Empiricist's Workflow

実証研究のためのモダンなツールと作法

柳本和春
神戸大学
yanagimoto@econ.kobe-u.ac.jp

2026 年 08 月 12 日

目次

はじめに

AI 時代のプログラミング知識

このコースでは，AI 時代に必要なプログラミング知識を学ぶという目的で作りました．プログラミングコードを AI がある程度の精度で書けるようになった今，プログラミングとはプロンプティングの技術に変わってきています．ここで求められるのは個別のパッケージや関数に関する理解ではなく，AI に適切な指示を与えられるかの能力になってきました．

このコースでは，プログラミングで何ができるのかを理解することに重きを置き，その上で AI に適切な指示を与えるための知識を学んでいきます．そのため，プログラミングの基礎的な知識は必要ですが，個別のパッケージや関数の使い方については深くは解説しません．分からなければ AI に聞くことを推奨します．

環境構築

授業を始める前に次の環境構築を完了させておいてください．また，チャプター 3 で用いるため，GitHub のアカウントも作成しておいてください．後述するように，GitHub の Education アカウントに登録することをお勧めします．

このコースでは Mac または Linux 環境での実行を想定しています．Windows ユーザーには，Windows Subsystem for Linux (WSL) 上での実行を推奨します．以下では，エディタ (VSCode)，文献管理ソフト (Zotero)，R のバージョン管理ツール (rig)，Quarto，バージョン管理システムの git のインストール手順を OS 別にまとめます．

Mac

まず，パッケージマネージャの Homebrew を導入します．ターミナルで以下を実行し，画面の指示に従ってください (インストール後に表示される `brew shellenv` の設定コマンドも忘れずに実行します)．

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

なお，バージョン管理の git は，Homebrew を入れるときに一緒に導入される Xcode Command Line Tools に含まれているため，別途インストールする必要はありません．`git --version` でバージョンが表示されれば導入済みです (まだなら `xcode-select --install` で入ります)．

Homebrew が入れば，必要なソフトウェアはすべて `brew` でインストールできます．

```
brew install --cask visual-studio-code zotero
```

```
# rig is distributed as a cask in the r-lib/rig tap
brew tap r-lib/rig
brew install --cask rig

brew install quarto
```

R 本体は rig を通じてインストールします. `rig add release` で最新の安定版が入ります.

```
rig add release
```

アップデートも Homebrew 経由で行います. 以下のコマンドで, brew でインストールしたソフトウェア (VSCode, Zotero, Quarto など) がまとめて最新版になります.

```
brew update && brew upgrade
```

R の新しいバージョンがリリースされたときは, `rig add release` を再実行すれば追加されます. インストール済みのバージョンは `rig list` で確認でき, `rig default 4.5.1` のように切り替えられます (不要になった古いバージョンは `rig rm` で削除できます).

Windows

VSCode と Zotero は Windows 側にインストールします. Windows 標準のパッケージマネージャ winget を使い, PowerShell で以下を実行してください.

```
winget install Microsoft.VisualStudioCode
winget install DigitalScholar.Zotero
```

次に, WSL を導入します. 管理者権限の PowerShell で以下を実行し, 再起動後に Ubuntu のユーザー名とパスワードを設定してください.

```
wsl --install
```

VSCode には拡張機能 WSL をインストールしておくと, WSL 内のファイルを VSCode から直接編集できます.

WSL の Ubuntu では, ソフトウェアの管理に標準のパッケージマネージャ apt を使います. Mac における Homebrew に相当するもので, 基本のコマンドは次の 3 つです.

```
# refresh the package catalog and upgrade all installed packages
sudo apt update && sudo apt upgrade -y
sudo apt install -y curl # install a package (here: curl)
```

先頭の `sudo` は, システムを変更する操作を管理者権限で実行するための命令で, 実行時には WSL セットアップ時に設定したパスワードを聞かれます (詳しくは セクション A.8 で説明します). `apt update` はパッケージの「カタログ」を最新にするだけで, 何もインストールしません. インストールやアップグレードの前には, まず `apt update` を実行するのが作法です. `-y` は, インストールやアップグレードの途中で聞かれる「続行しますか?」という質問に自動で「はい」と答えるオプションです.

この後の手順で使うダウンロードツール (curl, wget), R のパッケージをソースコードからビルドするときに必要になるコンパイラ類 (build-essential), そしてバージョン管理の git を, ここでまとめて入れておきます.

```
sudo apt update
sudo apt install -y build-essential git
```

rig と Quarto は, apt ではなく公式の配布物から WSL (Ubuntu) 側にインストールします. WSL のターミナルで以下を実行してください.

```
# rig (R installation manager)
curl  -Ls  https://github.com/r-lib/rig/releases/download/latest/rig-linux-$(arch)-latest.
tar.gz |
  sudo tar xz -C /usr/local
rig add release
```

Quarto は, バージョン番号を変数に入れて .deb パッケージをダウンロードし, インストールします.

```
# Latest release as of this page's build; see https://quarto.org/docs/download/
QUARTO_VERSION=1.10.18
wget       "https://github.com/quarto-dev/quarto-cli/releases/download/v${QUARTO_VERSION}/
quarto-${QUARTO_VERSION}-linux-amd64.deb"
sudo dpkg -i "quarto-${QUARTO_VERSION}-linux-amd64.deb"
```

アップデートするときは, `QUARTO_VERSION` を新しいバージョン番号に変えて同じコマンドを再実行するだけです. R のアップデートは Mac と同様で, `rig add release` を再実行します. apt で入れたソフトウェアは `sudo apt update && sudo apt upgrade` で, Windows 側のソフトウェアは `winget upgrade --all` でまとめて更新できます.

AI Coding Tools

私は AI coding tools として, Claude Code を主に用いますが, この授業を学ぶ上では チャプター 12 以外では必要ありません. また, チャプター 12 で紹介するワークフローは, Claude Code 以外の GitHub Copilot や Codex でも同じように使えます. AI coding tools を試す, という意味では無料で始められる GitHub Copilot から始めるのがいいでしょう. しかし, 研究で本格的に使うにはある程度お金を払う必要があります.

GitHub Copilot

GitHub Copilot は, GitHub が提供する AI コーディングアシスタントです. Visual Studio Code の拡張機能として提供されており, コードの補完や生成を支援します. Copilot は OpenAI の Codex モデルを基盤としており, プログラミング言語やフレームワークに関する知識を持っています. また, Claude や Gemini などの他のモデルも利用可能です.

学生や教員の場合は, GitHub Education のアカウントを作成することで, GitHub Pro の機能を無料で利用できます. これにより, Copilot の使用上限が無料アカウントと比べて拡大します. 教育機関の認証はやや厳しくなっていると言われており, 大学の構内からのアクセスが必要と言われています.

第 1 章
アーキテクチャ

この章では, コンピュータがどのような部品からできていて, その上でソフトウェアがどのように動いているのかを学びます. 普段の分析ではあまり意識しない話題ですが, 研究生活では意外なほど頻繁に顔を出します. 新しい PC を買うとき, ソフトウェアのインストーラを選ぶとき, 大学の計算サーバーやクラウドを使うとき, そして AI に環境構築を指示するとき, この章の語彙が判断の基礎になります.

1.1　コンピュータの基本構成

デスクトップ PC もノート PC も, そしてスマートフォンも, 基本的な構成は同じです. 図 1.1 のように, CPU, メモリ, ストレージ, GPU といった部品が, マザーボード (motherboard) と呼ばれる基板の上で, バス (bus) というデータの通り道によって結ばれています.

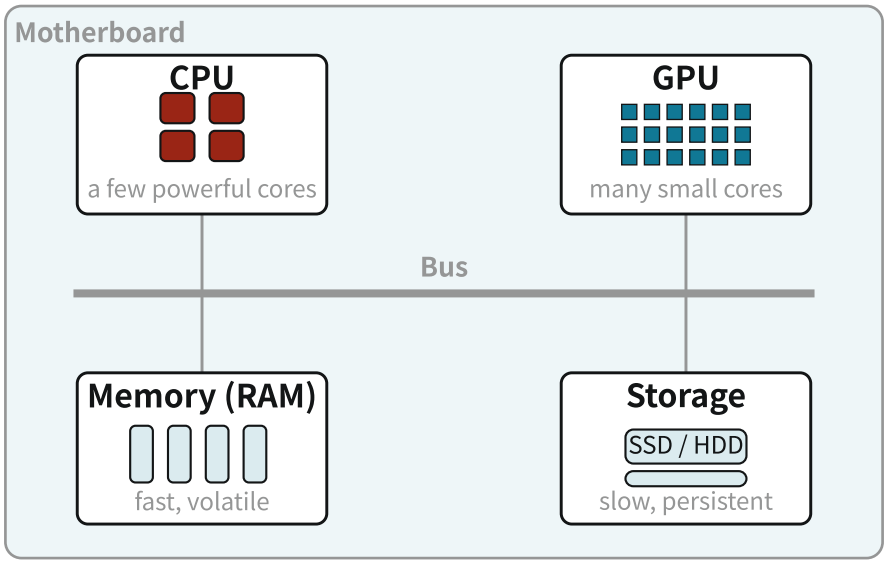


図 1.1: コンピュータの基本構成

1.1.1　CPU

図 1.2:　CPU (AMD Ryzen 5 2600)

CPU (Central Processing Unit, 中央演算処理装置) は, プログラムの命令を 1 つずつ実行するコンピュータの頭脳です (図 1.2). 性能を表す代表的な指標が 2 つあります.

- **クロック周波数** (clock speed): 1 秒間に刻む動作の回数で, GHz (10^9 回/秒) で表されます. おおまかには 1 コアあたりの処理の速さの目安です.
- **コア数** (cores): 独立して命令を実行できる演算ユニットの数です. 近年の CPU は 4 から 16 程度のコアを持つのが一般的です.

かつては世代ごとにクロック周波数が上がり, 何もしなくてもプログラムが速くなる時代がありました. しかし発熱の限界から 2000 年代半ばにクロック周波数の伸びは急減速し,[1] 以降の CPU はコア数を増やす方向に進化しています. 複数のコアを活かすには, プログラムの側で計算を分担させる並列化が必要になります (セクション 4.4).

1.1.2　メモリとストレージ

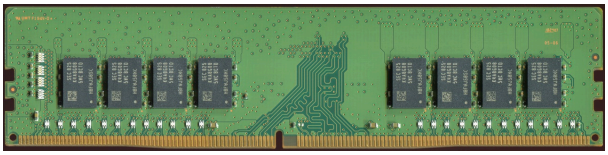


図 1.3:　メモリ (DDR4 DIMM)

メモリ (RAM, Random Access Memory) は, 実行中のプログラムとデータを置いておく作業スペースで (図 1.3), よく机の広さに例えられます. 机が広いほど多くの資料を同時に広げられるように, RAM が大きいほど大きなデータを一度に扱えます. 高速に読み書きできる一方で, 電源を切ると中身が消えます (揮発性, volatile).

[1] 1994 年から 2004 年の 10 年でクロック周波数は約 40 倍になりましたが (Pentium の 100 MHz から Pentium 4 の 3.8 GHz), その後の 20 年では 6 GHz 前後までしか伸びていません. 最近の CPU が謳う 5-6 GHz という数字も, 1 コア・短時間のブースト時のものです.

図 1.4: SSD (NVMe M.2)

図 1.5: HDD (カバーを開けた内部)

ストレージ (storage) は, ファイルを永続的に保存しておく場所で, SSD (Solid State Drive) や HDD (Hard Disk Drive) がこれにあたります (図 1.4, 図 1.5). こちらは本棚です. 容量は RAM の何十倍もありますが, 読み書きは桁違いに遅くなります.

「速いが小さい記憶」と「遅いが大きい記憶」を階層的に組み合わせるこの設計はメモリのヒエラルキーと呼ばれ, 計算の高速化を考えるうえで重要になります (セクション 4.1). なお, データ分析で「メモリが足りない」と言うときのメモリは RAM のことです. RAM に収まらないデータの扱い方は, データ処理の章で扱います.

1.1.3　GPU

図 1.6: GPU (GeForce RTX 5060 Ti)

GPU (Graphics Processing Unit) は, もともと画面描画のための部品です (図 1.6). 画面上の何百万というピクセルを同時に計算する必要があるため, 図 1.1 に描いたように, 単純な演算ユニットを数千個並べた構造をしています. 少数の強力なコアで複雑な処理を順にこなす CPU とは対照的な設計です.

この「単純な計算の大規模並列」は行列演算と相性が良いため, GPU は深層学習をはじめとする機

械学習の計算基盤になりました.[2] 経済学でも, 深層学習や大規模なシミュレーションを使う研究では GPU が登場しますが, 回帰分析が中心であれば必須ではありません.

> 研究用 PC を選ぶときの優先順位
>
> 1. RAM: 最優先です. 16GB を最低ラインとし, 大きめのデータを扱うなら 32GB 以上を勧めます.
> 2. CPU: コア数が多いほど並列化の恩恵を受けられます.
> 3. ストレージ: SSD で 512GB 以上が目安です. HDD は避けましょう.
> 4. GPU: 深層学習をしないなら不要です. 必要になったらクラウドで借りるという選択肢もあります.

1.2　CPU アーキテクチャ

ここからは, 同じ「CPU」の中にも互換性のない種類があるという話をします. まず, 自分の環境を R で確認してみましょう.

```
Sys.info()[c("sysname", "release", "machine")]
##  sysname  release  machine
## "Darwin" "25.5.0"  "arm64"
```

`machine` に表示されているのが CPU アーキテクチャです. 筆者の環境は Apple Silicon 搭載の Mac なので `arm64` と表示されています. Intel や AMD 製 CPU の PC なら `x86_64` になるはずです. この違いが何を意味するのかを見ていきましょう (`sysname` の Darwin については OS の節で説明します).

1.2.1　機械語と命令セット

CPU が直接理解できるのは, 機械語 (machine code) と呼ばれる 0 と 1 の列だけです. R のコードを実行するときも, R のインタープリタ (それ自体が機械語のプログラム) がコードを解釈し, 最終的にはすべて機械語の命令となって CPU に届いています.

どのような機械語の命令が使えるかという「語彙」の仕様を, **命令セットアーキテクチャ** (ISA, Instruction Set Architecture) といいます. 現在の主流は 表 1.1 の 2 系統です.

表 1.1: 主要な CPU アーキテクチャ

	x86-64 (AMD64)	ARM64 (AArch64)
設計	Intel, AMD	Arm 社が設計し各社へライセンス
採用例	Windows PC の大半, Intel Mac, サーバー	スマートフォン, Apple Silicon Mac, AWS Graviton
特徴	高性能重視, 長い後方互換性	省電力重視

[2]機械学習で事実上の標準となっているのが, NVIDIA 製 GPU 向けの計算基盤 CUDA です. 深層学習ライブラリの多くが CUDA を前提に開発されてきたため, 「GPU 計算といえば NVIDIA」という状況が長く続いています.

x86 は 1978 年の Intel 8086 に始まる系譜で, 40 年以上前のソフトウェアとの互換性を保ちながら拡張されてきました.[3] 一方 ARM は省電力を重視した設計で, スマートフォンのほぼすべてに採用されています.

長らく「PC とサーバーは x86, スマートフォンは ARM」という住み分けでしたが, 2020 年に Apple が Mac の CPU を自社設計の ARM チップ (Apple Silicon, M1 など) に切り替えたのを機に, この境界は崩れつつあります. クラウドでは AWS の Graviton など ARM サーバーが増えており, スーパーコンピュータ富岳の CPU も ARM 系です.

> **Intel と AMD, 型番と世代の読み方**
>
> x86-64 の CPU を作っているのは Intel と AMD の 2 社で, どちらを選んでもソフトウェアの互換性は変わりません. PC を買うときに見るべきは, 型番に埋め込まれたグレードと世代です.
>
> Intel の Core i7-13700 という型番なら, i7 がグレード (i3 < i5 < i7 < i9) を, 続く 13 が第 13 世代であることを表します. AMD の Ryzen 7 7700 なら, Ryzen 7 がグレード (3 < 5 < 7 < 9) を, 数字の先頭の 7 が 7000 シリーズという世代を表します. ノート PC 向けでは型番の末尾に文字が付き, U は省電力型, H は高性能型を意味します.
>
> 注意したいのは, グレードはあくまで同世代内での序列だという点です. 世代が新しい i5 が数世代前の i7 を上回ることは珍しくないため, セール品や中古品で「i7 搭載」とだけ書かれていたら, 必ず世代も確認しましょう.
>
> なお Intel は 2023 年末から, 新しい製品を Core Ultra (Core Ultra 5/7/9) というブランドに切り替えつつあります. 名前は変わりましたが, グレードと世代 (Core Ultra 7 155H なら先頭の 1 が第 1 世代) を確認するという読み方は同じです.

1.2.2　32-bit と 64-bit

アーキテクチャ名に付いている「64」は, CPU が一度に扱う整数やメモリアドレスの幅が 64 ビットであることを意味します. この幅が重要なのは, 扱えるメモリの上限を決めるからです (図 1.7). 32-bit の CPU が区別できるメモリアドレスは 2^{32} 通り, つまり 4GB 分しかありません. どれだけ RAM を積んでも, 1 つのプログラムは 4GB までしか使えないのです. 64-bit ではこの上限が 2^{64} バイト (約 1600 万 TB) となり, 事実上無制限になりました.

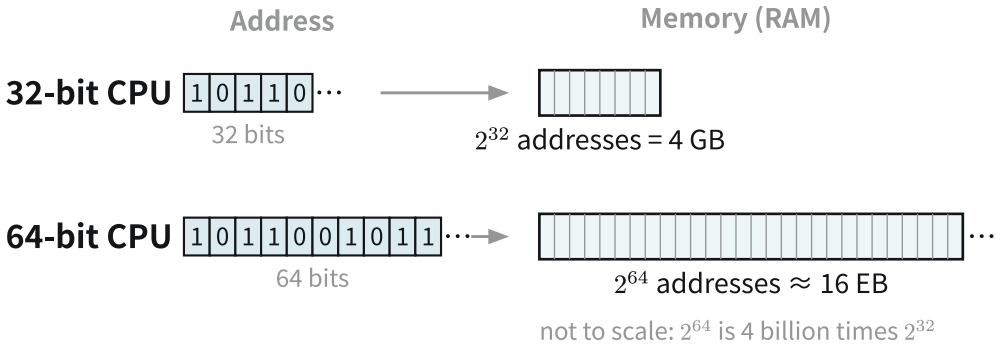


図 1.7: アドレス幅と扱えるメモリの上限

[3] 64-bit への拡張を設計したのは Intel ではなく後発の AMD だったため, x86-64 は AMD64 とも呼ばれます.

現在の PC とスマートフォンは, ほぼすべて 64-bit (x86-64 か ARM64) です. ただし名残はあちこちに残っていて, ダウンロードページの `i386` や `x86` は 32-bit 版を, `x86_64` や `x64` は 64-bit 版を指しています.

> **i** 64-bit 時代の 32-bit 整数
>
> 64-bit CPU の上でも, R の整数型 (integer) は 32-bit のままです.
>
> ```
> .Machine$integer.max
> ## [1] 2147483647
> .Machine$integer.max + 1L
> ## [1] NA
> ```
>
> この上限 ($2^{31}-1$, 約 21 億) を超える整数はオーバーフローして `NA` になります. 大規模な行政データでは ID や人数の合計が 21 億を超えることがあり, その場合は double (64-bit 浮動小数点) や文字列として扱うのが安全です.

1.2.3　バイナリと互換性

ソースコードを機械語に翻訳することをコンパイル (compile) といい, できあがった実行ファイルをバイナリ (binary) と呼びます. ここで重要なのは, バイナリは特定の ISA の機械語で書かれているという点です. x86-64 向けにコンパイルされたバイナリは, そのままでは ARM64 の CPU で動きません. 語彙の違う言語で書かれた指示書のようなものだからです.

そのため, 同じソフトウェアでもアーキテクチャごとに別のバイナリが配布されます. たとえば R の macOS 向けインストーラには, Apple Silicon 用 (arm64) と Intel Mac 用 (x86_64) の 2 種類があります. 自分の機種に合わない方を選ぶと, 動かないか, 動いても遅くなります.[4]

1.3　OS

1.3.1　OS の役割

オペレーティングシステム (OS) は, ハードウェアとアプリケーションの間に立つ土台のソフトウェアです (図 1.8). CPU 時間を各プログラムにどう割り振るか (プロセス管理), RAM をどう配分するか (メモリ管理), ストレージ上のデータをどうファイルとして見せるか (ファイルシステム), キーボードやネットワークとどうやり取りするか (デバイス管理) を一手に引き受けます.

[4] macOS には Rosetta 2 という変換レイヤーがあり, x86-64 のバイナリを ARM64 の命令に翻訳しながら実行できます. Intel Mac 用のソフトが Apple Silicon でも「一応動く」のはこのおかげですが, ネイティブなバイナリより遅くなります.

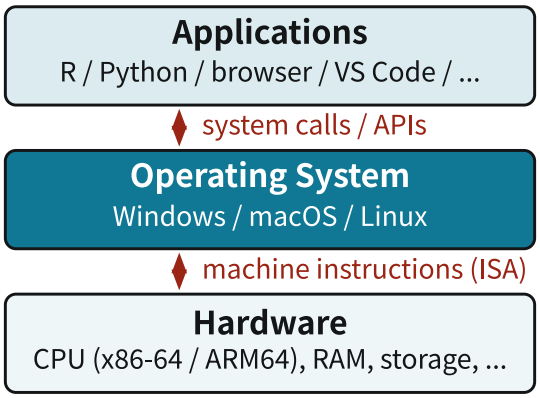


図 1.8: ハードウェア, OS, アプリケーションの階層

アプリケーションはハードウェアを直接触らず, OS が用意した窓口 (システムコール) を通じて機能を利用します. この窓口の仕様が OS ごとに異なるため, バイナリは ISA だけでなく OS にも縛られます. つまり, 配布されるバイナリは「OS × アーキテクチャ」の組み合わせごとに作られます. 身近な機種での組み合わせは 表 1.2 のとおりです.

表 1.2: 身近な機種の OS とアーキテクチャ

機種	OS	アーキテクチャ
Mac (Apple Silicon, 2020 年以降)	macOS	ARM64
Mac (Intel, 2020 年以前)	macOS	x86-64
Windows PC の大半	Windows	x86-64
Copilot+ PC (Snapdragon 搭載)	Windows	ARM64
計算サーバー・クラウド・スパコン	Linux	x86-64 (ARM64 も増加中)
iPhone / Android スマートフォン	iOS / Android	ARM64

⚠ Surface にはご用心

「Windows PC ならば x86-64」という経験則には例外が増えています. Microsoft の Surface シリーズは, 同じ製品ラインの中に Intel 製 CPU (x86-64) のモデルと Snapdragon 搭載 (ARM64) のモデルが混在しており, 見た目からは区別がつきません. 特に 2024 年以降の Copilot+ PC を名乗る Surface (Surface Pro 11 や Surface Laptop 7 など) は ARM64 です.

ARM64 の Windows でも x86-64 用のソフトはエミュレーションで一応動きますが, 速度は落ち, ドライバや一部のソフトはそもそも動きません. 研究用ソフトは ARM64 版 Windows への対応が遅れがちで, たとえば本書の執筆時点では, CRAN が配布する Windows 版 R のバイナリは x86-64 用のみです. Windows 機の購入時には, CPU が Intel / AMD (x86-64) か Snapdragon (ARM64) かを必ず確認しましょう. 手元の Windows 機で確かめるには, 「設定 > システム > バージョン情報」のシステムの種類に ARM と書かれていないかを見ます.

1.3.2 Unix の系譜

現在の主要な OS は Windows, macOS, Linux の 3 つです. この 3 つの関係を理解する鍵が, 1969 年にベル研究所で生まれた Unix という OS です (図 1.9).

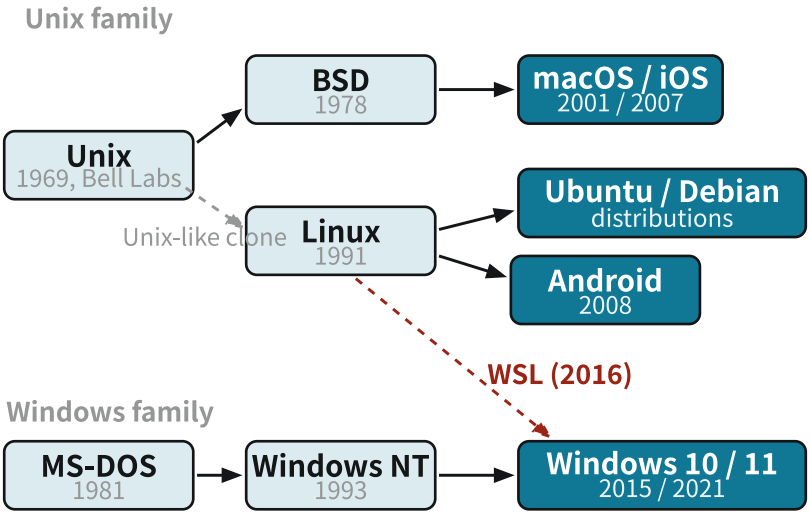


図 1.9:　OS の系譜

- **macOS**: Unix の直系の子孫である BSD を土台にしています. その中核部分の名前が Darwin で, 先ほど `Sys.info()` の `sysname` に表示されていたものです. iOS も同じ土台の上に作られています.
- **Linux**: 1991 年に Linus Torvalds が Unix を手本にゼロから書いた互換 OS (Unix-like) で, オープンソースで開発されています. 厳密には Linux はカーネル (OS の中核) の名前で, 利用者には Ubuntu や Debian などのディストリビューションという形で配布されます. Android も Linux カーネルの上に作られています.
- **Windows**: MS-DOS から Windows NT へと続く, Unix とは独立した系譜です.

この系譜は歴史の豆知識ではなく, そのまま実務に効いてきます. macOS と Linux は同じ Unix の流れを汲むため, コマンド, ディレクトリ構造, 開発ツールがほぼ共通です. そして研究計算の世界では Linux が標準です. 大学の計算サーバー, スーパーコンピュータ, クラウドは, ほぼすべて Linux で動いています.[5] 手元の Mac で書いた解析コードが計算サーバーでもほぼそのまま動くのは, この共通性のおかげです.

1.3.3　Windows と WSL

Windows は Unix の系譜から外れているため, シェルもコマンドも互換性がありません. この溝を埋めるのが WSL (Windows Subsystem for Linux) です. 現行の WSL2 は, Windows 上の軽量な仮想マシンで本物の Linux カーネルを動かす仕組みで, Windows のデスクトップを使いながら完全な Linux 環境 (通常は Ubuntu) を手に入れられます. VS Code は WSL 内のファイルをシームレスに開けるため, 編集は Windows 側, 実行は Linux 側という開発スタイルが自然に実現します. 本コースで Windows ユーザーに WSL を推奨しているのはこのためです.

1.4　Takeaways

最後に, この章の知識が役立つ場面をまとめます.

[5] スーパーコンピュータの性能ランキング TOP500 に載る計算機の OS は, 2017 年 11 月のリスト以降すべて Linux です (Sublist Generator で List と Operating System Family を指定すると確認できます).

- ソフトウェアのインストール: ダウンロードページでは「OS × アーキテクチャ」で選びます. Apple Silicon の Mac なら `arm64` / `aarch64`, それ以外の PC ならたいてい `x86_64` / `x64` / `amd64` です.
- エラーメッセージの解読: `wrong architecture` や `cannot execute binary file` といったエラーを見たら, ISA か OS の不一致を疑いましょう.
- サーバー・クラウドの利用: リモート環境はほぼ Linux で, 多くは x86-64 です. 手元の Mac (macOS + ARM64) でビルドしたバイナリをコピーしても動きません. 環境ごとにインストールし直すか, コンテナを使います.[6]
- AI への指示: 環境構築を AI に頼むときは, 「Apple Silicon の macOS」「WSL2 上の Ubuntu」のように OS とアーキテクチャを伝えると, 自分の環境に合った手順が返ってきやすくなります.

[6] Docker などのコンテナは OS（ディストリビューション）の差を吸収してくれますが, ISA の差は吸収しません. Apple Silicon 上で x86-64 用のイメージを動かすとエミュレーションになり, 大幅に遅くなります.

第 2 章
環境の再現性

この講義では rig で R を入れ rv でパッケージを揃えることを推奨しています. この章では, その裏で何が起きているのか, 「モダンな」パッケージ管理ツールの仕組みを学んでいきます. 半年後の自分や共同研究者が数コマンドで同じ計算を再現できる環境を作ることが目標です.

2.1 パッケージの依存関係

環境をそろえるのがなぜ難しいのかは, パッケージの依存関係を見ると分かります. パッケージは単独では動かず, 1 つ入れようとすると芋づる式に他のものまで必要になります. しかもその依存には, 性質の違う 3 つの種類があります (図 2.1).

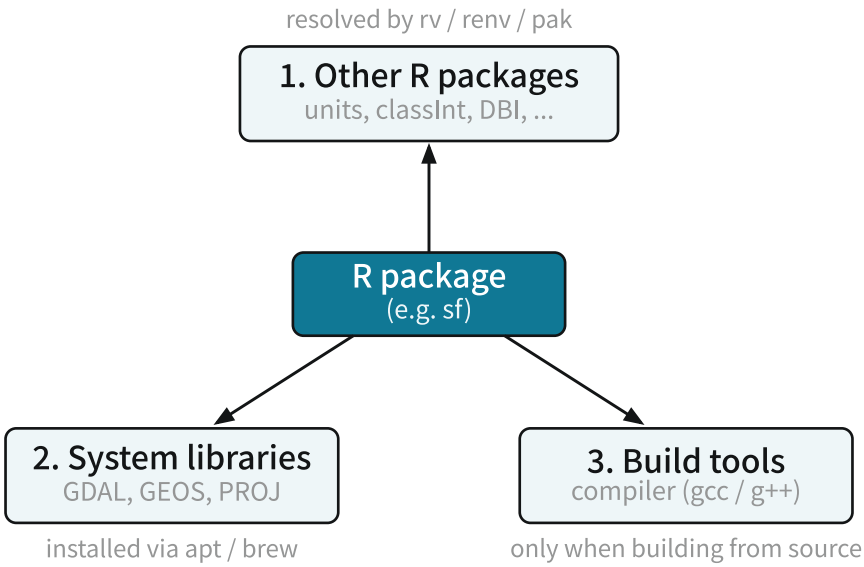


図 2.1: パッケージの 3 種類の依存関係

- **1. 他の R パッケージへの依存**: 一番よくある依存です. たとえば `ggplot2` は `rlang`, `scales`, `gtable`, `farver` などに依存し, それらがさらに別のパッケージに依存します. R のパッケージ管理ツール (install.packages / pak / rv / renv) が, この依存の木を自動でたどって一緒に入れ, ロックファイルに全体を記録します. R の世界の中で完結する依存です.
- **2. 他のアプリケーション・システムライブラリへの依存**: R の外にある C ライブラリや外部プログラムに依存するパッケージもあります. これらは R パッケージではないので, R のツールだけ

では入りません. たとえば空間データの `sf` は GDAL・GEOS・PROJ を必要とします. OS のパッケージマネージャ (セクション 1.3 の apt や Homebrew) で先に用意します.

- **3. ビルドに必要な依存**: パッケージをソースからインストールするとき, 中に C/C++/Fortran のコードがあれば, それを機械語に変換するコンパイラ (セクション C.2) が要ります. 実行時ではなく, インストール時にだけ必要な「道具」への依存です. Debian / Ubuntu なら `build-essential` (gcc / g++ / make) がこれにあたります.

この 3 つは責任の持ち主が違います. 1 は R のパッケージ管理ツールとロックファイルが引き受けますが, 2 と 3 は R の外 (OS 側) の話なので, ロックファイルだけでは再現しきれません. この章の残りは, この見取り図に沿って進めます. まず土台となる R 本体のバージョンをそろえ, つづいて依存 1 (R パッケージ), 依存 2 (システムライブラリ), 依存 3 (ビルドの道具) を順に見ていきます.

2.2　R 本体のバージョン

あるプロジェクトが特定の R のバージョンを要求する場合があります. また, 昔のプロジェクトを再現するために, 古い R のバージョンを使う必要があることもあります. OS のパッケージマネージャ (セクション 1.3) で R を 1 つだけ入れると, この要求の食い違いに対応できません. そこで, 複数のバージョンを並べて置き, 切り替えるための専用ツールを使います.

R では rig (R Installation Manager) がこれにあたります. 複数の R を独立に並置し, どれを既定にするかを切り替えます.

```
rig add release      # install the latest stable R
rig add 4.4          # keep an older version side by side
rig list             # show installed versions
rig default 4.6      # switch the `R` / `Rscript` on PATH
```

仕組みは単純で, rig は各バージョンを別々の場所にインストールし, `rig default` が選んだバージョンへの参照 (PATH 上の `R`) を張り替えるだけです. そのため切り替えは一瞬で, アンインストールも安全です. どのバージョンを使うかは, 後述の宣言ファイルにプロジェクト単位で書いておけます (このリポジトリなら `rproject.toml` の `r_version = "4.6"`). この「ランタイムのバージョンを並べて切り替える」道具は, R に限らずどの言語にもあります (のちほど「他の言語も同じ構図」で扱います).

2.3　他の R パッケージ

R 本体をそろえたら, 次は R パッケージ (依存の種類 1) です. すべてのプロジェクトが 1 つの共有ライブラリにパッケージを入れると, プロジェクト A のための更新がプロジェクト B を壊す, という事故が起きます. 解決策は, プロジェクトごとに独立したライブラリを持ち, その中身をロックファイルに正確に記録することです.

仕組みの中心は, プロジェクト起動時に走る `.Rprofile` です. R はプロジェクト直下の `.Rprofile` を起動時に読み込むので, ここでプロジェクト専用ライブラリを `.libPaths()` の先頭に差し込みます. す

ると `library()` はまずそのプロジェクトのライブラリを見にいき, ユーザー全体のライブラリから隔離されます. プロジェクトを開くだけで, そのプロジェクト用の環境に自動で入る, というわけです.

R には成り立ちの違う 2 つのツールがあります.

- **renv**: 確立された標準です. `renv::snapshot()` で「いま入っているパッケージ」を `renv.lock` に記録し, `renv::restore()` でそれを別のマシンに正確に再現します. スナップショット型で, 手元の状態を写し取る発想です.
- **rv**: 新しい Rust 製のツールで, このリポジトリが採用しています. 宣言型で, 欲しいパッケージを `rproject.toml` に書き並べ, `rv sync` を実行すると, 依存を解決してライブラリを構築し, `rv.lock` に固定します. 「入れた結果を記録する」renv に対し, 「欲しい状態を宣言し, ツールにそれを実現させる」発想です. 後述する uv や cargo に近い考え方です.

このリポジトリの `rproject.toml` は, 使う R のバージョンと, 取得元, 欲しいパッケージを宣言しています.

```
[project]
name = "workshop-graduate-2026"
r_version = "4.6"

repositories = [
    {alias = "CRAN", url = "https://cloud.r-project.org/"},
    {alias = "r-multiverse", url = "https://community.r-multiverse.org"},
]

dependencies = [
    "dplyr",
    "ggplot2",
    "tinytable",
    "fixest",
    # ...
]
```

`rv sync` を実行すると, この宣言を満たすようにパッケージが解決・インストールされ, 実際に選ばれた正確なバージョンが `rv.lock` に書き込まれます. `rproject.toml` が「欲しいもの」, `rv.lock` が「実際に入ったもの」で, 共同研究者は `rv.lock` から寸分違わぬ環境を復元できます. `.Rprofile` はこのプロジェクトライブラリを起動時に有効化する役割を担います.

2.3.1 どこに何が保存されるか

rv も renv も, パッケージを 2 か所に分けて置きます. ひとつはプロジェクト専用のライブラリ, もうひとつは全プロジェクトで共有するキャッシュです. このリポジトリ (rv) の実際の配置は次のようになっています.

```
workshop-graduate-2026/            # project root
├── .Rprofile                      # runs rv/scripts/activate.R on startup
├── rproject.toml                  # declared packages          ← committed
├── rv.lock                        # exact locked versions      ← committed
└── rv/
    ├── scripts/activate.R         # prepends project lib to .libPaths()  ← committed
    └── library/4.6/arm64/         # project-local library      ← gitignored
        ├── dplyr/                   (materialized from the cache)
        ├── ggplot2/
        └── ...

~/.cache/rv/                        # cache shared across all projects
├── 66bf25474b/                     (each package version stored once)
├── abfa7ade75/
└── ...
```

ポイントは 2 つあります. まず, プロジェクトライブラリ (`rv/library/4.6/arm64/`) は R のバージョンとプラットフォームごとに分かれていて, `.Rprofile` がこのパスを `.libPaths()` の先頭に差し込むことで, このプロジェクトの R だけがここを参照します. つぎに, パッケージの実体は共有キャッシュ (`~/.cache/rv/`) に版ごとに 1 回だけ保存され, 各プロジェクトのライブラリはそこから用意されます. このとき rv はキャッシュからコピーし (同じディスク上なので, ファイルシステムによっては copy-on-write で安価です), renv はキャッシュ実体へのシンボリックリンクを張ります. いずれもパッケージを再ビルドしないので, 2 つ目以降のプロジェクトではインストールがほぼ一瞬になり, ディスクも節約できます. 図 2.2 は, ひとつの共有キャッシュを複数のプロジェクトが利用し, それぞれが `.libPaths()` を通じて自分のライブラリだけを見る様子です.

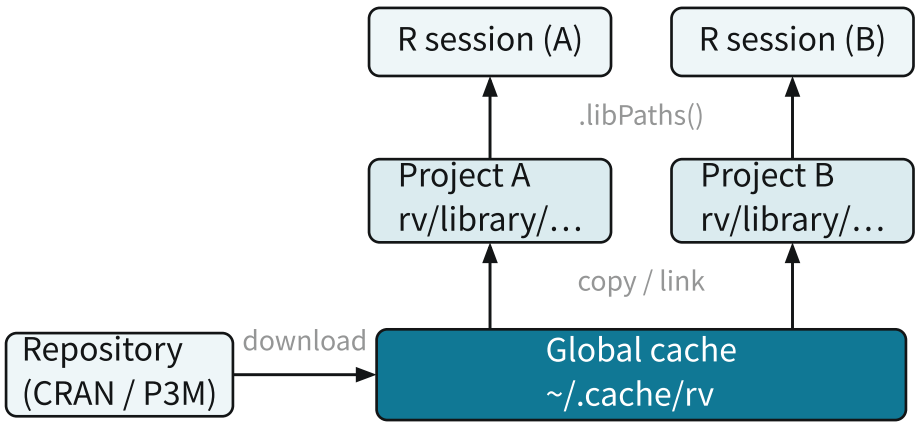


図 2.2: 共有キャッシュとプロジェクトライブラリ

git にコミットするのは, 宣言ファイル (`rproject.toml`), ロックファイル (`rv.lock`), 有効化スクリプト (`rv/scripts/`) だけです. ライブラリ本体 (`rv/library/`) は `.gitignore` で除外し, 各マシンで `rv sync`

が rv.lock から作り直します. 「環境そのもの」ではなく「環境の作り方」を共有する, というのが要点です. renv もまったく同じ構図で, renv/library/ … がプロジェクトライブラリ, ~/.cache/R/renv/ が共有キャッシュ, renv.lock が固定, という対応になります.

2.3.2 install.packages と pak

rv や renv も, 最終的には 1 つ 1 つのパッケージをどこかのライブラリにインストールしています. その最も基本的なコマンドが, R に標準で付いてくる install.packages() です. 呼ぶと, パッケージを取得して .libPaths() の先頭 (書き込める最初の場所) にインストールします.

```
install.packages("dplyr") # install into the first writable .libPaths()
```

install.packages() は確実ですが, 依存パッケージを 1 つずつ順に取ってくるため遅く, パッケージが必要とする OS 側のライブラリ (依存の種類 2) までは面倒を見ません. これを現代化したのが pak です. pak は依存関係をまとめて解決し, 並列にダウンロード・インストールするので速く, CRAN・Bioconductor・GitHub・URL を同じ書き方で扱え, 足りないシステム依存も教えてくれます.

```
pak::pak("dplyr")            # from CRAN
pak::pak("tidyverse/dplyr")  # from GitHub, same function
```

GitHub など CRAN 以外にあるパッケージは, install.packages() では入れられません. 従来は remotes::install_github("user/repo") を使ってきました (devtools::install_github() も中身は remotes で, devtools は開発用ツールの詰め合わせパッケージです). pak::pak("user/repo") は同じことをより速く行い, CRAN も GitHub も同じ関数で扱えるので, これから使うなら pak に一本化すると覚えることが減ります.

じつは rig で R を入れると, この pak も自動で一緒に入ります (rig add の既定で, --without-pak で無効化できます). そのため追加の準備なしに, R を入れた直後から pak::pak() が使えます. 手動で 1 つ足すだけなら pak が快適です. ただしプロジェクトの文脈では, これらを直接叩くよりも, rv や renv 経由で入れて rv.lock / renv.lock に記録するのが原則です (rv は独自のインストーラを内蔵し, renv は内部で pak を使えます). 手で入れたパッケージはロックファイルに残らず, 再現性の穴になるからです.

2.4 システムライブラリ

他の依存として, R の外にある C ライブラリや外部プログラムも考えられます. R パッケージではないので, install.packages() も rv も入れてくれません. これらは OS のパッケージマネージャ (セクション 1.3 の apt や Homebrew) で, プロジェクトとは別に用意します.

代表的な例を挙げます. 空間データの sf は GDAL・GEOS・PROJ, curl は libcurl, xml2 は libxml2, 画像処理の magick は ImageMagick, 文字描画の ragg / textshaping は freetype・harfbuzz を必要とします. sf が入らないときの多くは, R の問題ではなく, これらの C ライブラリが OS に無いことが原因です.

どのシステムライブラリが要るかは, pak が教えてくれます. `pak::pkg_sysreqs()` は, そのパッケージが必要とする OS パッケージと, それを入れるコマンドを返します (Ubuntu なら `apt install ...` にあたります).

```
pak::pkg_sysreqs("sf") # report the OS libraries sf needs, and how to install them
```

注意点として, ロックファイル (`rv.lock` など) はこの層を記録しません. そのため, 必要なシステムライブラリは README に書いておくか, コンテナ (Docker など) に固めるなどして, R パッケージとは別に共有する必要があります.

2.5　ソースかバイナリか

依存の種類 3 は, パッケージをソースからビルドするときに要るコンパイラ (セクション C.2) でした. これが必要になるかどうかは, パッケージをソースで取るか, ビルド済みのバイナリで取るかで決まります. R のパッケージは 2 つの形で配られます. ひとつはソース (`.tar.gz`) で, どの OS でも使えますが, 中の C/C++/Fortran コードはインストール時にコンパイルが必要です. もうひとつはバイナリで, ある OS と R のバージョン向けにコンパイル済みのため, 展開するだけで入ります.

	ソース	バイナリ
中身	未コンパイルのコード	コンパイル済み
インストール	ビルドが必要 (要コンパイラ)	展開するだけ
速度	遅い	速い
対応範囲	どの OS・R 版でも	OS・R 版に固定
主な入手元	CRAN (Linux の既定)	CRAN (Win / Mac), P3M (Linux)

どちらが使われるかは, 既定でほぼ自動的に決まります. `install.packages()` が取る型は `getOption("pkgType")` で決まり, Windows と macOS では実質バイナリ優先, Linux では `"source"` (ソースからビルド) です. これは配布側の事情を反映しています. CRAN は Windows と macOS にはバイナリを提供しますが, Linux にはソースしか提供してこなかったからです. そのため WSL などの Linux では, 既定だとパッケージをソースからビルドすることになり, コンパイラ (build-essential) やシステムライブラリのヘッダが必要になります. 環境構築で `build-essential` を入れたのはこのためです. ビルドには時間もかかります.

Linux でもバイナリを使う方法があります. Posit Public Package Manager (P3M) は, ディストリビューションごとにビルド済みのバイナリを配布しています. `rproject.toml` (や renv の設定) の `repositories` にその URL を指定すると, rv や renv はソースの代わりにバイナリを取得し, インストールが桁違いに速くなります. コンパイル環境の細かな違いに悩まされることも減ります. 速さと再現性の両立という点で, 共同作業や CI ではバイナリ配布のリポジトリを使うのが定石です.

2.6　Python / Julia

Python や Julia といった他の言語でも, 依存関係の管理は同じ構図です. ランタイムのバージョンを切り替えるツール, 欲しい依存を書く宣言ファイル, 実際の版を固定するロックファイルの三点セットです. Python の uv や Julia の Pkg は, R の rv / renv と同じ考え方で動きます.

言語	ランタイム管理	宣言ファイル	ロックファイル
R	rig	`rproject.toml` (rv) / DESCRIPTION (renv)	`rv.lock` / `renv.lock`
Python	uv, pyenv	`pyproject.toml`	`uv.lock`
Julia	juliaup	`Project.toml`	`Manifest.toml`

Python の uv は, この構図を 1 つのツールにまとめた好例です. Python 本体のバージョンとパッケージの両方を管理し, `pyproject.toml` に依存を宣言して `uv.lock` に固定します. 従来は pyenv (バージョン), venv (隔離環境), pip (インストール), poetry (依存解決) と役割ごとに分かれていたものを, Rust 製の高速な 1 つのツールに統合しています.

```
uv init            # start a project (creates pyproject.toml)
uv add polars      # declare and install a dependency
uv sync            # reproduce the environment from uv.lock
uv run script.py   # run inside the project environment
```

Julia は言語自体にパッケージマネージャ Pkg を内蔵しています. `Project.toml` に直接の依存を書き, `Manifest.toml` が依存の依存まで含めた完全なツリーを固定します. 環境の有効化は `julia --project` や `Pkg.activate()` で行い, `Pkg.instantiate()` が `Manifest.toml` から環境を復元します.

宣言ファイルとロックファイルを分ける理由は共通しています. 宣言ファイルには人間が読める大まかな要求 (「dplyr が欲しい」) を書き, ロックファイルには機械が解決した正確な版 (「dplyr 1.1.4, この取得元, このハッシュ」) が入ります. 前者は git で共有して意図を伝え, 後者は git で共有して環境を一致させます. どちらもコミットするのが原則です.

2.7 このプロジェクトでの実践

以上を踏まえると, このリポジトリの環境を復元する手順はごく短くなります.

```
rig add release                # 1. get R via rig
git clone <this repository>    # 2. clone the project
cd workshop-graduate-2026
rv sync                        # 3. build the project library from rv.lock
```

`rv sync` の後は, `.Rprofile` がプロジェクトライブラリを有効化するので, R を起動すればそのまま同じパッケージ環境で作業できます. 日々の開発でパッケージを増やすときは, `rproject.toml` に書き足して `rv sync` するか, `rv add <package>` を使います. どちらも `rv.lock` を更新するので, その差分をコミットすれば, 環境の変更履歴も git に残ります.

使い分け

- 新しく R のプロジェクトを始めるなら, 宣言型で速い rv が扱いやすいです. 既存資産や共同研究者が renv を使っているなら renv に合わせます.
- ロックファイル (`rv.lock` / `renv.lock` / `uv.lock` / `Manifest.toml`) は必ず git にコミットします. これが再現性の実体です.
- 環境の再現は, 計算の再現 (targets の章) と対になる話です. 同じ環境の上で同じパイプラインを回して, はじめて結果が完全に再現します.

第 3 章
Git & GitHub

3.1　なぜ Git を使うのか

研究の分析コードは, 一度書いて終わり, ということはまずありません. データの前処理をやり直したり, 推定の仕様を変えたり, 図を作り直したりと, 何度も書き換えていきます. その過程で「昨日まで動いていたのに今日は動かない」「論文のあの数字を出したのはどの版だったか」といった事態は, 誰しも一度は経験します. Git は, こうしたコードの変更履歴を管理するための道具です.

Git を使うと, 主に次の 4 つのことができるようになります.

- **Recording**: 自分と共同研究者の変更履歴を, すべて残せます.
- **Restore**: 過去の任意の時点の状態に, いつでも戻れます.
- **Compare**: 2 つの時点の違いだけを取り出して, バグの原因を絞り込めます.
- **Branch**: 完成した部分と作業中の部分を, 分けて進められます.

3.1.1　AI 時代のバージョン管理

経済学では個人で全体像がギリギリ把握できるサイズのプロジェクトが多いので, Git は十分普及してきませんでした. しかし, AI にコードを書かせられる今, 人間の直感よりも高速にコードを書き換えられるようになりました. そうなると, 変更履歴の管理の重要性はむしろ増しています.

また, Git 自体の複雑さも, AI に操作させることで解消できるようになりました. Git はトラブル解決時に必要となるコマンドや知識が多く, 初心者にとっては大きなハードルでした. しかし, そうした面倒は AI に任せて仕舞えば, 人間は Git の恩恵だけを受けられるようになりました.

3.1.2　Git と GitHub

Git と GitHub は名前が似ているため混同されがちですが, 別のものです. Git は手元のマシンで動くバージョン管理ツール (コマンドラインのアプリ) で, GitHub はそのコミット履歴を公開・共有し, 共同作業をするための Web サービスです(図 3.1).[7]

[7]そのため, GitLab といった同種の Web サービスもあります.

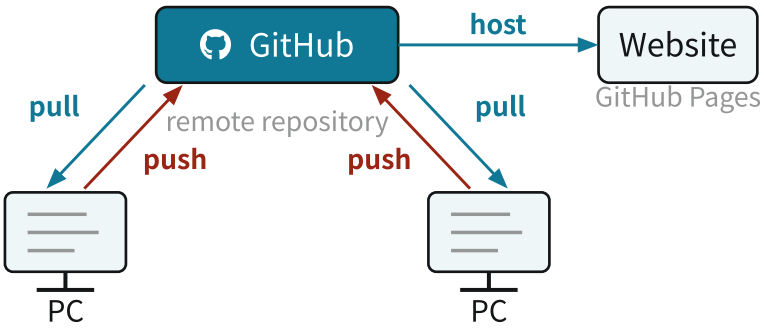


図 3.1: GitHub を介した履歴の共有と公開

3.1.3 コマンドライン

Git を操作する GUI アプリは数多くありますが (VSCode や RStudio にも), 私は学習の段階ではコマンドライン操作で学ぶことをお勧めしています. Git の仕組みがそもそも非直感的なので, GUI といった直感的なツールでは裏で起こっていることをマスクしてしまって, 帰って仕組みの理解を妨げるからです. また, トラブルが起きた時も GUI からは解決できないことが多く, コマンドライン操作の知識が結局必要です. コマンドライン操作を理解することが Git を理解することと同義なので, AI 時代であってもコマンドラインから学ぶことをお勧めします.

3.2 Git の基礎概念

3.2.1 レポジトリ

Git によるバージョン管理は, レポジトリ (repository) という単位で行われます. Git が一塊として管理していく単位のことなので, ほぼプロジェクトフォルダと考えてもらって構いません.

Git のプロジェクトを始めるにはプロジェクトフォルダ内で次のようにします.

```
git init
```

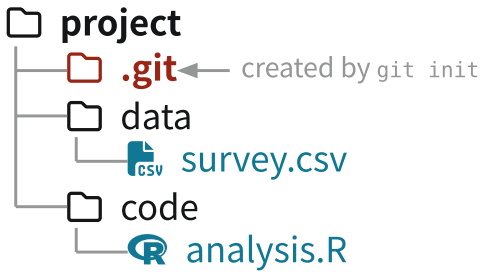


図 3.2: リポジトリ

この時, `.git` という隠しフォルダが作られます. 以降, Git はこのフォルダの中に, プロジェクトフォルダの状態を記録していきます. つまり, Git はプロジェクトフォルダの中身を丸ごと保存する仕組みです. なお, 競技にはこの `.git` フォルダのことを「リポジトリ」と呼ぶこともありますが, ここではプロジェクトフォルダ全体を指す意味で使います.

3.2.2 コミット

Git でもっとも大切な概念がコミット (commit) です. コミットとは, プロジェクトフォルダのある時点の状態を丸ごと保存したセーブポイントだと考えてください. 「Git を使う」とは, 突き詰めればこのコミットを一つずつ積み重ねていく作業にほかなりません (図 3.3).

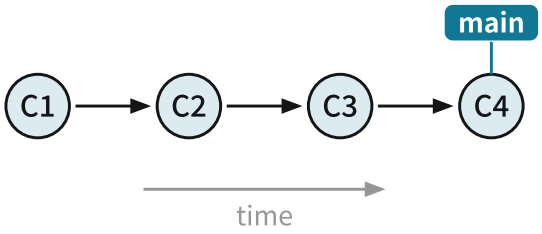


図 3.3: コミット

デフォルトでは `main` という名前のブランチ (後述) に 1 つ目のコミットが作られます.

HEAD

いま自分がどのコミットを見ているかを指す目印を HEAD と呼びます. HEAD を過去のコミットに移すと, フォルダの中身はその時点の状態に戻ります (図 3.4). 一度作ったコミットは消えないので, 失敗を恐れずに思い切って実験できます. この「いつでも戻れる」という安心感こそ, セーブポイントを積む意味です.

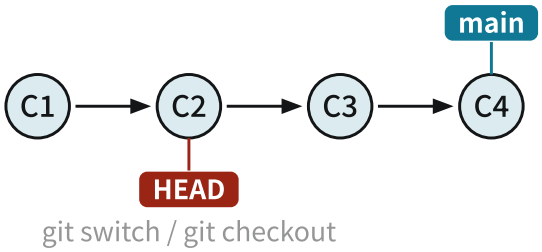


図 3.4: 過去のコミットに戻る

コミットの比較

任意の 2 つのコミットの差分 (diff) を取れば, 「どこが, どう変わったか」だけに注目できます (図 3.5). これはバグの発見やコードレビューで大きな助けになります. 例えば, 昨日まで出ていた数字が今日は変わってしまったとき, その間のコミットの差分を見れば, 原因の見当がつきます.

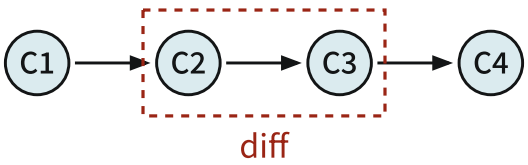


図 3.5: コミット間の差分 (diff)

i コミットの実体

私たちは「コミットは変更点 (差分) を記録している」と思いがちですが, 実際は違います. コミットが保存しているのは, その時点で Git が追跡しているファイル全体の状態, つまり丸ごとのスナップショットです (図 3.6).[8] 図 3.4 で見た「どの時点にも戻れる」性質は, 各コミットが完全なスナップショットだからこそ成り立ちます.

