Apple Admin Scripting for Beginners

...and the absolutely petrified

If you're an Apple admin who is terrified of scripting, lift high your courage: **this guide is for you!**

You can improve your speed, accuracy and the satisfaction of your end users with just a handful of simple scripts.

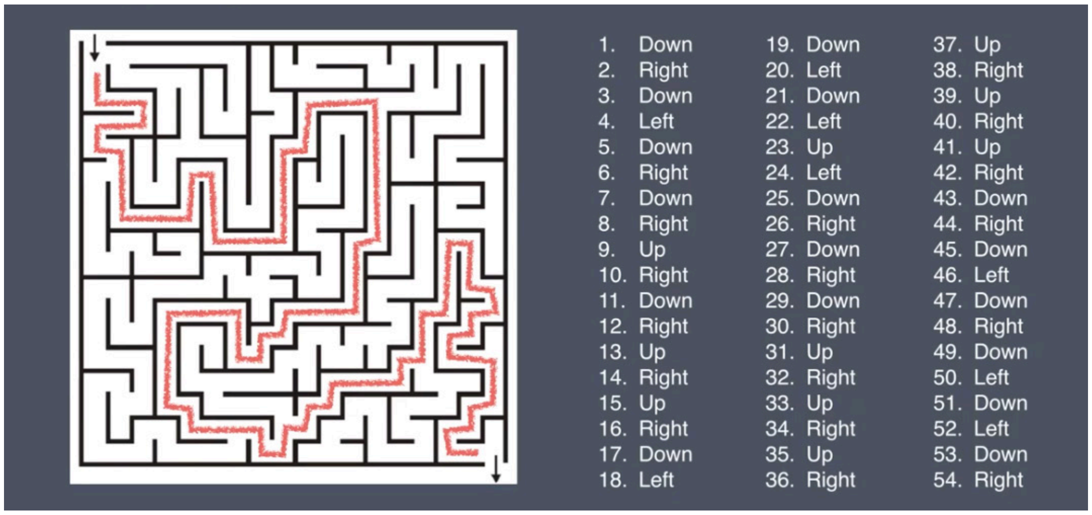
If you've tried your hand at scripting before and found yourself floundering, there's probably a reason for that: most tutorials jump directly into providing you with scripts to memorize that may or may not make sense to you, before they even explain what scripting is and why it's a good idea to learn some.

And getting started is easy, as long as you can understand how and why scripting works before trying to memorize strings of commands.

Why do we script?

Scripting lets us simplify.

We can reduce something complex like navigating a maze into a set of basic instructions. All you need are four commands: up, down, right, left. All you do is break down the problem into small steps, such as in this example:



1. Down
2. Right
3. Down
4. Left
5. Down
6. Right
7. Down
8. Right
9. Up
10. Right
11. Down
12. Right
13. Up
14. Right
15. Up
16. Right
17. Down
18. Left
19. Down
20. Left
21. Down
22. Left
23. Up
24. Left
25. Down
26. Right
27. Down
28. Right
29. Down
30. Right
31. Up
32. Right
33. Up
34. Right
35. Up
36. Right
37. Up
38. Right
39. Up
40. Right
41. Up
42. Right
43. Down
44. Right
45. Down
46. Left
47. Down
48. Right
49. Down
50. Left
51. Down
52. Left
53. Down
54. Right

You can accomplish what looks like a complex feat with only four commands.

Scripting lets us automate.

To open a file in your computer just takes a double-click. To open a file on your computer and another computer takes two double-clicks. But how about opening 24 files on 24 computers at the same time?

Using an MDM such as Jamf Pro, we can automate this command to all computers with a simple one-line command: open, along with the location of the file.

Scripting reduces errors.

When tasks depend on human input, they are also subject to human error. This can increase the need to re-do work as well as increasing calls to the help desk. Scripting tasks that are done routinely ensures that they are done correctly (and quickly!) every time.

How does Terminal work?

Interpreters

Don't know what those are? Actually, you do!

