

# ETC4500/ETC5450

## Advanced R programming

Week 5: Reactive programming with  
targets and renv



# Outline

- 1 Reactive programming
- 2 Shiny
- 3 targets
- 4 Reproducible environments

# Outline

- 1 Reactive programming
- 2 Shiny
- 3 targets
- 4 Reproducible environments

# Regular (imperative) programming

Consider how code is usually evaluated...

```
a <- 1  
b <- 2  
x <- a + b  
x
```

What is x?

```
a <- -1  
x
```

What is x now?

# Regular (imperative) programming

## Predictable programming

All programming we've seen so far evaluates code in sequential order, line by line.

Since  $x$  was not re-evaluated, its value stays the same even when its inputs have changed.

# Reactive programming

Within a reactive programming paradigm, objects *react* to changes in their inputs and automatically update their value!

# Reactive programming

Within a reactive programming paradigm, objects *react* to changes in their inputs and automatically update their value!



## Disclaimer

Reactive programming is a broad and diverse paradigm, we'll focus only on the basic concepts and how they apply in shiny applications.

# Reactive programming

We can implement *reactivity* with functions & environments.

```
library(rlang)
react <- function(e) new_function(alist(), expr(eval (!!enexpr(e))))
```

We'll learn how this function works later (metaprogramming).

Reactive programming is also smarter about '*invalidation*', results are **cached and reused** if the inputs aren't changed.



# Reactive programming

How does reactive programming differ?

```
a <- 1  
b <- 2  
y <- react(a + b)  
y()
```

What is y?

```
a <- -1  
y()
```

What is y now?

# Reactive programming

💡 (Un)predictable programming?

Reactive programming can be disorienting!

Reactive objects *invalidate* whenever their inputs change, and so its value will be recalculated and stay up-to-date.

# Reactive programming

## Your turn!

```
a <- 1  
b <- 2  
y <- react(a + b)  
y()
```

When was `a + b` evaluated?

How does this differ from ordinary (imperative) code?

# Imperative and declarative programming

## Imperative programming

- Specific commands are carried out immediately.
- Usually direct and exact instructions.
- e.g. read in data from this file.

## Declarative programming

- Specific commands are carried out when needed.
- Expresses higher order goals / constraints.
- e.g. make sure this dataset is up to date every time I see it.

# Imperative and declarative programming

Mastering Shiny: Chapter 3 (Basic Reactivity)

With imperative code you say “Make me a sandwich”.

With declarative code you say “Ensure there is a sandwich in the refrigerator whenever I look inside of it”.

*Imperative code is **assertive**;  
declarative code is **passive-aggressive**.*

# Use cases for reactive programming

## ! Use-less cases

This paradigm is rarely needed or used in R for data analysis.

## 💡 Useful cases

Reactive programming is useful for developing user applications (including web apps!).

In R, the shiny package uses reactive programming for writing app interactivity.

# Outline

- 1 Reactive programming
- 2 Shiny
- 3 targets
- 4 Reproducible environments

# A shiny app

Most shiny apps are organised into several files.

- `ui.R`: The specification of the user interface
- `server.R`: The reactive code that defines app behaviour
- `global.R`: Static global objects used across app
- `www/`: Folder for your web data (images, css, js, etc.)

Simple apps can consist of only an `app.R` script.




# Hello shiny!

## Follow along!

Create a shiny app. Save this code as `app.R`.

```
library(shiny)
ui <- fluidPage(
  textInput("name", "Enter your name: "),
  textOutput("greeting")
)
server <- function(input, output, session) {
  output$greeting <- renderText({
    sprintf("Hello %s", input$name)
  })
}
shinyApp(ui, server)
```

# Hello shiny!

 Follow along!

Launch the app by clicking **Run App**.

Use the text input field and see how the webpage changes.

Look at the server code to see how it 'reacts'.

# Shiny reactivity

Reactivity in shiny comprises of:

- Reactive **sources** (inputs):

UI inputs `input*()` and values `reactiveValues()`

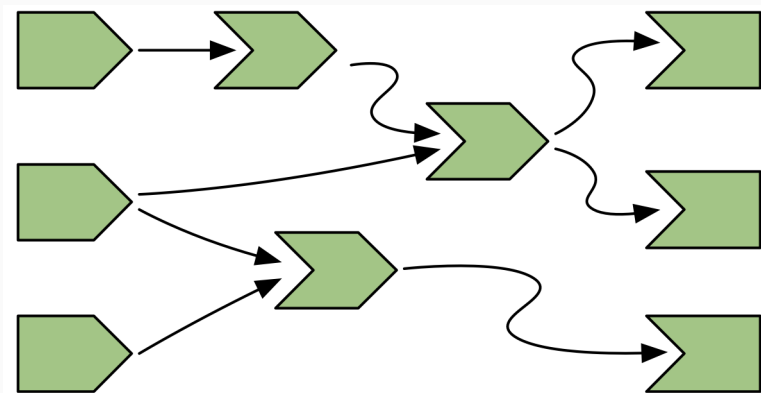
- Reactive **conductors** (intermediates):

