[Udemy] Bash Shell Scripting Tutorial for Beginners

Seção: 2 Bash Shell Scripting Tutorial

42. Bash Introduction [42_bash-introduction.sh]
    - Shell, Shell Script e Bash
    - Verificar os tipos de shell suportado pelo SO
    - Verificar localização do bash
    - Criar hello world bash script
    - Executar um script
43. using Variables and Comments [43_using-variables-and-comments.sh]
    - Comentários
    - Variáveis
45. Read User Input [45_read-user-input.sh]
    - Múltiplos inputs
    - Especificar o prompt na mesma linha do input
    - Tornar o input oculto/silencioso
    - Ler os inputs em um array
    - Comando read sem variável
47. Pass Arguments to a Bash-Script [47_pass-arguments-to-a-bash-script.sh]
    - Passando Argumentos em um Array
    - Imprimir o número de argumentos
49. If Statement (If then , If then else, If elif else) [49_if-statement.sh, 49_user.sh, 49_number.sh]
    - Expressions
51. File test operators [51_file-test-operators.sh]
53. How to append output to the end of text file [53_append-output-text-file.sh, 53_test]
55. Logical 'AND' Operator [55_logical-and-operator.sh]
57. Logical 'OR' Operator [57_logical-or-operator.sh]
59. Perform arithmetic operations in integers [59_arithmetic-operations.sh]
    - Alternativa
61. Floating point math operations in bash | bc Command [61_floating-point-math-operations.sh]
63. The case statement [63_the-case-statement.sh]
65. The case statement Example [65_the-case-statement-example.sh]
67. Array variables [67_array-variables.sh]
    - Variável como array
69. WHILE Loops [69_while-loops.sh]
71. using sleep and open terminal with WHILE Loops [71_while-loops.sh]
    - Usar sleep com loops
    - Abrir um terminal, usando o bash shell script
73. Read a file content in Bash [73_read-file-content.sh]
    - Input Redirection (redirecionamento de entrada)
    - 2º Método
    - 3º Método
74. UNTIL loop [74_until-loop.sh]
76. FOR loop [76_for-loop.sh]
    - 1ª Sintaxe

- 2ª Sintaxe
- 3ª Sintaxe

78. use FOR loop to execute commands [78_for-loop-execute-commands.sh]
80. Select loop [80_select-loop.sh]
81. Break and continue [81_break-continue.sh]
    - Declaração Break
    - Declaração Continue
82. Functions [82_functions.sh]
    - Argumentos
83. Local variables [83_local-variables.sh]
84. Function Example [84_function-example.sh]
    - Return Statement
85. Readonly command [85_readonly-command.sh]
    - Variáveis
    - Funções
    - Readonly
86. Signals and Traps [86_signals-traps.sh]
    - Signals
    - Traps
87. How to debug a bash script [87_debug-bash-script.sh]
    - bash -x
    - Alternativas ao bash -x

Seção: 2 Bash Shell Scripting Tutorial

42. Bash Introduction [42_bash-introduction.sh]

- Usar preferencialmente um sistema operacional Linux, mas o Mac OS e Windows atualmente também suportam programas Bash

➢ **Shell, Shell Script e Bash**

- Um programa shell UNIX interpreta os comandos do usuário inseridos diretamente no terminal ou lidos a partir de um arquivo, chamado shell script ou programa shell
- Os shell script são interpretados e não compilados. Então, quando se escreve um shell script, ele é interpretado pelo sistema operacional e não é preciso compilar para executá-lo
- Existem diferentes tipos de shell. O foco deste curso é o bash shell

➢ **Verificar os tipos de shell suportado pelo SO**

```
$ cat /etc/shells
```

- O sh é um shell ainda usado em sistemas Unix ou ambientes semelhantes. O bash significa born again shell (nasceu depois do shell), pois é uma versão melhorada do sh e atualmente é o shell padrão, usado na maior parte dos sistemas operacionais Linux ou baseado em Unix, incluindo o Mac OS e, hoje em dia, também o Windows

➢ **Verificar localização do bash**

```
$ which bash
```

➢ **Criar hello world bash script**

a. No terminal, executar

```
$ touch 42_bash-introduction.sh
```

✓ É importante notar aqui que a extensão .sh não é necessária para executar seu shell script, mas é útil ao abrir em um editor, pois permite entender que se trata de arquivo shell script
✓ Após criar um arquivo com o comando touch, ao executar `$ ls -al`, as permissões do arquivo são exibidas: permissão de leitura e escrita para você e para o seu grupo de usuário e apenas de leitura para os demais

b. Abrir o arquivo 42_bash-introduction.sh em um editor
c. Escrever, por standard practice, na primeira linha, a localização do bash (desta forma, o interpretador saberá que, neste caso, é um bash shell script)

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

d. Escrever os próximos comandos

➢ **Executar um script**

```
$ ./42_bash-introduction.sh
```

- Caso seja informado "Permissão negada", deve-se primeiro alterar a permissão desse arquivo e depois executá-lo. Para dar permissão de execução:

```
$ chmod +x 42_bash-introduction.sh
```

