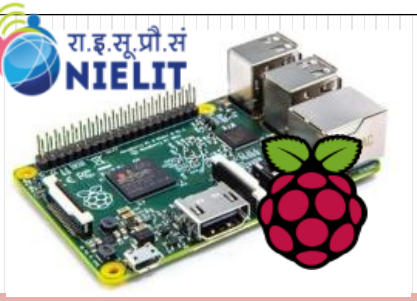


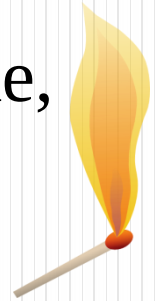
Linux shell programming for Raspberry Pi Users - 2

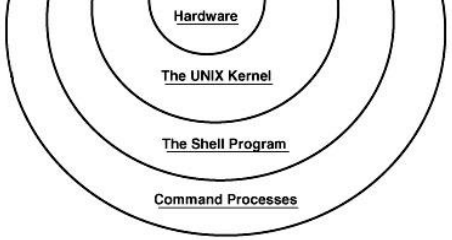
Sarwan Singh

Assistant Director(S)

NIELIT Chandigarh

Education is the kindling of a flame,
not the filling of a vessel.
- *Socrates*

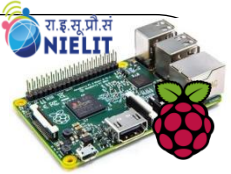




SHELL PROGRAMMING

"A shell script is a computer program designed to be run by the Unix shell, a command line interpreter."

Shell scripts allow us to program commands in chains and have the system execute them as a scripted event, just like batch files. They also allow for far more useful functions, such as **command** substitution.

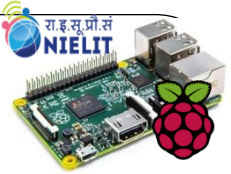


HOME

The HOME variable contains the path to your home directory.

When you use cd command with no arguments, the command uses the value of the HOME variable as the argument.

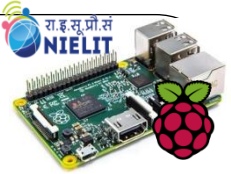
```
echo $HOME
    /home/pi
```



Metacharacters

- Shell metacharacters are characters that are handled specially by the shell.

- Below is an (incomplete) list of shell metacharacters
  - > >> < << | Command redirection, pipe
  - * [] ? {} File specification
  - ; & || && () Command Sequencing
  - # Line comment

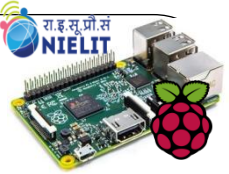


Metacharacters - quoting

- Single character escape - \ (backslash)
  - Character immediately following the backslash is treated literally
  - To remove a file named “#bogus”:

  rm \#bogus
- Weak quotes (“) inhibit
  - Space (token separator)
  - Filename expansion (wildcards)
- Strong quotes (') inhibit pretty much everything

Aliases



- You can create aliases for shell builtins and disk utilites
  - Common use is to add options to the default behavior of commands (e.g. ls, rm, mv, …)
- E.g.: alias rm='rm -i'
- To see current aliases, use the alias command

Alias (cont)

- Create an alias using the builtin command
  - alias *name*[=*value*]
    - You should usually enclose *value* in quotes.
    - no spaces around the '='
  - Without a value, *alias* prints out the alias associated with *name*

alias dir="ls"

alias ls="ls -CF"

dir

Output same as for "ls -CF"

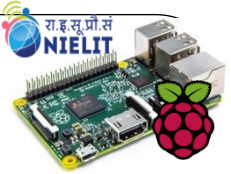
Variable Substitution

- Shells support the use of variables
- A variable is a name that is bound to a value.
- To set a variable (in Bash) just enter

  name=value

  - No spaces around '='
- To see the settings of all variables just enter "set".
- To kill a variable:

  unset *name*



Variable Substitution (cont)

- When processing a command, variable substitution occurs.
- A variable in a command is flagged by a dollar sign prefix "$"

```
$ echo My shell is $SHELL
```

My shell is /bin/bash

  - *echo* Writes out its arguments after substitution is performed.

