# プログラミング言語Elixir

# 資料の目的

- Elixir を触ってみることができるような気分になること
- Elixir でやりたいことがあったとき，どの辺を眺めたり探せばよいか，推測できるようになること

# Elixirの特徴

Elixir の特徴 - Qiita
http://qiita.com/niku/items/7c61d6a6af38896ac603

# Elixirの特徴

- スケールしやすさ
- 対障害性
- 関数型プログラミング
- 拡張しやすさと DSL
- 成長するエコシステム
- 対話型開発
- Erlang 互換

# ぼくがえりくさーですきなとこ

- なじみのある文面 <- [今日これ]
- ErlangVMで処理

# インストール

Installing Elixir - Elixir
http://elixir-lang.org/install.html

- OSX
- Unix
- Windows

# iexコマンド

IEx

http://elixir-lang.org/docs/stable/iex/IEx.html

- `iex` というコマンドで REPL が使える
- `iex` 内で打ち間違えたとき `#iex:break` と打つとやりなおせる

# 値

Basic types - Elixir

http://elixir-lang.org/getting-started/basic-types.html

# 値

```
1                 # 整数
0x1F              # 整数
1.0               # 小数
true              # ブーリアン
:atom             # アトム
"elixir"          # 文字列
[1, 2, 3]         # リスト
{1, 2, 3}         # タプル
```

# 整数

```
1                 # 整数 <-
0x1F              # 整数 <-
1.0               # 小数
true              # ブーリアン
:atom             # アトム
"elixir"          # 文字列
[1, 2, 3]         # リスト
{1, 2, 3}         # タプル
```

# 整数

```
1                # => 1
0x1F             # => 31
1 + 2            # => 3
5 * 5            # => 25
10 / 2           # => 5.0
div(10, 2) # => 5
rem(10, 3) # => 1
```

# 整数

```
# 同じものを指す
div(10, 2)          # => 5
div 10, 2           # => 5
Kernel.div(10, 2) # => 5
# 同じものを指す
1 + 2               # => 3
Kernel.+(1, 2)      # => 3
Kernel.+ 1, 2       # => 3
```

# Elixirの関数

- Elixir では(ErlangVMでは) **モジュール名.関数名/引数の数** で関数を指定する
- 例えば `Kernel.div(10, 2)` は `Kernel.div/2`

# 関数の調べかた

1 + 1 の + について知りたい場合

- 1 + 1 は Kernel.+(1, 1) のこと
- モジュールは Kernel 関数は + 引数の数は 2 つだから Kernel.+/2 だな
- iex から h Kernel.+/2 とするとドキュメントが引ける

# 関数の調べかた

```
iex(1)> h Kernel.+/2

                          def +(left, right)

Arithmetic plus.

Allowed in guard tests. Inlined by the compiler.

Examples

┃ iex> 1 + 2
┃ 3
```

# 関数の調べかた

h Kernel.+ と引数の数を省略すると，Kernel.+ で，引数の数を問わないで探す

```
iex(3)> h Kernel.+
                                    def +(value)

(省略)
                                  def +(left, right)

(省略)
```

# 関数の調べかた

web から

Elixir v1.0.4 Documentation
http://elixir-lang.org/docs/stable/elixir/

web 日本語版( @k1complete さん作)

elixirリファレンスのページ
http://ns.maqcsa.org/elixir/docs/

# 関数の調べかた

[Elixir - iexでの日本語版ヘルプの使い方 - Qiita
http://qiita.com/k1complete/items/511ef32b63869bc48d02](http://qiita.com/k1complete/items/511ef32b63869bc48d02)

h で表示する内容を日本語にもできるみたい(試していない)

# 整数を操作する関数

- Kernel に `+/2 -/2 div/2` などがある
- Integer に `is_odd/1` などがある

# 小数

```
1                 # 整数
0x1F              # 整数
1.0               # 小数 <-
true              # ブーリアン
:atom             # アトム
"elixir"          # 文字列
[1, 2, 3]         # リスト
{1, 2, 3}         # タプル
```

