はじめに

本書について

Julia は Jeff Bezanson, Stefan Karpinski, Viral B. Shah の 3 氏らによって開発された技術計算向けプログラミング言語で、2012 年に発表されました。Julia は高速で表現力が高く、また組み込み型と同等のユーザー定義型の追加やマクロによる構文の追加など、非常に強力な言語となっています。その一方で学習コストは低く、特にライブラリを組み合わせて使うだけならかなり単純・素直にプログラムを書くことが出来る言語です。日本でも 2014 年辺りから、主に R や Python などで統計処理や機械学習を行っていた人たちを中心として注目されてきています。

本書はそんなプログラミング言語 Julia の入門資料です。Julia で標準に用意されている型のうち基本的なものを学んだ後、ランダムウォークのシミュレーションという実例を通して、プログラミング言語 Julia の使い方を学んでいきます。

本書では、読者は何らかの言語（主に C や Fortran, Python といったもの）を用いたプログラミングの経験があるということを想定しています。変数や関数といった概念そのものについては説明をしていません。「コーディングを支える技術」（著・西尾泰和、発行・技術評論社）が、こういったプログラミング言語の各種概念に対するよい入門書・解説書になっているので、よくわからない部分があった場合はこの本を参考にすると良いでしょう。最近は Julia の日本語資料も増えてきたので、それらとあわせて読むのも良いかと思います。アクセスしやすい日本語資料は最後に軽くまとめておきました。

サンプルコード

サンプルコードは GitHub 上にあります (http://github.com/yomichi/Juliabook-Samples/latest)。

表紙絵

表紙にいるのは非公式キャラクターの Julia-tan です[*1]。AMII さん[*2]に描いていただきました。キャラクター自体の権利は@kimrin さんが、表紙絵の著作権は AMII さんがもっています。

おやくそく

本書の内容に基づく運用結果について、筆者は何ら責任を負いません。

本書に出てくる会社名・製品名は、一般に各社の登録商標または商標です。

本書は表紙絵を除き、Creative Commons Attribution-ShareAlike 4.0 International License（クリエイティブ・コモンズ 表示-継承 4.0 国際ライセンス, CC-BY-SA 4.0）の下に提供されています。

連絡先

Copyright : Yuichi Motoyama 2013-2015
URL : http://yomichi.hateblo.jp/entry/2015/12/01/022544
E-mail : yomichi@tsg.jp
twitter : @yomichi_137

[*1] http://www.mechajyo.org/wp/?page_id=6
[*2] http://www.pixiv.net/member.php?id=370464

目次

第 1 章

Julia とは

簡単に言うと、僕らはよくばりだからさ。

by Julia developers – Why We Created Julia

1.1 概説

Julia[*1] は、

> おおらかなライセンスを持つオープンソースで、C の速度と Ruby のダイナミックさ・表現力を併せ持ち、Lisp のように真のマクロを持つ、ソースコードそのものをデータとして扱える言語で、しかしながら Matlab のように自然に数式を表せて、Python のように一般的な用途に使え、R と同じぐらい統計処理を簡単に扱え、Perl のように文字列を自然に扱え、Matlab のように線形代数・行列演算に適しており、シェルスクリプトのように他のプログラムを簡単に結びつけることができ、さらに覚えるのがとっても簡単で、それでいて凄腕ハッカーをも満足させられて、対話的環境でも使えて、(JIT) コンパイルも出来て、そして、もう言ったと思うけれど大事なことなのでもう一回言うと、C ぐらい早い言語
>
> *Why We Created Julia*[*2]

が欲しかった開発者達によって作られたプログラミング言語です。完全に「ぼくのかんがえたさいきょうのプログラミング言語」という感じですが、後発の言語であることを活かして他の言語の良いところをはじめからたくさん取り入れているため、実際かなりいいところまでいっていると思います。特に速度に関しては、ベースとして LLVM を使用しており、関数の JIT コンパイルおよび

[*1] http://julialang.org/

[*2] http://julialang.org/blog/2012/02/why-we-created-julia/

表 1.1 速度比較 [Julia 公式ページより引用 (2015-12-01)]

language	version	fib	parse_int	quicksort	mandel	pi_sum	rms	rmm
Julia	0.4.0	*2.11*	*1.45*	*1.15*	*0.79*	*1.00*	*1.66*	*1.02*
Fortran	GCC 5.1.1	0.70	5.05	1.31	0.81	1.00	1.45	3.48
Python	3.4.3	77.76	17.02	32.89	15.32	21.99	17.93	1.14
R	3.2.2	533.52	45.73	264.54	53.16	9.56	14.56	1.57
Matlab	R2015b	26.89	802.52	4.92	7.58	1.00	14.52	1.12
Octave	4.0.0	9324.35	9581.44	1866.01	451.81	299.31	30.93	1.12
Mathematica	10.2.0	118.53	15.02	43.23	5.13	1.69	5.95	1.30
JavaScript	V8 3.28.71.19	3.36	6.06	2.70	0.66	1.01	2.30	15.07
Go	go1.5	1.86	1.20	1.29	1.11	1.00	2.96	1.42
LuaJIT	gsl-shell 2.3.1	1.71	5.77	2.03	0.67	1.00	3.27	1.16
Java	1.8.0_45	1.21	3.35	2.60	1.35	1.00	3.92	2.36

最適化、積極的な外部ライブラリの利用によって、他の各種スクリプト言語とは一線を画した、むしろ C/Fortran に近い性能になっています[*3]。

表 1.1 は、いくつかのアルゴリズム[*4]を、いくつかの言語[*5]を用いて実行した時にかかった時間[*6]です。それぞれ C (GCC 5.1.1) での計算時間で規格化されており、1 より小さければ C より早く、大きければ遅いという事を示しています。

このパフォーマンステストに使われたコードは Julia のソースコードと共に配布されており、自分の環境で試すこともできます。この表から、

- C と Fortran はやっぱり速い
  - C の複素数は少し遅い
  - Fortran の文字列も遅い
- Julia も見劣りしない
  - 再帰は少し遅い
- Go と JavaScript も結構速い
  - JavaScript では行列演算は BLAS/LAPACK を呼ばないためか遅い
- (Pure) Python は速くない
  - 行列演算である最後の 2 つは Numpy を使っているが、それ以外は Pure Python
- Mathematica はシンボル計算だけでなく数値計算もそれなりに強い
- Octave はどうしようもない

といったことが読み取れます。

[*3] 私見では、Julia は post python/R/Matlab というよりも modern C/Fortran と呼んだ方が近い。

[*4] 順番に、再帰処理 (`fib`)、文字列処理 (`parse_int`)、配列処理 (`quicksort`)、複素数処理 (`mandel`)、ループ (`pi_sum`)、乱数生成・行列演算 (`rand_mat_stat`)、乱数生成・行列行列積 (`rand_mat_mul`) のテスト。

[*5] それぞれの言語のユーザからは「もっと早いコードを書いて比較しろ」とか言われるようですが

[*6] ハードウェア環境は Intel Xeon CPU E7-8850 2.00 GHz and 1TB of 1067MHz DDR3 RAM で、OS は GNU/Linux

他の Julia の特徴として一部挙げると、

- オープンソースプロジェクト。ライセンスは MIT ライセンス
- 標準ライブラリとして提供される豊富な関数群
- 単純ながら強力な型システム
- 型注釈を用いた関数の多重定義と、多重ディスパッチ
- 軽量なスレッド
- 片側メッセージ通信による並列計算
- Unicode サポートと正規表現による文字列操作
- C/Fortran で書かれたライブラリの関数をラッパーなどを使わずに直接呼べる
- 構文木を直接データとして操作できる

といったものがあります。

1.2 インストール

1.2.1 Julia のバージョン

2015 年 12 月 15 日現在、Julia は v0.3.x (v0.3.12), v0.4.x (v0.4.2), v0.5-dev の 3 つのバージョンが使われます。v0.4.x は 2015 年 10 月に v0.4.0 がリリースされた最新安定版で、v0.3.x と比較していろいろと強力な機能が追加されたので、基本的にはこのバージョンをつかうことをおすすめします。v0.3.x は 1 年以上もの間最新安定版としての地位を保ってきたバージョンです。現在でも v0.3.x 向けに開発されていて、v0.4.x では完全にはうまく動かないような外部パッケージ等が存在します。開発版である v0.5-dev には毎日新しい変更が取り込まれますが、ごくたまに後方互換性を破るような強烈な変更が入り、昨日まで動いていたコードやパッケージがいきなり動かなくなる、といったことが起こりえます。慣れていればすぐに対処できる[*7]し、強力で便利な新機能[*8]もあるのですが、これから始めようという人や、周りにも薦めてみようという人は安定版を使うか、開発版でも頻繁にビルドしないようにすることを薦めます。本書では 現在 (2014-12-15) の最新安定版 (v0.4.2) をメインとして解説しますが、まだまだ v0.3.x のコードを見る機会もあるかと思うので、差異についても適宜触れていきます。ただし、これからプログラムを書く場合は素直に v0.4.x で書いたほうが良いでしょう。

[*7] むしろ楽しむこともできる

[*8] v0.5-dev では例えば、より強力な単体テスト構文、スレッド並列などが導入されている

1.2.2 ビルド済みバイナリのインストール

`http://julialang.org/downloads/` には、Julia のそれぞれのバージョンについて、各 OS 用のビルド用バイナリやパッケージが置いてあります。

1.2.3 ソースコードからのビルド

最新版のビルド

ソースコードは、GitHub にある Julia のリポジトリ `https://github.com/JuliaLang/julia` からクローンして手に入れます。git の `master` ブランチが最新開発版バージョンです。ビルドするには以下のツールが必要です。

- GNU make
- gcc & g++ or Clang or Intel C/C++ compiler
- gfortran
- git
- perl
- wget, curl, or fetch (FreeBSD)
- m4
- patch

できるだけ最新のバージョンを使いましょう。

これらのツールを揃え、ソースコードを手に入れたら、`make` とするだけでビルドされます。初回のビルドでは、外部ライブラリを大量にダウンロード・ビルドするため、かなり時間がかかります[*9]。大抵は大丈夫だと思いますが、もしもエラーが出たら、`README.md` を読んでください。各 OS ごとに注意点が書かれています。

ビルドが成功すると、`$JULIAHOME/usr` というディレクトリと[*10]、`$JULIAHOME/usr/bin/julia` を指すシンボリックリンク `$JULIAHOME/julia` が生成されます。

ビルドオプション

コンパイラの変更など、カスタムビルドをしたい場合、`Make.user` というファイルを作ってそこに色々書いていきます[*11]。例えば Intel compiler と Intel MKL とを使いたい場合は、

```
1 USEICC = 1
2 USEFCC = 1
```

[*9] ごはんを食べるなりお風呂に入るなりする時間がとれるレベルです:D
[*10] $JULIAHOME は julia をクローンしたディレクトリを指します。
[*11] 大抵はなにもしなくてよいかと思います

```
3 USE_INTEL_MKL = 1
4 USE_INTEL_LIBM = 1
```

といった具合に指定します。また、外部ライブラリ、例えば ARPACK のビルドでコケた場合、Make.user に

```
1 USE_SYSTEM_ARPACK=1
```

などと書いておくことで、もともと存在するライブラリを使用するようになります。他のライブラリについてどういう名前を使えばよいかは、Make.user の最初の方（だいたい 10 行目以降）を見てください。

アップデートをする場合も、git pull origin master してから make するだけで大抵 OK です。もしも最後にエラーが出た場合は、make cleanall してから make しましょう[*12]。ただ、あまり長いことアップデートしていないと、外部ライブラリとの齟齬が生じたりして、ビルドできなくなることがあるようです。何度やっても出来ない場合は、make distcleanall として外部ライブラリを消してから、改めてビルドしなおすとよいでしょう。

過去版のビルド

Git の master ブランチは最新の開発版を指しています。一方過去のバージョンは、Git のタグで保持されています。好きなバージョンを選んでチェックアウトして、ビルドします。

```
1 bash> git tag
2 ...
3 v0.3.12
4 ...
5 v0.4.2
6 bash> git checkout v0.4.2
7 ...
8 bash> make
9 ...
```

開発版 (v0.5-dev) で導入されたバグフィックスなどが、安定版 (v0.4.x) にも取り入れられることがあります。ある程度まとまったタイミングで安定版のパッチバージョン (v0.4.x の x の部分) が上がります。逆に言えばバグフィックスにタイムラグがあるということなので、常に（安定版の）最新版が欲しい場合は、release-0.4 などのリリースブランチを利用します。

複数バージョンの共存

make install すると、新しく$JULIAHOME/julia-（コミットハッシュ） というディレクトリが作られて、その中に Julia の実行に必要なファイルが全て収められます[*13]。このディレクトリ

[*12] 外部ライブラリに関しては clean しないので、初回ほどには時間はかかりません。

[*13] ハッシュ名はわかりづらいので、例えばバージョン番号などでリネームしておくとよいでしょう。また、Make.user に prefix = <*dirname*> として、インストール先を指定することもできます。

の下にある bin ディレクトリに Julia のインタープリタの実行ファイルがあるので、ここに環境変数 PATH を通すか、どこかパスの通った場所に、新しく出来た実行ファイルへのシンボリックリンクをおいておくことで、様々なバージョンの Julia を共存させることが出来ます。

```
bash> make install
（中略）
bash> mv julia-bb73f3489d julia-v0.4.2
bash> ln -s Julia-v0.4.2/bin/julia julia-0.4
```

1.3　公式ドキュメント

公式ドキュメントは http://docs.julialang.org/ にあります。構文の解説や標準ライブラリのリファレンス、多言語との比較、スタイルガイドにパフォーマンス TIPs などの豊富な内容が書いてある、非常に重要な文書です。結構更新が早くて追いつくのが難しいために、翻訳があまり推奨されていませんが、一部有志が日本語へと翻訳しているようです[*14]。

[*14] http://docs.julia.tokyo/ja/latest/

第 2 章

電卓としての Julia

2.1 REPL

Julia のインストールが完了したら、早速 Julia を体験してみましょう。それには対話環境 (REPL : Read-Eval-Print Loop [*1]) を使うのが最も簡単です。Julia を起動すると、次のようにロゴ[*2]とバージョン情報が表示された後[*3]、プロンプトが表示され入力待ち状態になります。

```
1
2                _
3    _       _ _(_)_     |  A fresh approach to technical computing
4   (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
5    _ _   _| |_  __ _   |  Type "?help" for help.
6   | | | | | | |/ _` |  |
7   | | |_| | | | (_| |  |  Version 0.4.2 (2015-12-08 21:47 UTC)
8  _/ |\__'_|_|_|\__'_|  |
9 |__/                   |  x86_64-apple-darwin14.5.0
```

`Ctrl-D` を押すか、`quit()` と入力して `Enter` を叩くと、REPL から抜けることができます。他にも `Ctrl-A` で行頭に移動するなど、簡単な Emacs バインディングが使えます[*4]。

Julia の REPL にはいくつかの便利機能がついています。まず、REPL で`?` キーを叩くとヘルプモードに移行することができて、ヘルプモードで標準ライブラリの関数などの名前を入力するとリファレンスドキュメントが表示されます。

```
1 help?> sin
2 search: sin sinh sind sinc sinpi asin using isinf asinh asind
      isinteger
3
4   sin(x)
5
```

[*1] 「入力を読み込んで、式として評価して、結果を表示するループ」という意味の、こういうスクリプト言語の対話環境としては一般的な用語です。

[*2] 実際には色がついています

[*3] オプションに `--quiet (-q)` を指定すると、表示させずに起動出来ます。

[*4] 詳しくは公式ドキュメントの “Interacting with Julia” を参照ください。

```
6   Compute sine of x, where x is in radians
```

REPL で; キーを叩くとシェルモードに移行します。Julia の REPL を抜けることなくシェルコマンドを使えます。

```
1  shell> ls
2  CONTRIBUTING.md     README.windows.md  julia-v0.3.10/
3  DEBUGGER.md         VERSION            julia-v0.3.11/
4  DISTRIBUTING.md     Windows.inc        julia-v0.3.12/
5  LICENSE.md          appveyor.yml       julia-v0.4.0/
6  Make.inc            base/              julia-v0.4.1/
7  Make.powerpc        contrib/           julia-v0.4.2/
8  Make.user           deps/              src/
9  Makefile            doc/               test/
10 NEWS.md             etc/               ui/
11 README.arm.md       examples/          usr/
12 README.md           julia@             usr-staging/
```

Ctrl-P で履歴をたどれますし、途中まで入力してからカーソルキーの上を押すと、前方一致する履歴をたどれます。また、REPL で Ctrl-R とすると、コマンド履歴から検索をすることができます。

2.2 四則演算

プロンプトが表示されたら適当な文字列を入力して Enter を押すと、Julia は入力された文字列を式として解析・評価して、その結果を表示します[*5]。例えば

```
1  julia> 1+2
2  3
3  julia> 1+2*3
4  7
5  julia> (1+2)*3
6  9
```