Variable Substitution (cont)

- Variable substitution will occur within double quotes

```
$ echo "My shell is $SHELL"
My shell is /usr/local/bin/tcsh
```

- Substitution does not occur within single quotes.

```
$ echo 'My shell is $SHELL'
My shell is $SHELL
```

Variable Substitution (cont)

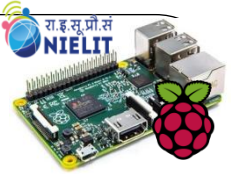
- When the usage of a variable name is not clear, enclose it within braces ${*name*}

```
$ prefix=cs265
$ suffix=.pdf
$ echo $prefix03$suffix
.pdf
$ echo ${prefix}03${suffix}
cs26503.pdf
```

- This occurs when constructing file names in script files.

Variable Substitution (cont)

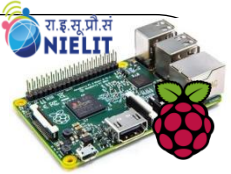
- Many programs use shell variables (also called environmental variables) to get configuration information.
- Examples
  - PRINTER is used by printing commands to determine the default printer.
  - TERM is used by programs (e.g., vi, pine) to determine what type of terminal is being used.
  - VISUAL is used by network news programs, etc., to determine what editor to use.



Command Substitution

Command substitution allows the output (stdout) of a command to replace the command name. There are two forms:

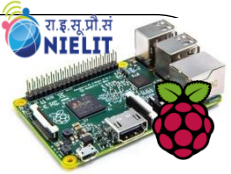
- The original Bourne:
  `` `command` ``
- The Bash (and Korn) extension:
  `$(command)`



Command-substitution

- The output of the command is substituted in:

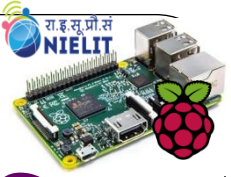
```
$ echo $(ls)
foo fee file?

$ echo "Today is $(date '+%A %d %B %Y')"
Today is Thursday 30 September 2010
```



Strong quoting – Single quotes

- Inhibits **all** substitution, and the special meaning of metacharacters:

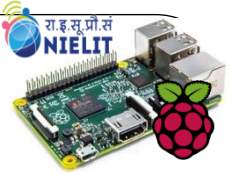
```
$ echo '$USER is $USER'
$USER is $USER
$ echo 'today is `date`'
today is `date`
$ echo 'No background&'
No background&
$ echo 'I said, "radio!"'
I said, "radio"
```



Weak quoting – double quotes

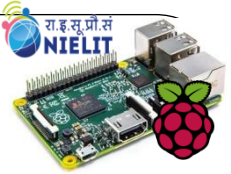
- Allows command and variable substitution
- Inhibits special meaning of all other metacharacters

```
$ echo "My name is $USER &"
My name is kschmidt &
$ echo "\$2.00 says `date`"
$2.00 says Sun Jan 15 01:43:32
EST 200
```



Command Execution

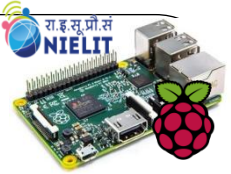
- Sometimes we need to combine several commands.
- There are four formats for combining commands into one line
  - Sequenced
  - Grouped
  - Chained
  - Conditional



Chained Commands

- Several commands on the same line
- Separated by semicolons
- There is no direct relationship between the commands.

```
command1; command2; command3
```

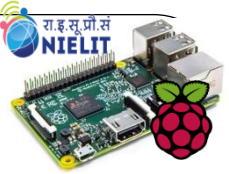


Grouped Commands

- If we apply the same operation to the group, we can group commands
- Commands are grouped by placing them into parentheses
- Commands are run in a subshell

Example:

```
echo "Month" > file; cal 10 2000 >> file

(echo "Month" ; cal 10 2000 ) > file
```



Conditional Commands

We can combine two or more commands using conditional relationships AND (&&) and OR (||).

