DIGITAL NOTES
ON
R17A01251 – INTRODUCTION TO SCRIPTING LANGUAGES

B.TECH III YEAR - I SEM
(2019-20)

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

	L	T/P/D	C
III Year B.Tech. IT - I Sem	**3**	**-/-/-**	**3**

(R17A01251) INTRODUCTION TO SCRIPTING LANGUAGES

UNIT I

Introduction to PERL and Scripting Scripts and Programs, Origin of Scripting, Scripting Today, Characteristics of Scripting Languages, Uses for Scripting Languages, Web Scripting, and the universe of Scripting Languages. PERL- Names and Values, Variables, Scalar Expressions, Control Structures, arrays, list, hashes, strings, pattern and regular expressions, subroutines.

UNIT II

HTML: HTML basics, Elements, Attributes and Tags, Basic Tags, Advanced Tags, Frames, Images.

Cascading style sheets: Adding CSS, CSS and page layout.

JavaScript: Introduction, Variables, Literals, Operators, Control structure, Conditional statements, Arrays, Functions, Objects, Predefined objects, Object hierarchy, Accessing objects.

UNIT III

JavaScript programming of reactive web pages elements: Events, Event handlers, multiple windows and Frames, Form object and Element, Advanced JavaScript and HTML, Data entry and Validation, Tables and Forms.

Introduction to Python Programming: History of Python, Need of Python Programming, Running Python Scripts, Variables, Assignment, Keywords, Input-Output, Indentation, Types - Integers, Strings, Booleans.

UNIT IV

Operators and Expressions: Operators- Arithmetic Operators, Comparison (Relational) Operators, Assignment Operators, Logical Operators, Bitwise Operators, Membership Operators, Identity Operators, Expressions and order of evaluations.

Data Structures: Lists - Operations, Slicing, Methods; Tuples, Sets, Dictionaries, Sequences.

UNIT V

Control Flow - if, if-else, for, while, break, continue, pass

Functions - Defining Functions, Calling Functions, Passing Arguments, Default Arguments, Variable-length arguments, Fruitful Functions(Function Returning Values), Scope of the Variables in a Function - Global and Local Variables. Development of sample scripts and web applications.

TEXT BOOKS:

1. The World of Scripting Languages, David Barron, Wiley Publications.
2. Learning Python, Mark Lutz, Orielly
3. Web Programming, building internet applications, Chris Bates 2nd Edition, WILEY
4. Beginning JavaScript with Dom scripting and AJAX, Russ Ferguson, Christian Heilmann, Apress.
5. Python Web Programming, Steve Holden and David Beazley, New Riders Publications.

INDEX

UNIT-1

Introduction to perl and scripts and programs

Scripting is the action of writing scripts using a scripting language, distinguishing neatly between programs, which are written in conventional programming language such as C,C++,java, and scripts, which are written using a different kind of language.

We could reasonably argue that the use of scripting languages is just another kind of programming. Scripting languages are used for is qualitatively different from conventional programming languages like C++ and Ada address the problem of developing large applications from the ground up, employing a team of professional programmers, starting from well-defined specifications, and meeting specified performance constraints. Scripting languages, on other hand, address different problems: Building applications from 'off the shelf' components Controlling applications that have a programmable interface Writing programs where speed of development is more important than run-time efficiency.

The most important difference is that scripting languages incorporate features that enhance the productivity of the user in one way or another, making them accessible to people who would not normally describe themselves as programmers, their primary employment being in some other capacity. Scripting languages make programmers of us all, to some extent.

Origin of scripting:

The use of the word 'script' in a computing context dates back to the early 1970s,when the originators of the UNIX operating system create the term 'shell script' for sequence of commands that were to be read from a file and follow in sequence as if they had been typed in at the keyword. e.g. an 'AWKscript', a 'perl script' etc.. the name 'script ' being used for a text file that was intended to be executed directly rather than being compiled to a different form of file prior to execution. Other early occurrences of the term 'script' can be found. For example, in a DOS-based system, use of a dial-up connection to a remote system required a communication package that used proprietary language to write scripts to automate the sequence of operations required to establish a connection to a remote system. Note that if we regard a scripts as a sequence of commands to control an application or a device, a configuration file such as a UNIX

‘make file’ could be regard as a script. However, scripts only become interesting when they have the added value that comes from using programming concepts such as loops and branches.

Scripting today:

SCRIPTING IS USED WITH 3 DIFFRENT MEANINGS: 1. A new style of programming which allows applications to be developed much faster than traditional methods allow,and maks it possible for applications to evolve rapidly to meet changing user requirements.This style of programming frequently uses a scripting language to interconnect ‘off the shelf ‘ components that are themselves written in conventional language.Applications built in this way are called ‘glue applications’ ,and the language is called a ‘glue language’. A glue language is a programming language (usually an interpreted scripting language) that is designed or suited for writing glue code – code to connect software components. They are especially useful for writing and maintaining: Custom commands for a command shell Smaller programs than those that are better implemented in a compiled language

A glue language is a programming language (usually an interpreted scripting language) that is designed or suited for writing glue code – code to connect software components. They are especially useful for writing and maintaining:

- Custom commands for a command shell
- Smaller programs than those that are better implemented in a compiled language
- "Wrapper" programs for executables, like a batch file that moves or manipulates files and does other things with the operating system before or after running an application like a word processor, spreadsheet, data base, assembler, compiler, etc.
- Scripts that may change Rapid prototypes of a solution eventually implemented in another, usually compiled, language.

Characteristics of scripting languages:

These are some properties of scripting languages which differentiate SL from programming languages.

Integrated compile and run:SL’s are usually characterized as interpreted languages,but this is just an oversimplification.They operate on an immediate execution,without need to issue separate commond to compile the program and then to run the resulting object file,and without

the need to link extensive libraries into he object code.This is done automatically.A few SL'S are indeed implemented as strict interpreters.

