Extracting scores with shell

There is a file in either the `start_dir/first_dir`, `start_dir/second_dir` or `start_dir/third_dir` directory called `soccer_scores.csv`. It has columns `Year,Winner,Winner Goals` for outcomes of a soccer league.

`cd` into the correct directory and use `cat` and `grep` to find who was the winner in 1959. You could also just `ls` from the top directory if you like!

Instructions
50 XP

Possible Answers

- Winner
- **Dunav**
- Botev

Searching a book with shell

There is a copy of Charles Dickens's infamous 'Tale of Two Cities' in your home directory called `two_cities.txt`.

Use command line arguments such as `cat`, `grep` and `wc` with the right flag to count the number of lines in the book that contain *either* the character 'Sydney Carton' or 'Charles Darnay'. Use exactly these spellings and capitalizations.

Instructions
50 XP

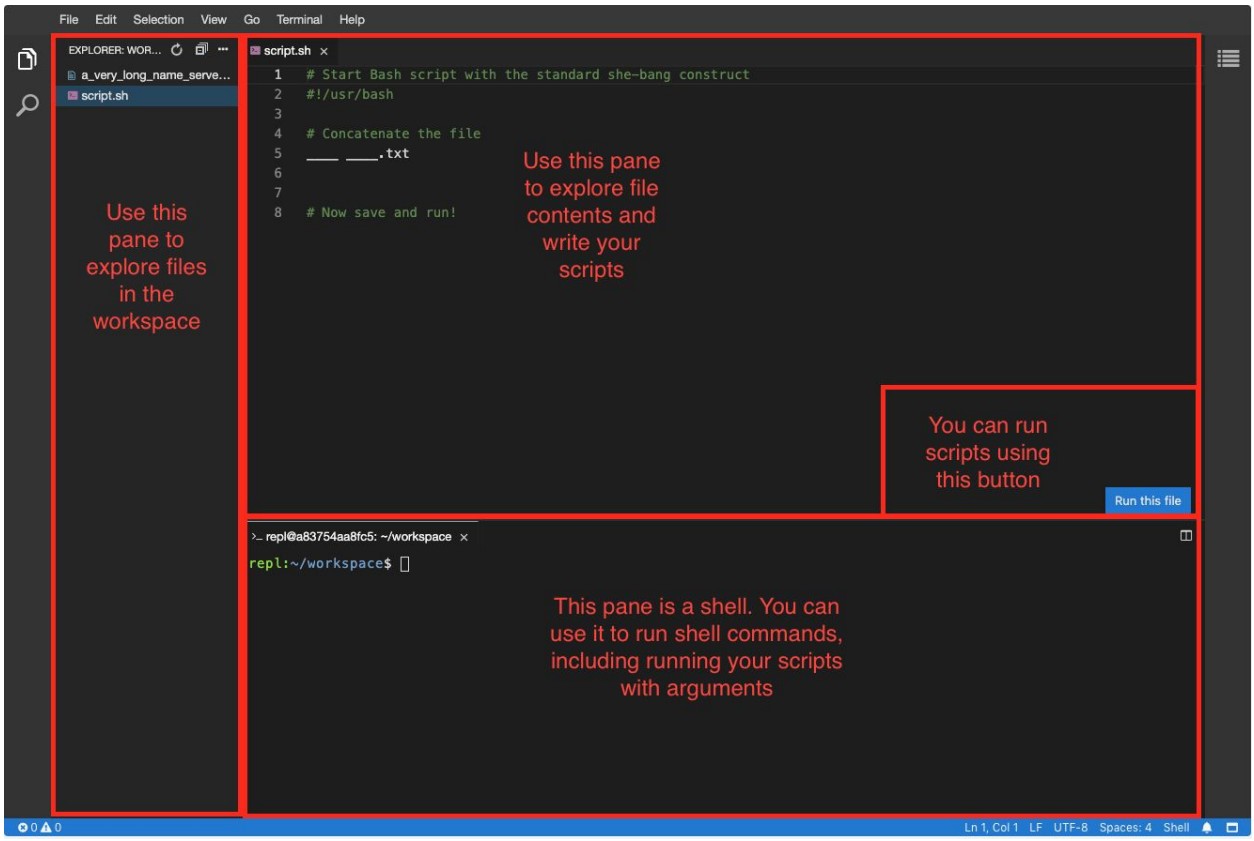
Possible Answers

- **77**
- 32
- 45

A simple Bash script

Welcome to your first IDE exercise! On the left, you have a directory of folders and several files. You can open the files and see the code in the right pane. This will auto-save every few seconds, though you can still use `cmd + s` (on mac, or `ctrl + s` on windows) to save.

