LECTURE NOTES

ON

SCRIPTING LANGUAGES

IV B.TECH II Semester

(JNTUH- R15)

Y.HARIKA
Assistant Professor

CH.SRI VIDYA
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(AUTONOMOUS)

DUNDIGAL, HYDERABAD - 500 043

UNIT-1

Scripts and programs

Scripting is the action of typing 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 makes 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 i,e an interpreted scripting language and is designed or suited for writing glue code – code is to connect software parts. 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.

Glue language examples:

• AppleScript • ColdFusion • DCL • Embeddable Common Lisp • ecl • Erlang • JCL • JScript and JavaScript • Lua	• m4 • Perl • PHP • Pure • Python • Rebol • Rexx • Ruby	• Scheme • Tcl • Unix Shell scripts (ksh, csh,bash, sh and others) • VBScript • Work Flow Language • Windows PowerShell • XSLT

2. Using a scripting language to 'manipulate, customize and automate the facilities of an existing system', as the ECMA Script definition puts it. Here the script is used to control an application that provides a programmable interface: this may be an API, though more commonly the application is constructed from a collection of objects whose properties and methods are exposed to the scripting language. Example: use of Visual Basic for applications to control the applications in the Microsoft Office Suite.

3. Using a scripting language with its rich functionality and ease of use as an alternate to a conventional language for general programming tasks ,particularly system programming and administration. Examples: are UNIX system administrators have for a long time used scripting languages for system maintenance tasks, and administrators of WINDOWS NT systems are adopting a scripting language ,PERL for their work.

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 a need to issue a separate command to compile the program and then to run the resulting object file, and without the need to link extensive libraries into the object code. This is done automatically. A few SL'S are indeed implemented as strict interpreters.

- Low overheads and ease of use:

Variables can be declared by using the number of different data types is usually limited everything is a string by context it will be converted as number(vice versa). Number of data structures is limited(arrays)

- Enhanced functionality: SL's usually have enhanced functionality in some areas. For example, most languages provide string manipulation based on the use of regular expressions, while other languages provide easy access to low-level operating system facilities, or to the API, or object exported by an application.
- Efficiency is not an issue: ease of use is achieved at the expense of efficiency 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 maybe 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

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 a 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.

1. 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,
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

1. Command scripting languages are the oldest class of scripting expressions. They appeared in 1960 when a need for programs and tasks control arises. The most known language from the first generation of such type of languages is JCL (Job Control Language), created by IBM OS/360 operating system. Examples of such languages are shell language, described above, and also text-processing languages, such as sed and awk. These languages were one of the first to directly include support for regular expression matching - a feature that later was included into more general-purpose languages, such as Perl.

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.

Dynamically generated HTML

Another form of dynamic Web page is one in which some or all of the HTML is generated by scripts executed on the server. A common application of the technique is to construct pages whose content is retrieved from a database. For example, Microsoft's IIS web server implements Active Server Pages (ASP), which incorporate scripts in Jscript or VBScript.

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
- 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 enterprise- wide document management system for a leading aerospace company.

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 names start with a special character that denotes the kind of thing that the name stands for - scalar data ($), array (@), hash (%), subroutine (&) etc. The syntax also allows a name that consists of a single non-alphanumeric character after the initial special character, eg. $$, $?; such names are usually reserved for the Perl system.

If we write an assignment, eg. j=j+1, the occurrence of j on the left denotes a storage location, while the right-hand occurrence denotes the contents of the storage location. We sometimes refer to these as the lvalue and rvalue of the variable: more precisely we are determining the meaning of the identifier in a left-context or a right-context. In the assignment a[j] = a[j] + 1, both occurrences of j are determined in a right-context, even though one of them appears on the left of the assignment.

In conventional programming languages, new variables are introduced by a declaration, which specifies the name of the new variable and also its type, which determines the kind of value that can be stored in the variable and, by implication, the operations that can be carried out on that variable.

Strings and numbers

In common with many scripting languages, Perl recognizes just two kinds of scalar data: strings and numbers. There is no distinction between integer and real numbers as different types. Perl is a dynamically typed language: the system keeps track of whether a variable contains a numeric value or a string value, and the user doesn't have to worry about the difference between strings and numbers since conversions between the two kinds of data are done automatically as required by the context in which they are used.

Boolean values

All programming languages need some way of representing truth values and Perl is no exception. Since scalar values are either numbers or strings, some convention is needed for representing Boolean values, and Perl adopts the simple rule that numeric zero, "0" and the empty string (" ") mean false, and anything else means true.

Numeric constants

Numeric constants can be written in a variety of ways, including specific notation, octal and hexadecimal. Although Perl tries to emulate natural human communication, the common practice of using commas or spaces to break up a large integer constant into meaningful digit groups cannot be used, since the comma has a syntactic significance in Perl. Instead, underscores can be included in a number literal to improve legibility.

String constants

String constants can be enclosed in single or double quotes. The string is terminated by the first next occurrence of the quote which started it, so a single-quoted string can include double quotes and vice versa. The q (quote) and qq (double quote) operators allow you to use any character as a quoting character. Thus q / any string/ or q (any string) are the same as 'any string' and qq / any string / or qq (any string) are the same as "any string"

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 ;

$b = "$ { a } Script" ; which gives $b the value "JavaScript".

<STDIN> - a special value

When the 'variable' <STDIN> appears in a context where a scalar value is required, it evaluates to a string containing the next line from standard input, including the terminating newline. If there is no input queued, Perl will wait until a line is typed and the return key pressed. The empty string is treated as false in a Boolean context. If <STDIN> appears on the right-hand side of an assignment to a scalar variable, the string containing the input line is assigned to the variable named on the ;eft. If it appears in any other scalar context the string is assigned to the anonymous variable: this can be accessed by the name $- : many operations use it as a default.

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 never-used 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.

Two examples illustrate this.

1.Auto increment

If a variable has only ever been used in a string context, the auto increment operator can be applied to it. If the value consists of a sequence of letters, or a sequence of letters followed by a sequence of digits, the auto increment takes place in string mode starting with the right most character, with ‘carry’ along the string. For example, the sequence

```
$a = ‘a0’ ; $b =
‘Az9’ ;

Print ++$a, ‘ ‘, ++$b;

“/n”; Prints a1 Ba0.
```

2.Unaryminus

This has an unusual effect on non numeric values. Unary minus applied to a string which starts with a plus or minus character returns the same string, but starting with the opposite sign. Unary minus applie to an identifier returns a string consists of minus prefixed to the characters of the identifiers. Thus if we have a variable named $config with the value “ foo”, then –config evaluates the string “-foo”. This is useful, for example, in constructing command arguments are introduced by

Comparison operators

The value of comparisons is returned as 1 if true, and an empty string (“ ”) if false, in accordance with the convention described earlier. Two families of comparison operators provide, one for numbers and one for strings. The operator used determines the context , and perl converts the operands as required to match the operator. This duality is necessary because a comparison between strings made up entirely numerical digits should apply the usual rules for sorting strings ASCII as a collating sequence, and this may not give the same result as the numerical comparison(‘5’ <’10’) returns the value true as a numerical comparison having been converted into (5<10) where as the string comparison (‘5’ lt ‘10’) returns false, since 10 comes before 5 in the canonical sort order for ASCII strings. The comparison operator (for numbers, cmp for strings), performs a three way test, returning -1 for less-than, 0 for equal an +1 for greater-than. Note that the comparison operators are non associative, so an expression like $a > $b > $c Is erroneous.

Logical operators

The logical operators allows to combine conditions using the usual logical operations ‘not’(!, not), ‘and’(&&,and) and ‘or’(||,or). Perl implements the ‘and’ and ‘or’ operators in ‘shortcut’ mode, i.e evaluation stops as soon as the final result is certain using the rules false &&b=false, and true||b=true. Before Perl 5,only the !, && and || operators were provided. The new set, not, and ,or, are provided partly to increase readability, and partly because their extra-low precedence makes it possible to omit brackets in most circumstances-the precedence ordering is chosen so that numerical expressions can be compared without having to enclose them in brackets, e.g. Print” OK\n” if $a<10 and $b<12;

Bitwise operators

The unary tilde(~) applied to a numerical argument performs bitwise negation on its operand, generating the one’s compliment. If applied to a string operand it complements all the bits in the string – effective way of inverting a lot of bits. The remaining bitwise operators - & (and), | (or) and ^(exclusive or)- have a rather complicated definition. If either operand is a number or a variable that has previously been used as a number, both operands are converted to integers if need be, and the bitwise operation takes place between the integers. If the both operands are strings, and if variables have never been used as numbers, Perl performs the bitwise operation

between corresponding bits in the two strings, padding the shorter strings with zeros as required.

Conditional expressions

A conditional expression is one whose values is chosen from two alternatives at run-time depending on the outcome of a test. The syntax is borrowed from C

Test ? true_exp: false_exp

The first expression is evaluated as Boolean value : if it returns true the whole expression is replaced by true_exp, otherwise it is replaced by false_exp, e.g.

$a= ($a<0)? 0 : $a;

Control structures

The Control Structures for conditional execution and repetition all the control mechanisms is similar to C.

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.

Conditions

A condition is a Perl expression which is evaluated in a Boolean context: if it evaluates to zero or the empty string the condition is false, otherwise it is true. Conditions usually make use of relational operators. Eg: $total>50

$total>50 and $total<100

Simple Conditions can be combined into a complex condition using the logical operators. A condition can be negated using the ! operator.

Eg: !($total>50 and $total<100)

CONDITIONAL EXECUTION

If-then-else statements

```
if ($total>0)
{
   print "$total\n"-
  if ($total>0)
  {
  print "$total\n"
  }
 else
 {
 print "bad total!\n"-
 }
 }
```

A single statement is a block and requires braces round it. The if statement requires that the expression forming the condition is enclosed in brackets. The construct extends to multiple selections

```
Eg: if ($total>70)
{
$grade="A";
}
 elsif ($total >50)
{
$grade="B";
}
elsif ($total>40)
{
$grade="C";
} else {
```

```
$grade=”F”;
$total=0;
 }
```

Alternatives to if-then-else

To use a conditional expression in place of an if-then-else construct.

```
if ($a<0)
($b=0)
else
($b=1)
```

can be written as

```
$b= ($a<0)? 0:1;
```

To use the ‘or’ operator between statements Eg: open (IN, $ARGV[0] or die “Can’t open $ARGV*0+\n”;

Statement qualifiers

A single statement(not a block) can be followed by a conditional modifier.

```
Eg:   print “OK\n”
      if      $volts>=1.5;
      print “Weak\n”
      if
      $volts>=1.2 and
      $volts<1.5;
       print “Replace\n”
      if      $volts<1.2;
```

Code using Conditional expressions,

```
Eg: print (($volts>=1.5)? “Ok\n”; (($volts>=1.2)? “Weak\n”; “Replace\n”));
```

REPETITION:

Repetition mechanisms include both

Testing Loops

Counting Loops

TESTING LOOPS

```
While ($a! = $b)
if ($a > $b) {
$a=$a-$b
} else {
$b=$b-$a
}
}
```

With the if statement, the expression that forms the condition must be enclosed in brackets. But now, while can be replaced by until to give the same effect. Single statement can use while and until as statement modifiers to improve readability.

```
Eg: $a += 2 while $a <$b;
$a += 2 until $a > $b;
```

Here, although the condition is written after the statement, it is evaluated before the statement is executed, if the condition is initially false the statement will be never executed. When the condition is attached to a do loop is same as a statement modifier, so the block is executed at least once.

```
do
{
…….
} while $a! = $b;
```

Counting Loops

In C,

```
for ($i= 1;$i<=10;$i++)
 {
$i_square=$i*$i;
$i_cube=$i**3;
print “$i\t$i_square\t$i_cube\n”;
}
```

In Perl,

```
foreach $i (1…10),
$i_square=$i* $i;
$i_cube=$i**3;
print "$i\t$i_square\t$i_cube\n";
}
```

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 comparisons 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;

Array Creation

Array variables are prefixed with the @sign and are populated using either parenthesis or the qw operator.

Eg: @array = (1 , 2 ,"Heelo");

@array = qw/This is an array/;

In C, C++, Java; Array is a collection of homogeneous elements, whereas; In Perl, Array is collection of heterogeneous elements.

Accessing Array Elements

- When accessing an individual element, we have to use the '$' symbol followed by variable name along with the index in the square brackets.

  Eg: $bar = $foo[2];

  $foo[2] = 7;
- A group of contiguous elements is called a slice , and is accessed using a simple syntax:

  @foo[1..3] is the same as the list ($foo[1], $foo[2], $foo[3])
- A slice can be used as the destination of an assignment, Eg: @foo*1..3+ = ("hop" , "skip" , "jump");
- Like a slice, a selection can appear on the left of an assignment: this leads to a useful idiom for rearranging the elements in a list.

  Eg: To swap the first two elements of an array, we write as;

```
@foo[0 , 1] = @foo[1 , 0];
```

Manipulating Lists

Perl provides several built-in functions for list manipulation. Three useful ones are:

- **shift LIST :** Returns the first item of LIST, and moves the remaining items down, reducing the size of LIST by 1.
- **unshift ARRAY, LIST :** The opposite of shift. Puts the items in LIST at the beginning of ARRAY, moving the original contents up by the required amount.
- **push ARRAY, LIST :** Similar to unshift, but adds the values in LIST to the end of ARRAY.

Iterating over Lists

foreach: The foreach loop performs a simple iteration over all the elements of a list.

```
Eg: foreach $item (list)
        {
         ……………

        }
```

The block is executed repeatedly with the variables $item taking each value from the list in turn. The variable can be omitted, in which case $_ will be used.The natural Perl idiom for manipulating all items in an array is ;

```
 foreach (@array)
{
……..#process $_

}
```

Working With Hashes

- A hash is a set of key/value pairs.
- Hash variables are preceded by a "%" sign.
- To refer to a single element of a hash, you will use the hash variable name preceded by a '$' sign and followed by the "key" associated with the value in the curly brackets.
- It is also called as associative array.

Creating Hashes

We can assign a list of key-value pairs to a hash, as, for example

%foo = (key1, value1, key2, value2, …….);

An alternative syntax is provided using the => operator to associate key-value pairs, thus:%foo = (banana => 'yellow' , apple => 'red' , grapes => 'green', ...);

Manipulating Hashes

Perl provides a number of built-in functions to facilitate manipulation of hashes. If we have a hash called HASH

- keys % HASH returns a list of the keys of the elements in the hash, and
- values % HASH returns a list of the values of the elements in the hash.

Eg: %foo = (banana => 'yellow' , apple => 'red' , grapes => 'green', ...); keys % HASH returns banana, apple ,grapes values % HASH returns yellow, red, green.

These functions provide a convenient way to iterate over the elements of a hash using foreach:

```
foreach (keys % HASH)
 {
    process $magic($_)
}
```

Other useful operators for manipulating hashes are delete and exists.

- delete $HASH{$key} removes the element
- exists $HASH{$key} returns true.

Strings, Pattern Matching & Regular Expressions in Perl

The most powerful features of Perl are in its vast collection of string manipulation operators and functions. Perl would not be as popular as it is today in bioinformatics applications if it did not contain its flexible and powerful string manipulation capabilities.

String concatenation

To concatenate two strings together, just use the (.) dot

- $a . $b;
- $c = $a . $b;
- $a = $a . $b;
- $a .= $b;

The first expression concatenates $a and $b together, but the the result was immediately lost unless it is saved to the third string $c as in case two. If $b is meant to be appended to the end of $a, use the .= operator will be more convenient. As is any other assignments in Perl, if you see an assignment written this way $a = $a op expression, where op stands for any operator and expr stands for the rest of the statement, you can make a shorter version by moving the op to the front of the assignment, e.g., $a op= expression.

Substring extraction

The counterpart of string concatenation is substring extraction. To extract the substring at certain location inside a string, use the substr function:

- $second_char = substr($a, 1, 1);
- $last_char = substr($a, -1, 1);
- $last_three_char = substr($a, -3);

The first argument to the substr function is the source string, the second argument is the start position of the substring in the source string, and the third argument is the length of the substring to extract. The second argument can be negative, and if that being the case, the start position will be counted from the back of the source string. Also, the third argument can be omitted. In that case, it will run to the end of the source string. Particularly interesting feature in Perl is that the substr function can be assigned into as well, meaning that in addition to string extraction, it can be used as string replacement:

- substr($a, 1, 1) = 'b'; # change the second character to b
- substr($a, -1) = 'abc'; # replace the last character as abc (i.e., also add two new letters c)

- substr($a, 1, 0) = 'abc'; #insert abc in front of the second character

Substring search

In order to provide the second argument to substr, usually you need to locate the substring to be extracted or replaced first. The index function does the job:

- $loc1 = index($string, "abc");
- $loc2 = index($string, "abc", $loc+1);
- print "not found" if $loc2<0;

The index function takes two arguments, the source string to search, and the substring to be located inside the source string. It can optionally take a third argument to mean the start position of the search. If the index function finds no substring in the source string anymore, then it returns -1.

Regular expression

Regular expression is a way to write a pattern which describes certain substrings. In general, the number of possible strings that can match a pattern is large, thus you need to make use of the regular expression to describe them instead of listing all possibilities. If the possible substring matches are just one, then maybe the index function is more efficient.

The following are some basic syntax rules of regular expression:

- Any character except the following special ones stands for itself. Thus abc matches 'abc', and xyz matches 'xyz'.
- The character . matches any single character. To match it only with the . character itself, put an escape \ in front of it, so \. will match only '.', but . will match anything. To match the escape character itself, type two of them \\ to escape itself.
- If instead of matching any character, you just want to match a subset of characters, put all of them into brackets [ ], thus [abc] will match 'a', 'b', or 'c'. It is also possible to shorten the listing if characters in a set are consecutive, so [a-z] will match all lowercase alphabets, [0-9] will match all single digits, etc. A character set can be negated by the special ^ character, thus [^0-9] will match anything but numbers, and [^a-f] will match

anything but 'a' through 'f'. Again, if you just want to match the special symbols themselves, put an escape in front of them, e.g., \[, \^ and \].

- All the above so far just match single characters. The power of regular expression lies in its ability to match multiple characters with some meta symbols. The * will match 0 or more of the previous symbol, the + will match 1 or more of the previous symbol, and ? will match 0 or 1 of the previous symbol.
- For example, a* will match 'aaaa...' for any number of a's including none '', a+ will match 1 or more a's, and a? will match zero or one a's. A more complicated example is to match numbers, which can be written this way [0-9]+. To matching real numbers, you need to write [0-9]+\.?[0-9]*. Note that the decimal point and fraction numbers can be omitted, thus we use ?, and * instead of +.
- If you want to combine two regular expressions together, just write them consecutively.If you want to use either one of the two regular expressions, use the | meta symbol. Thus, a|b will match a or b, which is equivalent to [ab], and a+|b+ will match any string of a's or b's. The second case cannot be expressed using character subset because [ab]+ does not mean the same thing as a+|b+.
- Finally, regular expressions can be grouped together with parentheses to change the order of their interpretation. For example, a(b|c)d will match 'abd' or 'acd'. Without the parentheses, it would match 'ab' or 'cd'.

The rules above are simple, but it takes some experience to apply them successfully on the actual substrings you wish to match. There are no better ways to learn this than simply to write some regular expressions and see if they match the substrings you have in mind.

The following are some examples:

- [A-Z][a-z]* will match all words whose first character are capitalized
- [A-Za-z_][A-Za-z0-9_]* will match all legal perl variable names
- [+-]?[0-9]+\.?[0-9]*([eE][+-]?[0-9]+)? will match scientific numbers
- [acgtACGT]+ will match all DNA strings
- ^> will match the > symbol only at the beginning of a string
- a$ will match the a letter only at the end of a string

In the last two examples above, we introduced another two special symbols. The ^ which when not used inside a character set to negate the character set, stands for the beginning of the string. Thus, ^> will match '>' only when it is the first character of the string. Similarly, $ inside a regular expression means the end of the string, so a$ will match 'a' only when it is the last character of the string. These are so called anchor symbols.

Another commonly used anchor is \b which stands for the boundary of a word. In addition, Perl introduces predefined character sets for some commonly used patterns, thus \d stands for digits and is equivalent to [0-9], \w stands for word letters or numbers, and \s stands for space characters ' ', \t, \n, \r, etc. The captial letter version of these negates their meaning, thus \D matches non-digit characters, \W matches non-word characters, and \S matches non-whitespaces. The scientific number pattern above can therefore be rewritten as:

- [+-]?\d+\.?\d*([eE][+-]?\d+)?

Pattern matching

Regular expressions are used in a few Perl statements, and their most common use is in pattern matching. To match a regular expression pattern inside a $string, use the string operator =~ combines with the pattern matching operator / /:

- $string =~ /\w+/; # match alphanumeric words in $string
- $string =~ /\d+/; # match numbers in $string

he pattern matching operator / / does not alter the source $string. Instead, it just returns a true or false value to determine if the pattern is found in $string:

- ```
  if ($string =~ /\d+/)
  {
    print "there are numbers in $string\n";
  }
  ```

Sometimes not only you want to know if the pattern exists in a string, but also what it actually matched. In that case, use the parentheses to indicate the matched substring you want to know, and they will be assigned to the special $1, $2, ..., variables if the match is successful:

- ```
  if ($string =~ /(\d+)\s+(\d+)\s+(\d+)/)
  {
    print "first three matched numbers are $1, $2, $3 in $string\n";
  }
  ```

Note that all three numbers above must be found for the whole pattern to match successfully, thus $1, $2 and $3 should be defined when the if statement is true. The same memory of matched substrings within the regular expression are \1, \2, \3, etc. So, to check if the same number happened twice in the $string, you can do this:

```
if ($string =~ /(\d).+\1/) {
 print "$1 happened at least twice in $string\n";
}
```

You cannot use $1 in the pattern to indicate the previously matched number because $ means the end of the line inside the pattern. Use \1 instead.

Pattern substitution

In addition to matching a pattern, you can replace the matched substring with a new string using the substitution operator. In this case, just write the substitution string after the pattern to match and replace:

```
$string =~ s/\d+/0/; # replace a number with zero
$string =~ s:/:\\:; # replace the forward slash with backward slash
```

Unlike the pattern matching operator, the substitution operator does change the $string if a match is found. The second example above indicates that you do not always need to use / to break the pattern and substitution parts apart; you can basically use any symbol right after the s operator as the separator. In the second case above, since what we want to replace is the forward slash symbol, using it to indicate the pattern boundary would be very cumbersome and need a lot of escape characters:

```
$string =~ s/\/\\/; # this is the same but much harder to read
```

For pattern matching, you can also use any separator by writing them with m operator, i.e., m:/: will match the forward splash symbol. Natually, the substitution string may (and often does) contain the \1, \2 special memory substrings to mean the just matched substrings. For example, the following will add parentheses around the matched number in the source $string:

```
$string =~ s/(\d+)/(\1)/;
```

The parentheses in the replacement string have no special meanings, thus they were just added to surround the matched number.

Modifiers to pattern matching and substitution

You can add some suffix modifiers to Perl pattern matching or substitution operators to tell them more precisely what you intend to do:

- /g tells Perl to match all existing patterns, thus the following prints all numbers in $string

  while ($string =~ /(\d+)/g)

  {

  print "$1\n";

  }
- $string =~ s/\d+/0/g; # replace all numbers in $string with zero
- /i tells Perl to ignore cases, thus

  $string =~ /abc/i; # matches AbC, abC, Abc, etc.
- /m tells perl to ignore newlines, thus

  "a\na\na" =~ /a$/m will match the last a in the $string, not the a before the first newline if /m is not given.

Perl- Subroutines

A Perl subroutine or function is a group of statements that together performs a task. You can divideup your code into separate subroutines. How you divide up your code among different subroutines is up to you, but logically the division usually is so each function performs a specific task.

Define and Call a Subroutine

The general form of a subroutine definition in Perl programming language is as follows –

```
sub subroutine_name
{
body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows – subroutine_nam e(list of argum ents);Let's have a look into the following example, which defines a simple function and then call it. Because Perl compiles your program before executing it, it doesn't matter

where you declare your subroutine.

```
#!/usr/bin/perl
# Function definition
 sub Hello{
print "Hello, World!\n";
}
# Function call Hello();
```

When above program is executed, it produces the following result − Hello, World!

Passing Arguments to a Subroutine

You can pass various arguments to a subroutine like you do in any other programming language and they can be accessed inside the function using the special array @_. Thus the first argument to the function is in [0], these concision_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities.

Passing Lists to Subroutines

Because the @_ variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from @_. If you have to pass a list along with other scalar arguments, then make list as the last argument as shown below −

```
#!/usr/bin/perl
# Function
definition sub
PrintList{
m y @ list = @ _;
print "Given list is @ list\n";
}
$ a = 10;
@ b = (1, 2, 3, 4);
# Function call with list param
eter PrintList($ a, @ b);
```

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

Given list is 10 1 2 3 4

Passing Hashes to Subroutines

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically translated into a list of key/value pairs. For example −

```
#!/usr/bin/perl

# Function definition sub
PrintHash{

m y (%hash) = @ _;

foreach m y $ key ( keys %hash
){ m y $ value = $ hash{$ key};

print "$ key : $ value\n";

}}

%hash = ('nam e' => 'Tom ', 'age' => 19);
# Function call with hash parameter
PrintHash(%hash);
```

When above program is executed, it produces the following result − nam

ee : Tom

age : 19

Returning Value from a Subroutine

You can return a value from subroutine like you do in any other programming language. If you are not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value. You can return arrays and hashes from the subroutine like any scalar but returning more than one array or hash normally causes them to lose their separate identities. So we will use references explained in the next chapter to return any array or hash from a function. Let's try the following example, which takes a list of numbers and then returns their average

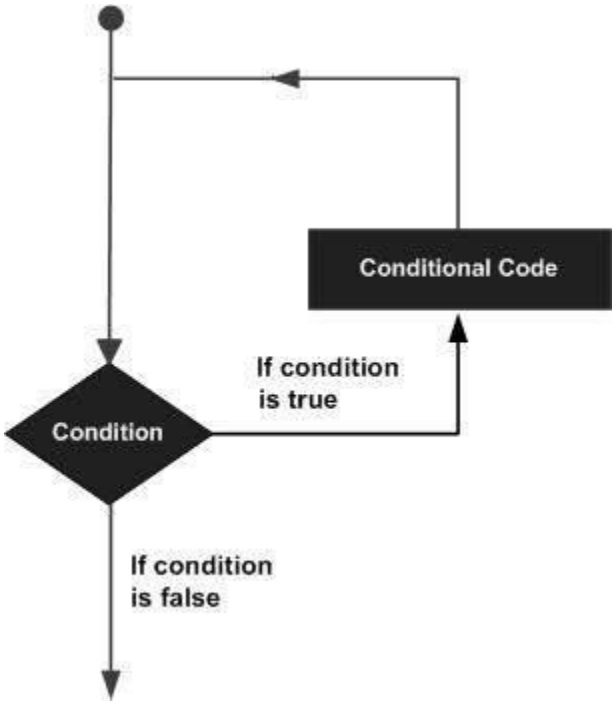
```
#!/usr/bin/perl
# Function definition sub Average{
# get total num ber of argum ents passed.
$ n = scalar(@ _);
$ sum = 0;
foreach $ item (@ _)
{
$ sum += $ item ;
}
$ average = $ sum / $ n; return $ average;
}
# Function call
$ num = Average(10, 20, 30);
print "Average for the given num bers : $ num \n";
```

When above program is executed, it produces the following result − Average for the given num bers : 20

UNIT-2

Finer Points Of Looping

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



Perl programming language provides the following types of loop to handle the looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
until loop	Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.

for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
foreach loop	The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change the execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.C supports the following control statements. Click the following links to check their detail.

Control Statement	Description
next statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
last statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	A continue BLOCK, it is always executed just before the conditional is about to be evaluated again.
redo statement	The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed.
goto statement	It is used to transfer control by jumping to other label inside a loop.

Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#!/usr/local/bin/perl

for(;;)
{
printf"This loop will run forever.\n";
}
```

You can terminate the above infinite loop by pressing the Ctrl + C keys. When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but as a programmer more commonly use the for (;;) construct to signify an infinite loop.