Expressions `reactive()` and events `eventReactive()`

- Reactive **endpoints** (results):

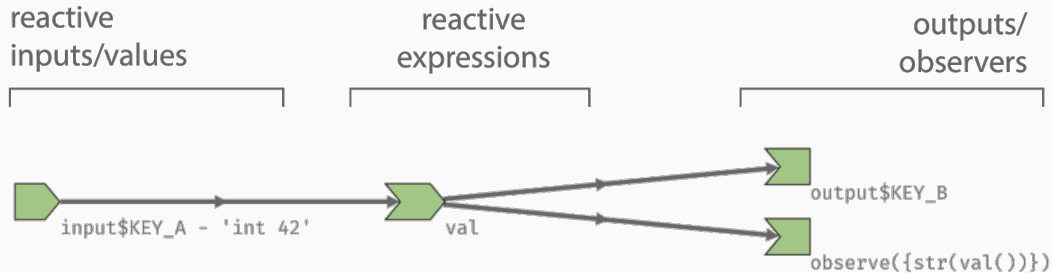
UI outputs `render*()` and side-effects `observe()`

# Reactive graphs



The reactivity of an app can be visualised with a graph.

# Reactive graphs



The graph shows relationships between reactive elements.

The `reactlog` package allows you to visualise an app's **reactive graph**.

To **enable logging** of an app's behaviour, run:

```
reactlog::reactlog_enable()
```


Then **start, use, and stop your app** to fill the log.

View the log with:

```
shiny::reactlogShow()
```

Or while your Shiny app is running, press the key combination Ctrl+F3 (Mac: Cmd+F3) to see the reactive log.

# Hello *reactlog*!

 Follow along!

Create a reactive log of the *hello shiny* app.

Start reactlog, then open the app and enter your name.

Close the app and view the log, see how the app reacts to changes to the input text.

# Reactive expressions

Reactive expressions are used in the shiny server as intermediate calculations.

They are expressions wrapped with `reactive()`.

For example:

```
simulation <- reactive(rnorm(input$n_samples))
```



# Reactive expressions

Reactive expressions are used in the shiny server as intermediate calculations.

They are expressions wrapped with `reactive()`.


For example:

```
simulation <- reactive(rnorm(input$n_samples))
```

The up-to-date value is obtained with `simulation()`.

Whenever the input ID `n_samples` changes, the reactive expression `simulation` *invalidates*.

# Reactive expressions

 Follow along!

Use a reactive expression to convert the name to ALLCAPS.

Look at the reactive graph and see how it changes.

# Preventing reactivity


Equally important to telling shiny **how** to react to changes, is describing **when** reactions should (not) occur.

# Preventing reactivity

Equally important to telling shiny **how** to react to changes, is describing **when** reactions should (not) occur.

The most useful way to prevent reactivity is with `req()`.  
It is similar to `stop()`, silently ending the reactive chain.  
`req()` *'requires'* inputs to be 'truthy' (not FALSE or empty).

# Preventing reactivity

 Follow along!

Use `req()` to prevent reactivity until text is entered.

Update `req()` to require at least 3 characters inputted.

# Preventing reactivity

Other ways reactivity might be prevented include:

## ■ Event reactivity

- ▶ `eventReactive(rnorm(input$n_samples), input$go)`
- ▶ `observeEvent(input$go, message("Go!"))`

## ■ Rate limiting

- ▶ `throttle(reactive())`: limits update frequency
- ▶ `debounce(reactive())`: waits for changes to stop