- Para verificar a permissão de execução:

```
$ ls -al
```

43. using Variables and Comments [43_using-variables-and-comments.sh]

➢ **Comentários**

- Comentários são linhas de código que não são executadas por um script (precedidos por #), mas são úteis para fornecer informações sobre os scripts.

```
# this is comment
echo "Hello World" # this is also a comment
```

➢ **Variáveis**

- Variáveis são 'contêineres' que armazenam alguns dados dentro delas
- Uma variável armazena algum tipo de dado, pode ser string, número ou qualquer outro tipo
- Nome de variáveis não inicia com número
- Há dois tipos de variáveis: variáveis do sistema e variáveis definidas pelo usuário

✓ Variáveis do sistema são criadas e mantidas pelo sistema operacional Linux ou Unix. Estas variáveis são predefinidas pelo sistema operacional e, por convenção, são definidas em maiúsculas. É possível usá-las nos scripts

```
echo $BASH              # informa o nome do bash ou shell
echo $BASH_VERSION      # informa a versão do bash
echo $HOME              # informa o diretório home
echo $PWD               # informa o diretório atual de trabalho
```

✓ Variáveis definidas pelo usuário são criadas e mantidas pelo usuário e geralmente são definidas em minúsculas

```
name=Mark               # declaração da variável name (sem espaços)
echo The name is $name  # utiliza o conteúdo da variável name
                        # (precedido de $)
```

- É possível concatenar strings com o conteúdo de variáveis (do sistema ou definidas pelo usuário)

```
echo Our shell name is $BASH
```

45. Read User Input [45_read-user-input.sh]

- Ler os inputs do terminal no script e inserir em uma variável
- Sempre que quiser receber um input do terminal, usar o comando read

```
echo "Enter name: "
read name
echo "Enterd name: $name
```

➢ **Múltiplos inputs**

- Inserir múltiplos inputs em diferentes variáveis no mesmo comando read
- No comando read, escrever as diferentes variáveis separadas por espaço
- O usuário deve inserir os inputs (valores) separados por espaço

```
echo "Enter names: "
read name1 name2 name3
echo "Names: $name1, $name2, $name3"
```

➢ **Especificar o prompt na mesma linha do input**

- É necessário usar o parâmetro "-p" seguido do prompt (mensagem) entre aspas simples e da variável

```
read -p 'username: ' user_var
echo "username: $user_var"
```

➢ **Tornar o input oculto/silencioso**

- Quando não quiser exibir o que o usuário está digitando, por exemplo, em um input de password
- É necessário usar o parâmetro "-s" seguido da variável

```
read -p 'username: ' user_var
read -sp 'password: ' pass_var
echo
echo "username: $user_var"
echo "password: $pass_var"
```

➢ **Ler os inputs em um array**

- Permitir que o usuário forneça vários inputs e salvá-los dentro de um array
- É necessário usar o parâmetro "-a" seguido da variável/array
- Então, para extrair um array, basta escrever o nome da variável/array e o índice
- O usuário deve inserir os inputs (valores) separados por espaço

```
echo "Enter names: "
read -a names
echo "Names: ${names[0]}, ${names[1]}"
```

➢ **Comando read sem variável**

- O input capturado pelo comando read (sem variável) é atribuído automaticamente a default build-in variable $REPLY

```
echo "Enter name: "
read
echo "Name: $REPLY"
```

47. Pass Arguments to a Bash-Script [47_pass-arguments-to-a-bash-script.sh]

- Sempre que se iniciar um script, pode-se passar algum argumento
- Sempre que você passar um argumento para o script bash, eles são armazenados em um argumento padrão $n. Assim, o primeiro argumento será armazenado em $1. Em seguida, o segundo argumento será armazenado $2 e assim por diante.

```
echo $1 $2 $3 ' > echo $1 $2 $3'
```

- Ao executar o script, deve-se passar os argumentos separados por espaço

```
./47_pass-arguments-to-a-bash-script.sh Mark Tom John
```

- O nome do arquivo shell script (.sh) é armazenado sempre na variável $0
- As variáveis reais, que são realmente passadas em um script como argumentos, começam a partir da variável $1 e, então, $2 até $n

➢ **Passando Argumentos em um Array**

- Deve-se usar a variável/array padrão $@

```
args=("$@")
echo ${args[0]} ${args[1]} ${args[2]}
```

- Ao executar o script, deve-se, da mesma forma anterior, passar os argumentos separados por espaço

```
./47_pass-arguments-to-a-bash-script.sh Mark Tom John
```

- Portanto, o importante a ser observado é sempre que se passar o argumento como um array, o primeiro argumento informado será atribuído ao array no índice 0. Diferentemente da situação anterior, em que a variável $0 armazena o nome do arquivo shell script (.sh)
- Se você quiser imprimir todo o argumento de uma só vez, pode simplesmente usar a variável/array padrão $@, que foi usada para converter os argumentos em um array

```
echo $@
```

➢ **Imprimir o número de argumentos**

- A variável padrão $# imprime o número de argumentos passados para o script bash

```
echo $#
```

49. If Statement (If then , If then else, If elif else) [49_if-statement.sh, 49_user.sh, 49_number.sh]

- Condicionais nos permitem decidir se executar uma ação ou não, esta decisão é tomada pela avaliação de uma expressão. As declarações if são:

```
if [ expression ]
then
  statements
fi

if [ expression ]
then
  statements
else
  statements
fi

if [ expression ]
then
  statements
elif [ expression ]
then
  statements
else
  statements
fi
```

- Sempre coloque espaços antes e depois dos colchetes e parêntesis e ao redor dos operadores e operandos.

