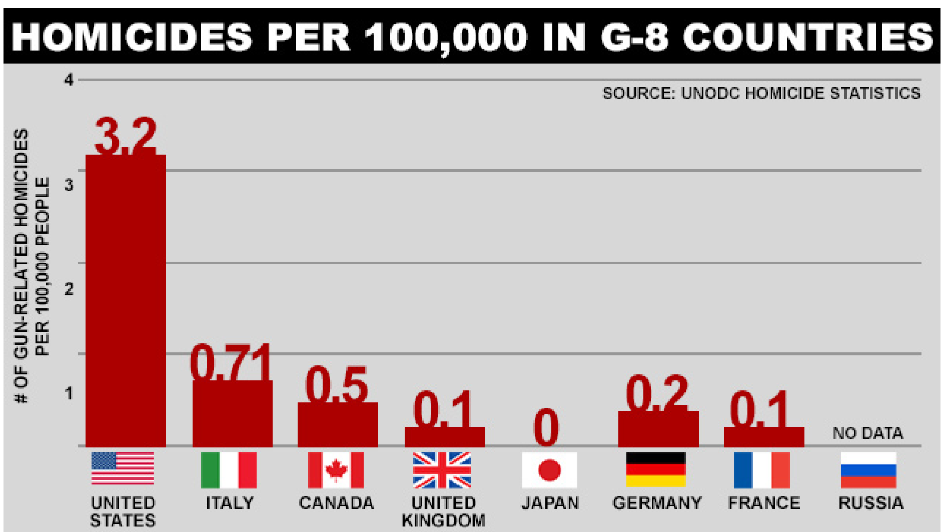
Introduction to R

In this course we will be using the R software environment for all our analysis. Throughout the course you will learn R and data analysis techniques simultaneously. However, we need to introduce basic R syntax to get you going. In this section, rather than cover every R skill you need, we introduce just enough so that you can follow along the remaining sections where we provide more in-depth coverage, building upon what you learn in this section. We find that we better retain R knowledge when we learn it to solve a specific problem.

In this section, as done throughout the course, we will use a motivating case study. We ask a specific question related to crime in the United States and provide a relevant dataset. Some basic R skills will permit us to answer the motivating question.

US gun murders

Imagine you live in Europe and are offered a job at a US company with many locations across all states. It is a great job but news with headlines such as **America is one of 6 countries that make up more than half of guns deaths worldwide** have you worried. Charts like this make you worry even more:



Or even worse, this version from everytown.org

But then you are reminded that the US is a large and diverse country with 50 very different states as well as the District of Columbia (DC).

California, for example, has a larger population than Canada and 20 US states have populations larger than that of Norway. In some respects the variability across states in the US is akin to the variability across countries in Europe. Furthermore, although not in the charts above, the murder rates in Lithuania, Ukraine, and Russia are higher than 4 per 100,000. So perhaps the news reports that worried you are too superficial. You have options of where to live and want to find out how safe each state is. We will gain some insights by examining data related to gun homicides in the US using R.

Now before we get started with our example, we need to cover logistics as well as some of the very basic building blocks that we need to gain more advanced R skills. Be aware that for some of these, it is not immediately obvious how it is useful, but later in the course you will appreciate having the knowledge under your belt.

Data types

Variables in R can be of different types. For example, we need to distinguish numbers from character strings and tables from simple lists of numbers. The function `class` helps us determine what type of object we have:

```
a <- 2
class(a)
```

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

To work efficiently in R it is important to learn the different types of variables and what we can do with these.

Data Frames

Up to now, the variables we have defined are just one number. This is not very useful for storing data. The most common way of storing a dataset in R is in a ***data frame***. Conceptually, we can think of a data frame as a table with rows representing observations and the different variables reported for each observation defining the columns. Data frames are particularly useful for datasets because we can combine different data types into one object.

We stored the data for our motivating example in a data frame. You can access this dataset by loading the `dslabs` library and loading the `murders` dataset using the `data` function:

```
library(dslabs)
data(murders)
```

To see that this is in fact a data frame we type

```
class(murders)
```

```
## [1] "data.frame"
```

Examining an object

The function `str` is useful to find out more about the structure of an object

```
str(murders)
```

```
## 'data.frame':    51 obs. of  5 variables:
##  $ state     : chr  "Alabama" "Alaska" "Arizona" "Arkansas" ...
##  $ abb       : chr  "AL" "AK" "AZ" "AR" ...
##  $ region    : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
##  $ population: num  4779736 710231 6392017 2915918 37253956 ...
##  $ total     : num  135 19 232 93 1257 ...
```

This tells us much more about the object. We see that the table has 51 rows (50 states plus DC) and five variables. We can show the first six lines using the function `head`:

```
head(murders)
```

```
##        state abb region population total
## 1    Alabama  AL  South    4779736   135
## 2     Alaska  AK   West     710231    19
## 3    Arizona  AZ   West    6392017   232
## 4   Arkansas  AR  South    2915918    93
## 5 California  CA   West   37253956  1257
## 6   Colorado  CO   West    5029196    65
```

In this dataset each state is considered an observation and five variables are reported for each state.

Before we go any further in answering our original question about different states, let's get to know the components of this object better.

The accessor

For our analysis we will need to access the different variables, represented by columns, included in this data frame. To access these variables we use the accessor operator `$` in the following way:

```
murders$population
```

```
##  [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097   897934
##  [9]   601723 19687653  9920000  1360301  1567582 12830632  6483802  3046355
## [17]  2853118  4339367  4533372  1328361  5773552  6547629  9883640  5303925
## [25]  2967297  5988927   989415  1826341  2700551  1316470  8791894  2059179
## [33] 19378102  9535483   672591 11536504  3751351  3831074 12702379  1052567
## [41]  4625364   814180  6346105 25145561  2763885   625741  8001024  6724540
## [49]  1852994  5686986   563626
```

But how did we know to use `population`? Above, by applying the function `str` to the object `murders`, we revealed the names for each of the five variables stored in this table. We can quickly access the variables names using:

```
names(murders)
```

```
## [1] "state"      "abb"        "region"     "population" "total"
```

It is important to know that the order of the entries in `murders$population` preserve the order of the rows in our data table. This will later permit us to manipulate one variable based on the results of another. For example, we will be able to order the state names by number of murders.

Tip: R comes with a very nice auto-complete functionality that saves us the trouble of typing out all the names. Try typing `murders$p` then hitting the *tab* key on your keyboard. RStudio has many useful auto-complete feature options.

Vectors: numerics, characters, and logical

Note that the object `murders$population` is not one number but several. We call these types of objects *vectors*. A single number is technically a vector but in general vectors refer to objects with several entries. The function `length` tells you how many entries are in the vector:

```
pop <- murders$population
length(pop)
```

```
## [1] 51
```

This particular vector is *numeric* since population sizes are numbers:

```
class(pop)
```

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

In a numeric vector, every entry must be a number.

To store character strings, vectors can also be of class *character*. For example, the state names are characters:

```
class(murders$state)
```

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

As with numeric vectors, all entries in a character vector need to be a character.

Another important type are *logical vectors*. These must be either TRUE or FALSE.

```
z <- 3 == 2
z
```

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

```
class(z)
```

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

Here the == is a relational operator asking if 3 is equal to 2. Remember that in R, you just use one = when you actually assign a value. You can see the other *relational operators* by typing

```
?Comparison
```

In future sections you will see how useful relational operators can be.

Factors

In the murders dataset we might expect the region to also be a character vector. However, it is not:

```
class(murders$region)
```

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

it is a *factor*. Factors are useful for storing categorical data. Notice that there are only 4 regions:

```
levels(murders$region)
```

```
## [1] "Northeast"     "South"         "North Central" "West"
```

So, in the background, R stores these *levels* as integers and keeps a map to keep track of the labels. This is more memory efficient than storing all the characters. However, factors are also a source of confusion as they can easily be confused with characters but behave differently in different contexts. We will see more of this later.

