Arguments

So far, we’ve been developing mostly self-contained programs that don’t accept input. Now, we will look at situations that would require us to get data from the user.

Arguments are variables that the user specifies while running a script. These could be a file or folder you want to create or work with, a user name, a search query, etc. You pass them into the script by entering them after the script name.

Arguments are assigned numbered variables in the order they are given at the command line.

```
Script
echo "The $0 script got the argument $1."
```

The script’s name is held inside `$0`. The first argument is contained within `$1`. We can place those into an echo statement within our script, and return to our command line to run it.

We can feed in an argument by typing it directly after the script name. We can see that when apple is passed as the argument, it returns in the echo statement.

```
Command Line Example 1: Apple as argument

#!/usr/bin/env Bash
~$ nano myscript
~$./myscript Apple
The ./myscript script got the argument Apple.
```

```
Command Line Example 2: Banana as argument
#!/usr/bin/env Bash
~$ nano myscript
~$./myscript Banana
The ./myscript script got the argument Banana.
```

Since arguments are assigned numbered variables in the order they are given at the command line, we could include additional variables within our script. The second variable would be `$2`, the third `$3`, and so on.

```
Script w/ 2 variables
echo "The $0 script got the argument $1."
echo "Argument 2 is $2."
```

```
Command Line Example 1: Banana Cherry as arguments
#!/usr/bin/env Bash
~$ nano myscript
~$./myscript Banana Cherry
The ./myscript script got the argument Banana.
Argument 2 is Cherry.
```

Here `$0` matches to the script's name. `$1` is the first argument (Banana), and `$2` is the second argument (Cherry).

Arguments that include a space require quotes.

```
Command Line Example w/ space: "Tasty apples!" as an argument
#!/usr/bin/env Bash
~$ nano myscript
~$./myscript "Tasty apples!" Kiwi
The ./myscript script got the argument Tasty apples!
Argument 2 is Kiwi.
```

If we wanted to deal with any amount of arguments, we might have to manually create variables for each one. However, bash gives us an array of arguments which can be referenced using `$@`. We can create a for-loop to walk over the arguments that Bash gives us.

Here we create a for-loop with for `i` in `$@`, do and echo `$i` and end the loop with done.

```
Script
#!/usr/bin/env Bash
for i in $@
do
    echo $i
done
```

```
Command Line
~$ nano myscript
~$./myscript Apple Banana Cherry Durian
Apple
Banana
Cherry
Durian
```

This could be used when building scripts that handle enormous amounts of files or other data.

Additionally, the number of arguments is stored in a variable `$#`. We can use an echo statement to call that variable.

```
Script
#!/usr/bin/env Bash

for i in $@
do
        echo $i
done

echo "There were $# arguments."
```

```
Command Line
~$ nano myscript
~$./myscript Apple Banana Cherry Durian Eggplant
Apply
Banana
Cherry
Durian
Eggplant
There were 5 arguments.
```

Arguments are commonly used to pass data into a Bash script, such as lists, paths, and usernames.

Checkpoint

Which argument would be represented by the variable $3 inside of the script myscript?

```
./myscript Sneakers Loafers Sandals Cleats
```

Options

Along with arguments, we can also pass information to programs by specifying **options**.

Options:
- are a dash-letter combinations that affect how a script works.
- can use the `getopts` keyword to access these and handle each potential option.
- can take arguments but do not have to and,
- can be used in any order.

Let's look at a script that accepts a username and password.

We could use arguments for something like this, but then we would have to make sure the login and password were entered in an exact order. We can use options to help us here.

```
while getopts :u:p option; do
        case ${option} in
                u) user=$OPTARG;;
                p) pass=$OPTARG;;
        esac
done

echo "user:$user / pass: $pass"
```

In our script, we can use the `getopts` keyword. We specify an opt string (option string) that defines the search criteria.

Here, we utilize `u:p:`. This means the script will have `-u` and `-p` options. A colon after each option indicates that the script expects an argument for each.

Within the loop, we assign each option to the variable option and utilize it in a case statement.

The `OPTARG` variable holds the argument value for each option.

Then, we finish our script with an echo statement, and save.

In the command line, we run our script, allowing the option `-u` and `-p`. We can see the script is collecting the username and password from their settings and setting them to the variables.

```
Command Line
~$ nano myscript
~$ ./myscript -u scott -p secret
user: scott / pass: secret
```

Note: With options, the order doesn’t matter. The script will still work if we swap them around.

```
Command Line
~$ nano myscript
~$ ./myscript -u scott -p secret
user: scott / pass: secret
~$ ./myscript -p s3cr3t -u notscott
user: notscott / pass: s3cr3t
```

Using options, there will be a value associated with each flag, `-u` for username, and `-p` for password. But if we add more flags without colons, that means we just want to know if those flags were used.

