Basic Programming: For Loops and If Statements

DRY vs. WET

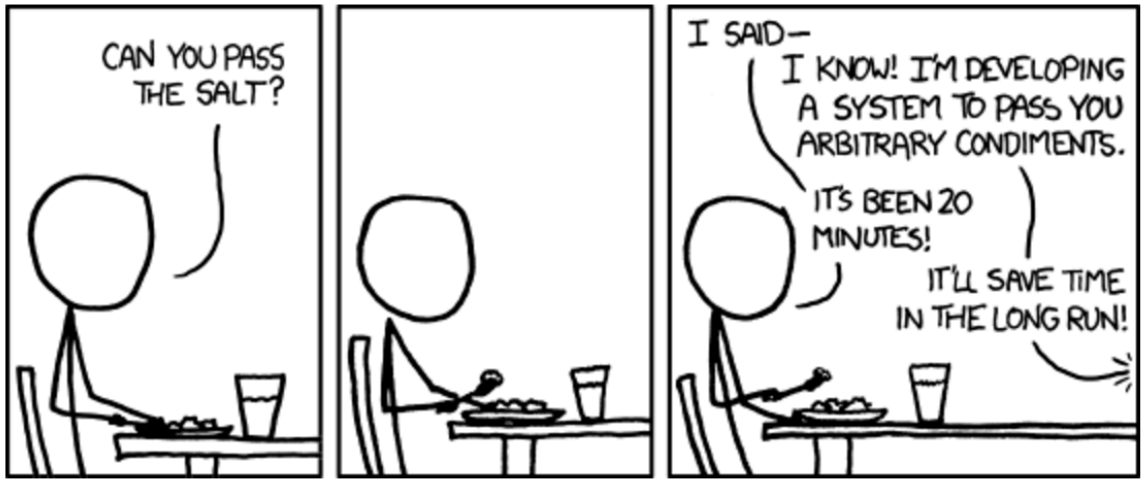
- In programming, there is a DRY credo: **D**on't **R**epeat **Y**ourself
- Why not?
  - Humans are error-prone
  - Believe it or not, computers are not! The errors they return are caused by humans making mistakes
  - Once a human has invested the abstract problem-solving to solve a task, it makes sense to generalize

WET

- The opposite of DRY is WET:
  - **W**rite **E**verything **T**wice
  - **W**e **E**njoy **T**yping
  - **W**aste **E**veryone's **T**ime
- This is *probably* too harsh… it's important not to get paralyzed by finding an optimal solution

WET

- The opposite of DRY is WET:
  - **W**rite **E**verything **T**wice
  - **W**e **E**njoy **T**yping
  - **W**aste **E**veryone's **T**ime
- This is *probably* too harsh… it's important not to get paralyzed by finding an optimal solution



For Loops

- For loops are staples of every programming language
- They allow the user to repeat the same action on a list of objects or using a list of values
- There are two commands for looping in Stata:

`foreach` and **`forvalues`**

- We will take a brief look at **`forvalues`** to practice basic looping and then return to foreach for more in-depth examples

Looping Rules

- Stata has a very specific format for for loops:

```
forvalues i = 1/3 { //only comments allowed
    commands ...
}
```

1. The open brace must appear on the same line as the **`forvalues`** or **`foreach`**
2. Nothing besides comments can appear after the open brace
3. The closing brace must appear on a line by itself

forvalues

- What happens when we run this loop?

```
forvalues i = 1/3 { //only comments allowed
    display “Loop Number `i'”
}
```

- The loop will run 3 times, iterating over our values (1, 2, 3) and **display**-ing “Loop Number 1”, “Loop Number 2”, and “Loop Number 3” by placing each value of the

- Notice the Stata iterator is passed into the body of the loop with a very particular method: **` followed by '**

  - ` is a “grave accent” but is often called a “backtick” or “backquote”

  - ' is an “apostrophe” or "single quote"

`forvalues` / `foreach`

- `forvalues` is specifically made to iterate over numbers
- It is optimized to be faster than foreach at numbered tasks, but this difference is not substantial for the average user
- `foreach` is a more general looping operator, but has a slightly more complicated syntax
- `foreach` can flexibly iterate over lists of numbers or variables

foreach

```
foreach name of listtype {
    commands . . .
}
```

- The important addition here is the **`listtype`** argument
- This will define the type of iterator we use:
  - `numlist` - list of numbers
  - `varlist` - list of variables
  - `newlist` - list of new variables
- Like **`forvalues`**, we can insert each element of our list using **`` `name' `` (backtick + apostrophe)**

foreach - numlist

```
foreach i of numlist 1/7 {
   generate random_variable_`i' = rnormal()
}
```