となります。これらの例から分かる通り、Julia では C や Python, Fortran などの他の多くの言事同じく、普通に数式を書く感覚[*6]で式を書くことができます。割り算については C や Fortran とは少し異なっていて、

```
1  julia> 4/2
2  2.0
3  julia> 3/2
4  1.5
```

[*5] 最後にセミコロン; をつけると結果表示は行いません。

[*6] 具体的には、中置演算子をサポートしていて、和差より乗除の方が優先度が高く、括弧でグループ化出来る、ということ

のように、整数を整数で割った結果はかならず浮動小数点数になります。整数のまま除算したければ、div を用いて

```
julia> div(4,2)
2
julia> div(3,2)
1
julia> div(3.0,2)
1.0
julia> div(3.7,2)
1.0
```

のようにします。ここで注意すべきことは、3,4 番目の例で分かる通り、div に浮動小数点数を与えても強制的に整数として認識され、結果では（必要なら再度打ち切ってから）浮動小数点数に戻るということです。この時の切り捨ては 0 に近い方に行われます。

べき乗の演算子は^です。剰余は rem と mod の 2 つあります。結果の符号は、rem は第 1 引数の符号と、mod は第 2 引数の符号とそれぞれ一致するようになります。また、演算子 % は rem の別名です。

```
julia> rem(10, 7)   #  10 =   7 + 3
3
julia> rem(-10, 7)  # -10 =  -7 - 3
-3
julia> rem(10, -7)  #  10 =   7 + 3
3
julia> rem(-10, -7) # -10 =  -7 - 3
-3
julia> mod(10, 7)   #  10 =   7 + 3
3
julia> mod(-10, 7)  # -10 = -14 + 4
4
julia> mod(10, -7)  #  10 =  14 - 4
-4
julia> mod(-10, -7) # -10 =  -7 - 3
-3
```

#から改行までの間はコメントとして無視されます。

ビット演算も用意されていて、ビット反転 (~)、ビット積 (&)、ビット和 (|)、排他的ビット和 ($)、右論理ビットシフト (>>>)、右算術ビットシフト (>>)、左ビットシフト (<<) があります。

```
julia> bits(123)
"0000000000000000000000000000000000000000000000000000000001111011"
julia> bits(~123)
"1111111111111111111111111111111111111111111111111111111110000100"
julia> bits(124)
"0000000000000000000000000000000000000000000000000000000001111100"
julia> bits(123 & 124)
"0000000000000000000000000000000000000000000000000000000001111000"
julia> bits(123 | 124)
```

```
"00000000000000000000000000000000000000000000000000000001111111"
julia> bits(123 | 234)
"00000000000000000000000000000000000000000000000000000011111011"
julia> bits(123 $ 124)
"00000000000000000000000000000000000000000000000000000000000111"
julia> bits(123 << 1)
"00000000000000000000000000000000000000000000000000000011110110"
julia> bits(-1)
"1111111111111111111111111111111111111111111111111111111111111111"
julia> bits(-1 >> 1)
"1111111111111111111111111111111111111111111111111111111111111111"
julia> bits(-1 >>> 1)
"0111111111111111111111111111111111111111111111111111111111111111"
```

bits は 2 進表現を文字列で返す関数です。

面白い演算子として、右辺を左辺で割る\ があります[*7]。

```
julia> 2\4
2.0
```

2.3　関数

これまでも div などの例で既に出て来ましたが、Julia での関数適用は一般に () を用いて行います。

```
julia> abs(1.0)
1.0
julia> abs(-1.0)
1.0
```

「一般に」と書いた以上、一般ではないケースもあるわけで、それが演算子適用です。Julia では + や* のような演算子もその正体は普通の関数であり、引数の間に置くという特殊な記法が許されているというだけです。実際に普通の関数のように呼び出すことも出来て、前節の最初の例を書き直すと、

```
julia> +(1,2)
3
julia> +(1,*(2,3))
7
julia> *(+(1,2),3)
9
```

となります。

なお、関数名とカッコの間に空白を入れることは、v0.4 において非推奨 (deprecated) とされました。deprecated な関数・文法を使うと次のように警告が出ます。

[*7] 他の言語では、Matlab や Octave にはあるようです。

```
julia> sin (1.0)

WARNING: deprecated syntax "sin␣(".
Use "sin(" instead.
0.8414709848078965
```

非推奨になったその次以降のバージョンでは削除されるので、今はまだ使えるからと無視せずに書き直しましょう。

v0.3

v0.3 では、apply をつかうこともできます。

```
julia> apply(+,1,2)
3
```

Julia では関数も数値などと同様にオブジェクトとして存在して、関数 (apply) に渡すことができます。この形式はマクロなどで構文木をいじったりするとちょくちょく出てきます。ただし、apply 関数は v0.4 で deprecated (非推奨) とされました。

2.4 変数

Julia で変数に代入する場合、他の言語と同様に name = expression といった具合に = を用います。

```
julia> x = 1
1
julia> 2*x
2
```

x = x + 1 のような、変数になにかを足したりかけたりして、その結果を再代入するという処理はよく行われるため、短く、わかりやすく書くためのシンタックスシュガーとして複合代入演算子が用意されています。

```
julia> x
1
julia> x += 1
2
julia> x
2
```

なお、x (op)= a はあくまで x = x (op) a のシンタックスシュガーなので、C++ などのように別の演算子があるわけではないことに注意してください[*8]。再定義などもできません。

[*8] 一時オブジェクトを作るわけなので、例えば大規模行列などで複合代入演算子を使うと性能が落ちる一因となります。

2.4.1 変数名

変数名には次のルールがあります。

- 以下の文字は使えない[*9] `+-*/\%&|$~^<>?:.,@(){}[]'"‘#`
- 先頭に`!`と数字は使えない
- 大文字と小文字は区別される
- 予約語は名前に出来ない

最初の 2 つのルールがあるのは、名前にこれらの文字を使うと演算子適用が優先されるためです。`!` や数字も単項演算子として機能するために、名前の先頭には置けません[*10]。予約語は次のとおり[*11][*12]です。

```
begin end function macro return for while break continue if elseif else
try catch finally do let module import export using
type abstract bitstype typealias immutable const global local ccall
```

構文上重要な意味を持つ語[*13]のみが予約語とされていて、これ以外は、たとえ組み込みや標準ライブラリで定義されたものであっても、書き換えることができます。ただ言うまでもなく、こういうことはやめておいたほうが無難です。

```
bash> julia -q
julia> pi  # 円周率
π = 3.1415926535897...
julia> pi = 3
Warning: imported binding for pi overwritten in module Main
3
julia> Int64 # 整数型
Int64
julia> Int64 = 42
Warning: imported binding for Int64 overwritten in module Main
42
```

Julia では Unicode を使っているので[*14]日本語の変数を使うこともできます。

```
julia> ほげ = 1
1
julia> 2*ほげ
```

[*9] `!` が含まれていないことに注意
[*10] 前者は否定の演算子。後者は後述。
[*11] マニュアルに予約語一覧がないので、もしかしたら抜けがあるかもしれません。。。
[*12] v0.3 では、`=>` も予約語です。
[*13] 構文解析に影響を与える語
[*14] UNICODE8 に対応。エンコーディングは UTF8

```
4 2
```

日本語なんかだと IME の切り替えが必要だったりして面倒くさいのですが、Julia の REPL ではギリシャ文字や数学記号などを LaTeX の記法、例えば\pi などからタブ補完を行うことで簡単に書くことができます。

```
1 julia> \pi #ここでタブキーを押すと
2 julia> π   #こうなる
3 π = 3.1415926535897...
```

今回の円周率 π のように、標準で他の定数や関数に結び付けられているギリシャ文字や数学記号が結構あります。絵文字も同様に、\:+1: などしてからタブ補完することで変換可能です[*15]。REPL だけではなく Vim や Emacs でも、Julia の開発チームがメンテナンスしている julia-vim[*16] や contrib/ にある julia-mode.ml を使うことで、同様のタブ補完が使えるようになります[*17]。

名前の付け方は上のルールに従う限り自由ですが、可読性のためにはある程度の規約・慣習・一貫性があるのがよいというのは周知の事実だと思います。Julia のマニュアルでは次の慣習が紹介されています[*18]。

- 変数名は小文字にし、単語の区切りとしてアンダースコア (_) を用いる（スネークケース）。
- 型名は大文字で始め、単語の区切りとして大文字を用いる（アッパーキャメルケース）。
- 関数名、マクロ名は小文字にし、キャメルケースやスネークケースを用いない。
  - どうしても読みづらい場合にはアンダースコアで区切る。
  - 複数の関数の組み合わせ（定型文）をひとつの関数にした場合、アンダースコアで元の関数名をつなげる
  - ある関数を、別のアルゴリズム・実装を用いて再実装した場合、アンダースコアで実装名を付加する。
- 渡された引数を変更する関数は、名前の最後に! をつける。このような関数はしばしば「破壊的な関数」などと呼ばれる。

もちろん慣習なので従わなくても良いですが、出来る限り従うと良いでしょう。

2.5 Julia の型

Julia ではすべての値にかならず 1 つの具体型がついています。値の型は typeof 関数で取得できます。

[*15] 使える絵文字とその名前の一覧は GitHub 等と同様で、http://www.emoji-cheat-sheet.com/ を参照してください。

[*16] https://github.com/JuliaLang/julia-vim

[*17] 他のエディタについてもあるかもしれませんが、調べていません

[*18] “3.2 Stylistic conventions” および “31.8 Use naming conventions consistent with Julia’s base/

```
# 64 bit 整数
julia> typeof(42)
Int64
# 64 bit 浮動小数点数
julia> typeof(3.14)
Float64
# 整数型1次元配列
julia> typeof([1,2,3])
Array{Int64,1}
# ASCII文字列
julia> typeof("Julia")
ASCIIString (constructor with 2 methods)
```

具体型のほかにも抽象型があります。これはいくつかの具体型を（階層的に）グループにまとめるためにあります。isa 関数を使うことで与えた値が与えた型に属しているかどうかがわかり、<: 演算子を使うことで与えた型が与えた抽象型に属しているかどうかがわかります。

```
julia> x = 42;
julia> typeof(x)
Int64
julia> isa(x, Int64)
true
julia> isa(x, Float64)
false
julia> isa(x, Integer)
true
julia> Int <: Integer
true
julia> Int <: AbstractFloat
false
```

型名をそのまま関数として呼び出すと、(引数が不正でなければ）その型の値を作ることができます。この関数はコンストラクタと呼ばれます。

```
julia> Int64(3)
3
julia> Integer(1)
1
julia> Float64(3)
3.0
julia> AbstractFloat(3.14)
3.14
julia> Complex(1,2)
1 + 2im
```

後ほど第 4 章でもう少し詳しく見ていきます。

2.6 数値型

すべての数値型は Number 抽象型に属します。Number 型のすぐ下に実数型 Real と複素数型 Complex とがあり、実数型の下には整数型 Integer、浮動小数点数型 AbstractFloat、有理数型 Rational、無理数型 Irrational とがあります。

異なる種類の数値型を混ぜた場合（混合演算）、自動的に型変換が行われて、同じ型同士の演算となります。この時、もとの 2 つの型の表せる数の範囲を含む、もっとも小さい型に変換されます[*19]。ただし、符号付き整数と符号なし整数との混合演算では、Int 以上になります。この暗黙の型変換で値と型がどう変わるかは、それぞれ promote と promote_type で取得できます。

```
julia> promote(1, 1.0)
(1.0,1.0)
julia> promote_type(Int, Float64)
Float64
julia> promote_type(UInt8, Int16)
Int64
julia> promote_type(UInt8, Int16)
Int64
julia> promote_type(UInt8, Int8)
Int64
julia> promote_type(UInt64, Int64)
UInt64
julia> promote_type(UInt128, Int128)
UInt128
julia> promote_type(UInt64, Int128)
Int128
```

ただし、無理数のまま計算をすることができないため、Irrational だけは自動的に相手にあわせて範囲を狭める形となります。演算結果が、もとの型の表せる範囲を超えていた場合にも、後述する一部の例外をのぞいて型変換は行われません。

明示的な型変換をしたい場合には、コンストラクタを呼び出すか、float、rationalize、complex、big を使います。非整数な実数を整数にしたい場合は、打ち切り方向に合わせて round(一番近い整数)、ceil（正の無限大方向)、floor(負の無限大方向) 、trunc（0 方向）を使い、第一引数に整数型 Int を与えてください。なお、コンストラクタによる明示的な型変換で、変換先の型で表せない値を変換しようとすると例外が発生します。

各種数値リテラルにおいて、位取り目的としてアンダースコア_ を使うことができます。アンダースコアを先頭や末尾においたり、連続して使った場合はアンダースコア以降が変数名だと見なされます。

```
julia> 10_000
```

[*19] そうなるように、変換のルールが 1 組ずつ定義されています。

```
10000
julia> 10_
ERROR: _ not defined
julia> _10
ERROR: _10 not defined
julia> 10__0
ERROR: __0 not defined
```

2.6.1 整数

整数を表す抽象型は Integer です。この下に符号付き整数型、符号なし整数型を表す抽象型 Signed と Unsigned、多倍長整数を表す具体型 BigInt と、真偽値を表す具体型の Bool 型があります。

普通に整数を書いた場合 (42 とか 137 とか)、32-bit もしくは 64-bit の符号付き 10 進整数リテラル (Int32, Int64) になります。どちらになるかは、Julia をコンパイルしたマシンの OS や CPU によります[*20]。整数リテラルに接頭辞として 0x、0o、0b をつけると、それぞれ符号なし 16 進、8 進、2 進整数リテラルになります。bit 長は、$8, 16, 32, 64, 128$ bit の中のうち、リテラルを表現できる最小のものが選ばれます (UInt64 など)。

多倍長整数型 BigInt 型もあり、big で作れます。また Int128 で表せない整数リテラルを書くと、自動的に BigInt 型の整数になります（オーバーフローはしません)。

```
julia> 42
42
julia> 0b101010
0x2a
julia> 0o52
0x2a
julia> 0x2a
0x2a
julia> uint128(1) << 128
0x00000000000000000000000000000000
julia> big(1) << 128
340282366920938463463374607431768211456
julia> 340282366920938463463374607431768211456
340282366920938463463374607431768211456
```

真偽値を表す Bool 型のリテラルとして、真の true と偽の false があります。比較演算子および論理演算子は C と同様に、==（等値)、!=（非等値)、<（小なり)、<=（以下)、>（大なり)、>=（以上)、!（論理否定)、&&（論理積)、||（論理和） があります。

```
julia> 1 == 1
true
julia> 1 == 2
```

[*20] Int が Int64 か Int32 の別名として自動的に定義されます。

```
false
julia> 1 < 2 && 2 <= 2
true
julia> 1 < 2 || 2 > 3
true
julia> !true
false
julia> 1 < 2 <= 3
true
```

最後の例のように、複数の比較演算子をつなげて書くこともできます。これの意味は

```
(1 < 2) && (2 <= 3)
```

です。

真偽値を数に変換すると、true が 1 で false が 0 となります。一方で数を真偽値にする場合、1 は true に、0 は false になりますが、これ以外の数の変換はエラーとなります。

```
julia> Int(true)
1
julia> Int(false)
0
julia> Float64(true)
1.0
julia> Bool(1)
true
julia> Bool(0)
false
julia> Bool(2)
ERROR: InexactError()
 in convert at /Users/yomichi/work/JuliaLang/julia/julia-v0.4.2/lib
    /julia/sys.dylib
 in call at essentials.jl:56
```

2.6.2 オーバーフロー・除算エラー

Int, UInt およびそれよりも広い整数型では、オーバーフローした場合にもエラー（例外）を吐かずに、あふれたビットは単純に無視されます。

```
julia> typemax(Int8) + Int8(1)
-128
julia> typemax(Int8) + Int8(1)
-128
julia> typemax(Int16) + Int16(1)
-32768
julia> typemax(Int32) + Int32(1)
-2147483648
julia> typemax(Int64) + Int64(1)
-9223372036854775808
```

```
julia> typemax(Int128) + Int128(1)
-170141183460469231731687303715884105728
julia> typemin(Int64) - Int64(1)
9223372036854775807
```

typemax と typemin はその数値型で表せる最大・最小の値を返します。

（div による）ゼロ除算および typemin を-1 で割る演算[*21]は例外を返します。

```
julia> div(1, 0)
ERROR: DivideError: integer division error
 in div at /Users/yomichi/work/JuliaLang/julia/julia-v0.4.2/lib/
    julia/sys.dylib
julia> div(typemin(Int), -1)
ERROR: DivideError: integer division error
 in div at /Users/yomichi/work/JuliaLang/julia/julia-v0.4.2/lib/
    julia/sys.dylib
```

v0.3

v0.3 では、符号なし整数型の名前は Uint と i が小文字になっています。

整数型への変換には int, int8, int16, ... などの関数を使います。この時にオーバーフローが発生した場合、単純にあふれたビットが無視されます。

```
julia-0.3> int8(128)
-128
julia-0.3> typemax(Int8) << 1
-2
```

