脱初心者を目指す Python

～ import this の次～

DS東京

記法、書き方などを紹介します。
理解度チェックテストもあります。
最後の演習問題にも挑戦してみてください。

目次

import this と PEP8

PEP8

Python のコーディング規約

コードは書くより読まれることの方が多いです。

フォーマッタツールを使うと勝手に整形してくれます。

autopep8

radon

```
>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

内包表記 ①

- リスト内包表記

```
# 0~100までの整数を格納するリストが欲しい
result = []
for i in range(100 + 1):
    result.append(i)
```

↓

```
result = [i for i in range(100 + 1)]
```

- if 文と組み合わせ

```
# 0~100までの整数の中から偶数だけを格納するリストが欲しい
result = [
    i for i in range(100 + 1)
    if i % 2 == 0
]
```

```
# 0~100までの整数の中で偶数以外は"odd"とするリストが欲しい
result = [
    i if i % 2 == 0 else "odd"
    for i in range(100 + 1)
]
```

内包表記 ②

- 辞書内包表記

```
# 0~100までの整数をキーとしその二乗値を値とする辞書
result = {i: i**2 for i in range(100 + 1)}
```

- セット内包表記

```
# 0~100までの整数の二乗値を格納する集合
result = {i**2 for i in range(100 + 1)}
```

- タプル型も可能か？

```
# これはタプルではなくジェネレータ
>> result = (i**2 for i in range(100 + 1))
>> print(result)
>> <generator object <genexpr> at 0x000002124BB59A10>
```

- 文字列はイテレータブル

```
>> original = "hello"
>> uppercase = ''.join([char.upper() for char in original])
>> print(uppercase)
>> 'HELLO'
```

イテレータブル ＝ ループで回せる

ジェネレータ

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fibo = fibonacci()
for _ in range(10):
    print(next(fibo))
```

- 関数の実行状態を保存できる
- ジェネレータ関数は `next` で初めて実行される
- `yield` は `return` かつ一時停止
- 重いリストを作りたくない時に使う

アンダーバーの使い方

- 未使用の変数を明示

```
i = (1, 2, 3)
t, _, _ = i
# ただし"_"には値が代入される...
```

- 数値の桁を見やすく

```
>>> print(100_000_000_000)
100000000000
>>> print(100000000000)
100000000000
```

- プライベート変数を明示

```
class MyClass:
    def __init__(self):
        self._private_variable1 = 37    # <- 外部から参照されたくない値
        self.__private_variable2 = 9    # <- 外部から参照されたくない値

    def _private_method(self):    # <- 外部からコールされないメソッド
        print("This is a private method.")
# ただの慣習、"_"一つだと普通に見れる
# 二つだと頑張れば見れる
```

- マジックメソッド

```
if __name__ == "__main__":    # <- これ
    main()
# __file__, __name__, __init__, __new__あたりはよく使う
```

with 文と contextmanager①

- ファイルを開いてデータを書き込む

```
with open('example.txt', 'w') as file:
    file.write('Hello, with statement!')
```

- 明示的な `close()` 呼び出しが不要
- 終了処理を勝手にやってくれる

- データベース接続の例

```
import sqlite3

with sqlite3.connect('example.db') as conn:
    cur = conn.cursor()
    cur.execute('[なんらかのSQLクエリ]')
```

- DB 接続例では with ブロック内の処理が 1 トランザクションになる
- with を抜けたときコミットする

with 文と contextmanager②

- ファイルを開いてデータを書き込む

```
# ファイルを操作する関数を__enter__, __exit__で自作するとこうなる。
class ReadFile:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        # 開始処理
        self.file = open(self.filename, 'r')
        return self.file

    def __exit__(self, type, value, traceback):
        # 終了処理
        self.file.close()

with ReadFile("file.txt") as f:
    f.read()
```

- `contextlib` を使うとシンプル

```
from contextlib import contextmanager


@contextmanager
def read_file(filename):
    # 開始処理
    file = open(filename, 'r')
    try:
        yield file
    finally:
        # 終了処理
        file.close()
```

デコレータ

```python
from functools import wraps
import time
# 関数の処理時間を計測するデコレータ
def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        func(*args, **kwargs)
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(elapsed_time)
    return wrapper

@timer
def example_method():
    # 関数の処理
```