You've already been using interpreters for years. The most common interpreters are Windows Explorer and Mac Finder. Their job is to interpret your mouse clicks and keyboard commands into computer actions. If you click a file, both Windows Explorer and Mac Finder interpret that click as you wanting to select a file. Double-click; they interpret that you want to open a file.

However, not all interpreters will interpret your actions the same way. For example: if you select a file in Explorer and hit ENTER, it opens the file. If you do the same on a Mac, it allows you to edit the file name.

Terminal is where Apple users run their scripts. Understanding how Terminal works helps you demystify scripting.

Terminal itself isn't itself an interpreter. It's just a window to an interpreter called bash. Your Mac contains multiple interpreters, but this e-book focuses on bash.

When you open Terminal from the application's utilities folder, it logs you in to an interpreter: bash. Bash is short for Bourne-AgainSHell. It's a revised version of the interpreter shell (sh) created by Steven Bourne 40 years ago.

How do commands work?

In the simplest sense, commands tell a computer what to do. You can do anything you'd do using the Finder. For instance, you can open a file by double-clicking on it. Or, you can open the Terminal and use the following command, replacing the name of the document with anything you currently have on your desktop:

```
open /Users/jamfwebinar/Desktop/scripting101.docx
```

How about a web page instead of a file? The open command does that, too. Just tell it the URL and which browser you want it to open with:

```
open https://www.jamf.com -a Safari
```

Here's the interactive part: you'll get the most out of this e-book if you open up Terminal and follow along as we go.

Applications > Utilities > Terminal

Let's start with the basics.

The first program most people learn is **'Hello, World,'** in Terminal.

This seems like a simple exercise, but you will be using the echo command in many different ways as you move forward with your scripting, so give it a try!

From the Terminal, type:

```
echo 'Hello World'
```

And hit return.

As you can see, the `echo` command means: **repeat what I say**.

To really give a command some teeth, you add in an argument: the text inside of the single quotes. The argument tells the command what to do.

To learn more simple scripts that have more relevance in your day-to-day workflows, you can start with doing basic things you would do on your desktop. A good practice is to start doing this as a matter of course throughout your day as you do routine tasks with the Finder, to help you feel more comfortable with scripting.

For example, you already know how to, through Mac Finder:

- Create a folder named ‘MyStuff.’
- Create a file with content in it.
- Put the file in the folder.

You can do this in Terminal, too, with the following commands:

`mkdir` = make directory (this is a new one and creates a folder)
`echo` = repeat what I say
`mv` = move (or rename; we’ll use it as ‘move’ in this exercise)

And you can use the commands in new ways.

To create a folder on your desktop named ‘MyStuff,’ in Terminal at the command line, key:

```
mkdir Desktop/MyStuff
```

And hit return.

To create a file with content in it, start backwards with the content, and then point (>) the content to where and in what format you would like that content to be saved, using the usual format for a file location address:

```
echo "The quick brown fox jumped over the lazy dogs." > Desktop/Quick-Brown-Fox.txt
```

And hit return.

Put the file in the folder. From the command line, key:

```
mv Desktop/Quick-Brown-Fox.txt Desktop/MyStuff
```

And hit return.

One thing about the command line in Terminal is that if you told it to do something, it does it. It doesn’t give you a confirmation message (unless you told it to).

So, for your peace of mind, check via the Finder to ensure you have a new folder called “MyStuff” and that it contains the document Quick-Brown-Fox.txt.

You can open the document through Terminal, as well!

Input:

```
open Desktop/MyStuff/Quick-Brown-Fox.txt
```

And hit return.

You can delete documents and folders through Terminal, as well, with the `rm` command.

`rm` = remove
`rm -r` = remove recursively

To remove a document, you simply need `rm`. Try the following in the command line:

```
rm Desktop/MyStuff/Quick-Brown-Fox.txt
```

Via the Finder, check the folder, and the Quick-Brown-Fox.txt document should be gone.

If you try the same thing to remove the folder:

```
rm Desktop/MyStuff
```

When Terminal cannot or will not do what you ask, you will receive an error message, as you did:

```
rm: Desktop/MyStuff: is a directory
```

To tell Terminal that you REALLY mean to remove the folder/directory and all contents in it, you need to include the remove recursively command, or `rm -r`.

