Data Processing

R for Stata Users

Luiza Andrade, Leonardo Viotti & Rob Marty

May 2019

Outline

Introduction

- The goal of this session is to recreate the WHR data set that we've been using for this training
- We'll take you through the same steps we've taken when we were preparing it
- We'll use a set of packages that are bundled into something called the `tidyverse`

Introduction

- In this session, you'll be introduced to some basic concepts of data cleaning in R. The contents covered are:
  - Exploring a data set
  - Creating new variables
  - Filtering and subsetting data sets
  - Merging data sets
  - Dealing with factor variables
  - Saving data
- There are many other tasks that we usually perform as part of data cleaning that are beyond the scope of this session

Introduction

Before we start, let's make sure we're all set:

1. Start a fresh session.
2. Load the `tidyverse` package.
3. Set your file paths.

Introduction

Here's a how you can do that:

```
# Install packages
install.packages("tidyverse",
                 dependencies = TRUE)

# Load packages
library(tidyverse)

#  Set folder paths
projectFolder <- "YOUR/FOLDER/PATH"
finalData     <- file.path(projectFolder,
                           "DataWork", "DataSets", "Final")
rawData       <- file.path(projectFolder,
                           "DataWork", "DataSets", "Raw")
```

Loading a data set from CSV

`read.csv(file, header = FALSE)`

- **file**: is the path to the file you want to open, including it's name and format (`.csv`)
- **header**: if `TRUE`, will read the first row as variable names
- **stringsAsFactors:** logical. See next slide for more.

Loading a data set from CSV

- R reads string variables as factors as default
- This format saves memory, but can be tricky if you actually want to use the variables as strings
- You can specify the option `stringsAsFactors = FALSE` to prevent R from turning strings into factors

Loading a data set from CSV

Exercise 1: Load data

Use the `read.csv` function to load the three `WHR` data sets from `DataWork > DataSets > Raw`. Create an object called `whrYY` with each data set.

- TIP 1: use the `file.path()` function to simplify the folder path.
- TIP 2: for this data set, we want to read strings as strings, not factors.

Loading a data set from CSV

```
# Load the data sets (we'll discuss why the
# stringsAsFactors argument is necessary soon)
whr15 <- read.csv(file.path(rawData,"WHR2015.csv"),
                  header = TRUE,
                  stringsAsFactors = FALSE)

whr16 <- read.csv(file.path(rawData,"WHR2016.csv"),
                  header = TRUE,
                  stringsAsFactors = FALSE)

whr17 <- read.csv(file.path(rawData,"WHR2017.csv"),
                  header = TRUE,
                  stringsAsFactors = FALSE)
```

Outline

Exploring a data set

Some useful functions:

- `View()`: open the data set
- `class()`: reports object type or type of data stored
- `dim()`: reports the size of each one of an object's dimension
- `names()`: returns the variable names of a data set
- `str()`: general information on an R object
- `summary()`: summary information about the variables in a data frame
- `head()`: shows the first few observations in the dataset
- `tail()`: shows the last few observations in the dataset

Outline

ID variables

Desired properties of an ID variable: *uniquely and fully identifying*

- An ID variable cannot have duplicates
- An ID variable may never be missing
- The ID variable must be constant across a project
- The ID variable must be anonymous

ID variables

`n_distinct(..., na.rm = FALSE)`

Counts the number of unique values of a variablelength of a vector

- `...`: a vector of values
- **na.rm**: if `TRUE`, missing values don't count

Exercise 2: identify the ID

Using the `n_distinct` function, can you tell if the following variables are IDs of the `whr15` data set?

1. Region
2. Country

ID variables

```
dim(whr15)
```

```
## [1] 158  12
```

```
n_distinct(whr15$Region, na.rm = TRUE)
```

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

```
n_distinct(whr15$Country, na.rm = TRUE)
```

```
## [1] 158
```

ID variables

We did the same for the other two data sets:

```
n_distinct(whr16$Country, na.rm = TRUE) == nrow(whr16)
```

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

```
n_distinct(whr17$Country, na.rm = TRUE) == nrow(whr17)
```

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

ID variables

- The data set we've been using in the last few sessions combines all three data sets we have now
- Before we combine them, we should take a better look at the ID variables to find out if they are consistent

Comparing vectors

`setdiff(first, second)`

Prints all the elements of the `first` object that are not in the `second` object (ignores duplicates).

