

2022年10月24日(月) 4限
@研究棟A302

プログラミングA2 第4回

担当：伏見卓恭

連絡先：fushimity@edu.teu.ac.jp

居室：研A1201

プログラミングA2の流れ

第 1回：＜復習編＞関数，ファイル入出力，コンテナデータ型

第 2回：＜復習編＞クラスとオブジェクト

第 3回：＜文法編＞関数の高度な利用法 1

第 4回：＜文法編＞関数の高度な利用法 2

第 5回：＜文法編＞オブジェクト指向プログラミング

第 6回：＜応用編＞データ構造とアルゴリズム 1

第 7回：＜応用編＞データ構造とアルゴリズム 2

第 8回：＜実践編＞HTTPクライアント

第 9回：＜実践編＞スクレイピング

第10回：＜実践編＞データベース

第11回：＜実践編＞並行処理

第12回：＜総合編＞総合演習(複合問題)

第13回：＜総合編＞まとめ

第14回：＜総合編＞Python力チェック ← 確認テストのこと

本日のお品書き

- 前回課題の振り返りとパイソニックな書き方
- 関数の高度な利用法 2
 - デコレータ
 - イテレータ
 - ジェネレータ
- 読みやすいコードのために
 - 型ヒント, 関数アノテーション
 - docstring

デコレータ

デコレータ

組み込みデコレータ：
@classmethod
@staticmethod
@property

既存の関数やメソッドを変更するのではなく、デコレートすることで機能を追加する高階関数をデコレータと呼ぶ。
引数として渡された関数やメソッドを拡張して返す。

デコレータ関数の定義

```
def デコレータ関数(デコレート対象の関数):  
    def decorated():  
        事前処理  
        デコレート対象の関数の呼び出し  
        事後処理  
        return  
    return decorated
```

← 簡単なデコレーション

← 簡単なデコレーション

デコレータの使用方法①

```
def デコレート対象の関数():  
    ...
```

デコレート後の関数
= デコレータ(デコレート対象の関数)

デコレータの使用方法②

```
@デコレータ名  
def デコレート対象の関数():  
    ...
```

※@デコレータでデコレートするとデコレート前の状態には戻せない。

コード例：decorator.py

デコレータ
デコレート対象
デコレート中
デコレート後

decorator.py

```
def bossy(func): # 偉そうなデコレータ
    def _func(t):
        print("オレの名前は", end="")
        func(t)
        print("様だ！")
    return _func

def humble(func): # 謙遜なデコレータ
    def _func(t):
        print("わたくしの名前は", end="")
        func(t)
        print("でございます")
    return _func
```

@bossy

@humble

```
def print_name(t): # デコレート対象の関数
    print(t, end="")
```

```
print_name("ピカチュウ") # 常にデコレートされた状態
```

実行例

オレの名前はピカチュウ様だ！

実行例

わたくしの名前はピカチュウでございます

練習問題：html.py

HTMLの...タグと...タグで文字列を囲むデコレータを実装せよ.

[要件]

1. get_title関数

- Monsterクラスのインスタンスを受け取り,
- インスタンスからtitle属性を取得し,
- そのtitle文字列を返す

2. concat_strs関数 stringsをconcatenateするの意味

- Monsterクラスのインスタンスがならぶリストを受け取り,
- 各要素に対してget_title関数を用いて文字列を取得し,
- それらを結合した文字列を返す

3. li_decorator関数

- 受け取った関数を実行する前後に, を付与するデコレータ

4. ul_decorator関数

- 受け取った関数を実行する前後に, を付与するデコレータ関数

5. 3のデコレータで1の関数をデコレートする

6. 4のデコレータで2の関数をデコレートする

解答例：html.py

デコレータ
デコレート対象
デコレート中
デコレート後

html.py：デコレート対象の関数

5

```
def get_title(mon):  
    return 1
```

6

```
def concat_strs(mon_lst):  
    s = []  
    for mon in mon_lst:  
        s.append(get_title(mon))  
    return 2
```

html.py：デコレータ関数

```
def li_decorator(対象):  
    def _func(t):  
        s1 = "<li>"  
        s2 = func(t)  
        s3 = "</li>"  
        return s1+s2+s3+"¥n"  
    return _func
```

```
def ul_decorator(対象):  
    def _func(t):  
        s1 = "<ul>"  
        s2 = func(t)  
        s3 = "</ul>"  
        return s1+s2+s3+"¥n"  
    return _func
```


