Lecture 7: Programming

James Sears*

AFRE 891 SS 24

Michigan State University

*Parts of these slides are adapted from **“Data Science for Economists”** by Grant McDermott and **“Advanced Data Analytics”** by Nick Hagerty.

Table of Contents

Prologue

Programming

So far in class we've learned how to do a lot of things in R, but we can exponentially increase our data analytics skills (and how quickly we get things done) by learning some **programming**.

- Write custom functions to execute specific tasks
  - Scrape all Yellowpages business links for a given search term in hundreds of different cities
- Conditionally define variables or execute different tasks
  - Create a variable conditional on another variables' values
- Perform a repeated task by looping over values
  - Create a set of state-level dummy variables from state FIPS codes
- Run tasks efficiently in parallel
  - Calculate parcel or farm-level measures of precipitation and temperature

Programming

Packages we'll use today:

```
pacman::p_load(dslabs, tidyverse, furrr, tictoc, future, progressr)
```

And let's load in the `murders` data from the `dslabs` package:

```
data(murders)
```

If/Else Statements

If/Else Statements

If/else statements are a type of **conditional expression**.

- Check to see if a logical condition is True
- If True, do a thing
- If False:
    - Do a different thing,
    - Do nothing, or
    - Check *another* condition, do a thing if True, etc.

If/Else Statements

For example: print the reciprocal of `a`, unless `a` is 0.

```r
a = 0

if(a ≠ 0) {

  print(1 / a)

} else {

  print("Reciprocal does not exist.")

}
```

```
## [1] "Reciprocal does not exist."
```

Statements like this are used for **control flow** of your code.

- Used all the time in software development
- Used occasionally in data analysis, more often in custom functions and packages.

If/Else Statements

You can also link together multiple condition with `else if`s.

```r
if(a > 0) {

  print("a is Positive")

} else if (a < 0){

  print("a is Negative")

} else {

  print("a is Zero")
}
```

```
## [1] "a is Zero"
```

If/Else Statements

A related function that you *will* use all the time in data analysis: `ifelse`.

syntax: `ifelse(CONDITION, ACTION_IF_TRUE, ACTION_IF_FALSE)`

- `CONDITION`: a logical condition *`ACTION_IF_TRUE`: what to do if the condition is true* `ACTION_IF_FALSE`: what to do if the condition is false

For example:

```
a = 0
ifelse(a > 0, 1/a, NA)
```

```
## [1] NA
```

If/Else Statements

syntax: `ifelse(CONDITION, ACTION_IF_TRUE, ACTION_IF_FALSE)`

`ifelse` is particularly useful because it is **vectorized** and can be applied over **a vector of elements all at once**

For example, to change negative numbers to missing:

```
b = c(0, 1, 2, -3, 4)
ifelse(b < 0, NA, b)
```

```
## [1]  0  1  2 NA  4
```

If/Else Statements

syntax: `ifelse(CONDITION, ACTION_IF_TRUE, ACTION_IF_FALSE)`

`ifelse` is particularly useful because it is **vectorized** and can be applied over **a vector of elements all at once**

Or for adding a conditional variable - for example, whether or not a state is Michigan

```
murders <- murders %>% mutate(
    is_michigan = ifelse(state == "Michigan", "Is Michigan", "Is Not Mich
                  )
murders[c(1, 23:26),]
```

```
##          state abb        region population total      is_michigan
## 1      Alabama  AL         South    4779736   135 Is Not Michigan
## 23    Michigan  MI North Central    9883640   413      Is Michigan
## 24   Minnesota  MN North Central    5303925    53 Is Not Michigan
## 25 Mississippi  MS         South    2967297   120 Is Not Michigan
## 26    Missouri  MO North Central    5988927   321 Is Not Michigan
```

case_when()

While it's technically possible to use nested ifelses, friends don't let friends nest ifelses.