# 小数

```
1.0                          # => 1.0
1.0e-5                       # => 1.0e-5
1.0e-5 === 0.00001 # => true
round(3.58)                  # => 4
trunc(3.58)                  # => 3
```

# 小数を操作する関数

- Kernel に `round/1` `trunc/1` などがある
- Float に `ceil/1` や `foor/1` などがある

# ブーリアン

```
1                 # 整数
0x1F              # 整数
1.0               # 小数
true              # ブーリアン <-
:atom             # アトム
"elixir"          # 文字列
[1, 2, 3]         # リスト
{1, 2, 3}         # タプル
```

# ブーリアン

```
true  # => true
false # => false
!true # => false
```

# ブーリアンを操作する関数

みつからなかった！

# アトム

```
1                 # 整数
0x1F              # 整数
1.0               # 小数
true              # ブーリアン
:atom             # アトム <-
"elixir"          # 文字列
[1, 2, 3]         # リスト
{1, 2, 3}         # タプル
```

# アトム

自分の名前が自分の値を表すような定数．他の言語だとシンボルと呼ばれるようなもの．

```
:foo                    # => :foo
:"foo-bar"              # => :"foo-bar"
# ブーリアン値はアトムでした
:true === true    # => true
:false === false # => true
```

# アトムを操作する関数

- Atom にシンボルから文字列に変換する関数がある

# 文字列

```
1                 # 整数
0x1F              # 整数
1.0               # 小数
true              # ブーリアン
:atom             # アトム
"elixir"          # 文字列 <-
[1, 2, 3]         # リスト
{1, 2, 3}         # タプル
```

# 文字列

Binaries, strings and char lists - Elixir
http://elixir-lang.org/getting-started/binaries-strings-and-char-lists.html

- Elixir では“ でくくった文字列と‘ でくくった文字列がある
- ほぼ全ての場合において“ の方を利用する

# 文字列

```
"abc"                              # => "abc"
"こんにちは"                       # => "こんにちは"
"1 + 2 は #{ 1 + 2 } です" # => "1 + 2 は 3 です"
# Elixir では文字列はバイナリとして扱われる
is_binary("abc")                   # => true
```

# 文字列を操作する関数

```
byte_size("日本語")      # => 9
String.length("日本語") # => 3
String.at("日本語", 1)  # => "本"
```

- Kernel にバイナリを操作する関数がある
- String モジュールに，UTF-8 エンコードされているバイナリをうまく扱う関数がある

# 複数の値を格納する値

```
1                 # 整数
0x1F              # 整数
1.0               # 小数
true              # ブーリアン
:atom             # アトム
"elixir"          # 文字列
[1, 2, 3]         # リスト <-
{1, 2, 3}         # タプル <-
```

# 複数の値を格納する値

- Basic types - Elixir
  http://elixir-lang.org/getting-started/basic-types.html#%28linked%29-lists
- Keywords, maps and dicts - Elixir
  http://elixir-lang.org/getting-started/maps-and-dicts.html

# 複数の値を格納する値

```
[1, 2, 3]                          # リスト <-
{1, 2, 3}                          # タプル
%{foo: "hoge", bar: "fuga"} # マップ
[{:foo, "hoge"},
 {:bar, "fuga"},
 {:foo, "moge"}]                   # キーワードリスト
```

# リスト

```
[1,2,3]              # => [1, 2, 3]
[:a, "b", 'c']       # => [:a, "b", 'c']
[[:x], [:y, :z]] # => [[:x], [:y, :z]]
```

- [] でくくる
- いわゆる配列．値はなんでも，何個でも入る

# リストを操作する関数

```
Enum.map([1,2,3], fn(x) -> x * 2 end) # => [2, 4, 6]
List.last([4,5,6])                    # => 6
```

- Enum には，繰り返しについての関数がある
- List には，List特有の関数がある

まず Enum を探すのがよい

# タプル

```
[1, 2, 3]                          # リスト
{1, 2, 3}                          # タプル <-
%{foo: "hoge", bar: "fuga"} # マップ
[{:foo, "hoge"},
 {:bar, "fuga"},
 {:foo, "moge"}]                   # キーワードリスト
```

