



# INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL, HYDERABAD - 500 043

**COMPUTER SCIENCE AND ENGINEERING**

## SCRIPTING LANGUAGES

**B.Tech VIII Semester**

Prepared by:

Ms. CH Srividya

Ms. Y Harika

# UNIT-I

## Introduction to Perl and Scripting

# Perl

- "Practical Extraction and Reporting Language" written by Larry Wall and first released in 1987.
- Perl has become a very large system of modules.
- Name came first, then the acronym designed to be a "glue" language to fill the gap between compiled programs (output of "gcc", etc.) and scripting languages.
- "Perl is a language for easily manipulating text, files and processes": originally aimed at systems administrators and developers.

# What is Perl?

- Perl is a High-level Scripting language.
- Faster than sh or csh, slower than C.
- No need for sed, awk, head, wc, tr, ...
- Compiles at run-time.
- Available for Unix, PC, Mac.
- Best Regular Expressions.

# What's Perl Good For?

- Quick scripts, complex scripts.
- Parsing & restructuring data files.
- CGI-BIN scripts.
- High-level programming
  - Networking libraries
  - Graphics libraries
  - Database interface libraries

# What's Perl Bad For?

- Compute-intensive applications. (use C)
- Hardware interfacing .(device drivers...)

# Executing Perl scripts

- "Bang path" convention for scripts.
  - can invoke Perl at the command line, or
  - add `#!/public/bin/perl` at the beginning of the script.
  - exact value of path depends upon your platform. (use "which perl" to find the path)

- One execution method

```
% perl  
print "Hello, World!\n";  
CTRL-D  
Hello, World!
```

- Preferred method

Set bang-path and ensure executable flag is set on the script file

# Perl Basics

- Comment lines begin with: #
- File Naming Scheme
  - *filename.pl* (programs)
  - *filename.pm* (modules)
- Example prog: `print —Hello, World!\n`;



# Perl Basics

- Statements must end with semicolon.
  - `$a = 0;`
- Should call `exit()` function when finished.
  - Exit value of zero means success
    - `exit (0);`      # successful
  - Exit value non-zero means failure
    - `exit (2);`      # failure

# Data Types

## ➤ Integer

- 25      750000      1\_000\_000\_000
- 8#100    16#FFFF0000

## ➤ Floating Point

- 1.25      50.0      6.02e23      -1.6E-8

## ➤ String

- `__hi there'    —hi there, $name||    qq(tin can)`
- `print —TextUtility, version $ver\n||;`

# Data Types

## ➤ Boolean

- 0 0.0 "0" represent False
- all other values represent True

# Variable Types

## ➤ Scalar

- `$num = 14;`
- `$fullname = —JohnH. Smith`;
- Variable Names are Case Sensitive
- Underlines Allowed: `$Program_Version = 1.0;`

# Scalars

## ➤ Usage of scalars

```
print ("pi is equal to: $pi\n");  
print "pi is still equal to: ", $pi, "\n";  
$c = $a + $b
```

## ➤ A scalar variable can be "used" before it is first assigned a value.

- Result depends on context.
- Either a blank string ("") or a zero (0).
- This is a source of very subtle bugs.
- If variable name is misspelled — what should be the result?
- Do not let yourself get caught by this – use the "-w" flag in the bang path:

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

# Variable Types

## ➤ List (one-dimensional array)

- `@memory = (16, 32, 48, 64);`
- `@people = (—Alice||, —Alex||, —Albert||);`
- First element numbered 0
- Single elements are scalar: `$names[0] = —Fred||;`
- Slices are ranges of elements
  - `@guys = @people[1..2];`
- How big is my list?
  - `print —Number of people: —,scalar @people, —\n||;`

# Variable Types

## ➤ Hash (associative array)

- `%var = { —name⇒ —paul, —age⇒ 33 };`
- Single elements are scalar.
  - `print $var{—name}; $var{age}++;`
- How many elements are in my hash?
  - `@allkeys = keys(%var);`
  - `$num = @allkeys;`

# Operators

## ➤ Math

- The usual suspects:  $+$   $-$   $*$   $/$   $\%$ 
  - `$total = $subtotal * (1 + $tax / 100.0);`
- Exponentiation:  $**$ 
  - `$cube = $value ** 3;`
  - `$cuberoot = $value ** (1.0/3);`
- Bit-level Operations
  - left-shift: `<<`     `$val = $bits << 1;`
  - right-shift: `>>`     `$val = $bits >> 8;`



# Operators

## ➤ Assignments

- As usual: `=` `+=` `-=` `*=` `/=` `**=` `<<=` `>>=`
  - `$value *= 5;`
  - `$longword <<= 16;`
- Increment: `++`
  - `$counter++`                      `++$counter`
- Decrement: `--`
  - `$num_tries----` `$num_tries`

# Arithmetic Operators

- Perl operators are the same as in C and Java these are only good for numbers.
  - But `$b = "3" + "5";`  
`print $b, "\n";` # prints the number 8
  - If a string can be interpreted as a number given arithmetic operators.
  - What is the value of `$b`?:  
`$b = "3" + "five" + 6?`
  - Perl semantics can be tricky to completely understand.

# Operators

## ➤ Boolean (against bits in each byte)

- Usual operators:  $\&$   $|$
- Exclusive-or:  $\wedge$
- Bitwise Negation:  $\sim$ 
  - `$picture = $backgnd &  $\sim$ $mask | $image;`

## ➤ Boolean Assignment

- $\&=$   $|=$   $\wedge=$ 
  - `$picture &= $mask;`

# Logical Operators

## ➤ Logical Operators

- `&&`                      And operator
- `||`                        Or operator
- `!`                         Not operator

# Short Circuit Operators

## □ `expr1 && expr2`

- `expr1` is evaluated.
- `expr2` is only evaluated if `expr1` was true.

## □ `expr1 || expr2`

- `expr1` is evaluated.
- `expr2` is only evaluated if `expr1` was false.

## □ Examples

- `open (...) || die —couldn't open file`;
- `$debug && print —user's name is`  
`$name\n`;

# Continued...

## ➤ Modulo: %

➤ `$a = 123 % 10;`    (`$a` is 3)

## ➤ Multiplier: x

➤ `print —rideon the ||, —choo-||x2, —train||;`  
(prints `—rideon the choo-choo-train||`)

➤ `$stars = —*||x 80;`

## ➤ Assignment:      %=      x=

# Operators

## ➤ String Concatenation: `.=`

- `$name = —Uncle . $space . —Sam`;
- `$cost = 34.99`;
- `$price = —HopeDiamond, now only \`;
- `$price .= —$cost`;

# Conditional Operators

	<u>Numeric</u>	<u>String</u>
➤ Equal:	==	eq
➤ Less/Greater Than:	< >	lt gt
➤ Less/Greater or equal:	<= >=	le ge
➤ Zero and empty-string means	False	
➤ All other values equate to	True	

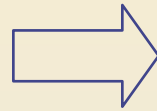


# Conditional Operators

	Numeric	String
➤ Comparison	<=>	cmp
■ Results in a value of -1, 0, or 1		
➤ Logical Not	!	
■ if (! \$done)		
{		
print —keep going\;		
}		

# Numeric Vs. String Comparisons

```
#!/usr/bin/perl
$a = "123";
$b = "1234";
$c =
"124";
if
($b > $c)
{
    print "$b > $c\n";
} else {
    print "$b <= $c\n";
}
if ($b gt $c)
{
    print "$b gt $c\n";
} else
{
    print "$b le $c\n";
}
```



1234 > 124  
1234 le 124

# Control Structures

## ➤ —IF statement - first style

```
if ($porridge_temp < 40)
{
    print —toohot.\n\;
}
elseif ($porridge_temp > 150)
{
    print —toocold.\n\;
}
else
{
    print —just right\n\;
}
```

# Control Structures

## ➤ —IF statement - second style

- Statement if condition;

- print —\ \$index is \$index \| if \$DEBUG;

- Single statements only

- Simple expressions only

## ◆ —Unless is a reverse —if

- Statement unless condition;

- print —millennium is here! \| unless \$year < 2000;

# Control Structures

## ➤ —For loop - first style

- `for (initial; condition; increment) { code }`
  - `for ($i=0; $i<10; $i++)`  
`{`  
`print —hello\n`  
`}`

## ➤ —For loop - second style

- `for [variable] (range) { code }`
  - `for $name (@employees)`
    - `{`  
`print —$name is an employee.\n`  
`}`

# Control Structures

## ➤ —For loop with default loop variable

- for (@employees)

- {

- print —\$\_ is an employee\n

- print; # this prints —\$\_

- }

## ➤ Foreach and For are actually the same.

# Control Structures

## ➤ — While loop

- `while (condition) { code }`
  - `$cars = 7;`  
`while ($cars > 0) {`  
    `print —carsleft: ||, $cars--, —\n||;`  
`}`
  - `while ($game_not_over) {...}`

# Control Structures

➤ —Until loop is opposite of —while

- `until (condition) { code }`

- `$cars = 7;`

- `until ($cars <= 0)`

- `{`

- `print —carsleft: ||, $cars--, —\n||;`

- `}`

- `while ($game_not_over) {...}`



# Control Structures

## ➤ Bottom-check Loops

- `do { code } while (condition);`
- `do { code } until (condition);`
  - `$value = 0;`  
`do {`  
    `print —Enter Value: \;`  
    `$value = <STDIN>;`  
    `} until ($value > 0);`

# No Switch Statement?!?

- Perl needs no Switch (Case) statement.
- Use if/else combinations instead
  - `if (cond1) { ... }`  
    `elsif (cond2) { ... }`  
    `elsif...`  
    `else...`
- This will be optimized at compile time.