Instead, use **dplyr's** `case_when()`

```
x <- 1:10
## dplyr::case_when()
case_when(
  x <= 3 ~ "small",
  x <= 7 ~ "medium",
  TRUE ~ "big"       # Default value
  )
```

```
##  [1] "small"  "small"  "small"  "medium" "medium" "medium" "medium" "big"
##  [9] "big"    "big"
```

case_when()

Works great within `mutate()` as well!

```
murders <- murders %>% mutate(
     my_opinion = case_when(
       state == "Michigan" ~ "Great State",
       state %in% c("California", "Hawaii") ~ "Also Solid State",
       state == "Missouri" ~ "More like Misery am I right",
       TRUE ~ "A State")
     )
murders[c(1, 5, 12, 23, 26, 38),c(1,7)]
```

```
##           state                  my_opinion
## 1       Alabama                     A State
## 5    California            Also Solid State
## 12       Hawaii            Also Solid State
## 23     Michigan                 Great State
## 26     Missouri More like Misery am I right
## 38       Oregon                     A State
```

For Loops

Abstraction

Often you will have tasks where you find yourself copying and pasting your code to do the same thing n times, with only minor tweaks each time.

Q: What's wrong with that?

- Annoying (especially if n is large)
- Hard to change later if needed
- Prone to errors/bugs

Instead, you can **abstract** your code: define it once, and run it multiple times. The rest of this lecture covers tools for abstraction in different situations.

A good rule to aim for is to **never copy-and-paste more than twice.** If you're pasting more than that, abstract it instead!

Abstraction Methods

There are several different methods for code abstraction that we'll go over:

1. **For loops:** when you want to repeat the same code for **different values of a variable or vector**
2. **Functions:** when you want to repeat the same code for potentially **different values of all arguments/variables** or with **different settings/samples**
3. **Vectorization and Functionals:** when you want to **repeat a function over different values of arguments**

For Loops

The **for loop** is a simple tool for **iteration**

```
for (INDEX in RANGE){
  action(INDEX)
```

]

- `INDEX` the name of the index you want to use (often `i` but can be anything)
- `RANGE` the vector of values to iterate over (can be numbers, characters, or objects)

For Loops

The **for loop** is a simple tool for **iteration**

```
for (i in 1:6){
  print(paste0("It is ", i, " O'Clock."))
}
```

```
## [1] "It is 1 O'Clock."
## [1] "It is 2 O'Clock."
## [1] "It is 3 O'Clock."
## [1] "It is 4 O'Clock."
## [1] "It is 5 O'Clock."
## [1] "It is 6 O'Clock."
```

For Loops

You can also combine for loops with if-else:

```r
for (i in c("Indiana", "Michigan", "Colorado")){
  if (i == "Michigan"){
      print("This is Michigan")
  } else {
      print("This is not Michigan")
  }
}
```

```
## [1] "This is not Michigan"
## [1] "This is Michigan"
## [1] "This is not Michigan"
```

For Loops

Suppose you wanted to calculate the mean of the numeric variables in `murders` and the murder rate. We could manually type and copy-paste:

```
murders <- mutate(murders, rate = total/population * 1e5)
mean(murders$total)
```

```
## [1] 184.3725
```

```
mean(murders$population)
```

```
## [1] 6075769
```

```
mean(murders$rate)
```

```
## [1] 2.779125
```

For Loops

Or we could avoid copy-past errors and use a for loop:

```
for (var in c("total", "population", "rate")){
print(mean(murders[[var]]))
}
```

```
## [1] 184.3725
## [1] 6075769
## [1] 2.779125
```

For Loops

We an also loop over an **object in memory:**

```r
numeric_col ← c("total", "population", "rate")
for (var in numeric_col){
print(mean(murders[[var]]))
}
```

```
## [1] 184.3725
## [1] 6075769
## [1] 2.779125
```

For Loops

Or **assign output to memory** too

```r
numeric_col ← c("total", "population", "rate")
means ← vector() # initiate an empty vector
for (var in numeric_col){
means[[var]] ← mean(murders[[var]])
}
```

There is one technical problem with this code. The vector storing the output **"grows" at each iteration**, which can make the loop **very slow**.

For Loops

Better: give your empty vector the **right length** *before* starting.

```
means <- vector("numeric", length = length(numeric_col)) # initiate an em
for (i in 1:length(numeric_col)){
  col_num <- which(colnames(murders) == numeric_col[i])
  means[[i]] <- mean(murders[[col_num]])
}
```

For Loops: Caveat

For-loops are actually **discouraged in R programming**.

- We're covering them because the concepts are foundational.
- But R has nicer ways to iterate, called **vectorization**.
- To do proper vectorization, we first need to know how to **write functions**.

