Augmented vectors, factor + labelled class

EDUC 263: Introduction to Programming and Data Management Using R

Libraries we will use

Load the packages we will use by running this code chunk:

```
library(tidyverse)
library(haven)
library(labelled)
library(lubridate)
```

If package not yet installed, then must install before you load. Install in "console" rather than .Rmd file:

- Generic syntax: `install.packages("package_name")`
- Install "tidyverse": `install.packages("tidyverse")`

Note: When we load package, name of package is not in quotes; but when we install package, name of package is in quotes:

- `install.packages("tidyverse")`
- `library(tidyverse)`

Dataset we will use

```
rm(list = ls()) # remove all objects

load(url("https://github.com/anyone-can-cook/rclass1/raw/master/data/prospect_l
```

Attributes and augmented vectors

Review data types and structures

Review data structures: **Vectors**

Two types of vectors:

1. **Atomic vectors**
2. **Lists**

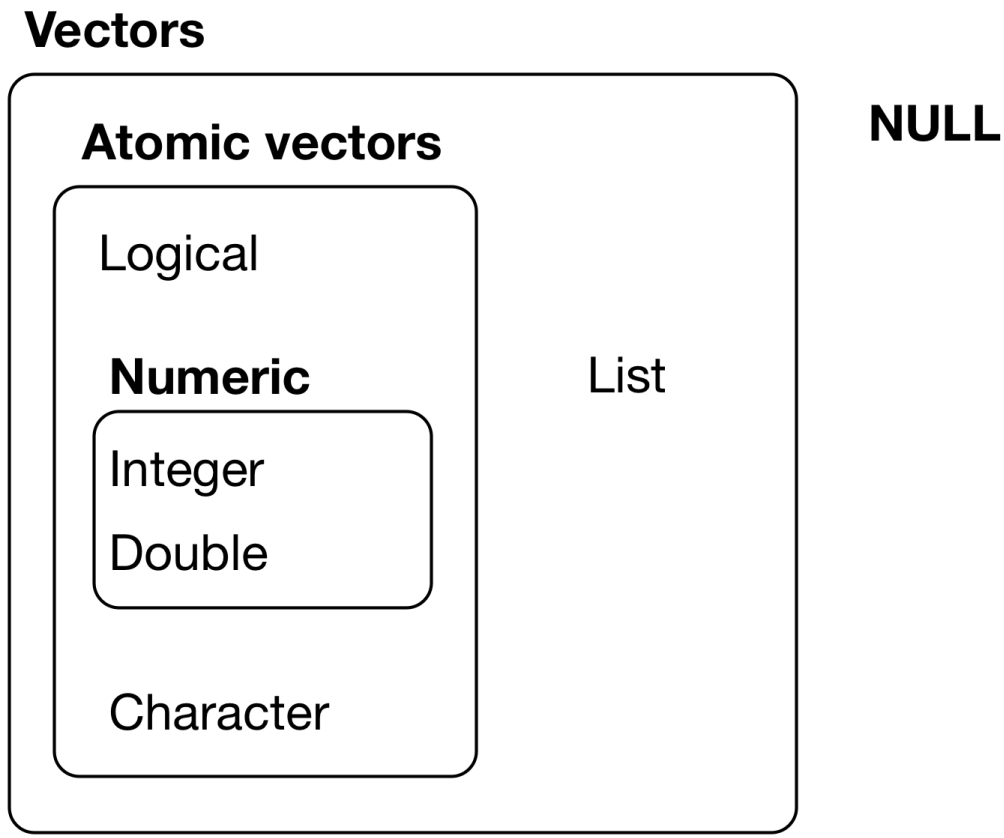

Figure 1: Overview of data structures (Grolemund and Wickham, 2018)

Review data structures: atomic vectors

An **atomic vector** is a collection of values

- Each value in an atomic vector is an **element**
- All elements within vector must have same **data type**

```
(a <- c(1,2,3)) # parentheses () assign and print object in one step
#> [1] 1 2 3
length(a) # length = number of elements
#> [1] 3
typeof(a) # numeric atomic vector, type=double
#> [1] "double"
str(a) # investigate structure of object
#>  num [1:3] 1 2 3
```

Can assign **names** to vector elements, creating a **named atomic vector**

```
(b <- c(v1=1,v2=2,v3=3))
#> v1 v2 v3
#>  1  2  3
length(b)
#> [1] 3
typeof(b)
#> [1] "double"
str(b)
#>  Named num [1:3] 1 2 3
#>  - attr(*, "names")= chr [1:3] "v1" "v2" "v3"
```

Review data structures: lists

- Like atomic vectors, **lists** are objects that contain **elements**
- However, **data type** can differ across elements within a list
  - E.g., an element of a list can be another list

```
list_a <- list(1,2,"apple")
typeof(list_a)
#> [1] "list"
length(list_a)
#> [1] 3
str(list_a)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : chr "apple"

list_b <- list(1, c("apple", "orange"), list(1, 2))
length(list_b)
#> [1] 3
str(list_b)
#> List of 3
#>  $ : num 1
#>  $ : chr [1:2] "apple" "orange"
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
```

Review data structures: lists

Like atomic vectors, elements within a list can be named, thereby creating a **named list**

```
# not named
str(list_b)
#> List of 3
#>  $ : num 1
#>  $ : chr [1:2] "apple" "orange"
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2

# named
list_c <- list(v1=1, v2=c("apple", "orange"), v3=list(1, 2, 3))
str(list_c)
#> List of 3
#>  $ v1: num 1
#>  $ v2: chr [1:2] "apple" "orange"
#>  $ v3:List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

Review data structures: data frames

A **data frame** is a list with the following characteristics:

- All the elements must be **vectors** with the same **length**
- Data frames are **augmented lists** because they have additional **attributes**

```
# a regular list
(list_d <- list(col_a = c(1,2,3), col_b = c(4,5,6), col_c = c(7,8,9)))
#> $col_a
#> [1] 1 2 3
#>
#> $col_b
#> [1] 4 5 6
#>
#> $col_c
#> [1] 7 8 9
typeof(list_d)
#> [1] "list"
attributes(list_d)
#> $names
#> [1] "col_a" "col_b" "col_c"
```

Review data structures: data frames