実行例：html.py

html.py

```
if __name__ == "__main__":
    monsters = [
        Monster("イーブイ"),
        Monster("シャワーズ"),
        Monster("サンダース"),
        Monster("ブースター"),
        Monster("エーフィ"),
        Monster("ブラッキー"),
        Monster("リーフィア"),
        Monster("グレイシア"),
        Monster("ニンフィア"),
    ]

    print(concat_strs(monsters))
```

実行例

```
<ul><li>イーブイ</li>
<li>シャワーズ</li>
<li>サンダース</li>
<li>ブースター</li>
<li>エーフィ</li>
<li>ブラッキー</li>
<li>リーフィア</li>
<li>グレイシア</li>
<li>ニンフィア</li>
</ul>
```

デコレートしなかった場合も確認してみよう。

パラメータを持つデコレータ

デコレータにパラメータを持たせたいときは、
パラメータを受け取るようなデコレータ生成関数で
さらにラッピングする必要がある。

パラメータを持つデコレータ関数の定義

```
def デコレータを作る関数(パラメータ)
    def デコレータ(デコレート対象の関数):
        def decorated():
            事前処理
            デコレート対象の関数の呼び出し
            事後処理
            return
        return decorated
    return デコレータ
```

デコレータの使用方法②

```
@デコレータ名(引数)
def デコレート対象の関数():
    ...
```

例題：html.py

タグの名前をパラメータとして受け取るデコレータを実装せよ

html.py：デコレート対象の関数

```
@tag_decorator("li")
def get_title(mon):
    return mon.title
```

```
@tag_decorator("ul")
def concat_strs(mon_lst):
    省略
```

html.py：デコレータ関数

```
def tag_decorator(tag):
    def _decorator(func):
        def _func(t):
            s1 = "<"+tag+">"
            s2 = func(t)
            s3 = "</"+tag+">"
            return s1+s2+s3+"¥n"
        return _func
    return _decorator
```

`_decorator`が実際のデコレータ
`tag_decorator`はデコレータを作る関数のような存在

デコレータスタック

複数のデコレータをスタックする（積み重ねる）ことで、複数の機能をネストさせて追加することができる。

デコレータスタック

@デコレータ3

@デコレータ2

@デコレータ1

def デコレート対象の関数():

...

← 内側（下側）のデコレータから動作することに注意

html.py

@tag_decorator("html")

@tag_decorator("body")

@tag_decorator("ul")

def concat_strs(mon_lst):

実行例

<html><body>イーブイ

シャワーズ

サンダース

省略

リーフィア

グレイシア

ニンフィア

</body>

</html>

functoolsモジュール

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

前ページまでの単純なデコレータ定義では、デコレート対象とは別の関数を定義してreturnしているため、元の関数オブジェクトが持つ特殊属性__name__や__doc__とは異なる。functoolsのwraps()やupdate_wrapper()を利用することでこれらの属性情報を保持できる。

decorator.py

```
def bossy(func):  
    @wraps(func)  
    def _func(t):  
        省略
```

```
@bossy  
def print_name(t):  
    print(t, end="")
```

```
print(print_name.__name__, func)
```

- @wraps()なしの場合

実行例

```
_func <function bossy.<locals>._func at 0x...>
```

- @wraps()ありの場合

実行例

```
print_name <function print_name at 0x...>
```

イテレータ

iterableとイテレータ

反復処理可能な

iterableオブジェクトとは、`for`文で要素を1つずつ取り出せるオブジェクトのことで、`__iter__()`メソッド、または、`__getitem__()`メソッドをシーケンスとして実装したクラスのインスタンスである。