Functions

Functions

We've already seen a **multitude of functions** in R

- pre-packaged with base R
- loaded by different packages (e.g. `dplyr::mutate()`)

Regardless of where they come from, the all follow the same basic syntax:

```
function_name(ARGUMENTS)
```

Custom Functions

While we will often use pre-made functions, you can --- and should! --- write your own functions too. This is easy to do with the generic **`function()`** function.[1]

If you only have a short function, you can write it all on a **single line:**

```
function(ARGUMENTS) OPERATIONS
```

[1] Yes, it's a function that let's you write functions. Very meta.

Custom Functions

Oftentimes we want our function code to span **multiple lines**. In this case we can use brackets:

```
function(ARGUMENTS) {
  OPERATIONS
  return(VALUE)
}
```

1. Yes, it's a function that let's you write functions. Very meta.

Custom Functions

Rather than write **anonymous** functions, we can **name our functions** to assign them to memory and reuse them throughout our file:

```
my_func <- function(ARGUMENTS) {
    OPERATIONS
    return(VALUE)
  }
```

Try to give your functions short, pithy names that are

- Informative to you
- Clear to anyone else who might read the code

Building Custom Functions

Let's start with a basic function: calculate a **number's square**.[2]

```
square <-         # function name
  function(x){ # the arguments of our function (here just one)
  x^2           # the operation(s) that our function performs
  }
```

```
square(4)
```

```
## [1] 16
```

[2] I want to note that this **isn't a useful function**. R's arithmetic function already handle vectorised exponentiation and do so very efficiently.

Specifying Return Values

We can **specify return values** with `return()`

- Helpful when our function performs a bunch of intermediate steps

```
square <- function(x){
  x_sq <- x^2 # assign squared value as intermediate object
  return(x_sq)
  }
```

Specifying Return Values

Testing:

```
square(3)
```

```
## [1] 9
```

Note that the intermediate objects **don't stay in memory** - they're automatically removed as soon as the function is done running.

If we left out the `return()`, the function will return the result of the very last operation

Specifying Return Values

If we want to return **multiple objects** from our function, we need to either

1. Use a List

```r
square_list <- function(x){
  x_sq <- x^2 # assign squared value as intermediate object
  res <- list(value = x, val_squared = x_sq)
  return(res)
  }

square(3)
```

```
## [1] 9
```

Specifying Return Values

If we want to return **multiple objects** from our function, we need to either

2. Build a data frame (a tidy solution!)

```
square_df <- function(x){
  x_sq <- x^2 # assign squared value as intermediate object
  res <- data.frame(value = x, val_squared = x_sq)
  return(res)
  }

square(3)
```

```
## [1] 9
```

Default Argument Values

We can also assign **default argument values**

- Allows for all/any arguments to be optional
- Use the supplied value when supplied
- Use default value when not

Suppose we wanted to expand our function to do any exponent and not just squares:

```
raise_power ← function(x = 2, power = 2){
  res ← data.frame(
      value = x,
      power = power,
      value_raised = x^power
  )
  return(res)
}
```

Default Argument Values

Setting default values doesn't affect typical function usage:

```
raise_power(x = 5, power = 3) # uses specified values
```

```
##   value power value_raised
## 1     5     3          125
```

But now any argument that we omit will **use the default values** and the function will run:

```
raise_power() # uses default values of x and power = 2
```

```
##   value power value_raised
## 1     2     2            4
```

Default Argument Values

Setting default values doesn't affect typical function usage:

```
raise_power(x = 5, power = 3) # uses specified values
```

```
##   value power value_raised
## 1     5     3          125
```

Without supplying argument values, our previous function wouldn't have worked:

```
square()
```

```
## Error in square(): argument "x" is missing, with no default
```

Indirection and Name Injection

Indirection

A common use-case for custom functions is **iterating over variables**

- Repeat a cleaning task over multiple variables in a data frame
- Run analysis with a different dependent variable

For example, let's go back to our `square` function. By default it applies over an entire vector:

```
square(murders$rate)
```

```
##  [1]   7.9773697   7.1566199  13.1734682  10.1722092  11.3848093   1.670435
##  [7]   7.3656453  17.9092897 270.6930871  11.5468718  14.3665453   0.264804
## [13]   0.5860059   8.0483466   4.7964199   0.4752012   4.8757523   7.146003
## [19]  59.9475608   0.6857300  25.7542601   3.2478494  17.4608856   0.998520
## [25]  16.3546200  28.7284388   1.4709757   3.0699848   9.6750631   0.144250
## [31]   7.8289826  10.5867194   7.1180103   8.9959426   0.3536860   7.220627
## [37]   8.7552904   0.8830065  12.9438141   2.3106835  20.0285200   0.965470
## [43]  11.9089569  10.2487076   0.6335858   0.1021576   9.7631255   1.912673
## [49]   2.1231443   2.9092373   0.7869696
```

