

# 私たちの R: ベストプラクティスの探究

宋財泫 (Jaehyun Song) • 矢内勇生 (Yuki Yanai)

改訂: 2022-03-03



# 目次

<b>第1部 R の導入</b>	<b>11</b>
<b>第1章 R?</b>	<b>13</b>
1.1 R とは . . . . .	13
1.2 Why R? . . . . .	15
1.3 GUI と IDE . . . . .	19
<b>第2章 R のインストール</b>	<b>27</b>
<b>第3章 IDE の導入</b>	<b>29</b>
<b>第4章 分析環境のカスタマイズ</b>	<b>31</b>
4.1 .Rprofile の設定 . . . . .	31
4.2 RStudio のカスタマイズ . . . . .	31
<b>第5章 R パッケージ</b>	<b>33</b>
5.1 パッケージとは . . . . .	33
5.2 パッケージのインストール . . . . .	34
5.3 パッケージの読み込み . . . . .	35
5.4 パッケージのアップデート . . . . .	37
5.5 {pacman}によるパッケージ管理 . . . . .	38
5.6 必須パッケージのインストール . . . . .	40

<b>第 II 部 R の基礎</b>	<b>41</b>
<b>第 6 章 基本的な操作</b>	<b>43</b>
6.1 コンピュータの基礎知識 . . . . .	43
6.2 「プロジェクト」のすゝめ . . . . .	51
6.3 電卓としての R . . . . .	56
6.4 格納とオブジェクトの作成 . . . . .	60
6.5 要素の抽出 . . . . .	65
演習問題 . . . . .	67
<b>第 7 章 データの入出力</b>	<b>69</b>
7.1 データの読み込み . . . . .	69
7.2 データの書き出し . . . . .	84
<b>第 8 章 データの型</b>	<b>89</b>
8.1 データ型とは . . . . .	89
8.2 Logical . . . . .	90
8.3 Numeric . . . . .	92
8.4 Complex . . . . .	94
8.5 Character . . . . .	95
8.6 Factor . . . . .	96
8.7 Date . . . . .	101
8.8 NA . . . . .	112
8.9 NULL . . . . .	117
8.10 NaN . . . . .	119
8.11 Inf . . . . .	119
<b>第 9 章 データの構造</b>	<b>121</b>
9.1 データ構造とは . . . . .	121
9.2 ベクトル (vector) . . . . .	121
9.3 行列 (matrix) . . . . .	126
9.4 データフレーム (data.frame) . . . . .	139
9.5 リスト (list) . . . . .	153
9.6 配列 (array) . . . . .	160

---

練習問題 . . . . .	166
第 10 章 R プログラミングの基礎	171
10.1 R 言語の基礎概念 . . . . .	172
10.2 R のコーディングスタイル . . . . .	177
10.3 反復 . . . . .	185
10.4 条件分岐 . . . . .	211
練習問題 . . . . .	225
第 11 章 関数の自作	227
11.1 関数の作成 . . . . .	227
11.2 ちょっと複雑な関数 . . . . .	233
11.3 関数 in 関数 . . . . .	238
練習問題 . . . . .	242
第 III 部 データハンドリング	247
第 12 章 データハンドリング [基礎編: 抽出]	249
12.1 データハンドリングと tidyverse . . . . .	249
12.2 パイプ演算子 (%>%) . . . . .	250
12.3 列の抽出 . . . . .	254
12.4 行の抽出 . . . . .	279
12.5 行のソート . . . . .	292
練習問題 . . . . .	298
第 13 章 データハンドリング [基礎編: 拡張]	299
13.1 記述統計量の計算 . . . . .	300
13.2 グルーピング . . . . .	311
13.3 変数の計算 . . . . .	322
13.4 行単位の操作 . . . . .	336
13.5 データの結合 . . . . .	339
練習問題 . . . . .	355
第 14 章 データハンドリング [基礎編: factor 型]	357

14.1	名目変数を含むグラフを作成する際の注意点	357
14.2	{forcats}パッケージについて	366
	練習問題	395
	<b>第 15 章 整然データ構造</b>	397
15.1	整然データ (tidy data) とは	397
15.2	Wide 型から Long 型へ	405
15.3	Long 型から Wide 型へ	412
15.4	列の操作	418
	<b>第 16 章 文字列の処理</b>	427
	<b>第 IV 部 可視化</b>	429
	<b>第 17 章 可視化 [理論]</b>	431
17.1	可視化のためのパッケージ	431
17.2	グラフィックの文法	437
17.3	グラフィックの構成要素	443
17.4	良いグラフとは	459
	<b>第 18 章 可視化 [基礎]</b>	471
18.1	本章の内容	471
18.2	実習用データ	471
18.3	日本語が含まれた図について	472
18.4	棒グラフ	475
18.5	ヒストグラム	485
18.6	箱ひげ図	505
18.7	散布図	521
18.8	折れ線グラフ	530
18.9	図の保存	536
18.10	まとめ	539
	練習問題	541
	<b>第 19 章 可視化 [応用]</b>	543

---

19.1	labs(): ラベルの修正 . . . . .	544
19.2	coord_(): 座標系の調整 . . . . .	548
19.3	scale_(): スケールの調整 . . . . .	558
19.4	theme_() と theme(): テーマの指定 . . . . .	598
19.5	図の結合 . . . . .	617
<b>第 20 章 可視化 [発展]</b>		627
20.1	概要 . . . . .	627
20.2	バイオリンプロット . . . . .	628
20.3	ラグプロット . . . . .	634
20.4	リッジプロット . . . . .	639
20.5	エラーバー付き散布図 . . . . .	648
20.6	ロリーポップチャート . . . . .	655
20.7	平滑化ライン . . . . .	664
20.8	ヒートマップ . . . . .	673
20.9	等高線図 . . . . .	680
20.10	地図 . . . . .	687
20.11	非巡回有向グラフ . . . . .	708
20.12	バンプチャート . . . . .	716
20.13	沖積図 . . . . .	723
20.14	ツリーマップ . . . . .	735
20.15	モザイクプロット . . . . .	742
20.16	その他のグラフ . . . . .	750
<b>第 V 部 再現可能な研究</b>		751
<b>第 21 章 R Markdown [基礎]</b>		753
21.1	R Markdown とは . . . . .	753
21.2	とりあえず Knit . . . . .	755
21.3	Markdown 文法の基本 . . . . .	756
21.4	チャンクのオプション . . . . .	770
21.5	ヘッダーのオプション . . . . .	777
21.6	日本語が含まれている PDF の出力 . . . . .	781

第 22 章 R Markdown [応用]	783
22.1 スライド作成 . . . . .	783
22.2 書籍 . . . . .	783
22.3 ホームページ . . . . .	783
22.4 履歴書 . . . . .	783
22.5 パッケージ開発 . . . . .	784
22.6 Web アプリケーション . . . . .	784
22.7 チュートリアル . . . . .	784
第 23 章 表の作成	785
第 24 章 モデルの可視化	787
第 25 章 分析環境の管理	789
第 VI 部 中級者向け	791
第 26 章 データハンドリング [応用編]	793
第 27 章 反復処理	795
27.1 概要 . . . . .	795
27.2 *apply() 関数群と map_*( ) 関数群 . . . . .	795
27.3 引数が 2 つ以上の場合 . . . . .	803
27.4 データフレームと{purrr} . . . . .	806
27.5 モデルの反復推定 . . . . .	813
第 28 章 オブジェクト指向型プログラミング	859
28.1 まずは例から . . . . .	859
28.2 OOP とは . . . . .	862
28.3 R における OOP . . . . .	869
28.4 例題 . . . . .	877
第 29 章 モンテカルロシミュレーション	885
29.1 モンテカルロシミュレーションとは . . . . .	885

---

29.2	乱数生成	886
29.3	例 1: 誕生日問題	890
29.4	例 2: モンティ・ホール問題	901
29.5	応用 1: 円周率の計算	904
29.6	応用 2: ブートストラップ法	912
第 30 章 スクレイピング		917
第 31 章 API		919
演習問題の回答		921
31.1	基本的な操作	921
31.2	データの構造	921
31.3	R プログラミングの基礎	924
31.4	関数の自作	926



## 第Ⅰ部

### R の導入



# 第1章

R?

## 1.1 R とは



図 1.1: R Logo

R は統計、データ分析、作図のためのインタープリタープログラミング言語である。R という名前は二人の開発者 Ross Ihaka と Robert Clifford Gentleman のイニシャルに由来する。R 言語は完全にゼロベースから開発されたものではなく、1976 年に開発された S 言語に起源をもつ。S 言語も R 言語同様、統計やデータ分析に特化した言語であり、S 言語の開発が中止された現在、R は S の正当な後継者であると言ってよいだろう。

R 以外にも、統計・データ分析のために利用可能なソフトウェアはたくさんある。社会科学におけるデータ分析の授業を履修したことがあるなら、SPSS や Stata という名前をきいたことがあるかもしれない。工学系なら MATLAB が有名かもしれない。企業

では SAS という高価なソフトもよく使われる<sup>1)</sup>。

これらのソフト（アプリ）は基本的に有料だが、お金がないとデータ分析のソフトウェアが使えないわけではない。無料でありながら優れたソフトウェアも多く公開されている。以下にその一部を示す。

ソフト・言語名	備考
PSPP	SPSS にとてもよく似た無料ソフトウェア。
JASP/jamovi	裏で動いているのは R。詳細は本章の後半で
gretl	時系列分析など、計量経済学で利用される手法に特化したソフト。
GNU Octave	MATLAB とほぼ同じ文法をもつ無料言語
HAD	清水裕士 先生が開発している Excel ベースのデータ分析マクロ

統計分析に特化したプログラミング言語として R の競合相手になるのは Julia と Python だろう。Julia 言語は一見 R によく似ているが、計算速度が R より速いと言われている<sup>2)</sup>。ただし、比較的新しい言語であるため、パッケージと解説書が R よりも少ないという難点がある。Python は現在のデータサイエンス界隈において、R とともに最も広く使われている言語である<sup>3)</sup>。Python は統計・データ分析に特化した言語ではないが、統計・データ分析のライブラリが非常に充実しており、機械学習（とくに、コンピュータービジョン）では R よりも広く使われている。Python 以外の言語、たとえば C や Java、Ruby、Fortran でも統計・データ分析は可能ある。実際、R や Python のパッケージの一部は C や Java で作成されている。ただし、R や Python に比べると、データ分析の方法を解説したマニュアル・参考書があまりないのがつらいところだ。ひとつの言語だけでなく、複数の言語を使えるようになったほうが良いことは言うまでもない。本書は R について解説するが、他の言語にもぜひ挑戦してほしい。

<sup>1)</sup> 最近、アカデミック版が無料で使えるようになった。

<sup>2)</sup> R もコードの書き方によってパフォーマンス向上ができる。

<sup>3)</sup> Python は統計分析以外でも利用されるので、Python 自体のユーザは R のユーザより多いだろう。

## 1.2 Why R?

私たち R ユーザにとっての神のような存在である Hadley Wickham (通称: 羽鳥先生) は Advanced R (2nd Ed.) で次のように仰せられた。

- It's free, open source, and available on every major platform. As a result, if you do your analysis in R, anyone can easily replicate it, regardless of where they live or how much money they earn.
1. R は無料で、オープンソースで、多くのプラットフォーム (訳注: macOS, Linux, Windows など) で利用できます。よって、R を使って分析すれば、どこに住んでいるか、いくらお金を稼いでいるかに関係なく、誰でも簡単にその分析を再現することができます。
    - R has a diverse and welcoming community, both online (e.g. the #rstats twitter community) and in person (like the many R meetups). Two particularly inspiring community groups are rweekly newsletter which makes it easy to keep up to date with R, and R-Ladies which has made a wonderfully welcoming community for women and other minority genders.
  2. オンライン (twitter の#rstat など)、オフライン (訳注: 日本では Tokyo.R が有名。筆者たちが開催している KUT.R もある) の両方で、多様な R コミュニティがあります。他にも最新の R をキャッチアップするためのニュースレターである rweekly や、女性や性的マイノリティにやさしい R-Ladies も活発に活動しています。
    - A massive set of packages for statistical modelling, machine learning, visualisation, and importing and manipulating data. Whatever model or graphic you're trying to do, chances are that someone has already tried to do it and you can learn from their efforts.
  3. 統計モデリング、機械学習、可視化、データ読み込みおよびハンドリングのための膨大なパッケージが用意されています。どのようなモデルやグラフでも、既に誰かがその実装を試みた可能性が高く、先人らの努力に学ることができます。

- Powerful tools for communicating your results. R Markdown makes it easy to turn your results into HTML files, PDFs, Word documents, PowerPoint presentations, dashboards and more. Shiny allows you to make beautiful interactive apps without any knowledge of HTML or javascript.
4. 分析結果を伝達する強力なツールを提供しています。R Makrdown は分析結果を HTML、PDF、Word、PowerPoint、Dashboard 形式に変換してくれます。HTML や JavaScript の知識がなくても、Shiny を使って美しい対話型のアプリケーションを開発することができます。
- RStudio, the IDE, provides an integrated development environment, tailored to the needs of data science, interactive data analysis, and statistical programming.
5. 代表的な統合開発環境である RStudio はデータサイエンス、対話型のデータ分析、そして統計的プログラミングが必要とするものに最適化されています。
- Cutting edge tools. Researchers in statistics and machine learning will often publish an R package to accompany their articles. This means immediate access to the very latest statistical techniques and implementations.
6. R は最先端のツールです。多くの統計学や機械学習の研究者は自分の研究成果と R パッケージを同時に公開しています。これは最先端の方法を誰よりも早く実施可能にします。
- Deep-seated language support for data analysis. This includes features like missing values, data frames, and vectorisation.
7. データ分析を根強くサポートする言語です。欠損値、データフレーム、ベクトル化などがその例です。
- A strong foundation of functional programming. The ideas of functional programming are well suited to the challenges of data science, and the R language is functional at heart, and provides many primitives needed for effective functional programming.
8. R はデータサイエンスに非常に有効である関数型プログラミングのための最適な

環境を提供しています。

- RStudio, the company, which makes money by selling professional products to teams of R users, and turns around and invests much of that money back into the open source community (over 50% of software engineers at RStudio work on open source projects). I work for RStudio because I fundamentally believe in its mission.

9. RStudio 社は営利企業ですが、その収益の多くをオープンソースコミュニティーに投資しています。

- Powerful metaprogramming facilities. R's metaprogramming capabilities allow you to write magically succinct and concise functions and provide an excellent environment for designing domain-specific languages like `ggplot2`, `dplyr`, `data.table`, and more.

10. メタプログラミングが強力です。R が持つメタプログラミング能力はあなたのコードを劇的に簡潔にするだけでなく、統計/データ分析に特化した `ggplot2`、`dplyr`、`data.table` などの開発も可能にしました。

- The ease with which R can connect to high-performance programming languages like C, Fortran, and C++.

11. R は C、C++、Fortran のようなハイパフォーマンス言語と容易に結合できるように設計されています。

しかし、他のプログラミング言語と同様、R は完璧な言語ではありません。以下は R 言語の短所の一部です。その多くは R そのものの問題というよりも、(プログラマーではなく) データ分析の研究者が中心となっている R ユーザーから起因する問題です。

- Much of the R code you'll see in the wild is written in haste to solve a pressing problem. As a result, code is not very elegant, fast, or easy to understand. Most users do not revise their code to address these shortcomings.

1. あなたが普段見る多くの R コードは「今の」問題を解決するために迅速に書かれたものです。この場合、コードはあまりエレガントでも、速くも、読みやすくあり

ません。ほとんどのユーザーはこの短所を克服するためのコード修正を行っておりません。

- Compared to other programming languages, the R community is more focussed on results than processes. Knowledge of software engineering best practices is patchy. For example, not enough R programmers use source code control or automated testing.
2. 他のプログラミング言語に比べ、R コミュニティは過程よりも結果に注目する傾向があります。多くのユーザーにおいて、ソフトウェアエンジニアリングの知識を蓄えるための方法が不完全です。たとえば、(GitHub などの) コード管理システムや自動化された検証を使用する R プログラマーは多くありません。
- Metaprogramming is a double-edged sword. Too many R functions use tricks to reduce the amount of typing at the cost of making code that is hard to understand and that can fail in unexpected ways.
3. R の長所でもあるメタプログラミングは諸刃の剣です。あまりにも多くの R 関数はコーディングのコストを減らすようなトリックを使用しており、その代償としてコードの理解が難しく、予期せぬ失敗の可能性があります。
- Inconsistency is rife across contributed packages, and even within base R. You are confronted with over 25 years of evolution every time you use R, and this can make learning R tough because there are so many special cases to remember.
4. 開発されたパッケージは、R 内蔵のパッケージさえも一貫性が乏しいです。あなたは R を使う度にこの 25 年を超える R の進化に直面することになります。また、R には覚えておくべきの特殊なケースが多く、これは R 学習の妨げとなっています。
- R is not a particularly fast programming language, and poorly written R code can be terribly slow. R is also a profligate user of memory.
5. R は格別に速い言語ではありません。下手に書かれたコードは驚くほど遅いです。また、R はメモリの浪費が激しい言語と知られています。
-

## 1.3 GUI と IDE

### 1.3.1 GUI

#### CUI と GUI

現在、多くのソフトウェアは GUI を採用している。GUI とは Graphical User Interface の略で、ソフトウェア上の入力および出力にグラフィックを利用する。単純にいうと「マウスでポチポチするだけで操作できる環境」のことだ。一方、R では基本的に CUI (Character User Interface) を利用する<sup>4)</sup>。これは全ての操作を文字列ベースで行う、つまりキーボードで行うことを意味する<sup>5)</sup>。

身近な例として、あるラーメン屋での注文（呪文）システムを考えてみよう。無料トッピングを指定するときに「決まった言い方」で指定するのが CUI 方式である。一文字でも間違うとオーダーは通りらない。たとえば、「野菜マッシマッシ!」とか「野菜 MashMash!」と言ってしまうと、店長さんに「は？」と言われ、周りの客から白い目で見られる<sup>6)</sup>。他方、食券の自動発売機でトッピングを指定できるのが GUI 方式だ<sup>7)</sup>。この場合、そもそも間違いは起きない。誰でも簡単に注文できるのが GUI 式のメリットだが、自分で注文するものが決まっていて、注文の仕方を知っているなら CUI 式のほうが早い。毎回同じ注文をするなら、注文内容を録音しておいて、次回はそれを再生することも可能である。GUI 方式では、注文方法を再現するのは難しい。

---

<sup>4)</sup> CLI (Command Line Interface) とも呼ばれる。

<sup>5)</sup> 多くの人が片手に 5 本も 5 指を有効に利用できないマウスという不便な機器から解放される。

<sup>6)</sup> すべての「ラーメン〇郎」がそうだというわけではない。

<sup>7)</sup> たとえば、「〇蘭」の注文システムのように。

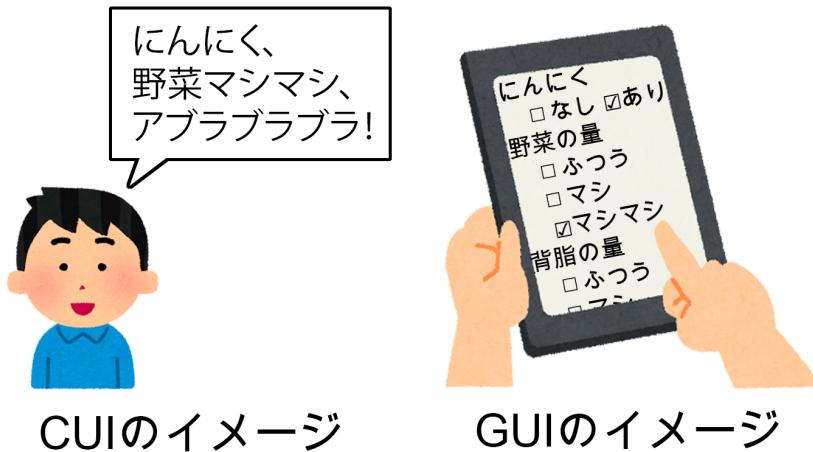


図 1.2: CUI と GUI の比較

CUI と GUI を比べたとき、一見すると GUI のほうが優れているように見える。マウスでポチポチするだけで操作できるほうが楽に見えるし、間違いの心配もなさそうな気がする。キーボードで長いコマンドを打つ CUI よりも、ボタンをクリックしたほうが手早く済みそうにも思える<sup>8)</sup>。そして、CUI のコマンドを覚えるのはしんどい<sup>9)</sup>。

しかし、CUI には CUI なりの長所がある。まず、コードが記録できる。マウスでポチポチした操作は、パソコンの画面全体を録画しない限り記録できない。よって、GUI では自分の分析プロセスを記録することが難しい。他方、すべてコマンドで操作した場合、入力したコマンドを文字列として記録することは容易である。また、CUI は GUI よりも柔軟である。先ほどのラーメン屋の例で考えると、CUI では「一応」野菜ちょいマシのような注文（呪文）のカスタマイズもできる<sup>10)</sup>。しかし、GUI（券売機）だと注文の自由なカスタマイズはできない。店が用意したボタンしか押せない（ソフトがメニューに表示しているコマンドしか実行できない）。また、コマンドの入力の時間や暗記も、それほど難しくはない。後で説明するように、R ユーザにとっての超有能な秘書である RStudio (IDE の 1 種) がコマンド入力を助けてくれる。スペルが間違っていたり、うろ覚えのコマンドなどがあっても、RStudio (あるいはその他の IDE) が瞬時に正解に導いてくれる。

<sup>8)</sup> これらはすべて錯覚である。

<sup>9)</sup> これは錯覚ではない。

<sup>10)</sup> 通るかどうかは別だが…

本書は CUI としての R について解説する。R は統計ソフトでありながら、言語でもある。外国語の勉強に喩えるなら、CUI を使うのは単語や熟語を覚え、文法やよく使う表現を学習して自分で考えて話すことであるのに対し、GUI を使うのは AI に翻訳を頼み、外国語については自分で一切考えないことに似ている。たいして興味のない国になぜか旅行で訪れることになった場合には GUI 方式で済ますのも良いだろう。しかし、それでその言語を「勉強した」とか「理解した」などという人はいないはずだ。R を「理解」したいなら、CUI 以外の選択肢はない<sup>11)</sup>。

それでもやはり GUI のほうがとっつきやすいという頑固なあなたのために代表的な R の GUI を紹介するので、以下のリストを確認したらこの本は閉じていただきたい。またいつかどこかでお会いしましょう。CUI を使う覚悟ができたあなたは、以下のリストを読みとばし、引き続き一緒に R の世界を楽しみましょう！

- R Commander

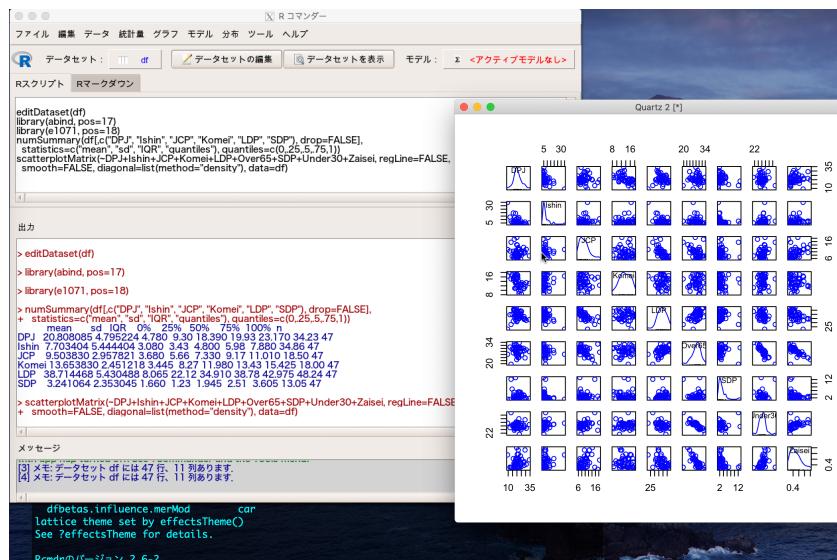


図 1.3: R Commander

- RKWard

<sup>11)</sup> macOS に R をインストールする場合は、途中で「GUI をインストールする」というチェックを外すことができるの、外しておこう。R の GUI は不要である。あっても困りはしないが。

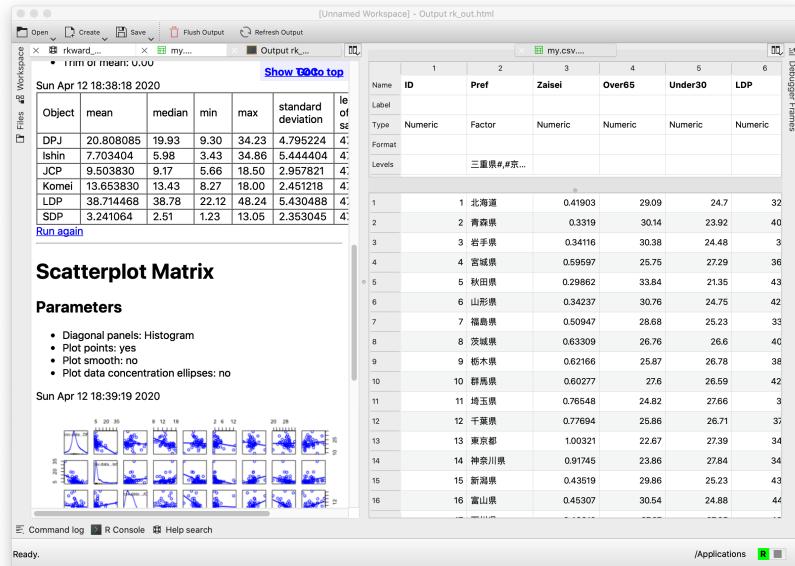


図 1.4: RKWard

- JASP

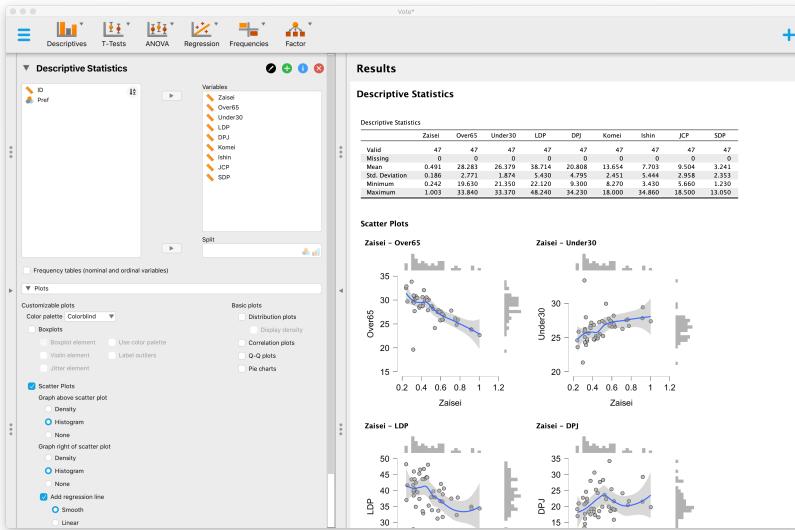


図 1.5: JASP

- jamovi

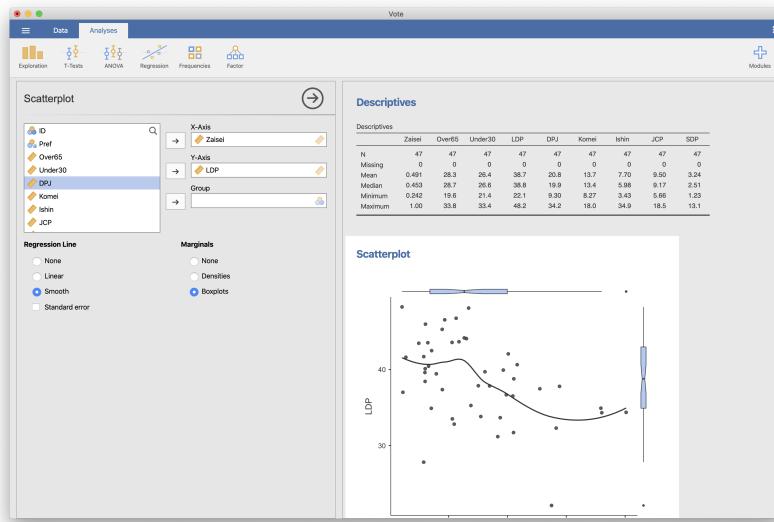


図 1.6: jamovi

- R AnalyticFlow

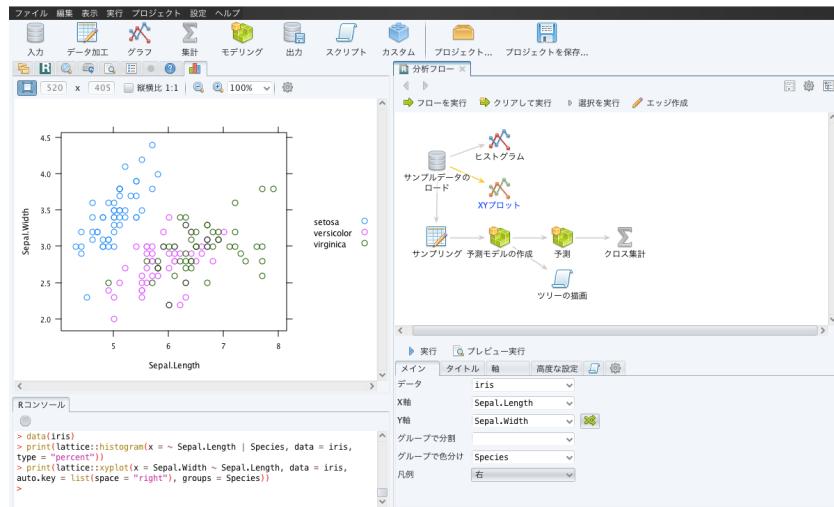


図 1.7: RAnalyticFlow (画像は公式ホームページから)

### 1.3.2 IDE

プログラミングは基本的にコードを書く作業の連続だが、コードを書く他にも様々な作業を行うことになる。たとえば、自分が書いたコードの結果が正しく動作するかの確認作業や、なにか問題がある場合の対処（デバッグ）などがある。また、コードを書く際、誤字やミスなどがないかも確認する必要がある。他にもプログラムで使用されるファイルを管理しなければならない。これらの仕事を手助けしてくれるのが統合開発環境（integrated development environment; IDE）と呼ばれるものである。

プログラマにとって優れた IDE を使うということは、優れた秘書を雇用するようなものだ。ファイルの管理、うろ覚えのコマンドの補完入力、コードの色分けなどを自動的に行ってくれる。さらに、コードの実行結果の画面をコードと同時に表示してくれたり、これまでの作業を記録してくれるなど、多くの作業を手助けしてくれる。R にはいくつかの優れた IDE が用意されている。本書では代表的な IDE である RStudio を使うことにする。ただし、プログラミングに IDE は必須ではない。IDE をインストールしなくとも、本書を読む上で特に問題はない（RStudio に関する説明の部分を除く）が、R の実行環境に特にこだわりがないなら RStudio の導入を強く推奨する。

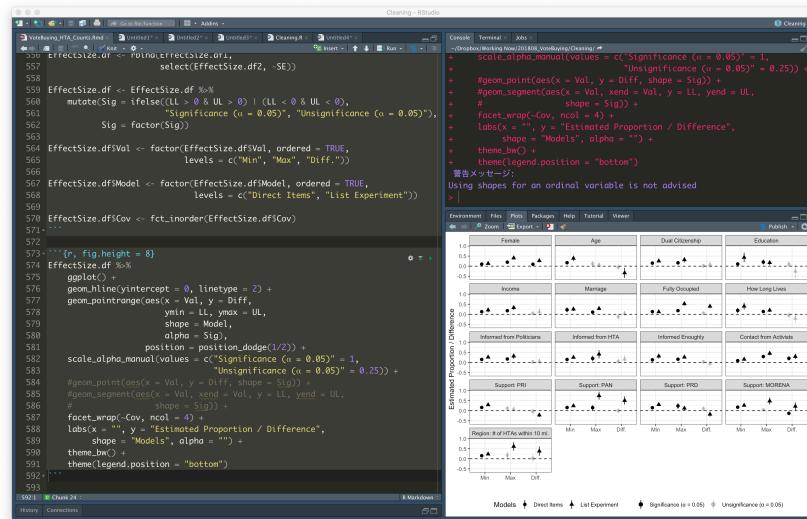


図 1.8: RStudio

RStudio 以外にも R の IDE はある。魔界において圧倒的なシェアを誇ると噂される Windows という名の OS を使用しているなら、R Tools for Visual Studio が RStudio の代替候補として有力だ。

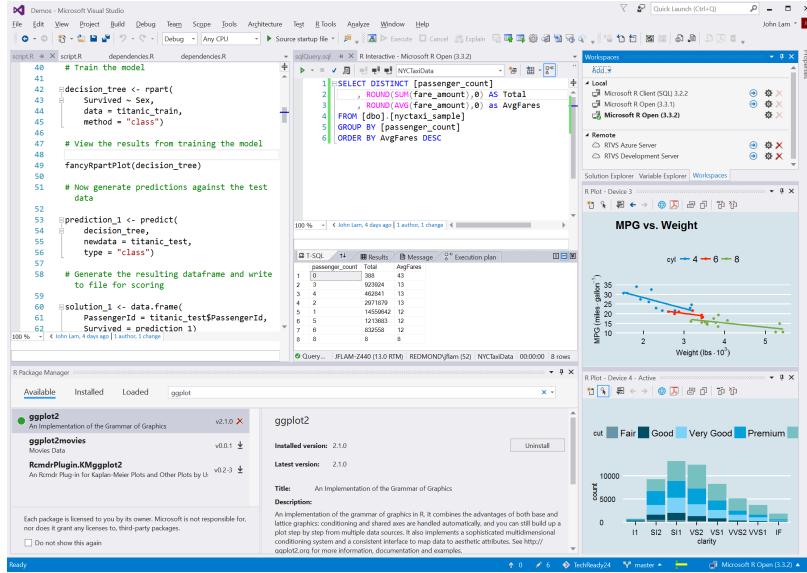


図 1.9: R Tools for Visual Studio

自分が使い慣れたテキストエディタを IDE として使うことも可能である。Sublime Text や Atom はもちろん、伝統のある Emacs や Vim を使うこともできる。



## 第 2 章

# R のインストール

本章の内容は今後、以下の資料に基づき、再作成する予定である。

- 矢内による資料 (macOS 編、Linux (Ubuntu) 編、Windows 編)



## 第3章

# IDE の導入

本章の内容は今後、以下の資料に基づき、再作成する予定である。

- 矢内による資料 (macOS 編、Linux (Ubuntu) 編、Windows 編)



## 第4章

# 分析環境のカスタマイズ

### 4.1 .Rprofile の設定

### 4.2 RStudio のカスタマイズ

1. Tools > Global Options

2. General

1. Basic タブ

- Restore .RData into workspace at startup のチェックを消す。
- Save workspace to .RData on exit: を Never に変更する。
- Always save history (even when not saving .RData) のチェックを消す。

3. Code

1. Editing タブ

- Insert spaces for tab のチェックを付ける。
  - Tab width は 2 か 4
- Auto-detect code indentation のチェックを付ける。
- Insert matching parens/quotes のチェックを付ける。
- Auto-indent code after paste のチェックを付ける。
- Vertically align arguments in auto-indent のチェックを付ける。
- Always save R scripts before sourcing のチェックを付ける。
- Ctrl + Return executes: を Multi-line R statement に変更する。

2. Display タブ

- Highlight selected word のチェックを付ける。
- Highlight selected line のチェックを付ける。
- Show line numbers のチェックを付ける。
- Show syntax highlighting in console input のチェックを付ける。
- Highlight R function calls のチェックを付ける。
- Rainbow parentheses のチェックを付ける。

3. Saving タブ

- Default text encoding: の Change をクリックし、UTF-8 を選択する。

4. Completion タブ

- Show code completion: を Automatically に変更する。
- Allow automatic completions in console のチェックを付ける。
- Insert parentheses after function completions のチェックを付ける。
- Show help tooltip after function completions のチェックを付ける。
- Insert spaces around equals for argument completions のチェックを付ける。
- Use tab for autocompletion のチェックを付ける。

4. Appearance

- 自分の好みのものを選択する。

5. Pane Layout

- 左上: Source
- 右上: Console
- 左下: 全てチェックを消す。
- 左下: 全てチェックを付ける。

6. 左下の Apply をクリック

## 第 5 章

# R パッケージ

### 5.1 パッケージとは

R には様々な関数 (functions) が提供されている。平均値を求める `mean()`、合計を求める `sum()`、線形回帰分析を行う `lm()`、平均値の検定を行う `t.test()` などがあり、全てを列挙することはできない。しかし、データ分析の技術は日々発展し、R がデフォルトで提供する関数では不可能ではないが、かなり長いコードが必要な分析を使わざる得ないケースもある。R は開発元だけでなく、誰でも関数を作ることができる。通常なら数百行のコードが必要な分析を一行のコードで実行可能とする関数を多くの R ユーザーが作ってきた。これらの関数を集めたのがパッケージである。R にはグラフ作成に特化したパッケージ、機械学習に特化したパッケージ、テキスト分析に特化したパッケージなど、数千のパッケージが開発されている。このパッケージの豊富さが R の最大のメリットでもある。誰かが新しい分析手法を提案したら、数日内、あるいはその手法が論文として出版される前から R パッケージとして公開されるケースが多い。

本章ではパッケージをインストールし、読み込む方法について説明する。また、パッケージの管理をアシストするパッケージ、`{pacman}` の使い方についても紹介する。

---

## 5.2 パッケージのインストール

R の環境は何かを作るための作業台に似ている。作業台にはモノを作るために材料だけでなく、工具・道具セットなども置いたりもする。この作業台が R における「環境 (environment)」であり、材料がベクトルや行列、データフレームなどのデータ、工具セットがパッケージになる。データについては後で説明するとし、ここではパッケージについて考えたい。

モノを作るためには素材・材料だけでは不十分だろう。多くの場合、なんらかの道具セットが必要となる。R には既にいくつかの必須道具セットを用意されているが、他にも様々な道具セットがある。そして、これら道具セットには、一般的に複数の道具が含まれている。一つ一つの道具のことを、ここでは「関数 (function)」と呼ぶ。これらの道具セットを購入し、作業台の収納に入れておくことがパッケージをインストールすることである。

```
1 install.packages("パッケージ名")
```

これらのパッケージは基本的に CRAN という R の公式道具屋からダウンロード・インストールされる。もう一つの大きな道具屋としては GitHub がある<sup>1)</sup>。GitHub は個人経営の道具屋が集まっているモールのようなものである。GitHub 道具屋を使用するためには、予め CRAN から`{devtools}`、または`{remotes}`というパッケージをインストールしておく必要がある。

ここでは`{devtools}`というパッケージをインストールしてみよう。`{devtools}`は CRAN に登録されているため、`install.packages()` 関数でインストールできる。パッケージ名を"で囲むことを忘れないこと。

```
1 install.packages("devtools")
```

もし、CRAN に登録されていないパッケージを GitHub からインストールするなら、`{devtools}`パッケージ、または`{remotes}`の`install_github()`関数を使う。

---

<sup>1)</sup> 他にも GitLab、Bitbucket などがある。

```
1 # あらかじめ{devtools}、または{remotes}をインストールしておく
2 # {remotes}をインストールした場合は、remotes::install_github() を使用する
3 devtools::install_github("作成者の GitHub の ID/パッケージ名")
```

たとえば、筆者 (Song) が作成しました{BalanceR}パッケージがインストールしたいなら、以下のように打つ。

```
1 # {remotes}をインストールした場合は、
2 # remotes::install_github("JaehyunSong/BalanceR")
3 devtools::install_github("JaehyunSong/BalanceR")
```

ここで `JaehyunSong` は `Song` の GitHub ID であり、`BalanceR` はパッケージ名である。

## 5.3 パッケージの読み込み

先ほど述べたように、パッケージのインストールは道具セットの購入と収納に似ている。ただし、実際に道具セットを使うためには、それを自分の作業台上に載せた方が効率がいいだろう<sup>2)</sup> <sup>3)</sup>。この作業がパッケージの読み込み (load) である。インストールしたパッケージを読み込むには `library()` または `require()` 関数を使う。`require()` は関数内に使う目的で設計された関数だが、パッケージを読み込むという点では全く同じである。

---

<sup>2)</sup> `mean()` や `sum()`、`lm()` のように、よく使われる関数 (=工具) は R 起動と同時に作業台上に載せられる。

<sup>3)</sup> 作業台上に載せずに、収納から必要な時だけ道具を取り出して使うことも可能である。この場合、パッケージ名:: 関数名() のように関数を使う。よく使うパッケージなら読み込んだ方が効率的だが、1、2回くらいしか使わないパッケージなら、このような使い方も良いだろう。



図 5.1: R パッケージと作業環境

```
1 library("/パッケージ名")
2 #または
3 require("/パッケージ名")
```

読み込まれたパッケージはセッションが開かれている時のみに有効である。一通りの作業が終わり、作業部屋から退出すると、作業台上の工具セットは収納に自動的に戻される。つまり、R または RStudio を閉じると読み込まれたパッケージは自動的に取り外されるということである。しかし、作業の途中に読み込んだパッケージをセッションから取り外したい時があるかも知れない。この場合、`detach()` 関数を使う。

```
1 detach("/パッケージ名")
```

## 5.4 パッケージのアップデート

R パッケージはバグが修正されたり、新しい機能 (=関数) が追加されるなど、日々更新される。できる限りパッケージは最新版に維持した方が良いだろう。パッケージのアップデートはパッケージのインストールと同じである。{dplyr}というパッケージを最新版にアップデートしたい場合、`install.packages("dplyr")` で十分である。

しかし、R を使っていくうちに数十個のパッケージをインストールしていくこととなり、一つ一つアップデートするのは面倒だろう。そもそも既に最新版が入っていて（または開発休止/中止）、アップデートが不要なパッケージがあるかも知れない。実は RStudio を使えば、アップデートが必要なパッケージのリストが表示され、一気にアップデートすることができる。RStudio の Packages ペインにある「Update」をクリックしてみれば、アップデート可能なパッケージの一覧が表示される。ここでアップデートしたいパッケージの左にチェックをするか、下段の「Select All」を選択して「Install Updates」をクリックすれば、チェックされているパッケージがアップデートされる。

ただし、場合によってはアップデート時、以下のようなメッセージがコンソールに表示されるかも知れない。

```
There are binary versions available but the source versions
are later:
  binary  source  needs_compilation
terra  1.5-17  1.5-21          TRUE
yaml    2.2.2   2.3.4          TRUE

Do you want to install from sources the packages which need compilation? (Yes/no/cancel)
```

コンソールに Yes、no、cancel のいずれかを入力して Return キー (Enter キー) を押す必要がある。どうしても最新のパッケージが欲しい場合は Yes を入力すれば良いが、インストールに時間がかかる場合がある。一方、no を入力した場合は、若干古いバージョンがインストールされるが、インストールに必要な時間が短いため、基本的には no でも問題ないだろう。cancel を入力した場合はアップデートが全てキャンセルされる。

## 5.5 {pacman}によるパッケージ管理

CRAN と GitHub などには数千の R パッケージが公開されており、R の使用歴が長くなればインストールされているパッケージが増えたり、一つのスクリプト内で使用するパッケージも増えていくだろう。また、パッケージは他のパッケージの機能に依存することがほとんどなので、自分の想像以上の数のパッケージがインストールされているかも知れない。このように膨大な数のパッケージを管理するためのパッケージが{pacman}である。{pacman}は CRAN から入手可能である。

```
1 install.packages("pacman")
```

### 5.5.1 インストール

パッケージを CRAN からインストールには `p_install()` 関数を使用する。使い方は `install.packages()` と同じであり、複数のパッケージをインストールしたい場合はパッケージ名の箇所に `c(パッケージ名1, パッケージ名2, ...)` を入れる。パッケージ名は"で囲んでも、囲まなくても良い。GitHub に公開されているパッケージは `p_install_gh()` 関数を使用する。これは{devtools}、または{remotes}の `install_github()` と同じ使い方となり、必ず"で囲む必要がある。

これらの関数を使う際、わざわざ `library(pacman)` を使う必要はない。パッケージのインストールや、読み込みなどはコード内に何回も使われることがほとんどないため、{pacman}を読み込みます `pacman:: 関数名()` で当該関数を使うことができる。

```
1 # CRAN からインストール
2 pacman::p_install(パッケージ名)
3 # github からインストール
4 pacman::p_install_gh("作成者の GitHub の ID/パッケージ名")
```

### 5.5.2 読み込み

パッケージの読み込みには `p_load()` 関数を使い、実はこの関数は {pacman} を使う最も大きな要素である。`p_load()` 関数の使い方は以下の通りである。

```
1 pacman::p_load(パッケージ名)
```

`p_load()` の便利なところは (1) 複数のパッケージが指定可能であることと、(2) インストールされていないパッケージは CRAN から自動的にインストールして読み込んでくれる点だ。たとえば、{tidyverse} と {broom}、{estimatr} という 3 つのパッケージを読み込む場合、`library()` 関数を使うと以下のようになる。

```
1 # library() を使う場合
2 library(tidyverse)
3 library(broom)
4 library(estimatr)
```

一方、{pacman} の `p_load()` を使えば、以下のように 3 つのパッケージを読み込むことができる。

```
1 # {pacman} の p_load() を使う場合
2 pacman::p_load(tidyverse, broom, estimatr)
```

また、`p_load()` 内のパッケージがインストールされていない場合、CRAN のパッケージリストから検索し、そのパッケージをインストールしてくれる。したがって、上で紹介した `p_install()` は実質的に使うケースはほぼない。ただし、GitHub 上のパッケージは自動的にインストールしてくれない。たとえば、GitHub 上のみにて公開されている {BalanceR} パッケージがインストールされていない場合、`p_load(BalanceR)` を実行しても {BalanceR} はインストールされない<sup>4)</sup>。あらかじめ `p_install_gh()` でインストー

---

<sup>4)</sup> GitHub からパッケージを検索し、インストール&読み込みをする `p_load_gh()` という関数もある。たとえば、`pacman::p_load_gh("JaehyunSong/BalanceR")` を実行した場合、{BalanceR} パッケージがインストールされていると読み込みのみ行い、インストールされていない場合は JaehyunSong レポジトリから {BalanceR} をインストールし、読み込む。コードの最上段に `p_load()` を使うなら、`p_load()` と `p_load_gh()` を分けて記述するのも良いだろう。

ルしておく必要がある。

### 5.5.3 アップデート

{pacman}にはアップデートが可能なパッケージを全てアップデートしてくれる `p_update()` という関数も用意されている。使い方は簡単で、コンソール上に `p_update()` のみの入力すれば良い。ただし、一部のパッケージのみをアップデートしたいのであれば、RStudio が提供するアップデート機能を使った方が良いかも知れない<sup>5)</sup>。

また、同じ機能の関数として `p_up()` があるが、コードの可読性のために `p_update()` の方を推奨したい。

```
1 pacman::p_update()
```

---

## 5.6 必須パッケージのインストール

ここでは現在の R において必須パッケージである{tidyverse}をインストールする。{tidyverse}は{dplyr}、{ggplot2}、{tidyverse}など、R において不可欠なパッケージを含むパッケージ群である。また、上で紹介した{devtools}も今のうちにインストールしておこう。既に導入済みの読者は走らせなくても良い。

```
1 install.packages("tidyverse")
2 install.packages("devtools")
```

---

<sup>5)</sup> `p_update(ask = TRUE)` を実行すれば個々のパッケージに対してアップデートするかどうかを決めることができるが、面倒である。

# 第 II 部

## R の基礎



## 第 6 章

# 基本的な操作

### 6.1 コンピュータの基礎知識

R を使い始める前に、コンピュータについて最低限知っておいてほしい基礎知識について説明する。

#### 6.1.1 ファイル名

まず、コンピュータ上のファイル名について説明する。

##### 6.1.1.1 ファイル名拡張子

コンピュータの中には、様々な種類のファイルが含まれている。例えば、多くの人は Microsoft Word や Microsoft Excel のファイルを作ったことがあるだろう。Word や Excel で作ったファイルは、基本的にはそれぞれ専用のソフトウェア（アプリ）で開く必要がある。Word で作ったファイルを Excel で開くことや、Excel で作ったファイルを Word で開くことはできない。

ユーザが特に意識しなくとも、Word で作ったファイルを [ダブル] クリックすれば Word が起動してそのファイルを開くし、Excel で作ったファイルを [ダブル] クリックすれば Excel が起動して開きたいファイルが開かれる。仮に、Word のファイルと Excel ファイルのファイル名が同じ `kadai01` だとしても、パソコンは正しいアプリを選んでく

れる。パソコンがファイルを区別し、正しいアプリを起動してくれるのはなぜだろうか。

実は、各アプリで作ったファイル名は、自分で名前を付けた部分（上の例では `kadai01`）の後に続きがある。Word で作ったファイルには自動的に “.docx” が付けられ、Excel の場合には “.xlsx” が同様に付けられている。したがって、自分では両方のファイルにまったく同じ `kadai01` という名前を付けたつもりでも、実際には、`kadai01.docx` と `kadai01.xlsx` という別のファイル名が付いている。この仕組みにより、パソコンは正しいアプリを選択することができる。ファイル名の末尾に `.docx` があるファイルがクリックされれば Word を、`.xlsx` があるファイルが選択されれば Excel を開くのである。

ファイル名の末尾にあってファイルの種類を区別する部分のことを **ファイル名拡張子 (filename extension)** と呼ぶ。上の例からわかるとおり、ファイル名拡張子はファイルの種類を区別する重要な情報である。プログラミングをしないパソコンユーザにとっては、ファイルの種類の違いを気にせずにパソコンを使えたほうが便利なので、パソコン購入時の初期設定ではファイル名拡張子が非表示になっている場合がある。しかし、プログラミングをする場合（つまり、この本を読んでいるあなた）は、ファイル名拡張子が見えないと困る。

例えば、R は様々な形式で保存されたデータファイルを扱うことができるが、種類に応じてデータを読み込む方法（読み込みに使う関数）が異なる。ファイル名拡張子でファイルの種類を区別し、どの方法を使うかを決めるので、拡張子が表示されていないと不便である（Mac の Finder や Windows のエクスプローラーで「ファイルの種類」を確認できるので、絶対無理というわけではないが、面倒くさい）。よく使うデータファイルのファイル名拡張子として、次のものがある。

- `.csv`
- `.tsv`
- `.txt`
- `.dat`
- `.dta`
- `.RData`
- `.Rds`

後の章で説明するが、この他にも種類がある。ファイルの種類がわからないと、様々な方法を試行錯誤することになってしまい、効率が悪い。ファイル名拡張子があればファイ

ルの種類がわかるので、正しい方法を選んで作業を進めることができる。

よって、R ユーザ（あるいはその他のプログラミングをする者）にとって、ファイル名拡張子の表示は必須である。自分のパソコンでファイル名拡張子が表示されていないなら、ファイル名拡張子を表示する設定に変えよう。macOS では、Finder の環境設定 (Preferences) で、詳細設定 (Advanced) タブを開き、「すべてのファイル名拡張子を表示する (Show all filename extensions)」にチェックマークを付ける。Windows では、エクスプローラー (Explorer)（注意：インターネットエクスプローラーではない。画面下部のタスクバーに表示されている、黄色のフォルダのアイコン）を開き、上部の [表示] タブをクリックする。すると、「ファイル名拡張子」という項目があるので、チェックマークを付ける。チェックマークを付けたら、Word ファイルに .docx（または .doc）、Excel ファイルに .xlsx（または .xls）、PDF ファイルに .pdf などが付いていることを確認しよう。

#### 6.1.1.2 ファイル名の付け方

ファイル名は、ファイルの中身がわかるように付けるのが基本である。例えば、日記を書いて保存するなら、diary\_20200701.txt, diary\_20200702.txt のように名前を付ければ、特定の日付の日記であることがすぐにわかる。

また、他人にファイルを渡す必要があるときは、相手の立場になってファイル名を付けるのが望ましい。例えば、比較政治学 (Comparative Politics) の授業のレポートを Word で書く場合を考えよう。レポートを書いている本人にとっては、cp\_report.docx というファイル名で中身がわかるので問題ないだろう。しかし、これをメールに添付して担当教員に提出するはどうだろうか。受講生が 100 人いて、全員がこの名前でレポートを提出してきたら、担当教員の元には同じ名前のファイルが 100 個届く。これでは、担当教員は困ってしまう。つまり、cp\_report.docx というファイル名は、受け取る相手のことを考えていない、思いやりのないファイル名である。代わりに、学籍番号（例：123456789）を使い、cp\_report\_123456789.docx のようにすると、ファイル名の重複がないので、受け取る相手（私たちのことだが）は喜ぶだろう。自分が 1 つのファイル名を変えるのは大した手間ではないのに対し、相手が全員分のファイル名を変えるの大変な手間だということを理解しよう。

加えて、ファイル名の付け方には形式的なルールがある。ファイル名は、英数字と特定

の記号（\_ [「アンダースコア」、「アンスコ」、「アンダーバー」などと読む] と - [ハイフン]）のみで付けるべきだ。ファイル名に日本語（または韓国語、中国語などのマルチバイト文字）を使うのは愚かなのでやめよう。日本語のファイル名でも問題ない場合が多いのは確かだが、問題がある場合もあるので使用を回避するのが賢い。日本語のファイル名だと、次のような問題が起こりうる。

- 日本語（マルチバイト文字）を扱えないプログラムが停止する。
- さらに悪いと、日本語（マルチバイト文字）を扱えないプログラムが想定外の動作をする。
- 文字コードの違いにより、ファイル名が文字化けする
  - 例えば、日本語のファイル名がついたファイルを Windows で圧縮 (zip) して macOS で展開すると、日本語部分が謎の暗号になるので、どのファイルを開いていいのかわからず、超絶面倒くさい。本当にやめてほしい。お願ひだからやめてください。

日本語のなかでも特に凶悪なのが「全角スペース」である。そもそも、スペースは存在に気付きにくい。万が一末尾にスペースがあると、スペースがない場合との区別が難しい。日本語（マルチバイト文字）を扱えないプログラムでは、半角スペースなら問題ないが、全角でスペースだと問題が起きることがある。しかし、目視で半角スペースと全角スペースの区別をするのは非常に困難である。よって、スペースは使うべきではないし、**全角スペースは絶対に使ってはいけない**。

また、ファイル名の最初の 1 文字はアルファベットにすることが望ましい。ファイルを並べ替えるときに、数字だとややわかりにくいところがある。例えば、1.txt, 12.txt, 110.txt という 3 つのファイルをファイル名で並べ替えると、1.txt, 110.txt, 12.txt という順番になる。多くの場合、これはユーザが期待する順番ではない。中身がわかるようにという大原則にしたがえば、アルファベットから始まるファイル名が自然に選ばれるだろう。

ファイル名の付け方をまとめると、次のようになる。

- ファイル名は、英数字 (A-Z, a-z, 0-9) と \_, - のみで付ける。
  - ただし、ハイフンがあると動かないプログラムもあるので、できればハイフンも避ける。
- ファイル名の 1 文字目はアルファベットにする（数字を 1 文字目にすることを避

ける)。

- . (ドット; ピリオド) は使わない (ファイル名拡張子との混同を避けるため)。
- ファイル名にスペースを使わない。全角スペースはもちろん、半角スペースも避けるべき。- 例えば、my diary.txt というファイル名の代わりに、my\_diary.txt または myDiary.txt、MyDiary.txt などのファイル名を使う。
  - ちなみに、my\_diary.txt のように単語をアンスコで繋ぐ書き方をスネークケース (snake case) [\_ が地を這うヘビである]、myDiary.txt のように単語の 1 文字目を大文字にして前の単語と区別する書き方をキャメルケース (camel case) [大文字部分がラクダのコブである] と呼ぶ。

ただし、次のような例外もある。

- 隠しファイル (Windows 以外では通常は非表示のファイル) の 1 文字目は . である。
  - R ユーザが作る隠しファイルとして、.Rprofile と .Renvironment がある。
- 自分で作るファイル以外には、\_ や # などの記号からファイル名が始まるものもある。

次の節で説明するフォルダ (ディレクトリ) の名前を付ける際も、基本的にはファイル名と同じルールに従うことが望ましい。

### 6.1.2 ファイルシステムとパス

次に、ファイルシステムについて解説する。私たちが使用しているコンピュータには、数千～数万 (あるいはそれ以上) のファイルが含まれている。これらのファイルは基本的には 1 つのドライブ (ハードディスクドライブ [Hard Disk Drive; HDD] またはソリッドステートドライブ [Solid State Drive; SSD]) に保存されているが、ドライブの中にあるファイルはグループ化・階層化されて保存されている。

図 6.1 はファイルシステムの例を示している。矢印の左側には、ドライブ内にあるファイル (の一部) が示されている。通常、これらのファイルは矢印の右側に示されているように、階層化されている。

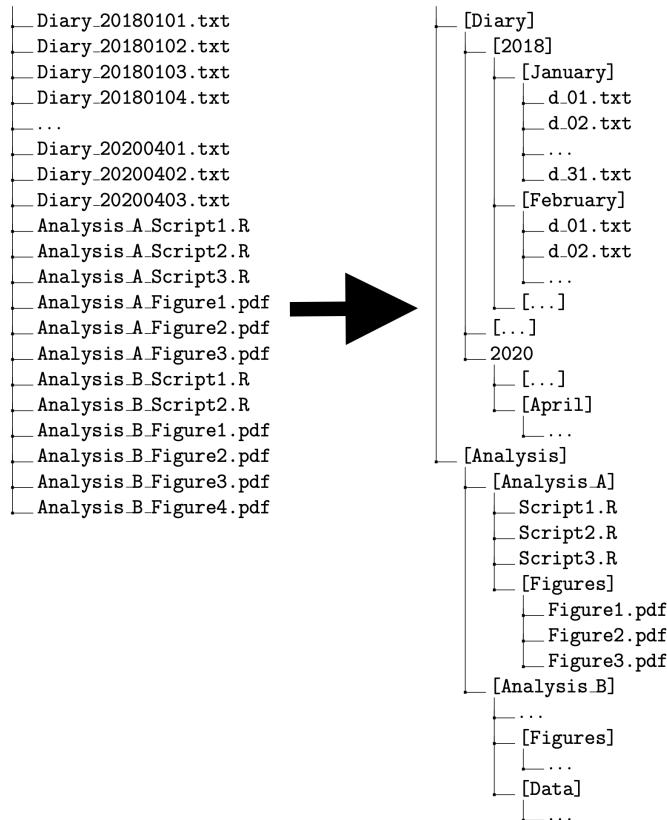


図 6.1: ファイルシステムの例

図 6.1 の左側にある `Diary_YYYYMMDD.txt` が YYYY 年 MM 月 DD 日の日記を保存したテキストファイルだとしよう<sup>1)</sup>。3 年間毎日日記を書くと、それだけでファイル数は 1000 個以上になる。また、`Analysis_blahtable.R` は R スクリプトである<sup>2)</sup>。R スクリプトファイルも複数ある。さらに、上の図には表示されていないが、自分で作ったファイル以外に、OS やソフトウェア（アプリ）を構成するファイルもドライブ内に保存されているだろう。これらのファイルが整理されずに 1 つの場所にまとめて置いてあるとしよう（上の図の左側の状態）。そうすると、特定のファイルを開いたり、それぞれのファイルがどのような目的で存在するのかを把握したりするのに少なからぬ労を要する。

単に面倒なだけならいい（私たちは面倒なことが大嫌いなので良くないと考える）が、

<sup>1)</sup> ファイル名拡張子が “.txt” のファイルを「テキストファイル」と呼ぶ。

<sup>2)</sup> ファイル名拡張子が “.R” または “.r” のファイルを「R スクリプト」と呼ぶ。

ファイルの置き場が 1 つだけだと解決できない問題がある。それは、ファイル名の重複が許されないということだ。世の中には、特定の目的のために使われる「お決まりのファイル名」というものがある。例えば、GitHub にレポジトリを追加するときは、そのレポジトリについて説明する README.md というファイルを作ることになっている。しかし、名前が同じだとファイルが区別できないので、同じ場所にまったく同じ名前のファイルを 2 つ置くことはできない。したがって、ドライブ内にファイル置き場が 1 つしかないとなると、1 つのパソコンで作れる README.md は 1 つだけということになってしまい、困ってしまう。

そこで、多くの OS ではファイルをグループ化して管理するという方法が採用されている。このグループのことを「**フォルダ (folder)**」または「**ディレクトリ (directory)**」と呼ぶ<sup>3)</sup>。

上の図の右側は、ファイルをフォルダに分けた様子を表している。日記のテキストファイルに注目すると、まず、“Diary” という名前のフォルダがあり、Diary フォルダの中に年ごとのフォルダ “2018”, “2019”, “2020” というフォルダがある。それぞれの年のフォルダの中には、“January”, “February”, …, “December” という月ごとのフォルダがある。そして、それぞれの月のフォルダの中に、日付がファイル名になったテキストファイル d\_01.txt, d\_02.txt, … が保存されている。この例からわかるように、フォルダの中にフォルダを作り、そのフォルダの中にフォルダを作り … ということができるので、フォルダを入れ子にした階層構造を利用してファイルを管理することができる。

ファイルを階層化して管理する場合、フォルダの構造と場所を把握することが必要になる。そのために使われるのが**パス (path)** である。パスは、コンピュータ内の住所のようなものだと考えればよい。例えば、“Diary” フォルダ内の “2018” フォルダ内の “January” フォルダ内の “d\_01.txt” というファイルのパスは、Diary/2018/January/d\_01.txt である。この例からわかるように、パスにはフォルダ名やファイル名がそのまま使われる。そして、フォルダの「中」であることは、/ (スラッシュ) 記号によって表される。例えば、Diary/ の部分が、Diary フォルダの中であることを示す。ただし、Windows では / の代わりに \ (バックスラッシュ) または ¥ (円記号; 日本語環境の場合) が使われる<sup>4)</sup>。Diary/2018/January/d\_01.txt と Diary/2018/February/d\_01.txt は、

---

<sup>3)</sup> 本書では、フォルダとディレクトリを同義語として扱う。

<sup>4)</sup> これ以降、原則としてパスの表記には / を使うので、Windows ユーザはご注意を。

ファイル名だけを見れば同じ `d_01.txt` だが、パスが異なるので異なるファイルとして認識され、1つのドライブ内に共存することができる。

しかし、上に書いたパスは、“Diary” フォルダがどこにあるかを指定していないので完全ではない。パソコンがファイルの場所を正しく把握するには、“Diary” フォルダの置き場所がどこかという情報も必要である。

パソコン内でパスの起点になる場所は、OS によって異なる。Linux や macOS では、パスの起点となる最上位フォルダは `/` であり、多くの Windows 機では `C:\` (C ドライブと呼ばれる) である<sup>5)</sup>。また、多くの OS では、ドライブ内に「ホーム (HOME)」と呼ばれる特別なフォルダがあらかじめ用意されており、通常はホームフォルダの中に自分で作ったファイルを保存する。ただし、ホームフォルダの名前は “HOME” ではないので注意が必要である。例えば、macOS ではユーザ名がホームフォルダの名前である。例えば、ユーザ名が `yuki` なら `/Users/yuki/` が、ユーザ名が `jaehyunsong` なら `/Users/jaehyunsong/` がホームフォルダのパスである。パスの先頭に `/` がついており、最上位フォルダの中の “Users” フォルダの中にユーザ名でホームフォルダが作られている。

ホームフォルダの中に `Diary` フォルダがあるとすると、2018年1月1日の日記までのパスは、`/Users/jaehyunsong/Diary/2018/January/d_01.txt` である。このように、ドライブ内の起点から書いた完全なパスを**絶対パス (absolute path)** または**フルパス (full path)** と呼ぶ。絶対パスを使えば、コンピュータ内の特定のファイルを一意に示すことができる。絶対パスにホームディレクトリが含まれている場合には、ホームディレクトリまでのパスを省略して`~[HOME]/` または `~/` と書くことができる。よって、上の絶対パスは、`~/Diary/2018/January/d_01.txt` と書くことができる。この書き方を使えば、パスを書き換えることなく共同研究者とファイルを共有することが可能になる（もちろん、ホームフォルダ以下の構造を揃える必要がある）。

自分が作業・操作の対象としているフォルダは、**作業フォルダ** (working directory; current directory) と呼ばれる<sup>6)</sup>。絶対パスの代わりに、作業フォルダから見た**相対パス**

<sup>5)</sup> かつては、A ドライブと B ドライブがフロッピーディスクに使われていた。その名残で C ドライブが使われている。

<sup>6)</sup> R では、`getwd()` で現在の作業フォルダを確認できる。また、`setwd()` で作業フォルダを変更できる。例えば、`setwd("~/Diary")` とすれば、ホームフォルダ内の `Diary` フォルダを作業フォルダに指定できる。しかし、これらの関数は基本的には使わない。この後説明するとおり、RStudio のプロジェクト

(relative path) を使うこともできる。例えば、現在の作業フォルダが `~/Diary/2018/` だとすると、2018年1月1日の日記への相対パスは、`Januaray/d_01.txt` と書ける。作業フォルダを指定していることを明示したい場合（プログラムを実行する場合にはこれが必要なことがある）には、`./Januaray/d_01.txt` と書く。つまり、`./` が作業フォルダを示す。また、作業フォルダよりも階層が1つ上のフォルダ（親 [parent] フォルダと呼ぶ）には、`../` でアクセスできる。例えば、現在の作業フォルダが `~/Diary/2018/` だとすると、`../` は `~/Diary/` なので、2020年1月1日の日記への相対パスは `../2020/Januaray/d_01.txt` と書くことができる。

相対パスの利点は、

1. 絶対パスより短い
2. 同じ構造を再利用できる

ということである。1は自明だろう。2は、例えば日記の一覧を作るためのプログラムを書き、それを「年」を表すフォルダに保存すれば、1つひとつの日記に毎年同じ相対パスでアクセスできる（ただし、2月29日は除く）ので、毎年同じプログラムを利用できて便利である。絶対パスを使うと、「年」の部分を毎年書き換えなければいけない。

絶対パスと相対パスは、目的に応じて使い分けることが必要である。

## 6.2 「プロジェクト」のすゝめ

Rを使ったプログラミングやデータ分析を進めていくと、自分が書いたRスクリプトや作成した図表だけでなく、Rが自動的に生成するファイルもどんどん溜まる。そう遠くない将来、ドライブ内のファイル数が数万に達しても不思議ではない。効率よくプログラミングを行うために、ファイルの管理方法を明確にしておいたほうが良い。そのためには、フォルダの階層化を利用してファイルを管理することが必要である。

しかし、フォルダによる階層化を導入すればファイルの管理が楽になるかというと、必ずしもそうとは限らない。かえって不便になる部分もある。前の節で見たとおり、フォルダを階層化すると、絶対パスが長く（複雑に）なる。ファイルを階層化によって整理した

としても、ファイルを利用するたびに長い絶対パスの入力が必要なら、ファイル管理の効率が上がったとは言えないだろう。

Rを使う場合にファイル管理の効率化を助けてくれるのが、RStudioの「プロジェクト」機能である。

### 6.2.1 プロジェクト

Rの既定（デフォルト）の作業フォルダはホームフォルダである。分析に使うデータが、ホームフォルダの中の Documents フォルダの中の R フォルダの中の Analysis1 フォルダ内の Data フォルダにある data.csv だとしたら、このファイルにアクセスするためには、"Documents/R/Analysis1/Data/data.csv"と入力する必要がある<sup>7)</sup>、<sup>8)</sup>。新たに作った図を "histogram.pdf" という名前でホームフォルダの中の Documents フォルダの中の R フォルダの中の Analysis1 フォルダ内の Figures というフォルダに保存するためには、"Documents/R/Analysis1/Analysis1/Figures/histogram.pdf"と入力する必要がある。どちらもかなり面倒で、効率が悪い。

しかし、作業フォルダが、~/Documents/R/Analysis1/ だとすれば、相対パスにより、"Data/data.csv" や"Figures/histogram.pdf" だけで済む。よって、作業フォルダを明示的に指定すればいいわけだが、作業フォルダを毎回指定するのも面倒だ。

そこで利用できるのが、RStudioのプロジェクト機能である。プロジェクトとは、特定のフォルダを作業フォルダに設定し、すべての作業をそのフォルダと下位フォルダのみに限定してくれる機能である<sup>9)</sup>。プロジェクト機能さえ使えば、ユーザが意識しなくとも、ユーザが書いたコード、保存したデータ、作成した図などが作業フォルダ内に集約され、管理が楽になる。

では、ここからプロジェクトの作り方を説明しよう。

#### 1. まずは RStudio を起動する。

<sup>7)</sup> Rでパスを指定するときは、パスを引用符で囲む。引用符は、""でも' 'でも良い。

<sup>8)</sup> file.path()を使って、file.path("Documents", "R", "Analysis", "Data", "data.csv")と書くこともできる。この書き方の利点は、/, \, ¥を環境に応じて使い分けてくれることである。

<sup>9)</sup> 何らかの事情により、作業ディレクトリ以外のフォルダにアクセスすることが必要なときは、絶対パスを使えば良い。

2. RStudio が起動したら、“File” から “New Project” を選択する。

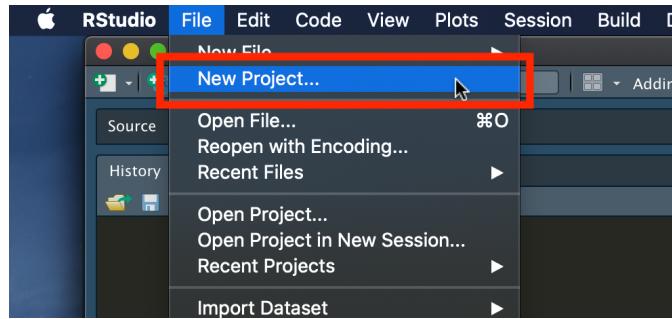


図 6.2: File から New Project...

3. 下の画面が表示されたら、“New Directory” を選択する。ただし、既存のフォルダを利用したい場合は、“Existing Directory” を選ぶ。

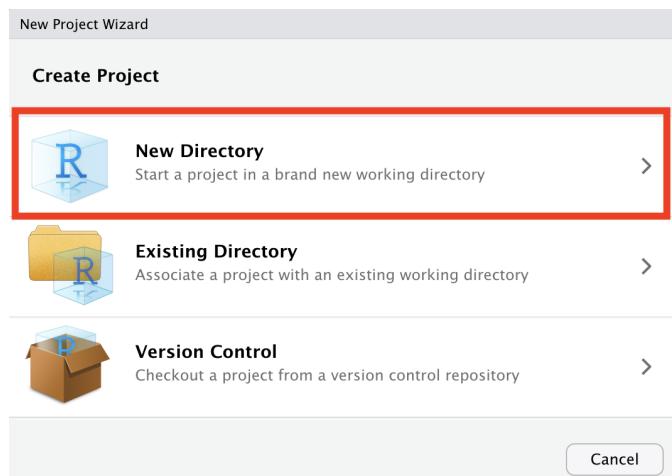


図 6.3: New Directory を選択

4. 下の画面が表示されたら、“New Project” を選択する。

- 前の手順で “Existing Directory” を選択した場合、この画面は表示されない。

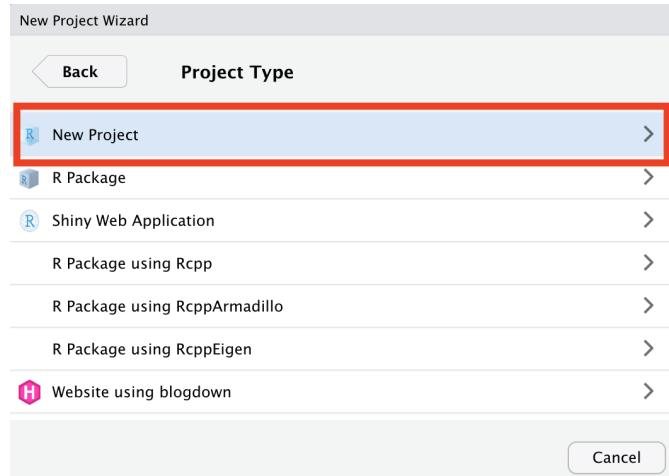


図 6.4: New Project を選択

5. 下の画面が表示されたら、“Directory name:” にプロジェクト名を入力する。これがフォルダ名になるので、**半角英数字のみ**の名前を付ける。ここでは第4章のコードということで、“Ch04” にした。統計学の授業用プロジェクトなら “statistics”、計量政治学の授業なら “quant\_methods\_ps” などの名前を付ければ良いだろう。英語が嫌なら “tokeigaku” のようにすれば良い。また、“Create project as subdirectory of:” では、プロジェクトのフォルダをどのフォルダの中に設置するかを指定する。“Browse...” をクリックし、親フォルダを選ぶ。ここでは ~/Dropbox/RStudy にプロジェクトのフォルダを入れることにする。ここまでできたら、“Create Project” をクリックする。
- 手順 3 で “Existing Directory” を選んだ場合、プロジェクトのフォルダとして使う既存フォルダを選択する画面が表示される。

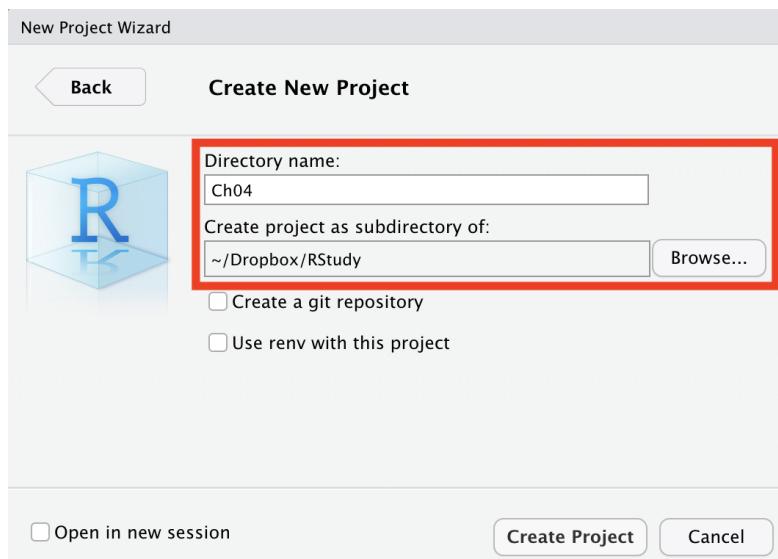


図 6.5: プロジェクト名と保存場所の指定

6. 以上の手順でプロジェクトができる。RStudio 右上に、プロジェクト名が表示されているはずだ。また、Console に `getwd()` と入力すると、プロジェクトまでの絶対パスが表示される。

念のため、Finder（Mac の場合）やエクスプローラー（Windows の場合）で、指定した場所にプロジェクトのフォルダ（上の例では Ch04）が生成されていることを確認しよう。プロジェクトフォルダを開いてみると、`Ch04.Rproj` というファイルが生成されていることがわかる（下の図）。

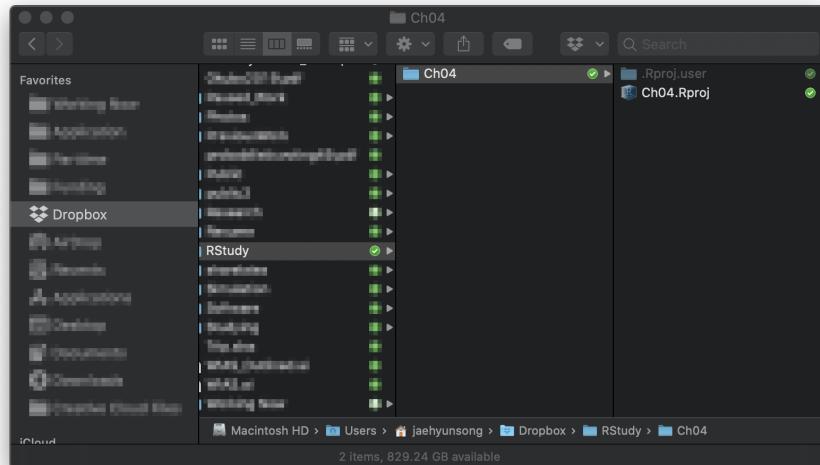


図 6.6: プロジェクトの確認

Ch04 プロジェクトを開くには、この Ch04.Rproj ファイルをダブルクリックすれば良い。RStudio が起動していない場合でも、指定のプロジェクトを開いた状態で RStudio が起ち上がる。

RStudio 右上の表示が “Project: (None)” となっているときは、プロジェクトが開かれていない。RStudio を使う場合には、必ず右上の表示を見て、プロジェクトが開かれていることを確認しよう。プロジェクトが開かれていない場合には、既存のプロジェクトを開くか、新たにプロジェクトを作ろう。

### 6.3 電卓としての R

前置きが長かったが、ここからは R を使っていこう。まず、新しい R Script を開くために、Cmd/Ctrl + Shift + N を入力する<sup>10)</sup> (“File” メニューから “New File” - “R Script” を選んでも良い)。すると、左上の Source Pane に “Untitled1” というタブが登場し、その pane 上でコードが入力できるようになる。ここで  $3 + 3$  と入力し、その行にカーソルを留めたまま command + return (Mac) または Ctrl + Enter を押してみよう。command + return というのは、command キーを押したまま、return キーも押す

<sup>10)</sup> Mac では、command + shift + N、Windows では Ctrl + Shift + N のショートカットが使える。

という意味である。

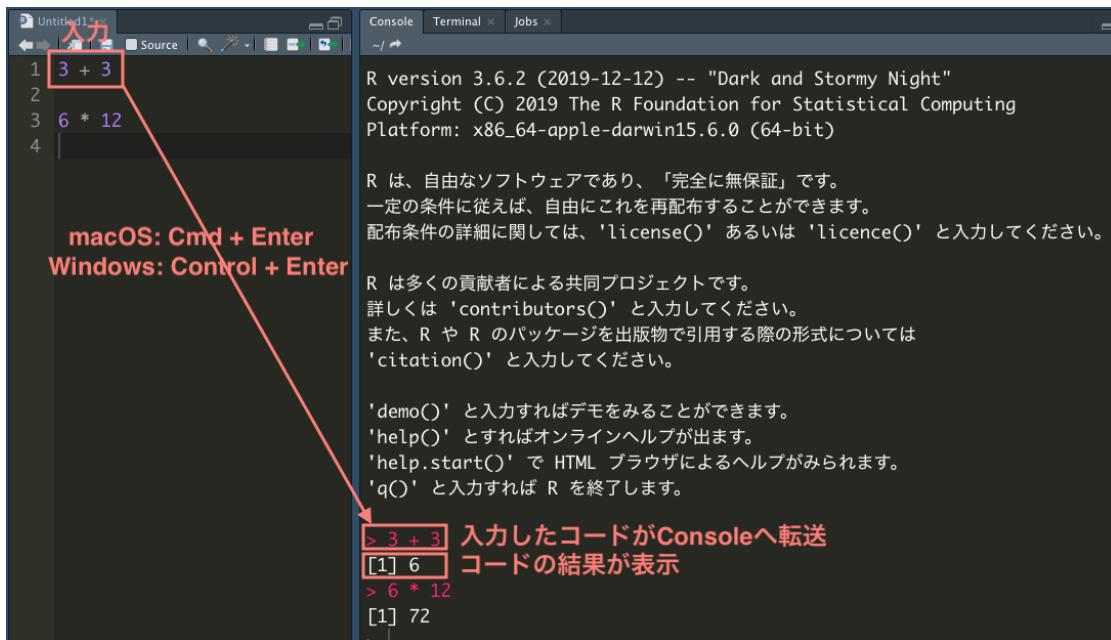


図 6.7: コード入力の例

Source ペイン (Untitled1) に入力したコードが Console (本書の説明どおりにカスタマイズしていれば、RStudio 内の右上画面) に転送され、計算結果が表示される。R のコードは Console に直接打ち込むこともできるが、Source ペインで入力してから Console に転送する方法が基本である<sup>11)</sup>。Source ペインの内容をファイルに保存すれば、後でもう 1 度同じコードを実行したり、コードを他のプロジェクトで再利用することができるようになる。先ほど `3 + 3` を入力した画面にカーソルを合わせ、Cmd/Ctrl + S を押してみよう。ファイルの保存を促されるので、ファイル名をつけて保存しよう。このとき、**.R というファイル名拡張子を付ける**。これにより、ファイルが **R スクリプト** として認識される。例えば、“practice01.R”という名前をつけて保存しよう。Source ペインの上部に表示されるタブの名前が、“Untitled1”から “practice01.R” に変わることが確認できるはずだ。

以下に R で計算する例を示すので、コードを `practice01.R` に入力し、command +

<sup>11)</sup> 電卓として使う程度なら、Console に直接入力してもかまわない。

return (Ctrl + Enter) で Console に送り、実行結果を確認しよう。背景が灰色になっている部分に示されているのが、R のコマンドである。ただし、**##から始まる部分は計算結果である**。また、コードブロックのうち、# (ハッシュ記号) で始まる部分はコメントであり、R で評価（計算）されない。コメントの使い方については第 21 章で詳しく解説する。

### 6.3.1 算術演算子

まずは、簡単な足し算と掛け算を実行してみよう。

```
1 3 + 3
```

```
## [1] 6
```

```
1 8 * 2
```

```
## [1] 16
```

これ以外の基本的な演算は以下のとおりである。

演算子	意味	例	結果
+	和	2 + 5	7
-	差	2 - 8	-6
*	積	7 * 3	21
/	商	16 / 5	3.2
^、**	累乗（べき乗）	2^3 または 2 ** 3	8
%%	剰余（モジュロ）	18 %% 7	4
%/%	整数商	18 %/% 7	2

### 6.3.2 論理演算子

論理演算子とは、入力した式が真か偽かを判定する演算子である。返り値（戻り値）は TRUE（真の場合）または FALSE（偽の場合）のいずれかとなる。たとえば、「3 > 2」

は真なので、TRUE が返される。しかし、「 $2 + 3 = 1$ 」は偽なので、FALSE が返される。  
実際にやってみよう。

```
1 3 > 2
## [1] TRUE
1 2 + 3 == 1
## [1] FALSE
```

このように、等しいかどうかを表す記号は `=` ではなく `==`（二重等号）なので注意されたい。

論理演算子にも、いくつかの種類がある。

演算子	意味	例	結果
1 <code>x &lt; y</code>	<code>x</code> は <code>y</code> より小さい	<code>3 &lt; 1</code>	FALSE
2 <code>x &lt;= y</code>	<code>x</code> は <code>y</code> と等しいか、小さい	<code>2 &lt;= 2</code>	TRUE
3 <code>x &gt; y</code>	<code>x</code> は <code>y</code> より大きい	<code>6 &gt; 5</code>	TRUE
4 <code>x &gt;= y</code>	<code>x</code> は <code>y</code> と等しいか、大きい	<code>4 &gt;= 5</code>	FALSE
5 <code>x == y</code>	<code>x</code> と <code>y</code> は等しい	<code>(2 + 3) == (4 + 1)</code>	TRUE
6 <code>x != y</code>	<code>x</code> と <code>y</code> は等しくない	<code>((2 * 3) + 1) != (2 * (3 + 1))</code>	TRUE

6 番目の例について少し説明する。通常の数式同様、R も括弧 () 内の記述を優先的に計算する。したがって、`!=` 左側の `((2 * 3) + 1)` は  $6 + 1 = 7$  であり、右側の `(2 * (3 + 1))` は  $2 * 4 = 8$  である。したがって、`7 != 8` が判定対象となり、TRUE が返される。`!` 記号は、「否定」を表すために使われるもので、`!=` は左右が等しくないときに TRUE を返す

上に挙げた論理演算子は基本的に数字を対象に使うが、TRUE と FALSE を対象に使うものもある。それが `and` を表す `&` と `or` を表す `|` である。。`&` は、`&` を挟む左右の両側が TRUE の場合のみ TRUE を返し、`|` は少なくとも一方が TRUE なら TRUE を返す。

演算子	意味	例	結果
1 x   y	x または y	$(2 + 3 == 5)   (1 * 2 == 3)$	TRUE
2 x & y	x かつ y	$(2 + 3 == 5) & (1 * 2 == 3)$	FALSE

1番の例では、|の左側は  $(2 + 3 == 5)$  であり、TRUE である。一方、右側の  $(1 * 2 == 3)$  は FALSE だ。判定対象は TRUE | FALSE となり、TRUE が返される。2番目の例は TRUE & FALSE なので、返り値は FALSE になる。

## 6.4 格納とオブジェクトの作成

- まず、123454321 \* 2 を計算しよう。
- 次に、123454321 \* 3 を計算しよう。
- 最後に、123454321 \* 4 を計算しよう。

これらの計算は簡単にできるだろう。しかし、123454321 を 3 回入力するのが面倒だっただろう<sup>12)</sup>。123454321 という数字を x とか a に代入し、数字の代わりに x や a が使えるなら、上の計算は楽になる。ここではその方法を説明する。

x というのに 123454321 という数字を入れるには、<- という演算子を使う。この演算子により、x という名のもの（オブジェクト）に 123454321 という数字を代入することができる。ここでは「代入」という表現を使ったが<sup>13)</sup>、<-の役割は代入よりも広いので、これからは「格納」という表現を使う。

<- は、< と- という **2 つの記号をスペースなしで** 入力することで作ることができます。RStudio では、option + - [マイナス, ハイフン] (macOS 場合) または Alt + - (Windows の場合) で、<- が入力できる。その際、演算子の前後に半角スペースが 1 つずつ挿入されるので、**このショートカットは必ず使うべき**である。

R を起動した時点では、x というオブジェクトは存在しない。RStudio 右下のペインにある Environment タブを開くと、現時点では何も表示されていないはずだ。しかし、

<sup>12)</sup> 反復作業が面倒だと感じるほどプログラミングが上達しやすい。

<sup>13)</sup> 代入の代わりに「付値」と言うこともある。

`x` に何かを格納することで、`x` というオブジェクトができる。実際に `x` に、123454321 を格納してみよう。

```
1 x <- 123454321 # x に 123454321 を格納
```

Environment タブに、`x` が登場し、格納した数字が右側に表示されていることが確認できるだろう。

オブジェクトの中身は、オブジェクト名をそのまま入力することで表示できる。

```
1 x
```

```
## [1] 123454321
```

`print(x)` でも同じ結果が得られるが、タイプする文字数を減らしたいので `x` のみにする。ただし、状況やオブジェクトの型（型については後で詳しく説明する）によっては、`print()` を表示しないと中身が表示されない場合もあるので、R Markdown ファイルで論文などを作成していて、結果を確実に表示したい場合には `print(x)` とするほうが安全である。

ちなみに、格納と同時にそのオブジェクトの中身を表示することもできる。そのためには、格納コマンド全体を () で囲む。例えば、次のようにする。

```
1 (y <- 2) # y に 2 を格納し、中身を表示
```

```
## [1] 2
```

値が格納されたオブジェクトは計算に利用できるので、先ほどの計算は、次のようにできる。

```
1 x * 2
```

```
## [1] 246908642
```

```
1 x * 3
```

```
## [1] 370362963
```

```

1 x * 4

## [1] 493817284

```

文字列を格納することもできる。ただし、文字列は必ず "" か ' ' で囲む必要がある。

```

1 x <- "猫の恋 やむとき闇の 龍月（芭蕉）"
2 x

```

```
## [1] "猫の恋 やむとき闇の 龍月（芭蕉）"
```

オブジェクトに格納できるのは1つの数値や文字列だけではない。複数の数値や文字列を格納することもできる。そのためには `c()` という関数を使う。`c()` の `c` は `concatenate` または `combine` の頭文字で、複数の要素からベクトル (vector) を作るのに使われる関数である。`c()` に含む要素はカンマ (,) で区切る。

```

1 # ある日の Lions の打順をベクトルに格納する
2 numeric_vec1 <- c(73, 6, 5, 3, 99, 10, 22, 9, 7)
3 numeric_vec1

```

```
## [1] 73 6 5 3 99 10 22 9 7
```

複数の文字列を格納することもできる。

```

1 character_vec <- c('cat', 'cheetah', 'lion', 'tiger')
2 character_vec

## [1] "cat"      "cheetah"  "lion"     "tiger"

```

ひとつひとつの要素を指定する代わりに、様々な方法でベクトルを作ることが可能である。たとえば、`seq()` 関数を使うと、一連の数字からなるベクトルを作ることができ。`from` で数列の初項を、`to` で数列の最終項を指定し、`by` で要素間の差（第2要素は第1要素に `by` を加えた値になる）を指定するか、`length.out` で最終的にできるベクトルの要素の数を指定する。Rのベクトルの `length` とは、要素の数のことなので、注意されたい。

いくつか例を挙げる。

```
1 # 1 から 20 までの整数。1:20 でも同じ
2 seq(from = 1, to = 20, by = 1)

## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 # 1 から 19 までの奇数
2 seq(from = 11, to = 20, by = 2)

## [1] 11 13 15 17 19

1 # 2 から 20 までの偶数
2 seq(from = 2, to = 20, by = 2)

## [1] 2 4 6 8 10 12 14 16 18 20

1 # 降順、間隔は 5
2 seq(from = 20, to = 1, by = -5)

## [1] 20 15 10 5

1 # 最小値が 1、最大値が 100 で、長さが 10 のベクトル
2 seq(from = 1, to = 100, length.out = 10)

## [1] 1 12 23 34 45 56 67 78 89 100

1 # 73 から 1 つずつ数が小さくなる長さが 10 のベクトル
2 seq(from = 73, by = -1, length.out = 8)

## [1] 73 72 71 70 69 68 67 66
```

このように 1 つの関数でも指定する内容は、`by` になったり `length.out` になったりする。`by` や `length.out`、`from`、`to` などのように、関数で指定する対象になっているもののことを **仮引数 (parameter)** と呼ぶ。また、`by = 1` の 1 や、`length.out = 10` の 10 のように、仮引数に実際に渡される値のことを**実引数 (argument)** と呼ぶ。特に誤解が生じないと思われる場合には、仮引数と実引数を区別せずに**引数 (ひきすう)** と呼ぶ。R では、1 つの関数で使う引数の数が複数あることが多いので、**仮引数を明示する**習慣を

身につけたほうがよい。ただし、第1引数（関数で最初に指定する引数）として必ず入力すべきものは決められている場合がほとんどなので、第1引数の仮引数は省略されることが多い。仮引数が省略される代わりに、第1引数の実引数はほぼ必ず入力する（いくつかの例外もある）。

`seq(from = x, to = y, by = 1)` の場合はより単純に `x:y` とすることができる。

```
1  21:30  # 21 から 30 までの整数
## [1] 21 22 23 24 25 26 27 28 29 30
1  10:1  # 10 から 1 までの整数 (降順)
## [1] 10 9 8 7 6 5 4 3 2 1
```

また、`rep()` 関数も便利である。例を挙げよう。

```
1  # 3 が 10 個のベクトル
2  rep(3, times = 10)
## [1] 3 3 3 3 3 3 3 3 3 3
1  # a が 3 つ, b が 1 つ, c が 2 つのベクトル
2  rep(c('a', 'b', 'c'), times = c(3, 1, 2))
## [1] "a" "a" "a" "b" "c" "c"
1  # C, A, T を 2 つずつ
2  rep(c('C', 'A', 'T'), each = 2)
## [1] "C" "C" "A" "A" "T" "T"
```

アルファベットのベクトルは、あらかじめ用意されている。

```
1  LETTERS # 大文字
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
1 letters # 小文字

## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

## 6.5 要素の抽出

オブジェクト名(ベクトル)の後に [抽出する要素のインデクス] を付けると、ベクトルの特定の要素を抽出することができる。ちなみに [ は関数である。Console に `help("[")` と打てば、これが Extract と言う名前の関数であることがわかる (`help("[]")` ではないので注意)。

ベクトルの要素を取り出してみよう。R のインデクスは、他の多くのプログラミング言語(例えば、C, C++, Python など)とは異なり「1」から始まるので注意されたい。

```
1 # numeric_vec1 の 5 番目の要素を抽出
2 numeric_vec1[5]
```

```
## [1] 99

1 # numeric_vec1 の 2, 4, 6 番目の要素を抽出
2 numeric_vec1[c(2, 4, 6)]
```

```
## [1] 6 3 10

1 # numeric_vec1 の 1 番目と 4 番目「以外」の要素を抽出
2 numeric_vec1[-c(1, 4)]
```

```
## [1] 6 5 99 10 22 9 7

1 # numeric_vec1 の 5 番目から 7 番目の要素を抽出
2 numeric_vec1[5:7]
```

```
## [1] 99 10 22
```

`c()` や `:` だけでなく、`seq()` も使える。

```

1 numeric_vec2 <- 1:20
2 # numeric_vec2 の奇数番目の要素を抽出
3 numeric_vec2[seq(1, 20, by = 2)]
## [1] 1 3 5 7 9 11 13 15 17 19

```

さらに、TRUE と FALSE を使うこともできる。この場合、抽出したい要素の場所を指定するのではなく、**それぞれの場所について抽出する (TRUE) か、しない (FALSE) かを指定する**。たとえば、character\_vec から 1, 3, 4 番目の要素を抽出するなら、[c(TRUE, FALSE, TRUE, TRUE)] と指定する。

```

1 character_vec[c(TRUE, FALSE, TRUE, TRUE)]
## [1] "cat"    "lion"   "tiger"

```

TRUE と FALSE が使えるので、論理演算子を [] の中で使うこともできる。たとえば、numeric\_vec1 の各要素が偶数かどうかを判定するためには、インデックスが 2 で割り切れるかどうか（2 で割った余りが 0 かどうか）を確認すれば良い。

```

1 (numeric_vec1 %% 2) == 0 # 偶数かどうかの判定
## [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE

```

これを利用すれば、numeric\_vec1 から偶数のみを抽出できる。

```

1 numeric_vec1[(numeric_vec1 %% 2) == 0]
## [1] 6 10 22

```

[と格納（代入）を組み合わせれば、「ベクトルの一部の要素を書き換える」ことができる。たとえば、numeric\_vec1 の 2 番目の要素は 5 だが、これを 100 に書き換えた場合、置換したい要素の場所を [] で指定し、<-で代入すれば良い。

```

1 numeric_vec1[2] <- 100
2 numeric_vec1
## [1] 73 100  5  3 99 10 22  9  7

```

複数の要素を置換することもできる。たとえば、偶数を全て 0 に置換したい場合、以下のようにする。

```
1 numeric_vec1[(numeric_vec1 %% 2) == 0] <- 0
2 numeric_vec1
## [1] 73 0 5 3 99 0 0 9 7
```

---

## 演習問題

**問 1** `my_vec1` という名のオブジェクトに  $(3, 9, 10, 8, 3, 5, 8)$  を格納し、表示せよ。

**問 2** `my_vec1` から 2, 4, 6 番目の要素を抽出せよ。

**問 3** `my_vec1` の要素の和を求めよ。

- ベクトル内要素の和は `sum(ベクトル名)` で計算できる。たとえば、ベクトル `c(1, 3, 5)` の和は `sum(c(1, 3, 5))` である。

**問 4** `my_vec1` の要素から奇数のみを抽出し、その和を求めよ。

**問 5** `my_vec2` という名のオブジェクトに長さ 7 のベクトル、 $(1, 2, 3, 4, 3, 2, 1)$  を格納し、表示せよ。

**問 6** `my_vec1` と `my_vec2` のそれぞれの要素の和を計算し、`my_vec3` に格納し、表示せよ。

**問 7** `my_vec3` の要素から 10 未満の要素を抽出せよ。

**問 7** 1 から 100 までの公差 1 の数列  $(1, 2, 3, 4, 5, \dots, 100)$  を作成し、`my_vec4` と名付けよ。

**問 9** `my_vec4` の各要素を二乗した総和  $(1^2 + 2^2 + 3^2 + \dots + 100^2)$  を求めよ。

**問 10** `my_vec4` から奇数のみ抽出し、二乗和  $(1^2 + 3^2 + 5^2 + 7^2 + \dots + 97^2 + 99^2)$  を求めよ。



## 第 7 章

# データの入出力

この章で使うパッケージを読み込む。

```
1 pacman::p_load(tidyverse, readxl, haven)
```

## 7.1 データの読み込み

### 7.1.1 csv ファイルの場合

csv は、comma separated values の略である。多くの人に馴染みがあると思われる Excel ファイル (.xlsx) と同じように、表形式（行列形式）のデータを保存できる。しかし、Excel ファイルとは異なり、文字の大きさ、セルの背景色、複数のセルの結合のような情報はもたず、純粋にデータに含まれる変数の名前と各変数の値（数値・文字列）のみが格納されているため、ファイルサイズが小さい。また、文字データしかもたないテキストファイルであり、テキストエディタで開くことができる。テキストファイルで csv ファイルを開けば、“csv”と呼ばれる理由がわかるだろう。csv フォーマットはデータを保存のための標準フォーマットの 1 つであり、多くのデータが csv 形式で保存されている。データ分析ソフトで csv ファイルを開けないものはおそらくないだろう。

R でも csv 形式のファイルは簡単に読み込める。実際にやってみよう。データのダウンロード方法については本書の巻頭を参照されたい。csv ファイルを読み込むに

は `read.csv()` または `readr::read_csv()` 関数を使う<sup>1)</sup>。読み込む際は前章のベクトルの生成同様、何らかの名前を付けて作業環境に保存する。Data フォルダーにある `FIFA_Women.csv` ファイルを読み込み、`my_df1` と名付ける場合、以下のようなコードを実行する<sup>2)</sup>。以下のコードで `my_df1 <-` の部分を入力しないと、データが画面に出力され、自分の作業スペースには保存されないので注意されたい。

```
1 my_df1 <- read.csv("Data/FIFA_Women.csv")
```

読み込まれたデータの中身を見るには、ベクトルの場合と同様に `print()` 関数を使うか、オブジェクト名を入力する。

```
1 my_df1
```

##	ID	Team	Rank	Points	Prev_Points	Confederation
## 1	1	Albania	75	1325	1316	UEFA
## 2	2	Algeria	85	1271	1271	CAF
## 3	3	American Samoa	133	1030	1030	OFC
## 4	4	Andorra	155	749	749	UEFA
## 5	5	Angola	121	1117	1117	CAF
## 6	6	Antigua and Barbuda	153	787	787	CONCACAF
## 7	7	Argentina	32	1659	1659	CONMEBOL
## 8	8	Armenia	126	1103	1104	UEFA
## 9	9	Aruba	157	724	724	CONCACAF
## 10	10	Australia	7	1963	1963	AFC
## 11	11	Austria	22	1792	1797	UEFA
## 12	12	Azerbaijan	76	1321	1326	UEFA
## 13	13	Bahrain	84	1274	1274	AFC
## 14	14	Bangladesh	134	1008	1008	AFC
## 15	15	Barbados	135	1002	1002	CONCACAF
## 16	16	Belarus	53	1434	1437	UEFA

<sup>1)</sup> `read.csv()` で読み込まれた表は `data.frame` クラスに、`read_csv()` 関数で読み込まれた表は `tibble` クラスとして保存される。詳細は第9章で解説する。

<sup>2)</sup> R のバージョンが 4.0.0 未満の場合は、引数として `stringsAsFactors = FALSE` を追加する。たとえば、`my_df1 <- read.csv("Data/FIFA_Women.csv", stringsAsFactors = FALSE)` とする。これを追加しないと、文字列で構成されている列が `factor` 型として読み込まれる。

## 17	17	Belgium	17	1819	1824	UEFA
## 18	18	Belize	150	824	824	CONCACAF
## 19	19	Bermuda	136	987	987	CONCACAF
## 20	20	Bhutan	154	769	769	AFC
## 21	21	Bolivia	91	1236	1236	CONMEBOL
## 22	22	Bosnia and Herzegovina	59	1411	1397	UEFA
## 23	23	Botswana	148	848	848	CAF
## 24	24	Brazil	8	1958	1956	CONMEBOL
## 25	25	Bulgaria	79	1303	1303	UEFA
## 26	26	Cameroon	51	1455	1486	CAF
## 27	27	Canada	8	1958	1958	CONCACAF
## 28	28	Chile	37	1640	1637	CONMEBOL
## 29	29	China PR	15	1867	1842	AFC
## 30	30	Chinese Taipei	40	1589	1584	AFC
## 31	31	Colombia	25	1700	1700	CONMEBOL
## 32	32	Comoros	156	731	731	CAF
## 33	33	Congo	104	1178	1178	CAF
## 34	34	Congo DR	110	1159	1159	CAF
## 35	35	Cook Islands	103	1194	1194	OFC
## 36	36	Costa Rica	36	1644	1630	CONCACAF
## 37	37	Côte d'Ivoire	63	1392	1392	CAF
## 38	38	Croatia	52	1453	1439	UEFA
## 39	39	Cuba	88	1240	1240	CONCACAF
## 40	40	Cyprus	123	1114	1123	UEFA
## 41	41	Czech Republic	29	1678	1678	UEFA
## 42	42	Denmark	16	1851	1839	UEFA
## 43	43	Dominican Republic	105	1173	1173	CONCACAF
## 44	44	El Salvador	109	1164	1164	CONCACAF
## 45	45	England	6	1999	2001	UEFA
## 46	46	Equatorial Guinea	71	1356	1356	CAF
## 47	47	Estonia	95	1210	1206	UEFA
## 48	48	Eswatini	151	822	822	CAF

## 49	49	Ethiopia	111	1151	1151	CAF
## 50	50	Faroe Islands	86	1259	1262	UEFA
## 51	51	Fiji	66	1373	1373	OFC
## 52	52	Finland	30	1671	1678	UEFA
## 53	53	France	3	2036	2033	UEFA
## 54	54	Gabon	130	1066	1066	CAF
## 55	55	Gambia	113	1143	1183	CAF
## 56	56	Georgia	115	1138	1145	UEFA
## 57	57	Germany	2	2090	2078	UEFA
## 58	58	Ghana	60	1401	1404	CAF
## 59	59	Greece	62	1396	1395	UEFA
## 60	60	Guam	82	1282	1282	AFC
## 61	61	Guatemala	80	1290	1290	CONCACAF
## 62	62	Haiti	64	1391	1368	CONCACAF
## 63	63	Honduras	116	1136	1136	CONCACAF
## 64	64	Hong Kong	74	1329	1335	AFC
## 65	65	Hungary	43	1537	1526	UEFA
## 66	66	Iceland	19	1817	1821	UEFA
## 67	67	India	55	1432	1432	AFC
## 68	68	Indonesia	94	1222	1222	AFC
## 69	69	IR Iran	70	1358	1358	AFC
## 70	70	Israel	67	1369	1371	UEFA
## 71	71	Italy	14	1889	1882	UEFA
## 72	72	Jamaica	50	1460	1461	CONCACAF
## 73	73	Japan	11	1937	1942	AFC
## 74	74	Jordan	58	1419	1419	AFC
## 75	75	Kazakhstan	77	1318	1318	UEFA
## 76	76	Kenya	137	986	986	CAF
## 77	77	Korea DPR	10	1940	1940	AFC
## 78	78	Korea Republic	18	1818	1812	AFC
## 79	79	Kosovo	125	1104	1109	UEFA
## 80	80	Kyrgyz Republic	120	1118	1118	AFC

## 81	81	Latvia	93	1223	1223	UEFA
## 82	82	Lebanon	141	967	967	AFC
## 83	83	Lesotho	147	850	850	CAF
## 84	84	Lithuania	107	1169	1168	UEFA
## 85	85	Luxembourg	119	1124	1124	UEFA
## 86	86	Madagascar	158	691	691	CAF
## 87	87	Malawi	145	887	887	CAF
## 88	88	Malaysia	90	1238	1238	AFC
## 89	89	Maldives	142	966	966	AFC
## 90	90	Mali	83	1276	1276	CAF
## 91	91	Malta	101	1197	1195	UEFA
## 92	92	Mauritius	159	357	357	CAF
## 93	93	Mexico	27	1686	1699	CONCACAF
## 94	94	Moldova	92	1228	1229	UEFA
## 95	95	Mongolia	123	1114	1114	AFC
## 96	96	Montenegro	97	1201	1206	UEFA
## 97	97	Morocco	81	1289	1280	CAF
## 98	98	Mozambique	152	814	814	CAF
## 99	99	Myanmar	45	1511	1527	AFC
## 100	100	Namibia	143	956	956	CAF
## 101	101	Nepal	99	1200	1200	AFC
## 102	102	Netherlands	4	2032	2035	UEFA
## 103	103	New Caledonia	96	1208	1208	OFC
## 104	104	New Zealand	23	1757	1760	OFC
## 105	105	Nicaragua	122	1116	1116	CONCACAF
## 106	106	Nigeria	38	1614	1614	CAF
## 107	107	North Macedonia	129	1072	1073	UEFA
## 108	108	Northern Ireland	55	1432	1433	UEFA
## 109	109	Norway	12	1930	1929	UEFA
## 110	110	Palestine	117	1131	1131	AFC
## 111	111	Panama	60	1401	1437	CONCACAF
## 112	112	Papua New Guinea	46	1504	1504	OFC

## 113 113	Paraguay	48	1490	1490	CONMEBOL
## 114 114	Peru	65	1376	1376	CONMEBOL
## 115 115	Philippines	67	1369	1369	AFC
## 116 116	Poland	28	1683	1677	UEFA
## 117 117	Portugal	32	1659	1667	UEFA
## 118 118	Puerto Rico	106	1172	1172	CONCACAF
## 119 119	Republic of Ireland	31	1666	1665	UEFA
## 120 120	Romania	44	1535	1542	UEFA
## 121 121	Russia	24	1708	1708	UEFA
## 122 122	Rwanda	144	899	899	CAF
## 123 123	Samoa	107	1169	1169	OFC
## 124 124	Scotland	21	1804	1794	UEFA
## 125 125	Senegal	87	1247	1245	CAF
## 126 126	Serbia	41	1558	1553	UEFA
## 127 127	Singapore	128	1089	1089	AFC
## 128 128	Slovakia	47	1501	1500	UEFA
## 129 129	Slovenia	49	1471	1467	UEFA
## 130 130	Solomon Islands	114	1140	1140	OFC
## 131 131	South Africa	53	1434	1434	CAF
## 132 132	Spain	13	1915	1900	UEFA
## 133 133	Sri Lanka	140	968	968	AFC
## 134 134	St. Kitts and Nevis	131	1050	1054	CONCACAF
## 135 135	St. Lucia	138	982	982	CONCACAF
## 136 136	Suriname	127	1093	1093	CONCACAF
## 137 137	Sweden	5	2007	2022	UEFA
## 138 138	Switzerland	20	1815	1817	UEFA
## 139 139	Tahiti	102	1196	1196	OFC
## 140 140	Tajikistan	132	1035	1035	AFC
## 141 141	Tanzania	139	978	978	CAF
## 142 142	Thailand	39	1596	1620	AFC
## 143 143	Tonga	88	1240	1240	OFC
## 144 144	Trinidad and Tobago	72	1354	1354	CONCACAF

```

## 145 145           Tunisia 78 1304      1313      CAF
## 146 146           Turkey 69 1365      1361      UEFA
## 147 147           Uganda 146 868       868       CAF
## 148 148           Ukraine 26 1692      1697      UEFA
## 149 149 United Arab Emirates 97 1201      1201      AFC
## 150 150           Uruguay 73 1346      1346      CONMEBOL
## 151 151 US Virgin Islands 149 843       843       CONCACAF
## 152 152           USA     1 2181      2174      CONCACAF
## 153 153           Uzbekistan 42 1543      1543      AFC
## 154 154           Vanuatu 117 1131      1131      OFC
## 155 155           Venezuela 57 1425      1425      CONMEBOL
## 156 156           Vietnam 35 1657      1665      AFC
## 157 157           Wales    34 1658      1659      UEFA
## 158 158           Zambia 100 1198      1167      CAF
## 159 159           Zimbabwe 111 1151      1151      CAF

```

しかし、通常はデータが読み込まれているかどうかを確認するためにデータ全体を見る必要はない。最初の数行のみで問題ないはずだ。そのために、`head()` 関数を使う。これは最初の 6 行 (`n` で表示行数を変える) を表示してくれる。

```
1 head(my_df1)
```

```

##   ID           Team Rank Points Prev_Points Confederation
## 1  1       Albania 75 1325      1316      UEFA
## 2  2       Algeria 85 1271      1271      CAF
## 3  3 American Samoa 133 1030      1030      OFC
## 4  4      Andorra 155  749       749      UEFA
## 5  5       Angola 121 1117      1117      CAF
## 6  6 Antigua and Barbuda 153  787       787      CONCACAF

```

同様に、最後の `n` 行は `tail()` で表示する。

```
1 tail(my_df1, n = 9)
```

```

##   ID           Team Rank Points Prev_Points Confederation

```

```

## 151 151 US Virgin Islands 149 843 843 CONCACAF
## 152 152 USA 1 2181 2174 CONCACAF
## 153 153 Uzbekistan 42 1543 1543 AFC
## 154 154 Vanuatu 117 1131 1131 OFC
## 155 155 Venezuela 57 1425 1425 CONMEBOL
## 156 156 Vietnam 35 1657 1665 AFC
## 157 157 Wales 34 1658 1659 UEFA
## 158 158 Zambia 100 1198 1167 CAF
## 159 159 Zimbabwe 111 1151 1151 CAF

```

ただし、今後、特殊な事情がない限り、データの読み込みは `read.csv()` を使用せず、`read_csv()` を使用しますが、使い方は同じです。`read_csv()` は `{tidyverse}` の一部である `{readr}` パッケージに含まれている関数であるため、あらかじめ `{tidyverse}` を読み込んでおく必要があります。

```

1 pacman::p_load(tidyverse)
2 my_df1 <- read_csv("Data/FIFA_Women.csv")
3
4 my_df1

## # A tibble: 159 x 6
##       ID Team      Rank Points Prev_Points Confederation
##   <dbl> <chr>   <dbl>   <dbl>      <dbl> <chr>
## 1     1 Albania 75 1325 1316 UEFA
## 2     2 Algeria 85 1271 1271 CAF
## 3     3 American Samoa 133 1030 1030 OFC
## 4     4 Andorra 155 749 749 UEFA
## 5     5 Angola 121 1117 1117 CAF
## 6     6 Antigua and Barbuda 153 787 787 CONCACAF
## 7     7 Argentina 32 1659 1659 CONMEBOL
## 8     8 Armenia 126 1103 1104 UEFA
## 9     9 Aruba 157 724 724 CONCACAF
## 10    10 Australia 7 1963 1963 AFC
## # ... with 149 more rows

```

同じファイルが読み込まれましたが、データを出力する際、最初の 10 行のみが表示されます。また、画面に収まらない横長のデータであれば、適宜省略し、見やすく出力してくれます。`read_csv()` で読み込まれた表形式データは `tibble` と呼ばれるやや特殊なものとして格納されます。R がデフォルトで提供する表形式データの構造は `data.frame` ですが、`tibble` はその拡張版です。詳細は第 9.4 章を参照してください。

### 7.1.2 エンコーディングの話

`Vote_ShiftJIS.csv` は Shift-JIS でエンコーディングされた csv ファイルである。このファイルを `read.csv()` 関数で読み込んでみよう。

```
1 ShiftJIS_df <- read.csv("Data/Vote_ShiftJIS.csv")
## Error in type.convert.default(data[[i]], as.is = as.is[i], dec = dec, :
# invalid multi
```

Windows ならなんの問題なく読み込まれるだろう。しかし、macOS の場合、以下のようなエラーが表示され、読み込めない。

```
## Error in type.convert.default(data[[i]], as.is = as.is[i], dec = dec, :
# ' <96>k<8a>C<
不正なマルチバイト文字があります
```

このファイルは、`read.csv()` の代わりに `readr` パッケージの `read_csv()` を使えば読み込むことができる<sup>3)</sup>。`read_csv()` で読み込み、中身を確認してみよう。

```
1 ShiftJIS_df1 <- read_csv("Data/Vote_ShiftJIS.csv")
2 head(ShiftJIS_df1)

## # A tibble: 6 x 11
##       ID Pref      Zaisei Over65 Under30    LDP    DPJ   Komei   Ishin    JCP    SDP
##   <dbl> <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1     1 "x96k\x8aC\x~  0.419    29.1    24.7    32.8    30.6   13.4    3.43   11.4    1.68
## 2     2 "x90\xc2\x90~  0.332    30.1    23.9    40.4    24.6   12.8    3.82   8.92    3.41
## 3     3 "x8a\xe2\x8e~  0.341    30.4    24.5    34.9    22.4    8.61    5.16   11.2    5.29
## 4     4 "x8b{\x8f\xe~  0.596    25.8    27.3    36.7    25.4   13.4    3.97   9.99    3.62
```

<sup>3)</sup> `readr` パッケージは `tidyverse` の一部なので、上で読み込み済みである。

```
## 5      5 "\x8fH\x93c\x~  0.299   33.8    21.4   43.5   22.7 11.2   5.17   7.56   5.1
## 6      6 "\x8eR\x8c`\x~  0.342   30.8    24.8   42.5   21.5 11.8   4.3    7.6    5.2
```

2列目の Pref 列には日本語で都道府県名が入っているはずだが、謎の文字列が表示される。read.csv() の場合、少なくとも Windows では問題なく読み始めたはずだ。なぜなら read.csv() は Windows の場合、Shift-JIS で、macOS の場合 UTF-8 でファイルを読み込む。一方、read\_csv() は OS と関係なく世界標準である UTF-8 でファイルを読み込むからだ。

正しい都道府県名を表示する方法はいくつかあるが、ここでは 3 つの方法を紹介する。

### 1. read.csv() 関数の fileEncoing 引数の指定

一つ目の方法は、read.csv() 関数の fileEncoding 引数を追加するというものだる。この引数に、指定したファイルのエンコーディングを指定すればよいが、Shift-JIS の場合、"Shift\_JIS" を指定する。ハイフン (-) ではなく、アンダーバー (\_) であることに注意されたい。この"Shift\_JIS"は"cp932" に書き換えるても良い。それではやってみよう。

```
1 ShiftJIS_df2 <- read.csv("Data/Vote_ShiftJIS.csv",
2                               fileEncoding = "Shift_JIS")
3 head(ShiftJIS_df2)
```

```
##   ID   Pref  Zaisei Over65 Under30     LDP     DPJ   Komei   Ishin     JCP     SDP
## 1  1 北海道 0.41903  29.09   24.70  32.82 30.62 13.41   3.43 11.44 1.68
## 2  2 青森県 0.33190  30.14   23.92  40.44 24.61 12.76   3.82  8.92 3.41
## 3  3 岩手県 0.34116  30.38   24.48  34.90 22.44  8.61   5.16 11.24 5.29
## 4  4 宮城県 0.59597  25.75   27.29  36.68 25.40 13.42   3.97  9.99 3.62
## 5  5 秋田県 0.29862  33.84   21.35  43.46 22.72 11.19   5.17  7.56 5.12
## 6  6 山形県 0.34237  30.76   24.75  42.49 21.47 11.78   4.30  7.60 5.20
```

Pref 列の日本語が正常に表示された。むろん、Windows なら fileEncoding 引数がなくても読み込める。むしろ、UTF-8 で書かれたファイルが読み込めない可能性がある。この場合は fileEncoding = "UTF-8" を指定すれば良い。

### 2. read\_csv() 関数の locale 引数の指定

二つ目の方法は `read_csv()` 関数の `locale` 引数を指定する方法である。`read_csv()` には `fileEncoding` 引数がないが、代わりに `locale` があるので、`locale = locale(encoding = "Shift_JIS")` を追加すれば良い。

```
1 ShiftJIS_df3 <- read_csv("Data/Vote_ShiftJIS.csv",
2                               locale = locale(encoding = "Shift_JIS"))
3 head(ShiftJIS_df3)

## # A tibble: 6 x 11
##       ID Pref  Zaisei Over65 Under30    LDP    DPJ Komei Ishin    JCP    SDP
##   <dbl> <chr> <dbl>   <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>   <dbl>
## 1     1 北海道  0.419    29.1    24.7   32.8   30.6  13.4   3.43  11.4   1.68
## 2     2 青森県  0.332    30.1    23.9   40.4   24.6  12.8   3.82  8.92   3.41
## 3     3 岩手県  0.341    30.4    24.5   34.9   22.4   8.61  5.16  11.2   5.29
## 4     4 宮城県  0.596    25.8    27.3   36.7   25.4  13.4   3.97  9.99   3.62
## 5     5 秋田県  0.299    33.8    21.4   43.5   22.7  11.2   5.17  7.56   5.12
## 6     6 山形県  0.342    30.8    24.8   42.5   21.5  11.8   4.3   7.6    5.2
```

### 3. LibreOffice などを利用した方法

第三の方法は、そもそも Shift-JIS ではなく、より一般的な UTF-8 でエンコーディングされたファイルを用意し、それを読み込むことである。ただし、この作業のためには R 以外のソフトが必要である。テキストエディタには文字コードを変更する機能がついているものが多いので、その機能を利用して文字コードを Shift-JIS から UTF-8 に変えれば良い。また、オープンソースのオフィススイートである LibreOffice は、CSV を開く際に文字コードを尋ねてくれるので、Shift-JIS を指定して上で使った csv ファイルを開こう。その後、文字コードを UTF-8 に変更し、別名で csv ファイルを保存すれば、文字コード以外の中身が同じファイルができる。そのようにして作ったのが、`Vote.csv` である。これを読み込んでみよう。

```
1 UTF8_df <- read.csv("Data/Vote.csv") # macOS の場合
2 UTF8_df <- read.csv("Data/Vote.csv", fileEncoding = "UTF-8") # Windows の場合
```

```

1 head(UTF8_df)

##   ID   Pref  Zaisei Over65 Under30     LDP     DPJ Komei Ishin     JCP   SDP
## 1 1 北海道 0.41903 29.09  24.70 32.82 30.62 13.41  3.43 11.44 1.68
## 2 2 青森県 0.33190 30.14  23.92 40.44 24.61 12.76  3.82  8.92 3.41
## 3 3 岩手県 0.34116 30.38  24.48 34.90 22.44  8.61  5.16 11.24 5.29
## 4 4 宮城県 0.59597 25.75  27.29 36.68 25.40 13.42  3.97  9.99 3.62
## 5 5 秋田県 0.29862 33.84  21.35 43.46 22.72 11.19  5.17  7.56 5.12
## 6 6 山形県 0.34237 30.76  24.75 42.49 21.47 11.78  4.30  7.60 5.20

```

第1引数以外の引数を何を指定しなくても、ファイルが正しく読み込まれ、都道府県名が日本語で表示されている。ただし、Windowsで文字化けが生じる場合はファイルのエンコーディングをUTF-8に指定して読み込もう。

### 7.1.3 その他のフォーマット

データ分析で用いられるデータの多くは表の形で保存されている。表形式のデータは、.csv以外に、.xlsx (Excel)、.dta (Stata)、.sav (SPSS)、.ods (LibreOfficeなど)などのファイル形式で保存されることがある。ここでは接する機会が多いExcel形式のファイルとStata形式のファイルの読み込みについて説明しよう<sup>4)</sup>。

Excelファイルを読み込むためにはreadxlパッケージを使う。インストールされていない場合、コンソール上でinstall.packages("readxl")を入力し、インストールする(上でpacman::p\_load(readxl)を実行したのでインストールされているはずだが)。以下ではreadxlパッケージがインストールされていると想定し、Soccer.xlsxファイルを読み込み、Excel\_DFと名付けてみよう。

```

1 Excel_DF <- read_xlsx("Data/Soccer.xlsx", sheet = 1)

```

Excelファイルには2つ以上のシートが含まれる場合が多いので、どのシートを読み込むかをsheetIndexで指定する。実際、Soccer.xlsxファイルをExcelまたはLibreOffice Calcで開いてみると、シートが3つある。そのうち、必要なデータは1つ目

---

<sup>4)</sup> .savファイルはhavenパッケージのread\_sav()で、.odsはreadODSパッケージのread.ods()関数で読みめる。

のシートにあるので、ここでは 1 を指定した。きちんと読み込めたか確認してみよう。

```
1 head(Excel_DF)
```

```
## # A tibble: 6 x 6
##   ID Team          Rank Points Prev_Points Confederation
##   <dbl> <chr>        <dbl>   <dbl>      <dbl> <chr>
## 1 1  Albania        75     1325      1316 UEFA
## 2 2  Algeria        85     1271      1271 CAF
## 3 3  American Samoa 133     1030      1030 OFC
## 4 4  Andorra         155     749       749  UEFA
## 5 5  Angola          121     1117      1117 CAF
## 6 6  Antigua and Barbuda 153     787       787 CONCACAF
```

Stata の.dta ファイルは **haven** パッケージの `read_dta()` 関数を使って読み込む。Stata 形式で保存された Soccer.dta を読み込み、Stata\_DF と名付けてみよう。

```
1 Stata_DF <- read_dta("Data/Soccer.dta")
2 head(Stata_DF)
```

```
## # A tibble: 6 x 6
##   id team          rank points prev_points confederation
##   <dbl> <chr>        <dbl>   <dbl>      <dbl> <chr>
## 1 1  Albania        75     1325      1316 UEFA
## 2 2  Algeria        85     1271      1271 CAF
## 3 3  American Samoa 133     1030      1030 OFC
## 4 4  Andorra         155     749       749  UEFA
## 5 5  Angola          121     1117      1117 CAF
## 6 6  Antigua and Barbuda 153     787       787 CONCACAF
```

Excel 形式のデータと同じ内容のデータであること確認できる（ただし、変数名は少し異なる）。

実際の社会科学の場合、入手するデータの多くは.csv、.xlsx (または.xls)、.dta であるため、以上のやり方で多くのデータの読み込みができる。

### 7.1.4 RData ファイルの場合

データ分析には表形式以外のデータも使われる。データ分析でよく使われるデータの形として、ベクトルや行列のほかに `list` 型とがある。表形式だけでなく、R で扱える様々なデータを含むファイル形式の 1 つが `.RData` フォーマットである。`.RData` には R が扱える形式のデータを格納するだけでなく、表形式のデータを複数格納することができる。また、データだけでなく、分析結果も保存することができる。`.RData` 形式のファイルは R でしか読み込めないため、データの保存方法としては推奨できないが、1 つのファイルにさまざまなデータが格納できるという大きな長所があるため、分析の途中経過を保存するためにしばしば利用される。

ここでは `Data` フォルダにある `Scores.RData` を読み込んでみよう。このファイルには学生 5 人の数学と英語の成績に関するデータがそれぞれ `MathScore` と `EnglishScore` という名で保存されている。このデータを読み込む前に、現在の実行環境にどのようなオブジェクトがあるかを `ls()` 関数を使って確認してみよう<sup>5)</sup>。

```
1  ls()
```

```
## [1] "Excel_DF"      "my_df1"        "ShiftJIS_df1" "ShiftJIS_df2" "ShiftJIS_df3"
## [6] "Stata_DF"       "UTF8_df"
```

現在の実行環境に 7 個のオブジェクトがあることがわかる。

では、`Scores.RData` を読み込んでみよう。`.RData` は、`load()` 関数で読み込む。ただし、これまでのファイルの読み込みとは異なり、保存先のオブジェクト名は指定しない。なぜなら、`.Rdata` の中に既にオブジェクトが保存されているからだ。

```
1  load("Data/Scores.RData")
```

ここでもう一度実行環境上にあるオブジェクトのリストを確認してみよう。

---

5) ちなみに RStudio から確認することもできる。第 4 章のとおりに RStudio を設定した場合、右下ペインの「Environment」タブに現在の実行環境に存在するオブジェクトが表示されているはずだ。

```
1  ls()
```

```
## [1] "EnglishScore" "Excel_DF"      "MathScore"      "my_df1"       "ShiftJIS_df1"
## [6] "ShiftJIS_df2"  "ShiftJIS_df3"  "Stata_DF"      "UTF8_df"
```

MathScore と EnglishScore という名前のオブジェクトが追加されていることが分かる。

このように、`load()` による `.RData` の読み込みは、`.csv` ファイルや `.xlsx` ファイルの読み込みと異なる。以下の 2 点が重要な違いである。

1. `.RData` は、1 つのファイルに複数のデータを含むことができる。
2. `.RData` の中に R のオブジェクトが保存されているので、ファイルの読み込みと同時に名前を付けて格納する必要がない。

問題なく読み込まれているか確認するため、それぞれのオブジェクトの中身を見てみよう。

```
1  MathScore # MathScore の中身を出力
```

```
##   ID   Name Score
## 1  1 Caracal  100
## 2  2 Cheetah  37
## 3  3 Jaguar   55
## 4  4 Leopard  69
## 5  5 Serval   95
```

```
1  EnglishScore # EnglishScore の中身を出力
```

```
##   ID   Name Score
## 1  1 Caracal  90
## 2  2 Cheetah  21
## 3  3 Jaguar   80
## 4  4 Leopard  45
## 5  5 Serval   99
```

## 7.2 データの書き出し

データを手に入れた時点でそのデータ（生データ）が分析に適した状態であることは稀である。多くの場合、分析をするためには手に入れたデータを分析に適した形に整形する必要がある。この作業を「データクリーニング」と呼ぶが、データ分析の作業全体に占めるデータクリーニングの割合は5から7割ほどで、大部分の作業時をクリーニングに費やすことになる。（クリーニングの方法については、データハンドリングの章で説明する。）

データクリーニングが終わったら、生データとクリーニングに使ったコード、クリーニング済みのデータをそれぞれ保存しておこう。クリーニングのコードさえあればいつでも生データからクリーニング済みのデータに変換することができるが、時間的にあまり効率的ではないので、クリーニング済みのデータも保存したほうが良い。そうすれば、いつでもそのデータを読み込んですぐに分析作業に取り組むことができる<sup>6)</sup>。

### 7.2.1 csv ファイル

データを保存する際にまず考えるべきフォーマットは.csvである。データが表の形をしていない場合には次節で紹介する.RData フォーマットが必要になるが、多くの場合、データは表の形をしている。読み込みのところで説明したとおり、.csv ファイルは汎用的なテキストファイルであり、ほとんどの統計ソフトおよび表計算ソフトで読み込むことができるので、特にこだわりがなければ業界標準のフォーマットとも言える csv 形式でデータ保存しておくのが安全だ<sup>7)</sup>。

まずは、架空のデータを作ってみよう。R における表形式のデータは data.frame 型で表現する。（詳細については第9章で説明する。）

<sup>6)</sup> 言うまでもないが、データクリーニングのコードと生データも必ず残しておくべきである。

<sup>7)</sup> Excel (.xlsx) も広く使われているが、Excel は有料の商用ソフトであり、もっていない人もいることを忘れてはいけない。矢内も自分のラップトップには Excel (MS Office) をインストールしていない。

```
1 my_data <- data.frame(  
2     ID     = 1:5,  
3     Name   = c("A さん", "B さん", "C さん", "D さん", "E さん"),  
4     Score  = c(50, 75, 60, 93, 51)  
5 )
```

上のコードを実行すると、`my_data` というオブジェクトが生成され、中には以下のようなデータが保持される。

```
1 my_data  
  
##   ID  Name Score  
## 1  1  A さん   50  
## 2  2  B さん   75  
## 3  3  C さん   60  
## 4  4  D さん   93  
## 5  5  E さん   51
```

このデータを `my_data.csv` という名前の csv ファイルで保存するには、`write.csv()` という関数を使う。必須の引数は 2 つで、1 つ目の引数は保存するオブジェクト名、2 つ目の引数 `file` は書き出すファイル名。もし、プロジェクトフォルダの下位フォルダ、たとえば、Data フォルダーに保存するなら、ファイル名を "Data/my\_data.csv" のように指定する。他によく使う引数として `row.names` があり、デフォルトは `TRUE` だが、`FALSE` にすることを推奨する。`TRUE` のままだと、データの 1 列目に行番号が保存される。

```
1 write.csv(my_data, file = "Data/my_data.csv", row.names = FALSE)
```

これを実行すると、プロジェクトのフォルダの中にある Data フォルダに `my_data.csv` が生成される。LibreOffice や Numbers、Excel などを使って `my_data.csv` を開いてみると、先ほど作成したデータが保存されていることが確認できる。

	A	B	C	D
1	ID	Name	Score	
2	1	Aさん	50	
3	2	Bさん	75	
4	3	Cさん	60	
5	4	Dさん	93	
6	5	Eさん	51	
7				

図 7.1: 保存した csv ファイルの中身

## 7.2.2 RData ファイル

最後に.RData 形式でデータを書き出してみよう。今回は先ほど作成した `my_data` と、以下で作成する `numeric_vec1`、`numeric_vec2`、`character_vec` を `my_RData.RData` という名のファイルとして `Data` フォルダに書き出す。使用する関数は `save()` である。引数として保存するオブジェクト名をすべてカンマ区切りで書き、最後に `file` 引数でファイル名を指定すればよい。

```

1 numeric_vec1 <- c(1, 5, 3, 6, 99, 2, 8)
2 numeric_vec2 <- 1:20
3 character_vec <- c('cat', 'cheetah', 'lion', 'tiger')
4 save(my_data, numeric_vec1, numeric_vec2, character_vec,
5     file = "Data/my_RData.RData")

```

実際に `my_RData.Rdata` ファイルが生成されているかを確認してみよう。ファイルの保存がうまくいっていれば、`my_RData.Rdata` を読み込むだけで、`my_data`、`numeric_vec1`、`numeric_vec2`、`character_vec` という 4 つのオブジェクトを一挙に作業スペースに読み込むことができるはずだ。実際にできるか確認しよう。そのためには、まず現在の実行環境上にある `my_data`、`numeric_vec1`、`numeric_vec2`、`character_vec` を削除する。そのため `rm()` 関数を使う。

```

1 rm(my_data)
2 rm(numeric_vec1)

```

```
3 rm(numeric_vec2)  
4 rm(character_vec)
```

このコードは以下のように1行のコードに書き換えられる。

```
1 rm(list = c("my_data", "numeric_vec1", "numeric_vec2", "character_vec"))
```

4のオブジェクトが削除されたか、ls()関数で確認しよう。

```
1 ls()
```

```
## [1] "EnglishScore" "Excel_DF"      "MathScore"      "my_df1"       "ShiftJIS_df1"  
## [6] "ShiftJIS_df2" "ShiftJIS_df3" "Stata_DF"      "UTF8_df"
```

それではDataフォルダー内のmy\_RData.RDataを読み込み、4つのオブジェクトが実行環境に読み込まれるか確認しよう。

```
1 load("Data/my_RData.RData") # Dataフォルダー内のmy_RData.RDataを読み込む  
2 ls()                      # 作業スペース上のオブジェクトのリストを確認する
```

```
## [1] "character_vec" "EnglishScore" "Excel_DF"      "MathScore"  
## [5] "my_data"        "my_df1"      "numeric_vec1" "numeric_vec2"  
## [9] "ShiftJIS_df1"  "ShiftJIS_df2" "ShiftJIS_df3" "Stata_DF"  
## [13] "UTF8_df"
```



## 第 8 章

# データの型

### 8.1 データ型とは

ここでは R におけるデータ型について説明します。第 9 章で説明するデータ「構造」とデータ「型」は異なる概念です。次章でも説明しますが、R におけるデータの最小単位はベクトルです。`c(1, 2, 3, 4, 5)` や `c("Yanai", "Song", "Hadley")` もベクトルですが、`30` とか "`R`" もベクトルです。後者のように要素が 1 つのベクトルは原子ベクトル (atomic vector) とも呼ばれますが、本質的には普通ベクトルです。このベクトルの要素の性質がデータ型です。たとえば `c(2, 3, 5, 7, 11)` は数値型ですし、`c("R", "Python", "Julia")` は文字型です。他にも第 6 章で紹介した `FALSE` や `TRUE` は論理型と呼ばれています。一つのベクトルは複数の要素で構成されることも可能ですが、必ず同じデータ型である必要があります。

しかし、データ分析を行う際はベクトル以外のデータも多いです。行列や表がその典型例です。しかし、行列でも表でも中身の一つ一つの要素は長さ 1 のベクトルに過ぎません。たとえば、以下のような 2 行 5 列のベクトルがあるとします。ここで 1 行 3 列目の要素は 11 であり、長さ 1 の数値型ベクトルです。あるいは 5 列目は `c(23, 29)` であり、長さ 2 の数値型ベクトルです。

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]     2     5    11    17   23
## [2,]     3     7   13    19   29
```

表についても考えてみましょう。以下の表の3行2列目の要素は"American Samoa"という長さ1の文字型ベクトルです。また、6列目はc("UEFA", "CAF", "OFC", "UEFA", "CAF", "CONCACAF")であり、これは長さ6の文字型ベクトルです。このようにRで扱う全てのデータは複数のベクトルが集まっているものです。

```
##   ID          Team Rank Points Prev_Points Confederation
## 1  1      Albania    75    1325      1316        UEFA
## 2  2      Algeria    85    1271      1271        CAF
## 3  3 American Samoa 133    1030      1030        OFC
## 4  4      Andorra   155     749      749        UEFA
## 5  5      Angola    121    1117      1117        CAF
## 6  6 Antigua and Barbuda 153     787      787    CONCACAF
```

つまり、複数のベクトルを綺麗に、または分析しやすく集めたのが行列や表であり、これがデータ構造に該当します。データ型ごとの処理方法については本書を通じて紹介して行きますので、本章は軽く読んで頂いても構いません。ただし、Factor型とDate & Datetime型の処理はやや特殊ですので、手を動かしながら読み進めることをおすすめします。

ここではデータ型について紹介し、次章ではデータ構造について解説します。

## 8.2 Logical

Logical型はTRUEとFALSEのみで構成されたデータ型です。練習としてなにかの長さ5の論理型ベクトルを作ってみましょう。

```
1 logical_vec1 <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
2
3 logical_vec1
```

```
## [1] TRUE FALSE TRUE TRUE FALSE
```

注意すべきこととしては、TRUEとFALSEは"で囲まないことです。"TRUE"、"FALSE"と

入力してしまえば logical 型として認識されません。もし、一つ間違えて 2 番目の要素である FALSE を"FALSE"と入力したらどうなるでしょうか。

```
1 logical_vec2 <- c(TRUE, "FALSE", TRUE, TRUE, FALSE)
2
3 logical_vec2
```

```
## [1] "TRUE"  "FALSE" "TRUE"  "TRUE"  "FALSE"
```

2 番目の要素だけでなく、他の全ての要素も"で囲まれるようになりました。実際に、この 2 つのベクトルのデータ型を確認してみましょう。ベクトルのデータ型を確認する関数は `class()` 関数です。

```
1 class(logical_vec1)
```

```
## [1] "logical"
```

```
1 class(logical_vec2)
```

```
## [1] "character"
```

`logical_vec1` は logical 型ですが、`logical_vec2` は character 型と認識されます。

他にも、`is.logical()` 関数を使ってあるベクトルが logical 型か否かを判定することも可能です。もし、ベクトルが logical 型なら `TRUE` が、logical 型以外なら `FALSE` が返って来ます。

```
1 is.logical(logical_vec1)
```

```
## [1] TRUE
```

```
1 is.logical(logical_vec2)
```

```
## [1] FALSE
```

Logical 型は様々な場面で使われますが、代表的な使い方は第 6.3 章で紹介しました要素の抽出と第 10 章で紹介する予定の条件分岐 (`if()` や `ifelse()`)、条件反復 (`while()`) があります。

### 8.3 Numeric

Numeric 型は数値型ですが、まずは numeric 型のベクトル numeric\_vec1 を作成し、データ型を確認してみましょう。

```
1 numeric_vec1 <- c(2, 0, 0, 1, 3)
2
3 class(numeric_vec1)
```

```
## [1] "numeric"
```

is.logical() に似た関数 is.numeric() も使用可能です。

```
1 is.numeric(numeric_vec1)
2
3 ## [1] TRUE
```

```
1 is.numeric(logical_vec1)
2
3 ## [1] FALSE
```

もうちょっと詳しく分けると integer 型と double 型があります。以下の内容はあまり意識的に区分して使う場面が稀ですので、(読み) 飛ばしても構いません。

integer は整数型であり、double は実数型です。これは class() 関数では確認できず、typeof() 関数を使います。

```
1 typeof(numeric_vec1)
2
3 ## [1] "double"
```

一般的に作成する numeric 型のベクトルは全て double 型です。もし、整数型のベクトルを作成したい場合、数値の後ろに L を付けます。

```
1 integer_vec1 <- c(2L, 0L, 0L, 1L, 3L)
2 typeof(integer_vec1)
```

```
## [1] "integer"
```

ここでも注意すべき点としては、一つでも L が付かない要素が含まれる場合、自動的に double 型に変換されるという点です。

```
1 integer_vec2 <- c(2L, 0L, 0, 1L, 3L)
2 typeof(integer_vec2)
```

```
## [1] "double"
```

もちろんですが、小数点のある数値に L を付けても integer 型にはならず、勝手に double 型になります。また、integer 型同士の割り算の結果も double 型になります。これは  $2L/1L$  のような場合でも同じです。足し算、引き算、掛け算は integer 型になります。

```
1 typeof(2.3L)
```

```
## [1] "double"
```

```
1 typeof(3L / 12L)
```

```
## [1] "double"
```

```
1 typeof(3L / 1L)
```

```
## [1] "double"
```

```
1 typeof(3L + 1L)
```

```
## [1] "integer"
```

```
1 typeof(3L - 4L)
```

```
## [1] "integer"
```

```
1 typeof(3L * 6L)
```

```
## [1] "integer"
```

一般的な分析において整数と実数を厳格に区別して使う場面は多くないと考えられますので、今のところはあまり気にしなくとも問題ないでしょう。

## 8.4 Complex

Complex 型は複素数を表すデータ型であり、実数部 + 虚数部  $i$  のように表記します。まず、複素数のベクトル `complex_vec1` を作成し、データ型を確認してみましょう。

```
1 complex_vec1 <- c(1+3i, 3+2i, 2.5+7i)
2 complex_vec1

## [1] 1.0+3i 3.0+2i 2.5+7i

1 class(complex_vec1)

## [1] "complex"
```

あまりおすすめはできませんが、虚数部  $i$  + 実数部のような書き方も可能です。

```
1 complex_vec2 <- c(3i+1, 2i+3, 7i+2.5)
2 complex_vec2

## [1] 1.0+3i 3.0+2i 2.5+7i

1 class(complex_vec2)

## [1] "complex"
```

`complex_vec1` と `complex_vec2` は同じベクトルであることを確認してみましょう。

```
1 complex_vec1 == complex_vec2

## [1] TRUE TRUE TRUE
```

もし、ベクトル内に numeric 型と complex 型が混在している場合、強制的に complex 型に変換されます。変換された後の値は実数部 + $0i$  のようになります。

```
1 complex_vec3 <- c(2+7i, 5, 13+1i)
2 complex_vec3
```

```
## [1] 2+7i 5+0i 13+1i
1 class(complex_vec3)

## [1] "complex"
```

---

## 8.5 Character

Character 型は文字列で構成されているデータ型です。R を含む多くの言語は文字列を表現するために、中身を"で囲みます。"abc"は character 型ですが、"1"や"3+5i"も character 型です。数字であっても"で囲んだらそれは文字列となります。それではいくつかの character 型ベクトルを作っていみましょう。

```
1 char_vec1 <- c("Yanai", "Song", "Shigemura", "Tani")
2 char_vec2 <- c(1, 2, 3, 4)
3 char_vec3 <- c("1", "2", "3", "4")
4
5 char_vec1

## [1] "Yanai"      "Song"       "Shigemura"  "Tani"
1 char_vec2
```

```
## [1] 1 2 3 4
1 char_vec3
```

```
## [1] "1" "2" "3" "4"
```

char\_vec2 と char\_vec3 の違いは通じを"で囲んだか否かです。ベクトルの中身を見ても、char\_vec2 は"で囲まれていません。データ型を見てみましょう。

```
1 class(char_vec1)

## [1] "character"
```

```

1  class(char_vec2)

## [1] "numeric"

1  class(char_vec3)

## [1] "character"

```

やはり `char_vec3` も `character` 型になっていることが分かります。

---

## 8.6 Factor

`Factor` 型はラベル付きの数値型データです。`Factor` 型の見た目は `character` 型とほぼ同じですし、分析の場面においても `character` 型とほぼ同じ扱いになります。`Factor` 型と `character` 型との違いは、「順序が付いている」点です。例えば、以下の質問文に対するアンケートの結果を考えてみましょう。

- あなたは猫が好きですか。
  1. めちゃめちゃ好き
  2. めちゃ好き
  3. 好き
  4. どちらかといえば好き

以下の表 8.1 は 5 人の結果です。

表 8.1: 猫好きの度合い

ID	Name	Cat
1	Yanai	めちゃめちゃ好き
2	Song	めちゃめちゃ好き
3	Shigemura	どちらかといえば好き
4	Tani	めちゃ好き
5	Hadley	好き

人間としてはこの表から、重村という人がどれだけ猫が嫌いなのかが分かります。ただし、R はそうではありません。R は日本語どころか、人間の言葉は理解できません。各項目ごとに順番を付けてあげる必要がありますが、そのために使われるのが factor 型です。

実習のために表 8.1 の Cat 列のみのベクトルを作つてみましょう。

```
1 factor_vec1 <- c("めちゃめちゃ好き", "めちゃめちゃ好き",
2                           "どちらかといえば好き", "めちゃ好き", "好き")
3
4 factor_vec1
5
6 ## [1] "めちゃめちゃ好き"      "めちゃめちゃ好き"      "どちらかといえば好き"
7 ## [4] "めちゃ好き"           "好き"
8
9 class(factor_vec1)
10
11 ## [1] "character"
```

factor\_vec1 は普通の文字列ベクトルであることが分かります。これを factor 型に変換するためには factor() 関数を使います。

```
1 factor_vec2 <- factor(factor_vec1, ordered = TRUE,
2                           levels = c("どちらかといえば好き", "好き",
3                                     "めちゃ好き", "めちゃめちゃ好き"))
4
5 class(factor_vec2)
6
7 ## [1] "ordered" "factor"
```

データ型が factor 型に変換されています。"ordered" というものも付いていますが、これについては後ほど説明します。それでは中身をみましょう。

```
1 factor_vec2
2
3 ## [1] めちゃめちゃ好き      めちゃめちゃ好き      どちらかといえば好き
4 ## [4] めちゃ好き           好き
5
6 ## Levels: どちらかといえば好き < 好き < めちゃ好き < めちゃめちゃ好き
```

いくつかの点で異なります。まず、文字列であるにもかかわらず、"で囲まれていない

点です。そして3行目に4 Levels: というのが追加されている点です。このlevelは「水準」と呼ばれるものです。4 Levels ですから、factor\_vec2は4つの水準で構成されていることを意味します。Factor型の値は予め指定された水準以外の値を取ることはできません。たとえば、2番目の要素を「超好き」に変えてみましょう。

```

1 factor_vec2[2] <- "超好き"

## Warning in `<- .factor`(`*tmp*`, 2, value = "超好き"): invalid factor level, NA
## generated

1 factor_vec2

## [1] めちゃめちゃ好き      <NA>          どちらかといえば好き
## [4] めちゃ好き          好き
## Levels: どちらかといえば好き < 好き < めちゃ好き < めちゃめちゃ好き

```

警告が表示され、2番目の要素が後ほど紹介する欠損値となっていることが分かります。それでは普通に「好き」を入れてみましょう。

```

1 factor_vec2[2] <- "好き"

2 factor_vec2

## [1] めちゃめちゃ好き      好き          どちらかといえば好き
## [4] めちゃ好き          好き
## Levels: どちらかといえば好き < 好き < めちゃ好き < めちゃめちゃ好き

```

今回は問題なく置換できましたね。このようにfactor型の取りうる値は既に指定されています。また、## 4 Levels: どちらかといえば好き < 好き < ... < めちゃめちゃ好きからも分かるように、その大小関係の情報も含まれています。猫好きの度合いは「どちらかといえば好き-好き-めちゃ好き-めちゃめちゃ好き」の順で高くなることをRも認識できるようになりました。

Factor型はこのように順序付きデータを扱う際に便利なデータ型ですが、順序情報を含まないfactor型もあります。これはfactor()を使う際、ordered = TRUE引数を削除するだけできます。

```

1  factor_vec3 <- factor(factor_vec1,
2                           levels = c("どちらかといえば好き", "好き",
3                                     "めちゃ好き", "めちゃめちゃ好き"))
4  factor_vec3

## [1] めちゃめちゃ好き      めちゃめちゃ好き      どちらかといえば好き
## [4] めちゃ好き            好き
## Levels: どちらかといえば好き 好き めちゃ好き めちゃめちゃ好き

1  class(factor_vec3)

## [1] "factor"

```

今回は 3 行目が## Levels: どちらかといえば好き 好き めちゃ好き めちゃめちゃ好きとなり、順序に関する情報がなくなりました。また、class() で確認しましたデータ型に"ordered"が付いていません。これは順序なし factor 型であることを意味します。「順序付けしないなら factor 型は要らないのでは...?」と思うかも知れませんが、これはこれで便利です。その例を考えてみましょう。

分析において factor 型は character 型に近い役割を果たしますが、factor 型なりの長所もあります。それは図や表を作成する際です。例えば、横軸が都道府県名で、縦軸がその都道府県の財政力指数を表す棒グラフを作成するとします。たとえば、表 8.2 のようなデータがあるとします。このデータは 3 つの列で構成されており、ID と Zaisei 列は numeric 型、Pref 列は character 型です。

表 8.2: 5 都道府県の H29 財政力指数

ID	都道府県	財政力指数
1	Hokkaido	0.44396
2	Tokyo	1.19157
3	Aichi	0.92840
4	Osaka	0.78683
5	Fukuoka	0.64322

可視化については第17章以降で詳しく解説しますが、このPref列をcharacter型にしたままグラフにしますと図8.1のようになります。

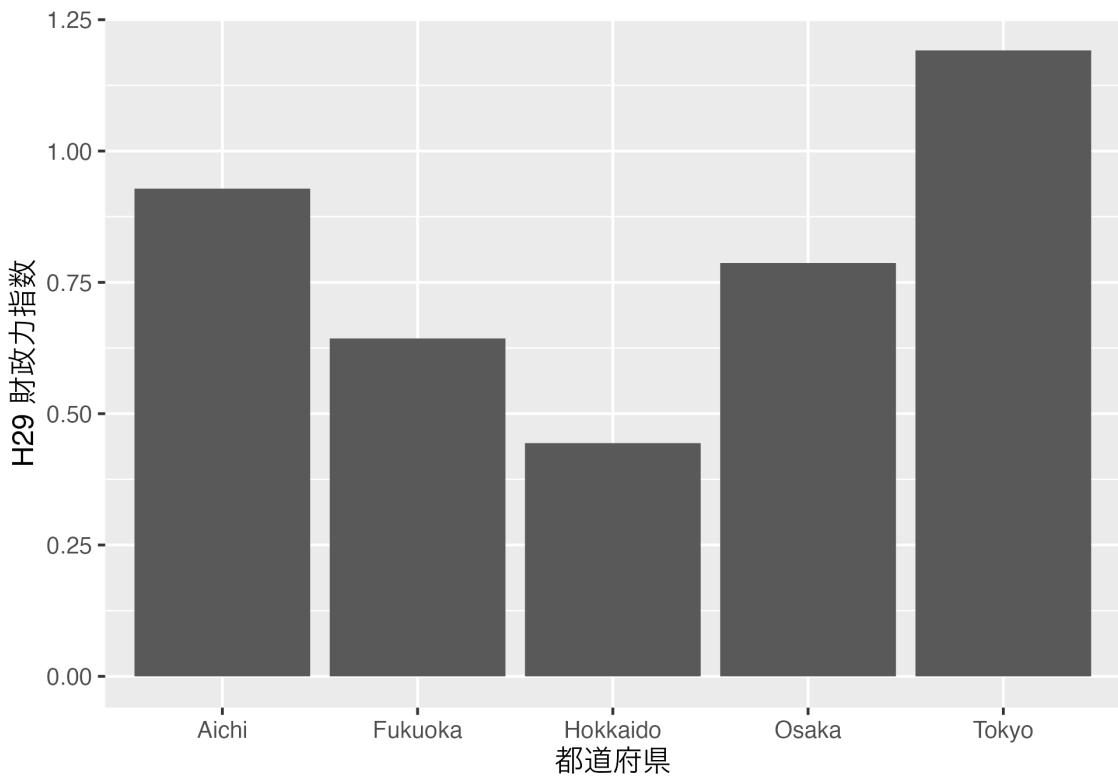


図8.1: 5都道府県のH29財政力指数

このようにアルファベット順で横軸が並び替えられます。別にこれでも問題ないと思う方もいるかも知れませんが、基本的に日本の都道府県は北から南の方へ並べるのが一般的な作法です<sup>1)</sup>。北海道と東京、大阪の間には順序関係はありません。しかし、表示される順番は固定したい。この場合、Pref列を順序なしfactor型にすれば良いです<sup>2)</sup>。データフレームの列を修正する方法は第9章で詳しく説明します。

```

1 zaisei_df$Pref <- factor(zaisei_df$Pref,
2                               levels = c("Hokkaido", "Tokyo", "Aichi", "Osaka", "Fukuok

```

1) アメリカの州ならアルファベット順ですね。

2) むろん、「北から南へ」という規則もあるので、順序付きfactor型にしても問題ありません。ただし、今回はあくまでも表示順番を設定したいだけですので、ordered = TRUEは省略しました。

`zaisei_df` の `Pref` 列を `factor` 型にしてから同じ図を描くと図 8.2 のようになります。

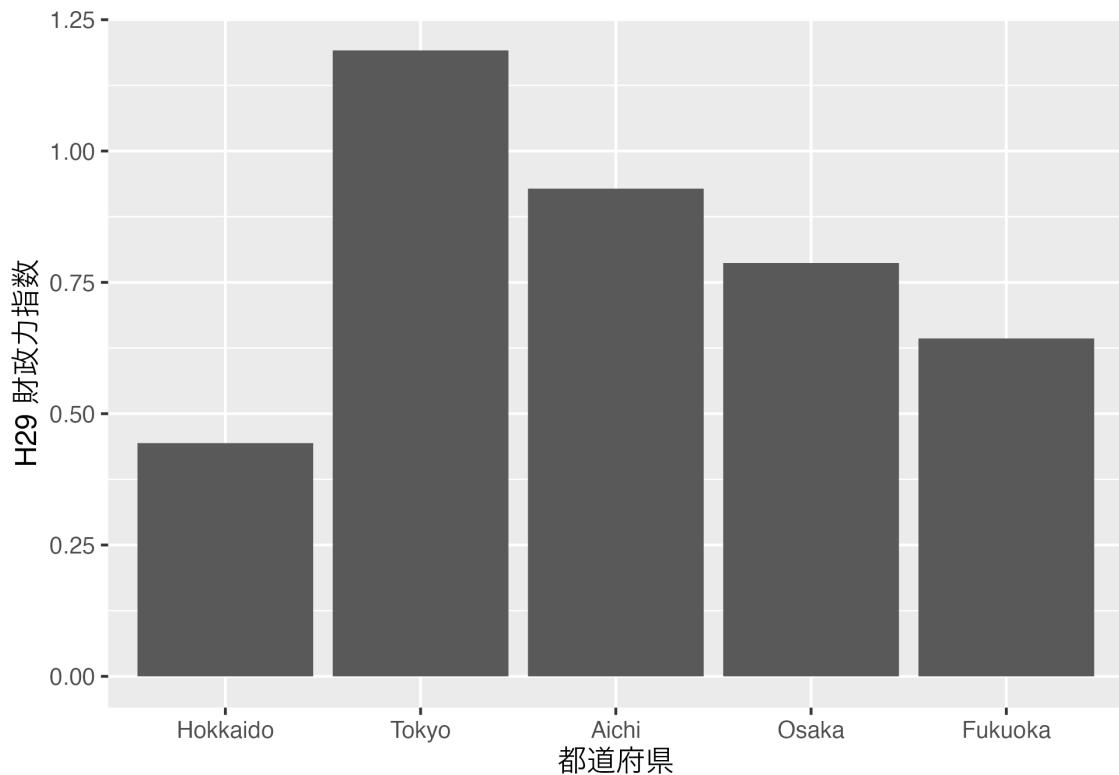


図 8.2: 5 都道府県の H29 財政力指数

都道府県以外にもこのような例は多くあります。順序尺度で測定された変数が代表的な例です。他にも政党名を議席数順で表示させたい場合も `factor` 型は有効でしょう。

## 8.7 Date

### 8.7.1 なぜ Date 型があるのか

Date 型は年月日を表すデータ型<sup>3)</sup>です。この 2 つのデータ型はかなり複雑ですが、ここでは簡単に説明します。Date 型は日付の情報を含むため、順序関係が成立します。その

---

<sup>3)</sup> 他にも時間まで表す `DateTime` 型があります。

意味では順序付き Factor 型とあまり挙動は変わらないかもしれません、実際はそうではありません。

たとえば、Song の 1 週間<sup>4)</sup>の睡眠時間を記録したデータ SongSleep があるとします。Date という列には日付が、Sleep 列には睡眠時間が記録されています。睡眠時間の単位は「分」です。

```

1 SongSleep <- data.frame(
2   Date  = c("2017-06-17", "2017-06-18", "2017-06-19", "2017-06-20",
3   "2017-06-21", "2017-06-22", "2017-06-23"),
4   Sleep = c(173, 192, 314, 259, 210, 214, 290)
5 )

```

中身をみると、以下のようになります。

```

1 SongSleep

##           Date  Sleep
## 1 2017-06-17    173
## 2 2017-06-18    192
## 3 2017-06-19    314
## 4 2017-06-20    259
## 5 2017-06-21    210
## 6 2017-06-22    214
## 7 2017-06-23    290

```

日付を横軸に、睡眠時間を縦軸にした散布図を描くと図@ref{fig:datatype-date-3}のようになります。{ggplot2}を利用した作図については第@ref{visualization}章で解説しますので、ここでは Date 型の特徴のみ理解してもらえたたら十分です。

```

1 ggplot(SongSleep,
2   mapping = aes(x = Date, y = Sleep)) +
3   geom_point() +
4   labs(x = "日付", y = "睡眠時間 (分)") +

```

---

<sup>4)</sup> 実際は 2017 年 6 月 17 日から 11 月 15 日まで記録しましたが、ここでは 1 週間分のみお見せします。

```
5     theme_gray(base_size = 12)
```

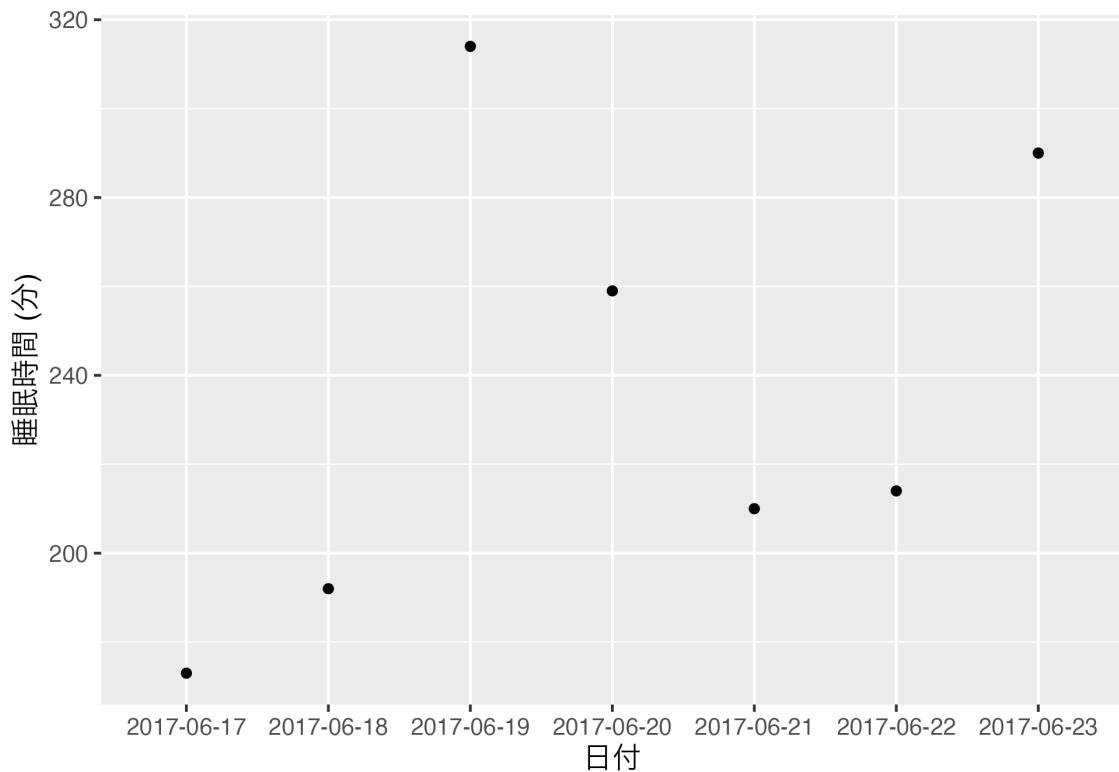


図 8.3: Song の睡眠時間

この図は全く問題ないように見えます。それでは、Date 列をそれぞれ Date 型に変換し、SongSleep データの DateD としてみます。データフレームの列追加については第 9.4 章で解説します。

```
1 SongSleep$DateD <- as.Date(SongSleep$date)
```

中身を見てみますが、あまり変わっていないようです。Date と DateD 列は全く同じように見えますね。

```
1 SongSleep
```

```
##           Date Sleep      DateD
## 1 2017-06-17    173 2017-06-17
```

```
## 2 2017-06-18 192 2017-06-18
## 3 2017-06-19 314 2017-06-19
## 4 2017-06-20 259 2017-06-20
## 5 2017-06-21 210 2017-06-21
## 6 2017-06-22 214 2017-06-22
## 7 2017-06-23 290 2017-06-23
```

図にすると実は先ほどの図と同じものが得られます。

```
1 ggplot(SongSleep,
2   mapping = aes(x = DateD, y = Sleep)) +
3   geom_point() +
4   labs(x = "日付", y = "睡眠時間 (分)") +
5   theme_gray(base_size = 12)
```

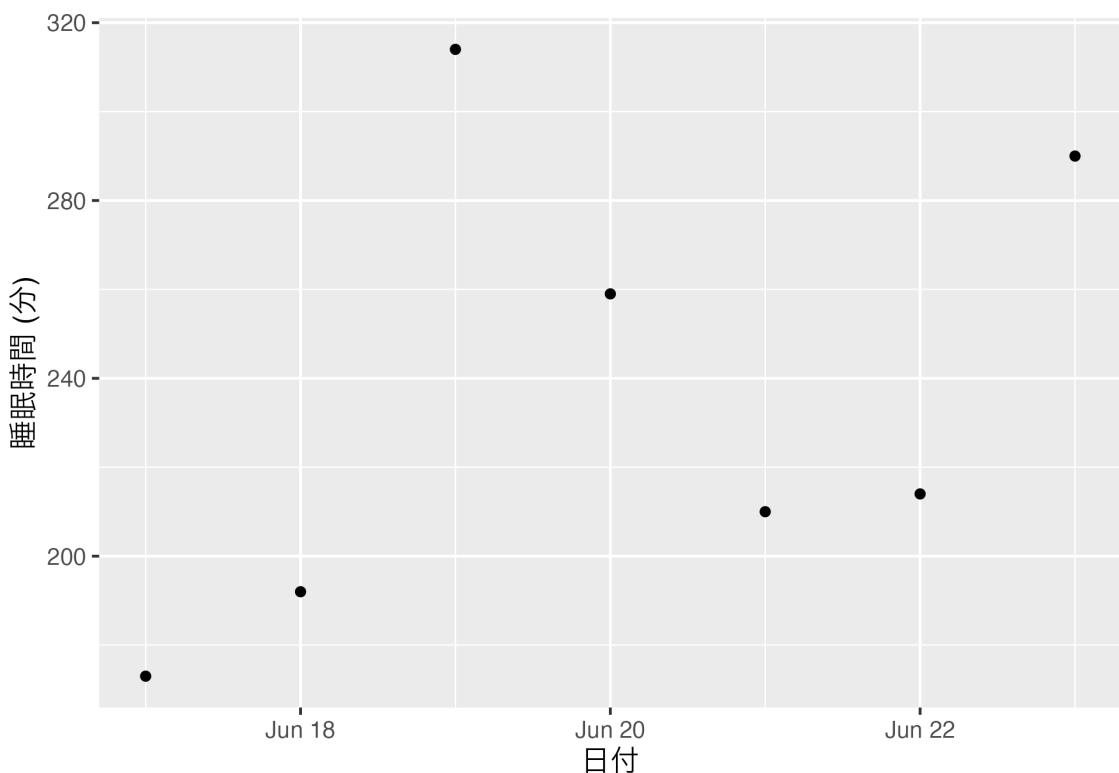


図 8.4: Song の睡眠時間

しかし、Song がうっかり 6 月 19 日に記録するのを忘れたとします。つまり、SongSleep データの 3 行目が抜けている状況を考えてみましょう。データフレームの要素抽出については第 9.4 章で解説します。

```
1 SongSleep2 <- SongSleep[-3, ]
```

中身をみると、以下のようになります。Date も DateD も同じように見えます。

```
1 SongSleep2
```

```
##          Date  Sleep      DateD
## 1 2017-06-17    173 2017-06-17
## 2 2017-06-18    192 2017-06-18
## 4 2017-06-20    259 2017-06-20
## 5 2017-06-21    210 2017-06-21
## 6 2017-06-22    214 2017-06-22
## 7 2017-06-23    290 2017-06-23
```

この状態で横軸を Date にしたらどうなるでしょうか（図 8.5）。

```
1 ggplot(SongSleep2,
2   mapping = aes(x = Date, y = Sleep)) +
3   geom_point() +
4   labs(x = "日付", y = "睡眠時間 (分)") +
5   theme_gray(base_size = 12)
```

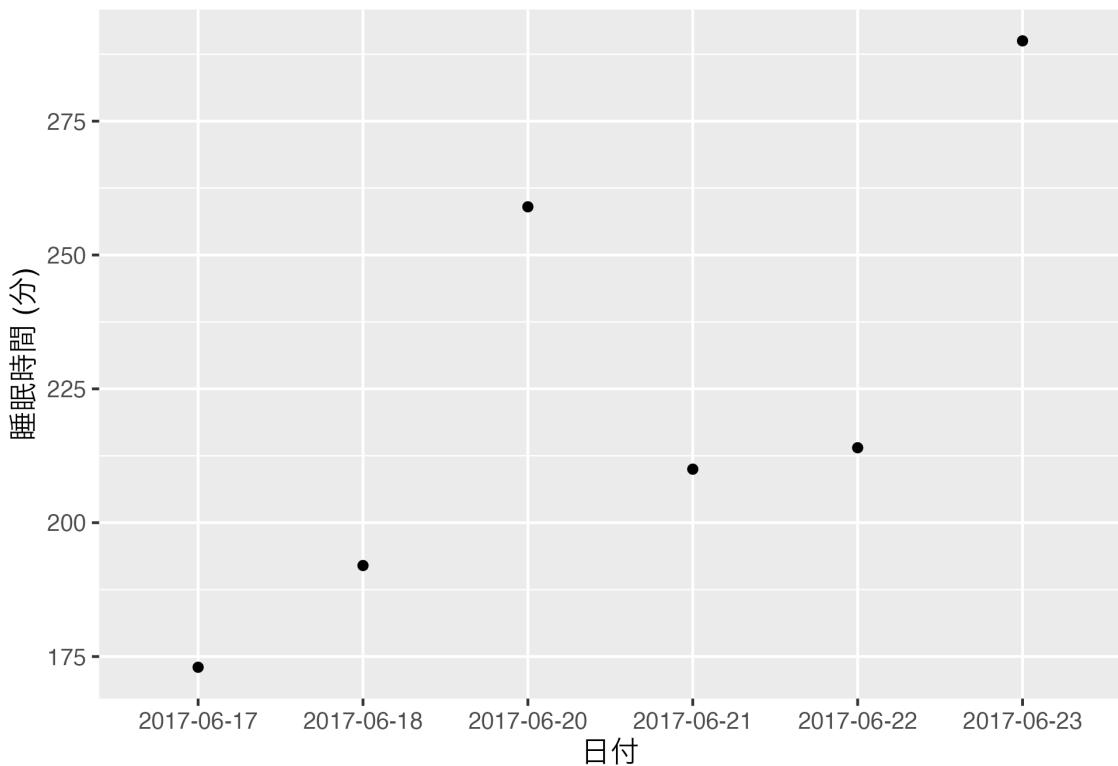


図 8.5: Song の睡眠時間

一方、横軸を DateD にしたものが図 8.6 です。

```
1 ggplot(SongSleep2,
2     mapping = aes(x = DateD, y = Sleep)) +
3     geom_point() +
4     labs(x = "日付", y = "睡眠時間 (分)") +
5     theme_gray(base_size = 12)
```

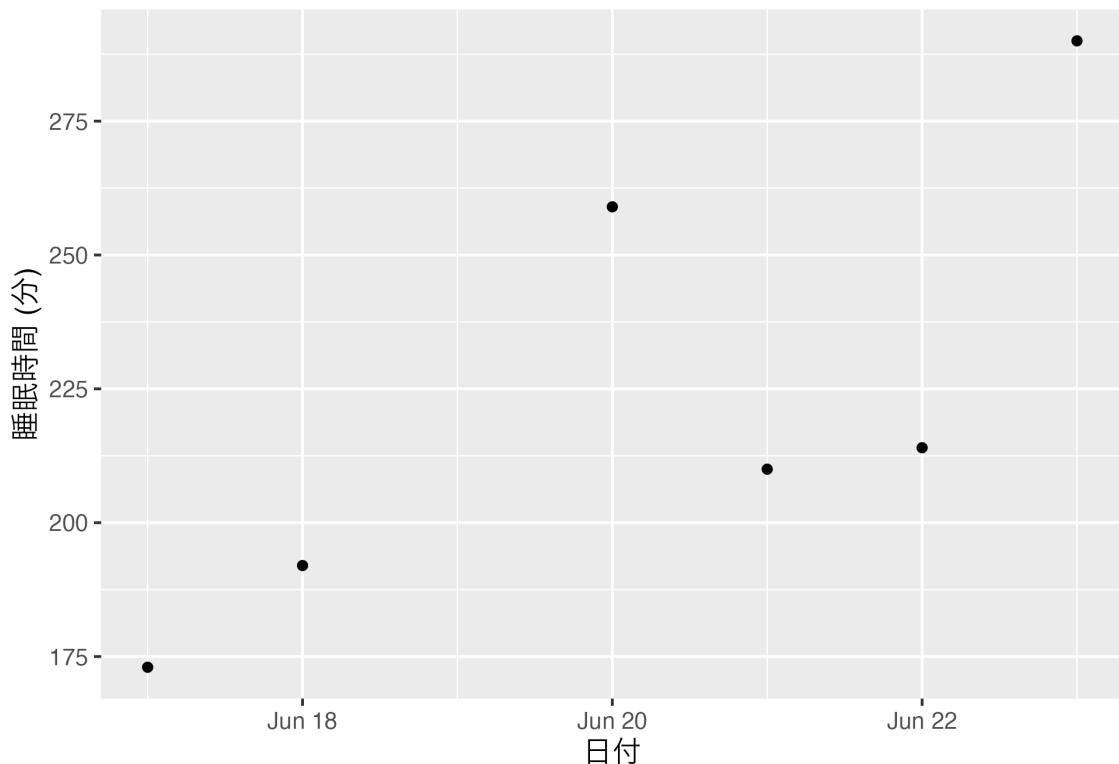


図 8.6: Song の睡眠時間

違いが分かりますかね。違いは抜けている 6 月 19 日です。図 8.5 を見ると、横軸の 6 月 18 日の次が 20 日になっています。一方、図 8.6 は 19 日になっており、ちゃんと空けてくれますね。これは Date 型でない場合、データにないものは図に表示されないことを意味します。一方、Date 型は抜けている日があっても、図に表示表示されます。一般的 character 型または factor 型でこのようなことを再現するためには、6 月 19 日の列を追加し、睡眠時間を欠損値として指定する必要があります。たとえば、SongSleep データにおいて 6 月 19 日の行は温存したまま、睡眠時間だけを欠損値にしてみましょう。

```

1 SongSleep3 <- SongSleep
2 SongSleep3$Sleep [SongSleep3$Date == "2017-06-19"] <- NA
3
4 SongSleep3

```

```

##           Date Sleep      DateD
## 1 2017-06-17 173 2017-06-17

```

```
## 2 2017-06-18 192 2017-06-18
## 3 2017-06-19 NA 2017-06-19
## 4 2017-06-20 259 2017-06-20
## 5 2017-06-21 210 2017-06-21
## 6 2017-06-22 214 2017-06-22
## 7 2017-06-23 290 2017-06-23
```

このように日付はあるが、睡眠時間が欠損している場合、図にしたもののが図 8.7 です。

```
1 ggplot(SongSleep3,
2   mapping = aes(x = Date, y = Sleep)) +
3   geom_point() +
4   labs(x = "日付", y = "睡眠時間 (分)") +
5   theme_bw()
```

## Warning: Removed 1 rows containing missing values (geom\_point).

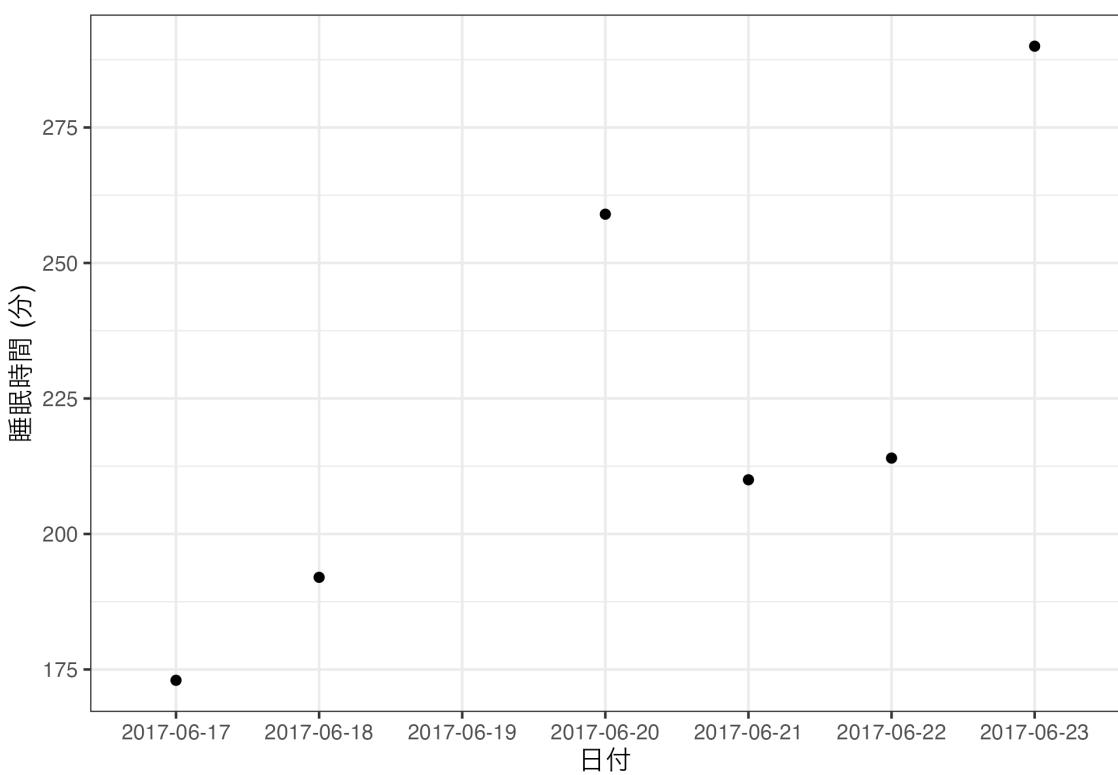


図 8.7: Song の睡眠時間

横軸上に 6 月 19 日が表示されます。このように Date 型でなくとも Date 型と同じように動かすことは可能ですが、非常に面倒です。その意味で Date 型は時系列データを扱う際に非常に便利なデータ型です。

### 8.7.2 Date 型の作り方

Date 型を作成方法はいくつかあります。

1. character 型を Date 型にする
2. numeric 型を Date 型にする

主に使う方法は 1 であり、既に前節でお見せしました `as.Date()` 関数を使います。方法 2 もまた `as.Date()` を使いますが、これは「xxxx 年 xx 月 xx 日から何日目」という書き方となり、起点となる日付 (`origin`)<sup>5)</sup>を指定する必要があります。

ここでは方法 1 について解説します。日付を表すいくつかのベクトルを作ってみましょう。

```
1 Date1 <- "2020-05-21"  
2 Date2 <- "2020-5-21"  
3 Date3 <- "2020/5/21"  
4 Date4 <- "20/05/21"  
5 Date5 <- "20200521"  
6 Date6 <- "2020 05 21"  
7 Date7 <- "2020.05.21"  
  
1 as.Date(Date1)
```

```
## [1] "2020-05-21"  
  
1 as.Date(Date2)
```

```
## [1] "2020-05-21"
```

---

<sup>5)</sup> 主に使う `origin` は 1970 年 1 月 1 日です。

```
1 as.Date(Date3)

## [1] "2020-05-21"

1 as.Date(Date4, "%y/%m/%d")

## [1] "2020-05-21"

1 as.Date(Date5, "%Y%m%d")

## [1] "2020-05-21"

1 as.Date(Date6, "%Y %m %d")

## [1] "2020-05-21"

1 as.Date(Date7, "%Y.%m.%d")

## [1] "2020-05-21"
```

`Date1`、`Date2`、`Date3`のようなベクトルの場合、`as.Date()`のみで Date 型に変換できます。つまり、日付が数字のみで構成され、年が4桁となっており、年月日が-または/で区切られている場合はこれでだけで十分です。しかし、年が2桁になっていたり、その他の記号が使われたり、区切られていない場合は、`format =`引数を指定する必要があります。たとえば `Date4` は年が2桁となっています。2桁の年は `%y` と表記します。この表記法の一部を以下の表で紹介します。

表記	説明	例
%y	年 (2桁)	20
%Y	年 (4桁)	2020
%m	月 (数字)	5, 10
%b	月 (文字)	Jan
%B	月 (文字)	January
%d	日	5, 05, 13

他にも様々な表記法がありますが、詳細は `?strptime` で確認してみてください。

他にも、日本では使わない表記法ですが、月を英語で表記したり、日月年の順で表記する場合があります。後者は `format =` 引数の順番を変えるだけで問題有りませんが、問題は前者です。そこで使うのが `%b` または `%B` です。`%b` は 3 文字の月表記で、`%B` はフルネームです。

```
1 as.Date("21may2020", format = "%d/%b/%Y")
```

```
## [1] "2020-05-21"
```

```
1 as.Date("May/21/2020", format = "%b/%d/%Y")
```

```
## [1] "2020-05-21"
```

うまくいかないですね。これはシステムの時間ロケールが日本になっているのが原因です。ロケール設定は `Sys.getlocale()` で確認できます。

```
1 Sys.getlocale(category = "LC_TIME")
```

```
## [1] "en_US.UTF-8"
```

これを `Sys.setlocale()` を使って、"C"に変更します。

```
1 Sys.setlocale(category = "LC_TIME", locale = "C")
```

```
## [1] "C"
```

それではもう一回やってみましょう。

```
1 as.Date("21may2020", format = "%d/%b/%Y")
```

```
## [1] "2020-05-21"
```

```
1 as.Date("May/21/2020", format = "%b/%d/%Y")
```

```
## [1] "2020-05-21"
```

うまく動くことが確認できました。念の為に、ロケールを戻しておきます。

```
1 Sys.setlocale(category = "LC_TIME", locale = "ja_JP.UTF-8")
```

```
## [1] "ja_JP.UTF-8"
```

### 8.7.3 POSIXct、POSIXlt 型について

POSIXct、POSIXlt 型は日付だけでなく時間の情報も含むデータ型です。これらは `as.POSIXct()`、`as.POSIXlt()` 関数で作成することができます。どちらも見た目は同じデータ型ですが、内部構造がことなります<sup>6)</sup>。詳細は`?as.POSIXct` または`?as.POSIXlt` を参照してください。

---

## 8.8 NA

NA は欠損値と呼ばれます。これは本来は値があるはずなのがなんらかの理由で欠損していることを意味します。表 8.4 の例を考えてみましょう。

表 8.4: 4 人の支持政党

ID	名前	支持政党の有無	支持政党
1	Yanai	ない	NA
2	Song	ある	ラーメン大好き党
3	Shigemura	ある	鹿児島第一党
4	Tani	ない	NA

3 列目で支持政党があるケースのみ、4 列目に値があります。Yanai と Tani の場合、支持する政党がないため、政治政党名が欠損しています。実際、多くのデータには欠損値が含まれています。世論調査データの場合はもっと多いです。理由としては「Q2 で”はい”を選んだ場合のみ Q3 に進み、それ以外は Q4 へ飛ばす」のようなものもありますが、単に回答を拒否した場合もあります。

<sup>6)</sup> POSIXct 型は基準日 (1970 年 1 月 1 日 00 時 00 分 00 秒 = UNIX 時間) からの符号付き経過秒数であり、POSIXlt 型は日付、時間などがそれぞれ数字として格納されています。可読性の観点からは POSIXlt 型が優れていますが、データ処理の観点から見ると POSIXct 型の方が優れていると言われます。

まずは欠損値が含まれたベクトル `na_vec1` を作ってみましょう。

```
1 na_vec1 <- c(1, NA, 3, NA, 5, 6)
2 na_vec1
```

```
## [1] 1 NA 3 NA 5 6
```

つづいて、データ型を確認してみましょう。

```
1 class(na_vec1)
## [1] "numeric"
```

NA が含まれていてもデータ型は numeric のままでです。これは「一応、欠損しているが、ここに何らかの値が割り当てられるしたらそれは numeric 型だろう」と R が判断しているからです。ある要素が NA か否かを判定するには `is.na()` 関数を使います。

```
1 is.na(na_vec1)
## [1] FALSE TRUE FALSE TRUE FALSE FALSE
```

2 番目と 4 番目の要素が欠損していることが分かります。

欠損値も要素の一つとしてカウントされるため、ベクトルの長さは 6 になります。ベクトルの長さは `length()` 関数で確認できます。

```
1 length(na_vec1)
## [1] 6
```

### 欠損値の取り扱い

欠損値を含むデータの処理方法はやや特殊です。まず、`na_vec1` の要素全てに 1 を足してみましょう。

```
1 na_vec1 + 1
## [1] 2 NA 4 NA 6 7
```

この場合、欠損値の箇所には 1 が足されず、それ以外の要素のみに 1 を足した結果が返っ

てきます。これは直感的に考えると自然です。問題になるのは欠損値が含まれるベクトルを関数に入れた場合です。たとえば、numeric型ベクトル内の要素の総和を求めるには `sum()` 関数を使います。`sum(c(1, 3, 5))` を入力すると 9 が返されます。`na_vec1` は欠損していない要素が 1, 3, 5, 6 であるため、総和は 15 のはずです。確認してみましょう。

```
1 sum(na_vec1)
```

```
## [1] NA
```

このように欠損値を含むベクトルの総和は NA となります。もし、欠損値を除いた要素の総和を求めるには、まずベクトルから欠損値を除去する必要があります。そのためには `is.na()` 関数を使って `na_vec1` の要素を抽出します。ただし、`is.na()` を使うと、欠損値であるところが TRUE になるため、これを反転する必要があります。この場合は `!is.na()` 関数を使います。それでは `is.na()` と `!is.na()` を使って要素を抽出してみましょう。

```
1 na_vec1[is.na(na_vec1)]
```

```
## [1] NA NA
```

```
1 na_vec1[!is.na(na_vec1)]
```

```
## [1] 1 3 5 6
```

`!is.na()` を使うことで欠損値を除いた要素のみを取り出すことができました。これなら `sum()` 関数も使えるでしょう。

```
1 sum(na_vec1[!is.na(na_vec1)])
```

```
## [1] 15
```

これで欠損値を除いた要素の総和を求めることができました。ただし、一部の関数には欠損値を自動的に除去するオプションを持つ場合があります。`sum()` 関数のその一部であり、`na.rm = TRUE` オプションを付けると、欠損値を除いた総和を返します。

```
1 sum(na_vec1, na.rm = TRUE)
```

```
## [1] 15
```

## 欠損値の使い方

主に欠損値を扱うのは入手したデータに含まれる欠損値に対してですが、NA をこちらから生成することもあります。それは空ベクトルを用意する時です。第 10 章では関数の作り方について解説します。関数内で何らかの処理を行い、その結果を返すことになりますが、その結果を格納するベクトルを事前に作っておくこともできます。こちらの方がメモリの観点からは効率的です。以下は第 10 章を読んでから読んでも構いません。

もし、長さ 10 のベクトル `result_vec1` を返すとします。ベクトルの要素として 1 から 10 の数字が入るとなります。一つ目の方法としてはまず、`result_vec1` に 1 を代入し、次は `c()` を使って要素を一つずつ足していく手順です。

```
1 result_vec1 <- 1
2 result_vec1 <- c(result_vec1, 2)
3 result_vec1 <- c(result_vec1, 3)
4 result_vec1 <- c(result_vec1, 4)
5 result_vec1 <- c(result_vec1, 5)
6 result_vec1 <- c(result_vec1, 6)
7 result_vec1 <- c(result_vec1, 7)
8 result_vec1 <- c(result_vec1, 8)
9 result_vec1 <- c(result_vec1, 9)
10 result_vec1 <- c(result_vec1, 10)
```

二つ目の方法はまず、10 個の NA が格納されたベクトル `Reuslt.Vec2` を作っておいて、その中に要素を置換していく方法です。

```
1 result_vec2 <- rep(NA, 10)
2 result_vec2[1] <- 1
3 result_vec2[2] <- 2
4 result_vec2[3] <- 3
5 result_vec2[4] <- 4
6 result_vec2[5] <- 5
7 result_vec2[6] <- 6
8 result_vec2[7] <- 7
9 result_vec2[8] <- 8
```

```
10  result_vec2[9] <- 9
11  result_vec2[10] <- 10
```

以上の手順を第10章で紹介する `for()` を使って反復処理するとなれば、以下のようなコードになります。

```
1  # 方法1
2  result_vec1 <- 1
3
4  for (i in 2:10) {
5      result_vec1 <- c(result_vec1, i)
6  }
7
8  # 方法2
9  result_vec2 <- rep(NA, 10)
10
11 for (i in 1:10) {
12     result_vec2[i] <- i
13 }
```

結果を確認してみましょう。

```
1  result_vec1
2
3  ## [1] 1 2 3 4 5 6 7 8 9 10
4
5  result_vec2
6
7  ## [1] 1 2 3 4 5 6 7 8 9 10
```

コードの書き方は異なりますが、どちらも結果は同じです。また、どちらが早いかというと、これくらいの計算ならどの方法でも同じです。ただし、より大規模の反復作業を行う場合、後者の方が時間が節約でき、コードの可読性も高いです。

## 8.9 NULL

NULL は「存在しない」、空っぽであることを意味します。先ほどの NA はデータは存在するはずなのに、何らかの理由で値が存在しない、または割り当てられていないことを意味しますが、NULL は「存在しません」。したがって、NULL が含まれたベクトルを作成しても表示されません。NULL が含まれた `null_vec1` を作ってみましょう。

```
1 null_vec1 <- c(1, 3, NULL, 5, 10)
2 null_vec1
```

```
## [1] 1 3 5 10
```

3 番目の要素である NULL は表示されません。ということは NA と違って、データの長さも 5 ではなく 4 でしょう。確認してみます。

```
1 length(null_vec1)
```

```
## [1] 4
```

この `null_vec1` のデータ型は何でしょう。

```
1 class(null_vec1)
```

```
## [1] "numeric"
```

`is.null()` 関数もありますが、どうでしょうか。

```
1 is.null(null_vec1)
```

```
## [1] FALSE
```

NULL は存在しないことを意味するため、`null_vec1` は要素が 4 の numeric 型ベクトルです。`is.null()` で NULL が判定できるのは `is.null(NULL)` のようなケースです。

図 8.8 は NA 型と NULL 型の違いについてまとめたものです。

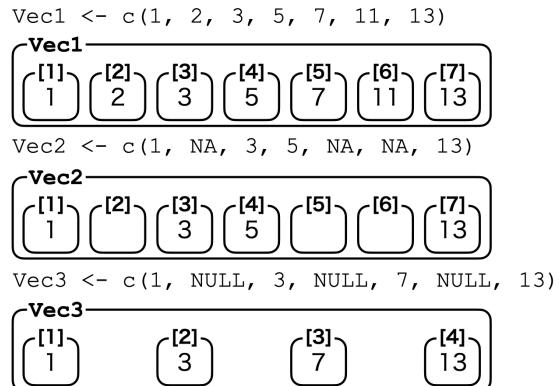


図 8.8: NA と NULL の違い

この NULL はいつ使うのでしょうか。実際、使う機会はありません。強いて言えば、空っぽのリストを作成する際に使うケースがあります。リストについては第 9 章で説明します。以下の例は第 9 章を読み終わってから目を通して下さい。

```

1 null_list1 <- list(Room1 = 1:3,
2                               Room2 = c("Yuki", "Jaehyun", "Hadley"),
3                               Room3 = NULL)
4
5 null_list1

## $Room1
## [1] 1 2 3
##
## $Room2
## [1] "Yuki"    "Jaehyun" "Hadley"
##
## $Room3
## NULL

```

このように予めリストの要素は作っておきたいが、とりあえず空けておく際に使います。続く分析の段階で `null_list1[["Room3"]]` に何かを格納したりすることに使えるでしょう。ちなみにこの場合は `is.null()` が使用可能です。

```
1 is.null(null_list1[["Room3"]])  
## [1] TRUE
```

---

## 8.10 NaN

NaN は numeric 型、中でも double 型の一種ですが、これは計算できない値を意味します。つまり、NaN 値を直接入力することはめったにありませんが、計算の結果として NaN が返されるケースがあります。代表的な例が 0 を 0 で割った場合です。実際、0 を 0 で割ることはできません。ここでは 0 を 0 で割った値を含む `nan_vec1` 作ってみましょう。

```
1 nan_vec1 <- c(2/5, 0/12, 0/0)  
2 nan_vec1
```

```
## [1] 0.4 0.0 NaN  
1 class(nan_vec1)
```

```
## [1] "numeric"
```

先ほどせつめいしましたように、NaN は numeric 型の一部ですので、データ型としては numeric になります。ある値が NaN か否かを判定するには `is.nan()` 関数を使います。

```
1 is.nan(nan_vec1)  
## [1] FALSE FALSE TRUE
```

---

## 8.11 Inf

Inf もまた numeric 型、中でも double 型の一部ですが、これは無限大を意味します。Inf も通常、自分から作成するケースはあまりなく、結果として帰ってくる場合があります。

一つの例が 0 以外の数値を 0 で割った場合です。それではなんらかの数値を 0 を割った値が含まれるベクトル `inf_vec1` を作ってみましょう。

```
1 inf_vec1 <- c(28/95, 3/0, -12/0, 0/0)
2 inf_vec1

## [1] 0.2947368      Inf      -Inf      NaN
1 class(inf_vec1)

## [1] "numeric"
```

正の値を 0 で割ったら `Inf` が負の値を 0 で割ったら `-Inf` が返ってきます。これは正の無限大、負の無限大を意味します。データ型は `NaN` と同様、`numeric` 型ですが、`is.infinite()` を使うと、無限大か否かが判定できます。

```
1 is.infinite(inf_vec1)

## [1] FALSE  TRUE  TRUE FALSE
```

## 第9章

# データの構造

### 9.1 データ構造とは

データ構造 (data structure) とは最小単位であるベクトルを何らかの形で集めたものです。ベクトル自体もデータ構造であり、同じデータ型のベクトルを積み重ねた行列、異なるデータ型の縦ベクトルを横に並べたデータフレームなど、R では様々なデータ構造を提供しています。また、R 内臓のデータ構造以外にも、パッケージ等で提供される独自のデータ構造もあります。ここでは R が基本的に提供している代表的なデータ構造について、その作り方と操作方法について解説します。

---

### 9.2 ベクトル (vector)

#### 9.2.1 ベクトルの作り方

Rにおいてベクトルとは同じデータ型が一つ以上格納されているオブジェクトを意味します。たとえば、以下の `myVec1` は長さ 1 のベクトルです。

```
1 myVec1 <- "R is fun!"  
2 myVec1
```

```
## [1] "R is fun!"
```

もちろん、2つ以上の文字列、または数字を格納することも可能です。以下の `myVec2` は長さ 5 のベクトルです。

```
1 myVec2 <- c(1, 3, 5, 6, 7)
2 myVec2
```

```
## [1] 1 3 5 6 7
```

注意すべきところは、ベクトル内の要素は必ず同じデータ型である必要があるということです。たとえば、数字と文字が混在した `myVec3` を考えてみましょう。

```
1 myVec3 <- c("A", "B", "C", 1, 2, 3)
2 myVec3
```

```
## [1] "A" "B" "C" "1" "2" "3"
```

数字であるはずの 1, 2, 3 が"で囲まれ、文字列に自動的に変換されていることが分かります。実際に、データ型を確認してみましょう。

```
1 class(myVec3)
```

```
## [1] "character"
```

"character"、つまりデータ型が文字列になっていることが分かります。これは `character` 型が `numeric` 型よりも優先順位が高いからです。それでは `numeric` と `logical` 型はどうでしょうか。

```
1 myVec4 <- c(1, 2, 3, TRUE, FALSE)
2 myVec4
```

```
## [1] 1 2 3 1 0
```

```
1 class(myVec4)
```

```
## [1] "numeric"
```

`TRUE` が 1、`FALSE` が 0 となり、自動的に `numeric` 型になりました。一般的によく使われるデータ型は `logical`、`numeric`、`character` ですが、優先順位は `logical < numeric < character` の関係になります。ここで重要なのは優先順位ではなく、異なるデータ型

が含まれるベクトルの場合、自動的にデータ型が統一されるということです。ベクトルを作成する際は、全ての要素が同じ型になるようにしましょう。

### 9.2.2 ベクトルの操作

#### ベクトルの長さ

ベクトルの長さはベクトルに含まれている要素の数です。ベクトルの長さは `length()` 関数で調べることができます。それでは前節で作成した 4 つのベクトルの長さを調べてみましょう。

```
1 length(myVec1) # "R is fun!"  
  
## [1] 1  
  
1 length(myVec2) # c(1, 3, 5, 6, 7)  
  
## [1] 5  
  
1 length(myVec3) # c("A", "B", "C", "1", "2", "3")  
  
## [1] 6  
  
1 length(myVec4) # c(1, 2, 3, 1, 0)  
  
## [1] 5
```

それぞれのベクトルの長さは 1、5、6、5 ということが分かりますね。

#### 要素の抽出

これは既に説明しましたので、6.5 を参照してください。

#### ベクトルの加減乗除

ここでは numeric 型のベクトルを用いた加減乗除について考えたいと思います。R においてベクトルは自動的に反復作業を行います。たとえば、`c(1, 2, 3, 4)` というベクトルがあり、ここに 5 を足すと、ベクトルの全ての要素に対して同じ計算を行います。これは引き算でも、掛け算でも、割り算でも同じです。むろん、べき乗などの様々な操作に

対しても同じです。それでは `myVec2` に対して、5を足したり、引いたり、色々してみましょう。

```
1 myVec2 + 5

## [1] 6 8 10 11 12

1 myVec2 - 5

## [1] -4 -2 0 1 2

1 myVec2 * 5

## [1] 5 15 25 30 35

1 myVec2 / 5

## [1] 0.2 0.6 1.0 1.2 1.4

1 myVec2 ^ 5

## [1] 1 243 3125 7776 16807
```

また、ベクトル同士の演算も可能です。まず、`myVec2` と同じ長さを持つ `myVec4` との計算を考えてみましょう。これは長さが同じであるため、それぞれ同じ位置の要素同士の計算となります。つまり、足し算の場合、`myVec2[1]` と `myVec4[1]` の和、`myVec2[2]` と `myVec4[2]` の和、...といった形です。

```
1 myVec2 + myVec4

## [1] 2 5 8 7 7

1 myVec2 - myVec4

## [1] 0 1 2 5 7

1 myVec2 * myVec4

## [1] 1 6 15 6 0
```

```
1 myVec2 / myVec4
```

```
## [1] 1.000000 1.500000 1.666667 6.000000      Inf
```

もし、ベクトルの長さが異なる場合はどうなるでしょう。たとえば、長さ 2 のベクトル myVec5 と myVec2 の足し算を考えてみましょう。

```
1 (myVec5 <- c(1, 10))
```

```
## [1] 1 10
```

```
1 (myVec6 <- myVec2 + myVec5)
```

```
## Warning in myVec2 + myVec5: longer object length is not a multiple of shorter
## object length
```

```
## [1] 2 13 6 16 8
```

警告メッセージは表示されますが、計算自体はできます。同じ長さのベクトル同士なら、同じ位置の要素同士の計算になりますが、この場合は、短い方のベクトルを繰り返すことにより、長さを合わせることになります。myVec5 の例だと、c(1, 10, 1, 10, 1) のように扱われます。具体的には以下の表のような関係となります。

myVec6[1]	myVec6[2]	myVec6[3]	myVec6[4]	myVec6[5]
=	=	=	=	=
myVec2[1]	myVec2[2]	myVec2[3]	myVec2[4]	myVec2[5]
+	+	+	+	+
myVec5[1]	myVec5[2]	myVec5[1]	myVec5[2]	myVec5[1]

実際は長さが異なるベクトル同士の計算を行うことは滅多にありません。例外としては長さ 1 のベクトルの計算くらいですね。

文字列ベクトルの扱い方はより複雑ですので、第 16 章で詳細に解説します。

## 9.3 行列 (matrix)

行列は名前とおり、行と列で構成されたデータ構造です。ただし、我々が一般に考える「表」とは異なる点があります。それは行列内部の要素に制約がある点です。具体的に、行列の要素となり得るデータ型は numeric、complex、NA 型のみです。ここでは numeric 型のみで構成された行列の作成および操作方法について解説します。

### 9.3.1 行列の作り方

行列を作成するには `matrix()` 関数を使います。引数として、行列に入る数値と行または列の数は必須です。数値はベクトルであり、行・列の数は整数です。以下のような行列を作るにはいくつかの方法があります。

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

```

1 # 方法 1: 行数を指定する
2 Matrix1 <- matrix(c(1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12), nrow = 3)
3 Matrix1

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12

1 # 方法 2: 列数を指定する
2 Matrix2 <- matrix(c(1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12), ncol = 4)
3 Matrix2

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8

```

```
## [3,] 9 10 11 12
```

いずれも同じ結果が得られます。注意してもらいたいところは、第一引数の書き方です。我々は「左から右へ、そして上から下へ」という順番で読むのになっていますが、R の行列は「上から下へ、そして左から右へ」の順番です。もし、「左から右へ、そして上から下へ」のような、より我々にとって読みやすい書き方をするためにはもう一つの引数が必要であり、それが `byrow = TRUE` です。

```
1 (matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), nrow = 3, byrow = TRUE))
```

```
## [,1] [,2] [,3] [,4]
## [1,] 1 2 3 4
## [2,] 5 6 7 8
## [3,] 9 10 11 12
```

むろん、初項 1、公差 1、最大値 12 の等差数列ですので、`1:12` のような書き方も可能です。

```
1 (matrix(1:12, nrow = 3, byrow = TRUE))
```

```
## [,1] [,2] [,3] [,4]
## [1,] 1 2 3 4
## [2,] 5 6 7 8
## [3,] 9 10 11 12
```

データ構造も確認してみましょう。

```
1 class(Matrix1)
```

```
## [1] "matrix" "array"
```

R 4.0.0 からの仕様変更により行列型は `matrix` 構造以外にも `array` の構造も持つようになりました。`array` 型については第 9.6 章で解説します。

## 単位行列

最後にちょっと特殊な行列である単位行列 (identity matrix) の作り方について説明します。単位行列とは行と列の数が同じである正方形の行列ですが、対角線上は全て 1、その

他は全て 0 となっている行列です。たとえば大きさが 4 の単位行列は、

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

です。これは `matrix(c(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1), nrow = 4)` で作成することも可能ですが、`diag()` 関数を使えば簡単に出来ます。サイズが 4 の単位行列は `diag(4)` です。

```
1 (diag(4))

##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

### 9.3.2 行列の操作

まず、以下のような行列 `A` を作ってみましょう。

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

```
1 A <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12),
2 nrow = 3, byrow = TRUE)
```

#### 行列のサイズ

行列の大きさ (行と列の数) を求める際は `dim()` 関数を使います。

```
1 dim(A)

## [1] 3 4
```

結果は長さ 2 のベクトルですが、1 番目の要素が行数、2 番目の要素が列数になります。もし、行列の行数のみ確認したい場合は `dim(A)[1]`、列数なら `dim(A)[2]` で確認することができます。また、全く同じ機能を持つ関数があり、それが `nrow()` と `ncol()` 関数です。

```
1 nrow(A) # 行列の行数を確認: dim(A)[1] と同じ
```

```
## [1] 3
```

```
1 ncol(A) # 列列の列数を確認: dim(A)[2] と同じ
```

```
## [1] 4
```

`dim()` および `nrow()`、`ncol()` 関数は第 9.4 章で紹介するデータフレームでも使用可能です。

### 要素の抽出

ベクトルと同様、要素の抽出は `[]` を使います。ただし、行列は横と縦の 2 次元構成となりますので、行と列のそれぞれの位置を指定する必要があります。たとえば、`A` の 2 行目、3 列目の要素を抽出するためには、

```
1 A[2, 3]
```

```
## [1] 7
```

のように書きます。行か列の位置を省略した場合、指定した列・行全てが抽出されます。2 行目の要素全てを抽出するなら `A[2, ]`、3 列目の要素全てを抽出するなら `A[, 3]` となります。

```
1 A[2, ]
```

```
## [1] 5 6 7 8
```

```
1 A[, 3]
```

```
## [1] 3 7 11
```

この場合、返される値のデータ構造はいずれも **行列** でなく、**ベクトル** です。確認してみま

しょう。

```
1 is.vector(A[2, 3])  
  
## [1] TRUE  
  
1 is.vector(A[, ])  
  
## [1] TRUE  
  
1 is.vector(A[, 3])  
  
## [1] TRUE
```

例外は複数の行と複数の列を同時に抽出した場合です。たとえば、行列 A の 1・2 行目と 2・3 列目の要素を全て抽出するとします。その場合は A[1:2, 2:3] のように書きます。もちろん、: を使わずに A[c(1, 2), c(2, 3)] のような書き方も可能です。抽出後、返された結果のデータ構造も確認してみましょう。

```
1 A[1:2, 3:4]  
  
##      [,1] [,2]  
## [1,]     3     4  
## [2,]     7     8  
  
1 class(A[1:2, 3:4])  
  
## [1] "matrix" "array"
```

この場合、返された結果のデータ構造は行列であることが分かります。

### 行列の足し算と引き算

行列の足し算 (引き算) には

1. 行列内の全要素に対して同じ数字を足す (引く)
2. 同じサイズの 2 つの行列から対応する要素を足す (引く)

2 パターンがあります。例えば、行列 A の全要素に 5 を足す場合は以下のように入力します。

```
1 (A_plus_5 <- A + 5)

##      [,1] [,2] [,3] [,4]
## [1,]     6     7     8     9
## [2,]    10    11    12    13
## [3,]    14    15    16    17
```

同様に、A から 10 を引く場合は-演算子を使います。

```
1 (A_minus_10 <- A - 10)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    -9    -8    -7    -6
## [2,]    -5    -4    -3    -2
## [3,]    -1     0     1     2
```

二つ目は同じサイズの行列同士の足し算と引き算です。行列 A は  $3 \times 4$  の行列ですので、同じサイズの行列 B を作成してみましょう。

```
1 (B <- matrix(c(3, 2, 1, 4, 5, 9, 7, 11, 6, 12, 8, 10),
2                               nrow = 3, byrow = TRUE))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     3     2     1     4
## [2,]     5     9     7    11
## [3,]     6    12     8    10
```

この行列 A と B の足し算はそれぞれ同じ位置の要素同士の和を行列をして返し、これは引き算も同じです。

```
1 A + B
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     4     4     4     8
## [2,]    10    15    14    19
## [3,]    15    22    19    22
```

```
1 A - B

##      [,1] [,2] [,3] [,4]
## [1,]    -2    0    2    0
## [2,]     0   -3    0   -3
## [3,]     3   -2    3    2
```

注意すべき点は、2つの行列は同じ大きさでなければならない点です。たとえば、行列 B の1列から3列までを抽出した  $3 \times 3$  行列 C を作成し、A + C をしてみましょう。

```
1 (C <- B[, 1:3])

##      [,1] [,2] [,3]
## [1,]     3    2    1
## [2,]     5    9    7
## [3,]     6   12    8
```

```
1 A + C
```

```
## Error in A + C: non-conformable arrays
```

このように足し算ができなくなり、これは引き算でも同じです。

### 行列の掛け算

やや特殊なのは行列の掛け算です。行列 A の全要素を 2 倍にしたい場合、これは足し算・引き算と同じやり方で十分です。

```
1 A * 2

##      [,1] [,2] [,3] [,4]
## [1,]     2    4    6    8
## [2,]    10   12   14   16
## [3,]    18   20   22   24
```

ただし、問題は行列同士の掛け算です。行列 A と B の掛け算といえば、直感的には足し算や引き算同様、同じ位置の要素の積と考えるかも知れません。実際 A \* B はそのように計算を行います。

```
1 A * B
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     3     4     3    16
## [2,]    25    54    49    88
## [3,]    54   120    88   120
```

ただし、このような掛け算は「アダマール積 (Hadamard product)」と呼ばれる計算方法であり<sup>1)</sup>、一般的に使う掛け算ではありません。実際、数学において  $A \times B$  はアダマール積を意味すのではなく、アダマール積は  $A \circ B$  または  $A \odot B$  と表記します。

それでは一般的な行列の積は何でしょう。詳しいことは線形代数の入門書に譲りますが、以下のような 2 つの行列  $C$  と  $D$  を考えてみましょう。

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, D = \begin{bmatrix} 2 & 7 & 17 \\ 3 & 11 & 19 \\ 5 & 13 & 23 \end{bmatrix}$$

行列の大きさが異なりますね。行列  $C$  の大きさは  $2 \times 3$ 、 $D$  は  $3 \times 3$  です。実はこれが正しいです。行列の積は  $n \times m$  の行列と  $m \times p$  の行列同士でないと計算できません。 $n$  と  $p$  は同じでも、同じでなくとも構いません。その意味で  $C \times D$  は計算可能でも、 $D \times C$  は計算できません。そして、2 つの行列の積の大きさは  $n \times p$  です。したがって、 $C \times D$  の大きさは  $2 \times 3$  です。

行列の積がどのように求められるかを確認するために、まず行列  $C$  と  $D$  を作成してみましょう。

```
1 (C <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, byrow = TRUE))
```

```
##      [,1] [,2] [,3]
## [1,]     1     2     3
## [2,]     4     5     6
```

---

<sup>1)</sup> 他にも要素ごとの積 (element-wise product)、シューア積 (Schur product) と呼ばれたりします。

```
1 (D <- matrix(c(2, 7, 17, 3, 11, 19, 5, 13, 23), nrow = 3, byrow = TRUE))
```

```
##      [,1] [,2] [,3]
## [1,]    2    7   17
## [2,]    3   11   19
## [3,]    5   13   23
```

この2つの積は`%*%`演算子で計算されますが、まずその結果から見ましょう。

```
1 (E <- C %*% D)
```

```
##      [,1] [,2] [,3]
## [1,]   23   68  124
## [2,]   53  161  301
```

2行2列の行列ができました。これらの数字、どうやった計算されたのでしょうか。Eの $[1, 1]$ は23であり、 $[1, 2]$ は68、 $[2, 1]$ は53、 $[2, 2]$ は161です。各値は以下のように求まります。

- $E[1, 1] = (C[1, 1] * D[1, 1]) + (C[1, 2] * D[2, 1]) + (C[1, 3] * D[3, 1]) = 23$
- $E[1, 2] = (C[1, 1] * D[1, 2]) + (C[1, 2] * D[2, 2]) + (C[1, 3] * D[3, 2]) = 68$
- $E[1, 3] = (C[1, 1] * D[1, 3]) + (C[1, 2] * D[2, 3]) + (C[1, 3] * D[3, 3]) = 124$
- $E[2, 1] = (C[2, 1] * D[1, 1]) + (C[2, 2] * D[2, 1]) + (C[2, 3] * D[3, 1]) = 53$
- $E[2, 2] = (C[2, 1] * D[1, 2]) + (C[2, 2] * D[2, 2]) + (C[2, 3] * D[3, 2]) = 161$
- $E[2, 3] = (C[2, 1] * D[1, 3]) + (C[2, 2] * D[2, 3]) + (C[2, 3] * D[3, 3]) = 301$

大きさ $n \times m$ 行列Cのi行目j列目の要素を $C_{i,j}$ と表記し、大きさ $n \times p$ 行列Dのi行目j列目の要素を $D_{i,j}$ と表記した場合、以下のような関係が成り立ちます。

行列Eのi行目j列目の要素を $e_{i,j}$ と表記した場合、以下のような関係が成り立ちます。

$$e_{i,j} = \sum_{k=1}^m C_{i,k} \cdot D_{k,j}.$$

したがって、行列  $C$  と  $D$  の積は

$$CD = E = \begin{bmatrix} e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,1} & e_{2,2} & e_{2,3} \end{bmatrix} \quad (9.1)$$

$$= \begin{bmatrix} \sum_{k=1}^m C_{1,k} \cdot D_{k,1} & \sum_{k=1}^m C_{1,k} \cdot D_{k,2} & \sum_{k=1}^m C_{1,k} \cdot D_{k,3} \\ \sum_{k=1}^m C_{2,k} \cdot D_{k,1} & \sum_{k=1}^m C_{2,k} \cdot D_{k,2} & \sum_{k=1}^m C_{2,k} \cdot D_{k,3} \end{bmatrix}. \quad (9.2)$$

このように業績同士の積はかなり求めるのが面倒ですが、R を使えば一瞬で終わります。

## 行列式

行列式 (determinant) は  $n \times n$  の正方行列のみに対して定義される値の一つです。主に一次方程式において解が存在するか否かを判断するために用いられる数値ですが<sup>2)</sup>、その詳しい意味や求め方については線形代数の入門書を参照してください。

行列  $A$  の行列式は一般的に  $\det(A)$ 、または  $|A|$  と表記されます。行列式を求める R の関数は `det()` です。それでは適当に正方行列  $F$  を作成し、その行列を求めてみましょう。

```
1 (F <- matrix(c(2, -6, 4, 7, 2, 3, 8, 5, -1), nrow = 3, byrow = 3))
```

```
##      [,1] [,2] [,3]
## [1,]    2   -6    4
## [2,]    7    2    3
## [3,]    8    5   -1
```

```
1 det(F)
```

```
## [1] -144
```

行列  $F$  の行列式は-144 です。この場合、以下の一次方程式に何らかの一組の解が存在することを意味します ( $p, q, r$  は任意の実数)。

---

<sup>2)</sup> 行列式が 0 でない場合において一次方程式には一組の解があります。

$$2x - 6y + 4z = p, \quad (9.3)$$

$$7x + 2y + 3z = q, \quad (9.4)$$

$$8x + 5y - 1z = r. \quad (9.5)$$

しかし、以下のような連立方程式はいかがでしょう。これは二組以上の解が存在する方程式です<sup>3)</sup>。

$$2x - 6y + 4z = p, \quad (9.6)$$

$$1x - 3y + 2z = q, \quad (9.7)$$

$$5x + 9y + 3z = r. \quad (9.8)$$

実際に行列 G を作成し、`det(G)` を計算してみましょう。

```
1 (G <- matrix(c(2, -6, 4, 1, -3, 2, 5, 9, 3), nrow = 3, byrow = 3))

##      [,1] [,2] [,3]
## [1,]     2    -6     4
## [2,]     1    -3     2
## [3,]     5     9     3

1 det(G)

## [1] 0
```

$|G|$  は 0 であることが分かりますね。

## 階数

行列の特徴を表す代表的な数値の一つが階数 (rank) です。詳しい説明は省きますが、一次方程式の例だと、階数が行列の行数と一致する場合、一組の解が存在することを意味します。階数を求める方法は `qr(行列)$rank` であり、行列 F と G の階数を確認してみましょう。

---

<sup>3)</sup> なぜなら 2 行目が 1 行目の 0.5 倍になっているからです。

```
1 qr(F)$rank
```

```
## [1] 3
```

```
1 qr(G)$rank
```

```
## [1] 2
```

どれも 3 行の行列ですが、行列 F の階数は 3、行列 G の階数は 2 です。G の階数は G の行数より小さいため、連立方程式に一組の解がないことが分かります。

階数の活用先は様々であり、行列式とは違って、正方行列でなくても計算可能です。

## 逆行列

逆行列とは掛け算すると単位行列となる行列を意味します。行列 A の逆行列は一般的に  $A'$  と表記し、 $A \times A' = I$  となります。この逆行列の求め方は非常に複雑であり、一定以上の大きさの行列になると手計算で解くのはほぼ不可能です。しかし、R では逆行列を計算する `solve()` 関数が内蔵されております。

以下の行列 A (A) の逆行列、 $A'$  (`Ap`) を求めてみましょう。そして  $A \times A'$  が単位行列になるかまで確認してみます。

```
1 # 行列 A の作成
```

```
2 (A <- matrix(c(1, -2, 2, 0, 1, -1, 1, 0, 1), nrow = 3))
```

```
##      [,1] [,2] [,3]
## [1,]     1     0     1
## [2,]    -2     1     0
## [3,]     2    -1     1
```

```
1 # 行列 A の逆行列を計算し、Ap に格納
```

```
2 (Ap <- solve(A))
```

```
##      [,1] [,2] [,3]
## [1,]     1    -1    -1
## [2,]     2    -1    -2
```

```

## [3,]    0    1    1
1 # A と Ap の積が単位行列であることを確認
2 A %*% Ap

## [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1

```

ちゃんと単位行列ができましたね。

### 転置

行列  $A$  の転置行列は  $A^T$  と表記され、以下のような関係となります。

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

正方行列の場合、対角成分を除き、全ての要素が対角成分を中心に反転していることが分かりますね。このような転置行列の作成には `t()` 関数を使います。それでは行列 `A` とその転置行列 `At` を作ってみましょう。

```

1 A <- matrix(1:9, byrow = TRUE, nrow = 3)
2 A

## [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9

1 At <- t(A)

```

ちなみに転置行列は正方行列でなくても作成できます。

## 9.4 データフレーム (data.frame)

データフレームはデータ分析の際に最もよく見るデータ構造です。我々が一般的に考える表形式のデータはデータフレームです。行列も見た目は表に近いですが、行列は中身の要素が numeric または complex 型に限定されるに対して<sup>4)</sup>、データフレームは character や factor、Date など様々なデータ型が許容されます。

### 9.4.1 データフレームの作成

まずは、表@ref(tab: datastructure-dataframe-1) のようなデータフレームを作成して見ましょう。データフレームを作成する際は `data.frame()` 関数を使います。

表 9.2: ‘myDF’の中身

ID	Name	Math	Stat
1	Yanai	50	25
2	Song	90	5
3	Shigemura	100	100
4	Tani	80	85

```
1 myDF <- data.frame(  
2   ID    = 1:4,  
3   Name = c("Yanai", "Song", "Shigemura", "Tani"),  
4   Math = c(50, 90, 100, 80),  
5   Stat = c(25, 5, 100, 85)  
6 )  
7  
8 myDF
```

```
##   ID      Name Math Stat  
## 1  1      Yanai  50   25
```

<sup>4)</sup> NA も可能です。

```
## 2 2      Song   90   5
## 3 3 Shigemura 100 100
## 4 4      Tani   80   85
```

データ構造を確認してみましょう。

```
1 class(myDF)

## [1] "data.frame"
```

データフレームを作成する際、事前にベクトルを用意してから作成することも可能です。

```
1 myDF_ID <- 1:4
2 myDF_Name <- c("Yanai", "Song", "Shigemura", "Tani")
3 myDF_Math <- c(50, 90, 100, 80)
4 myDF_Stat <- c(25, 5, 100, 85)
5
6 myDF2 <- data.frame(myDF_ID, myDF_Name, myDF_Math, myDF_Stat)
7
8 myDF2
```

```
##   myDF_ID myDF_Name myDF_Math myDF_Stat
## 1      1      Yanai       50      25
## 2      2      Song       90       5
## 3      3 Shigemura    100     100
## 4      4      Tani       80      85
```

この場合、列の名前はベクトル名そのままになります。もし、列名を指定したい場合は以下のように作成します。

```
1 myDF3 <- data.frame(
2   ID    = myDF_ID,
3   Name  = myDF_Name,
4   Math  = myDF_Math,
5   Stat  = myDF_Stat
6 )
```

```
7  
8 myDF3  
  
##   ID      Name Math Stat  
## 1 1      Yanai  50   25  
## 2 2      Song   90   5  
## 3 3 Shigemura 100  100  
## 4 4      Tani   80   85
```

以上をコードを考えてみると、データフレームは複数のベクトルを横方向にくっつけたものになります。注意すべき点としては各ベクトルの長さが一致している点です。myDFの場合、4つのベクトルで構成されていますが、全てのベクトルの長さは4です。むろん、長さが異なる場合もデータフレームは作成できます。この場合、長さが足りないベクトルは最も長さが長いベクトルに合わせて繰り返されます。例としてmyDF4を作つてみましょう。

```
1 myDF4 <- data.frame(  
2   ID    = 1:4,  
3   Name = c("Yanai", "Song", "Shigemura", "Tani"),  
4   Math = c(50, 90, 100, 80),  
5   Stat = c(25, 5, 100, 85),  
6   City = "Kobe",  
7   Food = c("Ramen", "Udon")  
8 )  
9  
10 myDF4
```

```
##   ID      Name Math Stat City  Food  
## 1 1      Yanai  50   25 Kobe Ramen  
## 2 2      Song   90   5  Kobe Udon  
## 3 3 Shigemura 100  100 Kobe Ramen  
## 4 4      Tani   80   85 Kobe Udon
```

City列はすべて"Kobe"が入り、Food列は長さが足りない3, 4番目の要素に1, 2番目

の要素が代入されます。実際はあまり使わない使い方ですが、ある列の要素が全て同じ場合、このような使い方をすることがあります。

#### 9.4.2 データフレームの操作

##### データフレームの大きさ

行列と同様、`dim()` から行と列の数を、`nrow()` からは行数を、`ncol()` から列数を計算することができます。

```
1 dim(myDF4)
  ## [1] 4 6
1 nrow(myDF4)
  ## [1] 4
1 ncol(myDF4)
  ## [1] 6
```

##### 要素の抽出

まず、データフレーム内要素の抽出についてですが、行列型と同じやり方で問題ありません。つまり、`[行番号, 列番号]` で要素の抽出が可能です。これに加え、データフレームには\$を使った抽出方法があります。

まず、行列と同じやり方で `myDF4` の 2 行目、6 列目の要素を抽出してみましょう。

```
1 myDF4[2, 6]
  ## [1] "Udon"
```

行を丸ごと抽出したい場合は、列を指定しません。`myDF4` の 3, 4 行目を抽出してみましょう。

```
1 myDF4[3:4, ] # myDF4[c(3, 4), ] も同じ
```

```
##   ID      Name Math Stat City  Food
## 3 3 Shigemura  100 100 Kobe Ramen
## 4 4      Tani   80  85 Kobe  Udon
```

同じやり方で列を抽出することも可能です。好きな食べ物 (Food) 列を抽出してみましょう。

```
1 myDF4[, 6]
```

```
## [1] "Ramen" "Udon"  "Ramen" "Udon"
```

列を抽出する場合は、列名を指定することも可能です。やり方は [, "列名"] による方法、\$列名による方法があります。Name 列を抽出してみましょう。

```
1 myDF4[, "Name"]
```

```
## [1] "Yanai"      "Song"       "Shigemura"  "Tani"
```

```
1 myDF4$Name
```

```
## [1] "Yanai"      "Song"       "Shigemura"  "Tani"
```

どれも同じ結果が返されます。後者の方が簡単ですが、一つの列しか抽出できない限界があります。それに比べ、前者は c() を使うことで複数の列を同時に抽出することも可能です。それぞれの場面に応じて使い分けていきましょう。

また、\$で列を抽出した場合、抽出されたものはベクトル扱いになるため、[] を使った要素の抽出が可能です。例えば myDF4 の Name 列の 2 番目の要素を抽出してみましょう。

```
1 myDF4$Name[2]
```

```
## [1] "Song"
```

あまり意識する必要はありませんが、抽出後のデータ構造について簡単に説明します。データフレームから一部の要素を抽出した結果物は必ずしもデータフレームにはなりません。

操作	返されるデータ型
1 1行を抽出する	データフレーム
2 複数の行を抽出する	データフレーム
3 1列を抽出する	ベクトル
4 複数の列を抽出する	データフレーム

注目するのは3番目の例ですが、これはRの最小単位がベクトルであり、データフレームもベクトルの集めだからです。データフレームは「縦」ベクトルを横に並べたものです。もし、行を抽出した場合、その要素は全て同じデータ型だとは限りません。実際、myDF4の3行目を抽出しても、中にはcharacter型とnumeric型と混在しています。しかし、一つの列のみを抽出した場合、全ての要素は必ず同じデータ型となるため、ベクトルとして扱うことが可能です。同様に、行列型の一列を抽出した場合も結果はベクトルとなります。ただし、行列は全ての要素がNAを除き、同じであるため、一行を抽出しても結果はベクトル型となります。

## セルの修正

特定のセルの修正する方法は簡単です。先ほど、データフレームから一つのセルを取り出すにはデータフレーム名 [行番号, 列番号] だけでした。そのセルを修正するにはベクトルと同様、データフレーム名 [行番号, 列番号] <- 新しい値のように入力します。

たとえば、重村さんが数学試験で不正が発覚し、0点になるとします。そのためには、myDF4の3行・3列目の要素を0に修正する必要がありますが、以下のようなコマンドで修正可能です。

```
1 myDF4[3, 3] <- 0
2 myDF4
```

```
##   ID      Name Math Stat City  Food
## 1  1      Yanai   50   25 Kobe Ramen
## 2  2      Song    90    5 Kobe  Udon
## 3  3 Shigemura     0  100 Kobe Ramen
## 4  4      Tani    80   85 Kobe  Udon
```

ちゃんと重村さんの数学点数が 0 点になりました。ざまあみろですね！

もう一つのやり方としては、一旦、列を取り出し、ベクトルにおける要素の置換操作を行う方法です。矢内大先生が神戸から離れ、高知県へ移住し、小物の宋は京都へ移住したとします。myDF4 の City 列の 1・2 番目要素を c("Kochi", "Kyoto") に修正する必要があります。そのためには以下のように入力します。

```
1 myDF4$City[c(1, 2)] <- c("Kochi", "Kyoto")
2 myDF4
```

```
##   ID      Name Math Stat  City  Food
## 1  1      Yanai   50   25 Kochi Ramen
## 2  2      Song    90    5 Kyoto Udon
## 3  3 Shigemura     0  100  Kobe Ramen
## 4  4      Tani    80   85  Kobe Udon
```

修正した要素が反映されました。

### 列の追加・修正

まずは、データフレームに列を追加する方法について紹介します。方法は

```
1 データフレーム名$新しい列名 <- ベクトル
```

のように入力するだけです。たとえば、4 人を対象に英語試験を行い、それぞれの点数が 95 点、50 点、80 点、5 点だとします。この英語試験の成績を myDF4 の English 列として追加してみましょう。

```
1 myDF4$English <- c(95, 50, 80, 5)
```

もちろん、以下のように予めベクトルを作成してから代入することも可能です。

```
1 English_Score <- c(95, 50, 80, 5)
2 myDF4$English <- English_Score
```

どれも同じ結果になりますが、結果を見てみましょう。

```

1 myDF4

##   ID      Name Math Stat  City  Food English
## 1  1      Yanai   50   25 Kochi  Ramen      95
## 2  2      Song    90    5 Kyoto  Udon      50
## 3  3 Shigemura    0  100  Kobe  Ramen      80
## 4  4      Tani    80   85  Kobe  Udon       5

```

「English が Stat の次じゃなくて気持ち悪い！」と思う方もいるかも知れませんが、列順番の変更については第 12 章で解説します。まずは、これで我慢しましょう。

次は、列の置換についてです。実はよく考えてみるとこれは列の追加と全く同じです。たとえば、英語試験において配点が 5 点の問題にミスが見つかり、全生徒の英語成績に 5 点を上乗せるとします。そのためにはベクトル myDF4\$English の全ての要素に 5 を足し、それをもう一回 myDF4\$English に代入すれば良いです。

```

1 myDF4$English <- myDF4$English + 5
2 myDF4

```

```

##   ID      Name Math Stat  City  Food English
## 1  1      Yanai   50   25 Kochi  Ramen     100
## 2  2      Song    90    5 Kyoto  Udon      55
## 3  3 Shigemura    0  100  Kobe  Ramen      85
## 4  4      Tani    80   85  Kobe  Udon      10

```

これで English 列の修正ができました。

### 行の追加・修正はなるべくしない

行の追加はなるべくしない方が良いです。その理由について考えてみましょう。今は 4 人の生徒のデータがありますが、ここにもう一人の生徒のデータを追加するとします。そのためには myDF4 の 5 行目に生徒の ID、名前、数学・統計学の成績、居住地域、好きな食べ物、英語の成績を入れれば良いでしょう。新しい学生、吐合さんの数学・統計学・英語成績は 50, 50, 50 点、居住地域は芦屋、好きな食べ物は二郎だとします。早速追加してみましょう。

```
1 myDF4[5, ] <- c(5, "Hakiai", 50, 50, "Ashiya", "Jiro", 50)
2 myDF4
```

```
##   ID      Name Math Stat   City  Food English
## 1  1      Yanai   50   25  Kochi  Ramen     100
## 2  2      Song    90    5  Kyoto  Udon      55
## 3  3 Shigemura     0  100   Kobe  Ramen      85
## 4  4      Tani    80   85   Kobe  Udon      10
## 5  5      Hakiai   50   50  Ashiya  Jiro      50
```

問題なく吐合さんのデータが追加されたように見えますが、実は問題があります。myDF4 の Stat 列を取り出して見ましょう。

```
1 myDF4$Stat
```

```
## [1] "25"  "5"   "100" "85"  "50"
```

異常に気づきましたか。それではこのベクトルのデータ型を確認してみましょう。

```
1 class(myDF4$Stat)
```

```
## [1] "character"
```

元々は numeric 型であるはずの Stat 列が character 型になりました。その理由は明白です。ベクトルの要素は全て同じデータ型だからです。そして、numeric と character 型が混在している場合は、自動的に（優先順位の高い）character 型になります。以下の 2 つのコマンドが同じであることは理解できるでしょう。

```
1 # Case 1
2 myDF4[5, ] <- c(5, "Hakiai", 50, 50, "Ashiya", "Jiro", 50)
3
4 # Case 2
5 Hakiai_Data <- c(5, "Hakiai", 50, 50, "Ashiya", "Jiro", 50)
6 myDF4[5, ] <- Hakiai_Data
```

こここのベクトル Hakiai\_Data が強制的に character 型になるため、myDF4 の 5 行目の要

素は全て character 型になります。また、データフレームは縦ベクトルを横に並べたものであるなら、myDF4\$Math 列に character 型の要素が追加されることによって、列も全て character 型になってしまいます。したがって、データフレームに character 型と numeric 型が混在している状況において新しい行の追加は全ての要素を character 型に変えてしまうのです。むろん、データフレームの全要素が numeric 型であれば、このような問題は生じますが、numeric 型のみで構成されたデータフレームはなかなかないでしょう。

5 行目を消しても問題は解決しないので、結局は列のデータ型を強制的に変更する必要があります。

