Embedding Computations & Code

Session 3

Mickaël CANOUIL, *Ph.D.*

pro@mickael.canouil.dev

Independent Contractor

Saturday, the 7th of February, 2026

 |

- Session 3: Embedding Computations & Code
- Session Objectives
- Learning Objectives

Session Overview

 |

Session 3: Embedding Computations & Code

- **Computing Environments**
  - Setup R, Python, and Julia for Quarto.
  - Engine configuration and code cell structure.
  - Understanding multi-language workflows.
- **Cache and Freeze**
  - Cache for development speed vs freeze for collaboration.
  - Understanding Jupyter and Knitr caching systems.
  - CLI commands for cache and freeze management.
- **Execution Options**
  - Code cell options using comment + pipe syntax.
  - Control code visibility, evaluation, and output.
  - Global execution settings in YAML.
- **Parameters**
  - Creating dynamic document variations.
  - Defining parameters in Python and R.
  - Command-line parameter passing and YAML files.

Session Objectives

This session explores Quarto's computational capabilities, focusing on seamless multi-language workflows, code cell options, and integration patterns for R and Python data science stacks.

Learning Objectives

 |

Learning Objectives

By the end of this session, participants will be able to:

 |

Learning Objectives

By the end of this session, participants will be able to:

- Execute R, Python, Julia code.

 |

Learning Objectives

By the end of this session, participants will be able to:

- Execute R, Python , Julia code.
- Configure code cells using “comment symbol” + “pipe” (|) syntax.

 |

Learning Objectives

By the end of this session, participants will be able to:

- Execute R, Python , Julia code.
- Configure code cells using “comment symbol” + “pipe” (|) syntax.
- Control code visibility and execution.

Learning Objectives

By the end of this session, participants will be able to:

- Execute R, Python, Julia code.
- Configure code cells using “comment symbol” + “pipe” (|) syntax.
- Control code visibility and execution.
- Cache computations for improved performance.

 |

- Setup R, Python, and/or Julia for Quarto
- Engine Configuration
- What is a Code Cell?
- Code Cell Structure
- Why Use Code Cells?

Computing Environments

 |

Setup R, Python, and/or Julia for Quarto

R | Python (uv) | Julia

R

```r
# Install renv if not already installed
install.packages("renv")

# Initialise project
renv::init()

# Install packages
renv::install("rmarkdown")

# Save current state
renv::snapshot()
```

 |

Setup R, Python, and/or Julia for Quarto

R | Python (uv) | Julia

BASH

```bash
# Install uv (if not installed)
pip install uv

# Initialise project
uv init --no-package --vcs none

# Create virtual environment
uv venv

# Activate environment
source .venv/bin/activate

# Add packages
uv add jupyter papermill
```

Setup R, Python, and/or Julia for Quarto

R Python (uv) **Julia**

JULIA

```julia
# Start Julia in your project directory
using Pkg

# Activate current directory as project
Pkg.activate(".")

# Add packages
Pkg.add("IJulia")

# Instantiate environment
Pkg.instantiate()
```

Engine Configuration

 |

Engine Configuration

Configure engines in your document header:

 |

Engine Configuration

Configure engines in your document header:

- `knitr`

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

Engine Configuration

Configure engines in your document header:

- `knitr`
- `jupyter`

YAML

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

YAML

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

 |

Engine Configuration

Configure engines in your document header:

- `knitr` 
- `jupyter` 
- `julia` 

YAML

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

YAML

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

YAML

```yaml
---
engine: julia
---
```

 |

Engine Configuration

Configure engines in your document header:

- `knitr` 
- `jupyter` 
- `julia` 

Tip

Set explicitly the engine in your document header instead of relying on auto-detection which does not work for inline code (*i.e.*, `` `{language} ...` ``).

YAML

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

YAML

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

YAML

```yaml
---
engine: julia
---
```

What is a Code Cell?

 |

What is a Code Cell?

