# テキスト処理と文章生成

## スクリプト言語演習

佐藤 正平、草野 翼、伊澤 侑祐、大久保 寛

Scripting Languages / Dynamic Languages / 07

# 今日の道具は、状態・表・ループ

道具を一度に組み合わせず、4 段階に分けて見ていく。

| 段階 | できるようになること | 主な道具 | 課題 |
|---|---|---|---|
| 1. 分ける | 文章を単語の列にする | `with`, 正規表現 | 1 |
| 2. 数える | 単語のつながりを表にする | `zip`, `Counter` | 2, 2-b |
| 3. 選ぶ | 表から次の状態を決める | `sorted`, 重み付き抽選 | 3 |
| 4. 動かす | 表を引きながら状態を更新する | ループ、スタック | 4, 4-b |

- 前の段階の出力が、次の段階の入力になる
- 最後に、同じ道具が文章生成と電卓の両方に使えることを確かめる
- 各スライドの右側に、コードの読み方を書いてある

# 完成形: 文章を作る機械の全体像

## 全体像

```
ファイル
  ↓ 1. 分ける       tokenize_file
単語のリスト
  ↓ 2. 数える       build_table
次の語と回数の表
  ↓ 3. 選ぶ         pick_next, generate
新しい単語列
```

```
the cat sat on the mat the dog sat on the log ...
```

- 本文を読み、「ある語の次に何が来たか」を数える
- 数えた回数を重みにして、次の語を選び続ける
- 4 段階を 1 つずつ作り、最後につなぐ

# 文章を「数えられる単語」に変える

## 1. 分ける

```
import re

line = "Hello, world! It's a nice day."

line.split()
re.findall(r"[a-z']+", line.lower())
# ['hello', 'world', "it's", 'a', 'nice', 'day']
```

**コードの読み方**

- `line.split()` は空白で区切るだけ。`'world!'` のように、記号がくっついたまま残る
- `line.lower()` で、全部小文字にする
- `re.findall(r"[a-z']+", ...)`: 英字とアポストロフィ (') が続く部分を、全部取り出す
- 記号 (, ! .) は捨てられ、単語だけのリストになる

# ファイルは with で開き、行ごとに処理する

1. 分ける — 課題 1

```
import re

tokens = []
with open("sample.txt", encoding="utf-8") as f:
    for line in f:
        if line.startswith("#"):
            continue
        tokens.extend(
            re.findall(r"[a-z']+", line.lower()))
```

**コードの読み方**

1. with open(...) as f: ファイルを開く。with を抜けると、自動で閉じる
2. for line in f: ファイルを 1 行ずつ読む
3. # で始まる行 (コメント行) は、continue で飛ばす
4. それ以外の行は、単語を取り出して tokens につなげる (extend)
5. append はリストを 1 つの要素として足す。extend は中身を 1 つずつ足す

# 隣り合う語を数えると、続きを引ける

## 2. 数える

1 語の出現回数だけでは、その語の後に何が続くか分からない。そこで、隣り合う語をペアにして数える。

```
from collections import Counter, defaultdict

tokens = ["i", "like", "cats", "i", "like", "dogs"]
print(list(zip(tokens, tokens[1:])))

table = defaultdict(Counter)
for current, following in zip(tokens, tokens[1:]):
    table[current][following] += 1

print(table["i"])  # Counter({'like': 2})
```

**コードの読み方**

1. `zip(tokens, tokens[1:])`: 「今の語」と「次の語」を組にする。`("i", "like")`, `("like", "cats")`, …
2. `table` は、表の中に数え表 (`Counter`) を持つ辞書
3. 組を 1 つずつ見て、`table[今の語][次の語]` を 1 増やす
4. `table["i"]` を見ると、`i` の次に `like` が 2 回来たと分かる

# 辞書の中にカウンターを置くと、二段で数えられる



```
table = defaultdict(Counter)
table["i"]["like"] += 1
table["i"]["love"] += 1
table["i"]["like"] += 1
```

**コードの読み方**

- 外側の辞書:「今の語」を引く
- 内側の Counter:「次の語」ごとの回数を数える
- defaultdict なので、初めて出てきた語でも、先に準備しなくてよい

```
"i"    -> {"like": 2, "love": 1}
"like" -> {"cats": 1, "dogs": 1}
```

