1. **Pipe:**
   - Denoted by "|".
   - Allows the user to run 2 or more commands consecutively.
   - The syntax is: command1|command2|...|commandn. Note that data flows left to right. This means that the output of command1 is the input for command2, and so on.
2. **Filter:**
   - A filter is an Unix command that reads from standard input, processes the input and writes on standard output.
     I.e. It takes some input, processes the input and produces some output based on the input.
   - Some Unix filter commands are:
     a. Grep
     b. Sort
     c. Uniq
     d. Cat
     e. More
     f. Head
     g. Tail
     h. Wc
3. **Shell Scripting:**
   - We use shell scripting because it helps us automate things.

   1) File Expansion:
   - ***** means 0 or more characters.
   - **?** means exactly 1 character.
   - **[x-y]** means one character in the range x to y, inclusive.
   - **[^oa]** means any character except o or a.
   - **~** means home directory.
   - **~u** means home directory of user u.
   - E.g.
     a. ls *.txt will list out all the text files in that directory.
     b. rm * will remove everything in that directory.
     c. cp ?? ~ will copy everything with 2 characters in that directory and put it into the home directory.

   2) Creating a shell script:
   a. Create a file using any editor and name the file with the extension **.sh**.
   b. Start the script with **#!/bin/bash**, or any of its variant. This is called the **shebang**, and is the path to your bash interpreter. It tells the shell how to interpret/execute the commands in this file.
      Note: The shebang is not the same on every machine. Use the which command to find the right one for your computer.
   c. Write some code. You can put any Unix commands in the script file.
   d. Save the file as filename.sh and execute the script using
      i. bash filename.sh

ii. ./filename.sh ← No Spaces!!!

Note: The file must be executable. Use the chmod command to change execute permission, if needed.

3) Variables and Comments:

a) A comment is denoted by #.

b) In any Unix system, including shell, there are 2 types of variables:

i) System Variables:

- These are the built-in variables.
- One of the most useful built-in variables is PATH.

ii) User-defined Variables:

- To assign a variable to a value, you do **var=value**. **Note:** There are no spaces before or after the equal sign.
- To get the value of a variable, you do **$var**.
- Variables are not declared, just assign a value.
- Variables have no type they can hold any type of data.
- If you access a variable without a value, you will get an empty string instead of an error.
- Variable names can only start with a letter or an underscore sign. However, after the first letter, variable names can have numbers, too.

hello_world.sh

```
#!/bin/bash

foo=589
bar="Hello World"
baz=$bar

echo $baz
echo $qux
```

```
$ ./hello_world.sh
Hello World

$
```

- Variables defined in a script are lost when the script ends, unless you use the source command to run the script.

hello_world.sh

```
#!/bin/bash

foo=589
bar=Hey!
baz="Hello World"

echo $baz
echo $qux
```

```
$ qux="Shells are awesome"
$ ./hello_world.sh
Hello World

$ echo $baz

$
```

- The subshell does not have access to the variables of the parent shell, unless you export the variable.

```
$ export qux="Shells are awesome"
$ source hello_world.sh
Hello World
Shells are awesome
$ echo $baz
Hello World
```

4) Quotes in Shell:

- Back quotes ``` ` ` ```: Anything in between back quotes would be treated as a command and would be executed.
  I.e. Back quotes does command substitution.
- Single quotes `‘’`: All special characters between these quotes lose their special meaning.
  I.e. Single quotes force shell to take string literally.
- Double quotes `“”`:Expand variables and do command substitution.
  Most special characters between these quotes lose their special meaning with these exceptions:
  - **i)** $
  - **ii)** `
  - **iii)** \$
  - **iv)** \'
  - **v)** \"
  - **vi)** \\
- E.g. Suppose we have 3 directories, a, b and c in our current working directory.
  $ echo * will print a b c
  $ echo ls * will print ls a b c
  $ echo `ls *` will print a b c
  $ echo “ls *” will print ls *
  $ echo ‘ls *’ will print ls *
  $ echo `*` will create an error.
- E.g. Consider the shell script below:
  #!/bin/sh
  $ date
  $ echo Today is `date`
  $ echo "Today is `date`"
  $ echo ’Today is `date`’
  The output is:
  Thu Jan 17 10:58:13 STD 2019
  Today is Thu Jan 17 10:58:13 STD 2019
  Today is Thu Jan 17 10:58:13 STD 2019
  Today is `date`

5) **Read:**
   - Read one line from standard input and assigns successive words to the specified variables.
   - Leftover words are assigned to the last variable.
   - E.g. Suppose we have this shell script:

```
#!/bin/sh
echo "Enter your name:"
read firstName lastName
echo $firstName
echo $lastName
```

   If the user runs the script and enters Alexander Graham Bell for read, then the output is:
   Alexander
   Graham Bell

6) **Commandline Arguments:**
   - Commandline arguments are placed in positional parameters. The commandline arguments are denoted by $1, $2, … . After $9, we use ${10}.
   - Command line arguments allows the users to either control the flow of the command or to specify the input data for the command.
   - Positional parameters in a shell script are the command line arguments passed to a shell script.
   - $0 is the name of the script
   - $# is the number of commandline arguments
   - $* and $@ list all commandline arguments
   - E.g. Consider the shell script below:

```
#!/bin/sh
echo firstname: $1
echo lastname: $2
echo Firstname is $1 and lastname is $2
echo $#
echo $@
echo $*
```

   If I run the script and type, **$ bash test.sh rick lan**, the output is:
   firstname: rick
   lastname: lan
   Firstname is rick and lastname is lan
   2
   rick lan
   rick lan

7) **Positional parameters and set and shift:**
   - The set command assigns its parameters to the positional parameters and all previous positional parameters are thrown away.

