2015年2月24日

Python 入門 関数型プログラミング

関数型プログラミング入門

SOURCE

An introduction to functional programming （2014-12） by Mary Rose Cook

本記事は、原著者の許諾のもとに翻訳・掲載しております。

多くの関数型プログラミングに関する記事が教えてくれるのは、抽象的な関数型のテクニックです。つまり関数合成やパイプライン、高階関数などです。この記事では違います。ここで

- 翻訳リクエストを送る
- 翻訳フィードバック
  - メール
  - GitHub Issue

最近の投稿

- 私たちはいかにして環状線で"悪さをする列車"を捕まえたか
- ーから学ぶベジェ曲線
- 型クラスはインターフェースとどう違うのか
- どれだけ速く文字列からスペースを削除できるのか
- Node.jsのパフォーマンス最適化を阻むものの見つけ方

タグ

ABテスト

AngularJS AWS

は、プログラマが毎日書く、命令型で非関数型のコードの例を示し、それを関数型の形式へ書き換えます。

最初のセクションでは、短いデータ変換のループを取り上げ、map関数やreduce関数に書き換えていきます。2つ目のセクションではより長いループを取り上げ、ユニットに分解し、それぞれのユニットを関数型に書き換えます。3つ目のセクションでは、連続した長いデータ変換のループを関数型のパイプラインに分解します。

ここではPythonでの例を取り扱います。というのも多くのプログラマはPythonを読むのは簡単だと思っているからです。多くの例では、mapやreduce、パイプラインなどの多くの言語に共通する機能を例示するため、Python的なものは避けます。

ガイドロープ

プログラマが関数型プログラミングについて語るとき、めまいを起こしそうな数の“関数型”的な特徴について言及します。彼らは変更不能なデータ[1]や第一級関数[2]や末尾呼び出し最適化[3]について述べます。これらは関数型プログラミングを助ける言語の特徴です。そして彼らはマッピング、リデューシング、パイプライン化、再帰法、カリー化[4]、そして高階関数の使用法についての話に移ります。これらは関数型のコードを書くのに使われるプログラミングテクニックです。それからさらに並列化[5]や遅延評価[6]、決定性[7]について言及するでしょう。これらは、関数型プログラムの長所です。

これらを全て忘れてください。関数型コードを特徴づけるのは、「副作用がない」という1点で

C++ Clojure CSS
C言語 Docker
D言語 EdTech
Elm Erlang Git
Github Google
Go言語 Haskell
iOS Java
JavaScript jQuery
Lisp MySQL
NodeJS NoSQL
Objective-C
OCaml PHP
podcast
PostgreSQL
Python React
Ruby Rust Scala
SEO SSL
Stack Overflow
Swift TDD
TypeScript
UIデザイン
Unix/Linux
webサーバ
Y Combinator
アジャイル
アルゴリズム
エコシステム
エンジニア採用
オープンソース
キャリアパス
グロースハック
ゲーム開発
コードレビュー
セキュリティ
ソフトウェアアーキテクチャ
データサイエンス

す。関数型のコードは現在の関数の外部に存在するデータに頼りません。そして現在の関数の外部に存在するデータを変更しません。
その他の“関数型”なものは、全てこの特性から派生しています。このことを学習の手がかりにしてください。

これは非関数型の関数です。

```
1. a = 0
2. def increment1():
3.     global a
4.     a += 1
```

これは関数型の関数です。

```
1. def increment2(a):
2.     return a + 1
```

リストをイテレートせず、mapとreduceを使う

Map

Mapは、関数とアイテムのコレクションを引数にとります。mapは新しい空のコレクションを生成し、オリジナルのコレクションの各アイテムに関数を実行します。
そして各々の戻り値を新しいコレクションに挿入し、新しいコレクションを返します。

これは、名前のリストを受け取り、それらの名前の長さのリストを返すシンプルなmapです。

```
1. name_lengths = map(len, ["Mary", "Isla",
   "Sam"])
2.
3. print name_lengths
4. # => [4, 4, 3]
```

