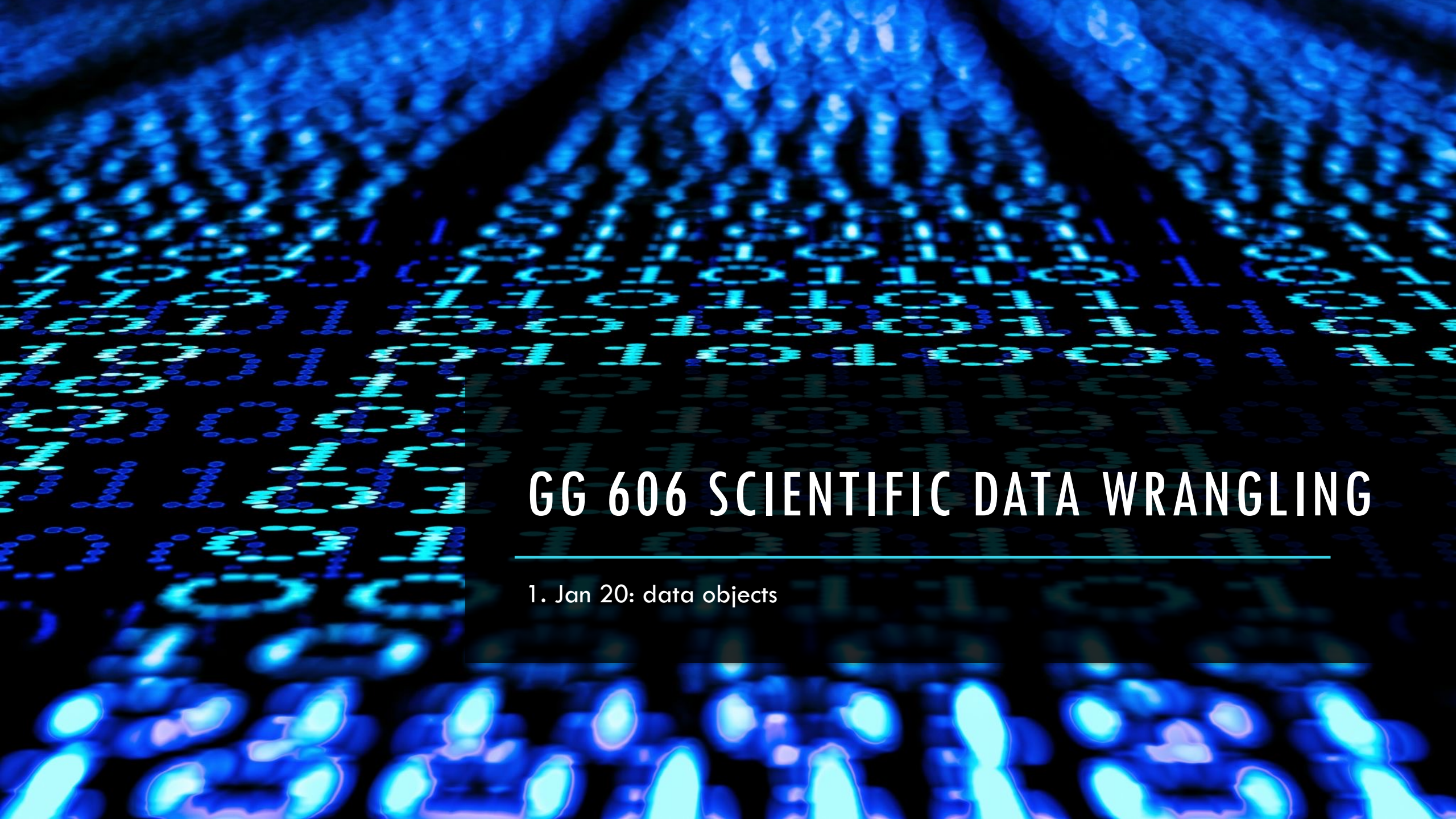


GG 606 SCIENTIFIC DATA WRANGLING

1. Jan 20: data objects

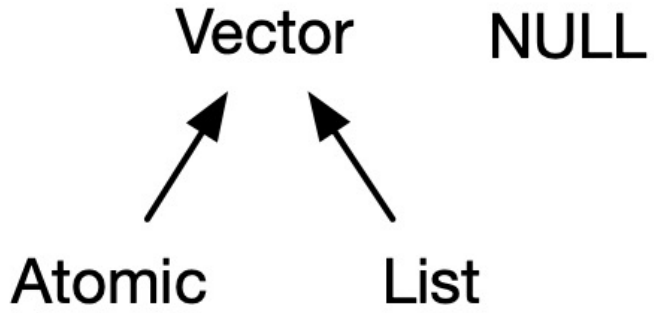
DIGGING DEEPER INTO R - 1

- five data types most often used in data analysis:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

- almost all other objects are derived from these five types
- note that even single numbers or characters are vectors
- we can find out about the structure of any object in R with the `str()` function

VECTORS IN R



- Vectors come in two flavours: atomic vectors and lists, each has
  - Type, `typeof()`, what it is.
  - Length, `length()`, how many elements it contains.
  - Attributes, `attributes()`, additional arbitrary metadata.
- four common types of atomic vector (created with `c()`):
  - logical, integer, double (often called numeric), and character
  - **all elements of an atomic vector must be the same type**
- You can convert (i.e., coerce) between types – and R usually does this automatically
  - so be aware of this behaviour which can be the source of unexpected results

```
str(c("a", 1))
#> chr [1:2] "a" "1"
```

```
str(c("TRUE", TRUE))
 #chr [1:2] "TRUE" "TRUE"
```

VECTORS IN R

- Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using `list()` instead of `c()`:

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 4
#> $ : int [1:3] 1 2 3
#> $ : chr "a"
#> $ : logi [1:3] TRUE FALSE TRUE
#> $ : num [1:2] 2.3 5.9
```

- lists can contain other lists

```
x <- list(list(list(list()))) str(x)
#> List of 1 #> $ :List of 1
#> ..$ :List of 1
#> .. ..$ : list()
```

GENERATING NUMERIC VECTORS

- Numeric vectors contain only numbers, with any number of decimal places.
