# Computational Social Science

## Data structures

Dr. Thomas Davidson

Rutgers University

September 9, 2024

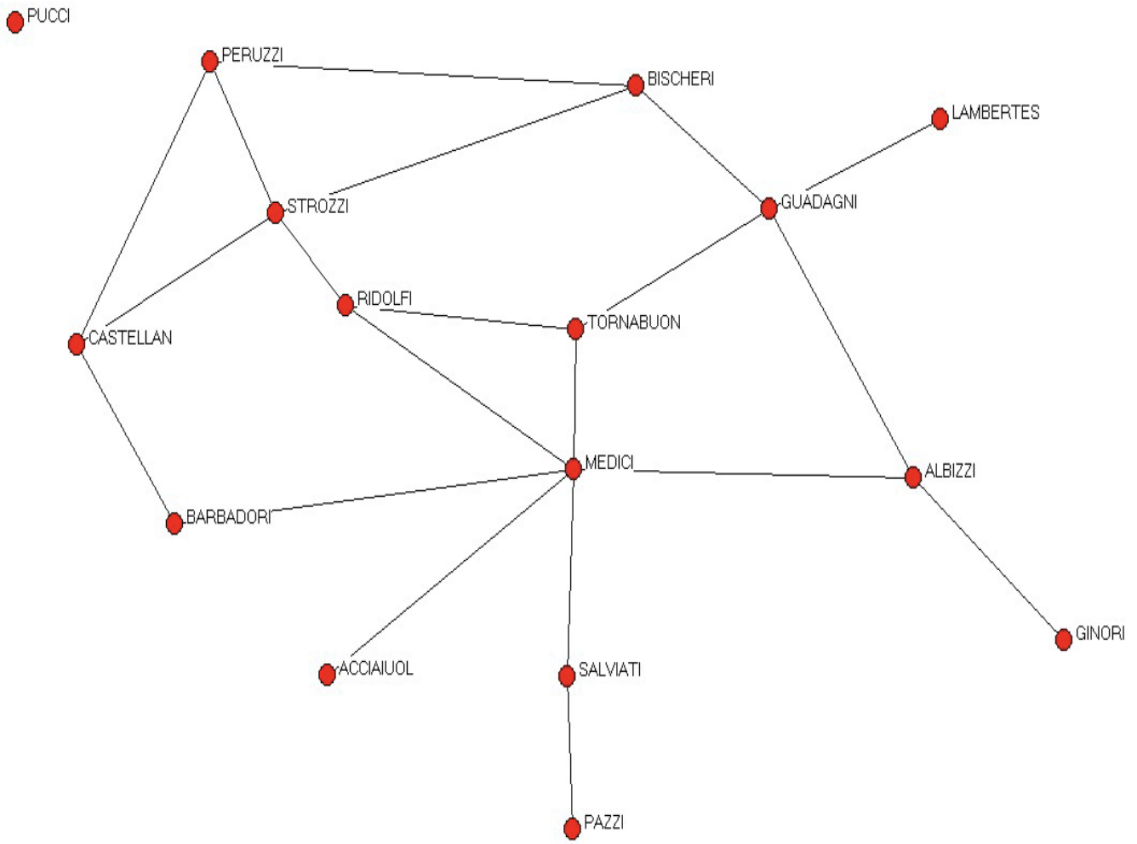
# Plan

- Introductions
- Part I : Social networks and social network analysis
  - What are social networks?
  - Visualizing social networks
- Part II : Data structures in R
  - Basic types
  - Vectors
  - Lists
  - Matrices
  - Data frames

# Introductions

- Name
- Area of study (major, minors, etc.) and year
- Interests in data science / social science

# Part I: Social networks and social network analysis



# Social networks

- A social network is a set of actors, or people, and the relationships between them
  - Family structures and kinship networks
  - Friendship and acquaintanceship networks
  - Organizations
  - Online social networks

