Functions

Functions in R

I've been denoting functions with parentheses: `func()`

We've seen functions such as:

- `mean()`
- `tbl_summary()`
- `init()`
- `create_github_token`

Functions take **arguments** and return **values**

Looking inside a function

If you want to see the code within a function, you can just type its name without the parentheses:

```
1  usethis::create_github_token
```

```
function (scopes = c("repo", "user", "gist", "workflow"), description = "DESCRIBE THE TOKEN'S USE
CASE",
    host = NULL)
{
    scopes <- glue_collapse(scopes, ",")
    host <- get_hosturl(host %||% default_api_url())
    url <- glue("{host}/settings/tokens/new?scopes={scopes}&description={description}")
    withr::defer(view_url(url))
    hint <- code_hint_with_host("gitcreds::gitcreds_set", host)
    ui_todo("\n    Call {ui_code(hint)} to register this token in the \\\n    local Git
credential store\n    It is also a great idea to store this token in any password-management \\\n
software that you use")
    invisible()
}
<bytecode: 0x10d9131a0>
<environment: namespace:usethis>
```

Structure of a function

```
1  func <- function()
```

You can name your function like you do any other object

- Just avoid names of existing functions

Structure of a function

```
1 func <- function(arg1,
2                  arg2 = defaul
3 }
```

What objects/values do you need to make your function work?

- You can give them default values to use if the user doesn't specify others

Structure of a function

```
1  func <- function(arg1,
2                   arg2 = defaul
3
4  }
```

Everything else goes within curly braces

- Code in here will essentially look like any other R code, using any inputs to your functions

Structure of a function

Structure of a function

```
1 func <- function(arg1,
2                  arg2 = defaul
3   new_val <- # do something wi
4   return(new_val)
5 }
```

Return something new that the code has produced

- The `return()` statement is actually optional. If you don't put it, it will return the last object in the code. When you're starting out, it's safer to always explicitly write out what you want to return.

Example: a new function for the mean

Let's say we are not satisfied with the `mean()` function and want to write our own.

Here's the general structure we'll start with.

```
1 new_mean <- function() {
2
3 }
```

New mean: arguments

We'll want to take the mean of a vector of numbers.

It will help to make an example of such a vector to think about what the input might look like, and to test the function. We'll call it `x`:

```
1 x <- c(1, 3, 5, 7, 9)
```

We can add `x` as an argument to our function:

```
1 new_mean <- function(x) {
2
3 }
```

New mean: function body

Let's think about how we calculate a mean in math, and then translate it into code:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

So we need to sum the elements of `x` together, and then divide by the number of elements.

We can use the functions `sum()` and `length()` to help us.

We'll write the code with our test vector first, before inserting it into the function:

```
1  n <- length(x)
2  sum(x) / n
```

```
[1] 5
```

New mean: function body

Our code seems to be doing what we want, so let's insert it. To be explicit, I've stored the answer (within the function) as `mean_val`, then returned that value.

```
1 new_mean <- function(x) {
2   n <- length(x)
3   mean_val <- sum(x) / n
4   return(mean_val)
5 }
```

Testing a function

Let's plug in the vector that we created to test it:

```
new_mean(x = x)
```

```
[1] 5
```

And then try another one we create on the spot:

```
new_mean(x = c(100, 200, 300))
```

```
[1] 200
```

Adding another argument

Let's say we plan to be using our `new_mean()` function to calculate proportions (i.e., the mean of a binary variable). Sometimes we'll want to report them as multiplier by multiplying the proportion by 100.

Let's name our new function `prop()`. We'll use the same structure as we did with `new_mean()`.

```
1 prop <- function(x) {
2     n <- length(x)
3     mean_val <- sum(x) / n
```

Testing the code

Now we'll want to test on a vector of 1's and 0's.

```
1  x <- c(0, 1, 1)
```

To calculate the proportion and turn it into a percentage, we'll just multiply the mean by 100.

```
1  multiplier <- 100
2  multiplier * sum(x) / length(x)
```

```
[1] 66.66667
```

Testing the code

We want to give users the option to choose between a proportion and a percentage. So we'll add an argument `multiplier`. When we want to just return the proportion, we can just set `multiplier` to be 1.

```
multiplier <- 1
multiplier * sum(x) / length(x)
```

```
[1] 0.6666667
```

```
multiplier <- 100
multiplier * sum(x) / length(x)
```

```
[1] 66.66667
```

Adding another argument

If we add `multiplier` as an argument, we can refer to it in the function body.

```
1 prop <- function(x, multiplier) {
2    n <- length(x)
3    mean_val <- multiplier * sum(x) / 
4    return(mean_val)
5 }
```

Adding another argument

Now we can test:

```
1 prop(x = c(1, 0, 1, 0), multiplier =
```

```
[1] 0.5
```

```
1 prop(x = c(1, 0, 1, 0), multiplier =
```

```
[1] 50
```

Making a default argument

Since we don't want users to have to specify `multiplier = 1` every time they just want a proportion, we can set it as a **default**.

```
1 prop <- function(x, multiplier = 1) {
2   n <- length(x)
3   mean_val <- multiplier * sum(x) / n
4   return(mean_val)
5 }
```

Now we only need to specify that argument if we want a percentage.

```
1 prop(x = c(0, 1, 1, 1))
```

```
[1] 0.75
```

```
1 prop(x = c(0, 1, 1, 1), multiplier = 100)
```

Caveats

- This is obviously not the best way to write this function!
- For example, it will still work if `x = c(123, 593, -192)`…. but it certainly won't give you a proportion or a percentage!
- We could also put `multiplier =` *any number*, and we'll just be multiplying the answer by that number – this is essentially meaningless.
- We also haven't done any checking to see whether the user is even entering numbers! We could put in better error messages so users don't just get an R default error message if they do something wrong.

```
1 prop(x = c("blah", "blah", "blah"))
```

Exercises

Create some functions!