Multiple Loop Variables:

For loop can iterate over two or more variables

simultaneously. Eg: for($m=1,$n=1,$m<10,$m++,$n+=2)

{

…….

}

Here (,) operator is a list constructor, it evaluates its left hand argument.

Pack and Unpack

Pack Function

The pack function evaluates the expressions in LIST and packs them into a binary structure specified by EXPR. The format is specified using the characters shown in Table below Each character may be optionally followed by a number, which specifies a repeat count for the type of value being packed. that is nibbles, chars, or even bits, according to the format. A value of * repeats for as many values remain in LIST. Values can be unpacked with the

unpack function. For example, a5 indicates that five letters are expected. b32 indicates that 32 bits are expected. h8 indicates that 8 nibbles (or 4 bytes) are expected. P10 indicates that the structure is 10 bytes long.

Syntax

Following is the simple syntax for this function

```
pack EXPR, LIST
```

Return Value

This function returns a packed version of the data in LIST using TEMPLATE to determine how it is coded.

Here is the table which gives values to be used in TEMPLATE.

Character	Description
A	ASCII character string padded with null characters
A	ASCII character string padded with spaces
B	String of bits, lowest first
B	String of bits, highest first
C	A signed character (range usually -128 to 127)
C	An unsigned character (usually 8 bits)
D	A double-precision floating-point number
F	A single-precision floating-point number

H	Hexadecimal string, lowest digit first
H	Hexadecimal string, highest digit first
I	A signed integer
I	An unsigned integer
L	A signed long integer
L	An unsigned long integer
N	A short integer in network order
N	A long integer in network order
P	A pointer to a string
S	A signed short integer
S	An unsigned short integer
U	Convert to uuencode format
V	A short integer in VAX (little-endian) order
V	A long integer in VAX order
X	A null byte
X	Indicates "go back one byte"

@	Fill with nulls (ASCII 0)

Example

```
#!/usr/bin/perl -w

$bits =pack("c",65);
# prints A, which is ASCII
65. print"bits are $bits\n";
$bits =pack("x");
# $bits is now a null chracter.
print"bits are $bits\n";
$bits =pack("sai",255,"T",30);
# creates a seven charcter string on most computers'
print"bits are $bits\n";

@array=unpack("sai","$bits");

#Array now contains three elements: 255, T and 30.
print"Array $array[0]\n";
```

Following is the example code showing its basic usage

```
print"Array $array[1]\n";
print"Array $array[2]\n";
```

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

```
bits are A

bits are
bits  are
T- Array
255 Array T
Array 30
```

Unpack Function

The unpack function unpacks the binary string STRING using the format specified in TEMPLATE. Basically reverses the operation of pack, returning the list of packed values according to the supplied format. You can also prefix any format field with a %<number> to indicate that you want a 16-bit checksum of the value of STRING, instead of the value.

Syntax

Following is the simple syntax for this function

```
unpack TEMPLATE, STRING
```

Return Value

This function returns the list of unpacked values.

Here is the table which gives values to be used in TEMPLATE.

Character	Description
A	ASCII character string padded with null characters
A	ASCII character string padded with spaces
B	String of bits, lowest first
B	String of bits, highest first
C	A signed character (range usually -128 to 127)
C	An unsigned character (usually 8 bits)
D	A double-precision floating-point number
F	A single-precision floating-point number

H	Hexadecimal string, lowest digit first
H	Hexadecimal string, highest digit first
I	A signed integer
I	An unsigned integer
L	A signed long integer
L	An unsigned long integer
N	A short integer in network order
N	A long integer in network order
P	A pointer to a string
S	A signed short integer
S	An unsigned short integer
U	Convert to uuencode format
V	A short integer in VAX (little-endian) order
V	A long integer in VAX order
X	A null byte
X	Indicates "go back one byte"

@	Fill with nulls (ASCII 0)

Example

Following is the example code showing its basic usage

```
print"bits are $bits\n";
$bits =pack("x");
# $bits is now a null chracter. print"bits are
$bits\n";
$bits =pack("sai",255,"T",30);
# creates a seven charcter string on most
computers' print"bits are $bits\n";
@array=unpack("sai","$bits");
#Array now contains three elements: 255, A and
47. print"Array $array[0]\n";
print"Array $array[1]\n";
```

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

```
bits
are A
bits
are
bits
are
T-
Array2
```

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.

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,
```

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.

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")ordie"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 opened in the writing mode.

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

To truncate the file first

```
open DATA,"+>file.txt"ordie"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.

Entities	Definition
< or r	Read Only Access
> or w	Creates, Writes, and Truncates

>> or a	Writes, Appends, and Creates
+< or r+	Reads and Writes
+> or w+	Reads, Writes, Creates, and Truncates
+>> or a+	Reads, Writes, Appends, and Creates

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

Entities	Definition
O_RDWR	Read and Write
O_RDONLY	Read Only
O_WRONLY	Write Only

the table, which gives the possible values of MODE.

O_CREAT	Create the file
O_APPEND	Append the file
O_TRUNC	Truncate the file
O_EXCL	Stops if file already exists
O_NONBLOCK	Non-Blocking usability

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";
```

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.

The<FILEHANDLE> 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")ordie"Can't open data";
@lines=<DATA
>;
close(DATA);
```

getc Function

The getc function returns a single character from the specified FILEHANDLE, or STDIN if none is specified

```
getc FILEHANDLE
getc
```

If there was an error, or the filehandle is at end of file, then undef is returned instead.

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.

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";
```

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 );
```

Renamingafile

Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is

```
#!/usr/bin/perl
rename("/usr/test/file1.txt","/usr/test/file2.txt");
```

available in /usr/test directory.

This function renames the takes two arguments and it just rename existing file.

Deletingan Existing File

Here is an example, which shows how to delete a file file1.txt using the unlink function.

```
#!/usr/bin/perl
unlink("/usr/test/file1.txt");
```

Positioning inside a File

You can use to tell function to know the current position of a file and seek function to point a particular position inside the file.

Tell Function

The first requirement is to find your position within a file, which you do using the tell function.

This returns the position of the file pointer, in bytes, within FILEHANDLE if specified, or the current default selected filehandle if none is specified.

```
tell FILEHANDLE
```

```
tell
```

Seek Function

The seek function positions the file pointer to the specified number of bytes within a file The

```
seek FILEHANDLE, POSITION, WHENCE
```

function uses the fseek system function, and you have the same ability to position relative to three different points: the start, the end, and the current position. You do this by specifying a value for

WHENCE. Zero sets the positioning relative to the start of the file. For example, the line sets the file pointer to the 256th byte in the file.

```
seek DATA, 256, 0;
```

File Information

You can test certain features very quickly within Perl using a series of test operators known collectively as -X tests. For example, to perform a quick test of the various permissions on a file, you might use a script like this

```
#/usr/bin/perl
my $file ="/usr/test/file1.txt";
my(@description, $size);
if(-e $file)
{
push@description,'binary'if(-B _);
push@description,'a socket'if(-S _);
push@description,'a text file'if(-T _);
push@description,'a block special file'if(-b _);
push@description,'a character special file'if(-c _);
push@description,'a directory'if(-d _);
push@description,'executable'if(-x _);
push@description,(($size =-s _))?"$size
bytes":'empty'; print"$file is ", join(',
',@description),"\n";
}
```

Here is the list of features, which you can check for a file or directory −

Operator	Definition
-A	Script start time minus file last access time, in days.
-B	Is it a binary file?
-C	Script start time minus file last inode change time, in days.

-M	Script start time minus file modification time, in days.
-O	Is the file owned by the real user ID?
-R	Is the file readable by the real user ID or real group?
-S	Is the file a socket?
-T	Is it a text file?
-W	Is the file writable by the real user ID or real group?
-X	Is the file executable by the real user ID or real group?
-b	Is it a block special file?
-c	Is it a character special file?
-d	Is the file a directory?
-e	Does the file exist?
-f	Is it a plain file?
-g	Does the file have the setgid bit set?
-k	Does the file have the sticky bit set?
-l	Is the file a symbolic link?
-o	Is the file owned by the effective user ID?

-p	Is the file a named pipe?
-r	Is the file readable by the effective user or group ID?
-s	Returns the size of the file, zero size = empty file.
-t	Is the filehandle opened by a TTY (terminal)?
-u	Does the file have the setuid bit set?
-w	Is the file writable by the effective user or group ID?
-x	Is the file executable by the effective user or group ID?
-z	Is the file size zero?

EVAL

The eval operator comes in 2 forms

In the first form ,eval takes an arbitrary string as an operand and evaluates the string and executed in the current context.

The value returned is the value of the last expression evaluated.In case of syntax error or runtime error,eval returns the value “undefined” and places the error in the variable $@

Ex:

```
$myvar =’ …’;
….
$value = eval “ \$$myvar “;
```

In the second form ,it takes a block as an argument and the block is compiled only once. If there is a runtime error,the error is returned in $@. Instead of try we use eval and instead of catch we test $@

Ex:

```
Eval
{
    …….
```

```
}
If ($@ ne ' ')
{
        ……..
}
```

Data Structures

Arrays Of Arrays

In perl a two dimensional array is constructed by creating an array of references to anonymous arrays.

For ex: @colors = ([35,39,43] , [4,5,8] , [32,31,25]) ;

The array composer converts each comma-separated list to an anonymous array in memory and returns a reference and when we write an exp. Like

$colors [0][1] = 64;

$colors [0] is a reference to an array and 2nd subscript represents the element present in that array. A two dimensional array can be dynamically created by using PUSH operator to add a reference to an anonymous array to the top level array. For ex: we are interested in converting a table set of data having white spaces between the fields can be converted to two dimensional array by repeteadly using split to put the fields of a line into list and then using push to add the reference to an array.

```
While (<STDIN>)
{
Push @table , [split]
}
```

COMPLEX DATA STRUCTURES

Not only an array of arrays can be created but we can create hashes of hashes ,arrays of hashes and hashes of arrays. By combining all these possibilities ,data structures of great complexity can be created ex: doubly linked list. We can make an element of the array a hash containing three fields with keys 'L'(left neighbour) ,' R'(right neighbour) and 'C'(content). The values related to L and R are references to element hashes and the value of C can be anything(scalar,variables,hash ,reference).

Ex:

We can move forwards along the list with

$current = $current->{‘R’} ; And backwards with

$current = $current->{‘L’} ; Create a new element

$new = { L =>undef , R=>undef , C=>…} ;

And we can insert new element after current element as

$new->{‘R’}=$current->{‘R’} ;

$current{‘R’}->{‘L’}= $new;

$current{‘R’}=$new ;

$new->{‘L’} = $current ;

And the current element can be deleted as

$current->{‘L’}->{‘R’} = $current->{‘R’} ;

$current->{‘R’}->{‘L’} = $current->{‘L’} ;

Packages

Packages are the basis for libraries, modules and objects. It is the unit of code with its own namespace(i.e. separate symbol table),which determines bindings of names both at compile-time and run-time. Initially code runs in default package main. Variables used in a package are global to that package only.

Ex:$A::x is the variable x in package A. Package A ;

$x = 0;

……. Package B ;

$x = 1 ;

……. Package A ; Print $x;

output: zero.

The package B declaration switches to a different symbol table then the package A points to the original symbol table having $x =0;.

Nested packages can be created of the form A::B provided the variables should be of the fully qualified form Ex>$A::B::x.

A package can have one or more BEGIN routines and also END routines. Package declaration is rarely used on its own.

Modules

Libraries and modules are packages contained within a single file and are units of program reusability. The power of perl is increased by the usage of modules that provide functionality

in specific application areas. To be fact module is nothing but a package contained in a separate file whose name is same as the package name with the extension .pm and makes use of built-in-support. The use of modules make mathematical routines in the library math.p1 are converted into a module math.pm and can be written as Ex: Use math ; at the start of the program and the subroutines are available .The subroutine names imported are those defined in the export list of the math module and it is possible to suppress the import of names but loses the point of the module.

Ex: use IO : : File ;

Indicates a requirement for the module File.pm which will be found in a directory called IO.

The use of " use math ('sin' , 'cos' , 'tan')" is same as BEGIN {Require " Math.pm" ;

Math :: import ('sin' , 'cos' , 'tan');}

The module names are imported by calling the import() method defined in the module. The package writer is free to define import() in any way.

Objects

Objects in Perl provide a similar functionality as objects in real object oriented programming (OOP), but in a different way. They use the same terminology as OOP, but the words have different meanings as given below.

- Object: An object with in Perl is a reference to a data type that knows what class it belongs to. The object is stored as a reference in a scalar variable. The object is said to be blessed into a class: this is done by calling the built in function bless in a constructor.
- Constructor: A constructor is just a subroutine that returns a reference to an object.
- Class: A class is a package that provides methods to deal with objects that belong to it.

  Method: A method is a subroutine that expects an object reference as its first argument.

Constructors

Objects are created by a constructor subroutine which is generally called new.

Eg.Package Animal;

```
sub new { my $ref ={ };
bless ref; return ref;
}
```

The flower brackets { } returns a reference to an anonymous hash. So the new constructor returns a reference to an object that is an empty hash, and knows that it belongs to the package Animal.

Instances

We can create the instances for the object with this defined constructor as

$Dougal = new Animal;

$Ermyntrude = new Animal;

This makes $Dougal and $Ermyntrude references to objects that are empty hashes, and know that they belong to the Animal class.

Method Invocation

Perl supports two syntactic forms for invoking methods one is by using arrow operator and another one is by using Indirect objects. If a class is used to invoke the method, that argument will be the name of the class. If an object is used to invoke the method, that argument will be the reference to the object. Whichever it is, we'll call it the method's invocant. For a class method, the invocant is the name of a package. For an instance method, the invocant is a reference that specifies an object.

Method Invocation Using the Arrow Operator:

For example if set_species, get_species are the methods they can be invoked using arrow operator as follows.

$Dougal -> set_species ‘Dog’;

$Dougal_is ->= $Dougal->get_species;

Method Invocation Using Indirect Objects:

The methods can be invoked by using indirect objects as given below set_species $Dougal, ‘Dog’;

$Dougal_is = get _species $Dougal;

Attributes

Subroutine declarations and definitions may optionally have attribute lists associated with them. An attribute is a piece of data belonging to a particular object. Unlike most object-

oriented languages, Perl provides no special syntax or support for declaring and manipulating attributes. Attributes are often stored in the object itself. For example, if the object is an anonymous hash, we can store the attribute values in the hash using the attribute name as the key.

```
E.g: sub species
{
 my  $self =shift;
my $was = $self->{'species'};
- - - - - - - - - -
- - - - - - - -
}
```

Class Methods And Attributes

There are operations that are relevant to the class and not need to operate on a specific instance are called class methods or static methods. Similarly attributes that are common to all instances of a class are called as class attributes. Class attributes are just package global variables and class methods are just subroutines that do not require an object reference as the first argument e.g the new constructor.

Inheritance

Perl only provides method inheritance. Inheritance is realized by including a special array @ISA in the package that defines the derived class. For single inheritance @ISA is an array of one element, the name of the base class. Multiple inheritance can be realized by making @Isa an array of more than one element. Each element in the array @ISA is the name of the another package that is being used as a class. If a method cannot be found, the packages referenced in @Isa are recursively searched, depth first. The current class is derived class and those referenced in @ISA are the base classes.

```
e.g : package Employee; use Person;
use strict;
our @ISA = qw(Person);        # inherits from Person
```

Interfacing to the OS:

Creating Internet Ware Applications

The internet is a rich source of information, held on web servers, FTP servers, POP/IMAP

mail servers, news servers etc. A web browser can access information on web servers and FTP servers, and clients access mail and news servers. however, this is not the way of to the information: an 'internet-aware' application can access a server and collect the information without manual intervention. For suppose that a website offers 'lookup' facility in which the user a query by filling in a then clicks the 'submit' button . the data from the form in sent to a CGI program on the server(probably written in which retrieves the information, formats it as a webpage, and returns the page to the browser. A perl application can establish a connection to the server, send the request in the format that the browser would use, collect the returned HTML and then extract the fields that form the answer to the query. In the same way, a perl application can establish a connection to a POP3 mail server and send a request which will result in the server returning a message listing the number of currently unread messages.

Much of the power of scripting languages comes from the way in which they hide the complexity of operations, and this is particularly the case when we make use of specialized modules: tasks that might pages of code in C are achieved in few lines. The LWP (library for WWW access in perl) collection of modules is a very good case in point it makes the kind of interaction described above almost trivial. The LWP:: simple module is a interface to web servers. It can be achieved by exploiting modules, LWP::simple we can retrieve the contents of a web page in a statement:

use LWP::simple $url=...http://www.somesite.com/index.html..;

$page=get($url);

Dirty Hands Internet Programming:

Modules like LWP: : Simple and LWP: :User Agent meet the needs of most programmers requiring web access, and there are numerous other modules for other types of Internet access.

EX:- Net: : FTP for access to FTP servers

Some tasks may require a lower level of access to the network, and this is provided by Perl both in the form of modules(e.g IO: : Socket) and at an even lower level by built-in functions. Support for network programming in perl is so complete that you can use the language to write any conceivable internet application Access to the internet at this level involves the use of sockets, and we explain what a socket is before getting down to details of the programming. Sockets are network communication channels, providing a bi-

directional channel between processes on different machines. Sockets were originally a feature of UNIX other UNIX systems adopted them and the socket became the de facto mechanism of network communication in the UNIX world.

The popular Winsock provided similar functionality for Windows, allowing Windows systems to communicate over the network with UNIX systems, and sockets are a built-in feature of Windows 9X and WindowsNT4.From the Perl programmer's point a network socket can be treated like an open file it is identified by a you write to it with print, and read it from operator. The socket interface is based on the TCP/IP protocol suite, so that all information is handled automatically. In TCP a reliable channel, with automatic recovery from data loss or corruption: for this reason a TCP connection is often described as a virtual circuit. The socket in Perl is an exact mirror of the UNIX and also permits connections using UDP(Unreliable Datagram Protocol).

UNIT-3

PHP Features

Every user has specific reasons for using PHP to implement a mission-critical application, although one could argue that such motives tend to fall into four key categories: practicality, power, possibility, and price.

Practicality

From the very start, the PHP language was created with practicality in mind. After all, Lerdorf's original intention was not to design an entirely new language, but to resolve a problem that had no readily available solution. Furthermore, much of PHP's early evolution was not the result of the explicit intention to improve the language itself, but rather to increase its utility to the user. The result is a language that allows the user to build powerful applications even with a minimum of knowledge.

PHP is a loosely typed language, meaning there is no need to explicitly create, typecast, or destroy a variable, although you are not prevented from doing so. PHP handles such matters internally, creating variables on the fly as they are called in a script, and employing a best-guess formula for automatically typecasting variables. For instance, PHP considers the following set of statements to be perfectly valid:

```
<?php
$number = "5"; // $number is a string
$sum = 15 + $number; // Add an integer and string to produce integer
$sum = "twenty"; // Overwrite $sum with a string.
?>
```

PHP will also automatically destroy variables and return resources to the system when the script completes.

Power

PHP developers have almost 200 native libraries containing well over 1,000 functions, in addition to thousands of third-party extensions.

Although you're likely aware of PHP's ability to interface with databases, manipulate form information, and create pages dynamically, you might not know that PHP can also do the following: Create and manipulate Adobe Flash and Portable Document Format (PDF) files.

Evaluate a password for guess ability by comparing it to language dictionaries and easily broken

patterns. Parse even the most complex of strings using the POSIX and Perl-based regular expression libraries. Authenticate users against login credentials stored in flat files, databases, and even Microsoft's Active Directory. Communicate with a wide variety of protocols, including LDAP, IMAP, POP3, NNTP, and DNS, among others. Tightly integrate with a wide array of credit-card processing solutions.

Possibility

PHP developers are rarely bound to any single implementation solution. On the contrary, a user is typically fraught with choices offered by the language. For example, consider PHP's array of database support options. Native support is offered for more than 25 database products, including Adabas D,dBase, Empress, FilePro, FrontBase, Hyperwave, IBM DB2, Informix, Ingres, InterBase, mSQL, Microsoft SQL Server, MySQL, Oracle, Ovrimos, PostgreSQL, Solid, Sybase, Unix dbm, and Velocis.PHP's flexible string-parsing capabilities offer users of differing skill sets the opportunity to not only immediately begin performing complex string operations but also to quickly port programs of similar functionality (such as Perl and Python) over to PHP.

Price

PHP is available free of charge! Since its inception, PHP has been without usage, modification, and redistribution restrictions. In recent years, software meeting such open licensing qualifications has been referred to as open source software. Open source software and the Internet go together like bread and butter. Open source projects such as Send mail, Bind, Linux, and Apache all play enormous roles in the ongoing operations of the Internet at large. Although open source software's free availability has been the point most promoted by the media, several other characteristics are equally important: Free of licensing restrictions imposed by most commercial products:

Open source software users are freed of the vast majority of licensing restrictions one would expect of commercial counterparts. Although some discrepancies do exist among license variants, users are largely free to modify, redistribute, and integrate the software into other products. Open development and auditing process: Although not without incidents, open source software has long enjoyed a stellar security record. Such high-quality standards are a result of the open development and auditing process. Because the source code is freely available for anyone to examine, security holes and potential problems are rapidly found and fixed.

This advantage was perhaps best summarized by open source advocate Eric S. Raymond, who wrote "Given enough eyeballs, all bugs are shallow."Participation is encouraged: Development

teams are not limited to a particular organization. Anyone who has the interest and the ability is free to join the project. The absence of member restrictions greatly enhances the talent pool for a given project, ultimately contributing to a higher-quality product.

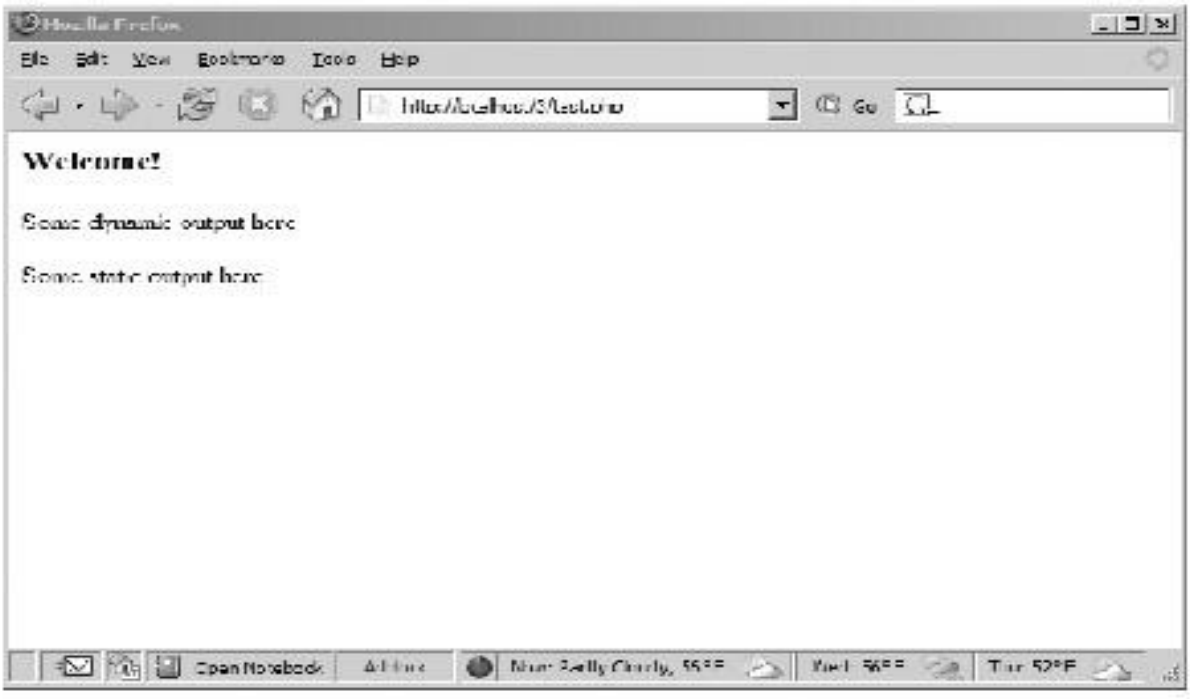
Embedding PHP Code in Your Web Pages

One of PHP's advantages is that you can embed PHP code directly alongside HTML. The engine needs some means to immediately determine which areas of the page are PHP-enabled. This is logically accomplished by delimiting the PHP code. There are four delimitation variants.