# Subroutines (Functions)

## ➤ Defining a Subroutine

- `sub name { code }`
- Arguments passed in via `—@_list`
  - `sub multiply`
  - ```
{  
    my ($a, $b) = @_;  
    return $a * $b;  
}
```
- Last value processed is the return value.  
(could have left out word `—return`, above)

# Subroutines (Functions)

## ➤ Calling a Subroutine

- `&subname;`      # no args, no return value
- `&subname (args);`
- `retval = &subname (args);`
- The `—&` is optional so long as...
  - `subname` is not a reserved word.
  - subroutine was defined before being called.

# Subroutines (Functions)

## ➤ Passing Arguments

- Passes the value
- Lists are expanded

- `@a = (5,10,15);`  
`@b = (20,25);`  
`&mysub(@a,@b);`

- This passes five arguments: 5,10,15,20,25
    - `mysub` can receive them as 5 scalars, or one array

# Subroutines (Functions)

## ➤ Examples

- sub good1

```
{  
    my($a,$b,$c) = @_;  
}  
&good1 (@triplet);
```

- sub good2

```
{  
    my(@a) = @_;  
}  
&good2 ($one, $two, $three);
```

# Subroutines (Functions)

## ➤ Examples

- sub good3

- {

- my(\$a,\$b,@c) = @\_;

- }

- &good3 (\$name, \$phone, @address);

- sub bad1

- {

- my(@a,\$b) = @\_;

- }

- @a will absorb all args, \$b will have nothing.

# Dealing with Hashes

- `Keys( )` - get an array of all keys
  - `foreach (keys (%hash)) { ... }`
- `Values( )` - get an array of all values
  - `@array = values (%hash);`
- `Each( )` - get key/value pairs
  - ```
while (@pair = each(%hash))  
{  
    print —element $pair[0] has $pair[1]\n;  
}
```



# Dealing with Hashes

- `Exists( )` - check if element exists
  - `if (exists $ARRAY{$key}) { ... }`
- `Delete( )` - delete one element
  - `delete $ARRAY{$key};`

# Launching External Programs

- The `—system` library call
  - example: `system (—s-la)`;
  - Returns exit status of program it launched
  - a shell actually runs, so you can use:
    - pipes (`—||`)
    - redirection (`—<|`, `—>|`)

# Launching External Programs

- Run a command, insert output inline
  - `$numlines = `wc -l < /etc/passwd`;`

# Command Line Arguments

- `$0` = program name
- `@ARGV` array of arguments to program
- Zero-based index (default for all arrays)
- Example
  - `yourprog -a somefile`
    - `$0` is `—yourprog`
    - `$ARGV[0]` is `—a`
    - `$ARGV[1]` is `—somefile`

# Basic File I/O

## ➤ Reading a File

```
■ open (FILEHANDLE, —$filename||) || die \  
  —open of $filename failed: $!||;  
while (<FILEHANDLE>)  
{  
    chomp $_;           # or just:  chomp;  
    print —$_\n||;  
}  
close FILEHANDLE;
```

# Basic File I/O

## ➤ Writing a File

```
■ open (FILEHANDLE, —>$filename) || die \
  —open of $filename failed: $!||;
while (@data) {
    print FILEHANDLE —$_\n||;
    # note, no comma!
}
close FILEHANDLE;
```

# Basic File I/O

## ➤ Predefined File Handles

- `<STDIN>`                      input
- `<STDOUT>`                    output
- `<STDERR>`                    output
  - `print STDERR —bigbad error occurred\n`;
- `<>`                              ARGV or STDIN

# Basic File I/O

## ➤ How does `<>` work?

- Opens each ARGV filename for reading
- If no ARGV's, reads from stdin
- Great for writing filters, here's `cat`:
  - ```
while (<>) {  
    print;           # same as  print —$_  
}
```



# Basic File I/O

## ➤ Reading from a Pipe

```
■ open (FILEHANDLE, —psaux ||) || die \
  —launch of _ps‘ failed: $!||;
while (<FILEHANDLE>)
{
    chomp;
    print —$_\n||;
}
close FILEHANDLE;
```

# Basic File I/O

## ➤ Writing to a Pipe

```
■ open (FILEHANDLE, —|mail frank|) || die \  
  —‘launch of _mail‘ failed: $!|;  
while (@data)  
{  
    print FILEHANDLE —$_\n|;  
}  
close FILEHANDLE;
```

# Common mistakes

- Writing comma after filehandle in print statement
- Using == instead of eq, and != instead of ne
- Leaving \$ off the front of a variable on the left side of an assignment
- Forgetting the & on a subroutine call
- Leaving \$ off of the loop variable of foreach
- Using else if or elif instead of elsif
- Forgetting trailing semicolon
- Forgetting the @ or @ on the front of variables
- Saying @foo[1] when you mean \$foo[1]

# UNIT-II

## Advanced Perl

# Why PERL ???

- Practical extraction and report language
- Similar to shell script but lot easier and more powerful
- Easy availability
- All details available on web

# Why PERL ???

- Perl stands for practical extraction and report language.
- Perl is similar to shell script. Only it is much easier and more akin to the high end programming.
- Perl is free to download from the GNU website so it is very easily accessible .
- Perl is also available for MS-DOS, WIN-NT and Macintosh.

# Basic Concepts

- Perl files extension .Pl
- Can create self executing scripts
- Advantage of Perl
- Can use system commands
- Comment entry
- Print stuff on screen

# Basics

- Can make perl files self executable by making first line as `#!/bin/perl`.
  - The extension tells the kernel that the script is a perl script and the first line tells it where to look for perl.
- The `-w` switch tells perl to produce extra warning messages about potentially dangerous constructs.



# Basics

- The advantage of Perl is that you don't have to compile create object file and then execute.
- All commands have to end in ";" can use unix commands by using.
  - ❑ `System("unix command");`
- EG: `system("ls *");`
  - ❑ Will give the directory listing on the terminal where it is running.

# Basics

- The pound sign "#" is the symbol for comment entry. There is no multiline comment entry , so you have to use repeated # for each line.
- The "print command" is used to write outputs on the screen.
  - Eg: `print "this is ece 902";`  
Prints "this is ece 902" on the screen .It is very similar to printf statement in C.
- If you want to use formats for printing you can use printf.

# How to Store Values

- Scalar variables
- List variables
- Push, pop, shift, unshift, reverse
- Hashes, keys, values, each
- Read from terminal, command line arguments
- Read and write to files

# Scalar Variables

- They should always be preceded with the \$ symbol.
- There is no necessity to declare the variable before hand .
- There are no data types such as character or numeric.
- The scalar variable means that it can store only one value.

# Scalar Variable

- If you treat the variable as character then it can store a character.
- If you treat it as string it can store one word .If you treat it as a number it can store one number.
- Eg \$name = "betty" ;
  - The value betty is stored in the scalar variable \$name .

# Scalar Variable

- EG: `print "$name \n";` The output on the screen will be betty.
- Default values for all variables is undef. Which is equivalent to null.

# List Variables

- They are like arrays. It can be considered as a group of scalar variables.
- They are always preceded by the @symbol.
  - Eg @names = ("betty", "veronica",— tom");
- Like in C the index starts from 0.

# List Variables

- If you want the second name you should use `$names[1]` ;
- Watch the `$` symbol here because each element is a scalar variable.
- `$` Followed by the list variable gives the length of the list variable.
  - Eg `$names` here will give you the value 3.



# Operators in Lists

- These are operators operating on the list variables.
- Push and pop treat the list variable as a stack and operate on it. They act on the higher subscript.
  - Eg `push(@names,"lily")` , now the `@names` will contain `("betty","veronica","tom","lily")`.
  - Eg `pop(@names)` will return `"lily"` which is the last value. And `@names` will contain `("betty","veronica","tom")`.

# Operators in Lists

- Shift and unshift act on the lower subscript.
  - Eg `unshift(@names,"lily")`, now `@names` contains `("lily","betty","veronica","tom")`.
  - Eg `shift(@names)` returns `"lily"` and `@names` contains `("betty","veronica","tom")`.
- Reverse reverses the list and returns it.