- There are shortcuts for generating common kinds of vectors:

```
seq(-3, 6, by = 1.75) # Sequence from -3 to 6, increments of 1.75

rep(c(-1, 0, 1), times = 3) # Repeat c(-1,0,1) 3 times

rep(c(-1, 0, 1), each = 3) # Repeat each element 3 times
```

- Integer vectors can be created as ranges using the colon :

```
> n <- 12
> 1:n
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> n:4
[1] 12 11 10  9  8  7  6  5  4
>
```

ATTRIBUTES

- All objects can have arbitrary additional attributes, used to store metadata about the object
- Attributes can be thought of as a named list (with unique names)
- Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`
- Three most important:
  - Names, a character vector giving each element a name, described in names.
  - Dimensions, used to turn vectors into matrices and arrays, described in matrices and arrays.
  - Class, used to implement the S3 object system

```
y <- 1:10
attr(y, "my_attribute") <- "This is a vector"
attr(y, "my_attribute")
#> [1] "This is a vector"
str(attributes(y))
#> List of 1 #> $ my_attribute: chr "This is a vector"
```

Not all elements of a vector need to have a name, If all names are missing, names() will return NULL.

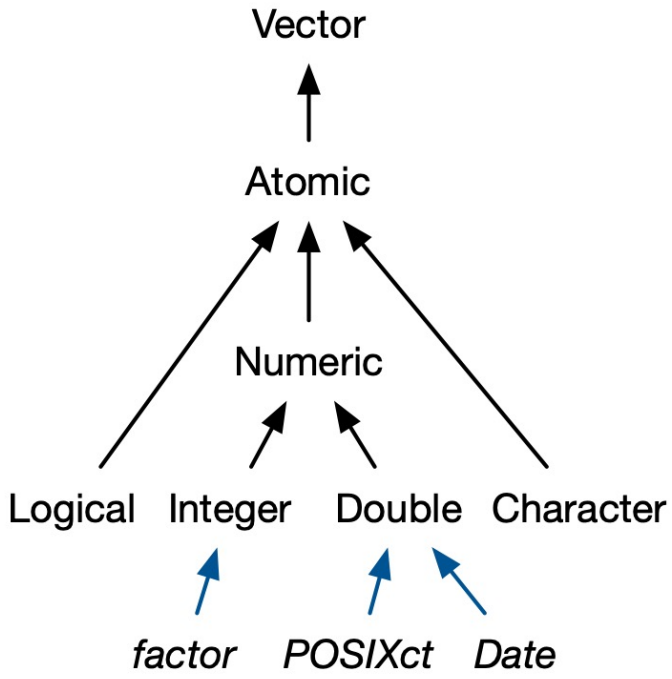
S3 ATOMIC VECTORS

- a **class** attribute turns an object into an **S3 object**
  - behave differently from a regular vector when passed to a **generic** function
  - You can get the underlying base type by unclass()ing it, which strips the class attribute, causing it to lose its special behaviour:
  - A **generic function** defines an interface, which uses a different implementation depending on the class of an argument