# Social network analysis

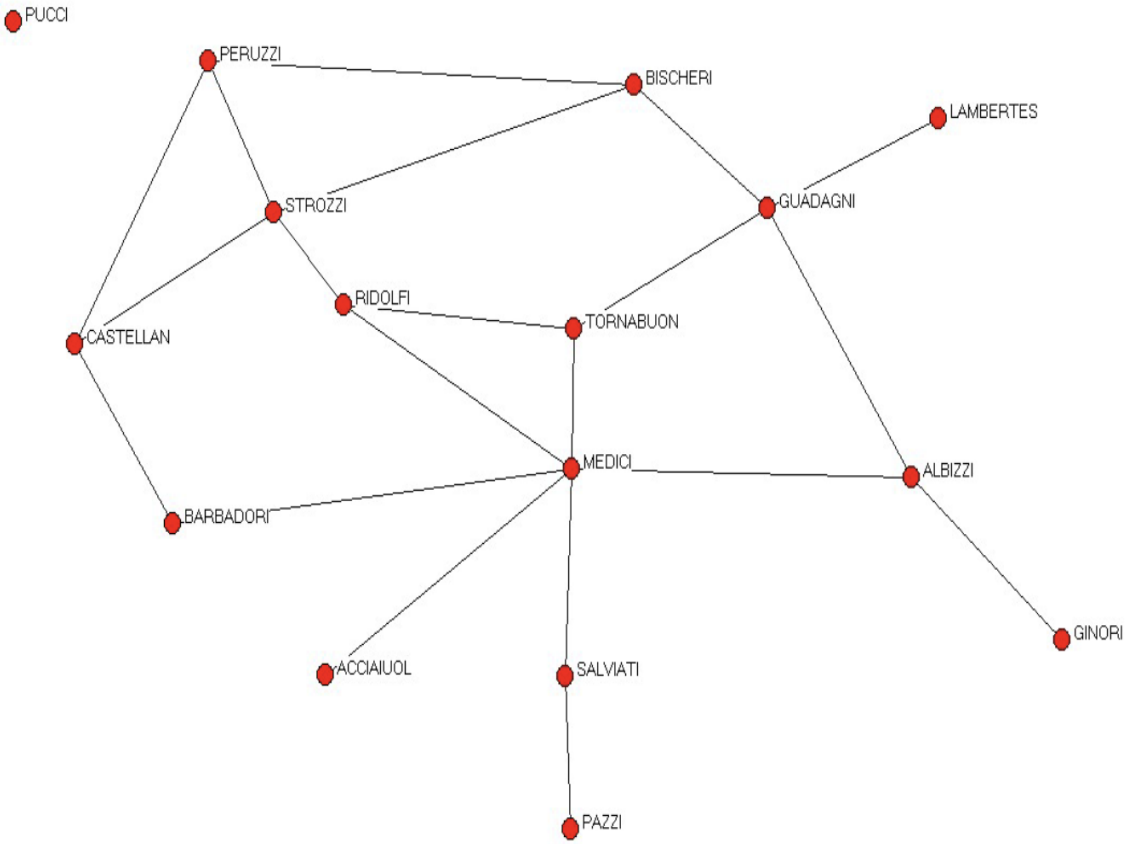
- Social network analysis is the study of social networks using mathematical and computational tools
- Networks can be represented using concepts from a branch of mathematics called graph theory
  - People or actors are represented as “nodes”
  - Relationships are represented as “edges”

# Social network analysis

- Social scientists use these representations to study the structure and dynamics of social networks
  - What does the network look like?
  - What are the patterns of connectivity?
  - How does information flow through the network?
  - How does the network change over time?