```
1 myDF4$ID      <- as.numeric(myDF4$ID)
2 myDF4$Math     <- as.numeric(myDF4$Math)
3 myDF4$Stat     <- as.numeric(myDF4$Stat)
4 myDF4$English <- as.numeric(myDF4$English)
5
6 class(myDF4$Math)

## [1] "numeric"
```

これでやっと元通りになりましたね。

どうしても行を追加したい場合は、以下のようなやり方もありますが、おすすめはできません。

```
1 myDF4[6, ] <- rep(NA, 7) # myDF4 の 6 行目を追加し、7 つの欠損値を代入
2
3 myDF4$ID[6]      <- 6          # myDF4$ID の 6 番目の要素に 6 を代入
4 myDF4>Name[6]    <- "Yukawa"  # myDF4>Name の 6 番目の要素に Yukawa を代入
5 myDF4$Math[6]     <- 80        # 以下、省略
6 myDF4$Stat[6]     <- 30
7 myDF4$City[6]     <- "Hiroshima"
8 myDF4$Food[6]     <- "Ramen"
9 myDF4$English[6]  <- 90
10
11 myDF4
```

```
##   ID      Name Math Stat      City  Food English
## 1  1      Yanai   50   25    Kochi  Ramen     100
## 2  2      Song    90    5    Kyoto  Udon      55
## 3  3 Shigemura    0  100    Kobe  Ramen     85
## 4  4      Tani    80   85    Kobe  Udon      10
## 5  5    Hakiai   50   50   Ashiya  Jiro      50
## 6  6    Yukawa   80   30 Hiroshima  Ramen     90
```

```
1 class(myDF4$English)
```

```
## [1] "numeric"
```

欠損値 (NA) はどのようなデータ型にも対応できる特徴を利用すれば、このような操作も可能ですが、かなり面倒です。そもそも実際の分析において任意の行を追加することは滅多にないはずです。

### 9.4.3 tibble 型

data.frame 型に似ているデータ型として tibble 型があります。これは R 内蔵のデータ型ではありませんが、R の必須パッケージとも言える{tidyverse}内に含まれているデータ型であり、これからもどんどん普及していくでしょう。tibble 型と data.frame 型の違いを説明するために、同じデータをそれぞれのデータ型で読み込んでみましょう。

とりあえず read.csv() で読み込み、VoteDF1 と名付けましょう。

```
1 VoteDF1 <- read.csv("Data/Vote.csv")
```

続いて VoteDF1 を as\_tibble() 関数を使って tibble 型にし、VoteDF2 と名付けます。そして、それぞれのデータ構造を確認してみます。

```
1 VoteDF2 <- as_tibble(VoteDF1)
```

```
2
```

```
3 class(VoteDF1)
```

```
## [1] "data.frame"
```

```
1 class(VoteDF2)
```

全く同じデータですが、VoteDF2 には `tbl` と `tbl_df` というクラスが追加されていることが分かります。それではデータの中身を覗いてみましょう。まずは、`data.frame` 型から

1 VoteDF1

##	ID	Pref	Zaisei	Over65	Under30	LDP	DPJ	Komei	Ishin	JCP	SDP
## 1	1	北海道	0.41903	29.09	24.70	32.82	30.62	13.41	3.43	11.44	1.68
## 2	2	青森県	0.33190	30.14	23.92	40.44	24.61	12.76	3.82	8.92	3.41
## 3	3	岩手県	0.34116	30.38	24.48	34.90	22.44	8.61	5.16	11.24	5.29
## 4	4	宮城県	0.59597	25.75	27.29	36.68	25.40	13.42	3.97	9.99	3.62
## 5	5	秋田県	0.29862	33.84	21.35	43.46	22.72	11.19	5.17	7.56	5.12
## 6	6	山形県	0.34237	30.76	24.75	42.49	21.47	11.78	4.30	7.60	5.20
## 7	7	福島県	0.50947	28.68	25.23	33.82	28.31	10.96	3.43	10.45	3.24
## 8	8	茨城県	0.63309	26.76	26.60	40.64	18.95	15.05	6.67	10.07	2.88
## 9	9	栃木県	0.62166	25.87	26.78	38.78	21.63	12.42	10.88	7.00	2.05
## 10	10	群馬県	0.60277	27.60	26.59	42.06	19.31	13.85	5.61	10.00	2.44
## 11	11	埼玉県	0.76548	24.82	27.66	32.30	20.40	16.00	7.23	13.94	1.91
## 12	12	千葉県	0.77694	25.86	26.71	37.79	21.70	13.98	5.46	11.34	2.01
## 13	13	東京都	1.00321	22.67	27.39	34.37	19.76	11.44	7.34	14.21	2.82
## 14	14	神奈川県	0.91745	23.86	27.84	34.92	21.49	12.18	7.77	12.46	2.79
## 15	15	新潟県	0.43519	29.86	25.23	43.66	25.25	8.27	4.39	8.00	3.76
## 16	16	富山県	0.45307	30.54	24.88	44.16	24.22	9.81	5.06	5.79	5.02
## 17	17	石川県	0.46812	27.87	27.25	48.09	18.54	11.00	6.36	7.07	2.36
## 18	18	福井県	0.37820	28.63	26.70	45.29	17.52	10.91	13.09	5.66	2.09
## 19	19	山梨県	0.37876	28.41	26.37	37.36	28.04	12.83	4.32	9.20	1.67
## 20	20	長野県	0.47586	30.06	25.52	35.27	27.69	10.70	4.23	12.61	3.59
## 21	21	岐阜県	0.52358	28.10	27.22	39.71	23.62	12.33	5.51	9.41	1.98
## 22	22	静岡県	0.70999	27.79	26.28	37.47	24.13	12.68	7.99	9.59	2.22
## 23	23	愛知県	0.92052	23.79	29.44	34.32	29.27	11.67	6.41	9.55	2.14
## 24	24	三重県	0.57544	27.90	26.74	33.67	34.23	13.07	5.09	7.59	1.76

```

## 25 25 滋賀県 0.53932 24.15 29.96 37.85 22.25 9.57 12.82 11.44 1.41
## 26 26 京都府 0.56713 27.51 27.76 31.18 19.93 12.25 11.16 18.50 1.55
## 27 27 大阪府 0.74980 26.15 27.55 22.12 9.30 16.39 34.86 11.37 1.25
## 28 28 兵庫県 0.62062 27.09 26.94 31.71 15.84 15.37 19.50 10.30 2.01
## 29 29 奈良県 0.41269 28.70 26.86 33.51 18.41 13.44 18.94 9.17 1.66
## 30 30 和歌山県 0.31955 30.89 25.08 39.61 12.10 18.00 13.51 10.43 1.23
## 31 31 鳥取県 0.25486 29.71 25.86 41.62 20.93 16.71 5.98 7.14 2.56
## 32 32 島根県 0.24170 32.48 24.59 48.24 19.14 13.43 4.75 7.50 2.51
## 33 33 岡山県 0.50096 28.66 27.44 37.87 19.91 17.18 10.67 7.87 1.60
## 34 34 広島県 0.58581 27.53 27.46 39.93 18.53 15.20 9.69 8.52 2.35
## 35 35 山口県 0.42560 32.07 24.91 46.75 17.27 16.50 5.31 7.39 1.78
## 36 36 徳島県 0.32018 30.95 24.31 38.44 17.72 16.87 9.46 9.10 1.81
## 37 37 香川県 0.46060 29.93 25.29 44.07 16.60 15.14 6.20 7.56 5.07
## 38 38 愛媛県 0.41181 30.62 24.75 43.57 19.28 14.82 6.77 6.97 2.40
## 39 39 高知県 0.24472 32.85 23.60 37.01 16.95 15.83 3.93 17.41 2.91
## 40 40 福岡県 0.61836 25.90 28.21 36.52 19.08 17.15 7.03 10.78 3.33
## 41 41 佐賀県 0.32938 27.68 27.97 43.53 21.10 15.48 4.85 5.67 4.16
## 42 42 長崎県 0.31562 29.60 25.84 41.70 20.68 16.86 5.12 6.27 3.48
## 43 43 熊本県 0.38688 28.78 27.18 46.54 19.29 15.27 4.53 6.32 2.60
## 44 44 大分県 0.35828 30.45 25.65 39.44 18.37 13.26 4.42 6.85 13.05
## 45 45 宮崎県 0.32034 29.49 26.26 40.11 14.50 17.11 5.74 7.27 6.81
## 46 46 鹿児島県 0.32140 29.43 26.04 45.97 16.19 14.49 6.47 6.52 3.62
## 47 47 沖縄県 0.31535 19.63 33.37 27.82 13.29 15.09 7.66 15.64 12.13

```

つづいて、tibble型は、

<sup>1</sup> VoteDF2

```

## # A tibble: 47 x 11
##       ID Pref   Zaisei Over65 Under30     LDP     DPJ Komei Ishin     JCP     SDP
##   <int> <chr>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>  <dbl>  <dbl>  <dbl>
## 1     1 北海道  0.419  29.1   24.7   32.8   30.6  13.4   3.43  11.4   1.68
## 2     2 青森県  0.332  30.1   23.9   40.4   24.6  12.8   3.82  8.92   3.41
## 3     3 岩手県  0.341  30.4   24.5   34.9   22.4   8.61  5.16  11.2   5.29

```

```

## 4 4 宮城県 0.596 25.8 27.3 36.7 25.4 13.4 3.97 9.99 3.62
## 5 5 秋田県 0.299 33.8 21.4 43.5 22.7 11.2 5.17 7.56 5.12
## 6 6 山形県 0.342 30.8 24.8 42.5 21.5 11.8 4.3 7.6 5.2
## 7 7 福島県 0.509 28.7 25.2 33.8 28.3 11.0 3.43 10.4 3.24
## 8 8 茨城県 0.633 26.8 26.6 40.6 19.0 15.0 6.67 10.1 2.88
## 9 9 栃木県 0.622 25.9 26.8 38.8 21.6 12.4 10.9 7 2.05
## 10 10 群馬県 0.603 27.6 26.6 42.1 19.3 13.8 5.61 10 2.44
## # ... with 37 more rows

```

同じデータですが、表示画面がやや異なります。data.frame 型は全ての列と行が表示されましたが、tibble 型の場合、「画面に収まる」程度しか表示されません。表示されなかった行や列に関しては最後に表示されています。今回は全ての列が表示されました。たとえば、VoteDF2 の場合、下段にこのように表示される場合があります（画面の大きさによって変わり、表示されないケースもあれば、SDP 以外の変数も省略されるケースがあります。）。

```
## # ... with 37 more rows, and 1 more variable: SDP <dbl>
```

これは表示されなかった行が 37 行あり、SDP という変数も表示されていないということです。他の違いとしては、tibble 型の場合、最初にデータのサイズ（47 行 11 列）が、各変数名の下にデータ型（<int>、<chr>、<dbl>など）が表示されるという点です。

本書は tibble 型と data.frame 型を区別して解説はしませんが、一部の章・節においては tibble 型を念頭において解説をします。tibble 型の作り方は先ほどのように as\_tibble() 使う方法もありますが、readr::read\_csv() を使う方法もあります<sup>5)</sup>。read\_csv() を使うと各列がどのようなデータ型として読み込まれたかも表示されるので便利です。{readr} パッケージは{tidyverse} に含まれているため、普段では以下のような使い方で十分です。

```

1 VoteDF3 <- read_csv("Data/Vote.csv")
2
3 class(VoteDF3) # VoteDF3 の構造

```

---

<sup>5)</sup> tibble 型を直接作成するには tibble() 関数を使いますが、data.frame() と同じ使い方で問題ありません。他にも tribble() 関数を使った方法もありますが、詳細は?tribble で確認してください。

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"          "data.frame"
```

tibble 型と data.frame 型の最も大きな違いは、data.frame 型の一つのセルには長さ 1 のベクトルしか入らない一方、tibble 型は何でも入るという点です。つまり、tibble 型なら一つのセルに数字や文字列だけでなく、長さ 2 以上のベクトル、後述するリスト型、行列、さらには tibble 型も入れることができます。これについては本書の中盤以降に解説します。

本書で「データフレーム」と書いた場合、それは tibble でも適用可能です。ただし、tibble と明示した場合、データフレームでは適用不可能です。

---

## 9.5 リスト (list)

リスト型は「様々なデータ構造を集めたもの」です。これは複数のデータフレームが格納されたオブジェクト、複数のベクトルが格納されたオブジェクト、行列とデータフレームが混在したオブジェクトなどを意味します。また、リスト型データはリスト型データを含むことも可能であり、非常に柔軟なデータ構造です。

### 9.5.1 リスト型データの作成

ここではまず、2 つのデータフレームを含むリストを作成してみます。それぞれのデータフレームは FIFA 国別サッカーランキングであり<sup>6)</sup>、本書のサンプルデータの `FIFA_Women.csv` と `FIFA_Men.csv` です。まずは、2 つのデータを読み込み、`Soccer_W` と `Soccer_M` という名のオブジェクトに格納しましょう。また、ここでは単に例を見せるだけなので、全データを利用するのではなく、最初の 10 行のみを利用します。

```
1 Soccer_W <- read.csv("Data/FIFA_Women.csv")
2 Soccer_M <- read.csv("Data/FIFA_Men.csv")
3
```

---

<sup>6)</sup> 厳密には国別ではありません。イギリスの場合、イングランド、スコットランド、北アイルランド、ウェールズがそれぞれ独立したチームとして FIFA に加盟しています。

```
4 Soccer_W <- Soccer_W[1:10, ]
5 Soccer_M <- Soccer_M[1:10, ]
```

リストの作成には `list()` 関数に入れたいオブジェクト名を指定するだけです。それでは `List1` という名前で `Soccer_W` と `Soccer_M` データフレームを入れてみましょう。格納後は `List1` のデータ構造を `class()` で確認します

```
1 List1 <- list(Soccer_W, Soccer_M)
2 class(List1)

## [1] "list"

1 List1

## [[1]]
#> #>   ID          Team  Rank Points Prev_Points Confederation
#> #> 1   1          Albania 75  1325      1316        UEFA
#> #> 2   2          Algeria 85  1271      1271        CAF
#> #> 3   3  American Samoa 133 1030      1030        OFC
#> #> 4   4          Andorra 155  749       749        UEFA
#> #> 5   5          Angola 121 1117      1117        CAF
#> #> 6   6 Antigua and Barbuda 153  787       787 CONCACAF
#> #> 7   7          Argentina 32  1659      1659      CONMEBOL
#> #> 8   8          Armenia 126 1103      1104        UEFA
#> #> 9   9          Aruba 157  724       724 CONCACAF
#> #> 10 10          Australia 7  1963      1963        AFC
#> #
#> #
## [[2]]
#> #>   ID          Team  Rank Points Prev_Points Confederation
#> #> 1   1  Afghanistan 149 1052      1052        AFC
#> #> 2   2          Albania 66 1356      1356        UEFA
#> #> 3   3          Algeria 35 1482      1482        CAF
#> #> 4   4  American Samoa 192  900       900        OFC
#> #> 5   5          Andorra 135 1082      1082        UEFA
```

```

## 6   6           Angola  124  1136      1136      CAF
## 7   7           Anguilla 210   821       821       CONCACAF
## 8   8 Antigua and Barbuda 126   1127      1127      CONCACAF
## 9   9           Argentina 9   1623      1623      CONMEBOL
## 10 10           Armenia 102   1213      1213      UEFA

```

一つのオブジェクトに 2 つのデータフレームが入っていますね。これは 2 つのデータフレームで構成された長さ 2 のリストです。ただし、どちらが女性ランキングで、どちらが男子ランキングか区別が難しいです。この場合、各データフレームに名前を付けることも可能です。

```

1 List2 <- list(Women = Soccer_W, Men = Soccer_M)
2 List2

```

```

## $Women
##   ID          Team Rank Points Prev_Points Confederation
## 1  1           Albania 75   1325      1316      UEFA
## 2  2           Algeria 85   1271      1271      CAF
## 3  3 American Samoa 133  1030      1030      OFC
## 4  4           Andorra 155   749       749      UEFA
## 5  5           Angola 121  1117      1117      CAF
## 6  6 Antigua and Barbuda 153   787       787      CONCACAF
## 7  7           Argentina 32  1659      1659      CONMEBOL
## 8  8           Armenia 126  1103      1104      UEFA
## 9  9           Aruba 157   724       724      CONCACAF
## 10 10          Australia 7   1963      1963      AFC
##
## $Men
##   ID          Team Rank Points Prev_Points Confederation
## 1  1 Afghanistan 149  1052      1052      AFC
## 2  2           Albania 66  1356      1356      UEFA
## 3  3           Algeria 35  1482      1482      CAF
## 4  4 American Samoa 192  900       900      OFC
## 5  5           Andorra 135 1082      1082      UEFA

```

```

## 6   6           Angola  124   1136   1136   CAF
## 7   7           Anguilla 210   821    821    CONCACAF
## 8   8 Antigua and Barbuda 126   1127   1127   CONCACAF
## 9   9           Argentina 9    1623   1623   CONMEBOL
## 10 10          Armenia  102   1213   1213   UEFA

```

これでどれが女性ランキングか、男性ランキングかが区別しやすくなりました。ここでは2つのデータフレームを入れましたが、様々なデータ構造が混在したリストも可能です。様々なデータ構造が混在したリストを自分で作成する場面はあまりありません。自分で作成した独自クラスを利用するパッケージを開発する際はよく使いますが、ここでは省略します。ただし、様々なデータ構造が含まれたリスト型を見ることはよくあります。たとえば、`lm()` 関数で回帰分析を行った際、その結果はリスト型であり、中にはベクトル、データフレーム、リストなどが混在しています。これについては今後詳細に解説していきたいと思います。

### 9.5.2 リスト型データの操作

リスト型の中身は何でもあり得るので、なんらかの操作方法があるわけではありません。リスト型の操作というのはリスト内の要素をどのように抽出するかであり、各要素（データフレーム、行列、ベクトルなど）の操作はこれまで説明してきた方法と同じです。したがって、ここではリストを構成する要素を抽出する方法についてのみ解説します。

#### 要素の番号を利用する方法

`List1` は各要素に名前が付いていないため、番号で抽出します。たとえば、あるリストの `i` 番目の要素を抽出するにはリスト名 `[[i]]` で抽出します。`[]` ではなく、`[[ ]]` であることに注意してください。それでは `List1` の 1 番目の要素を抽出してみましょう。

`1 List1[[1]]`

```

##   ID           Team Rank Points Prev_Points Confederation
## 1  1           Albania 75   1325    1316   UEFA
## 2  2           Algeria 85   1271    1271   CAF
## 3  3 American Samoa 133   1030    1030   OFC
## 4  4           Andorra 155   749     749   UEFA

```

```
## 5      Angola  121  1117    1117      CAF
## 6 6 Antigua and Barbuda  153   787     787      CONCACAF
## 7 7 Argentina   32  1659    1659      CONMEBOL
## 8 8 Armenia    126  1103    1104      UEFA
## 9 9 Aruba      157   724     724      CONCACAF
## 10 10 Australia   7  1963    1963      AFC
```

このように List1 内の 1 番目のデータが抽出されました。こちらのデータ構造は何でしょうか。

```
1 class(List1[[1]])  
  
## [1] "data.frame"
```

既に予想したかと思いますが、データフレームとして抽出されました。ここから更に、3 行目のデータを抽出するには、[[]] の後に [3, ] を付けます。

```
1 List1[[1]][3, ]  
  
##   ID      Team Rank Points Prev_Points Confederation
## 3 3 American Samoa  133   1030        1030      OFC
```

先ほどリストの要素の抽出には「[] でなく、[[]] です」と申しましたが、実は [] も使用可能です。やってみましょう。

```
1 List1[1]  
  
## [[1]]  
  
##   ID      Team Rank Points Prev_Points Confederation
## 1 1    Albania  75   1325     1316      UEFA
## 2 2    Algeria  85   1271     1271      CAF
## 3 3 American Samoa  133   1030     1030      OFC
## 4 4    Andorra  155   749      749      UEFA
## 5 5    Angola   121  1117    1117      CAF
## 6 6 Antigua and Barbuda  153   787     787      CONCACAF
## 7 7 Argentina   32  1659    1659      CONMEBOL
## 8 8 Armenia    126  1103    1104      UEFA
```

```
## 9 9             Aruba 157    724      724      CONCACAF
## 10 10            Australia 7 1963    1963      AFC
```

[[]] を使った抽出とあまり変わらないですね。しかし、重要な違いが一つあります。それはデータ構造です。

```
1 class(List1[1])
```

```
## [1] "list"
```

[] で抽出したリストの要素のデータ構造もまたリスト型です。この場合、先ほどのように [x, ] を使って、x 行目のデータを抽出することはできません。なぜなら、[行, 列] による要素の抽出はデータフレームのためのものであって、リスト型のためのものではないからです。

```
1 List1[1][3, ]
```

```
## Error in List1[1][3, ]: incorrect number of dimensions
```

このようにエラーが表示されます。

### 要素の名前を利用する方法

List2 のようにリストの要素に名前を付けた場合、これまでの方法に加え [[要素名]] と \$要素名 を使った操作が可能です<sup>7)</sup>。List2 から男子ランキング (Men) を抽出してみましょう。

```
1 List2[["Men"]]
```

```
##      ID              Team Rank Points Prev_Points Confederation
## 1    1    Afghanistan 149  1052      1052      AFC
## 2    2        Albania  66  1356      1356      UEFA
## 3    3        Algeria  35  1482      1482      CAF
## 4    4 American Samoa 192   900      900      OFC
## 5    5       Andorra 135  1082      1082      UEFA
## 6    6        Angola 124  1136      1136      CAF
```

---

<sup>7)</sup> [[要素名]] で抽出することも可能ですが、この場合、返される結果はリスト型になります。

```
## 7 7 Anguilla 210 821 821 CONCACAF
## 8 8 Antigua and Barbuda 126 1127 1127 CONCACAF
## 9 9 Argentina 9 1623 1623 CONMEBOL
## 10 10 Armenia 102 1213 1213 UEFA
```

```
1 List2$Men
```

```
## ID Team Rank Points Prev_Points Confederation
## 1 1 Afghanistan 149 1052 1052 AFC
## 2 2 Albania 66 1356 1356 UEFA
## 3 3 Algeria 35 1482 1482 CAF
## 4 4 American Samoa 192 900 900 OFC
## 5 5 Andorra 135 1082 1082 UEFA
## 6 6 Angola 124 1136 1136 CAF
## 7 7 Anguilla 210 821 821 CONCACAF
## 8 8 Antigua and Barbuda 126 1127 1127 CONCACAF
## 9 9 Argentina 9 1623 1623 CONMEBOL
## 10 10 Armenia 102 1213 1213 UEFA
```

どれも結果は同じです。また、それぞれのデータ構造もデータフレームです。

```
1 class(List2[["Men"]])
```

```
## [1] "data.frame"
```

```
1 class(List2$Men)
```

```
## [1] "data.frame"
```

したがって、[行番号, 列番号] のようにデータフレームの操作が可能です。Men 要素から 10 行目のデータを抽出してみましょう。

```
1 List2[["Men"]][10, ]
```

```
## ID Team Rank Points Prev_Points Confederation
## 10 10 Armenia 102 1213 1213 UEFA
```

```

1 List2$Men[10, ]

##      ID      Team Rank Points Prev_Points Confederation
## 10 10 Armenia  102    1213        1213        UEFA

```

もちろん、名前を付けた場合であっても、要素の番号を利用した操作も可能です。自分でリスト型を作成する際には各要素に名前を付けた方が分かりやすくて良いでしょう。

## 9.6 配列 (array)

配列は行列の拡張版であり、行列は配列の特殊な形です。これまでの R では行列と配列は区別されてきましたが、R 4.0.0 以降、行列は配列の属するデータ構造となりました。

配列型は簡単にいうと、同じサイズの行列を数枚重ねたものです。図 9.1 は配列型のイメージを表したものです。

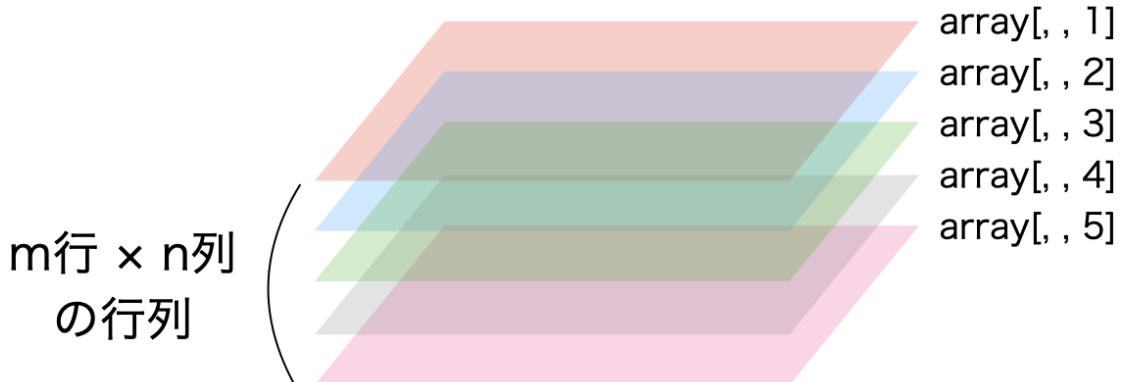


図 9.1: 配列型データのイメージ

したがって、配列型は行と列以外にも、層の要素も持つことになります。複数の行列で構成されている点では、リスト型と類似していますが、配列型は**同じサイズの行列**のみで構成されている点が特徴です。

配列型は普段、扱う機会があまりありませんが、マルコフ連鎖モンテカルロ法 (MCMC)

でベイジアン推定を行った後の事後分布データは配列型で格納される場合があります。

### 9.6.1 配列型データの作成

各行列は 3 行 4 列とし、4 層構造とします。まずは、同じサイズの行列を作成し、それぞれ Mat1、Mat2、Mat3、Mat4 と名付けます。

```
1 Mat1 <- matrix(sample(1:12, 12, replace = TRUE), byrow = TRUE, nrow = 3)
2 Mat2 <- matrix(sample(1:12, 12, replace = TRUE), byrow = TRUE, nrow = 3)
3 Mat3 <- matrix(sample(1:12, 12, replace = TRUE), byrow = TRUE, nrow = 3)
4 Mat4 <- matrix(sample(1:12, 12, replace = TRUE), byrow = TRUE, nrow = 3)
```

`sample()` 関数は初めてですね。これは与えられたベクトルの要素から無作為に要素を抽出する関数です。`sample(1:12, n)` なら `c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)` から無作為に `n` 個の要素を抽出するという意味です。`replace = TRUE` を指定すると、反復抽出、つまり、一回抽出された要素であっても抽出される可能性があることを意味します。デフォルトは `FALSE` ですが、この場合、一旦抽出された要素は二度と抽出されません。それではそれぞれの行列の中身を見てみましょう。

```
1 Mat1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     4    12     7     3
## [2,]     2     9     6     5
## [3,]    11     9     4     3
```

```
1 Mat2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     9     4    11    12
## [2,]     9     4    12     3
## [3,]     6     6    11     5
```

```
1 Mat3
```

```
##      [,1] [,2] [,3] [,4]
```

```

## [1,] 4 5 2 6
## [2,] 7 6 11 10
## [3,] 7 4 9 2
1 Mat4

```

```

## [,1] [,2] [,3] [,4]
## [1,] 5 2 11 7
## [2,] 10 12 9 7
## [3,] 3 5 8 8

```

配列型データを作成するには `array()` 関数を使います。引数としては行列名を `c()` で繋ぎ<sup>8)</sup>、`dim` = で `array` の大きさを指定するだけです。配列を作成した後はそのデータ構造も確認してみましょう。

```

1 Array1 <- array(c(Mat1, Mat2, Mat3, Mat4), dim = c(3, 4, 4))
2 class(Array1)
## [1] "array"

```

`dim = c(m, n, z)` の部分ですが、これは「`m` 行 `n` 列の行列が `z` 枚重ねる」という意味です。今回は 3 行 4 列の行列を 4 枚重ねるので `dim = c(3, 4, 4)` です。それでは `Array1` の中身を見てみます。

```

1 Array1
## , , 1
##
## [,1] [,2] [,3] [,4]
## [1,] 4 12 7 3
## [2,] 2 9 6 5
## [3,] 11 9 4 3
##
## , , 2

```

---

<sup>8)</sup> 元々はベクトルを入力しますが、行列を `c()` で繋ぐと自動的にベクトルに変換されます。

```
##  
##      [,1] [,2] [,3] [,4]  
## [1,]     9     4    11    12  
## [2,]     9     4    12     3  
## [3,]     6     6    11     5  
##  
## , , 3  
##  
##      [,1] [,2] [,3] [,4]  
## [1,]     4     5     2     6  
## [2,]     7     6    11    10  
## [3,]     7     4     9     2  
##  
## , , 4  
##  
##      [,1] [,2] [,3] [,4]  
## [1,]     5     2    11     7  
## [2,]    10    12     9     7  
## [3,]     3     5     8     8
```

このように一つのオブジェクト内に複数の行列が格納されたオブジェクトが生成されます。

### 9.6.2 配列型データの操作

配列型データの操作は行列型に似ていますが、層というもう一つの次元があるため、行列名 [行番号, 列番号] ではなく、配列名 [行番号, 列番号, 層番号] を使います。たとえば、Array1 の 3 番目の行列を抽出したい場合、Array1[, , 3] と入力します。

<sup>1</sup> `Array1[, , 3]`

```
##      [,1] [,2] [,3] [,4]  
## [1,]     4     5     2     6
```

```
## [2,]    7    6   11   10
## [3,]    7    4    9    2
```

また、2番目の行列の3行目・1列目の要素を抽出するなら `Array1[3, 1, 2]` と入力します。

```
1  Array1[3, 1, 2]
```

```
## [1] 6
```

配列型における操作の特徴の一つは全ての行列が同じ大きさを持つため、層を貫通した操作ができるという点です。たとえば、全ての層の2行目を抽出するなら、層番号を指定せず、行番号のみで抽出します。

```
1  Array1[2, , ]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    9    7   10
## [2,]    9    4    6   12
## [3,]    6   12   11    9
## [4,]    5    3   10    7
```

ただ、返された結果は配列型でなく行列型であることに注意しましょう。たとえば、1番目の行列の2行目の要素は2, 9, 6, 5ですが、これは先ほど抽出された行列の1列目に該当します。また、2番目の行列の2行目の要素は9, 4, 12, 3であり、これは先ほどの抽出された行列の2列目となります。全層において特定の一列を抽出しても同じです。これは各層において抽出されたデータが行列でなく、ベクトルだからです。

一方、**複数**の行または列を抽出した場合、結果は配列型です。`Array1` の各層から1・2行目と1・2列目の要素、つまり大きさが $2 \times 2$ の行列を抽出してみましょう。

```
1  Array1[1:2, 1:2, ]
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    4   12
```

```
## [2,] 2 9
##
## , , 2
##
## [,1] [,2]
## [1,] 9 4
## [2,] 9 4
##
## , , 3
##
## [,1] [,2]
## [1,] 4 5
## [2,] 7 6
##
## , , 4
##
## [,1] [,2]
## [1,] 5 2
## [2,] 10 12
```

このように各層において抽出されたデータが行列だからです。自分の操作から得られた結果がどのようなデータ型・データ構造かを予め知っておくことで分析の効率が上がるでしょう。

## 行列に名前を付けたい

今回は各行列に 1, 2, 3, 4 という番号のみ割り当てましたが、実は行列に名前を付けることも可能です。そのためには配列データ作成時、`dimnames` =引数を指定する必要があります。

```
5      )
```

`dimnames` =引数は必ず長さ 3 のリスト型である必要があります。1 番目と 2 番目の要素は行名と列名ですが、行列において一般的に使われないため、ここでは `NULL` にし、3 番目の要素、つまり各層の名前だけを指定します。ここではそれぞれの層を `M1`、`M2`、`M3`、`M4` と名付けました。ここで `M4` のみを抽出する場合は、以下のように操作します。

```
1  Array2[, , "M4"]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     5    2   11    7
## [2,]    10    12    9    7
## [3,]     3    5    8    8
```

もちろん、これまでと同様、番号で抽出することも可能です。

```
1  Array2[, , 4]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]     5    2   11    7
## [2,]    10    12    9    7
## [3,]     3    5    8    8
```

---

## 練習問題

リストと配列は要素の抽出が主な操作となるため、練習問題は掲載しておりません。

### 9.6.3 ベクトル

**問 1** 1 から 10 までの公差 1 の等差数列 (1, 2, 3, ..., 10) を作成し、`myVec1` と名付けよ。

**問 2** `myVec1` の長さを求めよ。

**問 3** `myVec1` から偶数のみを抽出せよ

問 4 `myVec1` を `myVec2` という名でコピーし、`myVec2` の偶数を全て 0 に置換せよ。

問 5 `myVec1` の全要素から 1 を引し、`myVec3` と名付けよ。

問 6 `myVec1` の奇数番目の要素には 1 を、偶数番目の要素には 2 を足し、`myVec4` と名付けよ。

問 7 `myVec4` から `myVec1` を引け。

#### 9.6.4 行列

問 1 以下のような 2 つの行列を作成し、それぞれ `myMat1`、`myMat2` と名付けよ。

$$\text{myMat1} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \text{myMat2} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

問 2 `myMat1` と `myMat2` の掛け算を行い、`myMat3` と名付けよ。掛け算はアダマール積でなく、一般的な行列間の積を求めること。

問 3 以下のような連立方程式を解くケースを考える。

$$3x - y + 2z = 12, \quad (9.9)$$

$$x + 2y + 3z = 11, \quad (9.10)$$

$$2x - y - z = 2. \quad (9.11)$$

1. 以下のような 2 つの行列を作成し、それぞれ `myMat4`、`myMat5` と名付けよ。

$$\text{myMat4} = \begin{bmatrix} 3 & -1 & 2 \\ 1 & 2 & 3 \\ 2 & -1 & -1 \end{bmatrix}, \text{myMat5} = \begin{bmatrix} 12 \\ 11 \\ 2 \end{bmatrix}$$

2. `myMat4` の逆行列を求めよ。

3. `myMat4` の逆行列と `myMat5` の積を求め、`myMat6` と名付けよ。

- これが連立方程式の解である。

4. `myMat4` と `myMat6` の積を求めよ。

### 9.6.5 データフレーム

問1 以下のようなデータフレームを作成し、myDF1 と名付けよ<sup>9)</sup>。

ID	Name	Rank	Socre
1	Japan	28	1500
2	Iran	33	1489
3	South Korea	40	1464
4	Australia	42	1457
5	Qatar	55	1396
6	Saudi Arabia	67	1351
7	Iraq	70	1344
8	UAE	71	1334
9	China	76	1323
10	Syria	79	1314

問2 myDF1 から Name 列を抽出せよ。

問3 myDF1 の Name 列から 3 番目の要素を抽出せよ。

問4 myDF1 の 3 行目を抽出せよ。

問5 FIFA\_Women.csv を tibble 型として読み込み、myTbl1 と名付けよ<sup>10)</sup>。

問6 myTbl1 の Rank 列を抽出し、それぞれの要素が 20 より小さいかを判定せよ。

問7 myTbl1 の Rank が 20 より小さい国名を抽出せよ。

<sup>9)</sup> 2020 年 4 月 9 日現在の FIFA 男子サッカーランキングである。データは AFC (アジアサッカー連盟) 所属の上位 10 カ国である。データ源は (<https://www.fifa.com/fifa-world-ranking/ranking-table/men/rank/id12882/>) である (アクセス日: 2020 年 4 月 11 日)。

<sup>10)</sup> 2020 年 3 月 27 日現在の FIFA 女子サッカーランキングである。データ源は ([www.fifa.com/fifa-world-ranking/ranking-table/women/](https://www.fifa.com/fifa-world-ranking/ranking-table/women/)) である (アクセス日: 2020 年 4 月 11 日)。それぞれの変数は ID: 国名の ID (アルファベット順)、Team: チーム名、Rank: FIFA ランキング、Points: 2020 年 3 月 27 日のポイント、Prev\_Points: 2019 年 12 月 13 日のポイント、Confederation: 所属しているサッカー連盟である。

問 8 myTbl1 からランキングが 20 より上位の行を抽出せよ。

この章で使うパッケージを読み込む。

```
1 pacman::p_load(tidyverse)
```



## 第 10 章

# R プログラミングの基礎

この章では統計ソフトウェアではなく、プログラミング言語としての R について解説する。プログラミングは難しいというイメージがあるかもしれないが、実は難しい。しかし、プログラミングをする場合にまず覚えるべき重要概念は、

1. データの読み書き<sup>1)</sup>
2. 条件分岐
3. 反復

の 3 つだけだ<sup>2)</sup>。この 3 つだけでほとんどのプログラムが作れる。ただ、この単純さがプログラミングの難しさもあるとも言える。

たとえば、ある数字の列を小さい順（昇順）に並び替えることを考えよう。`c(6, 3, 7, 2, 5, 1, 8, 4)` という数列が与えられたとき、人間ならあまり苦労することなく、`c(1, 2, 3, 4, 5, 6, 7, 8)` に並び替えられるだろう。しかし、その手順を「代入」、「条件分岐」、「反復」のみで明確に表現できるだろうか<sup>3)</sup>。もちろん、できる。よって、並べ替えはプログラミングによって実現することができる。R には、数字を並べ替えるた

<sup>1)</sup> 具体的にはメモリの特定箇所にデータを書き込んだり、それを読み取んだりすること。

<sup>2)</sup> チューリング完全 (Turing-complete) な言語の条件に「反復」はない。条件分岐でループを実行することができるからだ。よって、プログラミングを構成するのは「データの読み書き」と「条件分岐」のみとも考えられる。

<sup>3)</sup> 並べ替え（ソート）アルゴリズムはプログラミングを学習する際に定番の題材であり、様々なアルゴリズムがある。

めの `sort()` 関数や `order()` 関数などが用意されており、それらの組込関数を使えば良いのだが、組込関数は「代入」、「条件分岐」、「反復」を組み合わせて作られたものだ。

「代入」、「条件分岐」、「反復」という3つの概念さえ理解すれば何でもできるるという意味で、プログラミングは簡単だ。しかし、この3つだけで解決しないといけないという意味で、プログラミングは難しい。頻繁に利用する機能については、ほとんどのプログラミング言語があらかじめ作られた関数 (built-in function, 組込関数) を数多く提供しており、ユーザである私たちは組込関数を使うのが賢明である。それでも条件分岐や反復について勉強する必要があるのは、私たち一般ユーザにとってもこれが必要な場面が多いからである。たとえば、同じ分析をデータだけ変えながら複数回実行するために反復を利用する。反復をうまく使えば、コードの量を数十分の一に減らすことも可能である。また、特定の条件に応じてデータを分類するときは条件分岐を使う。条件分岐を使いこなせないと、Calc や Excel などのスプレッドシートでデータを目でみながら値を入力するという苦行を強いられるが<sup>4)</sup>、条件分岐が使えばそのような作業は必要ない。数行のプログラミングをするだけで、ほんの数秒で分類が終わる。

## 10.1 R 言語の基礎概念

### 10.1.1 オブジェクト

これまで「オブジェクト」や「関数」などという概念を定義せずに使ってきたが、ここでは R 言語の構成する基本的な概念についてもう少し詳細に解説する。これらは、プログラミングをする上である程度は意識的に使う必要があるものである。

まず、**オブジェクト (object)** とはメモリに割り当てられた「何か」である。「何か」に該当するのは、ベクトル (vector)、行列 (matrix)、データフレーム (data frame)、リスト (list)、関数 (function) などである。一般的に、オブジェクトにはそれぞれ固有の（つまり、他のオブジェクトと重複しない）名前が付いている。

たとえば、1 から 5 までの自然数の数列を

---

<sup>4)</sup> 苦行であるだけでなく、スプレッドシート上でデータを書き換えると、入力ミスをおかしやすいし、そのミスを後で見つけるのが非常に困難である。したがって、データをスプレッドシート上で編集するのには絶対にやめるべきである。

```
1 my_vec1 <- c(1, 2, 3, 4, 5) # my_vec1 <- 1:5 でも同じ
```

のように `my_vec1` という名前のオブジェクトに格納する。オブジェクトに名前をつけてメモリに割り当てると、その後 `my_vec1` と入力するだけでそのオブジェクトの中身を読み込むことができるようになる。

ここで、次のように `my_vec1` の要素を 2 倍にする操作を考えてみよう。

```
1 my_vec1 * 2
```

```
## [1] 2 4 6 8 10
```

`my_vec1` は、先ほど定義したオブジェクトである。では `2` はどうだろうか。`2` はメモリに割り当てられていないので、オブジェクトではないのだろうか。実は、この数字 `2` もオブジェクトである。計算する瞬間のみ `2` がメモリに割り当てられ、計算が終わったらメモリから消されると考えれば良い。「R に存在するあらゆるものはオブジェクトである (Everything that exists in R is an object)」(Chambers [2016, p.4])。`*`のような演算子さえもオブジェクトである。

## 10.1.2 クラス

**クラス (class)** とはオブジェクトを特徴づける属性である。既に何度か `class()` 関数を使ってデータ型やデータ構造を確認したが、`class()` 関数でオブジェクトのクラスを確認することができる。先ほど、`my_vec1` も `*` も `2` もオブジェクトであると説明した。これらがすべてオブジェクトであるということは、何らかのクラス属性を持っているということである。また、`class()` 関数そのものもオブジェクトなので、何らかのクラスをもつ。確認してみよう。

```
1 class(my_vec1)
```

```
## [1] "numeric"
```

```
1 class(`*`)
```

```
## [1] "function"
```

```
1  class(2)  
  
## [1] "numeric"  
  
1  class(class)  
  
## [1] "function"
```

統計分析をする際に、R のクラスを意識することはあまりない。しかし、R でパッケージを開発したり、複雑な関数を自作する場合、オブジェクト指向プログラミング (Object-oriented Programming; OOP) の考え方方が重要で、オブジェクトのクラスを厳密に定義する必要がある<sup>5)</sup>。

自分でパッケージを開発しないとしても、クラスの概念は知っておいたほうが良い。この後説明する関数には引数というものがあるが、引数には特定のクラスのオブジェクトしか指定できない。関数の使い方がわからないときには?関数名 でヘルプを表示するが、ヘルプには各引数に使うべきクラスが書かれている。クラスについて何も知らないと、ヘルプを読んでも意味がわからないかもしれない。

たとえば、R コンソール上で?mean を入力して実行すると、図 10.1 のような mean() 関数のヘルプが表示される。ヘルプに示されているとおり、mean() 関数に必要な引数は x、trim、na.rm である。仮引数 x の説明を読むと、x の実引数には numeric または logical のベクトルクラスが使えることが分かる。さらに、date、date-time、time interval が使えることも教えてくれる。

---

<sup>5)</sup> R は OOP を実装するために S3 クラスをデフォルトで採用しているが、これはオブジェクト指向プログラミング言語としては厳密性を欠くところがある。R では、S3 の代わりに S4 や R6 などの現代的な OOP を実装することもできる。

```
mean {base}                                         R Documentation

Arithmetic Mean

Description
Generic function for the (trimmed) arithmetic mean.

Usage
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments
x
  An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.

trim
  the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

na.rm
  a logical value indicating whether NA values should be stripped before the computation proceeds.
```

図 10.1: ‘mean()’関数のヘルプ画面

回帰分析を行う `lm()` 関数の場合、`data` という仮引数があるが、`data` の実引数に指定できるのは `data.frame` クラスまたは `tibble` クラスのオブジェクトである<sup>6)</sup>。通常はクラスを意識せずに R を使っても問題ないが、

1. 全てのオブジェクトにはクラスが付与されており、
2. 関数 (の引数) ごとに実引数として使えるクラスが異なる

という 2 点は覚えておこう。

### 10.1.3 関数と引数

**関数 (function)** は、入力されたデータを内部で決められた手順に従って処理し、その結果を返すものである。「R で起こるあらゆることは関数の呼び出しである (Everything that happens in R is a function call)」(Chambers [2016, p.4])。

関数は 関数名 (関数の入力となるオブジェクト) のように使う。たとえば、`class(my_vec1)` は `my_vec1` というオブジェクトのクラスを返す関数である。また、

---

<sup>6)</sup> 厳密には、`tibble` 型データは 3 つのクラスを内部に有しており、その中に `data.frame` が含まれる。

`sum(my_vec1)` は `my_vec1` の要素の総和を計算して返す関数である。

関数は自分で作成することもできる（次章で説明する）。複雑な作業を繰り返す場合、その作業を関数として記述することで、一行でその作業を再現することが可能になる。「2度以上同じことを繰り返すなら関数を作れ」というのが、R ユーザの心得である。

関数を使うためには、**引数（ひきすう）** と呼ばれるものが必要である。例えば、`sum()` 関数はこれだけだと何もできない。何らかのオブジェクトがが入力として与えられないと、結果を返すことができない。`sum(my_vec1)` のようにすることではじめて結果が返される。ここで `my_vec1` が `sum()` 関数の引数である。

関数は引数は複数持つことができる。たとえば、欠測値を含む以下の `my_vec2` を考えてみよう。

```
1 my_vec2 <- c(1, 2, 3, NA, 5)
```

この数列の総和を `sum()` 関数で求めようとすると、結果は欠測値 `NA` になる。

```
1 sum(my_vec2)
```

```
## [1] NA
```

これは `sum()` の基本仕様が「入力に欠測値が含まれている場合は欠測値を返す」ことになっているためである。入力されたデータのなかで欠測値を除いたものの総和を求めために、`sum()` はもう 1 つの引数が用意されている。それが `na.rm` である。`na.rm = TRUE` を指定すると、欠測値を除外した総和を返す。

```
1 sum(my_vec2, na.rm = TRUE)
```

```
## [1] 11
```

第 6 章で説明したとおり、`na.rm` などのように引数を区別するために関数によって用意されたものを**仮引数（parameter）** と呼び、`TRUE` のように引数の中身としてユーザが指定するものを**実引数（argument）** と呼ぶ。`my_vec2` は実引数である。

引数を指定するとき、`my_vec2` のように仮引数を明示しないこともある。多くの場合、それがなければ関数がそもそも動かないという第 1 引数の仮引数は明示しないことが多い。そもそも、第 1 引数には仮引数がない（名前がない）場合もある。`sum()` 関数の第 1

引数は numeric または complex 型のベクトルだが、仮引数がそもそもない（... で定義されている）。

しかし、第 1 引数以外については、`na.rm = TRUE` のように仮引数を明示すべきである。関数によっては数十個の引数をとるものもあり、仮引数を明示しないと、どの実引数がどの仮引数に対応するのかわかりにくい。

多くの引数は関数によって定義された既定値 (default value) をもっている。たとえば、`sum()` 関数の `na.rm` の既定値は `FALSE` である。既定値が用意されている場合、その引数を指定せずに関数を使うことができる。

ある関数がどのような引数を要求しているか、その既定値は何か、引数として何か決められたデータ型/データ構造があるかを調べたいときは `?関数名` (()) がないことに注意) または `help(関数名)` をコンソールに入力する。多くの関数に詳細なヘルプが付いているので、ネットで検索したり、誰かに質問する前にひとまずヘルプを読むべきである。

---

## 10.2 R のコーディングスタイル

R コードの書き方に唯一の正解はない。文法が正しければ、つまり、R が意図どおりの実行結果を出してくれさえすれば、それは「正しい」コードである。しかし、コードというのは、一度書けば二度と読まないというものではない。最初に「書く」とき以外に、「修正する」ときや「再利用」するときなどに繰り返し読むことになる。また、共同研究をする場合には共同研究者がコードを読む。さらに、近年では研究成果を報告する際に分析に利用したコードを後悔公開することも当たり前になりつつあり、その場合には自分で書いたコードを世界中の人人が読む可能性がある。

ここで大事なのは、R コードを読むのは R だけではないということである。人間も重要な「読者」である。R コードを書くときは人間に優しいコードを書くように心がえよう。唯一の正解がなくても、読みやすく、多くの人が採用している標準的な書き方はある。ここでは、人間に優しいコードを書くためのコーディングスタイルについて説明しよう。

### 10.2.1 オブジェクト名

ベクトル、データフレーム、自作の関数などのすべてのオブジェクトには名前をつける必要がある（ラムダ式などの無名関数を除く）。名前はある程度自由に付けることができるが、大事な原則がある。それは、

- オブジェクト名は英数字と限られた記号のみにする
- 数字で始まる変数名は避ける

1つ目は原則にすぎない。よって、日本語やハングルの変数名をつけることもできる。しかし、それは推奨しない。英数字以外（マルチバイト文字）は文字化けの可能性がある（文字コードの問題が発生する）し、コードを書く際列がずれる原因にもなる

```
1 var1 <- c(2, 3, 5, 7, 11)      # 推奨

1 变数1 <- c(2, 3, 5, 7, 11)      # 非推奨

1 변수1 <- c(2, 3, 5, 7, 11)      # 非推奨

1 var1

## [1]  2  3  5  7 11

1 变数1

## [1]  2  3  5  7 11

1 변수1
```

2つ目のルールは必ず守る必要がある。つまり、数字で始まるオブジェクト名は作成できない。

```
1 100A <- "R"

## Error: <text>:1:4: unexpected symbol
## 1: 100A
```

```
## ^
```

### 予約語を避ける

R が提供する組込の関数やオブジェクトと重複する名前を自分で作成するオブジェクトに付けるのは避けよう。例えば、R には円周率 ( $\pi$ ) が `pi` という名前で用意されている。

```
1 pi
```

```
## [1] 3.141593
```

`pi` という名前で新しい変数を作ることはできる。

```
1 pi <- 777
```

しかし、既存の `pi` が上書きされてしまうので避けたほうが良い。

```
1 pi # もはや円周率ではない
```

```
## [1] 777
```

元の円周率を使うこともできるが、手間が増える。

```
1 base::pi
```

```
## [1] 3.141593
```

また、ユーザが自由に使えない名前もある。それらの名前を「予約語」と呼ぶ。予約語の使用は禁止されている。

```
1 if <- "YY"
```

```
## Error: <text>:1:5: unexpected assignment
```

```
## 1: if <-
```

```
## ^
```

```
1 for <- "JS"
```

```
## Error: <text>:1:5: unexpected assignment
```

```
## 1: for <-
```

```
## ^  
1 TRUE <- "いつもひとつ!"  
## Error in TRUE <- "いつもひとつ!": invalid (do_set) left-hand side to assignment
```

このように、予約語はそもそも使えないで、それほど意識する必要はない。

ただし、以下のコードのようなことが起こるので注意してほしい。

```
1 vals <- 1:5  
2 vals[c(TRUE, TRUE, FALSE, FALSE, TRUE)]  
## [1] 1 2 5  
1 vals[c(T, T, F, F, T)]  
## [1] 1 2 5
```

ここから、T は TRUE、F は FALSE と同じ働きをしていることがわかる。ここで、次のコードを実行してみよう。

```
1 T <- "Taylor"  
2 F <- "Fourier"  
3 vals[c(T, T, F, F, T)]  
## [1] NA NA NA NA NA
```

このように、T と F はあらかじめ使える状態で用意されているものの、予約語ではないので値が代入できてしまう。しかし、T と F を自分で定義したオブジェクトの名前に使うと混乱の元になるので、使用は避けるのが無難である。

また、TRUE と FALSE を T や F で済ませる悪習は廃して常に完全にスペルすべきである。

### 短さと分かりやすさを重視する

オブジェクト名を見るだけでその中にどのようなデータが含まれているか推測できる名前をつけるべきである。たとえば、性別を表す変数を作るなら、

```
1 var2 <- c("female", "male", "male", "female")
```

ではなく、

```
1 gender <- c("female", "male", "male", "female")
```

としたほうが良い。gender という名前がついていれば、「この変数には性別に関する情報が入っているだろう」と容易に想像できる。

また、オブジェクト名は、短くて読みやすいほうが良い。数学の成績データをもつ次のオブジェクトについて考えよう。

```
1 mathematicscore <- c(30, 91, 43, 77, 100)
```

オブジェクト名から、中にどのような情報が含まれるか想像することはできる。しかし、この名前は長く、読みにくい。そこで、次のように名前を変えたほうが良い。

```
1 MathScore <- c(30, 91, 43, 77, 100)
2 mathScore <- c(30, 91, 43, 77, 100)
3 math_score <- c(30, 91, 43, 77, 100)
```

これらの例では、mathematics を math に縮め、math と score の間に区切りを入れて読みやすくしている。大文字と小文字の組み合わせで区切る方法は**キャメルケース (camel case)** と呼ばれ、大文字から始まるキャメルケースを**大文字キャメルケース (upper camel case)**、小文字から始まるキャメルケースを**小文字キャメルケース (lower camel case)** と呼ぶ。また、\_ (アンダーバー, アンスコ) で区切る方法は**スネークケース (snake case)** と呼ばれる。キャメルケースとスネークケースは、一貫した方法で使えばどちらを使っても良いだろう。

かつては “.” を使って単語を繋ぐのが標準的だった時代もあり、組込関数には “.” を使ったものも多い。data.frame() や read.csv() などがその例である。しかし、“.” はクラスのメソッドとして使われることがあり、混乱するので使うのは避けたほうが良い。tidyverse\* では、readr::read\_csv() のように、スネークケースが採用されている。他にも - (ハイフン) で区切るチェーンケース (chain case)\*\* というのもあるが、R で-

は「マイナス（減算演算子）」であり、チェーンケースは使えない<sup>7)</sup>。

### 10.2.2 改行

コードは1行が長すぎないように適宜改行する。Rやパッケージなどが提供している関数のなかには10個以上の引数を必要とするものもあり。コードを1行で書こうとするとコードの可読性が著しく低くなってしまう。

1行に何文字入れるべきかについて決まったルールはないが、1行の最大文字数を半角80字にするという基準が伝統的に使われてきた。これま昔のパソコンで使ったパンチカード（図10.2）では1行に80個の穴を開けることができたことに由来する。

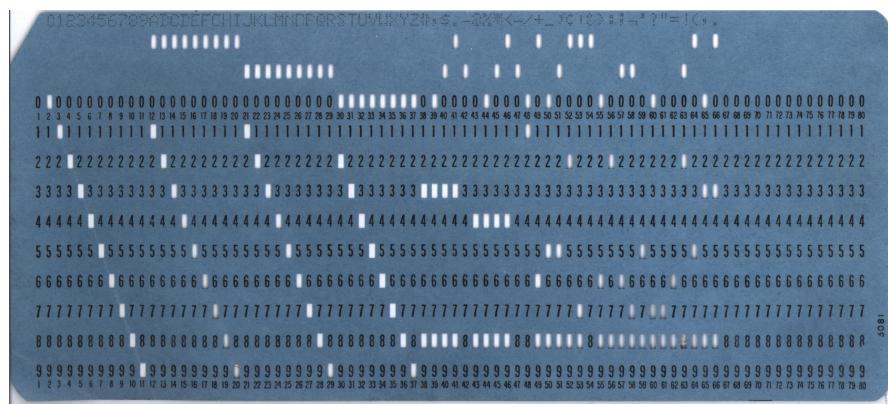


図10.2: パンチカードの例

最近は昔と比べてモニタのサイズが大きく、解像度も高いので、80文字にこだわる必要はない。自分のRStudioのSource Paneに収まるよう、切りがいいところで改行しよう。

### 10.2.3 スペースとインデント

適切なスペースはコードの可読性を向上させる。以下の2つのコードは同じ内容を実行するが、後者にはスペースがないので読にくい。

<sup>7)</sup> チェーンケースは、COBOLやLISPのような言語では使われる。どうしても使いたければ、変数名全体をバックティック (`) で囲んで `math-score` <- 10 のようにする方法もあるが、もちろん推奨しない。

```
1 # 良い例
2 sum(my_vec2, na.rm = TRUE)
3
4 # 悪い例
5 sum(my_vec2,na.rm=TRUE)
```

どこにスペースを入れるかについてのルールは特になり。R は「半角スペース」を無視するので、プログラムの動作に関して言えば、半角スペースはあってもなくても同じである。標準的なルールとして、「, の後にスペース」、「演算子の前後にスペース」などが考えらえる。ただし、^の前後にはスペースを入れないことが多い。また、後ほど紹介する `for(){}、while(){}、if(){}などのように、関数以外の () の前後にはスペースを入れる。それに対し、sum() や read.csv() などのように、関数のかっここの場合には、関数名と () の間にスペースを入れない。`

また、スペースを 2 回以上入れることもある。たとえば、あるデータフレームを作る例を考えよう。

```
1 # 良い例
2 data.frame(
3   name      = c("Song", "Yanai", "Wickham"),
4   favorite  = c("Ramen", "Cat", "R"),
5   gender    = c("Male", "Male", "Male")
6 )
7
8 # 悪い例
9 data.frame(
10  name = c("Song", "Yanai", "Hadley"),
11  favorite = c("Ramen", "Cat", "R"),
12  gender = c("Male", "Male", "Male")
13 )
```

上の 2 つのコードの内容は同じだが、前者のほうが読みやすい。

ここでもう 1 つ注目してほしいのは、「字下げ (indent)」の使い方である。上のコード

は次のように一行にまとめることができるが、読みにくい。

```
1 # 邪悪な例
2 data.frame(name=c("Song", "Yanai", "Hadley"), favorite=c("Ramen", "Cat", "R"), fender=c(
```

このように1行のコードが長い場合、「改行」が重要である。しかし、Rの最も基本的なルールは「1行に1つのコード」なので、改行するとこのルールを破ることになってしまう。そこで、コードの途中で改行するときは、2行目以降を字下げすることで、1つのコードが前の行から続いていることを明確にしよう。前の行の続きの行は2文字（または4文字）分字下げする。こうすることで、「この行は上の行の続き」ということが「見て」わかるようになる。

```
1 # 良い例
2 data.frame(
3   name      = c("Song", "Yanai", "Hadley"),
4   favorite  = c("Ramen", "Cat", "R"),
5   gender    = c("Male", "Male", "Male")
6 )
7
8 # 悪い例
9 data.frame(
10  name     = c("Song", "Yanai", "Hadley"),
11  favorite = c("Ramen", "Cat", "R"),
12  gender   = c("Male", "Male", "Male")
13 )
```

RStudioは自動的に字下げをしてくれるので、RStudioを使っているならあまり意識する必要はない。自動で字下げしてくれないエディタを使っている場合は、字下げに気をつけよう。

#### 10.2.4 代入

オブジェクトに値を代入する演算子として、これまで `<-` を使ってきましたが、`=`を使うこともできる。R以外の多くのプログラミング言語では、代入演算子として`=`を採用している。

しかし、引数の指定と代入を区別するためん、本書では`<-`の使用を推奨する。また、既に説明したとおり、`option + -` (macOS の場合) または `Alt + -` (Windows の場合) というショートカットを使うと、`<-` だけでなく、その前後のスペースを自動的に挿入してくれる。コードが読みやすくなるので、代入演算子は常にショートカットで入力する習慣をつけよう。

この節で説明したコードの書き方は、コーディングスタイルの一部に過ぎない。本書では、できるだけ読みやすいコードを例として示すことを心がけている。最初は本書（あるいは他の本やウェブサイトに掲載されているものなかで気に入ったもの）のスタイルを真似して書いてみてほしい。コードを書き写しているうちに、自にとって最善の書き方が見えてくるだろう。

コーディングスタイルについては、以下の 2 つの資料がさらに詳しい。とりわけ、羽鳥先生が書いた `The tidyverse style guide` は事実上の業界標準であり、`Google's Style Guide` もこのガイドをベースにしている。かなりの分量だが、パッケージ開発などを考えているなら一度は目を通しておいたほうが良いだろう。

- [The tidyverse style guide](#)
  - [Google's Style Guide](#)
- 

## 10.3 反復

人間があまり得意ではないが、コンピュータが得意とする代表的な作業が「反復作業」である。普通の人間なら数時間から数年かかるような退屈な反復作業でも、コンピュータを使えば数秒で終わることが多い。R でもさまざまな反復作業を行うことができる。

反復作業を行うときは、反復作業を「いつ終わらせるか」を考えなくてはならない。終わりを規程する方法として、以下の 2 つが考えられる。

1. 処理を繰り返し回数を指定し、その回数に達したら終了する: `for` 文を使う
2. 一定の条件を指定し、その条件が満たされたときに処理を終了する: `while` 文を使う

この節では、これら 2 つのケースのそれぞれについて解説する。

### 10.3.1 for による反復

for を利用した反復を、**for ループ** と呼ぶ。まず、for ループの雛形をみてみよう。

```
1 for (任意の変数 in ベクトル) {  
2   処理内容  
3 }
```

任意の変数の選び方はいろいろ考えられるが、よく使うのはインデックス (index) *i* である。このインデックスは for ループの内部で使うために用いられる変数である。そして、in の後の「ベクトル」には長さ 1 以上のベクトルを指定する。ベクトルは必ずしも numeric 型である必要はないが、インデクスを利用した for ループではインデクスを指定する正の整数を要素にもつベクトルを使う。

また、{}内の内容が 1 行のみの場合には、{}は省略しても良い。その場合、処理内容を () の直後に書。つまり、以下のような書き方もできる。

```
1 for (任意の変数 in ベクトル) 処理内容
```

これは for だけでなく、if や function() など、{}で処理内容を囲む関数に共通である。処理内容が 2 行以上の場合は、必ず{}で囲むようにしよう。

例として、インデクス *i* の内容を画面に表示することを N 回繰り返す for ループを書いてみよう。繰り返し回数を指定するために、for (i in 1:N) と書く。5 回繰り返すなら for (i in 1:5) とする。1:5 は c(1, 2, 3, 4, 5) と同じなので、for (i in c(1, 2, 3, 4, 5)) でもいいが、1:5 を使って書いたほうが、コードが読みやすい。

以下コードを作成し、実行してみよう。このコードは 3 行がひとつのかたまりになったコードである。for の行（あるいは最後の}の行）にカーソルを置いた状態で Cmd/Ctrl + Return/Enter を押せば、for ループ全体が一挙に実行される。

```
1 # 以下のコードはこのように一行でまとめることも可能  
2 # for(i in 1:5) print(i)  
3 for (i in 1:5) {
```

```
4     print(i)  
5 }
```

```
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5
```

1 から 5 までの数字が表示された。

ここで、この for ループがどのような順番で処理を実行したのか確認してみよう。上のコードは、次のようなステップを踏んで 1 から 5 までの数字を画面に表示している。

1. `i` にベクトル `1:5` の最初の要素を代入 (`i <- 1`)
2. `print(i)` を実行
3. `{}` 中身の処理が終わったら `i` にベクトル `1:5` 内の次の要素を代入 (`i <- 2`)
4. `print(i)` を実行
5. `{}` 中身の処理が終わったら `i` にベクトル `1:5` 内の次の要素を代入 (`i <- 3`)
6. `print(i)` を実行
7. `{}` 中身の処理が終わったら `i` にベクトル `1:5` 内の次の要素を代入 (`i <- 4`)
8. `print(i)` を実行
9. `{}` 中身の処理が終わったら `i` にベクトル `1:5` 内の次の要素を代入 (`i <- 5`)
10. `print(i)` を実行
11. `{}` 中身の処理が終わったら `i` にベクトル `1:5` 内の次の要素を代入するが、5 が最後の要素なので反復終了

この手順を要約すると、次のようになる。

- 任意の変数 (ここでは `i`) にベクトル (ここでは `1:5`) の最初の要素が格納され、`{}` 内の処理を行う。
- `{}` 内の処理が終わったら、ベクトル (ここでは `1:5`) の次の要素を任意の変数 (ここでは `i`) に格納し、`{}` 内の処理を行う。
- 格納できる要素がなくなったら反復を終了する。
  - したがって、反復はベクトル (ここでは `1:5`) の長さだけ実行される (ここで

は5回)。

反復回数はベクトルの長さで決まる。既に述べたとおり、1:5のような書き方でなく、普通のベクトルを指定しても問題ない。たとえば、長さ6のdmg\_valsというベクトルを作り、その要素を出力するコードを書いてみよう。

```

1  dmg_vals <- c(24, 64, 31, 46, 81, 102)
2
3  for (damage in dmg_vals) {
4      x <- paste0("トンヌラに", damage, "のダメージ!!!")
5      print(x)
6  }

## [1] "トンヌラに 24 のダメージ!!!"
## [1] "トンヌラに 64 のダメージ!!!"
## [1] "トンヌラに 31 のダメージ!!!"
## [1] "トンヌラに 46 のダメージ!!!"
## [1] "トンヌラに 81 のダメージ!!!"
## [1] "トンヌラに 102 のダメージ!!!"

```

スライムくらいなら一撃で撃破できそうな立派な勇者である。

ちなみに、paste0()は引数を空白なし<sup>8)</sup>で繋いだ文字列を返す関数である。詳細は第16で解説するが、簡単な例だけ示しておこう。

```

1  paste0("R は", "みんなの", "ともだち")

## [1] "R はみんなのともだち"

1  paste0("私の", "HP/MP は", 500, "/", 400, "です。")

## [1] "私の HP/MP は 500/400 です。"

```

文字列のベクトルを使ったforループもできる。10個の都市名が格納されたcitiesの要素を1つずつ出力するコードは次のように書ける。

---

<sup>8)</sup> paste()関数を使うと、sepで指定した文字列で繋ぐ。sepの既定値は半角スペース。

```
1 cities <- c("Sapporo", "Sendai", "Tokyo", "Yokohama", "Nagoya",
2           "Kyoto", "Osaka", "Kobe", "Hiroshima", "Fukuoka")
3 for (i in seq_along(cities)) {
4   x <- paste0("現在、city の値は", cities[i], "です。")
5   print(x)
6 }
```

```
## [1] "現在、city の値は Sapporo です。"
## [1] "現在、city の値は Sendai です。"
## [1] "現在、city の値は Tokyo です。"
## [1] "現在、city の値は Yokohama です。"
## [1] "現在、city の値は Nagoya です。"
## [1] "現在、city の値は Kyoto です。"
## [1] "現在、city の値は Osaka です。"
## [1] "現在、city の値は Kobe です。"
## [1] "現在、city の値は Hiroshima です。"
## [1] "現在、city の値は Fukuoka です。"
```

ここでは `for (i in 1:10)` ではなく、`for (i in seq_along(cities))` と表記。この例からわかるように、`seq_along()` を使うと、ベクトルのインデクス自動的に作ってくれる。上の例では、`seq_along(ten_cities)` が自動的にベクトル長さを計算し、`1:length(ten_cities)` すなわち `1:10` と同じ処理をしてくれる。ベクトルの長さが自明でないとき（ベクトルが非常に長いとき）には、`seq_along()` を使うのが便利である。後で `cities` の中身を書き換えたときに、`cities` の長さが変わることも考えられるので、ベクトルのインデクス指定には `seq_along()` を使おう。

また、インデクスを使わない `for` ループも書ける。上と同じ結果は、次のコードで実現することができる。

```
1 for (city in cities) {
2   x <- paste0("現在、city の値は", city, "です。")
3   print(x)
4 }
```

```
## [1] "現在、city の値は Sapporo です。"  
## [1] "現在、city の値は Sendai です。"  
## [1] "現在、city の値は Tokyo です。"  
## [1] "現在、city の値は Yokohama です。"  
## [1] "現在、city の値は Nagoya です。"  
## [1] "現在、city の値は Kyoto です。"  
## [1] "現在、city の値は Osaka です。"  
## [1] "現在、city の値は Kobe です。"  
## [1] "現在、city の値は Hiroshima です。"  
## [1] "現在、city の値は Fukuoka です。"
```

このように、ベクトルの中身が数字でない場合でも、for ループでベクトルの要素を 1 つずつ順番に利用することができる。

次に、"1 番目の都市名は Sapporo です"、"2 番目の都市名は Sendai です"、... のように出力する方法を考えよう。これまで出力する文字列のなかで 1箇所（都市名）のみを変えながら表示したが、今回は「i」と「city」の 2 箇所を同時に変える。これは、インデクス i を使って反復を実行しながら、cities の i 番目要素を呼び出すことで実現できる。以下のコードを実行してみよう。

```
1 for (i in seq_along(cities)) {  
2   msg <- paste0(i, "番目の都市名は", cities[i], "です。")  
3   print(msg)  
4 }  
  
## [1] "1 番目の都市名は Sapporo です。"  
## [1] "2 番目の都市名は Sendai です。"  
## [1] "3 番目の都市名は Tokyo です。"  
## [1] "4 番目の都市名は Yokohama です。"  
## [1] "5 番目の都市名は Nagoya です。"  
## [1] "6 番目の都市名は Kyoto です。"  
## [1] "7 番目の都市名は Osaka です。"  
## [1] "8 番目の都市名は Kobe です。"  
## [1] "9 番目の都市名は Hiroshima です。"
```

```
## [1] "10 番目の都市名は Fukuoka です。"
```

このように、インデクスとそれに対応するベクトルの要素を抽出すれば、望みどおりの処理ができる。

### 多重 for ループ

for ループの中でさらに for ループを使うともできる。最初はやや難しいかもしれないが、多重反復はプログラミング技術としてよく使われる所以<sup>9)</sup>、この機会に勉強しよう。

多重 for ループを理解するためにうってつけの例は掛け算九九である。積算演算子 \* の「左の数」と「右の数」のそれぞれに 1 から 9 までの数字を代入するすべての組み合わせを考えるために、2 つの for ループが必要だ<sup>10)</sup>。i \* j で i と j それぞれに 1 から 9 を代入しながら結果を出力するコードは次のように書ける。

```
1 for (i in 1:9) {      #
2   for (j in 1:9) {
3     print(paste(i, "*" , j, "=" , i * j))
4   }
5 }
```

```
## [1] "1 * 1 = 1"
## [1] "1 * 2 = 2"
## [1] "1 * 3 = 3"
## [1] "1 * 4 = 4"
## [1] "1 * 5 = 5"
## [1] "1 * 6 = 6"
## [1] "1 * 7 = 7"
## [1] "1 * 8 = 8"
## [1] "1 * 9 = 9"
## [1] "2 * 1 = 2"
## [1] "2 * 2 = 4"
```

<sup>9)</sup> しかし、for ループをあまりにも多く重ねるとコードの可読性が低下するので注意しよう。多重の反復処理が必要な場合には、反復処理のための関数を自作することで対応することもできる。

<sup>10)</sup> 交換法則が成り立ち、演算子の左右を区別する意味はないので、本当は 81 パターンも考える必要はない。

```
## [1] "2 * 3 = 6"  
## [1] "2 * 4 = 8"  
## [1] "2 * 5 = 10"  
## [1] "2 * 6 = 12"  
## [1] "2 * 7 = 14"  
## [1] "2 * 8 = 16"  
## [1] "2 * 9 = 18"  
## [1] "3 * 1 = 3"  
## [1] "3 * 2 = 6"  
## [1] "3 * 3 = 9"  
## [1] "3 * 4 = 12"  
## [1] "3 * 5 = 15"  
## [1] "3 * 6 = 18"  
## [1] "3 * 7 = 21"  
## [1] "3 * 8 = 24"  
## [1] "3 * 9 = 27"  
## [1] "4 * 1 = 4"  
## [1] "4 * 2 = 8"  
## [1] "4 * 3 = 12"  
## [1] "4 * 4 = 16"  
## [1] "4 * 5 = 20"  
## [1] "4 * 6 = 24"  
## [1] "4 * 7 = 28"  
## [1] "4 * 8 = 32"  
## [1] "4 * 9 = 36"  
## [1] "5 * 1 = 5"  
## [1] "5 * 2 = 10"  
## [1] "5 * 3 = 15"  
## [1] "5 * 4 = 20"  
## [1] "5 * 5 = 25"  
## [1] "5 * 6 = 30"  
## [1] "5 * 7 = 35"
```

```
## [1] "5 * 8 = 40"  
## [1] "5 * 9 = 45"  
## [1] "6 * 1 = 6"  
## [1] "6 * 2 = 12"  
## [1] "6 * 3 = 18"  
## [1] "6 * 4 = 24"  
## [1] "6 * 5 = 30"  
## [1] "6 * 6 = 36"  
## [1] "6 * 7 = 42"  
## [1] "6 * 8 = 48"  
## [1] "6 * 9 = 54"  
## [1] "7 * 1 = 7"  
## [1] "7 * 2 = 14"  
## [1] "7 * 3 = 21"  
## [1] "7 * 4 = 28"  
## [1] "7 * 5 = 35"  
## [1] "7 * 6 = 42"  
## [1] "7 * 7 = 49"  
## [1] "7 * 8 = 56"  
## [1] "7 * 9 = 63"  
## [1] "8 * 1 = 8"  
## [1] "8 * 2 = 16"  
## [1] "8 * 3 = 24"  
## [1] "8 * 4 = 32"  
## [1] "8 * 5 = 40"  
## [1] "8 * 6 = 48"  
## [1] "8 * 7 = 56"  
## [1] "8 * 8 = 64"  
## [1] "8 * 9 = 72"  
## [1] "9 * 1 = 9"  
## [1] "9 * 2 = 18"  
## [1] "9 * 3 = 27"
```

```
## [1] "9 * 4 = 36"  
## [1] "9 * 5 = 45"  
## [1] "9 * 6 = 54"  
## [1] "9 * 7 = 63"  
## [1] "9 * 8 = 72"  
## [1] "9 * 9 = 81"
```

上のコードがどのような処理をしているか考えてみよう。まず、`i` に 1 が代入される。次に、`j` に 1 から 9 までの数順番に 1 つずつ代入され、`1 * 1`、`1 * 2`、`1 * 3`、...、`1 * 9` が計算される。`1 * 9` の計算が終わったら、`i` に 2 が代入される。そして再び内側の for ループが実行され、`2 * 1`、`2 * 2`、`2 * 3`、...、`2 * 9` が計算される。これを 9 の段まで繰り返す。段の順番を降順にして 9 の段を最初に計算し、1 の段を最後に計算したい場合は、`i in 1:9` を `i in 9:1` に変えればよい。

```
1 for (i in 9:1) {  
2   for (j in 1:9) {  
3     print(paste(i, "*", j, "=", i * j))  
4   }  
5 }  
  
## [1] "9 * 1 = 9"  
## [1] "9 * 2 = 18"  
## [1] "9 * 3 = 27"  
## [1] "9 * 4 = 36"  
## [1] "9 * 5 = 45"  
## [1] "9 * 6 = 54"  
## [1] "9 * 7 = 63"  
## [1] "9 * 8 = 72"  
## [1] "9 * 9 = 81"  
## [1] "8 * 1 = 8"  
## [1] "8 * 2 = 16"  
## [1] "8 * 3 = 24"  
## [1] "8 * 4 = 32"  
## [1] "8 * 5 = 40"
```

```
## [1] "8 * 6 = 48"  
## [1] "8 * 7 = 56"  
## [1] "8 * 8 = 64"  
## [1] "8 * 9 = 72"  
## [1] "7 * 1 = 7"  
## [1] "7 * 2 = 14"  
## [1] "7 * 3 = 21"  
## [1] "7 * 4 = 28"  
## [1] "7 * 5 = 35"  
## [1] "7 * 6 = 42"  
## [1] "7 * 7 = 49"  
## [1] "7 * 8 = 56"  
## [1] "7 * 9 = 63"  
## [1] "6 * 1 = 6"  
## [1] "6 * 2 = 12"  
## [1] "6 * 3 = 18"  
## [1] "6 * 4 = 24"  
## [1] "6 * 5 = 30"  
## [1] "6 * 6 = 36"  
## [1] "6 * 7 = 42"  
## [1] "6 * 8 = 48"  
## [1] "6 * 9 = 54"  
## [1] "5 * 1 = 5"  
## [1] "5 * 2 = 10"  
## [1] "5 * 3 = 15"  
## [1] "5 * 4 = 20"  
## [1] "5 * 5 = 25"  
## [1] "5 * 6 = 30"  
## [1] "5 * 7 = 35"  
## [1] "5 * 8 = 40"  
## [1] "5 * 9 = 45"  
## [1] "4 * 1 = 4"
```

```
## [1] "4 * 2 = 8"  
## [1] "4 * 3 = 12"  
## [1] "4 * 4 = 16"  
## [1] "4 * 5 = 20"  
## [1] "4 * 6 = 24"  
## [1] "4 * 7 = 28"  
## [1] "4 * 8 = 32"  
## [1] "4 * 9 = 36"  
## [1] "3 * 1 = 3"  
## [1] "3 * 2 = 6"  
## [1] "3 * 3 = 9"  
## [1] "3 * 4 = 12"  
## [1] "3 * 5 = 15"  
## [1] "3 * 6 = 18"  
## [1] "3 * 7 = 21"  
## [1] "3 * 8 = 24"  
## [1] "3 * 9 = 27"  
## [1] "2 * 1 = 2"  
## [1] "2 * 2 = 4"  
## [1] "2 * 3 = 6"  
## [1] "2 * 4 = 8"  
## [1] "2 * 5 = 10"  
## [1] "2 * 6 = 12"  
## [1] "2 * 7 = 14"  
## [1] "2 * 8 = 16"  
## [1] "2 * 9 = 18"  
## [1] "1 * 1 = 1"  
## [1] "1 * 2 = 2"  
## [1] "1 * 3 = 3"  
## [1] "1 * 4 = 4"  
## [1] "1 * 5 = 5"  
## [1] "1 * 6 = 6"
```

```
## [1] "1 * 7 = 7"  
## [1] "1 * 8 = 8"  
## [1] "1 * 9 = 9"
```

九九を覚えるときにはこれで良いが、実数どうしの掛け算で  $i * j$  と  $j * i$  は同じ（交換法則が成り立つ）なので、2つの数字の積を知りたいだけなら片方だけ出力すれば十分だろう。そこで、`for (j in 1:9)` を `for (j in i:9)` に変えてみよう。

```
1 for (i in 1:9) {  
2   for (j in i:9) {  
3     print(paste(i, "*", j, "=", i * j))  
4   }  
5 }
```

```
## [1] "1 * 1 = 1"  
## [1] "1 * 2 = 2"  
## [1] "1 * 3 = 3"  
## [1] "1 * 4 = 4"  
## [1] "1 * 5 = 5"  
## [1] "1 * 6 = 6"  
## [1] "1 * 7 = 7"  
## [1] "1 * 8 = 8"  
## [1] "1 * 9 = 9"  
## [1] "2 * 2 = 4"  
## [1] "2 * 3 = 6"  
## [1] "2 * 4 = 8"  
## [1] "2 * 5 = 10"  
## [1] "2 * 6 = 12"  
## [1] "2 * 7 = 14"  
## [1] "2 * 8 = 16"  
## [1] "2 * 9 = 18"  
## [1] "3 * 3 = 9"  
## [1] "3 * 4 = 12"  
## [1] "3 * 5 = 15"
```

```
## [1] "3 * 6 = 18"  
## [1] "3 * 7 = 21"  
## [1] "3 * 8 = 24"  
## [1] "3 * 9 = 27"  
## [1] "4 * 4 = 16"  
## [1] "4 * 5 = 20"  
## [1] "4 * 6 = 24"  
## [1] "4 * 7 = 28"  
## [1] "4 * 8 = 32"  
## [1] "4 * 9 = 36"  
## [1] "5 * 5 = 25"  
## [1] "5 * 6 = 30"  
## [1] "5 * 7 = 35"  
## [1] "5 * 8 = 40"  
## [1] "5 * 9 = 45"  
## [1] "6 * 6 = 36"  
## [1] "6 * 7 = 42"  
## [1] "6 * 8 = 48"  
## [1] "6 * 9 = 54"  
## [1] "7 * 7 = 49"  
## [1] "7 * 8 = 56"  
## [1] "7 * 9 = 63"  
## [1] "8 * 8 = 64"  
## [1] "8 * 9 = 72"  
## [1] "9 * 9 = 81"
```

このコードは、1の段は  $1 * 1$  から  $1 * 9$  までのすべてを計算するが、2の段は  $2 * 2$  から、3の段は  $3 * 3$  から計算を始めて出力するコードである。2の段の場合、 $2 * 1$  は1の段で  $1 * 2$  が計算済みなのでスキップする。3の段の場合、 $3 * 1$  は1の段で、 $3 * 2$  は2の段でそれぞれ計算済みなのでとばす。それぞれの段で同様の処理を続け、最後の9の段では  $9 * 9$  のみが計算される。

このように多重 for ループは複数のベクトルの組み合わせて処理を行う場合に便利で

ある。

複数のベクトルでなく、データフレームなどの2次元以上データにも多重forループは使われる。データフレームは複数のベクトルで構成されている（各列が1つのベクトル）ため、実質的には複数のベクトルを扱うことになる。

例として、FIFA\_Men.csvを利用し、それぞれの国のチーム名、FAFAランキング、ポイントをまとめて表示するコードを書いてみよう。すべてのチームを表示すると結果が長くなるので、対象をOFC（オセアニアサッカー連盟）所属チームに限定する。

```
1 # FIFA_Men.csv を読み込み、myDF という名で保存
2 my_df <- read_csv("Data/FIFA_Men.csv")
3 # my_df の Confederation 列が OFC の行だけを抽出
4 my_df <- my_df[my_df$Confederation == "OFC", ]
5
6 my_df
```

```
## # A tibble: 10 x 6
##       ID Team          Rank Points Prev_Points Confederation
##   <dbl> <chr>     <dbl>   <dbl>      <dbl> <chr>
## 1     4 American Samoa     192     900       900 OFC
## 2     69 Fiji             163     996       996 OFC
## 3    135 New Caledonia    156    1035      1035 OFC
## 4    136 New Zealand     122    1149      1149 OFC
## 5    147 Papua New Guinea 165     991       991 OFC
## 6    159 Samoa            194     894       894 OFC
## 7   171 Solomon Islands   141    1073      1073 OFC
## 8   185 Tahiti           161    1014      1014 OFC
## 9   191 Tonga            203     862       862 OFC
## 10  204 Vanuatu          163     996       996 OFC
```

OFCに10チーム加盟していることがわかる。

以下のような内容が表示されるコードを書きたい。

=====1 番目のチーム情報=====

Team: American Samoa

Rank: 192

Points: 900

=====2 番目のチーム情報=====

Team: Fiji

Rank: 163

Points: 996

=====3 番目のチーム情報=====

Team: New Caledonia

...

これは1つのforループでも作成できるが、勉強のために2つのforループを使って書いてみよう。

```
1 for (i in 1:nrow(my_df)) {  
2   print(paste0("=====", i, "番目のチーム情報====="))  
3  
4   for (j in c("Team", "Rank", "Points")) {  
5     print(paste0(j, ": ", my_df[i, j]))  
6   }  
7 }
```

以上のコードを実行すると以下のような結果が表示される（全部掲載すると長くなるので、ここでは最初の2チームのみ掲載する。

```
## [1] "=====1 番目のチーム情報===== "  
## [1] "Team: American Samoa"  
## [1] "Rank: 192"  
## [1] "Points: 900"  
## [1] "=====2 番目のチーム情報===== "  
## [1] "Team: Fiji"  
## [1] "Rank: 163"  
## [1] "Points: 996"
```

このコードについて説明しよう。まず、外側の for ループでは任意の変数としてインデクス *i* を、内側の for ループではインデクス *j* を使っている。R は、コードを上から順番に処理する（同じ行なら、カッコの中から処理する）。したがって、まず処理されるのは外側の for ループである。*i* にはベクトル `1:nrow(my_df)` の要素が順番に割り当てられる。`nrow()` は行列またはデータフレームの行数を求める関数で、`my_df` は 10 行のデータなので `1:10` になる。つまり、外側の for ループは *i* に `1, 2, 3, ..., 10` の順で値を格納しながらループ内の処理を繰り返す。ここで、外側の for ループの中身を見てみよう。まず、`print()` を使って "===== *i* 番目のチーム情報=====" というメッセージを出力している。最初は *i* が 1 なので、"=====1 番目のチーム情報=====" が表示される。

次に実行されるのが内側の for ループである。ここでは *j* に `c("Team", "Rank", "Points")` を格納しながら内側の for ループの内のコードをが 3 回繰り返し処理される。内側のコードの内容は、たとえば、*i* = 1 の状態で、*j* = "Team"なら、`print(paste0("Team", ":", my_df[1, "Team"]))` である。第 9.4 章で説明した通り、`my_df[1, "Team"]` は `my_df` の `Team` 列の 1 番目の要素を意味する。この処理が終わると、次は *j* に "Rank" が代入され、同じコードを処理する。そして、*j* = "Points" まで処理が終わったら、内側の for ループは一度終了する。

内側の for ループが終わっても、外側の for ループはまだ終わっていない。次は、*i* に 2 が格納され、"=====2 番目のチーム情報=====" を表示し、**再度内側の for ループを最初から処理する**。*i* = 10 の状態で内側の for ループが終了すると外側の for ループも終了する。多重 for ループはこのように反復処理を実行する。

チーム名の次に所属連盟も表示したいときはどう直せば良いだろうか。正解は `c("Team", "Rank", "Points")` のベクトルで、"Team" と "Rank" の間に "Confederation" を追加するだけである。実際にやってみよう（スペースの関係上、最初の 2 チームの結果のみ掲載する）。

```
1 for (i in 1:nrow(my_df)) {  
2   print(paste0("=====", i, "番目のチーム情報====="))  
3  
4   for (j in c("Team", "Confederation", "Rank", "Points")) {  
5     print(paste0(j, ":", my_df[i, j]))  
6   }
```

```

7  }

## [1] "=====1 番目のチーム情報====="
## [1] "Team: American Samoa"
## [1] "Confederation: OFC"
## [1] "Rank: 192"
## [1] "Points: 900"
## [1] "=====2 番目のチーム情報====="
## [1] "Team: Fiji"
## [1] "Confederation: OFC"
## [1] "Rank: 163"
## [1] "Points: 996"

```

ちなみに、1つの for ループで同じ結果を実現するには次のようにする。結果は掲載しないが、自分で試してみてほしい。また、cat() 関数の使い方については?cat を参照されたい。ちなみに\n は改行コードである。

```

1 for (i in 1:nrow(my_df)) {
2   cat(paste0("=====", i, "番目のチーム情報=====\\n"))
3   cat(paste0("Team:", my_df$Team[i], "\\n",
4             "Rank:", my_df$Rank[i], "\\n",
5             "Points:", my_df$Points[i], "\\n"))
6 }

```

リスト型を対象とした多重 for ループの例も確認しておこう。複数のベクトルを含むリストの場合、3 番目のベクトルの 5 番目の要素を抽出するには、リスト名 [[3]][[5]] のように 2 つの位置を指定する必要がある。たとえば、3 人で構成されたグループが 3 つあり、それぞれのグループにおける id (例: 学籍番号) の順番で名前が格納されている my\_list を考えてみよう。

```

1 my_list <- list(A = c("Song", "Wickham", "Yanai"),
2                  B = c("Watanabe", "Toyoshima", "Fujii"),
3                  C = c("Abe", "Moon", "Xi"))

```

ここで、まず各クラスの出席番号 1 番の人の名字をすべて出力し、次は 2 番の人、最後

に 3 番の人を表示するにはどうすれば良いだろうか。以下のコードを見てみよう。

```
1 for (i in 1:3) {  
2  
3   for (j in names(my_list)) {  
4     print(my_list[[j]][i])  
5   }  
6  
7   print(paste0("==>ここまでが出席番号", i, "番の人です==>"))  
8 }
```

```
## [1] "Song"  
## [1] "Watanabe"  
## [1] "Abe"  
## [1] "==>ここまでが出席番号 1 番の人です==>"  
## [1] "Wickham"  
## [1] "Toyoshima"  
## [1] "Moon"  
## [1] "==>ここまでが出席番号 2 番の人です==>"  
## [1] "Yanai"  
## [1] "Fujii"  
## [1] "Xi"  
## [1] "==>ここまでが出席番号 3 番の人です==>"
```

このコードは以下のように動く。

- 1 行目: 任意の変数を `i` とし、1 から 3 までの数字を `i` に格納しながら、3 回反復作業を行う。
- 3 行目: 任意の変数を `j` とし、ここにはリストの要素名 (A, B, C) を格納しながら、リストの長さだけ処理を繰り返す。`names(my_list)` は `c("A", "B", "C")` である。
- 4 行目: `my_list[[j]][i]` の内容を出力する。最初は `i = 1`、`j = "A"` なので、`print(my_list[["A"]][1])`、つまり、`my_list` から "A" を取り出し、そこの 1 番目の要素を出力する

- 7行目: 各ベクトルから *i* 番目の要素を出力し、`print(paste0("==出席番号", i, "番の人です=="))` を実行する。*i* に次の要素を入れ、作業を反復する。*i* に格納する要素がなくなったら、反復を終了する。

多重 for ループは 3 重、4 重にすることも可能だが、コードの可読性が低下するため、多くても 3 重、できれば最大二重までにしておいたほうがよい。3 重以上に for ループを重ねる場合は、内側の for ループを後ほど解説する関数でまとめたほうがいいだろう。

上の例では、リストの各要素に名前 (A, B, C) が付けられていることを利用した。リストに名前がない場合はどうすれば良いだろうか (そもそも、名前のないリストなど作らないほうが良いのだが...)。例えば、次のような場合である。

```
1 my_list <- list(c("Song", "Wickham", "Yanai"),
2                   c("Watanabe", "Toyoshima", "Fujii"),
3                   c("Abe", "Moon", "Xi"))
```

この場合には、次のようにすればよい。

```
1 for (i in 1:3) {
2
3   for (j in seq_along(my_list)) {
4     print(my_list[[j]][i])
5   }
6
7   print(paste0("==出席番号", i, "番の人です=="))
8 }
```

```
## [1] "Song"
## [1] "Watanabe"
## [1] "Abe"
## [1] "==出席番号 1 番の人です=="
## [1] "Wickham"
## [1] "Toyoshima"
## [1] "Moon"
## [1] "==出席番号 2 番の人です=="
```

```
## [1] "Yanai"  
## [1] "Fujii"  
## [1] "Xi"  
## [1] "====ここまでが出席番号 3 番の人です===="
```

内側の for ループでリストの要素名を使う代わりに、リストの要素を位置で指定している。

### 10.3.2 while による反復

for によるループは任意の変数にベクトルの要素を 1 つずつ代入し、ベクトルの要素を使い尽くすまで反復処理を行う。したがって、ベクトルの長さによってループの回数が決まる。

それに対し、「ある条件が満たされる限り、反復し続ける」あるいは「ある条件が満たされるまで、反復し続ける」ことも可能である。そのときに使うのが、while による while ループである。while ループの書き方は for ループにとてもよく似ている。

基本的な使い方は、以下のとおりである。

```
1 while (条件) {  
2     条件が満たされた場合の処理内容  
3 }
```

for が while に変わり、() 内の書き方が (任意の変数 in ベクトル) から (条件) に変わった。この () 内の条件が満たされる間は{}内の内容を処理が繰り返し実行される。1 から 5 までの整数を表示するコードは、for ループを使うと次のように書ける。

```
1 for (i in 1:5) {  
2     print(i)  
3 }
```

これを while ループを使って書き直すと、次のようになる。

```
1 i <- 1  
2 while (i < 6) {
```

```
3   print(i)
4   i <- i + 1
5 }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

while ループを繰り返す条件は `i < 6` である。これにより、「`i` が 6 未満」なら処理が繰り返される。注意すべき点として、`i` が 6 未満か否かの判断をループの開始時点でので、あらかじめ変数 `i` を用意する必要があることがあげられる。1 行めにある `i <- 1` がそれである。そして、{}内の最終行に `i <- i + 1` があり、`i` を 1 ずつ増やしている。`i = 5` の時点では `print(i)` が実行されるが、`i = 6` になると、ループをもう 1 ど実行する条件が満たされないので、反復が停止する。

`i <- i + 1` のように内容を少しづつ書き換えるさいは、そのコードを置く位置にも注意が必要だ。先ほどのコードのうち、`i <- i + 1` を `print(i)` の前に移動してみよう。

```
1 i <- 1
2
3 while (i < 6) {
4   i <- i + 1
5   print(i)
6 }
```

```
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

2 から 6 までの整数が output された。これは `i` を出力する前に `i` に 1 が足されるためであ

る。`i <- i + 1` の位置をこのままにして、先ほどと同じように 1 から 5 までの整数を表示するには、次のようにコードを書き換える。

```
1 i <- 0
2
3 while (i < 5) {
4     i <- i + 1
5     print(i)
6 }
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

変更点したのは、

1. `i` の初期値を 0 にした 2. () 内の条件を `(i < 6)` から `(i < 5)` にした  
という 2 点である。

while ループは慣れないとややこしいと感じるだろう。for ループで代替できるケースも多い。それでも while 文が必要になるケースもある。それは先述した通り、「目標は決まっているが、その目標が達成されるまでは処理を何回繰り返せばいいか分からぬ」場合である。

たとえば、6 面のサイコロ投げを考えよう。投げる度に出た目を記録し、その和が 30 以上に達したときサイコロ投げを中止するにはどうすればいいだろうか。何回サイコロを投げればいいかがあらかじめわからぬと、for ループは使えない。連續で 6 が出るなら 5 回で十分かもしれないが、1 が出続けるなら 30 回投げる必要がある。このように、「反復処理は行うが、何回行えばいいか分からぬ。ただし、停止条件は知っている」場合に while ループを使う。

サイコロ投げの例は、以下のコードで実行できる。

```
1 total <- 0
2 trial <- 1
3
4 while (total < 30) {
5   die <- sample(1:6, size = 1)
6   total <- total + die
7
8   print(paste0(trial, "回目のサイコロ投げの結果: ", die,
9               " (これまでの総和: ", total, ") "))
10
11  # print() 文は以下のような書き方も可能
12  # Result <- sprintf("%d 回目のサイコロ投げの結果: %d (これまでの総和: %d) ",
13  #                      trial, die, total)
14  # print(Result)
15
16  trial <- trial + 1
17 }
```

```
## [1] "1 回目のサイコロ投げの結果: 5 (これまでの総和: 5) "
## [1] "2 回目のサイコロ投げの結果: 6 (これまでの総和: 11) "
## [1] "3 回目のサイコロ投げの結果: 4 (これまでの総和: 15) "
## [1] "4 回目のサイコロ投げの結果: 5 (これまでの総和: 20) "
## [1] "5 回目のサイコロ投げの結果: 2 (これまでの総和: 22) "
## [1] "6 回目のサイコロ投げの結果: 2 (これまでの総和: 24) "
## [1] "7 回目のサイコロ投げの結果: 3 (これまでの総和: 27) "
## [1] "8 回目のサイコロ投げの結果: 4 (これまでの総和: 31) "
```

このコードの中身を説明しよう。まず、これまで出た目の和を記録する変数 `total` を用意し、初期値として 0 を格納する。また、何回目のサイコロ投げかを記録するために `trial` という変数を用意し、初期値として 1 を格納する（コードの 1、2 行目）。

次に、`while` 文を書く。反復する条件は `total` が 30 未満と設定する。つまり、`total` が 30 以上になったら反復を終了する（コードの 4 行目）。

続いて、サイコロを投げ、出た目を `die` という変数に格納します。サイコロ投げは 1 から 6 の間の整数から無作為に 1 つを値を抽出することでシミュレートできる。そこで使われるのが `sample()` 関数である（コードの 5 行目）。`sample()` 関数は与えられたベクトル内の要素を無作為に抽出する。`sample(c(1, 2, 3, 4, 5, 6), size = 1)` は「`c(1, 2, 3, 4, 5, 6)` から 1 つの要素を無作為に抽出せよ」という意味である（乱数生成を利用したシミュレーションについては後の章で詳しく説明する）。サイコロの目が出たら、その目を `total` の値に足す（コードの 6 行目）。

その後、「1 回目のサイコロ投げの結果: 5 (これまでの総和: 5)」のように、1 回の処理の結果を表示する。（コードの 8 行目）。最後に `trial` の値を 1 増やし（コードの 14 行目）、1 度の処理が終了する。

1 度の処理が終わったら、`while` ループの条件判断に戻り、条件が満たされていれば再び処理を繰り返す。これを条件が満たされなくなるまで繰り返す。

この反復処理は、`for` ループで再現することもできる。次のようにすればよい。

```
1 total <- 0
2
3 for (trial in 1:30) {
4   die <- sample(1:6, 1)
5   total <- total + die
6
7   # print() の代わりに sprintf() を使うことも可能
8   result <- sprintf("%d 回目のサイコロ投げの結果: %d (これまでの総和: %d) ",
9                     trial, die, total)
10  print(result)
11
12  if (total >= 30) break() # () は省略可能
13 }
```

```
## [1] "1 回目のサイコロ投げの結果: 5 (これまでの総和: 5) "
## [1] "2 回目のサイコロ投げの結果: 2 (これまでの総和: 7) "
## [1] "3 回目のサイコロ投げの結果: 1 (これまでの総和: 8) "
## [1] "4 回目のサイコロ投げの結果: 4 (これまでの総和: 12) "
```

```
## [1] "5回目のサイコロ投げの結果: 2 (これまでの総和: 14) "
## [1] "6回目のサイコロ投げの結果: 6 (これまでの総和: 20) "
## [1] "7回目のサイコロ投げの結果: 6 (これまでの総和: 26) "
## [1] "8回目のサイコロ投げの結果: 1 (これまでの総和: 27) "
## [1] "9回目のサイコロ投げの結果: 2 (これまでの総和: 29) "
## [1] "10回目のサイコロ投げの結果: 3 (これまでの総和: 32) "
```

このコードの中身を説明しよう。事前にサイコロを投げる回数はわからないが、6面サイコロの場合30回以内には必ず合計が30になるので、`trial in 1:30`としておく。あとはwhileループの書き方とほぼ同じだが、forループでは`i`が自動的に更新されるので`i <- i + 1`は不要である。さらに、一定の条件が満たされたときにループを停止する必要がある。そこで登場するのが条件`if`と`break()`である。条件分岐は次節で説明するが、`if`から始まる行のコードは、「`total`が30以上になったらループから脱出せよ」ということを意味する。このようにループからの脱出を指示する関数が`break()`だ。`break()`関数は`()`を省略して`break`と書いてもよい。

これは余談だが、結果の表示に今回はこれまで使ってきた`print()`と`paste0()`（または`paste()`）の代わりに、`sprintf()`を使っている。`sprintf()`内の`%d`はその位置に指定された変数の値を整数として代入することを意味する。`sprintf()`に引数にはまず出力する文字列を指定し、次に代入する変数を順番に入力する。`%s`は文字(string)型、`%f`は実数(floating-point number)を意味する。`%f`では小数の桁数も指定可能であり、小数第2位まで表示させるには`%.2f`のように表記する。以下のコードは3つの変数の値と文字列を結合する処理を`print(paste0())`と`sprintf()`を用いて書いたもので、同じ結果が得られる。

```
1 name <- "Song"
2 bowls <- 50
3 height <- 176.2
4
5 print(paste0(name, "がひと月に食べるラーメンは", bowls, "杯で、身長は", height, "cm です"))
## [1] "Song がひと月に食べるラーメンは 50 杯で、身長は 176.2cm です。"
```

```
1 # %s に name を、%d に bowls を、%.1f に height を小数第 1 位まで格納し、出力
2 sprintf("%s がひと月に食べるラーメンは%d 杯で、身長は%.1fcm です。", name, bowls, height)

## [1] "Song がひと月に食べるラーメンは 50 杯で、身長は 176.2cm です。"
```

ただし、{}内の sprintf() はそのまま出力されないため、一旦、オブジェクトとして保存し、それを print() を使って出力する必要がある。詳細については、?sprintf を参照されたい。

今回例として挙げたサイコロ投げは「多くても 30 回以内に終わる」ことが分かっていたの for ループで書き換えることができた。しかし、終了までに必要な反復回数は未知であることもある。目的に応じて for と while を使い分けることが求められる。

## 10.4 条件分岐

### 10.4.1 if、else if、else による条件分岐

この節では条件分岐について説明する。条件分岐は、条件に応じて異なる処理を行いときに行利用する。if による条件分岐の基本形は次のとおりである。

```
1 if (条件) {
2   条件が満たされた場合のみ実行される処理の内容
3 }
```

while を使ったループによく似ているが、while 文は条件が満たされる限り{}の内容を繰り返し実行するのに対し、if 文では条件が満たされると{}の内容が 1 回だけ実行されて処理が終了するという違いがある。たとえば、名前が格納されているオブジェクト name の中身が"Song"なら「ラーメン大好き」と出力されるコードを考えてみよう。

```
1 name <- "Song"
2
3 if (name == "Song") {
4   print("ラーメン大好き")
```

```
5  }

## [1] "ラーメン大好き"
```

`if` 文の () 内の条件は、「`name` の中身が "Song"」の場合のみ {} 内の処理を実行せよということを意味する。`name` の中身が "Yanai" ならどうなるだろうか。

```
1 name <- "Yanai"

2

3 if (name == "Song") {
4   print("ラーメン大好き")
5 }
```

何も表示されない。このように、`if` 文単体だと、条件が **満たされない** 場合には何も実行されない。

条件に応じて異なる処理を実行するために、`else` を使う。これは `if` とセットで使われるもので、`if` の条件が満たされなかった場合の処理内容を指定することができる。`else` を加えると、次のようなコードが書ける。

```
1 if (条件) {
2   条件が満たされた場合の処理内容
3 } else {
4   条件が満たされなかった場合の処理内容
5 }
```

`else` 文は、以下のように改行して書くこともできる。

```
1 if (条件) {
2   条件が満たされた場合の処理内容
3 }
4 else {
5   条件が満たされなかった場合の処理内容
6 }
```

しかし、この書き方はあまり良くない。`else` は必ず `if` とセットで用いられるので、`if`

の処理の終わりである } の直後に半角スペースを 1 つ挟んで書くのが標準的なコーディングスタイルである。

それでは name が "Song" でない場合には、「ラーメン好きじゃない」表示されるコードを書いてみよう。

```
1 name <- "Song"  
2  
3 if (name == "Song") {  
4   print("ラーメン大好き")  
5 } else {  
6   print("ラーメン好きじゃない")  
7 }
```

```
## [1] "ラーメン大好き"
```

name が "Song" の場合、前と変わらず「ラーメン大好き」が出力される。name を "Yanai" に変えてみよう。

```
1 name <- "Yanai"  
2  
3 if (name == "Song") {  
4   print("ラーメン大好き")  
5 } else {  
6   print("ラーメン好きじゃない")  
7 }
```

```
## [1] "ラーメン好きじゃない"
```

「ラーメン好きじゃない」が表示される。

しかし、世の中はと「ラーメン大好き」と「ラーメン好きじゃない」のみで構成されているわけではない。「ラーメン大好き」なのは Song と Koike、「ラーメン好きじゃない」のは Yanai のみで、それ以外の人は「ラーメンそこそこ好き」だとしよう。つまり 3 つのパターンがある。それぞれに別の処理を実行するには、3 つ以上の条件に対応できる条件分岐が必要になる。

そのような場合には `else if` を `if` と `else` の間に挿入する。

```

1 if (条件 1) {
2   条件 1 が満たされた場合の処理内容
3 } else if (条件 2) {
4   条件 1 が満たされず、条件 2 が満たされた場合の処理内容
5 } else if (条件 3) {
6   条件 1, 2 が満たされず、条件 3 が満たされた場合の処理内容
7 } else {
8   条件が全て満たされなかった場合の処理内容
9 }
```

このように条件分岐を書くと、`if` 文の条件が満たされれば最初の{}内の処理を実行し、満たされなかったら次の `else if` の条件を判定する。その条件が満たされれば{}の処理を実行し、満たされなければ次の `else if` へ移動...を順に実施する。どの条件も満たされないときは、最終的に `else` の内容が実行される。

それでは実際にコードを書いてみよう。

```

1 name <- "Song"
2
3 if (name == "Song" | name == "Koike") {
4   # 上の条件は、(name %in% c("Song", "Koike")) でも OK
5   print("ラーメン大好き")
6 } else if (name == "Yanai") {
7   print("ラーメン好きじゃない")
8 } else {
9   print("ラーメンそこそこ好き")
10 }
```

## [1] "ラーメン大好き"

このコードでは、まず `name` が "Song" または "Koike" か否かを判定し、TRUE なら「ラーメン大好き」を表示する。FALSE なら次の `else if` 文へ移動する。ここでは `Name` が "Yanai" かどうかを判定する。

最初の条件に登場する | は「OR (または)」を意味する論理演算子であり、第 6 章で解説した。このように、条件の中で | や &などの論理演算子を使うことで、複数の条件を指定することができる。また、`name == "Song" | name == "Koike"` は `name %in% c("Song", "Koike")` に書き換えることができる。`x %in% y` は `x` が `y` に含まれているか否かを判定する演算子である。たとえば、`"A" %in% c("A", "B", "C")` の結果は TRUE だが、`"Z" %in% c("A", "B", "C")` の結果は FALSE になる。

それでは `name` を "Yanai"、"Koike"、"Shigemura"、"Hakiai" に変えながら結果を確認してみよう。

```
1 name <- "Yanai"  
2  
3 if (name %in% c("Song", "Koike")) {  
4   print("ラーメン大好き")  
5 } else if (name == "Yanai") {  
6   print("ラーメン好きじゃない")  
7 } else {  
8   print("ラーメンそこそこ好き")  
9 }
```

```
## [1] "ラーメン好きじゃない"
```

```
1 name <- "Koike"  
2  
3 if (name %in% c("Song", "Koike")) {  
4   print("ラーメン大好き")  
5 } else if (name == "Yanai") {  
6   print("ラーメン好きじゃない")  
7 } else {  
8   print("ラーメンそこそこ好き")  
9 }
```

```
## [1] "ラーメン大好き"
```

```
1 name <- "Shigemura"
2
3 if (name %in% c("Song", "Koike")) {
4   print("ラーメン大好き")
5 } else if (name == "Yanai") {
6   print("ラーメン好きじゃない")
7 } else {
8   print("ラーメンそこそこ好き")
9 }
```

```
## [1] "ラーメンそこそこ好き"
```

```
1 name <- "Hakiai"
2
3 if (name %in% c("Song", "Koike")) {
4   print("ラーメン大好き")
5 } else if (name == "Yanai") {
6   print("ラーメン好きじゃない")
7 } else {
8   print("ラーメンそこそこ好き")
9 }
```

```
## [1] "ラーメンそこそこ好き"
```

`if` 文が単独で使われることもそれほど多くない。`if` 文は主に反復処理を行う `for` ループまたは `while()` ループの中、もしくは次節で説明する自作関数内に使われることが多い。上の例では名前からラーメンに対する情熱を判定するために何度も同じコードを書いてきたが、同じコードを繰り返し書くのは効率が悪い。繰り返し行う作業を関数としてまとめれば便利である。関数の作成については第11章で説明する。

ここでは `for` ループと条件分岐の組み合わせについて考えてみよう。学生10人の成績が入っているベクトル `scores` があるとしよう。そして、成績が60点以上なら「合格」を、未満なら「不合格」を返すようなコードを書く。この作業を効率的に行うために、`for` ループと `if` 文を組み合わせる方法を考える。`for` ループではインデクス変数 `i` に1から

10までの数字を代入することを繰り返す。ここでの10はベクトル `scores` の長さなので、`seq_along()` を使う。そして、`scores` の `i` 番目要素に対して60点以上か否かの判定を行い、「合格」または「不合格」という結果を返す。

以上の内容を実行するコードは、次のように書ける。

```
1 scores <- c(58, 100, 81, 97, 71, 61, 47, 60, 73, 85)
2
3 for (i in seq_along(scores)) {
4   if (scores[i] >= 60) {
5     print(paste0("学生", i, "の判定結果: 合格"))
6   } else {
7     print(paste0("学生", i, "の判定結果: 不合格"))
8   }
9 }
```

```
## [1] "学生 1 の判定結果: 不合格"
## [1] "学生 2 の判定結果: 合格"
## [1] "学生 3 の判定結果: 合格"
## [1] "学生 4 の判定結果: 合格"
## [1] "学生 5 の判定結果: 合格"
## [1] "学生 6 の判定結果: 合格"
## [1] "学生 7 の判定結果: 不合格"
## [1] "学生 8 の判定結果: 合格"
## [1] "学生 9 の判定結果: 合格"
## [1] "学生 10 の判定結果: 合格"
```

このコードは、以下の処理を行っている。

- 1行目: まず、ベクトル `scores` を定義する。
- 3行目: `for` 文でループを作る。任意の変数としてインデクス `i` を使う。ベクトルの各要素をのインデクスを入れ替えながら反復を行います。したがって、`i in 1:10` でも問題ないが、ここでは `i in seq_along(scores)` を利用する。こうすることで、`scores` ベクトルの長さが変わっても、`for` の内容を修正する必要がなくなる。

- 4, 5 行目: `scores` の `i` 番目要素が 60 点以上かどうかを判定し、`TRUE` なら「学生 `i` の判定結果: 合格」を表示する。
- 6-8 行目: `scores` の `i` 番目要素が 60 点以上でない場合、「学生 `i` の判定結果: 不合格」を表示する。

`print()` 内で `paste0()` を使うとコードの可読性がやや落ちるので、`sprintf()` を使ってもよいかもしれない（後で説明するパイプ演算子 `%>%` を使えば、可読性の問題は解決する）。

```
1 scores <- c(58, 100, 81, 97, 71, 61, 47, 60, 73, 85)
2
3 for (i in seq_along(scores)) {
4   if (scores[i] >= 60) {
5     result <- sprintf("学生 %d の判定結果: 合格", i)
6   } else {
7     result <- sprintf("学生 %d の判定結果: 不合格", i)
8   }
9
10  print(result)
11 }
```

```
## [1] "学生 1 の判定結果: 不合格"
## [1] "学生 2 の判定結果: 合格"
## [1] "学生 3 の判定結果: 合格"
## [1] "学生 4 の判定結果: 合格"
## [1] "学生 5 の判定結果: 合格"
## [1] "学生 6 の判定結果: 合格"
## [1] "学生 7 の判定結果: 不合格"
## [1] "学生 8 の判定結果: 合格"
## [1] "学生 9 の判定結果: 合格"
## [1] "学生 10 の判定結果: 合格"
```

次に、結果を出力するのではなく、結果を別のベクトルに格納する例を考えてみよう。あらかじめ `scores` と同じ長さの空ベクトル `pf` を用意します。ここに各学生について合

否判定の結果を「合格」または「不合格」として格納する処理を行う。以下のようなコードが書ける。

```
1 scores <- c(58, 100, 81, 97, 71, 61, 47, 60, 73, 85)
2 pf      <- rep(NA, length(scores))
3
4 for (i in seq_along(scores)) {
5   if (scores[i] >= 60) {
6     pf[i] <- "合格"
7   } else {
8     pf[i] <- "不合格"
9   }
10 }
```

このコードでは、`scores` を定義した後、中身が `NA` のみで構成される長さが `scores` と同じベクトル `pf` を定義する。`rep(NA, length(scores))` は `NA` を `length(Scores)` 個（ここでは 10 個）並べたベクトルを生成する。続いて、点数によって条件分岐を行い、メッセージを出力するのではなく、`pf` の `i` 番目に"合格"か"不合格"を入れる。結果を確認してみよう。

```
1 pf
2
3 ## [1] "不合格" "合格"    "合格"    "合格"    "合格"    "合格"    "不合格" "合
4 ## [9] "合格"    "合格"
```

このように、`if` 文は反復処理を行う `for` ループまたは `while` ループと組み合わせることでその本領を発揮する。実は、今回の例については次に説明する `ifelse()` を使えば 1 行で処理することができる。しかし、複雑な処理を伴う反復と条件分岐を組み合わせるには、今回のように `for` ループと `if` 文を組み合わせるのが有効である。

### 10.4.2 `ifelse()` による条件分岐

`ifelse()` は与えられたベクトル内の各要素に対して条件分岐を行い、それをベクトルの全要素について繰り返す関数である。簡単な条件分岐と反復処理を同時に使う非常に便

利な関数である。`ifelse()` の基本的な書き方は以下のとおり。

```
1 ifelse(条件, 条件が TRUE の場合の処理、条件が FALSE の場合の処理)
```

たとえば、学生 10 人の成績が入っているベクトル `scores` に対し、60 点以上の場合に合格 ("Pass")、60 点未満の場合に不合格 ("Fail") の値を割り当て、`pf` というベクトルに格納しよう。

```
1 scores <- c(58, 100, 81, 97, 71, 61, 47, 60, 73, 85)
2 pf <- ifelse(scores >= 60, "Pass", "Fail")
3 pf
## [1] "Fail" "Pass" "Pass" "Pass" "Pass" "Pass" "Fail" "Pass" "Pass" "Pass"
```

条件は `scores` の値が 60 以上か否かであり、60 以上なら "Pass" を、それ以外なら "Fail" を返す。

返す値として新しい値を与える代わりに、一定の条件が満たされたら現状の値を保持し、それ以外なら指定した値に変更して返すこともできる。たとえば、世論調査では以下のように「答えたくない」または「わからない」を 9 や 99 などで記録することが多い。この値はこのままでは分析するのが難しいので、欠損値として扱いたいことがある。例として、次の質問を考えよう。

Q. あなたはラーメンが好きですか。

- 1. 大好き
- 2. そこそこ好き
- 3. 好きではない
- 9. 答えたくない

この質問に対する 10 人の回答が格納されたデータフレーム (`Ramen`) があるとしよう。このデータフレームには 2 つの列があり、`id` 列は回答者の ID (識別番号)、`ramen` 列にはは上の質問に対する答えが入っている。

```
1 Ramen <- data.frame(
2   id      = 1:10,
3   ramen  = c(1, 1, 2, 1, 3, 1, 9, 2, 1, 9)
```

```
4  )
5
6 Ramen
```

```
##      id ramen
## 1     1     1
## 2     2     1
## 3     3     2
## 4     4     1
## 5     5     3
## 6     6     1
## 7     7     9
## 8     8     2
## 9     9     1
## 10   10    9
```

この ramen 列に対し、ramen == 9 なら NA を割り当て、それ以外の場合は元の値をそのまま残したいとしよう。そしてその結果を Ramen の ramen 列に上書きする。これは次のコードで実行できる。

```
1 Ramen$ramen <- ifelse(Ramen$ramen == 9, NA, Ramen$ramen)
2 Ramen
```

```
##      id ramen
## 1     1     1
## 2     2     1
## 3     3     2
## 4     4     1
## 5     5     3
## 6     6     1
## 7     7     NA
## 8     8     2
## 9     9     1
## 10   10    NA
```

3つ以上のパターンに分岐させたいときは、`ifelse()` の中に `ifelse()` を使うこともできる。`scores` ベクトルの例を考えてよう。ここで 90 点以上なら S、80 点以上なら A、70 点以上なら B、60 点以上なら C を割り当て、それ以外は F を返すことにしよう。そして、その結果を `grade` という変数に格納してみよう。

```
1 grade <- ifelse(scores >= 90, "S",
2                   ifelse(scores >= 80, "A",
3                         ifelse(scores >= 70, "B",
4                               ifelse(scores >= 60, "C", "F"))))
```

このコードでは、`ifelse()` の条件が `FALSE` の場合、次の `ifelse()` での判定を行う。結果を確認しておこう。

```
1 grade
## [1] "F" "S" "A" "S" "B" "C" "F" "C" "B" "A"
```

この例からもわかるとおり、`ifelse()` を使いすぎるとコードの可読性が低くなるので、このようなコードはあまり推奨できない。複数の条件に応じて返す値を変える処理は、`{dplyr}` パッケージの `case_when()` 関数で行うのがよいだろう。この関数については、13 章で詳細に説明するが、ここでは上のコードと同じ処理を実施する例のみ示す。

```
1 grade2 <- dplyr::case_when(scores >= 90 ~ "S",
2                               scores >= 80 ~ "A",
3                               scores >= 70 ~ "B",
4                               scores >= 60 ~ "C",
5                               TRUE ~ "F")
6
7 grade2
## [1] "F" "S" "A" "S" "B" "C" "F" "C" "B" "A"
```

### 10.4.3 `switch()` による条件分岐

`switch()` は、与えられた長さ 1 の文字列<sup>11)</sup>を用いた条件分岐を行う。これが単体で使われることはほとんどなく、通常は関数のなかで使われる。したがって、初めて本書を読む場合はこの節を一旦とばし、第 11 章を読んだ後に必要に応じて戻ってきてほしい。

ここでは 2 つの数値 `x` と `y` の加減乗除を行う関数 `m_calc` という名前の関数を作る。この関数は `x` と `y` 以外に `operation` という引数をもち、この `operation` の値によって行う処理が変わる。たとえば、`operation = "+"`なら足し算を、`operation = "*"`なら掛け算を行う。

```
1 my_calc <- function(x, y, operation) {  
2   switch(operation,  
3     "+" = x + y,  
4     "-" = x - y,  
5     "*" = x * y,  
6     "/" = x / y,  
7     stop("operation の値に使えるのは +, -, *, / のみです。")  
8   )  
9 }
```

このコードの中身を確認しよう。重要なのは 2 行目から 8 行目の部分である。`switch()` の最初の引数は条件を判定する長さ 1 のベクトルである。ここではこれが `operation` である。そして、`operation` に指定される実引数の値によって異なる処理を行う。 `"+" = x + y` は「`operation` の値が "+"なら、`x + y` を実行する」ことを意味する。これを他の演算子に対しても指定する。最後の引数は、どの条件にも合わない場合の処理である。ここでは、`operation` の値に使えるのは `+, -, *, /` のみです。"というエラーメッセージを表示し、関数の実行を停止するために `stop()` 関数を指定している。

上のコードは、`if`、`else if`、`else` を使って書き換えることができる。

---

<sup>11)</sup> 長さ 1 の数値型ベクトルにも使えるが、使われる例があまりない。

```
1 my_calc2 <- function(x, y, operation = c("+", "-", "*", "/")) {  
2  
3   operation <- match.arg(operation)  
4  
5   if (operation == "+") {  
6     return(x + y)  
7   } else if (operation == "-") {  
8     return(x - y)  
9   } else if (operation == "*") {  
10     return(x * y)  
11   } else {  
12     return(x / y)  
13   }  
14 }
```

先ほどと異なる点は、`function()` の中で `operation` のデフォルト値をベクトルとしたとしたことだ。これは「`operation` 引数の値がこれらの値以外になることは許さない」ということを意味する。`match.arg()` 関数は、`operation` 引数が予め指定された引数の値と一致するか否かを判断する。指定された引数と一致しない場合、エラーを表示し、処理を中断する。ちなみに、`match.arg()` は `switch()` 文と組み合わせて使うこともできる。

今回の例の場合、`switch()` を使ったほうがコードの内容がわかりやすく、読みやすいが、`if` による条件分岐でも十分に対応可能である。また、条件によって行う処理が複雑な場合には `switch()` よりも `if` のほうが使いやすい。

作った関数を使ってみよう。

```
1 my_calc(5, 3, operation = "+")  
## [1] 8  
1 my_calc(5, 3, operation = "-")  
## [1] 2
```

```
1 my_calc(5, 3, operation = "*")
```

```
## [1] 15
```

```
1 my_calc(5, 3, operation = "/")
```

```
## [1] 1.666667
```

では "+"、 "-"、 "\*"、 "/"以外の値を `operation` に与えるとうなるだろうか。ためしに "^"を入れてみよう。

```
1 my_calc(5, 3, operation = "^")
```

```
## Error in my_calc(5, 3, operation = "^"): operation の値に使えるのは +, -, *, / のみです。
```

`stop()` 内に書いたエラーメッセージが表示され、処理が中止される。

---

## 練習問題

### 問 1

- 3つの6面サイコロ投げを考える。3つの目の和が15になるまでサイコロ投げを繰り返す。どのような目が出て、その合計はいくつか。そして、何回目のサイコロ投げかを表示するコードを
  - `for` ループを用いて書きなさい
  - `while` ループを用いて書きなさい
  - 上の2つのコードが同じ結果を表示することを確認したうえで、どちらの方がより効率的か考察しなさい。

```
## [1] "1 目のサイコロ投げの結果: 4, 2, 6 (合計: 12)"
```

```
## [1] "2 目のサイコロ投げの結果: 5, 4, 1 (合計: 10)"
```

```
## [1] "3 目のサイコロ投げの結果: 5, 6, 4 (合計: 15)"
```

## 問 2

- 以下のような長さ 5 の文字型ベクトル `cause` がある。「肥満の原因是 XX でしょう。」というメッセージを表示するコードを書きなさい。ただし、「XX」の箇所にはベクトルの各要素を代入すること。表示されるメッセージは 5 個でなければならない。

```
1 cause <- c("喫煙", "飲酒", "食べすぎ", "寝不足", "ストレス")
```

```
## [1] "肥満の原因是喫煙でしょう。"  
## [1] "肥満の原因是飲酒でしょう。"  
## [1] "肥満の原因是食べすぎでしょう。"  
## [1] "肥満の原因是寝不足でしょう。"  
## [1] "肥満の原因是ストレスでしょう。"
```

- 長さ 4 の文字型ベクトル `effect` を以下のように定義する。このとき、「YY の原因是 XX でしょう。」というメッセージを表示するコードを書きなさい。ただし、「YY」には `effect` の要素を、「XX」には `cause` の要素を代入すること。出力されるメッセージは 20 個でなければならない。

```
1 effect <- c("肥満", "腹痛", "不人気", "金欠")
```

- 長さ 3 の文字型ベクトル `solution` を以下のように定義する。このとき、「YY の原因是 XX ですが、ZZ 改善されるでしょう。」というメッセージを出力するコードを書きなさい。ただし、「YY」には `effect` の要素を、「XX」には `cause` の要素を、「ZZ」には `solution` の要素を代入すること。出力されるメッセージは 60 個でなければならない。

```
1 solution <- c("この薬を飲めば", "一日一麺すれば", "神戸大に登山すれば")
```

## 問 3

- 長さ 2 の numeric ベクトル `data` について考える。条件分岐を用いて `data` の 1 番目の要素 (`data[1]`) が 2 番目の要素 (`data[2]`) より大きい場合、1 番目の要素と 2 番目の順番を逆転させる条件分岐を作成せよ。たとえば、`data <- c(5, 3)` なら、条件分岐後の `data` の値が `c(3, 5)` になるようにするコードを書きなさい。

## 第 11 章

# 関数の自作

### 11.1 関数の作成

これまで `class()` や、`sum()`、`print()` など、様々な**関数 (functions)** を使ってきました。「R で起こるあらゆることは関数の呼び出しである (Everything that happens in R is a function call)」(Chambers [2016, p.4]) と言われるように、「R を使う」ということは、「R の関数を使う」ということである。

関数は関数名 () ように括弧とセットで表記されることが多い。1 つの関数でも、括弧の中に異なる引数を指定することで、さまざまな結果を出すことができる。例として、第 6 章でも説明した `seq()` 関数について考えてみよう。この関数を使うと、一連の数字からなるベクトルを作ることができる。`from` で数列の初項を、`to` で数列の最終項を指定し、`by` で要素間の差（第 2 要素は第 1 要素に `by` を加えた値になる）を指定するか、`length.out` で最終的にできるベクトルの要素の数を指定する。

R を含むプログラミングにおける関数は、何かを入力すると何かを出力する箱のようなものである。関数を使うだけなら、関数という箱の中で何が起こっているかを理解する必要はない。例えば、`mean()` という関数に実数のベクトルを入力すると、平均値を実数として返すということを知っていれば、`mean()` が具体的にどのような計算を行っているかを知らなくても、実用上は問題ない。つまり、関数をブラックボックスとして扱うことができる。

このように1つの関数でも指定する内容は、`by` になったり `length.out` になったりする。`by` や `length.out`、`from`、`to` などのように、関数で指定する対象になっているもののことを **仮引数 (parameter)** と呼ぶ。また、`by = 1` の1や、`length.out = 10` の10のように、仮引数に実際に渡される値のことを**実引数 (argument)** と呼ぶ。特に誤解が生じないと思われる場合には、仮引数と実引数を区別せずに**引数 (ひきすう)** と呼ぶ。Rでは、1つの関数で使う引数の数が複数あることが多いので、**仮引数を明示する**習慣を身につけたほうがよい。ただし、第1引数（関数で最初に指定する引数）として必ず入力すべきものは決められている場合がほとんどなので、第1引数の仮引数は省略されることが多い。仮引数が省略される代わりに、第1引数の実引数はほぼ必ず入力する（いくつかの例外もある）。

関数とは()内の引数のデータを関数内部の手続きに沿って処理し、その結果を返すものです。あるデータを引数として受け付ける関数であれば、その引数を変えるだけで「先ほどとは異なるデータに対し、同じ処理を行う」ことが可能となり、コーディングの労力が省けます。

ここでは、ベクトル `c(1, 2, 3, 4, 5)` の総和を計算する方法について考えてみましょう。まず、1つ目は単純に足し算をする方法があります。

```
1 1 + 2 + 3 + 4 + 5
```

```
## [1] 15
```

他にも反復処理を使うことも可能です。とりわけ、`1:100`のようなベクトルを1つ目の方法で記述するのは時間の無駄でしょう。`for()`文を使った方法は以下のようになります。

```
1 Result <- 0
2
3 for (i in 1:5) {
4   Result <- Result + i
5 }
6
7 Result
## [1] 15
```

数個の数字を足すだけなら方法1の方が楽でしょうし、数百個の数字の場合は方法2の

方が効率的です。それでもやはり `sum()` 関数の方が数倍は効率的です。また、関数を使うことで、スクリプトの量をへらすこともできます。1 から 100 までの総和なら、方法 2 の `for (i in 1:5)` を `for (i in 1:100)` に変えることで対応可能ですが、それでも全体としては数行のコードで書かなくてもなりません。一方、`sum(1:100)` なら一行で済みます。

`sum()` はまだマシな方です。たとえば、回帰分析をしたい場合、毎回回帰分析のコードを一から書くのはあまりにも非効率的です。`lm()` 関数を使うと、データや回帰式などを指定するだけで、一連の作業を全て自動的に行い、その結果を返してくれます。中には回帰式の係数も計算してくれますが、他にも残差や決定係数なども計算してくれます。その意味で、`lm()` という関数は複数の機能を一つの関数としてまとめたものでもあります。

これらの関数は既に R 開発チームが書いた関数ですが、ユーザー側から関数を作成することも可能です。長いコードを書く、同じ作業を繰り返す場合、関数の作成はほぼ必須とも言えます。ここではまず、簡単な関数を作成してみましょう。与えられた数字を二乗し、その結果を返す `myPower()` 関数を作ってみましょう。

```
1 myPower <- function(x) {  
2   x^2  
3 }  
  
1 # 引数が一つしかないので、myPower(24) も可能  
2 myPower(x = 24)  
  
## [1] 576
```

それではコードを解説します。関数は以下のように定義されます。

```
1 関数名 <- function (引数名) {  
2   処理内容  
3 }
```

まず、関数名を `myPower` とし、それが関数であることを宣言します。そして、この関数の引数の名前は `x` とします。それが

```
1 myPower <- function (x)
```

の部分です。続いて、{}内に処理内容を書きます。今回は  $x^2$  であり、これは  $x$  の 2 乗を意味します。そして、関数の処理結果が返されます、{}内の最後の行が結果として返されます。 $x^2$  の部分は `return(x^2)` と書き換えることも可能です。`return()` は「この結果を返せよ」という意味の関数ですが、返す結果が最後の行である場合、省略可能であり、Hadely 先生もこのような書き方を推奨しています。

それでは、もうちょっと複雑な関数を作成してみましょう。ベクトルを引数とし、その和を計算する `mySum()` という関数です。要するに `sum()` 関数を再現したものです。

```
1 # mySum 関数を定義し、引数は x のみとする
2 mySum <- function(x) {
3   # 結果を格納するベクトル Result を生成し、0を入れておく
4   Result <- 0
5
6   # x の要素を i に一つずつ入れながら反復処理
7   for (i in x) {
8     # Result に既存の Result の値に i を足した結果を上書きする
9     Result <- Result + i
10  }
11
12  # Result を返す
13  Result
14 }
```

```
1 mySum(1:5)
```

```
## [1] 15
```

普通の `sum()` 関数と同じ動きをする関数が出来上りました。よく見ると、上で説明した総和を計算する方法 2 のコードを丸ごと関数内に入っているだけです。変わったところがあるとすれば、`for()` 文であり、`for (i in 1:5)` が `for (i in x)` に変わっただけです。この  $x$  は `mySum <- function (x)` の  $x$  を意味します。このように関数を一

回作成しておくと、これからは総和を出す作業を 1 行に短縮することができます。

この `mySum()` ですが、一つ問題があります。それは `x` に欠損値が含まれている場合、結果が `NA` になることです。

```
1 mySum(c(1, 2, 3, NA, 5))
```

```
## [1] NA
```

実際、R 内蔵関数である `sum()` も同じですが、`sum()` には `na.rm =` というもう一つの引数があり、これを `TRUE` にすることで欠損値を除いた総和が計算できます。つまり、関数は複数の引数を持つことができます。それでは、`mySum()` を改良してみましょう。ここにも `na.rm` という関数を追加し、`na.rm` 引数が `TRUE` の場合、`x` から欠損値を除いた上で総和を計算するようにしましょう。

```
1 mySum <- function(x, na.rm = FALSE) {
2   if (na.rm == TRUE) {
3     x <- x[!is.na(x)]
4   }
5
6   Result <- 0
7
8   for (i in x) {
9     Result <- Result + i
10  }
11
12  Result
13 }
```

```
1 mySum(c(1, 2, 3, NA, 5))
```

```
## [1] NA
```

```
1 mySum(c(1, 2, 3, NA, 5), na.rm = FALSE)
```

```
## [1] NA
```

```
1 mySum(c(1, 2, 3, NA, 5), na.rm = TRUE)  
## [1] 11
```

変わったところは、まず `function (x)` が `function (x, na.rm = FALSE)` になりました。これは `x` と `na.rm` の引数が必要であるが、`na.rm` のデフォルト値は `FALSE` であることを意味します。デフォルト値が指定されている場合、関数を使用する際、その引数は省略できます。実際、`sum()` 関数の `na.rm` 引数も `FALSE` がデフォルトとなっており、省略可能となっています。

次は最初に条件分岐が追加されました。ここでは `na.rm` が `TRUE` の場合、`x` から欠損値を抜いたベクトルを `x` に上書きするように指定しました。もし、`FALSE` ならこの処理は行いません。

これで R 開発チームが作成した `sum()` 関数と同じものが出来上がりました。それでは引数の順番について簡単に解説し、もうちょっと複雑な関数を作ってみましょう。引数の順番は基本的に `function()` の () 内で定義した順番であるなら、引数名を省略することも可能です。

```
1 mySum(c(1, 2, 3, NA, 5), TRUE)  
## [1] 11
```

ただし、順番を逆にすると、以下のようにわけのわからない結果が返されます。

```
1 mySum(TRUE, c(1, 2, 3, NA, 5))  
  
## Warning in if (na.rm == TRUE) {: the condition has length > 1 and only the first  
## element will be used  
  
## [1] 1
```

任意の順番で引数を指定する場合、引数名を指定する必要があります。

```
1 mySum(na.rm = TRUE, x = c(1, 2, 3, NA, 5))  
## [1] 11
```

自分で関数を作成し、他の人にも使ってもらう場合、引数名、順番、デフォルト値を適切に設定しておくことも大事です。

## 11.2 ちょっと複雑な関数

それではちょっとした遊び心を込めた関数を作ってみましょう。その名もドラクエ戦闘シミュレーターです。

以下はドラクエ 11 のダメージ公式です。

- ダメージの基礎値 = (攻撃力 / 2) - (守備力 / 4)
  - 0 未満の場合、基礎値は 0 とする
- ダメージの幅 = (ダメージの基礎値 / 16) + 1
  - 端数は切り捨てます (floor() 関数使用)

ダメージの最小値は「ダメージの基礎値 - ダメージの幅」、最大値は「ダメージの基礎値 - ダメージの幅」となります。この最小値が負になることもあります、その場合は 0 扱いになります。実際のダメージはこの範囲内でランダムに決まります (runif() 関数使用)。

```
1 # DQ_Attack 関数を定義
2 DQ_Attack <- function(attack, defence, hp, enemy) {
3   # 引数一覧
4   # attack: 勇者の力 + 武器の攻撃力 (長さ 1 の数値型ベクトル)
5   # defence: 敵の守備力 (長さ 1 の数値型ベクトル)
6   # hp: 敵の HP (長さ 1 の数値型ベクトル)
7   # enemy: 敵の名前 (長さ 1 の文字型ベクトル)
8
9   # ダメージの基礎値
10  DefaultDamage <- (attack / 2) - (defence / 4)
11  # ダメージの基礎値が負の場合、0 とする
12  DefaultDamage <- ifelse(DefaultDamage < 0, 0, DefaultDamage)
13  # ダメージの幅
14  DamageWidth <- floor(DefaultDamage / 16) + 1
15
```

```
16 # ダメージの最小値
17 DamageMin      <- DefaultDamage - DamageWidth
18 # ダメージの最小値が負の場合、0 とする
19 DamageMin      <- ifelse(DamageMin < 0, 0, DamageMin)
20 # ダメージの最大値
21 DamageMax      <- DefaultDamage + DamageWidth
22
23 # 敵の残り HP を格納する
24 CurrentHP      <- hp
25
26 # 残り HP が 0 より大きい場合、以下の処理を繰り返す
27 while (CurrentHP > 0) {
28   # ダメージの最小値から最大値の間の数値を 1 つ無作為に抽出する
29   Damage <- runif(n = 1, min = DamageMin, max = DamageMax)
30   # 小数点 1 位で丸める
31   Damage <- round(Damage, 0)
32   # 残りの HP を更新する
33   CurrentHP <- CurrentHP - Damage
34   # メッセージを表示
35   print(paste0(enemy, "に", Damage, "のダメージ!!"))
36 }
37
38 # 上記の反復処理が終わったら勝利メッセージを出力
39 paste0(enemy, "をやっつけた！")
40 }
```

初めて見る関数が 3 つありますね。まず、`floor()` 関数は引数の端数は切り捨てる関数です。たとえば、`floor(2.1)` も `floor(2.6)` も結果は 2 です。続いて、`runif()` 関数は指定された範囲の一様分布から乱数を生成する関数です。引数は生成する乱数の個数 (n)、最小値 (min)、最大値 (max) の 3 つです。`runif(5, 3, 10)` なら最小値 3、最大値 10 の乱数を 5 個生成するという意味です。正規分布は平均値周辺の値が生成されやすい一方、一様分布の場合、ある値が抽出される確率は同じです。最後に `round()` 関数は

四捨五入の関数です。引数は 2 つあり、1 つ目の引数は数値型のベクトルです。2 つ目は丸める小数点です。たとえば、`round(3.127, 1)` の結果は 3.1 であり、`round(3.127, 2)` の結果は 3.13 となります。

それでは「ひのきのぼう」を装備したレベル 1 の勇者を考えてみましょう。ドラクエ 5 の場合、Lv1 勇者の力は 11、「ひのきのぼう」の攻撃力は 2 ですので、攻撃力は 13 です。まずは定番のスライムから突ってみましょう。スライムの HP と守備力は両方 7 です。

```
1 DQ_Attack(13, defence = 7, hp = 7, "スライム")
```

```
## [1] "スライムに 4 のダメージ!!"
## [1] "スライムに 4 のダメージ!!"
## [1] "スライムをやっつけた!"
```

まあ、こんなもんでしょう。それではスライムナイト (=ピエール) はどうでしょう。スライムナイトの HP のは 40、守備力は 44 です。

```
1 DQ_Attack(13, defence = 44, hp = 40, "スライムナイト")
```

```
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
```





```

## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"
## [1] "スライムナイトに 0 のダメージ!!"
## [1] "スライムナイトに 1 のダメージ!!"

## [1] "スライムナイトをやっつけた！"

```

これだと「なんと スライムナイトが おきあがりなかまに なりたそうに こちらを みている！」のメッセージを見る前に勇者ご一行が全滅しますね。エスターク (HP: 9000 / 守備力: 250) は計算するまでもないでしょう…

以上の関数に条件分岐を追加することで「かいしんの いちげき！」を入れることもできますし<sup>1)</sup>、逆に敵からの攻撃を計算して誰が先に倒れるかをシミュレーションすることも可能でしょうね。色々遊んでみましょう。

### 11.3 関数 in 関数

当たり前かも知れませんが、自作関数を他の自作関数に含めることもできます。ここでは乱数を生成する関数を作ってみましょう。パソコンだけだと完全な乱数を作成することは不可能ですが<sup>2)</sup>、乱数に近いものは作れます。このようにソフトウェアで生成された乱数は擬似乱数と呼ばれ、様々なアルゴリズムが提案されています。天才、フォン・ノイマン大先生も乱数生成では天才ではなかったらしく、彼が提案した平方採中法 (middle-square method) は使い物になりませんので、ここではもうちょっとマシな方法

<sup>1)</sup> 一般的に会心の一撃が出る確率は約 3.125% (=1/32) または約 1.563% (=1/64) と言われています。

<sup>2)</sup> 特殊なハードウェアを使えば周囲の雑音や電波などのノイズから乱数を生成することも可能です。

である線形合同法 (linear congruential generators; 以下、LCG) を採用します。平方採中法よりは式が複雑ですが、それでも非常に簡単な式で乱数が生成可能であり、Rによる実装に関しては平方採中法より簡単です。ちなみに、線形合同法にも様々な問題があり、Rのデフォルトは広島大学の松本眞先生と山形大学の西村拓士先生が開発しましたメルセンヌ・ツイスター (Mersenne twister) を採用しています。それでは、LCGのアルゴリズムから見ましょう。

乱数の列があるとし、 $n$  番目の乱数を  $X_n$  とします。この場合、 $X_{n+1}$  は以下のように生成されます。

$$X_{n+1} = (aX_n + c) \bmod m$$

$\bmod$  は余りを意味し、 $5 \bmod 3$  は 5 を 3 で割った際の余りですので、2 となります。 $a$  と  $c$ 、 $m$  は以下の条件を満たす任意の数です。

$$0 < m, \quad (11.1)$$

$$0 < a < m, \quad (11.2)$$

$$0 \leq c < m. \quad (11.3)$$

ベストな  $a$ 、 $c$ 、 $m$  も決め方はありませんが、ここでは Turbo C の設定を真似て  $a = 22695477$ 、 $c = 1$ 、 $m = 2^{32}$  をデフォルト値として設定します<sup>3)</sup>。そして、もう一つ重要なのが最初の数、つまり  $X_n$  をどう決めるかですが、これは自由に決めて問題ありません。最初の数 ( $X_0$ ) はシード (seed) と呼ばれ、最終的には使わない数字となります。それでは `seed` という引数からある乱数を生成する関数 `rng_number()` を作ってみましょう。

```

1  rng_number <- function(seed, a = 22695477, c = 1, m = 2^32) {
2      (a * seed + c) %% m
3  }
```

簡単な四則演算のみで構成された関数ですね。ちなみに `%%` は余りを計算する演算子です。とりあえず、`seed` を 12345 に設定し、一つの乱数を生成してみましょう。

<sup>3)</sup> ANSI C 標準の場合、 $a = 1103515245$ 、 $c = 12345$ 、 $m = 2^{31}$  です。

```
1  rng_number(12345)
```

```
## [1] 1002789326
```

かなり大きい数字が出ました。ちなみに線形合同法で得られる乱数の最大値は  $m$ 、最小値は 0 です。次は、今回得られた乱数  $1.0027893 \times 10^9$  を新しい `seed` とし、新しい乱数を作ってみましょう。

```
1  rng_number(1002789326)
```

```
## [1] 3785644968
```

この作業を繰り返すと、(疑似) 乱数の数列が得られます。続いて、この作業を `n` 回繰り返し、長さ `n` の乱数ベクトルを返す関数 `LCG` を作ってみましょう。いかがコードになります。

```
1  LCG <- function(n, seed, a = 22695477, c = 1, m = 2^32) {
2    rng_vec <- rep(NA, n + 1) # seed も入るので長さ n+1 の空ベクトルを生成
3    rng_vec[1] <- seed # 1 番目の要素に seed を入れる
4
5    # i に 2 から n+1 までの値を順次的に投入しながら、反復処理
6    for (i in 2:(n+1)) {
7      # rng_vec の i 番目に i-1 番目の要素を seed にした疑似乱数を格納
8      rng_vec[i] <- rng_number(rng_vec[i - 1], a, c, m)
9    }
10
11  rng_vec <- rng_vec[-1] # 1 番目の要素 (seed) を捨てる
12  rng_vec <- rng_vec / m # 最小値 0、最大値 1 になるように、m で割る
13
14  rng_vec # 結果を返す
15 }
```

それでは、詳細に解説します。

- 1 行目: 関数 `LCG` を定義し、必要な引数として `n` と `seed` を設定する。

- 2 行目: 結果を格納する空ベクトル `rng_vec` を生成。ただし、1 番目には `seed` が入るので、長さを `n+1` とする。
- 3 行目: `rng_vec` の 1 番目に `seed` を格納する。
- 6 行目: 疑似乱数を `n` 回生成し、格納するように反復作業を行う。任意の変数は `i` とし、`i` に代入される値は 2 から `n+1` までである。
- 8 行目: `rng_vec` の `i-1` 番目要素を `seed` にした疑似乱数を生成し、`rng_vec` の `i` 番目に格納する。1 回目の処理だと `i=2` であるため、`rng_vec[1]` (= `seed`) を `seed` にした疑似乱数が生成され、`rng_vec[2]` に格納される。
- 11 行目: `rng_vec` の 1 番目の要素は `seed` であるため、捨てる。
- 12 行目: 乱数が最小値 0、最大値 1 になるように、調整する。具体的には得られた乱数を `m` (デフォルトは  $2^{32}$ ) で割るだけである。
- 14 行目: 結果ベクトルを返す。

それでは、`seed` を 19861008 とした疑似乱数 10000 個を生成し、`LCG_Numbers` という名のベクトルに格納してみましょう。結果を全て表示させるのは無理があるので、最初の 20 個のみを確認してみます。

```
1 LCG_Numbers <- LCG(10000, 19861008)
2 head(LCG_Numbers, 20)
```

```
## [1] 0.58848246 0.09900449 0.21707060 0.89462981 0.23421890 0.72341249
## [7] 0.55965400 0.59552685 0.96594972 0.69050965 0.98383344 0.20136551
## [13] 0.25856502 0.37569497 0.50086451 0.03986446 0.89291806 0.24760102
## [19] 0.27912510 0.24493750
```

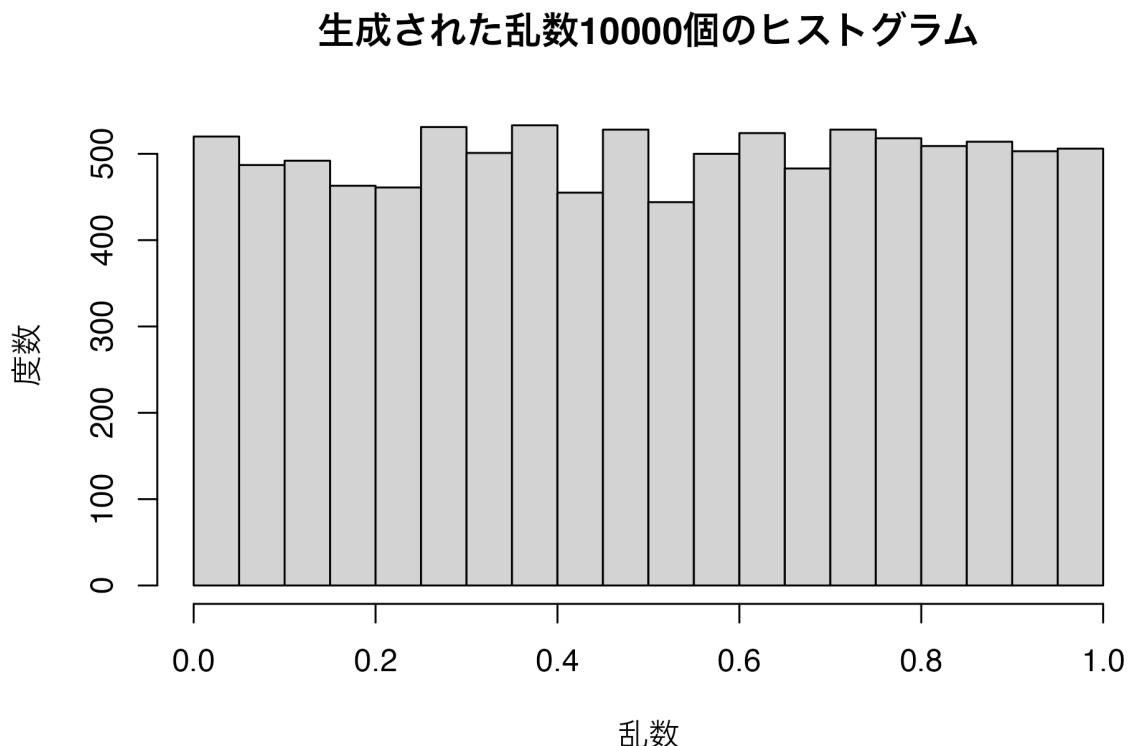
正直、これが乱数かどうかは見るだけでは分かりませんね。簡単な確認方法としては、これらの乱数列のヒストグラムを見れば分かります。得られた数列が本当に乱数 (に近いもの) なら、その分布は一様分布に従っているからです。ただし、一様分布に従っていることが乱数を意味するものではありません。

可視化については第 17 章以降で解説しますが、ここでは簡単に `hist()` 関数を使ってみましょう。必要な引数は numeric 型のベクトルのみです。以下のコードにある `xlab =` などはラベルを指定する引数ですが、省略しても構いません。

```

1 hist(LCG_Numbers, xlab = "乱数", ylab = "度数",
2      main = "生成された乱数10000個のヒストグラム")

```



ややギザギザしているように見えますが<sup>4)</sup>、これなら一様分布だと考えて良いでしょう。

## 練習問題

### 問 1

- 長さ 2 の numeric ベクトル Data について考える。条件分岐を用いて Data の 1 番目の要素 (Data[1]) が 2 番目の要素 (Data[2]) より大きい場合、1 番目の要素と 2 番目の順番を逆転させる条件分岐を作成せよ。たとえば、Data <- c(5, 3) な

<sup>4)</sup> 1万個ではなく、100万個、1億個など乱数を生成すればするほど、一様分布に近づきます。

ら、条件分岐後の Data の値が `c(3, 5)` になること。

## 問 2

- 与えられた正の実数の平方根を求める `my_sqrt()` を作成する。平方根を計算する方法は Hero of Alexandria の近似法<sup>5)</sup>を使用する。
  - 平方根を求める `x`、任意の初期値 `g`、非常に小さい正の実数 `e` を入力する。`e` の既定値は 0.001 とする。
  - `g` の 2 乗を計算し、`x - g^2` の絶対値を `gap` とする (`gap` の初期値は `Inf` とする)。
  - `gap` が `e` より小さい場合、`g` を `x` の平方根として返す。
  - `gap` が `e` より大きい場合、`g + (x / g)` を新しい `g` とする。
  - 2 へ戻る。
- `while()` 関数を使えば簡単である。

```
1 # R の sqrt() 関数
2 sqrt(29)

## [1] 5.385165

1 # 計算例 1
2 my_sqrt(29, 5) # 29 の平方根。初期値は 5

## [1] 5.385185

1 # 計算例 2
2 my_sqrt(29, 5, e = 0.00001) # 29 の平方根。初期値は 5。e は 0.00001

## [1] 5.385165

1 # 計算例 3
2 my_sqrt(-3, 5, e = 0.00001) # あえてエラーを出してみる

## Error in my_sqrt(-3, 5, e = 1e-05): x は正の実数でなければなりません。
```

<sup>5)</sup> 実はこの方法はアレクサンドリアのヘロンが考案したものではないと言われており、もっと昔の古代バビロニアで使用されたと推測される。

### 問3

- 問3ではベクトルの1番目要素と2番目の要素を比較し、前者が大きい場合において順番を入れ替える条件分岐を行った。これを長さ3以上のベクトルにも適用したい。長さ4のnumeric型ベクトルDataがある場合の計算手順について考えてみよう。
  1. Data[1]とData[2]の要素を比較し、Data[1] > Data[2]の場合、入れ替える。
  2. Data[2]とData[3]の要素を比較し、Data[2] > Data[3]の場合、入れ替える。
  3. Data[3]とData[4]の要素を比較し、Data[3] > Data[4]の場合、入れ替える。この段階でData[4]はDataの最大値が格納される。
  4. 続いて、Data[1]とData[2]の要素を比較し、Data[1] > Data[2]の場合、入れ替える。
  5. Data[2]とData[3]の要素を比較し、Data[2] > Data[3]の場合、入れ替える。この段階でData[3]にはDataの2番目に大きい数値が格納される。
  6. 最後にData[1]とData[2]の要素を比較し、Data[1] > Data[2]の場合、入れ替える。ここで並び替えは終了
- ヒント:** 2つのfor()文が必要となる。これは「バブルソート」と呼ばれる最も簡単なソートアルゴリズムである。イメージとしては、長さNのベクトルの場合、n番目の要素とn+1番目の要素を比較しながら、大きい方を後ろの方に追い込む形である。最初は、Data[4]最後の要素に最大値を格納し、続いて、Data[3]に次に大きい数値を、Data[2]に次に大きい数値を入れる仕組みである。
  - 最初はData[1]とData[2]、Data[2]とData[3]、Data[3]とData[4]の3ペアの比較を行う。
  - 続いて、Data[1]とData[2]、Data[2]とData[3]の2ペアの比較を行う
  - 最後に、Data[1]とData[2]の1ペアの比較を行う
  - つまり、外側のfor()文は「Dataの長さ-1」回繰り返すことになる。
  - 内側のfor()文は3, 2, 1ペアの比較を行うことになる。

### 問4

- 「問4」の練習問題で作成したコードを関数化せよ。関数名はmySort()であり、引数は長さ2以上のnumeric型ベクトルのみである。引数名は自由に決めて良い

いし、`data` や `x` などを使っても良い。

### 問 5

- 既に作成した `DQ_Attack` 関数を修正してみよう。今の関数は通常攻撃のみであるが、稀に「会心の一撃」が発生するように修正する。
  - 会心の一撃が発生する確率は  $1/32$  とする (Hint: 0 から 1 の間の乱数を生成し、その乱数が  $1/32$  以下であるか否かで条件分岐させる)。乱数の生成は `runif()` または、自作関数の `LCG()` でも良い。
  - 会心の一撃のダメージの最小値は攻撃力の 95%、最大値は 105% とする。
  - 会心の一撃が発生したら、"かいしんのいちげき！スライムに 10 のダメージ！" のようなメッセージを出力させること。

### 問 6

- 与えられたベクトル `x` から `n` 個の要素を無作為に抽出する関数、`mySample()` を定義せよ。`mySample()` の引数は `x` と `n`、`seed` とし、`x` は長さ 1 以上のベクトルとする。`n` は長さ 1 の整数型ベクトルである。
  - 以下の場合、エラーメッセージを出力し、処理を中止させること。
    - `n` と `seed` が長さ 1 ベクトルでない場合、
    - `seed` が numeric 型でない場合
    - `n` が整数でない場合 (`floor(n) == n` で判定)
  - ヒント:** `LCG()` 関数を用いて乱数を生成した場合、生成された擬似乱数は 0 以上 1 以下である。ベクトル `x` の長さを `length(x)` とした場合、擬似乱数に `length(x)` を掛けると、擬似乱数は 0 以上 `length(x)` 以下になる。これらの値を切り上げると (`ceiling()` 関数)、その値は 1 以上 `length(x)` 以下の整数となる。



# 第 III 部

## データハンドリング



## 第 12 章

# データハンドリング [基礎編: 抽出]

ここでは比較的綺麗に整形されているデータフレームを扱う方法について考えます。ここでいう「比較的綺麗なデータ」とは、すぐに分析に使えるレベルのデータを意味します。したがって、ここではデータ内の値を変更するような作業は行いません。基本的に分析しやすくなるように列の順番を替えたり、特定の列や行のみを抽出したり、データの順番を並び替える作業に注目します。

本章では以下の 3 つの内容を中心に解説します。

1. パイプ演算子 (`%>%`) に慣れる
2. 特定の行と列の抽出
3. データのソート

本章で学習する内容でデータを加工した場合、得られる結果物は元のデータの一部 (subset) となります。データの中身の値を変えたり、新しい列を追加したり、平均値などの記述統計量をまとめたりする方法については次の第 13 章で解説します。

---

### 12.1 データハンドリングと tidyverse

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and

data structures. (Tidyverse ホームページから)

Tidyverse とはデータサイエンスのために考案された、強い信念と思想に基づいた R パッケージの集合です。Tidyverse に属するパッケージは思想、文法およびデータ構造を共有しています。Tidyverse の中核をなすパッケージは{ggplot2} (第 17、18、19、20 章)、{dplyr} (第 12、13 章)、{tidyr} (第 15 章)、{readr} (第 7.1 章)、{purrr} (第 27 章)、{tibble} (第 9.4 章)、{stringr} (第 16 章)、{forcats} (第 14 章) があり、このパッケージを支える数十のパッケージが含まれています。これらのパッケージは個別に読み込む必要はなく、{tidyverse}パッケージを読み込むだけで十分です。

```
1 pacman::p_load(tidyverse)
```

R におけるデータハンドリング (データ操作) の標準が{dplyr}と{tidyr}中心となり、文字列の処理は{stringr}、factor 型の操作は{forcats}、大量のモデルを自動的に分析し、結果を処理するためには{tibble}と{purrr}、可視化は{ggplot2}が主に使われています。これらのパッケージ間、あるいはパッケージ内におけるオブジェクトのやり取りは全てパイプ演算子を通じて行われています。また、これらのパッケージは整然データ (tidydata) を想定するか、整然データの作成するに特化しています。

tidyverse の思想に基づいた「tidyverse 流のコーディング」は現在の R そのものと言っても過言ではありません。ただし、tidyverse じゃないと出来ないデータハンドリング、可視化などはありません。tidyverse はデータサイエンスの思想に近いものであり、「異なる思想を持っているからこのような分析はできない」といったものはありません。tidyverse という概念が提唱される前にも R は存在し、tidyverse 無き世界で今と同じことをやってきました。ただし、tidyverse 流で書かれた R コードは可読性が優れ、コードを書く手間も短くなります。tidyverse の考え方が R のける「標準語」として定着しつつあるのは否めない事実であり、学習する誘引としては十分すぎるでしょう。

## 12.2 パイプ演算子 (%>%)

{dplyr}パッケージを利用する前にパイプ演算子について説明します。パイプ演算子は{dplyr}に含まれている演算子ではなく、magrittr という別のパッケージから提供され

る演算子ですが、{tidyverse}パッケージを読み込むと自動的に読み込まれます。パイプ演算子は `x %>% y()` のような書き方となりますが、これは「`x`を`y()`の第一引数として渡す」ことを意味します。`x`の部分はベクトルやデータフレームのようなオブジェクトでも、関数でも構いません。なぜなら、関数から得られた結果もまたベクトルやデータフレームといったものになるからです。つまり、`x() %>% y()` という使い方も可能です。そして、パイプは無限に繋ぐこともできます。「データ `df` を関数 `x()` で処理をし、その結果をまた関数 `y()` で処理する」ことは、パイプを使うと `df %>% x() %>% y()` のような書き方となります。

たとえば、「`paste(3, "+", 5, "=", 8)`を実行し、その結果を `rep()` 関数を使って3回複製し、それを `print()` を使って出力する」コードを考えてみましょう。方法としては2つ考えられます。まずは、それぞれの処理を別途のオブジェクトに格納する方法です。そして二つ目は関数の中に関数を使う方法です。

```
1 # 方法1: 一関数一オブジェクト
2 Result1 <- paste(3, "+", 5, "=", 8)
3 Result2 <- rep(Result1, 3)
4 print(Result2)

## [1] "3 + 5 = 8" "3 + 5 = 8" "3 + 5 = 8"

1 # 方法2: 関数の中に関数
2 print(rep(paste(3, "+", 5, "=", 8), 3))

## [1] "3 + 5 = 8" "3 + 5 = 8" "3 + 5 = 8"
```

どれも結果は同じです。コードを書く手間を考えれば、後者の方が楽かも知れませんが、可読性があまりよくありません。一方、前者は可読性は良いものの、コードも長くなり、オブジェクトを2つも作ってしまうのでメモリの無駄遣いになります。

コードの可読性と書く手間、両方を満足する書き方がパイプ演算子 `%>%` です。まずは、例から見ましょう。

```
1 # %>% を使う
2 paste(3, "+", 5, "=", 8) %>% rep(3) %>% print()
```

```
## [1] "3 + 5 = 8" "3 + 5 = 8" "3 + 5 = 8"
```

まず、結果は先ほどと同じです。それではコードの説明をしましょう。まずは、`paste(3, "+", 5, "=", 8)` を実行します。そしてその結果をそのまま `rep()` 関数の第一引数として渡されます。つまり、`rep(paste(3, "+", 5, "=", 8), 3)` になるわけです。ここでは `rep(3)` と書きましたが、第一引数が渡されたため、3 は第二引数扱いになります（パイプ演算子前のオブジェクトを第二、三引数として渡す方法は適宜説明します。）。そして、これをまた `print()` 関数に渡します。結果としては `print(rep(paste(3, "+", 5, "=", 8), 3))` となります。

関数を重ねると読む順番は「カッコの内側から外側へ」になりますが、パイプ演算子を使うと「左（上）から右（下）へ」といったより自然な読み方が可能になります。また、以下のコードのように、パイプ演算子後に改行を行うことでより読みやすいコードになります。これからはパイプ演算子の後は必ず改行をします。

```
1 # 改行 (+ 字下げ) したらもっと読みやすくなる
2 paste(3, "+", 5, "=", 8) %>%
3   rep(3) %>%
4   print()
```

パイプ演算子を使わない方法は図 12.1 のようにイメージできます。一回の処理ごとに結果を保存し、それをまた次の処理時においてデータとして使うイメージです。

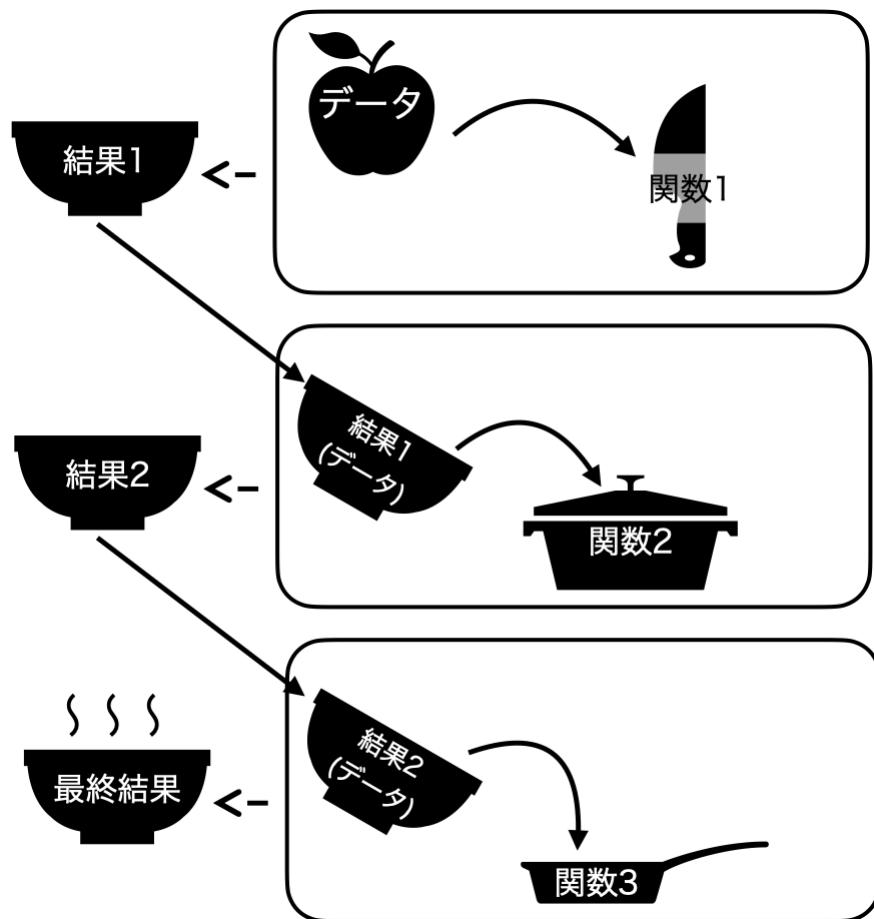


図 12.1: パイプ演算子を使わない場合

一方、図 12.2 はパイプ演算子を使う場合のプロセスです。処理後の結果を保存せず、すぐに次のプロセスに渡すことで、メモリ (図だとボウル) や時間、コードの無駄を減らすことができます。もちろん、図 12.1 の結果 1 を使って色々試してみたい場合は、一旦結果 1 までは格納し、適宜引き出して使った方が効率的でしょう。パイプ演算子はたしかに便利で、「今どき」の R の書き方を象徴するようなものですが、一つの結果を出すまであまりにも多くのパイプ演算子を使うことはあ望ましくありません。

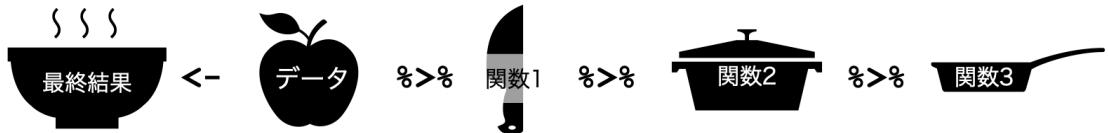


図 12.2: パイプ演算子を使う場合

データハンドリングもこれと同様に、様々な作業を順に沿って行う必要があります。例えば、「(1) 列を選択して、(2) 欠損値を含む列を除去して、(3) ある変数の値を 100 倍にして、(4) ある変数の値が小さい行から大きい順へ並び替える」といった手順です。これらの作業はパイプ演算子を使えば、スムーズに行うことが可能です。

### 12.3 列の抽出

それでは今回の実習用データを読み込みましょう。Ramen.csv には「ぐるなび」から取得したラーメン屋 6292 店舗の情報が入っています。具体的には東京、神奈川、千葉、埼玉、大阪、京都、兵庫、奈良、和歌山それぞれ都府県にあるラーメン屋の中から最大 1000 店舗の情報を抽出したものです。東京都は、ぐるなびに登録したラーメン屋が 3000 店舗以上ですが、1000 店舗の基準はぐるなびの「おすすめ」の順で上位 1000 店舗となります。また、店側またはぐるなびが登録したカテゴリを基準に抽出したため、実際はラーメン屋ではないにもかかわらずラーメン屋としてデータ内に含まれている可能性があります。

まず、このデータを読み込み、`df` という名付けます。

```
1 df <- read_csv("Data/Ramen.csv")
```

データの中身を確認してみましょう。

```
1 df
```

```
## # A tibble: 6,292 x 14
##   ID      Name   Pref  Zipcode Latitude Longitude Line   Station  Walk   Bus   Ca
##   <chr>   <chr>  <chr> <dbl>     <dbl>     <dbl> <chr>   <chr>   <dbl>   <dbl> <dbl>
## 1 e5396~ 居酒屋~ 東京~ 1040031     35.7     140. 地下~ 銀座~  1
## 2 ~       3       NA     NA
```

```

## 2 gfeb6~ 本 格~ 東 京~ 1100005      35.7      140. 地 下~ 仲 御 徒
~      1     NA     NA
## 3 ggt59~ 食べ~ 東京~ 1250041      35.8      140. J R~ 金町駅      2     NA     NA
## 4 g1813~ 博 多~ 東 京~ 1920904      35.7      139. J R 八 王 子
~      1     NA     NA
## 5 ggww1~ まさ~ 東京~ 1500042      35.7      140. 地下~ 渋谷駅      7     NA     NA
## 6 gdzk5~ 完 全~ 東 京~ 1000013      35.7      140. 地 下~ 虎 ノ 門
~      3     NA     NA
## 7 ga2g2~ 鶏 そ~ 東 京~ 1760006      35.7      140. 西 武~ 江 古 田
~      2     NA     NA
## 8 gg9m1~ 宴 会~ 東 京~ 1010021      35.7      140. J R 秋 葉 原
~      4     NA     NA
## 9 gdvk2~ 中 国~ 東 京~ 1000006      35.7      140. J R 有 樂 町
~      1     NA     NA
## 10 gggb2~ 中国~ 東京~ 1140002      35.8      140. 地下~ 王子駅      2     NA     NA
## # ... with 6,282 more rows, and 3 more variables: Budget <dbl>, ScoreN <dbl>,
## #   Score <dbl>

```

1行目の# A tibble: 2,000 x 12 から、ケース数(店舗数)は2000、変数は12個あることが分かります。各変数の詳細は以下の通りです。

それではここからは df を用いた{dplyr}の様々な機能を紹介していきます。

### 12.3.1 特定の列を抽出する

まずは、データフレームから特定の列のみを残す、除去する方法について紹介します。たとえば、df から ID、Name、Pref、Score のみを残すとします。{dplyr}を使わない方法と{dplyr}の select() 関数を使った方法を紹介します。

```

1 # dplyr を使わない方法
2 df[, c("ID", "Name", "Pref", "Score")]

## # A tibble: 6,292 x 4
##       ID     Name     Pref     Score
##   <dbl>   <chr>   <chr>    <dbl>
## 1     1  1000001  東京  35.7
## 2     2  1000002  東京  35.8
## 3     3  1000003  東京  35.7
## 4     4  1000004  東京  35.7
## 5     5  1000005  東京  35.7
## 6     6  1000006  東京  35.7
## # ... with 6,286 more rows
## # and 4 more variables: Budget <dbl>, ScoreN <dbl>, Score <dbl>, ...

```

変数名	説明
ID	店舗 ID
Name	店舗名
Pref	店舗の所在地 (都府県)
Zipcode	店舗の郵便番号
Latitude	緯度
Longitude	経度
Line	最寄りの駅の路線
Station	最寄りの駅
Walk	最寄りの駅からの距離 (徒歩; 分)
Bus	最寄りの駅からの距離 (バス; 分)
Car	最寄りの駅からの距離 (車; 分)
Budget	平均予算 (円)
ScoreN	口コミの数
Score	口コミ評価の平均値

```

## <chr> <chr> <chr> <dbl>
## 1 e539604 居酒屋 龍記 京橋店 東
京都 NA
## 2 gfeb600 本格上海料理 新錦江 上野御徒町本店 東
京都 4.5
## 3 ggt5900 食べ飲み放題 × 中華ビストロ NOZOMI (のぞみ) 東
京都 NA
## 4 g181340 博多餃子軒 八王子店 タピオカ店 Bull Pulu (ブルブル) 併設 東
京都 NA
## 5 ggww100 まさ屋 渋谷店 東
京都 NA
## 6 gdzk500 完全個室 上海レストラン 檸檬 霞ヶ関ビル内店 東
京都 NA
## 7 ga2g202 鶏そば きらり 東
京都 NA

```

```
## 8 gg9m100 宴会個室 × 餃子酒場 北京飯店 秋葉原本店 東
京都 3.33
## 9 gdvk200 中国料理 宝龍 東
京都 2.5
## 10 gggb200 中国料理 天安門 東
京都 NA
## # ... with 6,282 more rows

1 # dplyr::select() を使う方法
2 # select(df, ID, Name, Pref, Score) でも OK
3 df %>%
4   select(ID, Name, Pref, Score)

## # A tibble: 6,292 x 4
##       ID     Name     Pref     Score
##   <chr>   <chr>   <chr>   <dbl>
## 1 e539604 居酒屋 龍記 京橋店 東
京都 NA
## 2 gfeb600 本格上海料理 新錦江 上野御徒町本店 東
京都 4.5
## 3 gg5900 食べ飲み放題 × 中華ビストロ NOZOMI (のぞみ) 東
京都 NA
## 4 g181340 博多餃子軒 八王子店 タピオカ店 Bull Pulu (ブルブル) 併設 東
京都 NA
## 5 ggww100 まさ屋 渋谷店 東
京都 NA
## 6 gdzk500 完全個室 上海レストラン 檸檬 霞ヶ関ビル内店 東
京都 NA
## 7 ga2g202 鶏そば きらり 東
京都 NA
## 8 gg9m100 宴会個室 × 餃子酒場 北京飯店 秋葉原本店 東
京都 3.33
## 9 gdvk200 中国料理 宝龍 東
```

```
京都 2.5
```

```
## 10 gggb200 中国料理 天安門 東
京都 NA
## # ... with 6,282 more rows
```

どれも結果は同じですが、`select()` 関数を使った方がより読みやすいコードになっているでしょう。むろん、`select()` 関数を使わぬ方がスッキリする方も知るかも知れません。実際、自分でパッケージなどを作成する際は `select()` を使わない場合が多いです。ただし、一般的な分析の流れでは `select()` の方がコードも意味も明確となり、パイプ演算子でつなぐのも容易です。

`select()` 関数の使い方は非常に簡単です。第一引数はデータフレームですが、パイプ演算子を使う場合は省略可能です。第二引数以降の引数はデータフレームの変数名です。つまり、ここには残す変数名のみを書くだけで十分です。

また、`select()` 関数を使って列の順番を変えることもできます。たとえば、`ID`、`Pref`、`Name`、`Score` の順で列を残すなら、この順番で引数を書くだけです。

```
1 df %>%
2   select(ID, Pref, Name)

## # A tibble: 6,292 x 3
##       ID     Pref   Name
##   <chr>  <chr>   <chr>
## 1 e539604 東京都 居酒屋 龍記 京橋店
## 2 gfeb600 東京都 本格上海料理 新錦江 上野御徒町本店
## 3 ggt5900 東京都 食べ飲み放題 × 中華ビストロ NOZOMI (のぞみ)
## 4 g181340 東京都 博多餃子軒 八王子店 タピオカ店 Bull Pulu (ブルブル) 併設
## 5 ggww100 東京都 まさ屋 渋谷店
## 6 gdzk500 東京都 完全個室 上海レストラン 檸檬 霞ヶ関ビル内店
## 7 ga2g202 東京都 鶏そば きらり
## 8 gg9m100 東京都 宴会個室 × 餃子酒場 北京飯店 秋葉原本店
## 9 gdvk200 東京都 中国料理 宝龍
## 10 gggb200 東京都 中国料理 天安門
## # ... with 6,282 more rows
```

### 12.3.2 特定の列を抽出し、列名を変更する

また、特定の列を残す際、変数名を変更することも可能です。今回も ID、Name、Pref、Score のみを残しますが、Pref 列は Prefecture に変えてみましょう。

```
1 df %>%
2   select(ID, Name, Prefecture = Pref, Score)

## # A tibble: 6,292 x 4
##   ID      Name      Prefecture Score
##   <chr>   <chr>      <chr>      <dbl>
## 1 e539604 居酒屋龍記京橋店          東
京都      NA
## 2 gfeb600 本格上海料理新錦江上野御徒町本店      東
京都      4.5
## 3 ggt5900 食べ飲み放題×中華ビストロNOZOMI(のぞみ) 東
京都      NA
## 4 g181340 博多餃子軒八王子店タピオカ店Bull Pulu(ブルブル~ 東
京都      NA
## 5 ggww100 まさ屋渋谷店          東
京都      NA
## 6 gdzk500 完全個室上海レストラン檸檬霞ヶ関ビル内店      東
京都      NA
## 7 ga2g202 鶏そばきらり          東
京都      NA
## 8 gg9m100 宴会個室×餃子酒場北京飯店秋葉原本店 東京
都        3.33
## 9 gdvk200 中国料理宝龍          東
京都      2.5
## 10 gggb200 中国料理天安門         東
京都     NA
## # ... with 6,282 more rows
```

抽出する際、変数を新しい変数名 = 既存の変数名にするだけで、変数名が簡単に変更できました。もし、特定の列は抽出しないものの、変数名を変えるにはどうすれば良いでしょうか。ここでは `df` の `Pref` を `Prefecture` に、`Walk` を `Distance` に変更してみます。`{dplyr}` を使わない場合と`{dplyr}`の `rename()` 関数を使う場合を両方紹介します。

まずは、`name()` 関数についてですが、これはデータフレームの変数名をベクトルとして出力する関数です。

```
1 names(df)

##  [1] "ID"          "Name"        "Pref"        "Zipcode"      "Latitude"    "Longitude"
##  [7] "Line"        "Station"      "Walk"        "Bus"         "Car"         "Budget"
## [13] "ScoreN"      "Score"
```

察しの良い読者は気づいたかも知れませんが、`names(データフレーム名)` の結果はベクトルであり、上書きも可能です。つまり、`names(df)` の 3 番目と 9 番目の要素を"`Prefecture`"と"`Distance`"に上書きすることができるということです。

```
1 # dplyr を使わずに列名を変更する方法
2 names(df)[c(3, 9)] <- c("Prefecture", "Distance")
3
4 # df の中身を出力
5 df

## # A tibble: 6,292 x 14
##   ID      Name  Prefecture Zipcode Latitude Longitude Line  Station Distance
##   <chr>   <chr>   <chr>      <dbl>    <dbl>    <dbl> <chr> <chr>    <dbl>
## 1 e539604 居酒屋 ~ 東京都      1040031    35.7    140. 地下~ 銀座
## 2 gfeb600 本格上~ 東京都      1100005    35.7    140. 地下~ 仲御
## 3 ggt5900 食べ飲~ 東京都      1250041    35.8    140. J R~ 金町
## 4 g181340 博多餃~ 東京都      1920904    35.7    139. J R  八王
## 5 g181340 博多餃~ 東京都      1920904    35.7    139. J R  八王
## 6 g181340 博多餃~ 東京都      1920904    35.7    139. J R  八王
## # ... with 6,286 more rows, and 1 more variable:
## #   Distance: <dbl> (length=6292, .drop=TRUE)
```

```

## 5 ggww100 まさ屋 ~ 東京都      1500042  35.7    140. 地下~ 渋谷
駅      7
## 6 gdzk500 完全個~ 東京都      1000013  35.7    140. 地下~ 虎ノ
門~      3
## 7 ga2g202 鶏そば ~ 東京都      1760006  35.7    140. 西武~ 江古
田~      2
## 8 gg9m100 宴会個~ 東京都      1010021  35.7    140. J R  秋葉
原~      4
## 9 gdvk200 中国料~ 東京都      1000006  35.7    140. J R  有楽
町~      1
## 10 gggb200 中国料~ 東京都     1140002  35.8    140. 地下~ 王子
駅      2
## # ... with 6,282 more rows, and 5 more variables: Bus <dbl>, Car <dbl>,
## #   Budget <dbl>, ScoreN <dbl>, Score <dbl>

```

簡単に変数名の変更ができます。続いて、`{dplyr}`の `rename()` 関数を使った方法です。今回は、`Prefecture` を `Pref` に、`Distance` を `Walk` に戻して見ましょう。そして、出力するだけにとどまらず、`df` に上書きしましょう。

```

1 # df の Prefecture を Pref に、Distance を Walk に変更し、上書きする
2 df <- df %>%
3   rename(Pref = Prefecture, Walk = Distance)

```

これで終わりです。実は `select()` 関数と使い方がほぼ同じです。ただし、残す変数名を指定する必要がなく、名前を変更する変数名と新しい変数名を入れるだけです。変数が少ないデータなら `select()` でもあまり不便は感じないかも知れませんが、変数が多くなると `rename()` 関数は非常に便利です。

### 12.3.3 特定の列を除外する

逆に、一部の変数をデータフレームから除去したい場合もあるでしょう。たとえば、緯度 (`Latitude`) と経度 (`Longitude`) はラーメン屋の情報としては不要かもしれません。この 2 つの変数を除外するためにはどうすれば良いでしょうか。まず考えられるのは、こ

の2つの変数を除いた変数を指定・抽出する方法です。

```

1 df %>%
2   select(ID, Name, Pref, Zipcode,
3         Line, Station, Walk, Bus, Car, Budget, ScoreN, Score)
4
5 ## # A tibble: 6,292 x 12
6 ##   ID     Name   Pref   Zipcode Line   Station   Walk     Bus     Car Budget ScoreN Score
7 ##   <chr> <chr> <chr>   <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
8 ##   1 e539~ 居酒~ 東京~ 1040031 地下~ 銀座一~     3     NA     NA 3000     0 NA
9 ##   2 gfeb~ 本格~ 東京~ 1100005 地下~ 仲御徒~     1     NA     NA 2000     2 4.5
10 ##  3 ggt5~ 食べ~ 東京~ 1250041 JR~ 金町駅     2     NA     NA 2980     0 NA
11 ##  4 g181~ 博多~ 東京~ 1920904 JR~ 八王子~     1     NA     NA 2000     0 NA
12 ##  5 ggww~ まさ~ 東京~ 1500042 地下~ 渋谷駅     7     NA     NA 380      0 NA
13 ##  6 gdzk~ 完全~ 東京~ 1000013 地下~ 虎ノ門~     3     NA     NA 2980     0 NA
14 ##  7 ga2g~ 鶏そ~ 東京~ 1760006 西武~ 江古田~     2     NA     NA 850      0 NA
15 ##  8 gg9m~ 宴会~ 東京~ 1010021 JR~ 秋葉原~     4     NA     NA 2000     3 3.33
16 ##  9 gdvk~ 中国~ 東京~ 1000006 JR~ 有楽町~     1     NA     NA 1000     2 2.5
17 ## 10 gggb~ 中国~ 東京~ 1140002 地下~ 王子駅     2     NA     NA 2000     0 NA
18 ## # ... with 6,282 more rows

```

かなり長いコードになりましたね。しかし、もっと簡単な方法があります。それは-を使う方法です。

```

1 df %>%
2   select(-Latitude, -Longitude) # select(-c(Latitude, Longitude))
3
4 ## # A tibble: 6,292 x 12
5 ##   ID     Name   Pref   Zipcode Line   Station   Walk     Bus     Car Budget ScoreN Score
6 ##   <chr> <chr> <chr>   <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
7 ##   1 e539~ 居酒~ 東京~ 1040031 地下~ 銀座一~     3     NA     NA 3000     0 NA
8 ##   2 gfeb~ 本格~ 東京~ 1100005 地下~ 仲御徒~     1     NA     NA 2000     2 4.5
9 ##  3 ggt5~ 食べ~ 東京~ 1250041 JR~ 金町駅     2     NA     NA 2980     0 NA
10 ##  4 g181~ 博多~ 東京~ 1920904 JR~ 八王子~     1     NA     NA 2000     0 NA

```

```

## 5 ggww~ まさ~ 東京~ 1500042 地下~ 渋谷駅      7   NA   NA   380   0 NA
## 6 gdzk~ 完全~ 東京~ 1000013 地下~ 虎ノ門~      3   NA   NA  2980   0 NA
## 7 ga2g~ 鶏そ~ 東京~ 1760006 西武~ 江古田~      2   NA   NA   850   0 NA
## 8 gg9m~ 宴会~ 東京~ 1010021 JR 秋葉原~      4   NA   NA  2000   3 3.33
## 9 gdvk~ 中国~ 東京~ 1000006 JR 有楽町~      1   NA   NA  1000   2  2.5
## 10 gggb~ 中国~ 東京~ 1140002 地下~ 王子駅     2   NA   NA  2000   0 NA
## # ... with 6,282 more rows

```

除外したい変数名の前に-を付けただけです。また、-Latitude と-Latitude をそれぞれ指定せず、-c(Latitude, Longitude) のように c() でまとめるのも可能です。

#### 12.3.4 隣接した列を指定する

先ほど、df から緯度 (Latitude) と経度 (Longitude) を除外する例を考えてみましょう。-を使うと簡単ですが、場合によっては残す変数名を指定する必要があります。

```

1 df %>%
2   select(-ID, -Name, -Pref, -Zipcode,
3         -Line, -Station, -Walk, -Bus, -Car, -Budget, -ScoreN, -Score)

```

よく考えてみれば、ID から Zipcode は隣接した列ですし、Line から Score までもそうです。これは names() 関数で確認できます。

```

1 names(df)

## [1] "ID"          "Name"        "Pref"        "Zipcode"      "Latitude"     "Longitude"
## [7] "Line"        "Station"      "Walk"        "Bus"         "Car"         "Budget"
## [13] "ScoreN"      "Score"

```

ここで便利な演算子が: です。これまで、x から y までの公差 1 の等差数列を作成する際に x:y を使って来ましたが、これに非常に似ています。データフレームの「x 列から y 列まで」の表記も select() 関数内では: と書くことができます。したがって、上記のコードは以下のように短縮化可能です。

```

1 df %>%
2   select(ID:Zipcode, Line:Score)

```

「df の ID から Zipcode まで、そして Line から Score までの列を選択する」という意味です。非常に便利な演算子ですので、-と合わせて覚えておきましょう。

### 12.3.5 一部の列の順番だけを変える

ある列の位置を替えたいとします。たとえば、Score と ScoreN をそれぞれ 1 列目、2 列目にしたい場合、どうすれば良いでしょうか。これまで勉強したことを考えると、以下のようなコードで問題ないでしょう。

```

1 df %>%
2   select(Score, ScoreN, ID:Budget)

```

```

## # A tibble: 6,292 x 14
##   Score ScoreN ID     Name  Pref  Zipcode Latitude Longitude Line  Station  Wall
##   <dbl>  <dbl> <chr> <chr> <chr>  <dbl>     <dbl>    <dbl> <chr> <chr>  <dbl>
## 1 NA      0 e539~ 居 酒~ 東 京~ 1040031     35.7      140. 地 下
~ 銀座一~    3
## 2 4.5     2 gfeb~ 本 格~ 東 京~ 1100005     35.7      140. 地 下
~ 仲御徒~    1
## 3 NA      0 ggt5~ 食 ベ~ 東 京~ 1250041     35.8      140. J  R
~ 金町駅    2
## 4 NA      0 g181~ 博 多~ 東 京~ 1920904     35.7      139. J
R 八王子~    1
## 5 NA      0 ggww~ ま さ~ 東 京~ 1500042     35.7      140. 地 下
~ 渋谷駅    7
## 6 NA      0 gdzk~ 完 全~ 東 京~ 1000013     35.7      140. 地 下
~ 虎ノ門~    3
## 7 NA      0 ga2g~ 鶏 そ~ 東 京~ 1760006     35.7      140. 西 武
~ 江古田~    2
## 8 3.33    3 gg9m~ 宴 会~ 東 京~ 1010021     35.7      140. J

```

```
R 秋葉原~      4
## 9 2.5      2 gdvk~ 中 国~ 東 京~ 1000006      35.7      140. J
R 有楽町~      1
## 10 NA      0 gggb~ 中 国~ 東 京~ 1140002      35.8      140. 地 下
~ 王子駅      2
## # ... with 6,282 more rows, and 3 more variables: Bus <dbl>, Car <dbl>,
## #   Budget <dbl>
```

しかし、{dplyr}には `relocate()` というより便利な専用関数を提供しています。`relocate()` には変数名を指定するだけですが、ここで指定した変数がデータフレームの最初列の方に移動します。

```
1 df %>%
2   relocate(Score, ScoreN)

## # A tibble: 6,292 x 14
##   Score ScoreN ID      Name  Pref  Zipcode Latitude Longitude Line  Station  Walk
##   <dbl>  <dbl> <chr> <chr> <chr>   <dbl>     <dbl>   <chr> <chr>   <dbl>
## 1 NA      0 e539~ 居 酒~ 東 京~ 1040031      35.7      140. 地 下
~ 銀座一~      3
## 2 4.5      2 gfeb~ 本 格~ 東 京~ 1100005      35.7      140. 地 下
~ 仲御徒~      1
## 3 NA      0 ggt5~ 食 べ~ 東 京~ 1250041      35.8      140. J R
~ 金町駅      2
## 4 NA      0 g181~ 博 多~ 東 京~ 1920904      35.7      139. J
R 八王子~      1
## 5 NA      0 ggww~ ま さ~ 東 京~ 1500042      35.7      140. 地 下
~ 渋谷駅      7
## 6 NA      0 gdzk~ 完 全~ 東 京~ 1000013      35.7      140. 地 下
~ 虎ノ門~      3
## 7 NA      0 ga2g~ 鶏 そ~ 東 京~ 1760006      35.7      140. 西 武
~ 江古田~      2
## 8 3.33     3 gg9m~ 宴 会~ 東 京~ 1010021      35.7      140. J
R 秋葉原~      4
```

```

## 9 2.5      2 gdvk~ 中 国~ 東 京~ 1000006      35.7      140. J
R 有楽町~ 1
## 10 NA      0 gggb~ 中 国~ 東 京~ 1140002      35.8      140. 地 下
~ 王子駅 2
## # ... with 6,282 more rows, and 3 more variables: Bus <dbl>, Car <dbl>,
## #   Budget <dbl>

```

`relocate()` を使うと `ID:Budget` が省略可能となり、より短いコードになります。もう一つの例は、最初に持ってくるのではなく、「ある変数の前」または「ある変数の後」に移動させるケースです。これも `relocate()` で可能ですが、もう一つの引数が必要です。Pref と Zipcode の順番を変えるなら、まずは以下のような方法が考えられます。

```

1 df %>%
2   select(ID:Name, Zipcode, Pref, Latitude:Score)

## # A tibble: 6,292 x 14
##   ID      Name  Zipcode Pref  Latitude Longitude Line  Station  Walk   Bus   Ca
##   <chr>   <chr> <dbl>   <chr>     <dbl>     <dbl> <chr> <chr>   <dbl>   <dbl>   <dbl>
## 1 e5396~ 居 酒~ 1040031 東 京~      35.7      140. 地 下~ 銀 座 一
~      3     NA     NA
## 2 gfeb6~ 本 格~ 1100005 東 京~      35.7      140. 地 下~ 仲 御 徒
~      1     NA     NA
## 3 ggt59~ 食べ~ 1250041 東京~      35.8      140. J R~ 金町駅     2     NA     NA
## 4 g1813~ 博 多~ 1920904 東 京~      35.7      139. J R 八 王 子
~      1     NA     NA
## 5 ggww1~ まさ~ 1500042 東京~      35.7      140. 地下~ 渋谷駅     7     NA     NA
## 6 gdzk5~ 完 全~ 1000013 東 京~      35.7      140. 地 下~ 虎 ノ 門
~      3     NA     NA
## 7 ga2g2~ 鶏 そ~ 1760006 東 京~      35.7      140. 西 武~ 江 古 田
~      2     NA     NA
## 8 gg9m1~ 宴 会~ 1010021 東 京~      35.7      140. J R 秋 葉 原
~      4     NA     NA
## 9 gdvk2~ 中 国~ 1000006 東 京~      35.7      140. J R 有 楽 町
~      1     NA     NA

```

```
## 10 gggb2~ 中国~ 1140002 東京~ 35.8 140. 地下~ 王子駅 2 NA NA
## # ... with 6,282 more rows, and 3 more variables: Budget <dbl>, ScoreN <dbl>,
## #   Score <dbl>
```

これを `relocate()` で書き換えるなら、`.after` または `.before` 引数が必要になります。

`relocate(変数名 1, .after = 変数名 2)` は「変数 1 を変数 2 の直後に移動させる」ことを意味します。

```
1 df %>%
2   relocate(Pref, .after = Zipcode)

## # A tibble: 6,292 x 14
##   ID      Name  Zipcode Pref  Latitude Longitude Line  Station  Walk   Bus   Car
##   <chr>   <chr>  <dbl> <chr>    <dbl>    <dbl> <chr>  <chr>    <dbl>  <dbl> <dbl>
## 1 e5396~ 居酒屋~ 1040031 東京~ 35.7 140. 地下~ 銀座一
~ 3 NA NA
## 2 gfeb6~ 本格~ 1100005 東京~ 35.7 140. 地下~ 仲御徒
~ 1 NA NA
## 3 ggt59~ 食べ~ 1250041 東京~ 35.8 140. JR~ 金町駅 2 NA NA
## 4 g1813~ 博多~ 1920904 東京~ 35.7 139. JR~ 八王子
~ 1 NA NA
## 5 ggww1~ まさ~ 1500042 東京~ 35.7 140. 地下~ 渋谷駅 7 NA NA
## 6 gdzk5~ 完全~ 1000013 東京~ 35.7 140. 地下~ 虎ノ門
~ 3 NA NA
## 7 ga2g2~ 鶏そ~ 1760006 東京~ 35.7 140. 西武~ 江古田
~ 2 NA NA
## 8 gg9m1~ 宴会~ 1010021 東京~ 35.7 140. JR~ 秋葉原
~ 4 NA NA
## 9 gdvk2~ 中国~ 1000006 東京~ 35.7 140. JR~ 有楽町
~ 1 NA NA
## 10 gggb2~ 中国~ 1140002 東京~ 35.8 140. 地下~ 王子駅 2 NA NA
## # ... with 6,282 more rows, and 3 more variables: Budget <dbl>, ScoreN <dbl>,
## #   Score <dbl>
```

.before を使うことができます。この場合は「Zipcode を Pref の直前に移動させる」ことを指定する必要があります。結果は省略しますが、自分でコードを走らせ、上と同じ結果が得られるかを確認してみてください。

```
1 df %>%
2   relocate(Zipcode, .before = Pref)
```

### 12.3.6 select() の便利な機能

select() 関数は他にも便利な機能がいくつかあります。ここではいくつかの機能を紹介しますが、より詳しい内容は?dplyr::select を参照してください。

**starts\_with() と ends\_with()、contains()、num\_range(): 特定の文字を含む変数を選択する**

まずは、特定の文字を含む変数名を指定する方法です。starts\_with("X")、ends\_with("X")、contains("X") は変数名が "X" で始まるか、"X" で終わるか、"X" を含むかを判断し、条件に合う変数名を返す関数です。実際の例を見ましょう。

```
1 # ID、Name に続いて、Score で始まる変数名を抽出
2 df %>%
3   select(ID, Name, starts_with("Score"))

## # A tibble: 6,292 x 4
##   ID      Name          ScoreN ScoreC
##   <chr>   <chr>        <dbl>  <dbl>
## 1 e539604 居酒屋 龍記 京橋店        0  NA
## 2 gfeb600 本格上海料理 新錦江 上野御徒町本店    2  4.5
## 3 ggt5900 食べ飲み放題 × 中華ビストロ NOZOMI(のぞみ) 0  NA
## 4 g181340 博多餃子軒 八王子店 タピオカ店 Bull Pulu (ブルブル)
## 5 ggww100 まさ屋 渋谷店        0  NA
## 6 gdzk500 完全個室 上海レストラン 檸檬 霞ヶ関ビル内店 0  NA
## 7 ga2g202 鶏そば きらり        0  NA
```

```

## 8 gg9m100 宴会個室 × 餃子酒場 北京飯店 秋葉原本店          3 3.33
## 9 gdvk200 中国料理 宝龍                           2 2.5
## 10 gggb200 中国料理 天安門                      0 NA
## # ... with 6,282 more rows

1 # e で終わる変数名を除去
2 df %>%
3   select(-ends_with("e")) # !ends_with("e") も可能

## # A tibble: 6,292 x 8
##   ID     Pref   Station     Walk     Bus     Car Budget ScoreN
##   <chr>  <chr>  <chr>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 e539604 東京都 銀座一丁目駅      3     NA     NA  3000      0
## 2 gfeb600 東京都 仲御徒町駅      1     NA     NA  2000      2
## 3 ggt5900 東京都 金町駅          2     NA     NA  2980      0
## 4 g181340 東京都 八王子駅        1     NA     NA  2000      0
## 5 ggww100 東京都 渋谷駅          7     NA     NA   380      0
## 6 gdzk500 東京都 虎ノ門駅        3     NA     NA  2980      0
## 7 ga2g202 東京都 江古田駅        2     NA     NA   850      0
## 8 gg9m100 東京都 秋葉原駅        4     NA     NA  2000      3
## 9 gdvk200 東京都 有楽町駅        1     NA     NA  1000      2
## 10 gggb200 東京都 王子駅         2     NA     NA  2000      0
## # ... with 6,282 more rows

1 # re を含む変数名を抽出するが、ScoreN は除去する
2 df %>%
3   select(contains("re"), -ScoreN)

## # A tibble: 6,292 x 2
##   Pref   Score
##   <chr>  <dbl>
## 1 東京都  NA
## 2 東京都  4.5
## 3 東京都  NA

```

```

## 4 東京都 NA
## 5 東京都 NA
## 6 東京都 NA
## 7 東京都 NA
## 8 東京都 3.33
## 9 東京都 2.5
## 10 東京都 NA
## # ... with 6,282 more rows

```

他の使い方としては X1、X2 のような「文字 + 数字」の変数を選択する際、`starts_with()` が活躍します。たとえば、以下のような `myDF1` があるとします。

```

1 # tibble() でなく、data.frame() も使用可能です。
2 myDF1 <- tibble(
3   ID   = 1:5,
4   X1   = c(2, 4, 6, 2, 7),
5   Y1   = c(3, 5, 1, 1, 0),
6   X1D  = c(4, 2, 1, 6, 9),
7   X2   = c(5, 5, 6, 0, 2),
8   Y2   = c(3, 3, 2, 3, 1),
9   X2D  = c(8, 9, 5, 0, 1),
10  X3   = c(3, 0, 3, 0, 2),
11  Y3   = c(1, 5, 9, 1, 3),
12  X3D  = c(9, 1, 3, 3, 8)
13 )
14
15 myDF1

```

```

## # A tibble: 5 x 10
##       ID     X1     Y1   X1D     X2     Y2   X2D     X3     Y3   X3D
##   <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     2     3     4     5     3     8     3     1     9
## 2     2     4     5     2     5     3     9     0     5     1

```

```
## 3     3     6     1     1     6     2     5     3     9     3
## 4     4     2     1     6     0     3     0     0     1     3
## 5     5     7     0     9     2     1     1     2     3     8
```

この myDF1 から ID、Y1、Y2、Y3 を抽出するにはどうすれば良いでしょうか。これらの変数は隣接していないため、: も使えませんが、`starts_with()` を使えば簡単です。

```
1 myDF1 %>%
2   select(ID, starts_with("Y"))
```

```
## # A tibble: 5 x 4
##       ID     Y1     Y2     Y3
##   <int> <dbl> <dbl> <dbl>
## 1     1     3     3     1
## 2     2     5     3     5
## 3     3     1     2     9
## 4     4     1     3     1
## 5     5     0     1     3
```

それでは、ID、X1、X2、X3 はどうでしょうか。`starts_with("X")` だと、X1c なども選択されてしまいますね。ここで`-ends_with()` の出番です。つまり、「まずは`starts_with("X")` で X で始まる変数を選択し、続いて、D で終わるもの除外すればいいじゃん？」です。それでは、やってみましょうか。

```
1 myDF1 %>%
2   select(ID, starts_with("X"), -ends_with("D"))
```

```
## # A tibble: 5 x 3
##       X1     X2     X3
##   <dbl> <dbl> <dbl>
## 1     2     5     3
## 2     4     5     0
## 3     6     6     3
## 4     2     0     0
## 5     7     2     2
```

あらら、ID も同時になくなりましたね<sup>1)</sup>。実はこのような時のために用意された関数があり、それが `num_range()` です。`num_range()` の第一引数は `starts_with()` 関数と同じですが、第二引数も必要です。この第二引数には numeric 型のベクトルが必要です。`1:3` でも、`c(1, 2, 3)` でも構いません。たとえば、ID、X1、X2、X3 するには以下のように書きます。

```

1 myDF1 %>%
2   select(ID, num_range("X", 1:3))

## # A tibble: 5 x 4
##       ID     X1     X2     X3
##   <int> <dbl> <dbl> <dbl>
## 1     1     2     5     3
## 2     2     4     5     0
## 3     3     6     6     3
## 4     4     2     0     0
## 5     5     7     2     2

```

### all\_of() と any\_of(): 文字型ベクトルを用いた変数の選択

`all_of()` と `any_of()` は `select()` 内の変数名として文字型ベクトルを使う際に用いる関数です。これは抽出したい列名が既に character 型ベクトルとして用意されている場合、便利な関数です。たとえば、以下の `Name_Vec` を考えてみましょう。

```

1 Name_Vec <- c("X1", "X2", "X3")

```

この `Name_Vec` の要素と同じ列名を持つ列と ID 列を `myDF1` から抽出する方法は以下の2通りです。

```

1 myDF1[, c("ID", Name_Vec)]

## # A tibble: 5 x 4
##       ID     X1     X2     X3
##   <int> <dbl> <dbl> <dbl>
## 1     1     2     5     3
## 2     2     4     5     0
## 3     3     6     6     3
## 4     4     2     0     0
## 5     5     7     2     2

```

---

<sup>1)</sup> 実は `select(starts_with("X"), -ends_with("D"), ID)` のように順番を変えると ID は最後の列になりますが、とりあえず残ります。なぜなら、`select()` 関数は左側から右側の方へコードを実行するからです。

```
## <int> <dbl> <dbl> <dbl>
## 1 1 2 5 3
## 2 2 4 5 0
## 3 3 6 6 3
## 4 4 2 0 0
## 5 5 7 2 2

1 myDF1 %>%
2   select(ID, all_of(Name_Vec))

## # A tibble: 5 x 4
##       ID     X1     X2     X3
##   <int> <dbl> <dbl> <dbl>
## 1 1 2 5 3
## 2 2 4 5 0
## 3 3 6 6 3
## 4 4 2 0 0
## 5 5 7 2 2
```

今の例だと、`select()` を使わない前者の方が便利かも知れませんが、`select()` 内に外の変数名も指定する場合も多いので、後者の方が汎用性は高いです。私から見れば、今の例でも後者の方が読みやすく、使いやすいと思います。

それでは以下のような `Name_Vec` はどうでしょう。今回は、`myDF1` に含まれていない `X4` と `X5` もあります。

```
1 Name_Vec <- c("X1", "X2", "X3", "X4", "X5")
2
3 myDF1 %>%
4   select(all_of(Name_Vec))

## Error: Can't subset columns that don't exist.
##   Columns `X4` and `X5` don't exist.
```

このようにエラーが出てします。つまり、`all_of()` の場合、引数の要素全てがデー

タフレームに存在する必要があります。もし、ないものは無視して、合致する列だけ取り出したいはどうすれば良いでしょうか。そこで登場するのが `any_of()` です。

```
1 myDF1 %>%
2   select(any_of(Name_Vec))

## # A tibble: 5 x 3
##       X1     X2     X3
##   <dbl> <dbl> <dbl>
## 1     2     5     3
## 2     4     5     0
## 3     6     6     3
## 4     2     0     0
## 5     7     2     2
```

`any_of()` の方がより使いやすいと思う方も多いでしょうが、必ずしもそうとは限りません。たとえば、`Name_Vec` に誤字などが含まれる場合、`any_of()` だと誤字が含まれている変数は取り出しません。この場合はむしろちゃんとエラーを表示してくれた方が嬉しいですね。

### last\_col(): 最後の列を選択する

普段あまり使わない機能ですが、最後の列を選択する `last_col()` という関数もあります。たとえば、`last_col(0)` にすると最後の列を選択し、`last_col(1)` なら最後から2番目の列を選択します。たとえば、`df` から `ID` と最後の列を取り出してみましょう。

```
1 # ID と最後の列のみを抽出
2 df %>%
3   select(ID, last_col(0))

## # A tibble: 6,292 x 2
##       ID     Score
##   <chr>    <dbl>
## 1 e539604  NA
## 2 gfeb600  4.5
```

```
## 3 ggt5900 NA
## 4 g181340 NA
## 5 ggww100 NA
## 6 gdzk500 NA
## 7 ga2g202 NA
## 8 gg9m100 3.33
## 9 gdvk200 2.5
## 10 gggb200 NA
## # ... with 6,282 more rows
```

最後の 2 行分を取り出すことも可能です。この場合は `last_col()` の引数を長さ 1 ベクトルでなく、長さ 2 以上のベクトルにします。最後の行が 0、その手前の行が 1 ですから、中の引数は `1:0` となります。`0:1` でも可能ですが、結果が若干異なります。

```
1 # ID と最後の 2 列分を抽出 (引数を 1:0 と設定)
2 df %>%
3   select(ID, last_col(1:0))
```

```
## # A tibble: 6,292 x 3
##       ID     ScoreN Score
##   <chr>     <dbl> <dbl>
## 1 e539604     0     NA
## 2 gfeb600     2     4.5
## 3 ggt5900     0     NA
## 4 g181340     0     NA
## 5 ggww100     0     NA
## 6 gdzk500     0     NA
## 7 ga2g202     0     NA
## 8 gg9m100     3     3.33
## 9 gdvk200     2     2.5
## 10 gggb200    0     NA
## # ... with 6,282 more rows
```

```

1 # ID と最後の 2 列分を抽出 (引数を 0:1 と設定)
2 df %>%
3   select(ID, last_col(0:1))

## # A tibble: 6,292 x 3
##   ID      Score ScoreN
##   <chr>    <dbl>  <dbl>
## 1 e539604  NA      0
## 2 gfeb600  4.5     2
## 3 ggt5900  NA      0
## 4 g181340  NA      0
## 5 ggww100  NA      0
## 6 gdzk500  NA      0
## 7 ga2g202  NA      0
## 8 gg9m100  3.33    3
## 9 gdvk200  2.5     2
## 10 gggb200  NA      0
## # ... with 6,282 more rows

```

`last_col()` の引数を `1:0` にするか `0:1` によって抽出される順番が異なります。`1:0` は `c(1, 0)`、`0:1` は `c(0, 1)` と同じであることを考えると理由は簡単です。`c(1, 0)` の場合、`last_col(1)`、`last_col(0)` の順番で処理をし、`c(0, 1)` は `last_col(0)`、`last_col(1)` の順番で処理を行うからです。

この `last_col()` の引数を空っぽにするとそれは最後の列を意味します。これを利用すれば、「ある変数の最後の列へ移動させる」こともできます。たとえば、ID を最後の列に移動させたい場合、`relocate(ID, .after = last_col())` のように書きます。

### where(): データ型から変数を選択する

最後に、「numeric 型の列のみ抽出したい」、「character 型の列だけほしい」場合に便利な `where()` 関数を紹介します。`where()` の中に入る引数は一つだけであり、データ型を判定する関数名が入ります。たとえば、numeric 型か否かを判断する関数は `is.numeric` です。`df` から numeric 型の変数のみを抽出したい場合は以下のように書きます。

```
1 # numeric 型の列を抽出する
2 df %>%
3   select(where(is.numeric))

## # A tibble: 6,292 x 9
##   Zipcode Latitude Longitude Walk   Bus   Car Budget ScoreN Score
##   <dbl>     <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1040031     35.7     140.    3     NA     NA   3000     0   NA
## 2 1100005     35.7     140.    1     NA     NA   2000     2   4.5
## 3 1250041     35.8     140.    2     NA     NA   2980     0   NA
## 4 1920904     35.7     139.    1     NA     NA   2000     0   NA
## 5 1500042     35.7     140.    7     NA     NA   380      0   NA
## 6 1000013     35.7     140.    3     NA     NA   2980     0   NA
## 7 1760006     35.7     140.    2     NA     NA   850      0   NA
## 8 1010021     35.7     140.    4     NA     NA   2000     3   3.33
## 9 1000006     35.7     140.    1     NA     NA   1000     2   2.5
## 10 1140002    35.8     140.   2     NA     NA   2000     0   NA
## # ... with 6,282 more rows
```

!を使って条件に合致する列を除外することも可能です。もし、character 型の列を除外する場合は以下のように!where(is.character) を指定します。

```
1 # character 型でない列を抽出する
2 df %>%
3   select(!where(is.character))

## # A tibble: 6,292 x 9
##   Zipcode Latitude Longitude Walk   Bus   Car Budget ScoreN Score
##   <dbl>     <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1040031     35.7     140.    3     NA     NA   3000     0   NA
## 2 1100005     35.7     140.    1     NA     NA   2000     2   4.5
## 3 1250041     35.8     140.    2     NA     NA   2980     0   NA
## 4 1920904     35.7     139.    1     NA     NA   2000     0   NA
```

```
## 5 1500042      35.7      140.      7      NA      NA      380      0  NA
## 6 1000013      35.7      140.      3      NA      NA     2980      0  NA
## 7 1760006      35.7      140.      2      NA      NA      850      0  NA
## 8 1010021      35.7      140.      4      NA      NA     2000      3  3.33
## 9 1000006      35.7      140.      1      NA      NA     1000      2  2.5
## 10 1140002     35.8      140.      2      NA      NA     2000      0  NA
## # ... with 6,282 more rows
```

&を使って複数の条件を使うことも可能です。たとえば、ID変数に加えて「"L"で始まる変数の中で numeric型の列を抽出」するコードは以下のようになります。

```
1 # ID と、L で始まる numeric 型の列を抽出する
2 df %>%
3   select(ID, starts_with("L") & where(is.numeric))

## # A tibble: 6,292 x 3
##       ID      Latitude Longitude
##   <chr>     <dbl>     <dbl>
## 1 e539604     35.7     140.
## 2 gfeb600     35.7     140.
## 3 ggt5900     35.8     140.
## 4 g181340     35.7     139.
## 5 ggww100     35.7     140.
## 6 gdzk500     35.7     140.
## 7 ga2g202     35.7     140.
## 8 gg9m100     35.7     140.
## 9 gdvk200     35.7     140.
## 10 gggb200    35.8     140.
## # ... with 6,282 more rows
```

## 12.4 行の抽出

### 12.4.1 指定した行を抽出する

他にも特定の行を抽出する場合があります。たとえば、「df の最初の 5 行」や「df の 8 行目のケース」といった場合です。この操作には{dplyr}の `slice_*`() 関数群が便利です。それではそれぞれの関数の使い方について紹介していきます。その前に、実習用データとして df から一部の列のみを抽出した `select_df` を作成します。

```
1 select_df <- df %>%
2   select(ID, Name, Pref, Budget, Score)
```

#### `slice()`: 指定した番号の行のみ抽出する

`select_df` から 2, 8, 9 行目の行を抽出したいとします。このような簡単な操作はパッケージを使わず、以下のように抽出することができます。

```
1 # select_df から 2, 8, 9 行目の行を抽出し、出力する
2 select_df[c(2, 8, 9),]
```

```
## # A tibble: 3 x 5
##   ID      Name          Pref   Budget Score
##   <chr>   <chr>        <chr>   <dbl>  <dbl>
## 1 gfeb600 本格上海料理 新錦江 上野御徒町本店 東京都    2000   4.5
## 2 gg9m100 宴会個室 × 餃子酒場 北京飯店 秋葉原本店 東京都    2000   3.33
## 3 gdvk200 中国料理 宝龍          東京都    1000   2.5
```

しかし、以下の `slice()` 関数を使うとパイプ演算子を前後に付けることが可能であり<sup>2)</sup>、コードの可読性も高いです。`slice()` 関数には以下のように抽出したい行の番号を入れるだけです。

---

<sup>2)</sup> 実は `select_df[, c(2, 8, 9)]` でも前後にパイプ演算子を使うことは可能ですが、コードが読みにくくなるため、推奨しません。

```

1 # select_df から 2, 8, 9 行目の行を抽出し、出力する
2 select_df %>%
3   slice(2, 8, 9) # slice(c(2, 8, 9)) も OK

## # A tibble: 3 x 5
##   ID      Name          Pref  Budget  Score
##   <chr>   <chr>        <chr>  <dbl>   <dbl>
## 1 gfeb600 本格上海料理 新錦江 上野御徒町本店 東京都    2000   4.5
## 2 gg9m100 宴会個室 × 餃子酒場 北京飯店 秋葉原本店 東京都    2000   3.33
## 3 gdvk200 中国料理 宝龍          東京都    1000   2.5

```

slice(2, 8, 9) でも slice(c(2, 8, 9)) でも構いません。また、隣接した行でしたら: を使うこともできます。たとえば、10 行目から 15 行目まで抽出する場合は slice(10:15) のような書き方も出来ます。

### slice\_head(): 最初の n 行を抽出する

```

1 # select_df から最初の 3 行抽出し、出力する
2 select_df %>%
3   slice_head(n = 3)

## # A tibble: 3 x 5
##   ID      Name          Pref  Budget  Score
##   <chr>   <chr>        <chr>  <dbl>   <dbl>
## 1 e539604 居酒屋 龍記 京橋店          東京都    3000   NA
## 2 gfeb600 本格上海料理 新錦江 上野御徒町本店 東京都    2000   4.5
## 3 ggt5900 食べ飲み放題 × 中華ビストロ NOZOMI (のぞみ) 東京都    2980   NA

```

これは head(データ名, n = 出力する個数) と同じ動きをする関数です。注意点としては引数 n = を必ず付ける点です。たとえば、slice\_head(3) にすると、select\_df の 3 行目のみ抽出されます。

### slice\_tail(): 最後の n 行を抽出する

```

1 # select_df から最後の 7 行を抽出し、出力する
2 select_df %>%
3   slice_tail(n = 7)

```

```

## # A tibble: 7 x 5
##   ID      Name          Pref   Budget  Score
##   <chr>   <chr>        <chr>   <dbl>   <dbl>
## 1 5508852 場鶴          和歌山県     NA     NA
## 2 7113351 来来亭 橋本店  和歌山県     NA     NA
## 3 6364939 ばり馬 和歌山紀三井寺店 和歌山県     NA     NA
## 4 7103349 ramen BIRDMAN 和歌山県     NA     NA
## 5 7315303 薩摩ラーメン 斗天王 和歌山県     NA     NA
## 6 7703472 まるしげ       和歌山県     NA     NA
## 7 6395035 暴豚製麺所  和歌山県     NA     NA

```

これは `tail`(データ名, `n` = 出力する個数) と同じ動きをする関数です。ちなみに、この `n` 引数も `n =` を明記する必要があります。

### `slice_max()`: 指定した変数が大きい順で `n` 行抽出する

`slice_max()` は指定した変数が大きい順で `n` 行抽出する関数です。たとえば、`Budget` が高い順で 4 店舗を抽出する場合は以下のように書きます。

```

1 # select_df から Score の値が高い順で 5 行を抽出し、出力する
2 select_df %>%
3   slice_max(Budget, n = 4)

```

```

## # A tibble: 4 x 5
##   ID      Name          Pref   Budget  Score
##   <chr>   <chr>        <chr>   <dbl>   <dbl>
## 1 g670609 横浜ベイシェラトン ホテル & タワーズ 中国料理 彩龍 神奈川
## 2 g910420 JASMINE 憶 江 南           東
## 3 g670609 横浜ベイシェラトン ホテル & タワーズ 中国料理 彩龍 神奈川
## 4 g910420 JASMINE 憶 江 南           東

```

```
## 3 7176666 赤坂焼鳥鳳 東
京都 7000 NA
## 4 b612800 羽衣銀座本店 東
京都 6000 NA
```

**slice\_min(): 指定した変数が小さい順で n 行抽出する**

一方、`slice_min()` 関数が小さい順で抽出します。

```
1 # select_df から Score の値が低い順で 3 行を抽出し、出力する
2 select_df %>%
3   slice_min(Score, n = 3)
```

```
## # A tibble: 4 x 5
##   ID      Name          Pref   Budget Score
##   <chr>   <chr>        <chr>   <dbl>  <dbl>
## 1 6384909 葛西大勝軒    東京都    NA     1
## 2 6929243 由丸アトレヴィ大塚店 東京都    NA     1
## 3 5816075 ラーメン戯拉戯拉 千葉県    NA     1
## 4 5495086 らあめん花月嵐 坂戸わかば店 埼玉県    NA     1
```

ただし、`n = 3` と指定したはずなのに、4 行が抽出されました。これは同点のケースがあるからです。実際、`select_df` には `Score` が 1 のケースが 4 つあります。もし、同点の存在により `n` に収まらない場合、`slice_max()`、`slice_min()` 関数は `n` を超える行を出力します。これを強制的に `n` 行に合わせるために `with_ties = FALSE` 引数を付けます。この場合、データで格納されている順で `n` 個のみ出力されます。

```
1 select_df %>%
2   slice_min(Score, n = 3, with_ties = FALSE)
```

```
## # A tibble: 3 x 5
##   ID      Name          Pref   Budget Score
##   <chr>   <chr>        <chr>   <dbl>  <dbl>
## 1 6384909 葛西大勝軒    東京都    NA     1
## 2 6929243 由丸アトレヴィ大塚店 東京都    NA     1
```

```
## 3 5816075 ラーメン戯拉戯拉 千葉県 NA 1
```

### slice\_sample(): 無作為に n 行を抽出する

最後に無作為に n 行を抽出する `slice_sample()` 関数です。引数は `n` であり、抽出したい行数を指定します。たとえば、`select_df` から無作為に 10 行抽出したい場合は、

```
1 # select_df から無作為に 5 行を抽出し、出力する
2 select_df %>%
3   slice_sample(n = 10)
```

```
## # A tibble: 10 x 5
##   ID      Name      Pref  Budget Score
##   <chr>   <chr>   <chr>  <dbl>  <dbl>
## 1 6663363 麺 定食 嵐神 千葉県     NA     NA
## 2 5508133 栄太呂ラーメン 兵庫県     NA     NA
## 3 7068043 いろはラーメン店 京都府     NA     NA
## 4 5495228 北海道ラーメン 埼玉県     NA     NA
## 5 5499274 やすのたまぞう 東京都     NA     4
## 6 5494920 めんくい亭 埼玉県     NA     NA
## 7 5508502 馬力軒 津名店 兵庫県     NA     NA
## 8 6848207 香港屋台麵 埼玉県     NA     NA
## 9 7483838 ラーメン ギョーザ 飛龍 大阪府     NA     NA
## 10 7485257 縁乃助商店 大阪府    1000    NA
```

のように書きます。ブートストラップ法や機械学習における交差検証 (cross-validation) の際に有用な関数ですが、ブートストラップや機械学習のパッケージの多くはサンプル分割の関数を提供しているため、あまり使う機会はないでしょう。また、`slice_sample()` 関数をブートストラップ法のために用いる場合は、ケースを反復抽出する必要があり、`replace = TRUE` を付けると反復抽出を行います。デフォルト値は `FALSE` です。

### 12.4.2 条件に合致する行を抽出する

これまで見てきた `slice()` を用いる行の抽出は、実際あまり使う機会がありません。多くの場合、「何かの条件と合致するケースのみ抽出する」または、「何かの条件と合致しないケースのみを抽出する」やこれらの組み合わせで行の抽出を行います。そこで登場するのが`{dplyr}`パッケージの `filter()` 関数です。`filter()` 関数の使い方は以下の通りです。

```
1 # dplyr::filter() の使い方
2 filter(データフレーム名, 条件1, 条件2, ...)
```

もちろん、第一引数がデータですから、`%>%` を使うことも可能です。

```
1 # dplyr::filter() の使い方 (パイプを使う方法)
2 データフレーム名 %>%
3   filter(条件1, 条件2, ...)
```

まずは、条件が一つの場合を考えてみましょう。ここでは「`Pref` が"京都府"であるケースのみに絞り、`Name` と `Station`、`Score` 列のみを出力する」ケースを考えてみましょう。まず、`filter()` 関数で行を抽出し、続いて `select()` 関数で抽出する列を指定します。もちろん、今回の場合、`filter()` と `select()` の順番は替えて構いません。

```
1 # df から Pref が"京都府"であるケースのみ残し、df2 という名で保存
2 df2 <- df %>%
3   filter(Pref == "京都府")
4
5 # df2 から Name, Station, Score 列を抽出
6 df2 %>%
7   select(Name, Station, Score)

## # A tibble: 414 x 3
##       Name     Station     Score
##       <chr>    <chr>     <dbl>
## 1 中國料理鳳麟     くいな     1.00
```

橋駅 NA

## 2 黒毛和牛一頭買い焼肉と 炊き立て土鍋ご飯 市場小路 烏丸店 四条  
駅 3.19

## 3 京の中華ハマムラみやこみち店 京

都駅 NA

## 4 焼肉処真桂店 桂

駅 NA

## 5 祇園京都ラーメン 祇園四

条駅 NA

## 6 創作料理串カツトンカツ jiro 新田

辺駅 NA

## 7 祇園晩餐のあと 祇園四

条駅 NA

## 8 DETAIL 東

山駅 NA

## 9 めんや龍神 北大

路駅 NA

## 10 無尽蔵京都八条家 京

都駅 3.5

## # ... with 404 more rows

これは df から Pref == "京都府" のケースのみ残したものを df2 として格納し、それをまた select() 関数を使って列を抽出するコードです。これでも問題ありませんが、これだとパイプ演算子の便利さが分かりません。パイプ演算子は複数使うことが可能ですが。

```

1 df %>%
2   filter(Pref == "京都府") %>%
3   select(Name, Station, Score)

```

```

## # A tibble: 414 x 3
##   Name          Station     Score
##   <chr>        <chr>      <dbl>
## 1 中國料理鳳麟    くいな      4.5
## 2 橋駅 NA

```

## 2 黒毛和牛一頭買い焼肉と 炊き立て土鍋ご飯 市場小路 烏丸店 四条駅	3.19	
## 3 京の中華ハマムラ みやこみち店		京
都駅	NA	
## 4 焼肉処真桂店		桂
駅	NA	
## 5 祇園京都ラーメン		祇園四
条駅	NA	
## 6 創作料理串カツトンカツ jiro		新田
辺駅	NA	
## 7 祇園晩餐のあと		祇園四
条駅	NA	
## 8 DETAIL		東
山駅	NA	
## 9 めんや龍神		北大
路駅	NA	
## 10 無尽蔵京都八条家		京
都駅	3.5	
## # ... with 404 more rows		

全く同じ結果ですが、無駄に `df2` というデータフレームを作らず済むので、メモリの観点からも嬉しいですし、何よりコードが短く、しかも可読性も上がりました。

今回は`==`を使って合致するものに絞りましたが、`!=`を使って合致しないものに絞ることも可能です。または、比較演算子 (`<`、`>`、`>=`、`<=`など) を使うことも可能です。それでは、組み込み数 (ScoreN) が 0 ではないケースを取り出し、Name、Station、ScoreN、Score 列を出力させてみましょう。

```

## <chr> <chr> <dbl> <dbl>
## 1 本格上海料理 新錦江上野御徒町本店 仲御徒町
駅 2 4.5
## 2 宴会個室×餃子酒場 北京飯店 秋葉原本店 秋葉原駅 3 3.33
## 3 中国料理 宝龍 有楽町駅 2 2.5
## 4 麺達うま家 高田馬場
駅 2 3
## 5 刀削麺・火鍋・西安料理 XI'AN(シーアン) 後楽園店 後楽園駅 1 NA
## 6 七志らーめん 渋谷道玄坂店 渋谷駅 7 4.5
## 7 永楽 京成小岩
駅 6 4.42
## 8 よってこや お台場店 お台場海滨公
~ 1 4
## 9 ラーメン武藤製麺所 竹ノ塚駅 4 3.5
## 10 桂花ラーメン 新宿末広店 新宿三丁
駅 8 3
## # ... with 1,334 more rows

```

これで口コミ数が1以上の店舗のみに絞ることができました。ただし、店によっては口コミはあっても、評価(Score)が付いていないところもあります。たとえば、「刀削麺・火鍋・西安料理 XI'AN(シーアン) 後楽園店」の場合、口コミはありますが、評価はありません。したがって、今回は評価が付いている店舗に絞ってみましょう。

```

1 df %>%
2   filter(Score != NA) %>%
3   select(Name, Station, starts_with("Score"))

## # A tibble: 0 x 4
## # ... with 4 variables: Name <chr>, Station <chr>, ScoreN <dbl>, Score <dbl>

```

あらら、何の結果も表示されませんでした。これは filter() 内の条件に合致するケースが存在しないことを意味します。しかし、先ほどの結果を見ても、評価が付いている店はいっぱいありましたね。これはなぜでしょう。

察しの良い読者は気付いているかと思いますが、第8.8章で説明した通り、NAか否

かを判定する際は==や!=は使えません。`is.na()`を使います。`filter(is.na(Score))`なら「Score が NA であるケースに絞る」ことを意味しますが、今回は「Score が NA でないケースに絞る」ことが目的ですので、`is.na()` の前に!を付けます。

```

1 df %>%
2   filter(!is.na(Score)) %>%
3   select(Name, Station, starts_with("Score"))

## # A tibble: 1,134 x 4
##   Name           Station  ScoreN Score
##   <chr>          <chr>    <dbl>  <dbl>
## 1 本格上海料理 新錦江 上野御徒町本店 仲御徒町駅  2   4.5
## 2 宴会個室 × 餃子酒場 北京飯店 秋葉原本店 秋葉原駅  3   3.33
## 3 中国料理 宝龍          有楽町駅  2   2.5
## 4 麺達 うま家          高田馬場駅  2   3
## 5 七志らーめん 渋谷道玄坂店 渋谷駅    7   4.5
## 6 永楽           京成小岩駅  6   4.42
## 7 よってこや お台場店 お台場海浜公園駅 1   4
## 8 ラーメン武藤製麺所 竹ノ塚駅    4   3.5
## 9 桂花ラーメン 新宿末広店 新宿三丁目駅 8   3
## 10 北斗 新橋店        新橋駅    4   2.5
## # ... with 1,124 more rows

```

これで口コミ評価が登録された店舗に絞ることができました。

続いて、複数の条件を持つケースを考えてみましょう。例えば、「京都府内の店舗で、口コミ評価が 3.5 以上の店舗」を出力したい場合、以下のようなコードとなります。

```

1 df %>%
2   filter(Pref == "京都府", Score >= 3.5) %>%
3   select(Name, Station, ScoreN, Score)

## # A tibble: 53 x 4
##   Name           Station  ScoreN Score
##   <chr>          <chr>    <dbl>  <dbl>
## 1 本格上海料理 新錦江 上野御徒町本店 仲御徒町駅  2   4.5
## 2 宴会個室 × 餃子酒場 北京飯店 秋葉原本店 秋葉原駅  3   3.33
## 3 中国料理 宝龍          有楽町駅  2   2.5
## 4 麺達 うま家          高田馬場駅  2   3
## 5 七志らーめん 渋谷道玄坂店 渋谷駅    7   4.5
## 6 永楽           京成小岩駅  6   4.42
## 7 よってこや お台場店 お台場海浜公園駅 1   4
## 8 ラーメン武藤製麺所 竹ノ塚駅    4   3.5
## 9 桂花ラーメン 新宿末広店 新宿三丁目駅 8   3
## 10 北斗 新橋店        新橋駅    4   2.5
## # ... with 52 more rows

```

```

## 1 無尽蔵 京都八条家 京都駅      2 3.5
## 2 一蘭 京都河原町店 河原町駅      2 3.75
## 3 ミスター・ギョーザ 西大路駅      8 4.06
## 4 一蘭 京都八幡店 樟葉駅      3 4
## 5 中華料理 清華園 京都駅      3 5
## 6 まがり <NA>      2 4
## 7 魁力屋 北山店 北大路駅      2 4.25
## 8 大中 BAL 横店 <NA>      7 4.1
## 9 こうちやん 西舞鶴駅      1 5
## 10 大黒ラーメン 伏見桃山駅      4 4.25
## # ... with 43 more rows

```

条件を `filter()` 内に追加するだけです。今回は`!is.na(Score)` は不要です。なぜなら、`Score >= 3.5` という条件で既に欠損値は対象外になるからです。条件文が複数ある場合、AND か OR かを指定する必要があります。つまり、条件文 A と B がある場合、「A と B 両方満たすものを出力する」か「A と B どちらかを満たすものを出力するか」を指定する必要があります。今の結果って AND でしたよね。`filter()` 関数は、別途の指定がない場合、全て AND 扱いになります。R の AND 演算子は`&`ですので、以上のコードは以下のコードと同じです。

```

1 df %>%
2   filter(Pref == "京都府" & Score >= 3.5) %>%
3   select(Name, Station, ScoreN, Score)

```