```r
# a data frame
(df_a <- data.frame(col_a = c(1,2,3), col_b = c(4,5,6), col_c = c(7,8,9)))
#>   col_a col_b col_c
#> 1     1     4     7
#> 2     2     5     8
#> 3     3     6     9
typeof(df_a)
#> [1] "list"
attributes(df_a)
#> $names
#> [1] "col_a" "col_b" "col_c"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2 3
```

Attributes and augmented vectors

Atomic vectors versus augmented vectors

Atomic vectors [our focus so far]

- I think of atomic vectors as "just the data"
- Atomic vectors are the building blocks for augmented vectors

Augmented vectors

- **Augmented vectors** are atomic vectors with additional **attributes** attached

Attributes

- **Attributes** are additional "metadata" that can be attached to any object (e.g., vector or list)

Example: Variables of a dataset

- A data frame is a list
- Each element in the list is a variable, which consists of:
  - Atomic vector ("just the data")
  - Variable **name**, which is an attribute we attach to the element/variable
  - Any other attributes we want to attach to each element/variable

Other examples of attributes in R

- Value **labels**: Character labels (e.g., "Charter School") attached to numeric values
- Object **class**: Specifies how object is treated by object oriented programming language

Main takeaway:

- **Augmented vectors** are **atomic vectors** (just the data) with additional **attributes**

Attributes and functions to identify/modify attributes

Description of attributes from Grolemund and Wickham 20.6

- "Any vector can contain arbitrary additional **metadata** through its **attributes**"
- "You can think of **attributes** as named list of vectors that can be attached to any object"

Functions to identify and modify attributes

- `attributes()` function to describe all attributes of an object
- `attr()` to see individual attribute of an object or set/change an individual attribute of an object

attributes() function: describes all attributes of an object

```
# pull up help file for the attributes() function
?attributes
```

Attributes of a **named atomic vector**:

```
# create named atomic vector
(vector1 <- c(a = 1, b = 2, c = 3, d = 4))
#> a b c d
#> 1 2 3 4
attributes(vector1)
#> $names
#> [1] "a" "b" "c" "d"

# remove all attributes from the object
attributes(vector1) <- NULL
vector1
#> [1] 1 2 3 4
attributes(vector1)
#> NULL
```

attributes() function, attributes of a variable in a data frame

Accessing variable using `[[]]` subset operator

- Recall `object_name[["element_name"]]` accesses contents of the element
- If object is a data frame, `df_name[["var_name"]]` accesses contents of variable
  - For simple vars like `firstgen`, syntax yields an atomic vector ("just the data")
- Shorthand syntax for `df_name[["var_name"]]` is `df_name$var_name`

```
str(wwlist[["firstgen"]])
#>  chr [1:268396] NA "N" "N" "N" NA "N" "N" "Y" "Y" "N" "N" "N" "N" "N" "N" ...
attributes(wwlist[["firstgen"]])
#> NULL

str(wwlist$firstgen) # same same
#>  chr [1:268396] NA "N" "N" "N" NA "N" "N" "Y" "Y" "N" "N" "N" "N" "N" "N" ...
attributes(wwlist$firstgen)
#> NULL
```

Accessing variable using `[]` subset operator

- `object_name["element_name"]` creates object of same type as `object_name`
- If object is a data frame, `df_name["var_name"]` returns a data frame containing just the `var_name` column

```
str(wwlist["firstgen"])
attributes(wwlist["firstgen"])
```

attributes() function, attributes of lists and data frames

Attributes of a **named list**:

```
list2 <- list(col_a = c(1,2,3), col_b = c(4,5,6))
str(list2)
#> List of 2
#>  $ col_a: num [1:3] 1 2 3
#>  $ col_b: num [1:3] 4 5 6
attributes(list2)
#> $names
#> [1] "col_a" "col_b"
```

Note that the `names` attribute is an attribute of the list, not an attribute of the elements within the list (which are atomic vectors)

```
list2[['col_a']] # the element named 'col_a'
#> [1] 1 2 3
str(list2[['col_a']]) # structure of the element named 'col_a'
#>  num [1:3] 1 2 3
attributes(list2[['col_a']]) # attributes of element named 'col_a'
#> NULL
```

attributes() function, attributes of lists and data frames

Attributes of a **data frame**:

```
list3 <- data.frame(col_a = c(1,2,3), col_b = c(4,5,6))
str(list3)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ col_a: num  1 2 3
#>  $ col_b: num  4 5 6
attributes(list3)
#> $names
#> [1] "col_a" "col_b"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2 3
```

Note: attributes `names`, `class` and `row.names` are attributes of the data frame

- they are not attributes of the elements (variables) within the data frame, which are atomic vectors (i.e., just the data)

```
str(list3[['col_a']]) # structure of the element named 'col_a'
#>  num [1:3] 1 2 3
attributes(list3[['col_a']]) # attributes of element named 'col_a'
#> NULL
```

attr() function: get or set specific attributes of an object

Syntax

- Get: `attr(x, which, exact = FALSE)`
- Set: `attr(x, which) <- value`

Arguments

- `x` : an object whose attributes are to be accessed
- `which` : a non-empty character string specifying which attribute is to be accessed
- `exact` (logical): should `which` be matched exactly? default is `exact = FALSE`
- `value` : an object, new value of attribute, or `NULL` to remove attribute

Using `attr()` to get specific attribute of an object

```
vector1 <- c(a = 1, b= 2, c= 3, d = 4)
attributes(vector1)
#> $names
#> [1] "a" "b" "c" "d"
attr(x=vector1, which = "names", exact = FALSE)
#> [1] "a" "b" "c" "d"
attr(vector1, "names")
#> [1] "a" "b" "c" "d"
attr(vector1, "name") # we don't provide exact name of attribute
#> [1] "a" "b" "c" "d"
attr(vector1, "name", exact = TRUE) # don't provide exact name of attribute
#> NULL
```

attr() function: get or set specific attributes of an object

Syntax

- Get: `attr(x, which, exact = FALSE)`
- Set: `attr(x, which) <- value`

Arguments

- `x` : an object whose attributes are to be accessed
- `which` : a non-empty character string specifying which attribute is to be accessed
- `exact` (logical): should `which` be matched exactly? default is `exact = FALSE`
- `value` : an object, new value of attribute, or `NULL` to remove attribute

Using `attr()` to set specific attribute of an object (output omitted)

