Welcome & System Basics

Lecture 01

Dr. Colin Rundel

Course Details

Course Team

Instructor

- Dr. Colin Rundel
  - colin.rundel@duke.edu / cr173@duke.edu / rundel@gmail.com
  - Office hours: 204 Old Chem or Zoom
    - Time TBD or by appointment

TAs

- Ruibo Song
- William Mentz

Course website(s)

- GitHub pages - https://sta523-fa26.github.io
  - HTML, PDF, and qmds of Slides
  - Readings and other notes

- Canvas - https://canvas.duke.edu/courses/83612
  - Announcements
  - Gradebook
  - Gradescope

Assessment

We will be assessing you based on the following:

Assignment	Type	Value	n	Assigned
Homeworks	Team	35%	5-6	~ Every other week
Midterms	Individual	50%	2	~ Week 8 and 16
Quizzes	Individual	15%	~10-15	~ Weekly

Homeworks

- Roughly biweekly assignments
- Open ended, ~5 - 15 hours of work
- Randomly assigned teams of 3-4 students, teams change after each assignment
- Peer evaluation after completion
- Expectations and roles:
  - Everyone is expected to contribute equal *effort*
  - Everyone is expected to understand *all* code turned in
  - Individual contribution evaluated by multiple metrics (peer evaluation, commits, etc.)

Labs

- Attendance is expected - you must attend the lab you are enrolled in
- Opportunity to work on course assignments with TA support
- Labs will begin this week - Friday (8/28)

Midterms

Each exam will have two components:

1. Individual take home
    - Similar in scope to homework
    - ~1 week to complete
2. In class written exam (based on take home):
    - Read & evaluate code
    - Describe pseudo-code solutions

You must sit the in class exam in order to receive points for the take home - in the event of a *documented* emergency the exam will be replaced by an in person oral exam (which must be scheduled within one week of the original exam date).

Quizzes

- Roughly once a week
- Randomly in lecture or lab (start, middle, or end)
- 5 multiple choice questions, 5 minutes
- Your lowest 3-5 quiz scores will be dropped
- No excused absences for quizzes (covered by above)

Sharing / reusing code / AI policy

- We are aware that a huge volume of code is available on the web, and many tasks may have solutions posted.
- Unless explicitly stated otherwise, this course's policy is that you may make use of any online resources (e.g. Google, StackOverflow) but you must explicitly cite where you obtained any code you directly use or use as inspiration in your solution(s).
- Teams should not directly share answers / code with other teams; however, you are welcome to discuss the problems in general and ask for advice.
- Any recycled/copied code that is not explicitly cited will be treated as plagiarism, regardless of source.
- The same applies to the use of LLMs like ChatGPT, Claude, Gemini, or GitHub Copilot - you are welcome to make use of these tools as the basis for your solutions but you must cite the tool when using it for *significant* code generation.

What happens if you violate the academic honesty policy?