- In this example, we are creating seven new variables whose values are random pulls from a normal distribution
- This same loop could be made using a **forvalues** call:

```
forvalues j = 1/7 {
   generate random_variable_`i' = rnormal()
}
```

- However, foreach has the flexibility to also iterate over variable lists

foreach - varlist

```
foreach k of varlist make trunk-length {
   ameans `k'
}
```

- In this case, we are iterating over a list of variables and calling their means: `make` and all the variables from `trunk` and `length`
- Using varlist notation like this in scripts is powerful, but depends on the order of your variables
- You can also use the `*` symbol to iterate over all variables

foreach - newlist

```
foreach newvar of newlist r_num_1 - r_num_5 {
  generate `newvar' = rpoisson(5)
}
```

- In this case, we are iterating over a list of new variables (`r_num_1`, `r_num_2`, `r_num_3`, `r_num_4`, `r_num_5`) and creating observations based on a random pull from a poisson distribution
- Stata conveniently creates these numbered variables for the user with this **- (hyphen)** notation

foreach in

- Finally foreach can be used with `in` to create a shorter list made up of anything the user wishes
- This is useful for iterating over a short number of items, but the added features (like using **-** or **/**) from specifying the type of list (`varlist` or `numlist`) are lost

```
foreach file in autoexpense.dta autosize.dta{
        use `file', clear
        notes: Checked by Cale on 07/26/17
        save `file', replace
  }
```

`If/else` statements

- An `if` statement in a script is different than the `if` qualifier we have used so far
- The purpose of `if/else` statements is to execute code when certain conditions are satisfied (sometimes referred to as “control flow”)
- Often these statements are used inside of for loops to allow a single loop to behave differently based on inputs

`If/else` rules

- Stata has a very specific format for for if/else statements that will be very familiar:

```
if expression { //only comments allowed
    commands
}
```

1. The open brace must appear on the same line as the `if` or `else`
2. Nothing besides comments can appear after the open brace
3. The closing brace must appear on a line by itself

If/else

- The commands in the body of the `if` statement will only execute if the `expression` evaluates to true (1)
- When the `expression` is anything besides true (1) the body of the `else` statement will execute

```
foreach i of numlist 1/7 {
   if `i' == 4 {
        display “`i' is the best number”
   }
   else {
        display “`i' is a terrible number”
  }
}
```

If/else

- For loops and if/else clauses can also be used to iterate over variables and perform different commands depending on the variable

```
foreach var of varlist headroom trunk weight {
   if "`var'" == "trunk" {
        display “`var' summarize results below”
        summarize `var'
   }
   else {
        display “`var' codebook results below”
        codebook `var'
   }
}
```

Important use of " "

- Note this element of the `if` clause in the previous slide:

```
if "`var'" == "trunk" {
      display “`var' summarize results below”
      summarize `var'
}
```

- Without the quotation marks around "`**var**'" and "**trunk**", Stata would have checked to see if the *first observation* of our iterating variable and `trunk` were the same

- Therefore, when you want to write an `if / else` statement that checks if the iterator is a particular variable in the dataset, you can think of it as checking whether the ***names*** of the variables are equal

  - To do this, use quotation marks around both:

```
if "`var'" == "trunk"
```

Nested `for` loops

- For loops can also be nested for more advanced behavior:

```
foreach num of numlist 1/3 {
  use auto, clear
  sample `num'
  foreach var of varlist make mpg {
      list `var'
    }
}
```

- What do these for loops do?

Exercises (1)

1. Auto Data

   A. Create a **for loop** in your auto do-file which separately summarizes every variable except for make.

   B. Create a **for loop** in your auto do-file which creates three scatter plots. Price should be on the y-axis in all three, but the x-axis should differ each time: mpg, weight, turn.

   C. Create a **for loop** in your auto do-file which subsets the data based on the values of our price categorical variable and saves these subsets as three separate files.

Exercises (2)

1. For Loop Practice (create a new forloop.do)

   A. Create a for loop that opens our three practice files (auto, titanic, and movies) and shows their notes.

   B. Create a for loop that displays the results of the 8 times tables (8,16,24 etc. up to 8 * 25).

   C. Create five copies of our titanic data file, naming them titanic1, titanic2, etc. However, skip titanic3!

Hypothesis Testing and Stored Results

Hypothesis Testing

- One of the most common exercises for new users to statistical software is to perform basic hypothesis testing
- Stata has very easy to use commands to perform t-tests, anovas, linear regressions, and logistic regressions
- Let's begin with some t-tests concerning the mean price of cars from our `auto` data

ttest

```
ttest cont_variable == value
```

- This command performs a one-sample t-test for whether the mean of a sample continuous variable (`cont_variable`) is equal to a fixed `value`

```
ttest cont_variable, by(dich_variable)
```