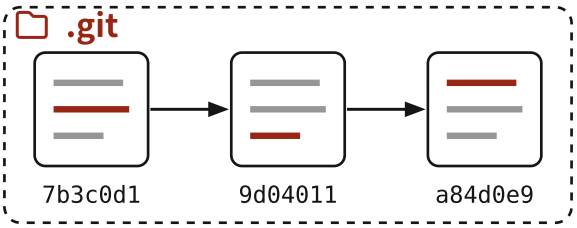


図 3.6: スナップショットとハッシュ名

図でオレンジに塗られた行は, そのコミットで変更されたファイルを表します. 変更の有無にかかわらず, コミットは毎回すべてのファイルの状態を丸ごと記録します. そして個々のコミットには, 人間が付ける連番ではなく, その中身から計算されたハッシュ値[9] (例: `a84d0e9 ...`) が名前として割り当てられます. 中身が少しでも違えば異なるハッシュになるため, 名前が一意に定まり,「このコミット」を確実に指し示せます. ふだんは, 図の各スナップショットの下に並ぶように, 先頭の 7 文字ほど (`a84d0e9`) だけを使ってコミットを指定することが多いです.

3.2.3 ローカルとリモート

手元のマシンにある, `.git` を含むフォルダをローカルリポジトリ (local repository) と呼びます. 図 3.2 のように, `data/` や `code/` といった普段のプロジェクトフォルダの直下に `.git` フォルダが置かれた状態が, ひとつのリポジトリです.

コードを共有したりバックアップしたりするには, GitHub 上のリモートリポジトリ (remote repository) を使います. この 2 つは, ローカルからリモートへ送る push と, リモートからローカルへ取り込む pull によって同期します (図 3.7).

[8] とはいえ, 変わっていないファイルまで毎回コピーするわけではありません. Git は内部で同じ中身のファイルを使い回す (重複排除する) ため, スナップショット方式でも容量はそれほど膨らみません.

[9] セクション C.3 を参照

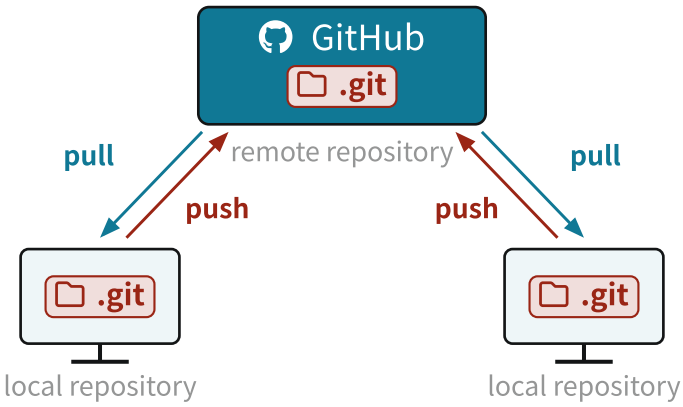


図 3.7: ローカルとリモートの同期

なお, Dropbox や Google Drive などと違って, 自分で明示的に push しない限り, ローカルの変更がリモートに反映されることはありません. つまり, ローカルでいくらコミットを積んでも, push しない限りリモートには何も変化が起きません. 逆も同様で, リモートの変更は pull しない限りローカルには反映されません.

3.3 基本の操作

3.3.1 ステージング

ファイルをいくつ変更しても, それが自動でコミットされるわけではありません. コミットの前に「次のコミットに含める変更」を選び, ステージングエリア (staging area) と呼ばれる場所に載せます. この載せる操作が `git add` で, 載せる作業をステージング (staging) と呼びます (図 3.8). 「次のコミットという舞台 (stage) に, 必要な変更だけを上げる」イメージだと考えてください.

ステージングエリアに載せた変更だけが, 続く `git commit` でコミットになります. 図の `draft.txt` のように, 変更してあっても `git add` しなければコミットには含まれません. こうして, 関係する変更だけをひとまとめにした, 意味のあるコミットを作れます.

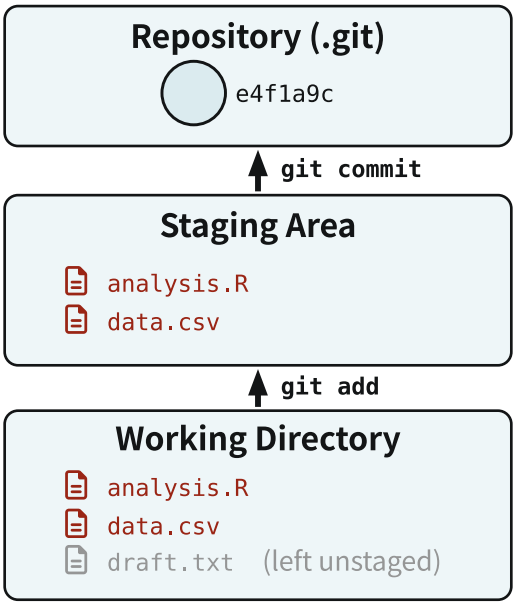


図 3.8: ステージングとコミット

```
git add foo1.txt              # stage one file
git add foo1.txt foo2.txt     # stage several files at once
git add .                     # stage all changed files
```

3.3.2　コミット

ステージしたら, メッセージを付けてコミットします. メッセージは必須で, あとから履歴を読み返すときの手がかりになります. 「何を」変更したのかが一目で分かる短い説明を書きましょう.

```
git commit -m "MESSAGE"
```

3.3.3　ログ

積み重ねたコミットの履歴は `git log` で確認できます. ただし標準の `git log` は, コミットごとにハッシュ・著者・日時・メッセージを数行ずつ表示するため, コミットが増えると一覧性が下がります. 普段は `--oneline` を付けて, 1 コミットを 1 行で表示するのがおすすめです.

```
git log --oneline
```

出力はたとえば次のようになります (上が新しいコミット).

```
a84d0e9 add computation-data
9d04011 migrate to quarto book
6761acb add high-speed
```

各行は, 先頭がコミットのハッシュ (短縮形), 続いてコミットメッセージです. メッセージを「何をしたか」が分かる短文にしておくと, この一覧がそのまま変更の目次になります. 先頭のハッシュは, あとで「どのコミットに戻るか」「どの 2 つを比較するか」を指定するときの目印になります.

3.3.4 Push と Pull

ローカルの変更をリモートへ送るのが push, リモートの変更をローカルへ取り込むのが pull です.

```
git push origin main    # send local commits to the remote
git pull origin main    # bring remote changes into local
```

`main` はブランチ名で, ここではデフォルトの main ブランチを指します. `origin` はリモートリポジトリの名前で, GitHub 上の自分のリポジトリを指します.

コミットからプッシュまでの一連の流れを, 実際に手を動かしてたどってみましょう.

1. GitHub でリポジトリを作ります
2. 研究用なら private リポジトリを選びます.
3. gitignore, README, License は選ばなくて構いません (後から追加できます).
4. `git clone REPO_URL` で, リモートレポジトリをローカルにコピーします.
5. 適当なテキストファイル (例: `foo1.txt`) を作り, 適当な一文 (例: “Hello Git!”) を書きます.
6. ステージを忘れずに, コミットを作ります.
7. リモートリポジトリへ push します. GitHub 上でファイルが更新されていることを確認しましょう.

3.4 ブランチとマージ

3.4.1 ブランチと HEAD

ブランチ (branch) は, あるコミットに付けたラベルのようなものです. よくある誤解として, ブランチは「変更の流れ」を表すものだと思われがちですが, 実際は「コミットの位置」を指すものです. そして HEAD は, いま自分がどのブランチ (どのコミット) にいるかを指します. 例えば, 安定版の main から枝分かれさせた dev のように, 複数のブランチを同時に持てます (図 3.9).

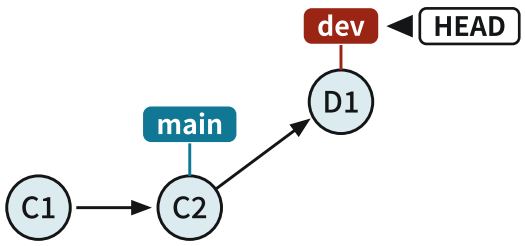


図 3.9: ブランチと HEAD

```
git branch BRANCH_NAME    # create a branch
git switch BRANCH_NAME    # move HEAD to a branch
```

3.4.2 マージ

あるブランチの変更を, いま HEAD のあるブランチへ取り込む操作をマージ (merge) と呼びます. dev を main にマージすると, 両方の変更を併せ持つ新しいコミット (図 3.10 の M) ができます.

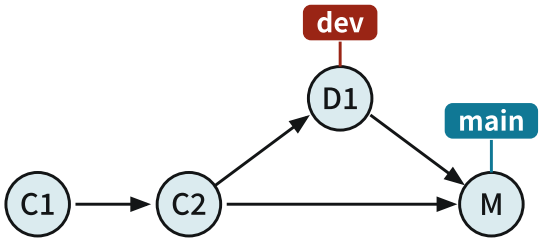


図 3.10: ブランチのマージ

```
git merge BRANCH_NAME
```

3.4.3 なぜブランチを使うのか

ブランチを使う最大の目的は, main ブランチをいつでもきれいに動く状態に保つことです. main が常に動くと分かっていれば, 新しく入れた変更がバグの原因かどうかを切り分けやすくなります. 共同研究者も, それぞれ main からブランチを切って作業します.

> **i** main-dev ワークフロー
>
> 私が勧めるのは, シンプルな main-dev ワークフローです.
>
> - 作業は dev ブランチでのみ行います (共同研究者がいる場合は, 自分の名前をブランチ名にします).
> - 意味のあるまとまりが完成したら, 不要なファイルを片付けてから dev を main にマージします.

3.4.4 2 種類のマージ

まず, 次のような状況を考えます (図 3.11). 手元 (Local) では, main の C2 から dev ブランチを切って D1 というコミットを作りました. 一方リモート (Remote) はまだ C1-C2 のままで, dev も D1 も持っていません. この D1 の変更を main に取り込み, 図 3.10 の M のようなマージコミットをリモートに作ることが目的です.

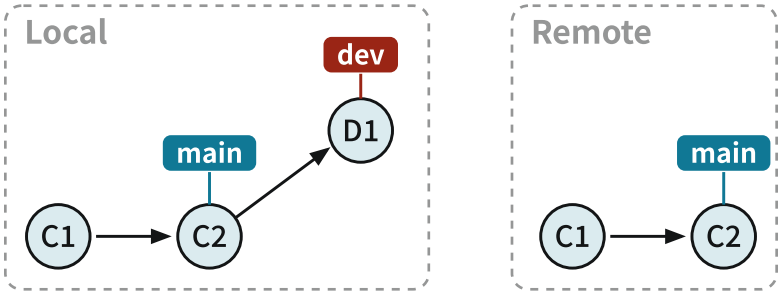


図 3.11: マージ前の Local と Remote

この dev を main に取り込む方法は, 大きく 2 つあります. 1 つ目は, ローカルでマージする方法です. ローカルで dev を main にマージし, その main をリモートへ push します (図 3.12). 手数が少なく, 一人で進めるプロジェクトに向いています.

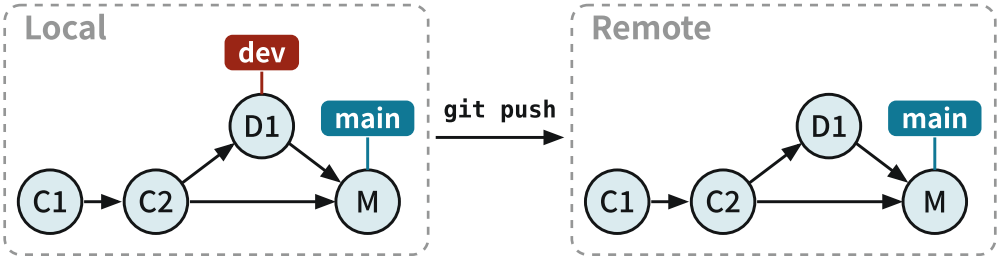


図 3.12: ローカルマージ

2 つ目は, リモートでマージする方法です. dev をリモートへ push し, GitHub 上で Pull Request を作って main に取り込みます (図 3.13). 変更をレビューしてから取り込めるので, 共同作業に向いています (私は一人のときもこちらを使います).

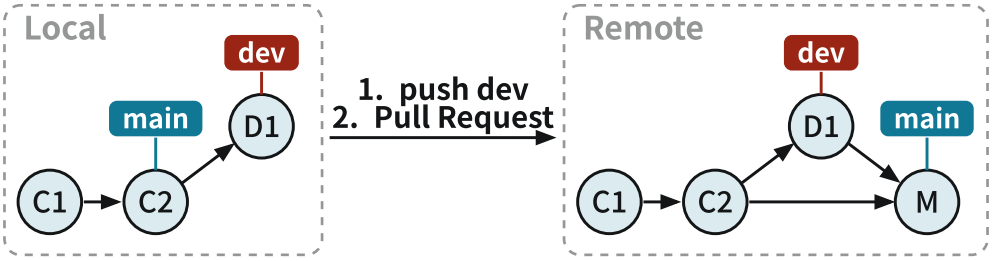


図 3.13: リモートマージ (Pull Request)

2 種類のマージ (ローカルマージとリモートマージ) を, どちらも実際に試してみましょう.

ローカルマージ

1. 新しいブランチ `dev` を作ります
2. HEAD を `dev` に移します
3. いくつかコミットを作ります
4. HEAD を `main` に戻します
5. `dev` を `main` にマージします
6. `origin/main` へ push します
7. (ローカルの) `dev` ブランチを消します (`git branch -d dev`)

リモートマージ

1. `dev` ブランチを作ります
2. `dev` に移ります
3. いくつかコミットを作ります
4. `origin/dev` へ push します
5. GitHub で Pull Request を作ります
6. `dev` を `main` にマージし, リモートの `dev` を消します
7. ローカルに戻り `main` に移ります
8. main を pull し, ローカルの `dev` を消します (`git branch -d dev`)

3.5 コンフリクト

Git を使っていると, 2 つのブランチをマージしたときに, Git がどちらの変更を採用すべきか判断できない場合があります. その状態をコンフリクト (conflict) と呼びます (図 3.14). Git はどちらを採用すべきか判断できないので, 解決を人間に委ねます.

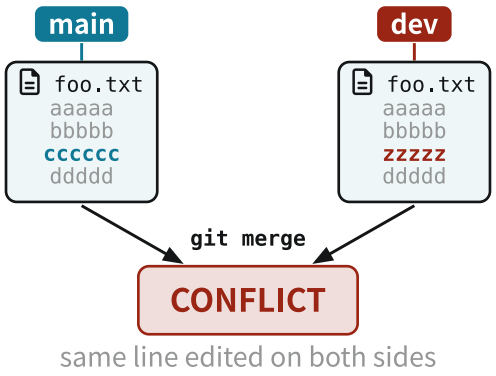


図 3.14: コンフリクト

コンフリクトが起きると, 該当するファイルの中に, 両方の変更が次のように並べて書き込まれます.

```
aaaaa
bbbbb
<<<<<<< HEAD
ccccc
=======
zzzzz
>>>>>>> dev
ddddd
```

解決するには, 残したい方を選び, 不要な行とマーカー (`<<<<<<<` `=======` `>>>>>>>`) を消してファイルを保存し, 解決のためのコミットを作ります. 例えば dev 側の `zzzzz` を残すなら, 次のように整えます.

```
aaaaa
bbbbb
zzzzz
ddddd
```

VSCode では, 「Accept Current Change」「Accept Incoming Change」などのボタンでどちらの変更を残すか選び, 保存してからコミットすれば済みます. なお, 同じファイルを複数人で同時に編集しないというルールを守るだけで, コンフリクトのほとんどは防げます.

3.5.1 Pull

実は pull は, 2 つの操作を続けて行っています (図 3.15).

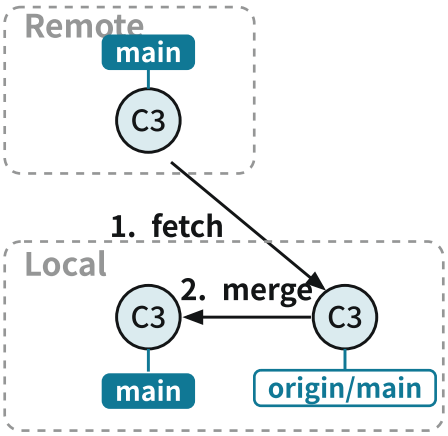


図 3.15: pull = fetch + merge

1. **fetch**: リモートのブランチをローカルへ取得して `origin/BRANCH_NAME` と名付ける
2. **merge**: `origin/BRANCH_NAME` をいまのブランチへ取り込む

```
git pull origin main
# shorthand for the two commands below
git fetch origin main
git merge origin/main
```

中身がマージである以上, pull でもコンフリクトが起こりうる点には注意してください.
3 種類のコンフリクトを自分で起こして, それぞれ解決してみましょう.

ローカルコンフリクト

1. `dev` に移り `foo1.txt` を作って一行 (例: “Tortilla sin Cebolla”) を書き, コミットします.
2. `main` に移り `foo1.txt` を作って別の一行 (例: “Tortilla con Cebolla”) を書き, コミットします.
3. `dev` を `main` にマージしてコンフリクトを起こします.
4. どちらかの行を選んで解決します.

リモートコンフリクト

1. `dev` に移り `foo2.txt` を作って一行 (例: “But the Emperor has nothing at all on!”) を書き, `origin/dev` へ push します.
2. `main` に移り `foo2.txt` を作って別の一行 (例: “Oh! How beautiful are our Emperor’s new clothes!”) を書き, `origin/main` へ push します.
3. GitHub で Pull Request を作り, コンフリクトを起こします.
   - 軽微なコンフリクトなら, GitHub 上で解決できます
   - 複雑なコンフリクトの場合, ローカルで `origin/main` を `dev` にマージし, コンフリクトを解決してから `origin/dev` へ push します.

Pull とコンフリクト

1. `dev` に移り `foo3.txt` を作って一行 (例: “Mountain of Mushroom”) を書き, コミットして `origin/dev` へ push します.
2. GitHub で `dev` を `main` にマージします.
3. ローカルに戻り `main` に移って `foo3.txt` に別の一行 (例: “Village of Bamboo Shoot”) を書き, コミットします.
4. `origin/main` から pull してコンフリクトを起こし, それを解決します.

3.6 研究のワークフロー

ここまでの道具を, 日々の研究で繰り返す一連の流れにまとめます. 私はだいたい次の 6 ステップを回しています. 最初はこの流れをそのまま真似るだけで十分です.

3.6.1 1. ローカルを同期する

作業を始める前に, まず main を最新の状態にしておきます (図 3.16).

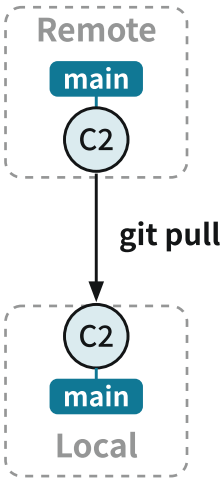


図 3.16: main を最新化

```
git switch main
git pull origin main
```

3.6.2 2. dev ブランチで書く

main から dev を切り, そこで作業します (図 3.17).

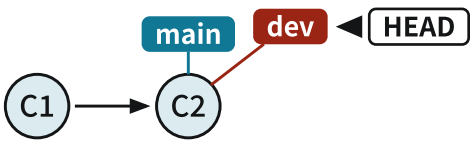


図 3.17: dev ブランチで作業

```
git switch -c dev    # create dev and switch to it (same as: git branch dev && git switch dev)
# ... write code ...
```

3.6.3 3. コミットする

意味のあるまとまりごとに, ステージしてコミットします (図 3.18).

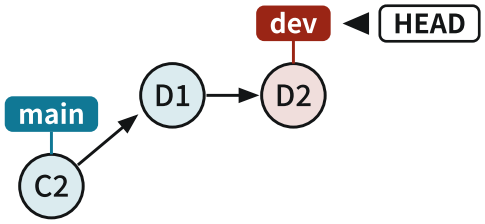


図 3.18: コミットを積む

```
git add foo1.txt foo2.txt    # stage files to commit (git add . for all)
git commit -m "MESSAGE"
```

3.6.4 4. リモートへ Push する

dev をリモートへ送ります (図 3.19).

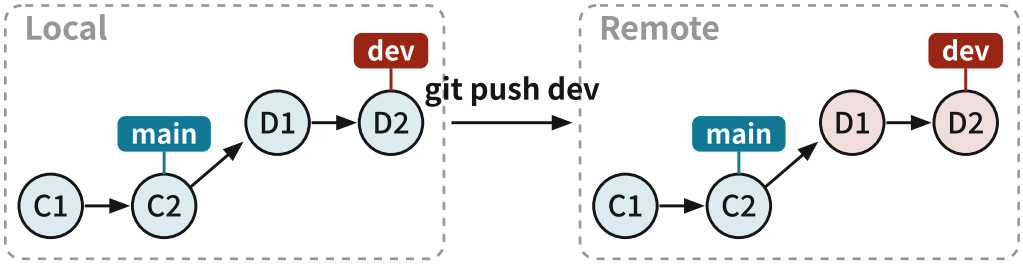


図 3.19: dev を push

```
git push origin dev
```

3.6.5 5. Pull Request とマージ

GitHub 上で Pull Request を作り, dev を main にマージします (図 3.20). マージが終わったら, リモートの dev ブランチを消しておくことを勧めます.

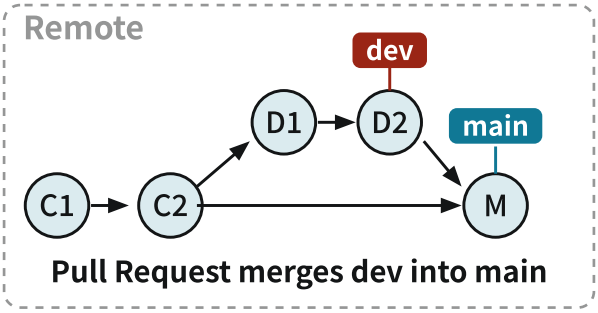


図 3.20: Pull Request でマージ

3.6.6 6. ローカルの main を最新化する

Pull Request でマージが済んだら, ローカルの main に戻って pull し, 役目を終えた dev を消します (図 3.21). これで手元の main にもマージ結果 (M) が反映され, 次の作業に備えられます.

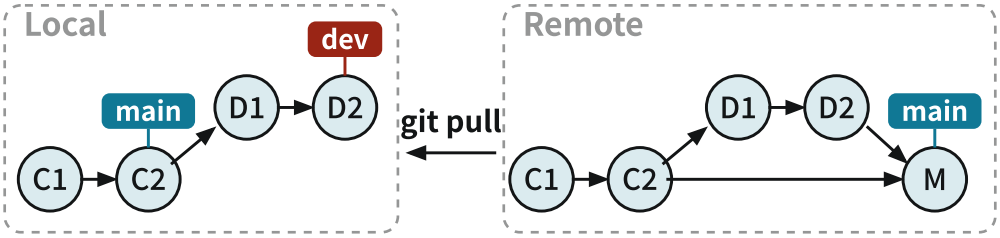


図 3.21: ローカルの main を最新化

```
git switch main
git pull origin main
git branch -d dev    # delete local dev branch
```

上で紹介した研究のワークフローを, 実際に真似てみましょう.

- ステップ 1〜6 を, いくつかコミットを作りながら一通りたどってみましょう.

3.7 困ったときは

git を使う最大の強みは, トラブルが起きた時に過去に戻って再開できることです.

3.7.1 履歴をたどる

まずは `git log --oneline` で, これまでのコミットを一覧します (セクション 3.3.3).

```
git log --oneline
```

```
a84d0e9 add computation-data
9d04011 migrate to quarto book
6761acb add high-speed
```

この一覧を図にすると 図 3.22 のようになります. 各コミットはハッシュ名で識別され, いまの作業位置を指す HEAD は, 最新コミット `a84d0e9` の main 上にあります. この一覧から, どのコミットのあたりで問題が起きたか, どの 2 つを見比べればよいかの見当をつけます.

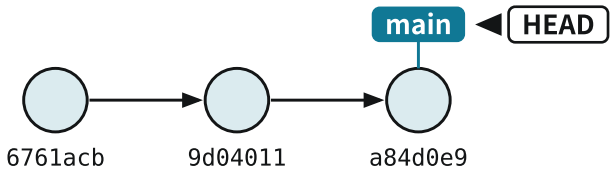


図 3.22:　コミット列と HEAD

3.7.2　変更を見比べる

怪しいコミットの見当がついたら, `git diff` で中身を見比べます. 図 3.5 で見たように, 差分 (diff) を取れば「どこが, どう変わったか」だけに注目できます.

```
git diff                 # uncommitted changes vs the last commit
git diff 9d04011 a84d0e9  # differences between two commits
```

出力では, 取り除かれた行が `-`, 加わった行が `+` で示されます.

```
- tip ← mean(tip_amount)
+ tip ← mean(tip_amount, na.rm = TRUE)
```

この例なら, `na.rm = TRUE` を足したことが変化の原因だと分かります. ここまで切り分けられれば, あとはその場で直すか, 前の状態に戻すかです.

3.7.3　操作を取り消す

戻りたいコミットが分かったら, `git reset` で HEAD をそこへ移します. 行き先は `git log` で調べたハッシュで指定します. `--soft` は変更内容を手元に残したまま HEAD の位置だけを戻し, `--hard` は作業ディレクトリごとその状態に戻します (`--hard` は未コミットの変更を捨てるので注意してください).

```
git reset --soft HEAD~1    # undo the last commit, keep its changes
git reset --hard 9d04011   # discard everything back to commit 9d04011
```

3.7.4　最終兵器: git reflog

`git reset --hard` で戻りすぎたり, コミットを消してしまったりして, `git log` を見ても目当てのコミットが見つからない. そんなときの最終兵器が `git reflog` です. `git log` がいまの HEAD からたどれるコミット (プロジェクトの公式の歴史) しか見せないのに対し, `git reflog` は HEAD がこれまで指してきた位置の足あとを, 操作の種類ごとにすべて見せます.

```
git reflog
```

出力はたとえば次のようになります.

```
a84d0e9 HEAD@{0}: commit: add computation-data
9d04011 HEAD@{1}: reset: moving to HEAD~1
1f2e3c4 HEAD@{2}: commit: experimental change
9d04011 HEAD@{3}: commit: migrate to quarto book
```

各行は, コミットのハッシュ, `HEAD@{N}` (N が何手前の状態か. 0 が最新), そして操作の種類と内容です. 注目してほしいのは `HEAD@{2}` の `1f2e3c4` です. これは一度コミットしたものの, その後の `reset` で main の履歴から外れたコミットで, `git log` にはもう現れません. それでも reflog には足あととして残っているので,

```
git reset --hard 1f2e3c4
```

とすれば復元できます. このように reflog は, 他の方法で見失ったコミットまで救い出せる, 最後の安全網です.

reset で履歴を巻き戻し, 戻りすぎたぶんを reflog で救い出す流れを体験してみましょう.

まず, 練習用のコミットをいくつか作ります.

1. `foo.txt` を作り, “line 1” と書いてコミットします.
2. “line 2” を追記してコミットします.
3. “line 3” を追記してコミットします.
4. `git log --oneline` で, 3 つのコミットが積まれたことを確認します.

次に, `reset` で 1 つ前に戻します.

1. 最新の “line 3” を取り消したいので, `git reset --hard HEAD~1` を実行します.
2. `git log --oneline` と `foo.txt` を見て, “line 3” のコミットと行が消えたことを確認します.

最後に, 戻りすぎたぶんを `reflog` で元に戻します.

1. やはり “line 3” が必要でした. `git reflog` を実行し, reset する前の状態 (`HEAD@{1}` の “line 3” のコミット) を探します.
2. `git reset --hard HEAD@{1}` で, その状態に戻します.
3. `git log --oneline` と `foo.txt` で, “line 3” が復活したことを確認します.

3.7.5　Git で悪いことは起きない

最後に覚えておいてほしいのは, Git はセーブポイントを保存しているだけなので, 悪いことは起きない, ということです. 実際にコミットから古いファイルを復元する場面はめったにありません. それでも「いつでも戻れる」という安心感が, 思い切ってコードを書き換える自由を支えてくれます.

第 4 章
高速化

4.1　計算量

アルゴリズムの効率性を評価する尺度として, 計算時間とメモリ使用量があります. ここでいう計算時間とは, アルゴリズムが終了するまでにかかるステップ数 (**時間計算量**, time complexity) のことをいい, 実際の PC 上での実行時間とは異なります. より高いスペックの PC を使えば, 同じアルゴリズムでも実行時間は短くなる一方で, ステップ数の次元が異なるアルゴリズムはマシンの性能に関わらず効率的と言えます. また, メモリ使用量とはアルゴリズムが終了するまでに必要なメモリの量 (**領域計算量**, space complexity) のことをいいます.

4.1.1　時間計算量

時間計算量は, 入力の大きさ n に対するステップ数 $T(n)$ の関数として表されます. 例えば, 配列の要素数が n のときに, すべての要素を 1 回ずつ見るアルゴリズムは, $T(n) = n$ となります. また, 2 重ループで配列のすべての組み合わせを調べるアルゴリズムは, $T(n) = n^2$ となります. このように, アルゴリズムの時間計算量は, 入力の大きさに対するステップ数の増加率によって特徴づけられます.

時間計算量を評価する際, 重要なのは支配的な項の次元数になります. 例えば, $T(n) = 3n^2 + 2n + 1$ の場合, n が大きくなると $3n^2$ の項が支配的になるため, n^2 に着目すれば十分です. この考え方に従ったとき, 計算量を $O(n^2)$ と表記します. より厳密に表現すると以下のようになります.

> **i** *O*-記法
>
> $f(n)$ が $O(g(n))$ とは, 任意の $n \geq 0$ に対して, ある定数 $C > 0$ が存在して, 以下の不等式が成り立つことをいう.
>
> $$|f(n)| \leq C|g(n)|.$$

定量モデルで気にする必要があるのは, ほとんどの場合, グリッド数によるオーダーです. 例えば, $V(k, z)$ のような 2 次元の価値関数をグリッド上で計算する場合, k のグリッド数 n_k と z のグリッド数 n_z に対して, 計算量は $n_k n_z$ の関数となります.

4.1.2　領域計算量

領域計算量は, アルゴリズムが終了するまでに必要なメモリの量を, 入力の大きさ n に対する関数として表します. 例えば, 配列の要素数が n のときに, すべての要素を保存するアルゴリズムは, 領域

計算量が $O(n)$ となります. また, 2 次元配列を保存するアルゴリズムは, 領域計算量が $O(n^2)$ となります.

メモリのヒエラルキー

なぜメモリの使用量が重要なのでしょうか. 近年の PC は何百 GB もの容量があるし, いくら使おうと問題ではないのではないか, と思うかもしれません. しかし, 実際には, メモリのヒエラルキー (memory hierarchy) によって, メモリの速度と容量が大きく異なります.

一般に, 高速なメモリは高価であるため搭載量が少なく, 低速なメモリは安価であるため搭載量が多いです. それらを合わせると, 図 4.1 のようなピラミッド型のヒエラルキーが形成されます.

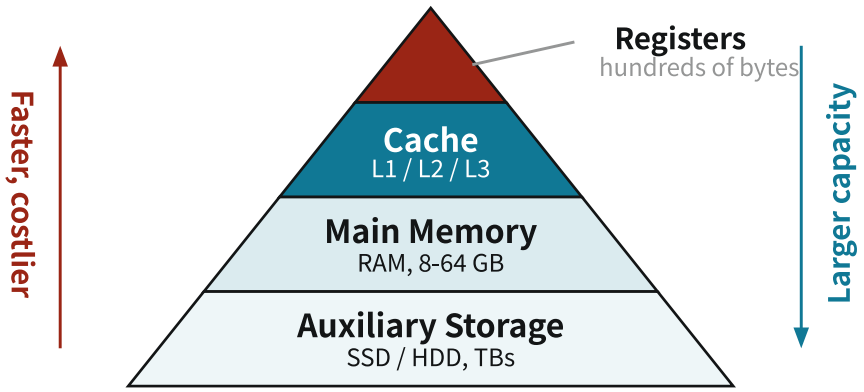


図 4.1:　メモリのヒエラルキー. 上にいくほど高速だが容量は小さい.

- レジスタ (Registers): CPU 内部にある最も高速なメモリ
- キャッシュ (Cache): (近年では) CPU 内部にある高速なメモリ. L1, L2, L3 などのレベルがある. ここまで CPU 内部にある.
- 主記憶 (Main Memory): RAM (Random Access Memory)
- 補助記憶 (Auxiliary Storage): SSD (Solid State Drive) や HDD (Hard Disk Drive)

たとえば, AMD の Ryzen 5 9600 (Zen 5 世代) の場合, L1 キャッシュは 480KB, L2 キャッシュは 6MB, L3 キャッシュは 32MB です. レジスタはサイズで表現することはあまりありませんが, ざっくり数百 B 程度と考えてください. 一方で, 一般にメモリと呼ばれる RAM は 8GB から 64GB 程度が一般的です. さらに, 一般にストレージとよばれる SSD や HDD は数百 GB から数 TB の容量があります.

各層は容量だけでなく速度も桁違いに異なります. CPU がレジスタやキャッシュ上のデータを読むのは数クロックで済む一方, RAM へのアクセスはその数十倍から数百倍かかり, SSD ではさらに桁が変わります. そのため, アルゴリズムが繰り返し参照するデータ (作業セット) がキャッシュに収まるかどうかで, たとえ計算回数が同じでも実行時間は大きく変わります. とくに連続したアドレスを順にたどるアクセスは, CPU が先読みしてまとめてキャッシュに載せられるため高速で, 逆に飛び飛びのアクセスはキャッシュミスを頻発させ, そのたびに低速な RAM を待つことになります. この「メモリ上での近さ」を局所性 (locality) と呼び, 後のベクトル化が速い理由の一つも, データが連続したメモリに並んでこの局所性を活かせる点にあります.

4.2 計算量の実践

実際の推定を例に, 計算量を測定してみましょう. ここでは処置効果の推定で広く使われる**最近傍マッチング** (nearest-neighbor matching) を考えます. 処置群の各個体について, 共変量 x (傾向スコアなど) が最も近い対照群の個体を 1 人見つけ, その個体を対応させることで ATT を推定する方法です.

最近傍マッチング

処置群を n_t 人, 対照群を n_c 人とします. ナイーブに実装すると, 処置群の各個体について対照群の全員との距離を計算し, 最も近い個体を探すことになります. コードで表すと次の `match_nn` 関数のようになります.

```
match_nn ← function(n_t, n_c, seed = 42L) {
  set.seed(seed)
  x_t ← rnorm(n_t) # covariate of treated units
  x_c ← rnorm(n_c) # covariate of control units

  matched ← integer(n_t)
  for (i in 1:n_t) {
    best_dist ← Inf
    best_j ← 0L
    for (j in 1:n_c) {
      d ← abs(x_t[i] - x_c[j])

      if (d < best_dist) {
        best_dist ← d
        best_j ← j
      }
    }

    matched[i] ← best_j
  }

  matched
}
```

時間計算量の数え方として, 四則演算, 比較 (`<` など), 代入, 配列へのアクセスは $O(1)$ とします. すると, 計算回数の主要部は二重ループ, つまり $O(n_t n_c)$ です. 以下では簡単のため $n_t = n_c = n$ とするので, 時間計算量は $O(n^2)$ になります. 一方で保存しているのはマッチング結果 `matched` (長さ n_t) と共変量ベクトルだけなので, 領域計算量は $O(n)$ です. では, n を変化させたときの計算時間を測定してみましょう.

bench パッケージ

R では, bench パッケージを使うと実際の計算時間を測定できます.[10]

```
bm ← bench::mark(
  "n = 100" = match_nn(100L, 100L),
  "n = 1000" = match_nn(1000L, 1000L),
  check = FALSE,
  filter_gc = FALSE
)
```

表 4.1: 観測数を変えたときのマッチングの計算時間とメモリ使用量

n	中央値 (ms)	メモリ
100	0.9	2.1 KB
1000	85.8	19.7 KB

n を 100 から 1000 へ 10 倍にすると, 計算時間は約 100 倍になる一方で, メモリ使用量は約 9 倍にしかなっていないことがわかります. これは, 計算時間が $O(n^2)$, 領域計算量が $O(n)$ であることと一致しています.

4.3　高速化

4.3.1　プログラミング言語の選択

プログラミング言語には, 明確に計算速度の壁があります. 経済学の分野で使われている言語だと, 次のような関係にあります.

$$\text{C/C++, Fortran, Julia} \gg \text{Python, Matlab} > \text{R}$$

[10] system.time() でも簡易的に測定できますが, 1 回の実行時間しか測れず, OS のスケジューリングなどの影響でばらつきます. bench::mark() は実行時間に応じて試行回数を自動で調整し, 実行時間の分布とメモリ割り当て量を返してくれるため, 関数の実行速度を正確に測定するのに適しています.

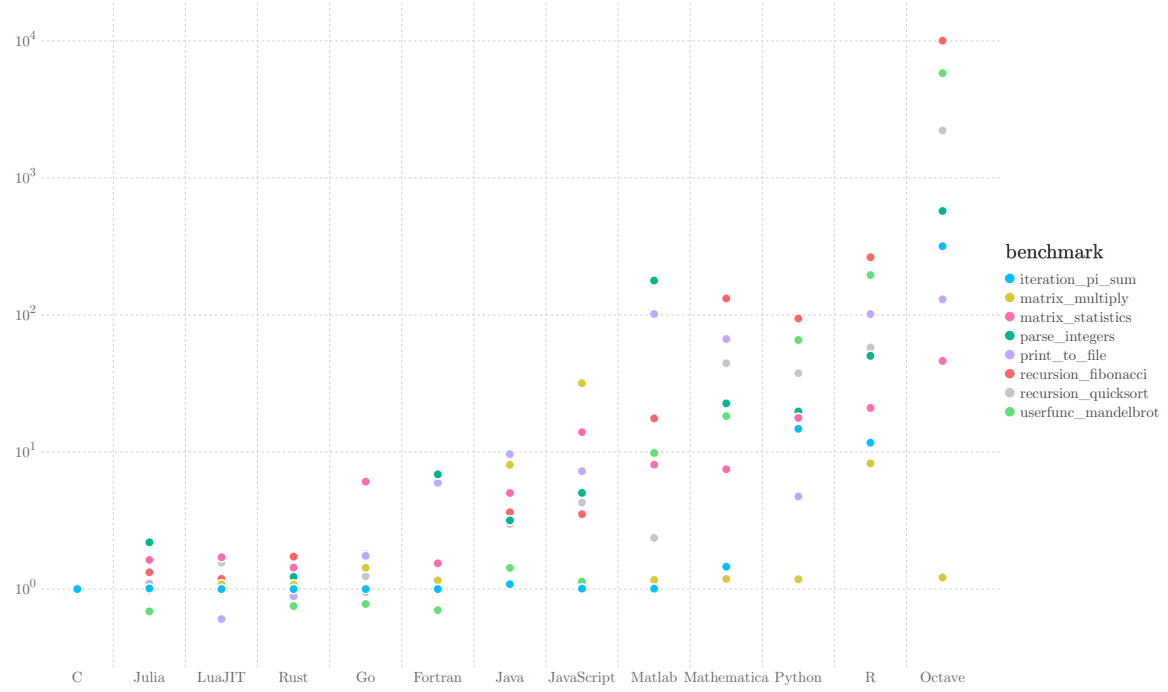


図 4.2: Julia Micro Benchmarks

おおむね, C/C++, Fortran, Julia は 10-100 倍ほど Python, Matlab, R より速く計算が可能です. なおベクトル化というテクニックや Python の Numba を用いることで, Python, Matlab, R でも同様の速度も出すことが可能ですが, C/C++, Fortran, Julia などの言語とはそもそも質的に異なるという事実は頭に入れておいた方が良いでしょう.

4.3.2 ベクトル化

Python, Matlab, R などの言語ではベクトル化することで計算速度が改善されます. しかし, これはセクション 4.1 から考えると矛盾しています. ベクトル化そのものは計算回数は変えませんし, ベクトル化することで確保されるメモリの量は増えるはずです. この矛盾はどこからくるのでしょう.

これは主には型推定の問題です. Python などのスクリプト言語は変数の型を実行時に確定させるため,[11] for ループの中で型推定を行う必要があります. ベクトル化とは, 型が定まったベクトル同士の計算を行うという性質上, この型推定にかかる時間を削減しています.

先ほどの `match_nn` を題材に, ベクトル化で実際に速くなるかを確かめてみましょう. `match_nn` のボトルネックは, 処置群の各個体について対照群すべてとの距離を 1 つずつ計算する内側ループでした. 共変量の生成は計測から外したいので, 以下では共変量ベクトル `x_t`, `x_c` を引数に取る 3 通りの実装を比較します.

まずは, 内側も外側も `for` で回す二重ループ版です.

```
match_nn_loop ← function(x_t, x_c) {
  matched ← integer(length(x_t))
```

[11] Julia では関数を定義する際に型を推定するため, 関数を実行時にはすでに型が確定しているため, スクリプト言語ではあるものの計算速度は C/C++や Fortran に匹敵するほど速いです.

```
  for (i in seq_along(x_t)) {
    best_dist ← Inf
    best_j ← 0L
    for (j in seq_along(x_c)) {
      d ← abs(x_t[i] - x_c[j])

      if (d < best_dist) {
        best_dist ← d
        best_j ← j
      }
    }

    matched[i] ← best_j
  }

  matched
}
```

次に，内側ループだけをベクトル化します．対照群すべてとの距離 `abs(xi - x_c)` を一度に計算し，`which.min()` で最も近い個体を選びます．処置群についての外側ループは，`for` の代わりに `purrr::map_int()` を使うと簡潔に書けます．[12]

```
match_nn_map ← function(x_t, x_c) {
  purrr::map_int(x_t, \(xi) which.min(abs(xi - x_c)))
}
```

最後に，外側ループも消した完全ベクトル化版です．`outer()` で全ペアの距離を $n_t \times n_c$ の行列に一度に計算し，各行の最小値の位置を `max.col()` で取り出します．

```
match_nn_outer ← function(x_t, x_c) {
  d ← abs(outer(x_t, x_c, "-"))
  max.col(-d, ties.method = "first")
}
```

3 通りの実装を比較してみましょう．`bench::mark()` はデフォルトで返り値の一致を確認するので，3 つが同じ結果を返すことも同時に検証されます．

```
n ← 1000L
x_t ← rnorm(n) # covariate of treated units
x_c ← rnorm(n) # covariate of control units
```

[12] `purrr::map_int()` は各要素に関数を適用して整数ベクトルを返す関数で，`for` ループとほぼ同じ速度です．後の並列化でこの形がそのまま活きてきます．

```
mb_vec ← bench::mark(
  loop = match_nn_loop(x_t, x_c),
  map = match_nn_map(x_t, x_c),
  outer = match_nn_outer(x_t, x_c),
  filter_gc = FALSE
)
```

表 4.2: match_nn の実装による実行時間とメモリ割り当て ($n = 1000$)

手法	中央値 (ms)	メモリ割り当て
二重ループ	88.5	54.9 KB
内側ベクトル化 (purrr::map)	3.7	8.4 MB
完全ベクトル化	4.3	30.6 MB

内側ループをベクトル化するだけで, 二重ループの約 24 倍高速になりました. `abs(xi - x_c)` のように型が定まったベクトル同士の演算にまとめることで, 対照群 1 人ずつに対して行っていた型推定のオーバーヘッドを取り除けるためです. 外側ループを `purrr::map_int()` に変えても速度はほとんど変わりません. 速くなったのはあくまで内側のベクトル化によるものです.

一方で, 完全ベクトル化はさらに速くなるわけではありません. むしろ 表 4.2 のとおり, メモリ割り当てが桁違いに大きくなっています. これは `outer()` が $n \times n$ の距離行列 ($8n^2$ バイト) を一度に確保するためで, セクション 4.1 でみた領域計算量そのものです. 内側ベクトル化も延べの割り当て量は大きいですが, 同時に保持するのは長さ n の距離ベクトル 1 本だけ ($O(n)$) なのに対し, 完全ベクトル化は行列全体を保持し続ける ($O(n^2)$) ため, n が大きいと RAM に収まらなくなります.

4.3.3 Rcpp

Rcpp は, R から C++のコードを呼び出すためのパッケージです. C++で書いた関数を R から呼び出すことができるため, 計算速度を大幅に改善することができます. 例えば, 先ほどの `match_nn` 関数のボトルネックである二重ループを, C++で書き直すと次のようになります.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector match_nn_cpp(NumericVector x_t, NumericVector x_c) {
  int n_t = x_t.size();
  int n_c = x_c.size();
  IntegerVector matched(n_t);

  for (int i = 0; i < n_t; i++) {
    double best_dist = R_PosInf;
    int best_j = 0;
```

```
    for (int j = 0; j < n_c; j++) {
      double d = std::abs(x_t[i] - x_c[j]);

      if (d < best_dist) {
        best_dist = d;
        best_j = j + 1;  // 1-based index for R
      }
    }
    matched[i] = best_j;
  }

  return matched;
}
```

`// [[Rcpp::export]]` を付けた関数は, コンパイル後に R から呼び出せるようになります. 比較対象の R 実装には, ベクトル化の節で定義した二重ループ版 `match_nn_loop` をそのまま使います.

では, R 版と Rcpp 版で実行時間を比較してみましょう. ここでは共変量の生成 (`rnorm`) を計測の外で行い, 二重ループそのものだけを測ります.[13] `bench::mark()` はデフォルトで両者の返り値が一致するかも確認するので, 書き直しても結果が変わっていないことを同時に検証できます.

```
bm_cpp ← bench::press(
  n = c(100L, 1000L),
  {
    x_t ← rnorm(n) # covariate of treated units
    x_c ← rnorm(n) # covariate of control units
    bench::mark(
      R = match_nn_loop(x_t, x_c),
      Rcpp = match_nn_cpp(x_t, x_c),
      filter_gc = FALSE
    )
  }
)
```

表 4.3: R と Rcpp によるマッチングの計算時間と領域計算量

	R		Rcpp	
n	中央値 (ms)	メモリ	中央値 (ms)	メモリ
100	0.86	2.1 KB	0.0069	2.1 KB
1000	85.31	19.7 KB	0.6117	19.7 KB

[13] `rnorm` によるデータ生成は $O(n)$ なので, 二重ループ ($O(n^2)$) と一緒に計測すると, 特に n が小さいときに純粋なアルゴリズムの比較になりません. `bench::press()` はパラメータごとにブロックを評価し, 内側の `bench::mark()` の式だけを計測するため, 生成部分を計測から除外できます.

同じ $O(n^2)$ のアルゴリズムでも, C++で書き直すことで $n = 100$ で約 125 倍, $n = 1000$ で約 139 倍高速になりました. 速度向上の倍率が n によらずほぼ一定であることに注目してください. これは, Rcpp が計算量のオーダー ($O(n^2)$) を変えているわけではなく, 1 ステップあたりの定数項を小さくしているためです. セクション 4.1 のベクトル化と同様に, R が各演算のたびに行う型推定などのオーバーヘッドを取り除いているのです. オーダーが同じである以上, n を 10 倍にすれば計算時間はやはり約 100 倍になります. アルゴリズム自体の計算量を下げられないボトルネックに対しては, このように Rcpp で定数項を削るのが有効な高速化の手段になります.

一方で領域計算量に注目すると, R でも Rcpp でも保存するのはマッチング結果と共変量だけなので $O(n)$ のまま変わりません. 完全ベクトル化が距離行列で $O(n^2)$ のメモリを消費したのとは対照的に, Rcpp は自然な二重ループのまま, メモリを増やさずに速度だけを引き上げられるのです.

4.4 並列化

現代の CPU は複数のコアを持っており, 独立な計算を同時に走らせることで実時間を短縮できます. これを並列化 (parallelization) といいます. `match_nn` の外側ループ, つまり処置群の各個体について最も近い対照群を探す処理は, 個体ごとに独立しています. このように各反復が互いに依存しない計算は並列化と相性がよく, embarrassingly parallel と呼ばれます.

4.4.1 コア数の確認

まず, 自分の PC が何コア使えるかを確認します. ここでは `future::availableCores()` を使います.

```
future::availableCores()
## system
##     16
```

コア数を数える関数には, もう一つ `parallel::detectCores()` があります. 名前は似ていますが, 答えている問いが違います. `detectCores()` が返すのは「このマシンに論理コアがいくつ載っているか」で, `availableCores()` が返すのは「そのうち何個を自分が使ってよいか」です. 手元の PC で普通に作業しているうちは両者は一致しますが, コンテナや HPC のジョブスケジューラで CPU が制限されている環境では食い違います. 例えば 32 コアのマシンで 4 コアだけ割り当てられたジョブの中では, `detectCores()` は制限を無視して 32 を返す一方, `availableCores()` は割り当てを読み取って 4 を返します. 前者を信じて 32 ワーカーを立てると, 4 コアを 32 プロセスで奪い合うことになり, かえって遅くなります. `future::plan()` が既定で使うワーカー数も `availableCores()` の値なので, 並列化の場面ではこちらを見ます.

コア数には物理コアと論理コアの 2 種類がある点にも注意しましょう. 物理コアは実際に搭載された演算ユニットの数で, 論理コアは OS から見かけ上使えるコアの数です. Intel や AMD の多くの CPU は, ハイパースレッディング (SMT, simultaneous multithreading) によって 1 つの物理コアを 2 つの論理コアに見せるため, 論理コア数は物理コア数の 2 倍になります. ただし論理コアは 1 つの物理コアを分け合っているだけなので, CPU をめいっぱい使う計算では, 論理コアの数だけワーカーを立てても物理コアの数ぶんほどの速度向上は得られないことが多いです. 一方, Apple Silicon の Mac は SMT を採用していないため物理コアと論理コアが一致し, どちらを数えても同じ値になります.

以下では `future` と `furrr` を使って並列化します. `furrr::future_map()` は `purrr::map()` の並列版で, ほぼ同じ書き方のまま処理を複数コアに分散できます.

4.4.2　並列化とベンチマーク

逐次版には, ベクトル化の節で定義した `match_nn_map` をそのまま使います. 並列版は, `purrr::map_int()` を `furrr::future_map_int()` に置き換えるだけです.

```
library(future)
library(furrr)

match_nn_future ← function(x_t, x_c) {
  furrr::future_map_int(x_t, \(xi) which.min(abs(xi - x_c)))
}
```

並列化のバックエンドは `future::plan()` で指定します. `multisession` は複数の R プロセスを立ち上げ, それぞれにタスクを割り当てます. ここでは 4 コアを使い, $n = 1000$ と $n = 30000$ で逐次版と比較します.

```
plan(multisession, workers = 4)

bm_par ← bench::press(
  n = c(1000L, 30000L),
  {
    x_t ← rnorm(n) # covariate of treated units
    x_c ← rnorm(n) # covariate of control units
    bench::mark(
      serial = match_nn_map(x_t, x_c),
      parallel = match_nn_future(x_t, x_c),
      filter_gc = FALSE
    )
  }
)

plan(sequential)
```

表 4.4: 逐次処理と 4 コア並列処理の計算時間

n	逐次 (ms)	並列 (ms)	速度向上 (倍)
1000	4	13	0.29
30000	3228	873	3.7

$n = 1000$ では, 並列版の方がむしろ約 3 倍遅くなっています. 各コアにデータとタスクを送り, 結果を集める通信のオーバーヘッドが, 計算そのものより大きいためです. 一方 $n = 30000$ では, 並列版が約 3.7 倍速くなりました.

ここで重要なのは 2 点です. 第一に, 並列化はタダではありません. タスクの分配と結果の集約にオーバーヘッドがかかるため, 1 つ 1 つの計算が軽い場合や問題が小さい場合には, かえって遅くなります. 第二に, 速度向上のおおよその目安はコア数 (ここでは 4) で, データの分配や集約のオーバーヘッドのぶん前後します. それでも, 各反復が独立で計算が十分重い場合には, 並列化はコア数に近い高速化をもたらす有効な手段です.

4.4.3 C++コードの並列化

Rcpp で書いたコードも並列化できます. 先ほどの `furrr` は独立した R プロセスを複数立ち上げる方式でしたが, ここでは `RcppParallel` を使い, 1 つのプロセス内で軽量なスレッドを並列に走らせます.[14]

並列化したい処理を `Worker` を継承した構造体の `operator()` に書き, `parallelFor()` で範囲を分割して各スレッドに割り当てます. 中身は逐次版 `match_nn_cpp` の二重ループと同じで, 処置群についての外側ループを `begin` から `end` までに区切って担当します.

```
#include <Rcpp.h>
#include <RcppParallel.h>
using namespace Rcpp;
using namespace RcppParallel;

// [[Rcpp::depends(RcppParallel)]]

struct MatchWorker : public Worker {
  const RVector<double> x_t;
  const RVector<double> x_c;
  RVector<int> matched;

  MatchWorker(const NumericVector x_t, const NumericVector x_c, IntegerVector matched)
    : x_t(x_t), x_c(x_c), matched(matched) {}

  void operator()(std::size_t begin, std::size_t end) {
    std::size_t n_c = x_c.length();
    for (std::size_t i = begin; i < end; i++) {
      double best_dist = R_PosInf;
      int best_j = 0;
      for (std::size_t j = 0; j < n_c; j++) {
        double d = std::abs(x_t[i] - x_c[j]);

        if (d < best_dist) {
```

[14] macOS の標準コンパイラ (Apple clang) は OpenMP を標準では使えないため, ここでは移植性の高い `RcppParallel` を使います. なお, スレッドはメモリを共有するため, ワーカー内では R のオブジェクトを直接触らず, スレッドセーフな `RVector` / `RMatrix` 経由でアクセスする必要があります.

```
          best_dist = d;
          best_j = j + 1;  // 1-based index for R
        }
      }
      matched[i] = best_j;
    }
  }
};

// [[Rcpp::export]]
IntegerVector match_nn_cpp_par(NumericVector x_t, NumericVector x_c) {
  IntegerVector matched(x_t.size());
  MatchWorker worker(x_t, x_c, matched);
  parallelFor(0, x_t.size(), worker);
  return matched;
}
```

`RcppParallel::setThreadOptions()` でスレッド数を指定できます (既定では利用可能なすべてのコアを使います). `furrr` と揃えて 4 スレッドにし, 逐次版 `match_nn_cpp` と比較します.