# Hashes

- Hashes are like arrays but instead of having numbers as their index they can have any scalars as index.
- Hashes are preceded by a % symbol.
  - Eg we can have %roll numbers = ("A",1,"B",2,"C",3);

# Keys

- If we want to get the roll number of A we have to say `$roll numbers{"a"}`. This will return the value of roll number of A.
- Here A is called the key and the 1 is called its value.
- `Keys()` returns a list of all the keys of the given hash.
- `Values` returns the list of all the values in a given hash.

# keys

- Each function iterates over the entire hash returning two scalar value the first is the key and the second is the value
  - Eg \$firstname,\$lastname = each(%lastname) ;
  - Here the \$firstname and the \$lastname will get a new key value pair during each iteration

# Read / Write to Files

- To read and write to files we should create something called handles which refer to the files.
- To create the handles we use the OPEN command.
  - Eg `open(filehandle1,"filename");` Will create the handle called FILEHANDLE1 for the file "filename".

# Read / Write to Files

- This handle will be used for reading.
  - Eg `open(filehandle2,">filename");` Will create the handle called FILEHANDLE2 for the file "filename".
- This handle will be used for writing.
- Watch out for the ">" symbol before the filename.  
This indicates that the file is opened for writing.

# Read / Write to Files

- Once the file handles have been obtained . the reading and writing to files is pretty simple.
  - Eg \$linevalue = <FILEHANDLE1> ;
- This will result in a line being read from the file pointed by the filehandle and the that line is stored in the scalar variable \$linevalue.



# Read / Write to Files

- When the end of file is reached the `<FILEHANDLE1>` returns a `undef`.
  - Eg `print FILEHANDLE2 "$linevalue\n";`
- This will result in a line with the value as in `$linevalue` being written to the file pointed by the `filehandle2`.
- For closing a filehandle use `lose(FILEHANDLE);`

# Control Structures

- If / unless statements
- While / until statements
- For statements
- Foreach statements
- Last , next , redo statements
- && And || as control structures

# If / Unless

- If similar to the if in C.
- Eg of unless.
  - Unless(condition){ }.
- When you want to leave the then part and have just an else part we use unless.

# While / Until / For

- While is similar to the while in C.
- Eg until.
  - Until(some expression){ }.
- So the statements are executed till the condition is met.
- For is also similar to C implementation.

# Foreach Statement

➤ This statement takes a list of values and assigns them one at a time to a scalar variable, executing a block of code with each successive assignment.

➤ Eg: `Foreach $var (list) { }`.

# Last / Next / Redo

- Last is similar to break statement of C.
  - Whenever you want to quit from a loop you can use this.
- To skip the current loop use the next statement.
  - It immediately jumps to the next iteration of the loop.
- The redo statement helps in repeating the same iteration again.

# && And || Controls

- Unless(cond1){cond2}.
  - This can be replaced by cond1&&cond2.
- Suppose you want to open a file and put a message if the file operation fails we can do.
  - (Condition)|| print "the file cannot be opened—;
- This way we can make the control structures smaller and efficient.

# Functions

- Function declaration
- Calling a function
- Passing parameters
- Local variables
- Returning values



# Function Declaration

- The keyword `sub` describes the function.
  - So the function should start with the keyword `sub`.
  - Eg `sub addnum { .... }`.
- It should be preferably either in the end or in the beginning of the main program to improve readability and also ease in debugging.

# Function Calls

- `$Name = &getname();`
- The symbol `&` should precede the function name in any function call.

# Parameters of Functions

- We can pass parameter to the function as a list.
- The parameter is taken in as a list which is denoted by `@_` inside the function.
- So if you pass only one parameter the size of `@_` list will only be one variable. If you pass two parameters then the `@_` size will be two and the two parameters can be accessed by `$_[0]`, `$_[1]` ....

# More About Functions

- The variables declared in the main program are by default global so they will continue to have their values in the function also.
- The result of the last operation is usually the value that is returned unless there is an explicit return statement returning a particular value.

# More About Functions

- There are no pointers in Perl but we can manipulate and even create complicated data structures.
- Local variables are declared by putting 'my' while declaring the variable.

# Regular Expressions

- Split and join
- Matching & replacing
- Selecting a different target
- `$&`, `$'`, And `$``
- Parenthesis as memory
- Using different delimiter

# Split And Join

- Split is used to form a list from a scalar data depending on the delimiter.
- The default delimiter is the space.
- It is usually used to get the independent fields from a record.
  - Eg: `$linevalue = "R101 tom 89%";`
  - `$_ = $linevalue.`
  - `@Data = split();`

# Split and Join

- Here \$data[0] will contain R101 , \$data[1] tom, \$data[2] .
- Split by default acts on \$\_ variable.
- If split has to perform on some other scalar variable. Syntax is
  - ❑ Split (//,\$linevalue);
- If split has to work on some other delimiter then syntax is.
  - ❑ Split(/<delimiter>/,\$linevalue);



# Special Variables

- `$&` Stores the value which matched with pattern.
- `$'` Stores the value which came after the pattern in the linevalue.
- `$`` Stores the value which came before the pattern in the linevalue.

# Split and Join

- Join does the exact opposite job as that of the split.
- It takes a list and joins up all its values into a single scalar variable using the delimiter provided.
  - Eg `$newlinevalue = join( @data);`

# Matching and Replacing

- Suppose you need to look for a pattern and replace it with another one you can do the same thing as what you do in unix the command in perl is
  - `S/<pattern>/<replace pattern>.`
- This by default acts on the `$_` variable. If it has to act on a different source variable (Eg `$newval`) then you have to use.

■ Eg `@newval=~s/<pattern>/<replace pattern> .`

# Parenthesis VS Memory

- Parenthesis as memory.
  - Eg `fred(.)Barney\1); .`
- Here the dot after the fred indicates the it is memory element. That is the `\1` indicates that the character there will be replaced by the first memory element. Which in this case is the any character which is matched at that position after fred.

# **UNIT-III**

## **Advanced PHP Programming**

# What is PHP?

- PHP == ‘\_PHP Hypertext Preprocessor’
- Open-source, server-side scripting language
- Used to generate dynamic web-pages
- PHP scripts reside between reserved PHP tags
- This allows the programmer to embed PHP scripts within HTML pages

# What is PHP (cont'd)

- Interpreted language, scripts are parsed at run-time rather than compiled beforehand
- Executed on the server-side
- Source-code not visible by client
- View Source in browsers does not display the PHP code
- Various built-in functions allow for fast development
- Compatible with many popular databases

# What does PHP code look like?

- Structurally similar to C/C++.
- Supports procedural and object-oriented paradigm.  
(to some degree)
- All PHP statements end with a semi-colon.
- Each PHP script must be enclosed in the reserved  
PHP tag.

```
<?php  
...  
?>
```



# Comments in PHP

- Standard C, C++, and shell comment symbols

```
// C++ and Java-style comment
```

```
# Shell-style comments
```

```
/* C-style comments
```

```
    These can span multiple  
lines */
```

# Variables in PHP

- PHP variables must begin with a `$` sign.
- Case-sensitive. (`$Foo` `!=` `$foo` `!=` `$fOo`)
- Global and locally-scoped variables.
- Global variables can be used anywhere.
- Local variables restricted to a function or class.
- Certain variable names reserved by PHP.
- Form variables. (`$_POST`, `$_GET`)
- Server variables. (`$_SERVER`)

# Variable Usage

```
<?php
$foo = 25;      // Numerical
variable
$bar = "Hello"; // String variable
$foo = ($foo * 7); // Multiplies
foo by 7
$bar = ($bar * 7); // Invalid
expression
?>
```

# Echo

- The PHP command `__echo` is used to output the parameters passed to it.
- The typical usage for this is to send data to the client's web-browser.

## Syntax

- `void echo (string arg1 [, string argn...])`
- In practice, arguments are not passed in parentheses since echo is a language construct rather than an actual function

# Echo example

```
<?php
$foo = 25;           // Numerical variable
$bar = "Hello";      // String variable
echo $bar;           // Outputs Hello
echo $foo,$bar;       // Outputs 25Hello
echo "5x5=", $foo;    // Outputs 5x5=25
echo "5x5=$foo";      // Outputs 5x5=25
echo „5x5=$foo";     // Outputs 5x5=$foo
?>
```

# Echo example

- Notice how `echo 5x5=$foo` outputs `$foo` rather than replacing it with `25`
- Strings in single quotes (`'`) are not interpreted or evaluated by PHP
- This is true for both variables and character escape-sequences (such as `—\n` or `—\\`)

# Arithmetic Operations

- $\$a - \$b$  // subtraction
- $\$a * \$b$  // multiplication
- $\$a / \$b$  // division
- $\$a += 5$  //  $\$a = \$a + 5$  Also works for  $*=$  and  $/=$

# Example Program

```
<?  php
    $a=15;
    $b=30;
    $total=$a+$b;
    Print $total;
    Print —<p><h1>$total</h1>||;
    // total is 45
?>
```



# Concatenation

- Use a period to join strings into one.

```
<?php
$string1="Hello";
$string2="PHP";
$string3=$string1 . " " .
$string2;
Print $string3;
?>
```

```
Hello PHP
```

