Dr.Umayal Ramanathan College for Women, Karaikudi.

(Accredited with B+ Grade by NAAC)

Affiliated to Alagappa University

Run by Dr.Alagappa Chettiar Educational Trust

Department of Information Technogy

Subject Name: **OPEN SOURCE SOFTWARE**

Subject code: **7BIT4C1**

II YEAR – IV SEMESTER

COURSE CODE: 7BIT4C1

CORE COURSE -VII-OPEN SOURCE SOFTWARE

Unit - I INTRODUCTION

Introduction to Open sources – Need of Open Sources –Advantages of Open Sources– Application of OpenSources. Open source operating systems: LINUX:Introduction– General Overview–Kernel Mode and usermode–Process–Advanced Concepts–Scheduling – Personalities – Cloning – Signals – Development with Linux. .

Unit – II OPEN SOURCE DATABASE

MySQL: Introduction Setting up account Starting, terminating and writing your ow nSQL programs –Record selection Technology– Working with strings – Date and Time– Sorting Query Results –GeneratingSummary – Working with metadata –Usingsequences – MySQL and Web.

Unit – III OPEN SOURCE PROGRAMMING LANGUAGES

PHP: Introduction – Programming in web environment – variables – constants– data;types – operators –Statements– Functions– Arrays – OOP – String Manipulation and regular expression –File handling and datastorage – PHP and SQL database – PHP and LDAP – PHP Connectivity –Sending and receiving E-mails –Debugging and error handling – Security – Templates.

Unit - IV PYTHON

Syntax and Style – Python Objects – Numbers – Sequences – Strings – Lists and Tuples – Dictionaries –Conditionals and Loops – Files – Input and Output – Errors and Exceptions – Functions – Modules –Classes andOOP – Execution Environment.

Unit – V PERL

Perl backgrounder – Perl overview– Perl parsing rules – Variables and Data – Statements and Controlstructures – Subroutines, Packages, and Modules- Working with Files –Data Manipulation.

Text Books:

1. Remy Card, Eric Dumas and Frank Mevel, “The Linux Kernel Book”, Wiley

Publications, 2003

2. Steve Suchring, “MySQL Bible”, John Wiley, 2002

Books for Reference:

1. Rasmus Lerdorf and Levin Tatroe, “Programming PHP”, O’Reilly, 2002
2. Wesley J. Chun, “Core Phython Programming”, Prentice Hall, 2001
3. Martin C. Brown, “Perl: The Complete Reference”, 2nd Edition, Tata McGraw-Hill Publishing Company Limited, Indian Reprint 2009.
4. Steven Holzner, “PHP: The Complete Reference”, 2nd Edition, Tata McGraw-Hill Publishing Company Limited, Indian Reprint 2009.
5. Vikram Vaswani, “MYSQL: The Complete Reference”, 2nd Edition, Tata McGraw-Hill Publishing CompanyLimited, Indian Reprint 2009.

	Unit - I	
S.NO	**CONTENT**	**PAGE NO**
1	**Introduction to open source**	**1**
2	**Need of open sources**	**1-2**
3	**Advantages of open sources**	**2**
4	**Application of open sources**	**3**
5	**LINUX - Introduction**	**3-4**
6	**Kernel mode and user mode**	**4-5**
7	**process**	**5-8**
8	**Process scheduling**	**9**
9	**personalities**	**9**

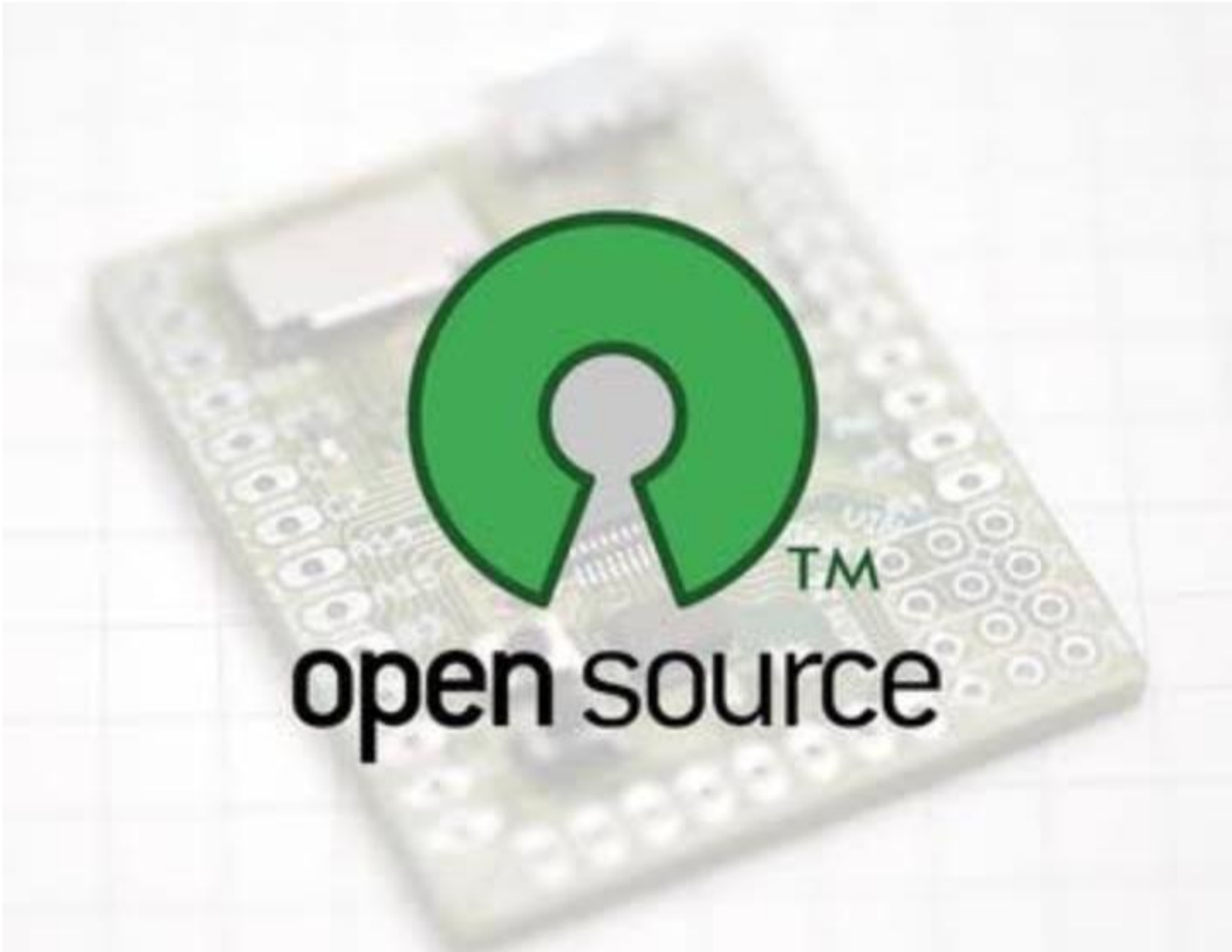
	Unit - II	
S.NO	**CONTENTS**	**PAGE NO**
1	**MYSQL Introduction**	**10**
2	**Setting up account**	**10-12**
3	**Starting , terminating and writing your own SQL programs**	**12-13**
4	**Record Selection Technology**	**13-19**
5	**Working with Strings**	**19-23**

6	Date and Time	23-27
7	Sorting Query Results	28-29
8	Generating summary	29-30
9	Working with Metadata	30-31
10	Using Sequences	31-32
11	MYSQL and web	32-33

	Unit - III	
S.NO	CONTENTS	PAGE NO
1	PHP – Introduction	34
2	PHP variables	35
3	PHP – Constants	36
4	Datatypes	37-39
5	Operators	39-43
6	Statements	43-46
7	Arrays	46-47
8	Functions	47-49
9	Object Oriented Programming(oops)	49-54
10	String manipulation	55-56
11	Regular expression	55-58
12	File handling and data storage	
13	PHP Connectivity	
14	Sending and receiving emails	

UNIT - IV		
S.NO	CONTENTS	PAGE NO
1	Python Introduction	59
2	Python syntax	59-64

3	Python numbers	64-65
4	Strings	65-68
5	Python list	68-70
6	Python Tuples	70-74
7	Python dictionaries	74-77
8	Python conditions and loops	77-88
9	Input and output	89-93
10	Errors and exceptions	93-98
11	Python functions	98-104
12	Python modules	104-107
13	Classes and oops	107-113
	Unit - v	
S.NO	CONTENTS	PAGE NO
1	PERL Introduction	114
2	PERL overview	115-119
3	PERL Parsing rules	116-121
4	Variables and data	121-129
5	Statements and control structures	129-140
6	Subroutines , packages and modules	140-144
7	Working with files	145-150


TM
open source

OPEN SOURCE SOFTWARE

Unit-1

Course outcome:To expose students to free open source software environment and introduce them to use open source packages.

1. Definition of Open Source:

Open source refers to any program whose source code is made available for use or modification as users or other developers see fit. Open source software is usually developed as a public collaboration and made freely available.
(Or)
Open source software refers to applications developed in which the user can access and alter the "source" code itself.

1.1 Need of Open Sources

Few reasons why you need an Open Source Strategy are:

1. **Reduce dependency on closed source vendors.** Stop being dragged through constant product upgrades that you are forced to do to stay on a supported version of the product.

2. **Your annual budget does not keep up with increases in software maintenance costs and increased costs of employee health care.** Your budget remains flat, you bought five new tools last year with new annual costs in the range of 18-20% of the original purchase price for "gold support", and your employees' health care costs shot up 25% again.

3. **More access to tools.** You can get your hands a variety of development and testing tools, project and portfolio management tools, network monitoring, security, content management, etc. without having to ask the boss man for a few hundred thousand green backs.

4. **Try before you buy.** Are you getting ready to invest in SOA, BPM, or ECM? Why not do a prototype without spending huge sums of money? First of all, it allows you to get familiar with the tools so you can be educated when you go through the vendor evaluation process. Second of all, you might find that the tool can do the job and you don't need to lock yourself in to another vendor.

5. **Great support and a 24/7 online community that responds quickly.** Despite the myths that you can't get support for open source software, the leading communities provide support far superior to most closed source vendors. Most communities have a great knowledgebase or wiki for self service support. You can also post a question and one of the hundreds of community members throughout the world will most likely respond in minutes. Make sure you chose software with strong community backing.

6. **Access to source code and the ability to customize if you desire.** You can see the code, change the code, and even submit your enhancements and/or fixes back to the community to be peer reviewed and possibly added to the next build.

7. **Great negotiating power when dealing with closed source vendors.** Tired of vendors pushing you around because you don't have options? I wonder if companies like Microsoft would be more willing to be flexible with their pricing if you have 20 desktops running Ubuntu as an alternative desktop pilot initiative.

8. **Feature set is not bloated and is driven by collaboration amongst the community.** Tired of products that consume huge amounts of memory and CPU power for the 2000 eye candy features that you will never use? With open source software, most features are driven by community demand. Closed vendors have to create one more feature then their competitors to get the edge in the marketplace.

9. **Bug fixes are implemented faster then closed source vendors.** Actually, many bugs are fixed by the community before they are even reported by the users.

1.3 Advantages of using Open Source

Below are some of the advantages that open source offers:

1. Core software is free
If you're just getting started in online business, cost can be a major factor. Using Open Source software can really cut down on your initial capital outlay. It's also my firm belief that the Open Source community has helped to rein in prices on commercial software over the years.
2. Evolving software
As mentioned, some Open Source software projects can have huge communities of programmers involved, allowing for the rapid implementation of new features and security fixes. The communities of users and programmers are also invaluable resources for asking questions relating to troubleshooting and suggesting enhancements.
3. Encourages hands on
When you're short on cash, you are more than likely to want to make modifications to software yourself. I'm no programmer, but the use of Open Source software has encouraged me to go beyond the user interface; to dig into code to try and understand what it does and to make minor edits. As a business owner, it doesn't hurt to understand a little of the voodoo that goes on behind the scenes in the software you use on your site.
4. Not tied to a single vendor

If you purchase a commercial application, you can then become reliant on a single company to solve your problems and maintain the software - which can also be very expensive. Some commercial software companies may only provide support and upgrades for a limited time before you need to fork out for any further enhancements or assistance.

5. Greater Security & Quality

Open source software is available publicly. A large amount of developers globally contribute and analyze the code making it more secure and constantly increasing the quality. The peer review process drive excellence in design.

1.4 Disadvantages of using Open Source

There's a flip side to everything, and in the case of Open Source software it all boils down to the old saying of "there's no such thing as a free lunch". Most of the disadvantages only apply if you're not somewhat code-savvy and willing to get your hands dirty:

1. Mostly used commercial applications.

2. Projects can die

3. Support issues

Know

1.5 Application of Open Sources

1. Accounting
2. Content Management Systems
3. CRM (Customer Relationship Management)
4. Desktop Environments/ Shell replacements
5. Email Clients
6. Encoding, Conversion & Ripping Tools
7. ERP
8. File sharing & FTP
9. Graphics-Design & Modeling Tools
10. Messengers & Communication Clients
11. Project Management
12. Reporting Tools
13. RSS
14. Web Browsers

1.6 LINUX: Introduction

1.6.1 What is Linux?

Linux is a UNIX-based operating system originally developed as for Intel-compatible PC's. It is now available for most types of hardware platforms, ranging from PDAs (and according to some reports, a wristwatch) to mainframes. Linux is a "modern operating system", meaning it has such features as virtual memory, memory protection, and preemptive multitasking.

1.6.2 Why use Linux?

Reasons to Install Linux
* Configurability
* Convenience
* Stability
* Community
* Freedom

1) Configurability

Linux distributions give the user full access to configure just about any aspect of their system. Options range from the simple and straightforward (for instance, changing the background image) to the more esoteric (for instance, making the "Caps Lock" key behave like "Control"). Almost any aspect of the user experience can be configured.

2) Stability

Linux is based on the UNIX kernel. It provides preemptive multitasking and protected memory. Preemptive multitasking prevents any application from permanently stealing the CPU and locking up the machine. Protected memory prevents applications from interfering with and crashing one-another.

3) Community

Linux is part of the greater open-source community. This consists of thousands of developers and many more users world-wide who support open software. This user and developer base is also a support base.

4)Freedom

Linux is free. This means more than just costing nothing. This means that you are allowed to do whatever you want to with the software. This is why Redhat, Mandrake, and Suse are all allowed to sell their own distributions of Linux. The only restriction placed on Linux is that, if you distribute Linux, you must grant all the privileges to the code that you had, including providing the source. This prevents a corporation from using the Linux kernel as the basis for their proprietary operating system.

1.6.3 Kernel mode and user mode

UML, like all Linux ports, has to provide to the generic kernel all of the facilities that it needs in order to run. A basic platform facility is a distinction between an unprivileged user mode and a privileged kernel mode. Hardware platforms provide a built-in mechanism for switching between these two modes and enforcing the lack of privileges in user mode. However, Linux provides no

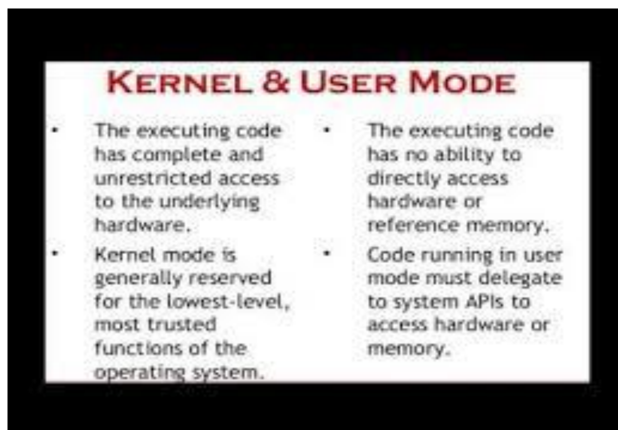
such mechanism to its processes, so UML constructs it using the trace system call tracing mechanism.

UML has a special thread whose main job is to trace almost all of the other threads. When a process is in user space, its system calls are being intercepted by the tracing thread. When it's in the kernel, it's not under system call tracing. This is the distinction between user mode and kernel mode in UML.

The transition from user mode to kernel mode is done by the tracing thread. When a process executes a system call or receives a signal, the tracing thread forces the process to run in the kernel if necessary and continues it without system call tracing.

The transition back is also performed by the tracing thread, but it's requested by the process. When it's finished executing in the kernel, because it finished either a system call or trap, it sends a signal (`SIGUSR1`) to itself. This is intercepted by the tracing thread, which restores the process state if necessary and continues the process with tracing on.

1.6.4 Difference between kernel mode and user mode



1.7 Process

- Process form the fundamental part of the LINUX operating system.
- A process is basically a program in execution. It is an entity which represents the basic unit of work to be implemented in system.
- As a multiuser system , LINUX allows many users to access the system at the same time.
- A process consist of program code , data , variables , file descriptors and an environment

1.7.1 Process structure

The Linux kernel refers process as task. Each process is allocated a unique number called a process identifier or PID. It is a positive integer between 2 and 32,768. No: 1 is reserved for init process. The variables used by the program are distinct for each process.

When a program is loaded into memory it becomes a process and it can be divided into four sections.