例：リスト、タプル、`range`、オブジェクト、文字列、辞書、集合、ファイルオブジェクトなど

※**iterable**は、取り出した要素の位置を管理しない。

イテレータとは、`__next__()`メソッドと`__iter__()`メソッドを実装したオブジェクトであり、要素を1つずつ扱うために用いられる。

※イテレータも**iterable**である。

※イテレータは、1個取り出すごとにどこまで取り出したのかという状態を記憶しているため、要素を順番に取り出すことができる。

iterableの要素の取り出し方①

`__getitem__()`メソッドを実装したクラスのインスタンスオブジェクトは
[インデックス]によって要素にアクセスできる。

iterableから要素の取り出し方

iterableオブジェクト[インデックス]

iterableオブジェクト.`__getitem__`(インデックス)

```
title = "イーブイ"  
print(title[2])  
print(title.__getitem__(2))
```

```
monsters = [Monster(title) for title in titles]  
print(monsters[132])
```

```
evols = {"シャワーズ", "サンダース", 省略 "ニンフィア"}  
# print(evols[1])
```

← `str`クラス（文字列）
のインスタンス

← `list`クラス（リスト）
のインスタンス

← `set`クラス（集合）には
`__getitem__()`がない

実行例

```
ブ  
ブ  
イーブイ  
TypeError: 'set' object is not subscriptable
```


iterableの要素の取り出し方②

`__next__()`メソッドを実装したクラスのインスタンスオブジェクトは
`next(イテレータ)`によって順番に要素にアクセスできる。

イテレータの作り方

```
イテレータ = iter(iterableオブジェクト)  
イテレータ = iterableオブジェクト.__iter__()
```

イテレータの使い方（次の要素を取り出す）

```
next(イテレータ)  
イテレータ.__next__()
```

● `__iter__()`：iterableとイテレータの特殊メソッド

- iterableの場合：イテレータオブジェクトを返す
- イテレータの場合：自分自身を返す

● `__next__()`：イテレータの特殊メソッド

- イテレータが指す次のオブジェクト（iterableの要素）を返す
- 次のオブジェクトがない時、`StopIteration`例外をraiseする

コード例：iterator.py

iterator.py

```
title = "イーブイ"
title_itr = iter(title) # iterableからイテレータを生成
print(next(title_itr)) # 1回目
print(next(title_itr)) # 2回目
print(next(title_itr)) # 3回目
print(next(title_itr)) # 4回目
#print(next(title_itr)) # 5回目
```

← 文字列はiterable

実行例

イ
ー
ブ
イ

← StopIteration例外が発生する

最後の要素までアクセスしたら、再利用できない

→イテレータの再生成が必要

例：ファイルオブジェクトはファイル内容を一度しか読みことができない

```
evols = {"シャワーズ", "サンダース", 省略 "ニンフィア"}
```

← 集合はiterable

ただし順序は保持されない

```
evols_itr = iter(evols)
print(next(evols_itr)) # 1回目
print(next(evols_itr)) # 2回目
print(next(evols_itr)) # 3回目
print(next(evols_itr)) # 4回目
```

実行例

エーフィ
シャワーズ
ニンフィア
リーフィア

for-in文の裏側

`for 要素 in iterable:` を実行すると、

- ①`iter()`関数を自動的に呼び出すことでイテレータを抽出し
- ②`next()`関数を繰り返し呼び出すことで要素を抽出し
- ③`StopIteration`例外が発生したら`for`構文から抜ける
という処理が行われている。

for-in文

```
title = "イーブイ"  
for t in title:  
    print(t)
```

裏側

```
title = "イーブイ"  
title_itr = iter(title)  
while True:  
    try:  
        t = next(title_itr)  
    except StopIteration:  
        break  
    print(t)
```

練習問題：zukan.py

Zukanクラスをイテレータとして実装せよ

[要件]

- `__init__`(ファイルパス)メソッドを実装する
 - ファイルパスを引数として静的メソッドを呼び出し、ポケモン名文字列のリストを受け取り、インスタンス変数に設定する【済】
 - どの要素まで取り出したかを記憶するインデックス`idx`をインスタンス変数として定義し、`0`で初期化する【未】
- `__iter__`()メソッドを実装する
 - Zukanクラスのインスタンス（=イテレータ）を返す【未】
- `__next__`()メソッドを実装する
 - インデックスがリストの長さと同しかったら、`StopIteration`例外を`raise`する【済】
 - 該当するリストの要素を抽出する【未】
 - インデックスをインクリメントしたあと、要素を返す【未】