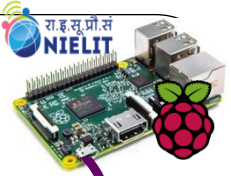
If we AND two commands, the second is executed only if the first is successful.

If we OR two commands, the second is executed only of the first fails.

```
cp file1 file2 && echo "Copy successful"

cp file1 file2 || echo "Copy failed"
```

Shell Syntax

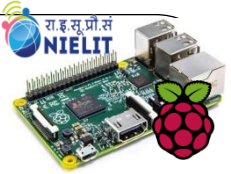
- Comments
  - # This is a comment
  - ls # list the files in the current directory
- Line continuation
  - echo A long \
  - > line
- ; #Command separator – you can list more than one command per line separated by ;
  - ls ; who
- / #Pathname separator
  - cd /home/jjohnson



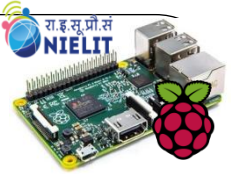
Shell Syntax – wildcards (globbing)

- Wildcards, and pathname expansion
  - * # match any string (including empty)
  - ? # match any single character
  - [set] # match characters listed in set (can be range)
  - [!set] # match any character not given in set
- Examples
  - ls *.c
  - ls *.?
  - ls *.[Hh][Tt][Ll]
  - ls [a-z]

Shell Syntax – redirection and pipes

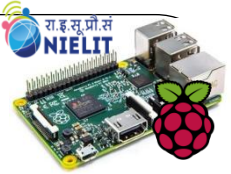


- File redirection and pipes
  - < # redirect input from specified source
  - > # redirect output to specified source
  - >> # redirect output and append to specified source
  - | # pipe the output from one command to the input to the next
- Examples
  - grep word < /usr/dict/words
  - ls > listing
  - ls >> listing
  - ls -l| wc -l



Shell Syntax - stderr

- Distinct from stdout
- Also goes to screen, by default
- stdout is designated by the file descriptor 1 and stderr by 2 (standard input is 0)
  - To redirect standard error use 2>
  - ls filenothere > listing 2> error
  - ls filenothere 2>&1 > listing # both stdout and stderr redirected to listing



Shell Syntax – bg jobs

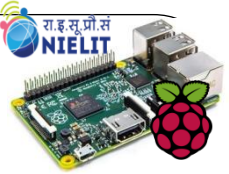
- Background jobs
  - `&` # run command in the background

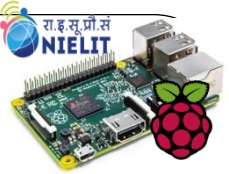
```
grep 'we.*'< /usr/word/dict > wewords &
```

    - This runs the grep command in the background – you immediately get a new prompt and can continue your work while the command is run.
  - `jobs` – Builtin. Lists active jobs (stopped, or running in the background). Also see the command `ps`
  - `kill` will take a PID (see `ps`) or jobspec (see `jobs`)

What Is A Script?

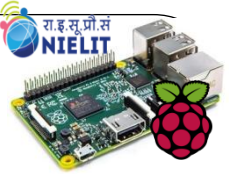
- A script is a small program that is executed by the shell.
- The script is a text file which will contain:
  - Shell commands you normally use.
  - Shell flow control constructs (e.g., if-then-else, etc.)
  - A heavier use of variables than you normally would use from the command line.





Why Write Scripts?

- Sequences of operations which you perform often can be placed into a script file and then executed like a single command.
  - For example, renaming all files of the form cs265l*.ppt to cs265l*n.ppt requires a *mv* command for each file.
- The Unix shells are very powerful, but there are some things that they do not do well.