```
library(RcppParallel)
setThreadOptions(numThreads = 4)

bm_cpp_par ← bench::press(
  n = c(1000L, 30000L),
  {
    x_t ← rnorm(n) # covariate of treated units
    x_c ← rnorm(n) # covariate of control units
    bench::mark(
      serial = match_nn_cpp(x_t, x_c),
      parallel = match_nn_cpp_par(x_t, x_c),
      filter_gc = FALSE
    )
  }
)
```

表 4.5: 逐次 Rcpp と 4 スレッド並列 Rcpp の計算時間

n	逐次 (ms)	並列 (ms)	速度向上 (倍)
1000	0.6	0.2	3.2
30000	547.3	123	4.4

同じ 4 コアでも, `furrr` が $n = 1000$ で逆に遅くなったのとは対照的に, `RcppParallel` は $n = 1000$ でも約 3.2 倍, $n = 30000$ で約 4.4 倍速くなりました. スレッドは同じプロセス内でメモリを共有するため, プロセスの起動やデータのコピーがなく, オーバーヘッドがずっと小さいからです. このように, Rcpp で定数項を削り (オーダーは $O(n^2)$ のまま), さらに並列化でコア数ぶん速くする, という 2 つの高速化を重ねがけできます.

なお, すべての計算が並列化できるわけではありません. 例えば後ろ向き帰納法で解く動的計画問題のように, 次の反復が前の反復の結果に依存する計算は, その方向には並列化できません.

第 5 章
大規模データ

5.1　大規模データとメモリ

`dplyr` や `data.frame` のような R の標準的な道具は, データ全体をメモリ (RAM) に読み込んでから処理します. セクション 4.1 のメモリのヒエラルキーで見たように, RAM の容量には限りがあります. データが RAM に収まらなくなると, 処理は途端に遅くなり, やがて止まってしまいます.

しかし, よく考えると, 分析のたびにデータ全体が必要になることはめったにありません. たいていのクエリは, 一部の列と一部の行しか使わないからです. そこで鍵になるのが, データ全体を RAM に載せるのではなく **必要な分だけ読む** という発想です. クエリに必要な列・行だけを読み込んで処理すれば, RAM を超えるデータも扱えます.

これを実現するには, 役割の違う 2 つの道具を組み合わせます. データの **保存形式** と, それを処理する **エンジン** です. この関係を整理したのが 図 5.1 です.

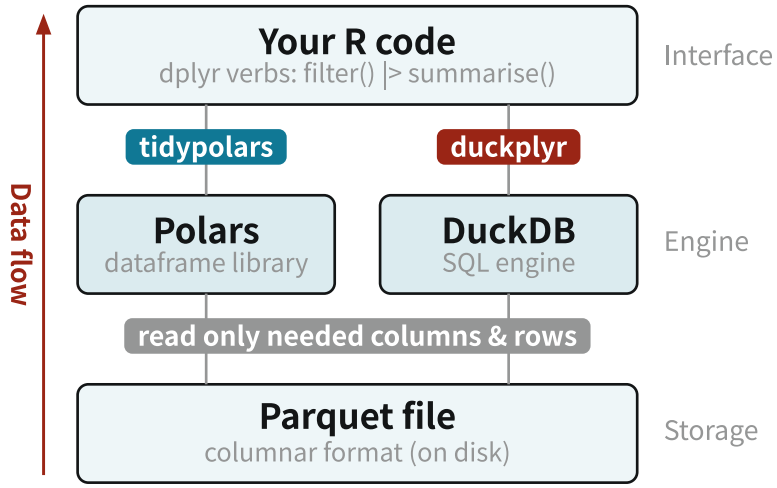


図 5.1:　Parquet・Polars・DuckDB の関係

- Parquet は保存形式. データをディスクにどう並べるかを決めるだけで, それ自体は計算しません.
- Polars と DuckDB は処理エンジン. ディスク上のデータを読み込み, 絞り込みや集計を実行します. 役割はほぼ同じで, 互いに置き換えられる選択肢です. Polars は Rust 製のデータフレームライブラリ, DuckDB は分析に特化した組み込み型のデータベースです.
- R からの書き方は共通です. どちらのエンジンも, dplyr とほぼ同じ構文で操作できます (Polars は `tidypolars`, DuckDB は `duckplyr` を通します).

エンジンが「必要な分だけ読む」には，それを許す保存形式が要ります．列指向フォーマット (Parquet) は，まさにそのための基礎技術です．データを Parquet で保存し，それを Polars か DuckDB で読んで処理する，というのが基本の形になります．

5.1.1　NYC タクシーデータ

データとしてニューヨーク市タクシー・リムジン委員会 (TLC) が公開している タクシー乗車記録 を使います．乗車 1 回ごとに乗車・降車時刻，距離，料金，チップ額などが記録された，取引レベルの実データです．ここでは 2024 年 1 月のイエロータクシー分 (2,964,624 行，列) を使います．[15]

このノートは，データが `data/` 以下に置かれている前提で進めます．ダウンロードは `targets` パイプラインに組み込んであるので，手元に無ければ次で取得できます．[16]

```
Rscript -e 'targets::tar_make(taxi_parquet)'
```

ファイルのパスは，プロジェクトのどこから実行しても解決できるよう `here` で組み立てます．

```
path_pq ← here::here("data", "yellow_tripdata_2024-01.parquet")
```

5.2　列指向フォーマット

大規模データを「必要な分だけ読む」ための鍵が，この **列指向フォーマット** です．CSV は **行指向** のテキストファイルで，1 行ずつ順に値がカンマ区切りで並びます．そのため，一部の列だけが欲しいときでも，結局すべての行・すべての列を読む必要があります．

Parquet は **列指向** のバイナリフォーマットです．同じテーブルでも，CSV が 1 行ぶんをまとめて並べるのに対し，Parquet は 1 列ぶんをまとめて並べます．この違いは，一部の列だけを読むときに大きく効いてきます (図 5.2)．

[15] このデータは元から Parquet 形式で配布されており，1 か月分でも約 48 MB あります．全期間 (2009 年〜) を合わせると数十 GB・十数億行になり，RAM には到底収まりません．まさに大規模データの練習にうってつけです．

[16] `data/` は `.gitignore` の対象です．`taxi_parquet` ターゲットが TLC のサーバーから Parquet を取得し，`data/yellow_tripdata_2024-01.parquet` に保存します．

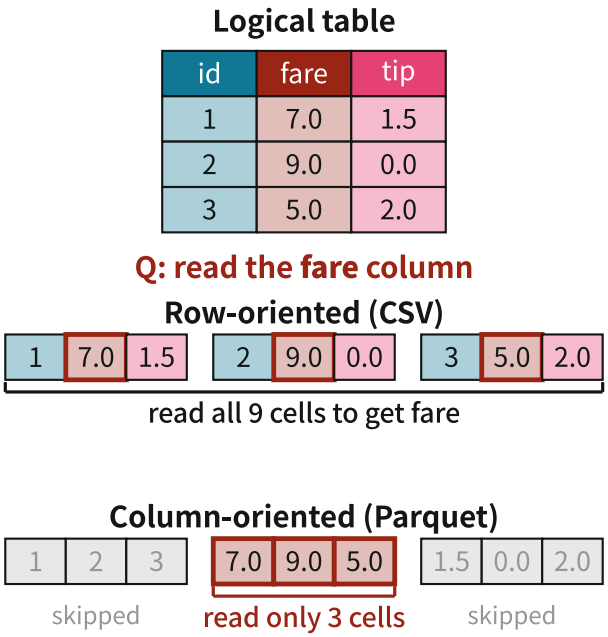


図 5.2: 行指向と列指向

同じ列の値が連続して並ぶことから, 次のような利点が生まれます.

- **必要な列だけ読める**: 19 列のうち 3 列しか使わないなら, その 3 列だけをディスクから読めます (projection pushdown).
- **行グループをスキップできる**: ファイルは行グループ (row group) に分かれており, 各グループの統計量 (最小値・最大値など) がメタデータに記録されています. 条件に合わない行グループは丸ごと読み飛ばせます (predicate pushdown).
- **小さい**: 列ごとに型がそろっているため圧縮が効きます.
- **型が保たれる**: CSV と違い, 数値・日付・文字列などの型情報がファイルに含まれます.

実際にどのくらい小さいのか, 同じデータを CSV に書き出して比べてみましょう. ここでは変換に DuckDB を使います.

```
path_csv ← file.path(tempdir(), "taxi.csv")
con ← dbConnect(duckdb::duckdb())
dbExecute(
  con,
  sprintf(
    "COPY (SELECT * FROM read_parquet('%s')) TO '%s' (FORMAT CSV, HEADER)",
    path_pq,
    path_csv
  )
)
## [1] 2964624
dbDisconnect(con)
```

表 5.1: CSV と Parquet のファイルサイズ

形式	ファイルサイズ
CSV	285.4 MB
Parquet	47.6 MB

Parquet は CSV のおよそ 6 分の 1 のサイズに収まっています. R からだと `nanoparquet` (軽量) や `arrow` で Parquet を読み書きできます. 分析用のデータは, できるだけ Parquet で保存しておくのがよいでしょう.

5.2.1　Arrow との関係

Parquet とよく一緒に名前が挙がる Arrow も列指向ですが, 担う層が違います. Parquet がディスク上の保存形式なのに対し, Arrow はメモリ (RAM) 上 の列指向フォーマットで, ディスクの Parquet を読み込むとメモリ上では Arrow の形になる, という対の関係です (図 5.3).

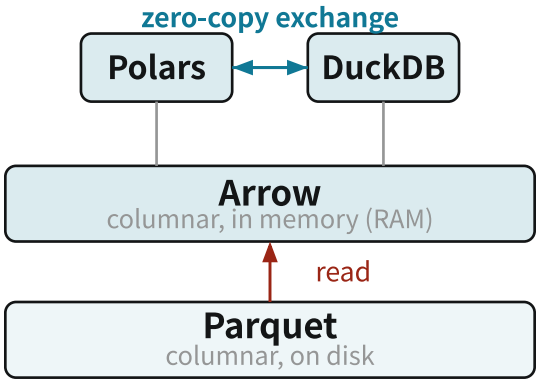


図 5.3: Parquet と Arrow

Arrow は標準的な形式になっているので, Arrow を使うツール同士 (Polars, DuckDB, さらに Python のツールなど) は, データをコピーせずに (zero-copy) 受け渡せます. 逆に, R のデータフレームは Arrow 形式ではないため, 変換のため `collect()` / `as_tibble()` などのステップが必要になります.[17]

i 他の列指向フォーマット

Parquet はデファクトスタンダードですが, 他にも列指向フォーマットはあります.

- **ORC**: Hadoop / Hive / Spark 系でよく使われる, Parquet と似たディスク列指向形式.
- **Feather** (Arrow IPC): Arrow のメモリ上の形をほぼそのままディスクに書いた形式. 圧縮をほとんどかけず読み書きが非常に速い反面ファイルは大きく, 長期保存より一時ファイルや R ↔ Python の高速な受け渡しに向きます (小さく圧縮して保存・大規模クエリ向けに最適化する Parquet とは設計の方向が逆です).
- クラウドのデータウェアハウス (BigQuery, Redshift, Snowflake など) も, 内部は列指向です.

[17] ややこしいことに R には `arrow` というパッケージもありますが, これも Arrow 形式を扱うため, Polars や DuckDB と同様に Parquet を遅延読み込みして集計するエンジンとしても使えます.

5.3 Polars

Polars は Rust で書かれた高速なデータフレームライブラリです. マルチスレッドで動き (セクション 4.1 の並列化を思い出してください), **遅延評価** (lazy evaluation) によってクエリを最適化します.

> **i** tidypolars のインストール
>
> ここでは Polars を `tidypolars` (dplyr 構文) 経由で使います. tidypolars は polars に依存しますが, どちらも CRAN では配布されていない (polars は 2024 年に削除) ため, R-multiverse からまとめてインストールします.
>
> ```
> install.packages(c("polars", "tidypolars"), repos = "https://community.r-multiverse.org")
> ```

ここでは `tidypolars` を通して, **dplyr とほぼ同じ構文** で Polars を使います.[18] 例として, 料金が正の乗車について, 支払い方法 (`payment_type`) ごとの平均チップ額を求めてみます.[19]

```
q ← scan_parquet_polars(path_pq) ▷
  filter(fare_amount > 0) ▷
  summarise(tip = mean(tip_amount), .by = payment_type)

q ▷
  arrange(payment_type) ▷
  as_tibble()
## # A tibble: 5 × 2
##   payment_type      tip
##          <dbl>    <dbl>
## 1            0 1.57
## 2            1 4.17
## 3            2 0.000146
## 4            3 0.00539
## 5            4 0.00523
```

`scan_parquet_polars()` はファイルを開くだけで, データはまだ読み込みません. **遅延評価** (lazy evaluation) とは, コマンドを書いた時点ではすぐに計算せず, 「何をするか」という計画 (プラン) だけを組み立てておく方式です. `as_tibble()` (や `collect()`) を呼んで初めて, 最適化されたプランが実行されます.

その最適化された計画は `explain()` で確認できます.

---

[18] Polars には Python 版に合わせた独自の構文 (`pl$col( ... )$agg( ... )`) もあります. Python も使う人にはそちらが便利でしょう.

[19] `payment_type` は支払い方法のコードで, 1 がクレジットカード, 2 が現金です. カード払いではチップが記録される一方, 現金のチップは記録されないことが多い, という現実が結果に表れます.

```
cat(explain(q))
## AGGREGATE[maintain_order: false]
##                          [when([(col("tip_amount").null_count().cast(Float64))        >
(0.0)]).then(null.strict_cast(Float64)).otherwise(col("tip_amount").mean()).alias("tip")]
BY [col("payment_type")]
##   FROM
##   simple π 2/2 ["payment_type", "tip_amount"]
##                  Parquet    SCAN    [/Users/kazuharu/github/workshop-graduate-2026/data/
yellow_tripdata_2024-01.parquet]
##     PROJECT 3/19 COLUMNS
##     SELECTION: [(col("fare_amount")) > (0.0)]
##     ESTIMATED ROWS: 2964624
```

`PROJECT 3/19 COLUMNS` は, 19 列のうち必要な 3 列 (`payment_type`, `fare_amount`, `tip_amount`) だけを読むこと (列の刈り込み) を表します. `SELECTION: [(col("fare_amount")) > 0.0]` は, `filter` の条件が Parquet の読み込み段階まで押し下げられていること (述語の押し下げ) を表します. このように Polars は, こちらが書いた順序にとらわれず, 「必要なデータだけを最小限読む」ように勝手に並べ替えてくれます.

5.4 DuckDB

DuckDB は, 分析用途に特化した組み込み型のデータベースです. SQLite が「手軽なトランザクション用 DB」なら, DuckDB は「手軽な分析用 DB」だと考えるとよいでしょう. マルチスレッドで動き, RAM に収まらないデータ (larger-than-memory) も扱えます. 先ほど CSV への変換に使ったのも, この DuckDB です.

R からは `duckplyr` を使うのが手軽です. これは **dplyr のドロップイン置換** で, 既存の dplyr コードをほぼそのまま, バックエンドだけ DuckDB に差し替えて高速化できます.[20]

```
read_parquet_duckdb(path_pq) ▷
  filter(fare_amount > 0) ▷
  summarise(tip = mean(tip_amount), .by = payment_type) ▷
  arrange(payment_type) ▷
  collect()
## # A tibble: 5 × 2
##   payment_type      tip
## *        <dbl>    <dbl>
## 1            0 1.57
```

[20] `duckplyr` は 2025 年に tidyverse の一員となり, CRAN で配布されています (`install.packages("duckplyr")`). Polars と違い, 追加のリポジトリ設定なしに入ります.

```
## 2           1 4.17
## 3           2 0.000146
## 4           3 0.00539
## 5           4 0.00523
```

`read_parquet_duckdb()` で遅延的に Parquet を開き, dplyr の動詞を並べ, `collect()` で結果を取り出します. Polars (tidypolars) と書き方がほとんど同じであることに注目してください. どちらも内部では遅延評価とクエリ最適化を行っています.

5.5　ベンチマーク

では, 同じ集計を `dplyr`, `tidypolars`, `duckplyr` の 3 通りで実行し, 速度とメモリを比べてみましょう. `dplyr` 版は, Parquet をいったん全部メモリに読み込んでから処理します.

```
bm ← bench::mark(
  dplyr = nanoparquet::read_parquet(path_pq) ▷
    filter(fare_amount > 0) ▷
    summarise(tip = mean(tip_amount), .by = payment_type) ▷
    arrange(payment_type),
  tidypolars = scan_parquet_polars(path_pq) ▷
    filter(fare_amount > 0) ▷
    summarise(tip = mean(tip_amount), .by = payment_type) ▷
    arrange(payment_type) ▷
    as_tibble(),
  duckplyr = read_parquet_duckdb(path_pq) ▷
    filter(fare_amount > 0) ▷
    summarise(tip = mean(tip_amount), .by = payment_type) ▷
    arrange(payment_type) ▷
    collect(),
  check = FALSE,
  iterations = 5,
  filter_gc = FALSE
)
```

表 5.2: 集計の計算時間とメモリ

手法	中央値 (ms)	メモリ割り当て
dplyr	401	1.3 GB
tidypolars	42	421.1 KB
duckplyr	21	85.9 KB

集計そのものも `tidypolars` と `duckplyr` の方が `dplyr` より約 10 倍速いですが, より目を引くのは **R が確保するメモリ** の差です. 表 5.2 のとおり, `dplyr` が数百 MB を確保するのに対し, `tidypolars` と `duckplyr` のそれは桁違いに小さくなっています.

理由は 2 つあります. 第一に, 遅延評価と Parquet の組み合わせにより, 「必要なのは `payment_type`, `fare_amount`, `tip_amount` の列と `fare_amount > 0` の行だけ」と見抜いて, その分しか読み込みません (セクション 5.3 の `explain()` で見た列の刈り込みと述語の押し下げです). 第二に, Polars と DuckDB はデータを R のメモリではなく自前のメモリ (Rust や C++ 側) に持ち, R へは最終的な集計結果だけを渡します. そのため R のヒープにはほとんど何も積まれません. 一方 `dplyr` は, 19 列すべてを R に読み込み, 中間結果まで含めて R オブジェクトとして抱えます.

この違いは, データが RAM を超えるようになると「動くか動かないか」を分けます. DuckDB は必要に応じて中間結果をディスクに退避できるため, RAM に収まらないデータも処理できます.

5.5.1　どれを使えばよいか

実証家として皆さんに覚えておいて欲しいのは次の三つです.

1. Parquet で保存する
2. データ処理のエンジンとしては, `duckplyr` を使う

特に理由がなければ, parquet 形式でデータを保存しておくのが高速化かつ軽量という点でおすすめです. Polars は Python で扱う分にはとても便利ですが, CRAN から外されたりなど R でのサポートが不安定であるため, `duckplyr` を使うのが便利かなと思っています.

なお, R のオブジェクトをそのまま高速に保存したいだけなら, `fst` や `qs2` も便利です. これらは `saveRDS()` の高速な代替として使えます.

第 6 章

API

6.1 Client-Server Model

私たちが普段見ている Web ページは，クライアントとサーバーのやり取りで成り立っています (図 6.1).

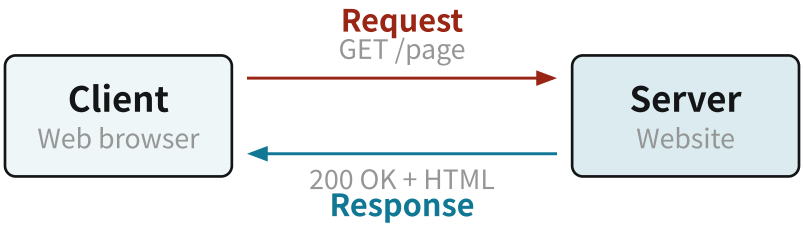


図 6.1: クライアントとサーバー

- **クライアント** (client) は Web ブラウザやスマホのアプリなど，ユーザーが直接操作する側のソフトウェアです
- **サーバー** (server) は Web サイトを運営する側のコンピュータで，クライアントからのリクエストに応じてデータを返す役割を担います

このやり取りは HTTP (Hypertext Transfer Protocol) という共通のルールに従います．中身は大きく 2 つに分かれます．

- **リクエスト** (request): 「どの URL の何が欲しいか」をサーバーに伝えます．例えば `GET /page` は「このページをください」という意味です (`GET` は取得，`POST` は送信を表す HTTP メソッドです).
- **レスポンス** (response): サーバーからの返事です．成否を表すステータスコード (`200 OK`, `404 Not Found` など) と，本文 (body) から成ります．本文は，Web ページなら HTML，API なら JSON であることが多いです．

この request と response のやり取りは，R からもそのまま再現できます．`httr2` パッケージで，シンプルな Web ページ https://example.com に GET リクエストを送ってみましょう．

```
resp ← request("https://example.com") ▷
  req_perform()
```

```
resp_status(resp)
## [1] 200
```

ステータスコードが `200` なら成功です. 続けてレスポンスの本文を取り出すと, ブラウザが受け取るのと同じ HTML がそのまま返ってきます (長いので先頭だけ表示します).

```
resp ▷
  resp_body_string() ▷
  substr(1, 320) ▷
  cat()
##     <!doctype     html><html     lang="en"><head><title>Example     Domain</title><link
rel="icon"   href="data:,"><meta   name="viewport"   content="width=device-width,   initial-
scale=1"><style>body{background:#eee;width:60vw;margin:15vh       auto;font-family:system-
ui,sans-serif}h1{font-size:1.5em}div{opacity:0.8}a:link,a:visited{color:#348}</s
```

私たちが普段ブラウザで見ているページは, この HTML をブラウザが解釈して描画したものです. ブラウザは, ここで R がやったのと同じリクエストを送り, 返ってきた HTML を画面に整形して表示しているにすぎません.

6.2 REST API の基本

人間向けの HTML を解析してデータを取り出す方法 (スクレイピング) もありますが, 多くのサービスは「プログラムがデータを取りに来るための窓口」をはじめから用意しています. これが API (Application Programming Interface) です. API にリクエストを送ると, 本文は HTML ではなく JSON のような構造化データで返ってきます. 整形済みのデータがそのまま手に入り, ページの見た目に左右されず仕様も安定しているので, データを取得するなら API が第一の選択です.

Web の API の多くは REST という様式に従っており, 普段ブラウザがページを取りに行くのと同じ HTTP のやり取り (セクション 6.1 で見たリクエストとレスポンス) でデータを受け渡します. 押さえるべきは次の 3 つです.

- **エンドポイント** (endpoint): リクエストの宛先となる URL です.「どのデータが欲しいか」が URL のパスに対応します.
- **クエリパラメータ** (query parameter): URL の末尾に `?key=value&key2=value2` の形でくっつける絞り込み条件です. 期間や対象, 出力形式などを指定します.
- **レスポンス** (response): 多くの場合 JSON 形式のテキストです. ステータスコード (`200` なら成功) もあわせて返ってきます.

JSON (JavaScript Object Notation) は, データ交換でもっとも広く使われる形式です.「キーと値の組」を `{ … }` で, その並びを `[ … ]` の配列で表し, これらを入れ子にしてデータを記述します. 例えば, 国ごとの 1 人あたり GDP を並べると次のようになります.

```
[
  {"country": "Japan", "year": 2022, "gdp_pc": 33834},
  {"country": "Korea", "year": 2022, "gdp_pc": 32423}
]
```

R では `httr2` がこの JSON を自動でリスト (`list`) に変換してくれるので, あとはそこから必要な値を取り出して整えるだけです.

6.3 Application: 世界銀行

R で API を叩くには `httr2` を使います. リクエストを少しずつ組み立て (`req_*`), 最後に送信して (`req_perform`), レスポンスから中身を取り出す (`resp_*`), という流れです.

例として, 世界銀行の World Bank Indicators API を使い, 日本・アメリカ・韓国の 1 人あたり GDP (現在価格, USD) の時系列を取得します.[21] この API のエンドポイントは `https://api.worldbank.org/v2/country/{国コード}/indicator/{指標コード}` という形です.

```
resp ← request("https://api.worldbank.org/v2") ▷
  req_url_path_append(
    "country",
    "JPN;USA;KOR",
    "indicator",
    "NY.GDP.PCAP.CD"
  ) ▷
  req_url_query(format = "json", date = "2000:2022", per_page = 20000) ▷
  req_perform()

resp_status(resp)
## [1] 200
```

`request()` でベース URL からリクエストを作り, `req_url_path_append()` でパスを, `req_url_query()` でクエリパラメータを付け足しています.[22] ここでは出力形式を JSON に, 期間を 2000〜2022 年に, 1 ページあたりの件数を十分大きく指定しています. `req_perform()` で実際にリクエストを送り, `resp_status()` が `200` なら成功です.

[21] 世界銀行 API は認証なしで使え, 経済データの宝庫なので練習に向いています. 指標コード `NY.GDP.PCAP.CD` が 1 人あたり GDP, 国は ISO3 コード (`JPN`, `USA`, `KOR`) で指定します.

[22] パイプで組み立てるので, パラメータの追加や差し替えが読みやすいのが `httr2` の利点です. 古い `httr` パッケージの後継にあたります.

6.3.1 JSON を tibble に整える

レスポンスの本文は `resp_body_json()` でリストに変換します. 世界銀行 API の場合, 返ってくる JSON は 2 要素の配列で, 1 つ目がページ情報 (総件数など) のメタデータ, 2 つ目が実データの配列です.[23]

```
body ← resp_body_json(resp)
records ← body[[2]]

length(records)
## [1] 69
str(records[[1]], max.level = 1)
## List of 8
##  $ indicator      :List of 2
##  $ country        :List of 2
##  $ countryiso3code: chr "JPN"
##  $ date           : chr "2022"
##  $ value          : num 35548
##  $ unit           : chr ""
##  $ obs_status     : chr ""
##  $ decimal        : int 1
```

各レコードは, 国・年・値などをキーに持つリストになっています. ここから欲しいキーだけを `purrr::map_*()` で抜き出して `tibble` に組み立てます. 値が欠損している年は `value` が `NULL` になっているので, `%||%` で `NA` に置き換えておきます.

```
gdp ← tibble(
  country = map_chr(records, \(r) r$country$value),
  year = as.integer(map_chr(records, \(r) r$date)),
  gdp_pc = map_dbl(records, \(r) r$value %||% NA_real_)
)

gdp ▷ arrange(country, year)
## # A tibble: 69 × 3
##    country  year gdp_pc
##    <chr>   <int>  <dbl>
##  1 Japan    2000 39753.
##  2 Japan    2001 34910.
##  3 Japan    2002 33316.
```

[23] この「メタデータ + データ」という二段構えは世界銀行 API 固有の作りです. API ごとに JSON の構造は異なるので, 最初は `str(body, max.level = 2)` などで形を確かめるのが近道です.

```
##  4 Japan    2003 35809.
##  5 Japan    2004 38678.
##  6 Japan    2005 38159.
##  7 Japan    2006 36354.
##  8 Japan    2007 36130.
##  9 Japan    2008 40294.
## 10 Japan    2009 41678.
## # ℹ 59 more rows
```

これで, ブラウザも HTML のパースも介さずに, 分析にそのまま使える整然データ (tidy data) が手に入りました. あとはいつものように可視化できます (図 6.2).

```
ggplot(gdp, aes(year, gdp_pc, color = country, linetype = country)) +
  geom_line(linewidth = 0.8) +
  scale_y_continuous(labels = scales::label_dollar()) +
  scale_color_discrete(name = NULL) +
  scale_linetype_discrete(name = NULL) +
  labs(x = NULL, y = "GDP per capita (current US$)") +
  theme_minimal() +
  theme(
    legend.position = "inside",
    legend.position.inside = c(0.18, 0.85)
  )
```

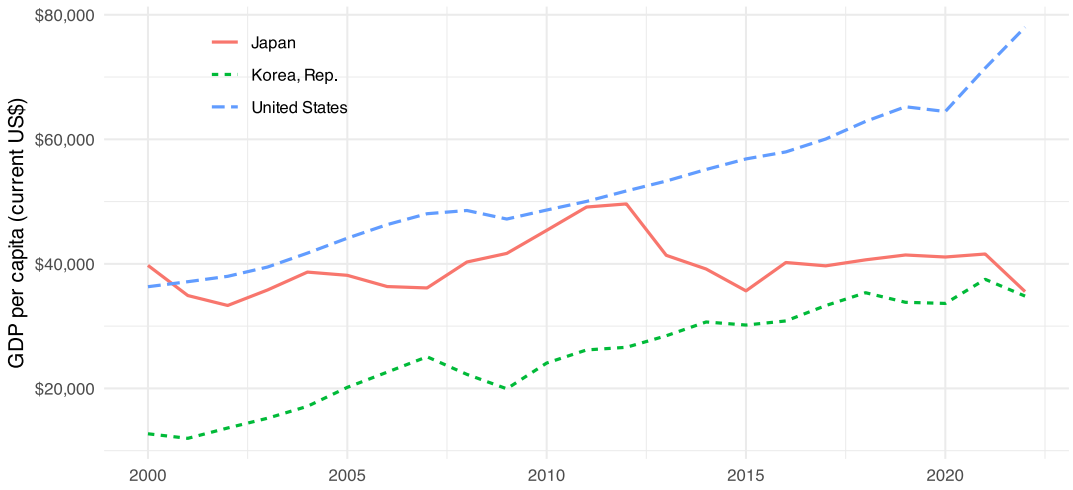


図 6.2: GDP per capita, 2000-2022 (World Bank)

6.3.2 ラッパーパッケージ: WDI

ここまで `httr2` で URL を組み立て, JSON を手でほどいてきました. 仕組みを理解するには大切な作業ですが, 世界銀行のような主要なデータソースには, この一連の流れを 1 つの関数に包んだラッ

パーパッケージが用意されていることがよくあります. 世界銀行なら `WDI` です. さきほどと同じデータが, 次の数行で手に入ります.

```
gdp_wdi ← WDI(
  country = c("JP", "US", "KR"),
  indicator = c(gdp_pc = "NY.GDP.PCAP.CD"),
  start = 2000,
  end = 2022
)

gdp_wdi ▷ as_tibble() ▷ arrange(country, year)
## # A tibble: 69 × 5
##    country iso2c iso3c  year gdp_pc
##    <chr>   <chr> <chr> <int>  <dbl>
##  1 Japan   JP    JPN    2000 39753.
##  2 Japan   JP    JPN    2001 34910.
##  3 Japan   JP    JPN    2002 33316.
##  4 Japan   JP    JPN    2003 35809.
##  5 Japan   JP    JPN    2004 38678.
##  6 Japan   JP    JPN    2005 38159.
##  7 Japan   JP    JPN    2006 36354.
##  8 Japan   JP    JPN    2007 36130.
##  9 Japan   JP    JPN    2008 40294.
## 10 Japan   JP    JPN    2009 41678.
## # ℹ 59 more rows
```

国コード・指標コード・期間を渡すだけで, さきほど手作業で組み立てたのと同じ整然データが返ってきます. `indicator` に名前付きベクトルを渡すと, 値の列名 (`gdp_pc`) もその場で指定できます. ラッパーがある API では, まずこちらを使うのが近道です.

6.4 Application: e-Stat

実用的な API の多くは, 誰がどれだけ使ったかを管理するために, 利用登録と API キー (api key) を求めます. 日本の官庁統計を横断的に提供する e-Stat (政府統計の総合窓口) もその一つです. ここでは, キーの安全な扱い方とあわせて, 都道府県別の 1 人当たり県民所得を取得してみます.

6.4.1 API キーを安全に扱う

e-Stat の API を使うには, まず https://www.e-stat.go.jp/api/ で無料のユーザー登録をして, アプリケーションID (appId) を発行します. このキーをコードに直接書くと, 誤って GitHub に公開したときにそのまま流出してしまいます. そこで, プロジェクト直下の `.Renviron` に次の 1 行を置き, 環境変数として読み込みます.

```
ESTAT_APP_ID=your_application_id_here
```

`.Renviron` は必ず `.gitignore` に加えて，Git の管理対象から外しておきます．R を起動すると `.Renviron` が自動で読み込まれるので，あとは `Sys.getenv()` でキーを取り出せます．キーの値そのものはコードにもログにも残りません．

```
app_id ← Sys.getenv("ESTAT_APP_ID")
```

6.4.2 統計表を探す

世界銀行では仕組みを理解するために手で組み立てましたが，e-Stat にも専用のラッパー `estatapi` があるので，ここでは最初からそれを使います．まず，どの統計表を使うかを探します．`estat_getStatsList()` に検索語を渡すと，条件に合う統計表の一覧が tibble で返ってきます．

```
tables ← estat_getStatsList(
  appId = app_id,
  searchWord = "都道府県データ 経済基盤"
)

tables ▷ distinct(`@id`, STATISTICS_NAME, TITLE)
## # A tibble: 2 × 3
##   `@id`      STATISTICS_NAME                TITLE
##   <chr>      <chr>                          <chr>
## 1 0000010103 都道府県データ 基礎データ      C  経済基盤
## 2 0000010203 都道府県データ 社会生活統計指標 C  経済基盤
```

`@id` の列が，それぞれの統計表の ID (`statsDataId`) です．ここでは「都道府県データ 基礎データ」の経済基盤の表，`0000010103` を使います．[24]

6.4.3 データを取得する

統計表の ID が分かったら，`estat_getStatsData()` に `appId`，`statsDataId`，そして絞り込み条件を渡します．1 人当たり県民所得 (`cdCat01 = "C122101"`) を，東京・愛知・大阪・沖縄の 4 都府県 (`cdArea`) について取得します．都道府県は 5 桁のコード (東京都なら `13000`) で指定します．

```
raw ← estat_getStatsData(
  appId = app_id,
  statsDataId = "0000010103",
  cdCat01 = "C122101",
  cdArea = c("13000", "23000", "27000", "47000")
)
```

[24] 統計表に含まれる指標や地域のコード (`cat01`，`area` など) の一覧は `estat_getMetaInfo()` で調べられます．これも estatapi の関数です．1 人当たり県民所得の指標コードは `C122101` です．

```
dim(raw)
## [1] 44 11
```

整然データの tibble が直接返ってくるので, `httr2` のときのような JSON のパースは要りません. `raw` には, コード列 (`area_code`, `time_code` など) と日本語ラベルの列, そして値 `value` が並んでいます. 必要な列だけ取り出して整えます. 時点コード `"2011100000"` は先頭 4 桁が年度なので, `str_sub()` でそこを取り出します.

```
income ← raw ▷
  transmute(
    area = area_code,
    year = as.integer(str_sub(time_code, 1, 4)),
    income = value
  )

income
## # A tibble: 44 × 3
##    area   year income
##    <chr> <int>  <dbl>
##  1 13000  2011   5219
##  2 13000  2012   5326
##  3 13000  2013   5642
##  4 13000  2014   5648
##  5 13000  2015   5840
##  6 13000  2016   5761
##  7 13000  2017   5843
##  8 13000  2018   5919
##  9 13000  2019   5711
## 10 13000  2020   5204
## # ℹ 34 more rows
```

6.4.4 可視化

単位は千円です. 県名を英語ラベルに直して, 1 人当たり県民所得の推移を描きます (図 6.3).

```
income ▷
  mutate(
    prefecture = recode(
      area,
      "13000" = "Tokyo",
```

```
    "23000" = "Aichi",
    "27000" = "Osaka",
    "47000" = "Okinawa"
  )
) ▷
ggplot(aes(year, income, color = prefecture, linetype = prefecture)) +
geom_line(linewidth = 0.8) +
scale_y_continuous(labels = scales::label_comma()) +
scale_color_discrete(name = NULL) +
scale_linetype_discrete(name = NULL) +
labs(x = NULL, y = "Income per capita (1,000 yen)") +
theme_minimal() +
theme(
  legend.position = "inside",
  legend.position.inside = c(0.18, 0.85)
)
```

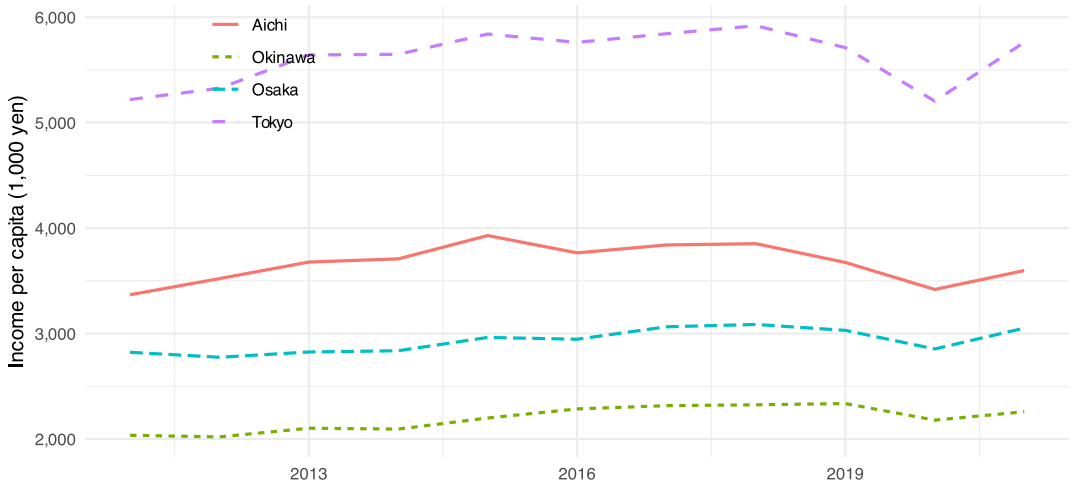


図 6.3: 1 人当たり県民所得の推移 (e-Stat)

⚠ ラッパーパッケージとリクエストの作法

有名なデータソースには, 本章で使った `WDI` や `estatapi` のように, API 呼び出しを R の関数で包んだラッパーパッケージが用意されていることがよくあります. FRED なら `fredr` など, ほかにも数多くあります. 新しいデータソースを使うときは, まずこうしたパッケージが無いか探すのが近道です. 一方で, ラッパーの無い API に出会ったら, 世界銀行の例で見たように `httr2` で自分でリクエストを組み立てることになります. 手で組み立てる方法とラッパーの両方を知っておくと心強いです.

また, API には単位時間あたりのリクエスト数の上限 (rate limit) があります. `httr2` で多数のリクエストを送るときは, `req_throttle()` で送信間隔をあけ, `req_retry()` で失敗時の再試行を設定しておくと安全です.

第 7 章
可視化の技術

この章では, データ可視化に関するいくつかの Tips を紹介します. しかし, これらのテクニックは次の一つの原則に基づいているに過ぎません.

原則 7.1 (可視化の原則). 図は主張をもち, その主張を最短時間で伝えられるようにデザインされなければならない.

これから紹介するテクニックは図の見た目を改善するためのものですが, それらは単に最短時間で図の主張を伝えるために手段であると言うことを忘れないでください.

```
library(dplyr)
library(ggplot2)
```

7.1 最少の要素

最短時間で図を伝えるためには, 過剰な情報を削ぎ落とすことが重要です. これを表した Tufte (2001 年) の有名な原則があります.

原則 7.2 (データインク比の原則). 図においては次のデータインク比を最大化しなければならない:

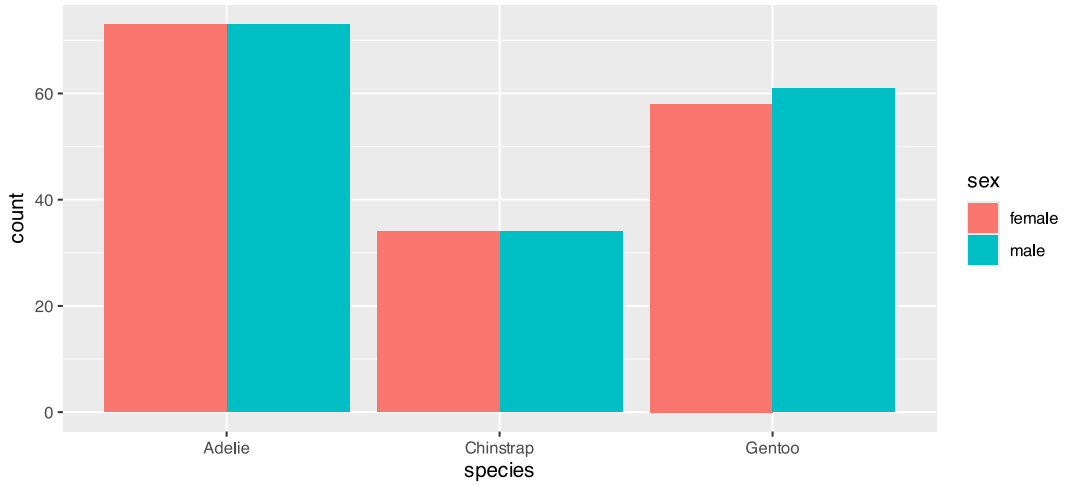
$$\text{data-ink ratio} := \frac{\text{data-ink}}{\text{total ink used to print the graphic}}.$$

ここでいうデータインクとは, 図の中でデータを表すために使われているインクの量のことであり, 基本的には図を構成するために最低限必要な要素のことを指します. ここから, 次のような規則も見出すことができます:

系 7.1. 削除しても図の主張を損なわない要素は削除しなければならない.

この原則を, R に標準で付属する `penguins` データセットで確かめてみましょう. ここではペンギンの種 (`species`) ごとに, 性別 (`sex`) 別の個体数を数えた棒グラフを描きます. まずは何の工夫もしていない, デフォルトのグラフです.

```
penguins_bar ← penguins ▷
  filter(!is.na(sex))


penguins_bar ▷
  ggplot(aes(x = species, fill = sex)) +
  geom_bar(position = "dodge")
```

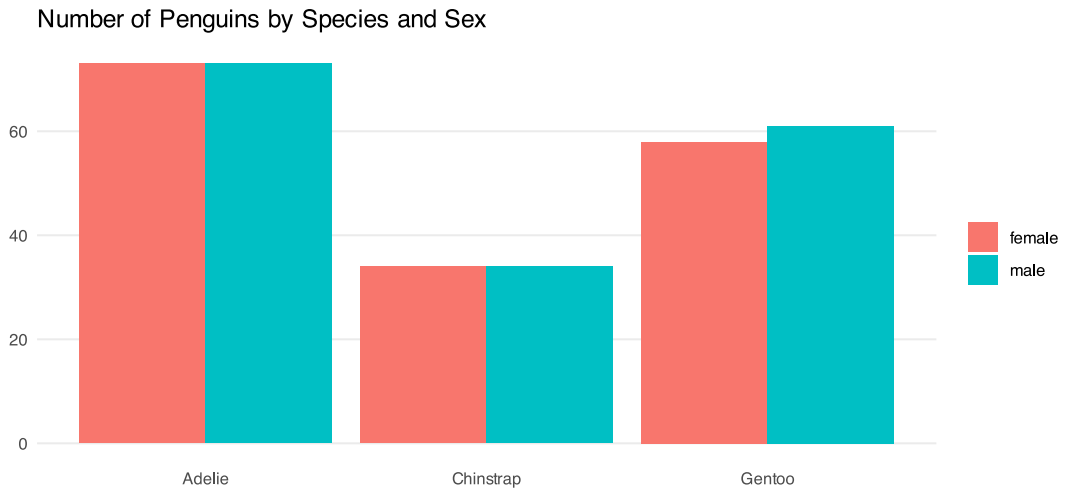


このグラフには

- グレーの背景
- 縦横のグリッド線
- 自明な軸ラベル

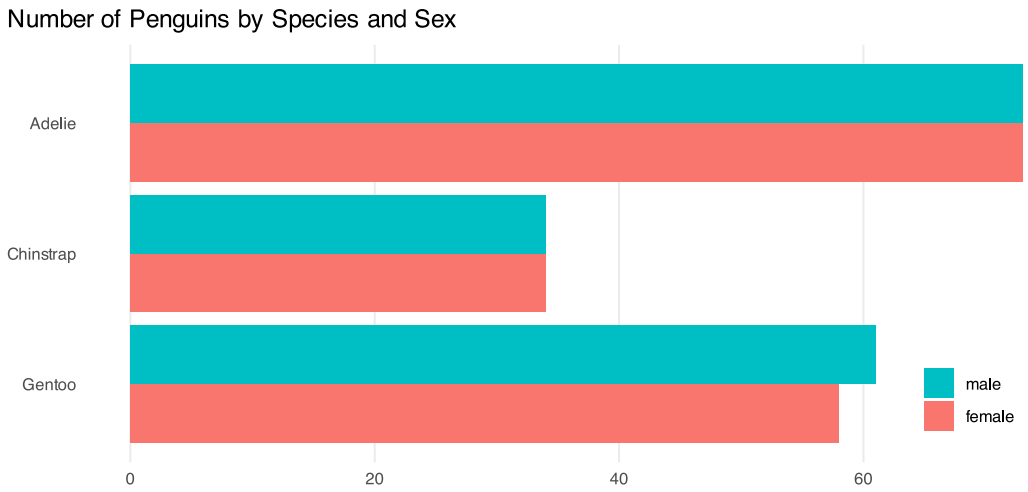
など, データを表していないインクが多く含まれています. データインク比の原則に従って, これらの要素を削ぎ落とします.

```
penguins_bar ▷
  ggplot(aes(x = species, fill = sex)) +
  geom_bar(position = "dodge") +
  labs(
    x = NULL,
    y = NULL,
    fill = NULL,
    title = "Number of Penguins by Species and Sex"
  ) +
  theme_minimal() +
  theme(
    panel.grid.minor = element_blank(),
    panel.grid.major.x = element_blank()
  )
```



さらに可読性を高めるために，棒を横向きにして凡例をパネル内の空白に移します．横向きにするとカテゴリ名が水平に並ぶので，ラベルが長くなっても読みやすくなります．また，凡例をパネル内の空白に移すことで，グラフ全体のバランスが良くなります．

```
penguins_bar ▷
  ggplot(aes(x = forcats::fct_rev(species), fill = sex)) +
  geom_bar(position = "dodge") +
  coord_flip() +
  labs(
    x = NULL,
    y = NULL,
    fill = NULL,
    title = "Number of Penguins by Species and Sex"
  ) +
  theme_minimal() +
  theme(
    panel.grid.minor = element_blank(),
    panel.grid.major.y = element_blank(),
    legend.position = "inside",
    legend.position.inside = c(0.9, 0.15),
    plot.title.position = "plot"
  ) +
  guides(fill = guide_legend(reverse = TRUE))
```



なお, 横向きに変えたことに伴って見づらくなってしまった点があるので

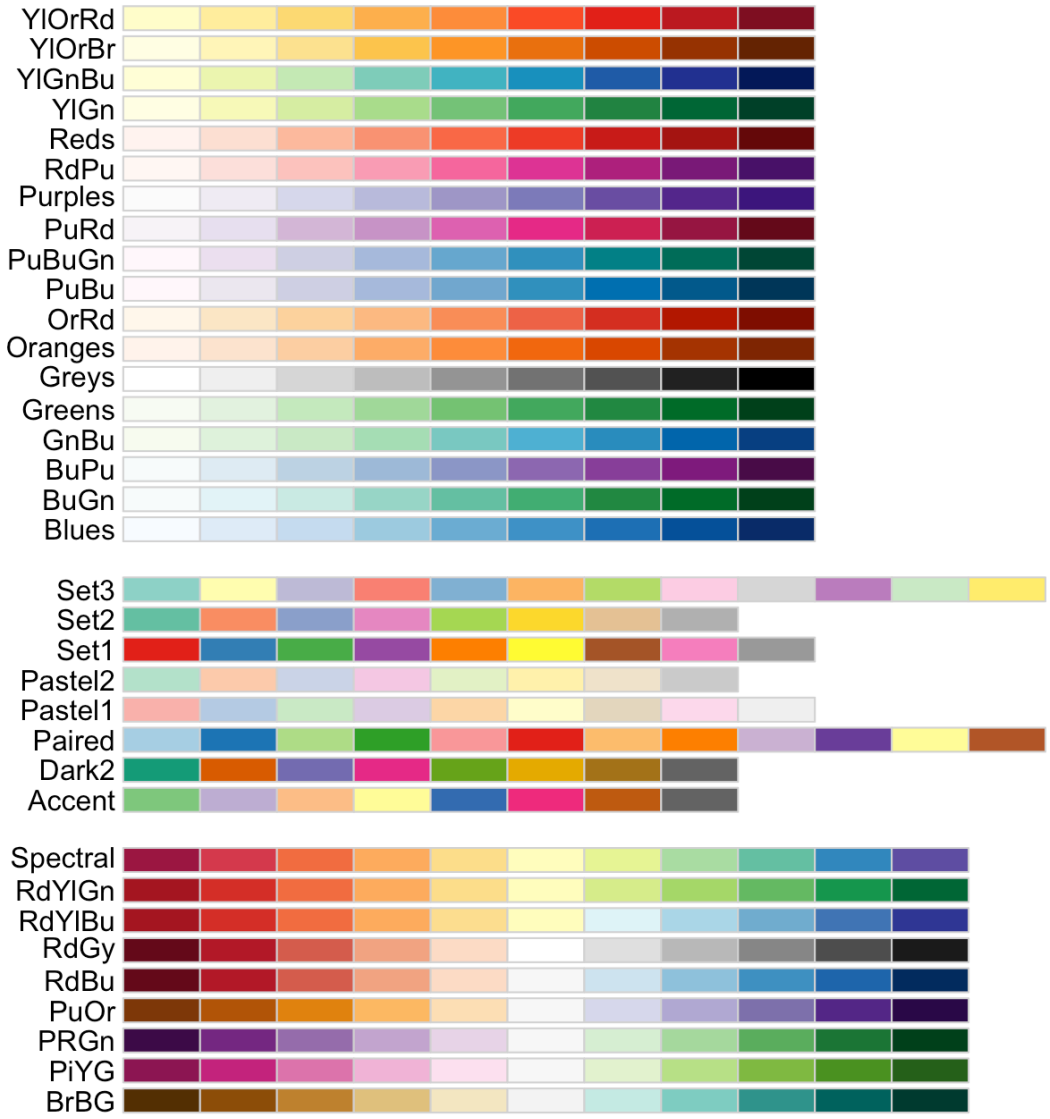
- `forcats::fct_rev()` 関数を使ってカテゴリの順番を逆にしています
- `guides(fill = guide_legend(reverse = TRUE))` を使って, 凡例の順番も逆にしています

7.2　色の選び方

色の選択は重要なテーマで様々な理論がありますが, ここでは深入りせずに用意されているカラーパレットを紹介します. 色彩論に関する簡単な解説は 付録 D を参照してください.

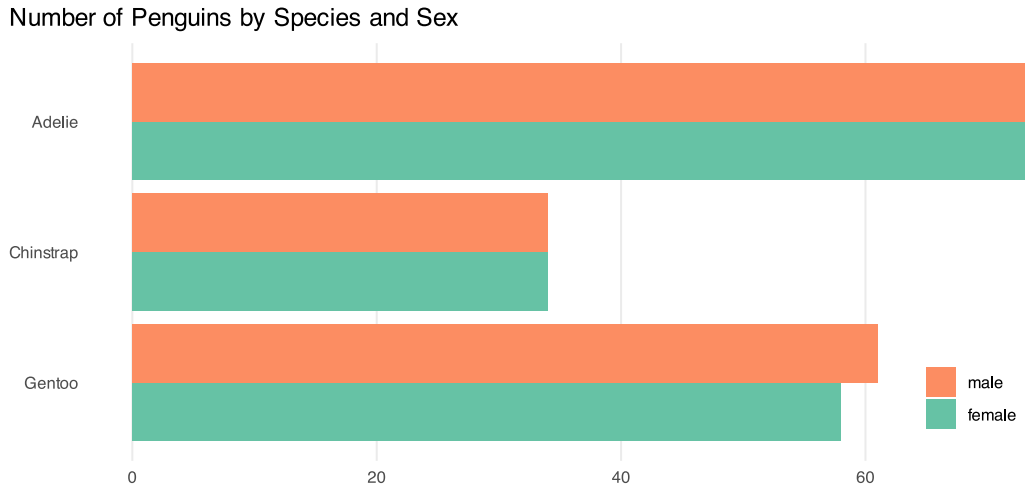
7.2.1　R Color Brewer's Palettes

現代的なカラーパレット集として有名な Color Brewer のパッケージとして, RColorBrewer が提供されています.




```
penguins_bar ▷
  ggplot(aes(x = forcats::fct_rev(species), fill = sex)) +
  geom_bar(position = "dodge") +
  coord_flip() +
  labs(
    x = NULL,
    y = NULL,
    fill = NULL,
```

```
    title = "Number of Penguins by Species and Sex"
  ) +
  scale_fill_brewer(palette = "Set2") +
  theme_minimal() +
  theme(
    panel.grid.minor = element_blank(),
    panel.grid.major.y = element_blank(),
    legend.position = "inside",
    legend.position.inside = c(0.9, 0.15),
    plot.title.position = "plot"
  ) +
  guides(fill = guide_legend(reverse = TRUE))
```

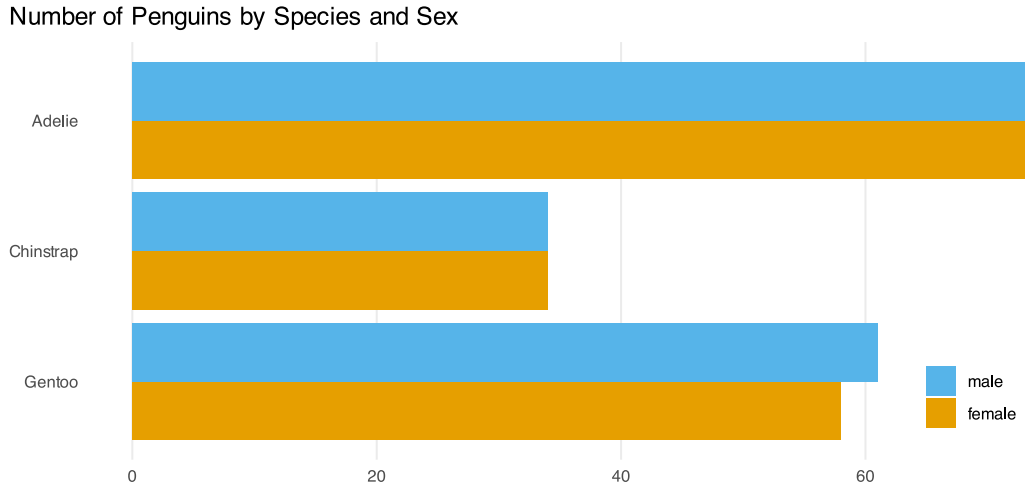


7.2.2　カラー・セーフティ: Okabe-Ito パレット

色選択の一つの方針として, 色覚異常の人でも見分けやすい色を使う, という考え方があります. カラー・セーフティなパレットとして提案された中でも有名なカラーパレットが Okabe-Ito カラーパレットです.