Try it in Terminal, and then check to ensure that the MyStuff folder is gone.

```
rm -r Desktop/MyStuff
```

Note: If you want a folder or document to have spaces in it, simply surround the name with quotation marks. For example, “My Stuff” or “Quick Brown Fox.txt”.

Writing scripts from Terminal Commands

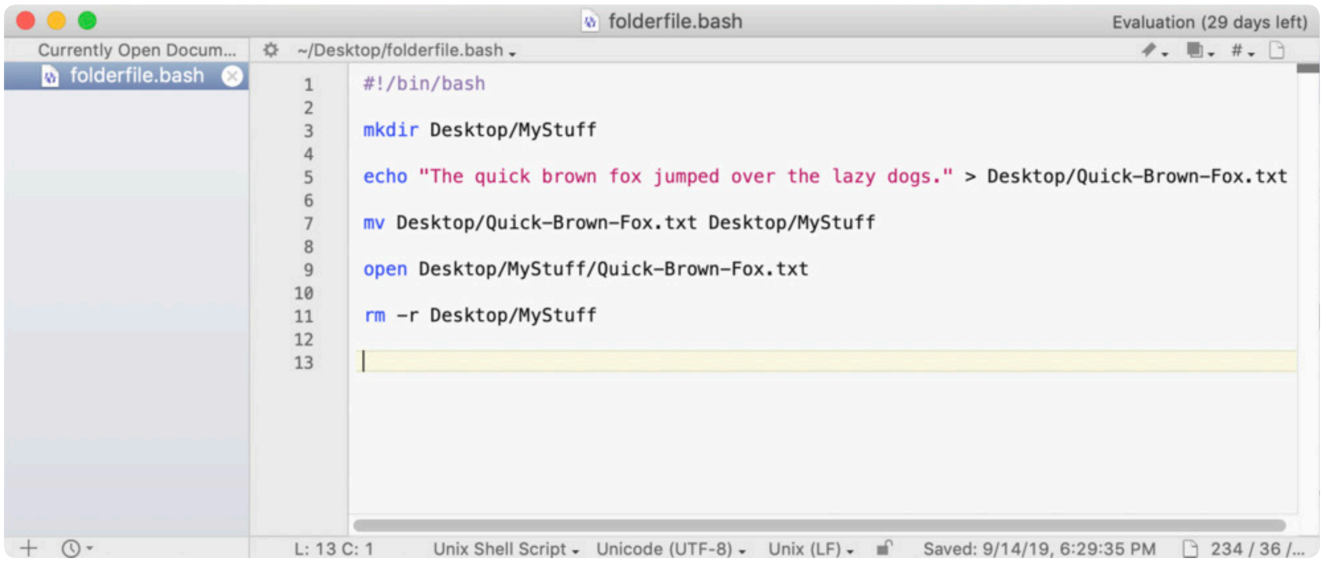
Now, we're going to make a script out of what you have just done in Terminal!

First, download and open a script editor. We like BBEdit, but there are others listed as resources at the end of this e-book. You can use a text editor, but a free script editor generally offers syntax highlighting: changing colors of specific parts of script so that you can see at a glance which is which, as well as find and fix any errors. This is especially helpful in understanding what is going on for those new to scripting syntax, and for troubleshooting. In this case, commands are blue and arguments are pink.

Always type at the top of each script something called a shebang: called hash + bang #! Followed by /bin/bash, so that when you use your scripts, Terminal will know to use bash as an interpreter.

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

Copy and paste each line of commands you have used already into the script editor. Adding blank lines helps to make it more readable.



```
#!/bin/bash

mkdir Desktop/MyStuff

echo "The quick brown fox jumped over the lazy dogs." > Desktop/Quick-Brown-Fox.txt

mv Desktop/Quick-Brown-Fox.txt Desktop/MyStuff

open Desktop/MyStuff/Quick-Brown-Fox.txt

rm -r Desktop/MyStuff

```

Save the script to your Desktop as “**folderfile.bash**”.