By adding -ab script, and then echo lines in my case statement to handle them.

```
Script

while getopts u:p:ab option; do
        case $option in
                u) user=$OPTARG;;
                p) pass=$OPTARG;;
                a) echo "got the A flag";;
                b) echo "got the B flag";;
        esac
done

echo "user:$user / pass: $pass"
```

We can see a change when we run script without the A flag, then with the A flag and any other flags.

```
Command Line
~$ nano myscript
~$ ./myscript -u scott -p secret
user: scott / pass: secret
~$ ./myscript -p s3cr3t -u notscott
user: notscott / pass: s3cr3t
~$ nano myscript
~$ ./myscript -u scott -p secret
user: scott / pass: secret
~$ ./myscript -u scott -p secret -a
got the A flag
user: scott / pass: secret
~$ ./myscript -u scott -p secret -ab
got the A flag
got the B flag
user: scott / pass: secret
```

These settings are frequently used to enable or disable script functionalities. Controlling the execution without needing to update a script.

There is one more adjustment we can make to the opt string.

A colon at the start indicates to Bash that we want to know about command-line flags that haven't been provided in the script. Adding a question mark will capture those in the case statement.

```
Script

while getopts :u:p:ab option; do
        case $option in
                u) user=$OPTARG;;
                p) pass=$OPTARG;;
                a) echo "got the A flag";;
                b) echo "got the B flag";;
                ?) echo "I don't know what $OPTARG is!";;
        esac
done

echo "user:$user / pass: $pass"
```

```
Command Line
~$ nano myscript
~$ ./myscript -u scott -p secret
user: scott / pass: secret
~$ ./myscript -p s3cr3t -u notscott
user: notscott / pass: s3cr3t
~$ nano myscript
~$ ./myscript -u scott -p secret
user: scott / pass: secret
~$ ./myscript -u scott -p secret -a
got the A flag
user: scott / pass: secret
~$ ./myscript -u scott -p secret -ab
got the A flag
got the B flag
user: scott / pass: secret

~$ ./myscript -u scott -p secret -a -z
got the A flag
I don't know what z is!
user:  / pass:
```

This can be used to remind the user that they're trying to use a feature that the script isn't built for.

Options are great for passing values or specifying which functions will be utilized in a more sophisticated script.

In many situations, options are more user-friendly than arguments when writing scripts for others or oneself.

Checkpoint

Options _____.

Challenge 1

In the file `login.sh`, write a Bash program that:

- Accepts accepts a name as an argument
- Accepts a username with the option -u
- Accepts a password with the option -p
- Accepts a security key with the option -s

When the program is run, it should return the name, username, password, and security key. The result should look like the output below.

```
bash login.sh -p CrystalGemz27 -u StevenUniverse -s CookieCat
Username: StevenUniverse
Password: CrystalGemz27
Security Key: CookieCat
```

When your program works as described, submit your script below.

Input During Execution

In a script, we'll frequently need interactive input. It's not always feasible to require the user to provide all necessary arguments or options. Or, the script may not always need the same information.

In these situations, we must follow the user's process and ask for information.

To do this, we can utilize **`read`**.

If we don't specify a file to read from, the read command pauses the script until the user responds with a text string. The string is then assigned to a variable.

In our script, we can ask for a name. This will display a text prompt, wait for the user to react, and then store their response in the variable name.

```
echo "What is your name?"
read name
```

We can also accept some options. If we want a password, we can use -s for silent. This hides the characters the user types in. We can use -p for prompt and response area on one line. Since the input area will come straight after the text string, if we don't expressly leave a space, the output will look cramped.

```
echo "What is your name?"
read name
echo "What is your password?"
read -s pass
read -p "What's your favorite animal? " animal

echo "name: $name, pass: $pass, animal: $animal"
```

We then conclude the script by displaying the collected variables, save, and run in the command line.

```
Command Line

~$ nano myscript
~$ ./myscript
What is your name?
Scott
What is your password?

What's your favorite animal? Cats
name: Scott, pass: secret, animal: Cats
~$
```

Here the user will be prompted for a name, then a password. The letters will not appear because we have called the silent option (`-s`) in the script. Pressing enter will lead to an inline prompt, and finally, we'll see all the values collected.

Other options exist, and when your script requires more flexibility, be sure to investigate them by using help read.

```
Run `help read` in the terminal.
```

There's also a user-friendly menu option for input. To do so, we'll use select. Then comes the word in, followed by a list of possibilities to choose from.

```
Script

echo "Which animal"
select animal in "cat" "dog" "bird" "fish"
do
             "You selected $animal!"
             break
done
```

Then comes a `do` block encapsulating the selection. If you don't interrupt the loop (break) when you get a response, the loop will never end.

