

Topics

- When to script
- Scripting Basics
- Some useful tricks
- Troubleshooting

When to Script

When To Scripts

Shell scripting can be applied to a wide variety of system tasks.

Repeated Tasks

Necessity *is* the mother of invention. The first candidates for shell scripts will be manual tasks which are done on a regular basis.

- Backups
- Log monitoring
- Check disk space

Occasional Tasks

Tasks which are performed rarely enough that their method, or even their need may be forgotten.

- Periodic business related reports (monthly/quarterly/yearly)
- Offsite backups
- Purging old data

Complex Manual Tasks

Some tasks must be performed manually but may be aided by scripting.

- Checking for database locks
- Killing runaway processes

These tasks may evolve into repeated tasks

Special Tasks

These are tasks which would not be possible without a programming language.

- Storing OS information (performance stats, disk usage, etc.) into the database
- High frequency monitoring (several times a day or more)

Scripting Basics

Before You Start Scripting

You will find shell scripting an iterative process, but it is best to have a good idea of your goals when you start.

- What are you trying to accomplish
- What are the dependencies
 - Which dependencies can we check first
 - Which dependencies cannot be checked
- How broad will the effects of this script be
- What happens if any step fails
 - Should the script continue or be halted
- What results or output do we want from the script
 - Who should be notified of the results and how
- What cleanup should be done when the script is complete
- What if two copies of the script get executed simultaneously

Scripting Tools

Any **plain text** editor will work.

- vi (Command line UNIX)
- Notepad (Windows)
- TextEdit (Mac OSX)

The Shell

Shell scripting allows us to use commands we already use at the command line. This considerably eases the learning curve.

We are familiar with the *interactive* mode of the shell. Almost anything can be done in a script which can be done at the command line.

Which Shell to Use

My preference is Bash (bash) because of its ubiquity and compatibility with Bourne (sh).

Other common shells include:

- C shell (csh)
- Korn shell (ksh)
- Z Shell (zsh)

It is important to pick a shell and stick with it. The differences between shells are often small but infuriating.

The Anatomy of a Command

```
grep -i localhost /etc/hosts
```

Command Option Arguments

Options change the behavior of a command

Arguments control what the command acts upon

Variables

Variables are set using the = sign

```
ORACLE_SID=oss
```

Variables and their contents **are** case sensitive, so the variable `ORACLE_SID` is different from the variable `oracle_sid`.

Shell variables are un-typed and may contain integers or text.

Numbers with a decimal point will be treated as text. (e.g. `3.14`)

Variable Naming

- Variables should have meaningful names
- Variable names do not need to be short
- All UPPER CASE typically indicates an environmental variable
- Local (script) variables are conventionally all lowercase
- Underscores (_) are best for separating words in variable names

Variable Scope

- Variables will be available within the script (or shell session) which sets them
- By exporting variables they can be made available to subsequently called scripts.

This is why we typically perform an

```
export ORACLE_SID
```

after setting the variable.

Exporting is not necessary when variables will only be used within the current script.

Using Variables

The dollar sign (\$) is used to retrieve the contents of a variable.

```
$ echo $ORACLE_SID
```

```
OSS
```

If you are trying to use a variable where it may be surrounded by other letters you may need to add curly braces {} around the name.

```
$ echo ${ORACLE_SID}_sid
```

```
oss_sid
```

Comments and Whitespace

- Anything appearing after a pound symbol (#) on a line will be ignored.
- Adding comments can aid troubleshooting and future editing of the script.
- Blank lines are ignored when a script is executed.
- Blank lines and other whitespace (tabs, spaces) can be used to improve script readability.