1) **Stack**: contains the temporary data such as method/function parameters and return address

2) **Heap**: It is dynamically allocated memory to a process.

3) **Text**: It includes the current activity represented by the program counter.

4) **Data**: contains the global and static variables.

1.7.2 Process creation

Most operating system implements a spawn mechanism to create a new process. Linux separates the generation into two distinct functions.

Fork (): creates a child process that is a copy of the current process. It differs from parent process in its PID and certain resources which are not inherited.

Exec (): It replaces a running process with another application loaded from an executable binary file.

1.7.3 Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

1) Start

This is the initial state when a process is first started/created.

2) Ready

The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.

Running

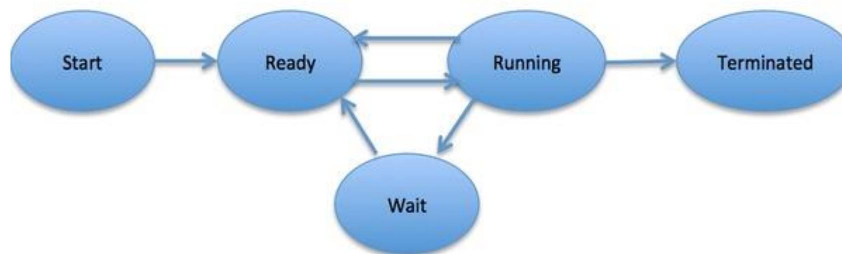
Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

Waiting

Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

Terminated or Exit

Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



1.7.4 Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

1	**Process privileges** This is required to allow/disallow access to system resources.
2	**Process ID** Unique identification for each of the process in the operating system.
3	**Pointer** A pointer to parent process.
4	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.

5	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
6	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
7	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
8	**IO status information** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −

The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

1.8 Process Scheduling

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Schedulers in Operating System are the process which decides which task and process should be accessed and run at what time by the system resources. It is required to maintain the multi tasking capabilities of a computer and to keep its performance at the highest level by scheduling the process according to their preferences and need. The Schedulers in Operating System are the algorithms which help in the system optimisation for maximum performance.

1.9 Personalities

Linux supports different execution domains, or personalities, for each process. Among other things, execution domains tell Linux how to map signal numbers into signal actions. The execution domain system allows Linux to provide limited support for binaries compiled under other Unix-like operating systems.

This function will return the current **personality** () when *persona* equals 0xffffffff. Otherwise, it will make the execution domain referenced by *persona* the new execution domain of the current process.

MySQL
MYSQL

Unit – II

MySQL

Course outcome: Learn the concepts and queries in MySQL
access simple databases from PHP using dynamically generated SQL♦To introduce basic PHP programming
To

2.1 MySQL Introduction

- MySQL is the most popular open source SQL database management system (DBMS).
- A fast, reliable, easy-to-use, multi-user multi-threaded relational database system.
- It is freely available and released under GPL (GNU General Public License).
- MySQL is a data storage area. In this storage area, there are small sections called Tables.

2.1.1 Advantages

- MySQL is Cross-Platform.
- MySQL is fast.
- MySQL is free.
- Reliable and easy to use.
- Multi-Threaded multi-user and robust SQL Database server.

2.1.2 Disadvantages

- Missing Sub-selects.
- MySQL doesn't yet support the Oracle SQL extension.
- Does not support Stored Procedures and Triggers.
- MySQL doesn't support views, but this is on the TODO.

2.2 Setting up Account

- In order to provide access the MySQL database you need to create an account to use for connecting to the MySQL server running on a given host.
- Use the GRANT statement to set up the MySQL user account. Then use that account's name and password to make connections to the server.

- User names, as used by MySQL for authentication purposes, have nothing to do with user names (login names) as used by Windows or Unix
- MySQL user names can be up to 16 characters long.
- MySQL passwords have nothing to do with passwords for logging in to your operating system.

2.2.1 Account Management Statements

- CREATE USER
- DROP USER
- RENAME USER
- REVOKE
- SET PASSWORD

2.2.1.1 CREATE USER

Syntax:

CREATE USER user [IDENTIFIED BY [PASSWORD] 'password'] [, user [IDENTIFIED BY [PASSWORD] 'password']] ...

Example:

CREATE USER 'monty'@'localhost' IDENTIFIED BY 'some_pass';

2.2.1.2 DROP USER

Syntax:

DROP USER user [, user] ...

Example:

DROP USER 'jeffrey'@'localhost';

RENAME USER

Syntax

RENAME USER old_user TO new_ user [, old_user TO new_ user] ... **Example**

RENAME USER 'jeffrey'@'localhost' TO 'jeff'@'127.0.0.1';

2.2.1.3 SET PASSWORD

Syntax

SET PASSWORD [FOR user] =

{

PASSWORD ('some password') | OLD_PASSWORD ('some password') | 'encrypted password'

}

Example:

SET PASSWORD FOR 'bob'@'%.loc.gov' = PASSWORD ('new pass');

2.3 Starting and Terminating mysql

To start the **mysql** program, try just typing its name at your command-line prompt. If **mysql** starts up correctly, you'll see a short message, followed by a mysql> prompt that indicates the program is ready to accept queries. To illustrate, here's what the welcome message looks like (to save space, I won't show it in any further examples):

```
% mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18427 to server version: 3.23.51-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

(i) starting Mysql:

*The mysql server can be started manually from the command line.

*To start the mysql server from the command line , you should start a console window for "DOS window" and enter this command - Shell:"c:/programFiles/MYSQL SERVER5.0/bin/mysql".

*The path to mysqld may vary depending on the install location of MYSQL on your system.

(ii) Login:

*To start the mysql program , type mysql at your command line prompt.If mysql starts up correctly , you'll see a short message , followed by a mysql>prompt that indicates the program is ready to accept queries.

*To illustrate , here's what the welcome message looks like (to save space / won't show it in any further examples).

(iii) %mysql

Welcome to yhe mysql monitor , commands ends with org. your mysql connection id is 18427 to server version : 3.2351 – log. Type 'help' or '/h'. Type '/c' to clear the buffer.

(iv) Mysql>

*If mysql tries to start but exists immediately with an "access denied" message , you'll need to specify connection parameters.

*The most commonly needed parameters are the host to connect to the host where the mysql server runs your mysql , username and a password.

For example :

% mysql-h localhost-p-u cbuser

Enter password : cbpass.

(v) Stopping mysql

You can stop the mysql server by executing this command : shell>-c:\program Files\MYSQL\MYSQL Server 5.0\bin\mysqladmin".

(vi) Logout :

To terminate a mysql a session , issue a QUIT statement : mysql>QUIT. You can also terminate the session by issuing an EXIT statement or (under UNIX) by typing ctrl+D.

2.4 Record selection technology (or) select statements

SELECT statement that is used for retrieving information from a database. It provides some essential background that shows various ways you can use SELECT to tell MySQL what you want to see.

SELECT gives you control over several aspects of record retrieval:

- Which table to use
- Which columns to display from the table
- What names to give the columns
- Which rows to retrieve from the table
- How to sort the rows

Many useful queries are quite simple and don't specify all those things. For example, some forms of SELECT don't even name a table—a fact used in Recipe 1.32, which discusses how to use **mysql** as a calculator. Other non-table-based queries are useful for purposes such as checking what version of the server you're running or the name of the current database:

```
mysql> SELECT VERSION( ), DATABASE( );
+-------------+------------+
| VERSION( )  | DATABASE( ) |
+-------------+------------+
| 3.23.51-log | cookbook   |
+-------------+------------+
```

2.4.1 Using the SELECT Command

SELECT is the SQL command used to retrieve records. This command syntax can be totally simplistic or very complicated. As you become more comfortable with database programming, you will learn to enhance your SELECT statements, ultimately making your database do as much work as possible and not overworking your programming language of choice.
The most basic SELECT syntax looks like this:

```
SELECT expressions_and_columns FROM table_name

[WHERE some_condition_is_true]

[ORDER BY some_column [ASC | DESC]]

[LIMIT offset, rows]
```

Start with the first line:

SELECT expressions_and_columns FROM table_name

One handy expression is the * symbol, which stands for "everything". So, to select "everything" (all rows, all columns) from the master_name table, your SQL statement would be

SELECT * FROM master_name;

Depending on how much data you inserted into the master_name table during the previous hour, your results will vary, but it may look something like this:

```
mysql> SELECT * FROM master_name;

+---------+---------------------+---------------------+-----------+----------+

| name_id | name_dateadded      | name_datemodified   | firstname | lastname |

+---------+---------------------+---------------------+-----------+----------+

|       1 | 2001-10-29 13:11:00 | 2001-10-29 13:11:00 | John      | Smith    |

|       2 | 2001-10-29 13:11:00 | 2001-10-29 13:11:00 | Jane      | Smith    |

|       3 | 2001-10-29 13:11:00 | 2001-10-29 13:11:00 | Jimbo     | Jones    |

|       4 | 2001-10-29 13:11:00 | 2001-10-29 13:11:00 | Andy      | Smith    |

|       7 | 2001-10-29 14:16:21 | 2001-10-29 14:16:21 | Chris     | Jones    |

|      45 | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | Anna      | Bell     |

|      44 | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | Jimmy     | Carr     |
```

```
|       43 | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | Albert    | Smith    |

|       42 | 0000-00-00 00:00:00 | 0000-00-00 00:00:00 | John      | Doe      |

+---------+---------------------+---------------------+-----------+----------+

9 rows in set (0.00 sec)
```

As you can see, MySQL creates a lovely table as part of the result set, with the names of the columns along the first row. If you want to select only specific columns, replace the * with the names of the columns, separated by commas. The following statement selects just the name_id, firstname and lastname fields from the master_name table.

```
mysql> SELECT name_id, firstname,

mysql> SELECT name_id, firstname, lastname FROM master_name;

+---------+-----------+----------+

| name_id | firstname | lastname |

+---------+-----------+----------+

|       1 | John      |   Smith  |

|       2 | Jane      |   Smith  |

|       3 | Jimbo     |   Jones  |

|       4 | Andy      |   Smith  |

|       7 | Chris     |   Jones  |

|      45 | Anna      |   Bell   |
```

```
|      44 | Jimmy     |   Carr   |

|      43 | Albert    |   Smith  |

|      42 | John      |   Doe    |

+---------+-----------+----------+

9 rows in set (0.00 sec)
```

A useful expression used with SELECT is DISTINCT, which (not surprisingly) will return distinct occurrences in a result set. For example, the master_name table has more than one person with the last name of "Smith". If you wanted to select last names without repeating results, you would use DISTINCT:

```
mysql> SELECT DISTINCT lastna

mysql> SELECT DISTINCT lastname FROM master_name;

+----------+

| lastname |

+----------+

| Bell     |

| Carr     |

| Doe      |

| Jones    |

| Smith    |
```

+----------+

5 rows in set (0.00 sec)

2.4.2 Ordering SELECT Results

By default, results of SELECT queries are ordered as they appear in the table. If you want to order results a specific way, such as by date, ID, name, and so on, specify your requirements using the ORDER BY clause. In the following statement, results are ordered by lastname:

mysql> SELECT name_id, firstname, lastname FROM master_name ORDER BY lastname;

+---------+-----------+----------+

| name_id | firstname | lastname |

+---------+-----------+----------+

| 45 | Anna | Bell |

44 | Jimmy | Carr |

| 42 | John | Doe |

| 3 | Jimbo | Jones |

| 7 | Chris | Jones |

| 1 | John | Smith |

| 2 | Jane | Smith |

| 4 | Andy | Smith |

| 43 | Albert | Smith |

+---------+-----------+----------+

9 rows in set (0.00 sec)

Sr.No.	Name & Description
1	ASCII() Returns numeric value of left-most character
2	BIN() Returns a string representation of the argument
3	BIT_LENGTH() Returns length of argument in bits
4	CHAR_LENGTH() Returns number of characters in argument
5	CHAR() Returns the character for each integer passed
6	CHARACTER_LENGTH() A synonym for CHAR_LENGTH()
7	CONCAT_WS() Returns concatenate with separator
8	CONCAT() Returns concatenated string
9	CONV() Converts numbers between different number bases
10	ELT()

	Returns string at index number
11	EXPORT_SET() Returns a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
12	FIELD() Returns the index (position) of the first argument in the subsequent arguments
13	FIND_IN_SET() Returns the index position of the first argument within the second argument
14	FORMAT() Returns a number formatted to specified number of decimal places
15	HEX() Returns a string representation of a hex value

1) LPAD (str, len, padstr)

Returns the string str, left-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

```
mysql> SELECT LPAD('hi',4,'??');
+---------------------------------------------------------+
|                 LPAD('hi',4,'??')                |
+---------------------------------------------------------+
|                      ??hi                    |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

2) LTRIM (str)

Returns the string str with leading space characters removed.

```
mysql> SELECT LTRIM('  barbar');
```

```
+-------------------------------------------------------+
|                LTRIM('  barbar')                 |
+-------------------------------------------------------+
|                    barbar                     |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

3) RPAD(str,len,padstr)

Returns the string str, right-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

```
mysql> SELECT RPAD('hi',5,'?');
+-------------------------------------------------------+
|                 RPAD('hi',5,'?')                 |
+-------------------------------------------------------+
|                     hi???                     |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

4) RTRIM(str)

Returns the string str with trailing space characters removed.

```
mysql> SELECT RTRIM('barbar   ');
+-------------------------------------------------------+
|                 RTRIM('barbar   ')                 |
+-------------------------------------------------------+
|                      barbar                     |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

5) STRCMP(str1, str2)

Compares two strings and returns 0 if both strings are equal, it returns -1 if the first argument is smaller than the second according to the current sort order otherwise it returns 1.

```
mysql> SELECT STRCMP('MOHD', 'MOHD');
+-------------------------------------------------------+
|              STRCMP('MOHD', 'MOHD')               |
+-------------------------------------------------------+
|                       0                       |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

Another example is −

```
mysql> SELECT STRCMP('AMOHD', 'MOHD');
+-------------------------------------------------------+
|              STRCMP('AMOHD', 'MOHD')                   |
+-------------------------------------------------------+
|                    -1                                  |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

6) UPPER(str)

Returns the string str with all characters changed to uppercase according to the current character set mapping.

```
mysql> SELECT UPPER('Allah-hus-samad');
+-------------------------------------------------------+
|              UPPER('Allah-hus-samad')                  |
+-------------------------------------------------------+
|                 ALLAH-HUS-SAMAD                        |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

2.6 Date and Time functions

Functions	Description
ADDDATE()	MySQL ADDDATE() adds a time value with a date.
ADDTIME()	In MySQL the ADDTIME() returns a time or datetime after adding a time value with a time or datetime.
CONVERT_TZ()	In MySQL the CONVERT_TZ() returns a resulting value after converting a datetime value from a time zone specified as the second argument to the time zone specified as the third argument.

CURDATE()	In MySQL the CURDATE() returns the current date in 'YYYY-MM-DD' format or 'YYYYMMDD' format depending on whether numeric or string is used in the function.
CURRENT_DATE()	In MySQL the CURRENT_DATE returns the current date in 'YYYY-MM-DD' format or YYYYMMDD format depending on whether numeric or string is used in the function.
CURRENT_TIME()	In MySQL the CURRENT_TIME() returns the current time in 'HH:MM:SS' format or HHMMSS.uuuuuu format depending on whether numeric or string is used in the function.
CURRENT_TIMESTAMP()	In MySQL the CURRENT_TIEMSTAMP returns the current date and time in 'YYYY-MM-DD HH:MM:SS' format or YYYYMMDDHHMMSS.uuuuuu format depending on whether numeric or string is used in the function.
CURTIME()	In MySQL the CURTIME() returns the value of current time in 'HH:MM:SS' format or HHMMSS.uuuuuu format depending on whether numeric or string is used in the function.
DATE_ADD()	MySQL DATE_ADD() adds time values (as intervals) to a date value. The ADDDATE() is the synonym of

	DATE_ADD().
DATE_FORMAT()	MySQL DATE_FORMAT() formats a date as specified in the argument. A list of format specifiers given bellow may be used to format a date.
DATE_SUB()	MySql date_sub() function subtract a time value (as interval) from a date.
DATE()	MySQL DATE() takes the date part out from a datetime expression.
DATEDIFF()	MySQL DATEDIFF() returns the number of days between two dates or datetimes.
DAY()	MySQL DAY() returns the day of the month for a specified date.
DAYNAME()	MySQL DAYNAME() returns the name of the week day of a date specified in the argument.
DAY OF MONTH()	MySQL DAYOFMONTH() returns the day of the month for a given date.
DAY OF WEEK()	MySQL DAYOFWEEK() returns the week day number (1 for Sunday,2 for Monday …… 7 for Saturday) for a date specified as an argument.