# Social network analysis

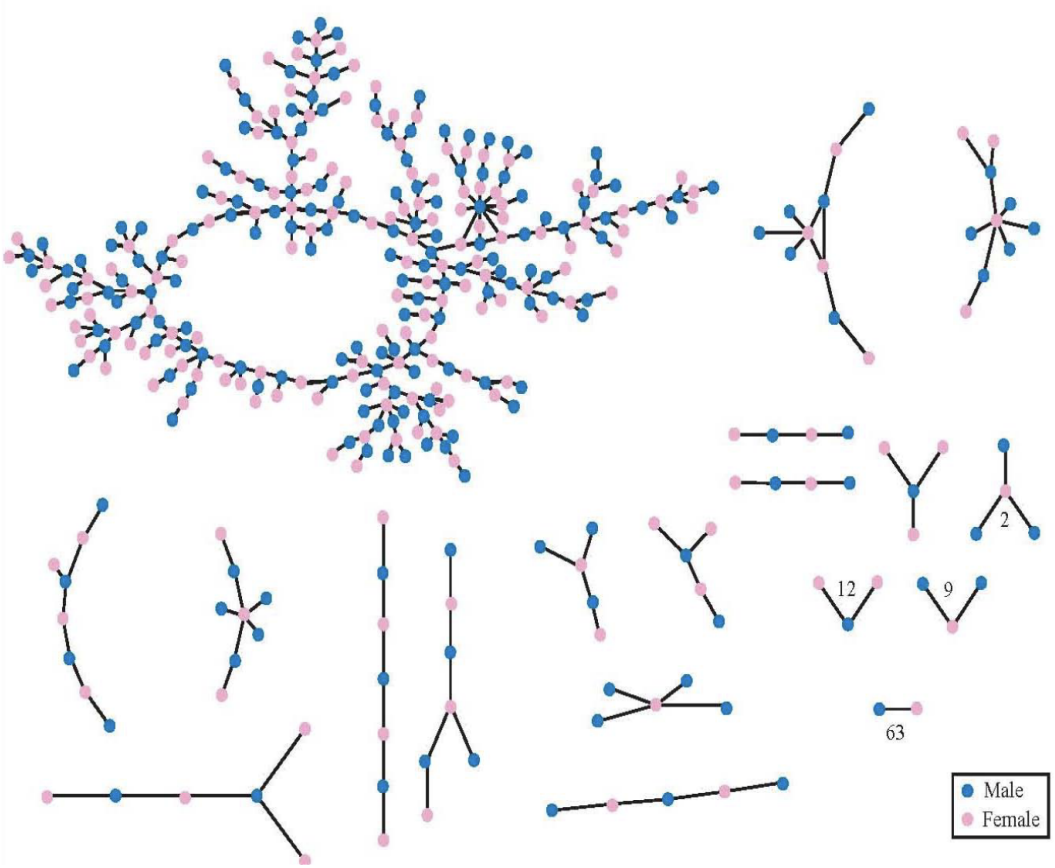
## Visualizing networks: Marriages




15th century Florentine marriage network (depicted in Jackson 2008)

# Social network analysis

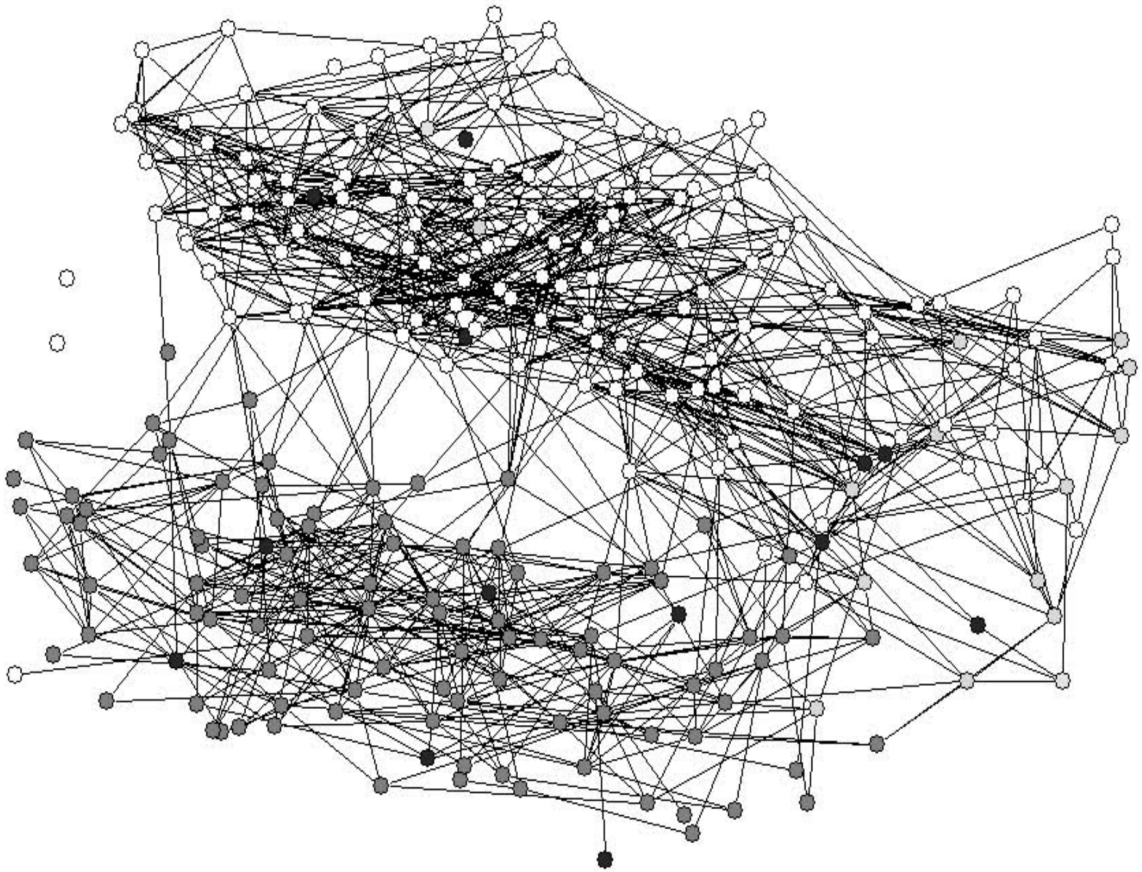
## Visualizing networks: Romantic encounters