```
penguins_bar ▷
  ggplot(aes(x = forcats::fct_rev(species), fill = sex)) +
  geom_bar(position = "dodge") +
  coord_flip() +
  labs(
    x = NULL,
    y = NULL,
    fill = NULL,
    title = "Number of Penguins by Species and Sex"
```

```
  ) +
  see::scale_fill_okabeito() +
  theme_minimal() +
  theme(
    panel.grid.minor = element_blank(),
    panel.grid.major.y = element_blank(),
    legend.position = "inside",
    legend.position.inside = c(0.9, 0.15),
    plot.title.position = "plot"
  ) +
  guides(fill = guide_legend(reverse = TRUE))
```

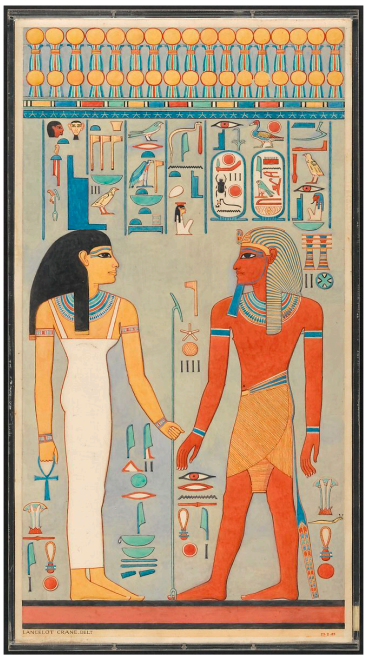


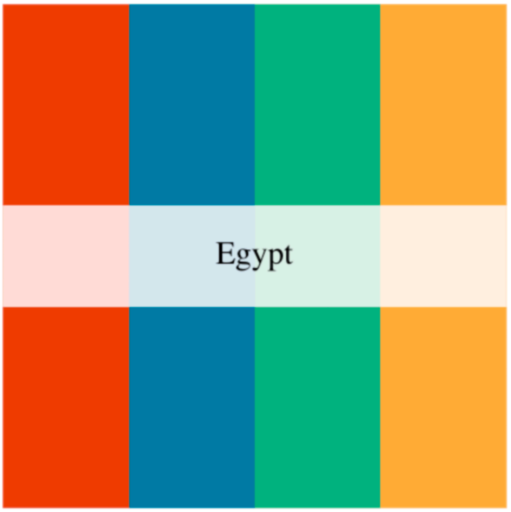
7.2.3 MetBrewer パレット

MetBrewer は，アメリカのメトロポリタン美術館のコレクションからインスピレーションを得たカラーパレットを提供するパッケージです．

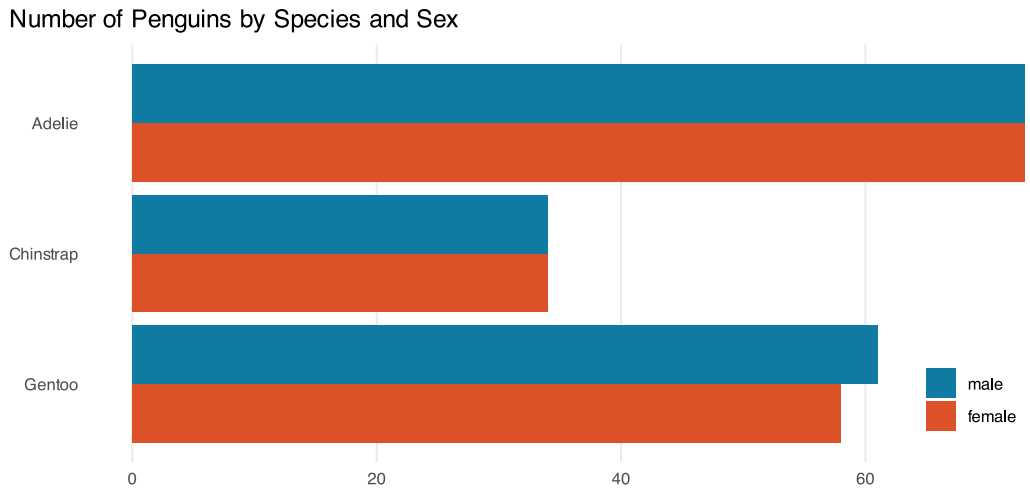
```
# install.packages("MetBrewer")
library(MetBrewer)
```

ここでは，赤・青・緑・橙がはっきり見分けられる Egypt のパレットを使ってみましょう．






```
penguins_bar ▷
  ggplot(aes(x = forcats::fct_rev(species), fill = sex)) +
  geom_bar(position = "dodge") +
  coord_flip() +
  labs(
    x = NULL,
    y = NULL,
    fill = NULL,
    title = "Number of Penguins by Species and Sex"
  ) +
  scale_fill_manual(values = met.brewer("Egypt")) +
  theme_minimal() +
  theme(
    panel.grid.minor = element_blank(),
    panel.grid.major.y = element_blank(),
    legend.position = "inside",
    legend.position.inside = c(0.9, 0.15),
    plot.title.position = "plot"
  ) +
  guides(fill = guide_legend(reverse = TRUE))
```



7.2.4 alpha の活用

alpha は色の透明度 (opacity) を指定するパラメータで, 彩度とは別物です. それでも白い背景の上では, alpha を下げると色が淡くなるので, ひとつの色相の中に順序をつける道具として使えます. 具体的には, グループの違い (ここでは男女) を色相で, グループ内の順序 (正規 → 非正規 → 非就業) を alpha で表す, という二段構えです. こうすると各パネルは 1 色でまとまり, 凡例も中立的なグレーになります (図 7.1).

```
emp ← targets::tar_read(employment_status, store = here::here("_targets")) ▷
  filter(status ≠ "other_employed") ▷ # keep regular / non-regular / not in work
  mutate(
    sex = recode(sex, male = "Men", female = "Women"),
    status = recode(
      status,
      regular = "Regular",
      nonregular = "Non-regular",
      not_employed = "Not in work"
    ),
    status = factor(
      status,
      levels = c("Regular", "Non-regular", "Not in work")
    ),
    age_group = factor(
      age_group,
      levels = c("25-29", "30-34", "35-39", "40-44", "45-49", "50-54", "Total")
    )
  )
```

```
ggplot(emp, aes(x = age_group, y = n, fill = sex, alpha = status)) +
  geom_col(position = position_fill(reverse = TRUE), width = 0.85) +
  facet_wrap(~sex) +
  scale_fill_manual(
    values = c(Men = "#217a6b", Women = "#8a4a9e"),
    guide = "none"
  ) +
  scale_alpha_manual(
    values = c("Regular" = 1, "Non-regular" = 0.55, "Not in work" = 0.25),
    name = NULL
  ) +
  scale_y_continuous(
    labels = scales::label_percent(),
    expand = expansion(c(0, 0.02))
  ) +
  labs(x = NULL, y = NULL) +
  theme_minimal(base_size = 12) +
  theme(
    panel.grid = element_blank(),
    axis.text.x = element_text(size = 8),
    legend.position = "bottom",
    strip.text = element_text(face = "bold", size = 13)
  )
```

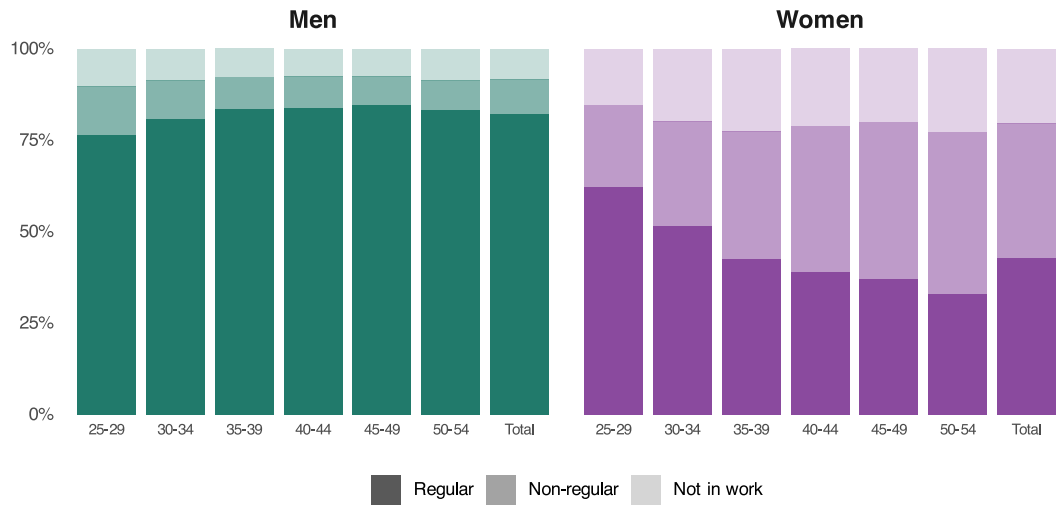


図 7.1:　男女・年齢別にみた就業状態の構成

データは就業構造基本調査 (令和 4 年) の全国・総数で, 各年齢層の正規・非正規・非就業の構成比です (自営業主・会社役員などは除いています). 男性はほとんどが正規, 女性は年齢とともに非正規と非就業が増えるという違いが, 色相 (男女) と alpha (状態) の重ねがけで一目で読み取れます.

ただし alpha は背景と混色する指定なので, この見せ方が有効なのは背景が一様 (白) のときに限られます. 背景が濃い場合や図を重ねる場合は, alpha ではなく明度をそろえた色 (HCL / OKLCh) で段階を作る方が安全です (付録 D).

7.3 フォント

7.3.1 フォントの種類

フォントには大きく分けて, セリフ体 (serif) とサンセリフ体 (sans-serif) の二種類があります. セリフ体は文字の端に装飾があるフォントで, サンセリフ体は装飾のないフォントです. 下の左の図で赤く示された, 文字の端の突起や飾りが「セリフ」です.

図 7.2: セリフ体 (serif)

図 7.3: サンセリフ体 (sans-serif)

日本語でも明朝体がセリフ体でゴシック体がサンセリフ体に相当します. 英語では, セリフ体は Times New Roman や Garamond などが有名で, サンセリフ体は Arial や Helvetica などが有名です.

7.3.2 フォントの選び方

基本的に図表はサンセリフ体を使うことが推奨されます. これは, サンセリフ体の方が小さいサイズでも読みやすいからです. `ggplot2` では, デフォルトでサンセリフ体 (多くの場合のシステムに入っている Helvetica か Arial) が使われています.

ただし, 日本語でラベルを追加しようと思うと, Helvetica や Arial には日本語が含まれていないため, 日本語が豆腐化 (□) してしまいます. 日本語を表示するためには, 日本語が含まれているフォントを指定する必要があります. 例えば, Windows では Meiryo, Mac では Hiragino Kaku Gothic ProN などが日本語を含むサンセリフ体のフォントとしてよく使われます.

7.3.3 フォントの指定方法

グラフのフォントを指定するには大きく 2 つの問題があります:

1. **システムにフォントが入っているか**. コードを共有する相手の PC に, そのフォントが入っていないと, そのフォントは使えない場合があります. 例えば, Mac のヒラギノ系は有料フォントであるため, Mac ユーザ以外が持っていることは想定しないほうが良いです.
2. **R がそのフォントを使えるか**. R の図はグラフィックデバイスとよばれる仕組みを通じて描かれており, フォントを見つけて出力に反映できるかどうかはデバイスごとに異なります (詳しくはセクション C.4 を参照). 例えば, PDF デバイスではシステムのフォントを自動では参照せず, デバイスに登録された Type 1 フォントしか使えません. さらにデフォルトではフォントを埋め込まず, PDF にはフォント名だけが書き込まれるため, 閲覧環境に同じフォントがなければ表示が崩れます. 埋め込むには Ghostscript 経由の `embedFonts()` が別途必要です.

これらの問題をシンプルに解決するのが, showtext パッケージです.[25] showtext は, 文字をフォントとしてではなくグリフのアウトライン (曲線) として描画することで, この問題をデバイスや閲覧環境によらず回避します. さらに font_add_google() を使えば, Google Fonts からフォントをダウンロードして登録できるため, フォントのインストール自体も不要になります.

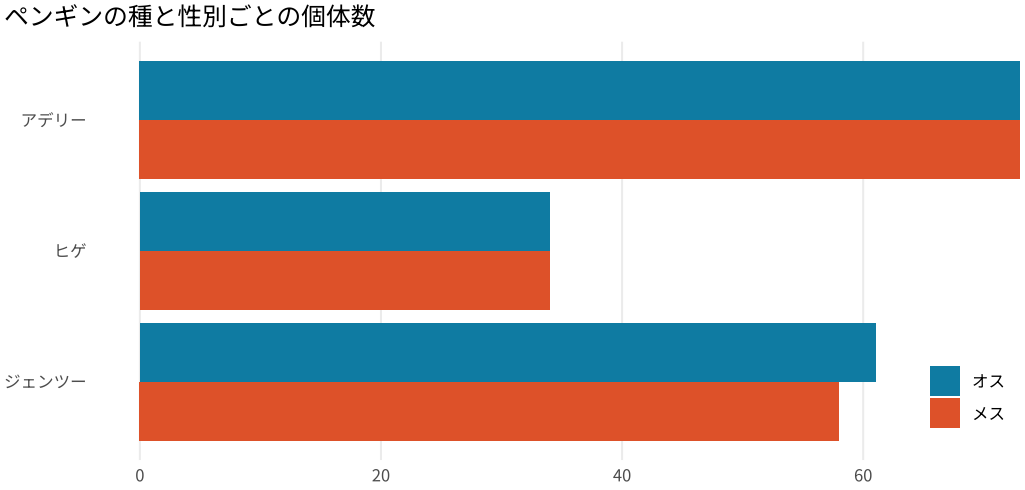
```
library(showtext)


font_add_google("Noto Sans JP")
showtext_auto()
```

showtext_auto() は, この描画方法を以後に開かれるすべての描画デバイスで自動的に有効にするスイッチです. Quarto (knitr) はチャンクごとに新しいデバイスを開くため, これを最初に一度呼んでおかないと, チャンクごとに showtext_begin() と showtext_end() で囲む必要が出てきます.

```
penguins_bar ▷
  mutate(
    species = recode(
      species,
      "Adelie" = "アデリー",
      "Chinstrap" = "ヒゲ",
      "Gentoo" = "ジェンツー"
    ),
    sex = recode(
      sex,
      "male" = "オス",
      "female" = "メス"
    )
  ) ▷
  ggplot(aes(x = forcats::fct_rev(species), fill = sex)) +
  geom_bar(position = "dodge") +
  coord_flip() +
  labs(
    x = NULL,
    y = NULL,
    fill = NULL,
    title = "ペンギンの種と性別ごとの個体数"
  ) +
  scale_fill_manual(values = met.brewer("Egypt")) +
```

[25]論文に適した図を作成する, 用途に応じて適切なグラフィックデバイスを選択する, といった観点では, showtext は厳密に言えば最適な解決策ではありません (セクション C.4 参照). しかし, フォントのインストールや埋め込みの手間を考えると, まずは showtext を使うのが最も簡単です.

```
  theme_minimal(base_family = "Noto Sans JP") +
  theme(
    panel.grid.minor = element_blank(),
    panel.grid.major.y = element_blank(),
    legend.position = "inside",
    legend.position.inside = c(0.9, 0.15),
    plot.title.position = "plot"
  ) +
  guides(fill = guide_legend(reverse = TRUE))
```



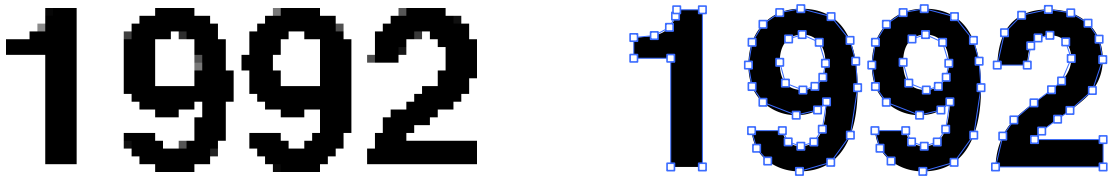
7.4 画像形式

レポートやスライドに図を載せる場合, 画像形式の選択が見栄えに影響を与えます. ここでは画像形式について最低限知っておくべきことを紹介します.

7.4.1 ラスターとベクター

画像形式は, 大きく分けてラスター (raster) 形式とベクター (vector) 形式の 2 種類があります. ラスター形式はビットマップ (bitmap) とも呼ばれ, 画像を色のついたピクセル (画素) の格子として記録します. 一方, ベクター形式は画像を点・線・曲線・塗りといった図形の数式として記録します. この記録方法の違いが, 拡大したときの見え方とファイルサイズを大きく左右します.

同じ「1992」という文字を, ビットマップ (ラスター) とベクターのそれぞれで表してみましょう. 左はピクセルの格子, 右は輪郭を点とパスで定義したものです.



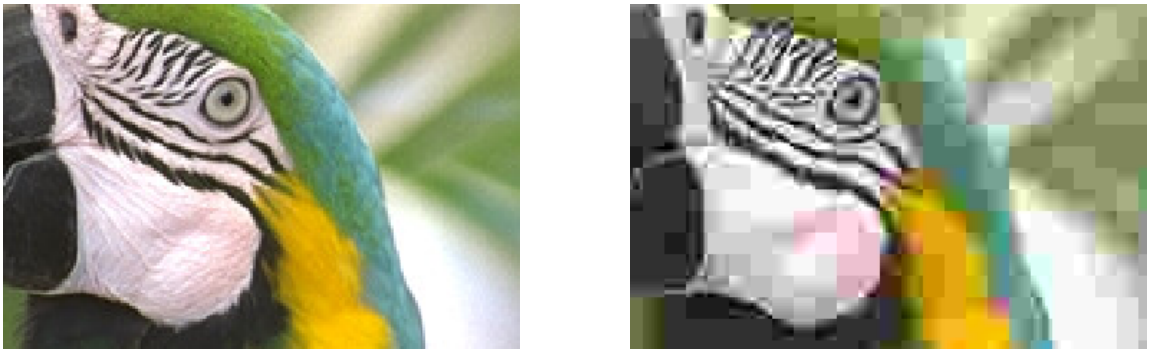
(a) Bitmap (raster)　　(b) Vector

図 7.4: The number 1992 as a bitmap (pixels) versus a vector (points and paths).

左のビットマップは, 拡大するとピクセルのギザギザ (ジャギー) が目立ちます. きれいに見せるにはピクセル数を増やすしかなく, その分ファイルサイズも大きくなります. 一方, 右のベクターは輪郭を点 (アンカーポイント) とパスで定義しているので, どれだけ拡大しても滑らかなままで, ファイルサイズも解像度に依存しません.

7.4.2　JPEG の非可逆圧縮

写真は本質的にラスター形式です. そして, ラスター形式のファイル形式のうち, JPEG は非可逆圧縮 (lossy compression) を採用しています. ファイルサイズを大きく減らせる代わりに, 圧縮を強めると輪郭のまわりにノイズ (アーティファクト) が生じます. 1 枚の写真[26] を高品質 (quality 90) と低品質 (quality 5) の JPEG で保存して, 同じ場所を拡大して比べてみます.



(a) JPEG quality 90　　(b) JPEG quality 5

図 7.5: The same photo saved as high- and low-quality JPEG, magnified.

低品質の JPEG では, 8x8 ピクセルのブロック状のムラや, 輪郭の周りのにじみがはっきり見えます. 一方で, ファイルサイズは大きく変わります. 同じ写真を PNG (可逆圧縮) と 2 種類の JPEG で保存し, サイズを比べてみましょう.

表 7.1: 同じ写真を各形式で保存したときのファイルサイズ

Format	Size (KB)
PNG	546
JPEG (quality 90)	95
JPEG (quality 5)	5

JPEG は, 写真のように色がなめらかに変化する画像にはとても効果的です. しかし, 輪郭のはっきりしたグラフや文字, ロゴでは, 線の周りに同じノイズが乗ってしまいます. そのため, グラフをラスター形式で保存するなら JPEG ではなく PNG を使い, 可能ならベクター形式を選びます.

[26] Kodak True Color Image Suite (kodim23). 画像圧縮のベンチマークに広く使われている, 利用制限なしで公開されたテスト画像です.

7.4.3 ベクター形式の利点

ベクター形式は図形を数式で記録するので, どれだけ拡大しても輪郭は滑らかなままで, ファイルサイズも解像度に依存しません. グラフは点・線・文字でできているので, ベクター形式と非常に相性が良いです. ggplot のグラフはベクター形式 (SVG や PDF) で出力できます.

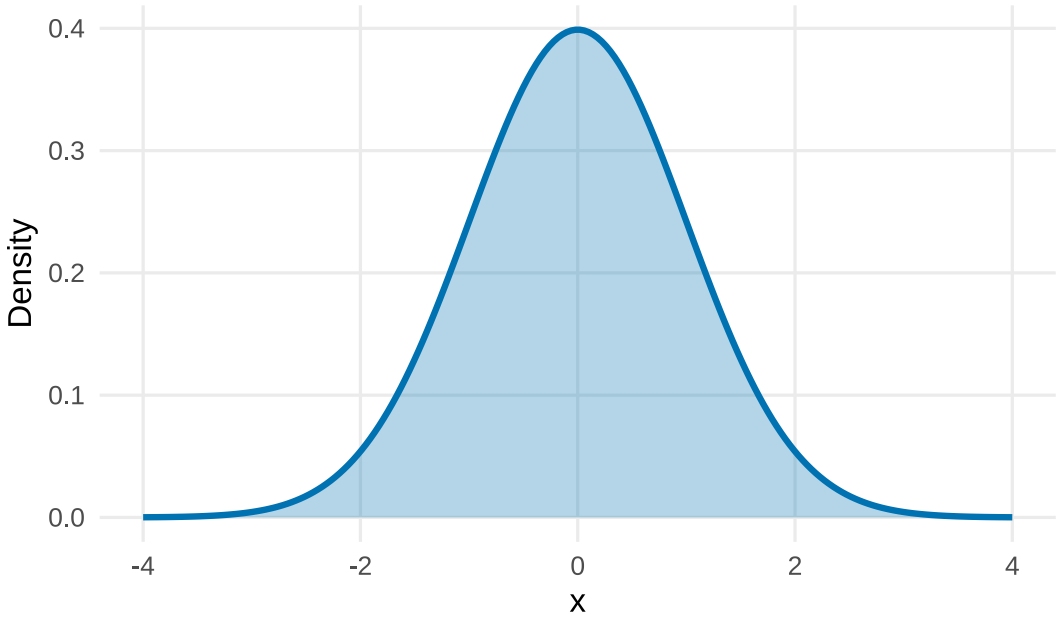


図 7.6: A vector graphic stays sharp at any zoom level.

この図は SVG (ベクター形式) で埋め込まれているので, ブラウザで拡大しても曲線も文字も滑らかなまま保たれます.

7.4.4 画像形式の選び方

知っておくべきベクター形式は SVG と PDF です.

SVG (Scalable Vector Graphics) は Web 上で使われるベクター形式です. Web ブラウザでそのまま表示でき, 拡大しても画質が劣化しません. XML 形式のテキストなので, エディタやプログラムで後から編集できます.

PDF (Portable Document Format) は, 印刷や論文の図でよく使われるベクター形式です. 拡大しても劣化せず, LaTeX をはじめ多くのアプリケーションがそのまま扱えます.

知っておくべきラスター形式は PNG と JPEG です.

PNG (Portable Network Graphics) は, 可逆圧縮のラスター形式です. 圧縮で画質が劣化せず, 透明な背景もサポートするので, ベクターが使えない場面でのグラフ出力に向いています.

JPEG (Joint Photographic Experts Group) は, 非可逆圧縮のラスター形式です. 写真のような複雑な画像を小さなファイルサイズで保存できますが, グラフや文字のような単純な画像には不向きです.

これらを踏まえると, 図の用途ごとの選び方は次のようにまとめられます.

用途	推奨形式
統計グラフ (Web)	SVG
統計グラフ (印刷・論文)	PDF

用途	推奨形式
ベクターが使えない場面のグラフ	PNG
写真	JPEG

統計グラフはほとんどの場合ベクター形式が適しています.

7.4.5　ベクター形式が不向きな場合

ただし, 例外もあります. ベクター形式は図形を 1 つずつ記録するので, 描く図形の数が増えるほどファイルが大きくなります. 例として, ggplot2 に付属する `diamonds` データセットで, 53,940 個のダイヤモンドの重さ (carat) と価格の散布図を描いてみます.

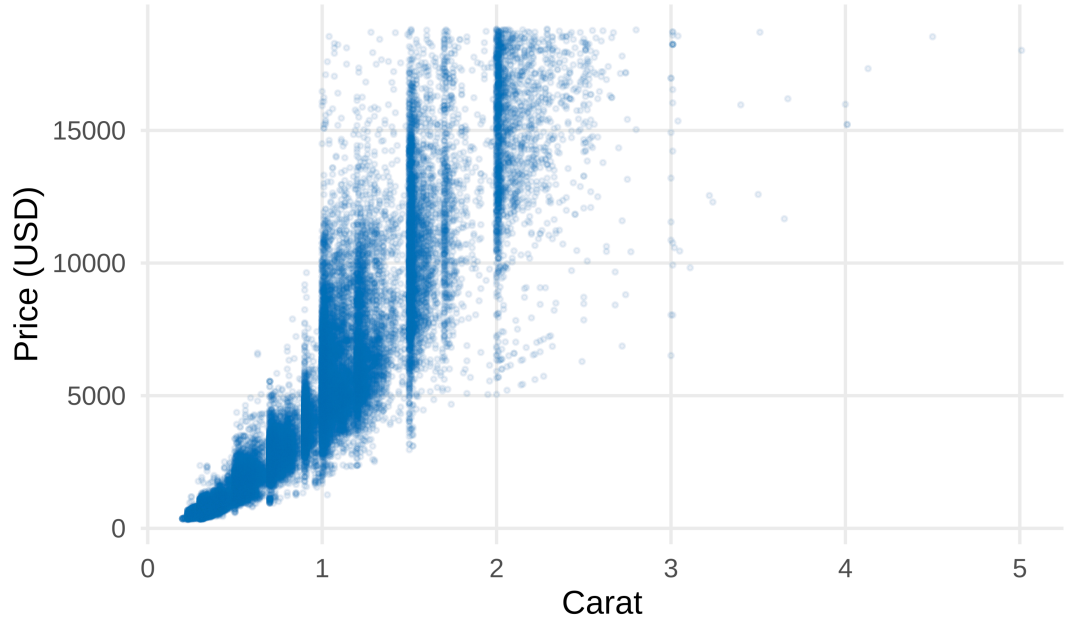


図 7.7: A scatter plot of 53,940 diamonds, embedded as a PNG.

この図を PNG, SVG, PDF のそれぞれで保存して, ファイルサイズを比べてみましょう.

表 7.2: 点の多い散布図を各形式で保存したときのファイルサイズ

Format	Pixels	Size (MB)
PNG (200 dpi)	1000 x 600	0.23
PNG (600 dpi)	3000 x 1800	1.48
SVG	-	21.22
PDF	-	1.36

ラスター形式のファイルサイズは解像度に依存するので, 表には PNG のピクセル数を併記し, Web 表示には十分な 200 dpi と, 印刷にも耐える 600 dpi (このページに埋め込んだ図と同じ解像度) の 2 通りを載せています. SVG は約 5.4 万個の点を XML のテキストとして 1 つずつ記録するため, 高解像度の PNG と比べても 1 桁以上大きくなります. PDF は圧縮が効くため, ファイルサイズだけなら高解像度の PNG と同程度です. しかしベクター形式の問題はサイズだけではありません. ブラウザや PDF ビューアは表示のたびにすべての点を描画し直すので, 表示やスクロールが目に見えて遅くなります. 一方 PNG のファイルサイズと描画の重さは解像度だけで決まり, 点の数には依存しません. こ

のように, データ点が非常に多い散布図では, ベクター形式ではなく高解像度の PNG を使う方が実用的です.

第 8 章
回帰分析

8.1　表

論文や資料に載せる表は, 長らく `kableExtra` や `gt` で作るのが定番でした. しかし近年は, それらより軽量で扱いやすい `tinytable` が標準になりつつあります. `tinytable` は base R だけで動く依存ゼロのパッケージでありながら, セルの結合や色付け, 数式の埋め込みといった凝った表現にも対応し, HTML・LaTeX・PDF・Typst のどの形式にも同じコードから書き出せます. 後で紹介する `modelsummary` の回帰表もそのまま `tinytable` のオブジェクトとして返ってくるため, 一度使い方を覚えれば表まわりはこれ一つで完結します.

ここでは R にに同包されている `penguins` データセットを使って, `tinytable` の基本的な使い方を紹介します.

8.1.1　基本: `tt()`

`tinytable` の出発点は `tt()` です. データフレーム (`data.frame`, `tibble`) を渡すだけで表になります. まずは種ごとの個体数と, くちばしの長さ・体重の平均をまとめてみましょう.

```
penguins_summary ← penguins ▷
  summarize(
    n = n(),
    bill_len = mean(bill_len, na.rm = TRUE),
    body_mass = mean(body_mass, na.rm = TRUE),
    .by = species
  )

penguins_summary ▷
  tt()
```

表 8.1: ペンギンの種別の要約統計量

species	n	bill_len	body_mass
Adelie	152	38.79139	3700.662
Gentoo	124	47.50488	5076.016
Chinstrap	68	48.83382	3733.088

8.1.2　数値の整形: format_tt()

表 8.1 は平均値の桁数がばらばらで読みにくいです. format_tt() を使うと, 列を指定して桁数や区切り文字をそろえられます. j で対象の列を, digits で小数点以下の桁数を, num_mark_big で 3 桁ごとの区切り文字を指定します.

```
penguins_summary ▷
  tt() ▷
  format_tt(j = "bill_len", digits = 1, num_fmt = "decimal") ▷
  format_tt(
    j = "body_mass",
    digits = 0,
    num_fmt = "decimal",
    num_mark_big = ","
  )
```

表 8.2: 桁数をそろえた要約統計量

species	n	bill_len	body_mass
Adelie	152	38.8	3,701
Gentoo	124	47.5	5,076
Chinstrap	68	48.8	3,733

欠損値の扱いも format_tt() の仕事です. 集計の途中で生じた NA は, 表の上では空白やハイフンに置き換えたいことがほとんどです. その場合は replace 引数を使います. 例えば種ごと・島ごとの個体数をクロス集計すると, ペンギンの生息分布には偏りがあるため (Gentoo は Biscoe 島, Chinstrap は Dream 島にしかいません), 観測のない組み合わせが NA になります.

```
penguins ▷
  summarize(n = n(), .by = c(species, island)) ▷
  pivot_wider(names_from = island, values_from = n) ▷
  tt() ▷
  format_tt(replace = "-")
```

表 8.3: 種と島ごとの個体数

species	Torgersen	Biscoe	Dream
Adelie	52	44	56
Gentoo	-	124	-
Chinstrap	-	-	68

8.1.3　スタイル: style_tt()

見出しを太字にする, 特定の行を強調する, セルに色を付けるといった装飾は style_tt() がまとめて引き受けます. 行は i, 列は j で指定し (i = 0 はヘッダ行を指します), bold・italic・color・background などを組み合わせます. ここでは, 体重が 4000g を超える行に背景色を付けて目立たせてみます. 条件は R 側で計算して i に渡すのがポイントです.

```
penguins_summary ▷
  tt() ▷
  format_tt(j = "bill_len", digits = 1, num_fmt = "decimal") ▷
  format_tt(
    j = "body_mass",
    digits = 0,
    num_fmt = "decimal",
    num_mark_big = ","
  ) ▷
  style_tt(i = 0, bold = TRUE) ▷
  style_tt(i = which(penguins_summary$body_mass > 4000), background = "#ffe9b3")
```

表 8.4: 平均体重が大きい種を強調

species	n	bill_len	body_mass
Adelie	152	38.8	3,701
Gentoo	124	47.5	5,076
Chinstrap	68	48.8	3,733

8.1.4 行と列のグループ化: `group_tt()`

複数の列を一つの見出しでまとめたり (LaTeX の `multicolumn` に相当), 行をカテゴリごとに束ねたり (`multirow` に相当) するのが `group_tt()` です. 列方向のグループは `j` に, 行方向のグループは `i` に, それぞれ名前付きリストで「見出し = 範囲」を渡します.

まず列をまとめてみます. くちばし関連の指標と体格関連の指標を, それぞれ一つの見出しの下に置きます.

```
penguins ▷
  summarize(
    bill_len = mean(bill_len, na.rm = TRUE),
    bill_dep = mean(bill_dep, na.rm = TRUE),
    flipper_len = mean(flipper_len, na.rm = TRUE),
    body_mass = mean(body_mass, na.rm = TRUE),
    .by = species
  ) ▷
  tt() ▷
  format_tt(j = 2:5, digits = 1, num_fmt = "decimal") ▷
  group_tt(j = list("Bill (mm)" = 2:3, "Body" = 4:5))
```

表 8.5: 指標をグループ化した要約統計量

	Bill (mm)		Body	
species	bill_len	bill_dep	flipper_len	body_mass
Adelie	38.8	18.3	190	3700.7
Gentoo	47.5	15	217.2	5076
Chinstrap	48.8	18.4	195.8	3733.1

次に行をまとめます. 性別ごとに種別の平均を並べ, 性別を行の見出しにします. `i` に渡す数値は「その行の直前に見出しを挿入する」位置を表すので, 雌が 1–3 行目, 雄が 4–6 行目なら `list("Female" = 1, "Male" = 4)` とします.

```
penguins ▷
  filter(!is.na(sex)) ▷
  summarize(
    bill_len = mean(bill_len),
    body_mass = mean(body_mass),
    .by = c(sex, species)
  ) ▷
  arrange(sex, species) ▷
  select(-sex) ▷
  tt() ▷
  format_tt(j = "bill_len", digits = 1, num_fmt = "decimal") ▷
  format_tt(
    j = "body_mass",
    digits = 0,
    num_fmt = "decimal",
    num_mark_big = ","
  ) ▷
  group_tt(i = list("Female" = 1, "Male" = 4))
```

表 8.6: 性別で行をまとめた平均値

species	bill_len	body_mass
Female		
Adelie	37.3	3,369
Chinstrap	46.6	3,527
Gentoo	45.6	4,680
Male		
Adelie	40.4	4,043
Chinstrap	51.1	3,939
Gentoo	49.5	5,485

8.1.5 キャプションと脚注

これまでの表ではキャプションを Quarto のチャンクオプション `#| tbl-cap:` で与えていました. こうしておくと番号付けと相互参照 (`@tbl-penguins-basic` のような書き方) が効くので, 文章中から表を参照する場合はこちらを使います. 一方で, 出典や注記を表の下に添えたいときは `tt()` の `notes` 引数を使います.

```
penguins_summary ▷
  tt(notes = "Source: Palmer Station LTER. body_mass の単位はグラム.") ▷
  format_tt(j = "bill_len", digits = 1, num_fmt = "decimal") ▷
  format_tt(
    j = "body_mass",
    digits = 0,
    num_fmt = "decimal",
    num_mark_big = ","
  )
```

表 8.7: 注を付けた表

species	n	bill_len	body_mass
Adelie	152	38.8	3,701
Gentoo	124	47.5	5,076
Chinstrap	68	48.8	3,733
Source: Palmer Station LTER. body_mass の単位はグラム.			

8.1.6 セル内の数式

列見出しやセルに数式を入れたいことはよくあります. `tinytable` ではセルの中に `$ ... $` で数式を書けます. ただし, セルの中身は出力形式へそのまま渡されるだけなので, 数式もその形式の文法で書く必要があります. HTML 出力は MathJax が描画するので LaTeX 記法 (`$\bar{x}$`), この本の PDF は Typst で組んでいるので Typst 記法 (`$macron(x)$`) になります. `$n$` のように単なる英字や添字だけなら両方で通りますが, それ以上の記法は下のように出力形式で切り替えます. HTML 側では, セットアップチャンクで `options(tinytable_html_mathjax = TRUE)` を有効にしておきます (この章の冒頭で設定済みです). 列名を後から付け替えるには `setNames()` を使うのが手軽です.

```
mean_cols ← if (knitr::is_html_output()) {
  c("$\\bar{x}_{\\mathrm{bill}}$", "$\\bar{x}_{\\mathrm{mass}}$")
} else {
  c("$macron(x)_\"bill\"$", "$macron(x)_\"mass\"$")
}


penguins_summary ▷
  setNames(c("Species", "$n$", mean_cols)) ▷
  tt() ▷
```

```
  format_tt(j = 3, digits = 1, num_fmt = "decimal") ▷
  format_tt(j = 4, digits = 0, num_fmt = "decimal", num_mark_big = ",")
```

表 8.8: 列見出しに数式を使った表

Species	n	$\bar{x}_{\text{bill}}$	$\bar{x}_{\text{mass}}$
Adelie	152	38.8	3,701
Gentoo	124	47.5	5,076
Chinstrap	68	48.8	3,733

8.1.7　出力形式とエクスポート

ここまで作った表は, Quarto 上でレンダリングする文書の形式 (HTML / PDF / Typst) に合わせて `tinytable` が自動的に適切な出力へ変換してくれます. 同じコードがどの形式でも通用するのが `tinytable` の大きな利点です. 表を単体のファイルとして書き出したいときは `save_tt()` を使い, 拡張子から形式を判断させます (例: `save_tt("table.tex")` で LaTeX, `save_tt("table.png")` で画像). LaTeX 文書に貼り付けるためにスタイルを落とした素の `tabular` が欲しい場合は, 保存前に `theme_tt("tabular")` を挟みます.

8.2　回帰分析

R で回帰分析を行う方法は数えきれないほどあります. 標準誤差の頑健化には `sandwich` や `estimatr`, 高次元固定効果には `lfe`, 操作変数法には `AER`, 限界効果には `margins` や `mfx`, と用途ごとに別々のパッケージを覚えるのが従来の常識でした. しかし結論から言えば, いまや applied micro の実証で必要になる推定のほとんどは `fixest` ひとつでまかなえます. この節では `lm()` を出発点に, なぜ `fixest` だけ覚えればよいのか, そしてその使い方を見ていきます. 引き続き `penguins` データセットを使い, ペンギンの体重 (`body_mass`) を体格から説明する回帰を例にします.

8.2.1　`lm()` による回帰

最も基本的な回帰は `lm()` です. 第一引数に `y ~ x1 + x2` という formula を, `data` に元データを渡します. 皆さんの多くはすでに使ったことがあるはずなので, ここでは formula 記法だけ手短に復習します.

```
fit_lm ← lm(body_mass ~ flipper_len + bill_len + species, data = penguins)
summary(fit_lm)
##
## Call:
## lm(formula = body_mass ~ flipper_len + bill_len + species, data = penguins)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -808.83 -230.35  -26.16  223.18 1050.37
```

```
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -3904.387    529.257  -7.377 1.27e-12 ***
## flipper_len         27.429      3.176   8.638 2.34e-16 ***
## bill_len            61.736      7.126   8.664  < 2e-16 ***
## speciesChinstrap  -748.562     81.534  -9.181  < 2e-16 ***
## speciesGentoo       90.435     88.647   1.020    0.308
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 340.1 on 337 degrees of freedom
##   (2 observations deleted due to missingness)
## Multiple R-squared:  0.8222, Adjusted R-squared:  0.8201
## F-statistic: 389.7 on 4 and 337 DF,  p-value: < 2.2e-16
```

formula の中ではいくつかの演算子が特別な意味を持ちます. 押さえておきたいのは次の 4 つです.

- `x1:x2` は交互作用項だけを, `x1 * x2` は主効果と交互作用項の両方 (`x1 + x2 + x1:x2`) を展開します.
- `as.factor(x)` や文字列・factor 型の変数は, 自動的にダミー変数に展開されます (上の例の `species` がそれです).
- `log(x)` や `poly(x, 2)` のように, formula の中で関数変換をそのまま書けます.
- `I(x^2)` のように, 算術演算をそのまま使いたいときは `I()` で囲みます.

例えば「フリッパー長の効果が種によって異なる」モデルは `flipper_len * species` と書けます.

```
lm(body_mass ~ flipper_len * species, data = penguins) ▷
  coef()
##                   (Intercept)                  flipper_len
##                 -2535.836802                    32.831690
##              speciesChinstrap                speciesGentoo
##                  -501.358972                 -4251.443811
## flipper_len:speciesChinstrap    flipper_len:speciesGentoo
##                     1.741704                    21.790812
```

8.2.2 パッケージは `fixest` だけでよい

`lm()` は手軽ですが, 実証で必要になる頑健標準誤差・クラスター標準誤差・高次元固定効果・操作変数法・非線形モデルには対応していません. これらを補うために従来は用途ごとのパッケージを使い分けていましたが, `fixest` はそのすべてを単一の統一的なインターフェースで提供します. 対応関係を整理すると次のようになります.

やりたいこと	従来のパッケージ	fixest での書き方
Robust SE	estimatr, sandwich	feols(... , vcov = "hetero")
Clustered SE	estimatr, sandwich	feols(... , vcov = ~ id)
高次元固定効果	lfe::felm	feols(y ~ x | fe1 + fe2)
操作変数法 (2SLS)	AER::ivreg	feols(y ~ x | fe | d ~ z)
ロジット・ポアソン	glm	feglm(), fepois()
イベントスタディ	(専用実装が必要)	feols(y ~ i(t, treat)) + iplot()

機能が豊富なだけではありません. fixest は固定効果の吸収アルゴリズムが非常に高速で, 同じ高次元固定効果モデルを lfe や Stata の reghdfe より速く推定できることがベンチマークで示されています[27]. 数千万行規模のパネルでもストレスなく回せるため, 速度の面でも fixest を選ばない理由はほとんどありません.

8.2.3　fixest の使い方

中心となる関数は feols() (線形回帰) です. 基本の書き方は lm() とほぼ同じで, formula に縦棒 | を加えると, その後ろに書いた変数を固定効果として吸収します. ここでは種 (species) と島 (island) を固定効果に入れてみます.

```
m_fe ← feols(
  body_mass ~ flipper_len + bill_len | species + island,
  data = penguins
)
m_fe
## OLS estimation, Dep. Var.: body_mass
## Observations: 342
## Fixed-effects: species: 3,  island: 3
## Standard-errors: IID
##             Estimate Std. Error t value   Pr(>|t|)
## flipper_len  27.7626     3.1996 8.67688  < 2.2e-16 ***
## bill_len     61.4634     7.1547 8.59063 3.3287e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 337.1     Adj. R2: 0.819624
##               Within R2: 0.463456
```

出力の標準誤差に注目してください. fixest は固定効果モデルでは, 既定で最初の固定効果 (ここでは species) でクラスタリングした標準誤差を返します. 標準誤差の種類は vcov 引数で明示的に切り替えられます. 推定し直さずに summary() で変えることもできます.

[27] ベンチマークは fixest の公式ドキュメント (Benchmarking) と Bergé ほか (2026 年) を参照してください.

```
# Heteroskedasticity-robust (HC1)
feols(body_mass ~ flipper_len | species, data = penguins, vcov = "hetero")
## OLS estimation, Dep. Var.: body_mass
## Observations: 342
## Fixed-effects: species: 3
## Standard-errors: Heteroskedasticity-robust
##             Estimate Std. Error t value  Pr(>|t|)
## flipper_len  40.7054    2.87211 14.1727 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 373.3     Adj. R2: 0.780719
##               Within R2: 0.342011

# Clustered on island
feols(body_mass ~ flipper_len | species, data = penguins, vcov = ~island)
## OLS estimation, Dep. Var.: body_mass
## Observations: 342
## Fixed-effects: species: 3
## Standard-errors: Clustered (island)
##             Estimate Std. Error t value Pr(>|t|)
## flipper_len  40.7054    6.51467 6.24826  0.02467 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## RMSE: 373.3     Adj. R2: 0.780719
##               Within R2: 0.342011
```

操作変数法も同じ関数で書けます. formula の末尾に `| 内生変数 ~ 操作変数` を足すだけです (固定効果と併用する場合は `y ~ 外生変数 | 固定効果 | 内生変数 ~ 操作変数` の順). 非線形モデルは `feglm()` (一般化線形モデル) と `fepois()` (ポアソン回帰) が担当し, 引数は `glm()` とほぼ同じです.

```
# Logit: probability of being male
pen ← penguins ▷
  filter(!is.na(sex)) ▷
  mutate(is_male = as.integer(sex == "male"))

feglm(is_male ~ body_mass + bill_len | species, data = pen, family = binomial)
## GLM estimation, family = binomial, Dep. Var.: is_male
## Observations: 333
## Fixed-effects: species: 3
## Standard-errors: IID
```

```
##           Estimate Std. Error z value   Pr(>|z|)
## body_mass 0.007056   0.000970 7.27537 3.4547e-13 ***
## bill_len  0.642816   0.113466 5.66528 1.4678e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## Log-Likelihood: -81.9   Adj. Pseudo R2: 0.627859
##            BIC: 192.8     Squared Cor.: 0.694077
```

推定したモデルをコンソールで手早く見比べたいだけなら, `fixest` 組み込みの `etable()` が便利です. 論文や資料に載せるきれいな回帰表は, 次の節で `modelsummary` を使って作ります.

8.2.4　回帰表を整える: `modelsummary`

複数のモデルを並べた回帰表は, 前節の `tinytable` がバックエンドの `modelsummary` で作ります. 推定結果のリストを渡すだけで表になり, `output = "tinytable"` を指定すれば「表」の節で紹介したスタイリングがそのまま使えます. まずは何の加工もしていない既定の出力です.

```
models ← list(
  "(1)" = feols(body_mass ~ flipper_len + bill_len, data = penguins),
  "(2)" = feols(
    body_mass ~ flipper_len + bill_len,
    data = penguins,
    vcov = "hetero"
  ),
  "(3)" = feols(
    body_mass ~ flipper_len + bill_len | species,
    data = penguins
  ),
  "(4)" = feols(
    body_mass ~ flipper_len + bill_len | species + island,
    data = penguins
  )
)

modelsummary(models)
```

表 8.9: modelsummary の既定の出力

	(1)	(2)	(3)	(4)
(Intercept)	-5736.897	-5736.897		
	(307.959)	(291.797)		
flipper_len	48.145	48.145	27.429	27.763
	(2.011)	(1.864)	(3.176)	(3.200)
bill_len	6.047	6.047	61.736	61.463
	(5.180)	(4.888)	(7.126)	(7.155)
Num.Obs.	342	342	342	342
R2	0.760	0.760	0.822	0.823
R2 Adj.	0.759	0.759	0.820	0.820
R2 Within			0.462	0.463
R2 Within Adj.			0.459	0.460
AIC	5061.5	5061.5	4962.7	4965.7
BIC	5073.0	5073.0	4981.9	4992.5
RMSE	392.34	392.34	337.62	337.09
FE: species			X	X
FE: island				X

既定の表は変数名がそのまま並び, 統計量も載りすぎて雑然としています. 論文用に仕上げるには, 主に次の 3 つの引数を使います.

- `coef_map`: 表示する係数の選択・並べ替え・改名を, 名前付きベクトルで一度に行います. ここに挙げた係数だけが, 指定した順で表示されます.
- `gof_map`: 表の下部に並ぶ適合度統計量 (goodness-of-fit) を選んで改名します. `raw` (内部名)・`clean` (表示名)・`fmt` (小数桁数) の 3 列を持つ `tibble` で渡すのが柔軟です. 固定効果の有無を示す行 (`FE: species` など) もここで改名できます.
- `stars`: 有意水準のアスタリスクを定義します.

これらを適用したのが次の表です. さらに `modelsummary` の返り値は `tinytable` オブジェクトなので, `group_tt()` を継ぎ足してモデルをグループ分けする見出し (column spanner) を付けられます.

```
cm ← c(
  "flipper_len" = "Flipper length",
  "bill_len" = "Bill length"
)

gm ← tibble(
  raw = c("FE: species", "FE: island", "nobs", "r2.within"),
  clean = c("FE: Species", "FE: Island", "Observations", "Within R2"),
  fmt = c(0, 0, 0, 3)
)
```

```
modelsummary(
  models,
  output = "tinytable",
  stars = c("*" = .1, "**" = .05, "***" = .01),
  coef_map = cm,
  gof_map = gm
) ▷
  group_tt(j = list("Pooled OLS" = 2:3, "Fixed effects" = 4:5))
```

表 8.10: 標準誤差と固定効果を変えた回帰の比較

	Pooled OLS		Fixed effects	
	(1)	(2)	(3)	(4)
Flipper length	48.145***	48.145***	27.429***	27.763***
	(2.011)	(1.864)	(3.176)	(3.200)
Bill length	6.047	6.047	61.736***	61.463***
	(5.180)	(4.888)	(7.126)	(7.155)
FE: Species			X	X
FE: Island				X
Observations	342	342	342	342
Within R2			0.462	0.463
* p < 0.1, ** p < 0.05, *** p < 0.01				

LaTeX 文書に貼り付けたい場合は，「表」の節で触れたように `theme_tt("tabular")` を挟んでから `save_tt("table_reg.tex")` で書き出します．同じ回帰表のコードから HTML でも PDF でも同じ表が得られるのが，`tinytable` ベースで作ることの利点です．

8.2.5　係数のプロット: `ggfixest`

回帰係数を点と区間で並べた係数プロットは，表よりも一目で大小と有意性が伝わります．`fixest` には `coefplot()` / `iplot()` という作図関数が組み込まれていますが，これらは base R グラフィックスを使うため `ggplot2` との相性がよくありません．そこで `ggfixest` を使うと，同じものを `ggplot` オブジェクトとして描けます (`ggcoefplot()` と `ggiplot()`)．

```
ggcoefplot(m_fe) +
  theme_minimal()
```

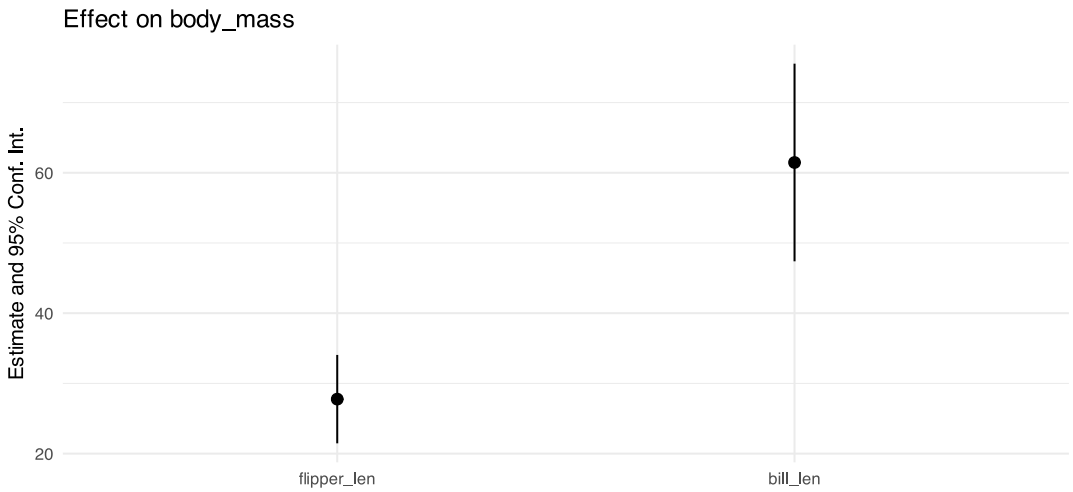


図 8.1: Coefficient plot for the two-way fixed-effects model.

`ggfixest` が特に活躍するのがイベントスタディです. `fixest` では formula の中に `i(time, treat, ref)` と書くと, 処置群について時点ごとの交互作用項 (= イベントスタディの係数) が一括で作られます. `ref` で基準時点を指定します. ここでは `fixest` に同梱されている擬似的な差分の差分 (difference-in-differences) データ `base_did` を使います.