# Escaping the Character

- If the string has a set of double quotation marks that must remain visible, use the \ [backslash] before the quotation marks to ignore and display them.

```
<?php  
$heading="\ "Computer Science\ "  
Print $heading;
```

```
?>
```

```
"Computer Science"
```

# PHP Control Structures

- Control Structures: The structures within a language that allow us to control the flow of execution through a program or script.
- Grouped into conditional (branching) structures (e.g. if/else) and repetition structures (e.g. while loops).
- Example if/else if/else statement:

```
if ($foo == 0)
{
    echo 'The variable foo is equal to 0';
}
else if (($foo > 0) && ($foo <= 5)) {
    echo 'The variable foo is between 1 and 5';
}
else {
    echo 'The variable foo is equal to `.$foo';
}
```

# If ... Else...

➤ If (condition)  
{  
Statements;  
}  
Else  
{  
Statement;  
}

```
<?php  
If ($user=="John")  
{  
Print "Hello John.";  
}  
Else  
{  
Print "You are not  
John.";  
}  
?>
```

# While Loops

## ➤ While (condition)

{

Statements;

}

```
hello PHP.  
hello PHP.  
hello PHP.
```

```
<?php  
$count=0;  
While ($count<3)  
{  
    Print "hello  
PHP. ";  
    $count += 1;  
    // $count =  
$count + 1;  
    // or  
    // $count++;  
?>
```

# Date Display

```
$datedisplay=date(—yyyy/m/d||);
```

```
Print $datedisplay;
```

```
# If the date is April 1st, 2009
```

```
# It would display as 2009/4/1
```

```
2009/4/1
```

```
$datedisplay=date(—l, F m, Y||);
```

```
Print $datedisplay;
```

```
# If the date is April 1st, 2009
```

```
# Wednesday, April 1, 2009
```

```
Wednesday, April 1, 2009
```

# Month, Day & Date Format Symbols

|   |         |
|---|---------|
| M | Jan     |
| F | January |
| m | 01      |
| n | 1       |

|              |   |        |
|--------------|---|--------|
| Day of Month | d | 01     |
| Day of Month | J | 1      |
| Day of Week  | l | Monday |
| Day of Week  | D | Mon    |

# Functions

- Functions MUST be defined before then can be called.
- Function headers are of the format.  

```
function functionName($arg_1, $arg_2, ..., $arg_n)
```
- Note that no return type is specified.
- Unlike variables, function names are not case sensitive. (foo(...) == Foo(...) == FoO(...))



# Functions Example Program

```
<?php
    // This is a function
    function foo($arg_1, $arg_2)
    {
        $arg_2 = $arg_1 * $arg_2;
        return $arg_2;
    }
    $result_1 = foo(12, 3); // Store the function
    echo $result_1;         // Outputs 36
    echo foo(12, 3);        // Outputs 36
?>
```

# Include Files

- Include —opendb.php||;
- Include —closedb.php||;
- This inserts files; the code in files will be inserted into current code.
- This will provide useful and protective means once you connect to a database, as well as for other repeated functions.

Include (—footer.php||);

The file footer.php might look like:

```
<hr SIZE=11 NOSHADE WIDTH=—100%||>  
<i>Copyright © 2008-2010 KSU </i></font><br>  
<i>ALL RIGHTS RESERVED</i></font><br>  
<i>URL: http://www.kent.edu</i></font><br>
```

# PHP - Forms

- Access to the HTTP POST and GET data is simple in PHP
- The global variables `$_POST[]` and `$_GET[]` contain the request data.

```
<? php
    if ($_POST["submit"])
        echo "<h2>You clicked Submit!</h2>";
    else if ($_POST["cancel"])
        echo "<h2>You clicked Cancel!</h2>";
?>
<form action="form.php" method="post">
    <input type="submit" name="submit" value="Submit">
    <input type="submit" name="cancel" value="Cancel">
</form>
```

# PHP Overview

- Easy learning.
- Syntax Perl- and C-like syntax. Relatively easy to learn.
- Large function library
- Embedded directly into HTML
- Interpreted, no need to compile
- Open Source server-side scripting language designed specifically for the web.

# PHP Overview (cont.)

- Conceived in 1994, now used on +10 million web sites.
- Outputs not only HTML but can output XML, images (JPG & PNG), PDF files and even Flash movies all generated on the fly. Can write these files to the file system.
- Supports a wide-range of databases (20+ODBC).
- PHP also has support for talking to other services using protocols such as LDAP, IMAP, SNMP, NNTP, POP3, HTTP.

# First PHP script

## ➤ Save as sample.php:

```
<!-- sample.php -->
<html><body>
<strong>Hello World!</strong><br />
<?php
echo —<h2>Hello, World</h2>||; ?>
<?php
    $myvar = "Hello World";
    echo $myvar;
?>
</body></html>
```

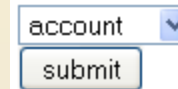
# Data in the tables

- Function: list all tables in your database.  
Users can select one of tables, and show all contents in this table.
- second.php
- showtable.php

# Second.Php

```
<html><head><title>MySQL Table Viewer</title></head><body>
<?php
// change the value of $dbuser and $dbpass to your username and password
$dbhost = 'hercules.cs.kent.edu:3306';
$dbuser = 'nruan';
$dbpass = '*****';
$dbname = $dbuser;
$table = 'account';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if (!$conn) {
    die('Could not connect: ' . mysql_error());
}
if (!mysql_select_db($dbname))
    die("Can't select database");
```

**Choose one table:**



account ▼  
submit

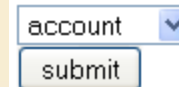


# Second.Php (Cont.)

```
$result = mysql_query("SHOW TABLES");
if (!$result) {
    die("Query to show fields from table failed");
}
$num_row = mysql_num_rows($result);
echo "<h1>Choose one table:<h1>";
echo "<form action=\"showtable.php\" method=\"POST\">";
echo "<select name=\"table\" size=\"1\" Font size=\"+2\">";
for($i=0; $i<$num_row; $i++) {
    $tablename=mysql_fetch_row($result);
    echo "<option value=\"{" $tablename[0]}\" >{" $tablename[0]}</option>";
}
echo "</select>";
echo "<div><input type=\"submit\" value=\"submit\"></div>";
echo "</form>";

mysql_free_result($result);
mysql_close($conn);
?>
</body></html>
```

**Choose one table:**



account ▼

submit

# Showtable.php

```
<html><head>
<title>MySQL Table Viewer</title>
</head>
<body>
<?php
$dbhost = 'hercules.cs.kent.edu:3306';
$dbuser = 'nruan';
$dbpass = '*****';
$dbname = 'nruan';
$table = $_POST[—table];
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if (!$conn)
    die('Could not connect: ' . mysql_error());
if (!mysql_select_db($dbname))
    die("Can't select database");
$result = mysql_query("SELECT * FROM {$table}");
if (!$result) die("Query to show fields from table failed!" . mysql_error());
```

**Table: account**

account_number	branch_name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350
A-111	Perryridge	900

# Showtable.php (Cont.)

```
$fields_num = mysql_num_fields($result);
echo "<h1>Table: {$table}</h1>";
echo "<table border='1'><tr>";
// printing table headers
for($i=0; $i<$fields_num; $i++) {
    $field = mysql_fetch_field($result);
    echo "<td><b>{$field->name}</b></td>";
}
echo "</tr>\n";
while($row = mysql_fetch_row($result)) {
    echo "<tr>";
    // $row is array... foreach( .. ) puts every element
    // of $row to $cell variable
    foreach($row as $cell)
        echo "<td>$cell</td>";
    echo "</tr>\n";
}
mysql_free_result($result);
mysql_close($conn);
?>
</body></html>
```

**Table: account**

account_number	branch_name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350
A-111	Perryridge	900

# Functions Covered

- `mysql_connect()`
- `mysql_select_db()`
- `include()`
- `mysql_query()`
- `mysql_num_rows()`
- `mysql_fetch_array()`
- `mysql_close()`

# History of PHP

- PHP began in 1995 when Rasmus Lerdorf developed a Perl/CGI script toolset he called the Personal Home Page or PHP.
- PHP 2 released 1997 (PHP now stands for Hypertext Processor). Lerdorf developed it further, using C instead PHP3 released in 1998 (50,000 users)
- PHP4 released in 2000 (3.6 million domains). Considered debut of functional language and including Perl parsing, with other major features
- PHP5.0.0 released July 13, 2004 (113 libraries>1,000 functions with extensive object-oriented programming)
- PHP5.0.5 released Sept. 6, 2005 for maintenance and bug fixes

# Create your own homepage

- Login loki.cs.kent.edu.
- Create directory —public\_html in your home directory.
- Create two php files (second.php and showtable.php) we have discussed
- Visit your homepage:
- [http://www.cs.kent.edu/~\[username\]/second.php](http://www.cs.kent.edu/~[username]/second.php)

# UNIT-IV

## Tcl/TK

# Learning Tcl/TK

- What is Tcl/TK?
  - An interpreted programming language
    - Build on-the-fly commands, procedures
    - Platform-independent
    - Easy to use for building GUIs
- Need little experience with programming
  - Easy
  - Programs are short, efficient
- Be willing to learn something new



# Why Tcl/TK

- Easy, fast programming
- Free
- Download & install Tcl/TK 8.4 on your own
  - CSE machines (state) are set up with Tcl/TK 8.0
  - <http://tcl.activestate.com/software/tcltk/downloadnow84.tml>
- Lots of online documentation, mostly free
- Solutions for AI homework will be in Tcl
- Base for the CSLU toolkit

# Hello World

## ➤ How to run your Tcl program

- Command line (state.cse.ogi.edu or DOS)
  - ◆ Type "tclsh" to launch the console
- Type your program directly on the console
- Use the command "source" (source filename)
- Double click your .tcl file (if associated)

## ◆ Output on the console

- Command: puts "Hello, world!"

