Python

Ch21_Japanese

日本語テキスト処理

形態素解析 janome

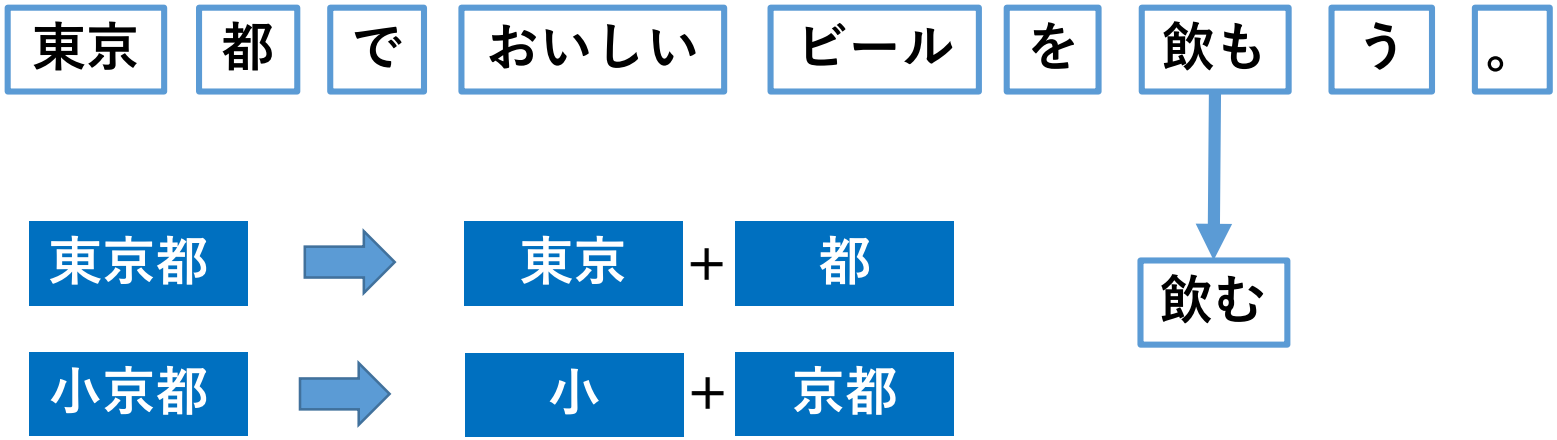
janomeのバージョンについて

- 教科書4章のコードはjanome0.3.8を前提に書かれている。
- メソッドtokenize()の戻り値の型が0.3.8ではリストだが，最新のjanome0.4.1ではジェネレータであるため、多くの箇所でlist用のメソッドが使えずエラーとなる。
- conda-forgeにはjanome0.3.8が存在しないので、pipでPyPIからインストールする。
- anacondaプロンプトで以下を実行

```
>conda activate ai
>conda uninstall janome
>pip install janome==0.3.8
>conda install tqdm
```

日本語のテキスト処理には分かち書きが必要

東京都でおいしいビールを飲もう。



分かち書きをするには「辞書」と「活用辞典」が必要

→ 形態素解析

Janomeによる形態素解析

```python
from janome.tokenizer import Tokenizer

t = Tokenizer()

text = '東京都でおいしいビールを飲もう。'
tokens = t.tokenize(text)

# 文章を分割する
for token in tokens:
    print(token)
```

```
東京	名詞,固有名詞,地域,一般,*,*,東京,トウキョウ,トーキョー
都	名詞,接尾,地域,*,*,*,都,ト,ト
で	助詞,格助詞,一般,*,*,*,で,デ,デ
おいしい	形容詞,自立,*,*,形容詞・イ段,基本形,おいしい,オイシイ,オイシイ
ビール	名詞,一般,*,*,*,*,ビール,ビール,ビール
を	助詞,格助詞,一般,*,*,*,を,ヲ,ヲ
飲も	動詞,自立,*,*,五段・マ行,未然ウ接続,飲む,ノモ,ノモ
う	助動詞,*,*,*,不変化型,基本形,う,ウ,ウ
。	記号,句点,*,*,*,*,。,。,。
```

```python
t.tokenize(text, wakati=True)
```

```
['東京', '都', 'で', 'おいしい', 'ビール', 'を', '飲も', 'う', '。']
```

