# Primitives in the R Environment

**PSS SUMMER SCHOOL**

Anton Badev, Daniel Nikolic, Justin Skillman

June 21, 2017

# Outline

- Data structures
  - Atomic vectors
  - Operations
  - Variables
  - Additional data types
- Functions
  - Environment and scope
  - Libraries
- Control flow
  - If Else statements
  - For While loops

# Data Types

Primitive Types

- *Integer* - no decimal places
- *Double* - float with decimal precision
- *Complex* - complex numbers (with imaginary components)
- *Logical* - True or False (boolean values)
- *Character* - all textual data

Other Types

- *Date* - dates in yyyy-mm-dd format
- *Factor* - categorical variables

# Examples

```
> typeof(c(1L,2L,3L,4L,5L))
[1] "integer"

> typeof(c('a','b','car'))
[1] "character"

> typeof(c(1,2,3,4,5))
[1] "double"

> typeof(c(TRUE, FALSE, T, T, F))
[1] "logical"

> typeof(c(5 + 1i, 9 + 2i))
[1] "complex"

> typeof(as.Date(c('2010-01-01','2011-01-01')))
[1] "double"

> typeof(factor(c('Male','Female')))
[1] "integer"
```

# Atomic Vector

Sequence of data elements of the same type

- Primitive types exist only as 1-element vectors in R
- Most basic data type which others build upon
- Have different operators depending on type
- Properties: type, length, attributes

# Primitive Operations

```
> 5 + 10  # Addition
[1] 15
> 12 - 4  # Subtraction
[1] 8
> 8 * 7   # Multiplication
[1] 56
> 36/6    # Division
[1] 6
> 8**3    # Exponent
[1] 64
> 15 %% 4 # Modulus
[1] 3
```

# Automatic Type Coercion

R will automatically change certain element data types to the same common type when creating an atomic vector (dynamically typed language)

```
> c(1,2,3,4,'5')
[1] "1" "2" "3" "4" "5"

> c(1,1,1,0,T,T,F)
[1] 1 1 1 0 1 1 0
```

# Manual Type Coercion

The data type of a vector can be changed manually by using coercion functions when inputs are compatible

```
> as.numeric(c(1,2,3,4,'5'))
[1] 1 2 3 4 5

> as.logical(c(1,1,1,0,T,T,F))
[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE

> as.character(c(1,1,1,0,T,T,F))
```

# Manual Type Coercion

The data type of a vector can be changed manually by using coercion functions when inputs are compatible

```
> as.numeric(c(1,2,3,4,'5'))
[1] 1 2 3 4 5

> as.logical(c(1,1,1,0,T,T,F))
[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE

> as.character(c(1,1,1,0,T,T,F))
[1] "1" "1" "1" "0" "1" "1" "0"
```

# Data Broadcasting

When operating on two vectors, smaller vector is repeated in sequence until two vectors of of the same size

```
> c(10,20,30,40,50,60)/10
[1] 1 2 3 4 5 6

> c(10,20,30,40,50,60)/c(10,2)
```

# Data Broadcasting

When operating on two vectors, smaller vector is repeated in sequence until two vectors of of the same size

```
> c(10,20,30,40,50,60)/10
[1] 1 2 3 4 5 6

> c(10,20,30,40,50,60)/c(10,2)
[1]  1 10  3 20  5 30

> c(10,20,30,40,50,60)/c(1,2,3,4,5,6)
[1] 10 10 10 10 10 10
```

# Variable Assignment/Modification

We can think about variables as containers in which we store data. We can use these containers just as we do the variables stored inside of them

```
> a <- c(10,20,30,40,50,60)

> b <- c(1,2,3,4,5,6)

> a/b
[1] 10 10 10 10 10 10
```

Assigning a new value to a variable which already contains data will overwrite the old data and replace it with the new data

```
> b <- 2

> a/b
[1]  5 10 15 20 25 30
```

# Indexing into Atomic Vectors

Once a variable has been assigned data, we can access a subset of that data by providing the indices of the desired data.

```
> a <- c('a','p','p','l','e')
> a
[1] "a" "p" "p" "l" "e"

> a[2]
[1] "p"

> a[2:4]
[1] "p" "p" "l"

> a[c(T,T,F,T,T)]
[1] "a" "p" "l" "e"
```

# More Structures

- *Atomic Vector* - Basic building block of R Data
- *List* - Similar to vectors but can contain multiple types
- *Matrix* - two dimensional vectors which are useful when dealing with linear algebra
- *Array* - extension of matrices for n dimensions
- *Data Frame* - Can contain multiple data types and is very conducive to tabular data formats