これは、渡されたコレクションの中での全ての

データベース
デバッグ ネタ
ハッカソン
パフォーマンス
ビジネスモデル
ビジュアライゼーション
ビジュアルデザイン
フリーランス
フレームワーク
プログラミング言語比較
プロジェクト管理
ベストプラクティス
マイクロサービス
まとめ
リファクタリング
入門 型システム
機械学習 生産性
経営組織論 統計
起業アイデア
起業家
関数型プログラミング
顧客開発

アーカイブ
2017年02月
2017年01月
2016年12月
2016年11月
2016年10月
2016年09月
2016年08月
2016年07月
2016年06月
2016年05月
2016年04月
2016年03月
2016年02月

数を二乗するmapです。

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])

print squares
# => [0, 1, 4, 9, 16]
```

このmapは、名前付き関数を引数にとりません。その代わり、ラムダで定義される無名のインライン関数をとります。ラムダのパラメータは、コロンの左に定義されます。関数の本体は、コロンの右に定義されます。関数の本体を実行させた結果は（暗黙的に）返されます。

下記の非関数型のコードは、本名のリストを受け取って、ランダムに割り当てられたコードネームと入れ替えます。

```
import random

names = ['Mary', 'Isla', 'Sam']
code_names = ['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']

for i in range(len(names)):
    names[i] = random.choice(code_names)

print names
# => ['Mr. Blonde', 'Mr. Blonde', 'Mr. Blonde']
```

（見てわかるとおり、このアルゴリズムは同じ秘密のコードネームを複数の諜報員に割り当てる可能性があります。秘密の任務に混乱をもたらさなければよいのですが）。

これはmapとして書き換えることができます。

```
import random

names = ['Mary', 'Isla', 'Sam']

secret_names = map(lambda x:
random.choice(['Mr. Pink',
```

2016年01月
2015年12月
2015年11月
2015年10月
2015年09月
2015年08月
2015年07月
2015年06月
2015年05月
2015年04月
2015年03月
2015年02月
2015年01月
2014年12月
2014年11月
2014年10月
2014年09月
2014年08月
2014年07月
2014年06月

```
6.
   'Mr. Orange',
7.
   'Mr. Blonde']),
8.                    names)
```

演習1. 以下のコードをmapとして書き換えてみましょう。このコードは本当の名前のリストを受け取り、より強力な戦略を用いて生み出されるコードネームと入れ替えます。

```
1. names = ['Mary', 'Isla', 'Sam']
2.
3. for i in range(len(names)):
4.     names[i] = hash(names[i])
5.
6. print names
7. # => [6306819796133686941,
   8135353348168144921, -1228887169324443034]
```

（諜報員達が抜群の記憶力で、任務中にお互いのコードネームを忘れずにいられることを祈ります）。

私の答え：

```
1. names = ['Mary', 'Isla', 'Sam']
2.
3. secret_names = map(hash, names)
```

Reduce

Reduceは関数とアイテムのコレクションを引数にとります。そしてアイテム同士を組み合わせて生成される戻り値を返します。

これはシンプルなreduceです。コレクションの全てのアイテムの合計を返します。

```
1. sum = reduce(lambda a, x: a + x, [0, 1, 2,
   3, 4])
2. print sum
3. # => 10
```

x は、イテレートされていく現在のアイテムです。 a は累計値です。前のアイテムに対するラムダの処理の戻り値です。 reduce() はアイテムを渡り歩きます。それぞれのアイテムについて、現在の a と x にラムダを実行し、結果を次の a として返します。
それでは最初の値としての a は、なんなのでしょうか？　前の繰り返しの結果がありません。 reduce() は1回目の繰り返しではコレクションの最初のアイテムを a として使い、2番目のアイテムから繰り返しを開始します。つまり最初の x は2つ目のアイテムなのです。
このコードは‘Sam’という単語が文字列のリストで何度現れるかカウントします。

```
sentences = ['Mary read a story to Sam and Isla.',
             'Isla cuddled Sam.',
             'Sam chortled.']

sam_count = 0
for sentence in sentences:
    sam_count += sentence.count('Sam')

print sam_count
# => 3
```

同じコードをreduceを使って書くとこうなります。

```
sentences = ['Mary read a story to Sam and Isla.',
             'Isla cuddled Sam.',
             'Sam chortled.']