Efficiency is not an issue:ease of use is achieved at the expense of effeciency,because efficiency is not an issue in the applications for which SL'S are designed.

A scripting language is usually interpreted from source code or bytecode. By contrast, the software environment the scripts are written for is typically written in a compiled language and distributed in machine code form.

Scripting languages may be designed for use by end users of a program – end-user development – or may be only for internal use by developers, so they can write portions of the program in the scripting language.

Scripting languages typically use abstraction, a form of information hiding, to spare users the details of internal variable types, data storage, and memory management.

Scripts are often created or modified by the person executing them, but they are also often distributed, such as when large portions of games are written in a scripting language.

The characteristics of ease of use,particularly the lack of an explicit compile-link-load sequence,are sometimes taken as the sole definition of a scripting language.

Users For Scripting Languages

Users are classified into two types 1. Modern applications 2. Traditional users Modern applications of scripting languages are: 1.Visual scripting: A collection of visual objects is used to construct a graphical interface.This process of constructing a graphical interface is known as visual scripting.the properties of visual objects include text on button,background and foreground colors.These properties of objects can be changed by writing program in a suitable language. The outstanding visual scripting system is visual basic.It is used to develop new applications.Visual scripting is also used to create enhanced web pages. 2.Scripting components:In scripting languages we use the idea to control the scriptable objects belonging to scripting architecture.Microsoft's visual basic and excel are the first applications that used the concept of scriptable objects.To support all the applications of microsoft the concept of scriptable objects was developed.3.Web scripting:web scripting is classified into three

forms.they are processing forms,dynamic web pages,dynamically generating HTML. Applications of traditional scripting languages are:

1. system administration,
2. experimental programming,

web scripting

3. controlling applications.

Application areas :

Four main usage areas for scripting languages:

1. Command scripting languages
2.Application scripting languages
3.Markup language
4. Universal scripting languages

web scripting:

Web is the most fertile areas for the application of scripting languages. Web scripting divides into three areas a. processing forms b. creating pages with enhanced visual effects and user interaction and c. generating pages 'on the fly' from material held in database.

Processing Web forms: In the original implementation of the web , when the form is submitted for processing, the information entered by the user is encoded and sent to the server for processing by a CGI script that generates an HTML page to be sent back to the Web browser. This processing requires string manipulation to construct the HTML page that constitutes the replay, and may also require system access , to run other processes and to establish network connections. Perl is also a language that uses CGI scripting. Alternatively for processing the form with script running on the server it possible to do some client –side processing within the browser to validate form data before sending it to the server by using JavaScript, VBScript etc. **Dynamic Web pages:** 'Dynamic HTML' makes every component of a Web page (headings, anchors, tables etc.) a scriptable object. This makes it possible to provide simple interaction with the user using scripts written in JavaScript/Jscript or VBScript, which are interpreted by the

browser. Microsoft's ActiveX technology allows the creation of pages with more elaborate user interaction by using embedded visual objects called ActiveX controls. These controls are scriptable objects, and can in fact be scripted in a variety languages. This can be scripted by using Perl scripting engine.

The universe of scripting languages:

Scripting can be traditional or modern scripting, and Web scripting forms an important part of modern scripting. Scripting universe contains multiple overlapping worlds: the original UNIX world of traditional scripting using Perl and Tcl the Microsoft world of Visual Basic and Active controls the world of VBA for scripting compound documents the world of client-side and server-side Web scripting. The overlap is complex, for example web scripting can be done in VBScript, JavaScript/Jscript, Perl or Tcl. This universe has been enlarged as Perl and Tcl are used to implement complex applications for large organizations e.g Tcl has been used to develop a major banking system, and Perl has been used to implement an enterprisewide document management system for a leading aerospace company.

Names and Values in Perl:

Names:

Like any other programming language,Perl manipulates variables which have a name (or identifier) and a value: a value is assigned to a variable by an assignment statement of the form name=value; Variable names resemble nouns in English, and like English, Perl distinguishes between singular and plural nouns.A singular name is associated with a variable that holds a single item of data (a scalar value), a plural name is associated with a variable that holds a collection of data items (an array or hash). A notable characteristic of Perl is that variable

Variables and assignment

Assignment: Borrowing from C, Perl uses '=' as the assignment operator. It is important to note that an assignment statement returns a value, the value assigned. This permits statements like

```
$b = 4 + ( $a = 3) ;
```

which assigns the value 3 to

$a and the value 7 to $b.

If it is required to interpolate a variable value without an intervening space the following syntax, borrowed from UNIX shell scripts , is used:

$a = "Java ;

Scalar Expressions:

Scalar data items are combined into expressions using operators. Perl has a lot of operators, which are ranked in 22 precedence levels. These are carefully chosen so that the 'obvious' meaning is what You get , but the old advice still applies: if in doubt ,use brackets to force the order of evaluation . In the following sections we describe the available operators in their natural groupings-arithmetic , strings,logical etc

Arithmetic operators: Following the principles of 'no surprises' Perl provides the usual Arithmetic operators, including auto-increment and auto-decrement operators after the manner of C: note that in

$c= 17 ; $d= ++$c;

The sequence is increment and the assign, whereas in

$c= 17 ; $d = $c++;

The sequence is assign then increment . As C, binary arithmetic operations can be combined with assignment,

e.g. $a += 3; This adds 3

to $a, being equivalent to

$a =$a + 3;

As in most other languages, unary minus is used to negate a numeric value; an almost neverused unary plus operator is provided for completeness. String Operators Perl provides very basic operators on strings: most string processing is one using built-in functions expressions, as described later. Unlike many languages use + as a concatenation operator for strings, Perl uses a period for this purpose: this lack of overloading means that an operator uniquely determines the context for its operands. The other string operator is x, which is used to replicate strings, e.g. $a =”Hello” x 3; Sets $a to “HelloHelloHello”. The capability of combining an operator with assignment is extended to string operations. E.g. $foo .= “ “ ; Appends a space to $foo. So far, things have been boringly conventional for the most part. However, we begin to get a taste of the

real flavor of perl when we see how it adds a little magic when some operators, normally used in arithmetic context, are used in a string context.

