# 3_type_function_plot

2017 年 3 月 16 日

## 1 Introduction to Data Science with Julia

### 1.1 目次



## 2 ループ処理の中断

for 文や while 文を使っていると、ある条件になったら繰り返し処理を止めたい場合があります。

たとえば、ある数 $n$ が素数かどうかを判定するプログラムを考えます。力任せにやるならば $n$ を $2 \sim \sqrt{n}$ までの整数で割って余りが 0 になるものが一つでもあったら $n$ は素数でない、余りが 0 にならなかったら $n$ は素数と判定することが出来ます。

例として $n = 30$ とすると、30 は 2 で割り切れるのでこれ以上の計算は不必要ですが、このプログラムを for 文で書くと $\lfloor \sqrt{n} \rfloor$ の数まで計算が続いてしまいます。

```
In [1]: # n = 30 の場合、無駄な計算が 2 度入る
        n = 30
        for i in 2:√ n
            if n % i == 0
                println(n, " は素数ではありません")
            end
        end
```

```
30 は素数ではありません
30 は素数ではありません
30 は素数ではありません
```

このようなときには、break を使うと for 文から抜け出すことが出来ます

```
In [2]: n = 30
        for i in 2:√ n
            if n % i == 0
                println(n, " は素数ではありません")
                break
            end
        end
```

```
30 は素数ではありません
```

break は while 文でも使用することが出来ます

```
In [3]: i = 0
        while true
            if i > 5
                break
            end
            println(i)
            i += 1
        end
```

```
0
1
2
3
4
5
```

```
In [ ]:
```

目次に戻る

## 3　ループ処理の一部中断

break ではループを抜けることが出来ました。次はループを抜けはしないけれど、ループの先頭に戻る continue を紹介します。

例として、1 ～ 10 の数を表示するけど、8 だけは表示しない場合は次のように書けます。

```
In [4]: for i in 1:10
            if i == 8
                continue # i = 8 のときだけ for 文の先頭に戻る。そのため println は実行されない
            end

            println(i)
```

```
        end
```

```
1
2
3
4
5
6
7
9
10
```

```
In [ ]:
```

目次に戻る

# 4　変数のデータ型

前回までの講義資料中にも整数型や配列型など「型」という言葉が度々出てきましたが、今回はその「型 (データ型) 」について詳しく勉強していきましょう。

データ型とはデータの分類のことです。日常的によく使うものとして、整数型や浮動小数点型などの数値型や文字列型などあらゆるデータは一つの型に分類されます。

## 4.1　静的型付き言語と動的型付き言語

- 静的型付き言語: C 言語や Java などはプログラミング言語のなかでは静的型付き言語という言語に分類されます。静的型付け言語の特徴として、変数使用時にその変数がどのような種類のデータを保存する変数なのかを宣言 (型宣言) する必要が有ります。

  ```
  int x // C 言語での型宣言。x は整数型の変数であることを宣言
  ```

  一度型宣言された変数は他の型のデータを保存することは出来ません。そのため上の例では x に浮動小数点や文字列を入れることは出来ません。
- 動的型付き言語: C 言語や Java と違い、Julia や Python, R, MATLAB などでは変数の型宣言は必要ありません。変数 x に整数型のデータが代入されれば x のデータ型は整数型になりますし、変数 x に文字列が入れば文字列型になります。このように代入されたデータ型に応じて自動的に変数のデータ型が決まるような言語は動的型付き言語と呼ばれます。動的型付き言語は型宣言が必要ないので手軽に書けますが、一方で変数の型をコンピュータが判断しなくてはならないので実行スピードは静的型付き言語に比べると遅くなります。しかし、Julia は JIT compile という方法を採用することによって動的型付き言語なのに C 言語並みの速さが出ます。

```
In [5]: x = 10
        x = "Hello!" # 動的型付きなのでエラーは出ない
```

```
Out[5]: "Hello!"
```

```
In [ ]:
```

目次に戻る

## 4.2 Julia でのデータ型

Julia は動的型付き言語なのでデータ型を意識することなく書けますが意識しても書くことも出来ます。ここでは簡単な紹介程度にとどめますが、詳しく知りたい場合は公式ドキュメントを読みましょう。