You can run scripts using the `run this file` button or via the terminal window below (using the command `bash script.sh`). Running in terminal is good practice for real use cases and will be necessary later when we cover arguments.



Let's start with a very basic example to practice turning command-line (shell) arguments into a Bash script. In the provided editor, the first line has already been written for you. Remember how that was called the 'hash-bang' or 'shebang'?

There is a file in your working directory called `server_log_with_todays_date.txt`. Your task is to write a simple Bash script that concatenates this out to the terminal so you can see what is inside.

Instructions

- Create a single-line script that concatenates the mentioned file.
- Save your script and run from the console.

script.sh

```
#!/usr.bash

# Concatenate the file
cat server_log_with_todays_date.txt

# Now save and run!
```

$./bash script.sh

Shell pipelines to Bash scripts

In this exercise, you are working as a sports analyst for a Bulgarian soccer league. You have received some data on the results of the grand final from 1932 in a csv file. The file is comma-delimited in the format `Year,Winner,Winner Goals` which lists the year of the match, the team that won and how many goals the winning team scored, such as `1932,Arda,4`.

Your job is to create a Bash script from a shell piped command which will aggregate to see how many times each team has won.

Don't worry about the `tail -n +2` part, this just ensures we don't aggregate the CSV headers!

Instructions

- Create a single-line pipe to `cat` the file, `cut` out the relevant field and aggregate (`sort` & `uniq -c` will help!) based on winning team.
- Save your script and run from the console.

```
#!/usr/bash

# Create a single-line pipe
cat soccer_scores.csv | cut -d "," -f 2 | tail -n +2 | sort | uniq -c

# Now save and run!
```

Extract and edit using Bash scripts

Continuing your work for the Bulgarian soccer league - you need to do some editing on the data you have. Several teams have changed their names so you need to do some replacements. The data is the same as the previous exercise.

You will need to create a Bash script that makes use of `sed` to change the required team names.

Instructions

- Create a pipe using `sed` twice to change the team `Cherno` to `Cherno City` first, and then `Arda` to `Arda United`.
- Pipe the output to a file called `soccer_scores_edited.csv`.
- Save your script and run from the console. Try opening `soccer_scores_edited.csv` using shell commands to confirm it worked (the first line should be changed)!

```
#!/usr.bash

# Create a sed pipe to a new file
cat soccer_scores.csv | sed 's/Cherno/Cherno City/g' | sed 's/Arda/Arda United/g' >
soccer_scores_edited.csv

# Now save and run!
```

https://www.geeksforgeeks.org/sed-command-in-linux-unix-with-examples/

Using arguments in Bash scripts

Often you will find that your Bash scripts are part of an overall analytics pipeline or process, so it's very useful to be able to take in arguments (ARGV) from the command line and use these inside your scripts.

Your job is to create a Bash script that will return the arguments inputted as well as utilize some of the special properties of ARGV elements in Bash scripts.

Since we are using arguments, you must run your script from the terminal pane, not using the 'run this file' button.

Instructions

- Echo the first and second ARGV arguments.
- Echo out the entire ARGV array in one command (not each element).
- Echo out the size of ARGV (how many arguments fed in).
- Save your script and **run from the terminal pane** using the arguments `Bird Fish Rabbit`. Don't use the `./script.sh` method.

```
# Echo the first and second ARGV arguments
echo $1
echo $2

# Echo out the entire ARGV array
echo $@

# Echo out the size of ARGV
echo $#

# run it using something like $bash script.sh bird fish rabbit
```

Using arguments with HR data

In this exercise, you are working as a data scientist in the HR department of a large IT company. You need to extract salary figures for recent hires, however, the HR IT system simply spits out hundreds of files into the folder `/hire_data`.

Each file is comma-delimited in the format `COUNTRY,CITY,JOBTITLE,SALARY` such as `Estonia,Tallinn,Javascript Developer,118286`

Your job is to create a Bash script to extract the information needed. Depending on the task at hand, you may need to go back and extract data for a different city. Therefore, your script will need to take in a city (an argument) as a variable, filter all the files by this city and output to a new CSV with the city name. This file can then form part of your analytics work.

