

PyTorch講座2

shun sato

前回の復習

- Gitを使おう
- ターミナルから実行しよう
- .pyファイルでpythonのプログラムを書こう

Pythonの基礎

Pythonでクラスを扱う

- `self`: クラスメソッドの第一引数
 - クラスインスタンス
➡ クラスのデータにアクセス可能
- `__init__()`: 最初に呼ばれる
 - コンストラクタ
- なぜクラスを使うのか？
 - a. データをまとめる
 - b. データを操作する関数をまとめる

```
main.py U X
pytorch > 02 > main.py > ...
1
2 class TestClass:
3
4     def __init__(self, num):
5         self.num = num
6
7     def show_num(self):
8         print("num:", self.num)
9
10 if __name__ == "__main__":
11     test_1 = TestClass(3)
12     test_1.show_num() # 3
13     test_2 = TestClass(10)
14     test_2.show_num() # 10
15
```

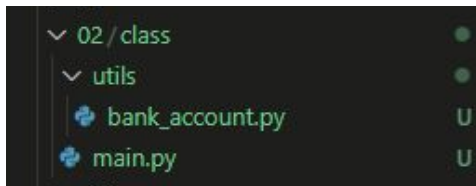
クラスの宣言と初期化

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 4
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02$ python main.py
num: 3
num: 10
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02$
```

演習：クラスの実装

演習: クラスの実装

- 銀行口座クラスの実装
- データ
 - name: 名前
 - balance: 預金
 - interest_rate: 金利



ディレクトリ構成

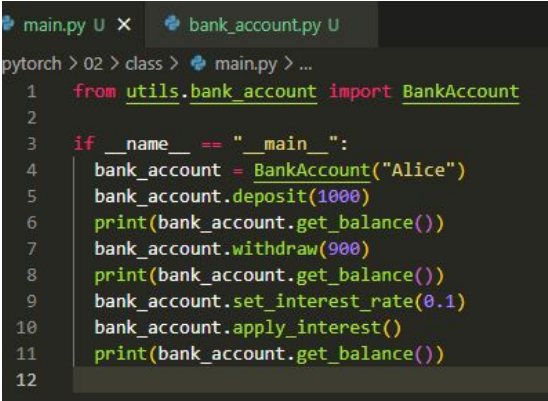
```
main.py U  bank_account.py U X
torch > 02 > class > utils > bank_account.py >
1  class BankAccount:
2
3      def __init__(self, name):
4          self.name = name
5          self.balance = 0
6          self.interest_rate = 0.01
7
```

クラスのテンプレート

演習: クラスの実装

実装するもの

- *deposit*: 預金を追加する
- *withdraw*: 預金を減らす
- *get_balance*: 現在の預金を返す
- *set_interest_rate*: 金利を設定する
- *apply_interest*: 金利を適用して預金を増やす



```
main.py U × bank_account.py U
pytorch > 02 > class > main.py > ...
1 from utils.bank_account import BankAccount
2
3 if __name__ == "__main__":
4     bank_account = BankAccount("Alice")
5     bank_account.deposit(1000)
6     print(bank_account.get_balance())
7     bank_account.withdraw(900)
8     print(bank_account.get_balance())
9     bank_account.set_interest_rate(0.1)
10    bank_account.apply_interest()
11    print(bank_account.get_balance())
12
```

main.py



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 9
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/class$ python main.py
1000
100
110.0
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/class$
```

実行例

Numpyの基礎

Numpyとは

- 数値計算を効率よく実行可能なPythonライブラリ
- 特徴
 - a. 多次元配列:ndarrayという柔軟な多次元配列オブジェクトを扱う
 - b. 高速処理:バックエンドの実装がC言語なので**高速**
 - c. 豊富な関数:**多次元配列の計算**に使える便利な関数が揃っている



基本的な配列

- pythonのリストからndarrayを作成
 - `np.asarray()`
- 配列の形状を確認
 - **`ndarray.shape`**
- 特定の値でndarrayを作成
 - `np.zeros()`
 - `np.ones()`
 - `np.fill()`

```
basic_array.py U x
pytorch > 02 > numpy > basic_array.py > ...
1  import numpy as np
2
3  a = np.asarray([1,2,3])
4  print("1D array", a.shape)
5  print(a)
6  b = np.asarray([[1,2,3],[4,5,6]])
7  print("2D array", b.shape)
8  print(b)
9  c = np.zeros((2,3), dtype=np.int32)
10 print("Zero array", c.shape)
11 print(c)
12
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  10
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$ python basic_array.py
1D array (3,)
[1 2 3]
2D array (2, 3)
[[1 2 3]
 [4 5 6]]
Zero array (2, 3)
[[0 0 0]
 [0 0 0]]
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$
```