Int よりも小さな整数型で暗黙の型変換をすると、Int になります。

```
julia-0.3> promote_type(Int8, Int16)
Int64
```

2.7　浮動小数点数

浮動小数点数を表す抽象型は AbstractFloat です。ピリオド . を数字に混ぜると、浮動小数点数リテラルになります。デフォルトでは 64-bit 浮動小数点数で、型は Float64 です[*22]。絶対値の非常に大きな、もしくは小さな浮動小数点数、例えば 6.022×10^{23} や 6.626×10^{-34} を表すときには、6.022e23 や 6.626e-34 のような、いわゆる科学技術表記も使えます。e は大文字小文字どちらでも構いません。科学技術表記で、e の代わりに f を使うと、32-bit 浮動小数点数になります。型は Float32 です。科学技術表記で 0x を頭につけ、e の代わりに p を使うと、16 進浮動小数点数表記ができます。指数の底は 2 です。

[*21] 掛け算ではオーバーフロー。

[*22] 規格は IEEE754-2008

```
julia> 3.14
3.14
julia> 0.314e1
3.14
julia> 314e-2
3.14
julia> 3.14f0
3.14f0
julia> 0x1.p0
1.0
julia> 0x1.1p0
1.0625
julia> 0x1.8p0
1.5
julia> 0x1.8p1
3.0
```

正数、負数、ゼロをゼロで割ると、その結果はそれぞれ正の無限大 (Inf)、負の無限大 (-Inf)、非数 (NaN) になります。

```
julia> 1/0
Inf
julia> -1/0
-Inf
julia> 0/0
NaN
```

整数に多倍長整数 BigInt があるのと同様に、浮動小数点数にも多倍長浮動小数点数が用意されています。その型は BigFloat です。BigInt 型ともども、big 関数で作ることができます。

```
julia> big(1.0)
1e+00 with 256 bits of precision
julia> big(0.1)
1.000000000000000055511151231257827021181583404541015625e-01 with
    256 bits of precision
julia> big"0.1"  # big("0.1") ではない
1.00 ...中略... 0000000000000000000000000000000000000002e-01 with
    256 bits of precision
julia> big(0.1) == big"0.1"
false
julia> big(0.1) - big"0.1"
5.551115123125782702118158340454101562499...-99784095786122638884e
    -18 with 256 bits of precision
```

たとえば 0.1 は有限の bit 数の 2 進の浮動小数点数では表せません。できるだけ誤差なく扱いたければ多倍長浮動小数点数を使う必要がありますが、この時 big にそのまま 0.1 を渡すと、big が処理する前に Float64 になってしまうために、64 bit 浮動小数点数の精度 (2 進 53 bit) のままになってしまいます。そのため、上記のように文字列（2.9 節）として渡した方が安全です。

なお、big"0.1" であり big("0.1") ではないことに注意が必要です[*23]。この変換はコンパイル時（＝関数の初回呼び出し時）に行われ、コンパイル後のコードには他のリテラル同様値がそのまま埋め込まれます。

BigFloat の精度はデフォルトでは 256 bit となっていますが、これは set_bigfloat_precision と get_bigfloat_precision とを使うことで取得・設定できます。また、with_bigfloat_precision を使うと、一時的に精度を変更することができます。

```
julia> big(1.0)
1e+00 with 256 bits of precision
julia> get_bigfloat_precision()
256
julia> set_bigfloat_precision(512)
512
julia> big(1.0)
1e+00 with 512 bits of precision
julia> with_bigfloat_precision(128) do
       big(1.0)
       end
1e+00 with 128 bits of precision
julia> big(1.0)
1e+00 with 512 bits of precision
```

do については 3.4.2 節で説明します。

v0.3

v0.3 では、浮動小数点数抽象型の名前は FloatingPoint です。また、big"0.1" は使えず、BigFloat("0.1") とする必要があります。こちらは通常の関数であり、実行時の変換なので、速度のオーバーヘッドが生じます。

2.7.1　係数リテラル

変数名の前に空白を入れずに数値リテラルを置くと、自動的に* が間に挟まっているものとして認識されます。

```
julia> x = 2
2
julia> 2x
4
julia> 2x^3x == 2*(x^(3*x))
true
```

[*23] 詳細は文字列の節で述べます。

2 つめの例は、べき乗演算子^ と組み合わせた時の結合順位を表したものです[*24]。この記法によって、多項式を書くのが非常に楽になります。実は後ろに来るものは変数だけではなく式でもよくて[*25]、

```
julia> 2(x+x)
8
julia> 2sin(1.0) == 2*sin(1.0)
true
```

のように書くこともできます。他の言語で慣れていると不思議な感じがするかと思いますが、慣れると（特に数値計算や次に出てくる複素数では）非常に便利です。なお、この演算は普通の乗除演算子*/\よりも優先度が高くなっています。そのため、たとえば $\frac{1}{2x}$ という数式を 1/2x で表すことができます。便利といえば便利ですが、1/2*x と紛らわしいので、注意が必要です[*26]。

なお、x0 や e0 などのような変数名に係数リテラルを付与しようとすると、非 10 進整数リテラルや科学技術表記浮動小数点数リテラルと同じ形になることがあります (0x00 や 1e0 など)。Julia ではこの場合、後者だとみなされることになります。非 10 進リテラルの場合、衝突しうる係数は 0 だけなので、実際に使うことはほとんどなく、事実上問題になりませんが、科学技術表記のほうはこの手の衝突が起こりえます。e, E, f, F のすぐ後に数字が続く変数名を使うときには注意してください。

```
julia> x10 = 3
3
julia> e1 = 3
3
julia> 3x10  # 3 * 3
9
julia> 0x10  # "16" の16進表現
0x10
julia> 3e1   # 3 * 10^1 の科学技術表記
30.0
```

2.7.2 有理数

有理数を表す型は Rational{T}です。この T は型パラメータと呼ばれるもので、T に具体的な型を入れることで 1 つの型になります[*27]。抽象型を用いることで、T として使える型を制限できます。実際に、Rational は Raional{T<:Integer} と定義されていて、T には Integer 抽象型、つまり整数のみが使えます。

[*24] 符号反転単項演算子- も同じ優先度です

[*25] 正確には、変数名単独でその変数の表すものを得るという式になっている、と考えたほうが良いでしょう。

[*26] 私見としては、分母に使う場合にはカッコで 1/(2x) とくくった方がわかりやすいと思います。

[*27] このように型パラメータを持つ型をパラメトリック型と呼びます。また、型パラメータを抜いた Rational はそれ自体が抽象型として扱われます。

p//q は p/q を表す有理数リテラルです。自動的に約分されます。num 関数と den 関数で、分子 (numerator) と分母 (denominator) を得られます。

```
julia> 2//4
1//2
julia> typeof(1//2)
Rational{Int64} (constructor with 1 method)
julia> 1//2 + 1//3
5//6
julia> num(1//2)
1
julia> den(1//2)
2
```

分子も分母もゼロの場合、エラーになります。分子が非ゼロの場合は問題なく、値が正負の Inf と等しい有理数になります。

```
julia> 1 // 0
1//0
julia> 1 // 0 == Inf
true
julia> rationalize(Inf)
1//0
julia> 0 // 0
ERROR: invalid rational: 0//0
 in Rational at rational.jl:6
 in // at rational.jl:15
```

2.7.3 無理数

Julia では以下の 5 つの無理数、円周率[*28]pi(π)、自然対数の底[*29]e、オイラー定数[*30]eulergamma(γ)、黄金比[*31]golden(φ)、カタラン定数[*32] catalan があらかじめ定義されています[*33]。これらは Irrational 型という型のオブジェクトとなっていて、数式の中では文脈に合わせて自動的に Float64 や BigFloat 型へと変換されます。自分で新しい無理数の定数を定義することもできます。

```
julia> Base.@irrational(root2, 1.4142135623730951, sqrt(big(2.0)) )
julia> root2
root2 = 1.4142135623730...
julia> float64(root2)
```

[*28] $\pi = 3.14159265358979323846\cdots$
[*29] $e = 2.71828182845904523536\cdots$
[*30] $\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \ln(n)\right) = 0.57721566490153286061\cdots$
[*31] $\varphi = \frac{1+\sqrt{5}}{2} = 1.61803398874989484820\cdots$
[*32] $G = \beta(2) = \sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)^2} = \frac{1}{1^2} - \frac{1}{3^2} + \frac{1}{5^2} - \frac{1}{7^2} + \cdots = 0.91596559417721901505\cdots$
[*33] それぞれ対応するギリシャ文字の変数も束縛済みです

```
1.4142135623730951
julia> big(root2)
1.41421356237309504880168872420969807856967...79737990732478462102e
    +00 with 256 bits of precision
```

@irrational は export されていないので、Base. が必要です[*34][*35]。Base.@irrational の第 1 引数は定数名、第 2 引数は Float64 にする時の値、最後の第 3 引数は BigFloat にする時に使う式です。

v0.3

v0.3 では、無理数型 Irrational は数学定数型 MathConst という名前であり、@irrational は@math_const という名前です。

2.7.4 実数型

整数型 Integer、浮動小数点数型 AbstractFloat、有理数型 Rational と、無理数型 Irrational のよっつを束ねる抽象型が実数型 Real です。

2.7.5 複素数

複素数の型は Complex{T}型です。T には Real 抽象型に属する型が入ります。complex 関数を使うことで複素数を作ることができます。im は虚数単位 $i = \sqrt{-1}$ を示す定数です。

```
julia> z = complex(2.0, 1.0)
2.0 + 1.0im
julia> real(z)
2.0
julia> imag(z)
1.0
julia> w = conj(z)
2.0 - 1.0im
julia> w*z
5.0 + 0.0im
julia> abs(z)
2.23606797749979
julia> abs2(z)
5.0
julia> angle(z)
0.4636476090008061
julia> cis(0.5pi)
6.123233995736766e-17 + 1.0im
```

[*34] export については 4.3 節を参照してください。

[*35] export されていない以上、ユーザが使うことを意図していないわけで、実際 Float64 か BigFloat を直接使うことが多いとは思いますが……。

real と imag で実部と虚部を得ます。conj は複素共役、abs と abs2 は絶対値とその自乗[*36]、angle は偏角で cis は cis(x) == complex(cos(x), sin(x)) です。

2.8 配列、ベクトル、行列

T 型の N 次元配列を表す型は Array{T,N}です。このように、型パラメータには数や真偽値などの値を入れることもできます。

角括弧 [] の間に要素をカンマ , で区切って並べたものを配列リテラルと呼びます。

```
julia> [1,2,3]
3-element Array{Int64, 1}:
 1
 2
 3
```

この作り方では、列ベクトル（N 行 1 列の行列）が出来上がります。要素の型は自動的に推論されます。今回は Int64 となりました。カンマ , のかわりに空白で区切ると、行ベクトル（1 行 N 列の行列）を作ることが出来ます。さらに、空白区切りの要素列をさらにセミコロン ; で区切ることで、一般の行列を作れます。

```
julia> [1 2 3]
1x3 Array{Int64, 2}:
 1  2  3
julia> [1 4 7; 2 5 8; 3 6 9]
3x3 Array{Int64, 2}:
 1  4  7
 2  5  8
 3  6  9
```

Julia の仕様として、最初の例は 1 次元配列（ベクトル）ですが、後の 2 つは 2 次元配列（普通の行列）として扱われます。普通の 1 次元配列を使いたい場合は、最初の書き方をするのが良いでしょう。

配列の要素は角括弧 [] を使ってアクセスできます。Julia では添字は 1 から始まり、2 次元以上の配列ではカンマ , で添字を並べることができます。[] の中で end を使うと、その場所に書ける最も大きな添字と等しくなります。また、多次元配列で添字を 1 つだけ書くと、1 次元配列とみなしてアクセスされます。この際の順番は、Fortran などと同様の行優先アクセスとなります。つまり、一番左側の添字から最初に動きます。

```
julia> mat= [1 4 7; 2 5 8; 3 6 9]
3x3 Array{Int64, 2}:
 1  4  7
 2  5  8
```

[*36] abs(z) = sqrt(abs2(z)) なので、ほんとうに必要になるまでは abs2 を使いましょう。

```
 3  6  9
julia> mat[2,1]
2
julia> mat[1,end]  # == mat[1,3]
7
julia> mat[4]
4
julia> mat[end]  # == mat[9]
9
julia> mat[2,2] = 0
0
julia> mat
3x3 Array{Int64,2}:
 1  4  7
 2  0  8
 3  6  9
```

さらに、: を添字で使うことで部分配列を切り出す事ができます。部分配列に対して代入することもできます。その場合、単一の値をすべての要素に代入するか、部分配列とおなじ大きさの配列を代入することができます。

```
julia> mat= [1 4 7; 2 5 8; 3 6 9]
3x3 Array{Int64, 2}:
 1  4  7
 2  5  8
 3  6  9
julia> mat[1:2, 1:2]
2x2 Array{Int64,2}:
 1  4
 2  5
julia> mat[1:2, 1:2] = 5;
julia> mat
3x3 Array{Int64,2}:
 5  5  7
 5  5  8
 3  6  9
julia> mat[1:2, 1:2] = [4 3; 2 1];
julia> mat
3x3 Array{Int64,2}:
 4  3  7
 2  1  8
 3  6  9
```

配列全体および各次元の長さは `length` および `size` 関数で取得できます。

```
julia> length(mat)
9
julia> size(mat)
(3,3)
julia> size(mat,1)
3
```

(3,3) はタプルと呼ばれるもので、複数の値を 1 つにまとめたものです。例えば関数から複数の値を返すのに用います。タプルのそれぞれの要素には、配列のように [] を用いてアクセスできます。

```
julia> n,m = 2,3;
julia> (n,m)
(2,3)
julia> n,m = m,n;
julia> (n,m)
(3,2)
```

複数の変数に同時に代入することもできますし、変数のスワップも簡単に書けます。

行列の和積演算は、そのまま演算子で繋ぐことで行えます。特に、浮動小数点数 (Float*) や複素数 (Complex*) の行列同士の演算なら、BLAS[*37] という線形演算ライブラリを用いて演算が行われるので、非常に高速になっています。和積の他にも、Julia では多数の行列演算が、例えば行列式や逆行列、対角化などの演算が用意されています。これらはやはり LAPACK という外部ライブラリを用いており、非常に高速に行えます。使える演算の一覧など、詳細はマニュアルを確認してください。

```
julia> col = Float64[1,2,3]; row = Float64[1 2 3]; mat = Float64[1
    2 3; 2 3 4; 3 4 5]
3x3 Array{Float64,2}:
 1.0  2.0  3.0
 2.0  3.0  4.0
 3.0  4.0  5.0
julia> row*col  # 内積
1-element Array{Float64,1}:
 14.0
julia> col*row  # 直積
3x3 Array{Float64,2}:
 1.0  2.0  3.0
 2.0  4.0  6.0
 3.0  6.0  9.0
julia> row*mat*col
1-element Array{Float64,1}:
132.0
julia> eig(mat) # 対角化
([-0.6234753829797987,0.0,9.623475382979784], # 固有値
3x3 Array{Float64,2}: # 固有ベクトル
 -0.827671   0.408248  -0.38509
 -0.142414  -0.816497  -0.55951
  0.542844   0.408248  -0.733931)
```

この例の最初のように、[] の前に型名を置くことで、要素の型を推論させずに直接指定出来ます。どんな要素も入る配列を作りたい場合は、Any 型を先頭に付けることで、Any 型の配列にしてください。すべての型は Any 抽象型に属するため、任意の要素を含むことができるようになります。

[*37] デフォルトでは OpenBLAS を使います。Intel MKL を使うことも出来ます。

逆に Float64 などの具体型の配列を作ると、要素の型を変えることができなくなりますが、最適化が働くようになります。

内積の計算からも分かる通り、* の結果行列が 1×1 のスカラーになったとしても、Julia では 1×1 の行列として扱われます。スカラーとして取り出したい場合は、明示的に添字アクセスしてください。

v0.3

v0.3 では、配列リテラルで [] のかわりに{} を使うこともできます。後者では型推論が働かずに、Any 型の配列となります。

2.9 文字列

2.9.1 文字列

文字 1 つを表す文字型は Char です。そのリテラルは、一文字をシングルクォーテーション'' で囲むことで作れます。また、文字列はダブルクォーテーション""で で囲みます。ASCII 文字列を表す ASCIIString と、UTF-8/16/32 でエンコードされたユニコード文字列を表す UTF8String, UTF16String, UTF32String とがあり、これらをまとめた抽象型が AbstractString です。

```
julia> 'a'
'a'
julia> 'α'
'α'
julia> "Julia"
"Julia"
```