```

## # A tibble: 53 x 4
##   Name           Station  ScoreN Score
##   <chr>          <chr>     <dbl> <dbl>
## 1 無尽蔵 京都八条家 京都駅      2 3.5
## 2 一蘭 京都河原町店 河原町駅      2 3.75
## 3 ミスター・ギョーザ 西大路駅      8 4.06
## 4 一蘭 京都八幡店 樟葉駅      3 4
## 5 中華料理 清華園 京都駅      3 5
## 6 まがり <NA>      2 4
## 7 魁力屋 北山店 北大路駅      2 4.25

```

```

## 8 大中 BAL 横店      <NA>      7  4.1
## 9 こうちゃん          西舞鶴駅      1  5
## 10 大黒ラーメン       伏見桃山駅     4  4.25
## # ... with 43 more rows

```

AND 演算子 (`&`) が使えるということは OR 演算子 (`|`) も使えることを意味します。たとえば、`Station` が"高田馬場駅"か"三田駅"の条件を指定したい場合、

```

1 df %>%
2   filter(Station == "高田馬場駅" | Station == "三田駅") %>%
3   select(Name, Station, ScoreN, Score)

## # A tibble: 14 x 4
##       Name           Station  ScoreN Score
##       <chr>          <chr>     <dbl> <dbl>
## 1 麺達 うま家        高田馬場駅     2     3
## 2 らあ麺 やまぐち    高田馬場駅     7   4.08
## 3 博多一瑞亭 三田店   三田駅      0     NA
## 4 つけ麺屋 ひまわり 高田馬場駅     4   2.75
## 5 石器ラーメン 高田馬場 高田馬場駅     0     NA
## 6 旨辛らーめん 表裏   高田馬場駅     0     NA
## 7 三歩一           高田馬場駅     8   4.56
## 8 えぞ菊 戸塚店     高田馬場駅     4   3.62
## 9 麺屋 宗           高田馬場駅     5   4.2
## 10 とんこつラーメン 博多風龍 高田馬場店 高田馬場駅     2     3
## 11 横浜家系ラーメン 馬場壱家     高田馬場駅     0     NA
## 12 らーめん よし丸     高田馬場駅     1     5
## 13 札幌ラーメン どさん子 三田店   三田駅      0     NA
## 14 天下一品 三田店     三田駅      0     NA

```

のように書きます（ちなみに高田馬場の「やまぐち」は本当に美味しいです）。もちろん、複数の変数を用いた OR も可能です。たとえば、「`Pref` が"京都府"か `Score` が 3 以上」のような条件も可能ですが (`Pref == "京都府" | Score >= 3`)、実際、このような例はありません。よく使うのは「変数 X が a か b か c か」のような例です。ただし、

この場合は|を使わないもっと簡単な方法があります。それは第 10.4 章で紹介した %in% 演算子です。以下のコードは上のコードと同じものです。

```
1 df %>%
2   filter(Station %in% c("高田馬場駅", "三田駅")) %>%
3   select(Name, Station, ScoreN, Score)
```

```
## # A tibble: 14 x 4
##   Name           Station  ScoreN Score
##   <chr>          <chr>     <dbl>  <dbl>
## 1 麺達 うま家    高田馬場駅     2     3
## 2 らあ麺 やまぐち 高田馬場駅     7     4.08
## 3 博多一瑞亭 三田店 三田駅      0     NA
## 4 つけ麺屋 ひまわり 高田馬場駅     4     2.75
## 5 石器ラーメン 高田馬場 高田馬場駅     0     NA
## 6 旨辛らーめん 表裏 高田馬場駅     0     NA
## 7 三歩一         高田馬場駅     8     4.56
## 8 えぞ菊 戸塚店 高田馬場駅     4     3.62
## 9 麺屋 宗        高田馬場駅     5     4.2
## 10 とんこつラーメン 博多風龍 高田馬場店 高田馬場駅     2     3
## 11 横浜家系ラーメン 馬場壱家 高田馬場駅     0     NA
## 12 らーめん よし丸 高田馬場駅     1     5
## 13 札幌ラーメン どさん子 三田店 三田駅      0     NA
## 14 天下一品 三田店 三田駅      0     NA
```

結局、|が使われるケースがかなり限定されます。あるとすれば、「変数 X が a 以下か、b 以上か」のようなケースですね。ただし、&と|を同時に使うケースは考えられます。たとえば、大阪駅と京都駅周辺のうまいラーメン屋を調べるとします。問題は美味しさの基準ですが、3.5 点以上としましょう。ただし、京都府民はラーメンに非常に厳しく、3 点以上なら美味しいと仮定します。この場合、「(Station が"大阪駅"かつ Score >= 3.5)、または (Station が"京都駅"かつ Score >= 3)」のような条件が必要になります。() は「()の中から判定せよ」という、普通の算数での使い方と同じです。それでは、実際に検索してみましょう。

```

1 df %>%
2   filter((Station == "大阪駅" & Score >= 3.5) | (Station == "京都駅" & Score >= 3))
3   select(Name, Station, Walk, ScoreN, Score)

## # A tibble: 6 x 5
##   Name           Station  Walk ScoreN Score
##   <chr>          <chr>    <dbl>  <dbl>  <dbl>
## 1 Lei can ting 大阪ルクア店 大阪駅      3     3   4
## 2 神座 ルクア大阪店 大阪駅      1     10   3.94
## 3 みつか坊主 醸 大阪駅      10    4    5
## 4 無尽蔵 京都八条家 京都駅      5     2    3.5
## 5 中華料理 清華園 京都駅      10    3    5
## 6 ますたに 京都拉麺小路店 京都駅      9     3   3.67

```

Song が大好きな神座がヒットして嬉しいです。

## 12.5 行のソート

続いて、行のソートについて解説します。「食べログ」などのレビューサービスを利用する場合、口コミ評価が高い順で見るのが一般的でしょう<sup>3)</sup>。また、サッカーのランキングも多くの1位から下の順位で掲載されるのが一般的です。ここではこのようにある変数の値順に行を並び替える方法について説明します。

ソートには{dplyr}パッケージの `arrange()` 関数を使います。引数は変数名のみです。たとえば、奈良県のラーメン屋を検索してみましょう。並び替える順は駅から近い店舗を上位に、遠い店舗を下位に並べます。このような順は**昇順 (ascending)** と呼ばれ、ランキング表などでよく見ます。駅から近い順にソートするので、まず最寄りの駅情報が欠損でないことが必要です。また、ラーメン屋の評価も気になるので口コミが1つ以上付いている店舗に絞りましょう。表示する列は店舗名、最寄りの駅、徒歩距離、口コミ数、点数です。

---

<sup>3)</sup> サービスによってはこの機能が有料になっていたりもしますね。

```

1 df %>%
2   filter(Pref == "奈良県", !is.na(Station), ScoreN > 0) %>%
3   select(Name, Station, Walk, ScoreN, Score) %>%
4   arrange(Walk) %>%
5   print(n = Inf)

```

```

## # A tibble: 24 x 5
##   Name           Station   Walk ScoreN Score
##   <chr>          <chr>     <dbl>  <dbl>  <dbl>
## 1 麺屋 あまのじやく 本店    富雄駅      2      2  4.5
## 2 紀州和歌山らーめんきぶんや 奈良富雄店 富雄駅      4      1  4
## 3 ラーメン家 みつ葉       富雄駅      4      1  3.5
## 4 天下一品 新大宮店       新大宮駅     6      1  3
## 5 麺屋 一徳             天理駅      7      1  3
## 6 丸源ラーメン 檜原店    金橋駅      8      1  3.5
## 7 らーめん食堂 よってこや 平群店   元山上口駅    10     1  4
## 8 天理スタミナラーメン本店  楠本駅      11     2  3.25
## 9 博多長浜らーめん夢街道 奈良土橋店 真菅駅      11     1  3.5
## 10 ぶ~け             奈良駅      11     1  5
## 11 つけめん らーめん元喜神 押熊店 学研奈良登美ヶ丘駅 12     4  4.12
## 12 彩華ラーメン 本店       前森駅      12     5  3.6
## 13 力皇             天理駅      13     1  3.5
## 14 らーめん きみちゃん  京終駅      14     2  4.5
## 15 無鉄砲がむしゃら    帯解駅      15     2  4
## 16 彩華ラーメン 田原本店  石見駅      15     1  4
## 17 神座 大和高田店     大和高田駅    17     2  3.75
## 18 彩華ラーメン 奈良店   尼ヶ辻駅     17     3  4.33
## 19 彩華ラーメン 桜井店   大福駅      18     1  3
## 20 天下一品 東生駒店    東生駒駅     19     1  3.5
## 21 まりお流ラーメン    新大宮駅     20     1  5
## 22 どうとんぼり神座 奈良柏木店  西ノ京駅     22     1  3
## 23 河童ラーメン本舗 押熊店   学研奈良登美ヶ丘駅 28     1  4

```

```
## 24 博多長浜らーめん 夢街道 四条大路店 新大宮駅 29 4 2.88
```

3行まではこれまで習ってきたもので、4行目がソートの関数、`arrange()` です。引数はソートの基準となる変数で、今回は最寄りの駅からの徒歩距離を表す `Walk` です。5行目は省略可能ですが、`tibble` クラスの場合、10行までしか出力されないので、`print(n = Inf)` で「すべての行を表示」させます。`n` を指定することで出力される行数が調整可能です。奈良県のラーメン屋の中で最寄りの駅から最も近い店は「麺屋 あまのじゃく 本店」で徒歩 2 分でした。京田辺店も駅から約 2 分ですし、近いですね。ちなみに `Song` はこの塩とんこつが好きです。世界一こってりなラーメンとも言われる「チョモランマ」で有名な「まりお流ラーメン」は新大宮駅から徒歩 20 分でかなり遠いことが分かります。

続いて、駅からの距離ではなく、評価が高い順にしてみましょう。評価が高いほど上に来るので、今回は昇順でなく、**降順 (descending)** でソートする必要があります。`arrange()` 関数は基本的に、指定された変数を基準に昇順でソートします。降順にするためには `desc()` 関数を更に用います。たとえば、`arrange(desc(変数名))` のようにです。それでは実際にやってみましょう。上のコードの 4 行目を `arrange(Walk)` から `arrange(desc(Score))` にちょっと修正するだけです。

```
1 df %>%
2   filter(Pref == "奈良県", !is.na(Station), ScoreN > 0) %>%
3   select(Name, Station, Walk, ScoreN, Score) %>%
4   arrange(desc(Score)) %>%
5   print(n = Inf)

## # A tibble: 24 x 5
##   Name           Station   Walk  ScoreN Score
##   <chr>          <chr>     <dbl>   <dbl>   <dbl>
## 1 まりお流ラーメン 新大宮駅     20      1     5
## 2 ぶ~け          奈良駅      11      1     5
## 3 麺屋 あまのじゃく 本店 富雄駅      2      2     4.5
## 4 らーめん きみちゃん 京終駅      14      2     4.5
## 5 彩華ラーメン 奈良店  尼ヶ辻駅     17      3     4.33
## 6 つけめん らーめん元喜神 押熊店 学研奈良登美ヶ丘駅 12      4     4.12
## 7 河童ラーメン本舗 押熊店  学研奈良登美ヶ丘駅 28      1     4
```

## 8 無鉄砲がむしゃら	帯解駅	15	2	4
## 9 紀州和歌山らーめんきぶんや 奈良富雄店	富雄駅	4	1	4
## 10 彩華ラーメン 田原本店	石見駅	15	1	4
## 11 らーめん食堂 よってこや 平群店	元山上口駅	10	1	4
## 12 神座 大和高田店	大和高田駅	17	2	3.75
## 13 彩華ラーメン 本店	前栽駅	12	5	3.6
## 14 天下一品 東生駒店	東生駒駅	19	1	3.5
## 15 力皇	天理駅	13	1	3.5
## 16 博多長浜らーめん夢街道 奈良土橋店	真菅駅	11	1	3.5
## 17 ラーメン家 みつ葉	富雄駅	4	1	3.5
## 18 丸源ラーメン 檜原店	金橋駅	8	1	3.5
## 19 天理スタミナラーメン本店	櫟本駅	11	2	3.25
## 20 麺屋 一徳	天理駅	7	1	3
## 21 どうとんぼり神座 奈良柏木店	西ノ京駅	22	1	3
## 22 彩華ラーメン 桜井店	大福駅	18	1	3
## 23 天下一品 新大宮店	新大宮駅	6	1	3
## 24 博多長浜らーめん 夢街道 四条大路店	新大宮駅	29	4	2.88

よく考えてみれば、「評価が同点の場合、どうなるの?」と疑問を抱く方がいるかも知れません。たとえば、7行目の「河童ラーメン本舗 押熊店」と8行目の「無鉄砲がむしゃら」はどちらも評価が4点ですが、「河童ラーメン本舗 押熊店」が先に表示されます。そのこれは簡単です。同点の場合、データセット内で上に位置する行が先に表示されます。これを確認するには `which()` 関数を使います。() 内に条件文を指定することで、この条件に合致する要素の位置を返します。もし、条件に合致するものが複数あった場合は全ての位置を返します<sup>4)</sup>。

```
1 which(df$name == "河童ラーメン本舗 押熊店")
```

```
## [1] 6021
```

<sup>4)</sup> たとえば、データ内に「ラーメンショップ」という店舗は3店舗あり、この場合、長さ3のベクトルが返されます。

```

1  which(df$name == "無鉄砲がむしゃら")
## [1] 6040

```

データ内に「河童ラーメン本舗 押熊店」がより上に位置することが分かります。「もし同点なら口コミ評価数が多いところにしたい」場合はどうすれば良いでしょうか。これは `arrange()` 内に変数名を足すだけで十分です。

```

1  df %>%
2    filter(Pref == "奈良県", !is.na(Station), ScoreN > 0) %>%
3    select(Name, Station, Walk, ScoreN, Score) %>%
4    arrange(desc(Score), desc(ScoreN)) %>%
5    print(n = Inf)

## # A tibble: 24 x 5
##   Name           Station     Walk ScoreN Score
##   <chr>          <chr>      <dbl>  <dbl> <dbl>
## 1 まりお流ラーメン 新大宮駅     20     1     5
## 2 ぶ~け          奈良駅      11     1     5
## 3 麺屋 あまのじやく 本店  富雄駅      2     2     4.5
## 4 らーめん きみちゃん  京終駅      14     2     4.5
## 5 彩華ラーメン 奈良店  尼ヶ辻駅     17     3     4.33
## 6 つけめん らーめん元喜神 押熊店 学研奈良登美ヶ丘駅 12     4     4.12
## 7 無鉄砲がむしゃら  帯解駅      15     2     4
## 8 河童ラーメン本舗 押熊店  学研奈良登美ヶ丘駅 28     1     4
## 9 紀州和歌山らーめんきぶんや 奈良富雄店 富雄駅      4     1     4
## 10 彩華ラーメン 田原本店  石見駅      15     1     4
## 11 らーめん食堂 よってこや 平群店 元山上口駅     10     1     4
## 12 神座 大和高田店    大和高田駅    17     2     3.75
## 13 彩華ラーメン 本店    前栽駅      12     5     3.6
## 14 天下一品 東生駒店  東生駒駅     19     1     3.5
## 15 力皇          天理駅      13     1     3.5
## 16 博多長浜らーめん夢街道 奈良土橋店 真菅駅      11     1     3.5

```

## 17 ラーメン家 みつ葉	富雄駅	4	1	3.5
## 18 丸源ラーメン 檜原店	金橋駅	8	1	3.5
## 19 天理スタミナラーメン本店	櫻本駅	11	2	3.25
## 20 麺屋 一徳	天理駅	7	1	3
## 21 どうとんぼり神座 奈良柏木店	西ノ京駅	22	1	3
## 22 彩華ラーメン 桜井店	大福駅	18	1	3
## 23 天下一品 新大宮店	新大宮駅	6	1	3
## 24 博多長浜らーめん 夢街道 四条大路店	新大宮駅	29	4	2.88

ソートの基準は `arrange()` 内において先に指定された変数の順番となります。「口コミ評価も評価数も同じなら、駅から近いところにしたい」場合は変数が3つとなり、`Score`、`ScoreN`、`Walk` の順で入れます。

```

1 df %>%
2   filter(Pref == "奈良県", !is.na(Station), ScoreN > 0) %>%
3   select(Name, Station, Walk, ScoreN, Score) %>%
4   arrange(desc(Score), desc(ScoreN), Walk) %>%
5   print(n = Inf)

```

		Station	Walk	ScoreN	Score
	Name	<chr>	<dbl>	<dbl>	<dbl>
## 1 ぶ~け	奈良駅		11	1	5
## 2 まりお流ラーメン	新大宮駅		20	1	5
## 3 麺屋 あまのじやく 本店	富雄駅		2	2	4.5
## 4 らーめん きみちゃん	京終駅		14	2	4.5
## 5 彩華ラーメン 奈良店	尼ヶ辻駅		17	3	4.33
## 6 つけめん らーめん元喜神 押熊店	学研奈良登美ヶ丘駅	12	4	4.12	
## 7 無鉄砲がむしゃら	帯解駅	15	2	4	
## 8 紀州和歌山らーめんきぶんや 奈良富雄店	富雄駅	4	1	4	
## 9 らーめん食堂 よってこや 平群店	元山上口駅	10	1	4	
## 10 彩華ラーメン 田原本店	石見駅	15	1	4	
## 11 河童ラーメン本舗 押熊店	学研奈良登美ヶ丘駅	28	1	4	
## 12 神座 大和高田店	大和高田駅	17	2	3.75	

## 13 彩華ラーメン 本店	前栽駅	12	5	3.6
## 14 ラーメン家 みつ葉	富雄駅	4	1	3.5
## 15 丸源ラーメン 檜原店	金橋駅	8	1	3.5
## 16 博多長浜らーめん夢街道 奈良土橋店	真菅駅	11	1	3.5
## 17 力皇	天理駅	13	1	3.5
## 18 天下一品 東生駒店	東生駒駅	19	1	3.5
## 19 天理スタミナラーメン本店	櫟本駅	11	2	3.25
## 20 天下一品 新大宮店	新大宮駅	6	1	3
## 21 麺屋 一徳	天理駅	7	1	3
## 22 彩華ラーメン 桜井店	大福駅	18	1	3
## 23 どうとんぼり神座 奈良柏木店	西ノ京駅	22	1	3
## 24 博多長浜らーめん 夢街道 四条大路店	新大宮駅	29	4	2.88

---

### 練習問題

## 第 13 章

# データハンドリング [基礎編: 拡張]

前章ではデータの一部 (subset) を抽出する方法について説明しましたが、本章はデータを拡張する、あるいは全く別のデータが得られるような処理について解説します。後者は主に元のデータを要約し (記述統計量)、その結果を出力する方法で、前者はデータ内の変数に基づき、指定された計算を行った結果を新しい列として追加する方法です。今回も前章と同じデータを使用します。

```
1 pacman::p_load(tidyverse)
2 # データのパスは適宜修正すること
3 # 文字化けが生じる場合、以下のコードに書き換える。
4 # df <- read_csv("Data/Ramen.csv", locale = locale(encoding = "utf8"))
5 df <- read_csv("Data/Ramen.csv")
```

データの詳細については第 12.3 章を参照してください。

---

## 13.1 記述統計量の計算

### 13.1.1 summarise() による記述統計量の計算

ある変数の平均値や標準偏差、最小値、最大値などの記述統計量（要約統計量）を計算することも可能です。これは `summarize()` または `summarise()` 関数を使いますが、この関数は後で紹介する `group_by()` 関数と組み合わせることで力を発揮します。ここではグルーピングを考えずに、全データの記述統計量を計算する方法を紹介します。

`summarise()` 関数の使い方は以下の通りです。

```
1 # summarise() 関数の使い方
2 データフレーム名 %>%
3   summarise(新しい変数名 = 関数名 (計算の対象となる変数名))
```

もし、`Score` 変数の平均値を計算し、その結果を `Mean` という列にしたい場合は以下のよ

うなコードになります。

```
1 df %>%
2   summarise(Mean = mean(Score))

## # A tibble: 1 x 1
##      Mean
##      <dbl>
## 1      NA
```

ただし、`mean()` 関数は欠損値が含まれるベクトルの場合、`NA` を返します。この場合方法は2つ考えられます。

1. `filter()` 関数を使って `Score` が欠損しているケースを予め除去する。
2. `na.rm` 引数を指定し、欠損値を除去した平均値を求める。

ここでは2番目の方法を使います。

```

1 df %>%
2   summarise(Mean = mean(Score, na.rm = TRUE))

## # A tibble: 1 x 1
##   Mean
##   <dbl>
## 1 3.66

```

df の Score 変数の平均値は NA であることが分かります。また、`summarise()` 関数は複数の記述統計量を同時に計算することも可能です。以下は Score 変数の平均値、中央値、標準偏差、最小値、最大値、第一四分位点、第三四分位点を計算し、`score_desc` という名のデータフレームに格納するコードです。

```

1 score_desc <- df %>%
2   summarize(Mean      = mean(Score,           na.rm = TRUE), # 平均値
3             Median    = median(Score,          na.rm = TRUE), # 中央値
4             SD        = sd(Score,            na.rm = TRUE), # 標準偏差
5             Min       = min(Score,           na.rm = TRUE), # 最小値
6             Max       = max(Score,            na.rm = TRUE), # 最大値
7             Q1        = quantile(Score, 0.25, na.rm = TRUE), # 第一四分位点
8             Q3        = quantile(Score, 0.75, na.rm = TRUE)) # 第三四分位点
9
10 score_desc

```

```

## # A tibble: 1 x 7
##   Mean Median SD  Min  Max   Q1   Q3
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 3.66  3.58  0.719     1     5     3     4

```

もちろん、複数の変数に対して記述統計量を計算することも可能です。たとえば、平均予算 (Budget)、口コミ数 (ScoreN)、口コミ評価 (Score) の平均値を求めるしたら、

```

1 df %>%
2   summarize(Budget_Mean = mean(Budget, na.rm = TRUE), # 平均予算の平均値
3             ScoreN_Mean = mean(ScoreN, na.rm = TRUE), # 口コミ数の平均値

```

```

4   Score_Mean  = mean(Score,  na.rm = TRUE) # 評価の平均値

## # A tibble: 1 x 3
##   Budget_Mean ScoreN_Mean Score_Mean
##       <dbl>      <dbl>      <dbl>
## 1     1232.     0.537     3.66

```

のように書きます。実は `summarise()` はこれくらいで十分便利です。ただし、以上の操作はもっと簡単なコードに置換できます。ただし、ラムダ式など、やや高度な内容になるため、以下の内容は飛ばして、次の節（グルーピング）を読んでいただいても構いません。

まずは、複数の変数に対して同じ記述統計量を求める例を考えてみましょう。たとえば、`Budget`、`ScoreN`、`Score` に対して平均値を求める例です。これは `across()` 関数を使うとよりコードが短くなります。まずは `across()` 関数の書き方から見ましょう。

```

1 # across() の使い方
2 データフレーム名 %>%
3   summarise(across(変数名のベクトル, 記述統計を計算する関数名, 関数の引数))

```

変数名のベクトルは長さ 1 以上のベクトルです。たとえば、`Budget`、`ScoreN`、`Score` の場合 `c(Budget, ScoreN, Score)` になります。これは `df` 内で隣接する変数ですから `Budget:Score` の書き方も使えます。また、`where()` や `any_of()`、`starts_with()` のような関数を使って変数を指定することも可能です。関数名は `mean` や `sd` などの関数名です。ここは関数名 () でなく、関数名であることに注意してください。引数は前の関数に必要な引数です。引数を必要としない関数なら省略可能ですが、`na.rm = TRUE` などの引数が必要な場合は指定する必要があります。それでは `Budget`、`ScoreN`、`Score` の平均値を計算してみましょう。

```

1 df %>%
2   summarize(across(Budget:Score, mean, na.rm = TRUE))

## # A tibble: 1 x 3
##   Budget ScoreN Score
##       <dbl>    <dbl> <dbl>
## 1     1232.    0.537  3.66

```

`across()` 使わない場合、4行必要だったコードが2行になりました。変数が少ない場合は `across()` を使わない方が、可読性が高くなる場合もあります。しかし、変数が多くなる場合、可読性がやや落ちても `across()` を使った方が効率的でしょう。

次は、ある変数に対して複数の記述統計量を計算したい場合について考えます。`Budget`、`ScoreN`、`Score` 変数の第一四分位点と第三四分位点を `across()` を使わずに計算すると家のような7行のコードになります。

```

1 df %>%
2   summarize(Budget_Q1 = quantile(Budget, 0.25, na.rm = TRUE),
3             Budget_Q3 = quantile(Budget, 0.75, na.rm = TRUE),
4             ScoreN_Q1 = quantile(ScoreN, 0.25, na.rm = TRUE),
5             ScoreN_Q3 = quantile(ScoreN, 0.75, na.rm = TRUE),
6             Score_Q1  = quantile(Score, 0.25, na.rm = TRUE),
7             Score_Q3  = quantile(Score, 0.75, na.rm = TRUE))

## # A tibble: 1 x 6
##   Budget_Q1 Budget_Q3 ScoreN_Q1 ScoreN_Q3 Score_Q1 Score_Q3
##       <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1     800      1000        0          0        3        4

```

この作業も `across()` を使ってより短縮することができます。ここではラムダ式の知識が必要になります。ラムダ関数とは関数名を持たない無名関数 (anonymous functions) を意味しますが、詳細は割愛します。興味のある読者は Wikipedia などを参照してください。簡単にいうとその場で即席に関数を作成し、計算が終わったら破棄する関数です。ただ、R は基本的にラムダ式を提供しているのではなく、`purrr` パッケージのラムダ式スタイルを使用します。まずは、書き方から確認します。

```

1 # ラムダ式を用いた across() の使い方
2 データフレーム名 %>%
3   summarise(across(変数名のベクトル, .fns = list(結果の変数名 = ラムダ式)))

```

先ほどの書き方と似ていますが、関数を複数書く必要があるため、今回は関数名を list 型にまとめ、`.fns` 引数に指定します。そして、結果の変数名は結果として出力されるデータフレームの列名を指定する引数です。たとえば、`Mean` にすると結果は元の変数名

`1_Mean`、元の変数名 `2_Mean`...のように出力されます。そして、ラムダ式が実際の関数が入る箇所です。とりあえず今回はコードを走らせ、結果から確認してみましょう。

```

1 df %>%
2   summarize(across(Budget:Score,
3     .fns = list(Q1 = ~quantile(.x, 0.25, na.rm = TRUE),
4                 Q3 = ~quantile(.x, 0.75, na.rm = TRUE))))
5
6
7 ## # A tibble: 1 x 6
8 ##   Budget_Q1 Budget_Q3 ScoreN_Q1 ScoreN_Q3 Score_Q1 Score_Q3
9 ##     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
10## 1       800     1000        0        0        3        4

```

結果の列名が `Budget_Q1`、`Budget_Q3`、`ScoreN_Q1`...のようになります。それぞれの変数の第一四分位点と第三四分位点が出力されます。問題はラムダ式の方ですが、普通の関数に非常に近いことが分かります。`across()` 内のラムダ式は~関数名 (`.x`, その他の引数) のような書き方になります。関数名の前に~が付いていることに注意してください。分位数を求める関数は `quantile()` であり、`quantile(ベクトル, 分位数)` であり、必要に応じて `na.rm` を付けます。この分位数が 0.25 なら第一四分位点、0.5 なら第二四分位点 (= 中央値)、0.75 なら第三四分位点になります。それではラムダ式`~quantile(.x, 0.25, na.rm = TRUE)` はどういう意味でしょうか。これは `.x` の箇所に `Budget` や `ScoreN`、`Score` が入ることを意味します。`.x` という書き方は決まりです。`.y` とか `.Song-san-Daisuki` などはダメです。そして、0.25 を付けることによって第一四分位点を出力するように指定します。また、`Budget`、`ScoreN`、`Score` に欠損値がある場合、無視するように `na.rm = TRUE` を付けます。

ラムダ式を第10章で解説した自作関数で表現すると、以下のようになります。

```

1 # 以下の3つは同じ機能をする関数である
2
3 # ラムダ式
4 ~quantile(.x, 0.25, na.rm = TRUE)
5
6 # 一般的な関数の書き方 1

```

```
7  名無し関数 <- function(x) {  
8      quantile(x, 0.25, na.rm = TRUE)  
9  }  
10  
11  # 一般的な関数の書き方 2  
12  名無し関数 <- function(x) quantile(x, 0.25, na.rm = TRUE)
```

この3つは全て同じですが、ラムダ式は関数名を持たず、その場で使い捨てる関数です。もちろん、ラムダ式を使わずに事前に第一四分位点と第三四分位点を求める関数を予め作成し、ラムダ式の代わりに使うことも可能です。まずは第一四分位点と第三四分位点を求める自作関数 FuncQ1 と FuncQ2 を作成します。

```
1  # ラムダ式を使わない場合は事前に関数を定義しておく必要がある  
2  FuncQ1 <- function(x) {  
3      quantile(x, 0.25, na.rm = TRUE)  
4  }  
5  FuncQ3 <- function(x) {  
6      quantile(x, 0.75, na.rm = TRUE)  
7  }
```

後は先ほどのほぼ同じ書き方ですが、今回はラムダ式を使わないと関数名に~を付けて、関数名のみで十分です。() も不要です。

```
1  # やっておくと、summarise() 文は簡潔になる  
2  df %>%  
3  summarize(across(Budget:Score, list(Q1 = FuncQ1, Q3 = FuncQ3)))
```

```
## # A tibble: 1 x 6  
##   Budget_Q1 Budget_Q3 ScoreN_Q1 ScoreN_Q3 Score_Q1 Score_Q3  
##       <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>  
## 1       800     1000         0         0         3         4
```

事前に関数を用意するのが面倒ですが、across() の中身はかなりスッキリしますね。もし、このような作業を何回も行うなら、ラムダ式を使わず、自作関数を用いることも可能

です。ただし、自作関数であっても引数が2つ以上必要な場合はラムダ式を使います。

### 13.1.2 summarise() に使える便利な関数

以下の内容は後で説明する group\_by() 関数を使っているため、まだ group\_by() に馴染みのない読者はまずはここを読み飛ばし、グルーピングの節にお進みください。

#### IQR(): 四分位範囲を求める

四分位範囲は第三四分位点から第一四分位点を引いた値であり、Rの内蔵関数である IQR() を使えば便利です。この関数は mean や sd() 関数と同じ使い方となります。

```

1 df %>%
2   filter(!is.na(Walk)) %>% # 予め欠損したケースを除くと、後で na.rm = TRUE が不要
3   group_by(Pref) %>%
4     summarise(Mean      = mean(Walk) ,
5                SD        = sd(Walk) ,
6                IQR       = IQR(Walk) ,
7                N         = n() ,
8                .groups = "drop") %>%
9   arrange(Mean)

## # A tibble: 9 x 5
##   Pref      Mean     SD     IQR     N
##   <chr>    <dbl>  <dbl>  <dbl>  <int>
## 1 東京都    4.29   4.49    4    919
## 2 大阪府    5.92   6.08    6    932
## 3 神奈川県  8.21   7.91   10    878
## 4 京都府    8.38   6.95    9    339
## 5 兵庫県    8.52   7.27   10    484
## 6 奈良県    10.6   6.59   10    123
## 7 千葉県    10.6   8.21   12    776
## 8 埼玉県    11.6   8.99   14    817
## 9 和歌山県  12.8   6.83    9    107

```

**first()、last()、nth(): n 番目の要素を求める**

稀なケースかも知れませんが、データ内、またはグループ内の n 番目の行を抽出する時があります。たとえば、市区町村の情報が格納されているデータセットで、人口が大きい順でデータがソートされているとします。各都道府県ごとに最も人口が大きい市区町村のデータ、あるいは最も少ない市区町村のデータが必要な際、first() と last() 関数が有効です。

それでは各都道府県ごとに「最も駅から遠いラーメン屋」の店舗名と最寄りの駅からの徒歩距離を出力したいとします。まずは、徒歩距離のデータが欠損しているケースを除去し、データを徒歩距離順でソートします。これは filter() と arrange() 関数を使えば簡単です。続いて、group\_by() を使って都府県単位でデータをグループ化します。最後に summarise() 関数内に last() 関数を使います。データは駅から近い順に鳴っているため、各都府県内の最後の行は駅から最も遠い店舗になるからです。

```
1 df %>%
2   filter(!is.na(Walk)) %>%
3   arrange(Walk) %>%
4   group_by(Pref) %>%
5   summarise(Farthest  = last(Name),
6             Distance = last(Walk))
```

```
## # A tibble: 9 x 3
##   Pref   Farthest           Distance
##   <chr>  <chr>                <dbl>
## 1 京都府  热烈らあめん          30
## 2 兵庫県  濃厚醤油 中華そば いせや 玉津店 43
## 3 千葉県  札幌ラーメン どさん子 佐原 51 号店 59
## 4 和歌山県 中華そば まる乃          30
## 5 埼玉県  札幌ラーメン どさん子 小鹿野店 116
## 6 大阪府  河童ラーメン本舗 岸和田店 38
## 7 奈良県  博多長浜らーめん 夢街道 四条大路店 29
## 8 東京都  てんがら 青梅新町店          30
## 9 神奈川県 札幌ラーメン どさん子 中津店 73
```

この `last()` を `first()` に変えると、最寄りの駅から最も近い店舗情報が表示されます。また、「`n` 番目の情報」が必要な際は `nth()` 関数を使います。`nth(Name, 2)` に変えることで 2 番目の店舗名が抽出できます。

### `n_distinct()`: ユニーク値の個数を求める

`n_distinct()` は何種類の要素が含まれているかを計算する関数であり、`length(unique())` 関数と同じ機能をします。たとえば、以下の `myVec1` に対して何種類の要素があるかを確認してみましょう。

```
1 myVec1 <- c("A", "B", "B", "D", "A", "B", "D", "C", "A")
2
3 unique(myVec1)
```

```
## [1] "A" "B" "D" "C"
```

`myVec1` は "A"、"B"、"D"、"C" の要素で構成されていることが分かります。これが `myVec1` の **ユニーク値 (unique values)** です。そして、このユニーク値の個数を調べるために `length()` を使います。

```
1 length(unique(myVec1))
```

```
## [1] 4
```

これで `myVec1` は 4 種類の値が存在することが分かります。これと全く同じ機能をする関数が `n_distinct()` です。

```
1 n_distinct(myVec1)
```

```
## [1] 4
```

この関数を `summarise()` に使うことで、都府県ごとに駅の個数が分かります。あるいは「東京都内の選挙区に、これまでの衆院選において何人の候補者が存在したか」も分かります。ここでは `df` 内の都府県ごとに駅の個数を計算してみましょう。最後の駅数が多い順でソートします。

```
1 df %>%
2   filter(!is.na(Station)) %>% # 最寄りの駅が欠損しているケースを除去
```

```
3   group_by(Pref) %>%
4     summarise(N_Station = n_distinct(Station),
5               .groups    = "drop") %>%
6     arrange(desc(N_Station))
```

```
## # A tibble: 9 x 2
##   Pref      N_Station
##   <chr>     <int>
## 1 東京都      368
## 2 大阪府      341
## 3 千葉県      241
## 4 神奈川県    240
## 5 兵庫県      199
## 6 埼玉県      185
## 7 京都府      123
## 8 奈良県      52
## 9 和歌山県    46
```

当たり前かも知れませんが、駅数が最も多いのは東京都で次が大阪府であることが分かります。

### any()、all(): 条件に合致するか否かを求める

any() と all() はベクトル内の全要素に対して条件に合致するか否かを判定する関数です。ただし、any() は一つの要素でも条件に合致すれば TRUE を、全要素が合致しない場合 FALSE を返します。一方、all() は全要素に対して条件を満たせば TRUE、一つでも満たさない要素があれば FALSE を返します。以下は any() と all() の例です。

```
1 myVec1 <- c(1, 2, 3, 4, 5)
2 myVec2 <- c(1, 3, 5, 7, 11)
3
4 any(myVec1 %% 2 == 0) # myVec1 を 2 で割った場合、一つでも余りが 0 か
## [1] TRUE
```

```

1  all(myVec1 %% 2 == 0) # myVec1 を 2 で割った場合、全ての余りが 0 か
## [1] FALSE

1  all(myVec2 %% 2 != 0) # myVec2 を 2 で割った場合、全ての余りが 0 ではないか
## [1] TRUE

```

それでは実際に df に対して any() と all() 関数を使ってみましょう。一つ目は「ある都府県に最寄りの駅から徒歩 60 分以上の店舗が一つでもあるか」であり、二つ目は「ある都府県の店舗は全て最寄りの駅から徒歩 30 分以下か」です。それぞれの結果を Over60 と Within30 という列で出力してみましょう。

```

1  df %>%
2    group_by(Pref) %>%
3    summarise(Over60    = any(Walk >= 60, na.rm = TRUE),
4               Within30 = all(Walk <= 30, na.rm = TRUE),
5               .groups   = "drop")

## # A tibble: 9 x 3
##   Pref      Over60 Within30
##   <chr>    <lgl>   <lgl>
## 1 京都府    FALSE    TRUE
## 2 兵庫県    FALSE    FALSE
## 3 千葉県    FALSE    FALSE
## 4 和歌山県  FALSE    TRUE
## 5 埼玉県    TRUE     FALSE
## 6 大阪府    FALSE    FALSE
## 7 奈良県    FALSE    TRUE
## 8 東京都    FALSE    TRUE
## 9 神奈川県  TRUE     FALSE

```

埼玉県と神奈川県において、最寄りの駅から徒歩 60 以上の店がありました。また、京都府、東京都、奈良県、和歌山県の場合、全店舗が最寄りの駅から徒歩 30 分以下ということが分かります。当たり前ですが Over60 が TRUE なら Within30 は必ず FALSE になります。

ますね。

---

## 13.2 グルーピング

### 13.2.1 group\_by() によるグループ化

先ほどの `summarise()` 関数は確かに便利ですが、特段に便利とも言いにくいです。`df` の `Score` の平均値を計算するだけなら、`summarise()` 関数を使わない方が楽です。

```
1 # これまでのやり方
2 df %>%
3   summarise(Mean = mean(Score, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   Mean
##   <dbl>
## 1 3.66
```

```
1 # 普通にこれでええんちゃう?
2 mean(df$Score, na.rm = TRUE)
```

```
## [1] 3.663457
```

しかし、これをグループごとに計算するならどうでしょう。たとえば、`Score` の平均値を都府県ごとに計算するとします。この場合、以下のようなコードになります。

```
1 mean(df$Score[df$Pref == "東京都"], na.rm = TRUE)
```

```
## [1] 3.674256
```

```
1 mean(df$Score[df$Pref == "神奈川県"], na.rm = TRUE)
```

```
## [1] 3.533931
```

```
1 mean(df$Score[df$Pref == "千葉県"], na.rm = TRUE)  
  
## [1] 3.715983  
  
1 mean(df$Score[df$Pref == "埼玉県"], na.rm = TRUE)  
  
## [1] 3.641573  
  
1 mean(df$Score[df$Pref == "大阪府"], na.rm = TRUE)  
  
## [1] 3.765194  
  
1 mean(df$Score[df$Pref == "京都府"], na.rm = TRUE)  
  
## [1] 3.684976  
  
1 mean(df$Score[df$Pref == "兵庫県"], na.rm = TRUE)  
  
## [1] 3.543936  
  
1 mean(df$Score[df$Pref == "奈良県"], na.rm = TRUE)  
  
## [1] 3.854762  
  
1 mean(df$Score[df$Pref == "和歌山県"], na.rm = TRUE)  
  
## [1] 3.96999
```

変わったのは df\$Score が df\$Score[df\$Pref == "東京都"] に変わっただけです。 df\$Pref が "東京都" であるか否かを TRUE と FALSE で判定し、これを基準に df\$Score を抽出する仕組みです。 df\$Score と df\$Pref は同じデータフレームですから、このような書き方で問題ありません。

これだけでもかなり書くのが面倒ですが、これが 47 都道府県なら、あるいは 200 ヶ国ならかなり骨の折れる作業でしょう。ここで大活躍するのが {dplyr} パッケージの group\_by() 関数です。引数はグループ化する変数名だけです。先ほどの作業を {dplyr} を使うなら Pref 変数でグループ化し、 summarise() 関数で平均値を求めるだけです。今回は Score だけでなく、ScoreN の平均値も求めてみましょう。そして、評価が高い順

にソートもしてみます。

```
1 # ScoreN と Score の平均値を Pref ごとに求める
2 df %>%
3   group_by(Pref) %>%
4   summarise(ScoreN_Mean = mean(ScoreN, na.rm = TRUE),
5             Score_Mean = mean(Score, na.rm = TRUE)) %>%
6   arrange(desc(Score_Mean))
```

```
## # A tibble: 9 x 3
##   Pref     ScoreN_Mean Score_Mean
##   <chr>       <dbl>      <dbl>
## 1 和歌山県     0.593      3.97
## 2 奈良県       0.306      3.85
## 3 大阪府       0.516      3.77
## 4 千葉県       0.259      3.72
## 5 京都府       0.522      3.68
## 6 東京都       1.16       3.67
## 7 埼玉県       0.278      3.64
## 8 兵庫県       0.389      3.54
## 9 神奈川県     0.587      3.53
```

評判が最も高い都府県は和歌山県、最も低いのは神奈川県ですね。Song も和歌山ラーメンは井出系も車庫前系も好きです。しかし、大事なのは「井出系」と「車庫前系」といった分類が正しいかどうかではありません。コードが非常に簡潔となり、ソートなども自由自在であることです。都府県ごとに ScoreN と Score の平均値を求める場合、dplyr() を使わなかつたら 18 行のコードとなり、ソートも自分でやる必要があります。一方、group\_by() 関数を使うことによってコードが 5 行になりました。

また、これは 2020 年 6 月に公開された{dplyr}1.0.0 からの問題ですが、group\_by() の後に summarise() を使うと以下のようなメッセージが出力されます。

```
## `summarise()`' ungrouping output (override with ` `.groups` argument)
```

これは group\_by() で指定された変数のグループ化が自動的に解除されたことを意味し

ます。なぜなら `summarise()` をする際は `Pref` をグループ変数として使いましたが、出力された結果の `Pref` 変数はもはやグループとして機能できなくなるからです。元の `df` には `Pref` が"東京都"だったケースが 1000 行、"京都府"だったのが 414 行だったので、`Pref` 変数でグループ化する意味がありました。しかし、`summarise()` から得られたデータフレームは `Pref == "東京都"` の行が 1 つしかありません。これはグループ化する意味がなくなったことを意味します。したがって、自動的にグループを解除してくれます。自動的にやってくれるのはありがたいことですが、可能ならば関数内に自分で明記することが推奨されます。そこで使う引数が`.groups` であり、"drop"を指定すると**全ての**グループ化変数を解除します。以下のようなコードだと先ほどのメッセージが表示されません。今後、意識的に入れるようにしましょう。

```

1 # ScoreN と Score の平均値を Pref ごとに求める
2 df %>%
3   group_by(Pref) %>%
4   summarise(ScoreN_Mean = mean(ScoreN, na.rm = TRUE),
5             Score_Mean = mean(Score, na.rm = TRUE),
6             .groups      = "drop") %>%
7   arrange(desc(Score_Mean))

## # A tibble: 9 x 3
##   Pref      ScoreN_Mean Score_Mean
##   <chr>        <dbl>      <dbl>
## 1 和歌山県     0.593      3.97
## 2 奈良県       0.306      3.85
## 3 大阪府       0.516      3.77
## 4 千葉県       0.259      3.72
## 5 京都府       0.522      3.68
## 6 東京都        1.16       3.67
## 7 埼玉県       0.278      3.64
## 8 兵庫県       0.389      3.54
## 9 神奈川県     0.587      3.53

```

続いて、一つ便利な関数を紹介します。それはグループのサイズを計算する関数、`n()`

です。この関数を `summarise()` 内に使うと、各グループに属するケース数を出力します<sup>1)</sup>。先ほどのコードを修正し、各グループのサイズを `N` という名の列として追加してみましょう。そしてソートの順番は `N` を最優先とし、同じ場合は `Score_Mean` が高い方を上に出力させます。また、`ScoreN_Mean` の前に、口コミ数の合計も出してみましょう。

```

1  # Pref ごとに口コミ数の合計、口コミ数の平均値、評価の平均値、店舗数を求める
2  # 店舗数-評価の平均値順でソートする
3  df %>%
4    group_by(Pref) %>%
5    summarise(ScoreN_Sum  = sum(ScoreN,  na.rm = TRUE),
6               ScoreN_Mean = mean(ScoreN,  na.rm = TRUE),
7               Score_Mean  = mean(Score,   na.rm = TRUE),
8               N           = n(),
9               .groups     = "drop") %>%
10   arrange(desc(N), desc(Score_Mean))

```

```

## # A tibble: 9 x 5
##   Pref      ScoreN_Sum ScoreN_Mean Score_Mean      N
##   <chr>        <dbl>      <dbl>      <dbl> <int>
## 1 大阪府        516       0.516      3.77  1000
## 2 千葉県        259       0.259      3.72  1000
## 3 東京都       1165       1.16       3.67  1000
## 4 埼玉県        278       0.278      3.64  1000
## 5 神奈川県      587       0.587      3.53  1000
## 6 兵庫県        230       0.389      3.54   591
## 7 京都府        216       0.522      3.68   414
## 8 奈良県         45        0.306      3.85   147
## 9 和歌山県       83        0.593      3.97   140

```

記述統計をグループごとに求めるのは普通にあり得るケースですし、実験データの場合

<sup>1)</sup> 今回の例のように記述統計量とケース数を同時に計算する場合は `n()` を使いますが、ケース数のみを計算する場合は `group_by()` と `summarise()` と組み合わせず、`count()` 関数だけ使うことも可能です。グループ化変数（ここでは `Pref`）を `count()` の引数として指定すると、`Pref` の値ごとのケース数が表示されます。

はほぼ必須の作業でう。統制群と処置群においてグループサイズが均一か、共変量のバラツキが十分に小さいかなどを判断する際に `group_by()` と `summarise()` 関数の組み合わせは非常に便利です。

### 13.2.2 複数の変数を用いたグループ化

グループ化変数は2つ以上指定することも可能です。たとえば、都府県 (Pref) と最寄りの駅の路線 (Line) でグループ化することも可能です。それでは Pref と Line でグループ化し、店舗数と口コミ数、評価の平均値を計算し、ソートの順番は店舗数、店舗数が同じなら評価の平均値が高い順にしましょう。今回も `summarise()` 内に `.group = "drop"` を指定し、グループ化を解除します。今回は Top 20 まで出してみましょう。

```

1 # ScoreN と Score の平均値を Pref ごとに求める
2 df %>%
3   filter(!is.na(Line)) %>% # Line が欠損していないケースのみ残す
4   group_by(Pref, Line) %>% # Pref と Line でグループ化
5   summarise(N          = n(),
6             ScoreN_Sum = sum(ScoreN, na.rm = TRUE),
7             Score_Mean = mean(Score, na.rm = TRUE),
8             .groups    = "drop") %>%
9   arrange(desc(N), desc(Score_Mean)) %>%
10  print(n = 20)

## # A tibble: 523 x 5
##   Pref     Line          N ScoreN_Sum Score_Mean
##   <chr>    <chr>     <int>      <dbl>      <dbl>
## 1 埼玉県   東武東上線     122       27       3.68
## 2 東京都    JR          104       231       3.56
## 3 神奈川県 小田急小田原線    96       31       3.59
## 4 埼玉県   東武伊勢崎線    96       18       3.51
## 5 神奈川県 横浜市営ブルーライン    82       77       3.66
## 6 千葉県    京成本線     82       29       3.34
## 7 神奈川県 京急本線      68       40       3.33

```

```

## 8 千葉県 東武野田線          63      2      4.75
## 9 神奈川県 小田急江ノ島線    62      8      3.79
## 10 大阪府 阪急京都本線      53      32     3.67
## 11 大阪府 南海本線          52      11     4.22
## 12 兵庫県 阪神本線          52      23     3.80
## 13 埼玉県 JR 高崎線          51      5      4
## 14 兵庫県 山陽電鉄本線      51      15     2.98
## 15 千葉県 JR 総武本線 (東京-銚子) 47      8      4
## 16 埼玉県 西武新宿線          45      8      4.17
## 17 埼玉県 秩父鉄道線          43      10     3.82
## 18 大阪府 京阪本線          43      10     3.69
## 19 千葉県 新京成電鉄          43      6      3.6
## 20 京都府 阪急京都本線      43      27     3.5
## # ... with 503 more rows

```

ぐるなびに登録されているラーメン屋が最も多い路線は埼玉県内の東武東上線で 122 店舗があります。東武東上線は東京都と埼玉県をまたがる路線ですので、東武東上線だけならもっと多いかも知れませんね。

ここで一つ考えたいのは `summarise()` 内の `.groups` 引数です。前回はグループ化に使った変数ごとに 1 行しか残っていなかったのでグループ化を全て解除しました。しかし、今回は状況がやや異なります。グループ化変数を使った `Pref` を考えると、まだ `Pref == "東京都"` であるケースがいくつかあります。やろうとすればまだグループ化出来る状態です。これは `Line` についても同じです。`Line == "東武東上線"` の行はここには表示されていないものの、まだデータに残っています。もし、これ以上グループ化しないなら今のように `.groups = "drop"` が正しいですが、もしもう一回グループ化したい場合はどうすればよいでしょうか。方法は 2 つ考えられます。

- もう一度パイプ演算子を使って `group_by()` 関数を使う (以下の 9 行目)。
  - 結果を見ると `## # Groups: Pref, Line [523]` で、ちゃんとグループ化されていることが分かります。

```

1 df %>%
2   filter(!is.na(Line)) %>%

```

```

3   group_by(Pref, Line) %>%
4     summarise(N           = n(),
5               ScoreN_Sum = sum(ScoreN, na.rm = TRUE),
6               Score_Mean = mean(Score, na.rm = TRUE),
7               .groups     = "drop") %>%
8   arrange(desc(N), desc(Score_Mean)) %>%
9   group_by(Pref, Line) %>% # group_by()、もう一度
10  print(n = 5)

## # A tibble: 523 x 5
## # Groups:   Pref, Line [523]
##   Pref     Line           N ScoreN_Sum Score_Mean
##   <chr>   <chr>       <int>      <dbl>      <dbl>
## 1 埼玉県   東武東上線     122        27       3.68
## 2 東京都    JR            104        231       3.56
## 3 神奈川県 小田急小田原線    96        31       3.59
## 4 埼玉県   東武伊勢崎線     96        18       3.51
## 5 神奈川県 横浜市営ブルーライン    82        77       3.66
## # ... with 518 more rows

```

2. .groups 引数を何とかする。

推奨される方法は2番です。具体的には.groups = "keep"を指定するだけであり、こっちの方が無駄なコードを省くことができます。

```

1 df %>%
2   filter(!is.na(Line)) %>%
3   group_by(Pref, Line) %>%
4     summarise(N           = n(),
5               ScoreN_Sum = sum(ScoreN, na.rm = TRUE),
6               Score_Mean = mean(Score, na.rm = TRUE),
7               .groups     = "keep") %>%
8   arrange(desc(N), desc(Score_Mean)) %>%
9   print(n = 5)

```

```
## # A tibble: 523 x 5
## # Groups:   Pref, Line [523]
##   Pref     Line           N  ScoreN_Sum Score_Mean
##   <chr>    <chr>       <int>     <dbl>      <dbl>
## 1 埼玉県   東武東上線     122      27       3.68
## 2 東京都    JR            104      231       3.56
## 3 神奈川県 小田急小田原線  96       31       3.59
## 4 埼玉県   東武伊勢崎線  96       18       3.51
## 5 神奈川県 横浜市営ブルーライン 82       77       3.66
## # ... with 518 more rows
```

.groups 引数は"drop"と"keep"以外にも"drop\_last"があります。実は summarise() に.groups 引数を指定したい場合のデフォルト値は.groups == "drop\_last" または"keep"ですが、ここがややこしいです。主なケースにおいてデフォルト値は"drop"となりますとなります。.groups == "drop\_last" これは最後のグループ化変数のみ解除する意味です。今回の例だと、2番目のグループ化変数である Line がグループ化変数から外され、Pref のみがグループ化変数として残る仕組みです。

それではデフォルト値が"keep"になるのはいつでしょうか。それは記述統計量の結果が長さ 2 以上のベクトルである場合です。平均値を求める mean()、標準偏差を求める sd() などは、結果として長さ 1 のベクトルを返します。しかし、長さ 2 以上のベクトルを返す関数もあります。たとえば、分位数を求める quantile() 関数があります。quantile(ベクトル名, 0.25) の場合、第一四分位点のみ返すため、結果は長さ 1 のベクトルです。しかし、quantile(ベクトル名, c(0.25, 0.5, 0.75)) のように第一四分位点から第三四分位点を同時に計算し、長さ 3 のベクトルが返されるケースもありますし、第二引数を省略すると、最小値・第一四分位点・第二四分位点・第三四分位点・最大値、つまり、長さ 5 のベクトルが返される場合があります。

```
1 # 第一四分位点のみを求める (長さ 1 のベクトル)
2 quantile(df$Walk, 0.25, na.rm = TRUE)
```

```
## 25%
```

```
## 2

1 # 引数を省略する (長さ 5 のベクトル)
2 quantile(df$Walk, na.rm = TRUE)

## 0% 25% 50% 75% 100%
## 1 2 5 12 116
```

.groups のデフォルト値が"keep"になるのは、このように長さ 2 以上のベクトルが返されるケースです。たとえば、都府県と最寄りの駅の路線でグループ化し、店舗までの歩き距離の平均値を求めるとしています。デフォルト値の変化を見るために、ここではあって.groups 引数を省略しました。

```
1 df %>%
2   filter(!is.na(Walk)) %>%
3   group_by(Pref, Line) %>%
4   summarise(Mean = mean(Walk))

## `summarise()` has grouped output by 'Pref'. You can override using the
## `.` argument.

## # A tibble: 509 x 3
## # Groups:   Pref [9]
##   Pref   Line      Mean
##   <chr>  <chr>    <dbl>
## 1 京都府  JR        4
## 2 京都府  JR 京都線    10
## 3 京都府  JR 奈良線    3.33
## 4 京都府  JR 奈良線     8
## 5 京都府  JR 小浜線    16.5
## 6 京都府  JR 小浜線     9
## 7 京都府  JR 山陰本線    19
## 8 京都府  JR 山陰本線 (京都-米子)  8.67
## 9 京都府  JR 山陰本線 (京都-米子)  9.23
## 10 京都府  JR 嵐山線      5
```

```
## # ... with 499 more rows
```

最初は Pref と Line でグループ化しましたが、`summarise()` の後、Line がグループ化変数から外されました。つまり、引数が"drop\_last"になっていることです。

それでは、平均値に加えて、第一四分位点と第三四分位点も計算し、`Quantile` という名で格納してみましょう。

```
1 df %>%
2   filter(!is.na(Walk)) %>%
3   group_by(Pref, Line) %>%
4   summarise(Mean      = mean(Walk),
5             Quantile = quantile(Walk, c(0.25, 0.75)))
```

```
## `summarise()` has grouped output by 'Pref', 'Line'. You can override using the
## `.`.groups` argument.
```

```
## # A tibble: 1,018 x 4
## # Groups:   Pref, Line [509]
##   Pref   Line       Mean Quantile
##   <chr> <chr>     <dbl>    <dbl>
## 1 京都府 JR        4        1.25
## 2 京都府 JR        4        5
## 3 京都府 JR 京都線 10       10
## 4 京都府 JR 京都線 10       10
## 5 京都府 JR 奈良線  3.33     2
## 6 京都府 JR 奈良線  3.33     5
## 7 京都府 JR 奈良線  8        4
## 8 京都府 JR 奈良線  8        9
## 9 京都府 JR 小浜線 16.5     9.75
## 10 京都府 JR 小浜線 16.5    23.2
## # ... with 1,008 more rows
```

同じ Pref、Line のケースが 2 つずつ出来ています。最初に来る数値は第一四分位点、次に来るのが第三四分位点です。そして最初のグループ化変数であった Pref と Line が、

`summarise()` 後もグループ化変数として残っていることが分かります。

`.groups` 引数は記述統計量だけを計算する意味ではありません。しかし、得られた記述統計量から何らかの計算をしたり、さらにもう一回記述統計量を求めたりする際、予期せぬ結果が得られる可能性があるため注意する必要があります。出来る限り`.groups` 引数は指定するようにしましょう。

---

## 13.3 変数の計算

### 13.3.1 `mutate()` 関数の使い方

続いて、データフレーム内の変数を用いて計算を行い、その結果を新しい列として格納する`mutate()` 関数について紹介します。`mutate()` 関数は新しい列を追加することが主な目的ですが、既存の列を修正することも可能です。まず、`mutate()` 関数の書き方からです。

```
1 # mutate() 関数の使い方
2 データフレーム名 %>%
3   mutate(新しい変数名 = 処理内容)
```

これは何らかの処理を行い、その結果を新しい変数としてデータフレームに追加することを意味します。新しく出来た変数は、基本的に最後の列になります。ここでは分単位である`Walk` を時間単位に変換した`Walk_Hour` 変数を作成するとします。処理内容は`Walk / 60` です。最後に、都府県名、店舗名、歩行距離(分)、歩行距離(時間)のみを残し、遠い順にソートします。

```
1 df %>%
2   filter(!is.na(Walk)) %>%
3   mutate(Walk_Hour = Walk / 60) %>%
4   select(Pref, Name, Walk, Walk_Hour) %>%
5   arrange(desc(Walk_Hour))
```

```
## # A tibble: 5,375 x 4
```

```

##   Pref      Name          Walk Walk_Hour
##   <chr>    <chr>        <dbl>    <dbl>
## 1 埼玉県  札幌ラーメン どさん子 小鹿野店    116     1.93
## 2 神奈川県 札幌ラーメン どさん子 中津店      73     1.22
## 3 千葉県  札幌ラーメン どさん子 佐原 51 号店    59     0.983
## 4 神奈川県 札幌ラーメン どさん子 山際店      50     0.833
## 5 千葉県  札幌ラーメン どさん子 関宿店      49     0.817
## 6 兵庫県  濃厚醤油 中華そば いせや 玉津店    43     0.717
## 7 大阪府  河童ラーメン本舗 岸和田店      38     0.633
## 8 埼玉県  ラーメン山岡家 上尾店      35     0.583
## 9 兵庫県  濃厚醤油 中華そば いせや 大蔵谷店    35     0.583
## 10 大阪府 河童ラーメン本舗 松原店     31     0.517
## # ... with 5,365 more rows

```

`mutate()` は 3 行目に登場しますが、これは `Walk` を 60 に割った結果を `Walk_Hour` としてデータフレームの最後の列として格納することを意味します。もし、最後の列でなく、ある変数の前、または後にしたい場合は、`.before` または`.after`引数を追加します。これは `select()` 関数の`.before` と`.after` と同じ使い方です。たとえば、新しく出来た `Walk_Hour` を `ID` と `Name` の間に入れたい場合は

```

1 # コードの 3 行名を修正 (.before 使用)
2 mutate(Walk_Hour = Walk / 60,
3        .before    = Name)
4
5 # コードの 3 行名を修正 (.after 使用)
6 mutate(Walk_Hour = Walk / 60,
7        .after     = ID)

```

のようにコードを修正します。

むろん、変数間同士の計算も可能です。たとえば、以下のような `df2` があり、1 店舗当たりの平均口コミ数を計算し、`ScoreN_Mean` という変数名で `ScoreN_Sum` の後に格納うするとしています。この場合、`ScoreN_Sum` 変数を `N` で割るだけです。

```

1 df2 <- df %>%
2   group_by(Pref) %>%
3   summarise(Budget_Mean = mean(Budget, na.rm = TRUE),
4             ScoreN_Sum = sum(ScoreN, na.rm = TRUE),
5             Score_Mean = mean(Score, na.rm = TRUE),
6             N = n(),
7             .groups = "drop")

1 df2 %>%
2   mutate(ScoreN_Mean = ScoreN_Sum / N,
3         .after = ScoreN_Sum)

## # A tibble: 9 x 6
##   Pref    Budget_Mean  ScoreN_Sum ScoreN_Mean Score_Mean    N
##   <chr>      <dbl>       <dbl>      <dbl>      <dbl> <int>
## 1 京都府     1399.       216       0.522      3.68   414
## 2 兵庫県     1197.       230       0.389      3.54   591
## 3 千葉県     1124.       259       0.259      3.72  1000
## 4 和歌山県    1252        83       0.593      3.97   140
## 5 埼玉県     1147.       278       0.278      3.64  1000
## 6 大阪府     1203.       516       0.516      3.77  1000
## 7 奈良県     1169.       45        0.306      3.85   147
## 8 東京都     1283.      1165      1.16       3.67  1000
## 9 神奈川県    1239.      587       0.587      3.53  1000

```

このように、データ内の変数を用いた計算結果を新しい列として追加する場合は、`mutate()` が便利です。これを `mutate()` を使わずに処理する場合、以下のようなコードになりますが、可読性が相対的に低いことが分かります。

```

1 df2$ScoreN_Mean <- df2$ScoreN_Sum / df2$N
2 df2 <- df2[, c("Pref", "Budget_Mean", "Walk_Mean",
3               "ScoreN_Sum", "ScoreN_Mean", "Score_Mean", "N")]

```

もちろんですが、計算には `+` や `/` のような演算子だけでなく、関数を使うことも可能です。

たとえば、Budget が 1000 円未満なら "Cheap"、1000 円以上なら "Expensive" と示す変数 Budget2 を作成する場合は if\_else() 関数 (または、ifelse() 関数) が使えます。

```
1 df %>%
2   mutate(Budget2 = if_else(Budget < 1000, "Cheap", "Expensive")) %>%
3   filter(!is.na(Budget2)) %>% # Budget2 が欠損した店舗を除外
4   group_by(Pref, Budget2) %>% # Pref と Budget2 でグループ化
5   summarise(N = n(),           # 店舗数を表示
6             .groups = "drop")
```

```
## # A tibble: 18 x 3
##   Pref   Budget2     N
##   <chr>  <chr>   <int>
## 1 京都府  Cheap     22
## 2 京都府  Expensive 28
## 3 兵庫県  Cheap     39
## 4 兵庫県  Expensive 27
## 5 千葉県  Cheap     64
## 6 千葉県  Expensive 72
## 7 和歌山県 Cheap     10
## 8 和歌山県 Expensive  5
## 9 埼玉県  Cheap     37
## 10 埼玉県 Expensive 45
## 11 大阪府 Cheap     104
## 12 大阪府 Expensive 115
## 13 奈良県 Cheap     11
## 14 奈良県 Expensive 10
## 15 東京都 Cheap     206
## 16 東京都 Expensive 236
## 17 神奈川県 Cheap     66
## 18 神奈川県 Expensive 54
```

これは各都府県ごとの予算 1000 円未満の店と以上の店の店舗数をまとめた表となります。もし、500 円未満なら "Cheap"、500 円以上~1000 円未満なら "Reasonable"、1000

円以上なら"Expensive"になる Budget3 変数を作るにはどうすればよいでしょうか。第10章で紹介しました `if_else()` を重ねることも出来ますが、ここでは `case_when()` 関数が便利です。まずは、`if_else()` を使ったコードは以下の通りです。

```

1 # if_else() を使う場合
2 df %>%
3   mutate(Budget3 = if_else(Budget < 500, "Cheap",
4                           if_else(Budget >= 500 & Budget < 1000, "Reasonable",
5                               "Expensive")))) %>%
6   filter(!is.na(Budget3)) %>%
7   group_by(Pref, Budget3) %>%
8   summarise(N = n(),
9             .groups = "drop")

```

`case_when()` を使うと以下のような書き方になります。

```

1 # case_when() を使う場合
2 df %>%
3   mutate(Budget3 = case_when(Budget < 500
4                             ~ "Cheap",
5                             Budget >= 500 & Budget < 1000 ~ "Reasonable",
6                             Budget >= 1000 ~ "Expensive"),
7     # 新しく出来た変数を factor 型にその場で変換することも可能
8     Budget3 = factor(Budget3,
9                        levels = c("Cheap", "Reasonable", "Expensive))) %>%
10    filter(!is.na(Budget3)) %>%
11    group_by(Pref, Budget3) %>%
12    summarise(N = n(),
13              .groups = "drop")

## # A tibble: 20 x 3
##   Pref     Budget3      N
##   <chr>    <fct>     <int>
## 1 京都府   Reasonable    22
## 2 京都府   Expensive     28

```

```
## 3 兵庫県 Reasonable 39
## 4 兵庫県 Expensive 27
## 5 千葉県 Reasonable 64
## 6 千葉県 Expensive 72
## 7 和歌山県 Reasonable 10
## 8 和歌山県 Expensive 5
## 9 埼玉県 Reasonable 37
## 10 埼玉県 Expensive 45
## 11 大阪府 Reasonable 104
## 12 大阪府 Expensive 115
## 13 奈良県 Reasonable 11
## 14 奈良県 Expensive 10
## 15 東京都 Cheap 1
## 16 東京都 Reasonable 205
## 17 東京都 Expensive 236
## 18 神奈川県 Cheap 1
## 19 神奈川県 Reasonable 65
## 20 神奈川県 Expensive 54
```

書く手間の観点では `case_when()` は `ifelse()` と大きく違いはないかも知れませんが、コードが非常に読みやすくなっています。`case_when()` 関数の書き方は以下の通りです。

```
1 # case_when() の使い方
2 データフレーム名 %>%
3   mutate(新変数名 = case_when(条件 1 ~ 条件 1 を満たす場合の結果値,
4                               条件 2 ~ 条件 2 を満たす場合の結果値,
5                               条件 3 ~ 条件 3 を満たす場合の結果値,
6                               ...,
7                               TRUE ~ その他の場合の結果値))
```

`case_when()` 内部での判定は先に指定された条件から順番に行われます。条件 1 に満たされない場合は、条件 2 で判定し、その条件 2 に満たされない場合は条件 3、...、そして全て満たされなかった場合は `TRUE` ~ 以降の値を返すといった感覚です。したがって、先ほどのコードにおける `Budget >= 500 & Budget < 1000` は効率的ではありません。

簡略化すると以下のようになります。

```
1 df %>%
2   mutate(Budget3 = case_when(Budget < 500 ~ "Cheap",
3                               Budget < 1000 ~ "Reasonable",
4                               Budget >= 1000 ~ "Expensive"),
5   Budget3 = factor(Budget3,
6                     levels = c("Cheap", "Reasonable", "Expensive")))) %>%
7   filter(!is.na(Budget3)) %>%
8   group_by(Pref, Budget3) %>%
9   summarise(N = n(),
10             .groups = "drop")
```

```
## # A tibble: 20 x 3
##   Pref    Budget3     N
##   <chr>  <fct>    <int>
## 1 京都府 Reasonable    22
## 2 京都府 Expensive     28
## 3 兵庫県 Reasonable    39
## 4 兵庫県 Expensive     27
## 5 千葉県 Reasonable    64
## 6 千葉県 Expensive     72
## 7 和歌山県 Reasonable    10
## 8 和歌山県 Expensive      5
## 9 埼玉県 Reasonable    37
## 10 埼玉県 Expensive     45
## 11 大阪府 Reasonable   104
## 12 大阪府 Expensive    115
## 13 奈良県 Reasonable     11
## 14 奈良県 Expensive      10
## 15 東京都 Cheap            1
## 16 東京都 Reasonable   205
## 17 東京都 Expensive    236
```

```
## 18 神奈川県 Cheap 1
## 19 神奈川県 Reasonable 65
## 20 神奈川県 Expensive 54
```

ただし、この TRUE には注意が必要です。たとえば、以下のようなコードを考えてみましょう。

```
1 df %>%
2   mutate(Budget3 = case_when(Budget < 500 ~ "Cheap",
3                               Budget < 1000 ~ "Reasonable",
4                               TRUE ~ "Expensive"),
5         Budget3 = factor(Budget3,
6                           levels = c("Cheap", "Reasonable", "Expensive")) %>%
7         filter(!is.na(Budget3)) %>%
8         group_by(Pref, Budget3) %>%
9         summarise(N = n(),
10           .groups = "drop")
```

```
## # A tibble: 20 x 3
##   Pref   Budget3     N
##   <chr> <fct>     <int>
## 1 京都府 Reasonable    22
## 2 京都府 Expensive    392
## 3 兵庫県 Reasonable    39
## 4 兵庫県 Expensive    552
## 5 千葉県 Reasonable    64
## 6 千葉県 Expensive    936
## 7 和歌山県 Reasonable    10
## 8 和歌山県 Expensive    130
## 9 埼玉県 Reasonable     37
## 10 埼玉県 Expensive    963
## 11 大阪府 Reasonable    104
## 12 大阪府 Expensive    896
## 13 奈良県 Reasonable     11
```

```

## 14 奈良県  Expensive    136
## 15 東京都    Cheap        1
## 16 東京都    Reasonable  205
## 17 東京都    Expensive   794
## 18 神奈川県 Cheap        1
## 19 神奈川県 Reasonable  65
## 20 神奈川県 Expensive   934

```

結果が異なりますね。なぜなら `Budget` の値が欠損値であっても "Expensive" と判定されるからです。対処方法としましては `mutate()` の前に `filter()` を使って `Budget3` が欠損した行を削除するか、`Budget` が欠損した場合の値を `case_when()` 内部に指定するかです。以下の例は `Budget` が欠損した場合の処理内容を加えたものです。

```

1 df %>%
2   mutate(Budget3 = case_when(is.na(Budget) ~ NA_character_,
3                               Budget < 500 ~ "Cheap",
4                               Budget < 1000 ~ "Reasonable",
5                               TRUE ~ "Expensive"),
6   Budget3 = factor(Budget3,
7                     levels = c("Cheap", "Reasonable", "Expensive")) %>%
8   filter(!is.na(Budget3)) %>%
9   group_by(Pref, Budget3) %>%
10  summarise(N = n(),
11            .groups = "drop")

```

```

## # A tibble: 20 x 3
##       Pref   Budget3     N
##       <chr>   <fct>   <int>
## 1 京都府 Reasonable    22
## 2 京都府 Expensive     28
## 3 兵庫県 Reasonable    39
## 4 兵庫県 Expensive     27
## 5 千葉県 Reasonable    64
## 6 千葉県 Expensive     72

```

```
## 7 和歌山県 Reasonable 10
## 8 和歌山県 Expensive 5
## 9 埼玉県 Reasonable 37
## 10 埼玉県 Expensive 45
## 11 大阪府 Reasonable 104
## 12 大阪府 Expensive 115
## 13 奈良県 Reasonable 11
## 14 奈良県 Expensive 10
## 15 東京都 Cheap 1
## 16 東京都 Reasonable 205
## 17 東京都 Expensive 236
## 18 神奈川県 Cheap 1
## 19 神奈川県 Reasonable 65
## 20 神奈川県 Expensive 54
```

Budget3 は character 型変数となるので、欠損値のタイプも NA\_character\_ にします。もし、numeric 型変数を作成するなら NA\_real\_ を使います。if\_else() や case\_when() など、{dplyr}が提供する関数は欠損値の型にも厳しいので、NAだけだとエラーが帰ってきます。

これまでの case\_when() 関数に似たような機能をする関数として recode() 関数があります<sup>2)</sup>。これは変数の値を単純に置換したい場合に便利な関数です。たとえば、都府県名をローマ字に変換するケースを考えてみましょう。

```
1 # recode() を使う場合
2 df2 %>%
3   mutate(Pref2 = recode(Pref,
4                         "東京都"    = "Tokyo",
5                         "神奈川県"  = "Kanagawa",
6                         "千葉県"    = "Chiba",
7                         "埼玉県"    = "Saitama",
```

<sup>2)</sup> recode() 関数は他にも{car}パッケージでも提供されています。{car}も広く使われている依存パッケージの一つであるため、dplyr::recode() と明示した方が良いかも知れません。

```

8      "大阪府" = "Osaka",
9      "京都府" = "Kyoto",
10     "兵庫県" = "Hyogo",
11     "奈良県" = "Nara",
12     "和歌山県" = "Wakayama",
13     .default = "NA"))

```

```

## # A tibble: 9 x 6
##   Pref      Budget_Mean ScoreN_Sum Score_Mean     N Pref2
##   <chr>        <dbl>      <dbl>      <dbl> <int> <chr>
## 1 京都府      1399.       216       3.68   414 Kyoto
## 2 兵庫県      1197.       230       3.54   591 Hyogo
## 3 千葉県      1124.       259       3.72  1000 Chiba
## 4 和歌山県     1252        83        3.97   140 Wakayama
## 5 埼玉県      1147.       278       3.64  1000 Saitama
## 6 大阪府      1203.       516       3.77  1000 Osaka
## 7 奈良県      1169.        45        3.85   147 Nara
## 8 東京都      1283.      1165       3.67  1000 Tokyo
## 9 神奈川県     1239.       587       3.53  1000 Kanagawa

```

使い方は非常に直感的です。

```

1 # recode() の使い方
2 データフレーム名 %>%
3   mutate(新変数名 = recode(元の変数名,
4                           元の値1 = 新しい値1,
5                           元の値2 = 新しい値2,
6                           元の値3 = 新しい値3,
7                           ...,
8                           .default = 該当しない場合の値))

```

最後の.default 引数は、もし該当する値がない場合に返す値を意味し、長さ 1 のベクトルを指定します。もし、指定しない場合は NA が表示されます。また、ここには紹介してお

りませんでしたが、`.missing`引数もあり、これは欠損値の場合に返す値を意味します。

もう一つ注意すべきところは、今回は character 型変数を character 型へ変換したため、「"東京都" = "Tokyo"」のような書き方をしました。しかし、numeric 型から character 型に変換する場合は数字の部分を`で囲む必要があります。たとえば、「`1` = "Tokyo"」といった形式です。ただし、character 型から numeric 型への場合は「"東京都" = 1」で構いません。

`recode()`は値をまとめる際にも便利です。たとえば、`EastJapan`という変数を作成し、関東なら 1 を、それ以外なら 0 を付けるとします。そして、これは `Pref` 変数の後に位置づけます。

```
1 # 都府県を関東か否かでまとめる
2 df2 %>%
3   mutate(EastJapan = recode(Pref,
4                             "東京都" = 1,
5                             "神奈川県" = 1,
6                             "千葉県" = 1,
7                             "埼玉県" = 1,
8                             "大阪府" = 0,
9                             "京都府" = 0,
10                            "兵庫県" = 0,
11                            "奈良県" = 0,
12                            "和歌山県" = 0,
13                            .default = 0),
14   .after = Pref)

## # A tibble: 9 x 6
##   Pref      EastJapan Budget_Mean ScoreN_Sum Score_Mean     N
##   <chr>      <dbl>      <dbl>      <dbl>      <dbl> <int>
## 1 京都府        0      1399.      216      3.68    414
## 2 兵庫県        0      1197.      230      3.54    591
## 3 千葉県        1      1124.      259      3.72   1000
## 4 和歌山県      0      1252       83      3.97    140
```

```

## 5 埼玉県      1    1147.      278    3.64  1000
## 6 大阪府      0    1203.      516    3.77  1000
## 7 奈良県      0    1169.      45     3.85  147
## 8 東京都      1    1283.      1165   3.67  1000
## 9 神奈川県    1    1239.      587    3.53  1000

```

ただし、関東以外は全て0になるため、以下のように省略することも可能です。

```

1 # .default 引数を指定する場合
2 df3 <- df2 %>%
3   mutate(EastJapan = recode(Pref,
4                             "東京都"    = 1,
5                             "神奈川県"  = 1,
6                             "千葉県"    = 1,
7                             "埼玉県"    = 1,
8                             .default    = 0),
9   .after = Pref)
10
11 df3

```

```

## # A tibble: 9 x 6
##   Pref    EastJapan Budget_Mean ScoreN_Sum Score_Mean     N
##   <chr>   <dbl>      <dbl>      <dbl>      <dbl> <int>
## 1 京都府      0        1399.      216     3.68   414
## 2 兵庫県      0        1197.      230     3.54   591
## 3 千葉県      1        1124.      259     3.72   1000
## 4 和歌山県    0        1252       83      3.97   140
## 5 埼玉県      1        1147.      278     3.64   1000
## 6 大阪府      0        1203.      516     3.77   1000
## 7 奈良県      0        1169.      45      3.85   147
## 8 東京都      1        1283.      1165   3.67   1000
## 9 神奈川県    1        1239.      587     3.53   1000

```

新しく出来た EastJapan のデータ型はなんでしょうか。

```
1  class(df3$EastJapan)
```

```
## [1] "numeric"
```

EastJapan は numeric 型ですね。もし、これを factor 型にしたい場合はどうすればよいでしょうか。それは `mutate()` 内で EastJapan を生成した後に `factor()` 関数を使うだけです。

```
1  # EastJapan 変数を factor 型にする
2  df3 <- df2 %>%
3    mutate(EastJapan = recode(Pref,
4      "東京都" = 1,
5      "神奈川県" = 1,
6      "千葉県" = 1,
7      "埼玉県" = 1,
8      .default = 0),
9      EastJapan = factor(EastJapan, levels = c(0, 1)),
10     .after = Pref)
11
12 df3$EastJapan
```

```
## [1] 0 0 1 0 1 0 0 1 1
```

```
## Levels: 0 1
```

EastJapan が factor 型になりました。実は、`recode` は再コーディングと同時に factor 化をしてくれる機能があります。ただし、`recode()` 関数でなく、`recode_factor()` 関数を使います。

```
1  # recode_factor() を使う方法
2  df3 <- df2 %>%
3    mutate(EastJapan = recode_factor(Pref,
4      "東京都" = 1,
5      "神奈川県" = 1,
6      "千葉県" = 1,
```

```
7         "埼玉県" = 1,  
8         .default = 0),  
9         .after = Pref)  
10  
11 df3$EastJapan
```

```
## [1] 0 0 1 0 1 0 0 1 1  
## Levels: 1 0
```

ただし、level の順番は `recode_factor()` 内で定義された順番になることに注意してください。factor 型のより詳細な扱いについては第14章で解説します。

---

## 13.4 行単位の操作

ここでは行単位の操作について考えたいと思います。第12.3章で使った `myDF1` を見てみましょう。

```
1 myDF1 <- data.frame(  
2   ID = 1:5,  
3   X1 = c(2, 4, 6, 2, 7),  
4   Y1 = c(3, 5, 1, 1, 0),  
5   X1D = c(4, 2, 1, 6, 9),  
6   X2 = c(5, 5, 6, 0, 2),  
7   Y2 = c(3, 3, 2, 3, 1),  
8   X2D = c(8, 9, 5, 0, 1),  
9   X3 = c(3, 0, 3, 0, 2),  
10  Y3 = c(1, 5, 9, 1, 3),  
11  X3D = c(9, 1, 3, 3, 8)  
12 )  
13  
14 myDF1
```

```
##   ID X1 Y1 X1D X2 Y2 X2D X3 Y3 X3D
## 1  1  2  3  4  5  3  8  3  1  9
## 2  2  4  5  2  5  3  9  0  5  1
## 3  3  6  1  1  6  2  5  3  9  3
## 4  4  2  1  6  0  3  0  0  1  3
## 5  5  7  0  9  2  1  1  2  3  8
```

ここで X1 と X2 と X3 の平均値を計算し、X\_Mean という名の変数にする場合、以下のような書き方が普通でしょう。

```
1 myDF1 %>%
2   mutate(X_Mean = mean(c(X1, X2, X3)))
##   ID X1 Y1 X1D X2 Y2 X2D X3 Y3 X3D   X_Mean
## 1  1  2  3  4  5  3  8  3  1  9  3.133333
## 2  2  4  5  2  5  3  9  0  5  1  3.133333
## 3  3  6  1  1  6  2  5  3  9  3  3.133333
## 4  4  2  1  6  0  3  0  0  1  3  3.133333
## 5  5  7  0  9  2  1  1  2  3  8  3.133333
```

あら、なんかおかしくありませんか。1 行目の場合、X1 と X2、X3 それぞれ 2、5、3 であり、平均値は 3.333 であるはずなのに 3.133 になりました。これは 2 行目以降も同じです。なぜでしょうか。

実は{dplyr}は行単位の計算が苦手です。実際、データフレームというのは既に説明したとおり、縦ベクトルを横に並べたものです。列をまたがる場合、データ型が異なる場合も多いため、そもそも使う場面も多くありません。したがって、以下のような書き方が必要でした。

```
1 myDF1 %>%
2   mutate(X_Mean = (X1 + X2 + X3) / 3)
##   ID X1 Y1 X1D X2 Y2 X2D X3 Y3 X3D   X_Mean
## 1  1  2  3  4  5  3  8  3  1  9  3.333333
## 2  2  4  5  2  5  3  9  0  5  1  3.0000000
```

```
## 3 3 6 1 1 6 2 5 3 9 3 5.0000000
## 4 4 2 1 6 0 3 0 0 1 3 0.6666667
## 5 5 7 0 9 2 1 1 2 3 8 3.6666667
```

先ほどの `mean(c(X1, X2, X3))` は (X1 列と X2 列、X3 列) の平均値です。X1 は長さ 1 のベクトルではなく、その列全体を指すものです。つまり、`mean(c(X1, X2, X3))` は `mean(c(myDF1$X1, myDF1$X2, myDF1$X3))` と同じことになります。だから全て 3.133 という結果が得られました。ただし、後者はベクトル同士の加減乗除になるため問題ありません。実際 `c(1, 2, 3) + c(3, 5, 0)` は同じ位置の要素同士の計算になることを既に第9.2章で説明しました。

ここで `mean()` 関数を使う場合には全ての演算を、一行一行に分けて行う必要があります。ある一行のみに限定する場合、`mean(c(X1, X2, X3))` の X1 などは長さ 1 のベクトルになるため、`(X1 + X2 + X3) / 3` と同じことになります。この「一行単位で処理を行う」ことを指定する関数が `rowwise()` 関数です。これは行単位の作業を行う前に指定するだけです。

```
1 myDF1 %>%
2   rowwise() %>%
3   mutate(X_Mean = mean(c(X1, X2, X3)))
```

```
## # A tibble: 5 x 11
## # Rowwise:
##   ID    X1    Y1    X1D    X2    Y2    X2D    X3    Y3    X3D X_Mean
##   <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     2     3     4     5     3     8     3     1     9   3.33
## 2     2     4     5     2     5     3     9     0     5     1   3
## 3     3     6     1     1     6     2     5     3     9     3   5
## 4     4     2     1     6     0     3     0     0     1     3   0.667
## 5     5     7     0     9     2     1     1     2     3     8   3.67
```

これで問題なく行単位の処理ができるようになりました。今回は変数が 3 つのみだったので、これで問題ありませんが、変数が多くなると: や `starts_with()`、`num_range()` などを使って変数を選択したくなります。この場合は計算する関数内に `c_across()` を入れます。ここでは X1 列から X3D 列までの平均値を求めてみましょう。

```
1 myDF1 %>%
2   rowwise() %>%
3   mutate(X_Mean = mean(X1:X3D))
```

```
## # A tibble: 5 x 11
## # Rowwise:
##   ID    X1    Y1    X1D    X2    Y2    X2D    X3    Y3    X3D X_Mean
##   <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     2     3     4     5     3     8     3     1     9     5.5
## 2     2     4     5     2     5     3     9     0     5     1     2.5
## 3     3     6     1     1     6     2     5     3     9     3     4.5
## 4     4     2     1     6     0     3     0     0     1     3     2.5
## 5     5     7     0     9     2     1     1     2     3     8     7.5
```

実は `rowwise()` 関数、2020年6月に公開された `dplyr 1.0.0` で注目された関数ですが、昔の `dplyr` にも `rowwise()` 関数がありました。ただし、`purrr` パッケージや `tidyverse` パッケージの `nest()` 関数などにより使い道がなくなりましたが、なぜか華麗に復活しました。データ分析に使うデータは基本単位は列であるため、実際に `rowwise()` が使われる場面は今の段階では多くないでしょう。また、簡単な作業なら `X1 + X2` のような演算でも対応できます。それでも、覚えておけば便利な関数であることには間違ひありません。

---

## 13.5 データの結合

### 13.5.1 行の結合

まずは、複数のデータフレームまたは `tibble` を縦に結合する方法について解説します。イメージとしては図 13.1 のようなものです。

```
New_Data <- bind_rows(Data1, Data2)
```

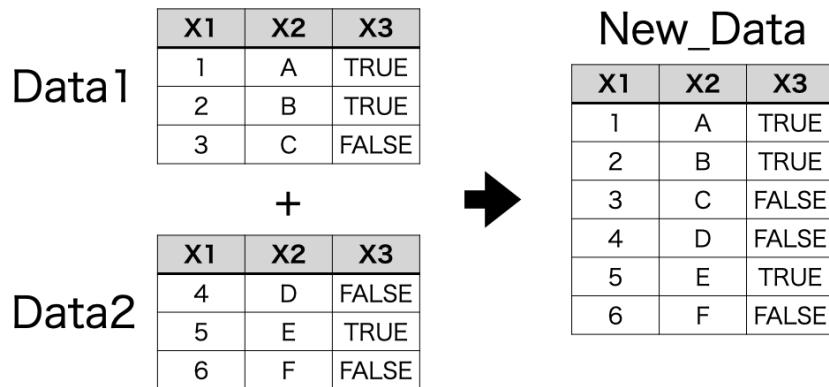


図 13.1: 行の結合

行を結合する際には{dplyr}パッケージの `bind_rows()` 関数を使います。この関数の使い方は以下の通りです。

```
1 # 新しいデータ名ではなく、既にあるデータ名にすると上書きとなる
2 新しいデータ名 <- bind_rows(データ1, データ2, ...)
```

それでは早速実際に使ってみましょう。実習のために、4つの tibble を作成します (tibble でなくデータフレームでも問題ありません)。

```
1 # tibble() の代わりに data.frame() も可
2 rbind_df1 <- tibble(X1 = 1:3,
3                      X2 = c("A", "B", "C"),
4                      X3 = c(T, T, F)) # TRUE と FALSE は T は F と省略可能
5
6 rbind_df2 <- tibble(X1 = 4:6,
7                      X2 = c("D", "E", "F"),
8                      X3 = c(F, T, F))
9
10 rbind_df3 <- tibble(X1 = 7:9,
11                      X3 = c(T, T, T),
12                      X2 = c("G", "H", "I"))
```

```
13
14 rbind_df4 <- tibble(X1 = 10:12,
15                         X2 = c("J", "K", "L"),
16                         X5 = c("Song", "Yanai", "Hadley"))
17
18 rbind_df1 # rbind_df1 を出力

## # A tibble: 3 x 3
##       X1     X2     X3
##   <int> <chr> <lgl>
## 1     1     A     TRUE
## 2     2     B     TRUE
## 3     3     C    FALSE

1 rbind_df2 # rbind_df2 を出力

## # A tibble: 3 x 3
##       X1     X2     X3
##   <int> <chr> <lgl>
## 1     4     D    FALSE
## 2     5     E     TRUE
## 3     6     F    FALSE

1 rbind_df3 # rbind_df3 を出力

## # A tibble: 3 x 3
##       X1     X3     X2
##   <int> <lgl> <chr>
## 1     7  TRUE    G
## 2     8  TRUE    H
## 3     9  TRUE    I

1 rbind_df4 # rbind_df4 を出力

## # A tibble: 3 x 3
```

```
##      X1 X2     X5
##  <int> <chr> <chr>
## 1     10 J      Song
## 2     11 K      Yanai
## 3     12 L      Hadley
```

まずは、`rbind_df1` と `rbind_df2` を結合してみます。この2つのデータは同じ変数が同じ順番で並んでいますね。

```
1 Binded_df1 <- bind_rows(rbind_df1, rbind_df2)
2 Binded_df1
```

```
## # A tibble: 6 x 3
##      X1 X2     X3
##  <int> <chr> <lgl>
## 1     1 A      TRUE
## 2     2 B      TRUE
## 3     3 C     FALSE
## 4     4 D     FALSE
## 5     5 E      TRUE
## 6     6 F     FALSE
```

2つのデータが結合されたことが確認できます。それでは `rbind_df1` と `rbind_df2`、`rbind_df3` はどうでしょうか。確かに3つのデータは同じ変数を持ちますが、`rbind_df3` は変数の順番が X1、X3、X2 になっています。このまま結合するとエラーが出るでしょうか。とりあえず、やってみます。

```
1 Binded_df2 <- bind_rows(rbind_df1, rbind_df2, rbind_df3)
2 Binded_df2
```

```
## # A tibble: 9 x 3
##      X1 X2     X3
##  <int> <chr> <lgl>
## 1     1 A      TRUE
## 2     2 B     FALSE
## 3     3 C     FALSE
## 4     4 D      TRUE
## 5     5 E     FALSE
## 6     6 F      TRUE
## 7     7 G      TRUE
## 8     8 H     FALSE
## 9     9 I     FALSE
```

```
## 3     3 C     FALSE
## 4     4 D     FALSE
## 5     5 E      TRUE
## 6     6 F     FALSE
## 7     7 G      TRUE
## 8     8 H      TRUE
## 9     9 I      TRUE
```

このように変数の順番が異なっても、先に指定したデータの変数順で問題なく結合できました。これまでの作業は{dplyr}パッケージの `bind_rows()` を使わずに、R 内蔵関数の `rbind()` でも同じやり方でできます。`bind_rows()` の特徴は、変数名が一致しない場合、つまり今回の例だと `rbind_df4` が含まれる場合です。`rbind_df1` から `rbind_df3` までは順番が違っても `X1`、`X2`、`X3` 変数で構成されていました。一方、`rbind_df4` には `X3` がなく、新たに `X4` という変数があります。これを `rbind()` 関数で結合するとエラーが出力されます。

```
1 # rbind() を使う場合
2 rbind(rbind_df1, rbind_df2, rbind_df3, rbind_df4)
```

```
## Error in match.names(clabs, names(xi)): names do not match previous names
```

一方、`bind_rows()` はどうでしょうか。

```
1 Binded_df3 <- bind_rows(rbind_df1, rbind_df2, rbind_df3, rbind_df4)
2 Binded_df3
```

```
## # A tibble: 12 x 4
##       X1 X2     X3     X5
##   <int> <chr> <lgl> <chr>
## 1     1 A     TRUE  <NA>
## 2     2 B     TRUE  <NA>
## 3     3 C    FALSE <NA>
## 4     4 D    FALSE <NA>
## 5     5 E     TRUE  <NA>
## 6     6 F    FALSE <NA>
```

```
## 7   G   TRUE  <NA>
## 8   H   TRUE  <NA>
## 9   I   TRUE  <NA>
## 10  J   NA    Song
## 11  K   NA    Yanai
## 12  L   NA    Hadley
```

X1 から X4 まで全ての列が生成され、元のデータにはなかった列に関しては NA で埋められています。

ならば、bind\_rows() の完全勝利かというと、そうとは限りません。自分で架空した複数のデータフレーム、または tibble を結合する際、「このデータは全て同じ変数を持っているはず」と事前に分かっているなら rbind() の方が効果的です。なぜなら、変数名が異なる場合、エラーが出力されるからです。bind\_rows() を使うと、コーディングミスなどにより、列名の相違がある場合でも結合してくれてしまうので、分析の結果を歪ませる可能性があります。

### 13.5.2 列の結合

実はデータ分析においてデータの結合といえば、列の結合が一般的です。これは図 13.2 のような操作を意味します。

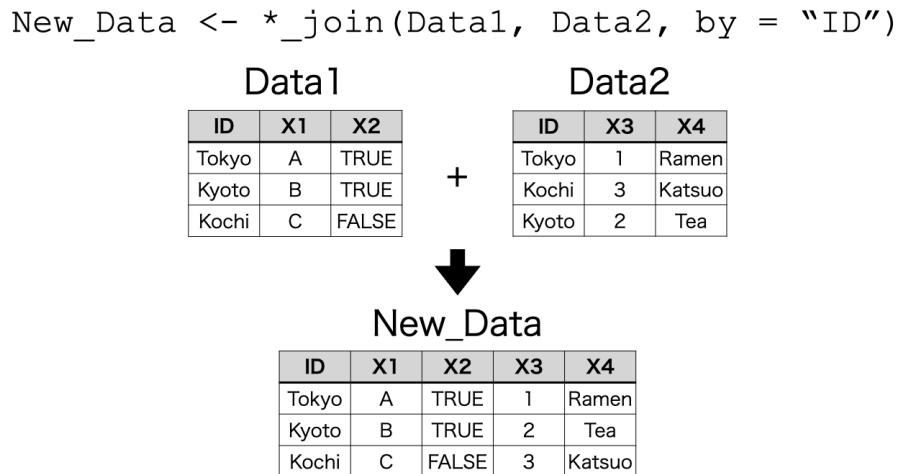


図 13.2: 列の結合

まずは、本章で作成した df2 をもう一回作ってみます。

```
1 df2 <- df %>%
2   group_by(Pref) %>%
3     summarise(Budget_Mean = mean(Budget, na.rm = TRUE),
4               ScoreN_Sum = sum(ScoreN, na.rm = TRUE),
5               Score_Mean = mean(Score, na.rm = TRUE),
6               N = n(),
7               .groups = "drop")
8
9 df2
```

```
## # A tibble: 9 x 5
##   Pref     Budget_Mean ScoreN_Sum Score_Mean     N
##   <chr>     <dbl>      <dbl>     <dbl> <int>
## 1 京都府     1399.      216      3.68    414
## 2 兵庫県     1197.      230      3.54    591
## 3 千葉県     1124.      259      3.72   1000
## 4 和歌山県    1252       83      3.97    140
## 5 埼玉県     1147.      278      3.64   1000
## 6 大阪府     1203.      516      3.77   1000
## 7 奈良県     1169.      45      3.85    147
## 8 東京都     1283.      1165     3.67   1000
## 9 神奈川県    1239.      587      3.53   1000
```

ラーメン屋の店舗ですが、たしかにデータには埼玉、東京、大阪などは 1000 店舗しか入っておりません。実はもっと多いですが、ぐるなび API の仕様上、最大 1000 店舗しか情報取得が出来ないからです。ここに実際の店舗数が入っている新しいデータセット、Ramen2.csv があります。これを読み込み、df3 という名で格納しましょう。

```
1 df3 <- read_csv("Data/Ramen2.csv")
2
3 df3
```

```
## # A tibble: 47 x 15
```

```

##   Pref   Pop   Area RamenN Turnout   LDP   CDP   DPFP Komei    JIP   JCP   SDP
##   <chr> <dbl> <dbl>
## 1 北海道 5.38e6 83424. 1454 53.8 32.3 20.8 6.65 11.7 7.78 11.6 1.31
## 2 青森県 1.31e6 9646. 336 42.9 39.8 22.0 7.24 11.3 3.4 8.31 2.36
## 3 岩手県 1.28e6 15275. 285 56.5 35.5 17.8 12.5 8.22 4.36 10.4 3.83
## 4 宮城県 2.33e6 7282. 557 51.2 39.6 17.8 9.02 11.1 4.6 7.89 2.1
## 5 秋田県 1.02e6 11638. 301 56.3 44.5 13.5 8.64 10.6 4.48 8.09 3.77
## 6 山形県 1.12e6 9323. 512 60.7 45.2 14.9 7.37 9.87 4.28 6.51 5.08
## 7 福島県 1.91e6 13784. 550 52.4 38.2 13.6 12.1 12.8 5.31 7.99 3.01
## 8 茨城県 2.92e6 6097. 663 45.0 39.3 15.2 7.15 15.1 6.73 7.73 1.46
## 9 栃木県 1.97e6 6408. 595 44.1 40.3 18.9 9.94 12.8 4.9 5.04 1.03
## 10 群馬県 1.97e6 6362. 488 48.2 40.6 16.4 9.76 12.4 4.67 7.58 1.87
## # ... with 37 more rows, and 3 more variables: Reiwa <dbl>, NHK <dbl>,
## #   HRP <dbl>

```

変数名	説明
Pref	都道府県名
Pop	日本人人口 (2015年国勢調査)
Area	面積 (2015年国勢調査)
RamenN	ぐるなびに登録されたラーメン屋の店舗数
Turnout	2019年参院選: 投票率 (比例)
LDP	2019年参院選: 自民党の得票率 (比例)
CDP	2019年参院選: 立憲民主党の得票率 (比例)
DPFP	2019年参院選: 国民民主党の得票率 (比例)
Komei	2019年参院選: 公明党の得票率 (比例)
JIP	2019年参院選: 日本維新の会の得票率 (比例)
JCP	2019年参院選: 日本共産党の得票率 (比例)
SDP	2019年参院選: 社会民主党の得票率 (比例)
Reiwa	2019年参院選: れいわ新選組の得票率 (比例)
NHK	2019年参院選: NHKから国民を守る党の得票率 (比例)
HRP	2019年参院選: 幸福実現党の得票率 (比例)

本データは都道府県ごとの人口、面積、ぐるなびに登録されたラーメン屋の店舗数、2019年参議院議員通常選挙の結果が格納されています。人口と面積は2015年国勢調査、ぐるなびの情報は2020年6月時点での情報です。

df2にデータ上の店舗数ではなく、実際の店舗数を新しい列として追加したい場合はどうすれば良いでしょうか。簡単な方法としてはdf3から情報を取得し、それを自分で入れる方法です。

```
1 df3 %>%
2   # df2 の Pref ベクトルの要素と一致するものに絞る
3   filter(Pref %in% df2$Pref) %>%
4   # 都道府県名とラーメン屋の店舗数のみ抽出
5   select(Pref, RamenN)
```

```
## # A tibble: 9 x 2
##   Pref      RamenN
##   <chr>     <dbl>
## 1 埼玉県     1106
## 2 千葉県     1098
## 3 東京都     3220
## 4 神奈川県   1254
## 5 京都府     415
## 6 大阪府     1325
## 7 兵庫県     591
## 8 奈良県     147
## 9 和歌山県   140
```

そして、この情報をdf2\$RamenN <- c(415, 1106, 1254, ...)のように追加すればいいですね。

しかし、このような方法は非効率的です。そもそもdf3から得られた結果の順番とdf2の順番も一致しないので、一々対照しながらベクトルを作ることになります。ここで登場する関数が{dplyr}の\*\_join()関数群です。この関数群には4つの関数が含まれております、以下のような使い方になります。

```
1 # 新しいデータ名ではなく、データ1またはデータ2の名前に格納すると上書きとなる
2
3 # 1. データ1を基準に結合
4 新しいデータ名 <- left_join(データ1, データ2, by = "共通変数名")
5
6 # 2. データ2を基準に結合
7 新しいデータ名 <- right_join(データ1, データ2, by = "共通変数名")
8
9 # 3. データ1とデータ2両方に共通するケースのみ結合
10 新しいデータ名 <- inner_join(データ1, データ2, by = "共通変数名")
11
12 # 4. データ1とデータ2、どちらに存在するケースを結合
13 新しいデータ名 <- full_join(データ1, データ2, by = "共通変数名")
```

4つの関数の違いについて説明する前に、`by`引数について話したいと思います。これは主にキー(key)変数と呼ばれる変数で、それぞれのデータに同じ名前の変数がある必要があります。`df2`と`df3`だとそれがPref変数です。どの\*\_join()関数でも、Prefの値が同じもの同士を結合することになります。

データのキー変数名が異なる場合もあります。たとえば、データ1の都道府県名はPrefという列に、データ2の都道府県名はPrefectureという列になっている場合、`by = "Pref"`でなく、`by = c("データ1のキー変数名" = "データ2のキー変数名")`、つまり、`by = c("Pref" = "Prefecture")`と指定します。

それでは、`df3`から都道府県名とラーメン屋の店舗数だけ抽出し、`df4`として格納しておきます。

```
1 df4 <- df3 %>%
2   select(Pref, RamenN)
3
4 df4
## # A tibble: 47 x 2
##       Pref   RamenN
```

```

## <chr> <dbl>
## 1 北海道 1454
## 2 青森県 336
## 3 岩手県 285
## 4 宮城県 557
## 5 秋田県 301
## 6 山形県 512
## 7 福島県 550
## 8 茨城県 663
## 9 栃木県 595
## 10 群馬県 488
## # ... with 37 more rows

```

これから共通変数名の値をキー (key) と呼びます。今回の例だと Pref が df2 と df4 のキー変数であり、その値である"東京都"、"北海道"などがキーです。

まずは、`inner_join()` の仕組みについて考えます。これは df2 と df4 に共通するキーを持つケースのみ結合する関数です。df4 には"北海道"というキーがありますが、df2 にはありません。したがって、キーが"北海道"のケースは結合から除外されます。これをイメージにしたものが図 13.3 です<sup>3)</sup>。それぞれ  $3 \times 2$  (3 行 2 列) のデータですが、キーが一致するケースは 2 つしかないとため、結合後のデータは  $3 \times 2$  となります。

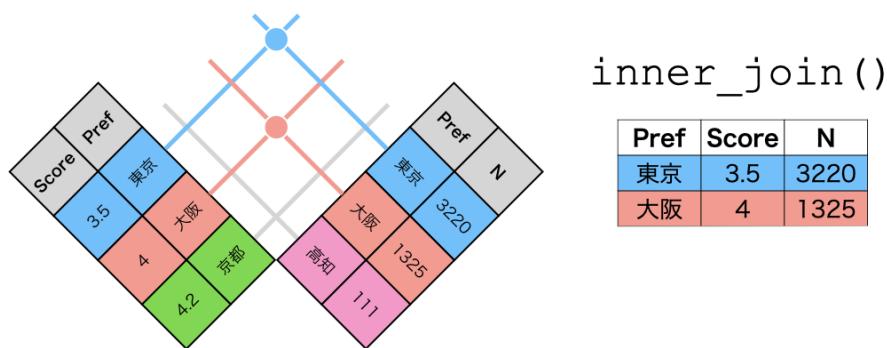


図 13.3: `inner_join()` の仕組み

<sup>3)</sup> これらの図は Garrett Grolemund and Hadley Wickham. 2017. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly. を参考にしました。

実際にやってみましょう。

```
1 inner_join(df2, df4, by = "Pref")  
  
## # A tibble: 9 x 6  
##   Pref     Budget_Mean ScoreN_Sum Score_Mean      N RamenN  
##   <chr>     <dbl>      <dbl>      <dbl> <int>   <dbl>  
## 1 京都府     1399.      216       3.68   414    415  
## 2 兵庫県     1197.      230       3.54   591    591  
## 3 千葉県     1124.      259       3.72   1000   1098  
## 4 和歌山県    1252       83        3.97   140    140  
## 5 埼玉県     1147.      278       3.64   1000   1106  
## 6 大阪府     1203.      516       3.77   1000   1325  
## 7 奈良県     1169.      45        3.85   147    147  
## 8 東京都      1283.     1165      3.67   1000   3220  
## 9 神奈川県    1239.      587      3.53   1000   1254
```

共通するキーは9つのみであり、結果として返されたデータの大きさも $9 \times 6$ です。df2に足されたdf4は2列のデータですが、キー変数であるPrefは共通するため、1列のみ足されました。キー変数を両方残す場合はkeep = TRUE引数を追加してください。

一方、full\_join()は、すべてのキーに対して結合を行います(図@ref(fig: handling2-merge-col-10))。たとえば、df2には"北海道"というキーがありません。それでも新しく出来上がるデータには北海道の列が追加されます。ただし、道内店舗の平均予算、口コミ数などの情報はないため、欠損値が代入されます。

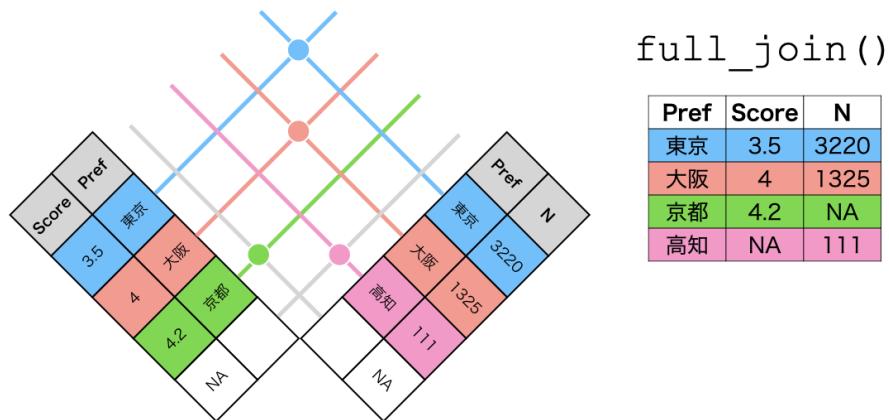


図 13.4: full\_join() の仕組み

それでは実際、結果を確認してみましょう。今回は結合後、RamenN が大きい順で出力します。

```
1 full_join(df2, df4, by = "Pref") %>%
2   arrange(desc(RamenN)) # ぐるなびに登録された店舗の多い都道府県から出力
```

```
## # A tibble: 47 x 6
##   Pref   Budget_Mean ScoreN_Sum Score_Mean      N RamenN
##   <chr>     <dbl>     <dbl>     <dbl> <int>   <dbl>
## 1 東京都    1283.     1165     3.67  1000   3220
## 2 北海道     NA       NA       NA     NA    1454
## 3 大阪府    1203.     516      3.77  1000   1325
## 4 愛知県     NA       NA       NA     NA    1255
## 5 神奈川県   1239.     587      3.53  1000   1254
## 6 埼玉県    1147.     278      3.64  1000   1106
## 7 千葉県    1124.     259      3.72  1000   1098
## 8 福岡県     NA       NA       NA     NA    985
## 9 新潟県     NA       NA       NA     NA    705
## 10 静岡県    NA       NA       NA     NA    679
## # ... with 37 more rows
```

df2 にはなかった北海道や愛知県などの行ができました。そして、df2 にはない情報はす

べて欠損値 (NA) となりました。

続いて、`left_join()` ですが、これは先に指定したデータに存在するキーのみで結合を行います (図@ref(fig: handling2-merge-col-12))。今回は `df2` が先に指定されていますが、`df2` のキーは `df4` のキーの部分集合であるため、`inner_join()` と同じ結果が得られます。

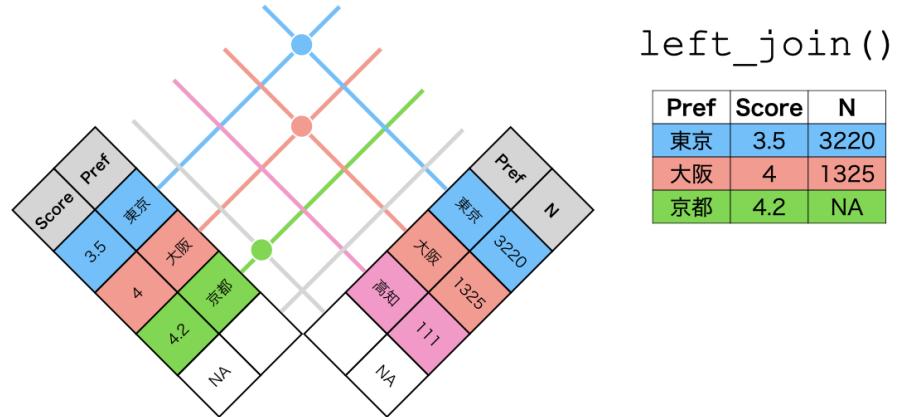


図 13.5: `left_join()` の仕組み

一方、`right_join()` は `left_join()` と逆の関数であり、後に指定したデータに存在するキーを基準に結合を行います (図@ref(fig: handling2-merge-col-13))。後に指定された `df4` のキーは `df2` のキーを完全に含むので、`full_join()` と同じ結果が得られます。

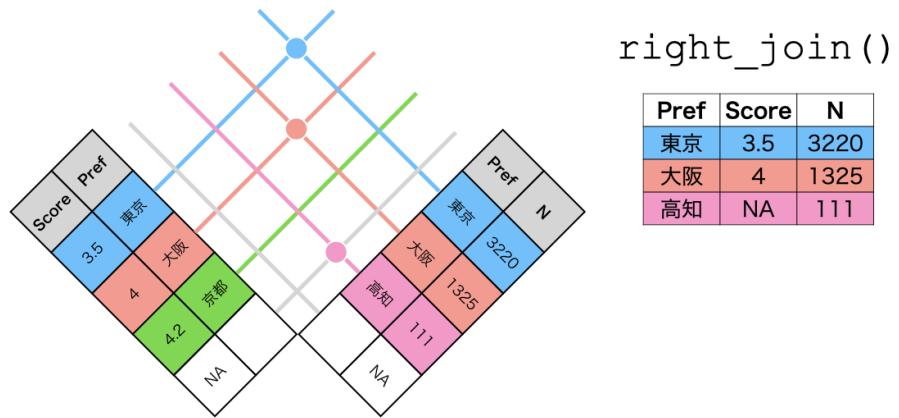


図 13.6: `right_join()` の仕組み

これからは `df2` と `df4` を結合することになりますが、この 2 つの tibble の大きさが異なります。`df2` は 9 つの都府県のみであるに対し、`df4` は 47 都道府県全てのデータが入っているからです。

ここまでではキー変数が一つである場合についてのみ考えましたが、複数のキー変数が必要な場合もあります。たとえば、市区町村の人口・面積データと市区町村の投票率データを結合するとします。各自治体に与えられている「全国地方公共団体コード」が両データに含まれている場合は、このコードをキー変数として使えば問題ありませんが、市区町村名をキー変数として使わざる得ないケースもあるでしょう。しかし、キー変数が複数ある場合もあります。たとえば、府中市は東京都と広島県にありますし、太子町は大阪府と兵庫県にあります。この場合、市区町村名のみでケースをマッチングすると、重複されてマッチングされる恐れがあります。この場合はキー変数を増やすことで対処できます。たとえば、同じ都道府県なら同じ市区町村は存在しないでしょう<sup>4)</sup>。キー変数を複数指定する方法は簡単です。たとえば、市区町村名変数が `Munip`、都道府県名変数が `Pref` なら `by = c("Munip", "Pref")` と指定するだけです。

最後に、キー変数以外の変数名が重複する場合について考えましょう。これはパネルデータを結合する時によく直面する問題です。同じ回答者に 2 回の調査を行った場合、回答者の ID でデータを結合することになります。ただし、それぞれのデータにおいて回答者の性別に関する変数が `F1` という名前の場合、どうなるでしょうか。同じデータの同じ名前の変数が複数あると、非常に扱いにくくなります。実際の結果を見てみましょう。

```
1 Wave1_df <- tibble(ID = c(1, 2, 3, 4, 5),
2                         F1 = c(1, 1, 0, 0, 1),
3                         F2 = c(18, 77, 37, 50, 41),
4                         Q1 = c(1, 5, 2, 2, 3))
5
6 Wave2_df <- tibble(ID = c(1, 3, 4, 6, 7),
7                         F1 = c(1, 0, 0, 0, 1),
8                         F2 = c(18, 37, 50, 20, 62),
9                         Q1 = c(1, 2, 2, 5, 4))
```

4) 行政区も含むなら多くの政令指定都市にある「南区」とか「北区」が重複しますが、ここでは考えないことにしましょう。

```
10
11 full_join(Wave1_df, Wave2_df, by = "ID")

## # A tibble: 7 x 7
##       ID   F1.x   F2.x   Q1.x   F1.y   F2.y   Q1.y
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     1    18     1     1    18     1
## 2     2     1    77     5    NA    NA    NA
## 3     3     0    37     2     0    37     2
## 4     4     0    50     2     0    50     2
## 5     5     1    41     3    NA    NA    NA
## 6     6    NA    NA    NA     0    20     5
## 7     7    NA    NA    NA     1    62     4
```

それぞれの変数名の後に.x と.y が付きます。この接尾辞 (suffix) は `suffix` 引数を指定することで、分析側からカスタマイズ可能です。たとえば、接尾辞を\_W1、\_W2 にしたい場合は

```
1 full_join(Wave1_df, Wave2_df, by = "ID", suffix = c("_W1", "_W2"))

## # A tibble: 7 x 7
##       ID   F1_W1   F2_W1   Q1_W1   F1_W2   F2_W2   Q1_W2
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     1    18     1     1    18     1
## 2     2     1    77     5    NA    NA    NA
## 3     3     0    37     2     0    37     2
## 4     4     0    50     2     0    50     2
## 5     5     1    41     3    NA    NA    NA
## 6     6    NA    NA    NA     0    20     5
## 7     7    NA    NA    NA     1    62     4
```

のように、データ1とデータ2それぞれの接尾辞を指定するだけです。

## 練習問題



## 第 14 章

# データハンドリング [基礎編: factor 型]

### 14.1 名目変数を含むグラフを作成する際の注意点

ここからは楽しい可視化、つまりグラフの作成について解説します。ただし、その前に、名目変数の扱いと簡潔データ構造について話したいと思います。本章では名目変数の扱いについて解説し、次章は簡潔データ構造について解説します。

横軸、または縦軸が気温、成績、身長のような連続変数ではなく、都道府県や国、企業のような名目変数になる場合があります。たとえば、棒グラフの横軸は図 14.1 のように、一般的に名目変数になる場合が多いです。

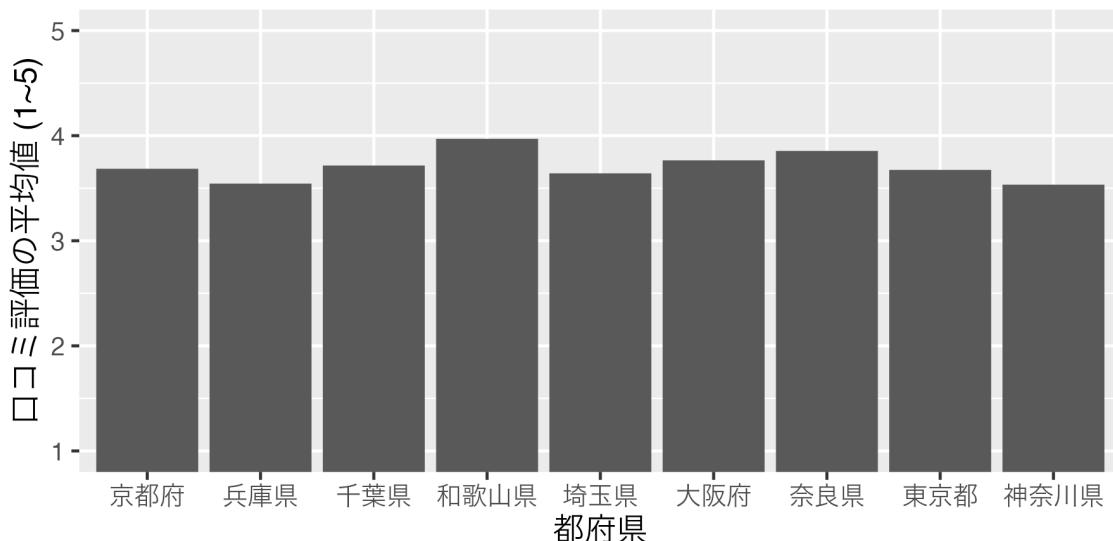


図 14.1: 横軸が名目変数の棒グラフ

ここでは横軸の順番に注目してください。京都府、埼玉県、神奈川県、…の順番になっていますね。「この順番で大満足だよ!」という方がいるかも知れませんが、そうでない方もおおいでしょう。普通考えられるものとしては、都道府県コードの順か、縦軸が高い順(低い順)でしょう。都道府県コードの順だと、埼玉県、千葉県、東京都、神奈川県、京都府、大阪府、兵庫県、奈良県、和歌山県の順番になります。または、縦軸(口コミ評価の平均値)が高い順なら和歌山県、奈良県、大阪府、…の順番になります。あるいは50音順も考えられるでしょう。アメリカの場合、州を並べる際、アルファベット順で並べます。

自分でこの順番をコントロールするには可視化の前の段階、つまりデータハンドリングの段階で順番を決めなくてはなりません。これを決めておかない場合、Rが勝手に順番を指定します。具体的にはロケール(locale)というパソコン内の空間に文字情報が含まれているわけですが、そこに保存されている文字の順番となります。たとえば、日本語ロケールには「京」が「埼」よりも先に保存されているわけです。

したがって、名目変数がグラフに含まれる場合は、名目変数の表示順番を決める必要があり、そこで必要なのが factor 型です。名目変数が character 型の場合、ロケールに保存されている順でソートされますが、factor 型の場合、予め指定した順番でソートされます。

たとえば、前章で使用したデータを用いて、都道府県ごとの口コミ評価の平均値を計算し、その結果を Score\_df として保存します。

```
1 # tidyverse パッケージの読み込み
2 pacman::p_load(tidyverse)
3 # データの読み込み
4 df <- read_csv("Data/Ramen.csv")
5
6 Score_df <- df %>%
7   group_by(Pref) %>%
8   summarise(Score = mean(Score, na.rm = TRUE),
9             .groups = "drop")
10
11 Score_df
12
13 ## # A tibble: 9 x 2
14 ##   Pref     Score
15 ##   <chr>    <dbl>
16 ## 1 京都府    3.68
17 ## 2 兵庫県    3.54
18 ## 3 千葉県    3.72
19 ## 4 和歌山県  3.97
20 ## 5 埼玉県    3.64
21 ## 6 大阪府    3.77
22 ## 7 奈良県    3.85
23 ## 8 東京都    3.67
24 ## 9 神奈川県  3.53
```

この時点で勝手にロケール順になります。実際、表示された Score\_df を見ると Pref の下に<chr>と表記されており、Pref は character 型であることが分かります。これをそのまま棒グラフに出してみましょう。可視化の方法は第 17 章以降で詳細に解説するので、ここでは結果だけに注目してください。

```
1 Score_df %>%
2   ggplot() +
3   geom_bar(aes(x = Pref, y = Score), stat = "identity") +
```

```

4   labs(x = "都府県", y = "口コミ評価の平均値 (1~5)") +
5   theme_gray(base_size = 12)

```

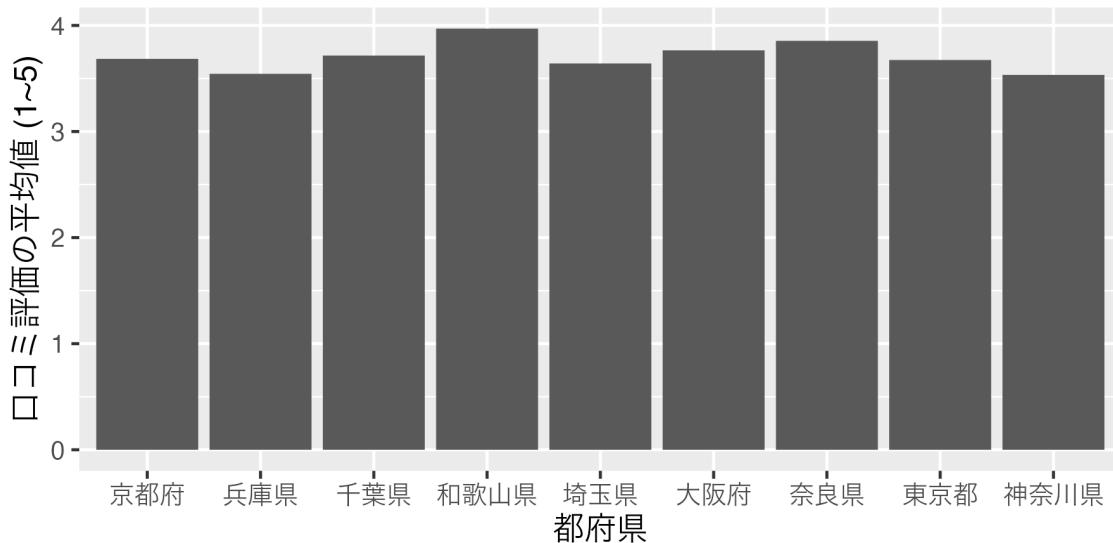


図 14.2: Pref が character 型の場合 (1)

横軸の順番があまり直感的ではありませんね。それでは、Score\_df を Score が高い順にソートし、Score\_df2 で保存してから、もう一回試してみます。

```

1 Score_df2 <- Score_df %>%
2   arrange(desc(Score))
3
4 Score_df2

```

```

## # A tibble: 9 x 2
##   Pref      Score
##   <chr>    <dbl>
## 1 和歌山県  3.97
## 2 奈良県    3.85
## 3 大阪府    3.77
## 4 千葉県    3.72
## 5 京都府    3.68

```

```
## 6 東京都 3.67
## 7 埼玉県 3.64
## 8 兵庫県 3.54
## 9 神奈川県 3.53
```

ここでも `Pref` は `character` 型ですが、とりあえず、これで図を出してみます。

```
1 Score_df2 %>%
2   ggplot() +
3   geom_bar(aes(x = Pref, y = Score), stat = "identity") +
4   labs(x = "都府県", y = "口コミ評価の平均値 (1~5)") +
5   theme_gray(base_size = 12)
```

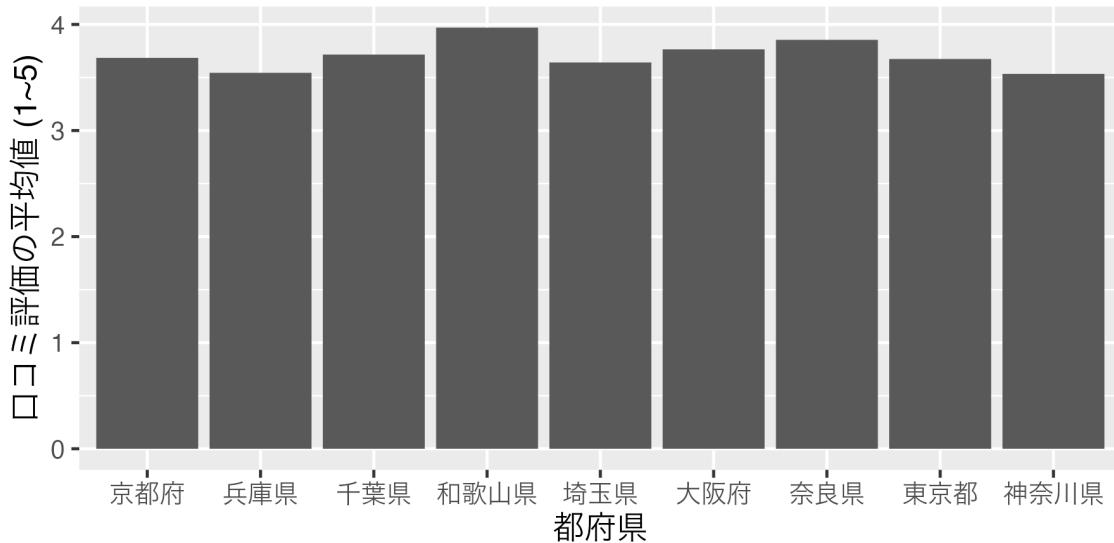


図 14.3: `Pref` が `character` 型の場合 (2)

結果は全く変わっておりません。それでは、`Score_df` の `Pref` 列を `factor` 型に変換し、順番は口コミ評価の平均値が高い順番にしてみましょう。結果は `Score_df_f1` という名で保存します。

```
1 Score_df_f1 <- Score_df %>%
2   mutate(Pref = factor(Pref, levels = c("和歌山県", "奈良県", "大阪府",
3                                "千葉県", "京都府", "東京都",
```

```
4 "埼玉県", "兵庫県", "神奈川県"))
5
6 Score_df_f1

## # A tibble: 9 x 2
##   Pref      Score
##   <fct>    <dbl>
## 1 京都府    3.68
## 2 兵庫県    3.54
## 3 千葉県    3.72
## 4 和歌山県  3.97
## 5 埼玉県    3.64
## 6 大阪府    3.77
## 7 奈良県    3.85
## 8 東京都    3.67
## 9 神奈川県  3.53
```

表示される順番は Score\_df と Score\_df\_f1 も同じですが、Pref のデータ型が<fct>、つまり factor 型であることが分かります。実際、Pref 列だけ抽出した場合、factor 型として、和歌山県から神奈川県の順になっていることが確認できます。

```
1 Score_df_f1$Pref

## [1] 京都府  兵庫県  千葉県  和歌山県  埼玉県  大阪府  奈良県  東京都
## [9] 神奈川県
## 9 Levels: 和歌山県 奈良県 大阪府 千葉県 京都府 東京都 埼玉県 ... 神奈川県
```

この Score\_df\_f1 データを使って、図 14.2 と全く同じコードを実行した結果が図 14.4 です。

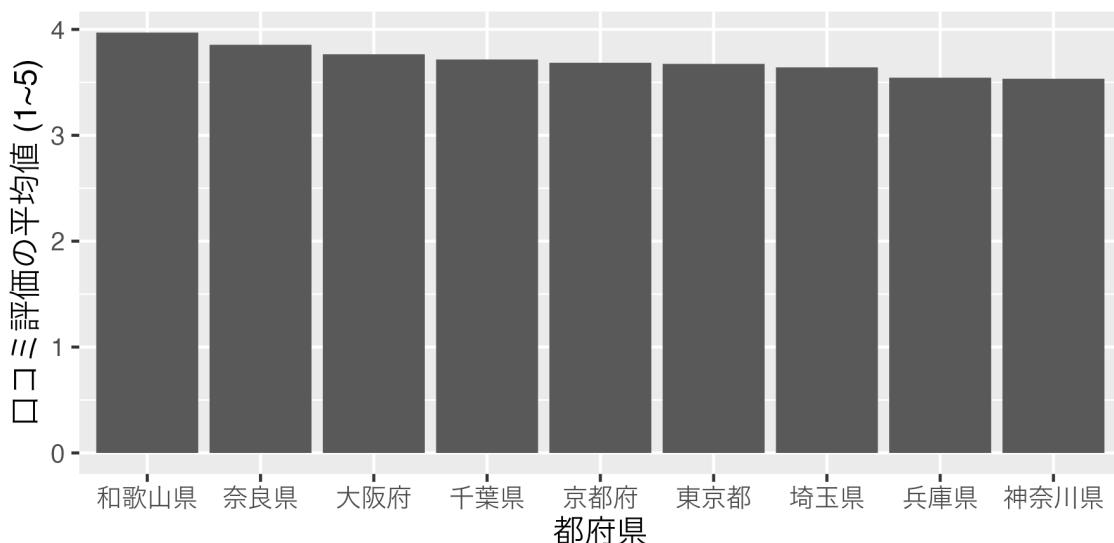


図 14.4: Pref が factor 型の場合

これまでの話をまとめると以下の 2 点が分かります。

1. 変数が character 型である場合、自動的にロケール順でソートされる。
2. 変数が factor 型である場合、データ内の順番やロケール順と関係なく、指定されたレベル（水準）の順でソートされる。

特に 2 番目の点についてですが、これは必ずしも順序付き factor である必要はありません。順序付き factor 型でなくても、`factor()` 内で指定した順にソートされます。もちろん、順序付き factor 型なら指定された順序でソートされます。

これからは factor 型変換の際に便利な関数をいくつか紹介しますが、その前に数値として表現された名目変数について話します。たとえば、`Score_df_f1` に関東地域なら 1 を、その他の地域なら 0 を付けた `Kanto` という変数があるとします。

```

1 Score_df_f1 <- Score_df_f1 %>%
2   mutate(Kanto = ifelse(Pref %in% c("東京都", "神奈川県", "千葉県", "埼玉県"), 1, 0))
3
4 Score_df_f1

```

```

## # A tibble: 9 x 3
##   Pref      Score Kanto
##   <fct>     <dbl> <dbl>
## 1 和歌山県  3.7   0
## 2 奈良県    3.7   0
## 3 大阪府    3.7   0
## 4 千葉県    3.7   0
## 5 京都府    3.7   0
## 6 東京都    3.7   1
## 7 埼玉県    3.7   0
## 8 兵庫県    3.7   0
## 9 神奈川県  3.7   1

```

```
## <fct> <dbl> <dbl>
## 1 京都府 3.68 0
## 2 兵庫県 3.54 0
## 3 千葉県 3.72 1
## 4 和歌山県 3.97 0
## 5 埼玉県 3.64 1
## 6 大阪府 3.77 0
## 7 奈良県 3.85 0
## 8 東京都 3.67 1
## 9 神奈川県 3.53 1
```

Kanto 変数のデータ型は、`<dbl>`、つまり numeric 型です。しかし、これは明らかに名目変数ですね。これをそのまま Kanto を横軸にした図を出すと図 14.5 のようになります。

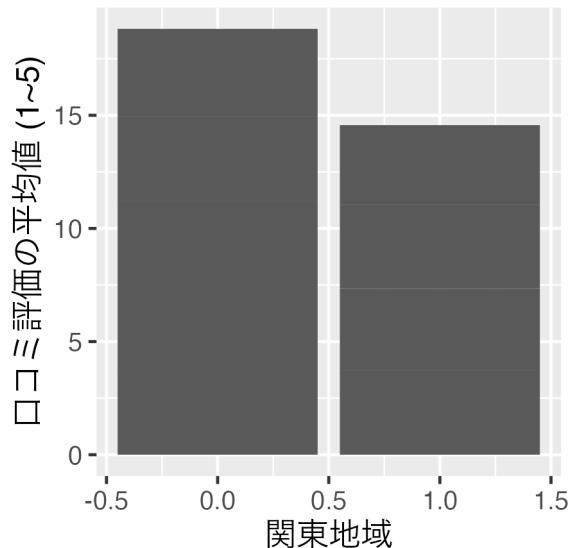


図 14.5: Kanto が numeric 型の場合

この場合、図の横軸は Kanto の値が小さい順でソートされます。ただし、このような図は非常に見にくいため、1 に"関東"、0 に"関西"とラベルを付けた factor 型に変換した方が望ましいです。numeric 型をラベル付きの factor 型にするためには、`levels` 引数には元の数値を、`labels` 引数にはそれぞれの数値に対応したラベルを指定します。また、関東の方を先に出したいので、`factor()` 内の `levels` 引数は `c(0, 1)` でなく、`c(1, 0)`

にします。

```
1 Score_df_f1 <- Score_df_f1 %>%
2   mutate(Kanto = factor(Kanto, levels = c(1, 0), labels = c("関東", "その他")))
3
4 Score_df_f1
```

```
## # A tibble: 9 x 3
##   Pref     Score Kanto
##   <fct>   <dbl> <fct>
## 1 京都府    3.68 その他
## 2 兵庫県    3.54 その他
## 3 千葉県    3.72 関東
## 4 和歌山県  3.97 その他
## 5 埼玉県    3.64 関東
## 6 大阪府    3.77 その他
## 7 奈良県    3.85 その他
## 8 東京都    3.67 関東
## 9 神奈川県  3.53 関東
```

Kanto 変数が factor 型に変換されたことが分かります。

```
1 Score_df_f1$Kanto
```

```
## [1] その他 その他 関東 その他 関東 その他 その他 関東 関東
## Levels: 関東 その他
```

また、"関東"、"その他"の順になっていますね。これを図として出力した結果が図 14.6 です。

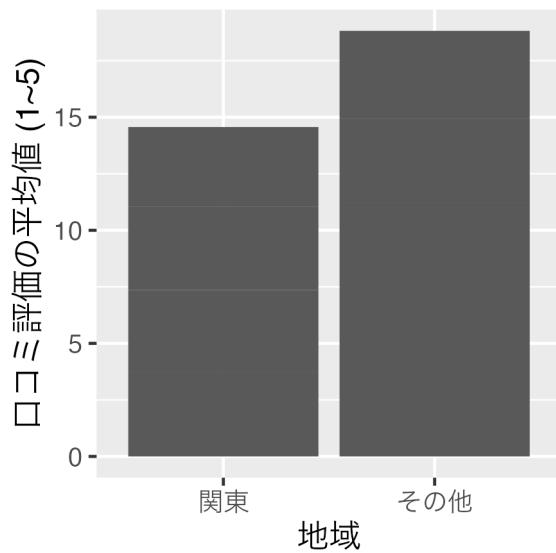


図 14.6: Kanto が factor 型の場合

このように数値型名目変数でも、factor化することによって、自由に横軸の順番を変えることができます。それでは、factor化に使える便利な関数をいくつか紹介します。

## 14.2 {forcats}パッケージについて

実は factor 型への変換や、順番に変更などは全て R 内蔵の `factor()` 関数で対応可能ですが、ここでは {forcats} パッケージが提供している `fct_*` 関数を使用します。{forcats} パッケージは {tidyverse} を読み込む際、自動的に読み込まれるため、既に {tidyverse} を読み込んでいる場合、別途のコードは要りません。

### 14.2.1 `fct_relevel()`: 水準の順番を変更する

`Score_df_f1` の `f1` は `Score` が高い順になっています。これを 50 音順に変更する際、`fct_relevel()` 関数を使います。

```

1 # 新しい変数名と元となる変数名が一致すると上書きになる
2 データフレーム名 %>%
3   mutate(新しい変数名 = fct_relevel(元となる変数名,
```

```
4 "水準 1", "水準 2", "水準 3", ...))
```

ここでは、Pref 変数を再調整した Pref2 変数を作つてみましょう。

```
1 Score_df_f1 <- Score_df_f1 %>%
 2   mutate(Pref2 = fct_relevel(Pref, "大阪府", "神奈川県", "京都府",
 3                               "埼玉県", "千葉県", "東京都",
 4                               "奈良県", "兵庫県", "和歌山県"))
 5
 6 Score_df_f1
```

```
## # A tibble: 9 x 4
##   Pref      Score Kanto  Pref2
##   <fct>    <dbl> <fct> <fct>
## 1 京都府    3.68 その他 京都府
## 2 兵庫県    3.54 その他 兵庫県
## 3 千葉県    3.72 関東   千葉県
## 4 和歌山県  3.97 その他 和歌山県
## 5 埼玉県    3.64 関東   埼玉県
## 6 大阪府    3.77 その他 大阪府
## 7 奈良県    3.85 その他 奈良県
## 8 東京都    3.67 関東   東京都
## 9 神奈川県  3.53 関東   神奈川県
```

一見、Pref と Pref2 変数は同じように見えますが、水準はどうなつてゐるでしょうか。

```
1 levels(Score_df_f1$Pref) # Pref の水準
## [1] "和歌山県" "奈良県"   "大阪府"    "千葉県"    "京都府"    "東京都"    "
## [7] "埼玉県"
## [8] "兵庫県"   "神奈川県"
1 levels(Score_df_f1$Pref2) # Pref2 の水準
## [1] "大阪府"   "神奈川県" "京都府"   "埼玉県"   "千葉県"   "東京都"   "
```

```
奈良県"
```

```
## [8] "兵庫県" "和歌山県"
```

問題なく50音順になっていることが分かります。他にも`fct_relevel()`には全ての水準名を指定する必要はありません。一部の水準名も可能です。たとえば、「関東が関西の先に来るなんでけしからん！」と思う読者もいるでしょう。この場合、関西の府県名を入れると、指定した水準が最初に位置するようになります。

```
1 Score_df_f1 <- Score_df_f1 %>%
2   mutate(Pref3 = fct_relevel(Pref, "京都府", "大阪府",
3                               "兵庫県", "奈良県", "和歌山県"))
4
5 levels(Score_df_f1$Pref3) # Pref3 の水準

## [1] "京都府" "大阪府" "兵庫県" "奈良県" "和歌山県" "千葉県" "
東京都"
## [8] "埼玉県" "神奈川県"
```

一部の水準名のみを指定するとその水準が最初に移動されますが、`after`引数を指定すると、位置を調整することも可能です。`after = 2`の場合、元となる変数の1、3番目の水準は維持され、3番目以降に指定した水準、それに続いて指定されていない水準の順番になります。Prefは和歌山、奈良、大阪の順ですが、ここで京都と東京を、奈良と大阪の間に移動するなら、

```
1 Score_df_f1 <- Score_df_f1 %>%
2   mutate(Pref4 = fct_relevel(Pref, "京都府", "東京都", after = 2))
3
4 levels(Score_df_f1$Pref4) # Pref4 の水準

## [1] "和歌山県" "奈良県" "京都府" "東京都" "大阪府" "千葉県" "
埼玉県"
## [8] "兵庫県" "神奈川県"
```

のように書きます。`after`を指定しない場合のデフォルト値は0であるため、最初に移動します。

### 14.2.2 fct\_recode(): 水準のラベルを変更する

fct\_recode() は水準のラベルを変更する時に使う関数で、以下のように使います。

```
1 # 新しい変数名と元となる変数名が一致すると上書きになる
2 データフレーム名 %>%
3   mutate(新しい変数名 = fct_recode(元となる変数名,
4                                     新しいラベル1 = "既存のラベル1",
5                                     新しいラベル2 = "既存のラベル2",
6                                     新しいラベル3 = "既存のラベル3",
7                                     ...))
```

注意点としては新しいラベルは"で囲まず、既存のラベルは"で囲む点です。それでは、Pref のラベルをローマ字に変更してみましょう。

```
1 Score_df_f1 <- Score_df_f1 %>%
2   mutate(Pref5 = fct_recode(Pref,
3                             Saitama = "埼玉県",
4                             Wakayama = "和歌山県",
5                             Kyoto = "京都府",
6                             Osaka = "大阪府",
7                             Tokyo = "東京都",
8                             Nara = "奈良県",
9                             Kanagawa = "神奈川県",
10                            Hyogo = "兵庫県",
11                            Chiba = "千葉県"))
12
13 Score_df_f1
```

```
## # A tibble: 9 x 7
##   Pref      Score Kanto  Pref2     Pref3     Pref4     Pref5
##   <fct>    <dbl> <fct> <fct>    <fct>    <fct>    <fct>
## 1 京都府    3.68 その他 京都府  京都府  京都府  Kyoto
```

```
## 2 兵庫県 3.54 その他 兵庫県 兵庫県 兵庫県 Hyogo
## 3 千葉県 3.72 関東 千葉県 千葉県 千葉県 Chiba
## 4 和歌山県 3.97 その他 和歌山県 和歌山県 和歌山県 Wakayama
## 5 埼玉県 3.64 関東 埼玉県 埼玉県 埼玉県 Saitama
## 6 大阪府 3.77 その他 大阪府 大阪府 大阪府 Osaka
## 7 奈良県 3.85 その他 奈良県 奈良県 奈良県 Nara
## 8 東京都 3.67 関東 東京都 東京都 東京都 Tokyo
## 9 神奈川県 3.53 関東 神奈川県 神奈川県 神奈川県 Kanagawa
```

`fct_recode()` の中に指定する水準の順番は無視されます。つまり、水準の順番はそのまま維持されるため、好きな順番で結構です。また、全ての水準を指定せず、一部のみ変更することも可能です。それでは `Pref5` の順番が `Pref` の順番と同じかを確認してみましょう。

```
1 levels(Score_df_f1$Pref) # Pref の水準
## [1] "和歌山県" "奈良県"    "大阪府"     "千葉県"     "京都府"     "東京都"     "
埼玉県"
## [8] "兵庫県"    "神奈川県"

1 levels(Score_df_f1$Pref5) # Pref5 の水準
## [1] "Wakayama" "Nara"      "Osaka"      "Chiba"      "Kyoto"      "Tokyo"      "Saitama"
## [8] "Hyogo"     "Kanagawa"
```

### 14.2.3 `fct_rev()`: 水準の順番を反転させる

水準の順番を反転することは非常によくあります。たとえば、グラフの読みやすさのために、左右または上下を反転するケースがあります。既に何回も強調しましたように、名目変数は基本的に `factor` 型にすべきであり、ここで `fct_rev()` 関数が非常に便利です。たとえば、`Pref2` の水準は 50 音順でありますが、これを反転し、`Pref6` という名の列として追加してみましょう。

```
1 Score_df_f1 <- Score_df_f1 %>%
2   mutate(Pref6 = fct_rev(Pref2))
```

```
3
4  levels(Score_df_f1$Pref6)

## [1] "和歌山県" "兵庫県"    "奈良県"    "東京都"    "千葉県"    "埼玉県"    "
京都府"
## [8] "神奈川県" "大阪府"
```

関数一つで水準の順番が反転されました。

#### 14.2.4 fct\_infreq(): 頻度順に順番を変更する

続いて、水準の順番を頻度順に合わせる `fct_infreq()` 関数です。たとえば、`Score` が欠損でないケースのみで構成された `df2` を考えてみましょう。

```
1 df2 <- df %>%
2   filter(!is.na(Score))
```

そして、都府県ごとのケース数を計算します。

```
1 table(df2$Pref)

##
##   京都府   兵庫県   千葉県 和歌山県   埼玉県   大阪府   奈良県   東京都
##       79       85      108       24      118      175       28      298
## 神奈川県
##       219
```

ここで `Pref` を `factor` 化しますが、水準の順番を店舗数が多い方を先にするにはどうすれば良いでしょうか。`fct_infreq()` 関数は指定された変数の各値の個数を計算し、多い順に `factor` の水準を調整します。

```
1 df2 <- df2 %>%
2   # 多く出現した値順で factor 化する
3   mutate(Pref = fct_infreq(Pref))
4
5  levels(df2$Pref) # df2 の Pref 変数の水準を出力
```

```
## [1] "東京都"    "神奈川県"  "大阪府"    "埼玉県"    "千葉県"    "兵庫県"    "
京都府"
## [8] "奈良県"    "和歌山県"
```

"東京都"、"神奈川県"、"大阪府"、...の順で水準の順番が調整され、これは `table(df$Pref2)` の順位とも一致します。

#### 14.2.5 `fct_inorder()`: データ内の出現順番に順番を変更する

続いて、`fct_inorder()` ですが、これは意外と頻繁に使われる関数です。たとえば、自分でデータフレームなどを作成し、ケースの順番も綺麗に整えたとします。しかし、既に指摘した通り、データフレーム（または、`tibble`）での順番とグラフにおける順番は一致するとは限りません。データフレームに格納された順番で `factor` の水準が設定できれば非常に便利でしょう。そこで使うのが `fct_inorder()` です。

たとえば、`df` の `Pref` は"東京都"が 1000 個並び、続いて"神奈川県"が 1000 個、"千葉県"が 1000 個、...の順番で格納されています。この順番をそのまま `factor` の順番にするには以下のように書きます。

```
1 df3 <- df %>%
2   # Pref 変数を factor 化し、水準は出現順とする
3   # 変換後の結果は Pref に上書きする
4   mutate(Pref = fct_inorder(Pref))
5
6 levels(df3$Pref)

## [1] "東京都"    "神奈川県"  "千葉県"    "埼玉県"    "大阪府"    "京都府"    "
兵庫県"
## [8] "奈良県"    "和歌山県"
```

#### 14.2.6 `fct_shift()`: 水準の順番をずらす

続いて、水準の順番をずらす `fct_shift()` 関数を紹介します。たとえば、「1: そう思う」～「5: そう思わない」、「9: 答えたたくない」の 6 水準で構成された変数があるとします。

```
1 df4 <- tibble(
2   ID = 1:10,
3   Q1 = c(1, 5, 3, 2, 9, 2, 4, 9, 5, 1)
4 )
5
6 df4 <- df4 %>%
7   mutate(Q1 = factor(Q1, levels = c(1:5, 9),
8                 labels = c("そう思う",
9                           "どちらかと言えばそう思う",
10                          "どちらとも言えない",
11                          "どちらかと言えばそう思わない",
12                          "そう思わない",
13                          "答えたくない")))
14
15 df4

## # A tibble: 10 x 2
##       ID   Q1
##   <int> <fct>
## 1     1  そう思う
## 2     2  そう思わない
## 3     3 どちらとも言えない
## 4     4 どちらかと言えばそう思う
## 5     5  答えたくない
## 6     6 どちらかと言えばそう思う
## 7     7 どちらかと言えばそう思わない
## 8     8  答えたくない
## 9     9  そう思わない
## 10    10  そう思う
```

水準の順番も「そう思う」～「答えたくない」順で綺麗に整っています。この水準を反転するには `fct_rev()` 関数が便利です。Q1 の水準を反転した変数を Q1\_R という新しい列として追加し、水準を確認してみましょう。

```

1 df4 <- df4 %>%
2   mutate(Q1_R = fct_rev(Q1))
3
4 df4

## # A tibble: 10 x 3
##       ID Q1          Q1_R
##   <int> <fct>       <fct>
## 1     1 そう思う      そう思う
## 2     2 そう思わない  そう思わない
## 3     3 どちらとも言えない どちらとも言えない
## 4     4 どちらかと言えばそう思う どちらかと言えばそう思う
## 5     5 答えたたくない  答えたたくない
## 6     6 どちらかと言えばそう思う どちらかと言えばそう思う
## 7     7 どちらかと言えばそう思わない どちらかと言えばそう思わない
## 8     8 答えたたくない  答えたたくない
## 9     9 そう思わない  そう思わない
## 10    10 そう思う      そう思う

```

```

1 levels(df4$Q1_R)

## [1] "答えたたくない"          "そう思わない"
## [3] "どちらかと言えばそう思わない" "どちらとも言えない"
## [5] "どちらかと言えばそう思う"    "そう思う"

```

「答えたたくない」が最初の順番に来ましてね。できれば、「そう思わない」～「そう思う」、「答えたたくない」の順番にしたいところです。ここで使うのが `fct_shift()` ですが、書き方がややこしいので、噛み砕いて解説します。

```

1 # fct_shift() の使い方
2 データ名 %>%
3   mutate(新しい変数名 = fct_shift(元の変数名, n = 左方向へずらす個数))

```

問題は `n =` 引数ですが、その挙動については以下の表を参照してください。

水準の順番	1 番目	2 番目	3 番目	4 番目	5 番目	6 番目
<code>n = -2</code>	"E"	"F"	"A"	"B"	"C"	"D"
<code>n = -1</code>	"F"	"A"	"B"	"C"	"D"	"E"
<code>n = 0</code>	"A"	"B"	"C"	"D"	"E"	"F"
<code>n = 1</code>	"B"	"C"	"D"	"E"	"F"	"A"
<code>n = 2</code>	"C"	"D"	"E"	"F"	"A"	"B"

具体的には水準は左方向へ `n` 個移動します。元の水準が A, B, C, ..., F の順で、`n = 1` の場合、A が F の後ろへ移動し、B, C, D, E, F が前の方へ 1 つずつ移動します。逆に右側へ 1 つ移動したい場合は `n = -1` のように書きます。今回は最初の水準を最後に移動させたいので、`n = 1` と指定します。

```

1 df4 <- df4 %>%
2   # Q1_R の水準を左方向で 1 ずらす
3   mutate(Q1_R = fct_shift(Q1_R, n = 1))
4
5 levels(df4$Q1_R)

## [1] "そう思わない"           "どちらかと言えばそう思わない"
## [3] "どちらとも言えない"     "どちらかと言えばそう思う"
## [5] "そう思う"               "答えたくない"

```

これで水準の反転が完了しました。`fct_shift()` はこのように世論調査データの処理に便利ですが、他にも曜日の処理に使えます。例えば、1 週間の始まりを月曜にするか日曜にするかによって、`fct_shift()` を使うケースがあります。

#### 14.2.7 `fct_shuffle()`: 水準の順番をランダム化する

あまり使わない機能ですが、水準の順番をランダム化することも可能です。使い方は非常に簡単で、`fct_shuffle()` に元の変数名を入れるだけです。たとえば、`Score_df` の `Pref` の順番をランダム化し、`Pref2` として追加します。同じことをもう 2 回繰り返し、それぞれ `Pref3` と `Pref4` という名前で追加してみましょう。

```
1 Score_df <- Score_df %>%
2   mutate(Pref2 = fct_shuffle(Pref),
3         Pref3 = fct_shuffle(Pref),
4         Pref4 = fct_shuffle(Pref))
5
6 Score_df
## # A tibble: 9 x 5
##   Pref     Score Pref2     Pref3     Pref4
##   <chr>    <dbl> <fct>    <fct>    <fct>
## 1 京都府    3.68 京都府    京都府    京都府
## 2 兵庫県    3.54 兵庫県    兵庫県    兵庫県
## 3 千葉県    3.72 千葉県    千葉県    千葉県
## 4 和歌山県  3.97 和歌山県  和歌山県  和歌山県
## 5 埼玉県    3.64 埼玉県    埼玉県    埼玉県
## 6 大阪府    3.77 大阪府    大阪府    大阪府
## 7 奈良県    3.85 奈良県    奈良県    奈良県
## 8 東京都    3.67 東京都    東京都    東京都
## 9 神奈川県  3.53 神奈川県  神奈川県  神奈川県
1 levels(Score_df$Pref2)
## [1] "東京都"    "神奈川県"   "京都府"    "大阪府"    "千葉県"    "兵庫県"    "
## 和歌山県"
## [8] "奈良県"    "埼玉県"
1 levels(Score_df$Pref3)
## [1] "兵庫県"    "京都府"    "千葉県"    "奈良県"    "神奈川県"   "大阪府"    "
## 埼玉県"
## [8] "東京都"    "和歌山県"
1 levels(Score_df$Pref4)
## [1] "奈良県"    "京都府"    "神奈川県"   "兵庫県"    "東京都"    "埼玉県"    "
```

```
大阪府"
```

```
## [8] "和歌山県" "千葉県"
```

Pref から Pref4 まで同じように見えますが、水準の順番が異なります (Pref は character 型だから水準がありません)。

#### 14.2.8 fct\_reorder(): 別の 1 変数の値を基準に水準の順番を変更する

fct\_infreq() は出現頻度順に並び替える関数でしたが、それと似たような関数として fct\_reorder() があります。ただし、これは出現頻度を基準にするのではなく、ある変数の平均値が低い順、中央値が高い順などでソートされます。まずは使い方から確認します。

```
1 データ名 %>%
2   mutate(新しい変数名 = fct_reorder(元の変数名, 基準となる変数,
3                                     関数名, 関数の引数))
```

必要な引数が多いですね。解説よりも実際の例を見ながら説明します。今回も Pref を factor 変数にし、Pref\_R という列で格納しますが、平均予算が安い順で factor の水準を決めたいと思います。

```
1 df <- df %>%
2   mutate(Pref_R = fct_reorder(Pref, Budget, mean, na.rm = TRUE))
3
4 levels(df$Pref_R)

## [1] "千葉県"    "埼玉県"    "奈良県"    "兵庫県"    "大阪府"    "神奈川県"  "
和歌山県"
## [8] "東京都"    "京都府"
```

Pref\_R の水準は千葉県、埼玉県、奈良県、...の順ですが、本当にそうでしょうか。 group\_by() と summarise() などを使って確認してみましょう。

```
1 df %>%
2   group_by(Pref) %>%
```

```

3   summarise(Budget = mean(Budget, na.rm = TRUE),
4     .groups = "drop") %>%
5   arrange(Budget)

## # A tibble: 9 x 2
##   Pref     Budget
##   <chr>    <dbl>
## 1 千葉県    1124.
## 2 埼玉県    1147.
## 3 奈良県    1169.
## 4 兵庫県    1197.
## 5 大阪府    1203.
## 6 神奈川県  1239.
## 7 和歌山県  1252
## 8 東京都    1283.
## 9 京都府    1399.

```

問題なくソートされましたね。注意点としては `fct_reorder()` 内に関数名を書く際、() は不要という点です。関数名の次の引数としてはその関数に別途必要な引数を指定します。引数が省略可能、あるいは不要な関数を使う場合は、省略しても構いませんし、数に制限はありません。

また、低い順ではなく、高い順にすることも可能です。次は `Score` の中央値が高い順に水準を設定した `Pref_R2` を作ってみましょう。

```

1 df <- df %>%
2   mutate(Pref_R2 = fct_reorder(Pref, Score, median, na.rm = TRUE, .desc = TRUE))
3
4 levels(df$Pref_R2)

## [1] "和歌山県" "奈良県"    "千葉県"    "大阪府"    "東京都"    "埼玉県"    "
## [6] "京都府"
## [8] "兵庫県"    "神奈川県"

```

変わったのは `mean` の代わりに `median` を使ったこと、そして `.desc` 引数が追加された点です。`fct_reorder()` には `.desc = FALSE` がデフォルトとして指定されており、省略した場合は昇順で `factor` の水準が決まります。ここで `.desc = TRUE` を指定すると、降順となります。実際、`Score` の中央値順になっているかを確認してみましょう。

```
1 df %>%
2   group_by(Pref) %>%
3   summarise(Score = median(Score, na.rm = TRUE),
4             .groups = "drop") %>%
5   arrange(desc(Score))
```

```
## # A tibble: 9 x 2
##   Pref     Score
##   <chr>   <dbl>
## 1 和歌山県 4
## 2 奈良県  3.88
## 3 千葉県  3.75
## 4 大阪府  3.75
## 5 東京都  3.64
## 6 埼玉県  3.61
## 7 京都府  3.5
## 8 兵庫県  3.5
## 9 神奈川県 3.5
```

#### 14.2.9 `fct_reorder2()`: 別の 2 変数の値を基準に水準の順番を変更する

この関数は別の変数を基準に水準が調整される点では `fct_reorder()` と類似しています。ただし、よく誤解されるのは「変数 A の値が同じなら変数 B を基準に...」といったものではありません。たとえば、`fct_reorder(x, y, mean)` の場合、y の平均値 (`mean()`) の順で x の水準を調整するという意味です。この `mean()` 関数に必要なデータはベクトル 1 つです。しかし、関数によっては 2 つの変数が必要な場合があります。

これは頻繁に直面する問題ではありませんが、この `fct_reorder2()` 関数が活躍するケースを紹介します。以下は6月27日から7月1日までの5日間、5地域におけるCOVID-19新規感染者数を表したデータです<sup>1)</sup>。入力が面倒な方はここからダウンロードして読み込んでください。

```
1 # 入力が面倒ならデータをダウンロードし、
2 # Reorder2_df <- read_csv("Data/COVID19.csv")
3 Reorder2_df <- tibble(
4   Country = rep(c("日本", "韓国", "中国 (本土)", "台湾", "香港"),
5                 each = 5),
6   Date     = rep(c("2020/06/27", "2020/06/28", "2020/06/29",
7                   "2020/06/30", "2020/07/01"), 5),
8   NewPat   = c(100, 93, 86, 117, 130,
9               62, 42, 43, 50, 54,
10              17, 12, 19, 3, 5,
11              0, 0, 0, 0, 0,
12              1, 2, 4, 2, 28)
13 )
14
15 Reorder2_df <- Reorder2_df %>%
16   mutate(Date = as.Date(Date))
17
18 Reorder2_df
```

```
## # A tibble: 25 x 3
##   Country Date     NewPat
##   <chr>   <date>   <dbl>
## 1 日本    2020-06-27 100
## 2 日本    2020-06-28 93
## 3 日本    2020-06-29 86
## 4 日本    2020-06-30 117
```

---

<sup>1)</sup> データの出典はGoogleです。

```
## 5 日本 2020-07-01 130
## 6 韓国 2020-06-27 62
## 7 韓国 2020-06-28 42
## 8 韓国 2020-06-29 43
## 9 韓国 2020-06-30 50
## 10 韓国 2020-07-01 54
## # ... with 15 more rows
```

可視化のコードはとりあえず無視し、グラフを出力してみましょう。

```
1 Reorder2_df %>%
2   ggplot() +
3   geom_line(aes(x = Date, y = NewPat, color = Country),
4             size = 1) +
5   scale_x_date(date_labels = "%Y年%m月%d日") +
6   labs(x = "年月日", y = "新規感染者数(人)", color = "") +
7   theme_gray(base_size = 12)
```

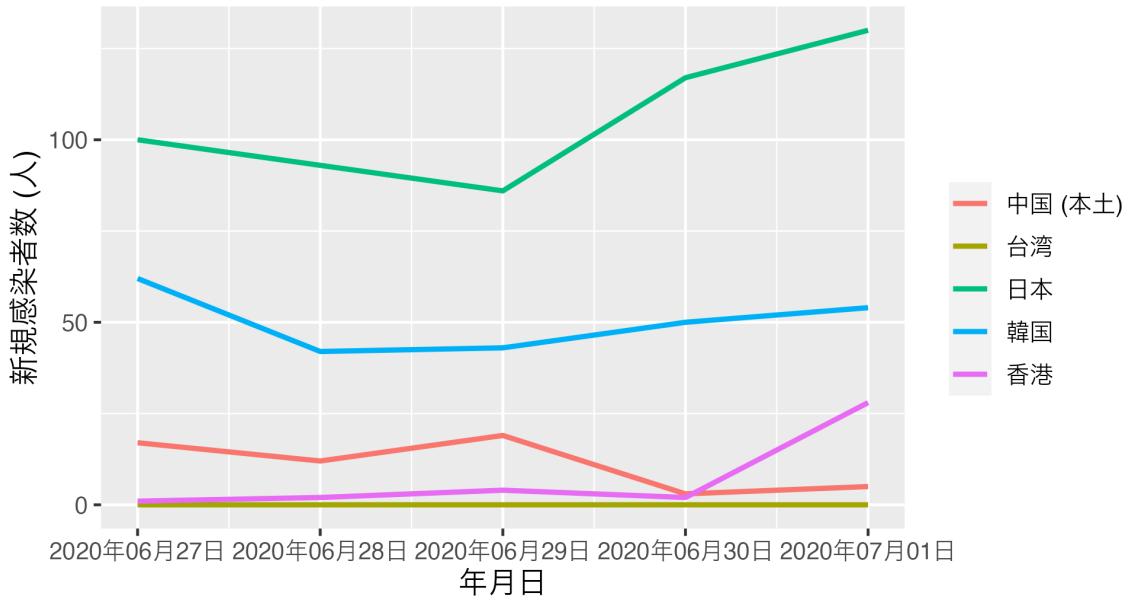


図 14.7: 国名の順番を変更した前

このグラフに違和感はありませんが、「読みやすさ」の面では改善の余地があります。たとえば、7月1日の時点で、新規感染者数が多いのは日本、韓国、香港、中国（本土）、台湾の順です。しかし、右側の凡例の順番はそうではありません。この順番が一致すれば、更に図は読みやすくなるでしょう。

```
1 factor(Reorder2_df$Country)

## [1] 日本      日本      日本      日本      日本      韓国
## [7] 韓国      韓国      韓国      韓国      中国 (本土) 中
国 (本土)
## [13] 中国 (本土) 中国 (本土) 中国 (本土) 台湾      台湾      台湾
## [19] 台湾      台湾      香港      香港      香港      香港
## [25] 香港

## Levels: 中国 (本土) 台湾 日本 韓国 香港
```

実際、何も指定せずに Reorder2\_df の Country を factor 化すると、韓国、香港、台湾、… の順であり、これは上のグラフと一致します。これをグラフにおける7月1日の新規感染者数の順で並べるために、Date を昇順にソートし、そして最後の要素 ("2020/07/01") 内で新規感染者数 (NewPat) を降順に並べ替えた場合の順番にする必要があります。実際、Reorder2\_df を Date で昇順、NewPat で降順にソートし、最後の5行を抽出した結果が以下のコードです。

```
1 Reorder2_df %>%
2   arrange(Date, desc(NewPat)) %>%
3   slice_tail(n = 5)

## # A tibble: 5 x 3
##   Country     Date   NewPat
##   <chr>       <date>  <dbl>
## 1 日本        2020-07-01 130
## 2 韓国        2020-07-01  54
## 3 香港        2020-07-01  28
## 4 中国 (本土) 2020-07-01   5
## 5 台湾        2020-07-01   0
```

このように、水準を調整する際に 2 つの変数 (Date と NewPat) が使用されます。fct\_reorder2() は fct\_reorder() と買い方がほぼ同じですが、基準となる変数がもう一つ加わります。

```
1 データ名 %>%
2   mutate(新しい変数名 = fct_reorder2(元の変数名,
3                                     基準となる変数 1, 基準となる変数 2,
4                                     関数名, 関数の引数))
```

重要なのはここの関数のところですが、fct\_reorder2() はデフォルトで last2() という関数が指定されており、まさに私たちに必要な関数です。したがって、ここでは関数名も省略できますが、ここでは一応明記しておきます。

```
1 Reorder2_df <- Reorder2_df %>%
2   mutate(Country2 = fct_reorder2(Country, Date, NewPat, last2))
```

それでは新しく出来た Country2 の水準を確認してみましょう。

```
1 levels(Reorder2_df$Country2)
## [1] "日本"          "韓国"          "香港"          "中国 (本土)" "台湾"
```

ちゃんと 7 月 1 日の新規感染者数基準で水準の順番が調整されたので、これを使ってグラフをもう一回作ってみます。

```
1 Reorder2_df %>%
2   ggplot() +
3   geom_line(aes(x = Date, y = NewPat, color = Country2),
4             size = 1) +
5   scale_x_date(date_labels = "%Y 年 %m 月 %d 日") +
6   labs(x = "年月日", y = "新規感染者数 (人)", color = "") +
7   theme_gray(base_size = 12)
```

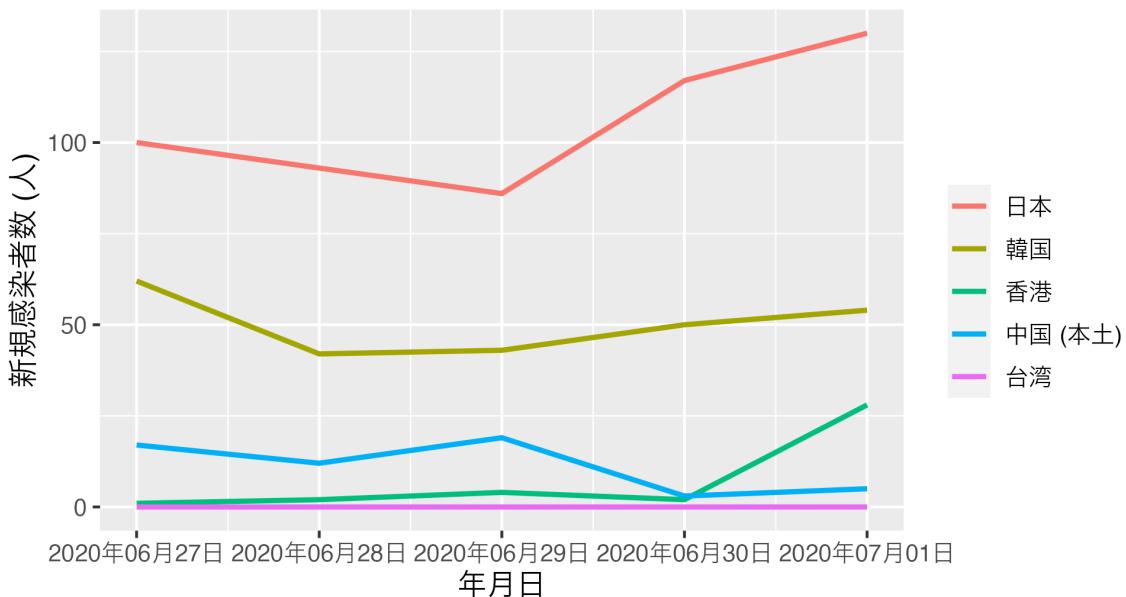


図 14.8: 国名の順番を変更した後

これで図がさらに読みやすくなりました。ちなみに、{forcats}パッケージは `last2()` 以外にも `first2()` という関数も提供しております。これを使うと、7月1日でなく、6月27日の新規感染者数の降順で水準の順番が調整されます。他にも引数を2つ使用する自作関数も使えますが、`fct_reorder2()` の主な使いみちは `last2()` で十分でしょう。

#### 14.2.10 `fct_collapse()`: 水準を統合する

水準数をより水準数に減らすためには、`fct_recode()` を使います。先ほど、`fct_shift()` で使った `df4` の例を考えてみましょう。`df4` の Q1 の水準数は6つです。

```
1 levels(df4$Q1)

## [1] "そう思う"          "どちらかと言えばそう思う"
## [3] "どちらとも言えない" "どちらかと言えばそう思わない"
## [5] "そう思わない"       "答えたくない"
```

これを4つに減らして見ましょう。具体的には「そう思う」と「どちらかと言えばそう思う」を「そう思う」に、「そう思わない」と「どちらかと言えばそう思わない」を「そう

思わない」に統合します。これを `fct_recode()` で処理したのが以下のコードです。

```
1 # fct_recode() を使った例
2 df4 <- df4 %>%
3   mutate(Q1_R2 = fct_recode(Q1,
4     そう思う = "そう思う",
5     そう思う = "どちらかと言えばそう思う",
6     どちらとも言えない = "どちらとも言えない",
7     そう思わない = "どちらかと言えばそう思わない",
8     そう思わない = "そう思わない",
9     答えたくない = "答えたくない"))
10
11 df4
## # A tibble: 10 x 4
##       ID Q1           Q1_R           Q1_R2
##   <int> <fct>        <fct>        <fct>
## 1     1 そう思う      そう思う      そ
## う思う
## 2     2 そう思わない そう思わない そ
## う思わない
## 3     3 どちらとも言えない どちらとも言えない どち
## らとも言~
## 4     4 どちらかと言えばそう思う どちらかと言えばそう思う そ
## う
## 5     5 答えたくない  答えたくない  答
## えたくない
## 6     6 どちらかと言えばそう思う どちらかと言えばそう思う そ
## う
## 7     7 どちらかと言えばそう思わない どちらかと言えばそう思わない そ
## う思
## ない
## 8     8 答えたくない  答えたくない  答
## えたくない
```

```

## 9      9 そう思わない          そう思わない          そ
う思わない
## 10     10 そう思う          そう思う          そ
う思う
1  levels(df4$Q1_R2)

## [1] "そう思う"          "どちらとも言えない" "そう思わない"
## [4] "答えたくない"

```

しかし、水準を統合するに特化した `fct_collapse()` を使えばより便利です。使い方は、`fct_recode()` に非常に似ているため省略しますが、=の右側を `c()` でまとめることができます。

```

# fct_collapse() を使った例
df4 <- df4 %>%
  mutate(Q1_R3 = fct_collapse(Q1,
                               そう思う = c("そう思う", "どちらかと言えばそう思う"),
                               どちらとも言えない = "どちらとも言えない",
                               そう思わない = c("どちらかと言えばそう思わない", "そう"),
                               答えたくない = "答えたくない"))

df4

## # A tibble: 10 x 5
##       ID Q1          Q1_R          Q1_R2          Q1_R3
##   <int> <fct>       <fct>       <fct>       <fct>
## 1     1 そう思う      そう思う      そう思う      そ
う思~ そう~
## 2     2 そう思わない  そう思わない  そう思わない そ
う思~ そう~
## 3     3 どちらとも言えない どちらとも言えない どち
ら~ どち~
## 4     4 どちらかと言えばそう思う どちらかと言えばそう思う そ
う思~ そ
う~

```

```

## 5 5 答えたくない 答えたくない 答
えた~ 答え~
## 6 6 どちらかと言えばそう思う どちらかと言えばそう思う そう思
~ そう~
## 7 7 どちらかと言えばそう思わない どちらかと言えばそう思わない そう思
~ そう~
## 8 8 答えたくない 答えたくない 答
えた~ 答え~
## 9 9 そう思わない そう思わない そ
う思~ そう~
## 10 10 そう思う そう思う そ
う思~ そう~

1 levels(df4$Q1_R3)

## [1] "そう思う"          "どちらとも言えない" "そう思わない"
## [4] "答えたくない"

```

`fct_recode()` の結果と同じ結果が得られました。元の水準数や、減らされる水準数などによっては書く手間があまり変わらないので、好きな方を使っても良いでしょう。

### 14.2.11 `fct_drop()`: 使われていない水準を除去する

水準としては存在するものの、データとしては存在しないケースもあります。これをここでは「空水準 (empty levels)」と呼びます。たとえば、以下のコードは `Pref` を factor 化してから `Pref == "奈良県"` のケースを落としたものです。

```

1 Score_df_f2 <- df %>%
2   mutate(Pref = fct_inorder(Pref)) %>%
3   filter(Pref != "奈良県") %>%
4   group_by(Pref) %>%
5   summarise(Score = mean(Score, na.rm = TRUE),
6             .groups = "drop")
7

```

```
8  Score_df_f2

## # A tibble: 8 x 2
##   Pref      Score
##   <fct>    <dbl>
## 1 東京都    3.67
## 2 神奈川県  3.53
## 3 千葉県    3.72
## 4 埼玉県    3.64
## 5 大阪府    3.77
## 6 京都府    3.68
## 7 兵庫県    3.54
## 8 和歌山県  3.97
```

このように結果としては、奈良県のデータを除外したため空水準である奈良県は表示されませんが、Pref変数はどうでしょうか。

```
1  levels(Score_df_f2$Pref)

## [1] "東京都"    "神奈川県"  "千葉県"    "埼玉県"    "大阪府"    "京都府"    "
兵庫県"
## [8] "奈良県"    "和歌山県"
```

このように水準としては残っていることが分かります。使われていない水準が分析や可視化に影響を与えないケースもありますが、与えるケースもあります。これもこれまで勉強してきた `fct_*`() 関数群で対応可能ですが、`fct_drop()` 関数を使えば一発で終ります。実際にやってみましょう。

```
1  Score_df_f2 <- Score_df_f2 %>%
2    mutate(Pref = fct_drop(Pref))

1  levels(Score_df_f2$Pref)

## [1] "東京都"    "神奈川県"  "千葉県"    "埼玉県"    "大阪府"    "京都府"    "
兵庫県"
```

```
## [8] "和歌山県"
```

水準から奈良県が消えました。同じ機能をする関数としては R 内蔵関数である `droplevels()` 関数があり、使い方は `fct_drop()` と同じです。

#### 14.2.12 `fct_expand()`: 水準を追加する

一方、空水準を追加することも可能です。`fct_expand()` 関数には元の変数名に加え、追加する水準名を入れるだけです。たとえば、`df` の `Pref` の水準は関東と関西の 9 都府県名となっていますが、ここに"滋賀県"という水準を追加してみます。。

```
1 df5 <- df %>%
2   mutate(Pref = fct_expand(Pref, "滋賀県"))
3
4 levels(df5$Pref)
```

```
## [1] "京都府"    "兵庫県"    "千葉県"    "和歌山県"    "埼玉県"    "大阪府"
## [7] "奈良県"    "東京都"    "神奈川県"    "滋賀県"
```

"滋賀県"という新しい水準が出来ましたね。ただし、新しく追加された水準は最後の順番になりますので、修正が必要な場合は `fct_relevel()` などを使って適宜修正してください。

新しく水準が追加されることによって、何かの変化はあるでしょうか。まずは都府県ごとに `Score` の平均値とケース数を計算してみましょう。

```
1 df5 %>%
2   group_by(Pref) %>%
3   summarise(Score    = mean(Score, na.rm = TRUE),
4             N        = n(),
5             .groups = "drop")
6
7 ## # A tibble: 9 x 3
8 ##   Pref     Score     N
9 ##   <fct>    <dbl> <int>
```

```
## 1 京都府 3.68 414
## 2 兵庫県 3.54 591
## 3 千葉県 3.72 1000
## 4 和歌山県 3.97 140
## 5 埼玉県 3.64 1000
## 6 大阪府 3.77 1000
## 7 奈良県 3.85 147
## 8 東京都 3.67 1000
## 9 神奈川県 3.53 1000
```

見た目は全く変わらず、滋賀県の行が新しく出来たわけでもありません。`dplyr` の `group_by()` の場合、空水準はグループ化の対象になりません。一方、多くの R 内蔵関数はケースとして存在しなくても計算の対象となります。たとえば、ベクトル内のある値が何個格納されているか確認する `table()` 関数の例を見てみましょう。

```
1 table(df5$Pref)

##
##    京都府    兵庫県    千葉県    和歌山県    埼玉県    大阪府    奈良県    東京都
##    414        591        1000        140        1000        1000        147        1000
##    神奈川県    滋賀県
##    1000        0
```

"滋賀県"という列があり、合致するケースが 0 と表示されます。`group_by()` でも空の水準まで含めて出力する引数 `.drop` があります。デフォルトは `TRUE` ですが、これを `FALSE` に指定してみます。

```
1 df5 %>%
2   group_by(Pref, .drop = FALSE) %>%
3   summarise(Score    = mean(Score, na.rm = TRUE),
4             N        = n(),
5             .groups = "drop")
```

```
## # A tibble: 10 x 3
##   Pref     Score     N
##   <fct>   <dbl> <dbl>
```

```
## <fct> <dbl> <int>
## 1 京都府 3.68 414
## 2 兵庫県 3.54 591
## 3 千葉県 3.72 1000
## 4 和歌山県 3.97 140
## 5 埼玉県 3.64 1000
## 6 大阪府 3.77 1000
## 7 奈良県 3.85 147
## 8 東京都 3.67 1000
## 9 神奈川県 3.53 1000
## 10 滋賀県 NaN 0
```

空水準も出力され、Score の平均値は計算不可 (NaN)、ケース数は 0 という結果が得られました。

### 14.2.13 fct\_explicit\_na(): 欠損値に水準を与える

まずは、実習用データ df6 を作ってみます。X1 は numeric 型変数ですが、これを factor 化します。最初から tibble() 内で factor 化しておいても問題ありませんが、練習だと思ってください。

```
1 df6 <- tibble(
2   ID = 1:10,
3   X1 = c(1, 3, 2, NA, 2, 2, 1, NA, 3, NA)
4 )
5
6 df6 <- df6 %>%
7   mutate(X1 = factor(X1,
8     levels = c(1, 2, 3),
9     labels = c("ラーメン", "うどん", "そば")))
10
11 df6
## # A tibble: 10 x 2
```

```

##      ID X1
##      <int> <fct>
## 1     1 ラーメン
## 2     2 そば
## 3     3 うどん
## 4     4 <NA>
## 5     5 うどん
## 6     6 うどん
## 7     7 ラーメン
## 8     8 <NA>
## 9     9 そば
## 10    10 <NA>

```

それでは X1 をグループ化変数とし、ケース数を計算してみましょう。

```

1 df6 %>%
2   group_by(X1) %>%
3   summarise(N      = n(),
4             .groups = "drop")

```

```

## # A tibble: 4 x 2
##   X1      N
##   <fct> <int>
## 1 ラーメン     2
## 2 うどん     3
## 3 そば      2
## 4 <NA>      3

```

NA もグループ化の対象となります。以下はこの欠損値も一つの水準として指定する方法について紹介します。欠損値を欠損値のままにするケースが多いですが、欠損値が何らかの意味を持つ場合、分析の対象になります。たとえば、多項ロジスティック回帰の応答変数として「分からぬ/答えたくない」を含めたり、「分からぬ/答えたくない」を選択する要因を分析したい場合は、欠損値に値を与える必要があります。なぜなら、一般的な分析において欠損値は分析対象から除外されるからです。

まずは、これまで紹介した関数を使ったやり方から紹介します。

```

1 df6 %>%
2   # まず、X1 を character 型に変換し、X2 という列に保存
3   mutate(X2 = as.character(X1),
4   # X2 が NA なら"欠損値"、それ以外なら元の X2 の値に置換
5   X2 = ifelse(is.na(X2), "欠損値", X2),
6   # X2 を再度 factor 化する
7   X2 = factor(X2,
8   levels = c("ラーメン", "うどん", "そば", "欠損値")))
9
10
11 ## # A tibble: 10 x 3
12 ##   ID     X1     X2
13 ##   <int> <fct> <fct>
14 ## 1     1 ラーメン ラーメン
15 ## 2     2 そば    そば
16 ## 3     3 うどん うどん
17 ## 4     4 <NA>   欠損値
18 ## 5     5 うどん うどん
19 ## 6     6 うどん うどん
20 ## 7     7 ラーメン ラーメン
21 ## 8     8 <NA>   欠損値
22 ## 9     9 そば    そば
23 ## 10    10 <NA>   欠損値

```

X1 を character 型に戻す理由<sup>2)</sup>は、水準にない値が入ると factor 化が解除されるからです。factor 型を character 型に戻さずに df6\$X1 の NA を"欠損値"に置換すると、以下のようになります。

```

1 # df6 の X1 が NA なら"欠損"、それ以外なら元の X1 の値を返す
2 ifelse(is.na(df6$X1), "欠損値", df6$X1)
3
4 ## [1] "1"      "3"      "2"      "欠損値" "2"      "2"      "1"      "

```

<sup>2)</sup> character 型でなく、numeric 型でも出来ます。

欠損値"

```
## [9] "3"      "欠損値"
```

"ラーメン"と"うどん"、"そば"が factor 化前の 1, 2, 3 に戻っただけでなく、NA が"欠損値"という character 型に置換されたため、全体が character 型に変換されました。このように欠損値に水準を与える作業は難しくはありませんが、面倒な作業です。そこで登場する関数が `fct_exlpicit_na()` 関数です。使い方は、元の変数に加え、欠損値の水準名を指定する `na_level` です。

```
1 df6 <- df6 %>%
2   # na_level のデフォルト値は"(Missing)"
3   mutate(X2 = fct_exlpicit_na(X1, na_level = "欠損値"))
4
5 df6
```

```
## # A tibble: 10 x 3
##       ID   X1     X2
##   <int> <fct> <fct>
## 1     1 ラーメン ラーメン
## 2     2 そば   そば
## 3     3 うどん うどん
## 4     4 <NA>   欠損値
## 5     5 うどん うどん
## 6     6 うどん うどん
## 7     7 ラーメン ラーメン
## 8     8 <NA>   欠損値
## 9     9 そば   そば
## 10    10 <NA>   欠損値
```

欠損値が一つの水準になったことが分かります。

```
1 df6 %>%
2   group_by(X2) %>%
3   summarise(N      = n(),
4             .groups = "drop")
```

```
## # A tibble: 4 x 2
##   X2      N
##   <fct>  <int>
## 1 ラーメン    2
## 2 うどん    3
## 3 そば     2
## 4 欠損値    3
```

むろん、`group_by()` を使ってちゃんと出力されます。

---

## 練習問題



## 第 15 章

# 整然データ構造

本章ではグラフの作成に適した形へデータを整形することについて学習します。ただし、これはグラフに限られた話ではありません。作図に適したデータは分析にも適します。

### 15.1 整然データ (tidy data) とは

分析や作図に適したデータの形は整然データ、または簡潔データ (tidy data) と呼ばれます。整然データの概念は tidyverse 世界の産みの親である Hadley Wickham 先生が提唱した概念であり、詳細は Wickham [2014] を参照してください。

整然データは目指す到達点は非常に単純です。それは「データの構造 (structure) と意味 (semantic) を一致させる」ことです。そして、この「意味」を出来る限り小さい単位で分解します。

例えば、3 人で構成されたあるクラス内の被験者に対し、投薬前後に測定した数学成績があるとします。投薬前の成績は "Control"、投薬後の状況を "Treatment" とします。これをまとめたのが表 15.1 です。

また、以上の表は転置も可能であり、以下のように表現することが可能です (表 15.2)。

2 つのデータが持つ情報は全く同じです。これは「同じ意味を持つが、異なる構造を持つ」とも言えます。このような多様性が生じる理由は行と列のあり方が各値を説明するに十分ではないからです。異なるデータ構造として表現される余地があるということです。

表 15.1: Messy Data の例 (1)

Name	Control	Treatment
Hadley	90	90
Song	80	25
Yanai	100	95

表 15.2: Messy Data の例 (2)

Treat	Hadely	Song	Yanai
Control	90	80	100
Treatment	90	25	95

たとえば、表 15.1 の場合、各列は以下のような 3 つの情報があります。

1. Name: 被験者名
2. Control: **投薬前の数学成績**
3. Treatment: **投薬後の数学成績**

このデータの問題は「投薬有無」と「数学成績」が 2 回登場したという点です。1 は問題ありませんが、2 と 3 の値は「投薬有無 × 数学成績」の組み合わせです。一つの変数に 2 つの情報が含まれていますね。これによって、投薬有無を行にしても列にしてもいいわけです。「ならばこっちの方が柔軟だしいいのでは?」と思う方もいるかも知れません。しかし、パソコンはこの曖昧さが嫌いです。なぜなら、人間のような思考ができないからです。データフレームは縦ベクトルの集合であるから、各列には一つの情報のみ格納する必要があります。たとえば、以下のように列を変更するとしましょう。

1. Name: 被験者名
2. Treat: 投薬有無
3. Math\_Score: 数学成績

Treat は投薬前なら"Control"の値を、投薬後なら"Treatment"の値が入ります。Math\_Score には数学成績が入ります。これに則って表に直したのが表 15.3 です。

表 15.3: 整然データの例

Name	Treat	Math_Score
Hadley	Control	90
Hadley	Treatment	90
Song	Control	80
Song	Treatment	25
Yanai	Control	100
Yanai	Treatment	95

表が長くなりましたが、これなら一つの列に 2 つ以上の情報が含まれることはあります。この場合、表 15.1 と表 15.2 のように、行と列を転置することができるでしょうか。

表 15.4: 表  
reftab:tidydata-intro-3 を転置した場合

Name	Hadley	Hadley	Song	Song	Yanai	Yanai
Treat	Control	Treatment	Control	Treatment	Control	Treatment
Math_Score	90	90	80	25	100	95

その結果が表 15.4 ですが、いかがでしょうか。まず、列名が重複している時点でアウトですし、人間が見ても非常に分かりにくい表になりました。また、一つの列に異なるデータ (この場合、character 型と numeric 型) が混在しています。パソコンから見てはわけのわからないデータになったわけです。

ここまで来たら整然データのイメージはある程度掴めたかも知れません。具体的に整然データとは次の 4 つの条件を満たすデータです [Wickham, 2014]。

- 1 つの列は、1 つの変数を表す。
- 1 つの行は、1 つの観測を表す。
- 1 つのセル (特定の列の特定の行) は、1 つの値を表す。
- 1 つの表は、1 つの観測単位 (unit of observation) をもつ (異なる観測単位が混ざっていない)。

以下でも、表 15.1 と表 15.3 を対比しながら、以上の 4 条件をより詳しく説明します。

### 15.1.1 1つの列は、1つの変数を表す

表 15.1 と表 15.3 に含まれる情報は以下の 3 つで共通しています。

1. 被験者名
  2. 投薬有無
  3. 数学成績

これらの情報がそれぞれデータの変数になるわけですが、整然データは一つの列が一つの変数を表します。それではまず、表 15.1 (図 15.1 の左) から考えてみましょう。この図には 3 つの情報が全て含まれています。しかし、数学成績は 2 列に渡って格納されており、「1 列 1 変数」の条件を満たしておりません。一方、表 15.3 (図 15.1 の右) は投薬前後を表す Treat 変数を作成し、その値に応じた数学成績が格納されており、「1 列 1 変数」の条件を満たしています。

雜然データ (messy data):

Name	Control	Treatment
Hadley	90	90
Song	80	25
Yanai	100	95

被験者名 (Subjects) 数学成績 (Math Scores)

処置有無 (Treatment Status):

整然データ (tidy data):

Name	Treat	Math_Score
Hadley	Control	90
Hadley	Treatment	90
Song	Control	80
Song	Treatment	25
Yanai	Control	100
Yanai	Treatment	95

被験者名 (Subjects) 処置有無 (Treatment Status) 数学成績 (Math Scores)

図 15.1: 1 つの列は、1 つの変数を表す

「1列1変数」は整然データの最も基本となる条件であり、整然データ作成の出発点とも言えます。

### 15.1.2 1 つの行は、1 つの観測を表す

図 15.2 の左は一行当たり、いくつの観察が含まれているでしょうか。そのためにはこのデータが何を観察しているかを考える必要があります。このデータは投薬前後の数学成績を観察し、量的に測定したものです。つまり、同じ人に対して 2 回観察を行ったことになります。したがって、投薬前の数学成績と投薬後の数学成績は別の観察であり、図 15.2 の左は 3 行の表ですが、実は 6 回分の観察が含まれていることになります。1 行に 2 つの観察が載っていることですね。

**雑然データ**

Name	Control	Treatment
Hadley	90	90
Song	80	25
Yanai	100	95

Yanaiの投薬前の数学成績&投薬後の数学成績

**整然データ**

Name	Treat	Math_Score
Hadley	Control	90
Hadley	Treatment	90
Song	Control	80
Song	Treatment	25
Yanai	Control	100
Yanai	Treatment	95

Yanaiの投薬後の数学成績

図 15.2: 1 つの行は、1 つの観測を表す

一方、図 15.2 の右は 6 行のデータであり、観察回数とデータの行数が一致しています。つまり、1 行に 1 観察となります。

今回は数学成績しか測っていたいので、簡単な例ですが、実際のデータには曖昧な部分があります。たとえば、投薬によって血圧が変化する可能性があるため、最高血圧もまた投薬前後に測定したとします。それが表 15.5 の左です。

3 人に投薬前後に数学成績と最高血圧を測定した場合の観察回数は何回でしょうか。3 人 × 2 時点 × 2 指標の測定だから 12 回の測定でしょうか。ならば、表 15.5 の右が整然データでしょう。しかし、この場合、1 列 1 変数という条件が満たされなくなります。Value 列には数学成績と血圧が混在しており、2 つの変数になります。ならば、どれも整然データではないということでしょうか。実は整然データは表 15.5 の左です。なぜなら、「1 観察 = 1 値」ではないからです。データにおける観察とは観察単位ごとに測定された**値の集合**

表 15.5: 1行1観察の例

Name	Treat	Math	Blood	Name	Treat	Type	Value
Hadley	Control	90	110	Hadley	Control	Math	90
Hadley	Treatment	90	115	Hadley	Control	Blood	110
Song	Control	80	95	Hadley	Treatment	Math	90
Song	Treatment	25	110	Hadley	Treatment	Blood	115
Yanai	Control	100	100	Song	Control	Math	80
Yanai	Treatment	95	95	Song	Control	Blood	95
				Song	Treatment	Math	25
				Song	Treatment	Blood	110
				Yanai	Control	Math	100
				Yanai	Control	Blood	100
				Yanai	Treatment	Math	95
				Yanai	Treatment	Blood	95

です。観察対象とは人や自治体、企業、国などだけでなく、時間も含まれます。たとえば、人の特徴（性別、身長、所得、政治関心など）を測定しもの、ある日の特徴（気温、株価など）を測定したもの全てが観察です。もちろん、人 × 時間のような組み合わせが観察単位ともなり得ます。この一つ一つの観察単位から得られた値の集合が観察です。表 15.5 の分析単位は「人 × 時間」です。成績や最高血圧は分析単位が持つ特徴や性質であって、分析単位ではありません。

### 15.1.3 1つのセルは、1つの値を表す

この条件に反するケースはあまりないかも知れません。たとえば、「Hadley は処置前後の数学成績が同じだし、一行にまとめよう」という意味で図 15.3 の左のような表を作る方もいるかも知れませんが、あまりいないでしょう。

## 雑然データ

Name	Treat	Math_Score
Hadley	Control, Treatment	90
Song	Control	80
Song	Treatment	25
Yanai	Control	100
Yanai	Treatment	95

図 15.3: 1 つのセルは、1 つの値を表す

図 15.3 の例は「1 セル 1 値」の条件に明らかに反します。しかし、基準が曖昧な変数もあり、その一つが日付です。

表 15.6: 日付の扱い方

Date	Stock	Year	Month	Date	Stock
2020/06/29	100	2020	6	29	100
2020/06/30	105	2020	6	30	105
2020/07/01	110	2020	7	1	110
2020/07/02	85	2020	7	2	85
2020/07/03	90	2020	7	3	90

表 15.6 の左側の表はどうでしょうか。5 日間の株価を記録した架空のデータですが、たしかに Date 列には日付が 1 つずつ、Stock には株価の値が 1 つずつ格納されています。しかし、解釈によっては「Date に年、月、日といった 3 つの値が含まれているぞ」と見ることもできます。この解釈に基づく場合、表 15.6 の右側の表が整然データとなり、左側は雑然データとなります。このケースは第一条件であった「一列一変数」とも関係します。なぜなら、Date という列が年・月・日といった 3 変数で構成されているとも解釈できるからです。

分析によっては左側のような表でも全く問題ないケースもあります。時系列分析でトレ

ンド変数のみ必要ならこれでも十分に整然データと呼べます。しかし、季節変動などの要素も考慮するならば、左側は雑然データになります。データとしての使い勝手は右側の方が優れているのは確かです。

データを出来る限り細かく分解するほど情報量が豊かになりますが、それにも限度はあるでしょう。たとえば、「Year は実は世紀の情報も含まれているのでは...?」という解釈もできますが、これを反映してデータ整形を行うか否かは分析の目的と分析モデルによって異なります。この意味で、明らかな雑然データはあり得ますが、明らかな整然データは存在しないでしょう。どちらかといえば、整然さの度合いがあり、「これなら十分に整然データと言えないだろうか」と判断できれば十分ではないかと筆者 (Song) は考えます。

#### 15.1.4 1つの表は、1つの観測単位をもつ

e-stat などから国勢調査データをダウンロードした経験はあるでしょうか。以下の図 15.4 は 2015 年度国勢調査データの一部です。

都道府県名	都道府県・市区町村名	人口	平成22年	平成22年～27年の	平成22年～27年の	面積
		総数	組替人口	人口増減数	人口増減率	
(人)	(人)	(人)	(%)	(km <sup>2</sup> )		
全国	全国	127,094,745	128,057,352	-962,607	-0.8	377,970.7
北海道	北海道	5,381,733	5,506,419	-124,686	-2.3	83,424.3
北海道	札幌市	1,952,356	1,913,545	38,811	2.0	1,121.2
北海道	札幌市 中央区	237,627	220,189	17,438	7.9	46.4
北海道	札幌市 北区	285,321	278,781	6,540	2.3	63.5
北海道	札幌市 東区	261,912	255,873	6,039	2.4	56.9
北海道	札幌市 白石区	209,584	204,259	5,325	2.6	34.4
北海道	札幌市 豊平区	218,652	212,118	6,534	3.1	46.2
北海道	札幌市 南区	141,190	146,341	-5,151	-3.5	657.4
北海道	札幌市 西区	213,578	211,229	2,349	1.1	75.1
北海道	札幌市 厚別区	127,767	128,492	-725	-0.6	24.3
北海道	札幌市 手稲区	140,999	139,644	1,355	1.0	56.7
北海道	札幌市 清田区	115,726	116,619	-893	-0.8	59.8
北海道	函館市	265,979	279,127	-13,148	-4.7	677.8
北海道	小樽市	121,924	131,928	-10,004	-7.6	243.8
北海道	旭川市	339,605	347,095	-7,490	-2.2	747.6
北海道	室蘭市	88,564	94,535	-5,971	-6.3	80.8
北海道	釧路市	174,742	181,169	-6,427	-3.5	1,362.6

図 15.4: 国勢調査データ

このデータの観察単位はなんでしょうか。データの1行目は全国の人口を表しています。つまり、単位は国となります。しかし、2行目は北海道の人口です。この場合の観測単位は都道府県となります。つづいて、3行目は札幌市なので単位は市区町村になります。4行目は札幌市中央区、つまり観測単位が行政区になっています。そして14行目は函館市でまた単位は市区町村に戻っています。実際、会社や政府が作成するデータには図15.4や図15.5のようなものが多いです。とりわけ、図15.5のように、最後の行に「合計」などが表記されている場合が多いです。

### 雑然データ

Name	Treat	Math_Score
Hadley	Control	90
Hadley	Treatment	90
Song	Control	80
Song	Treatment	25
Yanai	Control	100
Yanai	Treatment	95
Total		80

単位：個人

単位：クラス

図15.5: 1つの表は、1つの観測単位をもつ

このような表・データを作成することが悪いことではありません。むしろ、「読む」ための表ならこのような書き方が一般的でしょう。しかし、「分析」のためのデータは観察の単位を統一する必要があります。

---

### 15.2 Wide 型から Long 型へ

以下では「1列1変数」の条件を満たすデータの作成に便利な `pivot_longer()` と `pivot_wider()` 関数について解説します。この関数群はおなじみの{dplyr}ではなく、{tidyverse}パッケージが提供している関数ですが、どれも{tidyverse}パッケージ群に含まれているため、{tidyverse}パッケージを読み込むだけで十分です。本節では `pivot_longer()` を、次節では `pivot_wider()` を取り上げます。

まず、`pivot_longer()` ですが、この関数は比較的新しい関数であり、これまで `{tidyverse}` の `gather()` 関数が使われてきました。しかし、`gather()` 関数は将来、なくなる予定の関数であり、今から `{tidyverse}` を学習する方は `pivot_*` 関数群に慣れておきましょう。

まずは `{tidyverse}` パッケージを読み込みます。

```
1 pacman::p_load(tidyverse)
```

今回は様々な形のデータを変形する作業をするので、あるデータセットを使うよりも、架空の簡単なデータを使います。

```
1 df1 <- tibble(
2   Name      = c("Hadley", "Song", "Yanai"),
3   Control   = c(90, 80, 100),
4   Treatment = c(90, 25, 95),
5   Gender    = c("Male", "Female", "Female")
6 )
7
8 df1
```

```
## # A tibble: 3 x 4
##   Name   Control Treatment Gender
##   <chr>    <dbl>     <dbl> <chr>
## 1 Hadley     90        90  Male
## 2 Song        80        25 Female
## 3 Yanai      100       95 Female
```

このデータは既に指摘した通り「1列1変数」の条件を満たしております。この条件を満たすデータは以下のようになります。

```
## # A tibble: 6 x 4
##   Name   Gender Treat    Math_Score
##   <chr>  <chr>   <chr>      <dbl>
## 1 Hadley Male   Control      90
## 2 Hadley Male   Treatment   90
```

```
## 3 Song Female Control      80
## 4 Song Female Treatment   25
## 5 Yanai Female Control   100
## 6 Yanai Female Treatment  95
```

`Treat` 変数が作成され、元々は変数名であった"Control"と"Treatment"が値として格納されます。この変数をキー変数と呼びます。そして、キー変数の値に応じた数学成績が `Math_Score` という変数でまとめられました。この変数を値変数と呼びます。

「1列1変数」を満たさなかった最初のデータは「Wide 型データ」、これを満たすようなデータは「Long 型データ」と呼ばれます。これは相対的に最初のデータが横に広いから名付けた名前であって、「Wide 型=雑然データ」もしくは「Long 型=雑然データ」ではないことに注意してください<sup>1)</sup>。

Wide 型データを Long 型へ変換する関数が `pivot_longer()` であり、基本的な使い方は以下の通りです。

```
1 # pivot_longer() の使い方
2 データ名 %>%
3   pivot_longer(cols      = c(まとめる変数 1, まとめる変数 2, ...),
4                 names_to  = "キー変数名",
5                 values_to = "値変数名")
```

ここでは同じ変数が `Control` と `Treatment` 変数で分けられているため、まとめる変数はこの 2 つであり、`cols = c(Control, Treatment)` と指定します。`Control` と `Treatment` は"で囲んでも、囲まなくても同じです。また、`{dplyr}` の `select()` 関数で使える変数選択の関数 (`starts_with()`、`where()` など) や: 演算子も使用可能です。また、`cols` 引数は `pivot_longer()` の第 2 引数であるため、`cols =` は省略可能です（第一引数はパイプにより既に渡されています）。

`names_to` と `values_to` 引数はそれぞれキー変数名と値変数名を指定する引数で、ここは必ず"で囲んでください。この `df1` を Long 型へ変換し、`df1_L` と名付けるコードが以下のコードです。

---

<sup>1)</sup> たとえば、表 15.5 は右の方が Long 型データですが、整然データは Wide 型である左の方ですね。

```
1 df1_L <- df1 %>%
2   pivot_longer(Control:Treatment,
3     names_to = "Treat",
4     values_to = "Math_Score")
5
6 df1_L
```

```
## # A tibble: 6 x 4
##   Name   Gender Treat     Math_Score
##   <chr>  <chr>   <chr>     <dbl>
## 1 Hadley  Male    Control     90
## 2 Hadley  Male    Treatment   90
## 3 Song    Female   Control    80
## 4 Song    Female   Treatment   25
## 5 Yanai   Female   Control   100
## 6 Yanai   Female   Treatment  95
```

これだけでも `pivot_longer()` 関数を使って Wide 型から Long 型への変換は問題なくできますが、以下ではもうちょっと踏み込んだ使い方について解説します。「ここまで十分だよ」という方は、ここを飛ばしても構いません。

今回の実習データ `df3` は 3 人の体重を 3 日間に渡って計測したものです。ただし、ドジっ子の `Song` は 2 日目にうっかり測るのを忘れており、欠損値となっています。

```
1 df2 <- tibble(
2   Name = c("Hadley", "Song", "Yanai"),
3   Day1 = c(75, 120, 70),
4   Day2 = c(73, NA, 69),
5   Day3 = c(71, 140, 71)
6 )
7
8 df2
```

```
## # A tibble: 3 x 4
```

```
##   Name   Day1  Day2  Day3
##   <chr> <dbl> <dbl> <dbl>
## 1 Hadley    75    73    71
## 2 Song      120    NA   140
## 3 Yanai     70    69    71
```

まず、これをこれまでのやり方で Long 型へ変形し、df2\_L と名付けます。

```
1 df2_L <- df2 %>%
2   pivot_longer(starts_with("Day"),
3                 names_to = "Days",
4                 values_to = "Weight")
5
6 df2_L
```

```
## # A tibble: 9 x 3
##   Name   Days  Weight
##   <chr> <chr> <dbl>
## 1 Hadley Day1    75
## 2 Hadley Day2    73
## 3 Hadley Day3    71
## 4 Song   Day1   120
## 5 Song   Day2    NA
## 6 Song   Day3   140
## 7 Yanai  Day1    70
## 8 Yanai  Day2    69
## 9 Yanai  Day3    71
```

これでも問題ないかも知れませんが、以下のような操作を追加に行うとします。

1. Weight が欠損している行を除去する
2. Days 列の値から "Day" を除去し、numeric 型にする

以上の作業を行うには、dplyr が便利でしょう。ちなみに str\_remove() 関数が初めて登場しましたが、これについては第 16 章で詳細に解説します。簡単に説明しますと、

`str_remove("X123", "X")` は "X123" から "X" を除去し、"123" のみ残す関数です。残された値が数字のみであってもデータ型は character 型なので、もう一回、numeric 型に変換する必要があります<sup>2)</sup>。`dplyr` を使ったコードは以下の通りです。

```

1 # 1. Weight が NA のケースを除去
2 # 2. Days 変数の値から "Day" を除去
3 # 3. Days 変数を numeric 型へ変換
4 df2_L %>%
5   filter(!is.na(Weight)) %>% # 1
6   mutate(Days = str_remove(Days, "Day")), # 2
7   Days = as.numeric(Days)) # 3

## # A tibble: 8 x 3
##   Name    Days Weight
##   <chr>  <dbl>  <dbl>
## 1 Hadley     1     75
## 2 Hadley     2     73
## 3 Hadley     3     71
## 4 Song       1    120
## 5 Song       3    140
## 6 Yanai      1     70
## 7 Yanai      2     69
## 8 Yanai      3     71

```

実はこの作業、`pivot_longer()` 内で行うことも可能です。たとえば、`values_to` で指定した変数の値が欠損しているケースを除去するには `values_drop_na` 引数を `TRUE` に指定するだけです。

```

1 # Weight 変数が NA のケースを除去する
2 df2 %>%
3   pivot_longer(starts_with("Day"),
4                 names_to      = "Days",

```

---

<sup>2)</sup> 実は `parse_number()` を使えばもっと簡単ですが、これについては後ほど解説します。

```
5         values_to      = "Weight",
6         values_drop_na = TRUE)

## # A tibble: 8 x 3
##   Name   Days  Weight
##   <chr> <chr> <dbl>
## 1 Hadley Day1     75
## 2 Hadley Day2     73
## 3 Hadley Day3     71
## 4 Song    Day1    120
## 5 Song    Day3    140
## 6 Yanai   Day1     70
## 7 Yanai   Day2     69
## 8 Yanai   Day3     71
```

それでは、キー変数から共通する文字列を除去するにはどうすれば良いでしょうか。この場合、`names_prefix`引数を使います。これは`names_to`で指定した新しく出来る変数の値における接頭詞を指定し、それを除去する引数です。今回は"Day1"、"Day2"、"Day3"から"Day"を除去するので、`names_prefix = "Day"`と指定します。こうすることで、`Days`列から"Day"が除去されます。ただし、数字だけ残っても、そのデータ型は`character`型ですので、このデータ型を変換する必要があります。ここで使うのが`names_transform`引数であり、これは`list`型のオブジェクトを渡す必要があります。`Days`列を`numeric`型にする場合は`list(Days = as.numeric)`です。複数の列のデータ型を変える場合、`list()`の中に追加していきます。それでは実際に走らせてみましょう。

```
1 # 1. Day 变数の値から"Day"を除去する
2 # 2. Day 变数を integer 型に変換
3 # 3. Weight 变数が NA のケースを除去する
4 df2 %>%
5   pivot_longer(starts_with("Day"),
6                 names_to      = "Days",
7                 names_prefix   = "Day",                      # 1
8                 names_transform = list(Days = as.numeric), # 2
```

```
9         values_to      = "Weight",
10        values_drop_na = TRUE) # 3

## # A tibble: 8 x 3
##   Name    Days  Weight
##   <chr> <dbl>  <dbl>
## 1 Hadley     1     75
## 2 Hadley     2     73
## 3 Hadley     3     71
## 4 Song       1    120
## 5 Song       3    140
## 6 Yanai      1     70
## 7 Yanai      2     69
## 8 Yanai      3     71
```

これで Wide 型を Long 型が変換され、整然でありながら、より見栄の良いデータが出来上りました。他にも `pivot_longer()` は様々な引数に対応しており、詳細は `?pivot_longer` やレファレンスページを参照してください。

---

### 15.3 Long 型から Wide 型へ

ご存知の通り、「Long 型データ=整然データ」ではありません。実際、表 15.5 の右は Long 型データですが、1 列に 2 つの変数が含まれており、整然データとは言えません。このようなデータはいくらでもあります。とりわけ、「分析」のためじゃなく、「読む」ための表の場合において多く発見されます。

変数名が日本語になっていますが、これは「読むための表」を読み込むことを仮定しています。このように変数名として日本語は使えますが、自分でデータセットを作成する際、変数名はローマ字にすることを強く推奨します。

表 15.7 の左の場合、人口列に総人口と外国人人口といった 2 つの変数の値が格納されているため、整然データではありません。これを整然データにしたもののが右の表です。本

表 15.7: Long 型データの例

都道府県	区分	人口	面積
北海道	総人口	5381733	83424.31
	外国人	21676	83424.31
青森県	総人口	1308265	9645.59
	外国人	3447	9645.59
岩手県	総人口	1279594	15275.01
	外国人	5017	15275.01
宮城県	総人口	2333899	7282.22
	外国人	13989	7282.22

都道府県	総人口	外国人	面積
北海道	5381733	21676	83424.31
青森県	1308265	3447	9645.59
岩手県	1279594	5017	15275.01
宮城県	2333899	13989	7282.22

節では Long 型データを Wide 型データへ変換する `pivot_wider()` 関数を紹介します。この関数は同じく `{tidyverse}` が提供している `spread()` 関数とほぼ同じ関数ですが、今は `pivot_wider()` の使用が推奨されており、`spread()` はいずれか `{tidyverse}` から外される予定です。

まずは、実習用データを読み込みます。

```
1 df3 <- read.csv("Data/Population2015.csv")
```

```
## Rows: 94 Columns: 4
## -- Column specification:
## Delimiter: ","
## chr (2): 都道府県, 区分
## dbl (2): 人口, 面積
##
```

```

## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message
1 df3

## # A tibble: 94 x 4
##   都道府県 区分     人口   面積
##   <chr>    <chr>   <dbl>   <dbl>
## 1 北海道    総人口 5381733 83424.
## 2 <NA>      外国人  21676 83424.
## 3 青森県    総人口 1308265 9646.
## 4 <NA>      外国人  3447 9646.
## 5 岩手県    総人口 1279594 15275.
## 6 <NA>      外国人  5017 15275.
## 7 宮城県    総人口 2333899 7282.
## 8 <NA>      外国人  13989 7282.
## 9 秋田県    総人口 1023119 11638.
## 10 <NA>     外国人  2914 11638.
## # ... with 84 more rows

```

このデータは2015年国勢調査から抜粋したデータであり、各変数の詳細は以下の通りです。

変数名	説明
都道府県	都道府県名
区分	総人口/外国人人口の区分
人口	人口(人)
面積	面積(km\$^2\$)

まずは変数名が日本語になっているので、`rename()`関数を使ってそれぞれPref、Type、Population、Areaに変更します。

```

1 df3 <- df3 %>%
2   rename("Pref" = 都道府県, "Type" = 区分, "Population" = 人口, "Area" = 面積)

```

```
3
4 df3

## # A tibble: 94 x 4
##   Pref   Type   Population   Area
##   <chr>  <chr>     <dbl>     <dbl>
## 1 北海道 総人口    5381733  83424.
## 2 <NA>   外国人     21676  83424.
## 3 青森県 総人口    1308265  9646.
## 4 <NA>   外国人     3447   9646.
## 5 岩手県 総人口    1279594  15275.
## 6 <NA>   外国人     5017   15275.
## 7 宮城県 総人口    2333899  7282.
## 8 <NA>   外国人     13989   7282.
## 9 秋田県 総人口    1023119  11638.
## 10 <NA>  外国人     2914   11638.
## # ... with 84 more rows
```

次は、Pref列の欠損値を埋めましょう。この欠損値は、当該セルの一つ上のセルの値で埋まりますが、これはfill()関数で簡単に処理できます。欠損値を埋めたい変数名をfill()の引数として渡すだけです。

```
1 df3 <- df3 %>%
2   fill(Pref)
3
4 df3

## # A tibble: 94 x 4
##   Pref   Type   Population   Area
##   <chr>  <chr>     <dbl>     <dbl>
## 1 北海道 総人口    5381733  83424.
## 2 北海道 外国人     21676  83424.
## 3 青森県 総人口    1308265  9646.
## 4 青森県 外国人     3447   9646.
```

```

## 5 岩手県 総人口      1279594 15275.
## 6 岩手県 外国人       5017 15275.
## 7 宮城県 総人口      2333899 7282.
## 8 宮城県 外国人       13989 7282.
## 9 秋田県 総人口      1023119 11638.
## 10 秋田県 外国人      2914 11638.
## # ... with 84 more rows

```

そして、いよいよ `pivot_wider()` 関数の出番ですが、基本的に使い方は以下の通りです。

```

1 # pivot_wider() の使い方
2 データ名 %>%
3   pivot_wider(names_from = キー変数名,
4               values_from = 値変数名)

```

まず、キー変数名は列として展開する変数名であり、ここでは `Type` になります。そして、値変数名は展開される値の変数であり、ここでは `Population` になります。つまり、「`Population` を `Type` ごとに分けて別の列にする」ことになります。また、`values_from` 引数は長さ 2 以上のベクトルを指定することで、複数の値変数を指定することも可能です。たとえば、`df3` に `Income` という平均所得を表す列があり、これらも総人口と外国人それぞれ異なる値を持っているとしたら、`values_from = c(Population, Income)` のように複数の値変数を指定することが出来ます。今回は値変数が 1 つのみですが、早速やってみましょう。

```

1 df3_W <- df3 %>%
2   pivot_wider(names_from = Type,
3               values_from = Population)
4
5 df3_W

```

```

## # A tibble: 47 x 4
##       Pref     Area 総人口 外国人
##       <chr>    <dbl>   <dbl>   <dbl>
## 1 北海道 83424. 5381733 21676

```

```
## 2 青森県 9646. 1308265 3447
## 3 岩手県 15275. 1279594 5017
## 4 宮城県 7282. 2333899 13989
## 5 秋田県 11638. 1023119 2914
## 6 山形県 9323. 1123891 5503
## 7 福島県 13784. 1914039 8725
## 8 茨城県 6097. 2916976 41310
## 9 栃木県 6408. 1974255 26494
## 10 群馬県 6362. 1973115 37126
## # ... with 37 more rows
```

また、日本語の変数名が出来てしまったので、それぞれ Total と Foreigner に変更し、`relocate()` 関数を使って Area を最後の列に移動します。

```
1 df3_W <- df3_W %>%
2   rename("Total" = 総人口,
3         "Foreigner" = 外国人) %>%
4   relocate(Area, .after = last_col())
5
6 df3_W
```

```
## # A tibble: 47 x 4
##       Pref     Total Foreigner     Area
##       <chr>    <dbl>    <dbl>    <dbl>
## 1 北海道  5381733    21676  83424.
## 2 青森県  1308265    3447   9646.
## 3 岩手県  1279594    5017   15275.
## 4 宮城県  2333899    13989   7282.
## 5 秋田県  1023119    2914   11638.
## 6 山形県  1123891    5503   9323.
## 7 福島県  1914039    8725   13784.
## 8 茨城県  2916976    41310   6097.
## 9 栃木県  1974255    26494   6408.
## 10 群馬県 1973115    37126   6362.
```

```
## # ... with 37 more rows
```

これで整然データの出来上がりです。

この `pivot_wider()` 関数は `pivot_longer()` 関数同様、様々な引数を提供しておりますが、主に使う機能は以上です。他には `pivot_wider()` によって出来た欠損値を埋める引数である `values_fill` があり、デフォルト値は `NULL` です。ここに `0` や "Missing" などの長さ 1 のベクトルを指定すれば、指定した値で欠損値が埋まります。

`pivot_wider()` 関数の詳細は `?pivot_wider` もしくは、レファレンスページを参照してください。

---

## 15.4 列の操作

他にも「1列1変数」の条件を満たさないケースを考えましょう。`pivot_longer()` は1つの変数が複数の列に渡って格納されている際に使いましたが、今回は1つの列に複数の変数があるケースを考えてみましょう。たとえば、年月日が1つの列に入っている場合、これを年、月、日の3列で分割する作業です。また、これと関連して、列から文字列を除去し、数値のみ残す方法についても紹介します。

実習用データを読み込んでみましょう。

```
1 df4 <- read_csv("Data/COVID19_JK.csv")
2
3 df4
## # A tibble: 172 x 5
##       ID Date      Week Confirmed_Japan Confirmed_Korea
##   <dbl> <chr>    <chr>    <chr>           <chr>
## 1     1 2020/1/16 木     1人             <NA>
## 2     2 2020/1/17 金     0人             <NA>
## 3     3 2020/1/18 土     0人             <NA>
## 4     4 2020/1/19 日     0人             <NA>
```

```
## 5 5 2020/1/20 月 0 人 1 人
## 6 6 2020/1/21 火 0 人 0 人
## 7 7 2020/1/22 水 0 人 0 人
## 8 8 2020/1/23 木 0 人 0 人
## 9 9 2020/1/24 金 2 人 1 人
## 10 10 2020/1/25 土 0 人 0 人
## # ... with 162 more rows
```

このデータは 2020 年 1 月 16 日から 2020 年 7 月 5 日まで、COVID-19 (新型コロナ) の新規感染者数を日本と韓国を対象に収集したものです。データは Wikipedia (日本 / 韓国) から収集しました。韓国的新規感染者数は最初の 4 日分が欠損値のように見えますが、最初の感染者が確認されたのが 1 月 20 日のため、1 月 19 日までは欠損となっています。

変数名	説明
ID	ケース ID
Date	年月日
Week	曜日
Confirmed_Japan	新規感染者数 (日本)
Confirmed_Korea	新規感染者数 (韓国)

このデータの場合、観察単位は「国 × 日」です。しかし、df4 は 1 行に日本と韓国の情報が格納されており「1 行 1 観察」の条件を満たしておりません。したがって、pivot\_longer() を使って Long 型へ変換し、新しいデータの名前を df4\_L と名付けます。

```
1 df4_L <- df4 %>%
2   pivot_longer(cols      = starts_with("Confirmed"),
3                 names_to     = "Country",
4                 names_prefix = "Confirmed_",
5                 values_to    = "Confirmed")
6
7 df4_L
```

```
## # A tibble: 344 x 5
##   ID Date     Week Country Confirmed
##   <dbl> <chr>   <chr> <chr>   <chr>
## 1 1 2020/1/16 木 Japan  1人
## 2 2 2020/1/16 木 Korea  <NA>
## 3 3 2020/1/17 金 Japan  0人
## 4 4 2020/1/17 金 Korea  <NA>
## 5 5 2020/1/18 土 Japan  0人
## 6 6 2020/1/18 土 Korea  <NA>
## 7 7 2020/1/19 日 Japan  0人
## 8 8 2020/1/19 日 Korea  <NA>
## 9 9 2020/1/20 月 Japan  0人
## 10 10 2020/1/20 月 Korea  1人
## # ... with 334 more rows
```

続いて、新規感染者数を表す `Confirmed` 列から「人」を除去しましょう。人間にとてはなんの問題もありませんが、パソコンにとって1人や5人は文字列に過ぎず、分析ができる状態ではありません。ここで使う関数が `parse_number()` です。引数として指定した列から数値のみ抽出します。"\$1000"や"1, 324, 392"のような数値でありながら、character型として保存されている列から数値のみを取り出す際に使う関数です。使い方は以下の通りです。

```
1 データ名 %>%
2   mutate(新しい変数名 = parse_number(数値のみ抽出する変数名))
```

似たようなものとして `parse_character()` があり、これは逆に文字列のみ抽出する関数です。

ここでは `Confirmed` から数値のみ取り出し、`Confirmed` 列に上書きし、それを `df4_S` と名付けます。

```
1 df4_S <- df4_L %>%
2   mutate(Confirmed = parse_number(Confirmed))
3
```

```
4 df4_S
```

```
## # A tibble: 344 x 5
##       ID Date     Week Country Confirmed
##   <dbl> <chr>   <chr> <chr>     <dbl>
## 1     1 2020/1/16 木   Japan      1
## 2     1 2020/1/16 木   Korea      NA
## 3     2 2020/1/17 金   Japan      0
## 4     2 2020/1/17 金   Korea      NA
## 5     3 2020/1/18 土   Japan      0
## 6     3 2020/1/18 土   Korea      NA
## 7     4 2020/1/19 日   Japan      0
## 8     4 2020/1/19 日   Korea      NA
## 9     5 2020/1/20 月   Japan      0
## 10    5 2020/1/20 月   Korea      1
## # ... with 334 more rows
```

それでは国、曜日ごとの新規感染者数を調べてみます。求める統計量は曜日ごとの新規感染者数の合計、平均、標準偏差です。まず、曜日は月から日の順になるよう、factor型に変換します。そして、国と曜日ごとに記述統計量を計算し、df4\_S\_Summary1 という名で保存します。

```
1 df4_S <- df4_S %>%
2   mutate(Week = factor(Week,
3                     levels = c("月", "火", "水", "木", "金", "土", "日")))
4
5 df4_S_Summary1 <- df4_S %>%
6   group_by(Country, Week) %>%
7   summarise(Sum      = sum(Confirmed, na.rm = TRUE),
8             Mean     = mean(Confirmed, na.rm = TRUE),
9             SD       = sd(Confirmed, na.rm = TRUE),
10            .groups = "drop")
```

```
12 df4_S_Summary1

## # A tibble: 14 x 5
##   Country Week   Sum  Mean   SD
##   <chr>   <fct> <dbl> <dbl> <dbl>
## 1 Japan    月     2540 106.  156.
## 2 Japan    火     2093 87.2  111.
## 3 Japan    水     2531 105.  133.
## 4 Japan    木     2704 108.  151.
## 5 Japan    金     3083 123.  172.
## 6 Japan    土     3327 133.  189.
## 7 Japan    日     3244 130.  179.
## 8 Korea   月     1609 67.0  126.
## 9 Korea   火     1641 68.4  111.
## 10 Korea  水     1626 67.8  102.
## 11 Korea  木     1883 78.5  137.
## 12 Korea  金     2099 87.5  143.
## 13 Korea  土     2194 91.4  174.
## 14 Korea  日     2088 87    215.
```

df4\_S\_Summary1 はこの状態で整然データですが、もし人間が読むための表を作るなら、韓国と日本を別の列に分けた方が良いかも知れません。pivot\_wider() を使って、日本と韓国の新規感染者数を 2 列に展開します。

```
1 df4_S_Summary1 %>%
2   pivot_wider(names_from = Country,
3               values_from = Sum:SD)

## # A tibble: 7 x 7
##   Week   Sum_Japan Sum_Korea Mean_Japan Mean_Korea SD_Japan SD_Korea
##   <fct> <dbl>     <dbl>      <dbl>      <dbl>     <dbl>     <dbl>
## 1 月      2540      1609      106.       67.0      156.      126.
## 2 火      2093      1641      87.2      68.4      111.      111.
```

```
## 3 水      2531    1626    105.     67.8    133.    102.
## 4 木      2704    1883    108.     78.5    151.    137.
## 5 金      3083    2099    123.     87.5    172.    143.
## 6 土      3327    2194    133.     91.4    189.    174.
## 7 日      3244    2088    130.     87      179.    215.
```

これで人間にとて読みやすい表が出来ました。今は「日本の合計」、「韓国の合計」、「日本の平均」、…の順番ですが、これを日本と韓国それぞれまとめる場合は、`relocate()` を使います。

```
1 df4_S_Summary1 %>%
2   pivot_wider(names_from = Country,
3               values_from = Sum:SD) %>%
4   relocate(Week, ends_with("Japan"), ends_with("Korea"))

## # A tibble: 7 x 7
##   Week   Sum_Japan Mean_Japan SD_Japan Sum_Korea Mean_Korea SD_Korea
##   <fct>    <dbl>     <dbl>     <dbl>    <dbl>     <dbl>     <dbl>
## 1 1 月      2540      106.      156.     1609      67.0      126.
## 2 2 火      2093      87.2      111.     1641      68.4      111.
## 3 3 水      2531      105.      133.     1626      67.8      102.
## 4 4 木      2704      108.      151.     1883      78.5      137.
## 5 5 金      3083      123.      172.     2099      87.5      143.
## 6 6 土      3327      133.      189.     2194      91.4      174.
## 7 7 日      3244      130.      179.     2088      87       215.
```

新規感染者が確認されるのは金～日曜日が多いことが分かります。

曜日ではなく、月ごとに記述統計料を計算する場合は、まず `Date` 列を年、月、日に分割する必要があります。具体的には `Date` を"/"を基準に別ければいいです。そこで登場するのは `separate()` 関数であり、使い方は以下の通りです。

```
1 # separate() の使い方
2 データ名 %>%
3   separate(cols = 分割する変数名)
```

```

4     into  = 分割後の変数名,
5     sep   = "分割する基準")

```

cols には Date を指定し、into は新しく出来る列名を指定します。今回は Date が 3 列に分割されるので、長さ 3 の character 型ベクトルを指定します。ここでは Year、Month、Day としましょう。最後の sep 引数は分割する基準となる文字を指定します。df4 の Date は"2020/06/29"のように年月日が"/"で分けられているため、"/"を指定します。実際にやってみましょう。

```

1 df4_S <- df4_S %>%
2   separate(col = Date, into = c("Year", "Month", "Day"), sep = "/")
3
4 df4_S

```

```

## # A tibble: 344 x 7
##       ID Year Month Day Week Country Confirmed
##   <dbl> <chr> <chr> <chr> <fct> <chr>     <dbl>
## 1     1 2020  1     16   木   Japan      1
## 2     2 2020  1     16   木   Korea      NA
## 3     3 2020  1     17   金   Japan      0
## 4     4 2020  1     17   金   Korea      NA
## 5     5 2020  1     18   土   Japan      0
## 6     6 2020  1     18   土   Korea      NA
## 7     7 2020  1     19   日   Japan      0
## 8     8 2020  1     19   日   Korea      NA
## 9     9 2020  1     20   月   Japan      0
## 10   10 2020  1     20   月   Korea      1
## # ... with 334 more rows

```

新しく出来た変数は元の変数があった場所になります。ここまで来たら月ごとに新規感染者の記述統計量は計算できます。曜日ごとに行ったコードの Week を Month に変えるだけです。また、Month は数字のみで構成された character 型であるため、このままで問題なくソートされます。したがって、別途 factor 化の必要もありません（むろん、し

てもいいですし、むしろ推奨されます)。

```
1 df4_S_Summary2 <- df4_S %>%
2   group_by(Country, Month) %>%
3   summarise(Sum      = sum(Confirmed, na.rm = TRUE),
4             Mean     = mean(Confirmed, na.rm = TRUE),
5             SD       = sd(Confirmed, na.rm = TRUE),
6             .groups = "drop")
7
8 df4_S_Summary2
```

```
## # A tibble: 14 x 5
##   Country Month   Sum     Mean     SD
##   <chr>    <chr> <dbl>   <dbl>   <dbl>
## 1 Japan     1     17     1.06    1.34
## 2 Japan     2    213     7.34    7.81
## 3 Japan     3   1723    55.6    42.4
## 4 Japan     4  12135   404.     146.
## 5 Japan     5   2763    89.1    73.0
## 6 Japan     6   1742    58.1    23.0
## 7 Japan     7   929     186.    45.1
## 8 Korea     1     11     0.917   1.44
## 9 Korea     2   3139    108.    203.
## 10 Korea    3   6737    217.    222.
## 11 Korea    4    887    29.6    26.6
## 12 Korea    5    729    23.5    16.2
## 13 Korea    6   1348    44.9    10.4
## 14 Korea    7    289    57.8    6.61
```

```
1 df4_S_Summary2 %>%
2   pivot_wider(names_from = Country,
3               values_from = Sum:SD) %>%
4   relocate(Month, ends_with("Japan"), ends_with("Korea"))
```

```
## # A tibble: 7 x 7
##   Month Sum_Japan Mean_Japan SD_Japan Sum_Korea Mean_Korea SD_Korea
##   <chr>     <dbl>      <dbl>     <dbl>      <dbl>      <dbl>     <dbl>
## 1 1          17       1.06      1.34       11       0.917     1.44
## 2 2          213      7.34      7.81      3139      108.     203.
## 3 3         1723      55.6     42.4       6737      217.     222.
## 4 4        12135     404.      146.       887      29.6      26.6
## 5 5         2763      89.1      73.0       729      23.5      16.2
## 6 6         1742      58.1      23.0      1348      44.9      10.4
## 7 7          929      186.      45.1       289      57.8      6.61
```

平均値から見ると、日本は7都道府県を対象に緊急事態宣言が行われた4月がピークで緩やかに減少していますが、7月になって上がり気味です。韓国はカルト宗教団体におけるクラスターが発生した3月がピークで、6月からまた上がり気味ですね。傾向としては韓国が日本に1ヶ月先行しているように見えます。

それでは `separate()` 関数の他の引数についても簡単に紹介します。まず、`sep` 引数は `numeric` 型でも可能です。この場合、文字列内の位置を基準に分割されます。年月日が `20200629` のように保存されている場合は、何らかの基準となる文字がありません。この場合、`sep = c(4, 6)` にすると、「`20200629`」の4文字目と5文字目の間で分割、6文字目と7文字目の間で分割となります。また、`sep = c(-4, -2)` のように負の値も指定可能であり、この場合は右からの位置順で分割します。

また、`separate()` 後は元の変数がなくなりますが、`remove = FALSE` の場合、元の変数（ここでは `Date`）が残ります。他にも `convert` 引数もあります。`convert = TRUE` の場合、適切なデータ型へ変換してくれます。デフォルト値は `FALSE` であり、この場合、`character` 型として分割されます。先ほどの例だと `Year` も `Month` も `Day` も現在は `character` 型です。`separate()` 内で `convert = TRUE` を追加すると、分割後の `Year`、`Month`、`Day` は `numeric` 型として保存されます。

`separate()` の詳細は `?separate` または、レファレンスページを参照してください。

## 第 16 章

### 文字列の処理



第 IV 部

可視化



## 第 17 章

# 可視化 [理論]

### 17.1 可視化のためのパッケージ

R による可視化は様々な方法がありますが、可視化のために使うパッケージとして代表的なものは (1) パッケージを使わない方法、(2) `{lattice}` パッケージ、(3) `{ggplot2}` パッケージがあります。ここでは、以下のデータ (表 17.1) を可視化しながら、それぞれの特徴について簡単に解説します。

表 17.1: サンプルデータの最初の 10 行

Country	PPP	HDI	OECD
Afghanistan	2125	0.496	非加盟国
Albania	13781	0.791	非加盟国
Algeria	11324	0.759	非加盟国
Angola	6649	0.574	非加盟国
Argentina	22938	0.830	非加盟国
Armenia	12974	0.760	非加盟国
Australia	50001	0.938	加盟国
Austria	55824	0.914	加盟国
Azerbaijan	14257	0.754	非加盟国
Bahrain	43624	0.838	非加盟国

このデータは各国 (Country) の一人当たり購買力平価基準 GDP (PPP)、人間開発指数 (HDI)、OECD 加盟有無 (OECD) の変数で構成されています。このデータを使って横軸は PPP、縦軸は HDI とし、OECD の値によって色分けしたグラフを作成します。

### 17.1.1 Base R

R は統計学、データ分析に特化したプログラミング言語であるため、別途のパッケージなしで作図が可能です。後ほど紹介する{lattice}や{ggplot2}を用いた作図とは違って、Base R による作図は、紙にペンでグラフを書くイメージに近いです。キャンバスを用意し、そこにペンで点や線を描く感じです。これはレイヤーという概念を導入した{ggplot2}に近いかも知れませんが、{ggplot2}はレイヤーを変更出来る一方、Base R は図が気に入らない場合、一からやり直します。つまり、キャンバスに引いた線や点は消すことが出来ません。また、作成した図はオブジェクトとして保存することが出来ないため、もう一度図示するためには一から書く必要があります。Base R による作図は短所だけでなく、メリットもあります。まず、パッケージを必要としないため、R インストール直後から使える点です。そして、{lattice}や{ggplot2}よりも速いです。他にも人によっては Base R の方がシンプルでかっこいいという方もいますが、これは好みの問題でしょう。

以下の図 17.1 は Base R を使った散布図の例です。

