

BASH SCRIPTING

Shell Scripting

Shell scripting for beginners and intermediate levels

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

أَكْمَلَ لِلّٰهِ الَّذِي أَعْانَنِي عَلٰى الْإِنْتِهَاءِ مِنْ هَذَا الْكِتَابِ

هذا الكتاب هو ترجمة لكتاب الـ Shell Scripting للمؤلف Hamish Whittal

وهو كتاب يشرح الـ Shell scripting بأسلوب سهل وبسيط للمبتدئين والمستوى المتوسط حتى تصل اطلاعهم بسهولة ويسر

امني ان يحقق هذا الكتاب الاستفادة المرجوة ولا ابعدي به الا وجه الله الكريم

لا تنسونا من صالح دعائكم

TABLE OF CONTENTS

What is Shell Scripting	7
Introduction to info, man and the whatis database.....	9
info pages.....	9
Man Pages	12
The whatis Database.....	13
Revising some Basic Commands	14
The who command.....	14
The w Command	16
The "date" command.....	18
The 'echo' command	19
Running a Shell Script.....	20
File Commands.....	20
Wildcards.....	20
Returning to file commands.....	23
ls	23
cp	23
mv	23
wc	24
nl.....	24
System Commands	24
The df and du commands.....	24
the fdisk command.....	25
The free command.....	26
the vmstat command	27
the iostat command	27
stdin, stdout, stderr.....	27
stdin	28
stdout	28
Using stdin and stdout simultaneously	30
Appending to a file	30
stderr.....	31

stdout, stderr and using the ampersand(&)	32
Unnamed Pipes	34
Introduction	36
What is the login shell?	37
The job of the shell	42
Command Interpreter	42
Allows for variables to be set	43
I/O redirection	43
Pipelines	43
Customising your environment	43
Introduction	44
What are regular expressions?	44
The fullstop (period) ". "	44
Let's explore "sed" syntax	46
Square brackets ([]), the caret (^) and the dollar (\$)	48
Using sed and pipes	50
The splat (asterisk) (*)	52
The plus operator(+)	53
Matching a specified number of the pattern using the curly brackets{}	54
A detour - Using a different field separator in sed pattern matching	56
Using Word Encapsulating Characters	57
Returning from detour to our discussion on curly braces...	57
The tr command	57
The cut command	59
The paste command	60
The uniq command	61
The Sort command	62
The grep command	64
grep, egrep and fgrep	67
Section Techniques to use when writing, saving and executing Shell Scripts	68
Detour: File Extension labels	68
Comments in scripts	69

Variables	69
Shebang or hashpling!#	70
Exit.....	70
Null and unset variables.....	71
Variable Expansion.....	73
Environmental vs shell variables	74
Arithmetic in the shell.....	75
Introduction.....	77
Single Quotes or "ticks"	77
Double Quotes	78
Backticks.....	80
Shell Arithmetic's with expr and back quotes	81
Another tip when using quotation marks.....	82
Introduction.....	84
Positional Parameters 0 and 1 through 9.....	86
Other arguments used with positional parameters.....	87
\$# How many positional arguments have we got ?.....	87
\$* display all positional parameters	87
Using the "shift" command - for more than 9 positional parameters.....	88
Exit status of the previous command	88
Making Decisions.....	90
Testing for a true or false condition	90
The test command	90
What is "true" and "false".....	91
Different types of tests	92
Testing a string	92
Has a variable been set or not ?.....	93
Numeric Tests	95
File test.....	96
Logical Operators	98
Conditions in the shell.....	100
Using the "if" statement.....	100

The "if" "then" "else" statement.....	101
The "elif" statement	102
The "case" statement.....	102
Debugging your scripts.....	103
The NULL command	105
The and && commands	106
Introduction.....	109
The "for" loop.....	109
while and until loops	116
The break and continue commands	120
getopts Using arguments and parameters.....	123
Introduction.....	124
The read command.....	124
Presenting the output.....	127
The echo command.....	127
The printf command.....	128
The shell environmental variables pertaining to scripting.....	132
The Source command.....	134
the exec command	136
Other methods of executing a script or a series of commands	136
Execution with Round brackets	136
Execution with Curly brackets.....	137
Introduction.....	139
PARAM:-value	139
PARAM:=value	141
\${param:+value}	141
\${variable%pattern}	142
MAGIC%%r*a.....	143
variable#pattern	143
variable:OFFSET:LENGTH	145
#variable.....	145
Re-assigning parameters with set	146

Explaining the default field separator field - IFS	147
Setting variables as "readonly"	147
The eval command	149
Running commands in the background using&	152

CHAPTER 1. INTRODUCTION TO THE COURSE STRUCTURE

WHAT IS SHELL SCRIPTING

اذا اردت ان تكون system administrator جيد يجب ان تكون كسول ، هكذا يقول المثل المشهور ، لانك ستكون حينئذ تفك بطريقة ابسط وافضل واسرع

واذا اردت ان تتعلم scripting فلا تفعل اي شيء بطريقة manual مادام هناك script يستطيع عمل ذلك

ولكن لكي نتعلم scripting يجب ان نعلم ما هو ال shell ???

ال shell هو مترجم لل commands شبيه بال DOS ، حيث يقوم باستقبال ال commands التي تكتبها ال user على ال command line ويترجمها

ف اذا قمنا بكتابة هذا الامر في ال terminal

```
# ls -l
```

كيف يقوم ال linux بتنفيذ هذا الامر ، وكيف يعرف معنى "-l" ، وكيف يعرف ان الناتج سيظهر على الشاشة ، هل يعرف بالصدفة ??

لا ، هو لا يعرف بالصدفة ، بل يعرف عن طريق ال shell

وال shell script هو مجموعة من ال commands التي تكتب مع بعضها البعض بداخل ال shell ويتم وضعهم في ملف واحد لكي يقوموا بوظيفة معينة، مثله كمثل ال DOS batch

واذا اردنا تنفيذ هذين الامرین :

```
# free
```

والامر

```
# df -h
```

فالطريقة الاولى هي ان نقوم بتنفيذ كل امر علي حدة

```
[root@localhost ~]# free
total        used         free      shared  buffers   cached
Mem:    1017560     937548      80012          0     21364  442136
 -/+ buffers/cache:  474048      543512
Swap:  1023996       4572  1019424

[root@localhost ~]# df -h
Filesystem      Size  Used Avail Use% Mounted on
rootfs        15G  5.4G  8.4G  40% /
devtmpfs      488M     0  488M   0% /dev
tmpfs         497M   80K  497M   1% /dev/shm
tmpfs         497M   46M  451M  10% /run
/dev/sda3      15G  5.4G  8.4G  40% /
tmpfs         497M   46M  451M  10% /run
tmpfs         497M     0  497M   0% /sys/fs/cgroup
tmpfs         497M     0  497M   0% /media
/dev/sda5      985M   18M  917M   2% /home
/dev/sda2      485M  194M  266M  43% /boot
[root@localhost ~]#
```

او ان نقوم بتنفيذ الامرين مع بعض في نفس السطر وذلك بوضع علامة”; وهي نفس العلامة التي توضع بين الـ E-mail address لرسالة لآخر من شخص

```
[root@localhost ~]# free;df -h
total        used         free      shared  buffers   cached
Mem:    1017560     937796      79764          0     21392  442252
 -/+ buffers/cache:  474152      543408
Swap:  1023996       4572  1019424

Filesystem      Size  Used Avail Use% Mounted on
rootfs        15G  5.4G  8.4G  40% /
devtmpfs      488M     0  488M   0% /dev
tmpfs         497M   80K  497M   1% /dev/shm
tmpfs         497M   46M  451M  10% /run
/dev/sda3      15G  5.4G  8.4G  40% /
tmpfs         497M   46M  451M  10% /run
tmpfs         497M     0  497M   0% /sys/fs/cgroup
tmpfs         497M     0  497M   0% /media
/dev/sda5      985M   18M  917M   2% /home
/dev/sda2      485M  194M  266M  43% /boot
[root@localhost ~]#
```

اذا فهذه الطريقة توفر القليل من الجهد والوقت ، ولكن مازالت هناك طريقة افضل وتتوفر وقت وحده اكثـر ، الا وهي وضع هذين الامرـين في ملف ، وعندما نريد تنفيذهـما نقوم فقط بكتابـة اسم الملف فـتنظـم النـتيـجة عـلـي الشـاشـة ، وهذا المـلـف يـسـمـي script

```
[root@localhost ~]# vim script.sh
[root@localhost ~]# cat script.sh

#!/bin/bash
free; df -h

[root@localhost ~]# chmod +x script.sh
[root@localhost ~]# ./script.sh

total        used        free      shared      buffers      cached
Mem:       1017560     943812     73748          0      22872    446908
-/+ buffers/cache:     474032     543528
Swap:      1023996      4572   1019424

Filesystem      Size  Used Avail Use% Mounted on
rootfs         15G  5.4G  8.4G  40% /
devtmpfs       488M    0  488M   0% /dev
tmpfs          497M  224K  497M   1% /dev/shm
tmpfs          497M   46M  451M  10% /run
/dev/sda3       15G  5.4G  8.4G  40% /
tmpfs          497M   46M  451M  10% /run
tmpfs          497M    0  497M   0% /sys/fs/cgroup
tmpfs          497M    0  497M   0% /media
/dev/sda5       985M   18M  917M   2% /home
/dev/sda2       485M  194M  266M  43% /boot
[root@localhost ~]#
```

INTRODUCTION TO INFO, MAN AND THE WHATIS DATABASE

من المفيد التعرف على ال info pages وال man pages قبل التعامل مع ال scripts ، وال
man pages هي اختصار لل manual pages وتحتوي على معلومات كثيرة عن ال
info pages ولكن قد لا نجد ما نحتاجه في ال man pages لذلك قد نحتاج ال commands

INFO PAGES

ال info pages تشبه ال man pages كثيرا الا انها تحتوي على معلومات اكثرا، وطريقة
استخدام ال info pages بسيطة ، فاذا اردنا ان نستعرض معلومات عن امر معين فنقوم بكتابته
الامر بهذا الشكل :

```
# info ls
```

ويمكن استعراض معلومات عن مجموعة من ال commands التي تعامل مع ال files مثلا
عن طريق

```
# info coreutils
```

وال coreutils هو اختصار commands وهي مجموعة الادوات او ال tools تستخدمن في التعامل مع ال files مثل text files

```
File: coreutils.info, Node: Top, Next: Introduction, Up: (dir)

GNU Coreutils
*****  
  
This manual documents version 8.12 of the GNU core utilities, including  
the standard programs for text and file manipulation.  
  
Copyright (C) 1994-1996, 2000-2011 Free Software Foundation, Inc.  
  
Permission is granted to copy, distribute and/or modify this  
document under the terms of the GNU Free Documentation License,  
Version 1.3 or any later version published by the Free Software  
Foundation; with no Invariant Sections, with no Front-Cover Texts,  
and with no Back-Cover Texts. A copy of the license is included  
in the section entitled "GNU Free Documentation License".  
  
* Menu:  
  
* Introduction::           Caveats, overview, and authors  
* Common options::         Common options  
* Output of entire files:: cat tac nl od base64  
* Formatting file contents:: fmt pr fold  
* Output of parts of files:: head tail split csplit  
* Summarizing files::      wc sum cksum md5sum shalsum sha2  
* Operating on sorted files:: sort shuf uniq comm ptx tsort  
* Operating on fields::    cut paste join  
--zz-Info: (coreutils.info.gz)Top, 334 lines --Top-----  
Welcome to Info version 4.13. Type h for help, m for menu item.
```

وكما نري في هذا الشكل ال info page والتي تحتوي على :

Info و هو اسم ال file الذي يحتوي على هذه ال File: coreutils.info •

```
File: coreutils.info, Node: Top, Next: Introduction, Up: (dir)
```

وكلمة node و topic تعني ان هذا ال topic هو اول topic في info page •
هذا ال Node: Top •

```
File: coreutils.info, Node: Top, Next: Introduction, Up: (dir)
```

- وتعني ان ال topic اسمها introduction وللذهاب الى ال Next: introduction "n" التالية نستخدم الحرف

```
File: coreutils.info, Node: Top, Next: Introduction, Up: (dir)
```

- وتعني ان ال topic اسمها disk usage ويمكن Previous: disk usage "p" الانتقال اليها عن طريق الحرف

```
File: coreutils.info, Node: Printing text, Next: Conditions, Prev: Disk usage, \
Up: Top
```

- وتعني ال info page السابقة ويمكن الانتقال اليها عن طريق الحرف "u" Up : (dir)

```
File: coreutils.info, Node: Top, Next: Introduction, Up: (dir)
```

- ونلاحظ ان هناك عنوان يسمى menu وهي قائمة ال topics الموجودة في هذه ال info page والتي يمكن ان تنتقل بينهم

and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

* Menu:	
* Introduction::	Caveats, overview, and authors
* Common options::	Common options
* Output of entire files::	cat tac nl od base64
* Formatting file contents::	fmt pr fold
* Output of parts of files::	head tail split csplit
* Summarizing files::	wc sum cksum md5sum shalsum sha2

- ويمكن الذهاب مباشرة الى اي topic في الصفحة عن طريق النزول الى اسم ال topic الموجود تحت عنوان menu ثم الضغط على enter ، فاذا اردنا مثلا الذهاب files topic فنقوم بالنزول بالسهم الى فيه ثم نضغط enter
- وعندما نريد العودة الى صفحة ال coreutils الرئيسية dir: Up نضغط "u"
- واذا اردنا الذهاب الى بداية الصفحة node: top نضغط "t"
- وللحث عن كلمة معينة في ال info page نقوم بكتابه "/" يتبعها الكلمة المراد البحث عنها

```
/Directory
```

واذا اردنا ان نبحث عن نفس الكلمة مرة اخري نستخدم

```
/<ENTER>
```

- وللخروج من ال info page نكتب "q"
- للانتقال صفحة للاعلى نستخدم "backspace"
- للحصول على ال help الخاص بال info page نكتب "?"
- وللخروج من ال help نكتب "CTRL+x+0"

MAN PAGES

تحدثنا عن ال Info pages وهي وسيلة اضافية للحصول علي المساعدة ، ولكن الطريقة الاساسية للحصول علي المعلومات في اللينكس هي ال man pages

والحقيقة ان ال man page تعلم من خلال ال less command الذي يشبه ال more و لكنه يحتوي علي اكثر options

ولتشغيل ال man page نستخدم الامر :

```
man <command>
```

وهي تساوي

```
man ls | less
```

- اذا اردنا ان نبحث عن الكلمة في ال man page نقوم بكتابه "/" يتبعها الكلمة المراد البحث عنها كما يحدث في ال info page ولكنها هنا تقوم بعمل forward search اي ان البحث عن الكلمة يكون من اول ال page الي اخرها
- اذا اردنا عمل reverse search فنقوم باستخدام العلامة "?" بدلا من "/" ثم يتبعها الكلمة المراد البحث عنها من اسفل الملف الي اعلاه
- وللانتقال من صفحة لتي تليها نستخدم ال space bar
- وللانتقال من صفحة لتي تسبقها نستخدم الحرف "b"
- وللخروج من ال man page نكتب "q"
- وللحصول علي مساعدة في كيفية استخدام ال man pages والتنقل بداخليها نضغط "h"
- ، ونلاحظ من الصورة التالية ان المعرض هو ال help الخاص بالامر less وليس لل man page لأن ال man page هي مجرد صفحة عادية يتم عرضها بواسطة ال less command

SUMMARY OF LESS COMMANDS

Commands marked with * may be preceded by a number, N.
Notes in parentheses indicate the behavior if N is given.

```
h H          Display this help.  
q :q Q :Q ZZ Exit.
```

MOVING

```
e ^E j ^N CR * Forward one line (or N lines).  
y ^Y k ^K ^P * Backward one line (or N lines).  
f ^F ^V SPACE * Forward one window (or N lines).  
b ^B ESC-v * Backward one window (or N lines).  
z * Forward one window (and set window to N).  
w * Backward one window (and set window to N).  
ESC-SPACE * Forward one window, but don't stop at end-of-file.  
d ^D * Forward one half-window (and set half-window to N).  
u ^U * Backward one half-window (and set half-window to N).  
ESC-) RightArrow * Left one half screen width (or N positions).  
ESC-( LeftArrow * Right one half screen width (or N positions).
```

```
HELP -- Press RETURN for more, or q when done
```

: sections الى مجموعة من ال Man page وتنقسم الى

-1 command : هو اسم ال Name section

-2 SYNOPSIS section : او الملخص او الاختصار وهي تعني الصورة المختصرة والبسيطة لاستخدام الامر

-3 DESCRIPTION section : وهو الجزء الذي يشرح وظيفة الامر

-4 SEE ALSO section : وهذا الجزء يوضح بعض الطرق للحصول على مزيد من التفاصيل

THE WHATIS DATABASE

Whatis هو command يسمح لنا بمعرفة وظيفة ال command بشكل سريع ومختصر حيث يقوم بالبحث في كل ال manual pages في ال system عن الجزء الخاص بال name database الموجودة في هذه ال man page ثم تقوم بترتيبهم وتضعهم في section خاصة at night rebuild لها update ويتم عمل update لها

```
whatis nl
```

```
[root@localhost ~]# whatis nl  
nl (1p)           - line numbering filter  
nl (1)            - number lines of files  
[root@localhost ~]#
```

وإذا أردنا أن نقوم بعمل update على database يدويا نستخدم الامر :

```
makewhatis -u -w
```

REVISING SOME BASIC COMMANDS

سوف نتحدث في هذا الجزء عن بعض الـ basic commands والغرض هنا هو معرفة مما يتكون الـ commands وكيف نستخدمه

THE WHO COMMAND

يستخدم هذا الـ command في معرفة الـ user على الـ system ، وإذا استخدمناه بدون أي arguments أو options فسيظهر لنا التالي :

```
[root@localhost ~]# who  
root    tty1          2012-10-16 05:34 (:0)  
root    pts/0          2012-10-16 05:35 (:0.0)  
[root@localhost ~]#
```

Annotations:

- User ID: Points to "root" in the first line.
- Used consol: Points to "tty1" and "pts/0" in the first line.
- Time of logging: Points to the timestamp in the first line.
- X Consol: Points to the colon followed by "0" in the timestamp of the second line.

ويمكن ان نستخدم الـ help لعرض الـ options التي يمكن ان تستخدم مع هذا الـ command

```
who --help
```

```
[ahmed@localhost ~]$ who --help
Usage: who [OPTION]... [ FILE | ARG1 ARG2 ]
Print information about users who are currently logged in.

-a, --all      same as -b -d --login -p -r -t -T -u
-b, --boot     time of last system boot
-d, --dead     print dead processes
-H, --heading  print line of column headings
-l, --login    print system login processes
--lookup      attempt to canonicalize hostnames via DNS
-m            only hostname and user associated with stdin
-p, --process  print active processes spawned by init
-q, --count    all login names and number of users logged on
-r, --runlevel print current runlevel
-s, --short    print only name, line, and time (default)
-t, --time     print last system clock change
-T, -w, --mesg add user's message status as +, - or ?
-u, --users   list users logged in
--message     same as -T
--writable    same as -T
--help        display this help and exit
--version     output version information and exit
```

If FILE is not specified, use /var/run/utmp. /var/log/wtmp as FILE is common.
If ARG1 ARG2 given, -m presumed: `am i' or `mom likes' are usual.

ولعرض الـ **logged users** مع اظهار عنوان كل **column**

```
who -H
```

NAME	LINE	TIME	COMMENT
root	tty1	2012-10-16 05:34 (:0)	
root	pts/0	2012-10-16 05:35 (:0.0)	

column heading

ولعرض **logged users** عن الـ **short list**

```
who -s
```

```
[ahmed@localhost ~]$ who -s
root      tty1          2012-10-16 05:34 (:0)
root      pts/0          2012-10-16 05:35 (:0.0)
[ahmed@localhost ~]$
```

ولعرض الـ **Process ID's** والـ **logged users**

```
who -u
```

```
[ahmed@localhost ~]$ who -u
root    tty1          2012-10-16 05:34  old   1357 (:0)
root    pts/0          2012-10-16 05:35  old   1910 (:0.0)
[ahmed@localhost ~]$
```

وإذا استخدمنا "H" مع "u" فسوف يظهر اسم ال columns او ال heading ، وسوف نجد ان ال الرابع يسمى IDLE ، وهو يمثل ال idle time لكل user ويعني ان ال user لم يستخدم ال system لفترة زمنية معينة

وإذا لم يقوم ال user باستخدام ال system لمدة 15 min. يحدث له logout

```
who -uH
```

```
[ahmed@localhost ~]$ who -uH
NAME   LINE        TIME      IDLE      PID COMMENT
root   tty1        2012-10-16 05:34  old     1357 (:0)
root   pts/0        2012-10-16 05:35  old     1910 (:0.0)
root   pts/1        2012-10-16 06:03  old     1910 (:0.0)
[ahmed@localhost ~]$
```

THE W COMMAND

ما هو ال w command

إذا قمنا بالاستعلام عن ال w command باستخدام الامر whatis فسنجد الاتي :

```
[root@localhost ~]# whatis w
w (1)           - Show who is logged on and what they are doing.
[root@localhost ~]#
```

وهذا يعني ان هذا الامر يقوم بعرض الاشخاص الذين دخلوا علي ال system وماذا فعلوا

```
[root@localhost ~]# w
09:42:54 up 11 min, 2 users, load average: 0.00, 0.17, 0.22
USER   TTY      FROM          LOGIN@    IDLE     JCPU    PCPU WHAT
root    ttys1    :0           09:34      ?      3.93s  3.93s /usr/bin/Xorg :0
root    pts/0    :0.0         09:34      0.00s  0.21s  0.00s w
[root@localhost ~]#
```

كما نرى في الصورة ان السطر الاول والثاني يسميان heading او header

User	اسم ال user
tty	اسم ال tty الي عليها ال user
login@	توقيت دخول ال User
idle	الوقت الذي مر منذ اخر استخدام لل terminal
JCPU	الزمن الذي قضته جميع ال child processes وال terminal في استخدام ال processes
PCPU	الزمن الذي قضته ال active process في استخدام ال terminal
What	اسم ال command او ال process الذي يعمل

وهذه قائمة بال options المستخدمة مع هذا ال command

-h	تنع ظهور ال header
-u	Ignores the username while figuring out the current process and cpu times. To demonstrate this, do a "su" and do a "w" and a "w -u".
-s	يقوم بعرض short format ولا يقوم بعرض ال login time وال JCPU وال PCPU
-f	يمنع ظهور from من النتيجة
-V	يعرض معلومات عن ال version
user	يعرض معلومات عن user معين

ويتم تخزين ال user logged في ملف **/etc/utmp**

ويتم تخزين معلومات عن محاولات login الفاشلة في ملف **/etc/security/failedlogin**

THE "DATE" COMMAND

السبب في اننا نسترجع هذه ال commands البسيطة هي اننا سنقوم بعمل اول script في هذا ال course بهذه ال commands

وسنحتاج الي ال date command في ال scripting لأننا دائماً نحتاج ان نعرف متى بدأ ال script في العمل ومتى انتهي ، كما يقوم ال date command بعمل بعض الاشياء الجميلة التي تحتاجها مثل التحويل من ال unix time الى human time ، كما يمكن التحويل بين صيغ عرض التاريخ المختلفة

```
# info date
```

```
File: coreutils.info, Node: date invocation, Next: arch invocation, Up: System context

21.1 `date': Print or set system date and time
=====

Synopsis:

  date [OPTION]... [+FORMAT]
  date [-u|--utc|--universal] [ MMDDhhmm[[CC]YY][.ss] ]

Invoking `date' with no FORMAT argument is equivalent to invoking it
```

ولو نظرنا الي ال form الاولى فسنجد الشكل التالي :

```
date [+      ]
date + "      "
```

وهو يساوي

ويتم وضع ال form المطلوبة بين ال " "

```
[root@localhost ~]# date
Sun Oct 21 21:29:42 EET 2012
[root@localhost ~]# date +"%d-%b-%y"
21-Oct-12
[root@localhost ~]# date +"%d-%b-%Y"
21-Oct-2012
[root@localhost ~]# date +"%d-%B-%Y"
21-October-2012
[root@localhost ~]# date +"%D-%B-%Y"
10/21/12-October-2012
[root@localhost ~]# date +"%D"
10/21/12
[root@localhost ~]# date +"%H:%M"
21:30
[root@localhost ~]# date +"%l :%M"
9:30
[root@localhost ~]# date +"%l :%M %p"
9:30 PM
[root@localhost ~]#
```

use CAPITAL letters for complete typing

use " l " instead of " H " to change from 24h to 12h

THE 'ECHO' COMMAND

آخر امر سنتحدث عنه الان هو الامر echo وهو امر غاية في الاهمية ولا غني عنه في اي script ان ال commands التي تحدثنا عنها سابقا يمكن تنفيذها بسهولة على ال terminal ورؤيه نتيجتها ايضا، ولكن اذا وضعناها في script فلن تظهر النتيجة على الشاشة ، **اذا فما هي طريقة اظهارها على الشاشة؟؟**

الطريقة هي استخدام الامر echo الذي يقوم بطباعة الكلمات على الشاشة و echo له نوعين ، اولهما يكون built in the shell والآخر يكون external وسنعرف لاحقا متى نستخدم كل نوع ولكن الي ان نقوم بذلك فسوف نستخدمه في عرض التاريخ كالتالي :

```
[root@localhost ~]# cat date_script.sh
#!/sbin/bash
echo "Today is :"
date +"%D"
```

وستظهر النتيجة كالتالي

```
[root@localhost ~]# sh date_script.sh
Today is :
10/21/12
```

RUNNING A SHELL SCRIPT

لتشغيل ال script يجب ان يكون له execution permissions وذلك عن طريق استخدام الامر :

```
Chmod u+x script.sh
Chmod g+x script.sh
Chmod o+x script.sh
Chmod x script.sh
```

وهذا بالطبع يختلف عن الويندوز الذي يعتبر ان اي ملف له امتداد .exe او .com هو executable file بينما اللينكس ليس به extensions واستخدامنا ل ".sh" ليس ك extension وانما هو تمييز فقط علي ان هذا ال file هو script اي انها عملية تنظيمية فقط ، ولو ازناها فسيتم تنفيذ ال script ايضا

وهناك عدة طرق لتشغيل ال script مثل :

```
./script.sh
```

OR

```
sh script.sh
```

OR

يمكن وضعه في ال bin / وتنفيذ بدون استخدام "./".