# Outline

- 1 Reactive programming
- 2 Shiny
- 3 targets
- 4 Reproducible environments

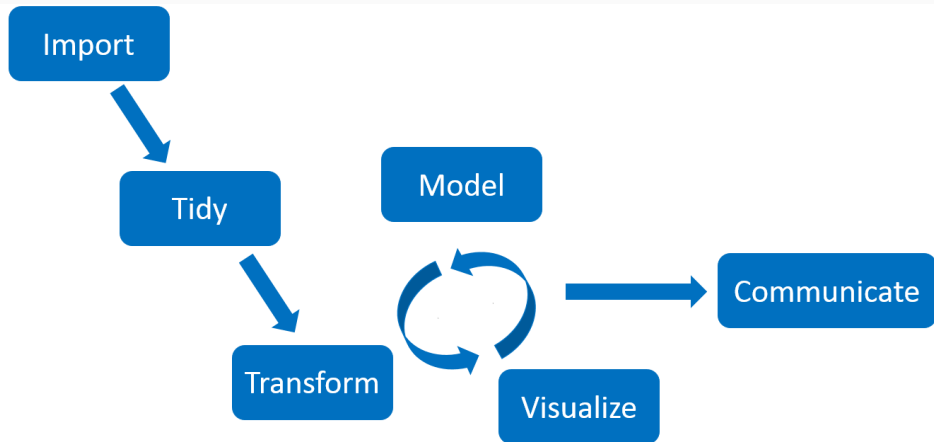
# targets: reproducible computation at scale



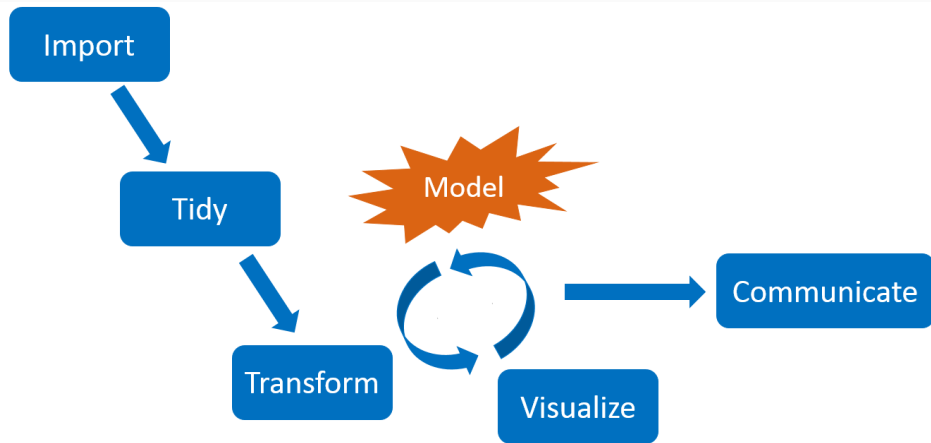
- Supports a clean, modular, function-oriented programming style.
- Learns how your pipeline fits together.
- Runs only the necessary computation.
- Abstracts files as R objects.
- Similar to Makefiles, but with R functions.