Default Syntax

The default delimiter syntax opens with <?php and concludes with ?>, like this:

```
<h3>Welcome!</h3>
<?php
echo "<p>Some dynamic output here</p>";
?>
<p>Some static output here</p>
```



Expected output:

Short-Tags

For less motivated typists, an even shorter delimiter syntax is available. Known as short-tags, this syntax forgoes the php reference required in the default syntax. However, to use this

feature, you need to enable PHP's short_open_tag directive. An example follows

<?

print "This is another PHP example.";

?

>

When short-tags syntax is enabled and you want to quickly escape to and from PHP to output a bitof dynamic text, you can omit these statements using an output variation known as short-circuit syntax:

<?="This is another PHP example.";?>

This is functionally equivalent to both of the following variations:

<? echo "This is another PHP example."; ?>

<?php echo "This is another PHP example.";?>

Script

Certain editors have historically had problems dealing with PHP's more commonly used escape syntax variants. Therefore, support for another mainstream delimiter variant, <script>, is offered:

<script language="php">
print "This is another PHP example.";
</script>

ASP Style

Microsoft ASP pages employ a delimiting strategy similar to that used by PHP, delimiting static from dynamic syntax by using a predefined character pattern: opening dynamic syntax with <%, and concluding with %>. If you're coming from an ASP background and prefer to continue using this escape syntax, PHP supports it. Here's an example:

<%

print "This is another PHP example.";

%>

Embedding Multiple Code Blocks

You can escape to and from PHP as many times as required within a given page. For instance, the following example is perfectly acceptable:

<html>

```
<head>
<title><?php echo "Welcome to my web site!";?></title>
</head>
<body>
<?php
$date = "July 26, 2010";
?>
<p>Today's date is <?=$date;?></p>
</body>
</html>
```

Outputting Data to the Browser

The print() Statement

The print() statement outputs data passed to it . Its prototype looks like this:

```
int print(argument)
```

All of the following are plausible print() statements:

```
<?php
print("<p>I love the summertime.</p>");
?>
<?php
$season = "summertime";
print "<p>I love the $season.</p>";
?>
<?php
print "<p>I love the summertime.</p>";
?>
```

All these statements produce identical output: I love the summertime.

The echo() Statement

Alternatively, you could use the echo() statement for the same purposes as print().echo()'s prototype looks like this:

void echo(string argument1 [, ...string argumentN])

To use echo(), just provide it with an argument just as was done with print(): echo "I love the summertime.";

As you can see from the prototype, echo() is capable of outputting multiple strings. Here's an example:

```
<?php
$heavyweight = "Lennox Lewis";
$lightweight = "Floyd Mayweather";
echo $heavyweight, " and ", $lightweight, " are great fighters.";
?>
```

This code produces the following

Lennox Lewis and Floyd May weather are great fighters.

The printf() Statement

The printf() statement is ideal when you want to output a blend of static text and dynamic information stored within one or several variables. It's ideal for two reasons. First, it neatly separates the static and dynamic data into two distinct sections, allowing for easy maintenance.Second, printf() allows you to wield considerable control over how the dynamic information is rendered to the screen in terms of its type, precision, alignment, and position.

Its prototype looks like this:

integer printf(string format [, mixed args])

For example, suppose you wanted to insert a single dynamic integer value into an otherwise static string:

printf("Bar inventory: %d bottles of tonic water.", 100);

Executing this command produces the following: Bar inventory: 100 bottles of tonic water.

Commonly Used Type Specifiers

%b Argument considered an integer; presented as a binary number

%c Argument considered an integer; presented as a character corresponding to that ASCII value

%d Argument considered an integer; presented as a signed decimal number

%f Argument considered a floating-point number; presented as a floating-point number

%o Argument considered an integer; presented as an octal number

%s Argument considered a string; presented as a string

%u Argument considered an integer; presented as an unsigned decimal number

%x Argument considered an integer; presented as a lowercase hexadecimal number

%X Argument considered an integer; presented as an uppercase hexadecimal number

The sprintf() Statement

The sprintf() statement is functionally identical to printf() except that the output is assigned to a string rather than rendered to the browser. The prototype follows:

string sprintf(string format [, mixed arguments])

An example follows:

$cost = sprintf("$%.2f", 43.2); // $cost = $43.20

PHP's Supported Data Types

A datatype is the generic name assigned to any data sharing a common set of characteristics. Common data types include Boolean, integer, float, string, and array.

Scalar Data Types

Scalar data types are used to represent a single value. Several data types fall under this category, including Boolean, integer, float, and string.

Boolean

The Boolean datatype is named after George Boole (1815–1864), a mathematician who is considered to be one of the founding fathers of information theory. The Boolean data type represents truth, supporting only two values: TRUE and FALSE (case insensitive). Alternatively, you can use zero to represent FALSE, and any nonzero value to represent TRUE.

A few examples follow:

$alive = false; // $alive is false.

$alive = 1; // $alive is true.

$alive = -1; // $alive is true.

$alive = 5; // $alive is true.

$alive = 0; // $alive is false.

Integer

An integer is representative of any whole number or, in other words, a number that does not contain fractional parts. PHP supports integer values represented in base 10 (decimal), base 8

(octal), and base 16 (hexadecimal) numbering systems.

Several examples follow

42 // decimal

-678900 // decimal

0755 // octal

0xC4E // hexadecimal

The maximum supported integer size is platform-dependent, although this is typically positive or negative 2^31 for PHP version 5 and earlier. PHP 6 introduced a 64-bit integer value, meaning PHP will support integer values up to positive or negative 2^63 in size.

Float

Floating-point numbers, also referred to as floats, doubles, or real numbers, allow you to specify numbers that contain fractional parts. Floats are used to represent monetary values, weights, distances, and a whole host of other representations in which a simple integer value won't suffice. PHP's floats can be specified in a variety of ways, several of which are demonstrated here:

4.5678

4.0

8.7e4

1.23E+11

String

Simply put, a string is a sequence of characters treated as a contiguous group. Strings are delimited by single or double quotes. The following are all examples of valid strings: "PHP is a great language"

"whoop-de-do" '*9subway\n' "123$%^789"

For example, consider the following string:

$color = "maroon";

You could retrieve a particular character of the string by treating the string as an array, like this:

```
$parser = $color[2]; // Assigns 'r' to $parser
```

Compound Data Types

Compound data types allow for multiple items of the same type to be aggregated under a single representative entity. The array and the object fall into this category.

Array

It's often useful to aggregate a series of similar items together, arranging and referencing them in some specific way. This data structure, known as an array, is formally defined as an indexed collection of data values. Each member of the array index (also known as the key) references a corresponding value and can be a simple numerical reference to the value's position in the series, or it could have some direct correlation to the value.

For example, if you were interested in creating a list of U.S. states, you could use a numerically indexed array, like so:

```
$state[0] = "Alabama";
$state[1] = "Alaska";
$state[2] = "Arizona";
...
$state[49] = "Wyoming";
```

But what if the project required correlating U.S. states to their capitals? Rather than base the keys on a numerical index, you might instead use an associative index, like this:

```
$state["Alabama"] = "Montgomery";
$state["Alaska"] = "Juneau";
$state["Arizona"] = "Phoenix";...
$state["Wyoming"] = "Cheyenne";
```

Object

The other compound data type supported by PHP is the object. The object is a central concept of the object-oriented programming paradigm. Unlike the other data types contained in the PHP language, an object must be explicitly declared. This declaration of an object's characteristics and behavior takes place within something called a class.Here's a general example of a class definition and subsequent invocation:

```
class Appliance {
 private $_power;
 function setPower($status) {
 $this->_power = $status;
 }
 }
 ...
 $blender = new Appliance;
```

A class definition creates several attributes and functions pertinent to a data structure, in this case a data structure named Appliance. There is only one attribute, power, which can be modified by using the method setPower().

Remember, however, that a class definition is a template and cannot itself be manipulated. Instead, objects are created based on this template. This is accomplished via the new keyword. Therefore, in the last line of the previous listing, an object of class Appliance named blender is created.

The blender object's power attribute can then be set by making use of the method setPower():

$blender->setPower("on");

Converting Between Data Types Using Type Casting

Converting values from one data type to another is known as type casting. A variable can be evaluated once as a different type by casting it to another. This is accomplished by placing the intended type in front of the variable to be cast.

Cast Operators	Conve
(array)	Array
(bool) or (boolean)	Boole
(int) or (integer)	Intege
(object)	Object
(real) or (double) or (float)	Float
(string)	String

Let's consider several examples. Suppose you'd like to cast an integer as a double:

$score = (double) 13; // $score = 13.0

Type casting a double to an integer will result in the integer value being rounded down, regardless of the decimal value. Here's an example:

$score = (int) 14.8; // $score = 14

What happens if you cast a string datatype to that of an integer? Let's find out:

$sentence = "This is a sentence"; echo (int) $sentence; // returns 0

While likely not the expected outcome, it's doubtful you'll want to cast a string like this anyway. You can also cast a datatype to be a member of an array. The value being cast simply becomes the first element of the array:

$score = 1114;
$scoreboard = (array) $score;

```
echo $scoreboard[0]; // Outputs 1114
```

One final example: any data type can be cast as an object. The result is that the variable becomes an attribute of the object, the attribute having the name scalar:

```
$model = "Toyota";
$obj = (object) $model;
```

The value can then be referenced as follows: print $obj->scalar; // returns "Toyota"

Adapting Data Types with Type Juggling

Because of PHP's lax attitude toward type definitions, variables are sometimes automatically cast to best fit the circumstances in which they are referenced. Consider the following snippet:

```
<?php
$total = 5; // an integer
$count = "15"; // a string
$total += $count; // $total = 20 (an integer)
?>
```

The outcome is the expected one; $total is assigned 20, converting the $count variable from a string to an integer in the process. Here's another example demonstrating PHP's type-juggling capabilities:

```
<?php
$total = "45 fire engines";
$incoming = 10;
$total = $incoming + $total; // $total = 55
?>
```

The integer value at the beginning of the original $total string is used in the calculation. However, if it begins with anything other than a numerical representation, the value is 0.

Consider another example:

```
<?php
$total = "1.0";
if ($total) echo "We're in positive territory!";
?>
```

In this example, a string is converted to Boolean type in order to evaluate the if statement.

Consider one last particularly interesting example. If a string used in a mathematical calculation includes ., e, or E (representing scientific notation), it will be evaluated as a float:

```
<?php
$val1 = "1.2e3"; // 1,200
$val2 = 2;
echo $val1 * $val2; // outputs 2400
?>
```

Type-Related Functions

A few functions are available for both verifying and converting data types.

Retrieving Types

The gettype() function returns the type of the provided variable. In total, eight possible return values are available: array, boolean, double, integer, object, resource, string, and unknown type. Its prototype follows:string gettype(mixed var)

Converting Types

The settype() function converts a variable to the type specified by type. Seven possible type values are available: array, boolean, float, integer, null, object, and string. If the conversion is successful, TRUE is returned; otherwise, FALSE is returned. Its prototype follows:boolean settype(mixed var, string type

Type Identifier FunctionsA number of functions are available for determining a variable's type, including is_array(), is_bool(),is_float(), is_integer(), is_null(), is_numeric(), is_object(), is_resource(), is_scalar(), and is_string(). Because all of these functions follow the same naming convention, arguments, and return values, their introduction is consolidated into a single example.The generalized prototype follows:

```
boolean is_name(mixed var)
```

All of these functions are grouped in this section because each ultimately accomplishes the same task. Each determines whether a variable, specified by var, satisfies a particular condition specified by the function name. If var is indeed of the type tested by the function name, TRUE is returned; otherwise, FALSE is returned.

An example follows:

```
<?php
$item = 43;
```

printf("The variable \$item is of type array: %d
", is_array($item)); printf("The variable \$item is of type integer: %d
", is_integer($item)); printf("The variable \$item is numeric: %d
", is_numeric($item));?>

This code returns the following:

The variable $item is of type array: 0 The variable $item is of type integer: 1 The variable $item is numeric: 1

Identifiers

Identifier is a general term applied to variables, functions, and various other user-defined objects. There are several properties that PHP identifiers must abide by:

An identifier can consist of one or more characters and must begin with a letter or an underscore. Furthermore, identifiers can consist of only letters, numbers, underscore characters, and other ASCII characters from 127 through 255.

Valid	Invalid
my_function	This&that
Size	!counter
_someword	4ward

- Identifiers are case sensitive. Therefore, a variable named $recipe is different from a variable named $Recipe, $rEciPe, or $recipE.
- Identifiers can be any length. This is advantageous because it enables a programmer to accurately describe the identifier's purpose via the identifier name.
- An identifier name can't be identical to any of PHP's predefined keywords.

Variables

A variable is a symbol that can store different values at different times. For example, suppose you create a web-based calculator capable of performing mathematical tasks.

Variable Declaration

A variable always begins with a dollar sign, $, which is then followed by the variable name.

Variable names follow the same naming rules as identifiers.

That is, a variable name can begin with either a letter or an underscore and can consist of letters, underscores, numbers, or other ASCII characters rangingfrom 127 through 255. The following

are all valid variables:

$color

$operating_system

$_some_variable

$model

Note that variables are case sensitive. For instance, the following variables bear no relation to one another:

$color

$Color

$COLOR

Value Assignment

Assignment by value simply involves copying the value of the assigned expression to the variable assignee. This is the most common type of assignment. A few examples follow:

$color = "red";

$number = 12;

$age = 12;

$sum = 12 + "15"; // $sum = 27.

Reference Assignment

PHP 4 introduced the ability to assign variables by reference, which essentially means that you can create a variable that refers to the same content as another variable does. Therefore, a change to any variable referencing a particular item of variable content will be reflected among all other variables referencing that same content. You can assign variables by reference by appending an ampersand (&) to the equal sign. Let's consider an example:

```
<?php
$value1 = "Hello";
$value2 =& $value1; // $value1 and $value2 both equal "Hello"
$value2 = "Goodbye"; // $value1 and $value2 both equal "Goodbye"
?>
```

An alternative reference-assignment syntax is also supported, which involves appending the ampersand to the front of the variable being referenced. The following example adheres to this new syntax

```
<?php
$value1 = "Hello";
$value2 = &$value1; // $value1 and $value2 both equal "Hello"
$value2 = "Goodbye"; // $value1 and $value2 both equal "Goodbye"
?>
```

Variable Scope

Scope can be defined as the range of availability a variable has to the program in which it is declared. PHP variables can be one of four scope types Local variables Function parameters Global variables Static variables

Local Variables

A variable declared in a function is considered local; that is, it can be referenced solely in that function. Any assignment outside of that function will be considered to be an entirely different variable from the one contained in the function

```
<?php
$x = 4;
function assignx ()
{
$x = 0;
print "\$x inside function is $x. <br />";
}
assignx();
print "\$x outside of function is $x. <br />";
?>
```

This will produce the following result

$x inside function is 0.

$x outside of function is 4.

Function Parameters

Function parameters are declared after the function name and inside parentheses. They are declared much like a typical variable would be

```
<?php// multiply a value by 10 and return it to the caller function multiply ($value)
{
$value = $value * 10; return $value;
}
$retval = multiply (10);
Print "Return value is $retval\n";
?>
```

This will produce the following result −

```
Return value is 100
```

Global Variables

In contrast to local variables, a global variable can be accessed in any part of the program. However, in order to be modified, a global variable must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword GLOBAL in front of the variable that should be recognized as global. Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example

```
<?php
$somevar = 15;
function addit()
{ GLOBAL $somevar;$somevar++;print "Somevar is $somevar";}
addit();
?>
```

This will produce the following result −

```
Somevar is 16
```

Static Variables

The final type of variable scoping that I discuss is known as static. In contrast to the variables declared as function parameters, which are destroyed on the function's exit, a static variable will not lose its value when the function exits and will still hold that value should the function be called again.

You can declare a variable to be static simply by placing the keyword STATIC in front of the variable name.

```
<?php
function keep_track() { STATIC $count = 0;
$count++; print $count; print "<br />";
}
keep_track(); keep_track(); keep_track();
?>
```

This will produce the following result – 1

2

3

PHP's Superglobal Variables

PHP offers a number of useful predefined variables that are accessible from anywhere within the executing script and provide you with a substantial amount of environment-specific information. You can sift through these variables to retrieve details about the current user session, the user's operating environment, the local operating environment, and more. PHP creates some of the variables, while the availability and value of many of the other variables are specific to the operating system and web server.

Therefore, rather than attempt to assemble a comprehensive list of all possible predefined variables and their possible values, the following code will output all predefined variables pertinent to any given web server and the script's execution environment:

```
foreach ($_SERVER as $var => $value)
{
echo "$var => $value <br />";
}
```

S.No	Variable & Description
1	$_GLOBALS: Contains a reference to every variable which is currently available within the global scope of the script. The keys of this array are the names of the global variables.

2	$_SERVER: This is an array containing information such as headers, paths, and script locations. The entries in this array are created by the web server. There is no guarantee that every web server will provide any of these. See next section for a complete list of all the SERVER variables.
3	$_GET: An associative array of variables passed to the current script via the HTTP GET method.
4	$_POST: An associative array of variables passed to the current script via the HTTP POST method.
5	$_FILES: An associative array of items uploaded to the current script via the HTTP POST method.
6	$_REQUEST: An associative array consisting of the contents of $_GET, $_POST, and $_COOKIE.
7	$_COOKIE: An associative array of variables passed to the current script via HTTP cookies.
8	$_SESSION: An associative array containing session variables available to the current script.
9	$_PHP_SELF: A string containing PHP script file name in which it is called.
10	$php_errormsg: $php_errormsg is a variable containing the text of the last error message generated by PHP.

Variable Variables

On occasion, you may want to use a variable whose content can be treated dynamically as a variable in itself. Consider this typical variable assignment

$recipe = "spaghetti";

Interestingly, you can treat the value spaghetti as a variable by placing a second dollar sign in front of the original variable name and again assigning another value

$$recipe = "& meatballs";

This in effect assigns & meatballs to a variable named spaghetti. Therefore, the following two snippets of code produce the same result

echo $recipe $spaghetti; echo $recipe ${$recipe};

The result of both is the string spaghetti & meatballs.

Constants

A constant is a value that cannot be modified throughout the execution of a program. Constants are particularly useful when working with values that definitely will not require modification, such as Pi (3.141592) or the number of feet in a mile (5,280). Once a constant has been defined, it cannot be changed (or redefined) at any other point of the program. Constants are defined using the define() function.

Defining a Constant

The define() function defines a constant by assigning a value to a name. Its prototype follows:

boolean define(string name, mixed value [, bool case_insensitive])If the optional parameter case_insensitive is included and assigned TRUE, subsequent references to the constant will be case insensitive. Consider the following example in which the mathematical constant Pi is defined:define("PI", 3.141592);The constant is subsequently used in the following listing: printf("The value of Pi is %f", PI);

$pi2 = 2 * PI;

printf("Pi doubled equals %f", $pi2);

This code produces the following results:

The value of pi is 3.141592. Pi doubled equals 6.283184.

Expressions

An expression is a phrase representing a particular action in a program. All expressions consist of at least one operand and one or more operators.

A few examples follow:

$a = 5; // assign integer value 5 to the variable $a

$a = "5"; // assign string value "5" to the variable $a

$sum = 50 + $some_int; // assign sum of 50 + $some_int to $sum

$wine = "Zinfandel"; // assign "Zinfandel" to the variable $wine

$inventory++; // increment the variable $inventory by 1

Operands

Operands are the inputs of an expression. You might already be familiar with the manipulation and use of operands not only through everyday mathematical calculations, but also through prior programming experience. Some examples of operands follow:

$a++; // $a is the operand

$sum = $val1 + val2; // $sum, $val1 and $val2 are operands

Operators

An operator is a symbol that specifies a particular action in an expression. Many operators may be familiar to you. Regardless, you should remember that PHP's automatic type conversion will convert types based on the type of operator placed between the two operands, which is not always the case in other programming languages. The precedence and associativity of operators are significant characteristics of a programming language. Operator Precedence, Associativity, and Purpose.

Operator	Associativity	Purpose
new	NA	Object instantiation
( )	NA	Expression sub grouping
[ ]	Right	Index enclosure
! ~ ++ --	Right	Boolean NOT, bitwise NOT, increment,
@	Right	Error suppression
/ * %	Left	Division, multiplication, modulus
+ - .	Left	Addition, subtraction, concatenation
<< >>	Left	Shift left, shift right (bitwise)
< <= > >=	NA	Less than, less than or equal to, greater than,
== != === <>	NA	Is equal to, is not equal to, is identical to, is
& ^ |	Left	Bitwise AND, bitwise XOR, bitwise OR
&& ||	Left	Boolean AND, Boolean OR
?:	Right	Ternary operator

= += *= /= .= %=&=	Right	Assignment operators
AND XOR OR	Left	Boolean AND, Boolean XOR, Boolean OR
,	Left	Expression separation

Operator Precedence

Operator precedence is a characteristic of operators that determines the order in which they evaluate the operands surrounding them. PHP follows the standard precedence rules used in elementary school math class. Consider a few examples:

$total_cost = $cost + $cost * 0.06;

This is the same as writing

$total_cost = $cost + ($cost * 0.06);

Because the multiplication operator has higher precedence than the addition operator.

Operator Associativity

The associativity characteristic of an operator specifies how operations of the same precedence are evaluated as they are executed. Associativity can be performed in two directions, left-to-right or right-to-left. Left-to-right associativity means that the various operations making up the expression are evaluated from left to right.

Consider the following example:

$value = 3 * 4 * 5 * 7 * 2;

The preceding example is the same as the following:

$value = ((((3 * 4) * 5) * 7) * 2);

Arithmetic Operators

There are following arithmetic operators supported by PHP language − Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by de-	B / A will give 2
%	Modulus Operator and remainder of after an	B % A will give 0

++	Increment operator, increases integer value by	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Assignment Operators

There are following assignment operators supported by PHP language

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A

%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

String Operators

PHP's string operators provide a convenient way in which to concatenate strings together. There are two such operators, including the concatenation operator (.) and the concatenation assignment operator (.=).

Example	Label	Outcome
$a = "abc"."def";	Concatenation	$a is assigned the string "abcdef"
$a .= "ghijkl";	Concatenation-assignment	$a equals its current value concatenated with "ghijkl"

Increment and Decrement Operators

The increment (++) and decrement (--) operators present a minor convenience in terms of code clarity, providing shortened means by which you can add 1 to or subtract 1 from the current value of a variable.

Example	Label	Outcome
++$a, $a++	Increment	Increment $a by 1
--$a, $a--	Decrement	Decrement $a by 1

Logical Operators

There are following logical operators supported by PHP language Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then condition becomes true.	(A and B) is true.

or	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(A or B) is true.
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator	!(A && B) is false.

Equality Operators

Equality operators are used to compare two values, testing for equivalence.

Example	Label	Outcome
$a == $b	Is equal to	True if $a and $b are equivalent
$a != $b	Is not equal to	True if $a is not equal to $b
$a ===	Is identical to	True if $a and $b are equivalent

Comparison Operators

There are following comparison operators supported by PHP language Assume variable A holds 10 and variable B holds 20 then −

Operatorr	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then	(A != B) is true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Bitwise Operators

Bitwise operators examine and manipulate integer values on the level of individual bits that make up the integer value (thus the name).

Example	Label	Outcome
$a & $b	AND	And together each bit contained in $a and $b
$a | $b	OR	Or together each bit contained in $a and $b
$a ^ $b	XOR	Exclusive—or together each bit contained in $a and $b
~ $b	NOT	Negate each bit in $b
$a << $b	Shift left $a	will receive the value of $b shifted left two bits
$a >> $b	Shift right $a	will receive the value of $b shifted right two bits

String Interpolation

To offer developers the maximum flexibility when working with string values, PHP offers a means for both literal and figurative interpretation. For example, consider the following string

The $animal jumped over the wall.\n

You might assume that $animal is a variable and that \n is a newline character, and therefore both should be interpreted accordingly. However, what if you want to output the string exactly as it is written, or perhaps you want the newline to be rendered but want the variable to display in its literal form ($animal), or vice versa? All of these variations are possible in PHP, depending on how the strings are enclosed and whether certain key characters are escaped through a predefined sequence.

Double Quotes Strings enclosed in double quotes are the most commonly used in PHP scripts because they offer the most flexibility. This is because both variables and escape sequences will be parsed accordingly. Consider the following example

```
<?php
$sport = "boxing";
echo "Jason's favorite sport is $sport.";
?>
```

This example returns the following: Jason's favorite sport is boxing.

Escape Sequences

Escape sequences are also parsed. Consider this example:

<?php

$output = "This is one line.\nAnd this is another line."; echo $output;

?>

This returns the following (as viewed from within the browser source): This is one line.

And this is another line.

Single Quotes

Enclosing a string within single quotes is useful when the string should be interpreted exactly as stated. This means that both variables and escape sequences will not be interpreted when the string is parsed.

For example, consider the following single-quoted string: print 'This string will $print exactly as it\'s \n declared.';

This produces the following:

This string will $print exactly as it's \n declared.

Curly Braces

While PHP is perfectly capable of interpolating variables representing scalar data types, you'll

find that variables representing complex data types such as arrays or objects cannot be so easily parsed when embedded in an echo() or print() string. You can solve this issue by delimiting the variable in curly braces, like this:

```
echo "The capital of Ohio is {$capitals['ohio']}.";
```

Heredoc

Heredoc syntax offers a convenient means for outputting large amounts of text. Rather than delimiting strings with double or single quotes, two identical identifiers are employed. An example follows:

```
<?php
$website = "http://www.romatermini.it"; echo <<<EXCERPT
<p>Rome's central train station, known as <a href = "$website">Roma Termini</a>, was built in 1867. Because it had fallen into severe disrepair in the late 20th century, the government knew that considerable resources were required to rehabilitate the station prior to the 50-year <i>Giubileo</i>.</p>
EXCERPT;
?>
```

Several points are worth noting regarding this example:

The opening and closing identifiers (in the case of this example, EXCERPT) must be identical.The opening identifier must be preceded with three left-angle brackets (<<<).Heredoc syntax follows the same parsing rules as strings enclosed in double quotes. That is, both variables and escape sequences are parsed. The only difference is that double quotes do not need to be escaped.The closing identifier must begin at the very beginning of a line. It cannot be preceded with spaces or any other extraneous character.

Nowdoc

Introduced in PHP 5.3, nowdoc syntax operates identically to heredoc syntax, except that none of the text delimited within a nowdoc is parsed. If you would like to display, for instance, a snippet of code in the browser, you could embed it within a nowdoc statement; when subsequently outputting the nowdoc variable, you can be sure that PHP will not attempt to interpolate any of the string as code.