Julia では typeof 関数を使うことによって変数のデータ型を確認することが出来ます。

```
In [6]: x = 10
        typeof(x)
```

```
Out[6]: Int64
```

```
In [7]: x = 10.0
        typeof(x)
```

```
Out[7]: Float64
```

```
In [ ]:
```

データ型を自分で明示的に指定したい場合は次のようにします。

```
In [8]: x = Int32(10) # Int32: 単精度整数型
        typeof(x)
```

```
Out[8]: Int32
```

```
In [9]: x = Float32(10) # Float32: 単精度整数型
        typeof(x)
```

```
Out[9]: Float32
```

整数は浮動小数点に変換できますが、逆は出来る場合と出来ない場合があります。10.0 など小数部が 0 の場合は整数に自動的に変換されますが、そうでない場合は自動的には変換されません。整数に変換したい場合は先に小数部を丸める必要があります。

```
In [10]: Int(10.0) # Int はアーキテクチャが 32bit だったら Int32, 64 bit だったら Int64 になる。
```

```
Out[10]: 10
```

```
In [11]: Int(10.5) # InexactError()。整数に変換することは出来ない
```

```
InexactError()
```

```
in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

in (::IJulia.##13#19)() at ./task.jl:360
```

```
In [12]: Int(round(10.5)) # round(Int, 10.5) でも可
```

```
Out[12]: 10
```

```
In [13]: Int(floor(10.5)) # floor(Int, 10.5) でも可
```

```
Out[13]: 10
```

```
In [14]: Int(ceil(10.5)) # ceil(Int, 10.5) でも可
```

```
Out[14]: 11
```

```
In [15]: Int(trunc(10.5)) # trunc(Int, 10.5) でも可
```

```
Out[15]: 10
```

round, floor, ceil, trunc の違いは ?round などとして見るか、公式ドキュメントを参照してください。

```
In [ ]:
```

目次に戻る

## 4.3 型を作る

今までは整数型や文字列型など既存の型を使ってきましたが、型は自分で作ることも出来ます。
型を作る時の基本構文

```
type Mytype
    body
end
```

Julia では型の変数名の頭文字は大文字にするのが慣例です。
例として学生を表す Student 型を作ってみましょう。

```
In [16]: type Student
             statistics::Int # 型を指定しないと Any 型になる
             programming::Int
         end
```

```
In [17]: 太郎 = Student(60, 74)
```

```
Out[17]: Student(60,74)
```

```
In [18]: typeof(太郎)
```

```
Out[18]: Student
```

```
In [19]: 太郎.statistics
```

```
Out[19]: 60
```

```
In [20]: 太郎.programming
```

```
Out[20]: 74

In [21]: 太郎.statistics = 100

Out[21]: 100

In [22]: 太郎 # 値が更新された

Out[22]: Student(100,74)

In [23]: 太郎.statistics = 100.5 # statistics は Int 型と指定したため 浮動小数点を入れることは出来な

         InexactError()



          in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

          in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

          in (::IJulia.##13#19)() at ./task.jl:360

In [ ]:
```

値を更新することができない型を作るには immutable を使います。

```
In [24]: immutable GoodStudent
             statistics::Float64
             programming::Int
         end

In [25]: 勉 = GoodStudent(90.7, 98)

Out[25]: GoodStudent(90.7,98)

In [26]: 勉.statistics = 99.9

         type GoodStudent is immutable



          in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

          in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

          in (::IJulia.##13#19)() at ./task.jl:360
```

In [ ]:

目次に戻る

# 5 関数

プログラムを書いていると計算式は同じだけど値だけが違うという計算が何度も出てきます。こういう時には関数を使って再利用性を上げましょう。

※プログラミングでの関数は数学での関数とは意味が少し異なります。引数を何も入れなくても値を返すこともありますし、引数をいれても何も値を返さないこともあります。

Julia では関数の定義の仕方が以下の 3 通りあります。

```
f(x) = ...
    or

function f(x)
    body
end

    or

x -> ...
```

一行で定義できるようなものは一番上の方法で、body が長くなりそうなら真ん中の方法で、わざわざ関数名をつけるほどでもないときは一番下の方法で定義することが多いです。