Janomeオブジェクト

- tokenをjanomeオブジェクトという。
- 組込み関数vars(オブジェクト)を用いると、属性とその値を辞書形式で取得できる。
- オブジェクト.属性で単語のプロパティが分かる。

```
1 vars(tokens[0])
```

```
{'surface': '東京',
 'part_of_speech': '名詞,固有名詞,地域,一般',
 'infl_type': '*',
 'infl_form': '*',
 'base_form': '東京',
 'reading': 'トウキョウ',
 'phonetic': 'トーキョー',
 'node_type': 'SYS_DICT'}
```

```
1 tokens[0].part_of_speech
```

```
'名詞,固有名詞,地域,一般'
```

```
1 vars(tokens[6])
```

```
{'surface': '飲も',
 'part_of_speech': '動詞,自立,*,*',
 'infl_type': '五段・マ行',
 'infl_form': '未然ウ接続',
 'base_form': '飲む',
 'reading': 'ノモ',
 'phonetic': 'ノモ',
 'node_type': 'SYS_DICT'}
```

```
1 tokens[6].infl_type
```

```
'五段・マ行'
```

形態素解析の限界

- Janome形態素解析は、かな文字だけの文章が苦手
- 辞書にある単語で検索しているだけ
- 意味の通じる文章の構成要素としての単語に分解しているわけではない。

```
text2 = "うらにわにはにわ、にわにはにわ、にわとりがいる。"
text3 = "裏庭には二羽、庭には二羽、鶏がいる。"
tokens2 = t.tokenize(text2, wakati=True)
tokens3 = t.tokenize(text3, wakati=True)
for tokens in [tokens2, tokens3]:
    wakati = []
    for token in tokens:
        wakati.append(token)
    print(wakati)
```

```
['うら', 'に', 'わに', 'はにわ', '、', 'に', 'わに', 'はにわ', '、', 'にわとり', 'が', 'いる', '。']
['裏庭', 'に', 'は', '二', '羽', '、', '庭', 'に', 'は', '二', '羽', '、', '鶏', 'が', 'いる', '。']
```

Bag of Words

Bag of Words

- Bag of Words
  - 文章中の、使用単語とその出現回数を情報として持つデータ構造
  - Bag of WordsからTF-IDFを算出して、文章中の重要な単語を推測できる。
- TF（Term Frequency）

$$\text{単語AのTF} = \frac{\text{単語}A\text{の出現数}}{\text{1つの文章中の総単語数}}$$

0での除算を防ぐために分母分子に1を足す

- IDF（Inverse Document Frequency）

$$\text{単語AのIDF} = \log \frac{\text{全文章の数} + 1}{\text{単語}A\text{が出現した文章の数} + 1}$$

- TF-IDF　　$= \text{TF} \times \text{IDF}$

指標	値の範囲	性質
TF	[0, 1]	ひとつの文中での当該単語の出現確率
IDF	[0, ∞)	当該単語を含んだ文が出現する頻度が高いほど小さい。多くの文で使われる「てにをは」「です」等はIDFが小さくなる。
TF-IDF	[0, ∞)	文に固有の単語ほどTF-IDFが大きい。すなわち、TD-IDFの大きな単語がその文のキーワードと考えられる。

複数の文章を単語に分割

- TF-IDFを計算して、文章中の重要単語を選び出す。

```
# 複数の文章を単語に分割する
from janome.tokenizer import Tokenizer

# ❶ Tokenizerのインスタンスを生成
t = Tokenizer()
# ❷ 対象となる文章のリスト
sentences = ['おいしいビールを飲む', 'コーヒーを飲む', 'おいしいクラフトビールを買う']
# ❸ 文章を分かち書き
words_list = []
for sentence in sentences:
    words_list.append(t.tokenize(sentence, wakati=True))
words_list
```

```
[['おいしい', 'ビール', 'を', '飲む'],
 ['コーヒー', 'を', '飲む'],
 ['おいしい', 'クラフト', 'ビール', 'を', '買う']]
```

