# creating a class
class Sample:
    def __init__(self, nv, pv):
        # normal variable
        self.nv = nv
        # private variable(not really)
        self._pv = pv
# creating an instance of the class Sample
sample = Sample('Normal variable', 'Private variable')
# accessing *nv*
print(sample.nv)
# accessing *_pv** using _Sample_pv name
print(sample._Sample_pv)
```

Output

If you run the above code, then you will get the following result.

Normal variable



## Private variable

Why the name of the variable `_pv` has changed?

In Python, there is a concept called name mangling. Python changes the names of the variables that start with a **double underscore**. So, any class variable whose name starts with a double underscore will change to the form `_className_variableName_`.

So, the concept will apply for the methods of the class as well. You can see it with the following code.

Example

```
class Sample:
    def __init__(self, a):
        self.a = a

    # private method(not really)
    def _get_a(self):
        return self.a

# creating an instance of the class Sample
sample = Sample(5)
# invoking the method with correct name
print(sample._Sample_get_a())
# invoking the method with wrong name
print(sample._get_a())
```

Output

If you run the above code, then you will get the following result.

```
5
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-19-55650c4990c8> in <module>
    14
    15 # invoking the method with wrong name
--> 16 print(sample._get_a())
AttributeError: 'Sample' object has no attribute '_get_a'
```

---

## Topic 25: Method Types

**Possible Question: Explain in detail about Method Types?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

There are basically three types of methods in Python:

- Instance Method
- Class Method
- Static Method

Let's talk about each method in detail.

### Instance Methods

The purpose of instance methods is to set or get details about instances (objects), and that is why they're known as instance methods. They are the most common type of methods used in a Python class.

They have one default parameter- **self**, which points to an instance of the class. *Although you don't have to pass that every time.* You can change the name of this parameter but it is better to stick to the convention i.e **self**.

Any method you create inside a class is an instance method unless you specially specify Python otherwise. Let's see how to create an instance method:

```
class My_class:
    def instance_method(self):
        return "This is an instance method."
```

It's as simple as that!

In order to call an instance method, you've to create an object/instance of the class. With the help of this object, you can access any method of the class.

```
obj = My_class()
obj.instance_method()
```

```
↳ 'This is an instance method.'
```

When the instance method is called, Python replaces the **self** argument with the instance object, **obj**. That is why we should add one default parameter while defining the instance methods. Notice that when **instance\_method()** is called, you don't have to pass self. Python does this for you.

Along with the default parameter self, you can add other parameters of your choice as well:

```
class My_class:
    def instance_method(self, a):
        return f"This is an instance method with a parameter a = {a}."
```

We have an additional parameter "a" here. Now let's create the object of the class and call this instance method:

```
obj = My_class()
obj.instance_method(10)
```

```
↳ 'This is an instance method with a parameter a = 10.'
```

Again you can see we have not passed 'self' as an argument, Python does that for us. But have to mention other arguments, in this case, it is just one. So we have passed 10 as the value of "a".

You can use "self" inside an instance method for accessing the other attributes and methods of the same class:

```
class My_class:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def instance_method(self):
        return f"This is the instance method and it can access the variables a = {self.a} and\
b = {self.b} with the help of self."
```

**Note** that the `__init__()` method is a special type of method known as a **constructor**. This method is called when an object is created from the class and it allows the class to initialize the attributes of a class.

```
obj = My_class(2,4)
obj.instance_method()
```

```
↳ 'This is the instance method and it can access the variables a = 2 and b = 4 with the help of self.'
```

Let's try this code in the live coding window below.

With the help of the "self" keyword- `self.a` and `self.b`, we have accessed the variables present in the `__init__()` method of the same class.

Along with the objects of a class, an instance method can access the class itself with the help of **self.\_\_class\_\_** attribute. Let's see how:

```
class My_class():

    def instance_method(self):
        print("Hello! from %s" % self.__class__.__name__)

obj = My_class()
obj.instance_method()
```

```
↳ Hello! from My_class
```

The `self.__class__.__name__` attribute returns the name of the class to which class instance(`self`) is related.

## 2. Class Methods

The purpose of the class methods is to set or get the details (status) of the class. That is why they are known as class methods. They can't access or modify specific instance data. They are bound to the class instead of their objects. Two important things about class methods:

- In order to define a class method, you have to specify that it is a class method with the help of the `@classmethod` decorator
- Class methods also take one default parameter- **cls**, which points to the class. Again, this not mandatory to name the default parameter "cls". But it is always better to go with the conventions

Now let's look at how to create class methods:

```
class My_class:

    @classmethod
    def class_method(cls):
        return "This is a class method."
```

As simple as that!

As I said earlier, with the help of the instance of the class, you can access any method. So we'll create the instance of this My\_class as well and try calling this class\_method():

```
obj = My_class()
obj.class_method()
```

```
↳ 'This is a class method.'
```

This works too! We can access the class methods with the help of a class instance/object. But we can access the class methods directly without creating an instance or object of the class.

Let's see how:

```
My_class.class_method()
```

```
↳ 'This is a class method.'
```

Without creating an instance of the class, you can call the class method with – **Class\_name.Method\_name()**.

But this is not possible with instance methods where we have to create an instance of the class in order to call instance methods. Let's see what happens when we try to call the instance method directly:

```
My_class.instance_method()
```

```
↳ -----
TypeError                                 Traceback (most recent call last)
<ipython-input-7-ea37e805ec52> in <module>()
----> 1 My_class.instance_method()

TypeError: instance_method() missing 1 required positional argument: 'self'
```

We got an error stating missing one positional argument – “self”. And it is obvious because instance methods accept an instance of the class as the default parameter. And you are not providing any instance as an argument. Though this can be bypassing the object name as the argument:

```
My_class.instance_method(obj)
```

```
↳ 'This is an instance method.'
```

Awesome!

### 3. Static Methods

Static methods cannot access the class data. In other words, they do not need to access the class data. They are self-sufficient and can work on their own. Since they are not attached to any class attribute, they cannot get or set the instance state or class state.

In order to define a static method, we can use the `@staticmethod` decorator (in a similar way we used `@classmethod` decorator). Unlike instance methods and class methods, we do not need to pass any special or default parameters. Let's look at the implementation:

```
class My_class:
    @staticmethod
    def static_method():
        return "This is a static method."
```

And done!

Notice that we do not have any default parameter in this case. Now how do we call static methods? Again, we can call them using object/instance of the class as:

```
obj = My_class()
obj.static_method()
```

```
↳ 'This is a static method.'
```

And we can call static methods directly, without creating an object/instance of the class:

```
My_class.static_method()
```

```
↳ 'This is a static method.'
```

You can notice the output is the same using both ways of calling static methods.

---

## Topic 26: Duck Typing

**Possible Question: Describe about Duck Typing?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

The main reason for using duck typing is to provide support for dynamic typing in Python programming. In Python, we don't need to specify the variable's data type and we can reassign the different data type values to same variable in further code. Let's see the following example.

**Example -**

1. `x = 12000`
2. `print(type(x))`
- 3.
4. `x = 'Dynamic Typing'`

5. `print(type(x))`
- 6.
7. `x = [1, 2, 3, 4]`
8. `print(type(x))`

**Output:**

```
<class 'int'>
<class 'str'>
<class 'list'>
```

As we can see in the above code, we assigned an integer to a variable `x`, making it of the `int` type. Then, we assigned a string and a list to the same variable. Python interpreter accepts the changes of data types of the same variable. This is a dynamic typing behavior.

Many other programming languages such as Java, Swift are the static type. We need to declare variable with the data types. In the below example, we try to do the same thing using the Swift instead of Python.

**Example -**

1. `# integer value assigning in JavaScript`
2. `var a = 10`
- 3.
4. `# Assigning string in swift`
5. `a = 'Swift language'`

Above code cannot be compiled, because we couldn't assign a string in Swift language. Because variable `a` was declared as an integer.

**Concept of Duck Typing**

Earlier, we have discussed that Python is a dynamic typed language. However, we can use the dynamic approach with custom data types. Let's understand the following example.

**Example -**

1. `class VisualStudio:`
2. `def execute(self):`
3. `print('Compiling')`
4. `print('Running')`
5. `print('Spell Check')`
6. `print('Convention Check')`
- 7.
8. `class Desktop:`
9. `def code(self, ide):`
10. `ide.execute()`
- 11.
- 12.
13. `ide = VisualStudio()`
14. `desk = Desktop()`
15. `desk.code(ide)`

**Output:**

```
Compiling
Running
Spell Check
Convention Check
```

In the above code, we have created a **VisualStudio** class that has to **execute()** method. In the desktop-class, we have passed the ide as an argument in the code(). An **ide** is an object of **VisualStudio** class. With the help of ide, we called the **execute()** method of VisualStudio class.

Let's see another example.

### Example - 2

```
1. class Duck:
2.     def swim(self):
3.         print("I'm a duck, and I can swim.")
4.
5. class Sparrow:
6.     def swim(self):
7.         print("I'm a sparrow, and I can swim.")
8.
9. class Crocodile:
10.    def swim_walk(self):
11.        print("I'm a Crocodile, and I can swim, but not quack.")
12.
13. def duck_testing(animal):
14.     animal.swim()
15.
16.
17. duck_testing(Duck())
18. duck_testing(Sparrow())
19. duck_testing(Crocodile())
```

### Output:

```
I'm a duck, and I can swim.
I'm a sparrow, and I can swim.
Traceback (most recent call last):
  File "<string>", line 24, in <module>
  File "<string>", line 19, in duck_testing
AttributeError: 'Crocodile' object has no attribute 'swim'
```

In the above code, the Duck class's instance is reflected by calling **the duck\_testing** function. It also happens with the Sparrow class, which implements the **swim()** function. But in the case of the Crocodile class, it fails the duck testing evaluation because it doesn't implement the **swim()** function.

### How duck typing supports EAFP

The duck typing is the most appropriate style for the EAFP because we don't need to focus on the **"type"** of the object. We only need to take care of its **behavior** and **capability**. Let's see the following statements.

When we see a lot of if-else blocks, then it is an LBYL coding style.

But if we see a lot of try-except blocks, then it is a probability an EAFP coder.

---

## Topic 27: Special Methods

**Possible Question: Describe about Special Methods?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

Python magic method is defined as the special method which adds "magic" to a class. It starts and ends with double underscores, for example, `__init__` or `__str__`.

The built-in classes define many magic methods. The `dir()` function can be used to see the number of magic methods inherited by a class. It has two prefixes, and suffix underscores in the method name.

It is most frequently used to define the overloaded behaviors of predefined operators.

#### `__init__`

The `__init__` method is called after the instance of the class has been created but before it returned to the caller. It is invoked without any call, when an instance of the class is created like constructors in other programming languages such as C++, Java, C#, PHP, etc. These methods are also known as initialize and are called after `__new__`. Its where you should initialize the instance variables.

#### `__str__`

This function computes "informal" or a nicely printable string representation of an object and must return a string object.

#### `__repr__`

This function is called by the `repr()` built-in function to compute the "official" string representation of an object and returns a machine-readable representation of a type. The goal of the `__repr__` is to be unambiguous.

#### `__len__`

This function should return the length of an object.

#### `__call__`

We can make an object callable by adding the `__call__` magic method, and it is another method that is not needed quite as often is `__call__`.

If defined in a class, then that class can be called. But if it was a function, instance itself rather than modifying.

#### `__del__`

Just as `__init__`, which is a constructor method, `__del__` is like a destructor. If you have opened a file in `__init__`, then `__del__` can close it.

#### `__bytes__`

It offers to compute a byte-string representation of an object and should return a string object.

#### `__ge__`

This method gets invoked when `>=` operator is used and returns True or False.

#### `__neg__`

This function gets called for the unary operator.

#### `__ipow__`

This function gets called on the exponents with arguments. e.g. `a**=b`.

#### `__le__`

This function gets called on comparison using `<=` operator.

#### `__nonzero__`

This function returns the Boolean value of the object. It gets invoked when the `bool` (self) function is called.

---

## Topic 28: Composition

**Possible Question: Explain in detail about Composition?**

**Possible Marks: 5 or 10 Marks**

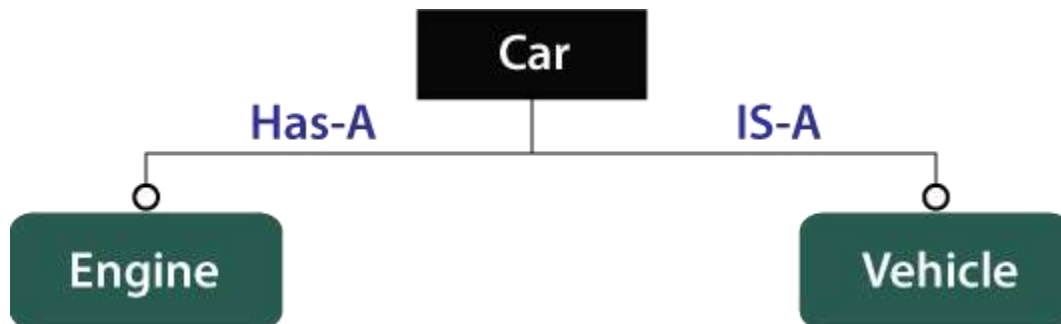
### Outcomes:

**The Composition** is a way to design or implement the "**has-a**" relationship. Composition and Inheritance both are design techniques. The Inheritance is used to implement the "**is-**



a" relationship. The "has-a" relationship is used to ensure the code reusability in our program. In Composition, we use an instance variable that refers to another object.

The composition relationship of two objects is possible when one object contains another object, and that object is fully dependent on it. The contained object should not exist without the existence of its parent object. In a simple way, we can say it is a technique through which we can describe the reference between two or more classes. And for that, we use the instance variable, which should be created before it is used.



### Key Points

- The Composition represents a part-of relationship.
- Both entities are related to each other in the Composition.
- The Composition between two entities is done when an object contains a composed object, and the composed object cannot exist without another entity. For example, if a university HAS-A college-lists, then a college is a whole, and college-lists are parts of that university.
- Favor Composition over Inheritance.
- If a university is deleted, then all corresponding colleges for that university should be deleted.

Let's take an example of a university and its colleges to understand the concept of **Composition**.

We create a class **College** that contains variables, i.e., name and address. We also create a class **University** that has a reference to refer to the list of colleges. A University can have more than one colleges. So, if a university is permanently closed, then all colleges within that particular university will be closed because colleges cannot exist without a university. The relationship between the university and colleges is Composition.

### CompositionExample.java

```
1. import java.io.*;
2. import java.util.*;
3. // class College
4. class College {
5.     public String name;
6.     public String address;
7.     College(String name, String address)
8.     {
9.         this.name = name;
10.        this.address = address;
11.    }
12. }
```

```

13. // University has more than one college.
14. class University {
15.     // reference to refer to list of college.
16.     private final List<College> colleges;
17.     University(List<College> colleges)
18.     {
19.         this.colleges = colleges;
20.     }
21.     // Getting total number of colleges
22.     public List<College> getTotalCollegesInUniversity()
23.     {
24.         return colleges;
25.     }
26. }
27. class CompositionExample {
28.     public static void main(String[] args)
29.     {
30.         // Creating the Objects of College class.
31.         College c1
32.             = new College("ABES Engineering College", "Ghaziabad");
33.         College c2
34.             = new College("AKG Engineering College", "Ghaziabad");
35.         College c3 = new College("ACN College of Engineering & Management Studies
36.             ",
37.                 "Aligarh");
38.         // Creating list which contains the no. of colleges.
39.         List<College> college = new ArrayList<College>();
40.         college.add(c1);
41.         college.add(c2);
42.         college.add(c3);
43.         University university = new University(college);
44.         List<College> colleges = university.getTotalCollegesInUniversity();
45.         for (College cg : colleges) {
46.             System.out.println("Name : " + cg.name
47.                 + " and "
48.                 + " Address : " + cg.address);
49.         }
50. }

```

**Output:**

```
C:\Windows\System32\cmd.exe
C:\Users\ajet\OneDrive\Desktop\programs>javac CompositionExample.java
C:\Users\ajet\OneDrive\Desktop\programs>java CompositionExample
Name : ABES Engineering College and Address : Ghaziabad
Name : AKG Engineering College and Address : Ghaziabad
Name : ACN College of Engineering & Management Sudies and Address : Aligarh
C:\Users\ajet\OneDrive\Desktop\programs>_
```

**Benefits of using Composition:**

- o Composition allows us to reuse the code.
- o In Java, we can use multiple Inheritance by using the composition concept.
- o The Composition provides better test-ability of a class.
- o Composition allows us to easily replace the composed class implementation with a better and improved version.
- o Composition allows us to dynamically change our program's behavior by changing the member objects at run time.

---

**Model Questions**

**Objective**

1. What will be the output of the following Python code?

```
i = 1
while True:
    if i%3 == 0:
        break
    print(i)

    i += 1
```

- a) 1 2 3                      b) error                      c) 1 2                      d) none of the mentioned

**Answer: b**

2. Which of the following functions can help us to find the version of python that we are currently working on?

- a) sys.version(1)      b) sys.version(0)      c) sys.version()      d) sys.version

**Answer: a**

3. Python supports the creation of anonymous functions at runtime, using a construct called

- a) pi                      b) anonymous c) lambda      d) none of the mentioned

**Answer: c**

4. What is the order of precedence in python?

- a) Exponential, Parentheses, Multiplication, Division, Addition, Subtraction  
b) Exponential, Parentheses, Division, Multiplication, Addition, Subtraction  
c) Parentheses, Exponential, Multiplication, Division, Subtraction, Addition  
d) Parentheses, Exponential, Multiplication, Division, Addition, Subtraction

**Answer: d**

5. What will be the output of the following Python code snippet if x=1?

```
x<<2
```

- a) 4                      b) 2                      c) 1                      d) 8

**Answer: a**

6. What does pip stand for python?

- a) unlimited length
- b) all private members must have leading and trailing underscores
- c) Preferred Installer Program
- d) none of the mentioned

**Answer: c**

Explanation: Variable names can be of any length.

7. Which of the following is true for variable names in Python?

- a) underscore and ampersand are the only two special characters allowed
- b) unlimited length
- c) all private members must have leading and trailing underscores
- d) none of the mentioned

**Answer: b**

8. What are the values of the following Python expressions?

`2**(3**2)`

`(2**3)**2`

`2**3**2`

- a) 512, 64, 512
- b) 512, 512, 512
- c) 64, 512, 64
- d) 64, 64, 64

**Answer: a**

9. Which of the following is the truncation division operator in Python?

- a) |
- b) //
- c) /
- d) %

**Answer: b**

10. What will be the output of the following Python code?