A **code cell** (also called a **code chunk**) is a section of executable code (defined by ```` ```{language} ````) embedded within your Quarto document.

What is a Code Cell?

A **code cell** (also called a **code chunk**) is a section of executable code (defined by `` ```{language} ``) embedded within your Quarto document.

Code cells allow you to:

- Execute programming code directly in your document.

What is a Code Cell?

A **code cell** (also called a **code chunk**) is a section of executable code (defined by `` ```{language} ``) embedded within your Quarto document.

Code cells allow you to:

- Execute programming code directly in your document.
- Display both the code and its output.

What is a Code Cell?

A **code cell** (also called a **code chunk**) is a section of executable code (defined by `` ```{language} ``) embedded within your Quarto document.

Code cells allow you to:

- Execute programming code directly in your document.
- Display both the code and its output.
- Create reproducible analyses and reports.

 |

Code Cell Structure

QMD

````
```{python}                                    (1)
#| label: my-plot                              (2)
#| echo: true
#| fig-cap: "Sample visualization"
                                               (3)
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Sine Wave")
plt.show()
```
````

Code Cell Structure

Computing language (*e.g.*, `python`, `r`, `julia`).

````
1  ```{python}
2  #| label: my-plot
3  #| echo: true
4  #| fig-cap: "Sample visualization"
5
6  import matplotlib.pyplot as plt
7  import numpy as np
8
9  x = np.linspace(0, 10, 100)
10 y = np.sin(x)
11
12 plt.plot(x, y)
13 plt.title("Sine Wave")
14 plt.show()
15 ```
````

1
2
3

Code Cell Structure

````
```{python}
#| label: my-plot
#| echo: true
#| fig-cap: "Sample visualization"

import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Sine Wave")
plt.show()
```
````

Code cell options (*e.g.*, `label`, `echo`, `eval`).

Code Cell Structure

QMD

````
1  ```{python}
2  #| label: my-plot
3  #| echo: true
4  #| fig-cap: "Sample visualization"
5
6  import matplotlib.pyplot as plt
7  import numpy as np
8
9  x = np.linspace(0, 10, 100)
10 y = np.sin(x)
11
12 plt.plot(x, y)
13 plt.title("Sine Wave")
14 plt.show()
15 ```
````

1
2
3

Code content (your actual programming code).

Code Cell Structure

QMD

```
1  ```{python}                                                    (1)
2  #| label: my-plot                                             (2)
3  #| echo: true
4  #| fig-cap: "Sample visualization"
5
6  import matplotlib.pyplot as plt                                (3)
7  import numpy as np
8
9  x = np.linspace(0, 10, 100)
10 y = np.sin(x)
11
12 plt.plot(x, y)
13 plt.title("Sine Wave")
14 plt.show()
15 ```
```

Why Use Code Cells?

 |

Why Use Code Cells?

Reproducibility

- Code and results stay together.
- Easy to re-run analyses.
- Version control friendly.

 |

Why Use Code Cells?

Reproducibility

- Code and results stay together.
- Easy to re-run analyses.
- Version control friendly.

Documentation

- Combine narrative and code.
- Show methodology clearly.
- Create living documents.

 |

- Setting Execution Options Globally
- Code Cells Options

Execution Options

 |

Setting Execution Options Globally

```yaml
execute:
  ...
```

Table 1: `execute` options

Option	Description
`eval`	Evaluate the code chunk (if `false`, just echos the code into the output).
`echo`	Include the source code in output
`output`	Include the results of executing the code in the output (`true`, `false`, or `asis` to indicate that the output is raw markdown and should not have any of Quarto's standard enclosing markdown).
`warning`	Include warnings in the output.
`error`	Include errors in the output (note that this implies that errors executing code will not halt processing of the document).
`include`	Catch all for preventing any output (code or results) from being included (e.g., `include: false` suppresses all output from the code block).
`renderings`	Specify rendering names for the plot or table outputs of the cell, e.g., `[light, dark]`. See Renderings.