# Data Structure Big Picture

| Dimension | Homogenous | Heterogenous |
|---|---|---|
| 1 | Vector | List |
| 2 | Matrix | Data Frame |
| 3 | Array | |

# List

A list is a vector which can contain multiple data types. It can also contain tags to reference different elements stored.

```
> list(c(1,2,3,4),c('apple'))
[[1]]
[1] 1 2 3 4

[[2]]
[1] "apple"

> list(digits=c(1,2,3,4),fruits=c('apple'))
$digits
[1] 1 2 3 4

$fruits
[1] "apple"
```

# List Indexing

```
> lst <- list(digits=c(1,2,3,4),fruits=c('apple'))

# Indexing by Position (Double bracket)
> lst[[2]]
[1] "apple"

# Indexing by tag
> lst$fruits
[1] "apple"
```

# Matrix

A matrix can store a two dimensional array of elements of the same primitive type. These structures are useful when dealing with matrix multiplication and other methods in linear algebra.

```
> mat <-  matrix(c(1,2,3,4,5,6),2,3)
> mat
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> mat[2,2]
[1] 4
```

# Array

Work essentially like matrices but with n dimensions.

```
> array(data=seq(1,12,1), dim=c(2,3,2))

, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

# Array Indexing

```
> ar <- array(data=seq(1,12,1), dim=c(2,3,2))

> ar[1,1,2]
[1] 7

> ar[1:2,1:2,1]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

# Data Frame

The primary data type for many data manipulation instances, the data frame can hold columns of different data types. Each column can be though of as its own atomic vector in the order of observations.

```
> data.frame(fruits=c('Apple','Banana','Carrot'),
calories=c(100,120,25))
```

```
  fruits calories
1  Apple      100
2 Banana      120
3 Carrot       25
```

# Data Frame Indexing

```
> df <- data.frame(fruits=c('Apple','Banana',
            'Carrot'), calories=c(100,120,25))

# Extracting a column
> df$calories
[1]  100 120 25

# Indexing like a matrix
> df[1,2]
[1] 100

# Conditional indexing
> df[df$fruits=='Banana',2]
[1] 120
```

# Examining Data Structure

There are several functions to used to determine the attributes of a given data structure.

```
> str(df)
'data.frame': 3 obs. of  2 variables:
 $ client : Factor w/ 3 levels "Alice","Bob",..: 1 2 3
 $ balance: num  500 1000 800
> View(df)
```

Additional functions are specific to certain data types

```
> nrow(df)
[1] 3
> ncol(df)
[1] 2
> nrow(c(1,2,3,4,5))
NULL
```

# Functions

Procedures which are followed given some input arguments to result in some desired output. Components: arguments, body, environment.

```
# Calculate the Hypotenuse of a right triangle
# INPUT  : a,b - two shorter edges of the triangle
# OUTPUT : c - length of hypotenuse of right triangle
calc_pyth <- function(a,b){
  c_sq <- a**2 + b**2
  c    <- sqrt(c_sq)
  return(c)
}

> calc_pyth(3,4)
[1] 5
> calc_pyth(a=3,b=4)
[1] 5
```

# Default Arguments

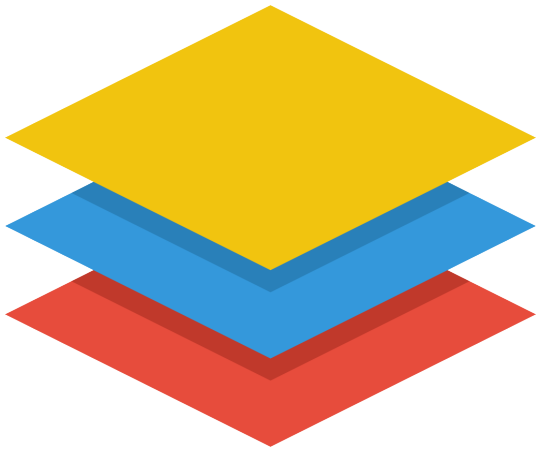
Functions can be defined with default arguments which the user can then have the option to change or remain with the default

```
calc_pyth <- function(a,b=15){
  c_sq <- a**2 + b**2
  c    <- sqrt(c_sq)
  return(c)
}
```

```
> calc_pyth(8)
[1] 17
> calc_pyth(3,4)
[1] 5
```

# Variable Scope

Variables can only be accessed in their respective environments. There are two general levels:



- *Global variables* - accessed any time throughout the program
- *Local variables* - accessed only while the variable is in scope

Local variables are in scope when they have been defined in the current function frame.

