10. Flow Control

Self-Paced Workshop #10

Prof. Jack Reilly

Fall 2026

Deciding What Runs, and How Often

Every script so far has run top to bottom, each line once. *Flow control* is how you change that: run this line only *if* something is true, run that one *over and over*, or package a few lines up under a name so you can run them whenever you like. This week's lecture covered the three tools that do it:

- **Conditionals** — `if` and `else` — decide *whether* code runs. You have used the idea already, inside `mutate()` with `if_else()`.
- **Loops** — `for` and `while` — decide *how often* it runs.
- **Functions** wrap code up under a name, with inputs and an output, so you can run it again on anything.

This workshop goes through each one on the GSS, which you have used since Workshop 8.

> **Before you start**
>
> `gssr` is a 30 MB download, so the page takes several seconds longer than usual to get ready. The boxes share one R session; run them top to bottom. One warning for this page in particular: a loop that never stops will freeze the R session in the browser. If a box runs for more than a minute, reload the page, which starts a fresh session.

> **A word on programming and complexity**
>
> Features of flow control - specifically, conditionals, loops, and functions - are the most complicated and finicky *programming* elements we talk about in our class. (There are more complicated things we do with wrangling, but we don't always use loops for them; we often use canned routines in the tidyverse instead.) Accordingly, while it is important for you to *understand* conceptually what a loop is, it is not yet critical for you to be able to *write* them flawlessly. So if this part flabbergasts you, don't feel like you need to get it perfectly yet. You'll have time to practice. And the most important thing is that you *know* what these things are so you can start parsing and understanding code you see that contains them.

Step 1. Load the packages and the data

```
library(dplyr)
library(ggplot2)
library(gssr)
data(gss_sub)

gss22 <- gss_sub |>
  filter(year == 2022)
```

The 2022 GSS, as in Workshop 9: 3,544 respondents and the 20 variables in `gss_sub`.

Part 1. Loops

Step 2. `if` and `else`

Loops are built out of conditionals, so start there. `if` runs a block of code — everything between the curly brackets — only when its condition is `TRUE`; `else` gives the block to run otherwise. Take the first respondent's age:

```
gss22$age[1]

if (gss22$age[1] >= 65) {
  "65 and over"
} else {
  "Under 65"
}
```

The `[1]` picks out the first value. Change it to `[3]` and run again: the third respondent is 57, so the other block runs. That is the whole idea; `if` just needs a single `TRUE` or `FALSE` to decide with.

We have already used the same idea in earlier in the class in early recoding workshops, so this shouldn't be brand new. What's new is using it iteratively.

Step 3. A `for` loop that recodes

To make that decision for every respondent, we need to repeat it once per row. A `for` loop does exactly that: it creates a variable (usually `i`) and then sets `i` to the initial number given in a sequence and runs the block. Then it runs the next number in the sequence through the block, and so on and so forth, until it reaches the last number in the given sequence.

So, below, `i` starts at 1 (because we give the sequence `1:nrow(gss22)`) and runs the block, then sets `i` to 2 and runs it again, and so on through every value in `1:nrow(gss22)`. In this case, with 3,544 respondents, it will run it until it reaches 3,544. If the loop below looks a

little complicated, ask yourself: isn't this better than writing the same code 3,544 times?[1]

So effectively, inside the block, `[i]` means "the current row", and the block just calculates whatever it says to in that row:

```
gss22$age_group <- NA

for (i in 1:nrow(gss22)) {
  if (gss22$age[i] >= 65) {
    gss22$age_group[i] <- "65 and over"
  } else {
    gss22$age_group[i] <- "Under 65"
  }
}
```

The first line makes an empty column for the loop to fill in.[2] Reading the loop in the order the lecture took it apart:

1. `for` — the kind of loop.
2. `(i in 1:nrow(gss22))` — the range: `i` takes each value from 1 to 3,544 in turn.
3. `{ ... }` — the brackets enclose what happens on each pass.
4. The `if/else` inside — what happens: look at row `i`'s age, and write a label into row `i` of the new column.