```
l=[1, 0, 2, 0, 'hello', ", []]
```

```
list(filter(bool, l))
```

- a) [1, 0, 2, 'hello', ", []]
- b) Error
- c) [1, 2, 'hello']
- d) [1, 0, 2, 0, 'hello', ", []]

**Answer: c**

### Subjective

1. Explain Code Structures in details about if, elif, and else Statement?
2. Explain about Repeat with while & iterate with for?
3. Discuss about Functions of Generators ?
4. Describe about Modules, Packages, and Programs?
5. Explain about Command-Line Arguments?
6. Explain about Objects and Classes?
7. Define a Class with class & Inheritance in detailed?
8. Explain detail about Override a Method & Add a Method?
9. Explain in detail about Get and Set Attribute Values with Properties?
10. Discuss about Method Types , Duck Typing & Special Methods

---

**Unit – II – END**

**Signature of the staff  
with date**



# Sri Ganesh College of Arts & Science – Salem- 14.

Department of Computer Science & Applications

Study Material – 2022(Odd Semester)

Sub: Open source computing

Paper Code: 21PCS

Class: II M.Sc CS

Staff I/c: K. Aravindhan M.Sc.,

Date:

Head:

---

## UNIT – III

Data Types: Text Strings – Binary Data. Storing and Retrieving Data: File Input/Output – Structured Text Files – Structured Binary Files - Relational Databases – No SQL Data Stores

---

### Topic 1: Data Types

**Possible Question: Discuss about Data Types in text strings?**

**Possible Marks: 5 or 10 Marks**

#### Outcomes:

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

Example

```
print("Hello")
```

```
print('Hello')
```

#### Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
```

```
print(a)
```

#### Multiline Strings

You can assign a multiline string to a variable by using three quotes:

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
```

```
consectetur adipiscing elit,
```

```
sed do eiusmod tempor incididunt
```

```
ut labore et dolore magna aliqua."""
```

```
print(a)
```

Or three single quotes:

Example

```
a = "Lorem ipsum dolor sit amet,
```

```
consectetur a dipiscing elit,
```

```
sed do eiusmod tempor incididunt
```

```
ut labore et dolore magna aliqua."
```

```
print(a)
```

#### Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"  
print(a[1])
```

### **Looping Through a String**

Since strings are arrays, we can loop through the characters in a string, with a for loop.

Example

Loop through the letters in the word "banana":

```
for x in "banana":  
    print(x)
```

Learn more about For Loops in our [Python For Loops](#)

### **String Length**

To get the length of a string, use the len() function.

Example

The len() function returns the length of a string:

```
a = "Hello, World!"  
print(len(a))
```

### **Check String**

To check if a certain phrase or character is present in a string, we can use the keyword in.

Example

Check if "free" is present in the following text:

```
txt = "The best things in life are free!"  
print("free" in txt)
```

Use it in an if statement:

Example

Print only if "free" is present:

```
txt = "The best things in life are free!"  
if "free" in txt:  
    print("Yes, 'free' is present.")
```

### **Check if NOT**

To check if a certain phrase or character is NOT present in a string, we can use the keyword not in.

Example

Check if "expensive" is NOT present in the following text:

```
txt = "The best things in life are free!"  
print("expensive" not in txt)
```

Use it in an if statement:

Example

print only if "expensive" is NOT present:

```
txt = "The best things in life are free!"  
if "expensive" not in txt:  
    print("No, 'expensive' is NOT present.")
```

---

## Topic 2: Binary Data

**Possible Question: Discuss about Binary Data?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

The **bytes** type in Python is immutable and stores a sequence of values ranging from 0-255 (8-bits). You can get the value of a single byte by using an index like an array, but the values can not be modified.

```
# Create empty bytes
empty_bytes = bytes(4)
print(type(empty_bytes))
print(empty_bytes)
```

### The Bytearray Type

To create a mutable object you need to use the bytearray type. With a bytearray you can do everything you can with other mutables like push, pop, insert, append, delete, and sort.

```
# Cast bytes to bytearray
mutable_bytes = bytearray(b'\x00\x0F')
```

```
# Bytearray allows modification
mutable_bytes[0] = 255
mutable_bytes.append(255)
print(mutable_bytes)
```

```
# Cast bytearray back to bytes
immutable_bytes = bytes(mutable_bytes)
print(immutable_bytes)
```

### The BytesIO Class

The [io.BytesIO](#) inherits from io.BufferedReader class comes with functions like read(), write(), peek(), getvalue(). It is a general buffer of bytes that you can work with.

```
binary_stream = io.BytesIO()
# Binary data and strings are different types, so a str
# must be encoded to binary using ascii, utf-8, or other.
binary_stream.write("Hello, world!\n".encode('ascii'))
binary_stream.write("Hello, world!\n".encode('utf-8'))
```

```
# Move cursor back to the beginning of the buffer
binary_stream.seek(0)
```

```
# Read all data from the buffer
stream_data = binary_stream.read()
```

```
# The stream_data is type 'bytes', immutable
print(type(stream_data))
print(stream_data)
```

```
# To modify the actual contents of the existing buffer
# use getbuffer() to get an object you can modify.
# Modifying this object updates the underlying BytesIO buffer
mutable_buffer = binary_stream.getbuffer()
```

```
print(typemutable_buffer)) # class 'memoryview'  
mutable_buffer[0] = 0xFF
```

```
# Re-read the original stream. Contents will be modified  
# because we modified the mutable buffer  
binary_stream.seek(0)  
print(binary_stream.read())
```

### Writing Bytes to a File

```
# Pass "wb" to write a new file, or "ab" to append  
with open("test.txt", "wb") as binary_file:  
    # Write text or bytes to the file  
    binary_file.write("Write text by encoding\n".encode('utf8'))  
    num_bytes_written = binary_file.write(b'\xDE\xAD\xBE\xEF')  
    print("Wrote %d bytes." % num_bytes_written)
```

Alternatively, you could explicitly call open and close, but if you do it this way you will need to do the error handling yourself and ensure the file is always closed, even if there is an error during writing. I don't recommend this method unless you have a strong reason.

```
binary_file = open("test.txt", "wb")  
binary_file.write(b'\x00')  
binary_file.close()
```

### Reading Bytes From a File

```
with open("test_file.dat", "rb") as binary_file:  
    # Read the whole file at once  
    data = binary_file.read()  
    print(data)
```

### Read file line by line

If you are working a text file, you can read the data in line by line.

```
with open("test.txt", "rb") as text_file:  
    # One option is to call readline() explicitly  
    # single_line = text_file.readline()  
  
    # It is easier to use a for loop to iterate each line  
    for line in text_file:  
        print(line)
```

### Getting the size of a file

```
import os  
file_length_in_bytes = os.path.getsize("test.txt")  
print(file_length_in_bytes)
```

### Seeking a specific position in a file

You can move to a specific position in file before reading or writing using seek(). You can pass a single parameter to seek() and it will move to that position, relative to the beginning of the file.

```
# Seek can be called one of two ways:  
# x.seek(offset)
```



```

# x.seek(offset, starting_point)

# starting_point can be 0, 1, or 2
# 0 - Default. Offset relative to beginning of file
# 1 - Start from the current position in the file
# 2 - Start from the end of a file (will require a negative offset)

with open("test_file.dat", "rb") as binary_file:
    # Seek a specific position in the file and read N bytes
    binary_file.seek(0, 0) # Go to beginning of the file
    couple_bytes = binary_file.read(2)
    print(couple_bytes)

```

## Integer to Bytes

```

i = 16

# Create one byte from the integer 16
single_byte = i.to_bytes(1, byteorder='big', signed=True)
print(single_byte)

# Create four bytes from the integer
four_bytes = i.to_bytes(4, byteorder='big', signed=True)
print(four_bytes)

# Compare the difference to little endian
print(i.to_bytes(4, byteorder='little', signed=True))

# Create bytes from a list of integers with values from 0-255
bytes_from_list = bytes([255, 254, 253, 252])
print(bytes_from_list)

# Create a byte from a base 2 integer
one_byte = int('11110000', 2)
print(one_byte)

# Print out binary string (e.g. 0b010010)
print(bin(22))

```

## Bytes to Integer

```

# Create an int from bytes. Default is unsigned.
some_bytes = b'\x00\xF0'
i = int.from_bytes(some_bytes, byteorder='big')
print(i)

# Create a signed int
i = int.from_bytes(b'\x00\x0F', byteorder='big', signed=True)
print(i)

# Use a list of integers 0-255 as a source of byte values
i = int.from_bytes([255, 0, 0, 0], byteorder='big')
print(i)

```

## Text Encoding

```
# Binary to Text
binary_data = b'I am text.'
text = binary_data.decode('utf-8')
print(text)

binary_data = bytes([65, 66, 67]) # ASCII values for A, B, C
text = binary_data.decode('utf-8')
print(text)

# Text to Binary
message = "Hello" # str
binary_message = message.encode('utf-8')
print(type(binary_message)) # bytes

# Python has many built in encodings for different languages,
# and even the Caesar cipher is built in
import codecs
cipher_text = codecs.encode(message, 'rot_13')
print(cipher_text)
```

## Base 64 Encoding

```
# Encode binary data to a base 64 string
binary_data = b'\x00\xff\x00\xff'

# Use the codecs module to encode
import codecs
base64_data = codecs.encode(binary_data, 'base64')
print(base64_data)

# Or use the binascii module
import binascii
base64_data = binascii.b2a_base64(binary_data)
print(base64_data)

# The base64_string is still a bytes type
# It may need to be decoded to an ASCII string
print(base64_data.decode('utf-8'))

# Decoding is done similarly
print(codecs.decode(base64_data, 'base64'))
print(binascii.a2b_base64(base64_data))
```

## Hexadecimal

```
# Starting with a hex string you can unhexlify it to bytes
deadbeef = binascii.unhexlify('DEADBEEF')
print(deadbeef)

# Given raw bytes, get an ASCII string representing the hex values
hex_data = binascii.hexlify(b'\x00\xff') # Two bytes values 0 and 255

# The resulting value will be an ASCII string but it will be a bytes type
```

```
# It may be necessary to decode it to a regular string
text_string = hex_data.decode('utf-8') # Result is string "00ff"
print(text_string)
```

## Format Strings

Format strings can be helpful to visualize or output byte values. Format strings require an integer value so the byte will have to be converted to an integer first.

```
a_byte = b'\xff' # 255
i = ord(a_byte) # Get the integer value of the byte
```

```
bin = "{0:b}".format(i) # binary: 11111111
hex = "{0:x}".format(i) # hexadecimal: ff
oct = "{0:o}".format(i) # octal: 377
```

```
print(bin)
print(hex)
print(oct)
```

## Bitwise Operations

```
# Some bytes to play with
byte1 = int('11110000', 2) # 240
byte2 = int('00001111', 2) # 15
byte3 = int('01010101', 2) # 85
```

```
# Ones Complement (Flip the bits)
print(~byte1)
```

```
# AND
print(byte1 & byte2)
```

```
# OR
print(byte1 | byte2)
```

```
# XOR
print(byte1 ^ byte3)
```

```
# Shifting right will lose the right-most bit
print(byte2 >> 3)
```

```
# Shifting left will add a 0 bit on the right side
print(byte2 << 1)
```

```
# See if a single bit is set
bit_mask = int('00000001', 2) # Bit 1
print(bit_mask & byte1) # Is bit set in byte1?
print(bit_mask & byte2) # Is bit set in byte2?
```

## Struct Packing and Unpacking

Packing and unpacking requires a string that defines how the binary data is structured. It needs to know which bytes represent values. It needs to know whether the entire set of bytes represents characters or if it is a sequence of 4-byte integers. It can be structured in any

number of ways. The format strings can be simple or complex. In this example I am packing a single four-byte integer followed by two characters. The letters i and c represent integers and characters.

### **import struct**

```
# Packing values to bytes
# The first parameter is the format string. Here it specifies the data is structured
# with a single four-byte integer followed by two characters.
# The rest of the parameters are the values for each item in order
binary_data = struct.pack("icc", 8499000, b'A', b'Z')
print(binary_data)

# When unpacking, you receive a tuple of all data in the same order
tuple_of_data = struct.unpack("icc", binary_data)
print(tuple_of_data)

# For more information on format strings and endiannes, refer to
# https://docs.python.org/3.5/library/struct.html
```

### **System Byte Order**

You might need to know what byte order your system uses. Byte order refers to big endian or little endian. The sys module can provide that value.

```
# Find out what byte order your system uses
import sys
print("Native byteorder: ", sys.byteorder)
```

### **Examples**

```
# diff.py - Do two files match?
# Exercise: Rewrite this code to compare the files part at a time so it
# will not run out of RAM with large files.
```

```
import sys
```

```
with open(sys.argv[1], 'rb') as file1, open(sys.argv[2], 'rb') as file2:
```

```
    data1 = file1.read()
```

```
    data2 = file2.read()
```

```
if data1 != data2:
```

```
    print("Files do not match.")
```

```
else:
```

```
    print("Files match.")
```

```
#is_jpeg.py - Does the file have a JPEG binary signature?
```

```
import sys
```

```
import binascii
```

```
jpeg_signatures = [
```

```
    binascii.unhexlify(b'FFD8FFD8'),
```

```
    binascii.unhexlify(b'FFD8FFE0'),
```

```
    binascii.unhexlify(b'FFD8FFE1')
```

```
]
```

```

with open(sys.argv[1], 'rb') as file:
    first_four_bytes = file.read(4)

    if first_four_bytes in jpeg_signatures:
        print("JPEG detected.")
    else:
        print("File does not look like a JPEG.")

```

---

### Topic 3: File Input/Output

**Possible Question: Explain in detail about File Input/Output?**

**Possible Marks: 5 or 10 Marks**

#### Outcomes:

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

#### Opening a file

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

#### Syntax:

```
file object = open(<file-name>, <access-mode>, <buffering>)
```

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

#### Example

1. #opens the file file.txt in read mode
2. fileptr = open("file.txt", "r")
- 3.
4. **if** fileptr:
5.     **print**("file is opened successfully")

#### Output:

```
<class '_io.TextIOWrapper'>
```

SN	Access mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

file is opened successfully

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement

### The close() method

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

## Syntax

1. fileobject.close()

Consider the following example.

1. # opens the file file.txt in read mode
2. fileptr = open("file.txt", "r")
- 3.
4. **if** fileptr:
5.     **print**("file is opened successfully")
- 6.
7. #closes the opened file
8. fileptr.close()

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

1. **try**:
2.     fileptr = open("file.txt")
3.     # perform file operations
4. **finally**:
5.     fileptr.close()

## The with statement

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.

1. with open(<file name>, <access mode>) as <file-pointer>:
2.     #statement suite

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Consider the following example.

*Example*

1. with open("file.txt", 'r') as f:
2.     content = f.read();
3.     **print**(content)

## Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

**w**: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

**a**: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

*Example*

1. # open the file.txt in append mode. Create a new file if no such file exists.
2. fileptr = open("file2.txt", "w")

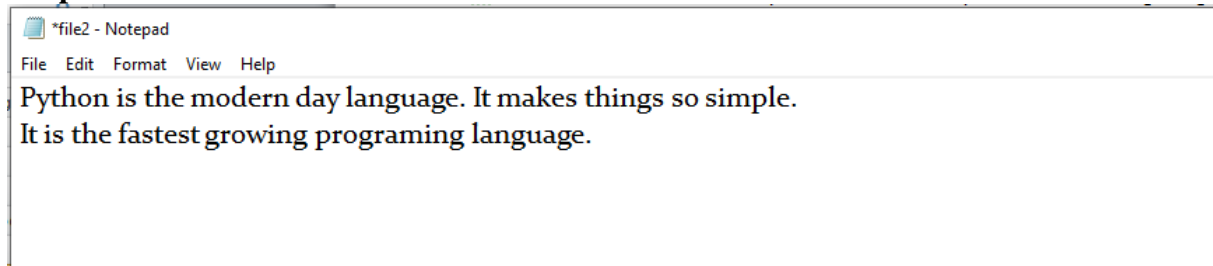
- 3.
4. # appending the content to the file
5. fileptr.write("""Python is the modern day language. It makes things so simple.
6. It is the fastest-growing programming language""")
- 7.
8. # closing the opened the file
9. fileptr.close()

**Output:**

File2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

**Snapshot of the file2.txt**



We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

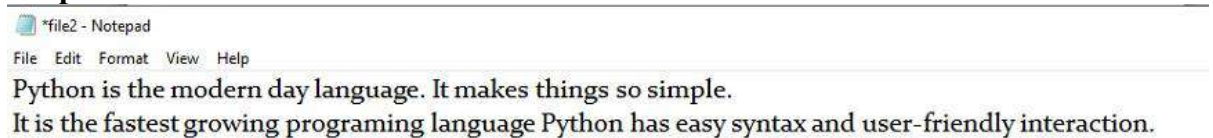
*Example 2*

1. #open the file.txt in write mode.
2. fileptr = open("file2.txt","a")
- 3.
4. #overwriting the content of the file
5. fileptr.write(" Python has an easy syntax and user-friendly interaction.")
- 6.
7. #closing the opened file
8. fileptr.close()

**Output:**

Python is the modern day language. It makes things so simple.  
It is the fastest growing programming language Python has an easy syntax and user-friendly interaction.

**Snapshot of the file2.txt**



We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

**Syntax:**



1. `fileobj.read(<count>)`

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

*Example*

1. `#open the file.txt in read mode. causes error if no such file exists.`
2. `fileptr = open("file2.txt", "r")`
3. `#stores all the data of the file into the variable content`
4. `content = fileptr.read(10)`
5. `# prints the type of the data stored in the file`
6. `print(type(content))`
7. `#prints the content of the file`
8. `print(content)`
9. `#closes the opened file`
10. `fileptr.close()`

**Output:**

`<class 'str'>`

Python is

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

1. `content = fileptr.read()`
2. `print(content)`

**Output:**

Python is the modern-day language. It makes things so simple.

It is the fastest-growing programming language Python has easy an syntax and user-friendly interaction.

**Read file through for loop**

We can read the file using for loop. Consider the following example.

1. `#open the file.txt in read mode. causes an error if no such file exists.`
2. `fileptr = open("file2.txt", "r");`
3. `#running a for loop`
4. `for i in fileptr:`
5. `print(i) # i contains each line of the file`

**Output:**

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the **readline()** method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

*Example 1: Reading lines using readline() function*

1. `#open the file.txt in read mode. causes error if no such file exists.`
2. `fileptr = open("file2.txt", "r");`
3. `#stores all the data of the file into the variable content`

4. `content = fileptr.readline()`
5. `content1 = fileptr.readline()`
6. `#prints the content of the file`
7. `print(content)`
8. `print(content1)`
9. `#closes the opened file`
10. `fileptr.close()`

**Output:**

Python is the modern day language.

It makes things so simple.

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

*Example 2: Reading Lines Using readlines() function*

1. `#open the file.txt in read mode. causes error if no such file exists.`
2. `fileptr = open("file2.txt", "r");`
- 3.
4. `#stores all the data of the file into the variable content`
5. `content = fileptr.readlines()`
- 6.
7. `#prints the content of the file`
8. `print(content)`
- 9.
10. `#closes the opened file`
11. `fileptr.close()`

**Output:**

['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']

**Creating a new file**

The new file can be created by using one of the following access modes with the function `open()`.