Control structures:

The Control Structures for conditional execution and repetition all the control mechanisms is similar to C. 1. BLOCKS: A block is a sequence i\of one or more statements enclosed in curly braces. Eg: { $positive =1;

$negative=-1;} The last statement is the block terminated by the closing brace. In, Perl they use conditions to control the evaluation of one or more blocks. Blocks can appear almost anywhere that a statement can appear such a block called bare block.

LIST,ARRAYS AND HASHES:

LISTS: A list is a collection of scalar data items which can be treated as a whole, and has a temporary existence on the run-time stack. It is a collection of variables, constants (numbers or strings) or expressions, which is to be treated as a whole. It is written as a comma-separated sequence of values, eg: “red” , “green” , “blue”. A list often appears in a script enclosed in round brackets. For eg: (“red” , “green”, “blue”) Shorthand notation is acceptable in lists, for eg: (1..8) (“A”..”H” , “O”..”Z”) qw(the quick brown fox) is a shorthand for (“the” , ”quick” , ”brown” , ”fox”)

Arrays and Hashes: These are the collections of scalar data items which have an assigned storage space in memory, and can therefore be accessed using a variable name. Arrays: An array is an ordered collection of data whose comparisions are identified by an ordinal index: It is usually the value of an array variable. The name of the variable always starts with an @, eg: @days_of_week. NOTE: An array stores a collection, and List is a collection, So it is natural to assign a list to an array. Eg: @rainfall = (1.2 , 0.4 , 0.3 , 0.1 , 0 , 0 , 0); A list can occur as an element of another list. Eg: @foo = (1 , 2 , 3, “string”); @foobar = (4 , 5 , @foo , 6); The foobar result would be (4 , 5 , 1 , 2 , 3 , “string” , 6);

Hashes: An associative array is one in which each element has two components : a key and a value, the element being ‘indexed’ by its key. Such arrays are usually stored in a hash table to facilitate efficient retrieval, and for this reason Perl uses the term hash for an associative array.

Names of hashes in Perl start with a % character: such a name establishes a list context. The index is a string enclosed in braces(curly brackets). Eg: $somehash{aaa} = 123; $somehash,"$a"- = 0; //The key is a the current value of $a. %anotherhash =%somehash;

UNIT-2

HTML

HTML

HTML stands for Hypertext Markup Language, and it is the most widely used language to write Web Pages.

Hypertext refers to the way in which Web pages (HTML documents) are linked together. Thus, the link available on a webpage is called Hypertext.

As its name suggests, HTML is a Markup Language which means you use HTML to simply "mark-up" a text document with tags that tell a Web browser how to structure it to display.

Originally, HTML was developed with the intent of defining the structure of documents like headings, paragraphs, lists, and so forth to facilitate the sharing of scientific information between researchers.

Now, HTML is being widely used to format web pages with the help of different tags available in HTML language.

Basic HTML Document

```
<!DOCTYPE html>
<html>
<head>
<title>This is document title</title>
</head>
<body>
<h1>This is a heading</h1>
<p>Document content goes here.....</p>
</body>
</html>
```

Elements:

An HTML element is defined by a starting tag. If the element contains other content, it ends with a closing tag, where the element name is preceded by a forward slash as shown below with few tags:

Start Tag	Content	End Tag
	This is paragraph content.	
<h1>	This is heading content.	</h1>
	This is division content.	
 		</br>

So here ....is an HTML element. ...is another HTML element. There are some HTML elements which don't need to be closed, such as and elements. These are known as void elements. HTML documents consists of a tree of these elements and they specify how HTML documents should be built, and what kind of content should be placed in what part of an HTML document.

Nested HTML Elements

It is very much allowed to keep one HTML element inside another HTML element:

```
<title>Nested Elements Example</title>
</head>
<body>
<h1>This is <i>italic</i> heading</h1>
<p>This is <u>underlined</u> paragraph</p>
</body>
</html>
```

This will display the following result:

This is italic heading This is underlined paragraph

HTML – ATTRIBUTES

We have seen few HTML tags and their usage like heading tags, , paragraph tag and other tags. We used them so far in their simplest form, but most of the HTML tags can also have attributes, which are extra bits of information.

An attribute is used to define the characteristics of an HTML element and is placed inside the element's opening tag. All attributes are made up of two parts: a name and a value:

The name is the property you want to set. For example, the paragraph element in the example carries an attribute whose name is align, which you can use to indicate the alignment of paragraph on the page

The value is what you want the value of the property to be set and always put within quotations. The below example shows three possible values of align attribute: left, center and right.