Indirection

We could use it *within* a mutate if we want a new column in our data frame:

```
murders <- murders %>%
  mutate(rate_sq = square(rate))
select(murders, starts_with("rate")) %>% head()
```

```
##       rate   rate_sq
## 1 2.824424  7.977370
## 2 2.675186  7.156620
## 3 3.629527 13.173468
## 4 3.189390 10.172209
## 5 3.374138 11.384809
## 6 1.292453  1.670435
```

But doing this for a lot of variables would require a lot of typing (and wouldn't vectorize over multiple variables well)

Indirection

What we might want to do is modify our function to use **variable names and the dataframe** as the arguments to directly add a new variable:

```r
square_df ← function(var, # variable to square
                     df){ # data frame to square variables in
  df ← mutate(df,
              newvar = var * var)
  return(df)
}
```

Indirection

However, if we try and use this function on the `rate` variable in `murders` with a string, we get an error:

```
square_df(
  var = "rate",
  df = murders)
```

```
## Error in mutate():
## ℹ In argument: newvar = var * var.
## Caused by error in var * var:
## ! non-numeric argument to binary operator
```

Indirection

We get a similar error if we give the variable argument as a **data-variable**

- **data-variable**: a "statistical" variable that lives **in a data frame**

```
square_df(
  var = rate,
  df = murders)
```

```
## Error in mutate():
## ℹ In argument: newvar = var * var.
## Caused by error:
## ! object 'rate' not found
```

Indirection

This is an issue of **indirection**, which occurs in cases like this

- Want to interpret the argument as an **environment-variable** rather than as as a **data-variable**.
- **env-variable**: "programming" variable/object that lives in your environment (i.e. data frame created with `←`)

Fortunately, there are a couple programmatic ways around this.

Indirection

Solution A: provide the argument as a **data-variable**, and

1. **defuse** the string with `enquo()`
2. **unquote** the defused string in operations with `!! defused_string`

```
square_def <- function(var, # data-var rather than a string
                            df){
  var <- enquo(var) # defuse the string

  df <- mutate(df,
               newvar = !! var * !! var # square the defused string
               )
  return(df)
}
square_def(rate, murders) %>% select(rate, newvar) %>% head()
```

```
##        rate    newvar
## 1 2.824424  7.977370
## 2 2.675186  7.156620
## 3 3.629527 13.173468
```

Indirection

Solution B: provide the argument as a **data variable**, and within function operations **embrace** the argument with double braces `{{ var }}`

```
square_embr ← function(var, # data-var rather than a string
                       df){
  df ← mutate(df,
              newvar = {{ var }} * {{ var }} )
  return(df)
}
square_embr(rate, murders) %>% select(rate, newvar) %>% head()
```

```
##       rate    newvar
## 1 2.824424  7.977370
## 2 2.675186  7.156620
## 3 3.629527 13.173468
## 4 3.189390 10.172209
## 5 3.374138 11.384809
## 6 1.292453  1.670435
```

Indirection

Solution C: defuse the string with `ensym()`

- Allows for supplying the argument as either a **character string** or a **data variable**

```
square_ensym <- function(var, df){

  df <- mutate(df,
                newvar =  !!ensym(var) * !!ensym(var) # square the defused
                )
  return(df)
}
square_ensym("rate", murders) %>% select(rate, newvar) %>% head(3)
```

```
##        rate   newvar
## 1 2.824424  7.97737
## 2 2.675186  7.15662
## 3 3.629527 13.17347
```

```
square_ensym(rate, murders) %>% select(rate, newvar) %>% head(3)
```

Name Injection

We can combine defusing or embracing with **name injection** to customize our variable names.