FILE COMMANDS

سنتكلم في هذا الجزء عن ال commands التي تعامل مع ال files مثل "ls" الذي يستخدم لعمل Listing لل files

WILDCARDS

ال wildcard هي مجموعة من ال characters التي تسمح لنا بالحصول على ال files التي نبحث عنها تحديداً حسب المواصفات التي نريدها مثل الملفات التي تبدأ بحرف معين أو تنتهي بحرف معين

wildcard characters وهذه بعض ال

Symbol	Name	Purpose
*	Splat	represents any character or group of characters
?	question mark	represents exactly one character
[]	square brackets	represents a group of characters enclosed in brackets
!	bang	means negation

WILDCARD CHARACTERS WITH THE ASTERISK(*)

يقوم بعمل character ل matching واحد او اكثـر مثل :

```
[root@localhost ~]# ls fil*
file file1 file2 film
[root@localhost ~]# ls file*
file file1 file2
[root@localhost ~]#
```

THE QUESTION MARK (?) WILDCARD CHARACTER

يقوم بعمل character ل matching اي واحد فقط

```
[root@localhost ~]# echo fil?
file film
[root@localhost ~]# echo file?
file1 file2
[root@localhost ~]#
```

THE SQUARE BRACKETS([])

```
[root@localhost ~]# echo file[1-3]
file1 file2 file3
[root@localhost ~]# echo file[123]
file1 file2 file3
```

تقوم بعمل matching للمدى الموجود بين الاقواس

لاحظ الفرق بين :

```
[root@localhost ~]# echo file[1-4][1-4]
file14 file24
```

وبين

```
[root@localhost ~]# echo file[1-4]
file1 file2 file3 file4
```

THE BANG(!)

اي ان هذه العلامة تقوم بعرض جميع النتائج ما عدا التي تحتوي على هذه القيم

```
[root@localhost ~]# ls file[!1-3]
file4
```

امثلة :

```
touch {planes,trains}_{10,11}.{bak,bat}
```

يقوم هذا ال command touch بعمل files في سطر واحد، وينتج عن هذا ال files هذه ال command

Planes_10.bak, planes_10.bat, planes_11.bak, planes_11.bat, trains_10.bak, trains_10.bat, trains_11.bak, trains_11.bat

RETURNING TO FILE COMMANDS

سنتحدث عن اهم ال commands المستخدمة في ال bash scripting والتي تعامل مع ال files

LS

وهذا ال command يستخدم في عمل List لكل ال files الموجودة في ال directory ، ويحتوي على بعض ال options

- : يعني long list وهذا ال option يقوم باظهار بيانات اكثر تفصيلا عن ال files مثل ال group و ال size و ال last modification time و ال permissions

```
[root@localhost ~]# ls -l
total 176
-rw-----. 1 root root 2054 Aug 7 03:21 anaconda-ks.cfg
-rwxr-xr-x. 1 root root 121 Oct 21 22:16 date_script.sh
drwxr-xr-x. 3 root root 4096 Oct 1 10:58 Desktop
```

-a : وهو يقوم باظهار ال hidden files والتي تحتوي على (. و ..)

```
[root@localhost ~]# ls -a
.          Documents      .gvfs        Public
..         Downloads       .ICEauthority .pulse
.abrt      .esd_auth     .icedtea    .pulse-cookie
```

وال files التي تحتوي على (.) هي ال files الموجودة في ال directory الحالي بينما ال files التي تحتوي على (..) هي ال files الموجودة في ال parent directory

CP

يقوم هذا ال copy command بعمل copy لـ files

MV

يقوم هذا ال command move بعمل files على

WC

يقوم بعمل count لعدد ال Lines وال words و characters

NL

يوضح عدد ال lines في ال file

SYSTEM COMMANDS

هي ال commands التي تتعامل مع ال system مثل df, du, fdisk, iostat, vmstat, free

df	يقوم بعرض ال free disk space
du	يقوم بعرض ال disk usage
fdisk	يقوم بعرض ال partitions الموجودة في ال hard disk
iostat	يقوم بعرض حالة ال input/output على ال hard disk
vmstat	يقوم بعرض حالة ال virtual memory
free	يقوم بعرض ال free memory

THE DF AND DU COMMANDS

df : يقوم بعرض ال partitions الموجودة على الجهاز ومعلومات عنهم مثل المساحة الفارغة والممساحة الممتلئة وال mounted partitions وغيرها

اذا فيمكن استخدام هذا ال command في عمل script يقوم بعمل check على ال partitions واظهار alarm عند امتلائه او وصوله الى threshold معين

وهذا ال command يقوم بعرض ال free disk space بال byte ، ولكن لعرضها بشكل يسهل قراءته فيمكن استخدام بعض ال options مثل :

```
df -hT
```

حيث ان ال "h" يعني human readable

وال "T" يعني type وهي توضح نوع ال partition هل هو ext2, ext3, extended,

```
[root@localhost ~]# df -hT
Filesystem      Type   Size  Used Avail Use% Mounted on
rootfs          rootfs  15G  5.6G  8.2G  41% /
devtmpfs        devtmpfs 234M     0  234M   0% /dev
tmpfs           tmpfs   245M   80K  245M   1% /dev/shm
tmpfs           tmpfs   245M  664K  244M   1% /run
/dev/sda3        ext4    15G  5.6G  8.2G  41% /
tmpfs           tmpfs   245M     0  245M   0% /sys/fs/cgroup
tmpfs           tmpfs   245M     0  245M   0% /media
/dev/sda5        ext4   985M   18M  917M   2% /home
/dev/sda2        ext4   485M  218M  242M  48% /boot
[root@localhost ~]#
```

ويمكن استخدام الامر du الذي يوضح اي الملفات لها حجم كبير وتسهلك مساحة في ال hard disk

```
184092 ./Desktop
4      ./Pictures
4      ./Documents
4      ./Music
148    ./pulse
```

ويمكن اضافة "-s" لعرض summary عن المساحة الكلية المستخدمة من ال file system كله

```
du -s
```

```
[root@localhost ~]# du -s
313888 .
[root@localhost ~]#
```

THE FDISK COMMAND

ال command fdisk يقوم في الأساس على اظهار ال partitions الموجودة في ال HDD

```
[root@localhost ~]# fdisk -l
WARNING: GPT (GUID Partition Table) detected on '/dev/sda'! The util fdisk doesn't support GPT. Use GNU Parted.

Disk /dev/sda: 21.5 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders, total 41943040 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

Device Boot Start End Blocks Id System
/dev/sda1 1 41943039 20971519+ ee GPT
[root@localhost ~]#
```

وهذا ال command لا يستطيع تنفيذه الا ال root

THE FREE COMMAND

هذا ال command يقوم باظهار ال used و ال free و ال total memory و مساحة ال swap و ال used و ال free

```
[root@localhost ~]# free
              total        used        free      shared  buffers   cached
Mem:       500064      403752      96312          0     18352    94036
-/+ buffers/cache:  291364      208700
Swap:    1023996      20492    1003504
[root@localhost ~]# free -k
              total        used        free      shared  buffers   cached
Mem:       500064      403752      96312          0     18352    94036
-/+ buffers/cache:  291364      208700
Swap:    1023996      20492    1003504
[root@localhost ~]# free -b
              total        used        free      shared  buffers   cached
Mem:  512065536  413188096  98877440          0  18800640  96292864
-/+ buffers/cache: 298094592  213970944
Swap: 1048571904  20983808 1027588096
[root@localhost ~]# free -m
              total        used        free      shared  buffers   cached
Mem:        488         394          94          0         17          91
-/+ buffers/cache:  284         204         979
```

THE VMSTAT COMMAND

يقوم هذا ال command باظهار how busy the system is وعدد ال blocked process

كما تقوم ايضاً باظهار معلومات عن ال swap daemon مثل مقدار استهلاك ال memory من ال cache وما قيمة ال buffer وال memory

وكم عدد ال process التي دخلت الى swap والتي خرجت منه وبيانات عن ال users وال idle and waiting time وال system

```
[root@localhost ~]# vmstat
procs -----memory----- swap-----io-----system-----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
0 0 20492 96560 18392 94120 1 14 329 31 107 168 2 3 90 4 0
```

THE IOSTAT COMMAND

يقوم هذا ال command بعرض معلومات عن ال input/output ومعلومات عن system في ال system performance

```
[root@localhost ~]# iostat
Linux 3.6.2-1.fc16.x86_64 (localhost.localdomain)           11/07/2012      _x86_64_      (1 CPU)

avg-cpu: %user   %nice %system %iowait  %steal   %idle
          1.58    0.26   2.24    3.52    0.00   92.41

Device:         tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda            14.29     256.60      24.17    486458      45813
```

ولمعرفة المزيد من المعلومات عن ال system performance يمكن عمل install لـ sysstat package

STDIN, STDOUT, STDERR

يقوم/linux على تشغيل ال commands المكتوبة في ال CLI واخراج النتيجة على الشاشة حتى ان كان يحتوي على errors وهذا يعني ان/linux يعتمد على ال inputs وال outputs وال errors ، ويرمز لكل منهم برمز معين

Type

Symbol

standard input (stdin)	0<
standard output (stdout)	1>
standard error (stderr)	2>

وهذه الارقام (0, 1, 2) تسمى file descriptor لكل نوع من الثلاثة

فال (0) يمثل standard input file descriptor لل

فال (1) يمثل standard output file descriptor لل

فال (2) يمثل standard error file descriptor لل

STDIN

هناك العديد من الطرق التي يمكن ادخال ال data للينكس منها :

-1 - commands : وهي ال CLI التي نكتب فيها ال terminal

-2 - device : وهي ال keyboard او ال device التي تقوم بادخال البيانات منها

-3 - file : وهو file يحتوي على data يستخدمها اللينكس في عملية اخر يقوم بها مثل

```
cat < myfirstscript
cat 0< myfirstscript
cat myfirstscript
```

وهذا ال command يطلب من ال cat ان يأخذ ال input من file اسمه myfirstscript بدلا من ال keyboard ويعرضه على الشاشة

```
[root@localhost ~]# cat < myfirstscript
hi
this is my first script
[root@localhost ~]# cat 0< myfirstscript
hi
this is my first script
[root@localhost ~]# cat myfirstscript
hi
this is my first script
```

STDOUT

يظهر ال output على الشاشة ، فاذا قمنا بكتابة الامر

```
cat myfirstscript
```

فسنجد ان ال output سيظهر علي الشاشة

```
[root@localhost ~]# cat myfirstscript  
hi  
this is my first script
```

ولكن يمكن تغيير ذلك ، فيمكن ان نقوم بعمل redirect لـ output الي مكان اخر كما يلي

```
cat myfirstscript > newscript  
cat myfirstscript 1> newscript
```

```
[root@localhost ~]# cat myfirstscript  
hi  
this is my first script  
[root@localhost ~]# cat myfirstscript > newscript  
[root@localhost ~]# cat newscript  
hi  
this is my first script
```

فعندها عرضنا محتويات myfirstscript بدون ال redirection على الشاشة، ولكن عندما استخدمنا ال redirection وجعلنا ال stdout يذهب الي file يسمى new script على الشاشة ولكن ذهب الي ال file فلم يظهر ال output على الشاشة

اذا كان ال newscript غير موجود اساسا فسوف يقوم بعمل creation له ، اما اذا كان ال newscript موجود مسبقا فسوف يمسح محتوياته ويكتب ال data الجديدة

اذا فان ال newscript هو ال stdout وال myfirstscript هو ال newscript

ولكن ماذا اذا لم يكن هناك newscript او stdin لكي انقل محتوياته الي

الاجابة هنا هي ان ال screen هي التي ستتصبح stdin كما في المثال التالي :

```
[root@localhost ~]# cat > newscript  
hello  
my name is ahmed  
[root@localhost ~]# cat newscript  
hello  
my name is ahmed
```

عندما استخدمنا امر cat مع ال redirection فان ال cat لم تجد محتوياته وتضعها في newscript لذلك تم الاستعاضة عن ال input file بال command line لكي يصبح هو ال input method البديلة ، فانتقل ال curser للسطر التالي وانتظر حتى ندخل ال inputs وبعد الانتهاء من ادخال ال inputs نقوم بالضغط على (^D) حتى نخبر ال cat ان عملية الادخال قد انتهت

USING STDIN AND STDOUT SIMULTANEOUSLY

يمكن استخدام ال stdin وال stdout مع بعضهما البعض كديل لعملية ال copy بين الملفات

```
cat myfirstscript newscript  
cat < myfirstscript > newscript
```

```
[root@localhost ~]# cat newscript  
[root@localhost ~]# cat myfirstscript  
hi  
this is my first script  
[root@localhost ~]# cat < myfirstscript > newscript  
[root@localhost ~]# cat newscript  
hi  
this is my first script
```

وهنا قمنا بنسخ محتويات myfirstscript الى newscript باستخدام ال redirection

APPENDING TO A FILE

ويمكن استخدام ال redirection في عملية appendin g وتعني اضافة البيانات الى ال file بدون مسح البيانات القديمة

```
[root@localhost ~]# cat newscript
hi
this is my first script
[root@localhost ~]# cat < myfirstscript
hi
this is my first script
[root@localhost ~]# cat myfirstscript >> newscript
[root@localhost ~]# cat newscript
hi
this is my first script
hi
this is my first script
[root@localhost ~]# cat < myfirstscript >> newscript
[root@localhost ~]# cat newscript
hi
this is my first script
hi
this is my first script
hi
this is my first script
```

ونجد هنا اننا عندما استخدمنا العلامة (<>) فان ال cat لم يقوم بالكتابة علي newscript وانما قام باضافة البيانات الجديدة اليه

ال redirection يجب ان يكون اخر command يتم تنفيذه في ال script



STDERR

عند تنفيذ اي command ينتج عنه نتيجة من اثنين:

- 1- نتائج صحيحة (valid output)
- 2- نتائج خطأ (error message)

وما يحدث مع ال output العادي يحدث مع ال error حيث يظهر كلاهما على الشاشة

فإذا قمنا بتنفيذ هذا ال command ك normal user وليس ك root user

```
find / -name "*" -print
```

فستظهر النتيجة على الشاشة كما بالصورة حيث تحتوي على valid output و error message

```
[ahmed@localhost ~]$ find /root -name "*" -print  
/root  
find: `/root': Permission denied
```

ولكن اذا اردنا اظهار ال output valid فقط فيمكن ان نقوم بارسال ال error الى file اخر حتى لا يظهر على الشاشة ، وهذا ال file غالبا ما يكون null /dev/null

وال /dev/null تشبه ال recycle bin الموجود في ال windows ولكن ال linux يمكن ارجاع محتوياتها مرة اخرى (كما يحدث في ال shift + delete) لذلك فهي تسمى black hole

ولارسال ال error الى ال /dev/null يجب استخدام ال file descriptor الخاص بال stderr حتى يستطيع اللينكس التمييز بين ال stdout وال stderr

```
find / -name "*" -print 2> /dev/null
```

```
[ahmed@localhost ~]$ find /root -name "*" -print 2> /dev/null  
/root
```

وهنا نجد ان ال output valid هو الوحيد الذي ظهر على الشاشة وقد تكون هذه ال errors مفيدة في حالة عمل troubleshooting او diagnostics ارسال هذه ال errors الى file خاص يمكن من خلاله الاطلاع على هذه ال errors بدلا من ارساله الى ال null

```
[ahmed@localhost ~]$ find /root -name "*" -print 2> myerrors  
/root  
[ahmed@localhost ~]$ cat myerrors  
find: `/root': Permission denied  
[ahmed@localhost ~]$
```

STDOUT, STDERR AND USING THE AMPERSAND(&)

يمكن الدمج بين ال stdout وال stderr معن وارسال command الى نفس file في نفس time فمثلا اذا اردنا تنفيذ command معين وارسال الى file يسمى validout وارسال الى error الى file يسمى errorout فيمكن تنفيذ ذلك في سطر واحد فقط

```
find / -name "*" -print 2> errorout > validout
```

```
[ahmed@localhost ~]$ find /root -name "*" -print 2> errorout > validout
[ahmed@localhost ~]$ cat errorout
find: `/root': Permission denied
[ahmed@localhost ~]$ cat validout
/root
[ahmed@localhost ~]$ █
```

وترتيب ال redirection غير هام

```
find / -name "*" -print 2> errorout > validout
```

```
find / -name "*" -print > validout 2> errorout
```

مرة اخري اذا قمنا بتنفيذ الامر بال user normal

```
find / -name "*" -print
```

فستظهر النتيجة على الشاشة كما بالصورة حيث تحتوي على valid output و error message

```
[ahmed@localhost ~]$ find /root -name "*" -print
/root
find: `/root': Permission denied
```

واذا قمنا باستخدام علامة (>) فسيتم عمل redirect لل valid output فقط

واذا قمنا باستخدام علامة (>2) فسيتم عمل redirect لل error فقط

ولكي نقوم بعمل redirect لل valid output وال error معا نقوم باستخدام العلامة (&) كما يلي

```
find / -name "*" -print 2> samefile 1>&2
```

حيث يقوم هذا ال command بارسال ال error الى ملف اسمه same file ثم يرسل ال valid output الى نفس ال file الذي تم ارسال ال error له ، ويمكن كتابته بهذا الشكل

```
find / -name "*" -print 1> file 2>&1
```

حيث يقوم هذا ال command بارسال ال output الي ملف اسمه file ثم يرسل ال output الي نفس ال file الذي تم ارسال ال error له

```
[ahmed@localhost ~]$ find /root/ -name "*" 2> samefile 1>&2
[ahmed@localhost ~]$ cat samefile
/root/
find: `/root/': Permission denied
```

```
&>file
```

يقوم بعمل redirect لل error وال valid output معا الي ال file

UNAMED PIPES

حتى الان استطعنا ان نرسل ال output الي file ، ولكن هل يمكن ان نرسل هذا ال output ك input الى command اخر وهكذا الي ان نحصل على النتيجة المطلوبة ؟؟؟

الاجابة هي نعم !!! وذلك عن طريق ال (|) pipe مثل

```
ls -la /etc | less
```

```
total 2720
drwxr-xr-x. 149 root      root      12288 Nov 10 15:33 .
dr-xr-xr-x.  22 root      root      4096 Aug  7 02:58 ..
drwxr-xr-x.   3 root      root      4096 Aug  7 03:04 abrt
drwxr-xr-x.   4 root      root      4096 Aug  7 03:16 acpi
-rw-r--r--.   1 root      root       44 Aug  7 03:32 adjtime
drwxr-xr-x.   4 root      root      4096 Aug  7 03:12 alchemist
```

وهنا يقوم ال "ls -la" بعرض long list للملفات الموجودة بداخل /etc/ ثم يقوم بأخذ هذا ال output ويدخله ك input الى الامر less الذي يقوم بعرض النتيجة الاولى صفحة ويتمكن استخدام ال pipe عدة مرات مثل :

```
ls -la | grep a | less
```

```
total 336
drwxr-xr-x. 2 root root 4096 Aug  7 04:20 .abrt
-rw-----. 1 root root 2054 Aug  7 03:21 anaconda-ks.cfg
-rw-----. 1 root root 1347 Nov  7 06:44 .bash_history
-rw-r--r--. 1 root root    18 Feb  9 2011 .bash_logout
-rw-r--r--. 1 root root   176 Feb  9 2011 .bash_profile
-rw-r--r--. 1 root root   229 Oct  1 12:27 .bashrc
-rw-r--r--. 1 root root 12288 Oct  1 13:37 .bashrc.swp
```