➢ **Expressions**

- Uma expressão pode ser: comparação de string, comparação numérica, operadores de arquivo e operadores lógicos e é representada por [expression]:

a. Number Comparisons

-eq	compare if two numbers are equal - is equal to	if ["$a" -eq "$b"]
-ne	compare if two numbers are not equal - is not equal to	if ["$a" -ne "$b"]
-gt	compare if one number is greater than another number - is greater than	if ["$a" -gt "$b"]
-ge	compare if one number is greater than or equal to a number - is greater than or equal to	if ["$a" -ge "$b"]
-lt	compare if one number is less than another number - is less than	if ["$a" -lt "$b"]
-le	compare if one number is less than or equal to a number - is less than or equal to	if ["$a" -le "$b"]

✓ Examples

[n1 -eq n2]	true if n1 same as n2, else false
[n1 -ge n2]	true if n1 greater then or equal to n2, else false
[n1 -le n2]	true if n1 less then or equal to n2, else false

[n1 -ne n2]	true if n1 is not same as n2, else false
[n1 -gt n2]	true if n1 greater then n2, else false
[n1 -lt n2]	true if n1 less then n2, else false

```
count=10
if [ $count -gt 9 ]
then
  echo "condition is true"
fi
```

✓ Se quiser usar os símbolos abaixo, então você precisa usar duplo parêntesis em vez de colchete na declaração if

<	is less than	(("$a" < "$b"))
<=	is less than or equal to	(("$a" <= "$b"))
>	is greater than	(("$a" > "$b"))
>=	is greater than or equal to	(("$a" >= "$b"))

```
count=10
if (( $count >= 9 )) # OR [ $count >= 9 ]
then
  echo "condition is true"
fi
```

b. String Comparison

✓ Diferente de outras linguagens, os símbolos = e == produzem o mesmo resultado

=	compare if two strings are equal - is equal to	if ["$a" = "$b"]
==	compare if two strings are equal - is equal to	if ["$a" == "$b"]
!=	compare if two strings are not equal - is not equal to	if ["$a" != "$b"]
-n	avalia se o tamanho da string é maior que zero	if [-n "$a"]
-z	avalia se o tamanho da string é igual a zero - string is null, that is, has zero length	if [-z "$a"]

✓ Examples

[s1 = s2]	true if s1 same as s2, else false
[s1 != s2]	true if s1 not same as s2, else false
[s1]	true if s1 is not empty, else false
[-n s1]	true if s1 has a length greater then 0, else false
[-z s2]	true if s2 has a length of 0, otherwise false

```
word=abc
if [ $word == "abccccccc" ]
then
  echo "condition is true"
fi
```

✓ Se quiser usar os símbolos abaixo (para verificar a ordem alfabética), então você precisa usar duplo colchetes em vez de colchete único na declaração if

<	is less than, in ASCII alphabetical order	if [["$a" < "$b"]]
>	is greater than, in ASCII alphabetical order	if [["$a" > "$b"]]

```
word=a
if [[ $word < "b" ]]
then
  echo "condition is true"
fi
```

51. File test operators [51_file-test-operators.sh]

- Operadores usados nas instruções if para verificar os arquivos em um shell script
- A flag -e serve para verificar se o arquivo existe ou não

```
if [ -e $file_name ]
then
  echo "$file_name found"
else
  echo "$file_name not found"
fi
```

- A flag -f serve para verificar se o arquivo existe e se é um arquivo normal (regular file) ou não

```
if [ -f $file_name ]
then
  echo "$file_name found"
else
  echo "$file_name not found"
fi
```

- A flag -d serve para verificar se um diretório existe ou não

```
if [ -d $dir_name ]
then
  echo "$dir_name found"
else
  echo "$dir_name not found"
fi
```

- A flag -s serve para verificar se um arquivo não está vazio ou está. Para validar que o arquivo está vazio, usar `$ ls -l` e verificar o tamanho do arquivo

```
if [ -s $file_name ]
then
  echo "$file_name not empty"
else
  echo "$file_name empty"
fi
```

- Para verificar se o arquivo tem a permissão de leitura, então usar a flag -r. Para verificar se o arquivo tem a permissão de escrita, então usar a flag -w. E para verificar se o seu arquivo tem a permissão de execução, então usar a flag -x

- Existem basicamente dois tipos de arquivo: arquivo especial de bloco e arquivo especial de caractere. O arquivo especial de caractere é um arquivo normal que contém algum texto ou dados. Já o arquivo especial de bloco é um arquivo binário, por exemplo, um arquivo de imagem ou vídeo. Para um arquivo especial de bloco, você pode usar a flag -b e, para um arquivo especial de caractere, você pode usar a flag -c