```
1 # Base R を用いた作図の例
2 plot(x = Country_df$PPP, y = Country_df$HDI, pch = 19,
3       col = ifelse(Country_df$OECD == "加盟国", "red", "blue"),
4       xlab = "一人当たり購買力平価 GDP (USD)", ylab = "人間開発指数")
5 legend("bottomright", pch = 19,
6       legend = c("OECD 加盟国", "OECD 非加盟国"),
7       col      = c("red", "blue"))
```

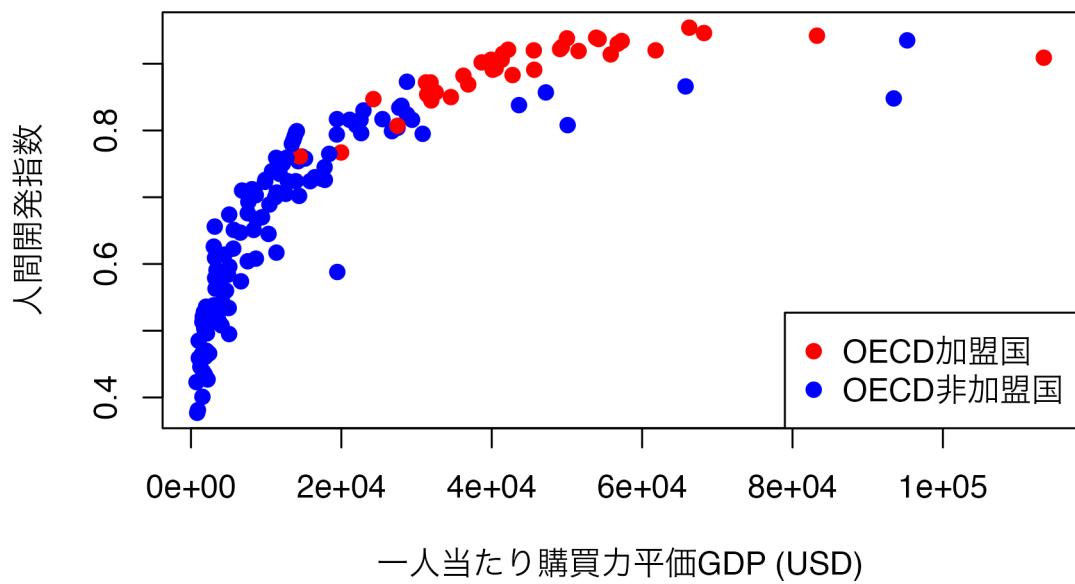


図 17.1: Base R によるグラフ

### 17.1.2 {lattice}パッケージ

{lattice}は Deepayan Sarkar によって開発された可視化パッケージです。このパッケージの最大特徴は「1つの関数で可視化が出来る」点です。作図に必要な様々な情報が1つの関数内に全て入ります。もちろん、指定しない情報に関しては多くの場合、自動的に処理してくれます。

図 17.1 を{lattice}を使って作る場合は以下のようないいコードになります。

```

1 # lattice を用いた作図の例
2 xyplot(HDI ~ PPP, data = Country_df,
3         group = OECD, pch = 19, grid = TRUE,
4         auto.key = TRUE,
5         key = list(title      = "OECD 加盟有無",
6                    cex.title = 1,
7                    space      = "right",
8                    points     = list(col = c("magenta", "cyan"),
9                                     pch = 19)),

```

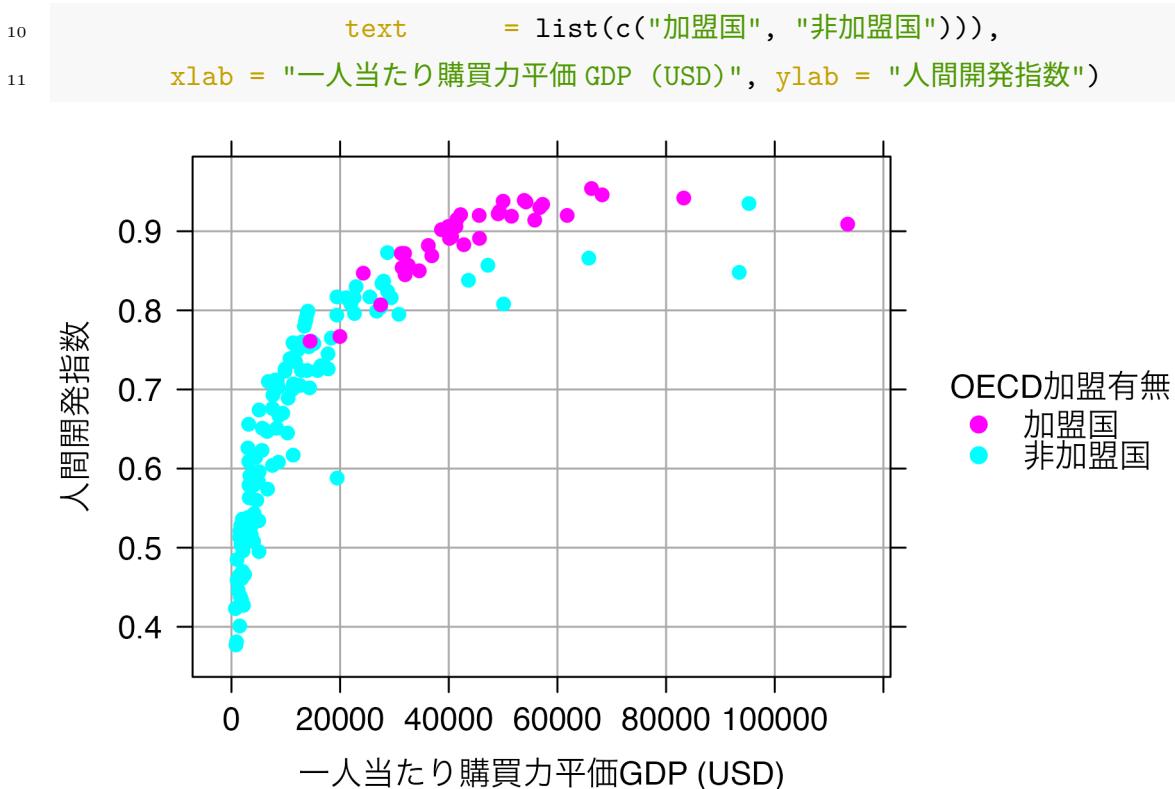


図 17.2: lattice によるグラフ

1つの関数で全てを処理するので、関数が非常に長くなり、人間にとって読みやすいコードにはなりにくいのが短所です。しかし、`{lattice}`はBase Rでは出来ない、プロットのオブジェクトとしての保存ができます。Base Rは出来上がったプロットをオブジェクトとして保存することが出来ず、同じ図をもう一回出力するためには、改めてコードを書く必要があります。しかし、`{lattice}`はオブジェクトとして保存ができるため、いつでもリサイクルが可能です。他にも、`{lattice}`は条件付きプロットの作成において非常に強力です。しかし、これらの特徴は今は`{ggplot2}`も共有しているため、`{lattice}`独自の長所とは言いにくいです。

### 17.1.3 {ggplot2}パッケージ

{ggplot2}は Hadley Wickham が大学院生の時に開発した可視化パッケージであり<sup>1)</sup>、 Wilkinson [2005] の「グラフィックの文法 (grammar of graphics)」の思想を R 上で具現化したものです。グラフィックの文法という思想は今は{ggplot2}以外にも Plotly や Tableau などでも採用されています。

グラフィックの文法は後ほど詳細に解説しますが、{ggplot2}による作図の特徴は「レイヤーを重ねる」ことです。グラフの様々な要素をそれぞれ 1 つの層 (layer) と捉え、これを重ねられていくことでグラフが出来上がる仕組みです。これは Base R の書き方に似ています。たとえば、{ggplot2}を使って図 17.1 を作る場合、以下のようなコードになります。

```
1 # ggplot2 を用いた作図の例
2 ggplot(data = Country_df) +
3   geom_point(aes(x = PPP, y = HDI, color = OECD)) +
4   labs(x = "一人あたり購買力平価 GDP (USD)", y = "人間開発指数",
5        color = "OECD 加盟有無") +
6   theme_bw()
```

<sup>1)</sup> 厳密に言えば、大学院生の時代に開発したパッケージは{ggplot2}ではなく、{ggplot}です。このパッケージもグラフィックの文法の思想に基づいた可視化パッケージではありますが、今の{ggplot2}とは別物に近いパッケージです。このパッケージは 2008 年 10 月、バージョン 0.4.2 を以って CRAN から削除されました。

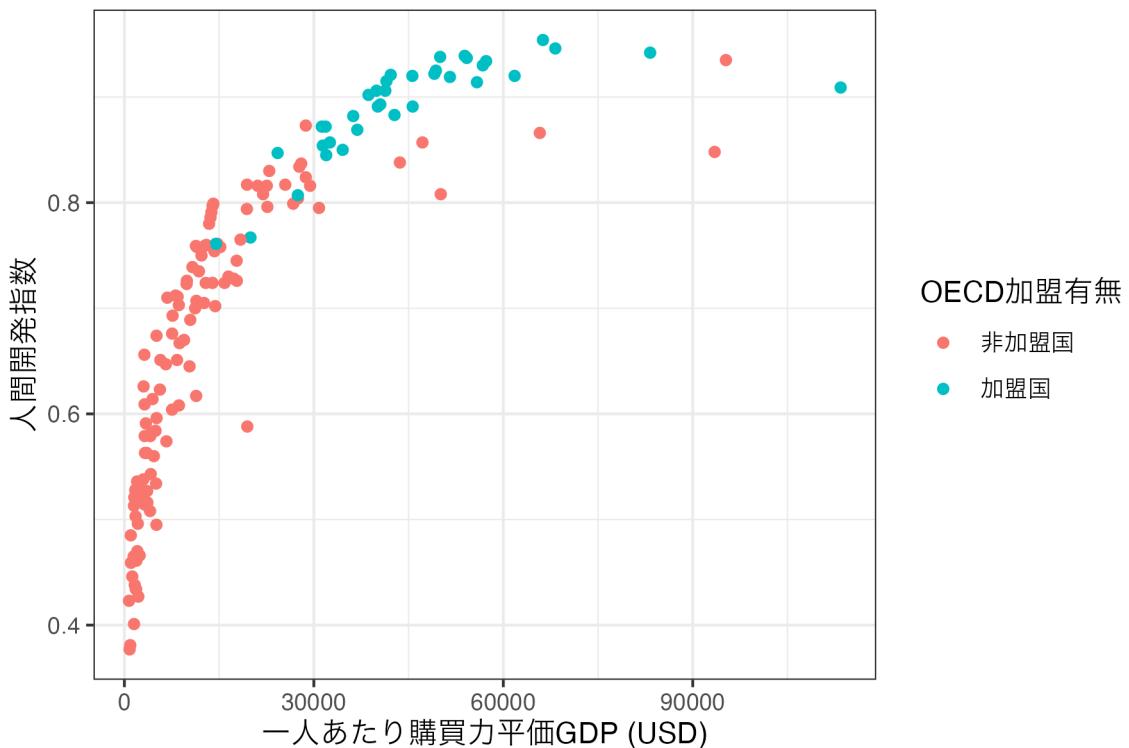


図 17.3: ggplot2 によるグラフ

このように{ggplot2}による作図コードはBase R や{lattice}に比べ、読みやすいのが特徴です。また、書く手間も大きく省かれる場合が多く、結果として出力されるグラフも綺麗です（これは好みによりますが）。しかし、{ggplot2}にも限界はあり、代表的なものとして(1)3次元グラフが作成でないこと、(2)処理速度が遅い点があります。後者は多くの場合においてあまり気にならない程度ですが、3次元プロットが必要な場合は{lattice}や別途のパッケージを使う必要があります。しかし、社会科学において3次元プロットが使われる機会は少なく、2次元平面であっても3次元以上のデータを表現することも可能です。本書では{ggplot2}を用いた可視化方法のみについて解説していきます。

## 17.2 グラフィックの文法

### 17.2.1 グラフィックの文法

本書では特別な事情がない限り、`{ggplot2}`による可視化のみを扱います。既に`{ggplot2}`の gg が「グラフィックの文法 (grammar of graphics)」だと説明しましたが、この概念は Wilkinson [2005] によって提唱された比較的新しいものであり、`{ggplot2}`は Hadley 先生がグラフィックの文法に則った作図のプロセスを R で具現化したものです。

グラフィックスの文法とは、グラフを構造化された方法で記述し、レイヤーを積み重ねることによってグラフを構築するフレームワークです。グラフは様々な要素で構成されています。横軸と縦軸、点、線、グラフ、凡例、図のタイトルなどがあります。横軸や縦軸は線の太さ、目盛りの間隔、数字の大きさなどに分割することも可能です。このように 1 つのグラフは数十、数百以上の要素の集合です。これら一つ一つの要素をレイヤーとして捉え、それを積み重ねることでグラフを作成します。これが簡単な`{ggplot2}`による作図のイメージですが、以下でもうちょっと目に見える形でこれを解説していきます。

### 17.2.2 `ggplot2` のイメージ

それでは`{ggplot2}`によるグラフが出来上がる過程を見ていきます。例えば、以下のようなデータセット `df` があるとします。変数は年度を表す `Year`、鉄道事業者のタイプを表す `Company_Type1`、一日利用者数の平均値を表す `P` があります。例えば、2 行目は 2011 年度における JR が管理する駅の一日平均利用者数が約 7399 名であることを意味します。

```
## # A tibble: 28 x 3
##   Year Company_Type1     P
##   <dbl> <fct>        <dbl>
## 1 2011 その他        4769
## 2 2011  JR           7399
## 3 2011  大手私鉄     19421
## 4 2011  準大手私鉄   7683
## 5 2012 その他        5014
```

```

## 6 2012 JR          7289
## 7 2012 大手私鉄    21286
## 8 2012 準大手私鉄  10471
## 9 2013 その他      5154
## 10 2013 JR         7383
## # ... with 18 more rows

```

このデータ `df` を使って図 17.4 のようなグラフを作成します。以下では作図のコードも載っていますが、詳しく理解しなくとも結構です。理解しなくともいいですが、必ずコードには目を通し、説明文との対応を自分で考えてください。

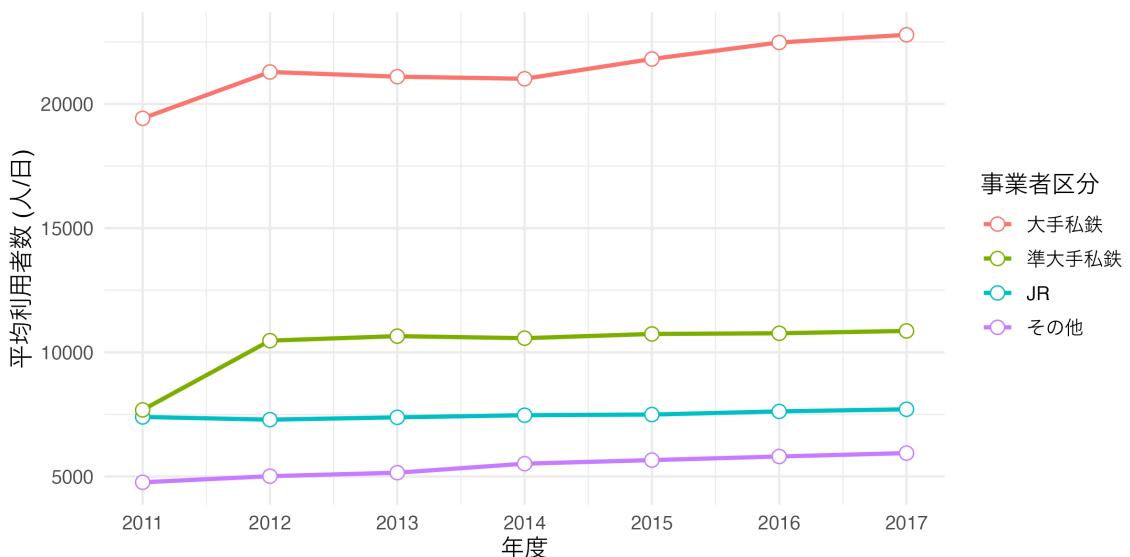


図 17.4: 鉄道駅の事業者区分による平均利用者数の推移

- まずは、グラフに使用するデータを指定し、空のキャンバスを用意します。

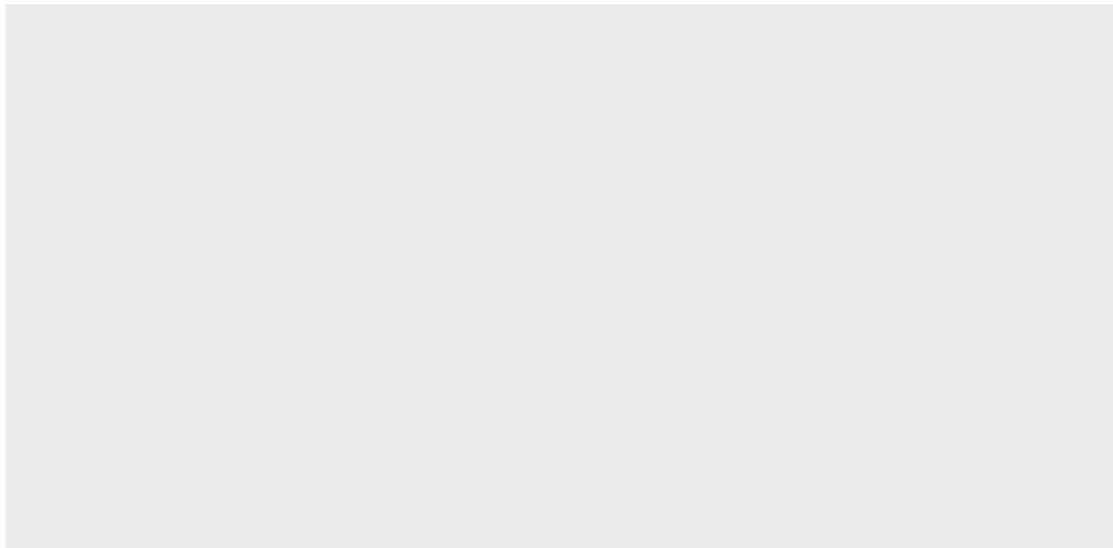


図 17.5: 第 1 層: データとキャンバスの用意

- 折れ線グラフを作成します。折れ線グラフは点の位置を指定すると、勝手に点と点の間を線で繋いでくれます。したがって、必要な情報は点の情報ですが、横軸 (X 軸) は Year、縦軸 (Y 軸) は P にした点を出力します。この点を Company\_Type1 ごとに色分けします。これで折れ線グラフが出来上がります。線の太さは 1 にします。

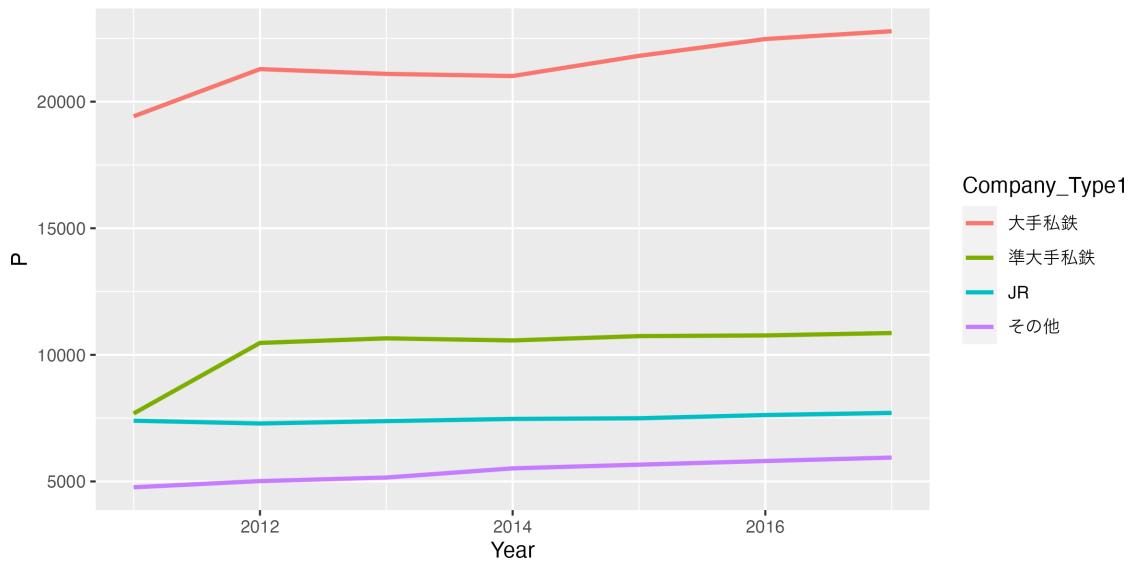


図 17.6: 第 2 層: 幾何オブジェクト (折れ線グラフ) の指定とマッピング

3. 続いて、折れ線グラフに散布図を載せます。これは線が引かれていない折れ線グラフと同じです。したがって、横軸、縦軸、色分けの情報は同じです。しかし、点と線が重なると点がよく見えないこともあるので、点の大きさを3にし、形は枠線付きの点にします。点の中身は白塗りをします。つまり、Company\_Type1によって変わるのは、点の枠線です。

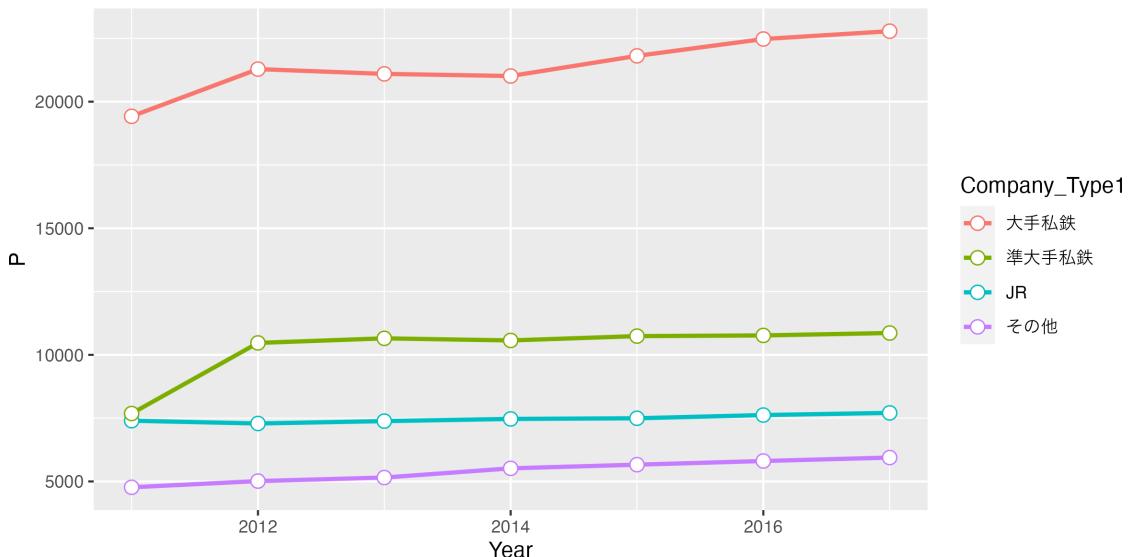


図 17.7: 第3層: 幾何オブジェクト (散布図) の指定とマッピング

4. 横軸と縦軸、そして凡例のタイトルを日本語に直します。日本語のレポート、論文なら図表も日本語にすべきです。横軸のラベルは"年度"、縦軸のラベルは"平均利用者数 (人/日)"にします。色の凡例タイトルは"事業者区分"にします。

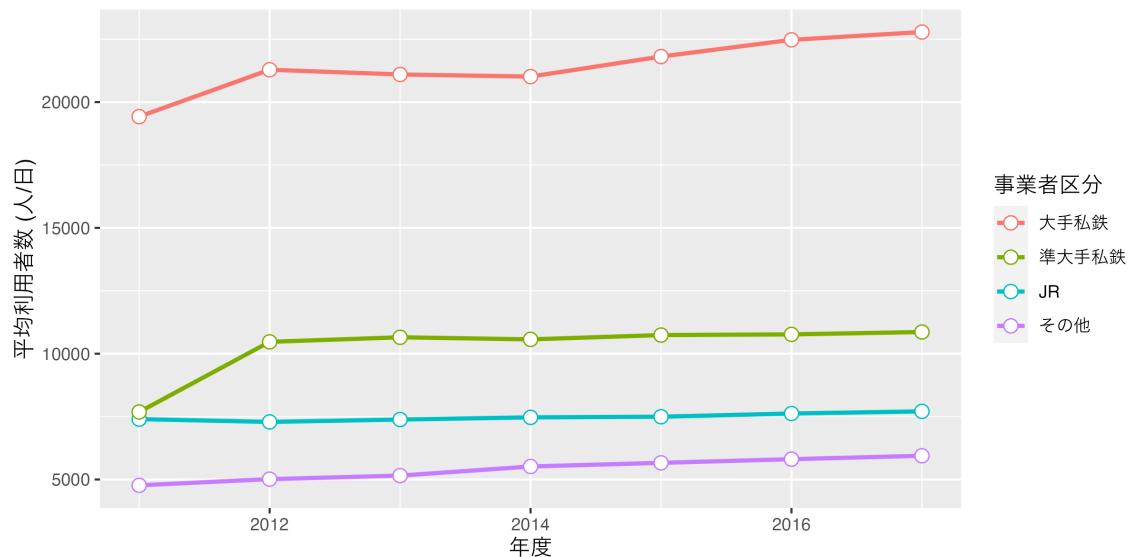


図 17.8: ラベルの修

5. 横軸のスケールを修正します。横軸は連続 (continuous) 変数です。今の目盛りは 2012、2014、2016 になっていますが、これを 1 年刻みにし、それぞれの目盛りの ラベルも 2011、2012、2013、... にします。

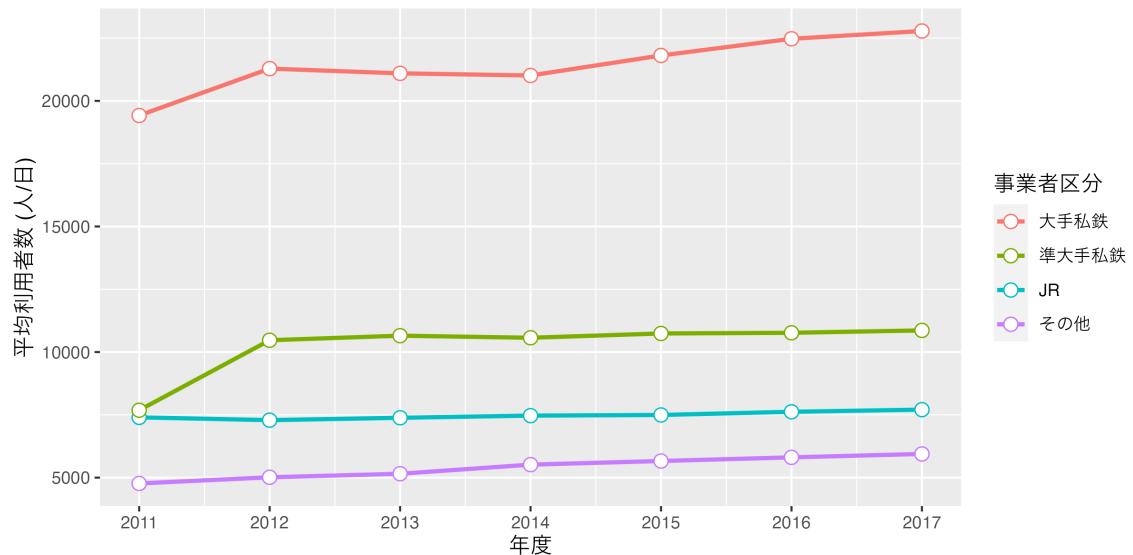


図 17.9: スケールの修正

6. グラフのテーマを {ggplot2} が基本的に提供している `minimal` に変更し、フォント

サイズを 12 に変更します。

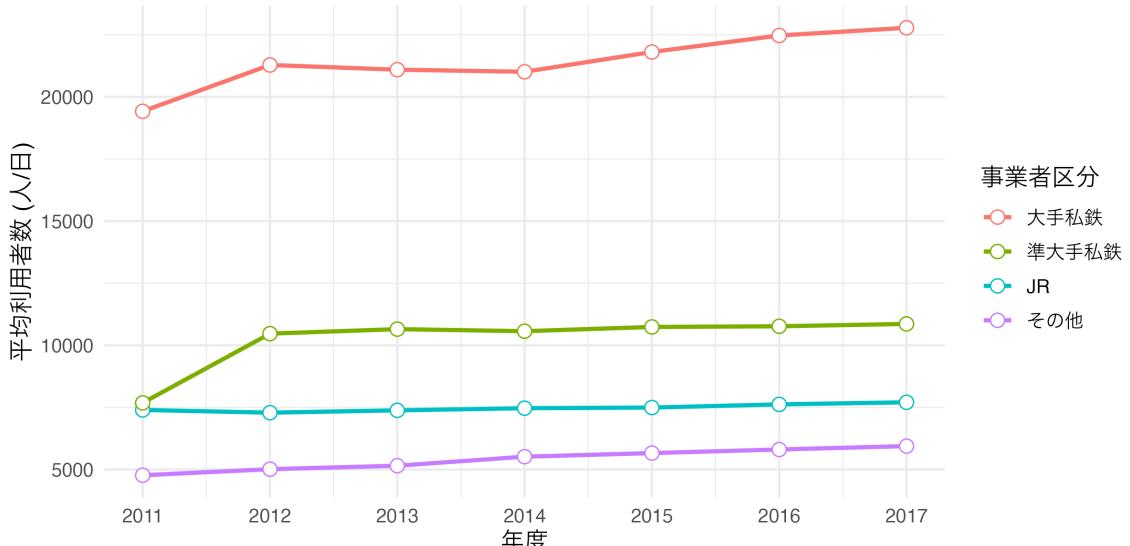


図 17.10: 見た目の調整

これで図が出来上りました。このように{ggplot2}では図の要素をレイヤーと捉えます。このレイヤーを生成する関数が `ggplot()`、`geom_line()`、`lab()` などであり、これらを `+` 演算子を用いて重ねていきます。このイメージを図にすると図 17.11 のように表現できます。

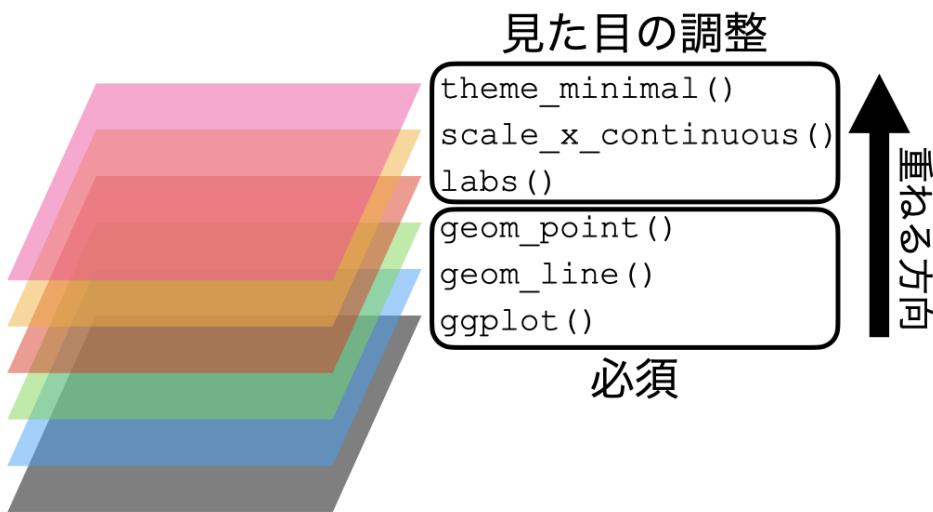


図 17.11: ggplot2 の図が出来上がるまで (全体像)

それでは、グラフは具体的にどのような要素で構成されているかを以下で解説します。

### 17.3 グラフィックの構成要素

{ggplot2}におけるプロット (plot) は「データ + 幾何オブジェクト + 座標系」で構成されます。

ここでいう**データ**は主にデータフレームまたは tibble です。これは主に ggplot() 関数の第一引数と指定するか、パイプで渡すのが一般的です。ただし、{ggplot2}で作図するためには、データを予め整然データに整形する必要があります。

**幾何オブジェクト (geometry object)** とは簡単に言うと図の種類です。散布図、折れ線グラフ、棒グラフ、ヒストグラムなど、{ggplot2}は様々なタイプの幾何オブジェクトを提供しており、ユーザー自作の幾何オブジェクトも R パッケージとして多く公開されています。幾何オブジェクトは関数の形で提供されており、geom\_で始まるといった共通点があります。散布図は geom\_point()、折れ線グラフは geom\_line() のような関数を使います。

この幾何オブジェクトに線や点、棒などを表示する際には、どの変数が横軸で、どの変数が縦軸かを明記する必要があります。また、変数によって点や線の色が変わったりする場

合も、どの変数によって変わるかを明記します。これを **マッピング (mapping)** と呼びます。また、必要に応じて位置 (position) と統計量 (stat) を明記する必要がありますが、これは指定しなくともとりあえず何らかの図は出力されます。

最後に**座標系 (coordinate system)** は幾何オブジェクトが表示される空間の特徴を定義します。最も重要な特徴は横軸と縦軸の下限と上限です。または、空間を回転することなどもできます。

{ggplot2}の図は以上の3つ要素を重ねることで出来ます。

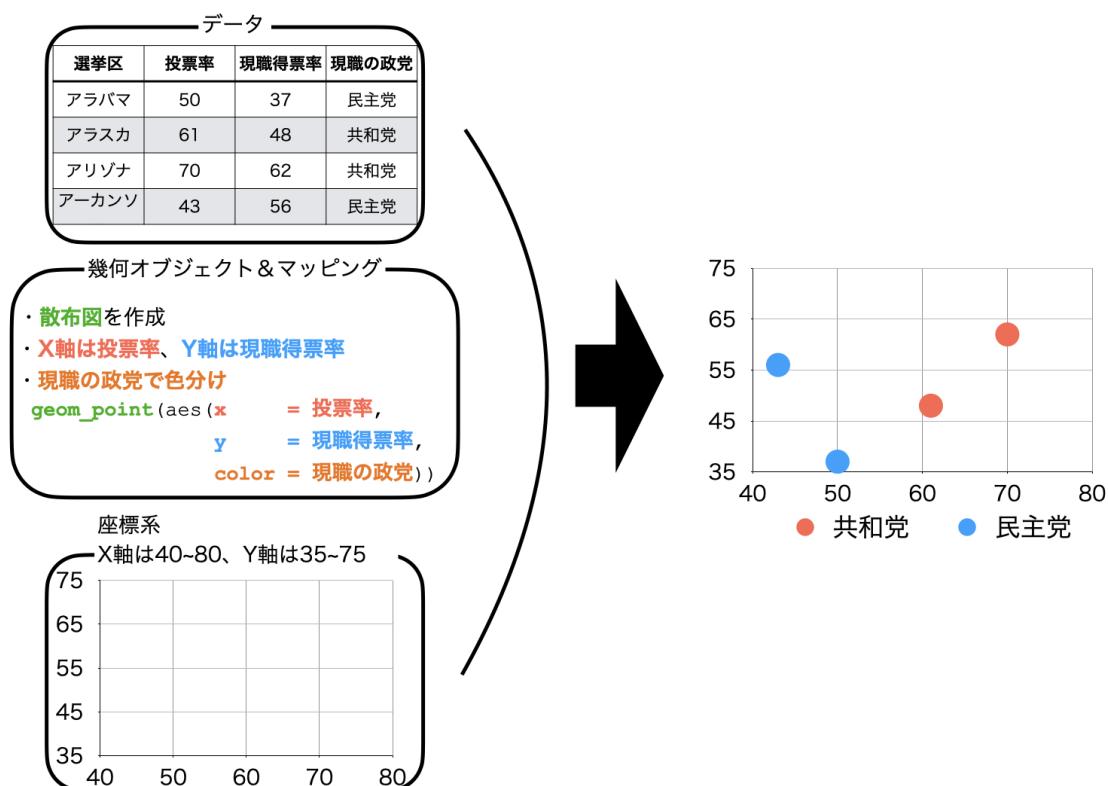


図 17.12: ggplot2 の構造の例

ただし、この中で座標系は適切だと判断される座標系に設定してくれるため、ユーザーが必ず指定すべきものはデータと幾何オブジェクトのみです。また、幾何オブジェクトはマッピングを含んでおり、これも必ず指定する必要があります。したがって、{ggplot2}で作図するための最小限のコードは以下のようになります。

```
1 # ggplot2 におけるプロットの基本形
2 # データは ggplot の第一引数と使う場合が多いため、「data =」は省略可能
3 # マッピングは主に幾何オブジェクトの第一引数として使うため、「mapping =」は省略可能
4 ggplot(data = データ名) +
5   幾何オブジェクト関数 (mapping = aes(マッピング))
6
7 # パイプを使う場合
8 データ名 %>%
9   ggplot() +
10  幾何オブジェクト関数 (mapping = aes(マッピング))
```

注意すべき点は{ggplot2}においてレイヤーを重ねる際は `%>%` ではなく、`+` を使う点です。パイプ演算子は左側の結果を右に渡す意味を持ちますが、{ggplot2}はデータを渡すよりも、**レイヤーを足していく**イメージですから、`+` を使います。

以下は{ggplot2}の必須要素であるデータと幾何オブジェクト、マッピングなどについて解説し、続いて図は見栄を調整するための関数群を紹介します。

### 17.3.1 データ

作図のためにはデータはなくてはなりません。データは `data.frame` 型、または `tibble` 型であり、一般的には `ggplot()` 関数の第一引数として指定します。例えば、`ggplot(data = データ名)` のように書いてもいいですし、`data =` は省略して、`ggplot(データ名)` でも構いません。

データを指定するもう一つの方法はパイプ演算子を使うことです。この場合、`データ名 %>% ggplot()` のように書きます。書く手間はほぼ同じですが、`dplyr` や `tidyverse` などでデータを加工し、それをオブジェクトとして保存せずにすぐ作図に使う場合は便利です。

実際、使う機会は少ないですが、1つのグラフに複数のデータを使う場合もあります。{ggplot2}は複数のデータにも対応していますが、とりあえずメインとなるデータを `ggplot()` に指定し、追加的に必要なデータは今度説明する幾何オブジェクト関数内で指定します。この方法については適宜必要に応じて説明します。

### 17.3.2 幾何オブジェクト

しかし、データを指定しただけで図が出来上がるわけではありません。指定したデータを使ってどのような図を作るかも指定する必要があります。この図のタイプが幾何オブジェクト (geometry object) であり、`geom_*`() 関数で表記します。たとえば、散布図を作る場合、

```
1 データ名 %>%
2   ggplot() +
3     geom_point()
```

のように指定します。以上のコードは「あるデータを使って (データ名 %>%), キャンバスを用意し (ggplot() +), 散布図を作成する (geom\_point())。」と読むことが出来ます。

また、この幾何オブジェクトは重ねることも可能です。よく見る例としては、散布図の上に回帰曲線 (`geom_smooth()`) や折れ線グラフ (`geom_line()`) を重ねたものであり、これらの幾何オブジェクトは + で繋ぐことが可能です。

`{ggplot2}` が提供する幾何オブジェクト関数は散布図だけでなく、棒グラフ (`geom_bar()`)、ヒストグラム (`geom_histogram()`)、折れ線グラフ (`geom_line()`)、ヒートマップ (`geom_tile()`) など、データ分析の場面で使われるほとんどの種類が含まれています。他にもユーザーが作成した幾何オブジェクトもパッケージとして多く公開されています（たとえば、非巡回有向グラフ作成のための`{ggdag}`、ネットワークの可視化のための`{ggnetwork}`など）。

### 17.3.3 マッピング

どのようなデータを使って、どのような図を作るかを指定した後は、変数を指定します。たとえば、散布図の場合、各点の横軸と縦軸における位置情報が必要です。ヒストグラムならヒストグラムに必要な変数を指定する必要があります。このようにプロット上に出力されるデータの具体的な在り方を指定するのをマッピング (mapping) と呼びます。

マッピングは幾何オブジェクト関数内で行います。具体的には `geom_*`() 内に

`mapping = aes(マッピング)` で指定します。`aes()` も関数の一種です。散布図なら `geom_point(mapping = aes(x = X 軸の変数, y = Y 軸の変数))` です。ヒストグラムなら横軸のみを指定すればいいので `geom_histogram(mapping = aes(x = 変数名))` で十分です。マッピングは一般的には `geom_*`() の第一引数として渡しますが、この場合、`mapping =` は省略可能です。

マッピングに必要な変数、つまり `aes()` に必要な引数は幾何オブジェクトによって異なります。散布図や折れ線グラフなら X 軸と Y 軸の情報が必須であるため、2 つ必要です (`x` と `y`)。ヒストグラムは連続変数の度数分布表を自動的に作成してからグラフが作られるから 1 つが必要です (`x`)。また、等高線図の場合、高さの情報も必要なので 3 つの変数が必要です (`x` と `y`、`z`)。これらの引数は必ず指定する必要があります。

以上の引数に加え、追加のマッピング情報を入れることも可能です。たとえば、鉄道事業者ごとの平均利用者数を時系列で示した最初の例を考えてみましょう。これは折れ線グラフですので、`mapping = aes(x = 年度, y = 利用者数)` までは必須です。しかし、この図にはもう一つの情報がありますね。それは事業者のタイプです。事業者のタイプごとに線の色を変えたい場合は、`aes()` 内に `color = 事業者のタイプ` を、線の種類を変えたい場合は、`linetype = 事業者のタイプ` のように引数を追加します。こうすると 2 次元のプロットに 3 次元の情報 (年度、利用者数、事業者タイプ) を乗せることができます。もちろん、4 次元以上にすることも可能です。たとえば、地域ごとに異なる色を、事業者タイプごとに異なる線のタイプを指定する場合は、`mapping = aes(x = 年度, y = 利用者数, color = 地域, linetype = 事業者のタイプ)` のように指定します。`color` や `linetype` 以外にも大きさ (`size`)、透明度 (`alpha`)、点のタイプ (`shape`)、面の色 (`fill`) などを指定することができます。

それでは、最初にお見せした図 17.10 のマッピングはどうなるでしょうか。図 17.10 の幾何オブジェクトは折れ線グラフ (`geom_line()`) と散布図 (`geom_point()`) の 2 つです。それぞれの幾何オブジェクトのマッピング情報をまとめたのが表 17.2 です。

先ほどマッピング引数は幾何オブジェクト関数内で指定すると言いましたが、実は `ggplot()` 内に入れ、`geom_*`() 内では省略することも可能です。幾何オブジェクトが 1 つのみならどっちでも問題ありません。しかし、幾何オブジェクトが 2 つ以上の場合は注意が必要です。全ての幾何オブジェクトがマッピングを共有する場合は `ggplot()` の方が書く手間が省きます。たとえば、表 17.2 を見ると、`geom_line()` と `geom_point()`

表 17.2: 図  
reffig:visual1-ggplot-image-8 のマッピング情報

幾何オブジェクト	マッピング要素	変数	引数
'geom_line'	X 軸	'Year'	'x'
	Y 軸	'P'	'y'
	線の色	'Company_Type1'	'color'
'geom_point'	X 軸	'Year'	'x'
	Y 軸	'P'	'y'
	枠線の色	'Company_Type1'	'color'

は `x`、`y`、`color` 引数の値が同じですから、`ggplot()` 内に `aes()` を入れることも可能です。しかし、幾何オブジェクトがマッピングを共有しない場合は幾何オブジェクト関数内に別途指定する必要があります。あるいは、共有するところだけ、`ggplot()` に書いて、共有しない部分だけ幾何オブジェクトで指定することも可能です。したがって、上の図は以下のコードでも作成することができます。

```

1 # geom_line() と geom_point() はマッピング情報を共有しているため、
2 # ggplot() 内に指定
3 df %>%
4   ggplot(aes(x = Year, y = P, color = Company_Type1)) +
5   geom_line(size = 1) +
6   geom_point(size = 3, shape = 21, fill = "white") +
7   labs(x = "年度", y = "平均利用者数（人/日）", color = "事業者区分") +
8   scale_x_continuous(breaks = 2011:2017, labels = 2011:2017) +
9   theme_bw()

```

#### 17.3.4 マッピング: その他

以上のことさえ覚えれば、とりあえず図は作れます。最後に、必須要素ではありませんが、幾何オブジェクトに使う引数について説明します。先ほど説明しました `color` や `linetype`、`size` などはマッピングの情報として使うことも可能ですが、`aes()` の外側に置くことも可能です。しかし、その挙動はかなり異なります。`color` が `aes()` の外側

にある場合は、幾何オブジェクトの全要素に対して反映されます。

たとえば、横軸が「ゲームのプレイ時間」、縦軸が「身長」の散布図を作成しようとします(図 17.13)。ここで `color` 引数を追加しますが、まずは `aes()` の外側に入れます。

```

1 # 例 1: 全ての点の色が赤になる
2 データ %>%
3   ggplot() +
4   geom_point(aes(x = ゲームのプレイ時間, y = 身長),
5             color = "red")

```

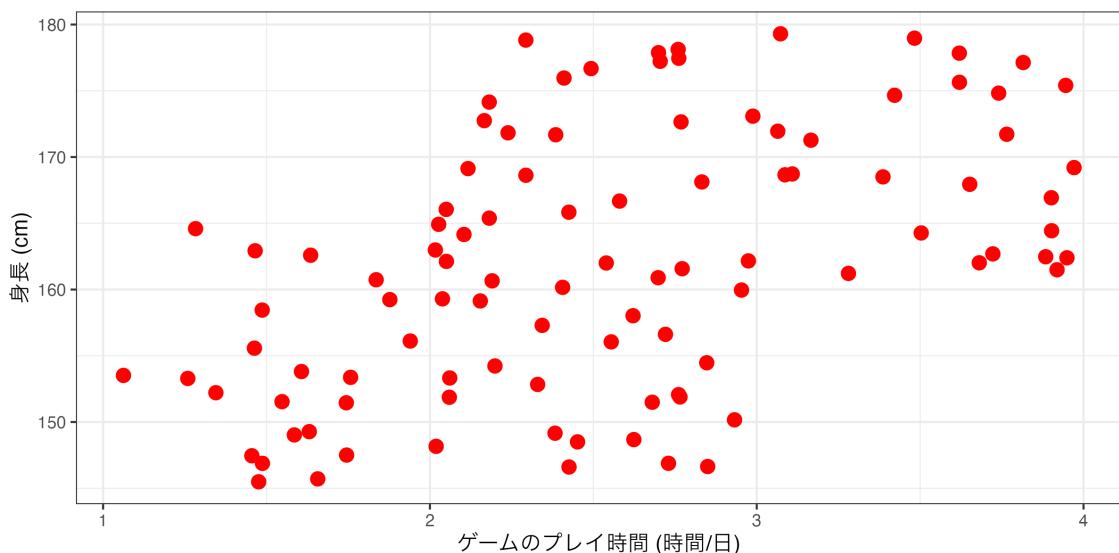


図 17.13: ‘color’を ‘aes()’外側に置いた場合

ここで注目する点は

1. 散布図におけるすべての点の色が `color` で指定した色 ("red" = 赤) に変更された点
2. `color` の引数は変数名でなく、具体的な色を指定する点

以上の 2 点です。`aes()` の内側に `color` を指定する場合(図 17.14)は、以下のように変数名を指定します。たとえば、性別ごとに異なる色を付けるとしたら、

```

1 # 例 2: 性別ごとに点の色が変わる
2 データ %>%
3   ggplot() +
4   geom_point(aes(x = ゲームのプレイ時間, y = 身長, color = 性別))

```

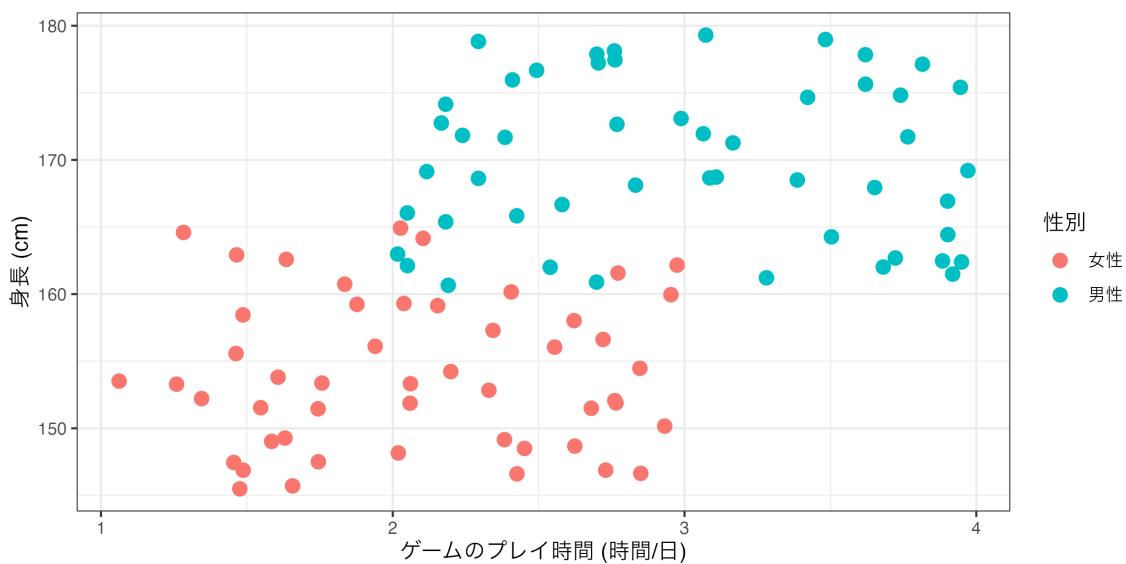


図 17.14: ‘color’を‘aes()’内側に置いた場合

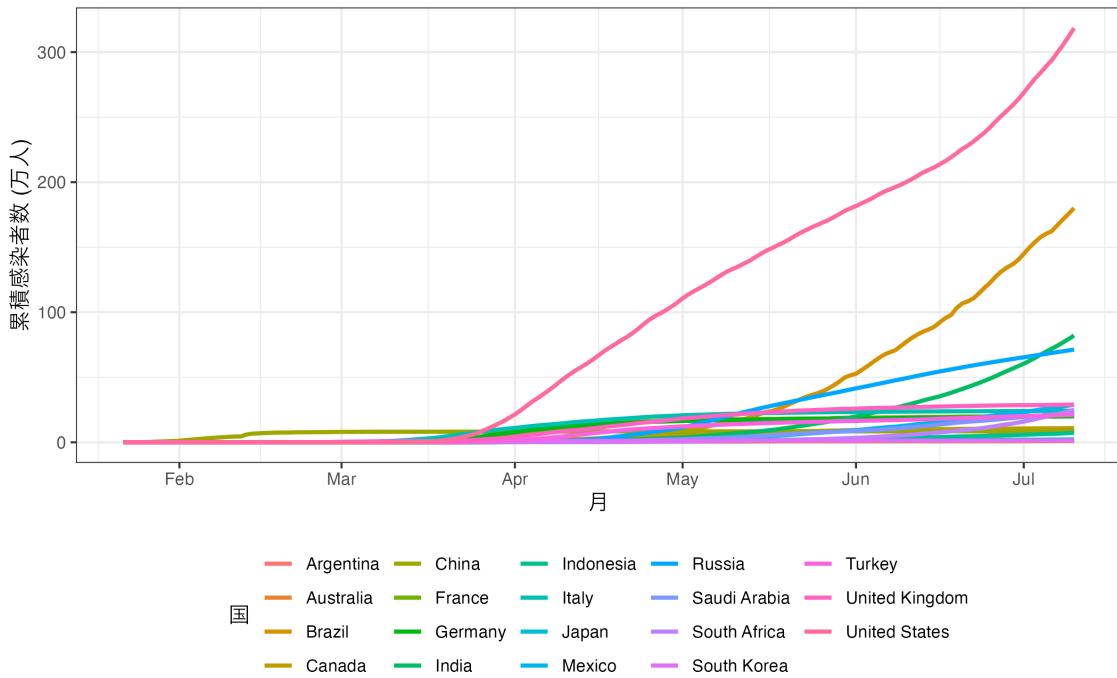
以上のように書きます。`aes()` の内部はマッピングの情報が含まれています。言い換えると、`aes()` の中はある変数がグラフにおいてどのような役割を果たしているかを明記するところです。2つ目の例では性別という変数が色分けをする役割を果たすため、`aes()` の内側に入ります。一方、1つ目の例では色分けが行われておりません。

### 17.3.5 座標系

座標系はデータが点や線、面などで出力される空間を意味し、`coord_*`() 関数群を用いて操作します。

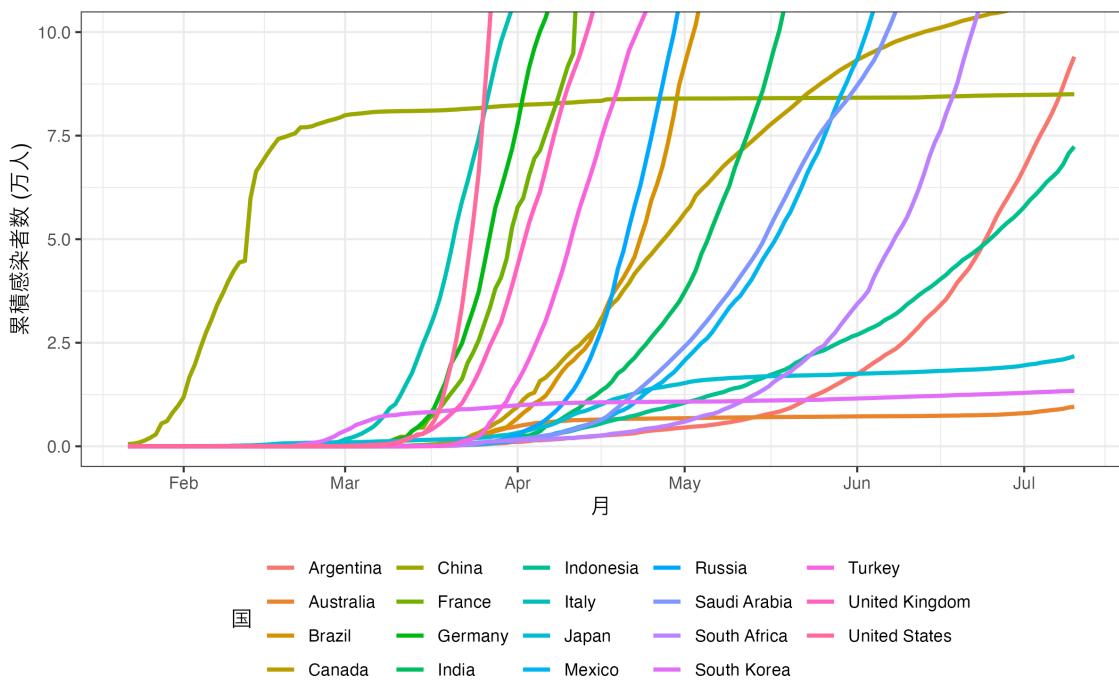
座標系のズームイン (zoom-in) やズームアウト (zoom-out) を行う `coord_cartesian()`、横軸と縦軸を交換する `coord_flip()`、横軸と縦軸の比率を固定する `coord_fixed()` がよく使われます。座標系の説明は次章以降で詳しく解説しますが、ここでは座標系の

ズームインを見てみましょう。以下の図 17.15 は各国の COVID19 の感染者数を時系列で示したものです。これを見るとアメリカ、ブラジル、インドは大変だなーくらいしかわかりません。感染者数が比較的に少ない国のデータは線としては存在しますが、なかなか区別ができません。



ここで座標系をズームインすると、一部の情報は失われますが、ズームインされた箇所はより詳細にグラフを観察できます。ズームインと言っても難しいものではありません。単に、軸の上限、下限を調整するだけです。たとえば、縦軸の上限を 10 万人に変更したのが図 17.16 です。アメリカなど感染者数が 10 万人を超える国家の時系列情報の一部は失われましたが、日中韓などのデータはより見やすくなったかと思います<sup>2)</sup>。

2) 実は図 17.161) と図 17.162) は似たような色が多く使われているため、裏められそうなグラフではありません。



また、同じ棒グラフや散布図、折れ線グラフでも座標系を変えることによって図の見方が劇的に変わることもあります。我々にとって最も馴染みのある座標系はデカルト座標系(直交座標系)です。このデカルト座標系に切片0、傾き1の直線を引きます。そして同じ図に対して座標系のみを極座標系(polar coordinates system)に変更します。この2つを比較したのが図17.17です。この2つの図は同じデータ、同じ変数、同じ幾何オブジェクトで構成されています。異なるのは座標系ですが、見方が劇的に変わります。

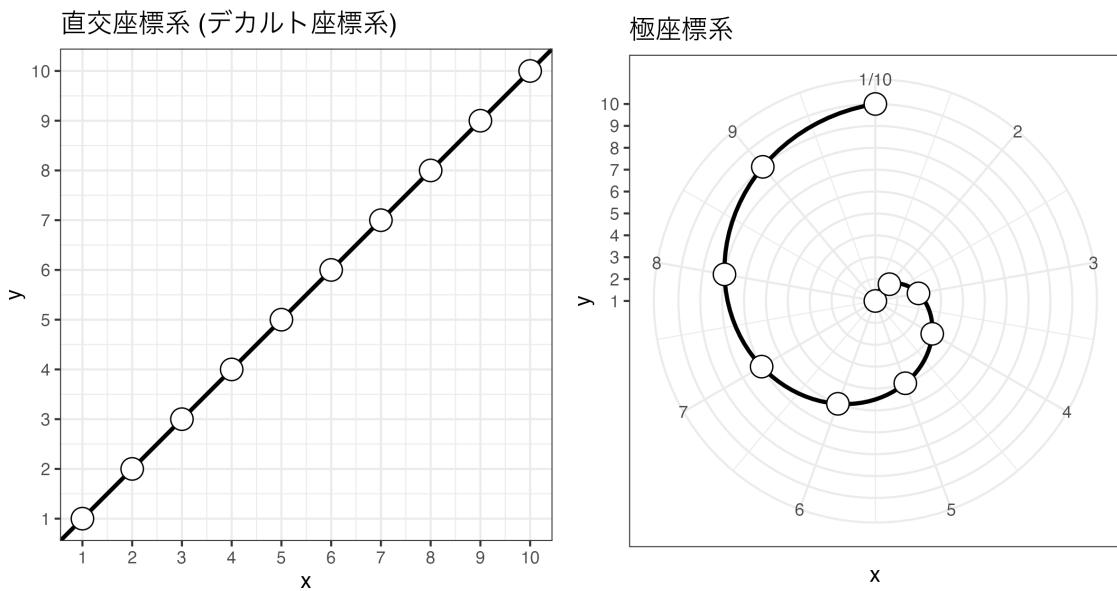


図 17.17: 直交座標系と極座標系の比較 (1)

こんな座標系を実際に使う機会は多くないかも知れませんが、極座標系は割と身近なところで見ることができます。それは積み上げ棒グラフと円グラフの関係です。図 17.18 はデカルト座標系上の積み上げ棒グラフを極座標系に変換したものです。

```

1  Coord_Fig4 <- data.frame(x = c(rep("男性", 4),
2                                rep("女性", 5),
3                                rep("ニュータイプ", 1))) %>%
4
5  ggplot() +
6  geom_bar(aes(x = factor(""), fill = factor(x)), width = 1) +
7  labs(fill = "性別", x = "", y = "人数") +
8  ggtitle("直交座標系 (デカルト座標系)") +
9  theme_bw() +
10 theme(legend.position = "bottom")
11
12 Coord_Fig5 <- Coord_Fig4 +
13   ggtitle("極座標系") +
14   coord_polar(theta = "y")

```

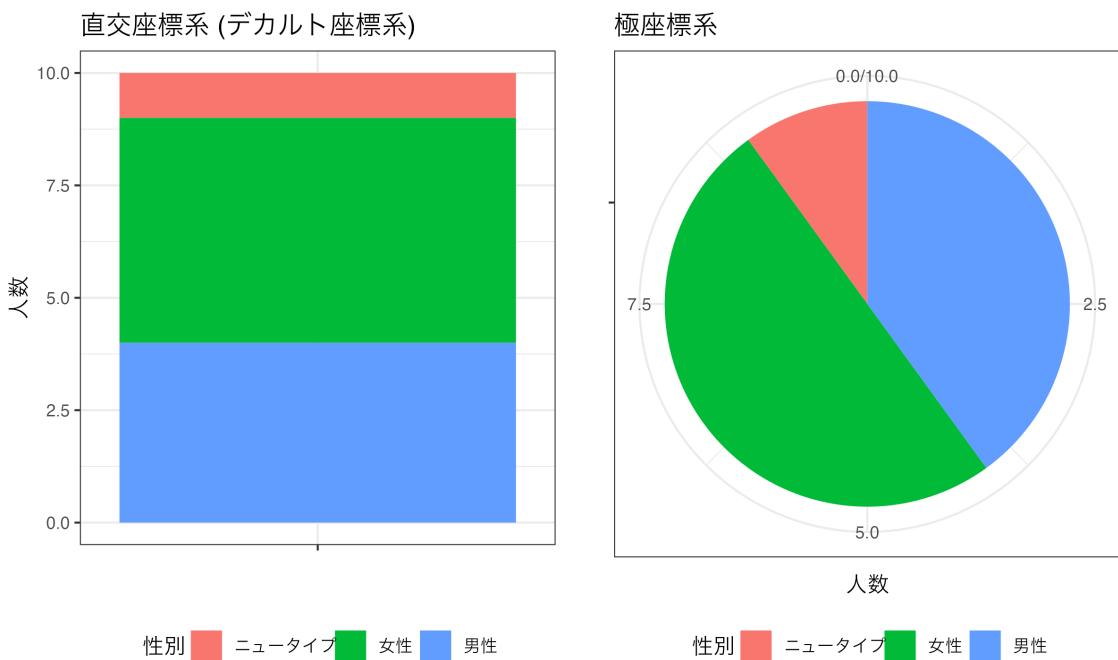


図 17.18: 直交座標系と極座標系の比較 (2)

他にも軸を対数スケールなどに変換する `coord_trans()`、地図の出力に使われる `coord_map()` や `coord_sf()` などがあり、適宜紹介していきます。

### 17.3.6 スケール

既に説明しました通り、`{ggplot2}`は様々な幾何オブジェクトがあり、これによってグラフ上におけるデータの示し方が変わります。例えば、散布図だとデータは点として表現され、折れ線グラフだと線、棒グラフやヒストグラムだと面で表現されます。これらの点、線、面などが持つ情報がスケール (scale) です。点だと縦軸上の位置、横軸上の位置が必ず含まれ、他にも点の形、点の大きさ、点の色、枠線の色、透明度などの情報を含むことが可能です。線も線が折れるポイントの横軸・縦軸上の位置、線の太さ、線の色などがあります。

この形、色、大きさ、太さなどを調整するためには `scale_*_*` 関数群を使います。例えば、図 17.13 の場合、点は横軸上の位置、縦軸上の位置、色の 3 つのスケールを持っています。横軸と縦軸は連続変数 (時間と身長)、色は名目変数 (性別) です。ここで横軸の

スケールを調整するためには `scale_x_continuous()` レイヤーを追加します。縦軸も同様に `scale_y_continuous()` を使います。この `x` と `y` はスケール、`continuous` は連続変数であることを意味します。それでは、色を調整するためにはどうすれば良いでしょうか。それは `scale_color_manual()` です。`color` は色を、`manual` は手動調整を意味しますが、主に性別や都道府県のような名目変数のスケール調整に使います。

これらの `scale_*_()` 関数群は点、線、面が持つ位置情報や性別を変更することではなく、その見せ方を変更するだけです。図 17.14 の縦軸の目盛りは「150、160、170、180」となっていますが、`scale_y_continuous()` を使うとこの目盛りが変わります。点の位置が変わることはありません。たとえば、以下のコードは `scale_y_continuous()` を使って縦軸の目盛りを 5cm 刻みに変更するためのコードであり、図 17.19 はその結果です。

```

1 # scale_y_continuous() の例
2 データ %>%
3   ggplot() +
4   geom_point(aes(x = ゲームのプレイ時間, y = 身長, color = 性別)) +
5   scale_y_continuous(breaks = c(145, 150, 155, 160, 165, 170, 175, 180),
6                      labels = c(145, 150, 155, 160, 165, 170, 175, 180))

```

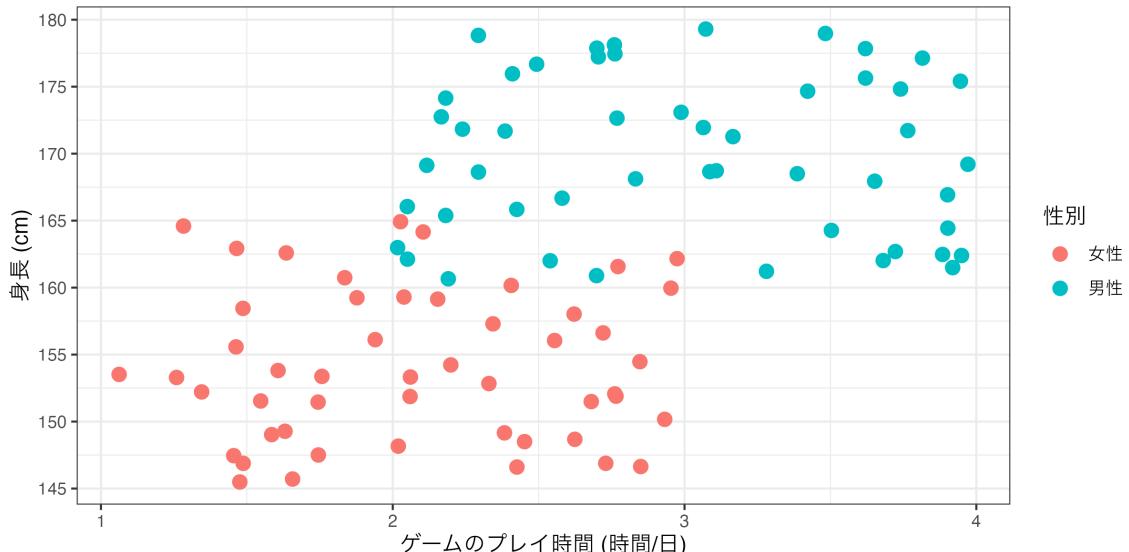


図 17.19: `scale_y_continuous()` を使って縦軸を 5cm 刻みに変更

また、性別ごとの色を変更する際は `scale_color_manual()` を使います（図 17.20）。

```

1 # scale_color_manual() の例
2 データ %>%
3   ggplot() +
4   geom_point(aes(x = ゲームのプレイ時間, y = 身長, color = 性別)) +
5   scale_y_continuous(breaks = c(145, 150, 155, 160, 165, 170, 175, 180),
6                      labels = c(145, 150, 155, 160, 165, 170, 175, 180)) +
7   scale_color_manual(values = c("男性" = "#ff9900", "女性" = "#339900"))

```

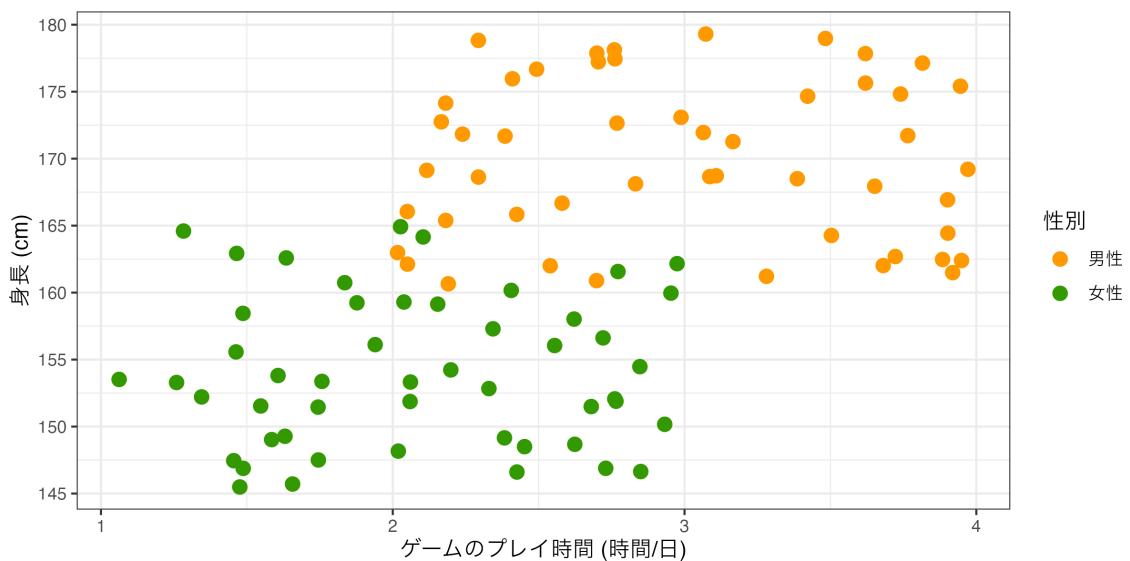


図 17.20: さらに `scale_color_manual()` を使って色を指定

`scale_*_*` 関数群は以上のように、`scale_`スケールのタイプ\_変数のタイプ()です。つまり、`scale_x_date()`、`scale_y_discrete()` や `scale_color_continuous()` など様々な組み合わせが可能であり、`{ggplot2}`は様々なタイプのスケールと変数のための関数群を提供しています。もちろん、ユーザーから独自のスケール関数を作ることも可能であり、パッケージとして公開することも可能です。

### 17.3.7 ファセット

ファセット (facet) は日本語では「面」、「切子面」などで訳されますが、誤解を招く可能性があるため、本書ではそのままファセットと訳します。ファセットとはデータの部分集合 (subset) を対象にしたグラフを意味します。

たとえば、フリーダムハウスでは毎年、各国を「自由」、「部分的に自由」、「不自由」と格付けをし、結果を公表しています。世界に自由な国、部分的に自由な国、不自由な国が何カ国あるかを大陸ごとに示したいとします。カテゴリごとの個数を示すには棒グラフが効果的であり、図 17.21 のように示すことができます。

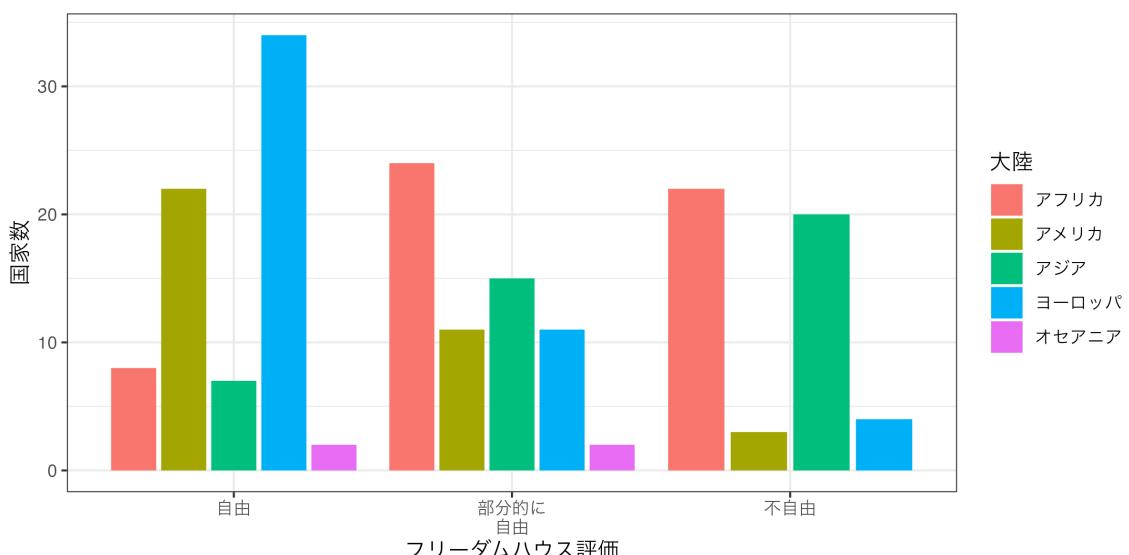


図 17.21: ファセットを分けないグラフの例

このグラフを見ると各大陸に自由な国がどれほどあるかが分かりますが、人によっては読みにくいかも知れません。できれば大陸ごとに分けたグラフの方が見やすいでしょう。そのためにはデータを特定の大陸に絞って、そのデータを用いた棒グラフを作り、最終的には出来上がったグラフたちを結合する必要があります。

しかし、{ggplot2}のファセットを指定するとそのような手間が省けます。これはデータを大陸ごとの部分集合に分割し、1つのプロット上に小さい複数のプロットを出力し

ます。

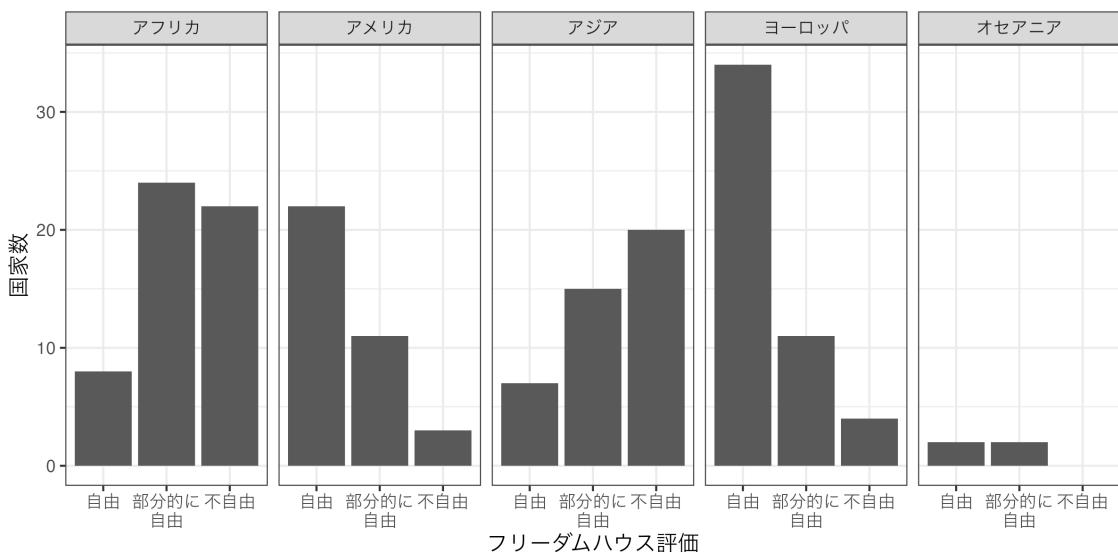


図 17.22: 大陸ごとにファセットを分けたグラフの例

ファセットを指定するには `facet_*`() 関数群を使います。具体的には `facet_wrap()` と `facet_grid()` がありますが、今回のように 1 つの変数（ここでは「大陸」）でファセットを分割する場合は主に `facet_wrap()` を、2 つの変数で分割する場合は `facet_grid()` を使います。

ここまでが`{ggplot2}`の入門の入門です。韓国旅行に例えると、やっと仁川国際空港の入国審査を通ったところです。The R Graph Gallery を見ると、主に`{ggplot2}`で作成された綺麗な図がいっぱいあります。しかし、ここまで勉強してきたものだけでは、このような図を作るのは難しいです。そもそもサンプルコードを見ても理解するのが難しいかも知れません。次章では本格的な`{ggplot2}`の使い方を解説します。到達目標は（1）「よく使う」グラフが作成できること、そして（2）The R Graph Gallery のサンプルコードを見て自分で真似できるようになることです。

## 17.4 良いグラフとは

本格的な作図に入る前に、「優れたグラフとは何か」について考えてみましょう。しかし、優れたグラフの条件を数ページでまとめるのは難しいです。筆者らがいつか暇になったら、これに關しても詳細に1つの章として解説しますが、ここでは参考になる資料をいくつかリストアップします。

- Yau, Nathan. 2011. *Visualize This: The FlowingData Guide to Design, Visualization, and Statistics*. Wiley
- Cairo, Alberto. 2016. *The Truthful Art: Data, Charts, and Maps for Communication*. New Riders.
- Healy, Kieran. 2018. *Data Visualization: A Practical Introduction*. New Riders. Princeton University Press.
- 藤俊久仁・渡部良一. 2019. 『データビジュアライゼーションの教科書』秀和システム. (第5章以降)
- 永田ゆかり. 2020. 『データ視覚化のデザイン』SBクリエイティブ.
- BBC Visual and Data Journalism cookbook for R graphics

### 17.4.1 データ・インク比

可視化が本格的に注目を浴びるようになったのは Edward R. Tufte の 1983 年著作、*The Visual Display of Quantitative Information* からですが、この本のメッセージは単純で、「データ・インク比 (Data-ink ratio) を最大化せよ」の一言に要約できます。データ・インク比は以下のように定義されます [Tufte, 2001]。

$$\text{Data-ink ratio} = \frac{\text{data-ink}}{\text{total ink used to print the graphic}} \quad (17.1)$$

$$= \text{proportion of a graphic's ink devoted to the} \quad (17.2)$$

$$\text{non-redundant display of data-information} \quad (17.3)$$

$$= 1.0 - \text{proportion of a graphic that can be erased} \quad (17.4)$$

$$\text{without loss of data-information.} \quad (17.5)$$

データ・インク比は「データ・インク」を「グラフの出力に使用されたインクの総量」で割ったものです。データ・インクとはデータの情報を含むインクの量を意味します。これを言い換えると、グラフにおいて情報損失なしに除去できるグラフの要素が占める割合を1から引いたものです。

ここで1つの例を紹介します。たとえば、G20加盟国におけるCOVID19の累積感染者数を時系列で示すとします。そこで最近、インドにおいて感染者数が急増していることを示したいとします。まず、図17.23から見ましょう。

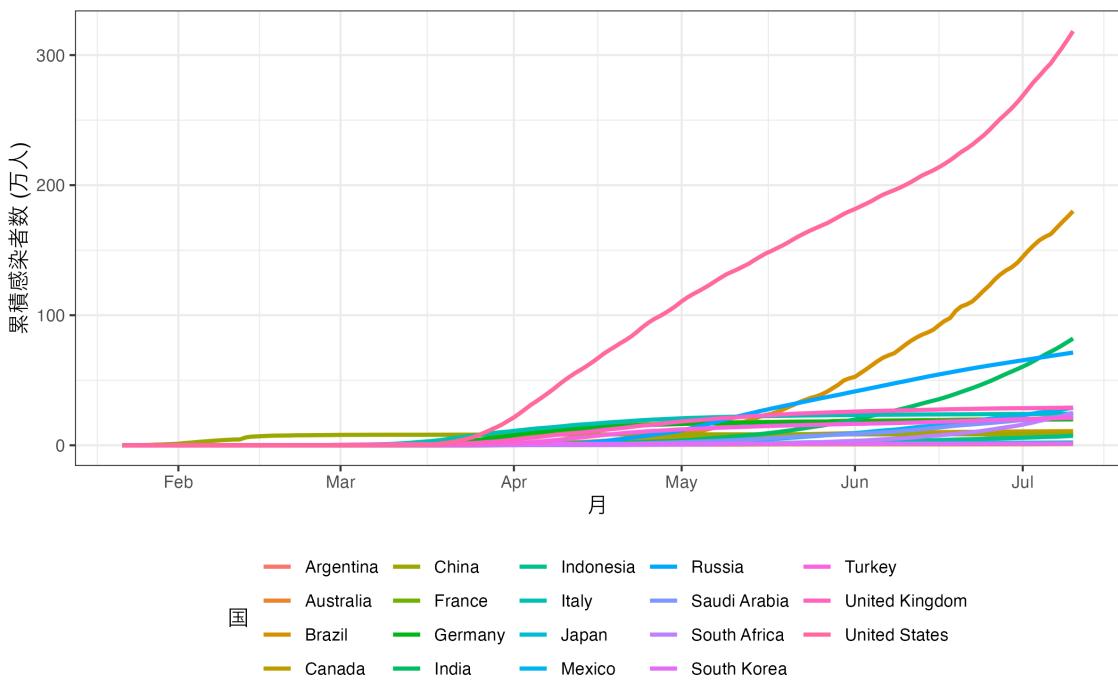


図17.23: データ・インク比の例 (改善前)

そもそもどの線がインドを表しているのかが分かりにくいです。カテゴリが増えると使える色に制約が生じてしまうからです。実際、図からフランス、ドイツ、インド、インドネシアを区別するのは非常に難しいでしょう。現実に色分けが出来る天才的な色覚を持つ読者ならこちらの方が情報も豊富であり、いいかも知れません。しかし、インドにおける感染者数の急増を示すには無駄な情報が多すぎます。そこでインドを除く国の色をグレーにまとめたものが図17.24です。

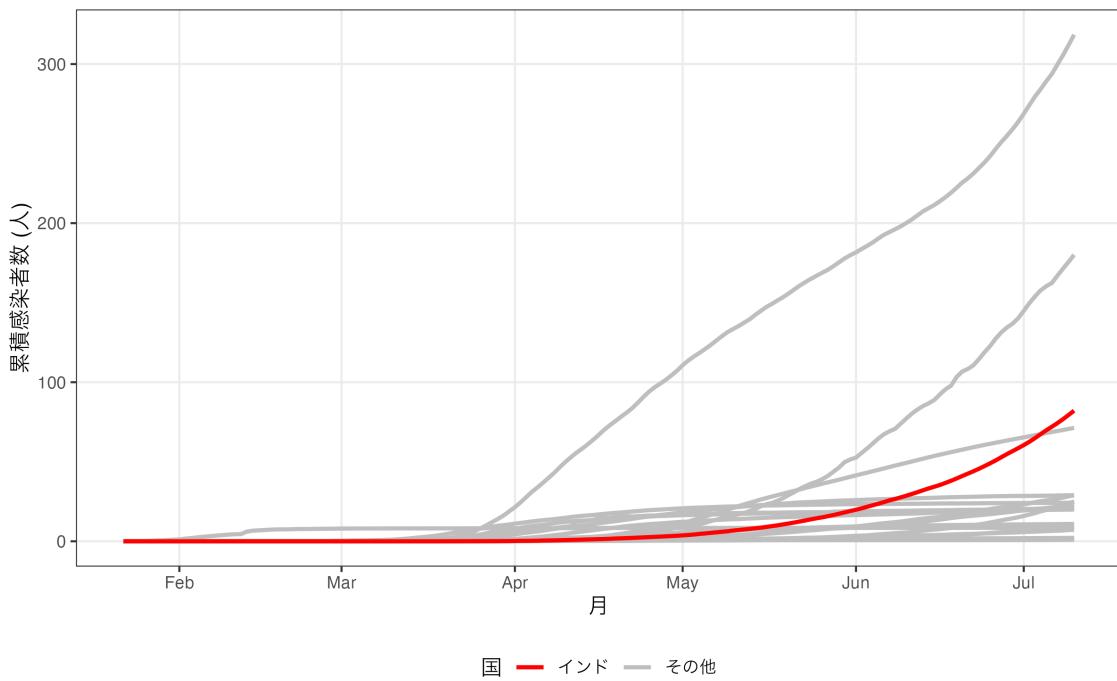


図 17.24: データ・インク比の例 (改善後)

こちらの方は多くの情報が失われています。アメリカや日本、韓国がどの線に該当するかが分かりません。しかし、図で示したいメッセージとは無関係でしょう。ここからもう一步踏み込んで、「ならばインド以外の線を消せばいいじゃん」と思う方もいるかも知れません。しかし、インドの線のみ残している場合、比較対象がなくなるため、急増していることを示しにくくなります。この場合、示したい情報の損失が生じるため、インド以外の国の線はデータ・インクに含まれます。

しかし、このデータ・インク比に基づく可視化は常に正しいとは言えません。そこでもう一つの例を紹介します。図 17.25 の左は Kuznicki and McCutcheon [1979] の論文に掲載された図であり、右は Tuft による改善案です。

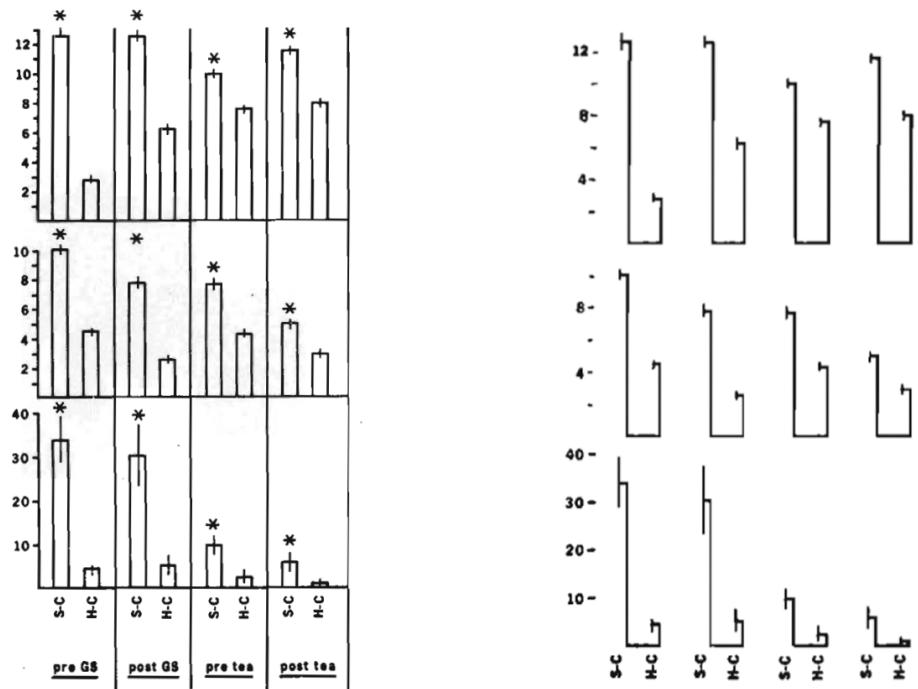


図 17.25: データ・インク比改善の例

確かに棒グラフは面を使用しており、高さを示すには線のみで十分かも知れません。また、エラー・バーも線の位置を若干ずらすことによって表現できます。それでは、読者の皆さんから見て、どのグラフが見やすいと思いますか。これについて興味深い研究結果があります。Inbar et al. [2007] は 87 人の学部生を対象に 3 つのグループに分けました。そして、学生たちは図 17.26 を「美しさ」、「明瞭さ」、「簡潔さ」の 3 つの面で評価し、最後にどの図が最も好きかを尋ねました。

- **グループ 1:** 図 17.26 の A と D を評価
- **グループ 2:** 図 17.26 の A と D を評価。
  - ただし、事前に Tufte スタイル (図 17.26 の D) の読み方について学習させる。
- **グループ 3:** 図 17.26 の A、B、C、D を評価

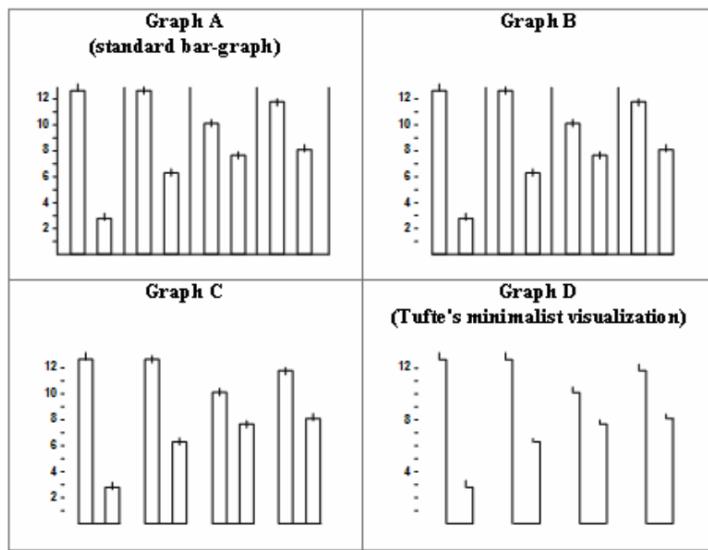


図 17.26: Inbar et al. [2007] から抜粋

皆さんもある程度は結果が予想できたかと思いますが、いずれのグループにおいても、Tufte が推奨する図 17.26 の D が「美しさ」、「明瞭さ」、「簡潔さ」のすべての点において最下位でした。また、どの図が最も好きかに対しても表 17.3 のような結果が得られました。

表 17.3: Inbar, Tractinsky, and Meyer (2007) の実験結果

Group	Graph A	Graph B	Graph C	Graph D
Group 1	24			3
Group 2	29			2
Group 3	14	3	12	0

図 17.26 の D はデータ・インク比の観点から見れば最も優れた図ですが、それが分かりやすさを意味するわけではありません。今は、人々の認知の観点からも図を評価するようになり、近年の可視化の教科書ではこれらに關しても詳しく触れているものが多いです。Tufte [2001] の本は可視化を勉強する人にとって必読の書かも知れませんが、これだけでは十分ではなく、近年の教科書も合わせて読むことをおすすめします。

### 17.4.2 3次元プロット

これまで見てきた全ての図は縦軸と横軸しか持たない2次元プロットです。しかし、実際の生活において3次元プロットを見にすることは珍しくないでしょう。3次元プロットの場合、縦軸と横軸以外に、高さ(深さ)といったもう一つの軸が含まれます。しかし、多くの場合、このもう一つの軸はグラフにおいて不要な場合が多いです。たとえば、図17.27を見てみましょう<sup>3)</sup>。これはCOVID19による定額給付金の給付済み金額を時系列で並べたものです。

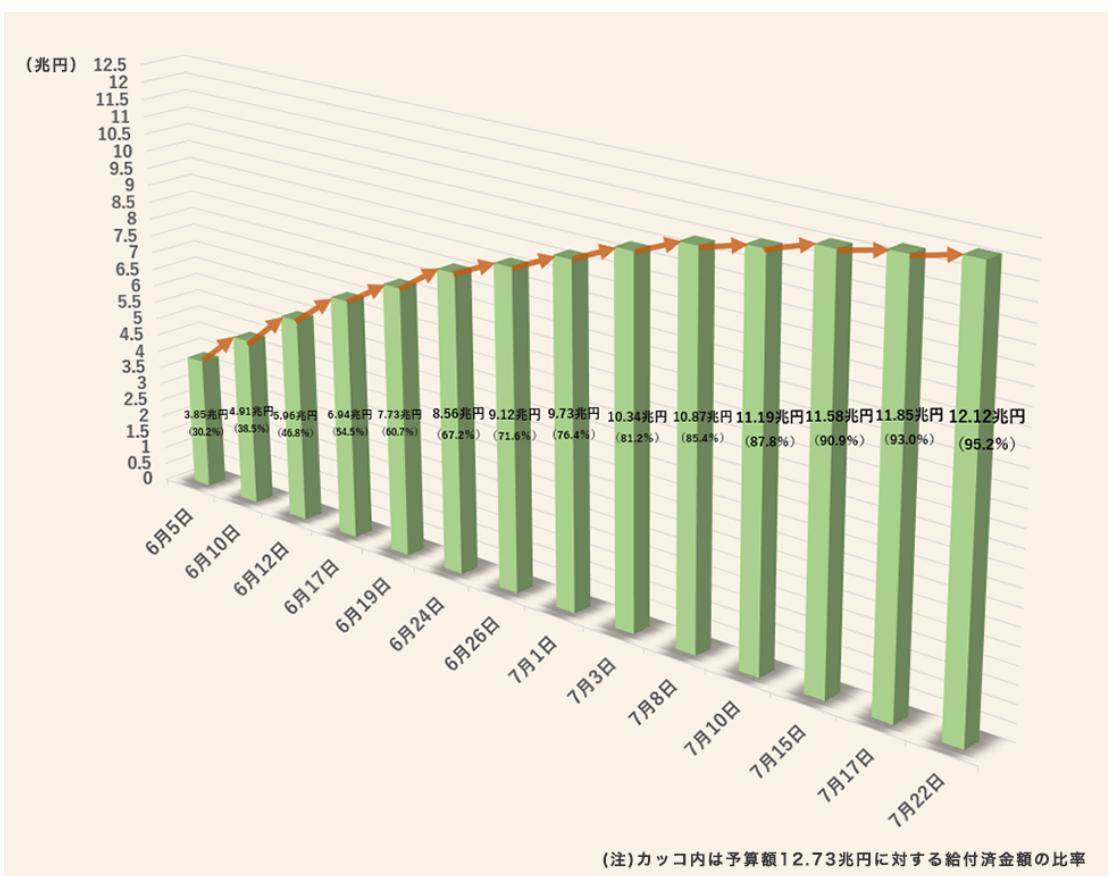


図17.27: 3次元プロットの例(1)

3) 総務省の定額給付金特設ホームページから取得（アクセス：2020年7月28日）

この図の目的は筆者にとってはよく分かりませんが、少なくとも給付金が順調（?）に配られているということですかね。その傾向を見るにはこの図は大きな問題はありません。しかし、細かい数値を見ようすると誤解が生じる可能性があります。たとえば、7月22日の給付済み額は12.12兆円です。しかし、図17.28を見ると、7月22日の棒は12兆円に達しておりません。なぜでしょうか。

これは棒と壁（?）との間隔が理由です。図17.28の右下にある青い円を見ればお分かりかと思いますが、棒が浮いています。この棒を壁（?）側に密着すると12の線を超えると考えられますが、このままだと12兆円に達していないのに12兆円超えてると、何かの入力ミスじゃないかと考えさせるかも知れません。



図17.28: 3次元プロットの罠

この図において3D要素の必要性は0と言えます。2次元の棒グラフ(図17.29)、または

折れ線グラフ(図17.30)の方がデータ・インク比の観点からも、分かりやすさからも優れていると言えるでしょう。

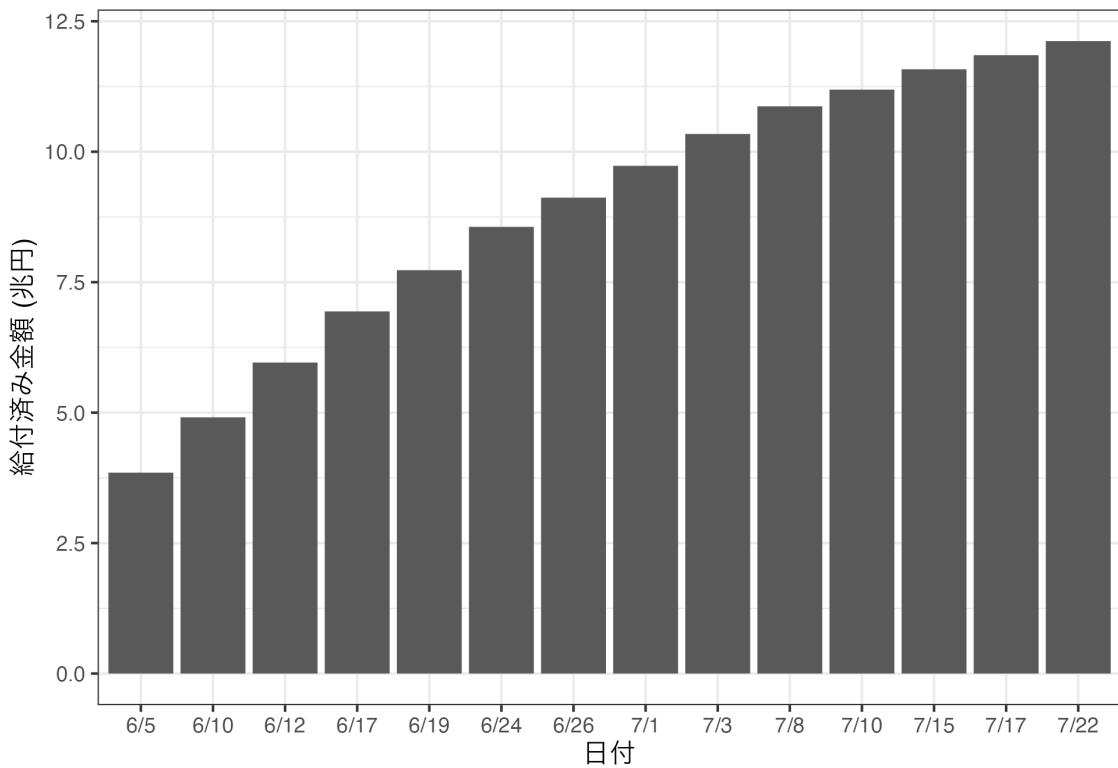


図 17.29: 図

reffig:visual1-3dplot-1 を 2 次元プロットに再構成した例 (棒グラフ)

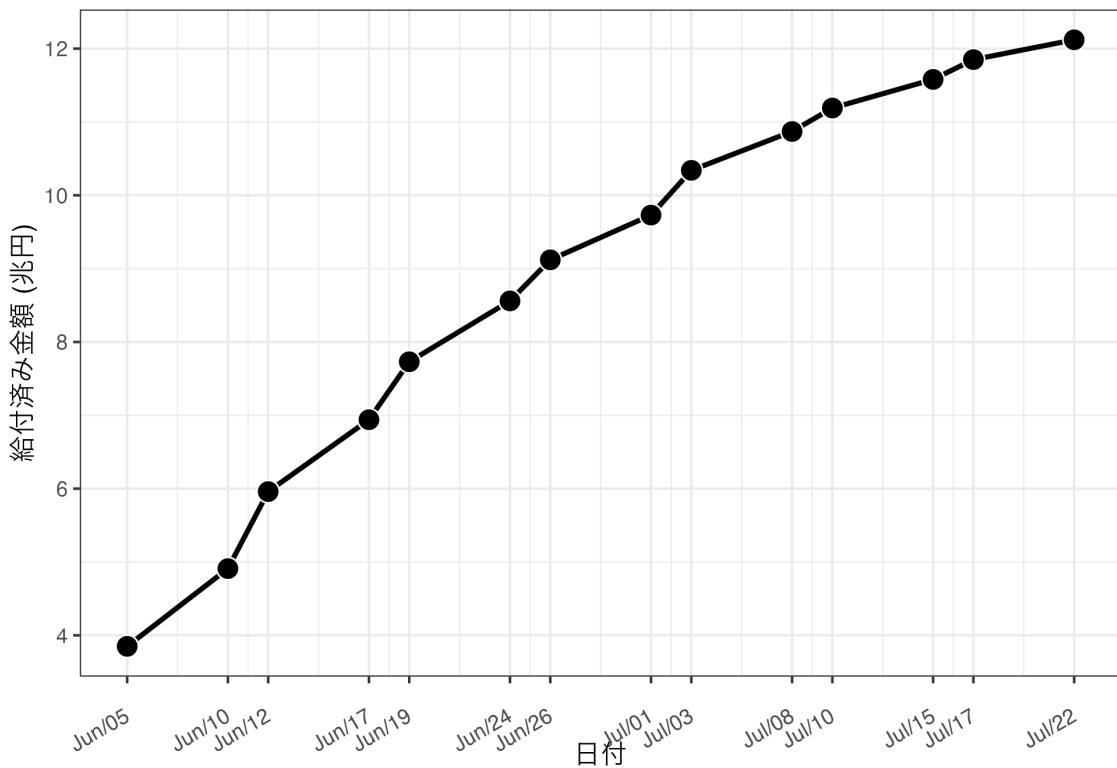


図 17.30: 図

reffig:visual1-3dplot-1 を 2 次元プロットに再構成した例 (折れ線グラフ)

ただ、総務省の図はまだマシかも知れません。世の中には誤解を招かすために作成された 3 次元プロットもあります。以下の図は早稲田アカデミーが作成した早慶高の合格者数を年度ごとに示した図です。2001 年は 754 人で 2012 年は 1494 人です。比較対象がないので本当に 12 年連続全国 No.1 かどうかは判断できませんが、2 倍近く増加したことは分かります。ただし、棒グラフの高さを見ると、2 倍どころか 3 倍程度に見えます。他にも一時期、合格者が減少した時期 (2002、2003 年) があるにもかかわらず、あまり目立ちません。これには 2 つの原因があります。それは (1) 多分、ベースラインが 0 人ではない、(2) 遠近法により遠いものは小さく見えることです。



図 17.31: 3 次元プロットの例 (2)

これを 2 次元棒グラフに直してものが図 17.32 です。こちらの方が合格者をより客観的に確認することができるでしょう。

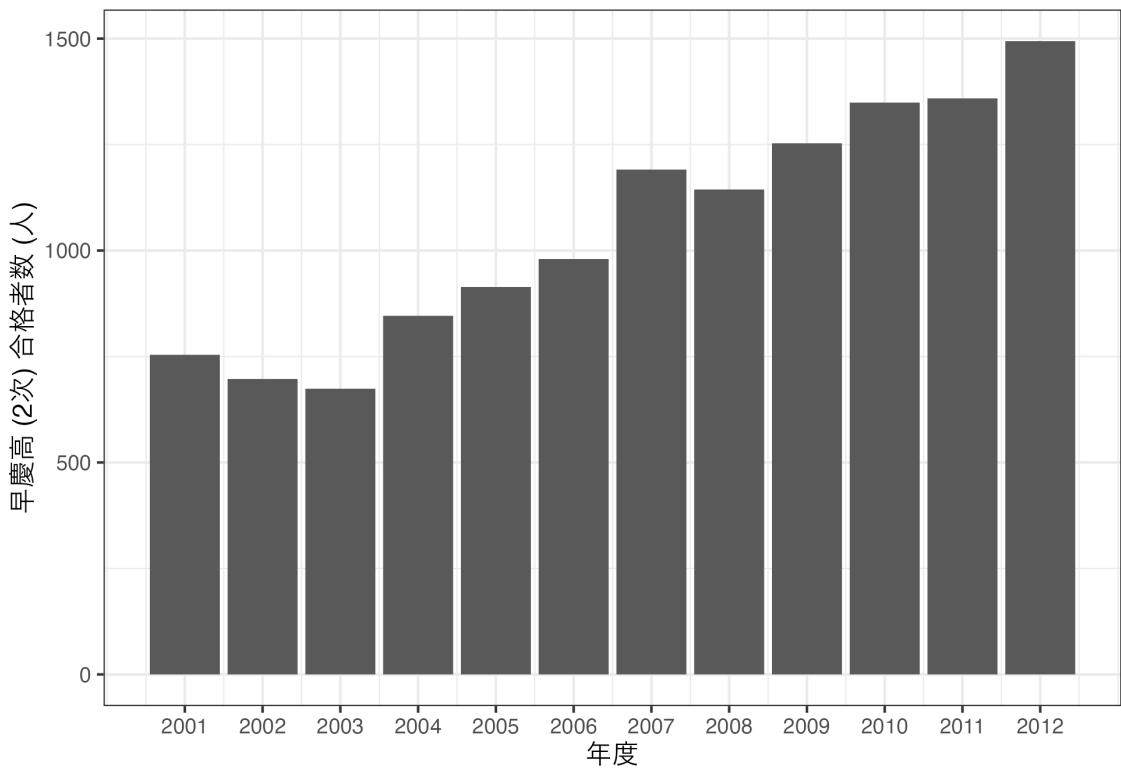


図 17.32: 図  
reffig:visual1-3dplot-5 を 2 次元プロットに再構成した例

むろん、3 次元プロットそのものが悪いわけではありません。むしろ、3 次元の方が解釈しやすい、見やすいケースもあるでしょう。それには共通点があり、深さというもう一つの軸も何らかの情報があるという点です。一方、ここでお見せしました 2 つの例の場合、深さは何の情報も持ちません。つまり、データ・インク比の観点から見れば望ましくない図です。



## 第 18 章

# 可視化 [基礎]

### 18.1 本章の内容

前章では{ggplot2}の仕組みおよびグラフィックの文法と良いグラフについて説明しました。本章では実際に簡単なグラフを作りながら{ggplot2}に慣れて頂きたいと思います。{ggplot2}で作れる図の種類は非常に多いですが、本章では、データサイエンスで頻繁に利用される以下の 5 つのプロットの作り方を紹介します。

1. 棒グラフ
2. ヒストグラム
3. 箱ひげ図
4. 散布図
5. 折れ線グラフ

その他の図や、図の細かい修正については第 19 章で解説します。

### 18.2 実習用データ

実習の前に本章で使用するデータと{ggplot2}パッケージが含まれている `tidyverse` を読み込みます。

- データ 1: 国家別民主主義および政治的自由データ

- データ2: COVID-19 データ

COVID19\_Worldwide.csvの場合、普通に読み込むとTest\_DayとTest\_Totalが数値型であるにも関わらず、logical型変数として読み込まれます。read\_csvはデフォルトだと最初の100行までのデータからデータ型を判断しますが、COVID19\_Worldwide.csvの場合、Test\_DayとTest\_Totalの最初の100要素は全て欠損しており、判断不可となるため、自動的にlogical型として判断します。これを避けるために、guess\_max = 10000を追加します。これは「データ型を判断するなら10000行までは読んでから判断しろ」という意味です。

```
1 pacman::p_load(tidyverse)
2
3 Country_df <- read_csv("Data/Countries.csv")
4 COVID19_df <- read_csv("Data/COVID19_Worldwide.csv", guess_max = 10000)
```

これらの変数はSONGが適当にインターネットなどで集めたデータであり、あくまでも実習用データとしてのみお使い下さい。各変数の詳細は以下の通りです。

---

## 18.3 日本語が含まれた図について

(以下は作成中の内容)

{ggplot2}で作成した図に日本語が含まれている場合、日本語が正しく表示されない場合があります。また、RStudioのPlotsペインには日本語が表示されていても、PDF/PNG/JPGなどのファイルとして出力した場合、表示されないケースもあります。{ggplot2}の使い方を解説する前に、ここでは日本語が正しく表示/出力されない場合の対処法について紹介します。

### 18.3.1 macOSの場合

一つ一つの図にフォント族(font family)を指定

表 18.1: 国家別民主主義および政治的自由データの詳細

変数名	説明	備考
Country	国名	
Population	人口	人
Area	面積	km\$^2\$
GDP	国内総生産 (GDP)	100 万米ドル
PPP	GDP (購買力平価):	100 万米ドル
GDP_per_capita	一人あたり GDP	米ドル
PPP_per_capita	一人あたり GDP (購買力平価)	米ドル
G7	G7 構成国	1: 構成国, 0: 構成国以外
G20	G20 構成国	1: 構成国, 0: 構成国以外
OECD	OECD 構成国	1: 構成国, 0: 構成国以外
HDI_2018	人間開発指数	2018 年基準
Polity_Score	民主主義の程度	Polity IV から; -10: 権威主義～10: 民主主義
Polity_Type	民主主義の程度 (カテゴリ)	
FH_PR	政治的自由の指標	2020 年基準; Freedom House から
FH_CL	市民的自由の指標	2020 年基準; Freedom House から
FH_Total	政治的自由と市民的自由の合計	2020 年基準; Freedom House から
FH_Status	総合評価	F: 完全な自由; PF: 一部自由; NF: 不自由
Continent	大陸	

```
theme(text = element_text(family = "HiraginoSans-W3"))
```

または、`theme_bw()` や `theme_minimal()` など、特定のテーマを使用するのであればそのテーマのレイヤーにフォントを指定することもできる。たとえば、`{ggplot2}` のデフォルトテーマである `gray` テーマを使用するなら、`{ggplot2}` のオブジェクトに以下のようなレイヤーを追加する。

表 18.2: COVID-19 データの詳細

変数名	説明
ID	ID
Country	国名
Date	年月日
Confirmed_Day	COVID-19 新規感染者数 (人)
Confirmed_Total	COVID-19 累積感染者数 (人)
Death_Day	COVID-19 新規死者数 (人)
Death_Total	COVID-19 累積死者数 (人)
Test_Day	COVID-19 新規検査数 (人)
Test_Total	COVID-19 累積検査数 (人)

```
theme_gray(base_familiy = "HiraginoSans-W3")
```

最初に宣言しておく

```
theme_update(text = element_text(family = "HiraginoSans-W3"))
```

ただし、`geom_text()`、`geom_label()`、`annotate()` のような文字列を出力する幾何オブジェクトを使用する際は、各レイヤーごとにフォント族を指定する必要がある (`family = "HiraginoSans-W3"`)。

ファイル (PDF) として出力する際は

```
quartz(file = "出力するファイル名", type = "pdf", width = 幅, height = 高さ)
{ggplot2}で作成した図のオブジェクト名
dev.off()
```

### 18.3.2 Windows の場合

#### 18.3.3 {ragg}の使用

```
pacman::p_install(ragg)
```

以下のやり方で Plots ペインでの日本語問題は解決

1. RStudio の Tools>Global Options...
2. 左の General>右側上段の Graphics タブ
3. Graphic Device の Backend で AGG を選択

RMarkdown (HTML 出力) の場合、最初のチャンクに以下のコードを追加する。PNG の場合、解像度が低いと見た目があまり良くないため、dpi は 150 以上がオススメ

```
knitr::opts_chunk$set(dev = "ragg_png", dpi = 300)
```

ファイルとして出力したい場合は (PNG)

- **方法 1:** ragg::agg\_png() を使用
- **方法 2:** ggsave() に device = ragg::agg\_png を指定

ファイルとして出力したい場合は (PDF)

---

## 18.4 棒グラフ

棒グラフについては以下の 2 つのタイプについて説明します。

1. ある変数の数の表す棒グラフ
2. 各グループの統計量を表す棒グラフ

前者は「データ内にアフリカのケースはいくつあるか、アジアの行はいくつあるか」のようなものであり、後者は「大陸ごとの民主主義の度合いの平均値はいくつか」を出力するグラフです。

### 18.4.1 ケース数のグラフ

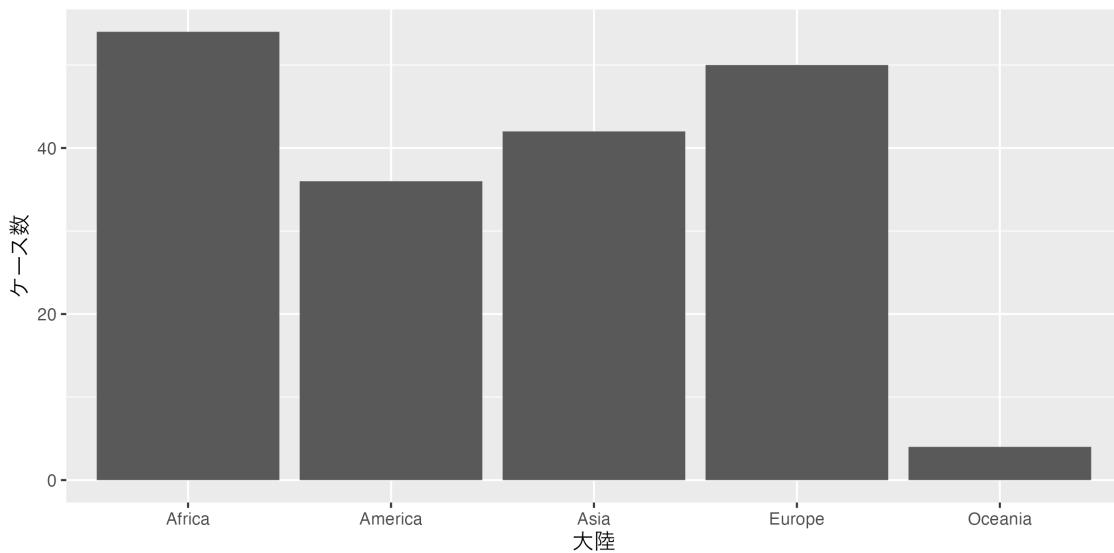
まず、Country\_df の Continent 変数における各値の頻度数を棒グラフとして出してみましょう。表としてまとめる簡単な方法は table() 関数があります。

```
1 table(Country_df$Continent)

## 
##   Africa America     Asia   Europe Oceania
##      54      36      42      50       4
```

ケース数の棒グラフは、大陸名を横軸に、ケース数を縦軸にしたグラフです。それではグラフを作ってみます。データは Country\_df であり、使う幾何オブジェクトは geom\_bar() です。ここで必要な情報は横軸、つまり大陸 (Continent) のみです。縦軸も「ケース数」という情報も必要ですが、{ggplot2}が勝手に計算してくれるので、指定しません。また、labs() レイヤーを追加します。labs() レイヤーはマッピング要素のラベルを指定するものです。これを指定しない場合、aes() 内で指定した変数名がそのまま出力されます。

```
1 Country_df %>%
2   ggplot() +
3   geom_bar(aes(x = Continent)) +
4   labs(x = "大陸", y = "ケース数")
```



これで初めての{ggplot2}を用いたグラフが完成しましたね！

#### 18.4.2 記述統計量のグラフ

次は記述統計量のグラフを出してみます。たとえば、大陸ごとに民主主義の指標の1つである Polity Score の平均値を図示します。まずは、dplyr を使って、大陸ごとに Polity Score (Polity\_Score) の平均値を計算し、Bar\_df1 という名で保存します。

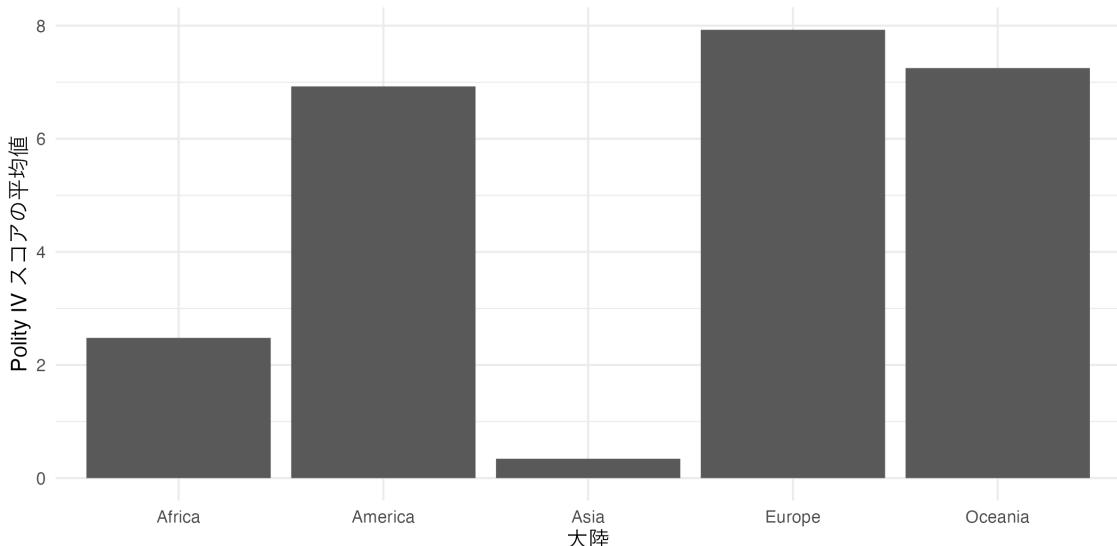
```
1 Bar_df1 <- Country_df %>%
2   group_by(Continent) %>%
3   summarise(Democracy = mean(Polity_Score, na.rm = TRUE),
4             .groups    = "drop")
5
6 Bar_df1
```

```
## # A tibble: 5 x 2
##   Continent Democracy
##   <chr>        <dbl>
## 1 Africa        2.48
## 2 America       6.93
## 3 Asia         0.342
```

```
## 4 Europe      7.93
## 5 Oceania     7.25
```

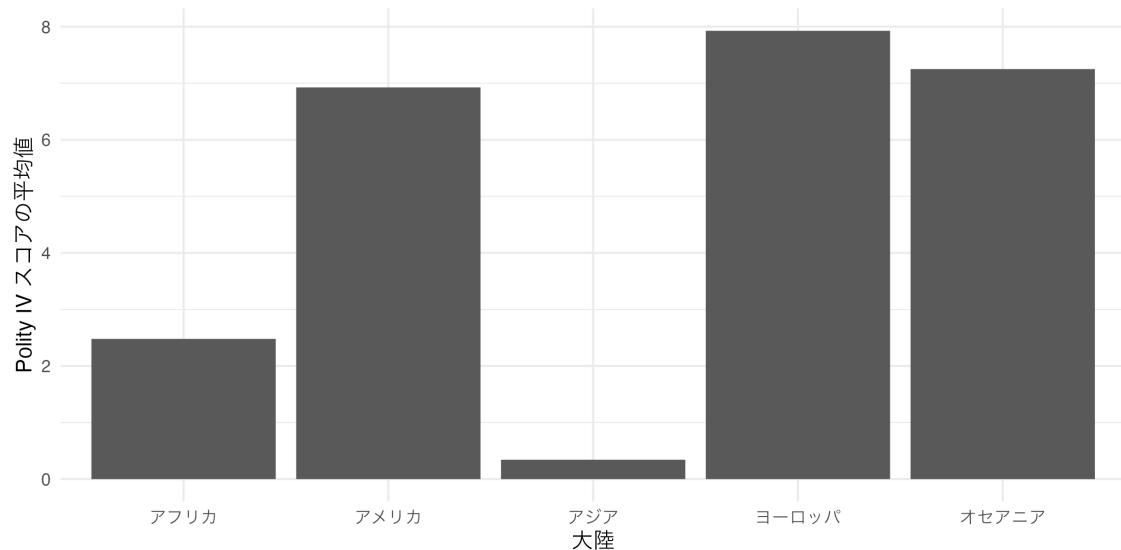
それでは、この Bar\_df1 を基にグラフを作りますが、今回は縦軸の情報も必要です。横軸は Continent、縦軸は Democracy 変数に指定します。そして、重要なものとして stat 引数を指定します。これはマッピングと関係なく、棒グラフの性質に関係するものなので、aes() の外側に位置します。これを指定しない場合、geom\_bar() は基本的にケース数を計算し、図示します。Passenger の値そのものを縦軸にしたい場合は stat = "identity" を指定します。後は、先ほどの棒グラフと同じです。

```
1 Bar_df1 %>%
2   ggplot() +
3   geom_bar(aes(x = Continent, y = Democracy), stat = "identity") +
4   labs(x = "大陸", y = "Polity IV スコアの平均値") +
5   theme_minimal()
```



考えてみれば、大陸名が英語になっていますね。図内の言語は統一するのが原則であり、図の言語は論文やレポート、報告書の言語とも一致させるべきです。ここは Bar\_df1 の Continent 列の値を日本語に置換するだけでいいので、recode() 関数を使います。recode() の使い方は第 13.3 章を参照してください。また、順番はローマ字順にしたいので、fct\_inorder() を使って、Bar\_df1 における表示順で factor 化を行います。

```
1 Bar_df1 %>%
2   mutate(Continent = recode(Continent,
3                             "Africa"    = "アフリカ",
4                             "America"   = "アメリカ",
5                             "Asia"      = "アジア",
6                             "Europe"    = "ヨーロッパ",
7                             .default    = "オセアニア"),
8   Continent = fct_inorder(Continent)) %>%
9   ggplot() +
10  geom_bar(aes(x = Continent, y = Democracy), stat = "identity") +
11  labs(x = "大陸", y = "Polity IV スコアの平均値") +
12  theme_minimal()
```



### 18.4.3 次元を追加する

記述統計量のグラフを見るとマッピング要素は 2 つであり、これは図が 2 つの次元、つまり大陸と Polity IV スコアの平均値で構成されていることを意味します。記述統計量のグラフはマッピング要素は 1 つですが、ケース数という次元が自動的に計算されるため 2 次元です。ここにもう一つの次元を追加してみましょう。たとえば、大陸ごとの Polity

IVスコアの平均値を出しますが、これを更にOECD加盟有無で分けてみましょう。そのためには、まず大陸とOECD加盟有無でグループを分けてPolity IVスコアの平均値を計算する必要があります。

```
1 Bar_df2 <- Country_df %>%
2   group_by(Continent, OECD) %>%
3   summarise(Democracy = mean(Polity_Score, na.rm = TRUE),
4             .groups    = "drop")
5
6 Bar_df2
```

```
## # A tibble: 9 x 3
##   Continent OECD Democracy
##   <chr>     <dbl>     <dbl>
## 1 Africa      0     2.48
## 2 America     0     6.55
## 3 America     1     8.6
## 4 Asia        0    -0.314
## 5 Asia         1      8
## 6 Europe      0     5.87
## 7 Europe      1     9.12
## 8 Oceania     0      4.5
## 9 Oceania     1     10
```

続いて、ContinentとOECD列を日本語に直します。また、順番を指定するためにfactor化しましょう。

```
1 Bar_df2 <- Bar_df2 %>%
2   mutate(Continent = recode(Continent,
3                             "Africa"    = "アフリカ",
4                             "America"   = "アメリカ",
5                             "Asia"      = "アジア",
6                             "Europe"    = "ヨーロッパ",
7                             .default    = "オセアニア"),
```

```
8     Continent = fct_inorder(Continent),  
9     OECD      = recode(OECD,  
10                    "0" = "OECD 非加盟国",  
11                    "1" = "OECD 加盟国"),  
12     OECD      = fct_inorder(OECD))  
13  
14 Bar_df2
```

```
## # A tibble: 9 x 3  
##   Continent OECD           Democracy  
##   <fct>     <fct>         <dbl>  
## 1 アフリカ   OECD 非加盟国     2.48  
## 2 アメリカ   OECD 非加盟国     6.55  
## 3 アメリカ   OECD 加盟国      8.6  
## 4 アジア     OECD 非加盟国    -0.314  
## 5 アジア     OECD 加盟国      8  
## 6 ヨーロッパ OECD 非加盟国     5.87  
## 7 ヨーロッパ OECD 加盟国      9.12  
## 8 オセアニア OECD 非加盟国     4.5  
## 9 オセアニア OECD 加盟国      10
```

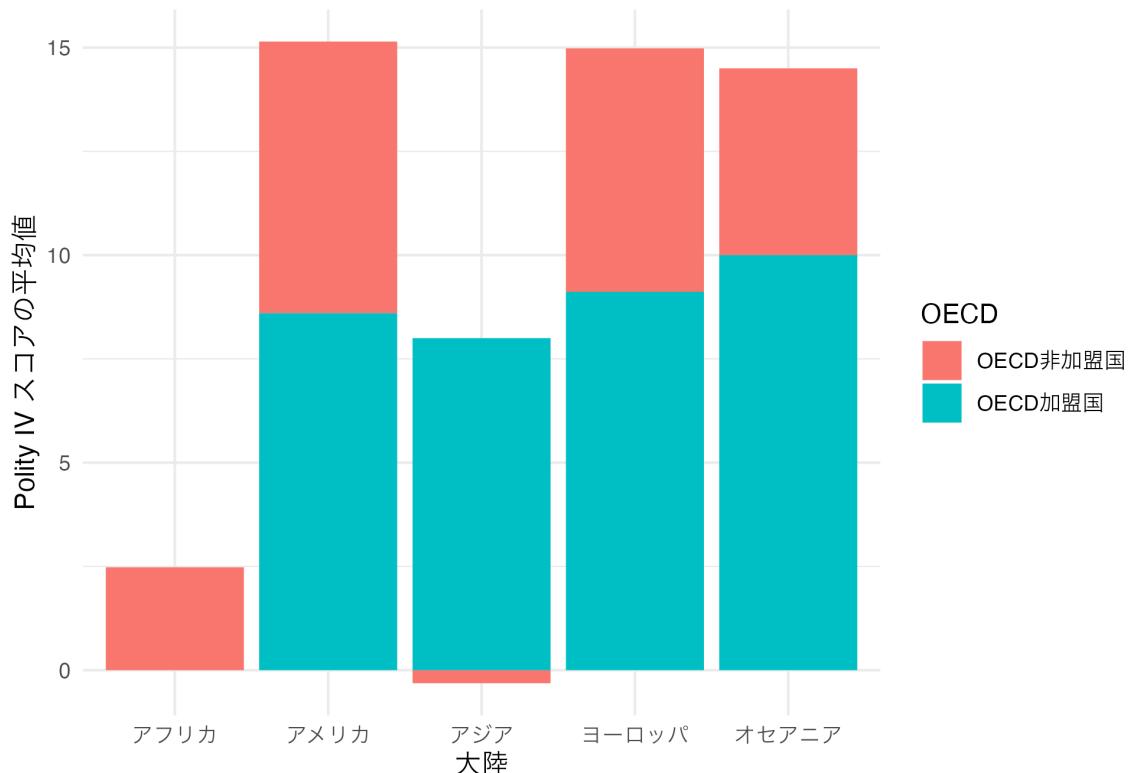
これで作図の準備ができました。それではこの新しい次元である OECD をどのように表現すれば良いでしょうか。Continent は横軸の位置で表現され、Democracy は縦軸の位置として表現されているため、x と y 以外の要素を考えてみましょう。棒グラフの場合、もう一つの軸が追加されたらそれは、棒の色です。棒の色は fill で指定できます。color でないことに注意してください。color も指定可能ですが、これは棒の色ではなく、棒を囲む線の色であり、通常は「なし」となっています。それでは fill を aes() 内に書き加えます。

```
1 Bar_df2 %>%  
2   ggplot() +  
3   geom_bar(aes(x = Continent, y = Democracy, fill = OECD),  
4             stat = "identity") +
```

```

5   labs(x = "大陸", y = "Polity IV スコアの平均値") +
6   theme_minimal()

```



なんか思ったものと違うものが出てきました。たとえば、アメリカ大陸の場合、Polity IV スコアの平均値が約 15 ですが、明らかにおかしいです。なぜなら Polity IV スコアの最大値は 10 だからです。これは 2 つの棒が積み上げられているからです。アメリカ大陸において OECD 加盟国の平均値は 8.6、非加盟国のそれは 6.55 であり、足したら 15.15 になります。これをずらすためには `position` を設定する必要があります。しかし、`position` というのは `Bar_df2` の何かと変数の値を表すわけではないため、`aes()` の外側に入れます。そして、その値ですが、ここでは "dodge" を指定します。これは棒の位置が重ならないように調整することを意味します。この `position` のデフォルト値は "stack" であり、言葉通り「積み上げ」です。

```

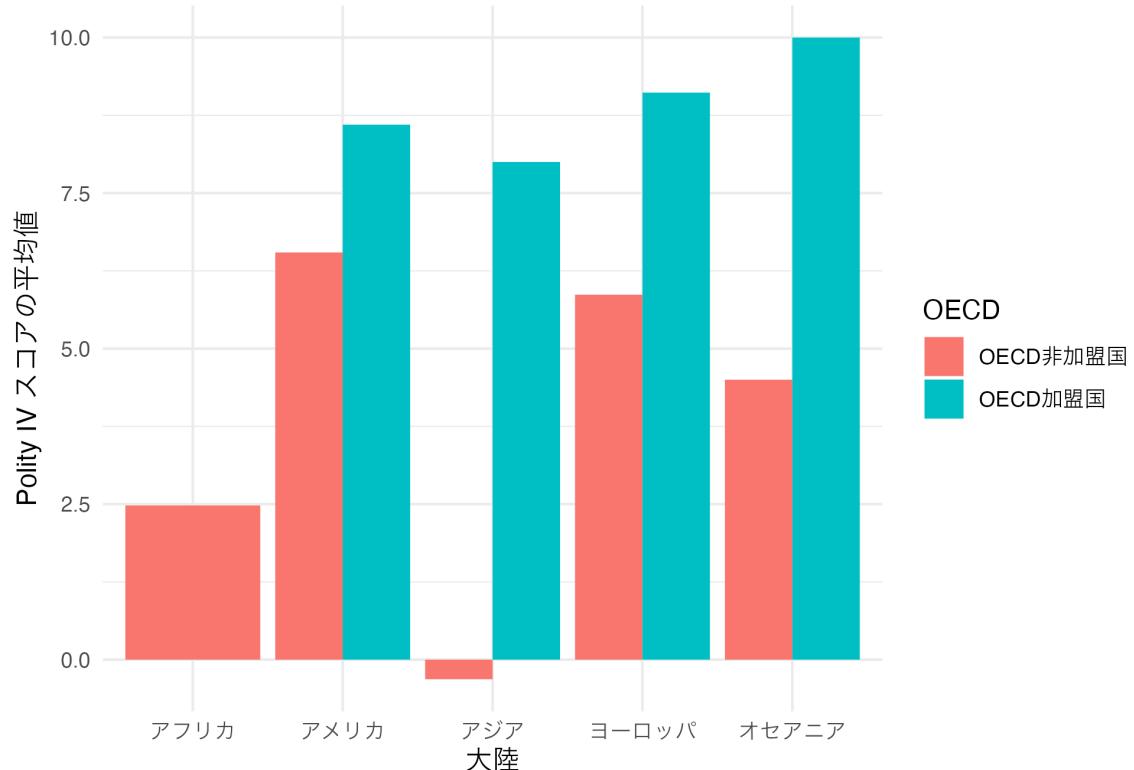
1 Bar_df2 %>%
2   ggplot() +

```

```

3   geom_bar(aes(x = Continent, y = Democracy, fill = OECD),
4           stat = "identity", position = "dodge") +
5   labs(x = "大陸", y = "Polity IV スコアの平均値") +
6   theme_minimal()

```



これで私たちが期待した図が出来上がりしました。「"dodge"の方が普通なのになぜデフォルトが"stack"か」と思う方もいるかも知れませんが、実は"stack"も頻繁に使われます。それはケース数のグラフにおいてです。

たとえば、大陸ごとにOECD加盟/非加盟国を計算してみましょう。

```

1 Bar_df3 <- Country_df %>%
2   group_by(Continent, OECD) %>%
3   summarise(N = n(),
4             .groups = "drop") %>%
5   mutate(Continent = recode(Continent,

```

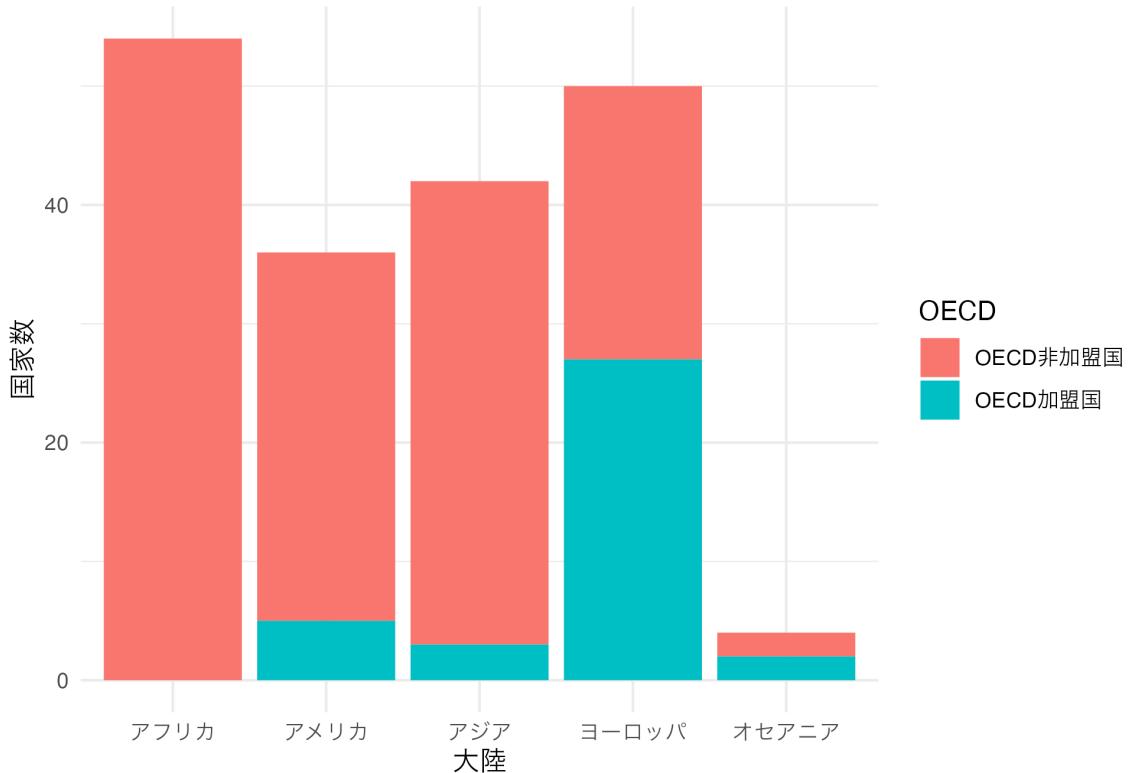
```
6           "Africa"    = "アフリカ",
7           "America"   = "アメリカ",
8           "Asia"       = "アジア",
9           "Europe"     = "ヨーロッパ",
10          .default    = "オセアニア"),
11          Continent = fct_inorder(Continent),
12          OECD      = recode(OECD,
13                         "0" = "OECD 非加盟国",
14                         "1" = "OECD 加盟国"),
15          OECD      = fct_inorder(OECD))
16
17 Bar_df3
```

```
## # A tibble: 9 x 3
##   Continent OECD          N
##   <fct>     <fct>      <int>
## 1 アフリカ   OECD 非加盟国    54
## 2 アメリカ   OECD 非加盟国    31
## 3 アメリカ   OECD 加盟国      5
## 4 アジア     OECD 非加盟国    39
## 5 アジア     OECD 加盟国      3
## 6 ヨーロッパ OECD 非加盟国    23
## 7 ヨーロッパ OECD 加盟国      27
## 8 オセアニア OECD 非加盟国    2
## 9 オセアニア OECD 加盟国      2
```

これを可視化したのが以下の図です。

```
1 Bar_df3 %>%
2   ggplot() +
3   geom_bar(aes(x = Continent, y = N, fill = OECD),
4             stat = "identity") +
5   labs(x = "大陸", y = "国家数") +
```

```
6     theme_minimal()
```



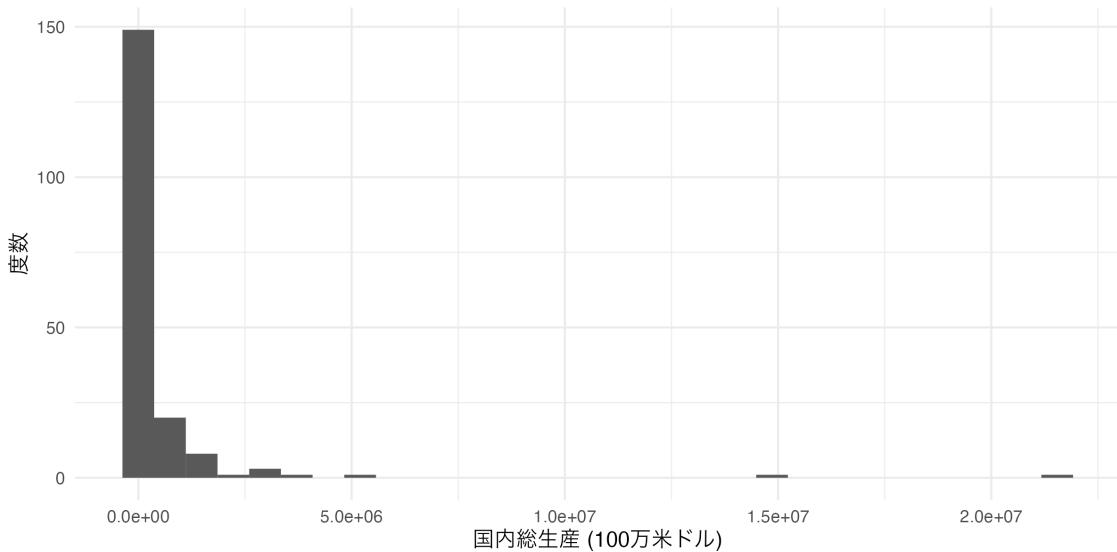
この図は `position = "dodge"` でも良いと思いますが、大陸内の比率を考えるなら `position = "stack"` でも問題ないでしょう。また、積み上げグラフの特性上、大陸ごとの国数の合計も一瞬で判別できるといった長所もあります。`position = "dodge"` だと、それが難しいですね。むろん、積み上げ棒グラフはベースラインが一致したいため、避けるべきという人も多いですし、著者 (SONG) も同意見です。どの図を作成するかは分析者の責任で判断しましょう。

## 18.5 ヒストグラム

ヒストグラムは棒グラフと非常に形が似ていますが、横軸が大陸のような離散変数ではなく、連続変数であるのが特徴です。連続変数をいくつの区間 (階級) に分け、その区間

内に属するケース数 (度数) を示したのが度数分布表、そして度数分布表をかしかしたものがヒストグラムです。連続変数を扱っているため、棒間に隙間がありません。それでもケース数の棒グラフと非常に似通っているため、マッピングの仕方も同じです。異なるのは幾何オブジェクトが `geom_bar()` でなく、`geom_histogram()` に変わるくらいです。世界の富がどのように分布しているかを確認するために、`Country_df` の GDP のヒストグラムを作ってみます。

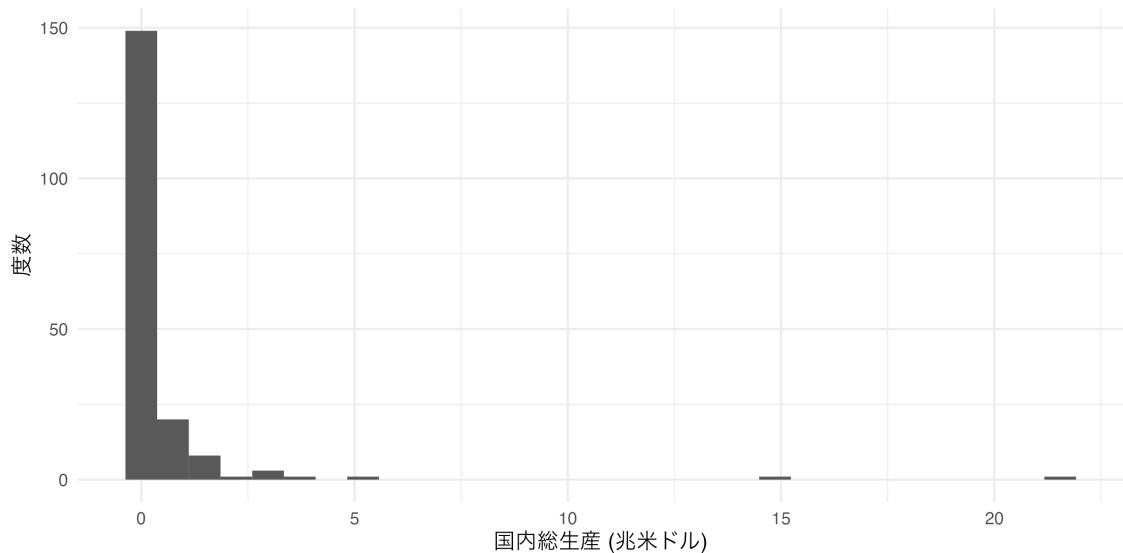
```
1 Country_df %>%
2   ggplot() +
3   geom_histogram(aes(x = GDP)) +
4   labs(x = "国内総生産 (100万米ドル)", y = "度数") +
5   theme_minimal()
```



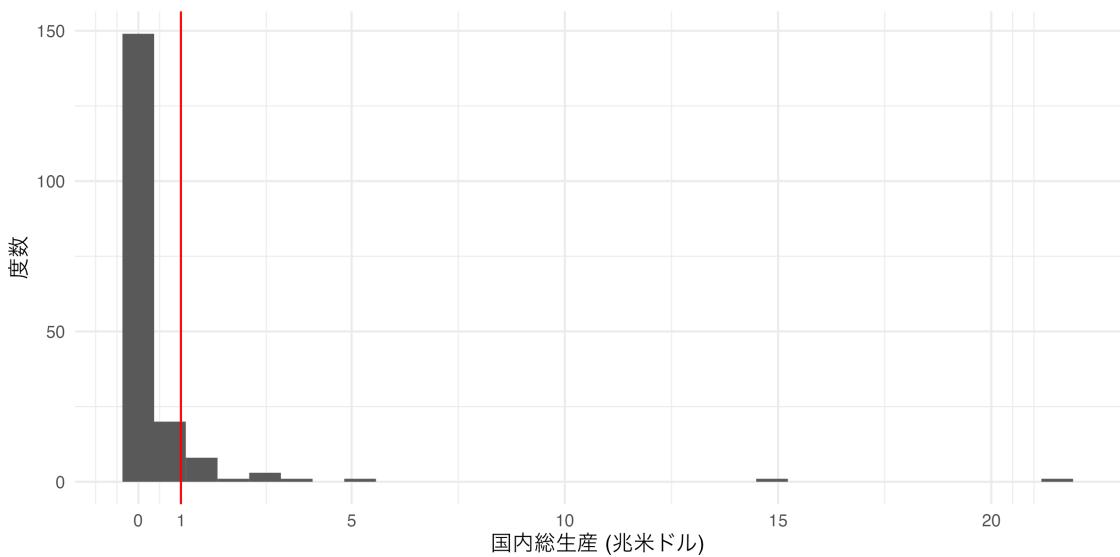
ケース数の棒グラフのコードとほぼ同じです。横軸の数値が  $2.0 \times 10^7$  になっているのは `2 \times 10^7`、つまり 2 千万を意味します。普通に表記すると 20,000,000 になりますね。また、GDP の単位は 100 万ドルであるため、実際の GDP は 20 兆ドルになります。つまり、今のヒストグラムにおいて横軸の目盛りは 5 兆ドルになっています。この軸の数値を「0, 5e+06, 1e+07, 1.5e+07, 2e+07」から「0, 5, 10, 15, 20」にし、X 軸のラベルを「国内総生産 (100万米ドル)」から「国内総生産 (兆米ドル)」に替えてみましょう。ここで使うのは `scale_x_continuous()` 関数です。これは横軸 (X 軸) が連続変数

(continuous) の場合のスケール調整関数です。目盛りの再調整には `breaks` と `labels` 引数が必要です。`breaks` は新しい目盛りの位置、`labels` は目盛りに表記する値です。それぞれベクトルが必要であり、`breaks` と `labels` の実引数の長さは必ず一致する必要があります。また、`breaks` は数値型ベクトルですが、`labels` は数値型でも文字型でも構いません。

```
1 Country_df %>%
2   ggplot() +
3   geom_histogram(aes(x = GDP)) +
4   labs(x = "国内総生産 (兆米ドル)", y = "度数") +
5   scale_x_continuous(breaks = c(0, 5000000, 10000000, 15000000, 20000000),
6                      labels = c(0, 5, 10, 15, 20)) +
7   theme_minimal()
```

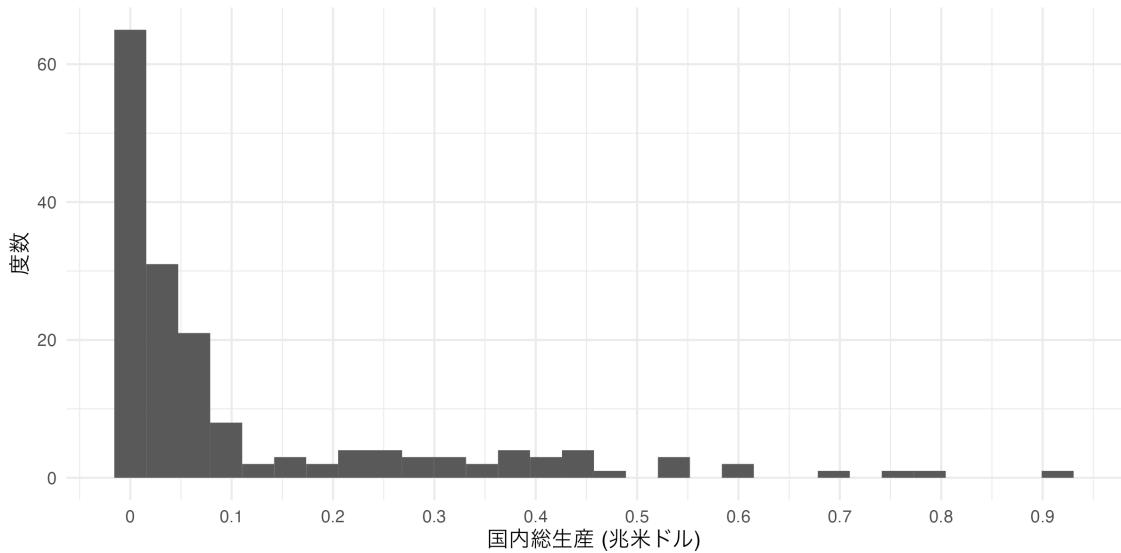


これで一通りヒストグラムが完成しました。ほんの一部の国は非常に高いGDPを誇っていることが分かります。GDPが10兆ドル以上の国はアメリカと中国のみであり、5兆ドルを国まで拡大しても日本が加わるだけです。そもそも1兆ドルを超える国はデータには16カ国しかなく、90%以上の国が図の非常に狭い範囲内(0~1兆ドル)に集まっていることが分かります。

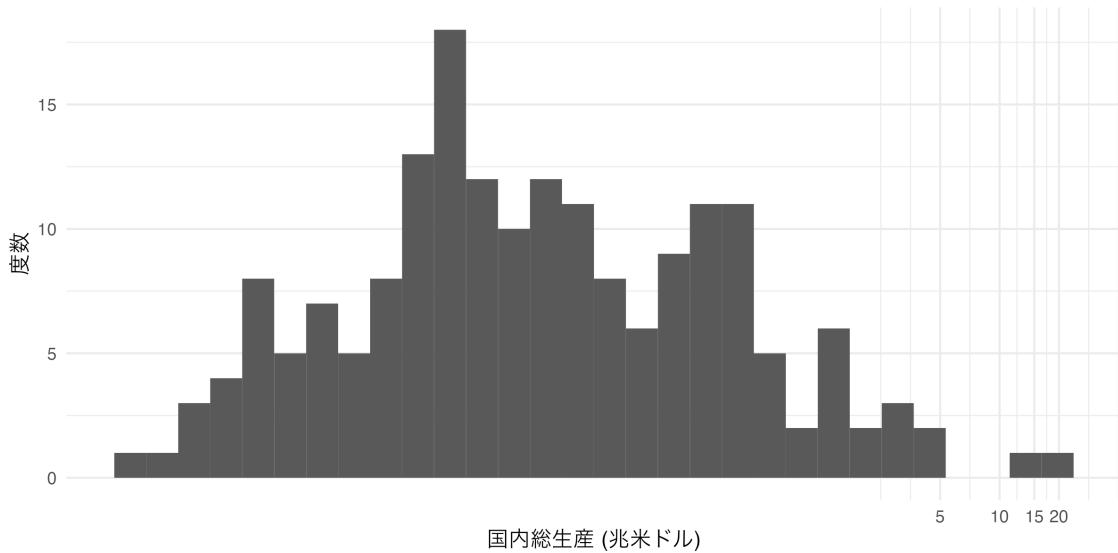


この場合、2つの方法が考えられます。1つ目は方法は情報の損失を覚悟した上で、GDP が1兆ドル未満の国でヒストグラムを書く方法です。これはデータを `ggplot()` 関数を渡す前に `filter()` を使って、GDP が100万未満のケースに絞るだけで出来ます。ただし、横軸の最大値が2000万ではなく、100万になるため、目盛りを調整した方が良いでしょう。

```
1 Country_df %>%
2   filter(GDP < 1000000) %>%
3   ggplot() +
4   geom_histogram(aes(x = GDP)) +
5   labs(x = "国内総生産 (兆米ドル)", y = "度数") +
6   scale_x_continuous(breaks = seq(0, 1000000, 100000),
7                      labels = seq(0, 1, 0.1)) +
8   theme_minimal()
```



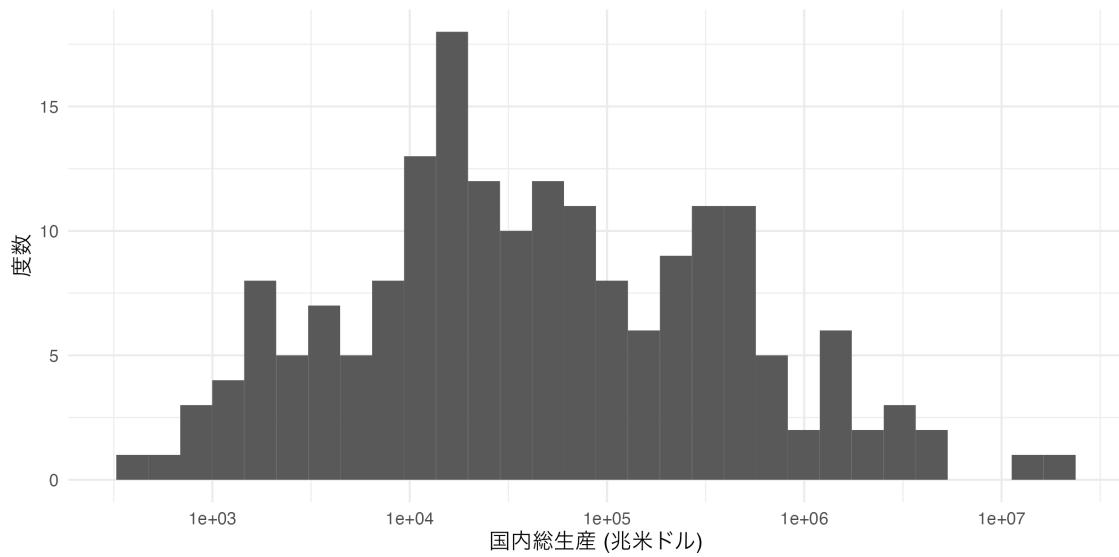
2つ目の方法は横軸を対数化することです。GDP を底 10 の対数化（常用対数）をすると、10兆のような非常に大きい値があっても比較的に狭い範囲内にデータを収めることができます。たとえば、10を常用対数化すると1, 1000は3, 10000000は7になります。自然対数（底が  $e$ ）も可能ですが、「読む」ためのグラフとしては底が10の方が読みやすいでしょう。横軸の変数が対数化されるということは、横軸のスケールを対数化することと同じです。そのためには `scale_x_continuous()` 内に `trans` 引数を指定し、"log10" を渡します。



対数化することによって GDP の分布が綺麗な形になりました。対数化すると横軸における目盛りの間隔が等間隔でないことに注意すべきです。0 から 5 兆ドルの距離はかなり広めですが、5 兆から 10 兆までの距離は短くなり、10 兆から 15 兆までの距離は更に短くなります。したがって、この図から「世界の GDP は鐘型に分布している」と解釈することは出来ません。分布を可視化するには対数化する前の図が適します。

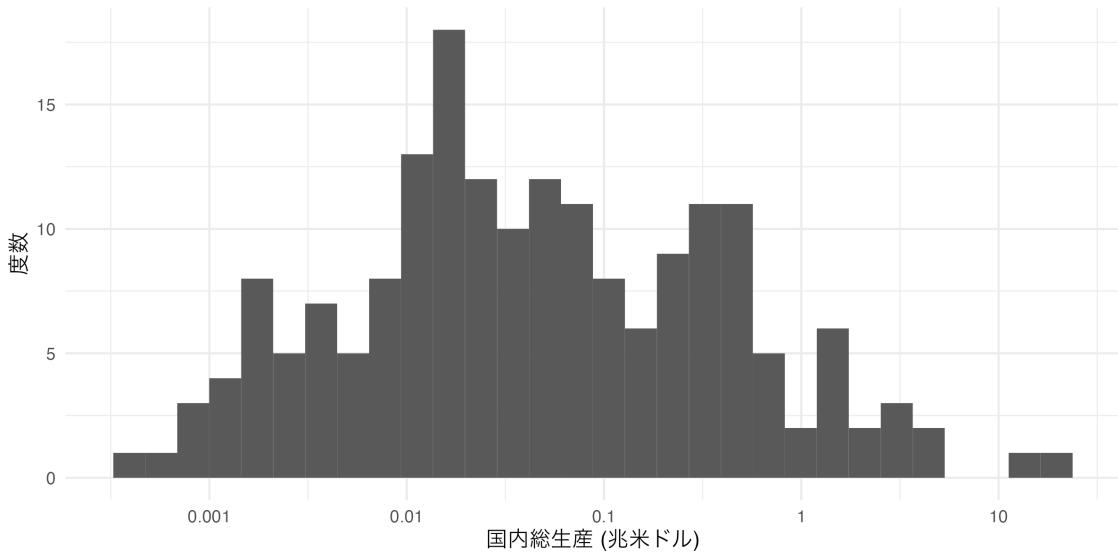
対数化のもう一つの方法は `scale_x_continuous()` の代わりに `scale_x_log10()` を使うことです。使い方は `scale_x_continuous()` と同じですが、`trans = "log10"` の指定は不要です。

```
1 Country_df %>%
2   ggplot() +
3   geom_histogram(aes(x = GDP)) +
4   labs(x = "国内総生産 (兆米ドル)", y = "度数") +
5   scale_x_log10() +
6   theme_minimal()
```



横軸を修正するには `scale_x_continuous()` と同様、`breaks` と `labels` 引数を指定します。

```
1 Country_df %>%
2   ggplot() +
3   geom_histogram(aes(x = GDP)) +
4   labs(x = "国内総生産 (兆米ドル)", y = "度数") +
5   scale_x_log10(breaks = c(0, 1000, 10000, 100000, 1000000, 10000000),
6                 labels = c(0, 0.001, 0.01, 0.1, 1, 10)) +
7   theme_minimal()
```

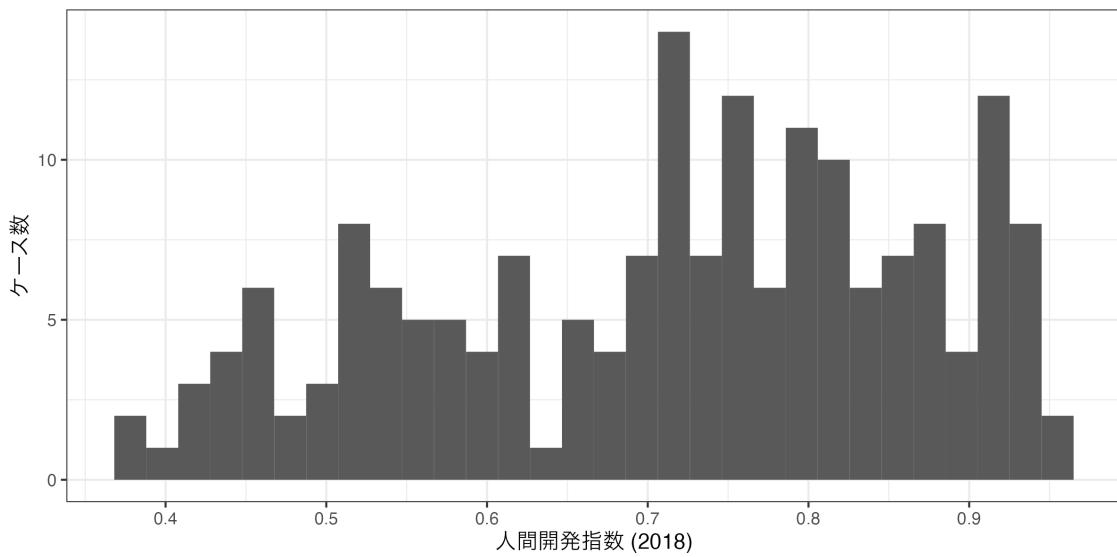


他にも `coord_*` 関数群、つまり座標系の操作を用いて軸を対数化することも可能です。他にも、データを `ggplot()` を渡す前に変数を対数化するのもあります。プログラミングにおいてある結果にたどり着く方法は複数あるので、色々試してみるのも良いでしょう。

### 18.5.1 ヒストグラムの棒の大きさを調整する

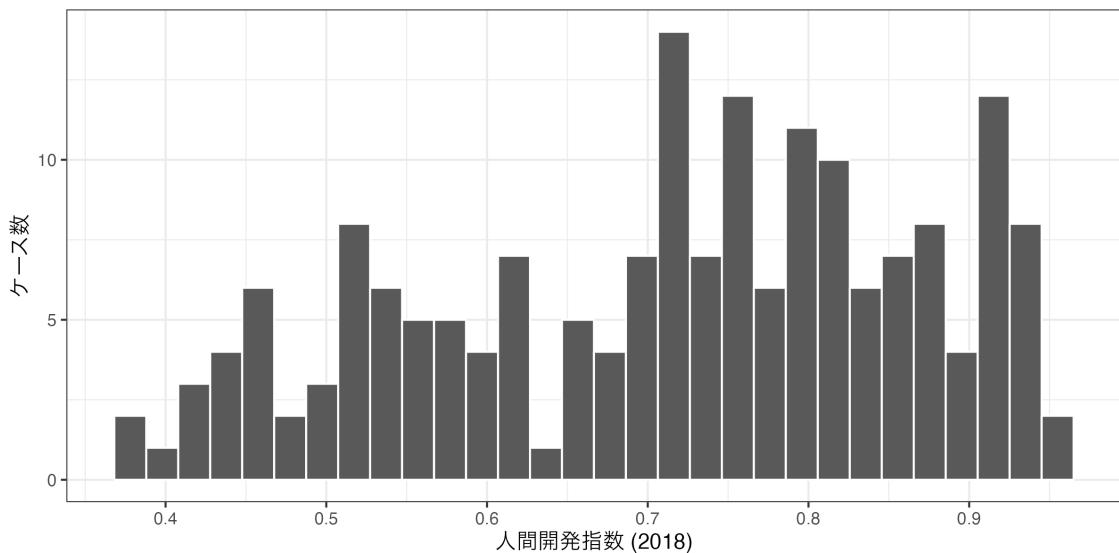
棒グラフのようにもう一つの次元を追加してみましょう。まず、人間開発指数 (HDI\_2018) の分布を示してみましょう。

```
1 Country_df %>%
2   ggplot() +
3   geom_histogram(aes(x = HDI_2018)) +
4   labs(x = "人間開発指数 (2018)", y = "ケース数") +
5   scale_x_continuous(breaks = seq(0, 1, 0.1),
6                      labels = seq(0, 1, 0.1)) +
7   theme_bw()
```



これでもヒストグラムとしては十分すぎるかも知れませんが、色々調整してみましょう。まずは、棒の枠線を白にしてみましょう。枠線はデータ内の変数に対応していないため、`aes()` の外側に入れます。枠線を指定する引数は `color` です。ちなみに棒の色を指定する引数は `fill` です。また、警告メッセージも気になるので、`HDI_2018` が欠損している行を除外します。

```
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot() +
4   geom_histogram(aes(x = HDI_2018), color = "white") +
5   labs(x = "人間開発指数 (2018)", y = "ケース数") +
6   scale_x_continuous(breaks = seq(0, 1, 0.1),
7                      labels = seq(0, 1, 0.1)) +
8   theme_bw()
```



人によってはこちらの方が見やすかも知れません。

これで一通りの作図は出来ましたが、これまで無視してきたメッセージについて考えてみましょう。

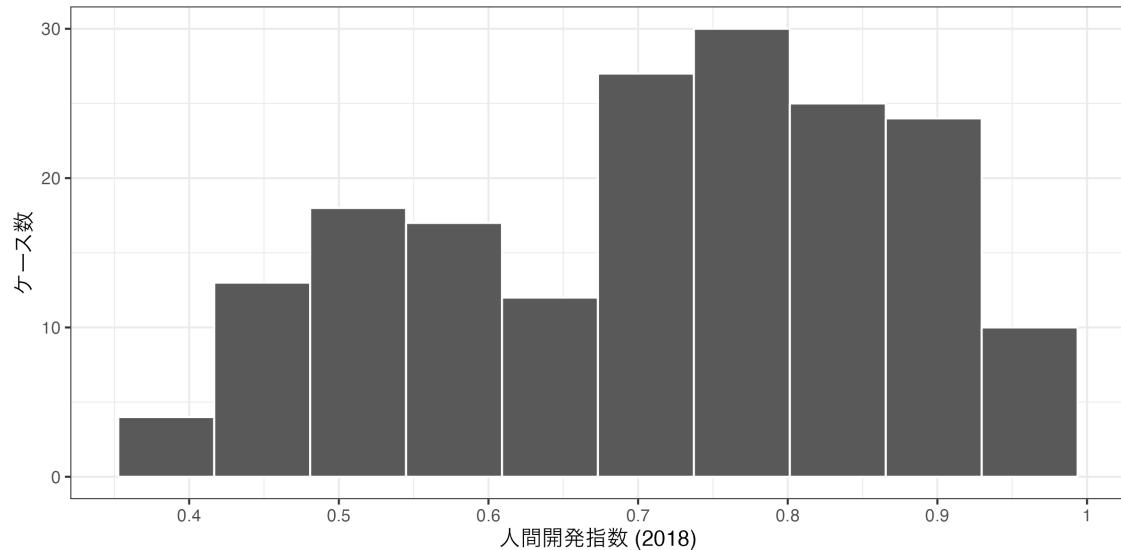
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

これは「連続変数 HDI\_2018 を 30 区間 (=階級) に分けました」という意味です。この区間数を調整する方法は 2 つあり、(1) 区間数を指定する、(2) 区間の幅を指定する方法があります。

区間数を指定することはすなわちヒストグラムの棒の数を指定することであり、`bins` 引数で調整可能です。たとえば、棒の数を 10 個にするためには `geom_histogram()` 内に `bins = 10` を指定します。むろん、棒の数もデータとは無関係であるため、`aes()` の外側に位置します。

```
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot() +
4   geom_histogram(aes(x = HDI_2018), color = "white", bins = 10) +
5   labs(x = "人間開発指数 (2018)", y = "ケース数") +
6   scale_x_continuous(breaks = seq(0, 1, 0.1),
7                      labels = seq(0, 1, 0.1)) +
```

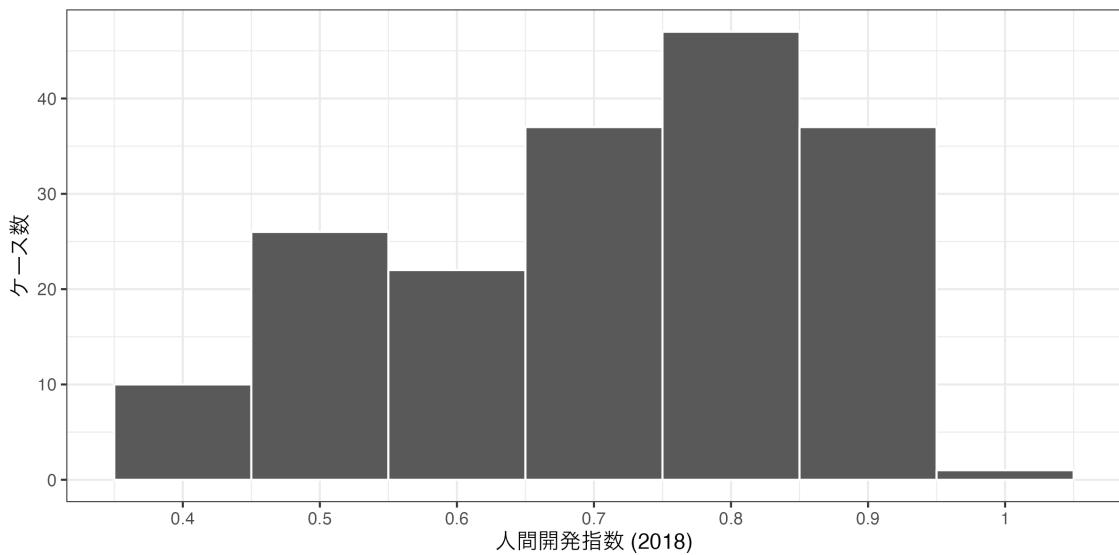
```
8     theme_bw()
```



数えてみると棒が 10 個だということが分かります。

他にも区間の幅を指定することも可能です。区間の幅は棒の幅と一致します。たとえば、棒の幅を 0.1 にしてみましょう。棒の幅は `binwidth` で調整可能です。

```
1 Country_df %>%
 2   filter(!is.na(HDI_2018)) %>%
 3   ggplot() +
 4   geom_histogram(aes(x = HDI_2018), color = "white", binwidth = 0.1) +
 5   labs(x = "人間開発指数 (2018)", y = "ケース数") +
 6   scale_x_continuous(breaks = seq(0, 1, 0.1),
 7                      labels = seq(0, 1, 0.1)) +
 8   theme_bw()
```



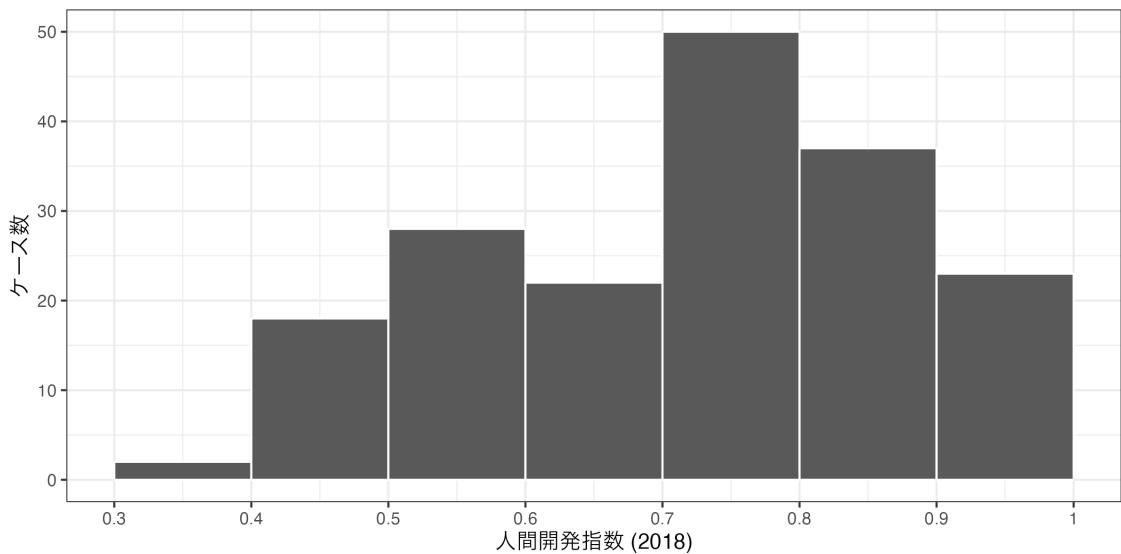
ヒストグラムが出来ましたが、棒の幅が `binwidth` で指定した 0.1 と一致することが分かります。たとえば、一番左の棒は 0.35 から 0.45 まで、つまり幅が 0.1 です。そして、一番右の棒は 0.95 から 1.05 に渡って位置します。ただし、ここで疑問を持つ読者もいるでしょう「なぜ 0.3 から 0.4、0.4 から 0.5、…ではなく、0.35 から 0.45、0.45 から 0.55 なのか」です。これは{ggplot2}の基本仕様です。ヒストグラムは度数分布表を基に作成されますが、本グラフの度数分布表は以下のようになります。

から	まで	度数
0.35	0.45	10
0.45	0.55	26
0.55	0.65	22
0.65	0.75	37
0.75	0.85	47
0.85	0.95	37
0.95	1.05	1

簡単に言うと、ヒストグラムの最初の棒は 0 を中央にした上で、棒の幅を 0.1 にしたとも言えます。これによってヒストグラムの境界線 (boundary) が、データより左右に 0.05 (`binwidth` の半分) ずつ広くなります。もし、これを調整したい場合は、`boundary` 引数を指定します。指定しない場合、`boundary` は「棒の広さ / 2」となります。棒がデータ

の範囲を超えないようにするためには、`geom_histogram()` 内に `boundary = 0` を指定します。

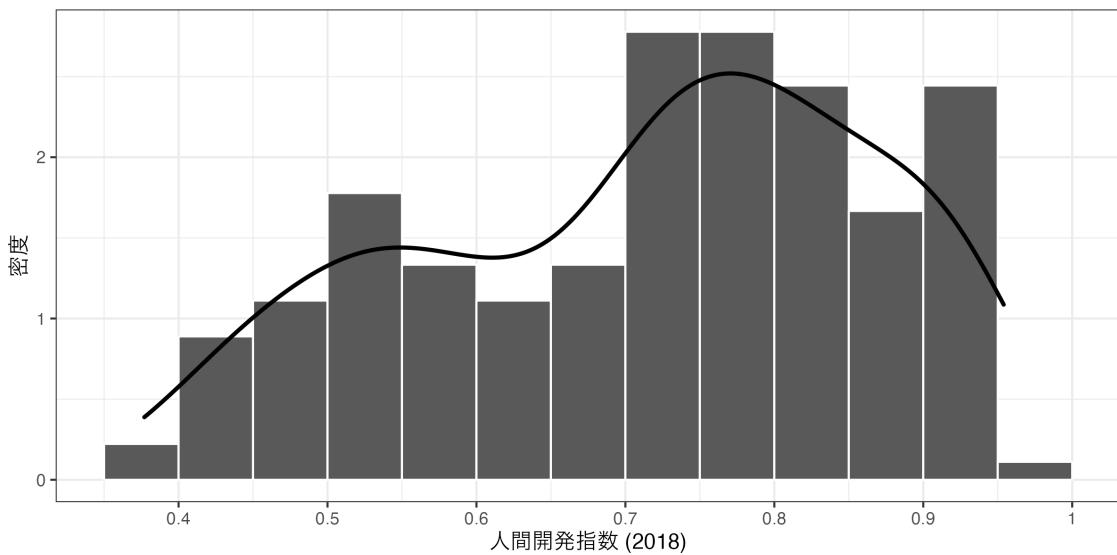
```
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot() +
4   geom_histogram(aes(x = HDI_2018), color = "white",
5                 binwidth = 0.1, boundary = 0) +
6   labs(x = "人間開発指数 (2018)", y = "ケース数") +
7   scale_x_continuous(breaks = seq(0, 1, 0.1),
8                      labels = seq(0, 1, 0.1)) +
9   theme_bw()
```



ここまで抑えとけば、普段使われるヒストグラムは問題なく作れるでしょう。

### 18.5.2 密度を追加する

ヒストグラムには以下のように密度の表す線を同時に載せるケースもあります。

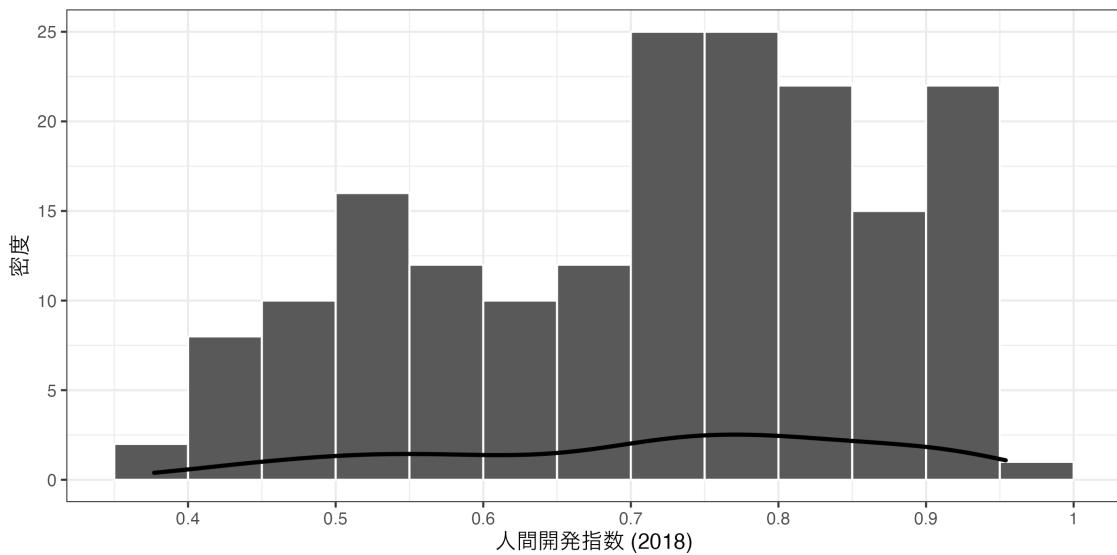


この密度の線を追加するには `geom_density()` という幾何オブジェクトを追加する必要があります。密度の線を示すには、横軸と縦軸両方の情報が必要です。横軸は `HDI_2018` で問題ないですが、縦軸はどうでしょう。縦軸には密度の情報が必要ですが、`Country_df` にそのような情報はありません。幸い、`{ggplot2}` は `y = ..density..` と指定するだけで、自動的に `HDI_2018` のある時点における密度を計算してくれます。したがって、マッピングは `aes(x = HDI_2018, y = ..density..)` のように書きます。こうなるとマッピング要素 `x` は `geom_histogram()` と `geom_density()` に共通するため、`ggplot()` に入れても問題ありません。

```

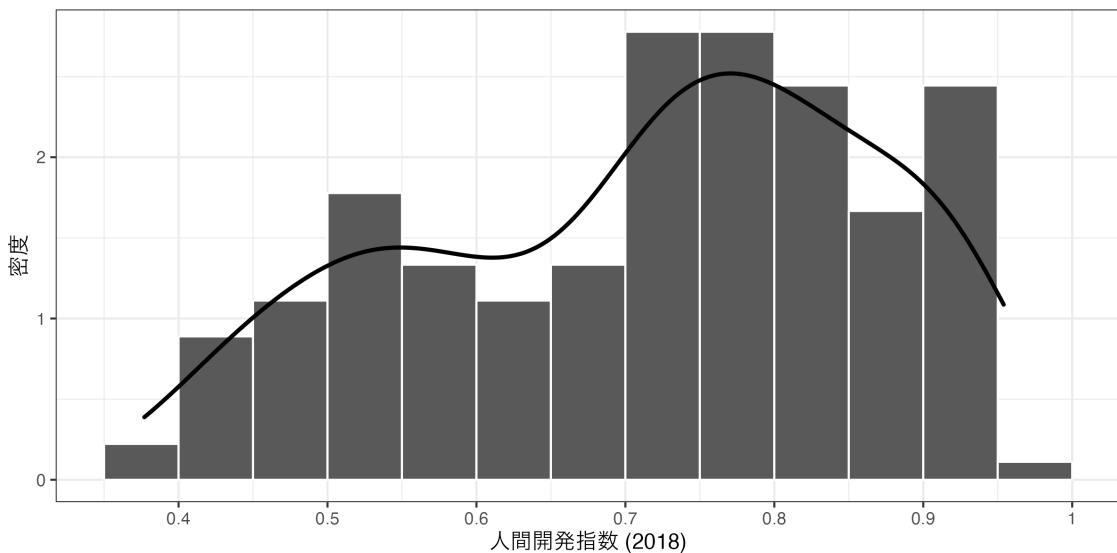
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   # 全幾何オブジェクトにおいて x を共有するため、ここでマッピング指定
4   ggplot(aes(x = HDI_2018)) +
5   geom_histogram(color = "white", binwidth = 0.05, boundary = 0) +
6   geom_density(aes(y = ..density..), size = 1) +
7   labs(x = "人間開発指數 (2018)", y = "密度") +
8   scale_x_continuous(breaks = seq(0, 1, 0.1),
9                      labels = seq(0, 1, 0.1)) +
10  theme_bw()

```



なんとも言えない微妙なグラフが出来ました。考えたものと全然違いますね。これはなぜでしょうか。それは `geom_density()` は密度を表す一方、`geom_histogram()` は度数を表すからです。2つは単位が全然違います。したがって、どちらかに単位を合わせる必要があります、この場合はヒストグラムの縦軸を度数でなく、密度に調整する必要があります。ヒストグラムの縦軸を密度にするためには、`y = ..density..` を指定するだけです。こうなると、`geom_histogram()` と `geom_density()` は `x` と `y` を共有するため、全部 `ggplot()` 内で指定しましょう。

```
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot(aes(x = HDI_2018, y = ..density..)) +
4   geom_histogram(color = "white", binwidth = 0.05, boundary = 0) +
5   geom_density(size = 1) +
6   labs(x = "人間開発指数 (2018)", y = "密度") +
7   scale_x_continuous(breaks = seq(0, 1, 0.1),
8                      labels = seq(0, 1, 0.1)) +
9   theme_bw()
```



これで密度を表す線が出来ました。

### 18.5.3 次元を追加する

次元を追加するには棒グラフと同様、`aes()` 内にマッピング要素を追加します。たとえば、OECD 加盟有無によって人間開発指数の分布がどう異なるかを確認してみましょう。OECD 変数は 0/1 であり、R では連続変数扱いになっているため、これを離散変数化するためには文字型か `factor` 型にする必要があります。`ifelse()` を使って、`OECD == 1` なら「OECD 加盟国」、`OECD == 0` なら「OECD 非加盟国」と置換したものを `OECD2` という列として追加します。そして、`OECD2` ごとに棒の色を変えるために、マッピング要素として `fill` を追加します。

```

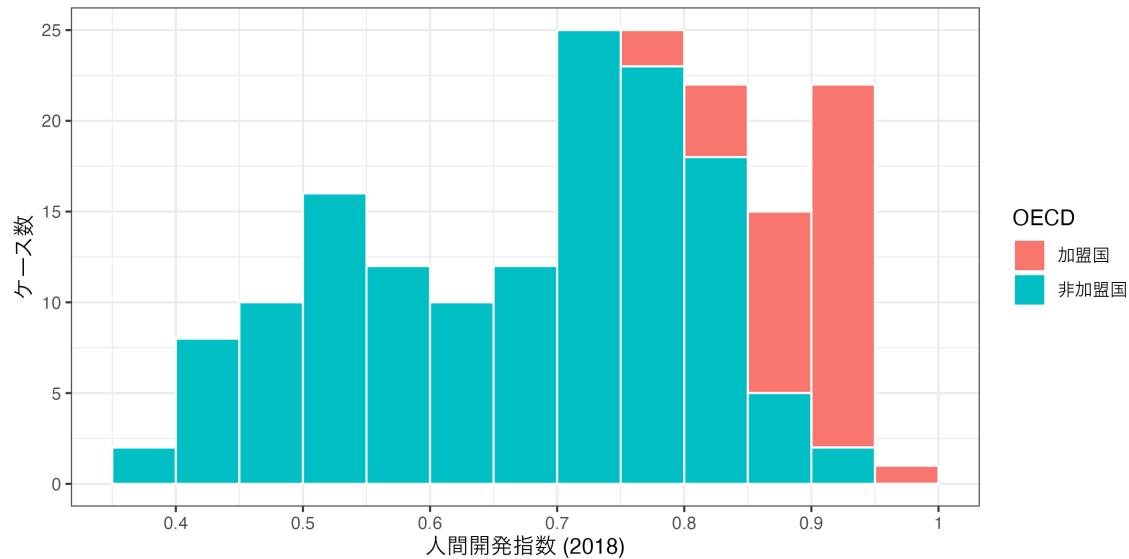
1 Country_df %>%
2   mutate(OECD2 = ifelse(OECD == 1, "加盟国", "非加盟国")) %>%
3   filter(!is.na(HDI_2018)) %>%
4   ggplot() +
5   geom_histogram(aes(x = HDI_2018, fill = OECD2),
6                 color = "white", binwidth = 0.05, boundary = 0) +
7   # fill 要素のラベルを OECD に変更
8   labs(x = "人間開発指数 (2018)", y = "ケース数", fill = "OECD") +
9   scale_x_continuous(breaks = seq(0, 1, 0.1)),

```

```

10      labels = seq(0, 1, 0.1)) +
11      theme_bw()

```



ヒストグラムが出来上がりしました。OECD 加盟国の場合、人間開発指数が相対的に高いことが分かります。しかし、積み上げヒストグラムになっています。これはある階級において OECD 加盟国と非加盟国の比率を比較する際に有効ですが、OECD 加盟国と非加盟国の分布の違いを見るにはやや物足りません。したがって、棒グラフ同様、`position` 引数で棒の位置を調整します。ただし、ヒストグラムの場合、`position = "dodge"` は向いていないので、ここでは `position = "identity"` を指定します。しかし、この場合、棒が重なってしまうと、一方の棒が見えなくなる可能性があるので、`alpha` 引数で棒の透明度を調整します。`alpha` の値は 0 から 1 までであり、0 になると、完全透明になります。ここでは 0.5 くらいにしてみましょう。

```

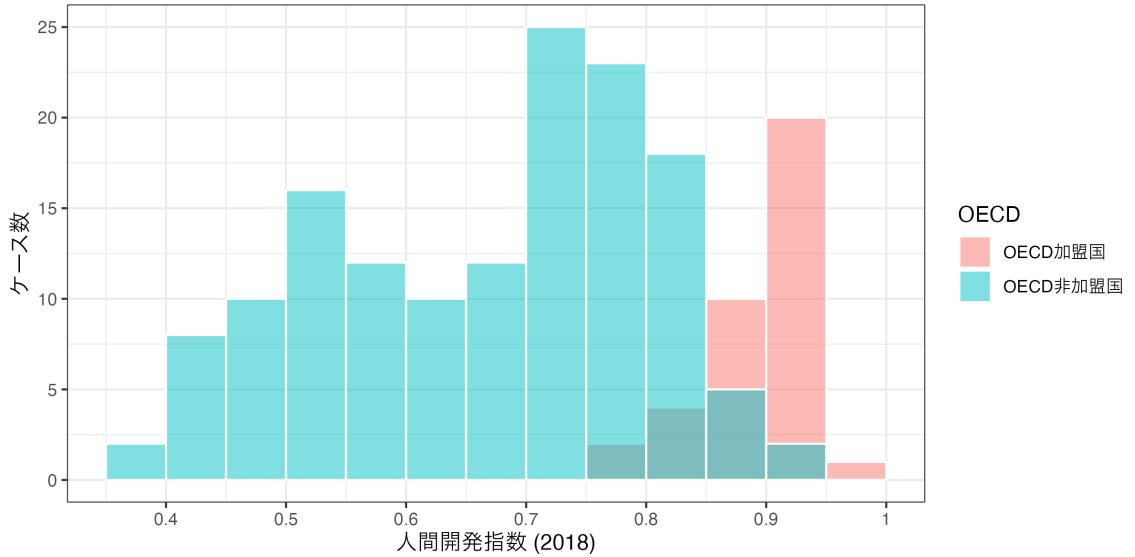
1 Country_df %>%
2   mutate(OECD2 = ifelse(OECD == 1, "OECD 加盟国", "OECD 非加盟国")) %>%
3   filter(!is.na(HDI_2018)) %>%
4   ggplot() +
5   geom_histogram(aes(x = HDI_2018, fill = OECD2),
6                 color = "white", alpha = 0.5, position = "identity",
7                 binwidth = 0.05, boundary = 0) +

```

```

8   labs(x = "人間開発指数 (2018)", y = "ケース数", fill = "OECD") +
9     scale_x_continuous(breaks = seq(0, 1, 0.1),
10                         labels = seq(0, 1, 0.1)) +
11   theme_bw()

```



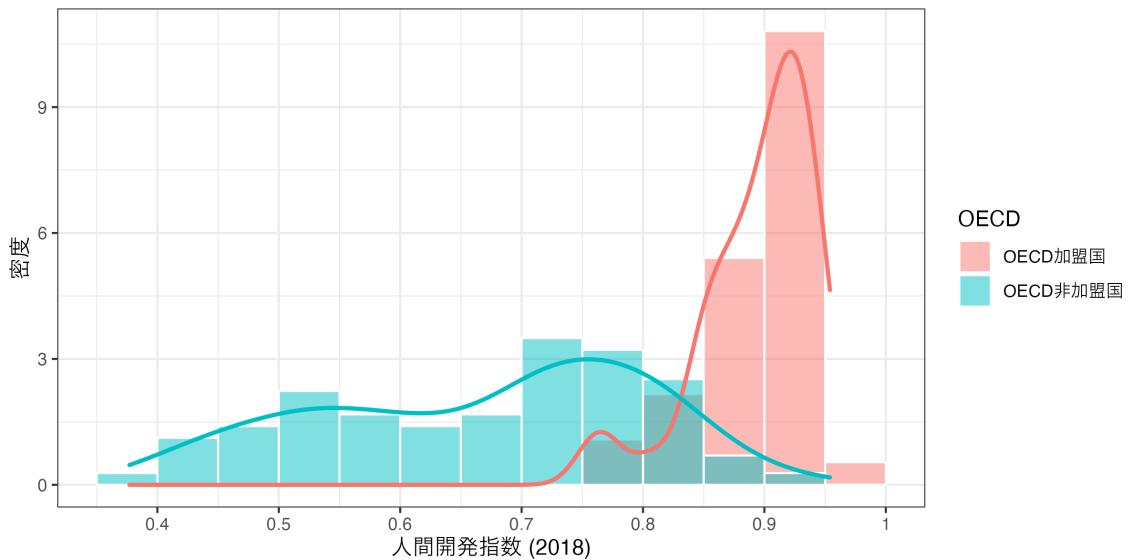
これで2つのヒストグラムを綺麗にオーバーラッピングできました。また、先ほど紹介しました`geom_density()`オブジェクトを重ねることも可能です。

```

1 Country_df %>%
2   mutate(OECD2 = ifelse(OECD == 1, "OECD 加盟国", "OECD 非加盟国")) %>%
3   filter(!is.na(HDI_2018)) %>%
4   # x は HDI_2018、y は密度(..density..) とする
5   ggplot(aes(x = HDI_2018, y = ..density..)) +
6   geom_histogram(aes(fill = OECD2),
7                 color = "white", alpha = 0.5, position = "identity",
8                 binwidth = 0.05, boundary = 0) +
9   # OECD2 ごとに異なる色を付ける。線の太さは1、凡例には表示させない
10  geom_density(aes(color = OECD2), size = 1, show.legend = FALSE) +
11  # 縦軸のラベルを「密度」に変更
12  labs(x = "人間開発指数 (2018)", y = "密度", fill = "OECD") +

```

```
13  scale_x_continuous(breaks = seq(0, 1, 0.1),  
14  labels = seq(0, 1, 0.1)) +  
15  theme_bw()
```



この場合、OECD 加盟国と非加盟国の密度をそれぞれ計算するため、ヒストグラムの見た目がこれまでのものと変わることに注意してください。

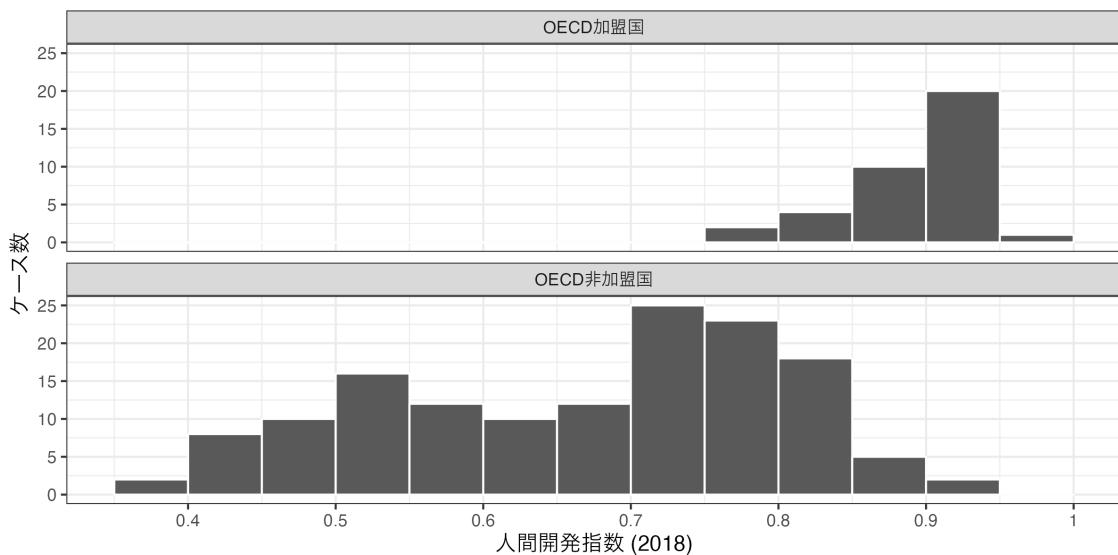
ここまで、次元拡張の方法としてヒストグラムのオーバーラッピングについて紹介しました。しかし、ヒストグラムを重なってしまうと、読みにくい人もいるかも知れません。オーバーラップ以外の方法はプロットを 2 つに分けることです。つまり、1 つのプロットに小さいプロットを複数載せることであり、この小さいプロットをファセット (facet) と呼びます。これには `facet_*`() 関数群の中の、`facet_wrap()` 関数を使います。`facet_wrap(~ 分ける変数名)` を追加すると変数ごとにプロットを分割してくれます。`ncol` や `nrow` 引数を指定すると、ファセットの列数や行数も指定可能です。

それではやってみましょう。`facet_wrap(~ OECD2)` を追加するだけです。2 つのファセットを縦に並べるために、`ncol = 1` を追加します。2 つのファセットを 1 列に並べるという意味です。

```

1 Country_df %>%
2   mutate(OECD2 = ifelse(OECD == 1, "OECD 加盟国", "OECD 非加盟国")) %>%
3   filter(!is.na(HDI_2018)) %>%
4   ggplot() +
5   geom_histogram(aes(x = HDI_2018),
6                 color = "white",
7                 binwidth = 0.05, boundary = 0) +
8   labs(x = "人間開発指数 (2018)", y = "ケース数", fill = "OECD") +
9   scale_x_continuous(breaks = seq(0, 1, 0.1),
10                      labels = seq(0, 1, 0.1)) +
11  facet_wrap(~ OECD2, ncol = 1) +
12  theme_bw()

```



OECD は加盟/非加盟だから、オーバーラッピングされたヒストグラムで十分かも知れません。しかし、大陸のように、3つ以上のグループになると、オーバーラッピングよりもファセットで分けた方が効率的です。たとえば、大陸ごとの人間開発指数のヒストグラムを作ってみましょう。

```

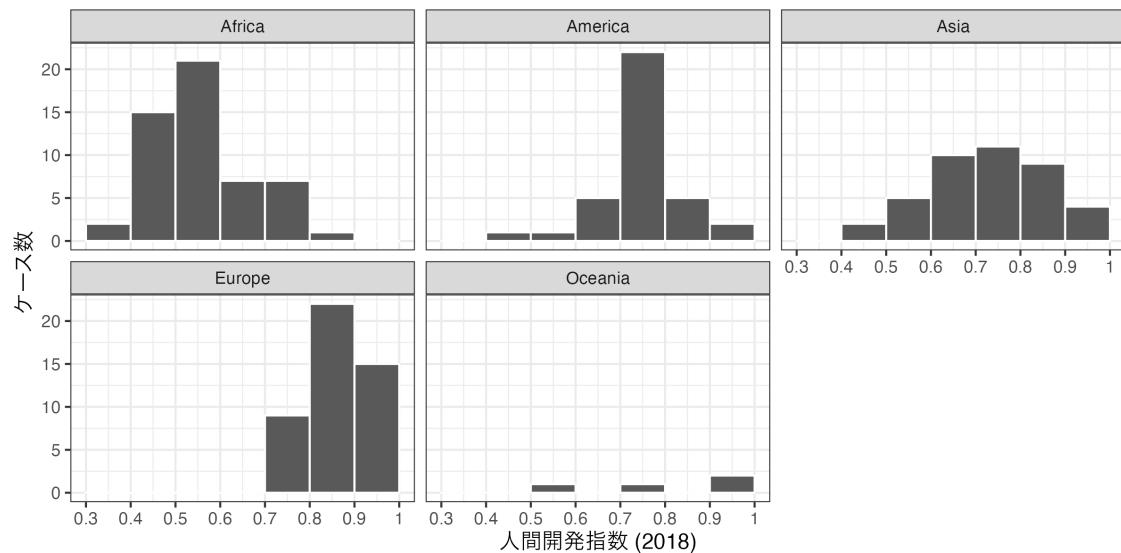
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%

```

```

3   ggplot() +
4     geom_histogram(aes(x = HDI_2018),
5                      color = "white",
6                      binwidth = 0.1, boundary = 0) +
7     labs(x = "人間開発指数 (2018)", y = "ケース数", fill = "OECD") +
8     scale_x_continuous(breaks = seq(0, 1, 0.1),
9                         labels = seq(0, 1, 0.1)) +
10    facet_wrap(~ Continent, ncol = 3) +
11    theme_bw()

```



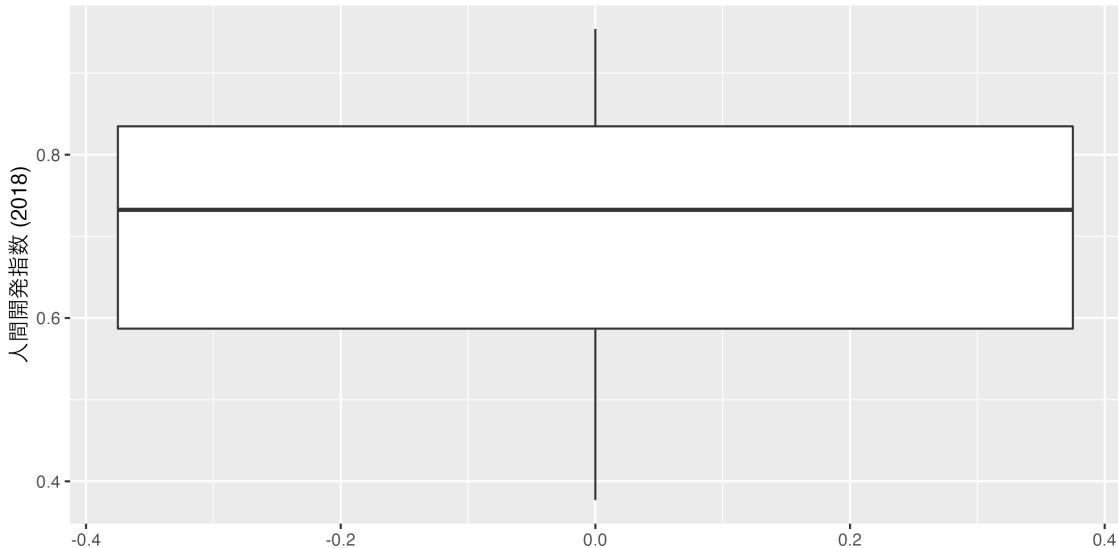
## 18.6 箱ひげ図

データの分布を示す際には、離散変数ならケース数(度数)の棒グラフ、連続変数ならヒストグラムがよく使われますが、連続変数の場合、もう一つの方法があります。それが箱ひげ図です。箱ひげ図はヒストグラムより情報量がやや損なわれますが、データの中央値、四分位点(四分位範囲)、最小値と最大値の情報を素早く読み取れます。また、複数の変数、またはグループの分布を1つのプロットに示すにはヒストグラムより優れてい

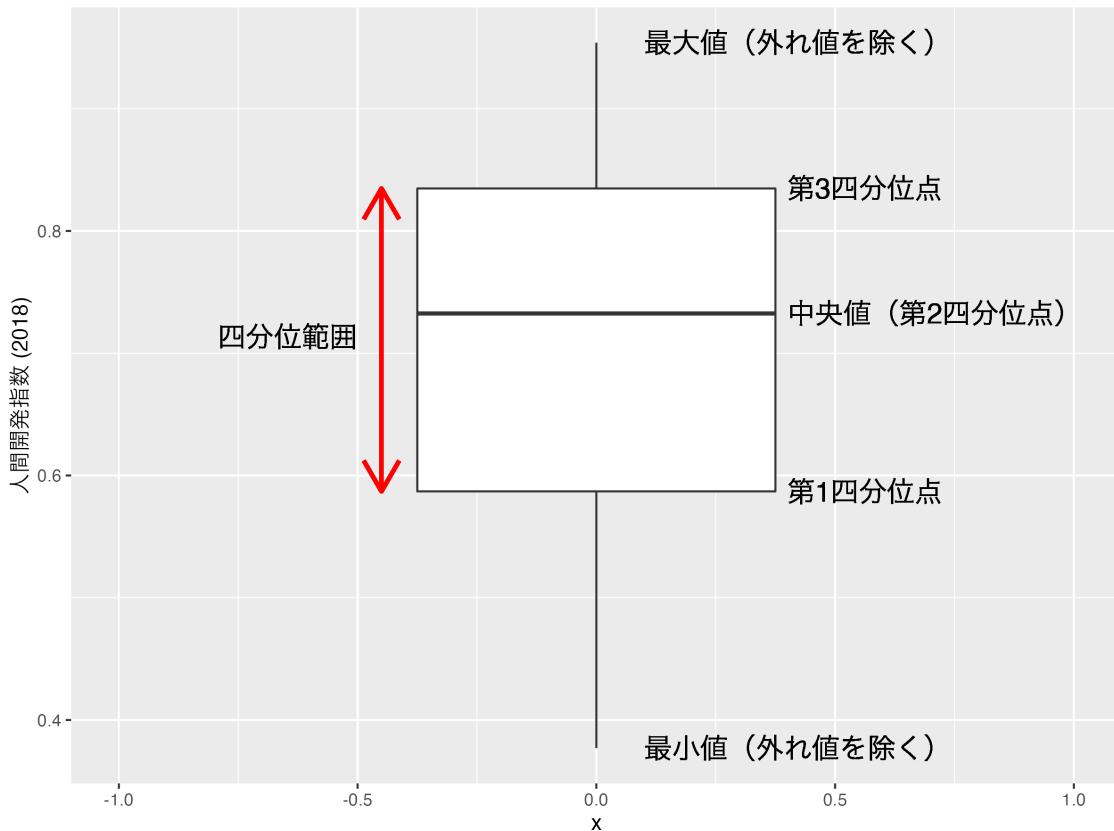
ます。

まずは人間開発指数の箱ひげ図を出してみます。使用する幾何オブジェクトは `geom_boxplot()` です。そして、指定するマッピング要素は `y` のみです。箱ひげ図から読み取れる情報は最小値・最大値、中央値、第1四分位点、第3四分位点ですが、これらの情報は縦軸として表現されます。それでは、とりあえず実際に作ってみましょう。

```
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot() +
4   geom_boxplot(aes(y = HDI_2018)) +
5   labs(y = "人間開発指数 (2018)")
```

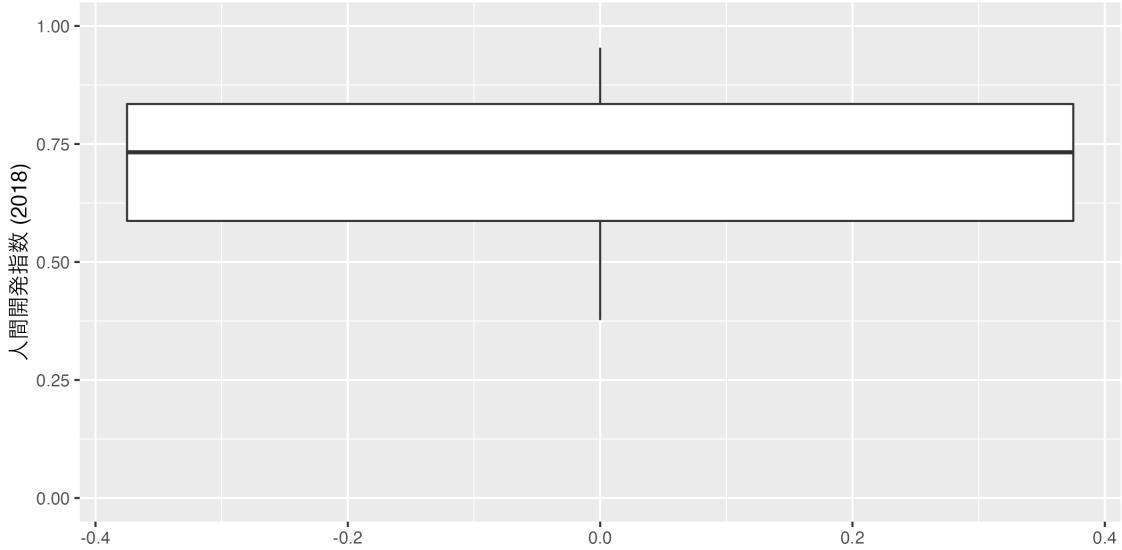


箱ひげ図の読み方は以下の通りです。



これで箱ひげ図は完成ですが、人間開発指数は 0 から 1 の相対を取るので、座標系の縦軸を調整してみましょう。座標系の操作は `coord_*`() 関数群を使いますが、現在使っているのは直交座標系（デカルト座標系）ですので `coord_cartesian()` を使います。ここで縦軸の上限と下限を指定する引数が `ylim` であり、長さ 2 の数値型ベクトルが必要です。下限 0、上限 1 ですので、`ylim = c(0, 1)` とします。

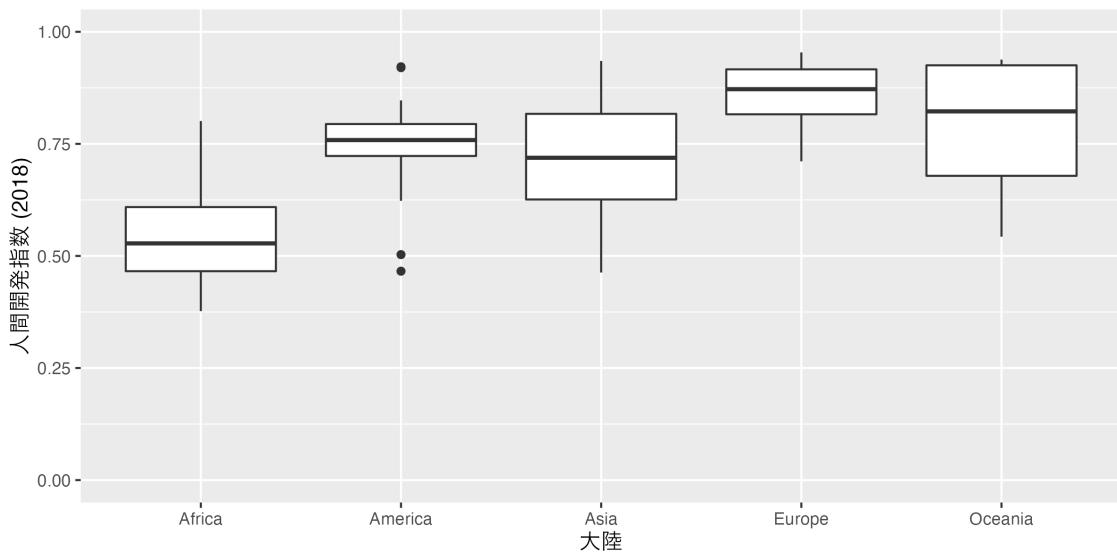
```
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot() +
4   geom_boxplot(aes(y = HDI_2018)) +
5   labs(y = "人間開発指数 (2018)") +
6   coord_cartesian(ylim = c(0, 1))
```



まだまだ改善の余地はあります、それなりの箱ひげ図の出来上がりです。しかし、一変数の箱ひげ図はあまり使われません。変数が1つだけならヒストグラムの方がより情報量は豊富でしょう。情報量が豊富ということはヒストグラムから（完璧には無理ですが）箱ひげ図を作ることは可能である一方、その逆は不可能か非常に難しいことを意味します。

ヒストグラムが力を発揮するのは複数の変数、または複数のグループごとの分布を比較する際です。先ほど、大陸ごとの人間開発指数の分布を確認するためにヒストグラムを5つのファセットで分割しました。箱ひげ図なら1つのグラフに5つの箱を並べるだけです。これは横軸を大陸(Continent)にすることを意味します。先ほどのコードのマッピングにx引数を追加してみましょう。

```
1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot() +
4   geom_boxplot(aes(x = Continent, y = HDI_2018)) +
5   labs(x = "大陸", y = "人間開発指数 (2018)") +
6   coord_cartesian(ylim = c(0, 1))
```



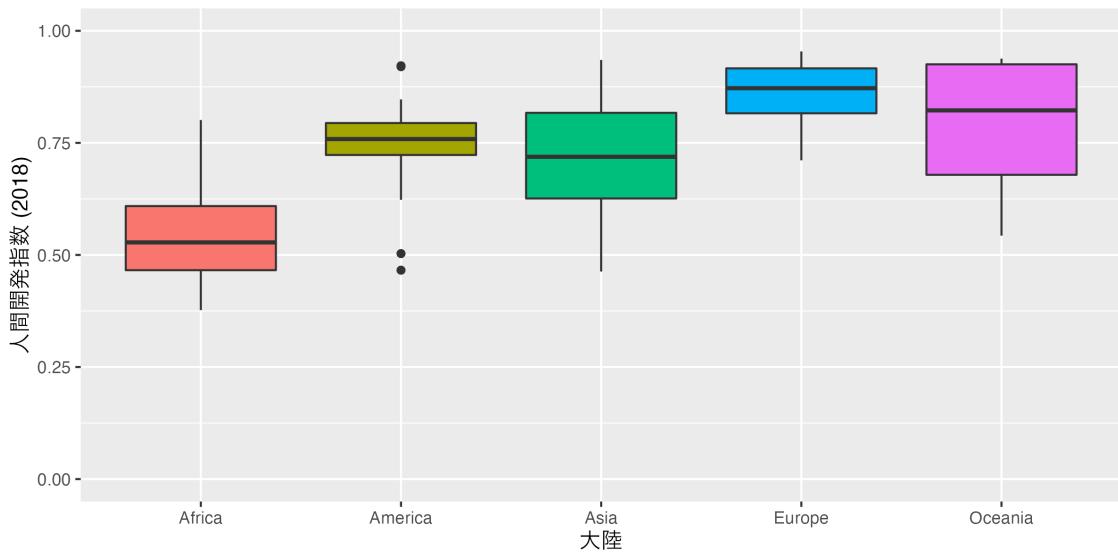
いかがでしょうか。5大陸の分布を素早く確認することができました。ヨーロッパの場合、人間開発指数が高く、バラツキも小さいことが分かります。一方、アジアとアフリカはバラツキが非常に大きいですね。アメリカ大陸はバラツキは非常に小さいですが、極端に高い国や低い国が含まれています。このアメリカ大陸に3つの点がありますが、これは外れ値です。つまり、最小値より小さい、または最大値より大きいケースを表します。最小値より小さい、または最大値より大きいという表現に違和感を感じるかも知れません。実は一般的な箱ひげ図の最小値は「第1四分位点 -  $1.5 \times \text{四分位範囲} \times 1.5$ 」より大きい値の中での最小値です。同じく最大値は「第3四分位点 +  $1.5 \times \text{四分位範囲} \times 1.5$ 」より小さい値の中での最大値です。普通に分布している場合、ほとんどのケースは箱ひげ図の最小値と最大値の範囲内に収まりますが、極端に大きい値、小さい値が含まれる場合は箱ひげ図の最小値と最大値の範囲からはみ出る場合があります。

また、複数の箱を並べる際、それぞれの箱に異なる色を付ける場合があります。これはデータ・インク比の観点から見れば非効率的ですが、可読性を落とすこともないでさほど問題はないでしょう。先ほどの箱ひげ図の場合、大陸ごとに異なる色を付けるにはマッピング要素として `fill = Continent` を追加するだけです。この場合、色に関する凡例が自動的に出力されますが、既に横軸で大陸の情報が分かるため、この判例は不要でしょう。したがって、`aes()` の外側に `show.legned = FALSE` を付けます。これは当該幾何オブジェクトに関する凡例を表示させないことを意味します。

```

1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot() +
4   geom_boxplot(aes(x = Continent, y = HDI_2018, fill = Continent),
5                 show.legend = FALSE) +
6   labs(x = "大陸", y = "人間開発指数 (2018)") +
7   coord_cartesian(ylim = c(0, 1))

```



彩りどりでちょっとテンションが上がる箱ひげ図ができました。

### 18.6.1 個別のデータを表示する

箱ひげ図は複数のグループや変数の分布を素早く比較できる長所がありますが、中央値、最小値、最大値、第1・3四分位点、外れ値の情報しか持ちません。人によってはもうちょっと情報量を増やすために個々の観測値を点として示す場合もあります。点を出力する幾何オブジェクトは次節で紹介する `geom_point()` です。以下の内容は散布図の節を一読してから読むのをおすすめします。

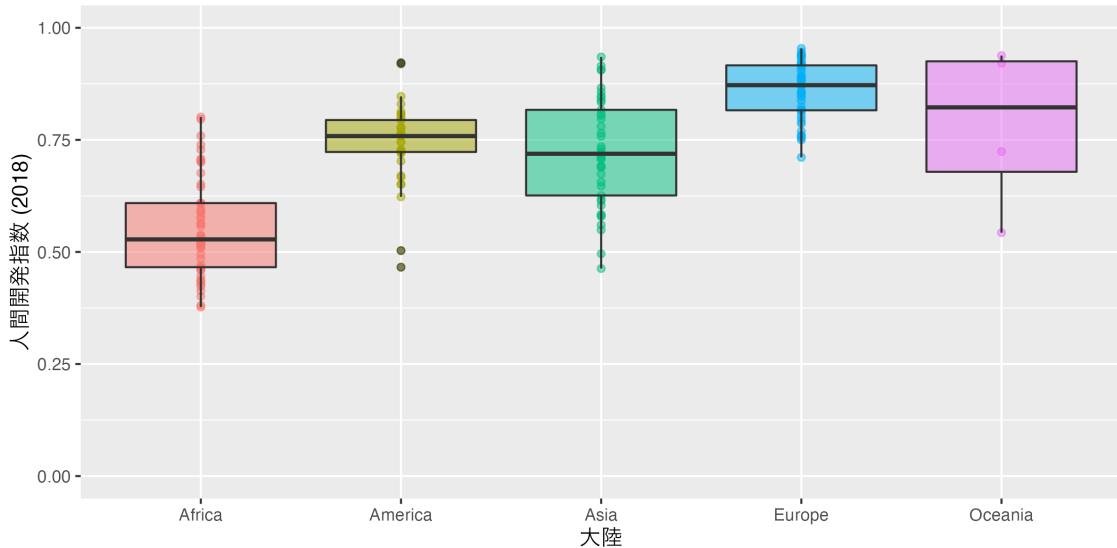
点の幾何オブジェクトにおける必須マッピング要素は点の横軸の位置と縦軸の位置、つまり `x` と `y` です。今回の場合、点の横軸が大陸 (`Continent`)、縦軸は人間開発指数

(HDI\_2018) です。つまり、`geom_boxplot()` のマッピングと同じです。したがって、`x` と `y` は `ggplot()` に入れておきます。あとは大陸ごとに店の色分けをしたいので、`color` 引数を `aes()` 内に指定します。また、点が重なる場合、読みづらくなる可能性があるため、`alpha` 引数を `aes()` 外に指定して透明度を調整します。また、箱ひげ図もある程度は透明にしないと、裏にある点が見えないため、こちらも `alpha` を調整します。

```

1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   # 全幾何オブジェクトにおいて x と y は共有されるため、ここで指定
4   ggplot(aes(x = Continent, y = HDI_2018)) +
5   geom_point(aes(color = Continent), alpha = 0.5,
6             show.legend = FALSE) +
7   geom_boxplot(aes(fill = Continent),
8                 alpha = 0.5, show.legend = FALSE) +
9   labs(x = "大陸", y = "人間開発指数 (2018)") +
10  coord_cartesian(ylim = c(0, 1))

```



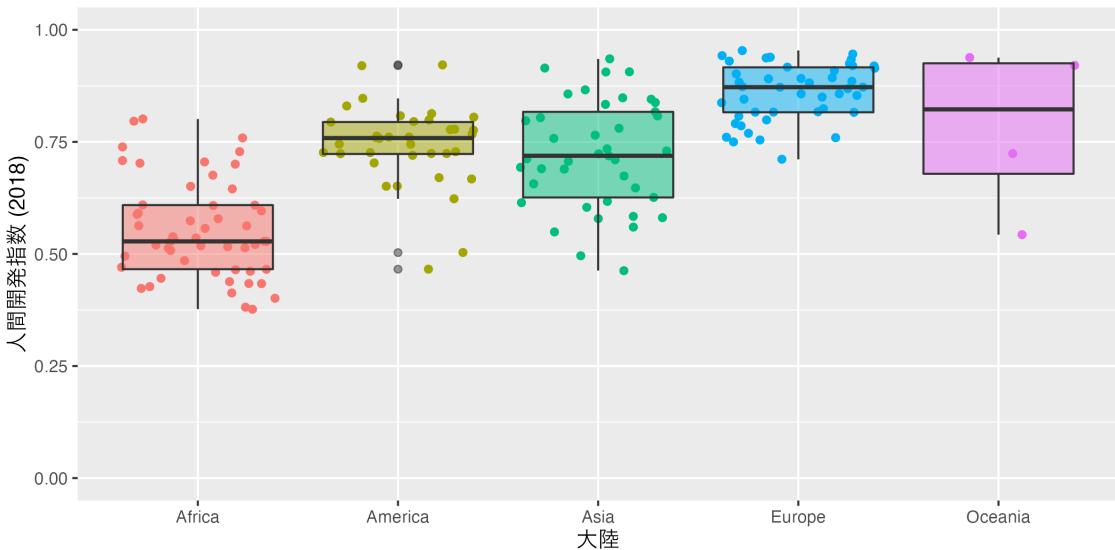
点が透明ではあるものの、それでも重なっている箇所は相変わらず読みにくいです。この場合有効な方法がジッター (jitter) です。これは点の位置に若干のノイズを付けることによって、点が重ならないようにすることです。ジッターの方法は 2 つありますが、ここ

ではジッター専用の幾何オブジェクト `geom_jitter()` を使います<sup>1)</sup>。`geom_jitter()` はノイズが追加された散布図ですので、`geom_point()` と使い方はほぼ同じです。

```

1 Country_df %>%
2   filter(!is.na(HDI_2018)) %>%
3   ggplot(aes(x = Continent, y = HDI_2018)) +
4   geom_jitter(aes(color = Continent),
5               show.legend = FALSE) +
6   geom_boxplot(aes(fill = Continent),
7                 alpha = 0.5, show.legend = FALSE) +
8   labs(x = "大陸", y = "人間開発指数 (2018)") +
9   coord_cartesian(ylim = c(0, 1))

```

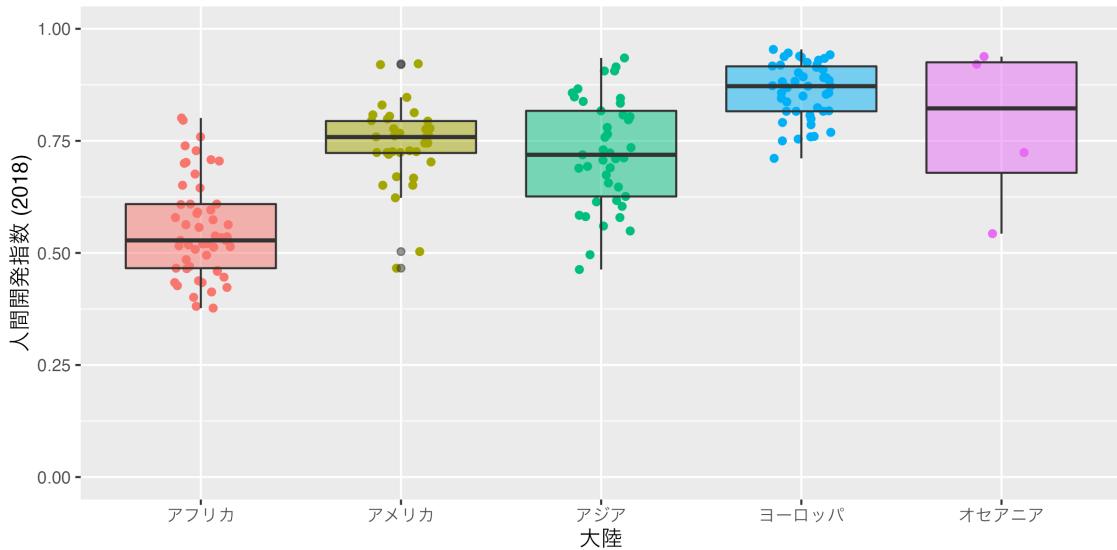


これで点が重ならなくなりましたが、ちょっと散らばりすぎるという印象もあります。この散らばり具合を調整する引数が `width` と `height` です。もちろん、これはデータの中身に対応する要素ではないため、`aes()` の外側にいれます。それぞれの実引数は 0 から 1 の間の数値になりますが、数字が大きいほど散らばり具合が大きくなり、0 になるとジッター無しの散布図と同じものになります。ここでは横の散らばり具合を 0.15 (`width = 0.15`)、縦の散らばり具合は 0 (`height = 0`) にしてみましょう。そして、横軸が英

<sup>1)</sup> もう一つの方法は `geom_point()` に `position = position_jitter()` を付けることです。

語のままなので、これも日本語に直します。

```
1 Country_df %>%
2   mutate(Continent2 = factor(Continent,
3                             levels = c("Africa", "America", "Asia",
4                                         "Europe", "Oceania"),
5                             labels = c("アフリカ", "アメリカ", "アジア",
6                                         "ヨーロッパ", "オセアニア")))) %>%
7   filter(!is.na(HDI_2018)) %>%
8   ggplot(aes(x = Continent2, y = HDI_2018)) +
9   geom_jitter(aes(color = Continent2),
10              width = 0.15, height = 0,
11              show.legend = FALSE) +
12   geom_boxplot(aes(fill = Continent2),
13                 alpha = 0.5, show.legend = FALSE) +
14   labs(x = "大陸", y = "人間開発指数 (2018)") +
15   coord_cartesian(ylim = c(0, 1))
```



### 18.6.2 次元を追加する

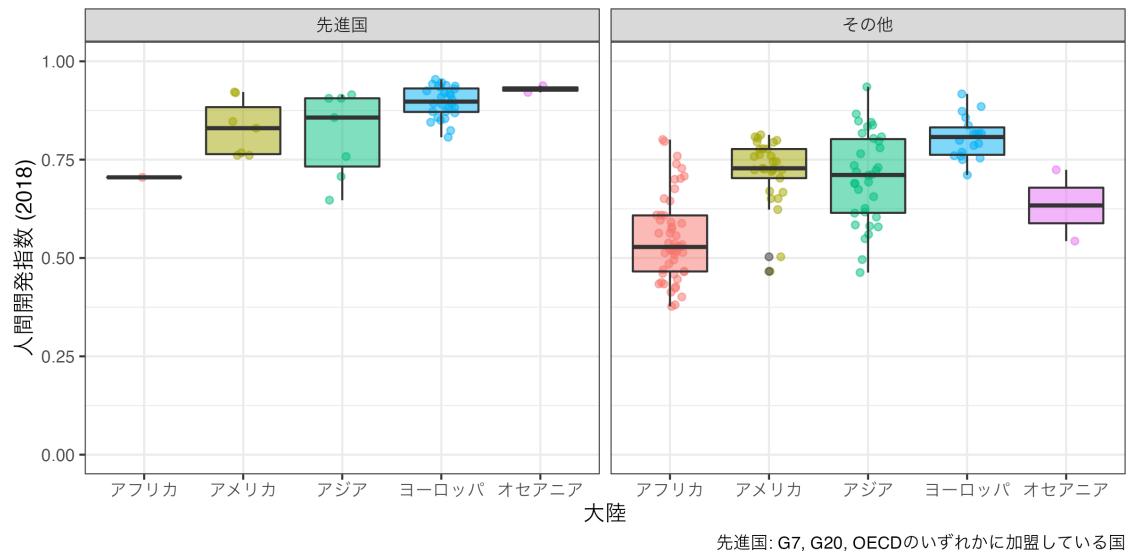
箱ひげ図に次元を追加する方法はこれまで見てきたように、1つのグラフに次元を追加するか、次元でファセットを分割するかの問題になります。まずは、簡単なファセット分割からやってみます。追加する次元は先進国か否かです。先進国の基準は不明瞭ですが、ここでは G7、G20、OECD いずれかに加盟していれば先進国と定義しましょう。そのために Country\_df に Developed 変数を追加します。この変数は G7、G20、OECD の合計が 1 以上の場合 1、0 の場合は 0 とします。そして、この Developed 変数を factor 化します。具体的には Developed が 1 だと "先進国"、0 だと "その他" にします。あとは facet\_wrap() でファセットを分割します。

```
1 Country_df %>%
2   mutate(Continent2 = factor(Continent,
3                             levels = c("Africa", "America", "Asia",
4                                       "Europe", "Oceania"),
5                             labels = c("アフリカ", "アメリカ", "アジア",
6                                       "ヨーロッパ", "オセアニア"))),
7   # G7 + G20 + OECD が 1 以上なら 1、0 なら 0 とする Developed 変数作成
8   Developed = ifelse(G7 + G20 + OECD >= 1, 1, 0),
9   # Developed 変数の factor 化
10  Developed = factor(Developed, levels = c(1, 0),
11                      labels = c("先進国", "その他"),
12                      ordered = TRUE)) %>%
13  filter(!is.na(HDI_2018)) %>%
14  ggplot(aes(x = Continent2, y = HDI_2018)) +
15  geom_jitter(aes(color = Continent2), alpha = 0.5,
16               width = 0.15, height = 0,
17               show.legend = FALSE) +
18  geom_boxplot(aes(fill = Continent2),
19               alpha = 0.5, show.legend = FALSE) +
20  # caption 引数で図の右下にテキストを入れる
21  labs(x = "大陸", y = "人間開発指数 (2018)",
```