Control Structures

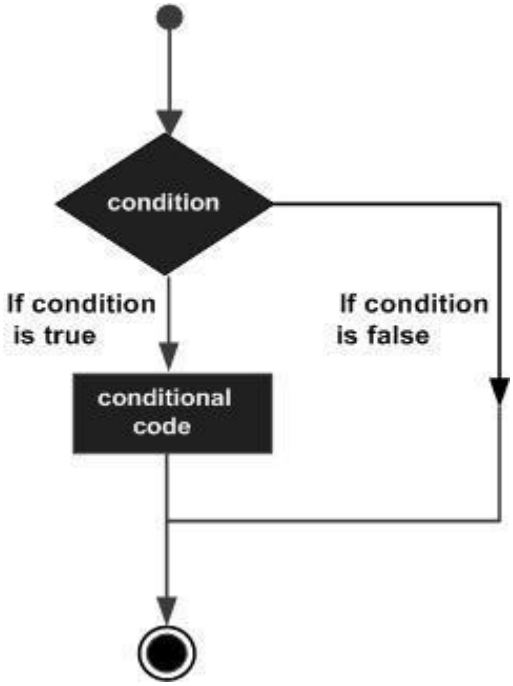
Control structures determine the flow of code within an application, defining execution characteristics such as whether and how many times a particular code statement will execute, as

well as when a code block will relinquish execution control.

Conditional Statements

Conditional statements make it possible for your computer program to respond accordingly to a wide variety of inputs, using logic to discern between various conditions based on input value. You can use conditional statements in your code to make your decisions. PHP supports following three decision making statements

- if...else statement − use this statement if you want to execute a set of code when a condition is true and another if the condition is not true
- elseif statement − is used with the if...else statement to execute a set of code if one of the several condition is true
- switch statement − is used if you want to select one of many blocks of code to be executed, use the Switch statement. The switch statement is used to avoid long blocks of if..elseif..else code.



The If...Else Statement

If you want to execute some code if a condition is true and another code if a condition is false, use the if....else statement.

Syntax

if (condition)

code to be executed if condition is true;

else

code to be executed if condition is false;

Example

The following example will output "Have a nice weekend!" if the current day is Friday, Otherwise, it will output "Have a nice day!":

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

It will produce the following result − Have a nice day!

The ElseIf Statement

If you want to execute some code if one of the several conditions are true use the elseif statement

Syntax

if (condition)

code to be executed if condition is true;

elseif (condition)

code to be executed if condition is true;

else

code to be executed if condition is false;

Example

The following example will output "Have a nice weekend!" if the current day is Friday, and "Have a nice Sunday!" if the current day is Sunday. Otherwise, it will output "Have a nice day!"

```
<html>
<body>
<?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!";
?>
</body>
</html>
```

It will produce the following result − Have a nice day!

The Switch Statement

If you want to select one of many blocks of code to be executed, use the Switch statement. The switch statement is used to avoid long blocks of if..elseif..else code.

Syntax

```
switch (expression)
{
case  label1:
code to be executed if expression = label1;
break;
case label2:
code to be executed if expression = label2;
break;
 default:code to be executed
if expression is different from both label1 and label2;
}
```

Example

The switch statement works in an unusual way. First it evaluates given expression then seeks a label to match the resulting value. If a matching value is found then the code associated with the matching label will be executed or if none of the label matches then statement will execute any specified default code.

```
<html>
<body>
<?php
$d = date("D");
switch ($d)
{
    case "Mon":
    echo "Today is Monday"; break;
    case "Tue":
    echo "Today is Tuesday"; break;
    case "Wed":
    echo "Today is Wednesday"; break;
    case "Thu":
    echo "Today is Thursday";
    break;
    case "Fri":
    echo "Today is Friday"; break;
    case "Sat":
    echo "Today is Saturday"; break;
    case "Sun":
    echo "Today is Sunday"; break;
    default:
    echo "Wonder which day is this ?";
}
?>
```

</body>

</html>

It will produce the following result −

Today is Wednesday

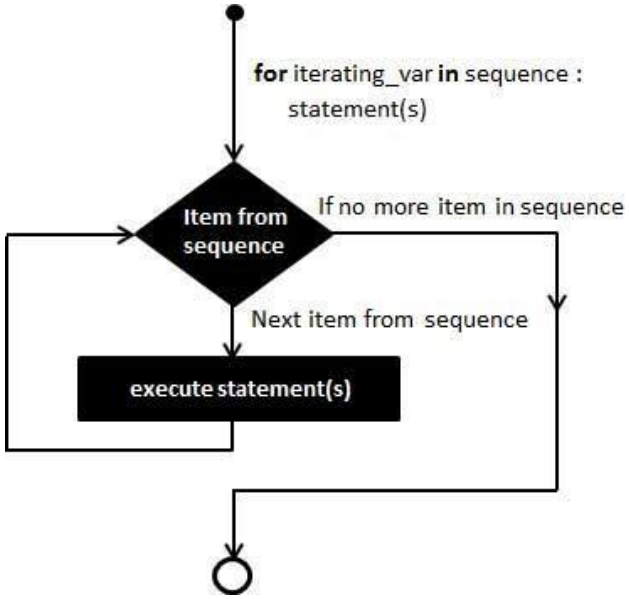
Looping Statements

Loops in PHP are used to execute the same block of code a specified number of times. PHP supports following four loop types.

- for − loops through a block of code a specified number of times.
- while − loops through a block of code if and as long as a specified condition is true. do...while loops through a block of code once, and then repeats the loop as long as a special condition is true.
- foreach − loops through a block of code for each element in an array.

The for loop statement

The for statement is used when you know how many times you want to execute a statement or a block of statements.



Syntax

```
for (initialization; condition; increment)
{
code to be executed;
}
```

The initializer is used to set the start value for the counter of the number of loop iterations. A variable may be declared here for this purpose and it is traditional to name it $i.

Example

The following example makes five iterations and changes the assigned value of two variables on each pass of the loop

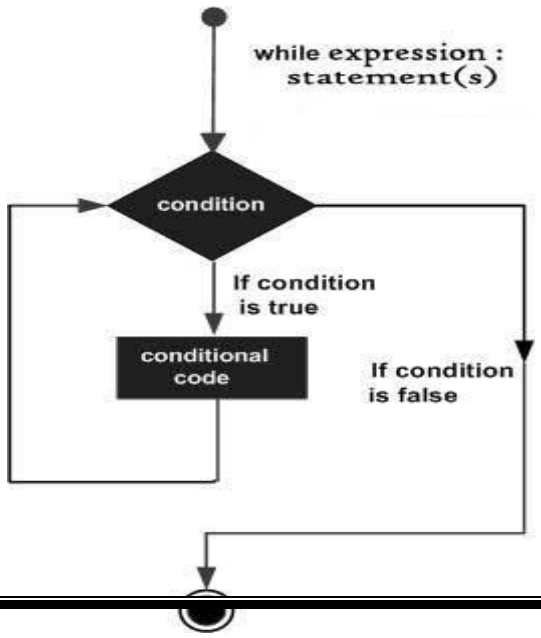
```
<html>
<body>
<?php
$a = 0;
$b = 0;
for( $i = 0; $i<5; $i++ )
{
$a += 10;
$b += 5;
}
echo ("At the end of the loop a = $a and b = $b" );
?>
</body>
</html>
```

This will produce the following result –

At the end of the loop a = 50 and b = 25

The while loop statement

The while statement will execute a block of code if and as long as the test expression is true. If



the test expression is true then the code block will be executed. After the code has executed the test expression will again be evaluated and the loop will continue until the test expression is found to be false.

Syntax

```
while (condition)
{
code to be executed;
}
```

Example

This example decrements a variable value on each iteration of the loop and the counter increments until it reaches 10 when the evaluation is false and the loop ends.

```
<html>
<body>
<?php
$i = 0;
$num = 50;
while( $i < 10)
{
$num--;
$i++;
}
echo ("Loop stopped at i = $i and num = $num" );
?>
</body>
</html>
```

This will produce the following result –

Loop stopped at i = 10 and num = 40

The do...while loop statement

The do...while statement will execute a block of code at least once it then will repeat the loop as long as a condition is true.

Syntax

```
do
{
code to be executed;
}
while (condition);
```

Example

The following example will increment the value of i at least once, and it will continue incrementing the variable i as long as it has a value of less than 10

```
<html>
<body>
<?php
$i = 0;
$num = 0;
do
{
$i++;
}
while( $i < 10 );
echo ("Loop stopped at i = $i" );
?>
</body>
</html>
```

This will produce the following result −

Loop stopped at i = 10

The foreach loop statement

The foreach statement is used to loop through arrays. For each pass the value of the current array element is assigned to $value and the array pointer is moved by one and in the next pass next element will be processed.

Syntax

```
foreach (array as value)
{
code to be executed;
}
```

Example

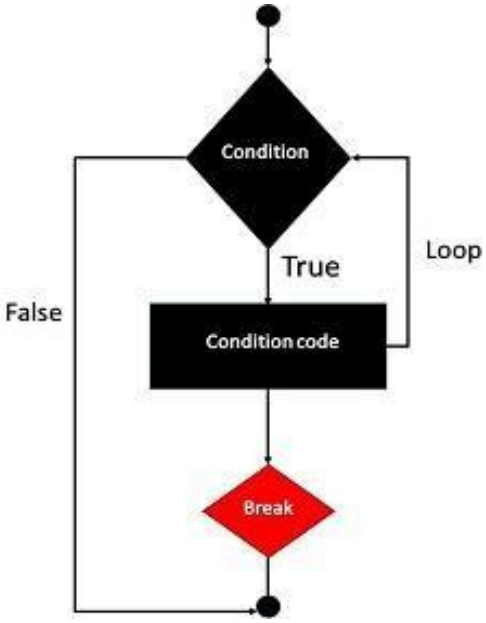
Try out following example to list out the values of an array.

```
<html>
<body>
<?php
$array = array( 1, 2, 3, 4, 5);
foreach( $array as $value )
 {
 echo "Value is $value <br />";
 }
 ?>
 </body>
 </html>
```

This will produce the following result –

```
Value is 1
Value is 2
Value is 3
Value is 4
Value is 5
```

BREAK statement : The PHP break keyword is used to terminate the execution of a loop prematurely. The break statement is situated inside the statement block. If gives you full control and whenever you want to exit from the loop you can come out. After coming out of a loop immediate statement to the loop will be executed.



Example

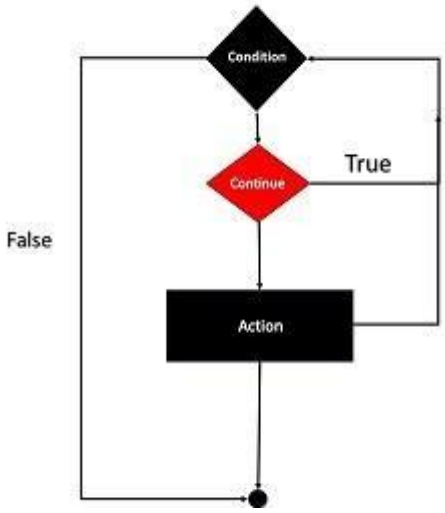
In the following example condition test becomes true when the counter value reaches 3 and loop terminates.

```
<html>
<body>
<?php
$i = 0;
while( $i < 10)
{
$i++;
if( $i == 3 )break;
}
echo ("Loop stopped at i = $i" );
?>
</body>
</html>
```

This will produce the following result –

Loop stopped at i = 3

Continue statement The PHP continue keyword is used to halt the current iteration of a loop but it does not terminate the loop. Just like the break statement the continue statement is situated inside the statement block containing the code that the loop executes, preceded by a conditional test. For the pass encountering continue statement, rest of the loop code is skipped and next pass starts.



Example

In the following example loop prints the value of array but for which condition becomes true it just skip the code and next value is printed.

```
<html>
<body>
<?php
$array = array( 1, 2, 3, 4, 5);
foreach( $array as $value )
{
 if( $value == 3 )continue;
echo "Value is $value <br />";
}
?>
</body>
</html>
```

This will produce the following result –

Value is 1

Value is 2

Value is 4

Value is 5

File Inclusion statements

You can include the content of a PHP file into another PHP file before the server executes it. There are two PHP functions which can be used to included one PHP file into another PHP file. The include() Function The require() Function This is a strong point of PHP which helps in creating functions, headers, footers, or elements that can be reused on multiple pages. This will help developers to make it easy to change the layout of complete website with minimal effort. If there is any change required then instead of changing thousand of files just change included file. The include() Function The include() function takes all the text in a specified file and copies it into the file that uses the include function. If there is any problem in loading a file then the include() function generates a warning but the script will continue execution. Assume you want to create a common menu for your website. Then create a file menu.php with the following content.

```
<a href="http://www.tutorialspoint.com/index.htm">Home</a> -
<a href="http://www.tutorialspoint.com/ebxml">ebXML</a> -
<a href="http://www.tutorialspoint.com/ajax">AJAX</a> -
<a href="http://www.tutorialspoint.com/perl">PERL</a> <br />
```

Now create as many pages as you like and include this file to create header. For example now your test.php file can have following content.

```
<html>
<body>
<?php include("menu.php"); ?>
<p>This is an example to show how to include PHP file!</p>
</body>
</html>
```

It will produce the following result –

Home - ebXML - AJAX - PERL

Ensuring a File is included only once:

The include_once() function verified whether the file has already been included.

include_once(filename);

If a file has already been included; include_once will not execute. Otherwise, it will include the file as necessary.

The require() Function

The require() function takes all the text in a specified file and copies it into the file that uses the include function. If there is any problem in loading a file then the require() function generates a fatal error and halt the execution of the script. So there is no difference in require() and include() except they handle error conditions. It is recommended to use the require() function instead of include(), because scripts should not continue executing if files are missing or misnamed. You can try using above example with require() function and it will generate same result. But if you will try following two examples where file does not exist then you will get different results.

```
<html>
<body>
<?php include("xxmenu.php"); ?>
<p>This is an example to show how to include wrong PHP file!</p>
</body>
</html>
```

This will produce the following result

This is an example to show how to include wrong PHP file!

```
<html>
<body>
<?php require("xxmenu.php"); ?>
<p>This is an example to show how to include wrong PHP file!</p>
</body>
</html>
```

Ensuring a File Is Required Only Once

As your site grows, you may find yourself redundantly including certain files. Although this might not always be a problem, sometimes you will not want modified variables in the included file to be overwritten by a later inclusion of the same file. Another problem that arises is the

clashing of function names should they exist in the inclusion file. You can solve these problems with the require_once() function. Its prototype follows:

require_once (filename)

The require_once() function ensures that the inclusion file is included only once in your script. After require_once() is encountered, any subsequent attempts to include the same file will be ignored.

Functions

Creating a Function

Although PHP's vast assortment of function libraries is a tremendous benefit to anybody seeking to avoid reinventing the programmatic wheel, sooner or later you'll need to go beyond what is offered in the standard distribution, which means you'll need to create custom functions or even entire function libraries. To do so, you'll need to define a function using PHP's supported syntax, which when written in pseudo code looks like this:

```
function function Name(parameters)
{
function-body
}
```

For example, consider the following function, generateFooter(), which outputs a page footer: function generateFooter()

```
{
echo "Department of Computer Science and Engineering";
}
```

Once defined, you can call this function like so:

```
<?php generateFooter();
?>
```

This yields the following result:

Department of Computer Science and Engineering

Passing Arguments by Value

You'll often find it useful to pass data into a function. As an example, let's create a function that calculates an item's total cost by determining its sales tax and then adding that amount to the price

```
function calcSalesTax($price, $tax)
{
$total = $price + ($price * $tax); echo "Total cost: $total";
}
```

This function accepts two parameters, aptly named $price and $tax, which are used in the calculation. Although these parameters are intended to be floating points, because of PHP's weak typing, nothing prevents you from passing in variables of any datatype, but the outcome might not be what you expect. In addition, you're allowed to define as few or as many parameters as you deem necessary; there are no language-imposed constraints in this regard.Once defined, you can then invoke the function as demonstrated

calcSalesTax() function would be called like so: calcSalesTax(15.00, .075);

Passing Arguments by Reference

On occasion, you may want any changes made to an argument within a function to be reflected outside of the function's scope. Passing the argument by reference accomplishes this. Passing an argument by reference is done by appending an ampersand to the front of the argument.Here's an example:

```
<?php
$cost = 20.99;
$tax = 0.0575;
function calculateCost(&$cost, $tax)
{
   // Modify the $cost variable
   $cost = $cost + ($cost * $tax);
   // Perform some random change to the $tax variable.
   $tax += 4;
}
calculateCost($cost, $tax); printf("Tax is %01.2f%% ", $tax*100); printf("Cost is: $%01.2f", $cost);
?>
```

Here's the result: Tax is 5.75%

Cost is $22.20

Default Argument Values

Default values can be assigned to input arguments, which will be automatically assigned to the argument if no other value is provided. You could then assign $tax the default value of 6.75 percent, like this

```
function calcSalesTax($price, $tax=.0675)
  {
  $total = $price + ($price * $tax); echo "Total cost: $total";
  }
```

You can still pass $tax another taxation rate; 6.75 percent will be used only if calcSalesTax() is invoked without the second parameter like this:

```
$price = 15.47; calcSalesTax($price);
```

Using Type Hinting

PHP 5 introduced a new feature known as type hinting, which gives you the ability to force parameters to be objects of a certain class or to be arrays. Unfortunately, type hinting using scalar data types such as integers and strings is not supported. If the provided parameter is not of the desired type, a fatal error will occur.

Returning Values from a Function

Often, simply relying on a function to do something is insufficient; a script's outcome might depend on a function's outcome or on changes in data resulting from its execution. Yet variable scoping prevents information from easily being passed from a function body back to its caller, so how can we accomplish this? You can pass data back to the caller by way of the return() statement.

The return Statement

The return() statement returns any ensuing value back to the function caller, returning program control back to the caller's scope in the process. If return() is called from within the global scope, the script execution is terminated.

```
function calcSalesTax($price, $tax=.0675)
 {
 $total = $price + ($price * $tax); return $total;
 }
```

Returning Multiple Values

It's often convenient to return multiple values from a function. For example, suppose that you'd like to create a function that retrieves user data from a database (say the user's name, e-mail address, and phone number) and returns it to the caller. Accomplishing this is much easier than you might think, with the help of a very useful language construct, list(). The list() construct offers a convenient means for retrieving values from an array, like so

```
<?php
    function retrieveUserProfile()
    {
    $user[] = "Arifa Tehseen";
    $user[] = "arifa@jbiet.edu.in";
    $user[] = "English"; return $user;
    }
    list($name, $email, $language) = retrieveUserProfile();
    echo "Name: $name, email: $email, language: $language";
?>
```

Executing this script returns the following:

Name: Arifa Tehseen, email:arifa@jbiet.edu.in, language: English

Recursive Functions

Recursive functions, or functions that call themselves, offer considerable practical value to the programmer and are used to divide an otherwise complex problem into a simple case, reiterating that case until the problem is resolved.

Function Libraries

Great programmers are lazy, and lazy programmers think in terms of reusability. Functions offer a great way to reuse code and are often collectively assembled into libraries and subsequently repeatedly reused within similar applications. PHP libraries are created via the simple aggregation of function definitions in a single file, like this:

```
<?php
function localTax($grossIncome, $taxRate)
{
// function body here
```

```
}
function stateTax($grossIncome, $taxRate, $age)
 {
// function body here
}
function medicare($grossIncome, $medicareRate)
 {
// function body here
}
?>
```

ARRAYS

PHP takes this definition a step further, forgoing the requirement that the items share the same data type. For example, an array could quite possibly contain items such as state names, ZIP codes, exam scores, or playing card suits. Each item consists of two components: the aforementioned key and a value. The key serves as the lookup facility for retrieving its counterpart, the value. Keys can be numerical or associative. Numerical keys bear no real relation to the value other than the value's position in the array.

```
$states = array(0 => "Alabama", 1 => "Alaska"...49 => "Wyoming");
$states = array("OH" => "Ohio", "PA" => "Pennsylvania", "NY" => "New York")
```

Creating an Array

Unlike other languages, PHP doesn't require that you assign a size to an array at creation time. In fact, because it's a loosely typed language, PHP doesn't even require that you declare the array before using it.

```
$state[0] = "Delaware";
```

Interestingly, if you intend for the index value to be numerical and ascending, you can omit the index value at creation time:

```
$state[] = "Pennsylvania";
$state[] = "New Jersey";
...
$state[] = "Hawaii";
```

Creating Arrays with array()

The array() construct takes as its input zero or more items and returns an array consisting of these input elements. Its prototype looks like this:

array array([item1 [,item2 ... [,itemN]]])

Extracting Arrays with list()

The list() construct is similar to array(), though it's used to make simultaneous variable assignments from values extracted from an array in just one operation. Its prototype looks like this:

void list(mixed...)This construct can be particularly useful when you're extracting information from a database or file.

Populating Arrays with a Predefined Value Range

The range() function provides an easy way to quickly create and fill an array consisting of a range of low and high integer values. An array containing all integer values in this range is returned. Its prototype looks like this:

array range(int low, int high [, int step])

For example, suppose you need an array consisting of all possible face values of a die:

```
$die = range(1, 6);
// Same as specifying $die = array(1, 2, 3, 4, 5, 6)
```

Testing for an Array

When you incorporate arrays into your application, you'll sometimes need to know whether a particular variable is an array. A built-in function, is_array(), is available for accomplishing this task. Its prototype follows

boolean is_array(mixed variable)

The is_array() function determines whether variable is an array, returning TRUE if it is and FALSE otherwise.

An example follows:

```
$states = array("Florida");
$state = "Ohio";
printf("\$states is an array: %s <br />", (is_array($states) ? "TRUE" : "FALSE"));
printf("\$state is an array: %s <br />", (is_array($state) ? "TRUE" : "FALSE"));
```

Executing this example produces the following:

```
$states is an array: TRUE
```

$state is an array: FALSE

Outputting an Array

The most common way to output an array's contents is by iterating over each key and echoing the corresponding value. For instance, a foreach statement does the trick nicely:

```
$states = array("Ohio", "Florida", "Texas"); foreach ($states AS $state) {
echo "{$state}<br />";
}
```

Printing Arrays for Testing Purposes

The array contents in most of the previous examples have been displayed using comments. While this works great for instructional purposes, in the real world you'll need to know how to easily output their contents to the screen for testing purposes. This is most commonly done with the print_r() function. Its prototype follows:

```
boolean print_r(mixed variable [, boolean return])
```

The print_r() function accepts a variable and sends its contents to standard output, returning TRUE on success and FALSE otherwise.

Adding and Removing Array Elements

PHP provides a number of functions for both growing and shrinking an array. Some of these functions are provided as a convenience to programmers who wish to mimic various queue implementations (FIFO, LIFO, etc.), as reflected by their names (push, pop, shift, and unshift).

Adding a Value to the Front of an Array

The array_unshift() function adds elements to the front of the array. All preexisting numerical keys are modified to reflect their new position in the array, but associative keys aren't affected. Its prototype follows:

```
int array_unshift(array array, mixed variable [, mixed variable...])
```

The following example adds two states to the front of the $states array:

```
$states = array("Ohio", "New York");
array_unshift($states, "California",
"Texas");
// $states = array("California", "Texas", "Ohio", "New York");
```

Adding a Value to the End of an Array

The array_push() function adds a value to the end of an array, returning the total count of

elements in the array after the new value has been added. You can push multiple variables onto the array simultaneously by passing these variables into the function as input parameters. Its prototype follows:

```
int array_push(array array, mixed variable [, mixed variable...])
```

The following example adds two more states onto the $states array:

```
$states = array("Ohio", "New York");
array_push($states, "California",
"Texas");
// $states = array("Ohio", "New York", "California", "Texas");
```

Removing a Value from the Front of an Array

The array_shift() function removes and returns the first item found in an array. If numerical keys are used, all corresponding values will be shifted down, whereas arrays using associative keys will not be affected. Its prototype follows

```
mixed array_shift(array array)
```

The following example removes the first state from the $states array:

```
$states = array("Ohio", "New York", "California", "Texas");
$state = array_shift($states);
// $states = array("New York", "California", "Texas")
// $state = "Ohio"
```

Removing a Value from the End of an Array

The array_pop() function removes and returns the last element from an array. Its prototype follows:

```
mixed array_pop(array array)
```

The following example removes the last state from the $states array:

```
$states = array("Ohio", "New York", "California", "Texas");
$state = array_pop($states);
// $states = array("Ohio", "New York", "California"
// $state = "Texas"
```

Locating Array Elements

The ability to efficiently sift through data is absolutely crucial in today's information-driven society.

Searching an Array	The in_array() function searches an array for a specific value, returning TRUE if the value is found and FALSE otherwise. Its prototype follows: boolean in_array(mixed needle, array haystack [, boolean strict])
Searching Associative Array Keys	The function array_key_exists() returns TRUE if a specified key is found in an array and FALSE otherwise. Its prototype follows: boolean array_key_exists(mixed key, array array)
Searching Associative Array Values	The array_search() function searches an array for a specified value, returning its key if located and FALSE otherwise. Its prototype follows: mixed array_search(mixed needle, array haystack [, boolean strict])
Retrieving Array Keys	The array_keys() function returns an array consisting of all keys located in an array. Its prototype follows: array array_keys(array array [, mixed search_value [, boolean preserve_keys]])
Retrieving Array Values	The array_values() function returns all values located in an array, automatically providing numeric indexes for the returned array. Its prototype follows: array array_values(array array)

Traversing Arrays

The need to travel across an array and retrieve various keys, values, or both is common, so it's not a surprise that PHP offers numerous functions suited to this need. Many of these functions do double duty: retrieving the key or value residing at the current pointer location, and moving the pointer to the next appropriate location.

Retrieving the Current Array Key	The key() function returns the key located at the current pointer position of the provided array. Its prototype follows: mixed key(array array)

Retrieving the Current Array Value	The current() function returns the array value residing at the current pointer position of the array. Its prototype follows: mixed current(array array)
Retrieving the Current Array Key and Value	The each() function returns the current key/value pair from the array and advances the pointer one position. Its prototype follows: array each(array array)

Moving the Array Pointer

Several functions are available for moving the array pointer.

