School of
Science

Scripting Languages

Module 3

Working with Data in bash

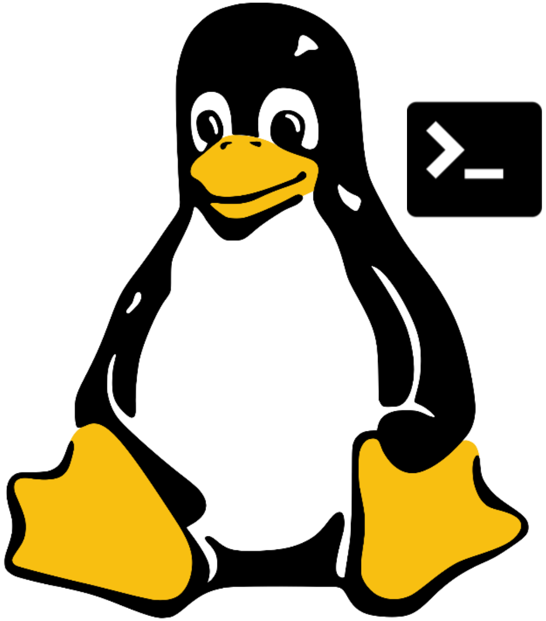
Contents

1. Acquiring User Input
2. Conditional Testing
3. Using Arrays
4. Doing Basic Maths in bash
5. Code Commenting
6. Resources

Learning Objectives

By the end of this Module you will :

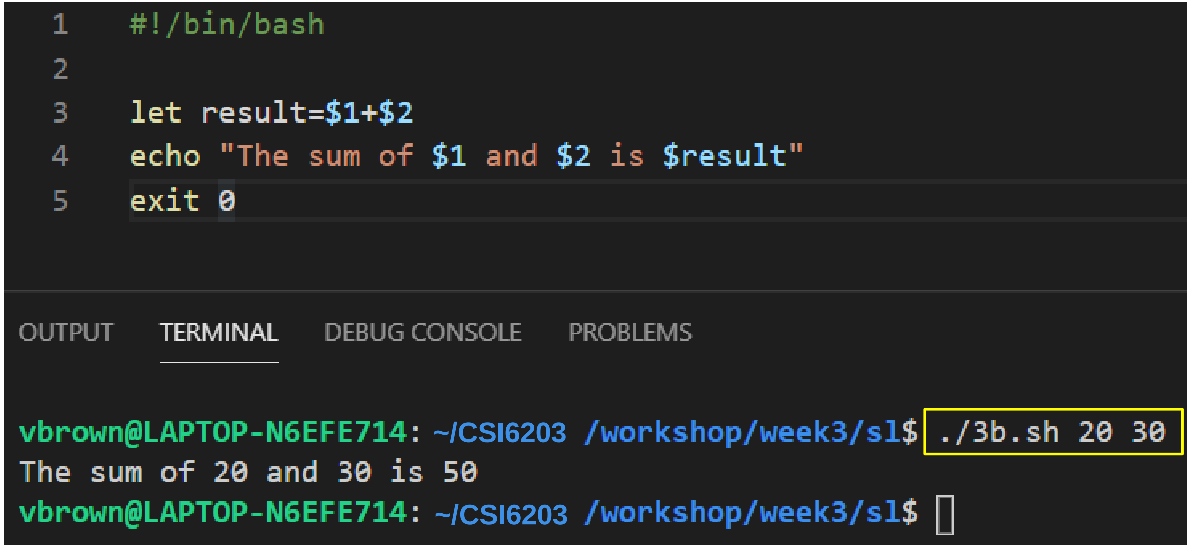
1. Be familiar with various methods of acquiring user input for script execution
2. Know how to apply a range of conditional testing techniques to build decision-making capability into your scripts
3. Know how to store set of data in, and work with arrays
4. Know how to perform basic mathematical operations in bash
5. Understand the importance of code commenting and how to place comments into your code



Acquiring User Input

User Input Using Command Line Arguments

- Command-line arguments like $1, $2, $3, etc is one way to make scripts interactive



```
#!/bin/bash

let result=$1+$2
echo "The sum of $1 and $2 is $result"
exit 0
```

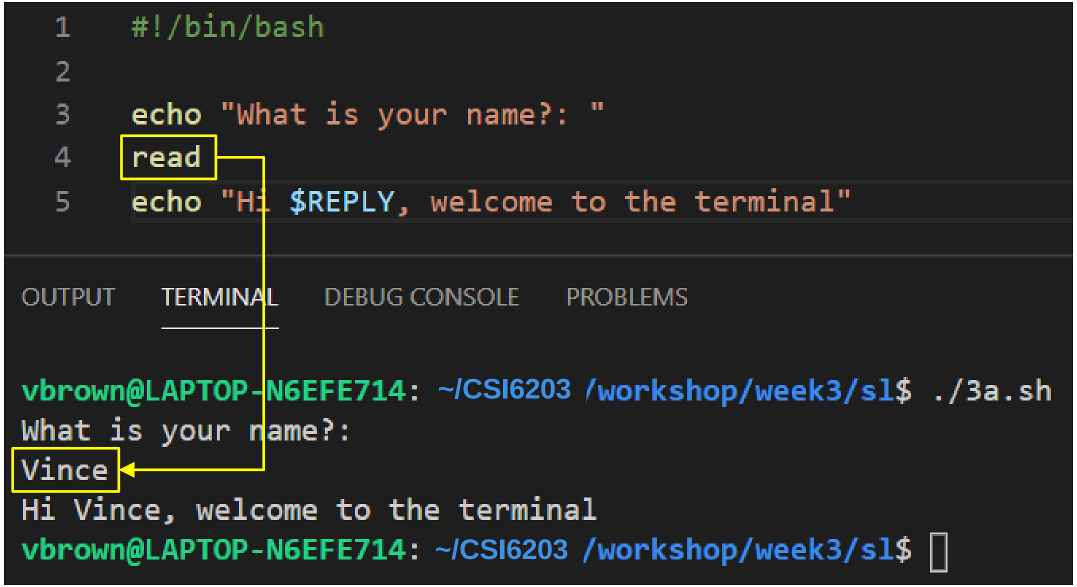
```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3b.sh 20 30
The sum of 20 and 30 is 50
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ 
```

Shell Script

Output in Terminal

Using *read* for input

- Whatever the user has typed in when the script hits the read line will be stored in the $REPLY variable by default which can then be used by subsequent lines of the script



```bash
#!/bin/bash

echo "What is your name?: "
read
echo "Hi $REPLY, welcome to the terminal"
```

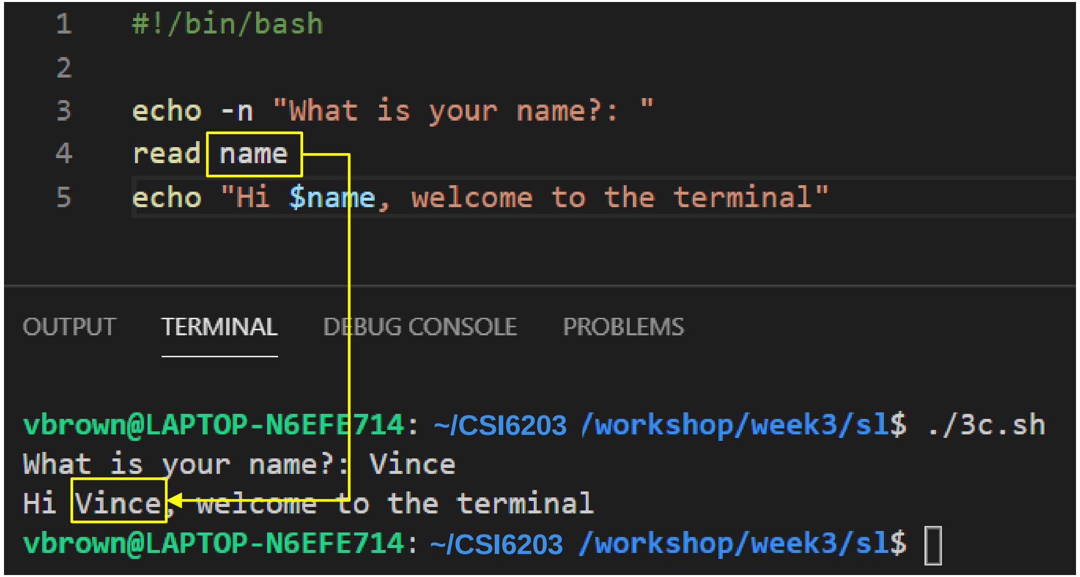
```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3a.sh
What is your name?:
Vince
Hi Vince, welcome to the terminal
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$
```

Shell Script

Output in Terminal

Providing read with a variable

- Instead of relying on the default $REPLY variable, a *custom variable* can be specified instead immediately after the read command



```
#!/bin/bash

echo -n "What is your name?: "
read name
echo "Hi $name, welcome to the terminal"
```

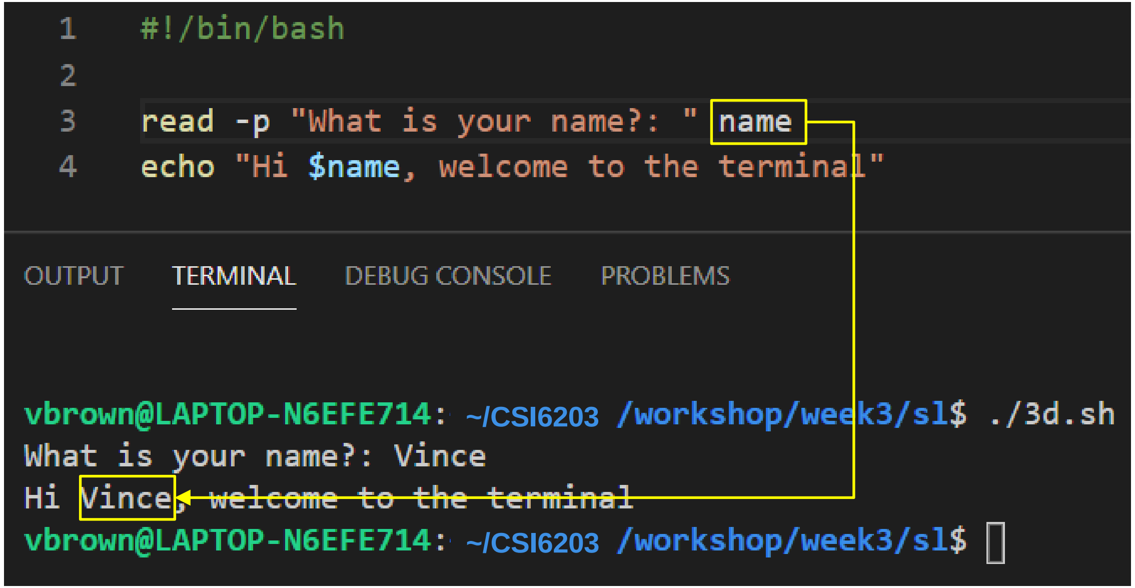
```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3c.sh
What is your name?: Vince
Hi Vince, welcome to the terminal
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ 
```

Shell Script

Output in Terminal

Using read –p for input

- Using **read** with **–p** can allow the read command to print a prompt, removing the need for a separate echo statement



```
#!/bin/bash

read -p "What is your name?: " name
echo "Hi $name, welcome to the terminal"
```

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3d.sh
What is your name?: Vince
Hi Vince, welcome to the terminal
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$
```

Shell Script

Output in Terminal

Using read –s to hide input

- The input text can also be hidden for sensitive information using the **–s** option

