余所では聞かないPythonと関数型の話

柴田 淳


プログラミング
と
抽象化

抽象的世界

入力

処理

出力

現実世界

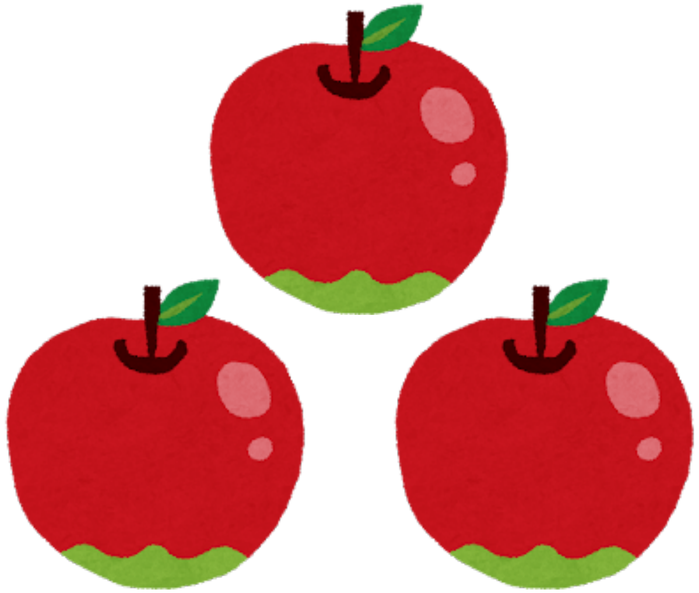
データ構造

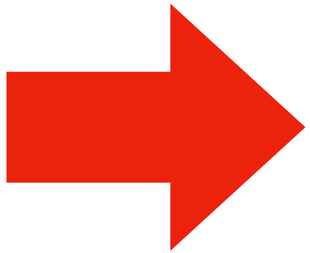
アルゴリズム

データ構造

抽象化

「状態」を使った抽象化 (1)





変数

```
apple = 3
```

現実世界

抽象的世界

「状態」を使った抽象化 (2)

データ構造

```
hp = 100
mp = 10
exp = 20
level = 1
```

```
hp = 40
mp = 50
exp = 100
level = 2
```

現実世界

抽象的世界

「状態」を使った抽象化 (3)

状態の変化

計算

```
hp = 10
mp = 0
```

```
hp = hp - 5
```

```
hp = 5
mp = 0
```

現実世界

抽象的世界

抽象化としての「副作用」

```
hp = 40
mp = 5 → 0
exp = 100
level = 2
```

```
hp = 25 → 35
mp = 10
exp = 20
level = 1
```