Attribute names and attribute values are case-insensitive. However, the World Wide Web Consortium (W3C) recommends lowercase attributes/attribute values in their HTML 4 recommendation.

```
<!DOCTYPE html>
<html>
<head>
<title>Align Attribute Example</title>
</head>
<body>
<p align="left">This is left aligned</p>
<p align="center">This is center aligned</p>
<p align="right">This is right aligned</p>
</body>
</html>
```

This will display the following result:

This is left aligned

This is center aligned

HTML – BASIC TAGS

Heading Tags

Any document starts with a heading. You can use different sizes for your headings. HTML also has six levels of headings, which use the elements and. While displaying any heading, browser adds one line before and one line after that heading.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Heading Example</title>
</head>
<body>
<h1>This is heading 1</h1>
<h2>This is heading 2</h2>
<h3>This is heading 3</h3>
<h4>This is heading 4</h4>
<h5>This is heading 5</h5>
<h6>This is heading 6</h6>
</body>
</html>
```

Paragraph Tag

The tag offers a way to structure your text into different paragraphs. Each paragraph of text should go in between an opening and a closing tag as shown below in the example:

```
<!DOCTYPE html>
<html>
<head>
<title>Paragraph Example</title>
</head>
<body>
<p>Here is a first paragraph of text.</p>
<p>Here is a second paragraph of text.</p>
<p>Here is a third paragraph of text.</p>
```

</body>

</html>

result

Here is a first paragraph of text.

Here is a second paragraph of text.

Here is a third paragraph of text.

FRAMES

FRAMES

<frame> Tags In HTML

How to Create Frames

While frames should not be used for new websites, learning how to use frames can be beneficial for webmasters who are managing older websites.

The Basic Idea Behind Frames

The basic concept behind frames is pretty simple:

- Use the frameset element in place of the body element in an HTML document.
- Use the frame element to create frames for the content of the web page.
- Use the src attribute to identify the resource that should be loaded inside each frame.

Create a different file with the contents for each frame.

<!DOCTYPE html>

<html>

<body>

<h1>Frame 1</h1>

Contents of Frame 1

</body>

```
</html>
```

CREATING VERTICAL COLUMNS

```
<!DOCTYPE html>
 <html>
 <frameset cols="*,*,*,*">
 <frame src="../file_path/frame_1.html">
<frame src="frame_2.html">
<frame src="frame_3.html">
 <frame src="frame_4.html">
</frameset> </html>
```

Creating Horizontal Rows

```
<!DOCTYPE html>
 <html>
<frameset rows="*,*,*,*">
<frame src="frame_1.html">
 <frame src="frame_2.html">
 <frame src="frame_3.html">
<frame src="frame_4.html">
</frameset>
</html>
```

CSS Introduction (cascading style sheet)

What is CSS?

CSS stands for Cascading Style Sheets

CSS describes **how HTML elements are to be displayed on screen, paper, or in other media**CSS **saves a lot of work**. It can control the layout of multiple web pages all at once

External stylesheets are stored in CSS files

Three Ways to Insert CSS

There are three ways of inserting a style sheet:

- External style sheet
- Internal style sheet

- Inline style

External Style Sheet

With an external style sheet, you can change the look of an entire website by changing just one file! Each page must include a reference to the external style sheet file inside the <link> element. The <link> element goes inside the <head> section:

```
<head>
<link rel="stylesheet" type="text/css" href="mystyle.css">
</head>
```

An external style sheet can be written in any text editor. The file should not contain any html tags. The style sheet file must be saved with a .css extension.

Here is how the "mystyle.css" looks:

```
body {
  background-color: lightblue;
}

h1 {
  color: navy;
  margin-left: 20px;
}
```

Internal Style Sheet

An internal style sheet may be used if one single page has a unique style.

Internal styles are defined within the <style> element, inside the <head> section of an HTML page:

```
<head>
<style>
body {
  background-color: linen;
}
```

```
h1 {
  color: maroon;
  margin-left: 40px;
}
</style>
</head>
```

Inline Styles

An inline style may be used to apply a unique style for a single element.To use inline styles, add the style attribute to the relevant element. The style attribute can contain any CSS property.The example below shows how to change the color and the left margin of a <h1> element:

```
<h1 style="color:blue;margin-left:30px;">This is a heading</h1>
```

CSS Website Layout

Header

A header is usually located at the top of the website (or right below a top navigation menu). It often contains a logo or the website name:

```
header {
  background-color: #F1F1F1;
  text-align: center;
  padding: 20px;
}
```

Navigation Bar

```
/* The navbar container */
.topnav {
  overflow: hidden;
  background-color: #333;
}
```

```
/* Navbar links */
.topnav a {
  float: left;
  display: block;
  color: #f2f2f2;
  text-align: center;
  padding: 14px 16px;
  text-decoration: none;
}

/* Links - change color on hover */
.topnav a:hover {
  background-color: #ddd;
  color: black;
}
```

Content

The layout in this section, often depends on the target users. The most common layout is one (or combining them) of the following:

1-column (often used for mobile browsers)

2-column (often used for tablets and laptops)

3-column layout (only used for desktops)

```
/* Create three equal columns that floats next to each other */
.column {
  float: left;
  width: 33.33%;
}