配列と同様に、[] を用いて文字や部分文字列を得ることができます。しかし、UTF8String, UTF16String では、文字ごとに必要なバイト数が異なるため、単純に [] を使ったアクセスをすることができません。

```
julia> sizeof("J")
1
julia> sizeof("言")
3
julia> str = "Julia言語"
"Julia言語"
julia> typeof(str)
UTF8String
julia> str[1]
'J'
julia> str[2]
'u'
julia> str[1:2]
"Ju"
julia> str[6]
```

```
'言'
julia> str[7]
ERROR: invalid character index
 in next at ./unicode/utf8.jl:69
 in getindex at strings/basic.jl:37
julia> str[9]
'語'
```

chr2ind(str, n) で n 文字目の添字が得られます。

```
julia> chr2ind(str, 7)
9
julia> str[ chr2ind(str, 7) ]
'語'
```

また、nextind(str, index) 関数を使うと、index の次に有効な添字が得られます。

```
julia> nextind(str, 1)
2
julia> nextind(str, 6)
9
```

v0.3

v0.3 では、文字列を表す抽象型の名前は String です。

2.9.2　変数埋め込み

文字列リテラル中に$x と書くと、変数 x の値を文字列化して埋め込むことができます。変数の値だけでなく、式の結果を埋め込むこともできます（string interpolation, 文字列補間）。

```
julia> answer = 42;
julia> "生命、宇宙、すべての答えは$answer"
"生命、宇宙、すべての答えは42"
julia> "sin(1)␣=␣$(sin(1))"
"sin(1)␣=␣0.8414709848078965"
```

2.9.3　文字列操作

* で文字列の結合ができます。

```
julia> str1 = "Julia"
"Julia"
julia> str2 = "言語"
"言語"
julia> str1 * str2
"Julia言語"
```

数値型の混合計算で自動的に型変換がなされるように、文字列型も ASCIIString から UTF8String へと自動的に変換されます。

strip 関数で前後の空白の取り除きができます。前か後ろのどちらかだけ取り除く場合は、それぞれ lstrip と rstrip を使います。

```
julia> strip("␣␣Julia␣Lang␣␣")
"Julia␣Lang"
julia> lstrip("␣␣Julia␣Lang␣␣")
"Julia␣Lang␣␣"
julia> rstrip("␣␣Julia␣Lang␣␣")
"␣␣Julia␣Lang"
```

文字列の分割には split 関数を使います。デフォルトでは空白文字で分割します。

```
julia> split("Hello,␣Julia␣Lang!")
3-element Array{SubString{ASCIIString},1}:
 "Hello,"
 "Julia"
 "Lang!"
julia> split("Hello,␣Julia␣Lang!", ',')
2-element Array{SubString{ASCIIString},1}:
 "Hello"
 "␣Julia␣Lang!"
julia> split("Hello,␣Julia␣Lang!", [',',',','␣'])
4-element Array{SubString{ASCIIString},1}:
 "Hello"
 ""
 "Julia"
 "Lang!"
julia> split("Hello,␣Julia␣Lang!", [',',',','␣'], keep=false)
3-element Array{SubString{ASCIIString},1}:
 "Hello"
 "Julia"
 "Lang!"
```

最後の keep=false はキーワード引数（3.3.1 節参照）です。これによって、分割文字（デリミタ）が連続したときの振る舞いを変更できます。

前方一致・後方一致は startswith, endswith で、真偽値が返ってきます。検索は search で、こちらは最初に見つかった部分文字列の添字を返します。

```
julia> startswith("JuliaLang", "Julia")
true
julia> startswith("JuliaLang", "Lang")
false
julia> endswith("JuliaLang", "Lang")
true
julia> search("JuliaLang", "lia")
3:5
julia> search("JuliaLang", "C")
0:-1
```

2.9.4　正規表現

正規表現は文字列を扱うための強力なメタ言語ですが、Julia でも当然扱うことができます[*38]。正規表現オブジェクトは Regex 型の値であり、r"pattern"flag という形のリテラルで表されます。

```
# 空白およびコメントのみからなる行を表す正規表現
julia> re = r"^\s*(#|$)"
r"^\s*(#|$)"
```

文字列中 str 中に正規表現パターン re が現れるかどうかを判別する関数は ismatch(re,str) です。

```
julia> ismatch(re, "x␣=␣42")
false
julia> ismatch(re, "#␣comment")
true
julia> ismatch(re, "␣")
true
```

マッチしたかどうかだけではなくて、具体的にどの文字列にどのようにマッチしたのかを見るには match 関数およびその返り値である RegexMatch 型の値を使います。

```
julia> m=match(r"^\s*(?:#(.*)|$)", "#comment")
RegexMatch("#comment", 1="comment")
julia> m.match
"#comment"
julia> m.captures
1-element Array{Union{SubString{UTF8String},Void},1}:
 "comment"
```

マッチしなかった場合は、nothing が返ってきます。これは Void 型に属する唯一の値で、今回のように関数が返すべきものがないときに使います。

```
julia> m = match(r"^\s*(?:#(.*)|$)", "not␣comment")
julia> typeof(m)
Void
julia> m == nothing
true
```

v0.3

v0.3 では nothing の型は Nothing です。

[*38] PCRE (Perl Compatible Reguler Expression) というライブラリを使っているため、Perl における正規表現に準拠しています。正規表現自体については適当なテキストを参照してください。

2.9.5 非標準文字列リテラル

2.7 節に出てきた big"0.1" や、前節で出てきた r"" は、それぞれ BigFloat 型や Regex 型の値を文字列から（コンパイル時に）作ります。これらのような、文字列リテラルを用いた他の型のリテラルを非標準文字列リテラルと呼びます。例えば r"pattern"flag は @r_str("pattern", "flag") というマクロ呼出しと等価になり、このマクロをコンパイルすると Regex("pattern", "flag") の結果の値へと変化します。単に Regex("pattern", "flag") と書いた場合は、毎回この正規表現オブジェクトを作る事になりますが、非標準文字列リテラルを使った場合、コンパイル時に作った正規表現オブジェクトを使い回すことになるため、性能が向上します。

2.10 順序対型

Pair{T,U}は T 型から U 型への順序対型です。=> 演算子を用いて生成可能です。

```
julia> leadship = "Kagerou" => 1
"Kagerou"=>1
julia> typeof(leadship)
Pair{ASCIIString,Int64}
julia> leadship[1]
"Kagerou"
julia> leadship[2]
1
```

要素には配列やタプルと同様に [] を用いてアクセス可能です。

主に次節の辞書型 Dict を作るために用います。

v0.3

v0.3 には Pair はありません。

2.11 辞書型

辞書、すなわち K 型から V 型への連想配列の型は Dict{K,V}です。辞書型の値を作るには、Dict コンストラクタに Pair 型を渡します。

```
julia> kagerou = Dict("Kagerou"=>1, "Shiranui"=>2)
Dict{ASCIIString,Int64} with 2 entries:
  "Kagerou"  => 1
  "Shiranui" => 2
```

のようにします。K と V とは型推論により自動で決まりますが、明示したい場合は Dict{K,V}とします。型推論を無効にしたい場合、それぞれに Any を与えてください。

要素の取得や更新、追加にはやはり配列のように [] を使います。

```
julia> kagerou["Kagerou"]
1
julia> kagerou["Kuroshio"] = 3
3
```

辞書にキーが登録されているかどうかは haskey を使います。

```
julia> haskey(kagerou, "Kagerou")
true
julia> haskey(kagerou, "Yukikaze")
false
```

v0.3

v0.3 では辞書型専用のリテラル [=>] があります。配列同様、{ => } とすることで型推論を抑制できます。

```
julia> ["Kagerou" => 1, "Shiranui" => 2]
Dict{ASCIIString,Int64} with 2 entries:
  "Kagerou"  => 1
  "Shiranui" => 2
```

2.12　その他

2.12.1　起動オプション

julia の起動オプションとして -E <code> を渡すと、<code> を Julia の式として実行し、その値を表示して終了します。-e <code> を使うと、実行だけして、値は表示しません。さらに、-i を同時に指定すると、式の実行後に終了せずにそのまま REPL に入ります。

また、--depwarn=no とすることで deprecated warning を抑制したり、逆に--depwarn=error でエラーとする事もできます。

他にもいろいろな起動オプションがあります。--help 起動オプションを使うことでそれらを確認できます。

2.12.2　REPL 関連

環境変数 JULIA_INPUT_COLOR と JULIA_ANSWER_COLOR とを使うことで、REPL の配色を変えることができます。black, red, yellow, blue, cyan, magenta, white, normal, bold が使えます。

ans は最後に評価されて得た値を示しています。

```
julia> 1+2
3
julia> ans
3
```

なお、REPL の中でしか使えません。

whos() を使うと現在定義されている名前と、それが指し示す値の使用メモリ、型と値を表示します。

```
julia> a = 1;
julia> whos()
      Base  23668 KB      Module : Base
      Core   3036 KB      Module : Core
      Main  26420 KB      Module : Main
         a       8 bytes  Int64 : 1
       ans       8 bytes  Int64 : 1
```

モジュール (Module) は、大雑把には名前の衝突を避けて、ライブラリを作りやすくするためのものです。他の言語で言う名前空間やパッケージといったものと大体同じものだと思って問題ありません。Main は一番上に位置するモジュールで、Core には組み込みの型や関数が定義されています。Base は標準ライブラリです。whos にこれらのモジュールを渡すと、そこに定義されている名前の一覧が表示されます。引数なしで使うと、現在いるモジュールが渡されたのと同じ事になります。上の例では、whos(Main) と同じ事になります。

workspace() 関数を使うと、Main モジュールを新しいものに差し替えることができます。つまり、定義した変数・関数などを実質的に消去することができます。なお、差し替える前の Main モジュールは LastMain モジュールとして参照可能です。

```
julia> x = 42
42
julia> x
42
julia> workspace()
julia> x
ERROR: UndefVarError: x not defined
julia> LastMain.x
42
```

第 3 章

Julia の構文とプログラミング: 1 次元ランダムウォーク

この章では具体的なコードを通して Julia の構文などを見ていきます。その具体例として、ランダムウォーク（酔歩）のシミュレーションを取り上げます。ランダムウォークの特徴として、

- 問題の定義が簡単でわかりやすい
- シミュレーションコードを書くのが容易
- ループ・条件分岐や配列操作などのプログラミングの（数値計算の）基本的な操作が現れる
- 結果が視覚的にあらわれるので楽しい
- ある条件では理論的な取り扱いがなされていて、計算結果の正しさがわかりやすい
- 問題の拡張も容易

といったものがあるため、あるプログラミング言語の入門にはかなり適していると考えられます[*1]。

3.1 ランダムウォーク

次の 2 つのルールにしたがって動く、数直線上の点 P を考えます。

1. 時刻 0 で原点 0 にいる。
2. 時刻 t で座標 x にいたなら、時刻 $t+1$ では確率 p で $x+1$ に、確率 $1-p$ で $x-1$ に移動する。

この時、P は一次元空間上をランダムウォークする、と言います[*2]。図 3.1 は実際のランダムウォークの例を示しています。左は一次元ランダムウォーク（横軸が時間）で、右は二次元（連続空間）ランダムウォークの軌跡です。

[*1] 他の言語を学ぶときにも使ってみるといいかもしれません。

[*2] 条件 1 は初期化をしただけなので、重要なのは条件 2 の方です。

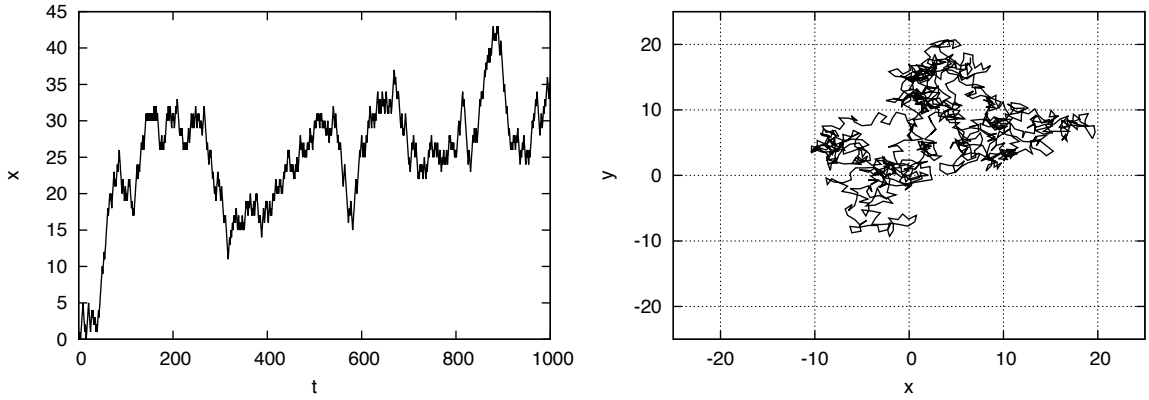

図 3.1　左は一次元ランダムウォークにおける軌跡の時系列で、右は二次元ランダムウォークにおける $x-y$ 平面での軌跡。

ランダムウォークの一番単純な問題として、時刻 t でウォーカー（しばしば点をこう呼びます）の座標の確率分布はどうなっているのか、期待値や分散はどうなっているのか、といったものがあります。一次元の時には中心極限定理により簡単に考えることが出来て、時刻 t の大きい極限で正規分布に近づいていき、その期待値は $E[X(t)] = (2p-1)t$、分散は $V[X(t)] = 4p(1-p)t$ となります。特に $p = 1/2$ では $E[X(t)] = 0, V[X(t)] = t$ となり、ランダムウォークで L だけ離れた場所にいくために平均して L^2 の時間がかかる、ということになります。

単純なケースでは手で容易に計算できるのですが、次元を増やしたり、制限（同じ場所に二回以上行っては行けない[*3]など）をつけたりしていくと、どんどん難しくなっていき、ついには手では解けなくなります。その場合、数値的にシミュレーションすることで解決します。実際に一次元ランダムウォークについて数値シミュレーションを行い、到達距離の期待値および分散をプロットしたものが図 3.2 です。期待値（左）は時間がたってもほとんど 0 のままで、一方の分散（右）は時間に比例して増えていくことが見て取れます。

ちなみに現実的な問題のモデルとしては、溶媒中の粒子の拡散（ブラウン運動）や、タンパク質のような高分子の折りたたみ（自己回避ランダムウォーク）、金融工学（ランダムウォーク理論）などに用いられます。

3.2　プログラムの実行

プログラムを書く前に、実行方法について触れておきます。これから示していくプログラムの実行方法には、4 つの実行方法があります。

1 つめは、REPL に入力するというものです。 一番基本的で単純な方法です。数行で書ける使い捨てのプログラムや構文の確認などで便利です。

[*3] 自己回避ランダムウォークとよびます

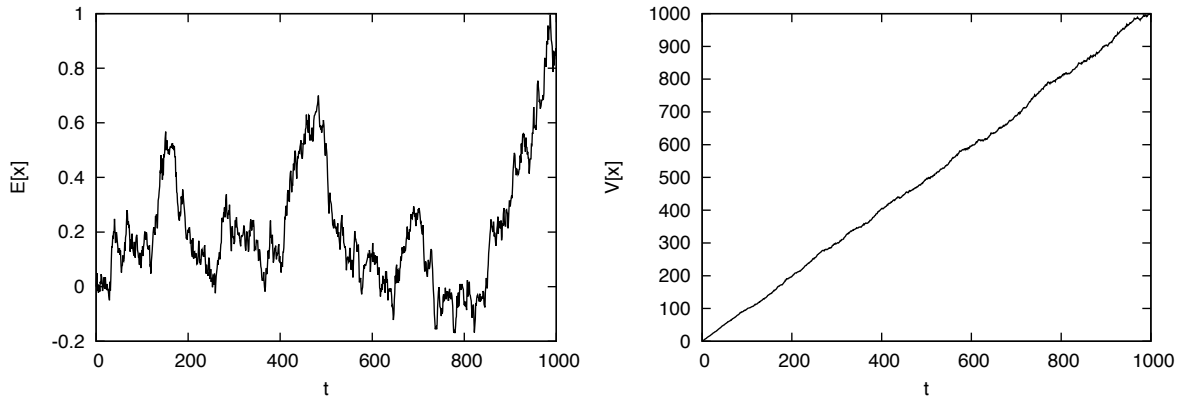

図 3.2 1 次元ランダムウォークにおける到達距離の期待値（左）および分散（右）

2 つめは、適当なファイルに保存してから REPL を起動し、include 関数でファイルを読み込むというものです。include 関数で読み込まれたファイルは、一行一行 Julia の文として構文解析され、評価されます。結果として、REPL に入力するのと同じ効果が得られます[*4]。作っているプログラムのデバッグに有効です。3 つめは、Julia を起動する際の引数としてソースファイルを渡すやり方です。こうすると REPL を起動することなくソースファイルの中身が評価され、最後まで評価し終わったら終了します。完成したプログラムを動かすのに有効です。4 つめは 3 つめの変形ですが、作ったスクリプトを実行可能ファイルにしてしまうことです。シェルスクリプトや Perl, Python などの他のスクリプト言語と同様に、1 行目で実行するインタープリタを指定しておき、chmod で実行パーミッションをつけることで、そのまま実行できるようになります。

```
#1
bash> julia -e 'println("Hello, Julia!")'
Jello, Julia!

#2
bash> echo 'println("Hello, Julia!")' > hello.jl
bash> julia -q
julia> include("hello.jl")
Hello, Julia!

#3
bash> julia hello.jl
Hello, Julia!

#4
bash> echo '#!/usr/bin/env julia\nprintln("Hello, Julia!")' > hello
    .jl
bash> chmod +x hello.jl
bash> ./hello.jl
Hello, Julia!
```