```
#!/bin/bash
read -p "What is your name?" name
echo "Hello $name"
read -s -p "What is your password?" pass
echo "the secret is: $pass"

read –n1 -p "Press any key to continue"
```

Escape Sequences

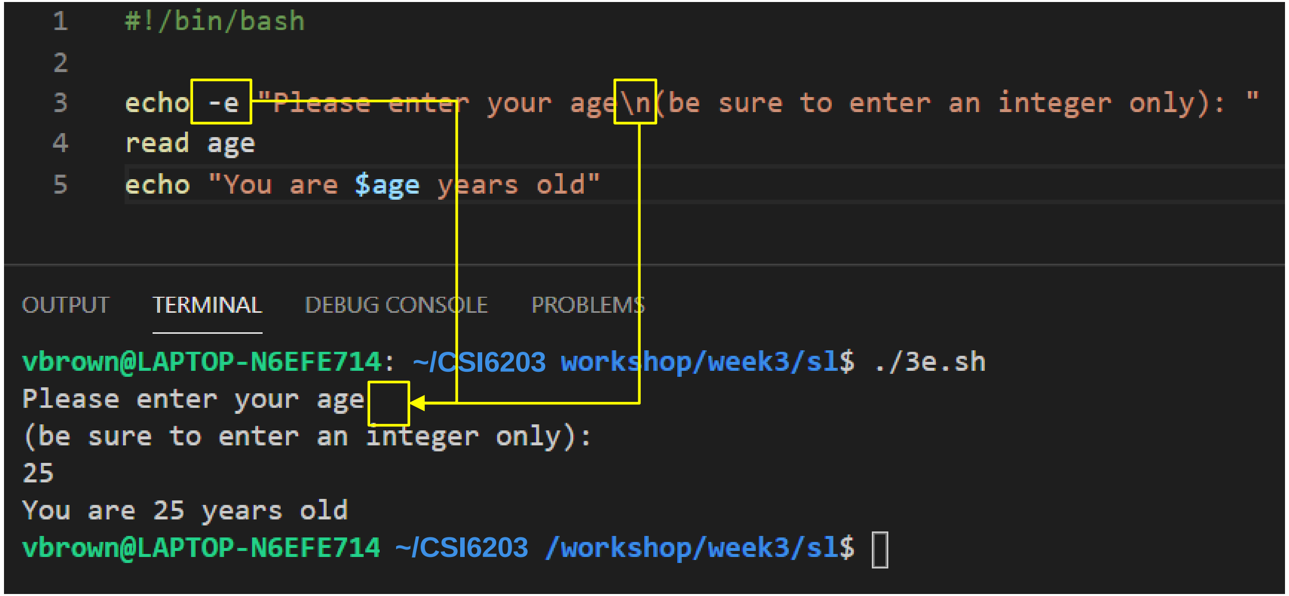
- Some characters are difficult to type in a script due to their use in the text editor, such as Enter, Tab, CRTL-C and Backspace
- These characters can be printed using *escape sequences*
- An example of an escape sequence is **\n** for *Newline*
- Escapes can be enabled in the echo command using the **-e** option

Escape Sequence	Output
\b	Backspace
\c	Remove extra output (such as newlines)
\n	Newline
\r	Return to start of line (Carriage Return)
\t	Tab character
\v	Vertical Tab character
\\	Backslash (\ )

Common Escape Sequences

Escape Sequences

- Escapes can be used to format outputs for example:

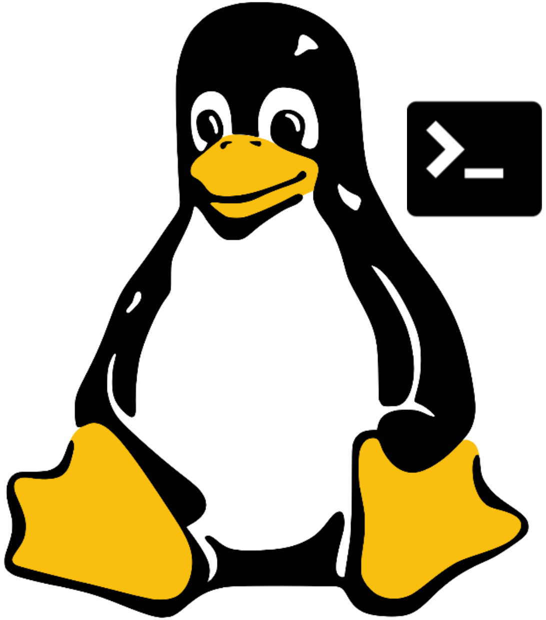


Shell Script

```bash
#!/bin/bash

echo -e "Please enter your age\n(be sure to enter an integer only): "
read age
echo "You are $age years old"
```

Output in Terminal

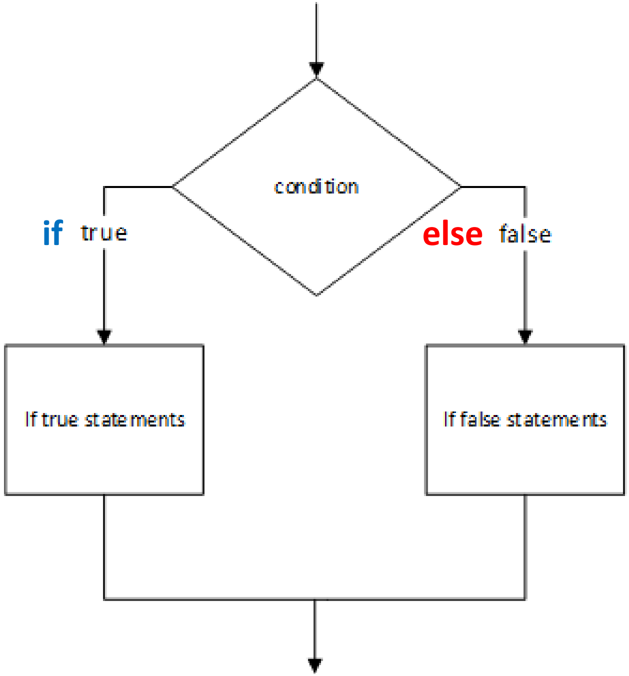
```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 workshop/week3/sl$ ./3e.sh
Please enter your age
(be sure to enter an integer only):
25
You are 25 years old
vbrown@LAPTOP-N6EFE714 ~/CSI6203 /workshop/week3/sl$ 
```



Conditional Testing

Conditional Tests in Scripts

- Conditional Tests are used to provide scripts the ability to make decision based on one or more criterion
- They allow you to make a script more useful in that it can execute only specific blocks of code as required instead of the whole script from beginning to end
- This approach makes scripts more robust, easier to use, and more reliable
- Conditional statements can be written with simple *command-line lists* of **AND** or **OR** commands combined together, or within the highly versatile **if** statement



Command-line Lists

- Command-line lists are some of the simplest forms of decision making
- They behave in a similar way to boolean operators in programming languages
- In bash **&&** acts as AND
- In bash **||** acts as OR

&&

- The && (AND) operator can be used to link multiple commands together
- The second command will only execute if the first command succeeds, and the third command only of the second succeeds, and so on

Command-line Lists

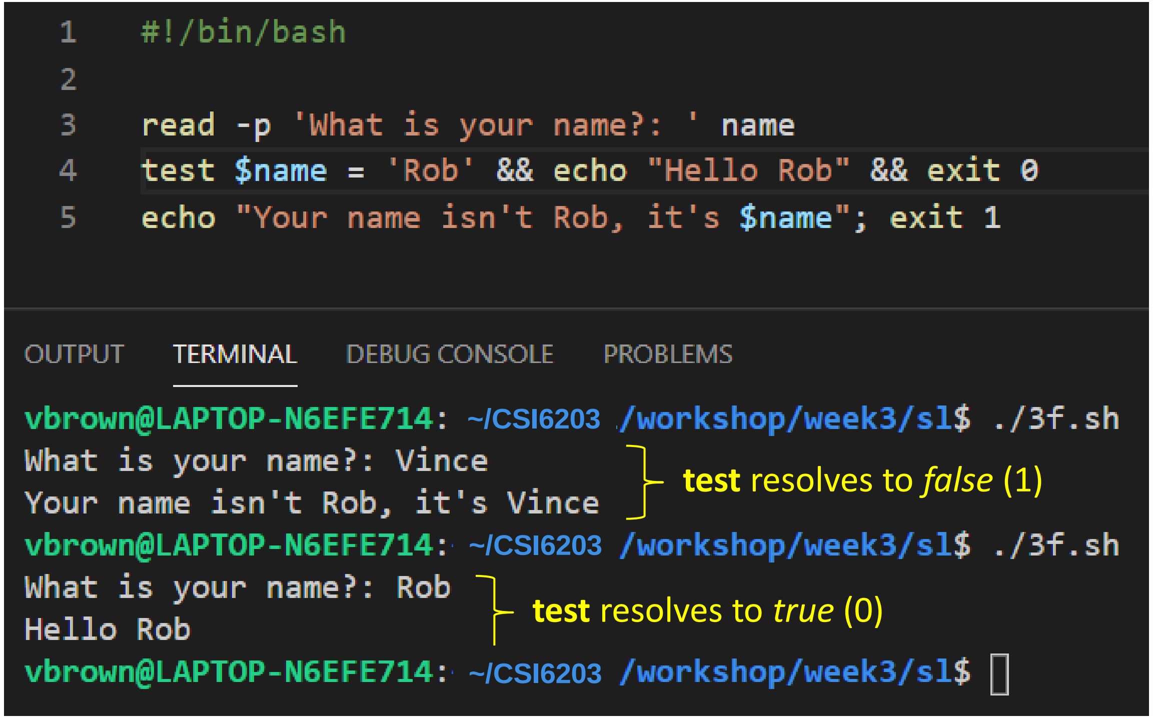
- The || (OR) operator can also be used to link multiple commands together
- The second command will only execute if the first command fails, and the third will only execute if the second fails, and so on

||

- The test command can be used to evaluate a true or false expression
- The test will succeed (return an exit status of 0) if the provided expression is true
- test will fail (return an exit status of 1) if the provided expression is false

test

- Command-line list expression using **test**:



Shell Script