解答例：zukan.py

zukan.py

```
class Zukan:
    def __init__(self, file_path):
        self.titles = __class__.read_file(file_path)
        インデックス

    def __iter__(self):
        イテレータを返す

    def __next__(self):
        if self.idx == len(self.titles):
            raise StopIteration()
        title = リストtitlesの要素
        インデックスの更新
        return title
```

← 例外の送出

@staticmethod 省略

実行例：zukan.py

zukan.py

```
if __name__ == "__main__":
    zukan = Zukan(sys.argv[1]) ← イテレータの生成
    print(next(zukan))
    print(next(zukan))

    for i, z in enumerate(zukan, 1):
        print(f"{i:03d}¥t{z}")
        if i == 5:
            break

    for i, z in enumerate(zukan, 1):
        print(f"{i:03d}¥t{z}")
        if i == 5:
            break
```

実行例

フシギダネ
フシギソウ

001 フシギバナ
002 ヒトカゲ
003 リザード
004 リザードン
005 ゼニガメ

001 カメール
002 カメックス
003 キャタピー
004 トランセル
005 バタフリー

zukan自体がイテレータであるため、
for-in文の中で自動的に呼ばれるiter()関数により
自分自身が返され、新たなイテレータが生成されない

ついでに__getitem__()も実装

zukan.py

```
class Zukan:
```

省略

```
def __getitem__(self, idx):  
    if idx < 0 or len(self.titles) <= idx:  
        raise IndexError()  
    return self.titles[idx]
```

← 例外の送出

```
if __name__ == "__main__":
```

省略

```
print(zukan[132])
```

実行例

イーブイ

イテレータでないiterable

イテレータは最後の要素まで抽出すると、それ以上利用できない。
イテレータの機能を別のクラスとして切り分けることで、
イテレータではないiterableなクラスとして実装できる。

zukan_iterable.py

```
class ZukanIterator():
```

← イテレータ

```
    def __init__(self, zukan):
```

```
        self.source = zukan
```

```
        self.idx = 0
```

```
    def __iter__(self):
```

```
        return self
```

```
    def __next__(self):
```

```
        if self.idx == len(self.source.titles):
```

```
            raise StopIteration()
```

```
        title = self.source.titles[self.idx]
```

```
        self.idx += 1
```

```
        return title
```

zukan_iterable.py

```
class Zukan:
```

← iterable

```
    def __init__(self, file_path):
```

```
        self.titles = 省略
```

```
    def __iter__(self):
```

```
        return ZukanIterator(self)
```


実行例：zukan_iterable.py

zukan_iterable.py

```
if __name__ == "__main__":
    zukan = Zukan(sys.argv[1]) # lec04/data/poke_names.txt

    # print(next(zukan)) # イテレータではない (__next__()を持っていない)
    # print(next(zukan)) # → next()関数により__next__()を呼び出せない

    for i, z in enumerate(zukan, 1):
        print(f"{i:03d}¥t{z}")
        if i == 5:
            break

    for i, z in enumerate(zukan, 1):
        print(f"{i:03d}¥t{z}")
        if i == 5:
            break
```

iterableオブジェクトであるzukanから
for-in文のたびに自動的にイテレータが
生成される

実行例

001	フシギダネ
002	フシギソウ
003	フシギバナ
004	ヒトカゲ
005	リザード

001	フシギダネ
002	フシギソウ
003	フシギバナ
004	ヒトカゲ
005	リザード

itertoolsモジュール

<https://docs.python.org/ja/3/library/itertools.html>

効率的なループのためのイテレータ生成関数を有する標準ライブラリ

●おもな関数その1

- `accumulate(iterable[, 関数, スタート])` :
`iterable`の各要素を関数に従って累積した値をイテレータとして返す
- `groupby(iterable, key=None)` :
`key`で指定した関数またはラムダ式の値(`key`値)が等しい
`iterable`の連続する要素をグループ化して,
`key`値とグループをイテレータとして返す
- `islice(iterable, スタート, ストップ[, ステップ])`
`islice(iterable, ストップ)` :
`iterable`のスタートからストップまでのステップごとの
要素をイテレータとして返す