Now, drag this file into Terminal, and hit ‘enter.’

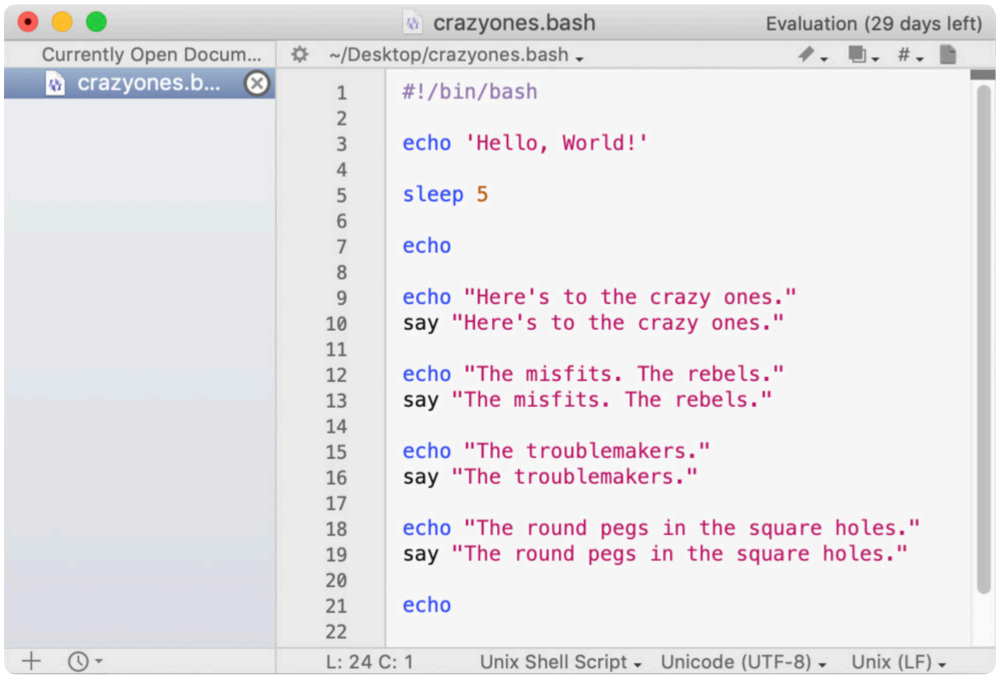
Did you blink? You might have missed it! Another good reason to use scripts: they’re fast!

Here's a fun one to try, this time starting from a script editor to begin with.

Open a script editor and enter:

```
#!/bin/bash
echo 'Hello, World!'
sleep 5
echo
echo "Here’s to the crazy ones."
say "Here’s to the crazy ones."
echo "The misfits. The rebels."
say "The misfits. The rebels."
echo "The troublemakers."
say "The troublemakers."
echo "The round pegs in the square holes."
say "The round pegs in the square holes."
echo
exit
```

This is how it will look:



```
1  #!/bin/bash
2  
3  echo 'Hello, World!'
4  
5  sleep 5
6  
7  echo
8  
9  echo "Here's to the crazy ones."
10 say "Here's to the crazy ones."
11 
12 echo "The misfits. The rebels."
13 say "The misfits. The rebels."
14 
15 echo "The troublemakers."
16 say "The troublemakers."
17 
18 echo "The round pegs in the square holes."
19 say "The round pegs in the square holes."
20 
21 echo
22 
```

As you can see, line three is an echo command with ‘Hello, World!’ echo is blue, and the variables are pink.

The fifth line is `sleep 5`.

`Sleep` means pause, and 5 means to pause for 5 seconds.

`Say` is a different command — and you'll see what it is in a minute.

A blank `echo` command is simply a way to put in a space: "echo [blankspace]."

All of these commands in a row is a script.

1. Save this one as your previous script, with `.bash` as a suffix.
2. Select the file and drag it into Terminal.
3. Make sure your sound is turned on.
4. Press return, and the script will run.

See what we did there? Now you know what `say` means.