ويمكن استخدام ال pipe مع ال redirection

```
[root@localhost ~]# ls -al | grep b > pipescript
[root@localhost ~]# cat pipescript
drwxr-xr-x. 2 root root 4096 Aug  7 04:20 .abrt
-rw-----. 1 root root 1347 Nov  7 06:44 .bash_history
-rw-r--r--. 1 root root    18 Feb  9 2011 .bash_logout
-rw-r--r--. 1 root root   176 Feb  9 2011 .bash_profile
-rw-r--r--. 1 root root   229 Oct  1 12:27 .bashrc
-rw-r--r--. 1 root root 12288 Oct  1 13:37 .bashrc.swp
```

CHAPTER 2. THE SHELL

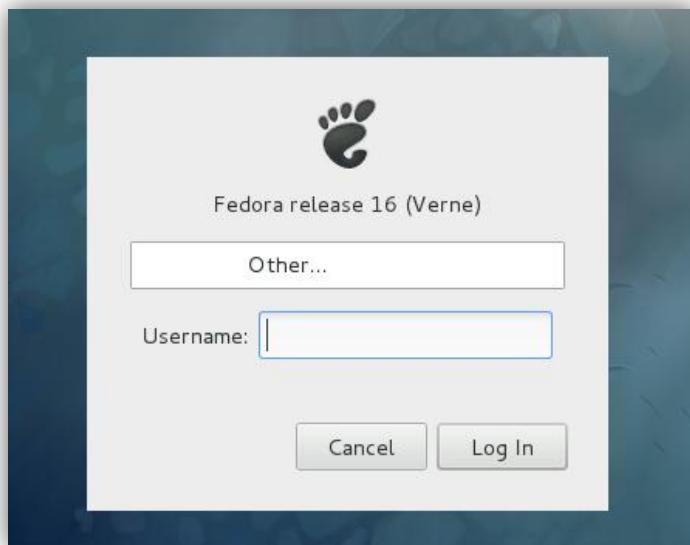
INTRODUCTION

ما هو ال shell ؟

ال shell هو مترجم ال commands ويوجد منه نوعان :

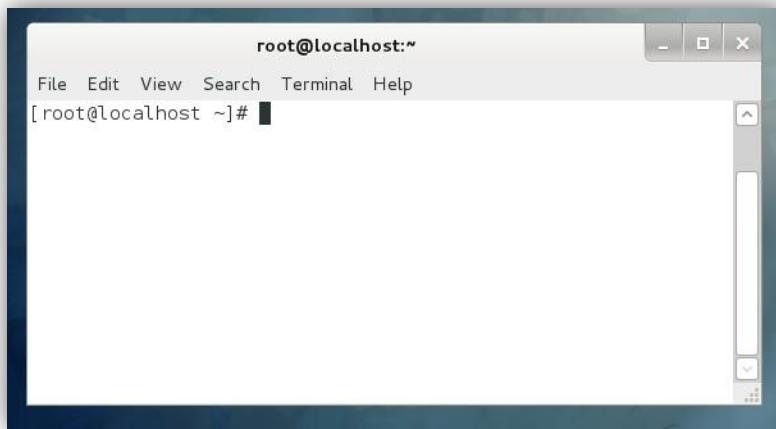
Login shell -1

وهو المسئول عن دخولك الى system والتي يظهر فيها ال username وال password



Non-login shell -2

هي ال terminal التي نكتب فيها ال commands مثل ال bash, sh, ksh, csh و وهناك ما يقرب من 50 non-login shell يمكن استخدامها



ولكي نفهم ال shell يجب ان نعرف كيف تعمل ، ولكن يجب ان نعرف انتا سنسخدم ال bash
لانه هو الاكثر استخداما في معظم توزيعات اللينكس

ولكي نعرف ما هي ال shell المستخدمة

```
echo $0
```

```
[root@localhost ~]# echo $0  
bash
```

WHAT IS THE LOGIN SHELL?

ال login shell هي المسئولة عن :

- تشغيل ال non-login shell
- يقوم بعمل check علي ال environment variables التي يحتاجها ال system مثل PATH, TERM, UID, GID
- تقوم بعمل ال default variables مثل HOME
- يقوم بعمل USERNAME, HISTSIZE, HOSTNAME,

```
[ahmed@localhost ~]$ echo $PATH  
/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin:/root/bin  
[ahmed@localhost ~]$ echo $TERM  
xterm  
[ahmed@localhost ~]$ echo $UID  
1000  
[ahmed@localhost ~]$ echo $HOSTNAME  
localhost.localdomain  
[ahmed@localhost ~]$ echo $USERNAME  
root  
[ahmed@localhost ~]$ echo $HISTSIZE  
1000  
[ahmed@localhost ~]$ echo $HOME  
/home/ahmed
```

وعند بداية تشغيل ال system يقوم ال login shell بعمل check علي نوعين من ال files :

User profile files -1
Shell rc files -2

وبالنسبة لـ system فهذه ال files تسمى bashrc و profile و توجد في /etc وهي تحتوي علي ال functions وال aliases كل و المسئول عنها هو ال root

```
[ahmed@localhost ~]$ ls /etc/profile  
/etc/profile  
[ahmed@localhost ~]$ ls /etc/bashrc  
/etc/bashrc
```

ويحتوي ال /etc/profile علي ال functions وال aliases التي يستعين ال system للدخول علي ال login shell

```
[root@localhost ~]# cat /etc/profile
# /etc/profile

# System wide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.

pathmunge () {
    case ":${PATH}:" in
        *:"$1":*)
            ;;
        *)
            if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
    esac
}

non-login functions على الـ /etc/bashrc والـ aliases التي يحتاجها الـ non-login
shell عند بداية الدخول الى الـ terminal للتعرف على الاعدادات التي يستخدمها الـ user في
التعامل مع الـ shell
```

بينما يحتوي الـ /etc/profile على الـ functions والـ aliases التي يحتاجها الـ non-login shell عند بداية الدخول الى الـ terminal للتعرف على الاعدادات التي يستخدمها الـ user في التعامل مع الـ shell

```
[root@localhost ~]# cat /etc/bashrc
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.

# are we an interactive shell?
if [ "$PS1" ]; then
    if [ -z "$PROMPT_COMMAND" ]; then
        case $TERM in
        xterm*)
            if [ -e /etc/sysconfig/bash-prompt-xterm ]; then
                PROMPT_COMMAND=/etc/sysconfig/bash-prompt-xterm
            else
                PROMPT_COMMAND='printf "\033]0;%s@%s:%s\007" "${USER}" "${HOSTNAME%%.*}"
${PWD/#$HOME/~}'
            fi
        ;;
        screen)

```

اما بالنسبة لـ normal users فهذه الملفات تسمى .bash_profile و .bashrc . وتوجد في الـ hidden home directory و هي مسئولة عن الـ functions والـ aliases الخاصة بالـ user نفسه ، ويستطيع الـ user التعديل عليها والاضافة لها

```
[ahmed@localhost ~]$ ls ~/.bashrc
/home/ahmed/.bashrc
[ahmed@localhost ~]$ ls ~/.bash_profile
/home/ahmed/.bash_profile
```

وكما نري في الصورة التالية ان ملف .bash_profile يحتوي على الـ functions والـ aliases الخاصة بالـ User والتي يحتاجها الـ system عند بداية الدخول الي الـ user profile

```
[root@localhost ~]# cat ~/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
```

بينما ملف `.bashrc` فهو يحتوي على الـ `aliases` والـ `functions` التي يستخدمها الـ `user` في التعامل مع الـ `non-login shell` (الـ terminal الخاصة به) اذا فالملفات الخاصة بالـ `login shell` هي

```
/etc/profile >>>> for system
.bash_profile >>>> for user
```

وتحتوي هذه الملفات على `system initialization data` عند الـ `login` بينما الملفات الخاصة بالـ `non-login shell` وهي :

```
/etc/bashrc >>>> for system
.bashrc >>>> for user
```

ويمكن ان يقوم الـ `User` بالتعديل على ملفاته واضافة اي `alias` او `function` يريد عملها ولكن الـ `root user` هو الوحيد القادر على تعديل ملفات الـ `/etc/profile` و `/etc/bashrc` وفي كل مرة تقوم بتشغيل الـ `(non-login shell or bash)` يتم تشغيل ملفي `~/.bashrc` و `/etc/bashrc`

ويمكن الدخول على `bash` اخرى من نفس الـ `bash` التي اعمل عليها عن طريق تنفيذ الامر

```
bash
```

فيتم تشغيل bash جديدة (وليس tab جديدة) في نفس ال terminal المفتوحة ، وللخروج نكتب الامر

exit

```
[ahmed@localhost ~]$ bash  
[ahmed@localhost ~]$  
[ahmed@localhost ~]$ su amr  
Password:  
[amr@localhost ahmed]$  
[amr@localhost ahmed]$ exit  
exit  
[ahmed@localhost ~]$
```

ahmed type bash and login to new bash

in new bash, ahmed change the user

amr exit this new bash and return to old one

وال bash الاصلية تسمى parent bash بينما ال bash الجديدة تسمى child bash او shell

وإذا قمنا بالدخول علي bash جديدة من ال bash القديمة فان ال system لا يقوم بتشغيل ال profile files مرة اخري

وإذا استمررنا بكتابه الامر exit فسوف نخرج من ال shell bash الى ال login

وفي اللينكس بشكل عام نجد ان جميع ال initialization files تحتوي على "rc" runlevel . هناك vimrc . المسئولة عن تشغيل ال rc0 , rc1 وهناك المسئولة عن ال initialization هكذا

THE JOB OF THE SHELL

إذا أردنا معرفة الوظيفة الفعلية لـ shell فسيجده أن الـ shell يقوم بالاتي :

COMMAND INTERPRETER

حيث يقوم ال shell بترجمة ال command عن طريق البحث عنه في ال PATH الخاص به، فمثلاً :

عند استخدام الامر cd فان ال command interpreter يعلم ان هذا ال command هو built-in فلا يحتاج للذهاب الى Path الخاص به لمعرفة كيف يقوم بتنفيذها

ولكن عند استخدام الامر `mv` فهو ليس `built-in command` لذلك يجب على الـ `shell` معرفة معنى هذا الـ `command` لكي يستطيع تنفيذه

ويقوم ال shell بترجمة ال command لمعرفة هل هو صحيح ام خاطئ

```
[root@localhost ~]# cd..  
bash: cd..: command not found  
[root@localhost ~]# cd ..  
[root@localhost /]#
```

ALLOWS FOR VARIABLES TO BE SET

يسمح ال shell ايضا لكل user بعمل ال variables الخاصة به والتعديل عليها مثل التعديل على ال HOSTNAME او ال PATH او ال HISTFILESIZE

I/O REDIRECTION

كما يسمح بعمل redirection لـ input وـ output وـ errors

PIPELINES

ويفهم ال shell ايضا ال pipe line

CUSTOMISING YOUR ENVIRONMENT

كما يسمح ال shell لـ User بالتحكم في ال environment الخاصه به عن طريق التحكم في ال variables وـ prompt وـ running scripts وغيرها

CHAPTER 3. REGULAR EXPRESSIONS

INTRODUCTION

الـ regular expression من اهم الاشياء في الـ scripting

WHAT ARE REGULAR EXPRESSIONS?

ال regular expression هي pattern لها مدلول او character له وظيفة معينه مثل (*) الذي يقوم بعمل لأي عدد من ال characters ، وستتحدث الان عن بعض انواع ال pattern

THE FULLSTOP (PERIOD) “.”

الـ fullstop تعنى match any character

فإذا قمنا مثلاً باستخدام الامر sed للبحث عن اي pattern او شكل يبدأ ب "ا" ثم "ن" ثم "ا" ثم يقوم بتبديل "ا" الى "L" و "ا" الى "I" و "ا" الى "N" و "ا" الى "U" ثم يقوم بتبديل ال pattern التالي فقط الى "X" فسيستخدم الامر

```
sed 's/Linu./LINUX/g' fullstop test
```

وهذا ال command يعني

```
s (search for) / Linu. / (replace with) LINUX / g (globally)  
<filename to search>
```

```
[root@localhost ~]# cat fullstop_test
linux
LINUS
Linus
linusac
[root@localhost ~]# sed 's/linu./LINUX/g' fullstop_test
LINUX
LINUS
Linus
LINUXac
[root@localhost ~]#
```

"sed" replace all patterns -not words- start with "l" then "i" then "n" then "u" followed by any character with "LINUX". notice word "linusac"

ولاحظ في الصورة التالية ان "linu" لم تغير ولم يتم استبدالها حيث يجب ان يكون ال pattern متبعا ب character اخر حتى يطبق عليه الشرط والا فلن يتم تطبيق "sed" على هذا ال pattern

```
[root@localhost ~]# cat >> fullstop_test
linu
[root@localhost ~]# sed 's/linu./LINUX/g' fullstop_test
LINUX
LINUS
Linus
LINUXac
linu
```

يمكن استخدام ال sed مع ال pipe كما بالمثال التالي، حيث "nl" يعني number the lines في file

```
sed 's/Linu./LINUX/g' fullstop_test | nl
```

```
[root@localhost ~]# cat fullstop_test
linux
LINUS
Linus
linusac
linu
[root@localhost ~]# sed 's/linu./LINUX/g' fullstop_test | nl
1 LINUX
2 LINUX
3 Linus
4 LINUXac
5 linu
```

LET'S EXPLORE "SED" SYNTAX

لنتحدث قليلا عن ال sed ، فهو يقوم بالبحث عن pattern معين سطر سطر واستبداله بآخر ، pattern يعني stream editor ، وال file المراد البحث فيه عن ال pattern يأتي من ال command العامa لهذا ال form هي

```
sed '[command] / pattern / [replace sequence] / [modifier]' [command]
```

```
sed 's/Linu./LINUX/g' fullstop_test
```

و sed يتبعها اخر يوضح ال action المراد عمله ، وهنا في مثالنا فان هذا ال search هو 's' ويعني command

ثم يأتي بعده ال pattern المراد البحث عنه ونضعه بين علامتين "/" وهو linu في مثالنا ويبيتعه ال replaced sequence وهو ال pattern الجديد الذي سيحل محل القديم ويوضع ايضا بين علامتين "/" وهو "LINUX" في مثالنا

ثم يأتي ال modifier وهو "g" في مثالنا هذا ويعني globally ، وهو يخبر ال sed ان يقوم بعمل هذا ال replace على ال file كله ولا يتوقف عند اول نتيجة

اما اذا اردنا ان نستبدل اول matched pattern فقط فنضع الرقم "1" بدلا من "g"

اما اذا اردنا ان نستبدل ثانى matched pattern فقط فنضع الرقم "2" بدلا من "g"

ثم يأتي في النهاية ال file المراد البحث بداخله عن ال patterns

```
[root@localhost ~]# cat fullstop_test | nl
1 linux
2 LINUS
3 Linus
4 linusac
5 linu
6 linux - linus
[root@localhost ~]# sed 's/linu./LINUX/g' fullstop_test | nl
1 LINUX
2 LINUS
3 Linus
4 LINUXac
5 linu
6 LINUX - LINUX
replace all matched patterns
[root@localhost ~]# sed 's/linu./LINUX/1' fullstop_test | nl
1 LINUX
2 LINUS
3 Linus
4 LINUXac
5 linu
6 LINUX - linus
replace the first matched pattern in each line
[root@localhost ~]# sed 's/linu./LINUX/2' fullstop_test | nl
1 linux
2 LINUS
3 Linus
4 linusac
5 linu
6 linux - LINUX
replace the second matched pattern in each line
[root@localhost ~]#
```

```
sed '/hi/p' myfirstscript
```

يمكن ان نضع "p" بدل "g" لكي يقوم بعمل print على مطابقة كل

```
[root@localhost ~]# cat myfirstscript
hi
this is my first script
[root@localhost ~]# sed '/hi/p' myfirstscript
hi
hi
this is my first script
this is my first script
[root@localhost ~]#
```

sed find the matched pattern. so, he print it

ويمكن ان نضع "d" لكي يقوم بعمل delete لكل ال lines التي تحتوي على "hi"

```
[root@localhost ~]# cat myfirstscript
hi
this is my first script
[root@localhost ~]# sed '/hi/d' myfirstscript
[root@localhost ~]#
```

no result

SQUARE BRACKETS ([]), THE CARET (^) AND THE DOLLAR (\$)

ال ([abc]) تعني انه يقوم بعمل match character J match square brackets ([abc]) واحد من بين الموجودة بين القوسين range of characters

ال (^hmed) تعني انه يقوم بعمل match character caret (^hmed) جميع ال lines التي تبدأ بأي موجود بين الاقواس

ال (ahme\$) تعني انه يقوم بعمل match character Dollar (ahme\$) جميع ال lines التي تنتهي بأي موجود بين الاقواس

```
sed '/^ahl/p' myfirstscript
```

```
[root@localhost ~]# cat myfirstscript
hi
this is my first script
[root@localhost ~]# sed '/^ahl/p' myfirstscript
hi
hi
```

sed find the matched line that start with "h"
so he print it

يمكن استخدام sed مع ال (^) في عمل invert selection عن طريق الصيغة "[ahl]"

```
sed '/^ahl/p' myfirstscript
```

لذلك فان هذا ال command يعني اطبع اي line يبدأ بأي character a,h,l غير ا

```
[root@localhost ~]# sed '/^[^ahl]/p' myfirstscript
hi
this is my first script
this is my first script
```

this line dont start with "a,h,l"

```
sed '/^[^ahl]/d' myfirstscript
```

```
[root@localhost ~]# cat myfirstscript
hi
this is my first script
[root@localhost ~]# sed '/^[^ahl]/d' myfirstscript
hi
[root@localhost ~]#
```

وهذا يعني امسح جميع ال lines التي تبدأ بـ اي حرف غير موجود بين الاقواس
وكما هو الحال في ال (^) يسرى ايضا على ال (\$) ولكنها توضع بعد ال
brackets

```
sed '/[it]$!/d' myfirstscript
```

ويعني هذا ال command ان لا تقوم بمسح كل ال lines التي تنتهي بال "z" او "t" وامسح
الباقي

```
[root@localhost ~]# sed '/[ahl]$!/d' myfirstscript
[root@localhost ~]# sed '/[i]$!/d' myfirstscript
hi
[root@localhost ~]# sed '/[it]$!/d' myfirstscript
hi
this is my first script
[root@localhost ~]#
```

وكما نرى في هذه الصورة ان ال command الاول قام بمسح كل ال lines الموجودة في ال
file: myfirstscript لم يظهر عنه اي نتيجة لأن ال file: myfirstscript
ينتهي بال "a" او "h" او "l" فقام بمسح كل ال lines

بينما في ال command الثاني قام ال sed بالاحتفاظ بال line الذي ينتهي بال "z" ومسح باقي
ال lines ومثله في المثال الثالث

كما يمكن استخدام الـ “^” مع الـ “\$” للدلالة على السطر الفارغ

```
sed '/^$/d' myfirstscript
```

```
[root@localhost ~]# cat myfirstscript
hi
this is my first script

hello

how are you

bye
[root@localhost ~]# sed '/^$/d' myfirstscript
hi
this is my first script
hello
how are you
bye
[root@localhost ~]# sed '/^$/!d' myfirstscript

[root@localhost ~]#
```

وملخص الـ expressions التي استخدمناها هي :

.	any single character
[]	a range of characters
^	start of line (when outside [])
^	do not (when inside [])
\$	end of line
^\$	empty line
*	0 or more of the previous pattern
+	1 or more of the previous pattern
\{n\}	
\{n, \}+	
\{n,m\}	

USING SED AND PIPES

يمكن استخدام الـ sed مع الـ pipe وذلك عند الرغبة في اجراء اي عملية على نتيجة الـ

```
sed '/^[^ahl]/d' bazaar.txt | sed '/^$/d' | nl
```

```
[root@localhost ~]# cat myfirstscript
hi
this is my first script

hello

how are you

bye
[root@localhost ~]# sed '/^[^ahl]/d' myfirstscript
hi

hello

how are you
[root@localhost ~]# sed '/^[^ahl]/d' myfirstscript | sed '/^$/d' | nl
1 hi
2 hello
3 how are you
```

ولتوفير الوقت يمكن دمج الـ 2 sed commands مع بعض بدون استخدام الـ pipe وباستخدام
واحدة فقط مع وضع الـ 2 command options ` `

```
sed '/^[^ahl]/d;/^$/d' bazaar.txt | nl
```

```
[root@localhost ~]# sed '/^[^ahl]/d;/^$/d' myfirstscript|nl
1 hi
2 hello
3 how are you
```

ويمكن استخدام form اخرى تعطى نفس النتيجة

```
cat myfirstscript | sed '/[^ahl]/d;/$/d'
```

```
[root@localhost ~]# cat myfirstscript | sed '/[^ahl]/d;/$/d' | nl
1 hi
2 hello
3 how are you
[root@localhost ~]# ■
```

THE SPLAT (ASTERISK) (*)

ال asterisk يقوم بعمل matching على pattern الموجود قبلها كله سواء كان هذا ال pattern مكرر ام لا

```
sed '/ahm*/p' splatscript
grep "a*" splatscript
```

```
[root@localhost ~]# grep "a*" splatscript
1998
a
ah
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]# grep "ah*" splatscript
a
ah
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]# grep "ahm*" splatscript
ah
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]# grep "ahme*" splatscript
ahm
ahme
ahmed
ahmed1
```

* match the line that contain at least "a"

* match the line that contain at least "ah"

* match the line that contain at least "ahm"

وهي هذه الصورة فان get يعرض ال pattern حتى لو كان ال character السابق لل "*" غير موجود كما في السطر الاول 1999

THE PLUS OPERATOR(+)

ال plus يقوم بعمل matching على الموجود قبلها بشرط ان يكون هذا ال pattern موجود كله

```
[ root@localhost ~]# grep "a\+" splatscript
a
ah
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]# grep "ah\+" splatscript
ah
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]# grep "ahm\+" splatscript
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]# grep "ahme\+" splatscript
ahme
ahmed
ahmed1
[ root@localhost ~]# grep "ahmed\+" splatscript
ahmed
ahmed1
[ root@localhost ~]# grep "ahmed1\+" splatscript
ahmed1
[ root@localhost ~]#
```

+ dont show the line of 1999 but show all lines contains at least "a"

+ show all lines that contain at least "ah"

MATCHING A SPECIFIED NUMBER OF THE PATTERN USING THE CURLY BRACKETS{}

ويعني هذا العنوان انه يمكن تكرار pattern معين لعدد من المرات باستخدام ال { }

$a\{4\} = aaaa$ وهذا الشكل

```
sed '/19\{3\}/p' splatscript
```

ولكن يجب وضع "\\" قبل ال "{" وال "}"

```
sed '/19\\{3\\}/p' splatscript
```

```
[root@localhost ~]# sed '/19{2}/p' splatscript
1998
a
ah
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]# sed '/19\{2\}/p' splatscript
1998
1998
a
ah
ahm
ahme
ahmed
ahmed1
[ root@localhost ~]#
```

sed dont print the line due to wrong syntax

sed print the matched line

او

```
grep "19\{2\}" splatscript
```

```
[root@localhost ~]# grep "19\{2\}" splatscript
1998
```

ولكن اذا اردنا ان عمل matching لا ي matching ال 3 characters فنستخدم ال form

```
grep "\<[a-z][a-z][a-z]\>" 3character
```

```
[root@localhost ~]# grep "\<[a-z][a-z][a-z]\>" 3character
cat
bat
hat
```

وهذه العلامة (\<) تسمى encapsulation للتحكم في عدد ال letters

ويمكن كتابة هذا ال command بهذا الشكل

```
grep "\<[a-z]\{3\}\>" 3character
```

```
[root@localhost ~]# grep "\<[a-z]\{3\}\>" 3character
cat
bat
hat
[root@localhost ~]#
```

A DETOUR - USING A DIFFERENT FIELD SEPARATOR IN SED PATTERN MATCHING

اذا اردنا البحث عن هذا ال pattern في ملف معين
/home/hamish/some_where

فييمكن استخدام ال sed بهذا الشكل

```
sed '/\//home\//hamish\//some_where/!d' pathscript
```

```
[root@localhost ~]# sed '/\//home\//ahmed/!d' pathscript
/home/ahmed
[root@localhost ~]#
```

ولكن هذه ال form صعبة الى حد ما ، ولتسهيل كتابة هذه ال form يمكن استبدال جميع ال
command بـ "%" واحدة فقط في بداية ال regular expressions وفي انتهائه

مع الاخذ في الاعتبار ان علامة "%" يجب ان يسبقها ايضا"\""
وهذا لان علامة ال "%" تقوم تخبر ال shell ان ما سوف يأتي بعدها هو regular expression

```
sed '\%/home/ahmed%!d' pathscript
```

```
[root@localhost ~]# sed '\%/home/ahmed%!d' pathscript
/home/ahmed
```

USING WORD ENCAPSULATING CHARACTERS

هذه العلامة (< >) تسمى encapsulation ويوضع قبلهم "\\" لكي يتعامل معهم ال shell على انهم regular expresions فيصبحوا بهذا الشكل (<\>) ووظيفة ال pattern هي تغليف ال encapsulation للحصول على النتيجة المطلوبه بدقة لاحظ الفرق في الصورة بين الامرین

```
[root@localhost ~]# sed 's/.../321/g' 3character
hi
321
321
321
321lo
[root@localhost ~]# sed 's/\<...>/321/g' 3character
hi
321
321
321
321
hello
```

حيث ان الامر الاول قام ال shell باستبدال اول 3 characters من كل line بالرقم 321 حتى وان كان هناك Line يحتوي على اكثر من 3 characters بينما في الامر الثاني قام ال shell باستبدال ال lines التي تحتوي على 3 characters فقط

RETURNING FROM DETOUR TO OUR DISCUSSION ON CURLY BRACES...