重複の無い単語リスト

```
1 # 一意な単語のリストを作成する
2 unique_words = []
3 for words in words_list:  # ❶各単語を取り出す
4     for word in words:
5         if word not in unique_words:  # ❷存在しなければ追加
6             unique_words.append(word)
7 unique_words
```

['おいしい', 'ビール', 'を', '飲む', 'コーヒー', 'クラフト', '買う']

Bag of Words

Bag of Wordsは単語の出現順番の情報を持たない。

```
# Bag of Words のデータを作成する
bow_list = []
for words in words_list:
    bag_of_words = []  # ❶ 1文章のBag of Wordsを格納する
    for unique_word in unique_words:
        num = words.count(unique_word)  # ❷ 一意な単語の出現回数を数える
        bag_of_words.append(num)
    bow_list.append(bag_of_words)
bow_list
```

```
[[1, 1, 1, 1, 0, 0, 0], [0, 0, 1, 1, 1, 0, 0], [1, 1, 1, 0, 0, 1, 1]]
```

```
#pandasを用いて見やすく表示
import pandas as pd
df = pd.DataFrame(bow_list, columns=unique_words, index=sentences)
df
```

	おいしい	ビール	を	飲む	コーヒー	クラフト	買う
おいしいビールを飲む	1	1	1	1	0	0	0
コーヒーを飲む	0	0	1	1	1	0	0
おいしいクラフトビールを買う	1	1	1	0	0	1	1

IDF

```
# IDFを計算する
from math import log  # ❶ logをインポート
num_of_sentences = len(sentences)
idf = []
for i in range(len(unique_words)):  # ❷ 一意な単語分繰り返す
    count = 0
    for bow in bow_list:  # ❸ Bag of Wordsに存在すれば1を足す
        if bow[i] > 0:
            count += 1
    idf.append(log((num_of_sentences + 1) / (count + 1)))  # ❹ 単語のIDFを計算

df = pd.Series(idf, index=unique_words)
df
```

count=0のとき0での除算にしないために分母分子に1を足す

```
おいしい     0.287682
ビール       0.287682
を           0.000000
飲む         0.287682
コーヒー     0.693147
クラフト     0.693147
買う         0.693147
dtype: float64
```

3つすべての文章で出現する単語「を」のIDFは0になる。IDFが小さいほど、どの文章にも現れる単語であることを示す。

TF

```
# TFを計算する
tf_list = []
for bow in bow_list:  # ❶ Bag of Wordsのリストで繰り返す
    num_of_words = sum(bow)  # ❷ 1文章の単語の数
    tf = []
    for n in bow:
        tf.append(n / num_of_words)  # ❸ TFを取得
    tf_list.append(tf)

import pandas as pd
df = pd.DataFrame(tf_list, columns=unique_words, index=sentences)
df
```

	おいしい	ビール	を	飲む	コーヒー	クラフト	買う
おいしいビールを飲む	0.25	0.25	0.250000	0.250000	0.000000	0.0	0.0
コーヒーを飲む	0.00	0.00	0.333333	0.333333	0.333333	0.0	0.0
おいしいクラフトビールを買う	0.20	0.20	0.200000	0.000000	0.000000	0.2	0.2

TF-IDF

```
1  # TF-IDFを計算する
2  import numpy as np
3  idf_a = np.array(idf)
4  tf_list_a = np.array(tf_list)
5  tfidf_list = tf_list_a * (idf_a + 1)
6
7  df = pd.DataFrame(tfidf_list, columns=unique_words, index=sentences)
8  df
```