A basic script

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

```
echo "The current database is $ORACLE_SID"
```

```
echo "The current running processes for  
$ORACLE_SID are"
```

```
ps -ef | grep $ORACLE_SID
```

A basic script

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



This first line indicates what interpreter to use when running this script

```
echo "The current database is $ORACLE_SID"
```

```
echo "The current running processes for  
$ORACLE_SID are"
```

```
ps -ef | grep $ORACLE_SID
```

A basic script

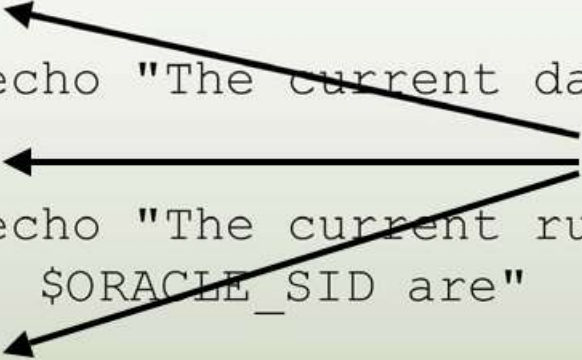
```
#!/bin/bash
```

```
echo "The current database is $ORACLE_SID"
```

```
echo "The current running processes for  
$ORACLE_SID are"
```

```
ps -ef | grep $ORACLE_SID
```

Whitespace is used to
separate commands to
improve readability.




A basic script

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

```
echo "The current database is $ORACLE_SID"
```

```
echo "The current running processes for  
$ORACLE_SID are"
```

```
ps -ef | grep $ORACLE_SID
```



Variables referenced here
must have already been
set and exported.


A basic script

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

```
echo "The current database is $ORACLE_SID"
```

```
echo "The current running processes for  
$ORACLE_SID are"
```

```
ps -ef | grep $ORACLE_SID
```



Note the variable being
used as an argument.
We'll see a lot of this.

The Shebang (# !)

The "shebang" is a special comment. Since it is a comment it will not be executed when the script is run. Instead *before* the script is run, the shell calling the script will check for the # ! pattern. If found it will invoke the script using that interpreter. If no # ! is found most shells will use the current shell to run the script.

The Shebang (cont)

Since the shells are installed in different locations on different systems you may have to alter the `#!` line. For example, the `bash` shell may be in `/bin/bash`, `/usr/bin/bash` or `/usr/local/bin/bash`.

Setting the shell explicitly like this assures that the script will be run with the same interpreter regardless of who executes it (or what their default shell may be.)

Script Naming

Descriptive names are important.

- Use full words
- Separate words with underscores
- Avoid using spaces or other unusual characters
- There is no requirement for script names, but typically they will end in `.sh`

Script Permissions

The execute permission must be turned on before a script can be executed. It can be turned on for the user (u), group (g) or all users (o) by using the `chmod` command.

```
chmod ugo+x test_script.sh
```

If execute has not been granted you will get an error like this:

```
-bash: ./test_script.sh: Permission denied
```

status.sh

```
#!/bin/sh

# Show the user and host name
echo "Hello $USER!"
echo "Welcome to `hostname`"
echo "--- Current Disk Usage ---"
df -h
# On some systems the -h (human readable) option will not work with df
# In that case you can use the -k option to display output in killobytes

echo "--- Current uptime, users and load averages ---"
uptime

echo "--- Load average numbers represent the 1, 5 and 15 minute load
averages ---"
echo "--- Lower numbers are better for load averages ---"
# These are the first two things I check when I think there is a problem
# with a system, but I'm sure you can think of some other things to add
here
```

status.sh

```
#!/bin/sh
```

```
# Show the user and host name
```

```
echo "Hello $USER!"
```

```
echo "Welcome to `hostname`"
```

```
echo "--- Current Disk Usage ---"
```

```
df -h
```

```
# On some systems the -h (human readable) option will not work with df
```

```
# In that case you can use the -k option to display output in killobytes
```

```
echo "--- Current uptime, users and load averages ---"
```

```
uptime
```


```
echo "--- Load average numbers represent  
averages ---"
```

```
echo "--- Lower numbers are better for 1
```


```
# These are the first two things I check
```

```
# with a system, but I'm sure you can th  
here
```

This output will help
the user identify what
they are looking at.