Looks great, right? Well, yes and no - if you run the code block above, you'll get one of those inscrutable error messages - `missing value where TRUE/FALSE needed`. This is one of the most common loop errors there are, so it is worth learning to diagnose. When a loop stops, `i` keeps the value it had when it broke, so ask R which row it was on, and what was there:

```
i
gss22$age[i]
```

Row 32, whose age is `NA(n)`: "no answer." `NA >= 65` is neither `TRUE` nor `FALSE` but `NA`, and `if` cannot decide on `NA`. Real data has missing values, so a loop over real data has to handle them, and the loop has to be *exhaustive* - it cannot continue past an element that it doesn't know what to do with.[3]

To fix it, we can add a first check, with `else if` to chain the conditions:

```
gss22$age_group <- NA
```

[1] Yes, yes it is. Anything is better than writing the same code 3,544 times. That's the value of a loop, even if they're annoying to debug.

[2] The lecture's loop skipped this and let the first assignment create the column. That works on an ordinary data frame; on a tibble, which is what `gssr` and the tidyverse give you, it works but complains with an "Unknown or uninitialised column" warning. Making the column first is the same habit as `numeric(20)` in the Fibonacci example below: set up the container, then fill it.

[3] Note that our in-class code did not have this problem, because the NES dataset we were using had *already* had all its missing observations deleted.

```
for (i in 1:nrow(gss22)) {
  if (is.na(gss22$age[i])) {
    gss22$age_group[i] <- NA
  } else if (gss22$age[i] >= 65) {
    gss22$age_group[i] <- "65 and over"
  } else {
    gss22$age_group[i] <- "Under 65"
  }
}

gss22 |>
  count(age_group)
```

824 respondents 65 and over, 2,512 under, and 208 missing — the loop ran all the way through. Look at the first few rows to check it did what you meant:

```
gss22 |>
  select(id, age, age_group) |>
  head(10)
```

Try it yourself: write a loop that makes a column `college_grad`, `"Yes"` when `degree` is 3 or more (a bachelor's or graduate degree) and `"No"` otherwise. `degree` has no missing values in 2022, but it's harmless to keep the `is.na()` check anyway, so there's no reason to discard it.

```
gss22$college_grad <- NA

for (i in 1:nrow(gss22)) {

}

gss22 |>
  count(college_grad)
```

Step 4. The same recode with `mutate()`

You have seen this recode before, in one line:

```
gss22 <- gss22 |>
  mutate(age_group2 = if_else(age >= 65, "65 and over", "Under 65"))

identical(gss22$age_group, gss22$age_group2)
```

`TRUE`: the same column, and `if_else()` handled the missing ages on its own, by returning `NA` for them.

Thus, an important lesson: **for a recode, use** `mutate()`, don't turn this into a loop just because you know how to now. `mutate()` is shorter, faster, and harder to get wrong. Most

of the time, if there is a "canned" routine - a command already written to do what you want - it is better to use the canned routine.

Loops earn their place when the job is more complex: perhaps something you have to custom write, something where the line is not "do the same thing to every row", when each step depends on the result of the one before,[4] or when you do not know in advance how many steps there will be. It's a more flexible and powerful tool for thornier types of data issues - but not one to embrace unless you need to. Remember: Keep It Simple.

Step 5. A sequence that builds on itself: Fibonacci

The Fibonacci sequence starts 1, 1, and every number after that is the sum of the two before it. Each step needs the previous two, so `mutate()` cannot do it, but a loop can:

```
fibonacci <- numeric(20)  # an empty vector of length 20
fibonacci[1] <- 1         # the first two numbers are both 1,
fibonacci[2] <- 1         # so we fill those in by hand

# start at 3, go to 20
for (i in 3:20) {
  # make the current number the sum of the last two
  fibonacci[i] <- fibonacci[i - 2] + fibonacci[i - 1]
}

fibonacci
```

Same pattern as Step 3: make the container, then fill it, one position per pass. The range starts at 3 because positions 1 and 2 were done by hand, and `fibonacci[i - 2]` would not exist for `i` of 1 or 2.

