Python 入門: 関数と辞書

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 / 06

本日の流れ: 関数を「書く」から「組み合わせる」へ

段階	できるようになること	主な道具	課題
0. 復習	値・リスト・辞書を使う	`list`, `dict`, `for`	
1. 関数を書く	定義し、呼び、自分自身も呼ぶ	`def`, 引数, 再帰	0, 1
2. 関数を値にする	変数・辞書に入れて選ぶ	関数を入れた辞書	2
3. 答えを覚える	同じ計算をくり返さない	答えを入れた辞書	3
4. 表に登録する	文字列から関数を実行する	関数を入れた辞書	4, 4-b

- 今日のゴール: 関数を「部品」として、変数や辞書に入れて使えるようになる
- 流れ: 関数を書く → 関数を辞書に入れる → 答えを辞書に覚える →文字列から呼ぶ
- 各スライドの右側に、コードの読み方を書いてある

復習: Python の書き方


```python
n, name, flag = 42, "Alice", True
x = 10
x = "文字列"

xs = [3, 1, 4]
xs.append(9)
xs.pop()

d = {"a": 1}
d.get("z", 0)

[x**2 for x in range(5) if x % 2 == 0]
```

コードの読み方

- 1 行目: 数・文字列・真偽値を、まとめて変数に入れる
- x には、あとから別の種類の値も入れ直せる
- xs: リスト。append で後ろに足し、pop で後ろから取り出す
- d: 辞書。get("z", 0) は ”z” を探し、無ければ 0 を返す (エラーにならない)
- 最後の行: 0〜4 のうち偶数の 2 乗を並べる。結果は [0, 4, 16]

復習: リストと繰り返し処理・例外

0. 復習

```python
xs = [3, 1, 4]
xs.append(9)          # [3, 1, 4, 9]

names = ["Alice", "Bob"]
scores = ["73", "92"]

for i, name in enumerate(names):
    print(i, name)
for a, b in zip(names, scores):
    print(a, b)

s = "1234"
try:
    n = int(s)
except ValueError:
    n = 0

print(n)
```

コードの読み方

- `enumerate`:「何番目か」と値を、一緒に取り出す。`0 Alice` / `1 Bob`
- `zip`: 2 つのリストを、同じ位置どうしで組にする。`Alice 73` / `Bob 92`
- `try` / `except`: 数に変えられなければ、`n = 0` にする
- `"1234"` は数に変えられるので、`n` は 1234

復習: 辞書

0. 復習

```
score = {"alice": 80, "bob": 92}
score["bob"]
score.get("carol", 0)
```

- この後、**値に関数を入れた辞書**を使う

コードの読み方

- 辞書は「名前 → 値」の対応表
- `score["bob"]` は 92
- `score.get("carol", 0)`: carol は表に無いので 0

関数: 名前を付けた処理のまとまり

1. 関数を書く

```
def greet(name, suffix="!"):
    return "Hello, " + name + suffix

greet("Alice")
greet(name="Bob", suffix=".")
```

コードの読み方

- def で関数を作る。かっこの中が、受け取る値 (引数)
- return が、返す値。return が無ければ None を返す
- suffix="!" は、省略したときの値 (デフォルト値)
- greet("Alice") は "Hello, Alice!"
- greet(name="Bob", suffix=".") は "Hello, Bob."

関数は自分自身も呼べる (再帰)

1. 関数を書く

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

fib(10)    # ==> 55
```

コードの読み方

- `fib(n)` は、n 番目のフィボナッチ数 (0, 1, 1, 2, 3, 5, 8 ...)。前の 2 つを足していく
- `n <= 1` なら、そのまま `n` を返して終わり (**止まる条件**)
- それ以外は、`fib(n - 1)` と `fib(n - 2)` を足す。自分自身を 2 回呼んでいる
- 止まる条件が無いと、永遠に呼び続ける

同じ *fib(2)* などが、何度も計算されている。後で効いてくる

関数の中から、外の変数を書き換えて呼び出しを数える

1. 関数を書く

```
CALLS = 0

def fib(n):
    global CALLS
    CALLS += 1
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