Instructions

- Echo the first ARGV argument so you can confirm it is being read in.
- `cat` all the files in the directory `/hire_data` and pipe to `grep` to filter using the city name (your first ARGV argument).

- On the same line, pipe out the filtered data to a new CSV called `cityname.csv` where `cityname` is the name of the relevant city from the first instruction.
- Save your script and run from the console twice, with the argument `Seoul` and then `Tallinn`.

```
# Echo the first ARGV argument
echo $1

# Cat all the files
# Then pipe to grep using the first ARGV argument
# Then write out to a named csv using the first ARGV argument
cat hire_data/* | grep "$1" > "$1".csv
```

Lesson Two()

Using variables in Bash

You have just joined a data analytics team at a new company after someone left very quickly to pursue a new job (lucky them!). Unfortunately they left so fast they did not have time to finish the Bash script they were working on as part of a new chatbot project.

There is an error with this script - it is printing out the words `yourname` rather than the person's name. You know this has something to do with variable assignment - can you help fix this script? Add the necessary components to ensure this script runs correctly.

Instructions

- Create a variable, `yourname` that contains the name of the user. Let's use the test name 'Sam' for this.
- Fix the echo statement so it prints the variable and not the word `yourname`.
- Run your script.

```
# Create the required variable

yourname="Sam"



# Print out the assigned name (Help fix this error!)

echo "Hi there $yourname, welcome to the website!"
```

Shell within a shell

Which of the following correctly uses a 'shell within a shell' to will print out the date? We do **not** want to select the option that will just print out the string 'date'.

You could try these in the console yourself!

Instructions
50 XP

Possible Answers

- `echo "Right now it is "date""`
- **`echo "Right now it is `date`"`**
- `echo "Right now it is $date"`

Converting Fahrenheit to Celsius

You work in the analytics department for an Australian company that recently purchased an American company. The files and data from the US company are in the imperial system and need to be converted to metric. This sounds like a great opportunity to use your Bash skills to create a program which will assist.

Your task is to write a program that takes in a single number (a temperature in Fahrenheit) as an ARGV argument, converts it to Celsius and returns the new value. There may be decimal places so you will need to undertake calculations using the `bc` program.

At all times use 2 decimal places using the `scale` command for `bc`.

The formula for Fahrenheit to Celsius is:

`C = (F - 32) x (5/9)`

Remember, since we are using arguments, you will need to run your script from the terminal pane rather than 'run this file' button.

Instructions

- Create a variable `temp_f` from the first ARGV argument.
- Call a shell-within-a-shell to subtract 32 from `temp_f` and assign to variable `temp_f2`.
- Using the same method, multiply `temp_f2` by 5 and divide by 9, assigning to a new variable `temp_c` and printing.
- Save and run your script (in the terminal) using 108 Fahrenheit (the forecast temperature in Parramatta, Sydney this Saturday!).

```
#!/usr/bash
# Get first ARGV into variable
temp_f=$1

# Subtract 32
temp_f2=$(echo "scale=2; $temp_f - 32" | bc)

# Multiply by 5/9 and print
temp_c=$(echo "scale=2; $temp_f2 * 5 / 9" | bc)

echo $temp_c
```

Extracting data from files

You are a data scientist for a climate research organization. To update some models, you need to extract temperature data for 3 regions you are monitoring. Unfortunately the temperature reading devices are quite old and can only be configured to dump data each day into a folder called `/temps` on your server. Each file contains the daily temperature for each region.

Your task is to extract the data from each file (by *concatenating*) into the relevant variable and print it out. The temperature in the file `region_A` needs to be assigned to the variable `temp_a` and so on.

You will later do some more advanced calculations on these variables.

Instructions

- Create three variables from the data in the three files within `temps` by concatenating the content into a variable using a shell-within-a-shell.
- Print out the variables to ensure it worked.
- Save your script and run from the command line.

```
#!/usr/bash
# Create three variables from the temp data files' contents
temp_a=$(cat temps/region_A)
temp_b=$(cat temps/region_B)
temp_c=$(cat temps/region_C)
```

```
# Print out the three variables
echo "The three temperatures were $temp_a, $temp_b, and $temp_c"
```

Creating an array

In this exercise, you will practice building and accessing key properties of an array. Understanding what key properties are built in to Bash is important for fully utilizing arrays. For example, when iterating through arrays, knowing their length is very handy. Similarly, knowing how to easily return all array elements is also important for looping and also for checking your work and printing.