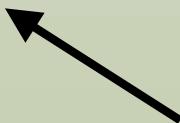


This comment explains the
command option used and
how it may need to be
changed on some systems.



status.sh Usage

```
$ ./status.sh
Hello oracle!
Welcome to glonk
--- Current Disk Usage ---
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup00-LogVol00
                          72G   6.5G   61G   10% /
/dev/hda1                  99M   9.8M   84M   11% /boot
/dev/shm                   252M     0   252M    0% /dev/shm
--- Current uptime, users and load averages ---
 19:17:41 up 10 days,  6:02,  2 users,  load average:
 0.00, 0.02, 0.00
--- Load average numbers represent the 1, 5 and 15 minute
    load averages ---
--- Lower numbers are better for load averages ---
```



This additional output provides very useful information on the results we're looking at.

Basic Script Setup

- Make a plan!
- Create a new text file
- Specify the interpreter to be used (`# !`)
- Set variables using `=`
- Retrieve variable contents using `$`
- Add `{ }` around variable name if necessary
- Use comments (`#`) and whitespace (blank lines, spaces and tabs) to improve readability
- Grant execute permissions to the appropriate users with `chmod`

Running Your Script

If the proper execute permissions have been applied:

```
./test_script.sh
```

```
/home/oracle/test_script.sh
```

If `.` is in your `$PATH` variable

```
test_script.sh
```


Keeping Your Scripts Organized

- Work with sysadmins and DBAs to come up with a convention
- Development should be done in an area away from production scripts
- Scripts for a specific database in
`/u01/app/oracle/admin/sid/scripts`
- Scripts used on multiple databases in
`/u01/app/oracle/admin/common/scripts`

Decisions and Loops

The if Statement

The simplest flow control statement is the `if` statement.

```
$ age=29
```

```
$ if [ $age -lt 30 ]
```

```
> then
```

```
> echo "You're still under 30"
```

```
> fi
```

```
You're still under 30
```

The if Statement

The simplest flow control statement is the `if` statement.

```
$ age=29
```

```
$ if [ $age -lt 30 ]
```

```
> then
```

```
> echo "You're st
```

```
> fi
```

```
You're still under 30
```

Note that the end of an if statement is indicated by the keyword `fi`

if, elseif and else

```
#!/bin/sh
age=39
if [ $age -lt 30 ]
then
    echo "You're still under 30"
elif [ $age -ge 30 -a $age -le 40 ]
then
    echo "You're in your 30s"
else
    echo "You're 40 or over"
fi
```

if, elseif and else

```
#!/bin/sh
```

```
age=39
```

```
if [ $age -lt 30 ]
```

```
then
```

```
    echo "You're still under 30"
```

```
elif [ $age -ge 30 -a $age -le 40 ]
```

```
then
```

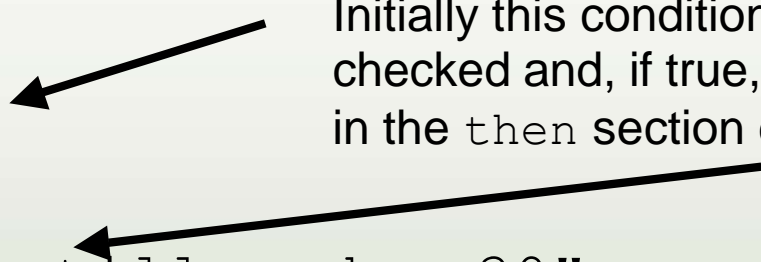
```
    echo "You're in your 30s"
```

```
else
```

```
    echo "You're 40 or over"
```


```
fi
```

Initially this condition is checked and, if true, the code in the `then` section executed



if, elif and else


```
#!/bin/sh
age=39
if [ $age -lt 30 ]
then
    echo "You're still under 30"
elif [ $age -ge 30 -a $age -le 40 ]
then
    echo "You're in"
else
    echo "You're 40 or over"
fi
```



Only if the initial condition has failed will the elif be considered

if, elif and else

```
#!/bin/sh
age=39
if [ $age -lt 30 ]
then
    echo "You're still under 30"
elif [ $age -ge 30 -a $age -le 40 ]
then
    echo "You're in your 30s"
else
    echo "You're 40 or over"
fi
```