```bash
#!/bin/bash

read -p 'What is your name?: ' name
test $name = 'Rob' && echo "Hello Rob" && exit 0
echo "Your name isn't Rob, it's $name"; exit 1
```

Output in Terminal

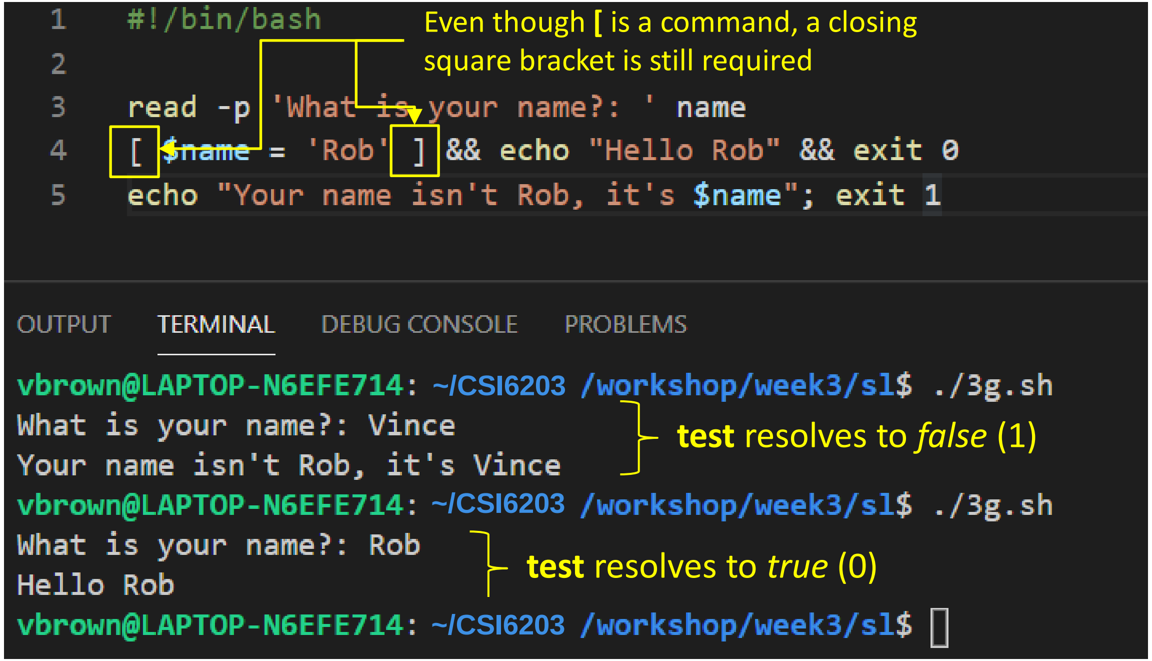
```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3f.sh
What is your name?: Vince
Your name isn't Rob, it's Vince
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3f.sh
What is your name?: Rob
Hello Rob
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ 
```

test resolves to *false* (1)

test resolves to *true* (0)

[as test

- There is also a shortcut for the test command
- The **[** command can be used with the same effect
- Command-line list expressions using [ ]:



```
#!/bin/bash

read -p 'What is your name?: ' name
[ $name = 'Rob' ] && echo "Hello Rob" && exit 0
echo "Your name isn't Rob, it's $name"; exit 1
```

Even though **[** is a command, a closing square bracket is still required

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3g.sh
What is your name?: Vince
Your name isn't Rob, it's Vince
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3g.sh
What is your name?: Rob
Hello Rob
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$
```

test resolves to *false* (1)

test resolves to *true* (0)

Shell Script

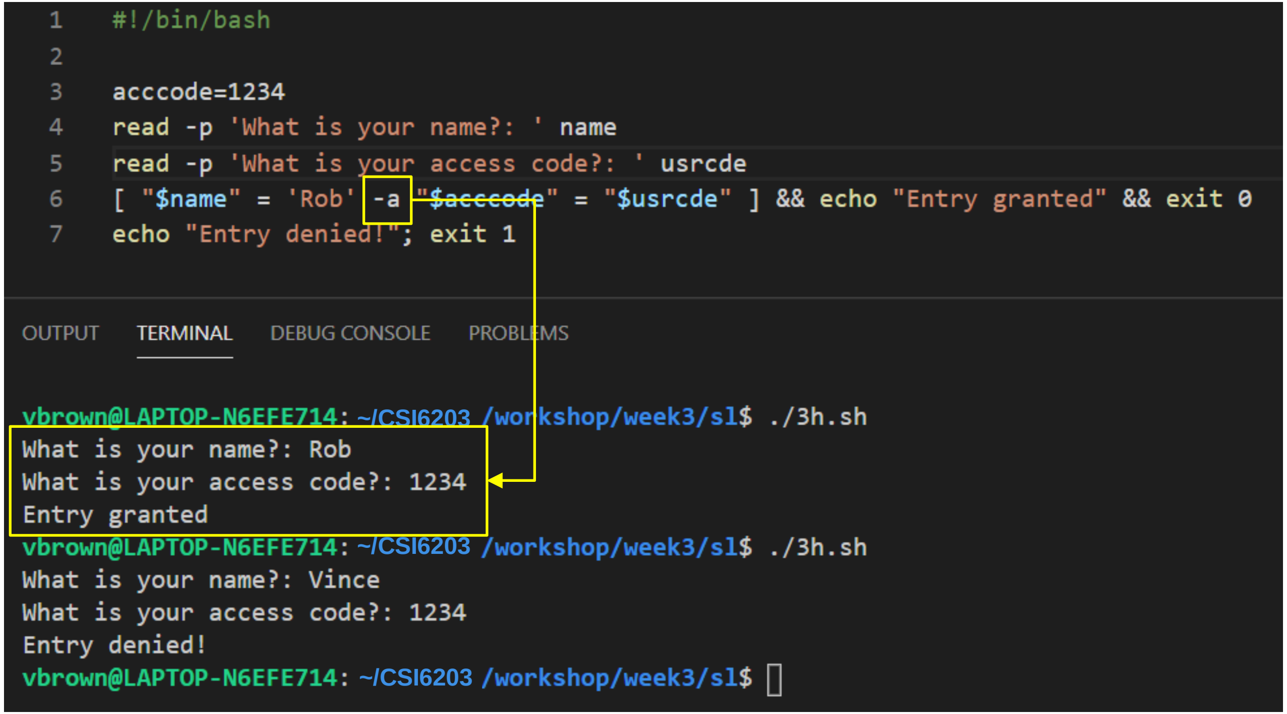
Output in Terminal

test booleans

- AND, OR and NOT boolean expressions can be used within the **test** statement to allow for more complex decision making logic to be implemented within a script
  - **-a** represents AND
  - **-o** represents OR
  - **!** represents NOT

test Boolean –a (AND)

- User must be Rob **AND** provide access code 1234



```bash
#!/bin/bash

acccode=1234
read -p 'What is your name?: ' name
read -p 'What is your access code?: ' usrcde
[ "$name" = 'Rob' -a "$acccode" = "$usrcde" ] && echo "Entry granted" && exit 0
echo "Entry denied!"; exit 1
```

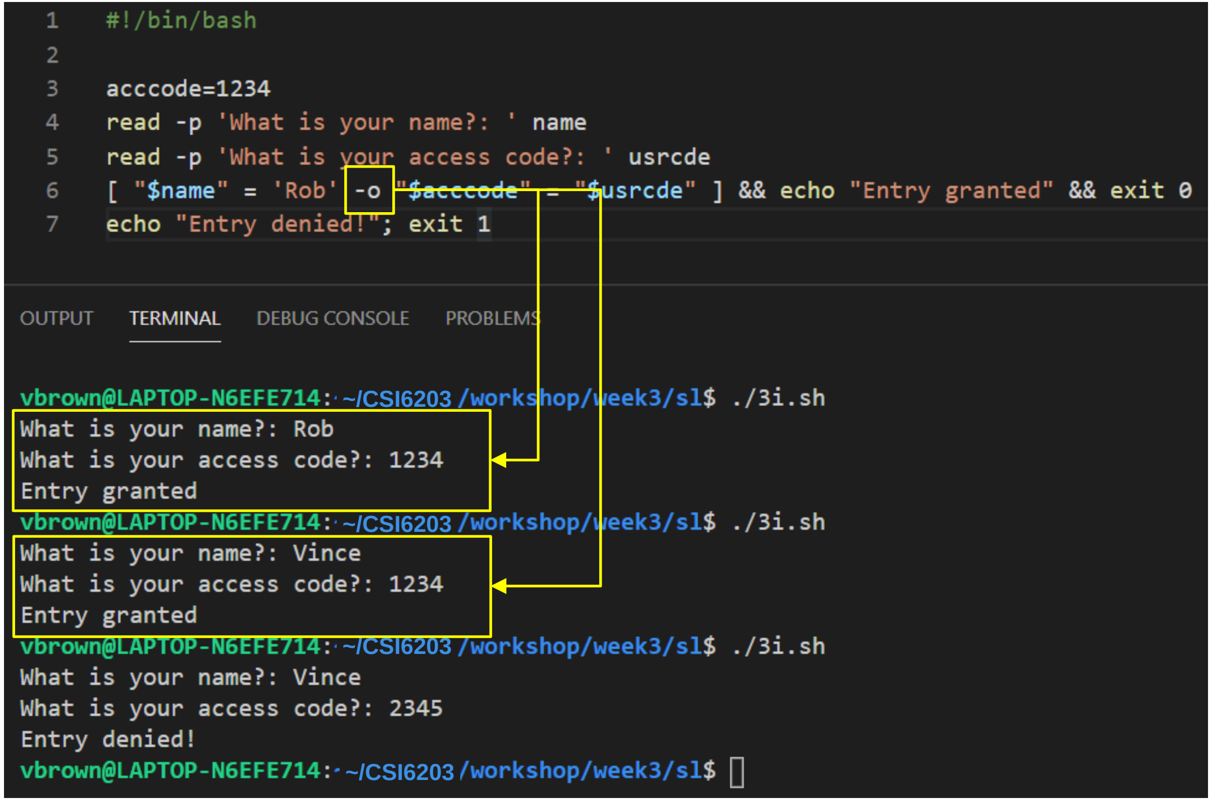
```
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3h.sh
What is your name?: Rob
What is your access code?: 1234
Entry granted
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$ ./3h.sh
What is your name?: Vince
What is your access code?: 1234
Entry denied!
vbrown@LAPTOP-N6EFE714: ~/CSI6203 /workshop/week3/sl$
```

Shell Script

Output in Terminal

test Boolean –o (OR)

- User can be Rob **OR** provide access code 1234 to enter



```bash
#!/bin/bash

acccode=1234
read -p 'What is your name?: ' name
read -p 'What is your access code?: ' usrcde
[ "$name" = 'Rob' -o "$acccode" = "$usrcde" ] && echo "Entry granted" && exit 0
echo "Entry denied!"; exit 1
```

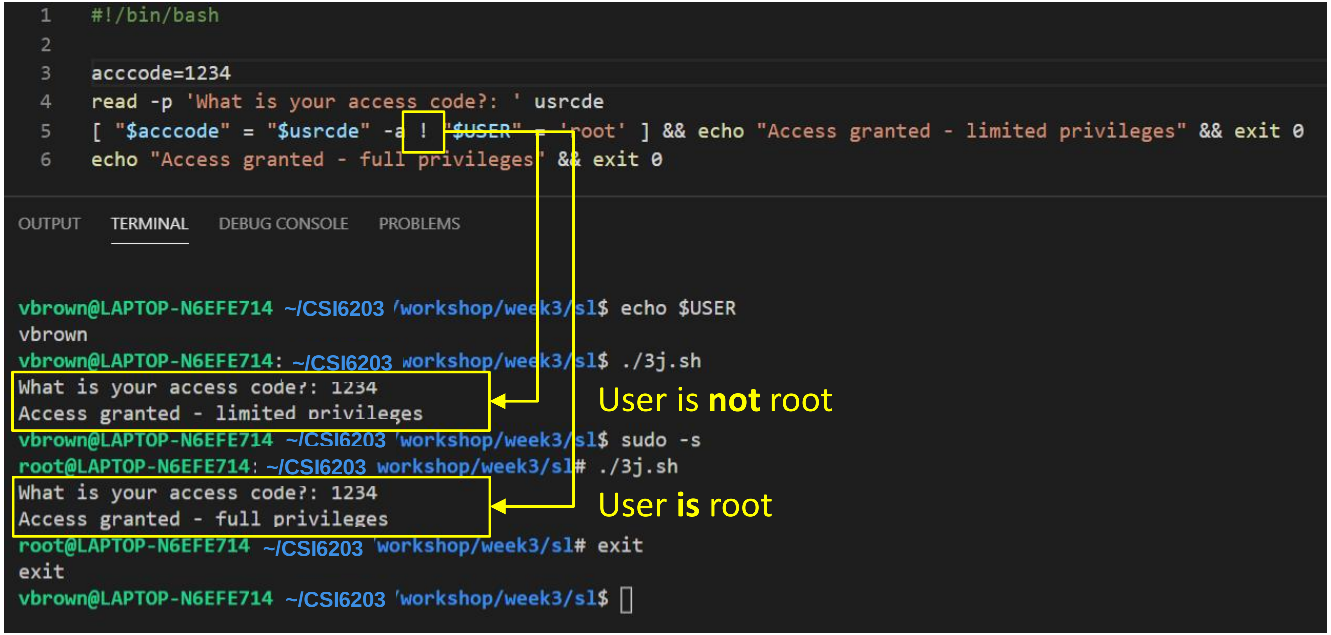
```
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshop/week3/sl$ ./3i.sh
What is your name?: Rob
What is your access code?: 1234
Entry granted
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshop/week3/sl$ ./3i.sh
What is your name?: Vince
What is your access code?: 1234
Entry granted
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshop/week3/sl$ ./3i.sh
What is your name?: Vince
What is your access code?: 2345
Entry denied!
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshop/week3/sl$ 
```