Try it yourself: change both `20`s to `30`. Then add a `print(i)` line inside the loop, to watch `i` count up.

Step 6. `while` loops

A `for` loop runs a set number of times. A `while` loop runs *as long as* a condition is true, checking it before each pass:

```
x <- 1
while (x < 5) {
  print(x)
  x <- x + 1
}
```

What will it print? Work it out before you run it. (`print()` is needed because a loop, unlike the console, does not show values on its own.) `x` starts at 1, and each pass prints it and adds

[4] Don't forget that a pipe also runs steps in sequence! The difference is that each step in a pipe is a command you write out yourself, so a pipe cannot repeat a step a number of times you don't know in advance, or feed one pass's result into the next.

1; when `x` reaches 5, `x < 5` is `FALSE` and the loop stops — so 1, 2, 3, 4, but not 5.

The line `x <- x + 1` is what lets it stop. Without it `x` stays 1 forever, `x < 5` is always `TRUE`, and the loop never ends: an *infinite loop*, and the reason for the warning at the top of the page. Every `while` loop needs something inside it that moves it toward its condition being false.

Try it yourself: make the loop count *down* from 10 to 1. You need to change the starting value, the condition, and the update line.

```
x <- 1
while (x < 5) {
  print(x)
  x <- x + 1
}
```

Step 7. Fibonacci with `while`

Now Fibonacci without saying how many numbers we want — only when to stop. Keep going until the latest number passes 100,000:

```
fibonacci <- numeric(20)  # R makes the vector longer if it needs to
fibonacci[1] <- 1
fibonacci[2] <- 1

i <- 3
while (fibonacci[i - 1] < 100000) {
  fibonacci[i] <- fibonacci[i - 2] + fibonacci[i - 1]
  i <- i + 1
}

fibonacci
```

26 numbers, the last 121,393. We did not know in advance it would take 26; the condition found it. Notice the two jobs `i` has here: it is the position being filled, and adding 1 to it each pass is the update that moves the loop toward stopping, like `x <- x + 1` in Step 6.

Step 8. Fibonacci and the golden ratio

Fibonacci numbers are remarkable not just because the sequence is so simple, but because when you divide each Fibonacci number by the one before it — 1/1, 2/1, 3/2, 5/3, 8/5, ... — the ratio converges to the golden ratio. We can show this in R:

```
ratios <- fibonacci[2:26] / fibonacci[1:25]
round(ratios, 6)
```

(`fibonacci[2:26] / fibonacci[1:25]` divides each number by the one before it, all at once: R divides vectors position by position, so no loop is needed here.) The ratios bounce above and below a single value and settle on it: 1.618034. The golden ratio is usually written ϕ (phi), and it has an exact formula:

```
phi <- (1 + sqrt(5)) / 2
print(phi, digits = 12)
```

Drawn as a graph, the bouncing shows the convergence clearly:

```
tibble(n = 2:26, ratio = ratios) |>
  ggplot(aes(x = n, y = ratio)) +
  geom_hline(yintercept = phi, linetype = "dashed") +
  geom_line() +
  geom_point() +
  labs(title = "Ratios of consecutive Fibonacci numbers converge on the golden ratio",
       x = "Position in the sequence", y = "Ratio to the previous number")
```