In this exercise, you will firstly build an array using two different methods and then access the length of the array. You will then return the entire array using a different special property.

Instructions 1/3

- Create a normal array called `capital_cities` which contains the cities `Sydney`, `New York` and `Paris`. Do not use the `declare` method; fill the array as you create it. Be sure to put quotation marks around each element!

```
# Create a normal array with the mentioned elements
capital_cities=("Sydney" "New York" "Paris")
```

Instructions 2/3

- Create a normal array called `capital_cities`. However, use the `declare` method to create in this exercise.
- Below, add each city, appending to the array. The cities were `Sydney`, `New York`, and `Paris`.

```
# Create a normal array with the mentioned elements using the declare method
declare -a capital_cities

# Add (append) the elements
capital_cities+=("Sydney")
capital_cities+=("New York")
```

```
capital_cities+=("Paris")
```

Instructions 3/3
30 XP

- 3
  - Now you have the array created, print out the entire array using a special array property.
  - Then print out the length of the array using another special property.

```
# The array has been created for you
capital_cities=("Sydney" "New York" "Paris")

# Print out the entire array
echo ${capital_cities[@]}

# Print out the array length
echo ${#capital_cities[@]}
```

Creating associative arrays

Associative arrays are powerful constructs to use in your Bash scripting. They are very similar to 'normal' arrays, however they have a few important differences in their creation, manipulation and key properties.

Associative arrays allow you to index using words rather than numbers, which can be important for ease of inputting and accessing properties. For example, rather than accessing 'index 4' of an array about a city's information, you can access the `city_population` property, which is a lot clearer!

In this exercise we will practice creating and adding to an associative array. We will then access some special properties that are unique to associative arrays. Let's get started!

Instructions 1/3

- Create an empty associative array on one line called `model_metrics`. Do not add any elements to it here.

- Add the following key-value pairs; (`model_accuracy`, `98`), (`model_name`, `"knn"`), (`model_f1`, `0.82`).

```
# Create empty associative array
declare -A model_metrics

# Add the key-value pairs
model_metrics[model_accuracy]=(98)
model_metrics[model_name]=("knn")
model_metrics[model_f1]=(.82)
```

Instructions 2/3
1 XP

- Now create an associative array called model_metrics and add the key-value pairs all in one line. (`model_accuracy`, `98`), (`model_name`, `"knn"`), (`model_f1`, `0.82`).
- Print out the array to see what you created.

```
# Declare associative array with key-value pairs on one line
declare -A model_metrics=([model_accuracy]=98 [model_name]="knn" [model_f1]=0.82)

# Print out the entire array
echo ${model_metrics[@]}
```

Instructions 3/3
30 XP

- Now that you've created an associative array, print out just the keys of this associative array.

```
# An associative array has been created for you
declare -A model_metrics=([model_accuracy]=98 [model_name]="knn" [model_f1]=0.82)

# Print out just the keys
echo ${!model_metrics[@]}
```

Climate calculations in Bash

You are continuing your work as a data scientist for the climate research organization. In a previous exercise, you extracted temperature data for 3 regions you are monitoring from some files from the `/temps` directory.

Remember, each file contains the daily temperature for each region. However, only two regions will be used in this exercise.

In this exercise, the lines from your previous exercise are already there which extract the data from each file. However, this time you will then store these variables in an array, calculate the average temperature of the regions and append this to the array.

For example, for temperatures of 60 and 70, the array should have 60, 70, and 65 as its elements.

Instructions

- Create an array with the two `temp` variables as elements.
- Call an external program to get the average temperature. You will need to sum array elements then divide by 2. Use the `scale` parameter to ensure this is to 2 decimal places.
- Append this new variable to your array and print out the entire array.
- Run your script.

```
# Create variables from the temperature data files
temp_b="$(cat temps/region_B)"
temp_c="$(cat temps/region_C)"

# Create an array with these variables as elements
region_temps=($temp_b $temp_c)

# Call an external program to get average temperature
average_temp=$(echo "scale=2; (${region_temps[0]} + ${region_temps[1]}) / 2" | bc)

# Append average temp to the array
region_temps+=($average_temp)

# Print out the whole array
echo ${region_temps[@]}
```

Lesson Three(Control Statements in Bash Scripting)

Sorting model results

You are working as a data scientist in a large corporation. The production environment for your machine learning models writes out text files into the `model_results/` folder whenever an experiment is completed. The files have the following structure (example):

```
Model Name: KNN
Accuracy: 89
F1: 0.87
Date: 2019-12-01
ModelID: 34598utjfddfgdg
```

You can see the model name, accuracy and F1 scores, the date the experiment completed and a unique ID to link the model experiment back into your experiment system.