コード例：use_itertools.py

```
use_itertools.py
import itertools

lst = list(range(1, 11))
for i in itertools.accumulate(lst):
    print(i, end=" ")

titles = ["フシギダネ", ..., "カメックス"]
for title in itertools.accumulate(titles):
    print(title)

for key_, group_ in
    itertools.groupby(titles,
        key=lambda title: title[0:2]):
    print(f"{key_}:", end="¥t")
    for title in group_:
        print(title, end=" ")
    print()

for title in itertools.islice(titles, 2, None, 3)
    print(title)
```

← key値を
最初の2文字とした

実行例

1 3 6 10 15 21 28 36 45 55

フシギダネ
フシギダネフシギソウ
フシギダネフシギソウフシギバナ
:

フシ: フシギダネ フシギソウ
フシギバナ
ヒト: ヒトカゲ
リザ: リザード リザードン
ゼニ: ゼニガメ
カメ: カメール カメックス

フシギバナ
リザードン
カメックス

itertoolsモジュール

<https://docs.python.org/ja/3/library/itertools.html>

効率的なループのためのイテレータ生成関数を有する標準ライブラリ

●おもしろ関数その2

- `product(*iterables)` :
複数の`iterable`から1要素ずつ選んで得られる直積
タプルを返すイテレータ
- `permutations(iterable[, サイズ])` :
`iterable`の要素からサイズ個選んで得られる順列
タプルを返すイテレータ
- `combinations(iterable, サイズ)` :
`iterable`の要素からサイズ個選んで得られる組合せ
タプルを返すイテレータ

コード例：use_itertools.py

use_itertools.py

```
import itertools
```

```
titles = ["フシギダネ", ..., "カメックス"]
```

← 9個の要素からなる

```
for tpl in itertools.product(titles, titles):
```

```
    print(tpl, end=" ")
```

```
print()
```

← $9 \times 9 = 81$ 個のタプル

('フシギダネ', 'フシギダネ')...
('カメックス', 'カメックス')

```
for tpl in itertools.permutations(titles, 2):
```

```
    print(tpl, end=" ")
```

```
print()
```

← $_9P_2 = 72$ 個の順列タプル

('フシギダネ', 'フシギソウ')...
('カメックス', 'カメール')

```
for tpl in itertools.combinations(titles, 3):
```

```
    print(tpl, end=" ")
```

```
print()
```

← $_9C_3 = 84$ 個の組合せタプル

('フシギダネ', 'フシギソウ', 'フシギバナ')...
('ゼニガメ', 'カメール', 'カメックス')

ジェネレータ

ジェネレータ関数

ジェネレータとは、**イテレータ**を生成する関数のようなものである。
ジェネレータイテレータを生成し、そのイテレータが実行されるたびに要素の値を1つずつ生成するため、リストのように要素数の分だけメモリを消費することではなく、メモリ消費量が少ないというメリットがある。

ジェネレータ関数の作り方

```
def ジェネレータ関数():  
    yield 値  
    yield 値  
    :
```

ジェネレータ関数の使い方 (=ジェネレータイテレータの作成)

```
ジェネレータイテレータ = ジェネレータ関数()  
for 要素 in ジェネレータイテレータ:  
    print(要素)
```

※**yield文**：関数を**一時的に**停止させて、途中経過を返し、呼び出し元へ処理が戻る。
再びジェネレータが実行されると、続き処理され、次の**yield文**まで実行される。

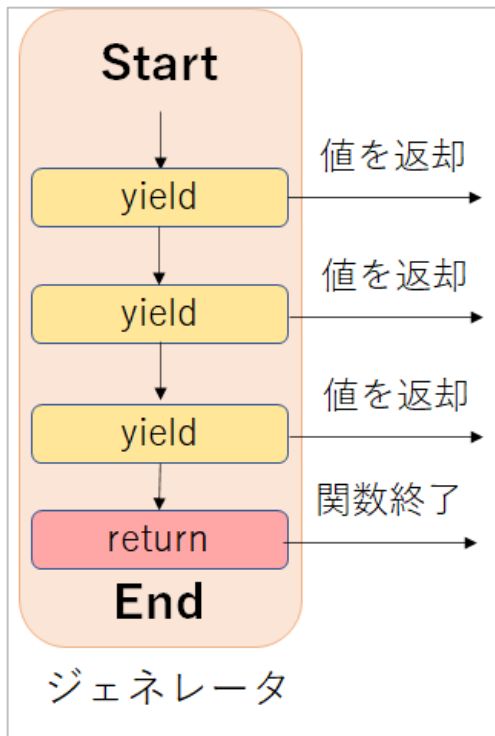
↔**return文**：関数を**完全に**終了させて、最終結果を返し、呼び出し元へ処理が戻る。