**x:** it creates a new file with the specified name. It causes an error a file exists with the same name.

**a:** It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

**w:** It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

*Example 1*

1. `#open the file.txt in read mode. causes error if no such file exists.`
2. `fileptr = open("file2.txt", "x")`
3. `print(fileptr)`
4. `if fileptr:`
5. `print("File created successfully")`

**Output:**

<\_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>

File created successfully

### File Pointer positions

Python provides the `tell()` method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

1. `# open the file file2.txt in read mode`
2. `fileptr = open("file2.txt","r")`
- 3.
4. `#initially the filepointer is at 0`
5. `print("The filepointer is at byte :",fileptr.tell())`
- 6.
7. `#reading the content of the file`
8. `content = fileptr.read();`
- 9.
10. `#after the read operation file pointer modifies. tell() returns the location of the fileptr.`
- 11.
12. `print("After reading, the filepointer is at:",fileptr.tell())`

### Output:

The filepointer is at byte : 0

After reading, the filepointer is at: 117

### Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the `seek()` method which enables us to modify the file pointer position externally.

The syntax to use the `seek()` method is given below.

### Syntax:

```
<file-ptr>.seek(offset[, from])
```

The `seek()` method accepts two parameters:

**offset:** It refers to the new position of the file pointer within the file.

**from:** It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

#### *Example*

1. `# open the file file2.txt in read mode`
2. `fileptr = open("file2.txt","r")`
- 3.
4. `#initially the filepointer is at 0`
5. `print("The filepointer is at byte :",fileptr.tell())`
- 6.
7. `#changing the file pointer location to 10.`
8. `fileptr.seek(10);`
- 9.

10. `#tell()` returns the location of the fileptr.

11. `print("After reading, the filepointer is at:",fileptr.tell())`

### **Output:**

The filepointer is at byte : 0

After reading, the filepointer is at: 10

Python OS module

### **Renaming the file**

The Python **os** module enables interaction with the operating system. The **os** module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the `rename()` method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

### **Syntax:**

1. `rename(current-name, new-name)`

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

### **Example 1:**

1. `import os`

2.

3. `#rename file2.txt to file3.txt`

4. `os.rename("file2.txt","file3.txt")`

### **Output:**

The above code renamed current **file2.txt** to **file3.txt**

### **Removing the file**

The **os** module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

1. `remove(file-name)`

### **Example 1**

1. `import os;`

2. `#deleting the file named file3.txt`

3. `os.remove("file3.txt")`

### **Creating the new directory**

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

### **Syntax:**

1. `mkdir(directory name)`

### **Example 1**

1. `import os`

2.

3. `#creating a new directory with the name new`

4. `os.mkdir("new")`

The `getcwd()` method

This method returns the current working directory.

The syntax to use the `getcwd()` method is given below.

### **Syntax**

1. `os.getcwd()`

### **Example**

1. `import os`

2. `os.getcwd()`

**Output:**

```
'C:\\Users\\DEVANSH SHARMA'
```

**Changing the current working directory**

The `chdir()` method is used to change the current working directory to a specified directory. The syntax to use the `chdir()` method is given below.

**Syntax**

1. `chdir("new-directory")`

*Example*

1. **import** os
2. # Changing current directory with the new directory
3. `os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")`
4. #It will display the current working directory
5. `os.getcwd()`

**Output:**

```
'C:\\Users\\DEVANSH SHARMA\\Documents'
```

**Deleting directory**

The `rmdir()` method is used to delete the specified directory. The syntax to use the `rmdir()` method is given below.

**Syntax**

1. `os.rmdir(directory name)`

**Example 1**

1. **import** os
2. #removing the new directory
3. `os.rmdir("directory_name")`

It will remove the specified directory.

**Writing Python output to the files**

In Python, there are the requirements to write the output of a Python script to a file.

The `check_call()` method of module **subprocess** is used to execute a Python script and write the output of that script to a file.

The following example contains two python scripts. The script `file1.py` executes the script `file.py` and writes its output to the text file **output.txt**.

**Example****file.py**

SN	Method	Description
1	<code>file.close()</code>	It closes the opened file. The file once closed, it can't be read or write anymore.
2	<code>File.fush()</code>	It flushes the internal buffer.
3	<code>File.fileno()</code>	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
4	<code>File.isatty()</code>	It returns true if the file is connected to a TTY device, otherwise returns false.
5	<code>File.next()</code>	It returns the next line from the file.
6	<code>File.read([size])</code>	It reads the file for the specified size.

7	File.readline([size])	It reads one line from the file and places the file pointer to the beginning of the new line.
8	File.readlines([sizehint])	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
9	File.seek(offset[,from])	It modifies the position of the file pointer to a specified offset with the specified reference.
10	File.tell()	It returns the current position of the file pointer within the file.
11	File.truncate([size])	It truncates the file to the optional specified size.
12	File.write(str)	It writes the specified string to a file
13	File.writelines(seq)	It writes a sequence of the strings to a file.

```

1. temperatures=[10,-20,-289,100]
2. def c_to_f(c):
3.     if c< -273.15:
4.         return "That temperature doesn't make sense!"
5.     else:
6.         f=c*9/5+32
7.         return f
8. for t in temperatures:
9.     print(c_to_f(t))

```

#### file.py

```

1. import subprocess
2.
3. with open("output.txt", "wb") as f:
4.     subprocess.check_call(["python", "file.py"], stdout=f)

```

The file related methods

The file object provides the following methods to manipulate the files on various operating systems.

### Topic 4: Structured Text Files

**Possible Question: Describe about Structured Text Files?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

A text file is structured as a sequence of lines.

- Each line of the text file consists of a sequence of characters.
- A separator, or delimiter, character like tab ('\t'), comma (','), or vertical bar (|). This is an example of the comma-separated values (CSV) format.
- '<' and '>' around tags. Examples include XML and HTML.
- Punctuation. An example is JavaScript Object Notation (JSON).

## Steps for writing to text files

To write to a text file in Python, you follow these steps:

- First, open the text file for writing (or appending) using the `open()` function.
- Second, write to the text file using the `write()` or `writelines()` method.
- Third, close the file using the `close()` method.

The following shows the basic syntax of the `open()` function:

```
f = open(path_to_file, mode)
```

The `open()` function accepts many parameters. But you'll focus on the first two:

- The `path_to_file` parameter specifies the path to the text file that you want to open for writing.
- The `mode` parameter specifies the mode for which you want to open the text file.

For writing to a text file, you use one of the following modes:

Mode	Description
'w'	Open a text file for writing text
'a'	Open a text file for appending text

The `open()` function returns a file object. And the file object has two useful methods for writing text to the file: `write()` and `writelines()`.

The `write()` method writes a string to a text file and the `writelines()` method write a list of strings to a file at once.

In fact, the `writelines()` method accepts an [iterable](#) object, not just a [list](#), so you can pass a [tuple](#) of strings, a [set](#) of strings, etc., to the `writelines()` method.

To write a line to a text file, you need to manually add a new line character:

```
f.write('\n')  
f.writelines('\n')
```

Code language: JavaScript (javascript)

And it's up to you to add the new line characters.

## Writing text file examples

The following example shows how to use the `write()` function to write a list of texts to a text file:

```
lines = ['Readme', 'How to write text files in Python']  
with open('readme.txt', 'w') as f:  
    for line in lines:  
        f.write(line)  
        f.write('\n')
```

Code language: JavaScript (javascript)

If the readme.txt file doesn't exist, the open() function will create a new file.

```
files > ⓘ readme.txt
1  README
2  How to write text files in Python
3  |
```

The following shows how to write a list of text strings to a text file:

```
lines = ['Readme', 'How to write text files in Python']
with open('readme.txt', 'w') as f:
    f.writelines(lines)
```

Code language: JavaScript (javascript)

If you treat each element of the list as a line, you need to concatenate it with the newline character like this:

```
lines = ['Readme', 'How to write text files in Python']
with open('readme.txt', 'w') as f:
    f.write('\n'.join(lines))
```

Code language: JavaScript (javascript)

```
files > ⓘ readme.txt
1  README
2  How to write text files in Python
3  |
```

### Appending text files

To append to a text file, you need to open the text file for appending mode. The following example appends new lines to the readme.txt file:

```
more_lines = ['Append text files', 'The End']
with open('readme.txt', 'a') as f:
    f.writelines('\n'.join(more_lines))
```

Code language: JavaScript (javascript)

Output:

```
files > ⓘ readme.txt
1  README
2  How to write text files in Python
3  Append text files
4  The End
```

### Writing to a UTF-8 text file

If you write UTF-8 characters to a text file using the code from the previous examples, you'll get an error like this:



UnicodeEncodeError: 'charmap' codec can't encode characters in position 0-44: character maps to <undefined>

Code language: HTML, XML (xml)

To open a file and write UTF-8 characters to a file, you need to pass the encoding='utf-8' parameter to the open() function.

---

## Topic 5: Structured Binary Files

**Possible Question: Describe about Structured Binary Files?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

The file that contains the binary data is called a binary file. Any formatted or unformatted binary data is stored in a binary file, and this file is not human-readable and is used by the computer directly. When a binary file is required to read or transfer from one location to another location, the file's content is converted or encoded into a human-readable format. The extension of the binary file is .bin. The content of the binary file can be read by using a built-in function or module. Different ways to read binary files in Python have been shown in this tutorial.

### Pre-requisite:

Before checking the examples of this tutorial, it is better to create one or more binary files to use in the example script. The script of two python files has given below to create two binary files. The binary1.py will create a binary file named **string.bin** that will contain string data, and the binary2.py will create a binary file named **number\_list.bin** that will contain a list of numeric data.

### Binary1.py

```
# Open a file handler to create a binary file

file_handler = open("string.bin", "wb")

# Add two lines of text in the binary file

file_handler.write(b"Welcome to LinuxHint.\nLearn Python Programming.")

# Close the file handler

file_handler.close()
```

### Binary2.py

```
# Open a file handler to create a binary file

file=open("number_list.bin","wb")

# Declare a list of numeric values

numbers=[10,30,45,60,70,85,99]

# Convert the list to array

barray=bytearray(numbers)

# Write array into the file

file.write(barray)
```

```
file.close()
```

### Example-1: Read the binary file of string data into the byte array

Many ways exist in Python to read the binary file. You can read the particular number of bytes or the full content of the binary file at a time. Create a python file with the following script. The **open()** function has used to open the **string.bin** for reading. The **read()** function has been used to read 7 characters from the file in each iteration of while loop and print. Next, the **read()** function has been used without any argument to read the full content of the binary file that will be printed later.

```
# Open the binary file for reading

file_handler = open("string.bin", "rb")

# Read the first three bytes from the binary file

data_byte = file_handler.read(7)

print("Print three characters in each iteration:")

# Iterate the loop to read the remaining part of the file

while data_byte:

    print(data_byte)

    data_byte = file_handler.read(7)

# Read the entire file as a single byte string

with open('string.bin', 'rb') as fh:

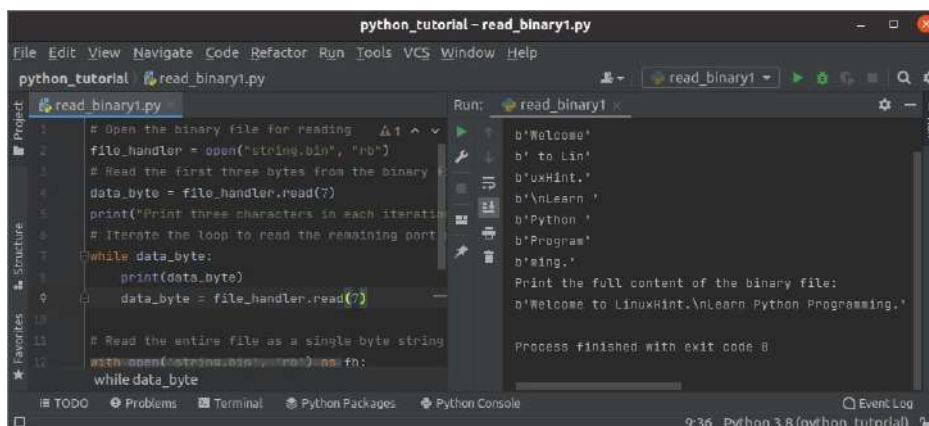
    content = fh.read()

print("Print the full content of the binary file:")

print(content)
```

### Output:

The following output will appear after executing the above script.



```
python_tutorial - read_binary1.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
python_tutorial read_binary1.py read_binary1
read_binary1.py
1 # Open the binary file for reading
2 file_handler = open("string.bin", "rb")
3 # Read the first three bytes from the binary
4 data_byte = file_handler.read(7)
5 print("Print three characters in each iterati
6 # Iterate the loop to read the remaining part
7 while data_byte:
8     print(data_byte)
9     data_byte = file_handler.read(7)
10
11 # Read the entire file as a single byte string
12 with open('string.bin', 'rb') as fh:
13     while data_byte:
14         print(data_byte)
15         data_byte = fh.read(7)
16
17 content = fh.read()
18
19 print("Print the full content of the binary file:")
20 print(content)
Run: read_binary1 x
b'Welcome'
b' to Lin'
b'uxHint.'
b'\nLearn '
b'Python '
b'Program'
b'ing.'
Print the full content of the binary file:
b'Welcome to LinuxHint.\nLearn Python Programming.'
Process finished with exit code 0
9:36 Python 3.8 (python_tutorial)
```

## Topic 6: Relational Databases

**Possible Question: Explain detail about Relational Databases?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

We can connect to relational databases for analysing data using the **pandas** library as well as another additional library for implementing database connectivity. This package is named as **sqlalchemy** which provides full SQL language functionality to be used in python.

### Installing SQLAlchemy

The installation is very straight forward using Anaconda [Data Science Environment](#)., run the following command in the Anaconda Prompt Window to install the SQLAlchemy package.

```
conda install sqlalchemy
```

### Reading Relational Tables

We will use SQLite3 as our relational database as it is very light weight and easy to use. Though the SQLAlchemy library can connect to a variety of relational sources including MySQL, Oracle and Postgresql and Mssql. We first create a database engine and then connect to the database engine using the **to\_sql** function of the SQLAlchemy library.

In the below example we create the relational table by using the **to\_sql** function from a dataframe already created by reading a csv file. Then we use the **read\_sql\_query** function from pandas to execute and capture the results from various SQL queries.

```
from sqlalchemy import create_engine
import pandas as pd

data = pd.read_csv('/path/input.csv')

# Create the db engine
engine = create_engine('sqlite:///memory:')

# Store the dataframe as a table
data.to_sql('data_table', engine)

# Query 1 on the relational table
res1 = pd.read_sql_query('SELECT * FROM data_table', engine)
print('Result 1')
print(res1)
print("")

# Query 2 on the relational table
res2 = pd.read_sql_query('SELECT dept,sum(salary) FROM data_table group by dept',
engine)
print('Result 2')
print(res2)
```

When we execute the above code, it produces the following result.

```
Result 1
  index id  name salary start_date  dept
0     0  1  Rick  623.30 2012-01-01    IT
1     1  2   Dan  515.20 2013-09-23  Operations
2     2  3  Tusar  611.00 2014-11-15    IT
3     3  4   Ryan  729.00 2014-05-11    HR
4     4  5   Gary  843.25 2015-03-27  Finance
5     5  6  Rasmi  578.00 2013-05-21    IT
6     6  7  Pranab  632.80 2013-07-30  Operations
7     7  8   Guru  722.50 2014-06-17  Finance

Result 2
  dept sum(salary)
```

```

0 Finance 1565.75
1 HR 729.00
2 IT 1812.30
3 Operations 1148.00

```

### Inserting Data to Relational Tables

We can also insert data into relational tables using `sql.execute` function available in pandas. In the below code we previous csv file as input data set, store it in a relational table and then insert another record using `sql.execute`.

```

from sqlalchemy import create_engine
from pandas.io import sql

import pandas as pd

data = pd.read_csv('C:/Users/Rasmi/Documents/pydatasci/input.csv')
engine = create_engine('sqlite:///memory:')

# Store the Data in a relational table
data.to_sql('data_table', engine)

# Insert another row
sql.execute('INSERT INTO data_table VALUES(?,?,?,?,' + engine,
params=[('id',9,'Ruby',711.20,'2015-03-27','IT')])

# Read from the relational table
res = pd.read_sql_query('SELECT ID,Dept,Name,Salary,start_date FROM data_table',
engine)
print(res)

```

When we execute the above code, it produces the following result.

```

id  dept  name  salary  start_date
0  1    IT   Rick  623.30  2012-01-01
1  2  Operations  Dan  515.20  2013-09-23
2  3    IT  Tusar  611.00  2014-11-15
3  4    HR   Ryan  729.00  2014-05-11
4  5  Finance  Gary  843.25  2015-03-27
5  6    IT  Rasmi  578.00  2013-05-21
6  7  Operations  Pranab  632.80  2013-07-30
7  8  Finance  Guru  722.50  2014-06-17
8  9    IT   Ruby  711.20  2015-03-27

```

### Deleting Data from Relational Tables

We can also delete data into relational tables using `sql.execute` function available in pandas. The below code deletes a row based on the input condition given.

```

from sqlalchemy import create_engine
from pandas.io import sql

import pandas as pd

data = pd.read_csv('C:/Users/Rasmi/Documents/pydatasci/input.csv')
engine = create_engine('sqlite:///memory:')
data.to_sql('data_table', engine)

sql.execute('Delete from data_table where name = (?) ', engine, params=[('Gary')])

res = pd.read_sql_query('SELECT ID,Dept,Name,Salary,start_date FROM data_table',
engine)

```

```
print(res)
```

When we execute the above code, it produces the following result.

```
id  dept  name salary start_date
0  1    IT   Rick 623.3 2012-01-01
1  2  Operations  Dan 515.2 2013-09-23
2  3    IT  Tusar 611.0 2014-11-15
3  4    HR   Ryan 729.0 2014-05-11
4  6    IT  Rasmi 578.0 2013-05-21
5  7  Operations  Pranab 632.8 2013-07-30
6  8  Finance  Guru 722.5 2014-06-17
```

---

## Topic 7: No SQL Data Stores

**Possible Question: Explain detail about No SQL Data Stores?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

As more and more data become available as unstructured or semi-structured, the need of managing them through NoSql database increases. Python can also interact with NoSQL databases in a similar way as it interacts with Relational databases. We will use python to interact with MongoDB as a NoSQL database. In case you are new to MongoDB, you can learn it in our tutorial [here](#).

In order to connect to MongoDB, python uses a library known as **pymongo**. You can add this library to your python environment, using the below command from the Anaconda environment.

```
conda install pymongo
```

This library enables python to connect to MongoDB using a db client. Once connected we select the db name to be used for various operations.