53. How to append output to the end of text file [53_append-output-text-file.sh, 53_test]

- Escrever um script para acrescentar/anexar algum texto ao final de um arquivo já existente

55. Logical 'AND' Operator [55_logical-and-operator.sh]

- Use && em condições separadas

```
if [ "$age" -gt 18 ] && [ "$age" -lt 30 ]
```

- Use -a em uma única condição

```
if [ "$age" -gt 18 -a "$age" -lt 30 ]
```

- Use && em uma única condição

```
if [[ "$age" -gt 18 && "$age" -lt 30 ]]
```

57. Logical 'OR' Operator [57_logical-or-operator.sh]

- Use || em condições separadas

```
if [ "$age" -eq 18 ] || [ "$age" -eq 30 ]
```

- Use -o em uma única condição

```
if [ "$age" -eq 18 -o "$age" -eq 30 ]
```

- Use || em uma única condição

```
if [[ "$age" -eq 18 || "$age" -eq 30 ]]
```

59. Perform arithmetic operations in integers [59_arithmetic-operations.sh]

- Realizar operações aritméticas básicas em números inteiros
- Para realizar operações aritméticas básicas, é preciso do símbolo $ e dos parêntesis duplo. Assim, a expressão é avaliada como uma operação aritmética

```
num1=20
num2=5

echo $(( num1 + num2 )) #adição
echo $(( num1 - num2 )) #subtração
echo $(( num1 * num2 )) #multiplicação
```

```
echo $(( num1 / num2 )) #divisão
echo $(( num1 % num2 )) #resto
```

➢ **Alternativa**

- Usar apenas um parêntesis, a palavra-chave expr e, antes de cada variável, usar o $
- Na multiplicação, usar o símbolo de escape \ (barra invertida) antes da operação (asterisco)

```
echo $(expr $num1 + $num2 )    #adição
echo $(expr $num1 - $num2 )    #subtração
echo $(expr $num1 \* $num2 )   #multiplicação
echo $(expr $num1 / $num2 )    #divisão
echo $(expr $num1 % $num2 )    #resto
```

61. Floating point math operations in bash | bc Command [61_floating-point-math-operations.sh]

- Realizar operações aritméticas em números decimais
- O bash shell script não suporta número decimal em operações aritméticas, usando os métodos dos números inteiros
- Há muitas ferramentas para realizar operações aritméticas em números decimais. Um das mais frequentemente utilizadas é denominada bc, que significa uma calculadora básica e, na maior parte da vezes, vem com a distribuição do Linux
- Trata-se de uma linguagem de calculadora de precisão arbitrária ("An arbitrary precision calculator language"), ou seja, faz mais do que somente operações aritméticas, conforme comando abaixo

```
$ man bc
```

- No lado esquerdo, adicionar a operação aritmética desejada, depois o pipe | e, do lado direito, o comando bc. O lado esquerdo será tratado como uma entrada para o comando bc no lado direito do pipe

```
echo "20.5+5" | bc     #adição
echo "20.5-5" | bc     #subtração
echo "20.5*5" | bc     #multiplicação
echo "20.5%5" | bc     #resto
```

- No entanto, para a operação de divisão é necessário definir uma escala, ou seja, quantas casas decimais terá o resultado. Caso não seja definido, o resultado é apresentado sem casas decimais

```
echo "20.5/5" | bc             #divisão (sem casas decimais)
echo "scale=2;20.5/5" | bc     #divisão (com duas casas decimais)
```

- Usando as variáveis ao invés dos valores nas operações aritméticas

```
echo "$num1+$num2" | bc
echo "$num1-$num2" | bc
```

- No lado esquerdo, definir a operação matemática desejada (no caso, raiz quadrada com escala de 2 casas decimais ou potência com escala de 2 casas decimais), depois o pipe | e, do lado direito, o comando bc -l (para chamar a biblioteca matemática na qual a função raiz quadrada ou potência está)

```
num=27
echo "scale=2;sqrt($num)" | bc -l
echo "scale=2;3^3" | bc -l
```

63. The case statement [63_the-case-statement.sh]

- Após a palavra-chave case, segue-se qualquer expressão ou variável para avaliar. Depois são dispostos os padrões (rígidos ou flexíveis, como uma expressão regular) para comparação, seguido do parêntese. Logo após, são listados os comandos e, em seguida, o ponto-e-vírgula duplo indica que esta declaração case foi concluída. A palavra-chave esac (case, ao contrário) finaliza a declaração case

```
case expression in
  pattern1 )
    statements ;;

  pattern2 )
    statements ;;

  ...

  * )
    statements ;;
esac
```

- Exemplo

```
vehicle=$1
case $vehicle in
  "car" )
    echo "Rent of $vehicle is 100 dollar" ;;
  "van" )
    echo "Rent of $vehicle is 80 dollar" ;;
  "bicycle" )
    echo "Rent of $vehicle is 5 dollar" ;;
  "truck" )
    echo "Rent of $vehicle is 150 dollar" ;;
  * )
    echo "Unknown vehicle" ;;
esac
```

