AI Starter Guide

Concepts • Tactics • Ideas

Edited by Joel Parker Henderson

Version 1.2.0

Contents

Aim for POSIX

Summary: Use POSIX when possible because of portability and standardization.

Why use POSIX instead of bash, zsh, fish, etc.?

- We often write scripts that are intended to run a wide range of systems, including many operating systems, many older systems, many third-party systems, and some embedded systems.
- We value portability and standardization more that using any special bash features; if we need special bash features, this is when we switch to a higher-level language such as python, perl, ruby, go, rust, etc.
- We use the POSIX standard to bring clarity to teamwork, because we have the POSIX standard documentation; we have not found a bash standard yet.
- We like the zsh shell; in practice we find it's just-different-enough that it takes more time to script.
- We love the fish shell; in practice we find it's too-different for heterogeneous teams to use quickly.
- Counterpoint: some companies, such as Google, choose to explicity write for the bash shell, and explicity reject the POSIX standard; this is good for Google because Google controls all its systems, and provides a specific version of bash, so the Google engineers know the target system specifics.

When to use a higher-level language?

- If we need to use arrays for anything more than assignment of ${PIPESTATUS}, we use a higher-level language.
- If we write a script that is more than a page or so of lines long, then we favor changing from shell to a higher-level language.

Protect scripts by using `set` flags

For POSIX scripts, please start with:

```
set -euf
```

For bash/zsh scripts, please start with:

```
set -euf -o pipefail
```

Meanings:

- `set -e` or `set -o errexit`: if there is an error, then exit immediately.
- `set -u` or `set -o nounset`: disallow unset variables.
- `set -f` or `set -o noglob`: diasble filename globbing.
- `set -o pipefail`: if a pipe fails, then exit immediately and use the pipe's exit status.

Note: some scripts won't be able use all these flags. For example, a script that needs filename globbing cannot use `set -f` because it disables globbing. Please note this in the code immediately above the `set` line.

For Bash help on the `set` command see: https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html

Advice

Some programmers debate the usefulness of these flags, because they cause immediate exits, which isn't always what the programmer expects or wants.

A more-sophisticated approach is write proper error handing for each function.

Print output with `printf` not `echo`

Print output with `printf` not `echo` because of security, consistency across systems, and stability.

The `printf` command does everything the `echo` command does, and more.

Right:

```
printf "%s\n" "foo"
```

Wrong:

```
echo "foo"
```

Some reasons:

- Echo isn't consistent across operating systems.
- Echo output can confused with echo flags.

Trap signals and exit

Trap signals, such as user pressing CTRL-C, then call an exiting function.

```
trap_exit() {
  # Do any cleanup here, such as closing open files,
  # deleting temp data, or printing results.
}

trap trap_exit EXIT
```

Use `EXIT` instead of TERM, INT, HUP, etc.

Other trap ideas

We favor the naming convention of trap_ such as:

```
trap_exit() { ... }
trap_term() { ... }
trap_int() { ... }
trap_hup() { ... }
```

The naming convention helps make it clear that the functions are related to the trap command and the specific signal.

Run executable with no file name extension

Run executable with no file name extension.

Yes:

```
$ foo
```

No:

```
$ foo.sh
$ foo.bash
$ foo.zsh…
```

Executables should omit file name extensions, such as `.sh` or `.bash` or `.zsh`.

This is because an executable file name extension is unnecessary, and also easy accidentally set incorrectly. For example, we have some customers with scripts that show the extension `.sh` yet the scripts require functionality of `bash`.

Shebang

To ensure which shell is needed, write the first line of the script as a shebang such as these below.

For a script that can run on any POSIX current shell:

```
#!/usr/bin/env sh
```

If your script must use the `bash` shell then use:

```
#!/usr/bin/env bash
```

If your script must use the `zsh` shell then use:

```
#!/usr/bin/env zsh
```

Exceptions

If you know what you're doing, then there are some rare times you may want to specify a file name extension, or specify a shell using shebang that isn't an actual script requirement.

For example, we have some customers who want to have shell-specific scripts, such as for optimizations, or deployments to different systems with different default shells. For example, one customer has `foo.sh` and `foo.bash` and `foo.zsh` all in the same directory; the deployment process chooses which script is deployed to which systems.

Source with dot operator `.` not keyword `source`

Source a file by using the dot operator `.` not the word `source`

The operator reads and executes commands from a filename argument in the current shell context.

- Using the dot operator `.` is POSIX compliant.
- Using the keyword `source` isn't POSIX compliant.