# Accessing Global Variables

```
foo <- function(b){
  c <- a + b
  return(c)
}

a <- 5 # Global Variable

> foo(10)
[1] 15
```

# Modifying Global Variables

```
bar <- function(x){
  a <- x + 5
  return(a)
}

a <- 5 # Global Variable

> bar(20)

> a
```

# Modifying Global Variables

```
bar <- function(x){
  a <- x + 5
  return(a)
}

a <- 5 # Global Variable

> bar(20)
[1] 25
> a
[1] 5
```

# Modifying Global Variables

```
bar <- function(x){
  a <<- x + 5
  return(a)
}

a <- 5 # Global Variable

> bar(20)
[1] 25
> a
[1] 25
```

# Local Variable Scope

```
inc <- function(){
  a <- 10
}
calc <- function(a,b){
  c <- b * 10
  inc()
  return(a + c)
}

> a <- 50
> b <- 100
> calc(a,b)
```

# Local Variable Scope

```
inc <- function(){
  a <- 10
}
calc <- function(a,b){
  c <- b * 10   # Deleted after return
  inc()         # Cannot modify local variables in frame
  return(a + c)
}

> a <- 50
> b <- 100
> calc(a,b)
[1] 1050
```

# Scoping Rules Visual

We can think of every function as being in its own world (environment). Every time a function is called a new world is generated.

- Each function makes a copy of given arguments in its environment
- Global variables can be accessed across environments
- Once a function is returned, its environment is destroyed
- (non-hierarchical) environments do not communicate–cannot access others variables

# Function Libraries

We do not always have to write all our functions from scratch. Import functions from R's community libraries:

library([library name])

```
>library(ggplot2)
```

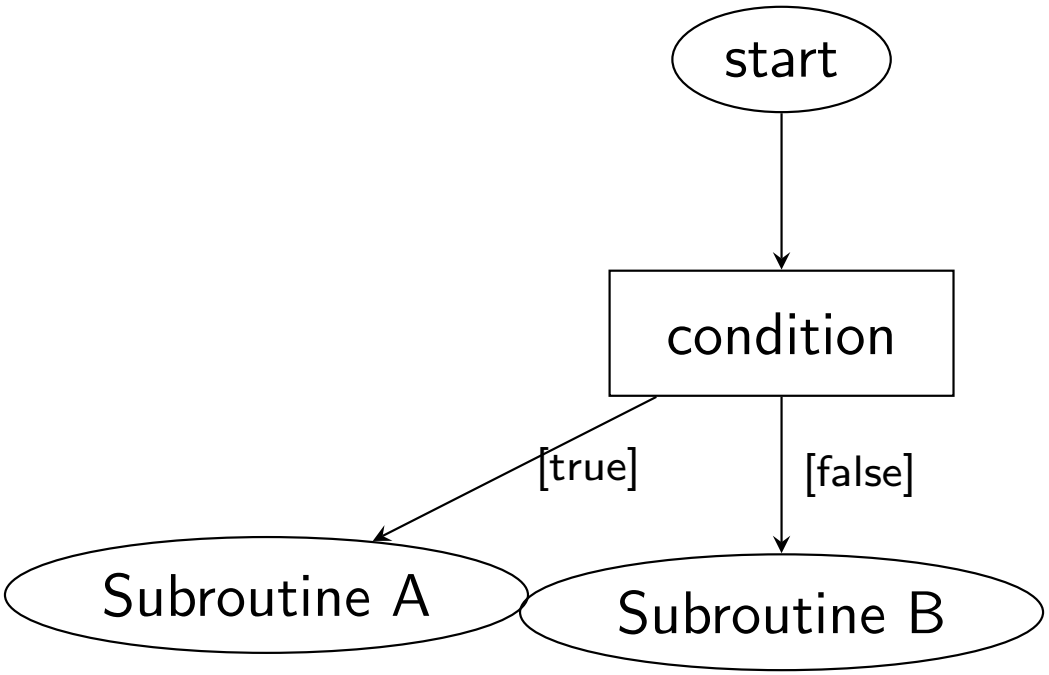
Will load the ggplot2 package which we will use for plotting later. Think of all functions in the base package as a library that is loaded by default.

```
>ls(package:base)
```

```
>help(print)
>help("print")
>?print
```