Commands you've learned so far:

`echo` = repeat what I enter

`sleep` = pause (should be used in conjunction with an argument for how many seconds)

`say` = read aloud

`open` = open any file

`mkdir` = create directory/folder

`mv` = move a file, or rename a file

`rm` = remove a file

`rm -r` = remove recursively (used when removing a folder/directory)

Variables: placeholders for things to come

Variables are like fill-in-the-blanks: placeholders for what is going to come later. The value we assign it could be a number, text, filename, device or any other type of data. A variable is not a thing in itself. It POINTS to actual data that we will pull in later.

Variables are powerful, because if you create them correctly, you can run the same script many times without having to rewrite it, all while working with changeable data.

Let's take a look at how they work using the simplest variable: a name. And, for now, we will work backwards, like we did with creating a file that contains content.

First: we tell computer what the data will be. In this case, type into terminal:

```
myName=Martin
```

And then hit 'enter.' Always use a variable name that is clear, straightforward and easy to remember for later use.

Then, we tell the computer what to do with the data. Whenever we are going to use a variable, indicate that it is a variable with the $ symbol. So, you'll type into terminal:

```
echo Hello, my name is $myName.
```

And hit enter.

In the Terminal, you should see:

```
Hello, my name is Martin.
```

This practice of defining a variable and then using the variable in a script has many practical uses for your workflow.

One-liners

Here's another useful command, modified by a variable. Let's say that you wanted to figure out if a Mac needs an OS upgrade.

Again: we're going to work backwards.

First, we need to decide which variable we are going to use — and we're going to let a command result find it for us.

This will require two new commands (really, one: simple, and then refined):

`sw_vers` = software version
`sw_vers -productVersion` = only the product version of the software

The `sw_vers` command tells us information about the macOS: the name of the OS, the version of the OS we're running, and the build number. If you want only the version, you can write the script with a -productVersion after the command, and it gives you only what you want.

Try it in Terminal, simply as a command.

The first version's results should look something like:

```
ProductName:  Mac OS X
ProductVersion: 10.14.4
BuildVersion:  18E226
```

For the second, the result would simply be:

```
    10.14.4
```

Working from already-existing commands means we don't have to set the variable ahead of time, but we do have to indicate that it is a variable we are using. You would set up the variable like this:

```
    $( sw_vers -productVersion )
```

We still have the $ but instead we're using the software version command as our variable, and we're putting that command in parenthesis. Like in algebra, what is inside the parenthesis is calculated first, and then anything outside of the parenthesis comes next.

Try dropping the below into your command line:

```
    echo My operating system is $( sw_vers -productVersion ).
```

To make this even more useful, we'll need to use something called conditionals.

It’s only logical: conditionals build intelligence into scripts with if and then commands

Conditionals are if/then questions. If/then statements are decisionmakers: we give them the conditions that need to be met, and then tell them to evaluate whether conditions are being met.

For example:

IF I have lemons, THEN I’ll make lemonade.

You can combine variables with conditionals for very useful results.

For example, enter this into a script editor:

```
#!/bin/bash
osVersion=$( sw_vers -productVersion )
if [ $osVersion = "10.14.5" ]; then
   echo No upgrade needed.
fi
```

Starting with a variable, an administrator can call up which OS version is on the Mac.

Adding in a conditional, starting with `if`, and then adding an `echo`, will cause Terminal to check which OS version a Mac is running, and if it is the version that an administrator wants, Terminal will report back “No upgrade needed.”

Conditionals end with `fi`, or ‘if’ backward. Clever, isn’t it?

Try this in Terminal by saving this script and dragging it into the Terminal.

How about if the `if` condition is false? Simply add an `else` statement and then specify what we want to happen. Else is, of course, short for ‘or else.’

Add to your script so that it reads:

```
#!/bin/bash
osVersion=$( sw_vers -productVersion )
if [ $osVersion = "10.14.5" ]; then
   echo No upgrade needed.
else
   echo Update required.
fi
```

Now run that in Terminal.

Can you run a script not only detecting which OS version a Mac has, but also automatically installing the newest version if it’s needed?