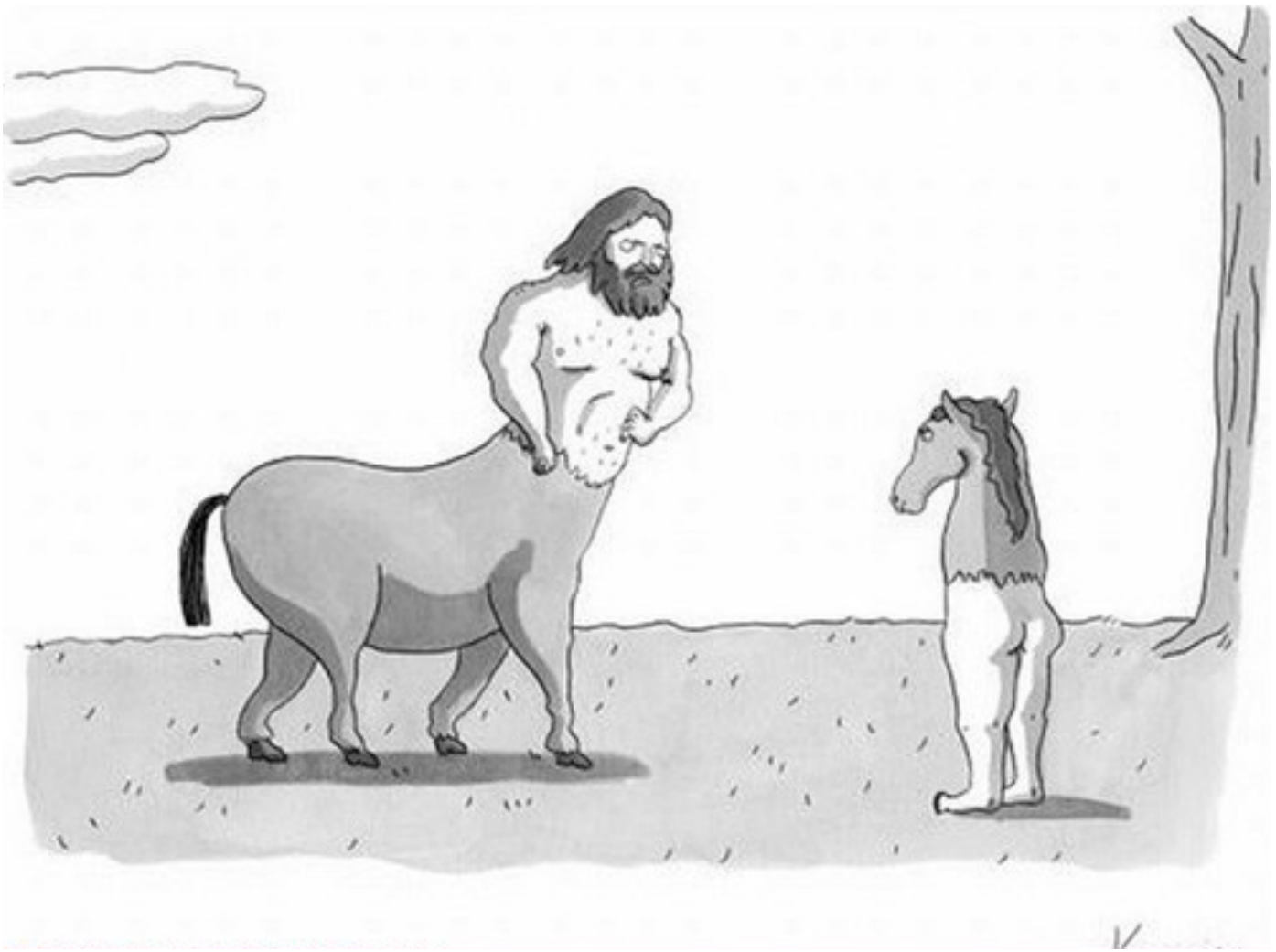
Any violation of the academic honesty standards outlined in the Duke Community Standard, the Graduate School's Standards of Conduct, or those specific to this course:

- will automatically result in a 0 for the relevant portion or the entirety of the assignment or assessment,
- can result in further deductions to your overall course grade (e.g. dropping down to the next letter grade or to an F), and
- can be reported to the Graduate School and the Office of Student Conduct & Community Standards for further action.

Brief thoughts on AI tools

- AI tools are not a replacement for understanding the material, but they can help you learn it.
- Reading code and writing code are skills that take time and practice to develop - both are essential.
- The nature of these tools is changing rapidly - autocomplete vs chatbots vs agentic tools

Centaurs vs Reverse Centaurs



Course Tools

Accessing Posit Workbench

To reduce friction, the preferred method is to use the department's RStudio server(s).

To access RStudio/Posit Workbench:

1. Navigate to https://rstudio.stat.duke.edu
2. Log in with your Duke NetID and password.

If off campus, use the VPN to create a secure connection from your computer to Duke. If you are on campus, be sure you

Troubleshooting DSS server access

If you cannot access RStudio via the DSS servers:

- Make sure you are on an authenticated Duke network (e.g. DukeBlue or VPN)
- Make sure you are not using a custom DNS server
  - e.g. `1.1.1.1` or `8.8.8.8`

Local R + Python + Positron

If working locally you should make sure that your environment meets the following requirements:

- latest R (4.6)
- latest Python (3.14) + uv (0.12)
- latest Positron (2026.08.1)
- working git installation
- ability to create SSH keys (for GitHub authentication)

Support policy for local installs - we will try to help you troubleshoot if we can but reserve the right to tell you to use the dept server.

GitHub

- We will be using a GitHub organization for this course github.com/sta523-fa26
- All assignments will be distributed and collected via GitHub
- All of your work and your membership (enrollment) in the organization is private
- We will be distributing a survey this week to collect your GitHub account names
  - Before lab you will be invited to the course organization.
- All course related repositories will be created for you

By Wednesday evening

- Not enrolled? Fill out https://bit.ly/enroll-sta523-fa26
- Complete the course survey (link via email)
- Create a GitHub account if you don't have one
- Make sure you can log in to the Department's Workbench server https://rstudio.stat.duke.edu
- Set up SSH key authentication with GitHub, see https://github.com/DukeStatSci/github_auth_guide

R packages

What are R packages?

