Introduction to Bash scripting language

Day 4

Alessandro Sciarra

Z02 – Software Development Center

Organised by the CSC Frankfurt

29.10.2020

Topics of the day

1. Asynchronous commands
2. Traps
3. Error handling
4. Sed (and Awk)

Exploring the dark side of bash

Today we will finalise your knowledge adding a few powerful bricks that will provide you with lots of versatility and great abilities!

Asynchronous commands

Huge magma bombs

Processes in the terminal

🔗 Bash manual v5.0 pages 104-107

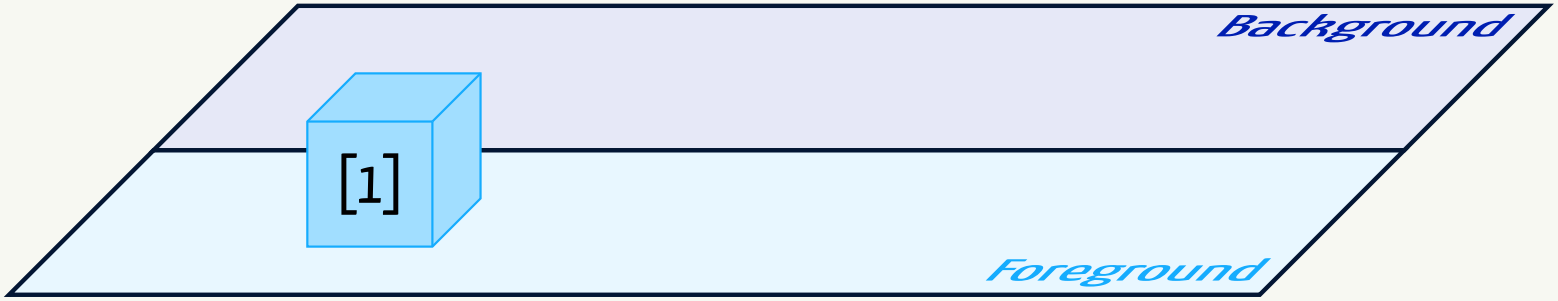
- The `ps` command displays information about (a selection of) the active processes
- A **process** is «an instance of a computer program that is being executed» – **Wikipedia**
- The shell associates a **job** with each pipeline
- It keeps a table of currently executing jobs, which may be listed with the `jobs` builtin
- When Bash (in interactive mode*) starts a job asynchronously (i.e. in background), it prints a line that looks like

```
$ sleep 30 & # <- The ampersand symbol starts
[1] 25647    #    a process in background
```

  indicating that this job is job number 1 and that the process ID of the last process in the pipeline associated with this job is 25647 { We will come back to the process ID later }
- All of the processes in a single pipeline are members of the same job
- Bash uses the job abstraction as the basis for job control

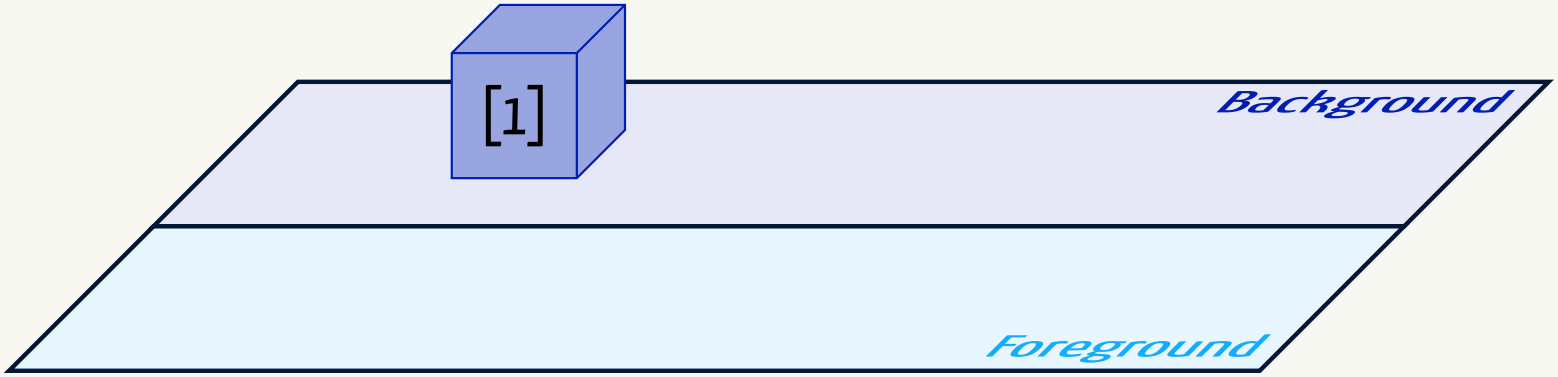
* Job control is by default turned off in non-interactive mode, but it might be activated via `set -m`.

Job control: A graphical overview



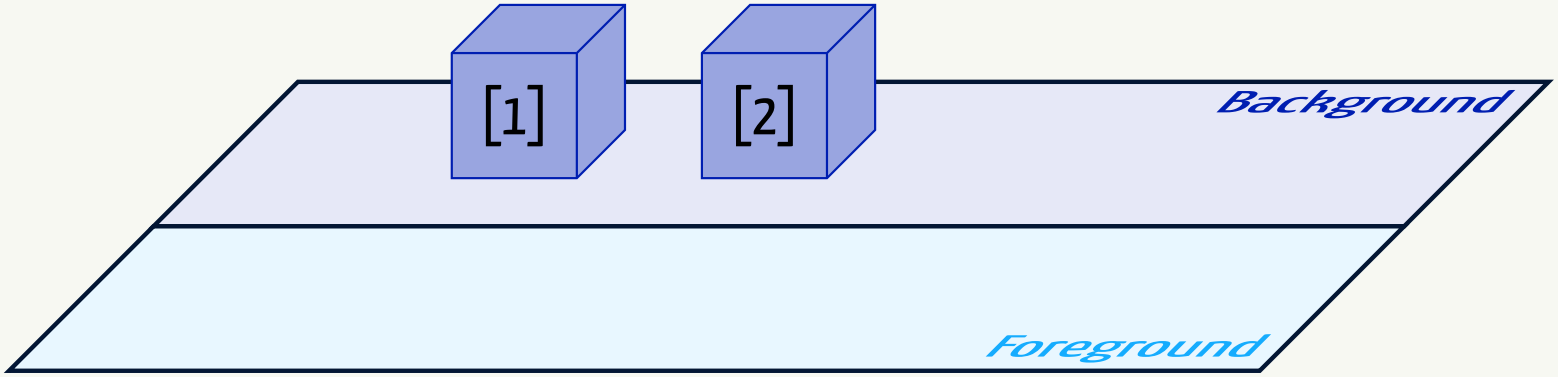

```
$ jobs
$ emacs Loops.tex   # <- each pipeline is a fg job by default
# Shell is blocked till you have the emacs process
# in foreground. You can close it to get the shell back.
```

Job control: A graphical overview



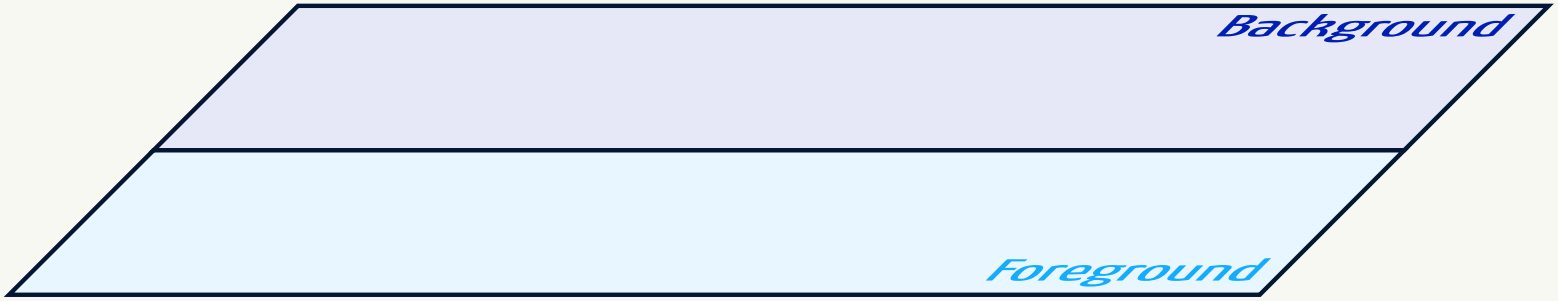

```
$ jobs
$ emacs Loops.tex   # <- each pipeline is a fg job by default
# Shell is blocked till you have the emacs process
# in foreground. You can close it to get the shell back.
$ emacs Loops.tex & # <- use the ampersand to start it in bg
[1] 5642
$ jobs
[1]+  Running                 emacs Loops.tex &
```

Job control: A graphical overview



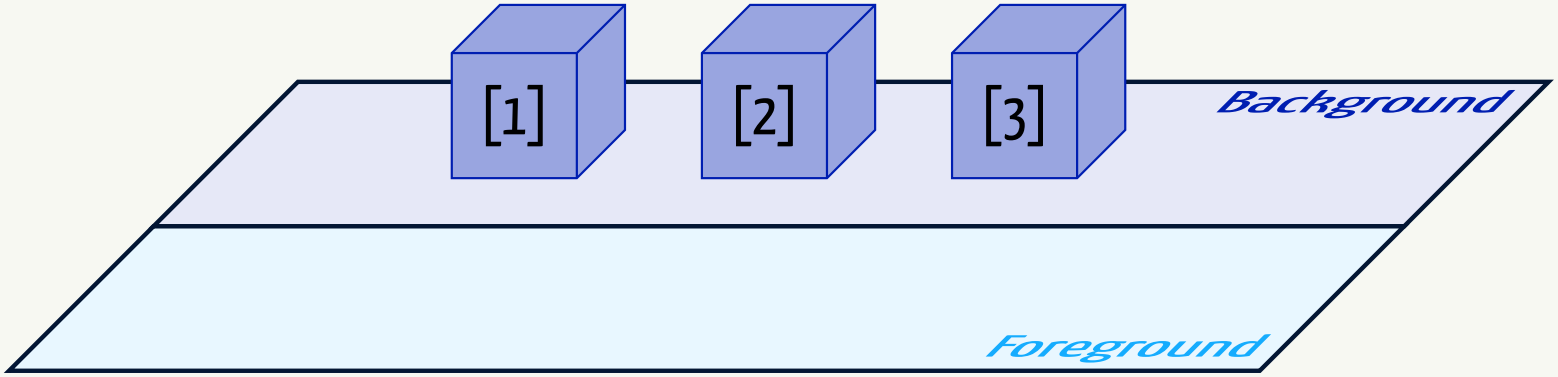

```
$ jobs
$ emacs Loops.tex   # <- each pipeline is a fg job by default
# Shell is blocked till you have the emacs process
# in foreground. You can close it to get the shell back.
$ emacs Loops.tex & # <- use the ampersand to start it in bg
[1] 5642
$ jobs
[1]+  Running                 emacs Loops.tex &
$ sleep 30 &
[2] 20943
$ jobs
[1]-  Running                 emacs Loops.tex &
[2]+  Running                 sleep 30 &
```

Job control: A graphical overview




```
# After half a minute and closing emacs
[2]+  Done                    sleep 30
[1]+  Done                    emacs Loops.tex
```