- i.e. call the new squared rate variable `rate_sq` rather than `newvar`

Often we want to programmatically create new variable names based either on

1. A supplied character string as a function argument, or
2. Iterating on the data-variable's name directly in the function

Name Injection

Approach 1: use **glue syntax** and **supply the new name as a third argument**:

- `newname` the new variable name as a character string
- Glue syntax with `"{newname}"`
- Programmatic assignment operator `:=` instead of `=`

```
square_inj_1 ← function(var, df,
                        newname){ # new variable name to use

  df ← mutate(df,
              "{newname}" := {{ var }} * {{ var }} )
  return(df)
}
square_inj_1(rate, murders, "rate_sq") %>% select(rate, rate_sq) %>% head(
```

```
##        rate   rate_sq
## 1 2.824424  7.977370
## 2 2.675186  7.156620
```

Name Injection

Approach 1 works with `ensym()` too

```
square_inj_1b <- function(var, df,
                            newname){ # new variable name to use

  df <- mutate(df,
                    "{newname}" := !!ensym(var) * !!ensym(var) )
  return(df)
}
square_inj_1b("rate", murders, "rate_squared") %>% select(rate, rate_squa
```

```
##       rate rate_squared
## 1 2.824424     7.977370
## 2 2.675186     7.156620
## 3 3.629527    13.173468
## 4 3.189390    10.172209
## 5 3.374138    11.384809
## 6 1.292453     1.670435
```

Name Injection

Approach 2A: use **glue syntax** and **create the name from the data-variable**:

- `expr()` "defuses" the supplied expression
  - Converts the data-variable (i.e. `rate`) to a name
- Glue syntax with `"{newname}"`
- Programmatic assignment operator `:=` instead of `=`

```
square_inj_2a <- function(var, df){
    new_var <- expr(rate) %>% paste0("_sq") # create new variable name in

  df <- mutate(df,
                 "{new_var}" := {{ var }} * {{ var }} ) # glue syntax to as
  return(df)
}
square_inj_2a(rate, murders) %>% select(rate, rate_sq) %>% head()
```

```
##         rate   rate_sq
## 1 2.824424  7.977370
## 2 2.675186  7.156620
```

Name Injection

Approach 2B: use **glue syntax** and **embracing**:

- Glue syntax with `"{{newname}}_sq"` (no intermediate name object)
- Programmatic assignment operator `:=` instead of `=`

```
square_inj_2b ← function(var, df){

  df ← mutate(df,
              "{{ var }}_sq" := {{ var }} * {{ var }} ) # Glue syntax wi
  return(df)
}
square_inj_2b(rate, murders) %>% select(rate, rate_sq) %>% head()
```

```
##       rate   rate_sq
## 1 2.824424  7.977370
## 2 2.675186  7.156620
## 3 3.629527 13.173468
## 4 3.189390 10.172209
## 5 3.374138 11.384809
## 6 1.292453  1.670435
```

Name Injection

Approach 2C: you guessed it, `ensym()` still works

```
square_inj_2c ← function(var, df){

  df ← mutate(df,
              "{{ var }}_sq" := !!ensym(var) * !!ensym(var) ) # Glue syn
  return(df)
}
square_inj_2c("rate", murders) %>% select(rate, rate_sq) %>% head()
```

```
##       rate   rate_sq
## 1 2.824424  7.977370
## 2 2.675186  7.156620
## 3 3.629527 13.173468
## 4 3.189390 10.172209
## 5 3.374138 11.384809
## 6 1.292453  1.670435
```

Vectorization

Vectorization

Where the real benefits of custom functions, indirection, and name injection come in are with **vectorization** and **functionals**.

These approaches give a new way to repeatedly iterate a function over a vector of argument values.

Two main approaches:

1. **apply** family
   - `apply()`, `lapply()`, `sapply()`, `mapply()`
2. Tidy**map** **list** functions in **purrr**
   - `map()` and `map2()` with `list_c()`, `list_rbind()`, `list_cbind()`
     - Recently superseded the `map_dfr()`, `map_dfc()` functions

apply Family

The base R **apply** family gives methods for iterating a function over a vector of arguments depending on the format and type of output we want

Function	Description	Output Type
`lapply(X, FUN)`	apply `FUN` to every element of `X`	list
`sapply(X, FUN)`	apply `FUN` to every element of `X`	vector, matrix, or array
`vapply(X, FUN)`	`sapply` with specified output types	vector or array
`mapply(FUN, ARG1, ARG2, ...)`	multivariate version of `sapply`	list
`apply(X, MARGIN, FUN)`	apply `FUN` to every element of `X` over dimension `MARGIN`	vector, matrix, array, or list

Apply

Suppose you wanted to standardize all the numeric variables in the `murder` data.