In general, we recommend avoiding factors as much as possible although they are sometimes necessary to fit models containing categorical data.

Lists

Data frames are a special case of *lists*. We will covers lists in more detail later but know that they are useful because you can store any combination of other types. Here is an example of a list we created for you:

```
record <- list(name = "John Doe",
               student_id = 1234,
               grades = c(95, 82, 91, 97, 93),
               final_grade = "A")
```

```
record
```

```
## $name
## [1] "John Doe"
##
## $student_id
## [1] 1234
##
## $grades
## [1] 95 82 91 97 93
##
## $final_grade
```

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

```
class(record)
```

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

We won't be using lists until later but you might encounter one in your own exploration of R. Note that, as with data frames, you can extract the components with the accessor `$`. In fact, data frames are a type of list.

```
record$student_id
```

```
## [1] 1234
```

We can also use double brackets like this:

```
record[["student_id"]]
```

```
## [1] 1234
```

You should get used to the fact that in R there are several ways to do the same thing, in particular accessing entries.

Vectors

The most basic unit available in R to store data are *vectors*. Complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector. Here we learn more about this important class.

Creating vectors

We can create vectors using the function `c`, which stands for concatenate. We use `c` to *concatenate* entires in the following way:

```
codes <- c(380, 124, 818)
codes
```

```
## [1] 380 124 818
```

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variables names.

```
country <- c("italy", "canada", "egypt")
```

Note that if you type

```
country <- c(italy, canada, egypt)
```

you receive an error because the variables `italy`, `canada` and `egypt` are not defined: R looks for variables with those names and returns an error.

Names

Sometimes it is useful to name the entries of a vector. For example, when defining a vector of country codes we can use the names to connect the two:

```
codes <- c(italy = 380, canada = 124, egypt = 818)
codes
```

```
##  italy canada  egypt
##    380    124    818
```

The object `codes` continues to be a numeric vector:

```
class(codes)
```

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

but with names

```
names(codes)
```

```
## [1] "italy"  "canada" "egypt"
```

If the use of strings without quotes looks confusing, know that you can use the quotes as well

```
codes <- c("italy" = 380, "canada" = 124, "egypt" = 818)
codes
```

```
##  italy canada  egypt
##    380    124    818
```

There is no difference between this call and the previous one: one of the many ways R is quirky compared to other languages.

We can also assign names using the `names` function:

```
codes <- c(380, 124, 818)
country <- c("italy", "canada", "egypt")
names(codes) <- country
codes
```

```
##  italy canada  egypt
##    380    124    818
```

Sequences

Another useful function for creating vectors generates sequences

```
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

The first argument defines the start, and the second the end. The default is to go up in increments of 1, but a third argument let's us tell it how much to jump by:

```
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

If we want consecutive integers we can use the following shorthand

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Note that when we use this function, R produces integers, not numerics, because they are typically used to index something:

```
class(1:10)
```

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

However, note that as soon as we create something that's not an integer the class changes:

```
class(seq(1, 10))
```

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

```
class(seq(1, 10, 0.5))
```

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

Subsetting

We use square brackets to access specific elements of a list. For the vector `codes` we defined above, we can access the second element using

```
codes[2]
```

```
## canada
##    124
```

You can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
```

```
## italy egypt
##   380   818
```

The sequences defined above are particularly useful if we want to access, say, the first two elements

```
codes[1:2]
```

```
##  italy canada
##    380    124
```

If the elements have names, we can also access the entries using these names. Here are two examples.

```
codes["canada"]
```

```
## canada
##    124
```

```
codes[c("egypt", "italy")]
```

```
## egypt italy
##   818   380
```

Coercion

In general, *coercion* is an attempt by R to be flexible with data types. When an entry does not match the expected, R tries to guess what we meant before throwing an error. This can also lead to confusion. Failing to understand *coercion* can drive a programmer crazy when attempting to code in R since it behaves quite differently from most other languages in this regard. Let's learn about it with some examples.

We said that elements of a vector must be all of the same type. So if we try to combine, say, numbers and characters you might expect an error

```
x <- c(1, "canada", 3)
```

But we don't get one, not even a warning! What happened? Look at `x` and its class:

```
x
```

```
## [1] "1"      "canada" "3"
```

```
class(x)
```

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

R *coerced* the data into characters. It guessed that because you put a character string in the vector you meant the 1 and 3 to actually be character strings `"1"` and `"3"`. The fact that not even a warning is issued is an example of how coercion can cause many unnoticed errors in R.

R also offers functions to force a specific coercion. For example you can turn numbers into characters with

```
x <- 1:5
y <- as.character(x)
y
```

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

And you can turn it back with `as.numeric`.

```
as.numeric(y)
```

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

This function is actually quite useful as datasets that include numbers as character strings are common.

Not Availables (NA)

When these coercion functions encounter an impossible case it gives us a warning and turns the entry into a special value called an `NA` for "not available". For example:

```
x <- c("1", "b", "3")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1 NA  3
```

R does not have any guesses for what number you want when you type `b` so it does not try.

Note that as a data scientist you will encounter `NA`s often as they are used for missing data, a common problem in real-life datasets.

Sorting

Now that we have some basic R knowledge under our belt, let's try to gain some insights into the safety of different states in the context of gun murders.

`sort`

We want to rank the states from least to most gun murders. The function `sort` sorts a vector in increasing order. So we can see the number of gun murders by typing

```
library(dslabs)
data(murders)
sort(murders$total)
```

```
##  [1]    2    4    5    5    7    8   11   12   12   16   19   21   22   27   32
## [16]   36   38   53   63   65   67   84   93   93   97   97   99  111  116  118
## [31]  120  135  142  207  219  232  246  250  286  293  310  321  351  364  376
## [46]  413  457  517  669  805 1257
```

However, this does not give us information about which states have which murder totals. For example, we don't know which state had 1257 murders in 2010.

order

The function `order` is closer to what we want. It takes a vector and returns the vector of indexes that sort the input vector. This may sound confusing so let's look at a simple example: we create a vector and sort it:

```
x <- c(31, 4, 15, 92, 65)
sort(x)
```

```
## [1]  4 15 31 65 92
```

Rather than sort the vector, the function `order` gives us back the index that, if used to index the vector, will sort it:

```
index <- order(x)
x[index]
```

```
## [1]  4 15 31 65 92
```

If we look at this index we see why it works:

```
x
```

```
## [1] 31  4 15 92 65
```

```
order(x)
```

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

Note that the second entry of `x` is the smallest so `order(x)` starts with `2`. The next smallest is the third entry so the second entry is `3` and so on.

How does this help us order the states by murders? First remember that the entries of vectors you access with `$` follow the same order as the rows in the table. So, for example, these two vectors, containing the state names and abbreviations respectively, are matched by their order:

```
murders$state[1:10]
```

```
##  [1] "Alabama"              "Alaska"               "Arizona"
##  [4] "Arkansas"             "California"           "Colorado"
##  [7] "Connecticut"          "Delaware"             "District of Columbia"
## [10] "Florida"
```

```
murders$abb[1:10]
```

```
##  [1] "AL" "AK" "AZ" "AR" "CA" "CO" "CT" "DE" "DC" "FL"
```

So this means we can now order the state names by their total murders by first obtaining the index that orders the vectors according to murder totals, and then indexing the state names or abbreviation vector:

```
ind <- order(murders$total)
murders$abb[ind]
```

```
##  [1] "VT" "ND" "NH" "WY" "HI" "SD" "ME" "ID" "MT" "RI" "AK" "IA" "UT" "WV" "NE"
## [16] "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI" "DC" "OK" "KY" "MA"
## [31] "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC" "MD" "OH" "MO" "LA" "IL" "GA"
## [46] "MI" "PA" "NY" "FL" "TX" "CA"
```

We see that California had the most murders.

max and which.max

If we are only interested in the entry with the largest value we can use `max` for the value

```
max(murders$total)
```

```
## [1] 1257
```

and `which.max` for the index of the largest value

```
i_max <- which.max(murders$total)
murders$state[i_max]
```

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

For the minimum we can use `min` and `which.min` in the same way.

So is California the most dangerous state? In a next section we argue that we should be considering rates not totals. Before doing that we introduce one last order related function: `rank`

`rank`

Although less useful than `order` and `sort`, the function `rank` is also related to order. For any given list it gives you a vector with the rank of the first entry, second entry, etc... of the vector. Here is a simple example.