```

22     caption = "先進国: G7, G20, OECD のいずれかに加盟している国") +
23     # ファセット分割
24     facet_wrap(~ Developed) +
25     coord_cartesian(ylim = c(0, 1)) +
26     theme_bw()

```



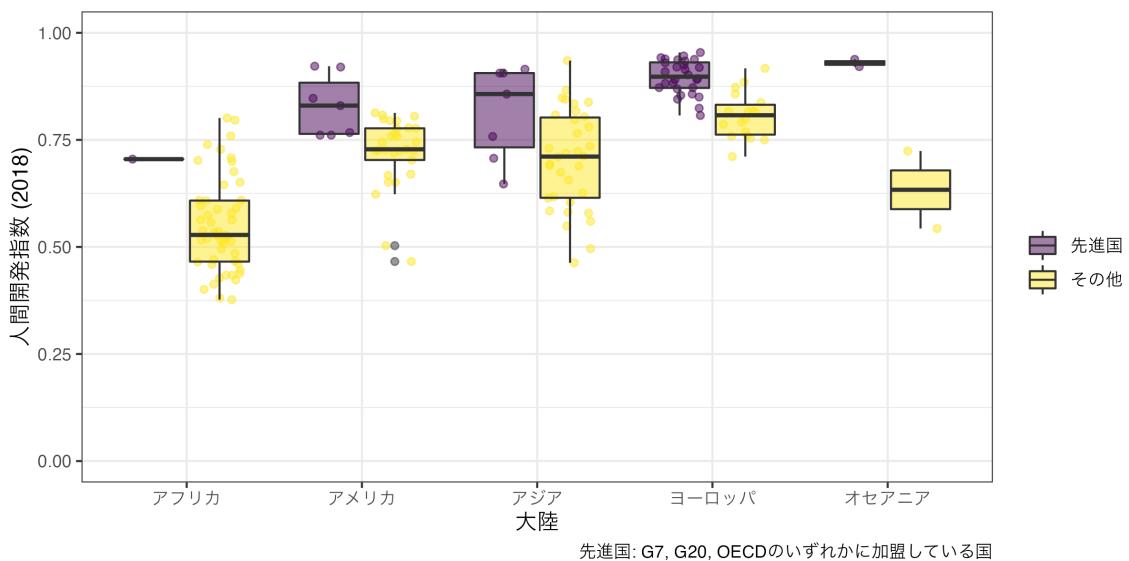
次の方法はファセットを分割せずに次元を追加する方法です。ファセットに分ける場合、「先進国における大陸别人間開発指数の分布」、「先進国外における大陸别人間開発指数の分布」は素早く読み取れます、「ある大陸における先進国/その他の国の人間開発指数の分布」を比較するにはあまり向いておりません。なぜなら目の動線が長いからです。先進国とその他の国を大陸ごとに横に並べると視線の動線が短くなり比較しやすくなります。

やり方はあまり変わりませんが、今回は色分けを `Continent` でなく `Developed` です必要があります。1つの画面に先進国とその他の国情報が同時に表示されるため、この2つを見分けるためには色分けが有効です。したがって、`geom_jitter()` と `geom_boxplot()` の `color` と `fill` を `Continent` から `Developed` へ変更します。

そしてもう一つ重要なことがあります。それは `geom_jitter()` の位置です。次元ごとに横軸の位置をずらす場合、`geom_bar()` は `position = "dodge"` を使いました。

しかし、`geom_jitter()` では "dodge" が使えません。その代わりに `position` の実引数として `position_jitterdodge()` を指定します。この中には更に `jitter.width` と `jitter.height` 引数があり、散らばりの具合をここで調整します。なぜ、`geom_jitter()` で `width` と `height` を指定するのではなく、`position_jitter()` 内で指定するかですが、これは `geom_jitter()` 関数の仕様です。`geom_jitter()` の場合、`position` と `width` または `height` を同時に指定することは出来ません。 それでは早速やってみましょう。

```
1 Country_df %>%
2   mutate(Continent2 = factor(Continent,
3                             levels = c("Africa", "America", "Asia",
4                                       "Europe", "Oceania"),
5                             labels = c("アフリカ", "アメリカ", "アジア",
6                                       "ヨーロッパ", "オセアニア"))),
7   Developed = ifelse(G7 + G20 + OECD >= 1, 1, 0),
8   Developed = factor(Developed, levels = c(1, 0),
9                       labels = c("先進国", "その他"),
10                      ordered = TRUE)) %>%
11   filter(!is.na(HDI_2018)) %>%
12   ggplot(aes(x = Continent2, y = HDI_2018)) +
13   # jitter で dodge を使うためには position_jitterdodge を使う
14   geom_jitter(aes(color = Developed), alpha = 0.5,
15               position = position_jitterdodge(jitter.width = 0.5,
16                                                jitter.height = 0),
17               show.legend = FALSE) +
18   geom_boxplot(aes(fill = Developed),
19               alpha = 0.5) +
20   labs(x = "大陸", y = "人間開発指数 (2018)", fill = "",
21         caption = "先進国: G7, G20, OECD のいずれかに加盟している国") +
22   coord_cartesian(ylim = c(0, 1)) +
23   theme_bw()
```



ファセット分割と色分け、どれが良いかという正解はありません。分析者の目的に依存するものです。もし、先進国の中での比較を強調したい場合はファセット分割が有効でしょう。しかし、同じ大陸内で先進国とその他の国との比較なら色分けの方が適切です。

### 18.6.3 複数の変数の場合

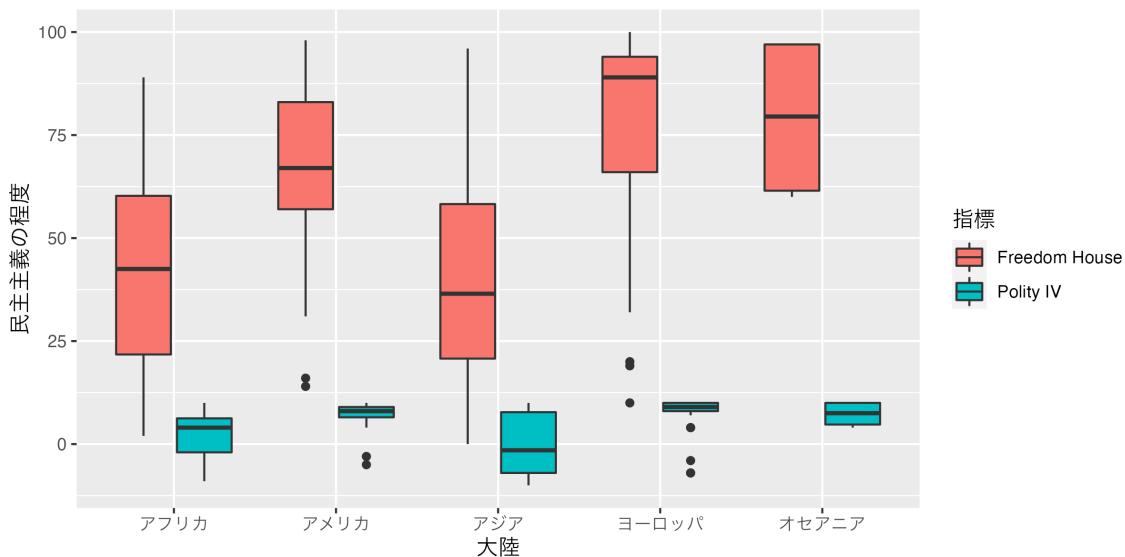
箱ひげ図はグループごとにある変数の分布を比較することができますが、複数の変数の比較も可能です。そのためにはデータの形を変形する必要があります。たとえば、Polity IV の民主主義指標 (Polity\_Score) と Freedom House の民主主義指標 (FH\_Total) の分布を、大陸ごとに示したいとします。Country\_df の場合、Polity\_Score と FH\_Total は別々の変数になっていますが、pivot\_longer() を使って、これらを 1 つの変数にまとめる必要があります。

```
9         "Polity_Score" = "Polity IV"))
10
11 Democracy_df

## # A tibble: 316 x 4
##   Country   Continent Type      Value
##   <chr>     <chr>     <chr>     <dbl>
## 1 Afghanistan Asia     Polity IV    -1
## 2 Afghanistan Asia     Freedom House 27
## 3 Albania     Europe   Polity IV     9
## 4 Albania     Europe   Freedom House 67
## 5 Algeria     Africa   Polity IV     2
## 6 Algeria     Africa   Freedom House 34
## 7 Angola      Africa   Polity IV    -2
## 8 Angola      Africa   Freedom House 32
## 9 Argentina   America  Polity IV     9
## 10 Argentina  America  Freedom House 85
## # ... with 306 more rows
```

続いて、箱ひげ図の作成ですが、これは次元の追加と全く同じやり方になります。fillで色分けをするか、ファセット分割をするかですね。ここでは箱の色分けをします。

```
1 Democracy_df %>%
2   ggplot() +
3   geom_boxplot(aes(x = Continent, y = Value, fill = Type)) +
4   labs(x = "大陸", y = "民主主義の程度", fill = "指標") +
5   scale_x_discrete(breaks = c("Africa", "America", "Asia", "Europe", "Oceania"),
6                     labels = c("アフリカ", "アメリカ", "アジア", "ヨーロッパ", "オセアニア"))
```



しかし、1つ問題があります。それは Polity IV は-10 から 10 までの指標なのに対しで、Freedom House は 0 から 100 までの指標になっている点です。この場合、正確な比較が出来ません。複数の変数を1つの箱ひげ図に出す際は、変数のスケールが一致させた方が良いでしょう。たとえば、複数の人、または団体に対する感情温度はスケールが 0 から 100 であるため、使えます。しかし、今回の場合はあまり良いケースではありません。

もし、スケールが異なるものを示すためにはスケールを調整する必要があります。よく使われるのはスケールを平均 0、標準偏差 1 にする「標準化」ですが、変数の分布を似通ったものにするため、あまり良くないかも知れません。ここでは Freedom House 指標から 50 を引き、そこから 5 で割った値を使います。こうすることで、Freedom House 指標が 100 なら 10、0 なら-10 になり、最小値と最大値のスケールを Polity IV に合わせることが可能です。

```

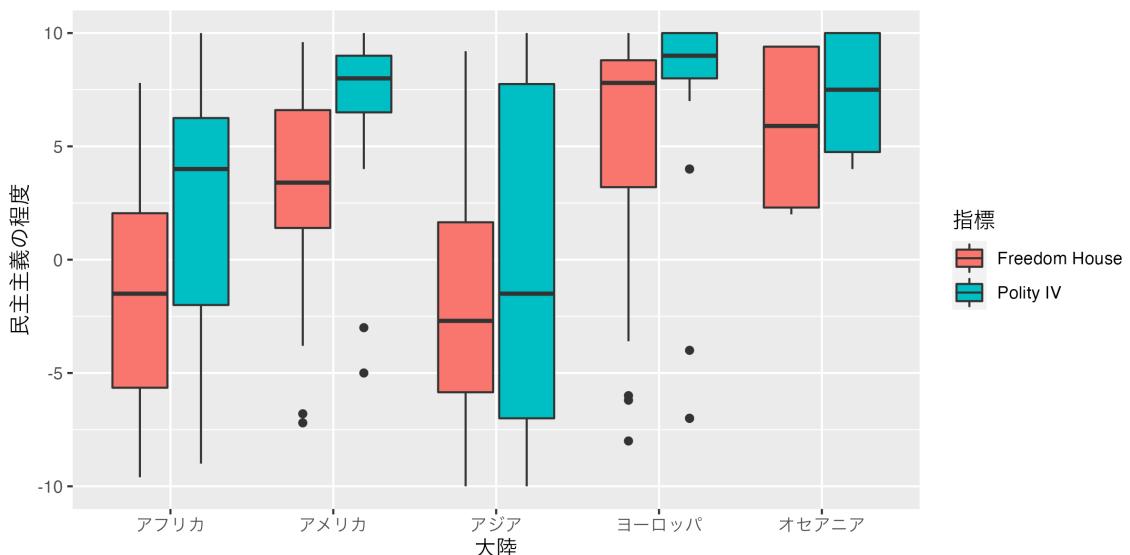
1 Democracy_df <- Country_df %>%
2   select(Country, Continent, Polity_Score, FH_Total) %>%
3   # FH_Total のすケース調整
4   mutate(FH_Total = (FH_Total - 50) / 5) %>%
5   filter(!is.na(Polity_Score), !is.na(FH_Total)) %>%
6   pivot_longer(cols      = contains("_"),
7                 names_to  = "Type",
8                 values_to = "Value") %>%

```

```

9  mutate(Type2 = recode(Type,
10    "FH_Total"      = "Freedom House",
11    "Polity_Score" = "Polity IV"))
12
13 Democracy_df %>%
14   ggplot() +
15   geom_boxplot(aes(x = Continent, y = Value, fill = Type2)) +
16   labs(x = "大陸", y = "民主主義の程度", fill = "指標") +
17   scale_x_discrete(breaks = c("Africa", "America", "Asia", "Europe", "Oceania"),
18                     labels = c("アフリカ", "アメリカ", "アジア", "ヨーロッパ", "オセアニア"))

```



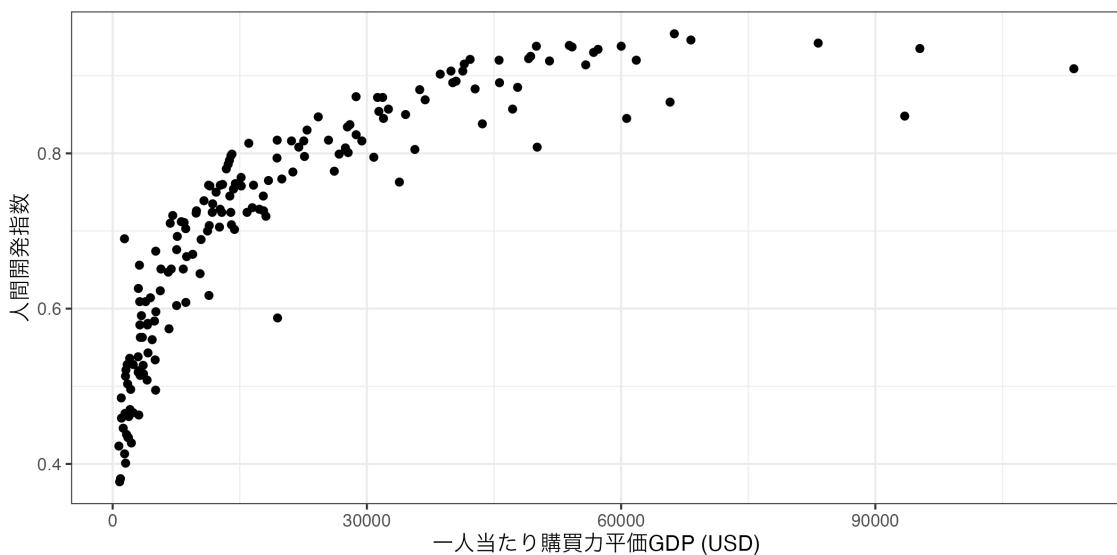
先よりは比較可能な図のように見えますが、あくまでも最小値と最大値を一致させたものであるため、厳密な意味ではこれもよくありません。複数の変数を1つの箱ひげ図としてまとめる場合は、スケールが一致するもののみを使うことを推奨します。

## 18.7 散布図

続いて、散布図の作成について解説します。散布図においてデータは点で表現され、点を表示するためには、少なくとも横軸と縦軸といった2つの情報が必要です。したがって、マッピングに使う変数は最低2つであり、横軸はx、縦軸はyで指定します。今回はCountry\_dfを使って、一人当たりGDP(購買力平価基準)と人間開発指数の関係を調べてみましょう。散布図の幾何オブジェクト関数はgeom\_point()です。そして、それぞれの変数はPPP\_per\_capita、HDI\_2018であるから、マッピングはaes(x = PPP\_per\_capita, y = HDI\_2018)になります。

```
1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = PPP_per_capita, y = HDI_2018)) +
4   labs(x = "一人当たり購買力平価 GDP (USD)", y = "人間開発指数") +
5   theme_bw()
```

```
## Warning: Removed 11 rows containing missing values (geom_point).
```



以下のメッセージが表示されますが、これは一人当たりGDP(購買力平価基準)また

は人間開発指数が欠損しているケースが11カ国あることを意味します。たとえば、教皇聖座 (Holy See; いわゆるバチカン) や西サハラ、ソマリアなどの国があります。

```
## Warning: Removed 11 rows containing missing values (geom_point).
```

どのようなケースが除外されたかは `dplyr::filter()` 関数を使えば簡単に調べられます。

```
1 Country_df %>%
2   filter(is.na(PPP_per_capita) | is.na(HDI_2018)) %>%
3   select(Country, PPP_per_capita, HDI_2018)

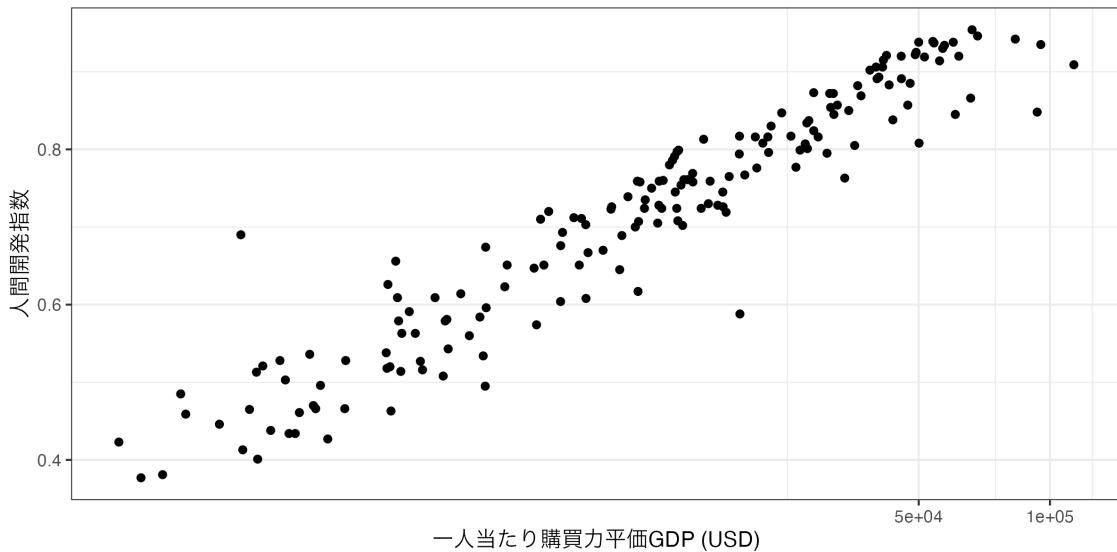
## # A tibble: 11 x 3
##   Country      PPP_per_capita HDI_2018
##   <chr>          <dbl>        <dbl>
## 1 Andorra        NA          0.857
## 2 Cuba           NA          0.778
## 3 Holy See        NA          NA
## 4 Kosovo          11078.       NA
## 5 Liechtenstein   NA          0.917
## 6 Monaco          NA          NA
## 7 San Marino     62554.       NA
## 8 Somalia          NA          0.557
## 9 Syria            NA          0.549
## 10 Taiwan          46145.       NA
## 11 Western Sahara NA          NA
```

このように何らかのケースが除外されたとメッセージが出力された場合、ちゃんとどのケースが除外されたかを確認することは重要です。この11カ国は未承認国や国内政治の不安定によりデータが正確に把握できないところがほとんどですね。

散布図を見ると経済力と人間開発指数の間には正の関係が確認できます。ただし、経済力が高くなるにつれ、人間開発指数の増加幅は減少していきます。どちらかと言えば対数関数のような関係でしょう。実際、`scale_x_log10()` や、`scale_x_continuous(trans = "log10")` などで横軸を対数化するとほぼ線形の関係が観察できます。今回は座標系を

変換して横軸を対数化してみます。座標系の調整は `coord_*` 関数群を使いますが、対数化には `coord_trans()` を使います。ここで変換する軸と変換方法を指定します。今回は横軸を底 10 の対数化を行うから `x = "log10"` と指定します。これは `scale_x_log10()` と全く同じですが、縦軸も対数化するなどの場面においては `coord_trans()` の方が便利でしょう。

```
1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = PPP_per_capita, y = HDI_2018)) +
4   labs(x = "一人当たり購買力平価 GDP (USD)", y = "人間開発指数") +
5   coord_trans(x = "log10") +
6   theme_bw()
```



対数化してみたら、かなり綺麗な線形関係が確認できます。事実を言うと、そもそも人間開発指数は所得も評価項目であるため、線形関係があるのは当たり前です。

### 18.7.1 次元を追加する

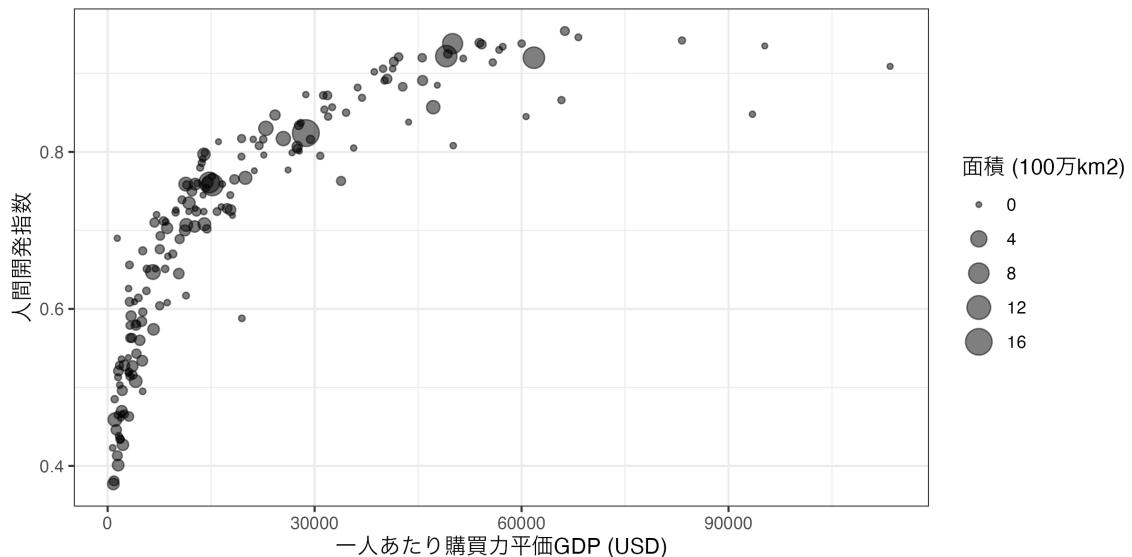
散布図の点は横軸の数値と縦軸の数値を持っています。2 次元平面で表現できる散布図は少なくとも 2 つの情報を持つことになります。しかし、2 次元平面であっても、3

つ以上の情報、つまり3次元以上の情報を示すことが可能です。たとえば、点ごとに色を変更することもできます。OECD加盟国と非加盟国に異なる色を与えると、2次元平面上であっても、3つの情報を含む散布図が作れます。他にも透明度 (alpha)、点の形 (shape)、大きさ (size) などで点に情報を持たせることができます。たとえば、国の面積に応じて点の大きさが変わる散布図を作成するなら aes() 内に size = Area を追加するだけです。面積の単位は非常に大きいので、Area を100万で割った値を使います。また、点が大きくなると重なりやすくなるため、半透明 (alpha = 0.5) にします。

```

1 Country_df %>%
2   mutate(Area2 = Area / 1000000) %>%
3   ggplot() +
4   geom_point(aes(x = PPP_per_capita, y = HDI_2018, size = Area2),
5             alpha = 0.5) +
6   labs(x = "一人あたり購買力平価 GDP (USD)", y = "人間開発指数",
7         size = "面積 (100万 km2)") +
8   theme_bw()

```



一人当たり GDP が非常に高い国の多くは面積が小さい国が多いですね。

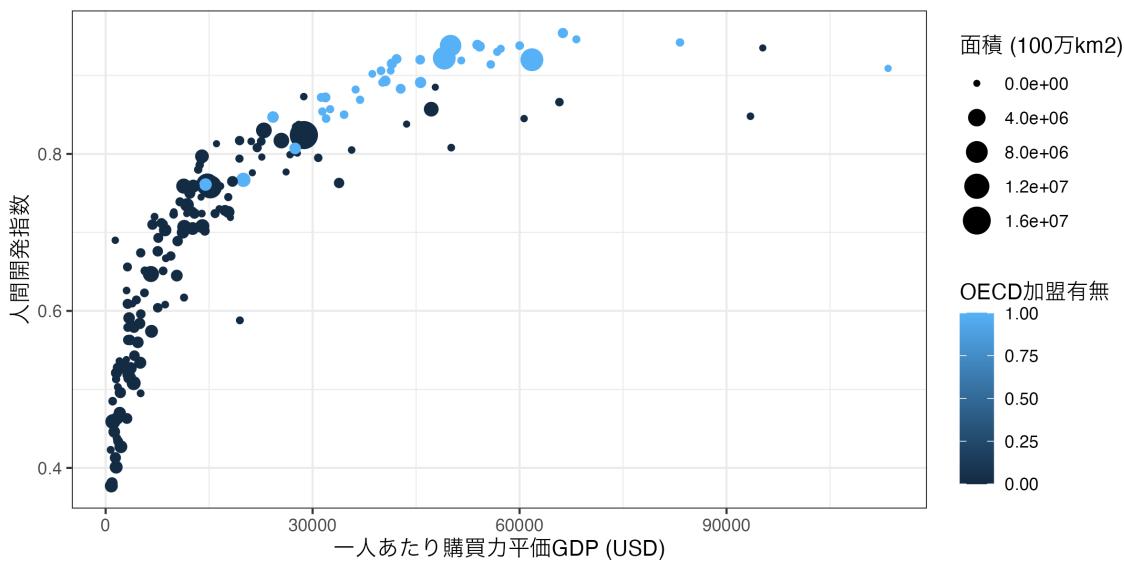
以上のコードでは、aes() だけでなく、labs() 内にも size 引数を指定しました。labs() 内に x と y 以外の引数を指定することはヒストグラムのところでもしましたが、

説明はしませんでした。ここで詳しく説明します。`labs()` 内の引数はマッピング要素と対応します。今回は `geom_point()` 異象オブジェクト内に `x`、`y`、`size` があり、それぞれにラベルを付けることになります。これを指定しない場合、変数名がデフォルト値として出力されます。`x` と `y` は縦軸と横軸のラベルですが、その他のマッピング要素は凡例として出力される場合が多いです。凡例のラベルを修正したい時にはとりあえず `labs()` から覗いてみましょう。

続きまして、もう一つ次元を追加してみましょう。散布図は他のグラフに比べ次元拡張が容易なグラフです。ここでは色分けをしてみましょう。たとえば、OECD 加盟有無 (OECD) で色分けする場合、これまでと同様、`color` を `aes()` 内に加えるだけです。

```

1 Country_df %>%
2   mutate(Area2 = Area / 1000000) %>%
3   ggplot() +
4   geom_point(aes(x = PPP_per_capita, y = HDI_2018,
5                 size = Area, color = OECD)) +
6   labs(x = "一人あたり購買力平価 GDP (USD)", y = "人間開発指数",
7         size = "面積 (100 万 km2)", color = "OECD 加盟有無") +
8   theme_bw()
```

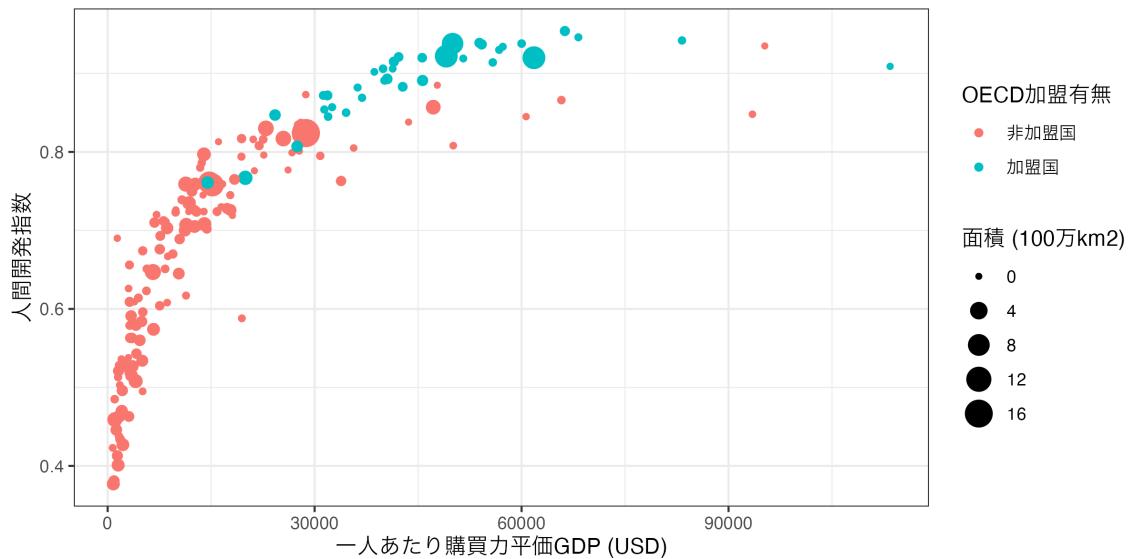


色分けはされていますが、凡例を見ると想像したものとやや違いますね。なぜなら

OECD変数が数値型になっているからです。実際のデータには `OECD = 0.5` や `OECD = 0.71` のような値は存在しませんが、数値型である以上、その値をとること自体は出来ます。したがって、0から1までの数値に対応できるように、色分けもグラデーションになっています。これを見やすく二分するためには、OECD変数を `factor`型か文字型に変換する必要があります。

```

1 Country_df %>%
2   mutate(Area2 = Area / 1000000,
3         OECD2 = factor(OECD, levels = 0:1,
4                           labels = c("非加盟国", "加盟国"))) %>%
5   ggplot() +
6   geom_point(aes(x = PPP_per_capita, y = HDI_2018,
7                   size = Area2, color = OECD2)) +
8   labs(x = "一人あたり購買力平価 GDP (USD)", y = "人間開発指数",
9         size = "面積 (100万 km2)", color = "OECD 加盟有無") +
10  theme_bw()
```



これで散布図の出来上がりです。2次元座標系で表現された散布図ですが、この図から分かる情報は何があるでしょうか。

### 1. 一人当たり購買力平価 GDP

- 2. 人間開発指數
  - 3. 面積
  - 4. OECD 加盟有無

1つの図から4つの情報が読み取れます。つまり、4次元グラフと言えます。ここにファセット分割、透明度の調整、点の形の変更まですると7次元のグラフもできます。ただし、あまりにも次元数の多いグラフは読みにくくなるため、必要だと思う変数のみマッピングしましょう。

### 18.7.2 一部の点をハイライトする

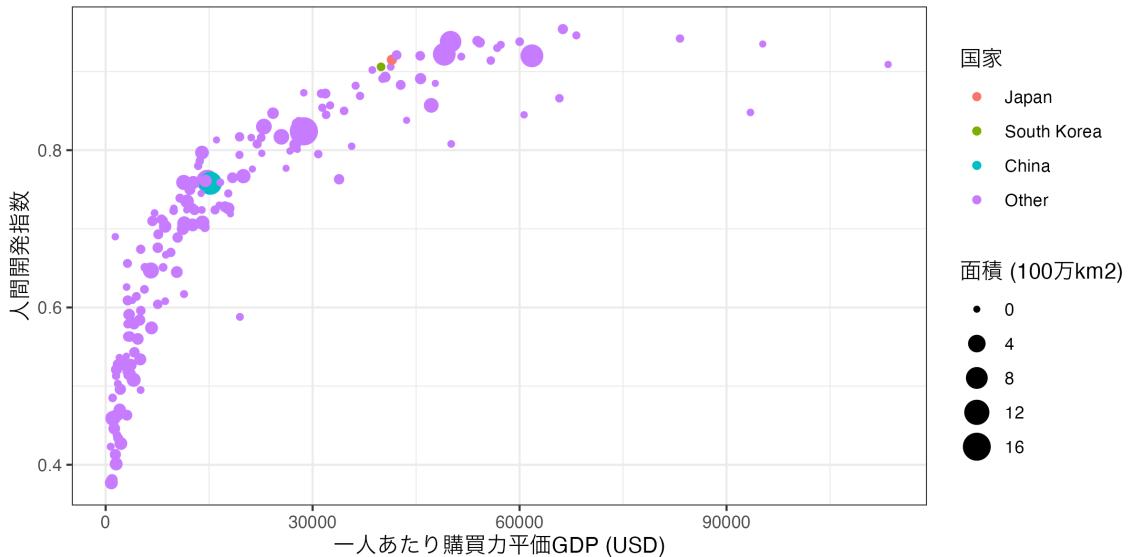
これまでの図と比べ、散布図は一般的に表示される情報が多いです。棒グラフ、箱ひげ図の場合、数個、あるいは十数個程度の項目ごとに棒や箱が出力されますが、散布図の場合、データのサイズだけ点が出力されます。むろん、散布図というのは個々の点の情報よりも二変数間の関係を確認するために使われるため、これは短所ではありません。

しかし、散布図の中で一部の点をハイライトしたい場合もあるでしょう。たとえば、先ほどの図において日中韓だけハイライトするためにはどうすれば良いでしょうか。まず、考えられるのは `color` によるマッピングでしょう。`Country` 変数を日本、韓国、中国、その他の 4 つの水準にまとめ、この値によって色分けをすればいいでしょう。実際にやってみましょう。

```

13  geom_point(aes(x = PPP_per_capita, y = HDI_2018,
14                  size = Area2, color = Country2)) +
15  labs(x = "一人あたり購買力平価 GDP (USD)", y = "人間開発指数",
16        size = "面積 (100 万 km2)", color = "国家") +
17  theme_bw()

```



日本と韓国を見つけるのは大変ですが、目的達成と言えるでしょう。ただ、もっと楽な方法があります。それが湯谷啓明さんが開発した{gghighlight}パッケージです。詳しい使い方は湯谷さんによる解説ページを参照して頂きますが、ここでは簡単な使い方のみ紹介します。まずは、`install.packages("gghighlight")`でパッケージをインストールし、読み込みます。

```
1 pacman::p_load(gghighlight)
```

続いてですが、国ごとに色分けする予定ですので、散布図の `color` は `Country` 変数でマッピングします。そして、{gghighlight}幾何オブジェクトを追加します。まずは使い方からです。

```

1 # gghighlight() の使い方
2 gghighlight(条件式, label_params = list(引数 1, 引数 2, ...))

```

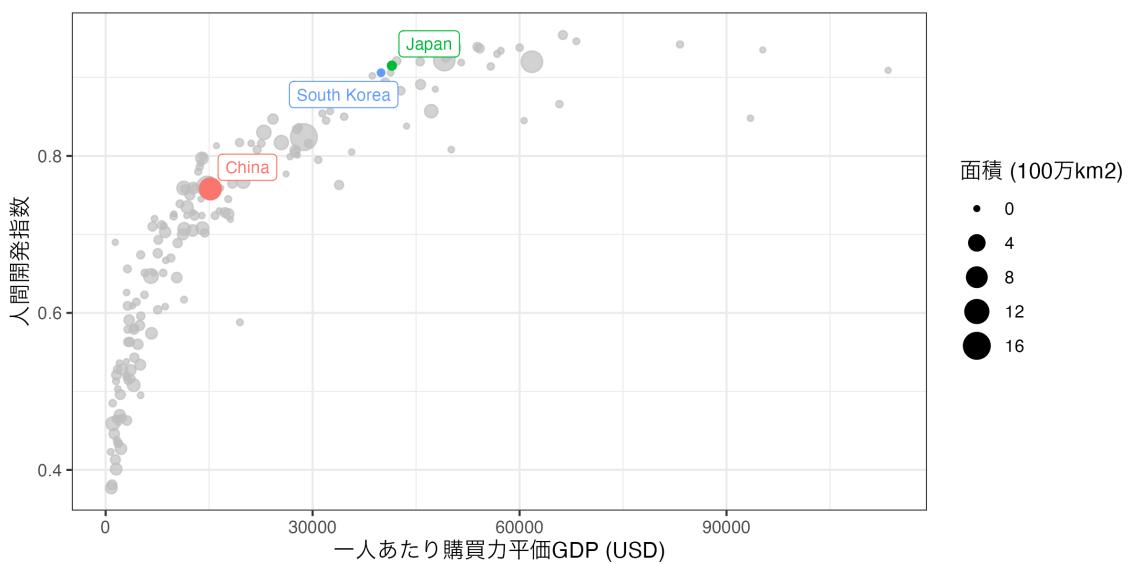
第一引数は条件式であり、これは `dplyr::filter()` の条件式をそのまま使えます。そして、`label_params` を指定しますが、実引数はリスト型です。中には `geom_labels()` 幾何オブジェクトに使用する引数が使えます。たとえば、ハイライトされた点にはラベルが付きますが、`size` 引数を入れてラベルの大きさを指定できます。それでは実際にやってみましょう。条件式は `Country %in% c("Japan", "South Korea", "China")` であります、ラベルの大きさは 3 にします。

```

1 Country_df %>%
2   mutate(Area2 = Area / 1000000) %>%
3   ggplot() +
4   geom_point(aes(x = PPP_per_capita, y = HDI_2018,
5                 size = Area2, color = Country)) +
6   gghighlight(Country %in% c("Japan", "South Korea", "China"),
7               label_params = list(size = 3)) +
8   labs(x = "一人あたり購買力平価 GDP (USD)", y = "人間開発指数",
9         size = "面積 (100 万 km2)", color = "OECD 加盟有無") +
10  theme_bw()

```

## label\_key: Country



非常に簡単なやり方で点のハイライトが出来ました。これは後ほど紹介する折れ線グ

ラフだけでなく、様々な幾何オブジェクトにも対応しています。湯谷さんの解説ページを参照して下さい。

---

## 18.8 折れ線グラフ

続きまして、折れ線グラフについて解説します。折れ線グラフは横軸が順序付き変数、縦軸は連続変数になります。横軸は順序付きであれば、年月日のような離散変数でも、連続変数でも構いません。注意すべき点は横軸上の値はグループ内において1回のみ登場するという点です。たとえば、株価のデータなら「2020年8月8日」はデータは1回だけ登場します。世界各国の株価データなら、グループ（国）内において「2020年8月8日」は一回のみ登場することになります。

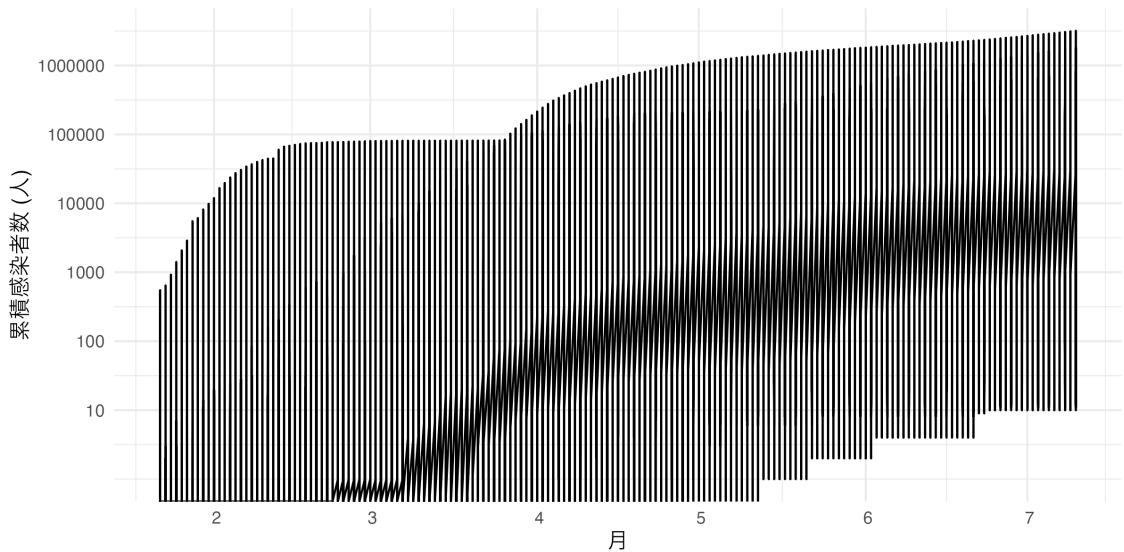
必要なマッピング要素は線の傾きが変化しうるポイントの横軸上の位置（x）と縦軸上の位置（y）です。ここではCOVID-19の新規感染者数データを使用します。まず、Date変数が文字型になっているため、これをDate型に変換します。文字型の場合、順序関係がないからです。基本的に折れ線グラフの横軸になり得るデータ型はnumeric、順序付きfactor、Date、complex型だと考えていただいても結構です。

```
1 COVID19_df <- COVID19_df %>%
2   mutate(Date = as.Date(Date))
```

それでは図を作成してみましょう。横軸は日付（Date）、縦軸は累積感染者数（Confirmed\_Total）にしましょう。また、縦軸のスケールは底10の対数を取ります。感染者数が数百万人となるアメリカと、数万人の国が同じグラフになると、見にくくなるからです。

```
1 COVID19_df %>%
2   ggplot() +
3   geom_line(aes(x = Date, y = Confirmed_Total)) +
4   scale_y_continuous(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
5                     labels = c("10", "100", "1000", "10000",
6                               "100000", "1000000"),
```

```
7     trans = "log10") +  
8     labs(x = "月", y = "累積感染者数 (人)") +  
9     theme_minimal()  
  
## Warning: Transformation introduced infinite values in continuous y-  
axis
```



?????????????????????

なんでしょう...。想像したものと違いますね。また、警告メッセージも出力されますが、これは 0 を対数化すると -Inf になるから出力されるものです。大きな問題はないので、ここでは無視しましょう。

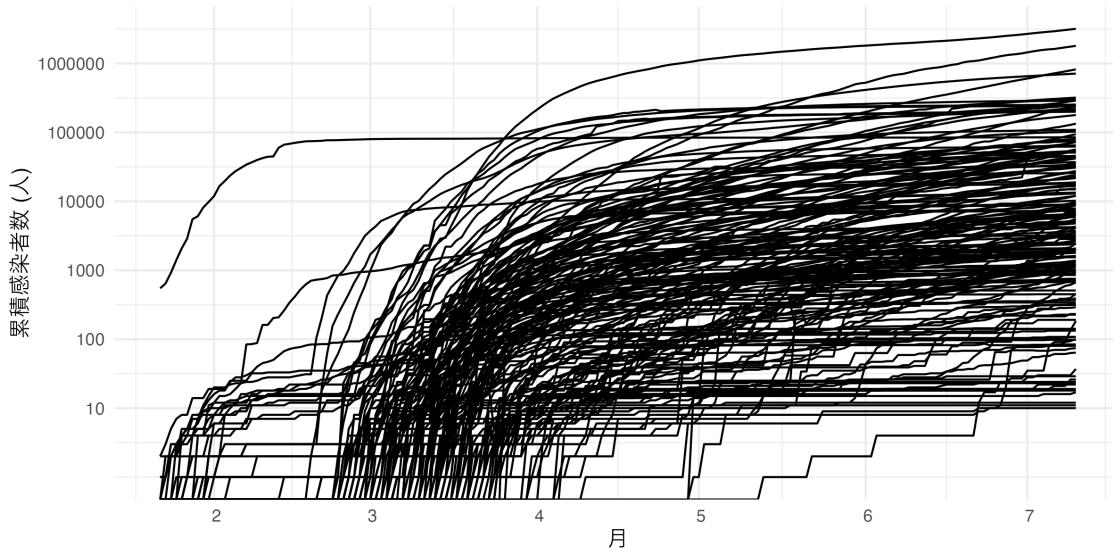
先ほど申し上げた通り、横軸の値はデータ内に 1 回のみ登場すべきです。しかし、COVID19\_df の場合、(たとえば) 2020 年 5 月 1 日の行が国の数だけあります。この場合は、ちゃんと国 (Country) ごとに線を分けるように指定する必要があります。ここで使うマッピング要素が group です。group で指定した変数の値ごとに異なる折れ線グラフを出力してくれます。

```
1 COVID19_df %>%  
2   ggplot() +
```

```

3  geom_line(aes(x = Date, y = Confirmed_Total, group = Country)) +
4    scale_y_continuous(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
5                      labels = c("10", "100", "1000", "10000",
6                        "100000", "1000000"),
7                      trans = "log10") +
8    labs(x = "月", y = "累積感染者数 (人)") +
9    theme_minimal()

```



これで折れ線グラフは出力されました。どの線がどの国かが分かりませんね。`group`の場合、凡例が表示されないので、`group`ではなく、`color`で色分けしてみましょう。

```

1 COVID19_df %>%
2   ggplot() +
3   geom_line(aes(x = Date, y = Confirmed_Total, color = Country)) +
4   scale_y_continuous(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
5                     labels = c("10", "100", "1000", "10000",
6                       "100000", "1000000"),
7                     trans = "log10") +
8   labs(x = "月", y = "累積感染者数 (人)") +
9   theme_minimal()

```

Angola	Congo (Brazzaville)	Angola	Iceland	Lesotho	Morocco	Uruguay
Argentina	Congo (Kinshasa)	Argentina	Fiji	India	Liberia	Marocco
Armenia	Costa Rica	Armenia	Finland	Indonesia	Libya	Mozambique
Australia	Cote d'Ivoire	Australia	France	Iran	Liechtenstein	Qatar
Austria	Croatia	Austria	Gabon	Iraq	Lithuania	Romania
Azerbaijan	Cuba	Azerbaijan	Gambia	Ireland	Luxembourg	Russia
Bahrain	Cyprus	Bahrain	Georgia	Israel	Madagascar	Namibia
Bangladesh	Czechia	Bangladesh	Germany	Italy	Nicaragua	Rwanda
Barbados	Denmark	Barbados	Ghana	Jamaica	Nepal	Saint Kitts and Nevis
Belarus	Djibouti	Belarus	Greece	Japan	North Macedonia	Saint Lucia
Belgium	Dominica	Belgium	Grenada	Jordan	Maldives	Saint Vincent and the Grenadines
Bolivia	Dominican Republic	Bolivia	Guatemala	Kazakhstan	Malta	San Marino
Bosnia and Herzegovina	Ecuador	Bosnia and Herzegovina	Guinea	Kenya	Mauritania	Sao Tome and Principe
Bulgaria	Egypt	Bulgaria	Guinea-Bissau	Kosovo	Mauritius	Saudi Arabia
Burkina Faso	El Salvador	Burkina Faso	Guyana	Kuwait	Mexico	Senegal
Burundi	Equatorial Guinea	Burundi	Haiti	Kyrgyzstan	Moldova	Serbia
Cambodia	Eritrea	Cambodia	Holy See	Laos	Monaco	Seychelles
Cameroon						Sierra Leone
Canada						Singapore
Chad						Slovakia
Chile						
China						
Colombia						
Croatia						
Cuba						
Cyprus						
Czechia						
Denmark						
Djibouti						
Dominica						
Dominican Republic						
Ecuador						
Egypt						
El Salvador						
Equatorial Guinea						
Eritrea						
Finland						
France						
Germany						
Georgia						
Germany						
Ghana						
Greece						
Grenada						
Guatemala						
Guinea						
Guinea-Bissau						
Haiti						
Holy See						
Iceland						
India						
Indonesia						
Iran						
Iraq						
Ireland						
Jamaica						
Japan						
Jordan						
Kazakhstan						
Kenya						
Kosovo						
Kuwait						
Kyrgyzstan						
Laos						
Liberia						
Liechtenstein						
Lithuania						
Luxembourg						
Maldives						
Malta						
Mali						
Mauritania						
Mauritius						
Mexico						
Moldova						
Monaco						
Morocco						
Mozambique						
Namibia						
Nepal						
Netherlands						
New Zealand						
Nicaragua						
Niger						
Nigeria						
North Macedonia						
Oman						
Pakistan						
Panama						
Papua New Guinea						
Paraguay						
Peru						
Romania						
Russia						
Rwanda						
Saint Kitts and Nevis						
Saint Lucia						
Saint Vincent and the Grenadines						
San Marino						
Sao Tome and Principe						
Saudi Arabia						
Senegal						
Serbia						
Seychelles						
Singapore						
Slovakia						

????????????????????????

なんか出ましたが、国数が多すぎて凡例だけで図が埋め尽くされています。この場合、方法は3つあります。1つ目の方法は図のサイズを大きくする方法です。これは保存の際、サイズを再調整するだけですので、ここでは割愛します。また、それでも凡例内の項目が非常に多いグラフをグラフの種類と関係なく推奨されません。2つ目は国を絞る方法であり、3つ目は`gghighlight()`で一部の国のみハイライトする方法です。まずは、G7（カナダ、フランス、ドイツ、イタリア、日本、イギリス、アメリカ）のみデータを絞って折れ線グラフを作ってみましょう。

```

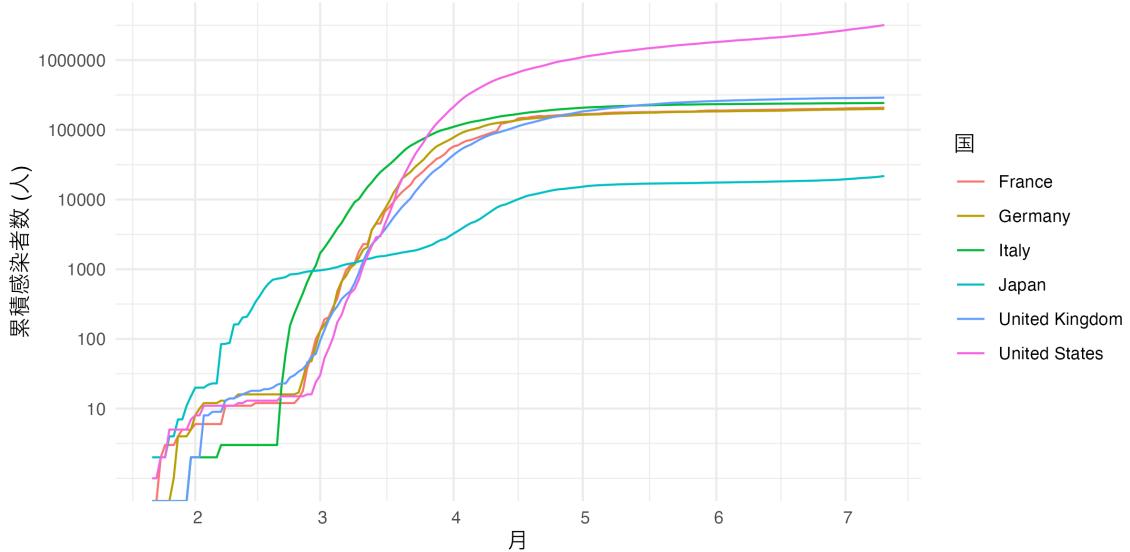
1 G7 <- c("Canada", "France", "Germany", "Italy", "Japan",
2                               "United Kingdom", "United States")
3
4 COVID19_df %>%
5   filter(Country %in% G7) %>%
6   ggplot() +
7   geom_line(aes(x = Date, y = Confirmed_Total, color = Country)) +
8   scale_y_continuous(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
9                     labels = c("10", "100", "1000", "10000",
10                           "100000", "1000000"),
11                     trans = "log10") +

```

```

12   labs(x = "月", y = "累積感染者数 (人)", color = "国") +
13   theme_minimal()

```

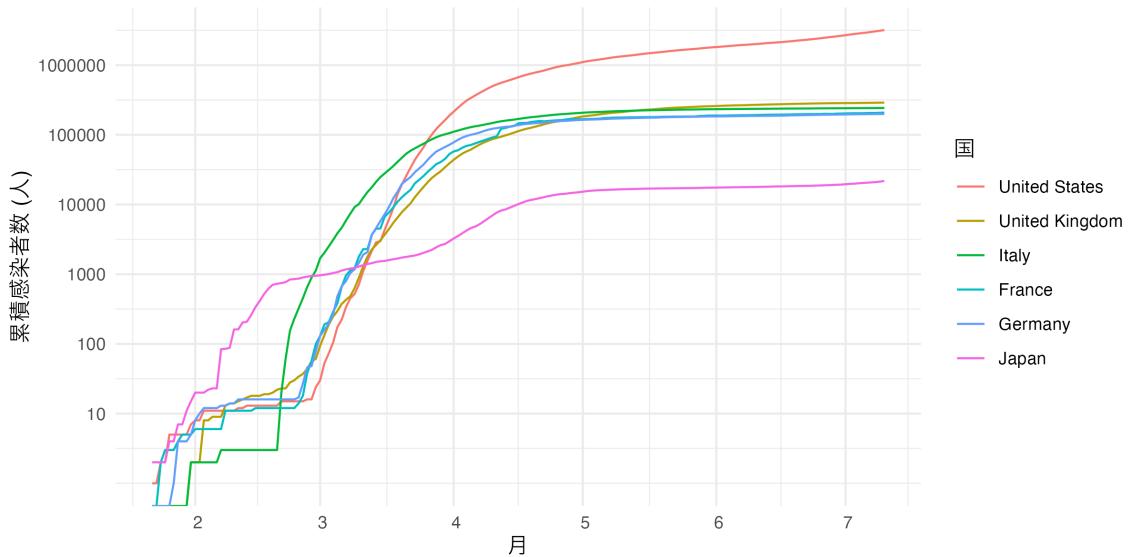


これだけでも G7 国の COVID-19 の状況が比較可能ですが、この図は改善の余地があります。それは凡例の順番です。できれば、一番最新の日付のデータを基準に凡例の順番を揃えることが出来たら、どの線がどの国かが分かりやすくなります。そこで登場するのは第14章で紹介しました `fct_reorder2` 関数です。実際にやってみましょう。

```

1 COVID19_df %>%
2   filter(Country %in% G7) %>%
3   mutate(Country = fct_reorder2(Country, Date, Confirmed_Total, last2)) %>%
4   ggplot() +
5   geom_line(aes(x = Date, y = Confirmed_Total, color = Country)) +
6   scale_y_continuous(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
7                     labels = c("10", "100", "1000", "10000",
8                               "100000", "1000000"),
9                     trans = "log10") +
10  labs(x = "月", y = "累積感染者数 (人)", color = "国") +
11  theme_minimal()

```

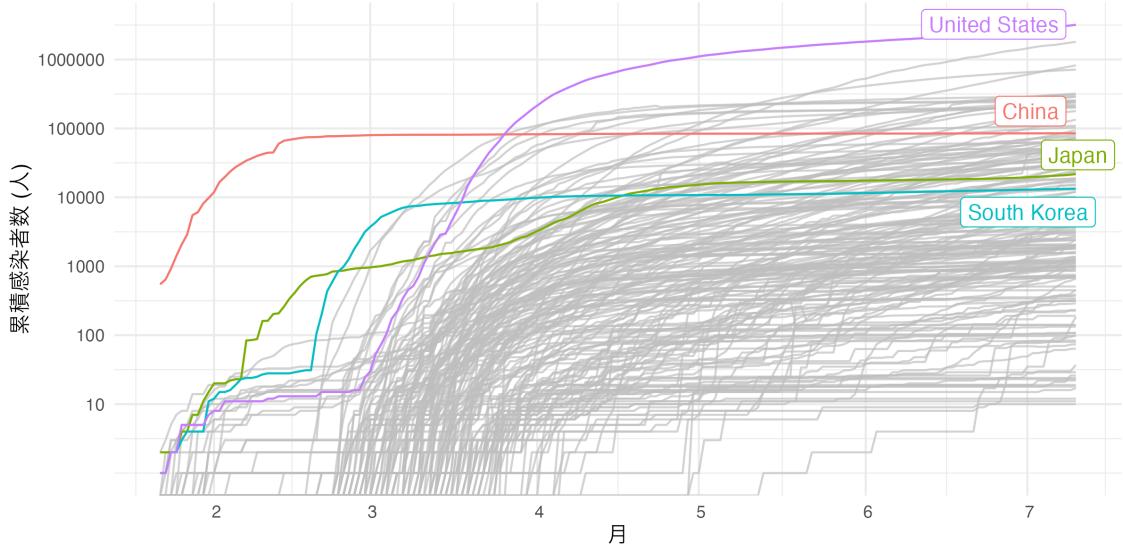


これでより読みやすい図が出来上りました。

### 18.8.1 一部の線をハイライトする

グループが多すぎる場合の対処法、その2つ目はデータを全て使い、一部の国のみハイライトする方法です。線のハイライトにも `gghighlight()` が使えます。同じく日米中韓の線のみハイライトしてみましょう。

```
1 COVID19_df %>%
2   ggplot() +
3   geom_line(aes(x = Date, y = Confirmed_Total, color = Country)) +
4   gghighlight(Country %in% c("Japan", "China", "South Korea",
5                           "United States")) +
6   scale_y_continuous(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
7                      labels = c("10", "100", "1000", "10000",
8                           "100000", "1000000"),
9                      trans = "log10") +
10  labs(x = "月", y = "累積感染者数 (人)") +
11  theme_minimal()
```



情報量の損失を最小化しながら、一部の国のみをハイライトすることによって世界における日米中韓のトレンドが確認出来ました。

## 18.9 図の保存

綺麗な図も出来上がりましたので、みんなに自慢してみましょう。ただ、自慢のためにパソコンを持ち歩くのは大変なので、ファイルと保存してみんなに配りましょう。自慢する相手がいないなら SONG に自慢しても結構です。図のフォーマットは PNG か PDF が一般的です。JPEG など圧縮フォーマットは絶対にダメです。PNG と PDF は前者はピクセルベース、後者がベクトルベースで、PDF の方が拡大してもカクカクせず綺麗です。ただし、一部の文書作成ソフトウェアで PDF を図として扱えない点や、サイズの問題（複雑な図になるほど、PDF の方がサイズが大きくなる）もあるので、PNG を使うケースも多いです。むろん、LaTeX で文章を作成するなら、PDF 一択です。

保存方法として以下の 2 つを紹介します。

1. `ggsave()` を利用: 簡単。ただし、macOS で日本語が含まれる PDF として保存する場合は問題が生じる。
2. `quartz()` を利用: macOS、かつ日本語が含まれる PDF ならこちらを利用

### 18.9.1 ggsave() を利用した図の保存

ggsave() が{ggplot2}が提供する図の保存関数です。まずは、使い方から紹介します。

```
1 # ggsave() を利用した保存方法
2 ggsave(filename = "ファイル名",
3         plot      = 図のオブジェクト名,
4         device    = "pdf"や"png"など,
5         width     = 図の幅,
6         height    = 図の高さ、
7         units     = "in"や"cm"など、幅/高さの単位)
```

deviceで指定可能なフォーマットとしては"png"、"eps"、"ps"、"tex"、"pdf"、"jpeg"、"tiff"、"bmp"、"svg"があります。また、unitsは"in"（インチ）、"cm"、"mm"が指定可能です。他にもdpi引数でdpi (dots per inch; 1平方インチにどれだけのドットの表現ができるか) の指定も可能ですが、デフォルトは300となっており、出版用としては一般的なdpiとなっています。それでは、本章の最初に作成した棒グラフをBarPlot1という名で保存し、Figsフォルダー内にBarPlot1.pngとして保存してみます。幅と高さは5インチにします。

```
1 BarPlot1 <- Country_df %>%
2   ggplot() +
3   geom_bar(aes(x = Continent)) +
4   labs(x = "大陸", y = "ケース数") +
5   theme_bw()
6
7 ggsave(filename = "Figs/BarPlot.png",
8         plot      = BarPlot1,
9         device    = "png",
10        width     = 5,
11        height    = 5,
12        units     = "in")
```

### 18.9.2 quartz() を利用した図の保存

macOS を使用し、日本語などのマルチバイトの文字が含まれる図の場合、`ggsave()` が正しく作動しません。この場合は `quartz()` と `dev.off()` 関数を利用して図を保存します。この関数の使い方は以下の通りです。

```
1 # quartz() を利用した保存方法
2 quartz(type = "pdf", file = "ファイル名", width = 図の幅, height = 図の高さ)
3 作図のコード、または図オブジェクトの呼び出し
4 dev.off()
```

`quartz()` の `width` と `height` 引数の場合、単位はインチ (=2.54cm) です。

たとえば、最初に作成した図を `Figs` フォルダーの `BarPlot1.pdf` という名で保存し、幅と高さを 5 インチにするなら以下のようない下のコードになります。

```
1 # 方法 1
2 quartz(type = "pdf", file = "Figs/BarPlot1.pdf", width = 5, height = 5)
3 Country_df %>%
4   ggplot() +
5   geom_bar(aes(x = Continent)) +
6   labs(x = "大陸", y = "ケース数") +
7   theme_bw()
8 dev.off()
```

または、予め図をオブジェクトとして保存しておいたなら（先ほどの `BarPlot1` オブジェクトのように）、以下のようない下のコードも可能です。

```
1 # 方法 2
2 quartz(type = "pdf", file = "Figs/BarPlot1.pdf", width = 5, height = 5)
3 BarPlot1
4 dev.off()
```

---

## 18.10 まとめ

以上の話をまとめると、データが与えられた場合、ある図を作成するためには少なくとも以下の情報が必要です。

1. どの幾何オブジェクトを使用するか
2. その幾何オブジェクトに必要なマッピング要素は何か

座標系や、スケール、ラベルの設定なども最終的には必要ですが、これらは設定しなくても`{ggplot2}`自動的に生成してくれます。しかし、幾何オブジェクトとマッピングは必須です。幾何オブジェクトはグーグルで「ggplot 棒グラフ」や「ggplot 等高線図」などで検索すれば、どのような幾何オブジェクトが必要化が分かります。問題はマッピングです。この幾何オブジェクトに使える引数は何か、そしてどの引数を`aes()`の中に入れるべきかなどはコンソールで`?geom_barplot`などで検索すれば分かります。

筆者のオススメは以下のチートシートを印刷し、手元に置くことです。

- `ggplot2 cheatsheet`
- `ggplot2 aesthetics cheatsheet`

2番目の資料は非常に分かりやすい資料ですが、Google Drive 経由で公開されているため、いつリンクが切れかが不明です。念の為に本ページにも資料を転載します。

### ggplot2 aesthetics cheat sheet

Use this table to find the right aesthetics for your geoms:

**Aesthetics that usually must be mapped to the data: use inside aes()**

**Aesthetics that can be mapped to the data: use in or outside aes()**

**Aesthetics that cannot be mapped to the data: use outside aes()**

e.g., `ggplot(mpg, aes(x = class, y = displ)) + geom_col(aes(fill = class), width = .9)`

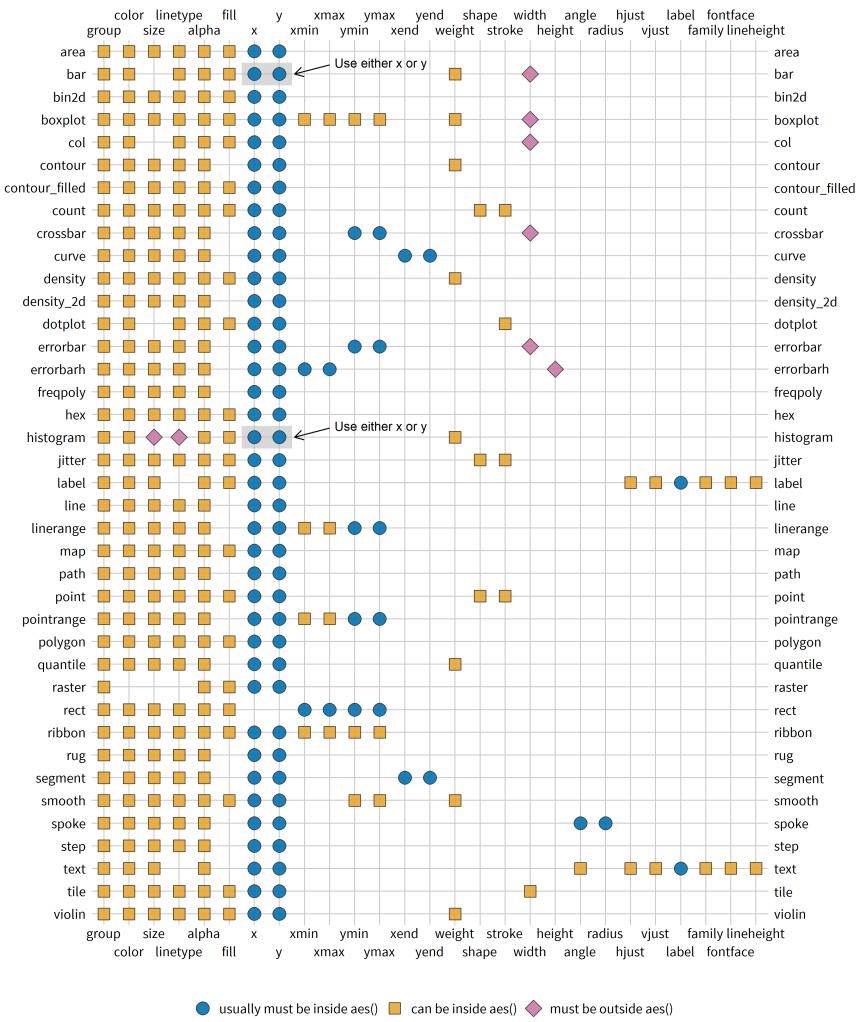


図 18.1: ggplot2 aesthetics cheatsheet

青い点は「ほとんど」のケースにおいて `aes()` の中に位置する引数です。ただし、`geom_bar()` と `geom_histogram()` は自動的に度数を計算してくれるため、`x` と `y` 一方

だけでも可能です。黄色い資格は `aes()` の中でも、外でも使うことが可能な引数です。`aes()` の中ならマッピング要素となるため、データの変数と対応させる必要があります。ピンク色のひし形四角形は必ず `aes()` の外に位置する引数です。

---

## 練習問題



## 第 19 章

# 可視化 [応用]

第 17 章と第 18 章では{ggplot2}の概念と 5 つの代表的なグラフ（棒、ヒストグラム、箱ひげ図、散布図、折れ線）の作り方について説明しました。本章では軸の調整、座標系の調整など、幾何オブジェクト以外のレイヤーについて説明します。第 18 章で紹介しなかった図の作成方法については第 20 章で解説します。

1. ラベル
2. 座標系
3. スケール
4. テーマ
5. 図の結合

本章で使用するデータは第 18 章で使用したものと同じデータを使います。データの詳細については第 18 章を参照してください。

```
1 pacman::p_load(tidyverse)
2 Country_df <- read_csv("Data/Countries.csv")
3 COVID19_df <- read_csv("Data/COVID19_Worldwide.csv", guess_max = 10000)
```

---

## 19.1 labs(): ラベルの修正

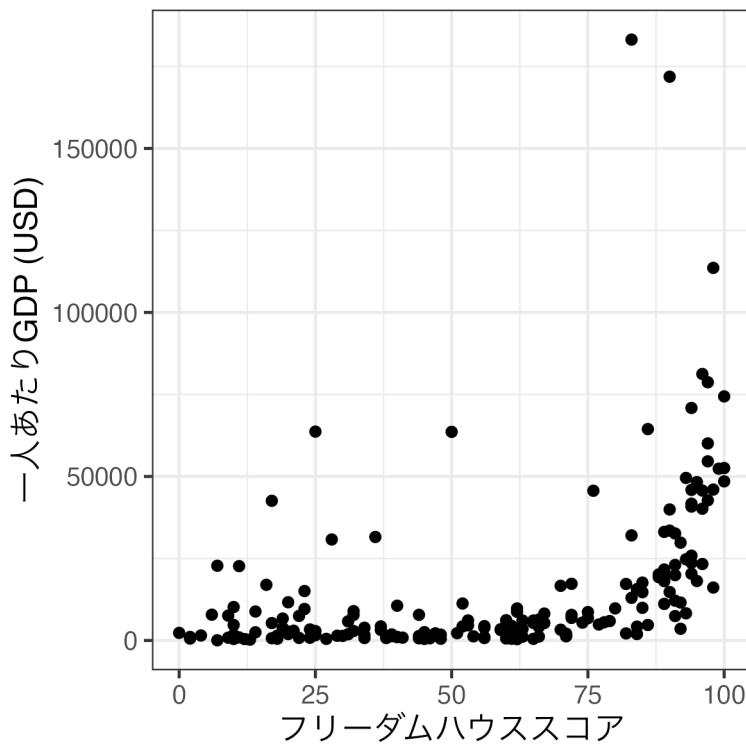
既に `labs()` レイヤーは第18章で使ったことがあるでしょう。ここでは `labs()` の仕組みについて簡単に解説します。

`labs()` 関数は軸、凡例、プロットのラベル（タイトルなど）を修正する際に使用する関数です。軸ラベルは横軸（x）と縦軸（y）のラベルを意味します。指定しない場合は、マッピングで指定した変数名がそのまま出力されます。これは凡例ラベルも同じです。`{ggplot2}` は2次元のグラフの出力に特化したパッケージであるため、出力される図には必ず横軸と縦軸があります。したがって、引数として `x` と `y` は常に指定可能です。

一方、凡例はマッピングされない場合、表示されません。幾何オブジェクトの `aes()` 内に `color`、`size`、`linetype` などの要素がマッピングされてから初めて凡例で表示されます。凡例が存在することは何かの要素にマッピングがされていることを意味します。このマッピング要素名（`color`、`size`、`linetype` など）を `labs()` の引数として使うことで凡例のラベルが修正されます。マッピングされていない要素に対してラベルを指定しても、図に影響はありません。たとえば、`Country_df` の一人あたり GDP（`GDP_per_capita`）を横軸、フリーダムハウスのスコア（`FH_Total`）を縦軸にした散布図を作成します。

```
1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = GDP_per_capita)) +
4   labs(x = "フリーダムハウススコア", y = "一人あたり GDP (USD)",
5        color = "大陸") +
6   theme_bw(base_size = 12)
```

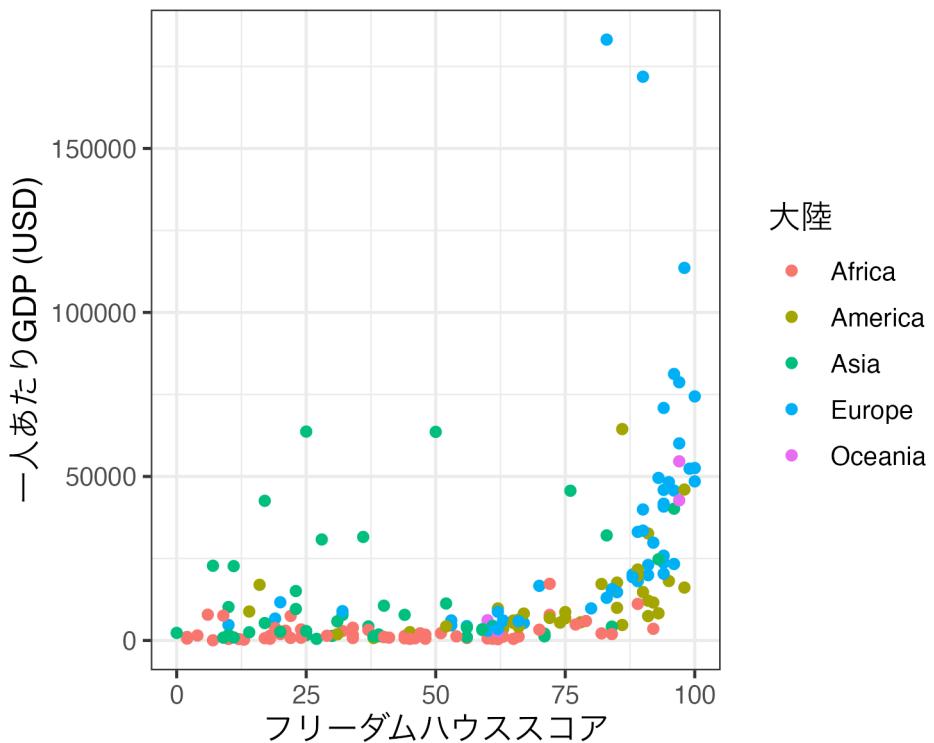
## Warning: Removed 1 rows containing missing values (geom\_point).



`geom_point()` は横軸と縦軸のみにマッピングをしているため、`labs()` に `color =` を指定しても何の変化もありません。そもそも凡例が存在しないからです。それでは大陸ごとに色分けした散布図に修正してみましょう。

```
1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = GDP_per_capita, color = Continent)) +
4   labs(x = "フリーダムハウススコア", y = "一人あたり GDP (USD)",
5        color = "大陸") +
6   theme_bw(base_size = 12)
```

## Warning: Removed 1 rows containing missing values (geom\_point).



`color` に `Continent` 変数をマッピングすることによって、各点の色は何らかの情報を持つようになりました。そして各色が `Continent` のどの値に対応しているかを示すために凡例が表示されます。凡例のラベルはデフォルトは変数名（この例の場合、「`Continent`」）ですが、ここでは「大陸」と修正されました。

ここまでが第18章で使用しました `labs()` レイヤーの使い方です。他にも `labs()` はプロットのラベルを指定することもできます。ここでいう「プロットのラベル」とはプロットのタイトルとほぼ同じです。使用可能な引数は `title`、`subtitle`、`tag` です。

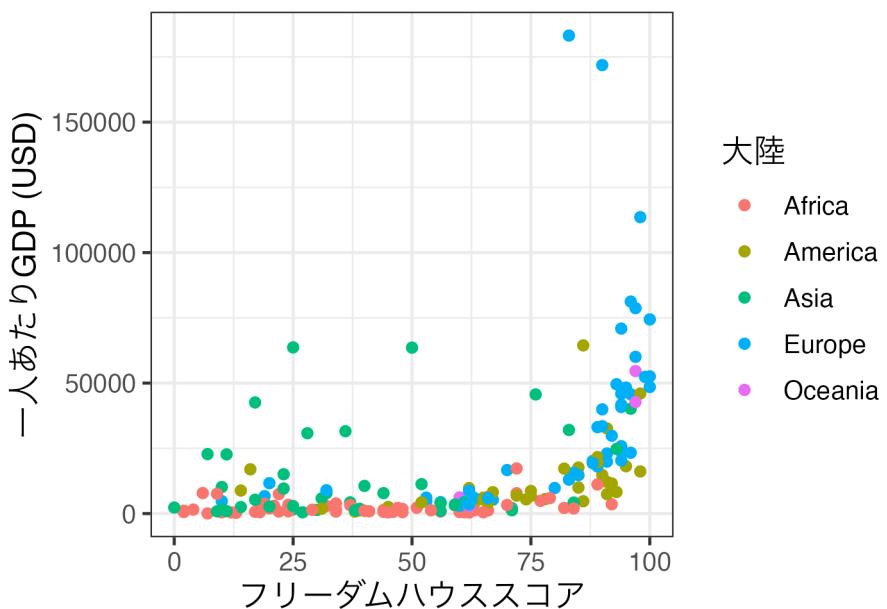
```

1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = GDP_per_capita, color = Continent)) +
4   labs(x = "フリーダムハウススコア", y = "一人あたり GDP (USD)", color = "大陸",
5         title = "民主主義の度合いと所得の関係", subtitle = "大陸別の傾向",
6         tag = "(a)") +
7   theme_bw(base_size = 12)

## Warning: Removed 1 rows containing missing values (geom_point).

```

(a) 民主主義の度合いと所得の関係  
大陸別の傾向



`title` は図のメインタイトルとおり、プロットのタイトルを意味します。上の図だと「民主主義の度合いと所得の関係」です。また、`subtitle` 引数を指定することでサブタイトルを付けることも可能です。上の図の「大陸別の傾向」がサブタイトルです。最後の `tag` は複数の図を並べる際に便利な引数です。図が横に 2 つ並んでいる場合、それぞれ (a) と (b) という識別子を付けると、文中において「図 3(a) は...」のように、引用しやすくなります。この「(a)」が `tag` 引数に対応します。複数の図を並べる方法は本章の後半にて説明します。

最後にキャプションの付け方について説明します。たとえば、外部のデータを利用して作図を行った場合、図の右下に「データ出典: ~~」や「Source: <https://www.jaysong.net>」などを付ける場合があります。このキャプションは `labs()` 関数内に `caption` 引数を指定するだけで付けることができます。たとえば、先程の図に「Source: Freedom House」を付けてみましょう。

```

1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = GDP_per_capita, color = Continent)) +

```

```

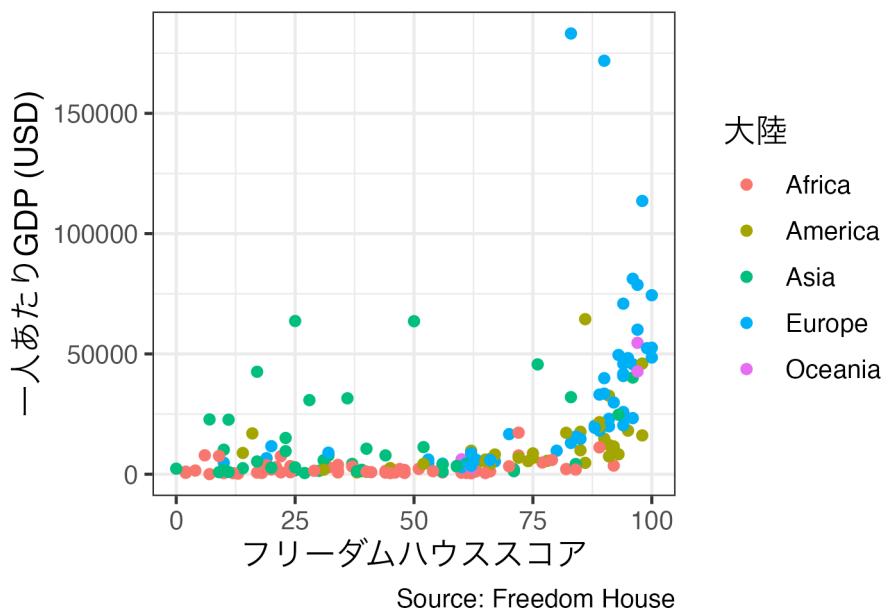
4   labs(x = "フリーダムハウススコア", y = "一人あたり GDP (USD)", color = "大陸",
5     title = "民主主義の度合いと所得の関係", subtitle = "大陸別の傾向",
6     tag = "(a)", caption = "Source: Freedom House") +
7     theme_bw(base_size = 12)

## Warning: Removed 1 rows containing missing values (geom_point).

```

(a)

民主主義の度合いと所得の関係  
大陸別の傾向



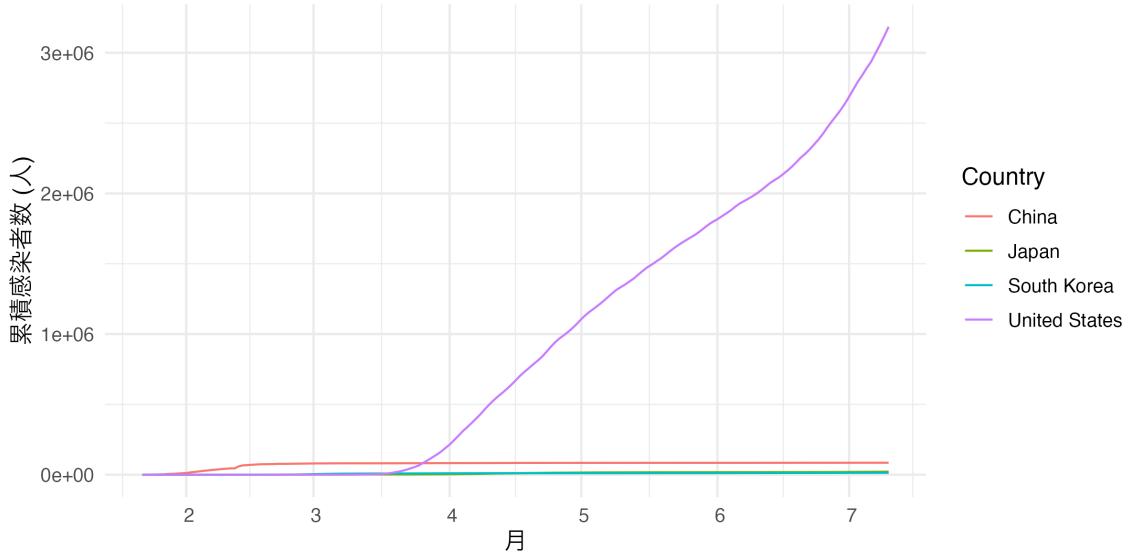
## 19.2 coord\_(): 座標系の調整

{ggplot2}はいくつかの座標系を提供しています。円グラフを作成する際に使われる極座標系 (coord\_polar()) や地図の出力によく使われる coord\_map() や coord\_sf() がその例です。中でも最も頻繁に使われる座標系はやはり縦軸と横軸は直交する直交座標系（デカルト座標系）でしょう。ここでは直交座標系の扱い方について解説します。

### 19.2.1 直交座標系の操作

まずは座標系の上限と下限を指定する方法から考えましょう。日米中間の COVID-19 累積感染者数の折れ線グラフを作成してみましょう。

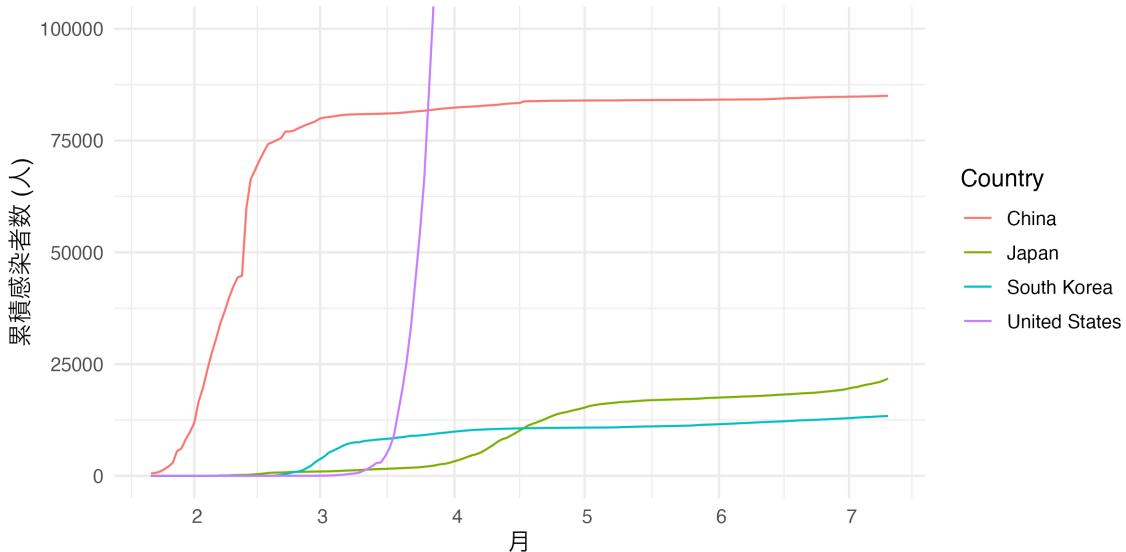
```
1 Fig1 <- COVID19_df %>%
2   mutate(Date = as.Date(Date)) %>%
3   filter(Country %in% c("Japan", "South Korea", "China", "United States")) %>%
4   ggplot() +
5   geom_line(aes(x = Date, y = Confirmed_Total, color = Country)) +
6   labs(x = "月", y = "累積感染者数 (人)") +
7   theme_minimal(base_size = 12)
8
9 print(Fig1)
```



アメリカの感染者が圧倒的に多いこともあり、日韓がほぼ同じ線に見えます。これを是正するために対数変換などを行うわけですが、対数変換したグラフは直感的ではないというデメリットがあります。それでもう一つの方法として、アメリカに関する情報は一部失われますが、縦軸の上限を 10 万にすることが考えられます。直交座標系の上限・

下限を調整する関数が `coord_cartesian()` です。横軸は `xlim`、縦軸は `ylim` 引数を指定し、実引数としては長さ 2 の numeric ベクトルを指定します。たとえば、縦軸の下限を 0、上限を 10 万にするなら、`ylim = c(0, 100000)` となります。先ほどの図は既に `Fig1` という名のオブジェクトとして保存されているため、ここに `coord_cartesian()` レイヤーを追加してみましょう。

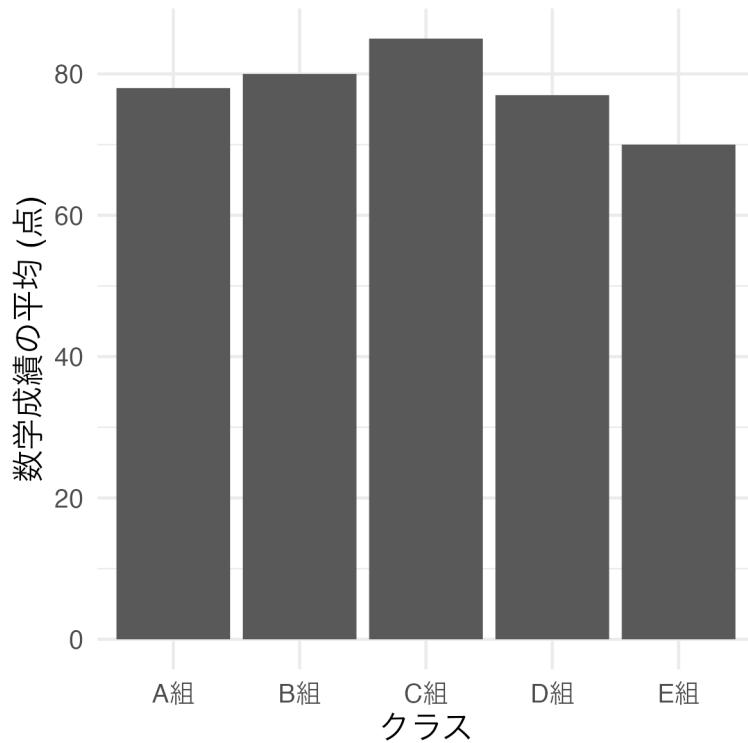
```
1 Fig1 +
2   coord_cartesian(ylim = c(0, 100000))
```



3月下旬以降、アメリカの情報は図から失われましたが、日中韓についてはよりトレンドの差が区別できるようになりました。`{ggplot2}`は座標系の上限と下限をデータの最小値と最大値に合わせて自動的に調整してくれます。たとえば、以下のような例を考えてみましょう。

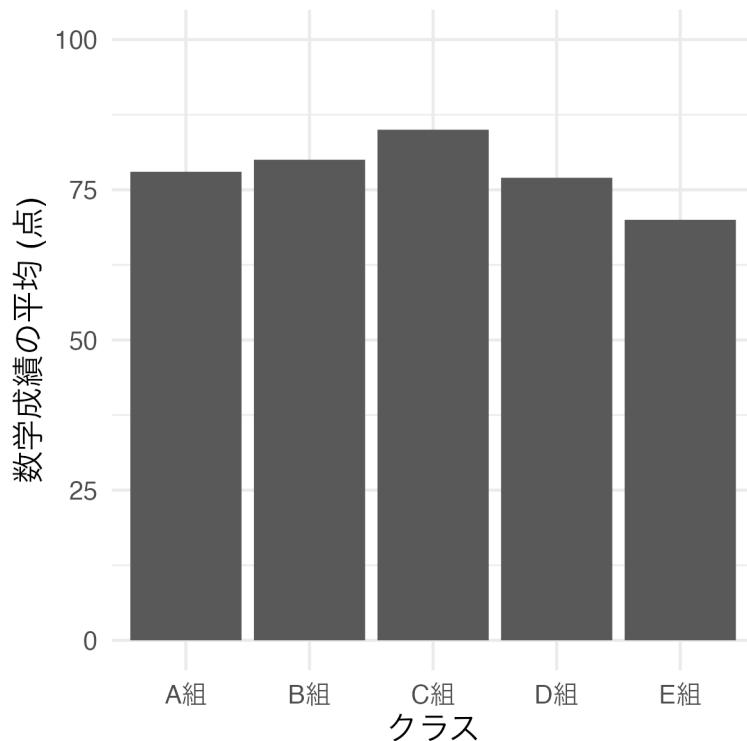
```
1 Fig2 <- tibble(Class = paste0(LETTERS[1:5], "組"),
2   Score = c(78, 80, 85, 77, 70)) %>%
3   ggplot() +
4   geom_bar(aes(x = Class, y = Score), stat = "identity") +
5   labs(x = "クラス", y = "数学成績の平均(点)") +
6   theme_minimal(base_size = 12)
7
```

```
8 print(Fig2)
```



数学成績平均値は最大 100 点まであります。手元のデータにおける最高得点が 85 点であるため、棒グラフの縦軸の上限が 85 点程度となります。この場合、上限は満点である 100 点に調整した方が良いでしょう。

```
1 Fig2 +  
2   coord_cartesian(ylim = c(0, 100))
```



このように上限を調整すると、成績の満点が何点かに関する情報が含まれ、グラフにより豊富な情報を持たせることができます。

### 19.2.2 座標系の変換

続きまして座標系の変換について説明します。座標系の変換については実は第18章でも取り上げました。対数化がその例です。例えば、連続型変数でマッピングされた横軸を底が10の対数化する場合、以下のような方法が考えれます。

1. `log10()` 関数を使用し、データレベルで値を対数化する
2. `scale_x_continuous()` レイヤーを重ね、`trans = "log10"`引数を指定する
3. `scale_x_log10()` レイヤーを重ねる
4. `coord_trans()` レイヤーを重ね、`x = "log10"`引数を指定する

どの方法でも得られる結果はさほど変わりませんが、`coord_trans()` は座標系全般を調整することができます。たとえば、`xlim` や `ylim` 引数を使って座標系の上限と下限を同時に指定することも可能です。たとえば、座標系の上限を横軸は [0, 100]、縦軸は

$[-100, 100]$  とし、全て対数化を行うとします。方法はいくつか考えられます。たとえば、`scale_*_log10()` と `coord_cartesian()` を組み合わせることもできます。

```
1 # coord_trans() を使用しない場合
2 ggplot オブジェクト +
3   scale_x_log10() +
4   scale_y_log10() +
5   coord_cartesian(xlim = c(0, 100), ylim = c(-100, 100))
```

しかし、上のコードは `coord_trans()` を使うと一行にまとめることができます。

```
1 # coord_trans() を使用する場合
2 ggplot オブジェクト +
3   coord_trans(x = "log10", y = "log10", xlim = c(0, 100), ylim = c(-100, 100))
```

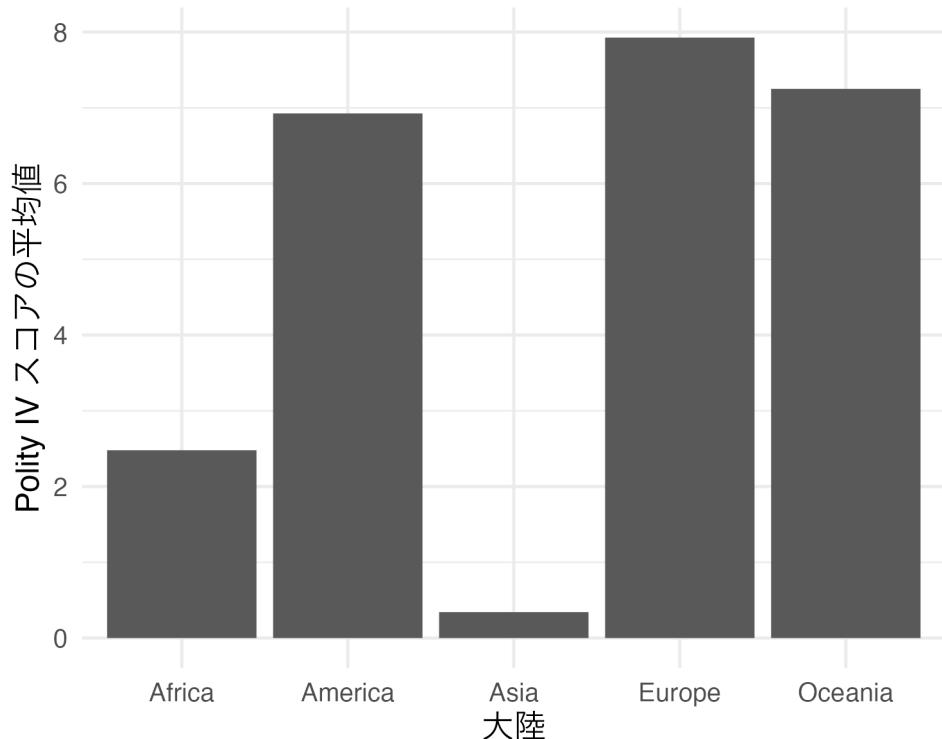
`coord_trans()` の `x`、`y` 引数は "log10" 以外にもあります。自然対数変換の "log"、反転を意味する "reverse"、平方根へ変換する "sqrt" などがあります。グラフを上下、または左右に反転する "reverse" は覚えておいて損はないでしょう。こちらは具体的には `{tidyverse}` パッケージ群に含まれている `{scales}` パッケージにある `*_trans()` 関数に対応することになります。詳細は `{scales}` パッケージのヘルプを参照してください。

### 19.2.3 座標系の回転

続きまして座標系を反時計方向回転する `coord_flip()` について紹介します。以下は `Country_df` を用い、大陸 (Continent) ごとに Polity IV スコア (Polity\_Score) の平均値を示した棒グラフです。

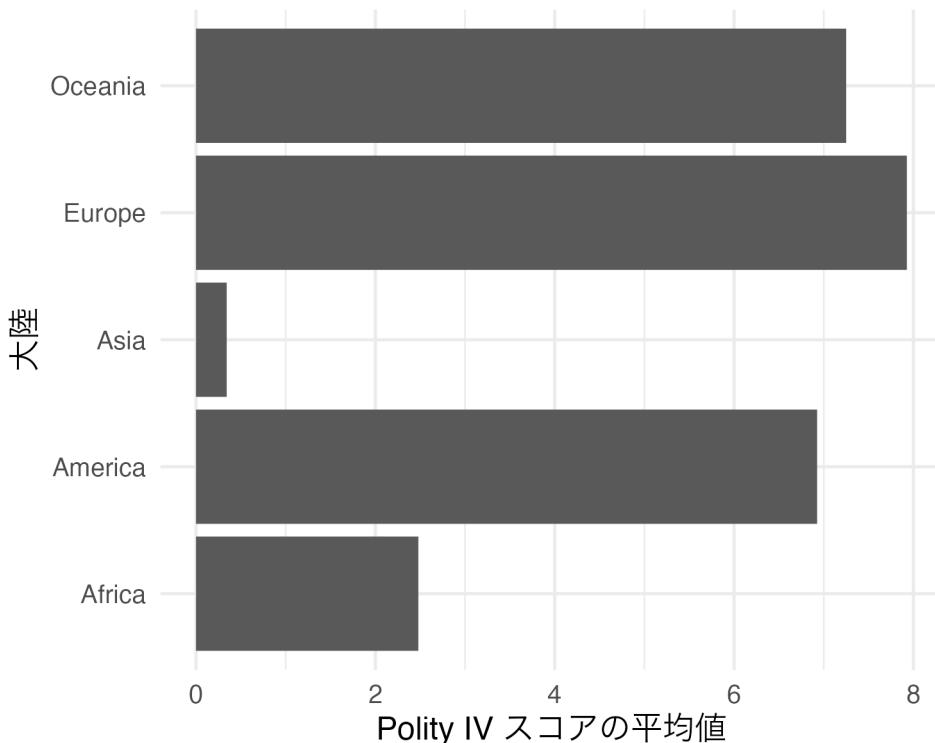
```
1 Flip_Fig <- Country_df %>%
2   group_by(Continent) %>%
3   summarise(Democracy = mean(Polity_Score, na.rm = TRUE),
4             .groups    = "drop") %>%
5   ggplot() +
6   geom_bar(aes(x = Continent, y = Democracy), stat = "identity") +
7   labs(x = "大陸", y = "Polity IV スコアの平均値") +
```

```
8     theme_minimal(base_size = 12)
9
10    print(Flip_Fig)
```



この図を反時計方向回転する場合は以上のプロットに `coord_flip()` レイヤーを追加します。

```
1 Flip_Fig +
2   coord_flip() # 座標系の回転
```



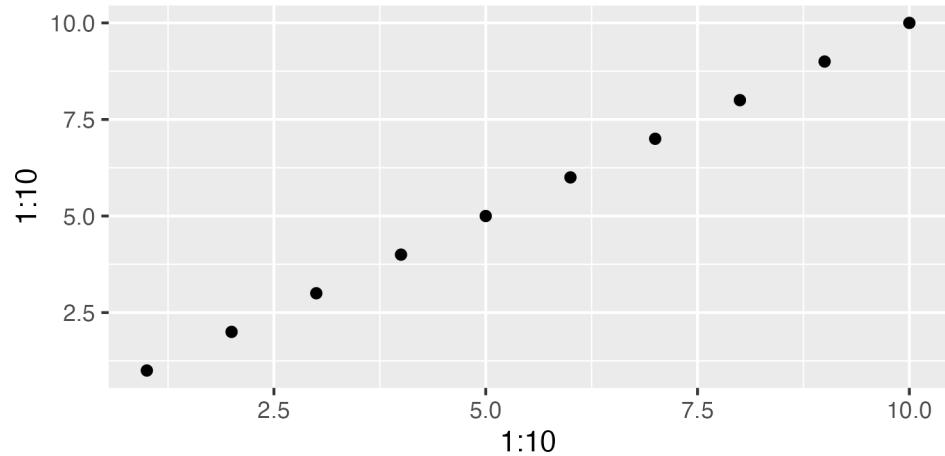
非常に簡単な方法で図を回転させることができました。しかし、実はこの `coord_flip()` 関数、最近になって使う場面がどんどん減っています。たとえば、先ほどの `geom_bar()` 幾何オブジェクトの場合、`x` を Polity IV スコアの平均値で、`y` を大陸名でマッピングすることができます。昔の{ggplot2}は横軸と縦軸にマッピングでいるデータ型が厳格に決まっていましたが、最近になってはますます柔軟となっていました<sup>1)</sup>。`coord_flip()` を使用する前に、各幾何オブジェクトのヘルプを確認し、`coord_flip()` を用いた回転が必要か否かを予め調べておくのも良いでしょう。

#### 19.2.4 座標系の固定

他にも地味に便利な機能として座標系比を固定する `coord_fixed()` を紹介します。これは出力される座標系の「横: 縦」を調整するレイヤーです。たとえば、以下のような散布図を考えてみましょう。

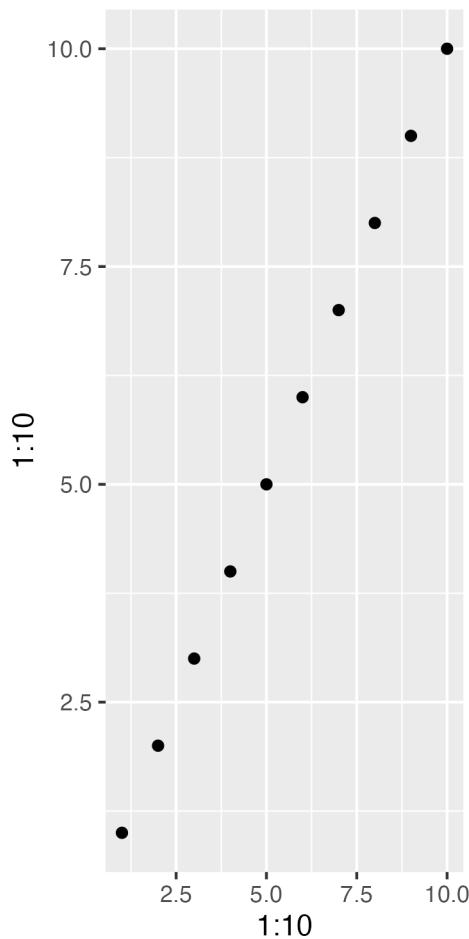
<sup>1)</sup> 例えば、回帰係数の点推定値と信頼区間を示す係数プロット（キャタピラー・プロット）を作成する際は `geom_pointrange()` と `coord_flip()` を組み合わせてきましたが、今の{ggplot2}は `geom_pointrange()` のみで自由自在に作成できます。

```
1 ggplot() +  
2   geom_point(aes(x = 1:10, y = 1:10))
```



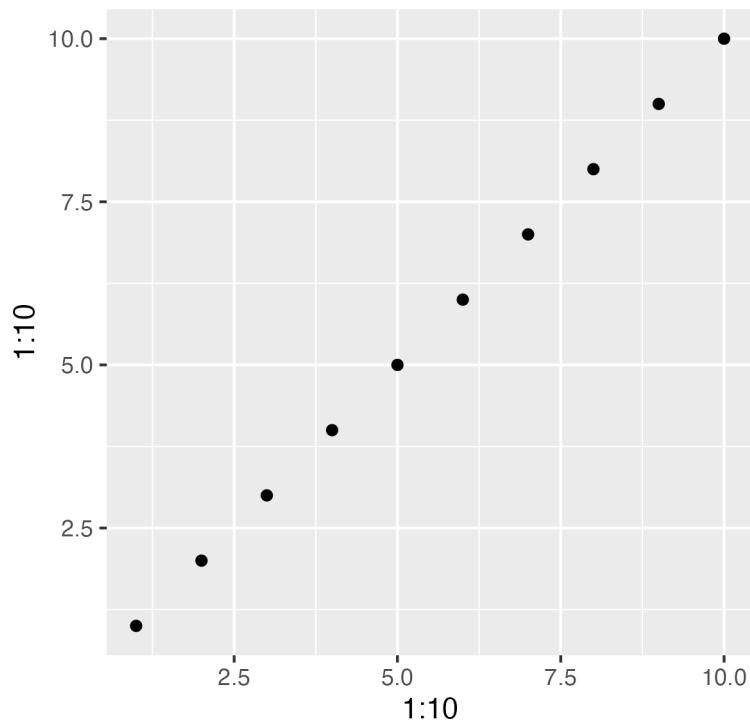
こちらは横と縦が同じスケールであります。図の大きさに応じて、見た目が変わってきます。たとえば、上の図だと、横軸における1間隔は縦軸のそれの約2倍です。もし、図を上下に大きくし、左右を縮小したら同じ図でありながら随分と見た目が変わってきます。

```
1 ggplot() +  
2   geom_point(aes(x = 1:10, y = 1:10))
```



2つの図は本質的に同じですが、図の見せ方によって、傾きが緩やかに見せたり、急に見せたりすることができます。ここで活躍するレイヤーが `coord_fixed()` です。これを追加すると横を 1 とした場合の縦の比率を指定することができます。

```
1 ggplot() +  
2   geom_point(aes(x = 1:10, y = 1:10)) +  
3   coord_fixed(ratio = 1)
```



`ratio = 1` を指定すると縦横比は 1:1 となり、図の高さや幅を変更してもこの軸は変わりません。たとえば、RStudio の Plots ペインの大きさを変更すると図の大きさが変わりますが、`coord_fixed(ratio = 1)` を指定すると 1:1 の比率は維持されるまま図が拡大・縮小されます。直接やってみましょう。

### 19.3 `scale_*`(): スケールの調整

続いて `scale_*`() 関数群を用いたスケールを解説しますが、こちらの関数は非常に多く、全てのスケールレイヤーについて解説することは難しいです。しかし、共通する部分も非常に多いです。本説ではこの共通項に注目します。

連続変数でマッピングされた横軸のスケールを調整する関数は `scale_x_continuous()` です。ここで `x` が横軸を意味し、`continuous` が連続であることを意味します。この `x` の箇所は幾何オブジェクトの `aes()` 内で指定した仮引数と一致します。つまり、`scale_x_continuous()` の `x` の箇所には `y`、`alpha`、`linetype`、`size` などがあります。そして、実引数として与えられた変数のデータ型が `continuous` の箇所に相当します。

もし、離散変数なら `discrete`、時系列なら `time`、全て手動で調整する場合は `manual` を使います。他にも 2020 年 10 月現在、最近追加されたものとして `binned` があり、こちらはヒストグラムに特化したものです (`scale_x_binned()` と `scale_y_binned()`)。つまり、スケール調整関数は `aes()` 内に登場した仮引数名とそのデータ型の組み合わせで出来ています。

たとえば、時系列の折れ線グラフにおいて横軸のスケールを調整するなら、`scale_x_time()` を使います。また、棒グラフのように横軸が名目変数なら `scale_x_manual()`、順序変数のような離散変数なら `scale_x_discrete()` を使います。また、連続変数の縦軸のスケール調整なら `scale_y_continuous()` を使います。グラフによっては `aes()` 内に `x` または `y` を指定しないケースもあります。前章において度数の棒グラフや 1 つの箱ひげ図を出す場合、前者は `x`、後者は `y` のみを指定しました。これは指定されていない `y` や `x` が存在しないことを意味しません。`{ggplot2}` が自動的に計算しマッピングを行ってくれることを意味します。`{ggplot2}` で出来上がった図は 2 次元座標系を持つため、横軸と縦軸は必ず存在します。したがって、`aes()` 内の引数と関係なく `scale_x_*` と `scale_y_*` 関数群は使用することが出来ます。

### 19.3.1 横軸・縦軸スケールの調整

軸のスケールを調整する目的は 1) 目盛りの調整、2) 目盛りラベルの調整、3) 第 2 軸の追加などがありますが、主な目的は 1 と 2 です。第 2 軸の追加はスケールが異なるグラフが重なる場合に使用しますが、一般的に推奨されません。したがって、ここでは 1 と 2 について説明します。

#### 19.3.1.1 連続変数の場合

軸に連続変数がマッピングされている場合は `scale_*_continuous()` レイヤーを追加します。\*の箇所は横軸の場合は `x`、縦軸の場合は `y` となります。それでは `Country_df` の `FH_Total` を横軸、`GDP_per_capita` を縦軸にした散布図を作成し、`Scale_Fig1` という名のオブジェクトに格納します。

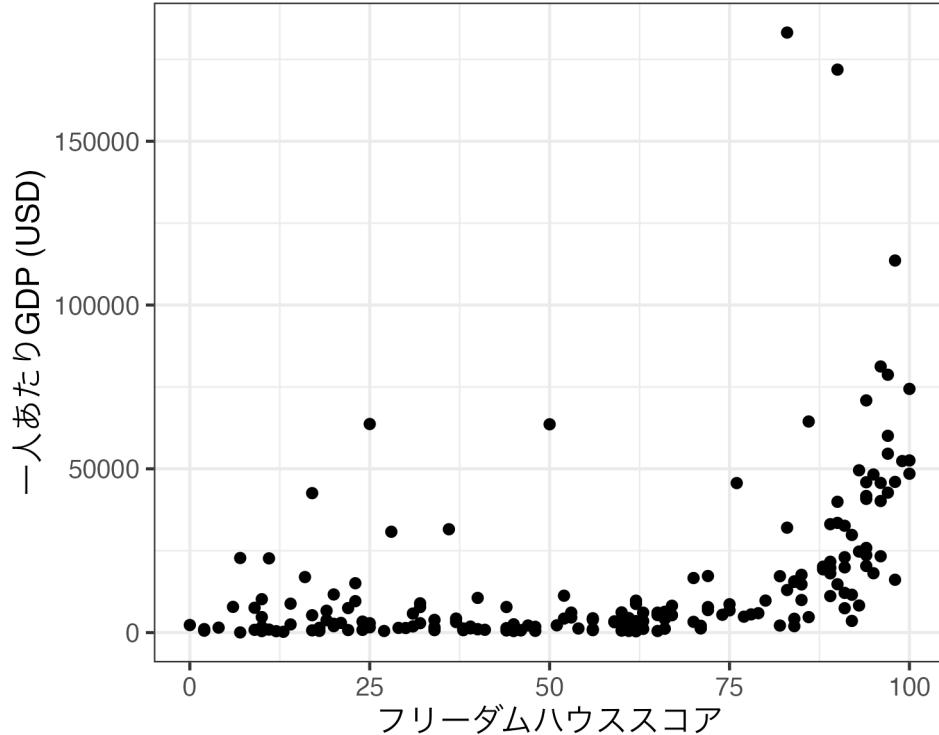
```
1 Scale_Fig1 <- Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = GDP_per_capita)) +
```

```

4   labs(x = "フリーダムハウススコア", y = "一人あたりGDP (USD)") +
5     theme_bw(base_size = 12)
6
7 print(Scale_Fig1)

```

## Warning: Removed 1 rows containing missing values (geom\_point).



Scale\_Fig1 の横軸の場合、最小値 0、最大値 100 であり、目盛りは 25 間隔となっております。ここではこの横軸を調整したいと思います。まず、プロットにおける最小値と最大値はスケールではなく座標系の問題ですので、`coord_*`() を使用します。ここでは目盛りを修正してみましょう。たとえば、目盛りを 10 間隔にし、そのラベルも 0, 10, 20, ..., 100 にします。FH\_Total は連続変数ですので、`scale_x_continuous()` を使います。使い方は以下の通りです。

```

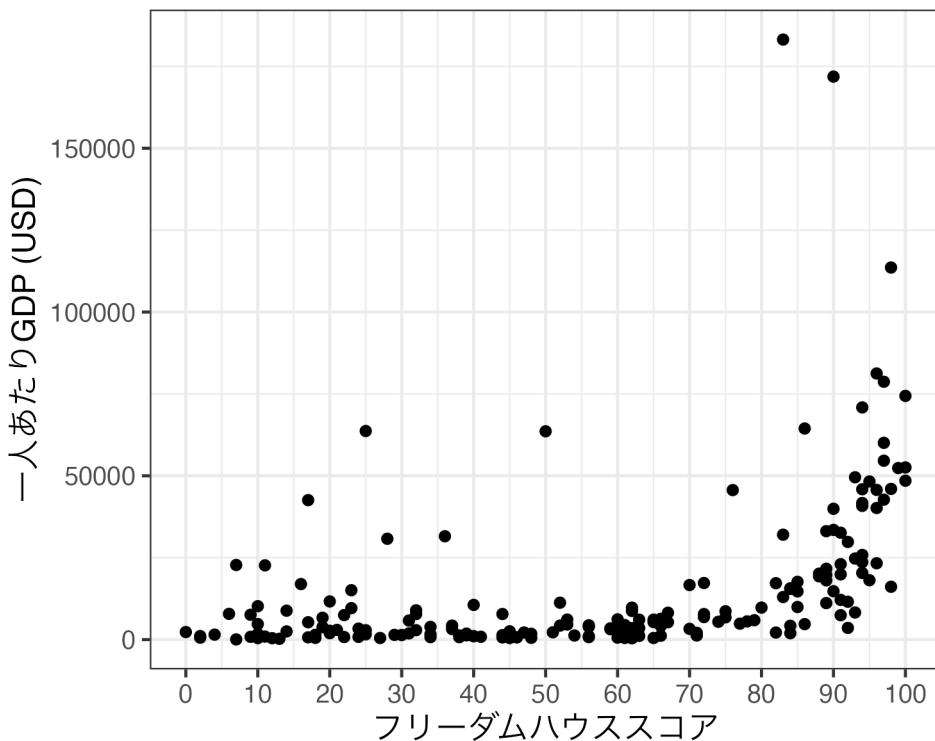
1 Scale_Fig1 +
2   scale_x_continuous(breaks = 目盛りの位置,
3                       labels = 目盛りのラベル)

```

`breaks` と `labels` の実引数としては数値型ベクトルを指定します。0 から 100 まで 10 刻みの目盛りとラベルなら、`seq(0, 100, by = 10)`、または `c(0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100)` と指定します。

```
1 Scale_Fig1 +  
2   scale_x_continuous(breaks = seq(0, 100, by = 10),  
3                       labels = seq(0, 100, by = 10))
```

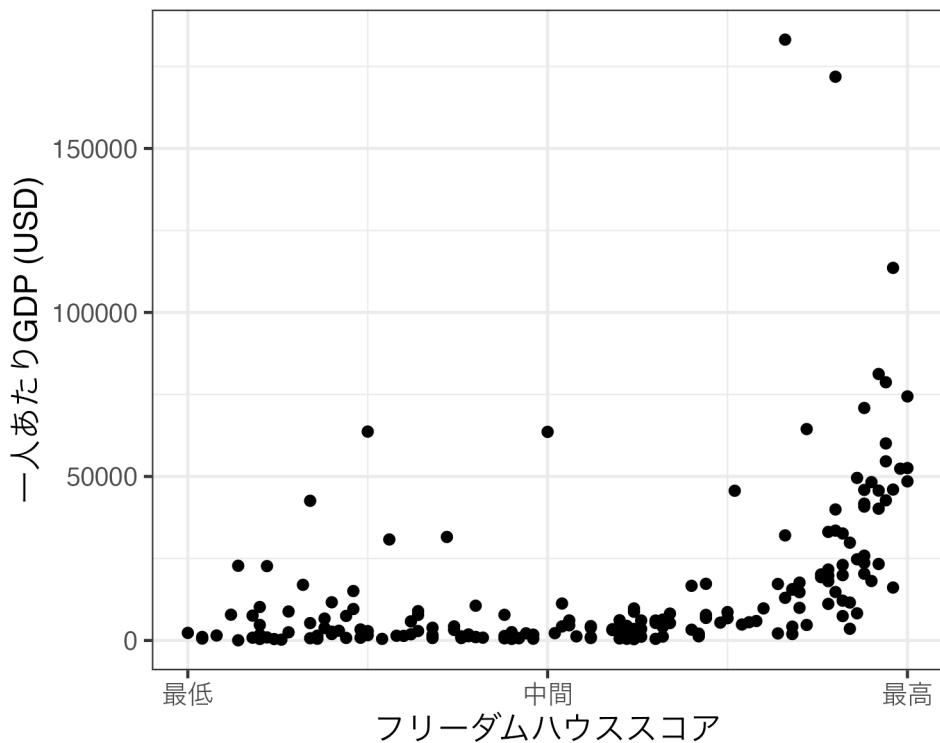
## Warning: Removed 1 rows containing missing values (geom\_point).



目盛りのラベルを文字型にすることも可能です。例えば、目盛りを 0、50、100 にし、それぞれ「最低」、「中間」、「最高」としたい場合は以下のようにします。

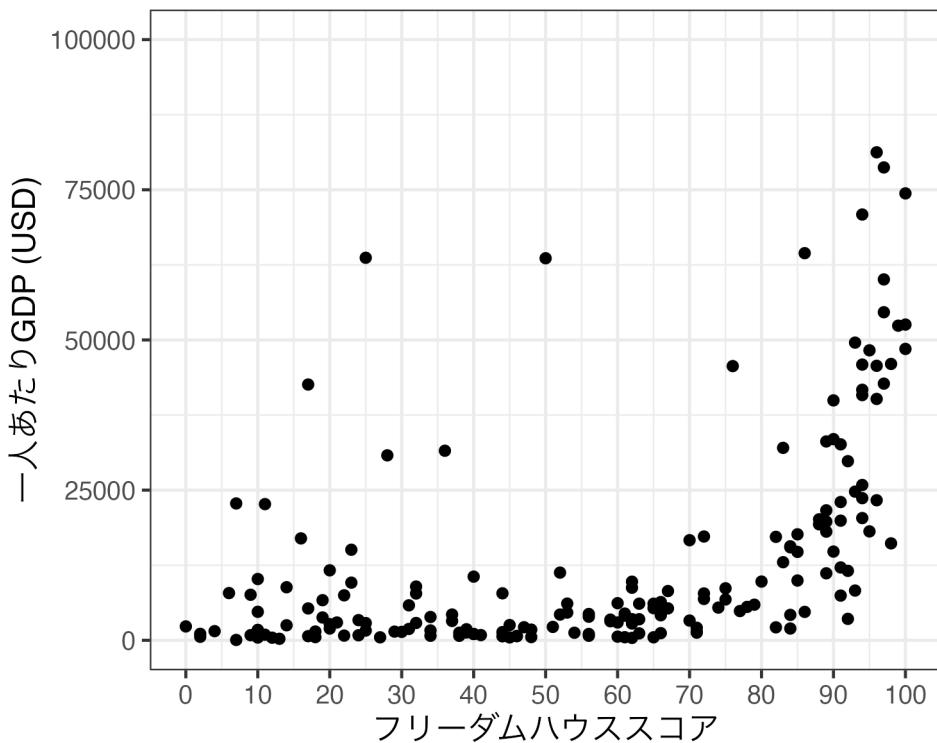
```
1 Scale_Fig1 +  
2   scale_x_continuous(breaks = c(0, 50, 100),  
3                       labels = c("最低", "中間", "最高"))
```

## Warning: Removed 1 rows containing missing values (geom\_point).



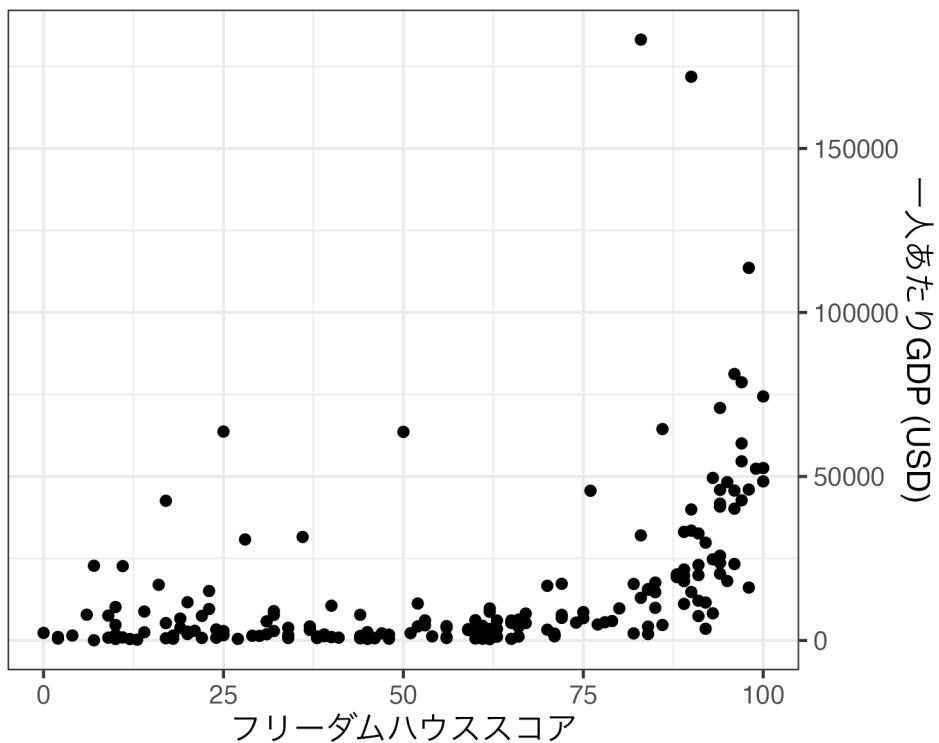
`scale_x_continuous()` は目盛りの調整が主な使い道ですが、他にも様々な機能を提供しています。たとえば、座標系の最小値と最大値の指定は `coord_*`() を使うと説明しましたが、実は `scale_*_continuous()` でも `limits` 引数で指定することも可能です。たとえば、縦軸の範囲を 0 ドルから 10 万ドルにしたい場合は `scale_y_continuous()` の中に `limits = c(0, 100000)` を指定します。

```
1 Scale_Fig1 +
2   scale_x_continuous(breaks = seq(0, 100, by = 10),
3                       labels = seq(0, 100, by = 10)) +
4   scale_y_continuous(limits = c(0, 100000))
## Warning: Removed 4 rows containing missing values (geom_point).
```



他にも目盛りと目盛りラベルの位置を変更することも可能です。これは `position` 引数を使います。基本的に横軸の目盛りは下 ("bottom")、縦軸は左 ("left") ですが、"top" や "right" を使うことも可能です。もし、縦軸の目盛りとラベル、軸のラベルを右側にしたい場合は `scale_y_continuous()` の中に `position = "right"` を指定します。

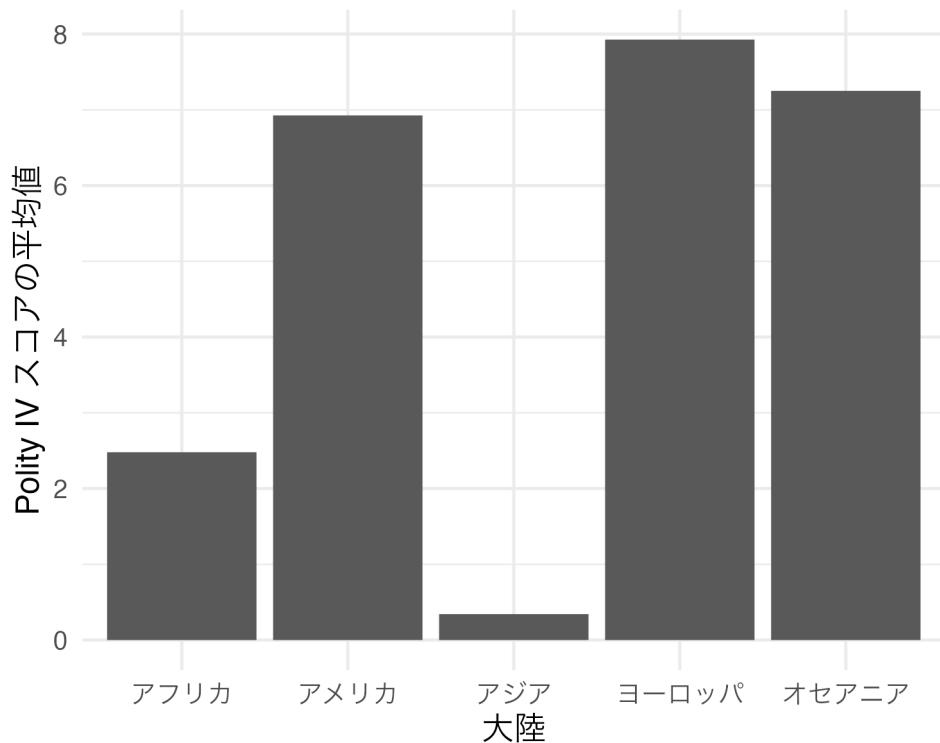
```
1 Scale_Fig1 +  
2   scale_y_continuous(position = "right")  
  
## Warning: Removed 1 rows containing missing values (geom_point).
```



### 19.3.1.2 離散変数の場合

もし、軸が名目変数や順序変数のような離散変数でマッピングされている場合は、`scale_*_discrete()` を使います。たとえば、座標系の回転の例で使いました `Flip_Fig` の場合、横軸の大陸名が英語のままになっています。これを日本語にする場合、データレベルで大陸名を日本語で置換することも可能ですが、`scale_x_discrete()` を使うことも可能です。使い方は `scale_*_continuous()` と同じであり、`breaks` と `labels` 引数を指定するだけです。

```
1 Flip_Fig +  
2   scale_x_discrete(breaks = c("Africa", "America", "Asia", "Europe", "Oceania"),  
3                     labels = c("アフリカ", "アメリカ", "アジア", "ヨーロッパ", "オセアニア"))
```



今回は大陸名が文字型の列でしたが、factor型の場合、いくつか便利な機能が使えます。たとえば、Polity\_Typeごとに国数を計算し、棒グラフを作成するとします。

```
1 Scale_df1 <- Country_df %>%
2   group_by(Polity_Type) %>%
3   summarise(N      = n(),
4             .groups = "drop")
5
6 Scale_df1
```

```
## # A tibble: 6 x 2
##   Polity_Type      N
##   <chr>          <int>
## 1 Autocracy      19
## 2 Closed Anocracy 23
## 3 Democracy      65
## 4 Full Democracy 31
```

```
## 5 Open Anocracy      20
## 6 <NA>                28
```

続きまして、Polity\_Type列をfactor型にします。最もスコアの低い独裁(Autocracy)から最もスコアの高い完全な民主主義(Full Democracy)の順番のfactorにします。

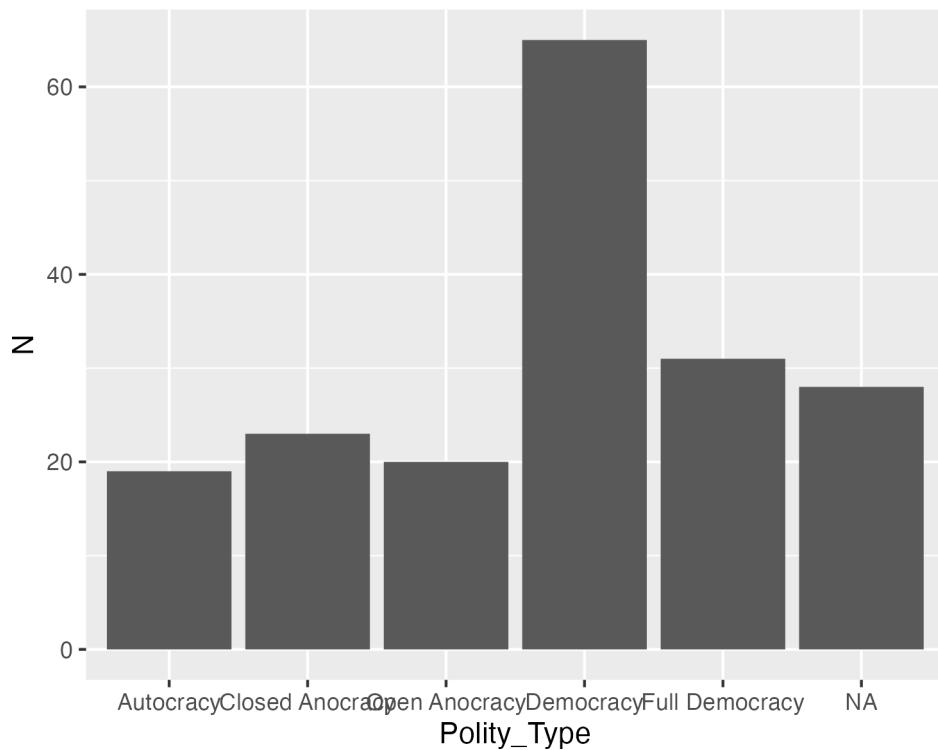
```
1 Scale_df1 <- Scale_df1 %>%
2   mutate(Polity_Type = factor(Polity_Type, ordered = TRUE,
3                               levels = c("Autocracy",
4                                         "Closed Anocracy",
5                                         "Open Anocracy",
6                                         "Democracy",
7                                         "Full Democracy")))
8
9 Scale_df1$Polity_Type
```

```
## [1] Autocracy      Closed Anocracy Democracy      Full Democracy
## [5] Open Anocracy   <NA>
## 5 Levels: Autocracy < Closed Anocracy < Open Anocracy < ... < Full Democracy
```

問題なくfactor化も出来たので、それでは作図をしてみましょう。

```
1 Scale_df1 %>%
2   ggplot() +
3   geom_bar(aes(x = Polity_Type, y = N), stat = "identity")
```

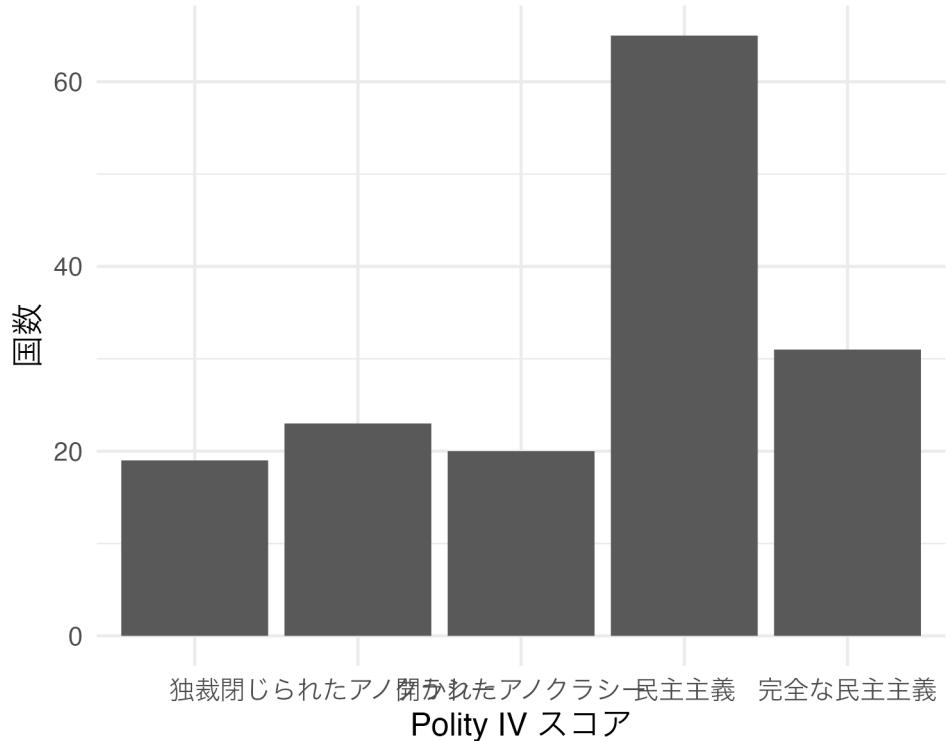


Polity プロジェクトの対象外があり、この場合は欠損値（NA）になります。そして、問題はこの欠損値も表示されることです。もちろん、欠損値のカテゴリも出力したいケースもありますが、もし欠損値カテゴリの棒を消すにはどうすれば良いでしょうか。1つ目の方法は `Scale_df1 %>% drop_na()` で欠損値を含む行を除去してから作図する方法です。2つ目の方法は `scale_x_discrete()` で `na.translate = FALSE` を指定する方法です。ここでは横軸の目盛りラベルも日本語に変更し、欠損値のカテゴリを除外してみましょう。また、地味に便利な機能として、軸ラベルも `scale_**()` で指定可能です。第1引数として長さ 1 の文字ベクトルを指定すると、自動的に軸ラベルが修正され、`labs()` が不要となります。

```
1 Scale_df1 %>%
2   ggplot() +
3   geom_bar(aes(x = Polity_Type, y = N), stat = "identity") +
4   scale_x_discrete("Polity IV スコア", # 第1引数で軸ラベルも指定可能
5                     breaks = c("Autocracy", "Closed Anocracy", "Open Anocracy",
6                               "Democracy", "Full Democracy"),
7                     labels = c("独裁", "閉じられたアノクラシー",
```

```
8      "開かれたアノクラシー", "民主主義",
9      "完全な民主主義"),
10     na.translate = FALSE) +
11   scale_y_continuous("国数") +
12   theme_minimal(base_size = 12)

## Warning: Removed 1 rows containing missing values (position_stack).
```



これで欠損値を除外することができました。目盛りラベルが重なる箇所があり、多少気になりますが、この問題に関しては第19.4節で取り上げます。

### 19.3.2 color スケールの調整

グラフの次元を増やす際に広く使われている方法は変数の値に応じて色分けをすることでした。この場合、幾何オブジェクトの `aes()` 内に `color =` マッピングする変数名を指定することになります。このように色が変数でマッピングされている場合は、色分けのスケールも調整可能です。たとえば、折れ線グラフにおいて日本が赤い線だった

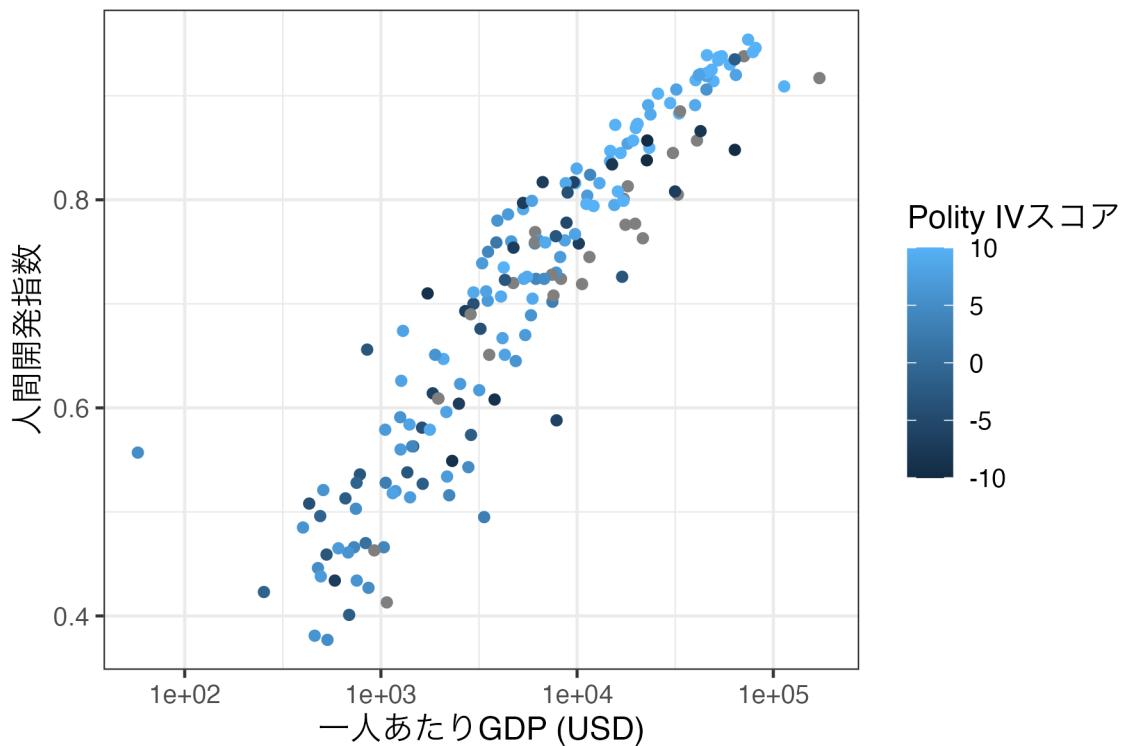
のを青い線に変更することが考えられます。ここでも `color` にマッピングされている変数のデータ型によって使用する関数が変わります。ここでは連続変数、順序付き離散変数、順序なし離散変数について説明します。

#### 19.3.2.1 `color` 引数が連続変数でマッピングされている場合

まずは、連続変数からです。横軸は底 10 の対数変換した一人あたり GDP (`GDP_per_capita`)、縦軸は人間開発指数 (`HDI_2018`) にした散布図を作成し、Polity IV スコア (`Polity_Score`) で点の色分けをしてみましょう。

```
1 Scale_Fig2 <- Country_df %>%
2   ggplot() +
3   geom_point(aes(x = GDP_per_capita, y = HDI_2018, color = Polity_Score)) +
4   labs(x = "一人あたり GDP (USD)", y = "人間開発指数", color = "Polity IV スコア") +
5   scale_x_log10() +
6   theme_bw(base_size = 12)
7
8 Scale_Fig2
```

## Warning: Removed 6 rows containing missing values (geom\_point).



これまで `color` 引数を離散変数でしかマッピングしませんでしたが、このように連続変数でマッピングすることも可能です。ただし、この場合は値に応じてはっきりした色分けがされるのではなく、グラデーションで色分けされます。この例だと、青に近いほど Polity IV スコアが高く、黒に近いほど低いことが分かります。この場合、`color` のスケール調整は最小値と最大値における色を指定するだけです。その間の色については `{ggplot2}` が自動的に計算してくれます。

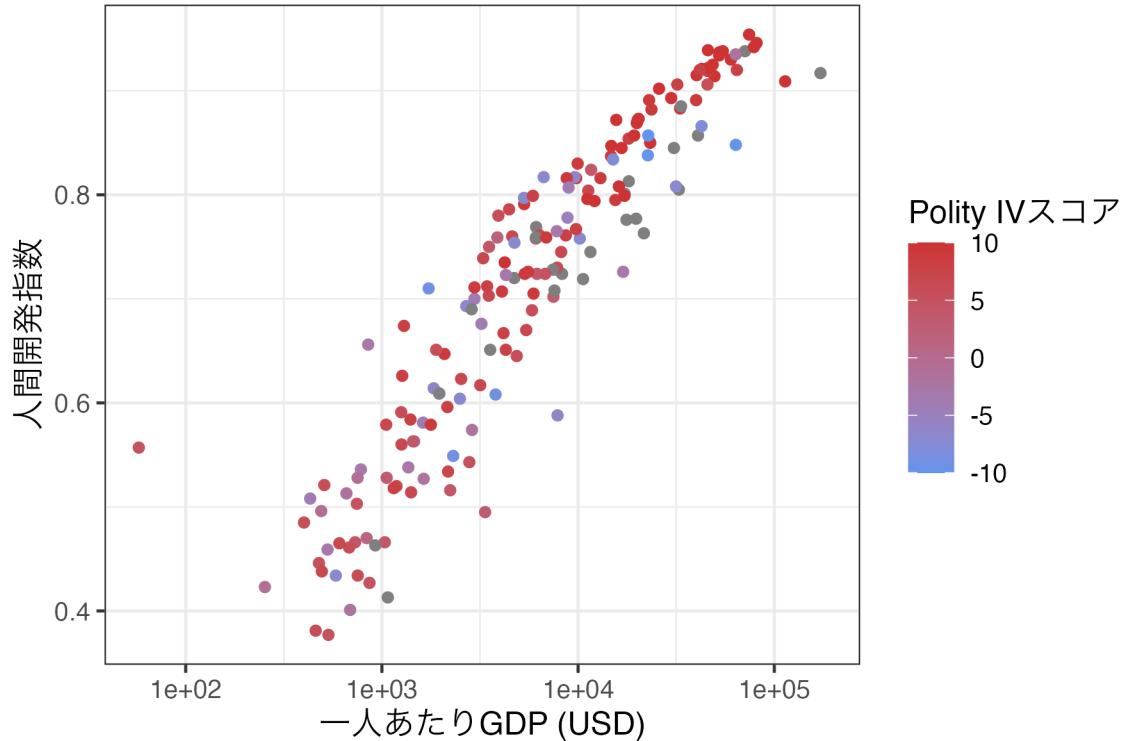
今回使用する関数は `scale_color_gradient()` です。これまでの例だと `scale_color_continuous()` かと思う方も多いでしょう。実際、`scale_color_continuous()` 関数も提供されており、使い方もほぼ同じです。ただし、`scale_color_continuous()` を使う際は引数として `type = "gradient"` を指定する必要があります。`scale_color_gradient()` の場合、最小値における色を `low`、最大値のそれを `high` で指定します。数値は "red" や "blue" なども可能であり、"#132B43" のような書き方も使えます。たとえば、先ほど `Scale_Fig2` において Polity IV スコアが高いほど brown3、低いほど cornflowerblue になるようにする場合は以下のように書きます。

```

1 Scale_Fig2 +
2   scale_color_gradient(high = "brown3", low = "cornflowerblue")

```

## Warning: Removed 6 rows containing missing values (geom\_point).



このような書き方だとどのような色名で使えるかを事前に知っておく必要があります。使える色名のリストは `colors()` から確認できます。全部で 657 種類がありますが、ここでは最初の 50 個のみを出力します。

```

1 head(colors(), 50)

## [1] "white"          "aliceblue"       "antiquewhite"    "antiquewhite1"
## [5] "antiquewhite2"  "antiquewhite3"  "antiquewhite4"  "aquamarine"
## [9] "aquamarine1"   "aquamarine2"   "aquamarine3"   "aquamarine4"
## [13] "azure"          "azure1"         "azure2"         "azure3"
## [17] "azure4"         "beige"          "bisque"         "bisque1"
## [21] "bisque2"        "bisque3"        "bisque4"        "black"
## [25] "blanchedalmond" "blue"          "blue1"          "blue2"

```

```

## [29] "blue3"          "blue4"          "blueviolet"       "brown"
## [33] "brown1"          "brown2"          "brown3"          "brown4"
## [37] "burlywood"        "burlywood1"      "burlywood2"      "burlywood3"
## [41] "burlywood4"        "cadetblue"       "cadetblue1"      "cadetblue2"
## [45] "cadetblue3"        "cadetblue4"       "chartreuse"       "chartreuse1"
## [49] "chartreuse2"       "chartreuse3"

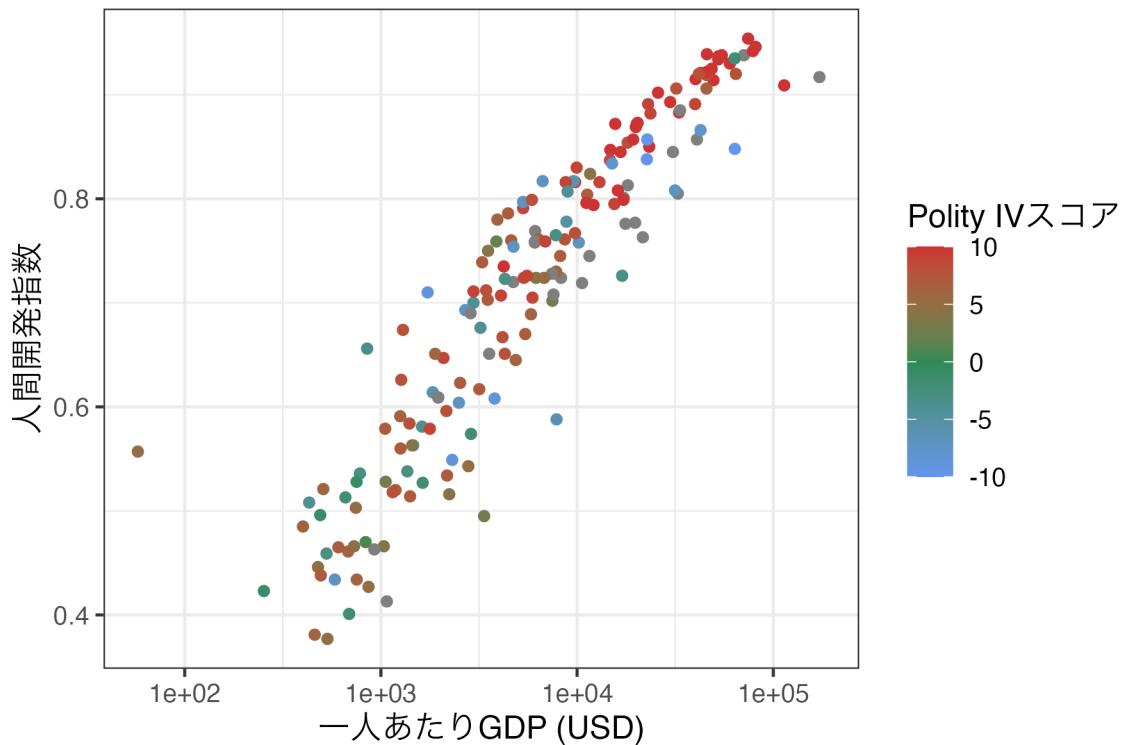
```

`scale_color_gradient()` から派生した関数として `scale_color_gradient2()` というものもあります。これは最小値と最大値だけでなく、`mid`引数を使って中間における色も指定可能な関数です。例えば、先ほどの例で真ん中を `seagreen` にしてみましょう。

```

1 Scale_Fig2 +
2   scale_color_gradient2(high = "brown3", mid = "seagreen", low = "cornflowerblue")
## Warning: Removed 6 rows containing missing values (geom_point).

```



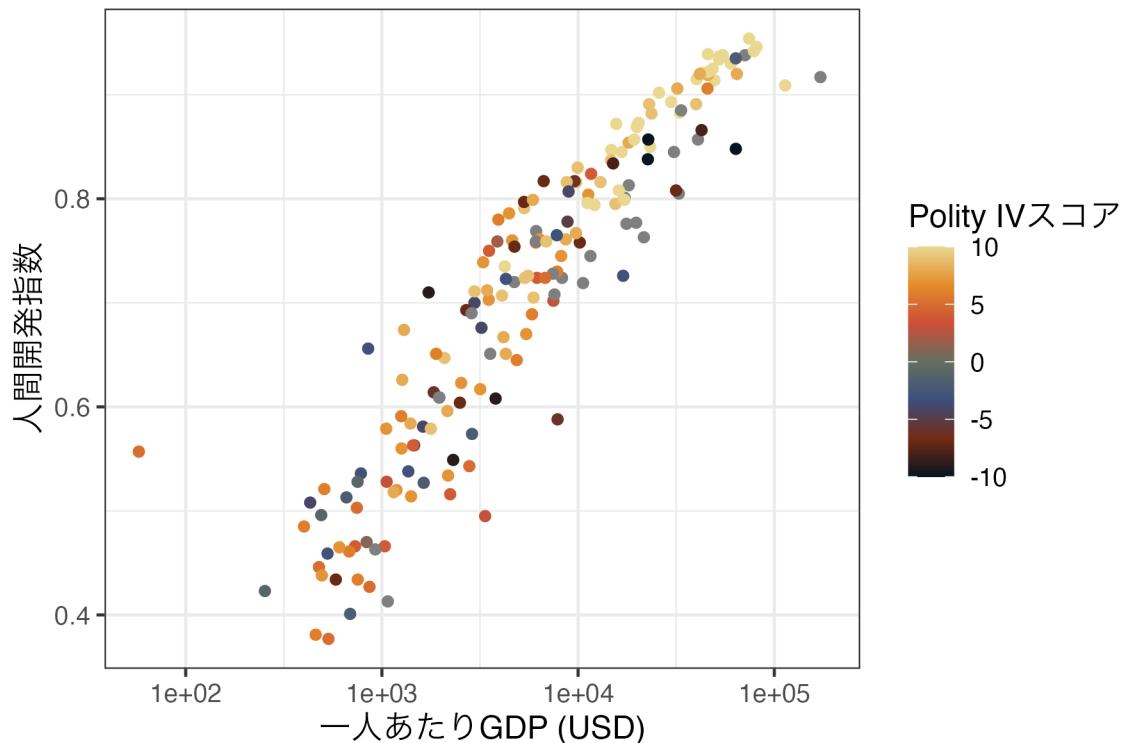
色は"`seagreen`"、"`red`"ではなく、"`#00AC97`"、"`#FF0000`"のように具体的なRGB値で指定することもできます。これは色を赤(R)、緑(G)、青(B)の3つの原色を混ぜ

て様々な色を表現する方法です。"#FF0000"の場合、最初の#はRGB表記であることを意味し、FFは赤が255であることの16進法表記、次の00と最後の00は緑と青が0であることの16進法表記です。各原色は0から255までの値を取ります。`{ggplot2}`でよく見る色としては#F8766D、#00BFC4、#C77CFF、#7CAE00があります。他にもGoogleなどで「RGB color list」などを検索すれば様々な色を見ることができます。

他にも`{ggplot2}`は様々なユーザー指定のパレットが使用可能であり、実際、パッケージの形式として提供される場合もあります。たとえば、ジブリ風のカラーパレットが`{ghibli}`パッケージとして提供されており、`install.packages("ghibli")`でインストール可能です。

```
1 pacman::p_load(ghibli)
2
3 Scale_Fig2 +
4   # 連続変数 (_c) 用の「もののけ姫」パレット (medium)
5   scale_color_ghibli_c("MononokeMedium")
```

```
## Warning: Removed 6 rows containing missing values (geom_point).
```

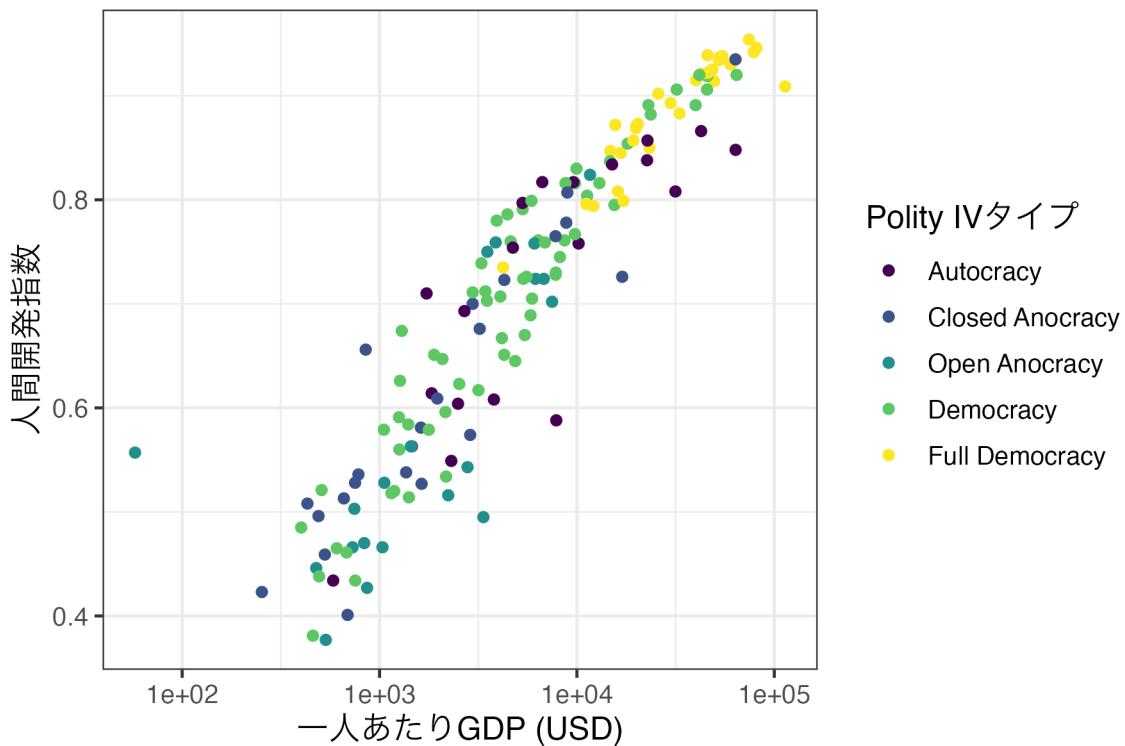


### 19.3.2.2 color引数が順序付き離散変数でマッピングされている場合

連続変数のようにグラデーションではあるものの、それぞれの値に対して具体的な色が指定されます。まずは先ほど作成しました Scale\_Fig2 の色分けを連続変数である Polity\_Score でなく、順序付き離散変数である Polity\_Type にします。Polity\_Type の順序は独裁 (Autocracy)、閉じられたアノクラシー (Closed Anocracy)、開かれたアノクラシー (Open Anocracy)、民主主義 (Democracy)、完全な民主主義 (Full Democracy) にします。また、Polity\_Type が定義されていない国もデータセットに含まれているため、drop\_na(Polity\_Type) を追加し、Polity\_Type が欠損している行を除去します。

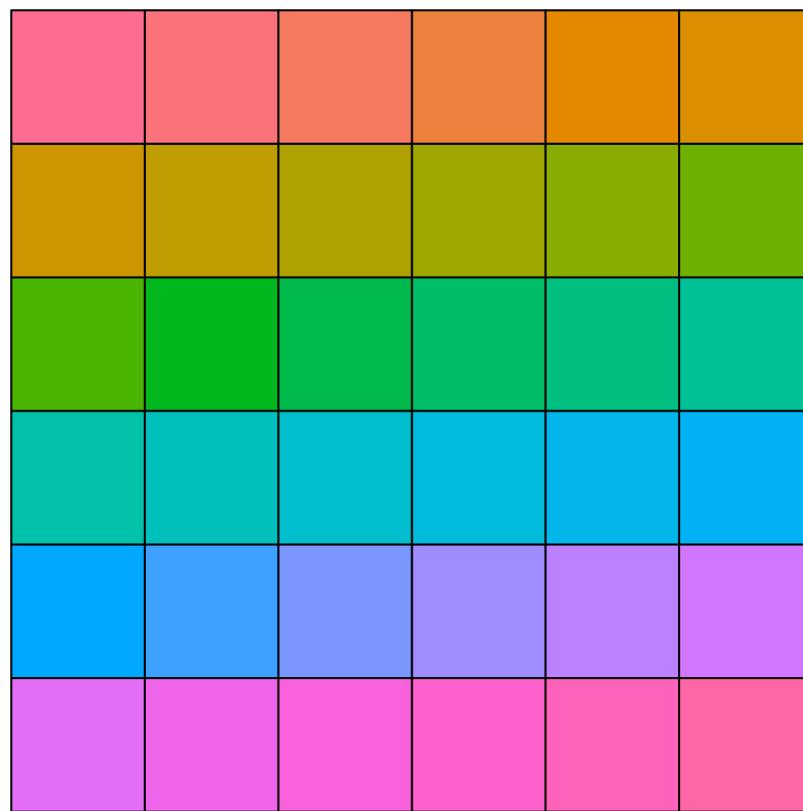
```
1 Scale_Fig3 <- Country_df %>%
2   mutate(Polity_Type = factor(Polity_Type, ordered = TRUE,
3                               levels = c("Autocracy",
4                                         "Closed Anocracy",
5                                         "Open Anocracy",
6                                         "Democracy",
7                                         "Full Democracy"))) %>%
8   drop_na(Polity_Type) %>%
9   ggplot() +
10  geom_point(aes(x = GDP_per_capita, y = HDI_2018, color = Polity_Type)) +
11  labs(x = "一人あたり GDP (USD)", y = "人間開発指数", color = "Polity IV タイプ") +
12  scale_x_log10() +
13  theme_bw(base_size = 12)
14
15 Scale_Fig3
```

## Warning: Removed 2 rows containing missing values (geom\_point).



今回は紫（独裁）から黄色（完全な民主主義）の順で色分けがされ、その間のカテゴリーも紫と黄色の間の値をとります。実は順序付き離散変数の場合、色のスケールを調整することはありませんし、これまでの方法に比べてやや複雑です。ここでは色相 (Hue) の範囲と強度、明るさを調整する方法について紹介します。

まずは、色相について知る必要があります。{ggplot2}において色相の範囲は 0 から 360 です。そして、色には強度 (intensity)、または彩度という概念があり、0 に近いほどグレイへ近づき、色間の区別がしにくくなります。{ggplot2}では彩度のデフォルト値は 100 であり、我々が普段{ggplot2}で見る図の色です。最後に明るさ (luminance) があり、0 から 100 までの値を取ります。値が大きいほど明るくなり、{ggplot2}のデフォルト値は 65 です。重要なのは色相のところであり、Hue の具体的な数値がどの色なのかを確認する必要があります。そのためには、{scales}パッケージの `hue_pal()` と `show_col()` 関数を使用します。



左上が 0、右下が 360 の色を意味します。順序変数でマッピングされた色スケールを色相に基づいて調整する際は、`scale_color_hue()` レイヤーを追加し、色相の範囲、彩度、明るさを指定します。たとえば、0 から 300 までの範囲の色を使用し<sup>2)</sup>、再度と明るさはデフォルトにしたい場合、以下のように書きます。

```

1 Scale_Fig3 +
2   scale_color_hue(h = c(0, 360), c = 100, l = 65)

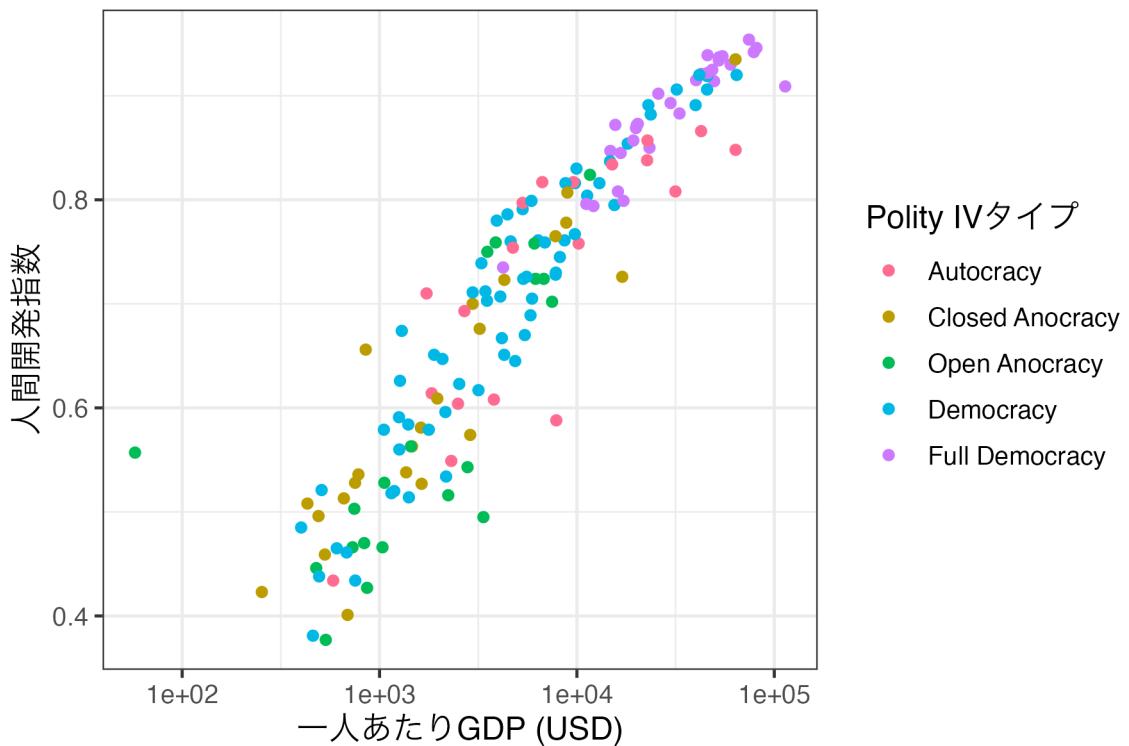
```

```

## Warning: Removed 2 rows containing missing values (geom_point).

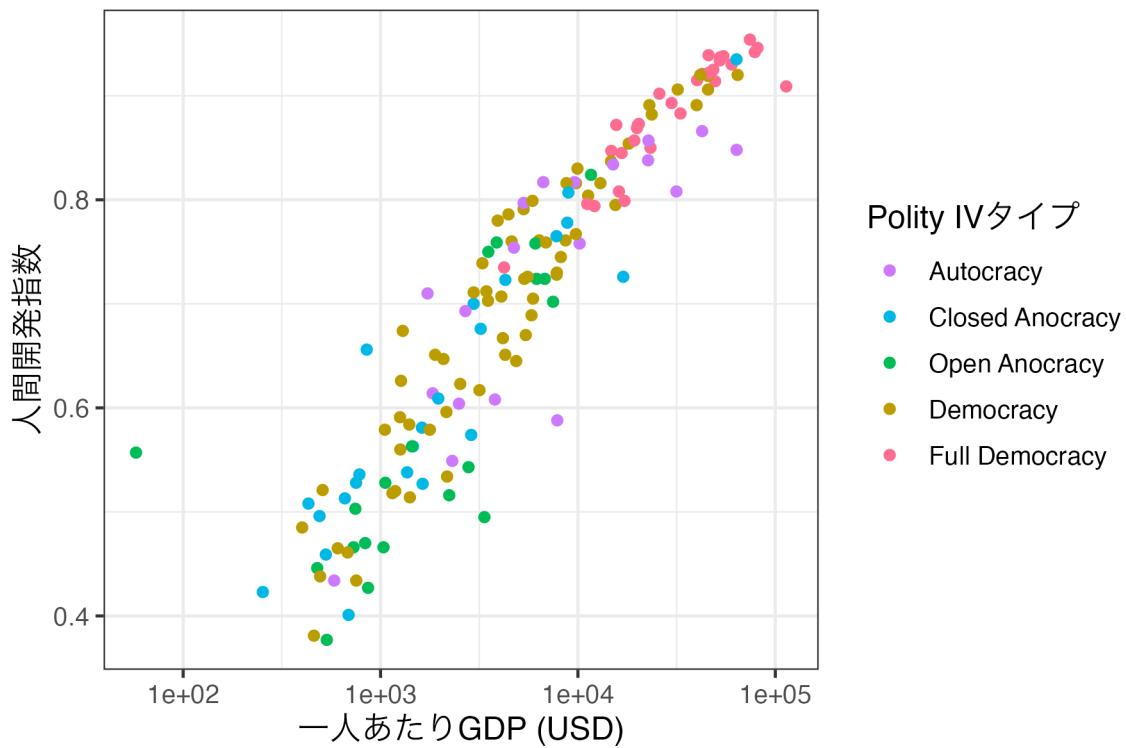
```

<sup>2)</sup> 厳密には 0 から 360 までの全範囲の色を使うわけではありません。なぜなら、Hue は 0 度から 360 度、つまり循環する仕組みになっているからです。実際、Hue が [0, 360] の図を見ても、左上の色と右下の色は非常に似ていることが分かります。したがって、0 から 360 を全て使用すると色の区別が難しくなるため、この場合は後半の色は仕様されません。



また、`direction = -1`を追加することで、色の順番を逆にすることも可能です。

```
1 Scale_Fig3 +  
2   scale_color_hue(h = c(0, 360), c = 100, l = 65, direction = -1)  
  
## Warning: Removed 2 rows containing missing values (geom_point).
```



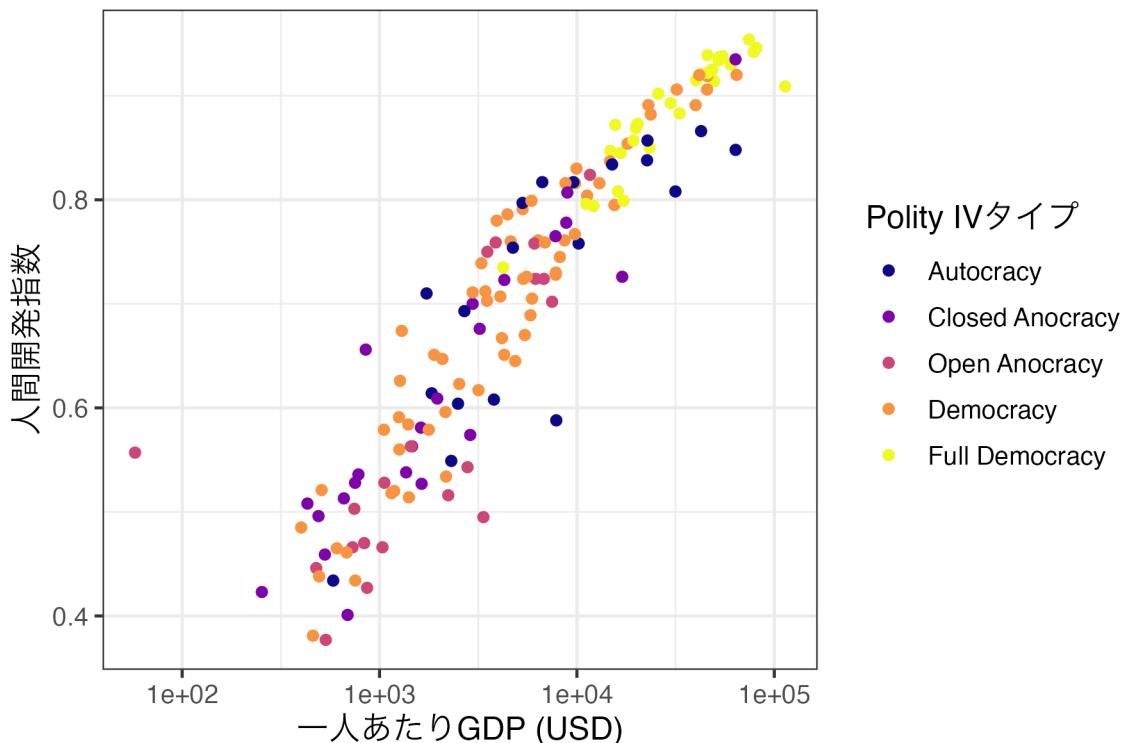
他にも `mpl colormaps` というカラーマップを使うことも可能ですが。この場合は `scale_color_hue()` ではなく、`scale_color_viridis_d()` を使用します。具体的な色の情報は `mpl colormaps` を参照してください。必要な引数は色のスタート地点 (`begin`) と終了地点 (`end`) です。そして、`option` の引数のデフォルト値は "D" であり、これは `VIRIDIS colormap` を意味します。実は `scale_color_*`() を付けなかった場合の色分けがこれです。たとえば、`VIRIDIS colormap` でなく、`PLASMA colormap` を使うなら `option = "C"` を付けます。

```

1 Scale_Fig3 +
2   # PLASMA colormap を使用する (option = "C")
3   scale_color_viridis_d(begin = 0, end = 1, option = "C")

```

```
## Warning: Removed 2 rows containing missing values (geom_point).
```



### 19.3.2.3 color引数が順序なし離散変数でマッピングされている場合

最後に順序なし離散変数の場合について解説します。ここでは `scale_color_manual()` 関数を使用し、マッピングされている変数のそれぞれ値に対して具体的な色を指定する方法です。以下で説明する方法は、順序付き離散変数でも使用可能であるため、Scale\_Fig3 の図をそのまま利用してみたいと思います。

`scale_color_manual()` の核心となる引数は `values` であり、ここに"変数の値" = "色"で指定します。この色は"red"のような具体的な色名でも、"#FF0000"のようなRGB表記でも構いません。

```

1 ggplot2オブジェクト +
2   scale_color_manual(values = c("変数の値1" = "色1",
3                             "変数の値2" = "色2",
4                             "変数の値3" = "色3",
5                             ...))

```

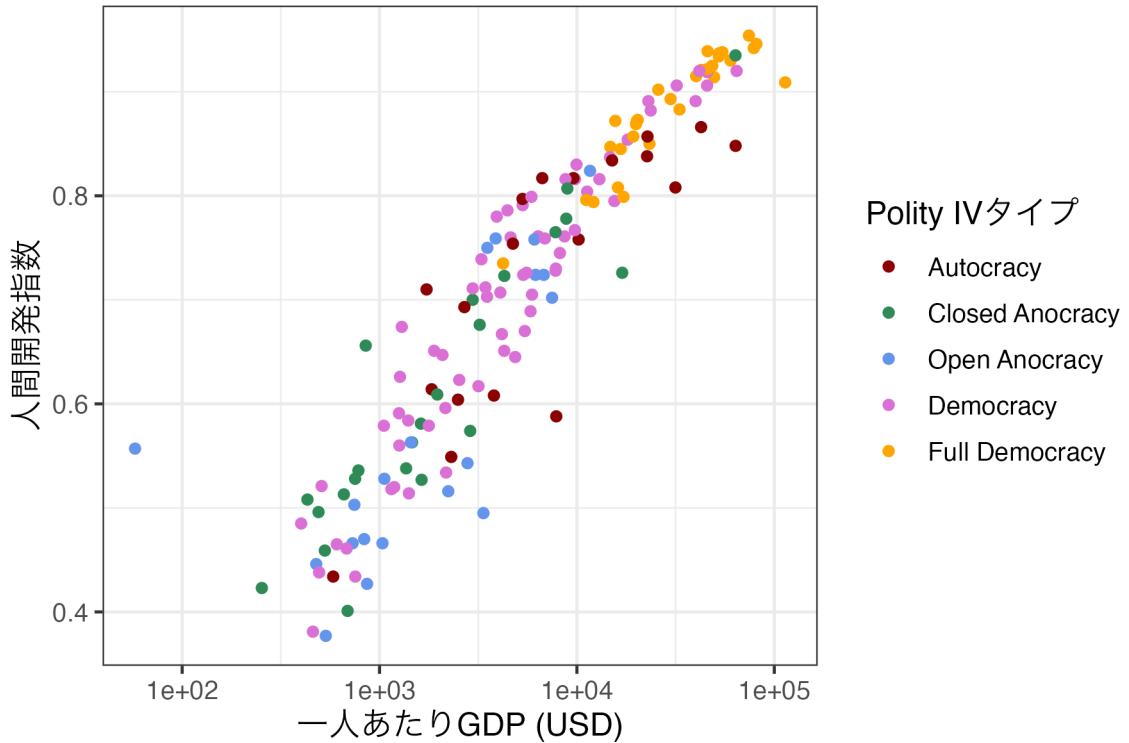
それでは `Polity_Type` の値が"Autocracy"の場合は `darkred`、"Closed Anocracy"

は seagreen、"Open Anocracy"は cornflowerblue、"Democracy"は orchid、"Full Democracy"は orangeにしてみましょう。

```

1 Scale_Fig3 +
2   scale_color_manual(values = c("Autocracy"      = "darkred",
3                               "Closed Anocracy" = "seagreen",
4                               "Open Anocracy"   = "cornflowerblue",
5                               "Democracy"       = "orchid",
6                               "Full Democracy"  = "orange"))
7
8 ## Warning: Removed 2 rows containing missing values (geom_point).

```



色を決める際は色覚多様性に気をつけるべきです。誰にとっても見やすい図には Universal Design が必要です。特によくある例が「緑と赤」の組み合わせです。緑も赤も暖色系であり、P型およびD型色弱の場合、両者の区別が難しいと言われています。しかも、色弱の方は意外と多いです。日本の場合、男性の5%、女性の0.2%と言われております。これには人種差もありますし、フランスや北欧の男性の場合は約10%です。一通り図を作成しましたら、色覚シミュレーターなどを使用して、誰にとっても見やすい色

であるかを確認することも良いでしょう。また、`{ggplot2}`がデフォルトで採用しているVIRIDISは色弱に優しいカラーパレットと言われています。一般的に二色の組み合わせの場合、最も区別しやすい色は青とオレンジと言われています。

`{ggplot2}`における `color` スケールは非常に細かく調整可能であり、関連関数やパッケージも多く提供されています。本書では全てを紹介することはできませんが、ここまで紹介してきました例だけでも、自分好みに色を調整できるでしょう。

### 19.3.3 `alpha` スケールの調整

次は点・線・面の透明度を指定する `alpha` スケールの調整です。もし、`alpha` に連続変数をマッピングすれば、マッピングした変数の値に応じて透明度が変わります。一般的に、値が大きいほど不透明となり、小さいほど透明になります。しかし、このような使い方はあまり見られません。プロット上のオブジェクトを透明度を指定するのは

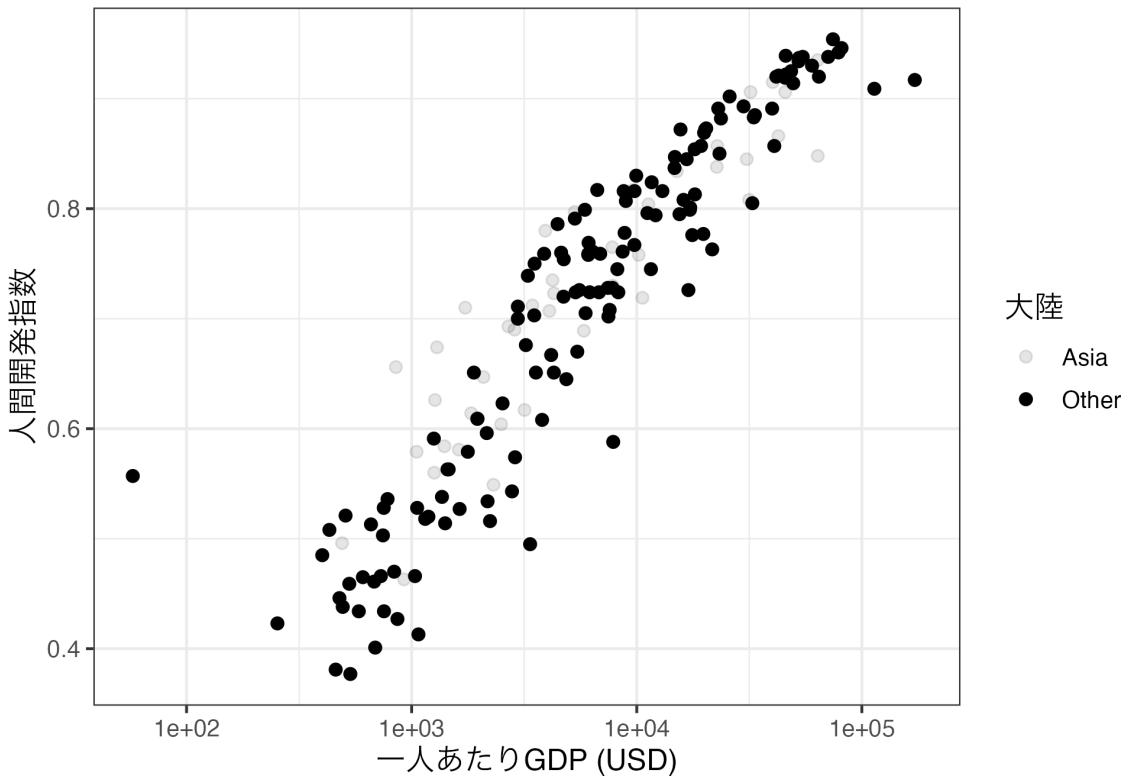
1. 特定箇所にオブジェクトが密集し、重なるオブジェクトの区別がつきにくい場合
2. 特定の点・線・面を強調するため

以上の2ケースでしょう。また、ケース1の場合、`alpha`引数を `aes()` の内部ではなく、外側に指定し、具体的な数値を指定することになります。ここで注目するのはケース2です。たとえば、横軸に一人あたりGDP、縦軸に人間開発指数をマッピングした散布図を作成し、アジアの国のみハイライトしたいとします。この場合、アジアとその他の国で色分けしたり、丸と四角といった形で分けるのも可能ですが、「強調」が目的であれば、その他の国をやや透明にすることも可能でしょう。まず、`Asia`という変数を作成し、`Continent`の値が"Asia"なら"Asia"、その他の値なら"Other"の値を入れます。そして、`geom_point()`の `aes()` 内に `alpha`引数を追加し、`Asia`変数でマッピングします。

```
1 Country_df %>%
2   mutate(Asia = if_else(Continent == "Asia", "Asia", "Other")) %>%
3   ggplot() +
4   geom_point(aes(x = GDP_per_capita, y = HDI_2018, alpha = Asia), size = 2) +
5   labs(x = "一人あたり GDP (USD)", y = "人間開発指数", alpha = "大陸") +
6   scale_x_log10() +
7   theme_bw(base_size = 12)
```

```
## Warning: Using alpha for a discrete variable is not advised.

## Warning: Removed 6 rows containing missing values (geom_point).
```

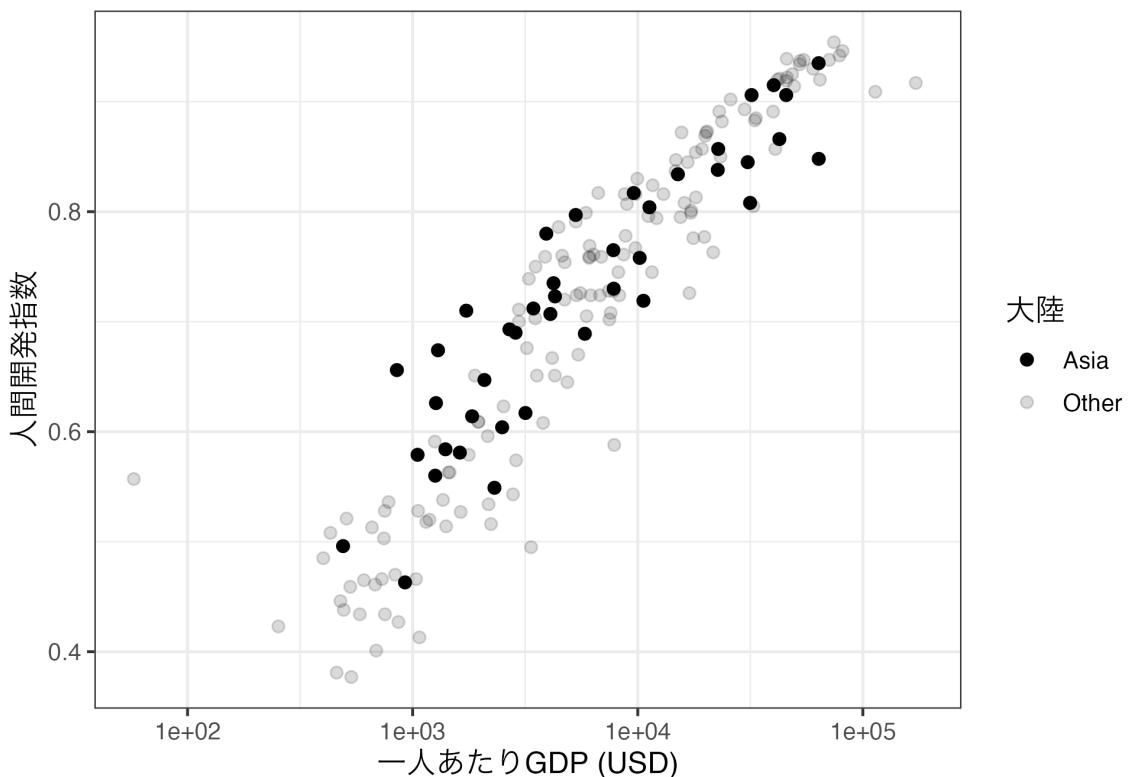


Asia の値によって透明度が異なりますが、私たちの目的は Asia を不透明にし、その他の点を透明にすることです。ここで登場するのが `scale_alpha_manual()` です。使い方はこれまで見てきた `scale_*_manual()` と非常に似ています。`values` 引数にそれぞれの値と透明度を指定するだけです。透明度は 1 が不透明、0 が透明です。アジアの透明度を 1.0、その他の透明度を 0.15 とするなら、以下のように書きます。

```
1 Country_df %>%
2   mutate(Asia = if_else(Continent == "Asia", "Asia", "Other")) %>%
3   ggplot() +
4   geom_point(aes(x = GDP_per_capita, y = HDI_2018, alpha = Asia), size = 2) +
5   labs(x = "一人あたり GDP (USD)", y = "人間開発指数", alpha = "大陸") +
6   scale_x_log10()
```

```
7  scale_alpha_manual(values = c("Asia" = 1.0, "Other" = 0.15)) +
8  theme_bw(base_size = 12)
```

```
## Warning: Removed 6 rows containing missing values (geom_point).
```



これでアジアの国々がプロット上で強調されました。透明度スケールは連続変数 (`scale_alpha_continuous()`) や離散変数 (`scale_alpha_discrete()`) に使うことも可能ですが、透明度を「区別」でなく「強調」の目的で使うならば、`scale_alpha_manual()` でも十分だと考えられます。

一つ注意して頂きたいのは、図を PDF で出力する際、半透明なオブジェクトが見えなかったり、逆に透明度が全く適用されない場合があります。PNG などのビットマップ画像ならこのような問題は生じませんが、PDF の場合は注意が必要です<sup>3)</sup>。この場合、画

<sup>3)</sup> 宋が論文を出した際、半透明なオブジェクトが査読者の PC では出力されなかったことがあり、「お前、一体図の何を見ろと言っているんだ」と怒られたことがあります。

像をPNGなどの形式にするか、半透明でなく「グレーと黒」のような組み合わせで作図した方が良いかも知れません。

#### 19.3.4 sizeスケールの調整

大きさに関するマッピングは幾何オブジェクトの `aes()` 内に `size` 引数を使用します。ここでの大さいというのは点の大きさと線の太さを意味します。`geom_point()` 内の `size` は点の大きさ、`geom_line()` や `geom_segment()` の `size` は線の太さ、`geom_pointrange()` では両方を意味します。ただし、実質的に変数の値に応じて線の太さを変えることは強調<sup>4)</sup>を除けばあまり見られません。ここでは点の大きさに焦点を当てて解説します。

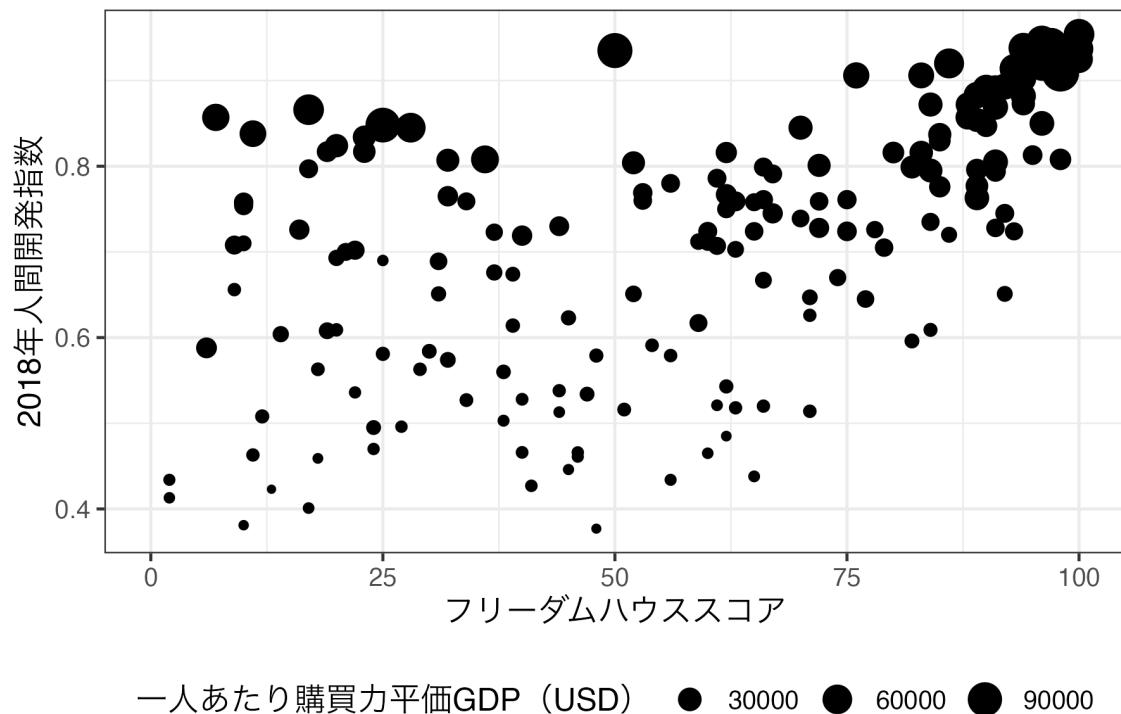
それでは、横軸はフリーダム・ハウスのスコア (`FH_Total`)、縦軸は2018年人間開発指数 (`HDI_2018`) として散布図を作成し、一人当たり購買力平価GDP (`PPP_per_capita`) に応じて点の大きさを指定します。

```
1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = HDI_2018, size = PPP_per_capita)) +
4   labs(x = "フリーダムハウススコア", y = "2018年人間開発指数",
5        size = "一人あたり購買力平価GDP(USD)") +
6   theme_bw(base_size = 12) +
7   theme(legend.position = "bottom") # 凡例の図の下段にする

## Warning: Removed 11 rows containing missing values (geom_point).
```

---

<sup>4)</sup> 強調したい線を太線にするなど



フリーダム・ハウススコアと人間開発指数は全般的には正の相関を示しています。そして、両指標が高い国（図の右上）は所得水準も高いことが分かります。ただし、フリーダム・ハウススコアが低くても人間開発指数が高い国（図の左上）もかなり見られますが、これらの国の共通点は所得水準が高いことです。つまり、人間開発指数と所得水準には強い相関関係があると考えられます（人間開発指数には所得水準も含まれるため、当たり前です）。

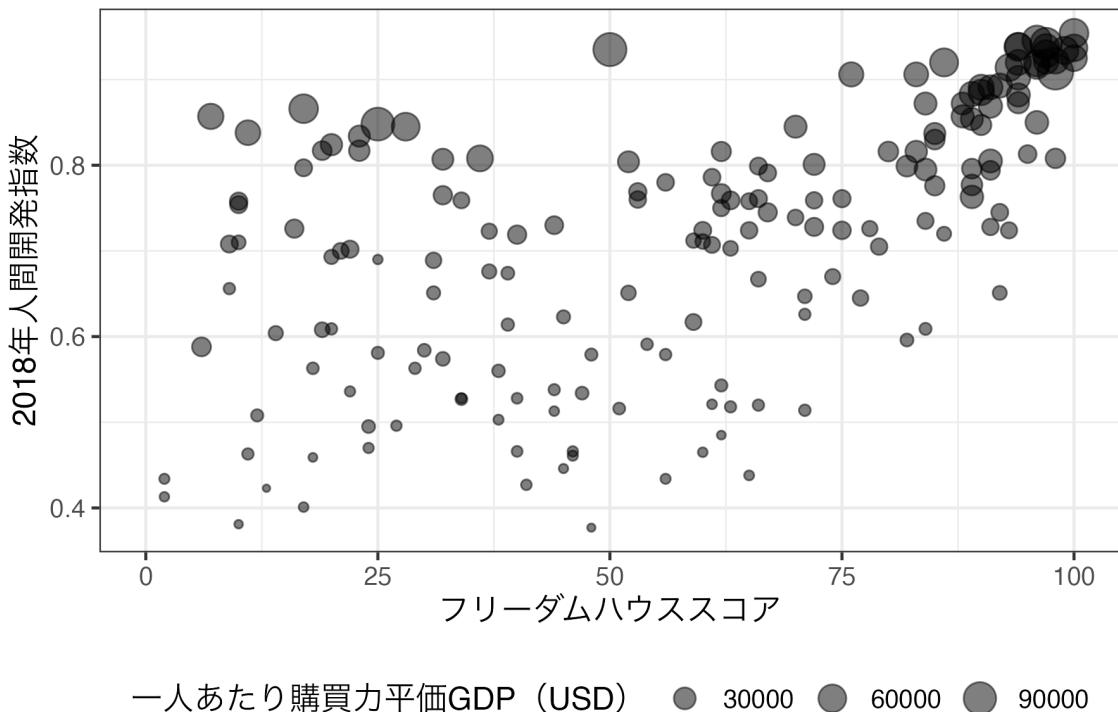
大きさをマッピングすると、点が重なる箇所が広くなりますので、alpha引数で半透明にすることも有効でしょう。

```

1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = HDI_2018, size = PPP_per_capita),
4             alpha = 0.5) + # 透明度の指定
5   labs(x = "フリーダムハウススコア", y = "2018 年人間開発指数",
6         size = "一人あたり購買力平価 GDP (USD)") +
7   theme_bw(base_size = 12) +
8   theme(legend.position = "bottom")

```

```
## Warning: Removed 11 rows containing missing values (geom_point).
```



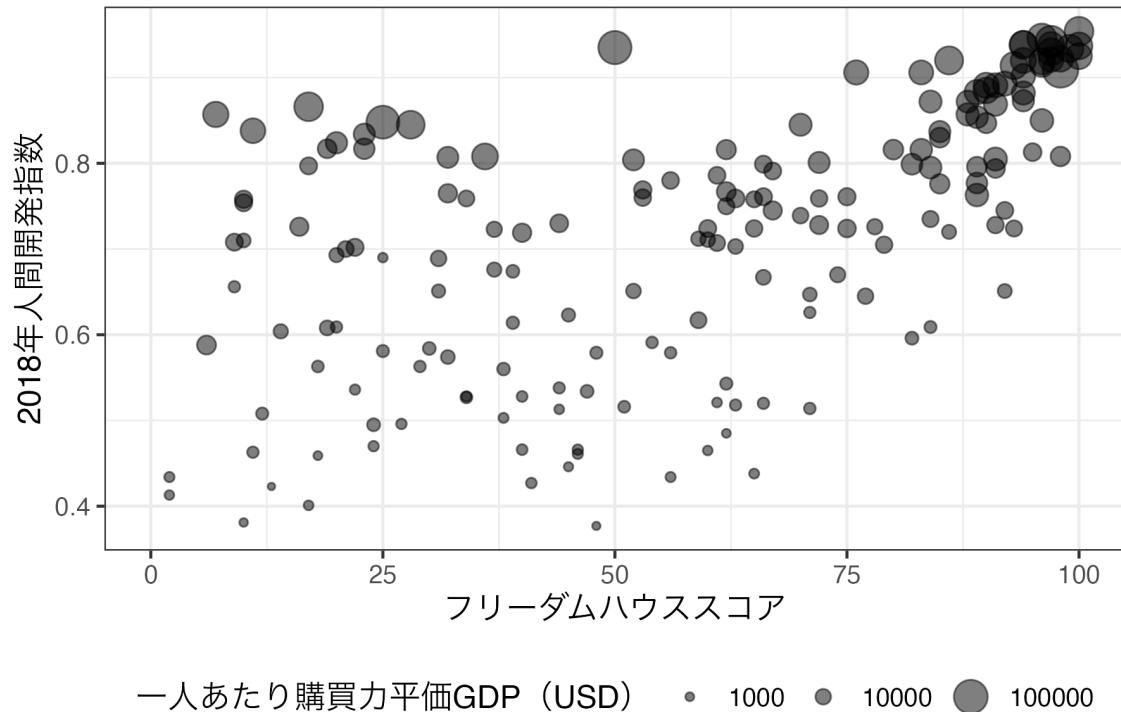
この大きさのスケール調整はマッピングされた変数の尺度によって、`scale_size_continuous()`、`scale_size_discrete()`、`scale_size_ordinal()`などを使用し、すべてマニュアルで調整したい場合は`scale_size_manual()`を使います。使い方はこれまでのスケール調整とほぼ同様です。たとえば、上記の図だと、大きさの凡例が3万、6万、9万となっていますが、これを1000、10000、100000といった対数スケールに変更してみましょう。

```
1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = HDI_2018, size = PPP_per_capita),
4             alpha = 0.5) +
5   labs(x = "フリーダムハウススコア", y = "2018年人間開発指数",
6        size = "一人あたり購買力平価GDP(USD)") +
7   scale_size_continuous(breaks = c(1000, 10000, 100000),
8                         labels = c("1000", "10000", "100000")) +
9   theme_bw(base_size = 12) +
```

```

10 theme(legend.position = "bottom")
## Warning: Removed 11 rows containing missing values (geom_point).

```



この場合、図内の点の大きさに変化はありません。変わるのは凡例のみであり、値に応じた点のサイズに調整されます。連続変数でマッピングされている場合、一つ一つの値に応じてサイズを指定するのは非現実的であります。この場合、点の大きさ調整は{ggplot2}に任せて、凡例のサイズを調整するのが無難でしょう。

ただし、点の最小サイズと最大サイズを調整したいケースもあるでしょう。最も小さい点のサイズを 0.1 に、最も大きい点のサイズを 10 にする場合、range 引数を指定します。range 引数の実引数は長さ 2 の数値型ベクトルです。

```

1 Country_df %>%
2   ggplot() +
3   geom_point(aes(x = FH_Total, y = HDI_2018, size = PPP_per_capita),
4               alpha = 0.5) +

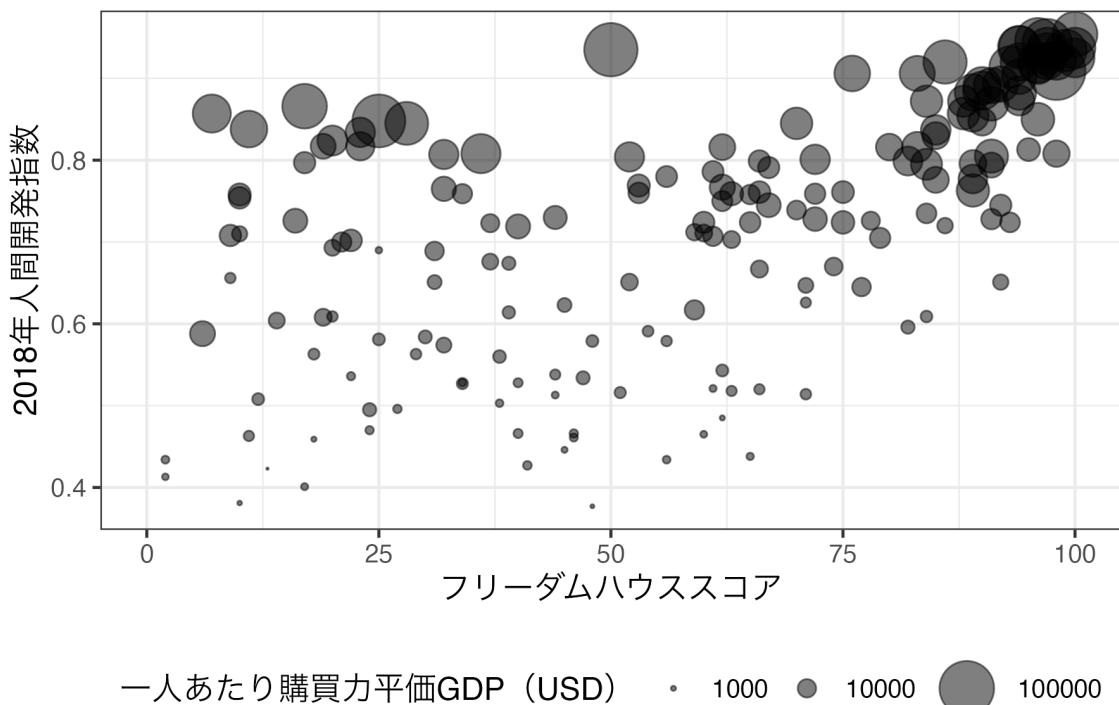
```

```

5   labs(x = "フリーダムハウススコア", y = "2018年人間開発指数",
6     size = "一人あたり購買力平価GDP(USD)") +
7     scale_size_continuous(breaks = c(1000, 10000, 100000),
8                           labels = c("1000", "10000", "100000"),
9                           range = c(0.1, 10)) +
10    theme_bw(base_size = 12) +
11    theme(legend.position = "bottom")

```

## Warning: Removed 11 rows containing missing values (geom\_point).



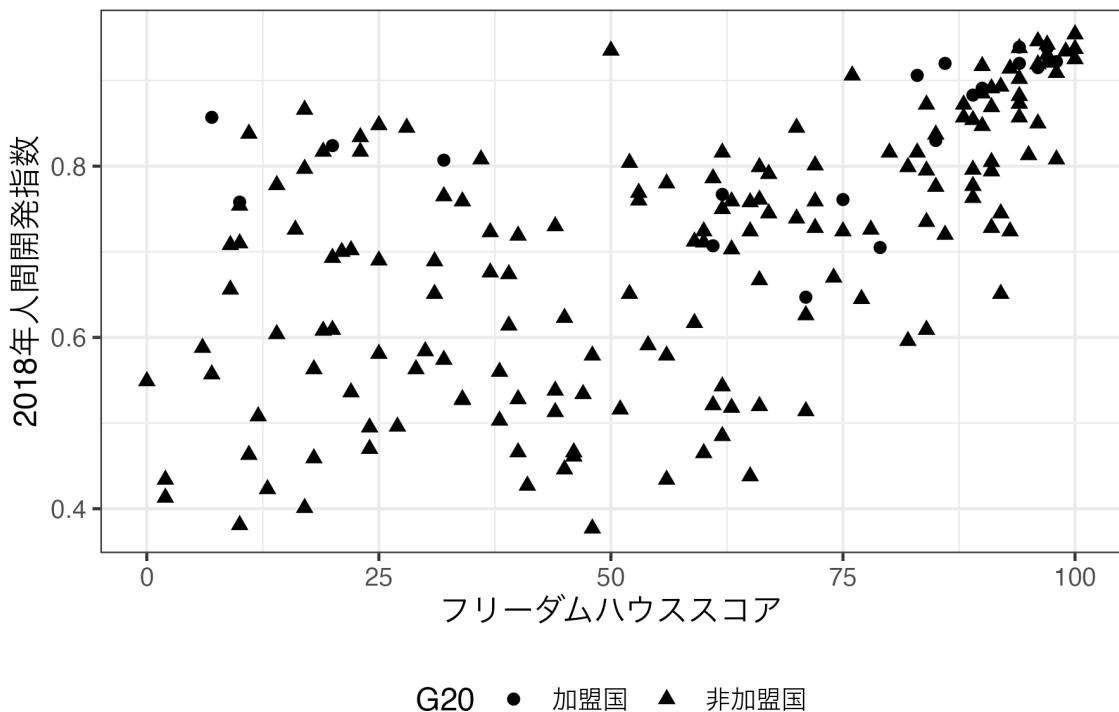
このように範囲が広がるほど、所得水準の差がより見やすくなります。他にも size は離散変数でマッピングされることも可能ですが、あまり相性は良くありません。離散変数でのマッピングはこれまで紹介しました color や size の方を参照してください。使い方は同じです。

### 19.3.5 `shape`、`linetype` スケールの調整

`size` は連続変数と相性が良いですが、点や線の形である `shape` と `linetype` は離散変数、とりわけ順序なし離散変数（名目変数）と相性が良いです。なぜなら、点や線の形は高低・大小の情報を持たないからです。たとえば、丸の点は四角の点より大きいとか実線は破線より小さいといった情報はありません。したがって、`size` や `linetype` は名目変数に使うのが一般的です。たとえば、フリーダムハウスのスコアを横軸、人間開発指数を縦軸とし、G20 加盟有無によって点の形が異なる散布図を作成するとします。G20 を `character` 型、あるいは `factor` 型に変換し、`geom_point()` の幾何オブジェクトの `aes()` 内に `shape` 引数を指定します。

```
1 Country_df %>%
2   mutate(G20 = if_else(G20 == 1, "加盟国", "非加盟国")) %>%
3   ggplot() +
4   geom_point(aes(x = FH_Total, y = HDI_2018, shape = G20),
5             size = 2) +
6   labs(x = "フリーダムハウススコア", y = "2018 年人間開発指数",
7         shape = "G20") +
8   theme_bw(base_size = 12) +
9   theme(legend.position = "bottom")
```

## Warning: Removed 6 rows containing missing values (geom\_point).



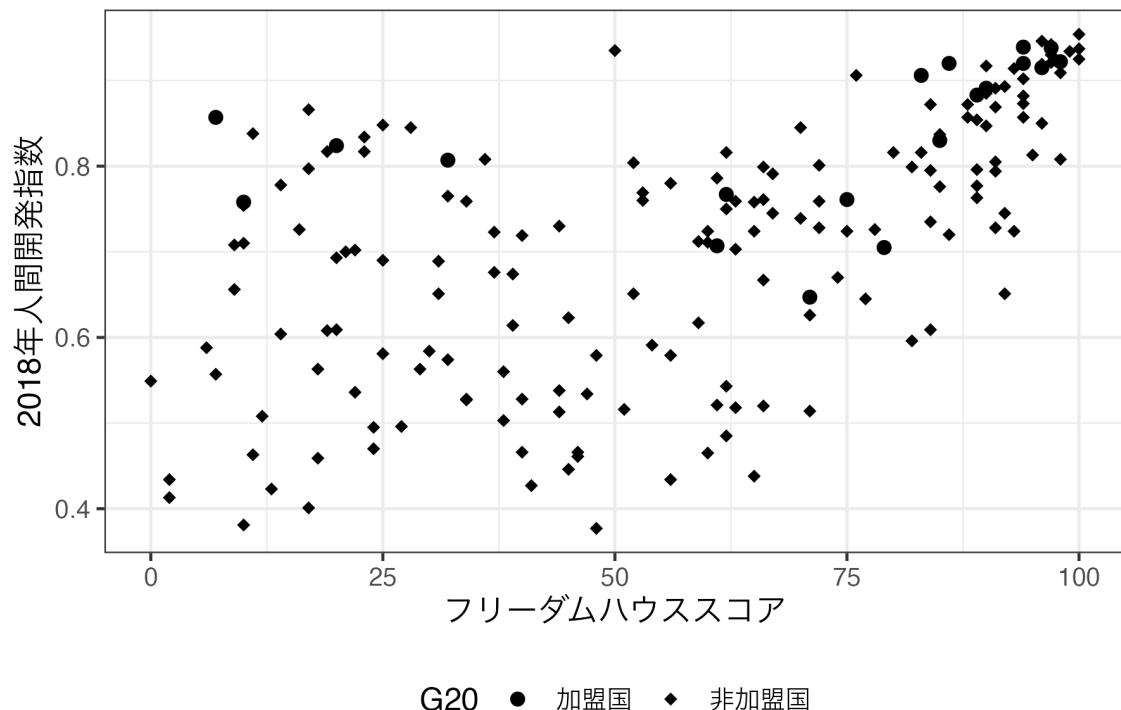
加盟国を丸 (16 または 19)、非加盟国をダイヤモンド型 (18) にするには `scale_shape_manual()` を使います。

```

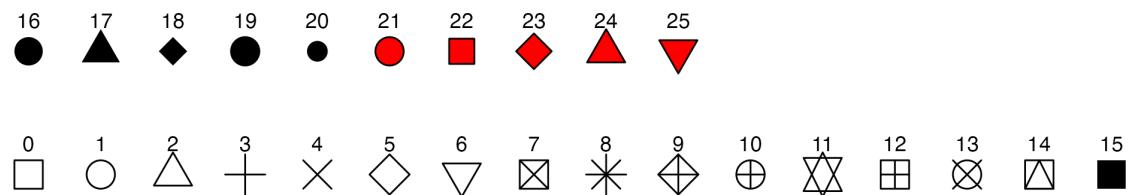
1 Country_df %>%
2   mutate(G20 = if_else(G20 == 1, "加盟国", "非加盟国")) %>%
3   ggplot() +
4   geom_point(aes(x = FH_Total, y = HDI_2018, shape = G20),
5             size = 2) +
6   scale_shape_manual(values = c("加盟国" = 19, "非加盟国" = 18)) +
7   labs(x = "フリーダムハウススコア", y = "2018年人間開発指数",
8        shape = "G20") +
9   theme_bw(base_size = 12) +
10  theme(legend.position = "bottom")

```

## Warning: Removed 6 rows containing missing values (geom\_point).

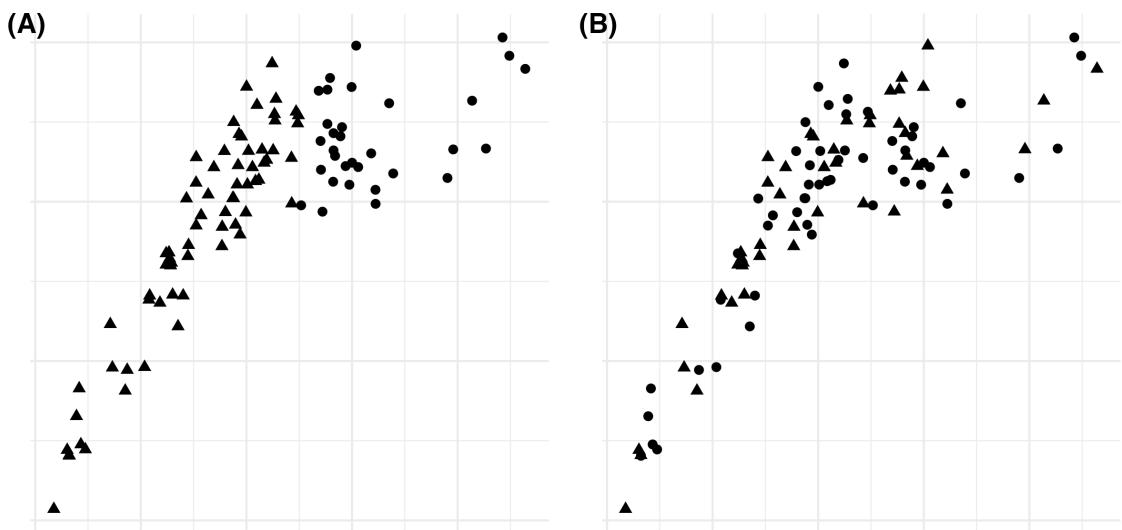


ただし、各数字がどの形に対応しているかを事前に知っておく必要があります。よく使うのは 0 から 25 までであり、それぞれ対応する shape を示したのが以下の図です。必要に応じてこのページを参照しても良いですし、R コンソール上で?pch を入力しても 0 から 23 までの例を見ることが出来ます。

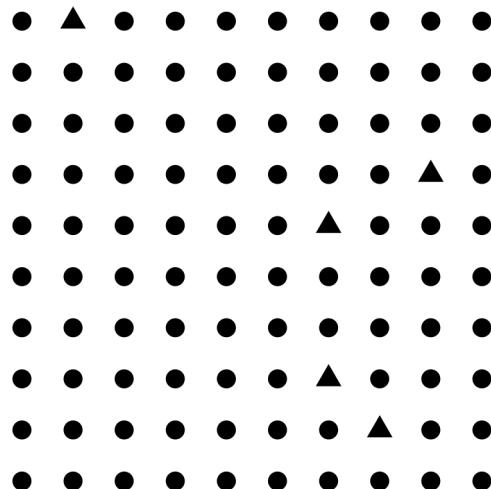


注意していただきたいのは、shape のマッピングが有効でない状況があるという点です。それが先ほどの例です。先ほどの散布図の場合、次元を増やすことは、新しい次元で条件づけた場合の変数の関係性を調べることとなります。しかし、新しい次元による条件付き関連性があまり見られない場合、あるいは二種類以上の点の形があまり分離されていない場合は、次元の追加がもたらす恩恵が感じにくくなるでしょう。例えば以下のような散布図を比較してみましょう。図 (A) の場合、縦軸の変数が閾値を超えるともう一つの変数との関係が弱まるということが分かります。たとえば、三角の点において両変数は

正の相関を持ち、丸の点においては無相間に近いことが分かります。この場合、点の形は非常に有用な情報を含んでいると判断できます。一方、図(B)の場合、点の形から読み取れる情報が少ないですね。あえて言えば、グループ間の違いがあまりないことくらいでしょう。

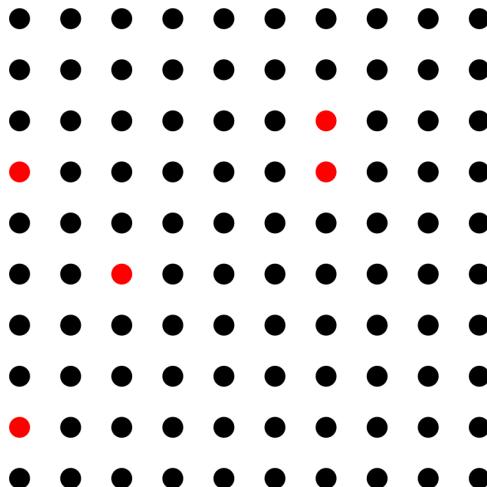


白黒のグラフの場合、色分けが出来ないため、次元拡張には点の形を変えることになります。しかし、色に来れば形は読み手にとって認知の負荷がかかりやすいです。下の図を見てください。100個の点がありますが、三角の点はいくつでしょうか。



正解は5つです。あまり難しい問題ではないでしょう。一方、下の図はいかがでしょ

うか。



どれも正解は 5 つです。本質的には同じ問題ですが、どの図の方が読みやすかったでしょうか。個人差はあるかも知れませんが、多くの方にとって後者の方が読みやすかったでしょう。最近、海外のジャーナルはカラーの図を使うことも可能ですので、色分けを優先的に考えましょう。白黒のみ受け付けられる場合でも、グループ数やサンプルサイズによっては点の形より、彩度や明るさの方が効果的な場合もあります。

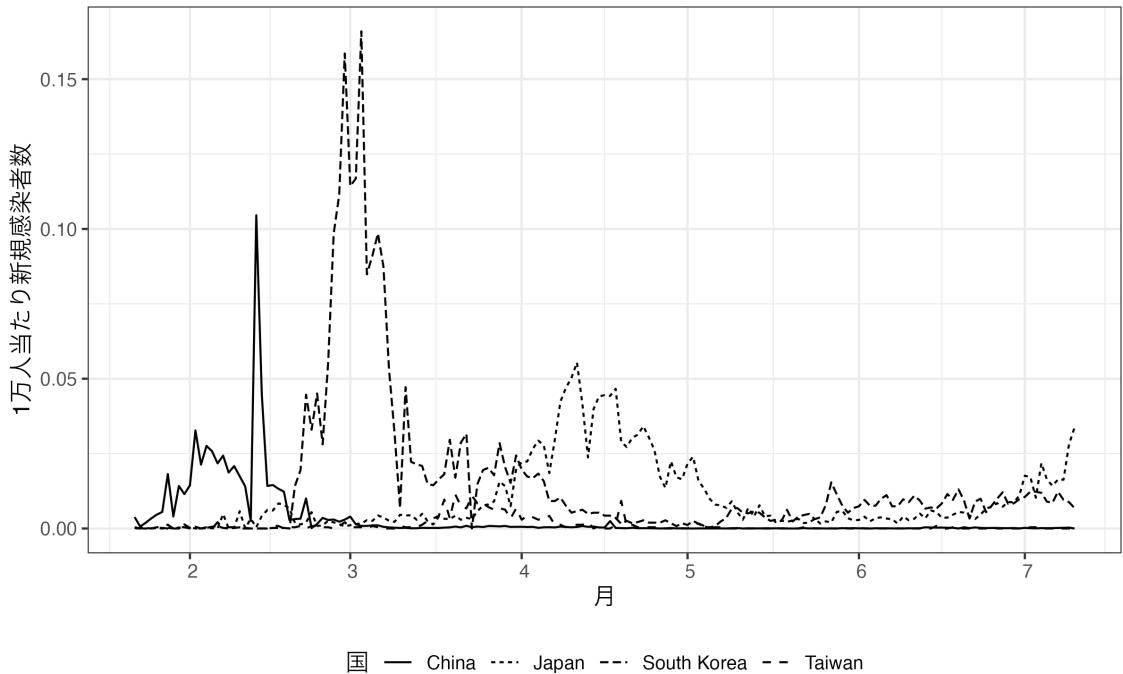
続きまして、`linetype`について解説します。線の形も色分けができない場合、よく使われる次元の増やし方です。ここでは日中韓台における人口 1 万人あたり COVID-19 新規感染者数の折れ線グラフを作成します。`COVID19_df` には人口のデータがないため、`left_join()` を使って `Country_df` と結合します。`left_join()` の使い方に関しては第 13 章を参照してください。続いて、Character 型である `Date` 変数を `as.Date()` 関数を使って `Date` 型へ変換します。1 万人あたり新規感染者数は新規感染者数 (`Confirmed_Day`) を人口 (`Population`) で割り、1 万をかけます。最後に `Country` 列を基準に `filter()` を使用し、日中韓台のデータのみ残します。後は折れ線グラフを作成しますが、`geom_line()` 内の `aes()` 内に `linetype` 引数を `Country` 変数でマッピングします。

```
1 left_join(COVID19_df, Country_df, by = "Country") %>%
2   mutate(Date = as.Date(Date),
3         Confirmed_per_capita = Confirmed_Day / Population * 10000) %>%
4   filter(Country %in% c("Japan", "China", "Taiwan", "South Korea")) %>%
```

```

5  ggplot() +
6  geom_line(aes(x = Date, y = Confirmed_per_capita, linetype = Country)) +
7  labs(x = "月", y = "1万人当たり新規感染者数", linetype = "国") +
8  theme_bw(base_size = 12) +
9  theme(legend.position = "bottom")

```



いかがでしょうか。linetype は color よりも識別性が非常に低いことが分かるでしょう。個人差もあるかも知れませんが、shape よりも低いのではないでしょうか。実線の中国を除けば、日本、韓国、台湾の線はなかなか区別できません。したがって、linetype は 2 つ、3 つまでが限界だと考えられます。3 つまででしたら、実線、破線、点線に分けることができるでしょう。ここでは、日本のみを実線とし、他の 3 カ国は「その他」として破線にしてみましょう。そのためには、日本か否かを示す Japan 変数を作成します。また、geom\_line() の aes() 内には groups 引数を追加し、linetype は Japan 変数でマッピングします。

```

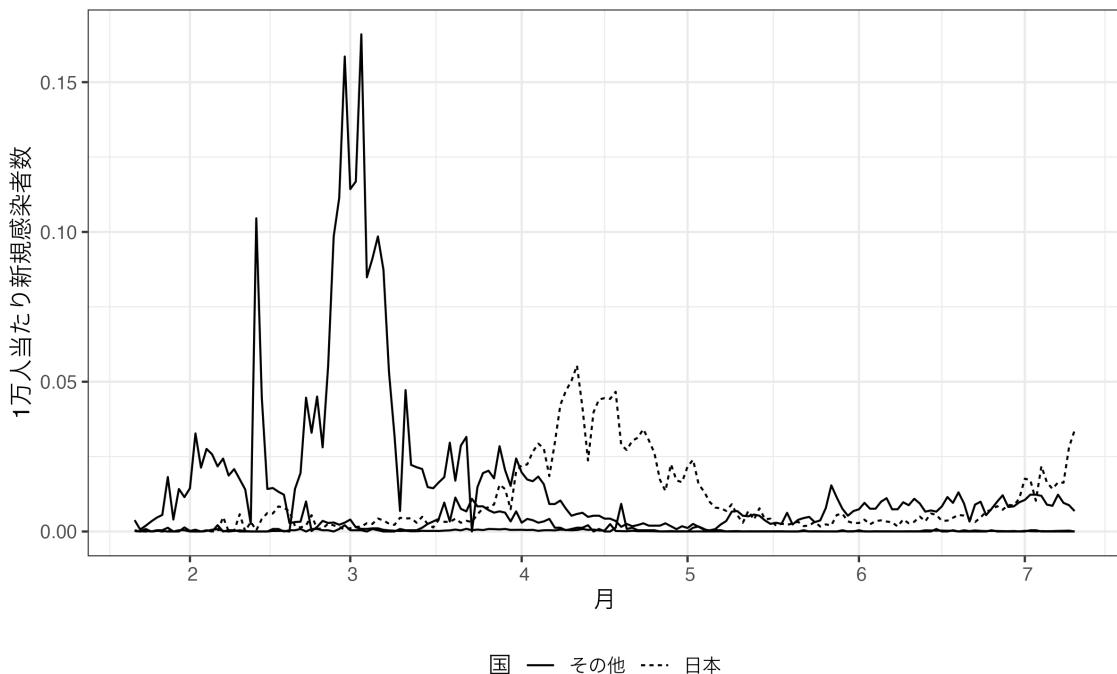
1 left_join(COVID19_df, Country_df, by = "Country") %>%
2   mutate(Date = as.Date(Date),

```

```

3     Confirmed_per_capita = Confirmed_Day / Population * 10000,
4     Japan                 = if_else(Country == "Japan", "日本", "その他")) %>%
5   filter(Country %in% c("Japan", "China", "Taiwan", "South Korea")) %>%
6   ggplot() +
7   geom_line(aes(x = Date, y = Confirmed_per_capita,
8                 group = Country, linetype = Japan)) +
9   labs(x = "月", y = "1万人当たり新規感染者数", linetype = "国") +
10  theme_bw(base_size = 12) +
11  theme(legend.position = "bottom")

```



その他の国が実線となっているので、`scale_linetype_manual()` で Japan の値ごとに線のタイプを指定します。

```

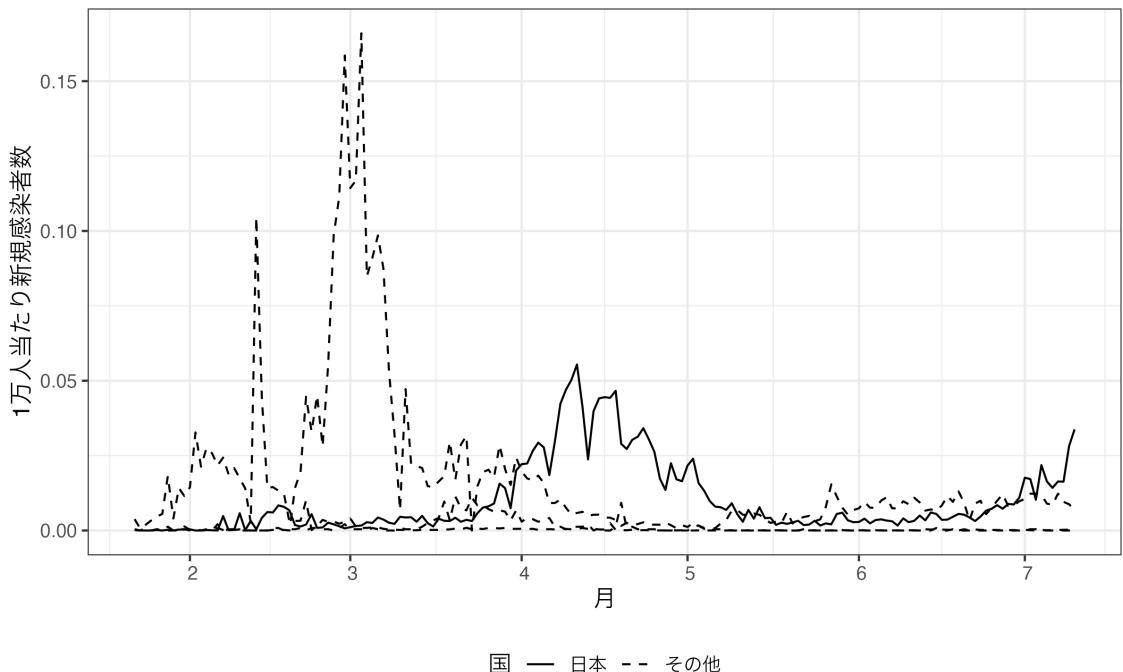
1 left_join(COVID19_df, Country_df, by = "Country") %>%
2   mutate(Date           = as.Date(Date),
3         Confirmed_per_capita = Confirmed_Day / Population * 10000,
4         Japan              = if_else(Country == "Japan", "日本", "その他")) %>%

```

```

5  filter(Country %in% c("Japan", "China", "Taiwan", "South Korea")) %>%
6  ggplot() +
7  geom_line(aes(x = Date, y = Confirmed_per_capita,
8              group = Country, linetype = Japan)) +
9  scale_linetype_manual(values = c("日本" = 1, "その他" = 2)) +
10 labs(x = "月", y = "1万人当たり新規感染者数", linetype = "国") +
11 theme_bw(base_size = 12) +
12 theme(legend.position = "bottom")

```



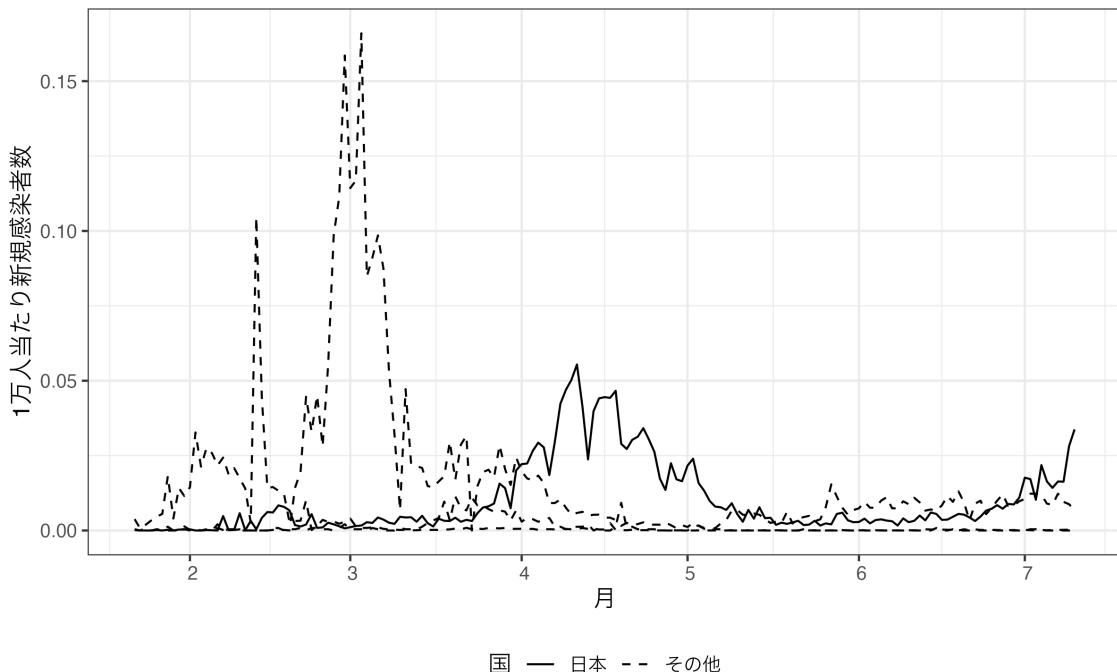
できれば、日本と他の順番も逆にしたいですね。こちらは `Japan` 変数を factor 化することで対応可能です。

```

1 left_join(COVID19_df, Country_df, by = "Country") %>%
2   mutate(Date           = as.Date(Date),
3         Confirmed_per_capita = Confirmed_Day / Population * 10000,
4         Japan             = if_else(Country == "Japan", "日本", "その他"),
5         Japan             = factor(Japan, levels = c("日本", "その他"))) %>%

```

```
6   filter(Country %in% c("Japan", "China", "Taiwan", "South Korea")) %>%
7     ggplot() +
8     geom_line(aes(x = Date, y = Confirmed_per_capita,
9                   group = Country, linetype = Japan)) +
10    scale_linetype_manual(values = c("日本" = 1, "その他" = 2)) +
11    labs(x = "月", y = "1万人当たり新規感染者数", linetype = "国") +
12    theme_bw(base_size = 12) +
13    theme(legend.position = "bottom")
```

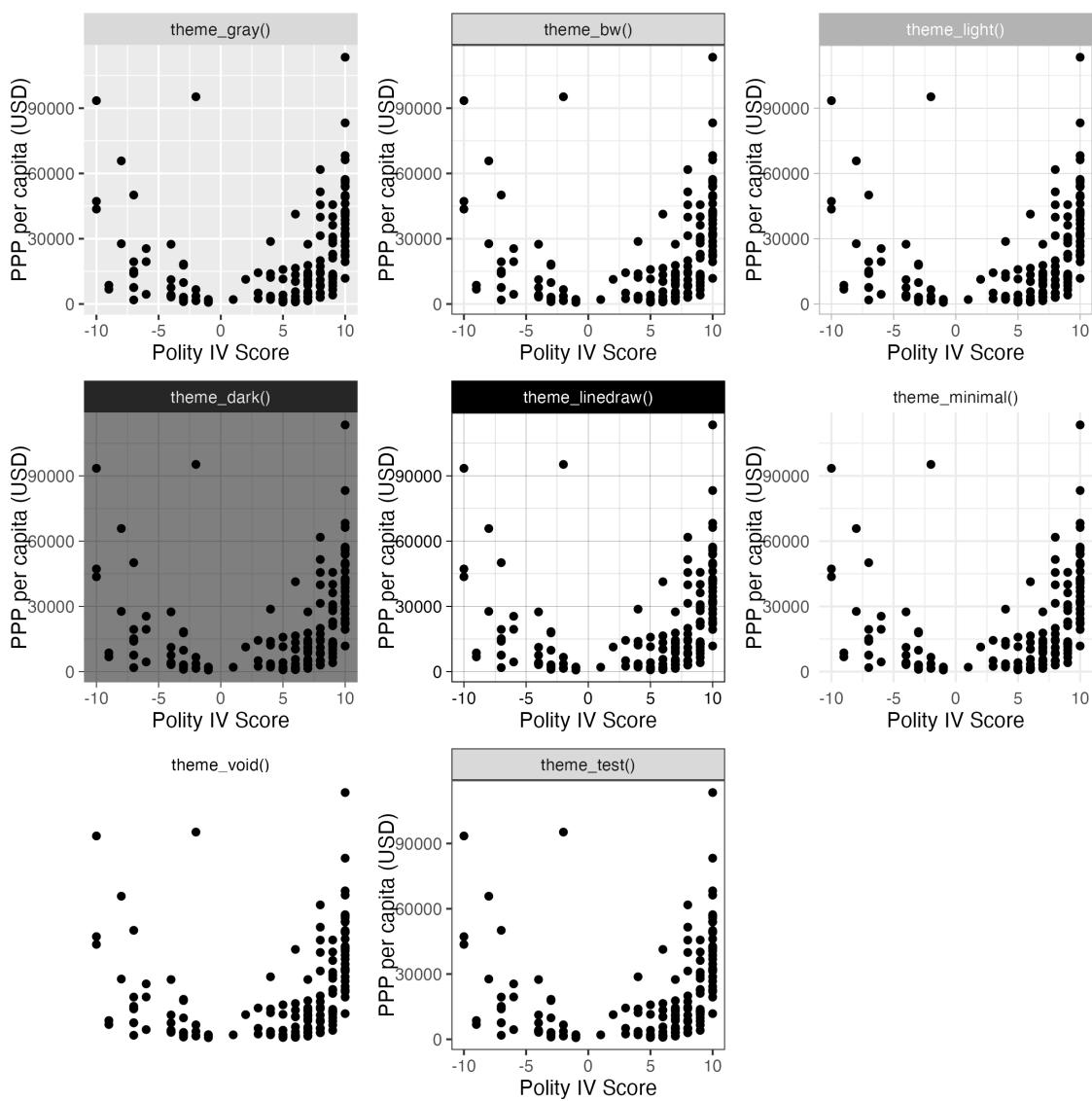


これで完成ですがいかがでしょうか。複数の線を識別するという意味では色分け (color) が優れていますし、ハイライトなら透明度 (alpha) か線の太さ (size) の方が良いでしょう。筆者 (SONG) としましては linetype による次元の追加はあまりオススメしませんが、知つといて損はないでしょう。

---

## 19.4 theme\_\*() と theme(): テーマの指定

続いて図全体の雰囲気を決める `theme_*`() レイヤーについて解説します。これらの使い方は非常に簡単であり、`ggplot` オブジェクトに `theme_*`() レイヤーを + で繋ぐだけです。もし、こちらのレイヤーを追加しない場合、デフォルトテーマとして `theme_gray()` が適用されます。`{ggplot2}` はいくつかのテーマを提供しており、以下がその例です。



他にも{ggplot2}用のテーマをパッケージとしてまとめたものもあります。興味のある方は ggtheme や ggthemr ページを確認してみてください。

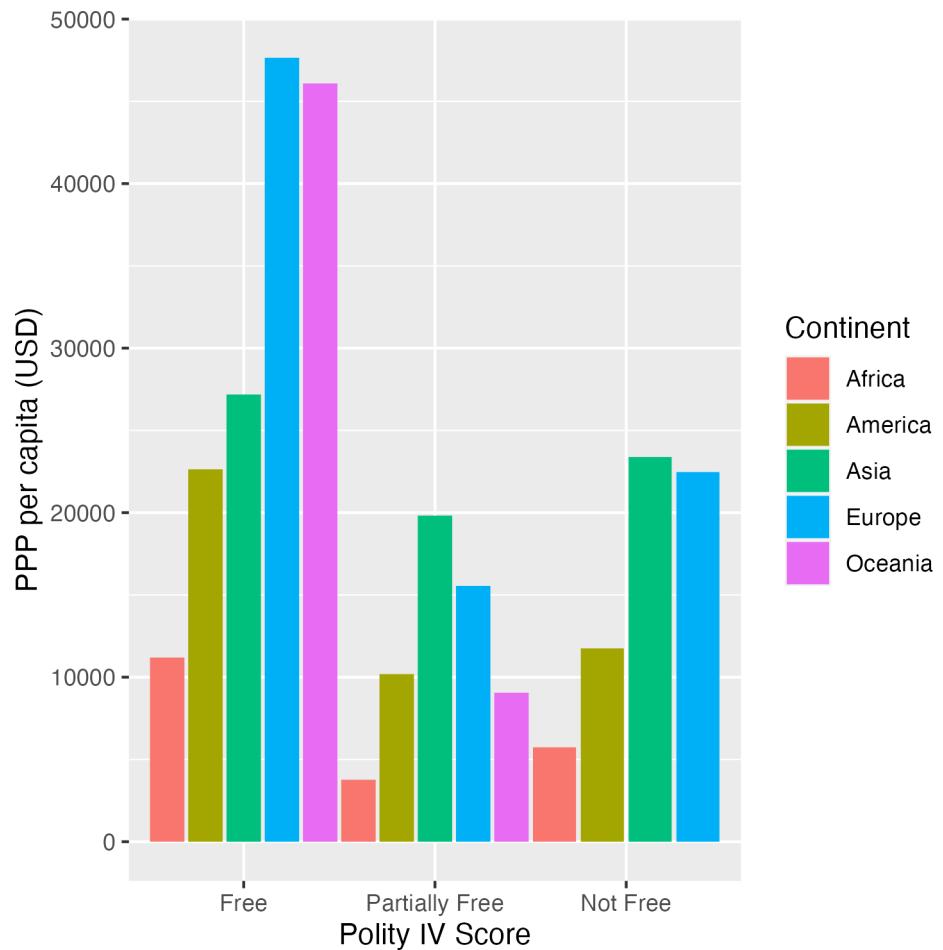
theme\_\*() 内部ではいくつかの引数を指定することができます。最もよく使われるが base\_family 引数であり、図で使用するフォントを指定する引数です。macOS ユーザーだとヒラギノ角ゴシック W3 が良く使われており、`base_family = "HiraginoSans-W3"`で設定可能です。他にも全体の文字サイズを指定する `base_size` などがあります。

テーマの微調整は主に theme() レイヤーで行います。こちらでは図の見た目に関する細かい調整ができます。実は theme\_\*() 関数群は調整済み theme() レイヤーとして捉えることも出来ます。theme\_\*() と theme() を同時に使うことも可能であり、「全般的には minimal テーマ (theme\_minimal()) が好きだけど、ここだけはちょっと修正したい」場合に使用します。theme() では図の見た目に関する全ての部分が設定可能であるため、引数も膨大です。詳しくはコンソール上で?theme を入力し、ヘルプを確認してください。ここではいくつかの例のみを紹介します。

まずは実習用データとして任意の棒グラフを作成し、Theme\_Fig という名のオブジェクトとして保存しておきます。