sam_count = reduce(lambda a, x: a + x.count('Sam'),
                   sentences,
                   0)
```

このコードでは最初の a はどのように判断するのでしょう？　‘Sam’の出現回数の初期値が、‘Maryは、SamとIslaに物語を読んで聞かせ

ました'という文であるはずはありません。最初の累積値は `reduce()` の3番目の引数で指定されます。ここでは、コレクションのアイテムと異なる型の値を使用することが可能です。

なぜmapとreduceの方がよいのか？

1つは、どちらも1行で完結することが多いからです。

2つ目は繰り返しの重要な部分であるコレクション、操作、そして戻り値が、同じ場所にあるからです。

3つ目に、ループのコードは、それ以前に定められる変数または実行後のコードに影響を及ぼすことがあるからです。習慣的に、mapとreduceは関数型です。

4つ目にmapとreduceは基礎的な操作だからです。どのプログラマもループを読むときには、1行1行ロジックを辿っていかなければなりません。彼らがコードを理解するための足がかりにできるような構造的な規則性は、ほぼありません。対照的にmapとreduceは、複雑なアルゴリズムを組み上げるための建築用ブロックとして使用することができ、コードの読み手がすぐに理解し、要約することができる要素となります。「なるほど、このコードは、このコレクションの各々のアイテムを変換していて、そのうちいくつかは削除している。そして残りを結合して1つの出力にしているのだな」。

5つ目に、mapとreduceは彼らの基本的なふるまいをより便利に、精細にしたバージョンを提供する多くの仲間を持っていることです。例えばfilterやall、any、findなどです。

演習2. 以下のコードをmap、reduce、filterを用いて書き直してみてください。filterは関数とコレクションを引数にとり、関数が `True` を返した全てのアイテムのコレクションを返します。

```
people = [{'name': 'Mary', 'height': 160},
    {'name': 'Isla', 'height': 80},
    {'name': 'Sam'}]

height_total = 0
height_count = 0
for person in people:
    if 'height' in person:
        height_total += person['height']
        height_count += 1

if height_count > 0:
    average_height = height_total / height_count

    print average_height
    # => 120
```

複雑に見えるようであるならば、データに対する操作については考えないでください。人のディクショナリから平均身長までのデータの移り変わりについて考えてみてください。複数の変換をまとめて考えないようにしてください。
各々を別個に考えて、結果を名前通りの変数に割り当ててください。コードが機能したら、それを短縮してください。
私の答え：

```
people = [{'name': 'Mary', 'height': 160},
          {'name': 'Isla', 'height': 80},
          {'name': 'Sam'}]

heights = map(lambda x: x['height'],
              filter(lambda x: 'height' in x, people))

if len(heights) > 0:
    from operator import add
    average_height = reduce(add, heights) / len(heights)
```

命令的にでなく、宣言的に書く

下記のプログラムでは、3台の車のレースを行います。各々の時間ステップで、各々の車は前に動くか停止します。各々の時間ステップで、プログラムはここまでの車の進路を表示します。5回のステップの後、レースは終わります。
これは出力の例です。

```
-
- -
- -

- -
- -
- - -

- - -
- -
- - -

- - - -
- - -
- - - -

- - - -
- - - -
- - - - -
```

これがプログラムです。

```
from random import random

time = 5
car_positions = [1, 1, 1]

while time:
    # decrease time
    time -= 1

    print ''
    for i in range(len(car_positions)):
        # move car
        if random() > 0.3:
            car_positions[i] += 1

```

```
16.         # draw car
17.         print '-' * car_positions[i]
```

このコードは命令型で書かれています。関数型バージョンは宣言的でなければなりません。どうやってやるかよりもなにをすべきかを書きましょう。

関数を使う

プログラムはコードを関数で束ねることによって、より宣言的になります。