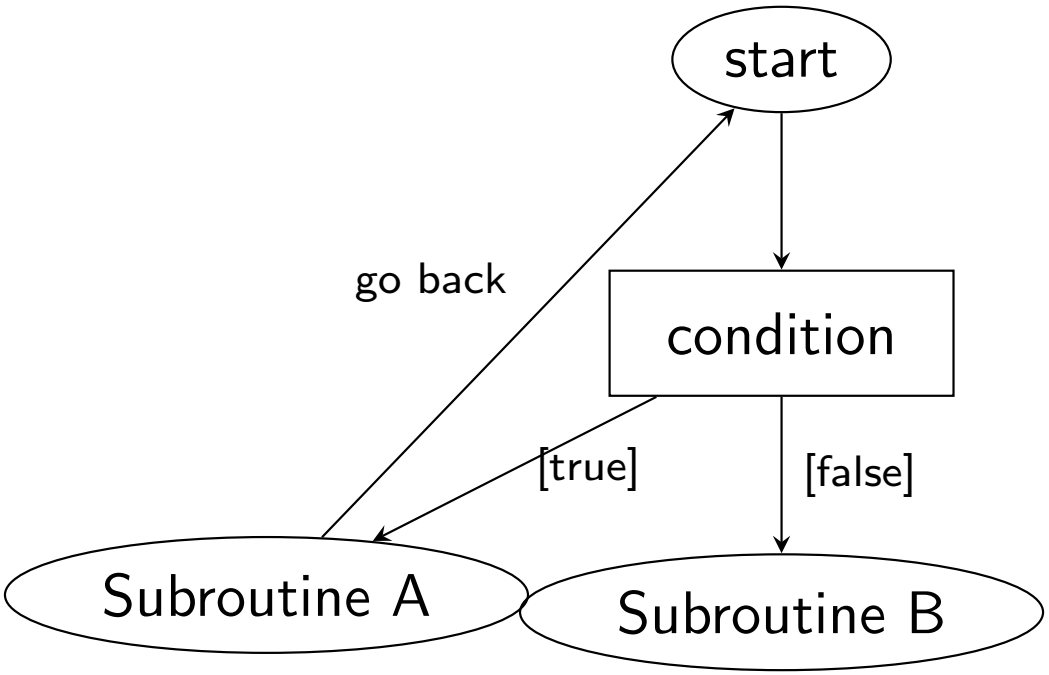
# Control Flow

Statements which govern which parts of your code are run. Usually these are accompanied by a condition which governs which subroutine to execute.



# Control Flow

Statements which govern which parts of your code are run. Usually these are accompanied by a condition which governs which subroutine to execute.



# Boolean Expressions

## Arithmetic Comparisons

| Operator | Description |
|---|---|
| == | Equals to |
| != | Not Equals to |
| > | greater than |
| >= | greater than or equal to |
| < | less than |
| <= | less than or equal to |

## R Indicator Functions

is.numeric(x), is.na(x), is.atomic(x)

# Boolean Expressions (Example)

## Arithmetic Comparisons

```
> a <- 5
> b <- 10
> a > b
[1] FALSE
```

## R Indicator Functions

```
> is.numeric(a)
[1] TRUE
```

# if else Statement

Branch execution given the result of a boolean expression.
Statement Syntax:

```
if([Conditional Expression]){
    [Execute subroutine A...]
}else{
    [Execute subroutine B...]
}
```

**Example**

```
> a <- 5; b <- 10
> if(a>b){
+   print('This condition is true')
+ }else{
+   print('This condition is false')
+ }

[1] "This condition is false"
```

# For Loop

Iterate over a set of values and executes a subroutine repeatedly during each of the iterations.

**Statement Syntax**

```
for([local variable] in [set]){
    [Execute subroutine A given local variable...]
}
```

During each iteration the local variable will be assigned to a different value in the given set until all values in the set are exhausted.

# While loop

Execute a subroutine repeatedly until the while condition evaluates to False and the loop is "Broken".

**Statement Syntax**

```
While([Conditional Expression]){
    [Execute subroutine A...]
}
```

# Loop Example

## For Loop

```
for(i in c(1,2,3,4)){
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
```

## While Loop

```
i <- 1
while(i < 5){
  print(i)
  i <- i + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
```

# Summary & assignment 1

- Data structures
  - Atomic vectors
  - Operations
  - Variables
  - Additional data types
- Functions
  - Environment and scope
  - Libraries
- Control flow
  - If Else statements
  - For While loops
- Assignment 1
  - There are 3 parts: R, LaTeX, Bonus