Yes:

```
. foo
```

No:

```
source foo
```

for arg do loop

See
http://gnu.ist.utl.pt/software/autoconf/manual/autoconf-2.57/html_chapter/autocon

To loop over positional arguments, use:

```
for arg
do
  printf %s\\n "$arg"
done
```

DANGER: You must not leave the do on the same line as for, such as in the example caode below. This is because some shells improperly grok it.

```
## DANGER
for arg; do
  printf %s\\n "$arg"
done
```

DANGER: If you want to explicitly refer to the positional arguments, given the `$@` bug, then you can use the example code below. But keep in mind that Zsh, even in Bourne shell emulation mode, performs word splitting on `${1+"$@"}`; see Shell Substitutions, item '$@', for more.

```
## DANGER
for arg in ${1+"$@"}; do
  printf %s\\n "$arg"
done
```

Parse options via `while` and `case` not `getopts` or `getopt`

Unix shell scripts have a variety of ways to do options parsing such as:

- `while` and `case` which you write yourself.
- `getopts` is a POSIX standard. See https://pubs.opengroup.org/onlinepubs/7908799/xcu/getopts.html.
- `getopts_long` is a POSIX script. See http://stchaz.free.fr/getopts_long.sh.
- `getopt` is a GNU tool. See https://www.gnu.org/software/libc/manual/html_node/Getopt.html.
- `zparseopts` is a Z shell tool. See the man page `zshmodules(1)`.

There are pros and cons to each.

If you want simple parsing, then we recommend writing your own via `while` and `case`.

If you want sophisticated parsing, then we recommend changing from a Unix shell script to a more-capable programming language.

`while` and `case`

Use a `while` infinite loop that contains a `case` statement for all the command line options.

Example:

```
## Set verbose level to default 0
verbose=0

## Set any options to defaults
foo=""

## Process all the command line options
while :; do
```

```
    case $1 in
        # Two hyphens ends the options parsing
        --)
            shift
            break
            ;;
        # Print help by calling a help() function that you define later
        -h|--help)
            help
            exit
            ;;
        # Each verbose option is treated as incrementing the verbose leve
        -v|--verbose)
            verbose=$((verbose + 1))
            ;;
        # Parse an option value via this syntax: --foo bar
        -f|--foo)
            if [ -n "$2" ]; then
                foo=$2
                shift
            else
                die "The command option --foo requires a value"
            fi
            ;;
        # Parse an option value via this syntax: --foo=bar
        --foo=?*)
            foo=${1#*=}
            ;;
        # Parse an option value via this syntax: --foo= (i.e. blank)
        --foo=)
            die "The command option --foo requires a value"
            ;;
        # Anything remaining that starts with a dash triggers a fatal erro
        -?*)
            die "The command line option is unknown: " "$1"
            ;;
        # Anything remaining is treated as content not a parseable option
        *)
            break
            ;;
    esac
    shift
done

help(){
cat << EOF
    # add your help documentation here
EOF
}

die(){
    >&2 printf %s\\n "$*"
    exit 1
}
```

getops

The POSIX `getopts` approach can be great for combining one-letter options, such as the classic `ps -aux`.

Unfortunately POSIX `getopts` doesn't support long options, such as –help. This means a script can't use long options that are descriptive such as –dry-run, –danger, –minimum, –maximum, etc.

For some teams, and some commands, the long options are worth the tradeoff because they improve readability, as well as flexibility when a programmer is creating a script's option names. For a majority of our use cases, we value the long options more highly than the short options combining letters, and we're willing to write the parsing ourselves.

getops versus getopt

In practice there are significant differences between `getopts` and `getopt` and their various versions on various systems:

- The `getopts` POSIX standard can parse a short option, but not a long option, whereas `getopt`can do both. Example: "-h" is a short option, and "–help" is a long option.
- The `getopt` tool has had parsing issues when there is an empty

argument string. Example: "–foo=bar" succeeds, whereas "–foo=" fails even though it could mean that foo should be set to a blank string.

- Different capabilities can happen depending on the specific system. Example: some systems do not come with `getopt`, whereas some systems do come with `getopt` yet there are different versions with different capabilties due to use of different package sources such as `util-linux` versus `linux-utils`.

getopts_long

This is written as a POSIX shell function that we embed within a shell script.

When you want the best capabilties for options parsing, and you do not want to change from a Unix shell script to a more-capable programming language, then try `getopts_long`.

zparseopts

This is an approach that is specific to Z shell (zsh).

When you have a specification that states the shell script will exclusively run on Z shell, then try `zparseopts`.

Version name: use semantic versioning

We prefer to give each script a version name a.k.a. version number, and we prefer to use semantic versioning: http://semver.org/

The important aspect to our teams is that documenting the version name makes it easier for our teammates to understand when to upgrade to a newer version, and easier to document features and issues.

We prefer to create a function `version()` such as:

```
version() { printf %s "1.0.0"; }
```

We work with some organizations that prefer to use a variable such as:

```
version="1.0.0"
```

We work with some organizations that prefer to use a comment such as:

```
## version 1.0.0
```

Help: use a function and HERE document

Provide help using a function and HERE document syntax such as:

```
help(){
cat << HERE
This is my script.

This is a good place for an explanation of the syntax,
and examples, and explanations of any options, etc.
HERE
}
```

To show the help, one way is like this:

```
if [ "$#" -ge 1 ]; then
    case "$1" in
        -h|--help)
            help; exit 0
            ;;
    esac
fi
```

Markdown

We prefer help text to use markdown formatting.