```
x <- c(31, 4, 15, 92, 65)
rank(x)
```

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

To summarize let's look at the results of the three functions we have introduced

original	sort	order	rank
31	4	2	3
4	15	3	1
15	31	1	2
92	65	5	5
65	92	4	4

Vector arithmetic

```
library(dslabs)
data(murders)
```

California had the most murders. But does this mean it is the most dangerous state? What if it just has many more people than any other state? We can very quickly confirm that, indeed, California has the largest population:

```
murders$state[which.max(murders$population)]
```

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

with over 37 million inhabitants! It is therefore unfair to compare the totals if we are interested in learning how safe the state is.

What we really should be computing is the murders per capita. The reports we describe in the motivating section used murders per 100,000 as the unit. To compute this quantity, the powerful vector arithmetic capabilities of R come in handy.

Rescaling

In R, arithmetic operations on vectors occur *element wise*. For a quick example suppose we have height in inches

```
heights <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)
```

and want to covert to centimeters. Note what happens when we multiply `heights` by 2.54:

```
heights * 2.54
```

```
##  [1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.42 170.18 177.80
```

it multiplied each element by 2.54. Similarly if we want to compute how many inches taller or shorter than the average, 69 inches, we can subtract it from every entry like this

```
heights - 69
```

```
##  [1]  0 -7 -3  1  1  4 -2  4 -2  1
```

Two vectors

If we have two vectors of the same length, and we sum them in R, they get added entry by entry like this

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} + \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} a+e \\ b+f \\ c+g \\ d+h \end{pmatrix}$$

The same holds for other mathematical operations such as `-`, `*` and `/`.

This implies that to compute the murder rates we can simply type

```
murder_rate <- murders$total / murders$population * 100000
```

Once we do this, we notice that California is no longer near the top of the list. In fact, we can use what we have learned to order the states by murder rate:

```
murders$state[order(murder_rate)]
```

```
##  [1] "Vermont"              "New Hampshire"        "Hawaii"
##  [4] "North Dakota"         "Iowa"                 "Idaho"
##  [7] "Utah"                 "Maine"                "Wyoming"
## [10] "Oregon"               "South Dakota"         "Minnesota"
## [13] "Montana"              "Colorado"             "Washington"
## [16] "West Virginia"        "Rhode Island"         "Wisconsin"
## [19] "Nebraska"             "Massachusetts"        "Indiana"
## [22] "Kansas"               "New York"             "Kentucky"
## [25] "Alaska"               "Ohio"                 "Connecticut"
## [28] "New Jersey"           "Alabama"              "Illinois"
## [31] "Oklahoma"             "North Carolina"       "Nevada"
## [34] "Virginia"             "Arkansas"             "Texas"
## [37] "New Mexico"           "California"           "Florida"
## [40] "Tennessee"            "Pennsylvania"         "Arizona"
## [43] "Georgia"              "Mississippi"          "Michigan"
## [46] "Delaware"             "South Carolina"       "Maryland"
## [49] "Missouri"             "Louisiana"            "District of Columbia"
```

Note that the states are listed in *ascending* order of murder rate. Thus, DC has the highest murder rate.

Indexing

```
library(dslabs)
data(murders)
```

R provides a powerful and convenient way of indexing vectors. We can, for example, subset a vector based on properties of another vector. We continue our US murders example to demonstrate.

Subsetting with logicals

We can calculate the murder rate using

```
murder_rate <- murders$total / murders$population * 100000
```

Say you are moving from Italy where, according to an ABC news report, the murder rate is only 0.71 per 100,000. You would prefer to move to a state with a similar rate. Another powerful feature of R is that we can we can use logicals to index vectors. Note that if we compare a vector to a single number, it actually performs the test for each entry. Here is an example related to the question above.

```
ind <- murder_rate < 0.71
ind
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [13] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [49] FALSE FALSE FALSE
```

Or if we want to know if its less than or equal to we can use

```
ind <- murder_rate <= 0.71
ind
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [13] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [49] FALSE FALSE FALSE
```

Note that we get back a logical vector with `TRUE` for each entry smaller than or equal to 0.71. To see which states these are, we can leverage the fact that vectors can be indexed with logicals.

```
murders$state[ind]
```

```
## [1] "Hawaii"        "Iowa"          "New Hampshire" "North Dakota"
## [5] "Vermont"
```

Note that to count how many are TRUE, the function `sum` returns the sum of the entries of a vector and logical vectors get *coerced* to numeric with `TRUE` coded as 1 and `FALSE` as 0. Thus we can count the states using

```
sum(ind)
```

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

Logical Operators

Suppose we like the mountains and we want to move to a safe state in the West region of the country. We want the murder rate to be at most 1. So we want two different things to be true. Here we can use the logical operator *and* which in R is `&`. This operation results in a true only when both logicals are true. To see this consider these examples:

```
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

```
FALSE & FALSE
```

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

We can form two logicals:

```
west <- murders$region == "West"
safe <- murder_rate <= 1
```

and we can use the `&` to get a vector of logicals that tells us which states satisfy both of our conditions:

```
ind <- safe & west
murders$state[ind]
```

```
## [1] "Hawaii"  "Idaho"   "Oregon"  "Utah"    "Wyoming"
```

`which`

Suppose we want to look up California's murder rate. For this type of operation, it is convenient to convert vectors of logicals into indexes instead of keeping long vectors of logicals. The function `which` tells us which entries of a logical vector are TRUE. So we can type:

```
ind <- which(murders$state == "California")
ind # this is the index that matches the California entry
```

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

```
murder_rate[ind]
```

```
## [1] 3.374138
```

`match`

If instead of just one state we want to find out the murder rates for several, say New York, Florida, and Texas, we can use the function `match`. This function tells us which indexes of a second vector match each of the entries of a first vector:

```
ind <- match(c("New York", "Florida", "Texas"), murders$state)
ind
```

```
## [1] 33 10 44
```

Now we can look at the murder rates:

```
murder_rate[ind]
```

```
## [1] 2.667960 3.398069 3.201360
```

`%in%`

If rather than an index we want a logical that tells us whether or not each element of a first vector is in a second, we can use the function `%in%`. So, say you are not sure if Boston, Dakota and Washington are states, you can find out like this

```
c("Boston", "Dakota", "Washington") %in% murders$state
```

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

Assessments

```
library(dslabs)
data(murders)
```

1. Compute the per 100,000 murder rate for each state and store it in an object called `murder_rate`. Then use the logical operators to create a logical vector, name it `low`, that tells us which entries of `murder_rate` are lower than 1.

```
murder_rate <- murders$total / murders$population * 100000
low <- murder_rate < 1
low
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [13]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
## [25] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
## [37] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
## [49] FALSE FALSE  TRUE
```

2. Now use the results from the previous exercise and the function `which` to determine the indices of `murder_rate` associated with values lower than 1.

```
ind <- which(murder_rate < 1)
```

3. Use the results from the previous exercise to report the names of the states with murder rates lower than 1.

```
murders$state[ind]
```

```
##  [1] "Hawaii"        "Idaho"         "Iowa"          "Maine"
##  [5] "Minnesota"     "New Hampshire" "North Dakota"  "Oregon"
##  [9] "South Dakota"  "Utah"          "Vermont"       "Wyoming"
```

4. Now extend the code from exercises 2 and 3 to report the states in the Northeast with murder rates lower than 1. Hint: Use the previously defined logical vector `low` and the logical operator `&`.

```
northeast <- murders$region == "Northeast"
ind <- low & northeast
murders$state[ind]
```

```
## [1] "Maine"         "New Hampshire" "Vermont"
```

