Functions, loops, conditionals

Week 2, Computational Genomics

Recap/what's ahead

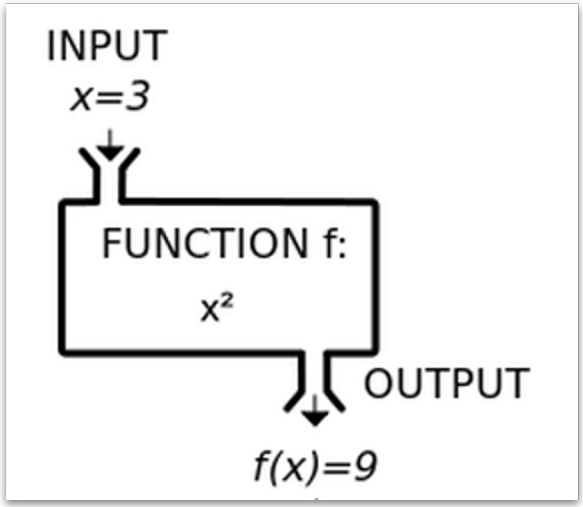
What has been discussed and practiced:

- How to write expressions on data types and data objects using functions and operations.

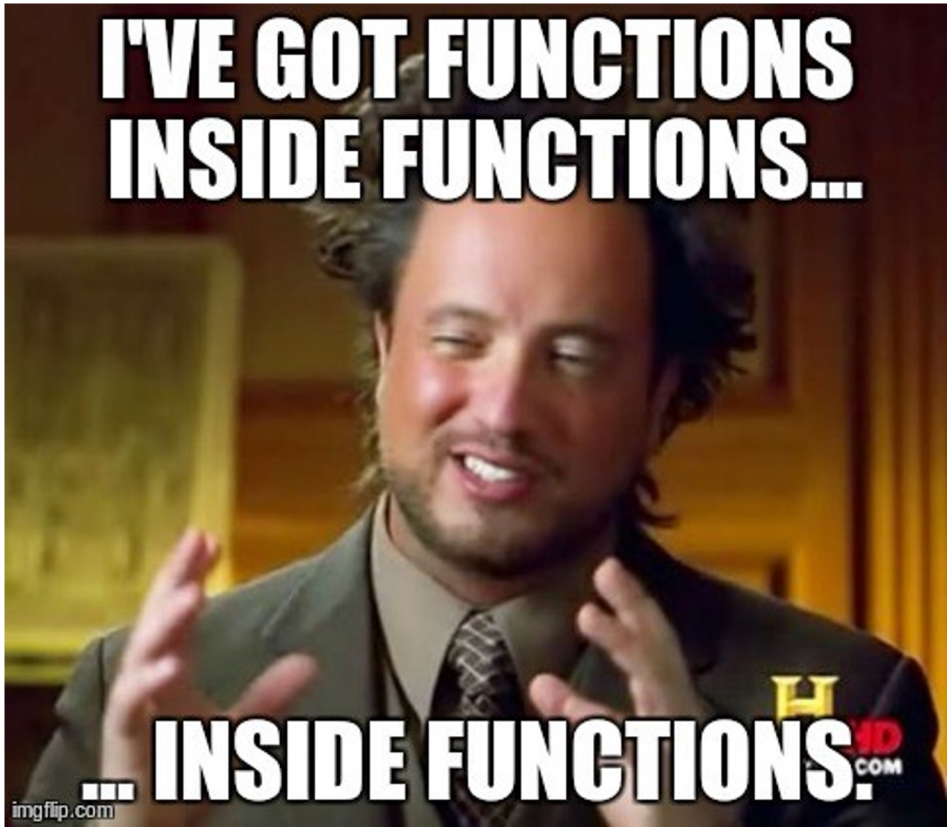
Let's go deeper in the machinery:

- How does one write a function?
- How does one automate repetitive tasks?
- How does one deal with complex logical scenarios?

