# Lecture 1: Introduction and Basics

*Statistical Computing, 36-350*

*Monday October 31, 2015*

## Outline

- Course overview and mechanics
- Cool example
- Built-in data types
- Built-in functions and operators
- First data structures: vectors and arrays

## Why good statisticians learn to program

- **Independence**: otherwise, you rely on someone else giving you exactly the right tool
- **Honesty**: otherwise, you end up distorting your problem to match the tools you have
- **Clarity**: turning your ideas into something a machine can do refines and clarifies your thinking
- **Fun**: these were the best of times (these were the worst of times)

## How this class will work

- Professor Ryan Tibshirani
- TAs Yo Joong Choe, Bryan Hooi, Sangwon Hyun, Kevin Lin, Taylor Pospisil
- No programming knowledge presumed
- Some statistics knowledge presumed
- Work mostly in R; some other languages introduced
- Class will be cumulative, so keep up with the readings and assignments!

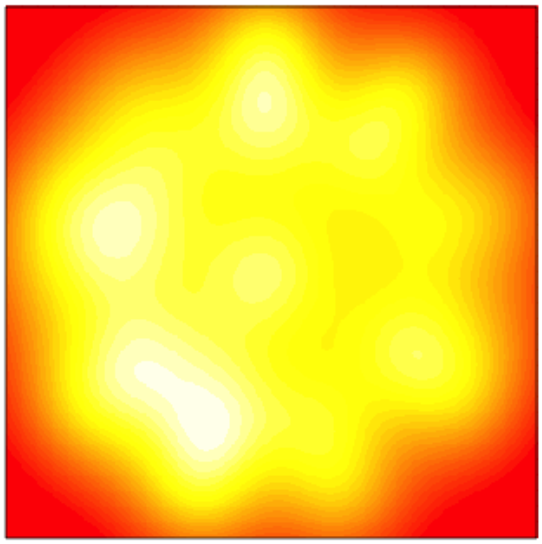
---

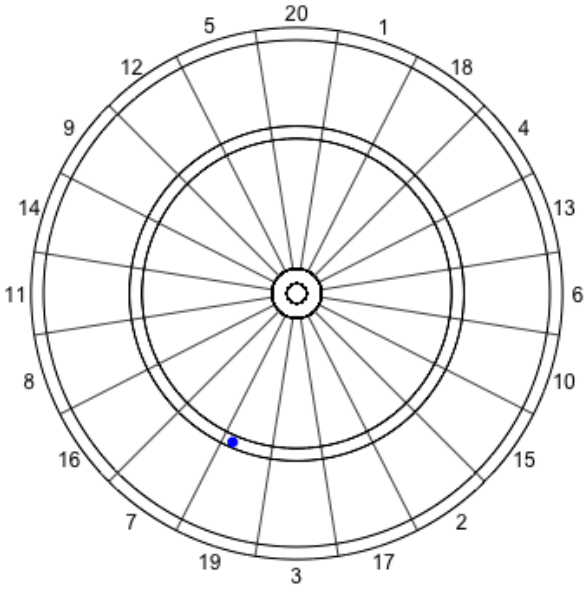
- Two lectures per week: concepts, methods, examples
- Weekly lab to try things out and get fast feedback (10%)
- Weekly homework to do longer and more complex things (40%)
- Midterm project (2 weeks) (20%)
- Final group project (1 month) (30%)

Class website: http://www.stat.cmu.edu/~ryantibs/statcomp/.
This will host class info, schedule, lecture notes, homework assignments, etc.

For announcements and discussions, sign up for the Piazza group. Blackboard will be used to collect submissions, and keep track of grades

## Cool example: statistics + darts






Statistical tools to analyze your darts game!

## This class in a nutshell: functional programming

Two basic types of things, or objects: **data** and **functions**

- **Data**: things like 7, "seven", 7.000, and $\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$
- **Functions**: things like `log`, `+` (takes two arguments), `<` (two), `%%` (two), and `mean` (one)

> A function is a machine which turns input objects, or **arguments**, into an output object, or a **return value** (possibly with side effects), according to a definite rule

---

Programming is writing functions to transform inputs into outputs

Good programming ensures the transformation is done easily and correctly

Machines are made out of machines; functions are made out of functions, like $f(a, b) = a^2 + b^2$

> The route to good programming is to take a big transformation and **break it down** into smaller ones, and then break those down, until you come to tasks which are easy (using built-in functions)

## Before functions, data

At base level, all data can represented in binary format, by **bits** (i.e., TRUE/FALSE, YES/NO, 1/0). Some basic data types:

- **Booleans**: Direct binary values: `TRUE` or `FALSE` in R
- **Integers**: whole numbers (positive, negative or zero), represented by a fixed-length block of bits
- **Characters**: fixed-length blocks of bits, with special coding; **strings**: sequences of characters
- **Floating point numbers**: an integer times a positive integer to the power of an integer, as in $3 \times 10^6$ or $1 \times 3^{-1}$
- **Missing or ill-defined values**: `NA`, `NaN`, etc.