Code Cells Options

 |

Code Cells Options

The “comment symbol” + “pipe” (`|`) syntax allows to pass options to code cells.

Code Cells Options

The “comment symbol” + “pipe” (`#|`) syntax allows to pass options to code cells.

Code Visibility Control:

QMD

```
1 ```{r}
2 #| echo: false
3 #| eval: false
4 #| include: false
5 #| code-fold: true
6 ```
```

 |

Code Cells Options

The “comment symbol” + “pipe” (`#|`) syntax allows to pass options to code cells.

Code Visibility Control:

QMD

```
1 ```{r}
2 #| echo: false
3 #| eval: false
4 #| include: false
5 #| code-fold: true
6 ```
```

Hide code, show outp

Code Cells Options

The “comment symbol” + “pipe” (`#|`) syntax allows to pass options to code cells.

Code Visibility Control:

QMD

```
1 ```{r}
2 #| echo: false
3 #| eval: false
4 #| include: false
5 #| code-fold: true
6 ```
```

Show code, don’t execute.

Code Cells Options

The “comment symbol” + “pipe” (`#|`) syntax allows to pass options to code cells.

Code Visibility Control:

QMD

```
1 ```{r}
2 #| echo: false
3 #| eval: false
4 #| include: false
5 #| code-fold: true
6 ```
```

Execute code, hide everything.

Code Cells Options

The “comment symbol” + “pipe” (`#|`) syntax allows to pass options to code cells.

Code Visibility Control:

QMD

```
1 ```{r}
2 #| echo: false
3 #| eval: false
4 #| include: false
5 #| code-fold: true
6 ```
```

1 2 3

Collapsible code bloc

Code Cells Options

The “comment symbol” + “pipe” (`#|`) syntax allows to pass options to code cells.

Code Visibility Control:

QMD

```
1 ```{r}
2 #| echo: false
3 #| eval: false
4 #| include: false
5 #| code-fold: true
6 ```
```

 |

Code Cells Options

Output Control:

QMD

```
1 ```{r}
2 #| warning: false
3 #| message: false
4 #| error: true
5 #| output: false
6 ```
```

1. `warning: false`
2. `message: false`
3. `error: true`
4. `output: false`

 |

Code Cells Options

Output Control:

QMD

```
1 ```{r}
2 #| warning: false
3 #| message: false
4 #| error: true
5 #| output: false
6 ```
```

Suppress warnings

Code Cells Options

Output Control:

QMD

```
1 ```{r}
2 #| warning: false
3 #| message: false
4 #| error: true
5 #| output: false
6 ```
```

1
2
3
4

Suppress messages

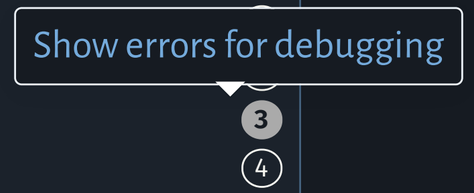
 |

Code Cells Options

Output Control:

QMD

```
1 ```{r}
2 #| warning: false
3 #| message: false
4 #| error: true
5 #| output: false
6 ```
```



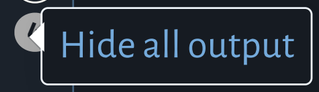
 |

Code Cells Options

Output Control:

QMD

```
1 ```{r}
2 #| warning: false
3 #| message: false
4 #| error: true
5 #| output: false
6 ```
```



 |

Code Cells Options

Output Control:

QMD

```
1 ```{r}
2 #| warning: false
3 #| message: false
4 #| error: true
5 #| output: false
6 ```
```

Cache and Freeze

- Two Systems
- Cache: Development Speed
- Freeze: Collaboration Control
- Key Behavioural Differences
- When to Use Each
- Essential CLI Commands
- Golden Rules

 |

Two Systems

 |

Two Systems

Cache: Stores computation results

- Speeds up development.
- Works during iteration.
- Engine-specific behaviour.

 |

Two Systems

Cache: Stores computation results

- Speeds up development.
- Works during iteration.
- Engine-specific behaviour.