[*4] include 関数そのものの返り値は、読み込んだファイルで最後に評価した値です。

`println` 関数は、渡された引数を順に文字列に変換[*5]して、標準出力に出力し、最後に改行を出力するする関数です。Julia のソースファイルの拡張子には `.jl` を用います。

3.3　最初のプログラム

それではランダムウォークのシミュレーションを Julia を用いて書いてみましょう。最初は一番簡単なケースとして、1 次元空間上で 1 つのウォーカーを動かすところから始めます。プログラミング設計は、数値計算の場合は特にですが、データ構造とアルゴリズムの決定がメイン[*6]なので、これらを考えます。

データ構造は、1 次元空間上の 1 粒子なので、座標を表す数 1 つで OK です[*7]。`x` という名前にして、ゼロ初期化します。

```
1 x = 0
```

3.3.1　関数定義、if 分岐、乱数

次にアルゴリズムですが、これは既にランダムウォークの条件として与えられています。

2. 時刻 t で座標 x にいたなら、時刻 $t+1$ では確率 p で $x+1$ に、確率 $1-p$ で $x-1$ に移動する。

これをソースコードに落とし込めば良いのです。このステップを表す関数を書いてみましょう。

ソースコード 3.1　update 関数

```
1 function update(x::Real, p::Real = 0.5)
2   if rand() < p
3     return x+one(x)
4   else
5     return x-one(x)
6   end
7 end
```

`function` キーワードを用いることで、関数定義を行うことができます。`function` の次にあるのが関数の名前、丸括弧の中にあるのが引数リストです。次の行から `end` キーワードまでが関数本体です。もっと短く `name(args) = body` という形で行うこともできます。

関数定義の引数に型注釈 `::` を書くことで、受け入れる型の種類を制限することができます。型を制限することによって、同じ名前の関数について、引数の型によって処理内容を切り替えること

[*5] `show` 関数を用います。
[*6] 生産性や再利用性、堅牢性なども考慮に入れると他のファクターが効いてきます。
[*7] 時刻はパラメータとして扱います

ができるようになります。このような、同じ名前の関数で、引数の型が違うものをそれぞれメソッドと呼び、引数の型の組み合わせから適切なメソッドを選ぶ機構のことを多重ディスパッチと呼びます。

```
julia> f(x::Integer) = x
f (generic function with 1 method)
julia> f(x::AbstractFloat) = 2x
f (generic function with 2 methods)
julia> f(42)
42
julia> f(3.14)
6.28
julia> f("Julia")
ERROR: ‘f‘ has no method matching f(::ASCIIString)
```

ここで型注釈に抽象型を与えましたが、実際には引数として渡された値の具体型それぞれ（Int64, Float64）について別メソッドが作られます。そして、具体型に対するメソッドごとに JIT コンパイルが働くので、きめ細やかな最適化がなされることになります。

今回の update 関数には引数が 2 つありますが、そのうち p の方には後ろに= 0.5 がついています。これはデフォルト値を表していて、これがある引数は指定を省略することができます。今回の例では、update(x) とすると自動的に update(x, 0.5) とみなされます。デフォルト引数は複数個用意することもできますが、これらに明示的に値を渡すときは、前から順番に渡す必要があります。Python などのように、（仮）引数を名前で指定することで柔軟な処理を呼び出しを行うためには、キーワード引数を使う必要があります。キーワード引数つきの関数は

ソースコード 3.2 update 関数 (2)

```
function update(x::Real; p::Real = 0.5)
  if rand() < p
    return x+one(x)
  else
    return x-one(x)
  end
end
```

のように、; の後に 名前 :: 型注釈 = デフォルト値 という形で定義します。こうして定義した関数では

```
julia> update(1, p=1.0)
2
```

のように名前でセミコロン以下の仮引数を指定できるようになります。

さて、2-6 行目にあるのが if 分岐（if ... elseif ... else ... end）です。elseif と else は省略可能です（今回は elseif がありませんね）。

Julia の if 文では他の型から真理値を表す Bool 型への暗黙的な変換は行われません。例えば if 文の条件部に Bool 型以外の値、例えば 1 を与えるとエラーになり、代わりに変換関数 Bool ない

し convert を使って明示的に変換する必要があります。なお、数値型 (Number) においてゼロ が偽 (false) で 1 が真 (true) である[*8]以外は、他の型と Bool 型との関係は全く定義されていないことにも注意してください。たとえば、空文字列から真偽値への変換 convert(Bool, "") は真でも偽でもなく、エラーです[*9]。むしろ、暗黙の型変換に頼るよりは、明示的に真となる条件をあたえる（たとえば、n != 0 や isodd(n) のように、ちゃんと書く）ことが求められています。

2 行目にある rand 関数は、0 以上 1 未満の一様乱数を生成する関数[*10]です。rand() < p は、確率 p で真になり、確率 $1-p$ で偽になります。

one 関数は、引数と同じ型の、1 を表す値を返す単位元関数です。one(Float64) のように型名を直接与えることもできます。他にも零元を返す zero 関数があります。

return キーワードを使うと、関数を途中で抜けることができます。return の後ろに式があった場合、それを評価してから関数を抜けます。return を使わなかった場合は、Ruby などのように関数が終了する前、最後に実行した式の値が関数の返り値になります。今回は if 分岐の後になにもないので、実は return キーワードは不要です[*11]。

3.3.2 if 分岐ファミリー

if 分岐を短く書く[*12]ための糖衣構文が 3 つあります。条件演算子（?と:の組）と論理積と論理和です。最初の条件演算子は condition ? body1 : body2 という形をしていて、condition が真なら body1 が、偽なら body2 が評価され、条件演算子全体の返り値となります。

```
if condition
  body1
else
  body2
end
```

と等価です。

論理積 a && b について、a, b がともに真偽値 Bool の値、true か false である時をまず考えます。a が true の時は、b の値がそのまま論理積の結果になります。一方で、a が false の場合は、b の値によらずに結果は false になります。この時、b の値を評価せずに飛ばす戦略を短絡評価とよび、Julia はこれを採用しています。これを踏まえると、論理積 a && b は

```
if a
  b
```

[*8] それ以外の数値を変換しようとすると実行時エラーとなります。

[*9] もちろん、convert 関数に新たなメソッド（後述）を加えてユーザーが定義することはできます。

[*10] エンジンはメルセンヌツイスタ (MT19937) です。なお、dSFMT という C で書かれたライブラリを呼び出します。

[*11] return を使うことで、そこで終了するということが明確になるので、私は大抵毎回書いています。

[*12] if condition body1 else body2 end と一行で書くことも一応できますが、ここで示す記法の方が区切りがわかりやすいでしょう。

```
3 else
4   false
5 end
```

と等価になります。ここまでは a, b ともに Bool だとして考えて上記の if 分岐を導いたのですが、逆にこの if 分岐を論理積 a && b の定義だとみなします。こうすると、a は if 分岐の条件文なので Bool である必要がありますが[*13]、b は Bool である必要はもはやありません。

論理和 a || b も論理積と同様にして、

```
1 if a
2   true
3 else
4   b
5 end
```

で定義されます。

if 分岐の糖衣構文としての論理積と論理和は、その結果を何かの変数に代入したりすることもありますが、右辺（上記 b） として副作用のある操作（副作用のある関数呼び出しや return 文など）を持ってくることが多くなります。特に、標準ライブラリではかなり多用されます。

最後に、if 分岐の仲間だけれど等価ではないものとして、ifelse 関数があります。これは ifelse(c, t, f) というように 3 つの引数をとる関数で、c ? t : f と同様に、c が true なら t を、c が false なら f を返すものです。これは関数なので、c, t, f がすべて評価されてから[*14]、その結果が ifelse に渡されて、先述のような仕組みで値が選択されます。すべての引数の評価が必要な代わりに、普通の if 分岐部分に比べてより高速な LLVM 命令にコンパイルされるので、早くなる可能性があります。

3.3.3 ユーザ定義ドキュメント

ユーザ定義の関数や変数などにも独自ドキュメントを書いて、それを標準と同じように help 関数で呼び出すことができます。ユーザ定義ドキュメントドキュメント内容を書いた文字列の後に、ドキュメントを結び付けたい名前を置くだけで登録完了です。

ソースコード 3.3 update 関数 (3)

```
1 shell> cat update.jl
2 "return␣‘x+1‘␣with␣probability␣‘p‘␣or␣‘x-1‘␣with␣‘1-p‘."
3 function update(x::Real,p::Real=0.5)
4   ifelse(rand() < p, x+one(x), x-one(x))
5 end
6
```

[*13] 前節で触れたとおり、Julia では Bool への暗黙の変換は行われません。

[*14] 第一引数から順番に評価されていきますが、これからもそうである保証は特にない（少なくとも著者は保証を見つけていない）ので注意してください。

```
julia> include("update.jl")
update (generic function with 2 methods)
help?> update
search: update

  return x+1 with probability p or x-1 with 1-p.
```

ドキュメントには Markdown 記法が使えます。本文が白黒なので伝わりませんが、`x+1` と `x-1` は地味に色が変わっています。

3.3.4 値の更新

さて、こうして与えられた座標からランダムで次の座標を返す関数が出来ました。実際に座標を更新するには、

```
x = update(x)
```

とする必要があります。一旦テストしてみるために、書いたコードを適当なファイル、例えば `update.jl` に保存して、実際に動かしてみます。

```
julia> include("update.jl")
julia> x = update(x)
1.0
julia> x = update(x)
2.0
julia> x = update(x)
3.0
julia> x = update(x)
2.0
julia> x = update(x)
1.0
```

うまくいっているようですね[*15]。

ここで、座標の更新部分 `x = update(x)` を関数にしたくなるかもしれませんが、素朴に `update!(x) = x = update(x)` とやってもうまくいきません。これは、関数に引数を渡すときは、先に引数が評価されて値（オブジェクト）となり、その値だけが渡されるためです。なお、配列などのオブジェクトの場合、変数を評価するとメモリ上に作られたオブジェクトを返します。関数には、このオブジェクトがコピーされずに渡されるので[*16]、関数内での変更が呼び出し側にも伝わってきます。

```
julia> update!(xs) = xs[1] = 137
julia> a = [1,2,3]
```

[*15] 乱数を用いたプログラミングの難しさとして、適当な統計処理をしなければ本当にうまくいっているかどうかわからないというのがありますが、今回は深く気にしないことにします。

[*16] C っぽく、アドレスを渡していると考えてもいいでしょう。

```
3-element Array{Int64,1}:
 1
 2
 3

julia> update!(a)
137

julia> a
3-element Array{Int64,1}:
 137
   2
   3
```

ただし、次は無意味です。

```
julia> bad_update!(xs) = xs = [4,5,6]  # 定義
julia> bad_update!(a)  # 呼び出し
julia> println(a)
[137,2,3]
```

これでは関数の仮引数 xs が指すものが [4,5,6] に変わるだけで、a が指すオブジェクトには何の影響も与えていないからです。

Julia ではメモリの上にオブジェクトがあって、変数はそれを指している名前にすぎない、と考えておくのが重要です[*17]。

マクロを使うことで update! をむりやり実現することができます。普通の関数が値をとって値を返す変換だとすると、マクロは式をとって式を返す変換です[*18]。意外と難しいのでこれ以上の説明はしません[*19]が、とりあえず@update! マクロの形だけ示しておくと、

```
# 定義
macro update!(ex)
  esc(:($ex = update($ex)))
end

# 呼び出し
x = 42
@update! x
```

となります。ただし、Julia のマクロ呼出しは関数呼び出しと形が違うというのと、ほかの関数、例えば後述する map 関数には渡せないという問題があります。

[*17] 浮動小数点数型 などの一部の型では、間接参照を避けるために変数はアドレスではなくバイナリエンコードされた値を直接保持していますが、概念としては変わりません。

[*18] Julia では式やプログラムも Expr という型のオブジェクトとして表現されます。

[*19] 気になる方は JuliaTokyo #3 (http://juliatokyo.connpass.com/event/13218/) で行った講演のスライドを参照してください。

3.3.5　ランダムウォーク：配列の初期化と for ループ

さて、1 ステップ先の状態を得る関数が用意出来ました。これを用いて、引数として何ステップ行うか (nsteps) と左右に動く確率 (prob) を受け取って、座標の時系列を表す配列を返す関数を書いてみましょう。

ソースコード 3.4　randomwalk 関数

```
1 function randomwalk(nsteps::Integer, prob::Real = 0.5)
2   result = zeros(Int, nsteps+1)
3   x = 0
4   for t in 1:nsteps
5     x = update(x)
6     result[t+1] = x
7   end
8   return result
9 end
```

まず、計算結果をためておく配列を初期化します。配列を任意の大きさで初期化するには Array コンストラクタを使います。

```
1 N = 1000
2 xs = Array(Float64, N)
3 for i in 1:length(xs)
4   xs[i] = zero(Float64)
5 end
```

Array コンストラクタには型名と、各次元の長さを渡します。このやり方で作った配列は初期化されていないため、自分で適当に初期化する必要があります。0 で配列を初期化するのはよくあるので、そのための便利な関数が用意されています。

```
1 N = 1000
2 xs = zeros(Float64,N)
```

zeros 関数は、与えられた型の要素を与えられた数だけ持つ配列を zero を用いてゼロ初期化する関数[*20]です。要素数を増やすことで多次元配列を作れます。この手の関数は他にもあって、ones は one で 1 として初期化するし、rand は rand によって要素をランダム初期化します。

4-7 行目は for ループです。Julia の for ループはいわゆる foreach ループで、in の後ろにあるオブジェクトがひとつずつ t に渡されます[*21]。for ループの仕組みは次のとおりです。

```
1 for x in xs
2   #
3   # body
```

[*20] 型名を省略すると Float64 を渡したものとして扱われます。

[*21] in の代わりに= を使うこともできます。両者に差は全くありません。

```
4   #
5 end
```

は、start、done、next 関数を使って、次の while ループに展開されます。

```
1 state = start(xs)
2 while !done(xs,state)
3   x,state = next(xs,state)
4   #
5   # body
6   #
7 end
```

while ループは while の直後においた式が真を返す限り、本体を実行し続けるループです。state はループの状態を表すものです。配列では、単なる数字、つまり現在何週目かを表す数です。done は xs と state を使って、ループが終了したかどうかを判定します。next は xs と state を使って、新しい状態を生成します。このような形で start、done、next 関数が適用出来る型はすべて、ユーザ定義型であっても、for ループで巡回できます。たとえば配列では次のようになっています。

```
1 start(xs :: Array) = 1
2 done(xs :: Array, state) = length(xs) == state
3 next(xs :: Array, state) = (xs[state], state+1)
```

このような型をイテラブルであると呼び、そのオブジェクトをイテレータと呼びます。

1:nsteps は $1, 2, 3, \ldots,$ nsteps の範囲を表す Range 型のイテレータを作ります。begin:step:end とすることで、begin から始まって、end を超えない最大の値まで step ずつだけ増えていく、という範囲を作れます。また、step を負にすることで、減少する範囲を表すこともできます（その場合、end 以上の最小の値で終了します)。さらに、整数だけでなく浮動小数点数や有理数でも範囲を指定出来ます。具体例を次に示します。

```
1  julia> print_range(r)=(for i in r print(i, "␣") end; println());
2  julia> print_range(1:6)
3  1 2 3 4 5 6
4  julia> print_range(1:2:6)
5  1 3 5
6  julia> print_range(6:-1:1)
7  6 5 4 3 2 1
8  julia> print_range(6:-2:1)
9  6 4 2
10 julia> print_range(0:0.5:1)
11 0.0 0.5 1.0
12 julia> print_range(0:1//2:1)
13 0//1 1//2 1//1
```

ここで使った print 関数は、引数を文字列に変換して標準出力に書き出す関数です。println と違って改行は行いません。(;) は複数の文を一つの文にまとめる構文です[*22]。

[*22] begin ... end のシンタックスシュガーです。

この for ループを使って、直接配列の初期化を行うのがリスト内包表記と呼ばれる方法です。

```
julia> [ 0.0 for i in 1:4 ]
4-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
julia> [ i for i in 1:4 ]
4-element Array{Int64,1}:
 1
 2
 3
 4
julia> [ binomial(4,i) for i in 0:4 ]
5-element Array{Int64,1}:
 1
 4
 6
 4
 1
```