In [27]:
```
# 一行書きで関数を定義
# sin を 2 乗するような関数
sin2(x) = sin(x)^2
```

Out[27]:
```
sin2 (generic function with 1 method)
```

In [28]:
```
sin(1), sin(1)^2, sin2(1)
```

Out[28]:
```
(0.8414709848078965,0.7080734182735712,0.7080734182735712)
```

In [29]:
```
function sinpow2(x)
    return sin(x)^2 # return の有無は任意。end の直上にある値が返り値となる
end
```

Out[29]:
```
sinpow2 (generic function with 1 method)
```

In [30]:
```
sinpow2(1)
```

Out[30]:
```
0.7080734182735712
```

In [31]:
```
# 無名関数
# 一度きりしか使わないような場合には便利です。以下の例では使う意味は全くありませんが、map 関数と組み
(x -> sin(x)^2)(1)
```

```
Out[31]: 0.7080734182735712
```

引数には数字だけでなく文字列や配列、関数など何でも入れることが出来ます。引数の数は複数個でも構いません。同様に返り値もなんでも大丈夫です。

```
In [32]: # 数字 a, b を引数にとり、a + b, a - b を返すような関数
         # 一行書きすると plusminus(a, b) = a + b, a - b

         function plusminus(a, b)
             return a + b, a - b
         end
         plusminus(10, 5)
```

```
Out[32]: (15,5)
```

```
In [33]: # フィボナッチ数列の第 n 項を返す関数
         # fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)
         function fib(n)
             if n < 2
                 return n
             end

             a, b, c = 0, 1, 0
             for iter in 2:n
                 c = a + b
                 a = b
                 b = c
             end

             return c
         end
         fib(10)
```

```
Out[33]: 55
```

```
In [34]: # 引数を取らない関数
         function hoge()
             println("hoge")
         end
         hoge()
```

```
hoge
```

```
In [35]: # 引数に関数 f と数字 x をとり、f(x)^2 を返すような関数
         function pow2fn(func, x)
             func(x)^2
```

```
end
pow2fn(sin, 1), sin(1)^2
```

```
Out[35]: (0.7080734182735712,0.7080734182735712)
```

```
In [ ]:
```

目次に戻る

## 5.1 オプション引数

オプション引数を使うと、デフォルトの値を設定が出来ます。
オプション引数

```
function myfunc(x, y, a=1, b=2, ...) # a, b, ... がオプション引数
    body
end
```

```
In [36]: sin_mul_a(x, a=3) = a * sin(x) # オプション引数は普通の引数の後に書く
         sin_mul_a(1), 3 * sin(1)
```

```
Out[36]: (2.5244129544236893,2.5244129544236893)
```

```
In [37]: sin_mul_a(1, 6)
```

```
Out[37]: 5.048825908847379
```

オプション引数は複数入れることも出来ます

```
In [38]: function foo(x, y, a=1, b=3)
             println("x = ", x)
             println("y = $y")
             println("a = ", a)
             println("b = ", b)
         end
```

```
Out[38]: foo (generic function with 3 methods)
```

```
In [39]: foo(1, 3)
```

```
x = 1
y = 3
a = 1
b = 3
```

```
In [40]: foo(1, 3, 10)
```

```
x = 1
y = 3
a = 10
```

```
b = 3
```

```
In [41]: foo(1, 3, 10, 20)
```

```
x = 1
y = 3
a = 10
b = 20
```

```
In [ ]:
```

目次に戻る

## 5.2 キーワード引数

オプション引数はデフォルトの値を設定することができるので便利ですが、数が多くなると大変です。上の例というと b の値だけデフォルトの値とは違う値を使いたい場合でも a の値を入力する必要が有ります。キーワード引数を使うと b = 10 などと明示的に指定することによって b の値だけを変えることが出来ます。キーワード引数を使う場合、; の後にキーワード引数にしたいものを書きます。

キーワード引数

```
function myfunc(x, y, ..., a=1, b=2...; c =3, d=4, ...) # x, y は普通の引数。a, b はオプシ
    body
end
```

```
In [42]: function piyo(x, y, a=1, b=2; c=3, d=4)
             println("x = ", x)
             println("y = $y")
             println("a = ", a)
             println("b = ", b)
             println("c = ", c)
             println("d = ", d)
         end
```

```
Out[42]: piyo (generic function with 3 methods)
```

```
In [43]: piyo(100, 200)
```

```
x = 100
y = 200
a = 1
b = 2
c = 3
d = 4
```

```
In [44]: piyo(100, 200, c=300)
```

```
x = 100
y = 200
a = 1
b = 2
c = 300
d = 4
```

```
In [45]: piyo(100, 200, d=200, c=100) # キーワード引数の順序は任意
```

```
x = 100
y = 200
a = 1
b = 2
c = 100
d = 200
```

```
In [46]: piyo(d=200, 1000, c=300, 2000, 3000, 4000) # キーワード引数がついていない部分から順に x,
                                                    # ただしこのような書き方は可読性が低くなるので
```

```
x = 1000
y = 2000
a = 3000
b = 4000
c = 300
d = 200
```

```
In [ ]:
```