# Hello World

- Command line (state.cse.ogi.edu or DOS)
  - Type "tclsh" to launch the console
  - Type tcl code into console

 state.cse.ogi.edu - PuTTY

```
state% tclsh
% puts "Hello, world!"
Hello, world!
% exit
state% █
```

# Hello World

## ➤ Sourced on the console

- Type "tclsh", followed by name of program file

```
##### hello.tcl #####
```

```
puts "Hello, world!"
```



A screenshot of a Windows 2000 command prompt window. The title bar reads "C:\WINNT\System32\cmd.exe". The window content shows the following text: "Microsoft Windows 2000 [Version 5.00.2195]", "(C) Copyright 1985-2000 Microsoft Corp.", "C:\>tclsh hello.tcl", "Hello, world!", and "C:\>".

# Hello World

- Double-clicking your .tcl file (if associated with wish84.exe)

```
##### hello.tcl #####
```

```
Hello.tcl
```

```
wm withdraw .
```

```
console show
```

```
puts "Hello, world!"
```



# Basic operations

## ➤ Print to screen (puts)

- ❑ `puts -nonewline "Hello, world!"`
- ❑ `puts "!!"`

## ➤ Assignment (set)

- ❑ `set income 32000`
- ❑ `puts "income is $income"` (using '\$' to get the value of a variable)

## ➤ Mathematical Expressions (expr)

- ❑ `set a 10.0`
- ❑ `expr $a + 5`
- ❑ `expr int($a/3)`

# Some Useful Commands

- Unset: destroy a variable

```
unset num
```

- Info: check whether the named variable has been defined

```
if {![info exists num]}
```

```
{
```

```
  set num 0
```

```
}
```

```
incr num
```

- Window commands

```
wm withdraw .
```

```
console show
```

# Special characters

`#` : single-line comments, similar to `"/"` in C

`;``#` : in-line comments, just like `"/"` in C

`\` : escape character, same function as in C also used to break a long line of code to two lines

`$` : get the value of a variable

- `var` : name of variable

- `$var` : value of variable

`[]` : evaluate command inside brackets



# Control structures

## ➤ If then else

```
set income 32000
if {$income > 30000} {
  puts "$income -- high"
} elseif {$income > 20000} {
  puts "$income -- middle"
} else {
  puts "$income -- low"
}
```

## ➤ while loops

```
set i 0
while {$i < 100} {
  puts "I am at count $i"
  incr i
}
```

# Control structures

## ➤ For loops

```
for {set i 0} {$i < 100} {incr i}
{
    puts "I am at count $i and going up"
    after 300
    update
}
for {set i 100} {$i > 0} {set i [expr $i - 1]} {
    puts "I am at count $i and going down"
}
```

## ➤ foreach loops

```
set lstColors {red orange yellow green blue purple}
foreach c $lstColors
{
    puts $c
}
```

# Control structures

## ➤Foreach loops (con't)

```
set lstColors {red orange yellow green blue purple}
foreach {a b c} $lstColors {
  puts "$c--$b--$a"
}

set lstFoods {apple orange banana lime berry grape}
foreach f $lstFoods c $lstColors {
  puts "a $f is usually $c"
}

foreach {a b} $lstFoods c $lstColors {
  puts "$a & $b are foods. $c is a color."
}
```

# Procedures

## ➤ Procedure calls (embedded commands)

- ❑ `set b [expr $a + 5]`
- ❑ `puts "The value of b is $b"`

## ➤ Create your own procedure (called by value only)

```
proc foo {a b c}
{
    return [expr $a * $b - $c]
}
puts [expr [foo 2 3 4] + 5]
proc bar { }
{
    puts "I'm in the bar procedure"
}
bar
```

# Variable scope

## Local and Global variables

```
set a 5
set b 6
set c 7
proc var_scope { }
{
    global a
    set a 3
    set b 2
    set ::c 1
}
var_scope
puts "The value for a b c is: $a $b $c"
```

# Lists in Tcl/TK

- Everything is a list!
- Many ways to create a list
  - `Set myList [list a b c]`
  - `Set myList "a b c"`
  - `Sset myList {a b c}`
  - `Sset myList [list $a $b $c]`
  - `Set myList {$a $b $c}`
  - `Set myList [list a b c]`
  - `Set myList "a b c"`
  - `Set s Hello`
  - `Puts "The length of $s is [string length $s]."`  
=> The length of Hello is 5.
  - `Puts {The length of $s is [string length $s].}`
  - => The length of \$s is [string length \$s].

# List operations

- Set lstStudents [list "Fan" "Kristy" "Susan"]
- Puts [**lindex** \$lstStudents 0]
- Puts [**lindex** \$lstStudents end]
- Puts [**llength** lstStudents] (unexpected result!)
- Puts [**llength** \$lstStudents]
- **Lappend** \$lstStudents "Peter" (wrong!)
- **Lappend** lstStudents "Peter"

# List operations

- Puts [**linsert** lstStudents 2 "Tom"] (wrong!)
- Puts [**linsert** \$lstStudents 2 "Tom"]
- Set lstStudents [**linsert** \$lstStudents 2 "Tom"]
- Set lstStudents [**lreplace** \$lstStudents 3 3  
"Rachel"]
- Set lstStudents [**lreplace** \$lstStudents end end]
- Set lstStudents [**lsort** –ascii \$lstStudents]
- Puts [**lsearch** \$lstStudents "Peter"]



# Lists of lists (of lists...)

```
Set a [list [list x y z]]
```

```
Puts [lindex $a 0]
```

```
Puts [lindex [lindex $a 0] 1]
```

```
Puts [lindex [lindex $a 1] 0] (unexpected result)
```

```
Set a [list x [list [list y] [list z]]]
```

=> How to get to the z?

```
Set arg1 [list g [list f [list h [list i X]]] [list r Y] k]
```

```
Set arg2 [list g [list f [list h [list i Y]]] [list r b] L]
```

```
Set both [list $arg1 $arg2]
```

```
Puts $both
```

# Array operations

**Associative arrays** (string as index)

set color(rose) red

set color(sky) blue

set color(medal) gold

set color(leaves) green

set color(blackboard) black

puts [array exists color] (tests if an array with  
the name "color" exists)

# Array operations

Puts [array exists colour]

Puts [array names color](returns a list of the index strings)

foreach item [array names color]

{

puts "\$item is \$color(\$item)"

} (iterating through array)

set lstColor [array get color] (convert array to list)

array set color \$lstColor (convert list to array)

# Regular expressions

## ➤ Regsub

```
set stmt "Fan is one of Shania's fans"
```

```
regsub -nocase "fan" $stmt "Kristy" newStmt
```

*?switches? exp string subSpec ?varName?*

```
puts "$newStmt"
```

```
regsub -nocase -all "fan" $stmt "Kristy" newStmt
```

```
puts "$newStmt"
```

## ➤ Regexp

(returns 1 if the regular expression matches the string, else returns 0)

```
puts [regexp -nocase "fan" $stmt]
```

*?switches? regexp string*

## ➤ Format

```
puts [format "%s is a %d-year-old" Fan 26]
```

*formatString ?arg arg ...?*

# String operations

Set statement " Fan is a student "

Set statement [string **trim** \$statement]

Puts [string **length** \$statement]

Puts [string **length** statement]

Puts [string **index** \$statement 4]

Puts [string **index** \$statement **end**]

Puts [string **first** "is" \$statement]

(string **last**)

Puts [string **first** \$statement "is"]

Puts [string **range** \$statement 4 end]

Puts [string **replace** \$statement 9 end "professor"]

Puts [string **match** "\*student" \$statement] (\* ? [])

# File operations

```
set fRead [open source.txt r]
set fWrite [open target.txt w]
while {[eof $fRead]}
{
    set strLine [gets $fRead] ;#or gets $fRead strLine
    regsub -nocase -all "fan" $strLine "kristy" strLine
    puts $fWrite $strLine
}
close $fRead
close $fWrite
##### source.txt #####
Fan is a CSE student.
Fan is also one of Shania's fans.
Kristy and Fan are classmates.
```

# Miscellaneous commands

- **Eval:** Execute a command dynamically built up in your program.

```
set Script
```

```
{
```

```
    set Number1 17
```

```
    set Number2 25
```

```
    set Result [expr $Number1 + $Number2]
```

```
}
```

```
eval $Script
```

- **Exec:** execute external programs.