Finally if the if condition and all elif conditions have failed the else, if present, will be executed

if, elif and else

- Conditional statements can compare numbers or text
- An `if` statement will need to have a `then` and an `fi` to indicate the end of the statement
- An `if` statement can have one or more `elif` statements or may have none
- An `if` statement may have one `else` statement but may have no `else` statement
- Only one section of code will be executed

Mathematical Comparators

Mathematical Comparators

Comparator	Mathematic Equivalent	Evaluates to true if
<u>-eq</u> or =	=	the values on each side of the comparator are equal
<u>-ne</u> or !=	≠	the two values are not equal
<u>-gt</u>	>	the first value is greater than the second
<u>-ge</u>	≥	the first value is greater than or equal to the second
<u>-lt</u>	<	the first value is less than the second
<u>-le</u>	≤	the first value is less than or equal to the second

String Comparators

String Comparators

Comparator	Evaluates to true if
=	the string on each side of the comparator are identical
!=	the string on each side of the comparator are not identical
<i>string</i>	the length of the string is not zero

Comparing Strings

```
$ if [ $ORACLE_SID = "oss" ]  
> then  
> echo "Using the sid for the Oracle Shell  
  Scripting database"  
> fi
```

```
Using the sid for the Oracle Shell Scripting  
database
```

Checking Variables

```
$ if [ $ORACLE_SID ]  
> then  
> echo "ORACLE_SID variable is set to $ORACLE_SID"  
> fi  
ORACLE_SID variable is set to oss
```

This statement checks to see if the variable
\$ORACLE_SID has been set.

The statement will fail if the variable has not
been set, or if it is set to a null value.

File Comparators

File Comparators

Comparator	Evaluates to true if
<u>-nt</u>	the file listed before is newer than the file listed after the comparator
<u>-ot</u>	the file listed before is older than the file listed after the comparator
<u>-e</u>	the file exists
<u>-d</u>	the file is a directory
<u>-h</u>	the file is a symbolic link
<u>-s</u>	the file is not empty (has a size greater than zero)
<u>-r</u>	the file is readable
<u>-w</u>	the file is writable

Checking Files

```
$ if [ -e  
  $ORACLE_HOME/dbs/init$ORACLE_SID.ora ]  
> then  
> echo "An init file exists for the  
  database $ORACLE_SID"  
> fi
```

An init file exists for the database oss

Complex Comparisons

Comparator Modifiers

Comparator	Evaluates to true if
-a	the expressions on each side of the comparator are both true
-o	one or both of the expressions are true
!	The following expression is false

Checking Multiple Files

```
$ if [ -e $ORACLE_HOME/dbs/init$ORACLE_SID.ora -a -e \  
> $ORACLE_HOME/dbs/spfile$ORACLE_SID.ora ]  
> then  
> echo "We seem to have both an spfile and an init file"  
> fi
```

We seem to have both an spfile and an init file

Case Statement

```
#!/bin/sh
case $ORACLE_SID
in
    oss)
        echo "Using the sid for the Oracle Shell
Scripting database"
        ;;
    db1)
        echo "Using the default Oracle database"
        ;;
    *)
        echo "I don't have a description for this
database"
        ;;
esac
```

Case Statement

```
#!/bin/sh
case $ORACLE_SID
in
    oss)
        echo "Using the sid for
Scripting database"
        ;;
    db1)
        echo "Using the default Oracle database"
        ;;
    *)
        echo "I don't have a description for this
database"
        ;;
esac
```

The beginning of a case statement is indicated by the case keyword. The end is indicated by case spelled backwards

Case Statement

```
#!/bin/sh
case $ORACLE_SID in
    oss)
        echo "Using the sid for the Oracle Shell
        Scripting database"
        ;;
    db1)
        echo "Using the default Oracle database"
        ;;
    *)
        echo "I don't have a description for this
        database"
        ;;
esac
```

The input given at the beginning will be compared to each value in the list

The asterisk is a wildcard and will match any string

The code to be executed for each option is terminated by a double semicolon.

Case Statement

- The code following the **first** matching option will be executed.
- If no match is found the script will continue on after the `esac` statement without executing any code.
- Some wildcards and regular expressions can be used.
- A `case` could be rewritten as a series of `elif` statements but a `case` is typically more easily understood.