計算

計算

魔法使いが魔法を使って勇者の体力を回復する

データ構造

変数

データ構造

変数


オブジェクト指向
VS
関数型

何が違うのか

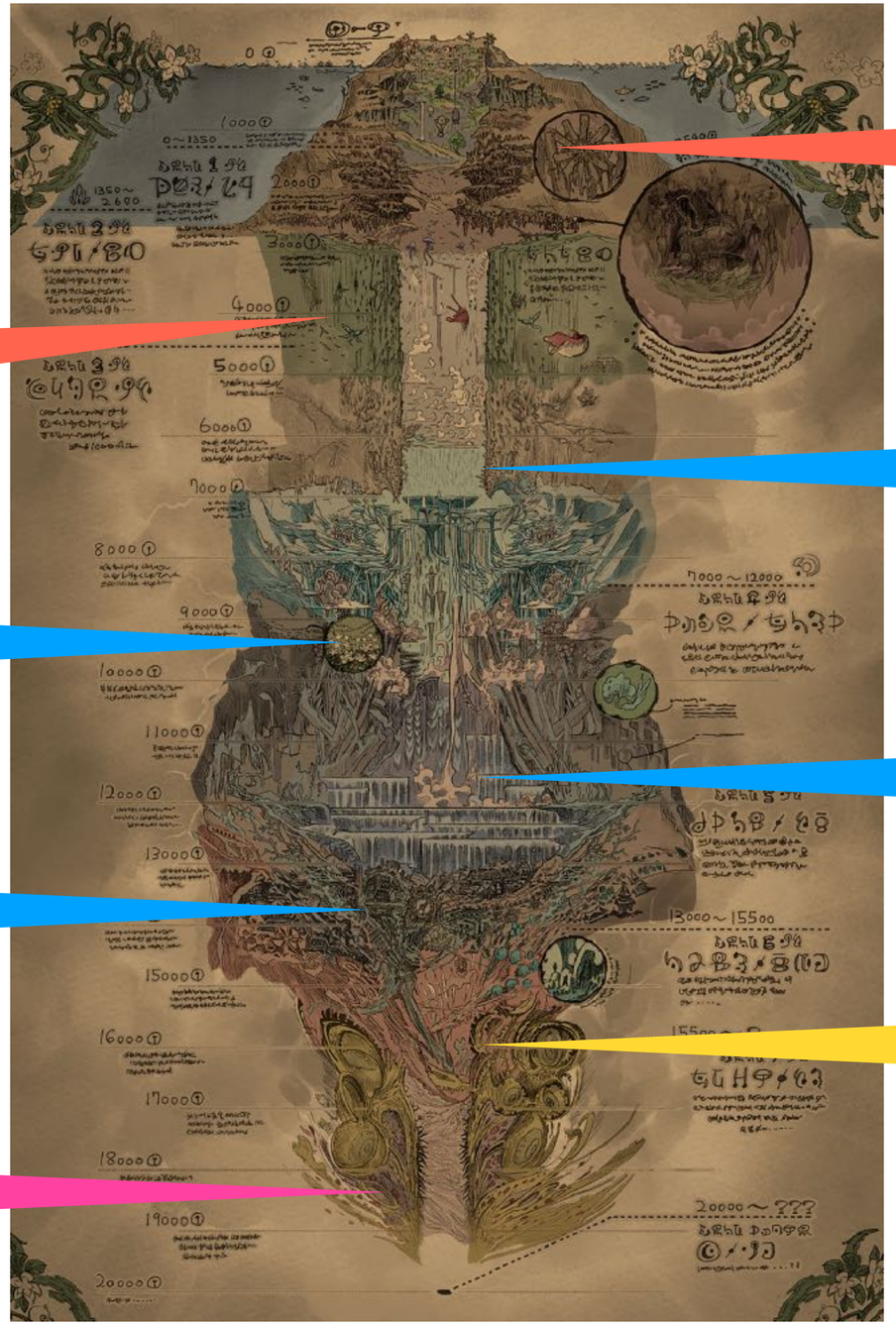
オブジェクト指向

抽象度が低い

関数型

抽象度が高い

抽象度による分類



エンドユーザ
アプリケーション

Excel，BIツール

Scratch

オブジェクト指向
言語

関数型言語

C，アセンブラ

マイクロコード

純粋数学的世界

オブジェクト指向

自然言語に近い

副作用を積極利用

The mage uses a magic to heal HP of the hero.

```
mage.use_heal(hero)
```

```
def use_heal(self, target):
    self.mp -= 10
    target.hp += 10
```

関数型(純粋なやつ)