يمكن عمل "a" 2 على الاقل عن طريق

```
sed '/a\{2,\}/!d' infinit
```

وإذا وضعنا داخل القوس {2,4} فهذا يعني match بحد ادنى 2 وبحد اقصى 4

```
sed '/a\{2,4\}/!d' infinit
```

THE TR COMMAND

يعني translate وهو يقوم بتغيير حالة الحرف من ال small capital الي ال

```
[root@localhost ~]# cat > small  
this is test script  
[root@localhost ~]# cat small | tr '[a-z]' '[A-Z]'  
THIS IS TEST SCRIPT  
[root@localhost ~]# cat > capital  
THIS IS FOR TEST  
[root@localhost ~]# cat capital | tr '[A-Z]' '[a-z]'  
this is for test  
[root@localhost ~]#
```

ويمكن تنفيذ ال command بهذا الشكل وعمل redirect لـ file على output

```
[root@localhost ~]# cat capital  
THIS IS FOR TEST  
[root@localhost ~]# tr '[A-Z]' '[a-z]'< capital > capital-result  
[root@localhost ~]# cat capital-result  
this is for test  
[root@localhost ~]#
```

يمكن استخدام ال command reports في تسهيل عمل ال ، حيث يساعدنا على التخلص من ال tabs وال spaces الموجودة بكثرة فيه وذلك عن طريق option : -s الذي يساعد في ضغط ال characters المتشابهه الي character واحد فقط

فمثلا اذا عرضنا الامر free فسنجد انه يحتوي علي الكثير من المسافات بين كل column واخر ، فإذا اردنا ان نضغط هذه المسافات وجعلها مسافة فنستخدم الامر

```
free | tr -s " "
```

ويمكن استبدال ال character بأي space اخر نريد ضغطه

```
[root@localhost ~]# free
total used free shared buffers cached
Mem: 500064 489564 10500 0 63468 93288
-/+ buffers/cache: 332808 167256
Swap: 1023996 27568 996428
[root@localhost ~]# free | tr -s " "
total used free shared buffers cached
Mem: 500064 489564 10500 0 63468 93288
-/+ buffers/cache: 332808 167256
Swap: 1023996 27568 996428
[root@localhost ~]# free | tr -s "0"
total used free shared buffers cached
Mem: 5064 489564 1050 0 63468 93288
-/+ buffers/cache: 332808 167256
Swap: 1023996 27568 996428
[root@localhost ~]#
```

THE CUT COMMAND

ال cut command : يستخدم في قطع جزء معين من ال line باستخدام delimiter لتحديد الفواصل التي سيتم القطع على أساسها

وهذه ال delimiters موجودة في ال shell variables باسم IFS (Input Field Separators)

```
[root@localhost ~]# set | grep IFS
IFS=$' \t\n'
[root@localhost ~]#
```

وكما نري في الصورة فان ال المعرفة لدى ال cut هي
" " = new line

```
free | tr -s ' ' | grep Mem | cut -d" " -f1-3
```

```
[root@localhost ~]# free | tr -s ' ' | grep Mem | cut -d" " -f1-3
Mem: 500064 482992
[root@localhost ~]#
```

-d: delimiter حيث

-f : field no. وال

وال field هو القيمة التي بعد المسافة ، بمعنى ان السطر ينقسم الى fields يفصل بينها مسافات (delimiter)

ويمكن ان نستخدم "c" لعمل cut على characters

```
free | tr -s ' ' | grep Mem | cut -c6-13,15,17
```

```
[root@localhost ~]# free | tr -s ' ' | grep Mem  
Mem: 500064 485864 14200 0 64404 86360  
[root@localhost ~]# free | tr -s ' ' | grep Mem | cut -c6-13,15,17  
500064 464  
[root@localhost ~]#
```

واذا اردت ان تستخدم tab بدلا من space ولكن ال character عبارة عن delimiter واحد فقط tab ولا يمكن ان نكتب ال tab بهذا الشكل "\t" كما في ال IFS ، لذلك نضغط CTRL+v ثم tab

THE PASTE COMMAND

ال paste command يقوم بدمج محتويات 2 files بين كل line ويضع "tab" حيث يوجد

```
paste pastescript pasteequal
```

```
[root@localhost ~]# cat pastescript  
hiiiiiiiiiiii  
this is the original file  
[root@localhost ~]# cat pasteequal  
hiiiiiiiiiiii  
this is test script for equal line  
[root@localhost ~]# cat pastediff  
hiiiiiiiiiiiiiiii  
this is a test script for diff no. of lines  
we will see the differents  
[root@localhost ~]# paste pastescript pasteequal  
hiiiiiiiiiiii hiiiiiiiiiiii  
this is the original file this is test script for equal line  
[root@localhost ~]# paste pastescript pastediff  
hiiiiiiiiiiii hiiiiiiiiiiiiiiiiii  
this is the original file this is a test script for diff no. of lines  
we will see the differents
```

this is the original file (2 lines)

this is a file with equal lines (2 lines)

this is a file with different lines (3 lines)

ويمكن استخدام command: cut كما يحدث في ال delimiters ولكن هنا يوضع هذا ال tab بدلا من ال delimiter

```
paste -d";" pastescript pasteequal
```

```
[root@localhost ~]# paste -d";" pastescript pastediff  
hiiiiiiiiiiii;hiiiiiiiiiiiiiiii  
this is the original file;this is a test script for diff no. of lines  
;we will see the differents  
[root@localhost ~]#
```

THE UNIQ COMMAND

الامر uniq يقوم بحذف السطور المكررة بشرط ان تكون متماثلة
فإذا قمنا باستعراض محتويات ملف /etc/passwd واردنا معرفة ما هي ال shells المستخدمة

```
[root@localhost ~]# cut -d":" -f7 /etc/passwd  
/bin/bash  
/sbin/nologin  
/sbin/nologin  
/sbin/nologin  
/sbin/nologin  
/bin/sync  
/sbin/shutdown  
/sbin/halt  
/sbin/nologin  
/sbin/nologin  
/sbin/nologin
```

فنجد ان الملف يحتوي على الكثير من السطور المكتوب بها نفس ال shells
فنقوم باستخدام uniq لكي نقوم بحذف السطور المتشابهة والحفاظ على المختلف فقط

```
cut -d":" -f7 /etc/passwd | uniq
```

```
[root@localhost ~]# cut -d":" -f7 /etc/passwd | uniq  
/bin/bash  
/sbin/nologin  
/bin/sync  
/sbin/shutdown  
/sbin/halt  
/sbin/nologin  
/bin/sh  
/sbin/nologin  
/bin/bash
```

وهنا نجد ان uniq قام بحذف كل السطور المتتالية والمتتشابهه ، ولكن ما زالت تظهر shells متتشابهه ، فالحل ان نقوم بعمل sort على file اولا ثم نقوم بعمل uniq

```
cut -d":" -f7 /etc/passwd | sort | uniq
```

```
[root@localhost ~]# cut -d":" -f7 /etc/passwd | sort | uniq  
/bin/bash  
/bin/sh  
/bin/sync  
/sbin/halt  
/sbin/nologin  
/sbin/shutdown  
[root@localhost ~]# █
```

THE SORT COMMAND

ال sort command يقوم بترتيب محتويات الملف من الاصغر الي الاكبر ومن "a" الي "z"

```
[root@localhost ~]# cat sort-script
0
1           file contents
2
3
2
[root@localhost ~]# sort sort-script
0
1           sort file contents
2
2
3
[root@localhost ~]# sort -u sort-script
0
1           sort file contents with uniq line
2           = sort -u sort-script | uniq
3
3
[root@localhost ~]# sort -ur sort-script
3
2           reverse sort for file contents
1
0
[root@localhost ~]#
```

يمكن استخدام الـ `-o` : مع الامر `sort` بدلا من علامة الـ redirection وهو يعني output

```
cut -d: -f7 /etc/passwd | sort -ruo test
```

```
[root@localhost ~]# cut -d":" -f7 /etc/passwd | sort -ruo test
[root@localhost ~]# cat test
/sbin/shutdown
/sbin/nologin
/sbin/halt
/bin/sync
/bin/sh
/bin/bash
[root@localhost ~]#
```

وهذه بعض الـ options التي تستخدم مع الامر `sort`

Option	Action
--------	--------

-o	يقوم بعمل redirect على file بدلا من ">"
-u	لعمل uniq sort
+t	هو ال delimiter الخاص بال sort
+3	عدد ال fields التي سيجاها لها
-n	لترتيب السطور حسب الارقام وليس الحروف
-r	لعمل reverse sort

```
cut -d":" -f3,7 /etc/passwd | sort -t: -rn
```

```
[root@localhost ~]# cut -d":" -f3,7 /etc/passwd | sort -t":" -rn
65534:/sbin/nologin
1001:/bin/bash
1000:/bin/bash
999:/sbin/nologin
998:/sbin/nologin
```

THE GREP COMMAND

يمكن ان نستخدم ال grep ك filter او للنظر داخل file كما يستطيع التعامل مع ال regular expression

اذا اردنا استخدام ال grep للنظر داخل file فيمكن ان نستخدم هذه ال form

```
grep "/bin/bash" /etc/passwd
```

```
[root@localhost ~]# grep "/bin/bash" /etc/passwd
root:x:0:0:root:/root:/bin/bash
postgres:x:26:26:PostgreSQL Server:/var/lib/pgsql:/bin/bash
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
news:x:9:13:News server user:/etc/news:/bin/bash
ahmed:x:1000:1000:ahmed:/home/ahmed:/bin/bash
amr:x:1001:1001::/home/amr:/bin/bash
[root@localhost ~]#
```

ولأن ال pattern الذي نبحث عنه يحتوي علي "/" فقمنا بوضع هذا ال pattern داخل علامتين "" ، وكذلك يجب ان نعرف ان ال grep لا يبحث عن كلمة ولكن يبحث عن pattern يبدأ ب "/" ثم "b" الخ

وهناك بعض ال options المفيدة لل grep فمثلا :

يمكن اضافة -B1 option لكي يقوم بعرض عدد معين من السطور قبل ال pattern

```
[root@localhost ~]# cat grepscript  
line 1  
line 2  
line 3  
line 3  
line 2  
line 1  
[root@localhost ~]# grep -B1 'line 2' grepscript  
line 1  
line 2  
--  
line 3  
line 2  
[root@localhost ~]#
```

show one line before the pattern

ويمكن اضافة -A1 لعرض سطر واحد بعد ال pattern

```
[root@localhost ~]# cat grepscript  
line 1  
line 2  
line 3  
line 3  
line 2  
line 1  
[root@localhost ~]# grep -A1 'line 2' grepscript  
line 2  
line 3  
--  
line 2  
line 1  
[root@localhost ~]#
```

show one line after the pattern

this mean that there is unmatched lines in between

ويمكن استخدام -v لعمل reverse pattern بمعنى انه يعرض كل السطور ما عدا ال pattern

```
[root@localhost ~]# grep -v 'line 2' grepscript
line 1
line 3
line 3
line 1
[root@localhost ~]#
```

ويمكن استخدام n - لعمل label لـ pattern (يضع رقم السطر الذي يحتوي على ال pattern)

```
[root@localhost ~]# cat grepscript
line 1
line 2
line 3
line 3
line 2
line 1
[root@localhost ~]# grep -n 'line 2' grepscript
2:line 2
5:line 2
[root@localhost ~]#
```

Option	Action
-B1	يعرض ال pattern وال line الذي يسبقه
-A1	يعرض ال pattern وال line الذي يليه
-v	يعرض كل السطور عدا التي تحتوي على ال pattern
-n	يضع رقم السطر الذي يحتوي على ال pattern بجانبه

اذا اردنا ان نستخرج كلمة ahmed من جميع ال files الموجودة في ال working directory فيمكن ان نستخدم ال form التالية :

```
[root@localhost ~]# grep -n 'ahmed' *
pathscript:1:/home/ahmed
sort-script:2:my name is ahmed
splatscript:6:ahmed
splatscript:7:ahmed1
test:50:ahmed:1000:/bin/bash
[root@localhost ~]#
```

حيث قمنا باستبدال اسم ال file ب "*" حتى يبحث في كل ال files ، وهنا قام بعرض اسم ال file الذي يحتوي على ال Pattern ورقم السطر في ال file

اما اذا اردنا استخدام grep كـ filter فيكون ذلك في حالة استخدام ال pipe مع ال grep حيث يدخل ال output كاملا على ال grep يقوم بعمل filtration له واستخراج ال data المطلوبة ويعمل ذلك بوضوح في التعامل مع ال logs كما في المثال التالي :

```
tail -10 /var/log/messages | grep '19:[0-5][0-9]:[0-5][0-9]'
```

```
[root@localhost ~]# tail -10 /var/log/messages | grep 'Nov 15 19:[0-1][0-0]:[0-5][0-9]' | nl
1 Nov 15 19:10:36 localhost dbus-daemon[826]: dbus[826]: [system] Activating service name='org.
2 Nov 15 19:10:36 localhost dbus[826]: [system] Activating service name='org.freedesktop.Packa
3 Nov 15 19:10:36 localhost dbus-daemon[826]: dbus[826]: [system] Successfully activated servi
4 Nov 15 19:10:36 localhost dbus[826]: [system] Successfully activated service 'org.freedeskto
[root@localhost ~]# █
```

فهنا اخذنا ال output من tail وهو 10 سطور وادخلناه علي grep فقام بعمل filter له واخرج 4 سطور فقط

GREP, EGREP AND FGREP

هناك 3 انواع من ال grep

Type	Function
Grep	هو ال standard grep
Egrep	يستخدم بكثرة مع ال extended regular expressions ولكن بطئ
Fgrep	لا يستخدم مع ال regular expressions لذلك فهو سريع

وال اسرع قليلا من ال grep

CHAPTER 4. PRACTICALLY SHELL SCRIPTING

SECTION TECHNIQUES TO USE WHEN WRITING, SAVING AND EXECUTING SHELL SCRIPTS

دعونا نتفق على اننا في بداية عمل اي script يجب ان نبدأ بتطبيق ال commands على ال script file وبعد التأكد من عملها بشكل جيد نقوم بنقلها الى ال command line فاما اردنا عمل script يقوم لمعرفة ال shells المستخدمة في ال system فنقوم اولاً بتطبيق ال terminal على ال command

```
cat -d: -f7 /etc/passwd | sort -u
```

ثم نقوم بعد ذلك بوضع هذا ال command في ملف ونسميه مثلا script.sh ولكن هناك بعض القواعد يجب ان نعلمها عن ال script وسوف نتعرف عليها الان

DETOUR: FILE EXTENSION LABELS

1- اللينكس لا يهتم بال extensions

ولا يعرفه لأن القاعدة الأساسية فيه أن everything is a file ولكننا قد نضيف الامتداد الي اسم file لكي نتذكر نحن ان هذا ال file هو script او ملف نصي الخ

script
script.sh

2- ال script يجب ان يكون executable وذلك بتنفيذ الامر

```
chmod +x script.sh
```

3- لتنفيذ ال script يجب ان نضع امامه "./" او "sh" حتى يعرف ال shell ان هذا ال file هو script

```
./script
```

COMMENTS IN SCRIPTS

اذا اردنا ان نشرح وظيفة سطر معين في ال script فيمكن ان نكتبه بداخل ال script علي اعتبار انه يقوم shell بتنفيذها ، ولعمل ذلك نضع (#) امام ال line المراد اعتباره comment

```
#A script to produce the unique shells using the /etc/passwd  
file  
cut -d: -f7 /etc/passwd |sort -u
```

VARIABLES

ال variables هي طريقة مؤقتة لتخزين المعلومات ويتم حجز مساحة في ال memory باسم variable ويوضع فيها ال variable name

```
NAME="ahmed"
```

حيث ال ahmed هو variable name ، وال NAME هو variable name
وهناك قاعدتين مهمتين للتعامل مع ال variables
-1 ال variable يجب ان يكون capital كما نرى كلمة NAME
-2 لا توجد اي مسافات بين ال variable name وعلامة "=" ولا بين ال " " وال value
وللتعامل مع هذا ال variable والاستفادة بال value يجب ان نضع امامه علامة "\$"

```
echo "$NAME"
```

```
[root@localhost ~]# NAME=ahmed  
[root@localhost ~]# echo $NAME  
ahmed  
[root@localhost ~]# echo NAME  
NAME  
[root@localhost ~]#
```

the value not appear

وهذه ابسط اشكال كتابة ال script

```
[root@localhost ~]# cat test
echo "ahmed"
[root@localhost ~]# sh test
ahmed
[root@localhost ~]# ./test
ahmed
```

SHEBANG OR HASHPLING!#

الكثير منا يستخدم ال bash لكن هناك البعض قد لا يستخدمه، واذا قمت بكتابة script معين على ال Bash فقد لا يعمل بشكل صحيح علي نوع اخر من ال shells لذلك فيجب اجبار ال script ان يعمل في shell معينه وذلك عن طريق واحد من الطريقتين :

- تشغيل ال script باستخدام الامر sh

```
sh script.sh
```

- اضافة ال shebang

وال shebang هو مزيج من ال "# ! : bang : hash" توضع بداخل ال script في بدايته للدلالة علي نوع ال interpreter الذي سيتعامل مع ال

```
#!/bin/bash
```

EXIT

بعد ان عرفنا الطريقة ال standard لبداية ال script سنتعرف علي الطريقة التي يجب ان تنهي بها ال script وذلك عن طريق ما يعرف بال ending status

فعقب انتهاء عمل اي برنامج تخرج قيمة معينة توضح كيف انتهي عمل هذا البرنامج (هل انتهي بشكل صحيح ام به اخطاء) وهذه القيمة تسمى ال exit value

إذا انتهي البرنامج بشكل صحيح ولم يحدث errors فتكون ال exit value = 0

اما اذا لم ينتهي البرنامج بشكل صحيح وحدث بعض ال errors فتكون ال exit value = 1-255

Exit Value	Exit Status
0 (Zero)	Success

Non-zero	Failure
2	Incorrect usage
127	Command Not found
126	Not an executable my be the script is not executable

لذلك يجب معرفة ال exit status في نهاية كل script عن طريق اضافة هذا السطر في نهاية ال script

```
echo $?
```

```
[root@localhost ~]# cat correct_script
echo "ahmed"
echo $?
[root@localhost ~]# sh correct_script
ahmed
0
[root@localhost ~]# cat err_script
echo "ahmed"
ls --option
echo $?
[root@localhost ~]# sh err_script
ahmed
ls: unrecognized option '--option'
Try `ls --help' for more information.
2
[root@localhost ~]#
```

NULL AND UNSET VARIABLES

هناك بعض ال variables الهامة التي يجب ان نتحدث عنها مثل null و unset و يعني ان ال variable هو ان ال null موجودة (set) ولكن ليس لها قيمة (null) كال التالي

```
NAME=""
NAME=
```

وهنا فان ال Variable=NAME موجود ولكن قيمته null ولاحظ عدم وجود مسافة بين ال " " وينتج عن هذا ال variable سطر فارغ

```
[root@localhost ~]# NULL=""  
[root@localhost ~]# echo $NULL  
  
[root@localhost ~]# NULL=  
[root@localhost ~]# echo $NULL  
  
[root@localhost ~]#
```

ولكن هذا لا يساوي

```
NAME=" "
```

لان المسافة بين ال " " تعبّر عن value لذلك فهي تساوي

```
NAME="ahmed"
```

اما ال Unset فهي تقوم بمسح ال value المخصص لل variable كما يلي

```
unset NAME
```

```
[root@localhost ~]# NAME=ahmed  
[root@localhost ~]# echo $NAME  
ahmed  
[root@localhost ~]# unset NAME  
[root@localhost ~]# echo $NAME  
bash: NAME: unbound variable  
[root@localhost ~]#
```

ويجب ان نفرق بين ال null value وال empty value ،ولكي يتضح الفرق اكثر نري المثال التالي ولاحظ الفرق بين الثلاث :variables

```

[ root@localhost ~]# NAME=""           " → empty variable
[ root@localhost ~]# echo "${NAME}x"
      x
[ root@localhost ~]# unset NAME → unset variable
[ root@localhost ~]# echo "${NAME}x"
bash: NAME: unbound variable
[ root@localhost ~]# NAME="" → null variable
[ root@localhost ~]# echo "${NAME}x"
      x
[ root@localhost ~]#

```

VARIABLE EXPANSION

وهنا سنتحدث عن كيف يقوم ال shell بترجمة ال variables ، فإذا كان variable=NAME ولها value = ahmed

```
NAME="ahmed"
```