65. The case statement Example [65_the-case-statement-example.sh]

- É avaliado cada caractere informado pelo usuário, classificando-o em caracteres alfabético minúsculos ou maiúsculos, números, caracteres especiais ou desconhecidos
- Para mais informações sobre expressões regulares, pesquisar na Wikipedia

67. Array variables [67_array-variables.sh]

- O bash shell suporta apenas array unidimensional simples e duas formas de declará-los
- Denomine o array e, após o sinal de atribuição =, use parêntesis. Dentro dos parêntesis, cada elemento do array (separado por espaço) é adicionado entre aspas simples

```
os=('ubuntu' 'windows' 'kali')
```

- Adicionar elementos ao array

```
#os[3]='mac'
#os[0]='mac'
os[6]='mac'
```

- Remover elementos do array (os índices não são alterados/reordenados)

```
unset os[2]
```

- Imprimir todos os elementos do array

```
echo "${os[@]}"
```

- Imprimir um elemento específico do array

```
echo "${os[0]}"
```

- Imprimir os índices do array

```
echo "${!os[@]}"
```

- Imprimir o comprimento/tamanho do array

```
echo "${#os[@]}"
```

➢ **Variável como array**

- Qualquer variável é tratada ou comporta-se como um array, mas o valor da variável/array será sempre atribuída ao índice 0

```
string=dasfdsafsadfasdf
echo "${string[@]}"
echo "${string[0]}"    # o elemento sempre ficará no índice 0
echo "${string[1]}"    # não há nenhum elemento
echo "${#string[@]}"   # imprimir o comprimento/tamanho do array
```

69. WHILE Loops [69_while-loops.sh]

- Os loops são usados para executar uma lista de comandos repetidamente

```
while [ condition ]
do
  command1
  command2
  command3
```

```
done
```

- Exemplo

```
while (( $n <= 10 )) # OR [ $n -le 10 ]
do
  echo "$n"
  (( ++n )) # OR (( n++ )) OR n=$(( n+1 ))
done
```

71. using sleep and open terminal with WHILE Loops [71_while-loops.sh]

➢ **Usar sleep com loops**

- Para permitir um delay (atraso/pausa) durante um loop, onde n é quantidade de segundos que a execução do loop pausará.

```
sleep n
```

- Exemplo

```
n=1
while [ $n -le 10 ]
do
  echo "$n"
  (( n++ ))
  sleep 1
done
```

➢ **Abrir um terminal, usando o bash shell script**

- Cada sistema operacional possui alguns terminais. O Ubuntu, por exemplo, possui os terminais gnome e xterm
- Abrir terminal Gnome

```
gnome-terminal &
```

✓ Exemplo

```
n=1
while [ $n -le 3 ]
do
  echo "$n"
  (( n++ ))
  gnome-terminal & #será aberto 3 terminais Gnome
done
```

- Abrir terminal xterm

```
xterm &
```

✓ Exemplo

```
n=1
while [ $n -le 3 ]
do
  echo "$n"
  (( n++ ))
  xterm & #será aberto 3 terminais xterm
done
```

73. Read a file content in Bash [73_read-file-content.sh]

- Existem várias maneiras de ler arquivos, usando o loop while
- A primeira maneira, é usar o input redirection (redirecionamento de entrada) e, a segunda maneira, é ler o conteúdo do arquivo em uma variável e depois imprimi-la
- Existem outros métodos de leitura de arquivos, por exemplo, usando descritores de arquivo (file descriptors)

➢ **Input Redirection (redirecionamento de entrada)**

- A primeira maneira de ler arquivos, usando o loop, é o input redirection (redirecionamento de entrada)
- Após a keyword while, usar o comando read e, em seguida, a variável, na qual será salvo o conteúdo do arquivo linha por linha. Após a keyword done, adicionar "< file.txt". O conteúdo do arquivo é redirecionado para o loop

```
while read p
do
  echo $p
done < 73_read-file-content.sh
```

➢ **2º Método**

- A segunda maneira é ler o arquivo em uma única variável e depois imprimi-la
- O conteúdo do arquivo que é lido, usando o comando cat, é usado como entrada para o comando while, que, neste caso, imprime o conteúdo do arquivo

```
cat 73_read-file-content.sh | while read p
do
  echo $p
done
```

➢ **3º Método**

- Agora, às vezes, é difícil ler os arquivos, usando os dois métodos anteriores, e o problema são alguns caracteres especiais no arquivo, por exemplo, um recuo de linha e outras coisas. Neste caso, deve-se usar IFRS, que significa separador de campo interno e é usado pelo shell para determinar como fazer a divisão de palavras e como reconhecer quais são os limites

```
while IFS=' ' read -r line # while IFS= read -r line
do
  echo $line
done < /etc/host.conf
```

74. UNTIL loop [74_until-loop.sh]

- Until Loop é muito semelhante a While Loop, mas existe uma ligeira diferença. No While Loop, os comandos são executados enquanto a condição for verdadeira, já no Until Loop, os comandos são executados enquanto a condição for falsa

```
until [ condition ]
do
  command1
  command2
  ...
  commandN
done
```