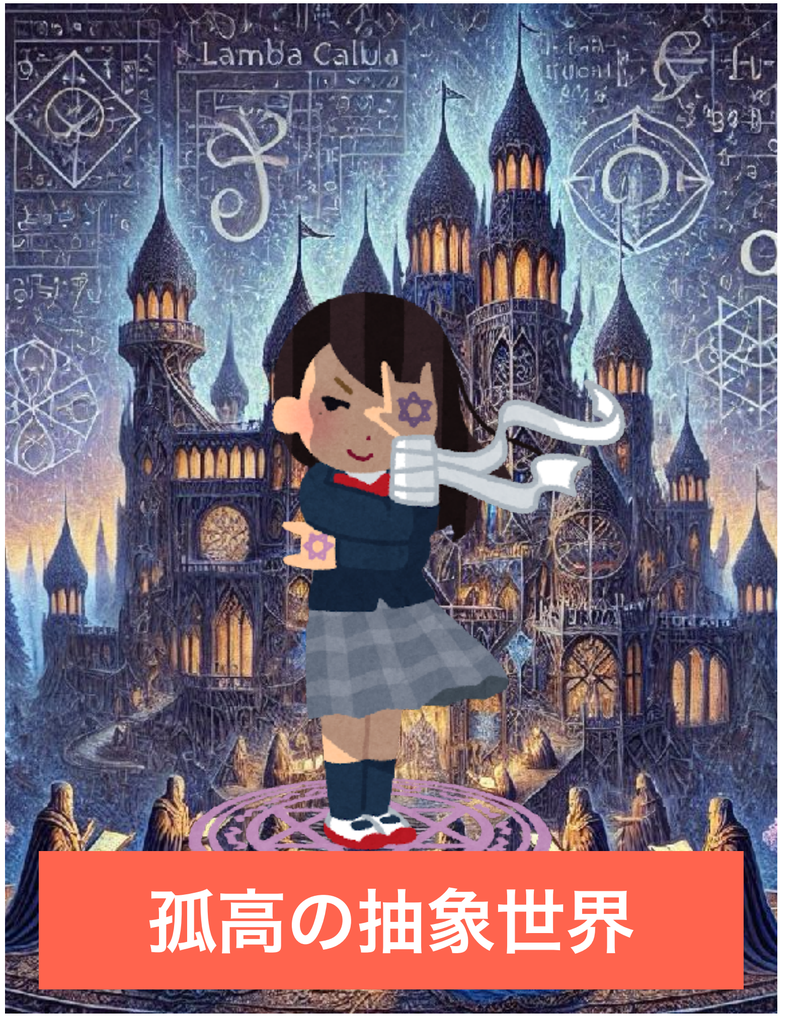
副作用を認めない

- 表示
- DB操作
- 入出力
- 状態変化

大きな制約

制約のメリット

- 抽象化
- エラーのない世界
- 最適化



我々の戦い

抽象化のイテレーション

抽象化

抽象化

抽象化

自然言語

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

データ構造

f

アルゴリズム

可読性

制約が少ない

開発効率


Python
標準の
関数型機能

抽象化マニアの時代(2.0以前)

高階関数

`map()` `filter()` `reduce()`

無名関数

```
lambda x : x*x
```

抽象化こそ至高……
メリットは
パフォーマンス向上……

```
# Given the list L, make a list of all strings
# containing the substring S.
sublist = filter(lambda s, substring=S:
                         string.find(s, substring) != -1,
                 L)
```

関数型「的」機能の導入

イテレータ

ループ，反復処理の抽象化

内包表記，ジェネレータ式

可読性向上

デコレータ

高階関数の可読性向上

```
# Given the list L, make a list of all strings
# containing the substring S.
sublist = filter(lambda s, substring=S:
                           string.find(s, substring) != -1,
                 L)
```

私が関数型に
しておいたから……

お前はすでに関数型

```
sublist = [ s for s in L if string.find(s, S) != -1 ]
```

我々の戦いは続く

抽象化のイテレーション

抽象化

抽象化

抽象化

自然言語

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

データ構造

f

アルゴリズム

関数型の導入

可読性

制約が少ない

Pythonの
オプショナルな
関数型機能

抽象化は「目的」を伴う

プログラミング的抽象化

```
pine = 2
apple = 3
fruit = pine+apple
```

数学的抽象化

$$x = 3$$
$$y = 2$$
$$z = x+y$$

可読性を伴った概念

```
hp = 100
mp = 10
exp = 20
level = 1
```

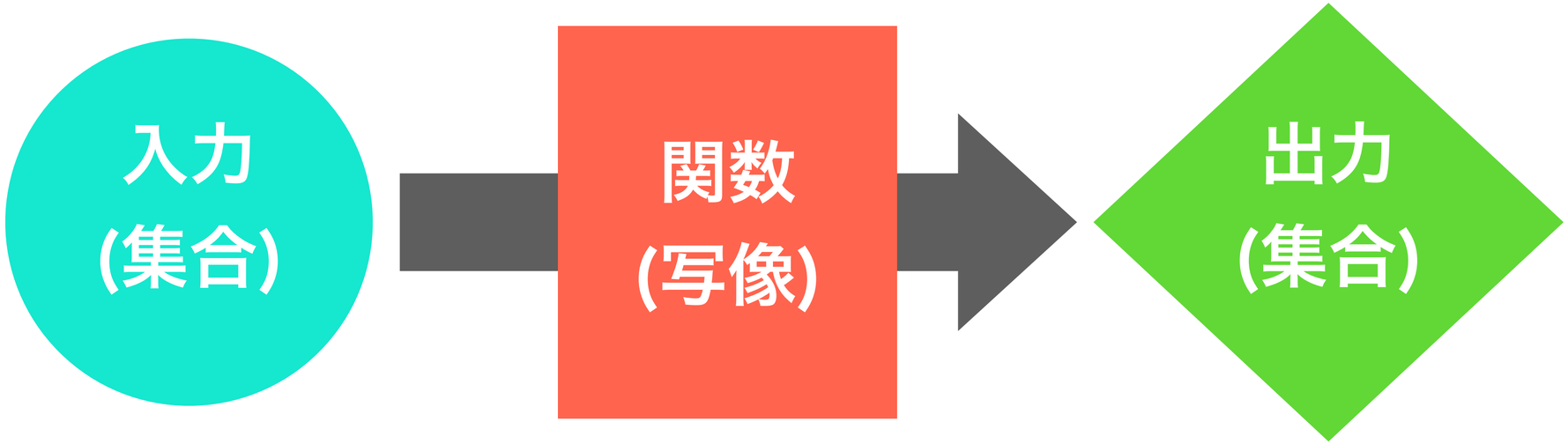
純粋な数の概念

$$\begin{pmatrix} 100 \\ 10 \\ 20 \\ 1 \end{pmatrix}$$

純粋関数型の目的

形式的証明可能性

プログラムが正しいことの
数学的証明



副作用が許されない

副作用とは(復習)