```
(vector1 <- c(a = 1, b= 2, c= 3, d = 4))
attributes(vector1) # see all attributes

attr(x=vector1, which = "greeting") <- "Hi!" # create new attribute
attr(x=vector1, which = "greeting") # see attribute

attr(vector1, "farewell") <- "Bye!" # create attribute

attr(x=vector1, which = "names") # see names attribute
attr(x=vector1, which = "names") <- NULL # delete names attribute

attributes(vector1) # see all attributes
```

attr() function, apply on data frames

Using `wwlist`, create data frame with three variables

```
wwlist_small <- wwlist[1:25, ] %>% select(hs_state,firstgen,med_inc_zip)
str(wwlist_small)
attributes(wwlist_small)
```

Get/set attribute of a data frame

```
#get/examine names attribute
attr(x=wwlist_small, which = "names")
str(attr(x=wwlist_small, which = "names")) # names attribute is character atomic

#add new attribute to data frame
attr(x=wwlist_small, which = "new_attribute") <- "contents of new attribute"
attributes(wwlist_small)
```

Get/set attribute of a variable in data frame

```
str(wwlist_small$med_inc_zip)
attributes(wwlist_small$med_inc_zip)

#create attribute for variable med_inc_zip
attr(wwlist_small$med_inc_zip, "inc attribute") <- "inc attribute contents"

#investigate attribute for variable med_inc_zip
attributes(wwlist_small$med_inc_zip)
str(wwlist_small$med_inc_zip)
```

Why add attributes to data frame or variables of data frame?

Pedagogical reasons

- Important to know how you can apply `attributes()` and `attr()` to data frames and to variables within data frames

Example practical application: interactive dashboards

- When creating "dashboard" you might want to add "tooltips"
  - "Tooltip" is a message that appears when cursor is positioned over an icon
  - The text in the tooltip is the contents of an attribute
- Example dashboard: LINK

Student exercises

1. Using `wwlist`, create data frame of 30 observations with three variables: `state`, `zip5`, `pop_total_zip`
2. Return all attributes of this new data frame using `attributes()`. Then, get the `names` attribute of the data frame using `attr()`.
3. Add a new attribute to the data frame called `attribute_data` whose content is `"new attribute of data"`. Then, return all attributes of the data frame as well as get the value of the newly created `attribute_data`.
4. Return the attributes of the variable `pop_total_zip` in the data frame.
5. Add a new attribute to the variable `pop_total_zip` called `attribute_variable` whose content is `"new attribute of variable"`. Then, return all attributes of the variable as well as get the value of the newly created `attribute_variable`.

Solution to student exercises

```r
# Part 1
wwlist_exercise <- wwlist[1:30, ] %>% select(state, zip5, pop_total_zip)

# Part 2
attributes(wwlist_exercise)
attr(x=wwlist_exercise, which = "names")

# Part 3
attr(x=wwlist_exercise, which = "attribute_data") <- "new attribute of data"

attributes(wwlist_exercise)
attr(wwlist_exercise, which ="attribute_data")

# Part 4
attributes(wwlist_exercise$pop_total_zip)

# Part 5
attr(wwlist_exercise$pop_total_zip, "attribute_variable") <- "new attribute of

attributes(wwlist_exercise$pop_total_zip)
attr(wwlist_exercise$pop_total_zip, "attribute_variable")
```

Object class

Object class

Every object in R has a **class**

- Class is an **attribute** of an object
- Object class controls how functions work and defines the rules for how objects can be treated by object oriented programming language
  - E.g., which functions you can apply to object of a particular class
  - E.g., what the function does to one object class, what it does to another object class

You can use the `class()` function to identify object class:

```
(vector2 <- c(a = 1, b= 2, c= 3, d = 4))
#> a b c d
#> 1 2 3 4
typeof(vector2)
#> [1] "double"
class(vector2)
#> [1] "numeric"
```

When I encounter a new object I often investigate object by applying `typeof()`, `class()`, and `attributes()` functions:

```
typeof(vector2)
#> [1] "double"
class(vector2)
#> [1] "numeric"
attributes(vector2)
#> $names
```

Why is object class important?

Functions care about object **class**, not object **type**

Specific functions usually work with only particular **classes** of objects

- "Date" functions usually only work on objects with a date class
- "String" functions usually only work on objects with a character class
- Functions that do mathematical computation usually work on objects with a numeric class

Functions care about object **class**, not object **type**

Example: `sum()` applies to **numeric**, **logical**, or **complex** class objects

Apply `sum()` to object with class = **logical**:

```
x <- c(TRUE, FALSE, NA, TRUE)
typeof(x)
#> [1] "logical"
class(x)
#> [1] "logical"
sum(x, na.rm = TRUE)
#> [1] 2
```

Apply `sum()` to object with class = **numeric**:

```
typeof(wwlist$med_inc_zip)
#> [1] "double"
class(wwlist$med_inc_zip)
#> [1] "numeric"
wwlist$med_inc_zip[1:5]
#> [1] 92320.5 63653.0 88344.5 88408.5 82895.0
sum(wwlist$med_inc_zip[1:5], na.rm = TRUE)
#> [1] 415621.5
```

What happens when we try to apply `sum()` to an object with class = **character**?

```
typeof(wwlist$hs_city)
class(wwlist$hs_city)
```

Functions care about object **class**, not object **type**

Example: `year()` from `lubridate` package applies to date-time objects

Apply `year()` to object with class = **Date**:

```
wwlist$receive_date[1:5]
#> [1] "2016-05-31" "2016-05-31" "2016-05-31" "2016-05-31" "2016-05-31"
typeof(wwlist$receive_date)
#> [1] "double"
class(wwlist$receive_date)
#> [1] "Date"
year(wwlist$receive_date[1:5])
#> [1] 2016 2016 2016 2016 2016
```

What happens when we try to apply `year()` to an object with class = **numeric**?