- **first:** an object
- **second:** an (other?) object

Comparing vectors

Exercise 3: compare vectors

Use the `setdiff()` function to see which countries are coming in and out of the WHR data set between 2015 and 2016.

Comparing vectors

```
# Any countries in 2015 that are not in 2016?
setdiff(whr15$Country, whr16$Country)
```

```
## [1] "Oman"                          "Somaliland region"
## [3] "Mozambique"                    "Lesotho"
## [5] "Swaziland"                     "Djibouti"
## [7] "Central African Republic"
```

```
# And vice-versa
setdiff(whr16$Country, whr15$Country)
```

```
## [1] "Puerto Rico"       "Belize"              "Somalia"
## [4] "Somaliland Region" "Namibia"             "South Sudan"
```

Replacing values

Wait, “Somaliland region” and “Somaliland Region” are not the same?!

Exercise 4: replacing values

Replace the occurrences of “Somaliland region” in whr15 with “Somaliland **R**egion”.

- TIP: use indexing to select only the observations of whr15$Country that are equal to “Somaliland region”

Replacing values

Wait, “Somaliland region” and “Somaliland Region” are not the same?!

```
# Now they are:
whr15$Country[whr15$Country == "Somaliland region"] <-
  "Somaliland Region"
```

Creating variables

- Ok, the ID variables are consistent now
- But once we merge the data sets, we need to still be able to identify them
- So let's add a `year` variable so we can tell them apart

Exercise 5: creating variables

Create a variable called `year` in each WHR data set identifying what year it refers to.

Creating variables

```
# Piece of cake!
whr15$year <- 2015
whr16$year <- 2016
whr17$year <- 2017
```

Outline

Appending data sets

- Now that we can identify the observations, we can combine the data set
- Here's a function to append objects by row:

`rbind(...)`

Take a sequence of vector, matrix or data-frame arguments and combine by rows.

- **`...`**: vectors or matrices to be combined (separated by comma)

Appending and merging

Exercise 6: append data sets

Use the `rbind` function to append the three WHR datasets into a data set called `whr_panel`.

Appending and merging data sets

```
# Append data sets
whr_panel <- rbind(whr15, whr16, whr17)
```

```
Error in rbind(deparse.level, ...)  :
numbers of columns of arguments do not match
```

Appending data sets

- Our data sets are still too different to use this function, as it has very strong requirements
- There's a number of ways to fix this, and we will explore them soon
- But first, here's how we could append the data as is if we wanted to

```
# This is the quick fix
whr_panel <- bind_rows(whr15, whr16, whr17)
```

- Now, let's take a closer look at the variables in our data sets and see how we can make them compatible

Exploring a data set (again)

```
names(whr15)
```

```
##  [1] "Country"                          "Region"
##  [3] "Happiness.Rank"                   "Happiness.Score"
##  [5] "Standard.Error"                   "Economy..GDP.per.Capita."
##  [7] "Family"                           "Health..Life.Expectancy."
##  [9] "Freedom"                          "Trust..Government.Corruption."
## [11] "Generosity"                       "Dystopia.Residual"
## [13] "year"
```

```
names(whr16)
```

```
##  [1] "Country"                          "Region"
##  [3] "Happiness.Rank"                   "Happiness.Score"
##  [5] "Lower.Confidence.Interval"        "Upper.Confidence.Interval"
##  [7] "Economy..GDP.per.Capita."         "Family"
##  [9] "Health..Life.Expectancy."         "Freedom"
## [11] "Trust..Government.Corruption."    "Generosity"
## [13] "Dystopia.Residual"                "year"
```

```
names(whr17)
```

```
##  [1] "Country"                          "Happiness.Rank"
##  [3] "Happiness.Score"                  "Whisker.high"
##  [5] "Whisker.low"                      "Economy..GDP.per.Capita."
##  [7] "Family"                           "Health..Life.Expectancy."
##  [9] "Freedom"                          "Generosity"
## [11] "Trust..Government.Corruption."    "Dystopia.Residual"
## [13] "year"
```

Exploring a data set (again)

There are a few issues here:

1. The data set for 2017 doesn't include a region identifier
2. The names for the same variables are different in 2017 and 2016 (and the variable names are terrible in general)
3. The data for 2015 only includes the standard error, not the confidence interval

Subsetting

To fix the first issue, we will merge the region variable from 2016 to the 2017 data set. But first, we need to isolate the variables we actually want to merge to 2017. To do this, we'll use our first `tidyverse` function:

`select(.data, ...)`

Keeps only the variables you mention.