Moving the Pointer to the Next Array Position	The next() function returns the array value residing at the position immediately following that of the current array pointer. Its prototype follows: mixed next(array array)
Moving the Pointer to the Previous Array Position	The prev() function returns the array value residing at the location preceding the current pointer location, or FALSE if the pointer resides at the first position in the array. Its prototype follows: mixed prev(array array)
Moving the Pointer to the First Array Position	The reset() function serves to set an array pointer back to the beginning of the array. Its prototype follows: mixed reset(array array)
Moving the Pointer to the Last Array Position	The end() function moves the pointer to the last position of an array, returning the last element. Its prototype follows: mixed end(array array)

Passing Array Values to a Function

The array_walk() function will pass each element of an array to the user-defined function. This is useful when you need to perform a particular action based on each array element. If you intend to actually modify the array key/value pairs, you'll need to pass each key/value to the function as a reference.

Its prototype follows

boolean array_walk(array &array, callback function [, mixed userdata])

Determining Array Size and Uniqueness

A few functions are available for determining the number of total and unique array values.

Determining the Size of an Array	The count() function returns the total number of values found in an array. Its prototype follows: integer count(array array [, int mode])
Counting Array Value Frequency	The array_count_values() function returns an array consisting of associative key/value pairs. Its prototype follows: array array_count_values(array array)
Determining Unique Array Values	The array_unique() function removes all duplicate values found in an array, returning an array consisting of solely unique values. Its prototype follows: array array_unique(array array [, int sort_flags = SORT_STRING])

Sorting Arrays

Reversing Array Element Order	The array_reverse() function reverses an array's element order. Its prototype follows: array array_reverse(array array [, boolean preserve_keys])
Flipping Array Keys and Values	The array_flip() function reverses the roles of the keys and their corresponding values in an array. Its prototype follows: array array_flip(array array)
Sorting an Array	The sort() function sorts an array, ordering elements from lowest to highest value. Its prototype follows: void sort(array array [, int sort_flags])
Sorting an Array While Maintaining Key/Value Pairs	The asort() function is identical to sort(), sorting an array in ascending order, except that the key/value correspondence is maintained. Its prototype follows: void asort(array array [, integer sort_flags])

Sorting an Array in Reverse Order	The rsort() function is identical to sort(), except that it sorts array items in reverse (descending) order.Its prototype follows: void rsort(array array [, int sort_flags])
Sorting an Array in Reverse Order While Maintaining	Like asort(), arsort() maintains key/value correlation. However, it sorts the array in reverse order. Its prototype follows: void arsort(array array [, int sort_flags])
Sorting an array Naturally	The natsort() function is intended to offer a sorting mechanism comparable to the mechanisms that people normally use. Its prototype follows:void natsort(array array)
Case-Insensitive Natural Sorting	The function natcasesort() is functionally identical to natsort(), except that it is case insensitive: void natcasesort(array array)
Sorting an Array by Key Values	The ksort() function sorts an array by its keys, returning TRUE on success and FALSE otherwise. Its prototype follows: integer ksort(array array [, int sort_flags])
Sorting Array Keys in Reverse Order	The krsort() function operates identically to ksort(), sorting by key, except that it sorts in reverse (descending) order. Its prototype follows: integer krsort(array array [, int sort_flags])
Sorting According to User-Defined Criteria	The usort() function offers a means for sorting an array by using a user-defined comparison algorithm, embodied within a function. This is useful when you need to sort data in a fashion not offered by one of PHP's built-in sorting functions. Its prototype follows: void usort(array array, callback function_name)

Merging, Slicing, Splicing, and Dissecting Arrays

Merging Arrays	The array_merge() function merges arrays together, returning a single, unified array. The resulting array will begin with the first input array parameter, appending each subsequent array parameter in the order of appearance. Its prototype follows: array array_merge(array array1, array array2 [, array arrayN])
Recursively Appending Arrays	The array_merge_recursive() function operates identically to array_merge(), joining two or more arrays together to form a single, unified array. The difference between the two functions lies in the way that this function behaves when a string key located in one of the input arrays already exists within the resulting array. Note that array_merge() will simply overwrite the preexisting key/value pair, replacing it with the one found in the current input array, while array_merge_recursive() will instead merge the values together, forming a new array with the preexisting key as its name. Its prototype follows: array array_merge_recursive(array array1, array array2 [, array arrayN])
Combining Two Arrays	The array_combine() function produces a new array consisting of a submitted set of keys and corresponding values. Its prototype follows: array array_combine(array keys, array values) Both input arrays must be of equal size, and neither can be empty.
Slicing an Array	The array_slice() function returns a section of an array based on a starting and ending offset value. Its prototype follows: array array_slice(array array, int offset [, int length [, boolean preserve_keys]])

Splicing an Array	The array_splice() function removes all elements of an array found within a specified range, returning those removed elements in the form of an array. Its prototype follows: array array_splice(array array, int offset [, int length [, array replacement]])
Calculating an Array Intersection	The array_intersect() function returns a key-preserved array consisting only of those values present in the first array that are also present in each of the other input arrays. Its prototype follows: array array_intersect(array array1, array array2 [, arrayN])
Calculating Associative Array Intersections	The function array_intersect_assoc() operates identically to array_intersect(), except that it also considers array keys in the comparison. Therefore, only key/value pairs located in the first array that arealso found in all other input arrays will be returned in the resulting array. Its prototype follows: array array_intersect_assoc(array array1, array array2 [, arrayN])
Calculating Array Differences	Essentially the opposite of array_intersect(), the function array_diff() returns those values located in the first array that are not located in any of the subsequent arrays: array array_diff(array array1, array array2 [, arrayN])
Calculating Associative Array Differences	The function array_diff_assoc() operates identically to array_diff(), except that it also considers array keys in the comparison. Therefore, only key/value pairs located in the first array but not appearing in any of the other input arrays will be returned in the result array. Its prototype follows: array array_diff_assoc(array array1, array array2 [, array arrayN])

Strings and Regular Expressions

PHP has long supported two regular expression implementations known as Perl and POSIX.

Regular Expressions

Regular expressions provide the foundation for describing or matching data according to defined syntax rules. A regular expression is nothing more than a pattern of characters itself, matched against a certain parcel of text. This sequence may be a pattern with which you are already familiar, such as the word dog, or it may be a pattern with specific meaning in the context of the world of pattern matching, <(?)>.*<\ /.?>

Regular Expression Syntax (POSIX)

POSIX stands for Portable Operating System Interface for Unix and is representative of a set of standards originally intended for Unix-based operating systems. POSIX regular expression syntax is an attempt to standardize how regular expressions are implemented in many programming languages.

Brackets:Brackets ([]) are used to represent a list, or range, of characters to be matched. For instance, contrary to the regular expression php, which will locate strings containing the explicit string php, the regular expression [php] will find any string containing the character p or h. Several commonly used character ranges follow:

- [a-z] matches any character from lowercase a through lowercase z.
- [A-Z] matches any character from uppercase A through uppercase Z.
- [A-Za-z] matches any character from uppercase A through lowercase z.
- [0-9] matches any decimal digit from 0 through 9.

Quantifiers

Sometimes you might want to create regular expressions that look for characters based on their frequency or position

- p+ matches any string containing at least one p.
- p* matches any string containing zero or more p's.
- p? matches any string containing zero or one p.
- p{2} matches any string containing a sequence of two p's.
- p{2,3} matches any string containing a sequence of two or three p's.
- p{2,} matches any string containing a sequence of at least two p's.
- p$ matches any string with p at the end of it.

Predefined Character Ranges (Character Classes)

For reasons of convenience, several predefined character ranges, also known as character classes, are available. Character classes specify an entire range of characters—for example,

the alphabet or an integer set. Standard classes include the following:

[:alpha:]: Lowercase and uppercase alphabetical characters. This can also be specified as [A-Za-z].

[:alnum:]: Lowercase and uppercase alphabetical characters and numerical digits. This can also be specified as [A-Za-z0-9].

[:cntrl:]: Control characters such as tab, escape, or backspace.

[:digit:]: Numerical digits 0 through 9. This can also be specified as [0-9].

[:graph:]: Printable characters found in the range of ASCII 33 to 126.

[:lower:]: Lowercase alphabetical characters. This can also be specified as [a-z].

[:punct:]: Punctuation characters, including ~ ` ! @ # $ % ^ & * ( ) - _ + = { } [ ] : ; '< > , . ? and /.

[:upper:]: Uppercase alphabetical characters. This can also be specified as [A-Z].

[:space:]:

Whitespace characters, including the space, horizontal tab, vertical tab, new line, form feed, or carriage return.

[:xdigit:]: Hexadecimal characters. This can also be specified as [a-fA-F0-9].

PHP's Regular Expression Functions (POSIX Extended)

PHP offers seven functions for searching strings using POSIX-style regular expressions: ereg(),ereg_replace(), eregi(), eregi_replace(), split(), spliti(), and sql_regcase().

Performing a Case- Sensitive Search	The ereg() function executes a case-sensitive search of a string for a defined pattern, returning the length of the matched string if the pattern is found and FALSE otherwise. Its prototype follows: int ereg(string pattern, string string [, array regs])
Performing a Case-Insensitive Search	The eregi() function searches a string for a defined pattern in a case-insensitive fashion. Its prototype follows: int eregi(string pattern, string string, [array regs])
Replacing Text in a Case-Sensitive Fashion	The ereg_replace() function operates much like ereg(), except that its power is extended to finding and replacing a pattern with a replacement string instead of simply locating it. Its prototype follows: string ereg_replace(string pattern, string replacement, string string)

Replacing Text in a Case-Insensitive Fashion	The eregi_replace() function operates exactly like ereg_replace(), except that the search for pattern in string is not case sensitive. Its prototype follows: string eregi_replace(string pattern, string replacement, string string)
Splitting a String into Various Elements Based on a Case-Sensitive Pattern	The split() function divides a string into various elements, with the boundaries of each element based on the occurrence of a defined pattern within the string. Its prototype follows: array split(string pattern, string string [, int limit])
Splitting a String into Various Elements Based on a Case-Insensitive Pattern	The spliti() function operates exactly in the same manner as its sibling, split(), except that its pattern is treated in a case-insensitive fashion. Its prototype follows: array spliti(string pattern, string string [, int limit])
Accommodating Products Supporting Solely Case-Sensitive Regular Expressions	The sql_regcase() function converts each character in a string into a bracketed expression containing two characters. If the character is alphabetical, the bracket will contain both forms; otherwise, the original character will be left unchanged. Its prototype follows: string sql_regcase(string string)

Regular Expression Syntax (Perl)

Modifiers

Often you'll want to tweak the interpretation of a regular expression; for example, you may want to tell the regular expression to execute a case-insensitive search or to ignore comments embedded within its syntax. These tweaks are known as modifiers, and they go a long way toward helping you to write short and concise expressions.

Ex:

I: Perform a case-insensitive search.

G: Find all occurrences (perform a global search).

Meta characters

Perl regular expressions also employ meta characters to further filter their searches. A meta character is simply an character or character sequence that symbolizes special meaning. A list of

useful meta characters follows

\A: Matches only at the beginning of the string.

\b: Matches a word boundary.

\B: Matches anything but a word boundary.

\d: Matches a digit character. This is the same as [0-9].

\D: Matches a nondigit character.

\s: Matches a whitespace character.

\S: Matches a nonwhite space character

[]: Encloses a character

(): Encloses a character grouping or defines a back reference.

$: Matches the end of a line.

^: Matches the beginning of a line.

^: Matches any character except for the newline.

\: Quotes the next metacharacter.

\w: Matches any string containing solely underscore and alphanumeric characters.

This is the same as [a-zA-Z0-9_].

W: Matches a string, omitting the underscore and alphanumeric characters.

PHP's Regular Expression Functions (Perl Compatible)

PHP offers eight functions for searching and modifying strings using Perl-compatible regular expressions: preg_filter(), preg_grep(), preg_match(), preg_match_all(), preg_quote(), preg_replace(), preg_replace_callback(), and preg_split().

Searching an Array	The preg_grep() function searches all elements of an array, returning an array consisting of all elements matching a certain pattern. Its prototype follows: array preg_grep(string pattern, array input [, int flags])
Searching for a Pattern	The preg_match() function searches a string for a specific pattern, returning TRUE if it exists and FALSE otherwise. Its prototype follows: int preg_match(string pattern, string string [, array matches] [, int flags [, int offset]]])

Matching All Occurrences of a Pattern	The preg_match_all() function matches all occurrences of a pattern in a string, assigning each occurrence to an array in the order you specify via an optional input parameter. Its prototype follows: int preg_match_all(string pattern, string string, array matches [, int flags] [, int offset]))
Delimiting Special Regular Expression Characters	The function preg_quote() inserts a backslash delimiter before every character of special significance to regular expression syntax. These special characters include $ ^ * ( ) + = { } [ ] | \\ : < >. Its prototype follows: string preg_quote(string str [, string delimiter])
Replacing All Occurrences of a Pattern	The preg_replace() function replaces all occurrences of pattern with replacement, and returns the modified result. Its prototype follows: mixed preg_replace(mixed pattern, mixed replacement, mixed str [, int limit [, int count]])
Creating a Custom Replacement Function	Consider a situation where you want to scan some text for acronyms such as IRS and insert the complete name directly following the acronym. To do so, you need to create a custom function and then use the function
Splitting a String into Various Elements Based on a Case- Insensitive Pattern	The preg_split() function operates exactly like split(), except that pattern can also be defined in terms of a regular expression. Its prototype follows: array preg_split(string pattern, string string [, int limit [, int lags]])

UNIT-4

TCL stands for “Tool Command Language” and is pronounced “tickle”; is a simple scripting language for controlling and extending applications. TCL is a radically simple open-source interpreted programming language that provides common facilities such as variables, procedures, and control structures as well as many useful features that are not found in any other major language. TCL runs on almost all modern operating systems such as Unix, Macintosh, and Windows (including Windows Mobile). While TCL is flexible enough to be used in almost any application imaginable, it does excel in a few key areas, including: automated interaction with external programs, embedding as a library into application programs, language design, and general scripting. TCL was created in 1988 by John Ousterhout and is distributed under a BSD style license (which allows you everything GPL does, plus closing your source code).

The current stable version, in February 2008, is 8.5.1 (8.4.18 in the older 8.4 branch). The first major GUI extension that works with TCL is TK, a toolkit that aims to rapid GUI development. That is why TCL is now more commonly called TCL/TK. The language features far-reaching introspection, and the syntax, while simple2, is very different from the Fortran/Algol/C++/Java world. Although TCL is a string based language there are quite a few object-oriented extensions for it like Snit3, incr Tcl4, and XOTcl5 to name a few. TCL is embeddable: its interpreter is implemented as a library of C procedures that can easily be incorporated into applications, and each application can extend the core TCL features with additional commands specific to that application.

Tcl was originally developed as a reusable command language for experimental computer aided design (CAD) tools. The interpreter is implemented as a C library that could be linked into any application. It is very easy to add new functions to the TCL interpreter, so it is an ideal reusable "macro language" that can be integrated into many applications. However, TCL is a programming language in its own right, which can be roughly described as a cross-breed between

- LISP/Scheme (mainly for its tail-recursion capabilities)
- C (control structure keywords, expr syntax) and
- Unix shells (but with more powerful structuring).

TCL Structure

The TCL language has a tiny syntax - there is only a single command structure, and a set of rules to determine how to interpret the commands. Other languages have special syntaxes for control structures (if, while, repeat...) - not so in TCL. All such structures are implemented as commands. There is a runtime library of compiled 'C' routines, and the 'level' of the GUI interface is quite high.

Comments: If the first character of a command is #, it is a comment.

TCL commands: TCL commands are just words separated by spaces. Commands return strings, and arguments are just further words.

command argument command
argument

Spaces are important

expr 5*3 has a single
argument expr 5 * 3 has three
arguments

TCL commands are separated by a new line, or a semicolon, and arrays are indexed by text

set a(a\ text\ index) 4

Syntax

Syntax is just the rules how a language is structured. A simple syntax of English could say(Ignoring punctuation for the moment) A text consists of one or more sentences A sentence consists of one or more words' Simple as this is, it also describes Tcl's syntax very well - if you say "script" for "text", and "command" for "sentence". There's also the difference that a Tcl word can again contain a script or a command. So if {$x < 0} {set x 0} is a command consisting of three words: if, a condition in braces, a command (also consisting of three words) in braces. Take this for example is a well-formed Tcl command: it calls Take (which must have been defined before) with the three arguments "this", "for", and "example". It is up to the command how it interprets its arguments, e.g. puts acos(-1) will write the string "acos(-1)" to the stdout channel, and return the empty string "", while expr acos(-1) will compute the arc cosine of -1 and return 3.14159265359 (an approximation of Pi), or string length acos(-1) will invoke the string command, which again dispatches to its length sub-command, which determines the length of the second argument and returns 8.A Tcl script is a string that is a sequence of commands, separated by newlines or semicolons. A command is a string that is a list of words, separated by blanks. The first word is the name of the command; the other words

are passed to it as its arguments.

In Tcl, "everything is a command" - even what in other languages would be called declaration, definition, or control structure. A command can interpret its arguments in any way it wants - in particular, it can implement a different language, like expression. A word is a string that is a simple word, or one that begins with { and ends with the matching } (braces), or one that begins with " and ends with the matching ". Braced words are not evaluated by the parser. In quoted words, substitutions can occur before the command is called: $[A-Za-z0-9_]+ substitutes the value of the given variable. Or, if the variable name contains characters outside that regular expression, another layer of bracing helps the parser to get it right

```
puts "Guten Morgen, ${Schuler}!"
```

If the code would say $Schuler, this would be parsed as the value of variable $Sch, immediately followed by the constant string üler.(Part of) a word can be an embedded script: a string in [] brackets whose contents are evaluated as a script (see above) before the current command is called.In short: Scripts and commands contain words. Words can again contain scripts and commands. (This can lead to words more than a page long...)

Arithmetic and logic expressions are not part of the Tcl language itself, but the language of the expr command (also used in some arguments of the if, for, while commands) is basically equivalent to C's expressions, with infix operators and functions.

Rules of TCL

The following rules define the syntax and semantics of the Tcl language:

(1) Commands A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

(2) Evaluation A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

(3) Words of a command are separated by white space (except for newlines, which are command separators).

(4) Double quotes If the first character of a word is double-quote (") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

(5) Braces If the first character of a word is an open brace ({) then the word is terminated by the matching close brace (}). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

(6) Command substitution If a word contains an open bracket ([) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (``]). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

(7) Variable substitution If a word contains a dollar-sign ($) then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

$name

Tcl: the Tool Command language

Name is the name of a scalar variable; the name is a sequence of one or more characters that are a letter, digit, underscore, or namespace separators (two or more colons).

$name(index)

Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, underscores, and namespace separators, and may be an empty string.

Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

${name}

Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces. There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

(8) Backslash substitution If a backslash (\) appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

\a

Audible alert (bell) (0x7).

\b

Backspace (0x8).

\f

Form feed (0xc).

\n

Newline (0xa).

\r

Carriage-return (0xd).

\t

Tab (0x9).

\v

Vertical tab (0xb).

\<newline>whitespace

A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

Contents

Literal backslash (\), no special effect.

\ooo

The digits ooo (one, two, or three of them) give an eight-bit octal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.

\xhh

The hexadecimal digits hh give an eight-bit hexadecimal value for the Unicode character that will be inserted. Any number of hexadecimal digits may be present; however, all but the last two are ignored (the result is always a one-byte quantity). The upper bits of the Unicode character will be 0.

\uhhhh

The hexadecimal digits hhhh (one, two, three, or four of them) give a sixteen-bit hexadecimal value for the Unicode character that will be inserted. Backslash substitution is not performed on words enclosed in braces, except for backslash newline as described above.

(9) Comments If a hash character (#) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

(10) Order of substitution Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command.

For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script. Substitutions take place from left to right, and each substitution is evaluated completely before attempting to evaluate the next. Thus, a sequence like set y [set x 0][incr x][incr x] will always set the variable y to the value, 012.

(11) Substitution and word boundaries Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

Variables and Data in TCL

As noted above, by default, variables defined inside a procedure are "local" to that procedure. And, the argument variables of the procedure contain local "copies" of the argument data used to invoke the procedure.

These local variables cannot be seen elsewhere in the script, and they only exist while the procedure is being executed. In the "getAvg" procedure above, the local variables created in the procedure are "n" "r" and "avg". TCL provides two commands to change the scope of a variable inside a procedure, the "global" command and the "upvar" command. The "global" command is used to declare that one or more variables are not local to any procedure. The value of a global variable will persist until it is explicitly changed. So, a variable which is declared with the "global" command can be seen and changed from inside any procedure which also declares that variable with the "global" command. Variables which are defined outside of any procedure are automatically global by default. The TCL "global" command declares that references to a given variable should be global rather than local. However, the "global" command does not create or set the variable … this must be done by other means, most commonly by the TCL "set" command.

For example, here is an adjusted version of our averaging procedure which saves the input list length in the global variable "currentLength" so that other parts of the script can access this information after "getAvgN" is called:

```
proc getAvgN { rList } \
{
    global currentLength

    set currentLength [llength $rList]

     if {!$currentLength} {return 0.0}

    set avg 0.0

    foreach r $rList \
    {
    set avg [expr $avg + $r]
    }

    set avg [expr $avg/double($currentLength)]

    return $avg
}
```

Then, this adjusted version "getAvgN" could be used elsewhere as follows

```
global currentLength
set thisList "1.0 2.0 3.0"
set a [getAvgN $thisList]
puts "List: $thisList Length: $currentLength Avg: $a"
```

We can also use global variables as an alternative to procedure arguments. For example, we can make a version of our averaging application which assumes that the input list is stored in a global variable called "currentList"

```
proc getCurrentAvg { } \
{
  global currentList currentLength

  set currentLength [llength $rList]

  if {!$currentLength} {return 0.0}

  set avg 0.0

  foreach r $currentList \
    {
     set avg [expr $avg + $r]
    }

  set avg [expr $avg/double($currentLength)]

  return $avg
}
```

Then, this adjusted version "getCurrentAvg" could be used elsewhere as follows

```
global currentList currentLength
set currentList "1.0 2.0 3.0"
set a [getCurrentAvg]
puts "List: $currentList Len: $currentLength Avg: $a"
```

A procedure can use global variables for persistent storage of information, including the possibility to test whether the procedure has been called previously; this is useful for procedures that might need to perform a one-time initialization. In these cases, a procedure will use a global variable which is not set anywhere else. This means, the first time the procedure is called, the global variable will not yet exist (recall that the "global" statement declares that a variable will be accessed as a global variable, but it does not define or create the variable itself).

The TCL command "info exists" will evaluate to true if the given variable exists. For example, suppose we wanted to make a version of our procedure "getAvg" which keeps an internal count of how many times it has been called. In this version, we use a global variable named "callCount_getAvg" to keep track of the number of times "getAvg" is called. Because this global variable will actually be used to store information for the specific use of the "getAvg" procedure, we need to choose a global variable name which will not be used for a similar purpose in some other procedure. The first time "getAvg" is called, the global variable does not yet exist, and must be set to zero.

```
proc getAvg { rList } \
{
 global callCount_getAvg
          if {![info exists callCount_getAvg]} \
          {
          set callCount_getAvg 0
          }
 incr callCount_getAvg
puts "getAvg has been called $callCount_getAvg times"
set n [llength $rList]
if {!$n}
{return 0.0}
set avg 0.0
foreach r $rList \
{
  set avg [expr $avg + $r]
}
set avg [expr $avg/double($n)]
return $avg
}
```

A more flexible way to manipulate persistent data is to use global arrays rather than scalar variables. For example, instead of the procedure-specific scalar variable "callCount_getAvg" used above, we can use a general-purpose array "callCount()" which could be used to record the call counts of any number of procedures, by using the procedure name as the array index. Many nmrWish TCL scripts use global arrays in this fashion, to simplify the sharing of

many data values between procedures. Here is a version of the "getAvg" procedure with the call count tallied in a global array location … note that an array is declared global simply by listing its name in a "global" command, exactly as for a scalar variable; no ( ) parenthesis or index values are used.

```
proc getAvg { rList } \
{
  global callCount

   if {![info exists callCount(getAvg)]} \
     {
      set callCount(getAvg) 0
     }

  incr callCount(getAvg)

  puts "getAvg has been used $callCount(getAvg) times"

  set n [llength $rList]

  if {!$n} {return 0.0}

 set avg 0.0

  foreach r $rList \
    {
     set avg [expr $avg + $r]
    }

  set avg [expr $avg/double($n)]

  return $avg
}
```

TCL Variable Scope and the upvar Command

We have already seen that TCL procedures can generate a return value as a way to pass information back to their caller. And, we have also seen that global variables can be used to share information between parts of a TCL script, and so these also serve as a mechanism for returning information to a caller. TCL includes the "upvar" command as a method for a given procedure to change the values of variables in the scope of its caller. This provides a way for a procedure to provide additional information to the caller, besides by using the procedure's return value.

In the "upvar" scheme, a procedure's caller provides the names of one or more of its own variables as arguments to the procedure. The procedure then uses the "upvar" command to map these variables from the caller onto variables in the procedure. For example, here the caller passes its variable name "count" as the first argument to procedure "getNAvg":

```
set count 0
set a [getNAvg count "1.0 2.0 3.0 4.0"]
```

Then, in this version of procedure "getNArg" the "upvar" command is used to map the first argument value "$nPtr" onto the procedure's variable called "n" … this means that whenever the procedure gets or changes the value of variable "n" it will actually be using the caller's variable "count".

```
proc getNAvg { nPtr rList } \
{
  upvar $nPtr n

  set n [llength $rList]

  if {!$n} {return 0.0}

  set avg 0.0

 foreach r $rList \
    {
     set avg [expr $avg + $r]
    }

  set avg [expr $avg/double($n)]

  return $avg
}
```

Control Flow: In Tcl language there are several commands that are used to alter the flow of a program. When a program is run, its commands are executed from the top of the source file to the bottom. One by one. This flow can be altered by specific commands. Commands can be executed multiple times. Some commands are conditional. They are executed only if a specific condition is met.

The if command

The if command has the following general form:

if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?

The if command is used to check if an expression is true. If it is true, a body of command(s) is then executed. The body is enclosed by curly brackets.