/* Clear floats after the columns */
.row:after {
```

```
  content: "";
  display: table;
  clear: both;
}

/* Responsive layout - makes the three columns stack on top of each other instead of next to each other on smaller screens (600px wide or less) */
@media screen and (max-width: 600px) {
  .column {
    width: 100%;
  }
}
```

JAVASCRIPT

JavaScript was designed to 'plug a gap' in the techniques available for creating web-pages.HTML is relatively easy to learn, but it is static. It allows the use of links to load new pages, images, sounds, etc., but it provides very little support for any other type of interactivity.

To create dynamic material it was necessary to use either:

- CGI (Common Gateway Interface) programs
  - Can be used to provide a wide range of interactive features, but...
  - Run on the server, i.e.:
  - A user-action causes a request to be sent over the internet from the client machine to the server.
  - The server runs a CGI program that generates a new page, based on the information supplied by the client.

- The new page is sent back to the client machine and is loaded in place of the previous page.
- Thus every change requires communication back and forth across the internet.
- Written in languages such as Perl, which are relatively difficult to learn.

Variables & Literals

A variable is a container which has a name. We use variables to hold information that may change from one moment to the next while a program is running.

For example, a shopping website might use a variable called total to hold the total cost of the goods the customer has selected. The amount stored in this variable may change as the customer adds more goods or discards earlier choices, but the name total stays the same. Therefore we can find out the current total cost at any time by asking the program to tell us what is currently stored in total.

A literal, by contrast, doesn't have a name - it only has a value.

For example, we might use a literal to store the VAT rate, since this doesn't change very often. The literal would have a value of (e.g.) 0.21. We could then obtain the final cost to the customer in the following way:

VAT is equal to total x 0.21

final total is equal to total + VAT

JavaScript accepts the following types of variables:

Numeric	Any numeric value, whether a whole number (an integer) or a number that includes a fractional part (a real), e.g., 12 3.14159 etc.
String	A group of text characters, e.g., Ian

Macintosh G4
etc.

Boolean A value which can only be either True or False, e.g.

completed
married
etc.

Operators

Operators are a type of command. They perform operations on variables and/or literals and produce a result.

JavaScript understands the following operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

These are known as binary operators because they require two values as input, i.e.:

4 + 3
7 / 2
15 % 4

In addition, JavaScript understands the following operators:

+ +	Increment	Increase value by 1

- -	Decrement	Decrease value by 1
-	Negation	Convert positive to negative, or vice versa

These are known as *unary* operators because they require only *one* value as input, i.e.:

4++	increase 4 by 1 so it becomes 5
7--	decrease 7 by 1 so it becomes 6
-5	negate 5 so it becomes -5

JavaScript operators are used in the following way:

```
var totalStudents = 60
var examPasses = 56
var resits = totalStudents – examPasses
```

JavaScript Statements

JavaScript statements are the commands to tell the browser to what action to perform. Statements are separated by semicolon (;).

JavaScript statement constitutes the JavaScript code which is translated by the browser line by line.

Example:

```
document.getElementById("demo").innerHTML = "Welcome";
```

Sr.No.	Statement	Description
1.	switch case	A block of statements in which execution of code depends upon different cases. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a **default** condition will be used.
2.	If else	The **if** statement is the fundamental control statement

		that allows JavaScript to make decisions and execute statements conditionally.
3.	While	The purpose of a while loop is to execute a statement or code block repeatedly as long as expression is true. Once expression becomes false, the loop will be exited.
4.	do while	Block of statements that are executed at least once and continues to be executed while condition is true.
5.	for	Same as while but initialization, condition and increment/decrement is done in the same line.
6.	for in	This loop is used to loop through an object's properties.
7.	continue	The continue statement tells the interpreter to immediately start the next iteration of the loop and skip remaining code block.
8.	break	The break statement is used to exit a loop early, breaking out of the enclosing curly braces.
9.	function	A function is a group of reusable code which can be called anywhere in your programme. The keyword function is used to declare a function.
10.	return	Return statement is used to return a value from a function.
11.	var	Used to declare a variable.
12.	try	A block of statements on which error handling is

		implemented.
13.	catch	A block of statements that are executed when an error occur.
14.	throw	Used to throw an error.

Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

String[] cars;

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

// Outputs Volvo

Change an Array Element

To change the value of a specific element, refer to the index number:

cars[0] = "Opel";

Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

cars[0] = "Opel";

System.out.println(cars[0]);

// Now outputs Opel instead of Volvo

Array Length

To find out how many elements an array has, use the length property:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4
```

Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.The following example outputs all elements in the cars array:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

Loop Through an Array with For-Each

There is also a "for-each" loop, which is used exclusively to loop through elements in arrays:

Syntax:

```
for (type variable : arrayname) {
  ...
}
```

Example:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

Multidimensional Arrays

A multidimensional array is an array containing one or more arrays. To create a two-dimensional array, add each array within its own set of curly braces:

Example

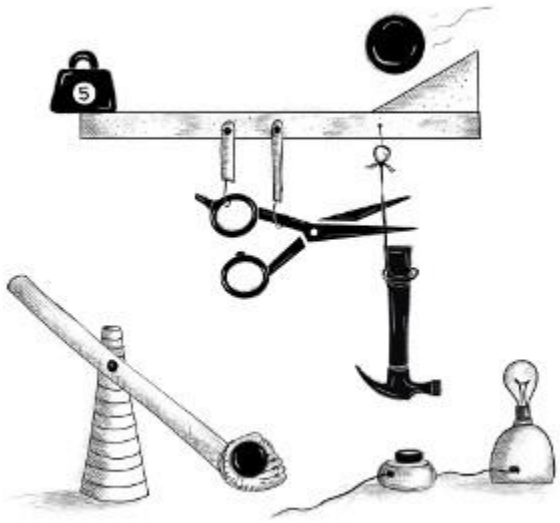
```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

```
public class MyClass {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
    }
  }
}
```

UNIT-3

JAVASCRIPT PROGRAMMING OF REACTIVE WEB PAGES ELEMENTS: HANDLING EVENTS

You have power over your mind—not outside events. Realize this, and you will find strength. Marcus Aurelius, Meditations



Some programs work with direct user input, such as mouse and keyboard actions. That kind of input isn't available as a well-organized data structure—it comes in piece by piece, in real time, and the program is expected to respond to it as it happens.

Event handlers

Imagine an interface where the only way to find out whether a key on the keyboard is being pressed is to read the current state of that key. To be able to react to keypresses, you would have to constantly read the key's state so that you'd catch it before it's released again. It would be dangerous to perform other time-intensive computations since you might miss a keypress.

Some primitive machines do handle input like that. A step up from this would be for the hardware or operating system to notice the keypress and put it in a queue. A program can then periodically check the queue for new events and react to what it finds there.

Of course, it has to remember to look at the queue, and to do it often, because any time between the key being pressed and the program noticing the event will cause the software to feel unresponsive. This approach is called polling. Most programmers prefer to avoid it.

A better mechanism is for the system to actively notify our code when an event occurs. Browsers do this by allowing us to register functions as handlers for specific events.