Shell Script

Output in Terminal

test Boolean ! (NOT)

- If user provides access code 1234 and is root user – full privileges
- If user provides access code 1234 and is **not** root user – limited privileges



Shell Script

```
#!/bin/bash

acccode=1234
read -p 'What is your access code?: ' usrcde
[ "$acccode" = "$usrcde" -a ! "$USER" = 'root' ] && echo "Access granted - limited privileges" && exit 0
echo "Access granted - full privileges" && exit 0
```

Output in Terminal

```
vbrown@LAPTOP-N6EFE714 ~/CSI6203 'workshop/week3/sl$ echo $USER
vbrown
vbrown@LAPTOP-N6EFE714: ~/CSI6203 workshop/week3/sl$ ./3j.sh
What is your access code?: 1234
Access granted - limited privileges
vbrown@LAPTOP-N6EFE714 ~/CSI6203 'workshop/week3/sl$ sudo -s
root@LAPTOP-N6EFE714: ~/CSI6203 workshop/week3/sl# ./3j.sh
What is your access code?: 1234
Access granted - full privileges
root@LAPTOP-N6EFE714 ~/CSI6203 'workshop/week3/sl# exit
exit
vbrown@LAPTOP-N6EFE714 ~/CSI6203 'workshop/week3/sl$ 
```

User is **not** root

User **is** root

Other applications of **test**

- **test** can be used to test numeric values…

Condition	Meaning
-eq	Is equal to
-gt	Is greater than
-lt	Is less than
-ge	Is greater than or equal to
-le	Is less than or equal to

- Test can be used to check status of files and directories as well

Condition	Meaning
-e	File or directory exists
-d	Is a directory
-f	Is a normal file (not a directory or device file)
-s	file is not zero size
-b	file is a block device (e.g. /dev/sda1)
-r	Is readable
-w	Is writeable
-x	Is executable
-nt	Is newer than
-ot	Is older than

If statements

- Although command-line lists are useful for short, simple logic, If statements are often more readable and a better choice when writing shell scripts
- The code within the **if** block will only execute if the condition evaluates to true
- The code within the else block will only execute if the condition evaluates to false

BASIC IF STATEMENT STRUCTURE

```
if [ $x -eq $y ]; then
    # code to execute if true
else
    # code to execute if false
fi
```

Elif

- The **if** statements can be further extended by providing multiple branching paths with **elif**:

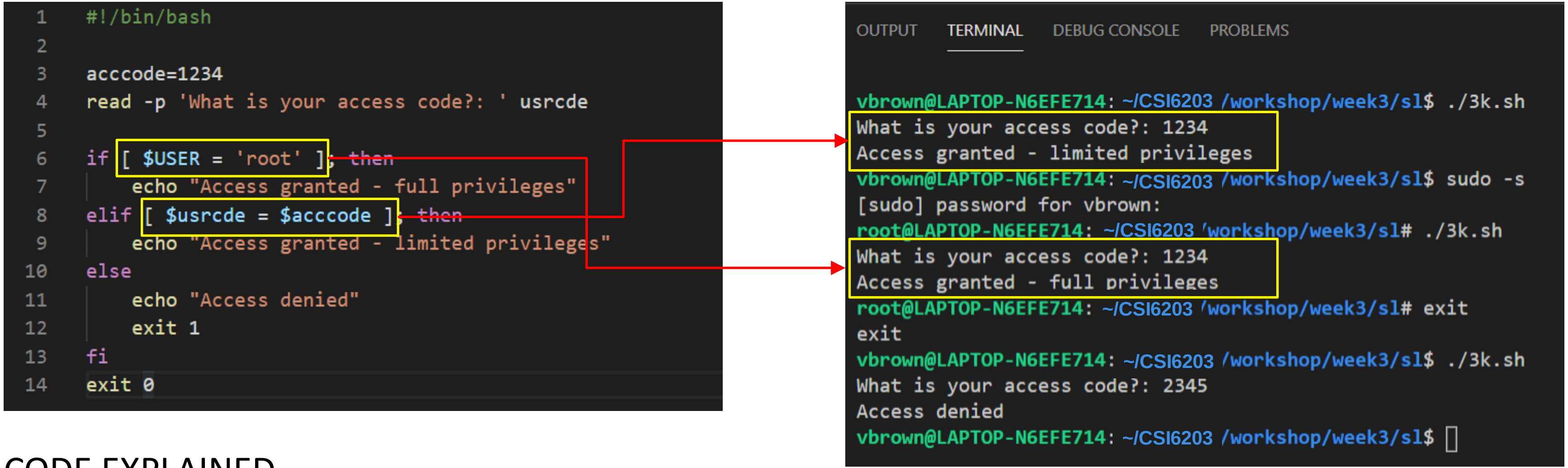
IF STATEMENT WITH ELIF FOR FURTHER BRANCHING

```
if [ $x -eq 1 ]; then
    # code to execute if true
elif [ $x -eq 2 ]; then
    # code to execute if true
else
    # code to execute if false
fi
```

if elif else – full example

Shell Script

Output in Terminal



```bash
#!/bin/bash

acccode=1234
read -p 'What is your access code?: ' usrcde

if [ $USER = 'root' ]; then
    echo "Access granted - full privileges"
elif [ $usrcde = $acccode ]; then
    echo "Access granted - limited privileges"
else
    echo "Access denied"
    exit 1
fi
exit 0
```

```
vbrown@LAPTOP-N6EFE714:~/CSI6203 /workshop/week3/sl$ ./3k.sh
What is your access code?: 1234
Access granted - limited privileges
vbrown@LAPTOP-N6EFE714:~/CSI6203 /workshop/week3/sl$ sudo -s
[sudo] password for vbrown:
root@LAPTOP-N6EFE714:~/CSI6203 /workshop/week3/sl# ./3k.sh
What is your access code?: 1234
Access granted - full privileges
root@LAPTOP-N6EFE714:~/CSI6203 /workshop/week3/sl# exit
exit
vbrown@LAPTOP-N6EFE714:~/CSI6203 /workshop/week3/sl$ ./3k.sh
What is your access code?: 2345
Access denied
vbrown@LAPTOP-N6EFE714:~/CSI6203 /workshop/week3/sl$
```

CODE EXPLAINED

- If the user is a root user, grant full privileges and exit the `if` statement
- If the user is *not* a root user but provides a *valid* access code, then grant them limited privileges and exit the `if` statement
- If the user is *not* a root user and has *not* provided a valid access code, grant them **no** privileges and exit the `if` statement

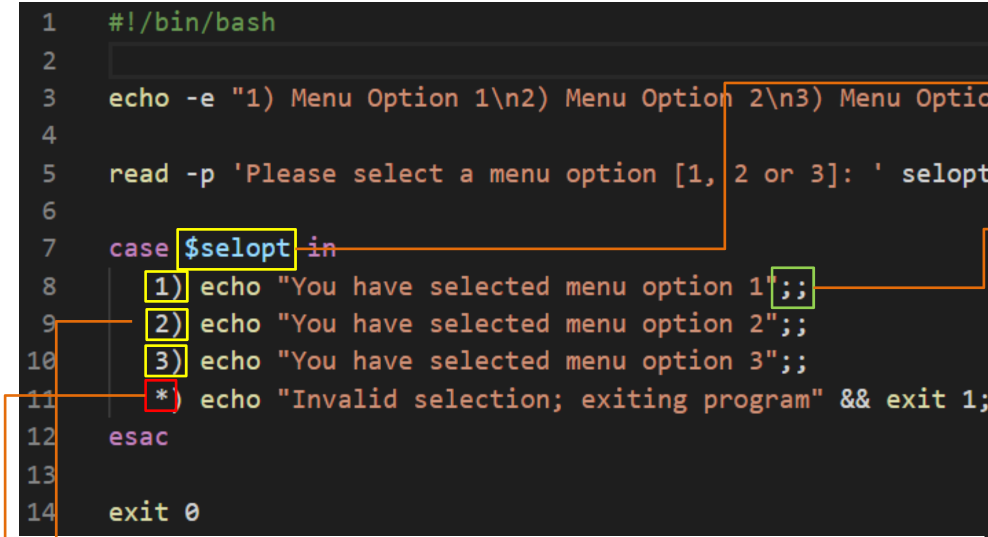
The case statement

- The case statement provides an alternative logical control structure when multilevel if-then-else-fi statements become too long and complex, in which case it easier to construct and
- The case statement enables the matching of several acceptable values against a singe variable, usually contained within a variable
- A general rule-of-thumb in programming is that if an if-else structure exceeds four (4) levels, use a case statement instead

BASIC CASE STATEMENT STRUCTURE

```
case test_value in
  val-opt_1) # code if true
       ;;
  val-opt_2) # code if true
       ;;
  val-opt_3) # code if true
       ;;
 *) # code if none above apply
       ;;
esac
```

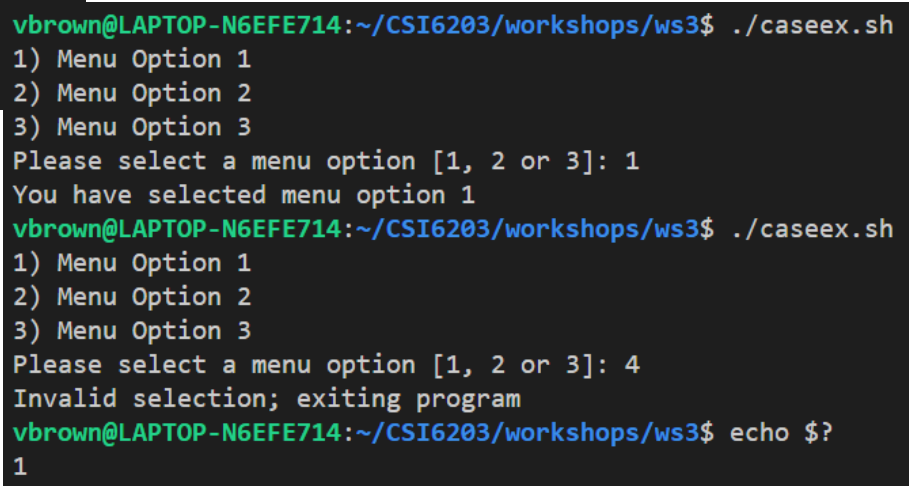
The case statement



```bash
#!/bin/bash

echo -e "1) Menu Option 1\n2) Menu Option 2\n3) Menu Option 3"

read -p 'Please select a menu option [1, 2 or 3]: ' selopt

case $selopt in
    1) echo "You have selected menu option 1";;
    2) echo "You have selected menu option 2";;
    3) echo "You have selected menu option 3";;
    *) echo "Invalid selection; exiting program" && exit 1;;
esac

exit 0
```