- Exemplo

```
n=1
until [ $n -gt 10 ]
do
  echo "$n"
  n=$(( n+1 ))
done
```

76. FOR loop [76_for-loop.sh]

- For Loop é usado para loops com uma lista de valores e, em seguida, os comandos no loop são executados
- Existem várias sintaxes For Loop

➢ **1ª Sintaxe**

- Na primeira a sintaxe, usa-se a keyword for, declara-se uma variável e então usa-se a keyword in e, por fim, dá-se uma lista de valores. Pode ser, por exemplo, 1 2 3 4 5, ou no formato de lista
- O formato de lista é melhor porque não é preciso escrever cada número, pode-se informar um intervalo entre as chaves {START..END}. Adicionando-se mais dois pontos, poderá fornecer o incremento pelo qual deseja aumentar o valor {START..END..INCREMENT}. O primeiro valor é o valor inicial, após os dois pontos, o segundo valor é o valor final e, em seguida, o terceiro valor, após os dois pontos, é o incremento
- O formato de lista com 3 valores está disponível apenas nas versões superiores a 4.0 do bash enquanto que com 2 valores nas versões superiores a 3.0. A variável $BASH_VERSION informa a versão do bash

```
for VARIABLE in 1 2 3 4 5 .. N
do
  command1
  command2
  commandN
done
```

- Exemplo

```
#for i in 1 2 3 4 5
#for i in {1..10}
for i in {1..10..2}
do
  echo $i
done
```

➢ **2ª Sintaxe**

- Outra maneira de usar o For Loop é semelhante a utilizada na programação C, por exemplo. Então, usa-se três expressões entre os parêntesis. A primeira expressão é usada para inicializar o valor. A segunda, para comparar, ou seja, na segunda expressão é verificada se a condição é satisfeita. E, na terceira expressão, por exemplo, pode-se incrementar o valor declarado na primeira expressão

```
for (( EXP1; EXP2; EXP3 ))
do
  command1
  command2
  commandN
done
```

- Exemplo

```
for (( i=0; i<5; i++ ))
do
  echo $i
done
```

➢ **3ª Sintaxe**

- Pode-se ter uma lista de arquivos ou, ainda, algum comando Linux ou Unix como entrada

```
for VARIABLE in file1 file2 file3
do
  command1 on $VARIABLE
  command2
  commandN
done
```

```
for OUTPUT in $(Linux-Or-Unix-Command-Here)
do
  command1 on $OUTPUT
  command2 on $OUTPUT
  commandN
done
```

78. use FOR loop to execute commands [78_for-loop-execute-commands.sh]

- Na sintaxe básica, após a keyword in é informado a entrada na forma de lista de comandos

```
for command in $(Linux-Or-Unix-Command-Here)
do
  echo $command
done
```

- Os comandos da lista serão executados um por um. No exemplo abaixo, primeiro, o comando ls será executado, em seguida, o comando pwd e, por fim, o comando date. Então é impresso o nome do comando (echo) e depois o comando em si é executado

```
for command in ls pwd date
do
  echo "---------$command---------"
  $command      #isso vai realmente executar esse comando
done
```

- Ao usar o comando asterisco, será carregado todos os itens (arquivos e diretórios) que estão no diretório corrente/atual

```
echo "all the files in directory--------"
for item in *
do
  if [ -f $item ]      #flag -f, se for um arquivo, imprimiremos o nome
  then
    echo $item
  fi
done
```

80. Select loop [80_select-loop.sh]

- Select Loop permite gerar menus fáceis. Portanto, sempre que necessário escrever um script que exija alguns menus, poderá usar Select Loop
- A sintaxe básica do Select Loop consiste na keyword select, em seguida, de uma variável e então a keyword in seguida de uma lista de itens. Entre as keywords do e done, executa-se algumas comandos com base na lista fornecida
- Qualquer que seja a lista fornecida, os itens são apresentados numerados em uma sequência e, em seguida, o prompt solicitará que seja fornecido qualquer número dessa lista
- Select Loop é semelhante a For Loop, mas o que ele faz é, antes de mais nada, ler a lista, fornecer esse tipo de estrutura de menu, então permite selecionar qualquer um dos itens do menu e executar algum comando ou declaração baseada no valor selecionado

```
select varName in list
do
  command1
  command2
  ...
  commandN
done
```

- Exemplo

```
select name in mark john tom bem
do
  echo "$name selected"
done
```

- Select Loop é frequentemente usado com a instrução Case, então, alguns tipos de Cases podem ser usados com o Select Loop. No exemplo a seguir, usou-se apenas o comando echo

para cada valor selecionado, no entanto, pode-se usar uma lógica mais complexa baseada no valor selecionado

```
select name in mark john tom bem
do
  case $name in
    mark )
      echo mark selected
      ;;
    john )
      echo john selected
      ;;
    tom )
      echo tom selected
      ;;
    ben )
      echo ben selected
      ;;
    * )
      echo "Error please provide the no. between 1..4"
  esac
done
```