# Debugging your program

- Use puts statements (with update and after when using wish84.exe to run program)
- Tk\_messageBox: pop up a message box
- Tk\_messageBox –message "run to here" –type ok
- Tclpro
- **Trace variable** variable Name operation procedure



# Common pitfalls

- Missing \$ or extraneous \$
- Using {a} vs "a" vs [list a]
- Creating list items that are empty lists
- a b {} d

# Maze Tcl example

## Pseudocode:

- create a path which just has the start state
- make this path the only member of the list of alternatives to be explored
- while list of alternatives is not empty and not done
  - set firstpath to be the first path from the list of alternatives
  - update alternatives so it doesn't include the first path
  - set last to be the last member of firstpath
  - for each cell connected to the last member
  - create newpath with cell at the end of firstpath
  - if cell is 16
  - display path
  - else
  - add newpath to end of list of alternatives

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

# Maze Tcl example

```
set bDone 0
set path [list 1]
set alternatives [list $path]
while {[llength $alternatives] > 0 && !$bDone} {
    set firstpath [lindex $alternatives 0]
    set alternatives [lrange $alternatives 1 end]
    set last [lindex $firstpath end]
    foreach cell $connected($last) {
        set newpath [linsert $firstpath end $cell]
        if {$cell == 16} {
            puts "Answer is $newpath"
            set bDone 1
            break
        }
        update
        after 1000
    } else {
        lappend alternatives $newpath
    }
}
}
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

# UNIT-V

## Python

# Python

- Open source general-purpose language.
- Object Oriented, Procedural, Functional
- Easy to interface with C/ObjC/Java/Fortran
- Easy-ish to interface with C++ (via SWIG)
- Great interactive environment

# Python

- Interpreted language: work with an evaluator for language expressions (like DrJava, but more flexible)
- Dynamically typed: variables do not have a predefined type
- Rich, built-in collection types:
  - Lists
  - Tuples
  - Dictionaries (maps)
  - Sets

# Language features

- Indentation instead of braces
- Several sequence types
- Strings ‘...’: made of characters, immutable
- Lists [...]: made of anything, mutable
- Tuples(...): made of anything, immutable
- Powerful collection and iteration abstractions

# Language features

- Powerful subscripting (slicing)
- Functions are independent entities (not all functions are methods)
- Exceptions as in Java
- Simple object system
- Iterators(like Java) and generators



# Comments

- Everything after "#" on a line is ignored. No block comments, but doc strings are a comment in quotes at the beginning of a module, class, method or function.
- Also, editors with support for Python often provide the ability to comment out selected blocks of code, usually with "##".

# Python Basic Syntax

- Unlike other languages, Python does not use an end of line character.
- To end a statement in Python, you do not have to type in a semicolon or other special character; you simply press Enter. For example, this code will generate a syntax error:
  - `message = 'Hello World!'` This will not:
  - `message = 'Hello World!'`

# Names and tokens

- Allowed characters: a-z A-Z 0-9 underscore, and must begin with a letter or underscore.
- Names and identifiers are case sensitive.
- Identifiers can be of unlimited length.
- Special names, customizing, etc. Usually begin and end in double underscores.

# Names and tokens

- Special name classes Single and double underscores.
- Single leading single underscore Suggests a "private" method or variable name. Not imported by "from module import \*".
- Single trailing underscore Use it to avoid conflicts with Python keywords.
- Double leading underscores Used in a class definition to cause name mangling (weak hiding).

# Keywords

<b>and</b>	<b>None</b>	<b>yield</b>	<b>def</b>
as	False	not	nonlocal
assert	finally	or	with
async	for	pass	lambda
break	exec	print	del
await	from	raise	while
continue	global	raise	is
class	if	return	elif
except	import	True	try
else	in		

# Data Types

➤ The built-in variable types are the most important basic types:

- Integer (short and long)
- Strings
- Booleans
- floating point (float)
- Lists and tuples
- Dictionaries

# Basic Operators

➤ Python language supports the following types of operators

- ☐ Arithmetic Operators
- ☐ Comparison Operators
- ☐ Assignment Operators
- ☐ Logical Operators
- ☐ Bitwise Operators
- ☐ Membership Operators
- ☐ Identity Operators

# Type Conversion

- Data can sometimes be converted from one type to another.
  - Ex: The string `—3.0` is equivalent to the floating point number 3.0, which is equivalent to the integer number 3
