BASH SHELL FUNCTIONS

...

TLO KNOWLEDGE AND SKILLS

Conditions:

- Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of Bash shell functions.

Knowledge:

- Identify the fundamentals of programming concepts.
- Understand what bash procedural programs are.

Skills:

- Working knowledge of the programming concepts of bash standard inputs and outputs.

OBJECTIVES

- Understand fundamental programming concepts.
- Create procedural programs.
- Identify Bash redirection with Standard Input, Standard Output and Standard Error

SHELL FUNCTIONS

Let’s review what we learned in lesson 1 about functions.

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

ourfunction () {
        # This is the body of our function
    echo "This is our function"
}

ourfunction
        # This is us calling the function in our script
----
```

SHELL FUNCTIONS

We know that variables are global by default and this will come into play when we define variables in scripts with functions.

The first way we can write and call variables in scripts is to declare them within our script and not within our functions.

This will allow the variable to be accessed anywhere including in function bodies.

SHELL FUNCTIONS VARIABLES

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

my_var="This is the value of my_var"

ourfunction () {
    echo $my_var "written from our function"
        # This is our function body
}

ourfunction
        # This is us calling our function that references our variable

echo $my_var "written outside our function"
        # This is us referencing our variable outside of the function
----
```

SHELL FUNCTIONS VARIABLES

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

ourfunction () {
    my_var="This is the value of my_var"
    echo $my_var "written from our function"
        # This is our function body
}

ourfunction
        # This is us calling our function that references our variable

echo $my_var "written outside our function"
        # This is us referencing our variable outside of the function
----
```

SHELL FUNCTIONS VARIABLES

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

my_var="This is the value of my_var"

ourfunction () {
    my_var="This is the new value of my_var"
    echo $my_var "written from our function"
        # This is our function body
}

echo $my_var "written before calling our function"

ourfunction
        # This is us calling our function that references our variable

echo $my_var "written after calling our function"
        # This is us referencing our variable outside of the function
---
```

SHELL FUNCTIONS VARIABLES

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

ourfunction () {
    local my_var="This is the value of my_var"
    echo $my_var "written from our function"
        # This is our function body
}

ourfunction
        # This is us calling our function that references our variable

echo $my_var "written outside our function"
        # This is us referencing our variable outside of the function
----
```

SHELL FUNCTIONS VARIABLES

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

my_var="This is the value of my_var"

ourfunction () {
    local my_var="This is the new value of my_var"
    echo $my_var "written from our function"
        # This is our function body
}

echo $my_var "written before calling our function"

ourfunction
        # This is us calling our function that references our variable

echo $my_var "written after calling our function"
        # This is us referencing our variable outside of the function
---
```

Function Script Demo

CHECK ON LEARNING

1. What is the purpose of a function?

    To reduce the amount of repetitive code within a script.

2. How do we call a function?

    By using the name of the function

3. True or False. Functions are always accessible outside of the script they're written in.

    False

NESTED FUNCTIONS

Nested functions are functions that contain another function within it.

This can be accomplished by calling an already defined function within a function or by writing a function within your function.

It's important to note that you can only call functions that have already been defined so if you define a function within a function it cannot be called outside of that function.

NESTED FUNCTIONS

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

functionOne() {
    echo "One"
    sleep 1
}

functionTwo() {
    echo "Two"
    sleep 1
}

functionThree() {
    functionOne
    functionTwo
    echo "Three"
    sleep 1
    echo "Four"
    xdg-open https://www.youtube.com/watch?v=o-YBDTqX_ZU
}

functionOne

functionTwo

functionThree
----
```

Nested Function Script Demo

Positional Parameters/Arguments

Parameters are set commands ranging from 0-9 that can be established within a script.

Parameters are referenced in functions just like a variable by using the $.

Parameters allow for specific input without having to change the variable or location of the variable every time.

Positional Parameters/Arguments

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

echo "Name: $1"
echo "Rank: $2"
echo "class: $3"
echo "Topic: $4"
---
```

Save your script and run the following command to pass through parameters to your script.