5. In a previous exercise we computed the murder rate for each state and the average of these numbers. How many states are below the average?

```
avg <- mean(murder_rate)
sum(murder_rate < avg)
```

```
## [1] 27
```

6. Use the match function to identify the states with abbreviations AK, MI, and IA. Hint: Start by defining an index of the entries of `murders$abb` that match the three abbreviations, then use the `[]` operator to extract the states.

```r
ind <- match(c("AK", "MI", "IA"), murders$abb)
murders$state[ind]
```

```
## [1] "Alaska"   "Michigan" "Iowa"
```

7. Use the `%in%` operator to create a logical vector that answers the question: which of the following are actual abbreviations: MA, ME, MI, MO, MU?

```r
c("MA", "ME", "MI", "MO", "MU") %in% murders$abb
```

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

8. Extend the code you used in exercise seven to report the one entry that is **not** an actual abbreviation. Hint: Use the `!` operator, which stands for "not" and turns `FALSE` into `TRUE` and vice-versa, then `which` to obtain an index.

```r
which(c("MA", "ME", "MI", "MO", "MU") %in% murders$abb != TRUE)
```

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

Basic Data Wrangling

```r
library(dslabs)
data(murders)
```

Up to now we have been changing vectors by reordering them and subsetting them through indexing. But once we start more advanced analyses, we will want to prepare data tables for data analysis. We refer to this task as **data wrangling**. For this purpose we will introduce the `dplyr` package which provides intuitive functionality for working with tables.

Once you install `dplyr` you can load it using

```r
library(dplyr)
```

This package introduces functions that perform the most common operations in data wrangling and uses names for these functions that are relatively easy to remember. For example, to change the data table by adding a new column, we use `mutate`. To filter the data table to a subset of rows we use `filter` and to subset the data by selecting specific columns we use `select`. We can also perform a series of operations. For example, select and then filter, by sending the results of one function to another using what is called the *pipe operator*: `%>%`. Some details are included below.

Adding a column with `mutate`

We want all the necessary information for our analysis to be included in the data table. So the first task is to add the murder rate to our data frame. The function mutate takes the data frame as a first argument and the name and values of the variable in the second using the convention `name = values`. So to add murder rate we use:

```r
murders <- mutate(murders, murder_rate = total / population * 100000)
```

Note that here we used `total` and `population` in the function, which are objects that are **not** defined in our workspace. What is happening is that `mutate` knows to look for these variables in the `murders` data frame because the first argument we put was the `murders` data frame. So the intuitive line of code above does exactly what we want. We can see the new column is added:

```r
head(murders)
```

```
##        state abb region population total murder_rate
```

```
## 1    Alabama  AL  South    4779736   135    2.824424
## 2     Alaska  AK   West     710231    19    2.675186
## 3    Arizona  AZ   West    6392017   232    3.629527
## 4   Arkansas  AR  South    2915918    93    3.189390
## 5 California  CA   West   37253956  1257    3.374138
## 6   Colorado  CO   West    5029196    65    1.292453
```

Also note that we have over-written the original `murders` object. However, this does *not* change the object that is saved and loaded with `data(murders)`. If we load the `murders` data again, the original will over-write our mutated version.

Note: If we reload the dataset from the `dslabs` package it will rewrite our new data frame with the original.

Subsetting with `filter`

Now suppose that we want to filter the data table to only show the entries for which the murder rate is lower than 0.71. To do this we use the `filter` function which takes the data table as an argument and then the conditional statement as the next argument. Like mutate, we can use the data table variable names inside the function and it will know we mean the columns and not objects in the workspace.

```
filter(murders, murder_rate <= 0.71)
```

```
##           state abb        region population total murder_rate
## 1        Hawaii  HI          West    1360301     7   0.5145920
## 2          Iowa  IA North Central    3046355    21   0.6893484
## 3 New Hampshire  NH     Northeast    1316470     5   0.3798036
## 4  North Dakota  ND North Central     672591     4   0.5947151
## 5       Vermont  VT     Northeast     625741     2   0.3196211
```

Selecting columns with `select`

Although our data table only has six columns, some data tables include hundreds. If we want to view just a few, we can use the `select` function. In the code below we select three columns, assign this to a new object and then filter the new object:

```
new_table <- select(murders, state, region, murder_rate)
filter(new_table, murder_rate <= 0.71)
```

```
##           state        region murder_rate
## 1        Hawaii          West   0.5145920
## 2          Iowa North Central   0.6893484
## 3 New Hampshire     Northeast   0.3798036
## 4  North Dakota North Central   0.5947151
## 5       Vermont     Northeast   0.3196211
```

Note that in the call to `select`, the first argument, `murders`, is an object but `state`, `region`, and `murder_rate` are variable names.

The pipe: `%>%`

In the code above we wanted to show the three variables for states that have murder rates below 0.71. To do this we defined an intermediate object. In `dplyr` we can write code that looks more like our description of what we want to:

$$\text{original data} \rightarrow \text{select} \rightarrow \text{filter}$$

For such an operation, we can use the pipe `%>%`. The code looks like this:

```
murders %>% select(state, region, murder_rate) %>% filter(murder_rate <= 0.71)
```

```
##           state        region murder_rate
## 1        Hawaii          West   0.5145920
## 2          Iowa North Central   0.6893484
## 3 New Hampshire     Northeast   0.3798036
## 4  North Dakota North Central   0.5947151
## 5       Vermont     Northeast   0.3196211
```

This line of code is equivalent to the two lines of code above. Note that when using the pipe we no longer need to specify the murders data frame since the `dplyr` functions assume that whatever is being *piped* is what should be operated on.

Summarizing data with `dplyr`

An important part of exploratory data analysis is summarizing data. It is sometimes useful to split data into groups before summarizing.

Summarize

The `summarize` function in `dplyr` provides a way to compute summary statistics with intuitive and readable code. We can compute the average of the murder rates like this.

```
murders %>% summarize(avg = mean(murder_rate))
```

```
##        avg
## 1 2.779125
```

However, note that the US murder rate is **not** the average of the state murder rates. Because in this computation the small states are given the same weight as the large ones. The US murder rate is proportional to the total US murders divided by the total US population.

To compute the country's average murder rate using the `summarize` function, we can do the following:

```
us_murder_rate <- murders %>%
  summarize(murder_rate = sum(total) / sum(population) * 100000)

us_murder_rate
```

```
##   murder_rate
## 1    3.034555
```

This computation counts larger states proportionally to their size and this results in a larger value.

Using the dot to access the piped data

The `us_murder_rate` object defined above represents just one number. Yet we are storing it in a data frame

```
class(us_murder_rate)
```

```
## [1] "data.frame"
```

since, as with most `dplyr` functions, `summarize` *always returns a data frame*.

This might be problematic if we want to use the result with functions that require a numeric value. Here we show a useful trick to access values stored in data piped via `%>%`: when a data object is piped it can be accessed using the dot `.`. To understand what we mean take a look at this line of code:

```
us_murder_rate %>% .$murder_rate
```

```
## [1] 3.034555
```

Note that this returns the value in the `murder_rate` column of `us_murder_rate` making it equivalent to `us_murder_rate$murder_rate`. To understand this line, you just need to think of `.` as a placeholder for the data that is being passed through the pipe. Because this data object is a data frame, we can access it's columns with the `$`.

To get a number from the original data table with one line of code we can type:

```
us_murder_rate <- murders %>%
  summarize( murder_rate = sum(total) / sum(population) * 100000) %>%
  .$murder_rate

us_murder_rate
```

```
## [1] 3.034555
```

which is now a numeric:

```
class(us_murder_rate)
```

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

We will see other instances in which using the `.` is useful. For now, we will only use it to produce numeric vectors from pipelines constructed with `dplyr`.

Group then summarize

A common operation in data exploration is to first split data into groups and then compute summaries for each group. For example, we may want to compute the median murder rate for each region. The `group_by` function helps us do this.

