Use Bash Strict Mode (Unless You Love Debugging)

Let's start with the punchline. Your bash scripts will be more robust, reliable and maintainable if you start them like this:

```
#!/bin/bash
set -euo pipefail
IFS=$'\n\t'
```

I call this the unofficial **bash strict mode**. This causes bash to behave in a way that makes many classes of subtle bugs impossible. You'll spend much less time debugging, and also avoid having unexpected complications in production.

There is a short-term downside: these settings make certain common bash idioms harder to work with. Most have simple workarounds, detailed below: jump to Issues & Solutions. But first, let's look at what these obscure lines actually do.

The "set" lines

These lines deliberately cause your script to fail. Wait, what? Believe me, this is a good thing. With these settings, certain common errors will cause the script to *immediately* fail, explicitly and loudly. Otherwise, you can get hidden bugs that are discovered only when they blow up in production.

`set -euo pipefail` is short for:

```
set -e
set -u
set -o pipefail
```

Let's look at each separately.

set -e

The `set -e` option instructs bash to immediately exit if any command [1] has a non-zero exit status. You wouldn't want to set this for your command-line shell, but in a script it's massively helpful. In all widely used general-purpose programming languages, an unhandled runtime error - whether that's a thrown exception in Java, or a segmentation fault in C, or a syntax error in Python - immediately halts execution of the program; subsequent lines are not executed.

By default, bash does **not** do this. This default behavior is exactly what you want if you are using bash on the command line - you don't want a typo to log you out! But in a script, you really want the opposite. If one line in a script fails, but the last line succeeds, the whole script has a successful exit code. That makes it very easy to miss the error.

Again, what you want when using bash as your command-line shell and using it in scripts are at odds here. Being intolerant of errors is a lot better in scripts, and that's what `set -e` gives you.

set -u

`set -u` affects variables. When set, a reference to any variable you haven't previously defined - with the exceptions of `$*` and `$@` - is an error, and causes the program to immediately exit. Languages like Python, C, Java and more all behave the same way, for

all sorts of good reasons. One is so typos don't create new variables without you realizing it. For example:

```
#!/bin/bash
firstName="Aaron"
fullName="$firstname Maxwell"
echo "$fullName"
```

Take a moment and look. Do you see the error? The right-hand side of the third line says "firstname", all lowercase, instead of the camel-cased "firstName". Without the -u option, this will be a silent error. But with the -u option, the script exits on that line with an exit code of 1, printing the message "firstname: unbound variable" to stderr. This is what you want: have it fail explicitly and immediately, rather than create subtle bugs that may be discovered too late.

set -o pipefail

This setting prevents errors in a pipeline from being masked. If any command in a pipeline fails, that return code will be used as the return code of the whole pipeline. By default, the pipeline's return code is that of the last command - even if it succeeds. Imagine finding a sorted list of matching lines in a file:

```
% grep some-string /non/existent/file | sort
grep: /non/existent/file: No such file or directory
% echo $?
0
```

(`%` is the bash prompt.) Here, grep has an exit code of 2, writes an error message to stderr, and an empty string to stdout. This empty string is then passed through sort, which happily accepts it as valid input, and returns a status code of 0. This is fine for a command line, but bad for a shell script: you almost certainly want the script to exit right then with a nonzero exit code... like this:

```
% set -o pipefail
% grep some-string /non/existent/file | sort
grep: /non/existent/file: No such file or directory
% echo $?
2
```

Setting IFS

The IFS variable - which stands for **I**nternal **F**ield **S**eparator - controls what Bash calls *word splitting*. When set to a string, each character in the string is considered by Bash to separate words. This governs how bash will iterate through a sequence. For example, this script:

```
#!/bin/bash
IFS=$' '
items="a b c"
for x in $items; do
    echo "$x"
done

IFS=$'\n'
for y in $items; do
    echo "$y"
done
```

... will print out this:

```
a
b
c
a b c
```

In the first for loop, IFS is set to `$' '`. (The `$'...'` syntax creates a string, with backslash-escaped characters replaced with special characters - like "\t" for tab and "\n" for newline.) Within the for loops, x and y are set to whatever bash considers a "word" in the original sequence. For the first loop, IFS is a space, meaning that words

are separated by a space character. For the second loop, "words" are separated by a *newline*, which means bash considers the whole value of "items" as a single word. If IFS is more than one character, splitting will be done on *any* of those characters.