ジェネレータイテレータには

- `__iter__()`と`__next__()`がある：イテレータである
- `__call__()`がない：`イテレータ()`はできない ↔ 関数オブジェクトとは違う
- `__getitem__()`がない：`イテレータ[インデックス]`はできない
- `__len__()`がない：`len(イテレータ)`はできない

ジェネレータ関数の処理の流れ

- ジェネレータ関数を呼び出すと、**yield**の部分で処理が一時停止し、**yield**に記載された値が呼び出し元に返される
- その後、再度ジェネレータ関数を呼び出すと、**yield**の続きから処理がスタートし、次の**yield**の部分でストップする



```
def eevee_generator():  
    yield "イーブイ"  
    yield "シャワーズ"  
    yield "サンダース"  
    yield "ブースター"  
    yield "エーフィ"  
    yield "ブラッキー"  
    yield "リーフィア"  
    yield "グレイシア"  
    yield "ニンフィア"
```

```
eevee_iter = eevee_generator()  
print(next(eevee_iter))  
  
for _ in range(3): print(f"<{next(eevee_iter)}>")  
  
for item in eevee_iter: print(f"[{item}]")  
  
print(next(eevee_iter))
```

イーブイ
<シャワーズ>
<サンダース>
<ブースター>
[エーフィ]
[ブラッキー]
[リーフィア]
[グレイシア]
[ニンフィア]
StopIteration

通常関数との違い

1. 値の返し方

- 通常関数：`return`文で1度に全ての戻り値を返す
- ジェネレータ関数：`yield`文で個々に戻り値を返す

2. 返すもの

- 通常関数：`return`文に指定したオブジェクトそのもの
- ジェネレータ関数：`yield`文に指定したオブジェクトを返すイテレータ

3. 再呼び出し時の状態

- 通常関数：局所変数などの値はクリアされている
- ジェネレータ関数：局所変数や未処理の`try`文などを記憶しており、前回の`yield`文で処理が戻った時のまま

リストを返す関数との比較

generator.py : 関数・リスト版

```
def fnc_square_even(stop):  
    lst = []  
    for i in range(stop):  
        if i%2 == 0:  
            lst.append(i*i)  
    return lst  
  
if __name__ == "__main__":  
    print(fnc_square_even)  
    lst = fnc_square_even(5)  
    print(lst, len(lst))  
    for item in lst:  
        print(item)
```

実行例

```
<function fnc_square_even at 0x>  
[0, 4, 16] 3  
0  
4  
16
```

generator.py : ジェネレータイテレータ版

```
def gen_square_even(stop):  
  
    for i in range(stop):  
        if i%2 == 0:  
            yield i*i  
  
if __name__ == "__main__":  
    print(gen_square_even)  
    itr = gen_square_even(5)  
    print(itr, len(itr))  
    for item in itr:  
        print(item)
```

← エラー

実行例

```
<function gen_square_even at 0x>  
<generator object gen_square_even at>  
0  
4  
16
```

ジェネレータイテレータの仕組み

generator.py : ジェネレータイテレータ版

```
def gen_square_even(stop):  
    for i in range(stop):  
        if i%2 == 0:  
            yield i*i
```

```
if __name__ == "__main__":  
    itr = gen_square_even(5)  
    item = next(itr)  最初のyieldまで実行される  
    print(item)      → 0  
  
    item = next(itr)  2つ目のyieldまで実行される  
    print(item)      → 4  
  
    item = next(itr)  3つ目のyieldまで実行される  
    print(item)      → 16  
  
    item = next(itr)  4つ目のyieldを実行しようとするが、存在しないためエラー  
    print(item)      → StopIteration
```