R packages are just collections of files - R code, compiled code (C, C++, Rust etc.), data, documentation, and others that live in your library path.

```r
.libPaths()
```

```
[1] "/Users/rundel/Library/R/arm64/4.6/library"
[2] "/Library/Frameworks/R.framework/Versions/4.6/Resources/library"
```

```r
dir(.libPaths())
```

```
  [1] "_build"                    "_cache"
  [3] "abind"                     "airports"
  [5] "and"                       "anytime"
  [7] "ape"                       "archive"
  [9] "arrayhelpers"              "arrow"
 [11] "AsioHeaders"               "askpass"
 [13] "assertthat"                "astsa"
 [15] "available"                 "babelwhale"
 [17] "backports"                 "BART"
 [19] "base"                      "base64enc"
 [21] "base64url"                 "BayesFactor"
 [23] "bayesplot"                 "bcf"
 [25] "beeswarm"                  "bench"
 [27] "berryFunctions"            "BH"
 [29] "bigD"                      "bit"
 [31] "bit64"                     "bitops"
 [33] "blastula"                  "blob"
```

```
[35] "bonsai"                    "bookdown"
[37] "boot"                      "bootstrap"
[39] "brew"                      "bridgesampling"
[41] "brio"                      "brms"
[43] "Brobdingnag"               "broom"
```

System vs user libraries

`.libPaths()` will usually report (at least) two library locations:

- System library - contains the base and recommended packages (e.g. `stats`, `utils`, `MASS`, etc.). Shared by all users of a machine - on shared systems (like the DSS servers) it is read-only and maintained by the administrators.
- User library - belongs to you and is where packages you install will end up. The path is specific to both your user account and the minor version of R (e.g. `4.6`)

```r
.libPaths()[1] |> dir() |> head(n = 16)
```

```
 [1] "_build"       "_cache"       "abind"
 [4] "airports"     "and"          "anytime"
 [7] "ape"          "archive"      "arrayhelpers
[10] "arrow"        "AsioHeaders"  "askpass"
[13] "assertthat"   "astsa"        "available"
[16] "babelwhale"
```

```r
.libPaths()[2] |> dir() |> head(n = 16)
```

```
 [1] "base"       "boot"       "class"
 [4] "cluster"    "codetools"  "compiler"
 [7] "datasets"   "foreign"    "graphics"
[10] "grDevices"  "grid"       "KernSmooth"
[13] "lattice"    "MASS"       "Matrix"
[16] "methods"
```

When loading a package, R searches the locations in `.libPaths()` in order and uses the first copy it finds - so a package

Installing packages

Generally packages come from somewhere on the internet, most commonly from CRAN or GitHub; the methods for installing from these locations are slightly different.

CRAN:

```r
install.packages("diffmatchpatch")
```

GitHub:

```r
remotes::install_github("rundel/diffmatchpatch")
```

Packages only need to be *installed* once (per R version), but must be *loaded* in every new R session where you want to use them.

What is CRAN

The Comprehensive R Archive Network is the central repository of R packages.

- Maintained by the R Foundation and run by a team of volunteers, ~23k packages
- Contains all *current* versions of released packages as well as previous releases (including archived packages)
- Similar in spirit to Perl's CPAN, TeX's CTAN, and Python's PyPI
- Some important features:
  - All submissions are reviewed by humans + automated checks
  - Strictly enforced submission policies and package requirements
  - All packages must be actively maintained and support upstream and downstream changes

pak

pak is a modern replacement for `install.packages()` and `remotes::install_github()` developed by the open source team at Posit.

- Fast - resolves, downloads, and installs packages in parallel (with caching)
- One interface for many sources - CRAN, Bioconductor, GitHub, URLs, local files, etc.
- Plans the full installation up front - shows what will be installed and catches dependency conflicts before anything is changed
- Can find and install needed system dependencies (e.g. system libraries on Linux servers)

Packages are specified using reference strings,

```R
pak::pak("diffmatchpatch")         # CRAN
pak::pak("rundel/diffmatchpatch") # GitHub
```

Python packages with uv

Python packaging landscape

Python's packaging ecosystem has historically been fragmented:

- Multiple tools: `pip`, `virtualenv`, `venv`, `conda`, `poetry`, `pipenv`, etc.
- Multiple config files: `requirements.txt`, `setup.py`, `pyproject.toml`, etc.
- Version management often handled separately (`pyenv`)

uv is a modern tool that aims to unify these concerns with a fast, Rust-based implementation.