Six months of relationships in an American high-school, nodes colored by sex (depicted in Jackson 2008)

# Social network analysis

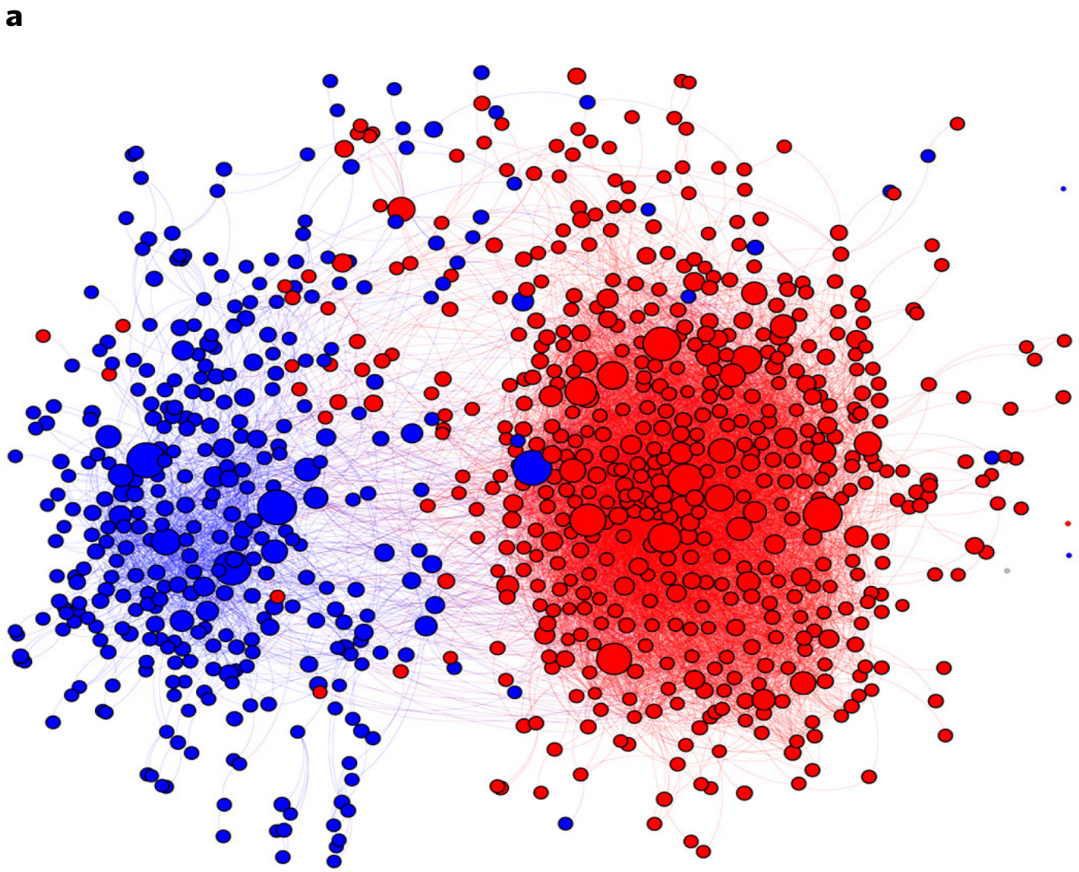
## Visualizing networks: Friendships



Six months of friendships in an American high-school, nodes colored by race/ethnicity (depicted in Jackson 2008)