```
data(base_did, package = "fixest")

est_did ← feols(
  y ~ x1 + i(period, treat, ref = 5) | id + period,
  data = base_did
)

ggiplot(est_did) +
  labs(x = "Period", y = "Estimate and 95% CI", title = NULL) +
  theme_minimal()
```

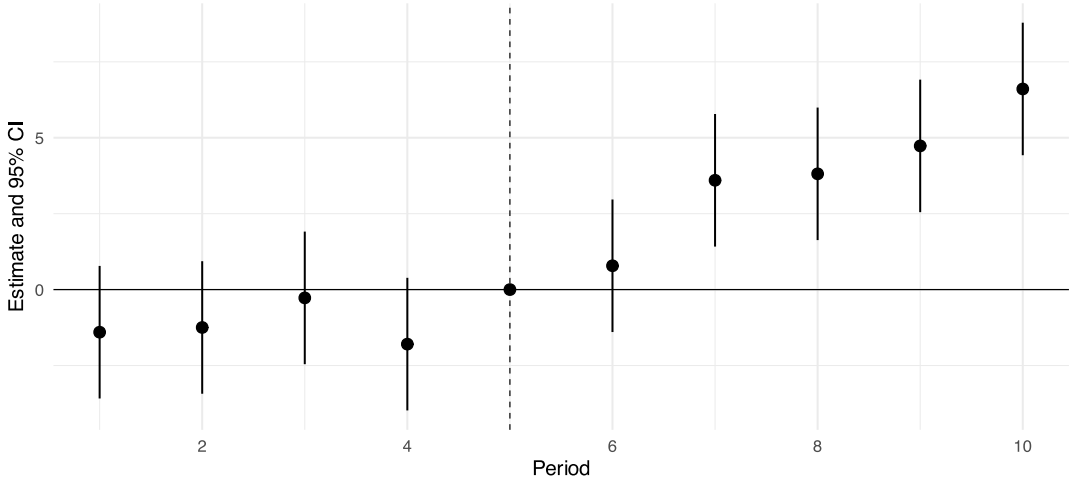


図 8.2: Event-study estimates around treatment (reference period = 5).

処置が時点によってずれて始まる staggered な設定では, 単純な二元配置固定効果推定が誤った推定値を与えうることが知られています (Sun と Abraham 2021 年). その場合は `i()` の代わりに `sunab()` を使うと, Sun と Abraham (2021 年) の補正済み推定量がそのまま得られ, `ggiplot()` で同じように描けます.

第 9 章
スライド

9.1　技術要件

アカデミックなスライドを作る際に次のようなアプリケーションやフレームワークの選択肢があると思います.

- **GUI**: PowerPoint, Keynote, Google Slides
- **LaTeX**: Beamer
- **Typst**: Touying, Polylux
- **Markdown**: Quarto (Reveal.js), Marp, Slidev

私は, この AI 時代の中では Quarto + Touying の組み合わせがもっとも効率的だと思っています. その理由を以下にまとめます.

GUI AI にスライドを作成させるという観点から考えると GUI ベースのアプリケーションは不利です. AI から操作することもできますが, 大量のトークンを消費してしまい非効率です. その点, LaTeX や Markdown はテキストベースであり, AI にとって扱いやすい形式です. さらに, LaTeX や Markdown はスライドの内容をコードとして管理できるため, バージョン管理や再利用が容易です.

HTML スライド Reveal.js, Marp, Slidev などは Markdown で作成するものの, HTML として出力されます. しかし, HTML スライドはアカデミックの文化とあまり相性が良くありません. それは結局 PDF の提出を求められるからです. 学会発表の場やホームページへの掲載, クラウドストレージによる事前共有など, PDF での配布が前提となる場面は多く, HTML スライドはそのままでは使えません.

これらのフレームワークは PDF に変換することもできますが, レイアウトが崩れてしまったり, 数式が正しく表示されなかったりするといった問題が起こりやすいです. また, スライド内のハイパーリンクが効かなくなってしまうため, 参考文献や補足資料へのリンクを貼ることができなくなります.

Quarto 私は, 研究の途中経過を報告する場では, Quarto を用いるのが最善だと思っています. それは, 図表の作成や数値にインラインの変数を使うことができるからです. 研究の途中経過を報告する場合, 発表の前日までデータや分析結果が変わることはよくあります. その時に, Quarto を使えば, 更新された結果が自動で反映されるスライドを作ることが可能です.

Touying Quarto で PDF スライドを作る場合, Official では Beamer がサポートされています. しかし, Beamer はコンパイルが遅く, カスタマイズ性も低いです.[28] Touying は Typst でスライドを作るた

[28]技術的には Beamer も Touying と同じだけのカスタマイズ性があるはずですが, LaTeX 独自の知識がないとカスタマイズが難しいです. またコンパイルが遅いので, 試行錯誤するだけでも時間がかかります.

めのフレームワークであり, Beamer とほぼ同等の機能を持ちながら, Typst の高速コンパイルを利用することができます.

以上を踏まえて, 私は Quarto + Touying の組み合わせでスライドを作っています. そのために以下のようなパッケージも開発しました.

もちろん, 共著のプロジェクトでは Beamer を使わないといけない場合もありますが…

9.2　アンチパターン

経済学の発表を聞いていると, いくつか典型的なアンチパターンがあることに気づきます. ここでは, それらを紹介しながら, スライドの作り方の鉄則を学びましょう.

9.2.1　不必要な要素

スライドからは聴衆の注意をそらす要素はできるだけ排除すべきです. しかし, Beamer のデフォルトの設定ではスライドの右下にナビゲーション記号を表示します. これをクリックしてスライドを操作する人はおらず, 不必要です.

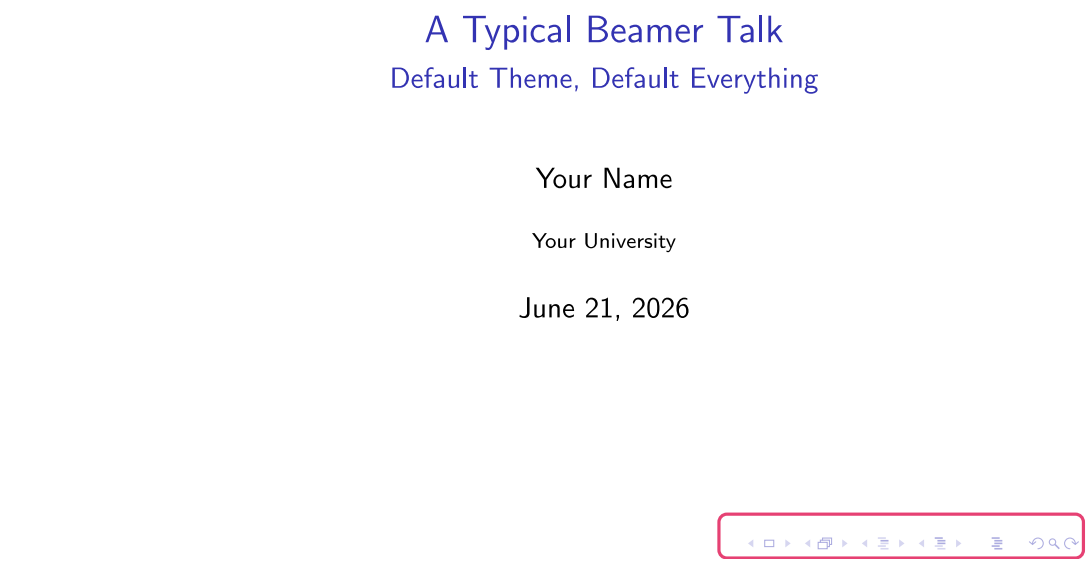


図 9.1: 不必要なナビゲーション

ナビゲーション記号は, プリアンブルに次の一行を加えれば消せます.

```
\setbeamertemplate{navigation symbols}{}
```

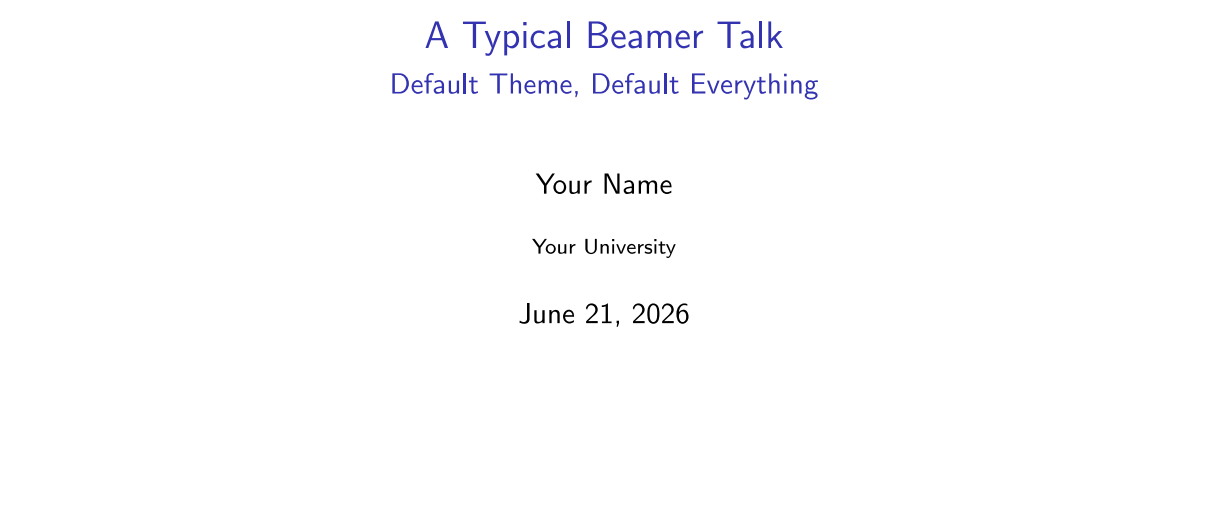


図 9.2:　ナビゲーションを消したスライド

9.2.2　4:3 vs. 16:9

Beamer のデフォルトのアスペクト比は 4:3 ですが，モニターでプレゼンする機会も多くなった現代では，16:9 のスライドの方が画面をより広く使えます．

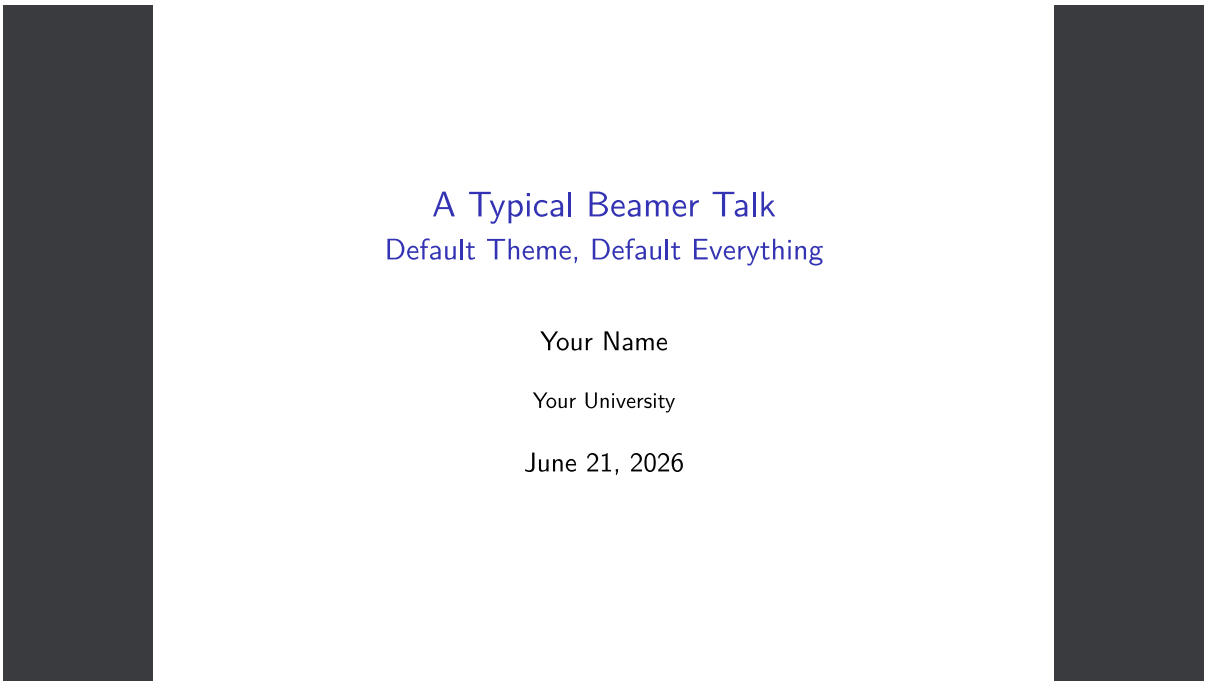


図 9.3:　モニターに写した 4:3 のスライド

16:9 にするには，プリアンブルのドキュメントクラスにオプションを加えます．

```
\documentclass[aspectratio=169]{beamer}
```

16:9 にするメリットは画面を大きく使えるだけでなく，改行によるデザインの崩れを防ぐこともできます. 下の例では, 4:3 だと幅が足りず一文が 2 行に折り返されてしまいますが, 16:9 なら同じ一文が 1 行に収まります.

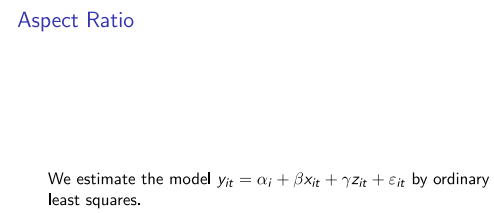


図 9.1: 4:3: 幅が足りず折り返される

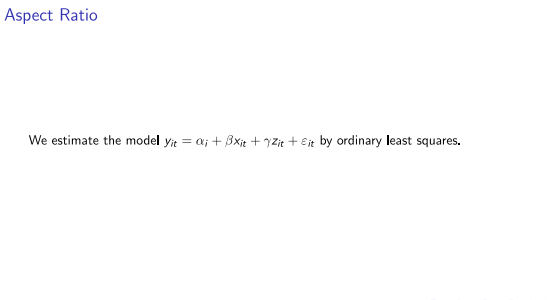


図 9.2: 16:9: 同じ一文が 1 行に収まる

9.2.3　高すぎる彩度

Warsaw や Madrid といった Beamer の定番テーマは, 彩度の高い青がヘッダーやフッター, ブロックなど画面の広い面積を占めます. 色は聴衆の注意を誘導するための手段ですが, テーマ自体が強い色を使っていると, 本来注目してほしい図や数式よりも装飾に目が行ってしまいます.

対策はシンプルで, 彩度を抑えたアクセントカラーを 1 色決め, それ以外は白と黒 (グレー) だけで構成することです. 下の例では, 同じ内容のスライドを Warsaw テーマと, デフォルトテーマ + 落ち着いたアクセントカラーで作っています.

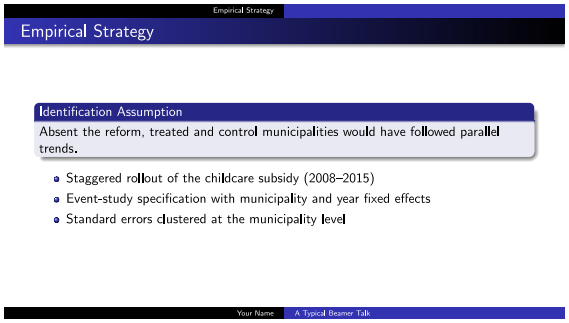


図 9.3: Warsaw Theme

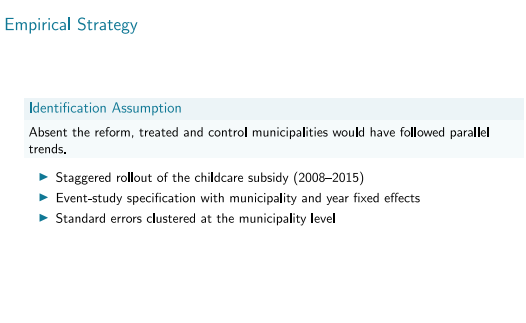


図 9.4: 落ち着いた配色

右のスライドは, プリアンブルで次のように設定しています. `structure` の色を変えると, フレームタイトルや箇条書きのマーカーなどの色がまとめて変わります.

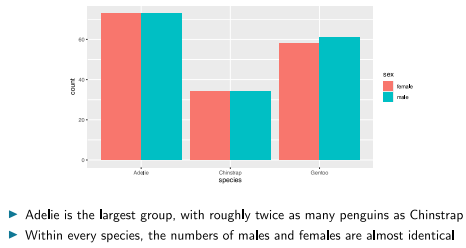
```
\usetheme{default}
\definecolor{accent}{HTML}{107895}
\usecolortheme[named=accent]{structure}
\setbeamercolor{block title}{fg=accent, bg=accent!8}
\setbeamercolor{block body}{bg=accent!4}
```

9.2.4　文字サイズが小さい

図の文字の小ささは，経済学の発表で最もよく見るアンチパターンかもしれません．ggplot2 のデフォルトの文字サイズは論文や画面上の文書を想定した 11pt で，図をスライドに縮小して貼ると，軸や凡例の文字はほとんど読めなくなります．

下の比較では，同じデータの図を同じ大きさでスライドに貼っています．左はデフォルト設定のままの ggplot2，右は チャプター 7 で作った図に文字サイズの指定を加えたものです．

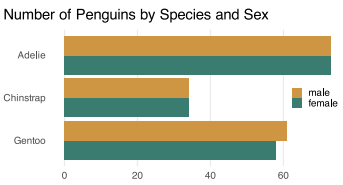
Results



- Adelie is the largest group, with roughly twice as many penguins as Chinstrap
- Within every species, the numbers of males and females are almost identical

図 9.5:　デフォルト設定 (base_size = 11)

Results



- Adelie is the largest group, with roughly twice as many penguins as Chinstrap
- Within every species, the numbers of males and females are almost identical

図 9.6:　チャプター 7 (base_size = 20)

図の文字サイズは，`theme_*()` 関数の `base_size` 引数でまとめて変えられます．右の図はこう指定しています．

```
theme_minimal(base_size = 20)
```

適切な値は図の保存サイズとスライドに貼る大きさに依存しますが，目安として，図中の文字がスライドの本文と同じくらいの大きさに見えれば，会場の後ろの席からも読めます．

9.2.5　囲み線

回帰表をスライドで見せるとき，注目してほしい係数を囲み線で囲んで強調するスライドをよく見かけます．しかし囲み線は，見てほしい情報の周りにインクを足すだけで，見なくてよい情報はそのまま残っています．聴衆の視線は結局，表全体に散らばってしまいます．

強調は「足す」よりも，注目してほしくない要素を「弱める」方が効果的です．下の右の例では，注目してほしい係数以外の行をグレーアウトし，係数の行を太字にしています．表自体が視線を誘導するので，囲み線は不要になります．

Results

	(1)	(2)	(3)
(Intercept)	79.610***	84.080***	62.101***
	(2.104)	(5.782)	(9.605)
Education	-0.862***	-0.963***	-0.980***
	(0.145)	(0.189)	(0.148)
Agriculture		-0.066	-0.155*
		(0.080)	(0.068)
Catholic			0.125***
			(0.029)
Infant.Mortality			1.078**
			(0.382)
Num.Obs.	47	47	47
R2	0.441	0.449	0.699

+ p < 0.1, * p < 0.05, ** p < 0.01, *** p < 0.001

図 9.7:　囲み線による強調

Results

	(1)	(2)	(3)
(Intercept)	79.610***	84.080***	62.101***
	(2.104)	(5.782)	(9.605)
Education	**-0.862*** **	**-0.963*** **	**-0.980*** **
	(0.145)	(0.189)	(0.148)
Agriculture		-0.066	-0.155*
		(0.080)	(0.068)
Catholic			0.125***
			(0.029)
Infant.Mortality			1.078**
			(0.382)
Num.Obs.	47	47	47
R2	0.441	0.449	0.699

+ p < 0.1, * p < 0.05, ** p < 0.01, *** p < 0.001

図 9.8:　グレーアウトと太字による強調

この表は fixest と modelsummary で作っています．`msummary()` に `output = "tinytable"` を指定すると，返ってきた表を `tinytable` の `style_tt()` でそのまま加工できます．

```
library(modelsummary)
library(tinytable)

tab ← msummary(models, output = "tinytable", stars = TRUE)
tab ▷
  style_tt(i = c(1:2, 5:12), color = "#8f8f8f") ▷ # gray out all other rows
  style_tt(i = 3, bold = TRUE) # bold the Education estimate
```

9.3　Quarto + Touying

ここからは, quarto-touying-typst を使ってスライドを作る流れを一通り見ていきます. この節で扱うのは日常的に使う機能だけです. 完全なチュートリアルは公式ドキュメントの Tutorial に, テーマごとの実際の見た目は Gallery にあります.

9.3.1　セットアップ

Quarto プロジェクトのフォルダで以下を実行すると, 拡張が `_extensions/` 以下にインストールされます.

```
quarto add kazuyanagimoto/quarto-touying-typst
```

ゼロから始める場合は, サンプルスライド付きのテンプレートを使うのが手軽です.

```
quarto use template kazuyanagimoto/quarto-touying-typst
```

スライドの本体は, `format: touying-typst` を指定した通常の `.qmd` ファイルです. 最小限の例を示します.

```
---
title: My Talk
subtitle: A subtitle
format: touying-typst
theme: metropolis
author:
  - name: Your Name
    affiliations: Your Institution
date: today
---
```

`quarto render slides.qmd` で PDF が生成されます. タイトルスライドは, YAML の `title`, `subtitle`, `author`, `date` から自動で作られます.

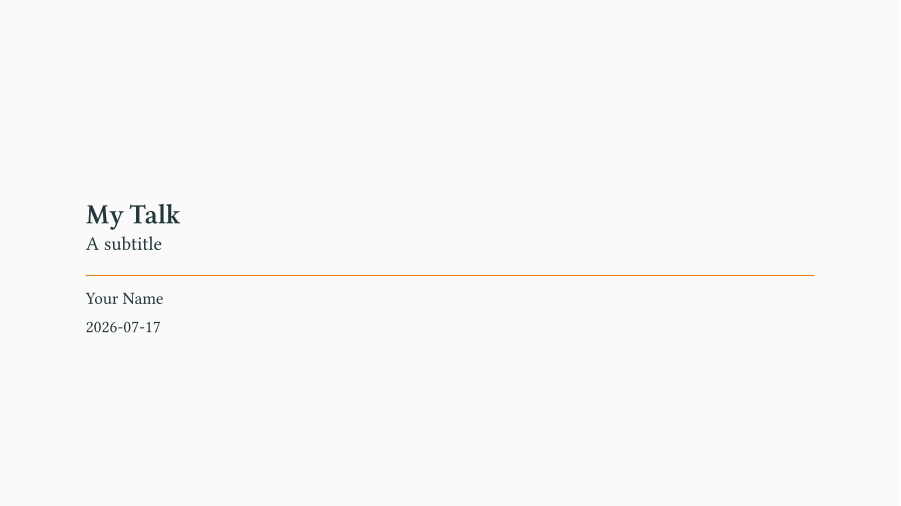


図 9.4: 上の YAML から生成されるタイトルスライド (metropolis テーマ)

Typst のコンパイルは高速なので, `quarto preview slides.qmd` としておけば, ファイルを保存するたびにほぼ即座にプレビューが更新されます. Beamer との一番の体感差はここです.

9.3.2 スライドの区切り

Reveal.js と同じく, Markdown の見出しがスライドの区切りになります.

- `#` (レベル 1 見出し): セクションの区切りスライド
- `##` (レベル 2 見出し): 通常のスライド
- 水平線 `---`: 見出しなしで新しいスライドを始める

```
# In the morning

## Getting up

- Turn off alarm
- Get out of bed

---

A follow-up slide without a heading.
```

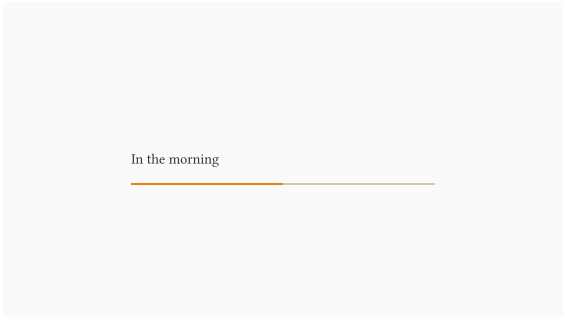


図 9.9: `# In the morning` はセクションの区切りに

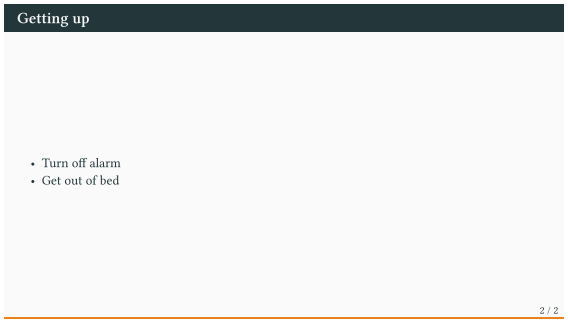


図 9.10: `## Getting up` は通常のスライドに

9.3.3　段階的な表示

Beamer の `\pause` に相当するのが, 単独の行に書く `. . .` です.

```
First this shows.

. . .

Then this appears.
```

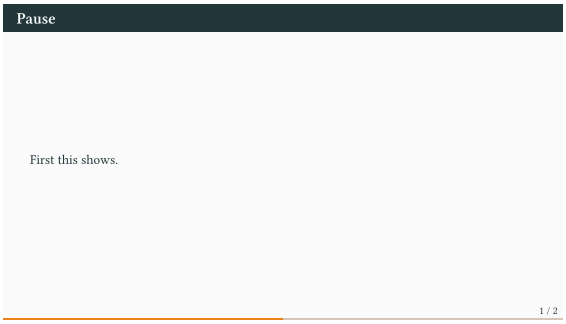


図 9.11: 1 ステップ目

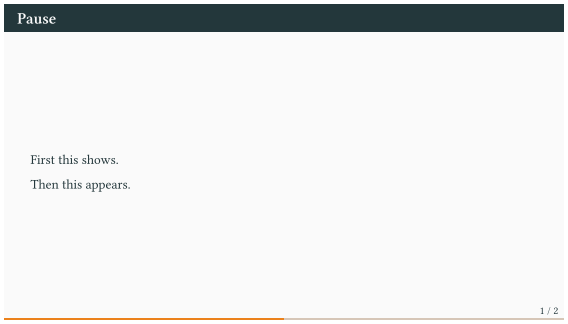


図 9.12: 2 ステップ目

箇条書きを 1 項目ずつ表示するには, `.incremental` で囲みます.

```
::: {.incremental}
- First
- Then second
- Then third
:::
```

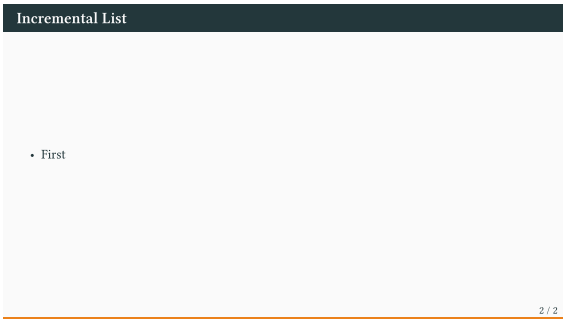


図 9.13: 1 項目ずつ現れ…

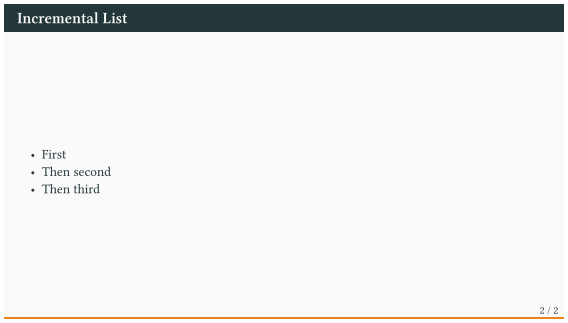


図 9.14: 最後に全項目が揃う

特定のサブスライドだけで表示・非表示を切り替えるような細かい制御には `.only` / `.uncover` が使えます (詳細は公式チュートリアルを参照). また, YAML に `handout: true` を加えると, 段階表示をすべて畳んだ配布用の PDF になります.

9.3.4 2 段組み

図と説明を並べるときは, `.columns` を使います.

```
:::: {.columns}
::: {.column width="45%"}
Left column:

- A bullet point
- Another bullet point
:::
::: {.column width="55%"}
Right column, a little wider:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$
:::
::::
```

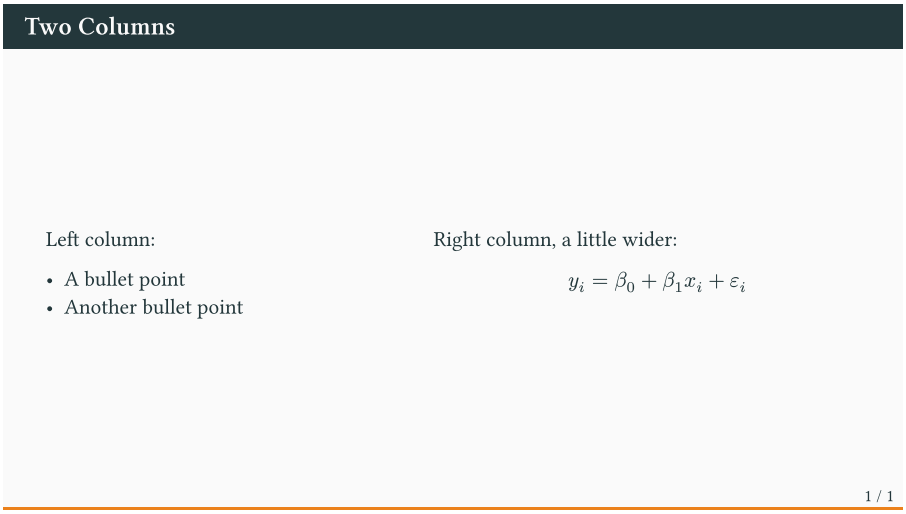


図 9.5:　`.columns` による 2 段組み

9.3.5　強調

インラインの強調用に, いくつかのクラスが用意されています.

記法	効果
`[text]{.alert}`	アクセントカラーで強調
`[text]{.fg options='fill: rgb("#5D639E")'}`	任意の色の文字
`[text]{.bg}`	マーカー風の背景色
`[[Back]{.button}](#sec-results)`	Beamer 風のジャンプボタン
`[text]{.small-cite}`	小さくグレーの引用表記

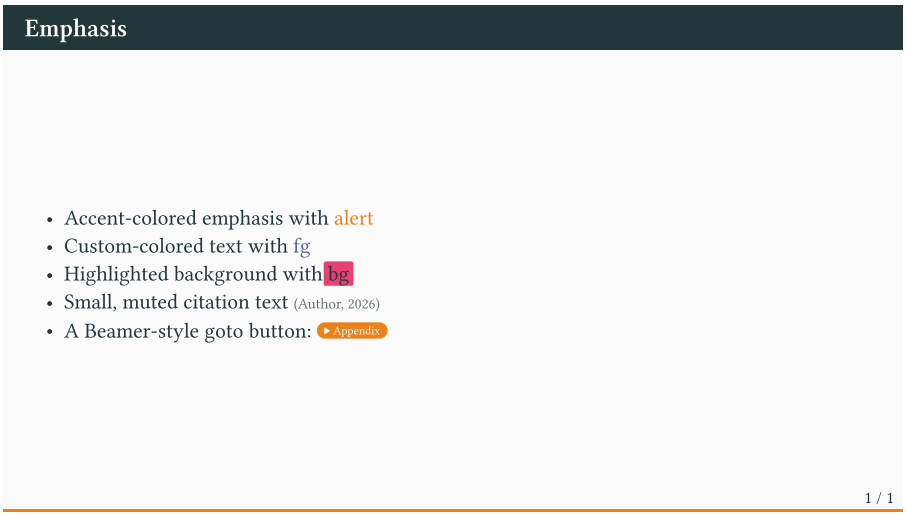


図 9.6:　強調クラスの表示例

使い方の方針は, アンチパターンの節で見たとおりです. 色はアクセントカラーを中心に絞り, `.alert` を「そのスライドで一番見てほしい一箇所」のために取っておくと, 視線誘導がよく効きます. `.button` は, 本編のスライドから補遺の詳細スライドへ飛ぶリンク (とその逆) に使います.

9.3.6 R チャンクと図表

通常の Quarto ドキュメントと同じく, R チャンクがそのまま実行でき, スライドではソースコードはデフォルトで非表示になります. 図の文字サイズをスライド用に大きくするのを忘れないようにしましょう.

````
```{r}
#| fig-width: 10
#| fig-height: 4.5
ggplot(penguins, aes(bill_len, bill_dep, color = species, shape = species)) +
  geom_point() +
  theme_minimal(base_size = 20)
```
````

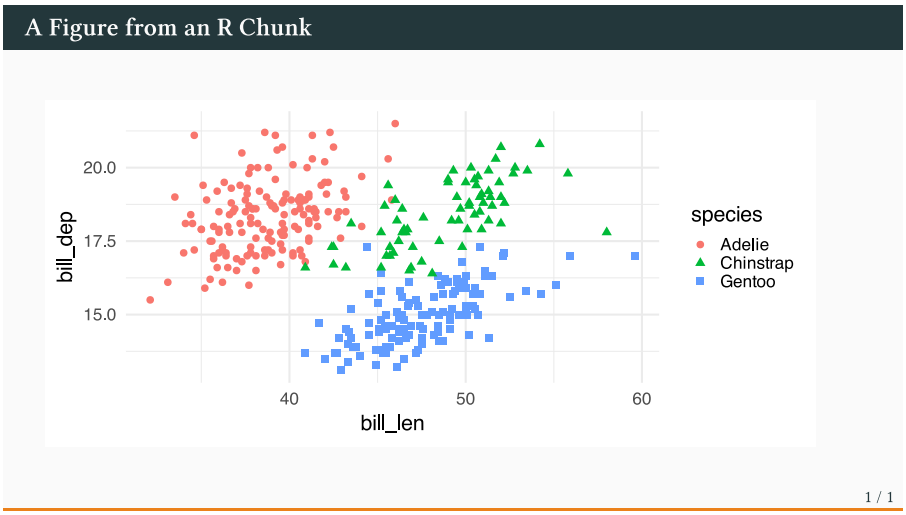


図 9.7: R チャンクから生成された図のスライド

本文中のインラインコード (`` `r ` ``) も使えるため, 推定結果の数値をスライドの文章に直接埋め込めます. データや分析が発表直前に更新されても, レンダリングし直すだけで図表と本文の数値がすべて追随します. これが冒頭で述べた, 研究報告で Quarto を使う最大の理由です. 回帰表も, アンチパターンの節で作ったような `modelsummary` + `tinytable` の表がそのまま使えます.

9.3.7 テーマとカスタマイズ

見た目は `theme:` の一行で切り替えられます. 組み込みテーマは `default`, `simple`, `metropolis`, `dewdrop`, `university`, `aqua`, `stargazer` の 7 つです.

色とフォントは, どのテーマでも YAML から調整できます.

```
accent: "107895"   # primary color (hex, no leading #)
accent2: "9a2515"  # secondary color
jet: "131516"      # body text color
sansfont: "Fira Sans"
```

プロジェクトに `_brand.yml` があれば, その `color` / `typography` の設定が自動で反映されます (本書の Web サイトの配色と同じ仕組みです).

組み込み以外の Touying テーマも使えます. 自作のテーマファイルでも, Typst Universe で配布されているテーマでも, `include-in-header` でテーマ関数を読み込み, `theme-typst` でその名前を指定するだけです. たとえば私が開発した clean テーマは次のように読み込みます.

```yaml
format:
  touying-typst:
    include-in-header:
      text: |
        #import "@preview/touying-quarto-clean:0.2.0": *
    theme-typst: clean-theme
```

clean テーマは, 所属・メールアドレス・ORCID を含む構造化された著者情報をタイトルスライドに表示できます. スライド全体の見た目は Gallery の clean のページ で確認できます.

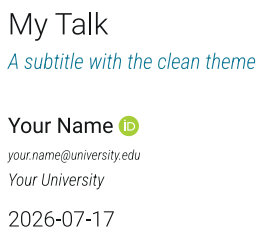


図 9.8: clean テーマのタイトルスライド

9.3.8　Appendix

発表スライドの後ろには, 質疑に備えた appendix を付けるのが通例です. `appendix` ショートコードを置くと, それ以降のスライドはページカウンタが止まり, フッターの総ページ数が本編のページ数のまま固定されます.

```
{{< appendix >}}

## Robustness Check {#sec-robustness}

Details shown only if someone asks.
```

Appendix のスライドが何十枚あっても「本編は 25 枚中 25 枚で終わり」と見えるので, 聞いている人に余計なプレッシャーを与えずに済みます.

第 10 章
文献と引用

10.1　Zotero による文献管理

Zotero を使うと, 文献情報 (著者, 年, タイトル, DOI など) を整理して, BibTeX 形式として書き出せます.

10.1.1　BibTeX とは

BibTeX は, 文献情報を管理するためのフォーマットの一つです. 文献ごとに `@article` や `@book` などのエントリータイプを指定し, キーと値のペアで書誌情報を記述します. 例えば, 本 1 冊 (`@book`) と論文 1 本 (`@article`) を BibTeX で表すと次のようになります. 以下ではこれを `references.bib` というファイル名で保存したものとして扱います.

```
@book{acemoglu2012,
  author    = {Acemoglu, Daron and Robinson, James A.},
  title     = {Why Nations Fail: The Origins of Power, Prosperity, and Poverty},
  year      = {2012},
  publisher = {Crown Publishers},
  address   = {New York}
}