- **.data:** a data set
- **...:** one or more unquoted expressions separated by commas indicating the names of the variables you want to keep

Subsetting

Exercise 7: subset the data

Create a new object called `regions` containing only the columns `Country` and `Region` of the `whr16` data set.

Subsetting

```
# Subset the whr 16 data set
regions <- select(whr16, Country, Region)

# Here's what the new dataset looks like
str(regions)
```

```
## 'data.frame':    157 obs. of  2 variables:
##  $ Country: chr  "Denmark" "Switzerland" "Iceland" "Norway"
##  $ Region : chr  "Western Europe" "Western Europe" "Western
```

Subsetting

```r
# This also works with a vector of variables
keepVars <-   c("Country",
                "Region",
                "year",
                "Happiness.Rank",
                "Happiness.Score",
                "Economy..GDP.per.Capita.",
                "Family",
                "Health..Life.Expectancy.",
                "Freedom",
                "Trust..Government.Corruption.",
                "Generosity",
                "Dystopia.Residual")

whr15 <- whr15[, keepVars]
whr16 <- select(whr16, keepVars)
```

Tidyverse

- As you might have noticed, some `tidyverse` functions have a syntax that is a little bit different from most R functions
- Its first argument is the name of a data set
- The following arguments are variable names
- You don't need to write variable names in quotes
- You don't need to name your arguments (and it will break if they are in the wrong order)

Merging

- The `tidyverse` package `dplyr` has a whole family of functions to do merging. You can look them up by typing `?join`
- The different `join` functions have a similar function to the `keep` and `keepusing` options of Stata's `merge` function
- We now want to merge the values in the `regions` object to the `whr17` object, keeping all observations in `whr17`, regardless of them having matches or not

Merging

`left_join(x, y, by)`

Return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

- **x, y:** data sets to join
- **by:** a character vector of variables to join by. If NULL, the default, *_join() will do a natural join, using all variables with common names across the two tables.

Merging

Exercise 8: Merge

Merge the `regions` data set into the `whr17` data set.

Merging

```
# Merge
whr17 <- left_join(whr17, regions)
```

```
## Joining, by = "Country"
```

```
# See the result
str(whr17)
```

```
## 'data.frame':    155 obs. of  14 variables:
##  $ Country                      : chr  "Norway" "Denmark" "Iceland" "Switzerland" ...
##  $ Happiness.Rank               : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Happiness.Score              : num  7.54 7.52 7.5 7.49 7.47 ...
##  $ Whisker.high                 : num  7.59 7.58 7.62 7.56 7.53 ...
##  $ Whisker.low                  : num  7.48 7.46 7.39 7.43 7.41 ...
##  $ Economy..GDP.per.Capita.     : num  1.62 1.48 1.48 1.56 1.44 ...
##  $ Family                       : num  1.53 1.55 1.61 1.52 1.54 ...
##  $ Health..Life.Expectancy.     : num  0.797 0.793 0.834 0.858 0.809 ...
##  $ Freedom                      : num  0.635 0.626 0.627 0.62 0.618 ...
##  $ Generosity                   : num  0.362 0.355 0.476 0.291 0.245 ...
##  $ Trust..Government.Corruption.: num  0.316 0.401 0.154 0.367 0.383 ...
##  $ Dystopia.Residual            : num  2.28 2.31 2.32 2.28 2.43 ...
##  $ year                         : num  2017 2017 2017 2017 2017 ...
##  $ Region                       : chr  "Western Europe" "Western Europe" "Western Europe" "Western Europe" .
```

Missing values

```
# Did that solve it?
any(is.na(whr17$Region))
```

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

```
sum(is.na(whr17$Region))
```

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

```
# Where is it still missing?
whr17$Country[is.na(whr17$Region)]
```

```
## [1] "Taiwan Province of China" "Hong Kong S.A.R., China"
## [3] "Mozambique"               "Lesotho"
## [5] "Central African Republic"
```

```
# Let's fix that
whr17$Region[whr17$Country %in% c("Mozambique",
                                  "Lesotho",
                                  "Central African Republic")] <-
  "Sub-Saharan Africa"

# That's better
any(is.na(whr17$Region))
```

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

Missing values

Unlike in Stata, R never[1] treats missings as zeros by default in any function.