```
1 Theme_Fig <- Country_df %>%
2   # Freedom House の Status、Continent でグループ化
3   group_by(FH_Status, Continent) %>%
4   # 欠損値のある行を除き、PPP_per_capita の平均値を計算
5   summarise(PPP      = mean(PPP_per_capita, na.rm = TRUE),
6             .groups = "drop") %>%
7   # 欠損値の行を除去
8   drop_na() %>%
9   # Freedom House の Status を再ラベリングし、factor 化
10  mutate(FH_Status = case_when(FH_Status == "F" ~ "Free",
11                                FH_Status == "PF" ~ "Partially Free",
12                                FH_Status == "NF" ~ "Not Free"),
13                                FH_Status = factor(FH_Status, levels = c("Free", "Partially Free",
14                                "Not Free")))) %>%
15                                ggpplot() +
```

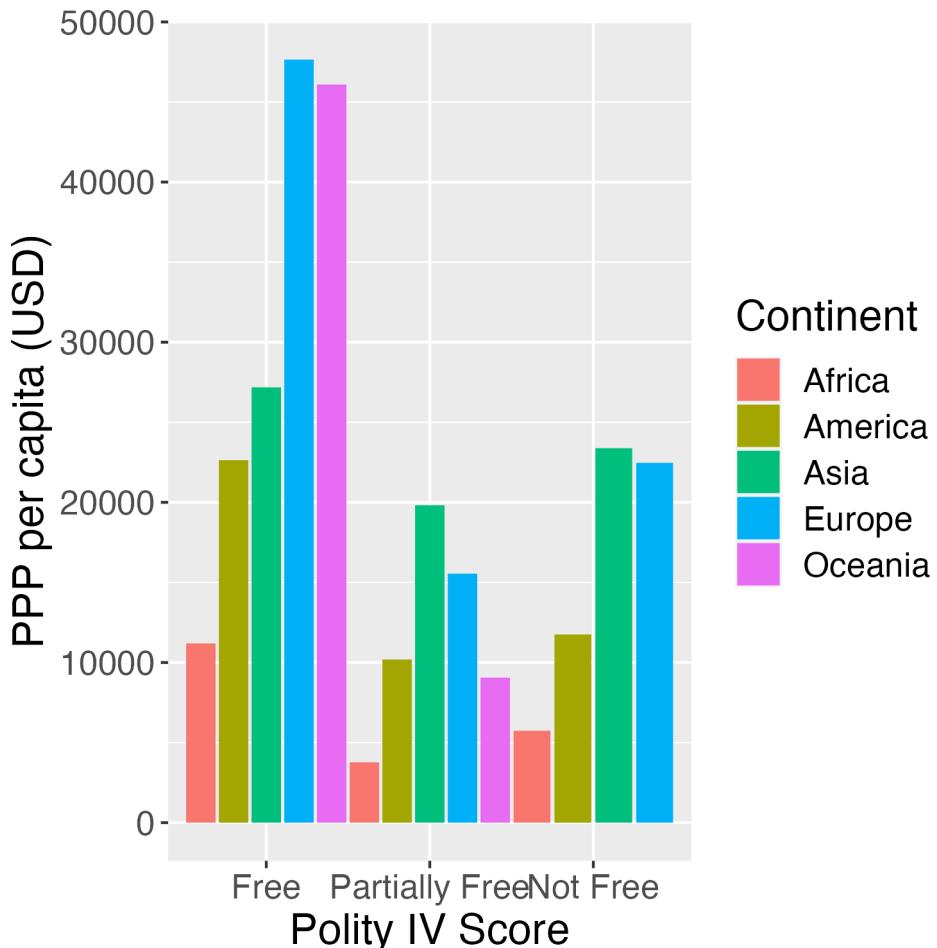
```
16  geom_bar(aes(x = FH_Status, y = PPP, fill = Continent),  
17      stat = "identity", position = position_dodge(width = 1)) +  
18  labs(x = "Polity IV Score", y = "PPP per capita (USD)")  
19  
20 Theme_Fig
```



#### 19.4.1 文字の大きさ

図全体における文字の大きさは `theme_*`() レイヤーの `base_size` から調整することができますが、`theme()` からも可能です。文字に関する調整は `text` 引数に対して `element_text()` 実引数を指定します。大きさの場合、`element_text()` 内に `size` 引数を指定します。

```
1 Theme_Fig +  
2   theme(text = element_text(size = 16))
```



`element_text()` では文字の大きさ以外にも色 (color)、回転の度合い (angle) などを指定することもできます。詳しくは R コンソール上で`?element_text` を入力し、ヘルプを確認してください。

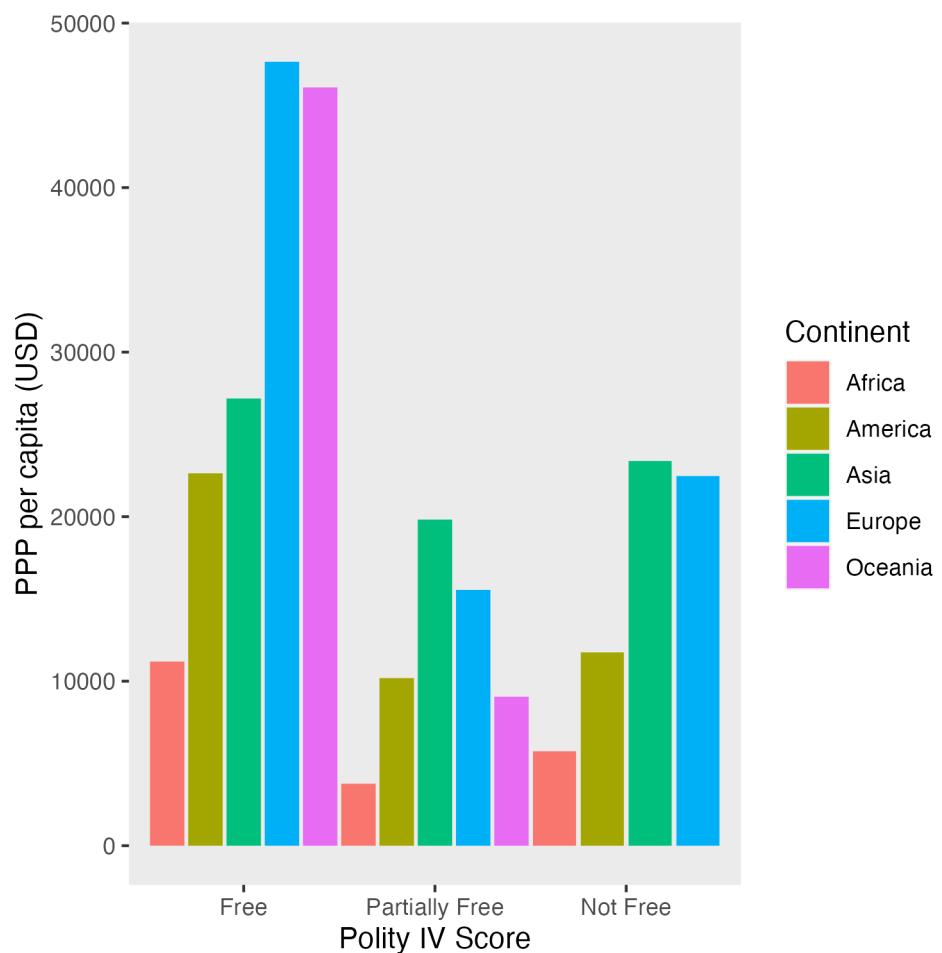
#### 19.4.2 背景のグリッド

背景のグリッドを調整する際は `panel.gird.*` 引数を使います。主に使用する場面はグリッドの除去する場合ですが、この場合は実引数として `element_blank()` を使います。もし、グリッドの色や太さなどを変更したい場合は `element_line()` を使用します。

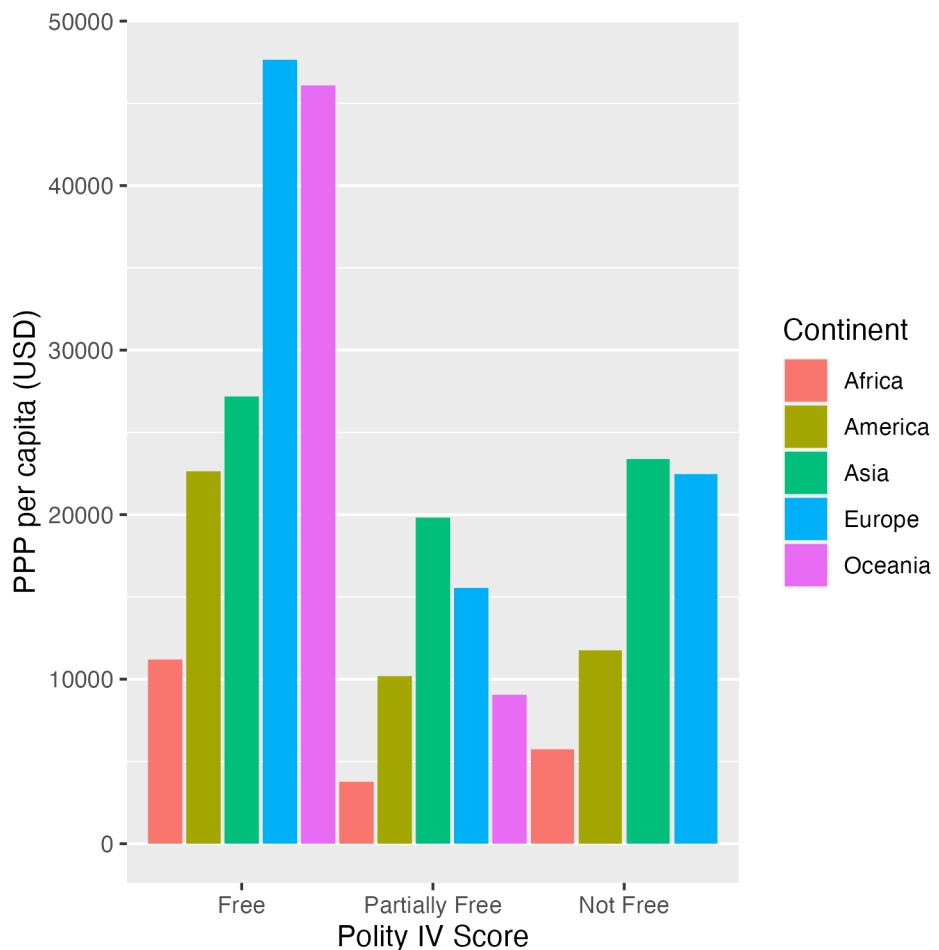
ここではグリッドを除去する方法について紹介します。

グリッドを除去する場合、どのグリッドを除去するかを決めないといけません。例えば、すべてのグリッドを除去するためには `panel.grid = element_blank()` を使います。また、メジャーグリッドの除去には `panel.grid.major` (すべて)、`panel.grid.major.x` (横軸のみ)、`panel.grid.major.y` (縦軸のみ) を指定します。マイナーグリッドの場合は `major` を `minor` に替えてください。以下にはいくつかの例をお見せします。

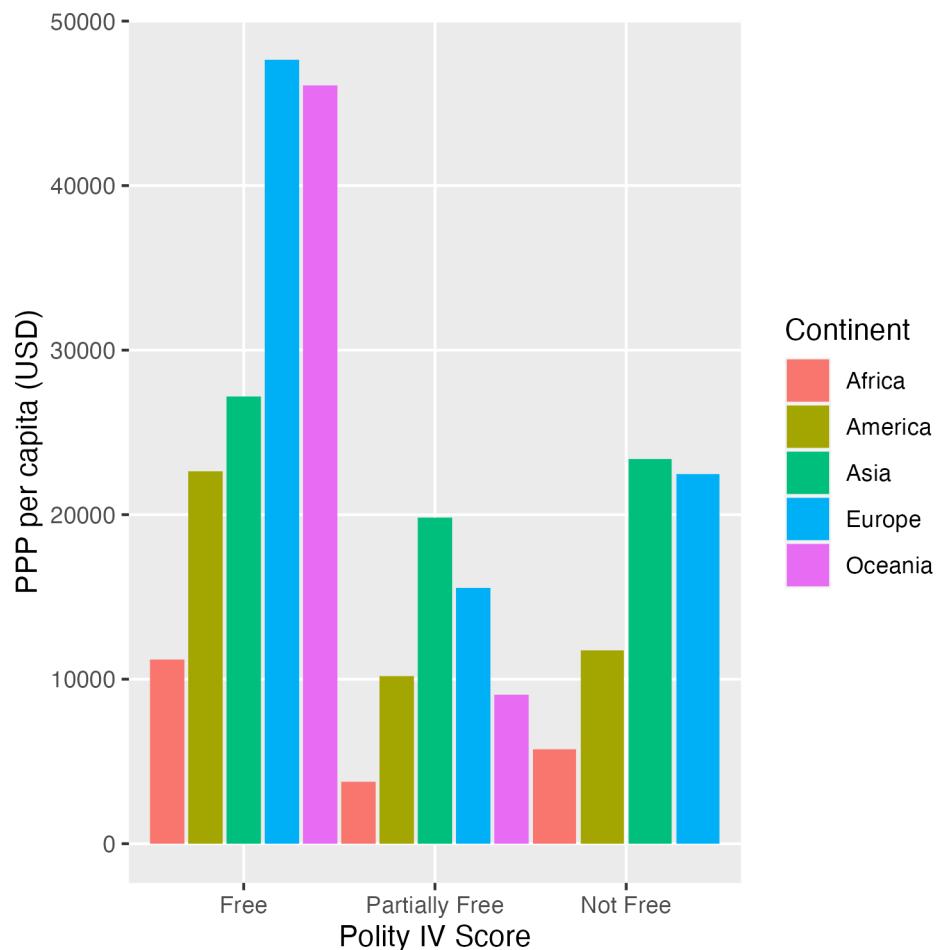
```
1 # すべてのグリッドを除去
2 Theme_Fig +
3   theme(panel.grid = element_blank())
```



```
1 # x 軸のメジャーグリッドを除去
2 Theme_Fig +
3   theme(panel.grid.major.x = element_blank())
```



```
1 # y 軸のマイナーグリッドのみ除去
2 Theme_Fig +
3   theme(panel.grid.minor.y = element_blank())
```



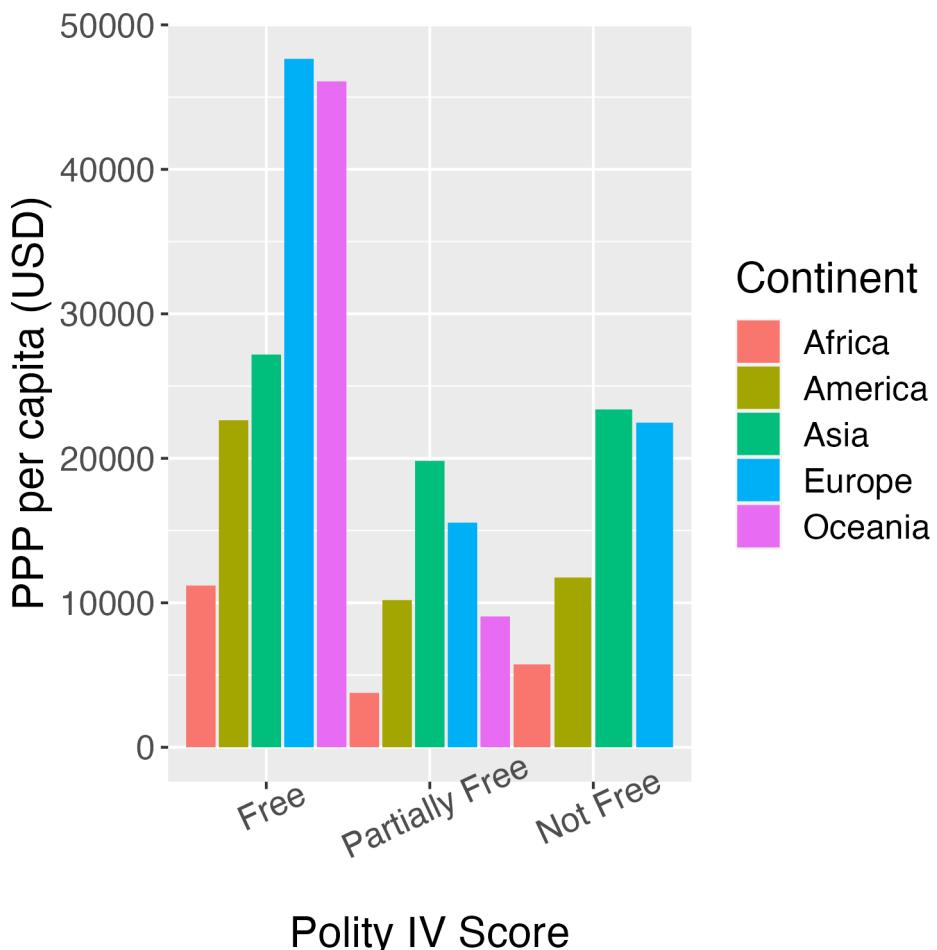
### 19.4.3 目盛りラベルの回転

横軸の目盛りラベルが長すぎるか大きすぎると、ラベルが重なる場合があります。この場合の対処方法としてはラベルの長さを短くしたり、改行 (\n) を入れることが考えられます。また、ラベルを若干回転することでも対処可能です。たとえば、横軸の目盛りラベルを調整する場合は `axis.text.x = element_text()` を指定し、`element_text()` 内に `angle` 引数を指定します。例えば、反時計回りで 25 度回転させる場合は `angle = 25` と指定します。

```

1 Theme_Fig +
2   theme(text = element_text(size = 16),
3         axis.text.x = element_text(angle = 25))

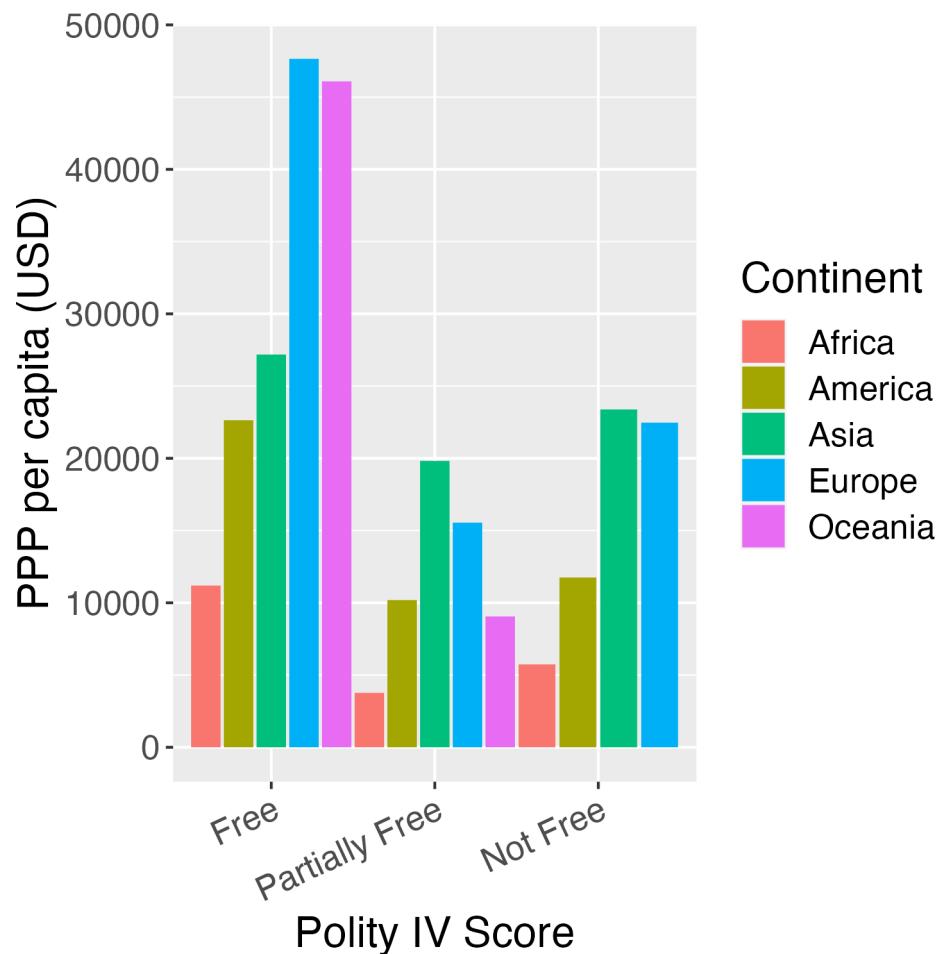
```



## Polity IV Score

この場合、ラベルは目盛りのすぐ下を基準に回転することになります。もし、ラベルの最後の文字を目盛りの下に移動させる場合は `hjust = 1` を追加します。

```
1 Theme_Fig +  
2   theme(text = element_text(size = 16),  
3         axis.text.x = element_text(angle = 25, hjust = 1))
```



ちなみに、`angle = 90`などで指定するとラベルが重なる問題はほぼ完全に解決されますが、かなり読みづらくなるので、できれば`angle`の値は小さめにした方が読みやすくなります。

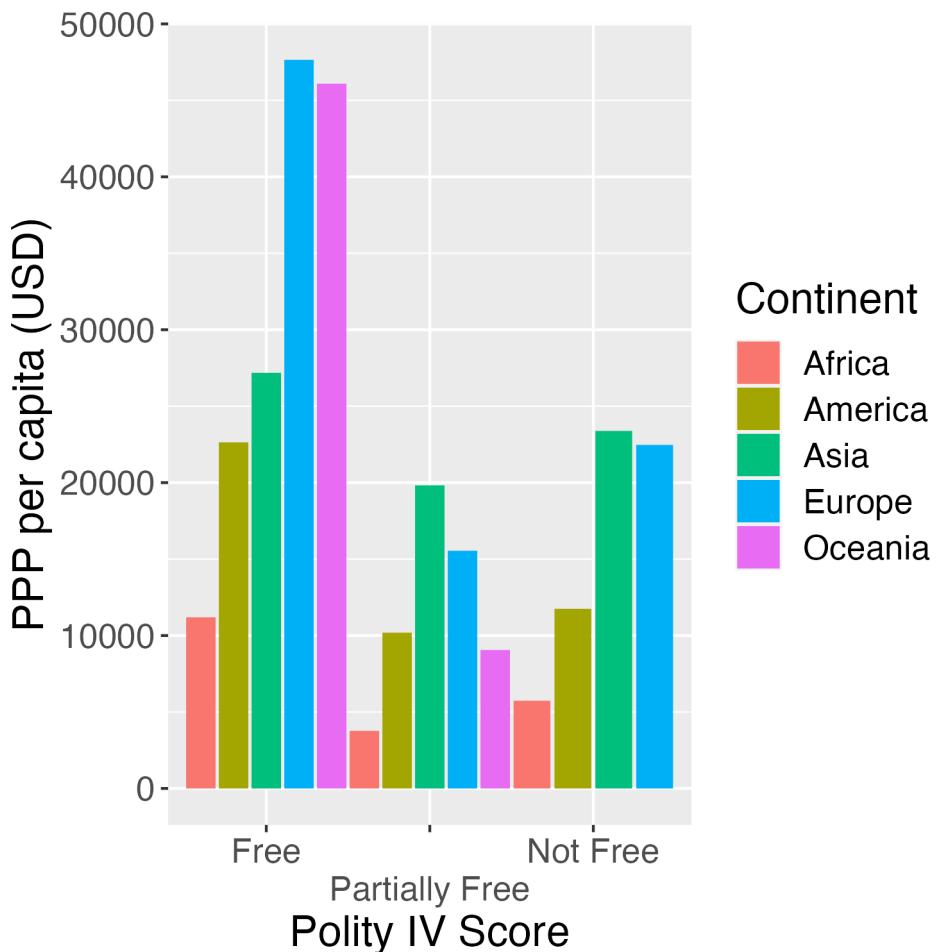
#### 19.4.4 `scale_*_()` を用いたラベル重複の回避

ラベルの重複を回避するもう一つの方法は「ラベルの位置をずらす」ことです。これは`{ggplot2}3.3.0`以降追加された機能であり、`theme()`でなく、`scale_*_()`関数の`guide`引数で指定することができます。使い方は以下の通りです。たとえば、横軸(x軸)のラベルを2行構成にしたい場合は以下のように指定します。

```
1 ggplot2 オブジェクト名 +
2   scale_x_discrete(guide = guide_axis(n.dodge = 2))
```

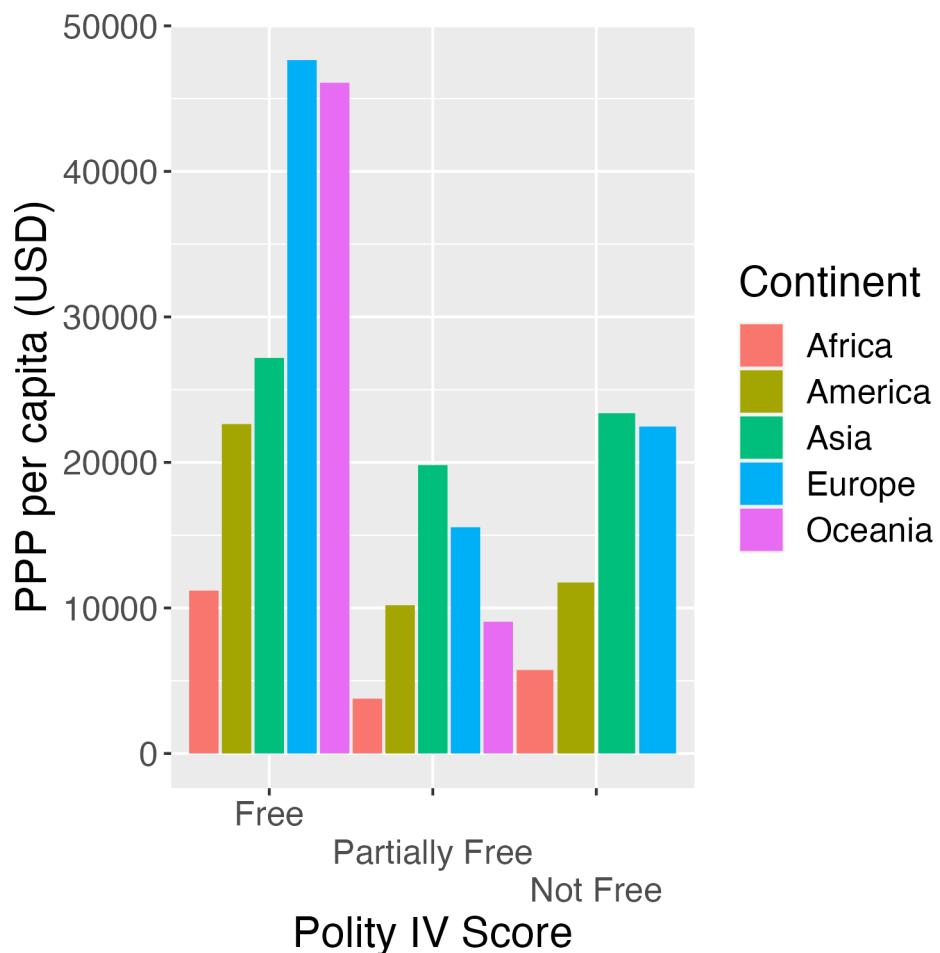
実際の結果を確認してみましょう。

```
1 Theme_Fig +
2   scale_x_discrete(guide = guide_axis(n.dodge = 2)) +
3   theme(text = element_text(size = 16))
```



横軸のラベルが 2 行構成になりました。左から最初のラベルは 1 行目に、2 番目のラベルは 2 行目に、3 番目のラベルは 1 行目になります。実際、ラベルをずらすだけなら `n.dodge = 2` で十分ですが、この引数の挙動を調べるために `n.dodge = 3` に指定してみましょう。

```
1  Theme_Fig +  
2    scale_x_discrete(guide = guide_axis(n.dodge = 3)) +  
3    theme(text = element_text(size = 16))
```

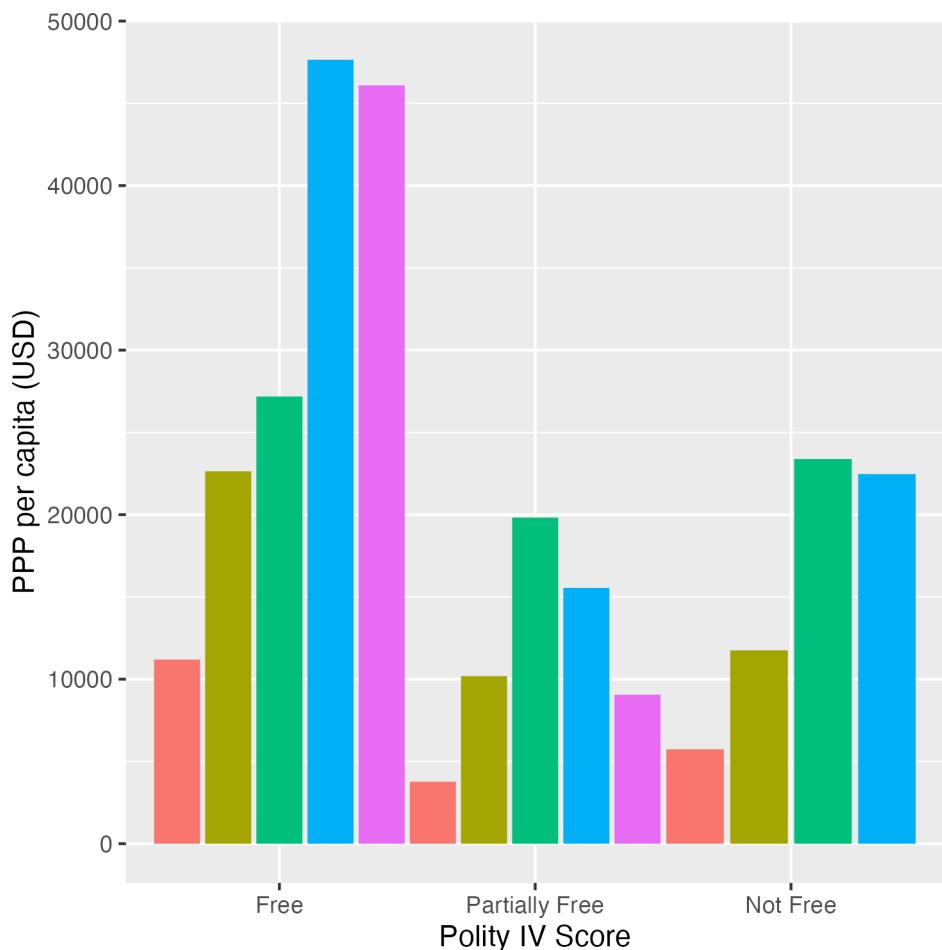


3番目のラベルが3行目に位置することになります。もし、4つ目のラベルが存在する場合、それは1行目に位置するでしょう。

#### 19.4.5 凡例の表示/非表示

凡例を無くす方法はいくつかありますが、まずはすべての凡例を非表示する方法について紹介します。それは後ほど紹介します `legend.position` の実引数として "none" を指定する方法です。

```
1 # 凡例を非表示にする
2 Theme_Fig +
3   theme(legend.position = "none")
```

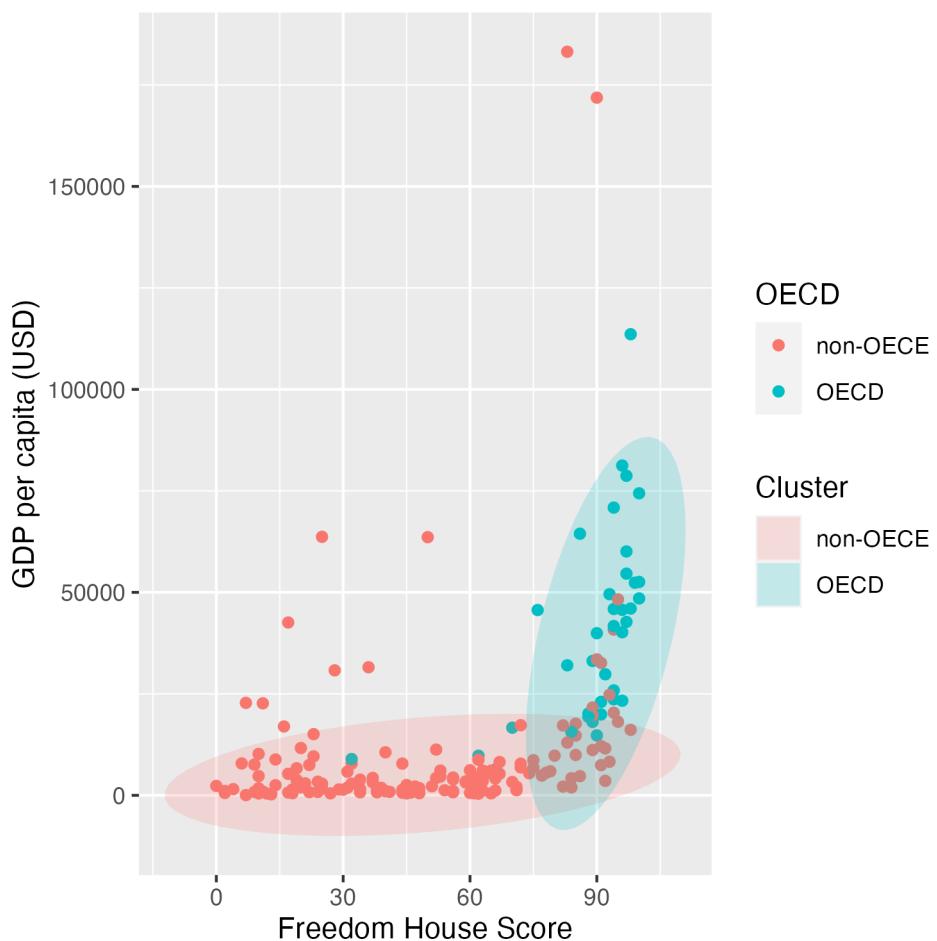


Theme\_Fig は x と y 以外に fill にマッピングをしたため、凡例は一つのみとなります。ただし、場合によってはもっと次元を増やすことによって 2 つ以上の凡例が表示されるケースがあります。別途の説明なくとも図だけで理解するのが理想なので凡例は出来る限り温存させた方が良いでしょう。しかし、実例はあまり多く見られないと思いますが、凡例がなくても理解に問題がないと判断される場合は一部の凡例を非表示することも考えられます。

たとえば、以下のような Theme\_Fig2 の例を考えてみましょう。

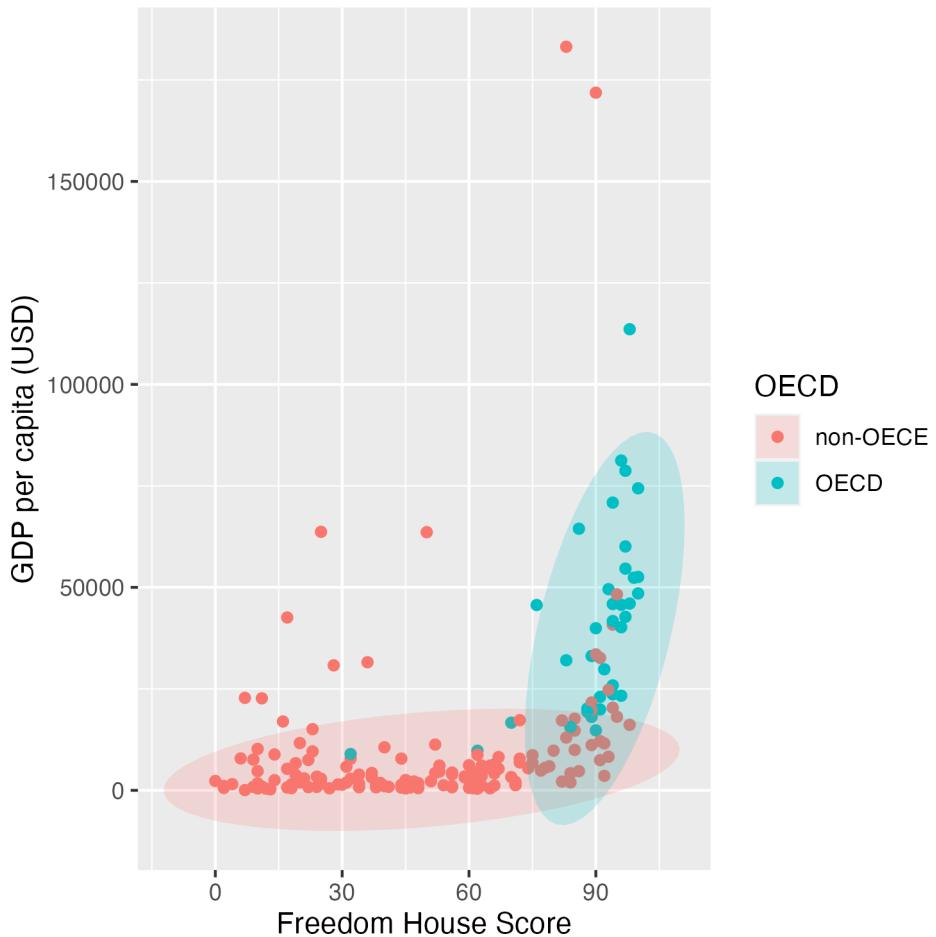
```
1 Theme_Fig2 <- Country_df %>%
2   mutate(OECD = if_else(OECD == 1, "OECD", "non-OECE")) %>%
3   ggplot(aes(x = FH_Total, y = GDP_per_capita)) +
4   geom_point(aes(color = OECD)) +
5   stat_ellipse(aes(fill = OECD), geom = "polygon", level = 0.95, alpha = 0.2) +
6   labs(x = "Freedom House Score", y = "GDP per capita (USD)",
7        color = "OECD", fill = "Cluster")
8
9 Theme_Fig2
```

## Warning: Removed 1 rows containing non-finite values (stat\_ellipse).  
## Warning: Removed 1 rows containing missing values (geom\_point).



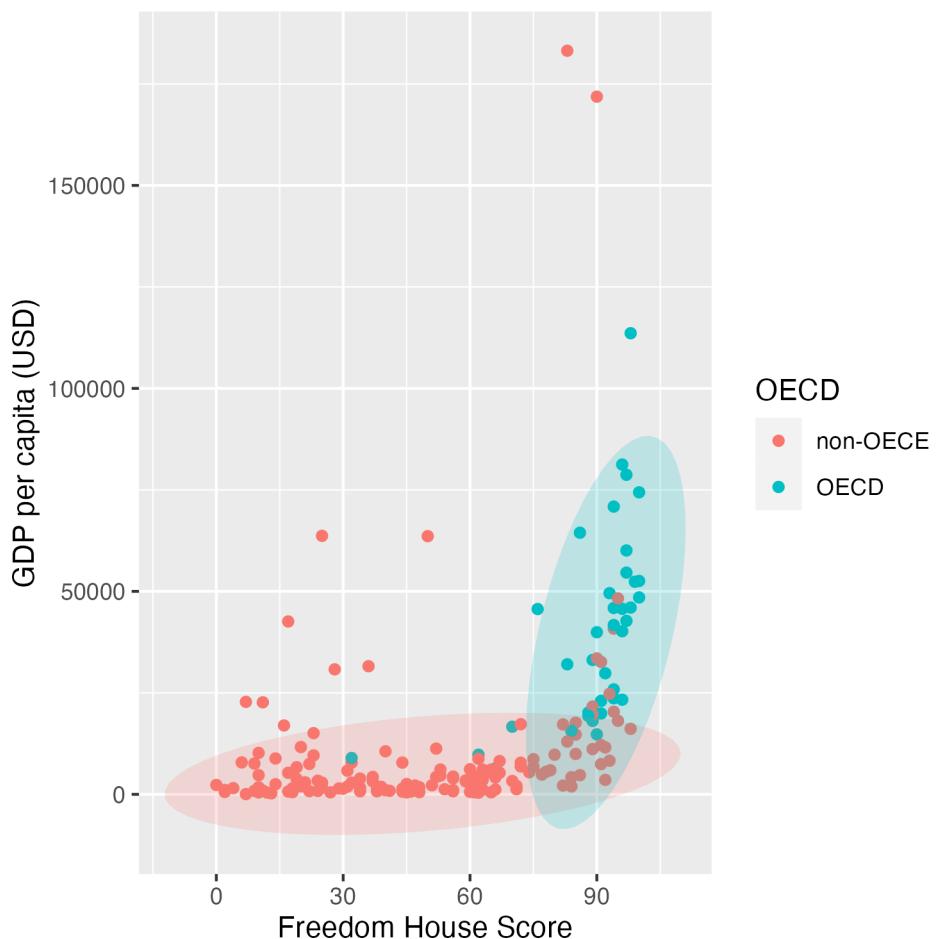
color と fill がそれぞれ別の凡例として独立しています。この例の場合、fill の凡例はなくても、図を理解するのは難しくないかも知れません。ここで考えられる一つの方法は color と fill の凡例をオーバラップさせる方法です。{ggplot2}の場合、同じ変数がマッピングされていれば凡例をオーバーラップさせることも可能です。ただし、凡例のタイトルが同じである必要があります。ここでは color と fill のタイトルを"OECD"に統一してみましょう。

```
1 Theme_Fig2 +  
2   labs(color = "OECD", fill = "OECD")  
  
## Warning: Removed 1 rows containing non-finite values (stat_ellipse).  
  
## Warning: Removed 1 rows containing missing values (geom_point).
```



これで十分でしょう。しかし、凡例から fill の情報を完全に消したい場合はどうすれば良いでしょうか。その時に登場するのが `guides()` 関数です。関数の中でマッピング要素 = "none"を指定すると、当該凡例が非表示となります。

```
1 Theme_Fig2 +
2   guides(fill = "none")  
  
## Warning: Removed 1 rows containing non-finite values (stat_ellipse).  
  
## Warning: Removed 1 rows containing missing values (geom_point).
```

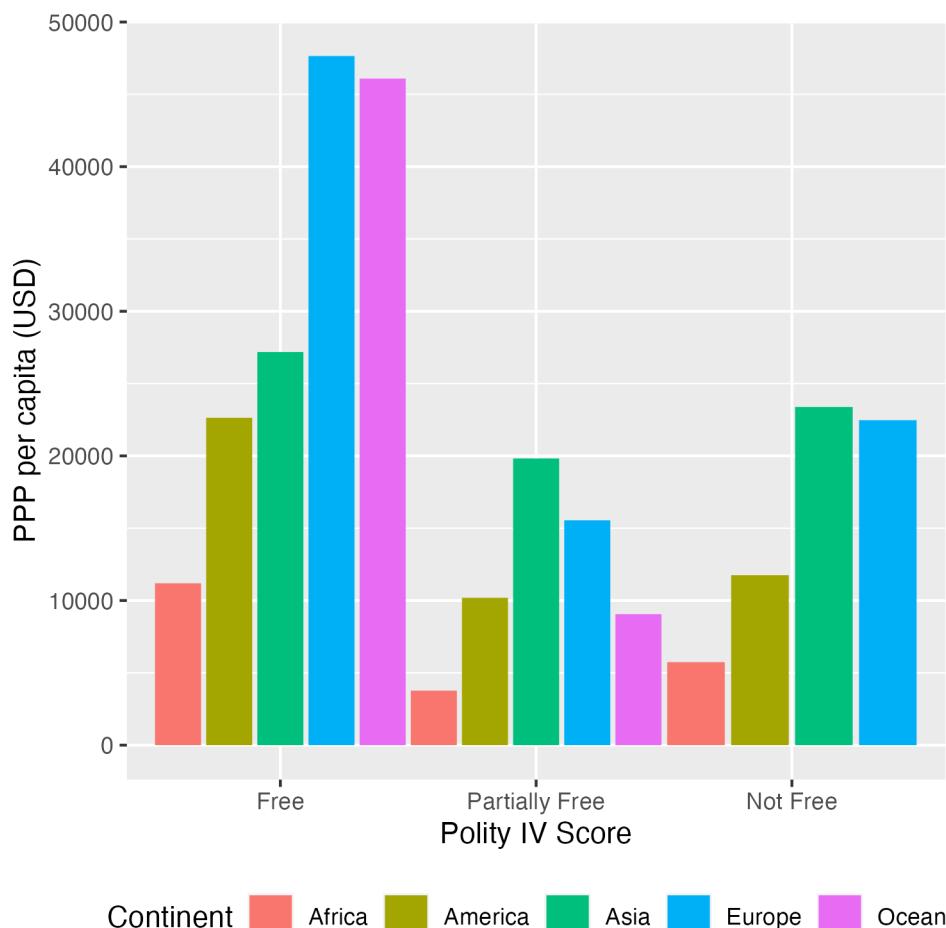


`guides()` 関数は凡例を細かく調整できる様々な機能を提供しています。興味のある方はヘルプ (`?guides`) を参照してください。

### 19.4.6 凡例の位置

凡例を図の下段に移動させる場合は `legend.position` 引数に "bottom" を指定するだけです。他にも "top" や "left" も可能ですが、凡例は一般的に図の右か下段に位置しますので、デフォルトのままに置くか、"bottom" くらいしか使わないでしょう。

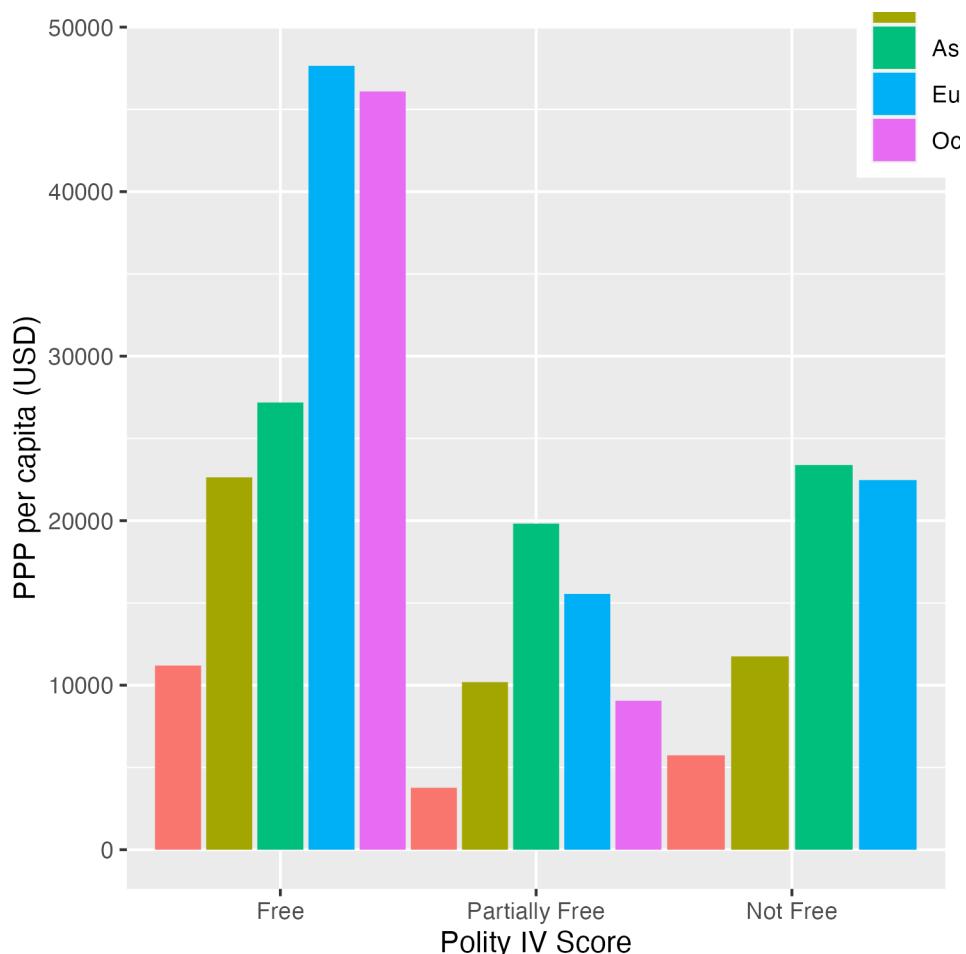
```
1 # 凡例を図の下段へ移動
2 Theme_Fig +
3   theme(legend.position = "bottom")
```



もし、凡例を図の内部に置く場合は、長さ 2 の numeric 型ベクトルを指定します。図の左下なら `c(0, 0)`、左上なら `c(0, 1)`、右上は `c(1, 1)`、右下は `c(1, 0)` となりま

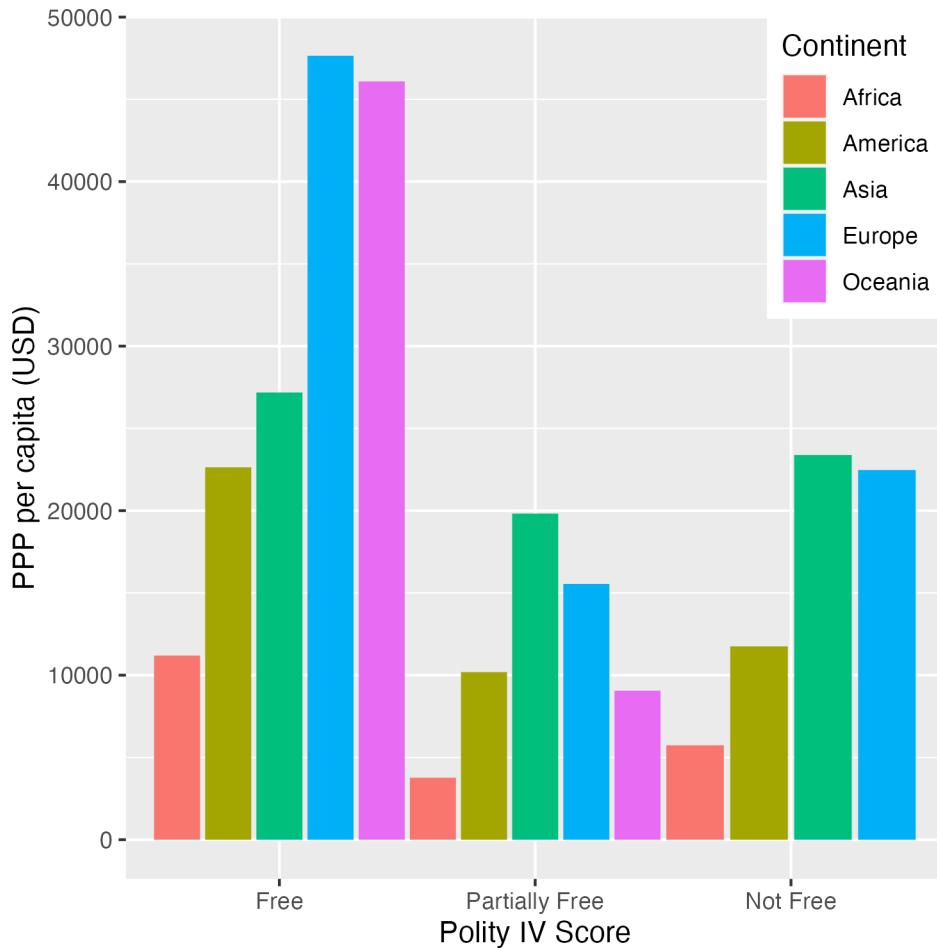
す。これは図の大きさを横縦それぞれ1とした場合の位置を意味します。ここでは凡例を右上へ置いてみましょう。

```
1 # 凡例をプロットの右上へ
2 Theme_Fig +
3   theme(legend.position = c(1, 1))
```



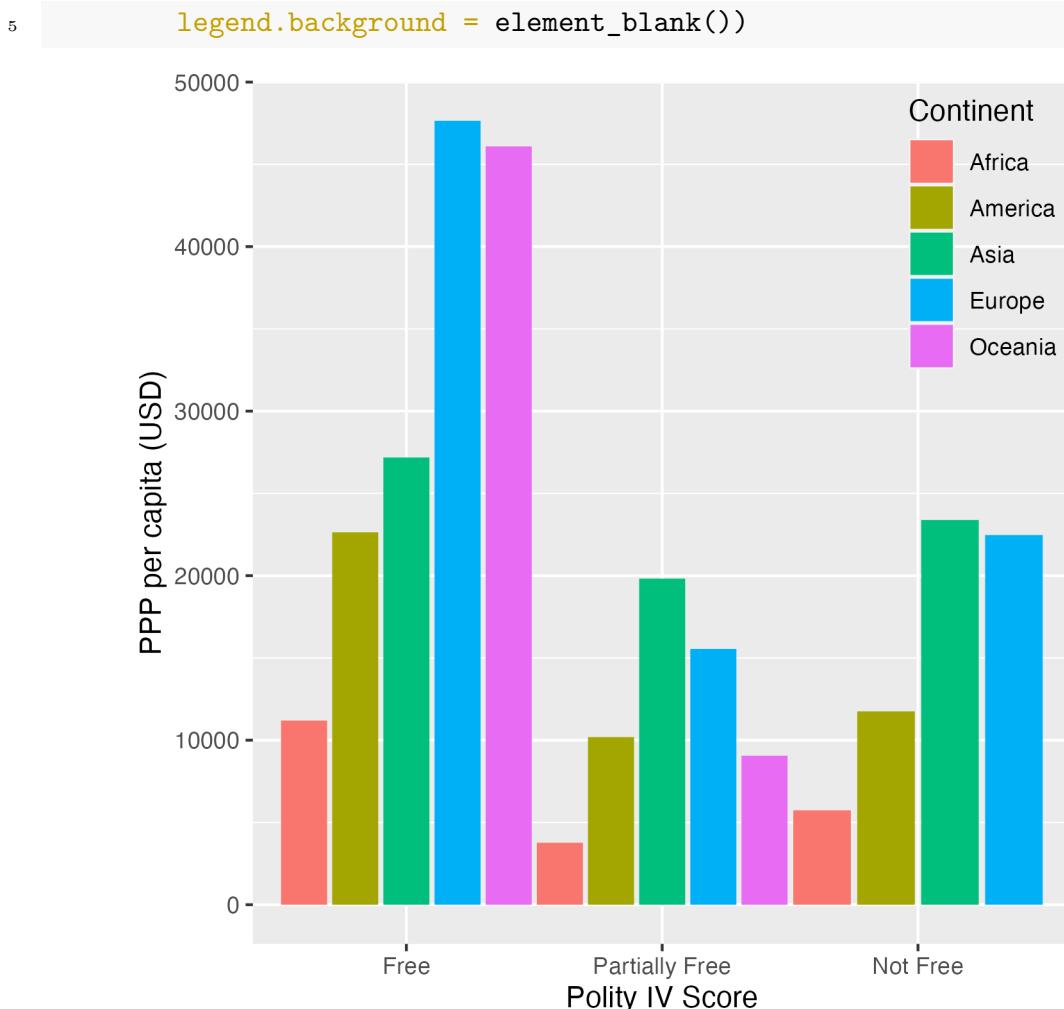
これは凡例の中央が右上に来るようになります。これを是正するためには `legend.justification` 引数を更に指定する必要があります。これにも長さ2の numeric 型ベクトルを指定しますが、これは凡例の中心をどこにするかを意味します。今回の例だと、**凡例の右上を `c(1, 1)` に位置させたいので、ここも `c(1, 1)` と指定します**。基本的に `legend.position` と `legend.justification` は同じ値にすれば問題ないでしょう。

```
1 # 凡例の中心を凡例の右上と指定
2 Theme_Fig +
3   theme(legend.position = c(1, 1),
4         legend.justification = c(1, 1))
```



また、凡例の背景を透明にしたい場合は `legend.background = element_blank()` を指定します。

```
1 # 凡例を背景を透明に
2 Theme_Fig +
3   theme(legend.position = c(1, 1),
4         legend.justification = c(1, 1),
```



`theme()` 関数が提供している昨日は非常に多く、それぞれの実引数として用いられる `element_text()` や `element_line()`、`element_rect()` にも様々な引数が提供されています。図を自分好みに微調整したい方はそれぞれの関数のヘルプを参照してください。

---

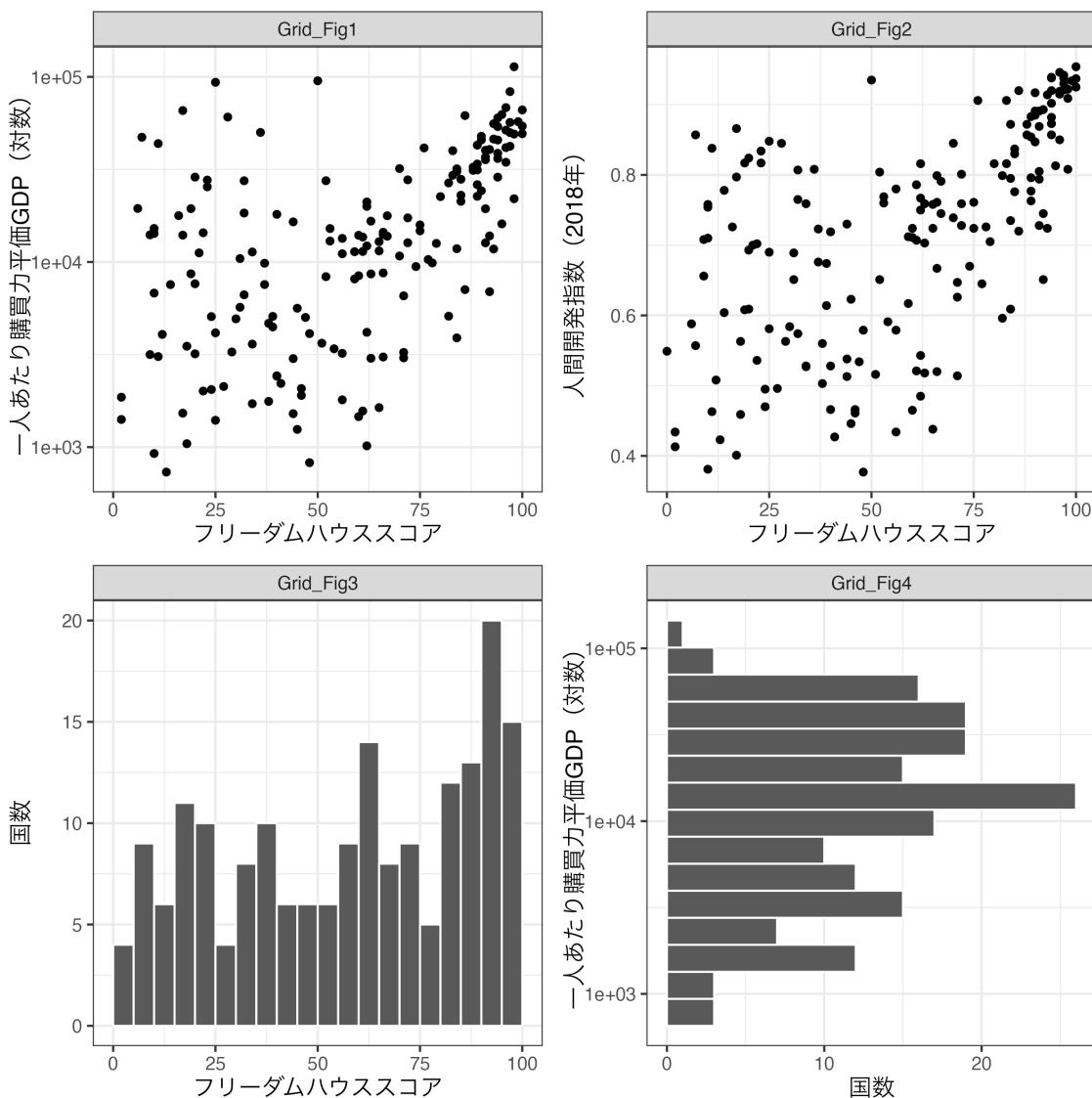
## 19.5 図の結合

{ggplot2}で作成した複数の図を一つの図としてまとめる場合、昔は{gridExtra}一択でした。しかし、今はもっと使いやすいパッケージがいくつか公開されており、ここでは{ggpubr}の `ggarrange()` を紹介します。使い方を紹介する前に、結合する図をいくつか用意し、それぞれ `Grid_Fig1`、`Grid_Fig2`、`Grid_Fig3`、`Grid_Fig4` と名付けます。

```
1 # Grid_Fig1: 散布図
2 # X 軸: フリーダムハウス・スコア / Y 軸: 一人あたり購買力平価 GDP (対数)
3 Grid_Fig1 <- Country_df %>%
4   ggplot() +
5   geom_point(aes(x = FH_Total, y = PPP_per_capita)) +
6   scale_y_log10() +
7   labs(x = "フリーダムハウススコア",
8        y = "一人あたり購買力平価 GDP (対数)") +
9   theme_bw(base_size = 12)
10
11 # Grid_Fig2: 散布図
12 # X 軸: フリーダムハウス・スコア / Y 軸: 2018 年人間開発指数
13 Grid_Fig2 <- Country_df %>%
14   ggplot() +
15   geom_point(aes(x = FH_Total, y = HDI_2018)) +
16   labs(x = "フリーダムハウススコア",
17        y = "人間開発指数 (2018 年)") +
18   theme_bw(base_size = 12)
19
20 # Grid_Fig4: ヒストグラム
21 # X 軸: フリーダムハウス・スコア
22 Grid_Fig3 <- Country_df %>%
23   ggplot() +
24   geom_histogram(aes(x = FH_Total), color = "white",
```

```
25             binwidth = 5, boundary = 0) +
26             labs(x = "フリーダムハウススコア", y = "国数") +
27             theme_bw(base_size = 12)
28
29 # Grid_Fig4: ヒストグラム
30 # Y 軸: 一人あたり購買力平価 GDP (対数)
31 # 時計回りで 90 度回転したヒストグラムを作成するために、y にマッピング
32 Grid_Fig4 <- Country_df %>%
33   ggplot() +
34   geom_histogram(aes(y = PPP_per_capita), color = "white",
35                  boundary = 0, bins = 15) +
36   scale_y_log10() +
37   labs(x = "国数", y = "一人あたり購買力平価 GDP (対数)") +
38   theme_bw(base_size = 12)
```

それぞれの図は以下の通りです。左上、右上、左下、右下の順でそれぞれ Grid\_Fig1、Grid\_Fig2、Grid\_Fig3、Grid\_Fig4 です。

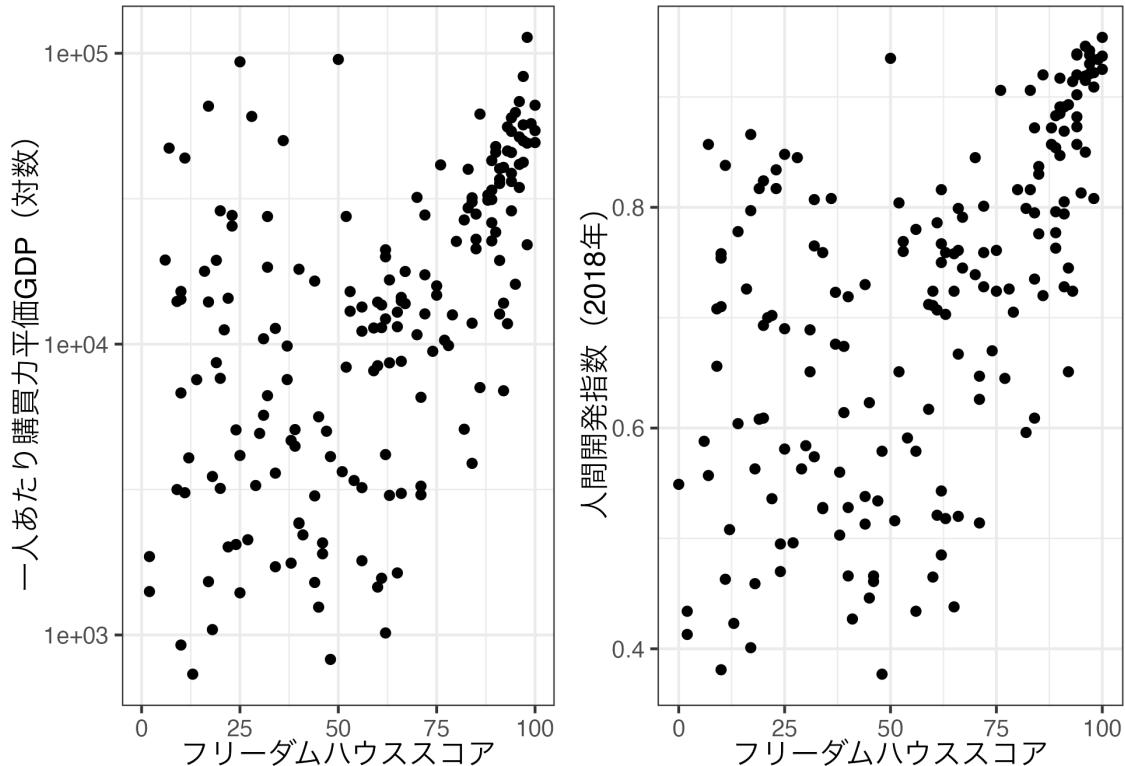


まずは Grid\_Fig1 と Grid\_Fig2 を横に並べてみましょう。まずは{ggpubr}を読み込みます。

```
1 pacman::p_load(ggpubr)
```

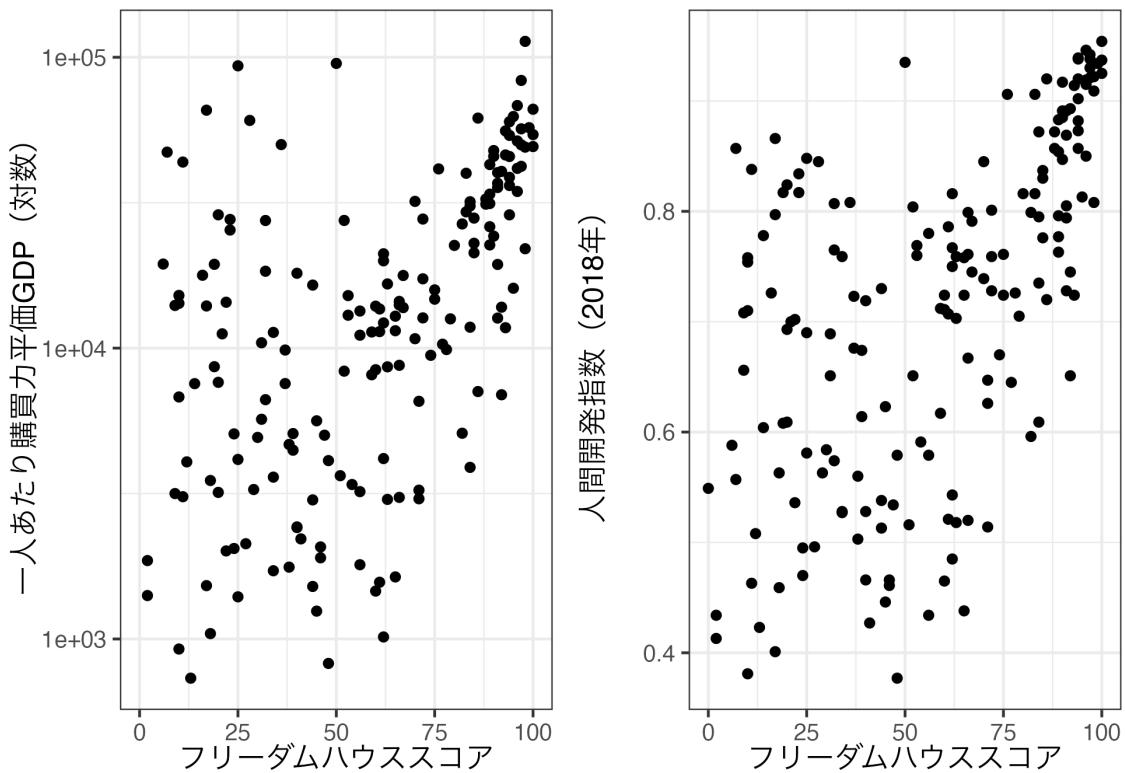
図を並べる際は ggarrange() 関数を使用し、まず、結合したい図のオブジェクトを入力します。また、2つの図を横に並べることは1行2列のレイアウトであることを意味するため、nrow と ncol 引数の実引数としてそれぞれ 1 と 2 を指定します。

```
1 ggarrange(Grid_Fig1, Grid_Fig2, nrow = 1, ncol = 2)
```



右側にある Grid\_Fig2 の幅が若干広いような気がします。この場合、align = "hv" を入れると綺麗に揃えられます。

```
1 ggarrange(Grid_Fig1, Grid_Fig2, nrow = 1, ncol = 2, align = "hv")
```



続きまして、3つの図を以下のように並べるとします。

Grid_Fig3	
Grid_Fig1	Grid_Fig4

今回は3つの図オブジェクトを入れるだけでは不十分です。なぜなら、`ggarrange()`関数は左上から右下の順番へ一つずつ図を入れるからです。もし、3つの図オブジェクトのみを入れると、以下のように配置されます。

Grid_Fig3	Grid_Fig1
Grid_Fig4	

これを回避するためには空欄とするグリッドにNULLを指定する必要があります。

```

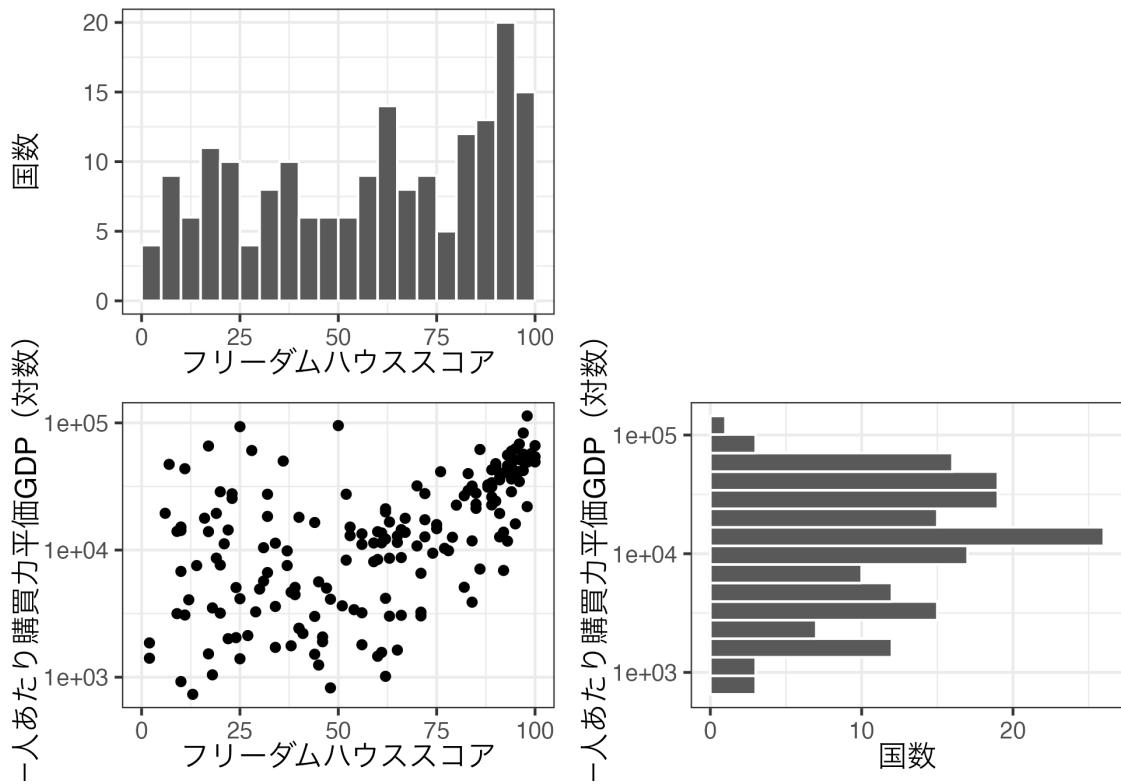
1 ggarrange(Grid_Fig3, # 1行1列目
2             NULL,      # 1行2列目
3             Grid_Fig1, # 2行1列目

```

```

4     Grid_Fig4, # 2行2列目
5     nrow = 2, ncol = 2, align = "hv")

```



次はヒストグラムを小さく調整します。左上の Grid\_Fig3 の上下の幅を、右下の Grid\_Fig4 は左右の幅を狭くします。このためには `widths` と `heights` 引数が必要です。たとえば、`heights = c(0.3, 0.7)` だと 1行目は全体の 30%、2行目は全体の 70% になります。今回は 1行目と 2行目の比率は 3:7 に、1列目と 2列目の比は 7:3 とします。また、それぞれの図に (a)、(b)、(c) を付けます。空欄となるグリッドのラベルは NA か "" にします。NULL ではないことに注意してください。

```

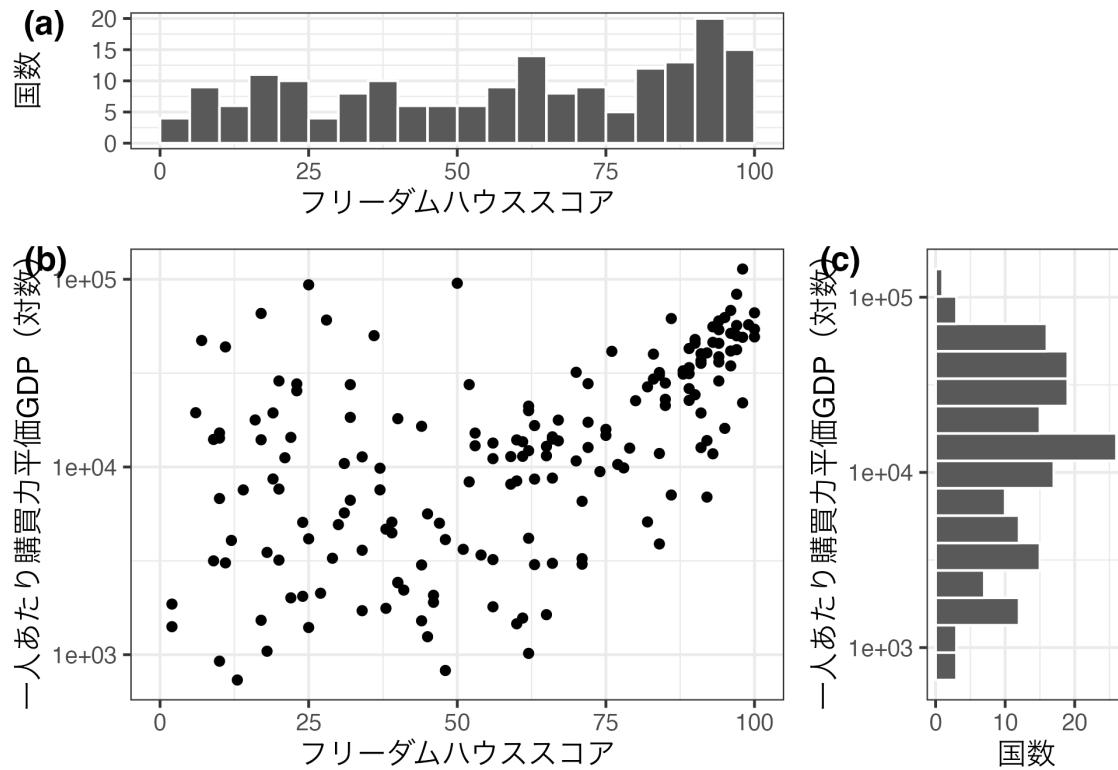
1 ggarrange(Grid_Fig3,
2             NULL,
3             Grid_Fig1,
4             Grid_Fig4,
5             nrow = 2, ncol = 2, align = "hv",

```

```

6      # グリットの大きさを調整
7      widths = c(0.7, 0.3), heights = c(0.3, 0.7),
8      # 各図にラベルを付ける
9      labels = c("(a)", NA, "(b)", "(c)"))

```



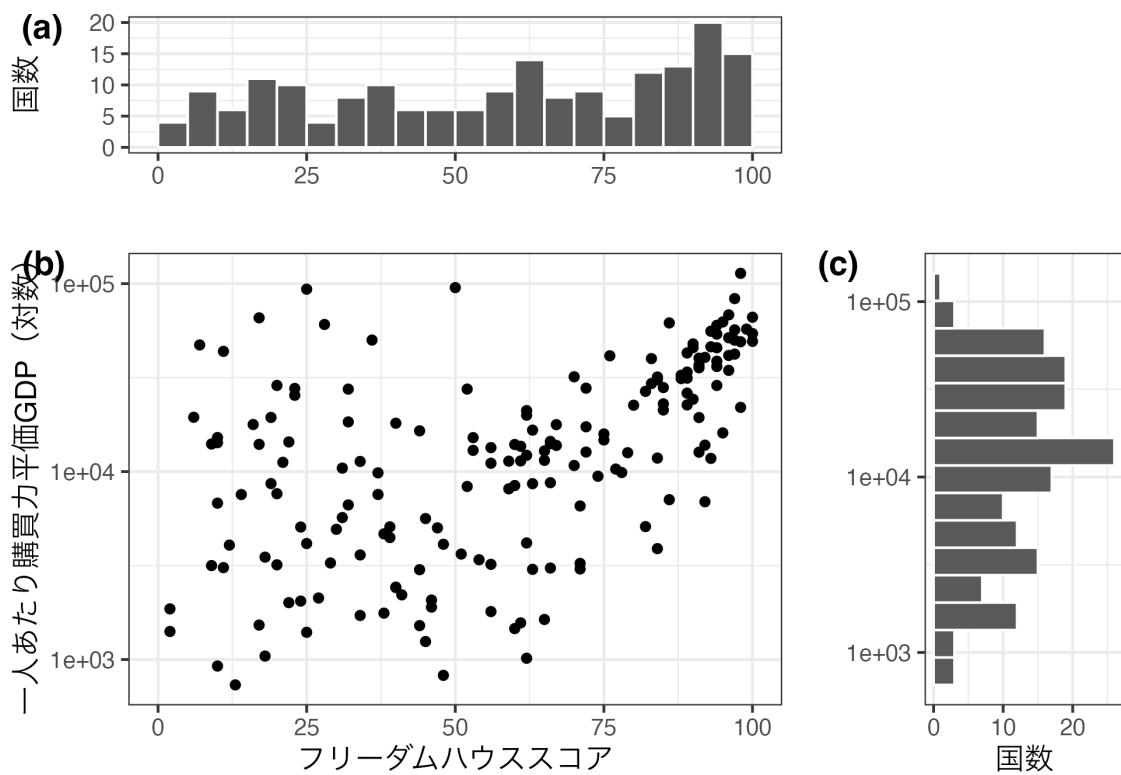
左上の Grid\_Fig3 と左下の Grid\_Fig1 は横軸のラベルを共有しており、左下の Grid\_Fig1 と右下の Grid\_Fig4 は縦軸のラベルを共有しています。以下では Grid\_Fig3 の横軸ラベル、Grid\_Fig4 の縦軸ラベルを消します。ggplot オブジェクトに `theme(axis.title.x = element_blank())` を + で繋ぐと横軸のラベルが表示されなくなります。ggarrange() に入れるオブジェクトに直接アクセスし、それぞれの軸ラベルを消してみましょう<sup>5)</sup>。

5) もし、横軸 (x) の目盛りも消したい場合は `theme()` 内に `axis.ticks.x = element_blank()` を、目盛りのラベルまで消したい場合は `axis.text.x = element_blank()` を追加してください。横軸 (y) の場合は `.x` の箇所を `.y` に書き換えてください。

```

1 ggarrange(Grid_Fig3 + theme(axis.title.x = element_blank()),
2           NULL,
3           Grid_Fig1,
4           Grid_Fig4 + theme(axis.title.y = element_blank()),
5           nrow = 2, ncol = 2, align = "hv",
6           widths = c(0.7, 0.3), heights = c(0.3, 0.7),
7           labels = c("(a)", NA, "(b)", "(c)"))

```



これで完成です。`ggarrange()` には他にもカスタマイズ可能な部分がいっぱいあります。詳細はコンソール上で`?ggarrange` を入力し、ヘルプを参照してください。

他にも`{egg}`パッケージの`ggarrange()`、`{cowplot}`の`plot_grid()`、`{patchwork}`の`+`、`l`、`/`演算子などがあります。それぞれ強みがあり、便利なパッケージです。興味のある読者は以下のページで使い方を確認してみてください。

- `egg`

- cowplot
- patchwork



## 第 20 章

# 可視化 [発展]

### 20.1 概要

第 17 章では{ggplot2}の仕組みについて、第 18 章ではよく使われる 5 種類のプロット（棒グラフ、散布図、折れ線グラフ、箱ひげ図、ヒストグラム）の作り方を、第 19 章ではスケール、座標系などの操作を通じたグラフの見た目調整について解説しました。本章では第 18 章の延長線上に位置づけることができ、紹介しきれなかった様々なグラフの作り方について簡単に解説します。本章で紹介するグラフは以下の通りです。

- バイオリンプロット
- ラグプロット
- リッジプロット
- エラーバー付き散布図
- ロリーポップチャート
- 平滑化ライン
- ヒートマップ<sup>9</sup>
- 等高線図
- 地図
- 非巡回有向グラフ
- バンプチャート
- 沖積図
- ツリーマップ<sup>10</sup>

- モザイクプロット
- 

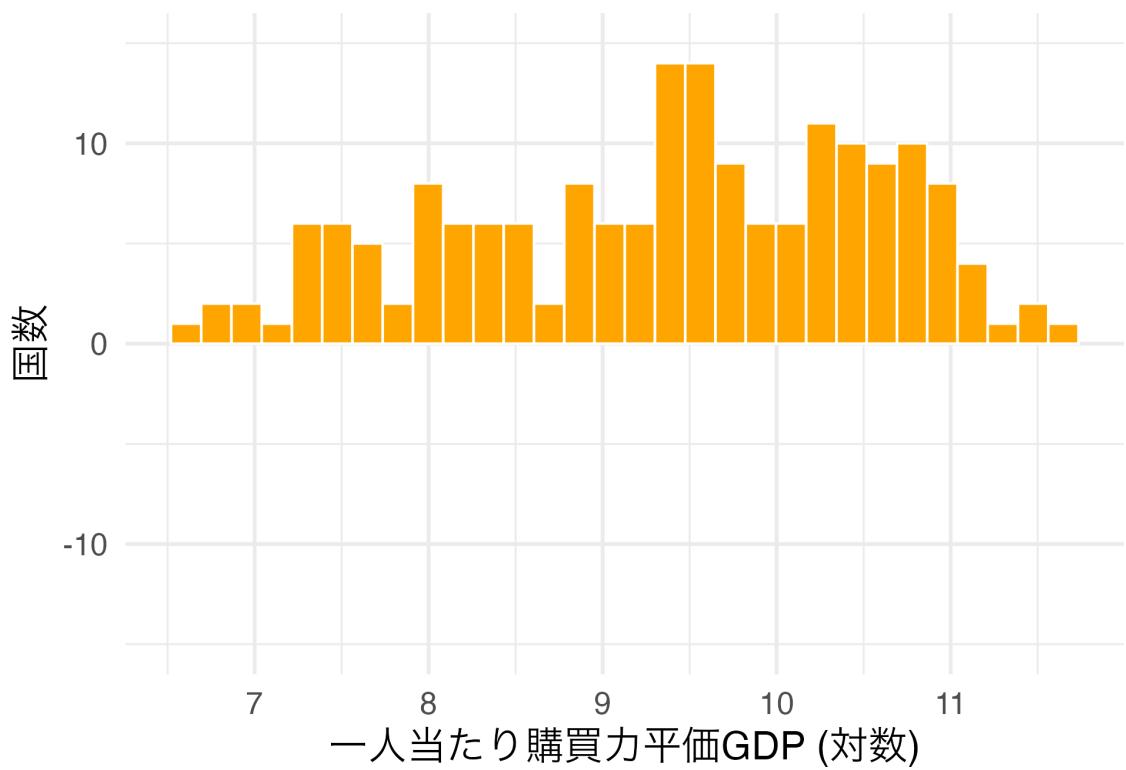
```
1 pacman::p_load(tidyverse)
2
3 Country_df <- read_csv("Data/Countries.csv")
4 COVID19_df <- read_csv("Data/COVID19_Worldwide.csv", guess_max = 10000)
```

---

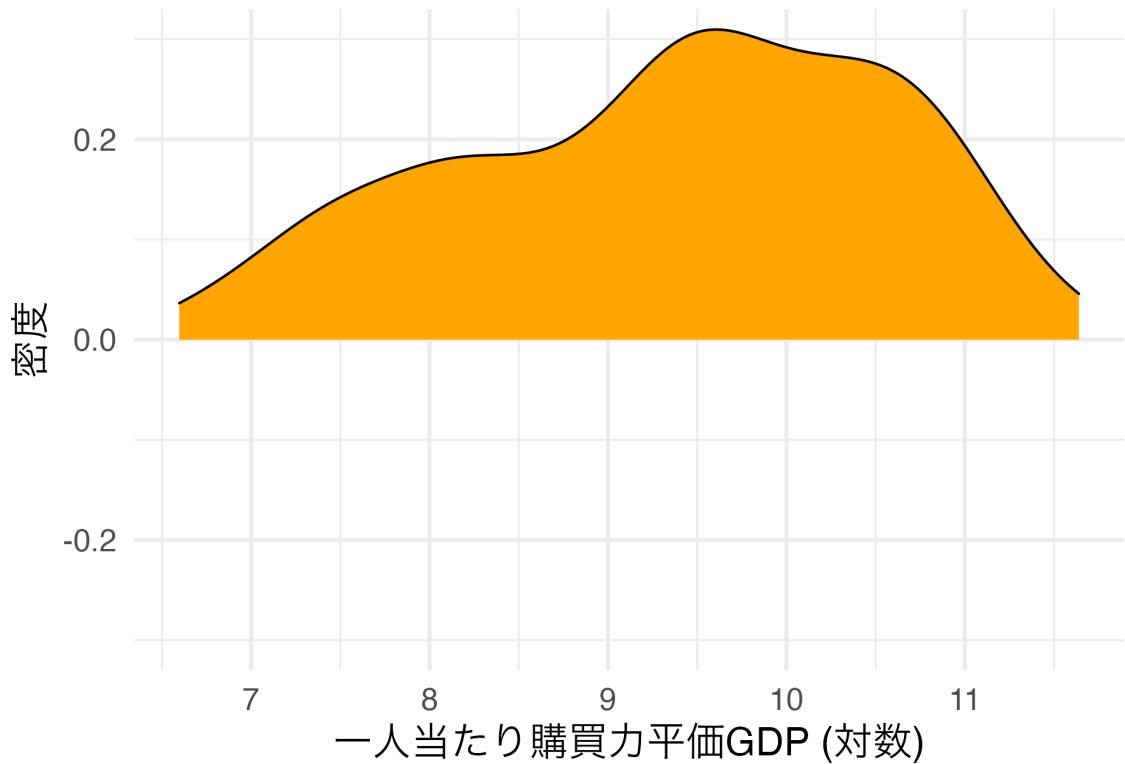
## 20.2 バイオリンプロット

バイオリンプロットは連続変数の分布を可視化する際に使用するプロットの一つです。第 17 章で紹介しましたヒストグラムや箱ひげ図と目的は同じです。それではバイオリンプロットとは何かについて例を見ながら解説します。

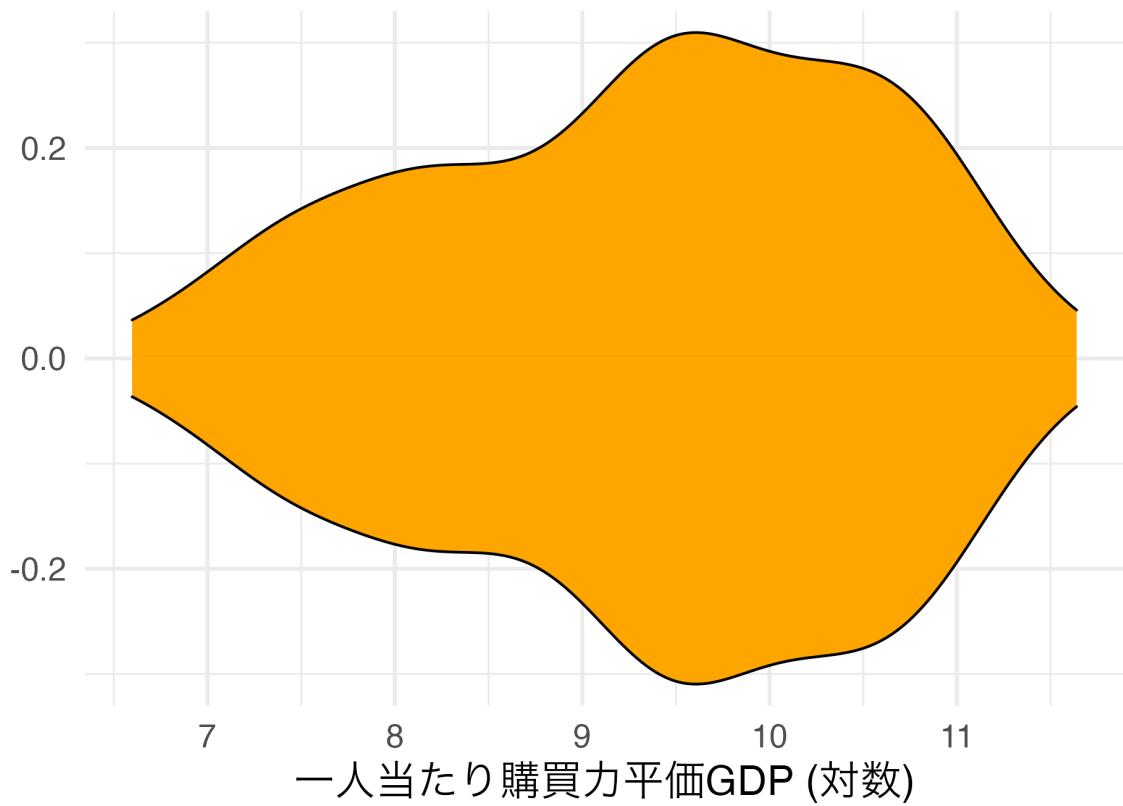
以下の図は対数化した一人当たり購買力平価 GDP (PPP\_per\_capita) のヒストグラムです。



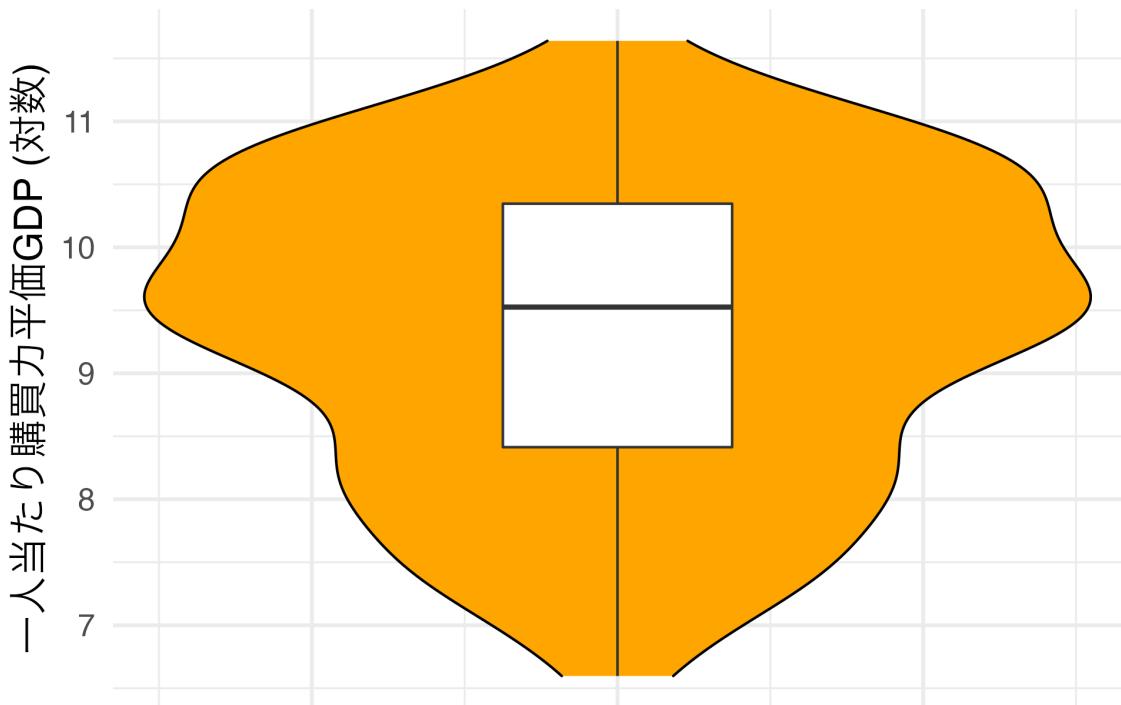
このヒストグラムをなめらかにすると以下の図になります。



この密度曲線を上下対称にすると以下のような図となり、これがバイオリンプロットです。ヒストグラムのようにデータの分布が分かりやすくなります。

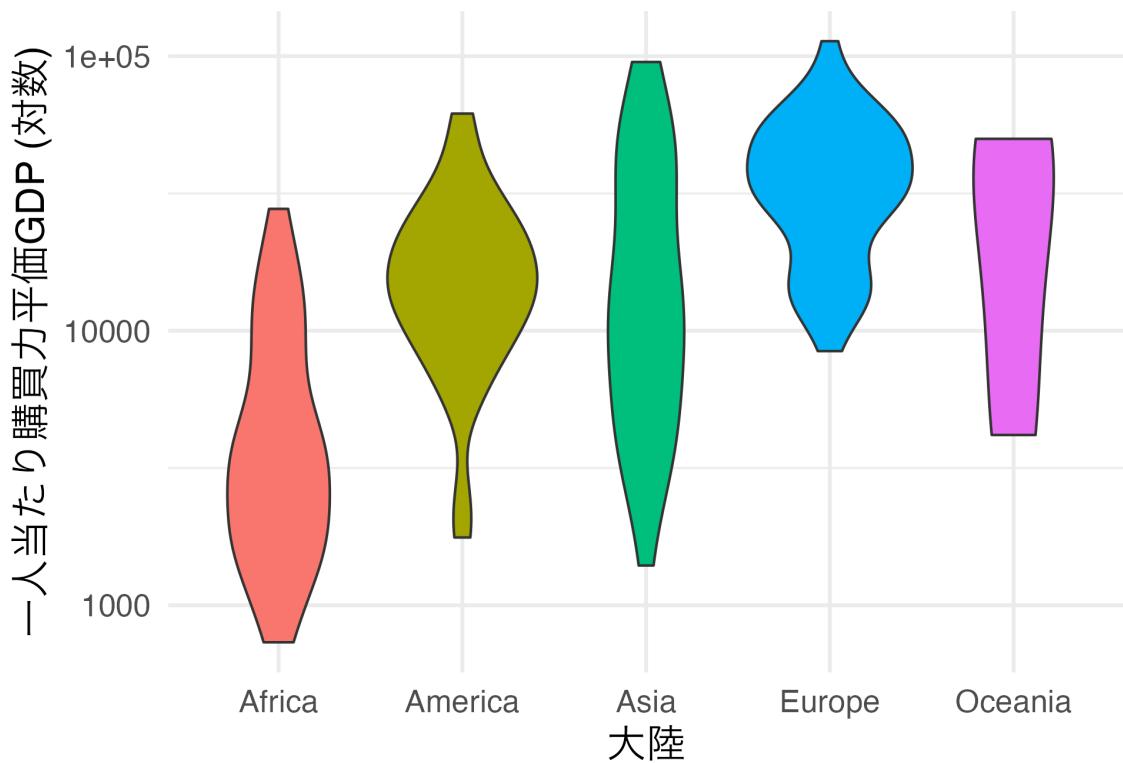


しかし、この図の場合、ヒストグラムと同様、中央値や四分位数などの情報が含まれておません。これらの箱ひげ図を使用した方が良いでしょう。バイオリンプロットの良い点はバイオリンの中に箱ひげ図を入れ、ヒストグラムと箱ひげ図両方の長所を取ることができます。たとえば、バイオリンプロットを 90 度回転させ、中にバイオリン図を入れると以下のようになります。



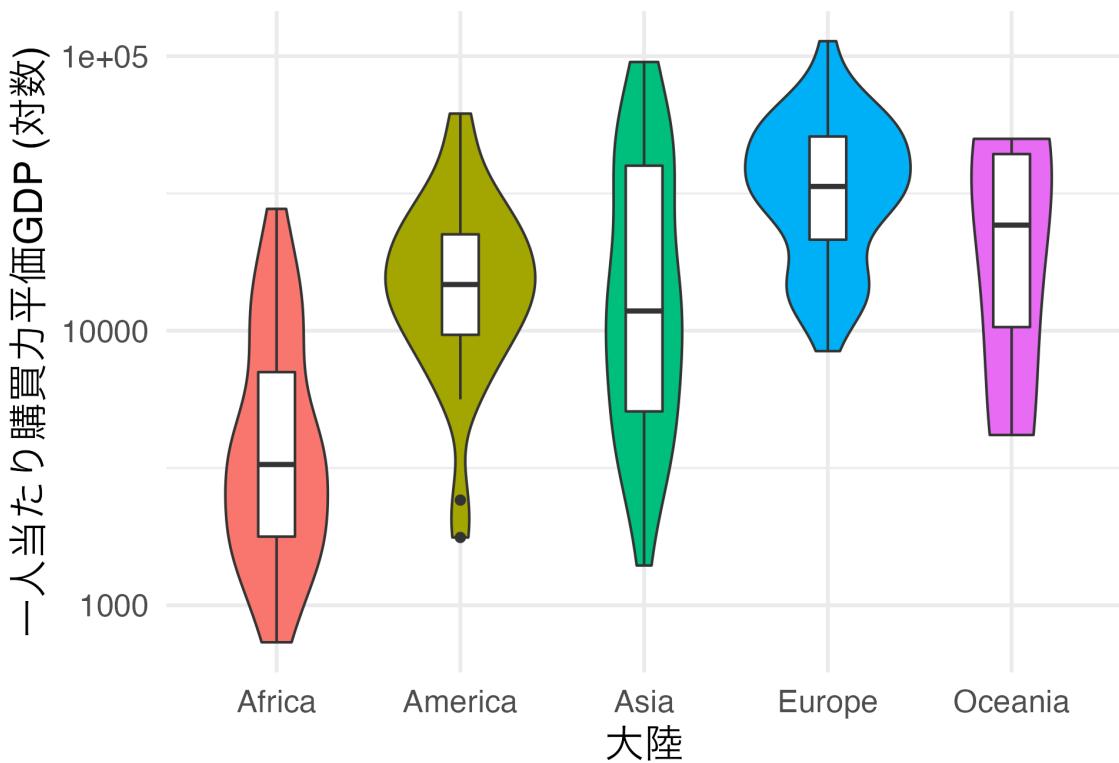
それでは実際にバイオリンプロットを作ってみましょう。使い方は箱ひげ図 (geom\_boxplot()) と同じです。たとえば、横軸は大陸 (Continent) に、縦軸は対数化した一人当たり購買力平価 GDP (PPP\_per\_capita) にしたバイオリンプロットを作るには作るには geom\_violin() 幾何オブジェクトの中にマッピングするだけです。大陸ごとに色分けしたい場合は fill 引数に Continent をマッピングします。

```
1 Country_df %>%
2   ggplot() +
3   geom_violin(aes(x = Continent, y = PPP_per_capita, fill = Continent)) +
4   labs(x = "大陸", y = "一人当たり購買力平価 GDP (対数)") +
5   scale_y_continuous(breaks = c(0, 1000, 10000, 100000),
6                      labels = c(0, 1000, 10000, 100000),
7                      trans = "log10") + # y 軸を対数化
8   guides(fill = "none") + # fill のマッピング情報の凡例を隠す
9   theme_minimal(base_size = 16)
```



ここに箱ひげ図も載せたい場合は、`geom_violin()` オブジェクトの後に `geom_boxplot()` オブジェクトを入れるだけで十分です。

```
1 Country_df %>%
2   ggplot() +
3   geom_violin(aes(x = Continent, y = PPP_per_capita, fill = Continent)) +
4   geom_boxplot(aes(x = Continent, y = PPP_per_capita),
5                 width = 0.2) +
6   labs(x = "大陸", y = "一人当たり購買力平価 GDP (対数)") +
7   scale_y_continuous(breaks = c(0, 1000, 10000, 100000),
8                      labels = c(0, 1000, 10000, 100000),
9                      trans = "log10") +
10  guides(fill = "none") +
11  theme_minimal(base_size = 16)
```

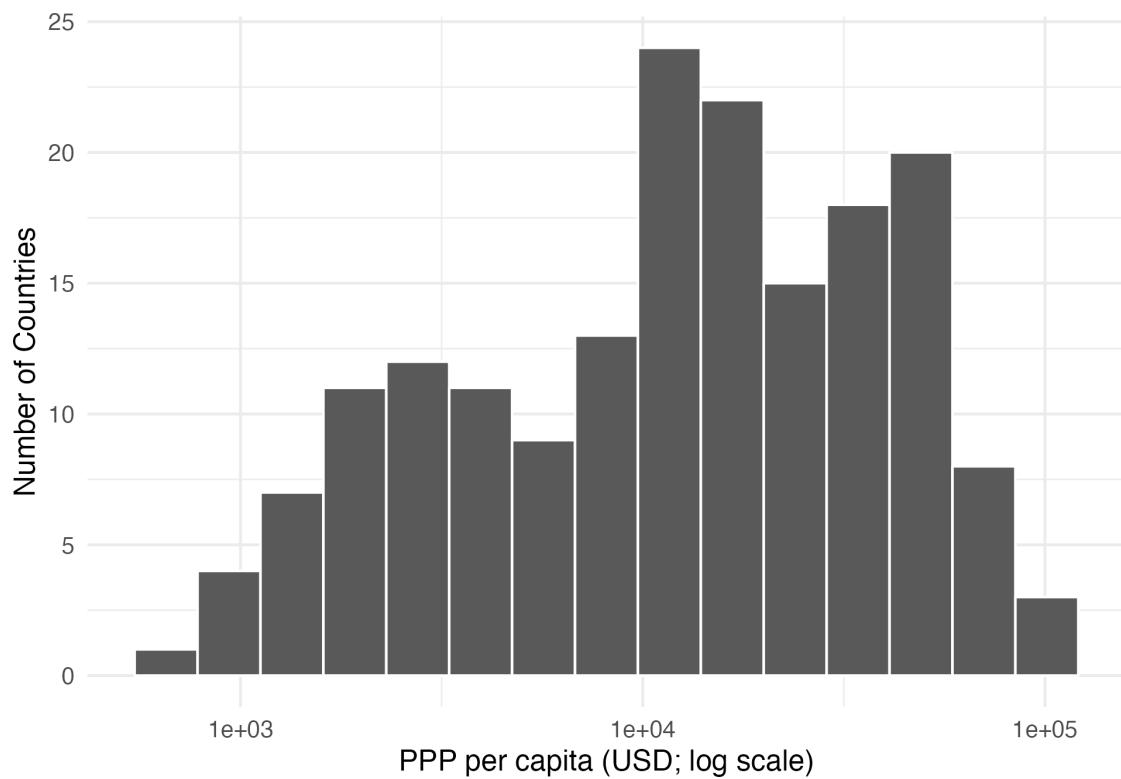


箱ひげ図は四分位範囲、四分位数、最小値、最大値などの情報を素早く読み取れますが、どの値当たりが分厚いかなどの情報が欠けています。これをバイオリンプロットで捕うことで、よりデータの分布を的確に把握することができます。

### 20.3 ラグプロット

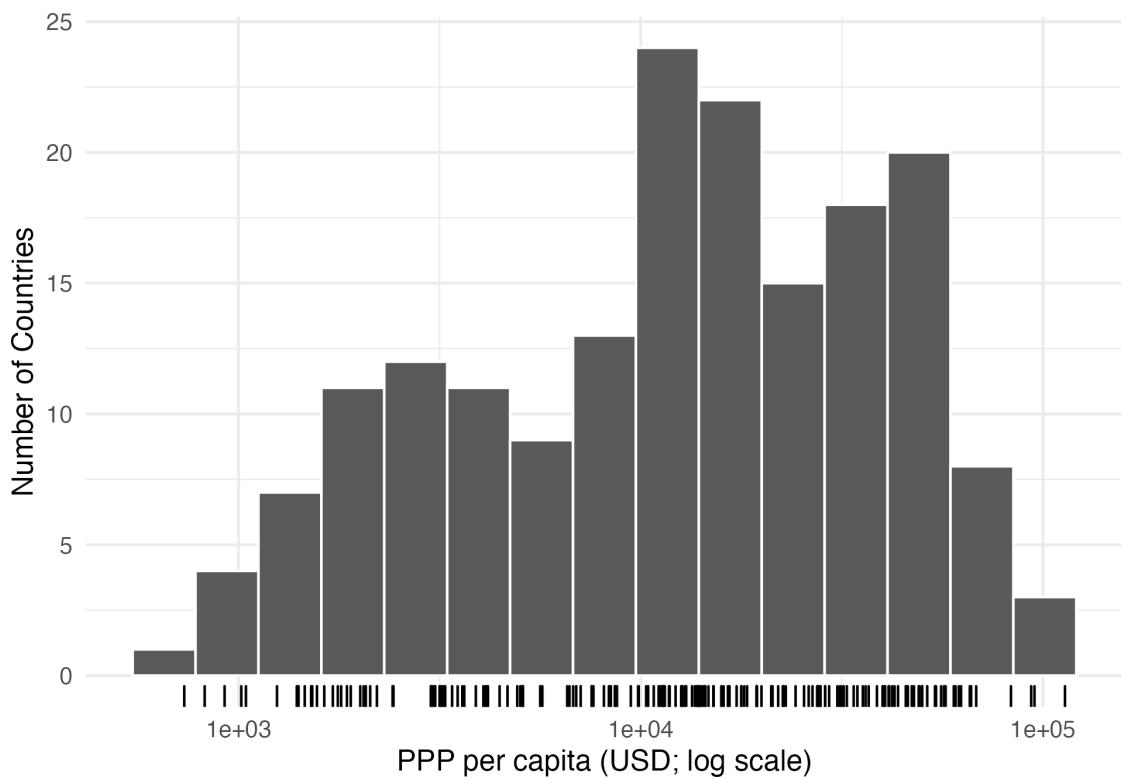
ラグプロット (rug plot) は変数の分布を示す点ではヒストグラム、箱ひげ図、バイオリンプロットと同じ目的を持ちますが、大きな違いとしてはラグプロット単体で使われるケースがない（または、非常に稀）という点です。ラグプロットは上述しましたヒストグラムや箱ひげ図、または散布図などと組み合わせて使うのが一般的です。

以下は `Country_df` の `PPP_per_capita` (常用対数変換) のヒストグラムです。



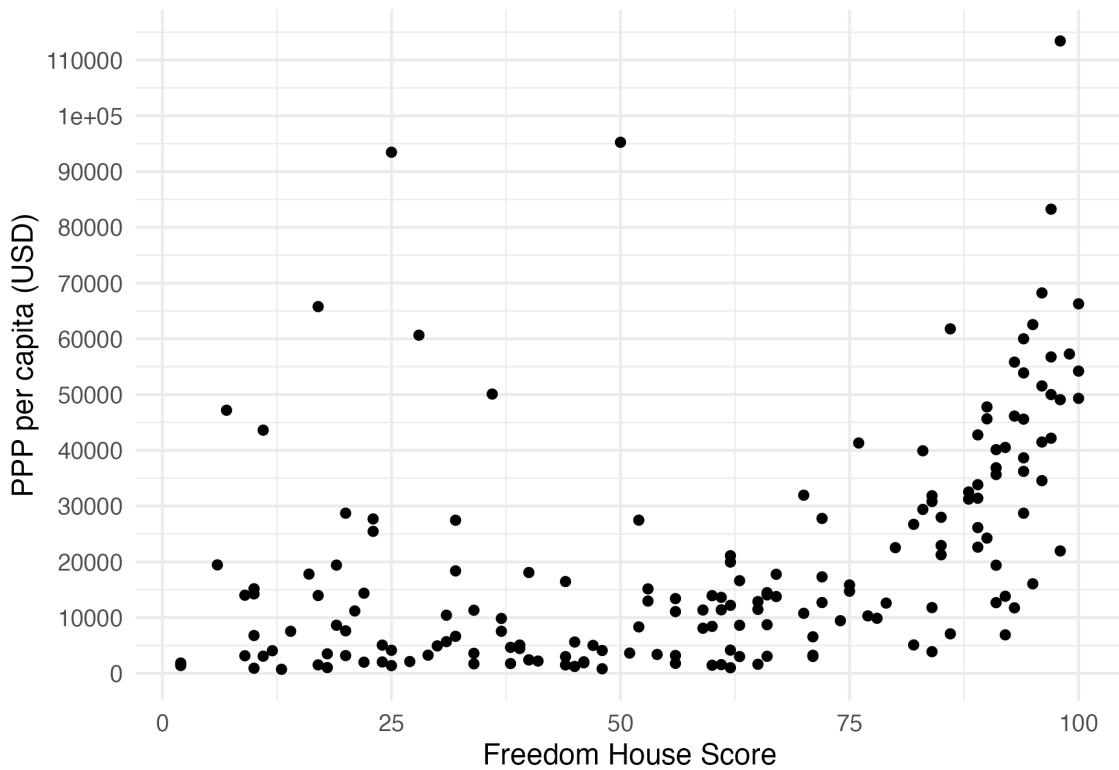
一変数の分布を確認する場合、ヒストグラムは情報量の損失が少ない方です。それでも値一つ一つの情報は失われますね。例えば、上記のヒストグラムで左端の度数は 1 です。左端の棒の区間はおおよそ 500 から 780 であり、一人当たり PPP がこの区間に属する国は 1 カ国ということです。ちなみに、その国はブルンジ共和国ですが、ブルンジ共和国の具体的な一人当たり PPP はヒストグラムから分かりません。情報量をより豊富に持たせるためには区間を細かく刻むことも出来ますが、逆に分布の全体像が読みにくくなります。

ここで登場するのがラグプロットです。これは座標平面の端を使ってデータを一時現状に並べたものです。多くの場合、点ではなく、垂直線（|）を使います。ラグプロットの幾何オブジェクトは{ggplot2}でデフォルトで提供されており、`geom_rug()`を使います。マッピングは `x` または `y` に対して行いますが、座標平面の下段にラグプロットを出力する場合は `x` に変数（ここでは `PPP_per_capita`）をマッピングします。



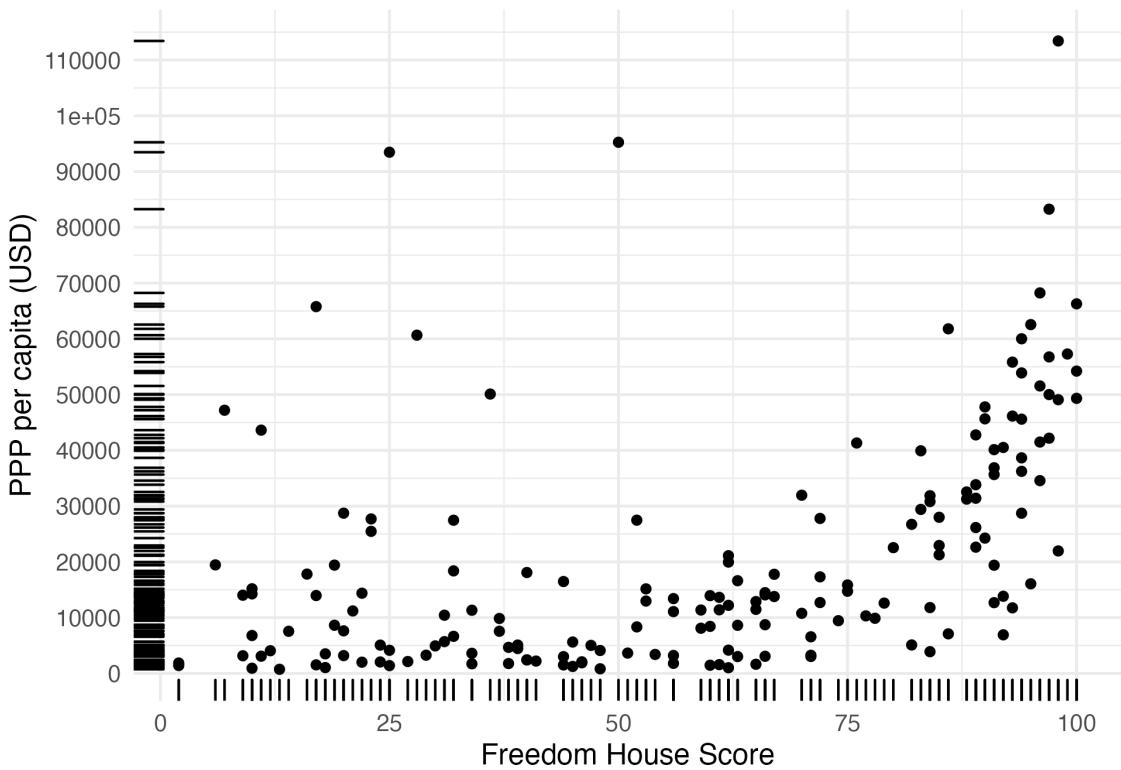
ラグプロットを使うと本来のヒストグラムの外見にほぼ影響を与えず、更に情報を付け加えることが可能です。点（|）の密度でデータの分布を確認することができますが、その密度の相対的な比較に関してはヒストグラムの方が良いでしょう。

ラグプロットは散布図に使うことも可能です。散布図は一つ一つの点が具体的な値がマッピングされるため、情報量の損失はほぼないでしょう。それでも散布図にラグプロットを加える意味はあります。まず、Country\_df のフリーダムハウス指数 (FH\_Total) と一人当たり PPP (PPP\_per\_capita) の散布図を作ってみましょう。



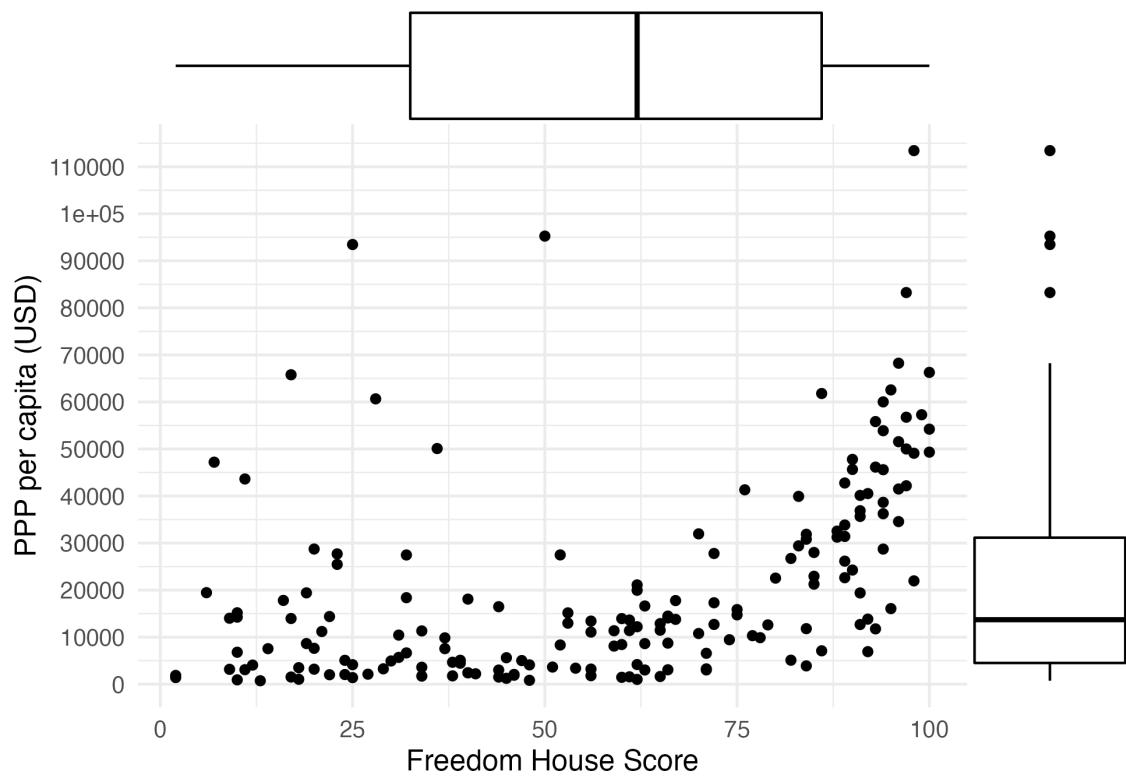
散布図の目的は二変量間の関係を確認することであって、それぞれの変数の分布を確認することではありません。もし、FH\_Total と PPP\_per\_capita の分布が確認したいなら、それぞれのヒストグラムや箱ひげ図を作成した方が良いでしょう。しかし、ラグプロットを使えば、点（|）の密度で大まかな分布は確認出来ますし、図の見た目にもほぼ影響を与えません。

横軸と縦軸両方のラグプロットは、`geom_rug()` に x と y 両方マッピングするだけです。



ここで FH\_Total はほぼ均等に分布していて、PPP\_per\_capita は 2 万ドル以下に多く密集していることが確認できます。

{ggExtra}の ggMarginal() を使えば、ラグプロットでなく、箱ひげ図やヒストグラムを付けることも可能です。{ggplot2}で作図した図をオブジェクトとして格納し、ggMarginal() の第一引数として指定します。第一引数のみだと密度のだけ出力されるため、箱ひげ図を付けるためには type = "boxplot"を指定します（既定値は"density"）。ヒストグラムを出力する場合は"histogram"と指定します。



## 20.4 リッジプロット

リッジプロット (ridge plot) はある変数の分布をグループごとに出力する図です。大陸ごとの人間開発指数の分布を示したり、時系列データなら分布の変化を示す時にも使えます。ここでは大陸ごとの人間開発指数の分布をリッジプロットで示してみましょう。

リッジプロットを作成する前に、`geom_density()` 幾何オブジェクトを用い、変数の密度曲線 (density curve) を作ってみます。マッピングは `x` に対し、分布を出力する変数名を指定します。また、密度曲線内部に色塗り (`fill`) をし、曲線を計算する際のバンド幅 (`bw`) は 0.054 にします。`bw` が大きいほど、なめらかな曲線になります。

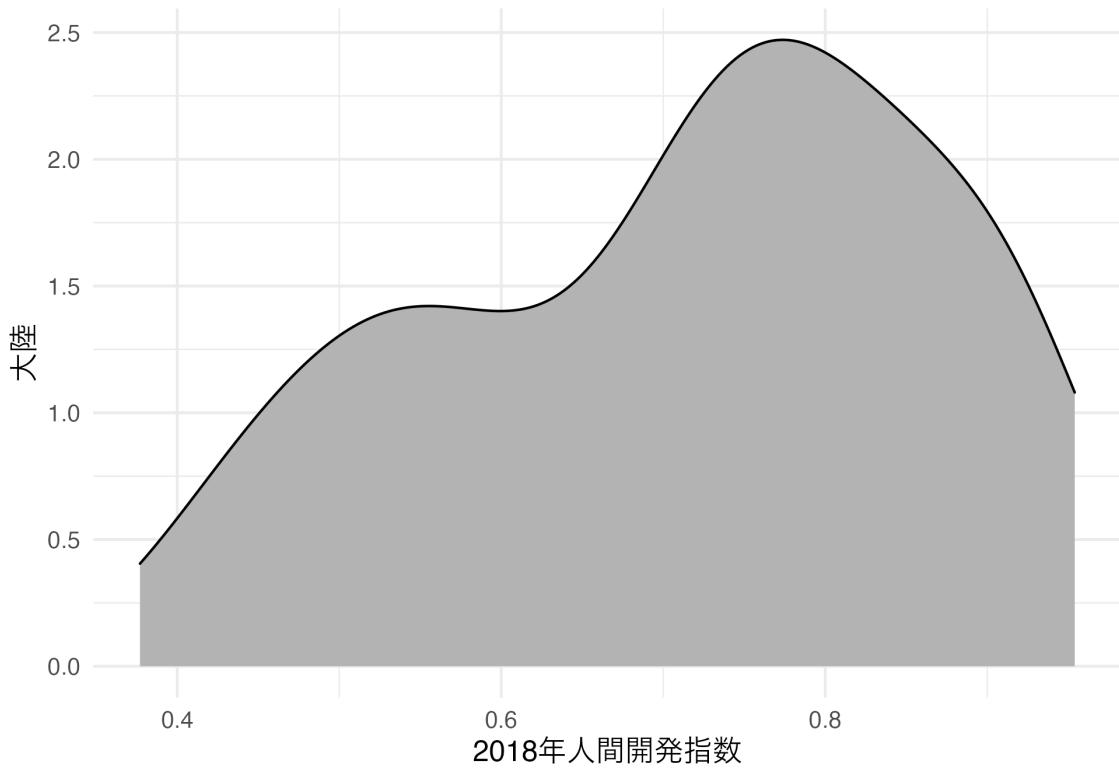
```

1 Country_df %>%
2   ggplot() +
3   geom_density(aes(x = HDI_2018), fill = "gray70", bw = 0.054) +
4   labs(x = "2018 年人間開発指数", y = "大陸") +

```

```
5   theme_minimal(base_size = 12)

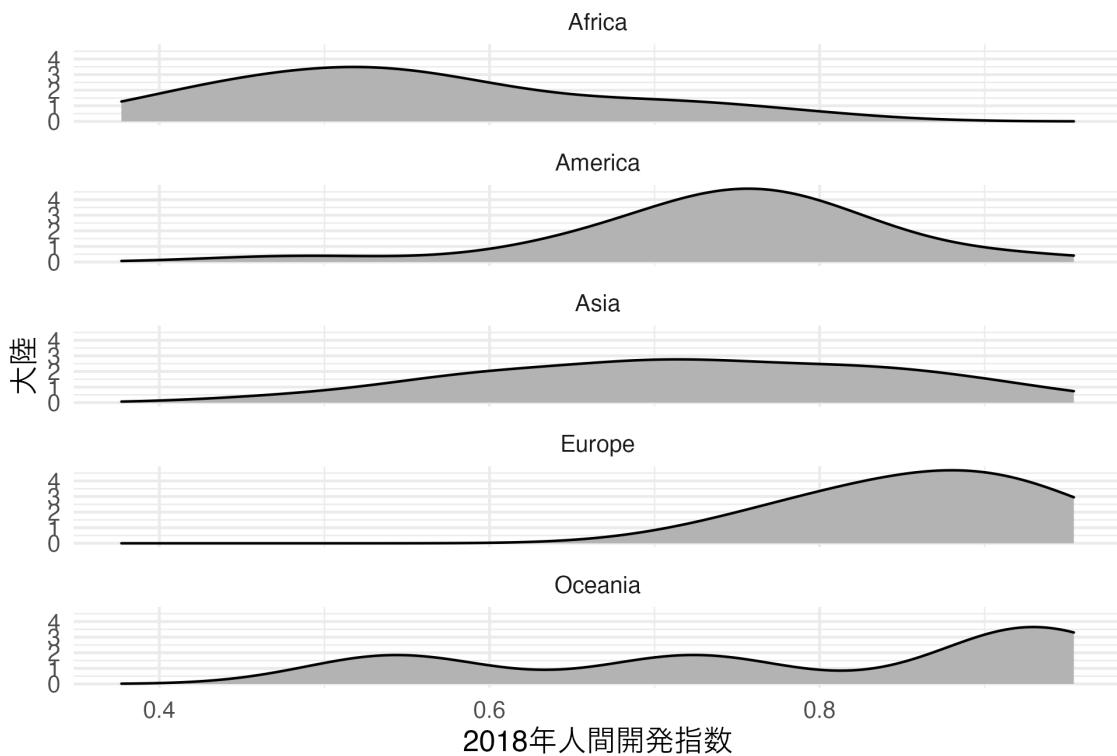
## Warning: Removed 6 rows containing non-finite values (stat_density).
```



これを大陸ごとに出力する場合、ファセット分割を行います。今回は大陸ごとに1列 (ncol = 1) でファセットを分割します。

```
1 Country_df %>%
2   ggplot() +
3   geom_density(aes(x = HDI_2018), fill = "gray70", bw = 0.054) +
4   labs(x = "2018年人間開発指数", y = "大陸") +
5   facet_wrap(~Continent, ncol = 1) +
6   theme_minimal(base_size = 12)

## Warning: Removed 6 rows containing non-finite values (stat_density).
```



それでは上のグラフをリッジプロットとして作図してみましょう。今回は{ggridges}パッケージを使います。

```
1 pacman::p_load(ggridges)
```

使用する幾何オブジェクトは `geom_density_ridges()` です。似たような幾何オブジェクトとして `geom_ridgeline()` がありますが、こちらは予め密度曲線の高さを計算しておく必要があります。一方、`geom_density_ridges()` は変数だけ指定すれば密度を自動的に計算してくれます。マッピングは `x` と `y` に対し、それぞれ分布を出力する変数名とグループ変数名を指定します。また、密度曲線が重なるケースもあるため、透明度 (`alpha`) も 0.5にしておきましょう。ここでは別途指定しませんが、ハンド幅も指定可能であり、`aes()` の外側に `bandwidth` を指定するだけです。

```
1 Country_df %>%
2   ggplot() +
3   geom_density_ridges(aes(x = HDI_2018, y = Continent),
4                       alpha = 0.5) +
5   labs(x = "2018 年人間開発指數", y = "大陸") +
```

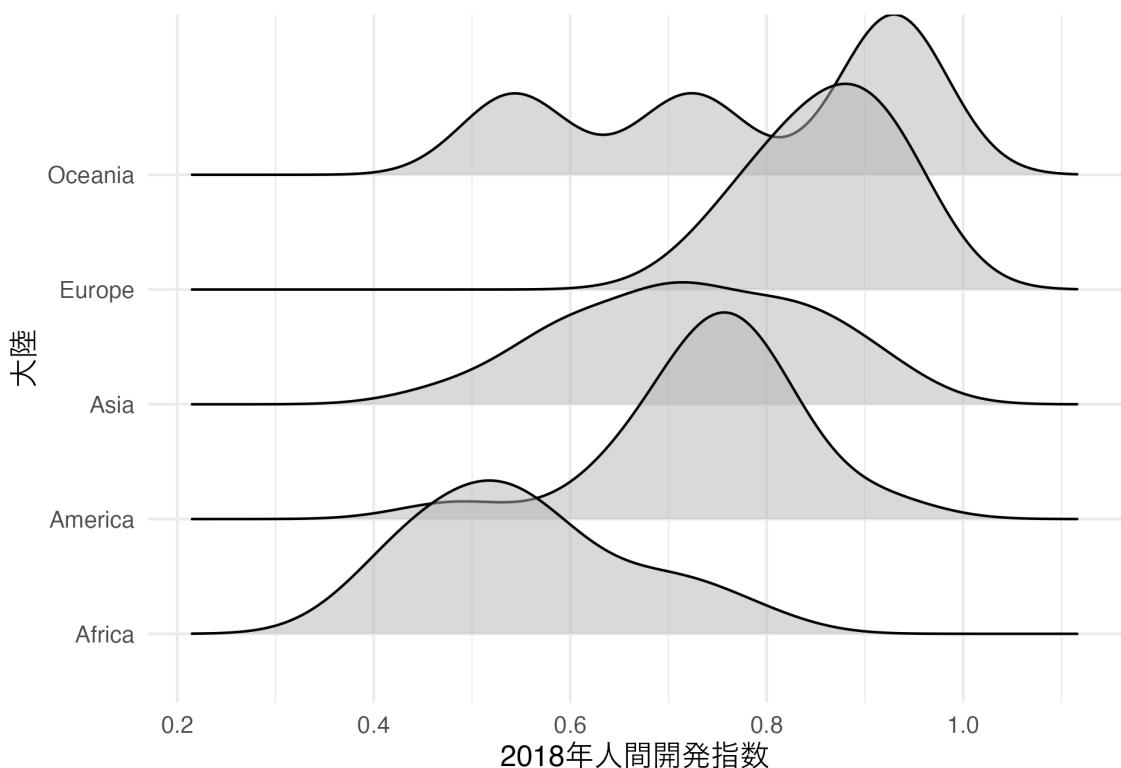
```

6   theme_minimal(base_size = 12)

## Picking joint bandwidth of 0.054

## Warning: Removed 6 rows containing non-finite values (stat_density_ridges).

```



先ほど作図した図と非常に似た図が出来上りました。ファセット分割に比べ、空間を最大限に活用していることが分かります。ファセットラベルがなく、グループ名が縦軸上に位置するからです。また、リッジプロットの特徴は密度曲線がオーバラップする点ですが、以下のように `scale = 1` を指定すると、オーバラップなしで作成することも可能です。もし、`scale = 3` にすると最大2つの密度曲線が重なることになります。たとえば最下段のアフリカはアメリカの行と若干オーバラップしていますが、`scale = 3` の場合、アジアの行までオーバーラップされうることになります。

```

1 Country_df %>%
2   ggplot() +

```

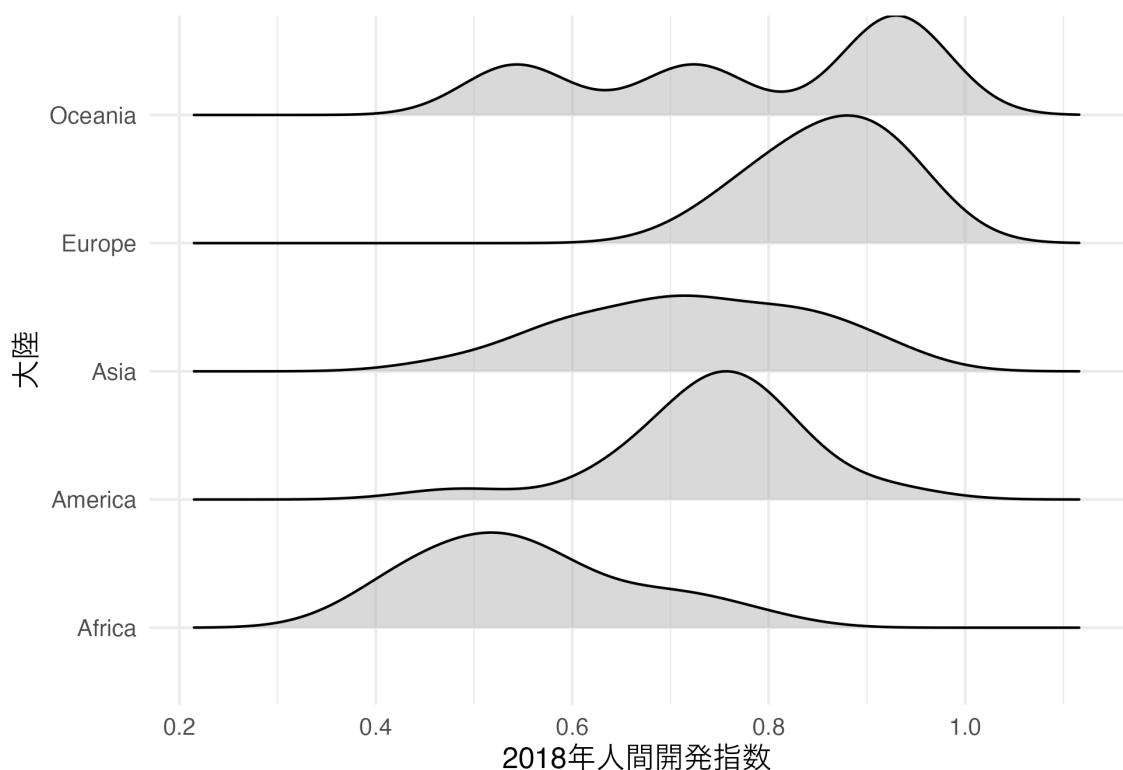
```

3   geom_density_ridges(aes(x = HDI_2018, y = Continent),
4                         scale = 1, alpha = 0.5) +
5   labs(x = "2018 年人間開発指数", y = "大陸") +
6   theme_minimal(base_size = 12)

## Picking joint bandwidth of 0.054

## Warning: Removed 6 rows containing non-finite values (stat_density_ridges).

```



また、横軸の値に応じて背景の色をグラデーションで表現することも可能です。この場合、`geom_density_ridges()` 幾何オブジェクトでなく、`geom_density_ridges_gradient()` を使い、`fill` にもマッピングをする必要があります。横軸（x）の値に応じて色塗りをする場合、`fill = stat(x)` とします。デフォルトでは横軸の値が高いほど空色、低いほど黒になります。ここでは高いほど黄色、低いほど紫ににするため、色弱にも優しい `scale_fill_viridis_c()` を使い<sup>1)</sup>、カラー

<sup>1)</sup> 最後の\_cはマッピングする変数（ここでは HDI\_2018）が連続変数（continuous）であることを意味し

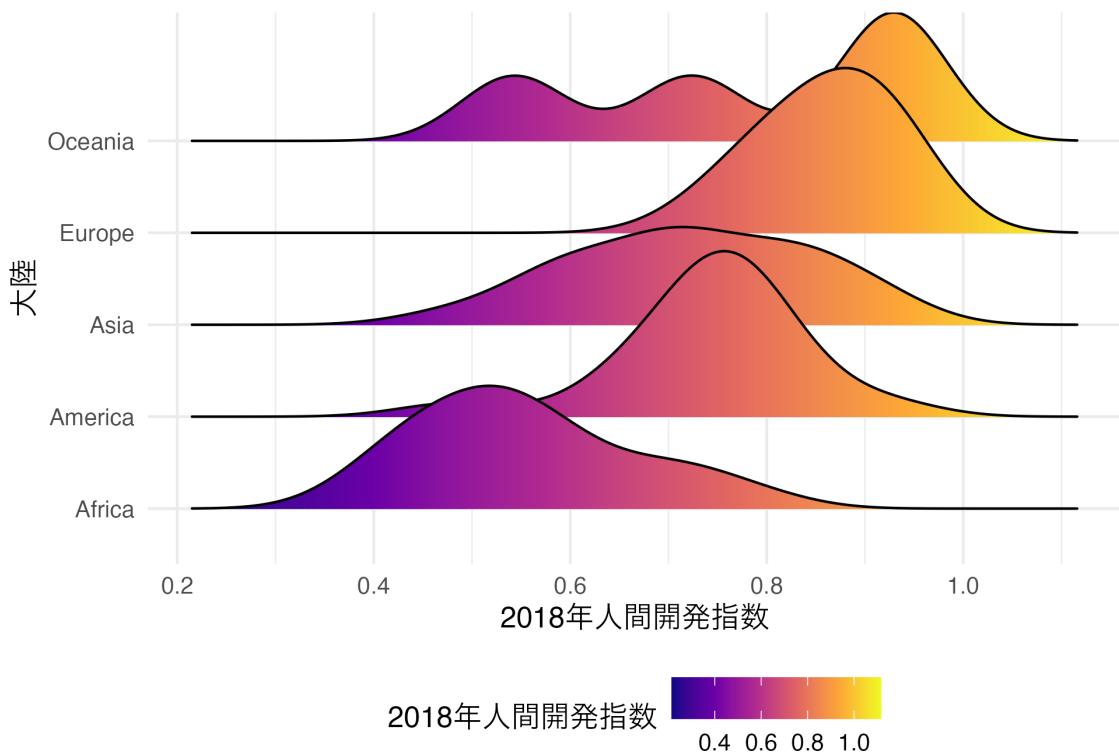
オプションは plasma にします (option = "c")。

```

1 Country_df %>%
2   ggplot() +
3   geom_density_ridges_gradient(aes(x = HDI_2018, y = Continent, fill = stat(x)),
4                                alpha = 0.5) +
5   scale_fill_viridis_c(option = "C") +
6   labs(x = "2018 年人間開発指数", y = "大陸", fill = "2018 年人間開発指数") +
7   theme_minimal(base_size = 12) +
8   theme(legend.position = "bottom")

```

## Picking joint bandwidth of 0.054



密度曲線は基本的にはなめらかな曲線であるため、データが存在しない箇所にも密度が高く見積もられるケースがあります。全体的な分布を俯瞰するには良いですが、情報の

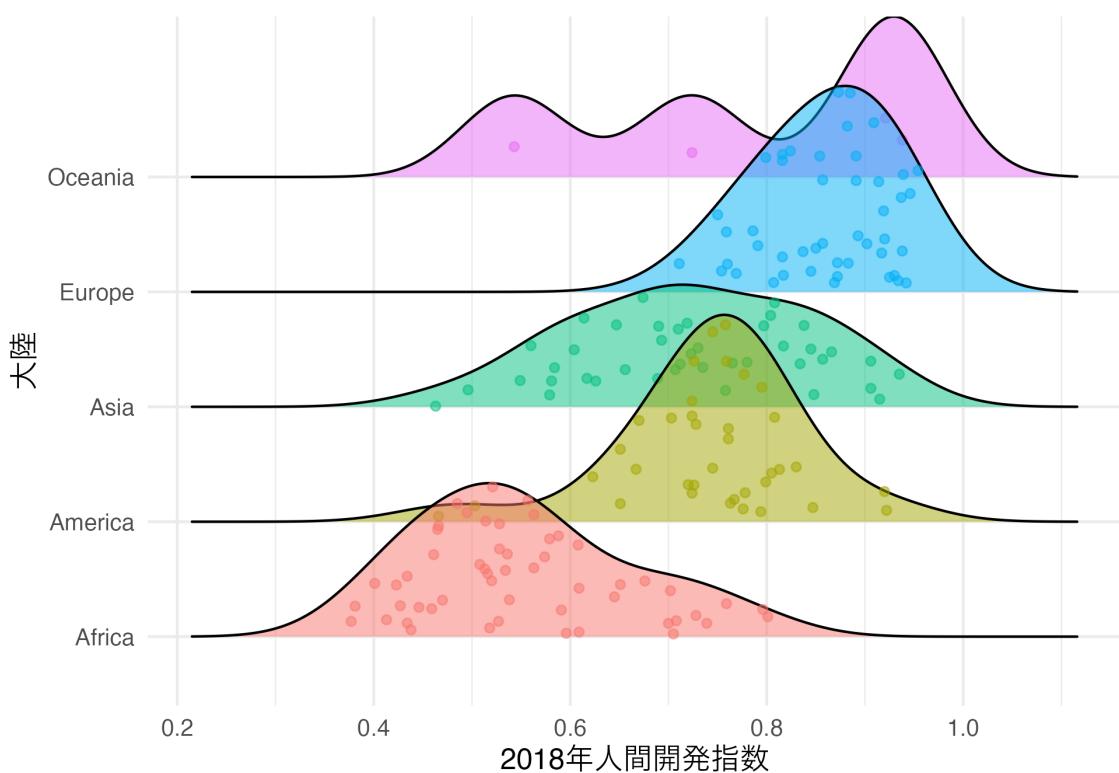
ます。

損失は避けられません。そこで出てくるのが点付きのリッジプロットです。HDI\_2018 の個々の値を点で出力するには `jittered_points = TRUE` を指定するだけです。これだけで密度曲線の内側に点が若干のズレ付き (jitter) で出力されます。ただし、密度曲線がオーバーラップされるリッジプロットの特徴を考えると、グループごとに点の色分けをする必要があります (同じ色になると、どのグループの点かが分からなくなるので)。この場合、`point_color` に対し、グループ変数 (Continent) をマッピングします。また、密度曲線の色と合わせるために密度曲線の色塗りも `fill` で指定します。

```
1 Country_df %>%
2   ggplot() +
3   geom_density_ridges(aes(x = HDI_2018, y = Continent, fill = Continent,
4                         point_color = Continent),
5                     alpha = 0.5, jittered_points = TRUE) +
6   guides(fill = "none", point_color = "none") + # 凡例を削除
7   labs(x = "2018 年人間開発指数", y = "大陸") +
8   theme_minimal(base_size = 12)

## Picking joint bandwidth of 0.054

## Warning: Removed 6 rows containing non-finite values (stat_density_ridges).
```



他にも密度曲線の下側にラグプロットを付けることも可能です。こうすれば点ごとに色を付ける必要もなくなります。ラグプロットを付けるためには点の形 (point\_shape) を「|」にする必要があります。ただ、これだけだと「|」が密度曲線内部に散らばる (jittered) だけです。散らばりをなくす、つまり密度曲線の下段に固定する必要があり、これは aes() その外側に position = position\_points\_jitter(width = 0, height = 0) を指定することで出来ます。

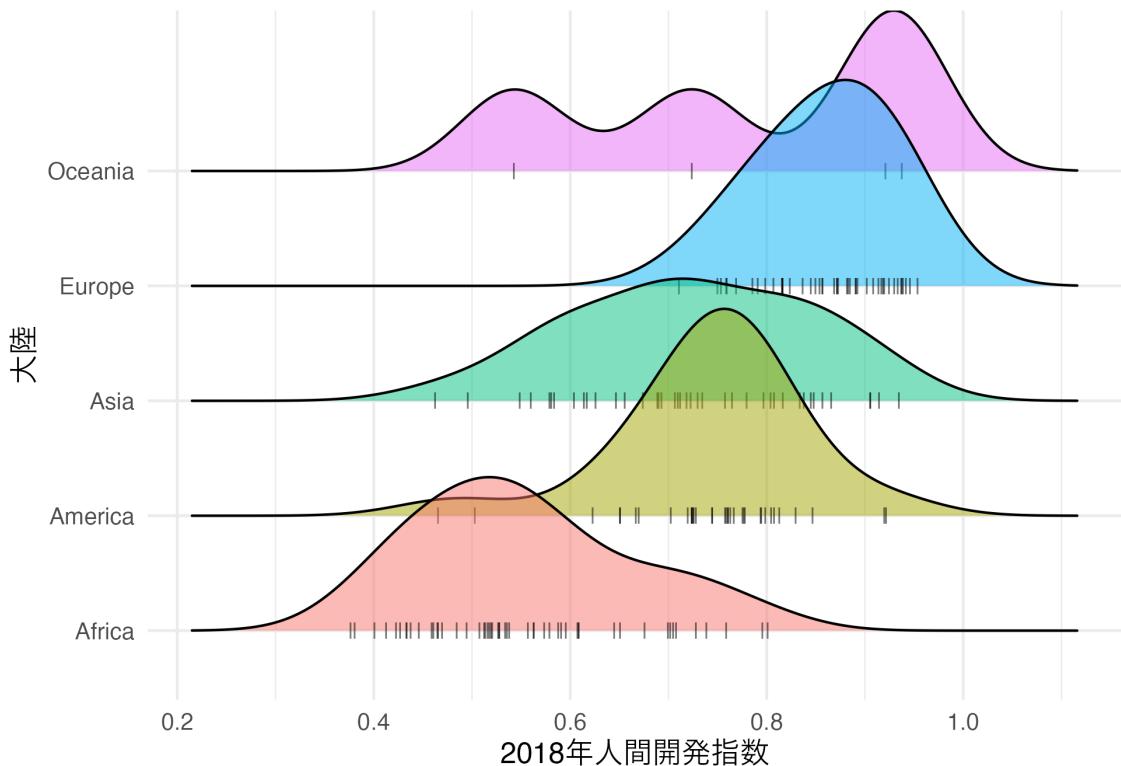
```

1 Country_df %>%
2   ggplot() +
3   geom_density_ridges(aes(x = HDI_2018, y = Continent, fill = Continent),
4                       alpha = 0.5, jittered_points = TRUE,
5                       position = position_points_jitter(width = 0, height = 0),
6                       point_shape = "|", point_size = 3) +
7   guides(fill = "none") +
8   labs(x = "2018 年人間開発指数", y = "大陸") +
9   theme_minimal(base_size = 12)

```

```
## Picking joint bandwidth of 0.054

## Warning: Removed 6 rows containing non-finite values (stat_density_ridges).
```

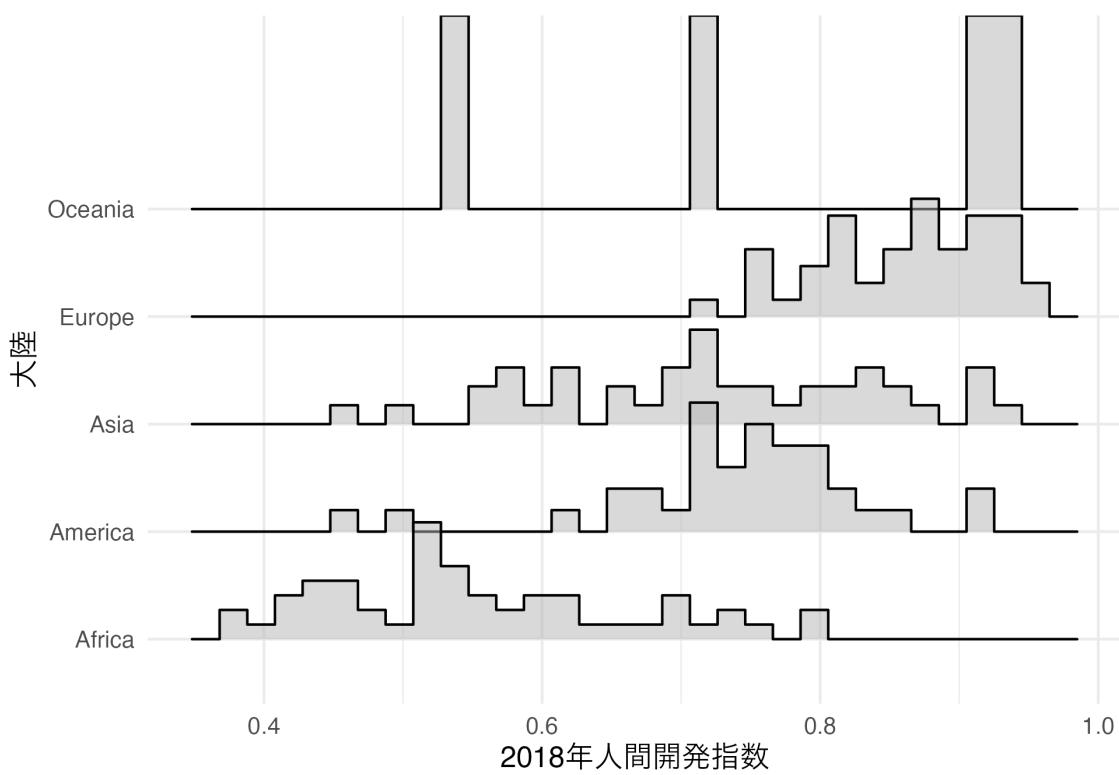


最後に密度曲線でなく、ヒストグラムで示す方法を紹介します。これは `geom_density_ridges()` の内部に `stat = "binline"` を指定するだけです。

```
1 Country_df %>%
2   ggplot() +
3   geom_density_ridges(aes(x = HDI_2018, y = Continent), alpha = 0.5,
4                       stat = "binline") +
5   labs(x = "2018 年人間開発指数", y = "大陸") +
6   theme_minimal(base_size = 12)

## `stat_binline()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 6 rows containing non-finite values (stat_binline).
```



## 20.5 エラーバー付き散布図

エラーバー付きの散布図は推定結果の点推定値とその不確実性（信頼区間など）を示す際によく使われる図です。以下の表は `Country_df` を用い、大陸（オセアニアを除く）ごとにフリーダムハウス・スコア (`FH_Total`) を一人当たり PPP GDP (`PPP_per_capita`) に回帰させた分析から得られたフリーダムハウス・スコア (`FH_Total`) の係数（以下の式の  $\beta_1$ ）の点推定値と 95% 信頼区間です。

$$\text{PPP per capita} = \beta_0 + \beta_1 \cdot \text{FH\_Total} + \varepsilon$$

```
Pointrange_df <- tibble(
  Continent = c("Asia", "Europe", "Africa", "America"),
  Coef      = c(65.3, 588.0, 53.4, 316.0),
```

```

Conf_lwr  = c(-250.0, 376.0, -14.5, 128.0),
Conf_upr  = c(380.0, 801.0, 121.0, 504.0)
)

```

1 Pointrange\_df

```

## # A tibble: 4 x 4
## # Groups:   Continent [4]
##   Continent   Coef Conf_lwr Conf_upr
##   <chr>     <dbl>    <dbl>    <dbl>
## 1 Asia       65.3    -250.     380.
## 2 Europe     588.     376.     801.
## 3 Africa     53.4    -14.5    121.
## 4 America    316.     128.     504.

```

実は以上のデータは以下のようないくつかのコードで作成されています。{purrr}パッケージの使い方に慣れる必要があるので、第 27 章を参照してください。

```

Pointrange_df <- Country_df %>%
  filter(Continent != "Oceania") %>%
  group_by(Continent) %>%
  nest() %>%
  mutate(Fit = map(data, ~lm(PPP_per_capita ~ FH_Total, data = .)),
        Est = map(Fit, broom::tidy, conf.int = TRUE)) %>%
  unnest(Est) %>%
  filter(term == "FH_Total") %>%
  select(Continent, Coef = estimate,
        Conf_lwr = conf.low, Conf_upr = conf.high)

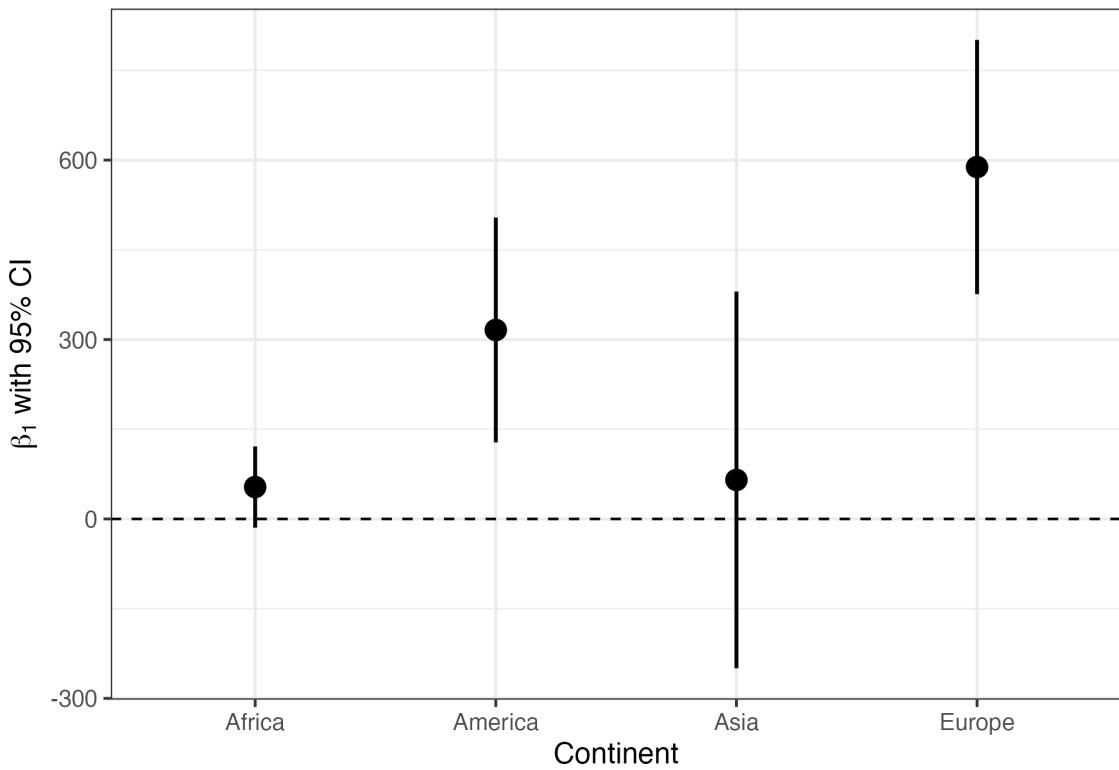
```

この Pointrange\_df を用いて横軸は大陸 (Continent)、縦軸には点推定値 (Coef) と 95% 信頼区間 (Conf\_lwr と Conf\_upr) を出力します。ここで使う幾何オブジェクトは geom\_pointrange() です。横軸 x と点推定値 y、95% 信頼区間の下限の ymin、上限の ymax にマッピングします。エラーバー付き散布図を立てに並べたい場合は y と x、xmin、xmax にマッピングします。

```

1 Pointrange_df %>%
2   ggplot() +
3   geom_hline(yintercept = 0, linetype = 2) +
4   geom_pointrange(aes(x = Continent, y = Coef,
5                     ymin = Conf_lwr, ymax = Conf_upr),
6                     size = 0.75) +
7   labs(y = expression(paste(beta[1], " with 95% CI"))) +
8   theme_bw(base_size = 12)

```



ここでもう一つの次元を追加することもあるでしょう。たとえば、複数のモデルを比較した場合がそうかもしれません。以下の Pointrange\_df2 について考えてみましょう。

```

Pointrange_df2 <- tibble(
  Continent = rep(c("Asia", "Europe", "Africa", "America"), each = 2),
  Term      = rep(c("Civic Liverty", "Political Right"), 4),

```

```

Coef      = c(207.747, 29.188, 1050.164, 1284.101,
           110.025, 93.537, 581.4593, 646.9211),
Conf_lwr  = c(-385.221, -609.771, 692.204, 768.209,
           -12.648, -53.982, 262.056, 201.511),
Conf_upr  = c(800.716, 668.147, 1408.125, 1801.994,
           232.697, 241.057, 900.863, 1092.331))

```

```
1 Pointrange_df2
```

```

## # A tibble: 8 x 5
## # Groups:   Continent [4]
##   Continent Term      Coef  Conf_lwr  Conf_upr
##   <chr>     <chr>    <dbl>    <dbl>    <dbl>
## 1 Asia      Civic Liberty 208.     -385.     801.
## 2 Asia      Political Right 29.2     -610.     668.
## 3 Europe    Civic Liberty 1050.    692.     1408.
## 4 Europe    Political Right 1285.    768.     1802.
## 5 Africa    Civic Liberty 110.     -12.6     233.
## 6 Africa    Political Right 93.5     -54.0     241.
## 7 America   Civic Liberty 581.     262.     901.
## 8 America   Political Right 647.     202.     1092.

```

このデータは以下の 2 つのモデルを大陸ごとに推定した  $\beta_1$  と  $\gamma_1$  の点推定値と 95% 信頼区間です。

$$\text{PPP per capita} = \beta_0 + \beta_1 \cdot \text{FH\_CL} + \varepsilon$$

$$\text{PPP per capita} = \gamma_0 + \gamma_1 \cdot \text{FH\_PR} + v$$

どの説明変数を用いたかでエラーバーと点の色分けを行う場合、color に対して Term をマッピングします。

```

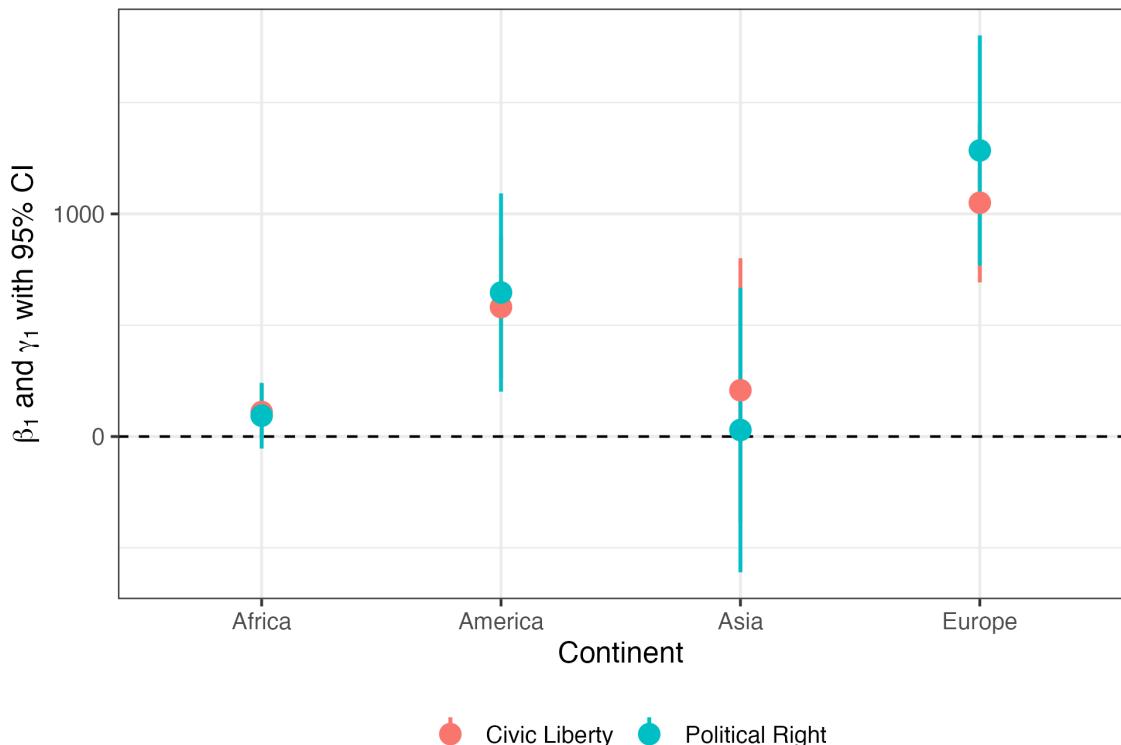
1 Pointrange_df2 %>%
2   ggplot() +
3   geom_hline(yintercept = 0, linetype = 2) +

```

```

4   geom_pointrange(aes(x = Continent, y = Coef,
5                     ymin = Conf_lwr, ymax = Conf_upr,
6                     color = Term), size = 0.75) +
7   labs(y = expression(paste(beta[1], " and ", gamma[1], " with 95% CI"))),
8         color = "") +
9   theme_bw(base_size = 12) +
10  theme(legend.position = "bottom")

```



何か違いますね。この2つのエラーバーと点の位置をずらす必要があるようです。これは3次元以上の棒グラフで使ったposition引数で調整可能です。今回は実引数としてposition\_dodge(0.5)を指定してみましょう。

```

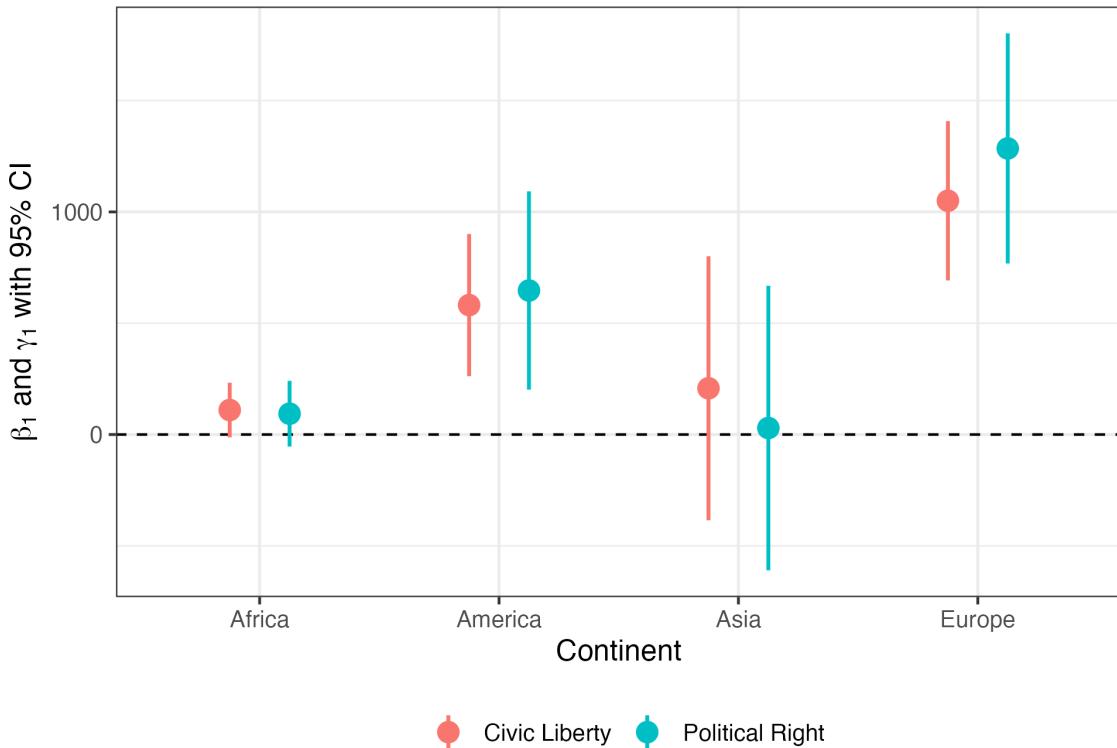
1 Pointrange_df2 %>%
2   ggplot() +
3   geom_hline(yintercept = 0, linetype = 2) +
4   geom_pointrange(aes(x = Continent, y = Coef,

```

```

5           ymin = Conf_lwr, ymax = Conf_upr,
6           color = Term),
7           size = 0.75, position = position_dodge(0.5)) +
8   labs(y = expression(paste(beta[1], " and ", gamma[1], " with 95% CI"))),
9   color = "") +
10  theme_bw(base_size = 12) +
11  theme(legend.position = "bottom")

```



これで完成です。更に、 $\alpha = 0.05$  水準で統計的に有意か否かを透明度で示し、透明度の凡例を非表示にしてみましょう。 $\alpha = 0.05$  水準で統計的に有意か否かは 95% 信頼区間の上限と下限の積が 0 より大きいか否かで判定できます。`ggplot()` にデータを渡す前に統計的有意か否かを意味する `Sig` 変数を作成し、`geom_pointrange()` の内部では `alpha` に `Sig` をマッピングします。

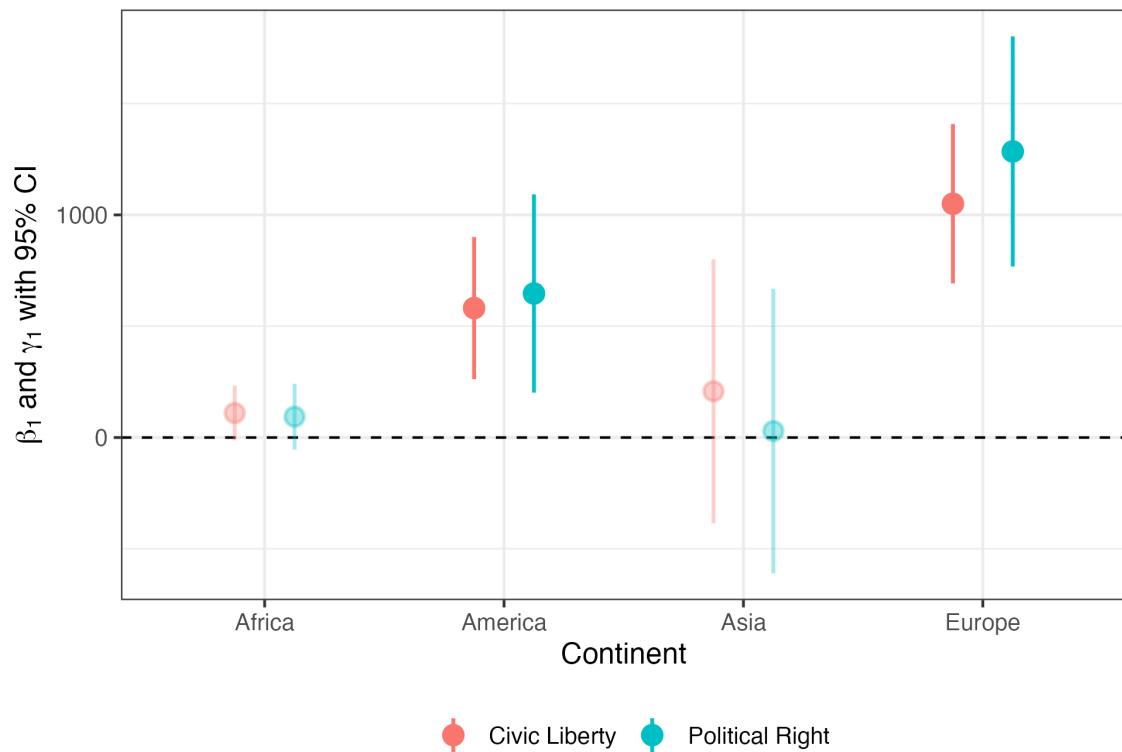
```

1 Pointrange_df2 %>%
2   mutate(Sig = if_else(Conf_lwr * Conf_upr > 0,

```

```
3             "Significant", "Insignificant")) %>%
4     ggplot() +
5     geom_hline(yintercept = 0, linetype = 2) +
6     geom_pointrange(aes(x = Continent, y = Coef,
7                           ymin = Conf_lwr, ymax = Conf_upr,
8                           color = Term, alpha = Sig),
9                           size = 0.75, position = position_dodge(0.5)) +
10    labs(y = expression(paste(beta[1], " and ", gamma[1], " with 95% CI")),
11          color = "") +
12    scale_alpha_manual(values = c("Significant" = 1, "Insignificant" = 0.35)) +
13    guides(alpha = FALSE) +
14    theme_bw(base_size = 12) +
15    theme(legend.position = "bottom")
```

## Warning: `guides(<scale> = FALSE)` is deprecated. Please use `guides(<scale> = ## "none")` instead.



## 20.6 ロリーポップチャート

ロリーポップチャートは棒グラフの特殊な形態であり、棒がロリーポップ（チュッパチャップス）の形をしているものを指します。したがって、2つの図は本質的に同じですが、棒が多い場合はロリーポップチャートを使うケースがあります。棒が非常に多い棒グラフの場合、図を不適切に縮小するとモアレが生じるケースがあるからです。

まず、Country\_df を用い、ヨーロッパ諸国の人一人当たり PPP GDP (PPP\_per\_capita) の棒グラフを作るとします。PPP\_per\_capita が欠損していないヨーロッパの国は 46 行であり、非常に棒が多い棒グラフになります。

```

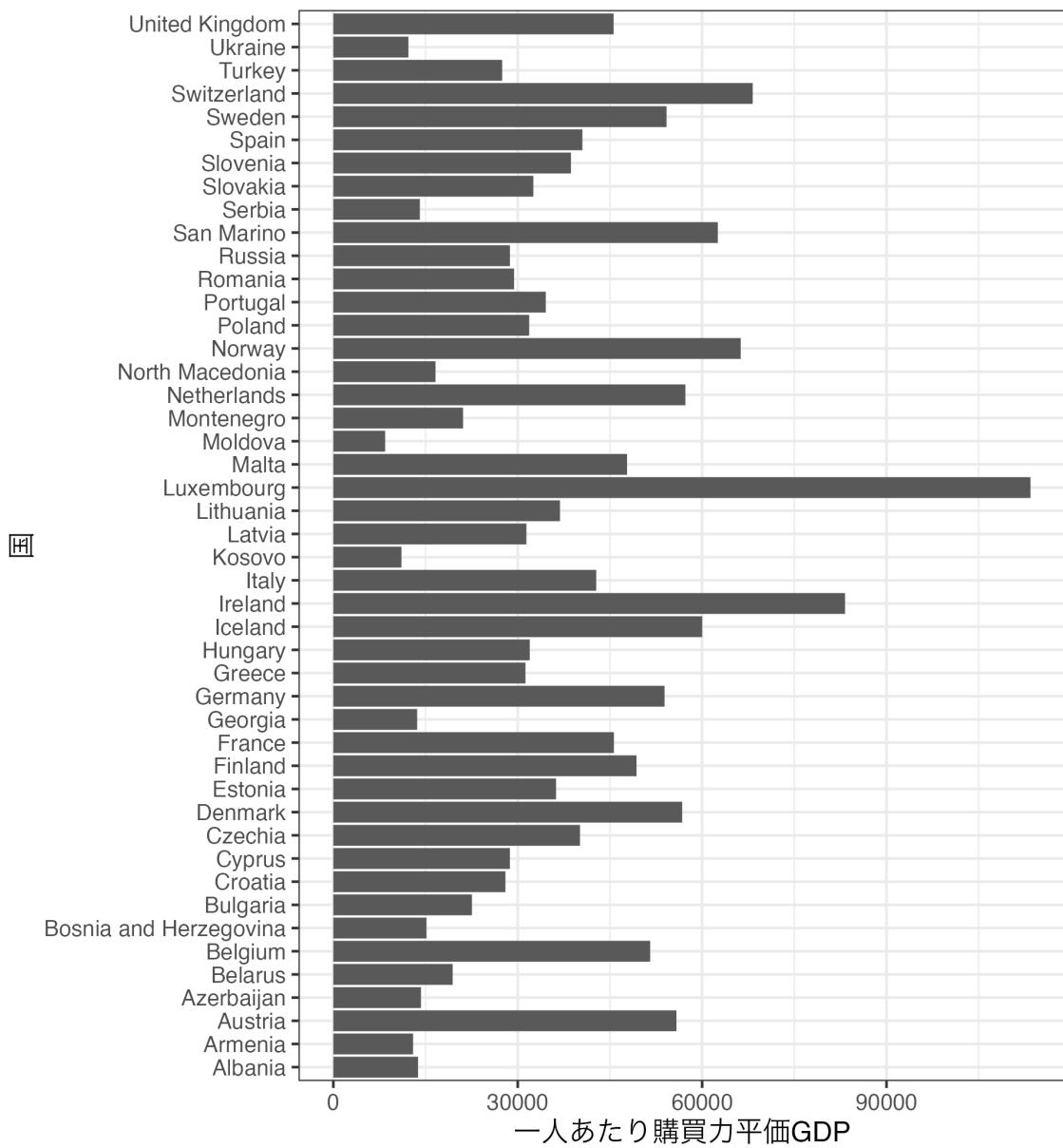
1 Country_df %>%
2   filter(Continent == "Europe") %>%
3   drop_na(PPP_per_capita) %>%
4   ggplot() +

```

```

5   geom_bar(aes(y = Country, x = PPP_per_capita), stat = "identity") +
6   labs(x = "一人あたり購買力平価 GDP", y = "国") +
7   theme_bw(base_size = 12)

```



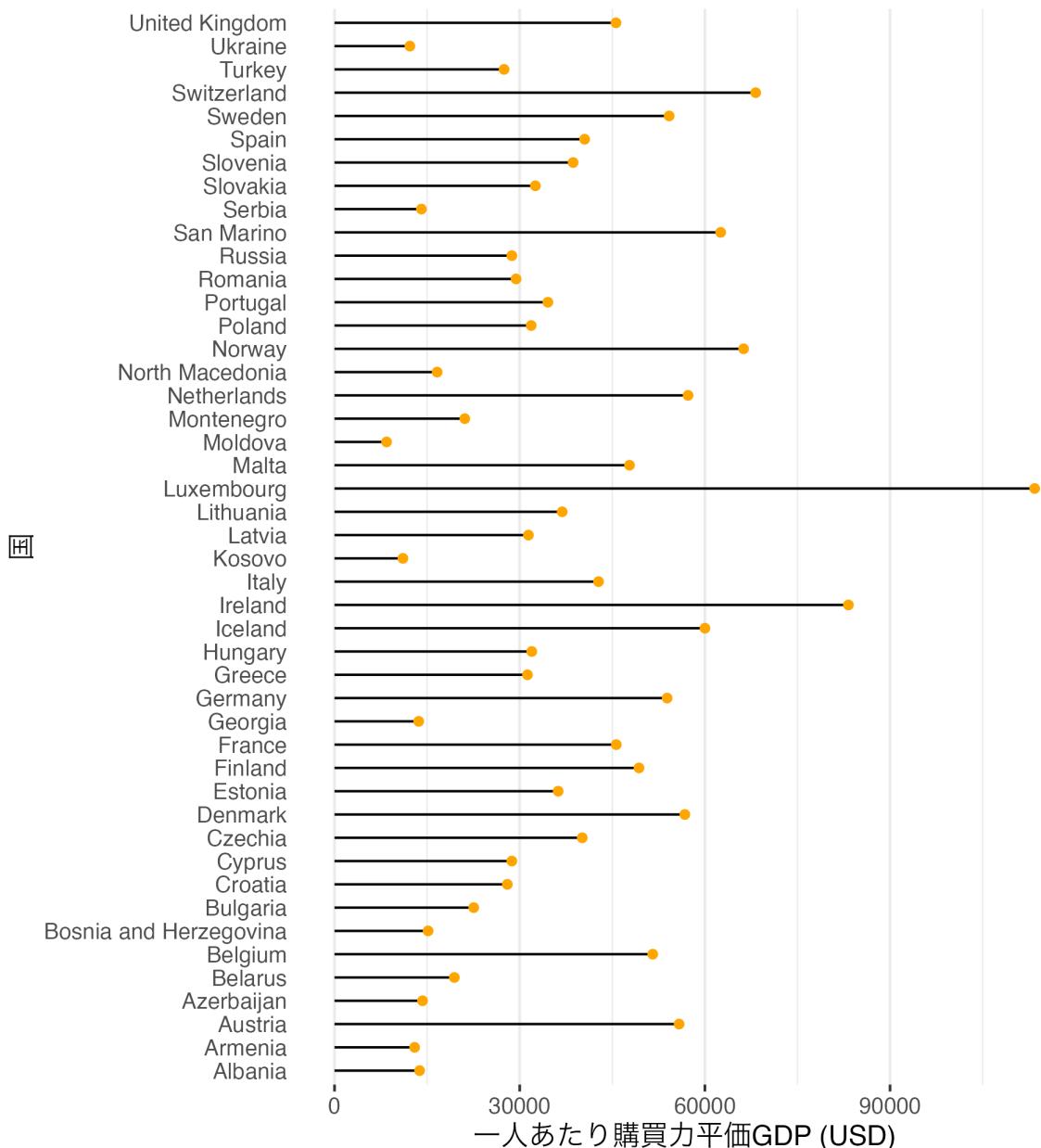
ここで登場するのがロリーポップチャートです。ロリーポップチャートの構成要素は棒とキャンディーの部分です。棒は線になるため `geom_segment()` を、キャンディーは散

布図 `geom_point()` を使います。散布図については既に第 18 章で説明しましたので、ここでは `geom_segment()` について説明します。

`geom_segment()` は直線を引く幾何オブジェクトであり、線の起点（`x` と `y`）と終点（`xend` と `yend`）に対してマッピングをする必要があります。横軸上の起点は 0、縦軸上の起点は `Country` です。そして横軸上の終点は `PPP_per_capita`、縦軸上のそれは `Country` です。縦軸上の起点と終点が同じということは水平線を引くことになります。

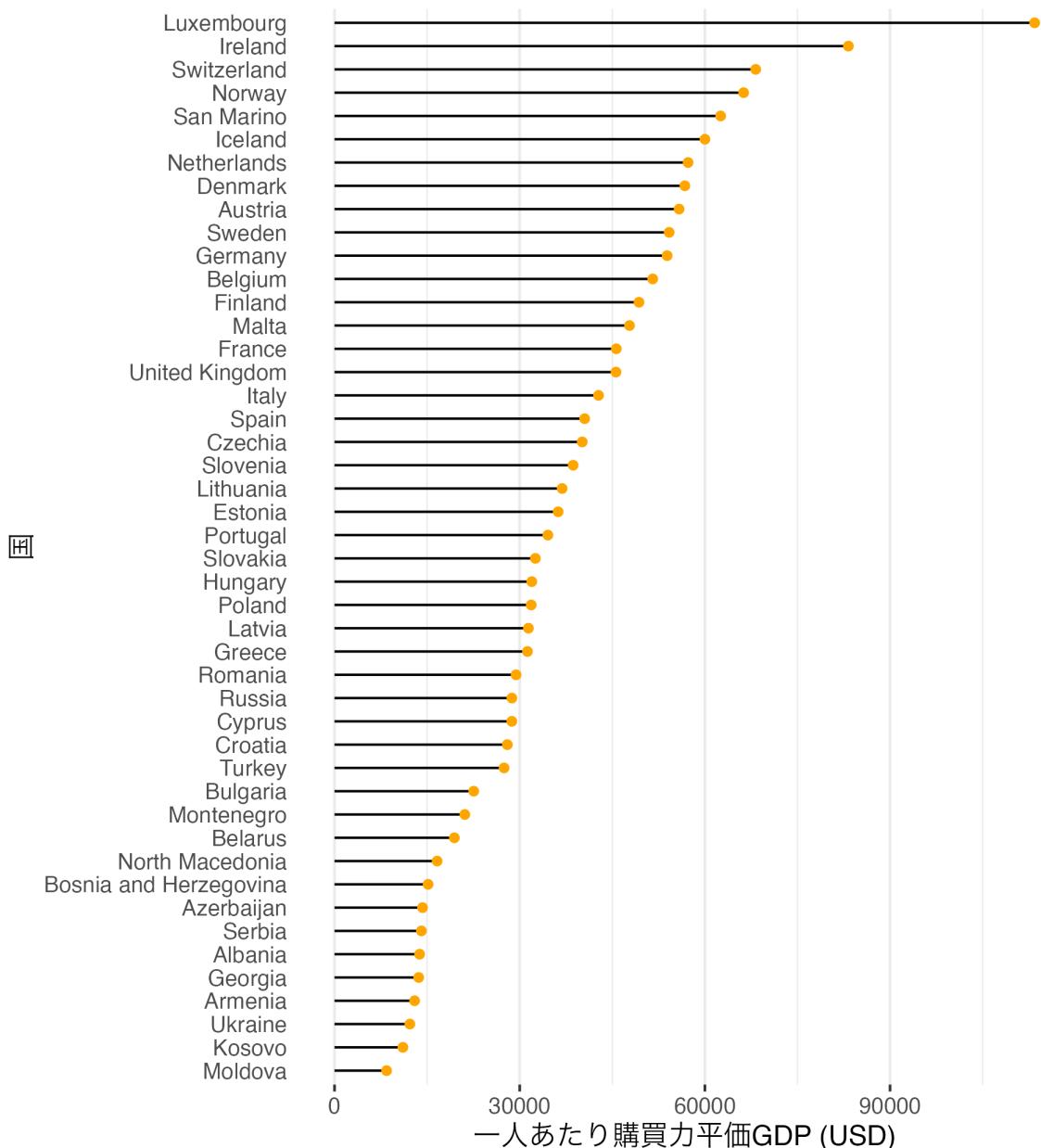
`geom_segment()` で水平線を描いたら、次は散布図をオーバーラップさせます。点の横軸上の位置は `PPP_per_capita`、縦軸上の位置は `Country` です。

```
1 Country_df %>%
2   filter(Continent == "Europe") %>%
3   drop_na(PPP_per_capita) %>%
4   ggplot() +
5   geom_segment(aes(y = Country, yend = Country,
6                     x = 0, xend = PPP_per_capita)) +
7   geom_point(aes(y = Country, x = PPP_per_capita), color = "orange") +
8   labs(x = "一人あたり購買力平価 GDP (USD)", y = "国") +
9   theme_bw(base_size = 12) +
10  theme(panel.grid.major.y = element_blank(),
11        panel.border = element_blank(),
12        axis.ticks.y = element_blank())
```



これで完成です。もし一人当たり PPP GDP 順で並べ替えたい場合は `fct_reorder()` を使います。Country を `PPP_per_capita` の低い方を先にくるようにするなら、`fct_reorder(Country, PPP_per_capita)` です。縦に並ぶの棒グラフなら最初に来る水準が下に位置されます。もし、順番を逆にしたいなら、更に `fct_rev()` で水準の順番を逆転させます。

```
1 Country_df %>%
2   filter(Continent == "Europe") %>%
3   drop_na(PPP_per_capita) %>%
4   mutate(Country = fct_reorder(Country, PPP_per_capita)) %>%
5   ggplot() +
6   geom_segment(aes(y = Country, yend = Country,
7                     x = 0, xend = PPP_per_capita)) +
8   geom_point(aes(y = Country, x = PPP_per_capita), color = "orange") +
9   labs(x = "一人あたり購買力平価 GDP (USD)", y = "国") +
10  theme_bw(base_size = 12) +
11  theme(panel.grid.major.y = element_blank(),
12        panel.border = element_blank(),
13        axis.ticks.y = element_blank())
```



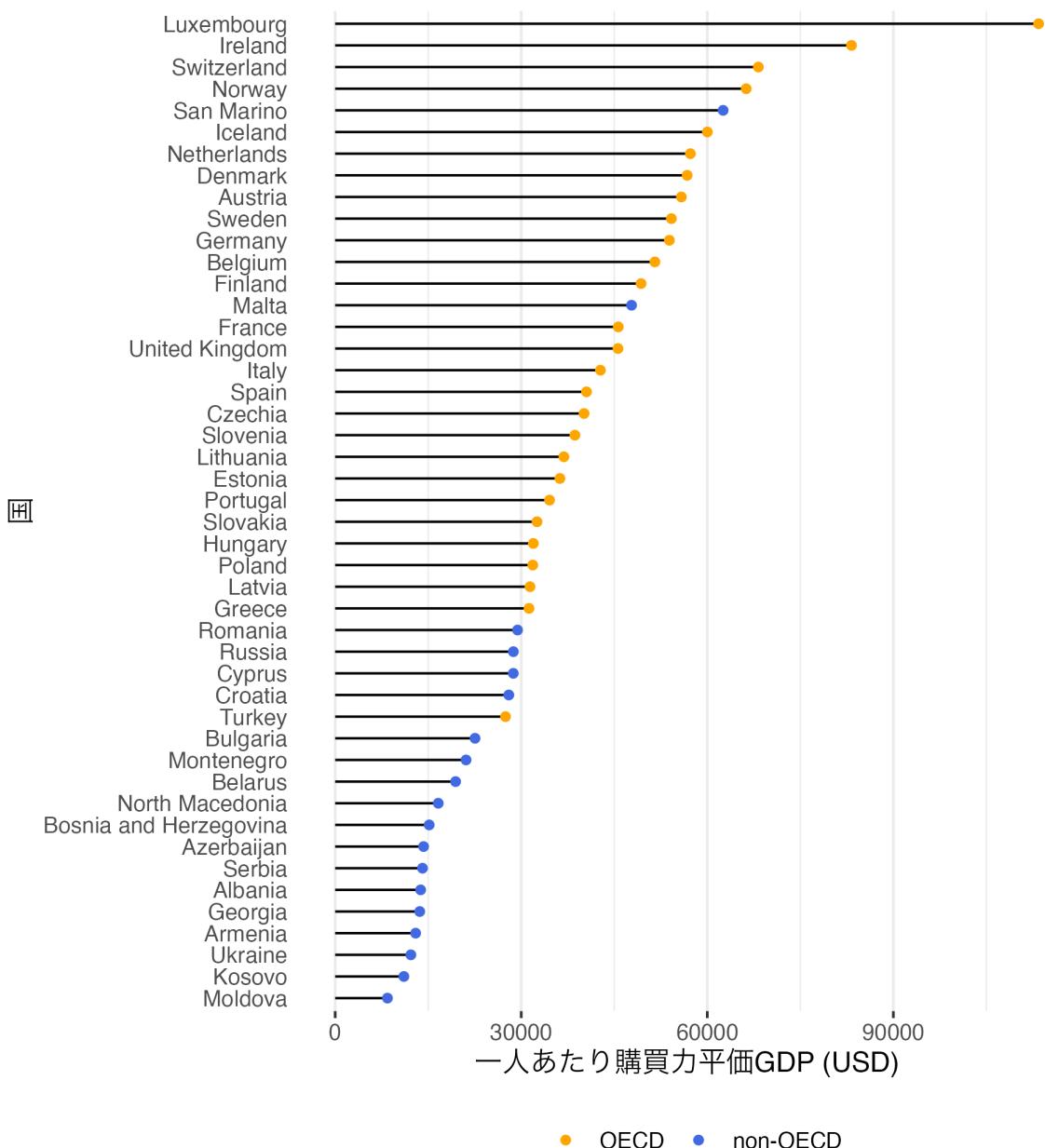
ロリーポップロリーポップチャートで次元を追加するには点（キャンディー）の色分けが考えられます。たとえば、OECD 加盟国か否かの次元を追加する場合、`geom_point()`において `color` をマッピングするだけです。

```

1 Country_df %>%
2   filter(Continent == "Europe") %>%
3   drop_na(PPP_per_capita) %>%

```

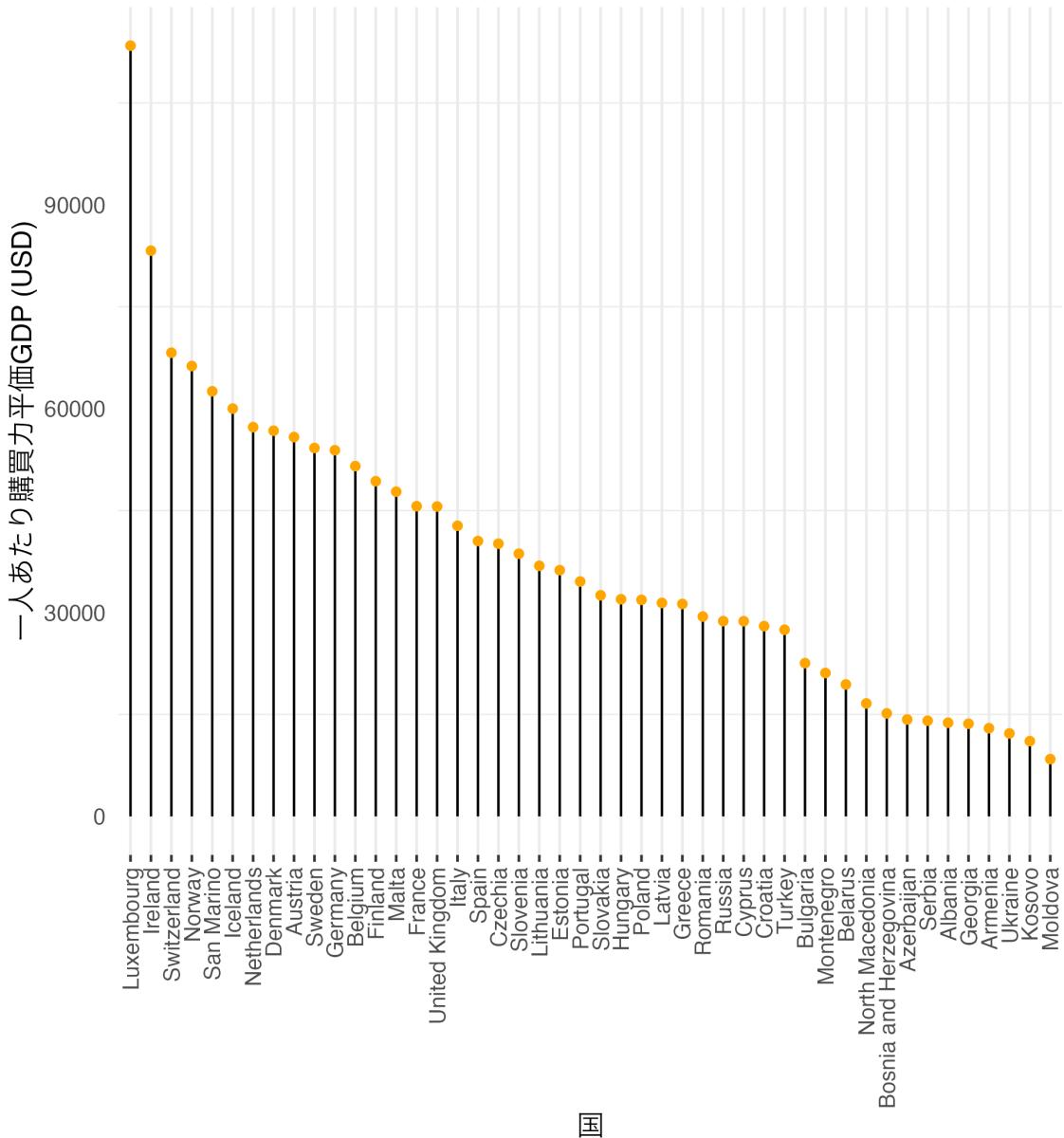
```
4   mutate(Country = fct_reorder(Country, PPP_per_capita),
5     OECD      = if_else(OECD == 1, "OECD", "non-OECD"),
6     OECD      = factor(OECD, levels = c("OECD", "non-OECD"))) %>%
7   ggplot() +
8   geom_segment(aes(y = Country, yend = Country,
9                     x = 0, xend = PPP_per_capita)) +
10  geom_point(aes(y = Country, x = PPP_per_capita, color = OECD)) +
11  scale_color_manual(values = c("OECD" = "orange", "non-OECD" = "royalblue")) +
12  labs(x = "一人あたり購買力平価 GDP (USD)", y = "国", color = "") +
13  theme_bw(base_size = 12) +
14  theme(panel.grid.major.y = element_blank(),
15        panel.border      = element_blank(),
16        axis.ticks.y      = element_blank(),
17        legend.position   = "bottom")
```



ファセット分割ももちろんできますが、この場合、OECD 加盟国の人一人当たり PPP GDP が相対的に高いことを示すなら、一つのファセットにまとめた方が良いでしょう。

以下のようにロリーポップを横に並べることもできますが、棒の数が多いケースがほとんどであるロリーポップチャートではラベルの回転が必要になるため、読みにくくなるかも知れません。

```
1 Country_df %>%
2   filter(Continent == "Europe") %>%
3   drop_na(PPP_per_capita) %>%
4   mutate(Country = fct_reorder(Country, PPP_per_capita),
5         Country = fct_rev(Country)) %>%
6   ggplot() +
7   geom_segment(aes(x = Country, xend = Country,
8                   y = 0, yend = PPP_per_capita)) +
9   geom_point(aes(x = Country, y = PPP_per_capita), color = "orange") +
10  labs(x = "国", y = "一人あたり購買力平価 GDP (USD)") +
11  theme_bw(base_size = 12) +
12  theme(panel.grid.major.y = element_blank(),
13        panel.border = element_blank(),
14        axis.ticks.y = element_blank(),
15        axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))
```



## 20.7 平滑化ライン

2次元平面上に散布図をプロットし、二変数間の関係を一本の線で要約するのは平滑化ラインです。{ggplot2}では `geom_smooth()` 幾何オブジェクトを重ねることで簡単に平滑化ラインをプロットすることができます。まずは、横軸をフリーダムハウス・スコ

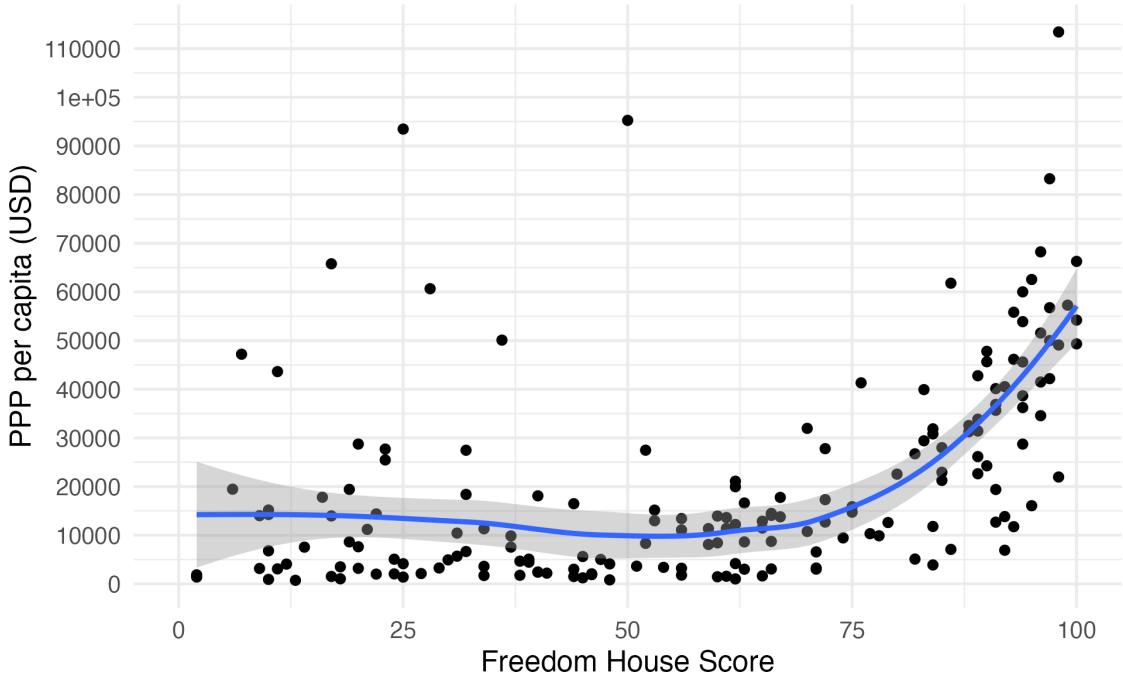
ア (FH\_Total)、縦軸を一人当たり購買力平価 GDP (PPP\_per\_capita) にした散布図を出力し、その上に平滑化ラインを追加してみましょう。geom\_smooth() にもマッピングが必要で、aes() の内部に x と y をマッピングします。今回は geom\_point() と geom\_smooth() が同じマッピング情報を共有するため、ggplot() 内部でマッピングします。

```
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = PPP_per_capita)) +
3   geom_point() +
4   geom_smooth() +
5   labs(x = "Freedom House Score", y = "PPP per capita (USD)") +
6   scale_y_continuous(breaks = seq(0, 120000, by = 10000),
7                      labels = seq(0, 120000, by = 10000)) +
8   theme_minimal(base_size = 12)

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'

## Warning: Removed 8 rows containing non-finite values (stat_smooth).

## Warning: Removed 8 rows containing missing values (geom_point).
```



青い線が平滑化ライン、網掛けの領域が 95% 信頼区間です。この線は LOESS (LOcal Estimated Scatterplot Smoothing) と呼ばれる非線形平滑化ラインです。どのようなラインを引くかは `method` 引数で指定しますが、この `method` 既定値が "loess" です。これを見るとフリーダムハウス・スコアが 75 以下の国では国の自由度と所得間の関係があまり見られませんが、75 からは正の関係が確認できます。

LOESS 平滑化の場合、`span` 引数を使って滑らかさを調整することができます。`span` の既定値は 0.75 ですが、これが小さいほど散布図によりフィットしたラインが引かれ、よりギザギザな線になります。たとえば、`span` を 0.25 にすると以下のグラフが得られます。

```

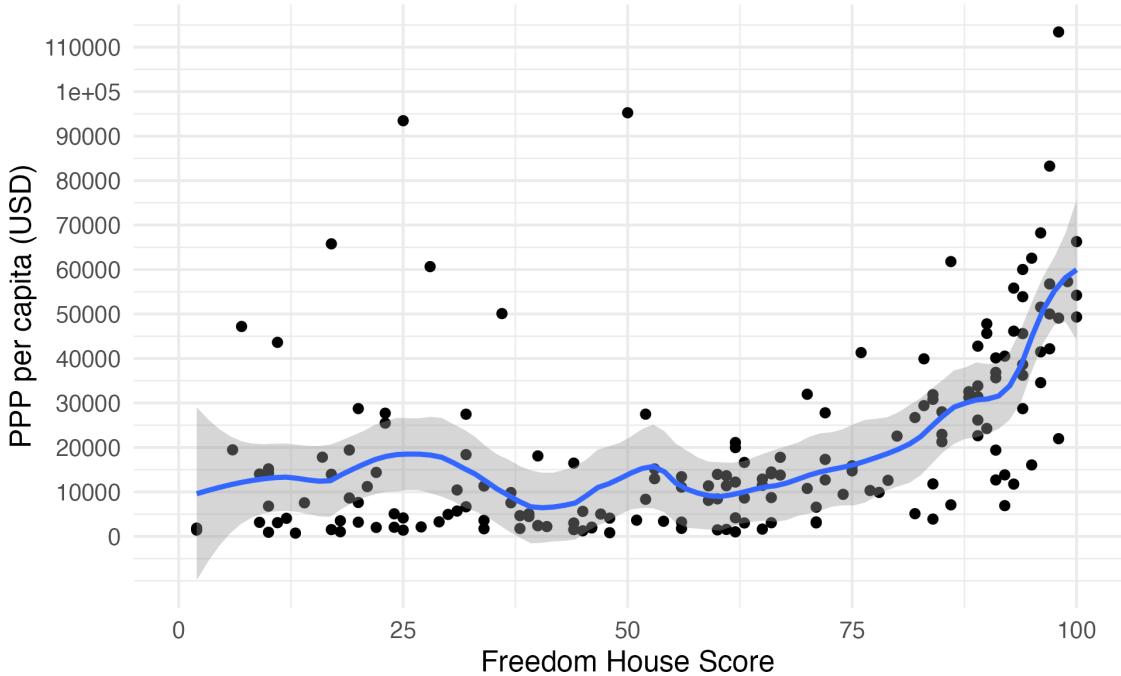
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = PPP_per_capita)) +
3   geom_point() +
4   geom_smooth(method = "loess", span = 0.25) +
5   labs(x = "Freedom House Score", y = "PPP per capita (USD)") +
6   scale_y_continuous(breaks = seq(0, 120000, by = 10000),
7                      labels = seq(0, 120000, by = 10000)) +
8   theme_minimal(base_size = 12)

```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 8 rows containing non-finite values (stat_smooth).

## Warning: Removed 8 rows containing missing values (geom_point).
```

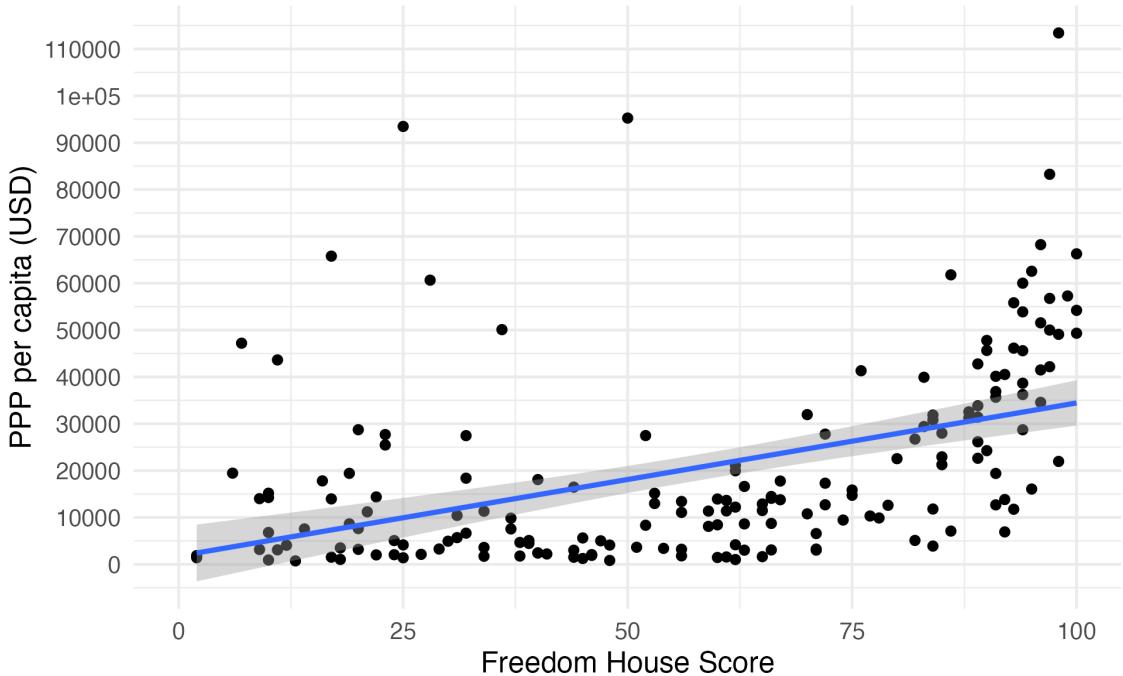


他にも定番の回帰直線を引くこともできます。method の実引数を"lm"に変えるだけです。

```
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = PPP_per_capita)) +
3   geom_point() +
4   geom_smooth(method = "lm") +
5   labs(x = "Freedom House Score", y = "PPP per capita (USD)") +
6   scale_y_continuous(breaks = seq(0, 120000, by = 10000),
7                      labels = seq(0, 120000, by = 10000)) +
8   theme_minimal(base_size = 12)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning: Removed 8 rows containing non-finite values (stat_smooth).
## Warning: Removed 8 rows containing missing values (geom_point).
```



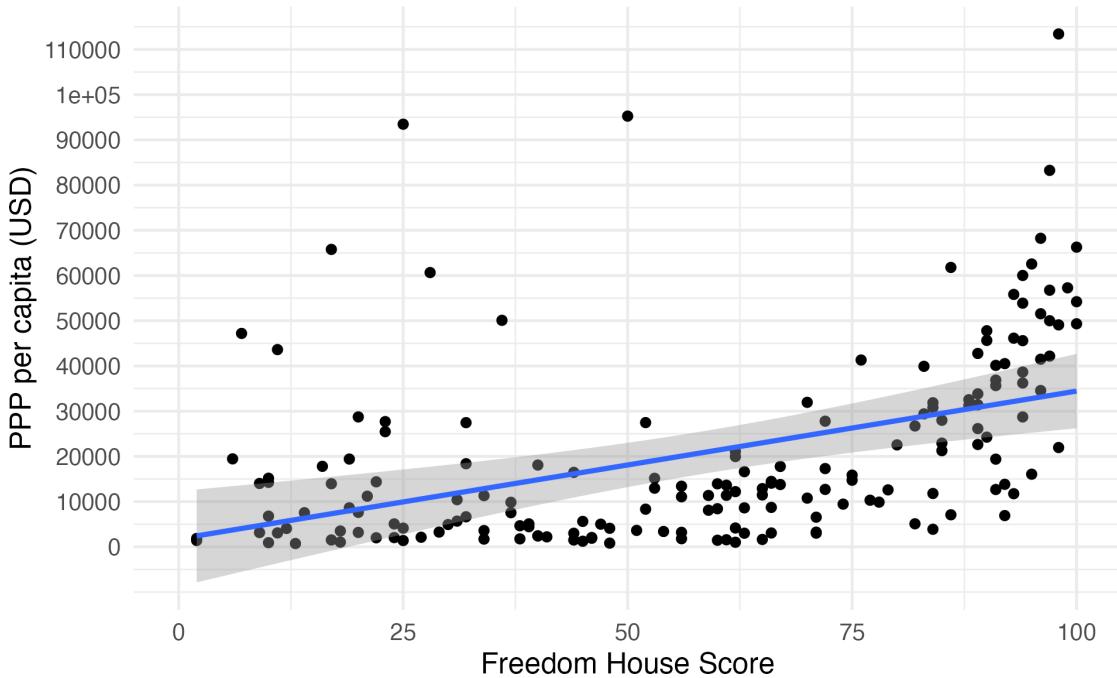
信頼区間は既定値だと 95% 信頼区間が表示されますが、`level` 引数で調整することができます。たとえば、99.9% 信頼区間を表示したい場合、`level = 0.999` を指定します。

```
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = PPP_per_capita)) +
3   geom_point() +
4   geom_smooth(method = "lm", level = 0.999) +
5   labs(x = "Freedom House Score", y = "PPP per capita (USD)") +
6   scale_y_continuous(breaks = seq(0, 120000, by = 10000),
7                      labels = seq(0, 120000, by = 10000)) +
8   theme_minimal(base_size = 12)

## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 8 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 8 rows containing missing values (geom_point).
```



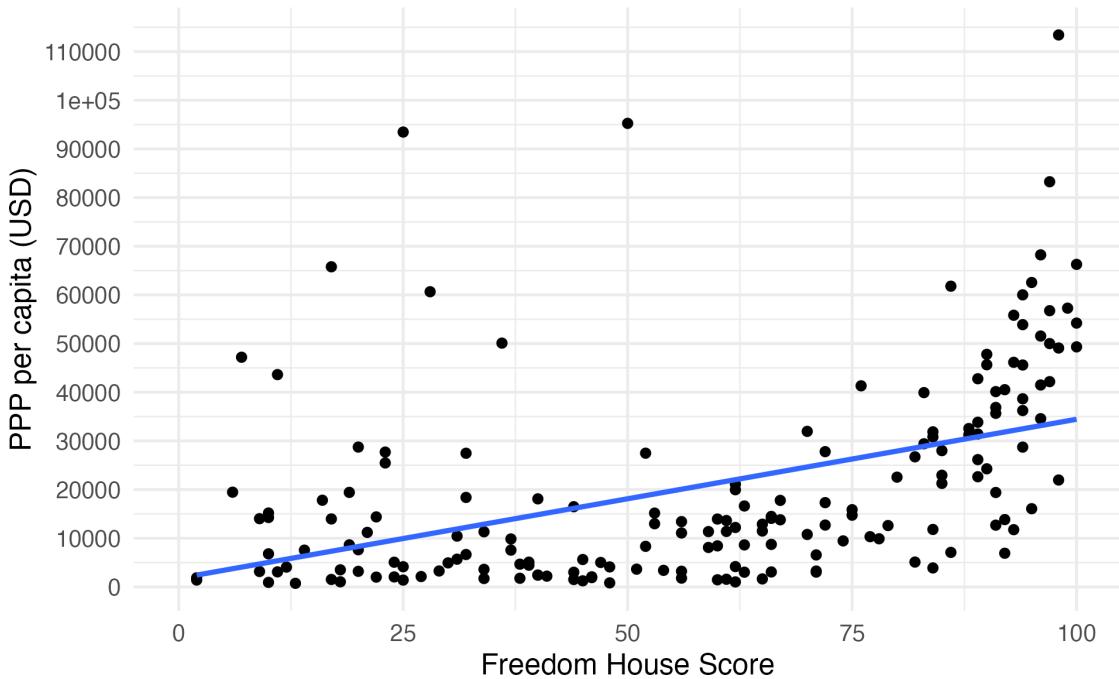
信頼区間を消したい場合は `se = FALSE` を指定します（既定値は `TRUE`）。

```
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = PPP_per_capita)) +
3   geom_point() +
4   geom_smooth(method = "lm", se = FALSE) +
5   labs(x = "Freedom House Score", y = "PPP per capita (USD)") +
6   scale_y_continuous(breaks = seq(0, 120000, by = 10000),
7                      labels = seq(0, 120000, by = 10000)) +
8   theme_minimal(base_size = 12)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning: Removed 8 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 8 rows containing missing values (geom_point).
```



最後にデータのサブセットごとに回帰直線を引く方法について説明します。散布図で色分けを行う場合、`aes()` 内で `color` 引数を指定しますが、これだけで十分です。今回はこれまでの散布図を OECD 加盟有無ごとに色分けし、それぞれ別の回帰直線を重ねてみましょう。回帰直線も色分けしたいので `color` 引数で次元を増やす必要があります、これは `geom_point()` と共に通であるため、`ggplot()` 内でマッピングします。

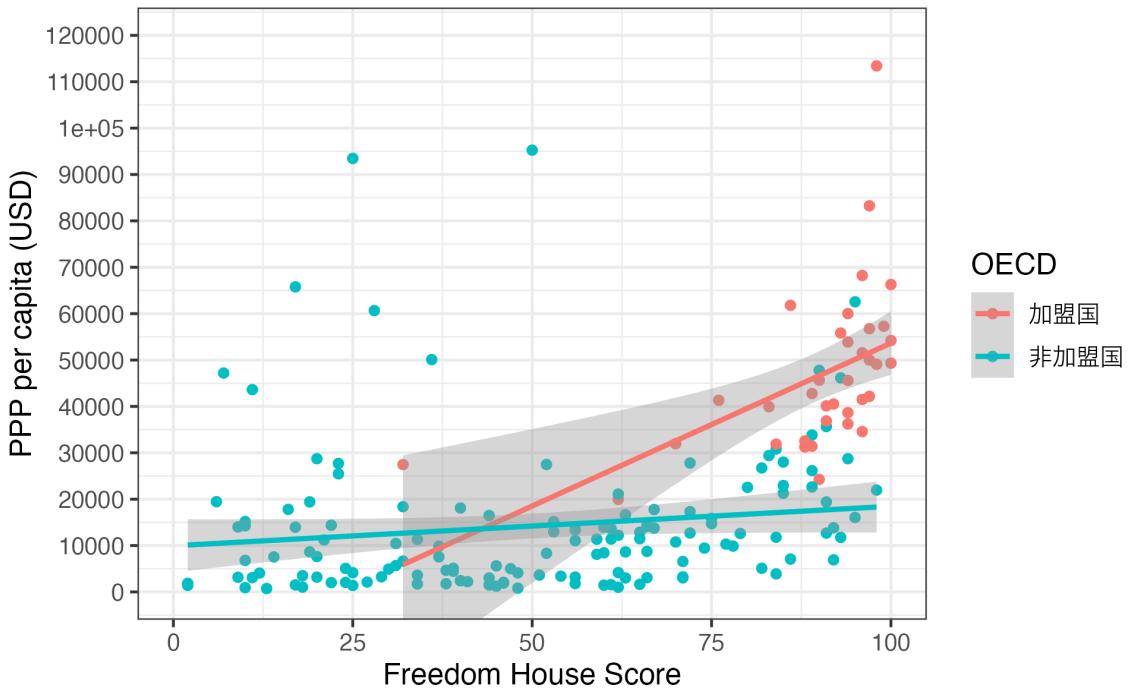
```

1 Country_df %>%
2   mutate(OECD = if_else(OECD == 1, "加盟国", "非加盟国")) %>%
3   ggplot(aes(x = FH_Total, y = PPP_per_capita, color = OECD)) +
4   geom_point() +
5   geom_smooth(method = "lm") +
6   labs(x = "Freedom House Score", y = "PPP per capita (USD)") +
7   scale_y_continuous(breaks = seq(0, 120000, by = 10000),
8                      labels = seq(0, 120000, by = 10000)) +
9   coord_cartesian(ylim = c(0, 120000)) +
10  theme_bw(base_size = 12)

## `geom_smooth()` using formula 'y ~ x'

```

```
## Warning: Removed 8 rows containing non-finite values (stat_smooth).
## Warning: Removed 8 rows containing missing values (geom_point).
```



これを見ると、国の自由度と所得の間に関係が見られるのは OECD 加盟国で、非加盟国では非常に関係が弱いことが分かります。

あまりいい方法ではないと思いますが、散布図は色 (color) で分け、回帰直線は線の種類 (linetype) で分けるならどうすれば良いでしょうか。この場合は color は geom\_point() 内部で、linetype は geom\_smooth() でマッピングします。

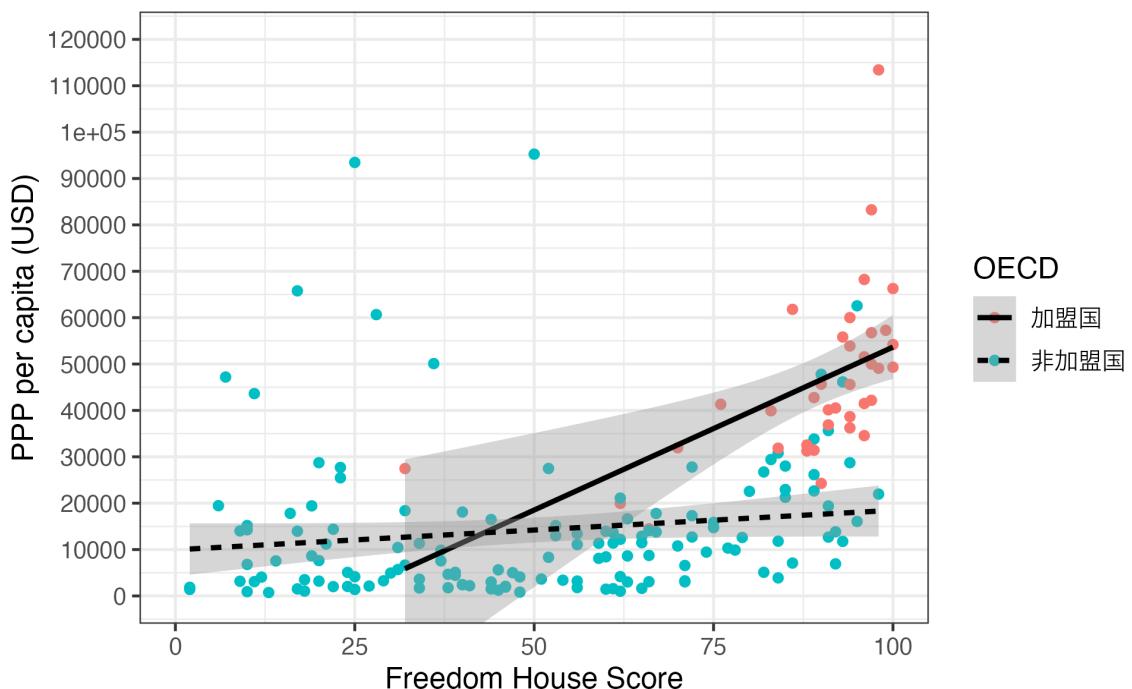
```
1 Country_df %>%
2   mutate(OECD = if_else(OECD == 1, "加盟国", "非加盟国")) %>%
3   ggplot(aes(x = FH_Total, y = PPP_per_capita)) +
4   geom_point(aes(color = OECD)) +
5   geom_smooth(aes(linetype = OECD), method = "lm", color = "black") +
6   labs(x = "Freedom House Score", y = "PPP per capita (USD)") +
7   scale_y_continuous(breaks = seq(0, 120000, by = 10000),
8                      labels = seq(0, 120000, by = 10000)) +
```

```
9     coord_cartesian(ylim = c(0, 120000)) +
10    theme_bw(base_size = 12)

## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 8 rows containing non-finite values (stat_smooth).

## Warning: Removed 8 rows containing missing values (geom_point).
```



{ggplot2}が提供する平滑化ラインには LOESS と回帰直線以外にも "glm" や "gam" などがあります。詳細は R コンソール上で?geom\_smooth を入力し、ヘルプを参照してください。

## 20.8 ヒートマップ

### 20.8.1 2つの離散変数の分布を表すヒートマップ

ヒートマップ (heat map) には2つの使い方があります。まずは、離散変数×離散変数の同時分布を示す時です。これは後ほど紹介するモザイク・プロットと目的は同じですが、モザイク・プロットはセルの面積で密度や度数を表すに対し、ヒートマップは主に色で密度や度数を表します。

ここでは一人当たり購買力平価 GDP (PPP\_per\_capita) を「1万ドル未満」、「1万ドル以上・2万ドル未満」、「2万ドル以上、3万ドル未満」、「3万ドル以上」の離散変数に変換し、大陸ごとの国家数をヒートマップとして示してみたいと思います。まずは、変数のリコーディングをし、全て factor 化します。最後に国家名 (Country)、大陸 (Continent)、所得 (Income)、フリーダム・ハウス・スコア (FH\_Total)、人間開発指数 (HDI\_2018) 列のみ抽出し、Heatmap\_df という名のオブジェクトとして格納しておきます。

```
1 Heatmap_df <- Country_df %>%
2   filter(!is.na(PPP_per_capita)) %>%
3   mutate(Continent = recode(Continent,
4                             "Africa"  = "アフリカ",
5                             "America" = "アメリカ",
6                             "Asia"     = "アジア",
7                             "Europe"   = "ヨーロッパ",
8                             .default   = "オセアニア"),
9   Continent = factor(Continent, levels = c("アフリカ", "アメリカ", "アジア",
10                      "ヨーロッパ", "オセアニア")),
11  Income     = case_when(PPP_per_capita < 10000 ~ "1万ドル未満",
12                      PPP_per_capita < 20000 ~ "1万ドル以上\n2万ドル未満",
13                      PPP_per_capita < 30000 ~ "2万ドル以上\n3万ドル未満",
14                      TRUE                 ~ "3万ドル以上"),
15  Income     = factor(Income, levels = c("1万ドル未満", "1万ドル以上\n2万ドル未満",
16                      "2万ドル以上\n3万ドル未満", "3万ドル以上"))
```

```

17   select(Country, Continent, Income, FH_Total, HDI_2018)
18
19 Heatmap_df

## # A tibble: 178 x 5
##   Country      Continent Income   FH_Total HDI_2018
##   <chr>        <fct>    <fct>    <dbl>    <dbl>
## 1 Afghanistan アジア    "1万ドル未満" 27      0.496
## 2 Albania      ヨーロッパ "1万ドル以上\n2万ドル未満" 67      0.791
## 3 Algeria      アフリカ  "1万ドル以上\n2万ドル未満" 34      0.759
## 4 Angola       アフリカ  "1万ドル未満"      32      0.574
## 5 Antigua and Barbuda アメリカ  "2万ドル以上\n3万ドル未満" 85      0.776
## 6 Argentina    アメリカ  "2万ドル以上\n3万ドル未満" 85      0.83
## 7 Armenia      ヨーロッパ "1万ドル以上\n2万ドル未満" 53      0.76
## 8 Australia    オセアニア "3万ドル以上"      97      0.938
## 9 Austria      ヨーロッパ "3万ドル以上"      93      0.914
## 10 Azerbaijan  ヨーロッパ "1万ドル以上\n2万ドル未満" 10      0.754
## # ... with 168 more rows

```

次は `group_by()` と `summarise()` を使って、各カテゴリーに属するケース数を計算し、`N` という名の列として追加します。

```

1 Heatmap_df1 <- Heatmap_df %>%
2   group_by(Continent, Income) %>%
3   summarise(N      = n(),
4             .groups = "drop")
5
6 Heatmap_df1

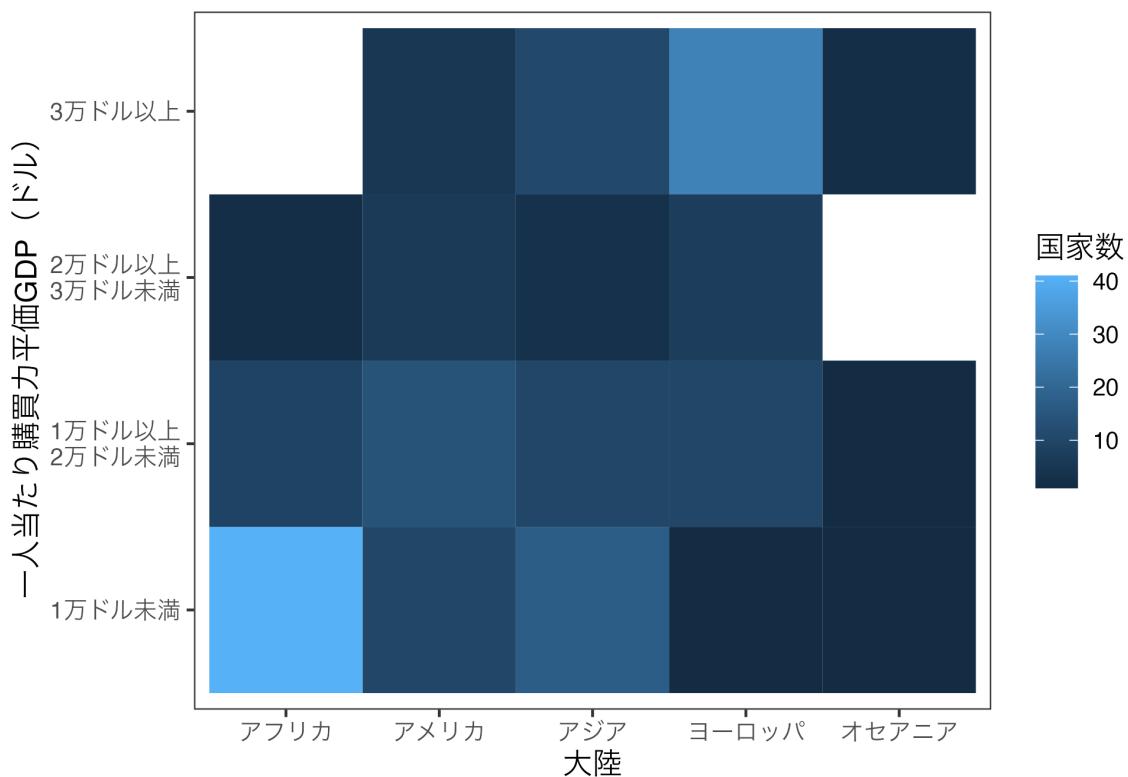
## # A tibble: 18 x 3
##   Continent Income      N
##   <fct>    <fct>    <int>
## 1 アフリカ  "1万ドル未満" 41

```

```
## 2 アフリカ "1 万ドル以上\n2 万ドル未満" 9
## 3 アフリカ "2 万ドル以上\n3 万ドル未満" 2
## 4 アメリカ "1 万ドル未満" 10
## 5 アメリカ "1 万ドル以上\n2 万ドル未満" 14
## 6 アメリカ "2 万ドル以上\n3 万ドル未満" 6
## 7 アメリカ "3 万ドル以上" 5
## 8 アジア "1 万ドル未満" 17
## 9 アジア "1 万ドル以上\n2 万ドル未満" 10
## 10 アジア "2 万ドル以上\n3 万ドル未満" 3
## 11 アジア "3 万ドル以上" 11
## 12 ヨーロッパ "1 万ドル未満" 1
## 13 ヨーロッパ "1 万ドル以上\n2 万ドル未満" 10
## 14 ヨーロッパ "2 万ドル以上\n3 万ドル未満" 7
## 15 ヨーロッパ "3 万ドル以上" 28
## 16 オセアニア "1 万ドル未満" 1
## 17 オセアニア "1 万ドル以上\n2 万ドル未満" 1
## 18 オセアニア "3 万ドル以上" 2
```

これでデータの準備は終わりました。ヒートマップを作成する幾何オブジェクトは `geom_tile()` です。同時分布を示したい変数を、それぞれ `x` と `y` にマッピングし、密度、または度数を表す変数を `fill` にマッピングします。ここでは横軸を大陸 (Continent)、縦軸を一人当たり購買力平価 GDP (Income) とし、`fill` には `N` 変数をマッピングします。

```
1 Heatmap_df1 %>%
2   ggplot() +
3   geom_tile(aes(x = Continent, y = Income, fill = N)) +
4   labs(x = "大陸", y = "一人当たり購買力平価 GDP (ドル)", fill = "国家数") +
5   theme_bw(base_size = 12) +
6   theme(panel.grid = element_blank()) # グリッドラインを消す
```



明るいほどカテゴリーに属するケースが多く、暗いほど少ないことを意味します。これを見ると世界で最も多くの割合を占めているのは、一人当たり購買力平価 GDP が1万ドル未満のアフリカの国で、次は一人当たり購買力平価 GDP が3万ドル以上のヨーロッパの国であることが分かります。欠損している（ケース数が0）セルは白の空白となります。

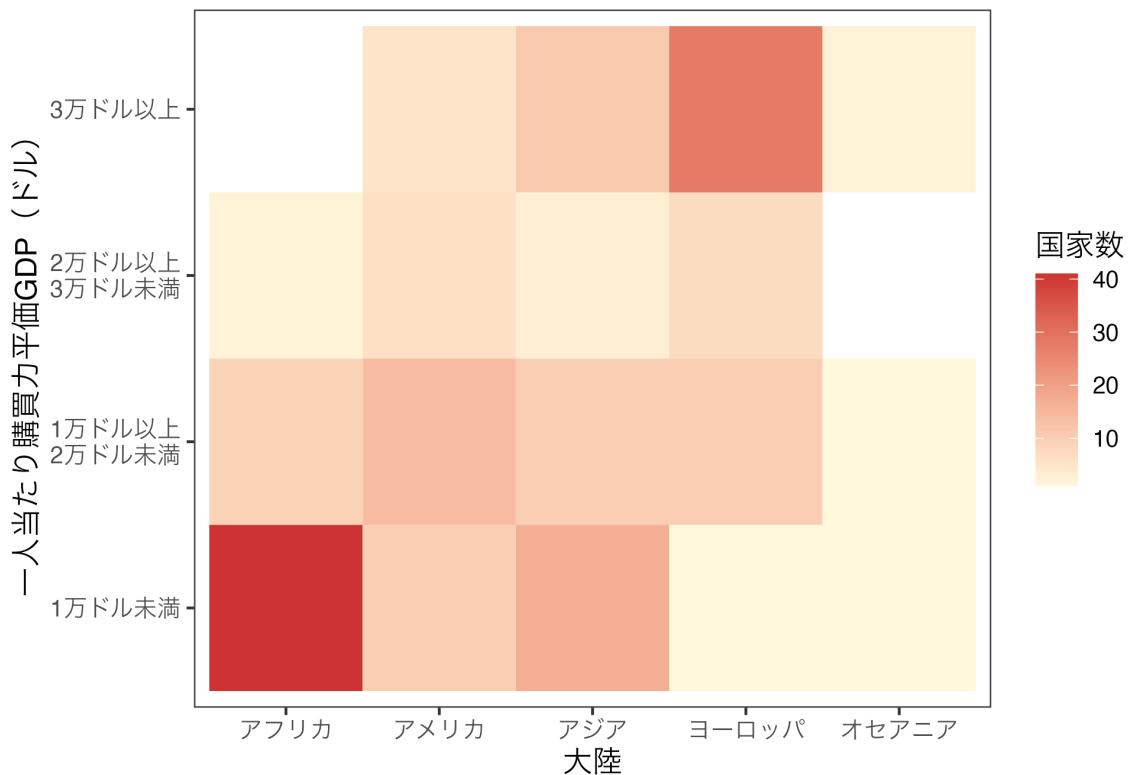
色をカスタマイズするには `scale_fill_gradient()` です。これは第19章で紹介しました `scale_color_gradient()` と同じです。`scale_fill_gradient()` は中間点なし、`scale_fill_gradient2()` は中間点ありの場合に使いますが、ここでは度数が小さい場合は `cornsilk` 色を、大きい場合は `brown3` 色を使います。それぞれ `low` と `high` に色を指定するだけです。

```

1 Heatmap_df1 %>%
2   ggplot() +
3   geom_tile(aes(x = Continent, y = Income, fill = N)) +
4   labs(x = "大陸", y = "一人当たり購買力平価 GDP (ドル)", fill = "国家数") +
5   scale_fill_gradient(low = "cornsilk",
6                       high = "brown3") +

```

```
7   theme_bw(base_size = 12) +
8   theme(panel.grid = element_blank()) # グリッドラインを消す
```



気のせいかも知れませんが、先ほどよりは読みやすくなったような気がしますね。

### 20.8.2 離散変数 × 離散変数における連続変数の値を示すヒートマップ

次は、離散変数 × 離散変数における連続変数の値を示すヒートマップを作ってみましょう。ヒートマップにおけるそれぞれのタイル (tile) は横軸上の位置と縦軸上の位置情報を持ち、これは前回と同様、離散変数でマッピングされます。そして、タイルの色は何らかの連続変数にマッピングされます。前回作成しましたヒートマップは度数、または密度であり、これも実は連続変数だったので、図の作り方は本質的には同じです。

ここでは大陸と所得ごとに人間開発指数の平均値を表すヒートマップを作ってみましょう。大陸 (Continent) と所得 (Income) でグループ化し、人間開発指数 (HDI\_2018)

の平均値を計算したものを Heatmap\_df2 という名のオブジェクトとして格納します。

```
1 Heatmap_df2 <- Heatmap_df %>%
2   group_by(Continent, Income) %>%
3   summarise(HDI      = mean(HDI_2018, na.rm = TRUE),
4             .groups = "drop")
5
6 Heatmap_df2
```

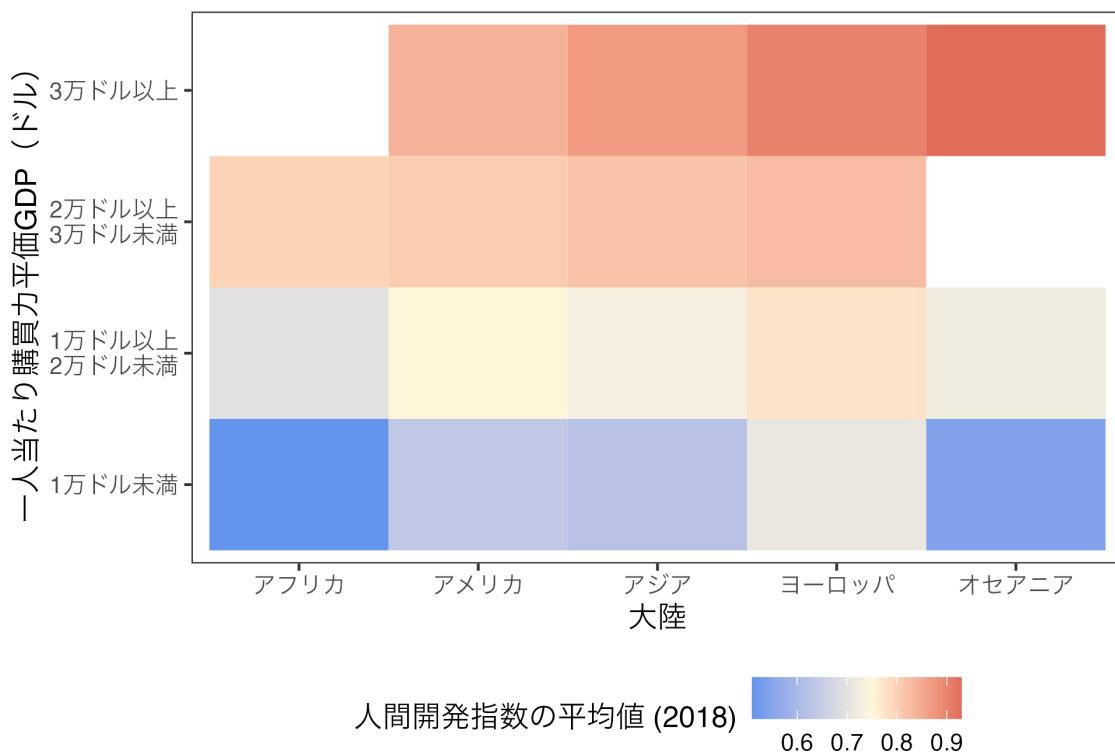
  

```
## # A tibble: 18 x 3
##   Continent Income          HDI
##   <fct>     <fct>        <dbl>
## 1 アフリカ   "1 万ドル未満"  0.510
## 2 アフリカ   "1 万ドル以上\n2 万ドル未満"  0.697
## 3 アフリカ   "2 万ドル以上\n3 万ドル未満"  0.798
## 4 アメリカ   "1 万ドル未満"  0.638
## 5 アメリカ   "1 万ドル以上\n2 万ドル未満"  0.752
## 6 アメリカ   "2 万ドル以上\n3 万ドル未満"  0.806
## 7 アメリカ   "3 万ドル以上"  0.841
## 8 アジア     "1 万ドル未満"  0.624
## 9 アジア     "1 万ドル以上\n2 万ドル未満"  0.730
## 10 アジア    "2 万ドル以上\n3 万ドル未満"  0.818
## 11 アジア    "3 万ドル以上"  0.872
## 12 ヨーロッパ "1 万ドル未満"  0.711
## 13 ヨーロッパ "1 万ドル以上\n2 万ドル未満"  0.776
## 14 ヨーロッパ "2 万ドル以上\n3 万ドル未満"  0.827
## 15 ヨーロッパ "3 万ドル以上"  0.902
## 16 オセアニア "1 万ドル未満"  0.543
## 17 オセアニア "1 万ドル以上\n2 万ドル未満"  0.724
## 18 オセアニア "3 万ドル以上"  0.930
```

作図の方法は前回と同じですが、今回はタイルの色塗り (fill) を人間開発指数の平均値 (HDI) でマッピングする必要があります。他の箇所は同じコードでも良いですが、ここでは色塗りの際、中間点を指定してみましょう。たとえば人間開発指数が 0.75 なら色

を cornsilk 色とし、これより低いほど cornflowerblue 色に、高いほど brown3 色になるように指定します。中間点を持つグラデーション色塗りは `scale_fill_gradient2()` で調整することができます。使い方は `scale_fill_gradient()` とほぼ同じですが、中間点の色 (`mid`) と中間点の値 (`midpoint`) をさらに指定する必要があります。

```
1 Heatmap_df2 %>%
2   ggplot() +
3   geom_tile(aes(x = Continent, y = Income, fill = HDI)) +
4   labs(x = "大陸", y = "一人当たり購買力平価 GDP (ドル)",
5        fill = "人間開発指数の平均値 (2018)") +
6   scale_fill_gradient2(low = "cornflowerblue",
7                        mid = "cornsilk",
8                        high = "brown3",
9                        midpoint = 0.75) +
10  theme_bw(base_size = 12) +
11  theme(legend.position = "bottom",
12        panel.grid     = element_blank())
```



## 20.9 等高線図

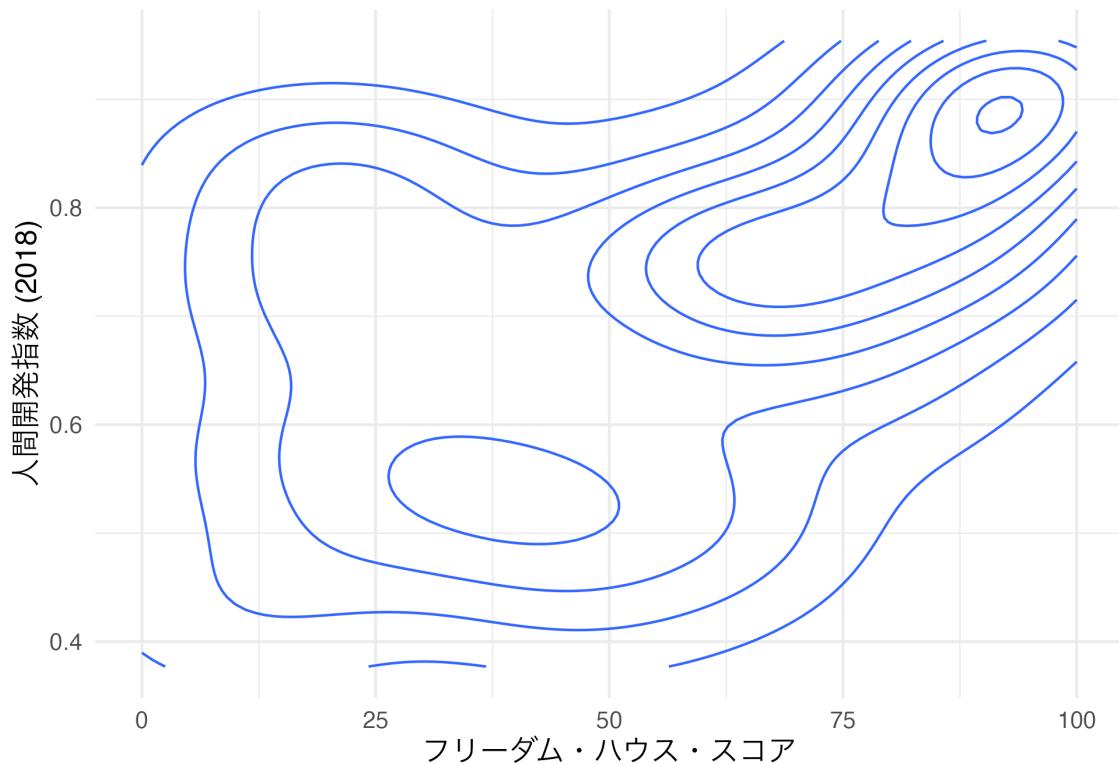
連続変数 × 連続変数の同時分布

```

1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = HDI_2018)) +
3   geom_density_2d() +
4   labs(x = "フリーダム・ハウス・スコア", y = "人間開発指数 (2018)") +
5   theme_minimal(base_size = 12)

## Warning: Removed 6 rows containing non-finite values (stat_density2d).

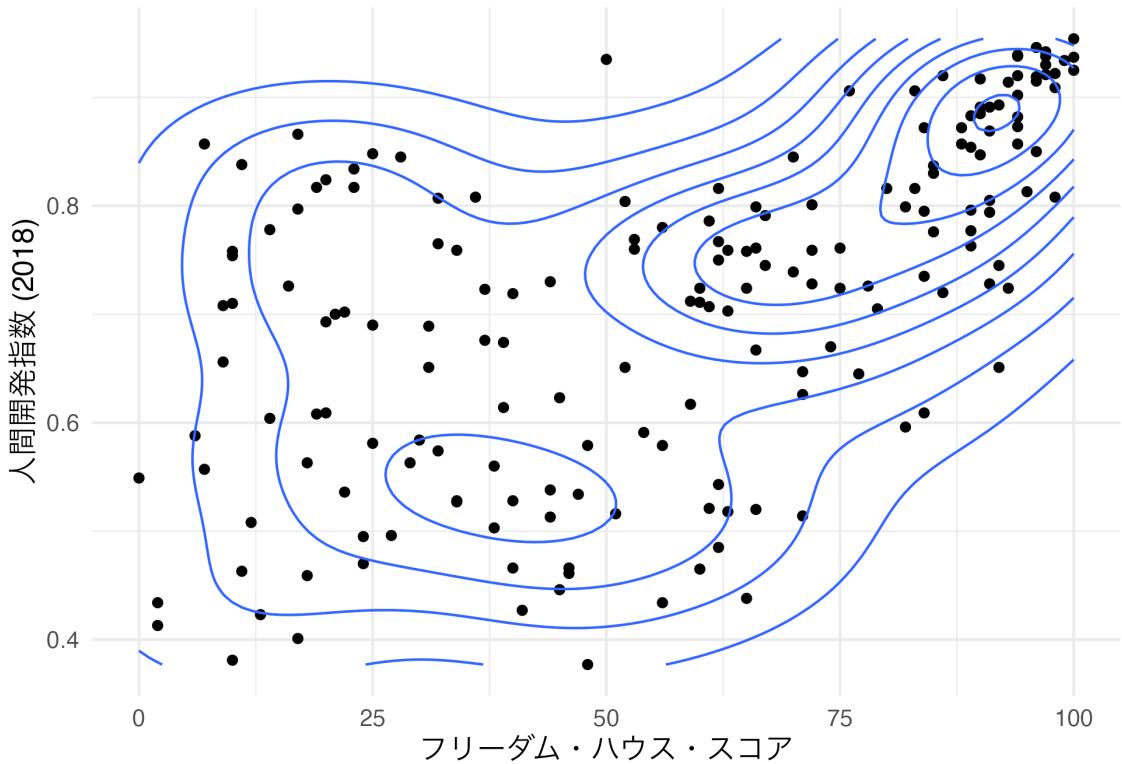
```



```
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = HDI_2018)) +
3   geom_point() +
4   geom_density_2d() +
5   labs(x = "フリーダム・ハウス・スコア", y = "人間開発指数 (2018)") +
6   theme_minimal(base_size = 12)

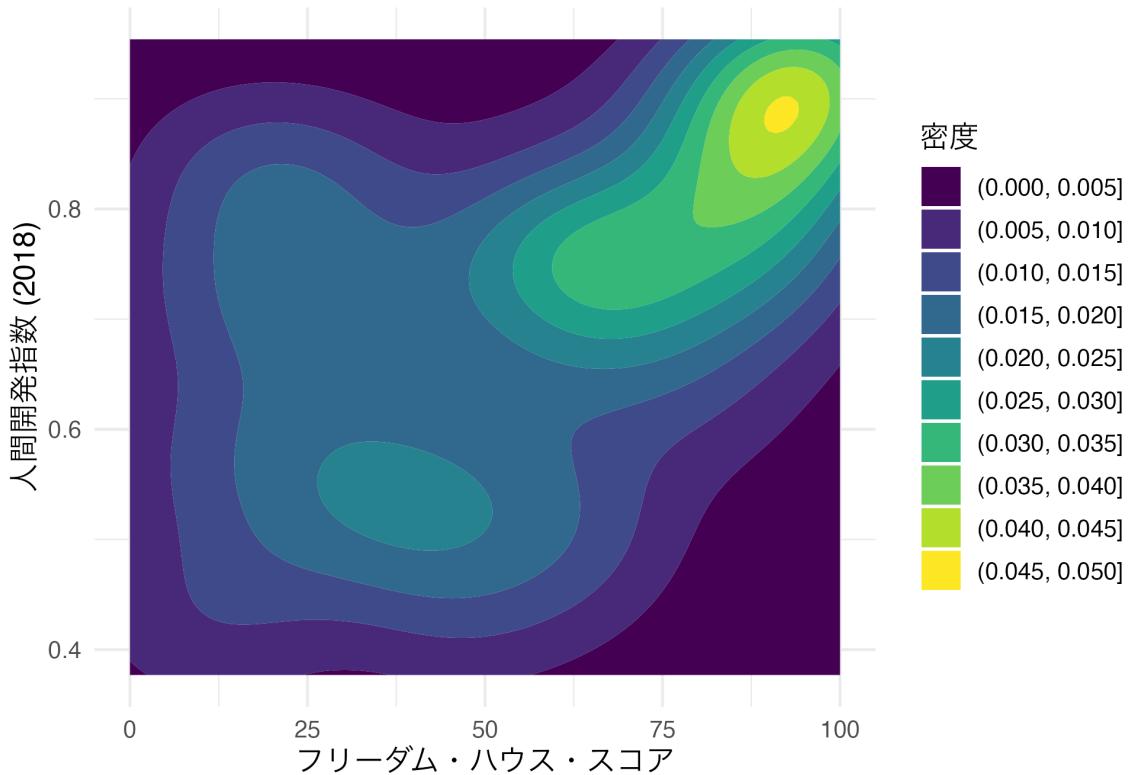
## Warning: Removed 6 rows containing non-finite values (stat_density2d).

## Warning: Removed 6 rows containing missing values (geom_point).
```



```
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = HDI_2018)) +
3   geom_density_2d_filled() +
4   labs(x = "フリーダム・ハウス・スコア", y = "人間開発指数 (2018)",
5        fill = "密度") +
6   theme_minimal(base_size = 12)
```

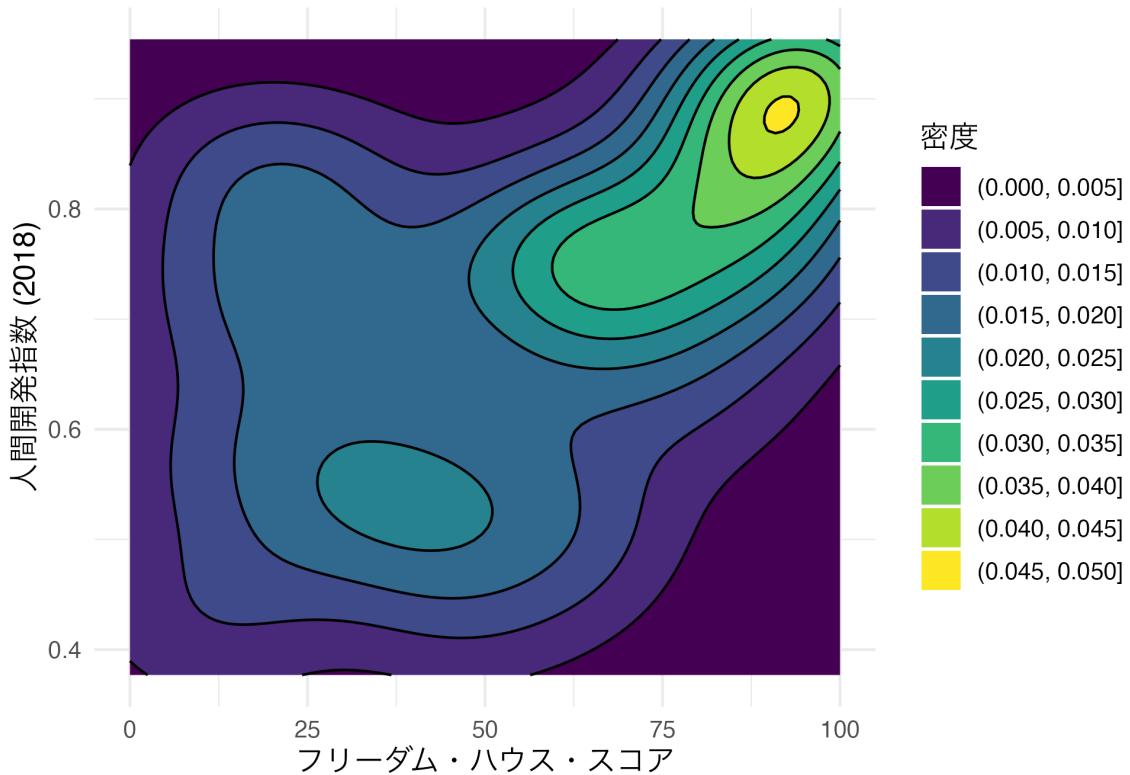
## Warning: Removed 6 rows containing non-finite values (stat\_density2d\_filled).