Of course you can, with the help of Jamf Policies.

Setting Policies

In this example, we will use the Jamf Helper binary, telling the Mac to run an event: an OS upgrade.

On every enrolled Mac, Jamf Pro installs its own jamf command line, or binary tool. The `jamf` command does things specific to Jamf management like running policies. We can give any policy a unique name, called a Custom Trigger, and then use the `jamf` command line tool to call that policy by its customer trigger name — and run it. In this case, the script we will be using is going to call the runUpgrade policy, which is exactly what it sounds like: running a software upgrade.

To do this on multiple computers at the same time, you'll need to use an MDM service like Jamf pro.

To add this script to Jamf Pro, go to **Settings** > **Computer Management** > **Scripts** and then select the 'New' button.

Here, you can name It something like CheckOSVersionandRunUpgrade, leave notes about it, put the script into the 'Script' tab, provide options and limitations, and save.

Put the following in the 'Script' tab:

```
#!/bin/bash
osVersion=$( sw_vers -productVersion )
if [ $osVersion = "10.14.5" ]; then
   echo No upgrade needed.
else
   jamf policy -event runUpgrade
fi
```

Jamf Pro offers a view of scripts that highlights types of scripting, much like a script editor. Select 'shell' one time from the dropdown, and then it will default to this easier-to-read editor.

You can add in echo commands that will then be reflected back into your policy logs. They can help with troubleshooting and locating devices that have not been updated.

To create the policy, create a new policy and scope it to a Smart Group. Make it a recurring event and link it to the code you have created and named above. (For more on creating policies, please see https://docs.jamf.com/10.1.0/jamf-pro/quickstart-computers/Create_a_Policy_to_Run_Software_Update.html).

While: the conditional that bides its time

Another conditional is the 'while' statement. Instead of evaluating whether something is or isn't true, it instead waits until something IS true.

```
#!/bin/bash
while [ $( pgrep TextEdit )]
do
   echo Waiting for TextEdit to quit.
   sleep 2
done
echo TextEdit has quit.
```

The `pgrep` command determines whether or not a process is running: in this case, the TextEdit application. The next two statements tell us what SHOULD happen while TextEdit is running:

1. echo that we're waiting for the process to quit.
2. wait two seconds with the sleep command.
3. again evaluate the 'while' condition.
4. As soon as TextEdit has quit, and the 'while' condition is false, the rest of the script can continue.

Opening TextEdit, running this script in Terminal, and then quitting TextEdit will show you how this script works.

For loops: take actions on files or processes.

We're going to run a `for` loop to run a list of files through a `for` loop and it's going to rename each file in the list until it's done.

We're using a few new commands in this example:

`cd` = change directory
~ is a shortcut for the home folder and called the tilde.
`ls` = list (creates a list of items in whichever directory is selected)
`aFile` = a variable name for 'any file,' which could be any phrase you prefer.
'`for`' in this case is both setting a variable and reading a variable at the same time. It is effectively saying: For each file in the file list, do the following. Echo that we're renaming the current file, and then use the move command to actually do it.

Here, we're going to pre-pend the name of every file in the MyStuff Folder with "webinar."

Here's how we do it.

```
#!/bin/bash

cd ~/Desktop/MyStuff

filelist=$( ls )

for aFile in $filelist
do
      echo Renaming file $aFile
      mv $aFile webinar-$aFile
done
```

All of the files in your MyStuff folder should now begin with webinar-.

Take a look ahead

Don't worry. We are not expecting you to know all of the commands below! But you might be surprised by how thoroughly you now get the gist of it.

Example: Encouraging users to upgrade to Mojave

Let's say an administrator wants to encourage users to upgrade to Mojave, and then wants to automatically update Macs that haven't updated themselves at the end of a certain time period.