### Inserting Data

To insert data into MongoDB we use the insert() method which is available in the database environment. First we connect to the db using python code shown below and then we provide the document details in form of a series of key-value pairs.

```
# Import the python libraries
from pymongo import MongoClient
from pprint import pprint

# Choose the appropriate client
client = MongoClient()

# Connect to the test db
db=client.test

# Use the employee collection
employee = db.employee
employee_details = {
    'Name': 'Raj Kumar',
    'Address': 'Sears Streer, NZ',
    'Age': '42'
}

# Use the insert method
result = employee.insert_one(employee_details)

# Query for the inserted document.
Queryresult = employee.find_one({'Age': '42'})
```

```
pprint(Queryresult)
```

When we execute the above code, it produces the following result.

```
{u'Address': u'Sears Streer, NZ',  
u'Age': u'42',  
u'Name': u'Raj Kumar',  
u'_id': ObjectId('5adc5a9f84e7cd3940399f93')}
```

### Updating Data

Updating an existing MongoDB data is similar to inserting. We use the update() method which is native to mongoDB. In the below code we are replacing the existing record with new key-value pairs. Please note how we are using the condition criteria to decide which record to update.

```
# Import the python libraries  
from pymongo import MongoClient  
from pprint import pprint  
  
# Choose the appropriate client  
client = MongoClient()  
  
# Connect to db  
db=client.test  
employee = db.employee  
  
# Use the condition to choose the record  
# and use the update method  
db.employee.update_one(  
    {"Age":'42'},  
    {  
        "$set": {  
            "Name": "Srinidhi",  
            "Age": '35',  
            "Address": "New Omsk, WC"  
        }  
    }  
)  
  
Queryresult = employee.find_one({'Age': '35'})  
  
pprint(Queryresult)
```

When we execute the above code, it produces the following result.

```
{u'Address': u'New Omsk, WC',  
u'Age': u'35',  
u'Name': u'Srinidhi',  
u'_id': ObjectId('5adc5a9f84e7cd3940399f93')}
```

### Deleting Data

Deleting a record is also straight forward where we use the delete method. Here also we mention the condition which is used to choose the record to be deleted.

```
# Import the python libraries  
from pymongo import MongoClient  
from pprint import pprint  
  
# Choose the appropriate client  
client = MongoClient()  
  
# Connect to db
```

```

db=client.test
employee = db.employee

# Use the condition to choose the record
# and use the delete method
db.employee.delete_one({"Age":'35'})

Queryresult = employee.find_one({'Age':'35'})

pprint(Queryresult)

```

When we execute the above code, it produces the following result.

None

So we see the particular record does not exist in the db any more.

## Model Questions

### Objective

1. Which of the following functions is a built-in function in python?

- a) factorial()            b) print()            c) seed()            d) sqrt()

**Answer: b**

2. Which of the following is the use of id() function in python?

- a) Every object doesn't have a unique id            b) Id returns the identity of the object  
c) All of the mentioned            d) None of the mentioned

**Answer: b**

3. The following python program can work with \_\_\_\_parameters.

```

def f(x):
    def f1(*args, **kwargs):
        print("Sanfoundry")
        return x(*args, **kwargs)
    return f1

```

- a) any number of            b) 0            c) 1            d) 2

**Answer: a**

4. What will be the output of the following Python function?

```
min(max(False,-3,-4), 2,7)
```

- a) -4            b) -3            c) 2            d) False

**Answer: d**

5. Which of the following is not a core data type in Python programming?

- a) Tuples            b) Lists            c) Class            d) Dictionary

**Answer: c**

6. What will be the output of the following Python expression if x=56.236?

```
print("%.2f"%x)
```

- a) 56.236            b) 56.23            c) 56.0000            d) 56.24

**Answer: d**

7. Which of these is the definition for packages in Python?

- a) A set of main modules  
b) A folder of python modules  
c) A number of files containing Python definitions and statements  
d) A set of programs making use of Python modules

**Answer: b**

8. What will be the output of the following Python function?

```
len(["hello",2, 4, 6])
```

- a) Error            b) 6            c) 4            d) 3

**Answer: c**

9. What will be the output of the following Python code?

```
x = 'abcd'  
for i in x:  
    print(i.upper())
```

- a) a B C D                      b) a b c d                      c) error                      d) A B C D

**Answer: d**

10. What is the order of namespaces in which Python looks for an identifier?

- a) Python first searches the built-in namespace, then the global namespace and finally the local namespace  
b) Python first searches the built-in namespace, then the local namespace and finally the global namespace  
c) Python first searches the local namespace, then the global namespace and finally the built-in namespace  
d) Python first searches the global namespace, then the local namespace and finally the built-in namespace

**Answer: c**

### **Subjective**

1. Explain in detail about Data Types?
2. Discuss about Text Strings?
3. Discuss about Binary Data?
4. Describe about Storing and Retrieving Data?
5. Explain in detail about File Input/Output?
6. Explain detail about Structured Text Files?
7. Discuss about Structured Binary Files?
8. Explain about Relational Databases?
9. Explain about No SQL Data Stores

---

**Unit – III – END**

**Signature of the staff  
with date**





# Sri Ganesh College of Arts & Science – Salem- 14.

Department of Computer Science & Applications

Study Material – 2022(Odd Semester)

Sub: Open source computing

Paper Code: 21PCS

Class: II M.Sc CS

Staff I/c: K. Aravindhan M.Sc.,

Date:

Head:

---

## UNIT – IV

Web: Web Clients – Web Servers – Web Services and Automation – Systems: Files – Directories – Programs and Processes – Calendars and Clocks

---

### Topic 1: Web Clients

**Possible Question: Explain detail about Web Clients?**

**Possible Marks: 5 or 10 Marks**

#### Outcomes:

The low-level network plumbing of the Internet is called Transmission Control Protocol/Internet Protocol, or more commonly, simply TCP/IP (“TCP/IP” goes into more detail about this). It moves bytes among computers, but doesn’t care about what those bytes mean. That’s the job of higher-level protocols—syntax definitions for specific purposes. HTTP is the standard protocol for web data interchange.

The Web is a client-server system. The client makes a request to a server: it opens a TCP/IP connection, sends the URL and other information via HTTP, and receives a response.

The format of the response is also defined by HTTP. It includes the status of the request, and (if the request succeeded) the response’s data and format.

The most well-known web client is a web browser. It can make HTTP requests in a number of ways. You might initiate a request manually by typing a URL into the location bar or clicking on a link in a web page. Very often, the data returned is used to display a website—HTML documents, JavaScript files, CSS files, and images—but it can be any type of data, not just that intended for display.

An important aspect of HTTP is that it’s stateless. Each HTTP connection that you make is independent of all the others. This simplifies basic web operations but complicates others. Here are just a few samples of the challenges:

#### Caching

Remote content that doesn’t change should be saved by the web client and used to avoid downloading from the server again.

#### Sessions

A shopping website should remember the contents of your shopping cart.

#### Authentication

Sites that require your username and password should remember them while you’re logged in.

Solutions to statelessness include cookies, in which the server sends the client enough specific information to be able to identify it uniquely when the client sends the cookie back.

### **Test with telnet**

HTTP is a text-based protocol, so you can actually type it yourself for web testing. The ancient telnet program lets you connect to any server and port and type commands.

Let's ask everyone's favorite test site, Google, some basic information about its home page. Type this:

```
$ telnet www.google.com 80
```

If there is a web server on port 80 at google.com (I think that's a safe bet), telnet will print some reassuring information and then display a final blank line that's your cue to type something else:

```
Trying 74.125.225.177...
```

```
Connected to www.google.com.
```

```
Escape character is '^['.
```

Now, type an actual HTTP command for telnet to send to the Google web server. The most common HTTP command (the one your browser uses when you type a URL in its location bar) is GET. This retrieves the contents of the specified resource, such as an HTML file, and returns it to the client. For our first test, we'll use the HTTP command HEAD, which just retrieves some basic information about the resource:

### **HEAD / HTTP/1.1**

That HEAD / sends the HTTP HEAD verb (command) to get information about the home page (/). Add an extra carriage return to send a blank line so the remote server knows you're all done and want a response. You'll receive a response such as this (we trimmed some of the long lines using ... so they wouldn't stick out of the book):

### **HTTP/1.1 200 OK**

```
Date: Sat, 26 Oct 2013 17:05:17 GMT
```

```
Expires: -1
```

```
Cache-Control: private, max-age=0
```

```
Content-Type: text/html; charset=ISO-8859-1
```

```
Set-Cookie: PREF=ID=962a70e9eb3db9d9:FF=0:TM=1382807117:LM=1382807117:S=y...  
  expires=Mon, 26-Oct-2015 17:05:17 GMT;
```

```
  path=/;
```

```
  domain=.google.com
```

```
Set-Cookie:
```

```
NID=67=hTvtVC7dZJmZzGktimbwVbNZxPQnaDijCz716B1L56GM9qvsqqeIGb...
```

```
  expires=Sun, 27-Apr-2014 17:05:17 GMT
```

```
  path=/;
```

```
  domain=.google.com;
```

```
  HttpOnly
```

```
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts...
```

```
Server: gws
```

```
X-XSS-Protection: 1; mode=block
```

X-Frame-Options: SAMEORIGIN

Alternate-Protocol: 80:quic

Transfer-Encoding: chunked

These are HTTP response headers and their values. Some, like Date and Content-Type, are required. Others, such as Set-Cookie, are used to track your activity across multiple visits

When you make an HTTP HEAD request, you get back only headers. If you had used the HTTP GET or POST commands, you would also receive data from the home page (a mixture of HTML, CSS, JavaScript, and whatever else Google decided to throw into its home page).

I don't want to leave you stranded in telnet. To close telnet, type the following:

---

## Topic 2: Web Servers

**Possible Question: Discuss about Web Servers?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

Web developers have found Python to be an excellent language for writing web servers and server-side programs. This has led to such a variety of Python-based web frameworks that it can be hard to navigate among them and make choices—not to mention deciding what deserves to go into a book.

A web framework provides features with which you can build websites, so it does more than a simple web (HTTP) server. You'll see features such as routing (URL to server function), templates (HTML with dynamic inclusions), debugging, and more.

I'm not going to cover all of the frameworks here—just those that I've found to be relatively simple to use and suitable for real websites. I'll also show how to run the dynamic parts of a website with Python and other parts with a traditional web server.

### The Simplest Python Web Server

You can run a simple web server by typing just one line of Python:

```
$ python -m http.server
```

This implements a bare-bones Python HTTP server. If there are no problems, this will print an initial status message:

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

That 0.0.0.0 means any TCP address, so web clients can access it no matter what address the server has. There's more low-level details on TCP and other network plumbing

You can now request files, with paths relative to your current directory, and they will be returned. If you type `http://localhost:8000` in your web browser, you should see a directory listing there, and the server will print access log lines such as this:

```
127.0.0.1 - - [20/Feb/2013 22:02:37] "GET / HTTP/1.1" 200 -
```

localhost and 127.0.0.1 are TCP synonyms for your local computer, so this works regardless of whether you're connected to the Internet. You can interpret this line as follows:

127.0.0.1 is the client's IP address

The first "-" is the remote username, if found

The second "-" is the login username, if required

[20/Feb/2013 22:02:37] is the access date and time

"GET / HTTP/1.1" is the command sent to the web server:

The HTTP method (GET)

The resource requested (/, the top)

The HTTP version (HTTP/1.1)

The final 200 is the HTTP status code returned by the web server

Click any file. If your browser can recognize the format (HTML, PNG, GIF, JPEG, and so on) it should display it, and the server will log the request. For instance, if you have the file oreilly.png in your current directory, a request for `http://localhost:8000/oreilly.png` should return the image of the unsettling fellow in Figure 7-1, and the log should show something such as this:

```
127.0.0.1 - - [20/Feb/2013 22:03:48] "GET /oreilly.png HTTP/1.1" 200 -
```

If you have other files in the same directory on your computer, they should show up in a listing on your display, and you can click any one to download it. If your browser is configured to display that file's format, you'll see the results on your screen; otherwise, your browser will ask you if you want to download and save the file.

The default port number used is 8000, but you can specify another:

```
$ python -m http.server 9999
```

You should see this:

```
Serving HTTP on 0.0.0.0 port 9999 ...
```

This Python-only server is best suited for quick tests. You can stop it by killing its process; in most terminals, press Ctrl+C.

You should not use this basic server for a busy production website. Traditional web servers such as Apache and Nginx are much faster for serving static files. In addition, this simple server has no way to handle dynamic content, which more extensive servers can do by accepting parameters.

### **Web Server Gateway Interface**

All too soon, the allure of serving simple files wears off, and we want a web server that can also run programs dynamically. In the early days of the Web, the Common Gateway Interface (CGI) was designed for clients to make web servers run external programs and return the results. CGI also handled getting input arguments from the client through the server to the

external programs. However, the programs were started anew for each client access. This could not scale well, because even small programs have appreciable startup time.

To avoid this startup delay, people began merging the language interpreter into the web server. Apache ran PHP within its `mod_php` module, Perl in `mod_perl`, and Python in `mod_python`. Then, code in these dynamic languages could be executed within the long-running Apache process itself rather than in external programs.

An alternative method was to run the dynamic language within a separate long-running program and have it communicate with the web server. FastCGI and SCGI are examples.

Python web development made a leap with the definition of Web Server Gateway Interface (WSGI), a universal API between Python web applications and web servers. All of the Python web frameworks and web servers in the rest of this use WSGI. You don't normally need to know how WSGI works (there really isn't much to it), but it helps to know what some of the parts under the hood are called.

### **Frameworks**

Web servers handle the HTTP and WSGI details, but you use web frameworks to actually write the Python code that powers the site. So, we'll talk about frameworks for a while and then get back to alternative ways of actually serving sites that use them.

If you want to write a website in Python, there are many Python web frameworks (some might say too many). A web framework handles, at a minimum, client requests and server responses. It might provide some or all of these features:

#### **Routes**

Interpret URLs and find the corresponding server files or Python server code

#### **Templates**

Merge server-side data into pages of HTML

#### **Authentication and authorization**

Handle usernames, passwords, permissions

#### **Sessions**

Maintain transient data storage during a user's visit to the website

In the coming sections, we'll write example code for two frameworks (bottle and flask). Then, we'll talk about alternatives, especially for database-backed websites. You can find a Python framework to power any site that you can think of.

---

### **Topic 3: Web Services and Automation**

**Possible Question: Explain about Web Services and Automation?**

**Possible Marks: 5 or 10 Marks**

#### **Outcomes:**

We've just looked at traditional web client and server applications, consuming and generating HTML pages. Yet the Web has turned out to be a powerful way to glue applications and data in many more formats than HTML.

The webbrowser Module

Let's start begin a little surprise. Start a Python session in a terminal window and type the following:

```
>>> import antigravity
```

This secretly calls the standard library's webbrowser module and directs your browser to an enlightening Python link.<sup>1</sup>

You can use this module directly. This program loads the main Python site's page in your browser:

```
>>> import webbrowser
```

```
>>> url = 'http://www.python.org/'
```

```
>>> webbrowser.open(url)
```

```
True
```

This opens it in a new window:

```
>>> webbrowser.open_new(url)
```

```
True
```

And this opens it in a new tab, if your browser supports tabs:

```
>>> webbrowser.open_new_tab('http://www.python.org/')
```

```
True
```

The webbrowser makes your browser do all the work.

## **Web APIs and Representational State Transfer**

Often, data is only available within web pages. If you want to access it, you need to access the pages through a web browser and read it. If the authors of the website made any changes since the last time you visited, the location and style of the data might have changed.

Instead of publishing web pages, you can provide data through a web application programming interface (API). Clients access your service by making requests to URLs and getting back responses containing status and data. Instead of HTML pages, the data is in formats that are easier for programs to consume, such as JSON or XML

Representational State Transfer (REST) was defined by Roy Fielding in his doctoral thesis. Many products claim to have a REST interface or a RESTful interface. In practice, this often only means that they have a web interface—definitions of URLs to access a web service.

A RESTful service uses the HTTP verbs in specific ways, as is described here:

### **HEAD**

Gets information about the resource, but not its data.

### **GET**

As its name implies, GET retrieves the resource's data from the server. This is the standard method used by your browser. Any time you see a URL with a question mark (?) followed by

a bunch of arguments, that's a GET request. GET should not be used to create, change, or delete data.

## **POST**

This verb updates data on the server. It's often used by HTML forms and web APIs.

## **PUT**

This verb creates a new resource.

## **DELETE**

This one speaks for itself: DELETE deletes. Truth in advertising!

A RESTful client can also request one or more content types from the server by using HTTP request headers. For example, a complex service with a REST interface might prefer its input and output to be JSON strings.

You could extract what you're looking for manually by doing the following:

Type the URL into your browser.

Wait for the remote page to load.

Look through the displayed page for the information you want.

Write it down somewhere.

Possibly repeat the process for related URLs.

However, it's much more satisfying to automate some or all of these steps. An automated web fetcher is called a crawler or spider (unappealing terms to arachnophobes). After the contents have been retrieved from the remote web servers, a scraper parses it to find the needle in the haystack.

If you need an industrial-strength combined crawler and scraper, Scrapy is worth downloading:

```
$ pip install scrapy
```

Scrapy is a framework, not a module such as BeautifulSoup. It does more, but it's more complex to set up. To learn more about Scrapy, read the documentation or the online introduction.

## Scrape HTML with BeautifulSoup

If you already have the HTML data from a website and just want to extract data from it, BeautifulSoup is a good choice. HTML parsing is harder than it sounds. This is because much of the HTML on public web pages is technically invalid: unclosed tags, incorrect nesting, and other complications. If you try to write your own HTML parser by using regular expressions

To install BeautifulSoup, type the following command (don't forget the final 4, or pip will try to install an older version and probably fail):

```
$ pip install beautifulsoup4
```

Now, let's use it to get all the links from a web page. The HTML `a` element represents a link, and `href` is its attribute representing the link destination. In the following example, we'll define the function `get_links()` to do the grunt work, and a main program to get one or more URLs as command-line arguments:

```
def get_links(url):
    import requests
    from bs4 import BeautifulSoup as soup
    result = requests.get(url)
    page = result.text
    doc = soup(page)
    links = [element.get('href') for element in doc.find_all('a')]
    return links

if __name__ == '__main__':
    import sys
    for url in sys.argv[1:]:
        print('Links in', url)
        for num, link in enumerate(get_links(url), start=1):
            print(num, link)
        print()
```

I saved this program as `links.py` and then ran this command:

```
$ python links.py http://boingboing.net
```

Here are the first few lines that it printed:

```
Links in http://boingboing.net/
1 http://boingboing.net/suggest.html
2 http://boingboing.net/category/feature/
3 http://boingboing.net/category/review/
4 http://boingboing.net/category/podcasts
5 http://boingboing.net/category/video/
6 http://bbs.boingboing.net/
7 javascript:void(0)
8 http://shop.boingboing.net/
9 http://boingboing.net/about
10 http://boingboing.net/contact
```

### Things to Do

9.1. If you haven't installed flask yet, do so now. This will also install `werkzeug`, `jinja2`, and possibly other packages.

9.2. Build a skeleton website, using Flask's `debug/reload` development web server. Ensure that the server starts up for hostname `localhost` on default port `5000`. If your computer is already using port `5000` for something else, use another port number.



9.3. Add a `home()` function to handle requests for the home page. Set it up to return the string `It's alive!`.

9.4. Create a Jinja2 template file called `home.html` with the following contents:

```
<html>
<head>
<title>It's alive!</title>
<body>
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
</body>
</html>
```

9.5. Modify your server's `home()` function to use the `home.html` template. Provide it with three GET parameters: `thing`, `height`, and `color`.