- Functions exist which will take data in one type and return data in another type.
- `Int()`-Converts compatible data into an integer. This function will truncate floating point numbers



# Type Conversion

- `Float()`-Converts compatible data into a float.
- `Str()`-Converts compatible data into a string.
  - ❑ Examples: `int(3.3)` produces 3
  - ❑ `str(3.3)` produces `—3.3`
  - ❑ `float(3)` produces 3.0
  - ❑ `float(—3.5)` produces 3.5
  - ❑ `int(—7)` produces 7
  - ❑ `int(—7.1)` throws an ERROR!
  - ❑ `float(—Test)` Throws an ERROR!

# Statements (Decision Making)

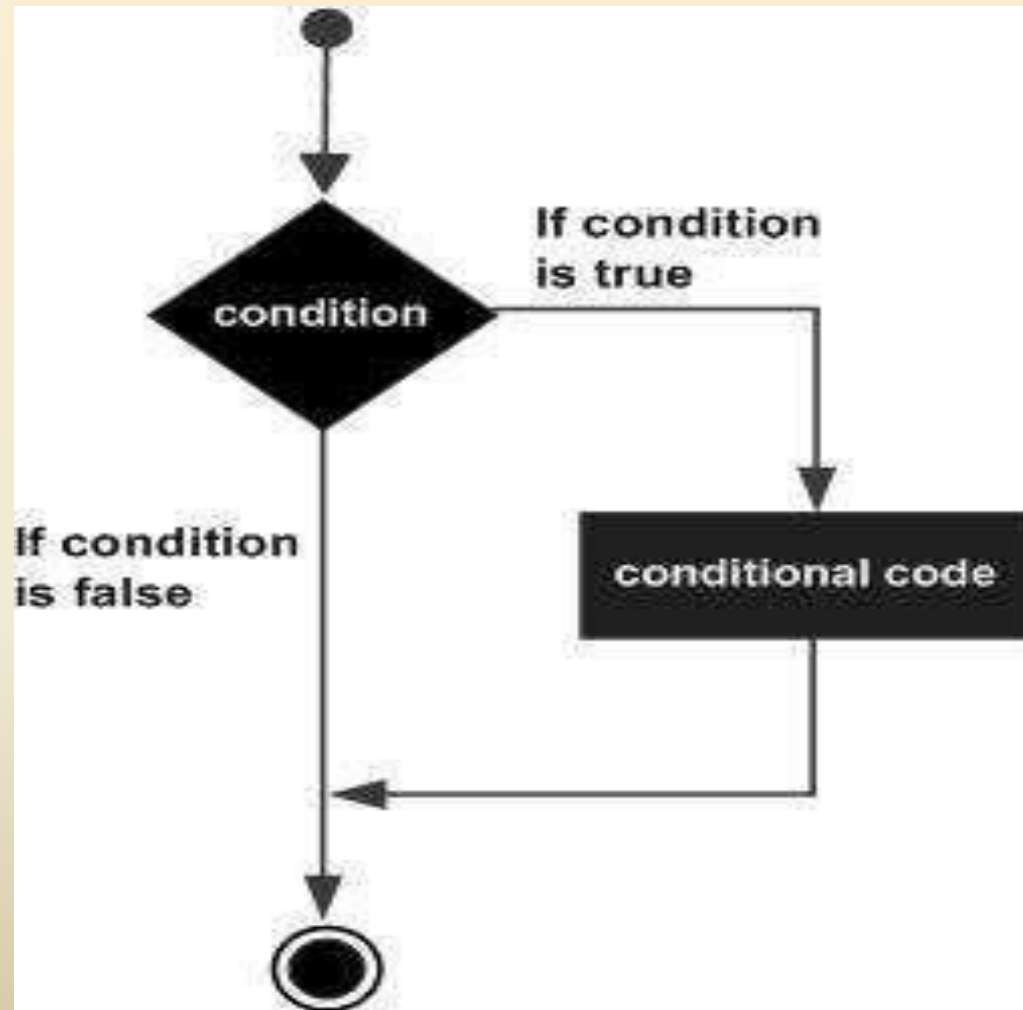
- Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.
- Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome.
- We have to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.
- Python programming language assumes any non-zero and non-null values as TRUE, and any zero or null values as FALSE value.

# If Statement

- An if statement consists of a Boolean expression followed by one or more statements.
- Syntax

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

# Flow Diagram



# IF...ELIF...ELSE Statements

- An else statement can be combined with an if statement. An else statement contains a block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.
- The else statement is an optional statement and there could be at the most only one else statement following if.

# Syntax of if...else

➤ The syntax of the **if...else** statement is if

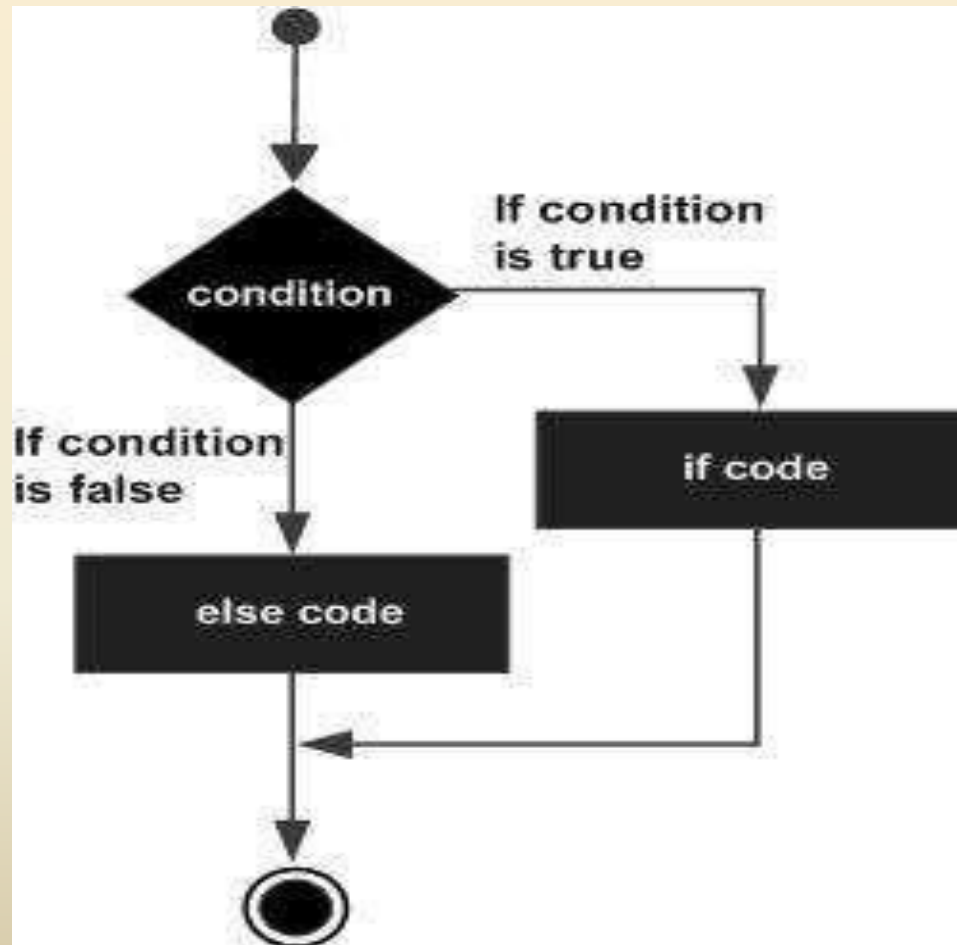
    □ expression:

        statement(s)

else:

    statement(s)

# Flow Diagram



# The elif Statement

- ◆ The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.
- ◆ Similar to the else, the elif statement is optional. However, unlike else, for which there can be at the most one statement, there can be an arbitrary number of elif statements following an if.



# Syntax



if expression1:

statement(s)

elif expression2:

statement(s)

elif expression3:

statement(s)

else:

statement(s)

# Nested IF Statement

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

# Syntax

➤ If expression1:  
    statement(s)  
if expression2:  
    statement(s)  
elif expression3:  
    statement(s)  
else  
    statement(s)  
elif expression4:  
    statement(s)  
else:  
    statement(s)

# Nested IF Statements

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

# Loops

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

# Loops

- A loop statement allows us to execute a statement or group of statements multiple times.
- The following diagram illustrates a loop statement.
- Python programming language provides the following types of loops to handle looping requirements.

# While Loop

- Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
- A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- The syntax of a while loop in Python programming language is

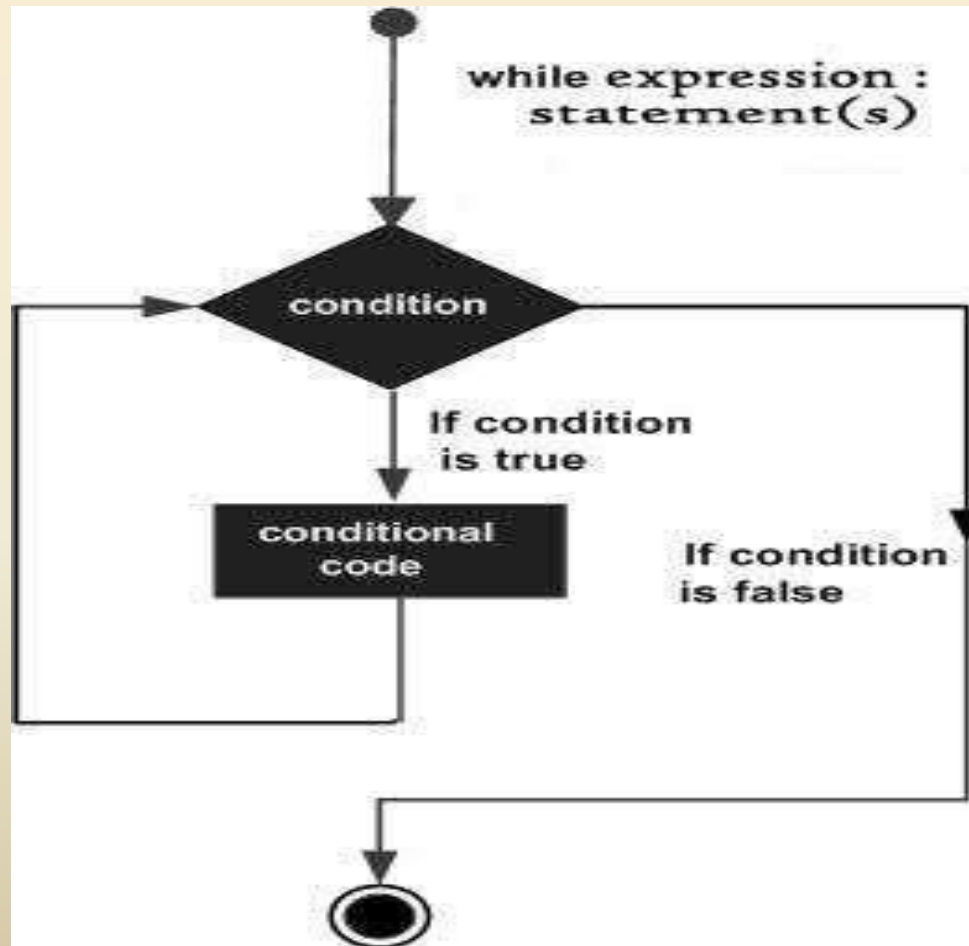
```
while expression:  
statement(s)
```
- Here, statement(s) may be a single statement or a block of statements with uniform indent.

# While Loop

- The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code.
- Python uses indentation as its method of grouping statements.



# Flow Diagram



# Using else Statement with Loops

- Python supports having an else statement associated with a loop statement.
- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

# For Loop

- The for statement in Python has the ability to iterate over the items of any sequence

## Syntax

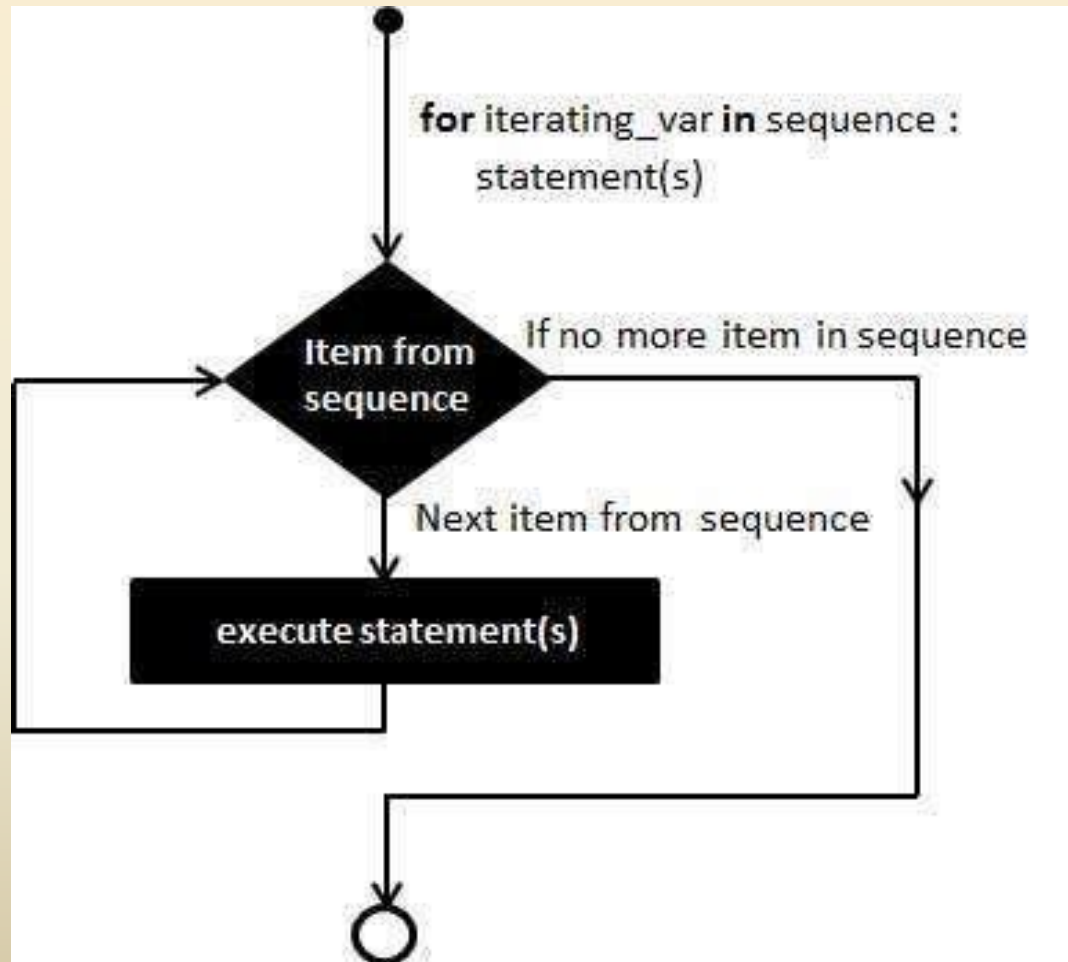
```
for iterating_var in sequence:  
    statements(s),
```

- If a sequence contains an expression list, it is evaluated first.

# For Loop

- Then, the first item in the sequence is assigned to the iterating variable `iterating_var`.
- Next, the statements block is executed.
- Each item in the list is assigned to `iterating_var`, and the statement(s) block is executed until the entire sequence is exhausted.

# Flow Diagram



# Functions in Python

- A function is a block of organized, reusable code that is used to perform a single, related action.
- Functions provide better modularity for your application and a high degree of code reusing.
- As you already know, Python gives you many built-in functions like print, etc. but you

# Defining a Function

- Function blocks begin with the keyword `def` followed by the function name and parentheses `()`.
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement the documentation string of the function or *docstring*.

# Defining a Function

- The code block within every function starts with a colon : and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

- **Syntax**