The company has a threshold of 90% for accuracy for a model to continue experimentation. Your task is to write a Bash script that takes in an ARGV argument (a filename), extracts the accuracy score and conditionally sorts the model result file into one of two folders: `good_models/` for those with accuracy *greater than or equal to* 90 and `bad_models/` for those less than 90.

Instructions

- Create a variable, `accuracy` by extracting the accuracy line (and accuracy value) in the first ARGV element (a file).
- Create an IF statement to move the file into `good_models/` folder if it is greater than or equal to 90 using a flag, not a mathematical sign.
- Create an IF statement to move the file into `bad_models/` folder if it is less than 90 using a flag, not a mathematical sign.
- Run your script from the terminal pane twice, feeding each name of the first two files `model_results` (`model_1.txt`, `model_2.txt`) respectively.

```
#!/usr/bash

# don't run this file....
```

```
# Extract accuracy from first ARGV element
accuracy=$(grep "Accuracy" $1 | sed 's/.* //')

# Conditionally move into good_model folder
if [ $accuracy -ge 90 ]; then
  mv $1 good_models/
fi

# Conditionally move into bad_model folder
if [ $accuracy -lt 90 ]; then
  mv $1 bad_models/
fi
```

Moving relevant files

You have recently joined a new startup as one of the few technical employees. Your manager has asked if you can assist to clean up some of the folders on the server. The company has been through a variety of server monitoring software and so there are many files that should be deleted.

Luckily you know that all the files to keep contain both `vpt` and `SRVM_` inside the file somewhere.

Your task is to write a Bash script that will take in file names as ARGV elements and move the file to `good_logs/` if it matches both conditions above.

Instructions

- Create a variable `sfile` out of the first ARGV element.
- Use an IF statement and `grep` to check if the first ARGV element contains `SRVM_` AND `vpt` inside.
- Inside the IF statement, move matching files to the `good_logs/` directory.
- Try your script on all of the files in the directory. It should move only one of them.

```
#!/usr/bash

#don't run this file.....
```

```
# Create variable from first ARGV element
sfile=$1

# Create an IF statement on first ARGV element's contents
if grep -q 'SRVM_' $sfile && grep -q 'vpt' $sfile ; then
  # Move file if matched
  mv $sfile good_logs/
fi
```

A simple FOR loop

You are working as a data scientist in an organization. Due to a recent merge of departments, you have inherited a folder with many files inside. You know that the `.R` scripts may be useful for your work but you aren't sure what they contain.

Write a simple Bash script to loop through all the files in the directory `inherited_folder/` that end in `.R` and print out their names so you can get a quick look at what sort of scripts you have. Hopefully the file names are useful!

Instructions

- Use a FOR statement to loop through files that end in `.R` in `inherited_folder/` using a glob expansion.
- `echo` out each file name into the console.

```
# Use a FOR loop on files in directory
for file in inherited_folder/*.R
do
  # Echo out each file
  echo $file
done
```

Correcting a WHILE statement

There is a script placed in this directory which has a mistake in it. It is some code to modify some employee files, but only for the first 1000 employees. Do not attempt to run the script as the necessary files are not in this directory.

Using your knowledge of WHILE scripts, can you determine where the mistake is? What will happen if it runs?

You can find the code by using `cat emp_script.sh` to print it to the console.

Instructions

Possible Answers

- There is no mistake, this script will run just fine.
- **It will run forever because `emp_num` isn't incremented inside the loop.**
- You cannot `cat` a `.txt` file so this will fail.

```
#!/usr/bash

employee_number=1
while [ $emp_num -le 1000 ];
do
  cat "$emp_num-dailySales.txt" | egrep 'Sales_total' | sed 's/.* ://' > "$emp_num-agg.txt"
done
```

Cleaning up a directory

Continuing your work as a data scientist in a large organization, you were told today that a colleague has left for their dream job (lucky them!). Unfortunately, when their logins were terminated, all their files were dumped into a single folder.

The good news is that most of their useful code has been backed up. However, all their python files using the Random Forest algorithm are buried in the file dump.

The task has fallen to you to sift through the hundreds of files to determine if they are both Python files and contain a Random Forest model. This sounds like a perfect opportunity to use your Bash skills, rather than checking every single file manually.