وقمت باستخدام ال echo لعرض قيمة ال Value

```
echo $NAME
```

فسيكون الناتج كما بالصورة

```

[ root@localhost ~]# NAME="ahmed"
[ root@localhost ~]# echo $NAME
ahmed
[ root@localhost ~]#

```

ولكن ماذا اذا كتبنا ال command بهذا الشكل

```
echo $NAMEX
```

فإن ال shell سيجد أن ال NAMEX مكتوبة كاملة خلف ال \$ فيظن أن هناك variable اسمها NAMEX ولكن عندما يبحث عنها فلن يجدتها وستكون النتيجة error كما يلي

```
[root@localhost ~]# echo $NAMEX  
bash: NAMEX: unbound variable  
[root@localhost ~]#
```

انظر الي الصورة التالية ولاحظ الفرق جيدا

```
[root@localhost ~]# NAME="ahmed"  
[root@localhost ~]# echo $NAME  
ahmed  
[root@localhost ~]# echo $NAMEX  
bash: NAMEX: unbound variable  
[root@localhost ~]# echo $NAME X  
ahmed X  
[root@localhost ~]#
```

وافضل طريقة للتفرق بين ال variable نفسه واي شيء يليه هي ان نضع ال variable بين curly brackets {}

```
echo ${NAME} X
```

```
[root@localhost ~]# echo ${NAME}X  
ahmedX  
[root@localhost ~]#
```

وسوف يفهم ال shell ان ال variable الموجودة بين الاقواس وان ال X ليس جزء منها

ENVIRONMENT VS SHELL VARIABLES

ال **variable** هي ال **shell variable** التي تطبق على ال shell التي نعمل عليها حاليا فقط ولن تطبق على اي shell جديدة نفتحها حتى اذا كانت shell مفتوحة داخل ال shell باستخدام الامر Bash

```
[ahmed@localhost ~]$ NAME=ahmed
[ahmed@localhost ~]$ echo $NAME
ahmed
[ahmed@localhost ~]$ bash
[ahmed@localhost ~]$ echo $NAME
[ahmed@localhost ~]$
```

ال **environment variables** تطبق على كل ال shells الموجودة في ال system ويقرأها login عند ال system

وإذا أردت أن أحوال ال shell variable نستخدم الأمر export

```
export NAME
```

```
[ahmed@localhost ~]$ NAME=ahmed
[ahmed@localhost ~]$ echo $NAME → here is the working shell
ahmed
[ahmed@localhost ~]$ bash → here is the new shell that
[ahmed@localhost ~]$ echo $NAME show no values
[ahmed@localhost ~]$ exit → exit the new shell and be sure
exit → that the variable still exist
[ahmed@localhost ~]$ echo $NAME
ahmed
[ahmed@localhost ~]$ export NAME → run export command
[ahmed@localhost ~]$ bash → open new shell and we
[ahmed@localhost ~]$ echo $NAME found the value
ahmed
[ahmed@localhost ~]$
```

ARITHMETIC IN THE SHELL

سنتحدث الان عن التعامل مع العلامات الحسابية باستخدام ال scripting وسنتحدث عن ال integer arithmetic

وهناك عدة طرق للتعامل مع القيم الحسابية الصحيحة باستخدام ال scripts :

1- وضع ال expression الحسابي او المعادلة الحسابية في قوسين مزدوجين (())

```
[ahmed@localhost ~]$ i=10
[ahmed@localhost ~]$ echo $i
10
[ahmed@localhost ~]$ echo $((i=i+5))
15
[ahmed@localhost ~]$ █
```

وهذه بعض العلامات الحسابية التي تستخدم في ال script

Arithmetic operators	action
+	علامة الجمع
-	علامة الطرح
*	علامة الضرب
/	علامة القسمة
%	علامة النسبة المئوية

ولاحظ ان كل هذه الطرق لكتابة ال expression تؤدي الى نفس النتيجة

```
((J=I+5))
J=$((I+5))
J=$(( I + 5 ))
$(( J = I + 5 ))
```

ولكن وجود مسافات قبل وبعد علامة "=" سوف ينتج عنه error

```
J = $(( ( I + 5 ) ))
```

```
[ahmed@localhost ~]$ j = $((i+5))
bash: j: command not found...
[ahmed@localhost ~]$ █
```

CHAPTER 5. USING QUOTATION MARKS

INTRODUCTION

في هذا ال chapter سنتحدث عن ال quotation marks ووظيفتها واختلافها عن الآخر ، وهناك 3 انواع من ال quotation marks

Our term	Real term	Symbol
Ticks	Single quotes	'
Back ticks	Back quotes	' (على حرف ال "ذ")
Quotes	Double quotes	"

SINGLE QUOTES OR "TICKS"

- نستخدم ال single quotes اذا اردنا طباعة ال characters كاملة بنفس الشكل حتى اذا كانت تحتوي على special characters فهي لا تترجمها

```
[ahmed@localhost ~]$ echo '$NAME'  
$NAME  
[ahmed@localhost ~]$ echo '$NAME'  
"$NAME"  
[ahmed@localhost ~]$ █
```

- لا يفضل استخدامه اثناء التعامل مع ال files التي تحتوي على spaces ، لاحظ الشكل التالي

```
[ahmed@localhost ~]$ NAME='ahmed@gmail'  
[ahmed@localhost ~]$ echo $NAME  
ahmed@gmail  
[ahmed@localhost ~]$ NAME='ahmed [REDACTED] gmail'  
[ahmed@localhost ~]$ echo $NAME  
ahmed@gmail
```

-3 يمكن استخدامها لعمل set لل variables ولا تستخدم في التعامل مع ال commands الموجودة في ال

```
[ahmed@localhost ~]$ NAME='ahmed gmail'  
[ahmed@localhost ~]$ echo $NAME  
ahmed gmail  
[ahmed@localhost ~]$ █
```

لاحظ ان ال shell اعتبرت ان ال value هي ahmed gmail كلها لانها مغلفة بال quotes

-4 يمكن استخدامها لعمل new line

```
[ahmed@localhost ~]$ echo 'hello  
> world'  
hello  
world  
[ahmed@localhost ~]$ █
```

DOUBLE QUOTES

-5 تستخدم مع ال special characters

```
[ahmed@localhost ~]$ NAME="ahmed"  
[ahmed@localhost ~]$ echo "$NAME"  
ahmed  
[ahmed@localhost ~]$ █
```

والاحظ ان هذه ال commands يؤدوا الي نفس النتيجة

```
[ahmed@localhost ~]$ echo hello my name is $NAME
hello my name is ahmed
[ahmed@localhost ~]$ echo "hello my name is $NAME"
hello my name is ahmed
[ahmed@localhost ~]$ echo hello my name is "$NAME"
hello my name is ahmed
[ahmed@localhost ~]$ echo "hello my name is \"$NAME\""
hello my name is ahmed
[ahmed@localhost ~]$
```

كل الـ special characters الموجودة بداخل الـ "" يتم التعامل معها كـ عادي ولا يتم عمل expansion لها عدا الـ \$ و `

```
[ahmed@localhost ~]$ echo "\$NAME"
\$NAME
[ahmed@localhost ~]$ echo "$NAME"
ahmed
[ahmed@localhost ~]$
```

اذا اردنا استخدام الـ \$" كـ character عادي وليس فنفع علامة "\" كال التالي

```
[ahmed@localhost ~]$ echo "this cost \$10"
this cost $10
```

وهنا نجد ان الـ shell يعامل الـ "\" كـ escape character تقوم بتحويل الـ special character الذي يليها الى character عادي لطبي لا يتم ترجمته يمكن استخدام الـ \" بشكل اخر مع الامر -e الذي يقوم بتغيير طريقة تعامل الـ shell مع الـ "\" حيث يكون

Action	Command
\n	النزول لسطر جديد
\t	اضافة tab

```
[ahmed@localhost ~]$ echo -e "my name is \t ahmed"
my name is      ahmed
[ahmed@localhost ~]$ echo -e "my name is \n ahmed"
my name is
ahmed
```

BACKTICKS

تستخدم ال (`) اذا اردنا ان ننفذ command ونأخذ ال output ونستخدمه مرة اخري، فاذا اردنا عرض التاريخ الحالي فنستخدم الامر

```
date
```

```
[ahmed@localhost ~]$ date
Sat Nov 17 15:05:42 EET 2012
[ahmed@localhost ~]$
```

ولكن اذا اردنا ان نعرض التاريخ الحالي اثناء تشغيل ال script فيجب ان نضعه في variable ثم نستخدم هذا ال Variable في ال script بهذا الشكل

```
[ahmed@localhost ~]$ DATE=`date`
[ahmed@localhost ~]$ echo $DATE
Sat Nov 17 15:02:50 EET 2012
[ahmed@localhost ~]$
```

ولكن لاحظ ان قيمة ال DATE ستظل كما هي ولن تتغير مرة اخري لان ال variable يتغير مرة واحدة فقط اذا تم تشغيل ال command بين ال backticks

اذا فما الحل ؟

يجب عمل DATE reset لـ كل مرة نريد ان يظهر ال date الحالي عن طريق اعادة تخصيص ال variable مرة اخري

```
DATE=`date`
```

وبذلك يقوم ال DATE بقراءة ال value الجديدة لـ variable

```
[ahmed@localhost ~]$ DATE=`date`  
[ahmed@localhost ~]$ echo $DATE  
Sun Nov 18 08:47:15 EET 2012  
[ahmed@localhost ~]$
```

كما يمكن استخدام الـ **DATE variable** لعرض الوقت وحده عن طريق هذا الـ command

```
TIME=`echo $DATE | cut -d' ' -f4`
```

```
[ahmed@localhost ~]$ TIME=`echo $DATE | cut -d' ' -f4`  
[ahmed@localhost ~]$ echo $TIME  
08:49:46
```

وبذلك فان التغيير في الـ **DATE variable** يتبعه تغيير في الـ **TIME variable** ولكن ماذا اذا قمنا بتنفيذ الـ back ticks نفسها بدون تخصيصها لـ **variable** كما في المثال التالي

```
[ahmed@localhost ~]$ `echo $DATE|cut -d" " -f4`  
bash: 08:49:46: command not found...
```

سنجد ان الامر تم تنفيذه ولكن خرجت النتيجة الى الـ command line لتنفيذها ، لذلك ظهر الـ error لانه لا يوجد command اسمه 08:49:46 ، لذلك يفضل تخصيص الـ back ticks commands

يمكن ايضا استخدام علامة (\$) بدلا من الـ back ticks لان الـ back ticks ربما يصعب رؤيتها في الـ script

```
[ahmed@localhost ~]$ DATE=$(date)  
[ahmed@localhost ~]$ echo $DATE  
Sun Nov 18 09:15:54 EET 2012  
[ahmed@localhost ~]$
```

SHELL ARITHMETIC'S WITH EXPR AND BACK QUOTES

لقد تحدثنا سابقا عن (())\$ التي تستخدم في العمليات الحسابية ولكن هناك بعض الـ shells القديمة لا تفهمها لذلك يمكن الاستعاضة عنها بـ command expr باسمه كالاتي

```
i=0  
i=$((i+1))
```

وهذا يساوي

```
i=0  
i=$(expr i+1)
```

وهو يساوي ايضا

```
i=`expr i+1`
```

ANOTHER TIP WHEN USING QUOTATION MARKS

اذا قمنا بتخصيص variable : file لعرض نتجة الامر ls وقمنا بعرض النتيجة

```
file=`ls -al`  
echo $file
```

فستظهر النتيجة كلها في سطر واحد Unformatted

```
[ahmed@localhost ~]$ file=`ls -al`  
[ahmed@localhost ~]$ echo $file  
total 136 drwx----- 20 ahmed ahmed 4096 Nov 17 13:38 . drwxr-xr-x. 5 root root 4096 Nov 11 2  
1:38 .. drwxrwxr-x. 2 ahmed ahmed 4096 Nov 11 20:42 .abrt -rw----- 1 ahmed ahmed 826 Nov 16  
11:18 .bash_history -rw-r--r-- 1 ahmed ahmed 18 Jun 22 2011 .bash_logout -rw-r--r-- 1 ahmed  
ahmed 193 Jun 22 2011 .bash_profile -rw-r--r-- 1 ahmed ahmed 164 Nov 11 22:04 .bashrc drwx--  
---- 4 ahmed ahmed 4096 Nov 11 20:42 .cache drwx----- 7 ahmed ahmed 4096 Nov 11 20:42 .conf  
ig drwx----- 3 ahmed ahmed 4096 Nov 11 20:42 .dbus drwxr-xr-x. 2 ahmed ahmed 4096 Nov 11 20:  
42 Desktop drwxr-xr-x. 2 ahmed ahmed 4096 Nov 11 20:42 Documents drwxr-xr-x. 2 ahmed ahmed 409  
6 Nov 11 20:42 Downloads -rw----- 1 ahmed ahmed 16 Nov 11 20:42 .esd_auth drwx----- 3 ah  
med ahmed 4096 Nov 11 20:42 .gconf drwxr-xr-x. 3 ahmed ahmed 4096 Nov 11 20:42 .gnome2 -rw-rw-r  
-- 1 ahmed ahmed 137 Nov 11 20:42 .gtk-bookmarks drwx----- 2 ahmed ahmed 4096 Nov 11 20:42  
.gvfs -rw----- 1 ahmed ahmed 310 Nov 11 20:42 .ICEauthority -rw-r--r-- 1 ahmed ahmed 1369  
Nov 11 20:42 .imsettings.log drwxr-xr-x. 3 ahmed ahmed 4096 Nov 11 20:42 .local drwxr-xr-x. 4  
ahmed ahmed 4096 Aug 7 02:58 .mozilla drwxr-xr-x. 2 ahmed ahmed 4096 Nov 11 20:42 Music drwxr-  
xr-x. 2 ahmed ahmed 4096 Nov 11 20:42 Pictures drwxr-xr-x. 2 ahmed ahmed 4096 Nov 11 20:42 Pub  
lic drwx----- 2 ahmed ahmed 4096 Nov 11 20:42 .pulse -rw----- 1 ahmed ahmed 256 Nov 11 20  
:42 .pulse-cookie drwxr-xr-x. 2 ahmed ahmed 4096 Nov 11 20:42 Templates drwxr-xr-x. 2 ahmed ah  
med 4096 Nov 11 20:42 Videos -rw----- 1 ahmed ahmed 3446 Nov 17 09:09 .viminfo -rw-----  
1 ahmed ahmed 9047 Nov 12 20:38 .xsession-errors -rw-r--r-- 1 ahmed ahmed 658 Feb 8 2011 .zsh  
rc
```

ولكي نعالج هذه المشكلة فيجب ان نضيف ""

```
echo "$file"
```

```
[ahmed@localhost ~]$ echo "$file"
total 136
drwx----- 20 ahmed ahmed 4096 Nov 17 13:38 .
drwxr-xr-x 5 root root 4096 Nov 11 21:38 ..
drwxrwxr-x 2 ahmed ahmed 4096 Nov 11 20:42 .abrt
-rw----- 1 ahmed ahmed 826 Nov 16 11:18 .bash_history
-rw-r--r-- 1 ahmed ahmed 18 Jun 22 2011 .bash_logout
-rw-r--r-- 1 ahmed ahmed 193 Jun 22 2011 .bash_profile
-rw-r--r-- 1 ahmed ahmed 164 Nov 11 22:04 bashrc
```

CHAPTER 6. SO, YOU WANT AN ARGUMENTS?

INTRODUCTION

كما علمنا ان لتشغيل اي script يجب ان يكون كالتالي

```
./script.sh
```

ولكن يمكن ايضا تشغيله كالتالي

```
./eatout.sh file.txt
```

وهذه ال form مفيدة اذا اردنا تنفيذ ال script علي file معين
ويمكن تنفيذ ال script علي كلمة توجد بداخل ال file ، وهذه الكلمة تسمى هنا argument

```
./eatout.sh Word-in-file.txt
```

نفترض ان لدينا script : eat.sh و file : restaurant.txt عن المطاعم
ونريد ان نستخرج بيانات نوع معين من المطعم وليكن Italian

```
[root@localhost ~]# cat restaurant.txt
mart,Parks,6834948,9
italian,Bardellis,6973434,5
steakhouse,Nelsons Eye,6361017,8
steakhouse,Butchers,Grill,6741326,7
smart,Joes,6781234,5
[root@localhost ~]#
```

وان ال script يحتوي على الاتي

```
[root@localhost ~]# cat eat.sh
#!/bin/bash
TYPE=$1
grep $TYPE restaurant.txt
```

ولكي نقوم بتنفيذ هذا ال script نقوم بكتابه الامر بالشكل التالي

```
./eat.sh Italian
```

لاحظ جيدا ان السطر الثاني في ال script يحتوي علي الرمز \$1 فما معناه وهل هناك \$0 و \$2 الخ

نعم فال shell تقوم بتقسيم هذا الامر الي مجموعة اجزاء

```
./eat.sh Italian
```

حيث تمثل ./eat.sh بالرمز \$0

وال Italian بالرمز \$1

واذا وضعنا اي اخر بعد argument italiano فيرمز له بالرمز \$2 الخ

وهذه الرموز تسمى positional parameters

فاما قمنا بتعديل ال script وجعلنا ال TYPE = \$2

```
[root@localhost ~]# cat eat.sh
#!/bin/bash
TYPE=$2
grep $TYPE restaurant.txt
[root@localhost ~]#
```

وقمي بتنفيذ ال script نجده قام بعرض بيانات ال الثاني وهو smart

```
[root@localhost ~]# ./eat.sh italiano smart
smart, Joes, 6781234, 5
[root@localhost ~]#
```

ويمكن ايضا ان نجعل اسم ال file متغير بحيث يمكن وضع اي file اخر ونشير اليه داخل ال script بالرمز \$2 كالتالي

```
[root@localhost ~]# cat eat.sh
#!/bin/bash
TYPE=$1
FILE=$2
echo "the script name is $0"
grep $TYPE $FILE
```

وعند تنفيذ ال script نستخدم ال form التالية

```
[root@localhost ~]# ./eat.sh italian restaurant.txt
the script name is ./eat.sh
italian,Bardellis,6973434,5
[root@localhost ~]#
```

POSITIONAL PARAMETERS 0 AND 1 THROUGH 9

كما ذكرنا ان ال parameters هي مجموعة من ال positional parameters التي تشير الي مكان ال argument الذي سيستخدم في ال script

واكبر عدد من ال parameters يمكن استخدامه مع ال script هو اي من 0 الى 9

ويشير ال \$0 الي اسم ال script

ويشير ال \$1 الي اول argument وهكذا

ولكن يجب ان تتذكر جيدا ان ال \$0 تشير الي اسم ال script اذا استخدم مع ال script بينما تشير الي اسم ال shell المستخدم اذا استخدمناه ك command منفصل كال التالي

```
[root@localhost ~]# echo $0
bash
[root@localhost ~]# cat eat.sh
#!/bin/bash
TYPE=$1
FILE=$2
echo "the script name is $0"
grep $TYPE $FILE
[root@localhost ~]# ./eat.sh italian restaurant.txt
the script name is ./eat.sh
italian,Bardellis,6973434,5
[root@localhost ~]#
```

يفضل تخصيص variable لل argument بمعنى انه من الافضل تخصيص \$TYPE=\$1 واستخدام ال variable نفسه في ال script وليس ال argument كما في السطر

```
grep $TYPE $FILE
```

لأننا بعد قليل سنتحدث عن كيفية عمل shift لل arguments وهذا سيؤدي إلى تغيير محتويات ال argument \$1 لن تحتوي على نفس ال positional arguments الأول

لذلك فالافضل ان نقوم بحفظ ال variable في positional argument

OTHER ARGUMENTS USED WITH POSITIONAL PARAMETERS

كما ذكرنا سابقا ان عدد ال positional argument هو 10 ولكن في الحقيقة يمكن استخدام اكثرا من ذلك

\$# HOW MANY POSITIONAL ARGUMENTS HAVE WE GOT ?

لكي نعرف كم عدد ال positional arguments التي نستخدمها في ال script يمكن ان نضع الرمز "\$#" في ال script

\$* DISPLAY ALL POSITIONAL PARAMETERS

ولكي نعرف ما هي ال positional arguments المستخدمة نستخدم الرمز "\$*" داخلا في script

```
[root@localhost ~]# cat eat.sh
#!/bin/bash
TYPE=$1
FILE=$2
echo "the script name is $0"
echo "we are using $# positional arguments in the script"
echo "the positional arguments used is $* for both \$1 and \$2"
grep $TYPE $FILE
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# ./eat.sh italian restaurant.txt
the script name is ./eat.sh
we are using 2 positional arguments in the script
the positional arguments used is italian restaurant.txt for both $1 and $2
italian,Bardellis,6973434,5
[root@localhost ~]#
```

USING THE "SHIFT" COMMAND - FOR MORE THAN 9 POSITIONAL PARAMETERS

لقد تحدثنا عن ال \$0 positional argument وقلنا ان \$0 مخصوص لل script ثم يليها \$1 الى \$9 وذكرنا ان هناك اكثر من ذلك من ال positional argument فكيف ذلك ؟

يمكن الحصول على اكتر \$9 عن طريق عمل shift لل positional arguments بأي مقدار فإذا قلنا

```
shift 5
```

فبذلك يتم اضافة \$5 : \$1 جديدة وتصبح ال \$1 >> \$6 وال \$6 >> \$14 وذلك قلنا انه من الافضل ان يتم تخصيص variable لكل positional argument

EXIT STATUS OF THE PREVIOUS COMMAND

اخر \$ سنتحدث عنها الان هي "\$?" وهي توضح ال exit status لآخر command تم تنفيذه (هل تم تنفيذه بنجاح ام ان هناك error)

```
[root@localhost ~]# cat eat.sh
#!/bin/bash
TYPE=$1
FILE=$2
grep $TYPE $FILE
echo $?
[root@localhost ~]# ./eat.sh italian restaurant.txt
italian,Bardellis,6973434,5
0
[root@localhost ~]# ./eat.sh italian fake
grep: fake: No such file or directory
2
[root@localhost ~]# ./eat.sh egypt restaurant.txt
1
[root@localhost ~]#
```

ونلاحظ هنا ان المثال الاول كان فيه ال 0 ومعنى ذلك انه تم تنفيذ ال script بشكل صحيح

ولكن في المثال الثاني كانت ال 2 لوجود error في تنفيذ ال script بشكل صحيح لعدم وجود file يسمى fake

وفي المثال الثالث كانت الـ 1 exit status موجود في تنفيذ الـ script لوجود error في تنفيذ الـ argument egypt يسمى

وكل command يتم تنفيذه يكون له exit status ولكنها by default لا تظهر علي الشاشة اذا استخدمنا الـ command : \$?

```
[root@localhost ~]# ping 199.199.199.2; echo $?
PING 199.199.199.2 (199.199.199.2) 56(84) bytes of data.
From 192.168.1.84 icmp_seq=3 Destination Host Unreachable
From 192.168.1.84 icmp_seq=6 Destination Host Unreachable
^C
--- 199.199.199.2 ping statistics ---
7 packets transmitted, 0 received, +2 errors, 100% packet loss, time 6005ms

1
[root@localhost ~]# ping 127.0.0.1; echo $?
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_req=1 ttl=64 time=0.098 ms
64 bytes from 127.0.0.1: icmp_req=2 ttl=64 time=0.091 ms
64 bytes from 127.0.0.1: icmp_req=3 ttl=64 time=0.108 ms
64 bytes from 127.0.0.1: icmp_req=4 ttl=64 time=0.085 ms
^C
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.085/0.095/0.108/0.013 ms
0
[root@localhost ~]#
```

CHAPTER 7. WHERE TO FROM HERE?