- This command performs a two-sample t-test for the equality of means of a (`cont_variable`) across two values of the dichotomous variable (`dich_variable`)

- There are many other options available via **`help ttest`**

ttest

```
ttest cont_variable == value
```

- Let's check example output from the following:

```
sysuse auto
```

```
ttest mpg == 30
```

and

```
ttest mpg, by(foreign)
```

Stored Results

- As we just saw with **`ttest`** Stata presents a lot of output information in response to a statistical command
- Often the user will want to use some aspect of these results later on in a script
- The *wrong way* to do this is to manually write down the result we want
- Luckily, Stata saves the outcome of your last command as results. The *right way* is to access these `returns` using an `r()` call
- Let's walk through an example using **`summarize`**

summarize r()

DON'T DO THIS:

```
summarize mpg

generate above_average_mpg = 1 if mpg > 21.2973
```

- Prone to copy/paste errors
- Not reproducible — what if the next time you run your script the mean changes and you forget this line?

summarize r()

INSTEAD, DO THIS!

```
summarize mpg

generate above_average_mpg = 1 if mpg > r(mean)
```

- Instead of manually copying down what we read from the summarize command call, we are accessing it directly in Stata's memory

- Stata saves the results of your last command in `returns`

return

return list

- This command will list out all the available saved returns available from a particular statistical command
- All of these values are available via `r()` results, but they only represent the last **command** issued
  - This can be important to remember when running multiple loops

ereturn

- Many statistical commands include estimated results, which are stored in `ereturn`:

```
ereturn list
```

- `ereturn` stored values are generally connected to model fitting and parameters

- Moving back to our **`ttest`** example we can see what results are available via `r()` calls and `e()` calls

regress

```
regress dependent_var ind_var1 ind_var2 ...
```

- The `regress` command will construct a linear model with a continuous outcome variable (`dependent_var`) and independent predictors (`ind_var1 ind_var2` etc)

- The output of this command contains what we would expect from a linear model, with easily interpretable model coefficients

```
regress price mpg headroom
```

regress

ereturn list

- Since regression is a modeling command the majority of its results are stored in `ereturn`, or estimated returns
- The difference between `return` and `ereturn` can be subtle, so if you are looking for a particular return, it is good to check both (along with the **`help`** file for the command you want to use)
- For many commands, there will be output that you might also want to use that are not stored in `return` or `ereturn`
- I was often interested in running regressions and recording the estimated coefficients for predictor variables

regress _b[var]

- It turns out, Stata holds on to both regression coefficients and their standard errors in `_b[var]` and `_se[var]` macros

```
regress price mpg headroom
```

- In this example we can access `_b[_cons]`, `_b[mpg]` and `_b[headroom]` values.

- This would allow us to estimate a particular value based on our model if we were interested:

```
display _b[_cons] + _b[mpg]*25 + _b[headroom]*2
```

- This displays a model estimate for the price of a car with 25 miles per gallon and 2 inches of headroom

Stored values in strings

- When writing a string, such as a note or title text in a graph, or a string to display, you can use stored values
- However, Stata needs to know that when you are writing r(mean) in a string you intend for it to reference a stored value, and not actually r(mean)!
- To indicate your intent to Stata, use the familiar backtick + apostrophe combination:

```
summarize mpg

notes mpg: The mean mpg is r(mean)

notes mpg: the mean mpg is `r(mean)'
```

Summary

- Using stored results help reduce human error when copying and pasting specific values
- Using stored `results` makes your scripts more reproducible and flexible — if the underlying data changes your code immediately adapts
- Check both `return` and `ereturn` for command results and estimated results
- Check **`help`** files and additional documentation for stored results (like `_b[var]` in **`regress`**) that may be useful for your particular purposes

Exercises (1)

1. Titanic Data

   A. Perform a one sample t-test to test the hypothesis that the mean age on the titanic is different from 32 years of age. Use stored results to add the two-sided p-value and t-statistic to a note on the age variable.

   B. Perform a two-sample t-test to test the hypothesis that age differs between those who survived the titanic and those who did not. Use stored results to add the two-sided p-value and t-statistics to a note on the survival variable.

Exercises (2)

2. Auto Data

   A. Write a loop that creates three new standardized variables for mpg, price, and headroom (subtract the mean and divide by the standard deviation). These should be named *var*_standard.

   B. Perform a linear regression with standardized price as the dependent variable and standardized mpg and standardized headroom as predictors. Create new variables beta_mpg and beta_headroom to hold the coefficients for mpg and headroom.

   C. Run the same regression as B, but with unstandardized variables – this time include the option **beta** in your regress command. Compare the Beta column of your output to your beta_mpg and beta_headroom variables. *Check the documentation and return lists – is there some way to pull out these standardized coefficients?*