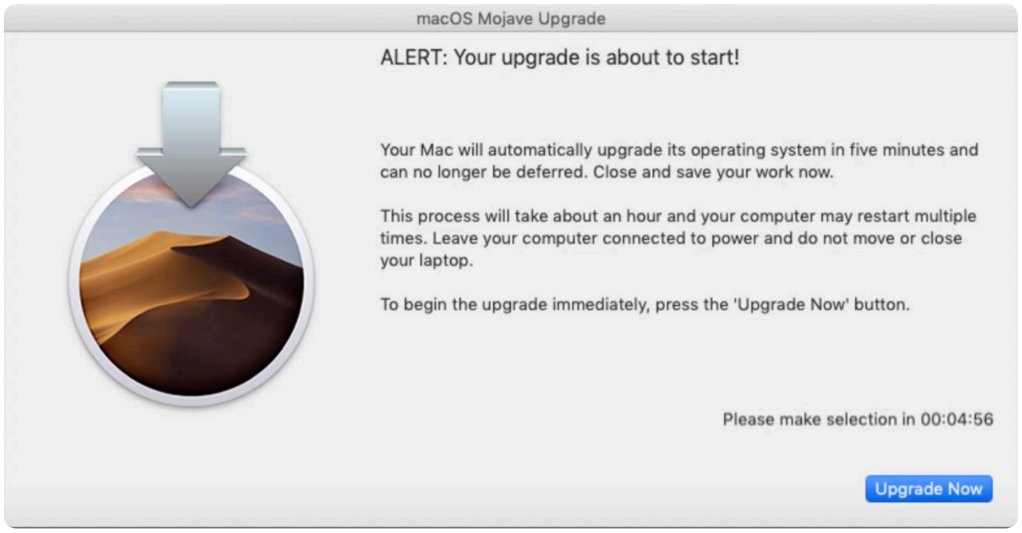
Here is what the script might look like after the warning period has ended.

```
#!/bin/bash

"/Library/Application  Support/JAMF/bin/jamfHelper.app/Contents/MacOS/jamfHelper" \
-windowType utility \
-lockHUD \
-title "macOS Mojave Upgrade" \
-heading "ALERT: Your OS upgrade is about to start!" \
-description "Your Mac will automatically upgrade its operating system in five
minutes and can no longer be deferred. Close and save your work now. The process
will take about an hour and your computer may restart multiple times. Leave your
computer connected to power and do not move or close your laptop. To begin the
upgrade immediately, select the 'Upgrade Now' button." \
-icon "/Applications/Install macOS Mojave.app/Contents/Resources/InstallAssistant.
icns" \
-iconSize 256
-button1 "Upgrade Now" \
-defaultButton 1 \
-countdown \
-timeout 300 \
-alignCountdown right

exit 0
```

This is what it looks like when it runs:



ALERT: Your upgrade is about to start!

Your Mac will automatically upgrade its operating system in five minutes and can no longer be deferred. Close and save your work now.

This process will take about an hour and your computer may restart multiple times. Leave your computer connected to power and do not move or close your laptop.

To begin the upgrade immediately, press the 'Upgrade Now' button.

Please make selection in 00:04:56

As you can see, it doesn't cover the entire screen (-iconSize 256). This allows the user the ability to click away and save their work. The countdown displays on the right side, and the default button says "Upgrade Now".

The button, or timing out, should trigger this process:

```
"/Applications/Install macOS Mojave.app/Contents/Resources/startosinstall"
--agreetolicense &
```

This script is built into most installers. This one-liner assumes Mojave is already in the folder, and the ampersand restarts the computer.

This peek into more complex code isn't meant to scare you – it's meant to show you how, with just a little bit of knowledge, you can get pretty far with scripts: automating and improving your workflows and your user's experience.

It's also to show you something else: even if you haven't built a code from scratch, you can still use it. You want to do something in a script? Google it. Someone else might have done it for you!

Which brings us to resources.