Examples:

- For code, indent 4 spaces.
- For options, use a bullet list: indent 2 spaces and use an asterisk, and use a blank line between each list item.
- For a section headline, start a line with two hash signs “##”.

Syntax and example

If a script is a typical command, then we prefer to show the syntax and an example like this:

```
Syntax:

    foo <text> [...]

Example:

    $ foo hello

    $ foo hello world
```

Note:

- We prefer using a "$" dollar sign to indicate a typical command line prompt.

Options

If a script has command line options, then we prefer to show them using a bullet list, like this:

```
Options:

    * -h --help:
    show this help information

    * -v --version:
    show the command name, version number, and updated date.
```

Tracking

If a script is intended for the public, then we prefer to show tracking metadata using a bullet list, like this:

```
Tracking:
    * Command: my-foo-script
    * Version: 1.0.0
    * Updated: 2018-01-01
    * License: GPL
    * Website: https://example.com/example/
    * Contact: Alice Adams (alice@example.com)
```

Tracking varaibles

If a script is intended for the public, and we want it to be professional, then we prefer to set tracking variables and print the variables in the help, like this:

```
program_command="my-foo-script"
program_version="1.0.0"
program_created="2016-01-01"
program_updated="2016-01-11"
program_license="GPL-2.0 or GPL-3.0"
program_website="https://example.com"
program_contact="Alice Adams (alice@example.com)"

help(){
cat << EOF
...

### Tracking

    * Command: $program_command
    * Version: $program_version
    * Created: $program_created
    * Updated: $program_updated
    * License: $program_license
    * Website: $program_website
    * Contact: $program_contact

EOF
}
```

To show the help, we parse the command line options; one way is this simple code:

```
if [ "$#" -ge 1 ]; then
    case "$1" in
        -h|--help)
            help; exit 0
            ;;
        -v|--version)
            out "$program_version"; exit 0
            ;;
        --program-command)
            out "$program_command"; exit 0
            ;;
        --program-version)
            out "$program_version"; exit 0
            ;;
        --program-updated)
            out "$program_version"; exit 0
            ;;
        --program-license)
            out "$program_license"; exit 0
            ;;
        --program-website)
            out "$program_website"; exit 0
            ;;
        --program-contact)
            out "$program_contact"; exit 0
            ;;
    esac
fi
```

Date & time format: use UTC and ISO8601

For date values and time values, we prefer:

- UTC (Universal Coordinated Time, a.k.a. Greenwich Mean Time, Zulu Time Zone, etc.).
- ISO8601 standard format.
- Sort-friendly ordering such as YYYY-MM-DD and HH:MM:SS and +00:00.

Format:

- Date format: “YYYY-MM-DD”.
- Time format: “HH:MM:SS.NNNNNNNNN”.
- Timezone format: “+HH:MM”.

Examples:

- Date: 2016-12-31
- Date with weekly format: 2016-W52-7
- Time: 12:59:59
- Time with nanoseconds: 12:59:59.123456789
- Timezone: +00:00
- Timestamps: 2016-12-31T12:59:59.123456789+00:00

Notes:

- For the time, we use nanoseconds because we want times to be compatible with high-precision systems.
- For the time nanoseconds, note that as of this writing, the date command on the BSD operating system does not have a nanoseconds option. Consider installling GNU date.

- For the time nanoseconds decimal, we use a period, rather than a comma, because a period looks more like a decimal, and is also easier to use in comma separated values (CSV) files.
- For the timezone, we use hours and minutes with a separator, rather than “Z” or “UTC” or “00” or “0000”, because we want timezone syntax to sort correctly among multiple timezones, and want to clearly spearate the minutes.

For booleans use true and false

Booleans using true and false can use the shell built-in commands like this:

```
x=true
if [ "$x" = true ]; then
    ...
fi
```

Notice that this is not using numbers (such as 0 or 1) and not using commands (such as `/bin/true` or `/bin/false`)

Subshells: use parentheses not backticks

To write a subshell command, you can use parentheses or backticks.

Example:

```
$(foo)   # yes
`foo`    # no
```

We prefer parentheses over backtick because parentheses are nestable.

Example:

```
$(foo $(goo $(hoo))
```

Be aware that escaping characters is slightly different, such as:

```
echo $(echo \$abc)
echo `echo \$abc`
```

Differences also exist for:

```
$(echo \`)
$(echo \\)
```

Thanks to http://stackoverflow.com/users/313821/peter-o

Trace using set -x then set +x without printing

To help with debugging a script, you can use the “echo before execute” setting:

Turn on “echo before execute”:

```
set -x
```

Turn off “echo before execute”:

```
set +x
```

Turn off “echo before execute” without printing the line:

```
{ set +x; } 2>/dev/null
```

Example to turn on, then run some commands, then turn off:

```
set -x
command1
command2
command3
{ set +x; } 2>/dev/null
```

Arg parse

Argument parsing using a loop in order to find each relevant argument, and remove it from the array.