3.3.6 可変長引数

他にも便利な配列の生成方法としてよく使われるものとして、reshape 関数と linspace 関数があります。reshape(itr, dims...) は、既存の配列や範囲オブジェクトから、要素が同じで、次元が違うものを作ります。

```
julia> reshape(1:6,2,3)
2x3 Array{Int64,2}:
 1  3  5
 2  4  6

julia> reshape(1:6,1,6)
1x6 Array{Int64,2}:
 1  2  3  4  5  6
```

テンソルの計算をする時や、連番で初期化するのに便利です。reshape のように関数定義の最後の仮引数に... をつけることで、可変長の引数を取る関数を作ることができます。

```
julia> vararg(x, xs...) = println(x, "␣", xs)
vararg (generic function with 1 method)
julia> vararg(1)
1 ()
julia> vararg(1, 2)
1 (2,)
julia> vararg(1, 2, 3.14)
1 (2,3.14)
julia> vararg(1, 2, 3.14, "Wasshoi!")
```

```
1 (2,3.14,"Wasshoi!")
julia> vararg()
ERROR: ‘vararg‘ has no method matching vararg()
```

この例でわかるとおり、可変長引数はタプルとして扱われます。

... のもうひとつの効果として、イテレータを展開して関数に渡す、というものがあります。

```
julia> vararg( [1,2,3])
[1,2,3] ()
julia> vararg( [1,2,3]...)
1 (2,3)
```

この例では、結果から予想されるように、vararg([1,2,3]...) は vararg(1,2,3) と展開されます。... の有無による違いに注意してください。なお、この展開は機械的に行われるため、可変長引数をもたない関数に対しても使うことができます。

linspace(start, stop, n) は、start から stop までの範囲を n-1 等分する n 要素の配列を表現するイテレータである、LinSpace 型のオブジェクトを返します。直接配列を返さないことで、使用メモリを抑えたり、本当に要素が必要になるまで評価を遅らせることができるようになります。collect 関数にイテレータを渡すことで、配列に変換することができます。

```
julia> linspace(1.0,2.0,3)
linspace(1.0,2.0,3)
julia> typeof(ans)
LinSpace{Float64}
julia> for x in linspace(1.0, 2.0, 3)
       println(x)
       end
1.0
1.5
2.0
julia> collect(linspace(1.0,2.0,3))
3-element Array{Float64,1}:
 1.0
 1.5
 2.0
```

3.3.7 その他のイテレータ

出力では、時刻と座標とを空白区切りで組にして、それを書き出すようにします。今回は配列の添字がそのまま時刻になるので、配列の添字と要素のペアを出力すればいいことになります[*23]。説明用の配列は次の通り。

```
julia> result = rand(1:5, 5)
```

[*23] gnuplot などたいていのプロットツールは、1 次元データを渡すと行数を x 座標、値を y 座標としてプロットしてくれますが、お行儀よく出力しておくことにします。

```
2 5-element Array{Int64,1}:
3  5
4  1
5  4
6  2
7  4
```

rand(Range) は与えた範囲からランダムに 1 つを返す関数で、この例のように第二引数以降に与えた要素数の配列を返すこともできます。

まず、添字と値のペアを得るのに一番単純な方法は、C 言語のように添字を for で回すことです。

```
1 julia> for i in 1:length(result)
2        println(i, "␣", result[i])
3        end
4 1 5
5 2 1
6 3 4
7 4 2
8 5 4
```

この例のように、配列の添字と値の両方が順番に欲しいということはよくあるので、もっときれいな方法が用意されています。

```
1 julia> for (i,x) in enumerate(result)
2        println(i, "␣", x)
3        end
4 1 5
5 2 1
6 3 4
7 4 2
8 5 4
```

enumerate 関数は配列の添字と対応する要素とのタプルを順番に生成するイテレータである、Enumerate 型のオブジェクトを返します。他のイテレータ生成関数として、複数のコレクションをタプルのイテレータに束ねる zip、最初の n 個およびそれ以外を返す take, drop などがあります。

```
1  julia> collect(enumerate(result))
2  5-element Array{(Int64,Int64),1}:
3   (1,5)
4   (2,1)
5   (3,4)
6   (4,2)
7   (5,4)
8  julia> collect(take(result, 2))
9  2-element Array{Int64,1}:
10  5
11  1
12 julia> collect(drop(result, 2))
```

```
13 2-element Array{Int64,1}:
14  4
15  2
16  4
```

enumerate を使うことで出力部分は次のように書けます。

```
1 result = randomwalk(10)
2
3 for (t,x) in enumerate(result)
4   println("$(t-1)␣$x")
5 end
```

Julia の配列の添字は 1 から始まりますが、今回の計算では時刻は 0 から始まるため、t ではなく t-1 になっていることに注意してください。

3.3.8 コマンドライン引数と条件演算子

最後に、randomwalk 関数に渡す入力パラメータを、コマンドライン引数を使って実行時に変えられるようにしてみましょう。

```
1 nsteps = length(ARGS)>0 ? int(ARGS[1])   : 1000
2 prob   = length(ARGS)>1 ? float(ARGS[2]) : 0.5
3
4 randomwalk(nsteps, prob)
```

ARGS はコマンドライン引数が順番に格納された、文字列の配列です。julia コマンドにスクリプト名を引数として渡すことでスクリプトの実行ができますが、この時はスクリプト名から後ろにある引数が ARGS に入ります。length は配列や文字列の長さを返す関数なので、length(ARGS) でコマンドライン引数の数がわかることになります。

1 行目はコマンドライン引数が 1 つ以上あれば最初の引数を int 関数を用いて整数に変換してから nsteps に代入し、なければ nsteps に 1000 を代入します。2 行目も同様に、コマンドライン引数が 2 つ以上あれば 2 つめの引数を float 関数を用いて浮動小数点数に変換して prob に代入し、なければ prob に 0.5 を代入します。こうして、非常に単純な形ですが、実行時（起動時）に入力を与えることが出来るようになりました。

先ほど作った randomwalk 関数に、このパラメータを渡せば完了です。実行すると標準出力にステップ数と座標が並んで出力されます。これをリダイレクトして適当なファイルに保存し、グラフソフト、例えば gnuplot で表示すれば、図 3.1 の左のようなものが出来上がります。

最後にこの節で作ったプログラムをまとめて載せておきます（ソースコード 3.5, 3.6, 3.7）。

ソースコード 3.5 update.jl

```
1 "return␣`x+1`␣with␣probability␣`p`␣or␣`x-1`␣with␣`1-p`."
2 function update(x::Real,p::Real=0.5)
3   ifelse(rand() < p, x+one(x), x-one(x))
```

```
4 end
```

ソースコード 3.6　rw_1d.jl

```
1  include("update.jl")
2
3  function randomwalk(nsteps::Integer,prob::Real = 0.5)
4    result = zeros(Int, nsteps+1)
5    x = 0
6    result[1] = x
7    for t in 1:nsteps
8      x = update(x,prob)
9      result[t+1] = x
10   end
11   return result
12 end
```

ソースコード 3.7　rw_1d_main.jl

```
1  include("rw_1d.jl")
2
3  nsteps = length(ARGS)>0 ? int(ARGS[1]) : 1000
4  prob = length(ARGS)>1 ? float(ARGS[1]) : 0.5
5
6  result = randomwalk(nsteps,prob)
7
8  for (t,x) in enumerate(result)
9    println("$(t-1)␣$x")
10 end
```

3.4　統計処理：配列で並行処理

前節では 1 本のランダムウォークを作ってみました。次は座標の期待値や分散を計算してみましょう。求めたいのは確率分布に対する期待値ですが、今回は独立なシミュレーションを複数回行い、それの平均をとることで計算します。母集団の統計量を、そこから抜き出した標本集団の統計量を用いて推定する、ということに相当します。

データ構造としては配列を用います。要素ごとに独立に、並行してランダムウォークシミュレーションをすることで、標本集団を得ることができます。

3.4.1　無名関数と高階関数

さて、座標の配列を更新します。for ループを使って
`for i in 1:length(xs) xs[i] = update(xs[i]) end` としてもいいのですが、次のようにすると

```
map!(x -> update(x,prob), xs)
```

一行ですみ、また何をする操作なのかが map! という関数名を見るだけでわかるようになります。

map! 関数の第一引数

```
x -> update(x, prob)
```

は一体なんでしょうか。答えは「引数を 1 つ受け取り、それを update の第一引数として渡し、第二引数には既に外側で定義されている prob を渡す」関数を表します。立派な関数なんですが、名前がついていないので、無名関数と呼ばれます。無名関数は (arg) -> body という形式（ラムダ式）で定義します[*24]。次は"Hello, " という文字列を出力してから引数を出力する無名関数を作って、それに"Functional World!" という引数を与えた例です。

```
(x -> println("Hello,␣" x)) ("Functional␣World!") # => Hello,
    Functional World!
```

名前こそついていませんが、やっぱり立派な関数であり、オブジェクトであるので、変数に代入する（≒名前をつける）こともできるし、他の関数に渡すこともできます。

```
g = (f,x) -> f(x)
println( g(x->2x, 2) )  # => 4
```

この例の 1 行目で作った無名関数（g に代入したもの）のように、引数として関数を受け取る関数を高階関数と呼びます。

元のソースに戻る前に、もうワンクッションおいて、map 関数の話をします。map は一変数関数と配列を引数にとって、配列の各要素に渡した関数を適用して新たな配列を作り、それを返す高階関数です。形式的な定義は以下のようになります[*25]。

```
function map(f,xs)
  result = Array(Any,0)
  for i in 1:length(xs)
    push!(result, f(xs[i]))
  end
  return result
end
```

ここで push! は配列の末尾に要素を付け加える関数です[*26]。map 関数と無名関数とを使うと、例えば与えられた配列の各要素を 2 倍したいときは

```
xs = [1, 2, 3]
xs = map(x->2x,xs)
println(xs) # => [2,4,6]
```

[*24] 引数が 1 つの場合は () は不要ですが、0 個もしくは 2 個以上の時には必要です。

[*25] 実際の定義とは異なります。

[*26] なお、末尾を落とす関数は pop! で、先頭に付け加える関数は unshift、先頭から削除する関数は shift です

と書けます[*27]。

ここで本題に戻って map! 関数ですが、これは大体予想がつく通り、map 関数の仲間です。map は渡された配列を渡された関数で操作して新しい配列を作りますが、map! は渡された配列そのものを作り替えるという違いがあります[*28]。つまり、

```
map!(x -> update(x, prob), xs)
```

は、配列 xs の各要素をランダムに 1 を足すか引くかする、という操作です。

3.4.2　高階関数と無名関数のシンタックスシュガー

今回のように、高階関数に渡す関数がその場限りのもので、特に名前をつける必要がなければ無名関数にするのが妥当です。しかし、その無名関数の本体が長くなると、次の例[*29]のように非常に見栄えの悪いものになってしまいます。

```
map( x -> begin
        y = x^2
        z = x*y+rand()
        u = w/z-32pi
        v = sin(z)+cos(u)
        return (y,z,u,v)
        end,
        xs)
```

このため、Julia では、高階関数と無名関数のためのシンタックスシュガーとして次の do 構文を用意しています。

```
func(args...) do arg
  body
end
```

これは func(arg->body, args...) と解釈されます。これを用いると、さっきの例は

```
map(xs) do x
  y = x^2
  z = x*y+rand()
  u = w/z-32pi
  v = sin(z)+cos(u)
  return (y,z,u,v)
end
```

[*27] この例の場合はもちろん、もっと簡単に 2xs と書けます

[*28] そのため、名前の最後が! で終わっています。

[*29] この無名関数は適当に書いたものなので、意味は深く考えないでください。引数リストの中に置くには面倒くさい何かだ、というだけです。

のように書きなおすことができます。第二引数以降が先に見える分 do 構文のほうが見やすくなります[*30]。とはいえ、誰が見ても見えづらいぐらい長い関数の場合、分割してそれぞれを普通の関数にするなどのリファクタリングを検討した方が良いでしょう。Julia では関数ごとに、また引数の型の組み合わせ毎に最適化が効くので、関数を分割すると最適化のチャンスが増えるという利点もあります[*31]。なお do 構文は、関数の第一引数として無名関数を渡す、というだけの構文なので、第一引数として関数を受け取れる、どの関数に対しても使うことができます。

3.4.3 統計処理関数

次に、得られたデータから情報を抽出します。今回は期待値と不偏分散を求めれば良いのですが、これらは標準ライブラリとして定義されているので、それを使うことにしましょう。mean と var は配列を標本集団とみなして、期待値や不偏分散を求める関数です。

```
1 xs = reshape(1:6,6)
2 println(mean(xs)) # => 3.5
3 println(var(xs)) # => 3.5
```

var で標本分散を計算することもできます。

```
1 println(var(xs, corrected = false)) # => 2.916666666
```

平均や分散の他にも、統計処理で使うような他の操作として、標準偏差 (std) や中央値 (median)、共分散 (cov) の計算などもできます。

3.4.4 ファイル I/O

データの初期化および生成し、そこから情報の抽出ができたので、最後に得られた情報を出力します。前回のように、標準出力に書きだしてもいいですが、今回は自分でファイルを開いてそこに書き出します。

ファイルオープンは open 関数を使って行います。ファイル名とファイルモードを渡すと、ファイルハンドラが返ってきます。ファイルモードには、次の 6 種類の文字列が指定出来ます。

r	読みこみ。ファイルが無ければエラー。(デフォルト)
r+	読み書き。ファイルが無ければエラー。
w	書き出し。ファイルが無ければ新規作成、ファイルがあったら上書き。
w+	読み書き。ファイルが無ければ新規作成、ファイルがあったら上書き。
a	書き出し。ファイルが無ければ新規作成、ファイルがあったら末尾に追加。
a+	読み書き。ファイルが無ければ新規作成、ファイルがあったら末尾に追加。

[*30] このぐらいだと個人の感性によると思います。
[*31] 特に無名関数は最適化されないので。

ファイルの読み込みでは、大きさの分からないファイルを行ごとに処理するのがほとんどなので、eachline 関数を使うのが良いでしょう。

```
io = open(filename)
for line in eachline(io)
  # line にはファイルの各行が文字列として順番に入る
end
```

他にも、各行を要素とする文字列の配列を得る readlines、1 行だけ読み込む readline、指定した文字が出るまで読み込む readuntil、全部単一の文字列として読み込む readall があります。

ファイルへの書きこみは、print や println の第一引数に手に入れたファイルハンドラをわたし、第二引数以降に出力したいものを渡すことで行えます。また、write と read 関数を使ってバイナリファイルの読み書きが可能です。

ファイルは close 関数で閉じることができます。この「開いたら閉じる」というのはかならずワンセットになっているのですが、特に長い処理を行うと忘れがちです。ファイルの閉じ忘れを防ぐために、open 関数には関数を第一引数としてとるものも用意されています。この形式でファイルを開くと、渡された関数がファイルハンドラを引数に起動し、この関数が終了した後にファイルが閉じられます。前節の do 構文を使うととてもスッキリします。

```
shell> ls -a
./  ../

julia> open("hoge.txt", "w") do io
       println(io, "hoge")
       end

shell> ls -a
./         ../        hoge.txt

julia> open("hoge.txt", "r") do io
       println(readall(io))
       end
hoge
```

STDIN、STDOUT、STDERR はそれぞれ標準入力、標準出力、標準エラー出力を指す名前です。これらをファイルハンドラの変わりに使うこともできます。

```
julia> str = readline(STDIN);
標準入出力 # <= キーボードから入力

julia> str
"標準入出力\n"

julia> println(STDOUT, str)
標準入出力
```

コマンドライン引数で出力ファイル名を指定して、指定がなければ標準出力に出すようにしてみましょう。まとめたものがソースコード 3.8 と 3.9 です。

ソースコード 3.8 rw_1d_stat.jl

```
include("update.jl")

function randomwalk(num::Integer, nsteps::Integer, prob::Real=0.5)
  means, vars = zeros(nsteps), zeros(nsteps)

  xs = zeros(num)
  means[1] = mean(xs)
  vars[1] = var(xs)
  for t in 1:nsteps
    map!(x -> update(x, prob), xs)
    means[t+1] = mean(xs)
    vars[t+1] = var(xs)
  end
  return means, vars
end
```

ソースコード 3.9 rw_1d_stat_main.jl

```
include("rw_1d_stat.jl")

num = length(ARGS)>0 ? int(ARGS[1]) : 1000
nsteps = length(ARGS)>1 ? int(ARGS[2]) : 1000
prob = length(ARGS)>2 ? float(ARGS[3]) : 0.5
filename = length(ARGS)>3 ? ARGS[4] : ""

if isempty(filename)
  io = open(filename, "w")
else
  io = STDOUT
end

means, vars = randomwalk(num, nsteps, prob=prob)

for (i, (m, v)) in enumerate(zip(means, vars))
  println(io, i-1, "␣", m, "␣", v)
end
```