Freeze: Prevents re-execution

- Enables collaboration.
- Project-wide control.
- Version control friendly.

 |

Cache: Development Speed

 |

Cache: Development Speed

Store expensive computation results to avoid re-running during development.

 |

Cache: Development Speed

Store expensive computation results to avoid re-running during development.

YAML

```yaml
execute:
  cache: true
```

 |

Cache: Development Speed

Store expensive computation results to avoid re-running during development.

YAML

```yaml
execute:
  cache: true
```

Two Types:

 |

Cache: Development Speed

Store expensive computation results to avoid re-running during development.

```yaml
execute:
  cache: true
```

Two Types:

- **Jupyter Cache:** Document-level (*all-or-nothing*).

 |

Cache: Development Speed

Store expensive computation results to avoid re-running during development.

YAML

```yaml
execute:
  cache: true
```

Two Types:

- **Jupyter Cache:** Document-level (*all-or-nothing*).
- **Knitr Cache:** Cell-level (*granular control*).

 |

Freeze: Collaboration Control

 |

Freeze: Collaboration Control

Prevent documents from re-executing during project renders.

 |

Freeze: Collaboration Control

Prevent documents from re-executing during project renders.

Results saved in `_freeze` directory (commit to version control).

 |

Freeze: Collaboration Control

Prevent documents from re-executing during project renders.

Results saved in `_freeze` directory (commit to version control).

YAML

```yaml
execute:
  freeze: auto    # Re-render only when source changes
  freeze: true    # Never re-render during project render
```

 |

Key Behavioural Differences

Action	Cache	Freeze
Individual document render	Used	Ignored
Project render	Used	Controls execution
Prose changes	No effect	No effect
Code changes	Invalidates cache	Triggers re-render (auto mode)

 |

When to Use Each

 |

When to Use Each

Use Cache When

- Developing multi format documents.
- Iterating on analysis.
- Working with expensive computations.
- Editing prose frequently.

 |

When to Use Each

Use Cache When

- Developing multi format documents.
- Iterating on analysis.
- Working with expensive computations.
- Editing prose frequently.

Use Freeze When

- Working in teams.
- Managing large projects.
- Environment setup is complex.
- Results need to be reproducible.

 |

Essential CLI Commands

 |

Essential CLI Commands

Cache Control

BASH

```bash
quarto render --cache          # Force cache use
quarto render --no-cache       # Disable cache
quarto render --cache-refresh  # Force refresh
```

 |

Essential CLI Commands

Cache Control

BASH

```bash
quarto render --cache          # Force cache use
quarto render --no-cache       # Disable cache
quarto render --cache-refresh  # Force refresh
```

Freeze Management

BASH

```bash
quarto render document.qmd     # Always executes (ignores freeze)
quarto render                  # Respects freeze settings
```

 |

Golden Rules

 |

Golden Rules

1. **Cache for Development.**
   Use when iterating on individual documents.

 |

Golden Rules

1. **Cache for Development.**
   Use when iterating on individual documents.

2. **Freeze for Collaboration.**
   Use when working with teams or managing large projects.

 |

Golden Rules

1. **Cache for Development.**
   Use when iterating on individual documents.

2. **Freeze for Collaboration.**
   Use when working with teams or managing large projects.

3. **Different Scopes.**
   - Cache: Session-level performance.
   - Freeze: Project-level workflow management.

 |

Golden Rules

1. **Cache for Development.**
   Use when iterating on individual documents.

2. **Freeze for Collaboration.**
   Use when working with teams or managing large projects.

3. **Different Scopes.**
   - Cache: Session-level performance.
   - Freeze: Project-level workflow management.