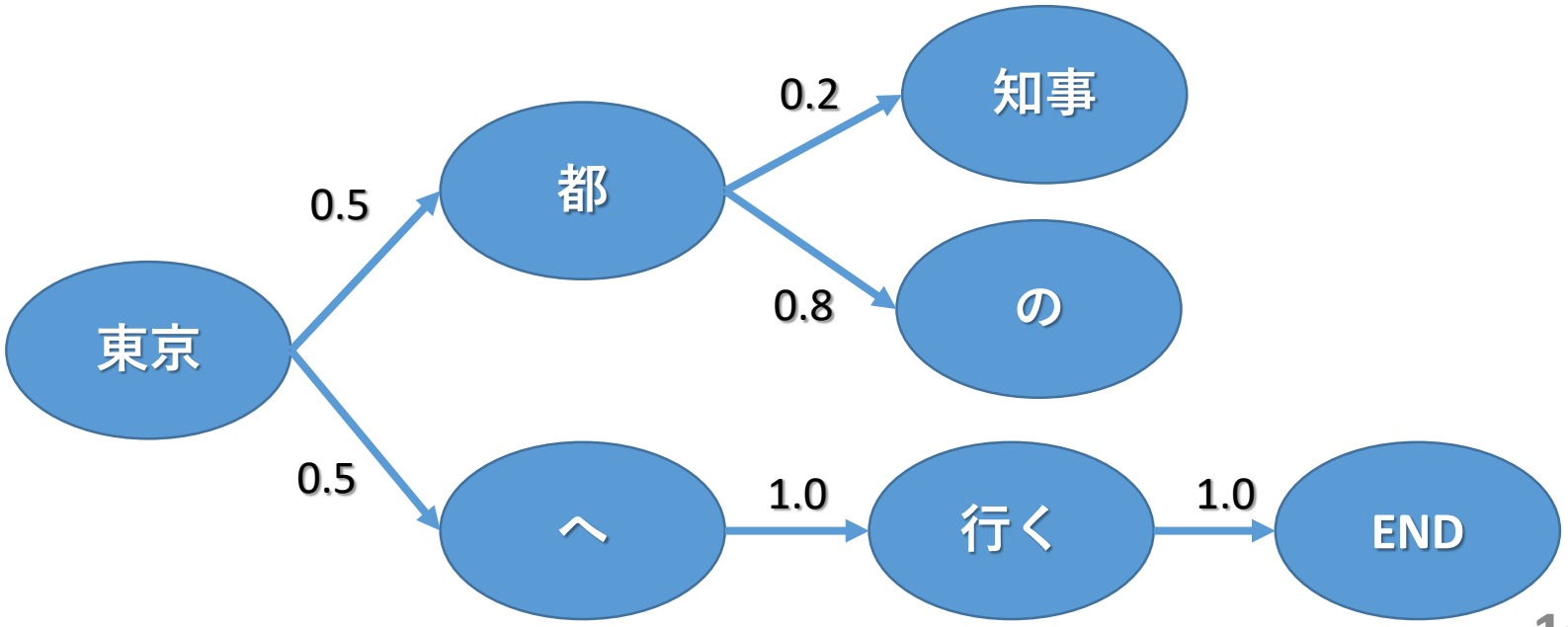
IDF=0の単語のTF-IDFが0にならないように補正

	おいしい	ビール	を	飲む	コーヒー	クラフト	買う
おいしいビールを飲む	0.321921	0.321921	0.250000	0.321921	0.000000	0.000000	0.000000
コーヒーを飲む	0.000000	0.000000	0.333333	0.429227	0.564382	0.000000	0.000000
おいしいクラフトビールを買う	0.257536	0.257536	0.200000	0.000000	0.000000	0.338629	0.338629