```
f <- factor(c("a", "b", "c"))
typeof(f)
#> [1] "integer"
attributes(f)
#> $levels
#> [1] "a" "b" "c"
#>
#> $class
#> [1] "factor”
unclass(f)
#> [1] 1 2 3
#> attr(,"levels")
#> [1] "a" "b" "c"
```

S3 ATOMIC VECTORS

- Four important S3 vectors used in base R
  - Categorical data, where values come from a fixed set of levels recorded in **factor** vectors.
  - Dates (with day resolution), which are recorded in **Date** vectors.
  - Date-times (with second or sub-second resolution), which are stored in **POSIXct** vectors.
  - Durations, which are stored in **difftime** vectors.



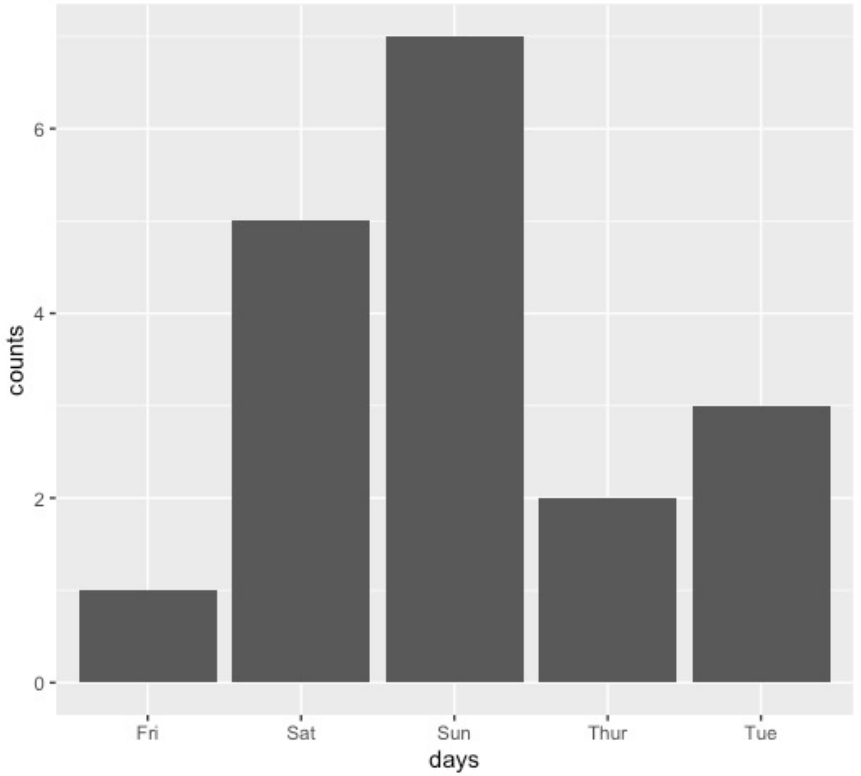
FACTORS

- A factor is a vector that can contain only predefined values, and is used to store categorical data
- Factors are **built on top of integer vectors** using two attributes:
  - the class, “factor”, which makes them behave differently from regular integer vectors, and
  - levels, which defines the set of allowed values.
- Factors are useful when you know the possible values a variable may take,
  - if you don’t see all values in a given dataset
- Factors look (and often behave) like character vectors, they are actually integers