```
def functionname( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```



# Calling A Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

# Types of functions

➤ One way to categorize functions in Python is:

1. Built-in functions: these functions pre-defined and are always available.
2. Functions defined in modules: these functions are pre-defined in particular modules and can only be used when the corresponding module is imported.
3. User defined functions: these are defined by the programmer.

# Built-in Functions

Method	Description
<a href="#"><u>abs()</u></a>	returns absolute value of a number
<a href="#"><u>all()</u></a>	returns true when all elements in iterable is true
<a href="#"><u>any()</u></a>	Checks if any Element of an Iterable is True
<a href="#"><u>ascii()</u></a>	Returns String Containing Printable Representation
<a href="#"><u>bin()</u></a>	converts integer to binary string
<a href="#"><u>bool()</u></a>	Coverts a Value to Boolean
<a href="#"><u>Python bytearray()</u></a>	returns array of given byte size
<a href="#"><u>bytes()</u></a>	returns immutable bytes object
<a href="#"><u>callable()</u></a>	Checks if the Object is Callable
<a href="#"><u>chr()</u></a>	Returns a Character (a string) from an Integer

# Built-in Functions

Method	Description
<a href="#"><u>complex()</u></a>	Creates a Complex Number
<a href="#"><u>delattr()</u></a>	Deletes Attribute From the Object
<a href="#"><u>dict()</u></a>	Creates a Dictionary
<a href="#"><u>dir()</u></a>	Tries to Return Attributes of Object
<a href="#"><u>divmod()</u></a>	Returns a Tuple of Quotient and Remainder
<a href="#"><u>enumerate()</u></a>	Returns an Enumerate Object
<a href="#"><u>eval()</u></a>	Runs Python Code Within Program
<a href="#"><u>exec()</u></a>	Executes Dynamically Created Program
<a href="#"><u>filter()</u></a>	constructs iterator from elements which are true
<a href="#"><u>float()</u></a>	returns floating point number from number, string

# Methods in Python

- Methods are just like functions, with two differences:
- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.
- Each method is associated with a class and is intended to be invoked on instances of that class.

# Modules in python

- A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

# Modules in python

## ➤ Example

➤ The Python code for a module named a name normally resides in a file named `aname.py`.

➤ Here is an example of a simple module, `support.py`

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

# Import Statement

- You can use any Python source file as a module by executing an import statement in some other Python source file.
- The import has the following syntax `import module1[, module2 [... moduleN]`
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module `hello.py`, you need to put the following command at the top of the script



# From...import Statement

- Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax

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

# From...import \* Statement

- It is also possible to import all the names from a module into the current namespace by using the following import statement from
- From module name import \*
- This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

# Executing Modules as Scripts

- Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.
- The code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`.

# Locating Modules

- When you import a module, the Python interpreter searches for the module in the following sequences
  - ☐ The current directory.
  - ☐ If the module is not found, Python then searches each directory in the shell variable `PYTHONPATH`.
  - ☐ If all else fails, Python checks the default path. On UNIX, this default path is normally `/usr/local/lib/python3/`.

# Locating Modules

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

# Exception Handling

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception.
- An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

# Handling an Exception

- If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a `try: block`.
- After the `try: block`, include an `except: statement`, followed by a block of code which handles the problem as elegantly as possible.

# Syntax

- Here is simple syntax of try....except...else blocks

Try:

You do your operations here

.....

except *ExceptionI*:

If there is ExceptionI, then execute this block.

except *ExceptionII*:

If there is ExceptionII, then execute this block.

.....

else:

If there is no exception then execute this block.



# Syntax

- A single try statement can have multiple except statements.
- This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause.  
The code in the else block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

# Web Applications in Python

- The idea behind creating a Python-driven web application is that you can use Python code to determine what content to show a user and what actions to take.
- The code is actually run by the web server that hosts your website, so your user doesn't need to install anything to use your application; if the user has a browser and an Internet connection, then everything else will be run online.

# Web Applications in Python

- Google App Engine
- Static Web App
- WSGI Application
- Dynamic Web App

# Web Application Framework

- A web framework is nothing but a collection of packages and modules that allows easier development of websites.
- It handles all low-level communication within the system and hides it from you to make no issues for performing common tasks for the development.
- Popular Python frameworks like Pyramid and Django are used by companies like Bitbucket, Pinterest, Instagram and Dropbox in their web application development..

# Web Application Framework

- So, it is safe to say these frameworks are able to handle almost everything you throw at them web frameworks are meant to hide and handle all low-level details.
- so that you as a developer, do not have to dig deep into how everything works when you are developing a web-enabled application.

# Web Application Framework

- One of the most important advantages of using a web framework as opposed to building something on your own is handling the security of your website.
- Since web frameworks have been used and backed by thousands, it inherently handles security, preventing any misuse of the web application.
- Good frameworks are built ensuring scalability from the very beginning of the development process.
- So, whenever you are planning to scale your website by adding a new component or using a new database, web frameworks are more likely to scale better than what you come up with when building from scratch.

# Web Frameworks In Python

- There are tons of Python web frameworks, and every framework has their own strengths and weaknesses.
- Thus, it is necessary to evaluate your project requirements and pick one the best one from the collection.
- Below are the three most popular web frameworks in Python.

# Django

- Django is probably the most popular Python web framework and is aimed mostly at larger applications.
- It takes a —batteries-included‖ approach and contains everything needed for web development bundled with the framework itself.
- So, you do not have to handle things like database administration, templating, routing, authentication and so on.
- With fairly less code, you can create great applications with Django.
- If you are building a mid-high ranged web applications and are quite comfortable with Python, you should go for Django.



# Pyramid

- The Pyramid is the most flexible Python web framework and just like Django, it is aimed at mid-high scale applications.
- If you think Django brings too much bloat to your web application, use Pyramid.
- It does not force you to use a single solution for a task, but rather gives you a pluggable system to plug-in according to your project requirements.
- You do have the basic web development capabilities like routing and authentication, but that is about it.
- So, if you want to connect to a database for storage, you ought to do that yourself using external libraries.

# Flask

- Flask is the new kid in town. Unlike Pyramid and Django, Flask is a micro-framework and is best suited for small-scale applications.
- Even if it is new, Flask has integrated great features of other frameworks.
- It includes features like unit testing and built-in development server that enable you to create reliable and efficient web applications.