@article{halljones1999,
  author  = {Hall, Robert E. and Jones, Charles I.},
  title   = {Why Do Some Countries Produce So Much More Output Per Worker Than Others?},
  journal = {The Quarterly Journal of Economics},
  year    = {1999},
  volume  = {114},
  number  = {1},
  pages   = {83--116}
}
```

エントリータイプによって必要なフィールドが異なる点に注目してください. `@book` は `publisher` を, `@article` は `journal` や `volume` を持ちます. 元々は LaTeX で文献管理するためのフォーマットですが, Typst でも BibTeX 形式のファイルを読み込んで引用に利用できます.

10.1.2　Zotero による BibTeX ファイルの作成

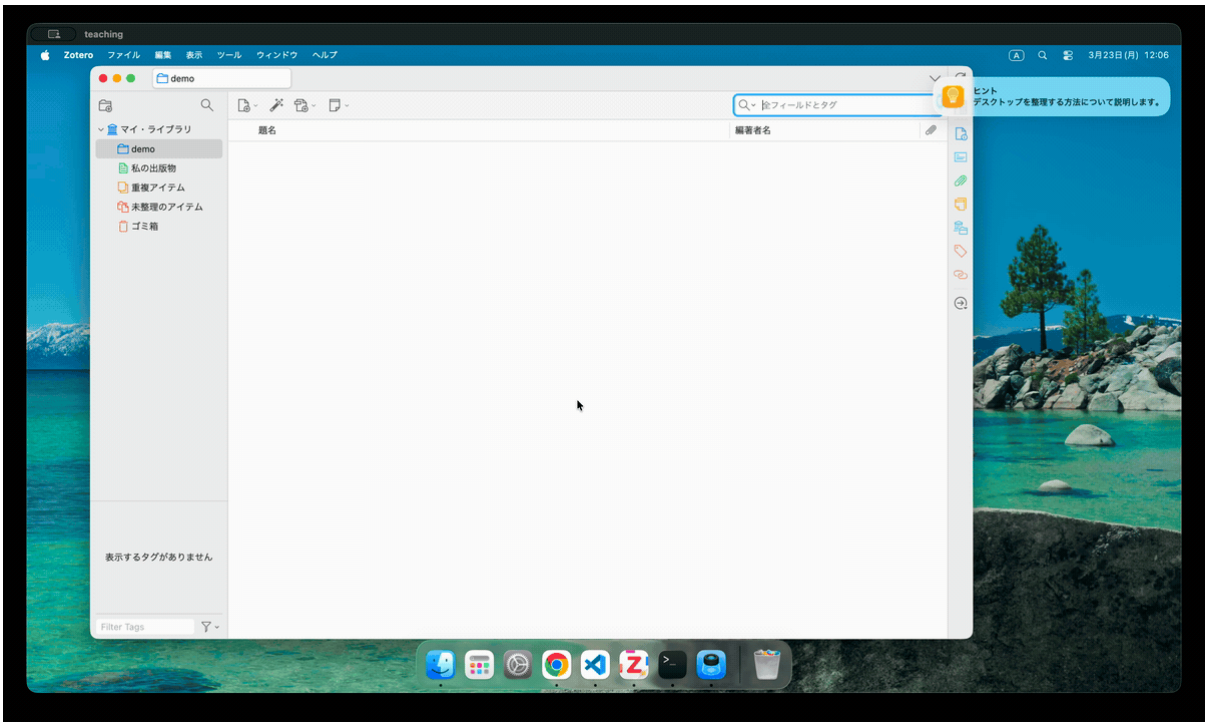
Zotero から BibTeX ファイルを作る手順は次の通りです.

Zotero → BiBTeX

1. コレクションを好きな名前で作る
2. Zotero に文献を入れる (ブラウザ拡張, ドラッグ & ドロップ, DOI など)
3. メタデータを確認・修正する (著者名の表記ゆれ, 年, 雑誌名など)
4. `references.bib` にエクスポートする (BibTeX 形式)

なお, Zotero 拡張の Better BibTeX を導入することで, BiBTeX ファイルを自動で更新するなどの便利な機能が使えるようになります.

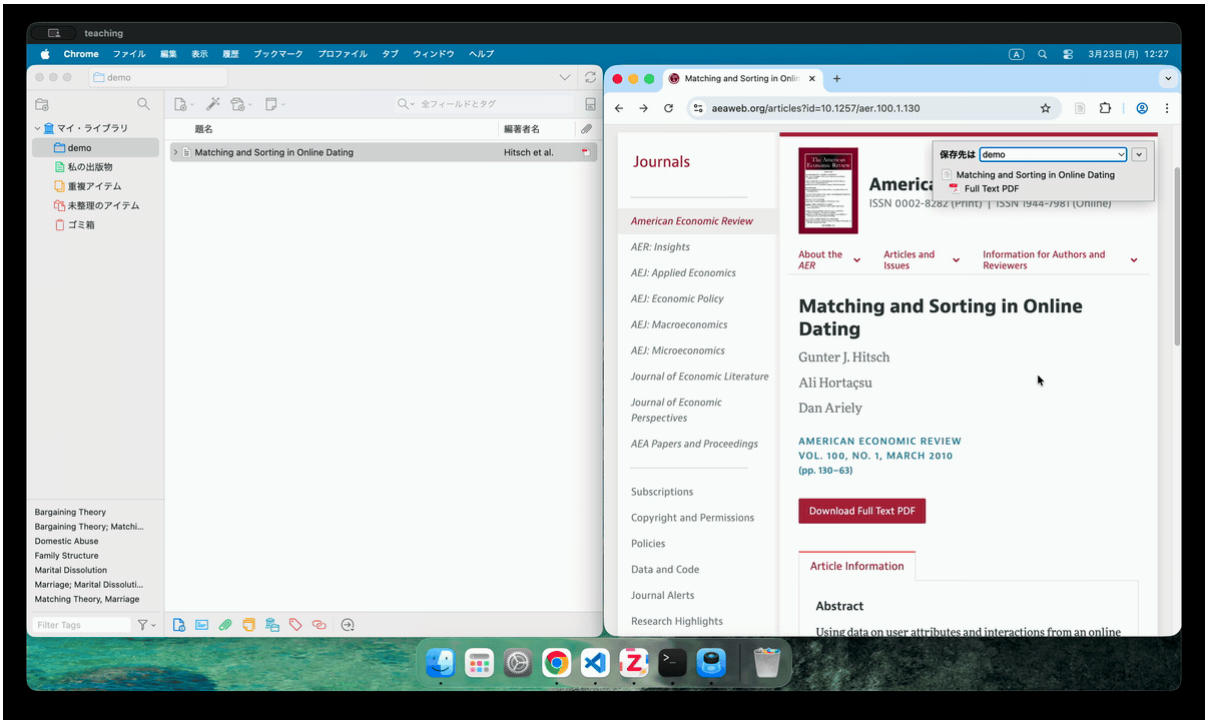
Step 1. コレクションを作る



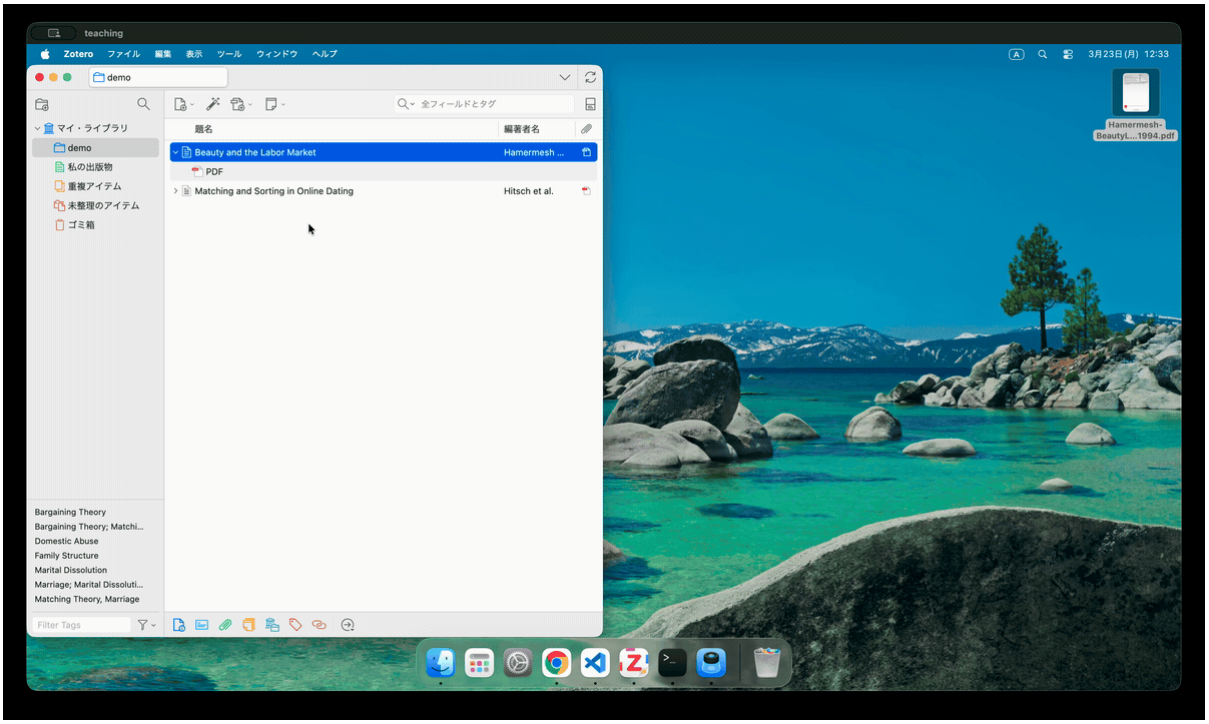
Zotero では全ての文献は「ライブラリ」に属します. ライブラリには複数の「コレクション」を作ることができ, 基本的には文献を管理したいプロジェクト (レポート, 論文) ごとに作成します. コレクションはフォルダのようなもので, 同じ文献を複数のコレクションに入れることもできます.

Step 2. Zotero に文献を入れる

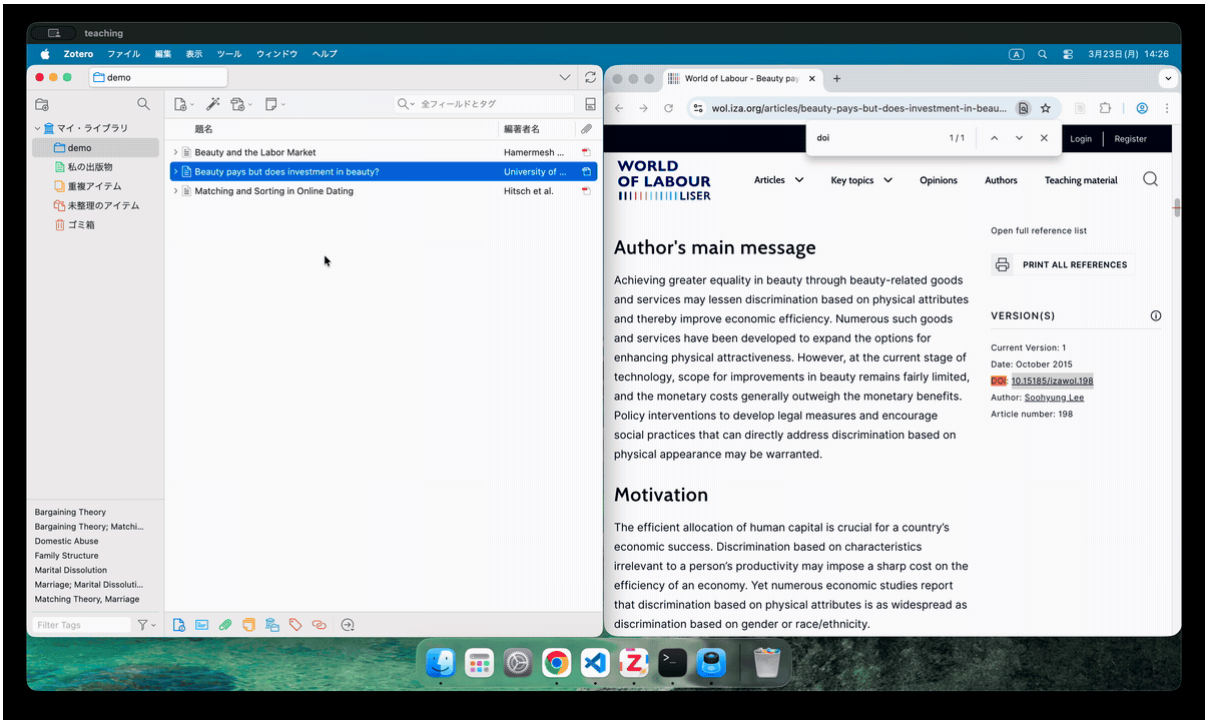
10.2　ブラウザ拡張



10.3　ドラッグ & ドロップ



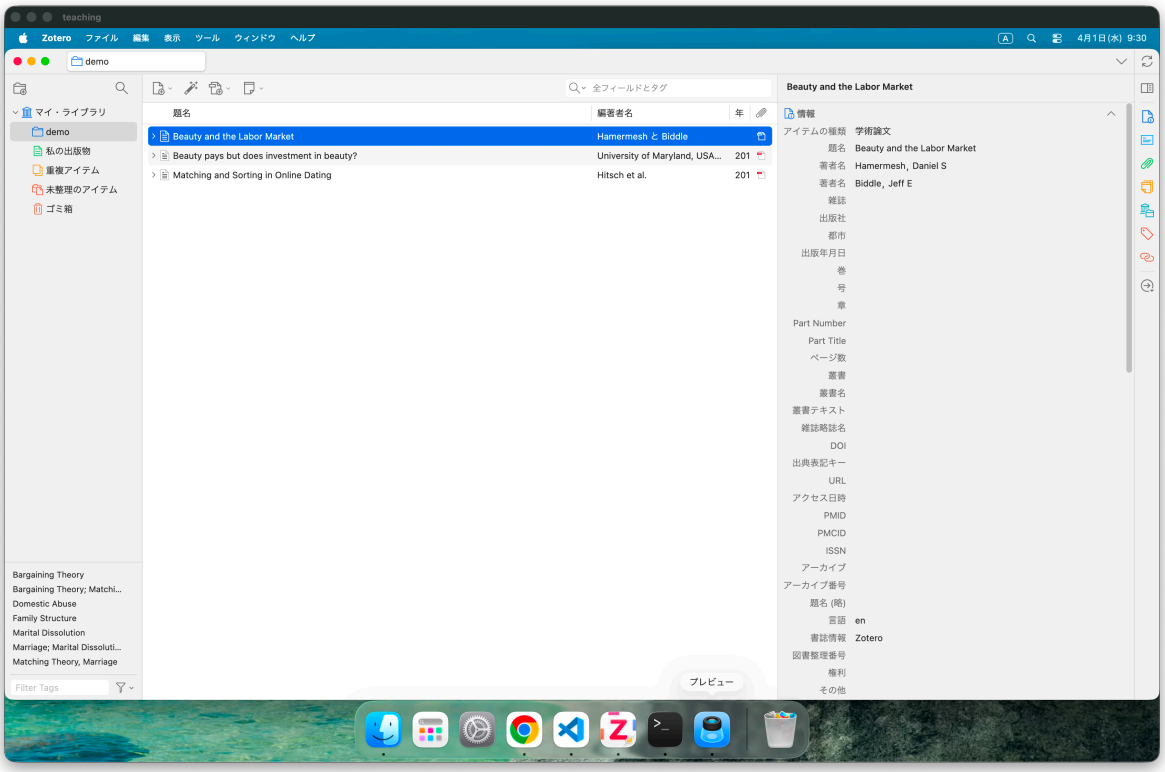
10.4　DOI



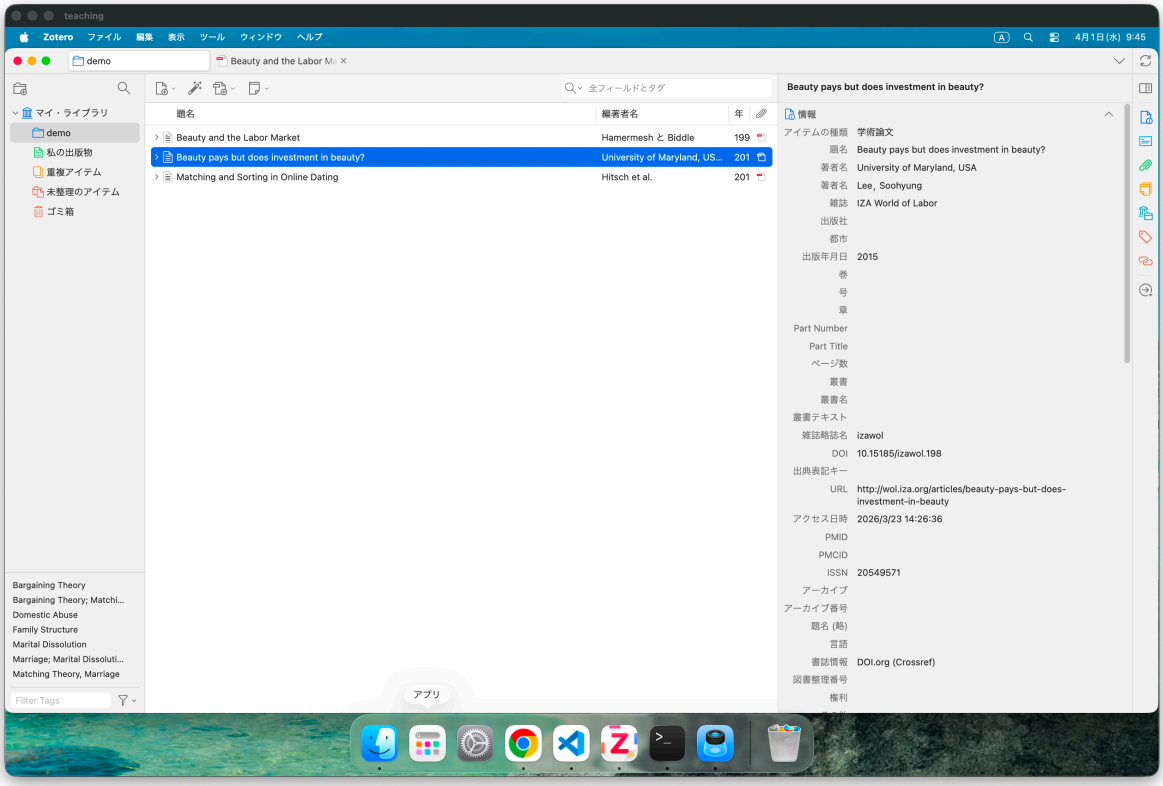
メタデータの取得

Zotero は DOI や URL から自動で書誌情報 (メタデータ) を取得しますが, 全ての情報が正確に入るわけではありません. 基本的には DOI という論文ごとのユニークな識別子が発見できた場合, CrossRef などのデータベースから情報を取得しています. しかし, これでも間違いがあったり, DOI がない資料があったりするため, メタデータは必ず確認して修正する必要があります.

Step 3. メタデータを確認・修正する

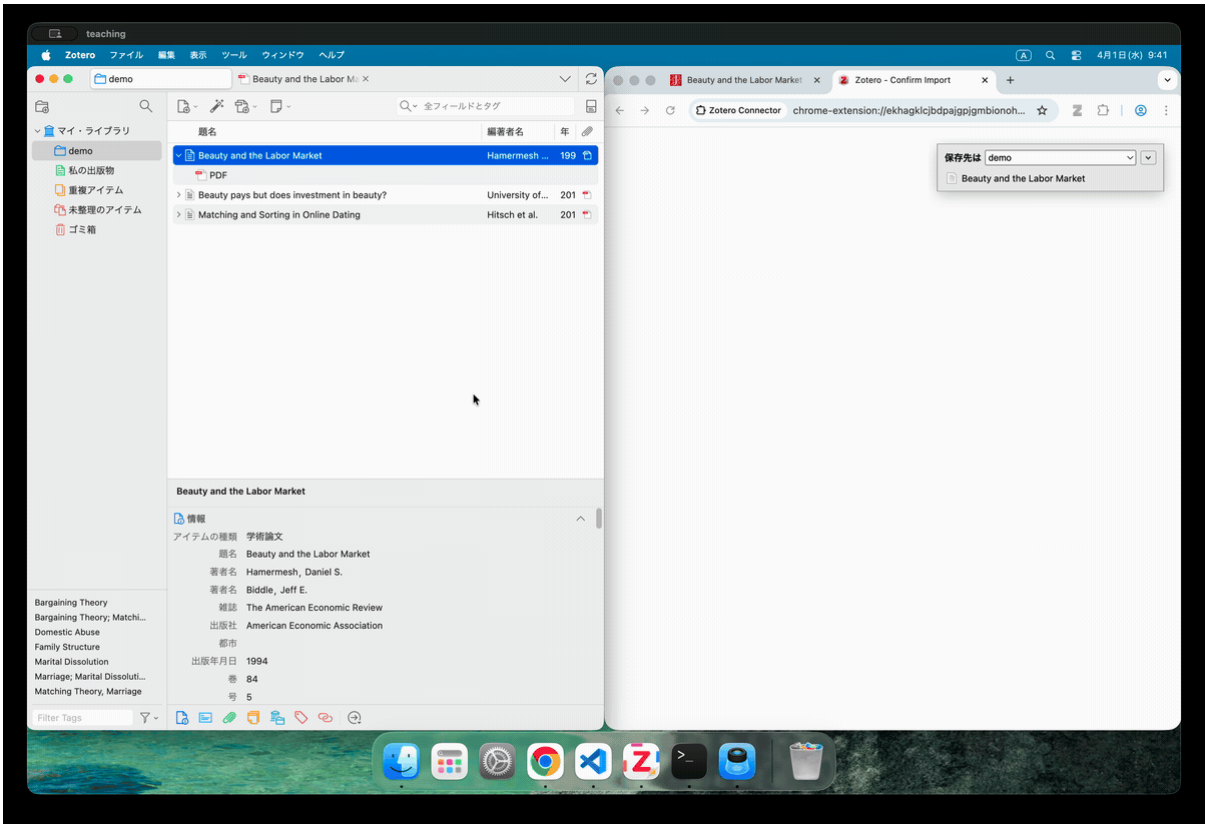


ドラッグ & ドロップで入れた “Beauty and Labor Market” は, タイトルと著者名は正しく入っていますが, その他の情報が空欄になっています. 書誌情報のところを見ると “Zotero” となっているので, これは Zotero が DOI などの情報を自動で取得できなかったことを示しています.



上の例は，CrossRef から取得したデータでも間違いがあったケースです．著者名に “University of Maryland, USA” という著者の所属する機関名が入ってしまっています．ここでは，RIS ファイルによる修正方法と，Zotero 上で直接修正する方法の両方を紹介します．

10.5　RIS ファイル

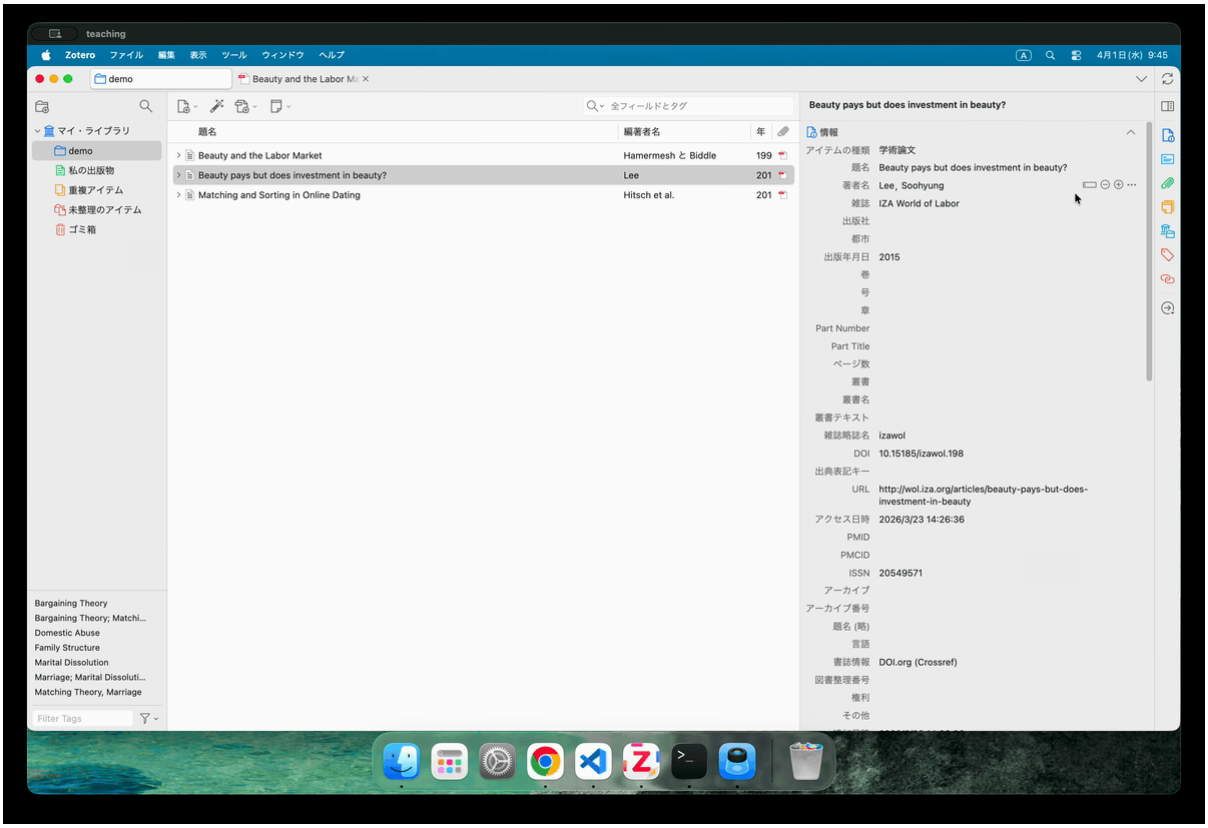


RIS ファイルとは, 文献情報を管理するためのファイル形式の一つです. 多くの出版社やデータベースが, 文献情報を RIS 形式でエクスポートする機能を提供しています. 本来は

1. RIS ファイルをダウンロード
2. Zotero にインポート

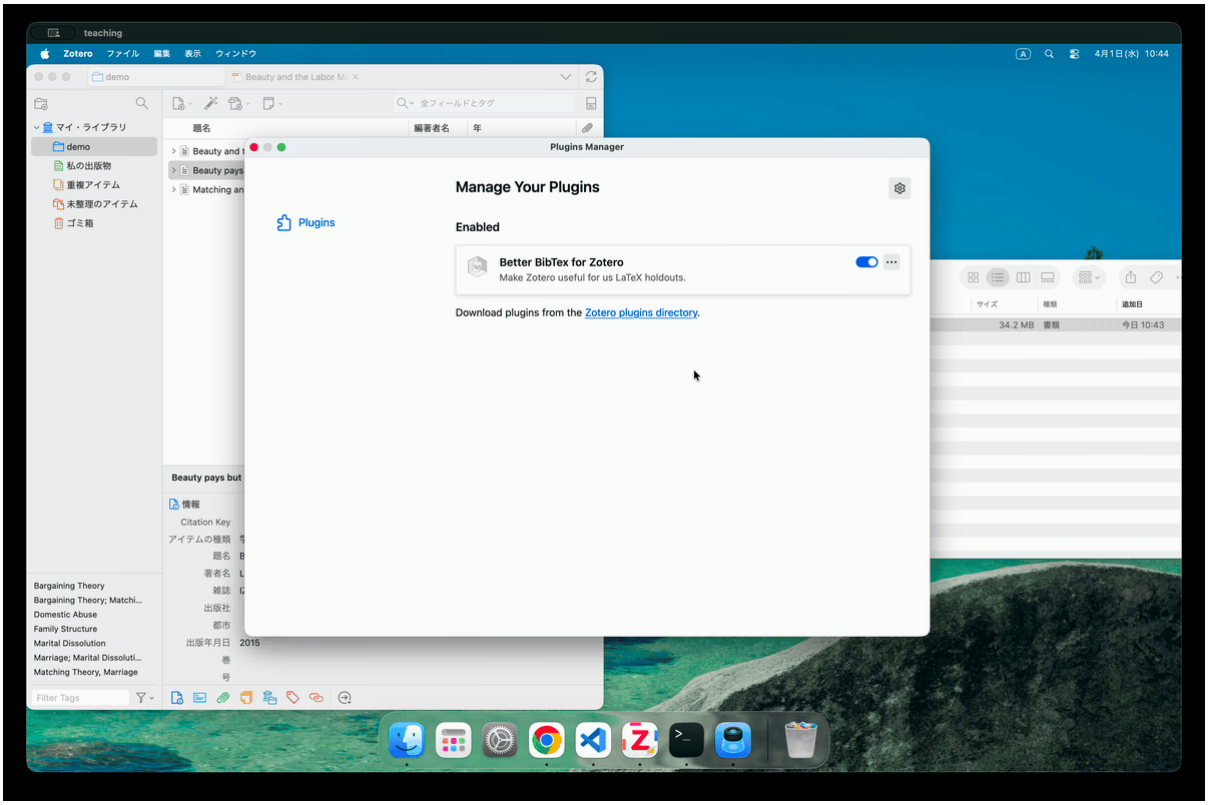
という手順で書誌情報を取り込むのですが, Zotero のブラウザ拡張では, RIS のダウンロードの際に自動で Zotero にインポートする機能もあります.

10.6　手動

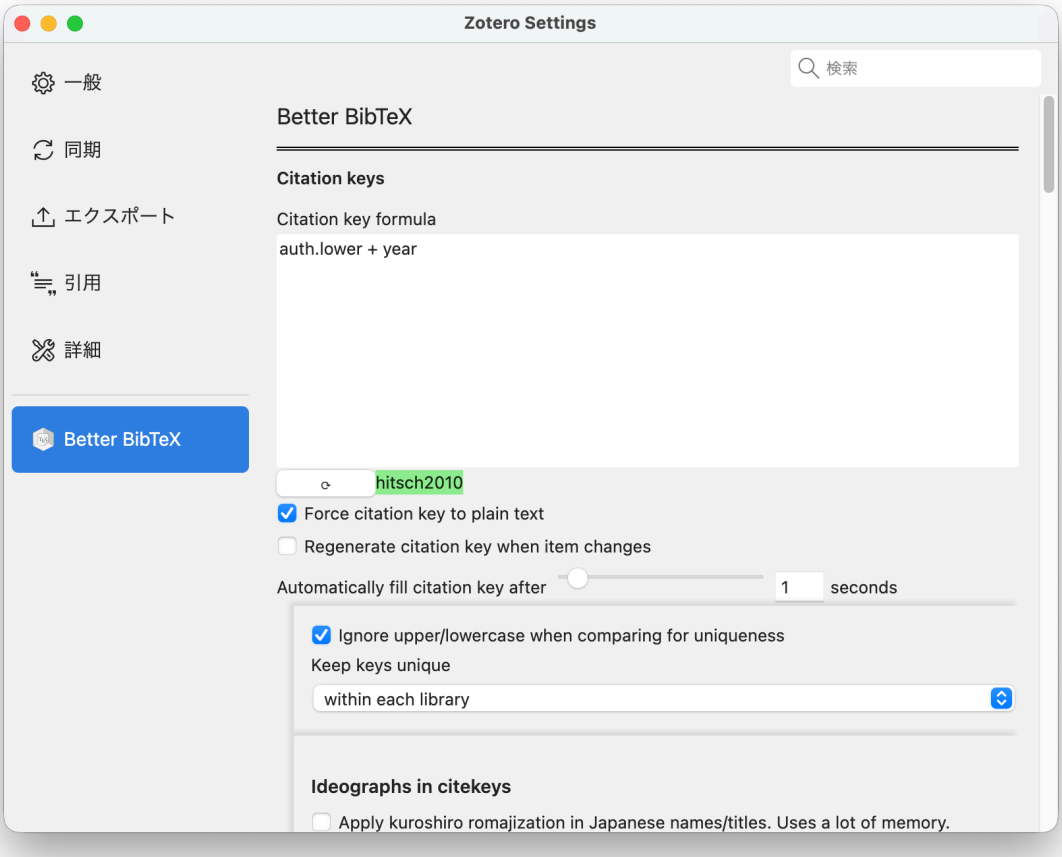


Zotero の画面上で直接書誌情報を修正することもできます. しかし, 手入力はミスが起こりやすいので, 可能な限り出版社やデータベースから正しい書誌情報を取得し, 最終手段として手入力で修正するのが望ましいです.

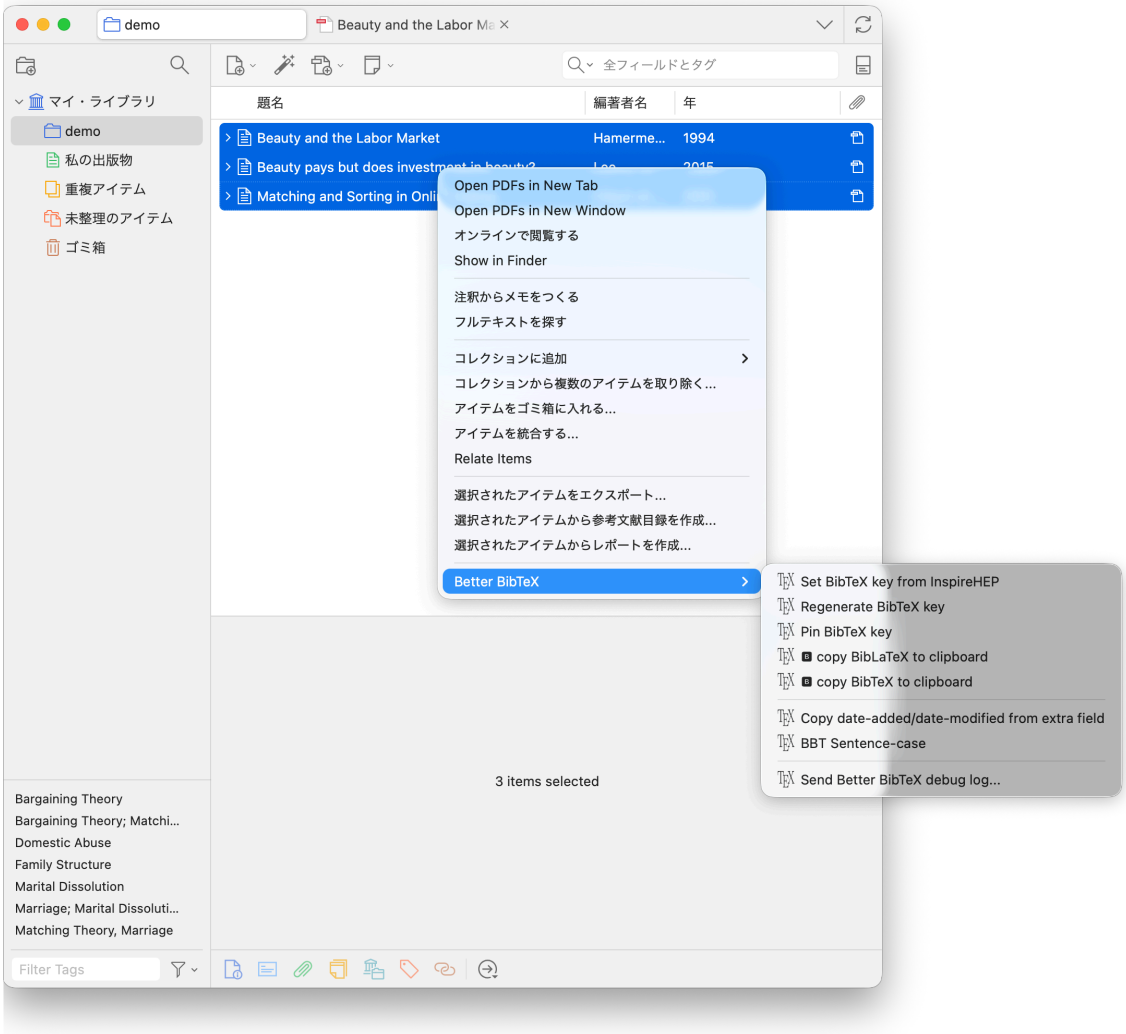
Step 4. `references.bib` にエクスポートする



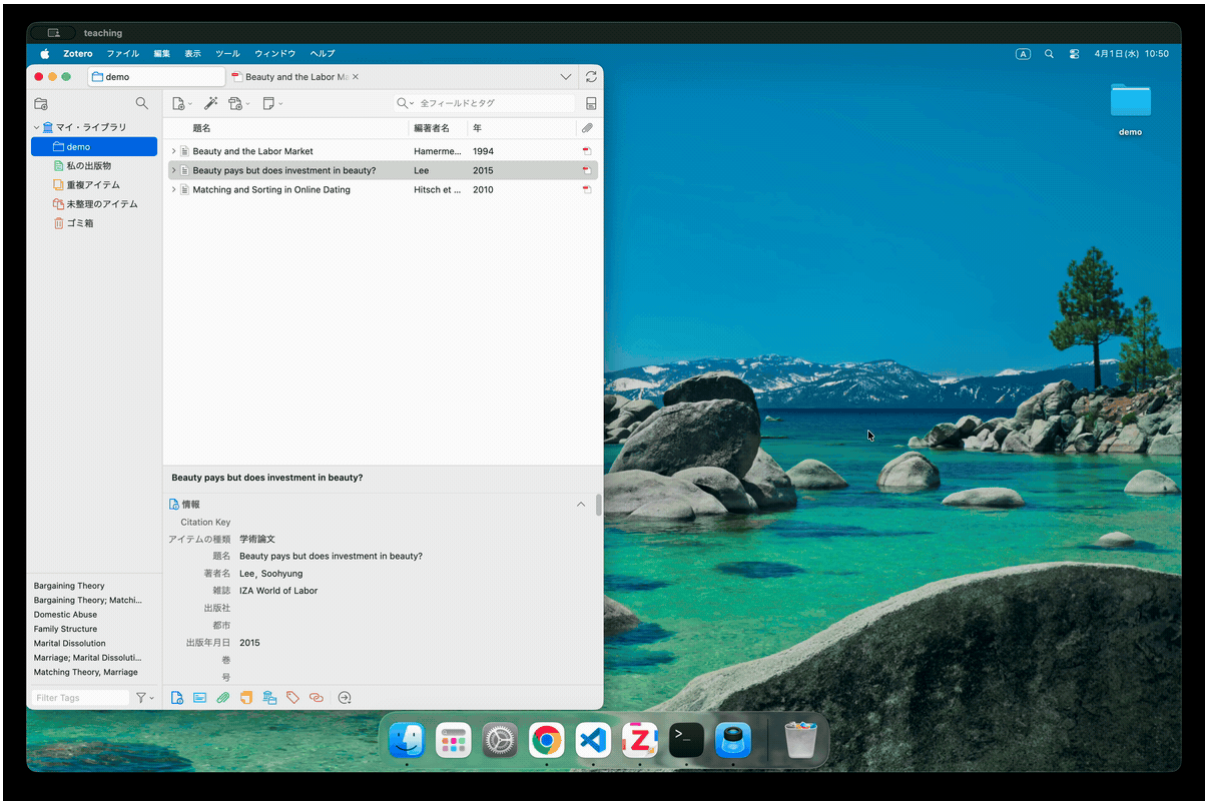
Better BibTeX は, Zotero の拡張機能で, BiBTeX 形式でのエクスポートを強化するものです. これを入れると, Zotero のコレクションを BiBTeX ファイルにエクスポートする際に, ファイルを自動で更新するなどの便利な機能が使えるようになります. インストールは, GitHub のリリースページからダウンロードした `.xpi` ファイルを, Zotero の「ツール」→「プラグイン」→「Install Plugin from File」からインストールするだけです.



Better BibTeX を入れると, Zotero の文献ごとに “citekey” という識別子が自動で割り当てられます. これが, Quarto の `[@...]` で引用するときのキーになります. 個人的には, `auth.lower + year` 形式 (例: `knuth1984`) が分かりやすいと思いますが, 好みに合わせて変更することもできます.



Better BibTeX が設定されていると, 自動的に citekey が生成されるようになりますが, 既に文献を入れてしまった後に Better BibTeX を入れた場合は, Zotero の「ツール」→「Better BibTeX」→「Regenerate BibTeX keys」で既存の文献にも citekey を割り当てることができます.



コレクション名を右クリックすることで，そのコレクションに入っている文献を BiBTeX 形式でエクスポートできます. 出力形式は “Better BibTeX” を選び，ファイル名は `references.bib` として，Quarto プロジェクトのルートディレクトリに保存してください. この時，「Keep updated」をチェックしておくと，Zotero のコレクションに文献を追加・削除したときに `references.bib` が自動で更新されるようになります.

10.7　LaTeX による引用

`references.bib` を用いることで，LaTeX などの文書作成システムで引用が可能になります. 経済学の文献は，ほぼ例外なく著者・年方式 (author-year) です.

> **i 著者・年方式**
>
> 著者・年方式とは，引用を番号 ([1], [2] など) ではなく，著者の姓と出版年で示す方式です. 使われ方は大きく 2 通りあります.
>
> 1.「Acemoglu and Robinson (2012) は …」
> 2.「… が知られています (Acemoglu and Robinson 2012).」
>
> 著者が 3 名以上のときは，慣例として「Acemoglu et al. (2012)」のように筆頭著者と et al. に省略します.

LaTeX で著者・年の引用を扱うには，大きく 2 つの選択肢があります. 一つは古くから使われてきた `natbib` パッケージ (バックエンドは BibTeX)，もう一つはより新しい `biblatex` パッケージ (バック

エンドは Biber) です. どちらも `.bib` ファイルをそのまま読めますが, 引用コマンドの書き方が異なります.

10.7.1 `natbib` + BibTeX

`natbib` は経済学で最も広く使われている方式です. 多くの学術誌が `natbib` 用のスタイルファイル (`.bst`) を配布しており (例: AEA の `aea.bst`, Econometrica の `ecta.bst`), 投稿規定もこれを前提にしていることが多いためです. 手元で AEA スタイルを試すだけなら, TeX Live に含まれる `econ-bst` パッケージの `econ-aea.bst` が手軽です (`tlmgr install econ-bst` で入ります).

`natbib` の引用コマンドは, 著者名を文の主語として地の文に組み込む `\citet` (textual) と, 文末などで括弧に入れる `\citep` (parenthetical) の使い分けが基本です. このほか, 著者名だけの `\citeauthor`, 年だけの `\citeyear`, 括弧なしの `\citealt` などがあります.

```
\documentclass[11pt]{article}
% The page is sized to the content so the figure needs no cropping.
\usepackage[paperwidth=17cm, paperheight=8.2cm, margin=1cm]{geometry}
\usepackage[round]{natbib}      % round → parentheses are ( )
\pagestyle{empty}
\setlength{\parindent}{0pt}

\begin{document}

\citet{acemoglu2012} argue that institutions are a fundamental cause
of long-run growth. This view builds on earlier work
\citep{halljones1999}, and is developed in detail elsewhere
\citep[see][chap.~2]{acemoglu2012}.

\bibliographystyle{econ-aea}    % AEA style provided by econ-bst
\bibliography{references}             % read references.bib

\end{document}
```

Acemoglu and Robinson (2012) argue that institutions are a fundamental cause of long-run growth. This view builds on earlier work (Hall and Jones, 1999), and is developed in detail elsewhere (see Acemoglu and Robinson, 2012, chap. 2).

References

Acemoglu, Daron, and James A. Robinson. 2012. *Why Nations Fail: The Origins of Power, Prosperity, and Poverty.* New York: Crown Publishers.

Hall, Robert E., and Charles I. Jones. 1999. "Why Do Some Countries Produce So Much More Output Per Worker Than Others?" *The Quarterly Journal of Economics* 114 (1): 83–116.

図 10.1: natbib + BibTeX による引用と文献リスト

コンパイルした結果が 図 10.1 です. `\citet` が地の文の「Acemoglu and Robinson (2012)」に, `\citep` が括弧内の「(Hall and Jones, 1999)」に, 後置注が「(see …, chap. 2)」になっている点に注目してください.

10.7.2　`biblatex` + Biber

`biblatex` は, 引用スタイルを LaTeX マクロとして柔軟に設定できる新しい仕組みで, Unicode (日本語やアクセント付き文字) をそのまま扱えるのが大きな利点です. バックエンドの Biber が `.bib` を解釈します. 著者・年方式は組み込みの `style=authoryear` でも出せますが, 経済学では Chicago の著者・年スタイルを実装した `biblatex-chicago` パッケージ (`tlmgr install biblatex-chicago`) を使うと, AEA に近い体裁になります.

ここで一点注意があります. biblatex には, natbib の `econ-aea.bst` のような AEA 専用スタイルが存在しません. もっとも近いのが Chicago の `biblatex-chicago` で, AEA のハウススタイル自体が Chicago 著者・年方式をベースにしているため見た目はほぼ揃いますが, 厳密に AEA の体裁が要求される投稿では結局 natbib + `.bst` のほうが確実です.

引用コマンドは biblatex 共通で, `\textcite` が `natbib` の `\citet` に, `\parencite` が `\citep` に対応します (どちらのパッケージでも `natbib=true` オプションを足せば `\citet` / `\citep` という慣れた名前も使えます). 著者名の最大表示数 (`maxcitenames`) や同一著者・同一年の区別 (2012a, 2012b) なども細かく制御できます.

```
\documentclass[11pt]{article}
% The page is sized to the content so the figure needs no cropping.
\usepackage[paperwidth=17cm, paperheight=8.2cm, margin=1cm]{geometry}
\usepackage[authordate, backend=biber]{biblatex-chicago}
\addbibresource{references.bib} % include the extension
\pagestyle{empty}
\setlength{\parindent}{0pt}
```

```
\begin{document}

\textcite{acemoglu2012} argue that institutions are a fundamental cause
of long-run growth. This view builds on earlier work
\parencite{halljones1999}, and is developed in detail elsewhere
\parencite[see][chap.~2]{acemoglu2012}.

\printbibliography              % print the reference list

\end{document}
```

Acemoglu and Robinson (2012) argue that institutions are a fundamental cause of long-run growth. This view builds on earlier work (Hall and Jones 1999), and is developed in detail elsewhere (see Acemoglu and Robinson 2012, chap. 2).

References

Acemoglu, Daron, and James A. Robinson. 2012. *Why Nations Fail: The Origins of Power, Prosperity, and Poverty.* New York: Crown Publishers.

Hall, Robert E., and Charles I. Jones. 1999. “Why Do Some Countries Produce So Much More Output Per Worker Than Others?” *The Quarterly Journal of Economics* 114 (1): 83–116.

図 10.2: biblatex + Biber による引用と文献リスト

コンパイルした結果が 図 10.2 です (スタイルは `biblatex-chicago` の author-date). `natbib` + `econ-aea` 版とほぼ同じ出力になっているのが分かります.

10.7.3　コマンドの対応

両者の主な引用コマンドと出力の対応をまとめると次のようになります.

やりたいこと	`natbib`	`biblatex`	出力例
地の文に組み込む	`\citet`	`\textcite`	Acemoglu and Robinson (2012)
括弧に入れる	`\citep`	`\parencite`	(Acemoglu and Robinson 2012)
著者名だけ	`\citeauthor`	`\citeauthor`	Acemoglu and Robinson
年だけ	`\citeyear`	`\citeyear`	2012

10.7.4　どちらを使うべきか

結論から言えば, 経済学では `natbib` + BibTeX を既定にしておくのが無難です. 多くのジャーナルが `natbib` 用の `.bst` を配布していること, そして投稿時にトラブルが起きにくいことが理由です. `biblatex` + Biber は Unicode 対応やスタイルの柔軟さで優れており, 凝った書式を組みたいときには魅力的ですが, その柔軟さが本当に必要になる場面は経済学の論文ではそう多くありません. 迷ったら `natbib` を選び, `biblatex` は明確な必要が出てから検討すれば十分です. いずれにせよ, 引用元の `references.bib` を Zotero で一元管理しておけば, あとからエンジンを乗り換えても問題は起きません.

> arXiv 投稿と `.bbl`
>
> `natbib` を勧めるもう一つの理由が, arXiv への投稿です. arXiv は投稿された `.tex` をコンパイルしますが, bibtex も biber も実行しません. 代わりに, 一緒にアップロードした `.bbl` ファイル (文献リストを組版した中間ファイル) をそのまま使います. ここで両者に差が出ます. BibTeX が生成する `.bbl` は素朴なテキストでバージョンに依存しないため, arXiv 上でもそのまま通ります. 一方 `biblatex` の `.bbl` は biblatex/biber のバージョンと強く結合しており, arXiv 側の biblatex が手元と違うバージョンだと読めずにエラーになることがあります. どちらも `.bbl` を同梱すれば投稿自体は可能ですが, この一点でも `natbib` + BibTeX のほうが安全です.

10.8　Quarto による引用

Quarto では LaTeX のように `\citet` / `\citep` を使わず, Markdown の `@` 記法で引用します. `@key` が地の文に入る形 (`\citet` 相当), `[@key]` が括弧に入る形 (`\citep` 相当) で, ページや章は `[@key, chap. 2]` のように後置します. スタイルを指定しなければ citeproc の既定である Chicago 著者・年方式になり, 経済学で標準的な体裁がそのまま得られます[29]. `link-citations: true` を指定すると, 本文の引用から対応する文献リストの項目へのリンクが張られます.

`example.qmd` のフロントマター (設定) は次の通りです.

```yaml
---
bibliography: references.bib
reference-section-title: References
link-citations: true
format:
  pdf:
    documentclass: article
    pagestyle: empty
```

[29]別のスタイルにしたいときは `csl` に CSL (Citation Style Language) ファイルを渡します (例: `econometrica.csl`, `the-quarterly-journal-of-economics.csl`; Zotero Style Repository から入手できます). なお AEA/AER 専用の CSL は公式リポジトリには無く, AEA のハウススタイル自体が Chicago 著者・年方式ベースなので, 既定のままで実質 AEA に近い体裁になります. LaTeX の `bibliographystyle` に当たる `bibliography-style` のようなキーは citeproc では使われません.

```
    geometry: [paperwidth=17cm, paperheight=8cm, margin=1cm]
---
```

続く本文では, `@` 記法で引用します.

```
@acemoglu2012 argue that institutions are a fundamental cause of
long-run growth. This view builds on earlier work [@halljones1999],
and is developed in detail elsewhere [see @acemoglu2012, chap. 2].
```

Acemoglu and Robinson (2012) argue that institutions are a fundamental cause of long-run growth. This view builds on earlier work (Hall and Jones 1999), and is developed in detail elsewhere (see Acemoglu and Robinson 2012, chap. 2).

References

Acemoglu, Daron, and James A. Robinson. 2012. *Why Nations Fail: The Origins of Power, Prosperity, and Poverty*. Crown Publishers.

Hall, Robert E., and Charles I. Jones. 1999. "Why Do Some Countries Produce so Much More Output Per Worker Than Others?" *The Quarterly Journal of Economics* 114 (1): 83–116.

図 10.3: Quarto による引用と文献リスト

これをレンダリングした結果が 図 10.3 です. natbib + `econ-aea` や biblatex-chicago の例とほぼ同じ出力が得られます. `link-citations: true` を入れたため, 本文の引用 (図中の色付き部分) が文献リストへのリンクになっています.

i Quarto の citeproc による引用処理

Quarto は引用の処理に citeproc を使っています. citeproc は, CSL (Citation Style Language) という XML 形式のスタイル定義を読み込み, 文献リストの整形や引用の組版を行う JavaScript ライブラリです. Quarto は citeproc を組み込んでおり, `.bib` ファイルを読み込んで引用を処理します. citeproc は LaTeX のように `.bbl` を生成するわけではなく, 文献リストも本文中の引用も, 組版済みの文字列として直接埋め込みます.

```
Acemoglu and Robinson (\citeproc{ref-acemoglu2012}{2012}) argue that
institutions are a fundamental cause of long-run growth. This view
builds on earlier work (\citeproc{ref-halljones1999}{Hall and Jones
1999}), and is developed in detail elsewhere (see
\citeproc{ref-acemoglu2012}{Acemoglu and Robinson 2012, chap. 2}).

\section*{References}\label{bibliography}

\begin{CSLReferences}{1}{1}
\bibitem[\citeproctext]{ref-acemoglu2012}
Acemoglu, Daron, and James A. Robinson. 2012. \emph{Why Nations Fail:
The Origins of Power, Prosperity, and Poverty}. Crown Publishers.

\bibitem[\citeproctext]{ref-halljones1999}
Hall, Robert E., and Charles I. Jones. 1999. {``Why Do Some Countries
Produce so Much More Output Per Worker Than Others?''} \emph{The
Quarterly Journal of Economics} 114 (1): 83--116.
\end{CSLReferences}
```

`\citet` や `\bibliography` といったコマンドは一切登場しません. 引用は `\citeproc{ref-acemoglu2012}{ … }` の形で, 組版済みの文字列を埋め込みつつ, 対応する文献エントリ (`\bibitem[ … ]{ref-acemoglu2012}`) へのリンクを張っています (`link-citations: true` の効果です). 文献リストも citeproc が整形したテキストが並ぶだけです. したがって, `natbib` や `biblatex` が走る必要がなく, arXiv に投稿する際も `.bbl` を同包する必要はありません.

10.8.1　Typst バックエンド

Quarto は PDF を LaTeX ではなく Typst で組むこともできます (`format: typst`). その既定スタイルは IEEE の番号引用 ([1], [2] …) なので, 経済学の著者・年方式にするには, `bibliographystyle` に Typst 組み込みのスタイル名 `chicago-author-date` を渡します.

```
---
bibliography: references.bib
```

```
bibliographystyle: chicago-author-date
link-citations: true
format: typst
---
```

Acemoglu and Robinson (2012) argue that institutions are a fundamental cause of long-run growth. This view builds on earlier work (Hall and Jones 1999), and is developed in detail elsewhere (Acemoglu and Robinson 2012, chap. 2).

Bibliography

Acemoglu, Daron, and James A. Robinson. 2012. *Why Nations Fail: The Origins of Power, Prosperity, And Poverty*. Crown Publishers.

Hall, Robert E., and Charles I. Jones. 1999. "Why Do Some Countries Produce so Much More Output Per Worker Than Others?." *The Quarterly Journal of Economics* 114 (1): 83–116.

図 10.4: Typst バックエンドでの引用と文献リスト

本文は 図 10.3 の例と同じです. コンパイルした結果が 図 10.4 で, LaTeX 経由の出力 (図 10.3) とほぼ同じ著者・年方式になります (Typst の規定の参考文献のタイトルは "Bibliography" になります).

第 11 章
パイプライン

{targets} は, 研究のワークフローを構築する R のパッケージです. 最大の特徴は, データ, 関数, 結果を R のオブジェクトとしてその依存関係を管理し, 上流のオブジェクトが変更されたときに, それに依存する下流のオブジェクトを自動的に再計算してくれることです. これにより, 再現性を保ち続けたまま研究を進めることができます.

さらに Quarto と組み合わせることで, 研究途中のレポートやスライド, 論文の執筆といった, 研究の全てのワークフローを一つのパイプラインで管理することができます. この章では Quarto + {targets} を用いた研究のワークフローを解説します. 実際に私が研究で用いており, 研究の進捗に合わせてステップを進めていくことができるようになっています. また, 研究とは直線的なものではなく, 試行錯誤を繰り返しながら道を見つけていくものだと思います. そのため, 論文を執筆し始める前の試行錯誤の段階に, ある程度の自由度を持たせています.

コードは kazuyanagimoto/quarto-research-blog にあります. コードやディレクトリ構成など参考にしてください.

11.1 ワークフロー

1. _targets.R の初期設定を行う
2. データクリーニングを定義する R/tar_data.R
3. ウェブサイトを設定する _quarto.yml
4. データ分析やモデルの試行錯誤 playground/yymmdd_*/index.qmd
5. 途中結果をスライドにまとめる slide/yymmdd_*/index.qmd
6. 3-5 を繰り返す. 主要な結果をパイプラインに組み込む R/tar_fact.R, R/tar_model.R
7. 最終的な結果で論文を執筆する manuscript/*.qmd

論文を執筆しおわった頃には, 以下のようなパイプラインができあがっているはずです.

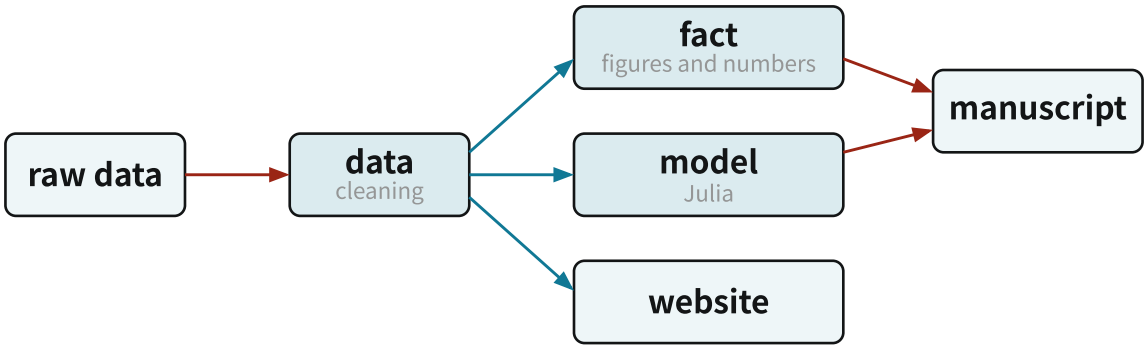


図 11.1: 論文を書き終えた頃にできあがっているパイプライン

次節ではワークフローの各部分を解説していきますが，その前に {targets} の基本的な使い方を解説します．

11.2 {targets} の基本

{targets} の基本を学ぶには公式のチュートリアルが良いですが，実用上は {targets} を拡張した {tarchetypes} の文法を使うことが多いです．そのため，ここでは {tarchetypes} の文法に基づいて最低限の使い方を解説します．この {targets} から {tarchetypes} への移行に関しては，このチュートリアルが参考になりました．

11.2.1 基本の 3 つの要素

{targets} の哲学は，研究のワークフローを三つの要素に分けて考えることです．それは，ファイル，関数，オブジェクト です．

- ファイル: tar_file() で定義される，ファイルのパス名を持つオブジェクト．ファイルのサイズやタイムスタンプも保存されているため，ファイルのパスが変更されていなくても，ファイルの中身が変更されていれば，依存するパイプラインが再計算される
- オブジェクト: 変数やデータフレームなどの R のオブジェクト
- 関数: R の関数．ただし，インプットには依存する全てのオブジェクトを指定する必要があり，アウトプットがファイルまたはオブジェクトである必要がある．これにより，依存関係を明示的にすることができる

イメージとしては，ファイルから始まり，それを読み込んでオブジェクトを作成し，そのオブジェクトを使って関数を実行して新しいオブジェクトを作成するという流れです．これらの要素は，tar_plan() の中で定義されます．

```
clean_data1 ← function(data1_raw) {
  data1_raw ▷
    filter(!is.na(col1))
}
```

```
tar_plan(
  tar_file(data1_raw_file, "path/to/file/data1.csv"),
  data1_raw = readr::read_csv(data1_raw_file),
  data1_cleaned = clean_data1(data1_raw),
)
```

上の例では，data1_raw_file がファイル，data1_raw と data1_cleaned がオブジェクト，readr::read_csv と clean_data1() が関数です．

11.2.2 実行と依存関係の管理

定義したパイプラインは，targets::tar_visnetwork() で可視化することができます．

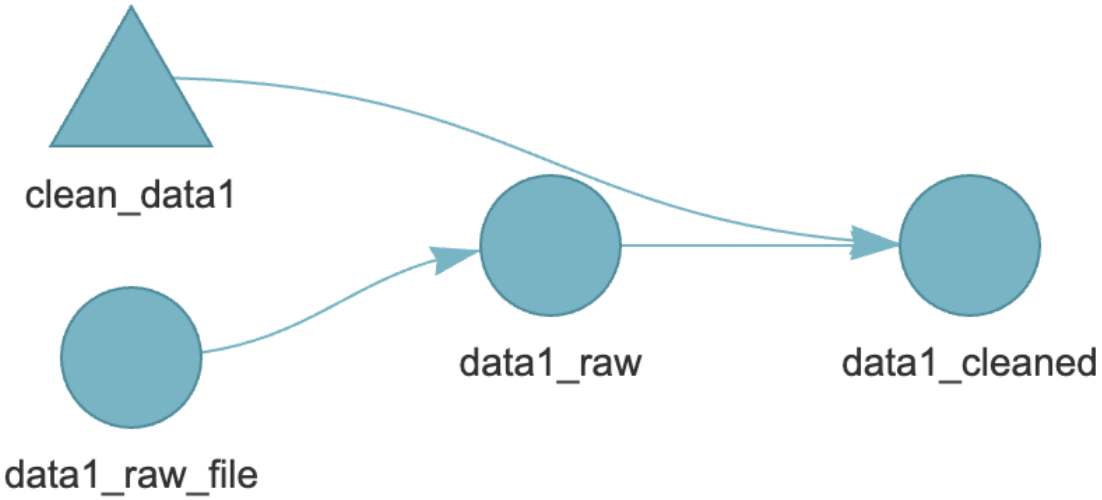


図 11.2: パイプラインの可視化．三角形が関数，丸がファイルとオブジェクト

ここでは三角形が関数，丸がファイルとオブジェクトを表していることがわかります．また，パイプラインが実行されていない状態が水色で表されています．ここで，targets::tar_make() を実行すると，

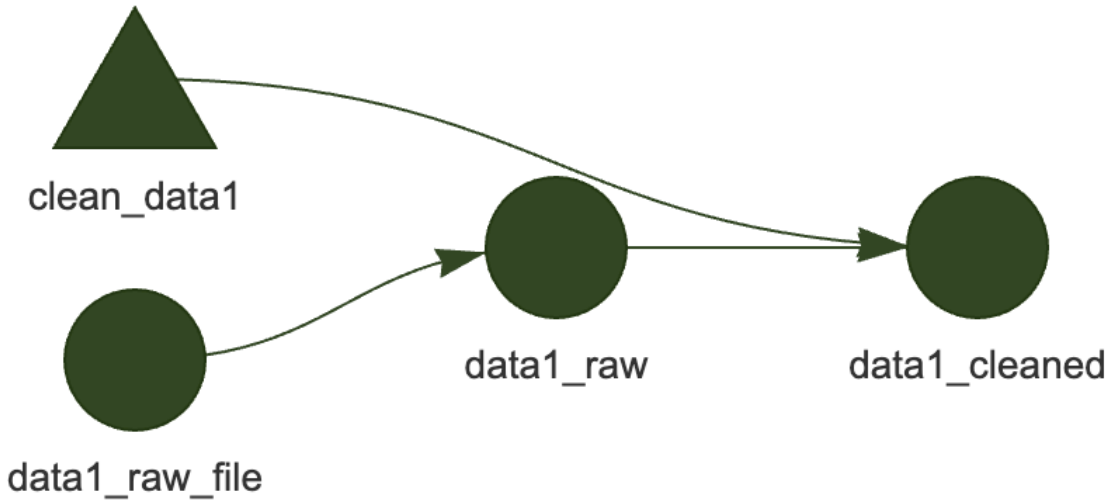


図 11.3: `tar_make()` で実行された部分はグレーに変わる

正常に実行されると, 実行された部分がグレーに変わります. ここで, `data1.csv` (`data1_raw_file`) の中身を変更すると,

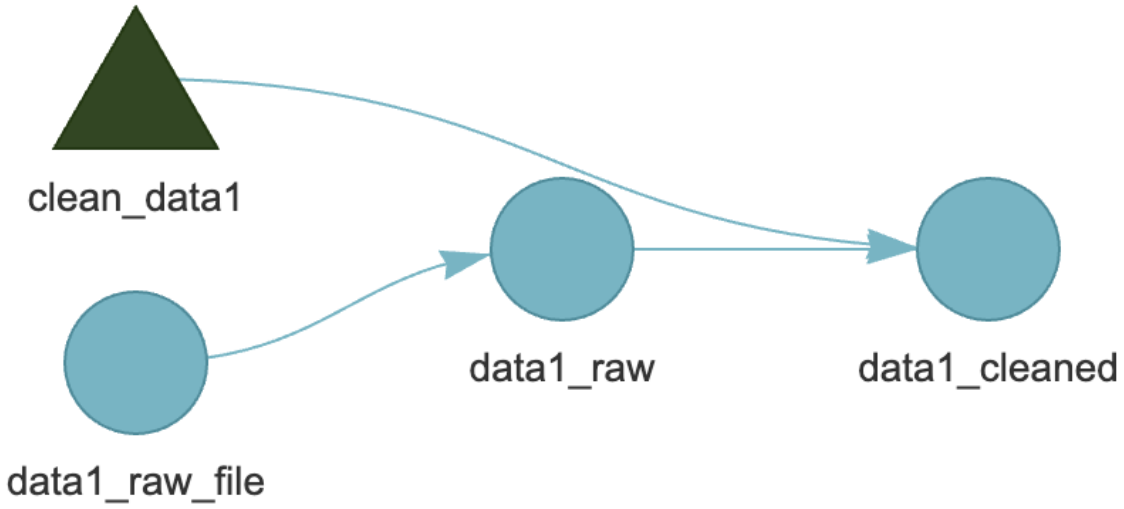


図 11.4: ファイルの中身を変更すると依存部分が未実行状態に戻る

依存関係のある部分が未実行状態に戻ります. もちろん, `targets::tar_make()` を実行すると, 依存関係のある部分が再計算されます. さらに, `clean_data1()` の中身を変更すると, 以下のようになります.

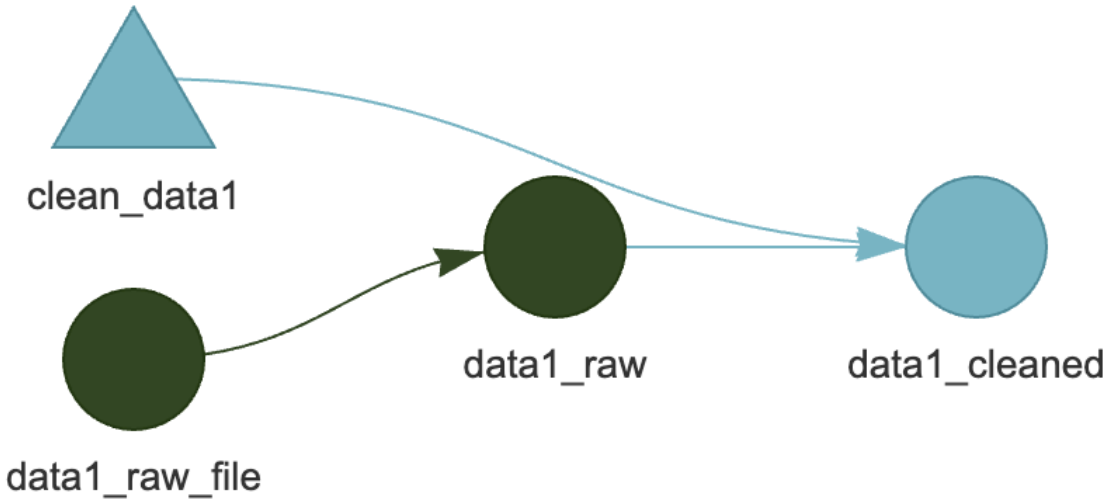


図 11.5: 関数の中身を変更しても, 依存部分だけが未実行状態に戻る

このように, tar_plan() 上でパイプラインの定義をし, tar_make() で実行するという流れを繰り返していくのが {targets} の基本的な使い方です.

11.3 Quarto + {targets} のワークフロー

11.3.1 1. _targets.R の初期設定を行う

プロジェクト全体の入り口となる _targets.R では, 重要なのは以下の 3 つです.

```
library(targets)
library(tarchetypes)
suppressPackageStartupMessages(library(dplyr))

here_rel ← function( ... ) {
  fs::path_rel(here::here( ... ))
}

tar_source()

tar_plan(
  # Data Preparation ──────────
  data,
  # Analysis ──────────────────
  fact,
  model,
```

```
  # Graphics ──────────────────
  fns_graphics = lst(theme_proj, color_base, color_accent, scale_fill_gender),
  # Manuscript ────────────────
  tar_quarto(manuscript_book, path = "manuscript", quiet = FALSE),
  tar_quarto(manuscript, path = "manuscript/main.qmd", quiet = FALSE),
  # Website ───────────────────
  tar_quarto(
    website,
    path = ".",
    quiet = FALSE,
    extra_files = here_rel("manuscript", "main.pdf")
  )
)
```

11.3.1.1 tar_source()

パイプラインで使用する関数のソースコードのあるディレクトリを指定します. デフォルトでは R/ ディレクトリが指定されています.

11.3.1.2 tar_plan()

パイプラインの定義を行います. 全てのパイプラインをここに記述する必要はなく, tar_source() で読み込まれるファイルの中でパイプラインを定義してそれを呼び出しても構いません. 例えば data というパイプラインは R/tar_data.R の中で定義されており, _targets.R ではその名前を書くだけで済みます.

11.3.1.3 here_rel()

here::here() は, プロジェクトのルートディレクトリを基準にした相対パスを指定するための関数です. これを使うことで, プロジェクトのルートディレクトリが変更されても, パスを変更する必要がなくなります. ただし, _targets.R の中で here::here() を使うと, 絶対パスが保存されてしまい, 他者と共有した場合に問題が発生します. そのため, here::here() の利便性を保ちつつ, 相対パスで保存できる関数を使用しています. この部分は Andrew Heiss さんのコードを参考にしています.

11.3.2 2. データクリーニングを定義する

データクリーニングのパイプラインを R/tar_data.R に定義します. 詳しくはコードを参照してください.

```
data ← tar_plan(
  dir_raw_accident_bike = download_accident_bike(
    here_rel("data", "accident_bike")
  ),
```

```
  raw_accident_bike = load_accident_bike(dir_raw_accident_bike),
  accident_bike = clean_accident_bike(raw_accident_bike)
)
```

ポイントとしては,

- raw データのファイルは `tar_file()` で定義する
- raw データの読み込みは, `readr::read_csv()` などの関数を使う
- クリーニングの関数は, raw データを引数に取り, クリーニング後のデータを返す

という工程を必ず踏むことです. また上記の最初の二つを統合して, 以下のような書き方もできます.

```
tar_plan(
  tar_file_read(
    data1_raw,
    here_rel("path", "to", "file", "data1.csv"),
    readr::read_csv(!!.x),
  ),
  data1_cleaned = clean_data1(data1_raw)
)
```

またオンライン上のファイルをダウンロードして `tar_file` で定義したい場合,

```
download_file <- function(url, destfile) {
  if (!file.exists(destfile)) {
    download.file(url, destfile)
  }
  return(destfile)
}
```

といった関数を定義して, `tar_file()` の中で呼び出すことができます. 保存先のパスを返す関数を定義するというのがポイントです.

11.3.3 3. ウェブサイトを設定する

ウェブサイトをパイプラインの中でビルドするには `tar_quarto()` を用います. ウェブサイトを用いる意味や具体的な設定は, 研究のためのウェブサイトの記事を参考にしてください.

11.3.4 4. データ分析やモデルの試行錯誤

上記で設定したウェブサイトの `playground` ディレクトリの中でノートブック形式のポストを作成し, データ分析やモデルの試行錯誤を行います. データクリーニングを行なった後のデータを `tar_load()` で読み込むことができます. また図などの設定は, `R/fns_graphics.R` 内で定義していて, それを読み込む形で使用しています.

```
targets::tar_config_set(
  store = here::here("_targets"),
  script = here::here("_targets.R")
)

targets::tar_load(c(accident_bike))
invisible(list2env(targets::tar_read(fns_graphics), .GlobalEnv))

theme_set(theme_proj())
```

ポイントとしては, 必要以上にパイプラインに組み込まないことです. 試行錯誤の段階のほとんどの分析は実際の論文には含まれません. そのため, それをパイプラインに組み込むと, ほとんど必要ないにも関わらず, 依存関係の管理が難しくなります. そのため, `playground` ディレクトリの中で試行錯誤を行い, 最終的に必要なものだけをパイプラインに組み込むようにしています.

`tar_quarto()` と Quarto の `freeze: auto` の設定を組み合わせることで, 一度コンパイルした結果は, 手動でコンパイルしない限り, 再計算されません. パイプラインの上流が変更された場合でもそれに依存する部分が再計算されないというデメリットはありますが, 不必要な部分の管理をほとんどしなくても良いというメリットがあります. 上流で変更があった場合には, プロジェクトの中で必要な部分だけを手動でコンパイルし直せば良いです.

11.3.5　5. 途中結果をスライドにまとめる

研究を進める中で, 途中結果をプレゼンする機会はよくあると思います. 試行錯誤した中で, 重要な結果をスライドにまとめるという形式が良いと思います. この段階も試行錯誤の過程の一つと考えるので, それをパイプラインに組み込む必要はありません. このプロジェクトでは, `slide` ディレクトリもウェブサイトのページの一部としてコンパイルされます.

私はスライドを, Quarto Clean Theme で作成しています. Quarto で作成されるため, このワークフローに自然に取り込むことができます. また, Typst を用いているため, Beamer などと比べて高速にコンパイルされます.

11.3.6　6. 3-5 を繰り返す. 主要な結果をパイプラインに組み込む

プレゼンなどでフィードバックをもらいながら, 研究を進めていきます. 論文の執筆に十分な結果が得られたら, 次のステップに進みます. ここで初めて, `R/tar_fact.R` や `R/tar_model.R` などのパイプラインを定義していきます.

11.3.6.1　主要な結果をパイプラインに組み込む

論文に必要な数値を `R/tar_fact.R` でパイプラインに組み込みます. 私は, `compute_*()` という関数を定義して, クリーニングされたデータを引数にとり, 計算した結果を数値あるいはデータフレームとして返すようにしています. 図表の作成は論文の中で行うので, ここでは行いません.

```
fact ← tar_plan(
  num_accident = compute_num_accident(accident_bike),
  logit_hospital_death = compute_logit_hospital_death(accident_bike)
)

compute_num_accident ← function(accident_bike) {
  accident_bike ▷
    summarize(n = n(), .by = c(type_person, gender))
}
```

LaTeX 等で執筆される方は，この段階で図表を保存すると良いと思います．例えば以下のようなパイプラインが考えられるでしょう．

```
tar_plan(
  tar_file(
    fig1_file,
    plot_fig1(data1, here_rel("path", "to", "file", "fig1.pdf"))
  )
)

plot_fig1 ← function(data1, path_fig1) {
  ggplot(data1, aes(x = col1, y = col2)) +
    geom_point()

  ggsave(path_fig1)
  return(path_fig1)
}
```

11.3.6.2 Julia のコードをパイプラインに組み込む

R の分析だけであれば，そのままパイプラインに組み込むことができますが，Julia のコードを組み込む場合は少し工夫が必要です．私は以下の方法を取っています．

1. Julia のソースコードファイルとして読み込み (`tar_file_read()`)，パイプラインに組み込む (`jl_file_*`)
2. Julia のソースコードを R の `system2()` でコマンドライン実行する (`run_model()`)
3. Julia の中で実行される結果は，CSV や YAML などのファイルとして保存しておき，R の中で読み込む (`tar_file_read()`)

```
model ← tar_plan(
  tar_map(
```

```
    values = list(name = c("main", "model")),
    names = name,
    tar_file_read(
      jl,
      here_rel("Julia", paste0(name, ".jl")),
      readLines(!!.x)
    )
  ),
  res_model = run_model(jl_file_main, jl_main, jl_model),
  tar_file_read(parameters, res_model[[1]], yaml::read_yaml(!!.x)),
  tar_file_read(demand_supply, res_model[[2]], read.csv(!!.x)),
  tar_file_read(equilibrium, res_model[[3]], yaml::read_yaml(!!.x))
)

run_model <- function(jl_file_main, ...) {
  system2(command = "julia", args = c("--project=.", jl_file_main))

  return(file.path(
    here_rel("output", "Julia"),
    c("parameters.yaml", "demand_supply.csv", "equilibrium.yaml")
  ))
}
```

ポイントは以下の 2 点です.

1. Julia 内の依存関係も含められるように, `run_model()` の引数に全ての依存関係を指定する
2. 結果を保存したファイルは, リストにまとめた上で, R から一つずつ読み込む

ちなみに, `system2()` で実行する場合, Julia のパスが通っていないことがあります (PC のデフォルトのシェルと R のシェルが異なるため). その場合は, `.Renviron` で定義した Julia のパスを `.Rprofile` で読み込むようにしています.