目次に戻る

## 5.3 関数 （発展編）

以下の内容は余裕のある人だけ読んでください。

### 5.3.1 再帰処理

関数定義時に返り値として自分自身を返す関数も作ることが出来ます。これを使うと上のフィボナッチ数列がよりスマートに書けます。

```
In [47]: function fib_rec(n)
             if n < 2 # 終了条件。これがないと無限ループに陥る
                 return n
             else
                 return fib_rec(n-1) + fib_rec(n-2)
             end
```

```
        end
        fib_rec(10)
```

```
Out[47]: 55
```

三項演算子と組み合わせると次のように一行で定義できます

```
In [48]: fib_rec2(n) = n < 2 ? n : fib_rec2(n-1) + fib_rec2(n-2)
         fib_rec2(10)
```

```
Out[48]: 55
```

目次に戻る

### 5.3.2 無名関数

sin 2 乗の例では無名関数の有り難みは全くありませんでしたが、使うと嬉しい例を少し紹介します。
次のように要素が Tuple であるような配列があったとします。

```
In [49]: AA = [(i,j) for (i,j) in zip(rand(1:100,10), rand(1:100,10))]
```

```
Out[49]: 10-element Array{Tuple{Int64,Int64},1}:
          (39,98)
          (51,11)
          (31,44)
          (91,5)
          (78,5)
          (5,18)
          (69,8)
          (52,71)
          (12,42)
          (39,10)
```

この配列から Tuple の第 1 成分を要素とする配列を作りたいと思った時、愚直にやると次のような感じでしょう

```
In [50]: tmparray = zeros(Int, length(AA))
         for i in 1:10
             tmparray[i] = AA[i][1]
         end
         tmparray
```

```
Out[50]: 10-element Array{Int64,1}:
          39
          51
          31
          91
          78
           5
```

```
 69
 52
 12
 39
```

なかなか野暮ったいです。ですが、無名関数と map を使うと以下のようになります。

map の使い方

```
map(func, collection)
```

map 関数は collecttion (配列や Tuple など) の各要素に関数を作用させる

```
In [51]: map(x -> x[1], AA)
```

```
Out[51]: 10-element Array{Int64,1}:
 39
 51
 31
 91
 78
  5
 69
 52
 12
 39
```

だいぶスマートになったのではないでしょうか？

※今の場合なら内包表記でも書けますけどね

```
In [52]: [x[1] for x in AA]
```

```
Out[52]: 10-element Array{Int64,1}:
 39
 51
 31
 91
 78
  5
 69
 52
 12
 39
```

```
In [ ]:
```

次の例として要素数 100 の正規分布に従った乱数の配列を 10 個作り、それをまた別の配列に保存します。

```
In [53]: A = [randn(100) for i in 1:10] # このコマンドの意味がわからなければ内包表記 (comprehensi
                                        # Array{Array{Float64,1},1}: A は配列でその要素も配列
```

```
Out[53]: 10-element Array{Array{Float64,1},1}:
          [-0.0979266,-0.0923629,-0.74266,-0.153902,1.59858,-0.18026,-0.328829,-0.227913
          [0.340565,0.407596,-0.964006,0.32134,-0.810267,1.05251,-0.242473,-0.476823,-0.
          [1.15775,-1.09497,0.562454,1.74505,0.0115943,0.251306,-1.93008,0.37916,-1.0569
          [0.309599,1.43712,-1.27951,-1.43293,0.208912,-0.738032,1.56883,-2.37002,1.2639
          [-0.310866,0.732228,-0.366928,-1.11012,-0.386008,1.02974,0.38998,0.94941,0.138
          [0.251238,0.716822,-0.156129,2.62192,0.266124,0.270463,-0.464465,-0.448382,-1.
          [-1.31056,0.362412,-1.24981,-0.385879,0.867861,0.032985,0.704788,-0.321812,-0.
          [0.0926006,-0.307484,-0.454482,-2.18101,1.66283,-1.56891,0.166988,-0.481405,0.
          [-0.75438,-1.24571,-1.31047,-1.33556,0.788849,0.426939,-1.02286,0.242872,-1.35
          [0.0121549,-0.846848,0.240033,-1.58049,-0.915652,0.823127,-0.252873,-0.292357,
```

