MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450
Advanced R programming

Week 4: Literate programming with Quarto

Outline

1. Programming paradigms
2. Reactive programming
3. Shiny
4. Literate programming
5. roxygen2
6. Rmarkdown
7. Quarto

Outline

1. Programming paradigms
2. Reactive programming
3. Shiny
4. Literate programming
5. roxygen2
6. Rmarkdown
7. Quarto

Outline

Programming paradigms

Functional programming (W5)

- Functions are created and used like any other object.
- Output should only depend on the function's inputs.

Programming paradigms

Functional programming (W5)

- Functions are created and used like any other object.
- Output should only depend on the function's inputs.

Object-oriented programming (W6-W7)

- Functions are associated with object types.
- Methods of the same 'function' produce object-specific output.

Programming paradigms

Reactive programming (W8)

- Objects are expressed using code based on inputs.
- When inputs change, the object's value updates.

Literate programming (W8)

- Natural language is interspersed with code.
- Aimed at prioritising documentation/comments.
- Now used to create reproducible reports/documents.

Outline

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

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()`

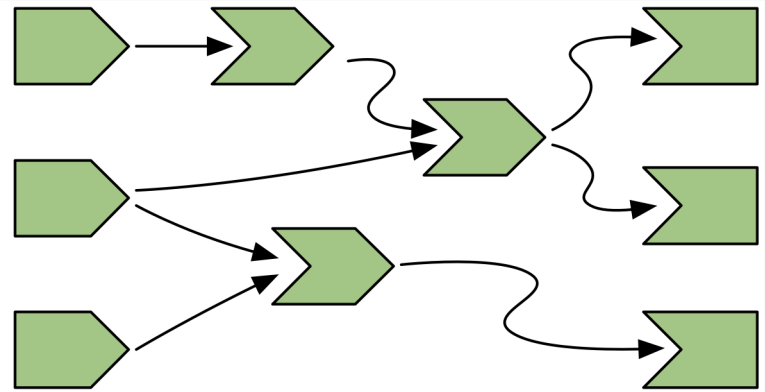
- Reactive **conductors** (intermediates):

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

- Reactive **endpoints** (results):

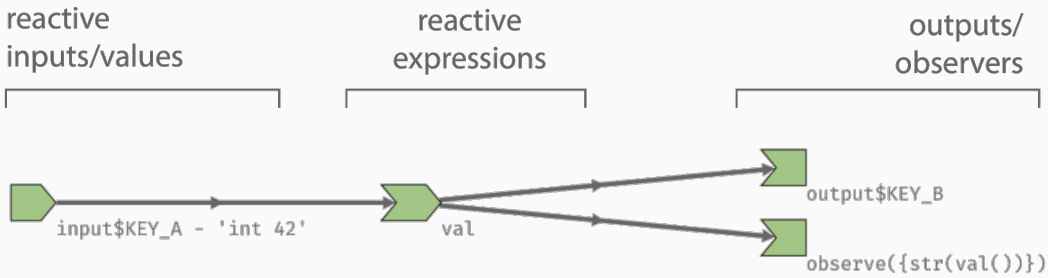
  UI outputs `render*()` and side-effects `observe()`

Reactive graphs



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

Reactive graphs



The graph shows relationships between reactive elements.

reactlog

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

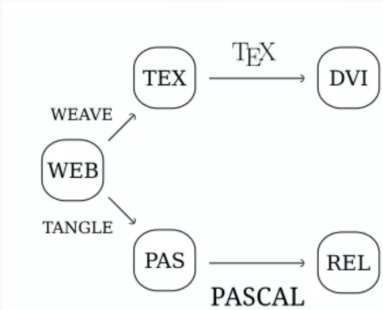
Other ways reactivity might be prevented include:

- 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

Literate programming

- Due to Donald Knuth (Stanford), 1984
- A script or document that contains an explanation of the program logic in a natural language (e.g. English), interspersed with snippets of source code, which can be compiled and rerun.
- Generates two representations from a source file: formatted documentation and “tangled” code.



Literate programming

- As a programming approach, it never quite caught on.
- But it has become the standard approach for reproducible documents.

Literate programming examples

- WEB (combining Pascal and TeX)
- roxygen2 comments
  - technically documentation generation rather than literate programming
  - documentation embedded in code, rather than code embedded in documentation
- Sweave documents
- Jupyter notebooks
- Rmarkdown documents
- Quarto documents

Outline

roxygen2

- roxygen2 documentation are just comments to R.
- roxygen2::roxygenize():
  - generates documentation from these comments in the form of Rd files
  - adds relevant lines to the NAMESPACE file.
- roxygen2::roxygenize() is called by devtools::document().
- Advantage: keeps documentation with the code. More readable, less chance for errors.

Outline

Markdown syntax

Markdown: a “markup” language for formatting text.

- Headings:

```
# Heading 1
## Heading 2
```

- **Bold**: `**bold**`.
- *Italic*: `*italic*`.
- Blockquotes:

```
> blockquote.
```

Markdown and Rmarkdown

- Markdown (markup language):
  - Extension either `.md` or `.markdown`.
  - Used in many places on the web, in note-taking apps, etc.
- Rmarkdown (markup language):
  - an extension of markdown that allows for embedded R code chunks.
  - Extension `.Rmd`.
- Rmarkdown (package):
  - an R package that allows for the conversion of `.Rmd` files to other formats.