```
typeof(wwlist$med_inc_zip)
class(wwlist$med_inc_zip)
year(wwlist$med_inc_zip[1:10])
```

Functions care about object **class**, not object **type**

Example: `tolower()` applies to **character** class objects

- Syntax: `tolower(x)`
- `x` is "a character vector, or an object that can be coerced to character by `as.character()`"

Most string functions are intended to apply to objects with a **character** class

- **type** = character
- **class** = character

Apply `tolower()` to object with class = **character**:

```
str(wwlist$hs_city)
#>  chr [1:268396] "Seattle" "Covington" "Everett" "Seattle" "Lake Stevens" ...
typeof(wwlist$hs_city)
#> [1] "character"
class(wwlist$hs_city)
#> [1] "character"

wwlist$hs_city[1:6]
#> [1] "Seattle"     "Covington"   "Everett"     "Seattle"     "Lake Stevens
#> [6] "Seattle"
tolower(wwlist$hs_city[1:6])
#> [1] "seattle"     "covington"   "everett"     "seattle"     "lake stevens
#> [6] "seattle"
```

Class and object-oriented programming

R is an object-oriented programming language

Definition of object oriented programming from this LINK

> *"Object-oriented programming (OOP) refers to a type of computer programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure."*

Object **class** is fundamental to object oriented programming because:

- Object class determines which functions can be applied to the object
- Object class also determines what those functions do to the object
  - E.g., a specific function might do one thing to objects of **class** A and another thing to objects of **class** B
  - What a function does to objects of different class is determined by whoever wrote the function

Many different object classes exist in R

- You can also create your own classes
  - Example: the `labelled` class is an object class created by Hadley Wickham when he created the `haven` package
- In this course we will work with classes that have been created by others

Class == factor

Recoding variable `ethn_code` from data frame `wwlist`

Let's first recode the `ethn_code` variable:

```
wwlist <- wwlist %>%
  mutate(ethn_code =
    recode(ethn_code,
      "american indian or alaska native" = "nativeam",
      "asian or native hawaiian or other pacific islander" = "api",
      "black or african american" = "black",
      "cuban" = "latinx",
      "mexican/mexican american" = "latinx",
      "not reported" = "not_reported",
      "other-2 or more" = "multirace",
      "other spanish/hispanic" = "latinx",
      "puerto rican" = "latinx",
      "white" = "white"
    )
  )

str(wwlist$ethn_code)
wwlist %>% count(ethn_code)
```

Factors

Factors are an object *class* used to display categorical data (e.g., marital status)

- A factor is an **augmented vector** built by attaching a **levels** attribute to an (atomic) integer vectors

Usually, we would prefer a categorical variable (e.g., race, school type) to be a factor variable rather than a character variable

- So far in the course I have made all categorical variables character variables because we had not introduced factors yet

Create factor version of character variable `ethn_code` using base R `factor()` function:

```
str(wwlist$ethn_code)
#>  chr [1:268396] "multirace" "white" "white" "multirace" "white" "multirace" .
class(wwlist$ethn_code)
#> [1] "character"

# create factor var; tidyverse approach
wwlist <- wwlist %>% mutate(ethn_code_fac = factor(ethn_code))
#wwlist$ethn_code_fac <- factor(wwlist$ethn_code) # base r approach

str(wwlist$ethn_code)
#>  chr [1:268396] "multirace" "white" "white" "multirace" "white" "multirace" .
str(wwlist$ethn_code_fac)
#>  Factor w/ 7 levels "api","black",..: 4 7 7 4 7 4 4 4 4 7 ...
```

Factors

Character variable `ethn_code`:

```
typeof(wwlist$ethn_code)
#> [1] "character"
class(wwlist$ethn_code)
#> [1] "character"
attributes(wwlist$ethn_code)
#> NULL
str(wwlist$ethn_code)
#>  chr [1:268396] "multirace" "white" "white" "multirace" "white" "multirace" ...
```

Factor variable `ethn_code_fac`:

```
typeof(wwlist$ethn_code_fac)
#> [1] "integer"
class(wwlist$ethn_code_fac)
#> [1] "factor"
attributes(wwlist$ethn_code_fac)
#> $levels
#> [1] "api"          "black"        "latinx"       "multirace"    "nativeam"
#> [6] "not_reported" "white"
#>
#> $class
#> [1] "factor"
str(wwlist$ethn_code_fac)
#>  Factor w/ 7 levels "api","black",..: 4 7 7 4 7 4 4 4 4 7 ...
```

Working with factor variables

Main things to note about variable `ethn_code_fac`

- **type** = integer
- **class** = factor, because the variable has a **levels** attribute
- Underlying data are integers, but the values of the **levels** attribute is what's displayed:

```
# Print first few obs of ethn_code_fac
wwlist$ethn_code_fac[1:5]
#> [1] multirace white     white     multirace white
#> Levels: api black latinx multirace nativeam not_reported white

# Print count for each category in ethn_code_fac
wwlist %>% count(ethn_code_fac)
#> # A tibble: 7 x 2
#>   ethn_code_fac      n
#>   <fct>          <int>
#> 1 api             2385
#> 2 black            563
#> 3 latinx          9245
#> 4 multirace      90584
#> 5 nativeam         202
#> 6 not_reported    5737
#> 7 white         159680
```

Working with factor variables

Apply `as.integer()` to display underlying integer values of factor variable

Investigate `as.integer()` function:

```
typeof(wwlist$ethn_code_fac)
#> [1] "integer"
class(wwlist$ethn_code_fac)
#> [1] "factor"

typeof(as.integer(wwlist$ethn_code_fac))
#> [1] "integer"
class(as.integer(wwlist$ethn_code_fac))
#> [1] "integer"
```

Display underlying integer values of variable `ethn_code_fac` :

```
wwlist %>% count(as.integer(ethn_code_fac))
#> # A tibble: 7 x 2
#>   `as.integer(ethn_code_fac)`      n
#>                         <int>  <int>
#> 1                           1   2385
#> 2                           2    563
#> 3                           3   9245
#> 4                           4  90584
#> 5                           5    202
#> 6                           6   5737
#> 7                           7 159680
```

Working with factor variables

Refer to categories of a factor (e.g., when filtering obs) using values of **levels** attribute rather than underlying values of variable

- Values of **levels** attribute for `ethn_code_fac` (output omitted)