MAKING DECISIONS

كثير من الاحيان نحتاج الي تنفيذ command معين عند حدوث action ما، بمعنى اذا حدث كذا افعل كذا ، اي انا نحتاج الي جمل شرطية

```
if [ expression ]
then
OK #Do something
else
NOT OK #Do something else
fi
```

TESTING FOR A TRUE OR FALSE CONDITION

لذلك سنتحدث عن بعض الجمل الشرطية مثل if-else ، ولكن قبل ان نتحدث عنها فسنقوم بالحديث عن الامر test

THE TEST COMMAND

فال Values compare file type check علي الـ test command يقوم بعمل check بين الـ form التالية فيمكن استبدال الـ form التالية

```
if [ x -eq 9 ]
then
echo equal
else
echo no equal
fi
```

بالـ form التالية

```
test $x -eq 9
if [ $? -eq 0 ]
then
echo equal
else
```

```
echo no equal  
fi
```

ويمكن معرفة مزيد من التفاصيل عن طريق

```
info test
```

WHAT IS "TRUE" AND "FALSE"

ماذا يعني true او false وكيف يتم التعبير عنها في ال shell ??
في اي لغة برمجة يتم التعبير عن false او true بقيمة معينة، وفي ال shell يتم التعبير عن ال value بال 0 والتعبير عن ال true بأي رقم اخر مثل 1 و25 و2000
ومثلا يتم التعبير عن false كما بالصورة التالية

```
[root@localhost ~]# ping 199.199.199.1  
PING 199.199.199.1 (199.199.199.1) 56(84) bytes of data.  
^C  
--- 199.199.199.1 ping statistics ---  
29 packets transmitted, 0 received, 100% packet loss, time 28165ms  
[root@localhost ~]# echo $?  
1  
[root@localhost ~]#
```

ومعنى ظهور ال 1 : هو ان نتيجة تنفيذ ال Ping كانت خاطئة ، ويتم التعبير عن true كما يلي

```
[root@localhost ~]# who; echo $?  
root    tty1        2012-11-18 11:01 (:0)  
root    pts/0        2012-11-18 11:02 (:0.0)  
0  
[root@localhost ~]#
```

ومعنى ظهور ال 0 : هو ان نتيجة تنفيذ who صحيحة

DIFFERENT TYPES OF TESTS

يمكن تنفيذ الـ test command على عدة parameters مثل :

1. a string test
2. a numeric test
3. a file test

TESTING A STRING

يقوم الـ test command بعمل strings check على الـ strings لنقم بتنفيذ هذه الـ commands

```
NAME="ahmed"  
test $NAME = ahmed  
echo $?
```

سنجد في الصورة التالية

```
[root@localhost ~]# NAME=ahmed  
[root@localhost ~]# test $NAME = ahmed  
[root@localhost ~]# echo $?  
0  
[root@localhost ~]# test $NAME = mohamed  
[root@localhost ~]# echo $?  
1  
[root@localhost ~]#
```

ان نتيجة تنفيذ الـ form الاولى ظهرت 0 لان الـ condition متحقق ولكن العكس في الـ second

سنجد ايضا في الـ test command ان هناك مسافة بين الـ variable name : NAME والـ variable value : ahmed والـ test condition : =

اما اذا اردنا ان نتأكد من وجود variable NAME ام لا فيمكن ذلك عن طريق

```
[root@localhost ~]# test "$NAME"
[root@localhost ~]# echo $?
0
```

والاحظ ان ال " " قامت بترجمة ال \$ ومعاملتها ك regular expression ، ولكن اذا اردنا التأكد من وجود variable HI تسمى HI

```
[root@localhost ~]# test $HI
[root@localhost ~]# echo $?
1
[root@localhost ~]#
```

فنجد ان ال 1 " وهذا يعني عدم وجود variable بهذا الاسم ولكن اذا وضعنا \$NAME بين علامتين ' ' ticks

```
[root@localhost ~]# test '$NAME'
[root@localhost ~]# echo $?
0
```

فسنجد ان ال 0 على الرغم من ان ال ticks يقوم بالغاء غمل expansion لل \$

فكيف ذلك ؟

لان ال test command اذا وجد ال ticks فانه لا يقوم بعمل check على ال variables ولكنه يقوم بعمل check على الكلمة الموجودة بين ال ticks هل هي string ام لا ، وبالطبع يجد انها string فتظهر ال status = 0

لذلك يجب الحذر جيدا ووضع علامتي ال " " حول ال variable name حتى يقوم بعمل strings على ال variables وليس على ال strings

```
test "$NAME" = "hamish"
test "$NAME" = hamish
```

HAS A VARIABLE BEEN SET OR NOT ?

وبعد ان تأكدنا من وجود الـ variable سوف نتأكد من هذا الـ value ام لا اي هل
هذا الـ Variable set or unset

التأكد من ذلك يتم بطريقة بسيطة وهي ان نستخدم مثل هذه الـ form

```
test "${NAME}x" = x
```

فإذا كانت الـ variable \${NAME} لها قيمة ahmed (set) فان نتيجة x
تساوي (ahmedx) وهي بالطبع لا تساوي (x) فتتصبح النتيجة false والـ 1

اما اذا كانت الـ NAME ليس لها قيمة (unset) فان نتيجة x
تساوي (x) فقط وهي تساوي الطرف الايمن من المادلة (x) فتتصبح النتيجة true والـ 0

```
[root@localhost ~]# NAME=ahmed
[root@localhost ~]# test "${NAME}x" = x
[root@localhost ~]# echo $?
1
[root@localhost ~]# unset NAME
[root@localhost ~]# test "${NAME}x" = x
[root@localhost ~]# echo $?
0
[root@localhost ~]#
```

وهناك ايضا طريقة اخرى للتأكد من طول الـ string

String test	Meaning
-z	Zero – length
-n	Non zero – length

فإذا كان لدينا الـ variable يحتوى على value تكون من 5 مسافات واردنا ان نتأكد اذا كانت
هذه الـ variable يحتوى على non value ام لا فنقوم بعمل check على طول الـ string او طول الـ value

```
[root@localhost ~]# NAME=""
[root@localhost ~]# test -z "$NAME"
[root@localhost ~]# echo $?
1
[root@localhost ~]# test -n "$NAME"
[root@localhost ~]# echo $?
0
[root@localhost ~]#
```

نعيد المثال مرة اخرى ولكن مع جعل ال variable لا يحتوى على اي value

```
[root@localhost ~]# NAME=""
[root@localhost ~]# test -z "$NAME"
[root@localhost ~]# echo $?
0
[root@localhost ~]# test -n "$NAME"
[root@localhost ~]# echo $?
1
[root@localhost ~]#
```

ستنترق الان لل test condition مثل علامة "= " فهناك الكثير من العلامات التي يمكن استخدامها مع ال string test مثل

String test condition	Name
=	يساوي
!=	لا يساوي
<=	اقل من او يساوي
>=	اكبر من او يساوي
<	اقل من
>	اكبر من

وهذه ال conditions خاصه بال string test

NUMERIC TESTS

بعد ان تحدثنا عن ال string test نأتي الان الى الحديث عن ال numeric test وهي تستخدم في التعامل مع الارقام ، مثل :

```
x=101
test "$x" -lt 10
echo $? $
```

وهنا نلاحظ وجود الرمز "-lt" فماذا يعني ؟

الرمز "-lt" يعتبر ال numeric test condition ويعنى less than وفيما يلى انواع ال numeric conditions

String test condition	Name
-eq	يساوي
-neq	لا يساوي

-le	اقل من او يساوي
-ge	اكبر من او يساوي
-lt	اقل من
-gt	اكبر من

```
[root@localhost ~]# x=10
[root@localhost ~]# test "$x" -lt 100
[root@localhost ~]# echo $?
0
[root@localhost ~]# test "$x" -lt 5
[root@localhost ~]# echo $?
1
[root@localhost ~]#
```

اذا فعند المقارنة بين ال string test conditions 2 نستخدم ال values
 اما عند المقارنة بين ال numeric test conditions فنستخدم ال values

FILE TEST

وهو ثالث واخر نوع سنتكلم عنه ، وهو يستخدم للتأكد من ما اذا كان هذا ال file هو directory او hard link او soft link او normal file او options مع test الامر

Option	Type
-f	File
-d	Directory
-L , -h	Symbolic link
-s	The size is greater than zero

وهناك الكثير من ال options ولكن سنكتفي بهذا وسنأخذ بعض الامثلة:

```
[root@localhost ~]# touch test_file
[root@localhost ~]# test -f test_file
[root@localhost ~]# echo $?
0
[root@localhost ~]# test -d test_file
[root@localhost ~]# echo $?
1
[root@localhost ~]#
```

مثال اخر عن طريق ال directory

```
[root@localhost ~]# mkdir test_dir
[root@localhost ~]# test -d test_dir
[root@localhost ~]# echo $?
0
[root@localhost ~]# test -s test_dir
[root@localhost ~]# echo $?
0
[root@localhost ~]# test -L test_dir
[root@localhost ~]# echo $?
1
```

واخيرا نأخذ هذا المثال

```
[root@localhost ~]# test -f .
[root@localhost ~]# echo $?
1
[root@localhost ~]# test -L .
[root@localhost ~]# echo $?
1
[root@localhost ~]# test -d .
[root@localhost ~]# echo $?
0
[root@localhost ~]#
```

ال ”.“ هي ال current directory لذك فاكتت ال status = 1 في كل من ال -L , -f , -d

ولكنها تساوي 0 في

LOGICAL OPERATORS

لقد انتهينا من الحديث عن الـ `test` وكيفية استخدامه مع

```
test if "x" is true then "y"
```

ولكن ماذا اذا اردت ان استخدم الـ `test` مع اكثـر من `condition` مثل :

```
test if "x" OR "y" is true then "z"
```

في هذه الحالة سنستخدم ما يعرف بال `logical operators` مثل

- NOT ويرمز لها بالرمز "!"
- AND ويرمز لها بالرمز "-a"
- OR ويرمز لها بالرمز "-o"

وكما نعلم انه اذا استخدمنا NOT فسوف يكون

```
! 1=0
```

```
! 0=1
```

واذا استخدمنا AND ورمـزنا لـ `true` بـ حـرف "T" وـالـ `false` بـ حـرف "F"

```
0 -a 0 = 0
```

```
0 -a 1 = 1
```

```
1 -a 0 = 1
```

```
1 -a 1 = 1
```

و كذلك

```
T and T = T
```

```
T and F = F
```

```
F and T = F
```

```
F and F = F
```

واذا استخدمنا OR ورمـزنا لـ `true` بـ حـرف "T" وـالـ `false` بـ حـرف "F"

```
0 -o 0 = 0
```

```
0 -o 1 = 0  
1 -o 0 = 0  
1 -o 1 = 1
```

وكذلك

```
T or T = T  
T or F = T  
F or T = T  
F or F = F
```

```
[root@localhost ~]# test "$NAME" = ahmed -o -n "$NAME"  
[root@localhost ~]# echo $?  
0  
[root@localhost ~]# test "$NAME" = amr -o -n "$NAME"  
[root@localhost ~]# echo $?  
0  
[root@localhost ~]# test "$NAME" = ahmed -a -n "$NAME"  
[root@localhost ~]# echo $?  
0  
[root@localhost ~]# test "$NAME" = amr -a -n "$NAME"  
[root@localhost ~]# echo $?  
1  
[root@localhost ~]# █
```

وهنا فان نتيجة `"$NAME" = ahmed` هي true او "T" و "-n" تعني ان هذا ال string يعني وهذا يعني انه true او "T" ايضا فتكون النتيجة "non zero length"

```
T OR T = T
```

وفي المثال الثاني

```
F OR T = T
```

وفي المثال الثالث

```
T AND T = T
```

وفي المثال الرابع

```
F and T = F
```

ويمكن ان نتأكد من نوع ال file كما يلي :

```
test \(-f .bashrc \)
```

```
[root@localhost ~]# test \(-f .bashrc \)
[root@localhost ~]# echo $?
0
```

```
test \(! -f .bashrc \)
```

```
[root@localhost ~]# test \(! -f .bashrc \)
[root@localhost ~]# echo $?
1
```

CONDITIONS IN THE SHELL

ال conditions هي من اهم الاشياء التي يجب ان نتعلمها في ال scripting ، وهناك عدة طرق للتعامل مع ال conditions

USING THE "IF" STATEMENT

كيف نتعامل مع if statement ؟ ان ابسط اشكال ال if statement هو :

```
if condition
then
    do some operations
fi
```

وهي كما نري في المثال يجب ان تبدأ ال if statement بـ :

- 1 "if"
- 2 يليها في السطر الثاني كلمة "then"
- 3 يليها في السطر الثالث ال commands التي نريد تنفيذها (ويفضل وضع tab او مسافتين قبل ال commands)
- 4 ثم نقوم بانتهاء ال statement بـ "fi"

THE "IF" "THEN" "ELSE" STATEMENT

نقوم الان باضافة مزيد من ال features على ال if statement والتي تسمح لنا باضافة خيارات اكثـر

```
if condition
then
    <condition was TRUE, do these actions>
else
    <condition was FALSE, do these actions>
fi
```

ويمكن عمل اكثـر من test عن طريق

```
if condition
then
    <condition was TRUE, do these actions>
else
    if condition
        then
            <condition was TRUE, do these actions>
        else
            <condition was FALSE, do these actions>
    fi
fi
```

THE "ELIF" STATEMENT

كما نري في المثال الاخير هناك الكثير من if statement else والكثير من then لذلک فلکي يقوم بتبسيط شكل ال if statement يمكن وضع بعض الاختصارات مثل استبدال elif else if بـ

```
if condition
then
    <condition was TRUE, do these actions>
elif condition
then
    <condition was TRUE, do these actions>
else
    <condition was FALSE, do these actions>
fi
```

وباستخدام هذه الاختصارات فلن نحتاج الي وجود "fi" مع كل elif ويكفي "fi" واحده فقط

ولكن ماذا اذا كان لدي 10 conditions فهل سنستخدم if 10 مرات ؟

بالطبع لا فهذا سيئ ، فماذا الحل ؟

الحل هو استخدام case statement بدل من if statement

THE "CASE" STATEMENT

ال case statement ليس loop ولكنها تقوم بعمل matcheing على ال conditions بطريقة اسهل

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
```

```
* )  
statements ;;  
  
...  
esac
```

حيث يقوم ال case بعمل expand لـ expression ثم يقوم بعمل matching بينه وبين كل pattern

فإذا وجد ال matched pattern فيقوم بتنفيذ ال statement حتى يصل الي ;;
وان لم يجد اي matche فيقوم بعمل مع ال "*" ثم ينفذ ال status ثم يقوم بانهاء عمل ال case statement واخراج ال script وهذه الطريقة تجعل التعامل مع ال script اسهل

DEBUGGING YOUR SCRIPTS

ستتحدث الان عن كيفية عمل debug لـ script ، وفي الحقيقة هناك عدة طريق لعمل debugging مثل :

1- اضافة x- الي اول سطر في ال script

```
#!/bin/bash -x
```

```
[root@localhost ~]# cat eatout.sh
#!/bin/bash -x
DATE=$(date +"%d %b %Y %H:%M")
TYPE=$1
echo "Queried on $DATE"
grep $TYPE restaurant.txt |sort -t, -n -k 3
exit 0

[root@localhost ~]# ./eatout.sh italian
++ date '+%d %b %Y %H:%M'
+ DATE='21 Nov 2012 06:05'
+ TYPE=italian
+ echo 'Queried on 21 Nov 2012 06:05'
Queried on 21 Nov 2012 06:05
+ sort -t, -n -k 3
+ grep italian restaurant.txt
italian,Bardellis,6973434,5
+ exit 0
```

2- تشغيل ال script بهذه الطريقة

```
bash -x eatout.sh
```

```
[root@localhost ~]# bash -x eatout.sh italian
++ date '+%d %b %Y %H:%M'
+ DATE='21 Nov 2012 06:01'
+ TYPE=italian
+ echo 'Queried on 21 Nov 2012 06:01'
Queried on 21 Nov 2012 06:01
+ sort -t, -n -k 3
+ grep italian restaurant.txt
italian,Bardellis,6973434,5
+ exit 0
```

وهاتين الطريقتين تستخدمان في عرض طريقة تنفيذ ال script سطر سطر كما نرى في الصورة
 الطريقة الثالثة وهي تنفيذ ال script جزء جزء مع عدم الانتقال من جزء الى اخر الا بعد التأكد من
 نتيجته ، ويتم ذلك عن طريق اضافة بعض ال lines الى ال script مثل

```
echo #?
```

```
echo "Hit any key to continue"...
read
```

```
[root@localhost ~]# cat eatout.sh
#!/bin/bash
DATE=$(date +"%d %b %Y %H:%M")
TYPE=$1
echo "Queried on $DATE"
echo $?
echo "press any key to continue"
read
grep $TYPE restaurant.txt |sort -t, -n -k 3
exit 0

[root@localhost ~]# ./eatout.sh italian
Queried on 21 Nov 2012 06:12
0
press any key to continue

italian,Bardellis,6973434,5
[root@localhost ~]#
```

THE NULL COMMAND

ال null command هو اضافة ال ":" الي ال script بهذا الشكل

```
if :
then
    echo "...."
fi
```

```
[root@localhost ~]# cat loop.sh
#!/bin/bash
if :
then
echo "true"
fi
[root@localhost ~]# ./loop.sh
true
[root@localhost ~]#
```

ودائما هذه الـ true form لأن الـ ":" دائمًا

THE || AND && COMMANDS

إذا كنا نبحث عن طريقة مختصرة لتنفيذ الـ if statement فنستطيع فعل ذلك باستخدام || و && الذي يسمح بتنفيذ العديد من الـ command في سطر واحد باستخدام ، **كيف هذا ؟**

لنقم بتنفيذ الـ commands 2 كما يلي ونلاحظ الفرق :

Command 1

```
grep italian restaurants.txt || echo "sorry no italians here"
```

```
[root@localhost ~]# grep italian restaurant.txt || echo "sorry no italians here"
italian,Bardellis,6973434,5
[root@localhost ~]#
```

Command 2

```
grep egypt restaurants.txt || echo "sorry no italians here"
```

```
[root@localhost ~]# grep egypt restaurant.txt || echo "sorry no egypt here"
sorry no egypt here
[root@localhost ~]#
```

لاحظ ان في المثال الاول كانت النتيجة هي ال record italiano بينما في المثال الثاني فلم يكن هناك record Egypt لذلک ظهرت رسالة "sorry no italians here"

فماذا يعني هذا ؟

ان هذا يساوي

```
if there are italians inside restaurants.txt  
then  
    return them  
[else]  
    return the string "sorry no italians here"
```

وبالنسبة لل shell فهذا يعني

```
if the result of the grep command is TRUE (0)  
then  
    you will get the lines containing the word italian from  
    the file restaurants.txt  
  
if the result of the grep command is FALSE (1)  
then  
    print 'sorry no italians here'
```

اذا فالعلامة || تعني OR (اذا تحقق ال command الاول ، فلا تنفذ ال command الثاني)

واذا لم يتحقق ال command الاول نفذ ال command الثاني)

```
echo "Oh, you're looking for italian, here they are : " && grep  
italian restaurants.txt
```

```
[root@localhost ~]# echo "oh, you are looking for italian, here they are:" && grep italian restaurant.txt
oh, you are looking for italian, here they are:
italian,Bardellis,6973434,5
[root@localhost ~]#
```

اذا فالعلامة **&&** تعني **and** (يجب ان يتحقق ال command الاول حتى ينتقل الي ال command الثاني)

مثال:

```
[root@localhost ~]# grep egypt restaurant.txt && echo "oh, you are looking for italian, here they are:"
[root@localhost ~]# echo $?
1
[root@localhost ~]#
```

وكما نري فان ال command الاول خطأ فلم ينفذ ال command الثاني

مثال اخر:

```
[root@localhost ~]# echo "oh, you are looking for italian, here they are:" && grep italian restaurant.txt
oh, you are looking for italian, here they are:
italian,Bardellis,6973434,5
[root@localhost ~]# echo $?
0
[root@localhost ~]#
```

وكما نري ان ال command الاول تم تنفيذه بشكل صحيح فقام بتنفيذ ال command الثاني

مثال اخر:

```
[root@localhost ~]# echo "oh, you are looking for italian, here they are:" && grep egypt restaurant.txt
oh, you are looking for italian, here they are:
[root@localhost ~]# echo $?
1
[root@localhost ~]#
```

هنا قام بتنفيذ ال command الاول بشكل صحيح ولكن ال command الثاني خطأ ، فعرض نتيجة ال command الاول فقط

CHAPTER 8. LOOPS

INTRODUCTION

هناك 3 انواع من ال loops يمكن استخدامهم مع ال shell

- 1- for loops
- 2- while loops
- 3- until loops

وجميعهم لهم نفس ال syntax كما سنرى

THE "FOR" LOOP

ال syntax لهذا ال loop

```
for variable in list
do
...
done
```

```
[root@localhost ~]# cat forloop.sh
#!/bin/bash
for i in 1 2 3 4
do
    echo ${i}
done
[root@localhost ~]# ./forloop.sh
1
2
3
4
[root@localhost ~]#
```

وكان ال for loop تقول

```
for every element in the list (1,2,3,4)
```

```
do something (echo $i in output)
```

وهذه ال List يمكن ان تكون اي شيئ (ارقام - اسماء - او variables او commands او محتوي file معين او اي شيئ اخر)

مثال على الاسماء في ال for loop

```
[root@localhost ~]# cat forname.sh
#!/bin/bash

for NAME in Ahmed Mohamed Amr Ali
do
echo "people involved in this project: $NAME"
done
[root@localhost ~]# ./forname.sh
people involved in this project: Ahmed
people involved in this project: Mohamed
people involved in this project: Amr
people involved in this project: Ali
[root@localhost ~]# ■
```

مثال اخر على استخدام ال commands في ال for loop

```
[root@localhost ~]# cat forfile.sh
#!/bin/bash
for FILES in `ls -1`
do
echo "file: `file $FILES`"
done
[root@localhost ~]# ./forfile.sh
file: 3character: ASCII text
file: anaconda-ks.cfg: ASCII English text
file: capital: ASCII text
file: capital-result: ASCII text
file: correct_script: ASCII text
file: date_script.sh: a /sbin/bash script, ASCII text executable
file: Desktop: directory
file: Documents: directory
file: Downloads: directory
```

لاحظ اننا وضعنا ال commands بين `` ويمكن وضعه داخل ()

والامر file يقوم بعرض نوع ال file

وان ال command هو ls -1 (one) وليس ls -a وهو يقوم بعرض كل file في سطر واحد
الفرق في الصورة التالية