3.4.5 変数のスコープ

前出の io 変数の例にもありますが、if 構文は新しい名前スコープを導入しないので、その中で作った変数をその後で参照することができます。この挙動が嫌ならば、local キーワードを使って local output = ... とすることで、局所変数にすることができます。他にも複数の文を 1 つにまとめる begin ... end も名前スコープを導入しません。名前空間を導入したい場合は let ... end が用意されています。

```
julia> begin
       a = 0
       end
0
julia> a
0
julia> begin
       local b = 0
       end
0
julia> b
ERROR: UndefVarError: b not defined
julia> let
       c = 0
       end
0
julia> c
ERROR: UndefVarError: c not defined
```

一方で global を使うことで、外側の名前スコープにある名前にアクセスできます。

```
julia> let
       global c = 0
       end
0
julia> c
0
```

しかし、let の外側にすでにある名前を使った場合、新しい名前は導入されずにそちらを使います。

```
julia> let
       c = 42
       end
42
julia> c
42
```

これを防ぐには、let name1=val1, name2=val2,... という具合に、let と同じ行で変数束縛を行います。

```
julia> a = 0
0
julia> let a=42
       println(a)
       end
42
julia> a
0
```

第 4 章

Julia の型システム：2 次元ランダムウォーク

前章で 1 次元ランダムウォークの実装をしましたが、この章では 2 次元に拡張します。

2 次元系の問題を解くのに一番手っ取り早い方法は、次のように x 座標の配列の他に y 座標の配列を追加することです。

```
xs = zeros(Float64, num)
ys = zeros(Float64, num)
for t in 0:nsteps
  # 期待値と期待値からのズレの計算
  m_x = mean(xs)
  m_y = mean(ys)
  v = 0.0
  for i in 1:num
    v += (xs[i]-m_x)^2 + (ys[i]-m_y)^2
  end
  println(output, t, "␣", v, "␣", m_x, "␣", m_y)

  # 座標の更新
  for i in 1:num
    theta = 2pi*rand()
    xs[i] += cos(theta)
    ys[i] += sin(theta)
  end
end
```

再利用や拡張を考えないならこれで十分ですが、そうでないならば、

1. x 座標の配列と y 座標の配列があるが、概念的には x 座標と y 座標の組の配列の方が望ましいし、わかりやすい[*1]

[*1] もちろん、概念的に最も正しい実装が、実用面（実装と計算のコスト）でも最も正しいとは限らないというのがプログラミングや数値計算の難しい（面白い）ところではあります。

2. 座標の更新がハードコーディングされていて、再利用しづらい
3. 1 次元と 2 次元では、詳細こそ違えど、「更新と計算を繰り返す」という構造は同じになるが、今回はそれがうまく取り入れられているとはいえない

といった難点があります。

4.1　Julia の型システム

前節の難点は、座標を表す型（合成型 composite type）を導入することで解消できます。具体的には、

1. x 座標と y 座標とをまとめた型を作ることで、座標の配列を自然に作れる
2. その型に対する各種操作、例えば更新 `update` を定義・再利用できる
3. 1 次元座標型と 2 次元座標型を作り、同じグループとしてまとめることで、統一的・並行的に扱える

といった具合に解消されます[*2]。

4.1.1　型の定義

Julia では `type` キーワードを用いて型定義ができます。たとえば `x` と `y` の 2 つのフィールドを持つ 2 次元座標型 `Point2D` は

```
type Point2D
  x :: Float64
  y :: Float64
end
```

として定義できます。ここで`::` は型指定をする演算子で、変数が指しうる値の型を制限できます。例えばこの場合、`x` と `y` とに代入できるのは `Float64` という型を持つ値だけになります[*3]。型指定を省略した場合、任意の値を代入することが出来ます。

4.1.2　値の作成：コンストラクタ

今回作った `Point2D` 型をもつ値（オブジェクト）は、次のようにして作ることが出来ます。

```
julia> p = Point2D(1.0, 2.0)
Point2D(1.0,2.0)
```

[*2] もちろん、型を作らずとも、タプルや関数などをうまく利用しても解決できますが、少なくとも Julia では型を作るのが一番綺麗な解決となります。

[*3] 重要な点として、変数はただの名前なので型をもたず、ただ具体的な値のみが型を持ちます。

型と同じ名前を持つ関数はコンストラクタと呼ばれます。後述する内部コンストラクタを自分で定義しなければ、型のフィールドの数だけ引数をとり、引数の値を上のフィールドから順番に代入していったオブジェクトを返す（内部）コンストラクタが自動的に定義されます。

コンストラクタには、型定義の内部で、型と同時に定義する（＝型の作成者にしか作れない）内部インストラクタと、型の外で定義できる（＝型の作成者じゃなくても作れる）外部コンストラクタの 2 種類あります。外部コンストラクタは、かならず内部コンストラクタを用いて型を作らないといけません。そのため、自分で内部コンストラクタを定義することで、型のオブジェクトが（生成時に）満たすべき条件を強制することが出来ます[*4]。内部コンストラクタ内では、new 関数を用いることで、現在定義中の型のオブジェクトを作ることが出来ます。

外部コンストラクタは、例えばよく使われる値での初期化など、内部コンストラクタのラッパーとしての役割を果たします。

```
1  type PositivePoint2D
2    x :: Float64
3    y :: Float64
4
5    # 内部コンストラクタ
6    PositivePoint2D(x,y) = x>0.0 && y>0.0 ? new(x,y) : error("
         parameter␣error!")
7  end
8
9  # 外部コンストラクタ
10 PositivePoint2D(x) = PositivePoint2D(x,x)
```

error は、与えられた文字列を表示して、プログラム全体を異常終了させる関数です。

4.1.3 フィールドの参照、更新

オブジェクトのフィールドは、. 演算子をつかって参照・更新を行うことが出来ます。

```
1 julia> p = Point2D(1.0, 2.0)
2 Point2D(1.0,2.0)
3 julia> p.x
4 1.0
5 julia> p.x = 42.0
6 42.0
7 julia> p
8 Point2D(42.0,2.0)
```

[*4] ただ、Julia ではオブジェクト内のアクセス権制御がなくカプセル化が完全には行えないので、後から壊すことはできてしまいます。

4.1.4　グルーピング：抽象型と具体型

64bit 整数や 32bit 整数など、整数をその bit 長によらず統一的に扱いたい、とか、もっと一般に浮動小数点数などをふくめた実数を統一的に扱いたい、こういったことがしばしば起こります。これに答えるために、Julia では抽象型 abstract type というものを用意していて、これを用いて型（具体型 concrete type）をグループ分け出来ます。抽象型自身も、より上位の抽象型を用いてグループ分けできて、全体として階層構造（木構造）をなしています。

例えば、64 bit 符号付き整数を表す具体型である `Int64` と 32 bit 符号付き整数を表す具体型 `Int32` は共に、符号付き整数型を表す抽象型 `Signed` に属しています。この関係を、`Int64` は `Signed` のサブタイプであり、`Signed` は `Int64` のスーパータイプであるといいます。`Signed` は、符号なし整数型を表す抽象型 `Unsigned` と共に、整数型を表す抽象型 `Integer` に属しています。もちろん、木構造をたどることで、`Int64` も `Integer` に属します。そして整数型 `Integer` は、浮動小数点数型（抽象型）`AbstractFloat` や有理数型（具体型） `Rational` らと共に実数型 `Real` に属します。たとえば冒頭の要望では、`Real` を使えばよい、ということになります。

型の木構造の根は、抽象型 `Any` です。つまり、全ての型は `Any` に属します。型指定を省略した場合、自動的に `Any` になるため、全ての値を代入可能になります。

抽象型は `abstract` キーワードを使って定義できます。例えば座標を表す抽象型 `Point` は

```
1 abstract Point
```

と定義できます。型と型との階層関係は`<:` 演算子を用いて表現します。例えば 2 次元座標 `Point2D` を `Point` のサブタイプ にしたければ、

```
1 type Point2D <: Point
2   x :: Float64
3   y :: Float64
4 end
```

と定義します。同様にして 1 次元座標も

```
1 type Point1D <: Point
2   x :: Float64
3 end
```

と定義しておきます。

4.1.5　型ユニオンによる抽象化

抽象型・具体型の関係は、具体型を定義した時に決まってしまい、あとから変えることはできません。また、直接の親抽象クラスはかならず 1 つだと決まっています。こういう状況の中、より柔軟に型のグルーピングを行う手法として型ユニオン `Union` があります。`Union{T,U}` とすると、`T` と `U` という 2 つの型をグループ化した特殊な抽象型を作ることができます。

```
julia> Int <: Union{Int, Float64}
true
julia> Float32 <: Union{Int, Float64}
false
```

v0.3

v0.3 では、型ユニオンに{} ではなく () を使います。

4.1.6 パラメトリック型

フィールドが指す値の型があらかじめわからないけれど、一度決まったら変更はされないということがわかっている場合、パラメトリック型 parametric type を用いることが出来ます。

```
type ParametricPoint{T <: Real} <: Point
  x :: T
end
```

この例では、T を型パラメータとして導入しています。この時、この例のように型パラメータ T が属するべき抽象型を条件付けることが出来ます。

パラメトリック型そのものは抽象型であり、型パラメータを渡して初めて具体型になります。

```
julia> ParametricPoint <: Point
true
julia> ParametricPoint{Int64} <: ParametricPoint
true
julia> ParametricPoint{Int64} <: ParametricPoint{Real}
false
```

ParametricPoint{Int64} も ParametricPoint{Real} も具体型であり、お互いにスーパークラスにはなれないため、最後の式は false になります。

パラメトリック型を作るコンストラクタもまたパラメトリックになります[*5]が、この型パラメータは引数から推論されるために、普通は明示的に指定する必要はありません。

```
julia> p_int = ParametricPoint(42)
ParametricPoint{Int64}(42)
julia> typeof(p_int.x)
Int64

julia> p_float = ParametricPoint(42.0)
ParametricPoint{Float64}(42.0)
julia> typeof(p_float.x)
Float64
```

[*5] パラメトリック関数は後述

フィールドの型が変わることが無いならば、抽象型を型指定するよりも、型パラメータで型指定して、型パラメータそのものに抽象型で制限をかける方が、より最適化が働くようになります。

4.2　型と関数

作った型のオブジェクトの操作は、直接フィールドを弄ってもいいのですが、操作毎に関数を定義して、それを用いて行うのがメンテナンスや拡張性の面で有利です。Julia では、C++ などのようにオブジェクトに関数をもたせるのではなく、C のように外の関数にオブジェクトを渡すことでオブジェクトを操作します。

4.2.1　関数とメソッド

関数定義をする時、仮引数に型指定をつけることが出来ます。これにより、同じ名前の関数でも、渡した引数の型によって、違う振る舞いをさせることが出来ます。Julia では、このそれぞれの振る舞いをメソッドと呼びます。既存の関数に、自分で作った型に関するメソッドを追加することで、自分の型を滑らかにつなげることが出来ます。

例として、零元関数 `Base.zero` に、`Point1D` の零元（`Point1D(0.0)`）を生成するメソッドを追加してみます。

```
1 import Base.zero
2 zero(p :: Point1D) = Point1D(0.0)
```

最初の `import Base.zero` は、今いるモジュール（特に指定していなければ、`Main`）に名前を持ち込む命令です。Julia の標準ライブラリは、全て `Base` というモジュールに入っているのですが、名前が衝突しない限りはモジュール名の修飾無しで呼び出せるようになっています。しかしながら、呼び出しを一回もしないうちに、同じ名前を定義してしまうと、それ以降はモジュール名を明示的に修飾しないと標準ライブラリを呼び出せなくなってしまいます。これを避けるために、`import` を明示的に行っています既存関数へのメソッドの追加は、こういうモジュール関係の問題以外は、普通に関数定義をするのと全く同様に書けます。

4.2.2　パラメトリック抽象型とシングルトン型

抽象型をパラメトリックにすることも出来ます。シングルトン型 `Type` はパラメトリック抽象型の重要な例です。`isa(A, Type{B})` は、`A` が `B` と同じ型を表している時、かつその時のみ `true` となります。シングルトン型を使ってメソッドを定義することで、関数に型名を渡して動作を切り替えることが出来ます。

例えば `zero` では、

```
1 zero(::Type{Point1D}) = Point1D(0.0)
```

と書けます。ちなみに今回、メソッド本体で引数を使っていないので、仮引数を省略することが出来ます[*6]。

4.3 座標型モジュール

ここまでの知識を使うことで、ランダムウォークに使う座標型の定義をすることが出来ます。必要な操作は、零元生成、原点からの距離 1 の点のランダム生成、ランダム更新、座標同士（ベクトル同士）の和差、スカラー倍、絶対値計算、文字列への変換です。

4.3.1 生成関数

前節で zero 関数に Point1D 型とそのシングルトン型に対するメソッドをしました。同様に Point1D 型の配列をゼロ初期化する zeros を作って行きます。

```
function zeros{P<:Point}(::Type{P}, dims...)
  res = Array(P, dims...)
  for i in 1:length(res)
    res[i] = zero(P)
  end
  return res
end
zeros{P<:Point}(A::AbstractArray{P}) = zeros(P, size(A))
```

ランダム生成に関しては、Base.rand 関数にメソッドを追加すればよいのですが、この時には第一引数に AbstractRNG をとり、第二引数にシングルトン型をとるメソッドを追加することで、その他の関数、例えば配列のランダム生成 rand(Point1D, 10) なども使えるようになります。

```
julia> function rand(r::AbstractRNG, ::Type{Point1D})
       Point1D( ifelse(rand(r)<0.5, 1.0, -1.0) )
       end;

julia> rand(Point1D)
Points.Point1D(1.0)

julia> rand(Point1D, 3)
3-element Array{Points.Point1D,1}:
 Points.Point1D(1.0)
 Points.Point1D(-1.0)
 Points.Point1D(1.0)

julia> rand(Point1D, 3, 3)
3x3 Array{Points.Point1D,2}:
 Points.Point1D(-1.0)  Points.Point1D(1.0)  Points.Point1D(-1.0)
 Points.Point1D(-1.0)  Points.Point1D(1.0)  Points.Point1D(-1.0)
```

[*6] この型名によるメソッド定義は、仮引数の省略も含めてほとんどイディオムのような形になっています。

```
 Points.Point1D(-1.0)  Points.Point1D(1.0)  Points.Point1D(-1.0)

julia> ps = zeros(Point1D, 3)
3-element Array{Points.Point1D,1}:
 Points.Point1D(0.0)
 Points.Point1D(0.0)
 Points.Point1D(0.0)

julia> rand!(ps)
3-element Array{Points.Point1D,1}:
 Points.Point1D(1.0)
 Points.Point1D(-1.0)
 Points.Point1D(1.0)
```

この AbstractRNG は乱数生成器を表す抽象型で、属する具体型としてメルセンヌツイスターを実装したライブラリである dSFMT のラッパである、MersenneTwister があります。複数の乱数生成器を同時に使いたい場合は、この型のオブジェクトを生成して[*7]、rand や randn などの乱数生成関数の第一引数に渡してください。これらの乱数生成器を渡さなかった場合、自動的にプログラムグローバルな乱数生成器 Base.Random.GLOBAL_RNG を第一引数として乱数関数が起動します。

4.3.2　モジュール

これらを 1 つのモジュールにまとめます。module ... end を使うことでモジュールを定義できます。import モジュール名 とすると、自動で (モジュール名).jl という名前のファイルを

1. 配列 LOAD_PATH にあるディレクトリ
2. $HOME/.julia/(バージョン)

および、これらの直下にある (モジュール名)/src ディレクトリの中から探しに行き、自動で include します。環境変数 JULIA_LOAD_PATH にディレクトリを:区切り[*8]でしてしておくと、LOAD_PATH に自動的に入ります。

モジュール内で定義した関数などの名前は、モジュール名. 名前 でアクセスできます。また、using モジュール名 とすると、モジュール内で export されている名前がモジュール名修飾無しで見えるようになります。using するときにモジュール名が見つからない場合は、自動的に import が行われます。

[*7] コンストラクタもしくは srand 関数で乱数の種を設定できます。
[*8] Windows では;

v0.3

v0.3 では import モジュール名 とすると、カレントディレクトリにある（モジュール名）.jl が（もしあれば）インクルードされます。

ソースコード 4.1 Points.jl

```
module Points

import Base: zero, zeros, rand
import Base: +, -, *, /, \
import Base: abs, abs2

export Point, update, plot_str

abstract Point

zero(x :: Point) = zero(typeof(x))
rand(x :: Point) = rand(typeof(x))

function zeros{P<:Point}(::Type{P}, dims...)
  ret = Array(P, dims...)
  for i in 1:length(ret)
    ret[i] = zero(P)
  end
  return ret
end

if VERSION < v"0.4.0"
  rand{P<:Point}(::Type{P}, dims::Integer...) = rand!(Array(P,dims
      ...))
  rand{P<:Point}(::Type{P}, dims::Dims) = rand!(Array(P,dims...))
end

*(p :: Point, a :: Real) = a*p
\(a :: Real, p :: Point) = p/a

abs(p :: Point) = sqrt(abs2(p))

update(p :: Point) = p + rand(p)

include("Point1d.jl")
include("Point2d.jl")

end ## of module Points
```