See related:

- for arg do loop
- Parse options via `while` and `case` not `getopts` or `getopt`

```
for arg
do
    shift
    case $arg in
    --[[:alnum:]][-_[:alnum:]]*)
        key="$( printf %s "$arg" | sed 's/^--//; s/[^-_[:alnum:]].*$//' )"
        var="$( printf %s "$key" | sed 's/-/_/g' | tr '[:lower:]' '[:uppe
        case $arg in
        --"$key"=)
            eval "$var"="${arg#*=}"
            ;;
        --"$key")
            eval "$var"=1
            ;;
        *)
            >&2 printf %s\\n "arg: $arg"
            ;;
        esac
        ;;
    *)
        set -- "$@" "$arg"
        ;;
    esac
done
```

Notes

The code `for arg` is shorthand for `for arg in "$@"`.

The code that sets key to value 1 is because the number 1 is how shell arithmetic represents "true".

Parse options via `while` and `case` not `getopts` or `getopt`

Unix shell scripts have a variety of ways to do options parsing such as:

- `while` and `case` which you write yourself.
- `getopts` is a POSIX standard. See https://pubs.opengroup.org/onlinepubs/7908799/xcu/getopts.html.
- `getopts_long` is a POSIX script. See http://stchaz.free.fr/getopts_long.sh.
- `getopt` is a GNU tool. See https://www.gnu.org/software/libc/manual/html_node/Getopt.html.
- `zparseopts` is a Z shell tool. See the man page `zshmodules(1)`.

There are pros and cons to each.

If you want simple parsing, then we recommend writing your own via `while` and `case`.

If you want sophisticated parsing, then we recommend changing from a Unix shell script to a more-capable programming language.

`while` and `case`

Use a `while` infinite loop that contains a `case` statement for all the command line options.

Example:

```
## Set verbose level to default 0
verbose=0

## Set any options to defaults
foo=""

## Process all the command line options
while :; do
```

```
    case $1 in
        # Two hyphens ends the options parsing
        --)
            shift
            break
            ;;
        # Print help by calling a help() function that you define later
        -h|--help)
            help
            exit
            ;;
        # Each verbose option is treated as incrementing the verbose leve
        -v|--verbose)
            verbose=$((verbose + 1))
            ;;
        # Parse an option value via this syntax: --foo bar
        -f|--foo)
            if [ -n "$2" ]; then
                foo=$2
                shift
            else
                die "The command option --foo requires a value"
            fi
            ;;
        # Parse an option value via this syntax: --foo=bar
        --foo=?*)
            foo=${1#*=}
            ;;
        # Parse an option value via this syntax: --foo= (i.e. blank)
        --foo=)
            die "The command option --foo requires a value"
            ;;
        # Anything remaining that starts with a dash triggers a fatal erro
        -?*)
            die "The command line option is unknown: " "$1"
            ;;
        # Anything remaining is treated as content not a parseable option
        *)
            break
            ;;
    esac
    shift
done

help(){
cat << EOF
    # add your help documentation here
EOF
}

die(){
    >&2 printf %s\\n "$*"
    exit 1
}
```

getops

The POSIX `getopts` approach can be great for combining one-letter options, such as the classic `ps -aux`.

Unfortunately POSIX `getopts` doesn't support long options, such as –help. This means a script can't use long options that are descriptive such as –dry-run, –danger, –minimum, –maximum, etc.

For some teams, and some commands, the long options are worth the tradeoff because they improve readability, as well as flexibility when a programmer is creating a script's option names. For a majority of our use cases, we value the long options more highly than the short options combining letters, and we're willing to write the parsing ourselves.

getops versus getopt

In practice there are significant differences between `getopts` and `getopt` and their various versions on various systems:

- The `getopts` POSIX standard can parse a short option, but not a long option, whereas `getopt`can do both. Example: "-h" is a short option, and "–help" is a long option.
- The `getopt` tool has had parsing issues when there is an empty

argument string. Example: "–foo=bar" succeeds, whereas "–foo=" fails even though it could mean that foo should be set to a blank string.

- Different capabilities can happen depending on the specific system. Example: some systems do not come with `getopt`, whereas some systems do come with `getopt` yet there are different versions with different capabilties due to use of different package sources such as `util-linux` versus `linux-utils`.

getopts_long

This is written as a POSIX shell function that we embed within a shell script.

When you want the best capabilties for options parsing, and you do not want to change from a Unix shell script to a more-capable programming language, then try `getopts_long`.

zparseopts

This is an approach that is specific to Z shell (zsh).

When you have a specification that states the shell script will exclusively run on Z shell, then try `zparseopts`.

Environment variables: test if set or unset

POSIX test if set:

```
if [ -n "${FOO+1}" ]; then
```

POSIX test if unset:

```
if [ -z "${FOO+1}" ]; then
```

How it works: `${var+1}` is a parameter expansion which evaluates to nothing if `var` is unset, and substitutes the string `1` otherwise.

We prefer `+1` over `+x` because `1` is already defined and already means true, and also because `+1` is the most-common way in POSIX posts that we read.