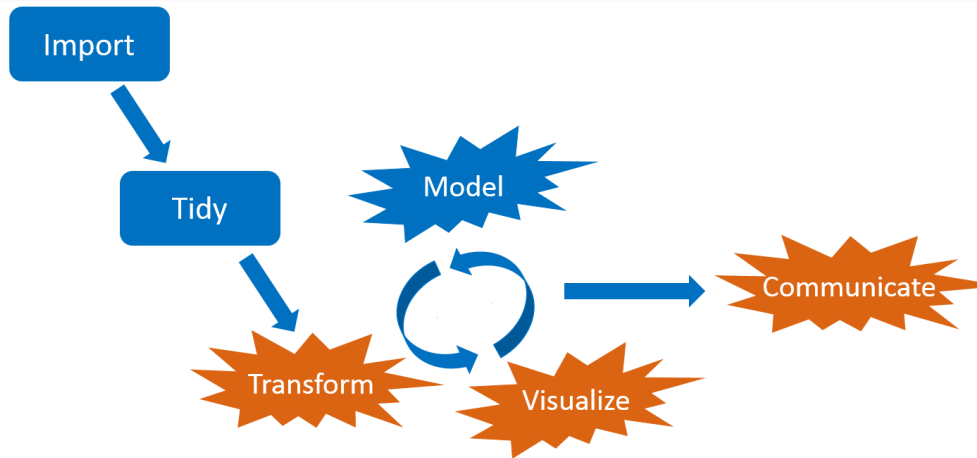
# Interconnected tasks



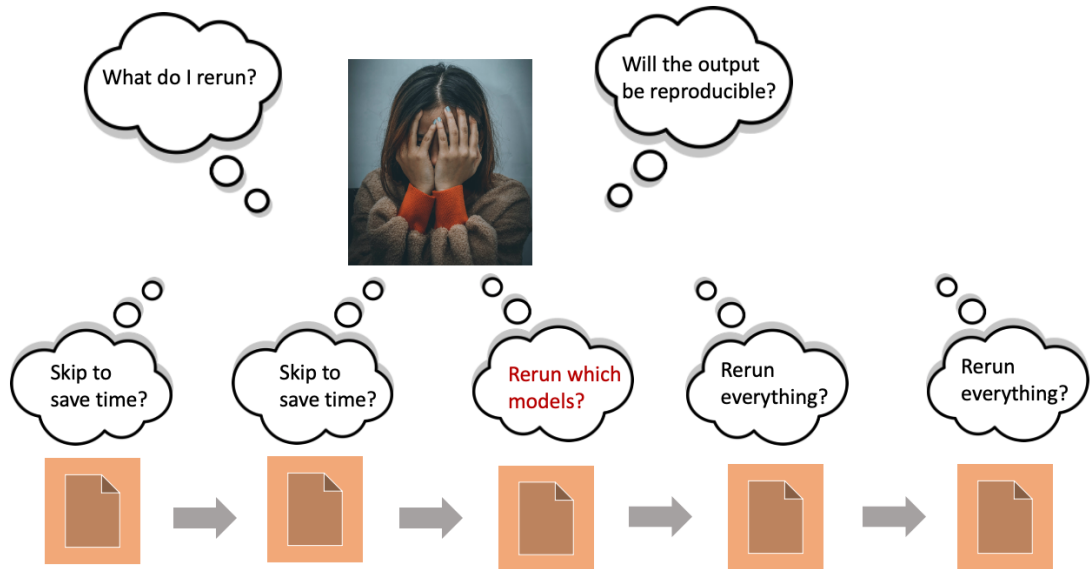
# Interconnected tasks



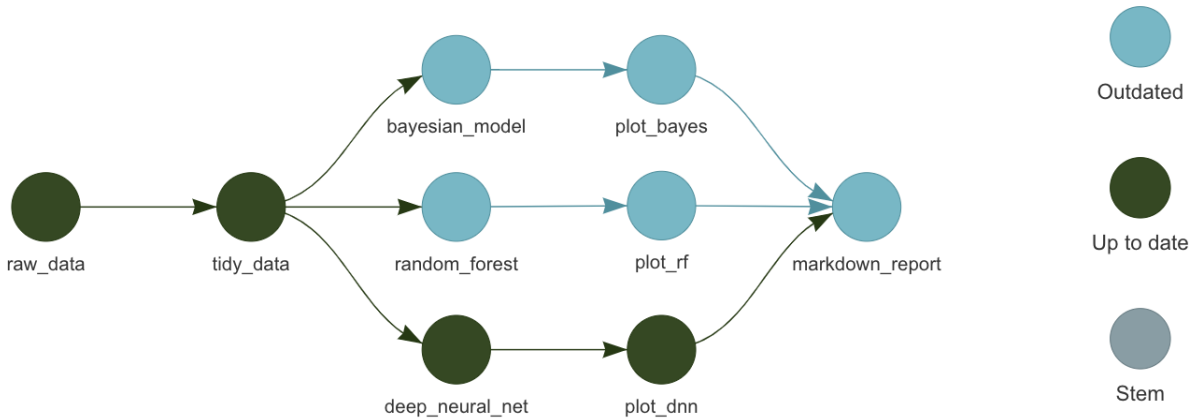
# Interconnected tasks



# Dilemma: short runtimes or reproducible results?



# Let a pipeline tool do the work



- Save time while ensuring computational reproducibility.
- Automatically skip tasks that are already up to date.

# Typical project structure

```
_targets.R # Required top-level configuration file.  
R/  
└─ functions.R  
data/  
└─ my_data.csv
```

## **\_targets.R**

```
library(targets)  
tar_source() # source all files in R folder  
tar_option_set(packages = c("tidyverse", "fable"))  
list(  
  tar_target(my_file, "data/my_data.csv", format = "file"),  
  tar_target(my_data, read_csv(my_file)),  
  tar_target(my_model, model_function(my_data))  
)
```

# Generate \_targets.R in working directory

```
library(targets)  
tar_script()
```

# Useful targets commands

- `tar_make()` to run the pipeline.
- `tar_make(starts_with("fig"))` to run only targets starting with “fig”.
- `tar_read(object)` to read a target.
- `tar_load(object)` to load a target.
- `tar_load_everything()` to load all targets.
- `tar_manifest()` to list all targets
- `tar_visnetwork()` to visualize the pipeline.
- `tar_destroy()` to remove all targets.
- `tar_outdated()` to list outdated targets.



