データ構造・繰り返し・分岐

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 / EECS / 06

本日の流れ: 値を「まとめる・回す・分ける」

段階	できるようになること	主な道具	課題
1. まとめる	たくさんの値を 1 つにまとめる	リスト、タプル、辞書、集合	
2. 分ける	条件によって処理を変える	`if` / `elif` / `else`	1
3. 回す	同じ処理をくり返す	`for`, `range`, `while`	2, 3
4. 組み合わせる	回しながら数える・計算する	辞書で集計、スタック	4, 5

- 今日のゴール: 値を入れ物にまとめ、くり返しと分岐で処理できるようになる
- 最後の課題 5 (スタックと、関数を入れた辞書) は、次回の簡易インタプリタの土台になる
- コードのあるスライドでは、右側に読み方を書いてある

リスト: 値を順番に並べた入れ物

1. まとめる

```
volts = [3.3, 5.0, 4.8]

print(volts[0])      # 3.3
print(volts[-1])     # 4.8
print(len(volts))    # 3

volts.append(5.2)
print(volts)         # [3.3, 5.0, 4.8, 5.2]
```

コードの読み方

1. `[...]` で、値を順番に並べたリストを作る
2. `volts[0]` は先頭 (番号は 0 から数える)。`volts[-1]` は最後
3. `len(volts)`: 要素の数
4. `append(5.2)`: 最後に 1 つ足す

スライスとタプル

1. まとめる

```
volts = [3.3, 5.0, 4.8, 5.2]
print(volts[1:3])   # [5.0, 4.8]
print(volts[:2])    # [3.3, 5.0]

point = (2, 5)      # タプル
x, y = point
print(x, y)         # 2 5
```

コードの読み方

1. `volts[1:3]`: 番号 1 から、番号 3 の手前まで (スライス)
2. `volts[:2]`: 先頭から 2 つ
3. `(2, 5)` はタプル。リストと似ているが、後から書き換えられない
4. `x, y = point`: 中身を、2 つの変数に分けて入れる

辞書:「鍵 → 値」の対応表

1. まとめる

```
color = {"黒": 0, "茶": 1, "赤": 2}

print(color["茶"])    # 1

color["橙"] = 3       # 追加
print(len(color))     # 4
```

コードの読み方

1. {鍵: 値, ...} で辞書を作る。ここでは抵抗のカラーコード (色 → 数字)
2. color["茶"]: 鍵 ”茶” に対応する値を引く。結果は 1
3. color["橙"] = 3: 新しい鍵と値を追加する
4. 表に無い鍵を引くとエラー (KeyError) になる

辞書: 鍵があるか調べる・全部取り出す

1. まとめる

```
color = {"黒": 0, "茶": 1, "赤": 2}

print("赤" in color)    # True
print("青" in color)    # False

for name, num in color.items():
    print(name, num)
```

コードの読み方

1. `"赤" in color`: 鍵 ”赤” が表にあれば `True`
2. 引く前に `in` で調べると、`KeyError` を防げる
3. `color.items()`: (鍵, 値) の組を順に取り出す
4. `for` で回すと、黒 `0` / 茶 `1` / 赤 `2` と表示される

集合: 重なりのない値の集まり

1. まとめる

```
a = {"R1", "R2", "C1"}
b = {"R2", "C1", "L1"}

print(a & b)          # R2 と C1
print(a | b)          # R1, R2, C1, L1
print(len({1, 1, 2}))  # 2
```

コードの読み方

1. `{...}` に値だけを並べると集合。同じ値は 1 つにまとまる
2. `a & b`: 両方に入っているもの
3. `a | b`: どちらかに入っているもの
4. `{1, 1, 2}` は `{1, 2}` になるので、要素の数は 2
5. 集合には順番がない。表示の順は実行ごとに変わることがある

比べる・組み合わせる

2. 分ける

書き方	意味	例 (v = 5.0)
== / !=	等しい / 等しくない	v == 5.0 → True
< <= > >=	大小	v < 4.5 → False
and	両方とも成り立つ	4.5 <= v and v <= 5.5 → True
or	どちらかが成り立つ	v < 4.5 or v > 5.5 → False
not	逆にする	not v < 4.5 → True

- = は「入れる」、== は「等しいか比べる」。取り違えやすい
- 4.5 <= v <= 5.5 のように、続けて書くこともできる

if / elif / else: 条件で処理を分ける


```
def check(v):
    if v < 4.5:
        return "低い"
    elif v <= 5.5:
        return "正常"
    else:
        return "高い"

print(check(3.3))    # 低い
print(check(5.0))    # 正常
```

字下げ (行の先頭の空白) が、どこまでが if の中かを表す

コードの読み方

1. `if` の条件が成り立てば、その下の字下げした行を実行する
2. 成り立たなければ、次の `elif` の条件を調べる
3. どれにも当てはまらなければ `else` の行を実行する
4. 上から順に調べ、最初に当てはまった所だけを実行する

for と range: 決まった回数くり返す

3. 回す

```
for i in range(3):
    print(i)          # 0, 1, 2

volts = [3.3, 5.0, 4.8]
for v in volts:
    print(v * 2)      # 6.6, 10.0, 9.6
```

