# Functions: Arguments and Scopes

# Reminder: defining functions and their structure

Variable in which Function is stored

Required arguments

optional arguments

```
calculator<- function(x, y, operation='addition')
{
  switch(operation,
         addition = x + y,
         subtraction = x - y,
         multiplication = x * y,
         division = x / y,
    stop("operation not recognized"))
}
```

Body of the function

```
> calculator(1,3, 'subtraction')
[1] -2
> calculator(1,3, 'multiplication')
[1] 3
> calculator(2,3, 'multiplication')
[1] 6
> calculator(2,3, '4')
[1] "operation not recognized"
```

> calculator(2,3, 4) #??

```
calculator<- function(x, y, operation='+')
{
  if (operation == "+")
  {
    x + y
  }
  else if (operation == '-')
  {
    x - y
  }
  else if (operation == '*')
  {
    x * y
  }
  else if (operation == '/')
  {
    x / y
  }
  else
  {
    print('operation not recognized')
  }
}
```

# Arguments input when calling a function

In how many ways can I input arguments when calling a function?

```
calculator<- function(x, y, operation='addition')
{
  switch(operation,
        addition= x + y,
        subtraction = x - y,
        multiplication = x * y,
        division = x / y,
   stop("operation not recognized"))
}
```

1. Full argument name and value assigned with an '='. E.g. calculator(x=1, y=3, operation='subtraction') > -2

2. Partially matching argument name and value assigned with an '='. E.g. calculator(x=1,y=3,op='subtraction') >-2

3. Positional matching, the value is assigned to the arguments in order of appearance. E.g. calculator(1,3,'subtraction') >-2

These methods of providing arguments are processed in this order. They can be mixed and matched.

Calculator(x=1,2,'subtraction')
>-1

Calculator(x=5,2,op='addition')
>7

Calculator(x=1,2,opa='subtraction')
>Error in calculator(1, x = 2, opa = "subtraction") : unused argument (opa = "subtraction")

Calculator(operator='subtraction', y=1,2)
>1

Calculator(1)
>Error: y is missing

# Creating functions with an arbitrary number of arguments

A function to calculate the mean of an arbitrary number of means calculated on an arbitrary number of vectors

The . . . argument

- Can be used in conjunction with other arguments, any unmatched argument will be assigned to . . ., even the ones with non-matching tags.
- Can be of any mix of types, vectors, variables, lists...
- Allows flexibility by unrestricting the number of arguments

```
meanMeans<-function(...)
{
  data<- list(...)
  means<- numeric(0)
  for (i in data)
  {
    means<- c(means, mean(i))
  }
  mean(means)
}
```

# Variable evaluation inside a function: Arguments and the Scope.

```
Farness<- function(x, y, factor=-1)
{
  distance <-log2(x/y)
  if (distance < 0)
  {
    distance <- distance * factor
  }
  print(distance)
}
```

Define a metric to check how ‘far’ two numbers are. Useful to check how consistent NGS replicates are.

Arguments are evaluated *lazily*. No argument is evaluated before being required by the function.

Variables referenced inside functions are searched in:

- The scope of the function
- The enclosing blocks if any
- The global environment

If no variable with that name is found the function throws an error

REMEMBER: variables defined inside functions, mask any variable with the same name in outer environments, *but only for the duration of the function*

These variables are called local variables

# Returning values: functions that output more than 1 result

```
meanMeans<-function(...)
{
  data<- list(...)
  means<- numeric(0)
  for (i in data)
  {
    means<- c(means, mean(i))
  }
  mean(means)
}
```

If nothing is specified, the value output by a function is the last statement evaluated before the end.

The return function forces the function to terminate and returns the specified value.

```
binaryMeanMeans<-function(...)
{
  data<- list(...)
  means<- numeric(0)
  for (i in data)
  {
    means<- c(means, mean(i))
  }

  if (mean(means) > 5)
  {
    return(1)
  }
  else
  {
    return(0)
  }
}
```

# Quick and dirty anonymous functions: all of the advantages, ~~none of the~~ less typing!

Disposable functions that do not need to be reused.

Follow the same rules as named functions.

```
Farness<- function(x,y)
{
  distance <-log2(x/y)
  return(distance)
}

Farness(1,3)
```

```
(function(x,y){distance <-log2(x/y);
return(distance)})(1,3)
```

Incredibly useful for functions that take other functions as arguments, such as….

Apply!

Margin: 1 for rows
2 for columns

Dataframe or matrix

Function!

```
Apply( x,   1,    mean)
```