- If your vector (column or row in your dataset) has at least one `NA`, any function that takes it as an argument will return `NA`.
- If you wish to treat missings as zeros or ignore them, you need to explicitly do it.
- E.g. `mean(myVector, na.rm = T)` or `rowSums(myDataFrame, na.rm = T)`

[1] Of course, there might be an obscure package with a function that does this or you can write your on function. But base R and all the major packages don't and we never came across any function on CRAN that does.

Renaming variables

The second problem we found, of different names for the same variable in different data sets, can be easily fixed with the rename function:

```
# Rename function: new name on the left!
whr17 <- rename(whr17,
                Lower.Confidence.Interval = Whisker.low,
                Upper.Confidence.Interval = Whisker.high)
```

Renaming

- If you need to change the name of variables in bulk, setting the whole vector of variable names would be easier, as in the example below
- However, to do that you need to be sure that the variables are in the right order!

Renaming

```
# Bulk rename
names(whr15)
```

```
##  [1] "Country"                    "Region"
##  [3] "year"                       "Happiness.Rank"
##  [5] "Happiness.Score"            "Economy..GDP.per.Capita."
##  [7] "Family"                     "Health..Life.Expectancy."
##  [9] "Freedom"                    "Trust..Government.Corruption."
## [11] "Generosity"                 "Dystopia.Residual"
```

```
newnames <-  c("country",
               "region",
               "year",
               "happy_rank",
               "happy_score",
               "gdp_pc",
               "family",
               "health",
               "freedom",
               "trust_gov_corr",
               "generosity",
               "dystopia_res")

names(whr15) <-  newnames
```

Ordering variables

Fortunately, `select` also reorders variables

```
# Subset the 2017 data set and order variables
whr17 <- select(whr17, keepVars)

# Rename variables
names(whr16) <-  newnames
names(whr17) <-  newnames

# Now we can append safely
whr_panel <- rbind(whr15, whr16, whr17)
```

Outline

Saving a data set to csv

- The data set you have now is the same data set we've been using for earlier sessions, so we can save it now
- As mentioned before, R data sets are often save as csv
- To save a data set, we use the `write.csv()` function:

```
write.csv(x, file, row.names = TRUE)
```

- **`x`:** the object (usually a data frame) you want to export to CSV
- **`file`:** the file path to where you want to save it, including the file name and the format (".csv")
- **`row.names`**: by default, R adds a column to the CSV file with the names (or numbers) of the rows in the data frame. Set it to `FALSE` if you don't want that column to be exported

Saving a data set to csv

Exercise 9: save the data set

Save the `whr_panel` data set to DataWork > DataSets > Final.

- TIP: Use the `file.path()` function and the object `finalData` created in the master to simplify the folder path.

Saving a data set to csv

```
# Save the whr data set
write.csv(whr_panel,
          file.path(finalData,"whr_panel.csv"),
          row.names = F)
```

Saving a data set as R data

- The problem with CSVs is that they cannot differentiate between strings and factors
- They also don't save factor orders
- Data attributes (which are beyong the scope of this training, but also useful to document data sets) are also lost in csv data

Saving a data set as R data

The R equivalent of a `.dta` file is a `.Rda` file. It can be saved and loaded using the following commands:

`saveRDS(object, file = "")`

Writes a single R object to a file.

- **object:** the R object to be save
- **file:** the file path to where it should be saved

`readRDS(file)`

Load a single R object from a file.

- **file:** the file path to the data set

Saving a data set as R data

```
# Save the data set
saveRDS(whr_panel,
        file = file.path(finalData,
                         "whr_panel.Rda"))

# And load it again
whr_panel <-
  readRDS(file.path(finalData,
                    "whr_panel.Rda"))
```

Saving a data set to Stata

`write.dta13(data, file)`

Part of the `readstata13` package. Writes a Stata dta-file bytewise and saves the data into a dta-file.