Got all that? The next question is, why are we setting IFS to a string consisting of a tab character and a newline? Because it gives us better behavior when iterating over a loop. By "better", I mean "much less likely to cause surprising and confusing bugs". This is apparent in working with bash arrays:

```
#!/bin/bash
names=(
  "Aaron Maxwell"
  "Wayne Gretzky"
  "David Beckham"
  "Anderson da Silva"
)

echo "With default IFS value..."
for name in ${names[@]}; do
  echo "$name"
done

echo ""
echo "With strict-mode IFS value..."
IFS=$'\n\t'
for name in ${names[@]}; do
  echo "$name"
done
```

(Yes, I'm putting my name on a list of great athletes. Indulge me.) This is the output:

```
With default IFS value...
Aaron
Maxwell
Wayne
Gretzky
David
Beckham
Anderson
da
Silva

With strict-mode IFS value...
Aaron Maxwell
Wayne Gretzky
David Beckham
Anderson da Silva
```

Or consider a script that takes filenames as command line arguments:

```
for arg in $@; do
    echo "doing something with file: $arg"
done
```

If you invoke this as `myscript.sh notes todo-list 'My Resume.doc'`, then with the default IFS value, the third argument will be mis-parsed as two separate files - named "My" and "Resume.doc". When actually it's a file that has a space in it, named "My Resume.doc".

Which behavior is more generally useful? The second, of course - where we have the ability to not split on spaces. If we have an array of strings that in general contain spaces, we normally want to iterate through them item by item, and not split an individual item into several.

Setting IFS to `$'\n\t'` means that word splitting will happen only on newlines and tab characters. This very often produces useful splitting behavior. By default, bash sets this to `$' \n\t'` - space, newline, tab - which is too eager. [2]

Issues & Solutions

I've been using the unofficial bash strict mode for years. At this point, it always immediately saves me time and debugging headaches. But at first it was challenging, because many of my usual habits and idioms didn't work under under these settings. The rest of this article catalogues some problems you may encounter, and how to quickly work around them.

(If you run into a problem you don't see here, email me and I will attempt to help.)

Sourcing A Nonconforming Document

Sometimes your script needs to source a file that doesn't work with your strict mode. What then?

```
source some/bad/file.env
# Your strict-mode script immediately exits here,
# with a fatal error.
```

The solution is to (a) temporarily disable that aspect of strict mode; (b) source the document; then (c) re-enable, on the next line.

The most common time you'll need this will be when the document references an undefined variable. Temporarily allow such a transgression with `set +u`:

```
set +u
source some/bad/file.env
set -u
```

(Remember, `set +u` *disables* this variable strictness, and `set -u` *enables* it. A bit counterintuitive, so take care here until it's second nature.)

You used to need this with Python virtual environments. If you are not familiar with Python: you can set up a custom, isolated environment - called a virtualenv - stored in a directory, named something like "venv". You elect to use this by sourcing a file named "bin/activate" within:

```
# This will update PATH and set PYTHONPATH to
# use the preconfigured virtual environment.
source /path/to/venv/bin/activate

# Now the desired version of Python is in your path,
# with the specific set of libraries you need.
python my_program.py
```

In modern versions of Python, this works great with bash strict mode. But virtual environments that are a bit older - yet young enough you might still run into them - do *not* work correctly with the -u option:

```
set -u
source /path/to/venv/bin/activate
_OLD_VIRTUAL_PYTHONHOME: unbound variable
# This causes your strict-mode script to exit with an error.
```

No problem, you just use the pattern above:

```
set +u
source /path/to/venv/bin/activate
set -u
```

In my experience, sourced documents rarely need `-e` or `-o pipefail` to be disabled. But if you ever encounter that, you deal with it the same way.

Positional Parameters

The `-u` setting causes the script to immediately exit if any undefined variable references are made, except for `$*` or `$@`. But what if your script takes positional arguments - `$1`, `$2`, etc. - and you want to verify that they were supplied? Consider this script named `sayhello.sh`:

If you run "sayhello.sh" by itself, here's what happens:

```
% ./sayhello.sh
./sayhello.sh: line 3: $1: unbound variable
```

A most unhelpful error message. The solution is to use parameter default values (http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion). The idea is that if a reference is made *at runtime* to an undefined variable, bash has a syntax for declaring a default value, using the ":-" operator:

```
# Variable $foo has not been set. In strict mode,
# next line triggers an error.
bar=$foo

# ${VARNAME:-DEFAULT_VALUE} evals to DEFAULT_VALUE if VARNAME undefined.
# So here, $bar is set to "alpha":
bar=${foo:-alpha}

# Now we set foo explicitly:
foo="beta"

# ... and the default value is ignored. Here $bar is set to "beta":
bar=${foo:-alpha}

# To make the default an empty string, use ${VARNAME:-}
empty_string=${some_undefined_var:-}
```