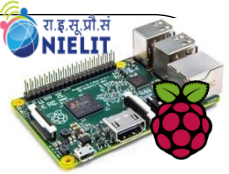
Shell scripting - Why Not ?

resource-intensive tasks, especially where speed is a factor

complex applications, where structured programming is a necessity

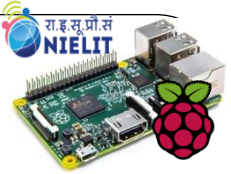
mission-critical applications upon which you are betting the ranch, or the future of the company

situations where security is important, where you need to protect against hacking



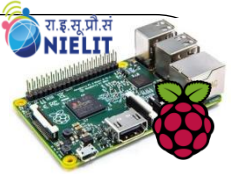
Why Not ? (cont.)

- Project consists of subcomponents with interlocking dependencies
- Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion)
- Need to generate or manipulate graphics or GUIs



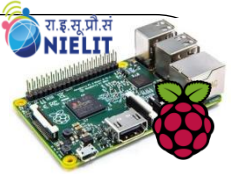
Why Not ? (cont.)

- Need direct access to system hardware
- Need port or socket I/O
- Need to use libraries or interface with legacy code



Very Simple Script

```
#!/bin/sh
echo Hello World
```



Example: Simple Script

- Let's create a shell script to give us information about the system.
- We create the script using a text editor.
  - Let's call the file "status"

```
#!/bin/bash
uptime
users # maybe hit [Enter] here
```

  - Exit the editor

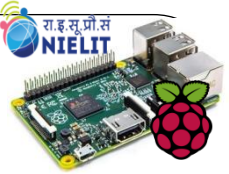
Running a Script

- To execute the shell we can do

```
$ bash status
10:37 up 23 days, 23:54, 14 users, load
  average …
afjhj billc …
```

- We can also execute the file as a command if the appropriate execute access is granted.

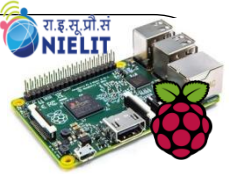
```
$ ./status
bash: ./status: Permission denied
$ chmod +x status
$ ./status # Works correctly.
```



Conditional Expressions

- To perform *ifs* and *whiles* we need to be able to construct conditional expressions.
- A conditional expression is one that evaluates to true or false depending on its operands
- A process' return value of 0 is taken to be *true* ; any nonzero value is *false*

Control statement

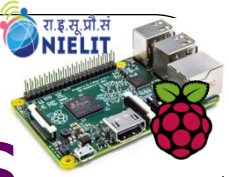


```
# First form

if condition ; then
    commands
fi
```

```
# Second form
if condition ; then
    commands
else
    commands
fi
```

```
#Third form
if condition ; then
    commands
elif condition ; then
    commands
fi
```

`test` – Conditional Expressions

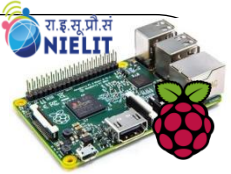


- Actually a disk utility
- [is just shorthand
- Provides for a great many tests
- Is available to all shells

test *expression*

Or

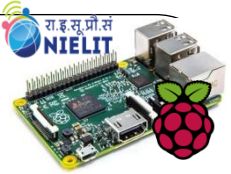
[*expression*] *Separate expression from brackets spaces*



Cond. Expressions (cont)

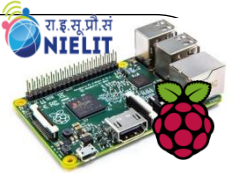
- **test** returns an exit status of zero (success) for *true*
- **test** uses a variety of operators
  - Unary file operators can test various file properties. Here are just a few:
    - -e True if file exists
    - -f True if file is a regular file
    - -d True if file is a directory
    - -w True if file exists and is writable
    - -O True if I own the file
  - E.g.

```
if [ -e ~kschmidt/public_html ] ; then
  echo "Kurt has a public web directory"
fi
```



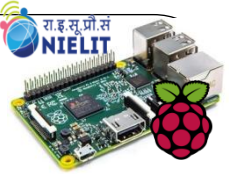
Expression	Description
-d file	True if file is a directory.
-e file	True if file exists.
-f file	True if file exists and is a regular file.
-L file	True if file is a symbolic link.
-x file	True if file is a file executable by you.
file1 -nt file2	True if file1 is newer than (according to modification time) file2
file1 -ot file2	True if file1 is older than file2
-z string	True if string is empty.
-n string	True if string is not empty.
string1 = string2	True if string1 equals string2.
string1 != string2	True if string1 does not equal string2.