画面表示

DB操作

入出力

状態変化

こんなんでプログラム作れるか！

モナド

金融，セキュリティ分野に有利だにゃ

アノテーション

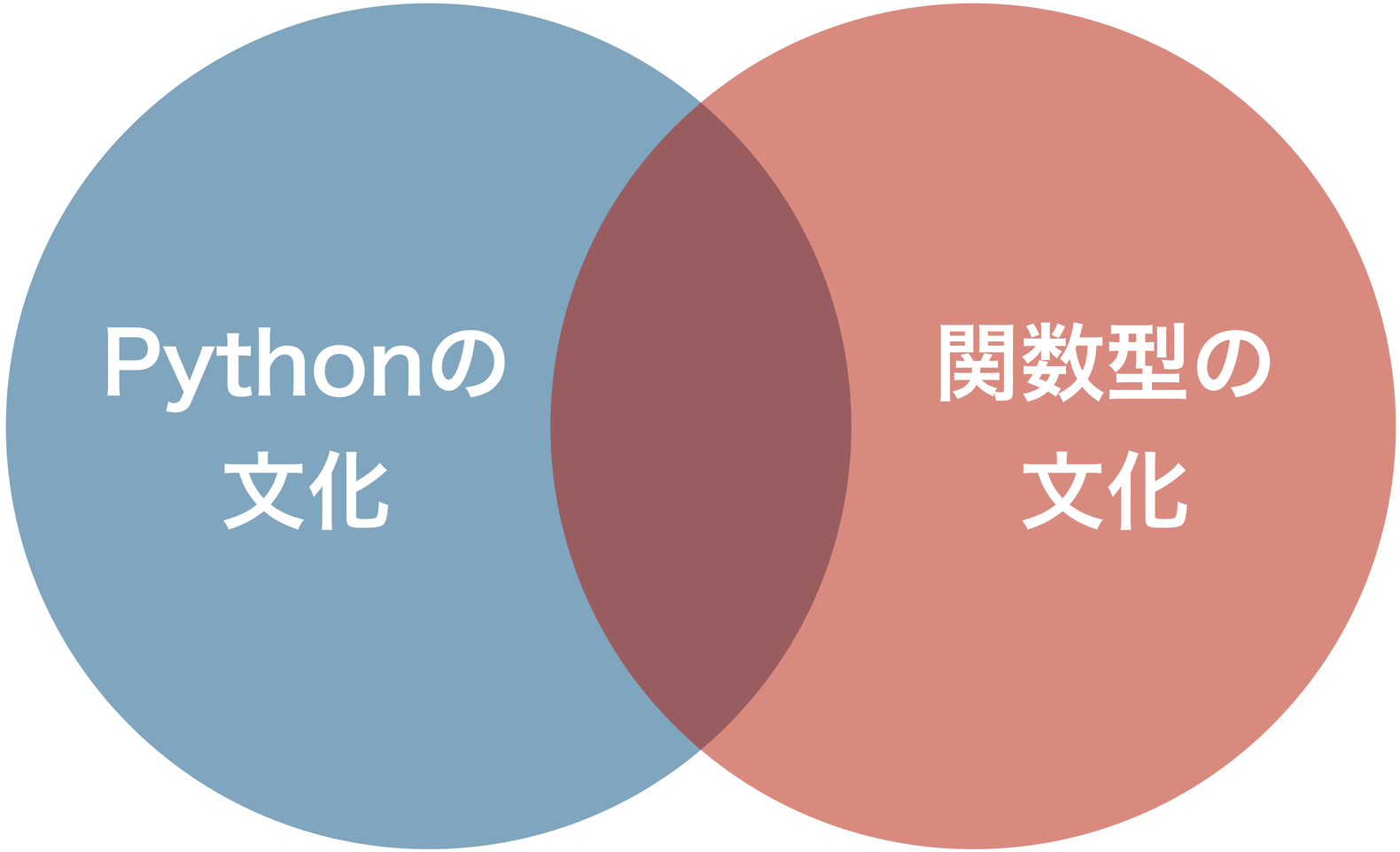
```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]
```

△ 関数型的文化

✕形式的証明可能性

言い方！言い方！

関数型との上手な付き合い方



functools

部分適用


```
from functools import partial
basetwo = partial(int, base=2)
basetwo.__doc__ = 'Convert base 2 string to an int.'
basetwo('10010')  # int('10010', base=2)
```

https://docs.python.org/ja/3/library/functools.html

functools(2)