```
attributes(wwlist$ethn_code_fac)
```

Example: Count the number of prospects in `wwlist` who identify as "white"

```
# referring to variable value; this doesn't work
wwlist %>% filter(ethn_code_fac==7) %>% count()
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1     0

#referring to value of level attribute; this works
wwlist %>% filter(ethn_code_fac=="white") %>% count()
#> # A tibble: 1 x 1
#>         n
#>     <int>
#> 1 159680
```

Working with factor variables

Example: Count the number of prospects in `wwlist` who identify as "white"

- To refer to underlying integer values, apply `as.integer()` function to factor variable

```
attributes(wwlist$ethn_code_fac)
#> $levels
#> [1] "api"          "black"        "latinx"       "multirace"    "nativeam"
#> [6] "not_reported" "white"
#>
#> $class
#> [1] "factor"
wwlist %>% filter(as.integer(ethn_code_fac)==7) %>% count
#> # A tibble: 1 x 1
#>         n
#>     <int>
#> 1 159680
```

How to identify the variable values associated with factor levels

Create a factor version of the character variable `psat_range`

```
wwlist %>% count(psat_range)
wwlist <- wwlist %>% mutate(psat_range_fac = factor(psat_range))
wwlist %>% count(psat_range_fac)
attributes(wwlist$psat_range_fac)
```

Investigate values associated with factor levels using `levels()` and `nlevels()`

```
levels(wwlist$psat_range_fac) #starts at 1
nlevels(wwlist$psat_range_fac) #7 levels total
levels(wwlist$psat_range_fac)[1:3] #prints levels 1-3
```

Once values associated with factor levels are known:

- Can filter based on underling integer values using `as.integer()`

```
wwlist %>% filter(as.integer(psat_range_fac)==4) %>% count()
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1  8348
```

- Or filter based on value of factor **levels**

```
wwlist %>% filter(psat_range=="1270-1520") %>% count()
#> # A tibble: 1 x 1
```

Creating factor variables from character variables or from integer variables

See Appendix

Factor student exercise

1. After running the code below, use `typeof()`, `class()`, `str()`, and `attributes()` functions to check the new variable `receive_year`
2. Create a factor variable from the input variable `receive_year` and name it `receive_year_fac`
3. Run the same functions (`typeof()`, `class()`, etc.) from the first question using the new variable you created
4. Get a count of `receive_year_fac`. (**hint:** you could also run this in the console to see values associated with each factor)

Run this code to create a year variable from the input variable `receive_date`:

```
# wwlist %>% glimpse()

library(lubridate) # load library if you haven't already
wwlist <- wwlist %>%
  mutate(receive_year = year(receive_date)) # create year variable with lubridat

# Check variable
wwlist %>%
  count(receive_year)

wwlist %>%
  group_by(receive_year) %>%
  count(receive_date)
```

Factor student exercise solutions

1. After running the code below, use `typeof()`, `class()`, `str()`, and `attributes()` functions to check the new variable `receive_year`

```
typeof(wwlist$receive_year)
#> [1] "double"
class(wwlist$receive_year)
#> [1] "numeric"
str(wwlist$receive_year)
#>  num [1:268396] 2016 2016 2016 2016 2016 ...
attributes(wwlist$receive_year)
#> NULL
```

Factor student exercise solutions

2. Create a factor variable from the input variable `receive_year` and name it `receive_year_fac`

```
# create factor var; tidyverse approach
wwlist <- wwlist %>%
  mutate(receive_year_fac = factor(receive_year))
```

Factor student exercise solutions

3. Run the same functions (`typeof()` , `class()` , etc.) from the first question using the new variable you created

```
typeof(wwlist$receive_year_fac)
#> [1] "integer"
class(wwlist$receive_year_fac)
#> [1] "factor"
str(wwlist$receive_year_fac)
#>  Factor w/ 3 levels "2016","2017",..: 1 1 1 1 1 1 1 1 1 1 ...
attributes(wwlist$receive_year_fac)
#> $levels
#> [1] "2016" "2017" "2018"
#>
#> $class
#> [1] "factor"
```

Factor student exercise solutions

4. Get a count of `receive_year_fac`. (**hint:** you could also run this in the console to see values associated with each factor)

```
wwlist %>%
  count(receive_year_fac)
#> # A tibble: 3 x 2
#>   receive_year_fac     n
#>   <fct>            <int>
#> 1 2016             89637
#> 2 2017             89816
#> 3 2018             88943
```

Class == labelled

Data we will use to introduce `labelled` class

High school longitudinal surveys from National Center for Education Statistics (NCES)

- Follow U.S. students from high school through college, labor market

We will be working with High School Longitudinal Study of 2009 (HSLS:09)

- Follows 9th graders from 2009
- Data collection waves
  - Base Year (2009)
  - First Follow-up (2012)
  - 2013 Update (2013)
  - High School Transcripts (2013-2014)
  - Second Follow-up (2016)

Using `haven` package to read SAS/SPSS/Stata datasets into R

`haven`, which is part of **tidyverse**, "enables R to read and write various data formats" from the following statistical packages:

- SAS
- SPSS
- Stata

When using `haven` to read data, resulting R objects have these characteristics:

- Data frames are **tibbles**, Tidyverse's preferred **class** of data frames
- Transform variables with "value labels" into the `labelled()` class
  - `labelled` is an object class, just like `factor` is an object class
  - `labelled` is an object **class** created by folks who created `haven` package
  - `labelled` and `factor` classes are both viable alternatives for categorical variables
  - Helpful description of `labelled` class HERE
- Dates and times converted to R date/time classes
- Character vectors not converted to factors

Using `haven` package to read SAS/SPSS/Stata datasets into R

Use `read_dta()` function from `haven` package to import Stata dataset into R

```
hsls <- read_dta(file="https://github.com/ozanj/rclass/raw/master/data/hsls/hsl
```

Must run this code chunk; permanently changes uppercase variable names to lowercase