```
x <- factor(c("a", "b", "b", "a"))
x
#> [1] a b b a
#> Levels: a b
```

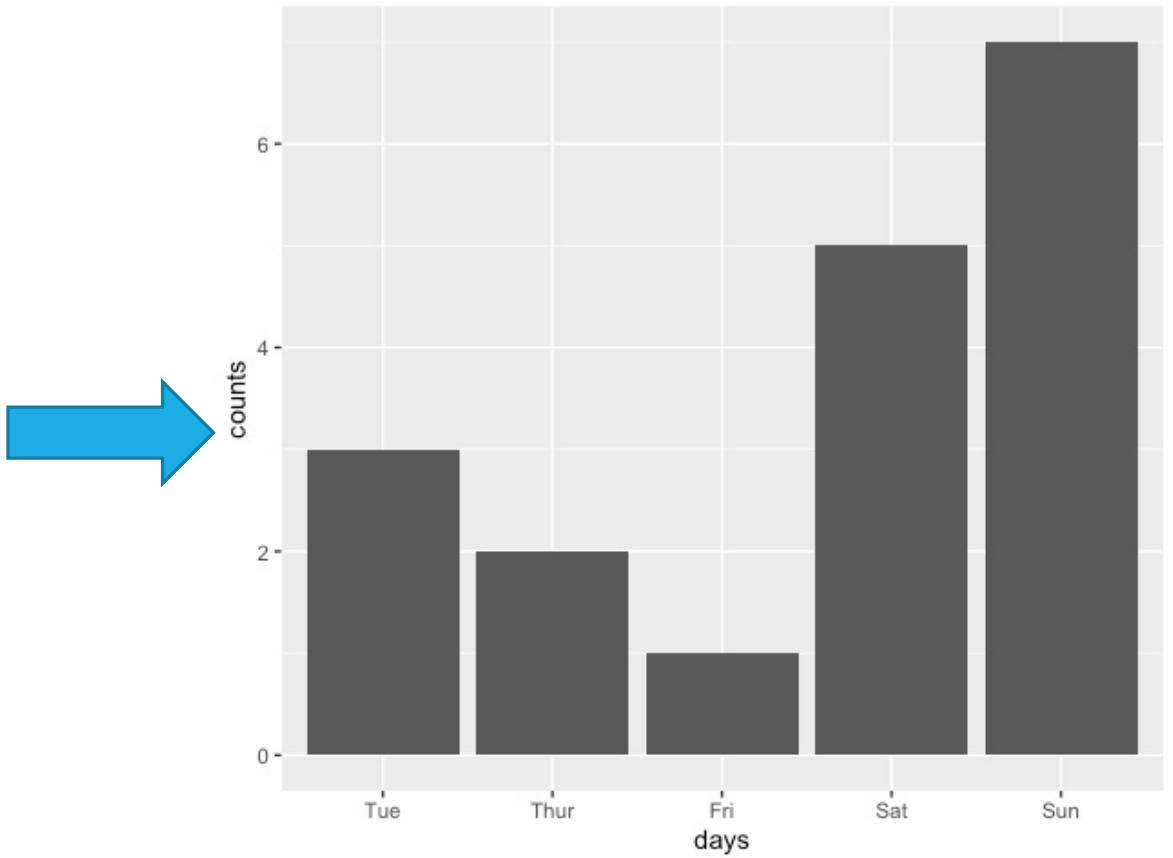
FACTORS