DAY OF YEAR()	MySQL DAYOFYEAR() returns day of the year for a date. The return value is within the range of 1 to 366.
EXTRACT()	MySQL EXTRACT() extracts a part of a given date.
FROM_DAYS()	MySQL FROM_DAYS() returns a date against a datevalue.
FROM_UNIXTIME()	MySQL FROM_UNIXTIME() returns a date /datetime from a version of unix_timestamp.
GET_FORMAT()	MySQL GET_FORMAT() converts a date or time or datetime in a formatted manner as specified in the argument.
HOUR()	MySQL HOUR() returns the hour of a time.
LAST_DAY()	MySQL LAST_DAY() returns the last day of the corresponding month for a date or datetime value.
LOCALTIME()	MySQL LOCALTIME returns the value of current date and time in 'YYYY-MM-DD HH:MM:SS' format or YYYYMMDDHHMMSS.uuuuuu format depending on the context (numeric or string) of the function.
LOCALTIMESTAMP()	MySQL LOCALTIMESTAMP returns the value of current date and time in 'YYYY-MM-DD HH:MM:SS' format or

	YYYMMDDHHMMSS.uuuuuu format depending on the context (numeric or string) of the function.
MAKEDATE()	MySQL MAKEDATE() returns a date by taking a value of a year and a number of days. The number of days must be greater than 0 otherwise a NULL will be returned.
MAKETIME()	MySQL MAKETIME() makes and returns a time value from a given hour, minute and seconds.
MICROSECOND()	MySQL MICROSECOND() returns microseconds from the time or datetime expression.
MINUTE()	MySQL MINUTE() returns a minute from a time or datetime value.
MONTH()	MySQL MONTH() returns the month for the date within a range of 1 to 12 (January to December).
MONTHNAME()	MySQL MONTHNAME() returns the full name of the month

2.7 Sorting Query Results

- SQL **SELECT** command to fetch data from MySQL table. When you select rows, the MySQL server is free to return them in any order, unless you instruct it otherwise by saying how to sort the result. But a query doesn't come out in the order you want.

2.7.1 Using ORDER BY to Sort Query Results

- Add an ORDER BY clause. ORDER BY will tell the MySQL server to sort the rows by a column.
- Define in which direction to sort, as the order of the returned rows may not yet be meaningful. Rows can be returned in ascending or descending order.

Syntax:

SELECT field1, field2,...fieldN table_name1, table_name2...ORDER BY field1, [field2...] [ASC [DESC]]

Example:

mysql> SELECT * from tutorials_tbl ORDER BY tutorial_author ASC

2.7.2 Sorting Subsets of a Table

- You don't want to sort an entire table, just part of it. Add a WHERE clause that selects only the records you want to see.
- mysql> **SELECT trav_date, miles FROM driver_log WHERE n'Henry' ORDER BY trav_date;**

2.7.3 Sorting and NULL Values

- To sort columns that may contain NULL values

mysql> **SELECT NULL = NULL;**

2.7.4 Sorting by Calendar Day

- To sort by day of the calendar year. Sort using the month and day of a date, ignoring the year. Sorting in calendar order differs from sorting by date.
- mysql> **SELECT date, description FROM event ORDER BY date;**

2.8 Generating Summary

Summaries are useful when you want the overall picture rather than the details.

2.8.1 Summary

- Using aggregate function we can achieve summary of values. Aggregate functions are COUNT (), MIN (), MAX (), SUM. (), AVG () and GROUP BY clause to group the rows into subsets and obtain an aggregate value for each one.
- To getting a list of unique values, use SELECT DISTINCT rather than SELECT.
- Using COUNT (DISTINCT) - To count how many distinct values there are.

2.8.2 Summarizing with COUNT( )

- To count the number of rows in an entire table or that match particular conditions, use the

COUNT( ) function.

- For example,

mysql> **SELECT COUNT(*) FROM emp_tab;**

2.8.3 Summarizing with MIN( ) and MAX( )

- Finding smallest or largest values in an entire table, use the MIN () and MAX () function.

For example,

mysql> **SELECT MIN(Sal) AS low, MAX(sal) AS**

high FROM emp_tab;

2.8.4 Summarizing with SUM( ) and AVG( )

- SUM( ) and AVG( ) produce the total and average (mean) of a set of values:

For Example,

mysql> **SELECT SUM(rno) AS 'No-Of Emp', AVG(sal) AS 'Avg Sal' FROM**

2.9 Working with Metadata

Introduction

Most of the SQL statements used so far have been written to work with the data stored in the database. That is, after all, what the database is designed to hold. But sometimes you need more than just data values. You need information that characterizes or describes those values—that is, the statement metadata. Metadata is used most often to process result sets, but also applies to other aspects of your interaction with MySQL

- Metadata is data about data.
- For example - Consider a file with a picture. The picture or the pixels inside the file are data. A description of the picture, like "JPEG format, 300x400 pixels, 72dpi", is metadata, because it describes the content of the file, although it's not the actual data.

2.9.1 Types of Metadata

- Information about the result of queries
- Information about tables and databases.
- Information about the MySQL server.

1) **Information about statement results**

- For statements that delete or update rows, you can determine how many rows were changed. For a SELECT statement, you can obtain the number of columns in the result set, as well as information about each column in the result set, such as the column name and its display width. For example, to format a tabular display, you can determine how wide to make each column and whether to justify values to the left or right.

2) **Information about databases and tables**

A MySQL server can be queried to determine which databases and tables it manages, which is useful for existence tests or producing lists. For example, an application might present a display enabling the user to select one of the available databases. Table metadata can be examined to determine column definitions; for example, to determine

the legal values for ENUM or SET columns to generate web form elements corresponding to the available choices.

2.10 Using Sequences

A sequence is a database object that generates numbers in sequential order. Applications most often use these numbers when they require a unique value in a table such as primary key values. The following list describes the characteristics of sequences.

2.10.1 Creating a Sequence

```
CREATE SEQUENCE sequence_name
[INCREMENT BY #]
[START WITH #]
[MAXVALUE # | NOMAXVALUE]
[MINVALUE # | NOMINVALUE]
[CYCLE | NOCYCLE]
```

2.10.2 Dropping a Sequence

- DROP SEQUENCE my_sequence

2.10.3 Using a Sequence

- Use sequences when an application requires a unique identifier.
- INSERT statements, and occasionally UPDATE statements, are the most common places to use sequences.
- Two "functions" are available on sequences:

- **NEXTVAL:** Returns the next value from the sequence.
- **CURVAL:** Returns the value from the last call to NEXTVAL by the current user during the current connection.

Examples

- To create the sequence:
  CREATE SEQUENCE customer_seq INCREMENT BY 1 START WITH 100

- To use the sequence to enter a record into the database: INSERT INTO customer (cust_num, name, address) VALUES (customer_seq.NEXTVAL, 'Kalam', '123 Gandhi Nagar.')

2.11 MySQL and Web

- MySQL makes it easier to provide dynamic rather than static content. Static content exists as pages in the web server's document that are served exactly as is.
- Visitors can access only the documents that you place in the tree, and changes occur only when you add, modify, or delete those documents.
- By contrast, dynamic content is created on demand.

2.11.1 Basic Web Page Generation

- Using HTML we can generate your own Web site.
- HTML is a language for describing web pages.
- HTML stands for **H**yper **T**ext **M**arkup **L**anguage
- HTML is not a programming language, it is a **markup language**
- A markup language is a set of **markup tags**
- HTML uses **markup tags** to describe web pages

2.11.2 HTML

- HTML documents **describe web pages**
- HTML documents **contain HTML tags** and plain text
- HTML documents are also **called web pages.**

1) **Static Web Page**

   A static web page shows the required information to the viewer, but do not accept any information from the viewer.

2) **Dynamic Web Page**

   A dynamic web page displays the information to the viewer and also accepts the information from the user Railway reservation, Online shopping etc. are examples of dynamic web page.

2.11.3 Client side scripting

It is a script, (ex. Javascript, VB script), that is executed by the browser (i.e. Firefox, Internet Explorer, Safari, Opera, etc.) that resides at the user computer

2.11.4 Server side scripting

- It is a script (ex. ASP .NET, ASP, JSP, PHP, Ruby, or others), is executed by the server (Web Server), and the page that is sent to the browser is produced by the serve-side scripting.

2.11.5 Using Apache to Run Web Scripts

- Open-Source Web server originally based on NCSA server(National Center for Supercomputing Applications).
- Apache is the most widely used web server software package in the world.
- Apache is highly configurable and can be setup to support technologies such as, password protection, virtual hosting (name based and IP based), and SSL encryption.

Unit -III

PHP

Couse Outcome:Introduce basic structure of PHP programming . To construct Web applications using php.

3.1 INTRODUCTION:

PHP started out as a small open source project that evolved as more and more people found out how useful it was. Rasmus Lerdorf unleashed the first version of PHP way back in 1994.

- PHP is a acronym for "PHP: Hypertext Preprocessor".
- PHP is a server side scripting language that is embedded in HTML. It is used to manage dynamic
content, databases, session tracking, even build entire ecommerce
sites.
- It is integrated with a number of popular databases, including MySQL, Postgre SQL, Oracle, Sybase,
Informix and Microsoft SQL Server.

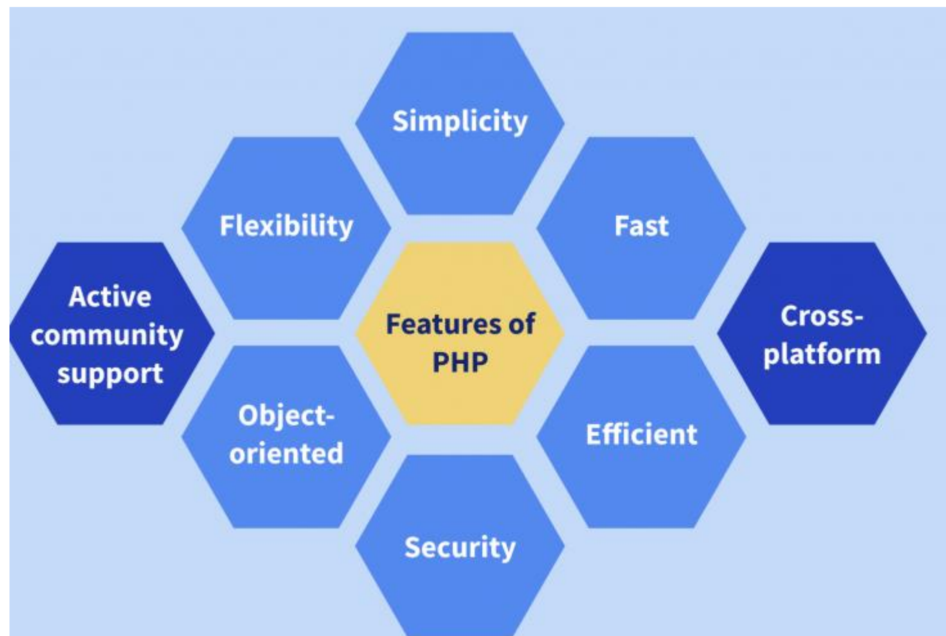
3.1.1 Common uses of PHP

- PHP performs system functions, i.e. from files on a system it can create, open, read, write, and close
them.
- PHP can handle forms, i.e. gather data from files, save data to a file, through email you can send data,
return data to the user.
- You add, delete, modify elements within your database through PHP.
- Access cookies variables and set cookies.
- Using PHP, you can restrict users to access some pages of your website.
- It can encrypt data.

3.1.2 Characteristics of PHP

Five important characteristics make PHP's practical nature possible −

- Simplicity
- Efficiency
- Security
- Flexibility
- Familiarity



3.2 PHP Variable

A variable is a name given to a memory location that stores data at runtime. The scope of a variable determines its visibility. A Php global variable is accessible to all the scripts in an application. A local variable is only accessible to the script that it was defined in.

Rules for creating variables in PHP.

 A variable starts with the $ sign, followed by the name of the variable.
 A variable name must start with a letter or the underscore char acter .
 A variable name cannot start with a number.
 A variable name can only contain alphanumer
ic characters and underscores (Az,09,and _)

Cr eating (Declaring) PHP Var iables

In PHP, a variable starts with the $ sign, followed by the name of the variable:

EXAMPLE:

```
<?php
$txt = "Hello world!";
$x = 5;
$y = 10.5;
?>
```

Output Var iables

The PHP echo statement is often used to output data to the screen

PHP has three different variable scopes:
- local
- global
- static

3.2.1 Global and Local Scope

A variable declared outside a function has a GLOBAL SCOPE and can only be accessed outside a function:

Example

```
<?php
$x = 5; // global scope
function myTest() {
// using x inside this function will generate an error
echo "<p>Variable x inside function is: $x</p>";
}
myTest();
echo "<p>Variable x outside function is: $x</p>";

?>
```

3.2.2 CONSTANT

*A constant is a name or an identifier for a fixed value. Constant are like variables except that onethey are defined, they cannot be undefined or changed (except magic constants).

*Constants are very useful for storing data that doesn't change while the script is running. Common examples of such data include configuration settings such as database username andpassword, website's base URL, company name, etc.

*Constants are defined using PHP's define() function, which accepts two arguments: the name of the constant, and its value. Once defined the constant value can be accessed at any time just byreferring to its name. Here is a simple example:

Example

Run this code »

```
<?php
// Defining constant
define("SITE_URL", "https://www.tutorialrepublic.com/");
// Using constant
echo 'Thank you for visiting '
. SITE_URL;
```

```
?>
```

3.2.3 PHP Echo and Print Statements

The PHP echo Statement

The echo statement can output one or more strings. In general terms, the echo statement can display anything that can be displayed to the browser, such as string, numbers, variables values, the results of expressions etc.
Since echo is a language construct not actually a function (like if statement), you can use it without parentheses e.g. echo or echo (). However, if you want to pass more than one parameter to echo, the parameters must not be enclosed within parentheses.

Example

```
<?php
echo "<h4>This is a simple heading.</h4>";

echo "<h4 style='color: red;'>This is heading with style.</h4>";
?>
```

3.2.4 The PHP print Statement

You can also use the print statement (an alternative to echo) to display output to the browser .Like echo the print is also a language construct not a real function. So you can also use it without parentheses like: print or print ().Both echo and print statement works exactly the same way except that the print statement can
Only output one string, and always returns 1. That's why the echo statement considered marginally faster than the print statement since it doesn't return any value.

3.3 PHP Data Types

Data Types defines the type of data a variable can store. PHP allows eight different types of datatypes:

1.Integer : Integers hold only whole numbers including positive and negative numbers, i.e., numbers without fractional part or decimal point. They can be decimal (base 10), octal (base 8) or hexadecimal (base 16). The default base is decimal (base 10). The octal integers can be declared with leading 0 and the hexadecimal can be declared with leading 0x. The range of integers must lie between 2^31 to 2^31.

Example

```
?php
```

```
// decimal base integers
$deci1 = 50;
$deci2 = 654;
// octal base integers
$octal1 = 07;
// hexadecimal base integers
$octal = 0x45;
$sum = $deci1 + $deci2;
echo $sum;
?>
```

output: 704

2.Double: Can hold numbers containing fractional or decimal part including positive and negative numbers. By default, the variables add a minimum number of decimal places.

Example:

```
<?php
$val1 = 50.85;
$val2 = 654.26;
$sum = $val1 + $val2;
echo $sum;
?>
```

Output:

705.11

3.Str ing : Hold letters or any alphabets, even numbers are included. These are written within double quotes during declaration. The strings can also be written within single quotes but it will be treated differently while printing variables. To clarify this look at the example below.

Example:

```
<?php
$name = "Krishna";
echo "The name of the Geek is $name \n";
echo 'The name of the geek is $name';
?>
```

Output:

The name of the Geek is Krishna
The name of the geek is $name

4.NULL: These are special types of variables that can hold only one value i.e., NULL. We follow the convention of writing it in capital form, but its case sensitive.

Example:

```
<?php
$nm = NULL;
echo $nm; // This will give no output
?>
```

5.Boolean: Hold only two values, either TRUE or FALSE. Successful events will return true and unsuccessful events return false. NULL type values are also treated as false in Boolean. Apart from NULL, 0 is also consider as false in boolean. If a string is empty then it is also considered as false in boolean data type.
Example:

```
<?php
if(TRUE)
echo "This condition is TRUE";
if(FALSE)
echo "This condition is not TRUE";
?>
```

Output:
This condition is TRUE
6.Ar r ays: Array is a compound datatype which can store multiple values of same data type. Below is an example of array of integers.

```
<?php
$intArray = array( 10, 20 , 30);
echo "First Element: $intArray[0]\n";
echo "Second Element: $intArray[1]\n";
echo "Third Element: $intArray[2]\n";
?>
```

Output:
First Element: 10
Second Element: 20
Third Element: 30

7.Objects: Objects are defined as instances of user defined classes that can hold both values and functions. This is an advanced topic and will be discussed in details in further articles.

8.Resour ces: Resources in PHP are not an exact data type. These are basically used to store references to some function call or to external PHP resources. For example, consider a database call. This is an external resource.