Variable that contains value to be tested, e.g. string, integer, float etc

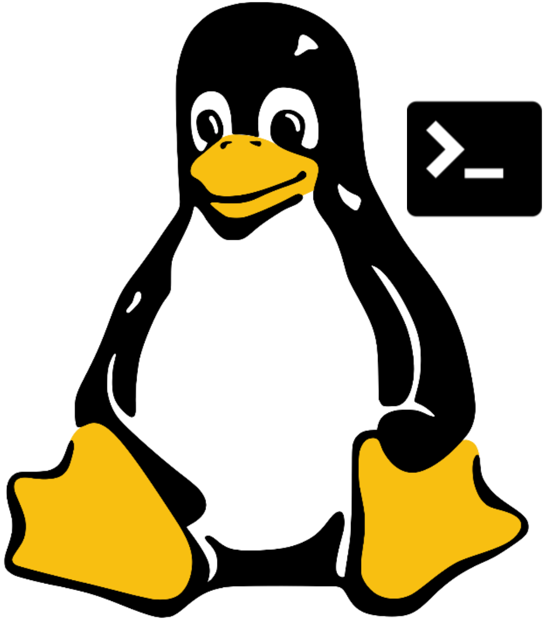
Each option in the case statement **must** end in *double semi-colon*



```
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshops/ws3$ ./caseex.sh
1) Menu Option 1
2) Menu Option 2
3) Menu Option 3
Please select a menu option [1, 2 or 3]: 1
You have selected menu option 1
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshops/ws3$ ./caseex.sh
1) Menu Option 1
2) Menu Option 2
3) Menu Option 3
Please select a menu option [1, 2 or 3]: 4
Invalid selection; exiting program
vbrown@LAPTOP-N6EFE714:~/CSI6203/workshops/ws3$ echo $?
1
```

Each allowable value must be immediately followed by a close parenthesis **)**

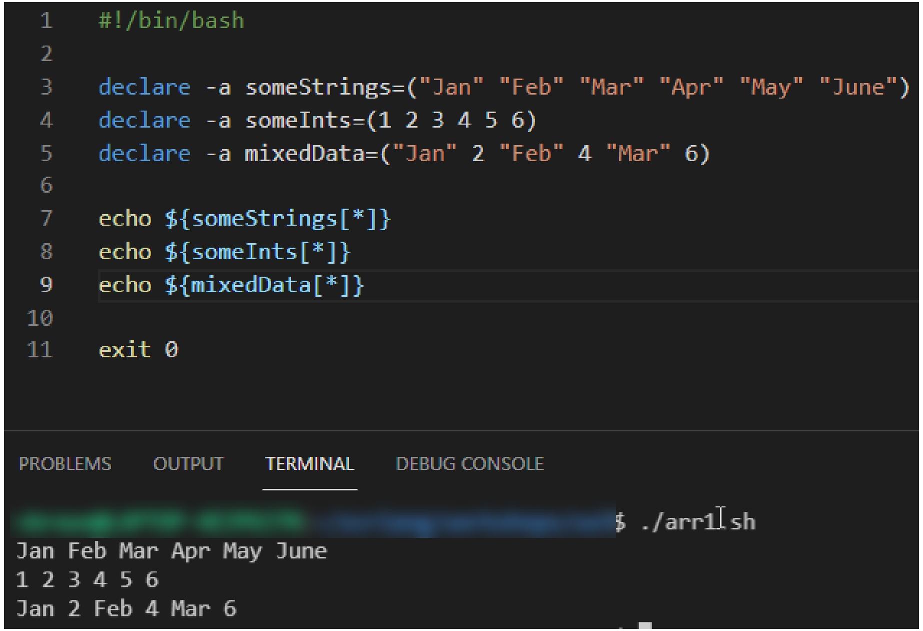
***** represents the default option to be executed if none of the allowable values in the list are matched; the ***** is optional



Working with Arrays

bash Arrays

- An array is a collection of data elements stored in contiguous memory locations
- Depending on the programming language being used, arrays can hold either *heterogeneous* or *homogenous* sets of data
- Arrays are an essential programmatic structure due to their ability to store sets of related data, which is something standard variables cannot generally do
- So whenever managing sets of related data, consider storing them in arrays

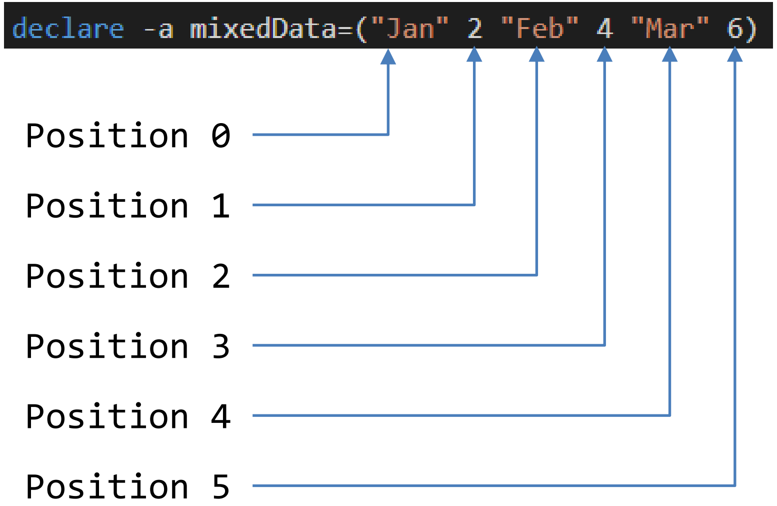


```
#!/bin/bash

declare -a someStrings=("Jan" "Feb" "Mar" "Apr" "May" "June")
declare -a someInts=(1 2 3 4 5 6)
declare -a mixedData=("Jan" 2 "Feb" 4 "Mar" 6)

echo ${someStrings[*]}
echo ${someInts[*]}
echo ${mixedData[*]}

exit 0
```

```
[redacted]$ ./arr1.sh
Jan Feb Mar Apr May June
1 2 3 4 5 6
Jan 2 Feb 4 Mar 6
```

bash Arrays

- Bash arrays come in two different types, these being *Indexed Arrays* and *Associative Arrays*
- An **Indexed Array** uses ordered integers as access keys (indexes) to identify array members positionally starting from 0. An Indexed Array is much like an ordered list
- An **Associative Array**, which is also sometimes referred to as a *hash table* uses keys (indexes) represented by arbitrary strings
- In this unit, we are mainly concerned with the use of *Indexed Arrays*



```
declare -a mixedData=("Jan" 2 "Feb" 4 "Mar" 6)
```

Position 0

Position 1

Position 2

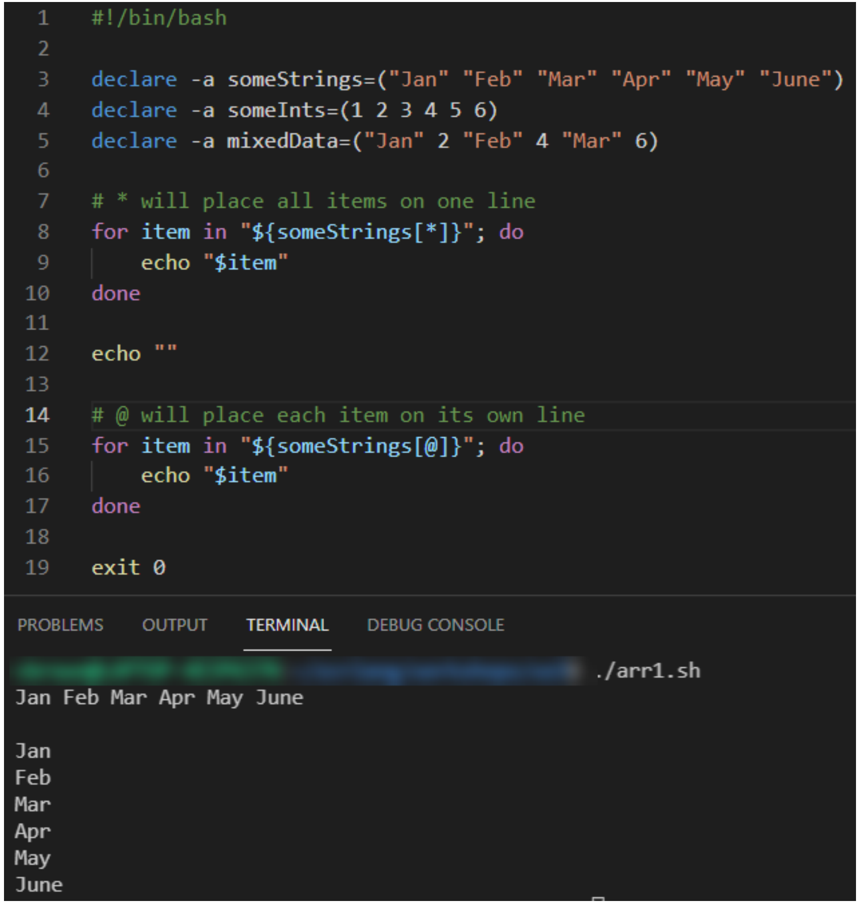
Position 3

Position 4

Position 5

bash Arrays

- There are multiple ways to access and output array data
- ***** will output all array members on a single line in the output destination, e.g. terminal, file etc
- **@** will output each array member on its own line in the output destination, e.g. terminal, file etc

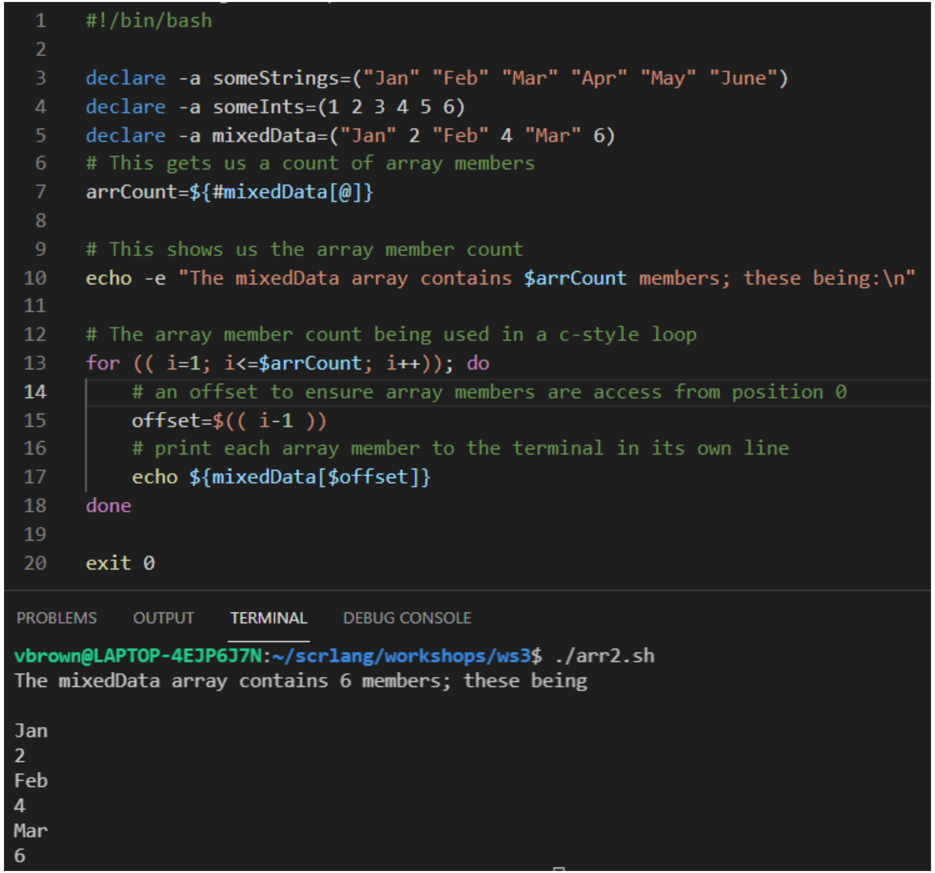


```bash
#!/bin/bash

declare -a someStrings=("Jan" "Feb" "Mar" "Apr" "May" "June")
declare -a someInts=(1 2 3 4 5 6)
declare -a mixedData=("Jan" 2 "Feb" 4 "Mar" 6)

# * will place all items on one line
for item in "${someStrings[*]}"; do
    echo "$item"
done

echo ""

# @ will place each item on its own line
for item in "${someStrings[@]}"; do
    echo "$item"
done

exit 0
```

```
[redacted] ./arr1.sh
Jan Feb Mar Apr May June

Jan
Feb
Mar
Apr
May
June
```

bash Arrays

- Individual array members can be accessed by their index number, starting from 0 for the first member
- # before the array name will get us a count of all of its members (Line 7)
- A loop counter can then be based in this count when stored in a variable or produced by command substitution