The if command evaluates an expression. The expression must return a boolean value. In Tcl, 1, yes, true mean true and 0, no, false mean false.

```
!/usr/bin/tclsh
if yes {
   puts "This message is always shown"
}
```

In the above example, the body enclosed by { } characters is always executed.

```
#!/usr/bin/tclsh
if true then {
   puts "This message is always shown"
}
```

The then command is optional. We can use it if we think, it will make the code more clear. We can use the else command to create a simple branch. If the expression inside the square brackets following the if command evaluates to false, the command following the else command is automatically executed.

```
#!/usr/bin/tclsh
set sex female
if {$sex == "male"}
 {
   puts "It is a boy"
} else
 {
   puts "It is a girl"
}
```

We have a sex variable. It has "female" string. The Boolean expression evaluates to false and we get "It is a girl" in the console.

$ ./girlboy.tcl

It is a girl

We can create multiple branches using the elseif command. The elseif command tests for another condition, if and only if the previous condition was not met. Note that we can use multiple elseif commands in our tests.

```
#!/usr/bin/tclsh
# nums.tcl
puts -nonewline "Enter a number: "
flush stdout
set a [gets stdin]
if {$a < 0}
{
   puts "the number is negative"
} elseif { $a == 0 }
 {
   puts "the numer is zero"
} else
{
   puts "the number is positive"
}
```

In the above script we have a prompt to enter a value. We test the value if it is a negative number or positive or if it equals to zero. If the first expression evaluates to false, the second expression is evaluated. If the previous conditions were not met, then the body following the else commands would be executed.

$ ./nums.tcl

Enter a number: 2
the number is positive
$ ./nums.tcl
Enter a number: 0

the numer is zero

$ ./nums.tcl

Enter a number: -3

the number is negative

Running the example multiple times.

Switch command

The switch command matches its string argument against each of the pattern arguments in order. As soon as it finds a pattern that matches the string it evaluates the following body argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last pattern argument is default then it matches anything. If no pattern argument matches string and no default is given, then the switch command returns an empty string.

```
#!/usr/bin/tclsh
# switch_cmd.tcl
puts -nonewline "Select a top level domain name:"
flush stdout
gets stdin domain
switch $domain
{
   us { puts "United States" }
   de { puts Germany }
   sk { puts Slovakia }
   hu { puts Hungary }
   default { puts "unknown" }
}
```

In our script, we prompt for a domain name. There are several options. If the value equals for example to us the "United States" string is printed to the console. If the value does not match to any given value, the default body is executed and unknown is printed to the console.

$ ./switch_cmd.tcl

Select a top level domain name:sk

Slovakia

We have entered sk string to the console and the program responded with Slovakia.

While command: The while command is a control flow command that allows code to be executed repeatedly based on a given Boolean condition. The while command executes the commands inside the block enclosed by curly brackets. The commands are executed each time the expression is evaluated to true.

```
#!/usr/bin/tclsh
# whileloop.tcl
set i 0
set sum 0
while { $i < 10 }
 {
   incr i
   incr sum $i
}
puts $sum
```

In the code example, we calculate the sum of values from a range of numbers. The while loop has three parts: initialization, testing, and updating. Each execution of the command is called a cycle.

```
set i 0
```

We initiate the i variable. It is used as a counter.

```
while { $i < 10 }
 {
...
}
```

The expression inside the curly brackets following the while command is the second phase, the testing. The commands in the body are executed, until the expression is evaluated to false.

```
incr i
```

The last, third phase of the while loop is the updating. The counter is incremented. Note that improper handling of the while loops may lead to endless cycles.

FOR command: When the number of cycles is know before the loop is initiated, we can use the for command. In this construct we declare a counter variable, which is automatically increased or decreased in value during each repetition of the loop.

```
#!/usr/bin/tclsh
for {set i 0} {$i < 10} {incr i}
 {
   puts $i
}
```

In this example, we print numbers 0..9 to the console.

```
for {set i 0} {$i < 10} {incr i}
 {
   puts $i
}
```

There are three phases. First, we initiate the counter i to zero. This phase is done only once. Next comes the condition. If the condition is met, the command inside the for block is executed. Then comes the third phase; the counter is increased. Now we repeat phases 2 and 3 until the condition is not met and the for loop is left. In our case, when the counter i is equal to 10, the for loop stops executing.

```
$ ./forloop.tcl

0

1
2
3
4
5
```

6
7
8
9

Here we see the output of the forloop.tcl script.

The foreach command: The foreach command simplifies traversing over collections of data. It has no explicit counter. It goes through a list element by element and the current value is copied to a variable defined in the construct.

```
#!/usr/bin/tclsh
set planets { Mercury Venus Earth Mars Jupiter Saturn     Uranus Neptune }
foreach planet $planets
 {
   puts $planet
}
```

In this example, we use the foreach command to go through a list of planets.

```
foreach planet $planets
 {
   puts $planet
}
```

The usage of the foreach command is straightforward. The planets is the list that we iterate through. The planet is the temporary variable that has the current value from the list. The for each command goes through all the planets and prints them to the console.

```
$ ./planets.tcl
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

Running the above Tcl script gives this output.

```
#!/usr/bin/tclsh
set actresses { Rachel Weiss Scarlett Johansson Jessica Alba \
   Marion Cotillard Jennifer Connelly}
foreach {first second} $actresses
{
   puts "$first $second"
}
```

In this script, we iterate througn pairs of values of a list.

```
foreach {first second} $actresses
{
   puts "$first $second"
}
```

We pick two values from the list at each iteration.

```
$ ./actresses.tcl
Rachel Weiss
Scarlett Johansson
Jessica Alba
Marion Cotillard
Jennifer Connelly
```

This is the output of actresses tcl script

```
#!/usr/bin/tclsh
foreach i { one two three } item {car coins rocks}
 {
   puts "$i $item"
}
```

We can iterate over two lists in parallel.

$ ./parallel.tcl

one car

two coins

three rocks

This is the output of the parallel.tcl script.

The break and continue commands: The break command can be used to terminate a block defined by while, for, or switch commands.

```
#!/usr/bin/tclsh
while true
{
    set r [expr 1 + round(rand()*30)]
    puts -nonewline "$r "
    if {$r == 22} { break }
}
puts ""
```

We define an endless while loop. We use the break command to get out of this loop. We choose a random value from 1 to 30 and print it. If the value equals to 22, we finish the endless while loop.

```
set r [expr 1 + round(rand()*30)]
```

Here we calculate a random number between 1..30. The rand() is a built-in Tcl procedure. It returns a random number from 0 to 0.99999. The rand()*30 returns a random number between 0 to 29.99999. The round() procedure rounds the final number.$ ./breakcommand.tcl 28 20 8 8 12 22 .We might get something like this.The continue command is used to skip a part of the loop and continue with the next iteration of the loop. It can be used in combination with for and while commands. In the following example, we will print a list of numbers that cannot be divided by 2 without a remainder.

```
#!/usr/bin/tclsh
set num 0
while { $num < 100 }
 {
   incr num
   if {$num % 2 == 0} { continue }
    puts "$num "
}
puts ""
```

We iterate through numbers 1..99 with the while loop.

```
if {$num % 2 == 0} { continue }
```

If the expression num % 2 returns 0, the number in question can be divided by 2. The continue command is executed and the rest of the cycle is skipped. In our case, the last command of the loop is skipped and the number is not printed to the console. The next iteration is started.

Data Structures

The list is the basic Tcl data structure. A list is simply an ordered collection of stuff; numbers, words, strings, or other lists. Even commands in Tcl are just lists in which the first list entry is the name of a proc, and subsequent members of the list are the arguments to the proc. Lists can be created in several way by setting a variable to be a list of values set lst {{item 1} {item 2} {item 3}} with the split command set lst [split "item 1.item 2.item 3" "."] with the list command. set lst [list "item 1" "item 2" "item 3"] An individual list member can be accessed with the index command. The brief description of these commands is

list **?arg1? ?arg2? ... ?argN?**
makes a list of the arguments
split **string ?splitChars?**
Splits the *string* into a list of items wherever the *splitChars* occur in the

code. *SplitChars* defaults to being whitespace. Note that if there are two or more *splitChars* then each one will be used individually to split the string. In other words: split "1234567" "36" would return the following list: {12 45 7}.lindex **list index**

Returns the *index*'th item from the list.

Note: lists start from 0, not 1, so the first item is at index 0, the second item is at index 1, and so on.llength **list.**Returns the number of elements in a list.The items in list can be iterated through using the foreach command.foreach **varname list body** The foreach command will execute the *body* code one time for each list item in *list*. On each pass, *varname* will contain the value of the next *list* item.In reality, the above form of foreach is the simple form, but the command is quite powerful. It will allow you to take more than one variable at a time from the list: foreach {a b} $listofpairs { ... }. You can even take a variable at a time from multiple lists! For xample: foreach a $listOfA b $listOfB { ... }

Examples

```
set x "a b c"
puts "Item at index 2 of the list {$x} is: [lindex $x 2]\n"
set y [split 7/4/1776 "/"]
puts "We celebrate on the [lindex $y 1]'th day of the [lindex $y 0]'th month\n"
set z [list puts "arg 2 is $y" ]
puts "A command resembles: $z\n"
set i 0
foreach j $x
{
   puts "$j is item number $i in list x"
   incr i
}
```

Adding and deleting members of a list

The commands for adding and deleting list members are

concat *?arg1 arg2 ... argn?*

Concatenates the *args* into a single list. It also eliminates leading and trailing spaces in the args and adds a single separator space between args. The *args*to concat may be either individual elements, or lists. If an *arg* is already a list, the contents of that list is concatenated

with the other *args*.

lappend *list Name ?arg1 arg2 ... argn?*Appends the *args* to the list *listName* treating each *arg* as a list element.

linsert *list Name index arg1 ?arg2 ... argn?*Returns a new list with the new list elements inserted just before the *index* th element of *listName*. Each element argument will become a separate element of the new list. If index is less than or equal to zero, then the new elements are inserted at the beginning of the list. If index has the value *end* , or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

lreplace *list Name first last ?arg1 ... argn?*Returns a new list with N elements of *listName* replaced by the *args*. If *first* is less than or equal to 0, lreplace starts replacing from the first element of the list.If *first* is greater than the end of the list, or the word end, then lreplace behaves like lappend. If there are fewer *args* than the number of positions between *first* and *last*, then the positions for which there are no *args* are deleted.

lset *varName index newValue*

The lset command can be used to set elements of a list directly, instead of using lreplace. Lists in Tcl are the right data structure to use when you have an arbitrary number of things, and you'd like to access them according to their order in the list. In C, you would use an array. In Tcl, arrays are associated arrays - hash tables, as you'll see in the coming sections. If you want to have a collection of things, and refer to the Nth thing (give me the 10th element in this group of numbers), or go through them in order via foreach. Take a look at the example code, and pay special attention to the way that sets of characters are grouped into single list elements.

Example

```
set b [list a b {c d e} {f {g h}}]
puts "Treated as a list: $b\n"
set b [split "a b {c d e} {f {g h}}"]
puts "Transformed by split: $b\n"
set a [concat a b {c d e} {f {g h}}]
puts "Concated: $a\n"
lappend a {ij K lm}                    ;
# Note: {ij K lm} is a single element
puts "After lappending: $a\n"
```

```
set b [linsert $a 3 "1 2 3"]            ;
# "1 2 3" is a single element
puts "After linsert at position 3: $b\n"
set b [lreplace $b 3 5 "AA" "BB"]
puts "After lreplacing 3 positions with 2 values at position 3: $b\n"
```

More list commands - lsearch, lsort, lrange

Lists can be searched with the lsearch command, sorted with the lsort command, and a range of list entries can be extracted with the lrange command.

lsearch ***list pattern***

Searches *list* for an entry that matches *pattern*, and returns the index for the first match, or a -1 if there is no match. By default, lsearch uses "glob" patterns for matching. See the section on globbing.

lsort ***list***

Sorts *list* and returns a new list in the sorted order. By default, it sorts the list into alphabetic order. Note that this command returns the sorted list as a result, instead of sorting the list in place. If you have a list in a variable, the way to sort it is like so: set lst [lsort $lst] lrange ***list first last***

Returns a list composed of the *first* through *last* entries in the list. If *first* is less than or equal to 0, it is treated as the first list element. If *last* is **end** or a value greater than the number of elements in the list, it is treated as the end. If *first* is greater than *last* then an empty list is returned.

Example

```
set list [list {Washington 1789} {Adams 1797} {Jefferson 1801} \
        {Madison 1809} {Monroe 1817} {Adams 1825} ]
set x [lsearch $list Washington*]
set y [lsearch $list Madison*]
incr x
incr y -1                ;# Set range to be not-inclusive
```

```
set subsetlist [lrange $list $x $y]
puts "The following presidents served between Washington and Madison"
foreach item $subsetlist {
   puts "Starting in [lindex $item 1]: President [lindex $item 0] "
}
set x [lsearch $list Madison*]
set srtlist [lsort $list]
set y [lsearch $srtlist Madison*]
puts "\n$x Presidents came before Madison chronologically"
puts "$y Presidents came before Madison alphabetically"
```

Input / Output

Tcl uses objects called channels to read and write data. The channels can be created using the open or socket command. There are three standard channels available to Tcl scripts without explicitly creating them. They are automatically opened by the OS for each new application. They are stdin, stdout and stderr. The standard input, stdin, is used by the scripts to read data. The standard output, stdout, is used by scripts to write data. The standard error, stderr, is used by scripts to write error messages.In the first example, we will work with the puts command. It has the following synopsis:

puts ?-nonewline? ?channelId? string

The channelId is the channel where we want to write text. The channelId is optional. If not specified, the default stdout is assumed.

```
#!/usr/bin/tclsh
puts "Message 1"
puts stdout "Message 2"
puts stderr "Message 3"
```

The puts command writes text to the channel.

```
puts "Message 1"

puts stdout "Message 2"
```

```
puts stderr "Message 3"
```

If we do not specify the channelId, we write to stdout by default. This line does the same thing as the previous one. We only have explicitly specified the channelId.

We write to the standard error channel. The error messages go to the terminal by default.

```
$ ./printing.tcl
Message 1
Message 2
Message 3
```

Example output.

The read command: The read command is used to read data from a channel. The optional argument specifies the number of characters to read. If omitted, the command reads all of the data from the channel up to the end.

```
#!/usr/bin/tclsh
set c [read stdin 1]
while {$c != "q"}
 {
   puts -nonewline "$c"
   set c [read stdin 1]
}
```

The script reads a character from the standard input channel and then writes it to the standard output until it encounters the q character.

```
set c [read stdin 1]
```

We read one character from the standard input channel (stdin).

```
while {$c != "q"} {
```

We continue reading characters until the q is pressed.

The gets command

The gets command reads the next line from the channel, returns everything in the line up to (but

not including) the end-of-line character.

```
#!/usr/bin/tclsh
puts -nonewline "Enter your name: "
flush stdout
set name [gets stdin]
puts "Hello $name"
```

The script asks for input from the user and then prints a message. The puts command is used to print messages to the terminal. The -no newline option suppresses the new line character. Tcl buffers output internally, so characters written with puts may not appear immediately on the output file or device. The flush command forces the output to appear immediately.

```
puts -no newline "Enter your name: "
flush stdout
set name [gets stdin]
```

The gets command reads a line from a channel.

```
$ ./hello.tcl
Enter your name: Jan
Hello Jan
```

Sample output of the script.

The pwd and cd commands

Tcl has pwd and cd commands, similar to shell commands. The pwd command returns the current working directory and the cd command is used to change the working directory.

```
#!/usr/bin/tclsh
set dir [pwd]
puts $dir
cd ..
set dir [pwd]
puts $dir
```

In this script, we will print the current working directory. Then we change the working directory and print the working directory again.

set dir [pwd]

The pwd command returns the current working directory.

cd ..

We change the working directory to the parent of the current directory. We use the cd command.

$ ./cwd.tcl

/home/janbodnar/prog/tcl/io

/home/janbodnar/prog/tcl

Sample output.

The glob command

Tcl has a glob command which returns the names of the files that match a pattern.

```
#!/usr/bin/tclsh
set files [glob *.tcl]
foreach file $files
 {
   puts $file
}
```

The script prints all files with the .tcl extension to the console. The glob command returns a list of files that match the *.tcl pattern.

```
set files [glob *.tcl]
foreach file $files
 {
   puts $file
}
```

We go through the list of files and print each item of the list to the console.

$ ./globcmd.tcl

attributes.tcl

allfiles.tcl
printing.tcl
hello.tcl
read.tcl
files.tcl
globcmd.tcl
write2file.tcl
cwd.tcl
readfile.tcl
isfile.tcl
addnumbers.tcl
This is a sample output of the globcmd.tcl script.

Procedures

A procedure is a code block containing a series of commands. Procedures are called functions in many programming languages. It is a good programming practice for procedures to do only one specific task. Procedures bring modularity to programs. The proper use of procedures brings the following advantages

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

There are two basic types of procedures: built-in procedures and user defined ones. The built-in procedures are part of the Tcl core language. For instance, the rand(), sin() and exp() are built-in procedures. The user defined procedures are procedures created with the proc keyword.The proc keyword is used to create new Tcl commands. The term procedures and commands are often used interchangeably. We start with a simple example.

```
#!/usr/bin/tclsh
proc tclver {}
{
```

```
    set v [info tclversion]
    puts "This is Tcl version $v"
}
tclver
```

In this script, we create a simple tclver procedure. The procedure prints the version of Tcl language. proc tclver {}

{

The new procedure is created with the proc command. The {} characters reveal that the procedure takes no arguments.

```
    {
    set v [info tclversion]
    puts "This is Tcl version $v"
     }
tclver
```

This is the body of the tclver procedure. It is executed when we execute the tclver command. The body of the command lies between the curly brackets.The procedure is called by specifying its name.

```
$ ./version.tcl
This is Tcl version 8.6
```

Sample output.

Procedure arguments: An argument is a value passed to the procedure. Procedures can take one or more arguments. If procedures work with data, we must pass the data to the procedures. In the following example, we have a procedure which takes one argument.

```
#!/usr/bin/tclsh
proc ftc {f}
{
    return [expr $f * 9 / 5 + 32]
}
```

```
puts [ftc 100]
puts [ftc 0]
puts [ftc 30]
```

We create a ftc procedure which transforms Fahrenheit temperature to Celsius temperature. The procedure takes one parameter. Its name f will be used in the body of the procedure.

```
proc ftc {f} {
return [expr $f * 9 / 5 + 32]
puts [ftc 100]
```

We compute the value of the Celsius temperature. The return command returns the value to the caller. If the procedure does not execute an explicit return, then its return value is the value of the last command executed in the procedure's body.The ftc procedure is executed. It takes 100 as a parameter. It is the temperature in Fahrenheit. The returned value is used by the puts command, which prints it to the console. Output of the example.

```
$ ./fahrenheit.tcl
212
32
86
```

Next we will have a procedure which takes two arguments.

```
#!/usr/bin/tclsh
proc maximum {x y}
{
      if {$x > $y}
 {
      return $x
   }
 else
 {
          return $y
   }
```

```
}
set a 23
set b 32
set val [maximum $a $b]
puts "The max of $a, $b is $val"
```

The maximum procedure returns the maximum of two values. The method takes two arguments.

```
proc maximum {x y}
{
if {$x > $y}
 {
   return $x
}
 else
{
       return $y
}
```

Here we compute which number is greater. We define two variables which are to be compared.

```
set a 23
set b 32
set val [maximum $a $b]
```

We calculate the maximum of the two variables. This is the output of the maximum.tcl script.

$ ./maximum.tcl

The max of 23, 32 is 32

Variable number of arguments A procedure can take and process variable number of arguments. For this we use the special arguments and parameter.

```
#!/usr/bin/tclsh
```

```
proc sum {args} {

    set s 0

    foreach arg $args {

        incr s $arg
    }

    return $s
}

puts [sum 1 2 3 4]
puts [sum 1 2]
puts [sum 4]
```

We define a sum procedure which adds up all its arguments. The sum procedure has a special args argument. It has a list of all values passed to the procedure.

```
proc sum {args}
{
foreach arg $args
{
   incr s $arg
}
We go through the list and calculate the sum.
puts [sum 1 2 3 4]
puts [sum 1 2]
puts [sum 4]
```

We call the sum procedure three times. In the first case, it takes 4 arguments, in the second case 2, in the last case one. Output of the variable tcl script

```
$ ./variable.tcl

10
3
```

4

Implicit arguments

The arguments in Tcl procedures may have implicit values. An implicit value is used if no explicit value is provided.

```
#!/usr/bin/tclsh
proc power {a {b 2}}
{
   if {$b == 2}
{
      return [expr $a * $a]
   }
   set value 1
   for {set i 0} {$i<$b} {incr i}
 {
      set value [expr $value * $a]
   }
   return $value
}
set v1 [power 5]
set v2 [power 5 4]
puts "5^2 is $v1"
puts "5^4 is $v2"
```

Here we create a power procedure. The procedure has one argument with an implicit value. We can call the procedure with one and two arguments.

```
proc power {a {b 2}} {
set v1 [power 5]
set v2 [power 5 4]
```

The second argument b, has an implicit value 2. If we provide only one argument, the power procedure then returns the value of a to the power 2.We call the power procedure with one and two arguments. The first line computes the value of 5 to the power 2. The second line value of 5 to the power 4. Output of the example.

```
$ ./implicit.tcl
5^2 is 25
5^4 is 625
```

Returning multiple values

The return command passes one value to the caller. There is often a need to return multiple values. In such cases, we can return a list.

```
#!/usr/bin/tclsh
proc tworandoms {}
{
   set r1 [expr round(rand()*10)]
   set r2 [expr round(rand()*10)]
   return [list $r1 $r2]
}
puts [two randoms]
puts [two randoms]
puts [two randoms]
puts [two randoms]
```

We have a two randoms procedure. It returns two random integers between 1 and 10. A random integer is computed and set to the r1 variable.

```
set r1 [expr round(rand()*10)]
return [list $r1 $r2]
```

Two values are returned with the help of the list command. A sample output.

```
$ ./tworandoms.tcl
```

3 7

1 3

8 7

9 9

Recursion

Recursion, in mathematics and computer science, is a way of defining functions in which the function being defined is applied within its own definition. In other words, a recursive function calls itself to do its job. Recursion is a widely used approach to solve many programming tasks. Recursion is the fundamental approach in functional languages like Scheme, OCalm, or Clojure. Recursion calls have a limit in Tcl. There cannot be more than 1000 recursion calls. A typical example of recursion is the calculation of a factorial. Factorial n! is the product of all positive integers less than or equal to n.

```
#!/usr/bin/tclsh
proc factorial n
 {
   if {$n==0}
            {
            return 1
            }
            else
            {
                  return [expr $n * [factorial [expr $n - 1]]]
             }
  }
# Stack limit between 800 and 1000 levels
puts [factorial 4]
puts [factorial 10]
puts [factorial 18]
```

In this code example, we calculate the factorial of three numbers.

```
return [expr $n * [factorial [expr $n - 1]]]
```

Inside the body of the factorial procedure, we call the factorial procedure with a modified argument. The procedure calls itself.

```
$ ./recursion.tcl
24
3628800
6402373705728000
```

These are the results. If we tried to compute the factorial of 100, we would receive "too many nested evaluations" error.

Scope

A variable declared inside a procedure has a procedure scope. The *scope* of a name is the region of a program text within which it is possible to refer to the entity declared by the name without qualification of the name. A variable which is declared inside a procedure has a procedure scope; it is also called a local scope. The variable is then valid only in this particular procedure.

```
#!/usr/bin/tclsh
proc test {}
 {
   puts "inside procedure"
   #puts "x is $x"
   set x 4
   puts "x is $x"
}
set x 1
puts "outside procedure"
puts "x is $x"
test
puts "outside procedure"
puts "x is $x"
```

In the preceding example, we have an x variable defined outside and inside of the test procedure. Inside the test procedure, we define an x variable. The variable has local scope, valid only inside this procedure.

```
set x 4
puts "x is $x"
set x 1
puts "outside procedure"
puts "x is $x"
```

We define an x variable outside the procedure. It has a global scope. The variables do not conflict because they have different scopes.

```
$ ./scope.tcl
outside procedure
x is 1
inside procedure
x is 4
outside procedure
x is 1
```

It is possible to change the global variable inside a procedure.

```
#!/usr/bin/tclsh
proc test {}
 {
   upvar x y
   puts "inside procedure"
   puts "y is $y"
   set y 4
   puts "y is $y"
}
set x 1
puts "outside procedure"
puts "x is $x"
test
puts "outside procedure"
puts "x is $x"
```

We define a global x variable. We change the variable inside the test procedure. We refer to the global x variable by the name y with the upvar command.

```
upvar x y
set y 4
```

We assign a value to the local y variable and also change the value of the global x variable.

```
$ ./scope2.tcl
outside procedure
x is 1
inside procedure
y is 1
y is 4
outside procedure
x is 4
```

From the output we can see the test procedure has changed the x variable. With the global command, we can refer to global variables from procedures.

```
#!/usr/bin/tclsh
proc test {}
{
   global x
   puts "inside test procedure x is $x"
   proc nested {}
 {
      global x
      puts "inside nested x is $x"
   }
}
set x 1
test
nested
```

```
puts "outside x is $x"
```

In the above example, we have a test procedure and a nested procedure defined within the test procedure. We refer to the global x variable from both procedures.

```
global x
puts "inside test procedure x is $x"
```

With the global command, we refer to the global x variable, defined outside the test procedure.

```
proc nested {}
{
   global x
   puts "inside nested x is $x"
}
```

It is possible to create nested procedures. These are procedures defined inside other procedures. We refer to the global x variable with the global command.

```
test
nested
```

We call the test procedure and its nested procedure.

```
$ ./scope3.tcl
inside test procedure x is 1
inside nested x is 1
outside x is 1
```

Strings patterns

Files

The file command manipulates file names and attributes. It has plenty of options.

```
#!/usr/bin/tclsh

puts [file volumes]
[file mkdir new]
```

The script prints the system's mounted values and creates a new directory. The file volumes command returns the absolute paths to the volumes mounted on the system.

```
puts [file volumes]
 [file mkdir new]
```

The file mkdir creates a directory called new.

```
$ ./voldir.tcl
/
$ ls -d */
doc/  new/  tmp/
```

On a Linux system, there is one mounted volume—the root directory. The ls command confirms the creation of the new directory.In the following code example, we are going to check if a file name is a regular file or a directory.