3.4 OPERATORS

Operators are symbols that tell the PHP processor to perform certain actions. For example, the Addition (+) symbol is an operator that tells PHP to add two variables or values, while the greater than (>) symbol is an operator that tells PHP to compare two values.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators
- Conditional (or ternary) Operators

1) PHP Arithmetic Operators

The arithmetic operators are used to perform common arithmetical operations,

such as addition, subtraction, multiplication etc. Here's a complete list of PHP's arithmetic operators:

% Modulus $x % $y Remainder of $x divided by $y
/ Division $x / $y Quotient of $x and $y
* Multiplication $x * $y Product of $x and $y.
- Subtraction $x - $y Difference of $x and $y.
+ Addition $x + $y Sumof $x and $y

Example
Run this code »
```
<?php
$x = 10;
$y = 4;
echo($x + $y); // 0utputs: 14
echo($x $
y); // 0utputs: 6
echo($x * $y); // 0utputs: 40
echo($x / $y); // 0utputs: 2.5
echo($x % $y); // 0utputs: 2
?>
```

2)PHP Assignment Operators

The following example will show you. The assignment operators are used to assign values to variables.
%= Divide and assign modulus $x %= $y $x = $x % $y
/= Divide and assign quotient $x /= $y $x = $x / $y
*= Multiply and assign $x *= $y $x = $x * $y
- = Subtract and assign $x - = $y $x = $x - $y
+= Add and assign $x += $y $x = $x + $y
= Assign $x = $y $x = $y

Example
Run this code »
```
<?php
$x = 10;
echo $x; // Outputs: 10
$x = 20;
$x += 30;
echo $x; // Outputs: 50
$x = 50;
$x =
20;
echo $x; // Outputs: 30
$x = 5;
$x *= 25;
```

```
echo $x; // Outputs: 125
$x = 50;
$x /= 10;
echo $x; // Outputs: 5
$x = 100;
$x %= 15;
echo $x; // Outputs: 10
?>
```

3)PHP Comparison Operators

The comparison operators are used to compare two values in a Boolean fashion.

Operator / Name / Example	Result
<= Less than or equal to $x <= $y	True if $x is less than or equal to $y
>= Greater than or equal to $x >= $y	True if $x is greater than or equal to $y
> Greater than $x > $y	True if $x is greater than $y
< Less than $x < $y	True if $x is less than $y
! == Not identical $x ! == $y	True if $x is not equal to $y, or they are not of the same type
<> Not equal $x <> $y	True if $x is not equal to $y
! = Not equal $x ! = $y	True if $x is not equal to $y
=== Identical $x === $y	True if $x is equal to $y, and they are of the same type
== Equal $x == $y	True if $x is equal to $y

Operator Name Example Result
The following example will show you these comparison operators in action:

Example
Run this code »

```
<?php
$x = 25;
$y = 35;
$z = "25";
var_dump($x == $z); // Outputs: boolean true
var_dump($x === $z); // Outputs: boolean false
var_dump($x != $y); // Outputs: boolean true
var_dump($x !== $z); // Outputs: boolean true
var_dump($x < $y); // Outputs: boolean true
var_dump($x > $y); // Outputs: boolean false
var_dump($x <= $y); // Outputs: boolean true
var_dump($x >= $y); // Outputs: boolean false
?>
```

4)PHP Incrementing and Decrementing Operator

The increment/decrement operators are used to increment/decrement a variable's value.
$x- - Post-decrement Returns $x, then decrements $x by one
- - $x Pre-decrement Decrements $x by one, then returns $x

$x++ Post-increment Returns $x, then increments $x by one
++$x Pre-increment Increments $x by one, then returns $x
Operator Name Effect
The following example will show you these increment and decrement operators in action:

Example
Run this code »

```
<?php
$x = 10;
echo ++$x; // Outputs: 11
echo $x; // Outputs: 11
$x = 10;
echo $x++; // Outputs: 10
echo $x; // Outputs: 11
$x = 10;
echo $
x; // Outputs: 9
echo $x; // Outputs: 9
$x = 10;
echo $x;
// Outputs: 10
echo $x; // Outputs: ?>
```

5)PHP Logical Operators

The logical operators are typically used to combine conditional statements.
! Not ! $x True if $x is not true
| | Or $x | | $y True if either $$x or $y is true
&& And $x && $y True if both $x and $y are true
xor Xor $x xor $y True if either $x or $y is true, but not both
or Or $x or $y True if either $x or $y is true
and And $x and $y True if both $x and $y are true
Operator Name Example Result

```
<?php $year = 2014; // Leap years are divisible by 400 or by 4 but not 100
i f ( ( $year % 400 == 0) | | ( ( $year % 100 ! = 0) && ( $year % 4 == 0) ) ) { echo
" $year i s a leap year . " ; } el se{ echo " $year i s not a leap year . " ; } ?>
```

6)PHP String Operators

There are two operators which are specifically designed for strings.
. = Concatenation assignment $st r 1 . = $st r 2 Appends the $str2 to the $str1
. Concatenation $st r 1 . $st r 2 Concatenation of $str1 and $str2
Example
Run this code »

```
<?php
$x = "Hello";
$y = " World!";
echo $x . $y; // Outputs: Hello World!
```

```
$x .= $y;
echo $x; // Outputs: Hello World!
?>
```

7)PHP Array Operators

The array operators are used to compare arrays:
=== Identity $x === $y True if $x and $y have the same key/value pairs in the same order and of the same types == Equality $x == $y True if $x and $y have the same key/value pairs + Union $x + $y Union of $x and $y
Operator Name Example Result
! == Nonidentity
$x ! == $y True if $x is not identical to $y
<> Inequality $x <> $y True if $x is not equal to $y
! = Inequality $x ! = $y True if $x is not equal to $y
The following example will show you these array operators in action:

Example

Run this code »

```
<?php
$x = array("a" => "Red", "b" => "Green", "c" => "Blue");
$y = array("u" => "Yellow", "v" => "Orange", "w" => "Pink");
$z = $x + $y; // Union of $x and $y
var_dump($z);
var_dump($x == $y); // Outputs: boolean false
var_dump($x === $y); // Outputs: boolean false
var_dump($x != $y); // Outputs: boolean true
var_dump($x <> $y); // Outputs: boolean true
var_dump($x !== $y); // Outputs: boolean true
?>
```

8)The Ternary Operator

The ternary operator provides a shorthand way of writing the if...else statements. The ternary operator is represented by the question mark (?) symbol and it takes three operands: a condition to check, a result for true, and a result for false.
To understand how this operator works, consider the following examples:

Example

Run this code »

```
<?php
if($age < 18){
echo 'Child'; // Display Child if age is less than 18
} else{
echo 'Adult'; // Display Adult if age is greater than or equal to 18
}?
>
```

Using the ternary operator the same code could be written in a more compact way:
Example
Run this code »

```
<?php echo ($age < 18) ? 'Child' : 'Adult'; ?>
```

3.5 PHP Conditional Statements

Like most programming languages, PHP also allows you to write code that perform different actions based on the results of a logical or comparative test conditions at run time. This means, you can create test conditions in the form of expressions that evaluates to either true or false and based on these results you can perform certain actions.
There are several statements in PHP that you can use to make decisions:

- The if statement
- The if...else statement
- The if...elseif....else statement
- The switch...case statement

The if Statement
The if statement is used to execute a block of code only if the specified condition evaluates to true. This is the simplest PHP's conditional statements and can be written like:

```
if(condition){
// Code to be executed
}
```

Example

```
<?php
$d = date("D");
if($d == "Fri"){
echo "Have a nice weekend!";
}?>
```

1)The if...else Statement

You can enhance the decision making process by providing an alternative choice through adding an else statement to the if statement. The if...else statement allows you to execute one block of Code if the specified condition is evaluates to true and another block of code if it is evaluates to False. It can be written, like this:

```
if(condition){
// Code to be executed if condition is true
} else{
// Code to be executed if condition is false
}
```

Example

Run this code »

```
<?php
$d = date("D");
if($d == "Fri"){
echo "Have a nice weekend!";
} else{
echo "Have a nice day!";
}
?>
```

2)The if...elseif...else Statement

The if...elseif...else a special statement that is used to combine multiple if...else statements.

```
if(condition1){
// Code to be executed if condition1 is true
} elseif(condition2){
// Code to be executed if the condition1 is false and condition2 is true
} else{
// Code to be executed if both condition1 and condition2 are false
}
```

Example

```
<?php
$d = date("D");
if($d == "Fri"){
echo "Have a nice weekend!";
} elseif($d == "Sun"){
echo "Have a nice Sunday!";
} else{
echo "Have a nice day!";
} ?>
```

3)SWITCH STATEMENTS

Switch case statement is an alternative to the if elseif else statement, which does almost the same thing. The switch case statement tests a variable against a series of values until it finds a match, and then executes the block of code corresponding to that match.

```
switch(n){
case label1:
// Code to be executed if n=label1
break;
case label2:
// Code to be executed if n=label2
break;
...
default:
// Code to be executed if n is different from all labels
}
```

Example:

```
<?php $today = dat e( " D" ) ;
swi t ch( $today)
{
case " Mon" :
echo " Today i s Monday. Clean your house. " ;
br eak;
case " Tue" :
```

```
echo " Today i s Tuesday. Buy some f ood. " ;
br eak;
defaul t :
echo " No information available f or that day. " ;
break;
}
?>
```

Switch case statement differs from the if elseif else statement in one important way.
The switch statement executes line by line (i.e. statement by statement) and once PHP finds
A case statement that evaluates to true, it's not only executes the code corresponding to that case statement, but also executes all the subsequent case statements till the end of the switch block automatically.

3.6 PHP Arrays

3.6.1 What is PHP Arrays

Arrays are complex variables that allow us to store more than one value or a group of values under a single variable name. Let's suppose you want to store colors in your PHP script. Storing the colors one by one in a variable could look something like this:

Example

Run this code »

```
<?php
$color1 = "Red";
$color2 = "Green";
$color3 = "Blue";
?>
```

But what, if you want to store the states or city names of a country in variables and this time this not just three may be hundred. It is quite hard, boring, and bad idea to store each city name in a separate variable. And here array comes into play.

3.6.2 Types of Arrays in PHP

There are three types of arrays that you can create. These are:

- **Indexed array** — An array with a numeric key.
- **Associative array** — An array where each key has its own specific value.
- **Multidimensional array** — An array containing one or more arrays within itself.

1)Indexed Arrays

An indexed or numeric array stores each array element with a numeric index. The following examples shows two ways of creating an indexed array, the easiest way is:

Example

```
<?php
```

```
// Define an indexed array
$colors = array("Red", "Green", "Blue");
?>
<?php
$col or s[ 0] = " Red" ;
$col or s[ 1] = " Gr een" ;
$col or s[ 2] = " Bl ue" ; ?>
```

2)Associative Arrays

In an associative array, the keys assigned to values can be arbitrary and user defined strings. In the following example the array uses keys instead of index numbers:

Example

Run this code »

```
<?php
// Define an associative array
$ages = array("Peter"=>22, "Clark"=>32, "John"=>28);
?>
<?php
$ages[ " Pet er " ] = " 22" ;
$ages[ " Cl ar k" ] = " 32" ;
$ages[ " John" ] = " 28" ;
?>
```

3)Multidimensional Arrays

The multidimensional array is an array in which each element can also be an array and each element in the subarray can be an array or further contain array within itself and so on. An example of a multidimensional array will look something like this:

Example

Run this code »

```
<?php
// Define a multidimensional array
$contacts = array(
array(
"name" => "Peter Parker",
"email" => "peterparker@mail.com",
),
array(
"name" => "Clark Kent",
"email" => "clarkkent@mail.com",
),
array(
"name" => "Harry Potter",
"email" => "harrypotter@mail.com",
)
);
```

3.7 PHP Functions

A function is a selfcontained block of code that performs a specific task.
PHP has a huge collection of internal or builtin functions that you can call directly within your PHP scripts to perform a specific task, like gettype(), print_r(), var_dump, etc.
Please check out PHP reference section for a complete list of useful PHP builtin functions.

3.7.1 Advantages of functions in php

- Functions reduces the repetition of code within a program
- Functions makes the code much easier to maintain
- Functions makes it easier to eliminate the errors
- Functions can be reused in other application

1. Creating the function
2. Calling the function

3.7.2 Creating and Invoking Functions

The basic syntax of creating a function:

```
function functionName( ) {
// Code t o be executed
}
```

The declaration of a user-defined function start with the word function, followed by the name of the function you want to create followed by parentheses i.e. ( ) and finally place your function's code between curly brackets { } .

```
<?php
// Defining function
function sample( )
{
echo " Today i s " . date( 'l' , mktime( ) ) ;
} // Calling f unction
sample( ) ;
?>
```

3.7.3 Functions with Parameters

To specify the parameters when you define function to accept input values at run time. The parameters work like placeholder variables within a function; they're replaced at run time by the values (known as argument) provided to the function at the time of invocation.

Syntax:

```
function myFunc($oneParameter, $anotherParameter){
// Code to be executed
}
```

Example:

```
<?php
// Defining f unction
function get Sum( $num1, $num2)
```

```
{
$sum = $num1 + $num2;
echo " Sum of t he two number s $num1 and $num2 i s : $sum" ;
} // Calling function
get Sum( 10, 20) ; ?>
```

The output of the above code will be:

Sum of the two numbers 10 and 20 is : 30

3.7.4 Returning Values from a Function

A function can return a value back to the script that called the function using the return statement. The value may be of any type, including arrays and objects.

Example

Run this code »

```
<?php
// Defining function
function getSum($num1, $num2)
{
$total = $num1 + $num2;
return $total;
}
// Printing returned value
echo getSum(5, 10); // Outputs: 15
?>
```

3.7.5 Passing Arguments to a Function by Reference

In PHP there are two ways you can pass arguments to a function: by value and by reference. By default, function arguments are passed by value so that if the value of the argument within the function is changed, it does not get affected outside of the function. However, to allow a function to modify its arguments, they must be passed by reference.

Example

```
<?php
/* Defining a function that multiply a number
by itself and return the new value */
function sample(&$number){
$number *= $number;
return $number;
}
$mynum = 5;
echo $mynum; // Outputs: 5
sample($mynum);
echo $mynum; // Outputs: 25
?>
```

3.8 Object oriented programming

The Object Oriented concepts in PHP are(oop)

- Class.
- Objects.
- Inheritance.
- Interface.
- Abstraction.
- Magic Methods.

 Class – This is a programmer defined data type, which includes local functions as well as local data. You can think of a class as a template for making many instances of the same kind (or class) of object.
 Object – An individual instance of the data structure defined by a class. You define a class once and then make many objects that belong to it. Objects are also known as instance.

 Member Variable – These are the variables defined inside a class. This data will be invisible to the outside of the class and can be accessed via member functions. These variables are called attribute of the object once an object is created.

 Member function – These are the function defined inside a class and are used to access object data.

 Inheritance – When a class is defined by inheriting existing function of a parent class then it is called inheritance. Here child class will inherit all or few member functions and variables of a parent class.

 Parent class − A class that is inherited from by another class. This is also called a base class or super class.

 Child Class – A class that inherits from another class. This is also called a subclass or derived class.

 Polymorphism – This is an object oriented concept where same function can be used for different purposes. For example function name will remain same but it take different number of arguments and can do different task.

 Overloading − a type of polymorphism in which some or all of operators have different implementations depending on the types of their arguments. Similarly functions can also be overloaded with different implementation.

 Data Abstraction – Any representation of data in which the implementation details are hidden (abstracted).

 Encapsulation – refers to a concept where we encapsulate all the data and member functions together to form an object.
 Constructor − refers to a special type of function which will be called automatically whenever there is an object formation from a class.

 Destructor − refers to a special type of function which will be called automatically whenever an object is deleted or goes out of scope.

PHP Class

Class is consist of properties and methods. Below is a PHP class. In this simple class. **$postCode** is a property and **ringBell()** is a method. They are all prefixed with a visibility keyword (public).

```
Class House {
public $postCode = “560121”;
public function ringBell() {
echo “Ding Dang Dong”;
}
}
```

To instantiate an object of a class, use the keyword new as below:

```
$house = new House();
```

Inheritance

It lets subclass inherits characteristics of the parent class. Parent class decides what and how its properties/methods to be inherited by declared visibility.

```
class Shape {
public function name() {
echo "I am a shape";
}
}
class Circle extends Shape {
}
$circle = new Circle();
$circle->name(); // I am a shape
```

The key word here is **extends**. When **Circle** class extends from **Shape** class, it inherits all of the public and protected methods as well as properties from **Shape** class.

Polymorphism

The provision of a single interface to entities of different types. Basically it means PHP is able to process objects differently depending on their data type or class. This powerful feature allows you to write interchangeable objects that sharing the same interface.

```
interface Shape {
public function name();
}
class Circle implements Shape {
public function name() {
echo "I am a circle";
}
}
class Triangle implements Shape {
public function name() {
echo "I am a triangle";
}
}
function test(Shape $shape) {
$shape->name();
}
test(new Circle()); // I am a circle
test(new Triangle()); // I am a triangle
```

Above example, **test(Shape $shape)** function declares(type hints) its only parameter to be **Shape** type. This function is not aware of Circle and Triangle classes. When either class is passed to this function as a parameter, it processes respectively.

Interface vs Abstract class

Interface

Interface declares what methods a class must have without having to implement them. Any class that implements the interface will have to implement details of those declared methods. Interface is not a class, so you can not instantiate an interface. It is useful when you need to enforce some classes to do something.

```
interface Vehicle {
```

```
public function startEngine();
}
class Car implements Vehicle {
public function startEngine() {
echo "Engine Started";
}
}
```

Vehicle is an interface with a declared method **startEngine()**. **Car** implements **Vechicle**, so it has to implement what **startEngine()** method does.

Abstract class

Abstract class is able to enforce subclasses to implement methods similar to interface. When a method is declared as **abstract** in an abstract class, its derived classes must implement that method.