```
1. from random import random
2.
3. def move_cars():
4.     for i, _ in enumerate(car_positions):
5.         if random() > 0.3:
6.             car_positions[i] += 1
7.
8. def draw_car(car_position):
9.     print '-' * car_position
10.
11. def run_step_of_race():
12.     global time
13.     time -= 1
14.     move_cars()
15.
16. def draw():
17.     print ''
18.     for car_position in car_positions:
19.         draw_car(car_position)
20.
21. time = 5
22. car_positions = [1, 1, 1]
23.
24. while time:
25.     run_step_of_race()
26.     draw()
```

このプログラムを理解するには、メインのループを読めば事足ります。“残り時間があるならば、レースのステップを実行して、結果を描け。絵を描いてください。そしてもう一度時間をチェックしろ” もし読み手が「レースを1ステップ進める」とか、「描画する」というのがど

ツノ進める」とか、「描画する」というのがどういうことなのか詳しく知りたければ、それぞれの関数のコードを読めばよいのです。

もうコメントは必要ありません。コード自身が語ってくれます。

コードを関数で分割するのは素晴らしいことです。コードの可読性を上げるために頭を悩ませる必要もありません。しかしこのテクニックでは関数を使用していますが、あたかもそれらをサブルーチンのようにして使っています。関数群によってコードが切り分けられていますが、このコードは、ガイドロープで書いたような意味での関数型のものではありません。このコードの関数では、引数として渡された以外の状態を使用しているからです。関数は値を返すのではなく、外部の変数を書き換えて、周辺のコードに影響を与えてしまっています。関数が実際に何をしているのかを確認するためには、読み手は各行を慎重に読まなければなりません。そこでもし外部の変数を見つけたら、その変数の出所を探す必要があります。他の関数がその変数を書き換えているのを発見するに違いありません。

宣言を除去する

こちらは関数型バージョンの車レースのコードです。

```
1. from random import random
2.
3. def move_cars(car_positions):
4.     return map(lambda x: x + 1 if random() >
   0.3 else x,
5.                car_positions)
6.
7. def output_car(car_position):
8.     return '-' * car_position
```

```
9.
10. def run_step_of_race(state):
11.     return {'time': state['time'] - 1,
12.             'car_positions':
    move_cars(state['car_positions'])}
13.
14. def draw(state):
15.     print ''
16.     print '\n'.join(map(output_car,
    state['car_positions']))
17.
18. def race(state):
19.     draw(state)
20.     if state['time']:
21.         race(run_step_of_race(state))
22.
23. race({'time': 5,
24.       'car_positions': [1, 1, 1]})
```

コードはやはり関数で分割されていますが、使われている関数は関数型になっています。その証拠が3つあります。1つは、共有されている変数がないということです。 `time` や `car_position` は順番に渡されていき、 `race()` で使用されます。次に、関数が引数を持っていることです。そして3つ目は、関数の内部で生成される変数がないということです。全てのデータの書き換えはreturn文で済んでいます。

`race()` は `run_step_of_race()` の結果を使って再帰的に呼び出されます。各ステップの結果は即座に次のステップへと渡されます。

さて、ここに2つの関数 `zero()` と `one()` があります。

```
1. def zero(s):
2.     if s[0] == "0":
3.         return s[1:]
4.
5. def one(s):
6.     if s[0] == "1":
7.         return s[1:]
```

`zero()` は文字列 `s` を受け取り、最初の文字が `'0'` だった場合に、残りの文字列を返します。

そうでない場合はPythonの関数のデフォルトの戻り値である `None` を返します。 `one()` も、最初の文字が `'1'` かどうかをみて、同様の動作をします。

`rule_sequence()` という関数を想像してみてください。これは文字列と `zero()` あるいは `one()` の形をしたルール関数のリストを受け取ります。それから最初のルールを文字列に対して実行します。 `None` が返らなければ、戻り値を受け取ってそれに2番目のルールを適用し、 `None` が返らなければ、戻り値を受け取って3番目のルールを適用し、というように続けていきます。もしいずれかのルールが `None` を返した場合には、 `rule_sequence()` は停止して `None` を返します。そうでなければ、最後のルールの戻り値を返します。
これはインプットとアウトプットの例です。

```
print rule_sequence('0101', [zero, one, zero])
# => 1

print rule_sequence('0101', [zero, zero])
# => None
```

これは `rule_sequence()` の命令型バージョンです。

```
def rule_sequence(s, rules):
    for rule in rules:
        s = rule(s)
        if s == None:
            break

    return s
```

演習3. 上記のコードではループが使われています。これを再帰呼び出しに変えて、宣言型のコードに書き換えてください。

私の答え:

```
1. def rule_sequence(s, rules):
2.     if s == None or not rules:
3.         return s
4.     else:
5.         return rule_sequence(rules[0](s),
   rules[1:])
```

パイプラインを使う

前のセクションでは命令型のループを、補助関数を呼び出す再帰処理に書き換えました。このセクションでは、異なったタイプの命令型のループを、パイプラインと呼ばれるテクニックを使って書き換えていきます。
以下のループはいくつかのバンドの名前と誤った出身地と活動状況を持つディクショナリを変換します。

```
1. bands = [{'name': 'sunset rubdown',
   'country': 'UK', 'active': False},
2.          {'name': 'women', 'country':
   'Germany', 'active': False},
3.          {'name': 'a silver mt. zion',
   'country': 'Spain', 'active': True}]
4.
5. def format_bands(bands):
6.     for band in bands:
7.         band['country'] = 'Canada'
8.         band['name'] =
   band['name'].replace('.', '')
9.         band['name'] = band['name'].title()
10.
11. format_bands(bands)
12.
13. print bands
14. # => [{'name': 'Sunset Rubdown', 'active':
    False, 'country': 'Canada'},
15. #     {'name': 'Women', 'active': False,
    'country': 'Canada' },
16. #     {'name': 'A Silver Mt Zion', 'active':
    True, 'country': 'Canada'}]
```

関数の名前を見ると心配になります。“format”というのがとても曖昧だからです。コードをよく

見ると、この心配が現実のものとなります。1つのループの中で3つのことが行われています。`'country'` キーの値に `'Canada'` がセットされ、バンド名からピリオドが取り除かれ、バンド名が大文字に変換されています。このコードの意図は理解し難く、見た通りのことをしているのかも分かりません。再利用は難しいですし、テストするのも並列化するのも困難です。

こちらのコードと比較してみてください。

```
1. print pipeline_each(bands,
   [set_canada_as_country,
2.
   strip_punctuation_from_name,
3.
   capitalize_names])
```

このコードは簡単に理解できます。補助関数が一連の処理になっていて、関数型なのだろうという印象を与えます。前の関数からの出力が次の関数の入力になっています。
これらの補助関数が関数型ならば、簡単に理解することができます。再利用やテスト、並列化も簡単です。
`pipeline_each()` の働きは、バンドを1つずつ `set_canada_as_counrty()` のような変換関数にかけていくことです。全てのバンドに関数が適用されたら、 `pipeline_each()` は変換済みのバンドをひとまとめにし、次の関数に渡します。
変換関数を見てみましょう。

```
1. def assoc(_d, key, value):
2.     from copy import deepcopy
3.     d = deepcopy(_d)
4.     d[key] = value
5.     return d
6.
7. def set_canada_as_country(band):
8.     return assoc(band, 'country', "Canada")
9.
```

```
10. def strip_punctuation_from_name(band):
11.     return assoc(band, 'name',
    band['name'].replace('.', ''))
12.
13. def capitalize_names(band):
14.     return assoc(band, 'name',
    band['name'].title())
```

それぞれの関数は、バンドのあるキーの値を新しい値と関連付けています。元のバンドのディクショナリを変更することなくこれを行うのは簡単ではありません。この問題は、 `assoc()` が `deepcopy()` を使い、渡されたディクショナリのコピーを生成することで解決しています。それぞれの変換関数は変換をコピーに対して実行し、コピーを戻り値として返します。

何も問題はなさそうです。元のバンドのディクショナリは、あるキーの値に新しい値が関連付けられた時も、変更から守られています。しかしこのコードには実は、他に2つの潜在的な変更が潜んでいるのです。

`strip_punctuation_from_name()` において、ピリオドをはずされた名前は `replace()` を元の名前に対して呼び出すことで生成されています。 `capitalize_names()` では、元の名前に `title()` を適用することで大文字の名前が生成されています。 `replace()` や `title()` は関数型ではありません。従って、 `strip_punctuation_from_name()` も `capitalize_names()` も関数型とは言えないのです。

運の良いことに、 `replace()` も `title()` も取り扱う文字列を変更しません。これはPythonでは文字列が変更不能だからです。例えば `replace()` がバンド名を操作するとき、元のバンド名がコピーされ、 `replace()` はコピーに対して実行されます。良かったですね。

Pythonにおける文字列とディクショナリとの変更に関する差異は、Clojureのような言語の魅力を実現したものです。プログラマはデータを変更してしまうかどうか考える必要は全くありません。データは変更されないからです。

演習4: `pipeline_each` 関数を書いてみましょう。操作の順序を考えてください。順番に並んだバンドが1つずつ、最初の変換関数に渡されます。変換が終わったバンドは1つずつ次の変換関数へ渡されます。これを繰り返します。

私の答え:

```
1. def pipeline_each(data, fns):
2.     return reduce(lambda a, x: map(x, a),
3.                   fns,
4.                   data)
```

これら3つの変換関数は渡されたバンドの特定のフィールドを変更するために必須なものです。 `call()` を使うことでこのことを抽象化することができます。 `call()` は適用する関数と、キー値を受け取ります。

```
1. set_canada_as_country = call(lambda x:
   'Canada', 'country')
2. strip_punctuation_from_name = call(lambda x:
   x.replace('.', ''), 'name')
3. capitalize_names = call(str.title, 'name')
4.
5. print pipeline_each(bands,
   [set_canada_as_country,
6.
   strip_punctuation_from_name,
7.
   capitalize_names])
```

あるいは可読性を犠牲にして簡略化するなら、単に次のようにすることもできます。

```
1. print pipeline_each(bands, [call(lambda x:
   'Canada', 'country'),
```

```
                                call(lambda x:
    x.replace('.', ''), 'name'),
                                call(str.title,
    'name')])
```

`call()` のコードは以下のようになります。

```
def assoc(_d, key, value):
    from copy import deepcopy
    d = deepcopy(_d)
    d[key] = value
    return d

def call(fn, key):
    def apply_fn(record):
        return assoc(record, key,
    fn(record.get(key)))
    return apply_fn
```

1つずつ見る

まず1つ、 `call()` は高階関数です。高階関数は関数を引数にとったり、関数を返したりします。あるいは `call()` のように両方行うこともあります。

そして2つ目に、 `apply_fn()` は3つの変換関数にとてもよく似ています。レコード(ここではバンド)を受け取り、 `record[key]` で値を参照します。そして `fn` をその値に対して呼び出します。結果はレコードのコピーに対して反映し、コピーを返します。

3つ目、 `call()` は実際には何の働きもしません。 `apply_fn()` が呼び出されて実際の処理を行います。上で示した `pipeline_each()` の使用例では、 `apply_fn()` のインスタンスが渡されたバンドの `'country'` に `'Canada'` をセットしています。他のインスタンスは渡されたバンドの名前を大文字に変換します。

4つ目です。 `apply_fn()` のインスタンスが動作する時、 `fn` と `key` はスコープ内にありません。これらはどちらも `apply_fn()` の引数で

も、ローカル変数でもありません。
しかしそれでもこれらにアクセスすることができます。関数が定義される時に、関数は使用する変数への参照を保持します。つまり関数のスコープの外側で定義され、関数の内側で使用されるような変数への参照を記憶するのです。関数が動作し、変数を参照する時には、Pythonはローカル変数と引数を参照しようとします。そしてもし変数が見つからなければ、保存された参照を見にいきます。こうして `fn` と `key` が見つかるのです。
次に5つ目、 `call()` のコードの中では、バンドについての記述がありません。なぜなら `call()` は、トピックに関わらず、あらゆるプログラムのパイプライン関数の生成に使用されるからです。関数型プログラミングは、一般的で、再利用可能な、コンポーザブルな関数によって構築するものだと言うこともできます。
お疲れ様です。クロージャ、高階関数、そして変数のスコープについて数パラグラフで学びました。美味しいレモネードでも飲んでください。

バンドの変換プロセスであと1つやることがあります。名前と国名を除く全てを取り除くことです。 `extract_name_and_country()` によってこの情報を引き出すことができます。

```
1. def extract_name_and_country(band):
2.     plucked_band = {}
3.     plucked_band['name'] = band['name']
4.     plucked_band['country'] =
   band['country']
5.     return plucked_band
6.
7. print pipeline_each(bands, [call(lambda x:
   'Canada', 'country'),
8.                             call(lambda x:
   x.replace('.', ''), 'name'),
```