4. **Version Control.**
   Always commit `_freeze` directory contents.

 |

Parameters

- What Are Parameters?
- Defining Parameters
- Passing Parameters: Command Line
- Passing Parameters: YAML Files
- Practical Example
- Best Practices
- Workflow Integration

 |

What Are Parameters?

 |

What Are Parameters?

Parameters allow you to create **dynamic variations** of the same document:

 |

What Are Parameters?

Parameters allow you to create **dynamic variations** of the same document:

- **Geographic reports** → Different locations

 |

What Are Parameters?

Parameters allow you to create **dynamic variations** of the same document:

- **Geographic reports** → Different locations
- **Time-based analysis** → Different periods

 |

What Are Parameters?

Parameters allow you to create **dynamic variations** of the same document:

- **Geographic reports** → Different locations
- **Time-based analysis** → Different periods
- **Scenario modelling** → Different assumptions

 |

What Are Parameters?

Parameters allow you to create **dynamic variations** of the same document:

- **Geographic reports** → Different locations
- **Time-based analysis** → Different periods
- **Scenario modelling** → Different assumptions
- **Multi-client reports** → Different datasets

 |

What Are Parameters?

Parameters allow you to create **dynamic variations** of the same document:

- **Geographic reports** → Different locations
- **Time-based analysis** → Different periods
- **Scenario modelling** → Different assumptions
- **Multi-client reports** → Different datasets

> **Tip**
>
> Think of parameters as **variables** that make your documents code reusable and flexible.

Defining Parameters

Python (`papermill`) | R (`knitr`)

```python
#| tags: [parameters]
alpha = 0.1
ratio = 0.5
country = "UK"
year = 2024
```

- Must use `#| tags: [parameters]` comment.
- Provide sensible default values.
- Parameters available in top-level environment.
- Works with `.qmd` and `.ipynb` files.

 |

Defining Parameters

Python (`papermill`) | R (`knitr`)

```yaml
---
title: "Sales Report"
params:
  region: "North"
  start_date: "2024-01-01"
  threshold: 1000
---
```

Access in R code:

```r
print(params[["region"]])
```

 |

Passing Parameters: Command Line

 |

Passing Parameters: Command Line

Use the `-P` (`--execute-param`) flag to override parameter values:

 |

Passing Parameters: Command Line

Use the `-P` (`--execute-param`) flag to override parameter values:

BASH

```bash
# Single parameter
quarto render report.qmd --execute-param country:France

# Multiple parameters
quarto render report.qmd -P year:2023 -P region:Europe

# With output specification
quarto render report.qmd -P country:Germany --to typst
```

Passing Parameters: YAML Files

 |

Passing Parameters: YAML Files

Create reusable parameter sets with YAML files:

 |

Passing Parameters: YAML Files

Create reusable parameter sets with YAML files:

params.yml

```yaml
country: "Spain"
year: 2023
budget: 50000
```

Passing Parameters: YAML Files

Create reusable parameter sets with YAML files:

params.yml

```yaml
country: "Spain"
year: 2023
budget: 50000
```

Terminal

```
quarto render report.qmd \
  --execute-params params.yml
```

 |

Passing Parameters: YAML Files

Create reusable parameter sets with YAML files:

params.yml

```yaml
country: "Spain"
year: 2023
budget: 50000
```

Terminal

```
quarto render report.qmd \
  --execute-params params.yml
```

Benefits

Version control, team sharing, complex configurations.

 |

Practical Example

 |

Practical Example

Setup (Jupyter/Python)

MARKDOWN

```{python}
#| tags: [parameters]
region = "All"
quarter = "Q1"
min_sales = 1000
```

Practical Example

Setup (Jupyter/Python)

MARKDOWN

```
```{python}
#| tags: [parameters]
region = "All"
quarter = "Q1"
min_sales = 1000
```
```