---

## Topic 4: Files

**Possible Question: Explain about Files?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

### File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

### Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

---

## Topic 5: Directories

**Possible Question: Discuss about Directories?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

If there are a large number of files to handle in our Python program, we can arrange our code within different directories to make things more manageable.

A directory or folder is a collection of files and subdirectories. Python has the os module that provides us with many useful methods to work with directories (and files as well).

### Get Current Directory

We can get the present working directory using the `getcwd()` method of the `os` module.

This method returns the current working directory in the form of a string. We can also use the `getcwdb()` method to get it as bytes object.

```
>>> import os
```

```
>>> os.getcwd()
```

```
'C:\\Program Files\\PyScripter'
```

```
>>> os.getcwdb()
```

```
b'C:\\Program Files\\PyScripter'
```

The extra backslash implies an escape sequence. The `print()` function will render this properly.

```
>>> print(os.getcwd())
```

```
C:\Program Files\PyScripter
```

### Changing Directory

We can change the current working directory by using the `chdir()` method.

The new path that we want to change into must be supplied as a string to this method. We can use both the forward-slash `/` or the backward-slash `\` to separate the path elements.

It is safer to use an escape sequence when using the backward slash.

```
>>> os.chdir('C:\\Python33')
```

```
>>> print(os.getcwd())
```

```
C:\Python33
```

### List Directories and Files

All files and sub-directories inside a directory can be retrieved using the `listdir()` method.

This method takes in a path and returns a list of subdirectories and files in that path. If no path is specified, it returns the list of subdirectories and files from the current working directory.

```
>>> print(os.getcwd())
```

```
C:\Python33
```

```
>>> os.listdir()
```

```
['DLLs',
```

```
'Doc',
```

```
'include',
'Lib',
'libs',
'LICENSE.txt',
'NEWS.txt',
'python.exe',
'pythonw.exe',
'README.txt',
'Scripts',
'tcl', 'Tools']
>>> os.listdir('G:\\')
['$RECYCLE.BIN',
'Movies',
'Music',
'Photos',
'Series',
'System Volume Information']
```

### **Making a New Directory**

We can make a new directory using the `mkdir()` method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

```
>>> os.mkdir('test')
>>> os.listdir()
['test']
```

### **Renaming a Directory or a File**

The `rename()` method can rename a directory or a file.

For renaming any directory or file, the `rename()` method takes in two basic arguments: the old name as the first argument and the new name as the second argument.

```
>>> os.listdir()
['test']
>>> os.rename('test','new_one')
>>> os.listdir()
['new_one']
```

### **Removing Directory or File**

A file can be removed (deleted) using the `remove()` method.

Similarly, the `rmdir()` method removes an empty directory.

```
>>> os.listdir()
```

```
['new_one', 'old.txt']
>>> os.remove('old.txt')
>>> os.listdir()
['new_one']
>>> os.rmdir('new_one')
>>> os.listdir()
```

**Note:** The `rmdir()` method can only remove empty directories.

In order to remove a non-empty directory, we can use the `rmtree()` method inside the `shutil` module.

```
>>> os.listdir()
['test']
>>> os.rmdir('test')
```

Traceback (most recent call last):

```
OSError: [WinError 145] The directory is not empty: 'test'
```

```
>>> import shutil
>>> shutil.rmtree('test')
>>> os.listdir()
```

---

## **Topic 6: Programs and Processes**

**Possible Question: Explain about Programs and Processes?**

**Possible Marks: 5 or 10 Marks**

### **Outcomes:**

When you run an individual program, your operating system creates a single process. It uses system resources (CPU, memory, disk space) and data structures in the operating system's kernel (file and network connections, usage statistics, and so on). A process is isolated from other processes—it can't see what other processes are doing or interfere with them.

The operating system keeps track of all the running processes, giving each a little time to run and then switching to another, with the twin goals of spreading the work around fairly and being responsive to the user. You can see the state of your processes with graphical interfaces such as the Mac's Activity Monitor (OS X), or Task Manager on Windows-based computers.

You can also access process data from your own programs. The standard library's `os` module provides a common way of accessing some system information. For instance, the following functions get the process ID and the current working directory of the running Python interpreter:

```
>>> import os
>>> os.getpid()
76051
>>> os.getcwd()
'/Users/williamlubanovic'
```

And these get my user ID and group ID:

```
>>> os.getuid()
501
>>> os.getgid()
20
```

### Create a Process with subprocess

All of the programs that you've seen here so far have been individual processes. You can start and stop other existing programs from Python by using the standard library's subprocess module. If you just want to run another program in a shell and grab whatever output it created (both standard output and standard error output), use the `getoutput()` function. Here, we'll get the output of the Unix `date` program:

```
>>> import subprocess
>>> ret = subprocess.getoutput('date')
>>> ret
'Sun Mar 30 22:54:37 CDT 2014'
```

You won't get anything back until the process ends. If you need to call something that might take a lot of time, see the discussion on concurrency in "Concurrency". Because the argument to `getoutput()` is a string representing a complete shell command, you can include arguments, pipes, `<` and `>` I/O redirection, and so on:

```
>>> ret = subprocess.getoutput('date -u')
>>> ret
'Mon Mar 31 03:55:01 UTC 2014'
```

Piping that output string to the `wc` command counts one line, six "words," and 29 characters:

```
>>> ret = subprocess.getoutput('date -u | wc')
>>> ret
'  1  6 29'
```

A variant method called `check_output()` takes a list of the command and arguments. By default it only returns standard output as type bytes rather than a string and does not use the shell:

```
>>> ret = subprocess.check_output(['date', '-u'])
>>> ret
b'Mon Mar 31 04:01:50 UTC 2014\n'
```

To show the exit status of the other program, `getstatusoutput()` returns a tuple with the status code and output:

```
>>> ret = subprocess.getstatusoutput('date')
>>> ret
(0, 'Sat Jan 18 21:36:23 CST 2014')
```

If you don't want to capture the output but might want to know its exit status, use `call()`:

```
>>> ret = subprocess.call('date')
Sat Jan 18 21:33:11 CST 2014
>>> ret
0
```

(In Unix-like systems, 0 is usually the exit status for success.)

That date and time was printed to output but not captured within our program. So, we saved the return code as ret.

You can run programs with arguments in two ways. The first is to specify them in a single string. Our sample command is `date -u`, which prints the current date and time in UTC (you'll read more about UTC in a few pages):

```
>>> ret = subprocess.call('date -u', shell=True)
```

```
Tue Jan 21 04:40:04 UTC 2014
```

You need that `shell=True` to recognize the command line `date -u`, splitting it into separate strings and possibly expanding any wildcard characters such as `*` (we didn't use any in this example).

The second method makes a list of the arguments, so it doesn't need to call the shell:

```
>>> ret = subprocess.call(['date', '-u'])
```

```
Tue Jan 21 04:41:59 UTC 2014
```

Create a Process with multiprocessing

You can run a Python function as a separate process or even run multiple independent processes in a single program with the multiprocessing module. Here's a short example that does nothing useful; save it as `mp.py` and then run it by typing `python mp.py`:

```
import multiprocessing
import os

def do_this(what):
    whoami(what)

def whoami(what):
    print("Process %s says: %s" % (os.getpid(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = multiprocessing.Process(target=do_this,
                                   args=("I'm function %s" % n,))
        p.start()
```

When I run this, my output looks like this:

```
Process 6224 says: I'm the main program
```

```
Process 6225 says: I'm function 0
```

```
Process 6226 says: I'm function 1
```

```
Process 6227 says: I'm function 2
```

```
Process 6228 says: I'm function 3
```

The `Process()` function spawned a new process and ran the `do_this()` function in it. Because we did this in a loop that had four passes, we generated four new processes that executed `do_this()` and then exited.

The multiprocessing module has more bells and whistles than a clown on a calliope. It's really intended for those times when you need to farm out some task to multiple processes to save overall time; for example, downloading web pages for scraping, resizing images, and so on. It includes ways to queue tasks, enable intercommunication among processes, and wait for all the processes to finish. "Concurrency" delves into some of these details.

Kill a Process with terminate()

If you created one or more processes and want to terminate one for some reason (perhaps it's stuck in a loop, or maybe you're bored, or you want to be an evil overlord), use terminate(). In the example that follows, our process would count to a million, sleeping at each step for a second, and printing an irritating message. However, our main program runs out of patience in five seconds and nukes it from orbit:

```
import multiprocessing
import time
import os
def whoami(name):
    print("I'm %s, in process %s" % (name, os.getpid()))
def loopy(name):
    whoami(name)
    start = 1
    stop = 1000000
    for num in range(start, stop):
        print("\tNumber %s of %s. Honk!" % (num, stop))
        time.sleep(1)
if __name__ == "__main__":
    whoami("main")
    p = multiprocessing.Process(target=loopy, args=("loopy",))
    p.start()
    time.sleep(5)
    p.terminate()
```

When I run this program, I get the following:

I'm main, in process 97080

I'm loopy, in process 97081

Number 1 of 1000000. Honk!

Number 2 of 1000000. Honk!

Number 3 of 1000000. Honk!

Number 4 of 1000000. Honk!

Number 5 of 1000000. Honk!

---

## Topic 7: Calendars and Clocks

**Possible Question: Explain about Calendars and Clocks?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Programmers devote a surprising amount of effort to dates and times. Let's talk about some of the problems they encounter, and then get to some best practices and tricks to make the situation a little less messy.

Dates can be represented in many ways—too many ways, actually. Even in English with the Roman calendar, you’ll see many variants of a simple date:

```
July 29 1984
29 Jul 1984
29/7/1984
7/29/1984
```

Among other problems, date representations can be ambiguous. In the previous examples, it’s easy to determine that 7 stands for the month and 29 is the day of the month, largely because months don’t go to 29. But how about 1/6/2012? Is that referring to January 6 or June 1? The month name varies by language within the Roman calendar. Even the year and month can have a different definition in other cultures.

Leap years are another wrinkle. You probably know that every four years is a leap year (and the summer Olympics and the American presidential election). Did you also know that every 100 years is not a leap year, but that every 400 years is? Here’s code to test various years for leapiness:

```
>>> import calendar
>>> calendar.isleap(1900)
False
>>> calendar.isleap(1996)
True
>>> calendar.isleap(1999)
False
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2002)
False
>>> calendar.isleap(2004)
True
```

Times have their own sources of grief, especially because of time zones and daylight savings time. If you look at a time zone map, the zones follow political and historic boundaries rather than every 15 degrees (360 degrees / 24) of longitude. And countries start and end daylight saving times on different days of the year. In fact, countries in the southern hemisphere advance their clocks when the northern hemisphere is winding them back, and vice versa. (If you think about it a bit, you will see why.)

Python’s standard library has many date and time modules: `datetime`, `time`, `calendar`, `dateutil`, and others. There’s some overlap, and it’s a bit confusing.

### **The datetime Module**

Let’s begin by investigating the standard `datetime` module. It defines four main objects, each with many methods:

- `date` for years, months, and days
- `time` for hours, minutes, seconds, and fractions
- `datetime` for dates and times together
- `timedelta` for date and/or time intervals

You can make a date object by specifying a year, month, and day. Those values are then available as attributes:

```
>>> from datetime import date
>>> halloween = date(2014, 10, 31)
>>> halloween
```



```
datetime.date(2014, 10, 31)
```

```
>>> halloween.day
```

```
31
```

```
>>> halloween.month
```

```
10
```

```
>>> halloween.year
```

```
2014
```

You can print a date with its `isoformat()` method:

```
>>> halloween.isoformat()
```

```
'2014-10-31'
```

The iso refers to ISO 8601, an international standard for representing dates and times. It goes from most general (year) to most specific (day). It also sorts correctly: by year, then month, then day. I usually pick this format for date representation in programs, and for filenames that save data by date. The next section describes the more complex `strptime()` and `strftime()` methods for parsing and formatting dates.

This example uses the `today()` method to generate today's date:

```
>>> from datetime import date
```

```
>>> now = date.today()
```

```
>>> now
```

```
datetime.date(2014, 2, 2)
```

This one makes use of a `timedelta` object to add some time interval to a date:

```
>>> from datetime import timedelta
```

```
>>> one_day = timedelta(days=1)
```

```
>>> tomorrow = now + one_day
```

```
>>> tomorrow
```

```
datetime.date(2014, 2, 3)
```

```
>>> now + 17*one_day
```

```
datetime.date(2014, 2, 19)
```

```
>>> yesterday = now - one_day
```

```
>>> yesterday
```

```
datetime.date(2014, 2, 1)
```

The range of `date` is from `date.min` (year=1, month=1, day=1) to `date.max` (year=9999, month=12, day=31). As a result, you can't use it for historic or astronomical calculations.

The `datetime` module's `time` object is used to represent a time of day:

```
>>> from datetime import time
```

```
>>> noon = time(12, 0, 0)
```

```
>>> noon
```

```
datetime.time(12, 0)
```

```
>>> noon.hour
```

```
12
```

```
>>> noon.minute
```

```
0
```

```
>>> noon.second
```

```
0
```

```
>>> noon.microsecond
```

```
0
```

The arguments go from the largest time unit (hours) to the smallest (microseconds). If you don't provide all the arguments, `time` assumes all the rest are zero. By the way, just because

you can store and retrieve microseconds doesn't mean you can retrieve time from your computer to the exact microsecond. The accuracy of subsecond measurements depends on many factors in the hardware and operating system.

The datetime object includes both the date and time of day. You can create one directly, such as the one that follows, which is for January 2, 2014, at 3:04 A.M., plus 5 seconds and 6 microseconds:

```
>>> from datetime import datetime
>>> some_day = datetime(2014, 1, 2, 3, 4, 5, 6)
>>> some_day
datetime.datetime(2014, 1, 2, 3, 4, 5, 6)
```

The datetime object also has an isoformat() method:

```
>>> some_day.isoformat()
'2014-01-02T03:04:05.000006'
```

That middle T separates the date and time parts.

datetime has a now() method with which you can get the current date and time:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2014, 2, 2, 23, 15, 34, 694988)
14
>>> now.month
2
>>> now.day
2
>>> now.hour
23
>>> now.minute
15
>>> now.second
34
>>> now.microsecond
694988
```

You can merge a date object and a time object into a datetime object by using combine():

```
>>> from datetime import datetime, time, date
>>> noon = time(12)
>>> this_day = date.today()
>>> noon_today = datetime.combine(this_day, noon)
>>> noon_today
datetime.datetime(2014, 2, 2, 12, 0)
```

You can yank the date and time from a datetime by using the date() and time() methods:

```
>>> noon_today.date()
datetime.date(2014, 2, 2)
>>> noon_today.time()
datetime.time(12, 0)
```

### Using the time Module

It is confusing that Python has a datetime module with a time object, and a separate time module. Furthermore, the time module has a function called—wait for it—time().

One way to represent an absolute time is to count the number of seconds since some starting point. Unix time uses the number of seconds since midnight on January 1, 1970.<sup>1</sup> This value is often called the epoch, and it is often the simplest way to exchange dates and times among systems.

The time module's `time()` function returns the current time as an epoch value:

```
>>> import time
>>> now = time.time()
>>> now
1391488263.664645
```

If you do the math, you'll see that it has been over one billion seconds since New Year's, 1970. Where did the time go?

You can convert an epoch value to a string by using `ctime()`:

```
>>> time.ctime(now)
'Mon Feb  3 22:31:03 2014'
```

In the next section, you'll see how to produce more attractive formats for dates and times.

Epoch values are a useful least-common denominator for date and time exchange with different systems, such as JavaScript. Sometimes, though, you need actual days, hours, and so forth, which time provides as `struct_time` objects. `localtime()` provides the time in your system's time zone, and `gmtime()` provides it in UTC:

```
>>> time.localtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=3, tm_hour=22, tm_min=31,
tm_sec=3, tm_wday=0, tm_yday=34, tm_isdst=0)
>>> time.gmtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=4, tm_min=31,
tm_sec=3, tm_wday=1, tm_yday=35, tm_isdst=0)
```

In my (Central) time zone, 22:31 was 04:31 of the next day in UTC (formerly called Greenwich time or Zulu time). If you omit the argument to `localtime()` or `gmtime()`, they assume the current time.

The opposite of these is `mktime()`, which converts a `struct_time` object to epoch seconds:

```
>>> tm = time.localtime(now)
>>> time.mktime(tm)
1391488263.0
```

This doesn't exactly match our earlier epoch value of `now()` because the `struct_time` object preserves time only to the second.

Some advice: wherever possible, use UTC instead of time zones. UTC is an absolute time, independent of time zones. If you have a server, set its time to UTC; do not use local time.

Here's some more advice (free of charge, no less): never use daylight savings time if you can avoid it. If you use daylight savings time, an hour disappears at one time of year ("spring ahead") and occurs twice at another time ("fall back"). For some reason, many organizations use daylight savings in their computer systems, but are mystified every year by data duplicates and dropouts. It all ends in tears.

---

## Model Questions

### Objective

1. What will be the output of the following Python code snippet?

```
for i in [1, 2, 3, 4][::-1]:  
    print(i)
```

- a) 4 3 2 1                      b) error                      c) 1 2 3 4                      d) none of the mentioned

**Answer: a**

Explanation: [::-1] reverses the list.

2. What will be the output of the following Python statement?

```
1. >>>"a"+"bc"
```

- a) bc                              b) abc                              c) a                              d) bca

**Answer: b**

3. Which function is called when the following Python program is executed?

```
f = foo()  
format(f)
```

- a) str()                              b) format()                              c) \_\_str\_\_()                              d) \_\_format\_\_()

**Answer: c**

4. Which one of the following is not a keyword in Python language?

- a) pass                      b) eval                      c) assert                      d) nonlocal

**Answer: b**

5. What will be the output of the following Python code?

```
1. class tester:  
2.     def __init__(self, id):  
3.         self.id = str(id)  
4.         id="224"  
5.  
6. >>>temp = tester(12)  
7. >>>print(temp.id)
```

- a) 12                              b) 224                              c) None                              d) Error

**Answer: a**

6. What will be the output of the following Python program?

```
def foo(x):  
    x[0] = ['def']  
    x[1] = ['abc']  
    return id(x)  
q = ['abc', 'def']  
print(id(q) == foo(q))
```

- a) Error                              b) None                              c) False                              d) True

**Answer: d**

7. Which module in the python standard library parses options received from the command line?

- a) getarg                              b) getopt                              c) main                              d) os

**Answer: b**

8. What will be the output of the following Python program?

```
z=set('abc')  
z.add('san')  
z.update(set(['p', 'q']))  
z
```

- a) {'a', 'c', 'c', 'p', 'q', 's', 'a', 'n'}                              b) {'abc', 'p', 'q', 'san'}  
c) {'a', 'b', 'c', 'p', 'q', 'san'}                              d) {'a', 'b', 'c', ['p', 'q'], 'san'}

**Answer: c**

9. What arithmetic operators cannot be used with strings in Python?

- a) \*                      b) –                      c) +                      d) All of the mentioned

**Answer: b**

10. What will be the output of the following Python code?