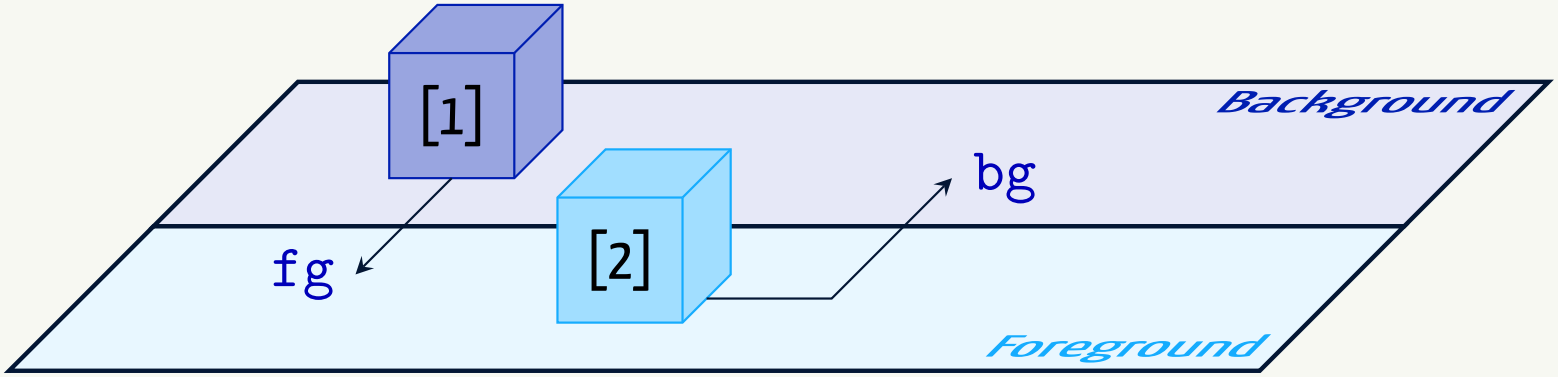
Job control: A graphical overview




```
14 # After half a minute and closing emacs
15 [2]+  Done                    sleep 30
16 [1]+  Done                    emacs Loops.tex
17 # You can start many jobs in background
18 $ for index in 30 60 90; do sleep ${index} & done && unset index
19 [1] 23559
20 [2] 23560
21 [3] 23561
22 $ jobs
23 [1]   Running                 sleep ${index} &
24 [2]-  Running                 sleep ${index} &
25 [3]+  Running                 sleep ${index} &
```

No semicolon!

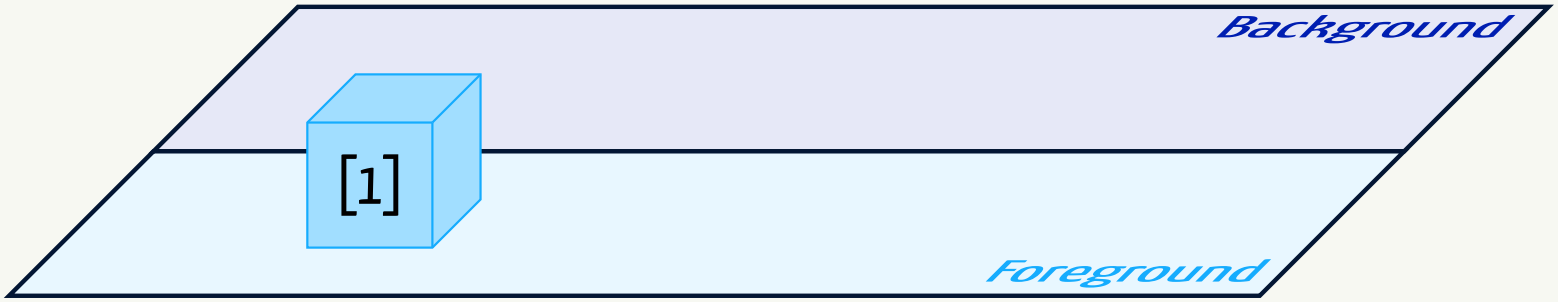
Job control: A graphical overview



- Every running process can receive signals during its execution
- Some signals are bound to common keyboard shortcuts

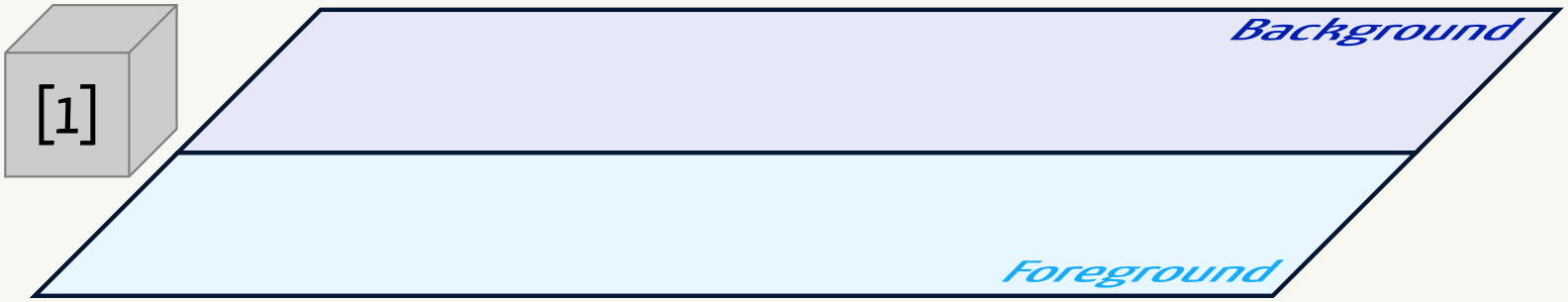
  CTRL-Z sends SIGTSTP to the foreground job { usually suspending it }
  CTRL-C sends SIGINT to the foreground job { usually terminating it }
  CTRL- sends SIGQUIT to the foreground job { usually causing it to dump core and abort }
- Jobs can be moved to background using the `bg` builtin
- Jobs can be moved to foreground using the `fg` builtin
- Any signal can be sent to a process/job using the `kill` builtin { More on it in a moment }
- The `disown` builtin make the shell "forget" a job

Job control: A graphical overview



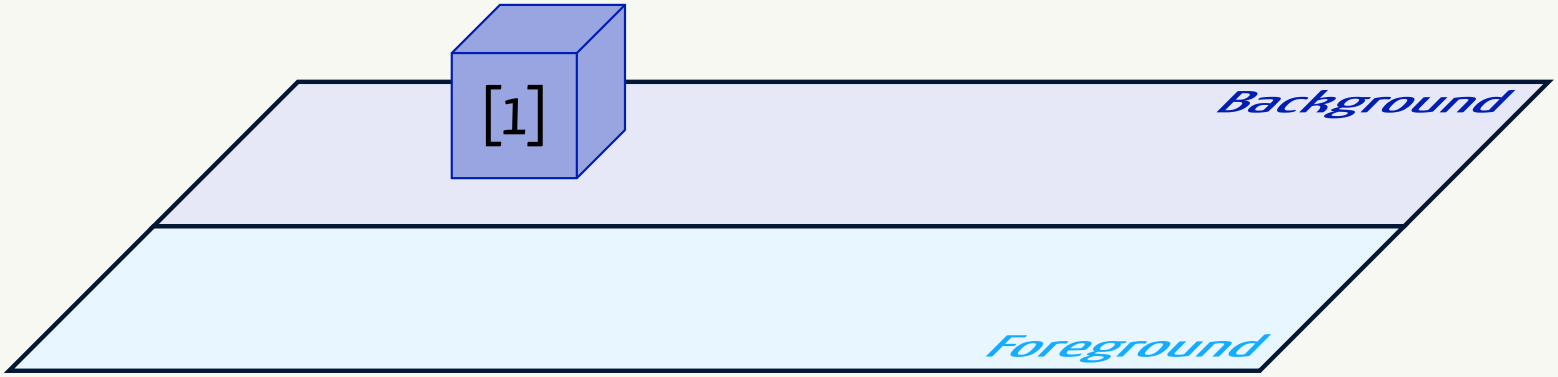

```
26 $ emacs Loops.tex
27 # Process in foreground, shell not usable
```

Job control: A graphical overview




```
$ emacs Loops.tex
# Process in foreground, shell not usable
# -> suspend via CTRL-Z
# You get your shell back, but emacs is blocked
[1]+  Stopped                 emacs Loops.tex
```

Job control: A graphical overview




```
$ emacs Loops.tex
# Process in foreground, shell not usable
# -> suspend via CTRL-Z
# You get your shell back, but emacs is blocked
[1]+  Stopped                 emacs Loops.tex
$ bg
# Alternative syntax:
#   bg %1      <- Using the jobnumber
#   bg %+      <- The last started in bg, or suspended from fg
#   bg %%      <- Equivalent to %+ (default if no job specified)
#   bg %ema    <- Job whose command begins with 'ema'
#   bg %?mac   <- Job whose command contains 'mac'
# It is an error if there is more than one matching job!
[1]+ emacs Loops.tex &
```

Handling processes in a script

Handling processes in a script

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script

- What we have discussed so far is very handy in interactive sessions
- In a script, job control is turned off by default and you will hardly need it!
- What is sometimes needed is to run processes in background
- In such a case, it is important to be aware of
  - what the `kill` builtin does
  - what is the `wait` builtin meant for
  - how to handle processes within a script → use of `$!`

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script

- What we have discussed so far is very handy in interactive sessions
- In a script, job control is turned off by default and you will hardly need it!
- What is sometimes needed is to run processes in background
- In such a case, it is important to be aware of
  - what the `kill` builtin does
  - what is the `wait` builtin meant for
  - how to handle processes within a script → use of `$!`
- Even **more important**, you have to understand the parent-child relationship!

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does → i.e. `init` will pick up orphaned processes

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does → i.e. `init` will pick up orphaned processes

System processes

A process PID will **never** be freed up for use after the process dies until the parent process waits for the PID to see whether it ended and retrieve its exit code. If the parent ends, the process is returned to `init`, which does this for you. This is important for one major reason: if the parent process manages its child process, it can be absolutely certain that, even if the child process dies, no other new process can accidentally recycle the child process PID until the parent process has waited for that PID and noticed the child died. This gives the parent process the guarantee that the PID it has for the child process will **always** point to that child process, whether it is alive or a "zombie". Nobody else has that guarantee.

Greg's Wiki

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does → i.e. `init` will pick up orphaned processes

System processes

A process PID will **never** be freed up for use after the process [illegible] until the parent process waits for the PID to see whether it ended and ret[illegible] exit [illegible] process is returned to `init`, which do[illegible] this for [illegible] parent process mana[illegible] child [illegible] process dies, no other new [illegible] parent process has waited [illegible] arent process the guarantee that the PI[illegible] oint to the child process, whether it is alive or a zombie". N[illegible] else has that guarantee.

Not true in the shell!

Greg's Wiki

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does → i.e. `init` will pick up orphaned processes

What happens in the shell

Shells aggressively reap their child processes and store the exit status in memory, where it becomes available to your script upon calling `wait`. But because the child might have already been reaped before you call wait, there is no zombie to hold the PID.

The kernel is free to reuse that PID, and your guarantee has been violated!

Greg's Wiki

We will see an explicit example in few slides

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

A bit of syntax: Retrieving a process PID

- The `$!` special parameter holds the PID of **the most recently executed background job**

```
#!/bin/bash

#...
command &
pid=$!
# ...
```

- You cannot reliably determine when or how a process was started purely from its identifier number (PID): do not make any assumption!
- As you might know, there are some metadata stored together with the PID, such as the **process name** and the **issued command** to start the process. **Never rely on those!**

Stay away from parsing the process tree!