それぞれの文のキーワード

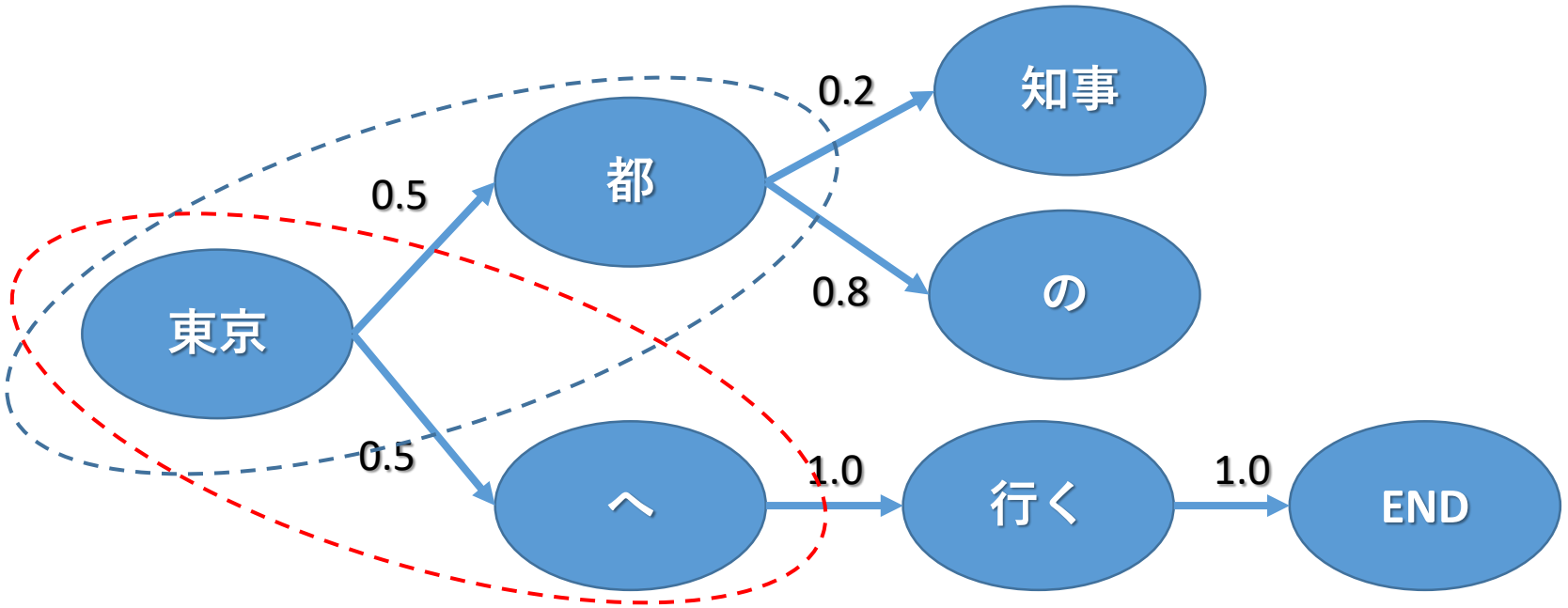
マルコフ連鎖による文章生成

- 文章中の単語Aの次に来る単語の出現確率は単語Aのみによって決まり、単語Aより前の文章には依存しない、と仮定した文章生成。
- 各単語の次に来る単語の出現確率は、既存の多くの文章から統計的に求めておく。



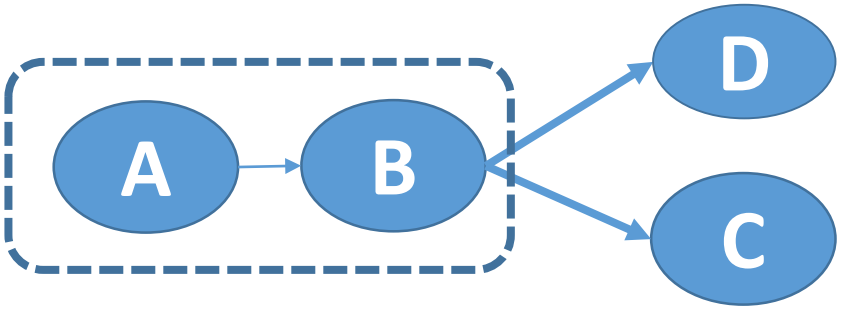
マルコフ連鎖による文章生成

- 日本語の助詞を一つの単語とみなすと、マルコフ連鎖で次に意味のある単語を確率的に選ぶのは、不適切と考えられる。
- 直前2単語から次の単語を生成するようにマルコフ連鎖を構成する。