The while Loop

The `while` loop will repeat a chunk of code as long as the given condition is true.

```
#!/bin/sh
i=1
while [ $i -le 10 ]
do
    echo "The current value of i is $i"
    i=`expr $i + 1`
done
```

The while Loop

```
#!/bin/sh
```

```
i=1
```

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


```
do
```

```
    echo "The current value of i is $i"
```

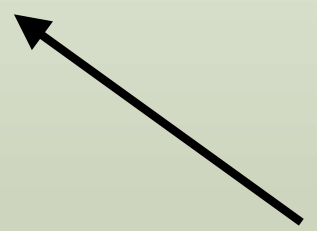
```
    i=`expr $i + 1`
```

```
done
```

Make sure your loop variable is initialized before the loop starts



Also makes sure that something will eventually cause the while condition to fail, otherwise you may end up in an infinite loop!



The for Loop

The for loop allows you to easily parse a set of values.

```
#!/bin/sh
count=0
for i in 2 4 6
do
    echo "i is $i"
    count=`expr $count + 1`
done
echo "The loop was executed $count times"
```


The for Loop

```
#!/bin/sh
```

```
count=0
```

```
for i in 2 4 6
```

```
do
```


```
    echo "i is $i"
```

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

```
done
```

```
echo "The loop was executed $count  
times"
```

This for loop will be executed three times, once with i=2, once with i=4 and once with i=6



Breaking Out of the Current Loop

The `break` statement will cause the shell to stop executing the current loop and continue on after its end.

```
#!/bin/sh
files=`ls`
count=0
for i in $files
do
    count=`expr $count + 1`
    if [ $count -gt 100 ]
    then
        echo "There are more than 100 files in the current
        directory"
        break
    fi
done
```

Prompting for User Input

For scripts which will be run interactively we can prompt the user to give us input.

The `read` command can be used to set a variable with a value read from user input.

```
#!/bin/sh
echo "Enter your name"
read name
echo "Hi $name.  I hope you like this script"
```

Prompting for User Input

```
$ ./welcome.sh
```

```
Enter your name
```

```
Jon
```

```
Hi Jon.  I hope you like this script
```

Note that the text input will be displayed on the screen. If you do not want the input displayed (like when accepting a password) use the `-s` option for the `read` command.

Using Arguments

Accepting arguments to your script can allow you to make a script more flexible.

The variables \$1, \$2, \$3 etc. refer to the arguments given in order.

The variable \$@ refers to the complete string of arguments.

The variable \$# will give the number of arguments given.

Using Arguments

```
if [ $1 ]
then
    ORACLE_SID=$1
    ORAENV_ASK=NO
    . oraenv
else
    if [ ! $ORACLE_SID ]
    then
        echo "Error: No ORACLE_SID set or provided as
        an argument"
        exit 1
    fi
fi
```

Using Arguments

```
if [ $1 ]
then
    ORACLE_SID=$1
    ORAENV_ASK=NO
    . oraenv
else
    if [ ! $ORACLE_SID ]
    then
        echo "Error: No ORACLE_SID set or provided as
        an argument"
        exit 1
    fi
fi
```

Check to see if an argument was given

If it was, we will use it to set the ORACLE_SID variable then execute oraenv

Getting Past the Password Problems

A combination of two methods can be used to get around the password problems.

- Place the password in a variable so it will not display in a process listing.
- Rather than placing the password in the shell script store it in a separate, secure file.

Placing the Password in a Variable

```
#!/bin/sh  
system_pw=manager  
sqlplus -S system/$system_pw @database_status.sql
```

When this command is running a process listing (ps) will show the variable name (\$system_pw) instead of the password.

Reading the Password from a Secure File

```
#!/bin/sh
system_pw=`cat
    /u01/app/oracle/admin/oss/pw/system.pw`
sqlplus -S system/$system_pw @database_status.sql
```

By reading the password from a text file the script is no longer required to have the password embedded in it.

This has the added advantage of providing a single location where passwords can be changed for all scripts at once.