単語のリストを隣接ペアに変え、そのペアをこの表へ加算する。

# 並べ替えると、よく現れる続きが見える

## 2. 数える — 課題 2-b

```
from collections import Counter

counts = Counter({"cat": 3, "zoo": 1, "ant": 1})
ranked = sorted(
    counts.items(),
    key=lambda item: (-item[1], item[0]),
)
print(ranked[:2])  # [('cat', 3), ('ant', 1)]
```

**コードの読み方**

- `counts.items()`: `("cat", 3)` のような (単語, 回数) の組を取り出す
- `sorted(..., key=...)`: `key` で並べ替えの基準を決める
- 基準は `(-回数, 単語)`。回数の大きい順 (マイナスを付けて逆順にする)。同じ回数なら、ABC 順
- `[:2]` で、先頭の 2 つだけ

- 並び順を決めておくと、同じ入力から、いつも同じ結果になる

# 回数は、次の語を選ぶ重みに使える

## 3. 選ぶ — 課題 3

i の次に like が 2 回、love が 1 回あれば、like を 2/3、love を 1/3 の割合で選ぶ。

```
import random

words = ["like", "love"]
weights = [2, 1]
picked = random.choices(words, weights=weights)[0]
```

**コードの読み方**

- words: 候補の語。weights: 選ばれやすさ (回数)
- random.choices(...) は、重みに比例した確率で、ランダムに選ぶ
- 戻り値はリストなので、[0] で 1 語を取り出す
- 似た名前の random.choice (s なし) は、等しい確率で選ぶ。今回は使わない

# ループでは、選んだ語が新しい状態になる

3. 選ぶ — 課題 3

```
word = "i"
result = [word]

for _ in range(5):
    # 今の word を使って次の語を選ぶ
    word = next_word
    result.append(word)
```

**コードの読み方**

- `word`: 今の語 (状態)。`result`: これまでに選んだ語のリスト
- 5 回くり返す。毎回、次の語を選んで、`word` を入れ替え、`result` に足す
- 次の候補が表に無ければ、そこで終わる

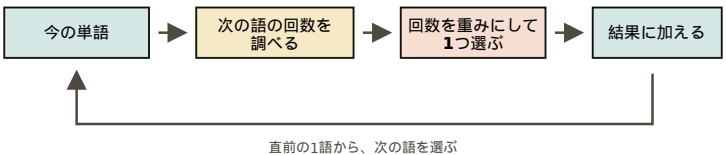


Figure: 今の単語を更新する生成ループ

# 状態は、単語だけでなく数の並びにもできる

4. 動かす

ここからは題材を変える。文章生成と同じ骨格で、**電卓**を作る。逆ポーランド記法では、数を積み、演算子で上の数を計算する。

| 読んだもの | 処理 | スタック |
|---|---|---|
| `3` | 数を push | `[3]` |
| `4` | 数を push | `[3, 4]` |
| `+` | 2 つ pop し、和を push | `[7]` |

*`10 3 -` は `7`。先に取り出される `3` が右側の値*

```
stack.append(3)
value = stack.pop()
```

- `append`: 一番上に積む (push)
- `pop`: 一番上から取り出す

# 表には、数だけでなく関数も登録できる

## 4. 動かす — 課題 4

文章生成では「語 → 次の語と回数」を登録した。電卓では「記号 →スタックを操作する関数」を登録する。

```
WORDS = {}

def add(stack):
    b, a = stack.pop(), stack.pop()
    stack.append(a + b)

WORDS["+"] = add
```

**コードの読み方**

- `WORDS`:「記号 → 関数」の表(普通の辞書)
- `add`: 上から 2 つ取り出し (`b` が先、`a` が次)、足して積む
- `WORDS["+"] = add`: 関数を `"+"` という名前で表に入れる(第 6 回の、辞書に関数を入れる形と同じ)
- `add` の後ろに `()` を付けない。呼ぶのではなく、関数そのものを入れる

# トークンを読むループが、表から処理を選ぶ

4. 動かす — 課題 4

```
for token in text.split():
    if token in WORDS:
        operation = WORDS[token]
        operation(stack)
    else:
        stack.append(int(token))
```

**コードの読み方**

1. 入力を空白で区切り、1 つずつ見る
2. 表 `WORDS` にあれば、その関数を取り出して呼ぶ
3. 表に無ければ、数として積む (`int(...)` で数に変える)
4. 新しいワードを登録しても、このループは変更しない