# Operators

- **Unary**: take just one argument. E.g., `-` for arithmetic negation, `!` for Boolean negation
- **Binary**: take two arguments. E.g., `+`, `-`, `*`, and `/` (though this is only a partial operator). Also, `%%` (for mod), and `^` (again partial)

Examples:

```
-7
```

```
## [1] -7
```

```
7+5
```

```
## [1] 12
```

```
7-5
```

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

---

```
7*5
```

```
## [1] 35
```

```
7^5
```

```
## [1] 16807
```

```
7/5
```

```
## [1] 1.4
```

```
7 %% 5
```

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

```
7 %/% 5
```

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

## The R console

Basic interaction with R is by typing in the **console**, i.e., **terminal** or **command line**

You type in commands, R gives back answers (or errors)

Menus and other graphical interfaces are extras built on top of the console

## Comparison operators

These are also binary operators; they take two objects, like numbers, and give back a Boolean. Examples:

```
7 > 5
```

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

```
7 < 5
```

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

```
7 >= 7
```

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

---

```
7 <= 5
```

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

```
7 == 5
```

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

```
7 != 5
```

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

(Note: `==` is a comparison operator, `=` is not!)

## Logical operators

These basic ones are `&` (and) and `|` (or). Examples:

```
(5 > 7) & (6*7 == 42)
```

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

```
(5 > 7) | (6*7 == 42)
```

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

```
(5 > 7) | (6*7 == 42) & (0 != 0)
```

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

```
(5 > 7) | (6*7 == 42) & (0 != 0) | (9-8 >= 0)
```

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

(The double forms `&&` and `||` are different! We will see them later)

## More types

The `typeof()` function returns the data type

`is.`*foo*`()` functions return Booleans for whether the argument is of type *foo*

`as.`*foo*`()` (tries to) "cast" its argument to type *foo*, to translate it sensibly into such a value

---

```
typeof(7)
```

```
## [1] "double"
```

```
is.numeric(7)
```

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

```
is.na(7)
```

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

```
is.na(7/0)
```

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

```
is.na(0/0)
```

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

(Why is `7/0` not `NA`, but `0/0` is?)

---

```
is.character(7)
```

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

```
is.character("7")
```

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

```
is.character("seven")
```

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

```
is.na("seven")
```

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

---

```
as.character(5/6)
```

```
## [1] "0.833333333333333"
```

```
as.numeric(as.character(5/6))
```

```
## [1] 0.8333333
```

```
6*as.numeric(as.character(5/6))
```

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

```
5/6 == as.numeric(as.character(5/6))
```

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

(Why is the last evaluation `FALSE`?)

## Data can have names

We can give names to data objects; these give us **variables**

A few variables are built in:

```
pi
```

```
## [1] 3.141593
```

Variables can be arguments to functions or operators, just like constants:

```
pi*10
```

```
## [1] 31.41593
```

```
cos(pi)
```

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

---

Create variables with the **assignment operator**, `<-` or `=`, as in:

```
approx.pi = 22/7
approx.pi
```

```
## [1] 3.142857
```

```
diameter = 10
approx.pi * diameter
```

```
## [1] 31.42857
```

The assignment operator also changes values:

```
circumference = approx.pi * diameter
circumference
```

```
## [1] 31.42857
```

```
circumference = 30
circumference
```

```
## [1] 30
```

---

The code you write will be made of variables, with descriptive names: easier to design, easier to debug, easier to improve, and easier for others to read

Avoid "magic constants"; use named variables (you will be graded on this!)