[] – file and string operators



- Binary file operators "*file1 op file2*"
  - -nt True is *file1* is newer than *file2*
  - -ot True is *file1* is older than *file2*
  - -ef True if *f1* and *f2* refer to the same inode
- Unary string operators "*op string*"
  - -z True if string is of zero length
  - -n True if string is not of zero length
  - -l *Returns length of string*
- E.g.

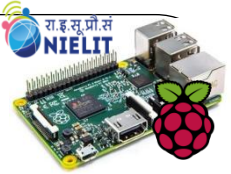
```
if [ -z "$myVar" ] ; then
  echo "\$myVar has null length"
fi
```



[] – string operators

- These compare lexical order
  - == != < > <= >=
  - Note, < > are file redirection. Escape them
- E.g.

```
if [ "abc" != "ABC" ] ; then
  echo 'See.  Case matters.' ; fi
if [ 12 \< 2 ] ; then
  echo "12 is less than 2?" ; fi
```



[] – arithmetic operators

- Only for integers
- Binary operators:

  `-lt  -gt  -le  -ge  -eq  -ne`

- E.g.

```
if [ 2 -le 3 ] ; then ;echo "cool!" ; fi
x=5
if [ "$x" -ne 12 ] ; then
  echo "Still cool" ; fi
```

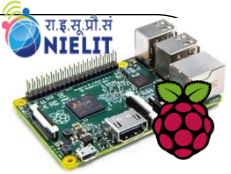
[] – Logical Operators

- Logical expression tools
  - ! *expression* Logical not (I.e., changes sense of expression)
  - *e1* -a *e2* True if both expressions are true.
  - *e1* -o *e2* True if *e1* or *e2* is true.
  - \(*expression* \) Works like normal parentheses for expressions; use spaces around the expression.

Examples:

`test -e bin -a -d /bin` is true

```
[ -e ~/.bashrc -a ! -d ~/.bashrc ] && echo true
```

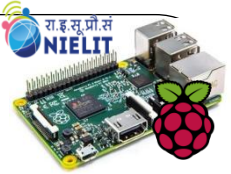


[[*test*]]

- Bash added [[]] for more C-like usage:

```
if [[ -e ~/.bashrc && ! -d ~/.bashrc ]]
then
echo "Let's parse that puppy"
fi


if [[ -z "$myFile" || ! -r $myFile ]]
…
```

- It's a built-in
- Why sometimes quote $myFile, sometimes not (it's usually a good idea to do so)?



Arithmetic Expressions

- Bash usually treats variables as strings.
- You can change that by using the arithmetic expansion syntax: ((*arithmeticExpr*))
- (()) shorthand for the **let** builtin statement

```
$ x=1
$ x=x+1 # "x+1" is just a string
echo $x
x+1
```

- Note, $[] is deprecated

Arithmetic Expression (cont)

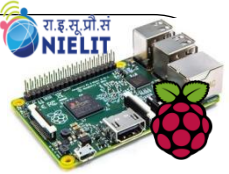
```
$ x=1
$ x=$x+1 # still just a string
$ echo $x
1+1
```

- Closer, but still not right.

```
$ x=1
$ (( x=x+1 ))
$ echo $x
2
```

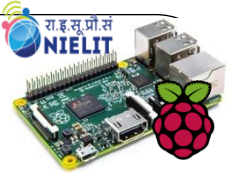
- Finally!