```
PATH_JULIA=/path/to/julia/
```

```
Sys.setenv(
  PATH = paste(Sys.getenv("PATH_JULIA"), Sys.getenv("PATH"), sep = ":")
)
```

11.3.7 7. 最終的な結果で論文を執筆する

論文の執筆は, manuscript ディレクトリの中で行います. ここでは, 上記で定義したパイプラインの実行結果を tar_load() で読み込んで, 論文を執筆します. 私は以下の 3 ステップで論文を執筆しています.

1. manuscript/_quarto.yml でディレクトリ自体を Quarto Book として設定する
2. Quarto Book として執筆する (01-intro.qmd, ⋯)
3. manuscript/main.qmd に 01-intro.qmd, 02-methods.qmd, ⋯ をまとめてコンパイルする

```
{{< include 01-intro.qmd >}}
{{< include 02-fact.qmd >}}
{{< include 03-model.qmd >}}
{{< include 04-result.qmd >}}
{{< include 05-conclusion.qmd >}}
```

11.3.7.1 なぜ Quarto Book として執筆するのか

Quarto Book として執筆すると, 複数ファイルに分けても cross-reference が効きます. これはかなり便利で, セクションごとにファイルを分けることで論文の構造が把握しやすくなり, cross-reference の入力補完が働くことで, 快適に執筆できます.

ただし, Quarto Book は出力形式としては論文に相応しくないので, それらのファイルをまとめて manuscript/main.qmd でコンパイルしています. この時, バックエンドは高速でコンパイルできる Typst を使用しています. またテンプレートは quarto-academic-typst を使用しています. ちなみに LaTeX のソースコードが必要になった場合は, quarto::quarto_render() で LaTeX のソースコードを生成することができます (私は, 博士論文の執筆の際, LaTeX テンプレートに合わせるために使用しました).

11.3.7.2 LaTeX で執筆する場合

LaTeX で執筆する場合も TinyTeX を用いてコンパイルするならば, パイプラインに組み込むことができます. 例えば以下のような形が考えられるでしょう. ただし, 実際には全ての図表の依存関係を入れる必要があると考えられます.

```
tar_plan(
  tar_file_read(
    manuscript,
    here_rel("manuscript", "main.tex"),
    readLines(!!.x)
  ),
  tar_file(
    manuscript_pdf,
    compile_latex(manuscript, here_rel("manuscript", "main.pdf"))
```

```
  )
)

compile_latex ← function(manuscript_file, path_pdf) {
  tinytex::xelatex(
    manuscript_file,
    pdf_file = path_pdf
  )
  return(path_pdf)
}
```

第 12 章
AI と研究する

12.1　文献調査

文献調査では, AI に手元の文献ライブラリを直接参照させると便利です. ここでは, 文献と引用 の章で使った Zotero を, MCP (Model Context Protocol) 経由で Claude につなぎます. これにより, 「このテーマの論文をライブラリから探して」「この論文の要点をまとめて」といった依頼を, Claude が実際の Zotero ライブラリを検索しながら答えられるようになります. MCP の仕組みそのものについては セクション C.5 を参照してください.

12.1.1　Zotero MCP Server

ここで使うのは, `kujenga/zotero-mcp` というオープンソースの MCP サーバーです. このサーバーは, Claude に対して次の 3 つのツールを提供します.

- `zotero_search_items`: ライブラリを検索する
- `zotero_item_metadata`: 文献の書誌情報を取得する
- `zotero_item_fulltext`: 文献の全文を取得する

Claude は, ユーザーの依頼に応じてこれらのツールを自分で選んで呼び出します.

12.1.1.1　事前準備

このサーバーは, 手元で動いている Zotero に接続します (ローカル API). そのため次の 2 つを用意します.

1. Zotero 7 以降のデスクトップアプリをインストールし, 起動しておきます.
2. Zotero の「設定」→「詳細」で, 「この PC 上の他のアプリケーションが Zotero と通信することを許可する」にチェックを入れます (英語表記では Preferences → Advanced → “Allow other applications on this computer to communicate with Zotero”).

サーバー自体は Python 製で, `uv` の `uvx` コマンドで起動します. `uv` が入っていなければ, 先にインストールしておいてください (公式サイトの手順に従います). `uvx` はパッケージを自動でダウンロードして実行するので, `zotero-mcp` を手動でインストールする必要はありません.

12.1.1.2　Claude Desktop での設定

Claude Desktop の設定ファイル `claude_desktop_config.json` に, サーバーを 1 つ追加します. このファイルの場所は, macOS では `~/Library/Application Support/Claude/`, Windows では `%APPDATA%\Claude\` です.

```
{
  "mcpServers": {
    "zotero": {
      "command": "uvx",
      "args": ["--upgrade", "zotero-mcp"],
      "env": {
        "ZOTERO_LOCAL": "true"
      }
    }
  }
}
```

`ZOTERO_LOCAL` を `true` にすると, 上で有効にしたローカル API 経由で手元の Zotero に接続します. 設定を保存して Claude Desktop を再起動すると, サーバーが読み込まれます.

12.1.1.3　Claude Code での設定

ターミナルで動く Claude Code なら, 設定ファイルを直接編集せずに, `claude mcp add` コマンドで追加できます.

```
claude mcp add zotero --env ZOTERO_LOCAL=true -- uvx --upgrade zotero-mcp
```

`--` (ダッシュ 2 つ) は, Claude Code 自身のオプションと, サーバーを起動するコマンドの境目を表します. `--` の後ろ (`uvx --upgrade zotero-mcp`) が, サーバーを起動するコマンドとしてそのまま実行されます.

12.1.1.4　使ってみる

設定できたら, Claude に自然な日本語で依頼するだけです. 例えば次のように頼むと, Claude は `zotero_search_items` でライブラリを検索し, 見つかった論文を一覧して答えます.

> Zotero のライブラリから, 最低賃金の雇用効果に関する論文を探して, それぞれ一行で要約して.

さらに特定の論文について「この論文の識別戦略を説明して」と頼めば, `zotero_item_fulltext` で全文を取得したうえで答えます. 手元のライブラリに基づいて答えるので, 存在しない論文をでっち上げる (ハルシネーション) 危険が減るのが利点です.

12.1.1.5　注意点

- Zotero が起動していないと, ローカル API に接続できず, ツールが失敗します. 使うときは Zotero を開いたままにしておきます.
- 全文取得 (`zotero_item_fulltext`) はローカル API では新しめの Zotero でのみ対応しています. うまくいかない場合や, Zotero を起動せずに使いたい場合は, ローカル API の代わりに Zotero の Web API を使う方法もあります. その場合は https://www.zotero.org/settings/keys で API キーとライブラリ ID を取得し, `ZOTERO_LOCAL` を `false` にして `ZOTERO_API_KEY` と `ZOTERO_LIBRARY_ID` を設定します. API キーはコードやリポジトリに直接書かず, API の章の e-Stat の例と同じように, 秘密情報として扱ってください.

12.2　ワークフロー

研究を始めるたびにディレクトリ構成や設定を一から作るのは無駄が多く, AI に手伝ってもらうにも「どこに何を置くか」が定まっていないと指示がぶれます. そこで, targets による再現性 の章で紹介した Quarto + `{targets}` のワークフローを, そのまま使えるテンプレートにまとめたものが `kazuyanagimoto/template-research` です. GitHub の「Use this template」から自分のリポジトリを作れば, 研究プロジェクトの骨格がすぐに手に入ります.

12.2.1　テンプレートの構成

テンプレートをクローンすると, 次のようなフォルダ構成になっています. targets による再現性 の章で説明したワークフローが, そのままディレクトリの形になっています.

```
template-research/
├── _targets.R        # pipeline composition
├── R/                # pipeline code: tar_data.R, tar_fact.R, ...
├── data/             # raw data (gitignored)
├── notes/            # exploratory Quarto notes (NN-name/)
├── slides/           # presentation decks
├── manuscript/       # paper: Quarto Book + Typst
├── references.bib    # bibliography
├── rproject.toml     # R version + packages (managed by rv)
└── CLAUDE.md         # project conventions for the AI assistant
```

各ディレクトリの役割は次のとおりです.

- `_targets.R` と `R/tar_*.R`: パイプラインの定義. パイプラインはデータオブジェクトだけを生成し, 図はファイルに保存せず, Quarto の中で `tar_read()` を使ってその場で描きます.
- `data/`: 生データ (gitignore の対象).
- `notes/NN-name/`: 試行錯誤のノート. 1 フォルダが 1 ラウンドの試行錯誤にあたり, ここでの計算はここに閉じておき, 固まったものだけをパイプラインに昇格させます.

- `slides/`: 発表スライド. パイプラインから切り離した「凍結スナップショット」で, 必要なデータを自分で持ち, `tar_load()` を呼びません (スライド の章の考え方です).
- `manuscript/`: 論文. `tar_load()` でパイプラインの結果を読み込み, 常に最新のデータを反映します. Quarto Book として書き, Typst で PDF にします.

パッケージは `install.packages()` ではなく `rv` で, R のバージョンは `rproject.toml` で固定し, コードの整形は `air` に任せます (この本のリポジトリも同じ構成です).

これらのフォルダがどう連携するかを図にすると, 図 12.1 のようになります. 生データがパイプラインを通ってデータオブジェクトになり, それを論文とスライドが受け取る, という流れです. `notes/` で固まった分析はパイプラインに昇格し, ルートの `CLAUDE.md` を読んだ AI がこのプロジェクト全体の作業を手伝います.

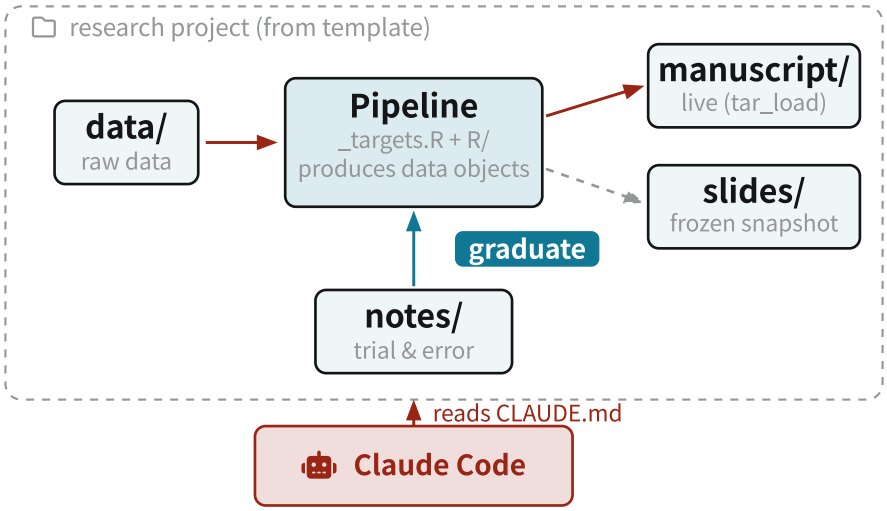


図 12.1: 研究プロジェクトの構成とデータの流れ

12.2.2　CLAUDE.md で AI にプロジェクトの規約を教える

このテンプレートの肝は, ルートに置かれた `CLAUDE.md` です. これは, Claude Code のような AI アシスタントが起動時に読み込む指示書で, このプロジェクトの規約を AI に伝えます. これがあると, AI は「このプロジェクトのやり方」に沿って作業します. 逆に, `CLAUDE.md` が無ければ, AI は一般的な, しばしばプロジェクトの流儀と食い違うやり方でコードを書いてしまいます.

テンプレートの `CLAUDE.md` には, 例えば次のような規約が書かれています.

- パイプラインはデータオブジェクトだけを生成し, 図はファイルに保存せず Quarto 内で `tar_read()` を使って描く.
- パッケージは `install.packages()` ではなく `rv` で管理する.
- ノート内だけの計算はノートに閉じ, 固まってからパイプラインに移す.
- スライドは凍結スナップショットとして, `tar_load()` を呼ばず自分でデータを持つ.
- 表は `knitr::kable()` ではなく `tinytable` で作り, 回帰は `fixest` を使う.
- 実証結果の数値を本文やキャプションに直接書かず, 計算した値をインラインコードで埋め込む.

これらは, この本のこれまでの章で推奨してきたルールとほぼ同じです. 大事なのは, こうした自分の流儀を `CLAUDE.md` に明文化しておくと, AI がそれを前提に働いてくれる, という点です. 新しいルールを決めるたびに `CLAUDE.md` に書き足していくことで, AI との共同作業は少しずつ自分のやり方に馴染んでいきます.

12.2.3 始め方

GitHub でテンプレートから自分のリポジトリを作り, クローンしたら, ツールチェインを用意します. R のバージョンは `rig` で, パッケージは `rv` で管理します.

```
rig add 4.6                 # install the pinned R version
brew install rv             # package manager (macOS; see repo for other OSes)
brew install air            # optional: R formatter

rv sync                     # restore the package library
R -e 'targets::tar_make()'  # run the pipeline
```

最後に, `.Renviron.example` を `.Renviron` にコピーして, API キーなどの秘密情報を書き込めば (API の章を参照), 準備は完了です.

12.2.4 AI と進める

研究は, ノートで試し, 固まったものをパイプラインに移し, 論文にまとめる, というサイクルの繰り返しです (図 12.2). 具体的な流れは targets による再現性 の章と同じですが, AI を使うと各段階が次のように楽になります.

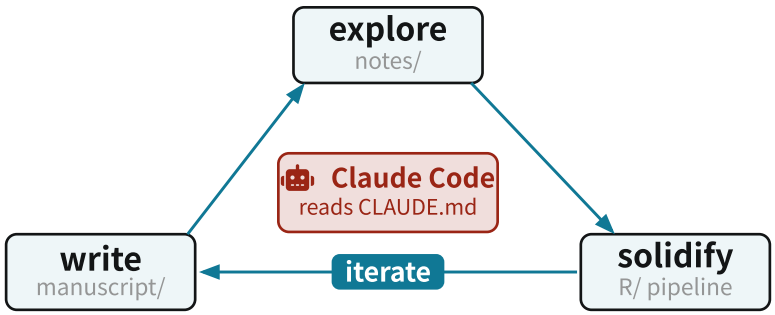


図 12.2: AI と回す研究のサイクル

- ノートでの試行錯誤: 「このデータで最低賃金の雇用効果を回帰して図にして」と頼めば, `notes/` の中にコードを書いてくれます.
- パイプラインへの昇格: 固まった分析を `R/tar_*.R` に移す作業を任せます.
- 論文執筆: `manuscript/` でパイプラインの結果を読み込み, 文章の下書きや相互参照の整理を手伝わせます.

いずれの段階でも, `CLAUDE.md` があるおかげで, AI はプロジェクトの規約 (パイプラインはデータオブジェクトだけ, 図は `tar_read()` で描く, など) を守ったコードを書きます. AI に任せきりにするの

ではなく, 規約を明文化して土台を整え, その上で AI に働いてもらう, というのがこのワークフローの考え方です.

付録 A
Shell

A.1 Shell とは

A.1.1 Unix 哲学

これから使うシェルという道具は, 前章の Unix の系譜 (セクション 1.3.2) で見た OS, Unix とともに Bell 研究所で生まれました. macOS と Linux でコマンドや開発ツールがほぼ共通なのは同じ Unix の流れを汲むためで, これから学ぶシェルの操作もこの系譜の上にあります. その設計の背景にあるのが Unix 哲学 (Unix philosophy) です.

> Do One Thing and Do It Well

小さくよく設計された部品を組み合わせ, つなぎ合わせることでより複雑で強力な処理を組み立てる, という思想です. 「ミニマルでモジュラー」とも言われるこの考え方は, シェルの設計そのものに色濃く表れています.

A.1.2 Shell, Terminal, CLI

前章で見たように, アプリケーションは OS が用意した窓口 (システムコール) を通じてその機能を利用するのでした (セクション 1.3). シェルもその一つで, OS への指示を人間がキーボードから直接打ち込むためのプログラムです. シェル (shell) やターミナル (terminal) といった言葉は, いずれも **コマンドラインインターフェース** (CLI, command line interface) を指しています. シェルにはいくつか種類がありますが, ここでは最も標準的な **Bash** (Bourne again shell) を中心に扱います. 各 OS の標準的なシェルは次のとおりです.

- Linux: Bash
- macOS: zsh (Bash 互換)
- Windows: PowerShell, cmd.exe, Git Bash

Windows の場合は Bash と非互換な PowerShell が使われることが多いですが, この授業では WSL を使うことを推奨しているため, Linux と同じく Bash を使ってもらいます.

A.1.3 なぜシェルを使うのか

GUI (マウス操作) で済むのに, なぜわざわざコマンドを打つのでしょうか. 理由は大きく 4 つあります.

- **力 (Power)**: GUI ではそもそもできないことができます. トラブルの解決もコマンドラインでしかできないことが多いです.
- **再現性 (Reproducibility)**: スクリプトに書いた操作は再現できますが, クリック操作は再現できません.
- **サーバーとの対話**: リモートのサーバーやスーパーコンピューターを使うとき, シェルが唯一の窓口になることが多いです.
- **AI**: AI はシェルを使ってコンピュータを操作します. シェルの仕組みがわかっていれば, AI に指示を出すときもより正確に伝えられます

A.2 シェルの基本

A.2.1 はじめてのシェル

まずはシェルを開いてみましょう. VSCode を使っているなら, `Ctrl` + `` ` `` で統合ターミナルが開きます. macOS のターミナルアプリや, WSL の Ubuntu などでも構いません.

開くと, 次のようなプロンプトが表示されます.

```
username@hostname:~$
```

これはシェル流の「あなたは誰で, いまどこにいるのか」という表示です.

- `username` は利用者の名前です (1 台の計算機に複数いることもあります).
- `@hostname` は計算機やサーバーの名前です.
- `:~` はいまいるディレクトリです (`~` はホームディレクトリを表します).
- `$` はコマンド入力の開始位置です. 管理者権限を持つ root のときはこれが `#` に変わります.

A.2.2 キーボードショートカット

シェルを快適に使うために, 最低限これだけは覚えておくとよいでしょう.

- `Tab`: 入力の補完
- `↑` / `↓`: 過去のコマンドをさかのぼる

A.2.3 コマンドの構文

Bash のコマンドは, すべて同じ基本構文を持っています.

```
command  option(s)  argument(s)
```

たとえば次のようになります.

```
ls -lh ~/Documents/
sort -u myfile.txt
```

コマンド (command) は何をするかを指定します.

- オプションや引数は無くても構いませんが, コマンドだけは必ず必要です.
- `ls` はディレクトリの中身を一覧表示するコマンド, `sort` は行を並べ替えるコマンドです.

オプション (option) はフラグ (flag) とも呼ばれ, 動作を細かく指定します.

- ダッシュ `-` で始まり, たいてい 1 文字です.
- `-l` は長い形式で表示する, `-h` は人間に読みやすい単位で表示する, `-u` は重複行を削除する, といった意味です.
- 複数のオプションは 1 つのダッシュにまとめられます.

```
ls -l -a -h /var/log    # this one
ls -lah /var/log        # and this one are the same
```

- まれに, 2 つのダッシュで始まる長い名前のオプションもあります.

```
ls --group-directories-first --human-readable /var/log
```

引数 (argument) は, 何に対して操作するかを指定します.

- たいていはファイルやパス, あるいはその集合です.

空白

シェルコマンドで大事なのは, 空白と順番によって option と argument が区別されているということです. シェルはまず行を空白で区切ってトークンに分け, その先頭をコマンド名, 残りを引数とみなします. だからこそ空白の有無が意味を大きく変えます.

たとえば環境変数の代入がそうです. 次の 2 つは, どちらも同じ `NAME`, `=`, `Alice` という語からできていますが, 空白の有無でまったく違う意味になります.

```
NAME=Alice      # OK: assign "Alice" to the variable NAME
NAME = Alice    # error: run a command called "NAME" with args "=" and "Alice"
```

`NAME=Alice` は空白を含まない 1 つのトークンなので, シェルは「代入」と解釈します. 一方 `NAME = Alice` は `NAME` / `=` / `Alice` という 3 つのトークンに割れてしまうため, シェルは `NAME` をコマンド名だと思って実行しようとし, `NAME: command not found` になります. 代入では `=` の前後に空白を入れてはいけません.

A.2.4　ヘルプの引き方

コマンドの使い方に困ったら, `man` (manual pages) が助けになります.

```
man ls
```

- スペースキーで 1 画面ずつ進みます.
- `/pattern` でマニュアル内を検索でき, `n` で次の一致へ進みます.
- `h` で `man` 自身のヘルプ, `q` で終了します.

`man` は網羅的ですが冗長なので, 最近なら AI に聞くのもよいでしょう.

A.3 ファイルとディレクトリ

A.3.1 移動する

ディレクトリ間の移動は, 次の 2 つが基本になります.

- `pwd`: いまいるディレクトリ (working directory, current directory) を表示する.
- `cd`: ディレクトリを移動する.

絶対パス

絶対パスとは, ルートディレクトリ `/` から始まるパスのことです. 例えば, この資料を作っているプロジェクトの絶対パスは, `/Users/kazuharu/github/workshop-graduate-2026` になります.

相対パス

相対パスとは, カレントディレクトリからの相対的な位置を表すパスです. たとえば, `examples` サブディレクトリに移動するには, 絶対パスでは `/Users/kazuharu/github/workshop-graduate-2026/examples` と書く必要がありますが, 相対パスなら単に `examples` と書くだけで済みます.

コマンド

絶対パスも使えますが, 相対パスと特殊記号を組み合わせる方が扱いやすいです. 特殊記号には, ホーム (`~`), カレントディレクトリ (`.`), 親ディレクトリ (`..`) があります.

```
cd examples    # enter the examples subdirectory
cd ../..       # go two directories up
pwd            # check where we are now
```

ここで一つ注意があり, ディレクトリ名に空白が含まれていると厄介です. たとえば `My Documents` というフォルダに `cd My Documents` としても動きません. Bash は空白を区切りとみなし, `My` と `Documents` を別々の引数だと解釈してしまうからです.

- 対処法 1: 引用符で囲む. `cd "My Documents"`
- 対処法 2: 空白をエスケープする. `cd My\ Documents` (`Tab` 補完を使うと自動でこの形になります)
- 一番よい対処法: そもそもファイル名・フォルダ名に空白を使わない

A.3.2 一覧表示する

`ls` はディレクトリの中身を一覧表示します. `-lh` オプション (長い形式, 人間に読みやすい単位) を付けると, 各オブジェクトの属性まで見えます.

```
ls -lh examples
```

出力の各行は, 次のような情報を持っています.

```
drwxr-xr-x 2 kazuharu users 4.0K Jan 12 22:12 ABC
```

これを左から順に読み解くと, 次のようになります.

- 先頭 1 文字はオブジェクトの種類です. `d` (ディレクトリ), `l` (リンク), `-` (ファイル).
- 続く 9 文字はパーミッション (権限) です. 3 文字ずつ, **所有者**, **所有者のグループ**, **その他のユーザー** の順に並びます.
  - 各文字は `r` (読み取り), `w` (書き込み), `x` (実行) の可否を表します. `-` はその権限が無いことを示します.
- 次の数字はハードリンクの数ですが, あまり重要ではありません.
- その後に, オブジェクトの **所有者** と **グループ** が続きます. グループの名前は環境によって変わり, この例の `users` の位置が, macOS では `staff`, Ubuntu (WSL) ではユーザー名と同名のグループになります.
- 最後に, サイズ, 作成日時, 名前が並びます.

パーミッションと所有者については, 後の節で詳しく説明します (セクション A.8).

A.3.3　作る・消す

シェルで最もよく行う作業の一つが, ファイルやディレクトリの作成です.

- `mkdir`: ディレクトリを作る.
- `touch`: 空のファイルを作る.

```
mkdir testing
touch testing/test1.txt testing/test2.txt testing/test3.txt
ls testing
```

消すときは次を使います.

- `rm`: ファイルを消す.
- `rmdir`: 空のディレクトリを消す.

```
rm testing/test1.txt
rmdir testing       # fails if anything is left inside
```

`rmdir` は中身の入ったディレクトリを消せません. そのときは `rm` に「再帰的」(`-r`) と「強制」(`-f`) のオプションを付けます.

```
rm -rf testing      # deletes it with its contents
```

> ⚠ `rm -rf` は取り消せない
>
> `rm -rf` はゴミ箱を経由せず, 確認もなく即座に消します. パスを間違えると取り返しがつかないので, 実行前に必ずパスを見直してください.

A.3.4　コピー・移動・名前変更

`cp` はコピー, `mv` は移動です.

```
cp reps.txt copies/reps-copy.txt    # copy under a new name
cp -r meals/ copies/                # -r copies a directory and its contents
mv ABC/abc.txt examples             # move abc.txt into examples
```

同じディレクトリ内で新しい名前を付けて「移動」すると, それは実質的にリネームになります.

```
mv reps-copy.txt reps2.txt          # moving == renaming
```

多数のファイルをまとめてリネームしたいときは, `rename` が便利です. パターンと置換文字列, 対象ファイルを指定します. ワイルドカード (次節) と組み合わせると真価を発揮します.

```
rename csv TXT meals/monday.csv     # change the extension of one file
rename csv TXT meals/*              # apply it to everything in meals
```

A.3.5 ワイルドカード

ワイルドカード は, 他の文字の代わりになる特殊文字です. 重要なのは次の 2 つです.

- `*`: 任意の長さの文字列にマッチします. ファイルをまとめて操作したいときに便利です.
- `?`: 任意の 1 文字にマッチします. 似た名前のファイルを区別したいときに便利です.

```
cp *.sh copies          # copy every file with the .sh extension
rm copies/*             # delete everything inside copies
ls meals/??nday.csv     # matches monday.csv and sunday.csv
ls meals/?onday.csv     # matches monday.csv
```

A.3.6 探す

ナビゲーション関連で最後に紹介するのが `find` です. 名前やサイズなど, さまざまな条件でファイルやディレクトリを探せます.

```
find examples -iname "monday.csv"   # search by name (recursive, case-insensitive)
find . -iname "*.txt"               # every .txt below the current directory
find . -size +100k                  # files larger than 100 KB
```

A.4 テキストファイルの操作

経済学を含むデータサイエンスでは, スクリプト, Markdown, CSV など, テキストファイルを扱う時間が長いです. シェルはテキスト処理がとても得意なので, ここで代表的な道具を紹介します.

A.4.1 数える

`wc` (word count) は, 行数・単語数・文字数を数えます.

```
wc sonnets.txt
```

なお文字数は手で数えるより多くなります. `wc` は目に見えない改行文字 `\n` も 1 文字として数えるからです.

A.4.2　読む

最も単純にテキストを読むコマンドが `cat` (concatenate) です. ただし `cat` は全文を一気に表示するので, 長いファイルでは扱いにくいです.

```
cat -n sonnets.txt      # -n adds line numbers
```

長いファイルを 1 画面ずつ読むには `more` や `less` を使います. `f` / `b` で前後に移動し, `q` で終了します.
先頭や末尾だけを覗くなら `head` と `tail` が便利です (既定では 10 行).

```
head -n 3 sonnets.txt       # first 3 lines
tail -n 1 sonnets.txt       # last line
tail -n +3024 sonnets.txt   # everything from line 3024 on
```

A.4.3　探す

テキストの中からパターンを探すには `grep` を使います. 正規表現によるマッチングができます.

```
grep -n "Shall I compare thee" sonnets.txt    # -n also prints the line number
```

`grep` はディレクトリ内の複数ファイルにまたがって探すこともできます. 関数名を含むファイルを探す, といった場面で特に役立ちます.

```
grep -Rl "pasta" meals    # -R recursive, -l print only the matching file names
```

`-i` (大文字小文字を無視) など, 便利なオプションが他にもあります.

A.4.4　加工する

テキストを加工する代表的なコマンドが `sed` と `awk` です. どちらも強力で柔軟ですが (特に `awk` は CSV に強いです), ここでは基本だけ示します.
ある文字列を別の文字列に置換するには, `sed` を使います.

```
sed -i 's/Jack/Bill/g' nursery.txt    # replace every Jack in the file with Bill
```

複数のコマンドをつなげれば, より複雑な処理もできます (パイプ `|` は次節で説明します). たとえば, シェイクスピアのソネットで最もよく使われる単語の上位 10 件は, 次のように求められます.

```
sed -e 's/\s/\n/g' < sonnets.txt | sort | uniq -c | sort -nr | head -10
```

A.4.5　並べ替え・重複削除

`sort` は行を並べ替えます. `-u` (unique) を付けると, 重複行を取り除けます.

```
sort -u reps.txt        # sort and drop duplicates
sort -ur reps.txt       # -r reverses the order
```

A.5　リダイレクトとパイプ

A.5.1　リダイレクト

`>` を使うと, コマンドの出力を画面ではなくファイルへ送れます. これを **リダイレクト** (redirect) といいます.

```
echo "first line"               # print to the screen
echo "first line" > note.txt    # write it out to note.txt
```

既存のファイルに **追記** したいときは `>>` を使います. `>` は上書きしてしまうので注意してください.

```
echo "second line" >> note.txt
cat note.txt
```

私はこれを `.gitignore` への追記によく使います.

```
echo "*.csv" >> .gitignore
```

A.5.2　パイプ

`|` (パイプ) は Bash で最も強力な機能の一つです. あるコマンドの出力を, 次のコマンドの入力として渡します. 単純な操作を数珠つなぎにして, 複雑な処理を組み立てられます (まさに Unix 哲学です).

```
cat -n sonnets.txt | head -n 100 | tail -n 10    # show lines 91-100
```

たとえば「336 行目から始まる 14 行のソネットだけを取り出す」なら, 次のように書けます.

```
tail -n +336 sonnets.txt | head -n 14
```

コマンド履歴の検索にも使えます. 入力したコマンドは `~/.bash_history` (zsh なら `~/.zsh_history`) に残るので, 次のようにたどれます.

```
cat ~/.bash_history | grep head
```

A.6　繰り返し (for ループ)

A.6.1　for の構文

Bash の for ループは, 他の言語のものとよく似ています. 基本の構文は次のとおりです.

```
for i in LIST
do
  OPERATION $i    # $ marks a variable
done
```

セミコロン ; を使えば 1 行にまとめられます.

```
for i in LIST; do OPERATION $i; done
```

A.6.2　例: 数列を表示する

```
for i in 1 2 3 4 5; do echo $i; done
```

長い数列は, ブレース展開 `{1..n}` を使うと書かずに済みます.

```
for i in {1..5}; do echo $i; done
```

A.6.3　例: 複数の CSV を結合する

私が実際によく使う, より現実的な例を示します. `meals` ディレクトリにある曜日ごとの CSV を, 1 つの `mealplan.csv` にまとめたいとします. 素朴にやると, 空のファイルを作り, 各ファイルの中身を追記していくことになります.

```
touch mealplan.csv
for i in $(ls meals/*day.csv)
do
  cat $i >> mealplan.csv
done
```

ただしこれだと, 各ファイルの見出し行 (ヘッダー) が繰り返し入ってしまいます. そこで, 最初に 1 つのファイルからヘッダーだけを書き出し, 残りは 2 行目以降だけを追記するようにします.

```
head -1 meals/monday.csv > mealplan.csv       # write out the header only
for i in $(ls meals/*day.csv)
do
  tail -n +2 $i >> mealplan.csv               # append everything from line 2 on
done
```

シェルで結合する利点は, **効率の良さ** にあります. すべてのファイルを同時にメモリ (RAM) に載せる必要がないので, ファイル数が多いとき, あるいは 1 つ 1 つが巨大なときに大きな差が出ます.

A.7 スクリプト

A.7.1 Hello World

データやファイル構造を探索するときは, シェルに対話的にコマンドを打つのが理にかなっています. 一方で, 一連のコマンドをまとめた **シェルスクリプト** を書いておくと, 再現可能になります. シェルスクリプトは `.sh` という拡張子で表します.

たとえば次のような短いスクリプト `hello.sh` を考えます.

```
#!/bin/sh
echo -e "\nHello World!\n"
```

- 1 行目の `#!/bin/sh` は シバン (shebang) と呼ばれ, どのプログラムで実行するかを指定します.
- 2 行目が実際に実行したいコマンドです. `echo` の `-e` はエスケープ文字 (`\n` = 改行) を解釈させるオプションです.

実行するには, ファイル名を打つか, `bash` に渡します.

```
bash hello.sh
```

A.7.2 Rscript

シェルで実行できるのはシェルスクリプトだけではありません. 同じ仕組みは他のプログラムにも当てはまります. この授業で特に重要なのが, R スクリプトを実行する `Rscript` です.

```
Rscript -e "cat('Hello World, from R!')"
```

より典型的な使い方は, R スクリプトのファイルを丸ごと実行することです. このとき, シェルから R へ追加の引数を渡せると便利な場面が多いです. R 側では, スクリプトの冒頭で次のように引数を受け取ります.

```
args ← commandArgs(trailingOnly = TRUE)
i ← args[1]
j ← args[2]
```

こうしておくと, シェルからファイル名の後ろに引数を並べて渡せます.

```
Rscript hello.R 12 9
```

引数を渡すかどうかは任意ですが, 大きなジョブをまとめて回す (バッチ処理) ようなときに役立ちます.

A.8　権限とパーミッション

A.8.1　スーパーユーザー: root と sudo

Linux には大きく 2 種類のユーザーがいます.

- 通常のユーザー
- スーパーユーザー (root とも呼ばれます)

違いは権限です. スーパーユーザーはシステムの変更, ソフトウェアのインストール, 他ユーザーのフォルダの閲覧などができます. 通常のユーザーにできることはずっと限られています. Unix 系 OS がウイルスなどの脅威に比較的強いのは, (悪意ある) ソフトウェアのインストールにスーパーユーザー権限が要るからです.

では, 通常のユーザーはどうやってソフトのインストールのような操作をするのでしょうか. 答えは `sudo` (“superuser do”) を使うことです. 実行したいコマンドの前に `sudo` を付けると, 一時的にスーパーユーザーの権限で実行できます.

```
ls /root          # fails
sudo ls /root     # succeeds
```

root として常時ログインすることも一応できますが, 安全装置も「元に戻す」機能も無いため, 一般に悪い習慣とされています.

A.8.2　パーミッションを変える: chmod

先ほど `ls -lh` で見た `ABC` ディレクトリを思い出しましょう.

```
drwxr-xr-x 2 kazuharu users 4.0K Jan 12 22:12 ABC
```

このパーミッションを変えるのが `chmod` です. パーミッションの表し方には 2 通りあります.

8 進数表記 (octal) は, `4` (読み取り), `2` (書き込み), `1` (実行) を足し合わせます.

- $7 = 4 + 2 + 1$ で, 読み・書き・実行のすべて.
- $5 = 4 + 0 + 1$ で, 読みと実行のみ (書き込み不可).
- 所有者・グループ・その他の 3 者ぶんの数字を並べます.

記号表記 (symbolic) は, 記号で指定します.

- 対象: `u` (所有者), `g` (グループ), `o` (その他), `a` (全員)
- 権限: `r` (読み), `w` (書き), `x` (実行)
- 変更: `+` (追加), `-` (削除), `=` (設定)

よく使う権限を並べると, 次のようになります.

8 進数	記号	パーミッション	意味
777	a=rwx	rwxrwxrwx	全員が読み・書き・実行
755	u=rwx,go=rx	rwxr-xr-x	所有者は全権, 他は読みと実行
700	u=rwx,go=	rwx——	所有者のみ全権

8 進数	記号	パーミッション	意味
644	u=rw,go=r	rw-r–r–	所有者は読み書き, 他は読みのみ

たとえば `ABC` ディレクトリを全員フルアクセスにするなら, 次のいずれでも構いません (ディレクトリなので再帰的に `-R` を付けます).

```
chmod -R 777 ABC        # octal notation
chmod -R a=rwx ABC      # symbolic notation
```

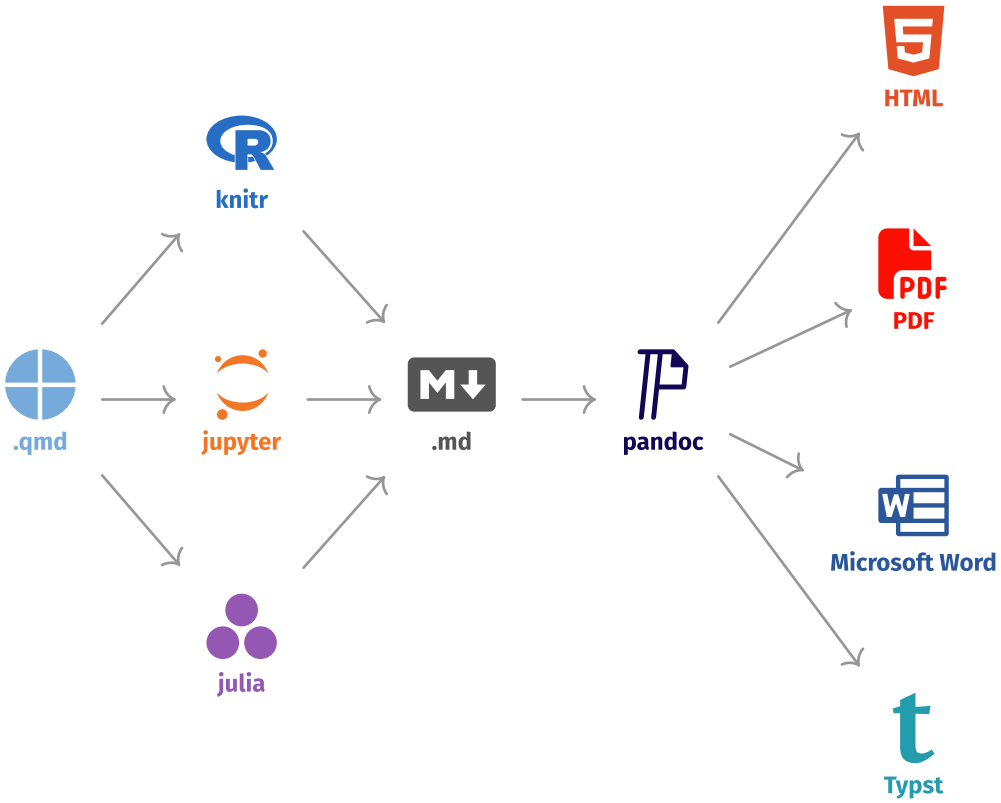
A.8.3 所有者を変える: chown

所有者の変更は `chown` で行います. 8 進数と記号の対応を覚える必要がない分, `chmod` より簡単です.

```
chown -R alice ABC      # make alice the owner of ABC
```

付録 B
Quarto

B.1　Quarto とは



Quarto はプログラミングコードを含んだレポートやスライドを作成するためのツールです. これによって, 研究者が再現可能なドキュメントを簡単に作成できるようになっています. Quarto は, `.qmd` という拡張子のファイルにあるコードを実行し, Markdown (`.md`) 形式のドキュメントを生成します. さらに, pandoc というツールを使って, HTML や PDF, Microsoft Word, Typst などの様々な形式に変換することができます.

B.2　Markdown 記法

コードの実行については後の章で説明しますが, まずは Markdown 記法について簡単に説明します.

B.2.1 Heading (見出し)

Input

```
# Heading 1


## Heading 2


### Heading 3
```

Output

Heading 1 Heading 2 Heading 3

`#` の個数で見出しのレベルが決まります. HTML のルールとして, `h1` はそのページのタイトルに使うことが推奨されているため, 通常は `h2` 以降を見出しとして使います. Quarto の場合は, 後に紹介する YAML フロントマターで指定したタイトルが `h1` として扱われるため, 基本的には `h2` 以降を使ってください.[30]

B.2.2 改行とパラグラフ

アカデミックな文章では, 改行は段落の区切りを意味します. Markdown では, パラグラフを明示的に区切るために, 空行を入れる必要があります. 逆に, markdown 上で改行しているだけでは, 出力上では改行されません.

Input

```
吾輩は猫である.
名前はまだない.


どこで生れたかとんと見当がつかぬ.
```

Output

吾輩は猫である. 名前はまだない.
どこで生れたかとんと見当がつかぬ.

B.2.3 強調

Markdown では必要最小限の以下の三つの装飾が用意されています.

Input

```
**太字** か __太字__ と書きます.


*斜体* か _斜体_ と書きます.
```

[30]論文の場合は, `h1` を節ごとに使うこともありますが, その場合は Quarto が `##` を `h1` に自動で変換するため, 基本的には `##` 以降を見出しとして使うと覚えておくと良いでしょう.

```
**太字かつ斜体**  か  ___太字かつ斜体___

~~打ち消し線~~  はあまり使いません.
```

Output

太字 か **太字** と書きます.

斜体 か 斜体 と書きます.

太字かつ斜体 か **太字かつ斜体**

~~打ち消し線~~ はあまり使いません.

B.2.4 リンク

Input

```
[Google](https://google.com) へのリンク
```

Output

Google へのリンク

B.2.5 箇条書き

番号付きでないものは - や * を使って箇条書きにできます. ネストも可能です.

Input

```
- 項目 1
- 項目 2
    - 項目 2.1
    - 項目 2.2
- 項目 3
```

Output

- 項目 1
- 項目 2
    - 項目 2.1
    - 項目 2.2
- 項目 3

番号つきのものは, 数字を使います. なお, 全ての行で `1.` と書いても, 出力上では正しい番号になります.

Input

```
1. 項目 1
1. 項目 2
    1. 項目 2.1
    1. 項目 2.2
1. 項目 3
```

Output

1. 項目 1
2. 項目 2
    1. 項目 2.1
    2. 項目 2.2
3. 項目 3

B.2.6 図の挿入

Input

```
![図のタイトル](path/to/image.png)
```

Output

図 B.1: 図のタイトル

ここでパスという概念が重要になります. Quarto では, ドキュメントのルートディレクトリを基準にして画像ファイルを指定します. 基本的には, `.qmd` ファイルと同じ階層に `img` というフォルダを作り, そこに画像ファイルを入れるのが一般的です. 例えば, `report.qmd` と同じ階層に `img/hokusai_kanagawa.jpg` というファイルがある場合は, `![図のタイトル](img/hokusai_kanagawa.jpg)` と書くことで画像を挿入できます.

B.2.7 表の挿入

Input

```
| Default | Left | Right | Center |
|---------|:-----|------:|:------:|
| 12      | 12   |    12 |   12   |
| 123     | 123  |   123 |  123   |

: 表のタイトル
```

Output

Default	Left	Right	Center
12	12	12	12
123	123	123	123

Markdown では, `|` と `-` を使って表を作ることができます. さらに, `:` を使うことで, 列の位置を指定することもできます. デフォルトでは, 列は全て左寄せになりますが, `:---` と書くと左寄せ, `---:` と書くと右寄せ, `:---:` と書くと中央寄せになります. GUI ツールで表を作って markdwon 形式でエクスポートするサービスとして, Tables Generator などがあります.

B.3 LaTeX Math 記法

Markdown では LaTeX という組版ソフトの数式記法を使って数式を表現することができます. 数式は, `$ ... $` で囲むとインライン数式, `$$ ... $$` で囲むとブロック数式 (改行して中央揃え) になります.

Input

```
$e^{i\pi} + 1 = 0$ は最も美しい数式だ.
```

Output

$e^{i\pi} + 1 = 0$ は最も美しい数式だ.

B.4 Input

```
ラマヌジャンの円周率公式は次の通り:
```

```
$$
\frac{1}{\pi}=\frac{2 \sqrt{2}}{99^2}
\sum_{n=0}^{\infty} \frac{(4 n)!}{(n!)^4}
\frac{26390 n+1103}{396^{4 n}}
$$
```

B.5 Output

ラマヌジャンの円周率公式は次の通り:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{99^2}\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4}\frac{26390n+1103}{396^{4n}}$$

B.6 Quarto 記法

ここからは Quarto 独自の記法についての説明です. 引用と文献情報については 文献と引用 の章で説明します.

B.6.1 Front Matter

```
---
title: タイトル
author: 著者名
date: 2026-04-01
format: typst
---

ここから本文
```

Quarto のドキュメントでは, 冒頭に `---` で囲まれた YAML 形式のフロントマターを置くことができます. ここには, ドキュメントのタイトルや著者名, 日付などのメタデータを記述することができます. また, `format` というフィールドで, 出力形式を指定することもできます. 例えば, `format: typst` と書くと, Typst 形式で出力されるようになります.

B.6.2 相互参照 (Cross Reference)

スライドでは必須ではありませんが, レポートなどを書く場合は, 図表番号や節番号を自動で管理できると便利です. Quarto では, `fig-xxxx`, `tbl-xxx`, `sec-xxx` などの特別なラベルを貼ることで, それらの番号を自動で管理することができます.

B.7 Input

```
![神奈川沖浪裏](img/hokusai_kanagawa.jpg){#fig-kanagawa}

@fig-kanagawa は葛飾北斎の有名な浮世絵である.
```

B.8 Output

図 B.2: 神奈川沖浪裏

図 B.2 は葛飾北斎の有名な浮世絵である.

B.9 Input

```
| fruit  | price  |
|--------|--------|
| apple  | 2.05   |
| pear   | 1.37   |
| orange | 3.09   |

: Fruit prices {#tbl-fruit}
```

```
@tbl-fruit は果物の値段を表している.
```

B.10 Output

表 B.1: Fruit prices

fruit	price
apple	2.05
pear	1.37
orange	3.09

表 B.1 は果物の値段を表している.

基本的には, 特別な prefix (`fig-`, `tbl-`, `sec-` など) をつけたラベルを `{#label}` という形式で指定し, 本文中では `@label` と書くことで, その要素の番号を自動で参照することができます. その他の要素 (式や命題など) も同様の方法で参照できるので Quarto のドキュメント を参照してください.

B.11 コードの実行

Quarto の最大の特徴は, ドキュメントの中にコードを埋め込んで実行できることです. コードは, ```` ```{r} ```` で始まり ```` ``` ```` で終わる コードチャンク (code chunk) の中に書きます. コンパイルのたびに Quarto がそのコードを実行し, コードと結果をドキュメントに挿入します.[31]

````
```{r}
1 + 1
```
````

```
## [1] 2
```

チャンクの冒頭にある `#|` で始まる行は, チャンクオプション (chunk option) です. YAML 形式で, そのチャンクの挙動を細かく制御します. 例えば次のチャンクには `label` と `eval` の 2 つのオプションが付いています.

````
```{r}
#| label: install-example
#| eval: false
install.packages("ggplot2")
```
````

`eval: false` を指定しているので, このコードは表示されるだけで実行されません. `install.packages()` のように, コンパイルのたびに走らせたくないコードを載せるときに便利です. 実

[31] ここでは `{r}` と書いて R 言語を使っていますが, `{python}` や `{julia}` に変えれば, それぞれの言語のコードも実行できます. 一つのドキュメントの中で複数の言語を混在させることも可能です.

際, 上のチャンクには結果が表示されていないことを確かめてください. よく使うオプションは次の通りです.

- `label`: チャンクにつける名前. 後述の相互参照で使います.
- `echo`: コード自体を出力に表示するか (`true` / `false`). 結果だけ見せたいときは `false` にします.
- `eval`: コードを実行するか. `false` にすると, 上の例のように, コードは表示されるだけで実行されません.
- `include`: コードも結果も出力に含めるか. `false` にすると, 実行はされますが何も表示されません (パッケージの読み込みなど, 裏方の処理に使います).
- `warning` / `message`: 警告やメッセージを表示するか.

B.12　コードと相互参照

前の「相互参照」の節では, 手で書いた画像や Markdown の表に番号を振りました. コードが生成する図や表にも, 同じ仕組みがそのまま使えます.

B.12.1　`ggplot2` による図

コードで生成する図には, チャンクオプションの `#| label:` を `fig-` で始め, `#| fig-cap:` でキャプションを付けます. 図そのものに `{# ... }` を付ける必要はありません.

````
```{r}
#| label: fig-penguins
#| fig-cap: "くちばしの長さとひれの長さの関係"
penguins |>
  ggplot(aes(x = flipper_len, y = bill_len, color = species)) +
  geom_point() +
  labs(
    x = "Flipper length (mm)",
    y = "Bill length (mm)",
    color = "Species"
  )
```
````

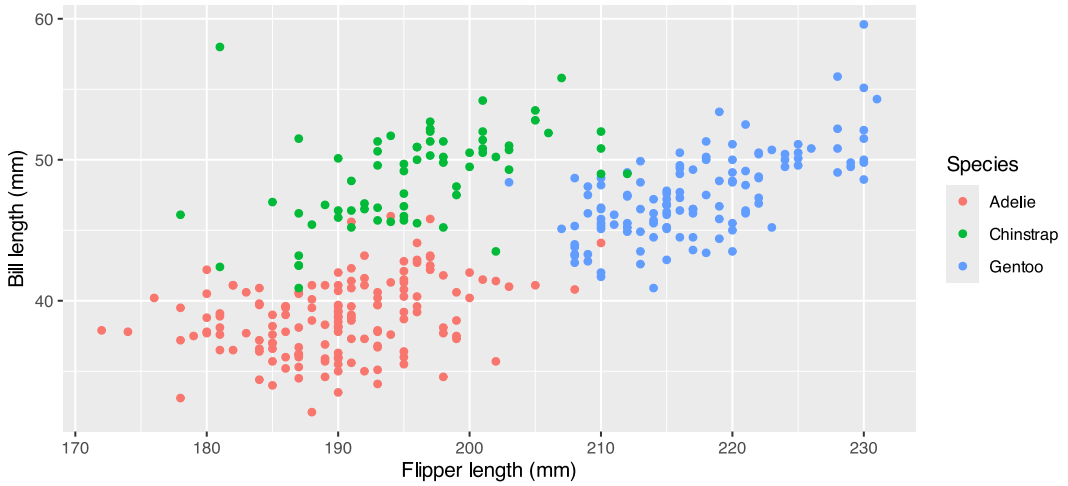


図 B.3: くちばしの長さとひれの長さの関係

本文で `@fig-penguins` と書くと 図 B.3 のように参照できます. 図番号は出現順に自動で振られるので, 図を増やしたり順番を入れ替えたりしても番号がずれません.

B.12.2 `tinytable` による表

表も同様に, チャンクラベルを `tbl-` で始め, `#| tbl-cap:` でキャプションを付けます. `tinytable` では `tt(caption = ...)` よりも, このチャンクオプションを使う方が相互参照が確実に効きます.

```
```{r}
#| label: tbl-penguins
#| tbl-cap: "ペンギンの種類ごとの平均値"
penguins ▷
  summarize(
    bill_len = mean(bill_len, na.rm = TRUE),
    flipper_len = mean(flipper_len, na.rm = TRUE),
    .by = species
  ) ▷
  tt()
```
```

表 B.2: ペンギンの種類ごとの平均値

species	bill_len	flipper_len
Adelie	38.79139	189.9536
Gentoo	47.50488	217.1870
Chinstrap	48.83382	195.8235

本文で `@tbl-penguins` と書くと 表 B.2 のように参照できます.

B.13　インラインコード

文章の途中に計算結果を埋め込みたいときは, インラインコード (inline code) を使います. バッククォートの中で `r` に続けて式を書くと, コンパイル時にその評価結果が文章に挿入されます.

Input

```
円周率はおよそ `r round(pi, 2)` です.
```

Output

円周率はおよそ 3.14 です.

数値を直接書く代わりにインラインコードで書いておくと, データが変わっても本文中の数値が自動で更新されます. 例えば「ペンギンのサンプルサイズは 344 である」のように書いておけば, データを差し替えたときに数値が追従するので, 書き間違いや更新漏れを防げます.

付録 C
用語解説

C.1　環境変数

C.1.1　環境変数とは

環境変数 (environment variable) は, 実行中のプロセスに紐づく「名前と値」の組です. OS やシェルが, プログラムに設定や実行時の文脈を渡すために使います. 例えば `HOME` はホームディレクトリの場所を, `LANG` は言語・文字コードの設定を, `USER` はログインユーザー名を保持しています. そして, 次に説明する `PATH` も環境変数の一つです.

シェルでは, 変数名の前に `$` を付けて値を参照できます. `echo $HOME` で自分のホームディレクトリが表示され, `printenv` (または `env`) で今のシェルが持つ環境変数の一覧が見られます. 新しく設定するには `export NAME=value` とします.

```
echo $HOME          # print one variable
printenv            # list all environment variables
export API_KEY=xxx  # set a variable for this shell and its child processes
```

環境変数の重要な性質が, 子プロセスへの継承です. あるプロセスが別のプロセスを起動すると, 環境変数はコピーされて引き継がれます. だから, シェルで `export` した変数は, そのシェルから起動したプログラム (R や Quarto など) からも見えます. API の章で, `.Renviron` に書いた `ESTAT_APP_ID` を R の `Sys.getenv()` で読み出せたのは, R が起動時にそれを環境変数として読み込み, 自分のプロセスの環境に入れているからです.

ただし環境変数はプロセス単位のもので, 別のシェルや再起動後には残りません. 恒久的に設定したいときは, 設定ファイルに書いておきます (後述).

C.1.2　コマンドはどこから実行されるのか

シェルで `ls` と打つと, ファイル一覧を表示するプログラムが動きます. しかしシェルは, `ls` というプログラムがディスク上のどこにあるのかを, どうやって知るのでしょうか.

その答えが環境変数 `PATH` です. `PATH` は, 実行ファイルを探すディレクトリを `:` で区切って並べたリストです. 例えば次のようになっています.

```
echo $PATH
# /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

コマンド名を打つと, シェルはこのリストの先頭から順にディレクトリを見ていき, 同じ名前の実行ファイルが最初に見つかった場所のものを実行します. 実際にどのファイルが使われるかは, `which` (または `type`) で確認できます.

```
which ls
# /bin/ls
```

つまり, `ls` と打つのは, 実質的に `/bin/ls` というファイルを実行しているということです (図 C.1). `git` や `quarto`, `R` といったコマンドも同じで, `PATH` のどこかのディレクトリに置かれた実行ファイルが呼び出されています. コマンドは魔法で動いているのではなく, ディスク上の実行ファイルを名前で探して起動しているだけなのです.