配列に var 関数を作用させると分散が計算できます

```
In [54]: var(A[1])

Out[54]: 0.7159354676505525
```

しかし、配列 A に作用させると ...

```
In [55]: var(A) # エラーが出る


        MethodError: no method matching zero(::Type{Array{Float64,1}})
    Closest candidates are:
      zero(::Type{Base.LibGit2.Oid}) at libgit2/oid.jl:88
      zero(::Type{Base.Pkg.Resolve.VersionWeights.VWPreBuildItem}) at pkg/resolve/versio
      zero(::Type{Base.Pkg.Resolve.VersionWeights.VWPreBuild}) at pkg/resolve/versionwei
      ...



          in #var#971(::Bool, ::Void, ::Function, ::Array{Array{Float64,1},1}) at ./stati

          in var(::Array{Array{Float64,1},1}) at ./statistics.jl:176

          in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

          in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

          in (::IJulia.##13#19)() at ./task.jl:360
```

配列 A[1] ～ A[10] までそれぞれの分散を知りたい場合は、map 関数を使う。

```
In [56]: map(var, A) # [var(A[1]), var(A[2]), ..., var(A[10])] と等価

Out[56]: 10-element Array{Float64,1}:
```

```
 0.715935
 0.720194
 0.821421
 1.02328
 0.889928
 0.789865
 0.933594
 0.953192
 1.22336
 0.871913
```

Julia の var 関数は標準で不偏分散を計算しますが、補正のない分散を計算する場合にはキーワード引数で corrected=false とする必要があります。しかし、map 関数ではキーワード引数を取ることが出来ません。このようなとき、わざわざ次のように corrected=false にした関数を作り map 関数を利用するのは不便です。なにより、この場限りでしか使わない関数に名前をつけるのも面倒です。

```
var_no_correction(x) = var(x, corrected=false)
map(var_no_correction, A)
```

このような時に無名関数を使えば、わざわざ関数名を考えずに済みます。

```
map(x -> var(x, corrected=false), A)
```

```
In [57]: var_no_correction(x) = var(x, corrected=false)
         map(var_no_correction, A)
```

```
Out[57]: 10-element Array{Float64,1}:
          0.708776
          0.712992
          0.813206
          1.01304
          0.881029
          0.781966
          0.924258
          0.94366
          1.21112
          0.863194
```

```
In [58]: map(x -> var(x, corrected=false), A)
```

```
Out[58]: 10-element Array{Float64,1}:
          0.708776
          0.712992
          0.813206
          1.01304
          0.881029
          0.781966
```

```
          0.924258
          0.94366
          1.21112
          0.863194
```

まぁ、分散の場合ならば map を使わずとも var.(A) とすれば計算できるのですけど。 この 関数名.(arg...) という使い方は配列演算の項目でまた詳しく説明します。

```
In [59]: var.(A, corrected=false)
```

```
Out[59]: 10-element Array{Float64,1}:
          0.708776
          0.712992
          0.813206
          1.01304
          0.881029
          0.781966
          0.924258
          0.94366
          1.21112
          0.863194
```

```
In [ ]:
```

一応、無名関数と map 関数の説明はしましたが、軽くプログラミングをする程度ならこの 2 つの関数は知らなくても大丈夫です。ただし、スピードを求めて並列化をする場合、map の並列版の pmap という関数があるので、後々並列化に挑戦したいなら覚えておいて損はないです。しかし、並列化はこの講義のレベルを遥かに超えるので、詳しく知りたい人は Julia の公式ドキュメントを読んでください。

```
In [ ]:
```

目次に戻る

### 5.3.3 多重ディスパッチ

Julia では関数の引数の型を明示的に書く必要はありませんが、明示的に指定することも出来ます。明示的に指定することで引数のデータ型によって関数の挙動を変えることができたり、また、実行スピードの向上につながります。この時の型には自分で作ったものも指定することが出来ます。