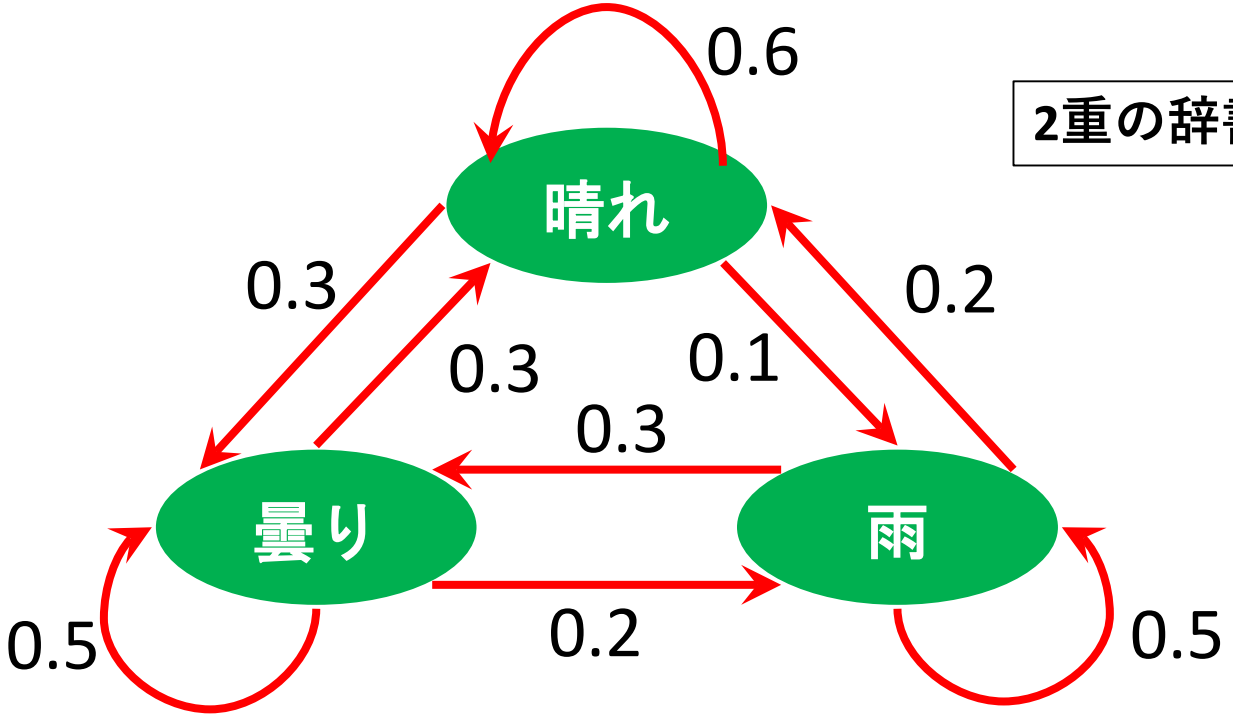
テキストデータからマルコフ連鎖用辞書データを作成する流れ

- 大量の日本語テキストデータを用意
  - 青空文庫 https://www.aozora.gr.jp
  - 自分のツィート履歴データ
  - ブログやメール
- 1つの文を3単語ずつの組みにする･･･[X,Y,Z]
- [A,B,C]の出現回数 ･･･ n
- [A,B,D]の出現回数 ･･･ m
- 辞書登録 {(A,B):{"words":[C,D], "weights":[n, m]}}



文章を生成するときにrandom.choice関数で使う。確率に変換しておく必要なし。

マルコフ連鎖の状態遷移図をデータで表す



2重の辞書構造で表現可能

```
weather = {
    '晴れ':{'next':['晴れ', '曇り', '雨'], 'weights':[0.6, 0.3, 0.1]},
    '曇り':{'next':['晴れ', '曇り', '雨'], 'weights':[0.6, 0.3, 0.1]},
    '雨':  {'next':['晴れ', '曇り', '雨'], 'weights':[0.6, 0.3, 0.1]},
}
```

マルコフ連鎖用辞書データの構造

('ビール', 'は') :

{

'words': [続く単語のリスト]

'weights' : [単語の出現頻度のリスト]