Save this, and when it runs run, we will get a nice list of prompted words, and a prompt to enter a number.

```
Command Line

~$ nano myscript
~$ ./myscript
Which animal
1) cat
2) dog
3) bird
4) fish
#?
```

```
Command Line

~$ nano myscript
~$ ./myscript
Which animal
1) cat
2) dog
3) bird
4) fish
#? 2
You selected dog!
```

This type of input works well with a case function to respond to user input.

Let’s work with the script a little bit.
We will use cat, dog, and quit. Then, we make a case using my do block. There are three options: cat, dog, and quit, and a reaction to each in a case statement. Asterisks are used in case the user enters something that doesn’t match one of the given options.

```
Script

echo "Which animal"
select animal in "cat" "dog" "quit"
do
       case $animal in
               cat) echo "Cats like to sleep.";;
               dog) echo "Dogs like to play catch.";;
               quit) break;;
               *) echo "I'm not sure what that is.";;
        esac
done
```

```
Command Line

~$ nano myscript
~$ ./myscript
Which animal
1) cat
2) dog
3) quit
#? 1
Cats like to sleep
#? 2
Dogs like to play catch.
#? 3
```

The script won't stop until the user opts to quit or escapes the script's execution with `control C`. In this case, the user can type 2 and get the response for that selection, or type 3 for quit and it breaks for that option.

In the options definition, we can insert a break statement to have the program exit after selecting an option. A double semicolon indicates the end of each set of commands. To run many commands, use a function instead of putting them all in one line. For interactive data collection, read and select offer a lot of freedom.

```
Script

echo "Which animal"
select animal in "cat" "dog" "quit"
do
       case $animal in
               cat) echo "Cats like to sleep."; break;;
               dog) echo "Dogs like to play catch.";;
               quit) break;;
               *) echo "I'm not sure what that is.";;
        esac
done
```

Checkpoint

Run

```
Script

echo "Which color?"
select animal in "Red" "Blue" "Green" "Quit"
do
       case $color in
               Red) echo "Red is the best color!";;
               Blue) echo "Blue is the best color!";;
               Green) echo "Green is the best color!";;
               Quit) break;;
               *) echo "I'm not sure what color that is.";;
        esac
done
```

```
Command Line

~$ nano myscript
~$ ./myscript
Which color?
1) Red
2) Blue
3) Green
4) Quit
#?
```

Responding

When we ask for input, the user may ignore the request. It's possible the user just presses `return` and leaves our input variable empty. This can lead to problems when we try to use the variable later in the script.

One solution to this, is using the `select` command.

We can also suggest a response to the user by adding the `-i` flag to the `read command`. If just presses `return`, the suggestion becomes the response.

Let's write a script that:
- Uses the `read` command to ask the user for their pet's name
- Suggests `Pabu` as the default response
- Saves the response in a variable called `petname`
- Display the variable in the terminal with `echo`

```
nano userResponse

#!/usr/bin/env bash

read -ep "What is your pet's name? " -i "Pabu" petname

echo $petname
```

Here, we have to use the`-e` read line interpreter. This let's us insert text into the editing buffer, the space where the user response is saved.

Let's save and run this program in the terminal.

When the prompt appears, the suggested value, `Pabu`, is already populated, keeping the response from being empty.

The user can either:
- Delete the suggested response and enter their own, or
- Press enter and accept the suggested response.

Responses Requiring arguments

There may be instances where we need to provide arguments to run a script.

Let's modify our script to check for three arguments.

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

if (($#<3>)); then
    echo "This command takes three arguments:"
    echo "petname, pettype, and petbreed."

else

    echo "Pet Name: $1"
    echo "Pet Type: $2"
    echo "Pet Breed: $3"

fi
```

▼ `#</code> </summary> <code>`# indicates the number of arguments given at the command line.

If there aren’t three arguments, the program offers the user a recommendation instead of just crashing when the variable needs to be used, but doesn’t contain any value.

Save and exit this script. Use the buttons below to run this script with 0, 2, 4, and 3 arguments.

0 Arguments

2 Arguments

4 Arguments

3 Arguments

While loops for user response

In some cases, we may want to use a loop to make sure the user cannot continue without supplying input.

Let’s create a script to:
- Ask the user for their dinner preference
- Create a while loop that:
- Uses the `-z` option to check if the variable `dinner` is empty
- Insists on a response from the user to fill the variable
- Display the user’s dinner choice to the terminal.