# Social network analysis

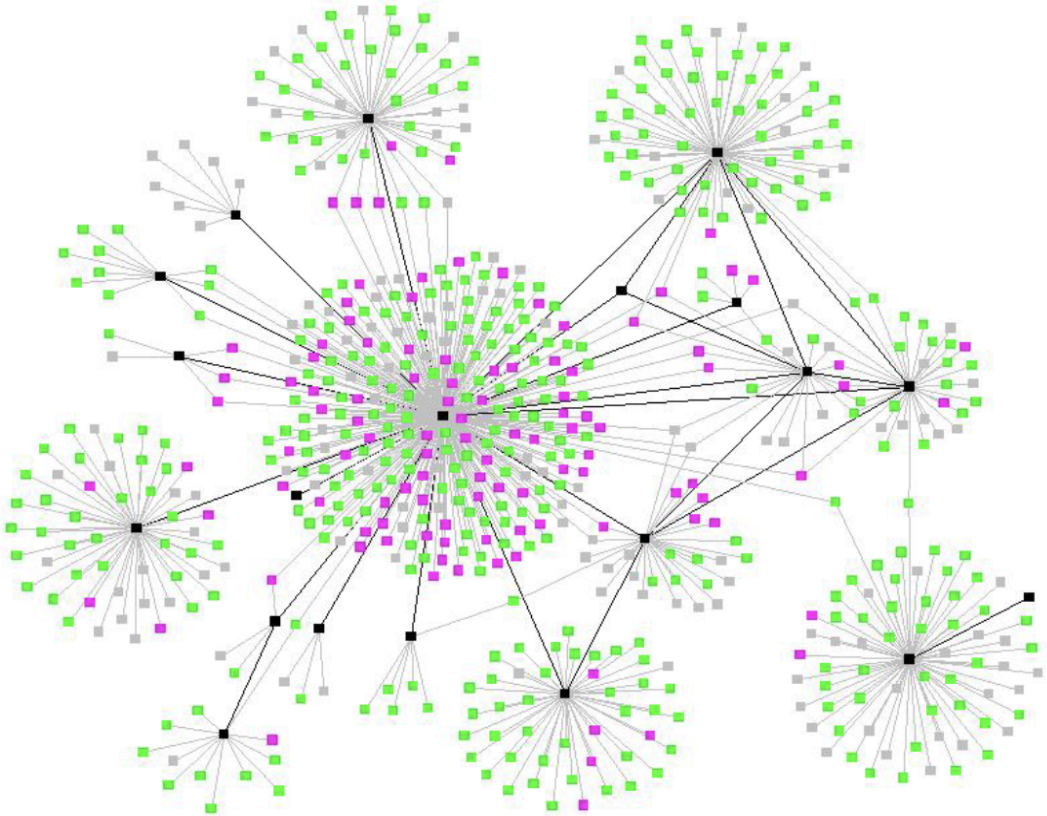
## Visualizing networks: Book co-purchases




Book co-purchases on Amazon, nodes colored by ideology (depicted in Jackson 2008)

# Social network analysis

## Visualizing networks: Disease transmission



Spread of tuberculosis in a social network, nodes colored by transmission status (depicted in Easley and Kleinberg 2010)