Using parameters

MARKDOWN

```
```{python}
if region ≠ "All":
    df = df[df['region'] == region]

df_filtered = df[df['sales'] ⩾ min_sales]
title = f"{region} Sales - {quarter}"
```
```

Practical Example

Setup (Jupyter/Python)

MARKDOWN

````
```{python}
#| tags: [parameters]
region = "All"
quarter = "Q1"
min_sales = 1000
```
````

Command line usage

BASH

```
quarto render sales.qmd \
  --execute-param region:North \
  --execute-param quarter:Q2
```

Using parameters

MARKDOWN

````
```{python}
if region ≠ "All":
    df = df[df['region'] == region]

df_filtered = df[df['sales'] ≥ min_sales]
title = f"{region} Sales - {quarter}"
```
````

Practical Example

Setup (Jupyter/Python)

MARKDOWN

```
```{python}
#| tags: [parameters]
region = "All"
quarter = "Q1"
min_sales = 1000
```
```

Using parameters

MARKDOWN

```
```{python}
if region ≠ "All":
    df = df[df['region'] == region]

df_filtered = df[df['sales'] ⩾ min_sales]
title = f"{region} Sales - {quarter}"
```
```

Command line usage

BASH

```
quarto render sales.qmd \
  --execute-param region:North \
  --execute-param quarter:Q2
```

Results

Customised report for North region, Q2 data.

Best Practices

 |

Best Practices

Do

 |

Best Practices

Do

- Provide meaningful defaults.

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.
- Use YAML for complex configs.

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.
- Use YAML for complex configs.

Avoid

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.
- Use YAML for complex configs.

Avoid

- Generic names (`x` , `val`).

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.
- Use YAML for complex configs.

Avoid

- Generic names (`x` , `val`).
- Parameters without defaults.

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.
- Use YAML for complex configs.

Avoid

- Generic names (`x` , `val`).
- Parameters without defaults.
- Hardcoded values.

 |

Best Practices

Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.
- Use YAML for complex configs.

Avoid

- Generic names (`x` , `val`).
- Parameters without defaults.
- Hardcoded values.
- Overly complex structures.

 |

Workflow Integration

 |

Workflow Integration

Automated reporting

BASH

```bash
# Daily reports
quarto render report.qmd -P date:"$(date +%Y-%m-%d)"

# Multiple outputs
for region in North South East West; do
  quarto render regional.qmd -P region:$region
done
```

 |

Workflow Integration

Automated reporting

BASH

```bash
# Daily reports
quarto render report.qmd -P date:"$(date +%Y-%m-%d)"

# Multiple outputs
for region in North South East West; do
  quarto render regional.qmd -P region:$region
done
```

Perfect for

CI/CD pipelines, batch processing, scheduled reports.

- Exercise Overview
- Part 1: Environment Setup
- Part 2: Replace Static with Dynamic
- Part 3: Create Your First Visualization
- Part 4: Experiment with Code Visibility
- Success Criteria

Hands-On Exercise: Adding Computational Power to Your Portfolio

 |

Exercise Overview

Objective: Transform your Session 2 portfolio document into a computational showcase by adding live code execution, demonstrating the power of embedding computations in Quarto documents using modern Python tools.

Example Code: Exercises

```bash
tar -xzf "03-exercises.tar.gz" -C "03-embedding-computations-code"
```

 |

Part 1: Environment Setup

Choose your language and update your YAML header:

Python | R | Julia

YAML

```yaml
---
title: "My Computational Portfolio"
author: "Your Name"
engine: jupyter
jupyter: python3
execute:
  echo: true
  warning: false
---
```

Install packages:

BASH

```bash
uv add polars plotnine # pip install polars plotnine
```

Part 1: Environment Setup

Choose your language and update your YAML header:

Python | **R** | Julia

```yaml
---
title: "My Computational Portfolio"
author: "Your Name"
engine: knitr
execute:
  echo: true
  warning: false
---
```