# タプル

```
{:a, 1}           # => {:a, 1}
{"x", "y", "z"} # => {"x", "y", "z"}
```

- {} でくくる
- 中に入る個数が決まった入れ物

# タプルを操作する関数

```
elem({:a, "abc"}, 1)                    # => abc
Tuple.delete_at({"x", "y", "z"}, 1) # => {"x", "z"}
```

- Kernel にはタプルの値を取得する関数がある
- Tuple にはタプル特有な関数がある

# マップ

```
[1, 2, 3]                          # リスト
{1, 2, 3}                          # タプル
%{foo: "hoge", bar: "fuga"} # マップ <-
[{:foo, "hoge"},
 {:bar, "fuga"},
 {:foo, "moge"}]                   # キーワードリスト
```

# マップ

```
%{:a => 1, :b => 2}                 # => %{a: 1, b: 2}
%{a: 1, b: 2}                       # => %{a: 1, b: 2}
%{:a => 1, :b => 2, :a => 3} # => %{a: 3, b: 2}
```

- %{} でくくる
- キーと，それに対応する値を一組に持ついわゆるキーバリューストア
- キーは重複して定義できず，上書きされる

# マップを操作する関数

```
Enum.any?(%{a: 1, b: 2}, fn({k,_v}) -> k === :b end) # => true
Dict.update!(%{a: 1, b: 2}, :a, fn(v) -> v + 10 end) # => %{a: 11, b: 2}
Map.new                                              # => %{}
```

- Enum には，繰り返しについての関数がある
- Dict には，とあるキーととある値が関連づいているようなデータについての関数がある
- Map には，Map 特有の関数がある

# キーワードリスト

```
[1, 2, 3]                          # リスト
{1, 2, 3}                          # タプル
%{foo: "hoge", bar: "fuga"} # マップ
[{:foo, "hoge"},
 {:bar, "fuga"},
 {:foo, "moge"}]                   # キーワードリスト <-
```

# キーワードリスト

```
[{:foo, "x"}, {:bar, "y"}]              # => [foo: "x", bar: "y"]
[foo: "x", bar: "y"]                    # => [foo: "x", bar: "y"]
[{:foo, "x"}, {:bar, "y"}, {:foo, "z"}] # => [foo: "x", bar: "y", foo: "z"]
```

- 1 つめの要素がアトム，2 つめの要素が任意の値になっているタプルを持つ配列
- Map とは異なり，同じ名前のキーを 2 つ保持することができる
- キーワード引数としての利用が多い

# キーワードリストを操作する関数

```
Keyword.get_values([foo: "x", bar: "y", foo: "z"], :foo) # => ["x", "z"]
```

特定のユースケースで多用されるため，Keyword モジュールにある関数で処理することが多い

- Enum / List / Dict 利用可能
- Keyword には，キーワードリスト特有の処理がある

# モジュールと関数の定義

Modules - Elixir
http://elixir-lang.org/getting-started/modules.html

# モジュールと関数の定義

```
defmodule MyOperand do
  def plus(x, y) do
    x + y
  end

  def minus(x, y) do
    x - y
  end
end
MyOperand.plus(1, 2)  # => 3
MyOperand.minus(5, 3) # => 2
```

# モジュールと関数の定義

- モジュールは，関数をあるグループにまとめて，探しやすくするためにある
- 自分達でモジュールを定義するには `defmodule` を使う
- 関数の定義は `def` を使う

# do-endはキーワードリストで実装されている

case, cond and if - Elixir
http://elixir-lang.org/getting-started/case-cond-and-if.html#do/end-blocks

# do-endはキーワードリストで実装されている

```
if true do
  "foo"
else
  "bar"
end
if(true, do: "foo", else: "bar")
if(true, [{:do, "foo"}, {:else, "bar"}])
```