# Social network analysis

## Visualizing networks: Game of Thrones

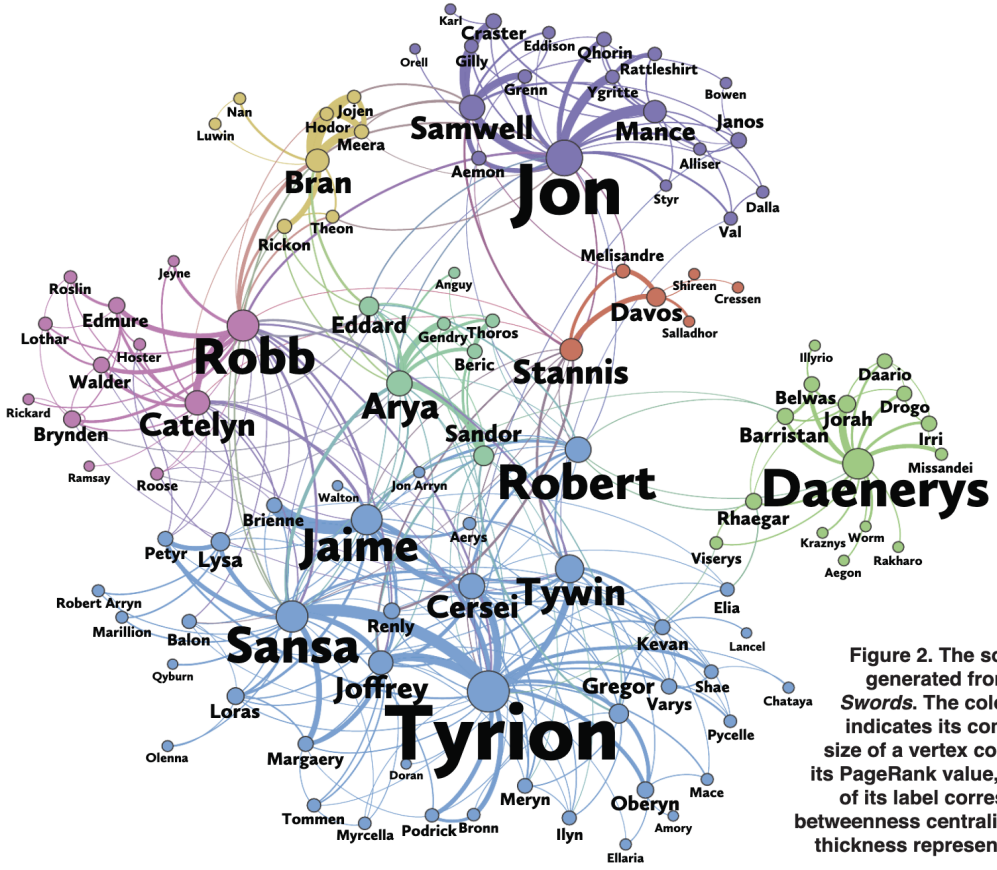



Figure 2. The social network generated from *A Storm of Swords*. The color of a vertex indicates its community. The size of a vertex corresponds to its PageRank value, and the size of its label corresponds to its betweenness centrality. An edge's thickness represents its weight.

Characters mentioned in *Game of Thrones* series, annotated with structural information (Beveridge and Shan 2010)

# Part II: Data structures

Open RStudio and load `lecture2-data-structures.Rmd`, located in this week's Canvas Module.

# Basic types

There are four basic types we will be using throughout the class. Here I used them to record some information about one of my cats. In R, it is convention to use the <- operator to assign an object to a name.

```
# Character (also known as "strings")
name <- "Gary"
# Numeric
weight <- 13.2
# Integer ("int" for short)
age <- 7L
# Logical
human <- FALSE
```

The other two are called `complex` and `raw`. See documentation

# Basic types

There are a few useful commands for inspecting objects.

```
print(name) # Prints value in console
```

```
## [1] "Gary"
```

```
class(name) # Shows class of object
```

```
## [1] "character"
```

```
typeof(name) # Shows type of object, not always equal to class
```

```
## [1] "character"
```

# Basic types

```
print(weight) # Prints value in console
```

```
## [1] 13.2
```

```
class(weight) # Shows class of object
```

```
## [1] "numeric"
```

```
typeof(weight) # Shows type of object, not always equal to class
```

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

# Basic types

We can use the == expression to verify the value of an object. We will discuss Boolean operations in more detail next lecture.

```
name == "Tabitha"
```

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

```
age == 4L
```

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

```
age >= 4L # is greater than
```

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

```
age != 4L # is not
```

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

# Vectors

A vector is a collection of elements of the *same* type. We can define an empty vector with `N` elements of a type. Empty vectors assume certain default values depending on the type.

```
N <- 5
x <- logical(N)
print(x)
```

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

```
y <- numeric(N)
print(y)
```

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

```
z <- character(N)
print(z)
```

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

# Vectors

Let's take a closer look at numeric vectors. We can use the combine function `c()` to concatenate multiple values into a vector.

```
v1 <- c(1,2,3,4,5)
v2 <- c(1,1,1,1,1)
class(v1) # check the class of v1
```

```
## [1] "numeric"
```

# Vectors

We can easily perform various mathematical operations on numeric vectors

```
v1 + v2 # addition
```

```
## [1] 2 3 4 5 6
```

```
v1 - v2 # subtraction
```

```
## [1] 0 1 2 3 4
```

```
v1 * v2 # multiplication
```

```
## [1] 1 2 3 4 5
```

```
sum(v1) # sum over v1
```

```
## [1] 15
```

Note how the different methods return different types of outputs. The arithmetic operations return vectors while `sum` returns a numeric value.

# Vectors

What happens if we try to combine objects of different types using combine?

```
t <- c("a", 1, TRUE)
typeof(t)
```

```
## [1] "character"
```

```
t
```

```
## [1] "a"    "1"    "TRUE"
```

# Vectors

There are lots of commands for generating special types of numeric vectors. Note how `N` has already been defined above.

```
seq(N) # generates a sequence from 1 to N
```

```
## [1] 1 2 3 4 5
```

```
rev(seq(N)) # reverses order
```

```
## [1] 5 4 3 2 1
```

```
rnorm(N) # samples N times from a normal distribution
```

```
## [1]  0.08222785  1.01981252 -0.48840314  0.26066555  0.20231876
```

# Vectors

We can use the help ? command to find information about each of these commands.

```
?seq
```