例として、まずは引数に整数型をとる関数を定義します。

```
In [60]: print_num_type(x::Int) = println("整数です")
```

```
Out[60]: print_num_type (generic function with 1 method)
```

```
In [61]: print_num_type(10) # 整数値を入れると動作します
```

```
整数です
```

```
In [62]: print_num_type(10.9) # 整数以外のものを入れるとエラーが出ます
```

```
    MethodError: no method matching print_num_type(::Float64)
Closest candidates are:
  print_num_type(::Int64) at In[60]:1


     in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

     in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

     in (::IJulia.##13#19)() at ./task.jl:360
```

```
In [63]: print_num_type("Hello") # 整数以外のものを入れるとエラーが出ます
```

```
    MethodError: no method matching print_num_type(::String)
Closest candidates are:
  print_num_type(::Int64) at In[60]:1


     in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

     in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

     in (::IJulia.##13#19)() at ./task.jl:360
```

次に、同じ関数名で浮動小数点を引数にとるように定義します

```
In [64]: print_num_type(x::Float64) = println("浮動小数点です")
```

```
Out[64]: print_num_type (generic function with 2 methods)
```

```
In [65]: print_num_type(10.9) # 先程はエラーが出ましたが、今回はエラーが出ません。
```

```
浮動小数点です
```

```
In [66]: print_num_type(5) # 整数を入れるとエラーは出ず、前と同じ動作になります
```

```
整数です
```

Julia では関数名が同じでも引数の型を変えて定義すれば関数は上書きされません。
関数定義時の出力結果を見ると始めの定義時は

```
print_num_type (generic function with 1 methods)
```

でしたが、2 回目では

```
print_num_type (generic function with 2 method)
```

と 2 methods になっています。これは引数の型によって 2 通りの挙動をすること示しています。
現在、どのような引数に対して定義されているのか確認するには methods 関数を使います。

```
In [67]: methods(print_num_type) # これより Float64, Int64 で定義されていることがわかります。

Out[67]: # 2 methods for generic function "print_num_type":
         print_num_type(x::Float64) at In[64]:1
         print_num_type(x::Int64) at In[60]:1

In [ ]:
```

現在の状態だと引数に 倍精度整数 Int64 の数字を入れることは出来ますが、単精度整数 Int32 などの他の整数は入れることが出来ません。

```
In [68]: print_num_type(Int32(10)) # エラーが出る


        MethodError: no method matching print_num_type(::Int32)
    Closest candidates are:
      print_num_type(::Float64) at In[64]:1
      print_num_type(::Int64) at In[60]:1



         in execute_request(::ZMQ.Socket, ::IJulia.Msg) at /home/tk/.julia/v0.5/IJulia/s

         in eventloop(::ZMQ.Socket) at /home/tk/.julia/v0.5/IJulia/src/eventloop.jl:8

         in (::IJulia.##13#19)() at ./task.jl:360
```

全ての整数に対して動作するようにするためにはより抽象的な型 (abstract type) である Integer 型を使います。今まで使ってきた Int32, Int64 などは具体的な型 (concrete type) と呼ばれ、 Int32, Int64 は 抽象的な型 Integer 型の subtype です。

```
In [69]: Int32 <: Integer # subtype かどうかは Type1 <: Type2 として調べることが出来ます。

Out[69]: true

In [70]: print_num_type(x::Integer) = println("整数です")

Out[70]: print_num_type (generic function with 3 methods)

In [71]: print_num_type(Int32(10))
```

整数です

In [ ]:

目次に戻る

# 6　外部パッケージの利用

Julia では標準機能として平均、分散の計算やフーリエ変換などが出来ますが、外部パッケージを利用することさらに様々なことが出来るようになります。

パッケージを追加するには

```
Pkg.add("パッケージ名")
```

とします。導入できるパッケージは Julia の公式ページより確認することが出来ます。

パッケージのアップデートは

```
Pkg.update()
```

とします。これをすることで、今までインストールしたパッケージ全てをアップデートすることが出来ます。

今回は外部パッケージの一例としてプロット関連のパッケージの導入から簡単な使い方を紹介します。

## 6.1　プロット

プロット関連の有名なパッケージとしては次のようなものが有ります。 - PyPlot: Python の matplotlib を使用している - GR - Plotly