By the tenth or so the points sit on the dashed line, as far as the eye can tell. *How* close they get is a perfect job for a `while` loop, because the question is "keep going until it is close enough" and we do not know how many steps that takes. Build the sequence, and the ratio, until the ratio is within one ten-billionth (`1e-10`) of ϕ:

```
fib   <- c(1, 1)
ratio <- fib[2] / fib[1]
i     <- 2

while (abs(ratio - phi) > 1e-10) {
  i        <- i + 1
  fib[i]   <- fib[i - 2] + fib[i - 1]
  ratio    <- fib[i] / fib[i - 1]
}

i
print(ratio, digits = 12)
```

`abs()` is the absolute value, so `abs(ratio - phi)` is the distance from ϕ whichever side the ratio is on. It takes 26 Fibonacci numbers to get that close — the same 121,393 where Step 7 stopped, which is a coincidence of the two stopping rules.

Try it yourself: change `1e-10` to `1e-4`, then to `1e-15`. How many more numbers does each extra digit of precision cost?[5]

Part 2. Functions

[5]Not many: 12 numbers for `1e-4`, 26 for `1e-10`, 38 for `1e-15` — about five more numbers for every two digits, because each step cuts the distance to ϕ by a factor of about $\phi^2 \approx 2.6$. Then try `0`, which asks for the ratio to *equal* ϕ. In exact arithmetic that never happens, and the loop would run forever. But a computer stores numbers to only about 16 significant digits, so at the 41st Fibonacci number the ratio and ϕ round to the same stored number, and the loop stops. A `while` condition that can only be met by luck is an infinite loop waiting to happen; not every one is this lucky.

You have used functions all semester: `mutate()`, `lm()`, `t.test()`, `read_csv()`. Each takes something in, does some work you do not see, and hands something back. You can write your own, and it is worth doing for anything you find yourself typing more than twice.

The best metaphor remains baking a cake. To perform the `bake()` function, you give it inputs - namely, cake batter - and some parameters (temperature, time), and the oven does things - hidden away from you - until it returns you a cake. It's not doing anything magical - well, nothing magical aside from producing cake - but it is *transforming* what you gave it according to a certain internal specified process.

Step 9. Your first function: `pct()`

In Workshops 8 and 9, we kept calculating the percentage of respondents for whom something is true. This is trivial to calculate, but its triviality makes it a good example function as well. For instance, the percent 65 and over:

```
100 * mean(gss22$age >= 65, na.rm = TRUE)
```

`gss22$age >= 65` is a `TRUE/FALSE` value for each respondent, the mean of `TRUE/FALSE` values is the share that are `TRUE`, and `na.rm = TRUE` leaves out the missing ages. We can wrap that functionality up as a function, named `pct()`, that will allow us to just *call the function* rather than rewriting the percentage calculation each and every time:

```
pct <- function(x) {
  100 * mean(x, na.rm = TRUE)
}
```

The anatomy and structure of the `pct()` function are as follows:

Part	Here
name	`pct`, assigned with `<-` like any object (look for it in the environment)
declaration	`function`
arguments — the inputs	`(x)`: one input, called `x` inside the function
code — the work, inside the curly brackets	`100 * mean(x, na.rm = TRUE)`
output	the result of that line

Importantly: running that box *defines* `pct()` but does not use it. It just creates a new function that can be called in your session (and you can see it in your environment in RStudio and Positron).

To call it, we use it like any other function, on any `TRUE/FALSE` question:

```
pct(gss22$age >= 65)
pct(gss22$degree >= 3)
pct(gss22$fefam <= 2)
```

The answers, in order: about 25% are 65 and over; 34% have a bachelor's degree or more; 24% agree a woman's place is in the home. Whatever you hand the function becomes `x` inside it, so the same code answers three different questions.

Try it yourself: what percent of 2022 respondents describe themselves as conservative (`polviews` of 5, 6 or 7)? As liberal (1, 2 or 3)?

```
pct(gss22$age >= 65)
```

Step 10. Guardians

`pct()` assumes its input is `TRUE/FALSE` values. What if it is not? Hand it the ages themselves, a mistake that is easy to make:

```
pct(gss22$age)
```

4917.7. Not an error, not a warning, but something *far worse*: a confident, wrong answer, because the mean of the ages times 100 is a number R is happy to compute. That is the worst way for code to fail, since nothing tells you it did, and this confident but clearly incorrect number will now propagate through the rest of your code, perhaps unnoticed by you for a while.