UNIX comes with a set of handy tools, among which is `ps`. This is a very helpful utility that you can use from the command line to get an overview of what processes are running. However, `ps` output is unpredictable, highly OS-dependent, and not built for parsing!

Do not parse it in shell scripts! Ever!

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

- If a job specification is given, all processes in the job are waited for
- Supplying the `-f` option, when job control is enabled, forces `wait` to wait for each `pid` or `jobspec` to terminate before returning its status, instead of returning when it changes status

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
# In a Bash script
date
{ sleep 1; exit 1; } & { sleep 3; exit 2; } &
{ sleep 5; exit 3; } & { sleep 7; exit 4; } &
wait # All currently active child processes are waited for
date
```

```
$ ./script-above.bash
Wed  4 Sep 16:13:27 CEST 2019
Wed  4 Sep 16:13:34 CEST 2019
```


A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
# In a Bash script
date; sleep 3 & pid=$!
echo "PID=${pid}";  ps -p ${pid};  wait ${pid}; date
```

```
$ ./script-above.bash
Wed  4 Sep 16:17:19 CEST 2019
PID=11353
  PID TTY          TIME CMD
11353 pts/11   00:00:00 sleep
Wed  4 Sep 16:17:22 CEST 2019
```

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
# In a Bash script (-n option since Bash v4.3)
{ sleep 1; exit 1; } & { sleep 3; exit 2; } &
{ sleep 2; exit 3; } &
for f in {1..3}; do
   wait -n; echo "wait return status: $? at $(date +'%X')"
done
```

```
$ ./script-above.bash
wait return status: 1 at 16:36:16
wait return status: 3 at 16:36:17
wait return status: 2 at 16:36:18
```

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
$ wait 1234
bash: wait: pid 1234 is not a child of this shell
$ echo $?
127
```

A bit of syntax: The `kill` builtin

```
kill [-s sigspec] [-n signum] [-sigspec] jobspec or pid
```

- Despite its name, the `kill` command sends a signal to a process
- There are different way to specify a signal, use what you prefer
- If no signal is specified, SIGTERM (15) is sent

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$ kill -l 15
TERM
```


The `kill` builtin offers more complex functionalities, like sending a signal to a process group when the used PID is negative. Read about it in the manual.

A bit of syntax: The `kill` builtin

```
kill [-s sigspec] [-n signum] [-sigspec] jobspec or pid
```

- Despite its name, the `kill` command sends a signal to a process
- There are different way to specify a signal, use what you prefer
- If no signal is specified, SIGTERM (15) is sent

`man 2 kill`

> If signal is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

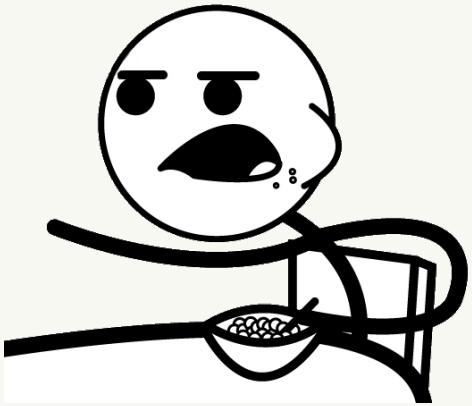
```
1 # Waiting for a process to end
2 # (when you have permissions to send signals)
3 while kill -0 "${pid}"; do
4     sleep 1
5 done
```


The `kill` builtin offers more complex functionalities, like sending a signal to a process group when the used PID is negative. Read about it in the manual.

A potentially catastrophic hidden danger

A potentially catastrophic hidden danger

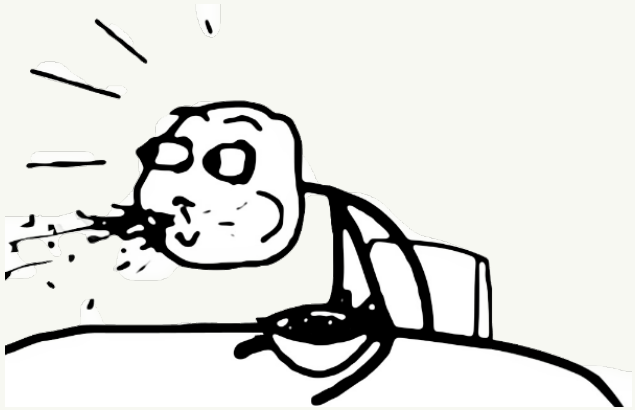
```
1  #!/bin/bash
2                                  #  EXAMPLE 1: wait
3  ( exit 12 ) &
4  pid=$!
5  while { sleep 0 & [[ "${pid}" != "$!" ]]; }; do
6      :
7  done
8  wait "$pid"
9  echo "$?"
```



A potentially catastrophic hidden danger

```
#!/bin/bash
                                #  EXAMPLE 1: wait
( exit 12 ) &
pid=$!
while { sleep 0 & [[ "${pid}" != "$!" ]]; }; do
    :
done
wait "$pid"
echo "$?"
```

```
$ ./script-above.bash
0   # Yes, 0 and not 12... WHAAAAAT?
```



A potentially catastrophic hidden danger

```
#!/bin/bash
                                #  EXAMPLE 1: wait
( exit 12 ) &
pid=$!
while { sleep 0 & [[ "${pid}" != "$!" ]]; }; do
    :
done
wait "$pid"
echo "$?"
```

```
$ ./script-above.bash
0   # Yes, 0 and not 12... WHAAAAAT?
```

Sad but true!

The way shells are programmed, as soon as a child process dies, the shell will call wait() on it right away (storing the termination status as part of its internal state), which will free the PID for reuse by another process.

A potentially catastrophic hidden danger

```
10 #!/bin/bash
11                                  #  EXAMPLE 2: kill
12 long_running_command &
13 pid=$!
14 echo "Killing long_running_command on PID ${pid} in 24h"
15 sleep 86400
16 echo 'Time up!'
17 kill "${pid}"
```

Who will receive the SIGTERM signal?!

Sad but true!

The way shells are programmed, as soon as a child process dies, the shell will call wait() on it right away (storing the termination status as part of its internal state), which will free the PID for reuse by another process.

I absolutely need to know if the PID is the right one, what should I do?

I absolutely need to know if the PID is the right one, what should I do?

Do not use Bash!

Yes, implement the code launching a background process in C or Python, Perl, Ruby, etc. **Not in shell.** Those will not have this problem, since they won't reap children by default (like the shell does) and you will have to do it explicitly there. Or consider launching background processes using a system manager, such as `systemd`.

I absolutely need to know if the PID is the right one, what should I do?

Do not use Bash!

Yes, implement the code launching a background process in C or Python, Perl, Ruby, etc. **Not in shell.** Those will not have this problem, since they won't reap children by default (like the shell does) and you will have to do it explicitly there. Or consider launching background processes using a system manager, such as `systemd`.

However

Note that the kernel will typically try hard to avoid reusing PIDs, at least try to delay reusing a PID, exactly because in some cases there are no guarantees that the PID hasn't been reused, so the kernel tries to minimise this situation where a signal will be delivered to the wrong process.

I absolutely need to know if the PID is the right one, what should I do?

Do not use Bash!

Yes, implement the code launching a background process in C or Python, Perl, Ruby, etc. **Not in shell.** Those will not have this problem, since they won't reap children by default (like the shell does) and you will have to do it explicitly there. Or consider launching background processes using a system manager, such as `systemd`.

Alternatively

You can save the start time of the original process and, before using it, check that the start time of the process with that PID matches what you saved. The pair PID, start-time is a unique identifier for the processes in Linux. However, this does not solve the issue here.

```
$ sleep 10 &
[1] 4451
$ ps -p 4451 -h -o lstart
Thu Sep  5 18:22:43 2019
```

A work-around for exit codes

- If you have several (maybe long) processes in background during the execution of a script and you are interested in their exit code, you can write them to a file
- Alternatively, a possible idea is to use redirection faking a file and taking advantage of the fact that in Bash the processes inside the process substitution are not waited for

```
#!/bin/bash
{
    command_pid=$!
    # ...
    another_command &
    # ...
    read <&3 exitCode
    if [[ "${exitCode}" -eq 0 ]]; then
        echo "command was successful!"
    fi
} 3< <(command > logfile 2>&1; echo "$?")
```

A work-around for exit codes

- If you have several (maybe long) processes in background during the execution of a script and you are interested in their exit code, you can write them to a file
- Alternatively, a possible idea is to use redirection faking a file and taking advantage of the fact that in Bash the processes inside the process substitution are not waited for

```
#!/bin/bash
{
    sleep_pid=$!
    while { ( exit 12 ) & [[ "${sleep_pid}" != "$!" ]]; }; do
        :
    done
    wait "${sleep_pid}"
    waitCode=$?
    read <&3 exitCode
    echo "Command in process substitution: ${exitCode}"
    echo "Command in           while loop: ${waitCode}"
} 3< <(sleep 1; echo "$?")
```

A work-around for exit codes

- If you have several (maybe long) processes in background during the execution of a script and you are interested in their exit code, you can write them to a file
- Alternatively, a possible idea is to use redirection faking a file and taking advantage of the fact that in Bash the processes inside the process substitution are not waited for

```
#!/bin/bash
{
    sleep_pid=$!
    while { ( exit 12 ) & [[ "${sleep_pid}" != "$!" ]]; }; do
        :
    done
    wait "${sleep_pid}"
    waitCode=$?
    read <&3 exitCode
    echo "Command in process substitution: ${exitCode}"
    echo "Command in           while loop: ${waitCode}"
} 3< <(sleep 1; echo "$?")
```