これらのパッケージは作図するまでの文法がそれぞれ違うため、どれか一つの文法を覚えたとしても他の作図パッケージでは使えません。ですが、Plots というパッケージを使うと、文法は変えずにバックエンドを変えることで、ある時は PyPlot で作図し、あるときは GR で作図しと使い分けることが出来ます。

In [72]:
```
# パッケージの追加
# ローカルで Julia 使っている人はコメントを外してコードを実行する。
# JuliaBox を使っている人は標準で入っているため実行する必要はない。

# Pkg.add("Plots"); Pkg.add("GR")
```

### 6.1.1　パッケージの読み込み

追加したパッケージを利用するには

```
import パッケージ名
```

or

```
using パッケージ名
```

とします。これで読み込んだパッケージの関数が使えるようになります。

import と using の違いは、import で読み込んだ場合はパッケージの関数を使うときに

**パッケージ名.関数名**(arg...)

と "パッケージ名." を入れて使用しますが、using で読み込んだ場合は

**関数名**(arg...)

と関数名だけで読み込んだパッケージの関数を使うことが出来ます。(import 同様パッケージ名を入れても使えます) 一見すると import など使わずに常に using を使うのが便利そうです。しかし、複数のパッケージを同時に読み込むと関数名が衝突することが有ります。例えば、PyPlot と GR にはどちらにも plot という関数があるので、どちらのパッケージも using を使って読み込むと plot がどちらのパッケージに依存した関数なのかわからなくなってしまいます。import を使えばパッケージ名を指定するのでこのような関数の衝突はなくなります。

今後、講義資料中ではどの関数がどのパッケージに依存する関数なのか明白にするために、パッケージ名.関数名 (arg...) と書きますが、みなさんの日頃のプログラミングではこうする必要はありません。

```
In [73]: import Plots
         Plots.gr() # バックエンドを GR に指定. PyPlot にしたい場合は pyplot(), plotly を使いたい場
         # Plots.pyplot()
         # Plots.plotly()
```

```
Out[73]: Plots.GRBackend()
```

まずは sin 関数を描写してみましょう。

Plots.jl のプロットの基本文法。 ある関数を $a < x < b$ の範囲でプロットする場合

```
plot(function, a, b)
```

```
In [74]: Plots.plot(sin, -3 π, 3 π)
```

```
In [ ]:
```

関数には自分で定義した関数も使用することが出来ます

```
In [75]: sin3(x) = sin(x)^3 # sin の 3 乗
         Plots.plot(sin3, -3 π, 4 π)
```

```
In [ ]:
```

```
In [76]: # 無名関数はプロットするときにも便利です
         Plots.plot(x -> sin(x)^3, -3 π, 4 π)
```

```
In [ ]:
```

目次に戻る

# 7 練習問題

## 7.1 1.

3 つの数 a, b, c を入力すると平均値を返すような関数を作成せよ

出力例

```
avg3(10, 20, 30) # -> 20.0
```

In [ ]:

## 7.2 2.

```
x = [74,51,98,27,29,2,40,75,75,12]
```

のような配列を引数にとり、要素の平均を出力する関数 mymean を作成せよ. (mean 関数は使ってはいけない)

出力例

```
mymean(x) #-> 48.3
```

In [ ]:

## 7.3 3.

半径 r の球の表面積と体積を計算する関数を作成せよ

出力例

```
areavol(4) #-> (201.06192982974676,268.082573106329)
```

In [ ]:

## 7.4 4.

西暦を入力すると、うるう年かどうか判定する関数を作成せよ

うるう年 - 西暦年が 4 で割り切れる年は閏年。 - ただし、西暦年が 100 で割り切れる年は平年。 - ただし、西暦年が 400 で割り切れる年は閏年。

Wikipedia より

出力例

```
leap(2000)
2000 年はうるう年です


leap(2100)
2100 年はうるう年ではありません
```

In [ ]:

## 7.5 5.

$y = x^2 + 3$ を $-3 < x < 3$ の範囲で図示せよ

In [ ]:

## 7.6　6.

福岡市の平均気温を調べ、横軸を月、縦軸を平均気温として図示せよ

In [ ]:

目次に戻る