./file.sh "Snuffy" "PVT" "22-000" "scripting"

Parameter Script Demo

CHECK ON LEARNING

1. How do you write parameters bigger than 9?

   You must include {} between the $ and the number (e.x ${10})

2. What are parameters?

   Arguments passed into your script or function.

SHELL EXPANSIONS

Shell expansion is a broad topic that can mean many different types of expansion. We'll cover:

- Brace expansion
- Tilde Expansion
- Command Substitution

We've already seen parameter expansion and variable expansion

BRACE EXPANSIONS

Brace expansion is a method by which arbitrary strings may be generated.

Brace expansion can be nested.

```
#!/bin/bash

mkdir directory{1..100}
echo s{mal,tal,mel}l
```

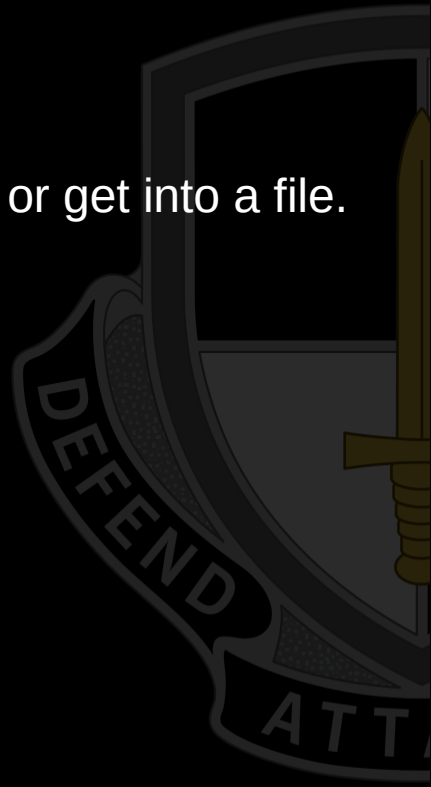
TILDE EXPANSIONS

The tilde (~) character references the home directory.

Tilde Expansion is a way to create a shortcut to change directories or get into a file.

#!/bin/bash

cat ~/Desktop/file.sh



COMMAND SUBSTITUTION

Command substitution allows the output of a command to replace the command itself.

Bash performs the expansion by executing COMMAND and replacing the command substitution with the standard output of the command

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

for i in $(ls)
do
    if [ -d $i ]
    then
        touch ~/"ls $i"
        ls $i > "ls $i"
    fi
done
----
```

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

while :
do
    mkdir "$(date)"
    for i in $(ls)
    do
        if [ -d $i ]
        then
            touch ~/"$(date)"/"ls $i"
            ls $i > ~/"$(date)"/"ls $i"
        fi
    done
    sleep 60
done
----
```

CHECK ON LEARNING

1. What allows you to use the output of a command within a command?

   command substitution $()

2. Write a simple brace expansion to echo the alphabet backwards.

   echo {z..a}

3. How would I write each letter on a new line?

   for i in {z..a}; do; echo $i; done

BASH REDIRECTION

Redirection allows commands' file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to.

- Standard input (stdin): File descriptor 0
- Standard output (stdout): File descriptor 1
- Standard error (sterr): File descriptor 2

REDIRECTING OUTPUT

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

touch file.sh
date > file.sh
nano file.sh
---
```

REDIRECTING INPUT

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

sort < file.txt
---
```

REDIRECTING APPEND

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

file1.txt >> file2.txt
echo file2.txt
---
```

REDIRECTING

You can use redirection for a number of things, one of the most common being redirecting error to */dev/null*

---

find / file.txt 2>/dev/null

CHECK ON LEARNING

1. What is the difference between > and >>

> just redirects standard output while >> appends and redirects standard output

2. What is the file descriptor for error?

2