# Vectors

We can use the `index` to access the specific elements of a vector. R uses square brackets for such indexing.

```
x <- rnorm(N)
print(x)
```

```
## [1] -1.2150595 -1.7637090 -1.1335369  0.8002207  0.2375676
```

```
print(x[1]) # R indexing starts at 1; Python and some others start at 0
```

```
## [1] -1.21506
```

```
x[1] <- 9 # We can combine indexing with assignment to modify elements
print(x)
```

```
## [1]  9.0000000 -1.7637090 -1.1335369  0.8002207  0.2375676
```

# Vectors

The `head` and `tail` commands are useful when we're working with larger objects. Here we draw 10,000 observations from an normal distribution.

```
x <- rnorm(10000)
length(x)
```

```
## [1] 10000
```

```
head(x)
```

```
## [1] -0.4545523 -0.1078159  1.0306054 -0.8866033 -1.9644835 -0.648300
```

```
tail(x)
```

```
## [1] -0.8437151 -0.1629172 -1.0637072 -1.1088996 -0.8201811 -1.082379
```

# Excercise: Vectors

Retrieve the final element from `x` using indexing.

# Vectors

Vectors can also contain null elements to indicate missing values, represented by the NA symbol.

```
x <- c(1,2,3,4,NA)
is.na(x) # The is.na function indicates whether each value is missing.
```

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

```
!is.na(x) # Prepending ! denotes the inverse of a logical operation
```

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

NA is a logical type but can exist within numeric and character vectors. It is an exception to the rule discussed above regarding the presence of multiple types in the same vector.

# Lists

A list is an object that can contain different types of elements, including basic types and vectors.

```
print(v1)
```

```
## [1] 1 2 3 4 5
```

```
l1 <- list(v1) # We can easily convert the vector v1 into a list.
print(l1)
```

```
## [[1]]
## [1] 1 2 3 4 5
```

# Lists

Lists have a slightly different form of indexing. This can be one of the most confusing aspects of R for beginners!

```
l1[1] # The first element of the list contains the vector
```

```
## [[1]]
## [1] 1 2 3 4 5
```

```
l1[[1]] # Double brackets allows us to access the vector itself
```

```
## [1] 1 2 3 4 5
```

```
class(l1[1]) # first element is a list
```

```
## [1] "list"
```

```
class(l1[[1]]) # double indexing gives us the contents
```

```
## [1] "numeric"
```

# Lists

We can access specific elements of a list by using standard indexing.

```
l1[[1]][1] # Followed by single brackets to access a specific element
```

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

```
l1[1][1] # If we're not careful, we will just get the entire sublist
```

```
## [[1]]
## [1] 1 2 3 4 5
```

# Lists

We can easily combine multiple vectors into a list.

```
v.list <- list(v1,v2) # We could store both vectors in a list
print(v.list)
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] 1 1 1 1 1
```

```
v.list[[2]][4] # We can use double brackets to get element 4 of list 1
```

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

# Lists

We index sublists using double brackets, then specific elements with single brackets.

```
v.list[[1]][4] # We can use double brackets to get element 4 of list 1
```

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

# Lists

We can make indexing easier if we start with an empty list and then add elements using a named index via the $ operator.

```
v <- list() # initialize empty list
v$v1 <- v1 # the $ sign is used for named indexing
v$v2 <- v2
print(v)
```

```
## $v1
## [1] 1 2 3 4 5
##
## $v2
## [1] 1 1 1 1 1
```

# Excercise: Lists

Combine `$` and square bracket indexing to extract the 5th element of `v1` from the list `v`.

# Lists

We can define lists more concisely by providing sublists as named arguments.

```
cats <- list(names = c("Gary", "Tabitha"), ages = c(5,2))
print(cats)
```

```
## $names
## [1] "Gary"    "Tabitha"
##
## $ages
## [1] 5 2
```

# Matrices

A matrix is a two-dimensional data structure. Like vectors, matrices hold objects of a single type. Here we're defining a matrix using two arguments, the number of rows and columns.

```
matrix(nrow=5,ncol=5) # Here there is no content so the matrix is empty
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   NA   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA   NA
## [4,]   NA   NA   NA   NA   NA
## [5,]   NA   NA   NA   NA   NA
```

# Matrices

We can also pass an argument to define the initial contents of a matrix.

```
M <- matrix(0L, nrow=5, ncol=5) # 5x5 matrix of zeros
M
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    0
```

# Matrices

We can create a matrix by combining vectors using `cbind`.

```
M1 <- cbind(v1,v2) # Treat vectors a columns
print(M1)
```

```
##      v1 v2
## [1,]  1  1
## [2,]  2  1
## [3,]  3  1
## [4,]  4  1
## [5,]  5  1
```