ソースコード 4.2 point1d.jl

```
# Point in 1D space

export Point1D
```

```
4
5  type Point1D <: Point
6    x :: Float64
7  end
8
9  point(x) = Point1D(x)
10 zero(::Type{Point1D}) = Point1D(0.0)
11
12 if VERSION < v"0.4.0"
13   function rand(::Type{Point1D})
14     return Point1D(ifelse(rand() < 0.5, 1.0, -1.0))
15   end
16   function rand(r::AbstractRNG, ::Type{Point1D})
17     return Point1D(ifelse(rand(r) < 0.5, 1.0, -1.0))
18   end
19 else
20   function rand(r::AbstractRNG, ::Type{Point1D})
21     return Point1D(ifelse(rand(r) < 0.5, 1.0, -1.0))
22   end
23 end
24
25 +(lhs :: Point1D, rhs :: Point1D) = Point1D(lhs.x + rhs.x)
26 -(lhs :: Point1D, rhs :: Point1D) = Point1D(lhs.x - rhs.x)
27 -(p :: Point1D) = Point1D(-p.x)
28 *(a :: Real, p :: Point1D) = Point1D(a*p.x)
29 /(p :: Point1D, a :: Real) = Point1D(p.x/a)
30
31 abs(p :: Point1D) = abs(p.x)
32 abs2(p :: Point1D) = p.x*p.x
33
34 plot_str(p::Point1D) = string(p.x)
```

ソースコード 4.3　point2d.jl

```
1  # Point in 2D space
2
3  export Point2D
4
5  type Point2D <: Point
6    x :: Float64
7    y :: Float64
8  end
9
10 point(x,y) = Point2D(x,y)
11 zero(::Type{Point2D}) = Point2D(0.0, 0.0)
12 if VERSION < v"0.4.0"
13   function rand(::Type{Point2D})
14     theta = 2pi*Base.rand()
15     return Point2D(cos(theta), sin(theta))
16   end
17   function rand(r::AbstractRNG, ::Type{Point2D})
18     theta = 2pi*Base.rand(r)
```

```
19      return Point2D(cos(theta), sin(theta))
20    end
21  else
22    function rand(r::AbstractRNG, ::Type{Point2D})
23      theta = 2pi*Base.rand(r)
24      return Point2D(cos(theta), sin(theta))
25    end
26  end
27
28  +(lhs :: Point2D, rhs :: Point2D) = Point2D(lhs.x + rhs.x, lhs.y +
        rhs.y)
29  -(lhs :: Point2D, rhs :: Point2D) = Point2D(lhs.x - rhs.x, lhs.y -
        rhs.y)
30  -(p :: Point2D) = Point2D(-p.x, -p.y)
31  *(a :: Real, p :: Point2D) = Point2D(a*p.x, a*p.y)
32  /(p :: Point2D, a :: Real) = Point2D(p.x/a, p.y/a)
33
34  abs(p :: Point2D) = hypot(p.x, p.y)
35  abs2(p :: Point2D) = p.x*p.x + p.y*p.y
36
37  plot_str(p :: Point2D) = string(p.x, "␣", p.y)
```

4.3.3 まとめ

これを用いると、1 次元もしくは 2 次元でのランダムウォークは次のようになります。

ソースコード 4.4 rw_stat.jl

```
1  include("Points.jl")
2  using Points
3
4  function mean_var{P<:Point}(ps :: Array{P})
5    m = mean(ps)
6    ps2 = map(p->abs2(p-m),ps)
7    v = sum(ps2)/(length(ps2)-1)
8    return abs(m),v
9  end
10
11 function randomwalk{P<:Point}(::Type{P}, num::Integer, nsteps::
       Integer)
12   means, vars = zeros(nsteps+1), zeros(nsteps+1)
13   ps = zeros(P,num)
14   means[1], vars[1] = mean_var(ps)
15   for t in 0:nsteps
16     map!(update, ps)
17     means[t+1], vars[t+1] = mean_var(ps)
18   end
19   means, vars
20 end
```

ソースコード 4.5　rw_stat_main.jl

```
1  include("rw_stat.jl")
2
3  dim = length(ARGS)>0 ? int(ARGS[1]) : 1
4  num = length(ARGS)>1 ? int(ARGS[2]) : 1000
5  nsteps = length(ARGS)>2 ? int(ARGS[3]) : 1000
6  filename = length(ARGS)>3 ? ARGS[4] : ""
7
8  if dim == 1
9    P = Point1D
10 elseif dim == 2
11   P = Point2D
12 else
13   error("dimension␣error!")
14 end
15
16 if !isempty(filename)
17   io = open(filename, "w")
18 else
19   io = STDOUT
20 end
21
22 means, vars = randomwalk(P, num, nsteps)
23
24 for (i, (m, v)) in enumerate(zip(means, vars))
25   println(io, i-1, "␣", m, "␣", v)
26 end
```

3 行目から 8 行目は、Point1D か Point2D の配列から、座標の期待値と、そこからの距離のばらつきを計算する関数です。この例のように、関数にも型パラメータを導入することが出来ます。基本的には、引数に現れるパラメトリック型の型パラメータを指定するために使います。

10 行目の randomwalk 関数では、第一引数として型名を受け取っています。そして、17 行目において、受け取った型名の配列をゼロ初期化して座標の配列を作ります。その後は、update や mean_var 関数をこの座標に適用していくのですが、それぞれの関数について、どのメソッドを適用すればいいのかは引数の型によって自動的に最適なものが選ばれます。

第一引数として渡す型名は、30-36 行目で決めています。今回は 2 次元までしか作っていないので、dim が 3 以上の時はエラーにしてありますが、3 次元以上に拡張したい場合は、各関数（update など）に関して適切なメソッドを定義しておけば、後は randomwalk 関数の第一引数を変えるだけで、関数本体に手を入れることなく拡張することが出来ます。

第 5 章

外部パッケージ

これまで、Julia そのものの構文と、標準として用意されている関数群を利用してどのようにプログラムが書けるのかということを見てきました。しかし Julia もまだまだ若いとはいえ、世に出てからそれなりに時間が経っており、公式含むさまざまな開発者が自分で書いたライブラリを公開しています。この章では、それらを使う方法を学びます。

5.1 Julia のパッケージシステム

R でいう CRAN や Python でいう PyPI などのような、パッケージを登録・公開する場、およびシステムは、Julia にもきちんと用意されています。それが Pkg モジュールと METADATA.jl[*1] です。Pkg モジュールはパッケージシステムの API を提供していて、METADATA.jl はパッケージのデータベースとして機能します。Julia の 1 つのパッケージはちょうど 1 つの Git リポジトリからなっており、METADATA.jl はパッケージの名前と Git リポジトリの URL を記録しています。

新しいパッケージをインストールするには Pkg.add 関数を使います。例えば Iterators.jl をインストールする場合は

```
julia> Pkg.add("Iterators")
INFO: Initializing package repository /Users/yomichi/.julia/v0.4
INFO: Cloning METADATA from git://github.com/JuliaLang/METADATA.jl
INFO: Cloning cache of Compat from git://github.com/JuliaLang/
    Compat.jl.git
INFO: Cloning cache of Iterators from git://github.com/JuliaLang/
    Iterators.jl.git
INFO: Installing Compat v0.7.8
INFO: Installing Iterators v0.1.9
INFO: Package database updated
```

とします。最後の.jl は省略可能です。一番最初に Pkg.add した場合、最初の 2 つの INFO が示すとおり、まずはじめにパッケージ全体の格納場所を作り、最新版の METADATA.jl

[*1] https://github.com/JuliaLang/METADATA.jl

を取得します。3 つめの INFO では Compat.jl をインストールしていますが、これは Iterators.jl が Compat.jl に依存しているためです。この依存関係は METADATA.jl に（手で）記録されています。こうしてインストールしたパッケージはデフォルトでは $HOME/.julia/<*JuliaVersion*>/<*PackageName*> に入ります。たとえば Julia v0.4.0 で Iterators.jl をインストールすると、$HOME/.julia/v0.4/Iterators に入ります。パッケージによっては自動で必要な外部ライブラリをビルド・インストールします。それらは同じパッケージと同じ場所に入るので、他の環境を汚すことはありません。また、パッケージのインストール場所は JULIA_PKGDIR 環境変数で指定することもできます。

手元の METADATA.jl およびインストール済みのパッケージを更新するには Pkg.update 関数を使います。Pkg.installed でインストール済みのパッケージ一覧を、Pkg.available で METADATA.jl に登録されているパッケージ一覧を、それぞれ文字列からバージョン文字列への辞書と、文字列の配列として得ることができます。

```
julia> Pkg.available()
759-element Array{AbstractString,1}:
 "AbstractDomains"
 "Accumulo"
 "ActiveAppearanceModels"
 "AffineTransforms"
 "AmplNLWriter"
 "AndorSIF"
 "AnsiColor"
 "AppConf"
 "AppleAccelerate"
 "ApproxFun"
 ^^e2^^8b^^ae
 "XSV"
 "YAML"
 "Yelp"
 "Yeppp"
 "YT"
 "ZChop"
 "ZipFile"
 "Zlib"
 "ZMQ"
 "ZVSimulator"

julia> Pkg.installed()
Dict{ASCIIString,VersionNumber} with 2 entries:
  "Iterators" => v"0.1.9"
  "Compat"    => v"0.7.8"
```

また、Pkg.status で現在のインストール状況が確認できます。

```
julia> Pkg.status()
1 required packages:
 - Iterators                     0.1.9
```

```
1 additional packages:
 - Compat                        0.7.8
```

上はユーザが Pkg.add で入れたパッケージ、下は依存関係の解決のため自動で入ったパッケージです。

アンインストールには Pkg.rm 関数を使います。

```
julia> Pkg.rm("Iterators")
INFO: Removing Compat v0.7.8
INFO: Removing Iterators v0.1.9
INFO: Package database updated

julia> Pkg.status()
No packages installed
```

ユーザが入れたわけではない Compat.jl は、Iterators.jl をアンインストールした結果不要になったため、自動でアンインストールされました。

METADATA.jl に登録されていないパッケージも、Pkg.clone に git リポジトリの URL を渡すことでインストール可能です。

```
julia> "RandomForests" in Pkg.available()
false

julia> Pkg.clone("git://github.com/bicycle1885/RandomForests.jl")
INFO: Cloning RandomForests from git://github.com/bicycle1885/
    RandomForests.jl
INFO: Computing changes...
INFO: Cloning cache of ArrayViews from git://github.com/JuliaLang/
    ArrayViews.jl.git
INFO: Cloning cache of DataArrays from git://github.com/JuliaStats/
    DataArrays.jl.git
INFO: Cloning cache of DataFrames from git://github.com/JuliaStats/
    DataFrames.jl.git
INFO: Cloning cache of Docile from git://github.com/MichaelHatherly
    /Docile.jl.git
INFO: Cloning cache of GZip from git://github.com/JuliaLang/GZip.jl
    .git
INFO: Cloning cache of MLBase from git://github.com/JuliaStats/
    MLBase.jl.git
INFO: Cloning cache of Reexport from git://github.com/simonster/
    Reexport.jl.git
INFO: Cloning cache of SortingAlgorithms from git://github.com/
    JuliaLang/SortingAlgorithms.jl.git
INFO: Cloning cache of StatsBase from git://github.com/JuliaStats/
    StatsBase.jl.git
INFO: Cloning cache of StatsFuns from git://github.com/JuliaStats/
    StatsFuns.jl.git
INFO: Installing ArrayViews v0.6.4
INFO: Installing Compat v0.7.8
INFO: Installing DataArrays v0.2.20
```

```
INFO: Installing DataFrames v0.6.10
INFO: Installing Docile v0.5.19
INFO: Installing GZip v0.2.18
INFO: Installing Iterators v0.1.9
INFO: Installing MLBase v0.5.2
INFO: Installing Reexport v0.0.3
INFO: Installing SortingAlgorithms v0.0.6
INFO: Installing StatsBase v0.7.4
INFO: Installing StatsFuns v0.2.0
```

さらっと使いましたが、in は二項演算子で、左の値が右のコレクションに含まれるかどうかの真偽値を返します。RandomForests.jl は METADATA.jl に登録されていないことがわかりますが、Pkg.clone を使うことで手軽にインストールできました。一度インストールすれば、それ以降は他のパッケージと同様にして更新などが可能です。

```
julia> Pkg.status()
13 additional packages:
 - ArrayViews                    0.6.4
 - Compat                        0.7.8
 - DataArrays                    0.2.20
 - DataFrames                    0.6.10
 - Docile                        0.5.19
 - GZip                          0.2.18
 - Iterators                     0.1.9
 - MLBase                        0.5.2
 - RandomForests                 0.0.0-             master (
     unregistered)
 - Reexport                      0.0.3
 - SortingAlgorithms             0.0.6
 - StatsBase                     0.7.4
 - StatsFuns                     0.2.0
```

第 6 章

その他の話題

紙面・時間の都合で触れられていない言語機能はまだまだたくさんあります。

6.1　例外処理

実行時エラー（例外）を捕捉して、別の処理を行うことができます。

6.2　マクロとメタプログラミング

Julia では変数代入や関数呼び出し、if 分岐などの式や構文・プログラムそのものもデータとして生成・操作することができます。関数が値から別の値を生成するように、マクロは式から別の式を生成します。

6.3　並行プログラミングと並列プログラミング

軽量スレッド（コルーチン）を使った並行プログラミングや、プロセス間通信を用いた並列プログラミングが提供されています。

6.4　外部プログラムの実行

シェルコマンドをデータとして生成・実行することができます。

6.5　C/Fortran で書かれたライブラリ関数の呼び出し

C/Fortran で書かれた関数を直接呼び出すことができます。

6.6　デバッグ・テスト・プロファイリング

単体テスト用の関数・マクロや、デバッグ・プロファイリング用の関数・マクロが標準で用意されています。

第 7 章

より勉強するために

公式ドキュメント　http://docs.julialang.org/en/latest/

何は無くとも公式ドキュメント。言語の説明からパフォーマンス用の TIPs、標準ライブラリのリファレンスなど幅広く載っています。精力的に更新が行われているので、翻訳が推奨されないという……

公式メーリングリスト　https://groups.google.com/forum/#!forum/julia-users

Julia Express　http://bogumilkaminski.pl/files/julia_express.pdf

Learn Julia in Y minutes　http://learnxinyminutes.com/docs/julia/

上記 2 つはソースコード断片多めの言語入門（英語）です。後者は日本語訳もありますが、少々バージョンが古いのがネックかも。

Seven More Languages in Seven Weeks

https://pragprog.com/book/7lang/seven-more-languages-in-seven-weeks

有名な “Seven Languages in Seven Weeks”[*1] の続編。この中の 1 章が Julia にあてられています。開発者とのインタビュー記事も載っています。たぶん世界初の、Julia に章を割いた商業誌。

Getting Started With Julia　https://www.packtpub.com/application-development/getting-started-julia-programming/

商業誌では（たぶん）初めての、まるごと一冊 Julia な本。

Matering Julia　https://www.packtpub.com/application-development/mastering-julia

Julia そのものよりも、Julia を使った応用例をメインとした本。

JuliaTokyo　http://julia.tokyo

日本における Julia のユーザーグループ。不定期に勉強会を開催したり、年末に Advent Calendar を開催したりしています。

JuliaTokyo Meetup　http://juliatokyo.connpass.com

[*1] 邦訳：「7 つの言語 7 つの世界」

東京で開かれている Julia の勉強会です。2015 年 12 月までに 5 回開催されました。Julia 言語を使っている人と実際に会って交流するよい機会です。発表資料も公開されています。

JuliaLang Advent Calendar　`http://qiita.com/advent-calendar/2014/julialang`

クリスマスに向けて、Julia に関する記事を 1 日 1 つずつ書いていく企画。2015 年もやってます！

Julia-wakalang　`https://github.com/JuliaTokyo/julia-wakalang`

Julia でわからんことなどがあるけれど、英語で聞くのは辛いという場合は、ここの Issue に投稿すると誰かが答えてくれるかもしれません。Slack というチャットサービスの部屋も立っているので、Issue をたてるほどでもない場合はぜひこちらに[*2]。

データサイエンティスト養成読本 R 活用編　`http://www.amazon.co.jp/dp/4774170577/`

2014 年 12 月 12 日に発売した R のムックですが、R 使いに対する Julia 入門のような記事が載っています。たぶん日本初の、Julia に章を割いた商業誌。

[*2] 私はほぼ常駐しています

索引

実例で学ぶ Julia 言語入門
v0.4.2 対応版

発行日：2015 年 12 月 31 日
発行　：夜道 / EREX 工房
連絡先：yomichi@tsg.jp
http://yomichi.hateblo.jp
@yomichi_137
印刷　：株式会社ポプルス