81. Break and continue [81_break-continue.sh]

- **Declaração Break**

- A instrução break é usada para sair do loop atual antes de sua execução normal. Então, sempre que você quiser sair do loop prematuramente, usar a declaração break

```
for (( i=1 ; i<=10 ; i++ ))
do
  if [ $i -gt 5 ]
  then
    break
  fi
  echo "$i"
done
```

- Então é assim que você pode usar break para sair do loop ou quebrar a execução normal do loop prematuramente antes de sua finalização normal

- **Declaração Continue**

- A instrução continue é um pouco diferente do break. Sempre que usar continue, ele pula a execução normal daquela iteração. Então, o que vier depois da keyword continue é ignorado e o programa irá para a próxima iteração

```
for (( i=1 ; i<=10 ; i++ ))
do
  if [ $i -eq 3 -o $i -eq 6 ]
  then
    continue
  fi
  echo "$i"
```

```
done
```

- Quando o programa ou o script vê a keyword continue no loop, então o que vier depois será ignorado e seu programa irá para a próxima iteração

82. Functions [82_functions.sh]

- Assim como qualquer outra linguagem de programação, o Bash também suporta função embora com um pouco de limitação
- Função é uma sub-rotina, é um bloco de código que implementa um conjunto de operações. Então, para qualquer usuário, é como uma caixa preta. Tem um nome e implementa uma funcionalidade. O usuário pode usar esta função uma ou várias vezes
- Existem duas maneiras de declarar funções

a. A primeira notação consiste na keyword function seguida do nome da função e, por fim, executa alguns comandos

```
function name() {
  commands
}
```

b. A segunda notação consiste em dá o nome diretamente à função seguido de parênteses e, entre as chaves, estão os comandos

```
name() {
  commands
}
```

- Para invocar uma função em um script, basta usar o nome dessa função. A sequência com que as funções são invocadas é importante, mas a declaração pode ser feita em qualquer sequência sendo antes de invocá-las

```
function Hello(){
  echo "Hello"
}

quit () {
  exit
}

Hello
echo "foo"
quit
```

➢ **Argumentos**

- Na declaração da função, os argumentos são denominados como $i, com i iniciando em 1 até n, dependendo de quantos argumentos esta função possua. A variável $1 é para o primeiro argumento, $2 para o segundo argumento, $3 para o terceiro e assim por diante
- Ao invocar uma função, os argumentos são passados logo após o nome da função

```
function print(){
```

```
  echo $1 $2 $3
}

print Hello World Again
```

83. Local variables [83_local-variables.sh]

- Por default, toda variável declarada no script, independente se dentro ou não de uma função, é uma variável global e isso significa que pode ser acessada de qualquer lugar no script

```
function print(){
  name=$1
  echo "the name is $name"
}

echo "The name is $name"            # The name is
name="Tom"
echo "The name is $name : Before"   # The name is Tom  : Before
print Max                           # the name is Max
echo "The name is $name : After"    # The name is Max : After
```

- Às vezes, a variável definida em uma função deve permanecer como variável local, ou seja, não deve ser alterada fora da função. Para isso, existe o comando local. E sempre que for adicionado o comando local à variável (em uma função), esta se tornará local e apenas poderá ser usada dentro da função. Desta forma, a função é apenas uma execução local e não afetará a variável global

```
function print(){
  local name=$1
  echo "the name is $name"
}

echo "The name is $name"            # The name is
name="Tom"
echo "The name is $name : Before"   # The name is Tom : Before
print Max                           # the name is Max
echo "The name is $name : After"    # The name is Tom : After
```

84. Function Example [84_function-example.sh]

- Script para verificar se um arquivo existe ou não. Portanto, sempre que um usuário executar o script com um argumento e esse argumento for o nome de um arquivo, por exemplo, o script verificará se esse arquivo existe ou não e, em seguida, imprimirá esta informação

➢ **Return Statement**

- Na programação bash, ao contrário das outras linguagens como, por exemplo, C, na avaliação de uma condition, o valor true é representado por 0 e o valor false é representado por 1.

```
is_file_exist() {
  local file="$1"
  [[ -f "$file" ]] && return 0 || return 1
```

```
  #if [ -f "$file" ]
  #then
  #  return 0 #0 is true
  #else
  #  return 1 #1 is false
  #fi
}

if ( is_file_exist "$1" )
then
  echo "File found"
else
  echo "File not found"
fi
```

85. Readonly command [85_readonly-command.sh]

- O comando readonly pode ser usado com variáveis e funções
- E pelo próprio nome, entende-se que é usado para fazer com que variáveis ou funções sejam somente para leitura, ou seja, não podem ser sobrescritas (ter o valor alterado)

➢ **Variáveis**

- Após a atribuição de um valor à variável, usar o comando readonly. A partir deste ponto, ocorrerá sempre o warning "readonly variable" ao tentar atribuir um novo valor à variável, ou seja, não será mais possível alterar o seu valor

```
var=31
readonly var
var=50 # warning - readonly variable
echo "var => $var" # var => 31
```