However it is very different from interface. You can have normal properties and methods as a normal class, because it is in fact a class, so it can be instantiated as a normal class.

```
abstract class Vehicle {
abstract public function startEngine();
public function stopEngine() {
echo "Engine stoped";
}
}
class Car extends Vehicle {
public function startEngine()
echo "Engine Started";
}
}
```

Vehicle is an abstract class. **Car** extends **Vechicle**, so it has to implement what **startEngine()** method does, because this method is declared as abstract. However **Car** does not have to anything with method **stopEngine()**, it is inherited as a normal class does.

3.8.1 Defining PHP Classes

The general form for defining a new class in PHP is as follows

```
<?php
```

```
cl ass phpCl ass {
var $var 1;
var $var 2 = " const ant st r i ng" ;
f unct i on myf unc ( $ar g1, $ar g2) {
[ . . ]
}
[ . . ]
}
?>
```

3.8.2 Creating Objects in PHP

Once you defined your class, then you can create as many objects as you like of that class type.
Following is an example of how to create object using newoperator.
$physics = new Books;
$maths = new Books;
$chemistry = new Books;
Here we have created three objects and these objects are independent of each other and they will have their existence separately. Next we will see how to access member function and process member variables.

3.8.3 Constructor Functions

Constructor Functions are special type of functions which are called automatically whenever an object is created. So we take full advantage of this behaviour, by initializing many things through constructor functions.
PHP provides a special function called __constr uct() to define a constructor. You can pass as many as arguments you like into the constructor function.
Following example will create one constructor for Books class and it will initialize price and title for the book at the time of object creation.

```
function __construct( $par1, $par2 ) {
$this>
title = $par1;
$this>
price = $par2;
}
```

3.8.4 Destructor

Like a constructor function you can define a destructor function using function__destr uct(). You can release all the resources within
a destructor.

3.9 Manipulating PHP Strings

PHP provides many built-in functions for manipulating strings like calculating the length of a string, find substrings or characters, replacing part of a string with different characters, take a string apart, and many others. Here are the examples of some of these functions.

3.9.1 Calculating the Length of a String

The strlen() function is used to calculate the number of characters inside a string. It also includes the blank spaces inside the string.

Example

Run this code »

```
<?php
$my_str = 'Welcome to Tutorial Republic';

// Outputs: 28
echo strlen($my_str);
?>
```

3.9.2 Counting Number of Words in a String

The str_word_count () function counts the number of words in a string.

```
<?php
$my_str = 'The quick brown fox jumps over the lazy dog.';

// Outputs: 9
echo str_word_count($my_str);
?>
```

3.9.3 Replacing Text within Strings

The str_replace() replaces all occurrences of the search text within the target string.

```
<?php
$my_str = 'If the facts do not fit the theory, change the facts.';

// Display replaced string
echo str_replace("facts", "truth", $my_str);
?>
```

The output of the above code will be:

If the truth do not fit the theory, change the truth.

You can optionally pass the fourth argument to the str_replace() function to know how many times the string replacements was performed, like this.

```
<?php
$my_str = 'If the facts do not fit the theory, change the facts.';

// Perform string replacement
str_replace("facts", "truth", $my_str, $count);

// Display number of replacements performed
echo "The text was replaced $count times.";
?>
```

The output of the above code will be:

The text was replaced 2 times.

3.9.4 Reversing a String

The strrev () function reverses a string.

```
<?php
$my_str = 'You can do anything, but not everything.';
// Display reversed string
echo strrev($my_str);
?>
```

The output of the above code will be:

.gnihtyreve ton tub ,gnihtyna od nac uoY

3.10 What is a Regular Expressions?

Regular expressions are powerful pattern matching algorithm that can be performed in a single expression.

Regular expressions use arithmetic operators such as (+,-,^) to create complex expressions.

Regular expressions help you accomplish tasks such as validating email addresses, IP address etc.

3.10.1 Regular expressions in PHP

PHP has built in functions that allow us to work with regular functions. Let’s now look at the commonly used regular expression functions in PHP.

- preg_match – this function is used to perform a pattern match on a string. It returns true if a match is found and false if a match is not found.

- preg_split – this function is used to perform a pattern match on a string and then split the results into a numeric array
- preg_replace – this function is used to perform a pattern match on a string and then replace the match with the specified text.

Below is the syntax for a regular expression function such as preg_match,preg_split or preg_replace.

```
<?php
function_name('/pattern/',subject);
?>
```

HERE,

- "function_name(...)" is either preg_match, preg_split or preg_replace.
- "/.../" The forward slashes denote the beginning and end of our regular expression
- "'/pattern/'" is the pattern that we need to matched
- "subject" is the text string to be matched against

Let's now look at practical examples that implement the above regular expression functions in PHP.

PHP Preg_match

The first example uses the preg_match function to perform a simple pattern match for the word guru in a given URL.

The code below shows the implementation for the above example.

```
<?php
$my_url = "www.guru99.com";
if (preg_match("/guru/", $my_url))
{
echo "the url $my_url contains guru";
}
else
{
echo "the url $my_url does not contain guru";
}
?>
```

python

Unit - IV

Python

Course outcome:To develop the ability to write database applications in ***Python***

4.1 INTRODUCTION:

Python is a powerful modern computer programming language. It bears some similarities to FORTRAN, one of the earliest programming languages, but it is much more powerful than FORTRAN. Python allows you to use variables without declaring them (i.e., it determines types implicitly), and it relies on indentation as a control structure. You are not forced to define classes in python (unlike java) but you are free to do so when convenient. Python was developed by

4.2 PYTHON SYNTAX

First Python Program

Let us execute programs in different modes of programming.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt –

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Type the following text at the Python prompt and press the Enter −

```
>>> print "Hello, Python!"
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Python files have extension **.py**. Type the following source code in a test.py file −print "Hello, Python!"

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows −

```
$ python test.py
```

This produces the following result −

```
Hello, Python!
```

Let us try another way to execute a Python script. Here is the modified test.py file −

Live Demo

```
#!/usr/bin/python

print "Hello, Python!"
```

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows −

```
$ chmod +x test.py     # This is to make file executable
$./test.py
```

This produces the following result −

```
Hello, Python!
```

4.3 Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.

- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

4.4 Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

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

4.5 Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

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

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

However, the following block generates an error −

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

```
print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks −

Note − Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.

```
#!/usr/bin/python

import sys

try:
# open file stream
file = open(file_name, "w")
except IOError:
print "There was an error writing to", file_name
sys.exit()
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
file_text = raw_input("Enter text: ")
if file_text == file_finish:
# close the file
file.close
break
file.write(file_text)
file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
print "Next time please enter something"
sys.exit()
try:
file = open(file_name, "r")
except IOError:
print "There was an error reading file"
sys.exit()
file_text = file.read()
file.close()
print file_text
```

4.6 Multi-Line Statements

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

```
total = item_one + \
item_two + \
```

```
item_three
```

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

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

4.6.1 Quotation in Python

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

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

4.6.2 Comments in Python

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

Live Demo

```
#!/usr/bin/python

# First comment
print "Hello, Python!" # second comment
```

This produces the following result −

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression −

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows −

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

```
'''
This is a multiline comment.
```

'''

4.6.3 Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it. In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

4.7 Python numbers

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 = 2.5×10^2 = 250).

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

4.8 Python strings

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 −

Live Demo

```
#!/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 −

Live Demo

```
#!/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

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

Built-in String Methods

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

Sr.No.	Methods with Description
1	capitalize() Capitalizes first letter of string
2	center(width, fillchar) Returns a space-padded string with the original string centered to a total of width columns.
3	count(str, beg= 0,end=len(string))

4.9 Python lists

The list is a most versatile data type 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"]
```

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 −

Live Demo

```
#!/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
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 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[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 chapter.

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

Built-in List Functions & Methods

Python includes the following list functions −

Sr.No.	Function with Description
1	cmp(list1, list2) Compares elements of both lists.
2	len(list) Gives the total length of the list.

4.10 Python tuples

A tuple is a sequence of immutable Python objects. 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 −

Live Demo

```
#!/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 −

Live Demo

```
#!/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 −

```
Live Demo
#!/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 chapter −

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	cmp(tuple1, tuple2) Compares elements of both tuples.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

4.11 Python dictionary

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 −

Live Demo

```
#!/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 −

Live Demo

```
#!/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 −

Live Demo

```
#!/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'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # 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 −

```
Live Demo
#!/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 −

```
Live Demo
#!/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	cmp(dict1, dict2) Compares elements of both dict.
2	len(dict) Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.

4.12 PYTHON CONDITIONALS

4.12.1 Conditional Statements in Python

In programming languages, most of the time we have to control the flow of execution of your program,.You want to execute some set of statements only if the given condition is

satisfied, and a different set of statements when it's not satisfied. We call it as control statements or decision making statements. We use these statements when we want to execute a block of code when the given condition is true or false.

In Python we can achieve decision making by using the below statements:

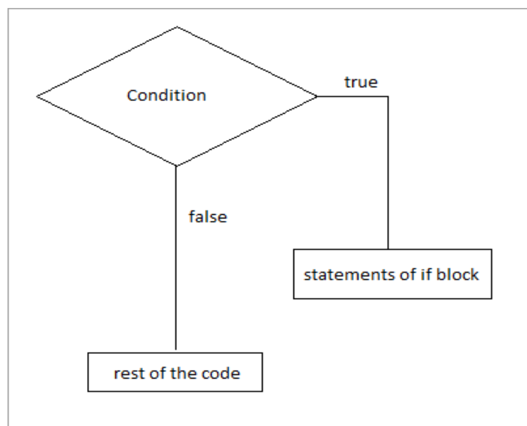
- If statements
- If-else statements
- Elif statements
- Nested if and if-else statements
- Elif ladder

1)If statements

If statement is one of the most commonly used conditional statement in most of the programming languages. It decides whether certain statements need to be executed or not. If statement checks for a given condition, if the condition is true, then the set of code present inside the if block will be executed. The If condition evaluates a Boolean expression and executes the block of code only when the Boolean expression becomes TRUE.

Syntax:
If (Boolean expression): Block of code #Set of statements to execute if the condition is true

Here, the condition will be evaluated to a Boolean expression (true or false). If the condition is true, then the statement or program present inside the if block will be executed and if the condition is false, then the statements or program present inside the if block will not be execu



If you observe the above flow-chart, first the controller will come to an if condition and evaluate the condition if it is true, then the statements will be executed, otherwise the code present outside the block will be executed.

Let's see some examples on if statements.

<u>Example: 1</u>

1 Num = 5

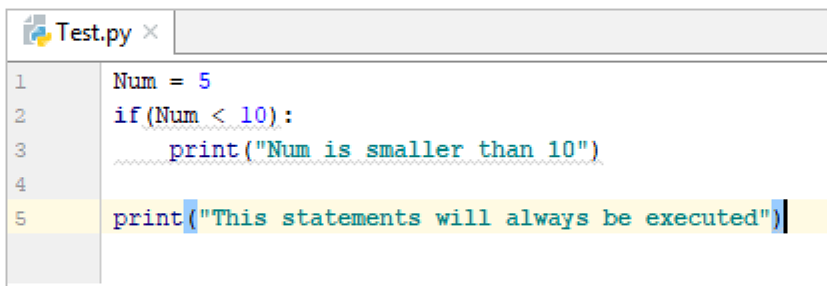
2 If(Num < 10):

3 print("Num is smaller than 10")

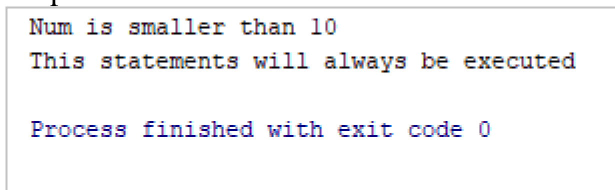
4

5 print("This statements will always be executed")

Output: Num is smaller than 10.
This statements will always be executed.



Output:



In the above example, we declared a variable called 'Num' with the value as 5 and in the if statement we are checking if the number is lesser than 10 or not if the condition is true then a set of statements inside the if block will be executed.

2) If-else statements

The statement itself tells that if a given condition is true then execute the statements present inside if block and if the condition is false then execute the else block.

Else block will execute only when the condition becomes false, this is the block where you will perform some actions when the condition is not true.

If-else statement evaluates the Boolean expression and executes the block of code present inside the if block if the condition becomes TRUE and executes a block of code present in the else b

lock if the condition becomes FALSE.

Syntax:
if(Boolean expression):

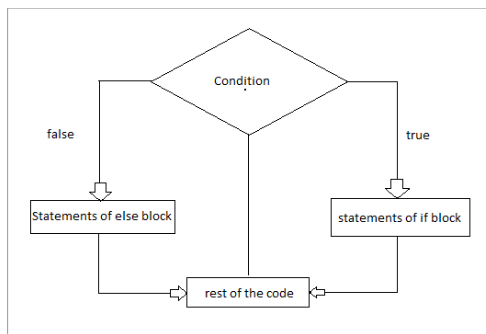
Block of code #Set of statements to execute if condition is true

else:

Block of code #Set of statements to execute if condition is false

Here, the condition will be evaluated to a Boolean expression (true or false). If the condition is true then the statements or program present inside the if block will be executed and if the condition is false then the statements or program present inside else block will be executed.

Let's see the flowchart of if-else



If you observe the above flow chart, first the controller will come to if condition and evaluate the condition if it is true and then the statements of if block will be executed otherwise else block will be executed and later the rest of the code present outside if-else block will be executed.

Example: 1

1 num = 5

2 if(num > 10):

3 print("number is greater than 10")

4 else:

5 print("number is less than 10")

6

7 print("This statement will always be executed")

Output:
number is less than 10.

This statement will always be executed.

```
Test.py
1  num = 5
2  if(num > 10):
3      print("number is greater than 10")
4  else:
5      print("number is less than 10")
6
7  print("This statement will always be executed")
8
```

Output:

```
number is less than 10
This statement will always be executed

Process finished with exit code 0
```

In the above example, we have declared a variable called 'num' with the value as 5 and in the if statement we are checking if the number is greater than 5 or not.

If the number is greater than 5 then, the block of code inside the if block will be executed and if the condition fails then the block of code present inside the else block will be executed

Elif statements

In python, we have one more conditional statement called elif statements. Elif statement is used to check multiple conditions only if the given if condition false. It's similar to an if-else statement and the only difference is that in else we will not check the condition but in elif we will do check the condition.

Elif statements are similar to if-else statements but elif statements evaluate multiple conditions.

Syntax:
if (condition):

#Set of statement to execute if condition is true

elif (condition):

#Set of statements to be executed when if condition is false and elif condition is true

else:

#Set of statement to be executed when both if and elif conditions are false

<u>Example: 1</u>

1 num = 10

2 if (num == 0):

3 print("Number is Zero")

4

5 elif (num > 5):

6 print("Number is greater than 5")

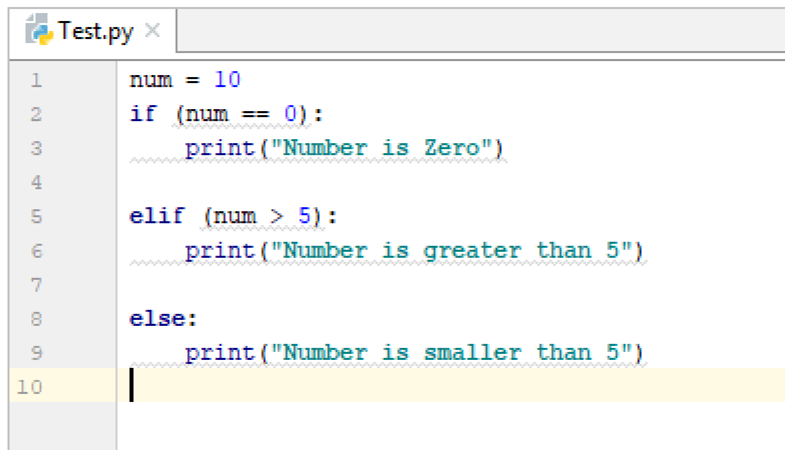
7

8 else:

9 print("Number is smaller than 5")

Output:

Number is greater than 5



Output:

```
Number is greater than 5

Process finished with exit code 0
```

In the above example we have declared a variable called 'num' with the value as 10, and in the if statement we are checking the condition if the condition becomes true. Then the block of code present inside the if condition will be executed.

If the condition becomes false then it will check the elif condition if the condition becomes true, then a block of code present inside the elif statement will be executed.

If it is false then a block of code present inside the else statement will be executed.

Nested if-else statements

Nested if-else statements mean that an if statement or if-else statement is present inside another if or if-else block. Python provides this feature as well, this in turn will help us to check multiple conditions in a given program.

An if statement present inside another if statement which is present inside another if statements and so on.

Nested if Syntax:

if(condition):

#Statements to execute if condition is true

if(condition):

#Statements to execute if condition is true

#end of nested if

#end of if

The above syntax clearly says that the if block will contain another if block in it and so on. If block can contain 'n' number of if block inside it.