Writing functions



Writing functions



Writing functions

Function definition consist of a *function* statement that has a comma-separated list of named *function arguments*, and a *return* expression. The function definition is assigned to a variable in the global environment.

In order to use the function, one defines or import it, then one calls it.

```
addFunction = function(argument1, argument2) {

        result = argument1 + argument2

        return(result)

}

z = addFunction(3, 4)
```

Composing functions

{ } represents variable scoping: within each { }, if variables are defined, they are stored in a *local environment*, and is only accessible within { }. All function arguments are stored in the local environment. The overall environment is called the *global environment* and can be also accessed within { }.

```
addFunction = function(argument1, argument2) {
        result = argument1 + argument2
        return(result)
}
z = addFunction(3, 4)
```

Console

```
> addFunction =
function(argument1, argument2)
{ ... }
```

Environment

```
addFunction = function(...)
```

Console

```
> addFunction =
function(argument1, argument2)
{ ... }

> z = addFunction(3, 4)
```

Environment

```
addFunction = function(...)
```

Local Environment

```
argument1 = 3

argument2 = 4
```

Console

```
> addFunction =
function(argument1, argument2)
{ ... }

> z = addFunction(3, 4)
```

```
result = argument1 +
argument2
```

Environment

```
addFunction = function(...)
```

Local Environment

```
argument1 = 3

argument2 = 4

result = 3 + 4
```

Console

```
> addFunction =
function(argument1, argument2)
{ ... }

> z = addFunction(3, 4)
```

```
result = argument1 +
argument2

return(result)
```

Environment

```
addFunction = function(...)
```

Local Environment

```
argument1 = 3

argument2 = 4

result = 7
```

Console

```
> addFunction =
function(argument1, argument2)
{ ... }

> z = addFunction(3, 4)
```

Environment

```
addFunction = function(...)

z = 7
```

Composing functions

Scoping makes functions modular: consider the following alternative:

```
x = 3

y = 4

addFunction = function(argument1, argument2) {

        result = x + y

        return(result)

}

z = addFunction(x, y)

w = addFunction(10, -5)
```

Why won't this do what we want????

Console

```
> x = 3

> y = 4

> addFunction =
function(argument1, argument2)
{ ... }

> w = addFunction(10, -5)
```

Environment

```
x = 3

y = 4

addFunction = function(...)
```

Console

```
> w = addFunction(10, -5)
```

```
result = x + y
```

Environment

```
x = 3

y = 4

addFunction = function(...)
```

Local Environment

```
argument1 = 10

argument2 = -5

result = 3 + 4
```

Console

```
> w = addFunction(10, -5)
```

```
result = x + y

return(result)
```

Environment

```
x = 3

y = 4

addFunction = function(...)
```

Local Environment

```
argument1 = 10

argument2 = -5

result = 3 + 4
```

Console

```
> w = addFunction(10, -5)
```

Environment

```
x = 3

y = 4

addFunction = function(...)

w = 7
```

Composing a function

From the first homework, you looked at chr. ploidy in the human genome.

Create your own function: *somaticToGermline*, which takes the ploidy vector and halves it, as if somatic cells undergo meiosis.

Input argument: a numeric vector of chromosome copies.

Output: a numeric vector of chromosome copies, halved.

Try it out, and *test it* to show that it works.

Composing a function

From the first homework, you looked at chr. ploidy in the human genome.