```
edit & run code by clicking it
edit & run code by clicking it
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

The window binding refers to a built-in object provided by the browser. It represents the browser window that contains the document. Calling its addEventListener method registers the second argument to be called whenever the event described by its first argument occurs.

Key events

When a key on the keyboard is pressed, your browser fires a "keydown" event. When it is released, you get a "keyup" event.

```
<p>This page turns violet when you hold the V key.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
    }
  });
</script>
```

Pointer events

There are currently two widely used ways to point at things on a screen: mice (including devices that act like mice, such as touchpads and trackballs) and touchscreens. These produce different kinds of events.

Mouse clicks

Pressing a mouse button causes a number of events to fire.
The "mousedown"and "mouseup" events are similar to "keydown" and "keyup" and fire when the button is pressed and released. These happen on the DOM nodes that are immediately below the mouse pointer when the event occurs.

After the "mouseup" event, a "click" event fires on the most specific node that contained both the press and the release of the button. For example, if I press down the mouse button on one paragraph and then move the pointer to another paragraph and release the button, the "click" event will happen on the element that contains both those paragraphs.

If two clicks happen close together, a "dblclick" (double-click) event also fires, after the second click event.

To get precise information about the place where a mouse event happened, you can look at its clientX and clientY properties, which contain the event's coordinates (in pixels) relative to the top-left corner of the window, or pageXand pageY, which are relative to the top-left corner of the whole document (which may be different when the window has been scrolled).

The following implements a primitive drawing program. Every time you click the document, it adds a dot under your mouse pointer. See Chapter 19 for a less primitive drawing program.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: blue;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

Using JavaScript to Work with Windows and Frames

JavaScript includes powerful features for working with browser windows. This lesson explains how to use JavaScript to open new windows, move and resize existing windows, display dialog boxes and prompts, and work with frames. It also explains how you can use timeouts to create self-updating pages

You should now have a basic understanding of the objects in the level 0 DOM, and the events that can be used with each object.

In this hour, you'll learn more about some of the most useful objects in the level 0 DOM—browser windows and frames—and how JavaScript can work with them. Hour 11 covers the following topics:

- The window object hierarchy
- Creating new windows with JavaScript
- Delaying your script's actions with timeouts
- Displaying alerts, confirmations, and prompts
- Using JavaScript to work with frames
- Creating a JavaScript-based navigation frame

Controlling Windows with Objects

In Hour 9, "Working with the Document Object Model," you learned that you can use DOM objects to represent various parts of the browser window and the current HTML document. You also learned that the history, document, and location objects are all children of the window object.

In this hour, you'll take a closer look at the window object itself. As you've probably guessed by now, this means you'll be dealing with browser windows. A variation of the window object also allows you to work with frames, as you'll see later in this hour.

The window object always refers to the current window (the one containing the script). The selfkeyword is also a synonym for the current window. As you'll learn in the next section,

you can have more than one window on the screen at the same time, and can refer to them with different names.

Creating a New Window

One of the most convenient uses for the window object is to create a new window. You can do this to display a document—for example, the instructions for a game—without clearing the current window. You can also create windows for specific purposes, such as navigation windows.

You can create a new browser window with the window.open() method. A typical statement to open a new window looks like this:

WinObj=window.open("URL", "WindowName", "Feature List");

The following are the components of the window.open() statement:

- The WinObj variable is used to store the new window object. You can access methods and properties of the new object by using this name.

- The first parameter of the window.open() method is a URL, which will be loaded into the new window. If it's left blank, no Web page will be loaded.

- The second parameter specifies a window name (here, WindowName). This is assigned to the window object's name property and is used to refer to the window.

- The third parameter is a list of optional features, separated by commas. You can customize the new window by choosing whether to include the toolbar, status line, and other features. This enables you to create a variety of "floating" windows, which may look nothing like a typical browser window.

The features available in the third parameter of the window.open() method include width and height, to set the size of the window, and several features that can be set to either yes (1) or no (0): toolbar, location, directories, status, menubar, scrollbars, and resizable. You can list only the features you want to change from the default. This example creates a small window with no toolbar or status line:

SmallWin = window.open("","small","width=100,height=120,toolbar=0,status=0");

Opening and Closing Windows

Of course, you can close windows as well. The window.close() method closes a window. Netscape doesn't allow you to close the main browser window without the user's permission; its main purpose is for closing windows you have created. For example, this statement closes a window called updatewindow:

updatewindow.close();

Listing 11.1 An HTML document that uses JavaScript to enable you to create and close windows

```
<html>
<head><title>Create a New Window</title>
</head>
<body>
<h1>Create a New Window</h1>
<hr>
<p>Use the buttons below to test opening and closing windows in JavaScript.</p>
<hr>
<form NAME="winform">
<input TYPE="button" VALUE="Open New Window"
onClick="NewWin=window.open('','NewWin',
'toolbar=no,status=no,width=200,height=100'); ">
<p><input TYPE="button" VALUE="Close New Window"
onClick="NewWin.close();" ></p>
<p><input TYPE="button" VALUE="Close Main Window"
onClick="window.close();"></p>
</form>
<br><p>Have fun!</p>
<hr>
</body>
</html>
```

List of JavaScript Events on HTML Elements

Most of the names describe exactly when they fire. A few of them are only applicable to a few kinds of elements. See JavaScript Elements for code samples and demonstrations.

On click

When the user clicks on the element. (Mouse button is pressed and released immediately.)

On context menu

When the context menu ("right-click") is opened (or when the user tries to open it — this can be used to disable the context menu).

On dblclick

When the element is double-clicked.