# Debugging

Errored targets to return NULL so pipeline continues.

```
tar_option_set(error = "null")
```

# Debugging

Errored targets to return NULL so pipeline continues.

```
tar_option_set(error = "null")
```

See error messages for all targets.

```
tar_meta(fields = error, complete_only = TRUE)
```

# Debugging

Errored targets to return NULL so pipeline continues.

```
tar_option_set(error = "null")
```

See error messages for all targets.

```
tar_meta(fields = error, complete_only = TRUE)
```

See warning messages for all targets.

```
tar_meta(fields = warnings, complete_only = TRUE)
```

# Debugging

- Try loading all available targets: `tar_load_everything()`.  
Then run the command of the errored target in the console.
- Pause the pipeline with `browser()`
- Use the debug option: `tar_option_set(debug = "target_name")`
- Save the workspaces:
  - ▶ `tar_option_set(workspace_on_error = TRUE)`
  - ▶ `tar_workspaces()`
  - ▶ `tar_workspace(target_name)`

# Random numbers

- Each target runs with its own seed based on its name and the global seed from `tar_option_set(seed = ???)`
- So running only some targets, or running them in a different order, will not change the results.

# Folder structure

```
├── .git/
├── .Rprofile
├── .Renviron
├── renv/
├── index.Rmd
├── _targets/
├── _targets.R
├── _targets.yaml
├── R/
│   ├── functions_data.R
│   ├── functions_analysis.R
│   └── functions_visualization.R
├── data/
└── input_data.csv
```

# targets with quarto

```
library(targets)
library(tarchetypes)
tar_source() # source all files in R folder
tar_option_set(packages = c("tidyverse", "fable"))
list(
  tar_target(my_file, "data/my_data.csv", format = "file"),
  tar_target(my_data, read_csv(my_file)),
  tar_target(my_model, model_function(my_data))
  tar_quarto(report, "file.qmd", extra_files = "references.bib")
)
```

①

②

- ① Load tarchetypes package for quarto support.
- ② Add a quarto target.

# Exercise

- Add a targets workflow to your quarto document.
- Create a visualization of the pipeline network using `tar_visnetwork()`.



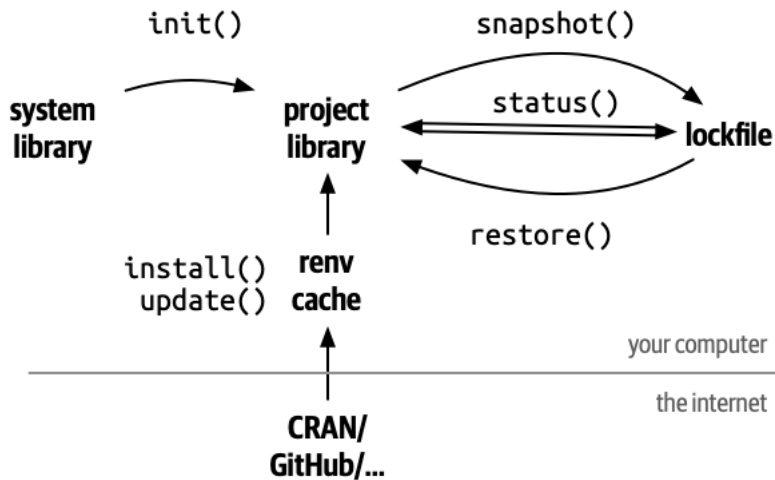
# Outline

- 1 Reactive programming
- 2 Shiny
- 3 targets
- 4 Reproducible environments

# Reproducible environments

- To ensure that your code runs the same way on different machines and at different times, you need the computing environment to be the same.
  - 1 Operating system
  - 2 System components
  - 3 R version
  - 4 R packages
- Solutions for 1–4: Docker, Singularity, containerit, rang
- Solutions for 4: packrat, checkpoint, renv

# renv package



- `renv::init()` : initialize a new project with a new environment. Adds:
  - ▶ `renv/library` contains all packages used in project
  - ▶ `renv.lock` contains metadata about packages used in project
  - ▶ `.Rprofile` run every time R starts.
- `renv::snapshot()` : save the state of the project to `renv.lock`.
- `renv::restore()` : restore the project to the state saved in `renv.lock`.

# renv package

- `renv` uses a package cache so you are not repeatedly installing the same packages in multiple projects.
- `renv::install()` can install from CRAN, Bioconductor, GitHub, Gitlab, Bitbucket, etc.
- `renv::update()` gets latest versions of all dependencies from wherever they were installed from.
- Only R packages are supported, not system dependencies, and not R itself.
- `renv` is not a replacement for Docker or Singularity.
- `renv::deactivate(clean = TRUE)` will remove the `renv` environment.