Note: For the unset test, quotes can be omitted, yet we prefer to use quotes because we believe they increase clarity, consistency, and to bulletproofing for novices in case someone changes the test from unset to set. The reason quotes can be ommitted for the unset test, i.e. using `${var+1}` instead of `"${var+1}"`, is because this syntax and usage guarantees this will only expand to something that does not require quotes (since it either expands to `1` (which contains no word breaks so it needs no quotes), or to nothing (which results in `[ -z ]`, which conveniently evaluates to the same value (true) that `[ -z "" ]` does as well)).

See
https://stackoverflow.com/questions/3601515/how-to-check-if-a-variable-is-set-in-ba

Do not use

Bash and Zsh above certain versions:

```
if [[ -v FOO ]]; then
if [[ -z FOO ]]; then
```

$FUNCNAME function name

To know the current function name, we must set it manually for POSIX, and we prefer to use the environmen variable `$FUNCNAME` because it matches the automatic Bash environment variable `$FUNCNAME`.

To make the code compatible on POSIX and Bash:

```
foo() {
    FUNCNAME=${FUNCNAME:-foo}
    echo "$FUNCNAME"
}
```

Script directory

To get the directory of a running script:

```
dir() { CDPATH= cd -- "$(dirname -- "$0")" && pwd -P ; }
```

This solution is POSIX. However, it's not perfect; it has issues with symlinks, and also does not work if the script is sourced.

https://stackoverflow.com/questions/29832037/how-to-get-script-directory-in-posix-s

Data directory: use $XDG_DATA_HOME

We prefer to have a user's data directory in a standard place:

- Use `XDG_DATA_HOME` environment variable if it's set.
- Otherwise use `$HOME/.local/share`.
- Append the program command, so the program uses its own subdirectory.

Code:

```
data_home() { out "${XDG_DATA_HOME:-$HOME/.local/share}" ; }; export -f da
data_dir() { out $(data_home)"/$program_command" ; };
```

References:

- https://wiki.archlinux.org/index.php/XDG_Base_Directory_support

Purpose:

- Where user-specific data files should be written (analogous to /usr/share).
- Should default to $HOME/.local/share.

Cache directory: use $XDG_CACHE_HOME

We prefer to have a user's cache directory in a standard place:

- Use `XDG_CACHE_HOME` environment variable if it's set.
- Otherwise use `$HOME/.cache`.
- Append the program command, so the program uses its own subdirectory.

Code:

```
cache_home() { out "${XDG_CACHE_HOME:-$HOME/.cache}" ; }; export -f cache_
cache_dir() { out $(cache_home)"/$program_command" ; };
```

References:

- https://wiki.archlinux.org/index.php/XDG_Base_Directory_support

Purpose:

- Where user-specific non-essential (cached) data should be written (analogous to /var/cache).
- Should default to $HOME/.cache.

Configuration directory: use $XDG_CONFIG_HOME

We prefer to have a user's configuration directory in a standard place:

- Use `XDG_CONFIG_HOME` environment variable if it's set.
- Otherwise use `$HOME/.config`.
- Append the program command, so the program uses its own subdirectory.

Code:

```
config_home() { out "${XDG_CONFIG_HOME:-$HOME/.config}" ; }; export -f cor
config_dir() { out $(config_home)"/$program_command" ; };
```

Notes:

- We prefer the terminology "config" over "conf", "cfg", etc.
- Our preference is because "config" is the Unix standard, and also is in the XDG environment variable name.

References:

- https://wiki.archlinux.org/index.php/XDG_Base_Directory_support

Purpose:

- Where user-specific configurations should be written (analogous to /etc).
- Should default to $HOME/.config.

Runtime directory: use $XDG_RUNTIME_HOME

We prefer to have a user's runtime directory in a standard place:

- Use `XDG_RUNTIME_HOME` environment variable if it's set.
- Otherwise use `$HOME/.runtime`.
- Append the program command, so the program uses its own subdirectory.

Code:

```
runtime_home() { out "${XDG_RUNTIME_HOME:-$HOME/.runtime}" ; }; export -f
runtime_dir() { out $(runtime_home)"/$program_command" ; };
```

Purpose:

- Used for non-essential, user-specific data files such as sockets, named pipes, etc.
- Not required to have a default value; warnings should be issued if not set or equivalents provided.
- Must be owned by the user with an access mode of 0700.
- Filesystem fully featured by standards of OS.
- Must be on the local filesystem.
- May be subject to periodic cleanup.
- Modified every 6 hours or set sticky bit if persistence is desired.
- Can only exist for the duration of the user's login.
- Should not store large files as it may be mounted as a tmpfs.

Temporary directory: use mktemp

To create a temporary directory we use:

- The command `mktemp` which creates the directory.
- If a temp prefix is provided, then use it; we prefer the use `program_command` which returns the name.
- Otherwise, use a ZID i.e. secure random 32-character hex lowercase string.

Code:

```
temp_home() { out $(mktemp -d -t "${1:-$(zid)}"); }; export -f temp_home;
temp_dir() { out $(temp_home "$program_command"; };
```

Notes:

- We prefer the terminology “temp” over “tmp”, “temporary”, etc.
- Our preference is because “mktemp” uses “temp”.
- And also because “temp” is easier for some cultures to pronounce than “tmp”.