```
print("abc. DEF".capitalize())
```

- a) Abc. Def                      b) abc. Def                      c) Abc. Def                      d) ABC. DEF

**Answer: a**

### **Subjective**

1. Explain in detail about Web?
2. Discuss about Web Clients?
3. Discuss about Web Servers?
4. Explain Web Services and Automation?
5. Describe about Systems?
6. Explain in detail about Files?
7. Explain in detail about Directories?
8. Discuss about Programs and Processes?
9. Explain about Calendars and Clocks?

---

**Unit – IV – END**

**Signature of the staff  
with date**



# Sri Ganesh College of Arts & Science – Salem- 14.

Department of Computer Science & Applications

Study Material – 2022(Odd Semester)

Sub: Open source computing

Paper Code: 21PCS

Class: II M.Sc CS

Staff I/c: K. Aravindhan M.Sc.,

Date:

Head:

---

## UNIT – V

Concurrency: Queues – Processes – Threads – Green Threads and event – twisted – Redis.  
Networks: Patterns – The Publish-Subscribe Model – TCP/IP – Sockets – Zero MQ –Internet  
Services – Web Services and APIs – Remote Processing – Big Fat Data and Map Reduce –  
Working in the Clouds.

---

### Topic 1: Concurrency Queues

**Possible Question: Discuss about Concurrency in queues?**

**Possible Marks: 5 or 10 Marks**

#### Outcomes:

The official Python site discusses concurrency in general and in the standard library. Those pages have many links to various packages and techniques;

In computers, if you're waiting for something, it's usually for one of two reasons:

I/O bound

This is by far more common. Computer CPUs are ridiculously fast—hundreds of times faster than computer memory and many thousands of times faster than disks or networks.

CPU bound

This happens with number crunching tasks such as scientific or graphic calculations.

Two more terms are related to concurrency:

synchronous

One thing follows the other, like a funeral procession.

asynchronous

Tasks are independent, like party-goers dropping in and tearing off in separate cars.

As you progress from simple systems and tasks to real-life problems, you'll need at some point to deal with concurrency. Consider a website, for example. You can usually provide static and dynamic pages to web clients fairly quickly. A fraction of a second is considered interactive, but if the display or interaction takes longer, people become impatient. Tests by companies such as Google and Amazon showed that traffic drops off quickly if the page loads even a little slower.

But what if you can't help it when something takes a long time, such as uploading a file, resizing an image, or querying a database? You can't do it within your synchronous web server code anymore, because someone's waiting.

On a single machine, if you want to perform multiple tasks as fast as possible, you want to make them independent. Slow tasks shouldn't block all the others.

“Programs and Processes” demonstrates how multiprocessing can be used to overlap work on a single machine. If you needed to resize an image, your web server code could call a separate, dedicated image resizing process to run asynchronously and concurrently. It could scale your application horizontally by invoking multiple resizing processes.

The trick is getting them all to work with one another. Any shared control or state means that there will be bottlenecks. An even bigger trick is dealing with failures, because concurrent

computing is harder than regular computing. Many more things can go wrong, and your odds of end-to-end success are lower.

A queue is like a list: things are added at one end and taken away from the other. The most common is referred to as *FIFO* (first in, first out).

Suppose that you're washing dishes. If you're stuck with the entire job, you need to wash each dish, dry it, and put it away. You can do this in a number of ways. You might wash the first dish, dry it, and then put it away. You then repeat with the second dish, and so on. Or, you might *batch* operations and wash all the dishes, dry them all, and then put them away; this assumes you have space in your sink and drainer for all the dishes that accumulate at each step. These are all synchronous approaches—one worker, one thing at a time.

As an alternative, you could get a helper or two. If you're the washer, you can hand each cleaned dish to the dryer, who hands each dried dish to the put-away-er (look it up; it's absolutely a real word!). As long as each of you works at the same pace, you should finish much faster than by yourself.

However, what if you wash faster than the dryer dries? Wet dishes either fall on the floor, or you pile them up between you and the dryer, or you just whistle off-key until the dryer is ready. And if the last person is slower than the dryer, dry dishes can end up falling on the floor, or piling up, or the dryer does the whistling. You have multiple workers, but the overall task is still synchronous and can proceed only as fast as the slowest worker.

*Many hands make light work*, goes the old saying (I always thought it was Amish, because it makes me think of barn building). Adding workers can build a barn, or do the dishes, faster. This involves *queues*.

In general, queues transport *messages*, which can be any kind of information. In this case, we're interested in queues for distributed task management, also known as *work queues*, *job queues*, or *task queues*. Each dish in the sink is given to an available washer, who washes and hands it off to the first available dryer, who dries and hands it to a put-away-er. This can be synchronous (workers wait for a dish to handle and another worker to whom to give it), or asynchronous (dishes are stacked between workers with different paces). As long as you have enough workers, and they keep up with the dishes, things move a lot faster.

---

## Topic 2: Processes

**Possible Question: Discuss about Processes?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

You can implement queues in many ways. For a single machine, the standard library's multiprocessing module (which you can see in "Programs and Processes") contains a Queue function. Let's simulate just a single washer and multiple dryer processes (someone can put the dishes away later) and an intermediate dish\_queue. Call this program dishes.py:

```
import multiprocessing as mp
def washer(dishes, output):
    for dish in dishes:
        print('Washing', dish, 'dish')
        output.put(dish)

def dryer(input):
    while True:
        dish = input.get()
        print('Drying', dish, 'dish')
```

```
input.task_done()
```

```
dish_queue = mp.JoinableQueue()
dryer_proc = mp.Process(target=dryer, args=(dish_queue,))
dryer_proc.daemon = True
dryer_proc.start()
```

```
dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

Run your new program thusly:

```
$ python dishes.py
Washing salad dish
Washing bread dish
Washing entree dish
Washing dessert dish
Drying salad dish
Drying bread dish
Drying entree dish
Drying dessert dish
```

This queue looked a lot like a simple Python iterator, producing a series of dishes. It actually started up separate processes along with the communication between the washer and dryer. I used a `JoinableQueue` and the final `join()` method to let the washer know that all the dishes have been dried. There are other queue types in the multiprocessing module, and you can read the documentation for more examples.

---

### Topic 3: Threads

#### Possible Question: Discuss about Threads?

#### Possible Marks: 5 or 10 Marks

#### Outcomes:

A thread runs within a process with access to everything in the process, similar to a multiple personality. The multiprocessing module has a cousin called `threading` that uses threads instead of processes (actually, multiprocessing was designed later as its process-based counterpart). Let's redo our process example with threads:

```
import threading

def do_this(what):
    whoami(what)

def whoami(what):
    print("Thread %s says: %s" % (threading.current_thread(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = threading.Thread(target=do_this,
                             args=("I'm function %s" % n,))
        p.start()
```



Here's what prints for me:

```
Thread <_MainThread(MainThread, started 140735207346960)> says: I'm the main
program
Thread <Thread(Thread-1, started 4326629376)> says: I'm function 0
Thread <Thread(Thread-2, started 4342157312)> says: I'm function 1
Thread <Thread(Thread-3, started 4347412480)> says: I'm function 2
Thread <Thread(Thread-4, started 4342157312)> says: I'm function 3
```

We can reproduce our process-based dish example by using threads:

```
import threading, queue
import time

def washer(dishes, dish_queue):
    for dish in dishes:
        print ("Washing", dish)
        time.sleep(5)
        dish_queue.put(dish)

def dryer(dish_queue):
    while True:
        dish = dish_queue.get()
        print ("Drying", dish)
        time.sleep(10)
        dish_queue.task_done()

dish_queue = queue.Queue()
for n in range(2):
    dryer_thread = threading.Thread(target=dryer, args=(dish_queue,))
    dryer_thread.start()

dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

One difference between multiprocessing and threading is that threading does not have a `terminate()` function. There's no easy way to terminate a running thread, because it can cause all sorts of problems in your code, and possibly in the space-time continuum itself.

Threads can be dangerous. Like manual memory management in languages such as C and C++, they can cause bugs that are extremely hard to find, let alone fix. To use threads, all the code in the program—and in external libraries that it uses—must be thread-safe. In the preceding example code, the threads didn't share any global variables, so they could run independently without breaking anything.

Imagine that you're a paranormal investigator in a haunted house. Ghosts roam the halls, but none are aware of the others, and at any time, any of them can view, add, remove, or move any of the house's contents.

You're walking apprehensively through the house, taking readings with your impressive instruments. Suddenly you notice that the candlestick you passed seconds ago is now missing. The contents of the house are like the variables in a program. The ghosts are threads in a process (the house). If the ghosts only looked at the house's contents, there would be no problem. It's like a thread reading the value of a constant or variable without trying to change it.

Yet, some unseen entity could grab your flashlight, blow cold air down your neck, put marbles on the stairs, or make the fireplace come ablaze. The really subtle ghosts would change things in other rooms that you might never notice.

Despite your fancy instruments, you'd have a very hard time figuring out who did it, and how, and when.

If you used multiple processes instead of threads, it would be like having a number of houses but with only one (living) person in each. If you put your brandy in front of the fireplace, it would still be there an hour later. Some lost to evaporation, perhaps, but in the same place.

Threads can be useful and safe when global data is not involved. In particular, threads are useful for saving time while waiting for some I/O operation to complete. In these cases, they don't have to fight over data, because each has completely separate variables.

But threads do sometimes have good reasons to change global data. In fact, one common reason to launch multiple threads is to let them divide up the work on some data, so a certain degree of change to the data is expected.

The usual way to share data safely is to apply a software lock before modifying a variable in a thread. This keeps the other threads out while the change is made. It's like having a Ghostbuster guard the room you want to remain unhaunted. The trick, though, is that you need to remember to unlock it. Plus, locks can be nested—what if another Ghostbuster is also watching the same room, or the house itself? The use of locks is traditional but notoriously hard to get right.

---

#### **Topic 4: Green Threads and gevent**

**Possible Question: Explain about Green Threads and gevent?**

**Possible Marks: 5 or 10 Marks**

##### **Outcomes:**

As you've seen, developers traditionally avoid slow spots in programs by running them in separate threads or processes. The Apache web server is an example of this design.

One alternative is event-based programming. An event-based program runs a central event loop, doles out any tasks, and repeats the loop. The nginx web server follows this design, and is generally faster than Apache.

The gevent library is event-based and accomplishes a cool trick: you write normal imperative code, and it magically converts pieces to coroutines. These are like generators that can communicate with one another and keep track of where they are. gevent modifies many of Python's standard objects such as socket to use its mechanism instead of blocking. This does not work with Python add-in code that was written in C, as some database drivers are.

##### **NOTE**

As of this writing, gevent was not completely ported to Python 3, so these code examples use the Python 2 tools pip2 and python2.

You install gevent by using the Python 2 version of pip:

```
$ pip2 install gevent
```

Here's a variation of sample code at the gevent website. You'll see the socket module's gethostbyname() function in the upcoming DNS section. This function is synchronous, so you wait (possibly many seconds) while it chases name servers around the world to look up that address. But you could use the gevent version to look up multiple sites independently. Save this as gevent\_test.py:

```

import gevent
from gevent import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
        'www.antique-taxidermy.com']
jobs = [gevent.spawn(gevent.socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)

```

There's a one-line for-loop in the preceding example. Each hostname is submitted in turn to a `gethostbyname()` call, but they can run asynchronously because it's the `gevent` version of `gethostbyname()`.

Run `gevent_test.py` with Python 2 by typing the following (in bold):

```

$ python2 gevent_test.py
66.6.44.4
74.125.142.121
78.136.12.50

```

`gevent.spawn()` creates a greenlet (also known sometimes as a green thread or a microthread) to execute each `gevent.socket.gethostbyname(url)`.

The difference from a normal thread is that it doesn't block. If something occurred that would have blocked a normal thread, `gevent` switches control to one of the other greenlets.

The `gevent.joinall()` method waits for all the spawned jobs to finish. Finally, we dump the IP addresses that we got for these hostnames.

Instead of the `gevent` version of `socket`, you can use its evocatively named monkey-patching functions. These modify standard modules such as `socket` to use greenlets rather than calling the `gevent` version of the module. This is useful when you want `gevent` to be applied all the way down, even into code that you might not be able to access.

At the top of your program, add the following call:

```

from gevent import monkey
monkey.patch_socket()

```

This inserts the `gevent` `socket` everywhere the normal `socket` is called, anywhere in your program, even in the standard library. Again, this works only for Python code, not libraries written in C.

Another function `monkey`-patches even more standard library modules:

```

from gevent import monkey
monkey.patch_all()

```

Use this at the top of your program to get as many `gevent` speedups as possible.

Save this program as `gevent_monkey.py`:

```

import gevent
from gevent import monkey; monkey.patch_all()
import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
        'www.antique-taxidermy.com']
jobs = [gevent.spawn(socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)

```

Again, using Python 2, run the program:

```
$ python2 gevent_monkey.py
66.6.44.4
74.125.192.121
78.136.12.50
```

There are potential dangers when using `gevent`. As with any event-based system, each chunk of code that you execute should be relatively quick. Although it's nonblocking, code that does a lot of work is still slow.

The very idea of monkey-patching makes some people nervous. Yet, many large sites such as Pinterest use `gevent` to speed up their sites significantly. Like the fine print on a bottle of pills, use `gevent` as directed.

---

## Topic 5: Twisted

**Possible Question: Explain about twisted?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

Twisted is an asynchronous, event-driven networking framework. You connect functions to events such as data received or connection closed, and those functions are called when those events occur. This is a callback design, and if you've written anything in JavaScript, it might seem familiar. If it's new to you, it can seem backwards. For some developers, callback-based code becomes harder to manage as the application grows.

Like `gevent`, `twisted` has not yet been ported to Python 3. We'll use the Python 2 installer and interpreter for this section. Type the following to install it:

```
$ pip2 install twisted
```

`twisted` is a large package, with support for many Internet protocols on top of TCP and UDP. To be short and simple, we'll show a little knock-knock server and client, adapted from `twisted` examples. First, let's look at the server, `knock_server.py` (notice the Python 2 syntax for `print()`):

```
from twisted.internet import protocol, reactor
```

```
class Knock(protocol.Protocol):
```

```
    def dataReceived(self, data):
```

```
        print 'Client:', data
```

```
        if data.startswith("Knock knock"):
```

```
            response = "Who's there?"
```

```
        else:
```

```
            response = data + " who?"
```

```
        print 'Server:', response
```

```
        self.transport.write(response)
```

```
class KnockFactory(protocol.Factory):
```

```
    def buildProtocol(self, addr):
```

```
        return Knock()
```

```
reactor.listenTCP(8000, KnockFactory())
```

```
reactor.run()
```

Now, let's take a glance at its trusty companion, `knock_client.py`:

```
from twisted.internet import reactor, protocol
```

```
class KnockClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("Knock knock")

    def dataReceived(self, data):
        if data.startswith("Who's there?"):
            response = "Disappearing client"
            self.transport.write(response)
        else:
            self.transportloseConnection()
            reactor.stop()
```

```
class KnockFactory(protocol.ClientFactory):
    protocol = KnockClient
```

```
def main():
    f = KnockFactory()
    reactor.connectTCP("localhost", 8000, f)
    reactor.run()
```

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

Start the server first:

```
$ python2 knock_server.py
```

Then start the client:

```
$ python2 knock_client.py
```

The server and client exchange messages, and the server prints the conversation:

Client: Knock knock

Server: Who's there?

Client: Disappearing client

Server: Disappearing client who?

Our trickster client then ends, keeping the server waiting for the punch line.

If you'd like to enter the twisted passages, try some of the other examples from its documentation.

---

## Topic 6: Redis

**Possible Question: Describe about Redis?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Our earlier dishwashing code examples, using processes or threads, were run on a single machine. Let's take another approach to queues that can run on a single machine or across a network. Even with multiple singing processes and dancing threads, sometimes one machine isn't enough, You can treat this section as a bridge between single-box (one machine) and multiple-box concurrency.

To try the examples in this section, you'll need a Redis server and its Python module. You can see where to get them in "Redis".

A quick way to make a queue is with a Redis list. A Redis server runs on one machine; this can be the same one as its clients, or another that the clients can access through a network. In either case, clients talk to the server via TCP, so they're networking. One or more provider clients pushes messages onto one end of the list. One or more client workers watches this list with a blocking pop operation. If the list is empty, they all just sit around playing cards. As soon as a message arrives, the first eager worker gets it.

Like our earlier process- and thread-based examples, `redis_washer.py` generates a sequence of dishes:

```
import redis
conn = redis.Redis()
print('Washer is starting')
dishes = ['salad', 'bread', 'entree', 'dessert']
for dish in dishes:
    msg = dish.encode('utf-8')
    conn.rpush('dishes', msg)
    print('Washed', dish)
conn.rpush('dishes', 'quit')
print('Washer is done')
```

The loop generates four messages containing a dish name, followed by a final message that says “quit.” It appends each message to a list called `dishes` in the Redis server, similar to appending to a Python list.

And as soon as the first dish is ready, `redis_dryer.py` does its work:

```
import redis
conn = redis.Redis()
print('Dryer is starting')
while True:
    msg = conn.blpop('dishes')
    if not msg:
        break
    val = msg[1].decode('utf-8')
    if val == 'quit':
        break
    print('Dried', val)
print('Dishes are dried')
```

This code waits for messages whose first token is “dishes” and prints that each one is dried. It obeys the quit message by ending the loop.

Start the dryer, and then the washer. Using the `&` at the end puts the first program in the background; it keeps running, but doesn't listen to the keyboard anymore. This works on Linux, OS X, and Windows, although you might see different output on the next line. In this case (OS X), it's some information about the background dryer process. Then, we start the washer process normally (in the foreground). You'll see the mingled output of the two processes:

```
$ python redis_dryer.py &
[2] 81691
Dryer is starting
$ python redis_washer.py
Washer is starting
Washed salad
Dried salad
Washed bread
```

Dried bread  
Washed entree  
Dried entree  
Washed dessert  
Washer is done  
Dried dessert  
Dishes are dried

[2] + Done                   python redis\_dryer.py

As soon as dish IDs started arriving at Redis from the washer process, our hard-working dryer process started pulling them back out. Each dish ID was a number, except the final sentinel value, the string 'quit'. When the dryer process read that quit dish ID, it quit, and some more background process information printed to the terminal (also system-dependent). You can use a sentinel (an otherwise invalid value) to indicate something special from the data stream itself—in this case, that we're done. Otherwise, we'd need to add a lot more program logic, such as the following:

Agreeing ahead of time on some maximum dish number, which would kind of be a sentinel anyway.

Doing some special out-of-band (not in the data stream) interprocess communication.

Timing out after some interval with no new data.

Let's make a few last changes:

Create multiple dryer processes.

Add a timeout to each dryer rather than looking for a sentinel.

The new redis\_dryer2.py:

```
def dryer():
    import redis
    import os
    import time
    conn = redis.Redis()
    pid = os.getpid()
    timeout = 20
    print('Dryer process %s is starting' % pid)
    while True:
        msg = conn.blpop('dishes', timeout)
        if not msg:
            break
        val = msg[1].decode('utf-8')
        if val == 'quit':
            break
        print('%s: dried %s' % (pid, val))
        time.sleep(0.1)
    print('Dryer process %s is done' % pid)
```