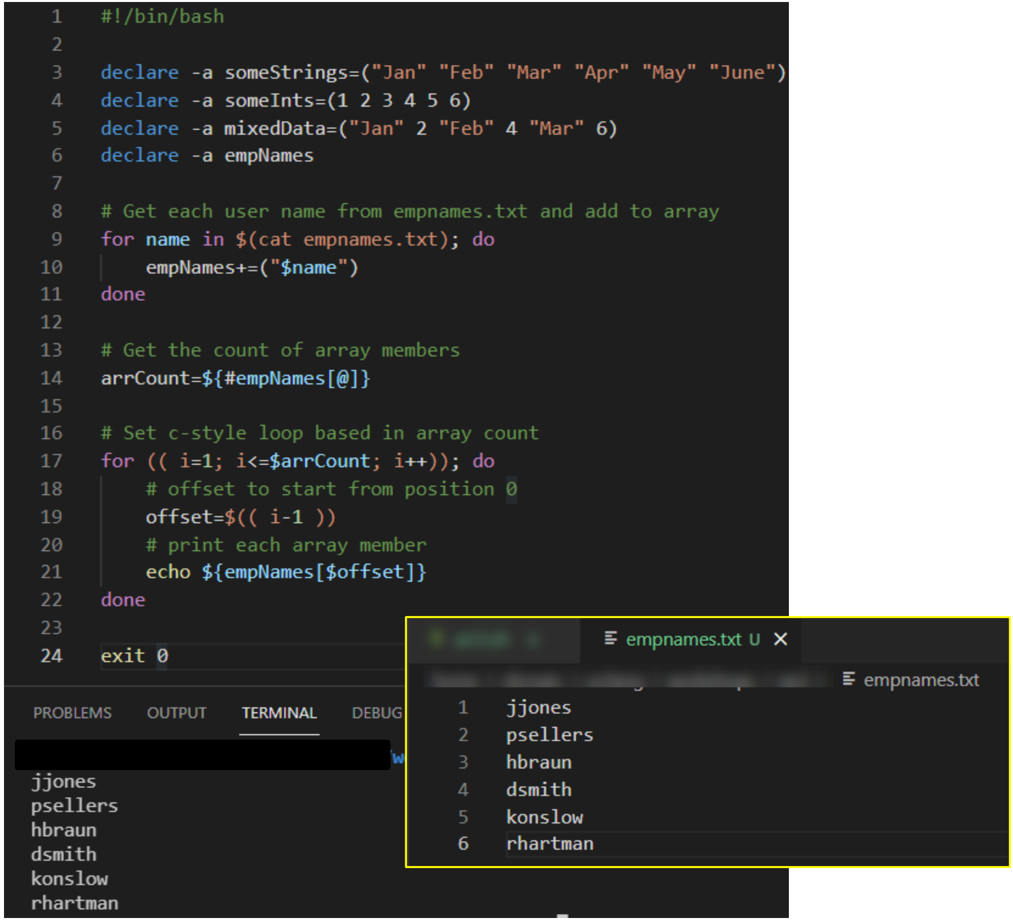
```
 1  #!/bin/bash
 2
 3  declare -a someStrings=("Jan" "Feb" "Mar" "Apr" "May" "June")
 4  declare -a someInts=(1 2 3 4 5 6)
 5  declare -a mixedData=("Jan" 2 "Feb" 4 "Mar" 6)
 6  # This gets us a count of array members
 7  arrCount=${#mixedData[@]}
 8
 9  # This shows us the array member count
10  echo -e "The mixedData array contains $arrCount members; these being:\n"
11
12  # The array member count being used in a c-style loop
13  for (( i=1; i<=$arrCount; i++)); do
14      # an offset to ensure array members are access from position 0
15      offset=$(( i-1 ))
16      # print each array member to the terminal in its own line
17      echo ${mixedData[$offset]}
18  done
19
20  exit 0
```

```
vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws3$ ./arr2.sh
The mixedData array contains 6 members; these being

Jan
2
Feb
4
Mar
6
```

bash Arrays

- Members can be added to an array when declared and initialized **or** afterwards when members become available
- The latter can be done using the **+=** method
- You can see this in action on Line 10 inside a C-style loop structure

NOTE: We'll be examining loop structures in detail in Module 5

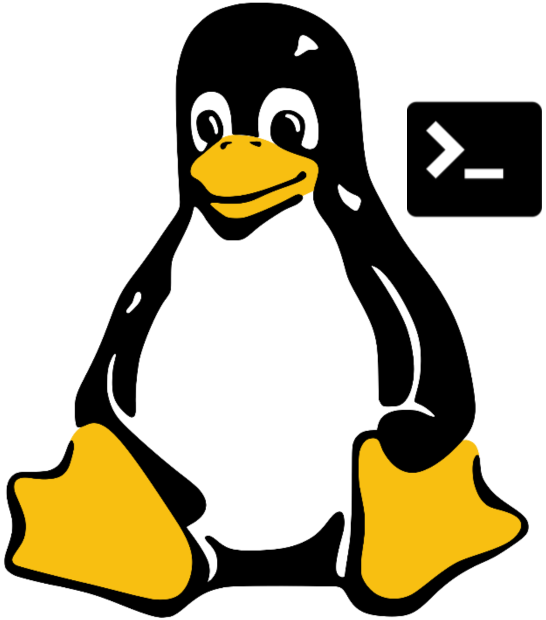


```
1   #!/bin/bash
2
3   declare -a someStrings=("Jan" "Feb" "Mar" "Apr" "May" "June")
4   declare -a someInts=(1 2 3 4 5 6)
5   declare -a mixedData=("Jan" 2 "Feb" 4 "Mar" 6)
6   declare -a empNames
7
8   # Get each user name from empnames.txt and add to array
9   for name in $(cat empnames.txt); do
10      empNames+=("$name")
11  done
12
13  # Get the count of array members
14  arrCount=${#empNames[@]}
15
16  # Set c-style loop based in array count
17  for (( i=1; i<=$arrCount; i++)); do
18      # offset to start from position 0
19      offset=$(( i-1 ))
20      # print each array member
21      echo ${empNames[$offset]}
22  done
23
24  exit 0
```

[redacted]

```
jjones
psellers
hbraun
dsmith
konslow
rhartman
```

```
jjones
psellers
hbraun
dsmith
konslow
rhartman
```



Basic bash Maths

Basic bash Maths

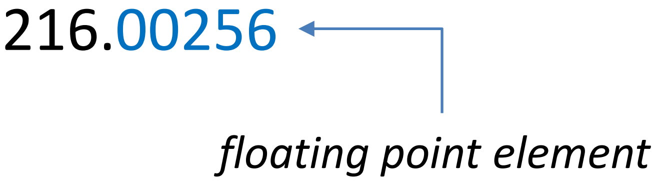
- Although shell script was not designed to handle complex mathematical equations, it does posses commands and features that allow you to perform integer and real number calculations. The *bash* shell supports a range of arithmetical operators with which to perform mathematical calculations. The most common of these are as follows:

Arithmetic Operator	Description
id++, id–	variable post-increment, post-decrement
++id, –id	variable pre-increment, pre-decrement
**	exponentiation
*, /, %	multiplication, division, remainder (modulo)
+, -	addition, subtraction
<=, >=, <, >	comparison
==, !=	equality, inequality
&&	logical AND
||	logical OR
=, *=, /=, %=, +=, -=, «=, »=, &=, ^=, |=	assignment

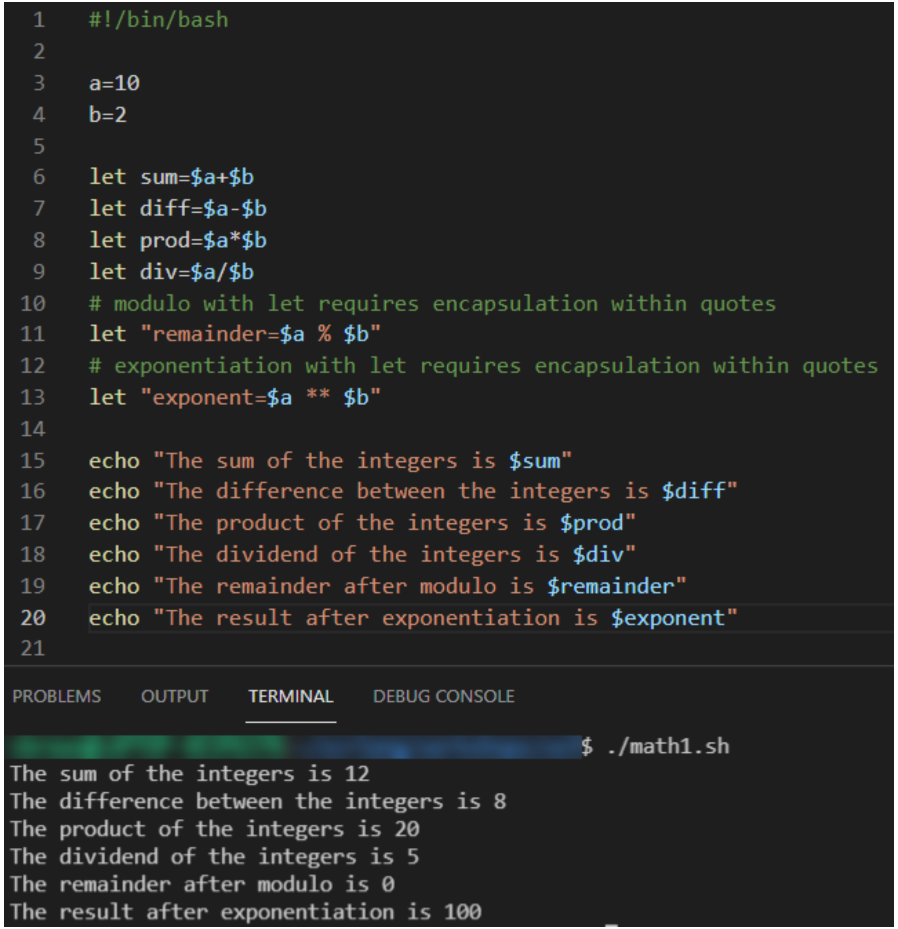
Basic bash Maths

The **let** keyword

- Arithmetical operators can be used in conjunction with the **let** keyword
- This approach is perhaps the simplest and most direct method by which to perform integer-based calculations, i.e., calculations that do **not** involve real numbers and thus must account for floating point elements