Write a script that loops through each file in the `robs_files/` directory to see if it is a Python file (ends in `.py`) AND contains `RandomForestClassifier`. If so, move it into the `to_keep/` directory.

Instructions

- Use a FOR statement to loop through (using glob expansion) files that end in `.py` in `robs_files/`.
- Use an IF statement and `grep` (remember the 'quiet' flag?) to check if `RandomForestClassifier` is in the file. Don't use a shell-within-a-shell here.
- Move the Python files that contain `RandomForestClassifier` into the `to_keep/` directory.

```
# Create a FOR statement on files in directory
for file in robs_files/*
do
  # Create IF statement using grep
  if grep -q 'RandomForestClassifier' $file ; then
    # Move wanted files to to_keep/ folder
    mv $file to_keep/
  fi
done
```

Days of the week with CASE

In your role as a Data Scientist, it is sometimes useful to associate dates with a 'working day' (Monday, Tuesday, Wednesday, Thursday, Friday) or a 'weekend' (Saturday & Sunday).

Your task is to build a small Bash script that will be useful to call in many areas of your data pipeline. It must take in a single argument (string of a day) into ARGV and use a CASE statement to print out whether the argument was a weekday or a weekend. You only need to account for the capitalized case for now.

You also don't need to worry about words or letters before and after. Just use exact matching for this example.

Remember the basic structure of a case statement is:

```
case MATCHVAR in
  PATTERN1)
  COMMAND1;;
  PATTERN2)
  COMMAND2;;
  *)
  DEFAULT COMMAND;;
esac
```

Instructions

- Build a CASE statement that matches on the first ARGV element.
- Create a match on each weekday such as `Monday`, `Tuesday` etc. using OR syntax on a single line, then a match on each weekend day (`Saturday` and `Sunday`) etc. using OR syntax on a single line.
- Create a default match that prints out `Not a day!` if none of the above patterns are matched.
- Save your script and run in the terminal window with `Wednesday` and `Saturday` to test.

```
# Create a CASE statement matching the first ARGV element
case $1 in
 # Match on all weekdays
 Monday|Tuesday|Wednesday|Thursday|Friday)
 echo "It is a Weekday!";;
 # Match on all weekend days
 Saturday|Sunday)
 echo "It is a Weekend!";;
 # Create a default
 *)
 echo "Not a day!";;
esac
```

Moving model results with CASE

You are working as a data scientist in charge of analyzing some machine learning model results. The production environment moves files into a folder called `model_out/` and names them `model_RXX.csv` where `XX` is a random number related to which experiment was run.

Each file has the following structure (example):

```
Model Name, Accuracy, CV, Model Duration (s)
Logistic,42,4,48
```

Your manager has told you that recent work in the organization has meant that tree-based models are to be kept in one folder and everything else deleted.

Your task is to use a CASE statement to move the tree-based models (`Random Forest`, `GBM`, and `XGBoost`) to the `tree_models/` folder, and delete all other models (`KNN` and `Logistic`).

Instructions

- Use a FOR statement to loop through (using glob expansion) files in `model_out/`.
- Use a CASE statement to match on the contents of the file (we will use `cat` and shell-within-a-shell to get the contents to match against). It must check if the text contains a tree-based model name and move to `tree_models/`, otherwise delete the file.
- Create a default match that prints out `Unknown model in FILE` where FILE is the filename.

```
# Use a FOR loop for each file in 'model_out/'
for file in model_out/*
do
  # Create a CASE statement for each file's contents
  case $(cat $file) in
    # Match on tree and non-tree models
    *"Random Forest"*|*GBM*|*XGBoost*)
    mv $file tree_models/ ;;
    *KNN*|*Logistic*)
```

```
    rm $file ;;
    # Create a default
    *)
    echo "Unknown model in $file" ;;
  esac
done
```

Finishing a CASE statement

What is the correct way to finish a CASE statement?

Answer the question

Possible Answers

- `esac`
- `fi`
- `done`

Lesson Four(Functions and Automation)

Uploading model results to the cloud

You are working as a data scientist on building part of your machine learning cloud infrastructure. Your team has many machine learning experiments running all the time. When an experiment is finished, it will output the results file and a configuration file into a folder called `output_dir/`.

These results need to be uploaded to your cloud storage environment for analysis. Your task is to write a Bash script that contains a function which will loop through all the files in `output_dir/` and upload the result files to your cloud storage.