Named variables are a first step towards **abstraction**

# The R workspace

What variables have you defined?

```
ls()
```

```
## [1] "approx.pi"     "circumference" "diameter"      "n"
## [5] "name"          "x"             "xval"          "y"
## [9] "z"
```

Getting rid of variables:

```
rm("circumference")
ls()
```

```
## [1] "approx.pi" "diameter"  "n"         "name"      "x"         "xval"
## [7] "y"         "z"
```

```
rm(list=ls()) # Be warned! This erases everything
ls()
```

```
## character(0)
```

## First data structure: vectors

A **data structure** is a grouping of relaed data values into an object

A **vector** is a sequence of values, all of the same type, as in:

```
x = c(7, 8, 10, 45)
x
```

```
## [1]  7  8 10 45
```

```
is.vector(x)
```

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

`c()` function returns a vector containing all its arguments in specified order

Hence `x[1]` would be the first element, `x[4]` the 4th element, and `x[-4]` a vector containing *all but* the fourth element

---

`vector(length=6)` returns an empty vector of length 6; helpful for filling things up later

```
weekly.hours = vector(length=5)
weekly.hours
```

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

```
weekly.hours[5] = 8
weekly.hours
```

```
## [1] 0 0 0 0 8
```

## Vector arithmetic

Arithmetic operator apply to vectors in a "componentwise" fashion:

```
y = c(-7, -8, -10, -45)
x+y
```

```
## [1] 0 0 0 0
```

```
x*y
```

```
## [1]   -49   -64  -100 -2025
```

## Recycling

**Recycling** repeat elements in shorter vector when combined with a longer one. Example:

```
x + c(-7,-8)
```

```
## [1]  0  0  3 37
```

```
x^c(1,0,-1,0.5)
```

```
## [1] 7.000000 1.000000 0.100000 6.708204
```

Single numbers are vectors of length 1 for purposes of recycling:

```
2*x
```

```
## [1] 14 16 20 90
```

---

Can also do componentwise comparisons with vectors:

```
x > 9
```

```
## [1] FALSE FALSE  TRUE  TRUE
```

(Note: this returns a Boolean vector)

Logical operators work elementwise:

```
(x > 9) & (x < 20)
```

```
## [1] FALSE FALSE  TRUE FALSE
```

---

To compare whole vectors, best to use `identical()` or `all.equal()`:

```
x == -y
```

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

```
identical(x,-y)
```

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

```
identical(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
```

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

```
all.equal(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
```

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

## Functions on vectors

Lots of functions can take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, and `sum()` return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram, a highly structured object, with the side effect of making a plot
- `ecdf()` similarly produces a cumulative-density-function object
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

## Indexing vectors

Vector of indices:

```
x[c(2,4)]
```

```
## [1]  8 45
```

Vector of negative indices:

```
x[c(-1,-3)]
```

```
## [1]  8 45
```

(Why that, and not `8 10`?)

---

Boolean vector:

```
x[x>9]
```

```
## [1] 10 45
```

```
y[x>9]
```

```
## [1] -10 -45
```

`which()` gives the elements of a Boolean vector that are `TRUE`:

```
places = which(x > 9)
places
```

```
## [1] 3 4
```

```
y[places]
```

```
## [1] -10 -45
```

## Named components

You can give names to elements or components of vectors, and index vectors accordingly

```
names(x) = c("v1","v2","v3","fred")
names(x)
```

```
## [1] "v1"   "v2"   "v3"   "fred"
```

```
x[c("fred","v1")]
```

```
## fred   v1
##   45    7
```

(Here R prints the labels, these are not actual components of `x`)

---

`names(x)` is just another vector (of characters):

```
names(y) = names(x)
sort(names(x))
```

```
## [1] "fred" "v1"   "v2"   "v3"
```

```
which(names(x)=="fred")
```

```
## [1] 4
```

## Summary

- We write programs by composing functions to manipulate data
- The basic data types let us represent Booleans, numbers, and characters
- Data structures let us group together related values
- Vectors let us group values of the same type
- Use variables rather a profusion of magic constants
- Name components of structures to make data more meaningful

## Peculiarites of floating-point numbers

The more bits in the fraction part, the more precision

The R floating-point data type is a **double** (what R calls **numeric**). Back when memory was expensive, the now-standard number of bits was twice the default

Finite precision $\Rightarrow$ arithmetic on doubles $\neq$ arithmetic on $\mathbb{R}$.

```
0.45 == 3*0.15
```

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

```
0.45 - 3*0.15
```

```
## [1] 5.551115e-17
```

---

Often ignorable, but not always

- Rounding errors tend to accumulate in long calculations
- When results should be $\approx 0$, errors can flip signs
- Usually better to use `all.equal()` than exact comparison

```
(0.5 - 0.3) == (0.3 - 0.1)
```

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

```
all.equal(0.5-0.3, 0.3-0.1)
```

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

## Peculiarities of integers

Typing a whole number in the terminal doesn't make it an integer; it makes it a double, whose fractional part is 0

```
is.integer(7)
```

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

```
as.integer(7)
```

```
## [1] 7
```

To test for being an integer in the mathematical sense, use `round()`:

```
round(7) == 7
```

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