CALLS = 0
fib(10)
print(CALLS)    # 177
```

コードの読み方

- CALLS は、呼ばれた回数を数える変数
- 関数の中で CALLS += 1 だけ書くと、外の CALLS ではなく、関数だけの変数になってしまう
- global CALLS と書くと、「外の CALLS を使う」という宣言になる
- fib が呼ばれるたびに 1 増える。fib(10) では 177 回

竹内関数: 値は単純でも、呼び出し回数が爆発する

1. 関数を書く — 課題 1

```
tarai(x, y, z) = y                          (x <= y のとき)
               = tarai(tarai(x-1, y, z),
                       tarai(y-1, z, x),
                       tarai(z-1, x, y))    (それ以外)
```

呼び出し	回数
`tarai(10, 5, 0)`	343,073 回
`tarai(12, 6, 0)`	12,604,861 回

- 読み方: `x` が `y` 以下なら `y` を返して終わり。そうでなければ、`tarai` を 3 回呼び、その 3 つの結果でもう一度 `tarai` を呼ぶ
- 形は `fib` と同じ: 止まる条件 → 自分を呼ぶ
- 引数は小さい整数なのに、1,260 万回も呼ばれる
- **同じ引数の組が何度も現れる。**これを後で減らす

関数は値なので、変数に入れたり、渡したりできる

2. 関数を値にする

```
def square(x):
    return x * x

f = square
f(5)                    # ==> 25

def apply_twice(func, x):
    return func(func(x))

apply_twice(square, 3)  # ==> 81
```

コードの読み方

- `f = square`: 関数 `square` を、`f` という別の名前でも呼べるようにする
- `f(5)` は `square(5)` と同じで 25
- `apply_twice(func, x)`: 受け取った関数 `func` を、`x` に 2 回かける
- `apply_twice(square, 3)` は `square(square(3))` = `square(9)` = 81

辞書に関数を入れると、名前で呼び分けられる

2. 関数を値にする — 課題 2

```
def calc(op, a, b):
    if op == "add":
        return a + b
    elif op == "sub":
        return a - b
    else:
        return -1
```

```
import operator
OPS = {"add": operator.add,
       "sub": operator.sub,
       "mul": operator.mul}

def calc(op, a, b):
    if op not in OPS:
        raise ValueError(op)
    return OPS[op](a, b)

calc("add", 2, 3)    # 5
```

- 左: 演算が増えるたびに `elif` が増える
- 右: `OPS` は「名前 → 関数」の表。`OPS[op]` で関数を取り出し、`(a, b)` で呼ぶ
- `operator.add` は、`+` と同じ働きをする関数。表に 1 行足すだけで演算を増やせる
- 表に無い名前は、`ValueError` (エラー) にする

答えを辞書に覚えて、同じ計算をしない

3. 答えを覚える — 課題 3

```
TARAI_CACHE = {}

def tarai_memo(x, y, z):
    key = (x, y, z)
    if key in TARAI_CACHE:
        return TARAI_CACHE[key]
    if x <= y:
        result = y
    else:
        result = tarai_memo(
            tarai_memo(x-1, y, z),
            tarai_memo(y-1, z, x),
            tarai_memo(z-1, x, y))
    TARAI_CACHE[key] = result
    return result
```

コードの読み方

1. `TARAI_CACHE = {}`: 答えを書きとめる、空のメモ帳 (辞書)
2. `key = (x, y, z)`: 3 つの引数を 1 組にして、メモ帳の見出しにする
3. メモ帳に見出しがあれば、計算せずに、書きとめた答えを返す
4. 無ければ、`tarai` と同じ計算をする。呼ぶ相手は `tarai_memo`
5. 答えをメモ帳に書いてから返す

答えを覚えるだけで、計算が 5 万分の 1 になる

3. 答えを覚える — 課題 3

`tarai(12, 6, 0)`	計算する回数	時間
覚えない (課題 1)	12,604,861 回	約 1 秒
辞書に覚える (課題 3)	**225 回**	一瞬

- 225 は、課題 1-b で数えた「引数の組の数」と同じ
- 同じ引数の組は、2 回目からメモ帳を見るだけで終わる
  同じ計算をくり返すなら、一度求めた答えを保存して使い回す

命令の名前と関数を、表に登録する

4. 表に登録する — 課題 4

```
COMMANDS = {}

def shout(text):
    return text.upper() + "!!"

def rev(text):
    return text[::-1]