`geom_density_2d_filled()` オブジェクトの後に `geom_density_2d()` オブジェクトを重ねると、区間の区画線を追加することもできます。

```
1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = HDI_2018)) +
3   geom_density_2d_filled() +
4   geom_density_2d(color = "black") +
5   labs(x = "フリーダム・ハウス・スコア", y = "人間開発指数 (2018)",
6        fill = "密度") +
7   theme_minimal(base_size = 12)
```

```
## Warning: Removed 6 rows containing non-finite values (stat_density2d_filled).
## Warning: Removed 6 rows containing non-finite values (stat_density2d).
```



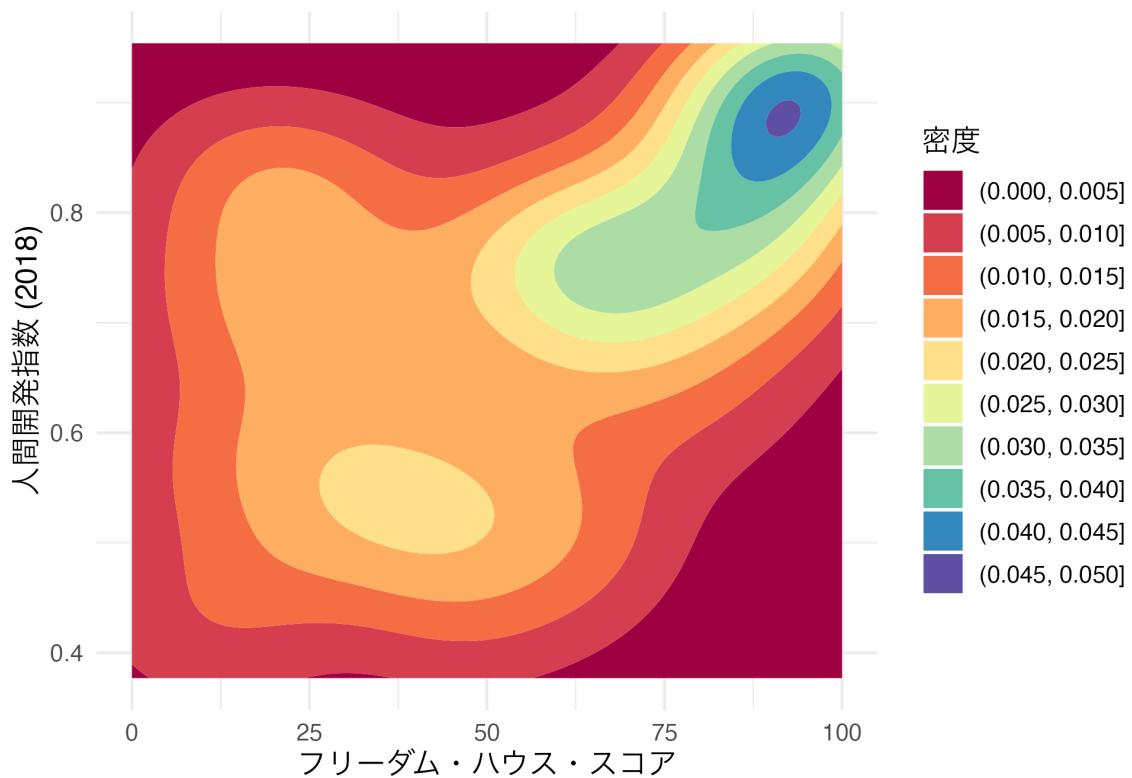
色が気に入らない場合、自分で調整することも可能です。`scale_fill_manual()` で各区間ごとの色を指定することもできませんが、あまり効率的ではありません。ここでは `scale_fill_brewer()` 関数を使って、ColorBrewer のパレットを使ってみましょう。引数なしでも使えますが、既定値のパレットは区間が 9 つまで対応します。今回の等高線図は全部で 10 区間ですので、あまり適切ではありません。ここでは 11 区間まで対応可能な "Spectral" パレットを使いますが、これは `palette` 引数で指定できます。

```

1 Country_df %>%
2   ggplot(aes(x = FH_Total, y = HDI_2018)) +
3   geom_density_2d_filled() +
4   scale_fill_brewer(palette = "Spectral") +
5   labs(x = "フリーダム・ハウス・スコア", y = "人間開発指数 (2018)" ,
6        fill = "密度") +
7   theme_minimal(base_size = 12)

## Warning: Removed 6 rows containing non-finite values (stat_density2d_filled).

```



`palette` で指定可能なカラーパレットの一覧は `{RColorBrewer}` の `display.brewer.all()` 関数で確認することができます。各パレットが何区間まで対応できるかを見てから自分でパレットを作成することも可能ですが、詳細はネット上の各種記事を参照してください。

```
1 RColorBrewer::display.brewer.all()
```

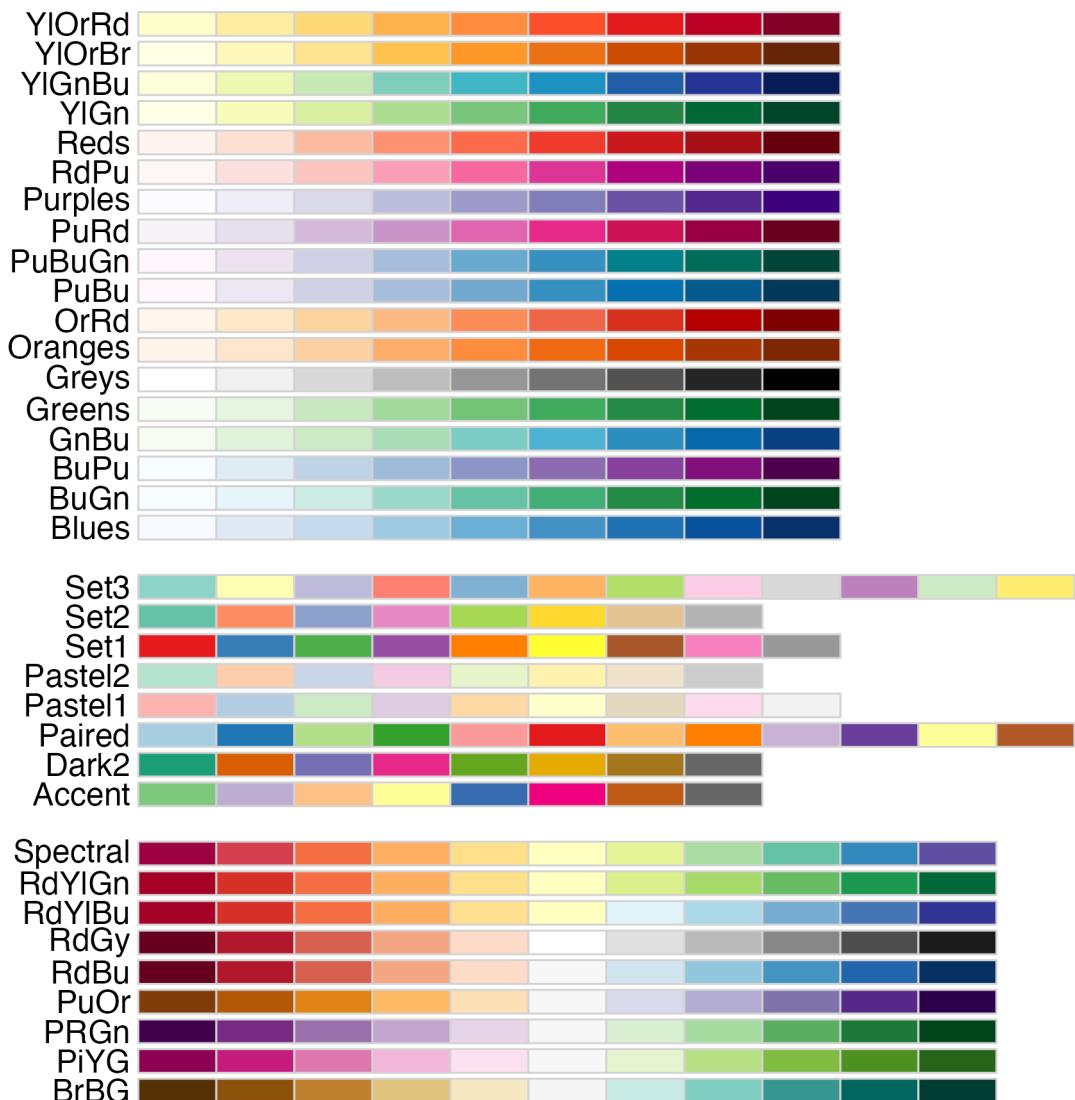


図 20.1: RColorBrewer が提供するパレート一覧

## 20.10 地図

### 20.10.1 世界地図

{ggplot2}で地図をプロットする方法は色々あります。理想としては各国政府が提供する地図データをダウンロードし、それを読み込み・加工してプロットすることでしょうが、ここではパッケージを使ったマッピングについて紹介します。

今回使用するパッケージは{rnatural-earth}、{rnatural-earthdata}、{rgeos}です。他にも使うパッケージはありますが、世界地図ならとりあえずこれで十分です。

```
1 pacman::p_load(rnaturalearth, rnaturalearthdata, rgeos)
```

世界地図を読み込む関数は{rnatural-earth}が提供する `ne_countries()` です。とりあえず指定する引数は `scale` と `returnclass` です。`scale` は地図の解像度であり、世界地図なら"small"で十分です。もう少し拡大される大陸地図なら"medium"が、一国だけの地図なら"large"が良いかも知れません。`returnclass` は"sf"と指定します。今回は低解像度の世界地図を sf クラスで読み込んでみましょう。

```
1 world_map <- ne_countries(scale = "small", returnclass = "sf")  
2  
3 class(world_map)
```

```
## [1] "sf"           "data.frame"
```

クラスは `data.frame` と `sf` であり、実際、`world_map` を出力してみると、見た目がデータフレームであることが分かります。地図の出力は `geom_sf()` 幾何オブジェクトを使用します。とりあえず、やってみましょう。

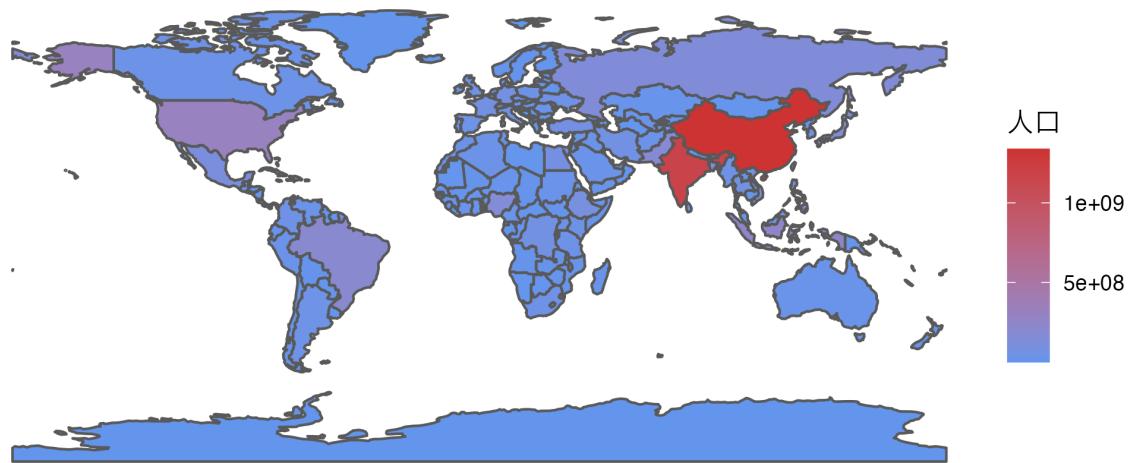
```
1 world_map %>%  
2   ggplot() +  
3   geom_sf() +  
4   theme_void() # 何もないテーマを指定する。ここはお好みで
```



もし、各国の人口に応じて色塗りをする場合はどうすれば良いでしょうか。実は、今回使用するデータがデータフレーム形式であることを考えると、これまでの{ggplot2}の使い方とあまり変わりません。{rnatural-earth}から読み込んだデータには既に `pop_est` という各国の人口データが含まれて負います。この値に応じて色塗りを行うため、`geom_sf()` 内に `fill = pop_est` でマッピングするだけです。

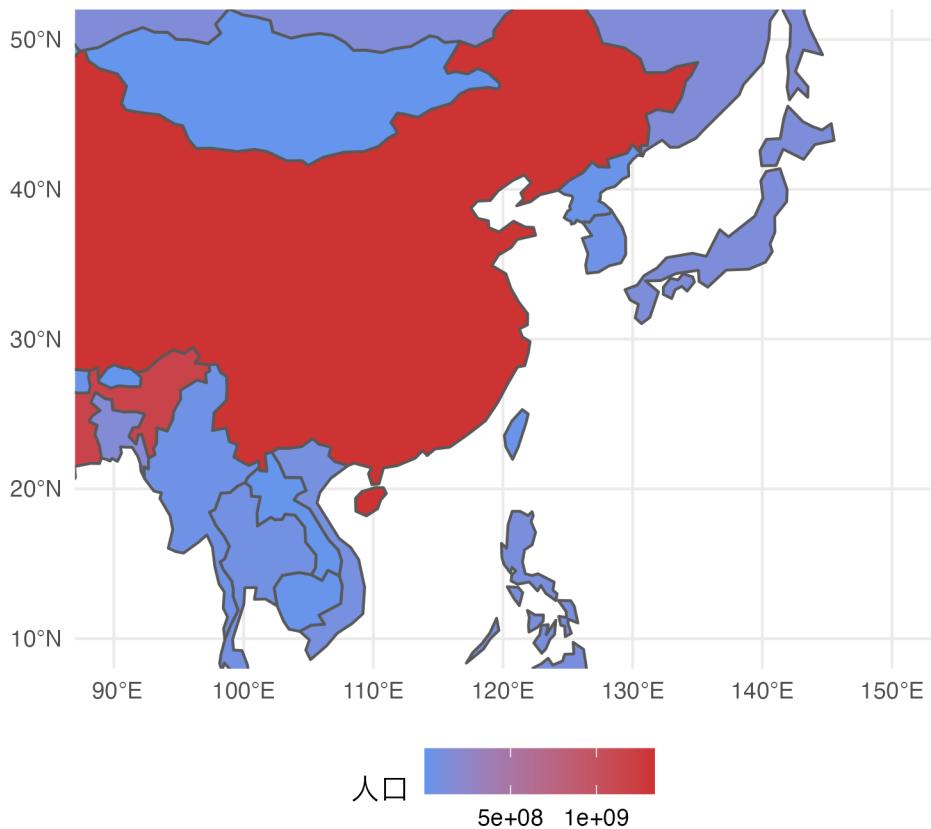
他にも自分で構築したデータがあるなら、データを結合して使用すれば良いでしょう。これについては後述します。

```
1 world_map %>%
2   ggplot() +
3   geom_sf(aes(fill = pop_est)) +
4   # 人口が少ない国は cornflowerblue 色に、多い国は brown3 色とする
5   scale_fill_gradient(low = "cornflowerblue", high = "brown3") +
6   labs(fill = "人口") +
7   theme_void()
```



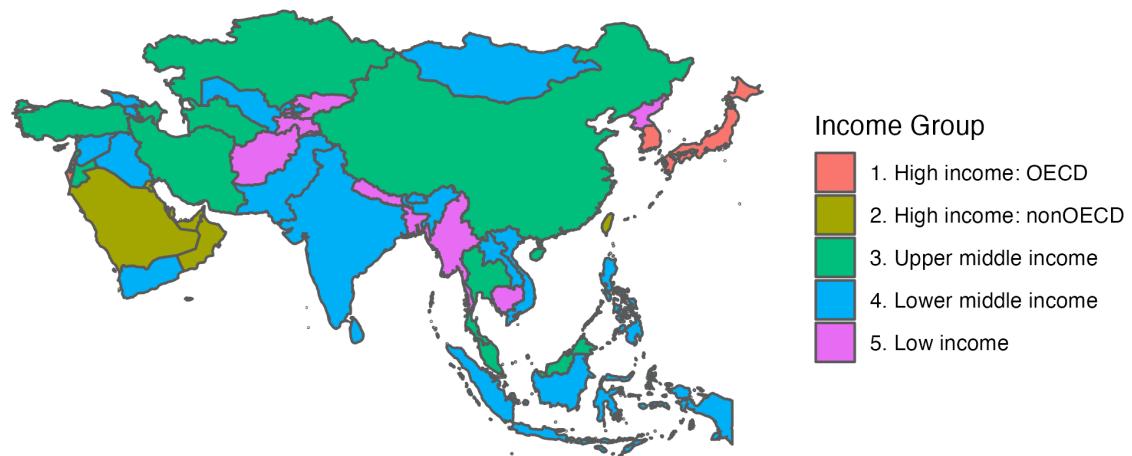
もし、世界でなく一部の地域だけを出力するなら、`coord_sf()` で座標系を調整します。東アジアと東南アジアの一部を出力したいとします。この場合、経度は 90 度から 150 度まで、緯度は 10 度から 50 度に絞ることになります。経度は `xlim` で、緯度は `ylim` で調整します。

```
1 world_map %>%
2   ggplot() +
3   geom_sf(aes(fill = pop_est)) +
4   scale_fill_gradient(low = "cornflowerblue", high = "brown3") +
5   labs(fill = "人口") +
6   coord_sf(xlim = c(90, 150), ylim = c(10, 50)) +
7   theme_minimal() +
8   theme(legend.position = "bottom")
```



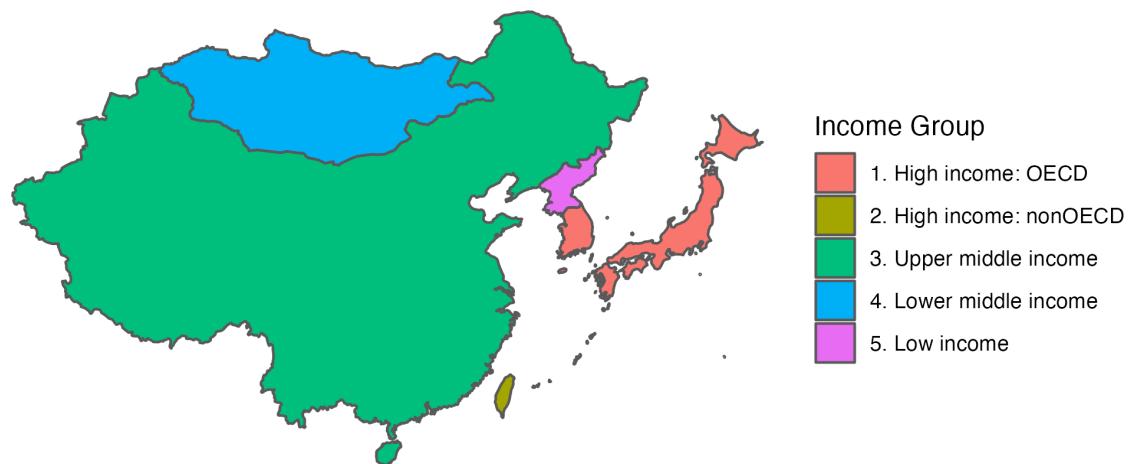
他にも `ne_countries()` 内に `continent` 引数を指定し、特定の大陸だけを読み込むことで可能です。ここではアジアの国のみを抽出し、`asia_map` という名のオブジェクトとして格納します。解像度は中程度とします。

```
1 asia_map <- ne_countries(scale = "medium", continent = "Asia",
2                             returnclass = "sf")
3
4 asia_map %>%
5   ggplot() +
6   # 所得グループで色塗り
7   geom_sf(aes(fill = income_grp)) +
8   theme_void() +
9   labs(fill = "Income Group")
```



アジアの中から更に東アジアに絞りたい場合は `filter()` を使用し、`subregion` 列を基準に抽出することも可能です。

```
1 asia_map %>%
2   filter(subregion == "Eastern Asia") %>%
3   ggplot() +
4   geom_sf(aes(fill = income_grp)) +
5   theme_void() +
6   labs(fill = "Income Group")
```



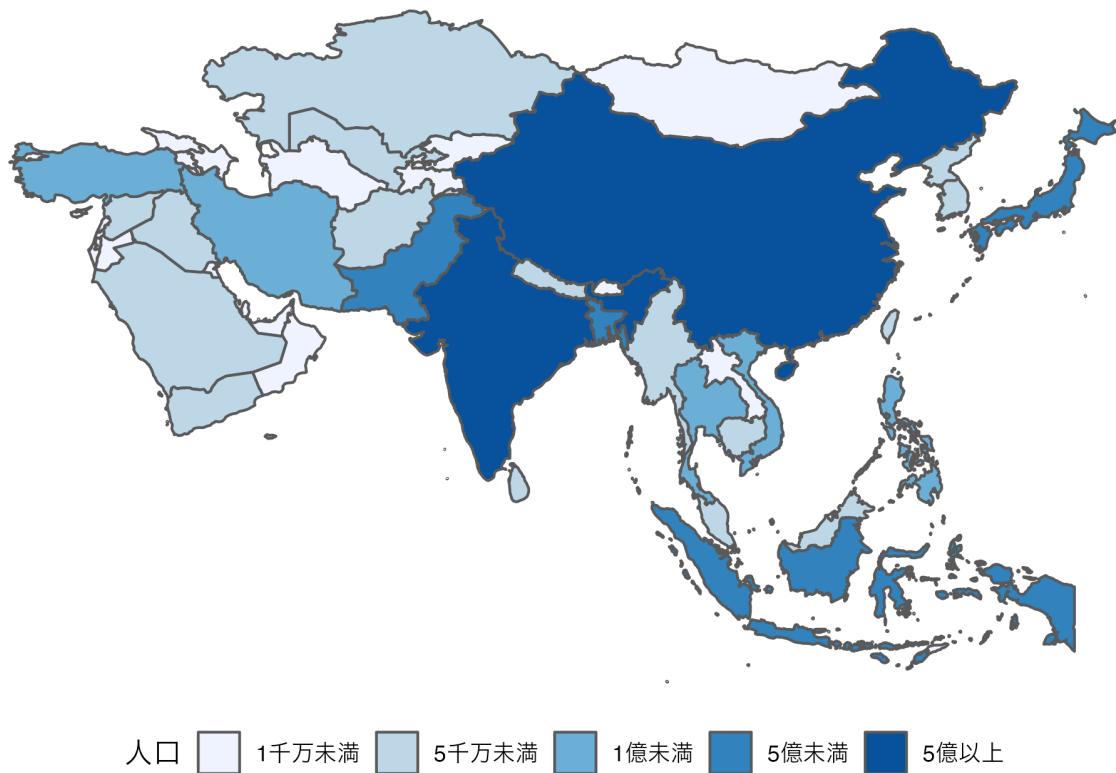
`subregion` の値は以下のように確認可能です。

```
1 unique(asia_map$subregion)

## [1] "Southern Asia"           "Western Asia"
## [3] "South-Eastern Asia"      "Eastern Asia"
## [5] "Seven seas (open ocean)" "Central Asia"
```

これまで使用してきたデータがデータフレームと同じ見た目をしているため、`{dplyr}`を用いたデータハンドリングも可能です。たとえば、人口を連続変数としてではなく、factor型に変換してからマッピングをしてみましょう。

```
1 asia_map %>%
2   mutate(Population = case_when(pop_est < 10000000 ~ "1 千万未満",
3                                 pop_est < 50000000 ~ "5 千万未満",
4                                 pop_est < 100000000 ~ "1 億未満",
5                                 pop_est < 500000000 ~ "5 億未満",
6                                 TRUE ~ "5 億以上"),
7   Population = factor(Population,
8                       levels = c("1 千万未満", "5 千万未満", "1 億未満",
9                       "5 億未満", "5 億以上"))) %>%
10  ggplot() +
11  geom_sf(aes(fill = Population)) +
12  scale_fill_brewer(palette = "Blues", drop = FALSE) +
13  labs(fill = "人口") +
14  theme_void() +
15  theme(legend.position = "bottom")
```

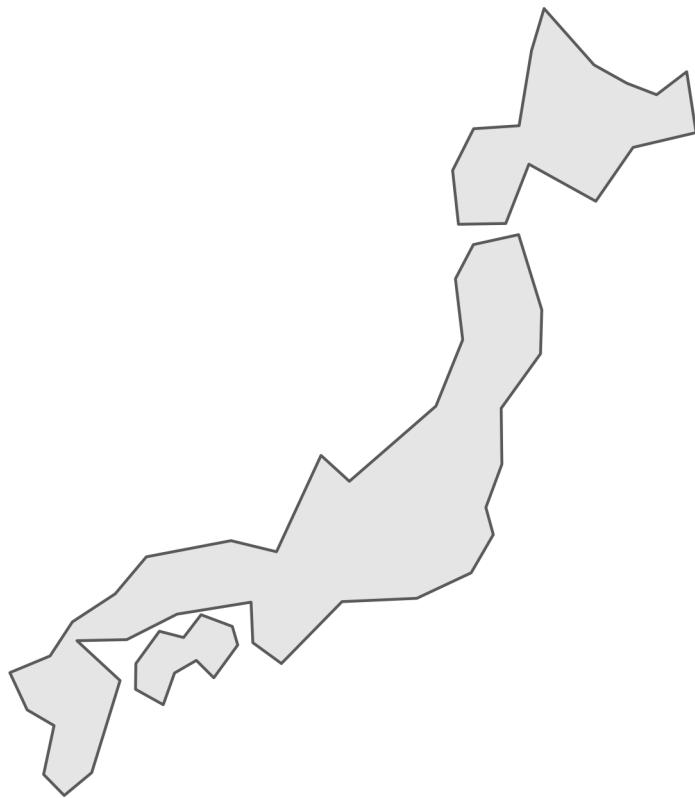


`scale_fill_brewer()` の `palette` 引数は等高線図のときに紹介しましたパレート一覧を参照してください。

### 20.10.2 日本地図（全体）

次は日本地図の出力についてです。日本全土だけを出力するなら、これまで使いました `ne_countries` に `country` 引数を指定するだけで使えます。たとえば、日本の地図だけなら、`country = "Japan"`を指定します。

```
1 ne_countries(scale = "small", country = "Japan", returnclass = "sf") %>%
2   ggplot() +
3   geom_sf() +
4   theme_void() # 空っぽのテーマ
```



これだと、物足りない感があるので、もう少し高解像度の地図にしてみましょう。高解像度の地図データを読み込む際は `scale = "large"` を指定します。

```
1 ne_countries(scale = "large", country = "Japan", returnclass = "sf") %>%
2   ggplot() +
3   geom_sf() +
4   theme_void() # 空っぽのテーマ
```



ただ、日本地図を出すという場合、多くは都道府県レベルでマッピングが目的でしょう。世界地図のマッピングならこれで問題ありませんが、一国だけなら、その下の自治体の境界線も必要です。したがって、先ほど使用しましたパッケージのより高解像度の地図が含まれている{rnatural earth hires}をインストールし、読み込みましょう。2022年Mar月03日現在、{rnatural earth hires}はCRANに登録されておらず、GitHubのropensciレポジトリのみで公開されているため、今回は{pacman}のp\_load()ではなく、p\_load\_gh()を使用します。

```
1 pacman::p_load_gh("ropensci/rnatural earth hires")
```

地図データの抽出にはne\_states()関数を使用します。第一引数として国家名を指定し、地図データのクラスはsfとします。抽出したデータの使い方は世界地図の時と同じです。

```
1 Japan_Map <- ne_states("Japan", returnclass = "sf")  
2  
3 Japan_Map %>%
```

```
4     ggplot() +  
5     geom_sf() +  
6     theme_void()
```



今回は各都道府県を人口密度ごとに色塗りをしてみましょう。`ne_states()` で読み込んだデータに人口密度のデータはないため、別途のデータと結合する必要があります。筆者が予め作成しておいたデータを読み込み、中身を確認してみます。

```
1 Japan_Density <- read_csv("Data/Japan_Density.csv")
```

```
## Rows: 47 Columns: 3
## -- Column specification:
## Delimiter: ","
## chr (1): Name
## dbl (2): Code, Density
##
```

```
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

1 Japan_Density

## # A tibble: 47 x 3
##       Code Name   Density
##       <dbl> <chr>   <dbl>
## 1       1 北海道    66.6
## 2       2 青森県   128.
## 3       3 岩手県    79.2
## 4       4 宮城県   316.
## 5       5 秋田県    82.4
## 6       6 山形県    115.
## 7       7 福島県   133
## 8       8 茨城県   470.
## 9       9 栃木県   302.
## 10     10 群馬県   305.
## # ... with 37 more rows
```

各都道府県の人口密度がついております、左側の Code は何でしょうか。これは各都道府県の ISO コードであり、このコードをキー変数としてデータを結合することとなります。各都道府県のコードは国土交通省のホームページから確認可能です。

それではデータを結合してみましょう。ne\_states() で読み込んだデータの場合、地域のコードは iso\_3166\_2 という列に格納されています。

```
1 Japan_Map$iso_3166_2
```

```
## [1] "JP-46" "JP-44" "JP-40" "JP-41" "JP-42" "JP-43" "JP-45" "JP-
36" "JP-37"
## [10] "JP-38" "JP-39" "JP-32" "JP-35" "JP-31" "JP-28" "JP-26" "JP-
18" "JP-17"
## [19] "JP-16" "JP-15" "JP-06" "JP-05" "JP-02" "JP-03" "JP-04" "JP-
07" "JP-08"
```

```

## [28] "JP-12" "JP-13" "JP-14" "JP-22" "JP-23" "JP-24" "JP-30" "JP-
27" "JP-33"
## [37] "JP-34" "JP-01" "JP-47" "JP-10" "JP-20" "JP-09" "JP-21" "JP-
25" "JP-11"
## [46] "JP-19" "JP-29"

```

こちらは文字列となっていますね。これを左から4番目の文字から切り取り、数値型に変換します。変換したコードは結合のために `Code` という名の列として追加しましょう。

```

1 Japan_Map <- Japan_Map %>%
2   mutate(Code = str_sub(iso_3166_2, 4),
3         Code = as.numeric(Code))
4
5 Japan_Map$Code

```

```

## [1] 46 44 40 41 42 43 45 36 37 38 39 32 35 31 28 26 18 17 16 15 6 5 2 3 4
## [26] 7 8 12 13 14 22 23 24 30 27 33 34 1 47 10 20 9 21 25 11 19 29

```

続いて、`Japan_Map` と `Japan_Density` を `Code` 列をキー変数として結合します。データの中身を確認すると、`Density` 列が最後(の直前)の列に追加されたことが分かります。

```

1 Japan_Map <- left_join(Japan_Map, Japan_Density, by = "Code")
2
3 Japan_Map

```

```

## Simple feature collection with 47 features and 86 fields
## Geometry type: MULTIPOLYGON
## Dimension: XY
## Bounding box: xmin: 122.9382 ymin: 24.2121 xmax: 153.9856 ymax: 45.52041
## Geodetic CRS: SOURCECRS
## First 10 features:
##           featurecla scalerank adm1_code diss_me iso_3166_2 wikipedia iso_a2
## 1 Admin-1 scale rank          2 JPN-3501      3501    JP-46      <NA>    JP
## 2 Admin-1 scale rank          6 JPN-1835      1835    JP-44      <NA>    JP
## 3 Admin-1 scale rank          6 JPN-1829      1829    JP-40      <NA>    JP

```

```

## 4 Admin-1 scale rank      6 JPN-1827 1827  JP-41  <NA>  JP
## 5 Admin-1 scale rank      2 JPN-3500 3500  JP-42  <NA>  JP
## 6 Admin-1 scale rank      6 JPN-1830 1830  JP-43  <NA>  JP
## 7 Admin-1 scale rank      6 JPN-1831 1831  JP-45  <NA>  JP
## 8 Admin-1 scale rank      6 JPN-1836 1836  JP-36  <NA>  JP
## 9 Admin-1 scale rank      6 JPN-1833 1833  JP-37  <NA>  JP
## 10 Admin-1 scale rank     6 JPN-1832 1832  JP-38  <NA>  JP

##      adm0_sr      name name_alt name_local type      type_en code_local code_hasc
## 1      5 Kagoshima      <NA>      <NA>  Ken  Prefecture      <NA>  JP.KS
## 2      1 Oita          <NA>      <NA>  Ken  Prefecture      <NA>  JP.OT
## 3      1 Fukuoka      Hukuoka      <NA>  Ken  Prefecture      <NA>  JP.FO
## 4      1 Saga          <NA>      <NA>  Ken  Prefecture      <NA>  JP.SG
## 5      3 Nagasaki      <NA>      <NA>  Ken  Prefecture      <NA>  JP.NS
## 6      1 Kumamoto      <NA>      <NA>  Ken  Prefecture      <NA>  JP.KM
## 7      1 Miyazaki      <NA>      <NA>  Ken  Prefecture      <NA>  JP.MZ
## 8      1 Tokushima    Tokusima      <NA>  Ken  Prefecture      <NA>  JP.TS
## 9      1 Kagawa         <NA>      <NA>  Ken  Prefecture      <NA>  JP.KG
## 10     4 Ehime          <NA>      <NA>  Ken  Prefecture      <NA>  JP.EH

##      note hasc_maybe region region_cod provnum_ne gadm_level check_me datarank
## 1 <NA>  JP.NR  Kyushu  JPN-KYS      3      1    20      9
## 2 <NA>  JP.ON  Kyushu  JPN-SHK     48      1    20      2
## 3 <NA>  JP.NS  Kyushu  JPN-KYS     46      1    20      2
## 4 <NA>  JP.OS  <NA>      <NA>     47      1    20      2
## 5 <NA>  JP.OY  <NA>      <NA>      5      1    20      9
## 6 <NA>  JP.NI  Kyushu  JPN-KYS      6      1    20      2
## 7 <NA>  JP.OT  Kyushu  JPN-KYS     49      1    20      2
## 8 <NA>  JP.SZ  Shikoku  JPN-SHK     45      1    20      2
## 9 <NA>  JP.SH  Shikoku  JPN-SHK      4      1    20      2
## 10 <NA>  JP.ST  Shikoku  JPN-SHK     14      1    20      2

##      abbrev postal area_sqkm sameascity labelrank name_len mapcolor9 mapcolor13
## 1 <NA>  <NA>      0      NA      2      9      5      4
## 2 <NA>  OT        0       7      7      4      5      4

```

```

## 3 <NA> FO 0 7 7 7 5 4
## 4 <NA> SG 0 NA 6 4 5 4
## 5 <NA> <NA> 0 NA 2 8 5 4
## 6 <NA> KM 0 NA 6 8 5 4
## 7 <NA> MZ 0 NA 6 8 5 4
## 8 <NA> TS 0 NA 6 9 5 4
## 9 <NA> KG 0 NA 6 6 5 4
## 10 <NA> EH 0 NA 6 5 5 4
## fips fips_alt woe_id woe_label woe_name latitude
## 1 JA18 JA28 2345867 Kagoshima Prefecture, JP, Japan Kagoshima 29.4572
## 2 JA30 JA47 2345879 Oita Prefecture, JP, Japan Oita 33.2006
## 3 JA07 JA27 58646425 Fukuoka Prefecture, JP, Japan Fukuoka 33.4906
## 4 JA33 JA32 2345882 Saga Prefecture, JP, Japan Saga 33.0097
## 5 JA27 JA31 2345876 Nagasaki Prefecture, JP, Japan Nagasaki 32.6745
## 6 JA21 JA29 2345870 Kumamoto Prefecture, JP, Japan Kumamoto 32.5880
## 7 JA25 JA30 2345874 Miyazaki Prefecture, JP, Japan Miyazaki 32.0981
## 8 JA39 JA37 2345888 Tokushima Prefecture, JP, Japan Tokushima 33.8546
## 9 JA17 JA35 2345866 Kagawa Prefecture, JP, Japan Kagawa 34.2162
## 10 JA05 JA34 2345855 Ehime Prefecture, JP, Japan Ehime 33.8141
## longitude sov_a3 adm0_a3 adm0_label admin geonunit gu_a3 gn_id
## 1 129.601 JPN JPN 4 Japan Japan JPN 1860825
## 2 131.449 JPN JPN 4 Japan Japan JPN 1854484
## 3 130.616 JPN JPN 4 Japan Japan JPN 1863958
## 4 130.147 JPN JPN 4 Japan Japan JPN 1853299
## 5 128.755 JPN JPN 4 Japan Japan JPN 1856156
## 6 130.834 JPN JPN 4 Japan Japan JPN 1858419
## 7 131.286 JPN JPN 4 Japan Japan JPN 1856710
## 8 134.200 JPN JPN 4 Japan Japan JPN 1850157
## 9 134.001 JPN JPN 4 Japan Japan JPN 1860834
## 10 132.916 JPN JPN 4 Japan Japan JPN 1864226
## gn_name gns_id gns_name gn_level gn_region gn_a1_code region_sub
## 1 Kagoshima-ken -231556 Kagoshima-ken 1 <NA> JP.18 <NA>

```

```

## 2      Oita-ken -240089      Oita-ken      1      <NA>      JP.30      <NA>
## 3      Fukuoka-ken -227382      Fukuoka-ken      1      <NA>      JP.07      <NA>
## 4      Saga-ken -241905      Saga-ken      1      <NA>      JP.33      <NA>
## 5      Nagasaki-ken -237758      Nagasaki-ken      1      <NA>      JP.27      <NA>
## 6      Kumamoto-ken -234759      Kumamoto-ken      1      <NA>      JP.21      <NA>
## 7      Miyazaki-ken -236958      Miyazaki-ken      1      <NA>      JP.25      <NA>
## 8      Tokushima-ken -246216      Tokushima-ken      1      <NA>      JP.39      <NA>
## 9      Kagawa-ken -231546      Kagawa-ken      1      <NA>      JP.17      <NA>
## 10     Ehime-ken -227007      Ehime-ken      1      <NA>      JP.05      <NA>
##      sub_code gns_level gns_lang gns_adm1 gns_region min_label max_label min_zoom
## 1      <NA>          1      jpn      JA18      <NA>          7          11          3
## 2      <NA>          1      jpn      JA30      <NA>          7          11          3
## 3      <NA>          1      jpn      JA07      <NA>          7          11          3
## 4      <NA>          1      jpn      JA33      <NA>          7          11          3
## 5      <NA>          1      jpn      JA27      <NA>          7          11          3
## 6      <NA>          1      jpn      JA21      <NA>          7          11          3
## 7      <NA>          1      jpn      JA25      <NA>          7          11          3
## 8      <NA>          1      jpn      JA39      <NA>          7          11          3
## 9      <NA>          1      jpn      JA17      <NA>          7          11          3
## 10     <NA>          1      jpn      JA05      <NA>          7          11          3
##      wikidataid name_ar name_bn          name_de          name_en
## 1      Q15701      <NA>      <NA> Präfektur Kagoshima Kagoshima Prefecture
## 2      Q133924      <NA>      <NA> Präfektur Oita      Oita Prefecture
## 3      Q123258      <NA>      <NA> Präfektur Fukuoka Fukuoka Prefecture
## 4      Q160420      <NA>      <NA> Präfektur Saga      Saga Prefecture
## 5      Q169376      <NA>      <NA> Präfektur Nagasaki Nagasaki Prefecture
## 6      Q130308      <NA>      <NA> Präfektur Kumamoto Kumamoto Prefecture
## 7      Q130300      <NA>      <NA> Präfektur Miyazaki Miyazaki Prefecture
## 8      Q160734      <NA>      <NA> Präfektur Tokushima Tokushima Prefecture
## 9      Q161454      <NA>      <NA> Präfektur Kagawa      Kagawa Prefecture
## 10     Q123376      <NA>      <NA> Präfektur Ehime      Ehime Prefecture
##          name_es          name_fr name_el name_hi

```

```

## 1 Prefectura de Kagoshima Préfecture de Kagoshima <NA> <NA>
## 2 Prefectura de Oita Préfecture d'Oita <NA> <NA>
## 3 Prefectura de Fukuoka Préfecture de Fukuoka <NA> <NA>
## 4 Prefectura de Saga Préfecture de Saga <NA> <NA>
## 5 Prefectura de Nagasaki Préfecture de Nagasaki <NA> <NA>
## 6 Prefectura de Kumamoto Préfecture de Kumamoto <NA> <NA>
## 7 Prefectura de Miyazaki Préfecture de Miyazaki <NA> <NA>
## 8 Prefectura de Tokushima Préfecture de Tokushima <NA> <NA>
## 9 Prefectura de Kagawa Préfecture de Kagawa <NA> <NA>
## 10 Prefectura de Ehime Préfecture d'Ehime <NA> <NA>
## name_hu name_id name_it name_ja
## 1 Kagosima prefektúra Prefektur Kagoshima prefettura di Kagoshima <NA>
## 2 Óita prefektúra Prefektur Oita prefettura di Oita <NA>
## 3 Fukuoka prefektúra Prefektur Fukuoka prefettura di Fukuoka <NA>
## 4 Szaga prefektúra Prefektur Saga Prefettura di Saga <NA>
## 5 Nagaszaki prefektúra Prefektur Nagasaki prefettura di Nagasaki <NA>
## 6 Kumamoto prefektúra Prefektur Kumamoto prefettura di Kumamoto <NA>
## 7 Miyazaki prefektúra Prefektur Miyazaki prefettura di Miyazaki <NA>
## 8 Tokusima prefektúra Prefektur Tokushima prefettura di Tokushima <NA>
## 9 Kagawa prefektúra Prefektur Kagawa prefettura di Kagawa <NA>
## 10 Ehime prefektúra Prefektur Ehime prefettura di Ehime <NA>
## name_ko name_nl name_pl name_pt name_ru name_sv
## 1 <NA> Kagoshima Prefektura Kagoshima Kagoshima <NA> Kagoshima prefektur
## 2 <NA> Oita Prefektura Oita Oita <NA> Oita prefektur
## 3 <NA> Fukuoka Prefektura Fukuoka Fukuoka <NA> Fukuoka prefektur
## 4 <NA> Saga Prefektura Saga Saga <NA> Saga prefektur
## 5 <NA> Nagasaki Prefektura Nagasaki Nagasaki <NA> Nagasaki prefektur
## 6 <NA> Kumamoto Prefektura Kumamoto Kumamoto <NA> Kumamoto prefektur
## 7 <NA> Miyazaki Prefektura Miyazaki Miyazaki <NA> Miyazaki prefektur
## 8 <NA> Tokushima Prefektura Tokushima Tokushima <NA> Tokushima prefektur
## 9 <NA> Kagawa Prefektura Kagawa Kagawa <NA> Kagawa prefektur
## 10 <NA> Ehime Prefektura Ehime Ehime <NA> Ehime prefektur

```

```

##          name_tr   name_vi name_zh      ne_id Code      Name Density
## 1   Kagoshima ili Kagoshima      <NA> 1159315225    46 鹿児島県  172.9
## 2        Oita      Oita      <NA> 1159311905    44 大分県  177.2
## 3      Fukuoka    Fukuoka      <NA> 1159311899    40 福岡県 1029.8
## 4        Saga      Saga      <NA> 1159311895    41 佐賀県  332.5
## 5   Nagasaki Nagasaki      <NA> 1159315235    42 長崎県  317.7
## 6   Kumamoto Kumamoto      <NA> 1159311901    43 熊本県  234.6
## 7 Miyazaki Miyazaki      <NA> 1159311903    45 宮崎県  138.3
## 8 Tokushima Tokushima      <NA> 1159311909    36 徳島県  173.5
## 9   Kagawa   Kagawa      <NA> 1159311907    37 香川県  506.3
## 10   Ehime   Ehime      <NA> 1159311139    38 愛媛県  235.2
##                               geometry
## 1 MULTIPOLYGON (((129.7832 31...
## 2 MULTIPOLYGON (((131.2009 33...
## 3 MULTIPOLYGON (((130.0363 33...
## 4 MULTIPOLYGON (((129.8145 33...
## 5 MULTIPOLYGON (((130.2041 32...
## 6 MULTIPOLYGON (((130.3446 32...
## 7 MULTIPOLYGON (((131.8723 32...
## 8 MULTIPOLYGON (((134.4424 34...
## 9 MULTIPOLYGON (((133.5919 34...
## 10 MULTIPOLYGON (((132.6399 32...

```

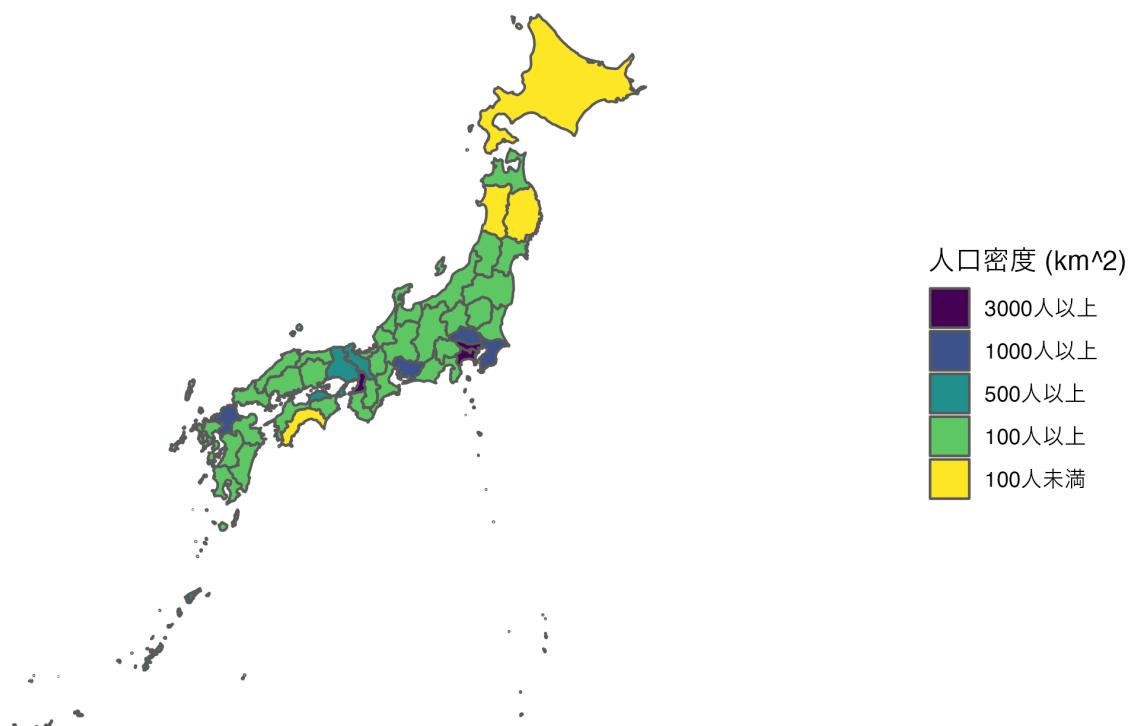
それではマッピングをしてみましょう。人口密度を5つのカテゴリーに順序付き factor 化してから、そのカテゴリーに応じて色塗りをします。

```

1 Japan_Map %>%
2   mutate(Density2 = case_when(Density >= 3000 ~ "3000人以上",
3                                Density >= 1000 ~ "1000人以上",
4                                Density >=  500 ~ "500人以上",
5                                Density >=  100 ~ "100人以上",
6                                TRUE           ~ "100人未満"),
7   Density2 = factor(Density2, ordered = TRUE,

```

```
8      levels = c("3000 人以上", "1000 人以上", "500 人以上",
9          "100 人以上", "100 人未満")) %>%
10
11     ggplot() +
12     geom_sf(aes(fill = Density2)) +
13     labs(fill = "人口密度 (km^2)") +
14     theme_void()
```



世界地図でも同じやり方でデータの結合が可能です。この場合は ISO3 コードか ISO2 コードがキー変数となります。ISO3 コードは `iso_a3`、ISO2 コードは `iso_a2` 列に格納されています。他に使用可能なキー変数は `iso_n3` であり、こちらは各国を識別する 3 桁の数字となります。

### 20.10.3 日本地図（特定の都道府県）

また日本地図のマッピングですが、今回は市区町村レベルまで見てみましょう。`ne_states()` では市区町村までマッピングすることはできませんので、今回は徳島大学

の瓜生真也先生が公開しました{jpn\_district}を使います。

```
1 pacman::p_load_gh("uribo/jpn_district")
```

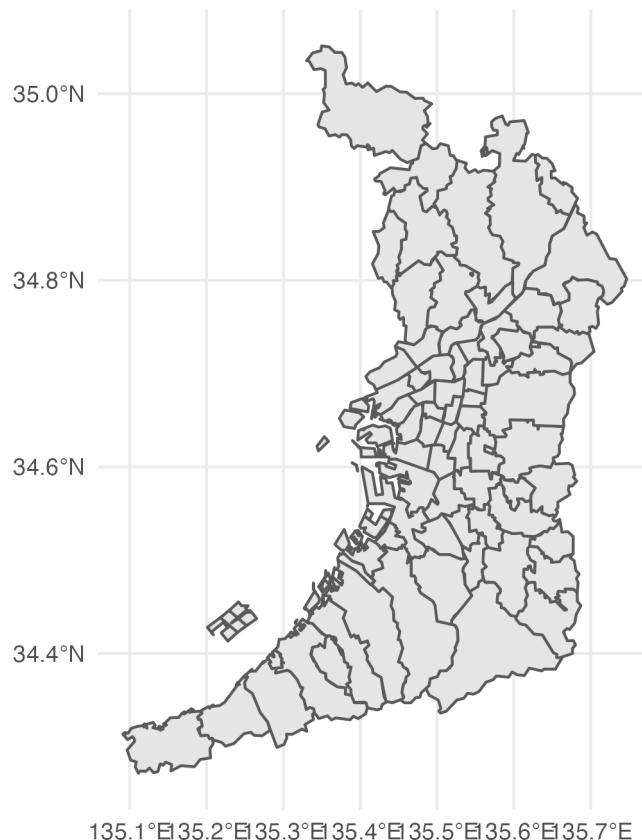
今回は大阪府の地図を出力してみましょう。特定の都道府県の地図を読み込むためには `jpn_pref()` 関数を使用します。都道府県は `pref_code` または `admin_name` で指定します。大阪のコードは 27 であるため、`pref_code = 27` でも良いですし、`admin_name = "大阪府"` でも同じです。

```
1 # Osaka_map <- jpn_pref(admin_name = "大阪府") でも同じ
2 Osaka_map <- jpn_pref(pref_code = 27)
3
4 class(Osaka_map)
```

```
## [1] "sf"           "tbl_df"        "tbl"           "data.frame"
```

プロットの方法は同じです。

```
1 Osaka_map %>%
2   ggplot() +
3   geom_sf() +
4   theme_minimal()
```



ここでもデータの結合&マッピングが可能です。大阪府内自治体の人口と学生数が格納されたデータを読み込んでみましょう。こちらは2015年国勢調査の結果から取得したデータです。

```
1 Osaka_Student <- read_csv("data/Osaka_Student.csv")  
  
## # Rows: 75 Columns: 4  
## -- Column specification -----  
## Delimiter: ","  
## chr (1): Name  
## dbl (3): Code, Pop, Student  
##  
## i Use `spec()` to retrieve the full column specification for this data.  
## i Specify the column types or set `show_col_types = FALSE` to quiet this message
```

## 1 Osaka\_Student

```
## # A tibble: 75 x 4
##   Code Name          Pop Student
##   <dbl> <chr>        <dbl>   <dbl>
## 1 27000 大阪府      8839469  438901
## 2 27100 大阪市      2691185  104208
## 3 27102 大阪市 都島区 104727   3889
## 4 27103 大阪市 福島区 72484    2448
## 5 27104 大阪市 此花区 66656    2478
## 6 27106 大阪市 西区  92430    2633
## 7 27107 大阪市 港区  82035    3072
## 8 27108 大阪市 大正区 65141    2627
## 9 27109 大阪市 天王寺区 75729    3480
## 10 27111 大阪市 浪速区 69766   1409
## # ... with 65 more rows
```

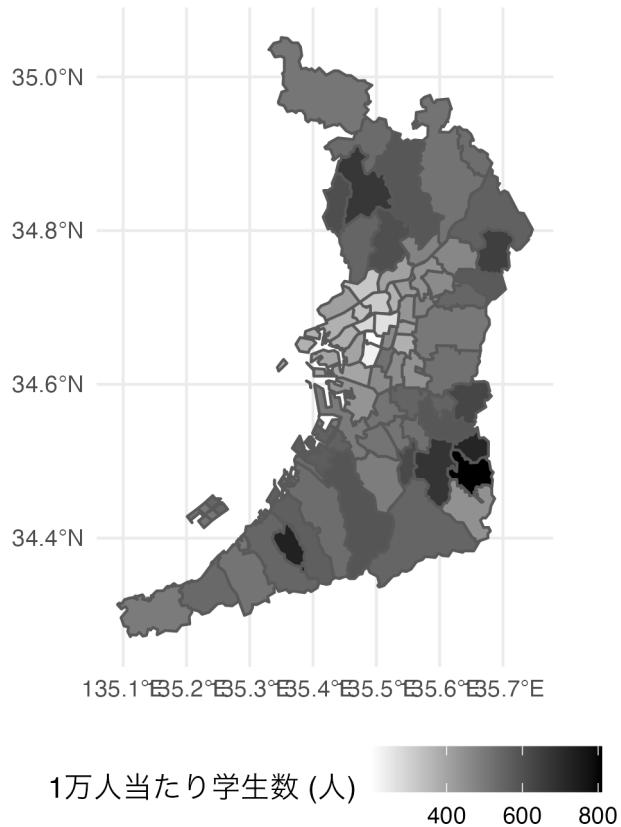
各市区町村にもコードが指定されており、Osaka\_Student では Code 列、Osaka\_map では citi\_code 列となります。Osaka\_map の city\_code は文字列であるため、こちらを数値型に変換し Code という名の列として追加しておきましょう。続いて、Code 列をキー変数とし、2 つのデータセットを結合します。

```
1 Osaka_map <- Osaka_map %>%
2   mutate(Code = as.numeric(city_code))
3
4 Osaka_map <- left_join(Osaka_map, Osaka_Student, by = "Code")
```

最後にマッピングです。ここでは人口 1 万人当たり学生数を Student\_Ratio という列として追加し、こちらの値に合わせて色塗りをしてみましょう。scale\_fill\_gradient() を使用し、人口 1 万人当たり学生数が少ないほど白、多いほど黒塗りします。

```
1 Osaka_map %>%
2   mutate(Student_Ratio = Student / Pop * 10000) %>%
3   ggplot() +
```

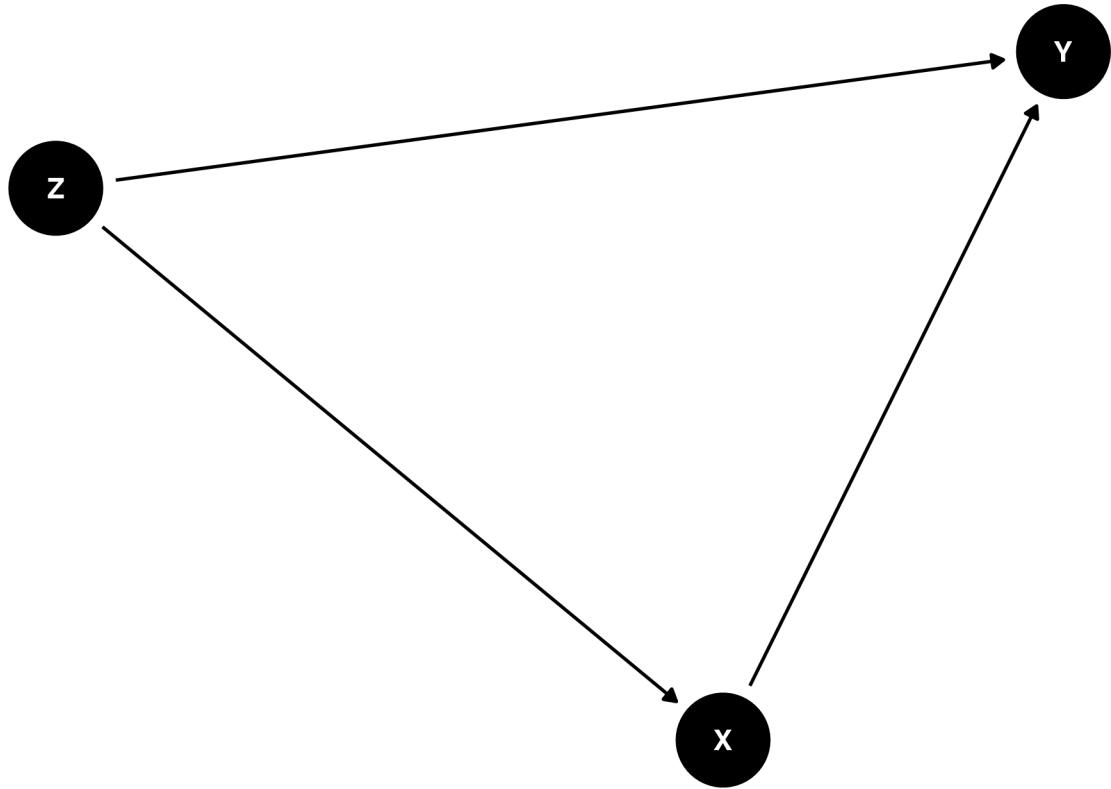
```
4  geom_sf(aes(fill = Student_Ratio)) +
5  scale_fill_gradient(low = "white", high = "black") +
6  labs(fill = "1 万人当たり学生数 (人)") +
7  theme_minimal() +
8  theme(legend.position = "bottom")
```



## 20.11 非巡回有向グラフ

近年、因果推論の界隈でよく登場する非巡回有向グラフ（DAG）ですが、「グラフ」からも分かるように、DAG の考え方に基づく因果推論の研究には多くの図が登場します。DAG を作図するには{ggplot2}のみでも可能ですが、{dagitty}パッケージで DAG の情報を含むオブジェクトを生成し、{ggdag}で作図した方が簡単です。以下の図は DAG の

一例です。



ここで X、Y、Z はノード (node) と呼ばれ、それぞれのノードをつなぐ線のことをエッジ (edge) と呼びます。また、これらのエッジには方向があります (有向)。簡単に言うと原因と結果といった関係ですが、DAG を描く際は、各ノード間の関係を記述する必要があります。

それではまず、以上の図を作ってみましょう。最初のステップとして {dagitty} と {ggdag} をインストールし、読み込みましょう。

```
1 pacman::p_load(dagitty, ggdag)
```

つづいて、DAG の情報を含むオブジェクトを作成します。使用する関数は `dagify()` であり、ここには結果 ~ 原因の形で各ノード間の関係を記述します。先ほどの図では X は Y の原因 ( $X \sim Y$ )、Z は X と Y の原因 ( $X \sim Z$  と  $Y \sim Z$ ) です。これらの情報を `dagify()` 内で指定します。

```
1 DAG_data1 <- dagify(X ~ Z,
2                         Y ~ Z,
3                         Y ~ X,
4                         exposure = "X",
5                         outcome = "Y")
6
7 DAG_data1
```

```
## dag {
## X [exposure]
## Y [outcome]
## Z
## X -> Y
## Z -> X
## Z -> Y
## }
```

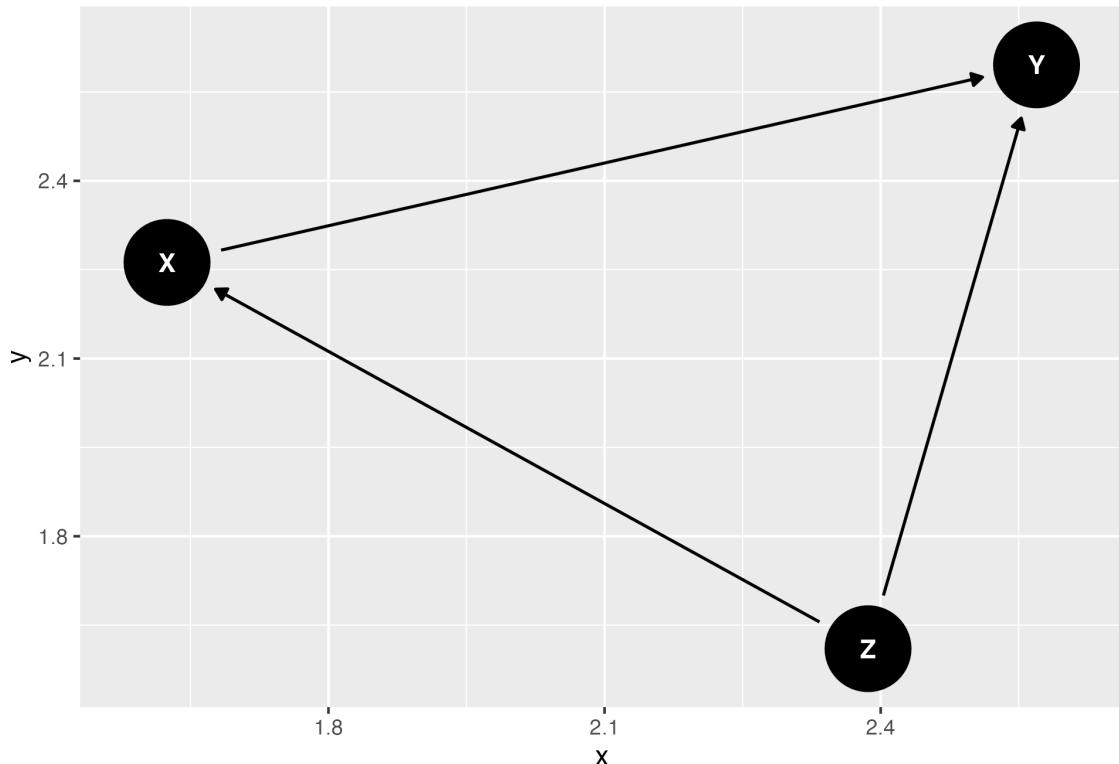
$Y \sim X$  と  $Y \sim Z$  は  $Y \sim X + Z$  とまとめることも可能です。これは「 $Y$  の原因は  $X$  と  $Z$  である」という意味であり、「 $Y$  の原因は  $X$  であり、 $Y$  の原因は  $Z$  である」と同じ意味です。また、DAG を作図する際、`dagify()` 内に `exposure` と `outcome` は不要ですが、もし `adjustmentSets()` 関数などを使って統制変数を特定したい場合は処置変数 (`exposure`) と応答変数 (`outcome`) にそれぞれ変数名を指定します。ちなみに、以上のコードは以下のように書くことも可能です。

```
1 DAG_data1 <- dagitty(
2   "dag{
3     X -> Y
4     X <- Z -> Y
5     X [exposure]
6     Y [outcome]
7   }")
8
9 DAG_data1
```

```
## dag {  
## X [exposure]  
## Y [outcome]  
## Z  
## X -> Y  
## Z -> X  
## Z -> Y  
## }
```

格納された DAG\_data1 オブジェクトのクラスは"dagitty"です。"dagitty"の可視化には{ggdag}の ggdag() を使用します。

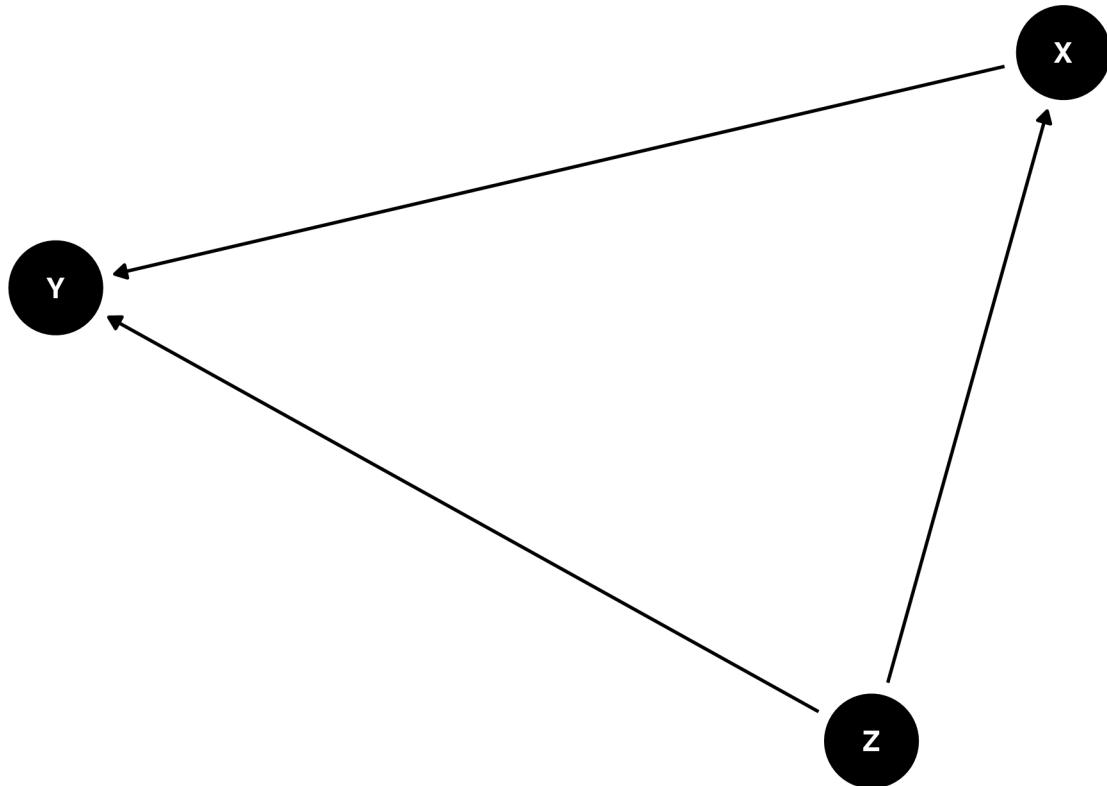
```
1 DAG_data1 %>%  
2 ggdag()
```



DAGにおいて背景、軸の目盛り、ラベルは不要ですので、theme\_dag\_blank() テー

マを指定して全て除去します。

```
1 DAG_data1 %>%
2   ggdag() +
3   theme_dag_blank()
```



### 20.11.1 ノードの位置を指定する

読者の多くは以上のグラフと異なるものが得られたかも知れません。ノード間の関係は同じはずですが、ノードの位置が異なるでしょう。また、同じコードを実行する度にノードの位置は変わります。以下ではノードの位置を固定する方法について紹介します。位置を指定するには `dagify()` 内で `coords` 引数に各ノードの情報が格納されたリスト型オブジェクトを指定する必要があります。リストの長さは 2 であり、それぞれの名前は `x` と `y` です。そしてリストの各要素にはベクトルが入ります。たとえば、ノード X の位置を  $(1, 1)$ 、Y の位置を  $(3, 1)$ 、Z の位置を  $(2, 2)$  に指定してみましょう。`dagify()` 内で直接リストを書くことも可能ですが、コードの可読性が落ちるため、別途のオブジェ

クト (DAG\_Pos2) として格納しておきます。

```
1 DAG_Pos2 <- list(x = c(X = 1, Y = 3, Z = 2),
2                     y = c(X = 1, Y = 1, Z = 2))
```

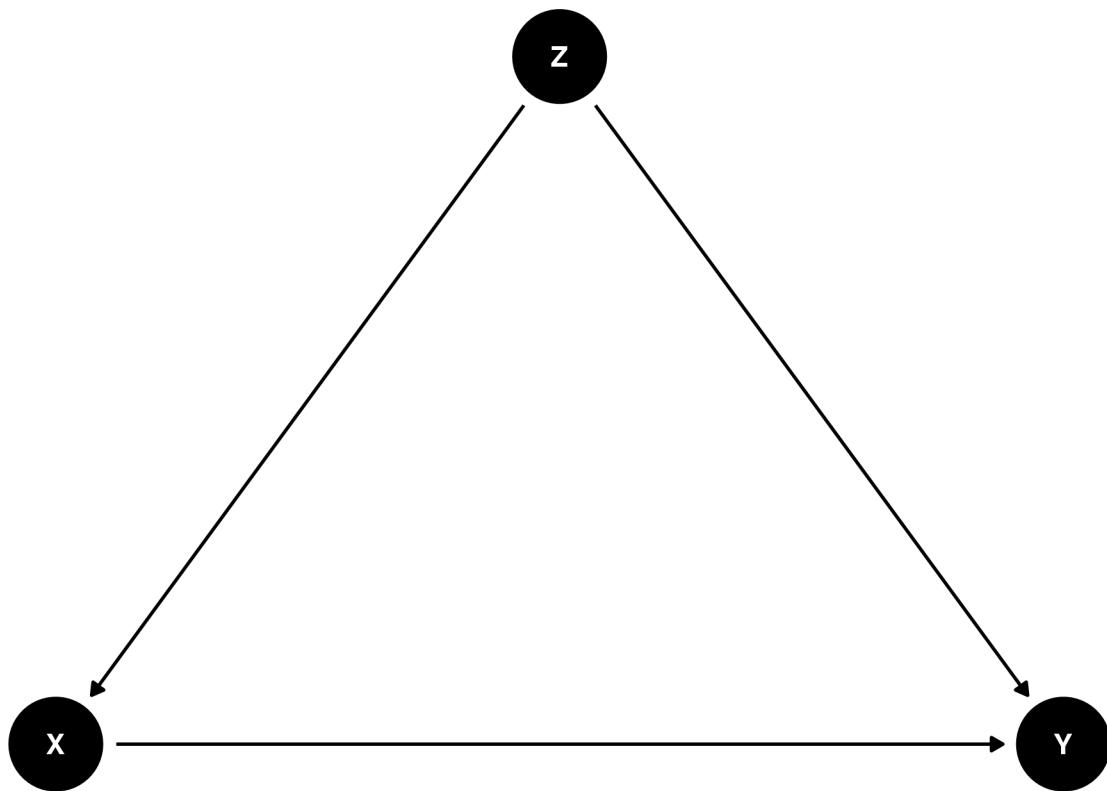
続いて、dagify() 内で coords 引数を追加し、ノードの位置情報が格納されている DAG\_Pos2 を指定します。

```
1 DAG_data2 <- dagify(X ~ Z,
2                       Y ~ X + Z,
3                       exposure = "X",
4                       outcome = "Y",
5                       coords = DAG_Pos2)
6
7 DAG_data2
```

```
## dag {
## X [exposure, pos="1.000,1.000"]
## Y [outcome, pos="3.000,1.000"]
## Z [pos="2.000,2.000"]
## X -> Y
## Z -> X
## Z -> Y
## }
```

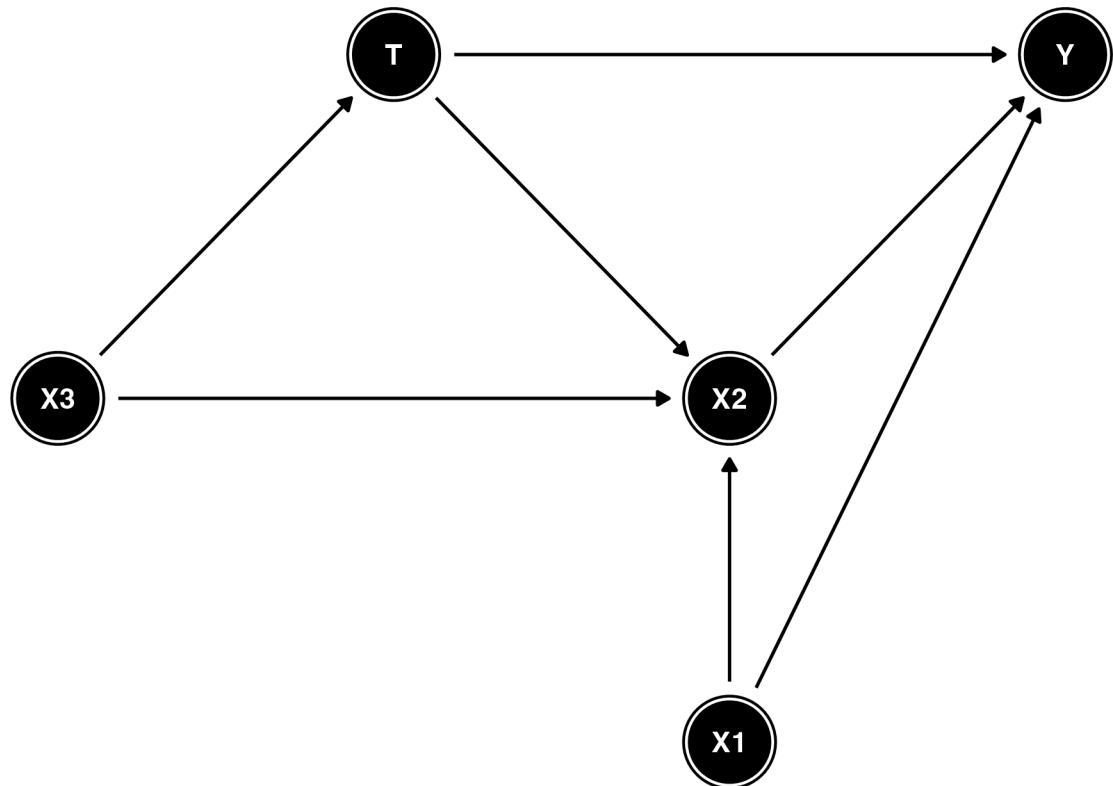
可視化の方法は同じです。

```
1 DAG_data2 %>%
2   ggdag() +
3   theme_dag_blank()
```



以上の使い方だけでも、ほとんどの DAG は描けるでしょう。また、ノードを若干オシャレ (?) にするには、`ggdag()` 内で `stylized = TRUE` を指定します。

```
1 DAG_Pos3 <- list(x = c(X1 = 3, X2 = 3, X3 = 1, T = 2, Y = 4),
2                         y = c(X1 = 1, X2 = 2, X3 = 2, T = 3, Y = 3))
3
4 DAG_data3 <- dagify(Y ~ T + X1 + X2,
5                         T ~ X3,
6                         X2 ~ T + X1 + X3,
7                         exposure = "T",
8                         outcome = "Y",
9                         coords = DAG_Pos3)
10
11 DAG_data3 %>%
12   ggdag(stylized = TRUE) +
13   theme_dag_blank()
```



可視化の話ではありませんが、`adjustmentSets()` 関数を用いると、処置変数 T の総効果 (total effect) を推定するためにはどの変数を統制 (調整) する必要があるかを調べることも可能です。

```
1 adjustmentSets(DAG_data3, effect = "total")
```

```
## { X3 }
```

X3 変数のみ統制すれば良いという結果が得られました。また、T から Y への直接効果 (direct effect) の場合、`effect = "direct"`を指定します。

```
1 adjustmentSets(DAG_data3, effect = "direct")
```

```
## { X1, X2 }
```

X1 と X2 を統制する必要があることが分かりますね。

## 20.12 バンプチャート

バンプチャート (bump chart) は順位の変化などを示す時に有効なグラフです。たとえば、G7 構成国の新型コロナ感染者数の順位の変化を示すにはどうすれば良いでしょうか。そもそもどのような形式のデータが必要でしょうか。まずは必要なデータ形式を紹介したいと思います。

まず、{ggplot2}によるバンプチャートの作成を支援する{ggbump}パッケージをインストールし、読み込みましょう<sup>2)</sup>。

```
1 pacman::p_load(ggbump)
```

ここでは G7 構成国の 100 万人当り新型コロナ感染者数の順位がどのように変化したのかを 2020 年 4 月から 7 月まで 1 ヶ月単位で表したデータが必要です。データは以下のようなコードで作成しますが、本書のサポートページからもダウンロード可能です。

```
1 Bump_df <- left_join(COVID19_df, Country_df, by = "Country") %>%
2   select(Country, Date, Population, Confirmed_Total, G7) %>%
3   separate(Date, into = c("Year", "Month", "Day"), sep = "/") %>%
4   mutate(Month = as.numeric(Month)) %>%
5   filter(Month >= 4, G7 == 1) %>%
6   group_by(Country, Month) %>%
7   summarise(Population           = mean(Population),
8             New_Cases          = sum(Confirmed_Total, na.rm = TRUE),
9             New_Cases_per_million = New_Cases / Population * 1000000,
10            .groups           = "drop") %>%
11   select(Country, Month, New_Cases_per_million)
```

<sup>2)</sup> これから登場する geom\_bump() 幾何オブジェクトが使えるようになります。{ggbump}を使用せず、{ggplot2}が提供する geom\_line() でも代替可能です。geom\_line() の場合、直線で順位の変化が表示され、geom\_bump() の場合は曲線で表示されます。丸い感じの図を好みの方なら、{ggbump}を読み込まず、geom\_bump() を geom\_line() に書き換えてください。

```
1 Bump_df <- Bump_df %>%
2   group_by(Month) %>%
3   mutate(Rank = rank(New_Cases_per_million, ties.method = "random")) %>%
4   ungroup() %>%
5   select(Country, Month, Rank, New_Cases_per_million)
```

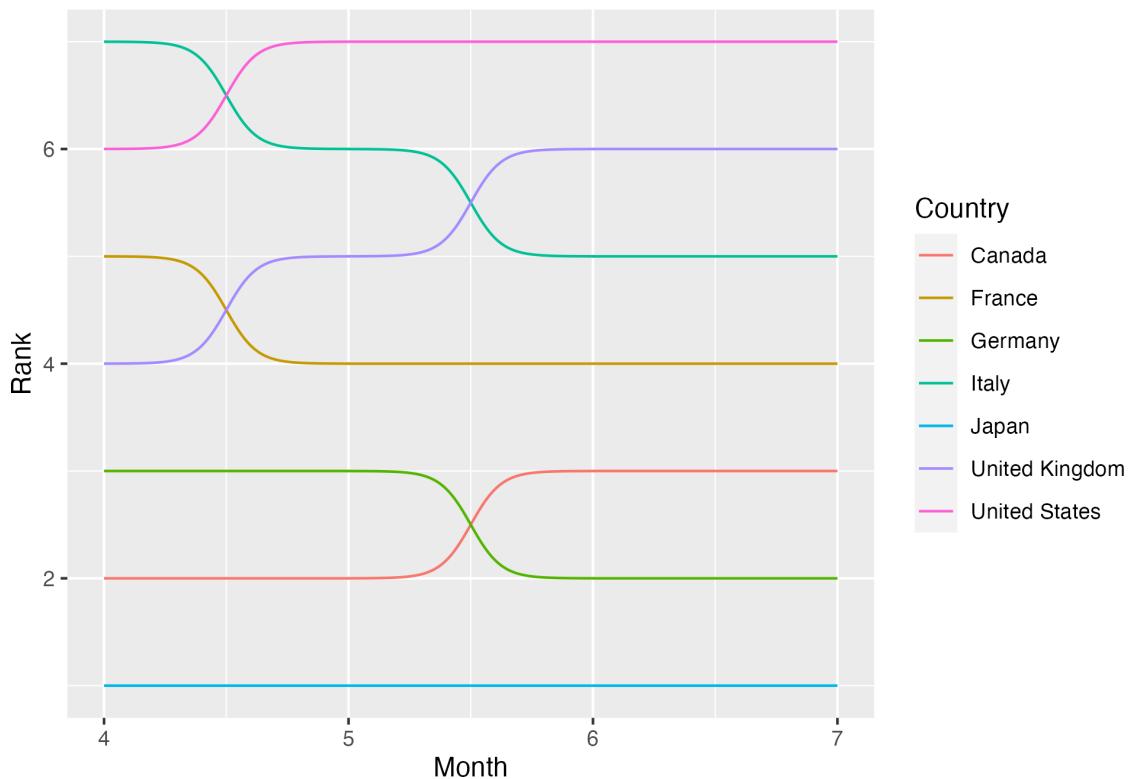
必要なデータは以下ののような形式です。ちなみにバンプチャートを作成するためには最後の New\_Cases\_per\_million 列 (100 万人当たり新型コロナ感染者数) は不要です。つまり、国名、月、順位のみで十分です。

```
# 以上のコードを省略し、加工済みのデータを読み込んでも OK
# Bump_df <- read_csv("Data/Bumpchart.csv")
Bump_df
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is i
```

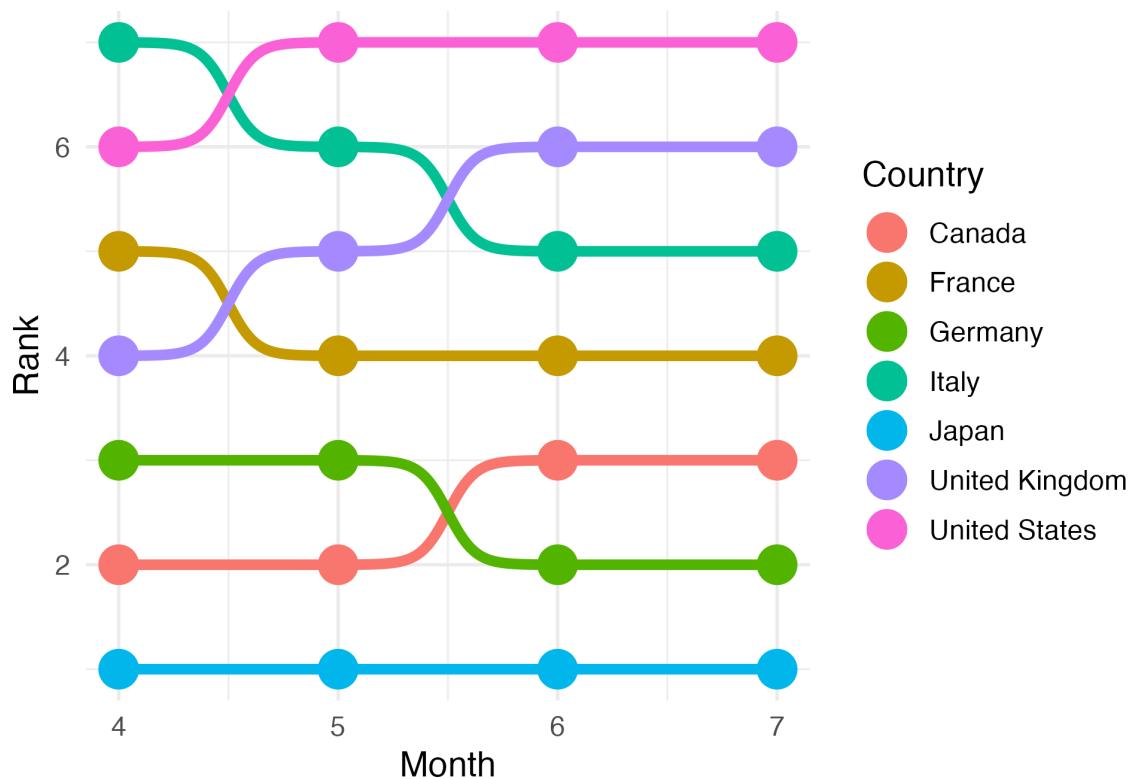
それでは{ggbump}が提供する geom\_bump() 幾何オブジェクトを使用し、簡単なバンプチャートを作成してみましょう。必要なマッピング要素は x と y、color です。x には時間を表す変数である Month を、y には順位を表す Rank をマッピングします。また、7 本の線が出るため、月と順位、それぞれの値がどの国の値かを特定する必要があります。groups に国名である Country をマッピングしても線は引けますが、どの線がどの国かが分からなくなるため、color に Country をマッピングし、線の色分けをします。

```
1 Bump_df %>%
2   ggplot(aes(x = Month, y = Rank, color = Country)) +
3   geom_bump()
```



これで最低限のバンプチャートはできましたが、もう少し見やすく、可愛くしてみましょう。今は線が細いのでややぶ厚めにします。これは `geom_bump()` レイヤーの `size` 引数で指定可能です。また、各月に点を付けることによって、同時期における順位の比較をよりしやすくしてみましょう。これは散布図と同じであるため、`geom_point()` 幾何オブジェクトを使用します。

```
1 Bump_df %>%
2   ggplot(aes(x = Month, y = Rank, color = Country)) +
3   geom_point(size = 7) +
4   geom_bump(size = 2) +
5   theme_minimal(base_size = 14)
```

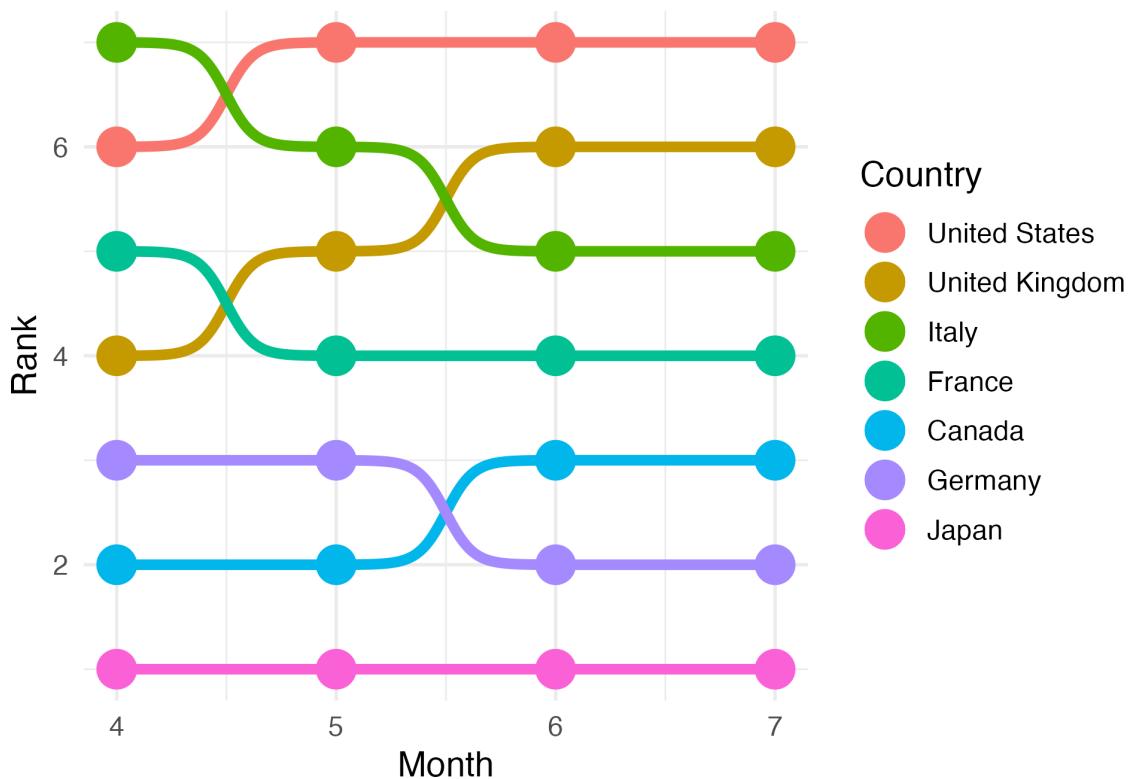


これでだいぶ見やすくなりましたが、凡例における国名の順番がやや気になりますね。7月の時点において順位が最も高い国はアメリカ、最も低い国は日本ですが、凡例の順番はアルファベット順となっています。この凡例の順番を7月時点におけるRankの値に合わせた方がより見やすいでしょう。ここで第14.2.9で紹介しましたfct\_reorder2()を使ってCountry変数の水準(level)を7月時点におけるRankの順位に合わせます。この場合、Country変数(.f = Country)の水準をMonthが(.x = Month)最も大きい(.fun = last2)時点におけるRankの順番に合わせる(.y = Rank)こととなります。fct\_reorder2()内の引数の順番の既定値は.f、.x、.y、.funとなります。

```

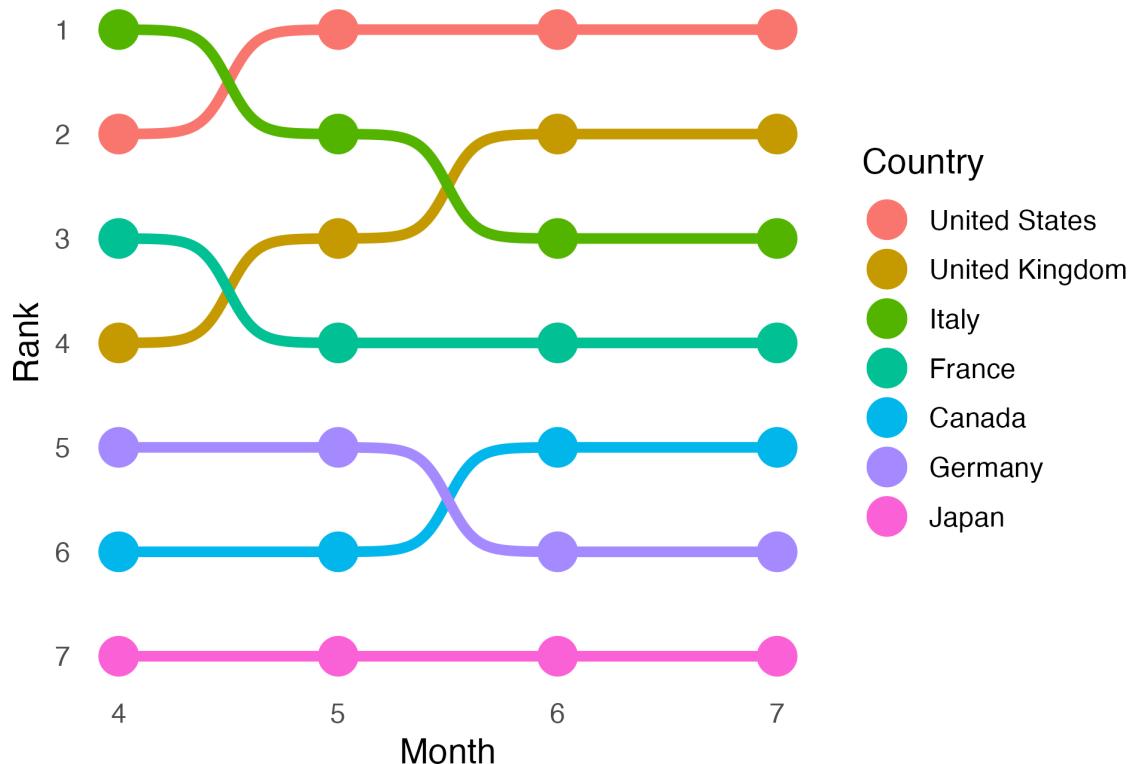
1 Bump_df %>%
2   mutate(Country = fct_reorder2(Country, Month, Rank, last2)) %>%
3   ggplot(aes(x = Month, y = Rank, color = Country)) +
4   geom_point(size = 7) +
5   geom_bump(size = 2) +
6   theme_minimal(base_size = 14)

```



最後に縦軸の目盛りラベルを付けます。上に行くほど順位が高くなりますので、1を7に、2を6に、...、7を1に変更します。また、図内のグリッドも不要ですので、theme()を使用し、グリッドを削除します (panel.grid = element\_blank())。

```
1 Bump_df %>%
2   mutate(Country = fct_reorder2(Country, Month, Rank, last2)) %>%
3   ggplot(aes(x = Month, y = Rank, color = Country)) +
4   geom_point(size = 7) +
5   geom_bump(size = 2) +
6   scale_y_continuous(breaks = 1:7, labels = 7:1) +
7   theme_minimal(base_size = 14) +
8   theme(panel.grid = element_blank())
```



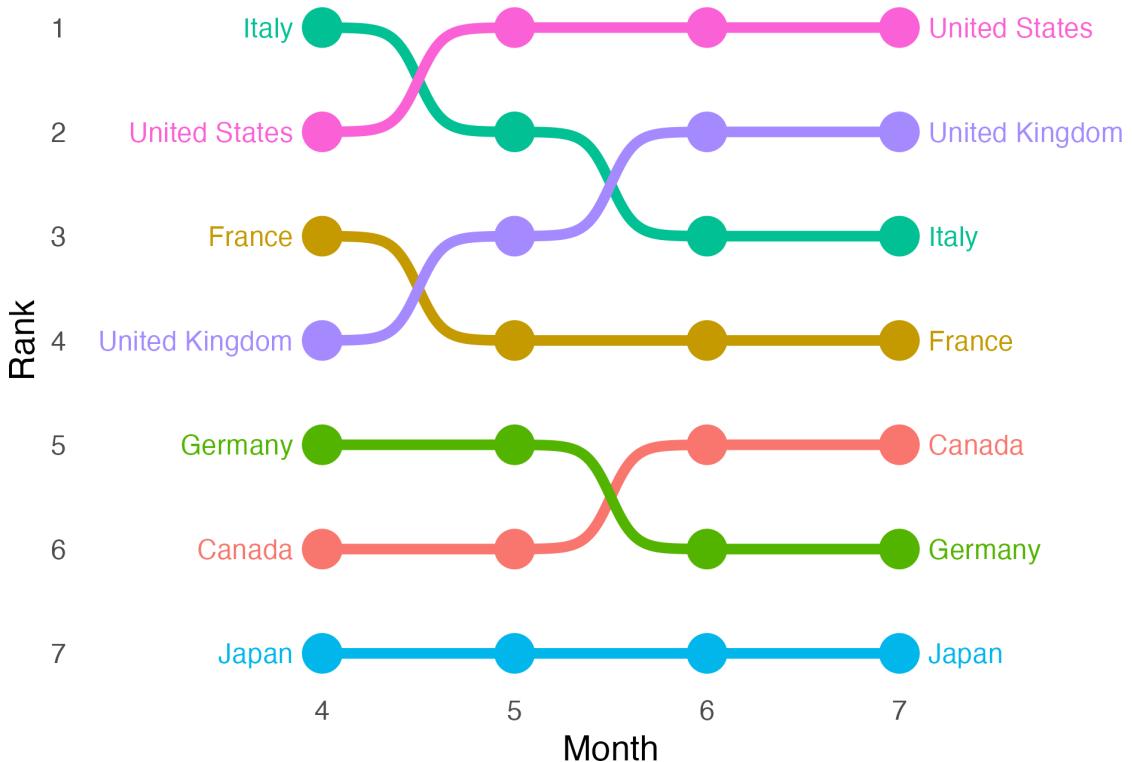
これでバンプチャートの完成です。このまでの良いかも知れませんが、もう少し手間を掛けることでより読みやすいグラフが作れます。たとえば、今のままだと「日本のトレンド」を確認したい場合、まず凡例から日本の色を確認し、その色に該当する点と線を見つけてトレンドを見る必要がありますね。もし、ここで図の左端と右端の点の横に国名を出力すると、凡例がなくても良いですし、4月の時点から日本のトレンドを確認することも、7月の時点から遡る形で日本のトレンドを確認することも可能かも知れません。

図に文字列を追加するためには `geom_text()` 異なるオブジェクトを使用します。マッピング要素は文字列の横軸上の位置 (x)、縦軸上の位置 (y)、そして出力する文字列 (label) です。左端に文字列を出力するのであれば、横軸上の位置は 4 (= 4 月) よりも若干左側が良いでしょう。ぴったり 4 になると、点と文字列が重なって読みにくくなりますね。縦軸上の位置は **4月の時点**での順位 (Rank) で、出力する文字列は国名 (Country) です。現在、使用しているデータは4月から7月までのデータですが、4月に限定したデータを使いたいので、`geom_text()` 内に `data` 引数を追加し、`Bump_df` から `Month` の値が 4 の行のみを抽出したデータを割り当てます (`data = filter(Bump_df, Month == 4)`)、または `data = Bump_df %>% filter(Month == 4)`)。右端についても同じです。横軸

上の位置は 7 から右方向へずらし、使用するデータは 7 月のデータとなります。

最後にもう一点調整が必要ですが、それは座標系です。図の座標系は `ggplot()` 関数で使用するデータに基づきます。`Bump_df` の場合、横軸 (`Month`) は 4 から 7 です。しかし、文字列を追加した場合、文字列がすべて出力されないかも知れません。したがって、座標系を横方向に広める必要があります。今回は 3 から 8 までに調整します。座標系や文字列の位置調整は出力結果を見ながら、少しづつ調整していきましょう。

```
1 Bump_df %>%
2   ggplot(aes(x = Month, y = Rank, color = Country)) +
3   geom_point(size = 7) +
4   geom_bump(size = 2) +
5   # 4月の時点での行のみ抽出し、x は Month より 0.15 分左方向、
6   # y は Rank の値の位置に国名を出力する。揃える方向は右揃え (hjust = 1)
7   geom_text(data = filter(Bump_df, Month == 4),
8             aes(x = Month - 0.15, y = Rank, label = Country), hjust = 1) +
9   # 7月の時点での行のみ抽出し、x は Month より 0.15 分右方向、
10  # y は Rank の値の位置に国名を出力する。揃える方向は左揃え (hjust = 0)
11  geom_text(data = filter(Bump_df, Month == 7),
12             aes(x = Month + 0.15, y = Rank, label = Country), hjust = 0) +
13  # 座標系の調整
14  coord_cartesian(xlim = c(3, 8)) +
15  scale_x_continuous(breaks = 4:7, labels = 4:7) +
16  scale_y_continuous(breaks = 1:7, labels = 7:1) +
17  labs(y = "Rank", x = "Month") +
18  theme_minimal(base_size = 14) +
19  theme(legend.position = "none",
20        panel.grid      = element_blank())
```



## 20.13 沖積図

沖積図 (alluvial plot) は同じ対象を複数回観察したデータ (パネル・データなど) から変化を示すことに適したグラフです。たとえば、同じ回答者を対象に 2 回世論調査を実施し、1 回目調査時の支持政党と 2 回目調査時の支持政党を変化を見ることも可能です。もし、変化が大きい場合は政党支持態度は弱いこととなりますし、変化が小さい場合は安定していると解釈できるでしょう。

ここでは 2009 年の衆院選と 2010 年の参院選における投票先の変化を沖積図で確認してみたいと思います。まずは、沖積図の作成に特化した{ggalluvial}パッケージをインストールし、読み込みます。

```
1 pacman::p_load(ggalluvial)
```

続きまして、実習用データを読み込みます。データは 中澤 [2014] の表 3 を元に筆者

(宋) が作成したものです。

```
1 Vote_0910 <- read_csv("Data/Vote_09_10.csv")
2
3 head(Vote_0910, 20)

## # A tibble: 20 x 3
##       ID Vote09     Vote10
##   <dbl> <chr>     <chr>
## 1     1 畏權     畏權
## 2     2 民主     民主
## 3     3 民主     民主
## 4     4 自民     民主
## 5     5 自民   共產・社民
## 6     6 民主   共產・社民
## 7     7 その他  その他
## 8     8 自民     自民
## 9     9 民主     民主
## 10    10 共產・社民 共產・社民
## 11    11 畏權     自民
## 12    12 自民     民主
## 13    13 畏權     畏權
## 14    14 自民     自民
## 15    15 共產・社民 共產・社民
## 16    16 DK       共產・社民
## 17    17 自民     DK
## 18    18 民主     民主
## 19    19 民主     民主
## 20    20 民主     民主
```

---

変数名	説明
ID	回答者 ID
Vote09	2009 年衆院選における投票先

変数名	説明
Vote10	2010 年参院選における投票先

たとえば、1 番目の回答者は 2009 年に棄権し、2010 年も棄権したことを意味する。また、5 番目の回答者は 2009 年に自民党に投票し、2010 年は共産党または社民党に投票したことを意味する。続いて、このデータを{ggalluvial}に適した形式のデータに加工します。具体的には「2009 年棄権、かつ 2010 年棄権」、「2009 年棄権、かつ 2010 年自民党投票」、「...」、「2009 年民主党投票、かつ 2010 年自民党投票」のように全ての組み合わせに対し、該当するケース数を計算する必要があります。今回のデータだと、投票先はいずれも自民、民主、公明、共産・社民、その他、棄権、DK (わからない) の 7 であるため、49 パターンができます。それぞれのパターンに該当するケース数を計算するためには Vote09 と Vote10 でデータをグループ化し、ケース数を計算します。

```

1 Vote_0910 <- Vote_0910 %>%
2   group_by(Vote09, Vote10) %>%
3   summarise(Freq      = n(),
4             .groups = "drop")
5
6 Vote_0910

```

```

## # A tibble: 47 x 3
##   Vote09  Vote10    Freq
##   <chr>   <chr>    <int>
## 1 DK      DK        111
## 2 DK      その他     8
## 3 DK      公明       7
## 4 DK      共産・社民  15
## 5 DK      棄権       57
## 6 DK      民主       116
## 7 DK      自民       29
## 8 その他 DK        18
## 9 その他 その他     73

```

```
## 10 その他 公明          4
## # ... with 37 more rows
```

2009 年の調査で「わからない」と回答し、2010 年の調査でも「わからない」と回答した回答者数は 111 名、2009 年の調査で「わからない」と回答し、2010 年の調査では「棄権」と回答した回答者数は 8 名、...といったことが分かりますね。

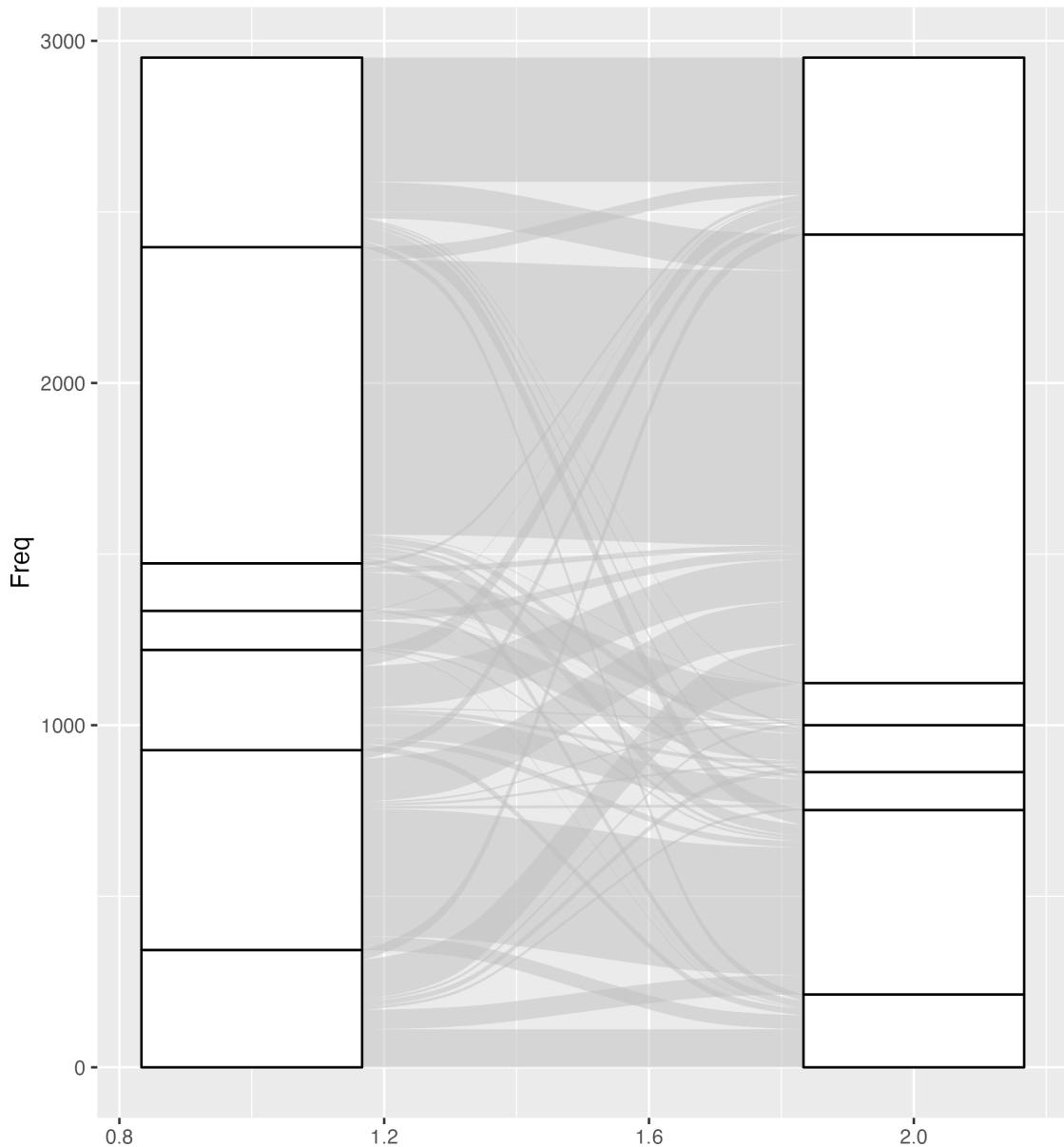
続いて、グラフを作成する前に投票先変数 (Vote09 と Vote10) を factor 化します。可視化の際、投票先が出力される順番に決まりはありませんが、自民、民主、公明、...、DK の順が自然かと思います。もちろん、こちらは自分から見て分かりやいように順番を決めましょう。ただし、2 変数における水準 (level) の順番は一致させた方が良いでしょう。

```
1 Vote_0910 <- Vote_0910 %>%
2   mutate(Vote09 = factor(Vote09, levels = c("自民", "民主", "公明", "共産・社民",
3                                "その他", "棄権", "DK")),
4         Vote10 = factor(Vote10, levels = c("自民", "民主", "公明", "共産・社民",
5                                "その他", "棄権", "DK")))
```

それでは沖積図を描いてみましょう。使用する幾何オブジェクトは geom\_alluvium() と geom\_stratum() です。必ずこの順番でレイヤーを重ねてください。マッピングは y、axis1、axis2 に対し、y には当該パターン内のケース数 (Freq)、axis1 は 2009 年の投票先 (Vote09)、axis2 は 2010 年の投票先 (Vote10) を指定します<sup>3)</sup>。これらのマッピングは geom\_stratum() と geom\_alluvium() 共通であるため、ggplot() 内でマッピングした方が効率的です。

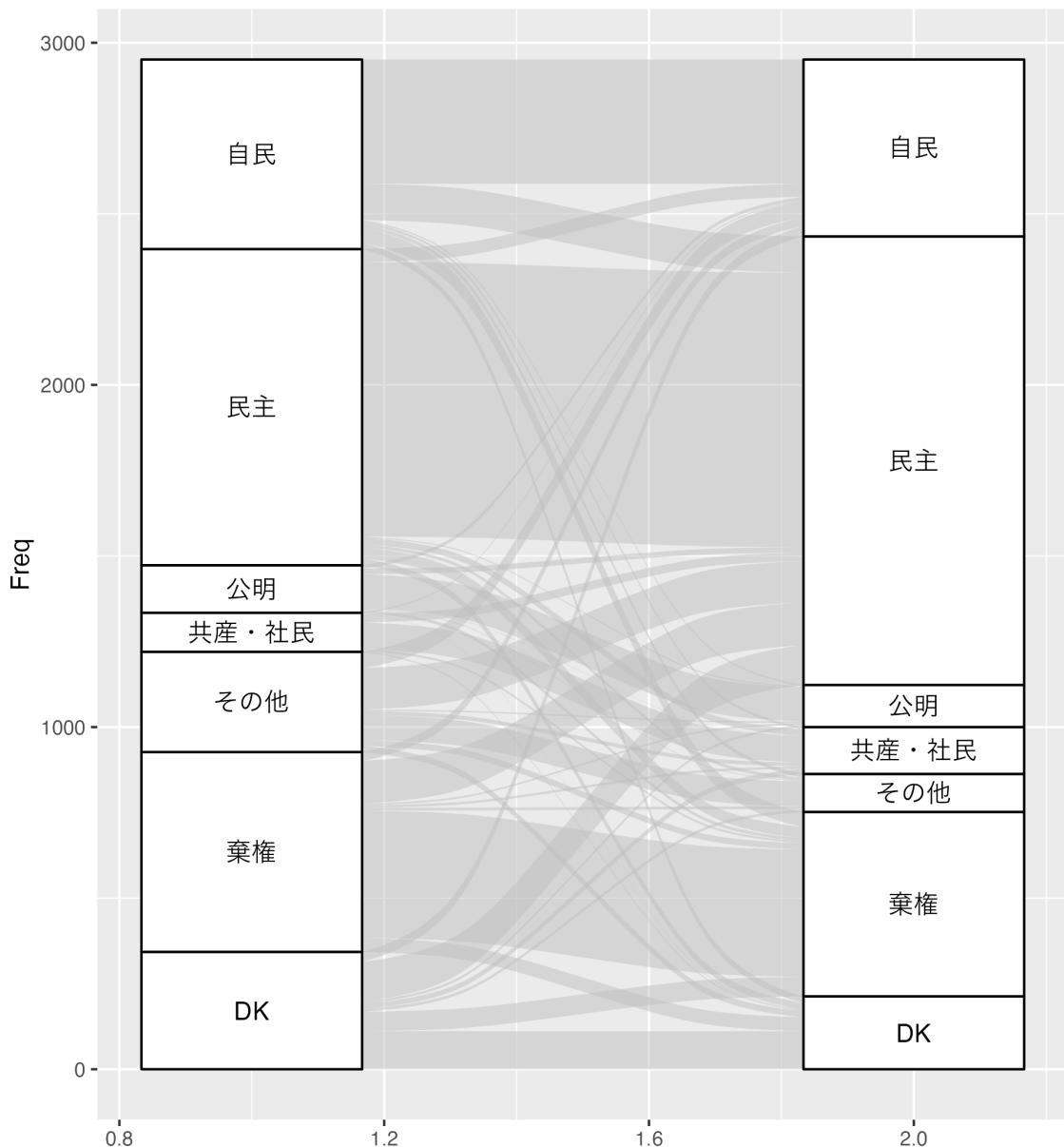
```
1 Vote_0910 %>%
2   ggplot(aes(y = Freq, axis1 = Vote09, axis2 = Vote10)) +
3     geom_alluvium() +
4     geom_stratum()
```

<sup>3)</sup> 今回は 2 期間の変化を示していますが、3 期以上の場合は、axis3、axis4、... と追加してください。



何かの図は出てきましたが、これだけだと、それぞれの四角形がどの政党を示しているのかが分かりませんね。四角形内に政党名を出力するためには{ggplot2}内蔵の `geom_text()` を使用します。マッピング要素は `ggplot()` 内でマッピングしたものに加え、`label` が必要ですが、ここでは `after_stat(stratum)` を指定します。そして、`aes()` のその側に `stat = "stratum"` を指定するだけです。フォントの指定が必要な場合は `family` を使います。`theme_*`() 内で `base_family` を指定した場合でも必要です。

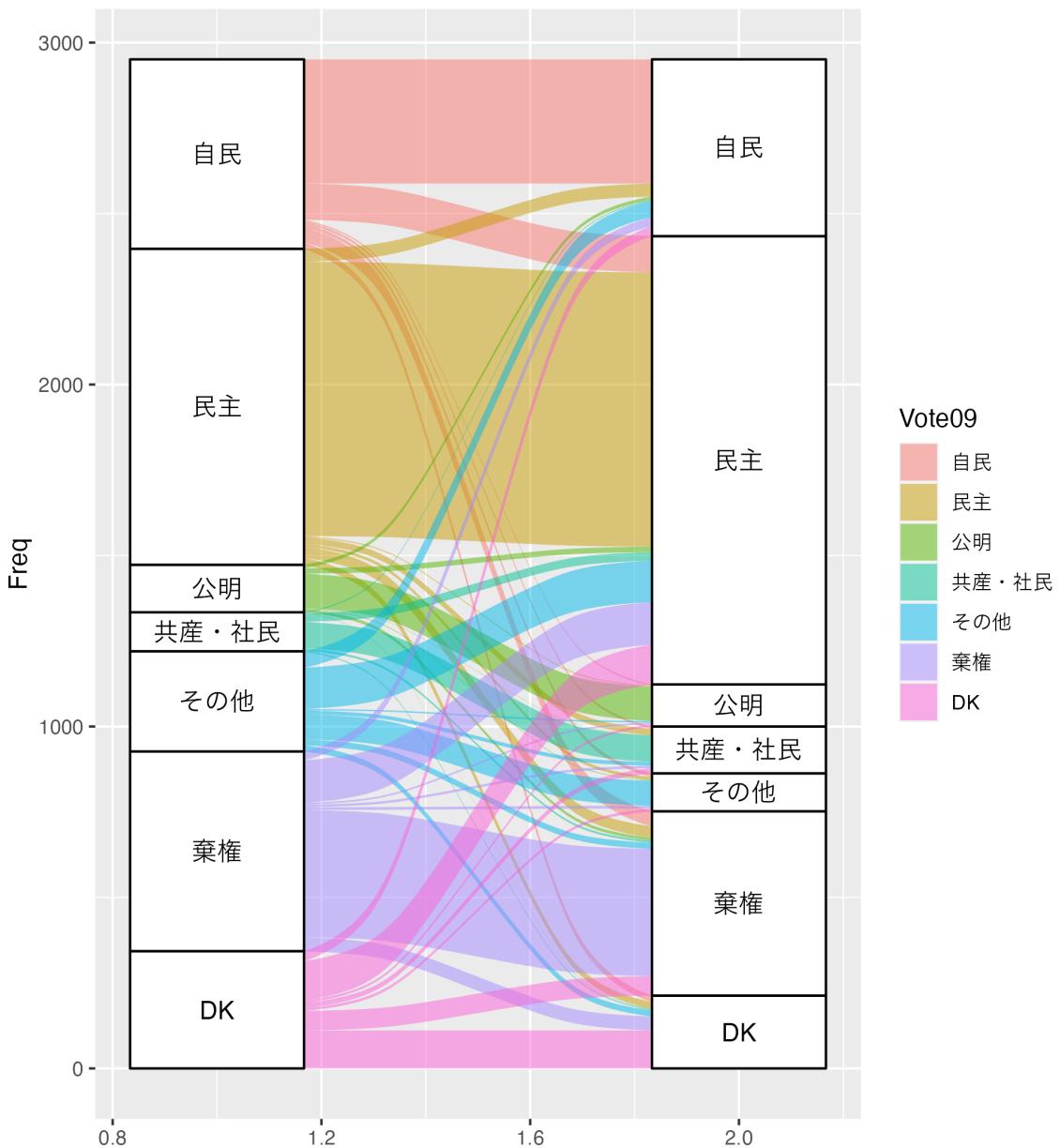
```
1 Vote_0910 %>%
2   ggplot(aes(y = Freq, axis1 = Vote09, axis2 = Vote10)) +
3   geom_alluvium() +
4   geom_stratum() +
5   geom_text(aes(label = after_stat(stratum)),
6             stat = "stratum")
```



これで沖積図はとりあえず完成ですが、少し読みやすく加工してみましょう。たとえば、2010年に民主党に投票した回答者において2009年の投票先の割合を見たいとした場合、`geom_alluvium()` 内に `fill = Vote09` をマッピングします。これで帯に2009年の投票先ごとの色付けができます。

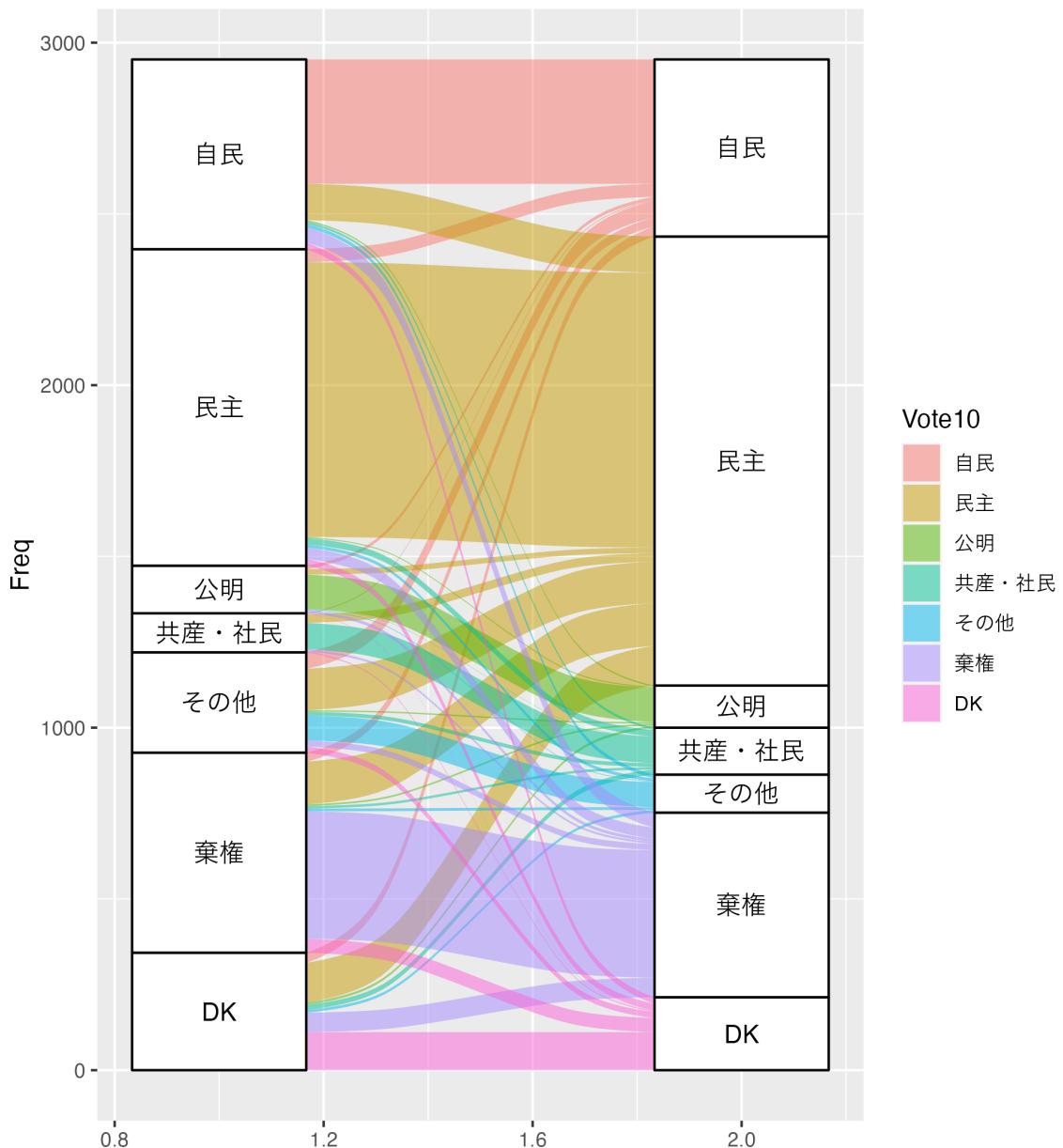
```
1 Vote_0910 %>%
2   ggplot(aes(y = Freq, axis1 = Vote09, axis2 = Vote10)) +
```

```
3     geom_alluvium(aes(fill = Vote09)) +
4     geom_stratum() +
5     geom_text(aes(label = after_stat(stratum)),
6               stat = "stratum", family = "HiraginoSans-W3")
```



`fill = Vote10` とマッピングした場合は、感覚が変わります。実際にやってみましょう。

```
1 Alluvial_Plot <- Vote_0910 %>%
2   ggplot(aes(y = Freq, axis1 = Vote09, axis2 = Vote10)) +
3   geom_alluvium(aes(fill = Vote10)) +
4   geom_stratum() +
5   geom_text(aes(label = after_stat(stratum)),
6             stat = "stratum", family = "HiraginoSans-W3")
7
8 Alluvial_Plot
```

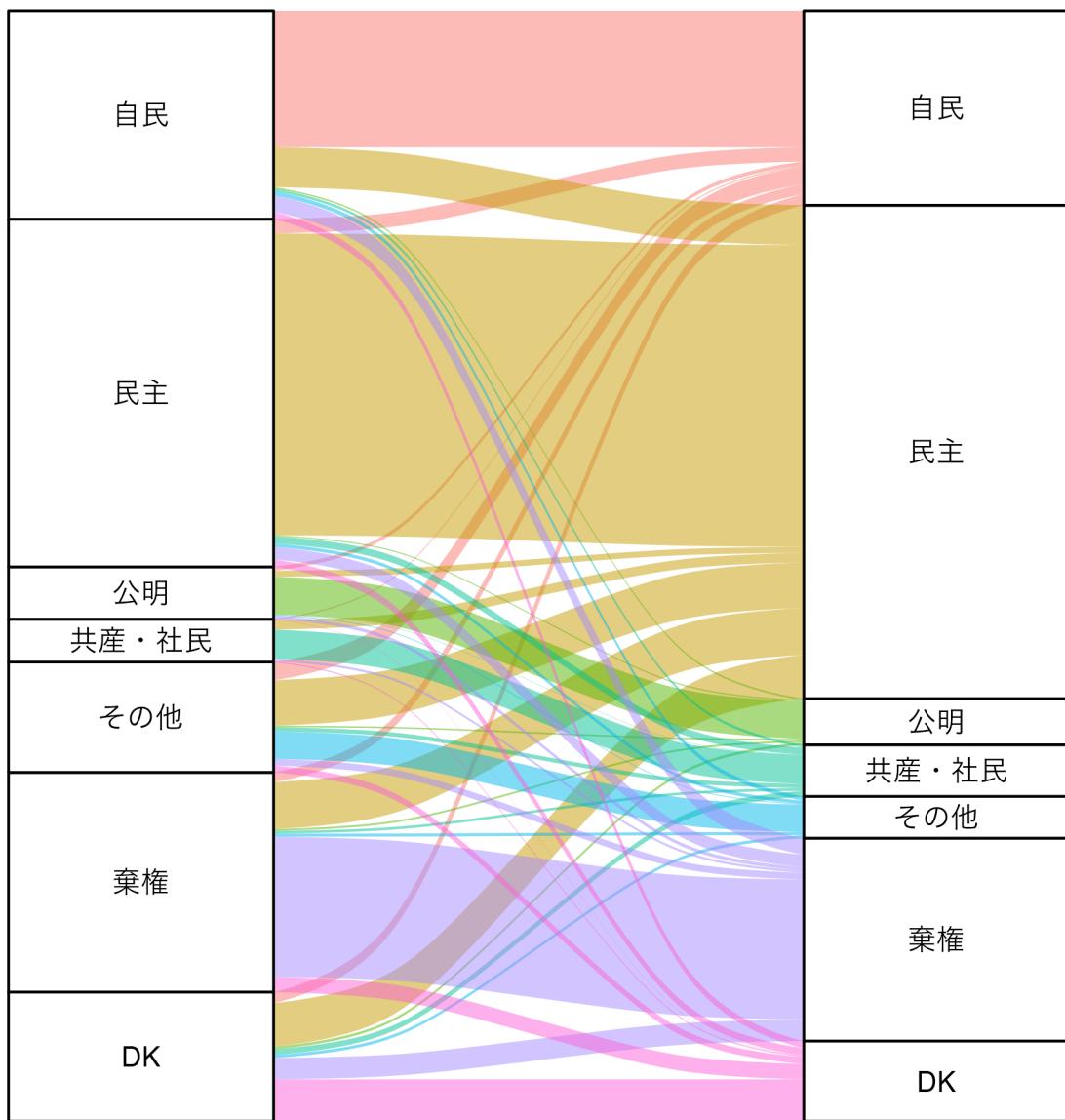


この場合、2009年に民主党に投票した人が2010年にどこに流れたかが分かりやすくなります。Vote09に色分けするか、Vote10に色分けするかは作成する人が決める問題であり、自分の主張・メッセージに適したマッピングをしましょう。

最後に、図をもう少し加工してみましょう。まず、横軸に0.8、1.2、1.6、2.0となっている目盛りを修正し、1と2の箇所に「2009年衆院選」と「2010年参院選」を出力します。これは `scale_x_continuous()` で調整可能です。そして、`theme_minimal()`

で余計なものを排除したテーマを適用します。最後に `theme()` 内で全ての凡例を削除 (`legend.position = "none"`) し、パネルのグリッド (`panel.grid`) と縦軸・横軸のタイトル (`axis.title`)、縦軸の目盛りラベル (`axis.text.y`) を削除 (`element_blank()`) します。

```
1 Alluvial_Plot +
2   scale_x_continuous(breaks = 1:2,
3     labels = c("2009 年衆院選", "2010 年参院選")) +
4   theme_minimal(base_size = 16) +
5   theme(legend.position = "none",
6     panel.grid = element_blank(),
7     axis.title = element_blank(),
8     axis.text.y = element_blank())
```



## 20.14 ツリーマップ

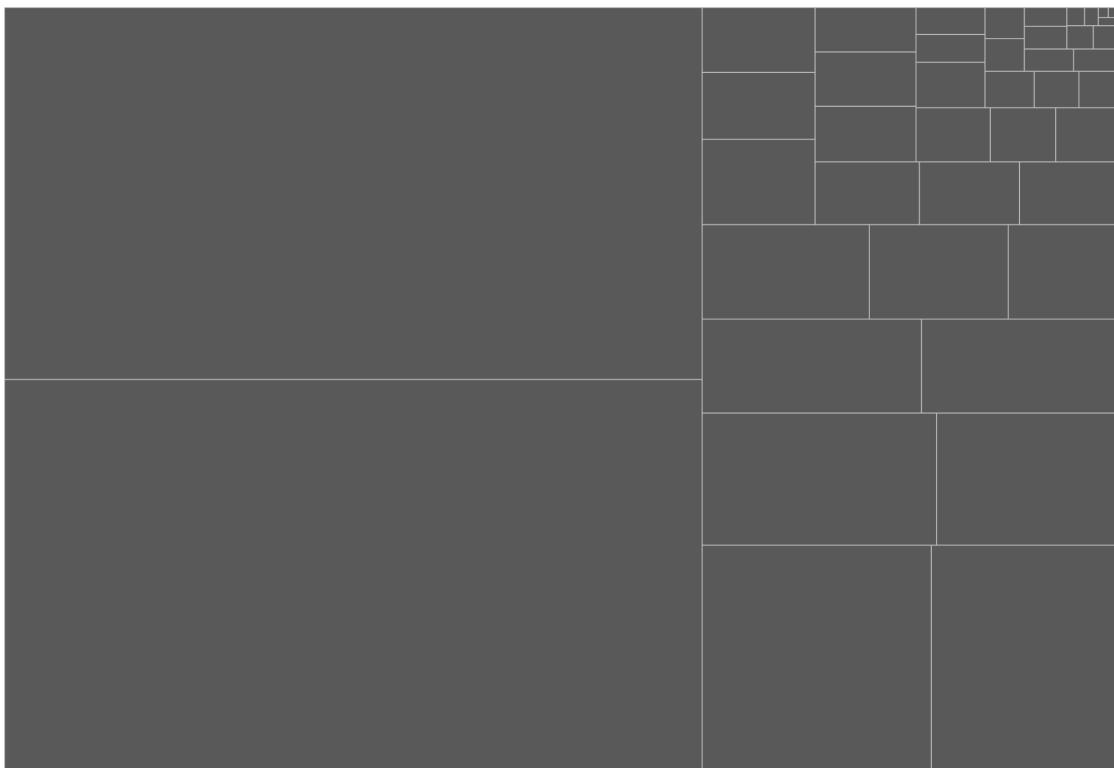
ツリーマップ (tree map) は変数の値の大きさを長方形の面積として表すグラフです。全体におけるシェアを示す時に使う点で、円グラフの代替案の一つになります。円グラフは項目が多すぎると読みにくいデメリットがありますが、ツリーマップは項目が多い場合でも有効です。むろん、多すぎると読みにくいことは同じですので、注意が必要です。

ツリーマップを作成するためには{treemapify}パッケージの `geom_treemap()` 異何オブジェクトを使用します。まず、{treemapify}をインストールし、読み込みます。

```
1 pacman::p_load(treemapify)
```

ここでは `Country_df` からアジア諸国の人団 (Population) をツリーマップをして可視化したいと思います。`geom_treemap()` の場合、各長方形は面積の情報のみを持ちます。この面積の情報を `area` にマッピングします。

```
1 Country_df %>%
  2   filter(Continent == "Asia") %>%
  3   ggplot() +
  4   geom_treemap(aes(area = Population))
```



これだけだと各長方形がどの国を指しているのかが分かりませんね。長方形の上に国名（Country）を追加するためには `geom_treemap_text()` 幾何オブジェクトを使用します。マッピングは `area` と `label` に対し、それぞれ面積を表す `Population` と国名を表す `Country` を指定します。`area` は `geom_treemap()` と `geom_treemap_text()` 両方で使われる所以 `ggplot()` の内部でマッピングしても問題ありません<sup>4)</sup>。また、`aes()` の外側に `color = "white"` で文字を白に指定し、`place = "center"` で長方形の真ん中にラベルが付くようにします。

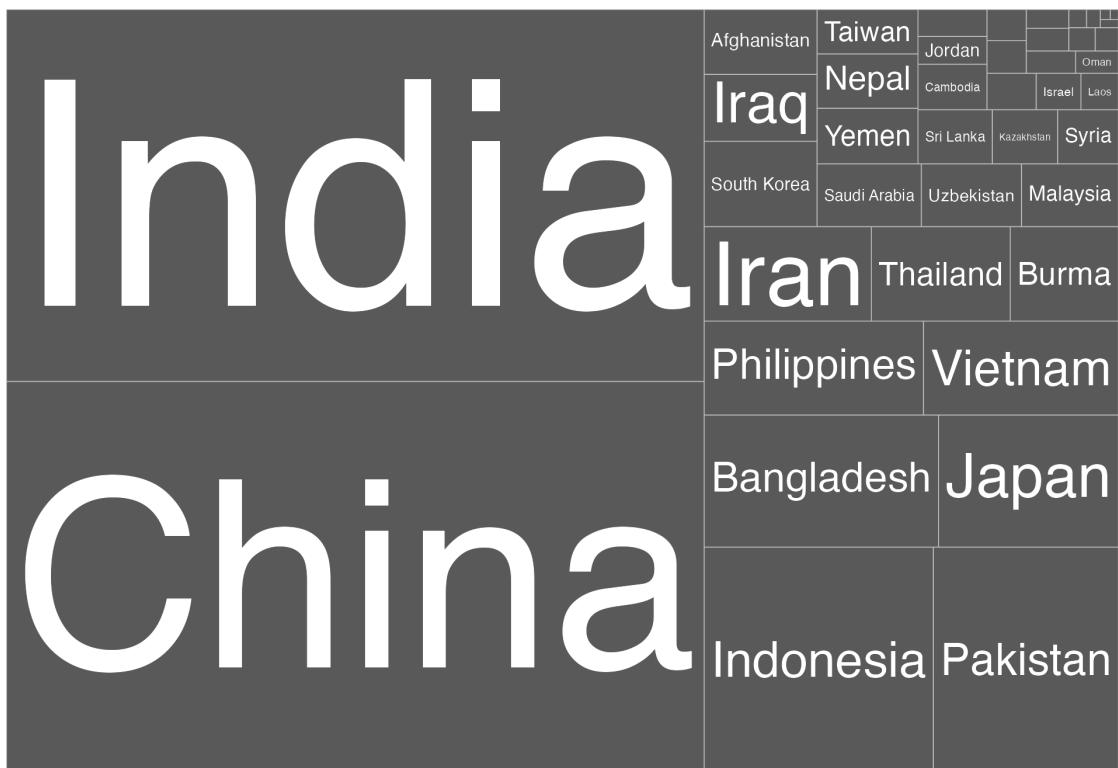
```
1 Country_df %>%
2   filter(Continent == "Asia") %>%
3   ggplot(aes(area = Population)) +
4   geom_treemap() +
5   geom_treemap_text(aes(label = Country), color = "white", place = "center")
```

<sup>4)</sup> 今回の例だと、`label` も `ggplot()` 内部でマッピング可能です。



これでツリーマップが完成しました。インドと中国の存在感がかなり大きいですね。更にラベルのサイズを長方形に合わせると、その存在感をより高めることができます。ラベルの大きさを長方形に合わせるには `geom_treemap_text()` の内部に `grow = TRUE` を指定します。

```
1 Country_df %>%
2   filter(Continent == "Asia") %>%
3   ggplot(aes(area = Population, label = Country)) +
4   geom_treemap() +
5   geom_treemap_text(color = "white", place = "center",
6                     grow = TRUE)
```



ここで更に次元を追加するために、色塗りをしてみましょう。たとえば、G20 加盟国か否かで色分けをしたい場合、`fill` に G20 をマッピングします。ただし、今のままだと G20 は連続変数扱いになりますので、character 型、または factor 型に変換します。

```

1 Country_df %>%
2   mutate(G20 = if_else(G20 == 1, "Member", "Non-member")) %>%
3   filter(Continent == "Asia") %>%
4   ggplot(aes(area = Population, fill = G20,
5             label = Country)) +
6   geom_treemap() +
7   geom_treemap_text(color = "white", place = "centre",
8                     grow = TRUE) +
9   labs(fill = "G20") +
10  ggttitle("Population in Asia") +
11  theme(legend.position = "bottom")

```

## Population in Asia



色塗りは連続変数に対して行うことも可能です。ここでは 2018 年人間開発指数 (HDI\_2108) の値に応じて色塗りをしてみます。また、HDI\_2018 が低い (low) と brown3、高い (high) と cornflowerblue 色にします。真ん中の値 (midpoint) は 0.7 とし、色 (mid) は cornsilk を使います。

連続変数でマッピングされた色塗り (fill) の調整には `scale_fill_gradient()`、または `scale_fill_gradient2()` を使います。前者は中間点なし、後者は中間点あります。これらの使い方は第 19 章で紹介しました `scale_color_gradient()` と同じです。

```

1 Country_df %>%
2   filter(Continent == "Asia") %>%
3   ggplot(aes(area = Population, fill = HDI_2018,
4             label = Country)) +
5   geom_treemap() +
6   geom_treemap_text(color = "white", place = "centre",
7                     grow = TRUE) +
8   scale_fill_gradient2(low = "brown3",

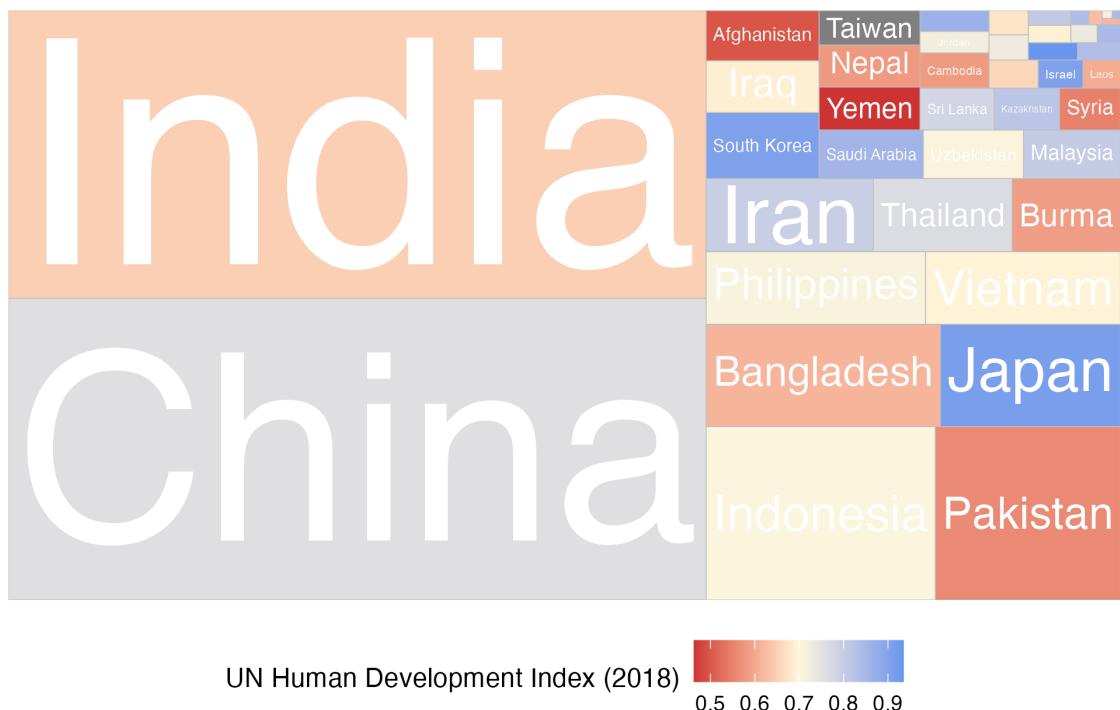
```

```

9      mid = "cornsilk",
10     high = "cornflowerblue",
11     midpoint = 0.7) +
12   labs(fill = "UN Human Development Index (2018)") +
13   ggtitle("Population in Asia") +
14   theme(legend.position = "bottom")

```

Population in Asia



ちなみに以上の図を円グラフにすると以下のようにになります（国名は人口の割合が2.5%を超える国のみ表示）。ツリーマップと比較してかなり読みにくいことが分かります。

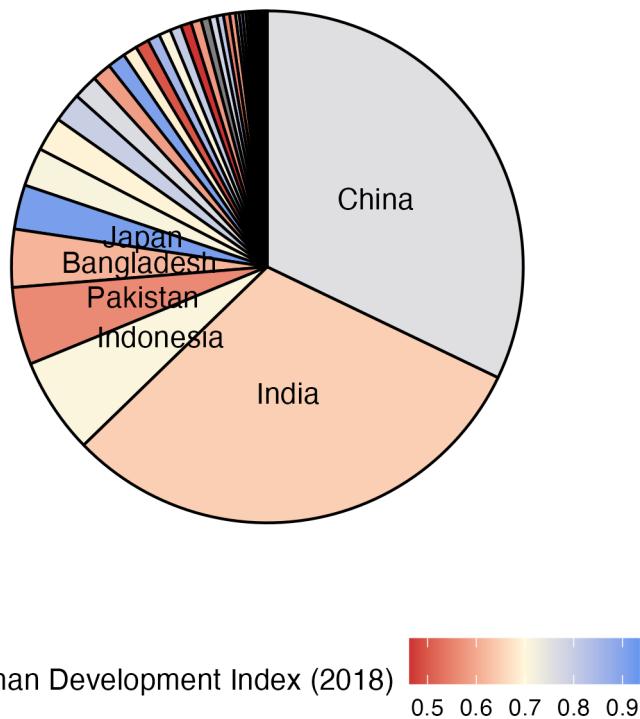
```

1 Country_df %>%
2   filter(Continent == "Asia") %>%
3   arrange(Population) %>%
4   mutate(Prop      = Population / sum(Population) * 100,
5         LabelY    = 100 - (cumsum(Prop) - 0.5 * Prop),

```

```
6     CountryName = if_else(Prop < 2.5, "", Country),
7     Country      = fct_inorder(Country)) %>%
8   ggplot() +
9   geom_bar(aes(x = 1, y = Prop, group = Country, fill = HDI_2018),
10          color = "black", stat = "identity", width = 1) +
11   geom_text(aes(x = 1, y = LabelY, label = CountryName)) +
12   coord_polar("y", start = 0) +
13   scale_fill_gradient2(low = "brown3",
14                      mid = "cornsilk",
15                      high = "cornflowerblue",
16                      midpoint = 0.7) +
17   labs(fill = "UN Human Development Index (2018)") +
18   ggtitle("Population in Asia") +
19   theme_minimal() +
20   theme(legend.position = "bottom",
21         panel.grid = element_blank(),
22         axis.title = element_blank(),
23         axis.text  = element_blank())
```

Population in Asia



ただし、ツリーマップが必ずしも円グラフより優れているとは言えません。たとえば、Heer and Bostock [2010] の研究では円グラフとツリーマップを含む9種類のグラフを用い、被験者に大小関係を判断してもらう実験を行いましたが、ツリーマップ（四角形の面積）は円グラフ（角度）よりも判断までの所要時間が長いことが述べています。

## 20.15 モザイクプロット

モザイクプロットは2つの離散変数（おもに名目変数）の関係を可視化するためにHartigan and Kleiner [1984] が考案した図です。2つの名目変数間の関係を見る際によく使われるものはクロス表（クロス集計表）でしょう。

```
1 pacman::p_load(ggmosaic)
```

```

1 Mosaic_df <- Country_df %>%
2   select(Country, Continent, Polity = Polity_Type, PPP = PPP_per_capita) %>%
3   mutate(Continent = factor(Continent,
4                             levels = c("Africa", "America", "Asia",
5                                         "Europe", "Oceania")),
6         Polity     = factor(Polity,
7                             levels = c("Autocracy", "Closed Anocracy",
8                                         "Open Anocracy", "Democracy",
9                                         "Full Democracy")),
10        PPP       = if_else(PPP >= 15000, "High PPP", "Low PPP"),
11        PPP       = factor(PPP, levels = c("Low PPP", "High PPP"))) %>%
12   drop_na()
1
1 head(Mosaic_df)

## # A tibble: 6 x 4
##   Country   Continent Polity        PPP
##   <chr>     <fct>    <fct>        <fct>
## 1 Afghanistan Asia   Closed Anocracy Low PPP
## 2 Albania     Europe  Democracy     Low PPP
## 3 Algeria     Africa  Open Anocracy Low PPP
## 4 Angola      Africa  Closed Anocracy Low PPP
## 5 Argentina   America Democracy   High PPP
## 6 Armenia     Europe  Democracy     Low PPP

```

クロス表を作成する内蔵関数としては `table()` があります。2つの変数が必要となり、第一引数が行、第二引数が列を表します。

```

1 Mosaic_Tab <- table(Mosaic_df$Continent, Mosaic_df$Polity)
2 Mosaic_Tab

##
##          Autocracy Closed Anocracy Open Anocracy Democracy Full Democracy
##  Africa           3                 14                 11                 18                 1

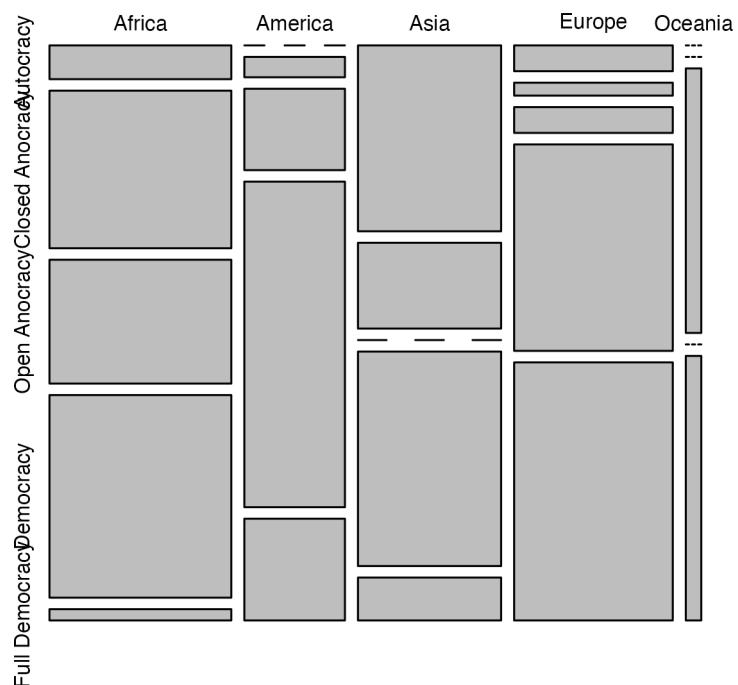
```

##	America	0	1	4	16	5
##	Asia	13	6	0	15	3
##	Europe	2	1	2	16	20
##	Oceania	0	0	2	0	2

この `Mosaic_Tab` のクラスは"table"ですが、"table"クラスのオブジェクトを `plot()` に渡すと別途のパッケージを使わずモザイクプロットを作成することができます。

```
1 plot(Mosaic_Tab)
```

**Mosaic\_Tab**



やや地味ではありますが、モザイクプロットが出来ました。ここからはより読みやすいモザイクプロットを作成するために`{ggmosaic}`パッケージの `geom_mosaic()` 関数を使います。

`geom_mosaic()` の場合、`x` のみのマッピングで十分です。ただし、特定の変数を指定するのではなく、`product(変数 1, 変数 2)` を `x` にマッピングする必要があります。`table()` 関数同様、変数 1 は行、変数 2 は列です。また、欠損値が含まれている行がある

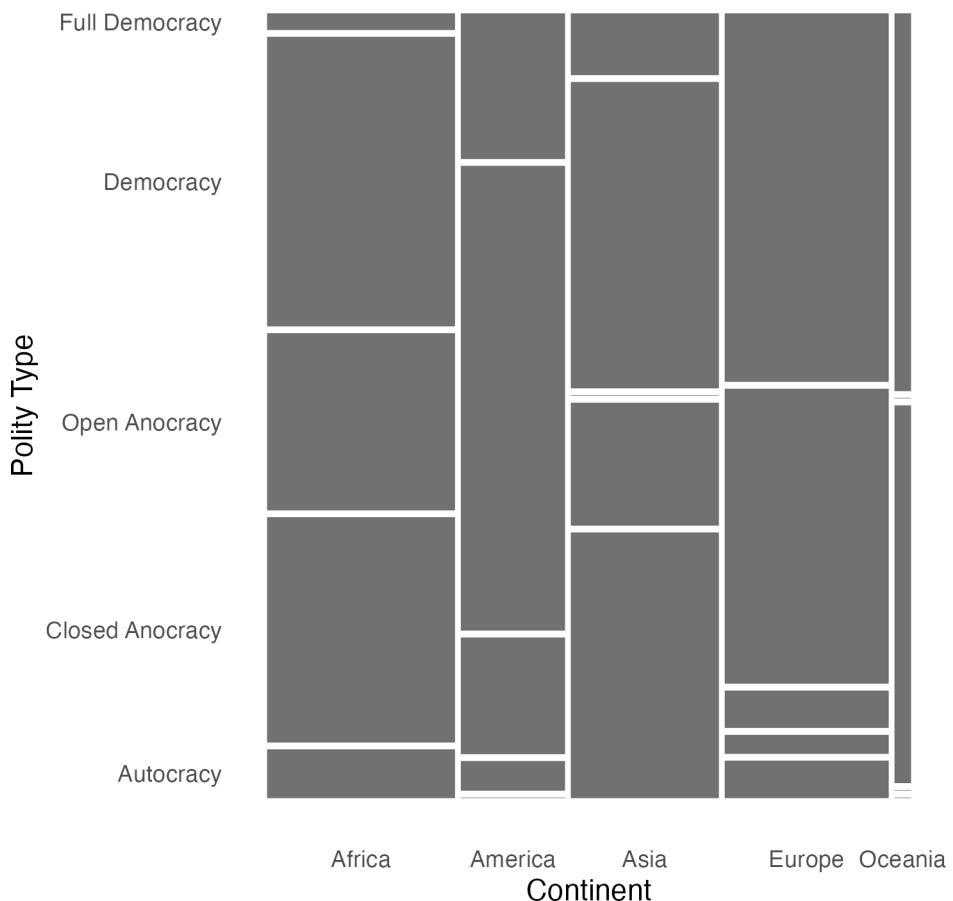
場合は、`aes()` の外側に `na.rm = TRUE` を指定する必要があります。今回は `drop_na()` で欠損値をすべて除外しましたが、念の為に指定しておきます。

```

1 Mosaic_df %>%
2   ggplot() +
3   geom_mosaic(aes(x = product(Polity, Continent)), na.rm = TRUE) +
4   labs(x = "Continent", y = "Polity Type") +
5   theme_minimal() +
6   theme(panel.grid = element_blank())

```

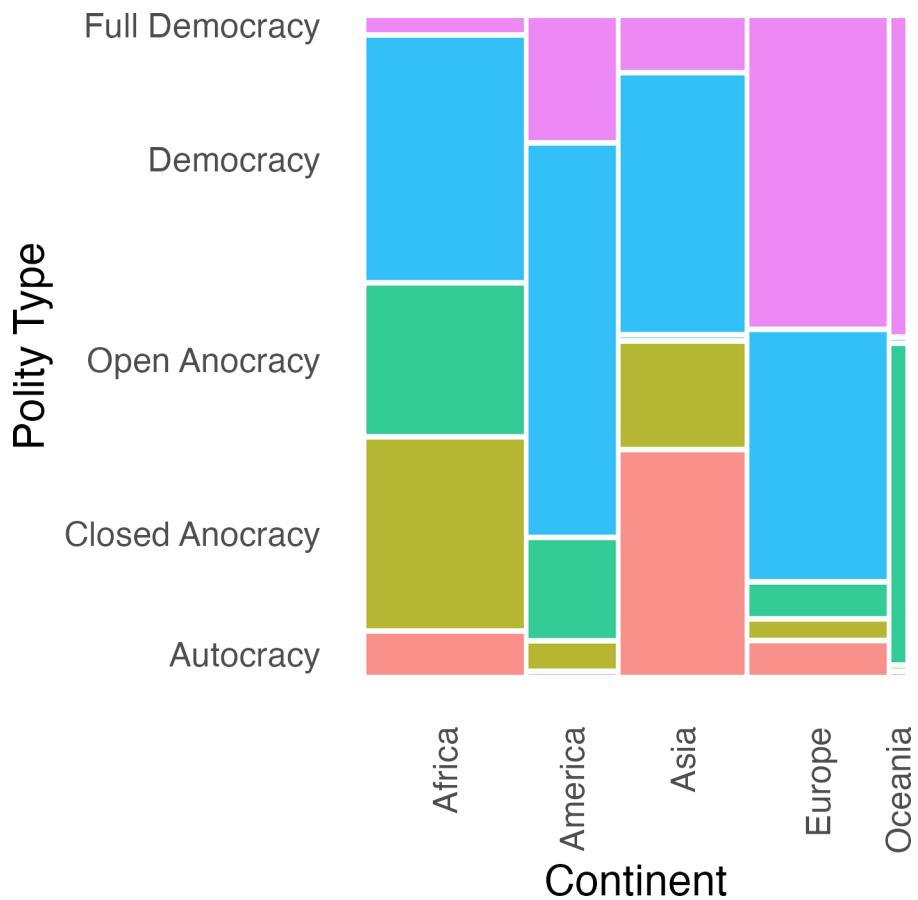
## Warning: `unite\_()` was deprecated in tidyverse 1.2.0.  
## Please use `unite()` instead.  
## This warning is displayed once every 8 hours.  
## Call `lifecycle::last\_lifecycle\_warnings()` to see where this warning was generated.



これで出来上がりですが、"table"オブジェクトを `plot()` に渡した結果とあまり変わらないですね。続いて、この図を少し改良してみましょう。まずはセルの色分けですが、これは `fill` に色分けする変数をマッピングするだけです。今回は政治体制ごとにセルを色分けしましょう。また、文字を大きめにし、横軸の目盛りラベルを回転します。

```
1 Mosaic_df %>%
2   ggplot() +
3   geom_mosaic(aes(x = product(Polity, Continent), fill = Polity),
4               na.rm = TRUE) +
5   labs(x = "Continent", y = "Polity Type") +
6   theme_minimal(base_size = 16) +
7   theme(legend.position = "none",
8         panel.grid     = element_blank(),
9         axis.text.x   = element_text(angle = 90, vjust = 0.5, hjust = 1))

## Warning: `unite_()` was deprecated in tidyverse 1.2.0.
## Please use `unite()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated
```

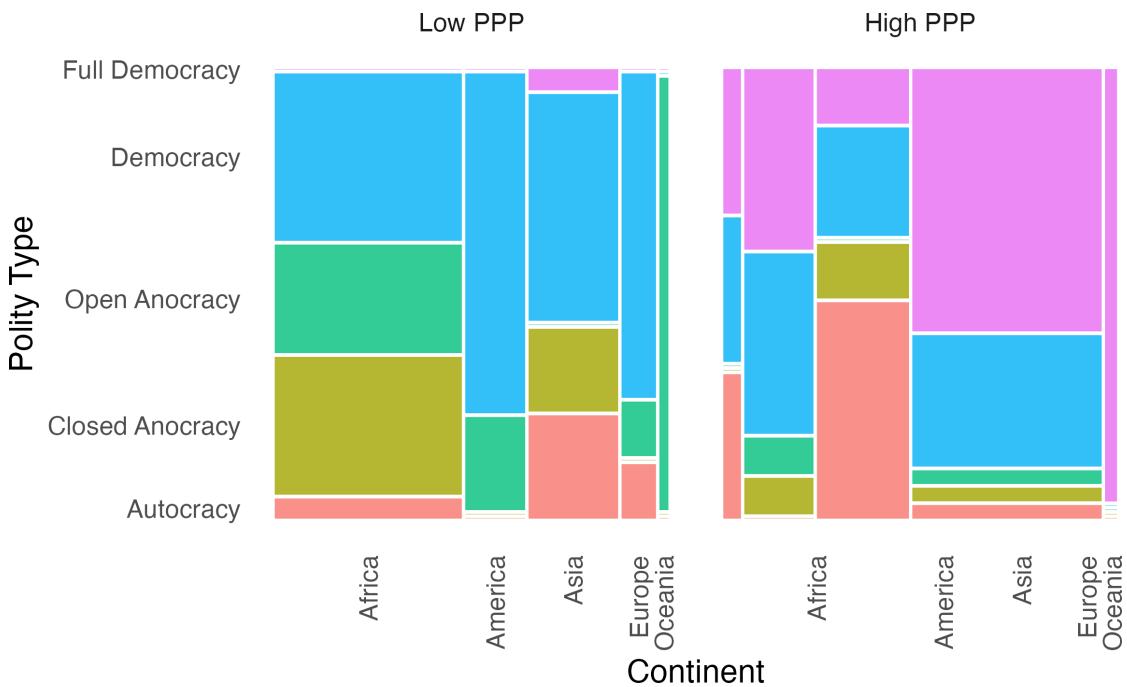


次元を追加するためにはファセット分割を使います。たとえば、一人当たり PPP GDP の高低 (PPP) でファセットを分割する場合、`facet_wrap(~PPP)` レイヤーを足すだけです。

```

1 Mosaic_df %>%
2   ggplot() +
3   geom_mosaic(aes(x = product(Polity, Continent), fill = Polity),
4               na.rm = TRUE) +
5   labs(x = "Continent", y = "Polity Type") +
6   facet_wrap(~PPP, ncol = 2) +
7   theme_minimal(base_size = 16) +
8   theme(panel.grid      = element_blank(),
9         axis.text.x    = element_text(angle = 90, vjust = 0.5, hjust = 1))
10

```



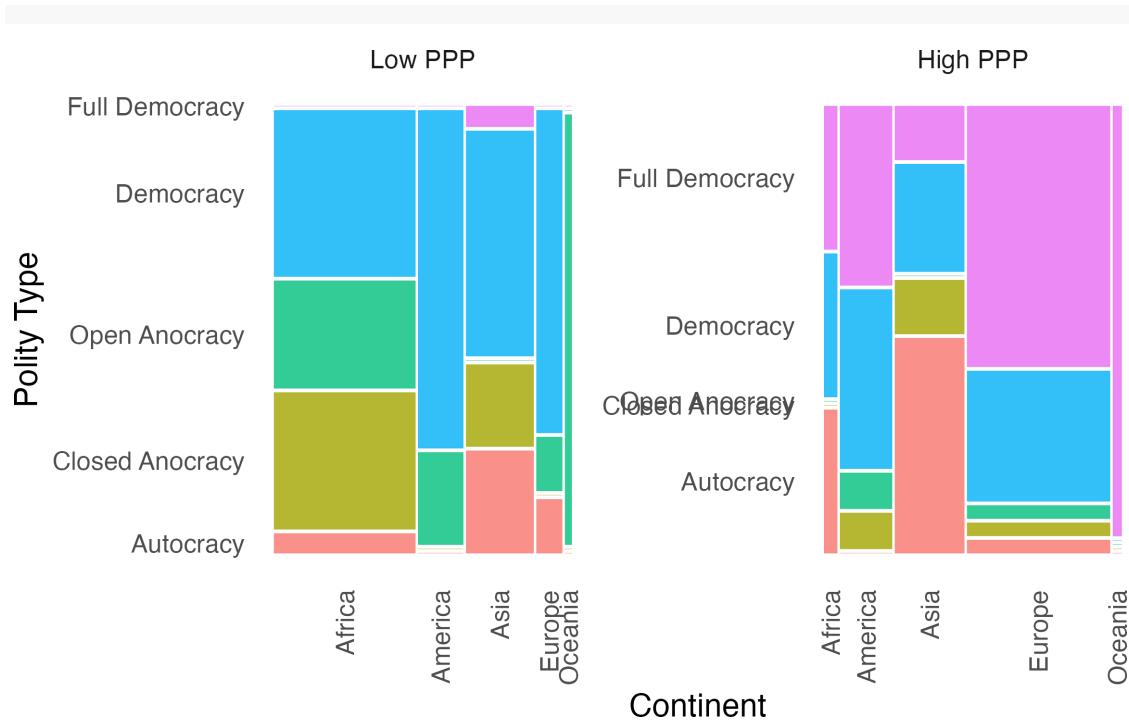
ただし、一つ問題があります。それは目盛りラベルの位置です。例えば、右側の横軸目盛りラベルの場合、セルの位置とラベルの位置がずれています。これは2つのファセットが同じ目盛りを共有し、左側の方に合わせられたため生じるものです。よく見ると横軸も縦軸も目盛りラベルに位置が同じであることが分かります。これを解消するためには、`facet_wrap()`の内部に `scale = "free"`を指定します<sup>5)</sup>。これは各ファセットが独自のスケールを有することを意味します。

```

1 Mosaic_df %>%
2   ggplot() +
3   geom_mosaic(aes(x = product(Polity, Continent), fill = Polity),
4               na.rm = TRUE) +
5   labs(x = "Continent", y = "Polity Type") +
6   facet_wrap(~PPP, ncol = 2, scale = "free") +
7   theme_minimal(base_size = 16) +
8   theme(legend.position = "none",
9         panel.grid      = element_blank(),
10        axis.text.x     = element_text(angle = 90, vjust = 0.5, hjust = 1))

```

5) 他にも縦軸のみ共有する"free\_x"、横軸のみ共有する"free\_y"があり、既定値は"fixed"です。

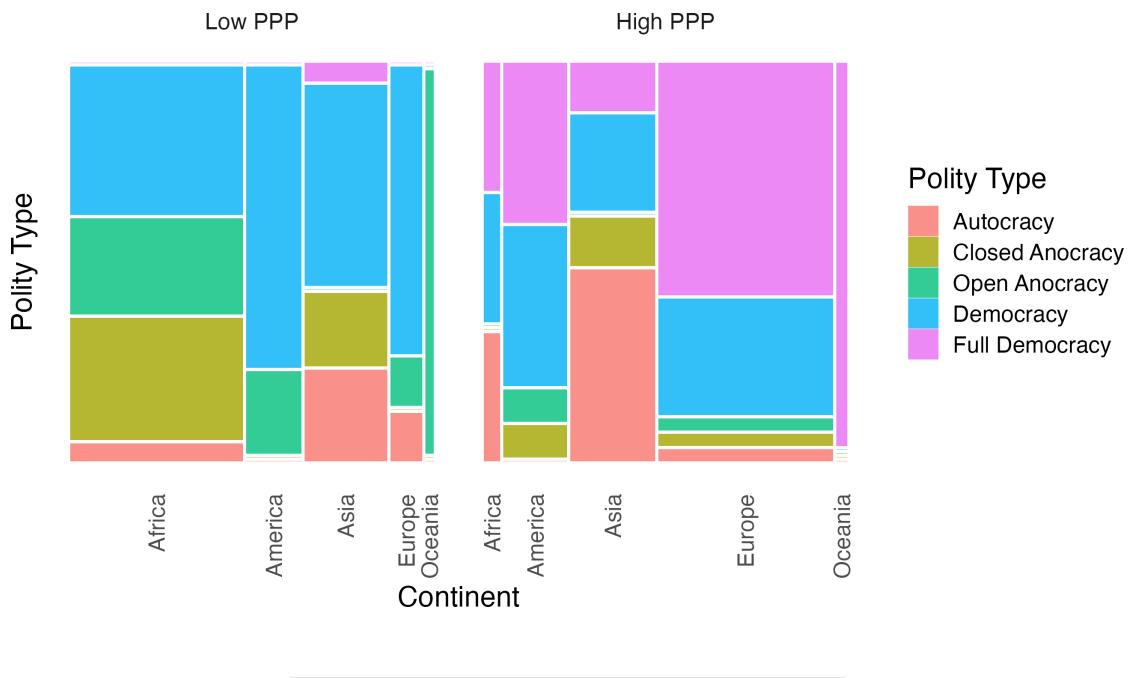


右側ファセットの横軸ラベルが重なってしまいましたが、これでとりあえず完成です。アフリカにおける Open Anocracy と Closed Anocracy の頻度が 0 であるため、これは仕方ありません。一つの対処方法としては以下のように縦軸目盛りを削除し、凡例で代替することが考えられます。

```

1 Mosaic_df %>%
2   ggplot() +
3   geom_mosaic(aes(x = product(Polity, Continent), fill = Polity),
4               na.rm = TRUE) +
5   labs(x = "Continent", y = "Polity Type", fill = "Polity Type") +
6   facet_wrap(~PPP, ncol = 2, scale = "free_x") +
7   theme_minimal(base_size = 16) +
8   theme(panel.grid = element_blank(),
9         axis.text.y = element_blank(),
10        axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))

```



## 20.16 その他のグラフ

The R Graph Gallery では本書で紹介できなかった様々な図のサンプルおよびコードを見るることができます。ここまで読み終わった方なら問題なくコードの意味が理解できるでしょう。{ggplot2}では作成できないグラフ（アニメーションや3次元図、インタラクティブなグラフ）についても、他のパッケージを利用した作成方法について紹介されているので、「こんな図が作りたいけど、作り方が分からん！」の時には、まず The R Graph Gallery に目を通してみましょう。

第 V 部

再現可能な研究



## 第 21 章

# R Markdown [基礎]

### 21.1 R Markdown とは

R Markdown とは名前通り、R と Markdown が結合されたものです。本書を通じて R を紹介してきましたが、Markdown は初めて出てくるものです。John Gruber と Aaron Swartz が 2004 年提案した軽量 Markup 言語ですが、そもそも Markup 言語とはなんでしょうか。

Markup 言語を知るためににはプレーンテキスト（plain text）とリッチテキスト（rich text、またはマルチスタイルテキスト）の違いについて知る必要があります。プレーンテキストとは書式情報などが含まれていない純粋なテキストのみで構成されている文書です。書式情報とは文書の余白、文字の大きさ、文字の色などがあります。これまで RStudio 上で書いてきた R コードもプレーンテキストです。コードに色が自動的に付けられますが、これは RStudio が色付けをしてくれただけで、ファイル自体はテキストのみで構成されています。macOS のTextEdit、Windows のメモ帳、Linux GUI 環境の gedit や KATE、CLI 環境下の vim、Emacs、nano などで作成したファイルは全てプレーンテキストです。これらのテキストエディターには書式設定や図表の挿入などの機能は付いておりません。一方、リッチテキストとは書式情報だけでなく、図表なども含まれる文書です。Microsoft Word とかで作成したファイルがその代表的な例です。他にも Pages や LibreOffice Writer から作成したファイルなどがあります。これらのワードプロセッサーソフトウェアは書式の設定や図表・リンクの挿入などができます。そして、Markup 言語とはプレーンテキストのみでリッチテキストを作成するための言語です。

Markup 言語の代表的な存在が HTML です。そもそも HTML の ML は Markup Language の略です。読者の皆さんのがウェブブラウザから見る画面のほとんどは HTML で書かれています。この『私たちの R』も HTML です。この文書には図表があり、太字、見出し、水平線など、テキスト以外の情報が含んでいます。しかし、HTML は純粋なテキストのみで書かれており、ウェブブラウザ（Firefox、Chrome、Edge など）がテキストファイルを読み込み、解釈して書式が付いている画面を出力してくれます。例えば、リンク（hyperlink）について考えてみましょう。「SONG の HP」をクリックすると宋のホームページに移動します。ある単語をクリックすることで、他のウェブサイトへ飛ばす機能を付けるためには HTML ファイルの中に、以下のように入力します。

```
<a href="https://www.jaysong.net">SONG の HP</a>
```

これをウェブブラウザが自動的に「SONG の HP」と変換し、画面に出力してくれます。これは、書式情報などが必要な箇所にコードを埋め込むことによって実現されます。そして、この Markup 言語をより単純な文法で再構成したものが Markdown です。例えば、以上の HTML は Markdown では以下のように書きます。

[SONG の HP] (<https://www.jaysong.net>)

同じ意味のコードですが、Markdown の方がより簡潔に書けます。この Markdown は最終的に HTML や Microsoft Word、PDF 形式で変換されます。一般的には HTML 出力を使いますが、自分の PC に LaTeX 環境が用意されている場合は PDF 形式で出力することも可能であり、個人的には推奨しておりませんが、Microsoft Word 文書ファイルへ変換することも可能です。また、HTML (+ JavaScript) へ出力可能であることを利用し、スライドショー、e-Book、ホームページの作成にも Markdown が使えます。

R Markdown は Markdown に R コードとその結果を同時に載せることができる Markdown です。それでもピンと来ない方も多いでしょう。それでは、とりあえず、R Markdown をサンプルコードから体験してみましょう。

## 21.2 とりあえず Knit

R Markdown の文法について説明する前に、とりあえず R Markdown というのがどういうものかを味見してみます。既に述べたように R Markdown には R コードも含まれるため、事前にプロジェクトを生成してから進めることをおすすめします。

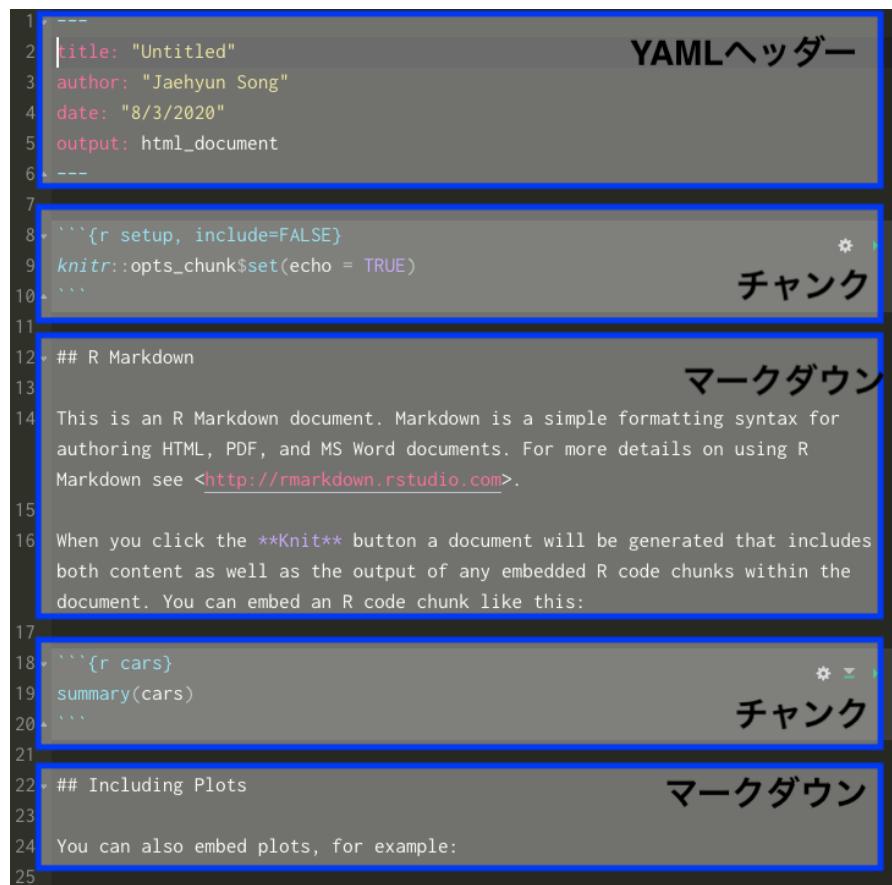
R Markdown ファイルを生成するには File -> New File -> R Markdown... を選択します。Title: と Author: には文書のタイトルと作成者を入力します。Default Output Format はデフォルトは HTML となっていますが、このままにしておきましょう。ここでの設定はいつでも変更可能ですので、何も触らずに OK を押しても大丈夫です。

OK を押したら自動的に R Markdown のサンプルファイルが生成されます。そしたら Source ペインの上段にあるボタン<sup>1)</sup>をクリックしてみましょう。最初はファイルの保存ダイアログが表示されますが、適切な場所（プロジェクトのフォルダなど）に保存すれば、自動的に Markdown 文書を解釈し始めます。処理が終われば Viewer ペインに何かの文章が生成されます。これから内容を進める前に Source ペインと Viewer ペインの中身をそれぞれ対応しながら、どのような関係があるのかを考えてみましょう。

R Markdown ファイルは大きく 3 つの領域に分けることができます（図@ref(fig:rmarkdown-sample-1)）。まず、最初に---と---で囲まれた領域はヘッダー（Header）と呼ばれる領域です。ここでは題目、作成者情報の入力以外にも、文書全体に通じる設定を行います。これは第 21.5 節で解説します。次は R Markdown の核心部であるチャンク（Chunk）です。チャンクは```{r}と```で囲まれた領域であり、R コードが入る箇所です。チャンクに関しては第 21.3 節の後半と第 21.4 節で解説します。その他の領域がマークダウン（Markdown）であり、文書に該当します。

---

<sup>1)</sup> Knit は「ニット」と読みます。



YAMLヘッダー

```
1 ---  
2 | title: "Untitled"  
3 | author: "Jaehyun Song"  
4 | date: "8/3/2020"  
5 | output: html_document  
6 ---  
7  
8 `r setup, include=FALSE}  
9 knitr::opts_chunk$set(echo = TRUE)  
10 ...  
11  
12 ## R Markdown  
13  
14 This is an R Markdown document. Markdown is a simple formatting syntax for  
authoring HTML, PDF, and MS Word documents. For more details on using R  
Markdown see <http://rmarkdown.rstudio.com>.  
15  
16 When you click the Knit button a document will be generated that includes  
both content as well as the output of any embedded R code chunks within the  
document. You can embed an R code chunk like this:  
17  
18 `r cars}  
19 summary(cars)  
20 ...  
21  
22 ## Including Plots  
23  
24 You can also embed plots, for example:  
25
```

チャンク

マークダウン

チャンク

マークダウン

図 21.1: R Markdown のサンプルページ

まずは、文章の書き方から説明します。非常に簡単な文法で綺麗、かつ構造化された文書が作成可能であり、これに慣れると Markdown 基盤のノートアプリなどを使って素早くノート作成、メモが出来ます。

### 21.3 Markdown 文法の基本

まずは、Markdown の文法について解説します。ここでは Markdown 文書内に以下のようなことを作成する方法を実際の書き方と、出力画面を対比しながら解説していきます。

- 改行
- 強調: 太字、イタリック、アンダーライン、取り消し線
- 箇条書き
- 見出し
- 区切り線
- 表
- 画像
- リンク
- 脚注
- 数式
- 引用
- コメント
- R コード

### 21.3.1 改行

Markdown における改行はやや特殊です。特殊といっても難しいことはありません。普段よりもう一行改行するだけです。Markdown の場合、1 回の改行は改行として判定されず、同じ行の連続と認識します。たとえば、**Input** のように入力すると **Output** のように文章 1 と文章 2 が繋がります。

#### **Input:**

```
文章 1  
文章 2
```

#### **Output:**

文章 1 文章 2

文章 1 と文章 2 を改行するためにはもう一行、改行する必要があります。以下の例を見てください。

#### **Input:**

文章 1

文章 2

**Output:**

文章 1

文章 2

こうすることで段落間の間隔を強制的に入れることとなり、作成者側にも読みやすい文書構造になります<sup>2)</sup>。

### 21.3.2 強調

文章の一部を強調する方法として**太字**、**イタリック**<sup>3)</sup>、**アンダーライン**があり、強調ではありませんが、ついでに取り消し線についても紹介します。いずれも強調したい箇所を記号で囲むだけです。

**Input:**

文章の一部を**\*\*太字\*\***にしてみましょう。

\*イタリック\*もいいですね。

~~取り消し線~~はあまり使わないかも。

**<u>アンダーライン</u>**は HTML タグを使います。

**Output:**

<sup>2)</sup> HTML に慣れている読者なら<br/>を使った改行もできます。ただし、一般的な Markdown の改行よりも行間が短めです。HTML に慣れている読者さんならお分かりかと思いますが、これは Markdown の改行（2 行改行）は HTML の<p></p>に相当するからです。

<sup>3)</sup> ただし、イタリックの場合、日本語には使わるのが鉄則です。強調の意味としてイタリックを使うのはローマ字くらいです。

文章の一部を**太字**にしてみましょう。

イタリックもいいですね。

取り消し線はあまり使わないかも。

アンダーラインは HTML タグを使います。

### 21.3.3 箇条書き

箇条書きには順序なしと順序付きがあります。順序なしの場合\*または-の後に半角スペースを 1 つ入れるだけです。また、2 文字以上の字下げで下位項目を追加することもできます。

#### Input:

- 項目 1
  - 項目 1-1
  - 項目 1-2
    - 項目 1-2-1
      - 項目 1-2-1-1
    - 項目 1-2-2
- 項目 2
- 項目 3

#### Output:

- 項目 1
  - 項目 1-1
  - 項目 1-2
    - \* 項目 1-2-1
      - 項目 1-2-1-1
    - \* 項目 1-2-2
- 項目 2
- 項目 3

続きまして順序付き箇条書きですが、これは-（または\*）を数字. に換えるだけです。順序なしの場合と違って数字の後にピリオド（.）が付くことに注意してください。また、下位項目を作成する際、順序なしはスペース 2 つ以上が必要でしたが、順序付きの場合、少なくとも 3 つが必要です。

**Input:**

1. 項目 1
  1. 項目 1-1
  2. 項目 1-2
2. 項目 2
  - \* 項目 2-1
  - \* 項目 2-2
3. 項目 3

**Output:**

1. 項目 1
  1. 項目 1-1
  2. 項目 1-2
2. 項目 2
  - 項目 2-1
  - 項目 2-2
3. 項目 3

#### 21.3.4 見出し

章、節、段落のタイトルを付ける際は#を使います。#の数が多いほど文字が小さくなります。章の見出しを##にするなら節は###、小節または段落は####が適切でしょう。見出しあは####まで使えます。

**Input:**

```
# 見出し 1
## 見出し 2
```

```
### 見出し 3  
#### 見出し 4
```

**Output:**

見出し 1

見出し 2

見出し 3

見出し 4

### 21.3.5 区切り線

区切り線は---または\*\*\*を使います。

**Input:**

---

**Output:**

---

### 21.3.6 表

Markdown の表は非常にシンプルな書き方をしています。行は改行で、列は|で区切れます。ただ、表の第 1 行はヘッダー（変数名や列名が表示される行）扱いとなり、ヘッダーと内容の区分は|---|で行います。以下は Markdown を利用した簡単な表の書き方です。ここでは可読性のためにスペースを適宜入れましたが、スペースの有無は結果に影響を与えません。

**Input:**

ID	Name	Math	English	Favorite food
1	John	Math	English	Italian

1	SONG	15	10	Ramen	
2	Yanai	100	100	Cat food	
3	Shigemura	80	50	Raw chicken	
4	Wickham	80	90	Lamb	

### Output:

ID	Name	Math	English	Favorite food
1	SONG	15	10	Ramen
2	Yanai	100	100	Cat food
3	Shigemura	80	50	Raw chicken
4	Wickham	80	90	Lamb

1 行目はヘッダーであり、太字かつ中央揃えになります。2 行目以降はデフォルトでは左揃えになりますが。ただし。|---|をいじることによって当該列の揃えを調整できます。|:---|は左（デフォルト）、|---:|は右、|:---:|は中央となります。また-の個数は 1 個以上なら問題ありません。つまり、|-|も|---|も同じです。

### 21.3.7 画像

R Markdown に画像を入れるには! [代替テキスト] (ファイル名) と入力します。当たり前ですが、画像ファイルがワーキングディレクトリにない場合はパスを指定する必要があります。[代替テキスト] は画像を読み込めなかった場合のテキストを意味します。これは画像が読み込めなかった場合の代替テキストでもありますが、視覚障害者用のウェブブラウザーのためにも使われます。これらのウェブブラウザーはテキストのみ出力されるものが多く、画像の代わりには代替テキストが読み込まれます。

例えば、HTML フォルダー内の favicon.png というファイルを読み込むとしたら以下のように書きます。

### Input:

! [『私たちの R』ロゴ] (HTML/favicon.png)

**Output:**



図 21.2: 『私たちの R』ロゴ

### 21.3.8 リンク

ハイパーリンクは [テキスト] (URL) のような形式で書きます。[] 内は実際に表示されるテキストであり、() は飛ばす URL になります。

**Input:**

毎日 1 回は [SONG のホームページ] (<https://www.jaysong.net>) ヘアクセスしましょう。

**Output:**

毎日 1 回は SONG のホームページヘアクセスしましょう。

### 21.3.9 脚注

脚注は [^固有識別子] と [^固有識別子]: 脚注内容の 2 つの要素が必要です。まず、文末脚注を入れる箇所に [^xxxx] を挿入します。xxxx は任意の文字列れ構いません。しかし、同じ R Markdown 内においてこの識別子は被らないように注意してください。実際の脚注の内容は [^xxxx]: 内容のように入力します。これはどこに位置しても構いません。文書の途中でも、最後に入れても、脚注の内容は文末に位置します。ただし、脚注を入れる段落のすぐ後の方が作成する側としては読みやすいでしょう。

**Input:**

これは普通の文章です [^foot1]。

[^foot1]: これは普通の脚注です。

**Output:**

これは普通の文章です<sup>4)</sup>。

### 21.3.10 数式

インライン数式は\$数式\$で埋め込むことができます。数式は LaTeX の書き方とほぼ同じです。ちなみに、R Markdown の数式は MathJax によってレンダリングされます。この MathJax ライブラリは HTML に埋め込まれているのではないため、インターネットに接続せずに HTML ファイルを開くと数式が正しく出力されません。

**Input:**

AINSHUTAINと言えば、\$e = mc^2\$でしょう。

**Output:**

AINSHUTAINと言えば、 $e = mc^2$  でしょう。

数式を独立した行として出力する場合は、\$の代わりに\$\$を使用します。

**Input:**

独立した数式の書き方

```
$$
y_i \sim \text{Normal}(\mathbf{X} \boldsymbol{\beta}, \sigma).
$$
```

**Output:**

---

<sup>4)</sup> これは普通の脚注です。

独立した数式の書き方

$$y_i \sim \text{Normal}(\mathbf{X}\beta, \sigma).$$

もし数式が複数の行で構成されている場合は \$\$ 内に aligned 環境 (`\begin{aligned}` ~`\end{aligned}`) を使用します。もちろん、LaTeX と使い方は同じです。

**Input:**

複数の行にわたる数式の書き方

```
$$
\begin{aligned}
Y_i &\sim \text{Bernoulli}(\theta_i), \\
\theta_i &= \text{logit}^{-1}(y_i^*), \\
y_i^* &= \beta_0 + \beta_1 x_1 + \beta_2 z_1.
\end{aligned}
$$
```

**Output:**

複数の行にわたる数式の書き方

$$\begin{aligned} Y_i &\sim \text{Bernoulli}(\theta_i), \\ \theta_i &= \text{logit}^{-1}(y_i^*), \\ y_i^* &= \beta_0 + \beta_1 x_1 + \beta_2 z_1. \end{aligned}$$

ここまで見ればお分かりかと思いますが、 \$\$ の中には LaTeX コマンドが使用可能で す。たとえば、行列を作成する際は以下のように `\begin{bmatrix}` 環境を使います。

**Input:**

行列の書き方

```
$$
```

```
x = \begin{bmatrix}
x_{11} & x_{12} \\
x_{21} & x_{22} \\
x_{31} & x_{32}
\end{bmatrix}.
$$
```

**Output:**

行列の書き方

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{bmatrix}.$$

### 21.3.11 引用

引用の際は文章の最初に>を入れるだけです。>の後に半角のスペースが1つ入ります。

**Input:**

「政治とは何か」についてイーストンは以下のように定義しました。

> [A] political system can be designated as those interactions through which value

**Output:**

「政治とは何か」についてイーストンは以下のように定義しました。

[A] political system can be designated as those interactions through which values are authoritatively allocated for a society.

### 21.3.12 コメント

R Markdown にもコメントを付けることができます。とりあえず書いたが要らなくなつた段落や文章があって、消すことがもったいない場合はコメントアウトするのも 1

つの方法です。ただし、コメントアウトの方法は R は#でしたが、これは R Markdown では見出しの記号です。R Markdown のコメントは<!--と-->で囲みます。

**Input:**

文章 1

```
<!--
ここはコメントです。
-->
```

文章 2

**Output:**

文章 1

文章 2

### 21.3.13 コード

以上の内容まで抑えると、R Markdown を使って、簡単な文法のみで構造化された文書が作成できます。しかし、R Markdown の意義は文章とコード、結果が統合されることです。それでは文書に R コードを入れる方法について紹介します。

コードは ```{r} と ``` の間に入力します。これだけです。これでコードと結果が同時に表示されます。たとえば、`print("Hello World!")` を走らすコードを入れてみます。

**Input:**

```
"Hello World!"を出力するコード
```{r}
print("Hello World!")
```
```

**Output:**

“Hello World!” を出力するコード

```
1 print("Hello World!")
```

```
## [1] "Hello World!"
```

```{r} と ```で囲まれた範囲を R Markdown では**チャンク (Chunk)** と呼びます。このチャンク内では R と全く同じことが出来ます。パッケージやデータの読み込み、オブジェクトの生成、データハンドリング、可視化など、全てです。

**Input:**

```
```{r}
# パッケージの読み込み
pacman::p_load(tidyverse)
# R 内蔵データセットの iris を使った可視化
iris %>%
  mutate(Species2 = recode(Species,
                            "setosa"      = "セトナ",
                            "versicolor"  = "バーシクル",
                            "virginica"   = "バージニカ")) %>%
  ggplot() +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Species2)) +
  labs(x = "萼片の長さ (cm)", y = "萼片の幅 (cm)", color = "品種") +
  theme_minimal(base_size = 12)
```
```

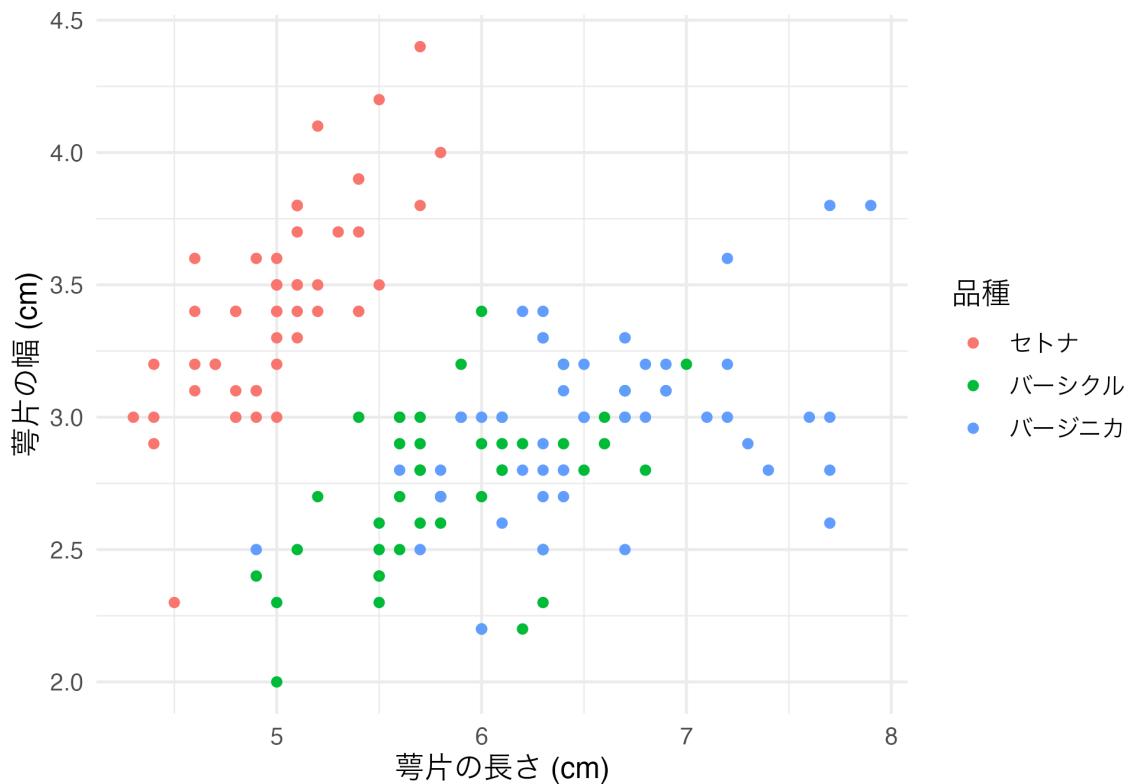
**Output:**

```
1 # パッケージの読み込み
2 pacman::p_load(tidyverse)
3 # R 内蔵データセットの iris を使った可視化
4 iris %>%
5   mutate(Species2 = recode(Species,
6                            "setosa"      = "セトナ",
7                            "versicolor"  = "バーシクル",
```

```

8           "virginica"  = "バージニカ")) %>%
9   ggplot() +
10  geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Species2)) +
11  labs(x = "萼片の長さ (cm)", y = "萼片の幅 (cm)", color = "品種") +
12  theme_minimal(base_size = 12)

```



他にも文中に R コードを埋め込むことも可能です。例えば、ベクトル `X <- c(2, 3, 5, 7, 12)` があり、この平均値を文中で示したいとします。もちろん、文中に「5.8」と書いても問題はありません。しかし、実は `X` の入力ミスが見つかり、実は `c(2, 3, 5, 7, 11)` にならざるを得ません。この「5.8」と書いた箇所を見つけて 5.6 と修正したいといけません。これは非常に面倒な作業であり、ミスも起こりやすいです。文中で R コードを入れるために ``r`` のように入力します。

```

```{r}
X <- c(2, 3, 5, 7, 11)

```

...

変数 `X` の平均値は `r mean(X)` です。

**Output:**

```
1 X <- c(2, 3, 5, 7, 11)
```

変数 X の平均値は 5.6 です。

ここで `X` ですが、単に ` ` で囲まれただけではコードと認識されません。これは主に文中に短いコードを入れる際に使う機能です。

---

## 21.4 チャンクのオプション

既に説明しましたとおり、R Makrdown は R + Markdown です。R はチャンク、Markdown はチャンク外の部分に相当し、それぞれのカスタマイズが可能です。分析のコードと結果はチャンクにオプションを付けることで修正可能であり、文章の部分は次節で紹介するヘッダーで調整できます。ここではチャンクのオプションについて説明します。

チャンクは `~~~{r}` で始まりますが、実は {r} の箇所にオプションを追加することができます。具体的には {r チャンク名, オプション 1, オプション 2, ...} といった形です。まずはチャンク名について解説します。

### 21.4.1 チャンク名とチャンク間依存関係

チャンク名は {r チャンク名} で指定し、r とチャンク名の間には, が入りません。これはチャンクに名前をしていするオプションですが、多くの場合分析に影響を与えることはありません。このチャンク名が重要となるのは cache オプションを付ける場合です。

cache オプションは処理結果を保存しておくことを意味します。チャンク内のコードは Knit する度に計算されます。もし、演算にかなりの時間を費やすコードが含まれてい

る場合、Knit の時間も長くなります。この場合、`cache = TRUE` オプションを付けておくと、最初の Knit 時に結果をファイルとして保存し、次回からはその結果を読み込むだけとなります。時間が非常に節約できるため、よく使われるオプションの 1 つです。ただし、チャンク内のコードが修正された場合、Knit 時にもう一回処理を行います。コードの実質的な内容が変わらなくても、つまり、スペースを 1 つ入れただけでも再計算となります。

ここで 1 つ問題が生じます。たとえば、以下のようなコードを考えてみてください。

```
```{r}
X <- c(2, 3, 5, 7, 10)
```

```{r, cache = TRUE}
mean(X)
````
```

この構造に問題はありません。しかし、ここで `X` の 5 番目の要素を 11 に修正したとします。そしてもう一回 Knit を行ったらどうなるでしょうか。正解は「何も変わらない」です。新しい `mean(X)` の結果は 5.6 のはずですが、5.4 のままです。なぜなら、2 番目のチャンクの結果は既に保存されており、コードも修正していないからです。もう一回強調しておきますが、`cache = TRUE` の状態で当該チャンクが修正されない場合、結果は変わりません。

このようにあるチャンクの内容が他のチャンク内容に依存しているケースがあります。この場合、`dependson` オプションを使います。使い方は `dependson = "依存するチャンク名"` です。もし、1 番目のチャンク名を `define_X` としたら、`dependson = "define_X"` とオプションを加えます。

```
```{r define_X}
X <- c(2, 3, 5, 7, 10)
```

```{r, cache = TRUE, dependson = "define_X"}
mean(X)
````
```

...

