PairExercise_Functions

HDS

2024-06-24

Creating your own functions

We have been using function of various types regularly so far.

Many of the libraries we use are simply collections of useful functions

Functions have

- a name
- input variables, values the user sends to the function -some variables have default values, which are values they take on unless the user sends them a different value

-one output variable -R is slightly odd in that it has only one output variable allowed, most languages allow many. We can use a list or a dataframe, or other structure if we want or need to include a lot of different information in a return variable

Most languages have functions, R and Lisp use them extensively.

I build complex analyses as a series of functions that carry out the steps I need. There are a number of advantages to this:

1.) I can test each function, one by one, to be sure it is correct.

2.) My testing is done on small pieces of code, not long ones.

3.) The functions are modular, I can re-use them again later, or create variations on them for other problems.

4.) Working on a team, different team members can be assigned to create and test different functions, so it is easy to split up the work.

Example

Here is a function that raises some input value x to a power n, where we are inputting x and n

the name is x2power, with inputs x and n

here is the declaration syntax

```
x2power<-function(x,n)
{
  # this is the function x2Power, that raises an input value x to the input
  # power n
  # written by HD Sheets, 6/24/2024 for DSE5002
  y=x^n
  return(y)
}
```

We use a function by calling it

```
x2power(3,2)
```

```
## [1] 9
```

Action Required

Write your self a function that takes in three variables, a, b, and c and returns (a+b)/c

```
a_plus_b <-function(a,b,c)
{

  y=(a+b)/c
  return(y)
}

a_plus_b(1,2,3)
```

```
## [1] 1
```

Default values

In the declaration line, we can set default values for the variable

If the user does not input a value, the function will

```
x2power<-function(x,n=2)
{
  # this is the function x2Power, that raises an input value x to the input
  # power n
  # written by HD Sheets, 6/24/2024 for DSE5002
  # updated on 6/24/2024 to include a default value on n of 2

  y=x^n
  return(y)
}

x2power(4)
```

```
## [1] 16
```

Error trapping

It is a wise idea to check that the function has received valid input data.

It there is an error, the function should tell you why and then return some value that indicates an error.

It is common to return -1 or NULL to indicate an error

```r
x2power<-function(x,n=2)
{
  # this is the function x2Power, that raises an input value x to the input
  # power n
  # written by HD Sheets, 6/24/2024 for DSE5002
  # updated on 6/24/2024 to include a default value on n of 2, HDS
  # further updated on 6/24/2024 to include some error trapping

  if(!is.numeric(x))
  {
    cat("Please enter only numeric values of x \n")
    return(NULL)
  }
  if(!is.numeric(n))
  {
     cat("Please enter only numeric values of n \n")
     return(NULL)
  }
  if(length(n)>1)
  {
     cat("Please enter a scalar value for n \n")
     return(NULL)
  }


  y=x^n
  return(y)
}

x2power(4)
```

```
## [1] 16
```

```r
x2power("bob","37")
```

```
## Please enter only numeric values of x

## NULL
```

```r
x2power(1:5,4:7)
```

```
## Please enter a scalar value for n

## NULL
```

Action Required

Add some basic error trapping to the function you wrote.

Make sure c is not zero

```
a_plus_b <-function(a,b,c)
{
  if(c==0)
{
  cat("Please enter a value for c greater than 0 \n")
     return(NULL)
  }
  y=(a+b)/c
  return(y)
}

a_plus_b(1,2,3)
```

```
## [1] 1
```

```
a_plus_b(1,2,0)
```

```
## Please enter a value for c greater than 0
```

```
## NULL
```

Do you really have to write extensive error trapping?

If it is a function only you will ever use, and you don't plan on using it constantly or sharing it, no, you can ship error trapping.

Never skip documenting it though! Failure to document it means you will rewrite it again later....

#Returning multiple variables

We can use a list or a dataframe to "pack" multiple values into a single variable

```
critical_KPI<-function(x,y,z)
{
  #example of returning two variables from a function, HDS 6/24/2024
  kpi_one=x/y
  kpi_two=(x^2+z)/sqrt(y)

  my_output_list=data.frame(kpi1=kpi_one,kpi2=kpi_two)
  return(my_output_list)
}

my_res=critical_KPI(2,3,4)

my_res$kpi1
```

```
## [1] 0.6666667
```

```
my_res$kpi2
```

```
## [1] 4.618802
```

Variable Scope

It is not uncommon to assign variable within a function.

These values are not visible outside the function, and don't alter values in the workspace. This protects your variables from unexpected changes when using a function

```r
a=5
my_function_a<-function(x)
{
  #demonstration of variable scope, part a
  # HDS, 6/24/2024

  # define a local varialbe a that is only used in the function
  a=2
  y=a*x
  return(y)
}

#let's check the value of a
print(a)
```

```
## [1] 5
```

```r
# now run the function
z=my_function_a(5)
print(z)
```

```
## [1] 10
```

```r
# did the variable a in the workspace change?
print(a)
```

```
## [1] 5
```

This is good!

Running the function didn't alter my variable a in the workspace, so if I was using "a" for some other use in the workspace, calling the function wouldn't produce odd results.

But…

Functions can inherit values from levels "above" them,

So a function can inherit a value from the workspace

```r
b=2

my_function_b<-function(x)
{
  y=x^b
}

print(b)
```

```
## [1] 2
```

```
z=my_function_b(6)
print(z)
```

```
## [1] 36
```

```
print(b)
```

```
## [1] 2
```

Notice that b is not defined in the function to be 2, the function just uses the b value from the next "level" up in the system.

With this feature you can do some cool things, but it is very unpredictable!

If you need to use variable inheritance, define a function within a function

```
c=2

my_function_c<-function(x)                    #outer function
{
  c=3

  my_inner_function<-function(x)           #defined within my_function_c
  {
    y=c*x
    return(y)
  }

  youter=my_inner_function(x)
  return(youter)
}

print(c)                                        # c value in the workspace
```

```
## [1] 2
```

```
z=my_function_c(6)                              # inner function uses value defined
                                                # in my_function_c
print(z)
```

```
## [1] 18
```

```
print(c)
```

```
## [1] 2
```

The scope control in R is not typical, most languages protect variables more carefully than this.

Try to avoid using variables from outside a function scope, it can generate errors that are very hard to locate and eliminate.

R has ways to change variables that are out of scope within a function, so that a function could alter a variable in the workspace.

Just don't do this.