```
#!/usr/bin/env bash
read -p "What would you like for dinner?" dinner

while [[ -z $dinner ]]
do
    read -p "Please submit your dinner order." dinner
done

echo "You will be having $dinner for dinner!"
```

Save and exit this script. Click the button below to run it.

Explore the output and see what happens if you just press `return`.

Using the while loop, if the user ignores the prompt and presses `return`, we can assume a default response by:

- Displaying the suggested response to the user in square brackets `[]`, and
- Setting the variable to the suggested value within the while loop

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

read -p "What would you like for dinner? [Chicken Noodle Soup] "
dinner

while [[ -z $dinner ]]
do
    dinner="Chicken Noodle Soup"
done

echo "You will be having $dinner$ for dinner!"
```

Save, exit, and run this script. This is useful because the user doesn't have to erase anything to change the response, but can still accept the default by pressing `return`.

Input that does not match a regular expression can also be rejected.

Let's ask for a zip code and supply a template specifying a string of five integers.

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

read -p "Please enter a 5-digit zip code: [nnnnn]" zipcode

until [[ $zipcode =~ [0-9]{5} ]]; do
    read -p "Still waiting on that zip code! [nnnnn] " zipcode
done

echo "Your zip code is $zipcode
```

We can use a regular expression to check whether:
- the input `$zipcode`
- matches a 5-digit pattern `{5}`
- containing digits `[0-9]`

When the condition is true, the loop stops and the program carries on.

Let’s save and run this program. If the user enters a value that does not pass the regular expression test, they will be prompted until a proper input is entered.

Checking for input that matches the parameters you expect can help avoid issues later in the script.

Discovery 1

User Interaction

Explore the manual page for the `read` command and try each of the following.

1. Split the user input into different variables using multiple arguments

▼

Solution

The `-e` option returns True if a specified file exists.

```
[ -e systemSummary.txt ]; echo $?
```

2. Split the user input into different variables using multiple arguments

▼

Solution

The `-e` option returns True if a specified file exists.

```
[ -e systemSummary.txt ]; echo $?
```

3. Prompt the user for their birthday and save it to a variable called `birthday`

▼

Solution

The `-e` option returns True if a specified file exists.

```
read -p "When is your birthday?" birthday
```

4. Prompt the user for their favorite show and impose a 20 character limit.

▼

Solution

The `-e` option returns True if a specified file exists.

```
read -n 20 -p "What is your favorite show?" faveShow
```

Creating Shareable Scripts

It is not unusual for our Bash scripts to run on other people's machines. When writing our scripts, it's good to keep in mind it's compatibility with many different machines.

We can't assume that the most recent version of Bash is on absolutely every computer that will run our script. It's always a good idea to check the Bash version, as newer Bash features won't work with older Bash versions.

We can use either `$BASH_VERSION` or `$BASH_VERSINFO` to get version information about Bash. We can use this information to inform the user if their Bash version isn't compatible with your script.

If there are tools in the script that aren't standard, we may want to make sure they're available or installed before using them by using a set of tests. If they're missing, it's a good idea to print a message about the requirement for them before exiting the script so the user knows what else they need to do to run it.

The user gets a technical error if they try to execute a command that is not available. Many Bash scripts are written in a way to make them work with the Bourne Shell, which is widely used across Linux, Unix, and Mac OS systems.

Be sure that scripts you distribute are not simply for computers, but also for people. Please keep in mind that your script may be read and edited by others. So keep things simple and comment frequently.

Ultimately, the amount of effort you put into portability depends on where your script will be run. Prepare for all of the potential environments where your script will be used.

Troubleshooting

Even with the greatest care, things can go wrong.

Let’s explore a few troubleshooting techniques that might be useful when trying to figure out why a script isn’t working as expected.

Read Bash Errors

The first step is to read any errors that Bash may give you. These messages give us useful information on why Bash encountered an error.

Errors in Bash often include the line number where the program encountered an error.

We can turn on line numbering in `nano` using the `-l` or `less -N` option.

Check Quotes and Escaping

Single and double quotes work differently in scripts, so keep an eye out for errors with quote usage.

Be careful with copying and pasting material from the web or other sources, as smart quotes may be among the characters. These are considered a different character to Bash and it may not know how to handle them.

Check Test Spacing

Spacing is important when you are testing or using regular expressions in Bash.

Be sure to:
- Check for proper spacing inside expressions
- Check for proper closure of expansions and substitutions
- Check the capitalization of variables, as they are case sensitive

We can use `set -x` to have Bash display commands as they run. By using this, we can see what runs just before we receive any error messages.

Some programmers may choose to place `echo` statements throughout their script to describe processes as they occur.

Remember to remove or comment them out when things are working properly.

troubleshooting Logic

While investigating the decision branches in your script, true and false built-ins provide a successful or failing exit code.

If all else fails, keep in mind that scripts are just a collection of commands. Check each piece of your script, either at the command line or in a simplified script, to determine what works and what doesn’t.