Securing the Password Files

In order to keep the passwords secure the files which contain them should have as restrictive permissions as possible. Using the `chmod` command we can grant the owner (typically the oracle user) read and write permissions and revoke all permissions for other users.

```
chmod u=rw,g=,o=  
/u01/app/oracle/admin/oss/pw/system.pw
```

Manipulating Other Commands

- These methods can also be applied with RMAN for backup and recovery.
- File markers can be used to emulate user input for many (but not all) commands.

Some Useful Tricks

Escape Character

The escape character will prevent the shell from interpreting the following character as anything other than text.

Backslash (\) is the escape character in the Bash shell.

Escaping special characters (such as * ' \$; and space) can help you get the output you want and to handle special characters in file names.

```
$ echo "The escape character in Bash is \"\\\""
```

The escape character in Bash is "\"

Single Quotes

Single quotes will cause all special characters (except the single quote) to be ignored.

```
$ echo 'In single quotes "double quotes",  
$ and even ; are all safe'
```

```
In single quotes "double quotes", $ and  
even ; are all safe
```

Double Quotes

Double quotes will cause most special characters to be ignored.

Variables and back quotes will be expanded and backslashes are interpreted as an escape character.

```
$ echo "In double quotes we can use  
variables like $ORACLE_SID"
```

```
In double quotes we can use variables like  
OSS
```


Back Quotes

Text between back quotes (```) is executed as a command and its output substituted in its place. This allows us to concatenate command results with text.

```
$ echo "The current date and time is  
`date`"
```

```
The current date and time is Sun May  6  
23:19:55 EDT 2007
```

Redirecting Output to a File

- Output from commands can easily be sent to a file instead of the display with a `>` or `>>`
- The `>` will replace the given file if it exists but the `>>` will concatenate the output on the end of the given file
- Both the standard output and the error output can be redirected to a file

Redirecting Standard Output

```
$ ls
log1.log  log3.log  myfile.txt  sample.txt
types_of_unix.txt
log2.log  marx.txt  output.txt  test_script.sh
$ ls > listing.txt
$ more listing.txt
listing.txt
log1.log
log2.log
log3.log
marx.txt
myfile.txt
output.txt
sample.txt
test_script.sh
types_of_unix.txt
```

Redirecting Error Output

```
$ find ./ -name "*.txt" >  
text_files.txt 2>errors.txt
```

While `>` or `>>` redirect standard output `2>` or `2>>` will redirect error output.

Standard or error output can be redirected to `/dev/null` (`2>/dev/null`) to discard the output

Linking Output to Input

The pipe (|) can be used to link the output of one command to the input of another.

```
$ ps -ef | grep oss
```

oracle	2684	1	0	14:02	?	00:00:00	ora_pmon_oss
oracle	2686	1	0	14:02	?	00:00:00	ora_psp0_oss
oracle	2688	1	0	14:02	?	00:00:00	ora_mman_oss
oracle	2690	1	0	14:02	?	00:00:02	ora_dbw0_oss
oracle	2692	1	0	14:02	?	00:00:03	ora_lgwr_oss
oracle	2694	1	0	14:02	?	00:00:01	ora_ckpt_oss
oracle	2696	1	0	14:02	?	00:00:06	ora_smon_oss
oracle	2698	1	0	14:02	?	00:00:00	ora_reco_oss

Performing Math in the Shell

- The `expr` command can be used to perform simple math in the shell.

```
$ expr 2 + 7
```

```
9
```

```
$ expr 4 + 3 \* 3
```

```
13
```

```
$ expr 13 / 2
```

```
7
```

The asterisk is used for multiplication but must be escaped by a backslash.

Results will be truncated to whole numbers.

Sending Email

Sending email is simple!

Use -s to specify a subject line, give an address as an argument (or list multiple addresses in quotes and separated by commas) and redirect a file *into* the command.

```
mail -s "Alert log from $ORACLE_SID `hostname`"
      oracle <
      /u01/app/oracle/admin/$ORACLE_SID/bdump/alert_$ORA
      CLE_SID.log
```

Scheduling with Cron

Repeated tasks may be scheduled with the `crontab` command.