A *guardian* is a check at the top of a function that its input is what it expects, and stops with a clear message if not:

```
pct <- function(x) {
  if (!is.logical(x)) {
    stop("pct() needs TRUE/FALSE values, such as gss22$age >= 65")
  }
  100 * mean(x, na.rm = TRUE)
}

pct(gss22$age)
```

`is.logical()` asks whether `x` is `TRUE/FALSE` values; the `!` means "not". If it is not, `stop()` halts the function and shows your message as an error. The right input goes straight past the check:

```
pct(gss22$age >= 65)
```

This gives the standard structure for a function:

(1) check the inputs,
(2) do the work,
(3) return the output.[6]

[6] You can, technically, also write a guardian on the output, making sure it conforms to a particular type, but this is unusual: your function should *never* return malformed data if the inputs were correct; if it does, that's the fault of your function. Only warn the user about things they control (the inputs), and then do the work to make sure what is returned is what is advertised.

Step 11. Explicit and implicit return

How does the function know what to hand back? So far, *implicitly*: R returns the value of the last line it ran. You can also say so *explicitly*, with `return()`:

```
pct_implicit <- function(x) {
  100 * mean(x, na.rm = TRUE)          # the last line's value is returned
}

pct_explicit <- function(x) {
  return(100 * mean(x, na.rm = TRUE))  # returned on purpose
}

pct_implicit(gss22$age >= 65)
pct_explicit(gss22$age >= 65)
```

Both return the same answer. For a short function it is a matter of style, and most tidyverse code uses the implicit form. `return()` matters when you want to stop *early*, from the middle of the function. `fefam` was not asked in 1972, so every 1972 answer is missing:

```
gss72 <- gss_sub |>
  filter(year == 1972)

pct(gss72$fefam <= 2)
```

`NaN`, "not a number": with every value removed, `mean()` divides zero by zero. A second guardian can catch that case and leave early, with an honest `NA`:

```
pct <- function(x) {
  if (!is.logical(x)) {
    stop("pct() needs TRUE/FALSE values, such as gss22$age >= 65")
  }
  if (all(is.na(x))) {
    return(NA)  # nothing to compute: leave now, skip the rest
  }
  100 * mean(x, na.rm = TRUE)
}

pct(gss72$fefam <= 2)
pct(gss22$fefam <= 2)
```

The first guardian *stops* the function with an error, because bad input is a mistake. The second *returns* early, because a year with no answers is not a mistake, just nothing to report. Both are explicit, mid-function exits; the last line is still returned implicitly when neither applies.

One more important rule: anything created *inside* a function belongs to the function and disappears when it finishes. Only the returned value survives:

```
pct_share <- function(x) {
  share <- mean(x, na.rm = TRUE)
  return(100 * share)
}

pct_share(gss22$age >= 65)
share
```

`object 'share' not found` — the function made `share`, used it, and threw it away. That is a feature: a function cannot quietly overwrite something in your environment that happens to have the same name.

Step 12. Options with defaults

Arguments can be options as well as inputs, and an option can have a *default* that applies unless you say otherwise. A `digits` option to round the result, defaulting to one decimal place:

```
pct <- function(x, digits = 1) {
  if (!is.logical(x)) {
    stop("pct() needs TRUE/FALSE values, such as gss22$age >= 65")
  }
  if (all(is.na(x))) {
    return(NA)
  }
  round(100 * mean(x, na.rm = TRUE), digits)
}

pct(gss22$degree >= 3)
pct(gss22$degree >= 3, digits = 3)
```

You have been using defaults all semester without noticing: `t.test()`'s `conf.level = .95` and `mean()`'s `na.rm = FALSE` are both defaults, which is why you only type them when you want something different.