What is uv?

uv is a Python package and project manager developed by Astral (creators of `ruff` and recent OpenAI acquisition)

Key features:

- Extremely fast (10-100x faster than pip)
- Manages Python versions
- Creates and manages virtual environments
- Installs packages (from PyPI)
- Handles project dependencies via `pyproject.toml`
- Drop-in replacement for `pip` and `virtualenv`
- Directly supported by Positron and Reticulate

What is PyPI

The Python Package Index is the central repository of Python packages.

- Maintained by the Python Software Foundation and run by volunteers, ~880k projects
- Contains all released versions of packages, as source distributions (`sdists`) and/or prebuilt binaries (`wheels`)
- Some important differences from CRAN:
  - No review process - anyone with an account can upload a package (name squatting and malicious packages are a recurring problem)
  - No requirement that packages build, pass checks, or be actively maintained
  - No requirement that packages stay compatible with newer versions of their dependencies (or vice versa) - version conflicts between packages are common
  - Package quality, documentation, and upkeep vary widely - it is up to you to vet what you install

Installing uv

uv is already installed on the departmental servers; for local installs:

On MacOS/Linux:

```bash
curl -LsSf https://astral.sh/uv/install.sh | sh
```

or with Homebrew:

```bash
brew install uv
```

or with pip / pipx:

```bash
pipx install uv
pip install uv
```

Verify installation

Once installed you should be able to run the following,

```bash
uv --version
```

```
uv 0.12.5 (Homebrew 2026-08-14 aarch64-apple-darwin)
```

As long as you have version `0.12.*` you should be fine.

Managing Python versions

uv can install and manage multiple Python versions,

```bash
uv python list
```

```
cpython-3.15.0rc1-macos-aarch64-none
cpython-3.15.0rc1+freethreaded-macos-aarch64-non
cpython-3.14.7-macos-aarch64-none
cpython-3.14.7-macos-aarch64-none
cpython-3.14.7-macos-aarch64-none
cpython-3.14.7-macos-aarch64-none
cpython-3.14.7+freethreaded-macos-aarch64-none
cpython-3.14.2-macos-aarch64-none
cpython-3.13.15-macos-aarch64-none
cpython-3.13.15+freethreaded-macos-aarch64-none
cpython-3.12.14-macos-aarch64-none
cpython-3.12.14-macos-aarch64-none
cpython-3.12.0-macos-aarch64-none
cpython-3.11.16-macos-aarch64-none
cpython-3.10.21-macos-aarch64-none
...
```

```bash
uv python install 3.14
```

```bash
uv python pin 3.14
```

The pinned version is stored in a `.python-version` file and will be used automatically for that directory (and its subdirectories).

Initializing a project

Use `uv init` to create a new project,

```bash
mkdir my-project
cd my-project
uv init
```

```
Initialized project `my-project`
```

```bash
ls -la
```

```
total 24
drwxr-xr-x@ 8 rundel  wheel  256 Aug 23 22:47 .
drwx------@ 3 rundel  wheel   96 Aug 23 22:47 ..
drwxr-xr-x@ 9 rundel  wheel  288 Aug 23 22:47 .git
-rw-r--r--@ 1 rundel  wheel  109 Aug 23 22:47 .gitignore
-rw-r--r--@ 1 rundel  wheel    5 Aug 23 22:47 .python-version
-rw-r--r--@ 1 rundel  wheel    0 Aug 23 22:47 README.md
-rw-r--r--@ 1 rundel  wheel  361 Aug 23 22:47 pyproject.toml
drwxr-xr-x@ 3 rundel  wheel   96 Aug 23 22:47 src
```

This creates a `pyproject.toml`, a minimal package skeleton in `src/` (with a `main()` entry point), a `README.md`, and basic git infrastructure. Generally, we only really care about the `pyproject.toml`, which we can generate on its own via `uv init --bare`.

pyproject.toml

Modern project metadata file, tracks the Python version and package dependencies among other details.

```toml
[project]
name = "my-project"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
authors = [
    { name = "Colin Rundel", email = "rundel@gmail.com" }
]
requires-python = ">=3.14"
dependencies = []

[project.scripts]
my-project = "my_project:main"

[build-system]
requires = ["uv_build>=0.12.5,<0.13.0"]
build-backend = "uv_build"
```