Under strict mode, you need to use this for all positional parameter references:

```
#!/bin/bash
set -u
name=${1:-}
if [[ -z "$name" ]]; then
    echo "usage: $0 NAME"
    exit 1
fi
echo "Hello, $name"
```

Intentionally Undefined Variables

In the default mode, a reference to an undefined variable evaluates to an empty string - a behavior that is sometimes relied on. That's not an option in strict mode, and there are a couple of ways to deal with it. I think the best one is to explicitly set the variable to an empty string early on in the script, before any reference to it:

```
someVar=""
# ...
# a bunch of lines of code that may or may not set someVar
# ...
if [[ -z "$someVar" ]]; then
# ...
```

An alternative could be to use the `${someVar:-}` syntax for default values, described under Positional Parameters. The problem there is that it's possible to forget and just say `$someVar`, plus it's more typing. Just set the default value explicitly at the top of your script. Then there's no way this can bite you.

Commands You Expect To Have Nonzero Exit Status

What happens when you *want* to run a command that will fail, or which you know will have a nonzero exit code? You don't want it to stop your script, because that's actually correct behavior.

There are two choices here. The simplest, which you will usually want to use, is to append "`|| true`" after the command:

```
# "grep -c" reports the number of matching lines. If the number is 0,
# then grep's exit status is 1, but we don't care - we just want to
# know the number of matches, even if that number is zero.

# Under strict mode, the next line aborts with an error:
count=$(grep -c some-string some-file)

# But this one behaves more nicely:
count=$(grep -c some-string some-file || true)

echo "count: $count"
```

This short-circuiting with the boolean operator makes the expression inside `$( ... )` always evaluate successfully.

You will probably find that trick almost always solves your problem. But what if you want to know the return value of a command, even if that return value is nonzero? Then you can temporarily disable the exit-immediately option:

```
# We had started out this script with set -e . And then...

set +e
count=$(grep -c some-string some-file)
retval=$?
set -e

# grep's return code is 0 when one or more lines match;
# 1 if no lines match; and 2 on an error. This pattern
# lets us distinguish between them.

echo "return value: $retval"
echo "count: $count"
```

Essential Clean-up

Suppose your script is structured like:

1. Spin up some expensive resource

2. Do something with it
3. Release that resource so it doesn't keep running and generate a giant bill

For "expensive resource", this can be something like an EC2 instance that costs you real money. Or it could be something much smaller - like a scratch directory - that you want to create for the script to use, then be sure to delete once it is complete (so it doesn't leak storage, etc.) With the `set -e` option, it is possible an error will cause your script to exit before it can perform the cleanup, which is not acceptable.

The solution: use bash exit traps (http://redsymbol.net/articles/bash-exit-traps/). The linked article explains this important pattern in detail, and I highly recommend you master the technique - having it in your toolbox significantly makes your scripts more robust and reliable. In brief, you define a bash function that performs the clean-up or release of resources, and then register the function to be automatically invoked on exit. Here's how you would use it to robustly clean up a scratch directory:

```
scratch=$(mktemp -d -t tmp.XXXXXXXXXX)
function finish {
  rm -rf "$scratch"
}
trap finish EXIT

# Now your script can write files in the directory "$scratch".
# It will automatically be deleted on exit, whether that's due
# to an error, or normal completion.
```

Short-Circuiting Considerations

The whole point of strict mode is to convert many kinds of hidden, intermittent, or subtle bugs into immediate, glaringly obvious errors. Strict mode creates some special concerns with short-circuiting, however. By "short-circuiting", I mean chaining together several commands with `&&` or `||` - for example:

```
# Prints a message only if $somefile exists.
[[ -f "$somefile" ]] && echo "Found file: $somefile"
```

The first short-circuit issue can happen when chaining three or more commands in a row:

```
first_task && second_task && third_task
# And more lines of code following:
next_task
```

The potential problem: if `second_task` fails, `third_task` will not run, and execution will continue to the next line of code - `next_task`, in this example. This may be exactly the behavior you want. Alternatively, you may be intending that if `second_task` fails, the script should immediately exit with its error code. In this case, the best choice is to use a block - i.e., curly braces:

```
first_task && {
    second_task
    third_task
}
next_task
```

Because we are using the `-e` option, if `second_task` fails, the script immediately exits.