- **data:** A data frame
- **file:** Path to the dta file you want to export

Saving a data set to Stata

```
# This command doesn't handle ordered factors well
whr_panel$region_ord <- NULL

# Export it
save.dta13(whr_panel,
           file = file.path(finalData,
                            "whr_panel.dta"))

# Load it again
whr_panel <- read.dta13(file.path(finalData,
                                  "whr_panel.dta"))
```

Outline

Creating variables based on a formula

- When we created the `year` variable, we just assigned a value to a column in the data frames using `$`
- This is great for simple variables, but can get tricky if it takes more complex values
- For example, to create a dummy variable that shows if the happy score is above the median, we would write the following

```
# Using $
whr_panel$happy_high <-
  whr_panel$happy_score > median(whr_panel$happy_score)
```

- The `tidyverse` function `mutate` make this process simpler

Creating variables base on a formula

`mutate(.data, ...)`

Adds new variables and preserves existing

- **.data:** the data set you want to add a variable to
- **...:** name-value pairs of expressions. Use NULL to drop a variable

Creating variables base on a formula

Exercise 10: Create a variable based on a formula

Use the `mutate` function to create a variable called `happy_high` in the `whr_panel` data set indicating whether the `happy_score` is above the median.

- TIP: as usual in `tidyverse`, you can refer to variables by their names, without quotes or `$`

Creating variables based on a formula

```
# Adding a new variable
whr_panel <-
  mutate(whr_panel,
         happy_high = happy_score > median(happy_score))
```

Creating variables based on a formula

```
# You can do this for multiple variables at a time
whr_panel <-
  mutate(whr_panel,
         happy_high = happy_score > median(happy_score),
         happy_low = happy_score < median(happy_score),
         dystopia_res = NULL) # NULL drops the variable
```

Creating factor variables

- When we imported this data set, we told R explicitly to not read strings as factor
- We did that because we knew that we'd have to fix the country names
- The region variable, however, should be a factor:

```
str(whr_panel$region)
```

```
##  chr [1:470] "Western Europe" "Western Europe" "Western Europe" ...
```

```
unique(whr_panel$region)
```

```
##  [1] "Western Europe"                "North America"
##  [3] "Australia and New Zealand"     "Middle East and Northern Africa"
##  [5] "Latin America and Caribbean"   "Southeastern Asia"
##  [7] "Central and Eastern Europe"    "Eastern Asia"
##  [9] "Sub-Saharan Africa"            "Southern Asia"
## [11] ""
```

Creating a factor

To create a factor variable, we use the `factor` function:

`factor(x, levels, labels)` : turns numeric or string vector `x` into a factor vector

- **`x`:** the vector you want to turn into a factor
- **`levels`:** a vector containing the possible values of `x`
- **`labels`:** a vector of strings containing the labels you want to apply to your factor variable
- **`ordered`:** logical flag to determine if the levels should be regarded as ordered (in the order given).

Converting strings into factors

If your categorical variable does not need to be ordered, and your string variable already has the label you want, making the conversion is quite easy.

Exercise 11: turn a string variable into a factor

Use the `mutate` function to create a variable called `region_cat` containing a categorical version of the `region` variable.

- TIP: to do this, you only need the first argument of the `factor` function

Converting strings into factors

```
# Create categorical region
whr_panel <-
  mutate(whr_panel,
         region_cat = factor(region))


class(whr_panel$region_cat)
```

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

```
table(whr_panel$region_cat)
```

```
##
##                                 Australia and New Zealand
##                                 2                         6
##      Central and Eastern Europe                  Eastern Asia
##                                87                        16
##     Latin America and Caribbean Middle East and Northern Africa
##                                68                        58
##                   North America              Southeastern Asia
##                                6                         26
##                   Southern Asia             Sub-Saharan Africa
##                                21                        117
##                  Western Europe
##                                63
```

Ordering factors

```r
# If you want your levels to be shown in a particular order, you need to order them
whr_panel <-
  mutate(whr_panel,
         region_ord = factor(region,
                             levels = c("Latin America and Caribbean",
                                        "North America",
                                        "Western Europe",
                                        "Central and Eastern Europe",
                                        "Middle East and Northern Africa",
                                        "Sub-Saharan Africa",
                                        "Eastern Asia",
                                        "Southeastern Asia",
                                        "Southern Asia",
                                        "Australia and New Zealand"),
                             ordered = TRUE))

class(whr_panel$region_ord)
```