```
#!/usr/bin/tclsh
set files [glob *]
foreach fl $files
 {
   if {[file isfile $fl]}
 {
            puts "$fl is a file"
   }
elseif
 { [file isdirectory $fl]}
  {
    puts "$fl is a directory"
   }
}
```

We go through all file names in the current working directory and print whether it is a file or a directory.

Using the glob command we create a list of file and directory names of a current directory. We execute the body of the if command if the file name in question is a file.

```
set files [glob *]
if {[file isfile $fl]}
 {
} elseif { [file isdirectory $fl]} {
```

The file is directory command determines, whether a file name is a directory. Note that on Unix, a directory is a special case of a file. The puts command can be used to write to files.

```
#!/usr/bin/tclsh
set fp [open days w]
set days {Monday Tuesday Wednesday Thursday Friday Saturday Sunday}
puts $fp $days
close $fp
```

In the script, we open a file for writing. We write days of a week to a file. We open a file named days for writing. The open command returns a channel id. This data is going to be written to the file. We used the channel id returned by the open command to write to the file.

```
set fp [open days w]
set days {Monday Tuesday Wednesday Thursday Friday Saturday Sunday}
puts $fp $days
close $fp
```

We close the opened channel.

```
$ ./write2file.tcl
$ cat days
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

We run the script and check the contents of the days file. In the following script, we are going to read data from a file.

```
$ cat languages
Python
Tcl
Visual Basic
Perl
Java
C
C#
Ruby
Scheme
```

We have a simple file called languages in a directory.

```
#!/usr/bin/tclsh
set fp [open languages r]
set data [read $fp]
puts -nonewline $data
close $fp
```

We read data from the supplied file, read its contents and print the data to the terminal. We create a channel by opening the languages file in a read-only mode. If we do not provide a second parameter to the read command, it reads all data from the file until the end of the file.

```
set fp [open languages r]
set data [read $fp]
puts -nonewline $data
```

We print the data to the console.

```
$ ./readfile.tcl
Python
Tcl
Visual Basic
Perl
```

Java

C

C#

Ruby

Scheme

Sample run of the readfile.tcl script.

The eof command checks for end-of-line of a supplied channel.

```
#!/usr/bin/tclsh
set fp [open languages]
while {![eof $fp]}
 {
   puts [gets $fp]
}
close $fp
```

We use the eof command to read the contents of a file.

```
while {![eof $fp]}
 {
   puts [gets $fp]
}
```

The loop continues until the eof returns true if it encounters the end of a file. Inside the body, we use the gets command to read a line from the file.

$ ./readfile2.tcl

Python

Tcl

Visual Basic

Perl

Java

C

C#

Ruby

Scheme

Sample run of the readfile2.tcl script.The next script performs some additional file operations.

```
#!/usr/bin/tclsh
set fp [open newfile w]
puts $fp "this is new file"
flush $fp
file copy newfile newfile2
file delete newfile
close $fp
```

We open a file and write some text to it. The file is copied. The original file is then deleted. The file copy command copies a file.

```
file copy newfile newfile2
file delete newfile
```

The original file is deleted with the file delete command. In the final example, we will work with file attributes.

```
#!/usr/bin/tclsh
set files [glob *]
set mx 0
foreach fl $files
{
      set len [string length $fl]
      if { $len > $mx}
     {
            set mx $len
      }
}set fstr "%-$mx\s %-s"
```

```
puts [format $fstr Name Size]
```

```
set fstr "%-$mx\s %d bytes"
foreach fl $files
{
   set size [file size $fl]
   puts [format $fstr $fl $size]
}
```

The script creates two columns. In the first column, we have the name of the file. In the second column, we display the size of the file.

```
foreach fl $files
{
      set len [string length $fl]
      if { $len > $mx}
     {
            set mx $len
      }
}
```

In this loop, we find out the most lengthy file name. This will be used when formatting the output columns.

```
set fstr "%-$mx\s %-s"
puts [format $fstr Name Size]
```

Here we print the headers of the columns. To format the data, we use the format command.

```
set fstr "%-$mx\s %d bytes"
foreach fl $files
 {
```

```
    set size [file size $fl]
    puts [format $fstr $fl $size]
}
```

We go through the list of files and print each file name and its size. The file size command determines the size of the file.

```
$ ./attributes.tcl
Name          Size
attributes.tcl 337 bytes
newfile2       17 bytes
allfiles.tcl   75 bytes
printing.tcl   83 bytes
languages      51 bytes
hello.tcl      109 bytes
days           57 bytes
read.tcl       113 bytes
files.tcl      140 bytes
globcmd.tcl    82 bytes
write2file.tcl 134 bytes
doc            4096 bytes
cwd.tcl        76 bytes
tmp            4096 bytes
readfile.tcl   98 bytes
isfile.tcl     219 byte
```

Eval

One difference between Tcl and most other compilers is that Tcl will allow an executing program to create new commands and execute them while running. A tcl command is defined as a list of strings in which the first string is a command or proc. Any string or list which meets this criteria can be evaluated and executed. The **eval** command will evaluate a list of strings as though they were commands typed at the **%** prompt or sourced from a file. The **eval** command normally returns the final value of the commands being evaluated. If the

commands being evaluated throw an error (for example, if there is a syntax error in one of the strings), then **eval** will will throw an error.Note that either **concat** or **list** may be used to create the command string, but that these two commands will create slightly different command strings.

eval ***arg1 ??arg2??? ... ??argn??***

Evaluates *arg1 - argn* as one or more Tcl commands. The *args* are concatenated into a string, and passed to **tcl_Eval** to evaluate and execute.

Eval returns the result (or error code) of that evaluation.

Example

```
set cmd {puts "Evaluating a puts"}

puts "CMD IS: $cmd"

eval $cmd

if {[string match [info procs newProcA] ""] } {

   puts "\nDefining newProcA for this invocation"

   set num 0;

   set cmd "proc newProcA "

   set cmd [concat $cmd "{} {\n"]

   set cmd [concat $cmd "global num;\n"]

   set cmd [concat $cmd "incr num;\n"]

   set cmd [concat $cmd " return \"/tmp/TMP.[pid].\$num\";\n"]

   set cmd [concat $cmd "}"]

   eval  $cmd

}

puts "\nThe body of newProcA is: \n[info body newProcA]\n"

puts "newProcA returns: [newProcA]"
```

```
puts "newProcA returns: [newProcA]"

#

# Define a proc using lists

#

if {[string match [info procs newProcB] ""] } {

    puts "\nDefining newProcB for this invocation"

    set cmd "proc newProcB "

    lappend cmd {}

    lappend cmd {global num; incr num; return $num;}

    eval  $cmd

}

puts "\nThe body of newProcB is: \n[info body newProcB]\n"

puts "newProcB returns: [newProcB]"
```

Source

This command takes the contents of the specified file or resource and passes it to the Tcl interpreter as a text script. The return value from source is the return value of the last command executed in the script. If an error occurs in evaluating the contents of the script then the source command will return that error. If a return command is invoked from within the script then the remainder of the file will be skipped and the source command will return normally with the result from the return command.

The -rsrc and -rsrcid forms of this command are only available on Macintosh computers. These versions of the command allow you to source a script from a TEXT resource. You may specify what TEXT resource to source by either name or id. By default Tcl searches

all open resource files, which include the current application and any loaded C extensions. Alternatively, you may specify the fileName where the TEXT resource can be found.

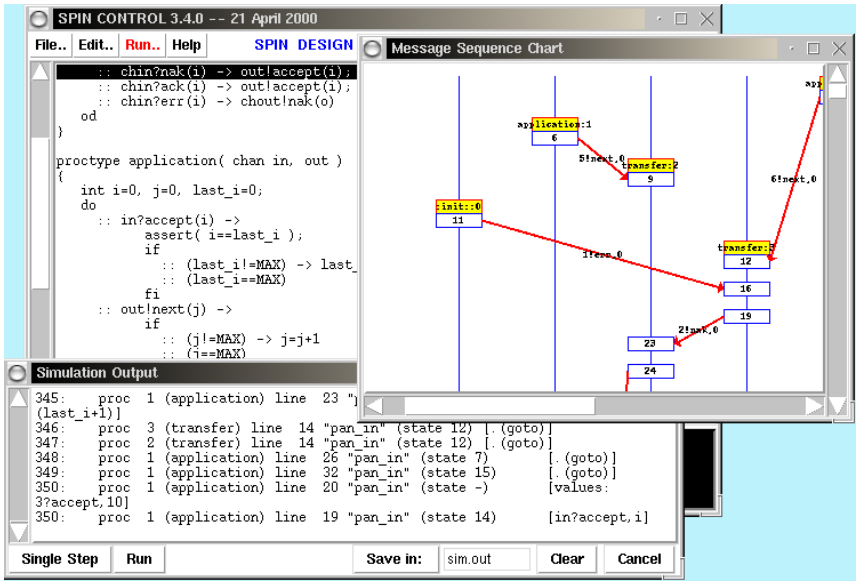
source fileName

source -rsrc resourceName ?fileName?

source -rsrcid resourceId ?fileName?

Tk

Wish - the windowing shell, is a simple scripting interface to the Tcl/Tk language. The language Tcl (Tool Command Language) is an interpreted scripting language, with useful inter-application communication methOds, and is pronounced 'tickle'. Tk originally was an X-window toolkit implemented as extensions to 'tcl'. However, now it is available native on all platforms.The program xspin is an example of a portable program in which the entire user interface is written in wish. The program also runs on PCs using NT or Win95, and as well on Macintoshes.



A first use of wish could be the following

manu> wish

wish> button .quit -text "Hello World!" -command {exit}

.quit

wish> pack .quit wish>

You can encapsulate this in a script:

CODE LISTING **HelloWorld.tcl**

```
#!/usr/local/bin/wish8.1 -f

button .quit -text "Hello World!" -command {exit}
pack .quit
```

If you create this as a file, and make it executable, you should be able to run this simple graphical program.

Structure of Tcl/Tk

The Tcl language has a tiny syntax - there is only a single command structure, and a set of rules to determine how to interpret the commands. Other languages have special syntaxes for control structures (if, while, repeat...) - not so in Tcl. All such structures are implemented as commands.

There is a runtime library of compiled 'C' routines, and the 'level' of the GUI interface is quite high.

Comments: If the first character of a command is #, it is a comment

Tcl commands: Tcl commands are just words separated by spaces. Commands return strings, and arguments are just further words.

command argument argument

command argument

Spaces are important

expr 5*3 has a single

argument expr 5 * 3 has three

arguments

Tcl commands are separated by a new line, or a semicolon, and arrays are indexed by text

set a(a\ text\ index) 4

Tcl/Tk quoting rules

The "quoting" rules come in to play when the " or { character are first in the word. ".." disables a few of the special characters - for example space, tab, newline and semicolon, and {..} disables everything except \{, \} and \nl. This facility is particularly useful for the control structures - they end up looking very like 'C'

```
While
 {a==10}
{ set b [tst a]
}
```

Tcl/Tk substitution rules

Variable substitution: The dollar sign performs the variable value substitution. Tcl variables are strings.

```
set a 12b          a will be "12b"
set b 12$a         b will be
"1212b"
```

Command substitution: The []'s are replaced by the value returned by executing the Tcl com- mand 'doit'.

```
set a [doit param1 param2]
```

Backslash substitution:

```
set a a\ string\ with\ spaces\ \ and\ a\
new\ line
```

Tcl/Tk command examples:

Procedures	File Access	Miscellaneous
`proc name {parameters} {body}`	`open <name>`	`source <NameOfFile>`
	`read <fileID>`	`global <varname>`
	`close <fileID>`	`catch <command>`
	`cd <directoryname>`	`format <formatstring> <value>`
		`exec <process>`
		`return <value>`

List operators

split <string> ?splitcharacters?

concat <list> <list>

lindex <list> <index>

... + lots more

Control structures

if {test} {thenpart} {elsepart}1while {test} {body} for {init}

{test} {incr} {body}

continue

[redacted]

% canvas <name> - optional parameter pairs ...

% button <name> - optional parameter pairs ...

% frame <name> - optional parameter pairs ...

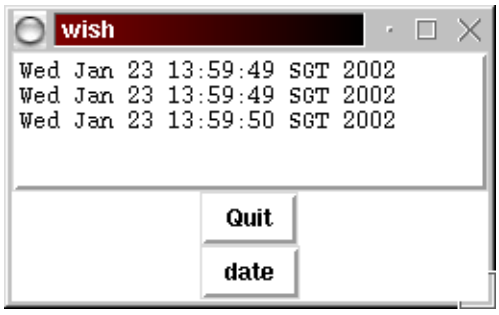
% ... and so on

When you create a widget ".b", a new command ".b" is created, which you can use to further communicate with it. The geometry managers in Tk assemble the widgets

% pack <name> .... where ....

Tcl/Tk example software

Here is a very small Tcl/Tk application, which displays the date in a scrollable window:



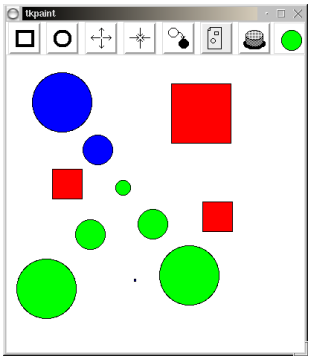
The code for this is

CODE LISTING **SimpleProg.tcl**

```
#!/usr/local/bin/wish8.1 -f

text .log -width 60 -height 5 -bd 2 -relief raised
pack .log
button .buttonquit -text "Quit" -command exit
pack .buttonquit
button .buttondate -text "date" -command
getdate pack .buttondate
proc getdate {} {
   set result [exec date]
   .log insert end $result
   .log insert end \n
}
```

Here is tkpaint - a drawing/painting program written in Tcl/Tk



The mainline of the source just creates the buttons, and packs the frame: Routines for dragging,

CODE LISTING **tkpaint1.tcl**

```
#! /usr/local/bin/wish -f

set thistool rectangle
set thisop grow
set thiscolour black
button .exitbtn   -bitmap @exit.xbm
                                    -command exit
button .squarebtn -bitmap @square.xbm    -command setsquaretool
button .circlebtn -bitmap @circle.xbm    -command setcircletool
button .shrnkbtn  -bitmap @shrink.xbm    -command "set thisop shrnk"
button .growbtn   -bitmap @grow.xbm      -command "set thisop grow"
button .printbtn  -bitmap @print.xbm     -command printit
button .colorbtn -bitmap @newcolour.xbm -command setanewcolour
canvas .net -width 400 -height 400 -background white -relief sunken
canvas .status -width 40 -height 40 -background white -relief sunken

pack .net -side bottom
pack .status -side right
pack .squarebtn .circlebtn -side left -ipadx 1m -ipady 1m -expand 1
pack .exitbtn .printbtn -side right -ipadx 1m -ipady 1m -expand 1
pack .colorbtn .shrnkbtn .growbtn -side right -ipadx 1m -ipady 1m -expand 1

bind .net <ButtonPress-1> {makenode %x %y}
.status create rectangle 10 10 37 37 -tag statusthingy -fill $thiscolour
set nodes 0; set oldx 0; set oldy 0;
```

scaling and printing

CODE LISTING **tkpaint4.tcl**

```
proc beginmove {x y} {
    global oldx oldy
    set oldx $x; set oldy $y
}

proc domove {item x y} {
    global oldx oldy
    .net move $item [expr "$x - $oldx"] [expr "$y - $oldy"]
    set oldx $x; set oldy $y
}

proc altersize {item x y z} {
    .net scale $item $x $y $z $z
}

proc printit {} {
    .net postscript -file "pic.ps"
}
```

Node operations for tkpaint

CODE LISTING **tkpaint2.tcl**

```
proc makenode {x y} {
   global nodes oldx oldy thistool thiscolor
   set nodes [expr "$nodes+1"]

   set x1 [expr "$x-20"]; set y1 [expr "$y-20"]
   set x2 [expr "$x+20"]; set y2 [expr "$y+20"] if
   {[string compare $thistool "oval"] == 0} {
      .net create oval $x1 $y1 $x2 $y2 -tag node$nodes -fill $thiscolor
   }
   if {[string compare $thistool "rectangle"] == 0} {
      .net create rectangle $x1 $y1 $x2 $y2 -tag node$nodes -fill $thiscolor
   }
   .net bind node$nodes <Enter> ".net itemconfigure node$nodes -width 5"
   .net bind node$nodes <Leave> ".net itemconfigure node$nodes -width 1"
   .net bind node$nodes <ButtonPress-3> "beginmove %x %y"
   .net bind node$nodes <B3-Motion> "domove node$nodes %x %y"
   .net bind node$nodes <ButtonPress-2> "dothisop node$nodes %x %y"
}

proc dothisop {item x y} {
   global thisop
   if {[string compare $thisop "shrink"] == 0} {
       altersize $item $x $y 0.5
   }
   if {[string compare $thisop "grow"] == 0} {
       altersize $item $x $y 2.0
   }
}
```

More routines

CODE LISTING **tkpaint3.tcl**

```
proc setcircletool {} {
    global thistool thiscolor
    set thistool oval
    .status delete statusthingy
    .status create oval 10 10 37 37 -tag statusthingy -fill $thiscolor
}

proc setsquaretool {} {
    global thistool thiscolor
    set thistool rectangle
    .status delete statusthingy
    .status create rectangle 10 10 37 37 -tag statusthingy -fill $thiscolor
}

proc setanewcolor {} {
    global thiscolor

    if {[string compare $thiscolor "black"] == 0} {
            set thiscolor green
    } { if {[string compare $thiscolor "green"] == 0} {
            set thiscolor blue
        } { if {[string compare $thiscolor "blue"] == 0} {
                set thiscolor red
            } { if {[string compare $thiscolor "red"] == 0} {
                    set thiscolor orange
                } { set thiscolor black }
              }
          }
       }
    .status itemconfigure statusthingy -fill $thiscolor
}
```

C/Tk

In the following example, a Tcl/Tk program is integrated with a C program, giving a very small codesize GUI application, that can be compiled on any platform - Windows, UNIX or even the Macintosh platform without changes.

CODE LISTING **CplusTclTk.c**

```
#include <stdio.h>
#include <tcl.h>
#include <tk.h>
char
    tclprog[
    ] = "\
    proc
    fileDial
    og {w}
    {\
     set types {\
```

```
   { \"Image
     files\"
    {.gif} }\
   { \"All files\" *}\
  };\
  set file [tk_getOpenFile
  −filetypes $types −parent
  $w];\ image create photo
  picture −file $file;\
  set glb_tx
  [image width
  picture];\ set
  glb_ty [image
  height
  picture];\
  .c configure −width $glb_tx −height $glb_ty;\
  .c create image 1 1 −anchor nw −image picture −tags \"myimage\";\
};\
frame .mbar
−relief raised −bd
2;\ frame .dummy
−width 10c
−height 0;\ pack
.mbar .dummy
−side top −fill x;\
menubutton .mbar.file −text File
−underline 0 −menu .mbar.file.menu;\
menu .mbar.file.menu −tearoff 1;\
.mbar.file.menu add command −label \"Open...\" −command
\"fileDialog .\";\
.mbar.file.menu add separator;\
.mbar.file.menu add command −label
\"Quit\" −command \"destroy .\";\ pack
```

```
        .mbar.file  −side left;\
        canvas .c −bd 2 −relief raised;\
        pack .c −side top
        −expand yes −fill
        x;\ bind .
        <Control−c>
        {destroy .};\
        bind .
        <Control−q>
        {destroy .};\
        focus .mbar";
int
main
        (argc
        ,
        argv)
        int
        argc;
        char
        **ar
        gv;
{
    Tk_Window
    mainWindow;
    Tcl_Interp
    *tcl_interp;

    setenv ("TCL_LIBRARY",
    "/cygnus/cygwin−b20/share/tcl8.0");
    tcl_interp = Tcl_CreateInterp ();
    if (Tcl_Init (tcl_interp) != TCL_OK || Tk_Init (tcl_interp) != TCL_OK)
    {
        if (*tcl_interp−>result)
```

```
            (void) fprintf (stderr, "%s: %s\n", argv[0],
        tcl_interp−>result); exit (1);
    }
    mainWindow = Tk_MainWindow (tcl_interp);
    if (mainWindow == NULL) {
        fprintf (stderr, "%s\n",
        tcl_interp−>result); exit (1);
    }
    Tcl_Eval (tcl_interp,
    tclprog);
    Tk_MainLoop ();
    exit (1);
}
```

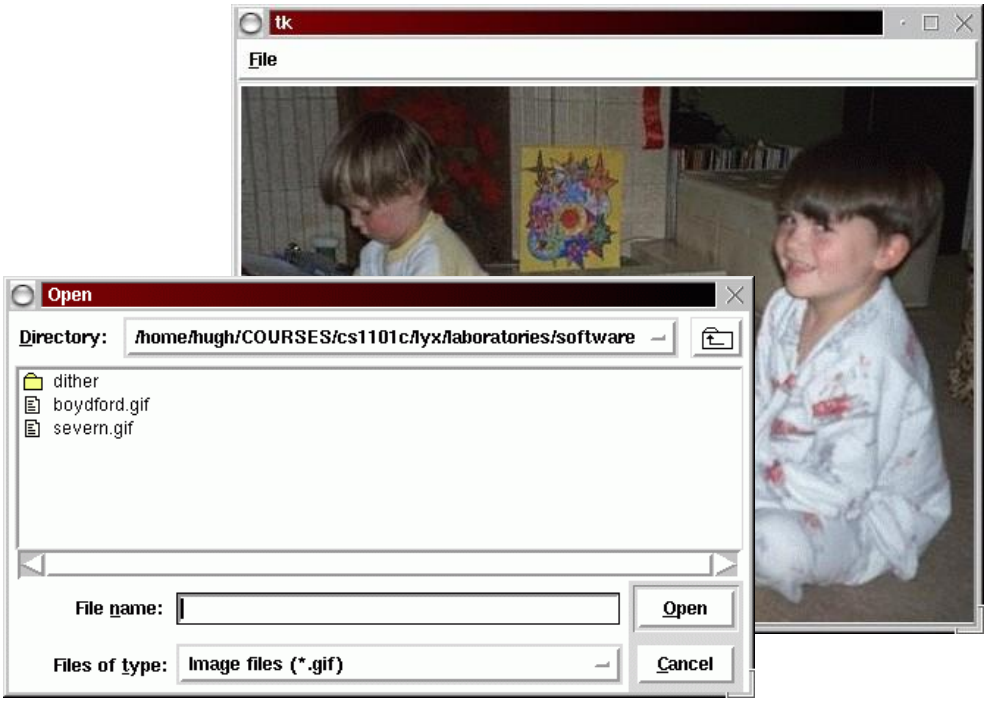
The first half of the listing is a C string containing a Tcl/Tk program. The second part of the listing is C code which uses this Tcl/Tk.On a Win32 system, we compile this as:

```
gcc -o CplusTclTk CplusTclTk.c -mwindows -ltcl80 -ltk80
```

On a UNIX system we use:

```
gcc -o CplusTclTk CplusTclTk.c -ltk -ltcl -lX11 -lm -ldl
```

And the result is a simple viewer for GIF images. The total code size is 57 lines. The application looks like this when running:



This section includes some extra material related to the use of Tcl/Tk for developing GUI appli- cations. In particular - constructing menu items, using the Tk Canvas and structured data items. There are pointers to some supplied reference material. Note the following points related to trying out Tcl/Tk:If you are using cygwin-b20, the wish interpreter is called cygwish80.exe. This file is found in the directory /cygnus/cygwin-b20/H-i586-cygwin32/cygwish80.exe. Make a copy of this file in the same directory, and call it wish8.0.exe for compatibility with UNIX Tcl/Tk scripts.In the first line of your tcl files, you should put #!wish8.0If you download the file ~cs3283/ftp/demos.tar and extract it into /cygnus, you will have a series of Tcl/Tk widget examples in /cygnus/Demos. Change into the directory

/cygnus/Demos, and type ./widget.

There is a Tcl/Tk tutor, and many learn-to-program-Tcl/Tk documents available at many sites on the Internet - if you continue to have trouble, you may wish to try them.

There is no substitute for just trying to program - set yourself a small goal, and discover how to do it in Tcl/Tk.

Tcl/Tk menus

The menu strategy is fairly simple

1. Make up a frame for the menu
2. Add in the top level menu items
3. For each top level item, add in the drop-menu items
4. For each nested item, add in any cascaded menus.
5. Remember to pack it...

As an example, the following code creates a fairly conventional application with menus, a help dialog, and cascaded menu items.

CODE LISTING **Menus.tcl**

```
#!/usr/bin/wish

frame .mbar -relief raised -bd 2
pack .mbar -side top -fill x

frame .dummy -width 10c -height 100
pack .dummy

menubutton .mbar.file -text File -underline 0 -menu .mbar.file.menu
menu .mbar.file.menu -tearoff 0
.mbar.file.menu add command -label "New..." -command "newcommand"
.mbar.file.menu add command -label "Open..." -command "opencommand"
.mbar.file.menu add separator
.mbar.file.menu add command -label Quit -command exit
pack .mbar.file  -side left

menubutton .mbar.edit -text Edit -underline 0 -menu .mbar.edit.menu
menu .mbar.edit.menu -tearoff 1
.mbar.edit.menu add command -label "Undo..." -command "undocommand"
.mbar.edit.menu add separator
.mbar.edit.menu add cascade -label Preferences -menu .mbar.edit.menu.prefs
menu .mbar.edit.menu.prefs -tearoff 0
.mbar.edit.menu.prefs add command -label "Load default" -command "defaultprefs"
.mbar.edit.menu.prefs add command -label "Revert" -command
"revertprefs" pack .mbar.edit  -side left

menubutton .mbar.help -text Help -underline 0 -menu .mbar.help.menu
menu .mbar.help.menu -tearoff 0
.mbar.help.menu add command -label "About ThisApp..." -command "aboutcommand"
pack .mbar.help  -side right

proc aboutcommand {} {
    tk_dialog .win {About this program} "Hugh wrote it!"  {} 0 OK
}
```

Tk canvas

The Tk canvas widget allows you to draw items on a pane of the application. Items may be tagged when created, and then these tagged items may be bound to events, which may be used to manipulate the items at a later stage.This process is described in detail in Robert Biddle's "Using the Tk Canvas Facility", a copy of which is found at ~cs3283/ftp/CS-TR-94-5.pdf.Note also the use of dynamically created variable names (node$nodes).