Install packages:

```r
install.packages(c("dplyr", "ggplot2"))
```

 |

Part 1: Environment Setup

Choose your language and update your YAML header:

Python | R | Julia

```yaml
---
title: "My Computational Portfolio"
author: "Your Name"
engine: julia
execute:
  echo: true
  warning: false
---
```

Install packages:

```julia
using Pkg
Pkg.add(["DataFrames", "Plots", "StatsPlots"])
```

 |

Part 2: Replace Static with Dynamic

Replace your skills table from Session 2:

Python | R | Julia

- Create a Polars `DataFrame` with your skills data.
- Include columns: skill name, confidence level, hours practiced.
- Use `print()` to display the data nicely.
- Calculate total practice hours with `pl.sum()`.
- Show the result using inline code `` `{python} your_calculation` ``.

 |

Part 2: Replace Static with Dynamic

Replace your skills table from Session 2:

Python | **R** | Julia

- Create a `data.frame` with your skills data.
- Include columns: skill name, confidence level, hours practiced.
- Use `knitr::kable()` to display the table nicely.
- Calculate total practice hours with `sum()`.
- Show the result using inline code `` `{r} your_calculation` ``.

 |

Part 2: Replace Static with Dynamic

Replace your skills table from Session 2:

Python | R | Julia

- Create a `DataFrame` with your skills data.
- Include columns: skill name, confidence level, hours practiced.
- Display the `DataFrame` directly.
- Calculate total practice hours with `sum()`.
- Show the result using inline code `` `{julia} your_calculation` ``.

 |

Part 3: Create Your First Visualization

Make a chart of your skills:

Python | R | Julia

- Use plotnine to create a bar chart.
- Try `geom_col()` or `geom_bar()`.
- Add proper labels with `labs()`.
- Use `theme_minimal()` for clean styling.
- Add a figure caption with `#| fig-cap:`.

Part 3: Create Your First Visualization

Make a chart of your skills:

Python | **R** | Julia

- Use ggplot2 to create a bar chart.
- Try `geom_col()` or `geom_bar()`.
- Add proper labels with `labs()`.
- Use `theme_minimal()` for clean styling.
- Add a figure caption with `#| fig-cap:`.

Part 3: Create Your First Visualization

Make a chart of your skills:

Python | R | Julia

- Use Plots.jl to create a bar chart.
- Try `bar()` function.
- Add proper labels with `xlabel!()` and `ylabel!()`.
- Add a title with `title!()`.
- Add a figure caption with `#| fig-cap:`.

Part 4: Experiment with Code Visibility

Try different code cell options:

Python | R | Julia

- Create a cell with `#| eval: false` (shows code, doesn't run).
- Make a collapsible section with `#| code-fold: true`.
- Hide code with `#| echo: false` but show output.
- Use `#| include: false` for data setup.

Part 4: Experiment with Code Visibility

Try different code cell options:

Python | R | Julia

- Create a cell with `#| eval: false` (shows code, doesn't run).
- Make a collapsible section with `#| code-fold: true`.
- Hide code with `#| echo: false` but show output.
- Use `#| include: false` for library loading.

Part 4: Experiment with Code Visibility

Try different code cell options:

Python | R | Julia

- Create a cell with `#| eval: false` (shows code, doesn't run).
- Make a collapsible section with `#| code-fold: true`.
- Hide code with `#| echo: false` but show output.
- Use `#| include: false` for package imports.

Success Criteria

You've successfully completed the exercise if you can:

- Transform static content into dynamic, computed content using Python , R , or Julia .
- Create and display data visualisations.
- Use inline code to insert computed values into prose.
- Experiment with different code cell visibility options.

 |

Mickaël CANOUIL, *Ph.D.*

Independent Contractor

pro@mickael.canouil.dev

Website
mickael.canouil.fr

GitHub
@mcanouil

LinkedIn
MickaelCanouil

Bluesky
@mickael.canouil.fr

Mastodon
@MickaelCanouil

 |