- Elixir の do xxx end は do: xxx というキーワード引数のシンタックスシュガー

# do-endはキーワードリストで実装されている

```
defmodule SingleLine
  def bar do: "hoge"
  def baz do: "fuga"
end
```

- 簡単な関数定義のときは def を 1 行で書くこともできる

# |>

```
[1,2,3,4,5,6]                 # => [1,2,3,4,5,6]
  .map { |e| e + 1 }          # => [2,3,4,5,6,7]
  .select { |e| e.odd? } # => [3,5,7]
  .select { |e| 3 < e }  # => [5,7]
```

- データに対して処理を連続して行う(Rubyの場合)
- オブジェクトに関係つけている関数をオブジェクト経由で呼び出している

# |>

```
array  = [1,2,3,4,5,6]                                    # => [1,2,3,4,5,6]
mapped = Enum.map(array,     fn(x) -> x + 1 end)          # => [2,3,4,5,6,7]
odd    = Enum.filter(mapped, fn(x) -> rem(x, 2) == 1 end) # => [3,5,7]
over3  = Enum.filter(odd,    fn(x) -> 3 < x end)          # => [5,7]
over3                                                     # => [5,7]
```

- Elixir では，データに対しては関数が関係ついていない
- データに対して処理を連続して行いたいときは，返り値を次の関数へ引数として渡すことになる

# |>

```
Enum.filter(
  Enum.filter(
    Enum.map([1,2,3,4,5,6], fn(x) -> x + 1 end),
    fn(x) -> rem(x, 2) == 1 end
  ),
  fn(x) -> 3 < x end
) # => [5,7]
```

- インライン化して変数を消すことができた
- 処理順が上から下ではなく括弧の内側から外側になってしまった……

# |>

```
[1,2,3,4,5,6]
|> Enum.map(fn(x) -> x + 1 end)
|> Enum.filter(fn(x) -> rem(x, 2) == 1 end)
|> Enum.filter(fn(x) -> 3 < x end)
# => [5,7]
```

- Elixirには |> という演算子がある
- |> は，左(上)側の評価結果を，右側の関数の第一引数へと代入してくれる

# |>

```
"hoge" |> String.upcase |> String.replace("H", "M") # => MOGE

x1 = "hoge"
x2 = String.upcase(x1)
x3 = String.replace(x2, "H", "M") # => MOGE
```

- 上と下の式はほとんど同じ意味を持っている

# |>

- 「subject は第一引数に取る」という鉄則があるので，|> でデータを繋いでいくことができる
- 自分で関数を定義するときは，操作対象を第一引数に取るようにすると，Elixir Way に乗って |> を使いやすい

# 束縛

```
x = 1
x           # => 1
y = :abc
y           # => :abc
# ↑実質y1
y = :def
y           # => :def
# ↑実質y2
```

# パターンマッチング

```
[h|t] = [1,2,3]
h # => 1
t # => [2,3]
{x, y} = {123, 456}
x # => 123
y # => 456
%{i: a} = %{i: "あい", j: "じぇい", k: "けい"}
a # => "あい"
%{i: b} = %{j: "じぇい", k: "けい"}
# => (MatchError) no match of right hand side value: %{j: "じぇい", k: "けい"}
```

- 複数の値を格納している値を，複数の変数に結びつけることができる

# パターンマッチング

Pattern matching - Elixir
http://elixir-lang.org/getting-started/pattern-matching.html

# 文字へのパターンマッチング

```
<<h, t :: binary>> = "abcdef"
h                  # => 97
t                  # => "bcdef"
<<97>> === "a" # => true
```

- 文字にも同じようにパターンマッチングできる

# 匿名関数

- Basic types - Elixir http://elixir-lang.org/getting-started/basic-types.html#anonymous-functions

# 匿名関数

```
add = fn(x, y) -> x + y end
add.(2, 3) # => 5
```

# 匿名関数

```
x1 = Enum.map([1,2,3], fn (x) -> x + 5 end)
x1 # => [6, 7, 8]
x2 = Enum.map([1,2,3], &(&1 + 5))
x2 # => [6, 7, 8]
```

- 関数(今回の例だと Enum.map )に匿名関数を渡して，やりたい事を書くときに使うことがほとんど
- fn(x) -> x + 5 end は &(&1 + 5) と書ける