このようにチャunk名と `cache`、`dependson` オプションを組み合わせると、依存するチャunkの中身が変わったら、`cache = TRUE` でも再計算を行います。

#### 21.4.2 コードまたは結果の表示/非常時

次は「コードだけ見せたい」、「結果だけ見せたい」場合使うオプションを紹介します。これはあまり使わないかも知れませんが、本書のような技術書にはよく使う機能です。コードのみ出力し、計算を行わない場合は `eval = FALSE` オプションを、コードを見せず、結果のみ出力する場合は `echo = FALSE` を指定するだけです。

他にもコードと結果を両方隠すことも可能です。つまり、チャunk内のコードは実行されるが、そのコードと結果を隠すことです。この場合に使うオプションが `include` であり、既定値は `TRUE` です。チャunkオプションに `include = FALSE` を追加すると、当該チャunkのコードは実行されますが、コードと結果は表示されません。

#### 21.4.3 プロット

既に見てきた通り、R Markdown は作図の結果も出力してくれます。そこで、図のサイズや解像度を変えることもできます。ここではプロットに関するいくつかのオプションを紹介します。

- `fig.height`: 図の高さ。単位はインチ。デフォルト値は 7
- `fig.width`: 図の幅。単位はインチ。デフォルト値は 7
- `fig.align`: 図の位置。デフォルトは "left"。"center" の場合、中央揃え、"right" の場合は右揃えになる。
- `fig.cap`: 図のキャプション
- `dpi`: 図の解像度。デフォルトは 72。出版用の図は一般的に 300 以上を使う

実際に高さ 5 インチ、幅 7 インチ、中央揃え、解像度 72dpi の図を作成し、キャプションとして「`iris` データセットの可視化」を付けてみましょう。

**Input:**

```
```{r, fig.height = 5, fig.width = 7, fig.align = "center", fig.cap = "iris`データセットの  
iris %>%  
  mutate(Species2 = recode(Species,  
    "setosa"      = "セトナ",  
    "versicolor" = "バーシクル",  
    "virginica"  = "/バージニカ")) %>%  
  ggplot() +  
  geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Species2)) +  
  labs(x = "萼片の長さ (cm)", y = "萼片の幅 (cm)", color = "品種") +  
  theme_minimal(base_size = 12)  
```
```

Output:

```
1 iris %>%  
2   mutate(Species2 = recode(Species,  
3     "setosa"      = "セトナ",  
4     "versicolor" = "バーシクル",  
5     "virginica"  = "/バージニカ")) %>%  
6   ggplot() +  
7   geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Species2)) +  
8   labs(x = "萼片の長さ (cm)", y = "萼片の幅 (cm)", color = "品種") +  
9   theme_minimal(base_size = 12)
```

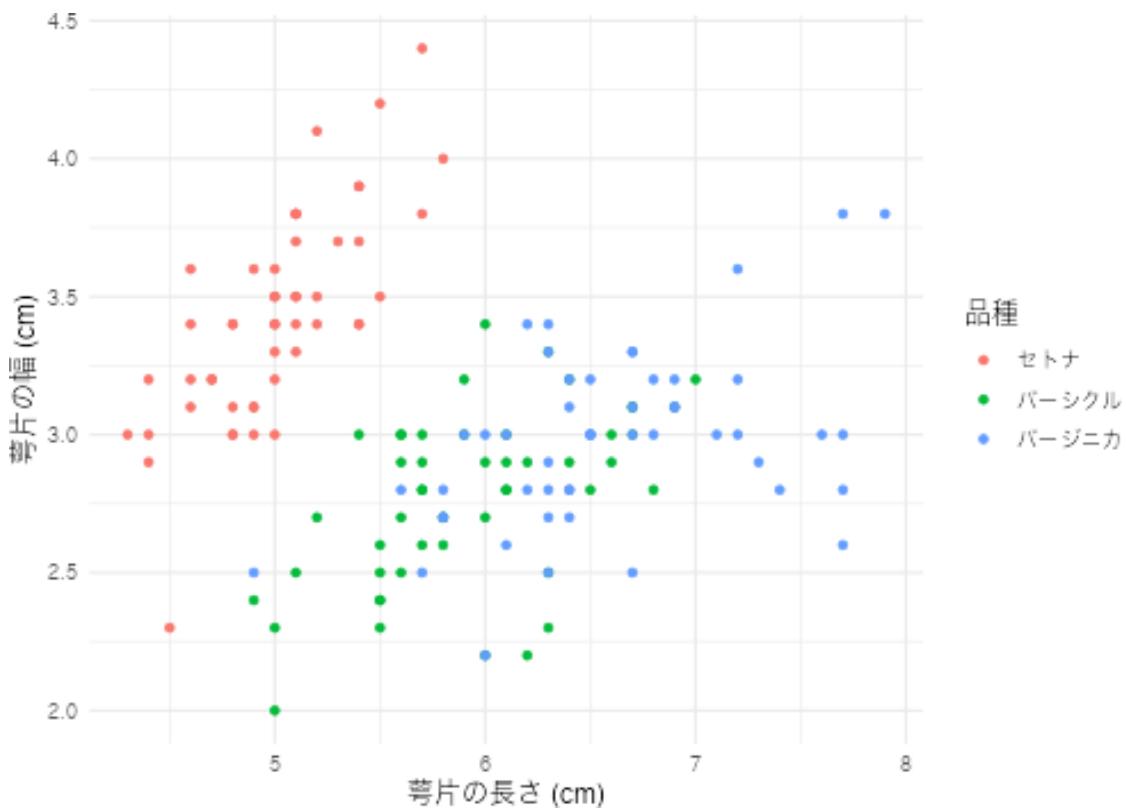


図 21.3: ‘iris’データセットの可視化

#### 21.4.4 コードの見栄

第 10 章ではスクリプトの書き方を紹介しましたが、常にその書き方に則った書き方をしているとは限りません。自分だけが見るコードなら別に推奨されない書き方でも問題ないかも知れませんが、R Markdown の結果は他人と共有するケースが多いため、読みやすいコードを書くのも大事です。ここで便利なオプションが `tidy` オプションです。`tidy = TRUE` を加えると、自動的にコードを読みやすい形に調整してくれます。たとえば、以下のコードは字下げもなく、スペースもほとんど入れていないコードですが、`tidy = TRUE` を付けた場合と付けなかった場合の出力結果の違いを見てみましょう。

**Input:**

```
```{r, eval = FALSE}
for(i in 1:10){
  print(i*2)
}
````
```

**Output:**

```
1 for(i in 1:10){
2   print(i*2)
3 }
```

**Input:**

```
```{r, eval = FALSE, tidy = TRUE}
for(i in 1:10){
  print(i*2)
}
````
```

**Output:**

```
1 for (i in 1:10) {
2   print(i * 2)
3 }
```

`tidy = TRUE` を付けただけで、読みやすいコードになりました。ちなみに `tidy` オプションを使うためには事前に `{formatR}` パッケージをインストールしておく必要があります。ただし、`{formatR}` パッケージは R Markdwon 内において読み込んでおく必要はありません。また、`{formatR}` パッケージは万能ではないため、普段から読みやすいコードを書くようにしましょう。

### 21.4.5 チャンクオプションのもう一つの書き方

これまでチャンクオプションは{r}の内部に指定すると述べましたが、チャンクオプションが多くなる場合、チャンクの第 1 行目が長くなり、コードの可読性が低下する可能性があります。ここで便利な機能が#|によるチャンクオプションの指定です。これはチャンク内部に#|を書いておけば、#|以降の内容がチャンクオプションとして認識される機能です。

図 21.3 の作図チャンクは以下のように書くことも可能です。

```
```{r}
#| fig.height: 5
#| fig.width: 7
#| fig.align: "center"
#| fig.cap: "`iris`データセットの可視化"
#| dpi: 72

iris %>%
  mutate(Species2 = recode(Species,
    "setosa"      = "セトナ",
    "versicolor" = "バーシクル",
    "virginica"  = "バージニカ")) %>%
  ggplot() +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Species2)) +
  labs(x = "萼片の長さ (cm)", y = "萼片の幅 (cm)", color = "品種") +
  theme_minimal(base_size = 12)
```
```

もう一つの方法は、全て(あるいは、ほとんど)のチャンクに渡って共通するチャンクオプションの場合、最初に宣言しておくことです。もし、全ての図を中央に揃え(fig.align = "center")、解像度を 300dpi (dpi = 300) にするなら、後述する YAML ヘッダーに続く最初のチャンクを以下のように書きます。

```
```{r, include = FALSE}
knitr::opts_chunk$set(fig.align = "center", dpi = 300)
```
```

こう書いておくと、その RMarkdown ファイルの全てのチャンクに `fig.align = "center"` と `dpi = 300` オプションが付くようになります。一部のチャンクに `fig.align` や `dpi` のオプションを付ける場合、当該チャンクのオプションが優先されます。

このようにチャンクオプションが見やすくなるメリットがありますが、現在（2022 年 03 月 03 日）の RStudio は、チャンク内部のオプション指定の入力補助に対応していないことに注意してください。

---

## 21.5 ヘッダーのオプション

R Markdown ファイルを生成すると、ファイルの最上段には以下のようなヘッダー（header）というものが生成されます。

```
---
title: "Untitled"
author: "Jaehyun Song"
date: "8/6/2020"
output: html_document
---
```

基本的には 4 つの項目が指定されており、それぞれ文書のタイトル（`title:`）、作成者名（`author:`）、作成日（`date:`）、出力形式（`output:`）があります。タイトルと作成者名はファイル生成時に指定した内容が、作成日は今日の日付が、出力形式は HTML で設定した場合 `html_document` になっています。R Markdown ヘッダーは YAML（やむる）形式で書かれています。こちらはいつでも修正可能であり、インライン R コードを埋め込むことも可能です。たとえば、作成日をファイル生成日でなく、Knit した日付にしたい場合は日付を出力する `Sys.Date()` 関数を使って、`date: "最終修正: `r Sys.Date()`"` のように書くことも可能です。他にもデフォルトでは指定されていません

が、`subtitle:` を使ってサブタイトルを指定したり、`abstract:` で要約を入れることも可能です。また、R Markdown のコメントは`<!--`と`-->`を使いますが、ヘッダー内のコメントは R と同様、`#`を使います。

ヘッダーで設定できる項目は数十個以上ですが、ここでは頻繁に使われる項目について紹介します。

### 21.5.1 目次の追加

R Markdown の文章が長くなったり、コードが多く含まれる場合、目次を入れたら文章の構造が一目で把握できるでしょう。目次は見出しに沿って自動的に生成されます。`#`は章、`##`は節といった形式です。

```
---
```

```
title: "タイトル"
subtitle: "サブタイトル"
author: "作成者名"
date: "最終修正: `r Sys.Date()`"
output:
  html_document:
    toc: FALSE
    toc_depth: 3
    toc_float: FALSE
    number_sections: FALSE
---
```

字下げには常に注意してください。`html_document` は `output:` の下位項目ですから、字下げを行います。また、`toc` は `html_document` の下位項目ですので、更に字下げをします。ここでは目次と関連する項目として以下の 4 つを紹介します。

- `toc:` 目次の出力有無
  - デフォルトは `FALSE`、目次を出力する際は `TRUE` にします。
- `toc_depth:` 目次の深さを指定
  - デフォルトは 3 であり、これはどのレベルの見出しまで目次に含むかをして

します。`toc_depth: 2`なら##見出しまで目次に出力されます

- `toc_float`: 目次のフローティング
  - デフォルトは `FALSE` です。これを `TRUE` にすると、目次は文書の左側に位置するようになります。文書をスクロールしても目次が付いてきます。
- `number_sections`: 見出しに通し番号を付けるか
  - デフォルトは `FALSE` です。これを `TRUE` にすると、見出しに通し番号が付きます。もちろん、付いた通し番号は目次でも出力されます。

### 21.5.2 コードのハイライト

R Markdown の場合、コードチャンク内のコードの一部に対して自動的に色付けを行います。たとえば、本書の場合、関数名は緑、引数は赤、文字列は青といった形です。これはコードの可読性を向上させる効果もあります。この色付けの詳細は `html_document`: の `highlight`: から調整できます。

```
---
```

```
title: "タイトル"
subtitle: "サブタイトル"
author: "作成者名"
date: "最終修正: `r Sys.Date()`"
output:
  html_document:
    highlight: "tango"

```

```
---
```

R Markdown 2.3 の場合、使用可能なテーマは以下の 10 種類です。自分の好みでハイライトテーマを替えてみましょう。

---

| highlight | 出力結果  |
|-----------|---|
| "tango"   | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "default" |   |

| highlight    | 出力結果  |
|--------------|---|
| "tango"      | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "pygments"   | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "kate"       | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "monochrome" | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "espresso"   | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "zenburn"    | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "haddock"    | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |

| highlight    | 出力結果  |
|--------------|---|
| "breezedark" | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |
| "textmate"   | <pre>i &lt;- 1:10 for (i in x) {   print(paste0("i = ", i)) }</pre> |

## 21.6 日本語が含まれている PDF の出力

日本語が含まれている PDF 出力には片桐智志さんの{rmdja}パッケージが提供するテンプレが便利です。{rmdja}パッケージの詳細は公式レポジトリに譲りますが、ここでは簡単な例をお見せします。以下の内容は自分の PC に LaTeX 環境が導入されている場合を想定しています<sup>5)</sup>。

まずは{rmdja}のインストールからです。CRAN に登録されていたいたため、GitHub 経由でインストールします。

```
# remotes:: の代わりに devtools:: も可
# pacman::p_install_gh() も可
remotes::install_github('Gedevan-Aleksizde/rmdja')
```

次は R Markdown 文書を作成しますが、RStudio の File > New File > R Markdown ... を選択します。左側の From Template を選択し、pdf article in Japanese を選択します。OK をクリックするとサンプル文書が表示されるので、YAML ヘッダー以下の内容を修正するだけです。図表番号などの相互参照 (cross reference) が初回 Knit では反映

<sup>5)</sup> 簡単な方法として、{rmdja}のインストール後、tinytex::install\_tinytex() で軽量版の LaTeX がインストールできます。

されない場合がありますが、2 回目からは表示されます。

## 第 22 章

# R Markdown [応用]

R Markdown でできることを紹介。使い方は説明しない

### 22.1 スライド作成

{xaringan}

### 22.2 書籍

{bookdown}

### 22.3 ホームページ

{distill}、{blogdown}

### 22.4 履歴書

{vitae}

## 22.5 パッケージ開発

{fusen}

## 22.6 Web アプリケーション

{shiny}

## 22.7 チュートリアル

{learnr}

## 第 23 章

### 表の作成

```
1 pacman::p_load(tidyverse, kableExtra, gt)
```

{kableExtra} と {gt}



## 第 24 章

### モデルの可視化



## 第 25 章

# 分析環境の管理

{renv}による分析環境の管理/共有



# 第 VI 部

## 中級者向け



## 第 26 章

# データハンドリング [応用編]

ここでは整形されていない状態のデータを



## 第 27 章

# 反復処理

### 27.1 概要

- `*apply()` 関数群と `map_*` 関数群
  - 引数が 2 つ以上の場合
  - データフレームと `{purrr}`
  - モデルの反復推定
    - サンプルの分割とモデル推定 (`split()` 利用)
    - サンプルの分割とモデル推定 (`nest()` 利用)
    - データの範囲を指定したモデル推定
    - 説明・応答変数を指定したモデル推定
- 

### 27.2 `*apply()` 関数群と `map_*` 関数群

まず、我らの盟友、`{tidyverse}`を読み込んでおきましょう。

```
1 pacman::p_load(tidyverse)
```

それでは、`*apply()` 関数群と `map_*` 関数群の動きとその仕組について調べてみましょう。まず、実習用データとして長さ 5 の numeric ベクトル `num_vec` を用意します。

```
1 num_vec <- c(3, 2, 5, 4, 7)
```

この `num_vec` の個々の要素に 2 を足す場合はどうすれば良いでしょうか。R はベクトル単位での演算が行われるため、`num_vec + 2` だけで十分です。`+` の右側にある 2 は長さ 1 のベクトルですが、`num_vec` の長さに合わせてリサイクルされます（第 9.2 章を参照）。

```
1 num_vec + 2
```

```
## [1] 5 4 7 6 9
```

賢明な R ユーザーなら上のコードが正解でしょう。しかし、これからの練習のために `+` を使わずに、`for()` 文を使用してみましょう（第 10.3 章を参照）。

```
1 for (i in num_vec) {  
2   print(i + 2)  
3 }
```

```
## [1] 5  
## [1] 4  
## [1] 7  
## [1] 6  
## [1] 9
```

実はこれと同じ役割をする関数が R には内蔵されており、それが `lapply()` 関数です。

```
1 lapply(オブジェクト名, 関数名, 関数の引数)
```

以下のコードからも確認出来ますが、足し算を意味する `+` も関数です。ただし、演算子を関数として使う場合は演算子を `~` で囲む必要があり、`+` だと `~+~` と表記します。

```
1 `+`(num_vec, 2)
```

```
## [1] 5 4 7 6 9
```

したがって、`lapply()` 関数を使用して `num_vec` の全要素に 2 を足す場合、以下のようないいコードとなります。

```
1 lapply(num_vec, `+`, 2)

## [[1]]
## [1] 5
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 7
##
## [[4]]
## [1] 6
##
## [[5]]
## [1] 9
```

これと同じ動きをする関数が{purrr}パッケージの `map()` です。{purrr}は{tidyverse}を読み込むと自動的に読み込まれます。

```
1 map(num_vec, `+`, 2)

## [[1]]
## [1] 5
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 7
##
## [[4]]
## [1] 6
```

```
##  
## [[5]]  
## [1] 9
```

ただし、`lapply()` と `map()` の場合、戻り値はリスト型となります。もし、ベクトル型の戻り値が必要な場合は更に `unlist()` 関数を使うか、`sapply()` を使います。

```
1 # unlist() を利用し、リストを解除する  
2 lapply(num_vec, `+`, 2) %>% unlist()  
  
## [1] 5 4 7 6 9  
  
1 # sapply() を利用すると戻り値はベクトルとなる  
2 sapply(num_vec, `+`, 2)  
  
## [1] 5 4 7 6 9
```

`map()` 関数なら `unlist()` でも良いですが、`sapply()` と同じ動きをする `map_dbl()` があります。これは戻り値が `double` の `numeric` ベクトルになる `map()` 関数です。

```
1 # map_dbl() を利用すると numeric (double) のベクトルが返される  
2 map_dbl(num_vec, `+`, 2)  
  
## [1] 5 4 7 6 9
```

もし、2を足すだけでなく、更に3で割るためにはどうすれば良いでしょうか。まず考えられるのは更に `sapply()` や `map_dbl` を使うことです。

```
1 map_dbl(num_vec, `+`, 2) %>%  
2   map_dbl(`/`, 3)  
  
## [1] 1.666667 1.333333 2.333333 2.000000 3.000000
```

もう一つの方法は `sapply()` や `map_dbl()` の第二引数に直接関数を指定する方法です。

```
1 sapply(num_vec, function(x){(x + 2) / 3})  
  
## [1] 1.666667 1.333333 2.333333 2.000000 3.000000
```

```

1 map_dbl(num_vec, function(x){(x + 2) / 3})

## [1] 1.666667 1.333333 2.333333 2.000000 3.000000

```

上のコードだと、`num_vec` の要素が第二引数で指定した関数の引数（`x`）として用いられます。関数が長くなる場合は、`sapply()` や `map()` の外側に関数を予め指定して置くことも可能です。

```

1 add_two_divide_three <- function(x){
2   (x + 2) / 3
3 }
4 sapply(num_vec, add_two_divide_three)

## [1] 1.666667 1.333333 2.333333 2.000000 3.000000

```

```

1 map_dbl(num_vec, add_two_divide_three)

## [1] 1.666667 1.333333 2.333333 2.000000 3.000000

```

ここまでの一例だと `sapply()` と `map_dbl()` はほぼ同じ関数です。なぜわざわざ`{purrr}` パッケージを読み込んでまで `map_dbl()` 関数を使う必要があるでしょうか。それは `map_dbl()` の内部にはラムダ（lambda）式、あるいは無名関数（anonymous function）と呼ばれるものが使用可能だからです。ラムダ式は第 13.1 章でも説明しましたが、もう一回解説します。ラムダ式は使い捨ての関数で、`map_dbl()` 内部での処理が終わるとメモリ上から削除される関数です。使い捨てですので、関数の名前（オブジェクト名）も与えられておりません。

このラムダ式の作り方ですが、~で始まり、引数の部分には`.x` が入ります<sup>1)</sup>。したがって、`~(.x + 2) / 3` は `function(x){(x + 2) / 3}` の簡略したものとなります。ただし、後者だと無名関数の引数に該当する `x` を `y` や `j` などに書き換えて問題ありませんが、ラムダ式では必ず`.x` と表記する必要があります。

```

1 map_dbl(num_vec, ~(.x + 2) / 3)

## [1] 1.666667 1.333333 2.333333 2.000000 3.000000

```

<sup>1)</sup> 引数が 2 つなら`.x` と`.y` を使用します。もし 3 つ以上なら`..1`、`..2`、`..3`、`...`と表記します。

かなり短めなコードで「`num_vec` の全要素に 2 を足して 3 で割る」処理ができました。一方、`sapply()` や `lapply()` のような`*apply()` 関数群だとラムダ式を使うことはできません。現在のメモリ上にある関数のみしか使うことができません。

```
1 sapply(num_vec, ~(.x + 2) / 3)
```

```
## Error in match.fun(FUN): '~(.x + 2)/3' is not a function, character or symbol
```

これまでの例は**正しい**コードではありますが、**良い**コードとは言えないでしょう。なぜなら `num_vec + 2` という最適解が存在するからです。`*apply()` と `map_*` はより複雑な処理に特化しています。たとえば、リスト型データの処理です。以下の例を考えてみましょう。

```
1 num_list <- list(List1 = c(1, 3, 5, 7, 9),
2                   List2 = c(2, 4, 6, 8, 10, 13, 3),
3                   List3 = c(3, 2, NA, 5, 8, 9, 1))
```

`num_list` は 3 つの numeric 型ベクトルで構成されたリスト型オブジェクトです。それぞれのベクトルの平均値を求めてみましょう。これを普通に `mean()` 関数のみで済まそうとすると、リストからベクトルを一つずつ抽出し、3 回の `mean()` 関数を使用する必要があります、

```
1 mean(num_list[["List1"]], na.rm = TRUE)
```

```
## [1] 5
```

```
1 mean(num_list[["List2"]], na.rm = TRUE)
```

```
## [1] 6.571429
```

```
1 mean(num_list[["List3"]], na.rm = TRUE)
```

```
## [1] 4.666667
```

もしリストの長さが大きくなると、以下のように `for()` 文の方が効率的でしょう。

```
1 for (i in names(num_list)) {
2   print(mean(num_list[[i]], na.rm = TRUE))
```

```
3  }

## [1] 5
## [1] 6.571429
## [1] 4.666667
```

計算結果をベクトルとして出力/保存する場合は予めベクトルを用意しておく必要があります。

```
1 # num_list の長さと同じ長さの空ベクトルを生成
2 Return_vec <- rep(NA, length(num_list))
3
4 for (i in 1:length(num_list)) {
5   Return_vec[i] <- mean(num_list[[i]], na.rm = TRUE)
6 }
7
8 Return_vec
```

```
## [1] 5.000000 6.571429 4.666667
```

以上の例は sapply()、または map\_dbl 関数を使うとより短くすることができます。

```
1 sapply(num_list, mean, na.rm = TRUE)
```

```
##    List1    List2    List3
## 5.000000 6.571429 4.666667
```

```
1 map_dbl(num_list, mean, na.rm = TRUE)
```

```
##    List1    List2    List3
## 5.000000 6.571429 4.666667
```

\*apply() も map\_\*() も、それぞれの要素に対して同じ処理を行うことを得意とする関数です。他の応用としては、各ベクトルから n 番目の要素を抽出することもできます。ベクトルから n 番目の要素を抽出するにはベクトル名 [n] と入力しますが、実はこの [ も関数です。関数として使う場合は + と同様、`[` と表記します。num\_list 内の 3 つのベ

クトルから 3 番目の要素を抽出してみましょう<sup>2)</sup>。

```
1 map_dbl(num_list, `[`, 3)
```

```
## List1 List2 List3
##      5      6     NA
```

ここまで来たら `map_*`() 関数群の仕組みについてイメージが出来たかと思います。`map_*`() 関数群の動きは図 27.1 のように表すことができます。第一引数はデータであり、そのデータの各要素に対して第二引数で指定された関数を適用します。この関数に必要な（データを除く）引数は第三引数以降に指定します。この関数部（第二引数）は R やパッケージなどで予め提供されている関数でも、内部に直接無名関数を作成することもできます。この無名関数は `function(x){}` のような従来の書き方も可能ですが、`map_*`() 関数群の場合~で始まるラムダ式を使うことも可能です。

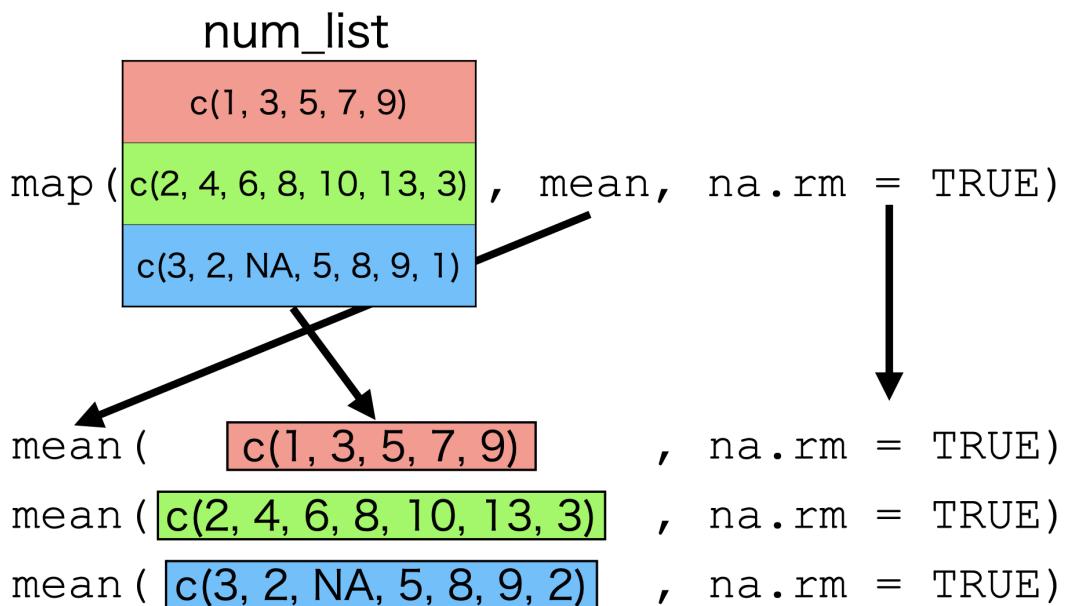


図 27.1: `map_*`() 関数群のイメージ

`map()` の場合、返り値はリストとなり、`map_dbl()` の返り値は numeric (double) 型のベ

<sup>2)</sup> 実は今回の例のように要素を抽出する場合、`[` すら要りません。`map_dbl(num_list, 3)` だけで十分です。

クトルとなります。他にも `map_*` 関数群には `map_int()`、`map_lgl()`、`map_chr()`、`map_df()` などがあり、それぞれ返り値のデータ型/データ構造を表しています。例えば、返り値が `character` 型のベクトルであれば、`map_chr()` を使います。`c(1, 2, 3, 4, 5)` のベクトルの各要素の前に "ID: " を付ける例だと以下のように書きます。

```
1 # 以下のコードでも OK
2 # map_chr(c(1, 2, 3, 4, 5), ~paste0("ID: ", .x))
3 c(1, 2, 3, 4, 5) %>%
4   map_chr(~paste0("ID: ", .x))

## [1] "ID: 1" "ID: 2" "ID: 3" "ID: 4" "ID: 5"
```

## 27.3 引数が 2 つ以上の場合

{purrr} を使った本格的な例を紹介する前に、引数が 2 つ以上の場合を考えたいと思います。まずは引数が 2 つの場合です。この場合、`map2_*` 関数群を使用します。例えば、`num_vec` に長さ 5 の numeric 型ベクトル `num_vec2` をかける例を考えてみましょう。

この場合、データとして `num_vec` と `num_vec2` が必要となり、ラムダ式にも 2 つの引数が必要です。まず、`num_vec2` を用意します。

```
1 num_vec2 <- c(1, 0, 1, 0, 1)
```

続いて `map2_dbl()` 関数を使用し `num_vec` と `num_vec2` の掛け算を行います。`map2_*` の使い方は `map_*` とほぼ同様です。

```
1 map2_dbl(データ 1, データ 2, 関数 or ラムダ式, 追加の引数)
```

`map_*` との違いとしては、(1) データが 2 つである、(2) 関数、またはラムダ式に 2 つの引数が必要である点です。この 2 点目の引数ですが、データ 2 は `.y` と表記します。したがって、データ 1 とデータ 2 の掛け算を意味するラムダ式は `~.x * .y` です。

```
1 map2_dbl(num_vec, num_vec2, ~.x * .y)
```

```
## [1] 3 0 5 0 7
```

`num_vec` と `num_vec2` が必ずしも同じデータ構造、データ型、同じ長さである必要がありません。数字の前に"ID:"を付ける先ほどの例を `map2_chr()` で書いてみましょう。

```
1 map2_chr(num_vec, "ID:", ~paste0(.y, .x))
```

```
## [1] "ID:3" "ID:2" "ID:5" "ID:4" "ID:7"
```

それでは3つ以上の引数について考えてみましょう。たとえば、`num_vec` の前に"ID"を付けるとします。そして"ID"と `num_vec` の間に" :"を入れたい場合はどうすればいいでしょう。もちろん、賢い解決方法は単純に `paste()` 関数を使うだけです。

```
1 paste("ID", num_vec, sep = ":")
```

```
## [1] "ID:3" "ID:2" "ID:5" "ID:4" "ID:7"
```

この場合、引数は3つです<sup>3)</sup>。この場合の書き方はどうなるでしょうか。`map2_*`() はデータを2つまでしか指定できません。3つ目以降は関数/ラムダ式の後ろに書くことになります。ただし、関数/ラムダ式の後ろに指定される引数は長さ1のベクトルでなければなりません。また、ラムダ式内の引数は`.x`と`.y`でなく、`..1`、`..2`、`..3`、`...`となります。

今回の例だと `map2_chr()` 内に `num_vec` がデータ1、"ID"がデータ2です。そして、期待される結果は、「"ID" + `sep` の実引数 + `num_vec` の値」となります。したがって、ラムダ式は `paste(..2, ..1, sep = ..3)` となります。

```
1 map2_chr(num_vec, "ID", ~paste(..2, ..1, sep = ..3), "-")
```

```
## [1] "ID-3" "ID-2" "ID-5" "ID-4" "ID-7"
```

データ2である"ID"は長さ1のcharacter型ベクトルであるため、以下のように `map_chr()` を使うことも可能です。

---

<sup>3)</sup> `paste()` 関数は `sep =` 以外はすべて結合の対象となります。引数が4つ以上になることもあります。たとえば、`paste("We", "love", "cats!", sep = " ")` です。

```

1 # データ 2 も長さ 1 なので map_chr() も OK
2 map_chr(num_vec, ~paste(..2, ..1, sep = ..3), "ID", "-")

```

```
## [1] "ID-3" "ID-2" "ID-5" "ID-4" "ID-7"
```

それではデータを 3 つ以上使うにはどうすれば良いでしょうか。そこで登場するのが `pmap_*` 関数です。以下の 3 つのベクトルを利用し、「名前: 数学成績」を出力してみましょう。ただし、不正行為がある場合 (`cheat_vec` の値が 1) は成績が 0 点になるようになります。

```

1 name_vec <- c("Hadley", "Song", "Yanai")
2 math_vec <- c(70, 55, 80)
3 cheat_vec <- c(0, 1, 0)

```

賢い解決法は普通に `paste0` 関数を使う方法です。

```

1 paste0(name_vec, ":", math_vec * (1 - cheat_vec))

```

```
## [1] "Hadley:70" "Song:0"      "Yanai:80"
```

今回があえて `pmap_*` 関数を使ってみましょう。`pmap_*` の場合、第一引数であるデータはリスト型で渡す必要があります。したがって、3 つのベクトルを `list` 関数を用いてリスト化します。第二引数には既存の関数やラムダ式を入力し、各引数は `..1`、`..2`、`..3` といった形で表記します。

```

1 pmap_chr(list(name_vec, math_vec, cheat_vec),
2           ~paste0(..1, ":", ..2 * (1 - ..3)))

```

```
## [1] "Hadley:70" "Song:0"      "Yanai:80"
```

第一引数はデータであるため、まずリストを作成し、パイプ演算子 (`%>%`) で `pmap_*` に渡すことも可能です。

```

1 list(name_vec, math_vec, cheat_vec) %>%
2   pmap_chr(~paste0(..1, ":", ..2 * (1 - ..3)))

```

```
## [1] "Hadley:70" "Song:0"      "Yanai:80"
```

## 27.4 データフレームと{purrr}

`map_*`() 関数のデータとしてデータフレームを渡すことも可能です。ここでは 5 行 3 列のデータフレームを作成してみましょう。

```
1 Dummy_df <- data.frame(X = seq(1, 5, by = 1),
2                           Y = seq(10, 50, by = 10),
3                           Z = seq(2, 10, by = 2))
```

各行の X、Y、Z の合計を計算するには{dplyr} の `rowwise()` と `mutate()` を組み合わせることで計算出来ることを第 13.4 章で紹介しました。

```
1 Dummy_df %>%
2   rowwise() %>%
3   mutate(Sum = sum(X, Y, Z)) %>%
4   # rowwise() は 1 行を 1 グループとする関数であるため、最後にグループ化を解除
5   ungroup()
```

```
## # A tibble: 5 x 4
##       X     Y     Z   Sum
##   <dbl> <dbl> <dbl> <dbl>
## 1     1    10     2    13
## 2     2    20     4    26
## 3     3    30     6    39
## 4     4    40     8    52
## 5     5    50    10    65
```

それでは各列の平均値を計算するにはどうすれば良いでしょうか。ここでは R 内蔵関数である `colMeans()` を使わないことにしましょう。

```
1 colMeans(Dummy_df)
```

```
##  X  Y  Z
```

```
## 3 30 6
```

まず、`mean(Dummy_df$X)` を 3 回実行する方法や、`{dplyr}` の `summarise()` 関数を使う方法があります。

```
1 Dummy_df %>%
2   summarise(X = mean(X),
3             Y = mean(Y),
4             Z = mean(Z))
```

```
## X Y Z
## 1 3 30 6
```

実はこの操作、`map_dbl()` 関数を使えば、より簡単です。

```
1 map_dbl(Dummy_df, mean)

## X Y Z
## 3 30 6
```

`map_dbl()` は numeric (double) 型ベクトルを返しますが、データフレーム（具体的には tibble）に返すなら `map_df()` を使います。

```
1 map_df(Dummy_df, mean)

## # A tibble: 1 x 3
##       X     Y     Z
##   <dbl> <dbl> <dbl>
## 1     3    30     6
```

なぜこれが出来るでしょうか。これを理解するためにはデータフレームと tibble が本質的にはリスト型と同じであることを理解する必要があります。たとえば、以下のようなリストについて考えてみましょう。

```
1 Dummy_list <- list(X = seq(1, 5, by = 1),
2                      Y = seq(10, 50, by = 10),
3                      Z = seq(2, 10, by = 2))
```

```
4
5 Dummy_list

## $X
## [1] 1 2 3 4 5
##
## $Y
## [1] 10 20 30 40 50
##
## $Z
## [1] 2 4 6 8 10
```

この Dummy\_list を `as.data.frame()` 関数を使用して強制的にデータフレームに変換してみましょう。

```
1 as.data.frame(Dummy_list)

##   X   Y   Z
## 1 1 10  2
## 2 2 20  4
## 3 3 30  6
## 4 4 40  8
## 5 5 50 10
```

`Dummy_df` と同じものが出てきました。逆に `Dummy_df` を、`as.list()` を使用してリストに変換すると `Dummy_list` と同じものが返されます。

```
1 as.list(Dummy_df)

## $X
## [1] 1 2 3 4 5
##
## $Y
## [1] 10 20 30 40 50
##
```

```
## $Z
## [1] 2 4 6 8 10
```

ここまで理解できれば、`map_*`() 関数のデータがデータフレームの場合、内部ではリストとして扱われることが分かるでしょう。実際、`Dummy_list` をデータとして入れても `Dummy_df` を入れた結果と同じものが得られます。

```
1 map_df(Dummy_list, mean)

## # A tibble: 1 x 3
##       X     Y     Z
##   <dbl> <dbl> <dbl>
## 1     3     30     6
```

### 27.4.1 tibble の話

ここまで「データフレーム」と「tibble」を区別せずに説明してきましたが、これからはこの 2 つを区別する必要があります。tibble は{tidyverse} のコアパッケージの一つである{tibble} が提供するデータ構造であり、データフレームの上位互換です。tibble もデータフレーム同様、本質的にはリストですが、リストの構造をより的確に表すことが出来ます。

データフレームをリストとして考える場合、リストの各要素は**必ずベクトル**である必要があります。たとえば、`Dummy_list` には 3 つの要素があり、それぞれ長さ 5 のベクトルです。一方、リストの中にはリストを入れることも出来ます。たとえば、以下のような `Dummy_list2` について考えてみましょう。

```
1 Dummy_list2 <- list(ID = 1:3,
2                         Data = list(Dummy_df, Dummy_df, Dummy_df))
3
4 Dummy_list2

## $ID
## [1] 1 2 3
##
```

```
## $Data
## $Data[[1]]
##   X  Y  Z
## 1 1 10 2
## 2 2 20 4
## 3 3 30 6
## 4 4 40 8
## 5 5 50 10
##
## $Data[[2]]
##   X  Y  Z
## 1 1 10 2
## 2 2 20 4
## 3 3 30 6
## 4 4 40 8
## 5 5 50 10
##
## $Data[[3]]
##   X  Y  Z
## 1 1 10 2
## 2 2 20 4
## 3 3 30 6
## 4 4 40 8
## 5 5 50 10
```

`Dummy_list2` には 2 つの要素があり、最初の要素は長さ 3 のベクトル、2 つ目の要素は長さ 3 のリストです。2 つ目の要素がベクトルでないため、`Dummy_list2` をデータフレームに変換することはできません。

```
1 Dummy_df2 <- as.data.frame(Dummy_list2)
```

```
## Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TR
```

一方、`as_tibble()` を使用して tibble 型に変換することは可能です。

```
1 Dummy_tibble <- as_tibble(Dummy_list2)
2
3 Dummy_tibble

## # A tibble: 3 x 2
##       ID Data
##   <int> <list>
## 1     1 <df [5 x 3]>
## 2     2 <df [5 x 3]>
## 3     3 <df [5 x 3]>
```

2列目の各セルには5行3列のデータフレーム(df)が格納されていることが分かります。たとえば、`Dummy_tibble`のData列を抽出してみましょう。

```
1 Dummy_tibble$Data
```

```
## [[1]]
##   X  Y  Z
## 1 1 10  2
## 2 2 20  4
## 3 3 30  6
## 4 4 40  8
## 5 5 50 10
##
## [[2]]
##   X  Y  Z
## 1 1 10  2
## 2 2 20  4
## 3 3 30  6
## 4 4 40  8
## 5 5 50 10
##
## [[3]]
```

```
##  X  Y  Z
## 1 10  2
## 2 20  4
## 3 30  6
## 4 40  8
## 5 50 10
```

長さ 3 のリスト出力されます。続いて、`Dummy_tibble$Data` の 2 番目のセルを抽出してみましょう。

```
1 Dummy_tibble$Data[2]
```

```
## [[1]]
##  X  Y  Z
## 1 10  2
## 2 20  4
## 3 30  6
## 4 40  8
## 5 50 10
```

データフレームが output されました。簡単にまとめると tibble はデータフレームの中にデータフレームを入れることが出来るデータ構造です。もちろん、これまでの通り、データフレームのように使うことも可能です<sup>4)</sup>。これが tibble の強みでもあり、`{purrr}`との相性も非常に高いです。たとえば、`Data` 列を `map()` 関数のデータとして渡せば、複数のデータセットに対して同じモデルの推定が出来るようになります。以下ではその例を紹介します。

---

<sup>4)</sup> ただし tibble は tibble 名 `[3, 2:4] <- c("A", "B", "C")` のような、列をまたがった値の代入は出来ません。これによって tibble に対応しない関数もまだあります。この場合、`as.data.frame()` を使って一旦データフレームに変換する必要があります。

## 27.5 モデルの反復推定

ここからは `map_*` 関数群を使って複数のモデルを素早く推定する方法について解説します。サンプルデータは第 18 章で使いました各国の政治経済データ (`Countries.csv`) を使用します。csv 形式データ読み込みの際、これまで R 内蔵関数である `read.csv()` と `{readr}^5)` の `read_csv()` を区別せずに使用してきました。しかし、ここからは `read_csv()` を使用します。2 つの使い方はほぼ同じですが、`read_csv()` は各列のデータ型を自動的に指定してくれるだけではなく、`tibble` 構造として読み込みます。`read.csv()` を使用する場合、`as.tibble(データフレーム名)` でデータフレームを `tibble` に変換してください。

```
1 Country_df <- read_csv("Data/Countries.csv")  
  
## # Rows: 186 Columns: 18  
## -- Column specification -----  
## Delimiter: ","  
## chr (4): Country, Polity_Type, FH_Status, Continent  
## dbl (14): Population, Area, GDP, PPP, GDP_per_capita, PPP_per_capita, G7, G2...  
##  
## i Use `spec()` to retrieve the full column specification for this data.  
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

### 27.5.1 サンプルの分割とモデル推定 (`split()` 利用)

まずは、サンプルを分割し、それぞれの下位サンプルを使った分析を繰り返す方法について解説します。サンプルを分割する方法は 2 つありますが、最初は R 内蔵関数である `split()` 関数を使った方法について紹介します。

```
1 # split() の使い方  
2 新しいオブジェクト名 <- split(データ名, 分割の基準となるベクトル)
```

---

<sup>5)</sup> `{tidyverse}` のコア・パッケージであるため、別途読み込む必要はございません。

たとえば、Country\_df を大陸ごとにサンプルを分けるとします。大陸を表すベクトルは Country\_df\$Continent です。したがって、以下のように入力します。

```
1 Split_Data <- split(Country_df, Country_df$Continent)
```

サンプルが分割された Split\_Data のデータ構造はリスト型です。

```
1 class(Split_Data)
```

```
## [1] "list"
```

中身を見るとリストの各要素として tibble (データフレーム) が格納されています。リストの中にはあらゆるデータ型、データ構造を入れることができます。

```
1 Split_Data
```

```
## $Africa
## # A tibble: 54 x 18
##   Country   Population   Area     GDP     PPP GDP_per_capita PPP_per_capita G
##   <chr>       <dbl>     <dbl>    <dbl>    <dbl>        <dbl>        <dbl> <dbl>
## 1 Algeria  43851044 2.38e6 1.70e5 4.97e5      3876.      11324.
## 2 Angola   32866272 1.25e6 9.46e4 2.19e5      2879.      6649.
## 3 Benin    12123200 1.13e5 1.44e4 3.72e4      1187.      3067.
## 4 Botswana 2351627  5.67e5 1.83e4 4.07e4      7799.      17311.
## 5 Burkina ~ 20903273 2.74e5 1.57e4 3.76e4      753.       1800.
## 6 Burundi  11890784 2.57e4 3.01e3 8.72e3      253.       733.
## 7 Cabo Ver~ 555987  4.03e3 1.98e3 3.84e3      3565.      6913.
## 8 Cameroon 26545863 4.73e5 3.88e4 9.31e4      1460.      3506.
## 9 Central ~ 4829767  6.23e5 2.22e3 4.46e3      460.       924.
## 10 Chad    16425864 1.26e6 1.13e4 2.51e4      689.      1525.
## # ... with 44 more rows, and 10 more variables: G20 <dbl>, OECD <dbl>,
## #   HDI_2018 <dbl>, Polity_Score <dbl>, Polity_Type <chr>, FH_PR <dbl>,
## #   FH_CL <dbl>, FH_Total <dbl>, FH_Status <chr>, Continent <chr>
## #
## $America
```

```
## # A tibble: 36 x 18
##   Country   Population   Area     GDP     PPP GDP_per_capita PPP_per_capita   G7
##   <chr>        <dbl>    <dbl>    <dbl>    <dbl>        <dbl>        <dbl> <dbl>
## 1 Antigua ~    97929 4.4e2 1.73e3 2.08e3    17643.    21267.    0
## 2 Argentina  45195774 2.74e6 4.50e5 1.04e6    9949.    22938.    0
## 3 Bahamas    393244 1.00e4 1.28e4 1.40e4   32618.    35662.    0
## 4 Barbados   287375 4.3e2 5.21e3 4.62e3   18126.    16066.    0
## 5 Belize      397628 2.28e4 1.88e3 2.82e3   4727.    7091.    0
## 6 Bolivia    11673021 1.08e6 4.09e4 1.01e5   3503.    8623.    0
## 7 Brazil      212559417 8.36e6 1.84e6 3.13e6   8655.    14734.    0
## 8 Canada     37742154 9.09e6 1.74e6 1.85e6   46008.    49088.    1
## 9 Chile       19116201 7.44e5 2.82e5 4.64e5   14769.    24262.    0
## 10 Colombia  50882891 1.11e6 3.24e5 7.37e5   6364.    14475.    0
## # ... with 26 more rows, and 10 more variables: G20 <dbl>, OECD <dbl>,
## #   HDI_2018 <dbl>, Polity_Score <dbl>, Polity_Type <chr>, FH_PR <dbl>,
## #   FH_CL <dbl>, FH_Total <dbl>, FH_Status <chr>, Continent <chr>
##
## $Asia
## # A tibble: 42 x 18
##   Country   Population   Area     GDP     PPP GDP_per_capita PPP_per_capita   G7
##   <chr>        <dbl>    <dbl>    <dbl>    <dbl>        <dbl>        <dbl> <dbl>
## 1 Afghanis~  38928346 6.53e5 1.91e4 8.27e4    491.    2125.    0
## 2 Bahrain     1701575 7.6e2 3.86e4 7.42e4   22670.    43624.    0
## 3 Banglade~  164689383 1.30e5 3.03e5 7.34e5   1837.    4458.    0
## 4 Bhutan      771608 3.81e4 2.45e3 8.77e3   3171.    11363.    0
## 5 Brunei      437479 5.27e3 1.35e4 2.65e4   30789.    60656.    0
## 6 Burma       54409800 6.53e5 7.61e4 2.68e5   1398.    4932.    0
## 7 Cambodia   16718965 1.77e5 2.71e4 6.93e4   1620.    4142.    0
## 8 China      1447470092 9.39e6 1.48e7 2.20e7   10199.   15177.    0
## 9 India       1380004385 2.97e6 2.88e6 9.06e6   2083.    6564.    0
## 10 Indonesia 273523615 1.81e6 1.12e6 3.12e6   4092.    11397.    0
## # ... with 32 more rows, and 10 more variables: G20 <dbl>, OECD <dbl>,
```

```
## #  HDI_2018 <dbl>, Polity_Score <dbl>, Polity_Type <chr>, FH_PR <dbl>,
## #  FH_CL <dbl>, FH_Total <dbl>, FH_Status <chr>, Continent <chr>
##
## $Europe
## # A tibble: 50 x 18
##   Country  Population   Area    GDP    PPP GDP_per_capita PPP_per_capita G
##   <chr>        <dbl>  <dbl>  <dbl>  <dbl>        <dbl>        <dbl> <dbl>
## 1 Albania     2877797 27400 1.53e4 39658.        5309.      13781.
## 2 Andorra      77265   470 3.15e3    NA        40821.        NA
## 3 Armenia      2963243 28470 1.37e4 38446.        4614.      12974.
## 4 Austria      9006398 82409 4.46e5 502771.       49555.      55824.
## 5 Azerbai~    10139177 82658 4.80e4 144556.       4739.      14257.
## 6 Belarus      9449323 202910 6.31e4 183461.       6676.      19415.
## 7 Belgium      11589623 30280 5.30e5 597433.       45697.      51549.
## 8 Bosnia ~    3280819  51000 2.00e4 49733.        6111.      15159.
## 9 Bulgaria     6948445 108560 6.79e4 156693.       9776.      22551.
## 10 Croatia     4105267  55960 6.04e4 114932.       14717.      27996.
## # ... with 40 more rows, and 10 more variables: G20 <dbl>, OECD <dbl>,
## #  HDI_2018 <dbl>, Polity_Score <dbl>, Polity_Type <chr>, FH_PR <dbl>,
## #  FH_CL <dbl>, FH_Total <dbl>, FH_Status <chr>, Continent <chr>
##
## $Oceania
## # A tibble: 4 x 18
##   Country  Population   Area    GDP    PPP GDP_per_capita PPP_per_capita G
##   <chr>        <dbl>  <dbl>  <dbl>  <dbl>        <dbl>        <dbl> <dbl>
## 1 Australia    25499884 7.68e6 1.39e6 1.28e6       54615.      50001.
## 2 Fiji         896445  1.83e4 5.54e3 1.25e4       6175.      13940.
## 3 New Zeala~   4842780 2.64e5 2.07e5 2.04e5       42729.      42178.
## 4 Papua New~  8947024 4.53e5 2.50e4 3.73e4       2791.      4171.
## # ... with 10 more variables: G20 <dbl>, OECD <dbl>, HDI_2018 <dbl>,
## #  Polity_Score <dbl>, Polity_Type <chr>, FH_PR <dbl>, FH_CL <dbl>,
## #  FH_Total <dbl>, FH_Status <chr>, Continent <chr>
```

それでは分割された各サンプルに対して Polity\_Score と FH\_Total の相関係数を計算してみましょう。その前に相関分析の方法について調べてみましょう。R には相関分析の関数が 2 つ用意されています。単純に相関係数のみを計算するなら `cor()`、係数の不確実性（標準誤差、信頼区間など）まで計算し、検定を行うなら `cor.test()` を使用します。ここではより汎用性の高い `cor.test()` の使い方について紹介します。

```
1 # 相関分析: 方法 1
2 cor.test(~ 変数名 1 + 変数名 2, data = データ名)
3
4 # 相関分析: 方法 2
5 cor.test(データ名$変数名 1, データ名$変数名 2)
```

それでは全サンプルに対して相関分析をしてみましょう。

```
1 Cor_fit1 <- cor.test(~ Polity_Score + FH_Total, data = Country_df)
2 Cor_fit1
```

```
##
##  Pearson's product-moment correlation
##
## data:  Polity_Score and FH_Total
## t = 19.494, df = 156, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.7896829 0.8821647
## sample estimates:
##      cor
## 0.8420031
```

ここから相関係数（0.8420031）のみを抽出するにはどうすれば良いでしょうか。それを確認するためには `Cor_fit1` というオブジェクトの構造を調べる必要があります。ここでは R 内蔵関数である `str()` を使って確認してみましょう。

```
1 str(Cor_fit1)

## List of 9
## $ statistic : Named num 19.5
## ..- attr(*, "names")= chr "t"
## $ parameter : Named int 156
## ..- attr(*, "names")= chr "df"
## $ p.value   : num 1.16e-43
## $ estimate  : Named num 0.842
## ..- attr(*, "names")= chr "cor"
## $ null.value: Named num 0
## ..- attr(*, "names")= chr "correlation"
## $ alternative: chr "two.sided"
## $ method    : chr "Pearson's product-moment correlation"
## $ data.name  : chr "Polity_Score and FH_Total"
## $ conf.int   : num [1:2] 0.79 0.882
## ..- attr(*, "conf.level")= num 0.95
## - attr(*, "class")= chr "htest"
```

相関係数は\$estimateで抽出できそうですね。実際にCor\_fit1から相関係数のみ抽出してみましょう。

```
1 Cor_fit1$estimate

##       cor
## 0.8420031
```

それではmap()関数を利用して分割された各サンプルを対象にPolity\_ScoreとFH\_Totalの相関係数を計算してみましょう。map()のデータはSplit\_Dataとし、関数はラムダ式を書いてみましょう。cor.test()内data引数の実引数は.x.または..1となります。最後にmap\_dbl("estimate")を利用し、相関係数を抽出、numeric(double)型ベクトルとして出力します。

```
1 Cor_fit2 <- Split_Data %>%
2   map(~cor.test(~ Polity_Score + FH_Total, data = .x))
```

```
1 # Cor_fit2 から相関係数のみ出力
2 map_dbl(Cor_fit2, "estimate")
```

```
##    Africa    America      Asia    Europe  Oceania
## 0.7612138 0.8356899 0.8338172 0.8419547 0.9960776
```

もし、小数点 3 位までの  $p$  値が欲しい場合は以下のように入力します。

```
1 # Cor_fit2 から相関係数の p 値を抽出し、小数点 3 位に丸める
2 map_dbl(Cor_fit2, "p.value") %>% round(3)
```

```
##    Africa America      Asia    Europe  Oceania
## 0.000 0.000 0.000 0.000 0.004
```

## 27.5.2 サンプルの分割とモデル推定 (nest() 利用)

それではデータフレーム内にデータフレームが格納可能な tibble 構造の長所を活かした方法について解説します。ある変数に応じてデータをグループ化する方法については第 13.2 章で解説しました。{tidyverse} パッケージにはグループごとにデータを分割し、分割されたデータを各セルに埋め込む nest() 関数を提供しています。具体的な動きを見るために、とりあえず、Country\_df を大陸 (Continent) ごとに分割し、それぞれがデータが一つのセルに埋め込まれた新しいデータセット、Nested\_Data を作ってみましょう。

```
1 Nested_Data <- Country_df %>%
2   group_by(Continent) %>%
3   nest()
4
5 Nested_Data
```

```
## # A tibble: 5 x 2
## # Groups:   Continent [5]
##   Continent data
```

```
## <chr> <list>
## 1 Asia <tibble [42 x 17]>
## 2 Europe <tibble [50 x 17]>
## 3 Africa <tibble [54 x 17]>
## 4 America <tibble [36 x 17]>
## 5 Oceania <tibble [4 x 17]>
```

ちなみに `group_by()` と `nest()` は `group_nest()` を使って以下のようにまとめることも可能です。

```
1 Nested_Data <- Country_df %>%
2   group_nest(Continent)
```

2列目の `data` 変数の各セルに `tibble` が埋め込まれたことがわかります。`Nested_Data` の `data` 列から 5つ目の要素（オセアニアのデータ）を確認してみましょう。

```
1 # Nested_Data$data[5] の場合、長さ 1 のリストが outputされる
2 # tibble 構造で outputする場合は [5] でなく、[[5]] で抽出
3 Nested_Data$data[[5]]
```

```
## # A tibble: 4 x 17
##   Country  Population   Area    GDP    PPP GDP_per_capita PPP_per_capita G
##   <chr>        <dbl>  <dbl>  <dbl>  <dbl>        <dbl>        <dbl> <dbl>
## 1 Australia  25499884 7.68e6 1.39e6 1.28e6      54615.      50001.
## 2 Fiji       896445  1.83e4 5.54e3 1.25e4      6175.      13940.
## 3 New Zeala~ 4842780 2.64e5 2.07e5 2.04e5     42729.      42178.
## 4 Papua New~ 8947024 4.53e5 2.50e4 3.73e4      2791.      4171.
## # ... with 9 more variables: G20 <dbl>, OECD <dbl>, HDI_2018 <dbl>,
## #   Polity_Score <dbl>, Polity_Type <chr>, FH_PR <dbl>, FH_CL <dbl>,
## #   FH_Total <dbl>, FH_Status <chr>
```

`Continent` が `Oceania` の行の `data` 列にオセアニアのみのデータが格納されていることが分かります。このようなデータ構造を入れ子型データ（nested data）と呼びます。この入れ子型データは `unnest()` 関数を使って解除することも可能です。`unnest()` 関数を使う際はどの列を解除するかを指定する必要があります、今回は `data` 列となります。

```
1 # 入れ子型データの解除
2 Nested_Data %>%
3   unnest(data)

## # A tibble: 186 x 18
## # Groups:   Continent [5]
##   Continent Country     Population     Area     GDP     PPP GDP_per_capita
##   <chr>     <chr>       <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 Asia      Afghanistan 38928346 652860 19101. 82737. 491.
## 2 Asia      Bahrain     1701575  760 38574. 74230. 22670.
## 3 Asia      Bangladesh  164689383 130170 302571. 734108. 1837.
## 4 Asia      Bhutan      771608  38117 2447. 8767. 3171.
## 5 Asia      Brunei     437479  5270 13469. 26536. 30789.
## 6 Asia      Burma       54409800 653290 76086. 268327. 1398.
## 7 Asia      Cambodia   16718965 176520 27089. 69253. 1620.
## 8 Asia      China       1447470092 9389291 14762792. 21967628. 10199.
## 9 Asia      India       1380004385 2973190 2875142. 9058692. 2083.
## 10 Asia     Indonesia  273523615 1811570 1119191. 3117334. 4092.
## # ... with 176 more rows, and 11 more variables: PPP_per_capita <dbl>,
## #   G7 <dbl>, G20 <dbl>, OECD <dbl>, HDI_2018 <dbl>, Polity_Score <dbl>,
## #   Polity_Type <chr>, FH_PR <dbl>, FH_CL <dbl>, FH_Total <dbl>,
## #   FH_Status <chr>
```

{dplyr}のgroup\_by()関数と{tidyverse}のnest()、unnest()関数を組み合わせることでデータを入れ子型データへ変換したり、解除することができます。以上の流れを図式化したもののが以下の図 27.2 です。

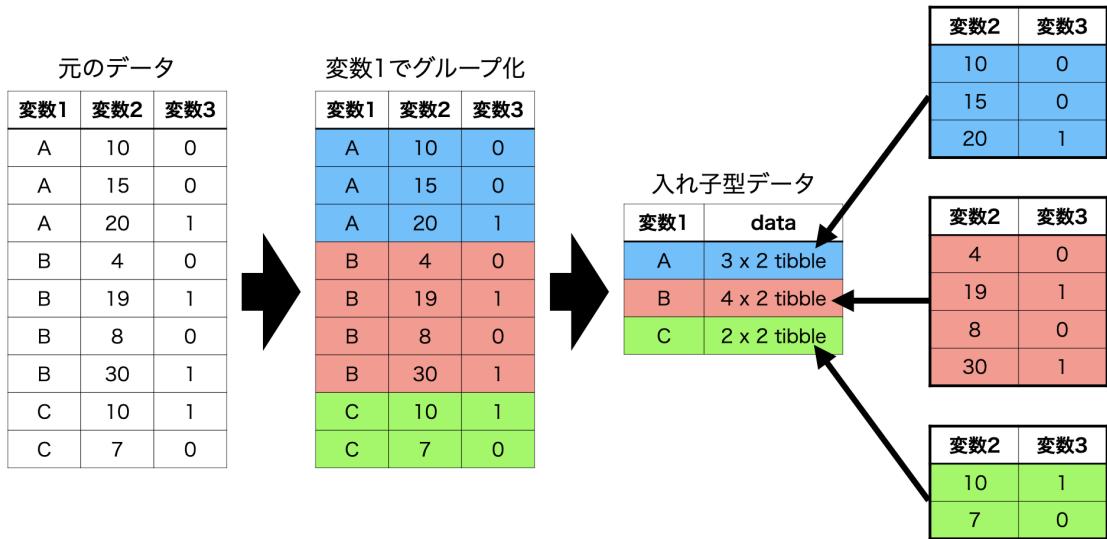


図 27.2: 入れ子型データの生成過程

それではこの入れ子型データを使用して大陸ごとの Polity\_Score と FH\_Total の相関係数を計算してみましょう。data 列をデータとした相関係数のラムダ式はどう書けば良いでしょうか。これまでの内容が理解できましたら、答えは難しくないでしょう。cor.test() 関数の data 引数の実引数として.x を指定するだけです。この結果を Cor\_test という列として追加し、Nested\_Data2 という名のオブジェクトとして保存します。

```

1 Nested_Data2 <- Nested_Data %>%
2   mutate(Cor_test = map(data, ~cor.test(~ Polity_Score + FH_Total, data = .x)))
3
4 Nested_Data2

```

```

## # A tibble: 5 x 3
## # Groups:   Continent [5]
##   Continent data          Cor_test
##   <chr>     <list>        <list>
## 1 Asia      <tibble [42 x 17]> <htest>
## 2 Europe    <tibble [50 x 17]> <htest>
## 3 Africa    <tibble [54 x 17]> <htest>

```

```
## 4 America  <tibble [36 x 17]> <htest>
## 5 Oceania  <tibble [4 x 17]>  <htest>
```

Cor\_test という列が追加され、htest というクラス（S3 クラス）オブジェクトが格納されていることが分かります。ちゃんと相関分析のオブジェクトが格納されているか、確認してみましょう。

```
1 Nested_Data2$Cor_test
```

```
## [[1]]
##
## Pearson's product-moment correlation
##
## data: Polity_Score and FH_Total
## t = 9.0626, df = 36, p-value = 8.05e-11
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.7009872 0.9107368
## sample estimates:
##       cor
## 0.8338172
##
## [[2]]
##
## Pearson's product-moment correlation
##
## data: Polity_Score and FH_Total
## t = 9.7452, df = 39, p-value = 5.288e-12
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.7210854 0.9130896
## sample estimates:
##       cor
```

```
## 0.8419547
##
##
## [[3]]
##
## Pearson's product-moment correlation
##
## data: Polity_Score and FH_Total
## t = 7.9611, df = 46, p-value = 3.375e-10
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.6087424 0.8594586
## sample estimates:
##
## cor
## 0.7612138
##
##
## [[4]]
##
## Pearson's product-moment correlation
##
## data: Polity_Score and FH_Total
## t = 7.6082, df = 25, p-value = 5.797e-08
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.6677292 0.9226837
## sample estimates:
##
## cor
## 0.8356899
##
##
## [[5]]
```

```
##  
## Pearson's product-moment correlation  
##  
## data: Polity_Score and FH_Total  
## t = 15.92, df = 2, p-value = 0.003922  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.8197821 0.9999220  
## sample estimates:  
## cor  
## 0.9960776
```

5つの相関分析結果が格納されています。続いて、ここから相関係数のみを抽出してみましょう。相関分析オブジェクトから相関係数を抽出するにはオブジェクト名\$estimateだけで十分です。`mpa_*`() 関数を使うなら第2引数として関数やラムダ式を指定せず、"estimate"のみ入力するだけです。

```
1 Nested_Data2 <- Nested_Data2 %>%  
2   mutate(Cor_coef = map(Cor_test, "estimate"))  
3  
4 Nested_Data2
```

```
## # A tibble: 5 x 4  
## # Groups: Continent [5]  
##   Continent data          Cor_test Cor_coef  
##   <chr>     <list>        <list>   <list>  
## 1 Asia      <tibble [42 x 17]> <htest> <dbl [1]>  
## 2 Europe    <tibble [50 x 17]> <htest> <dbl [1]>  
## 3 Africa    <tibble [54 x 17]> <htest> <dbl [1]>  
## 4 America   <tibble [36 x 17]> <htest> <dbl [1]>  
## 5 Oceania   <tibble [4 x 17]> <htest> <dbl [1]>
```

`Cor_coef` 列が追加され、それぞれのセルに numeric 型の値が格納されていることが分かります。`map()` 関数はリスト型でデータを返すため、このように出力されます。この

Cor\_coef列の入れ子構造を解除してみましょう。

```
1 Nested_Data2 %>%
2   unnest(cols = Cor_coef)

## # A tibble: 5 x 4
## # Groups:   Continent [5]
##   Continent data          Cor_test Cor_coef
##   <chr>     <list>        <list>    <dbl>
## 1 Asia      <tibble [42 x 17]> <htest>   0.834
## 2 Europe    <tibble [50 x 17]> <htest>   0.842
## 3 Africa    <tibble [54 x 17]> <htest>   0.761
## 4 America   <tibble [36 x 17]> <htest>   0.836
## 5 Oceania   <tibble [4 x 17]> <htest>   0.996

1 Nested_Data2

## # A tibble: 5 x 4
## # Groups:   Continent [5]
##   Continent data          Cor_test Cor_coef
##   <chr>     <list>        <list>    <list>
## 1 Asia      <tibble [42 x 17]> <htest> <dbl [1]>
## 2 Europe    <tibble [50 x 17]> <htest> <dbl [1]>
## 3 Africa    <tibble [54 x 17]> <htest> <dbl [1]>
## 4 America   <tibble [36 x 17]> <htest> <dbl [1]>
## 5 Oceania   <tibble [4 x 17]> <htest> <dbl [1]>
```

Cor\_coefの各列に格納された値がoutputされました。以上の作業を一つのコードとしてまとめるこども出来ます。また、相関係数の抽出の際に`map()`でなく、`map_dbl()`を使えば、numeric型ベクトルが返されるので`unnest()`も不要となります。

```
1 Country_df %>%
2   group_by(Continent) %>%
3   nest() %>%
4   mutate(Cor_test = map(data, ~cor.test(~ Polity_Score + FH_Total, data = .x)),
```

```
5 Cor_coef = map_dbl(Cor_test, "estimate"))

## # A tibble: 5 x 4
## # Groups:   Continent [5]
##   Continent data             Cor_test Cor_coef
##   <chr>     <list>          <list>    <dbl>
## 1 Asia      <tibble [42 x 17]> <htest>   0.834
## 2 Europe    <tibble [50 x 17]> <htest>   0.842
## 3 Africa    <tibble [54 x 17]> <htest>   0.761
## 4 America   <tibble [36 x 17]> <htest>   0.836
## 5 Oceania   <tibble [4 x 17]>  <htest>   0.996
```

たった5行のコードで大陸ごとの相関分析が出来ました。これを `map_*`() を使わずに処理するなら以下のようなコードとなります<sup>6)</sup>。

```
1 Cor_Test1 <- cor.test(~ Polity_Score + FH_Total,
2                           data = subset(Country_df, Country_df$Continent == "Asia"))
3 Cor_Test2 <- cor.test(~ Polity_Score + FH_Total,
4                           data = subset(Country_df, Country_df$Continent == "Europe"))
5 Cor_Test3 <- cor.test(~ Polity_Score + FH_Total,
6                           data = subset(Country_df, Country_df$Continent == "Africa"))
7 Cor_Test4 <- cor.test(~ Polity_Score + FH_Total,
8                           data = subset(Country_df, Country_df$Continent == "America"))
9 Cor_Test5 <- cor.test(~ Polity_Score + FH_Total,
10                          data = subset(Country_df, Country_df$Continent == "Oceania"))

11
12 Cor_Result <- c("Asia" = Cor_Test1$estimate, "Europe" = Cor_Test2$estimate,
13                  "Africa" = Cor_Test3$estimate, "America" = Cor_Test4$estimate,
14                  "Oceania" = Cor_Test5$estimate)
15
16 print(Cor_Result)
```

<sup>6)</sup> `cor.test()$estimate` で直接相関係数を抽出することも可能ですが、コードの可読性が著しく低下します。またオブジェクトの再利用（例えば、「今度は  $p$  値を抽出しよう！」）が出来なくなります。

```
##      Asia.cor  Europe.cor  Africa.cor America.cor Oceania.cor
## 0.8338172   0.8419547   0.7612138   0.8356899   0.9960776
```

それでは応用例として回帰分析をしてみましょう。今回も `Nested_Data` を使用し、`PPP_per_capita` を応答変数に、`FH_Total` と `Population` を説明変数とした重回帰分析を行い、政治的自由度 (`FH_Total`) の係数や標準誤差などを抽出してみましょう。まずは、ステップごとにコードを分けて説明し、最後には一つのコードとしてまとめたものをお見せします。

```
1 Nested_Data %>%
2   mutate(Model = map(data, ~lm(PPP_per_capita ~ FH_Total + Population,
3                         data = .x)))
```

```
## # A tibble: 5 x 3
## # Groups:   Continent [5]
##   Continent data             Model
##   <chr>     <list>           <list>
## 1 Asia      <tibble [42 x 17]> <lm>
## 2 Europe    <tibble [50 x 17]> <lm>
## 3 Africa    <tibble [54 x 17]> <lm>
## 4 America   <tibble [36 x 17]> <lm>
## 5 Oceania   <tibble [4 x 17]> <lm>
```

`Model` 列からから推定結果の要約を抽出するにはどうすれば良いでしょうか。1つ目の方法は `summary(lm オブジェクト名)$coefficients` で抽出する方法です。

```
1 lm_fit1 <- lm(PPP_per_capita ~ FH_Total + Population, data = Country_df)
2
3 summary(lm_fit1)$coefficients
```

```
##                               Estimate Std. Error    t value    Pr(>|t|) 
## (Intercept)  1.929308e+03 3.229792e+03  0.5973473 5.510476e-01
## FH_Total     3.256660e+02 4.871626e+01   6.6849553 2.979474e-10
## Population  -2.217793e-06 9.193256e-06 -0.2412413 8.096505e-01
```

もっと簡単な方法は{broom}の `tidy()` 関数を使う方法です。`tidy()` 関数は推定値に関

する様々な情報をデータフレームとして返す便利な関数です。デフォルトだと 95% 信頼区間は出力されませんが、`conf.int = TRUE` を指定すると信頼区間も出力されます。`tidy()` 関数を提供する`{broom}`パッケージは`{tidyverse}`の一部ですが、`{tidyverse}`読み込みの際に一緒に読み込まれるものではないため、予め`{broom}`パッケージを読み込んでおくか、`broom::tidy()`で使うことができます。

```
1 broom::tidy(lm_fit1, conf.int = TRUE)
```

```
## # A tibble: 3 x 7
##   term        estimate  std.error statistic  p.value conf.low conf.high
##   <chr>      <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) 1929.     3230.     0.597 5.51e- 1  -
## 2 FH_Total     326.      48.7      6.68  2.98e-10 2.30e+2  4.22e+2
## 3 Population   -0.00000222  0.00000919 -0.241 8.10e- 1  -
## 4             2.04e-5  1.59e-5
```

それでは `Model` 列にある `lm` オブジェクトから推定値の情報を抽出し、`Model2` という列に格納してみましょう。今回はラムダ式を使う必要がありません。なぜなら、`tidy()` の第一引数がデータだからです (`?tidy.lm` 参照)。他にも必要な引数 (`conf.int = TRUE` など) があれば、`map()` の第 3 引数以降で指定します。

```
1 Nested_Data %>%
2   mutate(Model = map(data, ~lm(PPP_per_capita ~ FH_Total + Population,
3                         data = .x)),
4         Model2 = map(Model, broom::tidy, conf.int = TRUE))
```

```
## # A tibble: 5 x 4
## # Groups:   Continent [5]
##   Continent data        Model Model2
##   <chr>     <list>      <list> <list>
## 1 Asia      <tibble [42 x 17]> <lm>   <tibble [3 x 7]>
## 2 Europe    <tibble [50 x 17]> <lm>   <tibble [3 x 7]>
## 3 Africa    <tibble [54 x 17]> <lm>   <tibble [3 x 7]>
```

```
## 4 America  <tibble [36 x 17]> <lm>  <tibble [3 x 7]>
## 5 Oceania  <tibble [4 x 17]>  <lm>  <tibble [3 x 7]>
```

Model2 列の各セルに 7 列の tibble が格納されているようです。ここからは data と Model 列は不要ですので select() 関数を使って除去し、Model2 の入れ子構造を解除してみます。

```
1 Nested_Data %>%
2   mutate(Model = map(data, ~lm(PPP_per_capita ~ FH_Total + Population,
3                         data = .x)),
4         Model2 = map(Model, broom::tidy, conf.int = TRUE)) %>%
5   select(-c(data, Model)) %>%
6   unnest(cols = Model2)

## # A tibble: 15 x 8
## # Groups:   Continent [5]
##   Continent term      estimate std.error statistic p.value conf.low conf.high
##   <chr>     <chr>      <dbl>     <dbl>      <dbl>    <dbl>    <dbl>    <dbl>
## 1 Asia      (Intercept) 2.13e+4   7.38e+3    2.89    6.32e-
3 6.39e+3   3.63e+4
## 2 Asia      FH_Total    6.99e+1   1.56e+2    0.449   6.56e-1 -
2.45e+2   3.85e+2
## 3 Asia      Population -1.26e-5  1.26e-5   -1.00   3.23e-1 -
3.81e-5   1.29e-5
## 4 Europe    (Intercept) -1.40e+4  9.41e+3   -1.48   1.45e-1 -
3.29e+4   5.00e+3
## 5 Europe    FH_Total    6.29e+2   1.08e+2    5.80    7.07e-
7 4.10e+2   8.48e+2
## 6 Europe    Population  1.17e-4   8.45e-5    1.39    1.73e-1 -
5.33e-5   2.88e-4
## 7 Africa    (Intercept) 3.83e+3   1.84e+3    2.09    4.20e-
2 1.44e+2   7.52e+3
## 8 Africa    FH_Total    5.11e+1   3.42e+1    1.49    1.42e-1 -
1.77e+1   1.20e+2
```

```

## 9 Africa Population -1.43e-5 2.31e-5 -0.619 5.39e-1 -
6.07e-5 3.21e-5
## 10 America (Intercept) -7.50e+3 5.93e+3 -1.27 2.15e-1 -
1.96e+4 4.57e+3
## 11 America FH_Total 3.12e+2 7.73e+1 4.03 3.20e-
4 1.54e+2 4.69e+2
## 12 America Population 9.13e-5 2.35e-5 3.89 4.77e-
4 4.35e-5 1.39e-4
## 13 Oceania (Intercept) -5.10e+4 2.34e+4 -2.18 2.73e-1 -
3.48e+5 2.46e+5
## 14 Oceania FH_Total 9.76e+2 3.26e+2 2.99 2.05e-1 -
3.17e+3 5.12e+3
## 15 Oceania Population 1.52e-4 6.27e-4 0.242 8.49e-1 -
7.81e-3 8.12e-3

```

各大陸ごとの回帰分析の推定結果が一つのデータフレームとして展開されました。ここでは FH\_Total の推定値のみがほしいので、filter() 関数を使用し、term の値が"FH\_Total"の行のみを残します。また、term列も必要なくなるので除外しましょう。

```

1 Nested_Data %>%
2   mutate(Model = map(data, ~lm(PPP_per_capita ~ FH_Total + Population, data = .x)),
3         Model2 = map(Model, broom::tidy, conf.int = TRUE)) %>%
4   select(-c(data, Model)) %>%
5   unnest(cols = Model2) %>%
6   filter(term == "FH_Total") %>%
7   select(-term)

## # A tibble: 5 x 7
## # Groups:   Continent [5]
##   Continent estimate std.error statistic      p.value conf.low conf.high
##   <chr>        <dbl>     <dbl>     <dbl>        <dbl>     <dbl>     <dbl>
## 1 Asia         69.9      156.      0.449  0.656        -245.     385.
## 2 Europe       629.      108.      5.80   0.000000707     410.     848.
## 3 Africa        51.1      34.2      1.49   0.142        -17.7     120.

```

```
## 4 America      312.      77.3      4.03  0.000320      154.      469.
## 5 Oceania      976.      326.      2.99  0.205      -3166.      5117.
```

これで終わりです。政治的自由度と所得水準の関係はアジアとアフリカでは連関の程度が小さく、統計的有意ではありません。オセアニアの場合、連関の程度は大きいと考えられますが、サンプルサイズが小さいため統計的有意な結果は得られませんでした。一方、ヨーロッパとアメリカでは連関の程度も大きく、統計的有意な連関が確認されました。

以上のコードをよりコンパクトにまとめると以下のようない下のコードとなります。

```
1 # {purrr}使用
2 Nested_Data %>%
3   mutate(Model = map(data, ~lm(PPP_per_capita ~ FH_Total + Population,
4                       data = .x)),
5         Model = map(Model, broom::tidy, conf.int = TRUE)) %>%
6   unnest(cols = Model) %>%
7   filter(term == "FH_Total") %>%
8   select(-c(data, term))
```

{purrr}パッケージを使用しない例も紹介します。むろん、以下のコードは可能な限り面倒な書き方をしています。for() 文や split() と\*apply() 関数を組み合わせると以下の例よりも簡潔なコードは作成できます。R 上級者になるためには{tidyverse}的な書き方だけでなく、ネイティブ R の書き方にも慣れる必要があります。ぜひ挑戦してみましょう。

```
1 # {purrr}を使用しない場合
2 # FH_Total の係数と標準誤差のみを抽出する例
3 lm_fit1 <- lm(PPP_per_capita ~ FH_Total + Population,
4                 data = subset(Country_df, Country_df$Continent == "Asia"))
5 lm_fit2 <- lm(PPP_per_capita ~ FH_Total + Population,
6                 data = subset(Country_df, Country_df$Continent == "Europe"))
7 lm_fit3 <- lm(PPP_per_capita ~ FH_Total + Population,
8                 data = subset(Country_df, Country_df$Continent == "Africa"))
9 lm_fit4 <- lm(PPP_per_capita ~ FH_Total + Population,
```

```
10          data = subset(Country_df, Country_df$Continent == "America"))
11 lm_fit5 <- lm(PPP_per_capita ~ FH_Total + Population,
12                 data = subset(Country_df, Country_df$Continent == "Oceania"))
13
14 lm_df <- data.frame(Continent = c("Asia", "Europe", "Africa", "America", "Oceania"),
15                       estimate = c(summary(lm_fit1)$coefficients[2, 1],
16                                   summary(lm_fit2)$coefficients[2, 1],
17                                   summary(lm_fit3)$coefficients[2, 1],
18                                   summary(lm_fit4)$coefficients[2, 1],
19                                   summary(lm_fit5)$coefficients[2, 1]),
20                       se      = c(summary(lm_fit1)$coefficients[2, 2],
21                                   summary(lm_fit2)$coefficients[2, 2],
22                                   summary(lm_fit3)$coefficients[2, 2],
23                                   summary(lm_fit4)$coefficients[2, 2],
24                                   summary(lm_fit5)$coefficients[2, 2]))
```

### 27.5.3 データの範囲を指定したモデル推定

これまでの例は名目変数でグループ化を行いましたが、連続変数を使うことも可能です。たとえば、人口 1 千万未満の国、1 千万以上 5 千万未満、5 千万以上 1 億未満、1 億以上でサンプルを分割することです。まず、Country\_df から PPP\_per\_capita、FH\_Total、Population 列のみを抽出し、Country\_df2 に格納します。

```
1 Country_df2 <- Country_df %>%
2   select(PPP_per_capita, FH_Total, Population)
```

続いて、`case_when()` 関数を使用し、Population の値を基準にケースがどの範囲内に属するかを表す変数 Group を作成します。

```
1 Country_df2 <- Country_df2 %>%
2   mutate(Group = case_when(Population < 10000000 ~ "1 千万未満",
3                           Population < 50000000 ~ "1 千万以上 5 千万未満",
4                           Population < 100000000 ~ "5 千万以上 1 億未満",
```

```
5 Population >= 100000000 ~ "1億以上"))
```

中身を確認してみましょう。

```
1 Country_df2
```

```
## # A tibble: 186 x 4
##   PPP_per_capita FH_Total Population Group
##       <dbl>      <dbl>      <dbl> <chr>
## 1     2125.      27     38928346 1千万以上5千万未満
## 2     13781.     67     28777797 1千万未満
## 3     11324.     34     43851044 1千万以上5千万未満
## 4       NA       94      77265 1千万未満
## 5     6649.      32     32866272 1千万以上5千万未満
## 6     21267.     85      97929 1千万未満
## 7     22938.     85     45195774 1千万以上5千万未満
## 8     12974.     53     2963243 1千万未満
## 9     50001.     97     25499884 1千万以上5千万未満
## 10    55824.     93     9006398 1千万未満
## # ... with 176 more rows
```

あとはこれまでの例と同じ手順となります。まず、データを Group 変数でグループ化し、入れ子構造に変換します。

```
1 Country_df2 <- Country_df2 %>%
2   group_nest(Group)
3
4 Country_df2
```

```
## # A tibble: 4 x 2
##   Group                           data
##   <chr>                           <list<tibble[,3]>>
## 1 1 億以上                      [14 x 3]
## 2 1千万以上5千万未満              [61 x 3]
```

```
## 3 1 千万未満 [96 x 3]
## 4 5 千万以上 1 億未満 [15 x 3]
```

続いて、`data` 列をデータとし、`map()` 関数でモデルの推定をしてみましょう。目的変数は所得水準、説明変数は政治的自由度とします。推定後、`{broom}` の `tidy()` 関数で推定値の情報のみを抽出し、入れ子構造を解除します。最後に `term` が `FH_Total` の行のみを残します。

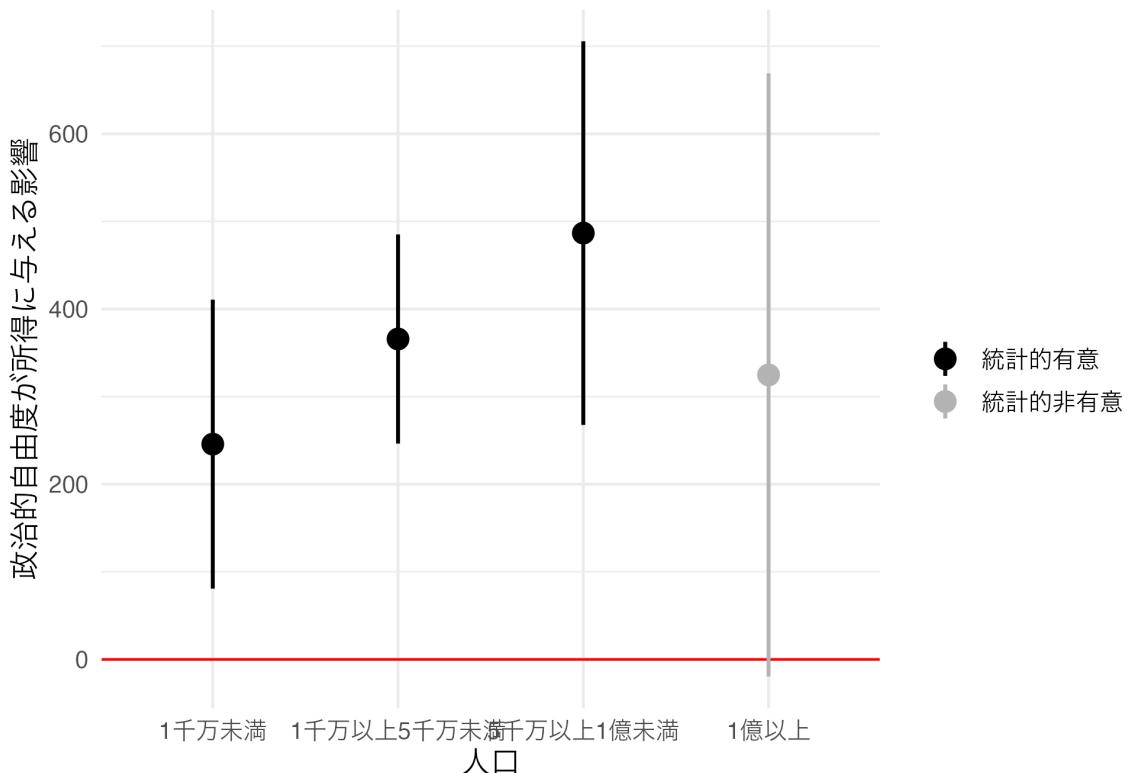
```
1 Country_df2 <- Country_df2 %>%
  2   mutate(model = map(data, ~lm(PPP_per_capita ~ FH_Total, data = .x)),
  3         est    = map(model, broom::tidy, conf.int = TRUE)) %>%
  4   unnest(est) %>%
  5   filter(term == "FH_Total") %>%
  6   select(!c(term, data, model))
  7
  8 Country_df2
```

```
## # A tibble: 4 x 7
##   Group      estimate std.error statistic   p.value conf.low conf.high
##   <chr>      <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 1 億以上     325.     158.      2.06     6.23e-2     -
## 2 1 千万以上 5 千万未満     366.     59.6      6.14     9.07e-
## 3 1 千万未満      246.     485.      246.      83.0      2.96     3.94e-
## 4 5 千万以上 1 億未満     487.     101.      487.      101.      4.80     3.46e-
```

せっかくなので推定結果を可視化してみましょう。横軸は人口規模 (Group) とし、縦軸は推定値とします。係数の点推定値と 95% 信頼区間を同時に表示するため、`geom_pointrange()` 線図オブジェクトを使用します。

```
1 Country_df2 %>%
2   mutate(
3     # 横軸の表示順番を指定するために、Group 変数を factor 化する
4     Group = factor(Group, levels = c("1 千万未満", "1 千万以上 5 千万未満",
5                                         "5 千万以上 1 億未満", "1 億以上")),
6     # 統計的有意か否かを表す Sig 変数の作成
7     Sig = if_else(conf.low * conf.high > 0, "統計的有意", "統計的非有意")
8   ) %>%
9   ggplot() +
10   geom_hline(yintercept = 0, color = "red") +
11   geom_pointrange(aes(x = Group, y = estimate,
12                       ymin = conf.low, ymax = conf.high, color = Sig),
13                     size = 0.75) +
14   labs(x = "人口", y = "政治的自由度が所得に与える影響", color = "") +
15   scale_color_manual(values = c("統計的有意" = "black",
16                                "統計的非有意" = "gray70")) +
17   theme_minimal(base_size = 12)
```



```

1  theme(legend.position = "bottom")

## List of 1
## $ legend.position: chr "bottom"
## - attr(*, "class")= chr [1:2] "theme" "gg"
## - attr(*, "complete")= logi FALSE
## - attr(*, "validate")= logi TRUE

```

政治的自由度が高くなるほど所得水準も高くなる傾向が確認されますが、人口が1億人以上の国においてはそのような傾向が見られないことが分かります。

上記の例は、ある行は一つのグループに属するケースです。しかし、ある行が複数のグループに属するケースもあるでしょう。たとえば、ある変数の値が一定値以上である行のみでグループ化する場合です。FH\_Score が一定値以上の Sub-sample に対して回帰分析を行う場合、FH\_Score が最大値（100）である国はすべての Sub-sample に属することになります。この場合は group\_by() でグループ化することが難しいかも知れません。しかし、{dplyr} の filter() を使えば簡単に処理することができます。

ここでは人口を説明変数に、所得水準を応答変数とした単回帰分析を行います。データは政治的自由度が一定値以上の国家のみに限定します。FH\_Score が 0 以上の国（すべてのケース）、10 以上の国、20 以上の国、...などにサンプルを分割してみましょう。まずは FH\_Score の最小値のみを格納した Range\_df を作成します。

```
1 Range_df <- tibble(Min_FH = seq(0, 80, by = 10))  
2  
3 Range_df  
  
## # A tibble: 9 x 1  
##   Min_FH  
##   <dbl>  
## 1     0  
## 2    10  
## 3    20  
## 4    30  
## 5    40  
## 6    50  
## 7    60  
## 8    70  
## 9    80
```

9 行 1 列の tibble が出来ました。続いて、Subset という列を生成し、ここにデータを入れてみましょう。ある変数の値を基準にサンプルを絞るには{dplyr}の filter() 関数を使用します。たとえば、Counter\_df の FH\_Total が 50 以上のケースに絞るには filter(Country\_df, FH\_Total >= 50) となります。パイプ演算子を使った場合は Country\_df %>% filter(FH\_Total >= 50) になりますが、ラムダ式とパイプ演算子の相性はあまり良くありませんので、ここではパイプ演算子なしとします。重要なのは Min\_FH の値をデータとした map() 関数を実行し、実行内容を「Country\_df から FH\_Total が Min\_FH 以上のケースのみを抽出せよ」にすることです。

```
1 Range_df <- Range_df %>%  
2   mutate(Subset = map(Min_FH, ~filter(Country_df, FH_Total >= .x)))  
3
```

```
4 Range_df
```

```
## # A tibble: 9 x 2
##   Min_FH Subset
##   <dbl> <list>
## 1      0 <spec_tbl_df [185 x 18]>
## 2     10 <spec_tbl_df [176 x 18]>
## 3     20 <spec_tbl_df [158 x 18]>
## 4     30 <spec_tbl_df [142 x 18]>
## 5     40 <spec_tbl_df [126 x 18]>
## 6     50 <spec_tbl_df [112 x 18]>
## 7     60 <spec_tbl_df [99 x 18]>
## 8     70 <spec_tbl_df [76 x 18]>
## 9     80 <spec_tbl_df [61 x 18]>
```

入れ子構造になっているのは確認できましたが、ちゃんとフィルタリングされているかどうかを確認してみましょう。たとえば、Range\_df\$Subset[[9]] だと FH\_Total が 80 以上の国に絞られていると考えられます。18 列の表ですから Country と FH\_Total 列のみを確認してみましょう。

```
1 Range_df$Subset[[9]] %>%
2   select(Country, FH_Total)
```

```
## # A tibble: 61 x 2
##   Country           FH_Total
##   <chr>              <dbl>
## 1 Andorra            94
## 2 Antigua and Barbuda 85
## 3 Argentina          85
## 4 Australia           97
## 5 Austria             93
## 6 Bahamas             91
## 7 Barbados            95
```

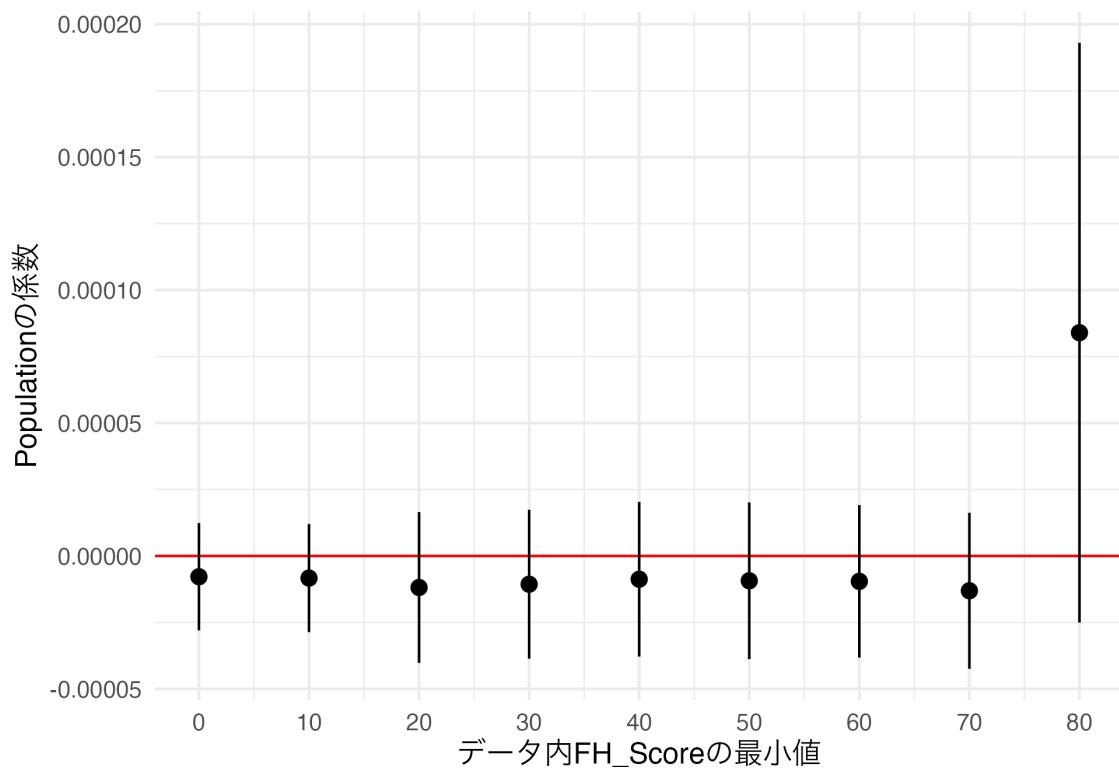
```
## 8 Belgium          96
## 9 Belize           86
## 10 Bulgaria        80
## # ... with 51 more rows
```

問題なくフィルタリングされているようですね。ここまで出来ればあとはこれまでの内容の復習でしょう。

```
1 Range_df <- Range_df %>%
2   mutate(Model = map(Subset, ~lm(PPP_per_capita ~ Population, data = .x)),
3         Model = map(Model, broom::tidy, conf.int = TRUE)) %>%
4   unnest(cols = Model) %>%
5   filter(term == "Population") %>%
6   select(-c(Subset, term))
```

今回も可視化してみましょう。

```
1 Range_df %>%
2   ggplot() +
3   geom_hline(yintercept = 0, color = "red") +
4   geom_pointrange(aes(x = Min_FH, y = estimate,
5                       ymin = conf.low, ymax = conf.high)) +
6   labs(x = "データ内 FH_Score の最小値", y = "Population の係数") +
7   scale_x_continuous(breaks = seq(0, 80, by = 10),
8                       labels = seq(0, 80, by = 10)) +
9   theme_minimal(base_size = 12)
```



以上の例は実際の分析では多く使われる方法ではないかも知れません。しかし、ノンパラメトリック回帰不連続デザイン (Regression Discontinuity Design; RDD) の場合、感度分析を行う際、バンド幅を色々と調整しながら局所処置効果を推定します。RDD の詳細については矢内の授業資料、および SONG の授業資料などに譲りますが、ハンド幅の調整はデータの範囲を少しづつ拡張（縮小）させながら同じ分析を繰り返すことです。以下ではアメリカ上院選挙のデータを例に、方法を紹介します。

使用するデータは{rdrobust}パッケージが提供する `rdrobust_RDsenate` というデータセットです。{pacman}パッケージの `p_load()` 関数を使ってパッケージのインストールと読み込みを行い、`data()` 関数を使って、データを読み込みます。

```

1 pacman::p_load(rdrobust) # {rdrobust}のインストール & 読み込み
2 data("rdrobust_RDsenate") # rdrobust_RDsenate データの読み込み
3
4 # データを Senate_df という名で格納
5 Senate_df <- rdrobust_RDsenate

```

Senate\_df の中身を確認してみます。

```
1 head(Senate_df)

##           margin      vote
## 1  -7.6885610 36.09757
## 2  -3.9237082 45.46875
## 3  -6.8686604 45.59821
## 4 -27.6680565 48.47606
## 5  -8.2569685 51.74687
## 6   0.7324815 39.80264
```

本データの詳細は Cattaneo et al. [2015] に譲りますが、ここでは簡単に説明します。`margin` はある選挙区の民主党候補者の得票率から共和党候補者の投票率を引いたものです。100 なら民主党候補者の圧勝、-100 なら共和党候補者の圧勝です。これが 0 に近いと辛勝または惜敗となり、この辺の民主党候補者と共和党候補者は候補者としての資質や資源が近いと考えられます。`vote` は次回の選挙における民主党候補者の得票率です。ある候補者が現職であることによるアドバンテージを調べる際、「民主党が勝った選挙区における次回選挙での民主党候補者の得票率」から「民主党が負けた選挙区における次回選挙での民主党候補者の得票率」を引くだけでは不十分でしょう。圧勝できた選挙区は次回でも得票率が高いと考えられますが、これは現職というポジションによるアドバンテージ以外にも、候補者がもともと備えている高い能力・資源にも起因するからです。だから辛勝・惜敗の選挙区のみに限定することで、現職と新人の能力・資源などを出来る限り均質にし、「現職」というポジションだけが異なる状況を見つけることになります。

問題は辛勝と惜敗の基準をどう決めるかですが、広く使われている方法としては Imbens and Kalyanaraman [2012] の最適バンド幅 (optimal bandwidth) が広く使われます。しかし、最適バンド幅を使うとしても感度分析 (sensitivity analysis) を行うケースも多く、この場合、バンド幅を少しづつ変えながら現職の効果を推定することになります。推定式は以下のようになります。 $I(\text{margin} > 0)$  は `margin` が 0 より大きい場合、1 となる指示関数 (indicator function) です。

$$\hat{\text{vote}} = \beta_0 + \beta_1 \cdot \text{margin} + \tau \cdot I(\text{margin} > 0) \quad \text{where} \quad -h \leq \text{margin} \leq h$$

それではまずは  $h$  が 100、つまり全データを使った分析を試しにやってみましょう。

```
1 RDD_Fit <- lm(vote ~ margin + I(margin > 0), data = Senate_df)
2
3 summary(RDD_Fit)

##
## Call:
## lm(formula = vote ~ margin + I(margin > 0), data = Senate_df)
##
## Residuals:
##       Min     1Q Median     3Q    Max
## -46.930 -6.402  0.132  7.191 46.443
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)    
## (Intercept)             47.33084   0.54192  87.340 < 2e-16 ***
## margin                  0.34806   0.01335  26.078 < 2e-16 ***
## I(margin > 0)TRUE     4.78461   0.92290   5.184 2.51e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.78 on 1294 degrees of freedom
## (93 observations deleted due to missingness)
## Multiple R-squared:  0.5781, Adjusted R-squared:  0.5774 
## F-statistic: 886.4 on 2 and 1294 DF,  p-value: < 2.2e-16
```

$I(margin > 0)TRUE$  の係数 4.785 が現職効果です。この作業を  $h = 100$  から  $h = 10$  まで、5 ずつ変えながら分析を繰り返してみます。まずは、RDD\_df という tibble を作成し、BW 列には `c(100, 95, 90, ... 10)` を入れます。続いて、`filter()` 関数を利用して Senate\_df から margin が-BW 以上、BW 以下のデータを抽出し、Subset という名の列として格納します。

```
1  RDD_df <- tibble(BW = seq(100, 10, by = -5))
2
3  RDD_df <- RDD_df %>%
4    mutate(Subset = map(BW, ~filter(Senate_df, margin >= -.x & margin <= .x)))
5
6  RDD_df
7
8  ## # A tibble: 19 x 2
9  ##       BW   Subset
10 ##   <dbl> <list>
11 ## 1 100 <df [1,390 x 2]>
12 ## 2 95 <df [1,327 x 2]>
13 ## 3 90 <df [1,319 x 2]>
14 ## 4 85 <df [1,308 x 2]>
15 ## 5 80 <df [1,303 x 2]>
16 ## 6 75 <df [1,294 x 2]>
17 ## 7 70 <df [1,287 x 2]>
18 ## 8 65 <df [1,270 x 2]>
19 ## 9 60 <df [1,256 x 2]>
20 ## 10 55 <df [1,237 x 2]>
21 ## 11 50 <df [1,211 x 2]>
22 ## 12 45 <df [1,178 x 2]>
23 ## 13 40 <df [1,128 x 2]>
24 ## 14 35 <df [1,077 x 2]>
25 ## 15 30 <df [995 x 2]>
26 ## 16 25 <df [901 x 2]>
27 ## 17 20 <df [778 x 2]>
28 ## 18 15 <df [639 x 2]>
29 ## 19 10 <df [471 x 2]>
30
31 RDD_df <- RDD_df %>%
32   mutate(
33     # 各 Subset に対し、回帰分析を実施し、Model 列に格納
```

```
4     Model  = map(Subset, ~lm(vote ~ margin * I(margin > 0), data = .x)),
5     # Model 列の各セルから{broom}の tidy() で推定値のみ抽出
6     Est     = map(Model, broom::tidy, conf.int = TRUE)
7     ) %>%
8     # Est 列の入れ子構造を解除
9     unnest(Est) %>%
10    # 現職効果に該当する行のみを抽出
11    filter(term == "I(margin > 0)TRUE") %>%
12    # 不要な列を削除
13    select(!c(Subset, Model, term, statistic, p.value))
14
15 RDD_df
```

```
## # A tibble: 19 x 5
##       BW estimate std.error conf.low conf.high
##   <dbl>    <dbl>     <dbl>    <dbl>     <dbl>
## 1    100     6.04    0.942     4.20     7.89
## 2     95     5.76    0.975     3.85     7.67
## 3     90     5.90    0.981     3.98     7.83
## 4     85     5.80    0.993     3.85     7.75
## 5     80     5.10    0.999     3.14     7.06
## 6     75     5.18    1.01      3.21     7.15
## 7     70     5.57    1.01      3.58     7.55
## 8     65     5.21    1.03      3.20     7.23
## 9     60     4.89    1.04      2.86     6.93
## 10    55     5.23    1.05      3.17     7.29
## 11    50     5.73    1.07      3.64     7.82
## 12    45     5.94    1.09      3.80     8.09
## 13    40     6.12    1.12      3.92     8.33
## 14    35     6.34    1.15      4.09     8.59
## 15    30     7.18    1.18      4.86     9.50
## 16    25     7.38    1.21      5.00     9.76
```

|       |    |      |      |      |      |
|-------|----|------|------|------|------|
| ## 17 | 20 | 7.03 | 1.32 | 4.43 | 9.62 |
| ## 18 | 15 | 6.96 | 1.50 | 4.01 | 9.92 |
| ## 19 | 10 | 6.90 | 1.75 | 3.46 | 10.3 |

19回の回帰分析が数行のコードで簡単に実施でき、必要な情報も素早く抽出することができました。

以上の例は、交互作用なしの線形回帰分析による局所処置効果の推定例です。しかし、ノンパラメトリック RDD では、割当変数 (running variable) 処置変数の交差項や割当変数の二乗項を入れる場合が多いです。今回の割当変数は `margin` です。また、閾値 (cutpoint) に近いケースには高い重みを付けることが一般的な作法であり、多く使われるのが三角 (triangular) カーネルです。上記の例はすべてのケースに同じ重みを付ける矩形 (rectangular) カーネルを使った例です。

以上の理由により、やはり RDD は専用のパッケージを使った方が良いかも知れません。既に読み込んである{rdroburst}も推定可能ですが、ここでは{rdd}パッケージを使ってみましょう。{rdd}パッケージによる RDD は `RDestimate()` 関数を使います。実際の例を確認してみましょう。`map()` を使う前に、オブジェクトの構造を把握しておくことは重要です。

```
1 # cutpoint 引数の既定値は 0 であるため、今回の例では省略可能
2 RDestimate(応答変数 ~ 割当変数, data = データオブジェクト, cutpoint = 閾値, bw = バンド幅)
3
4 pacman::p_load(rdd) # パッケージの読み込み
5
6 # バンド幅を指定した RDD 推定
7 RDD_Fit <- RDestimate(vote ~ margin, data = Senate_df, bw = 100)
8
9 # RDD の推定結果を見る
10 summary(RDD_Fit)
11
12 ## Call:
13 ## RDestimate(formula = vote ~ margin, data = Senate_df, bw = 100)
14 ##
```

```

## Type:
## sharp
##
## Estimates:
##           Bandwidth Observations Estimate Std. Error z value Pr(>|z|)
## LATE      100        1258     5.637   0.8845  6.374  1.842e-
##          10
## Half-BW   50         1127     6.480   1.0040  6.455  1.084e-
##          10
## Double-BW 200        1297     5.842   0.8568  6.818  9.210e-
##          12
##
## LATE      ***
## Half-BW   ***
## Double-BW ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## F-statistics:
##           F     Num. DoF Denom. DoF p
## LATE      316.0  3        1254    0
## Half-BW   179.7  3        1123    0
## Double-BW 506.0  3        1293    0

```

現職効果は 5.637、その標準誤差は 0.884 です。これらの情報はどこから抽出できるか確認してみます。

```

1 # 推定値の情報がどこに格納されているかを確認
2 str(RDD_Fit)

```

```

## List of 12
## $ type     : chr "sharp"
## $ call     : language RDestimate(formula = vote ~ margin, data = Senate_df, bw = 100
## $ est      : Named num [1:3] 5.64 6.48 5.84

```

```

##  ..- attr(*, "names")= chr [1:3] "LATE" "Half-BW" "Double-BW"
## $ bw      : num [1:3] 100 50 200
## $ se      : num [1:3] 0.884 1.004 0.857
## $ z       : num [1:3] 6.37 6.45 6.82
## $ p       : num [1:3] 1.84e-10 1.08e-10 9.21e-12
## $ obs     : num [1:3] 1258 1127 1297
## $ ci      : num [1:3, 1:2] 3.9 4.51 4.16 7.37 8.45 ...
## $ model   : list()
## $ frame   : list()
## $ na.action: int [1:93] 28 29 57 58 86 113 114 142 143 168 ...
## - attr(*, "class")= chr "RD"

```

\$est と \$se に私たちが探している数値が入っていますね。それぞれ抽出してみると長さ 3 の numeric ベクトルが output され、その中で 1 番目の要素が LATE ということが分かります。

```

1 # est 要素の 1 番目の要素が LATE
2 RDD_Fit$est

##      LATE    Half-BW Double-BW
##  5.637474  6.480344  5.842049

1 # se 要素の 1 番目の要素が LATE の標準誤差
2 RDD_Fit$se

## [1] 0.8844541 1.0039734 0.8568150

```

それでは分析に入ります。今回も *h* を予め BW という名の列で格納した RDD\_df を作成します。

```
1 RDD_df <- tibble(BW = seq(100, 10, by = -5))
```

今回はラムダ式を使わず、事前に関数を定義しておきましょう。関数の自作については第 11 章を参照してください。

```
1 # 引数を x とする RDD_Func() の定義
2 RDD_Func <- function(x) {
3   # バンド幅を x にした RDD 推定
4   Temp_Est <- RDestimate(vote ~ margin, data = Senate_df, bw = x)
5   # LATE とその標準誤差を tibble として返す
6   tibble(LATE = Temp_Est$est[1],
7         SE     = Temp_Est$se[1])
8 }
```

問題なく動くかを確認してみましょう。

```
1 RDD_Func(100)
```

```
## # A tibble: 1 x 2
##      LATE     SE
##      <dbl> <dbl>
## 1  5.64  0.884
```

それでは分析をやってみましょう。今回の場合、`map()` の第二引数はラムダ式ではないため~で始める必要ありません。関数名だけで十分です。

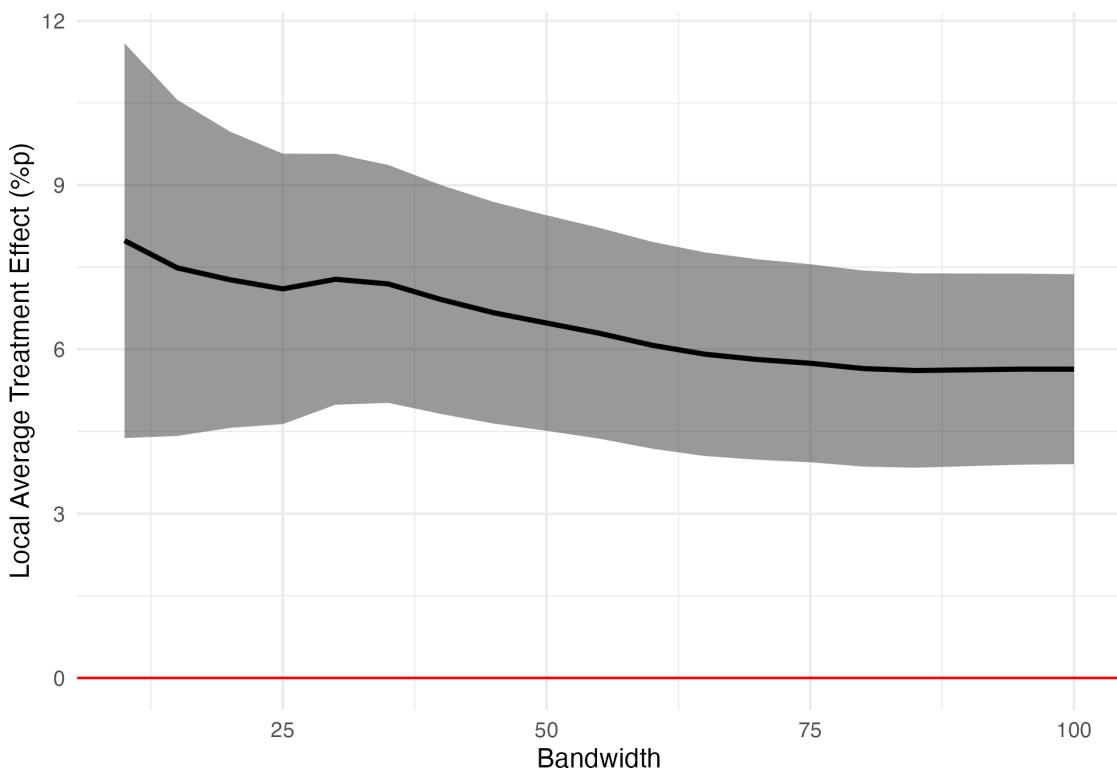
```
1 RDD_df <- RDD_df %>%
2   mutate(RDD = map(BW, RDD_Func)) %>%
3   unnest(RDD)
4
5 RDD_df
```

```
## # A tibble: 19 x 3
##      BW     LATE     SE
##      <dbl> <dbl> <dbl>
## 1    100  5.64  0.884
## 2     95  5.64  0.890
## 3     90  5.63  0.897
## 4     85  5.61  0.905
```

```
## 5 80 5.65 0.913
## 6 75 5.75 0.922
## 7 70 5.81 0.934
## 8 65 5.91 0.949
## 9 60 6.07 0.963
## 10 55 6.29 0.982
## 11 50 6.48 1.00
## 12 45 6.67 1.03
## 13 40 6.91 1.07
## 14 35 7.19 1.11
## 15 30 7.28 1.17
## 16 25 7.10 1.26
## 17 20 7.27 1.38
## 18 15 7.49 1.57
## 19 10 7.98 1.84
```

せっかくなので可視化してみましょう。今回は `geom_pointrange()` を使わず、`geom_line()` と `geom_ribbon()` を組み合わせます。`geom_line()` は LATE を、`geom_ribbon()` は 95% 信頼区間を表します。作図の前に 95% 信頼区間を計算しておきます。

```
1 RDD_df %>%
2   mutate(CI_lwr = LATE + qnorm(0.025) * SE,
3         CI_upr = LATE + qnorm(0.975) * SE) %>%
4   ggplot() +
5   geom_hline(yintercept = 0, color = "red") +
6   geom_ribbon(aes(x = BW, ymin = CI_lwr, ymax = CI_upr), alpha = 0.5) +
7   geom_line(aes(x = BW, y = LATE), size = 1) +
8   labs(x = "Bandwidth", y = "Local Average Treatment Effect (%p)") +
9   theme_minimal()
```



#### 27.5.4 説明・応答変数を指定したモデル推定

続いて、同じデータに対して推定式のみを変えた反復推定をやってみましょう。たとえば、応答変数は固定し、グループ変数のみを変えながら t 検定を繰り返すケースを考えてみましょう。たとえば、OECD に加盟しているかどうか (OECD) で購買力平価 GDP (PPP) の差があるかを検定してみましょう。使用する関数は `t.test()` です。第一引数には応答変数 ~ 説明変数とし、`data` 引数にはデータフレームや `tibble` オブジェクトを指定します。

```
1 Diff_test <- t.test(PPP ~ G20, data = Country_df)
```

```
2
```

```
3 Diff_test
```

```
##  
## Welch Two Sample t-test  
##
```

```
## data: PPP by G20
## t = -3.4137, df = 18.012, p-value = 0.003094
## alternative hypothesis: true difference in means between group 0 and group 1 is
## 95 percent confidence interval:
## -7667830 -1825482
## sample estimates:
## mean in group 0 mean in group 1
## 211287 4957943
```

ここからいくつかの情報が読み取れます。まず、OECD 加盟国 の PPP は 4957943、非加盟国は 211287 です。その差分は-4746656 です。また、t 値は-3.4136、p 値は 0.003 です。これらの情報を効率よく抽出するには `broom::tidy()` が便利です。

```
1 broom::tidy(Diff_test)

## # A tibble: 1 x 10
##   estimate estimate1 estimate2 statistic p.value parameter conf.low conf.high
##   <dbl>     <dbl>     <dbl>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 -4746656.  211287.  4957943. -3.41  0.00309  18.0 -7667830. -1825482.
## # ... with 2 more variables: method <chr>, alternative <chr>
```

差分、t 値、p 値、95% 信頼区間などが抽出できます。これを OECD だけでなく、G7 や G20 に対しても同じ検定を繰り返してみましょう。

`t.test()` の第一引数だけを変えながら推定をすれば良いでしょう。この第一引数、たとえば `PPP ~ G20` の方は formula 型と呼ばれる R におけるデータ型の一つです。これは character 型でないことに注意してください。

```
1 Formula1 <- "PPP ~ G20"
2 t.test(Formula1, data = Country_df)

## Warning in mean.default(x): argument is not numeric or logical: returning NA
## Warning in var(x): NAs introduced by coercion
```

```
## Error in t.test.default(Formula1, data = Country_df): not enough 'x' observations
```

このようにエラーが出ます。この `Formula` を `as.formula()` 関数を使って `formula` 型に変換し、推定をやってみると問題なく推定できることが分かります。

```
1 Formula2 <- as.formula(Formula1)
2 t.test(Formula2, data = Country_df)
```

```
##
##  Welch Two Sample t-test
##
## data:  PPP by G20
## t = -3.4137, df = 18.012, p-value = 0.003094
## alternative hypothesis: true difference in means between group 0 and group 1 is not equal to zero
## 95 percent confidence interval:
## -7667830 -1825482
## sample estimates:
## mean in group 0 mean in group 1
##           211287           4957943
```

これから私たちがやるのは"PPP ~ "の次に"G7"、"G20"、"OECD"を `paste()` や `paste0()` 関数を結合し、`formula` 型に変換することです。`formula` 型の列さえ生成できれば、あとは `map()` 関数に `formula` を渡し、データは `Country_df` に固定するだけです。

まずはどの変数を説明変数にするかを `Group` 列として格納した `Diff_df` を作成します。

```
1 Diff_df <- tibble(Group = c("G7", "G20", "OECD"))
2
3 Diff_df
```

```
## # A tibble: 3 x 1
##   Group
##   <chr>
## 1 G7
## 2 G20
```

```
## 3 OECD
```

続いて、Formula列を作成し、"PPP ~ "の次にGroup列の文字列を結合します。

```
1 Diff_df <- Diff_df %>%
2   mutate(Formula = paste0("PPP ~ ", Group))
3
4 Diff_df
```

```
## # A tibble: 3 x 2
##   Group Formula
##   <chr> <chr>
## 1 G7    PPP ~ G7
## 2 G20   PPP ~ G20
## 3 OECD  PPP ~ OECD
```

今のFormula列のデータ型を確認してみましょう。

```
1 class(Diff_df$Formula[[1]])
```

```
## [1] "character"
```

character型ですね。続いて、map()関数を使用してFormula列をformula型に変換します。

```
1 Diff_df <- Diff_df %>%
2   mutate(Formula = map(Formula, as.formula))
3
4 Diff_df
```

```
## # A tibble: 3 x 2
##   Group Formula
##   <chr> <list>
## 1 G7    <formula>
## 2 G20   <formula>
## 3 OECD  <formula>
```

データ型も確認してみましょう。

```
1 class(Diff_df$Formula[[1]])
```

```
## [1] "formula"
```

formula 型になっていることが分かります。

もう一つの方法としては最初から formula 型の変数を入れても良いでしょう。推定するモデルが多くない場合は、こちらの方が簡単かも知れません。たとえば、これまでの作業を直接 formula 型を格納する形式で行うなら以下のようにになります。

```
1 # 方法 1
2 Diff_df <- tibble(Group = c("G7", "G20", "OECD"),
3                     Formula = c(PPP ~ G7,
4                                PPP ~ G20,
5                                PPP ~ OECD))
6 # 方法 2
7 Diff_df <- list("G7" = PPP ~ G7,
8                  "G20" = PPP ~ G20,
9                  "OECD" = PPP ~ OECD) %>%
10 enframe(name = "Group", value = "Formula")
```

それでは *t* 検定を行います。ここでもラムダ式を使います。式が入る場所には.x を指定し、データは Country\_df に固定します。

```
1 Diff_df <- Diff_df %>%
2   mutate(Model = map(Formula, ~t.test(.x, data = Country_df)))
3
4 Diff_df
```

```
## # A tibble: 3 x 3
##   Group Formula  Model
##   <chr> <list>   <list>
## 1 G7    <formula> <htest>
## 2 G20   <formula> <htest>
```

```
## 3 OECD <formula> <htest>
```

broom::tidy() で推定値の要約を抽出し、入れ子構造を解除します。

```
1 Diff_df <- Diff_df %>%
2   mutate(Tidy = map(Model, broom::tidy)) %>%
3   unnest(Tidy)
4
5 Diff_df
```

```
## # A tibble: 3 x 13
##   Group Formula  Model   estimate estimate1 estimate2 statistic p.value
##   <chr> <list>   <list>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 G7   <formula> <htest> -5363390.   507033.   5870423.    -
2.14 0.0759
## 2 G20  <formula> <htest> -4746656.   211287.   4957943.    -
3.41 0.00309
## 3 OECD  <formula> <htest> -1166591.   475459.   1642050.    -
1.96 0.0564
## # ... with 5 more variables: parameter <dbl>, conf.low <dbl>, conf.high <dbl>,
## #   method <chr>, alternative <chr>
```

いくつか使用しない情報もあるので、適宜列を削除します。

```
1 Diff_df <- Diff_df %>%
2   select(-c(Formula, Model, estimate1, estimate2,
3             p.value, method, alternative))
4
5 Diff_df
```

```
## # A tibble: 3 x 6
##   Group  estimate statistic parameter  conf.low conf.high
##   <chr>    <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 G7    -5363390.    -2.14     6.04 -11486296.    759515.
## 2 G20   -4746656.    -3.41     18.0  -7667830.   -1825482.
```

```
## 3 OECD -1166591. -1.96 42.8 -2366187. 33005.
```

以上のコードをパイプ演算子を使用し、簡潔にまとめると以下のようにになります。

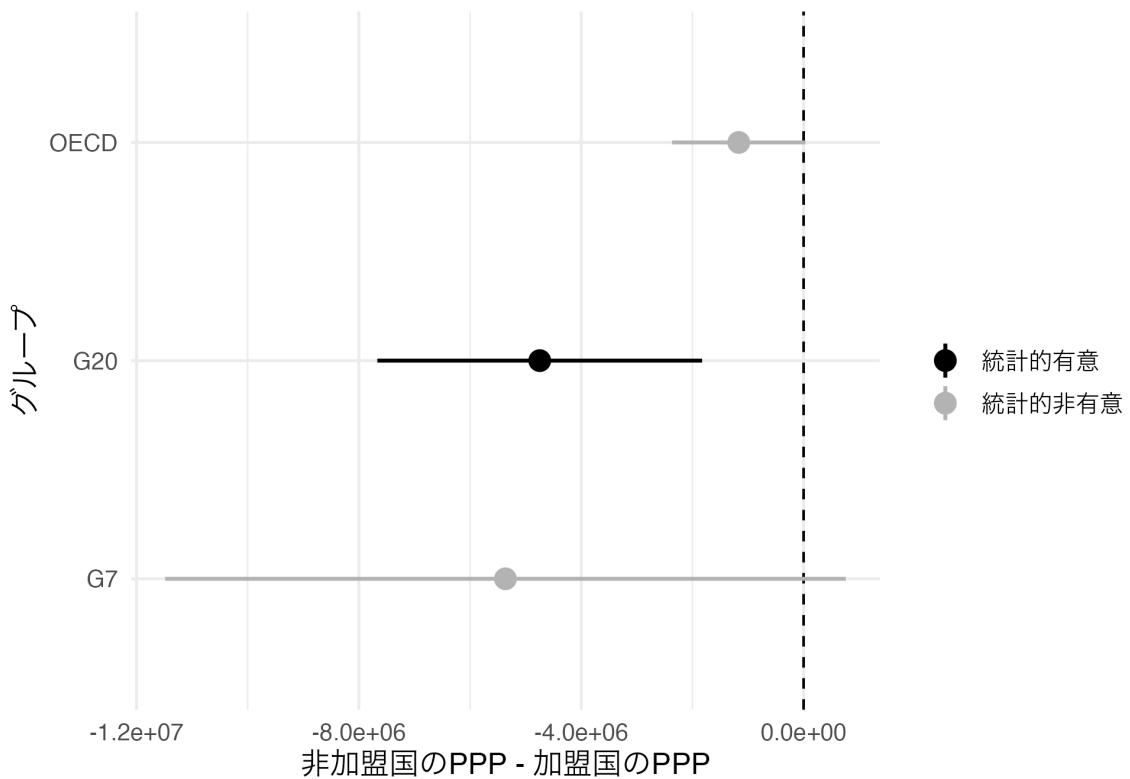
```
1 Diff_df <- tibble(Group = c("G7", "G20", "OECD"))

2

3 Diff_df <- Diff_df %>%
4   mutate(Formula = paste0("PPP ~ ", Group),
5         Formula = map(Formula, as.formula),
6         Model = map(Formula, ~t.test(.x, data = Country_df)),
7         Tidy = map(Model, broom::tidy)) %>%
8   unnest(Tidy) %>%
9   select(-c(Formula, Model, estimate1, estimate2,
10             p.value, method, alternative))
```

推定結果を可視化してみましょう。

```
1 Diff_df %>%
2   mutate(Group = fct_inorder(Group),
3         Sig = if_else(conf.low * conf.high > 0, "統計的有意",
4                       "統計的非有意")) %>%
5   ggplot() +
6   geom_vline(xintercept = 0, linetype = 2) +
7   geom_pointrange(aes(x = estimate, xmin = conf.low, xmax = conf.high,
8                     y = Group, color = Sig), size = 0.75) +
9   scale_color_manual(values = c("統計的有意" = "black",
10                      "統計的非有意" = "gray70")) +
11  labs(x = "非加盟国のPPP - 加盟国のPPP", y = "グループ", color = "") +
12  theme_minimal(base_size = 12)
```



```
1  theme(legend.position = "bottom")  
  
## List of 1  
## $ legend.position: chr "bottom"  
## - attr(*, "class")= chr [1:2] "theme" "gg"  
## - attr(*, "complete")= logi FALSE  
## - attr(*, "validate")= logi TRUE
```

以上 の方法を応用すると応答変数を変えながら分析を繰り返すこともできます。他にも説明変数が2つ以上のケースにも応用可能です。

## 第 28 章

# オブジェクト指向型プログラミング

### 28.1 まずは例から

```
1 Vector1 <- c(1, 5, 3, 7, 9, 12, 5, 4, 10, 1)
```

```
2 Vector2 <- c("A", "B", "D", "B", "E", "A", "A", "D", "C", "C")
```

```
1 summary(Vector1)
```

```
##   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
##   1.00    3.25   5.00   5.70   8.50  12.00
```

```
1 summary(Vector2)
```

```
##   Length   Class   Mode
##   10 character character
```

同じ `summary()` 関数ですが、中のデータのタイプによって動きが異なります。これがオブジェクト指向プログラミングにおいて「**多態性** (polymorphism)」と呼ばれる概念です。同じ関数でもデータ型、またはデータ構造に応じて異なる動きをすることです。ここでのデータ型やデータ構造を、OOP では「**クラス** (class)」と呼びます。クラスは

`class()` 関数で確認することができます。

```
1  class(Vector1)

## [1] "numeric"

1  class(Vector2)

## [1] "character"
```

`Vector1` は numeric、`Vector2` は character です。もし、無理矢理に `Vector1` のクラスを character に変えればどうなるでしょうか。クラスの変更は `class(オブジェクト名) <- "クラス名"` でできます。一つのオブジェクトは複数のクラスが持てますが、これは OOP の「**継承** (inheritance)」概念に関係するので後で解説します。ここではまず、`Vector1` のクラスを character にし、もう一回 `summary()` を使ってみましょう。

```
1  class(Vector1) <- "character"
2  summary(Vector1)

##      Length     Class      Mode
##          10 character character
```

データの中身は変わっていませんが、`summary()` 関数の動き方が変わりました。このように、R で頻繁に使う `summary()`、`print()`、`plot()` などの関数は様々なクラスの対応しております。`lm()` 関数を使った回帰分析の結果オブジェクトのクラス名は `lm` であり、その結果を見るためにも `summary()` 関数を使います。他にも `plot(lm オブジェクト名)` をすると回帰診断の図が表示されます。これができないと、各クラスに応じた関数を作成する必要がありますね。numeric 型専用の `numeric_print()`、character 型専用の `character_print()`、lm 型専用の `lm_plot()` など…、覚えなきゃいけない関数が増えてきます。ユーザー側でも大変ですが、コードを作成する側も大変です。実際、図 28.1<sup>1)</sup> を見ると、プログラマーにとって最も大変な仕事は「名付け」であることが分かります。

---

<sup>1)</sup> 図の出典は IT WORLD です (アクセス: 2020-05-21)。

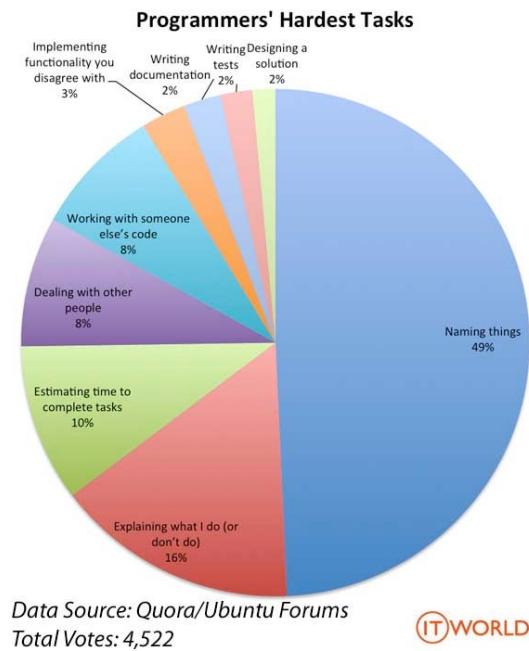


図 28.1: Programmers' Hardest Tasks

OOP の多態性にはこのような煩わしい仕事を軽減する機能があります。OOP にはここで紹介した多態性以外にも、「**継承** (inheritance)」、「**カプセル化** (encapsulation)」のような特徴があります。他にも人によっては「メッセージパッシング (message passing)」、「動的バインディング (dynamic binding)」などの特徴を述べたりしますが、詳しい話は専門書に譲りたいと思います。また、ここでは R の S3 クラスについて解説しますが、S3 はカプセル化に対応しておりません。したがって、ここでは以下の概念について例と一緒に解説していきたいと思います。

1. オブジェクト (object)
2. クラス (class)
3. メソッド (method)
4. 多態性 (polymorphism)
5. 継承 (inheritance)

## 28.2 OOP とは

### 28.2.1 オブジェクト

ここは第 10 章の内容の繰り返しですが、**オブジェクト (object)** とはメモリに割り当てられた「何か」です。「何か」に該当するのは、ベクトル (vector)、行列 (matrix)、データフレーム (data frame)、リスト (list)、関数 (function) などがあります。一般的に、オブジェクトにはそれぞれ固有の（つまり、他のオブジェクトと重複しない）名前が付いています。

たとえば、1 から 5 までの自然数の数列を

```
1 my_vec1 <- c(1, 2, 3, 4, 5) # my_vec1 <- 1:5 でも同じ
```

のように `my_vec1` という名前のオブジェクトに格納します。オブジェクトに名前をつけてメモリに割り当てると、その後 `my_vec1` と入力するだけでそのオブジェクトの中身を読み込むことができるようになります。

ここで、次のように `my_vec1` の要素を 2 倍にする操作を考えてみましょう。

```
1 my_vec1 * 2
## [1] 2 4 6 8 10
```

`my_vec1` は、先ほど定義したオブジェクトです。では 2 はどうでしょうか。2 はメモリに割り当てられていないので、オブジェクトではないでしょうか。実は、この数字 2 もオブジェクトです。計算する瞬間のみ 2 がメモリに割り当てられ、計算が終わったらメモリから消されると考えれば良いでしょう。もちろん、\* のような演算子でさえもオブジェクトです。

### 28.2.2 クラス

**クラス (class)** とはオブジェクトを特徴づける属性のことです。既に何度か `class()` 関数を使ってデータ型やデータ構造を確認しましたが、`class()` 関数でオブジェクトのク

ラスを確認することができます。先ほど、`my_vec1` も`2`もオブジェクトであると説明しました。これらがすべてオブジェクトであるということは、何らかのクラス属性を持っているということです。また、`class()` 関数そのものもオブジェクトなので、何らかのクラスを持ちます。確認してみましょう。

```
1 class(my_vec1)

## [1] "numeric"

1 class(`*`)

## [1] "function"

1 class(2)

## [1] "numeric"

1 class(class)

## [1] "function"
```

統計分析をする際に、R のクラスを意識することはありません。しかし、R でオブジェクト指向プログラミングを行う際は、オブジェクトのクラスを厳密に定義する必要があります。

R における全てはオブジェクトであり、全てのオブジェクトは一つ以上クラスが付与されています。このクラスの考え方はプログラミング言語によって異なります。たとえば、Python の場合、一つのクラスの内部にはオブジェクトのデータ構造が定義され、そのクラスで使用可能な関数も含んでいます。また、データを含む場合もあります。このようにクラス内部にデータ、データ構造、専用関数などを格納することをカプセル化 (encapsulation) と呼びます。

一方、R の (S3) クラスにはクラス専用関数がクラス内で定義されておらず、データのみが格納されています。

### 28.2.3 メソッドと多態性

各クラス専用の関数をメソッド (method) と呼びます。たとえば、`summary()` 関数を考えてみましょう。`lm()` 関数を用いた回帰分析から得られたオブジェクトのクラスは `lm` であり、`c()` で作られた数値型ベクトルのクラスは `numeric` です。しかし、同じ `summary()` 関数ですが、引数のクラスが `lm` か `numeric` かによって異なる動きを見せます。その例を見ましょう。

```
1 X <- c(1, 3, 5, 7, 9, 11)
2 Y <- c(1, 2, 3, 7, 11, 13)
3 lm_class <- lm(Y ~ X)
4 class(X)

## [1] "numeric"

1 class(lm_class)

## [1] "lm"

1 summary(X)

##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
##      1.0     3.5     6.0     6.0     8.5    11.0

1 summary(lm_class)

##
## Call:
## lm(formula = Y ~ X)
##
## Residuals:
##      1       2       3       4       5       6
##  1.3333 -0.2667 -1.8667 -0.4667  0.9333  0.3333
##
## Coefficients:
##
```

```

##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.6333     1.0546  -1.549  0.19637
## X           1.3000     0.1528   8.510  0.00105 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.278 on 4 degrees of freedom
## Multiple R-squared:  0.9477, Adjusted R-squared:  0.9346
## F-statistic: 72.43 on 1 and 4 DF,  p-value: 0.001046

```

このように同じ関数でもクラスによって異なる動作をすることを多態性 (polymorphism) と呼びます。しかし、実は R においてこれらの関数は別途作られた関数です。つまり、`summary()` という関数がクラスごとに定義されていることを意味します。`summary()` 関数がどのクラスで使用可能かを確認するためには `methods()` 関数を使います。

```

1 methods("summary")

## [1] summary,ANY-method           summary,DBIObject-method
## [3] summary.aov                 summary.aovlist*
## [5] summary.aspell*              summary.check_packages_in_dir*
## [7] summary.connection           summary.data.frame
## [9] summary.Date                 summary.default
## [11] summary.Duration*            summary.ecdf*
## [13] summary.factor               summary.ggplot*
## [15] summary.glm                 summary.haven_labelled*
## [17] summary.hcl_palettes*        summary.infl*
## [19] summary.Interval*            summary.lm
## [21] summary.loess*               summary.manova
## [23] summary.matrix               summary.mlm*
## [25] summary.nls*                 summary.packageStatus*
## [27] summary.Period*              summary.POSIXct
## [29] summary.POSIXlt              summary.ppr*
## [31] summary.prcomp*              summary.princomp*
## [33] summary.proc_time            summary.rlang_error*

```

```

## [35] summary.rlang_message*           summary.rlang_trace*
## [37] summary.rlang_warning*          summary.rlang:::list_of_conditions*
## [39] summary.srcfile                 summary.srcref
## [41] summary.stepfun                summary.stl*
## [43] summary.table                 summary.tukeysmooth*
## [45] summary.vctrs_sclr*            summary.vctrs_vctr*
## [47] summary.warnings
## see '?methods' for accessing help and source code

```

このように47種類のクラスに対して `summary()` 関数が定義されています<sup>2)</sup>。この関数の内部を確認するにはどうすれば良いでしょうか。関数のコードを見るときにはコンソール上に関数名を入力するだけです（()は不要）。

```

1  summary

## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x7f9d2fcb3580>
## <environment: namespace:base>

```

しかし、多態性を持つ関数の内部を見ることはできません。そもそも `summary()` 関数はクラスごとに異なるコードを持っているため、`summary`だけでは「どのクラスの `summary()` か」が分かりません。それでもRには `summary()` 関数が存在し、それをジェネリック関数(generic function)と呼びます。内部には `UseMethod("summary")` のみが書かれており、これは「この `summary()` 関数は様々なクラスのメソッドとして機能するぞ」と宣言しているだけです。各クラスに対応したメソッドの内部を見るには `getS3method("メソッド名", "クラス名")` を使います。`summary()` メソッドは `numeric` 型が別途指定されていないため、"default"となります。

```

1  getS3method("summary", "default")

## function (object, ..., digits, quantile.type = 7)
## {

```

---

2) 逆に特定のクラスで使用可能なメソッドを確認するときは `methods(class = "クラス名")` を入力します。



```

##           ll <- numeric(n)
##           for (i in 1L:n) {
##               ii <- object[[i]]
##               ll[i] <- length(ii)
##               cls <- oldClass(ii)
##               sumry[i, 2L] <- if (length(cls))
##                           cls[1L]
##                           else "-none-"
##               sumry[i, 3L] <- mode(ii)
##           }
##           sumry[, 1L] <- format(as.integer(ll))
##           sumry
##       }
##       else c(Length = length(object), Class = class(object), Mode = mode(object))
##       class(value) <- c("summaryDefault", "table")
##       value
##   }
## <bytecode: 0x7f9d3839ee80>
## <environment: namespace:base>

```

Rにはメソッドがクラス内部で定義されず、別途のメソッド名. クラス名()といった関数として作成されています。そしてジェネリック関数によって一つの関数の「ように」まとまっています。このように、ジェネリック関数経由でメソッドを呼び出すことをメソッド・ディスパッチ (method dispatch) と呼びます。

#### 28.2.4 繙承

クラスの継承 (inheritance) は一つのオブジェクトが2つ以上のクラスを持つ場合、子クラスが親クラスの特徴を継承することを意味します。たとえば、データフレームの拡張版とも言える tibble の場合、複数のクラスを持っています。

```

1 my_tibble <- tibble(X = 1:5, Y = 1:5)
2 class(my_tibble)

```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

この `my_tibble` は `tbl_df` と `tbl`、`data.frame` といった 3 つのクラスを持っており、先に出てきたものが子クラス、後に出てくるものが親クラスです。`tbl` クラスと `data.frame` クラス両方に同じメソッドが定義されている場合、まず子クラスであるメソッド `.tbl()` が実行されます。もし、子クラスにメソッドが定義されていない場合は `tbl` の親クラスである `data.frame` のメソッドが実行されます。`tibble` はデータフレームとは異なるクラスのオブジェクトですが、データフレームと（ほぼ）同じ操作ができるのは、クラスが継承されるからです。クラスの継承ができないと、`tibble` で使える全ての関数（列や行の抽出に使う `[や$なども！]`）を全て一から定義する必要があります<sup>3)</sup>、継承を使うことによってこのような手間を省くことが出来ます。

---

## 28.3 R における OOP

### 28.3.1 オブジェクトに任意のクラスを付ける

クラスを変えるのは簡単です。`class(オブジェクト) <- "新しいクラス名"`だけです。つまり、関数から何かの結果を返す直前にクラスを変更すれば良いです。

```
1 # 方法 1
2 関数名 <- function(...) {
3
4   ...
5
6   class(返すオブジェクト名) <- "任意のクラス名"
7
8   返すオブジェクト名 # return(オブジェクト名) でも OK
9 }
```

たとえば、入力された 2 つのベクトル（`x` と `y`）をリスト構造とし、クラス名を `Score` に

<sup>3)</sup> ただし、実際の場合、`[や $などは` `tbl` または `tbl_df` 用のメソッドが別途用意されています。`data.frame` クラスから継承されているメソッドとしては `dim()` や `t()` などがあります。

するにはどうすれば良いでしょうか。

```
1 Make_Score1 <- function(x, y) {  
2  
3   # result リストに x と y を格納  
4   result <- list(Score1 = x, Score2 = y)  
5  
6   # 以下は attr(result, "class") <- "Score" も可  
7   class(result) <- "Score" # result のクラスを"Score"とする  
8  
9   result           # result を返す  
10 }  
11  
12 My_Score1 <- Make_Score1(x = rnorm(10, 50, 10),  
13                           y = rnorm(10, 50, 10))  
14  
15 My_Score1 # My_Score1 の内部を見る  
  
## $Score1  
##  [1] 62.80613 66.26005 29.80258 30.81355 52.48558 34.56937 39.15835 57.25931  
##  [9] 57.12675 40.96228  
##  
## $Score2  
##  [1] 51.59609 50.16195 55.10895 33.66861 43.26039 42.86678 43.43412 54.30036  
##  [9] 61.47047 55.66079  
##  
## attr(),"class")  
## [1] "Score"  
1 class(My_Score1) # My_Score1 のクラスを表示  
  
## [1] "Score"
```

もう一つの方法は `structure()` 関数を使う方法です。`sturcture()` の第1引数に返す

オブジェクト名を指定し、`class = "クラス名"`引数でクラスを指定します。

```
1 Make_Score2 <- function(x, y) {  
2  
3   # result リストに x と y を格納  
4   result <- list(Score1 = x, Score2 = y)  
5  
6   structure(result, class = "Score") # result を返す  
7 }  
8  
9 My_Score2 <- Make_Score2(x = rnorm(10, 50, 10),  
10                           y = rnorm(10, 50, 10))  
11  
12 My_Score2 # My_Score2 の内部を見る  
  
## $Score1  
##  [1] 46.11811 65.11668 50.23508 46.89002 40.07429 53.31200 53.73675 43.63095  
##  [9] 45.35752 58.34067  
##  
## $Score2  
##  [1] 61.75599 37.46144 39.86834 44.84470 67.12061 52.22692 46.10938 47.62456  
##  [9] 49.54881 51.78699  
##  
## attr(,"class")  
## [1] "Score"  
1 class(My_Score2) # My_Score2 のクラスを表示  
  
## [1] "Score"
```

どれも同じ結果が得られます。

### 28.3.2 メソッドの作り方

#### 28.3.2.1 既に存在する関数名を使う

先ほど作成しました Score クラスのオブジェクトは長さ 2 のリスト構造をしています。これらの要素それぞれの平均値を求める場合は、`mean(My_Score1[[1]])` と `mean(My_Score1[[2]])` を実行する必要があります。なぜなら、`mean()` はベクトルしか計算できないからです。ここでは Score クラスのオブジェクト要素それぞれの平均値を求める関数 `mean()` を作成します。

しかし、問題があります。それは R に `mean()` 関数が既に存在することです。ここで勝手に上書きするのは良くないでしょう。ここで出てくるのがメソッドです。Score クラスのメソッドは「Score クラス専用の関数」であり、通常のベクトルなら R 内蔵の `mean()` 関数を、Score クラスのオブジェクトなら Score のメソッドである `mean()` を実行します。

メソッドの作り方は自作関数と同じです。相違点としては関数名を関数名. クラス名にすることです。Score クラスのメソッドとしての `mean()` 関数を定義する場合、関数名を `mean.Score` とします。

```
1 mean.Score <- function(x) {  
2   print(mean(x$Score1))  
3   print(mean(x$Score2))  
4 }  
5  
6 mean(c(1, 3, 5, 7, 9, 11)) # R 内蔵関数の mean() を使う  
## [1] 6  
1 mean(My_Score1) # Score クラスのメソッドである mean() を使う
```

```
## [1] 47.12439  
## [1] 49.15285
```

`mean(c(1, 3, 5, 7, 9, 11))` は引数が numeric 型ベクトルであるため、既存の `mean()` 関数が使用されます。一方、`mean(My_Score1)` は引数が Score クラスであ

るため、`mean.Score()` が使用されます。このように `mean_Score()` のような別途の関数を作る必要なく、既存の関数名が利用できます。実際、`methods(mean)` を実行すると、`Score` クラスのメソッドとして `mean()` 関数が用意されたことを確認できます。

```
1 methods(mean)

## [1] mean.Date      mean.default    mean.difftime   mean.POSIXct
## [5] mean.POSIXlt   mean.quosure*  mean.Score     mean.vctrs_vctr*
## see '?methods' for accessing help and source code
```

### 28.3.2.2 新しい関数を作る

もし、新しい関数名を使用し、その関数が様々なクラスに対応するとしましょう。今回は `Cat` というクラスを作ってみましょう。`Cat` クラスの内部は長さ 1 のリストで、要素の名前は `Name` とし、ここには長さ 1 の character 型ベクトルが入ります。この `Cat` クラスを作成する関数を `Make_Cat()` とします。

```
1 Make_Cat <- function(name) {
2
3   # result リストに x を格納
4   result <- list(Name = name)
5
6   structure(result, class = "Cat") # result を返す
7 }
8
9 My_Cat <- Make_Cat(name = "矢内")
10 My_Cat
```

```
## $Name
## [1] "矢内"
##
## attr(,"class")
## [1] "Cat"
```

```
1  class(My_Cat)
```

```
## [1] "Cat"
```

続いて、Cat クラスに使う `my_func()` を作成します。`my_func()` はそもそも存在しない関数ですので、普通に `my_func <- function()` で作成可能です。この関数は Cat の `Name` の後ろに": にゃーにゃー"を付けて出力する関数です。実際にやってみましょう。

```
1  my_func <- function(name) {  
2      print(paste0(name>Name, ": にゃーにゃー"))  
3  }  
4  
5  my_func(My_Cat)
```

```
## [1] "矢内: にゃーにゃー"
```

しかし、`my_func()` を Cat クラス以外にも使いたい場合はどうすればいいでしょうか。普通に `my_func.` クラス名 `()` で良いでしょうか。確かにそうですが、その前に一つの手順が必要です。それは、`my_func()` をジェネリック関数として定義することです。この関数そのものは関数として機能はしませんが、「これから `my_func()` がいろんなクラスのメソッドとして使われるぞ」と予め決めてくれます。ジェネリック関数を作成しないと関数名. クラス名は定義できません。そこで使うのが `UseMethod()` です。第一引数はメソッド名、第二引数は任意の引数ですが、通常、`x` が使われます。また、第二の引数は省略可能で、`UseMethod("メソッド名")` でも動きます。

```
1  my_func <- function(x) {  
2      UseMethod("my_func", x)  
3  }
```

これからは `my_func.` クラス名 `()` の関数を作るだけです。まず、Score 型オブジェクトに対してはそれぞれの要素の平均値を出力するとします。

```
1  my_func.Score <- function(x) {  
2      print(mean(x$Score1))  
3      print(mean(x$Score2))
```

```
4  }
5
6 my_func.Cat <- function(cat) {
7   print(paste0(cat$name, "にゃーにゃー"))
8 }
```

1 methods(my\_func)

```
## [1] my_func.Cat   my_func.Score
## see '?methods' for accessing help and source code
```

my\_func() 関数は Score と Cat といった 2 種類のクラスで使われることが確認できます。それでは問題なく作動するかを確認してみましょう。My\_Score1 と My\_Cat を、それぞれ my\_func() に渡します。

```
1 my_func(My_Score1)
```

```
## [1] 47.12439
## [1] 49.15285
```

```
1 my_func(My_Cat)
```

```
## [1] "矢内: にゃーにゃー"
```

同じ関数名でも、オブジェクトのクラスによって異なる処理が行われることが分かります。

### 28.3.3 検証用関数を作る

この作業は必須ではありませんが、今後、自分でパッケージ等を作ることになったら重要なかも知れません。

最初の例でもお見せしましたが、R では事後的にクラスを変更することができます。強制的にクラスを変更した場合、そのクラスに属するメソッドを使うことができますが、エラーが生じてしまうでしょう。例えば、任意の character 型ベクトル My\_Cat2 を作成し、Cat クラスを付与してみましょう。

```
1 My_Cat2 <- "宋"  
2 class(My_Cat2) <- "Cat"  
3 class(My_Cat2)  
  
## [1] "Cat"
```

My\_Cat2 のクラスは Cat であるため、my\_func.Cat() メソッドが使えます。しかし、my\_func.Cat() 仕組みを見る限り、うまく作動しないでしょう。

```
1 my_func(My_Cat2)  
  
## Error: $ operator is invalid for atomic vectors
```

間違った動作をするよりは、エラーが出て中断される方が良いですし、これで問題ないかも知れません。しかし、可能であれば、引数として使われたオブジェクトが、Cat クラスか否かを厳密にチェックする機能があれば良いでしょう。カプセル化されている場合、クラスの定義時にデータの構造が厳密に定義されているため、このような手続きの必要性はありませんが、カプセル化ができない R の S3 クラスでは検証用関数 (Validator) が必要です。

それでは Cat クラスの特徴をいくつか考えてみましょう。

- オブジェクトの中には Name という要素のみがある。
- Name は長さ 1 の Character 型ベクトルである。

以上の条件を全て満たしていればメソッドを実行し、一つでも満たさない場合はメソッドの実行を中止します。それでは検証用関数 Validation\_Cat() を作ってみましょう。

```
1 Validation_Cat <- function(x) {  
2   Message <- "正しい Cat クラスではありません。"  
3  
4   if (length(x) != 1) {  
5     stop(Message)  
6   } else if (is.null(names(x))) {  
7     stop(Message)  
8   } else if (names(x) != "Name") {
```

```
9     stop(Message)
10 } else if (length(x$Name) != 1 | class(x$Name) != "character") {
11     stop(Message)
12 }
13 }
```

この検証用関数を `my_func.Cat()` の最初に入れておきましょう。

```
1 my_func.Cat <- function(cat) {
2   Validation_Cat(cat)
3
4   print(paste0(cat$Name, "にゃーにゃー"))
5 }
```

それでは `My_Cat` と `My_Cat2` に対して `my_func()` メソッドを実行してみます。

```
1 my_func(My_Cat)
## [1] "矢内: にゃーにゃー"
1 my_func(My_Cat2)
```

## Error in Validation\_Cat(cat): 正しい Cat クラスではありません。

関数を実行する前に与えられたオブジェクトが正しい Cat クラスか否かが判断され、パスされた場合のみ、メソッドが実行されることが分かります。もし、あるクラスで使用可能なメソッドが一つだけでしたら、検証用関数はメソッド内に直接書き込んでも良いですが、2つ以上のメソッドを持つ場合は別途の検証用関数を作成しておきましょう。

---

## 28.4 例題

ここでは 2 つの numeric 型ベクトルとそのベクトル名入力し、相関係数を求める `My_Cor()` 関数を作ってみます。単に相関係数を求めるだけなら `cor()` や `cor.test()` があるので、いくつかの機能も追加してみましょう。

たとえば、「1日当たりゲーム時間」と「身長」といった2つのnumeric型ベクトルをそれぞれxとyで入力し、x\_nameとy\_nameで各ベクトルの名前も指定します。また、入力されたデータを用いて相関係数とその信頼区間を求めます。これらのデータはリスト型として格納されますが、クラスを" My\_Cor\_Object"とします。以下はその例です。

```

1 pacman::p_load(tidyverse)
2
3 set.seed(19861008)
4 Cor_Obj <- My_Cor(x      = rnorm(20, 2, 0.5),
5                     y      = rnorm(20, 165, 6),
6                     x_name = "1日当たりゲーム時間",
7                     y_name = "身長")
8
9 class(Cor_Obj)
## [1] "My_Cor_Object"

```

このCor\_Objの構造をstr()で確認してみます。

```

1 str(Cor_Obj)
## List of 4
## $ data      :'data.frame': 20 obs. of 2 variables:
##   ..$ x: num [1:20] 1.96 2.2 1.34 2.29 2.08 ...
##   ..$ y: num [1:20] 165 159 155 158 164 ...
## $ var_name: chr [1:2] "1日当たりゲーム時間" "身長"
## $ cor      : Named num 0.277
##   ..- attr(*, "names")= chr "cor"
## $ cor_ci   : num [1:2] -0.188 0.641
##   ..- attr(*, "conf.level")= num 0.95
## - attr(*, "class")= chr "My_Cor_Object"

```

Cor\_Objには元のデータがデータフレームとして格納され(\$data)、それぞれの変数名(\$var\_name)、相関係数(\$cor)、相関係数の95%信頼区間(\$cor\_ci)がCor\_Objの中に入っています。本質的にはリスト型のデータ構造ですが、クラス名がMy\_Cor\_Object

になっているだけです。

この `My_Cor_Object` クラスには 3 つのメソッド(専用関数)が用意されており、`print()`、`summary()`、`plot()` です。`print()` と `summary()` は同じ関数で、`x` と `y` の平均値、そして相関係数と信頼区間を出力します。`plot()` は散布図と相関係数を出力します。実際の例を見てみましょう。

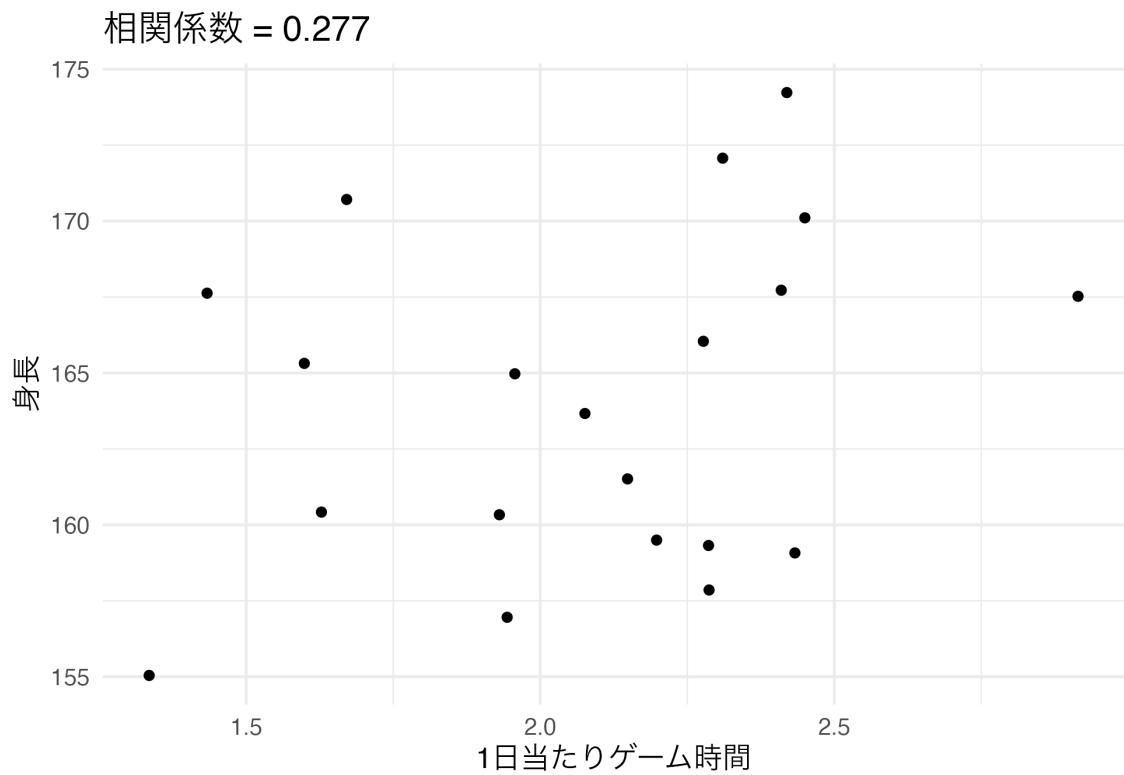
```
1 print(Cor_Obj)

## 1 日当たりゲーム時間の平均値: 2.085
## 身長の平均値: 164.001
## 相関係数: 0.277 [-0.188, 0.641]

1 summary(Cor_Obj) # summary() は print() と同じ

## 1 日当たりゲーム時間の平均値: 2.085
## 身長の平均値: 164.001
## 相関係数: 0.277 [-0.188, 0.641]

1 plot(Cor_Obj)
```



既存の `cor.test()` で作成される "htest" クラスに比べ、"My\_Cor\_Object" クラスは各変数の平均値が名前と一緒に表示され、`plot()` で簡単に散布図が作成できる大変便利なクラスです。この `My_Cor_Object` クラスとそのメソッドの構造を図示したものが図 28.2 です。

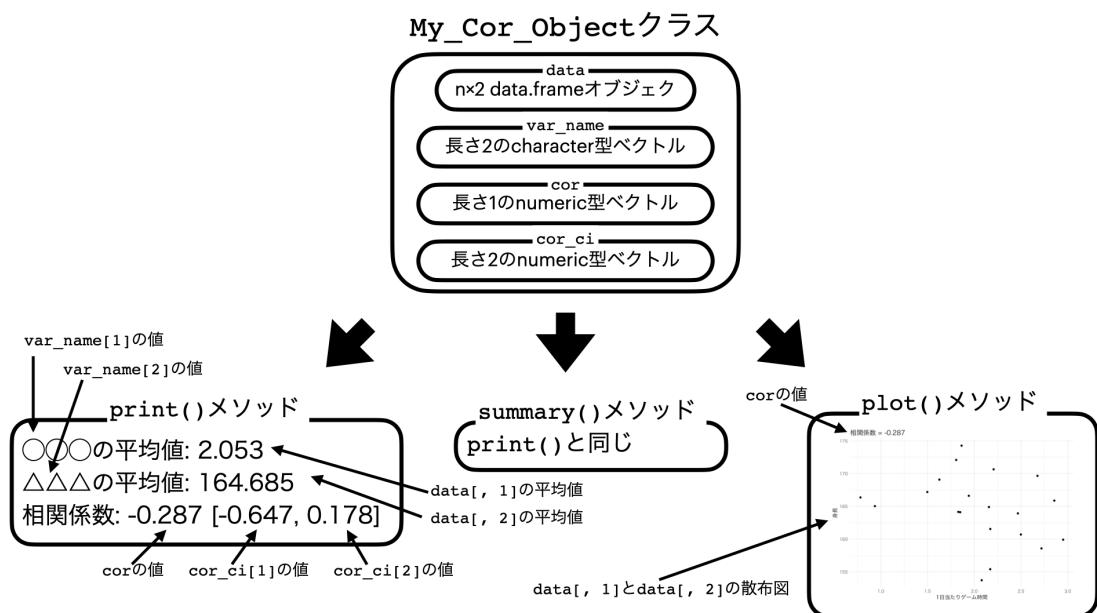


図 28.2: クラスの構造

それでは一つずつ作っていきましょう。まずは、"My\_Cor\_Object"クラスのオブジェクトを作成する `My_Cor()` 関数からです。

```

1  My_Cor <- function(x, y, x_name, y_name) {
2      if (!is.numeric(x) | !is.numeric(y)) {
3          stop("x または y が numeric 型ではありません。")
4      }
5      if (length(x) != length(y)) {
6          stop("x と y は同じ長さでなければなりません。")
7      }
8      if (!is.character(x_name) | !is.character(y_name)) {
9          stop("x_name または y_name が character 型ではありません。")
10     }
11
12     data      <- data.frame(x = x, y = y)
13     var_name <- c(x_name, y_name)
14     cor       <- cor.test(x, y)$estimate

```

```
15     cor_ci  <- cor.test(x, y)$conf.int
16
17     result  <- structure(list(data      = data,
18                               var_name = var_name,
19                               cor       = cor,
20                               cor_ci   = cor_ci),
21                               class = "My_Cor_Object")
22
23     result
24 }
```

最初の部分は入力されたデータが My\_Cor\_Object クラスに適した構造か否かを判断します。これは最初から想定外の My\_Cor\_Object クラスのオブジェクトが作成されることを防ぐことが目的です。もちろん、R (S3) の性質上、事後的にクラスを変更することができるですから、検証用関数も作っておきます。ここでは以下の条件を検証します。

- data という要素が存在し、2 列である。
- var\_name という要素が存在し、長さ 2 の character 型ベクトルである。
- cor という要素が存在し、長さ 1 の numeric 型ベクトルである。
- cor\_ci という要素が存在し、長さ 2 の numeric 型ベクトルである。

```
1 Validation <- function (x) {
2   UseMethod("Validation", x)
3 }
4
5 Validation.My_Cor_Object <- function(x) {
6   Message <- "正しい My_Cor_Object クラスではございません。"
7
8   if (is.null(x$data) | ncol(x$data) != 2) {
9     stop(Message)
10   }
11   if (is.null(x$var_name) | length(x$var_name) != 2 | class(x$var_name) != "charac"
12     stop(Message)
```

```
13     }
14     if (is.null(x$cor) | length(x$cor) != 1 | class(x$cor) != "numeric") {
15       stop(Message)
16     }
17     if (is.null(x$cor_ci) | length(x$cor_ci) != 2 | class(x$cor_ci) != "numeric") {
18       stop(Message)
19     }
20 }
```

ここでは `Validation()` をジェネリック関数として使用しました。自分が開発するパッケージで複数のクラスを提供する予定でしたら、このようなやり方が良いでしょう。

検証用関数は細かく書いた方が良いです。以上の `Validation()` もより細かくことが出来ます。たとえば、`data` が 2 列か否かを判定するだけでなく、`numeric` 型であるかなども判定した方が良いでしょう。

つづいて、`My_Cor_Object` クラス用の `print()` 関数（メソッド）を作成します。

```
1 print.My_Cor_Object <- function(data) {
2   Validation(data)
3
4   cat(sprintf("%s の平均値: %.3f\n",
5           data$var_name[1],
6           mean(data$data$x)))
7   cat(sprintf("%s の平均値: %.3f\n",
8           data$var_name[2],
9           mean(data$data$y)))
10  cat(sprintf("相関係数: %.3f [% .3f, %.3f]\n",
11           data$cor,
12           data$cor_ci[1],
13           data$cor_ci[2]))
14 }
```

次は、`summary()` メソッドですが、これは `print()` と同じ機能をする関数です。この場

合、UseMethod("メソッド名") を使うと、指定したメソッドを使うことになります。

```
1 summary.My_Cor_Object <- function(data) {  
2   UseMethod("print")  
3 }
```

最後は plot() メソッドです。

```
1 plot.My_Cor_Object <- function(data) {  
2   Validation(data)  
3  
4   data$data %>%  
5     ggplot(aes(x = x, y = y)) +  
6     geom_point() +  
7     labs(x = data$var_name[1], y = data$var_name[2]) +  
8     ggtitle(sprintf("相関係数 = %.3f", data[["cor"]])) +  
9     theme_minimal(base_size = 12)  
10 }
```

これで相関係数の計算および可視化が便利になる関数群が完成しました。R パッケージの開発はこれよりも数倍も複雑ですが、本記事の内容はプログラミングをより効率的に行うための入り口となります。

## 第 29 章

# モンテカルロシミュレーション

### 29.1 モンテカルロシミュレーションとは

```
1 pacman::p_load(tidyverse, ggforce)
```

モンテカルロ法 (Monte Carlo method) とは無作為に抽出された乱数を用い、数値計算やシミュレーションを行う手法を意味します。モンテカルロはヨーロッパのモナコ公国内の一つの地区であり、カジノで有名なところです。カジノではサイコロやルーレット、無作為に配られたカードなど、乱数が頻繁に使われることからこのように名付けられました。

モンテカルロ法を用いたシミュレーションがモンテカルロ・シミュレーションです。計算があまりにも複雑だったり、実質的に代数で解が得られない解析学上の問題などに強みを持つ手法です。一部、明快な例（共役事前分布が存在するなど）を除き、事後分布の計算が非常に複雑（実質、不可能）だと知られていたベイズ統計学もモンテカルロ法（マルコフ連鎖モンテカルロ法; MCMC）によって、ようやく使えるものになったなど、今になってモンテカルロ法は非常に広く用いられています。

モンテカルロ法から得られた結果には常に誤差が存在します。とりわけ、生成された乱数が少ない（＝試行回数が少ない）場合、この誤差は大きくなります。しかし、近年はパソコンの性能が飛躍的に発達しているため、かなり小さな誤差で、つまりより正確な結果が得られるようになりました。

以下ではまず、モンテカルロ法を理解するために必須知識である乱数生成について解説します。具体的には乱数生成のアルゴリズムでなく、Rで乱数を生成する方法について紹介します。続いて、モンテカルロ法を用いたシミュレーションの例として誕生日問題、モンティ・ホール問題、円周率の計算、ブートストラップ法を紹介します。

## 29.2 乱数生成

### 29.2.1 `sample()` によるサンプリング

無作為に値を抽出する方法には2つが考えられます。一つは**値の集合から**無作為に値を抽出する方法、もう一つは正規分布などの**確率分布から**値を抽出する方法です。ここではまず `sample()` 関数を用い、値の集合から無作為に値を抽出する方法について説明します。

```
1 sample(x = 値の集合ベクトル, size = 抽出の回数,  
2       replace = 復元抽出の有無, prob = 各要素が抽出される確率)
```

`replace` は復元抽出の有無を指定する引数であり、既定値は `FALSE`、つまり非復元抽出がデフォルトとなっています。これは一度抽出された要素は、二度と抽出されないことを意味します。値の集合が  $\{0, 1\}$  で、5個の値を抽出する (`=size` が `x` の長さより大きい) ならば、`replace` は必ず `TRUE` に設定する必要があります。抽選などは非復元抽出であるため、`replace` 引数は省略可能です。しかし、対数の法則やブートストラップなどは復元抽出を仮定している場合が多く、意識的に `replace` 関数は指定することを推奨します。`prob` は各要素が抽出される確率を意味し、`x` の実引数と同じ長さの numeric 型ベクトルを指定します。`prob` の実引数の総和は 1 であることが望ましいですが、総和が 1 でない場合、自動的に総和が 1 になるよう正則化を行います。つまり、`c(1, 3)` は `c(0.25, 0.75)` と同じことを意味します。

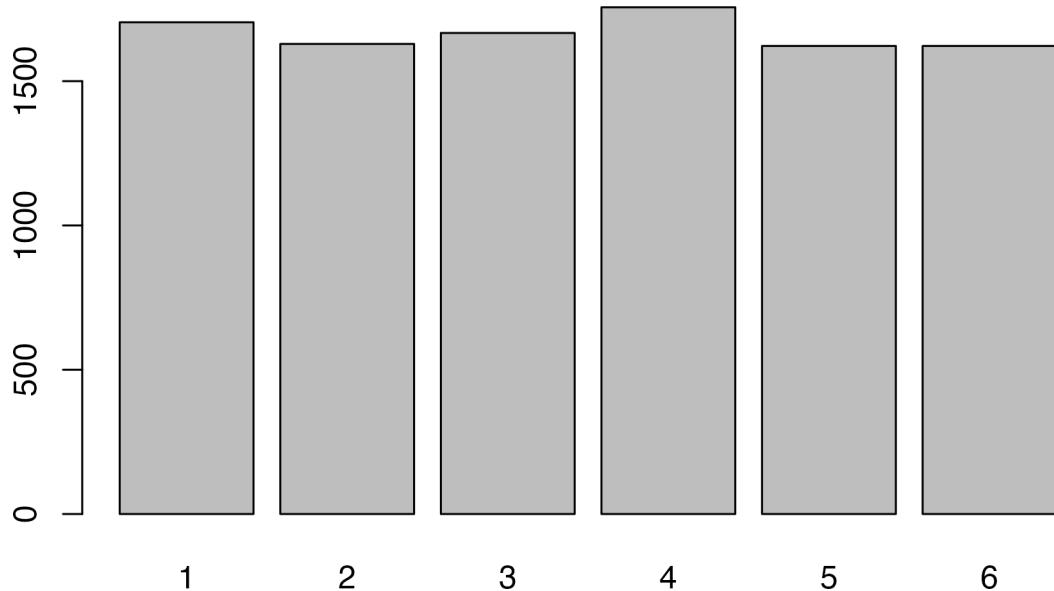
サイコロを3回振るコードを書くなら、値の集合 (`x`) は `c(1, 2, 3, 4, 5, 6)`、または `1:6` で、抽出の回数 (`size`) は 3 となります。また、一回出た目も抽出される可能性があるため、復元抽出を行う必要があります (`replace = TRUE`)。そして各目が出る確率は  $1/6$  ですが、各値が抽出される確率が等しい場合、省略可能です。

```
1 # この場合、prob 引数は省略可能
2 sample(1:6, 3, replace = TRUE, prob = rep(1/6, 6))
```

```
## [1] 5 2 6
```

今回はサイコロを 1 万回振り、それぞれの目が出た回数を棒グラフとして示してみます。無作為に抽出された値であれば、各目が出る回数は等しいはずです。ベクトルに対して `table()` 関数を使うと、各要素が出現した回数が output され、このオブジェクトを `barplot()` 関数に渡すと棒グラフを作成することができます。

```
1 Dice_vec <- sample(1:6, 10^4, replace = TRUE)
2 table(Dice_vec) %>% barplot()
```



1 から 6 までの目が出た回数がほぼ同じであることが確認できます。この 6 つの棒の高さがすべて同じになることはありませんが（そもそも 1 万を 6 で割ったら余りが出来ますね）、ほぼ同じ割合であることから、疑似乱数とは言え、シミュレーション用としては十分でしょう。

### 29.2.2 確率分布からの乱数制制

| 関数名                             | 確率分布        | パラメーター                          |
|---------------------------------|-------------|---------------------------------|
| <code>rbeta()</code>            | ベータ分布       | <code>n, shape1, shape2</code>  |
| <code>rbinom()</code>           | 二項分布        | <code>n, size, prob</code>      |
| <code>rcauchy()</code>          | コーシー分布      | <code>n, location, scale</code> |
| <code>rchisq()</code>           | $\chi^2$ 分布 | <code>n, df</code>              |
| <code>rexp()</code>             | 指数分布        | <code>n, rate</code>            |
| <code>rf()</code>               | $F$ 分布      | <code>n, df1, df2</code>        |
| <code>rgamma()</code>           | ガンマ分布       | <code>n, shape, scale</code>    |
| <code>rgeom()</code>            | 幾何分布        | <code>n, prob</code>            |
| <code>rhyper()</code>           | 超幾何分布       | <code>nn, m, n, k</code>        |
| <code>rlnorm()</code>           | 対数正規分布      | <code>n, meanlog, sdlog</code>  |
| <code>rmultinom()</code>        | 多項分布        | <code>n, size, prob</code>      |
| <code>rnbinom()</code>          | 負の二項分布      | <code>n, size, prob</code>      |
| <code>rnorm()</code>            | 正規分布        | <code>n, mean, sd</code>        |
| <code>rpois()</code>            | ポアソン分布      | <code>n, lambda</code>          |
| <code>rt()</code>               | $t$ 分布      | <code>n, df</code>              |
| <code>runif()</code>            | 一様分布        | <code>n, min, max</code>        |
| <code>rweibull()</code>         | ワイブル分布      | <code>n, shape, scale</code>    |
| <code>mvtnorm::rmvnorm()</code> | 多変量正規分布     | <code>n, mean, sigma</code>     |

以上の表に掲載されているパラメーター以外にも指定可能なパラメーターがあるため、詳細は各関数のヘルプを参照してください。たとえば、ガンマ分布の場合、`rate` で、負の二項分布の場合、`mu` で分布の形状を指定することができます。また、多変量正規分布の乱数を抽出するには`{mvtnorm}`パッケージの `rmvnorm()` を使いますが、ここでの `mean` は数値型ベクトル、`sigma` は行列構造の分散共分散行列を使います<sup>1)</sup>。

<sup>1)</sup> `sigma` は数値型ベクトル (`mean` の実引数の長さと同じ長さ) で指定することも可能ですが、この場合、共分散は 0 になります。

### 29.2.3 シードについて

特定の分布から乱数を抽出する場合、当たり前ですが、抽出の度に値が変わります。たとえば、平均 0、標準偏差 1 の正規分布（標準正規分布）から 5 つの値を抽出し、小数点 3 術に丸める作業を 3 回繰り返しちゃう。

```
1 rnorm(5) %>% round(3)

## [1] -0.057  0.100 -1.630 -1.394  0.942

1 rnorm(5) %>% round(3)

## [1] -0.643  1.951  0.585 -0.372 -1.410

1 rnorm(5) %>% round(3)

## [1]  0.261 -1.237 -0.152  0.801 -0.889
```

このように、抽出の度に結果が変わります。1 回きりのシミュレーションではこれで問題ないでしょうが、同じシミュレーションから同じ結果を得るために、乱数を固定する必要があります。そこで使うのがシード（seed）です。シードが同じなら抽出される乱数は同じ値を取ります。シードの指定は `set.seed(numeric 型スカラー)` です。たとえば、シードを 19861008 にし、同じ作業をやってみましょう。

```
1 set.seed(19861008)

2 rnorm(5) %>% round(3)

## [1] -0.086  0.396 -1.330  0.574  0.152

1 rnorm(5) %>% round(3)

## [1]  0.555  0.620 -1.133  0.572  0.900
```

シードを指定しても 2 つのベクトルは異なる値を取りますが、もう一度シードを指定してから乱数抽出をしてみましょう。

```

1 set.seed(19861008)
2 rnorm(5) %>% round(3)

## [1] -0.086  0.396 -1.330  0.574  0.152

```

先ほどのコードでシードを指定した直後に抽出した乱数と同じ乱数が得られました。モンテカルロ・シミュレーションにおいて乱数は非常に重要ですが、これはシミュレーションの度に異なる結果が得られることを意味します。つまり、自分が書いたコードから100%同じ結果が得られないだけでなく、自分も同じ結果を再現できないことを意味します。この場合、シードを指定すると乱数が固定され、シミュレーション結果の再現ができるようになります。

一つ注意すべき点は乱数を固定した後、複数回抽出を繰り返す場合、その順番も固定されるという点です。たとえば、シードを固定せずにもう一回5つの値を抽出してみましょう。

```

1 rnorm(5) %>% round(3)

## [1]  0.555  0.620 -1.133  0.572  0.900

```

この結果は先ほどシード指定後、2回目の抽出結果と同じ結果となります。結果を再現するという点では大きな問題はないはずですが、仕様を理解しておくことは重要でしょう。

---

### 29.3 例 1: 誕生日問題

まず簡単な例として誕生日問題 (birthday problem) をシミュレーションで確認してみましょう。誕生日問題とは「何人いれば、その中に誕生日が同じ2人以上がいる確率が50%を超えるか。」といった問題です。1年を365日で考えると(2月29日生まれの皆さん、すみません...)、366人がいれば確実に(=100%)同じ誕生日の人が2人以上いることになりますね。100%ではなく、50%まで基準を下げるならその半分である183人は必要じゃないかと思うかも知れません。しかし、実はたった23人が集まれば、その中で同じ誕生日の人が2人以上いる確率が50%になります。誕生日のパラドックスとも呼ばれ

るものですが、これをシミュレーションで確認してたいと思います。

学生の数を `n_student` とし、1 から 365 までの公差 1 の等差数列から、`n_student` 個の値を復元抽出します。

```
1 n_student <- 30 # 学生数
2 # 「1 から 365 までの公差 1 の等差数列」から n_student 個の値を復元抽出
3 Birth_vec <- sample(1:365, n_student, replace = TRUE)
4
5 Birth_vec

## [1] 340 122 230 148 12 136 87 46 16 310 104 351 137 53 361 65 106 361 108
## [20] 281 299 67 342 4 338 88 156 254 192 37
```

このベクトルの中で重複する要素があるかどうかを確認するには `duplicated()` 関数を使います。ある要素が他の要素と重複するなら `TRUE` が、なければ `FALSE` が表示されます。

```
1 duplicated(Birth_vec)

## [1] FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE
```

ただし、一つでも `TRUE` が含まれていれば、同じ誕生日の人が二人以上はいるということとなるので、更に `any()` 関数を使います。`any()` 内の条件文において、一つでも `TRUE` があれば返り値は `TRUE` となり、全て `FALSE` なら `FALSE` を返す関数です。

```
1 any(duplicated(Birth_vec))
```

```
## [1] TRUE
```

以上の作業を 100 回繰り返す場合、試行回数が 100 回となります。この場合、`TRUE` の結果が出る試行は何回でしょうか。

```
1 n_student <- 10 # 学生数
2 n_trials <- 100 # 試行回数
```

```

3
4 # 結果を格納する空ベクトルを用意する
5 Result_vec <- rep(NA, n_trials)
6
7 # 反復処理
8 for (i in 1:n_trials) {
9     Birth_vec <- sample(1:365, n_student, replace = TRUE)
10    Result_vec[i] <- any(duplicated(Birth_vec))
11 }
12
13 Result_vec

## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
## [25] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
## [49] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE
## [85] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [97] FALSE FALSE FALSE FALSE

```

100回の試行の中でTRUEが出たのは10回ですね。割合で考えると10%です。この試行回数を無限にすると確率として解釈できますが、無限回繰り返しは世の中が終わるまでやっても終わりませんね。ただし、十分に多い試行回数、たとえば1万回程度繰り返すと確率に近似できるでしょう。

```

1 n_student <- 10      # 学生数
2 n_trials <- 10000 # 試行回数
3
4 # 結果を格納する空ベクトルを用意する
5 Result_vec <- rep(NA, n_trials)
6

```

```
7  # 反復処理
8  for (i in 1:n_trials) {
9      Birth_vec      <- sample(1:365, n_student, replace = TRUE)
10     Result_vec[i] <- any(duplicated(Birth_vec))
11 }
12
13 sum(Result_vec)
## [1] 1147
```

1万回の試行から TRUE が出た回数は 1147 回であり、11.5% ですね。つまり、人が 10 人集まれば誕生日が同じ人が 2 人以上いる確率は約 11.5% ということになります。実はこの確率は厳密に計算可能であり、理論的な確率は約 11.7% です。シミュレーションから得られた結果が理論値にかなり近似していることが分かります。

今回は試行回数は 100 に固定し、学生数を 2 から 100 まで調整しながら同じ誕生日の人が 2 人以上いる割合を計算してみましょう。以下では割合を計算する関数 `Birthday_Func()` を作成し、`{purrr}` の `map_dbl()` 関数を使用して反復処理を行います。関数の作成は第 11 章を、`{purrr}` の使い方については第 27 章を参照してください。

```
1 # 割合を計算する関数を作成する
2 Birthday_Func <- function (n, n_trials) {
3
4     # 各試行の結果を格納する空ベクトルを用意する。
5     Result_vec <- rep(NA, n_trials)
6
7     # n_trials 回だけ {} 内コードを繰り返す。
8     for (i in 1:n_trials) {
9         # 1:365 から n 個の値を復元抽出
10        Birth_vec      <- sample(1:365, n, replace = TRUE)
11        # 重複する要素があるかをチェックし、結果ベクトルの i 番目に格納
12        Result_vec[i] <- any(duplicated(Birth_vec))
13    }
```

```
14
15      # 各試行結果が格納されたベクトルから TRUE の割合を返す
16      mean(Result_vec)
17  }
18
19  # 学生数を Students 列に格納したデータフレーム (tibble) を作成
20  Prob_df <- tibble(Students = 2:100)
21
22  # Students 列の値に応じて Birthday_Func() を実行
23  Prob_df <- Prob_df %>%
24      mutate(Probs = map_dbl(Students, ~Birthday_Func(.x, 100)))
```

{purrr}関数を使わずに、for() 文を使用した例は以下のようになります。

```
1 # {purrr}を使わない方法
2 Prob_df <- tibble(Students = 2:100,
3                     Probs      = NA)
4
5 for (i in 1:nrow(Prob_df)) {
6
7     Result_vec <- rep(NA, n_trials)
8
9     for (j in 1:n_trials) {
10         Birth_vec      <- sample(1:365, Prob_df$Students[i], replace = TRUE)
11         Result_vec[j] <- any(duplicated(Birth_vec))
12     }
13
14     Prob_df$Probs[i] <- mean(Result_vec)
15 }
```

結果を確認してみましょう。

```
1 Prob_df
```

```
## # A tibble: 99 x 2
##   Students Probs
##       <int>  <dbl>
## 1       2     0.01
## 2       3     0.02
## 3       4     0.06
## 4       5     0.01
## 5       6     0.02
## 6       7     0.08
## 7       8     0.06
## 8       9     0.09
## 9      10    0.12
## 10      11    0.18
## # ... with 89 more rows
```

学生数が何人いれば、同じ誕生日の人が 2 人以上いる割合が 50% になるのでしょうか。割合が 0.4 以上、0.6 以下の行を抽出してみましょう。

```
1 Prob_df %>%
2   filter(Probs >= 0.4 & Probs <= 0.6)
```

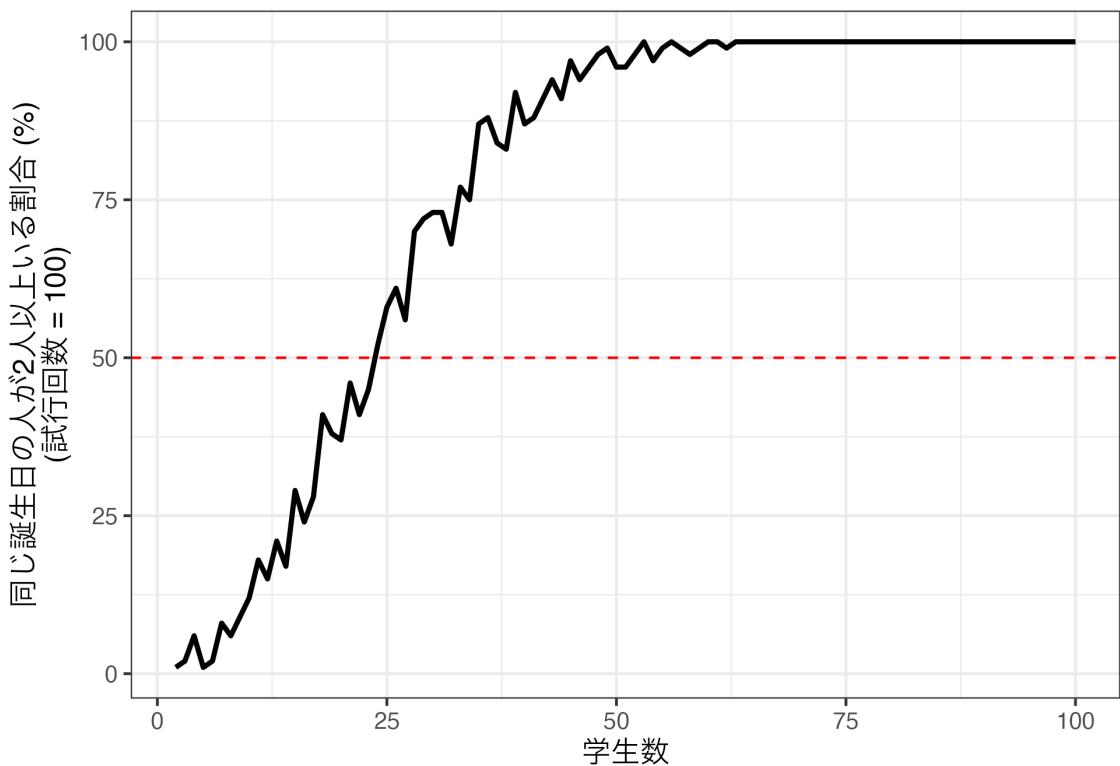
```
## # A tibble: 7 x 2
##   Students Probs
##       <int>  <dbl>
## 1       18    0.41
## 2       21    0.46
## 3       22    0.41
## 4       23    0.45
## 5       24    0.52
## 6       25    0.58
## 7       27    0.56
```

大体23人前後ですかね。試行回数を増やせばもう少し厳密に検証出来るかも知れませんが、とりあえず以上の結果を可視化してみましょう。

```

1 Prob_df %>%
2   ggplot() +
3   geom_line(aes(x = Students, y = Probs * 100), size = 1) +
4   geom_hline(yintercept = 50, color = "red", linetype = 2) +
5   labs(x = "学生数",
6        y = "同じ誕生日の人が2人以上いる割合 (%)\\n(試行回数 = 100)") +
7   theme_bw(base_size = 12)

```



非常に直感に反する結果かも知れませんが、50人程度いれば、**ほぼ**確実に同じ誕生日の人が2人以上いることが分かります。この誕生日問題は以下のように解くことができます。人が $n$ 人いる場合、同じ誕生日の人が2人以上いる確率 $p(n)$ は、

$$p(n) = 1 - \frac{365!}{365^n (365 - n)!}$$

!は階乗を意味し、5!は $5 \times 4 \times 3 \times 2 \times 1$ を意味します。R では `factorial()` 関数を使います。それでは  $p(50)$  はいくらでしょうか。

```
1  1 - factorial(365) / (365^50 * factorial(365 - 50))
```

```
## [1] NaN
```

あらら、NaN がでましたね。つまり、計算不可です。実際、`factorial(365)` だけでも計算結果は `Inf` が出ます。むろん、実際に無限ではありませんが、非常に大きい数値ということです。以上の式を計算可能な式に変形すると以下のようになります。

$$p(n) = 1 - \frac{n! \times_{365} C_n}{365^n}$$

$C$  は二項係数を意味し  ${}_n C_k$  は  $\frac{n!}{k!(n-k)!}$  です。R では `choose(n, k)` で計算可能です。

```
1  1 - ((factorial(50) * choose(365, 50)) / 365^50)
```

```
## [1] 0.9703736
```

結果は 0.9703736 です。つまり、人が 50 人いれば同じ誕生日の人が 2 人以上いる確率は約 97% ということです。それでは、以上の式を関数化し、 $p(22)$  と  $p(23)$  を計算してみましょう。

```
1 Birth_Expect <- function (n) {
2   1 - ((factorial(n) * choose(365, n)) / 365^n)
3 }
4
5 Birth_Expect(22)
```

```
## [1] 0.4756953
```

```
1 Birth_Expect(23)
```

```
## [1] 0.5072972
```

確率が 50% を超える人数は 23 人であることが分かります。先ほどのデータフレーム (`Prob_df`) にこの理論値を `Expect` という名の列として追加してみましょう。

```
1 Prob_df <- Prob_df %>%
2   mutate(Expect = map_dbl(Students, ~Birth_Expect(.x)))
3
4 Prob_df
```

```
## # A tibble: 99 x 3
##   Students Probs   Expect
##       <int>  <dbl>   <dbl>
## 1       2    0.01 0.00274
## 2       3    0.02 0.00820
## 3       4    0.06 0.0164
## 4       5    0.01 0.0271
## 5       6    0.02 0.0405
## 6       7    0.08 0.0562
## 7       8    0.06 0.0743
## 8       9    0.09 0.0946
## 9      10    0.12 0.117
## 10     11    0.18 0.141
## # ... with 89 more rows
```

これは人間にとっては読みやすい表ですが、可視化まで考えると、tidyなデータといは言えません。したがって、pivot\_longer()関数を使用してtidyなデータに整形します。{tidyverse}パッケージの使い方は第15章を参照してください。

```
1 Prob_df2 <- Prob_df %>%
2   pivot_longer(cols      = Probs:Expect,
3                 names_to  = "Type",
4                 values_to = "Prob")
5
6 Prob_df2
```

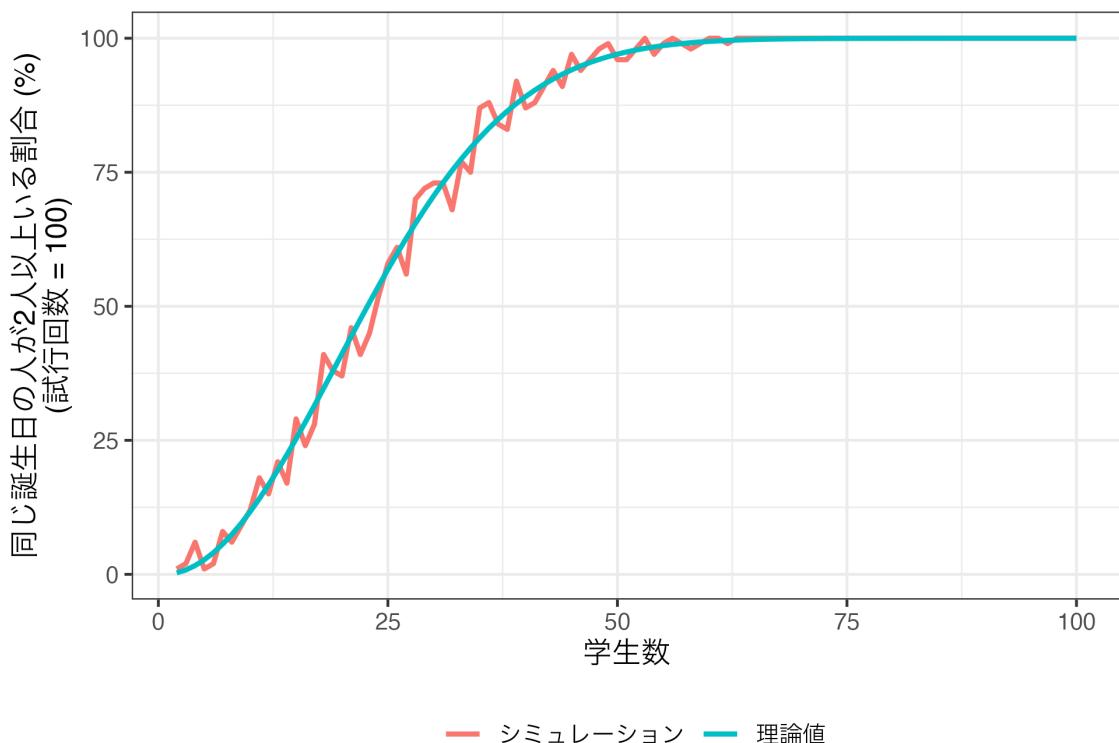
  

```
## # A tibble: 198 x 3
##   Students Type      Prob
##       <int> <chr>    <dbl>
## 1       2 "Type" 0.01
## 2       2 "Prob" 0.00274
## 3       3 "Type" 0.02
## 4       3 "Prob" 0.00820
## 5       4 "Type" 0.06
## 6       4 "Prob" 0.0164
## 7       5 "Type" 0.01
## 8       5 "Prob" 0.0271
## 9       6 "Type" 0.02
## 10      6 "Prob" 0.0405
## 11      7 "Type" 0.08
## 12      7 "Prob" 0.0562
## 13      8 "Type" 0.06
## 14      8 "Prob" 0.0743
## 15      9 "Type" 0.09
## 16      9 "Prob" 0.0946
## 17     10 "Type" 0.12
## 18     10 "Prob" 0.117
## 19     11 "Type" 0.18
## 20     11 "Prob" 0.141
## # ... with 188 more rows
```

```
##      <int> <chr>     <dbl>
## 1      2 Probs  0.01
## 2      2 Expect 0.00274
## 3      3 Probs  0.02
## 4      3 Expect 0.00820
## 5      4 Probs  0.06
## 6      4 Expect 0.0164
## 7      5 Probs  0.01
## 8      5 Expect 0.0271
## 9      6 Probs  0.02
## 10     6 Expect 0.0405
## # ... with 188 more rows
```

こちらのデータを使用し、シミュレーションから得られた結果と理論値を折れ線グラフで出力してみましょう。

```
1 Prob_df2 %>%
2   mutate(Type = ifelse(Type == "Probs", "シミュレーション", "理論値")) %>%
3   ggplot() +
4   geom_line(aes(x = Students, y = Prob * 100, color = Type), size = 1) +
5   labs(x      = "学生数",
6        y      = "同じ誕生日の人が 2 人以上いる割合 (%)\\n(試行回数 = 100)",
7        color = "") +
8   theme_bw(base_size = 12) +
9   theme(legend.position = "bottom")
```



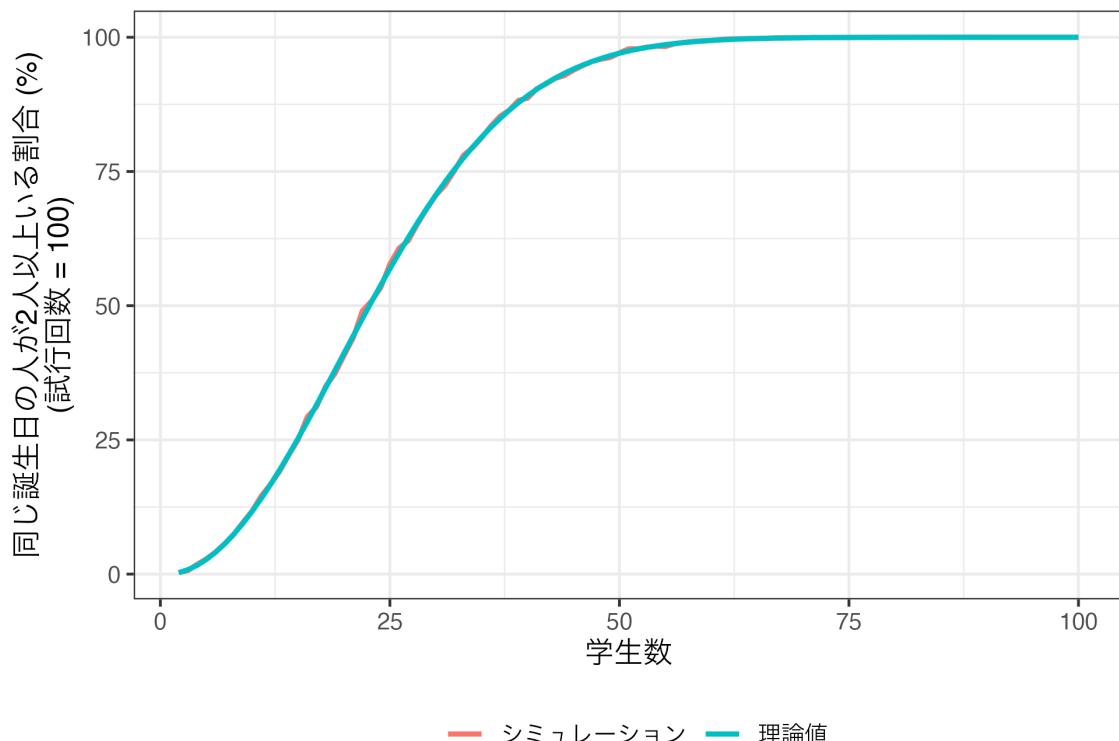
最後に以上の作業を試行回数 10000 としてもう一回やってみましょう。

```

1 # 学生数を Students 列に格納したデータフレーム (tibble) を作成
2 Prob_df <- tibble(Students = 2:100)
3
4 # Students 列の値に応じて Birthday_Func() を実行
5 Prob_df <- Prob_df %>%
6   mutate(Simul = map_dbl(Students, ~Birthday_Func(.x, 10000)),
7         Expect = map_dbl(Students, ~Birth_Expect(.x)))
8
9 Prob_df %>%
10   pivot_longer(cols      = Simul:Expect,
11                 names_to  = "Type",
12                 values_to = "Prob") %>%
13   mutate(Type = ifelse(Type == "Simul", "シミュレーション", "理論値")) %>%
14   ggplot() +
15   geom_line(aes(x = Students, y = Prob * 100, color = Type), size = 1) +

```

```
16     labs(x      = "学生数",  
17         y      = "同じ誕生日の人が 2 人以上いる割合 (%)\\n(試行回数 = 100)",  
18         color = "") +  
19         theme_bw(base_size = 12) +  
20         theme(legend.position = "bottom")
```



モンテカルロ・シミュレーションから得られた割合と理論上の確率が非常に近似していることが分かります。

## 29.4 例 2: モンティ・ホール問題

抽出される乱数が必ずしも数値である必要はありません。たとえば、コイン投げの表と裏、ポーカーで配られたカードなど、数値以外の乱数もあり得ます。ここでは「A と B、C」から一つを選ぶ例として、モンティ・ホール問題をモンテカルロ法で解いてみま

しょう。

モンティ・ホール問題はアメリカのテレビ番組「Let's make a deal」の中のゲームであり、この番組の司会者の名前がモンティ・ホール (Monty Hall) さんです。このゲームのルールは非常にシンプルです。

1. 3つのドアがあり、1つのドアの裏に商品（車）がある。残りの2つは外れ（ヤギ）である。
2. 参加者はドアを選択する。
3. 司会者が残りのドア2つの中で商品がないドアを開けて中身を見せる。
4. ここで参加者はドアの選択を変える機会が与えられる。

直観的に考えて、司会者が外れのドアを1つ教えてくれたなら、自分が選んだドアを含め、残りの2つのドアの1つに絶対に商品があります。直感的に考えてみると、当たる確率は半々であって、変えても、変えなくても当たる確率は同じだと考えられます。詳細は Wikipedia などを参照してください。この問題を巡る論争とともに紹介されていてなかなか面白いです。

結論から申しますと選択を変えた方が、変えなかった場合より当たる確率が2倍になります。これは条件付き確率とベイズの定理を用いることで数学的に説明できますが、ここではあえてモンテカルロ法で調べてみたいと思います。シミュレーションの具体的な手順は以下の通りです。

1. 結果を格納する長さ1万の空ベクトルを2つ用意する。（`Switch_Yes` と `Switch_No`）。
2. `i` の初期値を1とする。
3. 当たり（車）の位置を A, B, C の中から無作為に1つ決め、`Car_Position` に格納する。
4. 最初の選択肢を A, B, C の中から無作為に1つ決め、`Choice` に格納する。
5. 選択肢を変更した場合の結果を `Switch_Yes` の `i` 番目の要素として格納する。
  1. 当たりの位置と最初の選択肢が同じなら (`Switch_Yes == Choice`)、結果は外れ（ヤギ）
  2. 当たりの位置と最初の選択肢が同じでないなら (`Switch_Yes != Choice`)、結果は当たり（車）
6. 選択肢を変更しなかった場合の結果を `Switch_No` の `i` 番目の要素として格納

する。

1. 当たりの位置と最初の選択肢が同じなら (`Switch_Yes == Choice`)、結果は当たり (車)
2. 当たりの位置と最初の選択肢が同じでないなら (`Switch_Yes != Choice`)、結果は外れ (ヤギ)
7. `i` の値を 1 増やし、3 に戻る。
8. 3~7 の手順を 1 万回繰り返す。

以上の手順をコードで書くと以下のようになります。

```
1 Switch_Yes <- rep(NA, 10000)
2 Switch_No  <- rep(NA, 10000)
3
4 set.seed(19861009)
5 for (i in 1:10000) {
6   Car_Position <- sample(c("A", "B", "C"), 1)
7   Choice       <- sample(c("A", "B", "C"), 1)
8
9   Switch_Yes[i] <- ifelse(Car_Position == Choice, "Goat", "Car")
10  Switch_No[i]  <- ifelse(Car_Position == Choice, "Car", "Goat")
11 }
12
13 table(Switch_Yes)
```

```
## Switch_Yes
## Car Goat
## 6737 3263
```

```
1 table(Switch_No)
```

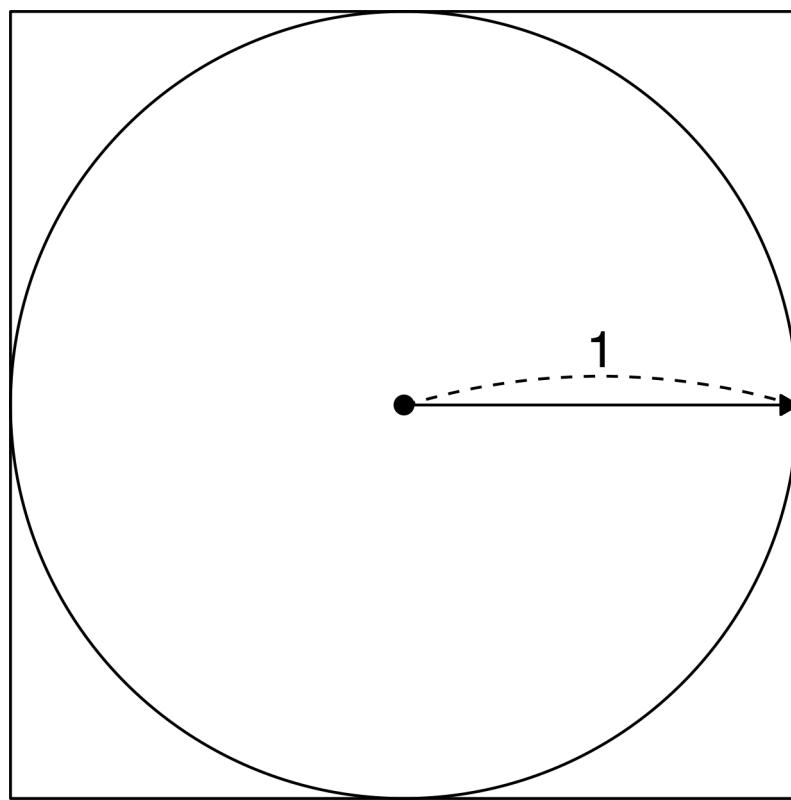
```
## Switch_No
## Car Goat
## 3263 6737
```

選択肢を変更し、車を獲得した回数は 10000 回中、6737 であり、約 67% です。つま

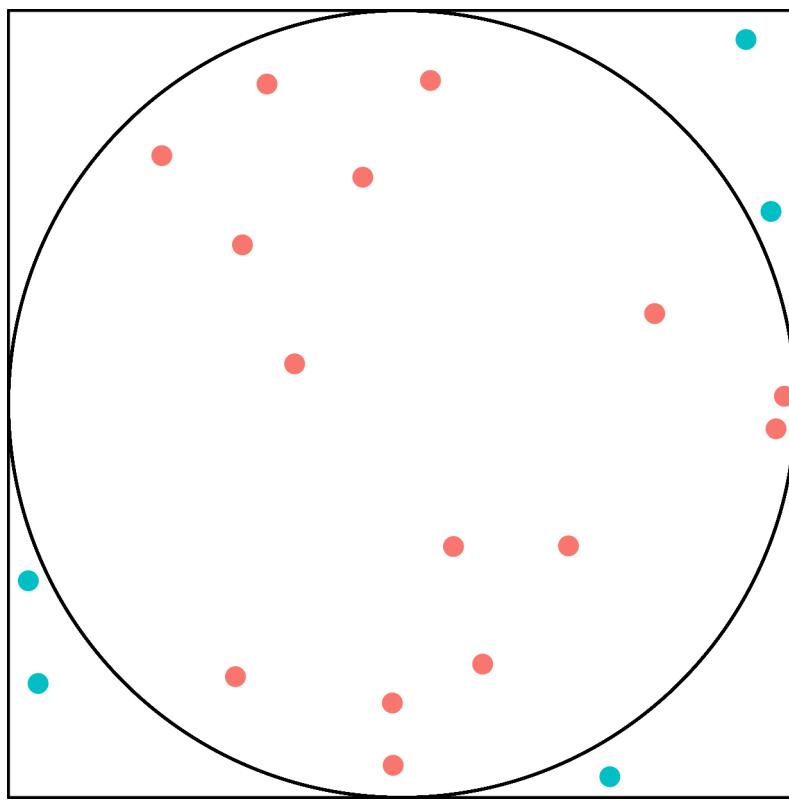
り、選択肢を変えた方が、変えなかった場合に比べ、車が当たる確率が約2倍高いことを意味します。むろん、車よりもヤギが重宝される地域に住んでいるなら、あえて選択肢を変えず、ヤギを狙った方が良いかも知れません。

## 29.5 応用 1: 円周率の計算

今回はもう一つの例として、円周率( $\pi$ )の計算を紹介したいと思います。 $\pi$ は無理数であるため、厳密な計算は出来ませんが、モンテカルロ・シミュレーションである程度近似できます。たとえば、半径1( $r = 1$ )の円を考えてみましょう。



円の面積は  $r^2\pi$  であるため、この円の面積は  $\pi$  です。また、四角形は辺の長さが2の正四角形ですから面積は4です。続いて、四角形の範囲内の点を付けます。無作為に20個を付けてみます。



20 個点のうち、円の外側にあるのは 5 個、円の内側は 15 個です。つまり、75% の点が円内にあることを意味します。点の位置は無作為ですので、もし円の大きさが  $\pi$  であれば、点が円内に入る確率は  $\frac{\pi}{4}$  です。今回の例だと  $\frac{\pi}{4} = 0.75$  であるため、 $\pi = 0.75 \times 4 = 3$  となります。実際の円周率は 3.141593...なので、そこそこ近似できていますね。

それではこれを実際にやってみましょう。今回は 20 個の点ではなく、100 個にしてみましょう。まず、100 個の点を無作為に抽出します。

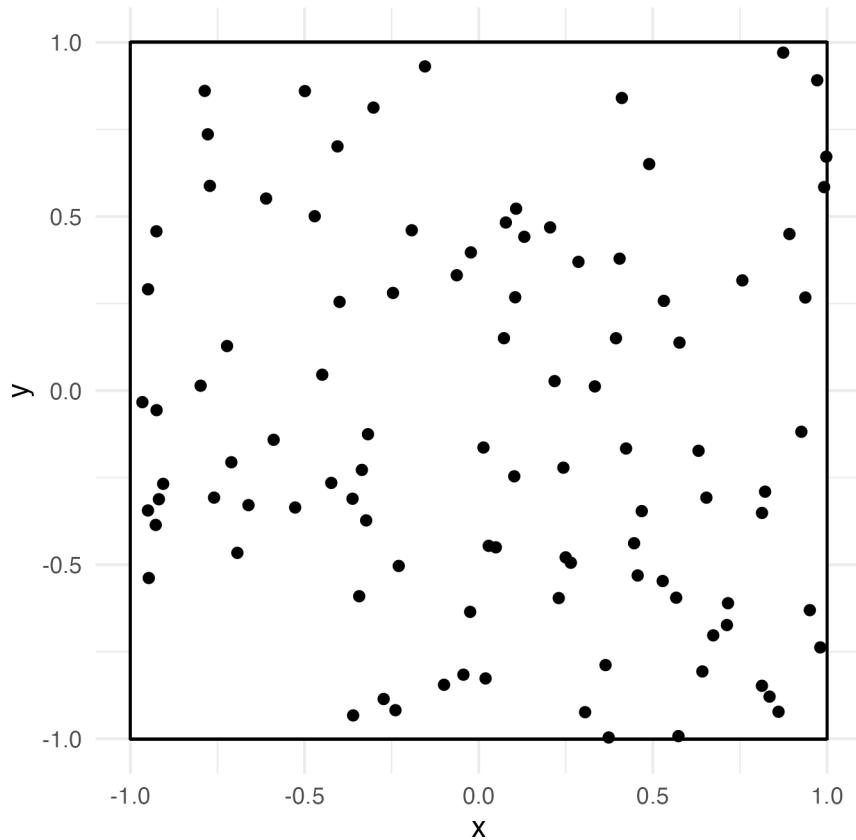
```
1 set.seed(19861009)
2 pi_df <- tibble(x = runif(100, -1, 1),
3                   y = runif(100, -1, 1))
4
5 pi_df
```

```
## # A tibble: 100 x 2
##       x     y
##   <dbl> <dbl>
```

```
## 1 -0.950 -0.344
## 2 0.0724 0.150
## 3 0.874 0.971
## 4 0.938 0.267
## 5 0.205 0.469
## 6 -0.343 -0.590
## 7 -0.0245 -0.635
## 8 -0.273 -0.886
## 9 -0.405 0.701
## 10 0.528 -0.547
## # ... with 90 more rows
```

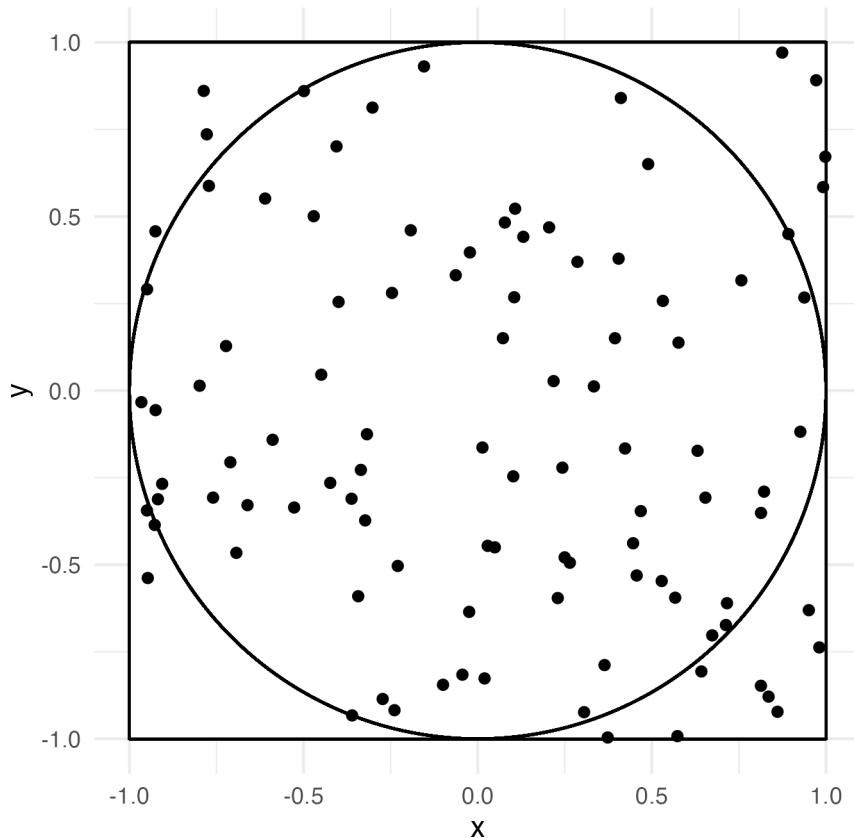
まず、各辺の長さが 2 の正四角形と `pi_df` で生成した 100 個の点をプロットします。四角形を描くときには `geom_rect()` 異幾オブジェクトを使用します。マッピングは四角形の左下の座標 (`xmin` と `ymin`)、右上の座標 (`xmax` と `ymax`) に行います。今回は原点が  $(0, 0)$  の半径 1 の円に接する四角形ですから、左下の座標は  $(-1, -1)$ 、右上の座標は  $(1, 1)$  となります。

```
1 pi_df %>%
2   ggplot() +
3   geom_rect(aes(xmin = -1, ymin = -1, xmax = 1, ymax = 1),
4             fill = "white", color = "black") +
5   geom_point(aes(x = x, y = y)) +
6   coord_fixed(ratio = 1) +
7   theme_minimal()
```



ここに円を追加してみましょう。円を描くときには{ggforce}パッケージのgeom\_circle()幾何オブジェクトを使用します。マッピングは円の原点(x0とy0)、円の半径(r)です。原点は(0, 0)で半径は1の円を重ねます。

```
1 pi_df %>%
2   ggplot() +
3   geom_rect(aes(xmin = -1, ymin = -1, xmax = 1, ymax = 1),
4             fill = "white", color = "black") +
5   geom_circle(aes(x0 = 0, y0 = 0, r = 1)) +
6   geom_point(aes(x = x, y = y)) +
7   coord_fixed(ratio = 1) +
8   theme_minimal()
```



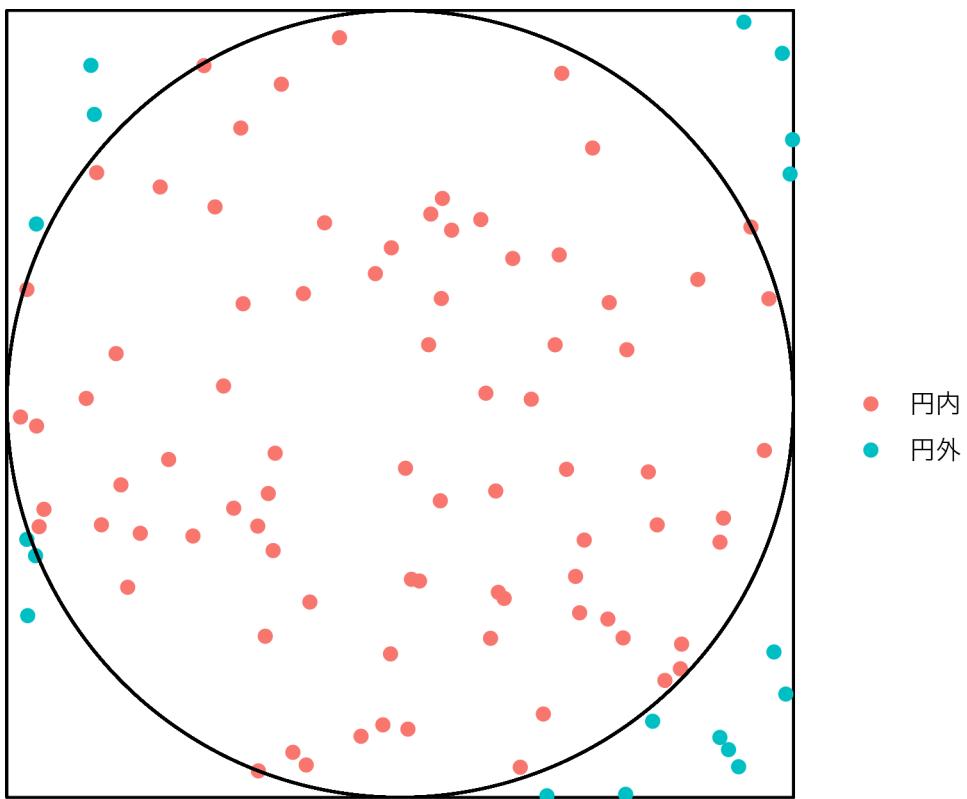
これだけだと読みづらいので、円の中か外かで点の色分けをしてみましょう。そのためには各点が円内に入っているかどうかを判定した変数 `in_circle` を追加します。点  $(x, y)$  が、原点が  $(x', y')$ 、かつ半径  $r$  の円内に入っている場合、 $(x - x')^2 + (y - y')^2 < r^2$  が成立します。今回は原点が  $(0, 0)$  で、半径が 1 であるため、 $x^2 + y^2 < 1$  か否かを判定します。この条件を満たしているかどうかを示す `in_circle` という変数を追加します。

```
1 pi_df <- pi_df %>%
2   mutate(in_circle = if_else(x^2 + y^2 < 1, "円内", "円外"))
```

散布図レイヤー (`geom_point()`) 内に `color` を `in_circle` 変数でマッピングします。

```
1 pi_df %>%
2   ggplot() +
3   geom_rect(aes(xmin = -1, ymin = -1, xmax = 1, ymax = 1),
4             fill = "white", color = "black") +
5   # 円の内側か外側かで色分け
```

```
6     geom_point(aes(x = x, y = y, color = in_circle), size = 2) +
7     geom_circle(aes(x0 = 0, y0 = 0, r = 1)) +
8     labs(x = "X", y = "Y", color = "") +
9     coord_fixed(ratio = 1) +
10    theme_void(base_size = 12)
```



実際に円内の点と円外の点の個数を数えてみましょう。

```
1 pi_df %>%
2   group_by(in_circle) %>%
3   summarise(N = n())
```

```
## # A tibble: 2 x 2
##   in_circle     N
##   <chr>     <int>
## 1 円内         82
```

```
## 2 円外          18
```

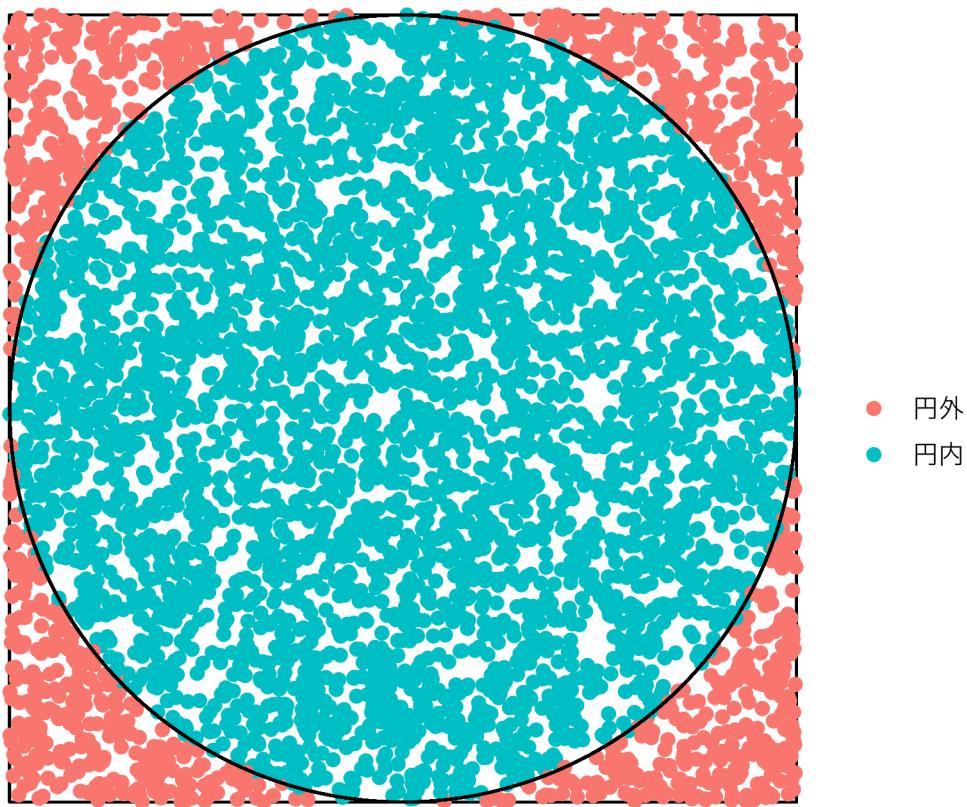
円内の点は82個、円外の点は18ですね。つまり、 $\frac{\pi}{4} = 0.82$ であり、 $\pi = 0.82 \times 4 = 3.28$ です。

```
1 82 / 100 * 4
```

```
## [1] 3.28
```

今回は100個の点で円周率の近似値を計算しましたが、点の数を増やすとより正確な近似値が得られます。以下の例は10000個の点から得られた円周率の例です。

```
1 set.seed(19861009)
2 pi_df2 <- tibble(x = runif(5000, -1, 1),
3                     y = runif(5000, -1, 1))
4
5 pi_df2 <- pi_df2 %>%
6     mutate(in_circle = if_else(x^2 + y^2 < 1, "円内", "円外"))
7
8 pi_df2 %>%
9     ggplot() +
10    geom_rect(aes(xmin = -1, ymin = -1, xmax = 1, ymax = 1),
11               fill = "white", color = "black") +
12    geom_point(aes(x = x, y = y, color = in_circle), size = 2) +
13    geom_circle(aes(x0 = 0, y0 = 0, r = 1)) +
14    labs(x = "X", y = "Y", color = "") +
15    coord_fixed(ratio = 1) +
16    theme_void(base_size = 12)
```



```
1 pi_df2 %>%
2   group_by(in_circle) %>%
3   summarise(N = n())
```

```
## # A tibble: 2 x 2
##   in_circle     N
##   <chr>     <int>
## 1 円外      1081
## 2 円内      3919
```

円内の点は 3919 個、円外の点は 1081 ですね。この結果から円周率を計算してみましょう。

```
1 3919 / 5000 * 4
```

```
## [1] 3.1352
```

より実際の円周率に近い値が得られました。

## 29.6 応用 2: ブートストラップ法

最後に、様々な分析から得られた統計量の不確実性を計算する方法の一つであるブートストラップ法 (bootstrapping) について説明します。たとえば、平均値の差分の検定 (*t* 検定) 差、回帰分析における標準誤差などは R の `t.test()`、`lm()` 関数を使用すれば瞬時に計算できます。こちらの標準誤差はデータを与えられれば、常に同じ値が得られるもので、何らかの計算式があります。しかし、世の中にはモデルが複雑すぎて、統計量の標準誤差がうまく計算できないケースもあります。そこで登場するのがブートストラップ法を用いると、不確実性の近似値が得られます。ブートストラップ法は Efron [1979] が提案した以来、データ分析において広く使われています。ブートストラップ法については優れた教科書が多くあるので詳細な説明は割愛し、以下ではブートストラップ法の簡単な例を紹介します。

ブートストラップにおいて重要なのは元のデータセットのサンプルサイズを  $n$  とした場合、復元抽出を用いてサンプルサイズ  $n$  のデータセットをもう一度構築することです。そしてその平均値を計算します。これらの手順を 5000 回繰り返せば、5000 個の平均値が得られます。この 5000 個の平均値の平均値は元のデータセットの平均値に近似し、5000 個の平均値の標準偏差は元のデータセットの平均値の標準誤差に近似できます。これは本当でしょうか。実際にやってみましょう。

まずは、長さ 100 のベクトルを 2 つ作成します。この 2 つのベクトルは平均値が 0.7、0.9、標準偏差 1 の正規分布に従うとします。

$$\text{Data1} \sim \text{Normal}(\mu = 0.7, \sigma = 1)$$
$$\text{Data2} \sim \text{Normal}(\mu = 0.9, \sigma = 1)$$

2 つの標本の平均値の差分は約 0.2 です。この差分の不確実性はいくらでしょうか。ここでは主に使われる平均値の差の検定 (*t* 検定) をやってみましょう。今は標準誤差が同じ 2 つの分布から得られた標本であるため、等分散を仮定する *t* 検定を行います (`var.equal = TRUE` を追加)。

```
1 set.seed(19861009)
2 Data1 <- rnorm(100, 0.7, 1)
3 Data2 <- rnorm(100, 0.9, 1)
4
5 ttest_result <- t.test(Data1, Data2, var.equal = TRUE)
6
7 ttest_result

##  
##  Two Sample t-test  
##  
## data: Data1 and Data2  
## t = -2.1707, df = 198, p-value = 0.03114  
## alternative hypothesis: true difference in means is not equal to 0  
## 95 percent confidence interval:  
## -0.58521800 -0.02807095  
## sample estimates:  
## mean of x mean of y  
## 0.6912143 0.9978588
```

平均値の差分の不確実性 (=標準誤差) は `ttest_result$stderr` で抽出可能であり、今回は約 0.141 です。標準誤差の値さえ分かれば、検定統計量も、信頼区間も、 $p$  値も計算できるため、重要なのはやはり標準誤差でしょう。以下では試行回数は 1 万のブートストラップ法で標準誤差の近似値を計算してみます。

1. 結果を格納する長さ 1 万の空ベクトル `Result_vec` を作成する。
2. `i` の初期値を 1 と設定する。
3. `Data1` から 100 個 (= `Data1` の大きさ) の値を無作為抽出 (復元抽出) し、`Sample1` に格納する。
4. `Data2` から 100 個 (= `Data2` の大きさ) の値を無作為抽出 (復元抽出) し、`Sample2` に格納する。
5. `Result_vec` の `i` 番目の位置に `Sample1` の平均値と `Sample2` の平均値の差分を格納する。

6.  $i$  を1増加させる。
7. 3~6の手順を1万回繰り返す。

```

1 n_trials  <- 10000
2 Result_vec <- rep(NA, n_trials)
3
4 for (i in 1:n_trials) {
5   Sample1 <- sample(Data1, 100, replace = TRUE)
6   Sample2 <- sample(Data2, 100, replace = TRUE)
7
8   Result_vec[i] <- mean(Sample1) - mean(Sample2)
9 }
```

`Result_vec` には平均値の差分が10000個格納されており、これらの値の平均値をブートストラップ推定量 (bootstrap estimate) と呼ぶとします。

```

1 mean(Result_vec) # ブートストラップ推定量 (平均値の差分の平均値)
## [1] -0.3045563
```

実際の平均値の差分は-0.3066445、ブートストラップ推定量は-0.3045563であるため、その差は0.0020881です。これをブートストラップ推定量のバイアスと呼びます。

ただし、我々に興味があるのは平均値の差分ではありません。平均値の差分はブートストラップ法を用いなくても普通に計算できるからです。ここで重要なのは平均値の差分の不確実性、つまり標準誤差でしょう。ブートストラップ推定量の標準誤差は `Result_vec` の標準偏差を計算するだけで十分です。

```

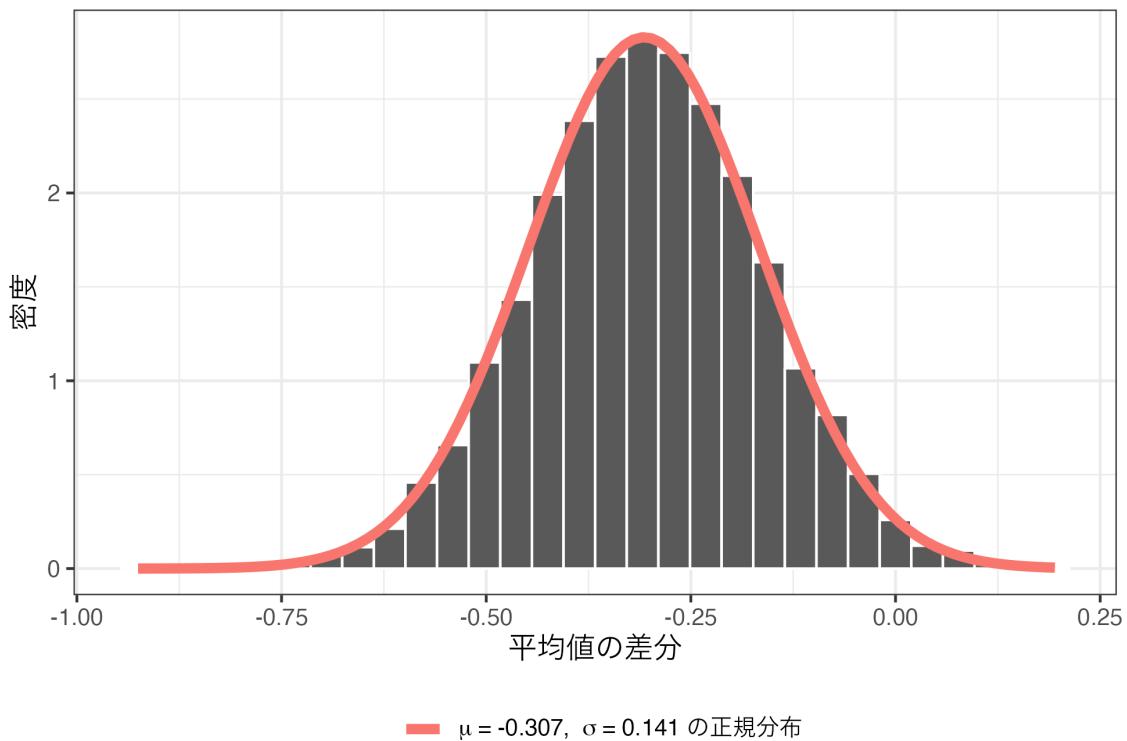
1 sd(Result_vec) # ブートストラップ推定量の標準誤差
## [1] 0.1404988
1 # 平均値の差分の平均値 / 平均値の差分の標準偏差
2 mean(Result_vec) / sd(Result_vec)
## [1] -2.167679
```

```
1 # p 値
2 (1 - pt(abs(mean(Result_vec) / sd(Result_vec)), df = 198)) * 2
## [1] 0.03137631

1 # 95% 信頼区間
2 quantile(Result_vec, c(0.025, 0.975))
##          2.5%      97.5%
## -0.58084484 -0.02999534
```

`Result_vec` のヒストグラムと  $\mu = -0.307$ 、 $\sigma = 0.141$  の正規分布を重ねて見る (-0.307 と 0.141 は  $t$  検定から得られた数値)。

```
1 enframe(Result_vec) %>%
2   ggplot() +
3   geom_histogram(aes(x = value, y = ..density..), color = "white") +
4   stat_function(aes(color = "Density"),
5                 size = 2, fun = dnorm, n = 101,
6                 args = list(mean = -0.307, sd = 0.141)) +
7   scale_color_discrete(labels = c("Density" = expression(paste(mu ~ " = -0.307, " ~ sig
8   labs(x = "平均値の差分", y = "密度", color = ""))
9   theme_bw(base_size = 12) +
10  theme(legend.position = "bottom")
```



*t* 検定とブートストラップ法の比較

## 第 30 章

# スクレイピング



## 第 31 章

### API



# 演習問題の回答

結果は載せておりません。自分で結果を確認しましょう。

## 31.1 基本的な操作

```

1 (myVec1 <- c(3, 9, 10, 8, 3, 5, 8)) # 問 1
2 myVec1[c(2, 4, 6)]                      # 問 2
3 sum(myVec1)                             # 問 3
4 sum(myVec1[(myVec1 %% 2 == 1)])        # 問 4
5 (myVec2 <- c(1, 2, 3, 4, 3, 2, 1))    # 問 5
6 (myVec3 <- myVec1 + myVec2)            # 問 6
7 myVec3[myVec3 < 10]                     # 問 7
8 myVec4 <- 1:100                         # 問 8
9 sum(myVec4^2)                           # 問 9
10 sum((myVec4[myVec4 %% 2 == 1])^2)       # 問 10

```

## 31.2 データの構造

### 31.2.1 ベクトル

```

1 # 問 1 1 から 10 までの公差 1 の等差数列を作成し、myVec1 と名付けよ。
2 myVec1 <- 1:10
3

```

```

4  # 問 2 myVec1 の長さを求めよ。
5  length(myVec1)
6
7  # 問 3 myVec1 から偶数のみを抽出せよ
8  myVec1[myVec1 %% 2 == 0]
9
10 # 問 4 myVec1 を myVec2 という名でコピーし、myVec2 の偶数を全て 0 に置換せよ。
11 myVec2 <- myVec1
12 myVec2[myVec2 %% 2 == 0] <- 0
13
14 # 問 5 myVec1 の全要素から 1 を引し、myVec3 と名付けよ。
15 myVec3 <- myVec1 - 1
16
17 # 問 6 myVec1 の奇数番目の要素には 1 を、偶数番目の要素には 2 を足し、myVec4 と名付けよ。
18 myVec4 <- myVec1 + c(1, 2)
19
20 # 問 7 myVec4 から myVec1 を引け。
21 myVec4 - myVec1

```

### 31.2.2 行列

問 1 以下のような 2 つの行列を作成せよ。

$$\text{myMat1} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \text{myMat2} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

```

1  # myMat1: byrow = を指定する場合
2  myMat1 <- matrix(1:6, nrow = 2, byrow = TRUE)
3  # myMat1: byrow = を指定しない場合
4  myMat1 <- matrix(c(1, 4, 2, 5, 3, 6), nrow = 2)
5

```

```
6 # myMat2: byrow = を指定する場合
7 myMat2 <- matrix(c(1, 4, 7, 2, 5, 8, 3, 6, 9), nrow = 3, byrow = TRUE)
8 # myMat2: byrow = を指定しない場合
9 myMat2 <- matrix(1:9, nrow = 3)
```

問 2 myMat1 と myMat2 の掛け算を行い、myMat3 と名付けよ。

```
1 myMat3 <- myMat1 %*% myMat2
```

問 3 連立方程式の解を求めよ。

```
1 # 問 3-1
2 myMat4 <- matrix(c(3, -1, 2,
3 1, 2, 3,
4 2, -1, -1), nrow = 3, byrow = TRUE)
5 myMat5 <- matrix(c(12, 11, 2), nrow = 3)
6
7 # 問 3-2
8 solve(myMat4)
9
10 # 問 3-3
11 myMat6 <- solve(myMat4) %*% myMat5
12
13 # 問 3-4
14 myMat4 %*% myMat6
```

### 31.2.3 データフレーム

```
1 # 問 1. 以下のようなデータフレームを作成し、myDF1 と名付けよ。
2 myDF1 <- data.frame(
3   ID      = 1:10,
4   Name   = c("Australia", "China", "Iran", "Iraq", "Japan",
5   "Qatar", "Saudi Arabia", "South Korea", "Syria", "UAE"),
```

```
6      Rank  = c(42, 76, 33, 70, 28, 55, 67, 40, 79, 71),
7      Socre = c(1457, 1323, 1489, 1344, 1500,
8                      1396, 1351, 1464, 1314, 1334)
9
10
11 # 問 2. myDF1 から Name 列を抽出せよ。
12 myDF1>Name
13
14 # 問 3. myDF1 の Name 列から 3 番目の要素を抽出せよ。
15 myDF1>Name[3]
16
17 # 問 4. myDF1 の 3 行目を抽出せよ。
18 myDF1[3, ]
19
20 # 問 5. FIFA_Women.csv を tibble 型として読み込み、myTbl1 と名付けよ。
21 myTbl1 <- read_csv("Data/FIFA_Women.csv")
22
23 # 問 6. myTbl1 の Rank 列を抽出し、それぞれの要素が 20 より小さいかを判定せよ。
24 myTbl1$Rank < 20
25
26 # 問 7. Rank が 20 より小さい国名を抽出せよ。
27 myTbl1$Team[myTbl1$Rank < 20]
28
29 # 問 8. myTbl1 からランキングが 20 位以内の行を抽出せよ。
30 myTbl1[myTbl1$Rank < 20, ]
```

### 31.3 R プログラミングの基礎

#### 問 1

while() を使う場合

```
1 Trial <- 1
2 Total <- 0
3
4 while (Total != 15) {
5   Dice <- sample(1:6, 3, replace = TRUE)
6   Total <- sum(Dice)
7
8   print(paste0(Trial, "目のサイコロ投げの結果: ",
9                 Dice[1], ", ", Dice[2], ", ", Dice[3],
10                " (合計: ", Total, ")"))
11
12   Trial <- Trial + 1
13 }
```

`for()` を使う場合

```
1 for (Trial in 1:10000) {
2   Dice <- sample(1:6, 3, replace = TRUE)
3   Total <- sum(Dice)
4
5   print(paste0(Trial, "目のサイコロ投げの結果: ",
6                 Dice[1], ", ", Dice[2], ", ", Dice[3],
7                 " (合計: ", Total, ")"))
8
9   if (Total == 15) {
10     break
11   }
12 }
```

## 問 2

```
1 # 問 2-1
2 Cause <- c("喫煙", "飲酒", "食べすぎ", "寝不足", "ストレス")
3
```

```
4 for (i in Cause) {  
5   Text <- sprintf("肥満の原因は %s でしょう。", i)  
6   print(Text)  
7 }
```

```
1 # 問 2-2  
2 Effect <- c("肥満", "ハゲ", "不人気", "金欠")  
3  
4 for (i in Effect) {  
5   for (j in Cause) {  
6     Text <- sprintf("%s の原因は %s でしょう。", i, j)  
7     print(Text)  
8   }  
9 }
```

```
1 # 問 2-3  
2 Solution <- c("この薬を飲めば", "一日一麺すれば", "Song に 100 万円振り込めば")  
3  
4 for (i in Effect) {  
5   for (j in Cause) {  
6     for (k in Solution) {  
7       Text <- sprintf("%s の原因は %s ですが、%s 改善されるでしょう。", i, j, k)  
8       print(Text)  
9     }  
10   }  
11 }
```

## 31.4 関数の自作

### 問 1

```
1 Data  <- c(5, 3)
2
3 if (Data[1] > Data[2]) {
4   Temp  <- Data[1]
5   Data[1] <- Data[2]
6   Data[2] <- Temp
7 }
8
9 Data
```

## 問 2

```
1 my_sqrt <- function(x, g, e = 0.001) {
2
3   if (!is.numeric(x) | x <= 0) {
4     stop("x は正の実数でなければなりません。")
5   }
6
7   gap = Inf
8
9   while (gap > e) {
10     gap <- abs(x - g^2)
11     if (gap > e) {
12       g <- (g + x / g) / 2
13     } else {
14       return(g)
15     }
16   }
17 }
```

## 問 3

```
1 Data <- c(5, 2, 4, 1)
2
3 for (i in (length(Data)-1):1) {
4   for (j in 1:i) {
5     if (Data[j] > Data[j+1]) {
6       Temp <- Data[j]
7       Data[j] <- Data[j + 1]
8       Data[j+1] <- Temp
9     }
10  }
11 }
12
13 Data
```

#### 問 4

```
1 mySort <- function(x) {
2
3   for (i in (length(x)-1):1) {
4     for (j in 1:i) {
5       if (x[j] > x[j+1]) {
6         Temp <- x[j]
7         x[j] <- x[j + 1]
8         x[j+1] <- Temp
9       }
10     }
11   }
12
13   x
14 }
15
16 # Bubble Sort の例
17 Data <- c(28, 92, 29, 84, 29, 27, 19, 23, 32, 30)
```

```
18 mySort(Data)
```

### 問 5

```
1 DQ_Attack2 <- function(attack, defence, hp, enemy) {
2   DefaultDamage <- (attack / 2) - (defence / 4)
3   DefaultDamage <- ifelse(DefaultDamage < 0, 0, DefaultDamage)
4   DamageWidth   <- floor(DefaultDamage / 16) + 1
5
6   DamageMin     <- DefaultDamage - DamageWidth
7   DamageMin     <- ifelse(DamageMin < 0, 0, DamageMin)
8   DamageMax     <- DefaultDamage + DamageWidth
9
10  CurrentHP    <- hp
11
12  while (CurrentHP > 0) {
13    Kaisin <- runif(n = 1, min = 0, max = 1)
14    if (Kaisin <= (1/32)) {
15      Damage <- runif(n = 1, min = attack * 0.95, max = attack * 1.05)
16    } else {
17      Damage <- runif(n = 1, min = DamageMin, max = DamageMax)
18    }
19
20    Damage <- round(Damage, 0)
21
22    CurrentHP <- CurrentHP - Damage
23
24    if (Kaisin <= (1/32)) {
25      print(paste0("かいしんのいちげき!",
26                  enemy, "|", Damage, "のダメージ!!"))
27    } else{
28      print(paste0(enemy, "|", Damage, "のダメージ!!"))
29    }
```

```
30     }
31
32     paste0(enemy, "をやっつけた！")
33 }
```

## 問 6

```
1 mySample <- function(x, n, seed) {
2
3     # 以下の条件が満たされない場合、エラーメッセージを出力し、関数を停止
4     stopifnot(
5         # length(n) == 1 が満たされない場合
6         "a length of n must be 1."      = (length(n) == 1),
7         # length(seed) == 1 が満たされない場合
8         "a length of seed must be 1."    = (length(seed) == 1),
9         # is.numeric(seed) == TRUE が満たされない場合
10        "seed must be integer or double." = is.numeric(seed),
11        # ceiling(n) == n が満たされない場合
12        "n must be interger."           = (ceiling(n) == n)
13    )
14
15    # LCG() を用いて n 個の乱数を生成し、x の長さだけ倍にする
16    index <- LCG(n = n, seed = seed) * length(x)
17    # 得られた疑似乱数を切り上げる
18    index <- ceiling(index)
19    # ベクトル x の index 番目要素を抽出し、Result に格納
20    Result <- x[index]
21
22    Result
23 }
```

# 參考資料

Matias D. Cattaneo, Brigham R. Frandsen, and Rocío Titiunik. Randomization inference in the regression discontinuity design: An application to the study of party advantages in the u.s. senate. *Journal of Causal Inference*, 3:1–24, 2015.

John M. Chambers. *Extending R*. CRC Press, Boca Raton, FL, 2016.

Bradley Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7:1–26, 1979.

J.A. Hartigan and Beat Kleiner. A mosaic of television ratings. *The American Statistician*, 38:32–35, 1984.

Jeffrey Heer and Michael Bostock. Crowdsourcing graphical perception: Using mechanical turk to assess visualization design. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010.

Guido Imbens and Karthik Kalyanaraman. Optimal bandwidth choice for the regression discontinuity estimator. *The Review of Economic Studies*, 79:933–959, 2012.

Ohad Inbar, Noam Tractinsky, and Joachim Meyer. Minimalism in information visualization: Attitudes towards maximizing the data-ink ratio. In *ECCE '07: Proceedings of the 14th European conference on Cognitive ergonomics: invent! explore!*, pages 185–188, 2007.

James T. Kuznicki and N. Bruce McCutcheon. Cross-enhancement of the sour taste

on single human taste papillae. *Journal of Experimental Psychology: General*, 108: 68–89, 1979.

Edward R. Tufte. *The Visual Display of Quantitative Information* (2nd Ed.). Graphics Press, 2001.

Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59, 2014.

Leland Wilkinson. *The Grammar of Graphics*. Springer, 2005.

渉 中澤. 政党支持と政治意識の変動—個人間の差異と個人内変動の関係. 東京大学社会科学研究所パネル調査プロジェクトディスカッションペーパーシリーズ, 81, 2014.