デコレータと高階関数

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1
```

```
def factorial(n):
    return n * factorial(n-1) if n else 1

cachedfactorial = cache(factorial)
```

なぜ関数型が大事なのか

The Zen of Python

Simple is better than complex.

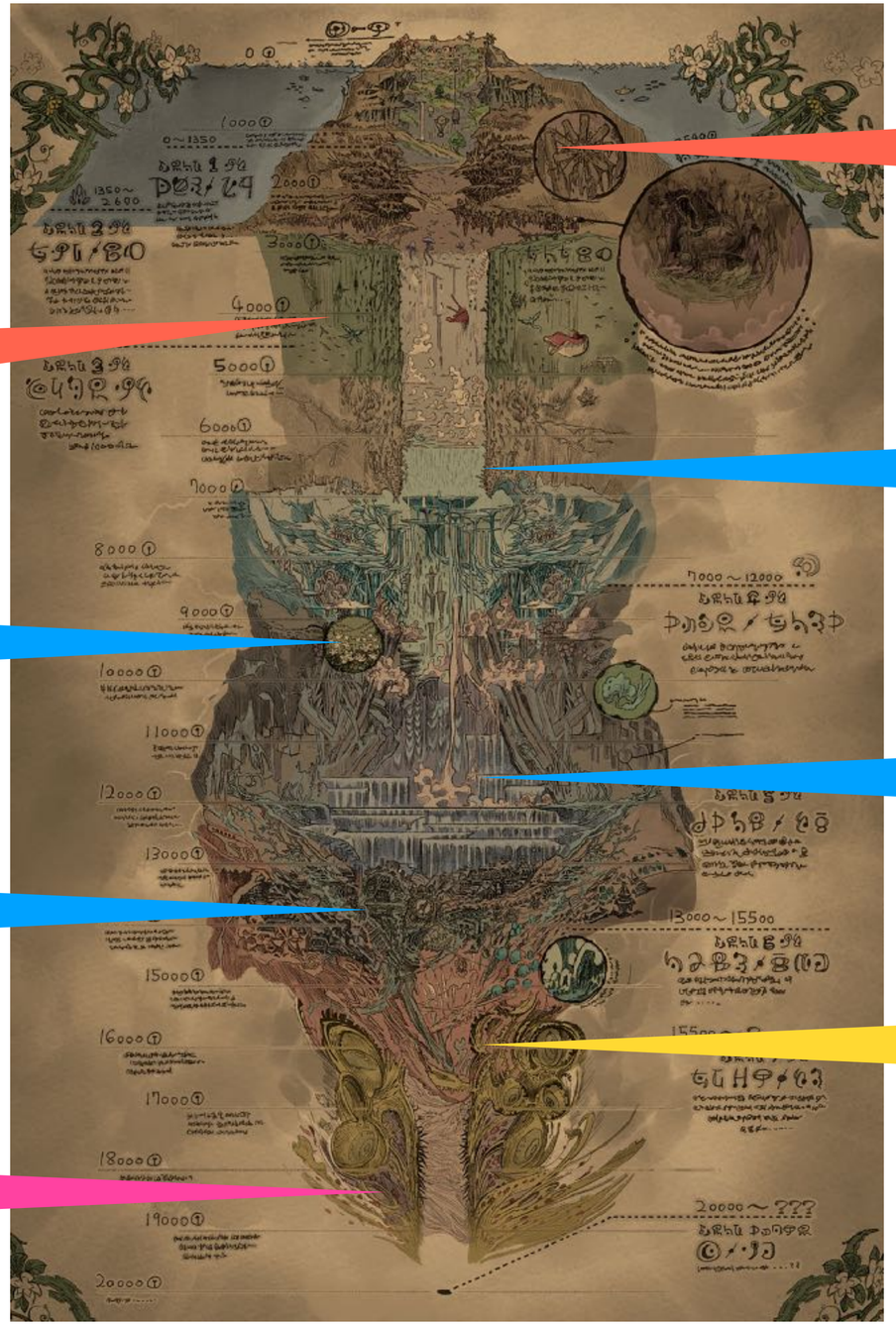
There should be one-- and preferably only one --obvious way to do it.

Readability counts.

Pythonic

成長目標としての関数型

抽象度による分類



エンドユーザ
アプリケーション

Excel，BIツール

Scratch

オブジェクト指向
言語

関数型言語

C，アセンブラ

マイクロコード

純粋数学的世界

筋の良いPythonの始め方

ありがとうございました