- 関数を装飾する関数
- 糖衣構文のひとつ

```python
# クラスベースのデコレータ
class Timer:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        start_time = time.time()
        self.func(*args, **kwargs)
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(elapsed_time)
```

可変長引数、スター

```
class SampleClass(SuperClass):
    def __init__(self, *args, **kwargs):    <- これ
        super().__init__(*args, **kwargs)    <- これ
```

- *args 位置引数
- **kwargs キーワード引数

```
def method(*args):
    # argsはタプル


method(1, 0, "A") # 何を引数にしてもタプルで渡される
```

```
def method(**kwargs):
    # kwargsは辞書


method(age=100, weight=100, tall=100) # 何を引数にしても辞書で渡される
```

- 辞書の合成

```
    i = {"key1": 1}
    j = {"key2": 2}
    k = {**i, **j}
```

- 引数の指定が面倒なとき

```
params = {'device': 'gpu',...} # 長いパラメータ
lgb_model = lgb.LGBMRegressor(**params)
```

構造的パターンマッチ、switch 文

```
point = (a, b)
match point:
    case (0, 0):
        print("Origin")
    case (x, 0):
        print(f"On the x-axis at {x}")
    case (0, y):
        print(f"On the y-axis at {y}")
    case (x, y):
        print(f"At coordinates ({x}, {y})")
```

- ver. 3.10 以降
- if 文と競合するが複雑なパターンだとこちらの方が書きやすい
- `_` でワイルドカードを指定できる
- 可変長のパターンも使える

リストの演算

- set 型を使って集合の演算をする

```
a = ["AA", "BB", "CC", "DD"]
b = ["CC", "DD", "EE", "FF"]
```

- 一度 set 型に変換して演算子を使う
- list 型に戻したいときは
  list(set(a) & set(b))

```
# 交差
>> set(a) & set(b)
{'CC', 'DD'}
```

```
# 和
>> set(a) | set(b)
{'CC', 'DD', 'EE', 'FF', 'AA', 'BB'}
```

```
# 差
>> set(a) - set(b)
{'AA', 'BB'}
```

```
# 対象差
>> set(a) ^ set(b)
{'EE', 'FF', 'AA', 'BB'}
```

all, any

- all
  - 全て `True` なら `True` を返す

```
>>> all([True, True, True])
True
>>> all([True, False, True])
False
```

- any
  - ひとつでも `True` なら `True`

```
>>> any([False, False, False])
False
>>> any([False, True, False])
True
```

- 例

```
if 論理式a and 論理式b and ...: # 長い条件
    pass

if all([                        # 若干簡潔になる
  論理式a,
  論理式b,
   ...
]):
    pass
```

- 動的につくったリストの判定などで使ったりする

その他

- 三項演算子
  - if 文が一行で書ける
- セイウチ演算子(名前付き式)
  - `:=`

```
# targetの長さが0ならlengthにNoneを入れたい
target = [...]
length = len(target) if len(target) != 0 else None
```

↓

```
target = [...]
length = n if (n:=len(target)) != 0 else None
```

- 可読性が悪化する場合がほとんど
- 分かりづらい
- 見やすいコードを書こう！

- ちなみに `...` は `Ellipsis`

```
>>> print(...)
Ellipsis
```

問題 ①

以下の `self` は何を指しているか？

```
class SampleClass():
    def __init__(self) -> None:
        pass
```

問題 ②

以下の論理式の出力は何か？

```
>> 0 == True
>> 1 == True
>> True if None else False
>> True if "" else False
>> True if " " else False
```

問題 ③

リストとタプルの違いは何か？

```
>> seq_A = (1, 2, 3)
>> seq_B = [1, 2, 3]

>> print(type(seq_A), seq_A)
<class 'tuple'> (1, 2, 3)
>> print(type(seq_B), seq_B)
<class 'list'> [1, 2, 3]
```

演習 ①

windows のフォルダ構造は一見して分かりずらい。
そこで以下のように深いフォルダも一覧で表示できるようにしたい。

```
[Root]    <- フォルダ
  ├ [Description]
  │  ├ overview.md
  │  └ [src]
  │    └ webpage.svg      <- ファイル
  ├ [Notebooks]
  ├ README.md
  └ [Submit]
    └ rough_submit.csv
```

要件

- 任意のフォルダに対して表示可能なこと
- 標準ライブラリのみで実装すること
- natsortは使ってもよい