```
import multiprocessing
DRYERS=3
for num in range(DRYERS):
    p = multiprocessing.Process(target=dryer)
    p.start()
```

Start the dryer processes in the background, and then the washer process in the foreground:

```
$ python redis_dryer2.py &
```

Dryer process 44447 is starting  
Dryer process 44448 is starting  
Dryer process 44446 is starting  
\$ python redis\_washer.py  
Washer is starting  
Washed salad  
44447: dried salad  
Washed bread  
44448: dried bread  
Washed entree  
44446: dried entree  
Washed dessert  
Washer is done  
44447: dried dessert  
One dryer process reads the quit ID and quits:  
Dryer process 44448 is done  
After 20 seconds, the other dryer processes get a return value of None from their blpop calls, indicating that they've timed out. They say their last words and exit:  
Dryer process 44447 is done  
Dryer process 44446 is done  
After the last dryer subprocess quits, the main dryer program ends:  
[1]+ Done                   python redis\_dryer2.py

---

## **Topic 7: Networks**

**Possible Question: Discuss about Networks in patterns?**

**Possible Marks: 5 or 10 Marks**

### **Outcomes:**

You can build networking applications from some basic patterns.

The most common pattern is request-reply, also known as client-server. This pattern is synchronous: the client waits until the server responds. You've seen many examples of request-reply in this book. Your web browser is also a client, making an HTTP request to a web server, which returns a reply.

Another common pattern is push, or fanout: you send data to any available worker in a pool of processes. An example is a web server behind a load balancer.

The opposite of push is pull, or fanin: you accept data from one or more sources. An example would be a logger that takes text messages from multiple processes and writes them to a single log file.

One pattern is similar to radio or television broadcasting: publish-subscribe, or pub-sub. With this pattern, a publisher sends out data. In a simple pub-sub system, all subscribers would receive a copy. More often, subscribers can indicate that they're interested only in certain types of data (often called a topic), and the publisher will send just those. So, unlike the push pattern, more than one subscriber might receive a given piece of data. If there's no subscriber for a topic, the data is ignored.

---

## **Topic 8: The Publish**

**Possible Question: Explain detail about The Publish?**

**Possible Marks: 5 or 10 Marks**

### **Outcomes:**



Publish-subscribe is not a queue but a broadcast. One or more processes publish messages. Each subscriber process indicates what type of messages it would like to receive. A copy of each message is sent to each subscriber that matched its type. Thus, a given message might be processed once, more than once, or not at all. Each publisher is just broadcasting and doesn't know who—if anyone—is listening.

Redis

You can build a quick pub-sub system by using Redis. The publisher emits messages with a topic and a value, and subscribers say which topics they want to receive.

Here's the publisher, `redis_pub.py`:

```
import redis
import random

conn = redis.Redis()
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
for msg in range(10):
    cat = random.choice(cats)
    hat = random.choice(hats)
    print('Publish: %s wears a %s' % (cat, hat))
    conn.publish(cat, hat)
```

Each topic is a breed of cat, and the accompanying message is a type of hat.

Here's a single subscriber, `redis_sub.py`:

```
import redis
conn = redis.Redis()
topics = ['maine coon', 'persian']
sub = conn.psub()
sub.subscribe(topics)
for msg in sub.listen():
    if msg['type'] == 'message':
        cat = msg['channel']
        hat = msg['data']
        print('Subscribe: %s wears a %s' % (cat, hat))
```

The subscriber just shown wants all messages for cat types 'maine coon' and 'persian', and no others. The `listen()` method returns a dictionary. If its `type` is 'message', it was sent by the publisher and matches our criteria. The 'channel' key is the topic (cat), and the 'data' key contains the message (hat).

If you start the publisher first and no one is listening, it's like a mime falling in the forest (does he make a sound?), so start the subscriber first:

```
$ python redis_sub.py
```

Next, start the publisher. It will send 10 messages, and then quit:

```
$ python redis_pub.py
Publish: maine coon wears a stovepipe
Publish: norwegian forest wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: siamese wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
```

Publish: maine coon wears a bowler  
Publish: persian wears a bowler  
Publish: norwegian forest wears a bowler  
Publish: maine coon wears a stovepipe  
The subscriber cares about only two types of cat:

```
$ python redis_sub.py  
Subscribe: maine coon wears a stovepipe  
Subscribe: maine coon wears a bowler  
Subscribe: maine coon wears a bowler  
Subscribe: persian wears a bowler  
Subscribe: maine coon wears a stovepipe
```

We didn't tell the subscriber to quit, so it's still waiting for messages. If you restart the publisher, the subscriber will grab a few more messages and print them.

You can have as many subscribers (and publishers) as you want. If there's no subscriber for a message, it disappears from the Redis server. However, if there are subscribers, the messages stay in the server until all subscribers have retrieved them.

ZeroMQ has no central server, so each publisher writes to all subscribers. The publisher, `zmq_pub.py`, looks like this:

```
import zmq  
import random  
import time  
host = '*'  
port = 6789  
ctx = zmq.Context()  
pub = ctx.socket(zmq.PUB)  
pub.bind('tcp://%s:%s' % (host, port))  
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']  
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']  
time.sleep(1)  
for msg in range(10):  
    cat = random.choice(cats)  
    cat_bytes = cat.encode('utf-8')  
    hat = random.choice(hats)  
    hat_bytes = hat.encode('utf-8')  
    print('Publish: %s wears a %s' % (cat, hat))  
    pub.send_multipart([cat_bytes, hat_bytes])
```

Notice how this code uses UTF-8 encoding for the topic and value strings.

The file for the subscriber is `zmq_sub.py`:

```
import zmq  
host = '127.0.0.1'  
port = 6789  
ctx = zmq.Context()  
sub = ctx.socket(zmq.SUB)  
sub.connect('tcp://%s:%s' % (host, port))  
topics = ['maine coon', 'persian']
```

for topic in topics:

```
sub.setsockopt(zmq.SUBSCRIBE, topic.encode('utf-8'))
```

while True:

```
cat_bytes, hat_bytes = sub.recv_multipart()
```

```
cat = cat_bytes.decode('utf-8')
```

```
hat = hat_bytes.decode('utf-8')
```

```
print('Subscribe: %s wears a %s' % (cat, hat))
```

In this code, we subscribe to two different byte values: the two strings in topics, encoded as UTF-8.

#### **NOTE**

It seems a little backward, but if you want all topics, you need to subscribe to the empty bytestring b"; if you don't, you'll get nothing.

Notice that we call `send_multipart()` in the publisher and `recv_multipart()` in the subscriber. This makes it possible for us to send multipart messages, and use the first part as the topic. We could also send the topic and message as a single string or bytestring, but it seems cleaner to keep cats and hats separate.

Start the subscriber:

```
$ python zmq_sub.py
```

Start the publisher. It immediately sends 10 messages, and then quits:

```
$ python zmq_pub.py
```

```
Publish: norwegian forest wears a stovepipe
```

```
Publish: siamese wears a bowler
```

```
Publish: persian wears a stovepipe
```

```
Publish: norwegian forest wears a fedora
```

```
Publish: maine coon wears a tam-o-shanter
```

```
Publish: maine coon wears a stovepipe
```

```
Publish: persian wears a stovepipe
```

```
Publish: norwegian forest wears a fedora
```

```
Publish: norwegian forest wears a bowler
```

```
Publish: maine coon wears a bowler
```

The subscriber prints what it requested and received:

```
Subscribe: persian wears a stovepipe
```

```
Subscribe: maine coon wears a tam-o-shanter
```

```
Subscribe: maine coon wears a stovepipe
```

```
Subscribe: persian wears a stovepipe
```

```
Subscribe: maine coon wears a bowler
```

#### Other Pub-sub Tools

You might like to explore some of these other Python pub-sub links:

#### RabbitMQ

This is a well-known messaging broker, and `pika` is a Python API for it. See the `pika` documentation and a pub-sub tutorial.

[pypi.python.org](http://pypi.python.org)

Go to the upper-right corner of the search window and type pubsub to find Python packages like pypubsub.

pubsubhubbub

This mellifluous protocol enables subscribers to register callbacks with publishers.

---

## **Topic 9: TCP/IP**

**Possible Question: Discuss about TCP/IP?**

**Possible Marks: 5 or 10 Marks**

### **Outcomes:**

We've been walking through the networking house, taking for granted that whatever's in the basement works correctly. Now, let's actually visit the basement and look at the wires and pipes that keep everything running above ground.

The Internet is based on rules about how to make connections, exchange data, terminate connections, handle timeouts, and so on. These are called protocols, and they are arranged in layers. The purpose of layers is to allow innovation and alternative ways of doing things; you can do anything you want on one layer as long as you follow the conventions in dealing with the layers above and below you.

The very lowest layer governs aspects such as electrical signals; each higher layer builds on those below. In the middle, more or less, is the IP (Internet Protocol) layer, which specifies how network locations are addressed and how packets (chunks) of data flow. In the layer above that, two protocols describe how to move bytes between locations:

UDP (User Datagram Protocol)

This is used for short exchanges. A datagram is a tiny message sent in a single burst, like a note on a postcard.

TCP (Transmission Control Protocol)

This protocol is used for longer-lived connections. It sends streams of bytes and ensures that they arrive in order without duplication.

UDP messages are not acknowledged, so you're never sure if they arrive at their destination.

If you wanted to tell a joke over UDP:

Here's a UDP joke. Get it?

TCP sets up a secret handshake between sender and receiver to ensure a good connection. A

TCP joke would start like this:

Do you want to hear a TCP joke?

Yes, I want to hear a TCP joke.

Okay, I'll tell you a TCP joke.

Okay, I'll hear a TCP joke.

Okay, I'll send you a TCP joke now.

Okay, I'll receive the TCP joke now.

... (and so on)

Your local machine always has the IP address 127.0.0.1 and the name localhost. You might see this called the loopback interface. If it's connected to the Internet, your machine will also have a public IP. If you're just using a home computer, it's behind equipment such as a cable modem or router. You can run Internet protocols even between processes on the same machine.

Most of the Internet with which we interact—the Web, database servers, and so on—is based on the TCP protocol running atop the IP protocol; for brevity, TCP/IP. Let's first look at some basic Internet services. After that, we'll explore general networking patterns.

---

## Topic 10: Sockets

### Possible Question: Discuss about Sockets?

### Possible Marks: 5 or 10 Marks

#### Outcomes:

We've saved this topic until now because you don't need to know all the low-level details to use the higher levels of the Internet. But if you like to know how things work, this is for you.

The lowest level of network programming uses a *socket*, borrowed from the C language and the Unix operating system. Socket-level coding is tedious. You'll have more fun using something like ZeroMQ, but it's useful to see what lies beneath. For instance, messages about sockets often turn up when networking errors take place.

Let's write a very simple client-server exchange. The client sends a string in a UDP datagram to a server, and the server returns a packet of data containing a string. The server needs to listen at a particular address and port—like a post office and a post office box. The client needs to know these two values to deliver its message, and receive any reply.

In the following client and server code, address is a tuple of (*address*, *port*). The address is a string, which can be a name or an *IP address*. When your programs are just talking to one another on the same machine, you can use the name 'localhost' or the equivalent address '127.0.0.1'.

First, let's send a little data from one process to another and return a little data back to the originator. The first program is the client and the second is the server. In each program, we'll print the time and open a socket. The server will listen for connections to its socket, and the client will write to its socket, which transmits a message to the server.

Here's the first program, *udp\_server.py*:

```
from datetime import datetime
import socket

server_address = ('localhost', 6789)
max_size = 4096

print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(server_address)

data, client = server.recvfrom(max_size)

print('At', datetime.now(), client, 'said', data)
server.sendto(b'Are you talking to me?', client)
server.close()
```

The server has to set up networking through two methods imported from the socket package. The first method, `socket.socket`, creates a socket, and the second, `bind`, *binds* to it (listens to any data arriving at that IP address and port). `AF_INET` means we'll create an Internet (IP) socket. (There's another type for *Unix domain sockets*, but those work only on the local machine.) `SOCK_DGRAM` means we'll send and receive datagrams—in other words, we'll use UDP.

At this point, the server sits and waits for a datagram to come in (`recvfrom`). When one arrives, the server wakes up and gets both the data and information about the client. The client variable

contains the address and port combination needed to reach the client. The server ends by sending a reply and closing its connection.

Let's take a look at *udp\_client.py*:

```
import socket
from datetime import datetime

server_address = ('localhost', 6789)
max_size = 4096

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.sendto(b'Hey!', server_address)
data, server = client.recvfrom(max_size)
print('At', datetime.now(), server, 'said', data)
client.close()
```

The client has most of the same methods as the server (with the exception of `bind()`). The client sends and then receives, whereas the server receives first.

Start the server first, in its own window. It will print its greeting and then wait with an eerie calm until a client sends it some data:

```
$ python udp_server.py
```

```
Starting the server at 2014-02-05 21:17:41.945649
```

```
Waiting for a client to call.
```

Next, start the client in another window. It will print its greeting, send data to the server, print the reply, and then exit:

```
$ python udp_client.py
```

```
Starting the client at 2014-02-05 21:24:56.509682
```

```
At 2014-02-05 21:24:56.518670 ('127.0.0.1', 6789) said b'Are you talking to me?'
```

```
Finally, the server will print something like this, and then exit:
```

```
At 2014-02-05 21:24:56.518473 ('127.0.0.1', 56267) said b'Hey!'
```

The client needed to know the server's address and port number but didn't need to specify a port number for itself. That was automatically assigned by the system—in this case, it was 56267.

## NOTE

UDP sends data in single chunks. It does not guarantee delivery. If you send multiple messages via UDP, they can arrive out of order, or not at all. It's fast, light, connectionless, and unreliable. Which brings us to TCP (Transmission Control Protocol). TCP is used for longer-lived connections, such as the Web. TCP delivers data in the order in which you send it. If there were any problems, it tries to send it again. Let's shoot a few packets from client to server and back with TCP.

*tcp\_client.py* acts like the previous UDP client, sending only one string to the server, but there are small differences in the socket calls, illustrated here:

```
import socket
from datetime import datetime

address = ('localhost', 6789)
max_size = 1000

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
client.connect(address)
client.sendall(b'Hey!')
data = client.recv(max_size)
print('At', datetime.now(), 'someone replied', data)
client.close()
```

We've replaced `SOCK_DGRAM` with `SOCK_STREAM` to get the streaming protocol, TCP. We also added a `connect()` call to set up the stream. We didn't need that for UDP because each datagram was on its own in the wild, wooly Internet.

*tcp\_server.py* also differs from its UDP cousin:

```
from datetime import datetime
import socket
```

```
address = ('localhost', 6789)
max_size = 1000
```

```
print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(address)
server.listen(5)
```

```
client, addr = server.accept()
data = client.recv(max_size)
```

```
print('At', datetime.now(), client, 'said', data)
client.sendall(b'Are you talking to me?')
client.close()
server.close()
```

`server.listen(5)` is configured to queue up to five client connections before refusing new ones. `server.accept()` gets the first available message as it arrives. The `client.recv(1000)` sets a maximum acceptable message length of 1,000 bytes.

As you did earlier, start the server and then the client, and watch the fun. First, the server:

```
$ python tcp_server.py
```

```
Starting the server at 2014-02-06 22:45:13.306971
```

```
Waiting for a client to call.
```

```
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
  proto=0> said b'Hey!'
```

Now, start the client. It will send its message to the server, receive a response, and then exit:

```
$ python tcp_client.py
```

```
Starting the client at 2014-02-06 22:45:16.038642
```

```
At 2014-02-06 22:45:16.049078 someone replied b'Are you talking to me?'
```

The server collects the message, prints it, responds, and then quits:

```
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
  proto=0> said b'Hey!'
```

Notice that the TCP server called `client.sendall()` to respond, and the earlier UDP server called `client.sendto()`. TCP maintains the client-server connection across multiple socket calls and remembers the client's IP address.

This didn't look so bad, but if you try to write anything more complex, you'll see how low-level sockets really are. Here are some of the complications with which you need to cope:

- UDP sends messages, but their size is limited, and they're not guaranteed to reach their destination.
  - TCP sends streams of bytes, not messages. You don't know how many bytes the system will send or receive with each call.
  - To exchange entire messages with TCP, you need some extra information to reassemble the full message from its segments: a fixed message size (bytes), or the size of the full message, or some delimiting character.
  - Because messages are bytes, not Unicode text strings, you need to use the Python bytes type.
- 

## **Topic 11: Zero MQ**

### **Possible Question: Discuss about Zero MQ?**

### **Possible Marks: 5 or 10 Marks**

#### **Outcomes:**

Sometimes described as sockets on steroids, ZeroMQ sockets do the things that you sort of expected plain sockets to do:

- Exchange entire messages
- Retry connections
- Buffer data to preserve it when the timing between senders and receivers doesn't line up

The online guide is well written and witty, and it presents the best description of networking patterns that I've seen. The printed version (*ZeroMQ: Messaging for Many Applications*, by Pieter Hintjens, from that animal house, O'Reilly) has that good code smell and a big fish on the cover, rather than the other way around. All the examples in the printed guide are in the C language, but the online version lets you pick from multiple languages for each code example. The Python examples are also viewable

ZeroMQ is like a Lego set, and we all know that you can build an amazing variety of things from a few Lego shapes. In this case, you construct networks from a few socket types and patterns. The basic "Lego pieces" presented in the following list are the ZeroMQ socket types, which by some twist of fate look like the network patterns we've already discussed:

- REQ (synchronous request)
- REP (synchronous reply)
- DEALER (asynchronous request)
- ROUTER (asynchronous reply)
- PUB (publish)
- SUB (subscribe)
- PUSH (fanout)
- PULL (fanin)

To try these yourself, you'll need to install the Python ZeroMQ library by typing this command:  
`$ pip install pyzmq`

The simplest pattern is a single request-reply pair. This is synchronous: one socket makes a request and then the other replies. First, the code for the reply (server), `zmq_server.py`:

```
import zmq
```



```

host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
while True:
    # Wait for next request from client
    request_bytes = server.recv()
    request_str = request_bytes.decode('utf-8')
    print("That voice in my head says: %s" % request_str)
    reply_str = "Stop saying: %s" % request_str
    reply_bytes = bytes(reply_str, 'utf-8')
    server.send(reply_bytes)

```

We create a Context object: this is a ZeroMQ object that maintains state. Then, we make a ZeroMQ socket of type REP (for REPLY). We call bind() to make it listen on a particular IP address and port. Notice that they're specified in a string such as 'tcp://localhost:6789' rather than a tuple, as in the plain socket examples.

This example keeps receiving requests from a sender and sending a response. The messages can be very long—ZeroMQ takes care of the details.

Following is the code for the corresponding request (client), zmq\_client.py. Its type is REQ (for REQuest), and it calls connect() rather than bind().