```
$ ./script-above.bash
Command in process substitution: 0
Command in           while loop: 12
```

Process Management: Conclusions

- It is a potentially tough world: Be scared but not too much
- Bash might not be the most appropriate tool
- Be aware of the truth (what we just discussed)
- If you want to have something running in background, why do you want so?
- There are tools which you might consider:
  1. `systemd` (system manager)
  2. `timeout` (run a command with a time limit)
  3. `xargs -P` or `parallel` (to run independent tasks in parallel)

Process Management: Conclusions

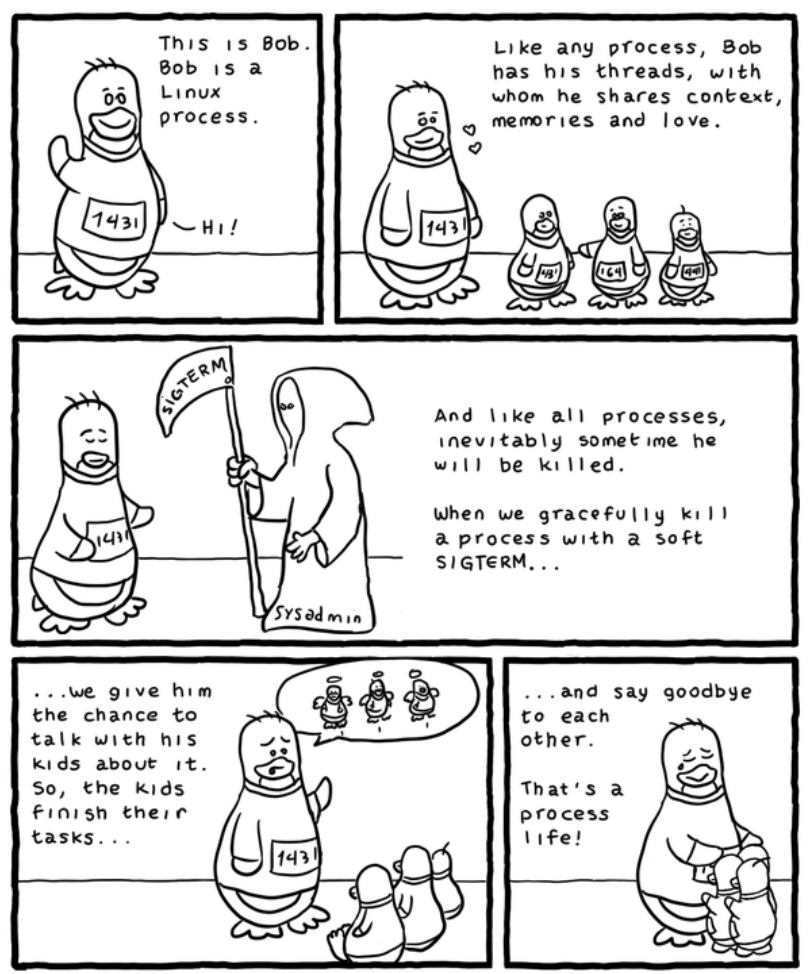
- It is a potentially tough world: Be scared but not too much
- Bash might not be the most appropriate tool
- Be aware of the truth (what we just discussed)
- If you want to have something running in background, why do you want so?
- There are tools which you might consider:
  1. `systemd` (system manager)
  2. `timeout` (run a command with a time limit)
  3. `xargs -P` or `parallel` (to run independent tasks in parallel)

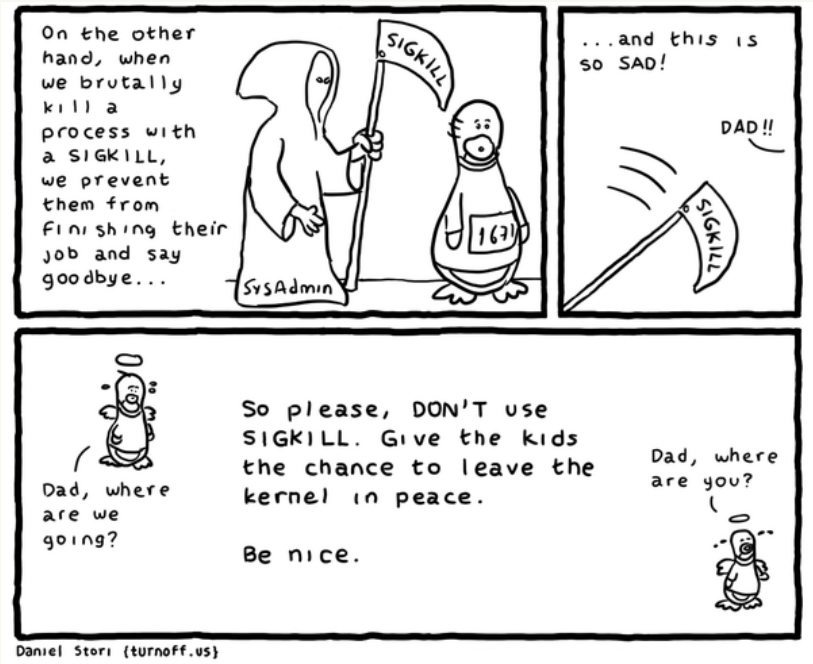
About using `kill -9`

Do not use kill -9, ever. For any reason. **Unless you wrote the program to which you're sending the SIGKILL, and know that you can clean up the mess it leaves.** Because you're debugging it. If a process is not responding to normal signals, it's probably in "state D" (as shown on `ps u`), which means it's currently executing a system call. If that's the case, you're probably looking at a dead hard drive, or a dead NFS server, or a kernel bug, or something else along those lines.
And you won't be able to kill the process anyway, SIGKILL or not.

Greg's Wiki

Process Management: Conclusions





Traps

Moulin on the Breiðamerkurjökull

Motivation

Consider the following script (not nice code):

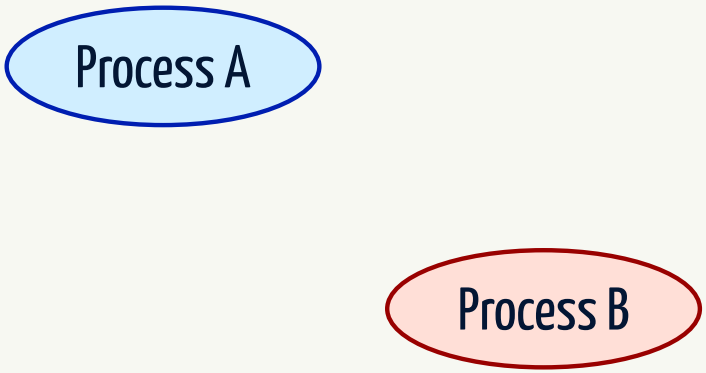
```
#!/bin/bash

function CreateAuxiliaryFiles(){
    # ...
}
function CleanAuxiliaryFiles(){
    # ...
}
CreateAuxiliaryFiles
gnuplot "${temporaryFileToPlot}"
if [[ "${savePlot}" = 'TRUE' ]]; then
    pdflatex "${outputFilename}.tex"
fi
CleanAuxiliaryFiles
exit 0
```

What happens if the script is terminated by the user, e.g. via CTRL-C?

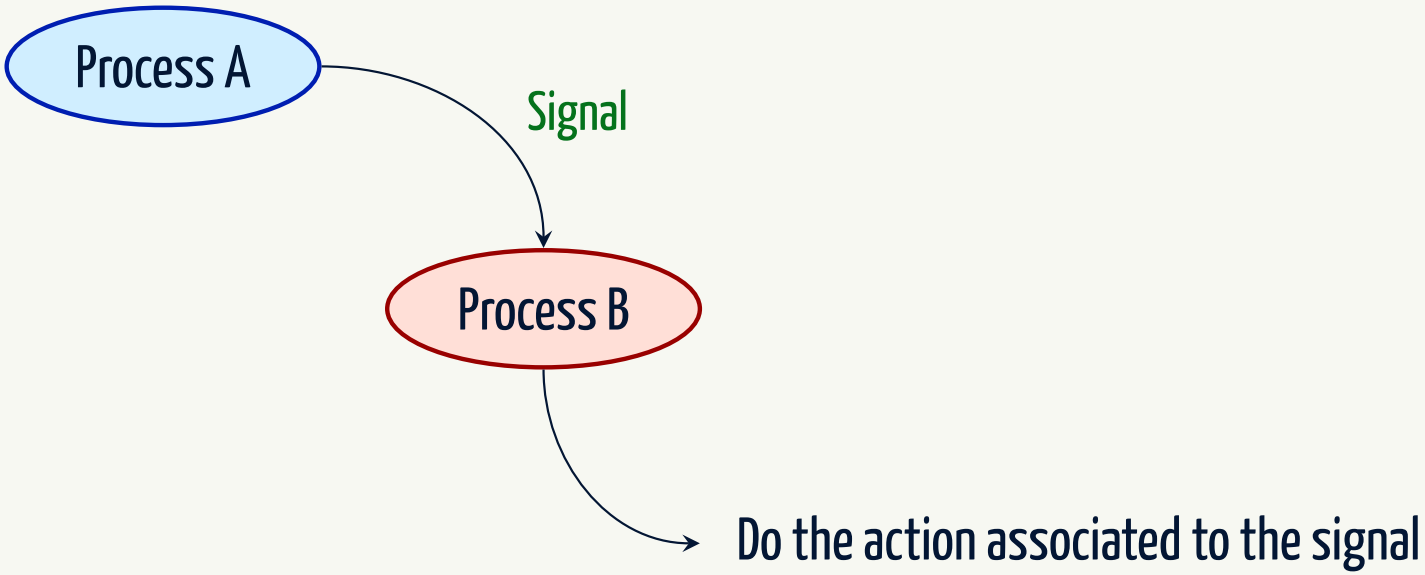
Signal handlers or traps

- As you know you can send signals to processes via the `kill` builtin
- However, in shell scripts, **it is possible to associate a behaviour upon receiving a signal!**
- If you think about it, it is a very powerful technique
- In order to perform an action when a signal is received, a trap has to be set up



Signal handlers or traps

- As you know you can send signals to processes via the `kill` builtin
- However, in shell scripts, **it is possible to associate a behaviour upon receiving a signal!**
- If you think about it, it is a very powerful technique
- In order to perform an action when a signal is received, a trap has to be set up



Signal handlers or traps

```
trap 'some code' signal(s)
```

Using this form, a signal handler is set up for each signal in the list

When one of these signals is received, the commands in the first argument will be executed

```
trap '' signal(s)
```

Using this form, each signal in the list will be ignored. Most scripts should not do this!

```
trap - signal(s)
```

Using this form, each signal in the list will be restored to its default behaviour

```
trap signal
```

Legacy syntax: As the previous but for one signal only! { Prefer the previous, slightly more explicit syntax }

```
trap
```

With no arguments, print a list of signal handlers

Common used signals

HUP Hang Up. The controlling terminal has gone away.

INT Interrupt. The user has pressed the interrupt key (usually **Ctrl-C** or **DEL**).

QUIT Quit. The user has pressed the quit key (usually **Ctrl-**). Exit and dump core.

KILL Kill. This signal cannot be caught or ignored. Unconditionally fatal. **No cleanup possible.**

TERM Terminate. This is the default signal sent by the `kill` builtin.

EXIT Not really a signal. In a Bash script (non-interactive), an EXIT trap is run on any exit, signalled or not. In other POSIX shells only when the shell process exits.

Remember

If you are asking a program to terminate, you should always use SIGTERM. This will give the program a chance to catch the signal and clean up. If you use SIGKILL, the program cannot clean up, and **may leave files in a corrupted state**.

A simple example

Common use case

Traps can be set up to intercept a fatal signal, perform cleanup, and then exit gracefully.

```
1 #!/bin/bash

2 temporaryFile=$(mktemp) || exit
3 trap 'rm -f "${temporaryFile}"' EXIT
```

Use a function whenever you need to achieve a more complex task

```
5 #!/bin/bash

6 function CleanAuxiliaryFiles(){ # If you trap INT (CTRL-C)
7     # ...                       # do not forget to exit unless
8 }                               # you have reasons not to do so
9 trap 'CleanAuxiliaryFiles' EXIT
```

Only in Bash!

The special name EXIT is preferred for any signal handler that simply wants to clean up upon exiting. So to clean up, just trap EXIT and call a cleanup function from there. **Don't trap a bunch of signals.**

When is the signal exactly handled?

A subtle but important feature

When Bash is executing an external command in the foreground, it does not handle any signals received until the foreground process terminates.

```
1 #!/bin/bash

2 echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
3 trap 'echo "Doing some cleaning before exiting!"' EXIT
4 echo "Wait 1h"
5 sleep 3600
```

- If you kill the script using `kill -s INT` from another terminal (not with CTRL-C), bash will wait for `sleep` to exit before calling the trap
  → **That's probably not what you expect!**

When is the signal exactly handled?

A subtle but important feature

When Bash is executing an external command in the foreground, it does not handle any signals received until the foreground process terminates.