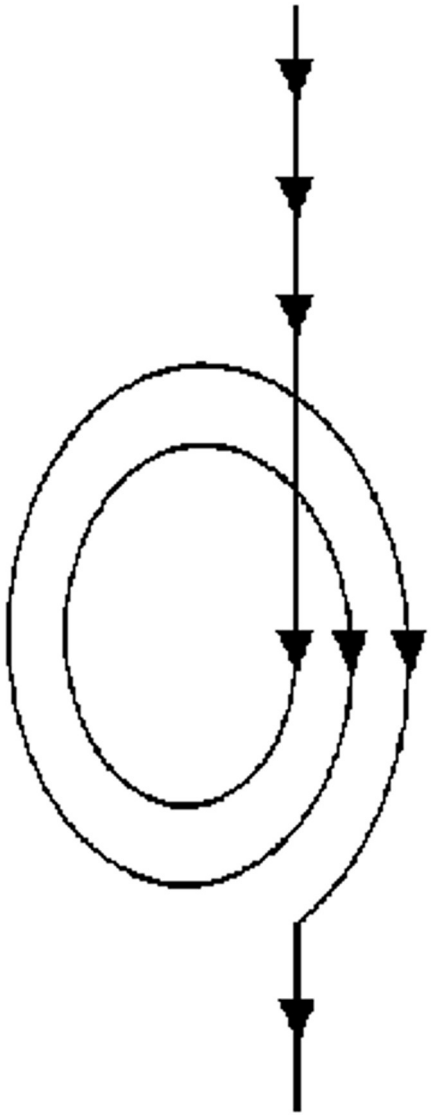
Create your own function: *numNonDiploid*, which takes the ploidy vector and returns the number of chromosomes that are not diploid (!= 2).

Input argument: a numeric vector of ploidy.

Output: a numeric value.

Try it out, and *test it* to show that it works.

```
numNonDiploid = function(myNumChrs) {
        nonDiploids = myNumChrs[myNumChrs != 2]
        return(length(nonDiploids))
}
numChrs = rep(2, 23)
nonDiploids_numChrs = numNonDiploid(numChrs)
```



Looping

```
for (i in my_vector) {

        ...

}
```

Within *for-loop* iteration, a chunk of code within a local scope is repeated many times. Within each iteration, a local *looping variable* re-assigns its value to elements from a *looping vector*.