UNIT-5

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 Guido van Rossum, and it is free software. Free as in "free beer," in that you can obtain Python without spending any money. But Python is also free in other important ways, for example you are free to copy it as many times as you like, and free to study the source code, and make changes to it. There is a worldwide movement behind the idea of free software, initiated in 1983 by Richard Stallman.1This document focuses on learning Python for the purpose of doing mathematical calculations. We assume the reader has some knowledge of basic mathematics, but we try not to assume any previous exposure to computer programming, although some such exposure would certainly be helpful. Python is a good choice for mathematical calculations, since we can write code quickly, test it easily, and its syntax is similar to the way mathematical ideas are expressed in the mathematical literature. By learning Python you will also be learning a major tool used by many web developers. Installation and documentation If you use Mac OS X or Linux, then Python should already be installed on your computer by default. If not, you can download the latest version by visiting the Python home page, at

http://www.python.org

where you will also find loads of documentation and other useful information. Windows users can also download Python at this website. Don't forget this website; it is your first point of reference for all things Python. You will find there, for example, reference [1], the excellent Python Tutorial by Guido van Rossum. You may find it useful to read along in the Tutorial as a supplement to this document.

2 Running Python as a calculator

The easiest way to get started is to run Python as an interpreter, which behaves similar to the way one would use a calculator. In the interpreter, you type a command, and Python produces the

answer. Then you type another command, which again produces an answer, and so on.In OS X or Linux, to start the Python interpreter is as simple as typing the command python on the command line in a terminal shell. In Windows, assuming that Python has already been installed, you need to find Python in the appropriate menu. Windows users may choose to run Python in a command shell (i.e., a DOS window) where it will behave very similarly to Linux or OS X. For all three operating systems (Linux, OS X, Windows) there is also an integrated development environment for Python named IDLE.

Once Python starts running in interpreter mode, using IDLE or a command shell, it produces a prompt, which waits for your input. For example, this is what I get when I start Python in a command shell on my Linux box:

```
doty@ brauer:~ % python
Python 2.5.2 ( r252 :60911 , Apr 21 2008 , 11 :12 :42 )
[ GCC 4.2.3 ( Ubuntu 4.2.3 -2 ubuntu 7 )] on linux2
Type " help", " copyright", " credits" or " license" for more information.
```

>>> where the three symbols >>> indicates the prompt awaiting my input.

So experiment, using the Python interpreter as a calculator. Be assured that you cannot harm anything, so play with Python as much as you like. For example:

```
>>> 2*1024 2048
>>> 3+ 4+ 9
16
>>> 2**100
1267650600228229401496703205376 L
```

In the above, we first asked for the product of 2 and 1024, then we asked for the sum of 3, 4, and 9 and finally we asked for the value of 2100. Note that multiplication in Python is represented by ∗ , addition by +, and exponents by **; you will need to remember this syntax. The L appended to the last answer is there to indicate that this is a long integer; more on this later. It is also worth noting that Python does arbitrary precision integer arithmetic, by default

```
>>> 2 ** 1000 10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251
8714528569231404359845775746985748039345677748242309854210746
```

0506237114187795418215304647498358194126739876755916554394607

706291457119647768654216766042983165262438683720566806937 6L

Here is another example, where we print a table of perfect squares:

```
>>> for n in [1 ,2 ,3 ,4 ,5 ,6]:
...   print n**2
.
.
.

1
4
9
16
25
36
```

2.Both Python and IDLE should be already preinstalled on all Loyola Windows computers. This illustrates several points. First, the expression [1,2,3,4,5,6] is a list, and we print the values of n2 for n varying over the list. If we prefer, we can print horizontally instead of vertically:

```
>>> for n in [1 ,2 ,3 ,4 ,5 ,6]:
...   print n**2 ,
...
1   4   9   16  25 36
```

Simply by adding a comma at the end of the print command, which tells Python not to move to a new line before the next print. These last two examples are examples of a compound command, where the command is divided over two lines (or more). That is why you see ... on the second line instead of the usual >>>,which is the interpreter's way of telling us it awaits the rest of the command. On the third line we entered nothing, in order to tell the interpreter that the command was complete at the second line. Also notice the colon at the end of the first line, and the indentation in the second line. Both are required in compound Python commands.

Quitting the interpreter: In a terminal you can quit a Python session by CTRL-D. (Hold down the CTRL key while pressing the D key.) In IDLE you can also quit from the menu. If the interpreter

gets stuck in an infinite loop, you can quit the current execution by CTRL-C. Loading commands from the library Python has a very extensive library of commands, documented in the Python Library Reference Manual [2]. These commands are organized into modules. One of the available modules is especially useful for us: the math module. Let's see how it may be used.

```
>>> from math import sqrt , exp
>>> exp(-1)
0 .36787944117144233
>>> sqrt(2)
1 .4142135623730951
```

We first import the sqrt and exp functions from the math module, then use them to compute
$e^{-1} = 1/e$ and $\sqrt{2}$.

Once we have loaded a function from a module, it is available for the rest of that session. When we start a new session, we have to reload the function if we need it. Note that we could have loaded both functions sqrt and exp by using a wildcard *:>>> from math import * which tells Python to import all the functions in the math module. What would have happened if we forgot to import a needed function? After starting a new session, if we type

```
>>> sqrt(2)
Traceback ( most recent call last): File "< stdin >", line 1 , in < module > N am eError: name ' sqrt' is not defined we see an example of an error message, telling us that Python does not recognize sqrt.
```

Defining functions

It is possible, and very useful, to define our own functions in Python. Generally speaking, if you need to do a calculation only once, then use the interpreter. But when you or others have need to perform a certain type of calculation many times, then define a function. For a simple example, the compound command

```
>>> def f(x):
... return x* x
...
```

defines the squaring function f (x) = x2, a popular example used in elementary math courses. In the definition, the first line is the function header where the name, f, of the function is specified. Subsequent lines give the body of the function, where the output value is calculated. Note that the final step is to return the answer; without it we would never see any results. Continuing the example, we can use the function to calculate the square of any given input:

```
>>> f(2) 4
>>> f(2.5)
6.25
```

The name of a function is purely arbitrary. We could have defined the same function as above, but with the name square instead of f; then to use it we use the new function name instead of the old:

```
>>>  def square(x):
...   return  x* x
...
>>>  square (3)
9
>>> square (2.5)
6.25
```

Actually, a function name is not completely arbitrary, since we are not allowed to use a reserved word as a function name. Python's reserved words are: and, def, del, for, is, raise, assert, elif, from, lambda, return, break, else, global, not, try, class, except, if, or, while, continue, exec, import, pass, yield.

By the way, Python also allows us to define functions using a format similar to the Lambda Calculus in mathematical logic. For instance, the above function could alternatively be defined in the following way:

```
>>>  square  =  lambda  x:  x* x
```

Here lambda x: x*x is known as a lambda expression. Lambda expressions are useful when you need to define a function in just one line; they are also useful in situations where you need a function but don't want to name it.

Usually function definitions will be stored in a module (file) for later use. These are

indistinguishable from Python's Library modules from the user's perspective.

Files

Python allows us to store our code in files (also called modules). This is very useful for more serious programming, where we do not want to retype a long function definition from the very beginning just to change one mistake. In doing this, we are essentially defining our own modules, just like the modules defined already in the Python library. For example, to store our squaring function example in a file, we can use any text editor3 to type the code into a file, such as

```
def
square(x)
: return
x* x
```

Notice that we omit the prompt symbols >>>, ... when typing the code into a file, but the indentation is still important. Let's save this file under the name "SquaringFunction.py" and then open a terminal in order to run it:

```
doty@ brauer:~ % python
Python 2.5.2 ( r252 :60911 , Apr 21 2008 , 11 :12 :42 )
[ GCC 4.2.3 ( Ubuntu 4.2.3 -2 ubuntu 7 )] on
linux2 Type " help", " copyright", " credits" or "
license" for more inform ation.
>>> from Squaring Function import square
>>> square (1.5)
2.25
```

Notice that I had to import the function from the file before I could use it. Importing a command from a file works exactly the same as for library modules. (In fact, some people refer to Python files as "modules" because of this analogy.) Also notice that the file's extension (.py) is omitted in the import command.

Testing code

As indicated above, code is usually developed in a file using an editor. To test the code, import it

into a Python session and try to run it. Usually there is an error, so you go back to the file, make a correction, and test again. This process is repeated until you are satisfied that the code works. The entire process is known as the development cycle. There are two types of errors that you will encounter. Syntax errors occur when the form of some command is invalid. This happens when you make typing errors such as misspellings, or call something by the wrong name, and for many other reasons. Python will always give an error message for a syntax error.

Scripts

If you use Mac OS X or some other variant of Unix (such as Linux) then you may be interested

---

in running Python commands as a script. Here's an example. Use an editor to create a file name SayHi containing the following lines

```
#! / usr/ bin/ python
print " Hello World!"
print "- From your friendly Python program"
```

3.Most developers rely on emacs for editing code. Other possible choices are Notepad for Windows, gedit for Linux/Gnome, and TextEdit for OS X. IDLE comes with its own editor, by the way.The first line tells Python that this is a script. After saving the file, make it executable by typing chmod 755 SayHi in the terminal. To run the script, type ./SayHi in the terminal. Note that if you move the script someplace in your search path, then you can run it simply by typing SayHi. Type echo $PATH to see what folders are in your search path, and type which python to see where your python program is — this should match the first line in your script. As far as I know, it is impossible to run Python scripts in a similar way on a Windows machine.

Python commands

Comments

In a Python command, anything after a # symbol is a comment. For example: print " Hello world" # this is silly Comments are not part of the command, but rather intended as documentation for anyone reading the code. Multiline comments are also possible, and are enclosed by triple double-quote symbols:""" This is an example of a long comment that goes on and on and on.""" Numbers and other data types

Python recognizes several different types of data. For instance, 23 and −75 are integers, while 5.0 and −23.09 are floats or floating point numbers. The type float is (roughly) the same as a real number in mathematics. The number 12345678901 is a long integer ; Python prints it with an “L” appended to the end. Usually the type of a piece of data is determined implicitly.
The type function

To see the type of some data, use Python’s built-in type function:

>>> type(-75)
< type ’ int’ >
>>> type(5.0)
< type ’ float’ >
>>> type (12345678901)
< type ’ long’ >
Another useful data type is complex, used for complex numbers. For example:

>>> 2 j
2 j
>>> 2j-1 (-1+2 j)
>>> com plex(2 ,3)
(2+ 3 j)
>>> type(-1+2 j)
< type ’ com plex’ >
Notice that Python uses j for the complex unit (such that j2 = −1) just as physicists do, instead of the letter i preferred by mathematicians.

Strings

Other useful data types are strings (short for “character strings”); for example "Hello World!". Strings are sequences of characters enclosed in single or double quotes

```
>>> " This is a string " ’ This is a string ’
>>> ’ This is a string , too’ ’ This is a string , too’
>>> type(" This is a string ")
< type ’ str’ >
```

Strings are an example of a sequence type.

Lists and Tuples

Other important sequence types used in Python include lists and tuples. A sequence type is formed by putting together some other types in a sequence. Here is how we form lists and tuples:

```
>>> [1 ,3 ,4 ,1 ,6]
[1 , 3 , 4 , 1 , 6]
>>> type( [1 ,3 ,4 ,1 ,6] )
< type ’ list’ >
>>> (1 ,3 ,2)
(1 , 3 , 2)
>>> type( (1 ,3 ,2) )
< type ’ tuple’ >
```

Notice that lists are enclosed in square brackets while tuples are enclosed in parentheses. Also note that lists and tuples do not need to be homogeneous; that is, the components can be of different types:

```
>>> [1 ,2 ," Hello" ,(1 ,2)]
[1 , 2 , ’ Hello ’ , (1 , 2)]
```

Here we created a list containing four components: two integers, a string, and a tuple. Note that components of lists may be other lists, and so on:

```
>>> [1 , 2 , [1 ,2], [1 ,[1 ,2]], 5]
[1 , 2 , [1 , 2], [1 , [1 , 2]], 5]
```

By nesting lists within lists in this way, we can build up complicated structures.

Sequence types such as lists, tuples, and strings are always ordered, as opposed to a set in mathematics, which is always unordered. Also, repetition is allowed in a sequence, but not in a set.

Range function

The range function is often used to create lists of integers. It has three forms. In the simplest form, range(n) produces a list of all numbers 0, 1, 2, . . . , n − 1 starting with 0 and ending with n − 1. For instance,

```
>>>  range (17)
[0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 , 16]
```

You can also specify an optional starting point and an increment, which may be negative. For instance, we have

```
>> range(1 ,10)
[1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]
>>> range(-6 ,0)
[-6 , -5 , -4 , -3 , -2 , -1]
>>> range (1 ,10 ,2)
[1 , 3 , 5 , 7 , 9]
>>> range(10 ,0 ,-2)
[10 , 8 , 6 , 4 , 2]
```

Note the use of a negative increment in the last example.

Boolean values

Finally, we should mention the Boolean type. This is a value which is either True or False.

```
>>> True True
>>> type( True)
< type   ' bool' >
>>>   False False
>>>   type( False)
< type   ' bool' >
```

Boolean types are used in making decisions.

Expressions

Python expressions are not commands, but rather form part of a command. An expression is anything which produces a value. Examples of expressions are: 2+2, 2**100, f((x-1)/(x+1)). Note that in order for Python to make sense of the last one, the variable x must have a value assigned and f should be a previously defined function. Expressions are formed from variables, constants, function evaluations, and operators. Parentheses are used to indicate order of operations and grouping, as usual.

Operators

The common binary operators for arithmetic are + for addition, - for subtraction, * for multiplication, and / for division. As already mentioned, Python uses ** for exponentiation. Integer division is performed so that the result is always another integer (the integer quotient):

```
>>> 25/3 8
>>> 5/2
2
```

This is a wrinkle that you will always have to keep in mind when working with Python. To get a more accurate answer, use the float type:

```
>>>  25.0/3 8 3333333333333339
>>> 5/2.0 2.5
```

If just one of the operands is of type float, then the result will be of type float. Here is another example of this pitfall:

```
>>> 2 **(1 /2 )
1
>>>  2**0.5 1
.4142135623730951
```

Another useful operator is %, which is read as ”mod”. This gives the remainder of an integer division, as in

```
>>>  5 %  2
1
>>>  25 %  3
1
```

which shows that 5 mod 2 = 1, and 25 mod 3 = 1. This operator is useful in number theory and cryptography. Besides the arithmetic operators we need comparison operators: <, >, <=, >=, ==, !=, <>. In order these are read as: is less than, is greater than, is less than or equal to, is greater than or equal to, is equal to, is not equal to, is not equal to. The result of a comparison is always a Boolean value

True or False.

```
>>> 2 < 3
True
>>> 3 <2
False
>>> 3 <= 2
False
```

Note that != and <> are synonymous; either one means not equal to. Also, the operator == means is equal to.

```
>>> 2 <> 3
True
      != 3
True
>>> 0 != 0
False
>>> 0 == 0
True
```

Variables and assignment

An assignment statement in Python has the form variable = expression. This has the following effect. First the expression on the right hand side is evaluated, then the result is assigned to the variable. After the assignment, the variable becomes a name for the result. The variable retains the same value until another value is assigned, in which case the previous value is lost. Executing the assignment produces no output; its purpose it to make the association between the variable and its value.

```
>>> x = 2+ 2
>>> print x 4
```

In the example above, the assignment statement sets x to 4, producing no output. If we want to see the value of x, we must print it. If we execute another assignment to x, then the previous value is

lost.

```
>>> x = 380.5
>>> print
x 380.5
>>> y = 2* x
>>> print
y 761.0
```

Remember: A single = is used for assignment, the double == is used to test for equality. In mathematics the equation x = x + 1 is nonsense; it has no solution. In computer science, the statement x = x + 1 is useful. Its purpose is to add 1 to x, and reassign the result to x. In short, x is incremented by 1.

```
>>> x = 10
>>> x = x + 1
>>>
print
x 11
>>> x = x + 1
>>>
print
x 12
```

Variable names may be any contiguous sequence of letters, numbers, and the underscore (_) character. The first character must not be a number, and you may not use a reserved word as a variable name. Case is important; for instance Sum is a different name than sum. Other examples of legal variable names are: a, v1, v_1, abc, Bucket, monthly_total, pi , TotalAssets.

Decisions

The if–else is used to make choices in Python code. This is a compound statement. The simplest form is

```
if c o n d i t i o n :
  a c t i o n −1
```

```
else:
    a c t i o n −2
```

The indentation is required. Note that the else and its action are optional. The actions action-1 and action-2 may consist of many statements; they must all be indented the same amount. The condition is an expression which evaluates to True or False.

Of course, if the condition evaluates to True then action-1 is executed, otherwise action-2 is executed. In either case execution continues with the statement after the if-else. For example, the code

```
x = 1
if x > 0:
print " Friday is wonderful"
else:
print " Monday sucks"
print " Have a good weekend"
```

results in the output

```
Friday is
wonderful Have a
good weekend
```

Note that the last print statement is not part of the if-else statement (because it isn't indented), so if we change the first line to say x = 0 then the output would be

```
Monday sucks
Have a good weekend
```

More complex decisions may have several alternatives depending on several conditions. For these the elif is used. It means "else if" and one can have any number of elif clauses between the if and the else. The usage of elif is best illustrated by an example:

```
if x >= 0 and x < 10: digits = 1
elif x >= 10 and x < 100: digits = 2
elif x >= 100 and x < 1000:digits = 3
elif x >= 1000 and x < 10000: digits = 4
else:
```

```
digits =  0# more than 4
```

In the above, the number of digits in x is computed, so long as the number is 4 or less. If x is negative or greater than 10000, then digits will be set to zero.

Loops

Python provides two looping commands: for and while. These are compound commands.

for loop

The syntax of a for loop is

```
for i te m  in  l i s t :
a c t i o n
```

As usual, the action consists of one or more statements, all at the same indentation level. These statements are also known as the body of the loop. The item is a variable name, and list is a list. Execution of the for loop works by setting the variable successively to each item in the list, and then executing the body each time. Here is a simple example (the comma at the end of the print makes all printing occur on the same line):

```
for i in [2 , 4 , 6 , 0]:
print i,
```

This produces the output

```
2  4  6 0
```

while loop

The syntax of the while loop is

```
while  c o n d i t i o n :
a c t i o n
```

Of course, the action may consist of one or more statements all at the same indentation level. The statements in the action are known as the body of the loop. Execution of the loop works as follows. First the condition is evaluated. If True, the body is executed and the condition evaluated again, and this repeats until the condition evaluates to False. Here is a simple example:

```
n  =  0
while  n
<  10:
```

```
print n,
n = n + 3
```

This produces the following output

```
0 3 6 9
```

Note that the body of a while loop is never executed if the condition evaluates to False the first time. Also, if the body does not change the subsequent evaluations of the condition, an infinite loop may occur. For example

```
while True:
print "
Hello",
```

will print Hellos endlessly. To interrupt the execution of an infinite loop, use CTRL-C.else in loopsA loop may have an optional else which is executed when the loop finishes. For example, the loop

```
for n in [10 ,9 ,8 ,7 ,6 ,5 ,4 ,3 ,2 ,1]:
print n,
else:
print " blastoff"
```

results in the output

```
10 9 8 7 6 5 4 3 2 1 blastoff
```

and the loop

```
n= 10
while n
> 0: print
n,
n = n - 1
else:
print " blastoff"
```

has the same effect (it produces identical output).

Break, Continue, And Pass

The break statement, like in C, breaks out of the smallest enclosing for or while loop. The

continue statement, also borrowed from C, continues with the next iteration of the loop. The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

Here is an example of the use of a break statement and an else clause in a loop.

```
for n in range(2 , 10):
for x in range(2 , n):
if n % x == 0:
print n, ' equals ' , x, '*' , n/x break
else:
# loop fell through without finding a factor
print n, ' is a prime number '
```

The above code searches for prime numbers between 2 and 10, and produces the following output.

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Lists: As already mentioned, a list is a finite sequence of items, and one could use the range function to create lists of integers. In Python, lists are not required to be homogeneous, i.e., the items could be of different types. For example,

```
a = [2 , " Jack", 45 , " 23 W entw orth Ave"]
```

is a perfectly valid list consisting of two integers and two strings. One can refer to the entire list using the identifier a or to the i-th item in the list using a[i].

```
>>> a = [2 , " Jack", 45 , " 23 W entw orth Ave"]
>>> a
[2 , ' Jack' , 45 , ' 23 W entw orth Ave']
>>> a[0] 2
```

```
>>> a[1]
' Jack'
>>> a[2] 45
>>> a[3]
' 23   W entw orth   Ave'
```

Note that the numbering of list items always begins at 0 in Python. So the four items in the above list are indexed by the numbers 0, 1, 2, 3.List items may be assigned a new value; this of course changes the list. For example, with a as above:

```
>>> a
[2 ,  ' Jack' ,  45 ,  ' 23   W entw orth   Ave']
>>>  a[0]  = 2002
>>> a
[2002 ,  ' Jack' ,  45 ,  ' 23   W entw orth   Ave']
```

Of course, the entire list may be assigned a new value, which does not have to be a list. When this happens, the previous value is lost:

```
>>> a
[2002 ,  ' Jack' ,  45 ,  ' 23   W entw orth   Ave']
>>>  a  =  ' gobbletygook'
>>> a
' gobbletygook'
```

Length of a list; empty list

Every list has a length, the number of items in the list, obtained using the len function:

```
>>>  x  =  [9 , 4 , 900 , -45]
>>> len(x) 4
```

Of special importance is the empty list of length 0. This is created as follows:

```
>>>  x  =  []
>>> len(x) 0
```

Sublists (slicing)

Sublists are obtained by slicing, which works analogously to the range function discussed before. If x is an existing list, then x[start:end] is the sub list consisting of all items in the

original list at index positions i such that

$$start \leq i < end.$$

Of course, we must remember that indexing items always starts at 0 in Python. For example,

```
>>> x= range(0 ,20 ,2)
>>>  x
[0 , 2 , 4 , 6 , 8 , 10 , 12 , 14 , 16 , 18]
>>> x[2:5]
[4 , 6 , 8]
>>> x[0:5]
[0 , 2 , 4 , 6 , 8]
```

When taking a slice, either parameter start or end may be omitted: if start is omitted then the slice consists of all items up to, but not including, the one at index position end, similarly, if end is omitted the slice consists of all items starting with the one at position start. For instance, with the list x as defined above we have

```
>>> x[:5]
[0 , 2 , 4 , 6 , 8]
>>> x[2:]
[4 , 6 , 8 , 10 , 12 , 14 , 16 , 18]
```

In this case, x[:5] is equivalent to x[0:5] and x[2:] is equivalent to x[2:len(x)].

There is an optional third parameter in a slice, which if present represents an increment, just as in the range function. For example,

```
>>>  list =  range (20)
>>> list
[0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 , 16 , 17]
>>> list [0 :16 :2 ]
[0 , 2 , 4 , 6 , 8 , 10 , 12 , 14]
>>> list [0 :15 :2 ]
[0 , 2 , 4 , 6 , 8 , 10 , 12 , 14]
```

Notice that one may cleverly use a negative increment to effectively reverse a list, as in

```
>>>  list[18::-1]
```

[17 , 16 , 15 , 14 , 13 , 12 , 11 , 10 , 9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 , 0]

In general, the slice x[len(x)::-1] reverses any existing list x. Joining two lists Two existing lists may be concatenated together to make a longer list, using the + operator

```
>>> [2 ,3 ,6 ,10] + [4 ,0 ,0 ,5 ,0]
[2 , 3 , 6 , 10 , 4 , 0 , 0 , 5 , 0]
```

List methods

If x is the name of an existing list, we can append an item to the end of the list using x. append(item)

For example,

```
>>> x = [3 , 6 , 8 , 9]
>>> x. append (999)
>>> x
[3 , 6 , 8 , 9 , 999]
```

A similar method is called insert, which allows an element to be inserted in the list at a specified position:

```
>>> x = [' a' , ' c' , ' 3 ' , ' d' , ' 7 ']
>>> x. insert (0 ,100)
>>> x
[100 , ' a' , ' c' , ' 3 ' , ' d' , ' 7 ']
>>> x. insert(3 ,' junk')
>>> x
[100 , ' a' , ' c' , ' junk' , ' 3 ' , ' d' , ' 7 ']
```

One can also delete the first occurrence of some item in the list (if possible) using remove as follows:

```
>>> x. remove(' a')
>>> x
[100 , ' c' , ' junk' , ' 3 ' , ' d' , ' 7 ']
```

To delete the item at index position i use x.pop(i), as in:

```
>>> x. pop(0)
100
>>>  x
[’ c’ ,  ’ junk’ ,  ’ 3 ’ ,  ’ d’ ,  ’ 7 ’]
```

Notice that pop not only changes the list, but it also returns the item that was deleted. Also, by default x.pop() pops off the last item:

```
>>> x. pop()
’ 7 ’
>>>  x
[’ c’ ,  ’ junk’ ,  ’ 3 ’ ,  ’ d’]
```

Many more methods exist for manipulating lists; consult the Python Tutorial [1] or Python Library Reference [2] for more details.

Strings

A string in Python is a sequence of characters. In some sense strings are similar to lists, however, there are important differences. One major difference is that Python strings are immutable, meaning that we are not allowed to change individual parts of them as we could for a list. So if x is an existing string, then x[i] gets the character at position i, but we are not allowed to reassign that character, as in x[5] = ’s’.

```
>>>  x  =  ’ gobbletygook’
>>> x[2]
’ b’
>>> x[5]
’ e’
>>>  x[5]  =  ’ s’
Traceback  ( most  recent  call  last): File  "< stdin >",  line  1 ,  in  < module >
```

TypeError: ’ str’ object does not support item assignment Just as for lists, string items are indexed starting at 0. Slicing for strings works exactly the same as for lists. The length function len is the same as for lists, and concatenation is the same too. But the list methods append, insert, delete, and pop are not available for strings, because strings are immutable. If you need to change an existing string, you must make a new, changed, one.There are many string methods for manipulating strings, documented in the Python Library Reference Manual [2]. For example, you

can capitalize an existing string x using x.capitalize(); this returns a new copy of the string in which the first character has been capitalized.

```
>>> a = ' gobbletygook is refreshing'
>>> a. capitalize()
' G obbletygook is refreshing'
```

Other useful methods are find and index, which are used to find the first occurrence of a substring in a given string. See the manuals for details.