```
names(hsls)
names(hsls) <- tolower(names(hsls)) # convert names to lowercase
names(hsls) # names now lowercase

str(hsls) # ugh
```

Investigate variable `s3classes` from data frame `hsls`

- Identifies whether respondent taking postsecondary classes as of 11/1/2013

```
typeof(hsls$s3classes)
class(hsls$s3classes)
str(hsls$s3classes)
```

Investigate attributes of `s3classes`

```
attributes(hsls$s3classes) # all attributes

#specific attributes: using syntax: attr(x, which, exact = FALSE)
attr(x=hsls$s3classes, which = "label") # label attribute
attr(x=hsls$s3classes, which = "labels") # labels attribute
```

What is object class = `labelled`?

Variable labels are labels attached to a specific variable (e.g., marital status) **Value labels** [in Stata] are labels attached to specific values of a variable, e.g.:

- Var value `1` attached to value label "married", `2` ="single", `3` ="divorced"

`labelled` is object class for importing vars with **value labels** from SAS/SPSS/Stata

- `labelled` object class created by `haven` package
- Characteristics of variables in R data frame with `class==labelled`:
  - Data `type` can be numeric(double) or character
  - To see `value labels` associated with each value:
    - `attr(df_name$var_name,"labels")`
    - E.g., `attr(hsls$s3classes,"labels")`

Investigate the attributes of `hsls$s3classes`

```
typeof(hsls$s3classes)
class(hsls$s3classes)
str(hsls$s3classes)
attributes(hsls$s3classes)
```

Use `attr(object_name,"attribute_name")` to refer to each attribute

```
attr(hsls$s3classes,"label")
attr(hsls$s3classes,"format.stata")
attr(hsls$s3classes,"class")
attr(hsls$s3classes,"labels")
```

`labelled` package

Purpose of the `labelled` package is to work with data imported from SPSS/Stata/SAS using the `haven` package

- `labelled` package contains functions to work with objects that have `labelled` class
- From package documentation:
  - "purpose of the `labelled` package is to provide functions to manipulate *metadata* as variable labels, value labels and defined missing values using the `labelled` class and the `label` attribute introduced in `haven` package."
- More info on the `labelled` package: LINK

Functions in `labelled` package

- Full list

Get variable and value labels

Functions to get variable labels and value labels

Get variable labels using `var_label()`

```
hsls %>% select(s3classes) %>% var_label()
#> $s3classes
#> [1] "S3 B01A Taking postsecondary classes as of Nov 1 2013"
```

Get value labels using `val_labels()`

```
hsls %>% select(s3classes) %>% val_labels()
#> $s3classes
#>                                     Missing
#>                                          -9
#>                              Unit non-response
#>                                          -8
#>                       Item legitimate skip/NA
#>                                          -7
#>                     Component not applicable
#>                                          -6
#> Item not administered: abbreviated interview
#>                                          -4
#>                                         Yes
#>                                           1
#>                                          No
#>                                           2
#>                                  Don't know
#>                                           3
```

Working with `labelled` class data

Create frequency tables with `labelled` class variables using `count()`

- Default setting is to show variable **values** not **value labels**

```
hsls %>% count(s3classes)
#> # A tibble: 5 x 2
#>                 s3classes     n
#>                 <dbl+lbl> <int>
#> 1 -9 [Missing]               59
#> 2 -8 [Unit non-response]   4945
#> 3  1 [Yes]                13477
#> 4  2 [No]                  3401
#> 5  3 [Don't know]          1621
```

To make frequency table show **value labels** add `%>% as_factor()` to pipe

- `as_factor()` is function from `haven` that converts an object to a factor

```
hsls %>% count(s3classes) %>% as_factor()
#> # A tibble: 5 x 2
#>   s3classes              n
#>   <fct>              <int>
#> 1 Missing               59
#> 2 Unit non-response   4945
#> 3 Yes                13477
#> 4 No                  3401
#> 5 Don't know          1621
```

Working with `labelled` class data

To isolate values of `labelled` class variables in `filter()` function:

- Refer to variable **value**, not the **value label**

Task

- How many observations in var `s3classes` associated with "Unit non-response"
- How many observations in var `s3classes` associated with "Yes"

General steps to follow:

1. Investigate object
2. Use `filter()` to isolate desired observations

Investigate object

```
class(hsls$s3classes)
hsls %>% select(s3classes) %>% var_label() #show variable label
hsls %>% select(s3classes) %>% val_labels() #show value label

hsls %>% count(s3classes) # freq table, values
hsls %>% count(s3classes) %>% as_factor() # freq table, value labels
```

Filter specific values

```
hsls %>% filter(s3classes==-8) %>% count() # -8 = unit non-response
hsls %>% filter(s3classes==1) %>% count() # 1 = yes
```

Set variable and value labels

Functions to set variable labels and value labels

Set variable labels using `var_label()` or `set_variable_labels()`

```
# Set one variable label
var_label(df_name$var_name) <- 'variable label'

# Set multiple variable labels
df_name <- df_name %>%
  set_variable_labels(
    var_name_1 = 'variable label 1',
    var_name_2 = 'variable label 2',
    var_name_3 = 'variable label 3'
  )
```

Set value labels using `val_label()` or `set_value_labels()`

```
# Set one value label
val_label(df_name$var_name, 'variable_value') <- 'value_label'

# Set multiple value labels
df_name <- df_name %>%
  set_value_labels(
    var_name_1 = c('value_label_1' = 'variable_value_1',
                   'value_label_2' = 'variable_value_2',
    var_name_2 = c('value_label_3' = 'variable_value_3',
                   'value_label_4' = 'variable_value_4')
  )
```

Create example data frame