```
[root@localhost home]# ls
ahmed amr lost+found
[root@localhost home]# ls -1
ahmed
amr
lost+found
[root@localhost home]# ls -l
total 24
drwx----- 20 ahmed ahmed 4096 Nov 17 13:38 ahmed
drwx----- 4 amr amr 4096 Nov 11 21:40 amr
drwx----- 2 root root 16384 Aug 7 02:54 lost+found
[root@localhost home]#
```

مثال اخر على استخدام محتويات ال files في ال for loop

```
[root@localhost ~]# cat names.txt
ahmed
Amr
Mohamed
[root@localhost ~]# cat forname.sh
#!/bin/bash
for NAME in `cat $1`
do
echo "people involved in this project: $NAME"
done
[root@localhost ~]# ./forname.sh names.txt
people involved in this project: ahmed
people involved in this project: Amr
people involved in this project: Mohamed
[root@localhost ~]#
```

مثال اخر على استخدام ال for loop مع الاعداد

```
[root@localhost ~]# cat seq.sh
#!/bin/bash
for count in `seq 20`
do
echo "$count"
done
[root@localhost ~]# ./seq.sh
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
[root@localhost ~]#
```

حيث يقوم الامر seq بعمل print numeric sequence
مثال اخر بدون استخدام كلمة "in" الموجودة داخل ال script

```
[root@localhost ~]# cat count.sh
#!/bin/bash
for ((i=1; i<=10; i=i+1))
do
echo "$i"
done
[root@localhost ~]# ./count.sh
1
2
3
4
5
6
7
8
9
10
[root@localhost ~]#
```

ويمكن استبدال “`i--`” بـ “`i=i-1`” واستبدال “`i++`” بـ “`i=i+1`”

```
[root@localhost ~]# cat count.sh
#!/bin/bash
for ((i=10; i>=0; i--))
do
echo "$i"
done

[root@localhost ~]# ./count.sh
10
9
8
7
6
5
4
3
2
1
0
[root@localhost ~]#
```

نستطيع الاستغناء عن الكلمة "in" الموجودة داخل ال script ولكن سنضع هذه ال values في الخارج بجانب ال script كما يلي

```
[root@localhost ~]# cat for.sh
#!/bin/bash
for arg
do
echo $arg
done
[root@localhost ~]# ./for.sh one two three four
one
two
three
four
[root@localhost ~]#
```

there is no "in" inside the script

we put the values outside the script

وهذه طريقة اخرى في استخدام ال loop for وهي وضع variable count قيمته 1 يقوم بحساب عدد العمليات التي قامت بها ال loop

```
[root@localhost ~]# cat for.sh
#!/bin/bash
count=1
for arg
do
echo "Argument $count is $arg"
$((count=count+1))
done
```

لاحظ اننا استخدمنا ال form

```
$ ( (count=count+1) )
```

وهذه ال form لا تفهمها ال shell لذلك ستظهر النتيجة كما يلي

```
[root@localhost ~]# ./for.sh one two three four
Argument 1 is one
./for.sh: line 6: 2: command not found
Argument 2 is two
./for.sh: line 6: 3: command not found
Argument 3 is three
./for.sh: line 6: 4: command not found
Argument 4 is four
./for.sh: line 6: 5: command not found
[root@localhost ~]
```

وإذا نظرنا إلى error الظاهر نجد انه ينبع بسبب ال line 6 وهو الخاص بزيادة قيمة ال count حيث قام ال shell بتنفيذ هذه ال form وزيادة قيمة ال count ولم يتوقف عند ذلك بل قام بتنفيذها لأنها command لذلک فلم يجد اسمه 2 او 3 او 4 او 5

اذا فما الحل ؟

هناك عدة طرق لحل هذه المشكلة

الطريقة الاولى هي وضع (":noop") امام هذه ال form لتكون كالتالي

```
: $((count=count+1))
```

```
[root@localhost ~]# cat for.sh
#!/bin/bash
count=1
for arg
do
echo "Argument $count is $arg"
: $((count = count +1))
done
[root@localhost ~]# ./for.sh one two three four
Argument 1 is one
Argument 2 is two
Argument 3 is three
Argument 4 is four
```

الطريقة الثانية وهي استبدال هذه ال form ب آخر مثل

```
count=$((count+1))
```

```
[root@localhost ~]# cat for.sh
#!/bin/bash
count=1
for arg
do
echo "Argument $count is $arg"
count=$((count +1))
done
[root@localhost ~]# ./for.sh one two three four
Argument 1 is one
Argument 2 is two
Argument 3 is three
Argument 4 is four
```

الطريقة الثالثة هي استبدال هذه الـ form بـ form اخرى

```
count=`expr $count + 1`
```

```
[root@localhost ~]# cat for.sh
#!/bin/bash
count=1
for arg
do
echo "Argument $count is $arg"
count=`expr $count + 1`
done
[root@localhost ~]# ./for.sh one two three four
Argument 1 is one
Argument 2 is two
Argument 3 is three
Argument 4 is four
```

WHILE AND UNTIL LOOPS

ستتحدث الان عن نوعين اخرين من الـ loops وهما الـ while loop والـ until loop ولها نفس
الsyntax

الـ syntax لها while loop

```
while <condition is true>
do
    Action
done
```

وال syntax لها until loop

```
until <condition is true>
do
    Action
done
```

اذا فما الفرق بينهم

ال while loop تقوم بتنفيذ ال true conditions التي لها status 0 فقط
بينما ال until loop ستقوم بتنفيذ ال false conditions حتى تصل الي ال true condition

مثال

```
i=5
while test "$i" -le 10
do
    echo $i
done
```

```
[root@localhost ~]# cat while.sh
#!/bin/bash
i=5
while test "$i" -le 10
do
echo "$i"
i=$((i+1))
done

[root@localhost ~]# ./while.sh
5
6
7
8
9
10
[root@localhost ~]#
```

ويمكن استبدال

```
while test "$i" -le 10
```

بـ

```
while [ "$i" -le 10 ]
```

```
[root@localhost ~]# cat while.sh
#!/bin/bash
i=5
while [ "$i" -le 10 ]
do
echo "$i"
i=$((i+1))
done

[root@localhost ~]# ./while.sh
5
6
7
8
9
10
```

مثال اخر

```
while [ ! -d `ls` ]
do
    echo "file"
done
```

وهذا ال script يقوم بتنفيذ ال ls : command ويتأكد من ان هذه ال files ليست directories ثم يخرج كلمة file

مثال اخر

اذا اردنا ان نعرف ما اذا كان user معين قد قام بالدخول علي ال system ام لا، فهناك عدة طرق لعمل ذلك

اولا : نقوم بعمل script يقوم بتنفيذ who command علي هذا ال User

```
user=$1
while `who | grep "$user" > /dev/null`
do
echo "User logged in"
done
```

او باستخدام Until الذي يقوم بعرض سالة "User not logged in" حتى يدخل ال user الي ال system فيتوقف ال script عن العمل

```
user=$1
until `who | grep "$user" > /dev/null`
do
echo "User not logged in"
done
```

وهنا لا يهمنا ال command output لذلك وضعنا /dev/null

ولكن هذا ال script سيقوم بعرض كم هائل من الرسائل ، لذلك يمكن ان نجعل ال script ينتظر لمدة 10 ثواني قبل ان يعيذ ال check مرة اخري وذلك عن طريق اضافة sleep كما يلي

```

#!/bin/bash

user=$1

until who |grep "$user">> /dev/null
do
    echo "User not logged in"
    sleep 10
done
echo "Finally!! $user has entered the OS"
exit 0

```

THE BREAK AND CONTINUE COMMANDS

قد نحتاج احيانا الي ان نوقف ال script عن العمل اثناء تنفيذه ونكسر ال loop
فمثلا اذا كان لدينا script كبير ويحتوي على loops متداخلة مثل :

```

for
do
    for
        do
            if
                then
                    break;
                fi
            done
        done
done

```

للخروج من ال loop الداخلية يجب ان نقوم بكسرها عن طريق break
وهذا مثال يوضح كيفية استخدام ال break في عمل جدول في صفحة انترنت يقوم بعد رقم ال column ورقم ال row وكتابتهم في الخانة الخاصة بها في الجدول

```
#!/bin/bash

# Start by warning the browser that a table is starting
echo "<TABLE BORDER='1'>"

#Start the ROWs of the table (4 rows)
for row in `seq 4`
do
    #Start the row for this iteration
    echo "<TR>"
    #Within each row, we need 3 columns (or table-data)
    for col in `seq 3`
    do
        #If this row 2, then break out of this inner (column) loop,
        #returning to
        if [ $row -eq 2 ]
        then
            break;
        fi
        #If this is NOT row 2, then put the cell in here.
        echo " <TD>$row,$col</TD>"
    done
    #End this ROW
    echo "</TR>"
    done
    #End this table.
    echo "</TABLE>"
exit 0
```

ويمكن كسر inner loops 2 عن طريق اضافة الرقم 2 الى break

```
break 2
```

ستتحدث الان عن ال command : break الذي يعتبر عكس ال command : continue يقوم بالخروج من ال loop بينما ال command : break يقوم بالعودة الي بداية ال loop مرة اخرى ويتجاهل اي command يأتي بعده

وآخر ما سنتحدث عنه في هذا الجزء هو اذا اردنا التعامل مع ال script output بأي شكل مثل ارسال ال output الى file او تحويل ال output من small letters الى capital letters فان ذلك يتم في اخر ال script بعد كلمة done

فاما فرضنا اننا نريد حفظ ال loop output في file معين فسنسخدم علامة ال redirection في نهاية ال script كما يلي

```
for ((i=0;i<10;i++))  
do  
echo "Number is now $i"  
done > forloop.txt
```

```
[root@localhost ~]# cat loop.sh  
#!/bin/bash  
for ((i=0;i<10;i++))  
do  
echo "Number is now $i"  
done > forloop.txt  
  
[root@localhost ~]# ./loop.sh  
[root@localhost ~]# cat forloop.txt  
Number is now 0  
Number is now 1  
Number is now 2  
Number is now 3  
Number is now 4  
Number is now 5  
Number is now 6  
Number is now 7  
Number is now 8  
Number is now 9  
[root@localhost ~]# █
```

وكذلك اذا اردنا نحول الـ small letters الى capital letters

```
for ((i=0;i<10;i++))  
do  
echo "Number is now $i"  
done |tr '[a-z]' '[A-Z]' > forloop.txt
```

GETOPTS USING ARGUMENTS AND PARAMETERS

نستطيع ان ننفذ الـ script بعدة اشكال مثل :

```
./eatout.sh -a -i -r <parameter>  
./eatout.sh -ai -r <parameter>  
./eatout.sh -air <parameter>  
./eatout.sh -r <parameter> -ai
```

والـ getopts هي اختصار getting options وتعني getting options والتأكد من ان الـ shell يستطيع ان يتعامل بهذه parameters وبناء عليه false او true يخرج

اذا كيف نتعامل مع الـ getopts

CHAPTER 9. USER INPUT TO A SCRIPT

INTRODUCTION

نستطيع الان عمل script يعمل من ال CLI والان سنتعلم كيف نقوم بعمل

THE READ COMMAND

يعمل ال shell command مع جميع انواع ال shells ، وهو read command ولمعرفة ما اذا كان ال command هو built-in ام نقوم بعمل الاتي

type command

```
[root@localhost ~]# type read
read is a shell builtin
[root@localhost ~]# type type
type is a shell builtin
[root@localhost ~]# type ls
ls is hashed (/bin/ls)
[root@localhost ~]# █
```

نعود الي ال command : read فاذا كتبنا

read X Y

وضغطنا enter فسينزل الي السطر التالي وينتظر كتابة قيمة كل من Y,X

```
[root@localhost ~]# read X Y
12 24
[root@localhost ~]# echo $X
12
[root@localhost ~]# echo $Y
24
[root@localhost ~]# █
```

وإذا طبقنا هذا ال command بداخل ال script فسيكون كالتالي

```
[root@localhost ~]# cat eat.SH
#!/bin/bash
read TYPE
echo "you are asking for $TYPE food, here is"
FILE=$1
grep $TYPE $FILE
echo $?
[root@localhost ~]# ./eat.SH restaurant.txt
italian
you are asking for italian food, here is
italian,Bardellis,6973434,5
0
[root@localhost ~]#
```

يمكن اضافة -p : لعرض رسالة لل user لكي تخبره عن نوع ال Input الذي سيقوم بادخاله مثل :

```
[root@localhost ~]# cat eat.SH
#!/bin/bash
read -p "Enter the type of food you want " TYPE
echo "you are asking for $TYPE food, here is"
FILE=$1
grep $TYPE $FILE
echo $?
[root@localhost ~]# ./eat.SH restaurant.txt
Enter the type of food you want|
```

وهنا كما نرى فان ال curser ينتظر ال Input الذي سنكتبه

```
[root@localhost ~]# ./eat.SH restaurant.txt
Enter the type of food you want italian
you are asking for italian food, here is
italian,Bardellis,6973434,5
0
[root@localhost ~]#
```

وهناك وظيفة اخرى هامة جداً لل read command وهي قراءة محتويات ال files كما انه يفهم ال delimiter (يفضل ان يكون ال delimiter هو علامة "، وليس مسافة)

وهنا نجد ان ال restaurant.txt file يحتوي على :

نوع المطعم type ، مكانه place ، رقم التليفون tel ، درجة المطعم rating بالترتيب

```
[root@localhost ~]# cat restaurant.txt
smart,Parks,6834948,9
italian,Bardellis,6973434,5
steakhouse,Nelsons Eye,6361017,8
steakhouse,Butchers Grill,6741326,7
smart,Joes,6781234,5
```

ونريد استخدام ال read command في تغيير هذا الترتيب ليكون نوع المطعم type ، درجة المطعم rating ، مكانه place ، رقم التليفون tel

```
smart,9,Parks,6834948
italian,5,Bardellis,6973434
steakhouse,8,Nelsons Eye,6361017
steakhouse,7,Butchers Grill,6741326
smart,5,Joes,6781234
```

لذلك نستخدم هذا ال script الي يحتوي على ال read command

```
[root@localhost ~]# cat read.sh
#!/bin/bash
IFS=","
while read TYPE PLACE TEL RATING
do
echo "$TYPE,$RATING,$PLACE,$TEL"
done < restaurant.txt
```

ونلاحظ ان ال IFS هي input field separator وهي توضح نوع ال separator الذي سيفصل بين كل field واخر

و By default فان ال IFS=' \t\n' وتعني new line او tab او space ولكننا في هذا المثال قمنا بتغيير ال IFS

ونلاحظ ايضاً اننا وضعنا ال file في اخر ال script ك STDIN

PRESENTING THE OUTPUT

THE ECHO COMMAND

هناك نوعين مختلفين من الـ echo command

built in shell -1

```
[root@localhost ~]# type echo  
echo is a shell builtin  
[root@localhost ~]#
```

external program /bin/echo -2

```
[root@localhost ~]# type /bin/echo  
/bin/echo is /bin/echo  
[root@localhost ~]#
```

وإذا استعنا بال help الخاص بال echo program :

```
[root@localhost ~]# /bin/echo --help  
Usage: /bin/echo [SHORT-OPTION]... [STRING]...  
or: /bin/echo LONG-OPTION  
Echo the STRING(s) to standard output.  
  
-n                  do not output the trailing newline
```

هناك بعض الـ special characters التي يمكن اضافتها مع الـ echo command والتي تضيع مزيد من الـ features على الـ command مثل :

\n	النّزول الى new line
\t	اضافة Tab
\c	يطبع كل شيء في سطر واحد
\r	ويمسح كل ما بعدها
\b	لمسح كل ما قبلها
	لمسح حرف من الخلف Back space

ولكن الـ echo يقوم بكتابية كل ما بين الاقواس كما هو ولا يقوم بترجمة الـ special characters ، اذا فكيف سيقوم بترجمة "\\" ولا يقوم بكتابتها كما هي ؟

الحل هو ان هناك بعض ال options التي تستخدم مع ال echo مثل

-e
-n

يسمح بترجمة ال special characters

امثلة :

في المثال الاول هنا نري ان ال "\n" ظهرت كما هي ولكن في المثال الثاني بعد اضافة "-e" قامت ال shell بترجمة ال "\n"

```
[root@localhost ~]# echo "ahmed \ngamil"  
ahmed \ngamil  
[root@localhost ~]# echo -e "ahmed \ngamil"  
ahmed  
gamil
```

وهذا مثال ايضا علي استخدام ال "\c" في اظهار النتيجة في نفس السطر وال "\t" لاضافة tab

```
[root@localhost ~]# cat echo.sh  
#!/bin/bash  
while read rating type place tel  
do  
/bin/echo -e "$type \t $rating \t $place \t $tel \c"  
done < restaurant.txt  
[root@localhost ~]# ./echo.sh  
smart,Parks,6834948,9  
Eye,6361017,8 steakhouse,Nelsons  
italian,Bardellis,6973434,5  
Grill,6741326,7 steakhouse,Butc
```

THE PRINTF COMMAND

بعد ان تحدثنا عن ال echo وقمنا بعرض بعض الامثلة ، لاحظنا ان ال output دائمًا ما يكون unformatted ، **فهل هناك طريقة يمكن ان يجعل بها ال output يظهر بشكل افضل؟**

نعم ، فيمكن استخدام ال printf لعمل formatting لل output ، وال printf يخضع لهذه ال form

```
printf(%[flags][width][.precision]type)
```

وابسط اشكال ال printf هي

```
printf(%type)
```

وال type هو ال characters التي سيتم طباعتها:

s	Srting
d	Decimal
o	Octal
x	Hexadecimal
u	Unsigned integers

مثال : سنقوم بطباعة كلمة ahmed باستخدام ال printf وسنوضح لـ command ان ما سيقوم بطباعته هو string

```
printf '%s' ahmed
```

```
[root@localhost ~]# printf '%s' ahmed
ahmed[root@localhost ~]# █
```

ونلاحظ ان ال prompt تم عرضها بجوار كلمة ahmed وليس في سطر مستقل لأننا لم نخبر ال command اتنا نريده ان يعرض new line بعد تنفيذ ال command ، حيث يجب ان نخبر ال printf بما نريده حرفياً لذلك اذا اردنا ان نعرض سطر جديد بعد تنفيذ ال command او اضافة tab او اي options مثلما فعلنا في حالة ال echo فيجب اضافة

\n	النزول الى new line
\t	اضافة Tab
\c	يطبع كل شيء في سطر واحد ويمسح كل ما بعدها
\r	لمسح كل ما قبلها
\b	لمسح حرف من الخلف Back space

```
[root@localhost ~]# printf '%s\n' ahmed
ahmed
[root@localhost ~]# printf '%s\t' ahmed
ahmed [root@localhost ~]# printf '%s\b' ahmed
ahme[root@localhost ~]# printf '%s\r' ahmed
[root@localhost ~]# █
```

مثال اخر:

باضافة الـ options precision وهي احد الـ options التي يمكن اضافتها لتحديد دقة العرض:

```
printf '%.5d' 12
```

```
[root@localhost ~]# printf '%.5d\n' 12
00012
[root@localhost ~]# █
```

وهنا طلبنا من printf طباعة الرقم 12 ولكن باستخدام digits 5 لاحظ الفرق بينه وبين الصورة التالية

```
[root@localhost ~]# printf '%.3d\n' 12
012
[root@localhost ~]# █
```

مثال اخر:

باضافة الـ flags مثل "+" او "-"
وال "+" تطلب من ال shell عرض ال digits بالاشارة الخاصة بها سواء كانت "+" او "-"

```
ahme[root@localhost ~]# printf '%s\r' ahmed
[root@localhost ~]# printf '+%.5d %.5d %.3d\n' 9 12 -16
+00009 +00012 -016
[root@localhost ~]# █
```

مثال اخر:

لاحظ الصورة التالية والفرق بين الامثلة الثلاثة :

```
[root@localhost ~]# printf '%s\n' flying
flying
[root@localhost ~]# printf '%10s\n' flying
      flying
[root@localhost ~]# printf '%-10s\n' flying
flying
[root@localhost ~]# printf '%-3s\n' flying
flying
[root@localhost ~]# █
```

في المثال الاول قمنا بعرض ال type string: flying باستخدام ال flying options

وفي المثال الثاني قمنا باضافة size option الذي يقوم بتخصيص 10 خانات لعرض الكلمة flying ، فقام بعرض flying في اخر ال 10 خانات

وفي المثال الثالث قمنا باضافة ال flag وهي "-" لعرض flying في بداية ال 10 خانات

وفي المثال الرابع قمنا بتصغير ال size المخصص ل flying فقام بعرض flying كاملا

CHAPTER 10. ADDITIONAL INFORMATION

THE SHELL ENVIRONMENTAL VARIABLES PERTAINING TO SCRIPTING

سنلقي نظرة على ال shell وال environment variables المتعلقة بها وقد تكلمنا سابقاً عن الفرق بين ال shell variable وال environment variable وتحدثنا عن ال subshell وكيفية الدخول والخروج من ال subshell ولمعرفة ال environment variables نستخدم الامر

Env

```
[root@localhost ~]# env
XDG_VTNR=1
XDG_SESSION_ID=2
HOSTNAME=localhost.localdomain
IMSETTINGS_INTEGRATE_DESKTOP=yes
GPG_AGENT_INFO=/tmp/keyring-tVo9Gm/gpg:0:1
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
```

وهذه بعض اهم ال environmental variables لها عند ال startup يتم عمل set لها عند ال

```
[root@localhost ~]# echo $HOME
/root
[root@localhost ~]# echo $USERNAME
root
[root@localhost ~]# echo $LOGNAME
root
[root@localhost ~]# echo $MAIL
/var/spool/mail/root
[root@localhost ~]# echo $HOSTNAME
localhost.localdomain
[root@localhost ~]# █
```

ولمعرفة الـ shell variables نستخدم الامر

```
set
```

```
[root@localhost ~]# set | head
ALL_OPTS='-l --list -S -o -n --noheading -h --help'
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force
ractive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION=/etc/bash_completion
```

ومن اهم الـ shell variables هي

```
[root@localhost ~]# echo $PS1
[\u@\h \W]\$
[root@localhost ~]# echo $PS2
>
[root@localhost ~]#
```

والـ prompt التالي يسمى bash prompt

```
[root@localhost ~]#
```

وطريقة كتابة هذا الـ prompt تعتمد على الـ form التالية

```
[root@localhost ~]# echo $PS1
[\u@\h \W]\$
```

ولذينا ايضا prompt اخر يظهر عند كتابة command ناقص وهو ">"

وهذا الـ prompt يسمى PS2

```
[root@localhost ~]# echo "hi  
> all"  
hi  
all  
[root@localhost ~]#
```

وهذه الـ shell variables هي PS1, PS2 وغيرها
وهناك special environment variables مثل "-~" و "~"
حيث ترمز "~" الى الـ home directory
وال "-~" ترمز الى الـ previous directory