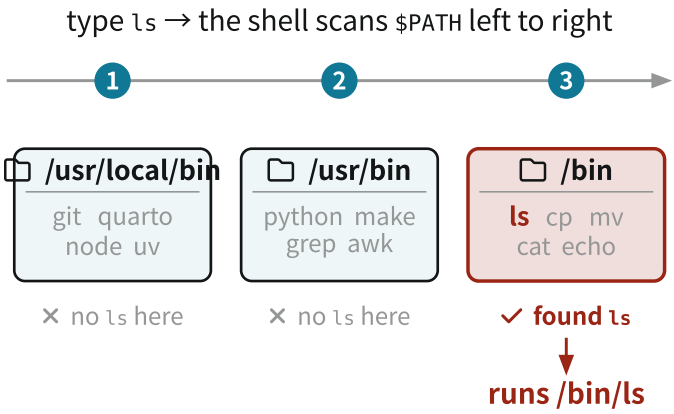


図 C.1: シェルが PATH からコマンドを探す流れ

リストの先頭から順に探すので, 同じ名前のコマンドが複数の場所にあると, `PATH` で前に並んでいるディレクトリのものが優先されます. また, コマンドを打って `command not found` と出るのは, 「そのプログラムが存在しない」とは限らず,「`PATH` のどのディレクトリにも見つからない」だけのこともあります.

C.1.3　パスを通すとは

新しいプログラムをインストールしても, その実行ファイルの置き場所が `PATH` に含まれていなければ, シェルは名前だけでは見つけられません. その場合, 毎回フルパス (例えば `/opt/foo/bin/foo`) で打つ必要があり, 面倒です.

「パスを通す」とは, その置き場所のディレクトリを `PATH` に加えて, コマンド名だけで呼び出せるようにすることです. 次のように, 既存の `$PATH` に新しいディレクトリを足します.

```
export PATH="/opt/foo/bin:$PATH"   # prepend: the new directory takes priority
```

先頭に足すと新しいディレクトリが優先され, 末尾 (`export PATH="$PATH:/opt/foo/bin"`) に足すと既存のものが優先されます.

ただし, この `export` はそのシェルの中だけで有効で, 新しいシェルを開くと消えてしまいます. 恒久化するには, シェルの設定ファイル (`~/.zshrc` や `~/.bashrc` など, シェルを開くたびに読み込まれる

ファイル) に同じ行を書いておきます. すると, シェルを開くたびに読み込まれて, いつでもそのコマンドが使えるようになります.

R でも同じことが起こります. R は起動時に `.Renviron` を読んで環境変数を設定します. targets による再現性 の章では, R の中から Julia を呼ぶために, `.Renviron` に `PATH_JULIA= ...` と書き, `.Rprofile` で `Sys.setenv(PATH = paste(Sys.getenv("PATH_JULIA"), Sys.getenv("PATH"), sep = ":"))` として R プロセスの `PATH` に足していました. これも, R のプロセスに対して「パスを通す」操作にほかなりません.

C.2　コンパイラ

C.2.1　コンパイラとは

コンパイラ (compiler) は, 人が書いたソースコードを, CPU が直接実行できる機械語 (セクション 1.3 で触れた機械語命令) に翻訳するプログラムです. C, C++, Fortran, Rust, Go などは, 実行前にコンパイラでまとめて機械語に変換してから動かします. このような言語をコンパイル型言語とよびます.

これと対になるのがインタプリタ (interpreter) で, ソースコードをその場で 1 行ずつ読んで実行します. R や Python はこのインタプリタ型の言語です. あらかじめ機械語に変換しておくコンパイル型は, 変換に時間がかかるかわりに実行が速く, その場で解釈するインタプリタ型は, 手軽で対話的に使えるかわりに実行は遅くなりがちです.

C.2.2　R とコンパイラ

R 本体はインタプリタ型ですが, 速度が要る部分では, 多くのパッケージが内部に C や C++, Fortran のコードを抱えています (Rcpp がその代表的な仕組みです). こうしたパッケージをソースからインストールするとき (チャプター 2), その過程でコンパイラが走り, C/C++ のコードを R が読み込める機械語の共有ライブラリ (`.so` や `.dll`) に変換します.

そのため, パッケージをソースからビルドするにはコンパイラ一式 (ツールチェーン) が必要です. Debian や Ubuntu では `build-essential` が C/C++ コンパイラ (gcc / g++) と make をまとめて入れ, Fortran には別途 gfortran が要ります. macOS では Xcode Command Line Tools (clang), Windows では Rtools が同じ役割を担います. 一方, コンパイル済みのバイナリパッケージ (チャプター 2) を使えば, この変換は済んでいるので, コンパイラなしで速く導入できます.

C.3　ハッシュ

C.3.1　ハッシュ関数

ハッシュ関数 (hash function) とは, 任意の長さのデータを受け取り, 決まった長さの値へと変換する関数のことをいいます. 入力は何バイトでも構いませんが, 出力の長さは入力によらず常に一定です. この出力をハッシュ値とよびます (セクション C.3.2).

データをビット列とみなすと, ハッシュ関数は次のように書けます.

i ハッシュ関数

ハッシュ関数とは, 任意長のビット列の集合 $\{0,1\}^*$ から, 固定長 n ビットの集合 $\{0,1\}^n$ への写像

$$h : \{0,1\}^* \to \{0,1\}^n$$

のことをいう.

定義域 $\{0,1\}^*$ は長さに上限のないビット列全体からなる無限集合であるのに対し, 値域 $\{0,1\}^n$ の要素数は 2^n 個と有限です. たとえば Git が標準で用いる SHA-1 では $n = 160$ で, 取りうるハッシュ値は 2^{160} 通りあります.

良いハッシュ関数, とくに Git のように改ざん検知を目的とする暗号学的ハッシュ関数 (cryptographic hash function) には, 次のような性質が求められます.

- **決定性** (deterministic): 同じ入力からは必ず同じハッシュ値が返ります.
- **高速性**: 入力からハッシュ値を計算するのは高速です.
- **なだれ効果** (avalanche effect): 入力をほんの少し変えただけで, 出力は予測できないほど大きく変わります. 1 ビットの違いでも, 出力のおよそ半分のビットが入れ替わるのが理想です.
- **一方向性** (preimage resistance): ハッシュ値から, それを生んだ入力を逆算するのは事実上不可能です.
- **衝突困難性** (collision resistance): 同じハッシュ値になる異なる入力の組 (衝突) を意図的に見つけるのは事実上不可能です.

衝突困難性について補足します. 入力は無限に作れるのに出力は有限なので, 異なる入力が同じハッシュ値になる組 (衝突) は, どんなハッシュ関数にも必ず存在します (鳩の巣原理). つまり衝突困難性とは, 衝突が存在しないという意味ではなく, 衝突を意図的に見つけ出すのが現実的な時間では不可能, という意味です.

偶然の衝突についても直感を持っておきましょう. 誕生日のパラドックス (birthday problem) として知られるように, 取りうるハッシュ値の種類の, およそ平方根くらいの個数のデータを集めると, どれかが偶然衝突し始めます. SHA-1 は 2^{160} 通りなので, その平方根はおよそ 2^{80} (10 進で 10^{24} 程度) という天文学的な数です. ふだん Git を使っている限り, 偶然ハッシュが衝突してコミットが取り違えられる心配はまずありません.

ただし, ここで「心配ない」と言えるのは, ハッシュ値が一様ランダムに散らばると仮定したときの, 偶然による衝突に限った話です. これとは別に, ハッシュ関数の内部構造の弱点を突いて, 同じハッシュ値をもつデータを意図的に作り出す攻撃があります. こうした攻撃では, 上で見積もった $2^{n/2}$ という壁を大きく下回る計算量で衝突を作れてしまうことがあり, 実際 SHA-1 はこの意味ですでに破ら

[32] 王小云 (Wang Xiaoyun) のグループは, まず 2004 年に, SHA-1 より古い MD5 などについて, 実際に同じハッシュ値となる具体的な入力対 (数値例) を構成してみせました (Wang ほか 2004 年). 続く 2005 年には, 同グループが SHA-1 についても, 衝突探索の計算量を総当たりの 2^{80} より小さい 2^{69} 程度まで下げられることを理論的に示します (Wang ほか 2005 年) そして 2017 年, Google と CWI のグループが, 実際に同じ SHA-1 ハッシュをもつ 2 つの PDF を構成してみせ, 初めて現実の SHA-1 衝突を実証しました (SHAttered と呼ばれます) (Stevens ほか 2017 年).

れています[32]. そのため, 悪意ある改ざんまで防ぎたい用途では SHA-1 はもはや安全とは言えず, Git でも後継の SHA-256 ($n = 256$) への移行が進められています.

C.3.2　ハッシュ値

ハッシュ値 (hash value) とは, ハッシュ関数がデータに対して返す固定長の出力のことです. 同じデータからは必ず同じハッシュ値が得られ, 中身が少しでも違えばまったく異なるハッシュ値になるため, ハッシュ値はデータの中身そのものから決まる一意な「指紋」として使えます.

実際の表示では, n ビットの値をそのまま 0 と 1 の羅列で見せても扱いにくいので, 4 ビットを 1 文字に対応させた 16 進数 (`0-9`, `a-f`) で書くのが一般的です. SHA-1 の 160 ビットは $160/4 = 40$ 桁の 16 進数になります. Git のコミットに割り当てられる `a84d0e9…` のような名前は, まさにこのハッシュ値で, 普段はその先頭の 7 文字ほどを取り出して使います.

C.4　グラフィックデバイス

C.4.1　グラフィックデバイスとは

R で `plot()` や `ggplot2` の図を描くとき, 点や線, 文字を実際の画面やファイルに変換しているのは, グラフィックデバイス (graphics device) とよばれる仕組みです. R 本体は「この座標に文字列を書く」「ここからここまで線を引く」といった抽象的な描画命令を発行するだけで, それをピクセルや PDF のオブジェクトとして書き出すのは各デバイスの仕事です. そのため, 同じコードでも, どのデバイスで描くかによって出力の性質が変わります.

デバイスは大きく分けて 2 種類あります.

- **スクリーンデバイス**: ウィンドウに直接描画するデバイスです. macOS の `quartz`, Windows の `windows`, Linux の `x11` などがあり, RStudio の Plots ペーンに表示される図もこの一種が描いています.
- **ファイルデバイス**: 図をファイルとして書き出すデバイスです. ラスタ形式の `png()` や `jpeg()`, ベクタ形式の `pdf()`, `cairo_pdf()`, `svg()` のほか, パッケージが提供する `ragg::agg_png()` や `svglite::svglite()` などがあります.

Quarto (knitr) で文書をレンダリングするときは, チャンクごとにファイルデバイスが開かれ, 図が一度ファイルとして書き出されてから文書に埋め込まれます. どのデバイスを使うかはチャンクオプション `dev` で指定でき, この資料では SVG デバイス (`dev: svg`) を使っています.

C.4.2　デバイスとフォント

文字の描画もデバイスの仕事であるため, 指定されたフォント名を実際のフォントファイルにどう対応づけるかは, デバイスごとに異なります. 代表的なデバイスの挙動は次のとおりです.

- **`pdf()`**: R 標準の PDF デバイスですが, フォントの扱いは最も原始的です. システムのフォントを自動では参照せず, デバイスに登録された Type 1 フォント (日本語は `Japan1` などの CID フォント) しか使えません. さらにデフォルトではフォントを埋め込まず, PDF にはフォント名だけが書

き込まれるため, 閲覧環境に同じフォントがなければ表示が崩れます. 埋め込むには Ghostscript 経由の `embedFonts()` が別途必要です.

- **cairo 系 (`cairo_pdf()`, `svg()` など)**: fontconfig というライブラリを通じてシステムフォントを検索します. `cairo_pdf()` は使用したフォントをサブセット化して埋め込むため `pdf()` よりずっと扱いやすいですが, フォントの解決は OS 側の設定に依存します.
- **quartz (macOS)**: macOS のフォント機構をそのまま使うため, Mac 上ではヒラギノなども問題なく使えますが, 他の OS では動きません.
- **ragg (`agg_png()` など)**: `systemfonts` パッケージ経由でシステムフォントを解決する現代的なラスタデバイスです. インストール済みフォントの検索が最も堅牢で, ウェイトの指定などにも対応します.

つまり, システムに目的のフォントが入っていることと, いま使っているデバイスがそのフォントを見つけて出力に反映できることは, 別の問題です.「フォントを指定したのに豆腐化する」「手元では正しく表示されるのに, 共有した PDF では崩れる」といった現象は, ほとんどがこのデバイス側のフォント解決と埋め込みの問題に由来します.

C.4.3 showtext の仕組み

`showtext` パッケージは, このデバイス依存性を根本から回避します. テキストの描画命令をフックし, `sysfonts` で登録されたフォントファイルから各文字のグリフ (字形) を取り出して, 文字をアウトライン (曲線) として描画します. 出力ファイルには文字ではなく図形が書き込まれるため, デバイスがフォントを解決できるかどうかも, 閲覧環境にフォントがあるかどうかも, 一切関係なくなります.

副作用もあります. 文字が図形化されるため, PDF や SVG の中でテキストの選択・検索ができなくなり, ファイルサイズも増えます. また, ラスタデバイスでは DPI の解釈が R 本体とずれるため, `showtext_opts(dpi = ...)` を出力の解像度 (Quarto の `fig-dpi`) に合わせる必要があります.

`showtext_auto()` は, このフックを以後に開かれるすべてのデバイスで自動的に有効化する関数です. knitr はチャンクごとに新しいデバイスを開くため, これを最初に一度呼んでおかないと, 図ごとに `showtext_begin()` と `showtext_end()` で囲むことになります.

C.4.4 showtext を使わない選択肢

`showtext` の対極にあるのは, 文字をアウトライン化せず, フォントを正しく解決できるデバイスを出力形式ごとに選ぶ, という方針です.

- **PDF**: `cairo_pdf()` を使います. システムフォントを解決し, サブセット化して埋め込むため, 文字はテキストとして残り, 検索・コピーもできます. 論文に載せる図はこれが標準的な選択です.
- **SVG**: `svglite::svglite()` は文字を `<text>` 要素のまま残します. ただし SVG 自体はフォントを埋め込まないため, 表示には閲覧環境側に同じフォントが必要です (ウェブサイトなら CSS で Web フォントを読み込んで補います). 一方, cairo ベースの `svg()` は文字をパスに変換するため, 検索できない代わりに閲覧環境へのフォント要求がなく, この点では showtext と似た性質になります.
- **ラスタ (PNG など)**: `ragg::agg_png()` が `systemfonts` 経由でシステムフォントを堅牢に解決します. ラスタ画像では文字はピクセルになるので, 選択・検索の概念はそもそもありません.

この方針の弱点は，レンダリングするマシンに目的のフォントがインストールされていることが前提になる点で，コードを共有したときの再現性の問題は残ります．`showtext` が本領を発揮するのは，まさにこの前提を外したい場合です．すなわち，`font_add_google()` でフォントのインストール自体をコードに閉じ込めたい場合や，`pdf()` を含むあらゆるデバイスで同じ見た目を保証したい場合には `showtext` を，文字をテキストとして残すことが重要な場合には上記のデバイスを選ぶ，という使い分けになります．

C.5 MCP (Model Context Protocol)

C.5.1 MCP とは

MCP (Model Context Protocol) は，AI アシスタント (Claude など) のようなアプリケーションを，外部のツールやデータソースに接続するためのオープンな規格です．2024 年に Anthropic が公開し，仕様は modelcontextprotocol.io で公開されています．特定の企業の製品ではなく公開された規格なので，誰でも MCP に対応したサーバーを作れますし，MCP に対応したアプリケーション (Claude Desktop, Claude Code, VSCode など) ならどれからでも使えます．

MCP が解決するのは，「AI アシスタントごと・ツールごとに，個別の接続方法を実装しなければならない」という手間です．AI に Zotero・ファイル・データベースの 3 つを触らせたいとき，MCP が無ければ，アプリと AI の組み合わせごとに専用の接続を書くことになります．MCP は，この接続部分に共通の「話し方」を定めます．USB が，機器ごとに違うケーブルを不要にしたのと同じ発想です．ツールを提供する側は MCP に従ってサーバーを 1 つ作れば，MCP に対応したどの AI アプリからも使えるようになります．

C.5.2 クライアント・サーバーモデル

MCP の構造は，セクション 6.1 で見たクライアント・サーバーモデルとよく似ています．登場人物は 3 つです．

- ホスト (host): ユーザーが直接操作する AI アプリケーションです．Claude Desktop や，ターミナルで動く Claude Code などがこれにあたります．
- クライアント (client): ホストの内部にあり，1 つのサーバーとの接続を担当する部品です．ホストは，使いたいサーバーの数だけクライアントを持ちます．
- サーバー (server): 特定のツールやデータへの窓口です．Zotero MCP サーバーなら Zotero のライブラリへの窓口になります．

ユーザーがホスト (Claude) に「Zotero でこの論文を探して」と頼むと，ホストは対応するサーバーにリクエストを送り，サーバーは Zotero を実際に操作して結果を返します．AI はその結果を読んで回答に使います．AI 自身が Zotero を直接触るのではなく，あいだにサーバーを挟むことで，何ができるか・何をしてよいかがサーバー側で明確に管理される点が重要です．

C.5.3 サーバーが提供するもの

MCP サーバーがホストに提供できるものは，大きく 3 種類に分かれます．

- ツール (tools): AI が実行できる操作です. 「ライブラリを検索する」「論文の全文を取得する」といった, 引数を受け取って結果を返す関数だと考えてください.
- リソース (resources): AI が読み取れるデータです. ファイルの中身やデータベースのレコードなど, 参照用の情報を指します.
- プロンプト (prompts): 定型的な指示のテンプレートです. よく使う依頼を, あらかじめ形にして呼び出せるようにしたものです.

このうち, 実際の研究作業でよく使うのはツールです. AI がユーザーの依頼を解釈し, 必要なツールを自分で選んで呼び出し, 返ってきた結果を使って作業を進めます.

C.5.4 トランスポート: ローカルとリモート

ホストとサーバーが通信する経路 (トランスポート) には, 大きく 2 種類あります.

- 標準入出力 (stdio): サーバーを自分の PC 上のプロセスとして起動し, その入出力を通じて通信します. サーバーが手元で動くので, ローカルの Zotero やファイルにアクセスするのに向いています.
- HTTP: サーバーがネットワーク上の別の場所で動き, HTTP を通じて通信します. クラウド上のサービスを MCP サーバーとして公開する場合などに使われます.

ローカルの stdio 方式では, サーバーはユーザーの権限で PC 上で動きます. したがって, 信頼できるサーバーだけを追加することが大切です. 素性の分からない MCP サーバーを設定に加えると, PC 上のファイルや認証情報にアクセスされる恐れがあります. 導入するのは, ソースコードが公開され広く使われているサーバーに限るのが安全です. どのサーバーを許可するかはユーザーが設定で決められるので, AI にできることの範囲は自分でコントロールできます.

付録 D
色彩論

D.1　なぜダサいスライドが生まれるのか

BeamerやRのコードで色を指定するとき, 色に頓着のない人は原色を選んでしまいます. つまり, 純粋な赤 `#FF0000`, 緑 `#00FF00`, 青 `#0000FF` をそのまま使うことです. 色を選ぶ意思があっても, 色彩の知識がないと, ついカラーピッカーの `R, G, B` のスライダーを動かして適当に色を決めてしまいます. これらの結果は, うまく言語化できないにしろ, ダサいと感じるでしょう. これは, 人間が色から受ける印象が `R, G, B` という 3 つの数値に対して線形に対応していないからです. 分かりやすい例が明るさです. ここでいう純色とは, 赤・緑・青の各チャンネルが 0 か最大値 255 のいずれかだけでできた, 最も鮮やかな色のことです. 純色を並べてみると, 同じ最大の色でも, 感じる明るさは違います.

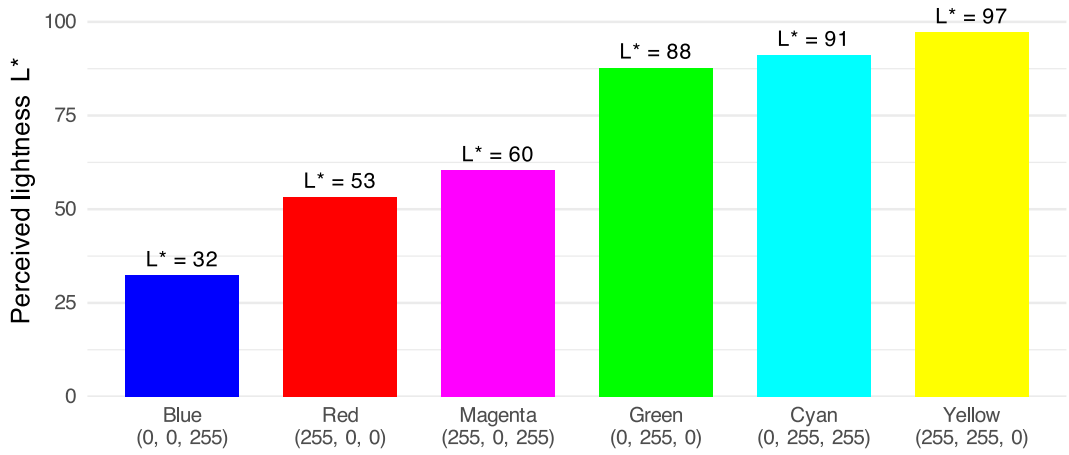


図 D.1: 純色の知覚明度 L*

図 D.1 の縦軸 `L*` は, 後で定義する知覚的な明度です. 純色の青の明度は 32 しかないのに, 純色の黄は 97 もあります. つまり原色をいくつか並べただけで, 明るさが大きくばらついた, ちらちらとまとまりのない配色ができあがります. これがダサさの正体です. さらに原色は彩度も最大なので, どの色も自己主張が強く, 図の主張 (チャプター 7) を邪魔します.

解決の方針はひとつです. 色を `R, G, B` ではなく, 人間が感じる 3 つの軸, すなわち明度 (lightness), 彩度 (saturation), 色相 (hue) で考えることです. こうすれば「明るさをそろえて色相だけ変える」といった, 知覚に沿った操作ができるようになります.

D.2　色の 3 要素とカラーマップ

色は, 色相 (hue), 彩度 (saturation), 明度 (lightness) という 3 つの知覚的な軸で捉えられます. 色相は色味そのもの (赤・黄・緑・青・紫…) で, 円環の角度で表します. 彩度は鮮やかさ, つまり灰色からどれだけ離れているかです. 明度は明るさです. まずはこの 3 つを, ひとつずつ動かして目で確かめましょう.

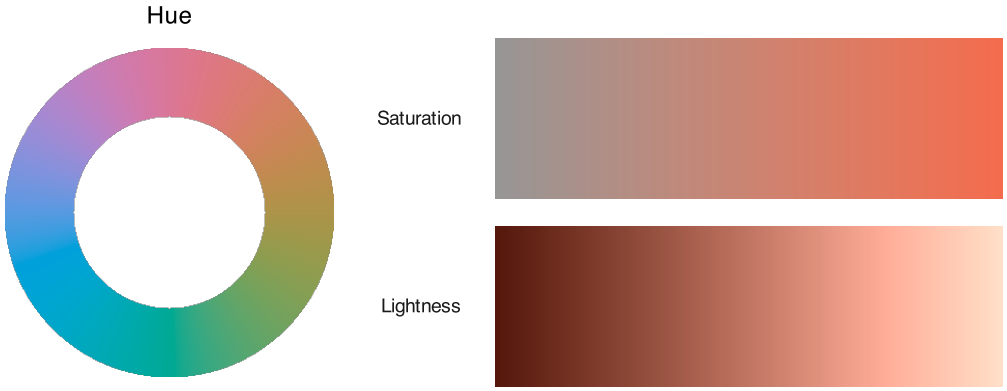


図 D.2: 色の 3 要素: 色相・彩度・明度

図 D.2 は, 3 つの軸のうち 2 つを固定して, ひとつだけを動かしたものです. 左の輪は色相環で, 明るさと鮮やかさを一定に保ったまま色味だけを一周させています. 色相が角度に対応することがよく分かります. 右上は色相を固定して彩度を上げたもので, 左の灰色から右の鮮やかなオレンジへ進みます. 右下は色相を固定して明度を上げたもので, 左の暗いところから右の明るいところへ進みます. どんな色も, この 3 つの数値の組で一意に決まります.

D.2.1　色空間: 3 要素の測り方

色を表す座標系を色空間 (color space) と呼びます. 目的に応じて使い分けるのが大切なので, まず代表的なものを整理します.

- **RGB / sRGB**: 赤・緑・青の光をどれだけ足すか, という加法混色の座標です. 画面が実際に光を出すしくみそのものであり, 画像の保存形式でもあります. 一方で, 数値と知覚が対応しないため, 色を「選ぶ」のには向きません.
- **HSL / HSV**: RGB を円筒座標に置き直したものです. 色相 (hue) を角度, 彩度 (saturation) と明度 (lightness) を半径・高さで表します. 直感的でピッカーに便利ですが, あくまで RGB の座標変換にすぎず, 知覚的には均等ではありません.
- **CIELAB / LCh**: 人間の色差の実験に基づいて設計された知覚均等 (perceptually uniform) な色空間です. 明度 L^*, 彩度 C^*, 色相 h の円筒座標が LCh で, これが実質的に HCL と呼ばれるものです. 距離が知覚的な色の違いに近いので, パレット設計に向きます.
- **OKLab / OKLCh**: CIELAB の弱点を近年改良した色空間で, グラデーションやパレット設計での挙動がより素直です. 考え方は LCh と同じ (L, C, h の 3 軸) です.

3 要素の定義そのものは素朴でした. 難しいのは, これらを「知覚的に均等」に測ることです. ここで色空間による差が出ます. HSL の明度と, 人が本当に感じる明度は一致しません. 実際, HSL で明度を

50% に固定し彩度を最大にして色相だけ一周させると, 感じる明るさは大きく波打ちます. 一方, 知覚均等な HCL で L^* を固定すれば, 色相を変えても明るさはほぼ一定に保てます.

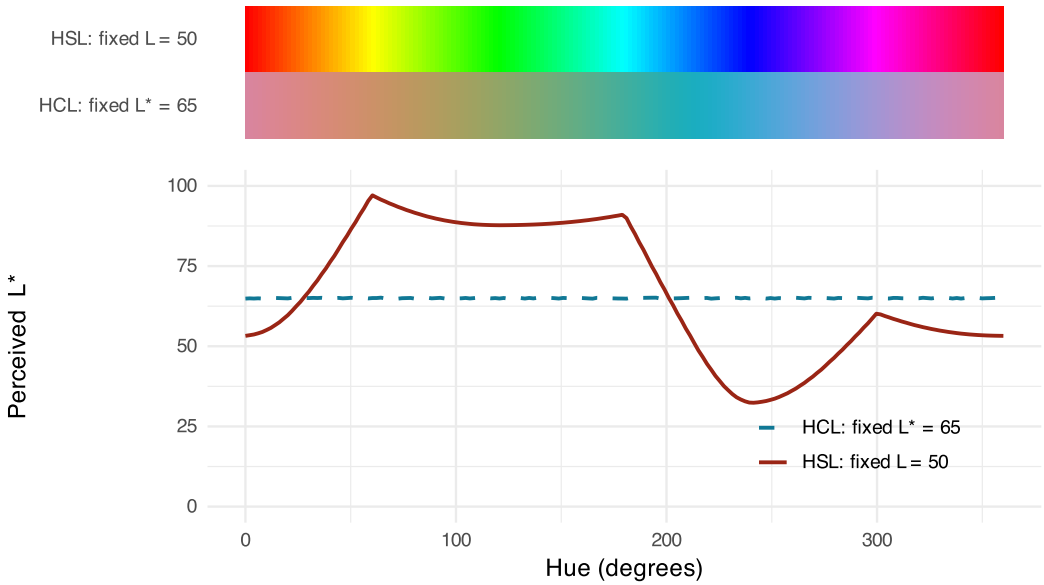


図 D.3: 同じ「明度」でも色相で知覚明度は動く

図 D.3 の上段が色帯, 下段がその知覚明度です. HSL の帯 (実線) は黄で明るく青で暗く, 明度が大きく上下します. HCL の帯 (破線) はほぼ平らで, どの色相でも明るさがそろっています. だからこそ, 質的な色分けをするときは HCL で明度と彩度を固定し, 色相だけを動かすのが定石になります.

ではもうひとつの軸, 色相はどうでしょうか. 3 つの空間はどれも色相を角度で表しますが, その角度の割り当ては同じではありません. 同じ色が, 空間によって別の角度に置かれます.

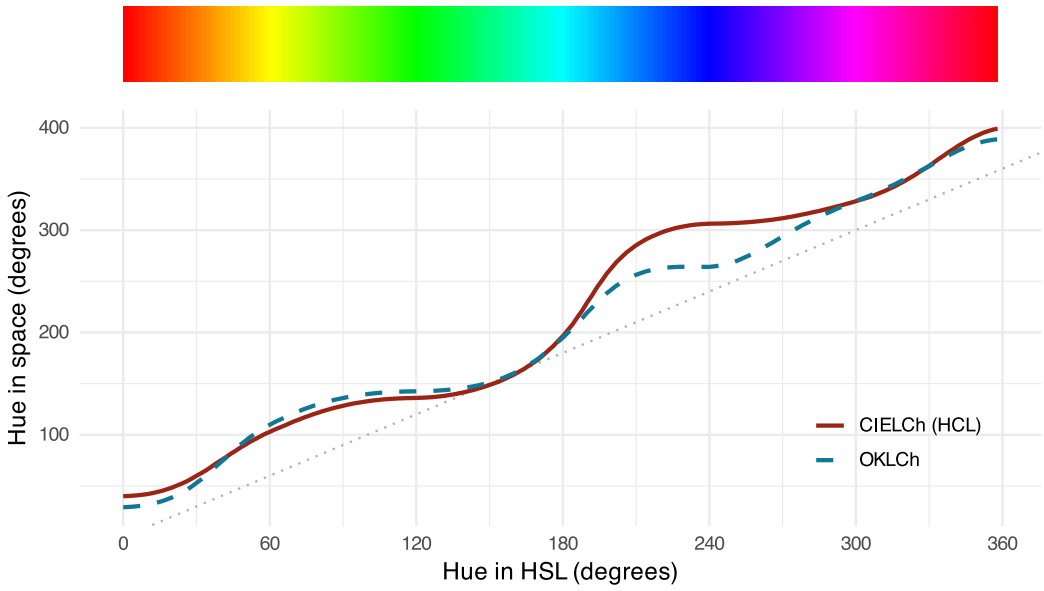


図 D.4: 色空間による色相角の違い

図 D.4 は, HSL で色相を一周させた各色が, CIELCh と OKLCh では何度に対応するかを示しています. 点線は「もし 3 つの空間で色相角が完全に一致するなら」という基準線です. まず, どちらの曲

線も基準線からずれています. HSL は原色を赤 0°, 緑 120°, 青 240° と機械的に等間隔で並べますが, 知覚的な空間はそうではありません. さらに注目すべきは青の領域 (HSL で 180° から 260° あたり) です. ここだけ CIELCh と OKLCh が大きく食い違います. これは CIELAB では青が紫の方向へ曲がってしまう有名な欠陥で, OKLab はまさにこれを補正するように設計されています. 赤や緑では両者はほぼ重なります.

明度と色相を見てきました. 残るは彩度です. これも HSL では素直ではありません. HSL の彩度を最大 (100%) に固定して色相を一周させると, 知覚的な鮮やかさ (chroma) は大きく上下します.

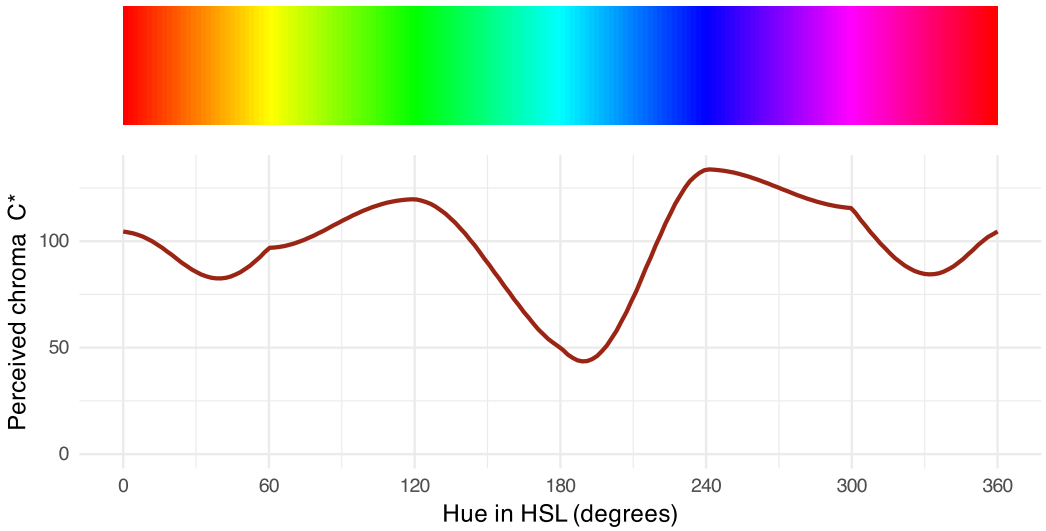


図 D.5: 同じ HSL 彩度でも知覚的な鮮やかさは動く

図 D.5 の上段は, どれも彩度 100% の色です. ところが下段の知覚的な彩度 C* を見ると, 最も鈍い水色のあたりでは 44 ほどしかないのに, 最も鮮やかな青のあたりでは 134 を超えます. 同じ「最大の彩度」でも, 感じる鮮やかさは色相によっておよそ 3.1 倍も違うのです. 一方 HCL や OKLCh では, この C* そのものを直接指定するので, 色相をまたいで鮮やかさをそろえられます.

まとめると, HSL の 3 つの数値はどれも知覚とずれています. 明度 (図 D.3) も彩度 (図 D.5) も色相で暴れ, 色相角の割り当ても空間で違います (図 D.4). そのため, パレット設計では, 3 軸すべてが知覚に沿う HCL か OKLCh を使い, 明度と彩度を固定して色相だけを動かすのが定石になります. 青の扱いまで素直な OKLCh を使えるなら, さらに安全です.

D.2.2　カラーマップ

連続量を色で表すときの色の並びをカラーマップ (colormap) と呼びます. 用途に応じて 4 つの類型があります.

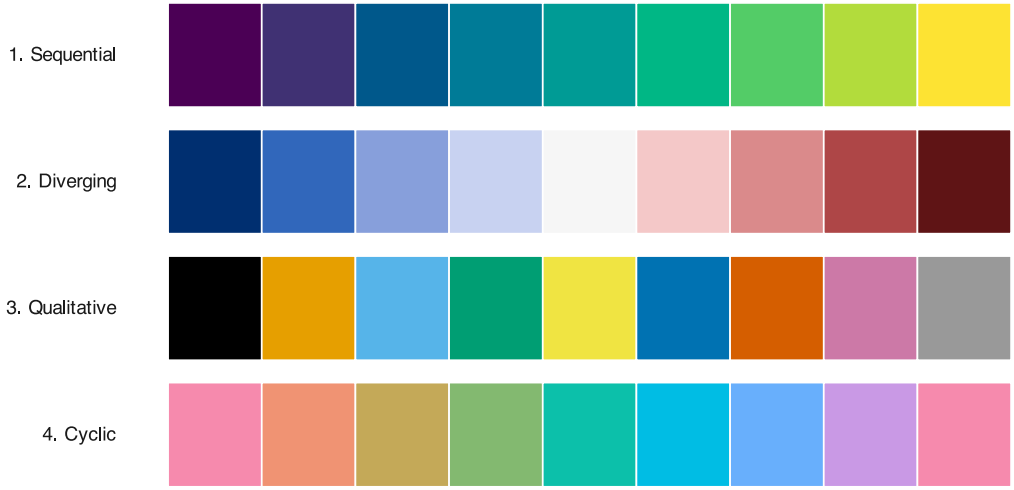


図 D.6:　カラーマップの 4 類型

- Sequential (連続): 低い値から高い値へ一方向に進むデータ (人口密度, 濃度など) に使います. 明度が単調に変化するのが条件です.
- Diverging (発散): 中央 (ゼロや平均) から両側に離れる量 (偏差, 相関など) に使い, 中央を淡く, 両端を濃くします.
- Qualitative (質的): 順序のないカテゴリ (国, 品種など) に使い, 明度と彩度をそろえて色相だけを変えます.
- Cyclic (循環): 角度や時刻のように端がつながる量に使い, 始点と終点の色を一致させます.

悪いカラーマップの代表が, かつてよく使われた虹 (rainbow, jet) です[33]. 見た目は派手ですが, 明度が単調でないため, データにない偽の境界 (false boundary) を作り出し, さらにグレースケール印刷や色覚異常で情報が失われます. これに対し viridis のような現代的なカラーマップは, 明度が単調に増加するよう設計されています.

[33]jet は長らく MATLAB の既定カラーマップでしたが, ここで述べる明度の欠陥のため, MATLAB 自身も 2014 年 (R2014b) に既定を parula へ切り替えました. Python の matplotlib も既定は jet でしたが, van der Walt と Smith が SciPy 2015 で発表した知覚均等な viridis を, バージョン 2.0 (2017 年) から既定に採用しました. viridis には同じ設計思想の姉妹マップ magma, inferno, plasma があり, のちに色覚異常向けに最適化された cividis も加わりました. R では viridisLite / viridis パッケージから使えます.

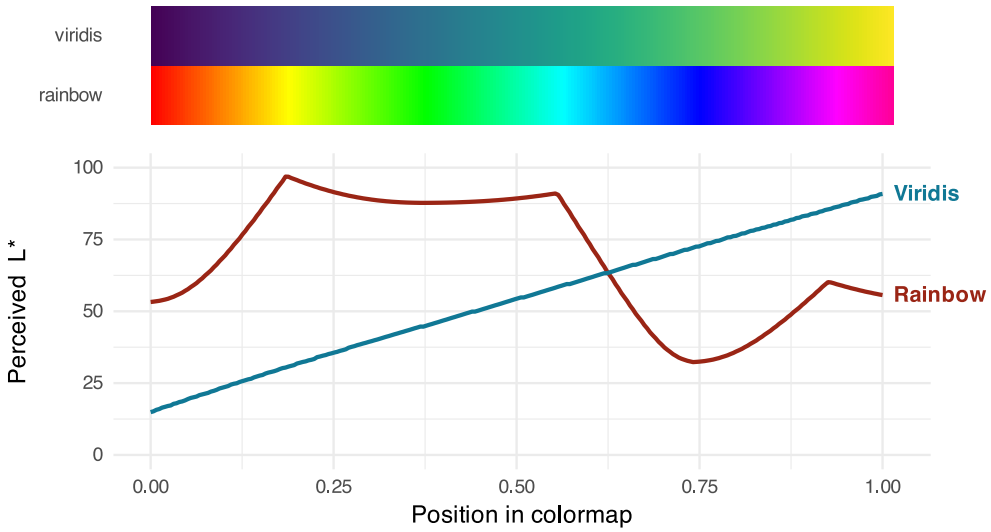


図 D.7: Rainbow と viridis の知覚明度プロファイル

図 D.7 の下段を見ると, 虹の明度はでこぼこで, 特に黄の位置に鋭いピークがあります. このピークが偽の等高線に見えてしまうのです. viridis の明度は右肩上がりの直線に近く, 値の大小がそのまま明るさの順序として読めます.

viridis はひとつのカラーマップではなく, 同じ設計思想をもつ姉妹マップの一群です. どれも明度が単調で, 色覚異常やグレースケールに強いという性質は共通で, 色調の好みや図の背景に合わせて選べます. これらは Python の matplotlib で提案・標準化され, R でも viridisLite パッケージ (この章の図もこれを使っています) から同じものが使えます.

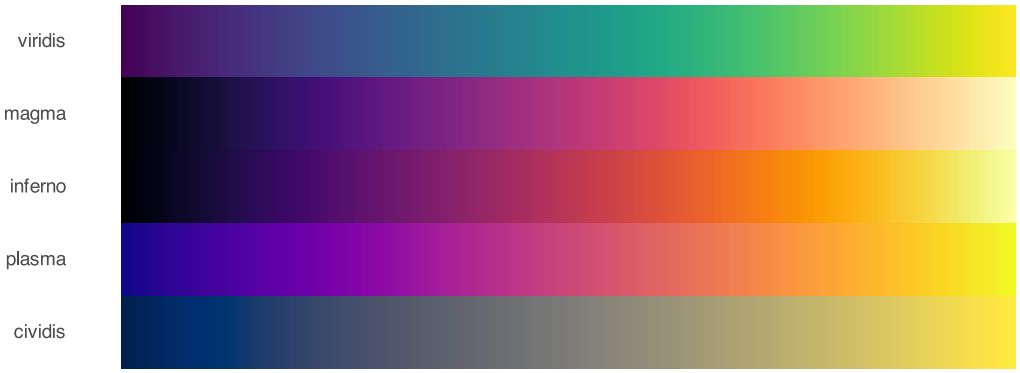


図 D.8: viridis 系のカラーマップ

D.3　色の選び方

原理はここまでで出そろいました. 質的なパレットが欲しいなら, 明度 L^* と彩度 C^* を固定し, 色相 h を等間隔にずらすだけです. 連続的なグラデーションなら, 色相を固定して明度を動かします. これを手作業でやるのは面倒なので, 色相環の上で直感的に選べるツールを使うのが近道です.

その前に, 色相環がどんな形をしているのかを見ておきましょう. ある明度をひとつ固定すると, その明るさで表示できる色だけが, 色相を角度, 彩度を中心からの距離としてリング状に並びます. 色域からはみ出す色はそもそも存在しないので, 縁はいびつな形になります.

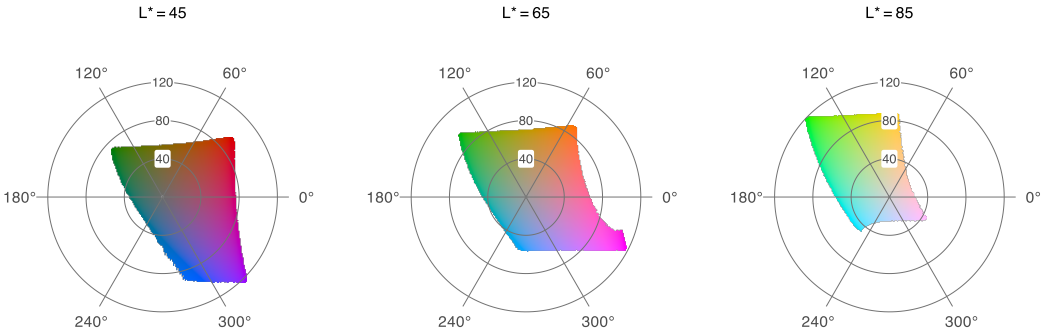


図 D.9:　明度ごとの色相環 (CIELAB, sRGB 色域内のみ)

図 D.9 は, 明度を 45, 65, 85 に固定したときの色相環です. 外周の角度ラベルが色相 (度), 同心円が等彩度 ($C^* = 40, 80, 120$) を表します. 中心からの角度が色相, 距離が彩度で, その明度で sRGB に収まる色だけを描いています. 縁がいびつで, しかも明度によって形が変わることに注目してください. 明度が低いと青や赤が, 高いと黄や緑が, より鮮やかなところまで届きます. これが「明るさと鮮やかさは同時に最大化できない」という色域 (gamut) の制約です.

D.3.1　hue360 の使い方

hue360 は, まさにこの色相環の上で色を選ぶためのオンラインツールです. 明度 (Munsell でいう Value) をひとつ決めてから, 色相環をクリックして色を拾っていくと, 明度と彩度のそろったパレットが手に入ります. ここまで見てきた「明度をそろえて色相を回す」という操作を, そのまま画面の上で行えます (図 D.10).

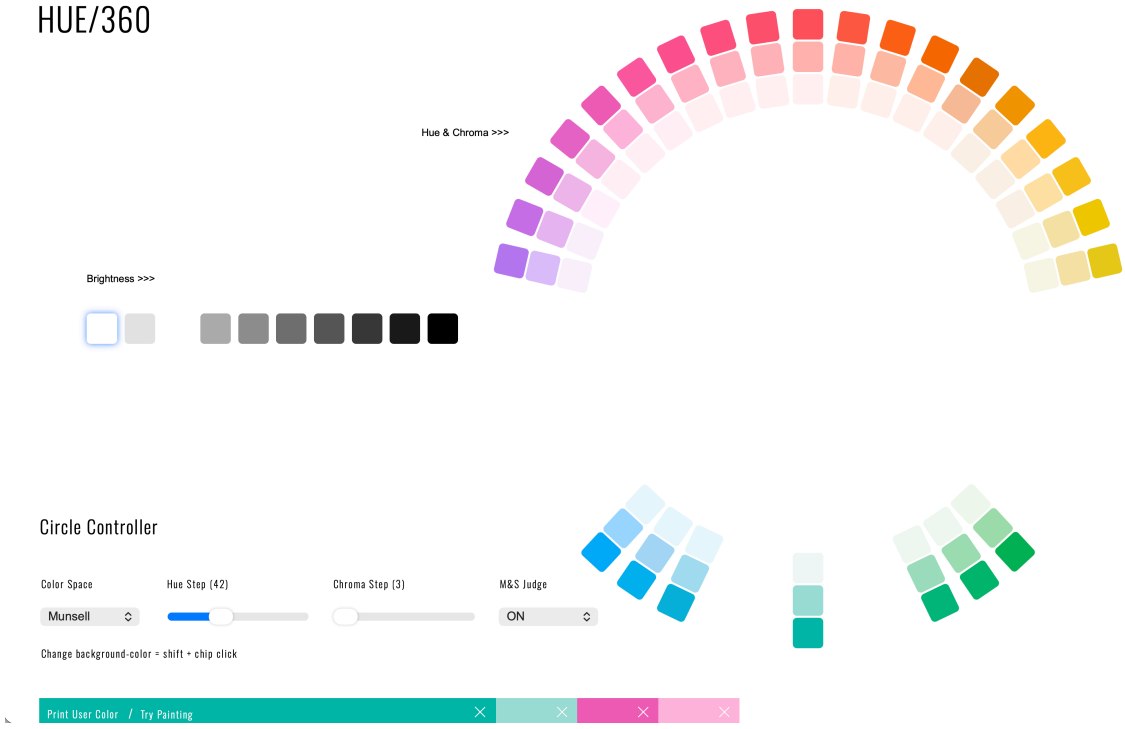


図 D.10:　Hue360

使い方の流れは次のとおりです.

1. まず明度 (Brightness) を決めます. 私は, デフォルトの白を使います.
2. ベースとなる色を選ぶましょう. 選択肢の中で彩度が最大のものが使いやすいと思います.
3. ベース色と合わない (Munsell の理論において) 色は自動的に選べなくなります.
4. 次に使いたい色を選びましょう. ここは自分の感覚で合いそうと思う色を選んでください.
5. 選んだ色の彩度が低い色も選んでおくと, 第 3, 第 4 の色として使えるので便利です.
6. “Print User Color” をクリックして, HEX コードを確認します.

D.4　補足: 色を感じる仕組み

最後に, なぜ黄がまぶしく見えるのか, という素朴な問いを通して, 色を感じる生理を覗いておきます. ここは実務に直結しませんが, これまでの話の裏側にある理屈です.

人間の網膜には, 色を感じる錐体細胞 (cone cell) が 3 種類あります. それぞれ短波長 (S, 青あたり), 中波長 (M, 緑あたり), 長波長 (L, 赤あたり) に感度のピークをもちます. 脳はこの 3 つの信号の比から色を復元します. つまり私たちは光の波長そのものではなく, 3 種類の錐体がどれだけ反応したか, という 3 つの数値だけを手がかりにしているのです.

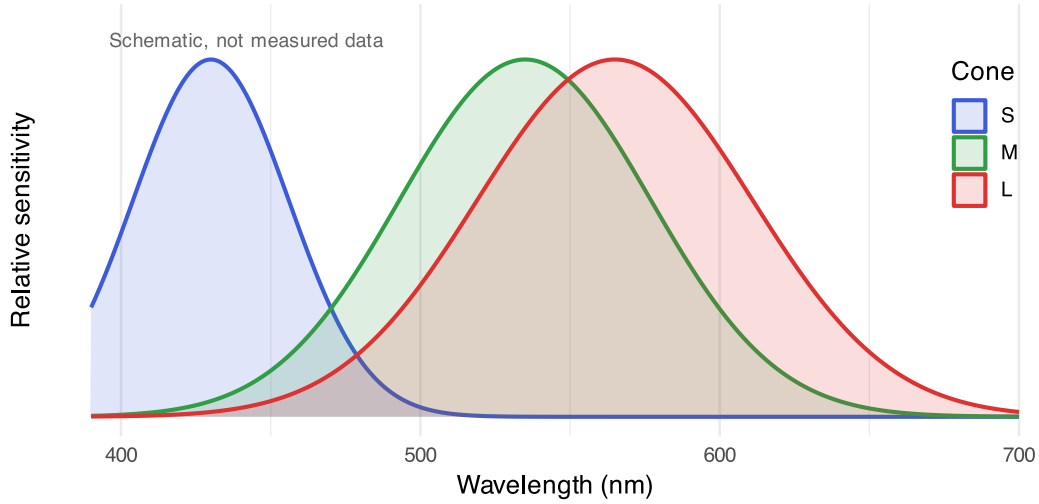


図 D.11: 錐体細胞の分光感度 (模式図)

図 D.11 のように, M (緑) と L (赤) の感度は大きく重なっていて, そのちょうど中間, 黄のあたりの波長で両方が同時に強く反応します. ここから 2 つの面白い帰結が出ます.

ひとつは条件等色 (metamerism) です. 波長 580 nm 前後の単色光としての黄と, 赤い光と緑の光を混ぜて作った黄は, 物理的にはまったく別物です. 前者はひとつの波長, 後者は 2 つの波長の重ね合わせです. それでも, どちらも L 錐体と M 錐体を同じ比率で刺激するなら, 私たちには区別できず, 同じ黄に見えます. 画面が 3 色の光だけであらゆる色を表現できるのは, この「錐体をだませば同じ色に見える」性質のおかげです.

もうひとつが, 黄がまぶしく見える理由です. 明るさの感覚は, おもに L 錐体と M 錐体の反応の和で決まり, その感度は黄緑 (およそ 555 nm) 付近で最大になります. 黄はこのピークのすぐそばにあるため, 同じエネルギーの光でも明るく感じられます. 画面ではさらに事情が重なります. 加法混色では

黄は赤と緑の両方を最大にした色 `#FFFF00` であり, ひとつの原色しか光らせない青 `#0000FF` より, 単純に多くの光を出しています.

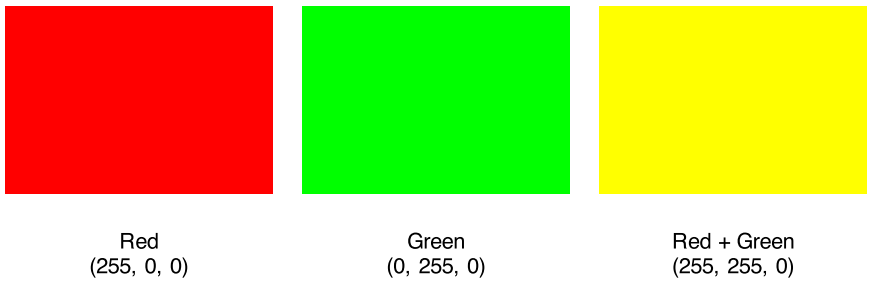


図 D.12: 加法混色: 赤 + 緑 = 黄

この 2 つの効果が重なって, 純色の黄は際立って明るく (図 D.1 の L^* で最大), 相対輝度でみると純色の青のおよそ 13 倍にもなります. だから白背景に黄の文字を置くと明るすぎて読めず, 逆に純色の青は暗すぎて重く見えるのです. スライドで原色を避けるべき理由は, 見た目の好みだけでなく, こうした知覚と生理の裏づけがあるのです.

実践のまとめ

- 原色 (`#FF0000` などの純色) と, 虹 (rainbow / jet) のカラーマップは避ける.
- 質的な色分けは, 明度と彩度をそろえて色相だけを変える (HCL / OKLCh で考える).
- 連続量には viridis のような明度が単調なカラーマップを使う.
- 色は色覚異常やグレースケールでも通じるよう, 明度差でも区別をつける (チャプター 7).

参考文献

Bergé, Laurent R., Kyle Butts, と Grant McDermott. 2026 年. *Fixest: A Fast and Feature-Rich Framework for Econometric Estimations in R*. arXiv:2601.21749. arXiv. https://doi.org/10.48550/arXiv.2601.21749.

Stevens, Marc, Elie Bursztein, Pierre Karpman, Ange Albertini, と Yarik Markov. 2017 年. 「The First Collision for Full SHA-1」. *Advances in Cryptology – CRYPTO 2017*, 編集者: Jonathan Katz と Hovav Shacham, vol. 10401. Springer International Publishing. https://doi.org/10.1007/978-3-319-63688-7_19.

Sun, Liyang, と Sarah Abraham. 2021 年. 「Estimating Dynamic Treatment Effects in Event Studies with Heterogeneous Treatment Effects」. *Journal of Econometrics* 225 (2): 175–99. https://doi.org/10.1016/j.jeconom.2020.09.006.

Tufte, Edward R. 2001 年. *The Visual Display of Quantitative Information*. 2nd ed. Graphics Press.

Wang, Xiaoyun, Dengguo Feng, Xuejia Lai, と Hongbo Yu. 2004 年. *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*. 2004/199.

Wang, Xiaoyun, Yiqun Lisa Yin, と Hongbo Yu. 2005 年. 「Finding Collisions in the Full SHA-1」. *Advances in Cryptology – CRYPTO 2005*, 編集者: David Hutchison, Takeo Kanade, Josef Kittler, ほか, vol. 3621. Springer Berlin Heidelberg. https://doi.org/10.1007/11535218_2.

図版クレジット

本書で使用している部品写真と一部の図版は, いずれも Wikimedia Commons で公開されているものです.

- 図 7.2, 図 7.3 (セリフ体・サンセリフ体): 作成 Stannered, Wikimedia Commons, CC BY-SA 3.0
- 図 1.2 (CPU): 撮影 Fritzchens Fritz, Wikimedia Commons, CC0 1.0
- 図 1.3 (メモリ): 撮影 PantheraLeo1359531, Wikimedia Commons, CC BY 4.0
- 図 1.4: 撮影 D-Kuru, Wikimedia Commons, CC BY-SA 4.0
- 図 1.5: 撮影 Mk2010, Wikimedia Commons, CC BY 4.0
- 図 1.6 (GPU): 撮影 FreeMediaKid!, Wikimedia Commons, CC BY-SA 4.0