```
7  #!/bin/bash
8  echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
9  trap 'echo "Doing some cleaning before exiting!"' EXIT
10 echo "Wait 1h"; sleep 3600 & wait $!
11 # Note that sleep 3600 will not be killed and will
12 # continue to run when you send a INT signal!
```

- If you kill the script using `kill -s INT` from another terminal (not with CTRL-C), bash will wait for `sleep` to exit before calling the trap
  → **That's probably not what you expect!**
- A work-around is to use a **builtin** that will be interrupted, such as `wait`
- Any bash internal command will be interrupted by a (non-ignored) incoming signal!

When is the signal exactly handled?

A subtle but important feature

When Bash is executing an external command in the foreground, it does not handle any signals received until the foreground process terminates.

```
14 #!/bin/bash

15 echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
16 unset -v pid
17 trap 'echo "Cleaning!"; [[ $pid ]] && kill "${pid}"' EXIT
18 echo "Wait 1h"
19 sleep 3600 & pid=$!; wait
```

- If you kill the script using `kill -s INT` from another terminal (not with CTRL-C), bash will wait for `sleep` to exit before calling the trap
  → **That's probably not what you expect!**
- A work-around is to use a **builtin** that will be interrupted, such as `wait`
- Any bash internal command will be interrupted by a (non-ignored) incoming signal!
- If you want the background job to be killed when the script is killed, add that to the trap!

What indeed is CTRL-C doing?

Let us consider again the previous example

```
#!/bin/bash

echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
trap 'echo "Doing some cleaning before exiting!"' EXIT
echo "Wait 1h"; sleep 3600
```

- As said, sending to this script the INT signal from another terminal does not terminate it
- However, CTRL-C does terminate it... but why?

Note that you can't rely on the process group ID of a script to be the same as $$, as that depends greatly on how the script was started.

What indeed is CTRL-C doing?

Let us consider again the previous example

```
#!/bin/bash

echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
trap 'echo "Doing some cleaning before exiting!"' EXIT
echo "Wait 1h"; sleep 3600
```

- As said, sending to this script the INT signal from another terminal does not terminate it
- However, CTRL-C does terminate it... but why?
- Because **processes are organised in groups**:
  - The leader process (i.e. the process that created the group)
  - Any other process started by the leader process
- **CTRL-C sends the INT signal to the entire process group**
- Terminals keep track of the foreground process group: When receiving a CTRL-C, they send the SIGINT to the entire foreground group!

Note that you can't rely on the process group ID of a script to be the same as $$, as that depends greatly on how the script was started.

What indeed is CTRL-C doing?

Let us consider again the previous example

```
#!/bin/bash

echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
trap 'echo "Doing some cleaning before exiting!"' EXIT
echo "Wait 1h"; sleep 3600
```

- As said, sending to this script the INT signal from another terminal does not terminate it
- However, CTRL-C does terminate it... but why?
- Because **processes are organised in groups**:
  - The leader process (i.e. the process that created the group)
  - Any other process started by the leader process
- **CTRL-C sends the INT signal to the entire process group**
- Terminals keep track of the foreground process group: When receiving a CTRL-C, they send the SIGINT to the entire foreground group!

```
kill -s INT -123 # will kill the process group with the ID 123
```


Note that you can't rely on the process group ID of a script to be the same as $$, as that depends greatly on how the script was started.

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT
- The idea is that, if your script calls `vi` for instance, and you press CTRL-C within `vi` to cancel an action, that should not be considered as a request to abort the script

What does it mean?

Imagine you're writing a script and that script exits normally upon receiving SIGINT.
That means that if that script is invoked from another bash script,
CTRL-C will no longer interrupt that other script!

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT

The `ping` command returns with 0 when host is reachable (the ping has been answered) and non-zero otherwise when interrupted (which is the only way for ping to return in that case).

```
1 #!/bin/bash
2 for index in {1..254}; do
3     ping -c 2 "192.168.1.${index}"
4 done
```

CTRL-C will send a INT signal to the script, but since `ping` just returns 0 or 1 and is not killed by SIGINT, only the **current** `ping` will terminate and the `for` loop will continue!

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT

Commands that don't have a SIGINT handler (like `sleep`) or **do the right thing of killing themselves with SIGINT upon receiving SIGINT** (like `bash` itself does) don't exhibit the problem!

```
5  #!/bin/bash
6  index=1
7  while [[ "${index}" -le 100 ]]; do
8    printf "%d " "$((index++))"
9    sleep 1
10 done; echo  # This script terminates correctly via CTRL-C
```

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT

Take-home lesson

If you choose to set up a handler for SIGINT (rather than using the EXIT trap), you should be aware that **a process that exits in response to SIGINT should kill itself with SIGINT rather than simply exiting**, to avoid causing problems for its caller. The same goes for SIGQUIT.

```
trap 'DoYourStuffHere; trap - INT; kill -s INT "$$"' INT
```

Error handling

How the Breiðamerkurjökull shrank from 2017 to 2018

Error handling in the shell

Naturally an extremely difficult task

The goal of automatic error detection is a noble one, but it requires the ability to tell when an error actually occurred. In modern high-level languages, most tasks are performed by using the language's builtin commands or features. The language knows whether (for example) you tried to divide by zero, or open a file that you can't open, and so on. It can take action based on this knowledge.

But in the shell, most of the tasks you actually care about are done by external programs. The shell can't tell whether an external program encountered something that it considers an error – and even if it could, it wouldn't know whether the error is an important one, worthy of aborting the entire program, or whether it should carry on.

Greg's Wiki

- If you come from a language like C, you might be used to it.
- If you come from a language like Python or C++, do not be frustrated.

What does Bash do when an error occurs?

I.e. when a command has a non-zero exit code!

The default behaviour

An error is printed and execution continues.

```
#!/bin/bash

# A test.bash script
echo "Before error"
notExistingCommand
echo "After error"   # <-- This line is exectued!
```

```
$ ./test.bash
Before error
./test.bash: line 5: notExistingCommand: command not found
After error
```

What does Bash do when an error occurs?

I.e. when a command has a non-zero exit code!

The default behaviour

An error is printed and execution continues.

Do not forget it!

Things that we consider an error are not necessarily an error for bash!

```
$ invisibleVariable="I am magic"
$ echo "_${invisibelVariable}_"
__
```

What does Bash do when an error occurs?

I.e. when a command has a non-zero exit code!

The default behaviour

An error is printed and execution continues.

Do not forget it!

Things that we consider an error are not necessarily an error for bash!

We clearly want more than this!

1. We often need to react on error and avoid execution to continue.
2. It might be crucial to prevent undefined variables to be used!

```
# About 1: Somewhere in a script
cd NotExistingFolder
rm -r *    # AAAAAAARGH!
```

What does Bash do when an error occurs?

I.e. when a command has a non-zero exit code!

The default behaviour

An error is printed and execution continues.

Do not forget it!

Things that we consider an error are not necessarily an error for bash!

We clearly want more than this!

1. We often need to react on error and avoid execution to continue.
2. It might be crucial to prevent undefined variables to be used!

```
#!/bin/bash
# About 2: And if you call this script without arguments!?
pathPrefix="$1"
rm -r "${pathPrefix}/bin"    # AAAAAAARGH!
```

Exit codes are the only available ingredient

...the rest is basically up to you!

Purist approach

Check and handle errors properly testing exit codes **when needed**.

- `$?` holds the exit code of the last command executed before any given line.
- Any executed command will terminate with an exit code that can be inspected.
- You can expect external command exit codes to be well documented.

```
# From 'man grep'
...
EXIT STATUS
   Normally the exit status is 0 if a line is selected,
   1 if no lines were selected, and 2 if an error occurred.
   However, if the -q or --quiet or --silent is used and a
   line is selected, the exit status is 0 even if an
   error occurred.
...
```

Exit codes are the only available ingredient

...the rest is basically up to you!

Purist approach

Check and handle errors properly testing exit codes **when needed**.

- `$?` holds the exit code of the last command executed before any given line.
- Any executed command will terminate with an exit code that can be inspected.
- You can expect external command exit codes to be well documented.

```
# In a script at some point
grep 'Hello' MyBook.txt
case $? in
    0 ) echo 'Found' ;;
    1 ) echo 'Not found' ;;
    2 ) echo 'Error occurred in grep!'; exit 1 ;;
    * ) echo 'Matrix changed something...'; exit 2 ;;
esac
```

Exit codes are the only available ingredient

...the rest is basically up to you!

Purist approach

Check and handle errors properly testing exit codes **when needed**.

- `$?` holds the exit code of the last command executed before any given line.
- Any executed command will terminate with an exit code that can be inspected.
- You can expect external command exit codes to be well documented.

```
# Coming back to the previous example
cd NotExistingFolder || exit 1  # Exit on failure!
rm -r *                         # Now better!
```

```
# Alternatively not exiting
if cd NotExistingFolder; then
    rm -r *
fi
```

Of course, this is just an example, you would rather write `rm NotExistingFolder/*` , right!?

Do I have to add `|| exit 1` everywhere in my scripts?

Do I have to add `|| exit 1` everywhere in my scripts?

Purist approach

Check and handle errors properly testing exit codes **when needed**.

Do I have to add `|| exit 1` everywhere in my scripts?

Purist approach

Check and handle errors properly testing exit codes **when needed**.

> The developers of the original Bourne shell decided that they would create a feature that would allow the shell to check the exit status of every command that it runs, and abort if one of them returns non-zero. Thus, `set -e` was born. But many commands return non-zero even when there wasn't an error. [...] So the shell implementers made a bunch of special rules, like "commands that are part of an if test are immune", and "commands in a pipeline, other than the last one, are immune".
>
> Greg's Wiki

The `set -o errexit` is probably the most controversial feature of bash!

- Many articles are against using it, e.g. 🔗 Greg's Wiki .
- People encourage its use, e.g. 🔗 David Pashley .

Both references are definitely worth reading, especially if you decide at the end to use `set -e` in your scripts.

Enabling the `-e` shell option: Let's read the manual

- Exit immediately if a pipeline, which may consist of a single simple command, a list, or a compound command returns a non-zero status.
- The shell does not exit if the command that fails
  - is part of the command list immediately following a `while` or `until` keyword,
  - is part of the test in an `if` statement,
  - is part of any command executed in a `&&` or `||` list except the command following the final `&&` or `||`,
  - is any command in a pipeline but the last,

  or if the command's return status is being inverted with `!`.
- If a compound command other than a subshell returns a non-zero status because a command failed while `-e` was being ignored, the shell does not exit.
- A trap on `ERR`, if set, is executed before the shell exits.
- This option applies to the shell environment and each subshell environment separately, and may cause subshells to exit before executing all the commands in the subshell.

And few more rules are there about when a compound command or shell function executes (or sets `-e`) in a context where `-e` is being ignored...

Enabling the `-e` shell option: Let's read the manual

Your reaction might be like:

Enabling the `-e` shell option: Let's read the manual

Your reaction might be like:

But what does it mean, in practice?

- You need to **deeply** understand this shell option
- Your way of programming has to adapt to all corner cases
- Checking and storing error codes via `$?` is not possible as before

But what does it mean, in practice?

- You need to **deeply** understand this shell option
- Your way of programming has to adapt to all corner cases
- Checking and storing error codes via `$?` is not possible as before

```
1 #!/usr/bin/env bash
2 set -e
3 i=0
4 ((i++))           # ((...)) is a compound command!
5 echo "i is $i"    # This is NOT printed
```

The postfix increment operator returns the old value of the variable.