Temporary file using `mktemp` and `trap`

To create a temporary file:

```
file=$(mktemp)
```

Why use `mktemp` instead of `tempfile`?

- Because `mktemp` is available on more systems.

To remove a temporary file when the program exist:

```
trap "rm -f $file" EXIT
```

Why trap on EXIT, instead of TERM, INT, HUP?

- Because EXIT covers all the cases.

Find files with filter for "Permission Denied"

The `find` command can print error messages that say "Permission Denied".

To filter out these errors and use a robust POSIX syntax:

```
{ LC_ALL=C find . 3>&2 2>&1 1>&3 | grep -v 'Permission denied' >&3; [ $? -
```

How it works:

- `LC_ALL=C` configures the command to show C programming language error messages. These which always use the text "Permission Denied". If we skipped this step, and the system is configured to show localized error messages, then the error messages could be in other languages and thus not filtered.

- `find .` is the typical find command and the local directory. You can put any of your options here.

- `3>&2 2>&1 1>&` creates a custom file descriptor 3 and temporarily swaps stdout (1) and stderr (2).

- `grep -v 'Permission denied'` filters out any lines that contain the text.

- `[ $? -eq 1 ];` Set the exit code to indicate whether any errors other than Permission denied occurred: 1 if so, 0 otherwise. In other words: the exit code now reflects the true intent of the command: success (0) is reported, if no errors at all or only permission-denied errors occurred.

- `3>&2 2>&1` reswaps the file descriptors, restoring typical stdout and stderr.

Details:

- This is POSIX-compliant.

- Credit:
  https://stackoverflow.com/questions/762348/how-can-i-exclude-all-permission

Alternatives

Prune Prune any unreadable items:

```
find . ! -readable -prune -o -print:
```

- Pro: Fast if your version of `find` has these options.
- Con: This typically requires GNU find, which typically isn't on vanilla macOS.

Ignore Ignore errors altogether:

```
find . 2>/dev/null > files_and_folders
```

- Pro: Fast if you're willing to skip all errors.
- Con: this sacrifices all other errors.

Sudo Use the `sudo` command to enable permissions for everything:

```
sudo find . > foo
```

- Pro: Easy and clear, if you want just to print results.
- Con: anything else in the command runs using sudo as well.

Merge streams Merge the streams for stdio and stederr:

```
find . 2>&1 | grep -v 'Permission denied' > foo
```

- Pro: None, compared to others on this page.
- Con: Real errors don't go to stderr.

Find files with special characters

When we use the `find` command, we want to handle paths with special characters, such as spaces, tabs, newlines, etc.

Simple find -exec with ‘;’. This may be unwieldy if COMMAND is large. This creates a separate process per file.

```
find . -exec COMMAND... {} \;
```

Simple find -exec with ‘+’. If COMMAND is able to take multiple files, this is faster.

```
find . -exec COMMAND... {} \+
```

Find items in the current directory that start with a space or end with a space:

```
find . -type d \( -regex '\./ .*' -o -regex '.* ' \) -exec echo "==={}==='
```

Find and switch to each file’s directory then execute a command from there:

```
find . -name '*.txt' -execdir /mycmd {} \;
```

Find using portable semicolon and portable null termination:

```
find . -exec printf %s\\0 '{}' \;
```

Find using portable semicolon and portable null termination, then a while loop:

```
find . -exec printf %s\\0 '{}' \; | while read -d $'\0' file; do ...
```

Find items and run a shell script on each item. This works portably. Use “” for single-quote in command. This runs a subshell, so variable values are lost after each iteration.

```
find . -exec sh -c '
for file do
    ...  # Use "$file" not $file
done' sh {} +
```

Credit: http://www.dwheeler.com/essays/filenames-in-shell.html

Find files with leading and/or trailing spaces then fix them:

```
find . -maxdepth 1 \( -regex '\./ .*' -o -regex '.* ' \) -exec sh -c '
for src do
    src=${src/#.\//}
    dst="$src"
    dst=${dst/# */}
    dst=${dst/% */}
    echo "==$src== ==$dst=="
    mv --interactive "$src" "$dst"
done' sh {} +
```

Find files with readable permissions

The `find` command can return `Permission denied` directory errors.

To skip these, test the permissions. If there are no readable permissions, then prune it:

```
find . ! -perm +222 -prune -o -name '*foo*' -print
```

Some versions of the `find` command provide a non-POSIX `-readble` flag:

```
find . ! -readable -prune -o -name '*foo*' -print
```

Find files with exectuable, perm, test, exec

To find files that are executable by the user, using a POSIX compatible syntax:

```
find . -type f \( -perm -u=x -o -perm -g=x -o -perm -o=x \) -exec test -x
```

To find files and run them:

```
find . -type f \( -perm -u=x -o -perm -g=x -o -perm -o=x \) -exec test -x
```

Details

From
https://stackoverflow.com/questions/4458120/unix-find-search-for-executable-files

Finding files that are executable can refer to two distinct use cases:

- user-centric: find files that are executable by the current user.
- file-centric: find files that have (one or more) executable permission bits set.