Example: 1

```
1 num = 5
2 if(num >0):
3 print("number is positive")
4
5 if(num<10):
6 print("number is less than 10")
```

Output:

number is positive

number is less than 10

Nested if Syntax:

if(condition):

#Statements to execute if condition is true

if(condition):

#Statements to execute if condition is true

#end of nested if

#end of if

The above syntax clearly says that the if block will contain another if block in it and so on. If block can contain ‘n' number of if block inside it.

Example: 1

1 num = 5

2 if(num >0):

3 print(“number is positive”)

4

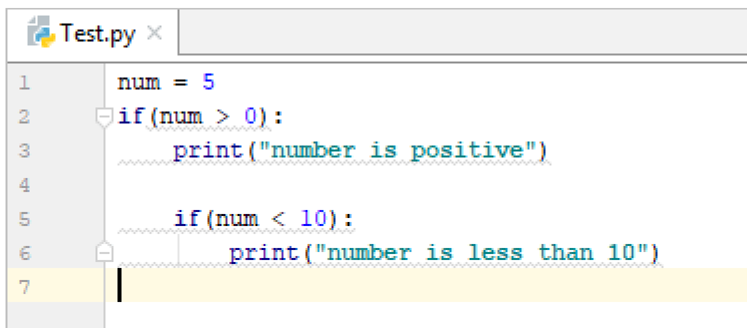
5 if(num<10):

6 print(“number is less than 10”)

Output:

number is positive

number is less than 10



Nested if-else Syntax:

if(condition):

#Statements to execute if condition is true

if(condition):

#Statements to execute if condition is true

else:

#Statements to execute if condition is false

else:

#Statements to execute if condition is false

Here we have included if-else block inside an if block, you can also include if-else block inside else block.

Example: 3

```
1 num = -7
2 if (num != 0):
3 if (num > 0):
4 print("Number is positive")
5 else:
6 print("Number is negative")
7 else:
8 print("Number is Zero")
```

Output:

Number is negative

elif Ladder

We have seen about the elif statements but what is this elif ladder. As the name itself suggests a program which contains ladder of elif statements or elif statements which are structured in the form of a ladder.

This statement is used to test multiple expressions.

Syntax:

if (condition):

#Set of statement to execute if condition is true

elif (condition):

#Set of statements to be executed when if condition is false and elif condition is true

elif (condition):

#Set of statements to be executed when both if and first elif condition is false and second elif condition is true

elif (condition):

#Set of statements to be executed when if, first elif and second elif conditions are false and third elif statement is true

else:

#Set of statement to be executed when all if and elif conditions are false

Example: 1

```
1 my_marks = 89
2 if (my_marks &lt; 35):
3 print(“Sorry!!!, You are failed in the exam”)
4 elif(my_marks  &lt; 60): print(“Passed in Second class”) elif(my_marks  &gt; 60 andmy_marks
  &lt; 85):
5 print(“Passed in First class”)
6 else:
7 print(“Passed in First class with distinction”)
```

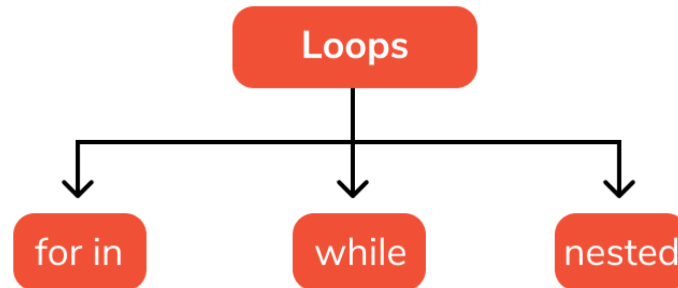
Output:
Passed in First class with distinction

The above example describes the elif ladder. Firstly the control enters the if statement and evaluates the condition if the condition is true then the set of statements present inside the if block will be executed else it will be skipped and the controller will come to the first elif block and evaluate the condition.

The similar process will continue for all the remaining elif statements and in case all if and elif conditions are evaluated to false then the else block will be executed.

Looping Statements in Python

Looping statements in python are used to execute a block of statements or code repeatedly for several times as specified by the user.



Python provides us with 2 types of loops as stated below:

- While loop
- For loop

While loop:

While loop in python is used to execute multiple statement or codes repeatedly until the given condition is true. We use while loop when we don't know the number of times to iterate.

Syntax:

while (expression): block of statements Increment or decrement operator

In while loop, we check the expression, if the expression becomes true, only then the block of statements present inside the while loop will be executed. For every iteration, it will check the condition and execute the block of statements until the condition becomes false.

Example:

```
1 number = 5
2 sum = 0
3 i = 0
4
5 while (i<number)
  :
6 sum = sum + i
7 i = i+1
8
9 print(sum)
```

Output:

10

For loop:

For loop in python is used to execute a block of statements or code several times until the given condition becomes false.

We use for loop when we know the number of times to iterate.

Syntax:
for var in sequence: Block of code
Here var will take the value from the sequence and execute it until all the values in the sequence are done.

Example:

```
1 language = [‘Python’, ‘Java’, ‘Ruby’]
2
3 for lang in language:
4 print(“Current language is: “, lang)
```

Output:
Current language is: Python

Current language is: Java

Current language is: Ruby

For loop using range () function:
Range () function is used to generate a sequence of numbers.

For Example, range (5) will generate numbers from 0 to 4 (5 numbers).

Example:

```
1 language = [‘Python’, ‘Java’, ‘Ruby’]
2
3 for lang in range(len(language)):
4 print(“Current language is: “, language[lang])
```

Output:
Current language is: Python

Current language is: Java

Current language is: Ruby

Python files

Printing to the Screen

The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows −

```
#!/usr/bin/python

print "Python is really a great language,", "isn't it?"
```

This produces the following result on your standard screen −

```
Python is really a great language, isn't it?
```

The *raw_input* Function

The *raw_input([prompt])* function reads one line from standard input and returns it as a string (removing the trailing newline).

```
#!/usr/bin/python

str = raw_input("Enter your input: ")
print "Received input is : ", str
```

This prompts you to enter any string and it would display same string on the screen. When I typed "Hello Python!", its output is like this −

```
Enter your input: Hello Python
Received input is :  Hello Python
```

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

The *open* Function

Before you can read or write a file, you have to open it using Python's built-in *open()*function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

file object = open(file_name [, access_mode][, buffering])

Here are parameter details −

- **file_name** − The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Here is a list of the different modes of opening a file −

Sr.No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.

The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close()
```

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

This produces the following result −

```
Name of the file:  foo.txt
```

Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

The *write()* Method

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string −

Syntax

```
fileObject.write(string)
```

Here, passed parameter is the content to be written into the opened file.

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opend file
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

```
Python is a great language.
Yeah its great!!
```

The *read()* Method

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

```
fileObject.read([count])
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

Let's take a file *foo.txt*, which we created above.

```
#!/usr/bin/python
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend file
fo.close()
```

This produces the following result −

```
Read String is :  Python is
```

Renaming and Deleting Files

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

The rename() Method

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_file_name, new_file_name)
```

Example

Following is the example to rename an existing file *test1.txt* −

```
#!/usr/bin/python
import os

# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

The *remove()* Method

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

```
os.remove(file_name)
```

Example

Following is the example to delete an existing file *test2.txt* −

```
#!/usr/bin/python
import os

# Delete file test2.txt
os.remove("text2.txt")
```

Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>>
>>> while True print('Hello world')
File "<stdin>", line 1
while True print('Hello world')
^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>>
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-Cor whatever the operating system supports); note that a user-generated interruption is signalled by raising the KeyboardInterrupt exception.

```
>>>
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B(Exception):
pass

class C(B):
```

```
pass

class D(C):
pass

for cls in [B, C, D]:
try:
raise cls()
except D:
print("D")
except C:
print("C")
except B:
print("B")
```

Note that if the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
import sys

try:
f = open('myfile.txt')
s = f.readline()
i = int(s.strip())
except OSError as err:
print("OS error: {0}".format(err))
except ValueError:
print("Could not convert data to an integer.")
except:
print("Unexpected error:", sys.exc_info()[0])
raise
```

The try … except statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
try:
f = open(arg, 'r')
except OSError:
```

```
print('cannot open', arg)
else:
print(arg, 'has', len(f.readlines()), 'lines')
f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines__str__() so the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>>
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
>>>
>>> def this_fails():
```

```
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

Python functions

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 −

Live Demo

```
#!/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.

Live Demo

```
#!/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 assig 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 −

Live Demo

```
#!/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 −

Live Demo

```
#!/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.

Live Demo

```
#!/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 −

Live Demo

```
#!/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 −

Live Demo

```
#!/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 *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 −

Live Demo

```
#!/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
```

Python module

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.

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

print Money
AddMoney()
print Money
```

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.

What Is Object-Oriented Programming (OOP)?

Object-oriented Programming, or *OOP* for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual *objects*.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Classes in Python

Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an Animal() class to track properties about the Animal like the name and age.

It’s important to note that a class just provides structure—it’s a blueprint for how something should be defined, but it doesn’t actually provide any real content itself. The Animal() class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal’s name or age is.

It may help to think of a class as an *idea* for how something should be defined

How To Define a Class in Python

Defining a class is simple in Python:

```
class Dog:
pass
```

You start with the class keyword to indicate that you are creating a class, then you add the name of the class (using CamelCase notation, starting with a capital letter.)

Also, we used the Python keyword pass here. This is very often used as a place holder where code will eventually go. It allows us to run this code without throwing an error.

Note: The above code is correct on Python 3. On Python 2.x ("legacy Python") you'd use a slightly different class definition:

```
# Python 2.x Class Definition:
class Dog(object):
pass
```

The (object) part in parentheses specifies the parent class that you are inheriting from (more on this below.) In Python 3 this is no longer necessary because it is the implicit default.

Instance Attributes

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the __init__() method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). This method must have at least one argument as well as the self variable, which refers to the object itself (e.g., Dog).

```
class Dog:


# Initializer / Instance Attributes
def __init__(self, name, age):
self.name = name
self.age = age
```

In the case of our Dog() class, each dog has a specific name and age, which is obviously important to know for when you start actually creating different dogs. Remember: the class is just for defining the Dog, not actually creating *instances* of individual dogs with specific names and ages; we'll get to that shortly.

Similarly, the self variable is also an instance of the class. Since instances of a class have varying values we could state Dog.name = name rather than self.name = name. But since not all dogs share the same name, we need to be able to assign different values to different instances. Hence the need for the special self variable, which will help to keep track of individual instances of each class.

NOTE: You will never have to call the __init__() method; it gets called automatically when you create a new 'Dog' instance.

Class Attributes

While instance attributes are specific to each object, class attributes are the same for all instances—which in this case is *all* dogs.

```
class Dog:

# Class Attribute
species = 'mammal'

# Initializer / Instance Attributes
def __init__(self, name, age):
self.name = name
self.age = age
```

So while each dog has a unique name and age, every dog will be a mammal.

Let's create some dogs…

Instantiating Objects

Instantiating is a fancy term for creating a new, unique instance of a class.

For example:

```
>>>

>>> class Dog:
...     pass
...
>>> Dog()
<__main__.Dog object at 0x1004ccc50>
>>> Dog()
<__main__.Dog object at 0x1004ccc90>
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

We started by defining a new Dog() class, then created two new dogs, each assigned to different objects. So, to create an instance of a class, you use the the class name, followed by parentheses. Then to demonstrate that each instance is actually different, we instantiated two more dogs, assigning each to a variable, then tested if those variables are equal.

What do you think the type of a class instance is?

```
>>>
```

```
>>> class Dog:
...     pass
...
>>> a = Dog()
>>> type(a)
<class '__main__.Dog'>
```

Let's look at a slightly more complex example…

```
class Dog:

# Class Attribute
species = 'mammal'

# Initializer / Instance Attributes
def __init__(self, name, age):
self.name = name
self.age = age



# Instantiate the Dog object
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 6)

# Access the instance attributes
print("{} is {} and {} is {}.".format(
philo.name, philo.age, mikey.name, mikey.age))

# Is Philo a mammal?
if philo.species == "mammal":
print("{0} is a {1}!".format(philo.name, philo.species))
```

NOTE: Notice how we use dot notation to access attributes from each object.
Save this as *dog_class.py*, then run the program. You should see:

```
Philo is 5 and Mikey is 6.
```

```
Philo is a mammal!
```

Instance Methods

Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to perform operations with the attributes of our objects. Like the __init__ method, the first argument is always self:

```
class Dog:

# Class Attribute
species = 'mammal'

# Initializer / Instance Attributes
def __init__(self, name, age):
self.name = name
self.age = age

# instance method
def description(self):
return "{} is {} years old".format(self.name, self.age)

# instance method
def speak(self, sound):
return "{} says {}".format(self.name, sound)

# Instantiate the Dog object
mikey = Dog("Mikey", 6)

# call our instance methods
print(mikey.description())
print(mikey.speak("Gruff Gruff"))
```

Save this as *dog_instance_methods.py*, then run it:

```
Mikey is 6 years old
Mikey says Gruff Gruff
```

In the latter method, speak(), we are defining behavior. What other behaviors could you assign to a dog? Look back to the beginning paragraph to see some example behaviors for other objects.

Modifying Attributes

You can change the value of attributes based on some behavior:

>>>

```
>>> class Email:
...     def __init__(self):
...         self.is_sent = False
...     def send_email(self):
...         self.is_sent = True
...
>>> my_email = Email()
>>> my_email.is_sent
False
>>> my_email.send_email()
>>> my_email.is_sent
True
```

Here, we added a method to send an email, which updates the is_sent variable to True.

Python Object Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called *child classes*, and the classes that child classes are derived from are called *parent classes*.

It's important to note that child classes override *or* extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an object, which generally all other classes inherit as their parent.

When you define a new class, Python 3 it implicitly uses object as the parent class. So the following two definitions are equivalent:

```
class Dog(object):
pass

# In Python 3, this is the same as:

class Dog:
pass
```

perl

Programming Language

Unit -5

Perl

Course outcome: Describe the features of Perl Programming Language and to Implement Basic I/O operations.

5.1 What is Perl?

- Perl is a stable, cross platform programming language.
- Though Perl is not officially an acronym but few people used it as **Practical Extraction and Report Language**.
- It is used for mission critical projects in the public and private sectors.
- Perl is an *Open Source* software, licensed under its *Artistic License*, or the *GNU General Public License (GPL)*.
- Perl was created by Larry Wall.
- Perl 1.0 was released to usenet's alt.comp.sources in 1987.
- At the time of writing this tutorial, the latest version of perl was 5.16.2.
- Perl is listed in the *Oxford English Dictionary*.

PC Magazine announced Perl as the finalist for its 1998 Technical Excellence Award in the Development Tool category.

PERL is a general purpose programming language originally developed for text manipulation and now used for a wide range of tasks including web development, system administration, network programming, code generation and more.

Supports both procedural and oo programming . It has powered built in support for text processing . It has one of the world's most impressive collections of third party modules.

5.2 Advantages of Perl

- Quick development phases
- Open source and free licencing
- Excellent text handling and regular expressions
- Large experienced active user base
- Fast, for an interpreted language
- Code developed is generally cross platform
- Very easy to write powerful programs in a few lines of code.

5.3 Disadvantages of Perl

- Limited GUI support
- Can look complicated, initially, particularly if you're not familiar with regular expression.

5.4 PERL OVERVIEW

Perl is a programming language. In the web development environment, Perl is a very powerful way to create dynamic web pages. Perl can be used for virtually any web application you can think of, from hit counters to database management.

To use Perl, you write a *script*. A script is basically a simple computer program which runs on the server and produces web pages. The script can take input from the user or other source and customise the resulting web pages based on the results. For example, a script could ask for a password and provide an error page if the password is incorrect.

The process of creating a Perl script goes like this:

- Write a script in the Perl language
- Upload the script to a place in your website which allows scripts to run (typically the cgi-bin)
- Set the permissions of the script to allow anyone to run it

Example Script

The example below is a simple Perl script. When run, this produces a web page which says "Hello World!".

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";
print "<html><head><title>Test Page</title></head><body>";
print "Hello World!";
print "</body></html>";
```

5.5 History

Perl was created in the mid-1980s by Larry Wall. Its original intent was to fill in the gap between so-called "low-level" languages like Cand C++, and the higher level scripting languages like *awk*, *sed*, and shell scripts. It has evolved considerably since then, becoming the de facto language of choice for many *nix system administrators and power users. It has found broad appeal on the Internet (a mostly *nix based network) as the background code behind many sites on the World Wide Web. A notable example is Slashdot.

5.6 Perl Features

- Perl takes the best features from other languages, such as C, awk, sed, sh, and BASIC, among others.
- Perls database integration interface DBI supports third-party databases including Oracle, Sybase, Postgres, MySQL and others.
- Perl works with HTML, XML, and other mark-up languages.
- Perl supports Unicode.
- Perl is Y2K compliant.
- Perl supports both procedural and object-oriented programming.
- Perl interfaces with external C/C++ libraries through XS or SWIG.
- Perl is extensible. There are over 20,000 third party modules available from the Comprehensive Perl Archive Network (CPAN).

The Perl interpreter can be embedded into other systems

Running Perl programs

```
1. perl progname.pl
```

Alternatively, put this as the first line of your script:

```
1. #!/usr/bin/env perl
```

... and run the script as */path/to/script.pl*. Of course, it'll need to be executable first, so `chmod 755 script.pl` (under Unix).

(This start line assumes you have the **env** program. You can also put directly the path to your perl executable, like in `#!/usr/bin/perl`).

For more information, including instructions for other platforms such as Windows and Mac OS, read perlrun

Basic syntax overview

A Perl script or program consists of one or more statements. These statements are simply written in the script in a straightforward fashion. There is no need to have a `main()` function or anything of that kind.

Perl statements end in a semi-colon:

```
1. print "Hello, world";
```

Comments start with a hash symbol and runline

```
1. # This is a comment
```

Whitespace is irrelevant:

```
1. print
2. "Hello, world"
3. ;
```

... except inside quoted strings:

```
1. # this would print with a linebreak in the middle
2. print "Hello
3. world";
```

Double quotes or single quotes may be used around literal strings:

```
1. print "Hello, world";
2. print 'Hello, world';
```

However, only double quotes "interpolate" variables and special characters such as newlines (`\n`):

```
1. print "Hello, $name\n";     # works fine
2. print 'Hello, $name\n';     # prints $name\n literally
```

Numbers don't need quotes around them:

```
1. print 42;
```

You can use parentheses for functions' arguments or omit them according to your personal taste. They are only required occasionally to clarify issues of precedence.