```
1 #!/usr/bin/env bash
2 set -e
3 i=1
4 ((i++))           # ((...)) is a compound command!
5 echo "i is $i"    # This is here printed
```

But what does it mean, in practice?

- You need to **deeply** understand this shell option
- Your way of programming has to adapt to all corner cases
- Checking and storing error codes via `$?` is not possible as before

```
1  #!/usr/bin/env bash
2  set -e
3  i=0
4  ((i++)) || true
5  echo "i is $i"   # This is ALWASY printed

6  i=$((i++))       # Simple command without command subst.
7  echo "i is $i"   # This is ALWASY printed

8  i=$(false)       # Simple command with failing $(...)
9  echo "i is $i"   # This is NOT printed
```

Simple command expansion

> [...] If one of the expansions contained a command substitution, the exit status of the command is the exit status of the last command substitution performed. If there were no command substitutions, the command exits with a status of zero.

— Bash manual

But what does it mean, in practice?

- You need to **deeply** understand this shell option
- Your way of programming has to adapt to all corner cases
- Checking and storing error codes via `$?` is not possible as before

```
1  #!/usr/bin/env bash
2  set -e
3
3  function Failure() { false; }
4  function FailureMsg() {
5      false
6      echo "Failed!" >&2
7  }
8
8  ( false )           # Subshell -> new environment, -e unset
9
9  echo 'Begin'
10 aVar=$(FailureMsg) # No failure because of echo in function
11 echo 'Middle'
12 aVar=$(Failure)    # Function fails
13 echo "End"         # This is NOT printed
```

But what does it mean, in practice?

- You need to **deeply** understand this shell option
- Your way of programming has to adapt to all corner cases
- Checking and storing error codes via `$?` is not possible as before

```
1  #!/usr/bin/env bash
2  set -e
3  function Simulation(){ return 1; }
4
4  if Simulation; then
5      echo 'Success!'
6  else
7      echo 'Failure!'
8  fi
9
9  Simulation                  # Here the script exits!
10
10 if [[ $? -eq 0 ]]; then     # This if-clause is skipped!
11     echo 'Success!'
12 else
13     echo 'Failure!'
14 fi
```

But what does it mean, in practice?

- You need to **deeply** understand this shell option
- Your way of programming has to adapt to all corner cases
- Checking and storing error codes via `$?` is not possible as before

Spend some time on it

This afternoon you will deal again in detail with some corner cases!

And if I do not want to use it?

Do I have other alternatives beyond the purist approach and enabling `errexit`?

Not really...

Consider to enable `inherit_errexit` shell option

It was introduced in Bash v4.4

> If set, command substitution inherits the value of the errexit option, instead of unsetting it in the subshell environment. This option is enabled when POSIX mode is enabled.
>
> — Bash manual

```
1  #!/usr/bin/env bash
2  set -e
3  shopt -s inherit_errexit
4
4  function Failure() { false; }
5  function FailureMsg() {
6      false
7      echo "Failed!" >&2
8  }
9
9  echo 'Begin'
10 aVar=$(FailureMsg) # Now the script exits here before echo
11 echo 'Middle'
12 aVar=$(Failure)    # Function fails
13 echo "End"         # This is NOT printed
```

Trap on ERR and the `errtrace` shell option

The ERR signal specification

The trap on `ERR` is executed exactly when the bash would exit with the `errexit` option enabled. Said differently, the same conditions obeyed by the `errexit` option apply to traps on `ERR`.

The `errtrace` shell option

If set, any trap on *ERR* is inherited by shell functions, command substitutions, and commands executed in a subshell environment. The *ERR* trap is normally not inherited in such cases.

Bash manual

`set -E` in combination with *ERR* traps is a bit like an improved version of *`set -e`* in that it allows you to define your own error handling.

Stéphane Chazelas

Use the `functrace` shell option to inherit any trap on DEBUG and RETURN in shell functions, command substitutions, and subshells.

Trap on ERR and the `errtrace` shell option

```
1  #!/usr/bin/env bash
2  set -E
3  trap '[ "$?" -ne 77 ] || exit 77' ERR
4  (
5    echo 'Here'
6    (
7      echo 'There'
8      ( exit 12 )       # not 77, exit only this subshell
9      echo 'Somewhere'
10     exit 77           # exit all subshells
11   )
12   echo 'Not here'
13 )
14 echo 'Not here either'
```

```
$ ./previous-script.bash
Here
There
Somewhere
```


Use the `functrace` shell option to inherit any trap on DEBUG and RETURN in shell functions, command substitutions, and subshells.

Enabling the `-u` shell option

`set -o nounset`

> Treat unset variables and parameters other than the special parameters `@` or `*` as an error when performing parameter expansion. An error message will be written to the standard error, and a non-interactive shell will exit.
>
> — Bash manual

```
$ echo "_${unsetVariable}_"

__
$ set -u; echo "_${unsetVariable}_"    # A script would
bash: unsetVariable: unbound variable # exit here!
$ echo "_${unsetVariable-}_"; set +u

__
```

Few (minor) drawbacks exist

1. Turning `nounset` on in a correct existing script might still break it.
2. If some code is not executed, `set -u` will have no effect on it either.
3. Versions of bash prior to v4.4 🔗 handle the `[@]`-expansion of empty arrays differently.

On Greg's Wiki you can 🔗 read more about `set -u`.

Is the glass half full or half empty?

A reasonable person might say:

I don't want to rely on an unreliable feature. I understood the behaviour of the shell when an error occurs and I can do error handling myself.

`set -o pipefail` is also another interesting feature with minor drawbacks.

Is the glass half full or half empty?

A reasonable person might say:

I don't want to rely on an unreliable feature. I understood the behaviour of the shell when an error occurs and I can do error handling myself.

Another reasonable person might say:

Yes, *`errexit`* is not a perfect feature of bash, but it has useful semantics, so I don't want to exclude it from the toolbox. Still I have to work a bit to get acquainted with all corner cases, but I think it is worth the effort. Especially because I do not want to be bitten by mistakes in doing error handling by hand. Hopefully, future versions of bash will improve this aspect.

`set -o pipefail` is also another interesting feature with minor drawbacks.

Is the glass half full or half empty?

A reasonable person might say:

I don't want to rely on an unreliable feature. I understood the behaviour of the shell when an error occurs and I can do error handling myself.

Another reasonable person might say:

Yes, *`errexit`* is not a perfect feature of bash, but it has useful semantics, so I don't want to exclude it from the toolbox. Still I have to work a bit to get acquainted with all corner cases, but I think it is worth the effort. Especially because I do not want to be bitten by mistakes in doing error handling by hand. Hopefully, future versions of bash will improve this aspect.

Choose your way

Probably consistency within a script is the most important aspect. My recommendation is to explore this area of bash in all ways before taking your decision.

`set -o pipefail` is also another interesting feature with minor drawbacks.

Sed (and Awk)

A double wave

Why do we need them?

- In science, most of our time in the terminal is spent acting on files
- Mastering file handling in general can allow us to simplify the data analysis software
- Typical operations might be
  - Prepare data for plotting in a given format
  - Search patterns in a file
  - Extract portion of a file
  - Transform a file (e.g. remove trailing spaces)
  - ...

GNU Core Utilities

`head`	`cat`	`sum`	`column`	`sort`	`join`	`expand`
`tail`	`tac`	`cksum`	`paste`	`uniq`	`comm`	`unexpand`
	`cut`	`md5sum`			`tr`	`split`

A complete GNU coreutils list is available on Wikipedia.

Why do we need them?

- In science, most of our time in the terminal is spent acting on files
- Mastering file handling in general can allow us to simplify the data analysis software
- Typical operations might be
  - Prepare data for plotting in a given format
  - Search patterns in a file
  - Extract portion of a file
  - Transform a file (e.g. remove trailing spaces)
  - ...

GNU Core Utilities

head cat column sort expand
tail paste uniq expand
split

sed

awk

A complete GNU coreutils list is available on Wikipedia.

Sed: A marvellous utility

The awful truth about `sed`

Sed is extremely powerful. Unfortunately, most people never learn its real power. The language is simpler than the average user thinks, because the documentation is far from being ideal. Sed has several commands, but most people only learn the substitute command `'s'`. And the GNU manual begins exactly with examples about the `'s'` command.

Not bad!

The substitute command is just one command. Sed has at least 20 different commands for you. You can even write Tetris in it (not to mention that it's Turing complete).

Some references

- The official GNU manual { The PDF is 85 pages long... }
- A very nice tutorial { Bash code there often uses deprecated features, but you should know it by now! }
- One-liners, with some comments
- More one-liners, explained but advanced!

The PDF manual is 85 pages long. What should I learn here?

- We will focus on the abstract idea of **how** `sed` processes a file
- Some of the simplest commands (acting on a single line) will be introduced
- Learning Sed is almost like learning a new programming language, mastering it is probably not needed for a physicist either

The PDF manual is 85 pages long. What should I learn here?

- We will focus on the abstract idea of **how** `sed` processes a file
- Some of the simplest commands (acting on a single line) will be introduced
- Learning Sed is almost like learning a new programming language, mastering it is probably not needed for a physicist either

```
# Invocation
sed [OPTIONS...] [SCRIPT] [INPUTFILE...]

# General command syntax to be put in [SCRIPT]
[addr]X[options]
```

The PDF manual is 85 pages long. What should I learn here?

- We will focus on the abstract idea of **how** `sed` processes a file
- Some of the simplest commands (acting on a single line) will be introduced
- Learning Sed is almost like learning a new programming language, mastering it is probably not needed for a physicist either

```
# Invocation
sed [OPTIONS...] [SCRIPT] [INPUTFILE...]