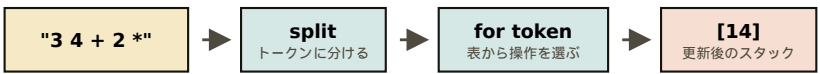


Figure: スタックを更新する処理の流れ

# スタック操作は、入る前と出た後で考える

4. 動かす — 課題 4-b

| ワード | 動作 | スタックの変化 |
|---|---|---|
| DUP | 一番上を複製 | (a → a a) |
| DROP | 一番上を捨てる | (a → ) |
| SWAP | 上 2 つを入れ替える | (a b → b a) |
| OVER | 2 番目を上にコピー | (a b → a b a) |
| ROT | 3 番目を一番上へ移す | (a b c → b c a) |

- (a b → b a) は、左が実行前、右が実行後のスタック
- どれも関数を書いて WORDS["DUP"] = dup のように入れる。評価ループは変えなくてよい
- 例: 3 DUP * は、3 を複製して掛けるので 9

# 道具は違っても、処理の骨格は変わらない

## 4 段階を別の題材へ移す

| | 文章を作る機械 | 計算する機械 |
|---|---|---|
| 入力の分割 | 正規表現で単語へ | 空白でトークンへ |
| 状態 | 今の単語 | 数のスタック |
| 表 | 次の語と回数 | 記号と操作関数 |
| 表を引いた結果 | 次の語を選ぶ | 関数を呼ぶ |
| 更新 | 選んだ語へ進む | スタックを書き換える |
| 停止 | 次の語がない | 入力を読み終える |

文章と電卓は題材が違う。それでも、表を引いて状態を更新する点は同じである。
次回 (第 8 回) は、この電卓に「新しいワードを定義する」仕組みを足して、小さな言語を作る。課題 4・4-b のコード (`WORDS` / `evaluate` と各ワード) は、そのまま土台に使う。

# 道具を一つずつ確かめてから組み合わせる

## まとめ

| やりたいこと | 使う道具 |
|---|---|
| ファイルを安全に読む | `with open(...)` |
| 必要な文字だけを抜く | `re.findall(...)` |
| 隣接する要素を組にする | `zip(xs, xs[1:])` |
| 二段の表で回数を数える | `defaultdict(Counter)` |
| 順位を決める | `sorted(..., key=...)` |
| 回数に応じて選ぶ | `random.choices(..., weights=...)` |
| 数を積み下ろしする | `append`, `pop` |
| 記号から処理を引く | 関数を値に持つ辞書 |

課題では、必要な道具を選び、状態・表・ループとして組み合わせる。

# 課題の進め方

| 課題 | 書くもの | 対応する段階 |
|---|---|---|
| 1 | `tokenize_file` | 1. 分ける |
| 2, 2-b | `build_table`, `top_following` | 2. 数える |
| 3 | `pick_next`, `generate` | 3. 選ぶ |
| 4, 4-b | `evaluate`, 四則演算, スタック操作ワード | 4. 動かす |
| 発展 | 温度、trigram | 3. 選ぶ (発展) |

- 各課題は「例題セルを実行 → `TODO` の手順を埋める → 確認セルが `OK`」
- 課題 1 → 2 → 3 は、前の関数をそのまま使ってつながる

# 付録: 講義では扱わない補足

発展的な内容や復習。時間があるときや、質問が出たときに使う。

- 重みを変形すると、選び方の大胆さが変わる
- 状態を長くすると、つながりは自然になる

# 重みを変形すると、選び方の大胆さが変わる

付録

重みを 1 / T 乗してから選ぶと、表を作り直さずに出力を変えられる。

```
counts = [3, 1]
weights = [count ** (1 / temperature) for count in counts]
```

| 温度 | 重み (3, 1) | 出力の傾向 |
|---|---|---|
| 0.1 | (59049, 1) | 最頻語に強く偏る |
| 1.0 | (3, 1) | 元の集計どおり |
| 5.0 | (1.25, 1) | 等確率に近づく |

変えるのは選び方だけ。生成ループの骨格は変わらない

# 状態を長くすると、つながりは自然になる

付録

| 見る文脈 | 表のキー | 出力と表の傾向 |
|---|---|---|
| 直前 1 語 | `"the"` | 候補は多いが、意味が崩れやすい |
| 直前 2 語 | `("the", "queen")` | 自然になるが、組み合わせが増える |
| 長い文脈 | 長いタプル | 元文のコピーが増え、未知の並びに弱い |

```
for a, b, following in zip(tokens, tokens[1:], tokens[2:]):
    table[(a, b)][following] += 1
```

状態を 1 語から 2 語へ増やしても、表を引いて更新する流れは同じ