`crontab -e` will edit the current user's crontab with the default editor.

Comments can be put into the crontab with the `#`.

```
# Weekly full hot backup
00 03 * * 0
    /u01/app/oracle/admin/common/scripts/hot_backup.sh
    oss 0
```


Crontab entries are executed when all the specified time conditions are met.*

```
00 03 * * 0      /u01/app/oracle/admin/com...
```

This entry will be executed at 0 minutes past the hour, the hour of 3(am), any day of the month, any month of the year, but only if it is Sunday.

Field	Minute	Hour	Day of Month	Month	Day of Week	Command
Valid values	0-59	0-23	1-31	1-12	0-6	Command path/command

*On many platforms if the day of the week and day of month/month of year are both specified the job will be executed when *either* condition is met.

So, the following job would run on the first Sunday of the month on some platforms, but on others would run every Sunday *and* the 1st through 7th of each month.

00 03 1-7 * 0

Scheduling One-time Tasks with `at`

Use `at` for one-time tasks which need to be run off-hours or at a specific time.

`at` can easily schedule jobs to run at a specific time today, tomorrow, or on a specified date and time.

Just like with cron output from commands run with `at` will be sent to the user via email. If you would like an email when the job completes, regardless of output just add the `-m` flag.

Run an export at 11:30 pm today:

```
$ at 23:30
```

```
at>
```

```
    /u01/app/oracle/admin/oss/scripts/full_exp  
    ort.sh
```

```
at> ctrl-d <EOT>
```

```
job 5 at 2007-01-21 23:30
```

Run an export at 11:00 am tomorrow and email me when complete:

```
$ at -m 11:00 tomorrow
```

```
at>
```

```
    /u01/app/oracle/admin/oss/scripts/full_exp  
    ort.sh
```

```
at> ctrl-d <EOT>
```

```
job 6 at 2007-01-22 11:00
```

Managing at Jobs

The atq command will list jobs in the at queue.

```
$ atq
```

```
6          2007-01-22 11:00 a oracle  
5          2007-01-21 23:30 a oracle
```

To remove a job use atrm with the job number from the queue.

```
$ atrm 6
```

Troubleshooting Tips

Determining where a failure is happening

Add lines like `echo "Completed first for loop"` or `echo "About to launch sqlplus"` to help pinpoint errors.

Echo count variables. `echo "Loop completed time $i"`

When you're done with these markers simply comment them out with a pound rather than removing them. You might need them again.

Debug Mode

Running a script in debug mode will print each line of the shell script (including comments) before it is executed.

Enable debug mode by adding `-v` after the interpreter listing at the shebang.

```
#!/bin/sh -v
```

Leaving this mode on will generate a lot of output and may expose passwords. Debug mode should be enabled when needed and immediately disabled when done.


```
$ ./status.sh
#!/bin/bash -v
#
# status.sh script
# Published in Oracle Shell Scripting, Rampant
#   TechPress, 2007
#
# A simple script to provide some information about
#   the system
# Show the user and host name
echo "Hello $USER!"
Hello oracle!
echo "Welcome to `hostname`"
hostname
Welcome to glonk
echo "--- Current Disk Usage ---"
--- Current Disk Usage ---
df -h
```

Show Commands After Variable Substitution

Another option, `-x`, will show each command once variables have been substituted in.

Debug output will appear with a `+` at the beginning of each line.

This can help determine where problems are with loops and `if` statements.

`-vx` can be specified if both debug modes are desired

Again, this mode should only be enabled when needed.

```
$ ./status.sh
```

```
+ echo 'Hello oracle!'
```

```
Hello oracle!
```

```
++ hostname
```

```
+ echo 'Welcome to glonk'
```

```
Welcome to glonk
```

```
+ echo '--- Current Disk Usage ---'
```

```
--- Current Disk Usage ---
```

```
+ df -h
```

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/mapper/VolGroup00-LogVol00	72G	6.6G	61G	10%	/
/dev/hda1	99M	9.8M	84M	11%	/boot
/dev/shm	252M	0	252M	0%	/dev/shm