For technical reasons, no files will be uploaded; we will simply echo out the file name. Though you could easily replace this section with code to upload to Amazon S3, Google Cloud or Microsoft Azure!

Instructions

- Set up a function using the 'function-word' method called `upload_to_cloud`.
- Use a FOR statement to loop through (using glob expansion) files whose names contain `results` in `output_dir/` and `echo` that the filename is being uploaded to the cloud.
- Call the function just below the function definition using its name.

```
# Create function
function upload_to_cloud () {
 # Loop through files with glob expansion
 for file in output_dir/*results*
 do
   # Echo that they are being uploaded
   echo "Uploading $file to cloud"
 done
}

# Call the function
upload_to_cloud
```

Get the current day

Whilst working on a variety of Bash scripts in your data analytics infrastructure, you find that you often need to get the current day of the week. You can see this is unnecessary duplication of code and can be refactored (reduce the size by writing more efficient code) using your new skills in Bash functions.

Write a simple Bash function that, when called, will simply print out the current day of the week. This will involve parsing the output of the `date` program from a shell-within-a-shell.

A reminder that the output of `date` will look something like this (depending on the timezone you are calling it from!)

```
Fri 27 Dec 2019 14:24:33 AEDT
```

You want to extract the `Fri` part only.

Instructions

- Set up a function called `what_day_is_it` **without** using the word `function` (as you did using the function-word method).
- Parse the output of `date` into a variable called `current_day`. The extraction component has been done for you.
- Echo the result.
- Call the function just below the function definition.

```
# Create function
what_day_is_it () {

 # Parse the results of date
 current_date=$(date | cut -d " " -f1)

 # Echo the result
 echo $current_date
}

# Call the function
what_day_is_it
```

A percentage calculator

In your work as a data scientist, you often find yourself needing to calculate percentages within Bash scripts. This would be a great opportunity to create a nice modular function to be called from different places in your code.

Your task is to create a Bash function that will calculate a percentage of two numbers that are given as arguments and return the value.

A test example you can think of to use in this script is a model that you just ran where there were 456 correct predictions out of 632 on the test set.

Remember that the shell can't natively handle decimal places, so it will be safer to employ the use of `bc`.

Instructions

- Create a function called `return_percentage` using the function-word method.
- Create a variable inside the function called `percent` that divides the first argument fed into the function by the second argument.
- Return the calculated value by echoing it back.
- Call the function with the mentioned test values of 456 (the first argument) and 632 (the second argument) and echo the result.

```
# Create a function
function return_percentage () {

 # Calculate the percentage using bc
 percent=$(echo "scale=4; $1 / $2" | bc)

 # Return the calculated percentage
 echo $percent
}

# Call the function with 456 and 632 and echo the result
return_test=$(return_percentage 456 632)
echo "456 out of 632 as a percent is $return_test"
```

Sports analytics function

You have been contracted back to the a soccer league to help them with some sports analytics. You notice that a number of the scripts undertake aggregations, just like you did in a previous exercise. Since there is a lot of duplication of code, this is an excellent opportunity to create a single useful function that can be called at many places in the script.

Your task is to create a Bash function that will take in a city name and find out how many wins they have had since recording began.

Inside the main function, this script will call out to a shell-within-a-shell which is captured in a (by default, global) variable. You can then access this variable outside the function. This was the first technique discussed in the video for getting data out of a function.

Most of the shell pipeline used has been done for you, though of course feel free to explore and understand what is happening here. Nothing there should be new to you!

Instructions

- Create a function called `get_number_wins` using the function-word method.
- Create a variable inside the function called `win_stats` that takes the argument fed into the function to filter the last step of the shell-pipeline presented.
- Call the function using the city `Etar`.
- Below the function call, try to access the `win_stats` variable created inside the function in the `echo` command presented.

```
# Create a function

function get_number_wins () {



 # Filter aggregate results by argument

 win_stats=$(cat soccer_scores.csv | cut -d "," -f2 | grep -v 'Winner'| sort | uniq -c | grep $1)



}



# Call the function with specified argument

get_number_wins "Etar"
```

```
# Print out the global variable

echo "The aggregated stats are: $win_stats"
```

Summing an array

A common programming task is obtaining the sum of an array of numbers. Let's create a function to assist with this common task.

Create a Bash function that will take in an array of numbers and return its sum. We will use `bc` rather than `expr` to ensure we can handle decimals.