- E.g.
  $ set pizza spaghetti rice
  $ echo $1 $2 $3 will prints pizza spaghetti rice
  The set command stored pizza in $1, spaghetti in $2 and rice in $3.
- Shift moves all positional parameters to the left.
  I.e. $1 becomes the old $2, etc.
- The syntax for the shift command is **shift number**, which will shift the positional parameters that many times to the left.
  E.g. shift 1 will shift the positional parameters one to the left.
  E.g. shift 2 will shift the positional parameters two to the left.
  E.g. shift 0 will not do anything.
- The shift command is useful for iterating over all positional parameters.
- E.g. Consider the shell script below:

```
#!/bin/sh
set x y z
shift 1
echo $@
```

  This will print y z
- E.g. Consider the shell script below:

```
#!/bin/sh
set x y z
shift 0
echo $@
```

  This will print x y z

8) if statement and test command:

- The test command takes an expression and returns 0 if its true and 1 if its false.
- E.g.

```
if test $str1 = $str2;
then
        echo "The strings are identical"
else
        echo "The strings are different"
fi
```

- A short form of test is [ ].
- E.g.
  if [$str1 = $str2] ← The spaces are mandatory.
  then …

Test commands for String	**Description**
-z string	True if empty string
str1 = str2	True if str1 equals str2
str1 != str2	True if str1 not equal to str2
Test commands for Integer	**Description**
int1 -eq int2	True if int1 equals int2
int1 -ge int2	True if int1 >= int2
int1 -gt int2	True if int1 > int2
int1 -lt int2	True if int1 < int2
int1 -le int2	True if int1 <= int2
-a	and
-o	or
Test commands for Files and Directories	**Description**
-d filename	Exists as a directory
-f filename	Exists as a regular file
-r filename	Exists as a readable file
-w filename	Exists as a writable file
-x filename	Exists as an executable file

- The syntax for the if statement is:

```
if [ condition1 ]
then
    Statement(s) to be executed if condition1 is true
elif [ condition2 ]
then
   Statement(s) to be executed if condition2 is true
else
   Statement(s) to be executed if no condition is true
fi
```

- In bash the if statement checks the return value of a condition and proceeds to “then” if the return value is 0, to “elif” if the return value is not 0, and to “else” if none of the return values are 0.

- Unix requires that programs return 0 for success and some other number for failure. You can use $? to check for the return value.
- E.g. Consider the shell script below:

```
#!/bin/sh
a=10
b=20
if [ a != b ] ← The spaces are mandatory.
then
        echo "Correct: $?"
else
        echo "False: $?"
fi
```

The output of this is: Correct: 0

- E.g. Consider the shell script below:

```
#!/bin/sh
a=10
b=20
if [ a == b ] ← The spaces are mandatory.
then
        echo "Correct: $?"
else
        echo "False: $?"
fi
```

The output of this is: False: 1

9) expr:

- Since shell scripts work by text replacement, we need a special function for arithmetic.
- expr works only for integer arithmetic.
- E.g. Consider the shell script below:

```
#!/bin/sh
x=1+3
y=`expr 1 + 3` ← The spaces are mandatory.
z=`expr 1 \* 5` ← Need to escape the * sign first.
echo $x
echo $y
echo $z
```

The output is:

1+3
4
5

10)<u>while loop:</u>

- The syntax is:
  while [condition]
  do
  some code
  done

- E.g. Consider the shell script below:

```
#!/bin/sh
x=10

while [ $x -gt 0 ]
do
    echo $x
    x=`expr $x - 1`
done
```

  The output is:
  10
  9
  8
  7
  6
  5
  4
  3
  2
  1
- We can also use the while loop to read from a file, one line at a time.
- E.g.

```
#!/bin/bash
file="my_file.txt"
while read line
do
    echo $line
done < $file
```

- We can also use the while loop to iterate over arguments.
- E.g.

```
#!/bin/sh
while test "$1" != ""
do
    echo $1
    shift
done
```

- Don't use this one unless you know that the argument list will always be short.

11) <u>for loop:</u>

- The general syntax is:
  for var in something
  do
  some code
  done

- E.g.
  a.
```
#!/bin/bash
for i in 1 2 3 4 5
do
        echo $i
done
```
  b.
```
#!/bin/bash
for i in `ls`
do
        mv $i $i.txt
done
```
  This appends all filenames in the current directory with the extension .txt. A file called x would now be called x.txt.
  c.
```
#!/bin/bash
for i in $*
do
        mv $i $i.txt
done
```
  This appends all files whose names are given to the script as command-line arguments with the extension .txt.