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

```r
table(whr_panel$region_ord)
```

```
##
##     Latin America and Caribbean                   North America
##                              68                               6
##                  Western Europe       Central and Eastern Europe
##                              63                              87
## Middle East and Northern Africa              Sub-Saharan Africa
##                              58                             117
##                     Eastern Asia               Southeastern Asia
##                              16                              26
##                     Southern Asia        Australia and New Zealand
##                              21                               6
```

Labelling values

The `labels` argument of the `factor` function can be used to assign labels to specific values

```
# Assign 'not so happy'  and 'happy' as labels
# for above of below median happy score, respectively
whr_panel <-
  mutate(whr_panel,
         happy_cat = factor(happy_high,
                            levels = c(TRUE,
                                       FALSE),
                            labels = c("Happy",
                                       "Not so happy")))

table(whr_panel$happy_cat)
```

```
##
##         Happy Not so happy
##           235           235
```

Outline

Outline

spread and gather

- These are the `tidyverse` functions to reshape data
- We've `spread` in Lab 2 to create a table of the happy score by year

```
# Spread data
happy_table <-
  spread(select(whr_panel,
                country, year, happy_score), # data
         key = year,                         # new variables' names
         value = happy_score)                # var to populate the cells

# See result
head(happy_table)
```

```
##       country  2015  2016  2017
## 1 Afghanistan 3.575 3.360 3.794
## 2     Albania 4.959 4.655 4.644
## 3     Algeria 5.605 6.355 5.872
## 4      Angola 4.033 3.866 3.795
## 5   Argentina 6.574 6.650 6.599
## 6     Armenia 4.350 4.360 4.376
```

spread and gather

If we wanted to make the resulting data set long again, we'd use `gather`:

```
# Gather the happy_table
happy_table_long <-
  gather(happy_table,
         key = year,
         value = happy_score,
         `2015`, `2016`, `2017`) # name of columns that will be gathered

# See result
head(happy_table_long)
```

```
##       country year happy_score
## 1 Afghanistan 2015       3.575
## 2     Albania 2015       4.959
## 3     Algeria 2015       5.605
## 4      Angola 2015       4.033
## 5   Argentina 2015       6.574
## 6     Armenia 2015       4.350
```

dcast and melt

- These are functions from the `reshape2` package

```
install.packages("reshape2")
library(reshape2)
```

dcast: reshape from long to wide

dcast(data, formula, value.var)

- **data:** a **data.table**
- **formula:** a formula of the format `LHS ~ RHS`, where LHS is the unique ID in the final data set and `RHS` are the `j` variables in Stata's `reshape`
- **value.var:** name of the columns to be reshaped

dcast: reshape from long to wide

```
library(data.table)
```

```
# Making the data long
happy_long <-
  dcast(setDT(whr_panel),
        country ~ year,
        value.var = c("happy_score",
                      "happy_rank"))

# See result
head(happy_long)
```

```
##        country happy_score_2015 happy_score_2016 happy_score_2017
## 1: Afghanistan            3.575            3.360            3.794
## 2:     Albania            4.959            4.655            4.644
## 3:     Algeria            5.605            6.355            5.872
## 4:      Angola            4.033            3.866            3.795
## 5:   Argentina            6.574            6.650            6.599
## 6:     Armenia            4.350            4.360            4.376
##    happy_rank_2015 happy_rank_2016 happy_rank_2017
## 1:             153             154             141
## 2:              95             109             109
## 3:              68              38              53
## 4:             137             141             140
## 5:              30              26              24
## 6:             127             121             121
```

`melt`: reshape from wide to long

‘melt(data, id.vars, measure.vars)

- **data:** a **data.table** object to melt
- **id.vars:** a vector of unique IDs in `data`
- **measure.vars:** a list of variables to melt
- **variable.name:** the name of the new ID var (the one that was in wide)
- **value.name:** a vector of names for the reshaped variables

melt: reshape from wide to long

```
# Reshape
happy_wide <-
  melt(happy_long,
       id.vars = "country",
       measure.vars = patterns("^happy_score", "^happy_rank"),
       variable.name = "year",
       value.name = c("happy_score", "happy_rank"))

# See result
head(happy_wide)
```

```
##         country year happy_score happy_rank
## 1: Afghanistan    1       3.575        153
## 2:     Albania    1       4.959         95
## 3:     Algeria    1       5.605         68
## 4:      Angola    1       4.033        137
## 5:   Argentina    1       6.574         30
## 6:     Armenia    1       4.350        127
```