# General command syntax to be put in [SCRIPT]
[addr]X[options]
```

- `X` is a `sed` command
- `[addr]` is an optional line address
  If `[addr]` is specified, the command `X` will be executed only on the matched lines;
  `[addr]` can be a single line number, a regular expression, or a range of lines
- Additional `[options]` are used for some `sed` commands
- Quote the commands to avoid shell globbing conflicts
  → usually you want to use single quotes

Some of the most used commands

s/regexp/replacement/[flags]

Match the regular-expression against the content of the pattern space.
If found, replace matched string with replacement.

p

Print the pattern space, up to the first newline.

d

Delete the pattern space; immediately start next cycle.

=

Print the current input line number (with a trailing newline).

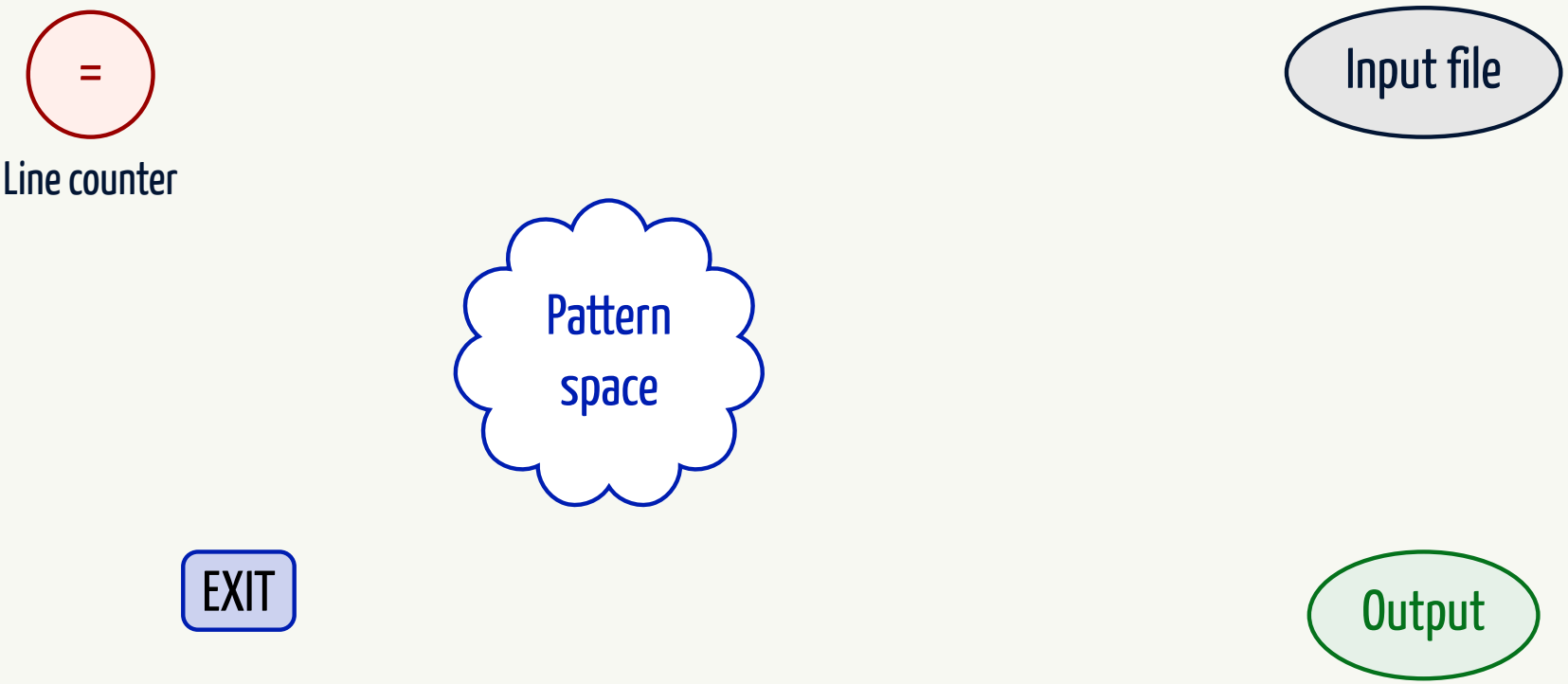
{ cmd ; cmd ...}

Group several commands together.

q

Exit `sed` without processing any more commands or input.

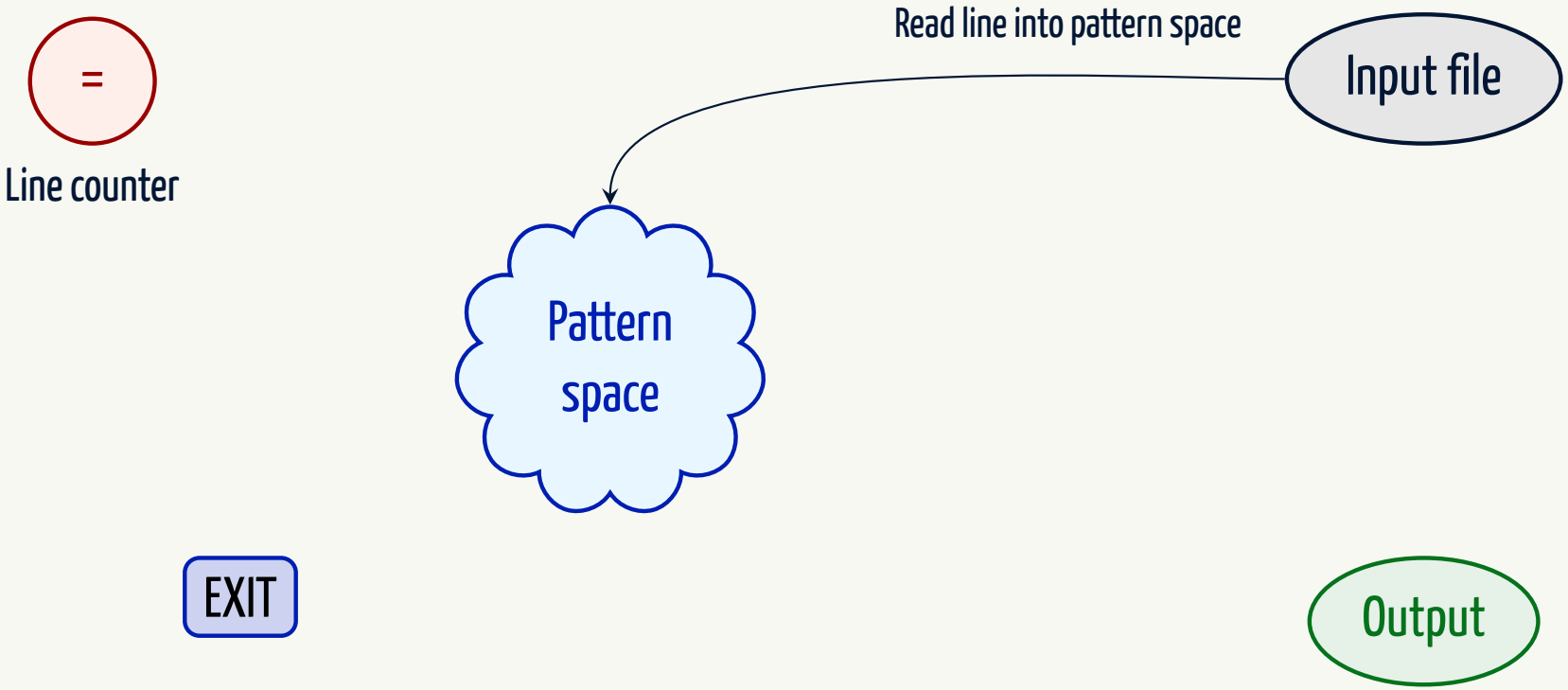
A `sed` cycle: Being line oriented



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

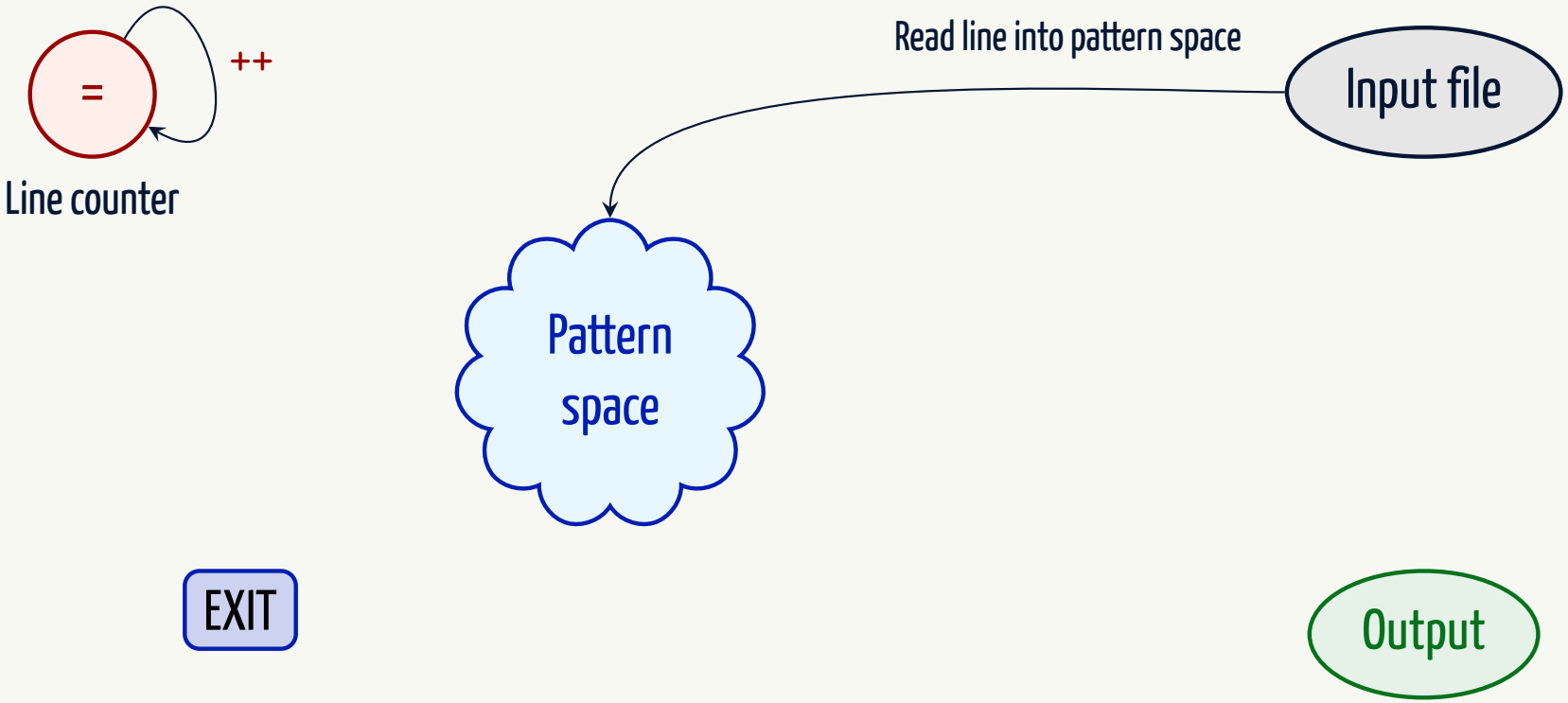
1. The next line is read from the input file and placed it in the pattern space.
   If the end of file is found, and if there are additional files to read, the current file is closed, the next file is opened, and the first line of the new file is placed into the pattern space.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

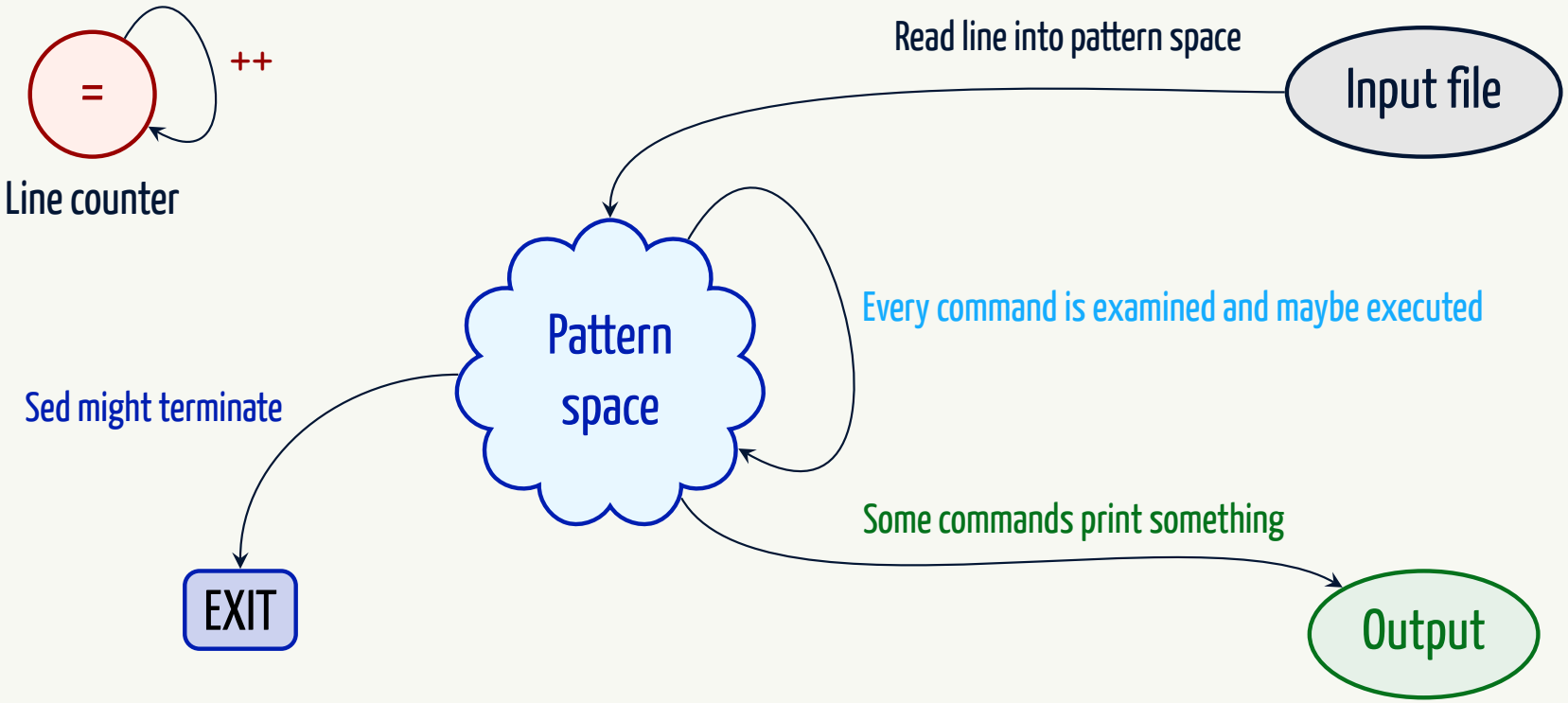
2 The line count is incremented by one.
Opening a new file does not reset this number.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

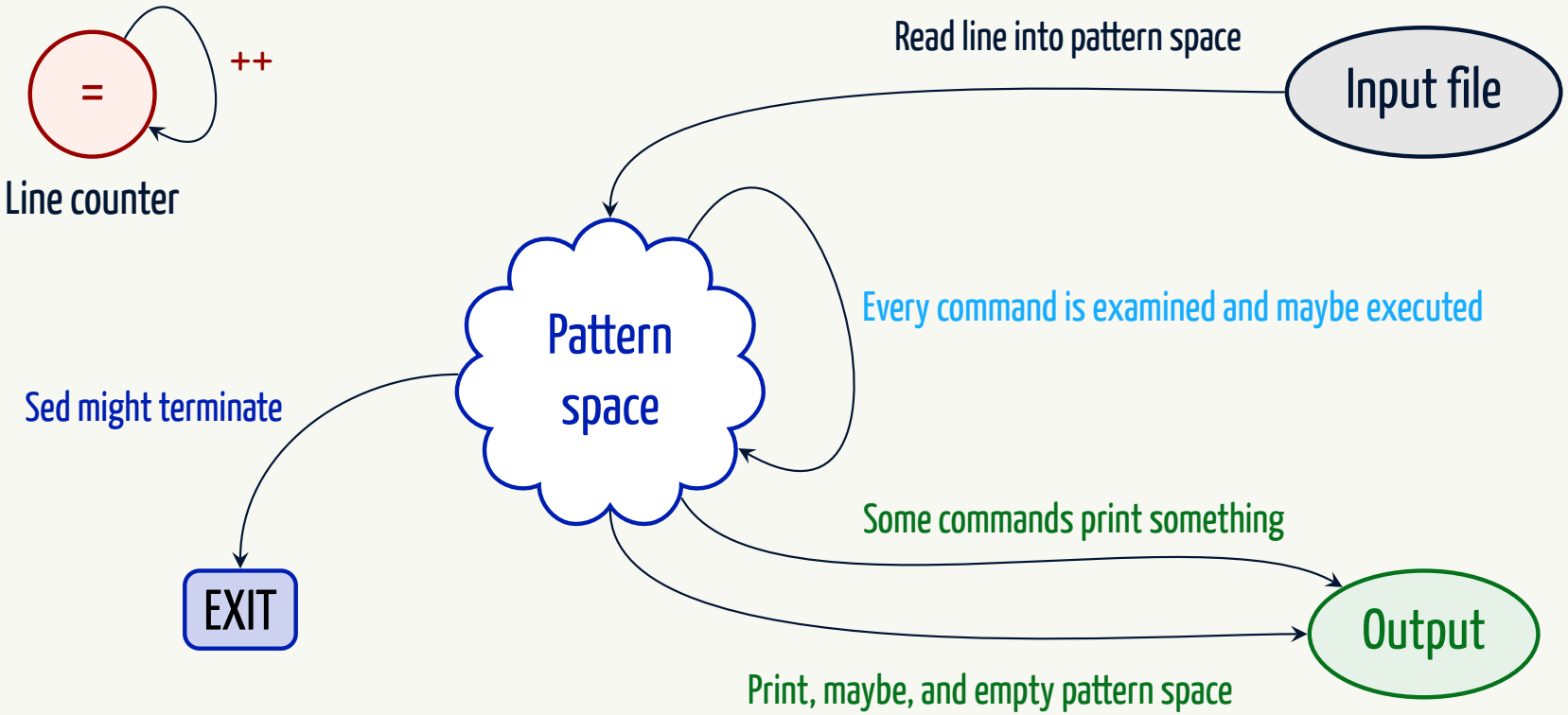
3. Each `sed` command is examined. If there is a restriction placed on the command, and the current line in the pattern space meets that restriction, the command is executed. Some commands, like `'d'` cause `sed` to go to the top of the loop. The `'q'` command causes `sed` to stop. Otherwise the next command is examined.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

4 After all of the commands are examined, the pattern space is printed to the output (unless sed has the optional "-n" argument) and the pattern space is emptied.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

Address restrictions

A command can be limited to some lines using restrictions

- A positive line number (use $ to refer to the last line)
- A regular expression `/regex/` or `\•regex•` where • is any character { not to be matched! }
- Use an exclamation mark after the address to negate that address
- A comma separated pair to specify a range of lines