Rmarkdown files

- Structure:
  1. YAML header
  2. Markdown content
  3. R code chunks surrounded by ```` ```{r} ```` and ```` ``` ````
  4. Inline R surrounded by `` `r `` and `` ` ``
- Rmarkdown documents can be compiled to HTML, PDF, Word, and other formats
- Compile with `rmarkdown::render("file.Rmd")`

Rmarkdown, knitr and pandoc

- `rmarkdown::render()`
  - Uses `knitr` to run all code chunks, and "knit" the results into a markdown file (replacing R chunks with output).
  - Uses `pandoc` to convert the markdown file to the desired output format.
  - If PDF output is desired, LaTeX then converts the tex file (from pandoc output) to pdf.

knitr functions

- `knitr::knit()`: knits a single Rmd file — runs all code chunks and replaces them with output in a markdown file.
- `knitr::purl()`: extracts all R code from an Rmd file and saves it to a new file.
- `knitr::spin()`: knits a specially formatted R script file into an Rmd file.

Rmarkdown packages

- rmarkdown (to html, pdf, docx, odt, rtf, md, etc.)
- bookdown (to html, pdf, epub)
- blogdown (to html) – uses hugo rather than pandoc
- xaringan (to html) – uses remark.js rather than pandoc
- beamer (to pdf)
- rticles (to pdf)
- tufte (to html, pdf)
- vitae (to pdf)
- distill (to html)
- flexdashboard (to html)

Some chunk options

- `eval`: whether to evaluate the code chunk
- `echo`: whether to display the code chunk
- `include`: whether to include the code chunk in the output
- `results = 'hide'` hides printed output.
- `results = 'asis'` includes the output as is.
- `message`: whether to display messages
- `warning`: whether to display warnings
- `error = TRUE`: continue even if code returns an error.
- `fig.cap`: caption for the figure
- `fig.width`, `fig.height`: width and height of the figure
- `cache`: whether to cache the code chunk

Global chunk options

````
```{r setup, include=FALSE}
knitr::opts_chunk$set(
  comment = "#>",
  collapse = TRUE,
  echo = FALSE,
  message = FALSE,
  warning = FALSE
)
```
````

- The chunk named `setup` will be run before any other chunks.

Debugging

- The Rmarkdown document is compiled in a different environment from your R console.
- If you get an error, try running all chunks (Ctrl+Alt+R).
- If you can't reproduce the error, check the working directory (add `getwd()` in a chunk).
- Try setting `error = TRUE` on problem chunk to help you diagnose what happens. (But change it back!)
- Look at the intermediate files (`.md` or `.tex`) to see what is happening.

Caching

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(cache = TRUE)
```
```

or by chunk:

```
```{r, cache = TRUE}
```
```

Caching

- When evaluating code chunks, knitr will save the results of chunks with caching to files to be reloaded in subsequent runs.
- Caching is useful when a chunk takes a long time to run.
- It will re-run if the code in the chunk changes in any way (even comments or spacing).
- Beware of inherited objects from earlier chunks. A chunk will not re-run if inherited objects change without explicit dependencies.
- Beware of dependence on external files.

Caching

````
```{r chunk1, cache = TRUE}
x <- 1
```


```{r chunk2, cache = TRUE, dependson = "chunk1"}
y <- x*3
```
````

Caching

```
```{r chunk1, cache = TRUE}
x <- 1
```


```{r chunk2, cache = TRUE, dependson = "chunk1"}
y <- x*3
```
```

```
```{r chunk1, cache = TRUE}
x <- 1
```


```{r chunk2, cache = TRUE, cache.extra = x}
y <- x*3
```
```

Caching

Cache will be rebuilt if:

- Chunk options change except `include`
- Any change in the code, even a space or comment
- An explicit dependency changes

Do not cache if:

- setting R options like `options('width')`
- setting knitr options like `opts_chunk$set()`
- loading packages via `library()` if those packages are used by uncached chunks

Caching with random numbers

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(cache.extra = knitr::rand_seed)
```
```

- `rand_seed` is an unevaluated expression.
- Each chunk will check if `.Random.seed` has been changed since the last run.
- If it has, the chunk will be re-run.

Some caching options

- `cache.comments` If `FALSE`, changing comments does not invalidate the cache.
- `cache.rebuild` If `TRUE`, the cache will be rebuilt even if the code has not changed. e.g.,
  `cache.rebuild = !file.exists("some-file")`
- `dependson` A character vector of labels of chunks that this chunk depends on.
- `my_new_option` A new option that you can use in your code to invalidate the cache. e.g., `my_new_option = c(x,y)`
- `autodep` If `TRUE`, the dependencies are automatically determined. (May not be reliable.)