On mouse down

When the user presses the mouse button down while inside (or over) the element. (Mouse button is pressed and held.)

Onmouseenter

When the mouse enters the element. (Sometimes called "hover," but that is not the name of the event.)

On mouse leave

When the mouse leaves an element.

On mouse move

When the mouse moves while inside an element.

On mouse over

When the mouse enters an element, or one of its children.

On mouse out

When the mouse exits an element, or one of its children.

On mouse up

When the user releases the mouse button, while the mouse is inside the element.

on error

When an error occurs while loading an external file (most often used on media files).

on load

When the elements completes loading (mostly used on media files, but can be used also for <body>).

JavaScript - Form Validation

Form validation normally used to occur at the server, after the client had entered all the necessary data and then pressed the Submit button. If the data entered by a client was incorrect or was simply missing, the server would have to send all the data back to the client and request that the form be resubmitted with correct information. This was really a lengthy process which used to put a lot of burden on the server.

JavaScript provides a way to validate form's data on the client's computer before sending it to the web server. Form validation generally performs two functions.

Basic Validation − First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.

Data Format Validation − Secondly, the data that is entered must be checked for correct form and value. Your code must include appropriate logic to test correctness of data.

Example

We will take an example to understand the process of validation. Here is a simple form in html format.

```
<html>
  <head>
    <title>Form Validation</title>
    <script type = "text/javascript">
      <!--
        // Form validation code will come here.
      //-->
    </script>
  </head>

  <body>
    <form action = "/cgi-bin/test.cgi" name = "myForm" onsubmit = "return(validate());">
```

```
<table cellspacing = "2" cellpadding = "2" border = "1">

  <tr>
    <td align = "right">Name</td>
    <td><input type = "text" name = "Name" /></td>
  </tr>

  <tr>
    <td align = "right">EMail</td>
    <td><input type = "text" name = "EMail" /></td>
  </tr>

  <tr>
    <td align = "right">Zip Code</td>
    <td><input type = "text" name = "Zip" /></td>
  </tr>

  <tr>
    <td align = "right">Country</td>
    <td>
      <select name = "Country">
        <option value = "-1" selected>[choose yours]</option>
        <option value = "1">USA</option>
        <option value = "2">UK</option>
        <option value = "3">INDIA</option>
      </select>
    </td>
  </tr>

  <tr>
```

```
            <td align = "right"></td>
            <td><input type = "submit" value = "Submit" /></td>
          </tr>

        </table>
      </form>
    </body>
</html>
```

History of Python

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

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Need of Python Programming:

Python can be used on a server to create web applications. Python can be used alongside software to create workflows. Python can connect to database systems. It can also read and modify files. Python can be used to handle big data and perform complex mathematics. Python can be used for rapid prototyping, or for production-ready software development. Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc). Python has a simple syntax similar to the English language.

Python Features

Python's features include −

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

- Interactive Mode − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- Portable − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- Extendable − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- Databases − Python provides interfaces to all major commercial databases.
- GUI Programming − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- Scalable − Python provides a better structure and support for large programs than shell scripting.

Running Python Scripts:

Let us write a simple Python program in a script. 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 −
#!/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!

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. 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.

Keywords

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

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.

Comments

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.

Variables

Variables are nothing but reserved memory locations to store values. Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.
The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

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

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example
a = b = c = 1
Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example
a,b,c = 1,2,"john"
Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

The data stored in memory can be of many types. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Numbers

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

```
var1 = 1
```

```
var2 = 10
```

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

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

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

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

Python supports four different numerical types −

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

Strings

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

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

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

Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example −

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print list          # Prints complete list
```

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

Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses. The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([])

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

Input

To allow flexibility we might want to take the input from the user. In Python, we have the input() function to allow this. The syntax for input() is

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Output

We use the print() function to output data to the standard output device.

```
print('This sentence is output to the screen')
a = 5
print('The value of a is', a)
```

UNIT-4

Operators

Operators are the constructs which can manipulate the value of operands. Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Python Arithmetic Operators

Operator	Description
+ Addition	Adds values on either side of the operator.
- Subtraction	Subtracts right hand operand from left hand operand.
* Multiplication	Multiplies values on either side of the operator

/ Division	Divides left hand operand by right hand operand
% Modulus	Divides left hand operand by right hand operand and returns remainder
** Exponent	Performs exponential (power) calculation on operators
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −

Python Comparison Operators

Operator	Description
==	If the values of two operands are equal, then the condition becomes true.
!=	If values of two operands are not equal, then condition becomes true.
<>	If values of two operands are not equal, then condition becomes true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

Python Assignment Operators

Operator	Description
=	Assigns values from right side operands to left side operand

+= Add AND	It adds right operand to the left operand and assign the result to left operand
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand
/= Divide AND	It divides left operand with the right operand and assign the result to left operand
%= Modulus AND	It takes modulus using two operands and assign the result to left operand
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand
//= Floor Division	It performs floor division on operators and assign value to the left operand

Python Bitwise Operators

Operator	Description
& Binary AND	Operator copies a bit to the result if it exists in both operands
| Binary OR	It copies a bit if it exists in either operand.
^ Binary XOR	It copies the bit if it is set in one operand but not both.
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.

Python Logical Operators

Operator	Description
and Logical AND	If both the operands are true then condition becomes true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.
not Logical NOT	Used to reverse the logical state of its operand.

Python Membership Operators

Operator	Description
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Python Identity Operators

Operator	Description
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

Python Operators Precedence

Sr.No.	Operator & Description
1	** Exponentiation (raise to the power)

2	~ + - Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % // Multiply, divide, modulo and floor division
4	+ - Addition and subtraction
5	>> << Right and left bitwise shift
6	& Bitwise 'AND'
7	^ | Bitwise exclusive `OR' and regular `OR'
8	<= < > >= Comparison operators
9	<> == != Equality operators
10	= %= /= //= -= += *= **= Assignment operators
11	is is not Identity operators
12	in not in Membership operators
13	not or and Logical operators

Data Structures

Computers store and process data with an extra ordinary speed and accuracy. So it is highly essential that the data is stored efficiently and can be accessed fast. Data structures deal with how the data is organized and held in the memory when a program processes it.

Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets.

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 −

#!/usr/bin/python

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

list2 = [1, 2, 3, 4, 5, 6, 7]

print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]

When the above code is executed, it produces the following result −
list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

Updating Lists

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

#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000]

print "Value available at index 2 : "