}

```
{
 (BEGIN, 'おいしい'): {'words':['ビール'], 'weights':[1.0],},
 (BEGIN, 'ビール'): {'words':['は','を'], 'weights':[0.4,0.6]},
 ('ビール','は'): {'words':['生'], 'weights':[1.0]},
 …
 ('は','生'): {'words':[END], 'weights':[1.0]},
}
```

collectionsの便利なツール
Counter と defaultdict

```
from collections import Counter
lista = ["a", "b", "c", "a", "a", "b"]
lista_count = Counter(lista) #要素をキー、出現数をvalueとする辞書を生成
print(lista_count["a"])
lista_count
```

```
3

Counter({'a': 3, 'b': 2, 'c': 1})
```

```
from collections import defaultdict
dic0 = defaultdict(int)
dic0["key0"] += 1 #辞書にないキーの値に加算が可能
print(dic0["key0"])
dic0
```

```
1

defaultdict(int, {'key0': 1})
```

演習21-1

- テキストのlesson31-33のコードを入力し、マルコフ連鎖を用いた文章の自動生成の方法を確認しなさい。

Pygame Zero

pygameの簡易版

Scratchのあとに教えるプログラムの位置づけ

Pygame Zeroのインストール

- Pygame Zeroは最も簡単なゲーム作成用パッケージ
- コマンドプロンプトを起動する。
- PythonWorksの下にPyGameZeroフォルダを作成する。
- PyGameZeroフォルダに移動し、venvで仮想環境を作成し、pipでpgzeroとPillowをインストールする。

```
PythonWorks>mkdir PyGameZero
PythonWorks>cd PyGameZero
・・・¥PyGameZero>python -m venv .venv
・・・¥PyGameZero>.venv¥Scripts¥activate.bat
(.venv)・・・¥PyGameZero>pip install pgzero
(.venv)・・・¥PyGameZero>pip install Pillow
```

ゲーム用フリー画像のダウンロード

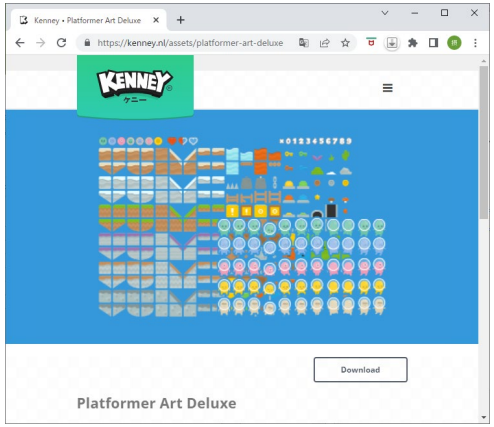
- Kennyのサイトからゲーム用画像をダウンロード
- 今回は以下の2種類の素材パックをダウンロードする

  https://kenney.nl/assets/platformer-art-deluxe
  https://kenney.nl/assets/space-shooter-extension

Downloadボタンをクリックすると、各zipファイル

- platformer-art-complete-pack-0.zip
- kenney_spaceshooterextension.zip

がダウンロードフォルダにダウンロードされる。
通常は自動で解凍され、デスクトップに同じ名前のフォルダが出来る。（自動解凍されない場合は、zipファイルをWクリックして解凍する。）



imagesフォルダの準備

- PyGameZeroフォルダの下にimagesフォルダを作成し、デスクトップに解凍された下の2つのフォルダを移動する。
  - platformer-art-complete-pack-0
  - kenney_spaceshooterextension
- 今回のプログラムで使用するpngファイルをフォルダ内からimages直下にコピーし、下のようにファイル名を変更する。

platformer-art-complete-pack-0から

- Base pack/Tiles/boxAlt.png → box.png
- Base pack/Tiles/castleCenter.png → floor.png
- Base pack/Tiles/signExit.png → exit.png
- Base pack/Player/p3_walk/PNG/p3_walk3.png → alien.png

Kenny_spaceshooterextensionから

- PNG/Sprites X2/Rockets/spaceRockets_001.png →spaceRockets_001.png

画像サイズの調整

- VScodeを起動し、PyGameZeroフォルダ下に下のプログラムimg_resize.pyを作成する。
- 実行すると、alien.pngとspaceRockets_001.pngの縮小画像がplayer.png, rocket.pngという名でimagesフォルダ内に作成される。

img_resize.py > ...

```
1   from PIL import Image
2
3   img = Image.open("./images/alien.png")
4   img_resize = img.resize((70, 70))
5   img_resize.save('./images/player.png')
6
7   img = Image.open("./images/spaceRockets_001.png")
8   img_resize = img.resize((50, 100))
9   img_resize.save('./images/rocket.png')
```