Your company's security rules state that all variables in functions must be restricted local scope so you will need to keep this in mind.

An array of numbers you can use for a test of you function would be the daily sales in your organization this week (in thousands):

`14 12 23.5 16 19.34` which should sum to `84.84`

Instructions

- Create a function called `sum_array` and add a base variable (equal to 0) called `sum` with **local** scope. You will loop through the array and increment this variable.
- Create a FOR loop through the ARGV array inside `sum_array` (hint: This is **not** `$1`! but another special array property) and increment `sum` with each element of the array.
- Rather than assign to a global variable, `echo` back the result of your FOR loop summation.
- Call your function using the test array provided and echo the result. You can capture the results of the function call using the shell-within-a-shell notation.

```
# Create a function with a local base variable

function sum_array () {

 local sum=0
```

```
 # Loop through, adding to base variable
 for number in "$@"
 do
   sum=$(echo "$sum + $number" | bc)
 done
 # Echo back the result
 echo $sum
 }
# Call function with array
test_array=(14 12 23.5 16 19.34)
total=$(sum_array "${test_array[@]}")
echo "The sum of the test array is $total"
```

Analyzing a crontab schedule

Using your knowledge of the structure of a crontab schedule, can you determine how often the `script.sh` Bash script will run if the following line is in the user's crontab?

```
15 * * * 6,7 bash script.sh
```

Answer the question

Possible Answers

- 15 minutes past every hour for every day in June and July.
- **15 minutes past every hour on Saturdays and Sundays.**
- Run for 15 days then stop for 6 or 7 days, then resume.

Creating cronjobs

You are working as a data scientist managing an end-to-end machine learning environment in the cloud. You have created some great Bash scripts but t is becoming tedious to have to run these scripts every morning and afternoon. You recently learned about `cron` which you think can greatly assist here!

An example file has been placed in your directory where you can create some crontab jobs.

Remember that a crontab schedule has 5 stars relation to the time periods minute, hour, day-of-month, month-of-year, day-of-week.

Note that where all time periods are not specified in the instructions below, you can assume those time periods are 'every' (`*`).

Don't try to run the scripts or use crontab as neither will work.

A useful tool for constructing crontabs is **https://crontab.guru/**.

Instructions
100 XP

- Create a crontab schedule that runs `script1.sh` at 30 minutes past 2am every day.
- Create a crontab schedule that runs `script2.sh` every 15, 30 and 45 minutes past the hour.
- Create a crontab schedule that runs `script3.sh` at 11.30**pm** on Sunday evening, every week. For this task, assume Sunday is the 7th day rather than 0th day (as in some unix systems).

```
# Create a schedule for 30 minutes past 2am every day
30 2 * * * bash script1.sh

# Create a schedule for every 15, 30 and 45 minutes past the hour
15,30,45 * * * * bash script2.sh

# Create a schedule for 11.30pm on Sunday evening, every week
30 23 * * 7 bash script3.sh
```

Scheduling cronjobs with crontab

As the resident data scientist, you have been asked to automate part of a data processing pipeline. You will use one of the `cronjob` schedules created in a previous exercise.

You will use the built in `crontab` functionality of this workspace to schedule a data pipeline script, `extract_data.sh` to run at 30 minutes past 2am every day. Running early in the morning saves a great amount of cost by utilizing cheaper server power.

You will end up in the `nano` text editor where you save a file with `ctrl` + `o` (press enter) and exit with `ctrl` + `x`. Useful Nano documentation is **here**.

At any stage you can refresh your browser window and you will get a fresh workspace to try again if you accidentally make a mistake in the workspace.

Instructions 1/3

- Verify there are no existing `cronjobs` by listing them.

repl:~$ crontab -l

This is your crontab file.

Schedule your crontabs below.

Instructions 1/3

- Use the edit command for `crontab` (select `nano`) then schedule `extract_data.sh` to run with Bash at 2:30am every day.

repl:~$ crontab -e

Select an editor. To change later, run 'select-editor'.
1. /bin/nano <---- easiest
2. /usr/bin/vim.basic

Choose 1-2 [1]: 1
Inside nano:-->

```
# This is your crontab file.
# Schedule your crontabs below.
30 2 * * * bash extract_data.sh
```

```
crontab: installing new crontab
repl:~$ cronjobs
bash: cronjobs: command not found
repl:~$ crontab -l
# This is your crontab file.
# Schedule your crontabs below.
30 2 * * * bash extract_data.sh
repl:~$
```