# Matrices

If we want to treat the vectors as rows, we alternatively use `rbind`. We could also get the same result by *transposing* `M1`.

```
M2 <- rbind(v1, v2) # Vectors as rows
print(M2)
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2    1    1    1    1    1
```

```
print(t(M1)) # t() is the transpose function
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2    1    1    1    1    1
```

# Matrices

The `dim` function provides us with information about the dimensions of a given matrix. It returns the number of rows and columns.

```
dim(M1) # Shows the dimensions of the matrix
```

```
## [1] 5 2
```

```
dim(M2)
```

```
## [1] 2 5
```

# Matrices

We can get particular values using two-dimensional indexing. By convention `i` denotes the row and `j` the column.

```
i <- 1 # row index
j <- 2 # column index
M1[i,j] # Returns element i,j
```

```
## v2
##  1
```

```
M1[i,] # Returns row i
```

```
## v1 v2
##  1  1
```

```
M1[,j] # Returns column i
```

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

# Matrices

Like lists, we can also name rows and columns to help make indexing easier. The `colnames` and `rownames` functions show the names of each column and row.

```
colnames(M1)
```

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

```
rownames(M1)
```

```
## NULL
```

# Matrices

We can use these functions to assign new names.

```
colnames(M1) <- c("X", "Y")
rownames(M1) <- seq(1, nrow(M1))
print(M1)
```

```
##   X Y
## 1 1 1
## 2 2 1
## 3 3 1
## 4 4 1
## 5 5 1
```

# Putting it all together

## Storing names in vectors

Let's see how these types of objects can be used to create formal representations of social networks. We can start by defining a vector of names for a fictional friend group.

```
names <- c("Jay", "Amy", "Pete", "Mia")
```

# Putting it all together

## Representing relationships in a matrix

Next, we can define a matrix to store relationship ties. Each person is referenced according to their index in the name vector

```
X <- matrix(0, nrow = length(names), ncol = length(names))
X[1,2] <- 1 # edge from Jay to Amy.
X[1,3] <- 1
X[3,2] <- 1
X[1,4] <- 1
X[4,2] <- 1

print(X)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    1    1    1
## [2,]    0    0    0    0
## [3,]    0    1    0    0
## [4,]    0    1    0    0
```

# Putting it all together

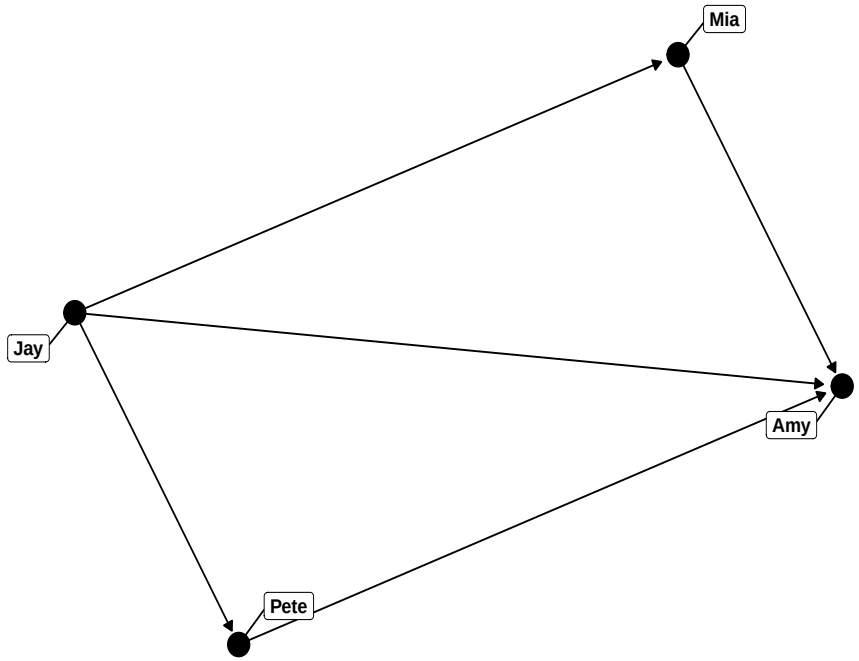
## Storing information in a list

```
network_info <- list(names = names, network = X)
print(network_info)
```

```
## $names
## [1] "Jay"  "Amy"  "Pete" "Mia"
##
## $network
##      [,1] [,2] [,3] [,4]
## [1,]    0    1    1    1
## [2,]    0    0    0    0
## [3,]    0    1    0    0
## [4,]    0    1    0    0
```

# Putting it all together

## Plotting the network



# Questions?