```
1. print("Hello, world\n");
2. print "Hello, world\n";
```

More detailed information about Perl syntax can be found in perlsyn.

Example: Perl hello world

```
#!/usr/bin/perl
print "Hello, world!";
```

Output:

Hello, world!

5.7 Download & Install Perl - Windows, Mac & Linux

How to get Perl?

5.7.1 Need to Install Two Options are available

1. Strawberry Perl is an open source binary distribution of Perl for the Windows OS. It includes a compiler and pre-installed modules that offer the ability to install XS CPAN modules directly from CPAN
2. ActiveState perl provide a binary distribution of Perl (for many platforms), as well as their own Perl package manager (PPM)

5.7.2 Install perl for Windows:

First, download the Active Perl from this link. Follow these steps to install ActivePerl on Windows system. See the below screenshots for the same.

Step 1: Once you download the installer and start the installation you will see the below window, click on next to proceed.

Step 2: Accept Licensing agreement to proceed the installation.

Step 3: Below are different packages that will be installed. By default, all will be selected. The only thing different is PPM (Perl Package Manager). This is the utility provided by Active Perl to install external Perl modules or libraries in your system. Click on Next to proceed.

Step 4: These are different types of Perl extensions that can be used for Perl. Mostly we will be using .Pl, .Plx and .Pm for Perl. Perl modules basically use .Pm as their file extension to refer to a library file. Select all the options and click on the Next button.

Step 5: Click on Install button to proceed with the installation.

Step 6: Once installed, execute the command 'Perl –v' to check whether Perl is successfully installed in your system.

5.7.3 PERL PARSING RULES

Perl has its own set of rules for parsing the perl script,identifying the components and actually executing the script.

The execution process

Perl parses a script is to take a top level look at a perl takes the source text and executes it. It takes raw input , parses for each statement and converts into a series of opcodes

1.Read the source code and parse the contents to verify the source code against the core rules. This is the stage at which external modules are imported.

2.Compile the source into a series of opcodes. This involves the use of a parser which translates the PERL source code.

3.Execute the opcode.

Syntax and parsing rules:

PERL parser consider a number of different items when it takes in a source script for execution. It identifies different elements such as operators,constants and terms and then evaluate to produce a particular result.

The PERL parser examines the following:

1**. Basic syntax:** core layout, line termination etc

2**. Comments**: comments are ignored

3**. Component identity**: individual terms such as variables, functions and constants are identified.

4**. Precedence:** The parsers process the statements according to the precedence rules.

5**. Context:** It evaluates the individual elements in a single statement.

6**. Logic syntax:** The parser treats different values whether constant or variable as true or false value.

Basic syntax overview

A Perl script or program consists of one or more statements. These statements are simply written in the script in a straightforward fashion. There is no need to have a `main()` function or anything of that kind.

Perl statements end in a semi-colon:

```
2. print "Hello, world";
```

Comments start with a hash symbol and run to the end of the line

```
2. # This is a comment
```

Whitespace is irrelevant:

```
4. print
5. "Hello, world"
6. ;
```

... except inside quoted strings:

```
4. # this would print with a linebreak in the middle
5. print "Hello
6. world";
```

Double quotes or single quotes may be used around literal strings:

```
3. print "Hello, world";
4. print 'Hello, world';
```

However, only double quotes "interpolate" variables and special characters such as newlines (`\n`):

```
3. print "Hello, $name\n";     # works fine
4. print 'Hello, $name\n';     # prints $name\n literally
```

Numbers don't need quotes around them:

```
2. print 42;
```

You can use parentheses for functions' arguments or omit them according to your personal taste. They are only required occasionally to clarify issues of precedence.

```
3. print("Hello, world\n");
4. print "Hello, world\n";
```

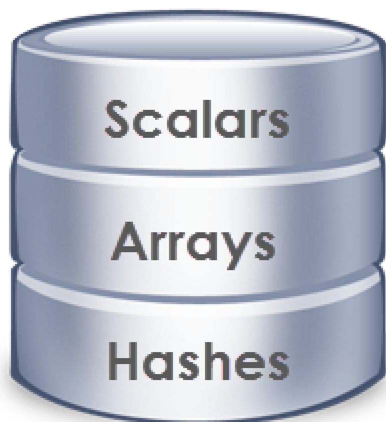
5.8 VARIABLES AND DATA

Variables are core part of any language allow you to store dynamic values into named location.

5.8.1 Perl variable

Now, we'll talk about variables. You can imagine variable like kind of container which holds one or more values. Once defined, the name of variable remains the same, but the value or values change over and over again.

There are 3 Types of variables:



The easiest ones are scalars, and this is ours today subject

5.8.2 Scalar Variable

This type of variable holds a single value.

Its name begins with a dollar sign and a Perl identifier (it's the name of our variable).

5.8.3 Naming Convention

If you are familiar with other programming languages, then you would know that there are certain rules about naming variables. Similarly, Perl has three rules for naming scalars.

1. All scalar names will begin with a $. It is easy is to remember to prefix every name with $. Think of it as a $scalar.
2. Like PHP. after the first character $, which, is special in Perl, alphanumeric characters i.e. a to z, A to Z and 0 to 9 are allowed. Underscore character is also allowed. Use underscore to split the variable names into two words. `But the First character cannot be a number`````

3. Even though numbers can be part of the name, they cannot come immediately after $. This implies that first character after $ will be either an alphabet or the underscore. Those coming from C/C++ background should be immediately able to recognize the similarity. Examples

Perl Example:

```
$var;
$Var32;
$vaRRR43;
$name_underscore_23;
```

These, however, are not legal scalar variable names.

```
mohohoh         # $ character is missing
$               # must be at least one letter
$47x            # second character must be a letter
$variable!      # you can't have a ! in a variable name
```

The general rule says, when Perl has just one of something, that's a scalar. Scalars can be read from devices, and we can use it to our programs.

Two Types of Scalar Data Types

Numbers

Strings

5.8.4 Numbers:

In this type of scalar data we could specify:

- integers, simply it's whole numbers, like 2, 0, 534
- floating-point numbers, it's real numbers, like 3.14, 6.74, 0.333

5.8.5 Integer literals:

It consists of one or more digits, optionally preceded by a plus or minus and containing underscores.

5.8.6 Floating-point literals:

It consists of digits, optionally minus, decimal point and exponent.

5.9 Strings

Strings: It's also very simple type of scalar.

The maximum length of a string in Perl depends upon the amount of memory the computer has. There is no limit to the size of the string, any amount of characters, symbols, or words can make up your strings. The shortest string has no characters. The longest can fill all of the system memory. Perl programs can be written entirely in 7-bit ASCII character set. Perl also permits you to add any 8-bit or 16-bit character set aka. non-ASCII characters within string literals. Perl has also added support for Unicode UTF-8.

Like numbers there are two different types of strings:

- Single quotes string literals
- Double quotes string literals

5.9.1 Single-quoted string literals

Single quotation marks are used to enclose data you want to be taken literally. A short example and everything should be clear:

Perl Examples:

```
#!/usr/bin/perl
$num = 7;
$txt = 'it is $num';
print $txt;
```

OUTPUT:

it is $num

Here due to single quotes value of $num in not taken and the literal characters '$','n', 'u' & 'm' are added to the value of $txt

5.9.2 Double-quoted string literals

Double quotation marks are used to enclose data that needs to be interpolated before processing. That means that escaped characters and variables aren't simply literally inserted into later operations, but are evaluated on the spot. Escape characters can be used to insert newlines, tabs, etc.

Perl Examples:

```
$num = 7;
$txt = "it is $num";
```

```
print $txt;
```

OUTPUT:

it is 7

5.10 Perl Array

What is Perl Array?

An Array is a special type of variable which stores data in the form of a list; each element can be accessed using the index number which will be unique for each and every element. You can store numbers, strings, floating values, etc. in your array. This looks great, So how do we create an array in Perl? In Perl, you can define an array using '@' character followed by the name that you want to give. Let's consider defining an array in Perl.

my @array;

This is how we define an array in Perl; you might be thinking how we need to store data in it. There are different ways of storing data in an array. This depends on how you are going to use it.

```
my @array=(a,b,c,d);
print @array;
```

Output:

abcd

This is an array with 4 elements in it.

The array index starts from 0 and ends to its maximum declared size, in this case, the max index size is 3.

@array= qw(a b c d)

a	b	c	d
Index – 0	1	2	3

Indices are memory locations where data is stored

You can also declare an array in the above way; the only difference is, it stores data into an array considering a white space to be the delimiter. **Here, qw() means quote word.** The significance of this function is to generate a list of words. You can use the qw in multiple ways to declare an array.

```
@array1=qw/a b c d/;
@array2= qw' p q r s';
@array3=qw { v x y z};
print @array1;
```

```
print @array2;
print @array3;
```

Output:

abcdpqrsvxyz

Suppose you want to assign a value to the 5th element of an array, how are we going to do that.

$array [4] ='e';

5.10.1 Sequential Array

Sequential arrays are those where you store data sequentially. Suppose, you want to store 1-10 numbers or alphabets a-z in an array. Instead of typing all the letters, you can try something like below -

```
@numbers= (1..10);
print @numbers;                  #Prints numbers from 1 to 10;
```

Output:

12345678910

5.10.2 Perl Array Size

We have an array which is already available, and you don't know what the size of that array is, so what is the possible way to find that.

```
@array= qw/a b c d e/;
print $size=scalar (@array);
```

Can we get the size of an array without using functions? Yes, we can.

```
@array= qw/a b c d e/;
print $size=scalar (@array);
print "\n";
print $size=$#array + 1;                    # $#array will print the Max Index of the array, which is
 5 in this case
```

Output:

5

5

5.10.3Dynamic Array

The above method of declaring an array is called **static arrays**, where you know the size of an array.

What is Dynamic Array?

Dynamic arrays are those that you declare without specifying any value on them. So when exactly do we store values in that array? Simple, we store them during run time. Here is a simple program for that.

We will be using some inbuilt Perl functions for doing this task.

```
my $string="This is a kind of dynamic array";
my @array;
@array=split('a',$string);
foreach(@array)
{
print "$_ \n”;
# This is a special variable which stores the current value.
}
```

Output:

This is

kind of dyn

mic

rr

y

The split function splits the content of string into an array based on the delimiter provided to it. This function will also eliminate the delimiter from the string, in this case, it is 'a';

5.11 Perl Hashes

What are Hashes?

A hash can also hold as many scalars as the array can hold. The only difference is we don't have any index rather we have keys and values. A hash can be declared, starting with % followed by the name of the hash. Let's see an example how we can define a Perl hash and how we can differentiate this from array

Consider an example of three people and their ages are represented in an array.

```
@array=('Sainath',23,'Krishna',24,'Shruthi',25);        #This is how an array looks.
print @array;
```

Output:

Sainath33Krishna24Shruthi25

5.11.1 Add Perl Hashes

As you can see we already have a hash %newHash, and now we need to add more entries into it.

```
$newHash{'Jim'}=25;
$newHash{'John'}=26;
$newHash{'Harry'}=27;
print %newHash;
```

Output:

Jim25John26Harry27

5.11.2 Perl Delete key

You may want to delete an entry from a hash. This is how we can do that.

```
delete $newHash{'Jim'};#This will delete an entry from the hash.
```

Delete is an inbuilt function of Perl. Here, we will see an example of assigning a hash to an array.

```
@array=%newHash;
print "@array";
```

Note: Whenever you print a hash or when you store hash into an array. The order may always differ. It's not the same always.

We can assign only keys or values of a hash to an array.

```
@arraykeys= keys(%newHash);
@arrayvalues=values(%newHash);
print "@arraykeys\n";
print "@arrayvalues\n";    # \n to print new line.
```

To remove all the entries in the hash, we can directly assign the hash to null.

%newHash=();# This will redefine the hash with no entries.

5.12 LISTS

Lists are special type of array which is a temporary construct that holds a series of values.

Eg: @array=(10,20,30);

Merging list:

List is just a comma-separated sequence of values. We can combine list together as @numbers=(1,3(4,5,6));

Selecting arguments from list:

The list notation is identical to that of arrays. We can extract an element from an array by appending (adding) square to the list.

Eg: @array=(10,20,30)[20];

We can extract slices from the list.

Eg: @numbers=(1,2,3,4,5)[/..3];

5.13 TYPEGLOB

Typeglob is a special type of variable which means that you can refer the varaiable to any different type. It starts with asterisk.

Foo contain the value of $foo, %foo @foo,&foo

5.14 STATEMENTS AND CONTROL STRUCTURES

Basic Perl Programming

Loop and control statement

`if (…){}`	Executes when a specified condition is true
`if (…) {} else {}`	Chooses between two alternatives
`if (…){} elsif() {} else {}`	Chooses between more than two alternatives
`for (…) {}`	Repeats a group of statements a specified number of times
`foreach $var_name1 (@var_name2)`	Special loop to access all elements in array variable
`while(…) {}`	Repeats a group of statements a specified number of times

21

5.14.1 STATEMENTS:

Statements are the building blocks of the program. They control the execution of the script.

5.14.2 Code blocks:

A sequence of statements is called a code block or just a "block". The block could be an entire file or a sequence of statements enclosed by a pair curly braces ({ }).

Eg: if ($exp)

{

$a=5+2;

Print "result";

}

Blocks allow to segregate sequence of code for use with loops and control structures.

5.15 Perl Conditional Statements

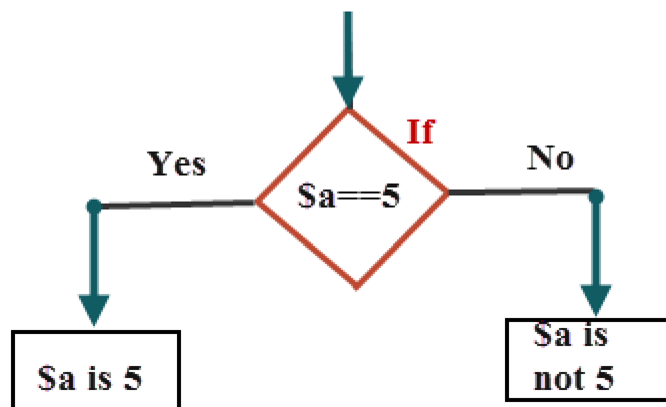
We can use conditional Statements in Perl. So, what are conditional statements? Conditional statements are those, where you actually check for some circumstances to be satisfied in your code.

Think about an example, you are buying some fruits, and you don't like the price to be more than 100 bucks. So, the rule here is 100 bucks.

Perl supports two types of conditional statements; they are if and unless.

5.15.1 Perl If

If code block will be executed, when the condition is true.




```
my $a=5;
if($a==5)
{
print "The value is $a";
}
```

Output:

5

5.15.2 Perl If Else

This looks good. Let us think about a situation where $a is not 5.

```
my $a=10;
if($a==5)
{
```

```
print "The values is $a ---PASS";
}
else
{
print "The value is $a ---FAIL";
}
```

Output:

The value is 10 ---FAIL

This way we can control only one condition at a time. Is it a limitation? No, you can also control various conditions using if… elsif … else.

5.15.3 Perl Else If

```
my $a=5;
if($a==6)
{
print "Executed If block -- The value is $a";
}
elsif($a==5)
{
print "Executed elsif block --The value is $a";
}
else
{
print "Executed else block – The value is $a";
}
```

Output:

Executed elsif block --The value is 5

In the above case, the elsif block will be executed as $a is equal to 5.

There could be situations where both if and elsif code blocks will be failed. In this scenario, the else code block will be executed. You can actually eliminate the else code check if you don't like to include.

5.15.4 Perl Nested If

In this case, you can use if code block in one more if code block.

```
my $a=11; #Change values to 11,2,5 and observe output
```

```
if($a<10){
print "Inside 1st if block";
if($a<5){
print "Inside 2nd if block --- The value is $a";
}
else{
print " Inside 2nd else block --- The value is $a";
}
}
else{
print "Inside 1st else block – The value is $a";
}
```

Output:

Inside 1st else block – The value is 11

Execute the same code by change the value of $a; you can find out the rest.

5.15.5 Perl Unless

You have already got an idea what if does (If the condition is true it will execute the code block). Unless is opposite to if, unless code block will be executed if the condition is false.