Note that in either scenario it may make sense to use find -L … instead of just find … in order to also find symlinks to executables.

Note that the simplest file-centric case - looking for executables with the executable permissions bit set for ALL three security principals (user, group, other) - will typically, but not necessarily yield the same results as the user-centric scenario - and it's important to understand the difference.

User-centric

To find user-centric excutable files:

- If you are using GNU find, then use the `-executable` flag. The flag matches only files the current user can execute (there are edge cases).

- If you are using BSD-based platforms, including macOS, then there is no `-executable` flag.

File-centric

To answer file-centric questions, it is sufficient to use the POSIX-compliant -perm primary (known as a test in GNU find terminology).

- `-perm` allows you to test for any file permissions, not just executability.
- Permissions are specified as either octal or symbolic modes. Octal modes are octal numbers (e.g., 111), whereas symbolic modes are strings (e.g., a=x).
- Symbolic modes identify the security principals as “u” (user), “g” (group) and “o” (other), or “a” (all three). Permissions are expressed as x for executable, for instance, and assigned to principals using operators =, + and -; for a full discussion, including of octal modes, see the POSIX spec for the chmod utility.

Notes

In the context of find:

- Prefixing a mode with - (e.g., -ug=x) means: match files that have all the permissions specified (but matching files may have additional permissions).
- Having NO prefix (e.g. 755) means: match files that have this full, exact set of permissions.
- Caveat: Both GNU find and BSD find implement an additional, nonstandard prefix with are-ANY-of-the-specified-permission-bits-set logic, but do so with incompatible syntax.

Program name using a string name or basename

We prefer a program to be able to return its own name.

Our convention: create a function `program()`.

The function can return a string name such as:

```
program() { echo "our-demo-tool"; }
```

If the program may use dynamic naming, then the function can calculate a name:

```
program(){ echo "$(basename "$0")"; }
```

There may be a better way; if you know a better way, please let us know.

Functions: out, err, die, log, now, sec, zid, cmd, etc.

Our scripts may use these all-purpose functions:

- `out` is for output messages; it prints to STDOUT.
- `err` is for error messages; it prints to STDERR.
- `die` is for fatal messages; it prints to STDERR then exits.
- `big` is for banner messsages; it prints to STDOUT.
- `log` call `out()` prepending a time stamp and PID.
- `now` return a timestamp using UTC and ISO 8601:2004.
- `sec` return a timestamp using UTC and Unix epoch second.
- `zid` return a 128-bit secure random hex lowercase ID.
- `cmd` return a path to the default runnable command in $1.

Code:

```
out() { printf %s\\n "$*" ; }; export -f out
err() { >&2 printf %s\\n "$*" ; }; export -f err
die() { >&2 printf %s\\n "$*" ; exit 1 ; }; export -f die
big() { printf \\n###\\n#\\n#\ %s\\n#\\n###\\n\\n "$*"; }; export -f big
log() { printf '%s %s %s\n' "$( now )" $$ "$*" ; }; export -f log
now() { date -u "+%Y-%m-%dT%H:%M:%S.%NZ" ; }; export -f now
sec() { date -u "+%s" ; }; export -f sec
zid() { hexdump -n 16 -v -e '16/1 "%02x" "\n"' /dev/random ; }; export -f
cmd() { command -v "$1" >/dev/null 2>&1 ; }; export -f cmd
```

Caveat about now()

If you use `now()` on a platform where the date command does not support the nanosecond format, then the output will show “%N”. This is harmless.

If you wish, you can install a date command such as GNU date that does support the nanosecond format.

For example on macOS 10.13, the built-in date command does not support the nanosecond format. If you wish, you can install GNU date

and many other GNU commands, and also prefer them to the built-in commands, by using Homebrew like this:
`brew install coreutils --with-default-names`.

Caveat about sec()

If you use `sec()` on many platforms, then see sec() function portability

Assert functions

We like to test our code during runtime by using assertions a.k.a. assert functions:

```
assert_empty() { [ -z "$1" ]        || err $FUNCNAME "$@" ; }; export -f ass
assert_equal() { [ "$1" = "$2" ]  || err $FUNCNAME "$@" ; }; export -f ass
```

Examples:

```
assert_empty ""
assert_equal "foo" "foo"
```

sec() function portability

Our scripts define the `sec()` function. It returns the count of seconds since 1970-01-01 00:00:00 UTC.

We use this code:

```
sec() { date "+%s" }
```

If you need to support a wide range of platforms, then we recommend this code:

```
sec() {
    if date '+%s' >/dev/null 2>&1; then
        date '+%s'
    elif command -v perl >/dev/null 2>&1; then
        perl -e "print time"
    elif command -v truss >/dev/null 2>&1 && [ "$(uname)" = SunOS ]; then
        truss date 2>&1 | grep ^time | awk -F"= " '{print $2}'
    elif command -v truss >/dev/null 2>&1 && [ "$(uname)" = FreeBSD ]; the
        truss date 2>&1 | grep ^gettimeofday | cut -d "{" -f2 | cut -d ".'
    fi
}
```

Thanks to whetu for the portable version.