If we type this:

```
murders %>%
  group_by(region) %>%
  summarize(median_rate = median(murder_rate))
```

```
## # A tibble: 4 x 2
##   region        median_rate
##   <fct>               <dbl>
## 1 Northeast            1.80
## 2 South                3.40
## 3 North Central        1.97
## 4 West                 1.29
```

we get a table with the median murde rate for each of the four regions.

Sorting data tables

When examining a dataset it is often convenient to sort the table by the different columns. We know about the `order` and `sort` functions, but for ordering entire tables, the `dplyr` function `arrange` is useful. For example, here we order the states by population size:

```
murders %>%
  arrange(population) %>%
  head()
```

```
##                  state abb        region population total murder_rate
## 1              Wyoming  WY          West     563626     5   0.8871131
## 2 District of Columbia  DC         South     601723    99  16.4527532
## 3              Vermont  VT     Northeast     625741     2   0.3196211
```

```
## 4         North Dakota  ND North Central     672591     4   0.5947151
## 5               Alaska  AK          West     710231    19   2.6751860
## 6         South Dakota  SD North Central     814180     8   0.9825837
```

Note that we get to decide which column to sort by. To see the states by murder rate, from smallest to largest, we arrange by `murder_rate` instead:

```
murders %>%
  arrange(murder_rate) %>%
  head()
```

```
##           state abb        region population total murder_rate
## 1       Vermont  VT     Northeast     625741     2   0.3196211
## 2 New Hampshire  NH     Northeast    1316470     5   0.3798036
## 3        Hawaii  HI          West    1360301     7   0.5145920
## 4  North Dakota  ND North Central     672591     4   0.5947151
## 5          Iowa  IA North Central    3046355    21   0.6893484
## 6         Idaho  ID          West    1567582    12   0.7655102
```

Note that the default behavior is to order in ascending order. In `dplyr`, the function `desc` transforms a vector to be in descending order. So if we want to sort the table in descending order we can type

```
murders %>%
  arrange(desc(murder_rate)) %>%
  head()
```

```
##                  state abb        region population total murder_rate
## 1 District of Columbia  DC         South     601723    99   16.452753
## 2            Louisiana  LA         South    4533372   351    7.742581
## 3             Missouri  MO North Central    5988927   321    5.359892
## 4             Maryland  MD         South    5773552   293    5.074866
## 5       South Carolina  SC         South    4625364   207    4.475323
## 6             Delaware  DE         South     897934    38    4.231937
```

Nested Sorting If we are ordering by a column with ties we can use a second column to break the tie. Similarly, a third column can be used to break ties between the first and second and so on. Here we order by `region` then within region we order by murder rate:

```
murders %>%
  arrange(region, murder_rate) %>%
  head()
```

```
##           state abb    region population total murder_rate
## 1       Vermont  VT Northeast     625741     2   0.3196211
## 2 New Hampshire  NH Northeast    1316470     5   0.3798036
## 3         Maine  ME Northeast    1328361    11   0.8280881
## 4  Rhode Island  RI Northeast    1052567    16   1.5200933
## 5 Massachusetts  MA Northeast    6547629   118   1.8021791
## 6      New York  NY Northeast   19378102   517   2.6679599
```

The top n In the code above we have used the function `head` to avoid having the page fill with the entire data. If we want to see a larger proportion we can use the `top_n` function. Here are the first 10 murder rates:

```
murders %>% top_n(10, murder_rate)
```

```
##                   state abb        region population total murder_rate
## 1               Arizona  AZ          West    6392017   232    3.629527
## 2              Delaware  DE         South     897934    38    4.231937
```

```
## 3  District of Columbia  DC         South     601723    99   16.452753
## 4               Georgia  GA         South    9920000   376    3.790323
## 5             Louisiana  LA         South    4533372   351    7.742581
## 6              Maryland  MD         South    5773552   293    5.074866
## 7              Michigan  MI North Central    9883640   413    4.178622
## 8           Mississippi  MS         South    2967297   120    4.044085
## 9              Missouri  MO North Central    5988927   321    5.359892
## 10       South Carolina  SC         South    4625364   207    4.475323
```

Note that `top_n` picks the highest `n` based on the column given as a second argument. However, the rows are not sorted.

If the second argument is left blank, then it just takes the first `n` columns. This means that to see the top 10 states ranked by murder rate, sorted by murder rate we can type:

```
murders %>%
  arrange(desc(murder_rate)) %>%
  top_n(10)
```

```
## Selecting by murder_rate
```

```
##                   state abb        region population total murder_rate
## 1  District of Columbia  DC         South     601723    99   16.452753
## 2             Louisiana  LA         South    4533372   351    7.742581
## 3              Missouri  MO North Central    5988927   321    5.359892
## 4              Maryland  MD         South    5773552   293    5.074866
## 5        South Carolina  SC         South    4625364   207    4.475323
## 6              Delaware  DE         South     897934    38    4.231937
## 7              Michigan  MI North Central    9883640   413    4.178622
## 8           Mississippi  MS         South    2967297   120    4.044085
## 9               Georgia  GA         South    9920000   376    3.790323
## 10              Arizona  AZ          West    6392017   232    3.629527
```

Creating a data frame

It is sometimes useful for us to create our own data frames. You can do this using the `data.frame` function:

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90))
grades
```

```
##   names exam_1 exam_2
## 1  John     95     90
## 2  Juan     80     85
## 3  Jean     90     85
## 4   Yao     85     90
```

Warning: By default the function `data.frame` turns characters into factors:

```
class(grades$names)
```

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

To avoid this we use the rather cumbersome argument `stringsAsFactors`:

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90),
```

```
                          stringsAsFactors = FALSE)
class(grades$names)
```

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

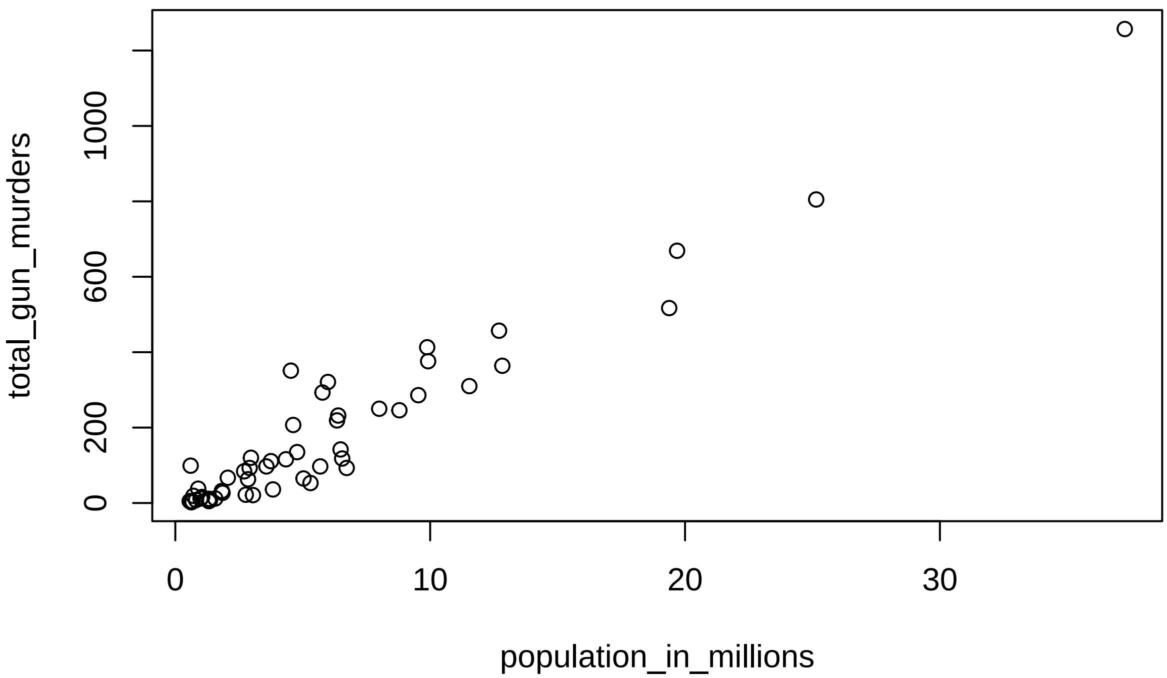
Basic plots

Exploratory data visualization is perhaps the strength of R. One can quickly go from idea to data to plot with a unique balance of flexibility and ease. For example, Excel may be easier than R but it is no where near as flexible. D3 may be more flexible and powerful than R, but it takes much longer to generate a plot. The next section is dedicated to this topic, but here we introduce some very basic plotting functions.