配列へのアクセス

- pythonのlistと同様のアクセスが可能
 - indexを指定
- カンマ区切りで複数のindexを指定可能
- 「:」で全体を指定
- スライスも可能

```
index.py U x
pytorch > 02 > numpy > index.py > ...
1  import numpy as np
2
3  a = np.asarray([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
4
5  # index
6  print(a.shape)
7  print(a[1])
8  print(a[1,0])
9  # slice
10 print(a[:,1])
11 print(a[:,1:3])
12
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 10
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$ python index.py
(3, 4)
[5 6 7 8]
5
[ 2  6 10]
[[ 2  3]
 [ 6  7]
 [10 11]]
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$
```

基本的な関数

- 平均値の取得
 - `np.mean()`
 - `ndarray.mean()`
- 総和・最大・最小
 - `np.sum()`
 - `np.max()`
 - `np.min()`
 - (`ndarray.sum()`)なども可能)
- 計算軸の指定
 - `axis=n`

```
basic_func.py M x
codes > 02 > numpy > basic_func.py > ...
1  import numpy as np
2
3  a = np.asarray([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
4
5  # mean
6  print(np.mean(a))
7  print(a.mean())
8  print(np.mean(a, axis=0))
9  print(np.mean(a, axis=1))
10 # sum, max, min
11 print(np.sum(a))
12 print(np.sum(a, axis=0))
13 print(np.sum(a, axis=1))
14 print(np.max(a))
15 print(np.min(a))
16
```

```
shun@shun-System-Product-Name:~/Documents/pytorch-training/codes/02/numpy$ python basic_func.py
6.5
6.5
[5. 6. 7. 8.]
[ 2.5  6.5 10.5]
78
[15 18 21 24]
[10 26 42]
12
1
shun@shun-System-Product-Name:~/Documents/pytorch-training/codes/02/numpy$
```

配列の計算

- 行列の要素ごとの演算(行列積ではない)
 - アダマール積(要素積)
- 次元の異なる行列の演算
 - ブロードキャスト(形状変換)
 - 実験をして挙動を確認する
- スカラーのブロードキャスト
 - 要素全体に値が適用される
- numpyのブロードキャストについて
<https://note.nkmk.me/python-numpy-broadcasting/>

```
operation.py U X
pytorch > 02 > numpy > operation.py > ...
1  import numpy as np
2
3  a = np.asarray([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
4  b = np.ones(4, dtype=np.int32)
5
6  # element-wise operation
7  print(a+a)
8  print(a*a)
9  # matrix operation
10 print(a+b)
11 # broadcasting
12 print(a+10)
13 print(a*10)
14
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 10
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$ python operation.py
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]]
[[ 1  4  9 16]
 [25 36 49 64]
 [81 100 121 144]]
[[ 2  3  4  5]
 [ 6  7  8  9]
 [10 11 12 13]
 [14 15 16 17]
 [18 19 20 21]
 [22 23 24 25]]
[[10 20 30 40]
 [50 60 70 80]
 [90 100 110 120]]
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$
```

行列演算

- 行列の転置
 - `ndarray.T`
- 行列積
 - `ndarray.dot()`
 - `np.dot()`
- 逆行列
 - `np.linalg.inv()`
- 行列式
 - `np.linalg.det()`
- 固有値・固有ベクトル
 - `np.linalg.eig()`

```
matrix.py U X
pytorch > 02 > numpy > matrix.py > ...
1  import numpy as np
2
3  a = np.asarray([[1,2],[3,4]])
4
5  print(a)
6  print(a.T)
7  print(a.dot(a))
8  print(np.linalg.inv(a))
9  print(a.dot(np.linalg.inv(a)))
10 print(np.linalg.det(a))
11 print(np.linalg.eig(a))
12
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 10
shun@DESKTOP-VI3TN4M: ~/dev/pytorch/02/numpy$ python matrix.py
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
[[ 7 10]
 [15 22]]
[[-2.  1.]
 [ 1.5 -0.5]]
[[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
-2.0000000000000004
EigResult(eigenvalues=array([-0.37228132,  5.37228132]), eigenvectors=array([[ -0.82456484, -0.41597356],
 [ 0.56576746, -0.90937671]]))
shun@DESKTOP-VI3TN4M: ~/dev/pytorch/02/numpy$
```