```
library(ggplot2)
days <- c("Fri", "Sat", "Sun", "Tue", "Thur")
counts <- c(1,5,7,3,2)
df <- data.frame(days, counts)
ggplot(df, aes(x=days, y=counts)) +
geom_bar(stat="identity")
```



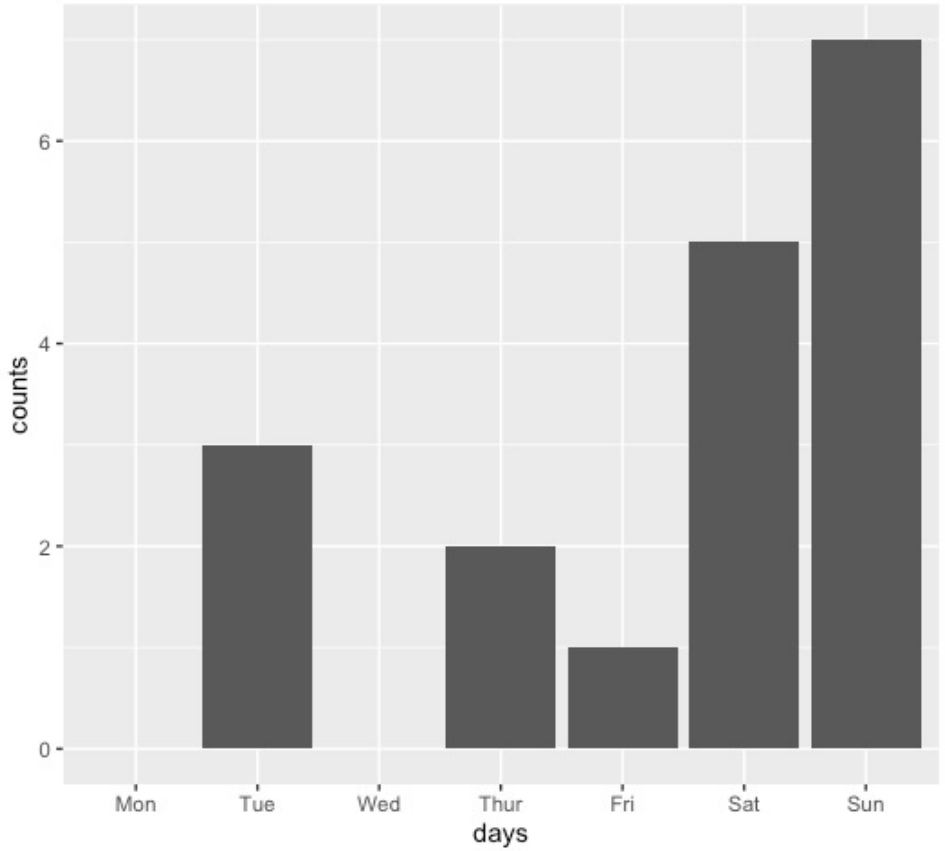
FACTORS

```
days <- factor(c("Fri", "Sat", "Sun", "Tue", "Thur"),
levels=c("Mon", "Tue", "Wed", "Thur", "Fri", "Sat",
"Sun"), ordered = TRUE)
counts <- c(1,5,7,3,2)
df <- data.frame(days, counts)
ggplot(df, aes(x=days, y=counts)) +
geom_bar(stat="identity")
```



FACTORS

```
ggplot(df, aes(x=days, y=counts)) +
geom_bar(stat="identity") +
scale_x_discrete(drop=FALSE)
```



- Character data usually can't go directly into a statistical model or graph but factor data can
  - It has an underlying numeric representation:

DATES

- Date vectors are built on top of **double** vectors. They have class “Date” and no other attributes:

```
today <- Sys.Date()
typeof(today)
#> [1] "double"
attributes(today)
#> $class
#> [1] "Date”
today+1
[1] "2022-01-20"
as.numeric(today)
[1] 19011
as.numeric(as.Date("1970-01-01"))
[1] 0
as.numeric(as.Date("1970-01-03"))
[1] 2
as.numeric(as.Date("1970-01-02"))
[1] 1
```

MATRICES AND ARRAYS

- multi-dimensional **arrays**
- **matrix** is a special case of the array, which has two dimensions
- used in statistical manipulations, rarely read in directly from data

DATA FRAMES

