ShellFunctions

April 15, 2024

1 Shell Functions?

Creating Functions To declare a function, simply use the following syntax –

```
function_name () {
   list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

In shell scripting, a function is a named block of code that can be reused in multiple places within a script. Functions help to modularize the code, making it more readable and maintainable.

Here's the basic syntax for defining a function:

[ ]:
```
function_name () {
  # commands
}
```

You can call a function simply by writing its name:

[ ]:
```
function_name
```

Functions can accept arguments, which are accessed within the function as `$1`, `$2`, etc.:

[ ]:
```
function_name () {
  echo "The first argument is $1"
  echo "The second argument is $2"
}

function_name arg1 arg2
```

Functions can also return a value to the calling code using the `return` statement. The return value must be a number:

[ ]:
```
function_name () {
  # commands
  return 0  # success
}
```

```

function_name
echo "The return value is $?"
```

2 Pass Parameters to a Function?

In shell scripting, you can pass arguments to a function in the same way you pass arguments to a script or command, by providing them after the function name. Inside the function, you can access these arguments using the special variables `$1`, `$2`, `$3`, etc., where `$1` corresponds to the first argument, `$2` to the second argument, and so on.

Here's an example of a shell function that takes two arguments:

[ ]:
```
greet() {
  echo "Hello, $1"
  echo "How are you today, $2?"
}

greet Alice Bob
```

In this example, the `greet` function is defined to take two arguments. When the function is called with the arguments `Alice` and `Bob`, the function will print:

[ ]:
```
Hello, Alice
How are you today, Bob?
```

You can pass any number of arguments to a function, and they will be assigned to the variables `$1`, `$2`, etc. in order. If you need to access all arguments as a list, you can use the special variable `$@`.

[ ]:
```
print_args() {
  for arg in "$@"; do
    echo "Argument: $arg"
  done
}

print_args Alice Bob Charlie
```

This will output:

[ ]:
```
Argument: Alice
Argument: Bob
Argument: Charlie
```

3 Returning Values from Functions?

In shell scripting, functions return a status code, not values as in other programming languages. The status code is a numerical value where 0 typically indicates success, and non-zero values indicate different types of errors.

You can set the status code of a function using the `return` statement followed by a number. Here's an example:

[ ]:
```
my_function() {
  # some commands
  return 0  # success
}
```

In this example, `my_function` will return a status code of 0.

If you want to "return" a value from a function, you typically do so by echoing the value and then capturing that output when you call the function. Here's an example:

[ ]:
```
my_function() {
  local my_result='some value'
  echo "$my_result"
}

result=$(my_function)  # Call the function and capture its output
echo "The result is $result"
```

In this example, `my_function` echoes a value, which is then captured into the `result` variable. This is a common pattern for "returning" values from functions in shell scripting.

4 Nested Functions?

In shell scripting, you can define a function inside another function. This is known as a nested function. The nested function can be called from the function in which it is defined.

Here's an example:

[ ]:
```
outer_function() {
  echo "This is from the outer function."

  inner_function() {
    echo "This is from the inner function."
  }

  inner_function
}

outer_function
```

In this example, `inner_function` is defined inside `outer_function`. When `outer_function` is called, it first prints a message, then calls `inner_function`, which prints another message.

However, it's important to note that in shell scripting, functions are global, not block-scoped. This means that once a function is defined, it can be called from anywhere in the script, not just from the function in which it was defined. So even though `inner_function` is defined inside `outer_function`, it can be called from anywhere in the script after it has been defined.

[ ]:
```
outer_function() {
  inner_function() {
    echo "This is from the inner function."
  }

  inner_function
}

outer_function
inner_function  # This will also work
```

In this example, `inner_function` is called both from inside `outer_function` and from outside it. Both calls will work, because `inner_function` is global.

5 Function Call from Prompt?

In shell scripting, you can call a function from the command prompt or terminal after it has been defined in the current shell session.

First, you need to define the function in the shell. You can do this by typing the function directly into the terminal, or by sourcing a script file that contains the function definition with the `source` command or the `.` (dot) command:

[ ]:
```
source script.sh  # or . script.sh
```

After the function has been defined, you can call it just like any other command:

[ ]:
```
function_name arg1 arg2
```

For example, if you have the following function defined in your script:

[ ]:
```
Hello () {
   echo "Hello World $1 $2"
   return 10
}
```

You can call this function from the command prompt like this:

[ ]:
```
Hello Zara Ali
```

This will print “Hello World Zara Ali” and return 10. The return value can be accessed with the special variable `$?`:

[ ]:
```
echo "Return value is $?"
```

This will print “Return value is 10”.

6 Thank You!