# 制御構造

case, cond and if - Elixir
http://elixir-lang.org/getting-started/case-cond-and-if.html

# 制御構造

```
if true do
  "true"
else
  "false"
end
# => "true"
```

- Elixir で false として扱われるのは，　false と nil だけ

# 制御構造

```
x = if true do
      "**true**"
    else
      "**false**"
    end
x # => **true**
```

- IF 式は返り値を持つので，結果を変数に束縛することもできる

# 制御構造

```
x = 2
case x do
  1 -> "x => 1"
  2 -> "x => 2"
end
# => "x => 2"
```

- true / false 以外で分岐させたい場合は case を使う

# 制御構造

```
x = 2; y = 3; z = 2
case x do
  y -> "x === y, #{y}"
  z -> "x === z, #{z}"
end
# => "x === y, 2"
```

- x と z が等しいので “x = z” の方を期待していた

# 制御構造

```
x = 2; y = 3; z = 2
case x do
  ^y -> "x === y, #{y}"
  ^z -> "x === z, #{z}"
end
# => "x === z, 2"
```

- y の値が変わってほしくないという意思を伝える書き方 ^y がある

# 制御構造

```
x = 3
case x do
  1 -> "x => 1"
  2 -> "x => 2"
end
# ** (CaseClauseError) no case clause matching: 3
#     elixir_src.exs:2: (file)
#     (elixir) lib/code.ex:307: Code.require_file/2
```

- x に 3 がきたときエラーでプロセスが落ちてしまった

# 制御構造

```
x = 3
case x do
  1 -> "x => 1"
  2 -> "x => 2"
  _ -> "another x"
end
# => "another x"
```

- 最後に true か _ を使ってマッチングすると未知の値も拾える

# 制御構造

しかし！ErlangVM では “Let it crash” という哲学があり，
未知の値は無理にハンドリングせずに落としてしまう方が好ましい．

# 制御構造

プロセスは落ちてしまう．プロセスが落ちるとプログラム自体が終わる！と思うかもしれないがErlangVM ではプロセスが落ちてもプログラム自体を終わらせない方法が標準で用意されている．

# 制御構造

今回は取り上げないが OTP や Supervisor について調べるとよい.
ひとまずここでは「予想できていること, 状態についてだけ扱う」ということだけ覚えておいてほしい.

# 制御構造

```
x = 2
y = 3
result = case {x, y} do
           {1, 2} -> :a
           {1, 3} -> :b
           {2, a} -> a
           {3, _} -> :c
         end
result # => 3
```

# 制御構造

```
x = 1
result = case x do
           a when is_integer(x) -> a
           a when is_float(x)   -> trunc(a)
           a when is_binary(x)  -> String.to_integer(a)
         end
result # => 1
```

# 制御構造

```
x = 1.1
result = case x do
           a when is_integer(x) -> a
           a when is_float(x)   -> trunc(a)
           a when is_binary(x)  -> String.to_integer(a)
         end
result # => 1
```

# 制御構造

```
x = "1"
result = case x do
           a when is_integer(x) -> a
           a when is_float(x)   -> trunc(a)
           a when is_binary(x)  -> String.to_integer(a)
         end
result # => 1
```

# 制御構造

[case, cond and if - Elixir http://elixir-lang.org/getting-started/case-cond-and-if.html#expressions-in-guard-clauses](http://elixir-lang.org/getting-started/case-cond-and-if.html#expressions-in-guard-clauses)

- guard という方法を使えば，異なる型，例えば整数の 1 と小数点つきの数 1.1，文字列 “1” を扱うこともできる

# 今日やったこと

- インタラクティブシェル iex
- ドキュメントの調べかた
- さまざまな値と，値を操作する関数
- パイプ演算子 |>
- パターンマッチング
- 制御構造

# 簡単なElixirのはじめかた

- 毎週木曜日にやっている sapporo.beamで聞こう
- http://sapporo-beam.github.io/
- オンライン参加も歓迎しております！