```
df <- tribble(
  ~id, ~edu, ~sch,
  #--/--/----
  1, 2, 2,
  2, 1, 1,
  3, 3, 2,
  4, 4, 2,
  5, 1, 2
)
df
#> # A tibble: 5 x 3
#>      id   edu   sch
#>   <dbl> <dbl> <dbl>
#> 1     1     2     2
#> 2     2     1     1
#> 3     3     3     2
#> 4     4     4     2
#> 5     5     1     2
str(df)
#> tibble [5 x 3] (S3: tbl_df/tbl/data.frame)
#>  $ id : num [1:5] 1 2 3 4 5
#>  $ edu: num [1:5] 2 1 3 4 1
#>  $ sch: num [1:5] 2 1 2 2 2
```

Set variable labels

Use `set_variable_labels()` or `var_label()` to manually set variable labels

```
str(df$sch)
#>  num [1:5] 2 1 2 2 2
var_label(df$sch)
#> NULL

# Using set_variable_labels()
df <- df %>%
  set_variable_labels(
    id = "Unique identification number",
    edu = "Education level"
  )

# Using var_label()
var_label(df$sch) <- 'Type of school attending'

str(df$sch)
#>  num [1:5] 2 1 2 2 2
#>  - attr(*, "label")= chr "Type of school attending"
var_label(df$sch)
#> [1] "Type of school attending"
```

Set value labels

Use `set_value_labels()` or `val_label()` to manually set value labels

```
val_labels(df$sch)
#> NULL

# Using set_value_labels()
df <- df %>%
  set_value_labels(
    edu = c('High School' = 1,
            'AA degree' = 2,
            'BA degree' = 3,
            'MA or higher' = 4),
    sch = c('Private' = 1))

# Using val_label()
val_label(df$sch, 2) <- 'Public'

str(df$sch)
#>  dbl+lbl [1:5] 2, 1, 2, 2, 2
#>  @ labels: Named num [1:2] 1 2
#>   ..- attr(*, "names")= chr [1:2] "Private" "Public"
#>  @ label : chr "Type of school attending"
val_labels(df$sch)
#> Private  Public
#>       1       2
```

View the set variable and value labels

```r
# View variable and value labels using attributes()
attributes(df$sch)
#> $labels
#> Private  Public
#>       1       2
#>
#> $label
#> [1] "Type of school attending"
#>
#> $class
#> [1] "haven_labelled" "vctrs_vctr"     "double"

# View variable label
var_label(df$sch)
#> [1] "Type of school attending"
attr(df$sch, 'label')
#> [1] "Type of school attending"

# View value labels
val_labels(df$sch)
#> Private  Public
#>       1       2
attr(df$sch, 'labels')
#> Private  Public
#>       1       2
```

`labelled` student exercise

1. Get variable and value labels of the variable `s3hs` in the `hsls` data frame
2. Get a count of the variable `s3hs` showing the values and the value labels (**hint**: use `as_factor()`)
3. Get a count of the rows whose value for `s3hs` is associated with "Missing" (**hint**: use `filter()`)
4. Get a count of the rows whose value for `s3hs` is associated with "Missing" or "Unit non-response"
5. Add variable label for `pop_asian_zip` & `pop_asian_state` in data frame `wwlist`
6. Add value labels for `ethn_code` in data frame `wwlist`

`labelled` student exercise solutions

1. Get variable and value labels of the variable `s3hs` in the `hsls` data frame

```
hsls %>%
  select(s3hs) %>%
  var_label()
#> $s3hs
#> [1] "S3 B01F Attending high school or homeschool as of Nov 1 2013"

hsls %>%
  select(s3hs) %>%
  val_labels()
#> $s3hs
#>                                        Missing
#>                                             -9
#>                                Unit non-response
#>                                             -8
#>                         Item legitimate skip/NA
#>                                             -7
#>                        Component not applicable
#>                                             -6
#> Item not administered: abbreviated interview
#>                                             -4
#>                                            Yes
#>                                              1
#>                                             No
#>                                              2
#>                                      Don't know
```

`labelled` student exercise solutions

2. Get a count of the variable `s3hs` showing the values and the value labels (**hint**: use `as_factor()`)

```
hsls %>%
  count(s3hs)
#> # A tibble: 6 x 2
#>                                 s3hs     n
#>                          <dbl+lbl> <int>
#> 1 -9 [Missing]                        22
#> 2 -8 [Unit non-response]            4945
#> 3 -7 [Item legitimate skip/NA]     16770
#> 4  1 [Yes]                           624
#> 5  2 [No]                            985
#> 6  3 [Don't know]                    157

hsls %>%
  count(s3hs) %>%
  as_factor()
#> # A tibble: 6 x 2
#>   s3hs                          n
#>   <fct>                     <int>
#> 1 Missing                      22
#> 2 Unit non-response          4945
#> 3 Item legitimate skip/NA   16770
#> 4 Yes                         624
#> 5 No                          985
#> 6 Don't know                  157
```

`labelled` student exercise solutions

3. Get a count of the rows whose value for `s3hs` is associated with "Missing" (**hint**: use `filter()`)

```
hsls %>%
  filter(s3hs== -9) %>%
  count()
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1    22
```

`labelled` student exercise solutions

4. Get a count of the rows whose value for `s3hs` is associated with "Missing" or "Unit non-response"

```
hsls %>%
  filter(s3hs== -9 | s3hs== -8) %>%
  count()
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1  4967
```

`labelled` student exercise solutions

5. Add variable label for `pop_asian_zip` & `pop_asian_state` in data frame `wwlist`

```
# variable labels
wwlist %>% select(pop_asian_zip, pop_asian_state) %>% var_label()
#> $pop_asian_zip
#> NULL
#>
#> $pop_asian_state
#> NULL

# set variable labels
wwlist <- wwlist %>%
  set_variable_labels(
    pop_asian_zip = "total asian population in zip",
    pop_asian_state ="total asian population in state"
  )

# attribute of variable
attributes(wwlist$pop_asian_zip)
#> $label
#> [1] "total asian population in zip"
attributes(wwlist$pop_asian_state)
#> $label
#> [1] "total asian population in state"
```

`labelled` student exercise solutions

6. Add value labels for `ethn_code` in data frame `wwlist`

```
# count
wwlist %>% count(ethn_code)