ジェネレータイテレータ :

yield文まで実行され、処理が一時的に戻る。

→ 1要素ずつ生成する

next()で再び実行されると、

次の**yield**文まで実行され、

処理が一時的に戻る。

これが繰り返される

実行例

0

4

16

StopIteration

※リストの場合：あらかじめ3要素からなるコンテナを構築し、1要素ずつ取り出す

練習問題：name_generator.py

ポケモンの名前のリストと部分文字列を受け取り，リストから部分文字列を含む名前を1つずつ返すジェネレータ関数を定義せよ．

name_generator.py

```
def name_generator(lst, char):  
    for title in lst:
```

部分文字列を含む名前
を1つずつ返す

```
titles = read_names(sys.argv[1])  
char = sys.argv[2]  
name_generator = name_generator(titles, char)  
for res in ジェネレータイテレータ  
    print(res)
```

実行例

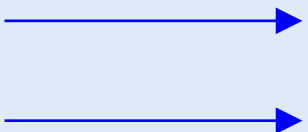
```
python lec04/name_generator.py lec04/data/poke_names.txt リー  
バタフリー  
ニドリーナ  
:  
メリープ  
ハリーセン
```

ジェネレータ式（内包表記）

ジェネレータ式の作り方

(式 for 変数 in iterable if 条件式)

```
gen = (i*i for i in range(5) if i%2 == 0)
print(gen)
for item in gen:
    print(item)
```



実行例

<generator object <genexpr> at 0x>

0
4
16

内包表記

パイソニックな書き方

● メリット

- 短く簡潔に書ける／処理速度が速い

● 書き方

- リスト： `[式 for 変数 in iterable if 条件式]`

↑ 要素となる

- タプル：ない

※以下はジェネレータとなる

(式 for 変数 in iterable if 条件式)

- 集合： `{式 for 変数 in iterable if 条件式}`

↑ 要素となる

- 辞書： `{式:式 for 変数 in iterable if 条件式}`

↑ キー:値となる

※式が三項選択式(if-else)であるのと、条件式は別物

`[三項選択式 for 変数 in iterable]`

読みやすいコードのために
コメント, `docstring`, 型ヒント

コメント

Pythonでは、複数行のコメントを書く構文はないが、ダブルクォーテーション3つで複数行の文字列を作ることができる。単なる文字列に対しては何もしないため、コメントとして機能する。

コメント

```
print()
```

```
# 1行のコメント
```

```
"""
```

```
複数行の文字列
```

```
複数行の文字列
```

```
複数行の文字列"""
```

- ← 「#」の後には半角スペース1つ入れる
- ← インデントの深さはコードと合わせる
- ← 長いコメントは行末（インラインコメント）ではなく、独立した行コメントにする
- ← 何をやっているか明らかなことは書かない

docstring

モジュールやクラス、関数に関する説明の複数行コメントのことで、モジュールの先頭、クラス定義の直後、関数定義の直後に書かれる。
`docstring`の内容は特殊属性`__doc__`に格納される。
`help()`関数により、`docstring`の内容を確認することもできる。

docstring

```
def hoge():  
    """  
    複数行の文字列  
    複数行の文字列  
    """  
  
print(hoge.__doc__)
```

```
def read_names(file_path: str):  
    """  
    poke_names.txtを読み込む関数  
    引数：ファイルのパス  
    戻り値：名前文字列のリスト、タイプリストのリスト、...  
    """  
  
print(read_names.__doc__)
```

← `help(read_names)`でもOK



```
read_names(file_path: str)  
poke_names.txtを読み込む関数  
引数：ファイルのパス  
戻り値：名前文字列のリスト、タイプリストのリスト、進化先リストのリスト
```


型ヒント, 関数アノテーション

Pythonのオブジェクトは, 型(type), 値(value), 同一性(id)の3要素からなり, 型を明示的に示すことを型ヒントor関数アノテーションという

一般的な書き方

宣言と代入を同時に行う場合
名前: 型 = 値

宣言と代入を別々に行う場合
名前: 型
名前 = 値

一般的な書き方の例

宣言と代入を同時に行う場合
level: int = 51

明らかな場合は省略しても問題ない

宣言と代入を別々に行う場合
level: int
level = 51

関数定義部で用いられることの方が多い → 関数アノテーションとも呼ばれる

関数定義部の書き方

def 関数名(引数名1: 型1, 引数名2: 型2) -> 戻り値の型:

関数定義部の書き方の例

def __init__(self, name: str, types: str) -> None:

明らかな場合は省略しても問題ない