```
                    # Command applies
3                   #  - to line 3 only
/^#/                #  - to lines starting by #
\%^//%              #  - to lines starting by //
5!                  #  - not to line 5
1,3                 #  - to first three lines
3,$                 #  - from line 3 to the end
1,/one/             #  - till the first line matching 'one'
/begin/,/end/       #  - from the first line matching 'begin'
                    #    to the first matching 'end' (included)
```

The main commands: Substitute

`s/regexp/replacement/[flags]`

The `'s'` command attempts to match the pattern space against the supplied regular expression `regexp`; if the match is successful, then that portion of the pattern space which was matched is replaced with `replacement`.

- The `/` is called delimiter and can replaced with any character
- The delimiter can appear in `regexp` or `replacement` only if escaped
- Unescaped `&` characters reference the whole matched portion of the pattern space
- The replacement can contain `\1`, ..., `\9` references, which refer to the portion of the match which is contained between the first, ..., nineth `\(` and its matching `\)`
- Important-to-know flags are

`g`	Apply the replacement to all matches to the `regexp`, not just the first
number	Only replace the number-th match of the `regexp` (between 1 and 512)
`p`	If the substitution was made, then print the new pattern space

The main commands: Substitute

```
$ printf '11\n12\n' | sed 's/12/twelve/'
11
twelve
$ printf '11\n12\n' | sed 's/1/X/'
X1
X2
$ printf '11\n12\n' | sed 's/1/X/g'
XX
X2
$ printf '11\n12\n' | sed 's/1/X/2'
1X
12
$ printf '11\n12\n' | sed 's/2/Y/p'
11
1Y
1Y
$ sed 's/[0-9]\+/(&)/' <<< 'Wed 09 Sep 2019'
Wed (09) Sep 2019
$ sed 's/[0-9]\+/(&)/g' <<< 'Wed 09 Sep 2019'
Wed (09) Sep (2019)
$ sed -r 's/[0-9]+/(&)/g' <<< 'Wed 09 Sep 2019'
Wed (09) Sep (2019) # -r option for ERE
```

The main commands: Substitute

```
$ printf '11\n12\n' | sed 's/1/&&/'
111
112
$ sed 's/\([a-z]\+\) \([a-z]\+\)/\2 \1/' <<< 'hello world'
world hello
$ sed 's/^\(.\)\(.\)\(.\)\(.\) /\3\2\1\4/' <<< 'nice star'
cinestar
$ printf "=%0.s" {1..20} | sed 's/./&:/10'; echo
==========:==========
$ cd /usr/local/bin
$ sed 's/\/usr\/local\/bin/\/common\/bin/' <<< "${PWD}/emacs"
/common/bin/emacs
$ sed 's_/usr/local/bin_/common/bin_' <<< "${PWD}/emacs"
/common/bin/emacs
$ sed 's:/usr/local/bin:/common/bin:' <<< "${PWD}/emacs"
/common/bin/emacs
$ sed 's|/usr/local/bin|/common/bin|' <<< "${PWD}/emacs"
/common/bin/emacs
$ value='3 5'; sed 's/Y/'"${value}"'/' <<< "XYZ"
X3 5Z
$ value='3 5'; sed 's/Y/'${value}'/' <<< "XYZ"
sed: -e expression #1, char 5: unterminated `s' command
```

The main commands: Print

- Print out the pattern space
- Note that the printing induced by the `p` command is unrelated to the printing done by `sed` at the end of the cycle
- Hence, this command is usually only used in conjunction with the `-n` command-line option of `sed` (whose long name is `--quiet` or `--silent`)
- It is commonly used together with an address restriction

```
$ sed 'p' <<< 'Echo me'
Echo me
Echo me
$ sed -n 'p' <<< 'Only once'
Only once
$ printf '%d\n' {1..9} | sed -n '5 p'
5
$ printf '%d\n' {1..9} | sed -n '7,$ p'
7
8
9    # Try out: printf '%d\n' {1..10} | sed -n '1~3 p'
```

The main commands: Delete

- Delete the pattern space and immediately start next cycle
- It is used to delete lines from the input file(s)
- It is commonly used together with an address restriction

```
$ printf '%d\n' {1..10} | sed '3,$ d'
1
2
$ sed '2 d' <<< $'A\n nice but\n cold place'
A
 cold place
$ sed '/b/,/f/ d' <<< $'a\nb\nc\nd\ne\nf\ng'
a
g
$ sed '/^#/ d' filename  # Remove bash comments
# ... <- look up option -i of sed to edit the file in place
$ sed '/^$/ d' filename  # Remove empty lines
```

The main commands: quit, line number and groups

- The `'q'` command exits `sed` without processing any more commands or input. This command exits at the end of the cycle, i.e. it prints the current pattern space.
- The `'='` command prints out the current input line number (with a trailing newline).
- A group of commands may be enclosed between `{` and `}` characters. This is particularly useful when you want a group of commands to be triggered by a single address (or address-range) match.

```
$ printf '%d\n' {1..15} | sed '2q'
1
2
$ sed '=' <<< $'aaa\nbbb'
1
aaa
2
bbb
$ sed -n '$=' filename  # Equivalent to 'wc -l < filename'
$ printf '%d\n' {1..3} | sed -n '2{s/2/X/ ; p}'
X
```