Sample: Basic conditional example if .. then



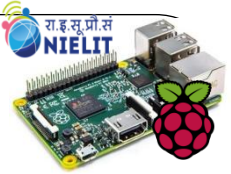
```
#!/bin/bash
if [ "$1" = "foo" ] ; then
   echo expression \
evaluated as true
fi
```

Sample: Basic conditional example if .. then ... else




```
#!/bin/bash
if [ "$1" = "foo" ]
then
   echo 'First argument is "foo"'
else
   echo 'First arg is not "foo"'
fi
```

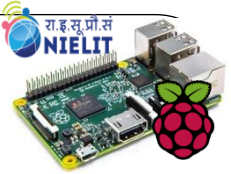
Sample: Conditionals with variables



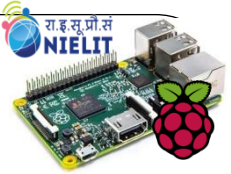
```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" == "$T2" ] ; then
  echo expression evaluated as true
else
  echo expression evaluated as false
fi
```

- Always quote variables in scripts!

Checking return value of a command



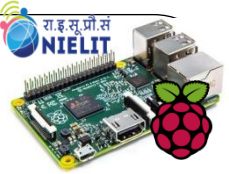

```
if diff "$fileA" "$fileB" > /dev/null
then
  echo "Files are identical"
else
  echo "Files are different"
fi
```



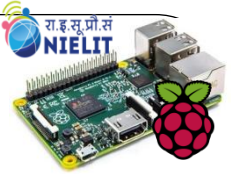
Case Statement

```
case $opt in
    a  ) echo "option a";;
    b  ) echo "option b";;
    c  ) echo "option c";;
    \? ) echo \
     'usage: alice [-a] [-b] [-c] args...'
          exit 1;;
esac
```

Control statement



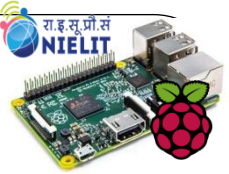
```
case word in
        patterns ) statements ;;
esac

echo -n "Enter a number between 1 and 3 inclusive > " read
   character
case  $character in
   1 ) echo "You entered one." ;;
   2 ) echo "You entered two." ;;
   3 ) echo "You entered three." ;;
   * ) echo "You did not enter a number"
       echo "between 1 and 3."
esac
```

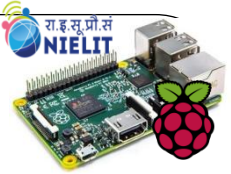



```
echo -n "Type a digit or a letter > "
read character
case $character in
        # Check for letters
   [a-z] | [A-Z] ) echo "You typed letter $character" ;;
        # Check for digits
   [0-9] ) echo "You typed the digit $character" ;;
        # Check for anything else
   *) echo "You did not type a letter or a digit"
esac
```

Special Variables

- $# the number of arguments
- $* all arguments
- $@ all arguments (quoted individually)
- $? return value of last command executed
- $$ process id of shell
- $HOME, $IFS, $PATH, $PS1, $PS2





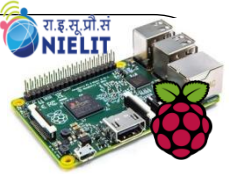
Scripts and Arguments

- Scripts can be started with parameters, just like commands

  aScript arg1 arg2 …

- The scripts can access these arguments through shell variables:
  - "$*n*" Is the value of the n^{th} paramter.
    - The command is parameter zero
  - "$#" Is the number of parameters entered.
  - "$*" Expands as a list of all the parameters entered except the command.

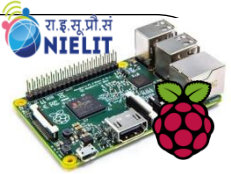
Scripts and Parameters (cont)



- Let's quickly write a script to see this:
  - (this first line is a quick and dirty way to write a file)

```
$ cat > xx # cat reads from stdin if no
  file specified
echo $0
echo $#
echo $1 $2
echo $*
C-d # Control-D is the end of file
  character.
$ chmod +x xx
```

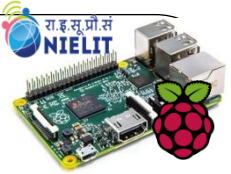
    - The file xx is now an executable shell script.