条件式の適用

- 通常の比較演算子は使用可能
 - {<, >, =}
 - boolの配列に変換される
- マスクの活用方法
 - bool配列のことを**マスク**という
 - マスクをインデックスに入れるとTrueの値を取り出すことができる
 - **True=1, False=0**なので掛け算でフィルタリングもできる
- 条件の真偽で値を操作
 - `np.where()`

```
condition.py ×
codes > 02 > numpy > condition.py > ...
1  import numpy as np
2
3  a = np.asarray([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
4
5  print(a>5)
6  print(a[a>5])
7  print(a*(a>5))
8  # np.where(condition, value if true, value if false)
9  print(np.where(a>5, a, -1))
10
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 10
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$ python condition.py
[[False False False False]
 [False True True True]
 [ True True True True]]
[ 6  7  8  9 10 11 12]
[[ 0  0  0  0]
 [ 0  6  7  8]
 [ 9 10 11 12]]
[[-1 -1 -1 -1]
 [-1  6  7  8]
 [ 9 10 11 12]]
shun@DESKTOP-VI3TN4M:~/dev/pytorch/02/numpy$
```

配列の変形

- 配列の形状を変更
 - `np.reshape()`
 - `ndarray.reshape()`
- 次元数を変更することも可能
- 次元の自動補完も可能
 - -1

```
reshape.py M x
codes > 02 > numpy > reshape.py > ...
1  import numpy as np
2
3  a = np.asarray([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
4
5  print(a.shape)
6  print(a)
7
8  print(np.reshape(a, (12)))
9  print(np.reshape(a, (2,6)))
10 print(np.reshape(a, (6,2)))
11 print(np.reshape(a, (2,-1)))
12 print(a.reshape(12))
13
```

```
shun@shun-System-Product-Name:~/Documents/pytorch-training/codes/02/numpy$ python reshape.py
(3, 4)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[ 1  2  3  4  5  6  7  8  9 10 11 12]
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
[ 1  2  3  4  5  6  7  8  9 10 11 12]
shun@shun-System-Product-Name:~/Documents/pytorch-training/codes/02/numpy$
```


Numpyのポイント

- Pythonのfor文の置き換え
 - numpyの多次元配列演算やスライスを活用して高速化
- PythonのListとndarrayは違う
 - 異なるデータ型や数値型でないものは扱えない(文字列などは不可)
- ブロードキャストの活用
 - 次元の異なるndarrayの演算がどのように作用するか理解する
- マスクの活用
 - 条件に一致する値をマスク操作で一括に扱う

演習 : Numpy応用

Numpy応用

- ある学年の期末テストのデータ:3クラス・各クラス5人・2科目
 - 以下のデータをコピーしてください

```
data = np.array([  
    [[85, 78], [67, 82], [92, 88], [75, 70], [60, 64]],  
    [[70, 68], [77, 72], [85, 90], [60, 65], [78, 76]],  
    [[80, 84], [88, 87], [66, 68], [72, 73], [64, 60]]])
```

問題

1. データの形状を確認
2. クラスごとの科目別平均点
3. 全クラスの番号3番の学生での2科目目の最高得点
4. 全クラスの各科目の最高得点と最低得点の差
5. 各クラスの1科目目が80点以上の人数
6. 2科目の合計得点が135点を超えている人数
7. 全生徒の1科目目と2科目目の相関係数(np.corrは使わない！)

Numpy

回答例

- 右図と同じ出力ができれば正解

発展課題

- 各問題を1行で実装する

```
shun@shun-System-Product-Name:~/Downloads/pytorch/02/numpy$ python main.py
Data shape: (3, 5, 2)
各クラスの科目別平均点:
[[69.8 75.6]
 [70.4 77.6]
 [68.8 74.8]]
全クラスの番号3番の学生の中で2科目目の最高得点: 68
各科目で一番点数が高い人と低い人の点差: [38 24]
各クラスで1科目目が80点以上の人数: [1 2 1]
2科目の合計点が135点を超えている人数: 11
全生徒の1科目目と2科目目の相関係数 -0.1843166688708733
shun@shun-System-Product-Name:~/Downloads/pytorch/02/numpy$
```