You might write a function like this:

```r
calculate_z = function(x) {
  z = (x - mean(x)) / sd(x)
  return(z)
}
```

apply Functions

However, applying it over all the numeric variables at once leads to this:

```
numeric_cols = c("total", "population", "rate")
murder_numbers = murders[numeric_cols]

calculate_z(murder_numbers)
```

```
## Error in is.data.frame(x): 'list' object cannot be coerced to type 'double'
```

This is an example of a function that **isn't vectorized.**

apply Functions

While we could put our function into a for loop, a more efficient/legible approach would use `sapply`[3]:

`sapply(X, FUN)`

```
sapply(murder_numbers, calculate_z) %>% head()
```

```
##           total  population        rate
## [1,] -0.2090939 -0.18890769  0.01844305
## [2,] -0.7003568 -0.78207215 -0.04231860
## [3,]  0.2017034  0.04609577  0.34623812
## [4,] -0.3869650 -0.46057478  0.16703781
## [5,]  4.5426034  4.54448196  0.24225740
## [6,] -0.5055457 -0.15254681 -0.60529343
```

3. `sapply` is an example of a **functional:** a function that takes another function as an argument.

map and list_ Functions

The tidy alternative to the apply functions are the `map_` family in **purrr**

- Work a lot like the `apply_` functions, but with tidyverse syntax
- Combine with `list_` functions to convert to a vector or dataframe

Function	Description	Output Type
`map(X, FUN)`	apply `FUN` to every element of `X`	list
`map2(X1, X2, FUN)`	apply `FUN` to every element of `X1` and `X2`	list
`list_c()`	combine list elements into a vector	vector
`list_rbind()`	combines elements into a data frame row-wise	data frame
`list_cbind()`	combines elements into a data frame column-wise	data frame

map()

Just like with `sapply()` we can iterate our `calculate_z()` over all numeric variables:

`map(X, FUN)`

```
z_map ← map(murder_numbers, calculate_z)
class(z_map)
```

```
## [1] "list"
```

```
z_map
```

```
## $total
##  [1] -0.20909395 -0.70035678  0.20170342 -0.38696497  4.54260341 -0.50554566
##  [7] -0.37002488 -0.61989131 -0.36155483  2.05240907  0.81154693 -0.75117707
## [13] -0.73000195  0.76072663 -0.17944878 -0.69188673 -0.51401570 -0.28955941
## [19]  0.70567132 -0.73423697  0.46003990 -0.28108936  0.96824283 -0.55636595
## [25] -0.27261931  0.57862059 -0.73000195 -0.64530146 -0.42508019 -0.75964712
## [31]  0.26099376 -0.49707561  1.40868537  0.43039473 -0.76388214  0.53203532
```

list_

The `list_` functions provide a convenient way to convert `map()` output directly to a dataframe:

- Loop our `square_inj_2c()` function over all three numeric variables
- Combine each of the dataframes

```
map_sq <- map(
  c("total", "population", "rate"), # first argument: variable names
  square_inj_2c, # function to iterate over
  df = murders # additional static arguments
) %>%
  list_cbind(name_repair = "unique") # account for duplicated names
class(map_sq)
```

```
## [1] "data.frame"
```

```
colnames(map_sq)
```

```
##  [1] "state...1"          "abb...2"           "region...3"
```

Parallelization

Parallelization

One distinct advantage of R over Stata is the ability to **run code in parallel**

- i.e. split a repeated task across multiple CPU cores simultaneously
- Useful in any situation where we would use `map()` - i.e. bootstrapping, extracting parcel-level raster information

Stata

- SE: runs in "serial" on one core
- MP Student: 4 core ($375/yr)
- MP 8 Core: $655/yr

R and **furrr**

- `future_map` functions work exactly like **purrr's** `map()`
- Run across as many cores as your system has
- See progress with **progressr**
- Annual cost: $0

The Power of Parallel

To see the benefit of running code in parallel, let's write a **purposefully slow function:**

```
slow_square <- function(x = 1){
  Sys.sleep(1/2) # wait half a second
  return(x^2)
}
```

The Power of Parallel

How long does it take to run this function?[5]

- Use `tic()` and `toc()` from **tictoc** to calculate elapsed time

```
tic()
square_serial <- map(1:24, slow_square)
toc()
```

```
## 12.2 sec elapsed
```

The function runs in **serial**. so it takes approximately $1/2 * 24 = 12$ seconds

- Using one core, runs for $x = 1$, then when done moves on to $x = 2, \ldots, 24$

[5] `sapply()` and `map()` take nearly the exact same time. There are also several **type-specific versions** of `map` in case you want output to be a logical, integer, double, or character, etc.