```

import zmq

host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
for num in range(1, 6):
    request_str = "message #%s" % num
    request_bytes = request_str.encode('utf-8')
    client.send(request_bytes)
    reply_bytes = client.recv()
    reply_str = reply_bytes.decode('utf-8')
    print("Sent %s, received %s" % (request_str, reply_str))

```

Now it's time to start them. One interesting difference from the plain socket examples is that you can start the server and client in either order. Go ahead and start the server in one window in the background:

```
$ python zmq_server.py &
```

Start the client in the same window:

```
$ python zmq_client.py
```

You'll see these alternating output lines from the client and server:

```

That voice in my head says 'message #1'
Sent 'message #1', received 'Stop saying message #1'
That voice in my head says 'message #2'
Sent 'message #2', received 'Stop saying message #2'
That voice in my head says 'message #3'

```

Sent 'message #3', received 'Stop saying message #3'

That voice in my head says 'message #4'

Sent 'message #4', received 'Stop saying message #4'

That voice in my head says 'message #5'

Sent 'message #5', received 'Stop saying message #5'

Our client ends after sending its fifth message, but we didn't tell the server to quit, so it sits by the phone, waiting for another message. If you run the client again, it will print the same five lines, and the server will print its five also. If you don't kill the `zmq_server.py` process and try to run another one, Python will complain that the address is already in use:

```
$ python zmq_server.py
```

```
[2] 356
```

```
Traceback (most recent call last):
```

```
File "zmq_server.py", line 7, in <module>
```

```
    server.bind("tcp://%s:%s" % (host, port))
```

```
File "socket.pyx", line 444, in zmq.backend.cython.socket.Socket.bind
```

```
(zmq/backend/cython/socket.c:4076)
```

```
File "checkrc.pxd", line 21, in zmq.backend.cython.checkrc._check_rc
```

```
(zmq/backend/cython/socket.c:6032)
```

```
zmq.error.ZMQError: Address already in use</pre>
```

The messages need to be sent as byte strings, so we encoded our example's text strings in UTF-8 format. You can send any kind of message you like, as long as you convert it to bytes. We used simple text strings as the source of our messages, so `encode()` and `decode()` were enough to convert to and from byte strings. If your messages have other data types, you can use a library such as `MessagePack`.

Even this basic REQ-REP pattern allows for some fancy communication patterns, because any number of REQ clients can connect() to a single REP server. The server handles requests one at a time, synchronously, but doesn't drop other requests that are arriving in the meantime. ZeroMQ buffers messages, up to some specified limit, until they can get through; that's where it earns the Q in its name. The Q stands for Queue, the M stands for Message, and the Zero means there doesn't need to be any broker.

Although ZeroMQ doesn't impose any central brokers (intermediaries), you can build them where needed. For example, use DEALER and ROUTER sockets to connect multiple sources and/or destinations asynchronously.

Multiple REQ sockets connect to a single ROUTER, which passes each request to a DEALER, which then contacts any REP sockets that have connected to it (Figure 11-1). This is similar to a bunch of browsers contacting a proxy server in front of a web server farm. It lets you add multiple clients and servers as needed.

The REQ sockets connect only to the ROUTER socket; the DEALER connects to the multiple REP sockets behind it. ZeroMQ takes care of the nasty details, ensuring that the requests are load balanced and that the replies go back to the right place.

Another networking pattern called the ventilator uses PUSH sockets to farm out asynchronous tasks, and PULL sockets to gather the results.

The last notable feature of ZeroMQ is that it scales up and down, just by changing the connection type of the socket when it's created:

- tcp between processes, on one or more machines
- ipc between processes on one machine

- inproc between threads in a single process

That last one, inproc, is a way to pass data between threads without locks, and an alternative to the threading example in “Threads”.

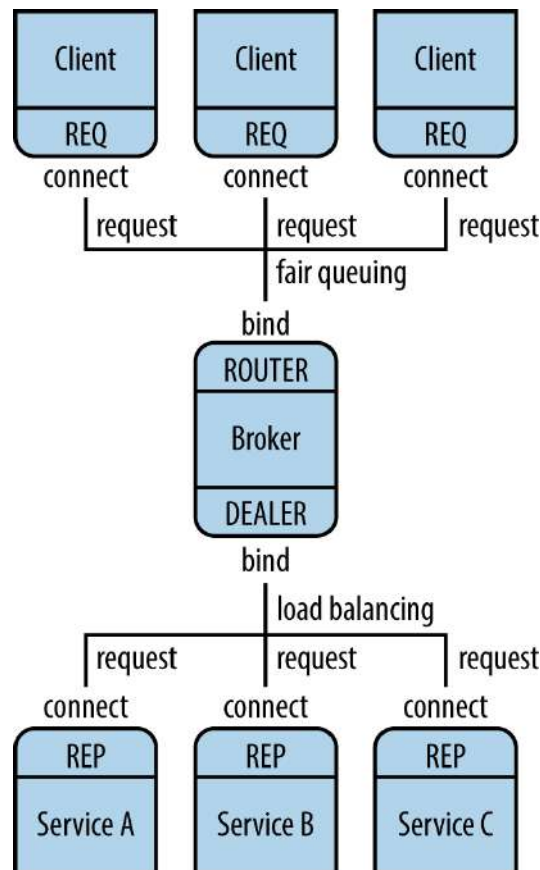


Figure. Using a broker to connect multiple clients and services  
After using ZeroMQ, you might never want to write raw socket code again.

---

## Topic 12: Internet Services

**Possible Question: Explain about Internet Services?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Python has an extensive networking toolset. In the following sections, we’ll look at ways to automate some of the most popular Internet services. The official, comprehensive documentation is available online.

### Domain Name System

Computers have numeric IP addresses such as 85.2.101.94, but we remember names better than numbers. The Domain Name System (DNS) is a critical Internet service that converts IP addresses to and from names via a distributed database. Whenever you’re using a web browser and suddenly see a message like “looking up host,” you’ve probably lost your Internet connection, and your first clue is a DNS failure.

Some DNS functions are found in the low-level socket module. `gethostbyname()` returns the IP address for a domain name, and the extended edition `gethostbyname_ex()` returns the name, a list of alternative names, and a list of addresses:

```
>>> import socket
>>> socket.gethostbyname('www.crappytaxidermy.com')
'66.6.44.4'
```

```
>>> socket.gethostbyname_ex('www.crappytaxidermy.com')
```

```
('crappytaxidermy.com', ['www.crappytaxidermy.com'], ['66.6.44.4'])
```

The `getaddrinfo()` method looks up the IP address, but it also returns enough information to create a socket to connect to it:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80)
```

```
[(2, 2, 17, "", ('66.6.44.4', 80)), (2, 1, 6, "", ('66.6.44.4', 80))]
```

The preceding call returned two tuples, the first for UDP, and the second for TCP (the 6 in the 2, 1, 6 is the value for TCP).

You can ask for TCP or UDP information only:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80, socket.AF_INET,  
socket.SOCK_STREAM)
```

```
[(2, 1, 6, "", ('66.6.44.4', 80))]
```

Some TCP and UDP port numbers are reserved for certain services by IANA, and are associated with service names. For example, HTTP is named `http` and is assigned TCP port 80.

These functions convert between service names and port numbers:

```
>>> import socket
```

```
>>> socket.getservbyname('http')
```

```
80
```

```
>>> socket.getservbyport(80)
```

```
'http'
```

### Python Email Modules

The standard library contains these email modules:

- `smtplib` for sending email messages via Simple Mail Transfer Protocol (SMTP)
- `email` for creating and parsing email messages
- `poplib` for reading email via Post Office Protocol 3 (POP3)
- `imaplib` for reading email via Internet Message Access Protocol (IMAP)

The official documentation contains sample code for all of these libraries.

If you want to write your own Python SMTP server, try `smtpd`.

A pure-python SMTP server called `Lamson` allows you to store messages in databases, and you can even block spam.

### Other protocols

Using the standard `ftplib` module, you can push bytes around by using the File Transfer Protocol (FTP). Although it's an old protocol, FTP still performs very well.

You've seen many of these modules in various places in this book, but also try the documentation for standard library support of Internet protocols.

---

## Topic 13: Web Services and APIs

**Possible Question: Explain about Web Services and APIs?**

**Possible Marks: 5 or 10 Marks**

### Outcomes:

Information providers always have a website, but those are targeted for human eyes, not automation. If data is published only on a website, anyone who wants to access and structure the data needs to write scrapers (as shown in “Crawl and Scrape”), and rewrite them each time

a page format changes. This is usually tedious. In contrast, if a website offers an API to its data, the data becomes directly available to client programs. APIs change less often than web page layouts, so client rewrites are less common. A fast, clean data pipeline also makes it easier to build mashups—combinations that might not have been foreseen but can be useful and even profitable.

In many ways, the easiest API is a web interface, but one that provides data in a structured format such as JSON or XML rather than plain text or HTML. The API might be minimal or a full-fledged RESTful API (defined in “Web APIs and Representational State Transfer”), but it provides another outlet for those restless bytes.

At the very beginning of this book, you can see a web API: it picks up the most popular videos from YouTube. This next example might make more sense now that you’ve read about web requests, JSON, dictionaries, lists, and slices:

```
import requests
url = "https://gdata.youtube.com/feeds/api/standardfeeds/top_rated?alt=json"
response = requests.get(url)
data = response.json()
for video in data['feed']['entry'][0:6]:
    print(video['title']['$t'])
```

APIs are especially useful for mining well-known social media sites such as Twitter, Facebook, and LinkedIn. All these sites provide APIs that are free to use, but they require you to register and get a key (a long-generated text string, sometimes also known as a token) to use when connecting. The key lets a site determine who’s accessing its data. It can also serve as a way to limit request traffic to servers. The YouTube example you just looked at did not require an API key for searching, but it would if you made calls that updated data at YouTube.

Here are some interesting service APIs:

- New York Times
- YouTube
- Twitter
- Facebook
- Weather Underground
- Marvel Comics

---

## **Topic 14: Remote Processing**

**Possible Question: Describe about Remote Processing?**

**Possible Marks: 5 or 10 Marks**

**Outcomes:**

Most of the examples in this book have demonstrated how to call Python code on the same machine, and usually in the same process. Thanks to Python’s expressiveness, you can also call code on other machines as though they were local. In advanced settings, if you run out of space on your single machine, you can expand beyond it. A network of machines gives you access to more processes and/or threads.

### **Remote Procedure Calls**

Remote Procedure Calls (RPCs) look like normal functions but execute on remote machines across a network. Instead of calling a RESTful API with arguments encoded in the URL or request body, you call an RPC function on your own machine. Here’s what happens under the hood of the RPC client:

1. It converts your function arguments into bytes (sometimes this is called marshalling, or serializing, or just encoding).
2. It sends the encoded bytes to the remote machine.

And here's what happens on the remote machine:

1. It receives the encoded request bytes.
2. After receiving the bytes, the RPC client decodes the bytes back to the original data structures (or equivalent ones, if the hardware and software differ between the two machines).
3. The client then finds and calls the local function with the decoded data.
4. Next, it encodes the function results.
5. Last, the client sends the encoded bytes back to the caller.

And finally, the machine that started it all decodes the bytes to return values.

RPC is a popular technique, and people have implemented it in many ways. On the server side, you start a server program, connect it with some byte transport and encoding/decoding method, define some service functions, and light up your RPC is open for business sign. The client connects to the server and calls one of its functions via RPC.

The standard library includes one RPC implementation that uses XML as the exchange format: `xmlrpc`. You define and register functions on the server, and the client calls them as though they were imported. First, let's explore the file `xmlrpc_server.py`:

```
from xmlrpc.server import SimpleXMLRPCServer
def double(num):
    return num * 2
```

```
server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(double, "double")
server.serve_forever()
```

The function we're providing on the server is called `double()`. It expects a number as an argument and returns the value of that number times two. The server starts up on an address and port. We need to register the function to make it available to clients via RPC. Finally, start serving and carry on.

Now, you guessed it, `xmlrpc_client.py`:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
num = 7
result = proxy.double(num)
print("Double %s is %s" % (num, result))
```

The client connects to the server by using `ServerProxy()`. Then, it calls the function `proxy.double()`. Where did that come from? It was created dynamically by the server. The RPC machinery magically hooks this function name into a call to the remote server.

Give it a try—start the server and then run the client:

```
$ python xmlrpc_server.py
```

Next, run the client:

```
$ python xmlrpc_client.py
```

```
Double 7 is 14
```

The server then prints the following:

```
127.0.0.1 - - [13/Feb/2014 20:16:23] "POST / HTTP/1.1" 200 -
```

Popular transport methods are HTTP and ZeroMQ. Common encodings besides XML include JSON, Protocol Buffers, and MessagePack. There are many Python packages for JSON-based RPC, but many of them either don't support Python 3 or seem a bit tangled. Let's look at something different: MessagePack's own Python RPC implementation. Here's how to install it:

```
$ pip install msgpack-rpc-python
```

This will also install tornado, a Python event-based web server that this library uses as a transport. As usual, the server comes first (msgpack\_server.py):

```
from msgpackrpc import Server, Address
class Services():
    def double(self, num):
        return num * 2
```

```
server = Server(Services())
server.listen(Address("localhost", 6789))
server.start()
```

The Services class exposes its methods as RPC services. Go ahead and start the client, msgpack\_client.py:

```
from msgpackrpc import Client, Address

client = Client(Address("localhost", 6789))
num = 8
result = client.call('double', num)
print("Double %s is %s" % (num, result))
```

To run these, follow the usual drill: start the server, start the client, see the results:

```
$ python msgpack_server.py
$ python msgpack_client.py
Double 8 is 16
```

---

## **Topic 15: Big Fat Data and Map Reduce**

### **Possible Question: Discuss about Big Fat Data and Map Reduce?**

#### **Possible Marks: 5 or 10 Marks**

#### **Outcomes:**

As Google and other Internet companies grew, they found that traditional computing solutions didn't scale. Software that worked for single machines, or even a few dozen, could not keep up with thousands.

Disk storage for databases and files involved too much seeking, which requires mechanical movement of disk heads. (Think of a vinyl record, and the time it takes to move the needle from one track to another manually. And think of the screeching sound it makes when you drop it too hard, not to mention the sounds made by the record's owner.) But you could stream consecutive segments of the disk more quickly.

Developers found that it was faster to distribute and analyze data on many networked machines than on individual ones. They could use algorithms that sounded simplistic, but actually worked better overall with massively distributed data. One of these is MapReduce, which spreads a calculation across many machines and then gathers the results. It's similar to working with queues.

After Google published its results in a paper, Yahoo followed with an open source Java-based package named Hadoop (named after the toy stuffed elephant of the lead programmer's son). The phrase big data applies here. Often it just means "data too big to fit on my machine": data that exceeds the disk, memory, CPU time, or all of the above. To some organizations, if big data is mentioned somewhere in a question, the answer is always Hadoop. Hadoop copies data among machines, running them through map and reduce programs, and saving the results on disk at each step.

This batch process can be slow. A quicker method called Hadoop streaming works like Unix pipes, streaming the data through programs without requiring disk writes at each step. You can write Hadoop streaming programs in any language, including Python.

Many Python modules have been written for Hadoop, and some are discussed in the blog post "A Guide to Python Frameworks for Hadoop". The Spotify company, known for streaming music, open sourced its Python component for Hadoop streaming, Luigi. The Python 3 port is still incomplete.

A rival named Spark was designed to run ten to a hundred times faster than Hadoop. It can read and process any Hadoop data source and format. Spark includes APIs for Python and other languages. You can find the installation documents online.

Another alternative to Hadoop is Disco, which uses Python for MapReduce processing and Erlang for communication. Alas, you can't install it with pip; see the documentation.

See Appendix C for related examples of parallel programming, in which a large structured calculation is distributed among many machines.

---

## **Topic 16: Working in the Clouds**

**Possible Question: Explain detail about Working in the Clouds?**

**Possible Marks: 5 or 10 Marks**

### **Outcomes:**

Not so long ago, you would buy your own servers, bolt them into racks in data centers, and install layers of software on them: operating systems, device drivers, file systems, databases, web servers, email servers, name servers, load balancers, monitors, and more. Any initial novelty wore off as you tried to keep multiple systems alive and responsive. And you worried constantly about security.

Many hosting services offered to take care of your servers for a fee, but you still leased the physical devices and had to pay for your peak load configuration at all times.

With more individual machines, failures are no longer infrequent: they're very common. You need to scale services horizontally and store data redundantly. You can't assume that the network operates like a single machine. The eight fallacies of distributed computing, according to Peter Deutsch, are as follows:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.



You can try to build these complex distributed systems, but it's a lot of work, and a different toolset is needed. To borrow an analogy, when you have a handful of servers, you treat them like pets—you give them names, know their personalities, and nurse them back to health when needed. But at scale, you treat servers more like livestock: they look alike, have numbers, and are just replaced if they have any problems.

Instead of building, you can rent servers in the cloud. By adopting this model, maintenance is someone else's problem, and you can concentrate on your service, or blog, or whatever you want to show the world. Using web dashboards and APIs, you can spin up servers with whatever configuration you need, quickly and easily—they're elastic. You can monitor their status, and be alerted if some metric exceeds a given threshold. Clouds are currently a pretty hot topic, and corporate spending on cloud components has spiked.

---

## Model Questions

### Objective

1. Which Python library runs a function as thread?

- A. None                      B. `_threading`                      C. `threading`                      D. `thread`

**Answer: D**

2. How does `run()` method is invoked?

- A. By `Thread.run()`    B. By `Thread.start()`    C. By `Thread.create()`                      D. None

**Answer: B**

3. How to terminate a blocking thread?

- A. `thread.stop()` & `thread.wait()`                      B. `thread.stop()`                      C. `thread.terminate()`    D. None

**Answer: A**

4. Which method is used to identify a thread?

- A. `getThread()`                      B. None                      C. `getName()`                      D. `get_ident()`

**Answer: C**

5. What are the libraries in Python that support threads?

- A. `_threading`                      B. `th`                      C. None                      D. `thread`

**Answer: D**

6. How does global value mutation used for thread-safety?

- A. via GIL (Global Interpreter Lock)                      B. via Locking                      C. via Mutex    D. None

**Answer: A**

7. What is the method to retrieve the list of all active threads?

- A. `getList()`                      B. `getThreads()`                      C. `enumerate()`                      D. `threads()`

**Answer: C**

8. Which thread method is used to wait until it terminates?

- A. `wait for thread()`                      B. `join()`                      C. None                      D. `wait()`

**Answer: B**

9. Multithreading is also called as \_\_\_\_\_

- A. Concurrency                      B. Simultaneity                      C. Crosscurrent    D. Recurrent

**Answer: A**

10. A single sequential flow of control within a program is \_\_\_\_\_

- A. Process                      B. Task                      C. Thread                      D. Structure

**Answer: C**

### Subjective

1. Describe about Concurrency?

2. Explain about Queues?
3. Explain about Processes & Threads?
4. Describe about Green Threads and event?
5. Discuss about twisted & Redis?
6. Describe about Networks Patterns?
7. Explain about The Publish-Subscribe Model?
8. Explain in detail about TCP/IP & Sockets?
9. Explain about Zero MQ?
10. Explain about Internet Services & Web Services and APIs?
11. Explain about Big Fat Data and Map Reduce?
12. Discuss about Working in the Clouds?

---

**Unit – V – END**

**Signature of the staff  
with date**