Caching

Build automatic dependencies among chunks

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(cache=TRUE, autodep = TRUE)
```
```

Make later chunks depend on previous chunks

```
```{r setup, include=FALSE}
dep_prev() # Don't use with `autodep = TRUE`
```
```

Child documents

```
```{r, child=c('one.Rmd', 'two.Rmd')}
```
```

Child documents

```
```{r, child=c('one.Rmd', 'two.Rmd')}
```
```

Conditional inclusion

```
```{r, child = if(condition) 'file1.Rmd' else 'file2.Rmd'}
```
```

Child documents

```
```{r, child=c('one.Rmd', 'two.Rmd')}
```
```

Conditional inclusion

```
```{r, child = if(condition) 'file1.Rmd' else 'file2.Rmd'}
```
```

R Script files

```
```{r, file = c("Rscript1.R", "Rscript2.R")}
```
```

- Better than `source("Rscript1.R")` because output of script included and dependencies tracked.

Other language engines

````
```{python}
print("Hello Python!")
```
````

````
```{stata}
sysuse auto
summarize
```
````

- Python and Stata need to be installed with executables on PATH

Other language engines

```
names(knitr::knit_engines$get())
```

```
 [1] "awk"       "bash"      "coffee"    "gawk"      "groovy"
 [6] "haskell"   "lein"      "mysql"     "node"      "octave"
[11] "perl"      "php"       "psql"      "Rscript"   "ruby"
[16] "sas"       "scala"     "sed"       "sh"        "stata"
[21] "zsh"       "asis"      "asy"       "block"     "block2"
[26] "bslib"     "c"         "cat"       "cc"        "comment"
[31] "css"       "ditaa"     "dot"       "embed"     "eviews"
[36] "exec"      "fortran"   "fortran95" "go"        "highlight"
[41] "js"        "julia"     "python"    "R"         "Rcpp"
[46] "sass"      "scss"      "sql"       "stan"      "targets"
[51] "tikz"      "verbatim"  "ojs"       "mermaid"   "glue"
[56] "glue_sql"  "gluesql"
```

Outline

Quarto

- Generalization of Rmarkdown (not dependent on R)
- Supports R, Python, Javascript and Julia chunks by using either `knitr`, `jupyter` or `ObservableJS` engines.
- More consistent yaml header and chunk options.
- Many more output formats, and many more options for customizing format.
- Heavier reliance on pandoc Lua filters
- Uses pandoc templates for extensions

Choose your engine

Specify the engine in the yaml header:

```
---
engine: knitr
---
```

```
---
engine: jupyter
jupyter: python3
---
```

Default: If any `{r}` blocks found, use `knitr` engine; otherwise use `jupyter` (with kernel determined by first block).

Execute options

- `execute` option in yaml header can be used instead of a `setup` chunk:

```yaml
execute:
  cache: true
  echo: false
  warning: false
```

- `setup` chunk still allowed.

Chunk options

Rmarkdown syntax recognized for R chunks.

More consistent chunk options use the hash-pipe `#|`

````
```{r}
#| label: fig-chunklabel
#| fig-caption: My figure
#| fig-width: 6
#| fig-height: 4
mtcars |>
  ggplot(aes(x = mpg, y = wt)) +
  geom_point()
```
````

Reference the figure using `@fig-chunklabel`.

Chunk options

- Quarto consistently uses hyphenated options (`fig-width` rather than `fig.width`)
- The Rmarkdown `knitr` options are recognized for backwards compatibility.
- Options that are R expressions need to be prefaced by `!expr`

````
```{r}
#| fig-cap: !expr paste("My figure", 1+1)
```
````

Extensions and templates

- Quarto extensions modify and extend functionality.
- They are stored locally, in the `_extensions` folder alongside the qmd document.
- See https://quarto.org/docs/extensions/ for a list.
- Templates are extensions used to define new output formats.
- Journal templates at https://quarto.org/docs/extensions/listing-journals.html
- Monash templates at https://robjhyndman.com/hyndsight/quarto_templates.html

quarto on the command line

- `quarto render` to render a quarto or Rmarkdown document.
- `quarto preview` to preview a quarto or Rmarkdown document.
- `quarto add <gh-org>/<gh-repo>` to add an extension from a github repository.
- `quarto update <gh-org>/<gh-repo>` to update an extension
- `quarto remove <gh-org>/<gh-repo>` to remove an extension
- `quarto list extensions installed`
- `quarto use template <gh-org>/<gh-repo>` to use existing repo as starter template.

Add a custom format

From the CLI: `quarto add numbats/monash-quarto-memo`

Add a custom format

From the CLI: `quarto add numbats/monash-quarto-memo`

New folder/files added

```
├── _extensions
│   └── numbats
│       └── memo
│           └── ...
```

Add a custom format

From the CLI: `quarto add numbats/monash-quarto-memo`

New folder/files added

```
├── _extensions
│   └── numbats
│       └── memo
│           └── ...
```

Update YAML

```
---
title: "My new file using the `memo-pdf` format"
format: memo-pdf
---
```

Exercise

- Set up a new project.
- Create a quarto document using an html format.
- Add a code chunk to generate a figure with a caption.
- Reference the figure in the text using `@fig-chunklabel`.
- Add the monash memo extension and generate a pdf output.