コードの読み方

1. `range(3)` は 0, 1, 2 の 3 つ。`i` に順に入る
2. `for v in volts`: リストの値を、先頭から 1 つずつ `v` に入れる
3. 字下げした行が、くり返しの中身

for で合計と最大を求める

3. 回す — 課題 2

```
volts = [3.5, 5.0, 4.5]

total = 0
biggest = volts[0]
for v in volts:
    total = total + v
    if v > biggest:
        biggest = v

print(total, biggest)   # 13.0 5.0
```

コードの読み方

1. 合計は 0 から始める。最大は、とりあえず先頭の値にしておく
2. 1 つずつ見て、`total` に足していく
3. 今の最大より大きければ、`biggest` を入れ替える
4. くり返しが終わると、合計と最大が求まっている

while: 条件が成り立つ間くり返す

3. 回す — 課題 3

```
value = 1
count = 0
while value < 100:
    value = value * 2
    count = count + 1

print(value, count)   # 128 7
```

コードの読み方

1. `while 条件:` は、条件が成り立つ間、中身をくり返す
2. 毎回 `value` を 2 倍にし、回数 `count` を 1 増やす
3. `value` が 100 以上になったら止まる (1 → 2 →⋯→ 128 で 7 回 2 倍した)
4. 中で `value` を変え忘れると、いつまでも止まらない

辞書で数える

4. 組み合わせる — 課題 4

```
parts = ["R", "C", "R", "L", "R"]

counts = {}
for p in parts:
    if p in counts:
        counts[p] = counts[p] + 1
    else:
        counts[p] = 1

print(counts)   # {'R': 3, 'C': 1, 'L': 1}
```

コードの読み方

1. 空の辞書 {} から始める
2. 1 つずつ見て、表にすでにあれば、回数を 1 増やす
3. まだ表に無ければ、回数 1 で追加する
4. 最後に「部品の種類 → 個数」の表ができる

リストをスタックとして使う

4. 組み合わせる — 課題 5

```
stack = []
stack.append(3)      # [3]
stack.append(4)      # [3, 4]

top = stack.pop()    # 4 を取り出す
print(top, stack)    # 4 [3]
```

コードの読み方

1. スタック: 積み重ねた皿のように、最後に置いたものから取り出す入れ物
2. append: 一番上に積む
3. pop(): 一番上から取り出す(取り出したものはリストから消える)

辞書に関数を入れて、記号で計算を選ぶ

4. 組み合わせる — 課題 5

```python
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

OPS = {"+": add, "-": sub}

print(OPS["+"](3, 4))   # 7
```

コードの読み方

1. 関数も値なので、辞書に入れられる
2. `OPS["+"]` で関数 `add` を取り出す
3. 後ろに `(3, 4)` を付けて呼ぶと、`add(3, 4)` と同じ
4. `if` を並べなくても、記号から計算を選べる

逆ポーランド記法の電卓

4. 組み合わせる — 課題 5

数を先に、記号を後に書く書き方を逆ポーランド記法という。例: `3 4 +` は `3 + 4`。

```
stack = []
for t in "3 4 +".split():
    if t in OPS:
        b = stack.pop()
        a = stack.pop()
        stack.append(OPS[t](a, b))
    else:
        stack.append(int(t))

print(stack[-1])   # 7
```

コードの読み方

1. 文字列を空白で区切り、1 つずつ見る
2. 記号なら、上から 2 つ取り出して計算し、結果を積む
3. 数なら、`int(t)` で数に変えて積む
4. 先に取り出す `b` が右側の値 (`10 3 -` は `10 - 3`)

次回は、この電卓に「新しい命令を作る」仕組みを足して、小さな言語を作る

まとめと課題 I

まとめ

やりたいこと	使う道具
値を順に並べる	リスト `[...]`
名前で値を引く	辞書 `{鍵: 値}`
重なりを除く・共通を探す	集合 `{...}`, `&`
条件で分ける	`if` / `elif` / `else`
決まった回数・全部を回す	`for`, `range`
条件が成り立つ間回す	`while`

まとめと課題 II

まとめ

課題	書くもの	配点
1	`check_voltage` (分岐)	15
2	`total_and_max` (for)	20
3	`count_doublings` (while)	15
4	`count_items` (辞書で数える)	20
5	`rpn` (スタックと関数の辞書)	15
発展 1・2	`resistance`, `common_parts`	10, 5

付録: 講義では扱わない補足

質問が出たときに使う。

- よくある間違い

よくある間違い

付録

症状	よくある原因
`IndentationError`	字下げがそろっていない。`if` / `for` の中は同じ数の空白で下げる
`SyntaxError` (`if v = 5:`)	比べるときは`==`。`=` は入れるとき
`KeyError`	辞書に無い鍵を引いた。先に `in` で調べる
`IndexError`	空のリストから `pop()` した、または番号が大きすぎる
止まらない	`while` の中で、条件に使う変数を変えていない
答えが `None`	関数に `return` を書いていない