Scatter plots

Earlier we inferred that states with larger populations are likely to have more murders. This can be confirmed with an exploratory visualization that plots these two quantities against each other:

```
library(dslabs)
data(murders)
population_in_millions <- murders$population/10^6
total_gun_murders <- murders$total
plot(population_in_millions, total_gun_murders)
```



We can clearly see a relationship.

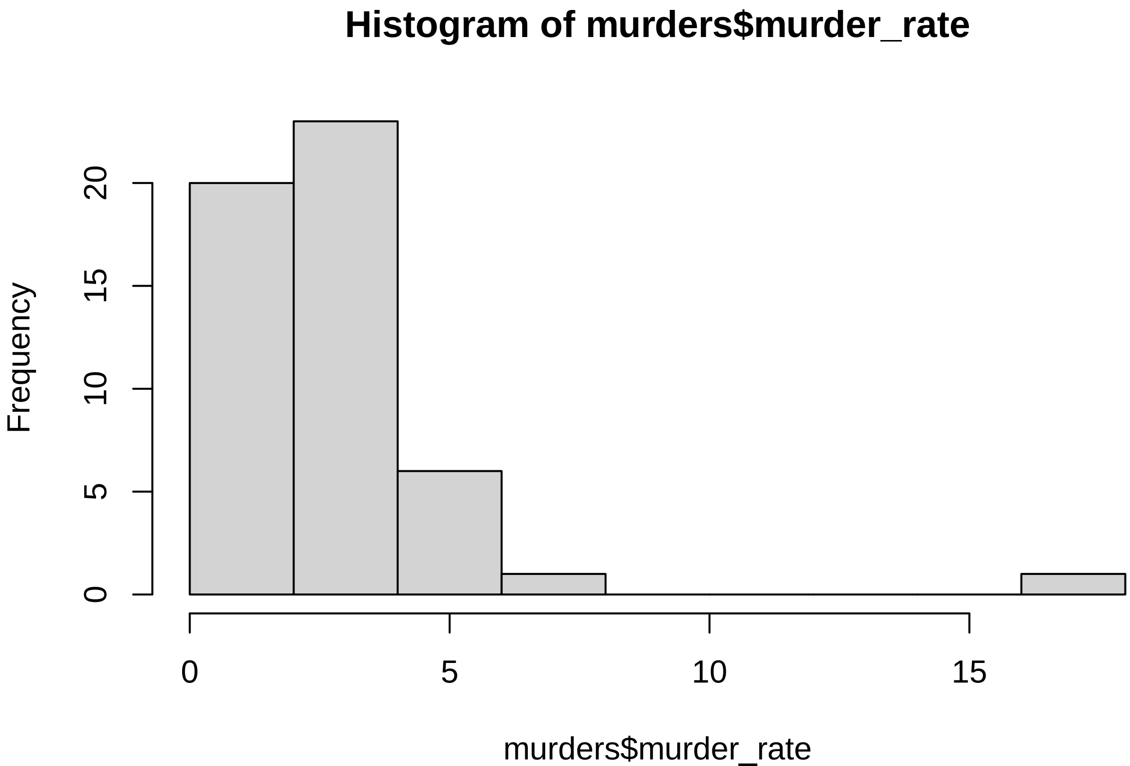
Advanced: For a quick plot that avoids accessing variables twice, we can use the `with` function

```
with(murders, plot(population, total))
```

Histograms

We will describe histograms as they relate to distributions in the next section. Here we will simply note that histograms are a powerful graphical summary of a list of numbers that gives you a general overview of the types of values you have. We can make a histogram of our murder rates by simply typing

```
library(dplyr)
murders <- mutate(murders, murder_rate = total / population * 100000)
hist(murders$murder_rate)
```



We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme case with a murder rate of more than 15:

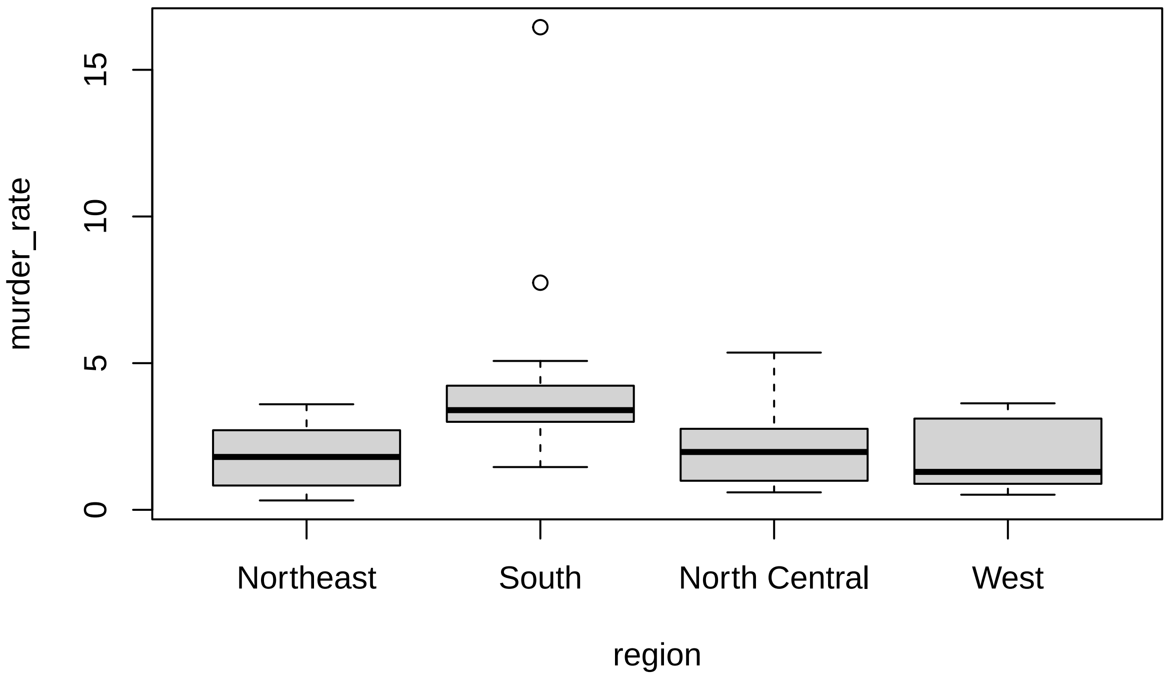
```
murders$state[which.max(murders$murder_rate)]
```

```
## [1] "District of Columbia"
```

Boxplot

Boxplots will be described in more detail in the next section as well. But here we say that they provide a more terse summary than the histogram - but they are easier to stack with other boxplots. Here we can use them to compare the different regions.

```
boxplot(murder_rate~region, data = murders)
```



We can see that the South has larger murder rates than the other three regions.

Importing Data

Thus far we have used a dataset already stored in an R object. A data scientist will rarely have such luck and will have to import data into R from either a file, a database, or some other source. We cover this in more detail later on. But because it is so common to read data from a file, we will briefly describe the key approach and function, in case you want to use your new knowledge on one of your own datasets.

Small datasets such as the one used in this lecture are typically commonly stored as Excel files. Although there are R packages designed to read Excel (xls) format, you generally want to avoid this format and save files as comma delimited (Comma-Separated Value/CSV) or tab delimited (Tab-Separated Value/TSV/TXT) files. These plain-text formats make it easier to share data since commercial software is not required for working with the data.