Parallelization

We can **speed this up**. Modern CPUs are made up of multiple **cores** (processing units) that can all be given tasks simultaneously, allowing us to run code in **parallel**.

First, use `future::availableCores()` to determine how many cores you have:

```
availableCores()
```

```
## system
##     24
```

Your number of cores will likely differ

- Most laptops have at least 4-8 cores these days.
- Even recent Chromebooks have 6!

furrr

furrr functions make it easy to **parallelize** in just a few steps.

1. Set a "plan" for how the code will be run in parallel
   - Number of cores to use, how to execute tasks
2. Use `future_` version of your preferred `map_` function
3. Close parallel plan

First, we will **set the plan** and tell R how to execute the parallel session:

```
# Calculate a "safe" number of cores (allow for background processes)
n_cores = availableCores() - 2

# Set the "plan"
plan(strategy = "multisession", # run in parallel in separate background
     workers = n_cores # use the desired number of cores
     )
```

furrr

furrr functions make it easy to **parallelize** in just a few steps.

1. Set a "plan" for how the code will be run in parallel
    - Number of cores to use, how to execute tasks
2. Use `future_` version of your preferred `map_` function
3. Close parallel plan

Next, let's repeat the previous analysis with `future_map()`.

```
tic()
square_parallel <- future_map(1:24, slow_square)
toc()
```

```
## 9 sec elapsed
```

furrr

furrr functions make it easy to **parallelize** in just a few steps.

1. Set a "plan" for how the code will be run in parallel
    - Number of cores to use, how to execute tasks
2. Use `future_` version of your preferred `map_` function
3. Close parallel plan

Now that we're done with our parallel session, reset things back to serial:

```
plan("sequential")
```

Benefits of Parallelization

Here we reduced execution time by ~ 1/3 due to some overhead of creating/assigning objects to the cores. However, the benefits of parallel increase substantially with

- Larger objects
- Greater number of repetitions (must be independent tasks)
- More cores

For example, if we run our slow function over the integers 1 to 1,000:

Approach	Time	Time Savings
Serial	1,017.3 Seconds	0%
Parallel, 5 cores	204.33 Seconds	80%
Parallel, 10 cores	103.87 Seconds	90%
Parallel, 20 cores	51.49 Seconds	95%

Progress with *progressr*

For longer tasks, it can be helpful to see progress. We can do this by using the functions within **progressr**.

First, let's add a **progress indicator** to our function.

```r
slow_square_prog ← function(x = 1){
  p() # add in progress indicator
  Sys.sleep(1/2) # wait half a second
  return(x^2)
}
```

Progress with *progressr*

Next, write a **wrapper function** to our future map to add in the progress bar:

```
par_slow_square ← function(x){
  p ← progressor(steps = length(x))
  future_map(x, slow_square_prog)
}
```

Progress with *progressr*

Finally, wrap the function in `with_progress({})` to get a **visible progress bar**.

```
with_progress({
  par_slow_square(1:24)
})
```

Tweaking Progress Bar

There are a lot of **different progress bar options**, including

- Change the shape used in the ASCII progress bar

```
pacman::p_load(cli)
handlers(handler_txtprogressbar(char = cli::col_red(cli::symbol$smiley)))

with_progress({
  par_slow_square(1:24)
})
```

Tweaking Progress Bar

There are a lot of **different progress bar options**, including

- Continuous color bar

```r
handlers("cli")

with_progress({
  par_slow_square(1:24)
})
```

Tweaking Progress Bar

There are a lot of **different progress bar options**, including

- Audible beep at cocnlusion

```
pacman::p_load(beepr)
handlers("cli", "beepr")

with_progress({
  par_slow_square(1:24)
})
```

Tweaking Progress Bar

We can customize the sounds more fully with `handler_beepr()`:

```
sound_path <- paste0(getwd(), "/images/finish.wav")
handlers(list(
  "cli",
          handler_beepr(
            initiate = NA_integer_,
            update = NA_integer_,
            finish = sound_path
             )
           )
         )
with_progress({
  par_slow_square(1:10)
})
```

Table of Contents