```
dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
str(dfl) #> 'data.frame': 3 obs. of 2 variables:
#> $ x: int 1 2 3
#> $ y:List of 3
#> ..$ : int 1 2
#> ..$ : int 1 2 3
#> ..$ : int 1 2 3 4
#> ..- attr(*, "class")= chr "AsIs"
```

- a data frame is a list of **equal-length** vectors
- data frame has `names()`, `colnames()`, and `rownames()`
- You create a data frame using `data.frame()`, which takes named vectors as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
'data.frame':    3 obs. of  2 variables:
 $ x: int  1 2 3
 $ y: chr  "a" "b" "c"
```

- You can have any data type as a column in a data frame
  - HOWEVER many functions that work with data frames assume that all columns are atomic vectors

R IS A FUNCTIONAL PROGRAMMING LANGUAGE

- functions are objects in their own right and the fundamental building block of the language
- use functions as much as possible in your own data analysis code/pipelines
  - reduce duplication
  - improve readability
  - reduced chances of errors in your code
- some functions can accept a function as an argument, these are known as **functionals**
  - **apply functions are functionals**

USING FUNCTIONS IN R

Imagine you've loaded a data file, like the one below, that uses –99 to represent missing values. You want to replace all the –99s with NAs.

```r
# Generate a sample dataset
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6,
rep = TRUE)))
names(df) <- letters[1:6]
df #> a b c d e f
#> 1 7 5 -99 2 5 2
#> 2 5 5 5 3 6 1
#> 3 6 8 5 9 9 4
#> 4 4 2 2 6 6 8
#> 5 6 7 6 -99 10 6
#> 6 9 -99 4 7 5 1
```

https://github.com/colinr23/stampr

1)

```r
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

2)

```r
fix_missing <- function(x)
{   x[x == -99] <- NA
x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)
```

3)

```r
fix_missing <- function(x) {
x[x == -99] <- NA
x
}
df[] <- lapply(df, fix_missing)
```

WHY WRITE FUNCTIONS IN R

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).

DIGGING DEEPER INTO R - 2

- dplyr uses the magrittr forward pipe operator, usually called simply a **pipe**.
- We write pipes like %>% (Ctrl+Shift+M or ⌘ +Shift+M).
- Pipes take the object on the *left* and apply the function on the *right*:
  - `x %>% f()` = apply function f to object x

```
library(dplyr)
library(gapminder)
gapminder %>% filter(country == "Canada") %>% head(2)
```

```
## # A tibble: 2 x 6
##   country continent  year lifeExp      pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Canada  Americas   1952    68.8 14785584    11367.
## 2 Canada  Americas   1957    70.0 17010154    12490.
```

USING PIPES

- Pipes are clearest to read when you have each function on a separate line
- Stuff to the left of the pipe is passed to the *first argument* of the function on the right. Other arguments go on the right in the function.
- If you ever find yourself piping a function where data are not the first argument, use . in the data argument instead.

```
take_this_data %>%
    do_first_thing(with = this_value) %>%
    do_next_thing(using = that_value) %>% ...
```

- When creating a new object from the output of piped functions, you place the assignment operator *at the beginning*.

LOGICAL OPERATORS

- Logical operators test boolean conditions and return TRUE, FALSE, or NA

```
gapminder %>%
    filter(country == "Oman" &
           year > 1980 &
           year <= 2000 )
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp     pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Oman    Asia       1982    62.7 1301048    12955.
## 2 Oman    Asia       1987    67.7 1593882    18115.
## 3 Oman    Asia       1992    71.2 1915208    18617.
## 4 Oman    Asia       1997    72.5 2283635    19702.
```

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

REFERENCES AND ADDITIONAL RESOURCES

- Some materials in this lecture obtained from the following sources:
  - Advanced R Textbook - http://adv-r.had.co.nz/
  - Intermediate R workshop - https://clanfear.github.io/Intermediate_R_Workshop/

https://swcarpentry.github.io/r-novice-inflammation/02-func-R/

https://besjournals.onlinelibrary.wiley.com/doi/10.1111/2041-210X.12282

SUMMARY

- While you can get by without it – eventually – you will need to get a deeper understanding of R as a functional programming language and its basic data types
- vectors are a core data structure in R, and are the base of other object types
- functions are a core feature in R, and you should strive to write functions in your R code as much as possible

GG 606 SCIENTIFIC DATA WRANGLING

1. Jan 20: data objects