```
my $a=5;
unless($a==5)
{
print "Inside the unless block --- The value is $a";
}
else
{
print "Inside else block--- The value is $a";
}
```

Output:

Inside 1st else block – The value is 5

Guess what will be the output. You are right!!!!!. The output will be the print statement of the else block. Because of the condition in unless code block is true, remember unless block will be executed only if the condition is false. Change the value of $a and execute the code you will see the difference.

5.15.6 Perl Using if

```
$a= " This is Perl";
if($a eq "SASSDSS"){
print "Inside If Block";
}
else
{
print "Inside else block"
}
```

Output:

Inside else block

5.15.7 Using unless

```
$a= " This is Perl";
unless($a eq "SASSDSS"){
print "Inside unless Block";
}
else
{
print "Inside else block"
}
```

Output:

Inside unless Block

5.16 Perl Loops - Control Structures

Perl supports control structures similar to other programming languages. Perl supports four types of control structures for, foreach, while and until. We use these statements to, repeatedly execute some code.

5.16.1 For loop Perl

For code block will execute till the condition is satisfied. Let's take an example of how to Perl loop an array.

```
my @array=(1..10);
for(my $count=0;$count<10;$count++)
{
print "The array index $count value is $array[$count]";
print "\n";
```

```
}
```

Output:

The array index 0 value is 1

The array index 1 value is 2

The array index 2 value is 3

The array index 3 value is 4

The array index 4 value is 5

The array index 5 value is 6

The array index 6 value is 7

The array index 7 value is 8

The array index 8 value is 9

The array index 9 value is 10

Here, in for () expression, there are many statements included. There is a meaning for each of them.

for (initialization ; condition; incrementing)

Here is another way of using for.

```
for(1..10)
{
print "$_ n";
print "\n";
}
```

Output:

1n

2n

3n

4n

5n

6n

7n

8n

9n

10n

5.16.2 Perl Foreach

The for each statement can be used in the same way as for; the main difference is we don't have any condition check and incrementing in this.

Let's take the same example with foreach perl.

```
my @array=(1..10);
foreach my $value (@array)
{
print " The value is $value\n";
}
```

Output:

The value is 1

The value is 2

The value is 3

The value is 4

The value is 5

The value is 6

The value is 7

The value is 8

The value is 9

The value is 10

Foreach takes each element of an array and assigns that value to $var for every iteration. We can also use $_ for the same.

```
my @array=(1..10);
foreach(@array)
{
print " The value is $_ \n"; # This is same as the above code.
}
```

Output:

The value is 1

The value is 2

The value is 3

The value is 4

The value is 5

The value is 6

The value is 7

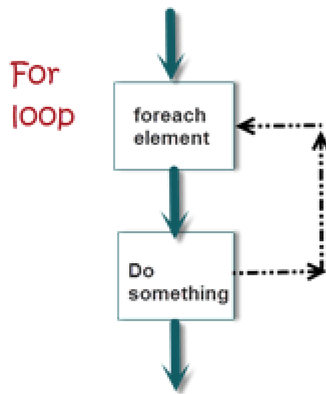
The value is 8

The value is 9

The value is 10

This looks good for accessing arrays. How about Hashes, how can we obtain hash keys and values using foreach?

We can use foreach to access keys and values of the hash by looping it.




```
my %hash=( 'Tom' => 23, 'Jerry' => 24, 'Mickey' => 25);
foreach my $key (keys %hash)
{
print "$key \n";
}
```

Output:

Mickey

Tom

Jerry

You might be wondering, Why we used Keys in foreach(). Keys is an inbuilt function of Perl where we can quickly access the keys of the hash. How about values? We can use values function for accessing values of the hash.

```
my %hash=( 'Tom' => 23, 'Jerry' => 24, 'Mickey' => 25);
foreach my $value(values %hash) # This will push each value of the key to $value
{
print " the value is $value \n";
}
```

Output:

the value is 24

the value is 23

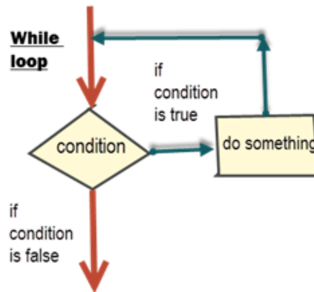
the value is 25

5.16.3 Perl While

The Perl While loop is a control structure, where the code block will be executed till the condition is true.

The code block will exit only if the condition is false.

Let's take an example for Perl While loop.



Here is a problem, which will require input from the user and will not exit until the number provided as '7'.

```
#!/usr/bin/perl
$guru99 = 0;
$luckynum = 7;
print "Guess a Number Between 1 and 10\n";
$guru99 = <STDIN>;
while ($guru99 != $luckynum)
{
print "Guess a Number Between 1 and 10 \n ";
$guru99 = <STDIN>;
}
print "You guessed the lucky number 7"
```

Output:

Guess a Number Between 1 and 10

9

Guess a Number Between 1 and 10

5

Guess a Number Between 1 and 10

7

You guessed the lucky number 7

In the above example, the while condition will not be true if we enter input other than '7'.

If you see how while works here, the code block will execute only if the condition in a while is true.

5.16.4 Perl do-while

Do while loop will execute at least once even if the condition in the while section is false.

Let's take the same example by using do while.

```
$guru99 = 10;
do {
print "$guru99 \n";
$guru99--;
}
while ($guru99 >= 1);
print "Now value is less than 1";
```

Output:

10

9

8

7

6

5

4

3

2

1

Now value is less than 1

5.16.5 Perl until

Until code block is similar to unless in a conditional statement. Here, the code block will execute only if the condition in until block is false.

Let's take the same example which we used in case of a while.

Here is a problem, which will require input from the user and will not exit until the name provided as other than 'sai'.

```
print "Enter any name \n";
my $name=<STDIN>;
chomp($name);
until($name ne 'sai')
{
print "Enter any name \n";
$name=<STDIN>;
chomp($name);
}
```

Output:

Enter any name sai

5.16.6 Perl do-until:

Do until can be used only when we need a condition to be false, and it should be executed at-least once.

```
print "Enter any name \n";
my $name=<STDIN>;
chomp($name);
do
{
print "Enter any name \n";
$name=<STDIN>;
chomp($name);
}until($name ne 'sai');
```

Output:

Enter any name Howard

Enter any name Sheldon

Enter any name sai

Execute while, do-while, until and do-until example codes to see the difference.

5.17 SUBROUTINES,PACKAGES AND MODULES

Subroutines in Perl:

•User-defined functions in Perl are called subroutines. Perl programs are typically written by combining new subroutines the programmer writes with built-in functions available in Perl. build-in function,◊

•We refer "function" user-defined function◊"subroutine"

•Perl provides a rich collection of build-in functions; additionally there are many libraries of Perl code available for download from the Comprehensive Perl Archive Network (CPAN) (www.cpan.org)

•When possible, use Perl build-in functions instead of writing new subroutines. type man perlfunc◊

•Unix/Linux type perldoc perlfunc or online documentation◊

•Other Systems

•A subroutine is invoked by a subroutine call.

The general format of a subroutine definition

Sub subroutine Name

{statements}

Every subroutine definition begins with keyword sub.

5.17.1Argument List

* Subroutines can also receive arguments. The list of arguments passed to a function is stored in the special array variable @_ .

•Special variable @_ is interesting that it flattens all arrays and hashes

•This means that if you send a subroutine an array value and a scalar value, @_ will contain one list with all the data from the array and the scalar value.

•Similarly, a hash is flattened into a list of its key/value pairs (e.g., key1, value1, key2, value2, etc)

•In the body of a subroutine, if you try to assign the contents of @_ to an array and a scalar by using the list assignment (@first, $second) = @_; then @first will take all the data in @_ and $second will remain undefined.

5.17.2 Returning Values

•When a subroutine completes its task, data are returned to the subroutine caller via the return keyword, as shown in the next slide.

•In Perl, the return value from a subroutine can return a scalar, or it can return a list.

Other ways to invoke a subroutine

•Actually there are several ways to call a subroutine

•We call subroutines by using the syntax demonstrated in the last several programs

•Perl uses type identifier characters to distinguish types. Prefixing the type identifier ampersand (&) to a subroutine name, as in &subroutine1; calls subroutine1.

•One other syntax for calling subroutines is the bareword. In this case, there are no special symbols around the subroutine name to help Perl determine the purpose of the name in the program. If the bareword appears before the subroutine definition, Perl interprets the bareword like a string and does not call the subroutine.

BEGIN and END Blocks

You may define any number of code blocks named BEGIN and END which act as constructors and destructors respectively.

```
BEGIN { ... }
END { ... }
BEGIN { ... }
END { ... }
```

- Every **BEGIN** block is executed after the perl script is loaded and compiled but before any other statement is executed
- Every END block is executed just before the perl interpreter exits.
- The BEGIN and END blocks are particularly useful when creating Perl modules.

5.17.3 MODULES

What are Perl Modules?

A Perl module is a reusable package defined in a library file whose name is the same as the name of the package (with a .pm on the end).

A Perl module file called "Foo.pm" might contain statements like this.

```
#!/usr/bin/perl

package Foo;
sub bar {
print "Hello $_[0]\n"
}

sub blat {
print "World $_[0]\n"
}
1;
```

Few noteable points about modules

- The functions **require** and **use** will load a module.
- Both use the list of search paths in **@INC** to find the module (you may modify it!)
- Both call the **eval** function to process the code
- The **1;** at the bottom causes eval to evaluate to TRUE (and thus not fail)

5.17.4 The Require Function

A module can be loaded by calling the **require** function

```
#!/usr/bin/perl

require Foo;

Foo::bar( "a" );
Foo::blat( "b" );
```

Notice above that the subroutine names must be fully qualified (because they are isolated in their own package)

It would be nice to enable the functions bar and blat to be imported into our own namespace so we wouldn't have to use the Foo:: qualifier.

5.17.5 The Use Function

A module can be loaded by calling the **use** function

```
#!/usr/bin/perl

use Foo;

bar( "a" );
blat( "b" );
```

Notice that we didn't have to fully qualify the package's function names?

The use function will export a list of symbols from a module given a few added statements inside a module

```
require Exporter;
@ISA = qw(Exporter);
```

Then, provide a list of symbols (scalars, lists, hashes, subroutines, etc) by filling the list variable named **@EXPORT**: For Example

```
package Module;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(bar blat);

sub bar { print "Hello $_[0]\n" }
sub blat { print "World $_[0]\n" }
sub splat { print "Not $_[0]\n" }  # Not exported!

1;
```

5.17.6 Create the Perl Module Tree

When you are ready to ship your PERL module then there is standard way of creating a Perl Module Tree. This is done using **h2xs** utility. This utility comes alongwith PERL. Here is the syntax to use h2xs

```
$h2xs -AX -n  Module Name

# For example, if your module is available in Person.pm file
$h2xs -AX -n Person

This will produce following result
Writing Person/lib/Person.pm
Writing Person/Makefile.PL
Writing Person/README
Writing Person/t/Person.t
Writing Person/Changes
Writing Person/MANIFEST
```

Here is the descritpion of these options

- **-A** omits the Autoloader code (best used by modules that define a large number of infrequently used subroutines)
- **-X** omits XS elements (eXternal Subroutine, where eXternal means external to Perl, i.e. C)
- **-n** specifies the name of the module

So above command creates the following structure inside Person directory. Actual result is shown above.

5.18 WORKING WITH FILES

The basics of handling files are simple: you associate a **filehandle** with an external entity (usually a file) and then use a variety of operators and functions within Perl to read and update the data stored within the data stream associated with the filehandle.

A filehandle is a named internal Perl structure that associates a physical file with a name. All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle. However, when you associate a filehandle, you can specify the mode in which the filehandle is opened.

Three basic file handles are - **STDIN**, **STDOUT**, and **STDERR,** which represent standard input, standard output and standard error devices respectively.

5.18.1 Opening and Closing Files

There are following two functions with multiple forms, which can be used to open any new or existing file in Perl.

```
open FILEHANDLE, EXPR
open FILEHANDLE

sysopen FILEHANDLE, FILENAME, MODE, PERMS
sysopen FILEHANDLE, FILENAME, MODE
```

Here FILEHANDLE is the file handle returned by the **open** function and EXPR is the expression having file name and mode of opening the file.

5.18.2 Open Function

Following is the syntax to open **file.txt** in read-only mode. Here less than < sign indicates that file has to be opend in read-only mode.

```
open(DATA, "<file.txt");
```

Here DATA is the file handle, which will be used to read the file. Here is the example, which will open a file and will print its content over the screen.

```
#!/usr/bin/perl

open(DATA, "<file.txt") or die "Couldn't open file file.txt, $!";

while(<DATA>) {
print "$_";
}
```

Following is the syntax to open file.txt in writing mode. Here less than > sign indicates that file has to be opend in the writing mode.

```
open(DATA, ">file.txt") or die "Couldn't open file file.txt, $!";
```

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.

For example, to open a file for updating without truncating it −

```
open(DATA, "+<file.txt"); or die "Couldn't open file file.txt, $!";
```

To truncate the file first −

```
open DATA, "+>file.txt" or die "Couldn't open file file.txt, $!";
```

You can open a file in the append mode. In this mode, writing point will be set to the end of the file.

```
open(DATA,">>file.txt") || die "Couldn't open file file.txt, $!";
```

A double >> opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can't read from it unless you also place a plus sign in front of it −

```
open(DATA,"+>>file.txt") || die "Couldn't open file file.txt, $!";
```

Following is the table, which gives the possible values of different modes

Sr.No.	Entities & Definition
1	**< or r** Read Only Access
2	**> or w** Creates, Writes, and Truncates

3	**>> or a** Writes, Appends, and Creates
4	**+< or r+** Reads and Writes
5	**+> or w+** Reads, Writes, Creates, and Truncates
6	**+>> or a+** Reads, Writes, Appends, and Creates

5.18.3 Sysopen Function

The **sysopen** function is similar to the main open function, except that it uses the system **open()** function, using the parameters supplied to it as the parameters for the system function −

For example, to open a file for updating, emulating the **+<filename** format from open −

sysopen(DATA, "file.txt", O_RDWR);

Or to truncate the file before updating −

sysopen(DATA, "file.txt", O_RDWR|O_TRUNC);

You can use O_CREAT to create a new file and O_WRONLY- to open file in write only mode and O_RDONLY - to open file in read only mode.

The **PERMS** argument specifies the file permissions for the file specified, if it has to be created. By default it takes **0x666**.

Following is the table, which gives the possible values of MODE.

Sr.No.	Entities & Definition
1	**O_RDWR** Read and Write

2	**O_RDONLY** Read Only
3	**O_WRONLY** Write Only
4	**O_CREAT** Create the file
5	**O_APPEND** Append the file
6	**O_TRUNC** Truncate the file
7	**O_EXCL** Stops if file already exists
8	**O_NONBLOCK** Non-Blocking usability

5.18.4 Close Function

To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the **close** function. This flushes the filehandle's buffers and closes the system's file descriptor.

```
close FILEHANDLE
close
```

If no FILEHANDLE is specified, then it closes the currently selected filehandle. It returns true only if it could successfully flush the buffers and close the file.

```
close(DATA) || die "Couldn't close file properly";
```

5.18.5 Reading and Writing Files

Once you have an open filehandle, you need to be able to read and write information. There are a number of different ways of reading and writing data into the file.

5.18.6 The <FILEHANDL> Operator

The main method of reading the information from an open filehandle is the <FILEHANDLE> operator. In a scalar context, it returns a single line from the filehandle. For example −

```
#!/usr/bin/perl

print "What is your name?\n";
$name = <STDIN>;
print "Hello $name\n";
```

When you use the <FILEHANDLE> operator in a list context, it returns a list of lines from the specified filehandle. For example, to import all the lines from a file into an array −

```
#!/usr/bin/perl

open(DATA,"<import.txt") or die "Can't open data";
@lines = <DATA>;
close(DATA);
```

5.18.7 read Function

The read function reads a block of information from the buffered filehandle: This function is used to read binary data from the file.

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALAR if no OFFSET is specified. Otherwise data is placed after OFFSET bytes in SCALAR. The function returns the number of bytes read on success, zero at end of file, or undef if there was an error.

5.18.8 print Function

For all the different methods used for reading information from filehandles, the main function for writing information back is the print function.

```
print FILEHANDLE LIST
print LIST
print
```

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current output filehandle (STDOUT by default). For example −

```
print "Hello World!\n";
```

5.18.9 Copying Files

Here is the example, which opens an existing file file1.txt and read it line by line and generate another copy file file2.txt.

```
#!/usr/bin/perl

# Open file to read
open(DATA1, "<file1.txt");

# Open new file to write
open(DATA2, ">file2.txt");

# Copy data from one file to another.
while(<DATA1>) {
print DATA2 $_;
}
close( DATA1 );
close( DATA2 );
```

5.18.10 Renaming a file

Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is available in /usr/test directory.

```
#!/usr/bin/perl
rename ("/usr/test/file1.txt", "/usr/test/file2.txt" );
```

This function **renames** takes two arguments and it just renames the existing file.