The second issue is truly devious. It can sneak up when using this common idiom:

```
# COND && COMMAND
[[ -f "$somefile" ]] && echo "Found file: $somefile"
```

When people write `COND && COMMAND`, typically they mean "if COND succeeds (or is boolean true), then execute COMMAND. Regardless, proceed to the next line of the script." It's a very convenient shorthand for a full "if/then/fi" clause. But when such a construct is the *last line* of the file, strict mode can give your script a surprising exit code:

```
% cat short-circuit-last-line.sh
#!/bin/bash
set -euo pipefail
# omitting some lines of code...

# Prints a message only if $somefile exists.
# Note structure: COND && COMMAND
[[ -f "$somefile" ]] && echo "Found file: $somefile"

% ./short-circuit-last-line.sh
% echo $?
1
```

When the script got to the last line, `$somefile` did *not* in fact exist. Thus `COND` evaluated false, and `COMMAND` was not executed - which is what should happen. But the script exits with a non-zero exit code, which is a bug: the script in fact executed correctly, so it really should exit with 0. In fact, if the last line of code is something else, that's exactly what we get:

```
% cat short-circuit-before-last-line.sh
#!/bin/bash
set -euo pipefail
# omitting some lines of code...

# Prints a message only if $somefile exists.
# Structure: COND && COMMAND
# (When we run this, $somefile will again not exist,
# so COMMAND will not run.)
[[ -f "$somefile" ]] && echo "Found file: $somefile"

echo "Done."

% ./short-circuit-before-last-line.sh
Done.
% echo $?
0
```

What's going on? It turns out the `-e` option has a special exception in short-circuiting expressions like this: if `COND` evaluates as false, `COMMAND` will not run, and flow of execution will proceed to the next line. However, the result of the entire line - the entire short-circuit expression - will be nonzero, because `COND` was. And being the last line of the script, that becomes the program's exit code.

This is the kind of bug we don't want to have, since it can be subtle, non-obvious, and hard to reproduce. And it's mainly difficult to deal with because it shows up *only* if it's the last command of the file; on any other line, it is well-behaved and causes no problem. It's easy to forget this in normal, day-to-day development, and have it slip through the cracks. For example, what if you delete an innocuous-looking echo statement from the end, making a short-circuit line now be last?

In the specific example above, we can expand the expression in a full "if" clause. This is perfectly well-behaved:

```
# Prints a message only if $somefile exists.
if [[ -f "$somefile" ]]; then
    echo "Found file: $somefile"
fi

# If COND is a command or program, it works the same. This:
first_task && second_task

# ... becomes this:
if first_task; then
     second_task
fi
```

What's the final, full solution? You could decide to trust yourself and your teammates to always remember this special case. You can all freely use short-circuiting, but simply don't allow a short-circuit expression to be on the last line of a script, for anything actually deployed. This may work 100% reliably for you and your team, but I don't believe that is the case for myself and many other developers. Of course, some kind of linter or commit hook might help.

Perhaps a safer option is to decide never to use short-circuiting at all, and always use full if-statements instead. This may be unappealing, though: short-circuiting is convenient, and people like to do it for a reason. For now, I'm still seeking a more satisfactory solution. Please contact me if you have a suggestion.

Feedback / If You Get Stuck

If you have feedback or suggestions for improvements, I'd love to hear it. Reach me (Aaron Maxwell) by email, at amax at redsymbol dot net.

Conversely, if you find strict mode causes a problem that I don't tell you how to solve above, I want to know about that too. When you email me, please include a **minimal** bash script that demonstrates the problem in the body of the email (not as an attachment). Also very clearly state what the desired output or effect should be, and what error or failure you are getting instead. You are much more likely to get a response if your script isn't some giant monster with obtuse identifiers that I would have to spend all afternoon parsing.

Footnotes

- [1] Specifically, if any pipeline; any command in parentheses; or a command executed as part of a command list in braces exits with a non-zero exit status, the script exits immediately with that same status. There are other subtleties to this; see docs for the bash "set" builtin (http://www.gnu.org/software/bash/manual/bashref.html#The-Set-Builtin) for details.
- [2] Another approach: instead of altering IFS, begin the loop with `for arg in "$@"` - double quoting the iteration variable. This changes loop semantics to produces the nicer behavior, and even handles a few edge cases better. The big problem is maintainability. It's easy for even experienced developers to forget to put in the double quotes. Even if the original author has managed to impeccably ingrain the habit, it's foolish to expect that all future maintainers will. In short, relying on quotes has a high risk of introducing subtle time-bomb bugs. Setting IFS renders this impossible.