Console

```
> sum = 0

> for(i in 1:10) {

        sum = sum + i

}
```

Environment

```
sum = 0
```

Local Environment

```
i = 1
```

Console

```
> sum = 0

> for(i in 1:10) {

        sum = sum + i

}
```

Environment

```
sum = 1
```

Local Environment

```
i = 1
```

Console

```
> sum = 0

> for(i in 1:10) {

        sum = sum + i

}
```

Environment

```
sum = 3
```

Local Environment

```
i = 2
```

Console

```
> sum = 0

> for(i in 1:10) {

        sum = sum + i

}
```

Environment

```
sum = 6
```

Local Environment

```
i = 3
```

Console

```
> sum = 0

> for(i in 1:10) {

        sum = sum + i

}
```

Environment

```
sum = 55
```

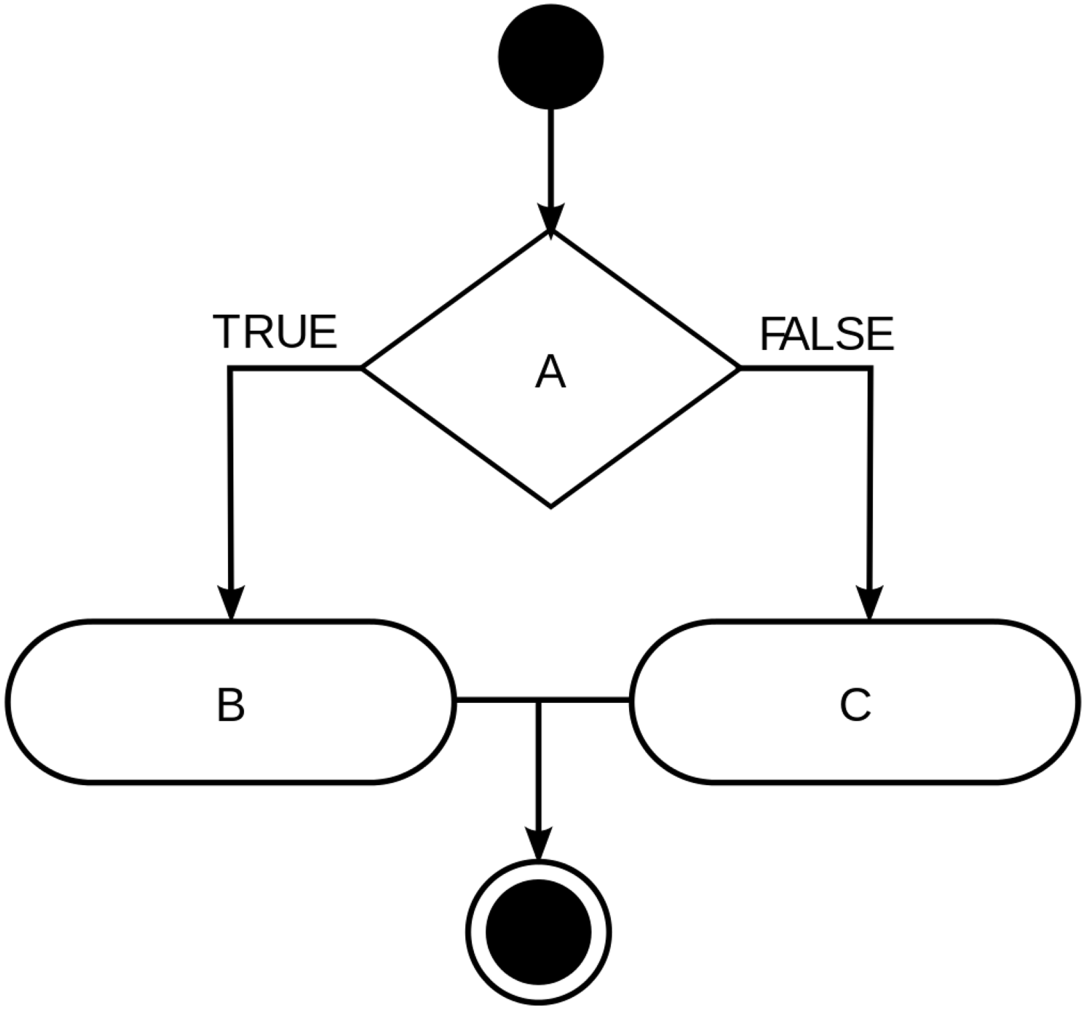
Looping

```
numChrs = rep(2, 23)

for(i in 1:length(numChrs)) {

        numChrs[i] = numChrs[i] / 2

}
```

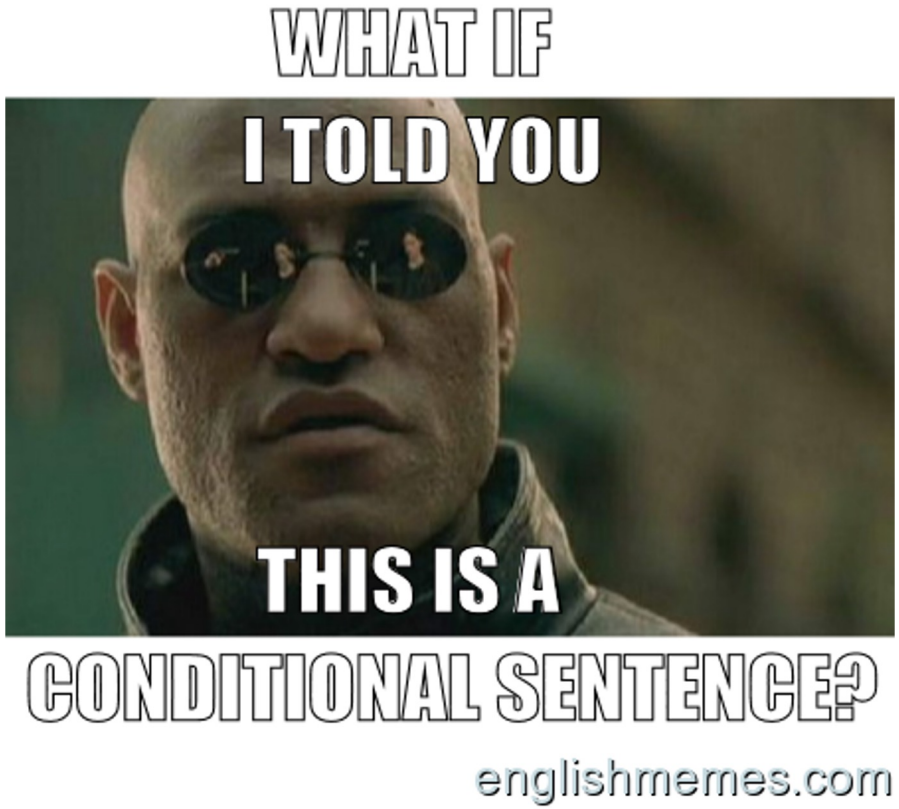
Compare it to:

```
numChrs = numChrs / 2
```