216.00256

floating point element



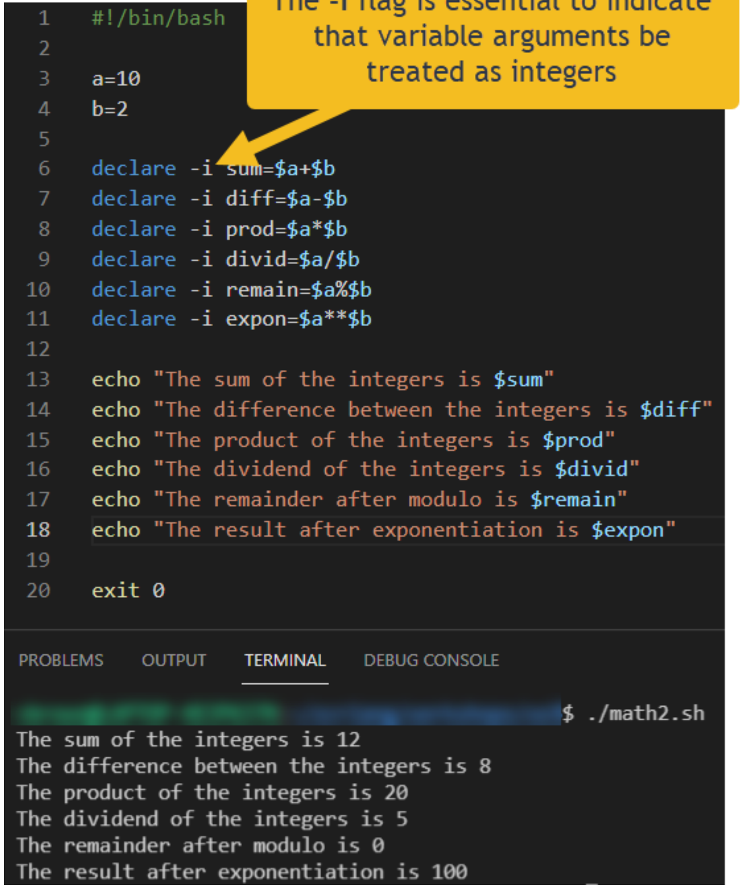
```bash
#!/bin/bash

a=10
b=2

let sum=$a+$b
let diff=$a-$b
let prod=$a*$b
let div=$a/$b
# modulo with let requires encapsulation within quotes
let "remainder=$a % $b"
# exponentiation with let requires encapsulation within quotes
let "exponent=$a ** $b"

echo "The sum of the integers is $sum"
echo "The difference between the integers is $diff"
echo "The product of the integers is $prod"
echo "The dividend of the integers is $div"
echo "The remainder after modulo is $remainder"
echo "The result after exponentiation is $exponent"

```

```
[redacted]$ ./math1.sh
The sum of the integers is 12
The difference between the integers is 8
The product of the integers is 20
The dividend of the integers is 5
The remainder after modulo is 0
The result after exponentiation is 100
```

Basic bash Maths

The **declare** keyword

- Bash does not employ strict data typing, so the **declare** keyword can be used to tell the system that a variable's resident argument is to be treated as a specific data type, e.g. *integer*, *array*, etc
- To this end, we can use **declare** to indicate that variable arguments being referred to within the context of arithmetical operation be treated as integers using the **-i** flag



The -i flag is essential to indicate that variable arguments be treated as integers

```
#!/bin/bash

a=10
b=2

declare -i sum=$a+$b
declare -i diff=$a-$b
declare -i prod=$a*$b
declare -i divid=$a/$b
declare -i remain=$a%$b
declare -i expon=$a**$b

echo "The sum of the integers is $sum"
echo "The difference between the integers is $diff"
echo "The product of the integers is $prod"
echo "The dividend of the integers is $divid"
echo "The remainder after modulo is $remain"
echo "The result after exponentiation is $expon"

exit 0
```

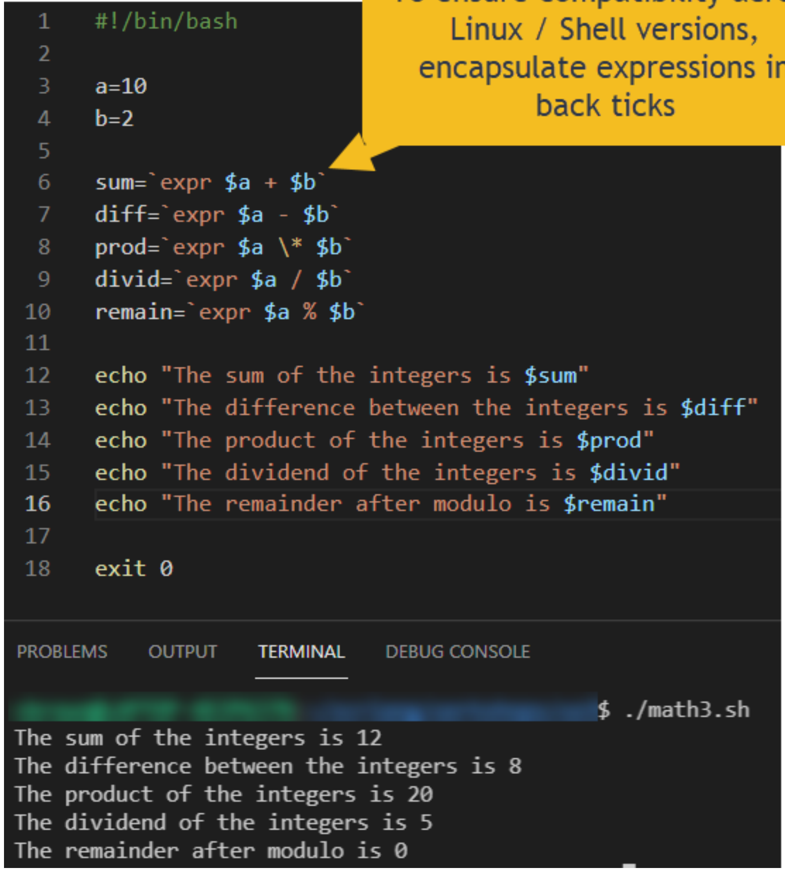
```
[redacted]$ ./math2.sh
The sum of the integers is 12
The difference between the integers is 8
The product of the integers is 20
The dividend of the integers is 5
The remainder after modulo is 0
The result after exponentiation is 100
```

Basic bash Maths

The **expr** command

- The **expr** command is extremely useful in a wide range of contexts and is also very handy for performing arithmetical operations on integers
- The **expr** command is also great for evaluating regular expressions and performing string operations, e.g. substring, string length etc
- Note – the **expr** command does not support exponentiation

To ensure compatibility across Linux / Shell versions, encapsulate expressions in back ticks



```bash
#!/bin/bash

a=10
b=2

sum=`expr $a + $b`
diff=`expr $a - $b`
prod=`expr $a \* $b`
divid=`expr $a / $b`
remain=`expr $a % $b`

echo "The sum of the integers is $sum"
echo "The difference between the integers is $diff"
echo "The product of the integers is $prod"
echo "The dividend of the integers is $divid"
echo "The remainder after modulo is $remain"

exit 0
```

```
[redacted]$ ./math3.sh
The sum of the integers is 12
The difference between the integers is 8
The product of the integers is 20
The dividend of the integers is 5
The remainder after modulo is 0
```

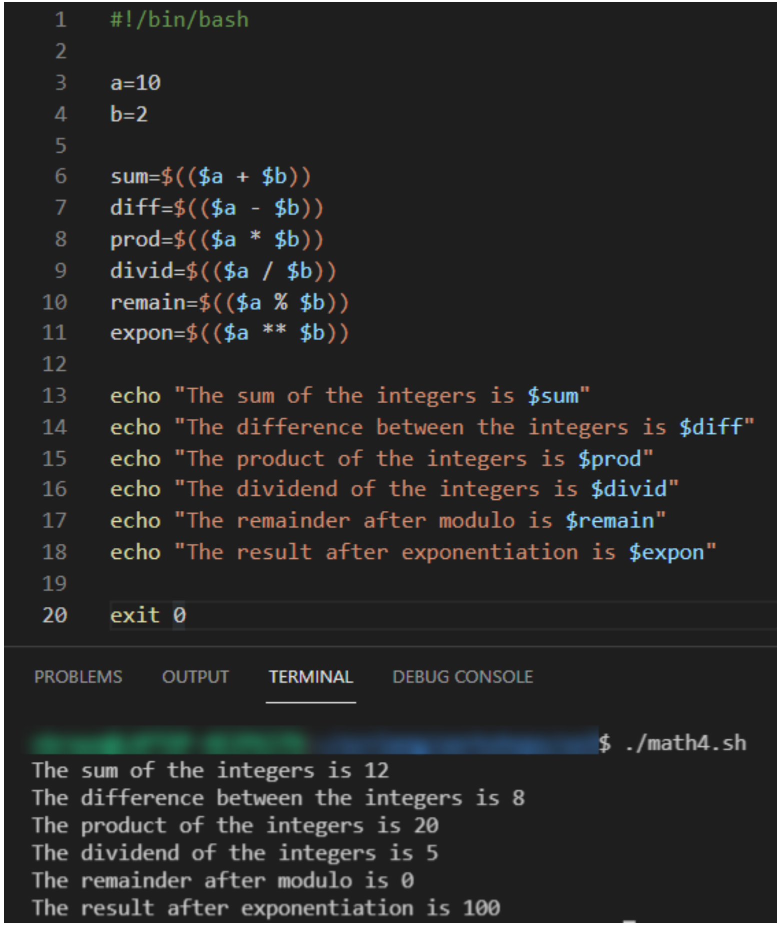
Basic bash Maths

Arithmetic Expansion

- **Arithmetic expansion** allows the in-situ evaluation of an arithmetic expression and subsequent substitution of the result where required, e.g. assigned to a variable
- It provides yet another convenient means by which to performing integer arithmetic operations in your scripts
- The format for arithmetic expansion is:

```
$(( expression ))
```

- *Note – when using arithmetic expansion, expressions can be nested and the normal order of precedence applies*