```
print list[2]
list[2] = 2001
print "New value available at index 2 : "
print list[2]
```

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]
```

Basic List Operations

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

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.

tup1 = ('physics', 'chemistry', 1997, 2000);

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

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

Accessing Values in Tuples

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

#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5, 6, 7);

print "tup1[0]: ", tup1[0];

print "tup2[1:5]: ", tup2[1:5];

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

tup1[0]: physics

tup2[1:5]: [2, 3, 4, 5]

Updating Tuples

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

#!/usr/bin/python

tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');

```
tup3 = tup1 + tup2;
print tup3;
```

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

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

Delete Tuple Elements

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

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

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

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

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
 File "test.py", line 9, in <module>
     print tup;
NameError: name 'tup' is not defined
```

Basic Tuples Operations

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

Sets

Set is a collection of items not in any particular order. The elements in the set cannot be duplicates. The elements in the set are immutable (cannot be modified) but the set as a whole is mutable. There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it's elements and carry out these mathematical operations as shown below.

Creating a set

A set is created by using the set() function or placing all the elements within a pair of curly braces.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

Accessing Values in a Set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
```

```
for d in Days:
        print(d)
```

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

```
Wed
Sun
Fri
Tue
Mon
Thu
Sat
```

Adding Items to a Set

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
Days.add("Sun")
print(Days)
```

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

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Removing Item from a Set

We can remove elements from a set by using discard() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
Days.discard("Sun")
print(Days)
```

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

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB
print(AllDays)
```

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

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed'])
```

Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element “Wed” is present in both the sets so it will not be found in the result set.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Mon', 'Tue'])
```

Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
SubsetRes = DaysA <= DaysB
SupersetRes = DaysB >= DaysA
print(SubsetRes)
print(SupersetRes)
```

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

```
True
True
```

Dictionary

In 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 −

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

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

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

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

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

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

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

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

Updating Dictionary

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

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

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

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

Delete Dictionary Elements

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

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

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
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 −

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print "dict['Name']: ", dict['Name']
```

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

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

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

```
#!/usr/bin/python
dict = {['Name']: 'Zara', 'Age': 7}
print "dict['Name']: ", dict['Name']
```

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

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = {['Name']: 'Zara', 'Age': 7};
```

TypeError: list objects are unhashable

UNIT-5

Control Flow

if:

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

```
if test expression:
   statement(s)
```

Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True. If the text expression is False, the statement(s) is not executed. Python interprets non-zero values as True. None and 0 are interpreted as False.

Ex:

```
num = 3
if num > 0:
        print(num, "is a positive number.")
print("This is always printed.")
```

if-else:

```
if test expression:
   Body of if
else:
   Body of else
```

The if..else statement evaluates test expression and will execute body of if only when test condition is True.If the condition is False, body of else is executed. Indentation is used to separate the blocks.

Ex:

```
num = 3
if num >= 0:
        print("Positive or Zero")
else:
        print("Negative number")
```

for:

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

Syntax:

```
for val in sequence:
```

```
        Body of for
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Ex:
```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
sum = 0
for val in numbers:
        sum = sum+val
print("The sum is", sum)
```

while:

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

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

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

Ex:
```
n = 10
sum = 0
i = 1
while i <= n:
        sum = sum + i
        i = i+1    # update counter
print("The sum is", sum)
```

break statement:

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Ex:
```
for val in "string":
if val == "i":
        break
        print(val)
```

```
        print("The end")
```

output:
s
t
r
The end

continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```
Ex:     for val in "string":
        if val == "i":
                continue
        print(val)
        print("The end")
```

output:
s
t
r
n
g
The end

pass statement:

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when pass is executed. It results into no operation (NOP).Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. So, we use the pass statement to construct a body that does nothing.

```
Ex:     sequence = {'p', 'a', 's', 's'}
        for val in sequence:
                pass
```

We can do the same thing in an empty function or class as well.

```
def function(args):
   pass
```

class example:

pass

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for the application and a high degree of code reusing. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Defining a Function:

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 of Function

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

ex:
```
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
```

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.

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.

Ex:
```
# 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" )
```

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.

Ex:

```
# 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 ):
        "docstring"
```

return [expression]

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
#!/usr/bin/python
# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

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

```
Output is:
10
Output is:
70
60
50
```

The 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.

Scope of Variables

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

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

- Global variables
- Local variables

Global vs. Local variables

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

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

```
#!/usr/bin/python
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
        # Add both the parameters and return them."
        total = arg1 + arg2; # Here total is local variable.
        print "Inside the function local total : ", total
        return total;
# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

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

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

Fruitful Functions(Function Returning Values)

A fruitful function is one that returns a value.

Ex:
```
def area(radius):
        temp = math.pi * radius**2
```

```
    return temp
```

In a fruitful function the return statement includes an expression. This statement means: “Return immediately from this function and use the following expression as a return value. As soon as a return statement executes, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):
  if x < 0:
    return -x
  if x > 0:
    return x
```

This function is incorrect because if x happens to be 0, neither condition is true, nor the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is none, which is not the absolute value of 0.