```
9.                              call(str.title,
   'name'),
10.
   extract_name_and_country])
11.
12. # => [{'name': 'Sunset Rubdown', 'country':
   'Canada'},
13. #     {'name': 'Women', 'country':
   'Canada'},
14. #     {'name': 'A Silver Mt Zion',
   'country': 'Canada'}]
```

`extract_name_and_country()` は `pluck()` と呼ばれる一般的な関数で書くことができます。`pluck()` は以下のようになります。

```
1. print pipeline_each(bands, [call(lambda x:
   'Canada', 'country'),
2.                             call(lambda x:
   x.replace('.', ''), 'name'),
3.                             call(str.title,
   'name'),
4.                             pluck(['name',
   'country'])])
```

演習5. `pluck()` は各レコードから抽出するキーのリストを受け取ります。書いてみてください。これは高階関数である必要があるでしょう。

私の答え:

```
1. def pluck(keys):
2.     def pluck_fn(record):
3.         return reduce(lambda a, x: assoc(a,
   x, record[x]),
4.                       keys,
5.                       {})
6.     return pluck_fn
```

今度は何？

関数型のコードは他の形式で書かれたコードと非常にうまく共存することができます。この記事で扱った変換のプロセスは、あらゆる言語のあらゆるコードベースに適用することができま

あらゆるコードベースに適用することができます。あなた自身のコードに当てはめてみてください。

MaryとIslaとSamを思い出してください。リストの繰り返し処理を、mapとreduceで置き換えました。

レースはどうでしたか。コードを関数で分割しました。そしてそれらの関数を関数型にしました。繰り返しは再帰呼び出しに書き換えました。

そしてバンドです。連続した操作をパイプラインで置き換えましたね。

---

1. 変更不能なデータは、変更することができません。（Clojureのような）いくつかの言語は、デフォルトですべての値を変更不能にします。どんな“突然変異の”操作でも値をコピーして、変更した後コピーを返します。これは、これは、プログラマが想定していなかったような状態にプログラムが入ってしまうことによって生じるバグを取り除きます。 ↩

2. 第一級関数をサポートする言語では、関数を他の値と同じように取り扱うことができます。これは、関数を生成したり、関数に渡したり、関数から受け取ったり、データの構造体に保持できることを意味します。 ↩

3. 末尾呼び出し最適化は、プログラミング言語の機能です。関数が再帰的に呼び出されるたびに、新しいスタック・フレームが生成されます。スタック・フレームは、現在の関数を呼び出した時の引数とローカル値を保持するのに用いられます。関数がかなりの回数、再帰的に呼び出されると、インタプリタやコンパイラがメモリを使い果たしてしまうことが

あります。末尾呼び出し最適化を用いる言語は、一連の再帰呼び出しの中で、同じスタックフレームを使用します。Pythonのような末尾呼び出し最適化を用いない言語は、一般に関数の再帰呼び出しの回数が数千回に限られています。 `race()` 関数の場合、たったの5ステップなので問題ありません。 ↩

4. カリー化は複数の引数をとる関数を、分解し、最初の引数を受け取って「次の引数を引数とする」関数を返す、ということを全ての引数について繰り返すことをいいます。 ↩

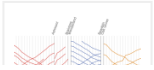
5. 並列化は、同期なしで並行して同じコードを実行することを意味します。これらの並列のプロセスは、しばしば複数のプロセッサで実行されます。 ↩

6. 遅延評価は、結果が必要となるまで、コードを実行させることを避けるコンパイラ技術です。 ↩

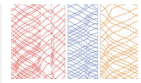
7. 繰り返しが毎回同じ結果を生じる時、あるプロセスは決定的であるといえます。 ↩

翻訳に対するフィードバックがございましたら、メールまたはGitHubのIssueよりお寄せください。

関連した投稿



私たちはいかにして環状線で”悪さをする列車”を捕まえたか



2017.02.24

映画制作におけるPython

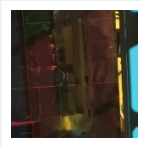
2017.02.07

Haskell、OCaml、RacketでGCのレイテンシを測る

2017.01.26

確率的プログラミング

2017.01.20



OSのデバッグ：メモリアロケーション講座

2017.01.19

NEXT POST

ハードウェアのスタートアップが国内生産をすべき5つの理由

PREVIOUS POST

D言語はデータサイエンスのためにある