# value labels
wwlist %>% select(ethn_code) %>% val_labels

# set value labels to ethn_code variable
wwlist <- wwlist %>%
  set_value_labels(
    ethn_code = c("asian or native hawaiian or other pacific islander" = "api",
                  "black or african american" = "black",
                  "cuban or mexican/mexican american or other spanish/hispanic
                  "other-2 or more" = "multirace",
                  "american indian or alaska native" = "nativeam",
                  "not reported" = "not_reported",
                  "white" = "white"
    )
  )
```

Comparing labelled class to factor class

Comparing `class==labelled` to `class==factor`

	`class==labelled`	`class==factor`
data type	numeric or character	integer
name of value label attribute	labels	levels
refer to data using	variable values	levels attribute

So should you work with `class==labelled` or `class==factor`?

- No right or wrong answer; this is a subjective decision
- Personally, I prefer 'labelled' class
  - Easier to control underlying variable value
  - Feels more suited to working with survey data variables, where there are usually several different values that represent different kinds of "missing" values

Converting `class==labelled` to `class==factor`

The `as_factor()` function from `haven` package converts variables with `class==labelled` to `class==factor`

- Can be used for descriptive statistics

```
hsls %>% select(s3classes) %>% count(s3classes)
hsls %>% select(s3classes) %>% count(s3classes) %>% as_factor()
```

- Can create object with some or all `labelled` vars converted to `factor`

```
hsls_f <- as_factor(hsls, only_labelled = TRUE)
```

Let's examine this object

```
glimpse(hsls_f)
hsls_f %>% select(s3classes,s3clglvl) %>% str()
typeof(hsls_f$s3classes)
class(hsls_f$s3classes)
attributes(hsls_f$s3classes)

hsls_f %>% select(s3classes) %>% var_label()
hsls_f %>% select(s3classes) %>% val_labels()
```

Working with `class==factor` data

Showing factor levels associated with a factor variable

```
hsls_f %>% count(s3classes)
#> # A tibble: 5 x 2
#>   s3classes             n
#>   <fct>             <int>
#> 1 Missing              59
#> 2 Unit non-response  4945
#> 3 Yes               13477
#> 4 No                 3401
#> 5 Don't know         1621
```

Showing variable values associated with a factor variable

```
hsls_f %>% count(as.integer(s3classes))
#> # A tibble: 5 x 2
#>   `as.integer(s3classes)`     n
#>                     <int> <int>
#> 1                       1    59
#> 2                       2  4945
#> 3                       6 13477
#> 4                       7  3401
#> 5                       8  1621
```

Working with `class==factor` data

When sub-setting observations (e.g., filtering), refer `level` attribute not variable value

```
hsls_f %>% filter(s3classes=="Yes") %>% count(s3classes)
#> # A tibble: 1 x 2
#>   s3classes     n
#>   <fct>     <int>
#> 1 Yes       13477
```

Appendix: Creating factor variables

Create factors [from string variables]

To create a factor variable from string variable:

1. Create a character vector containing underlying data
2. Create a vector containing valid levels
3. Attach levels to the data using the `factor()` function

```
# Underlying data: months my fam is born
x1 <- c("Jan", "Aug", "Apr", "Mar")
# Create vector with valid levels
month_levels <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
# Attach levels to data
x2 <- factor(x1, levels = month_levels)
```

Note how attributes differ:

```
str(x1)
#>  chr [1:4] "Jan" "Aug" "Apr" "Mar"
str(x2)
#>  Factor w/ 12 levels "Jan","Feb","Mar",..: 1 8 4 3
```

Sorting also differs:

```
sort(x1)
#> [1] "Apr" "Aug" "Jan" "Mar"
sort(x2)
#> [1] Jan Mar Apr Aug
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Create factors [from string variables]

Let's create a character version of variable `hs_state` and then turn it into a factor:

```
#wwlist %>%
#  count(hs_state)
# Subset obs to West Coast states
wwlist_temp <- wwlist %>%
  filter(hs_state %in% c("CA", "OR", "WA"))

# Create character version of high school state for West Coast states only
wwlist_temp$hs_state_char <- as.character(wwlist_temp$hs_state)

# Investigate character variable
str(wwlist_temp$hs_state_char)
table(wwlist_temp$hs_state_char)

# Create new variable that assigns levels
wwlist_temp$hs_state_fac <- factor(wwlist_temp$hs_state_char, levels = c("CA","
str(wwlist_temp$hs_state_fac)
attributes(wwlist_temp$hs_state_fac)

#wwlist_temp %>%
#  count(hs_state_fac)
rm(wwlist_temp)
```

Create factors [from string variables]

How the `levels` argument works when underlying data is character:

- Matches value of underlying data to value of the level attribute
- Converts underlying data to integer, with level attribute attached

See Chapter 15 of Wickham for more on factors (e.g., modifying factor order, modifying factor levels)

Creating factors [from integer vectors]

Factors are just integer vectors with level attributes attached to them. So, to create a factor:

1. Create a vector for the underlying data
2. Create a vector that has level attributes
3. Attach levels to the data using the `factor()` function

```
a1 <- c(1,1,1,0,1,1,0) # A vector of data
a2 <- c("zero","one") # A vector of labels

# Attach labels to values
a3 <- factor(a1, labels = a2)
a3
#> [1] one  one  one  zero one  one  zero
#> Levels: zero one
str(a3)
#>  Factor w/ 2 levels "zero","one": 2 2 2 1 2 2 1
```

Note: By default, `factor()` function attached "zero" to the lowest value of vector `a1` because "zero" was the first element of vector `a2`

Creating factors [from integer vectors]

Let's turn an integer variable into a factor variable in the `wwlist` data frame

Create integer version of `receive_year`:

```
#typeof(wwlist_temp$receive_year)
wwlist$receive_year_int <- as.integer(wwlist$receive_year)
str(wwlist$receive_year_int)
#>  int [1:268396] 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 ...
typeof(wwlist$receive_year_int)
#> [1] "integer"
```

Assign levels to values of integer variable:

```
wwlist$receive_year_fac <- factor(wwlist$receive_year_int,
      labels=c("Twenty-sixteen","Twenty-seventeen","Twenty-eighteen"))
str(wwlist$receive_year_fac)
str(wwlist$receive_year)

#Check variable
wwlist %>%
  count(receive_year_fac)

wwlist %>%
  count(receive_year)
```