Paths and the Working Directory The first step is to find the file containing your data and know its *path*. When you are working in R it is useful to know your *working directory*. This is the folder in which R will save or look for files by default. You can see your working directory by typing:

```
getwd()
```

You can also change your working directory using the function `setwd`. Or you can change it through RStudio by clicking on "Session".

The functions that read and write files (there are several in R) assume you mean to look for files or write files in the working directory. Our recommended approach for beginners will have you reading and writing to the working directory. However, you can also type the full path, which will work independently of the working directory.

We have included the US murders data in a CSV file as part of the `dslabs` package. We recommend placing your data in your working directory.

Because knowing where packages store files is rather advanced, we provide the following code that finds the directory and copies the file:

```
dir <- system.file(package="dslabs") #extracts the location of package
filename <- file.path(dir,"extdata/murders.csv")
file.copy(filename, "murders.csv")
```

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

You should be able to see the file in your working directory and can check using

```
list.files()
```

`read.csv`

We are ready to read in the file. There are several functions for reading in tables. Here we introduce one included in base R:

```
dat <- read.csv("murders.csv")
head(dat)
```

```
##        state abb region population total
## 1    Alabama  AL  South    4779736   135
## 2     Alaska  AK   West     710231    19
## 3    Arizona  AZ   West    6392017   232
## 4   Arkansas  AR  South    2915918    93
## 5 California  CA   West   37253956  1257
## 6   Colorado  CO   West    5029196    65
```

We can see that we have read in the file.

Warning: `read.csv` automatically converts characters to factors. Note for example that:

```
class(dat$state)
```

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

You can avoid this using

```
dat <- read.csv("murders.csv", stringsAsFactors = FALSE)
class(dat$state)
```

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

With this call the region variable is no longer a factor but we can easily change this with:

```
require(dplyr)
dat <- mutate(dat, region = as.factor(region))
```

Programming basics

We teach R because it greatly facilitates data wrangling and analysis, the main topic of this course. Coding in R we can efficiently perform exploratory data analysis, build data analysis pipelines and prepare data visualization to communicate results. However, R is not just a data analysis environment, but a programming language. Advanced R programmers can develop complex packages and even improve R itself, but we do not cover advanced programming in this course. In this section we introduce three key programming concepts: conditional expressions, for-loops and functions. These are not just key building blocks for advanced programming, but occasionally come in handy during data analysis. We also provide a list of power functions that we dot not cover in the course but are worth knowing about as they are powerful tools commonly used by expert data analysts.

Conditional expressions

Conditional expressions are one of the basic features of programming. The most common conditional expression is the `if-else` statement. In R, we can actually perform quite a bit of data analysis without conditionals. However, they do come up occasionally and once you start writing your own functions and packages you will definitely need them.

Here is a very simple example showing the general structure of an `if-else` statement. The basic idea is to print the reciprocal of `a` unless `a` is 0:

```
a <- 0

if(a!=0){
  print(1/a)
} else{
  print("No reciprocal for 0.")
}
```

```
## [1] "No reciprocal for 0."
```

Let's look at one more example using the US murders data frame.

```
library(dslabs)
data(murders)
murder_rate <- murders$total/murders$population*100000
```

Here is a very simple example that tells us if the lowest murder rate is smaller than 0.5 per 100,000. The else statement protects us from the case in which the state with the smallest murder rate doesn't satisfy the condition.

```
ind <- which.min(murder_rate)

if(murder_rate[ind] < 0.5){
  print(murders$state[ind])
} else{
  print("Minumum murder rate is not that low.")
}
```

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

If we try it again with a rate of 0.25 we get a different answer:

```
if(murder_rate[ind] < 0.25){
  print(murders$state[ind])
} else{
  print("Minumum murder rate is not that low.")
}
```

```
## [1] "Minumum murder rate is not that low."
```

A related function that is very useful is `ifelse`. This function takes three arguments: a logical and two possible answers. If the logical is `TRUE` the first answer is returned and if `FALSE` the second. Here is an example:

```
a <- 0
ifelse(a > 0, 1/a, NA)
```

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

The function is particularly useful because it works on vectors. It examines each element of the logical vector and returns corresponding answers from them accordingly.

```
a <- c(0, 1, 2, -4, 5)
result <- ifelse(a > 0, 1/a, NA)
result
```

```
## [1]  NA 1.0 0.5  NA 0.2
```

This table helps us see what happned:

a	is_a_positive	answer1	answer2	result
0	FALSE	Inf	NA	NA
1	TRUE	1.00	NA	1.0
2	TRUE	0.50	NA	0.5
-4	FALSE	-0.25	NA	NA
5	TRUE	0.20	NA	0.2

Here is an example of how this function can be readily used to replace all the missing values in a vector with zeros:

```
data(na_example)
no_nas <- ifelse(is.na(na_example), 0, na_example)
sum(is.na(no_nas))
```

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

Two other useful functions are `any` and `all`. The `any` function takes a vector of logicals and returns `TRUE` if any of the entries are `TRUE`. The `all` function takes a vector of logicals and returns `TRUE` if all of the entries are `TRUE`. Here is an example.

```
z <- c(TRUE, TRUE, FALSE)
any(z)
```

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

```
all(z)
```

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

Defining Functions

As you become more experienced you will find yourself needeing to perform the same operations over and over. A simple example is computing the average. We can compute the average of a vector `x` using the `sum` and `length` functions: `sum(x)/length(x)`. But because we do this so often it is much more efficient to write a function that performs this operation and thus someone already wrote the `mean` function. However, you will encounter situations in which the function does not already exist so R permits you to write your own. A simple version of a function that computes the average can be defined like this:

```
avg <- function(x){
  s <- sum(x)
  n <- length(x)
  s/n
}
```

Now `avg` is a function that computes the mean:

```
x <- 1:100
identical(mean(x), avg(x))
```

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

Note that variables defined inside a function are not saved in the workspace. So while we use `s` and `n` when we call `avg`, their values are created and changed only during the call. Here are illustrative examples:

```
s <- 3
avg(1:10)
```

```
## [1] 5.5
```

```
s
```

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

Note how `s` is still 3 after we call `avg`.

In general, functions are objects, so we assign them to variable names with `<-`. The function `function` tells R you are about to define a function. The general form of a function definition looks like this.

Also note that the functions you define can have multiple arguments as well as default values. For example, we can define a function that computes either the arithmetic or geometric average depending on a user defined variable like this:

```
avg <- function(x, arithmetic = TRUE){
  n <- length(x)
  ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))
}
```

We will learn more about how to create functions through experience as we face more complex tasks.

For loops

The formula for the sum $1 + 2 + \cdots + n$ is $\frac{n(n+1)}{2}$. What if we weren't sure that was the right function? How could we check? Using what we learned about functions we can create one that computes the sum S_n:

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
```

Now if we can compute S_n for various vales of n, say $n = 1, \dots, 25$ how do we do it? Do we write 25 lines of code calling `compute_s_n`? No, that is what for loops are for in programming. Note that we are performing exactly the same task over and over and that the only thing that is changing is the value of n. For loops let us define the range that our variable takes (in our example $n = 1, \dots, 25$), then change the value as you *loop* through and evaluate an expression. The general form looks like this:

```
for(i in 1:5){
  print(i)
}
```

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

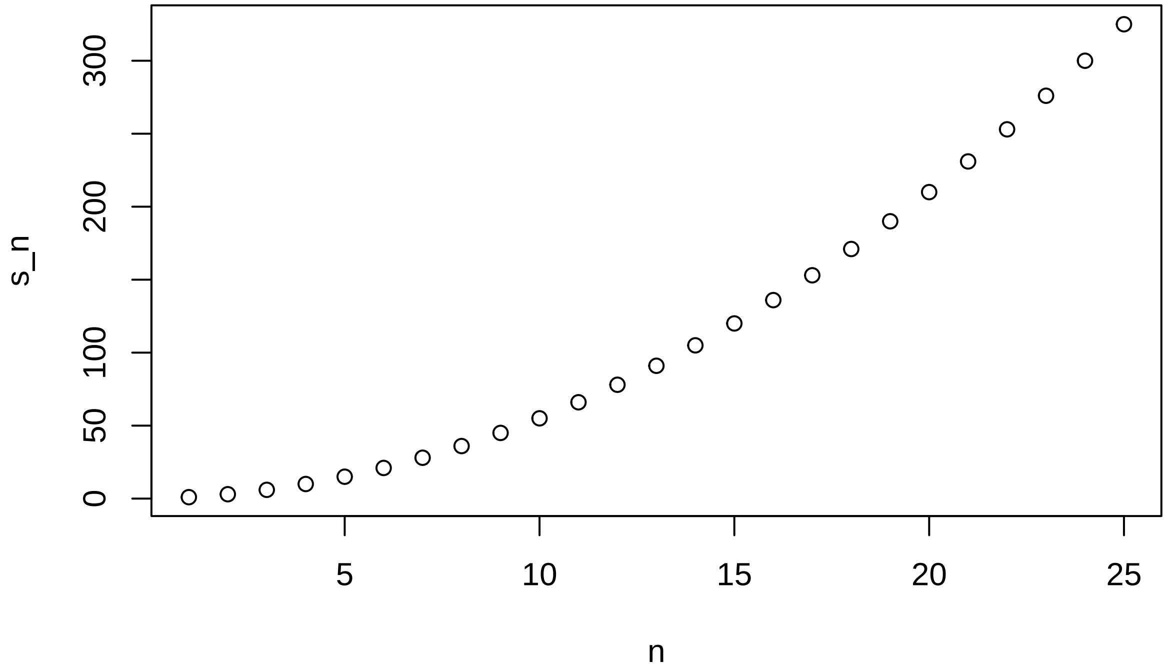
And here is the for loop we would write for our S_n example:

```
m <- 25
s_n <- vector(length = m) # create an empty vector
for(n in 1:m){
  s_n[n] <- compute_s_n(n)
}
```

In each iteration $n = 1$, $n = 2$, etc..., we compute S_n and store it in the nth entry of `s_n`.

Now we can create a plot to search for a pattern.

```
n <- 1:m
plot(n, s_n)
```

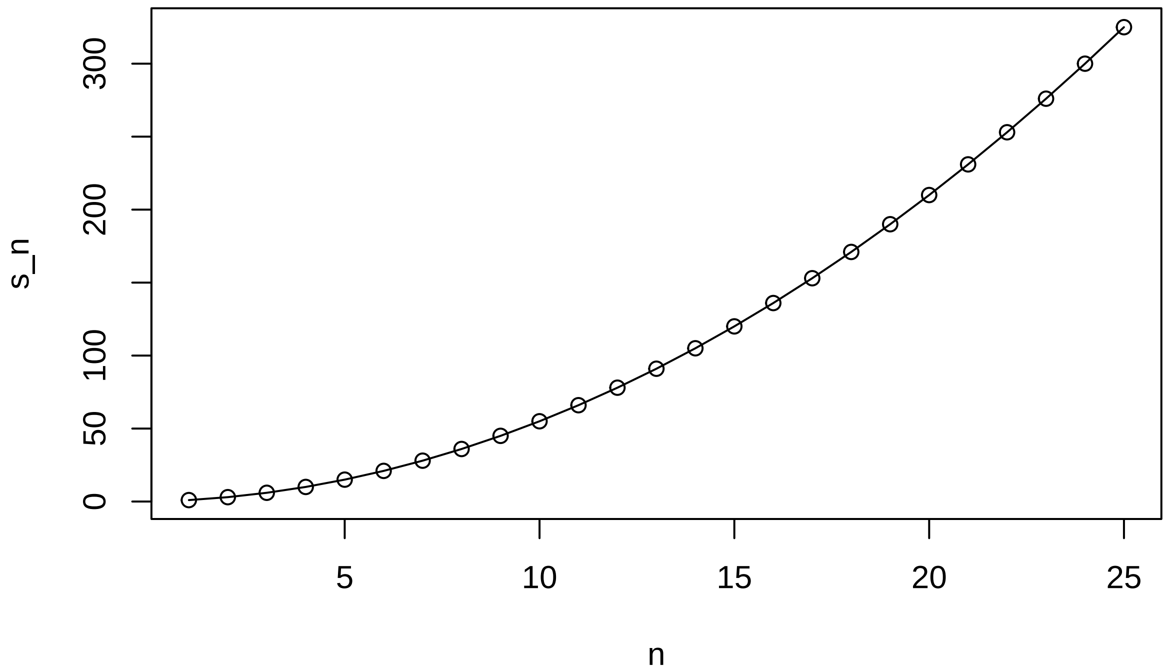


If you noticed that it appears to be a quadratic, you are on the right track because the formula is $\frac{n(n+1)}{2}$ which we can confirm with a table:

```
##   s_n formula
## 1   1       1
## 2   3       3
## 3   6       6
## 4  10      10
## 5  15      15
## 6  21      21
```

We can also overlay the two results by using the function `lines` to draw a line over the previously plotted points:

```
plot(n, s_n)
lines(n, n*(n+1)/2)
```



Other functions

It turns out that we rarely use for loops in R. This is because there are usually more powerful ways to perform the same task. Functions that are typically used instead of for loops are the apply family: `apply`, `sapply`, `tapply`, and `mapply`. We do not cover these functions in this course but they are worth learning if you intend to go beyond this introduction. Other functions that are widely used are `split`, `cut`, and `Reduce`.

Assessments

1. What will this conditional expression return?

```
x <- c(1,2,-3,4)

if(all(x>0)){
  print("All positives")
} else{
  print("Not all positives")
}
```

```
## [1] "Not all positives"
```

"Not all positives", because there is at least one element of `x` that is not positive, namely the 3rd one.

2. Which of the following expressions is always `FALSE` when at least one entry of a logical vector `x` is TRUE?

   A. `all(x)` B. `any(x)` C. `any(!x)` D. `all(!x)`

D. If at least one entry of `x` is `TRUE`, the at least one entry of `!x` is `FALSE`. Therefore, `all(!x)` is `FALSE` since there is at least one entry of `!x` that is `FALSE`.

Here are counterexample for the other three choices:

```r
# counter-example for A that returns TRUE
x <- c(TRUE, TRUE)
all(x)
```

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

```r
# counter-example for B and C that return TRUE
x <- c(TRUE, FALSE)
any(x)
```

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

```r
any(!x)
```

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

3. The function `nchar` tells you how many characters long a character vector is. For example:

```r
library(dslabs)
data(murders)
char_len <- nchar(murders$state)
char_len[1:5]
```

```
## [1]  7  6  7  8 10
```

Write a line of code that assigns to the object `new_names` the state abbreviation when the state name is longer than 8 characters.

```r
new_names <- ifelse(nchar(murders$state) > 8,
                    murders$abb,
                    murders$state)
new_names
```

```
##  [1] "Alabama"  "Alaska"   "Arizona"  "Arkansas" "CA"       "Colorado"
##  [7] "CT"       "Delaware" "DC"       "Florida"  "Georgia"  "Hawaii"
## [13] "Idaho"    "Illinois" "Indiana"  "Iowa"     "Kansas"   "Kentucky"
## [19] "LA"       "Maine"    "Maryland" "MA"       "Michigan" "MN"
## [25] "MS"       "Missouri" "Montana"  "Nebraska" "Nevada"   "NH"
## [31] "NJ"       "NM"       "New York" "NC"       "ND"       "Ohio"
## [37] "Oklahoma" "Oregon"   "PA"       "RI"       "SC"       "SD"
## [43] "TN"       "Texas"    "Utah"     "Vermont"  "Virginia" "WA"
## [49] "WV"       "WI"       "Wyoming"
```