```
[root@localhost ~]# cd /etc/  
[root@localhost etc]#  
[root@localhost etc]# cd ~  
[root@localhost ~]#  
[root@localhost ~]# cd --  
[root@localhost etc]#
```

THE SOURCE COMMAND

الـ source command هي طريقة لتشغيل الـ scripts وتنفيذ الـ commands من الـ new sub shell بدون فتح current shell
ويتم ذلك عن طريق وضع ". " قبل الـ script او الـ command ولاحظ ان هناك مسافة بعد الـ "

اذا دعونا نتذكرة كيف يتم تشغيل الـ script ؟

يتم تشغيل الـ script بالطريقة التالية :

```
./shellscript
```

فيقوم الـ shell بفتح new subshell وينفذ الـ script فيه لأن الـ script غالباً ما يبدأ بـ "#!/bin/bash"

وفي معظم الأحوال ينتهي الـ script بـ 0 فيقوم الـ shell بانهاء عمل الـ subshell والعودة إلى الـ parent shell

اذا فما الذي يحدث اذا استخدمنا الـ source command ؟

```
./shellscript
```

يقوم ال current shell بتنفيذ ال script

فإذا كان ال script ينتهي بـ 0 فإنه سينهي عمل ال current shell ، أما إذا لم يكن هناك 0 فإن ال current shell ستظل مفتوحة

وهذا يحدث لأن ال command sourcing يقوم بالغاء عمل ال shebang ، لذلك فإن ال script الموجودة في ال current shell تعمل من على ال login shell وبالتالي في حالة وجود 0 فإننا نخرج من ال current shell ونذهب إلى ال

اذا فما هي وظيفة ال sourcing ؟

كل user على ال system له variables خاصة به يتم حفظها في files يحتوي ال system على مجموعة من ال startup scripts التي تعمل عند بداية تشغيل ال system

عندما يقوم ال User ahmed بعمل login على ال system تقوم هذه ال scripts بعمل call لـ files التي تحتوي على ال variables عن طريق sourcing كما يلي

هناك User اسمه ahmed@gmail الخاصة به توجد في file var باسمه

```
[root@localhost ~]# cat var
NAME=ahmed
SURNAME=gamil
COMPANY="TE Data"
TELNUM='0233320700'
[root@localhost ~]#
```

ونفترض أن هناك startup script vars.sh يسمى vars.sh يعمل عند بداية تشغيل ال system يحتوي على :

```
[root@localhost ~]# cat vars.sh
#!/bin/bash
. var #sourcing the variables in the file var.
echo "$NAME $SURNAME was here"
exit 0
[root@localhost ~]#
```

ونلاحظ هنا ان ال "var" هي ال sourcing الذي ينادي على ال variable file :

وعند تشغيل هذا ال script يحدث الاتي :

```
[root@localhost ~]# ./vars.sh
ahmed gmail was here
[root@localhost ~]#
```

ويمكن وضع script داخل script اخر واستخدام ال source معه

THE EXEC COMMAND

ال exec هو command يقوم بعمل replace لـ parent process بـ another process

فمثلا اذا قمنا بتنفيذ هذا ال command

```
exec ls -l
```

نلاحظ ان هذا ال command قام باغلاق ال terminal ، لماذا ؟

لان ال terminal في البداية لم يكن يحتوي على اي process تعمل ، فقمنا بتنفيذ ال exec command الذي قام بتنفيذ ال -ls : وبعد ذلك يعود الي login prompt ولكن فعليا لا يوجد اي parent process لذلك فان ال shell تغلق ونعود الي ال

OTHER METHODS OF EXECUTING A SCRIPT OR A SERIES OF COMMANDS

ال shell مبني علي نظامين :

-1 round brackets ()
-2 curly brackets {}

EXECUTION WITH ROUND BRACKETS

ال () round brackets يعني ان يقوم ال shell بتنفيذ ال command الموجودة بين القوسين في subshell جديد
فإذا نفذنا الأمر التالي

```
Pwd
```

فإن النتيجة ستكون

```
[root@localhost ~]# pwd  
/root
```

بينما اذا قمنا بتغيير ال path وتنعيم pwd ولكن بين القوسين فسيقوم بعرض ال path الجديد على الرغم من ان ال prompt لم يتغير

```
[root@localhost ~]# (cd /bin/; pwd)  
/bin  
[root@localhost ~]# █
```

وهذا يعني ان ال shell لم يتغير ولكن تم تنفيذ ال commands الموجودة بين الاقواس في current shell الجديدة ثم يعود مرة اخرى الى ال subshell

EXECUTION WITH CURLY BRACKETS

ال {} curly brackets يعني ان يقوم ال shell بتنفيذ ال command الموجودة بين القوسين في نفس ال shell وليس في subshell جديد

مثال:

اذا كان لدينا بعض ال commands تقوم بارسال ال error messages الي /tmp/error فهناك طريقتين لعمل ذلك :

الطريقة الاولى :

```
Command1 2>/tmp/errors  
command2 2>/tmp/errors  
command3 2>/tmp/errors
```

الطريقة الثانية :

```
{ command1; command2; command3; } 2>/tmp/errors
```

ويجب ان يتبع كل commssnd واخر علامة ";"
فهذا ال form سينتج عنها error لعدم وجود ";" بعد pwd

```
{ls -l; pwd}
```

وال form الصحيحة هي

```
{ls -l; pwd;}
```

```
[root@localhost srv]# { ls -l;pwd }  
> _____  
> ^C  
[root@localhost srv]# { ls -l;pwd; }  
total 0  
-rw-r--r--. 1 root root 0 Nov 27 19:06 test  
-rw-r--r--. 1 root root 0 Nov 27 19:08 test1  
-rw-r--r--. 1 root root 0 Nov 27 19:07 test2  
/srv  
[root@localhost srv]#
```

CHAPTER 11. POSITIONAL PARAMETERS & VARIABLES RE-VISITED

INTRODUCTION

تكلمنا سابقا عن بعض الـ parameters مثل \$0, \$#, \$? وظيفة كل واحد فيهم وستتحدث الان عن بعض الطرق التي يمكن ان نتعامل بها مع الـ parameters ولكن بشكل مختلف

لنفترض مثلا ان لدينا parameter : NAME : ahmed يحتوي على value : ahmed

```
NAME=ahmed
```

وللتتأكد من ان الـ parameters أصبح لها value (اي حدث لها set)

```
[root@localhost srv]# NAME=ahmed
[root@localhost srv]# echo $NAME
ahmed
[root@localhost srv]#
```

نأتي الان الي اول طريقة للتعامل مع الـ parameters

PARAM:-VALUE

تستخدم هذه الـ form للتأكد من ان الـ parameter : null or unset (وتعني لا تضع اي variable ولكن تأكد فقط من وجودها)

فإذا كان الـ parameter له value او set فستظهر الـ value كنتيجة لهذا الـ command
اما اذا كانت الـ parameter ليس لها value او unset فسيظهر الـ value الموجودة في الـ form

```
 ${PARAM:-value}
```

فمثلا في حالة الـ parameter NAME السابق ، قمنا بعمل set له بقيمة ahmed

```
NAME=ahmed
```

فإذا قمنا بعمل

```
echo ${NAME:-'AMR'}
```

سيظهر لنا ahmed وليس AMR لأن NAME مخصصة له أساساً وبذلك تكون قد تأكيناً من أن الـ value لها مخصصة لها

```
[root@localhost srv]# NAME=ahmed
[root@localhost srv]# echo $NAME
ahmed
[root@localhost srv]# echo ${NAME:-'amr'}
ahmed
```

ولكن إذا قمنا بعمل unset على NAME واعدنا تنفيذ الأمر مرة أخرى

```
[root@localhost srv]# unset NAME
[root@localhost srv]# echo $NAME

[root@localhost srv]# echo ${NAME:-'amr'}
amr
[root@localhost srv]#
```

هنا ظهر AMR وهذا يدل على أن الـ value لها مخصصة ، وهذا لا يعني بالتأكيد أن هذه الـ variable أصبح لها قيمة جديدة

```
[root@localhost srv]# echo ${NAME:-'amr'}
amr
[root@localhost srv]# echo $NAME
[  
[root@localhost srv]#
```

وكما نرى في الصورة السابقة أن الـ NAME مازالت على الرغم من أننا نفذنا عليها الـ unset ، وهذا يدل على أن هذه الـ form تستخدم للتأكد فقط من وجود value

PARAM:=VALUE

اذا فكيف نقوم بعمل set لـ parameter ؟

نقوم باستخدام form شبيهه جدا بالسابقة وهي

```
PARAM:=value
```

هذه الـ form تعني انه اذا كانت الـ value parameter : (unset or null) فضع الـ value الجديد
والا فاترك الـ value القديمة (وتعني ضع الـ value فقط ان لم تكن موجودة)

وهي تقوم بعمل set لـ parameters او الـ variables في حالة انها unset او ليس لها
value كما بالصورة التالية

```
[root@localhost ~]# unset NAME
[root@localhost ~]# echo ${NAME:='ahmed'}
ahmed
[root@localhost ~]#
```

اذا فهذه الـ form تقوم بعمل check على الـ Variable لترى هل هو set ام unset او هل له
value ام لا

فإذا كانت set او لها value فتتركها كما هي ، اما اذا كانت unset او ليس لها value فتقوم
باضافة الـ value الجديدة

```
[root@localhost ~]# unset NAME
[root@localhost ~]# echo ${NAME:='ahmed'}
ahmed
[root@localhost ~]# echo $NAME
ahmed
[root@localhost ~]# echo ${NAME:='AMR'}
ahmed
[root@localhost ~]# echo $NAME
ahmed
[root@localhost ~]#
```

`\${PARAM:+VALUE}`

هذه الـ form تعني انه اذا كانت الـ parameter : (unset or null) فاتركها كما هي والا
فاستبدل الـ value الموجودة بالـ value الجديدة (وتعني استبدل الـ value فقط ان وجدت)

```
[root@localhost ~]# unset NAME
[root@localhost ~]# echo ${NAME:+ahmed}
[root@localhost ~]# NAME=amr
[root@localhost ~]# echo ${NAME:+ahmed}
ahmed
[root@localhost ~]# echo $NAME
amr → NAME is still amr
```

ولكن لاحظ اخر سطر في الصورة الذي يوضح ان ال value لم تتغير ، وان ال previous form كانت فقط للتأكد ايضا من وجود set او value لهذه ال parameter

`\${VARIABLE%PATTERN}`

هذه ال command تقوم بعمل match لـ variable value ولكن من نهايته وتقوم بحذفه وعرض الباقي (ولكن هذا لا يؤثر ايضا على ال value نفسها)

match the shortest pattern from the end of it

لاحظ المثال التالي :

```
[root@localhost ~]# NAME="ahmed gamil"
[root@localhost ~]# echo ${NAME%[a]}
ahmed gami
[root@localhost ~]# echo ${NAME%il}
ahmed gam
[root@localhost ~]# echo ${NAME%gamil}
ahmed
[root@localhost ~]# █
```

ويمكن استخدام ال wild cards في هذه ال command

```
[root@localhost ~]# NAME="ahmed gamil"
[root@localhost ~]# echo ${NAME%a*l}
ahmed g
[root@localhost ~]# echo ${NAME%m?l}
ahmed ga
[root@localhost ~]# echo ${NAME%m[a-z]l}
ahmed ga
```

وهنا قام بالبحث عن اقرب "a" و "l" من بعض وحذفهم وما بينهم من characters
والاحظ انه اذا كان هذا ال pattern مكرر اكثر من مرة في ال value فان ال form تقوم بعمل match مع اقرب pattern

```
[root@localhost ~]# NAME="ahmedahmed"
[root@localhost ~]# echo ${NAME%med}
ahmedah
```

MAGIC%%R*A

وهي مثل ال form السابقة مع الفرق ان هذه ال form تقوم بعمل matching لل longest pattern وليس ال shortest كما في ال form السابقة
انظر المثال التالي :

```
[root@localhost ~]# NAME="ahmedahmed"
[root@localhost ~]# echo ${NAME%%h*d}
a
[root@localhost ~]# echo ${NAME%*m*d}
ah
[root@localhost ~]#
```

وهنا قام بالبحث ابعد "h" و "d" عن بعض وحذفهم وحدث معهم ال الموجودة بينهم

VARIABLE#PATTERN

نلاحظ في ال الساقطة ان ال value يكون من نهاية ال matching ، **فهل يوجد value من بداية ال matching**

الاجابة هي نعم ، يمكن عمل ال value من بداية ال Matching باستخدام الصيغة

```
variable#pattern
```

ومثل ال "%" وال "# "فإن ال "#" هنا تقوم بعمل matching لـ pattern و هي shortest pattern اقرب pattern من بداية ال value (عكس "%" الموجودة في form سابقة)

```
[root@localhost ~]# NAME="ahmedahmed"
[root@localhost ~]# echo ${NAME#a*m}
edahmed
[root@localhost ~]#
```

وهنا قام بعمل Matching لاقرب "a" و "m" من بعض من بداية ال value وحذفها وحذف ما بينهم من ال characters

وال "#" تقوم بعمل matching لـ longest pattern عن بداية ال value

```
[root@localhost ~]# echo ${NAME##a*m}
ed
```

وهنا قام بعمل Matching لـ بعد "a" و "m" عن بعض من بداية ال value وحذفها وحذف ما بينهم من ال characters

اذا كيف نتذكر ان ال "#" تقوم بعمل matching من بداية ال value وال "%" من نهاية ال value ؟

كما نعلم ان ال "#" تستخدم كعلامة comment في ال scripts ، ودائما ما تكون موجودة في بداية السطر المراد عمل comment له ، لذلك فإن ال "#" تستخدم دائما في البداية

وكما نعلم ان علامة ال "%" تستخدم دائما في نهاية العملية الحسابية للحصول على ال percentage ، لذلك فهي تستخدم دائما في النهاية

اذا كيف نتذكر ان ال "%" تدل على ال shortest وان ال "%" تدل على ال longest وال "#" ايضا كذلك ؟

يمكن ان نتذكرة ذلك اذا اعتبرنا ان ال "%" هي مجرد علامة واحده فهي تدل على ال shortest بينما "%" علامتين اي انها اكبر فبذلك تدل على ال longest

VARIABLE:OFFSET:LENGTH

هذه ال form تقوم بعمل cut لعدد محدد من ال characters من اي جزء في ال variable ولا يشترط ان يكون من بداية ال variable او نهايته

فإذا كان ال variable ahmed فيمكن تحديد اي جزء من ال variable وحذف الباقي مثل :
ahmed او me او ahm

وهذه ال form تعني الاتي :

-1 : وهي ال variable التي سنعمل عليها مثل ahmed

-2 : وهي عدد ال characters التي ستجاهلها (من بداية ال variable) وهذا

يعني اننا اذا اردنا الحصول على حرفي "me" يجب ان نتجاهل حرفي "ah" فبذلك يكون
ال offset = 2

-3 : وهو عدد ال characters التي سنستخرجها من ال variable فإذا اردنا
استخراج حرفي "me" فان ال length=2

فإذا اردنا استخراج حروف "med" من ال variable ahmed فان الصيغة ستكون كالتالي :

```
variable:OFFSET:LENGTH  
ahmed:1:3
```

```
[root@localhost ~]# FIRST_NAME=ahmed  
[root@localhost ~]# SHORT_FIRST=${FIRST_NAME:0:3}  
[root@localhost ~]# echo $SHORT_FIRST  
ahm
```

#VARIABLE

باختصار تقوم هذه ال form بعرض عدد ال characters التي تتكون منها ال value

```
[root@localhost ~]# NAME=ahmed
[root@localhost ~]# echo $NAME
ahmed
[root@localhost ~]# echo ${#NAME}
5
[root@localhost ~]#
```

RE-ASSIGNING PARAMETERS WITH SET

تحدثنا سابقا عن الامر `set` الذي يستخدم في عرض الـ `shell variables` ، ولكننا سنشتخدمه الان بشكل اخر

فال `set` يمكن ان يستخدم في عمل `assign` لل `parameters` فال `command` يقوم ال `script` التالي بقراء `3 parameters` وعرضهم

```
[root@localhost ~]# cat superfluous.sh
#!/bin/bash
echo Command line positional parameters: $1 $2 $3
#set a b c
echo Reset positional parameters: $1 $2 $3
```

وتكون النتيجة كما يلي

```
[root@localhost ~]# ./superfluous.sh one two three
Command line positional parameters: one two three
Reset positional parameters: one two three
```

ولكن ماذا اذا وضعنا `set` في ال `script` كما يلي

```
[root@localhost ~]# cat superfluous.sh
#!/bin/bash
echo Command line positional parameters: $1 $2 $3
set a b c
echo Reset positional parameters: $1 $2 $3
[root@localhost ~]#
```

فستصبح النتيجة كالتالي

```
[root@localhost ~]# ./superfluous.sh one two three
Command line positional parameters: one two three
Reset positional parameters: a b c
[root@localhost ~]#
```

فماذا حدث ؟

بعد ان قام ال script بتحصيص \$1, \$2, \$3 الى one, two, three ، قامت set باعادة تحصيص ال a, b, c الى \$1, \$2, \$3

EXPLAINING THE DEFAULT FIELD SEPARATOR FIELD - IFS

تحدثنا ايضا في السابق عن ال delimiter IFS: input field separator وهي ال الخاصة بال cut command

```
[root@localhost ~]# set |grep IFS
IFS=$' \t\n'
```

وهذا يعني (space and tab and new line) ولكن يمكن تغييرها عن طريق

```
IFS=" , "
```

ولكن يمكن تغيير هذه ال value عند التعامل مع script معين فقط وذلك عن طريق

```
(IFS=' , ' ; script.sh)
```

ولاحظ وجود قوسين مستديرين () وهذا يدل علي ان هذا التغيير سيتم في shell subshell فقط وليس في ال shell الرئيسي ولن تتغير ال IFS الاصلية

SETTING VARIABLES AS "READONLY"

واخيرا في هذا ال chapter سنتحدث عن كيفية تحصين ال variable وجعلها readonly بحيث لا يمكن تغييرها الا اذا اغلقت ال shell او قمت بعمل kill لها

ويتم ذلك عن طريق هذه الصيغة:

Readonly NAME=ahmed

وبذلك فان ال variable NAME لن يمكن تغيير ال value الخاصة بها او حتى عمل unset لها

```
[root@localhost ~]# readonly NAME=ahmed
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# NAME=amr
bash: NAME: readonly variable
[root@localhost ~]#
[root@localhost ~]# unset NAME
bash: unset: NAME: cannot unset: readonly variable
[root@localhost ~]#
```

CHAPTER 12. BITS AND PIECES – TYING UP THE LOOSE ENDS

THE EVAL COMMAND

اذا قمنا بتنفيذ هذا ال command واستخدمنا ال " | " pipe

```
ls | wc -l
```

```
[root@localhost ~]# ls |wc -l
22
[root@localhost ~]#
```

ولكن هل يمكن وضع ال " | " في variable بحيث يمكن استبداله بال variable كال التالي ؟

```
PIPE='|'
ls $PIPE wc -l
```

الاجابة هي لا ، والصورة التالية توضح ذلك

```
[root@localhost ~]# ls |wc -l
22
[root@localhost ~]# PIPE='|'
[root@localhost ~]# ls $PIPE wc -l
ls: cannot access |: No such file or directory
ls: cannot access wc: No such file or directory
[root@localhost ~]#
```

لان ال ls command يقوم بعمل variable expansion اولا ثم تقوم بتنفيذ ال ls command بدون ان يعيد قراءة القيمة الجديدة لل variable ثم يقوم بالبحث عن file يسمى ' | ' وآخر يسمى wc ، لذلك ظهر ال error : No such file or directory

وهنا تأتي وظيفة ال eval command فهو يقوم باعادة قراء ال command line بعد استبدال ال variable

```
[root@localhost ~]# PIPE='|'  
[root@localhost ~]# ls $PIPE wc -l  
ls: cannot access |: No such file or directory  
ls: cannot access wc: No such file or directory  
[root@localhost ~]# eval ls $PIPE wc -l  
22
```

مثال اخر:

```
[root@localhost ~]# cmd='cat file* | sort'  
[root@localhost ~]# echo $cmd  
cat file file1 file2 file3 | sort  
[root@localhost ~]# eval echo $cmd  
cat file file1 file2 file3  
[root@localhost ~]#
```

نلاحظ ان ال eval قام بترجمة ال pipe ايضا ، ولاحظ في الصورة التالية اننا عندما نفذنا ال error ظهر بدون echo command

```
[root@localhost ~]# cmd='cat file* | sort'  
[root@localhost ~]# $cmd  
cat: |: No such file or directory  
cat: sort: No such file or directory  
[root@localhost ~]# echo $?  
1
```

ولكن عندما وضعنا eval

```
[root@localhost ~]# cmd='cat file* | sort'  
[root@localhost ~]# eval $cmd  
[root@localhost ~]# echo $?  
0  
[root@localhost ~]#
```

مثال اخر:

اذا كان لدينا $x=100$ و $ptr=x$

فإذا عرضنا قيمة ال ptr باستخدام ال echo فسنجد أنها ستتساوي x وليس 100

```
[root@localhost ~]# x=100
[root@localhost ~]# ptr=x
[root@localhost ~]# eval echo $ptr
x
[root@localhost ~]#
```

ولكي نعرض ال `x` يجب ان نضع النتيجة علي الشكل التالي

```
eval echo $x
```

او بمعنى اخر سنقوم بالتعويض عن `x` في ال `form`

```
eval echo $$ptr
```

ولكننا سنجد ان النتيجة ظهرت كالتالي

```
[root@localhost ~]# eval echo $$ptr
2100ptr
[root@localhost ~]#
```

سنلاحظ اننا عندما وضعنا `$` قام بترجمة ال `$` على انها ال PID ، اذا فيجب ان نفصل بين ال `$` الاولى والثانية باستخدام `{}`

```
eval echo ${$ptr}
```

```
[root@localhost ~]# eval echo ${$ptr}
bash: ${$ptr}: bad substitution
[root@localhost ~]#
```

وهنا فان ال shell تفهم ال `$` الموجودة خارج ال `{}` علي انها special character فلا تستطيع ال `echo` ان تترجم `form` تحتوي علي `2` special characters ، لأن هذه `2` تعني `$ptr` ولا تعني `{x}`

لذلك يجب وضع "\\" قبل ال `$`

```
eval echo \\${$ptr}
```

```
[root@localhost ~]# eval echo \$ptr  
100  
[root@localhost ~]#
```

RUNNING COMMANDS IN THE BACKGROUND USING &

لأن ال linux هو multi-task operating system فيمكن تنفيذ اكثرا من command في نفس الوقت

ولكن لكي يتم ذلك يجب ان نجعل ال process التي يتم تنفيذها تعمل في الخلفية لكي نستطيع ان نحصل على curser جديد وننفذ ال process الجديدة

ويتم ذلك عن طريق اضافة الرمز "&" بجانب ال command الاول كالتالي

```
find / -name "ahmed" &
```

```
[root@localhost ~]# find / -name "ahmed" &  
[7] 5380
```

ونلاحظ ظهور ال PID الخاص بال process

واذا اردنا ان نتأكد منها يمكن تنفيذ هذه ال form

```
echo $!
```

```
[root@localhost ~]# echo $!  
5380
```