While loop with index counter

To create a while loop with an index counter, and make it POSIX standard, we create the index variable before the loop.

Example:

```
i=0
max=3
while [ "$i" -le "$max" ]; do
    echo "output: $i"
    true $((i=i+1))
done
```

Notes:

- `true $(( i++ ))` doesn't work in all cases, so we use `true $((i=i+1))`.
- Credit: http://stackoverflow.com/questions/1445452/shell-script-for-loop-syntax

Do while loop

One way to accomplish a `do...while` concept is by using `while true` and a `break`:

```
while true; do
    ... commands ...
    condition || break
done
```

A versatile version of a `do...while` concept can use this structure:

```
while
    ... commands ...
do :; done
```

Example:

```
i=3
while
    echo "example $i"          # this command runs each iteration
    : ${start=$i}              # capture the starting value of i
    echo "in the loop"         # your code goes here; needed for the loop
    i="$((i+1))"               # increment the variable of the loop.
    [ "$i" -lt 20 ]            # test the limit of the loop.
do :;  done
```

Notes:

- Credit: http://unix.stackexchange.com/questions/60098/do-while-or-do-until-in-posix-

Case statement that skips option dash flags

Example:

```
case "$x" in
    --) echo "a double dash terminates options" ;;
    -*) echo "a single dash is an option" ;;
    *) echo "anything else is an argument" ;;
esac
```

Export function

We export functions when we want them to be available to other scripts.

One example is when we want to use GNU parallel, and we want parallel to be able to call functions.

Example:

```bash
foo() { echo "hello"; }
export -f foo
```

Number functions

int()

Code:

```
int() { awk '{ print int($1) }' ; export -f int
```

We considered other implementations, but they did not work on more systems, or had issues with negative numbers, or interpreting a leading zero as octal, etc.:

```
printf -v int '%d\n' "$1" 2>/dev/null
```

sum()

Code:

```
sum() { awk '{for(i=1; i<=NF; i++) sum+=$i; } END {print sum}' ; }; export
```

Array functions

We use simple array functions.

- The function `array_n` returns the array number of fields a.k.a. length.
- The function `array_i` returns the array item at index i.

Syntax:

```
array_n <string> [field separator]
array_i <string> [field separator] <index>
```

Examples:

```
array_n 'a b c' => 3
array_n 'a-b-c' '-' => 3

array_i 'a b c' 2 =>  'b'
array_i 'a-b-c' '-' 2 =>  'b'
```

Source:

```
array_n() { [ $# == 2 ] && awk -F "$2" "{print NF}"   <<< "$1" || awk "{pr
array_i() { [ $# == 3 ] && awk -F "$2" "{print \$$3}" <<< "$1" || awk "{pr
```

URL encode and URL decode

Credit:
https://stackoverflow.com/questions/296536/how-to-urlencode-data-for-curl-comma

```
url_encode() {
  local string="${1}"
  local strlen=${#string}
  local encoded=""
  local pos c o
  for (( pos=0 ; pos<strlen ; pos++ )); do
     c=${string:$pos:1}
     case "$c" in
        [-_.~a-zA-Z0-9] ) o="${c}" ;;
        * )               printf -v o '%%%02x' "'$c"
     esac
     encoded+="${o}"
  done
  echo "${encoded}"
}

## Returns a string in which the sequences with percent (%) signs followed
## two hex digits have been replaced with literal characters.
url_decode() {
  # Because all escape characters must be encoded, we can replace %NN with
  # and pass the lot to printf -b, which will decode hex for us.
  printf -v REPLY '%b' "${1//%/\\x}" # You can either set a return variabl
  echo "${REPLY}"
}
```

awk match_between

To do a POSIX awk match between two strings:

```
function match_between(s, open, shut){
    open_index=index(s, open)
    if (open_index>0){
        open_length=length(open)
        RSTART=open_index+open_length
        shut_index=index(substr(s, RSTART), shut)
        if (shut_index>0){
            RLENGTH=shut_index-1
            return RSTART
        }
    }
    return 0
}
```

`readlink` on macOS behaves differently

The `readlink` command on macOS behaves differently from `readlink` on other Unix systems.

`realpath` is‘ not available on macOS default

The `realpath` is not available on macOS default.

psql helpers

We work with PostgreSQL frequently

We have simple shell functions to help us inspect our PostgreSQL servers.

```
psql_user_names() { psql -tAc "SELECT usename FROM pg_catalog.pg_user;" ;
psql_user_name_exist() { [ "$( psql -tAc "SELECT 1 FROM pg_catalog.pg_use
psql_database_names() { psql -tAc "SELECT datname FROM pg_database;" ; }
psql_database_name_exist() { [ "$( psql -tAc "SELECT 1 FROM pg_database WH
```