➢ **Funções**

- As funções também podem ser feitas somente para leitura
- Para tornar uma função somente leitura, usar a keyword readonly seguido da flag -f
- Quando tentar sobrescrever uma função readonly, ocorrerá o warning "readonly function"

```
hello() {
  echo "Hello World"
}

readonly -f hello

hello() {
  echo "Hello Word Again" # warning - readonly function
}

hello   # Hello World
```

➢ **Readonly**

- Executar o comando readonly sem uma variável ou função atrelada, irá exibir todas as variáveis do sistema e também do script (definidas pelo usuário) que são somente leitura

- readonly ou readonly -p apresentam o mesmo resultado

```
readonly
#readonly -p
```

- Já para exibir todas as funções somente leitura do script (definidas pelo usuário), usar readonly -f

```
readonly -f
```

86. Signals and Traps [86_signals-traps.sh]

➢ **Signals**

- Existem alguns sinais que podem ser usados para encerrar um processo (ou um script em execução)
- Enquanto um script está em execução, alguns cenários (provocados por sinais) podem acontecer:

a. Usuário pode pressionar Ctrl + C para finalizar o script. Este sinal Ctrl + C é chamado de sinal de interrupção (SIGINT value 2)
b. Usuário pode pressionar Ctrl + Z para parar o script. Este sinal Ctrl + Z é chamado de sinal de suspensão (SIGTSTP value 18,20,24)
c. Usuário pode matar o processo (script em execução), através do comando kill, utilizando outro terminal, sendo o PID conhecido. O sinal -9 é chamado de kill signal (SIGKILL value 9): `kill -9 <PID>`

✓ Para saber o PID de um script em execução

```
echo "pid is $$"
```

- Para acessar a página do manual de signal e saber mais sobre os sinais, no terminal, digitar

```
man 7 signal
```

➢ **Traps**

- Há alguns cenários (provocados por sinais), nos quais o script é interrompido, durante a execução e antes de sua finalização normal. No entanto, algum sinal ou comportamento inesperado pode ocorrer, interrompendo a execução do script. Por isso, o comando trap é usado
- O comando trap permite ao script capturar o sinal, interromper a execução e depois limpar o sinal dentro do script
- O comando trap permite saber o tipo de sinal utilizado, por exemplo, kill -9 <PID>, Ctrl + Z ou Ctrl + C, capturar esse tipo de sinal e então fazer algum tratamento antes de encerrar
- Quando algum comando/sinal de saída é detectado, é executado o comando trap, de acordo com o sinal recebido. Então, quando o sinal for recebido, executará o comando especificado logo após a keyword trap

```
trap "comando_especificado" n°_signal
```

- Exemplo

```
trap "echo Exit signal is detected" SIGINT    # (SIGINT OR 2) Ctrl + C
```

- Cada sinal tem o seu valor e são sempre maiores que zero. Então, a saída/sinal zero é um sinal de sucesso

```
trap "echo Exit command is detected" 0
echo "Hello world"
exit 0
```

- O comando trap não consegue capturar dois sinais (SIGKILL e SIGSTOP). Desta forma, nenhum comando será executado após esses sinais e, portando não devem ser usados

```
trap "echo Exit signal is detected" SIGKILL SIGSTOP
#(SIGKILL OR 9) kill -9 <PID>
```

- Exemplo

✓ Quando receber os sinais 0, 2 ou 15, remover um arquivo e sair do script. O ponto e vírgula é usado para combinar dois ou mais comandos

```
file=/home/wjuniori/git/bash-shell/86_file.txt
trap "rm -f $file && echo file deleted; exit" 0 2 15
# SIGTERM kill -15 <PID>
```

87. How to debug a bash script [87_debug-bash-script.sh]

- Às vezes, quando as coisas não saem de acordo com o esperado, é preciso determinar o que exatamente faz com que o script falhe
- Para alguns erros, por exemplo, de sintaxe, o bash lhe dará o erro conhecido e até o número da linha onde o problema está ocorrendo
- Para situações mais complexas, o bash fornece extenso recurso de depuração/debug

➢ **bash -x**

- O mais comum é executar o shell script com a flag -x

```
bash -x ./87_debug-bash-script.sh
```

- Os passos ou comandos são executados em sequência e são impressos (mostra passo-a-passo o que está acontecendo no script)
- Assim, sempre que ocorrer algum problema, você estará imediatamente ciente de que o script não está funcionando de acordo com o esperado e onde o problema está ocorrendo

➢ **Alternativas ao bash -x**

- Executar o script normalmente

```
./87_debug-bash-script.sh
```

- Dentro do script

a. Adicionar, na primeira linha, a flag -x

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

b. Iniciar/ativar o debug a partir de um determinado ponto, adicionando neste ponto

```
set -x
```

c. Marcar um bloco de código onde o debug deve ocorrer (marcar o início e o fim do bloco)

```
set -x

file=/home/wjuniori/git/bash-shell/87_file.txt
trap "rm -f $file && echo file deleted; exit" 0 2 15
# SIGTERM kill -15 <PID>

set +x
```