Adding dependencies

Once we have our project set up, we can add (and install) dependencies directly via uv. `uv add` updates `pyproject.toml` and installs the package (creating a venv if needed).

```bash
uv add numpy
```

```
Using CPython 3.14.2
Creating virtual environment at: .venv
Resolved 2 packages in 157ms
Installed 1 package in 27ms
 + numpy==2.4.1
```

```bash
uv add pandas matplotlib scikit-learn
```

```
Resolved 18 packages in 490ms
Prepared 5 packages in 9.18s
Installed 15 packages in 134ms
 + contourpy==1.3.3
 + cycler==0.12.1
 + fonttools==4.61.1
 + joblib==1.5.3
 + kiwisolver==1.4.9
 + matplotlib==3.10.8
 + packaging==25.0
 + pandas==3.0.0
 + pillow==12.1.0
 + pyparsing==3.3.2
 + python-dateutil==2.9.0.post0
 + scikit-learn==1.8.0
 + scipy==1.17.0
 + six==1.17.0
 + threadpoolctl==3.6.0
```

```bash
uv add "pydantic<2"
```

```
Resolved 26 packages in 336ms
Prepared 1 package in 238ms
Installed 2 packages in 3ms
 + pydantic==1.10.26
 + typing-extensions==4.15.0
```

```bash
uv add --dev pytest ruff
```

```
Resolved 24 packages in 337ms
Prepared 4 packages in 859ms
Installed 5 packages in 27ms
 + iniconfig==2.3.0
 + pluggy==1.6.0
 + pygments==2.19.2
 + pytest==9.0.2
 + ruff==0.14.13
```

Updated `pyproject.toml`

```toml
[project]
name = "my-project"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
authors = [
    { name = "Colin Rundel", email = "rundel@gmail.com" }
]
requires-python = ">=3.14"
dependencies = [
    "matplotlib>=3.10.8",
    "numpy>=2.4.1",
    "pandas>=3.0.0",
    "pydantic<2",
    "scikit-learn>=1.8.0",
]

[project.scripts]
my-project = "my_project:main"

```

Virtual environments

Virtual environments isolate project dependencies from the system Python and other projects. Packages are installed in a local folder in your project.

As we just saw, using `uv add` will create a new virtual environment in `.venv` by default if there is not an existing venv.

To explicitly create your own venv you can use,

```bash
uv venv                 # Create a virtual environment
uv venv --python 3.13   # Or specify Python version
uv venv .venv2          # Or specify directory name
```

Activating environments

To use the virtual environment certain environment variables need to be set correctly (e.g. `PATH`, `PYTHONPATH`, etc.) so that the correct Python binary and libraries are used.

From the command line / terminal you can run the following in your project directory:

```bash
source .venv/bin/activate  # macOS/Linux (bash/zsh)
.venv\Scripts\Activate.ps1 # Windows (PowerShell)
.venv\Scripts\activate.bat # Windows (cmd)
```

Alternatively (*strongly recommended*), use `uv run` to execute commands in the environment without activating,

```bash
uv run python script.py
uv run pytest
uv run quarto render test.qmd
```

`uv sync`

Since the `.venv` folder is system-specific (and large) it is not typically committed to git. Instead you will likely clone a repository that just has a `pyproject.toml` file.

Use `uv sync` to construct the venv and install all dependencies for the project

```bash
uv sync
```

```
Using CPython 3.14.2
Creating virtual environment at: .venv
Resolved 26 packages in 8ms
Installed 23 packages in 96ms
 + contourpy==1.3.3
 + cycler==0.12.1
 + fonttools==4.61.1
 + iniconfig==2.3.0
 + joblib==1.5.3
 + kiwisolver==1.4.9
 + matplotlib==3.10.8
 + numpy==2.4.1
 + packaging==25.0
 + pandas==3.0.0
 + pillow==12.1.0
 + pluggy==1.6.0
   ...
```

Common workflows

New project setup:

```bash
mkdir my-project
cd my-project
uv init --bare
uv add pandas numpy
uv run python project.py
```

Clone existing project:

```bash
git clone <repo-url>
cd <repo>
uv sync
uv run python script.py
```

uv and Positron

Positron automatically detects virtual environments in your project directory. When you open a folder containing a `.venv` directory (created by uv), Positron will:

- Detect the environment and offer to use it
- Show the active Python interpreter in the status bar
- Use the environment for the Python console and when running scripts

If not automatically detected, you can manually select the interpreter via the Command Palette (`Cmd+Shift+P` / `Ctrl+Shift+P`) and searching for “Python: Select Interpreter”.