演習21-2

VSCodeで配布資料の2つのサンプルプログラムをPyGameZeroフォルダに作成し、実行する。

- maze.py
- moon_lander.py

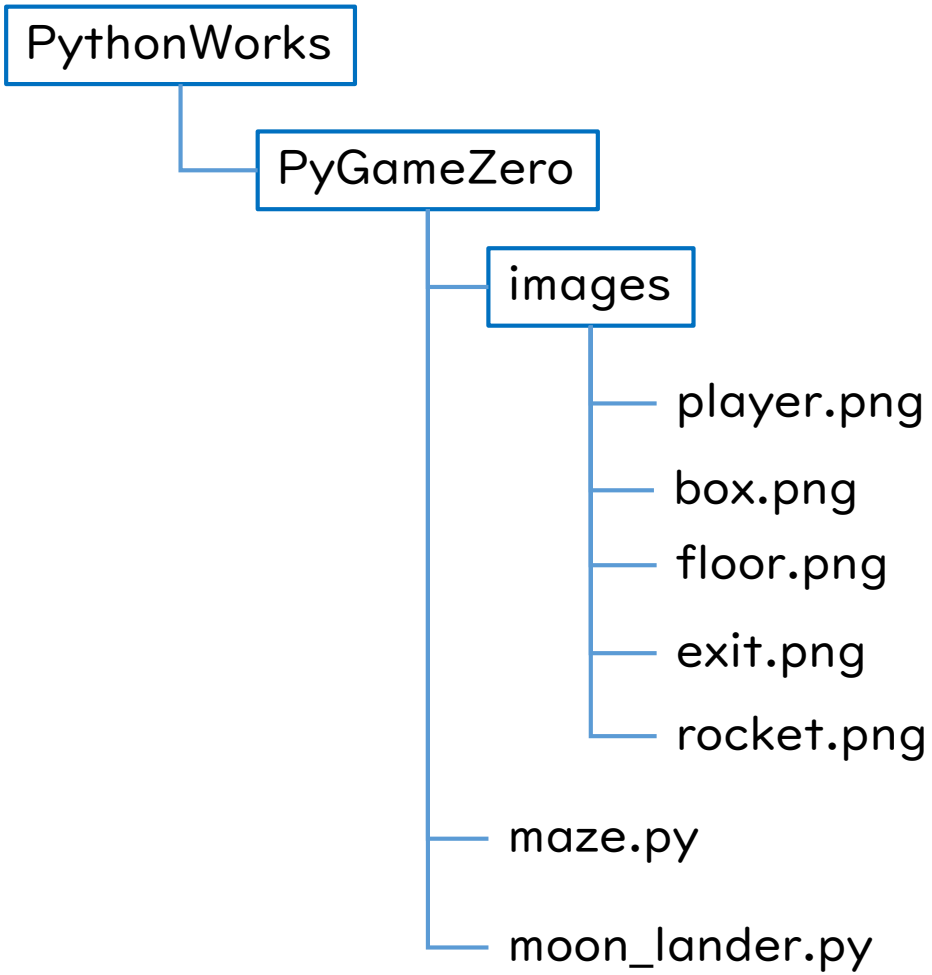
- 配布プログラムの修正箇所
  - 資料のままだとActorクラスが認識されないので、以下のActorをインポートするコードを追加すること

    from pgzero.builtins import Actor
  - VSCode上ではscreenやkeysにも波線が付くが、これらのオブジェクトは実行後に組み込まれるものなのでエラーではない。そのまま実行可能。
- Pygame Zero参考サイト

  https://pygame-zero.readthedocs.io/ja/latest/

  https://python.atelierkobato.com/pygame-zero/

ディレクトリ構造


PythonWorks
PyGameZero
images
player.png
box.png
floor.png
exit.png
rocket.png
maze.py
moon_lander.py


EXIT