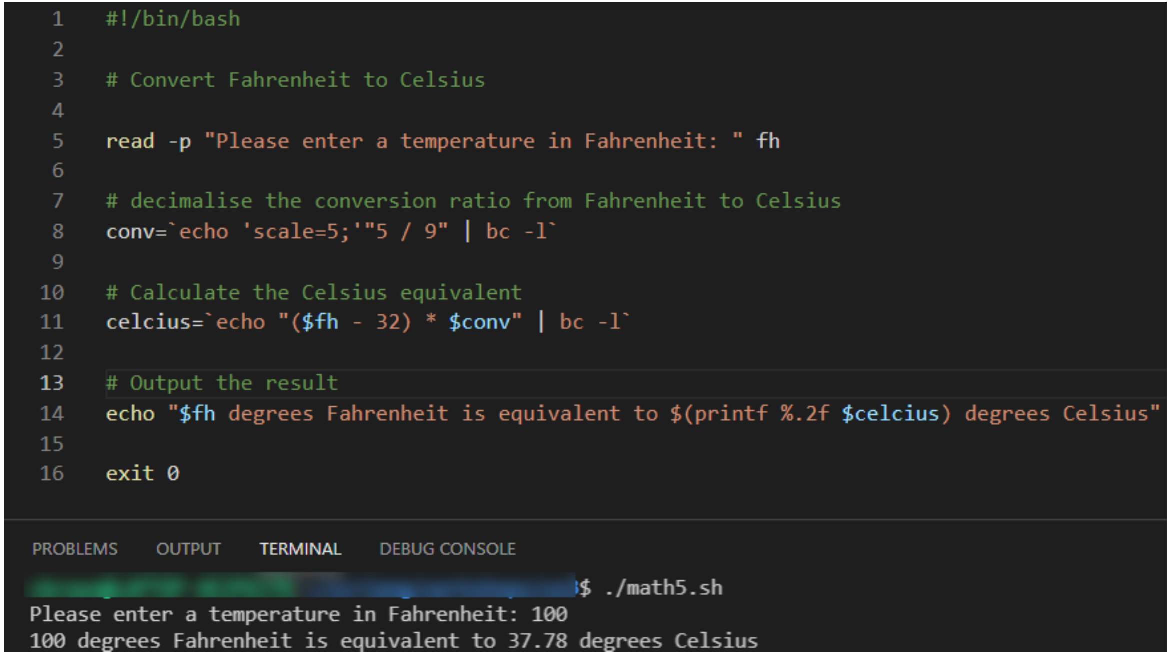
```
#!/bin/bash

a=10
b=2

sum=$(($a + $b))
diff=$(($a - $b))
prod=$(($a * $b))
divid=$(($a / $b))
remain=$(($a % $b))
expon=$(($a ** $b))

echo "The sum of the integers is $sum"
echo "The difference between the integers is $diff"
echo "The product of the integers is $prod"
echo "The dividend of the integers is $divid"
echo "The remainder after modulo is $remain"
echo "The result after exponentiation is $expon"

exit 0
```

```
[redacted]$ ./math4.sh
The sum of the integers is 12
The difference between the integers is 8
The product of the integers is 20
The dividend of the integers is 5
The remainder after modulo is 0
The result after exponentiation is 100
```

Basic bash Maths

Float Calculations Using the bash Calculator (bc)

- The **bc** command is very useful for floating point calculations
- **bc** stands for *__b__ash __c__alculator*
- Generally, you will pass the **bc** a *scale* and an *expression* to be calculated
- This will often be employed within a *command substitution* **$( )** structure so a result can be stored in variable, array etc



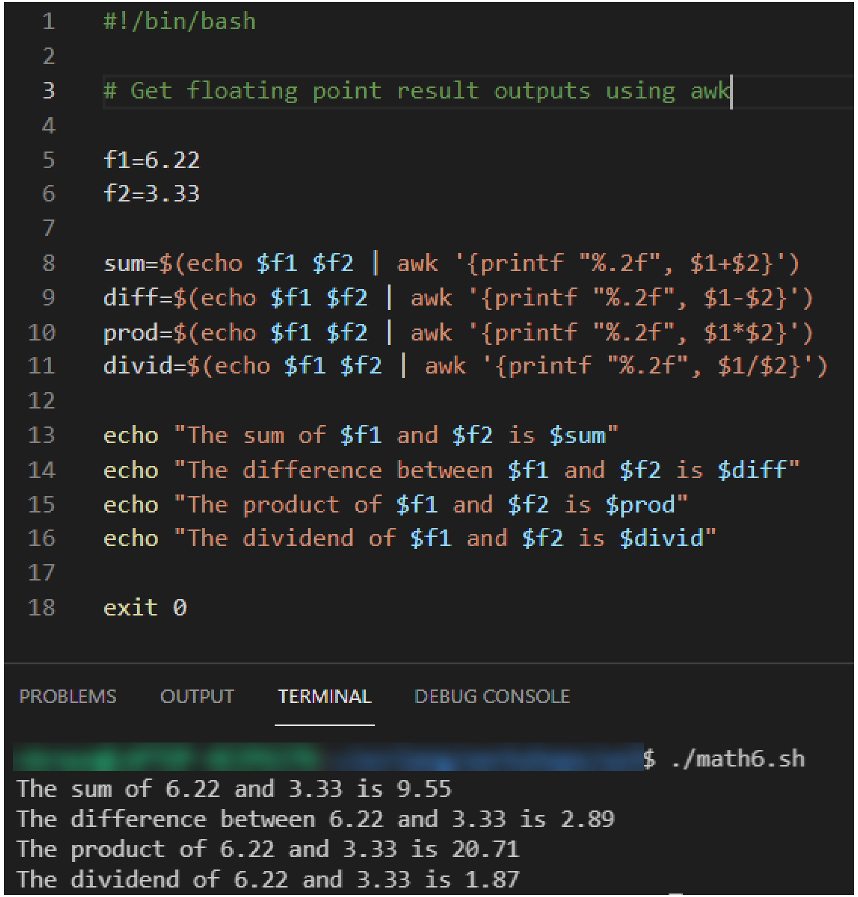
```
#!/bin/bash

# Convert Fahrenheit to Celsius

read -p "Please enter a temperature in Fahrenheit: " fh

# decimalise the conversion ratio from Fahrenheit to Celsius
conv=`echo 'scale=5;'"5 / 9" | bc -l`

# Calculate the Celsius equivalent
celcius=`echo "($fh - 32) * $conv" | bc -l`

# Output the result
echo "$fh degrees Fahrenheit is equivalent to $(printf %.2f $celcius) degrees Celsius"

exit 0
```

```
[redacted]$ ./math5.sh
Please enter a temperature in Fahrenheit: 100
100 degrees Fahrenheit is equivalent to 37.78 degrees Celsius
```

Basic bash Maths

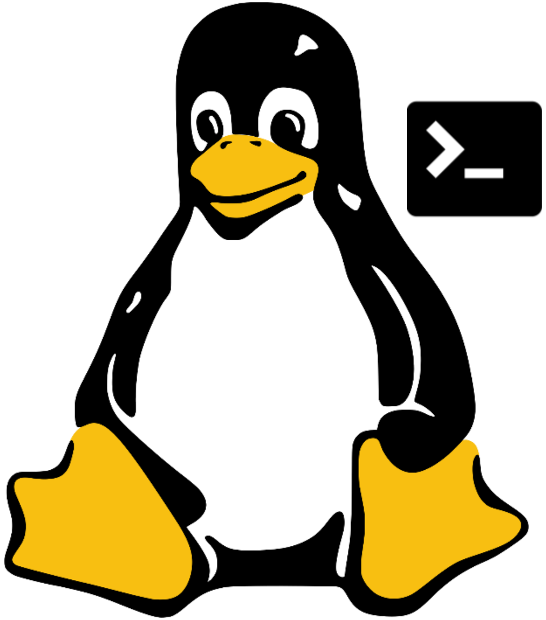
Float Calculations Using awk

- The **awk** utlity is also very useful for floating point calculations as it handles complex maths by design
- Once again, awk can be called from within a *command substitution* **$( )** structure so a result can be stored in variable, array etc
- We'll look at awk in more depth in *Module 8*



```
#!/bin/bash

# Get floating point result outputs using awk

f1=6.22
f2=3.33

sum=$(echo $f1 $f2 | awk '{printf "%.2f", $1+$2}')
diff=$(echo $f1 $f2 | awk '{printf "%.2f", $1-$2}')
prod=$(echo $f1 $f2 | awk '{printf "%.2f", $1*$2}')
divid=$(echo $f1 $f2 | awk '{printf "%.2f", $1/$2}')

echo "The sum of $f1 and $f2 is $sum"
echo "The difference between $f1 and $f2 is $diff"
echo "The product of $f1 and $f2 is $prod"
echo "The dividend of $f1 and $f2 is $divid"

exit 0
```

```
[redacted]$ ./math6.sh
The sum of 6.22 and 3.33 is 9.55
The difference between 6.22 and 3.33 is 2.89
The product of 6.22 and 3.33 is 20.71
The dividend of 6.22 and 3.33 is 1.87
```

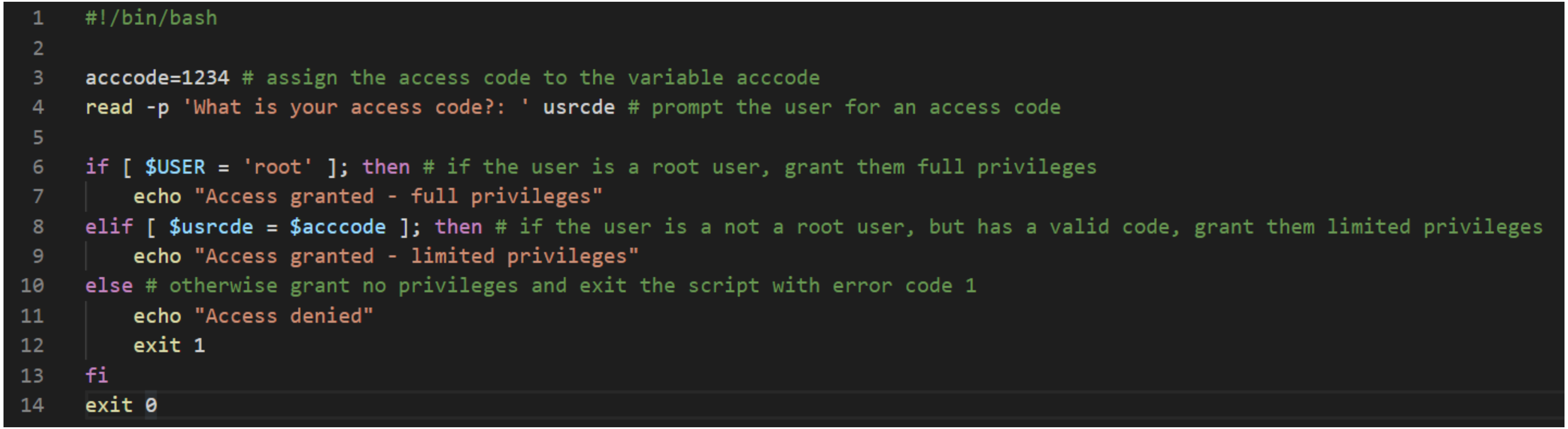


Code Commenting

Comments

- Its important to comment your scripts
- Comments describe the purpose of the code used to accomplish the purpose.
- The purpose of a script should be understandable without ever having to look at the code and by simply reading the comments.
- Commenting is best done before actually writing the code for your program.
- Comments are specially marked lines of text in the program that are not evaluated.
- Any statement in a script that starts with a ‘#’ symbol is ignored by the computer

Comments



```bash
#!/bin/bash

acccode=1234 # assign the access code to the variable acccode
read -p 'What is your access code?: ' usrcde # prompt the user for an access code

if [ $USER = 'root' ]; then # if the user is a root user, grant them full privileges
    echo "Access granted - full privileges"
elif [ $usrcde = $acccode ]; then # if the user is a not a root user, but has a valid code, grant them limited privileges
    echo "Access granted - limited privileges"
else # otherwise grant no privileges and exit the script with error code 1
    echo "Access denied"
    exit 1
fi
exit 0
```

IMPORTANT NOTE

Thorough commenting will form part of your assessments so be sure to get into the habit of writing them

References and Resources

- Ebrahim, M. and Mallet, A. (2018) *Mastering Linux Based Scripting (2nd Ed).* Chapter 2, pp 35-52

- Free Software Foundation, Inc. (2019) *GNUBash Reference Manual. Edition 5.0 for Bash Version 5.0*. Available at https://www.gnu.org/software/bash/manual/bash.pdf

- SS64.com. (2020) *An A-Z Index of the Linux command line: bash + utilities*. Available at https://ss64.com/bash/