Tutorial 5 sample answer

Note that due to the nature of programming, there are multiple ways to write the functions. Any way of writing is fine as long as desired output is achieved.

1

Command/code:

```
vector_mean <- function(x){
    n = length(x)
    output = sum(x)/n
    return(output)
}
```

Example output:

```
> vector_mean(c(1,2,3))
[1] 2
> mean(c(1,2,3))
[1] 2
```

Comment:
The function `length()` gives the length of *x* (how many elements in *x*). Then to calculate the mean, just take the sum of *x* using `sum(x)` and divide by size of *x*.

2

Command/code:

```
multiple <- function(a,b){
    remainder <- a %% b
    if(remainder == 0){
        output=TRUE
    }
    if(remainder != 0){
        output=FALSE
    }
    return(output)
}
```

Example output:

```
> multiple(3,2)
[1] FALSE
> multiple(3,3)
[1] TRUE
> multiple(27,5)
[1] FALSE
> multiple(27,9)
[1] TRUE
```

Comment:
The important part here is to use the if statements. The code can also use if else statement.

3

Command/code:

```
square_root <- function(x){
    if(!is.numeric(x) | x<0){
        solution <- NA
        error <- TRUE
    } else {
        solution <- sqrt(x)
        error <- FALSE
    }
    return(list(solution=solution, error=error))
}
```

Example output:

```
> square_root("A")
$solution
[1] NA

$error
[1] TRUE

> square_root(9)
$solution
[1] 3

$error
[1] FALSE

> square_root(-9)
$solution
[1] NA

$error
[1] TRUE
```

Comment:
The function `is.numeric()` is used to check whether *x* is numerical or not.
Since multiple outputs are required from the function, the `list()` is used.

4

Command/code:

```
i <- 0
product <- 1
while(product < 10000000){
    i <- i+1
    product <- product*i
}
print(i)
```

Output:

```
> print(i)
[1] 11
```

Comment:
In the code above, we initialize *i* as 0, and product as 1. Then at each loop, we add 1 to *i*, and multiply product by *i*, until the product is greater than or equal to 10 million.

5

Command/code:

```
Fibonacci <- function(n){
  output <- rep(1,n)
  if(n > 2){
    for(i in 3:n){
      output[i] <- output[i-1] + output[i-2]
    }
  }
  return(output)
}

#OR

Fibonacci <- function(n){
    if(n == 1){
        output <- 1
    } else if(n==2){
        output <- c(1,1)
    } else{
        output <- c(1,1)
        for(i in 3:n){
            Fn <- output[i-1]+output[i-2]
            output <- c(output,Fn)
        }
    }
    return(output)
}

#OR

Fibonacci <- function(n){
    output <- rep(0,n)
    for(i in 1:n){
        if(i==1){
            output[i]=1
        } else if(i==2){
            output[i]=1
        } else{
            output[i]=output[i-1]+output[i-2]
        }
    }
    return(output)
}
```

Example output:
```
> Fibonacci(1)
[1] 1
> Fibonacci(2)
[1] 1 1
> Fibonacci(3)
[1] 1 1 2
> Fibonacci(5)
[1] 1 1 2 3 5
```

Comment:
In the first example, we initialize output to be a vector of 1s with length *n*. Then for the 3rd term and above, we set the term according to the Fibonacci formula.

In the second example, we will set output to be 1 if $n = 1$, or c(1,1) if $n = 2$. For $n \geq 3$, we will have to calculate using the equation for Fibonacci number. The line `c(output,Fn)` will combine the vector of output, with the new Fibonacci number `Fn`.

In the third example, we initialize output by a vector of 0's with length *n* using `rep(0,n)`. Another alternative is to use `output <- vector("numeric", n).`

6 Command/code:
```
summation <- 0
for(i in 1:10){
    die_value <- sample(1:6,1)
    summation <- summation + die_value
}
print(summation)
```

Output:
```
> print(summation)
[1] 41
```

Comment:
In the code, we initialize summation to be 0, and at each loop, we add the die value to the summation. Due to randomness, the output may differ if you try it yourself.

7

Command/function:

```
quadratic <- function(a, b, c){
    disc <- b^2 - 4*a*c
    if(disc==0){
        x <- -b/(2*a)
        number_solution <- 1
    } else if(disc>0){
        x1 <- (-b-sqrt(disc))/(2*a)
        x2 <- (-b+sqrt(disc))/(2*a)
        x <- c(x1,x2)
        number_solution <- 2
    } else{
        x <- NA
        number_solution <- 0
    }
    return(list(solution = x, number_of_solution =
           number_solution))
}
```

Example output:

```
> quadratic(1,0,-9)
$solution
[1] -3  3

$number_of_solution
[1] 2

> quadratic(1,-6,9)
$solution
[1] 3

$number_of_solution
[1] 1

> quadratic(1,0,9)
$solution
[1] NA

$number_of_solution
[1] 0
```

Comment:
Many students do not return a list from the function, but instead print out hte message of the solution and number of solutions. This is not fully correct.

The reason why we do not want to just print out the solution and number of solutions is because sometimes, the output from the function may be used in the next line of code. Only printing results will not allow us for this.

To return multiple output, use the list environment, as shown in the example.

8	Command/codes:
	`price <- 1`
	`day <- 0`
	`while(price < 1.5 & price > 0.75){`
	`    price <- price + sample(c(-0.05,0,0.05),1)`
	`    day <- day+1`
	`}`
	`print(day)`
	
	Example output:
	`> print(day)`
	`[1] 42`
	
	Comment:
	Some students set a limit for the number of days it simulates. For example, some may use “for(i in 1:1000)”. For large enough upper limit, the function may work fine. But it is better to use the while loop instead.
	
	Some made the wrong condition in the while loop. E.g. “while(price <= 1.5 & price >= 0.75)” is incorrect because it will run the code even when price is 1.5 or 0.75.
	
	Using break when price is 1.5 or 0.75, as done by some students can mitigate these errors.