Loops for, while and until

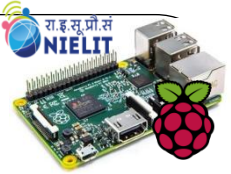
- The for loop is a little bit different from other programming languages. Basically, it let's you iterate over a series of 'words' within a string.
- The while executes a piece of code if the control expression is true, and only stops when it is false (or a explicit break is found within the executed code.
- The until loop is almost equivalent to the while loop, except that the code is executed while the control expression evaluates to false.

```
function press_enter
{
    echo ""
    echo -n "Press Enter to
    continue"
    read
    clear
}
selection=
until [ "$selection" = "0" ]; do
    echo "PROGRAM MENU"
    echo "1 - display free disk space"
    echo "2 - display free memory"
    echo "0 - exit program"
    echo -n "Enter selection: "
    read selection
    case $selection in
        1 ) df ; press_enter ;;
        2 ) free ; press_enter ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0";
    press_enter
    esac
done
```



Flow Control

```
for variable in words; do
    statements
done

for i in word1 word2 word3; do
    echo  $i
done
```



For samples

- Example

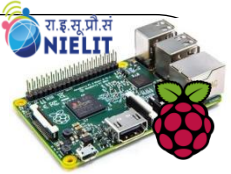
```
$ for x in 1 2 a; do
>   echo $x
> done
```

```
1
2
a
```

- Example

```
$ for x in *; do
>  echo $x
> done
```

```
bin
mail
public_html
…
```



For samples

```
#!/bin/bash
for i in $(cat list.txt) ; do
  echo item: $i
done
```

```
#!/bin/bash
for (( i=0; i<10; ++i )) ; do
  echo item: $i
done
```

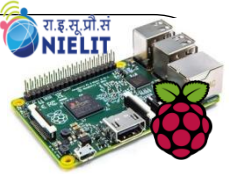
While sample

```
COUNTER=0
while [  $COUNTER -lt 10 ] ; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

```
COUNTER=0
while ((  COUNTER < 10 )) ; do
   echo The counter is $COUNTER
   (( COUNTER = COUNTER+1 ))
done
```

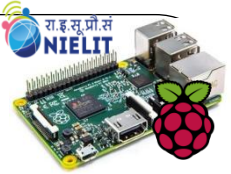
Until sample

```
#!/bin/bash
    COUNTER=20
    until [  $COUNTER -lt 10 ]
    do
          echo COUNTER $COUNTER
          let COUNTER-=1
    done
```



Loop Control

- `break` terminates the loop
- `continue` causes a jump to the next iteration of the loop



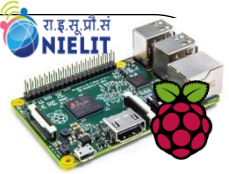
Debugging Tip

- If you want to watch the commands actually being executed in a script file, insert the line "set -x" in the script.

```
set -x
for n in *; do
  echo $n
done
```

  - *Will display the expanded command before executing it.*

```
+ echo bin
bin
+ echo mail
mail
```

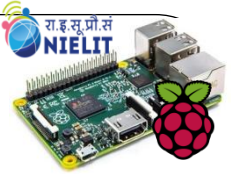


Functions

As in almost any programming language, you can use functions to group pieces of code in a more logical way or practice the divine art of recursion.

Declaring a function is just a matter of writing function my_func { my_code }.

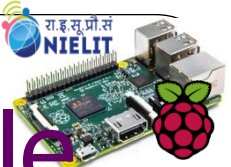
Calling a function is just like calling another program, you just write its name.



Local variables

```
#!/bin/bash
HELLO=Hello
function hello {
  local HELLO=World
    echo $HELLO
}

$ echo $HELLO
$ hello
$ echo $HELLO
```

Functions with parameters sample



```
#!/bin/bash

function quit {
    echo 'Goodbye!'
    exit
}
function hello {
  echo "Hello $1"
}


for name in Vera Kurt ;
do

  hello $name

done

quit
```

Output:

```
Hello Vera

Hello Kurt

Goodbye!
```