Step 13. Putting it together

A function and a loop together: the share agreeing with `fefam` in every year of the GSS. Loop over the years, take each year's respondents, and hand their answers to `pct()`:

```
years <- unique(gss_sub$year)
agree <- numeric(length(years))

for (i in seq_along(years)) {
  one_year <- gss_sub |>
    filter(year == years[i])
  agree[i] <- pct(one_year$fefam <= 2)
```

```
}

tibble(year = years, agree = agree)
```

`seq_along(years)` is 1, 2, 3, ... up to the number of years, the safe way to loop over the positions of a vector. Each pass filters one year and stores its answer in the matching position of `agree`. The years when `fefam` was not asked come back `NA` — the early `return()` from Step 11, doing its job 11 times.

And, as with the recode in Step 4, the tidyverse does this kind of loop for you. Your own function works inside `summarise()` exactly like `mean()` does:

```
gss_sub |>
  group_by(year) |>
  summarise(agree = pct(fefam <= 2)) |>
  filter(!is.na(agree)) |>
  ggplot(aes(x = year, y = agree)) +
  geom_line() +
  geom_point() +
  labs(title = "Agree it is better for the man to work and the woman to tend the home",
       x = NULL, y = "Percent")
```

The same line as in Workshop 8, now computed by a function you wrote.

Some Perspective

This is **a lot**. It's OK. If you want more genuine programming in R, I strongly recommend checking out Braun and Murdoch's book, *A First Course in Statistical Programming with R*, which is a classic for a reason and shows exactly how and why R's programming language-ness can be very useful for working with statistics and data.

That said, many people who use R regularly **do not need much programming** - at least, programming that uses principles from flow control. Conditionals, loops, and functions are the parts of R that make it a programming language rather than just a fancy calculator, but the tidyverse and assorted packages handle a lot of basic data manipulation tasks just fine.[7]

For a lot of everyday data work, in fact, the tidyverse already has the loop built in without you having to write it as such — `mutate()`, `summarise()`, `group_by()` — so reach for those first, and a loop when you end up outside what canned routines can do for you.

So why learn it? There are at least three good reasons:

[7] Informally some use "scripting", as opposed to "programming", to refer to a lighter form of programming, where you don't use as much in terms of flow control but do write code to give the computer commands. Scripts also tend to be more narrow and task-oriented in nature. But there's not a hard and fast rule here, just customary vagaries of (human) language.

1) As with many things, you need to know what tools are in the toolkit. Some tools (i.e., `mutate()`) you'll be well practiced with, and know like second nature, but other tools are ones that might hang on the shelf for a while, only to be picked up and dusted off when you run into the unique problem that needs them. Flow control principles might be one of those tools for you. However, even if you don't use it much, it's important to know that it's there to use if you want it.

2) LLMs (at least, as of this writing) *love* using loops and functions, including sometimes when they don't need to. As most of us will be spending more and more time as the human in the loop reviewing what LLMs give us after a prompt, it's important to be able to *interpret* a loop and not be thrown by it, even if you yourself may not be the one actually writing it.

3) Nothing is worse than writing the same line of code over and over again. Typically, once you get comfortable with them, writing a loop *saves* you a lot of time when you find yourself copy/pasting the same code over and over, even if it takes more effort to get the loop running correctly.

Finally, there is always more! We are not going to cover `apply()` commands, for instance, in this class, but they are another way of controlling the flow of your code (you can almost think of them like a super-loop), and their parallel versions (such as `parallel::mclapply()`) are one of the easiest ways to divide your work across multiple CPU cores, saving you time on truly gargantuan data tasks.

However, for now, store loops in the back of your brain for when you need them. And if they don't click immediately, that's OK: this is a higher level of coding than we do in most of the class, so it might take a couple times returning to it (in this class or another) to get it down.