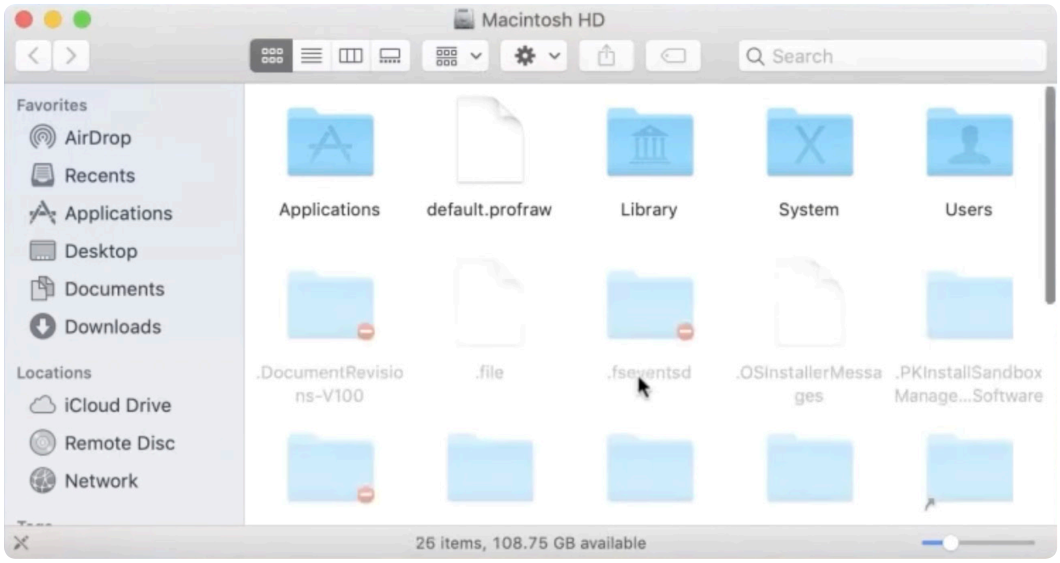
What next: Resources

Your Mac

If you'd like to discover new commands, your biggest resource is already staring you in the face: your computer is actually already a tremendous resource.

Try this:

From a Finder window, select shift ⇧, command ⌘, period . to show hidden files and folders. Commands in Terminal are, after all, backed up by something. And your computer is holding all of the instructions that come with each command in a separate file.



The first folder you should investigate is bin. Bin is short for binary. Open to see what's inside. This folder contains about 35 items: echo, bash, mkdir – some familiar, but some new.

You don't have to learn what each command does. Instead, when writing your scripts, think about what you would be doing in the finder and then seek commands that do what you need. Google is your friend! If you can do it in the Finder, there's a command line for it.

Other folders to investigate include sbin and uxr (which stands for Unix system resources). Take a look around!

Combined, you have at your fingertips more than 1000 commands.

Instructional videos

This guide is a simplified version of a Scripting for Beginners webinar. If you're more of an auditory learner, you would do well to watch it, and there are more examples to learn there: **Scripting 101 for Apple Admins**

Once you've watched the video, head on over to the blog post that follows up on it, which offers ideas for next steps and how to continue learning, with a more in-depth list of resources than in this guide: **Just 10 Commands, Then Keep Learning**

Trainingcatalog.jamf.com

We have 15 hours of short how-to videos on scripting with basic and advanced courses. Check out our scripting series, available to subscription customers.

The open source community

The open source community on **github.com/jamf** has many open-source and free scripts you may use. Don't re-invent the wheel! Look around to see if someone has already written something for your task. You will also find a terrific, involved MacAdmin community at Jamf Nation: an independent message board hosted by Jamf for Apple admins: **https://www.jamf.com/jamf-nation/**

A good script editor

We hope we've sold you on using a script editor so that you can clearly see the syntax of scripts. Here are just a few. Most are free.

bbedit: **barebones.com** (free basic)

Atom: **atom.io** (free basic)

Coderunner: **coderunnerapp.com** $14.99 as of this printing

We hope this guide has demystified scripting enough to empower you to go out and experiment, learn and improve your workflows.

While scripting alone can be helpful to you, scripting combined with a good MDM solution can really make the difference between hours and hours of work and the click of a button. We'd love to suggest Jamf Pro.

jamf | PRO

Request Trial

jamf

www.jamf.com

Or contact your preferred authorized reseller of Apple devices to take Jamf Pro for a test drive.