Conditionals



Conditionals



Conditionals

```
if(boolean_value) {

        ...

}
```

```
if(boolean_value_1) {

        …

}else if(boolean_value_2) {

        …

}else {

...

}
```

In a **conditional statement**, if the logical value tested (usually by comparison operation) is evaluated as True, then the body code within the local environment will run. If the logical value tested is evaluated as False, then the body code within the local environment will not run. Longer conditional statements can be built from *else if* and *else* conditional statements.

Conditionals

```
if(length(numChrs) == 23) {
        print(“Human chromosome detected.”)
else{
        print(“Non-human chromosome detected.”)
}
```

Let's make one together!

somaticToGermline is a function that will give us the number of chromosomes someone has divided by 2, but includes the following:

1) Will reject the input if the number of chromosomes is not 23
2) Will round our chromosome number to the nearest multiple of 2 if toRound input is TRUE
3) Otherwise, it will just divide the chromosome number by 2

```
somaticToGermline = function(myNumChrs, toRound) {
        if(length(myNumChrs) != 23) {
                stop("Invalid input!")
    }
    if(toRound) {
        return(round(myNumChrs / 2))
        else {
                return(myNumChrs / 2)
        }
}
somaticToGermline(numChrs, True)
somaticToGermline(numChrs, False)
```

Function arguments can be set as *optional* if one assigns a value in the function definition.

```
template = c("A", "C", "G", "T", "T", "C")

coding = rep(NA, length(template))

for(i in 1:length(template)) {

          nuc = template[i]

          if(nuc == "A") {

          result = "T"

          }else if(nuc == "T") {

          result = "A"

          }else if(nuc == "C") {

          result = "G"

          } else if(nuc == "G")  {

          result = "C"

          } else {

                    result = "N"

     coding[i] = result

}
```

Function on *iris*

Write the following function:

Input arguments
- A dataframe variable (expecting iris dataset)
- A string variable indicating which species to work with - setosa, versicolor, virginica

Output
- A dataframe variable

The function should do the following in its body:
- Using the input dataframe, double the value of Petal.Width *only* for the Species indicated in the second input argument.
- Return this new dataframe.

Function telephone

Think of a function to implement on the ploidy vector *or* iris dataframe.

Explain to your partner each argument in the function, what the function task is, and what the return value is. Be clear about the data type!

Implement the function your partner gives you, and test with a few examples to see whether it works.

Share it back to your partner!

Additional Resources

Datacamp: Introduction to R. Sections 1, 2, 3: Conditionals, Functions, Loops.
https://learn.datacamp.com/courses/intermediate-r

R Programming for Data Science. Chapters 13 and 14.
https://bookdown.org/rdpeng/rprogdatascience/