COMMANDS["shout"] = shout
COMMANDS["rev"] = rev
```

コードの読み方

- `COMMANDS = {}`:「命令の名前 → 関数」の表。最初は空
- `shout`, `rev` は、ふつうの関数
- `COMMANDS["shout"] = shout`: 名前 `"shout"` で、関数 `shout` を表に入れる (課題 2 の OPS と同じ形)
- 命令を増やすときは、関数を書いて 1 行登録するだけ

文字列による命令実行 I

4. 表に登録する — 課題 4-b

```
def run(line):
    name, *args = line.split()
    if name not in COMMANDS:
        raise ValueError(name)
    return COMMANDS[name](*args)

run("shout hello")    # 'HELLO!!'
```

コードの読み方

1. `"shout hello".split()` は `["shout", "hello"]`
2. `name, *args = ...`: 先頭を `name` に、残りを `args` に入れる
3. 表に無い名前なら、エラーにする
4. あれば、`COMMANDS["shout"]("hello")`。つまり `shout("hello")` を呼ぶ

文字列による命令実行 II

4. 表に登録する — 課題 4-b



Figure: 文字列から命令を実行する流れ

要点

- **関数は値**: 変数や辞書に入れたり、引数に渡したりできる
- **辞書に関数を入れる**: `if` の連鎖が、表の参照になる
- **答えを辞書に覚える**: 同じ計算をくり返さずに済む

課題一覧

- 必須課題: 0〜4-b
- 発展課題: 任意 (講義では扱わない)

#	課題	ねらい	対応する段階
0	関数を書く・渡す	引数、デフォルト値、関数を引数に	1, 2
1	竹内関数を実装し、呼び出し回数を数える	再帰、爆発する計算量	1
2	辞書に関数を入れて電卓	辞書に関数を入れる	2
3	`tarai_memo` で答えを辞書に覚える	同じ計算をくり返さない	3
4, 4-b	`COMMANDS` に登録し、`run()` で実行	文字列から関数を実行する	4

課題の進め方

- 「Colab で開く」から課題を開き、「ドライブにコピーを保存」
- 各課題の **例題セル** を実行して動きを見てから、`TODO` を埋める
- 直下の確認セルが `OK` になれば、その課題は完了
- 確認セルの保存
  - 教員は別のテストケースで採点するため、削除しても採点結果は不変

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。

- デコレータ: 関数を包んで、機能を足す (`@memo`, `@command`)
- クロージャ: 外側の引数を覚えた関数を返す
- 復習: 値と変数
- 復習: 型・文字列・演算子
- 実行時の機能追加
- よくある落とし穴: 新しく値を返すか、元を書き換えるか

デコレータ: 関数を包んで、機能を足す I

付録

```
def logged(func):
    def wrapper(*args):
        print(f"{func.__name__}{args}")
        return func(*args)
    return wrapper

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

add = logged(add)
add(2, 3)    # add(2, 3) と表示され、5 を返す
```

コードの読み方

1. logged は、関数 func を受け取る
2. 中で、新しい関数 wrapper を作る。wrapper は「名前と引数を表示する → 元の func を呼ぶ」
3. wrapper を返す
4. add = logged(add) で、add が「表示つきの add」に入れ替わる
5. *args は、受け取った引数をそのまま元の関数へ渡すための書き方

- add = logged(add) は、def add の上に@logged と書いても同じ意味
- 元の関数 add の中身は書き換えない

デコレータ: 関数を包んで、機能を足す II

付録

- 外側の `func` を `wrapper` が覚えている仕組みは、次の「クロージャ」のスライド

デコレータによる計算結果の再利用

付録

```
def memo(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memo
def tarai(x, y, z):
    if x <= y:
        return y
    return tarai(tarai(x-1, y, z),
                 tarai(y-1, z, x),
                 tarai(z-1, x, y))
```

コードの読み方

1. `cache = {}`: 計算結果を書きとめる空のメモ帳 (辞書)
2. 引数 `args` が、メモ帳に無ければ、元の関数を呼んで、答えを書きとめる
3. メモ帳にあれば、元の関数は呼ばずに、書きとめた答えを返す
4. `@memo` を付けると、`tarai` の呼び出しが、すべてメモ帳を通る

- `tarai` の中身は変えていない。本編の `TARAI_CACHE` と同じことを、`@memo` の 1 行で行う

関数を定義と同時に、表へ登録する

付録

```
COMMANDS = {}

def command(name):
    def deco(func):
        COMMANDS[name] = func
        return func
    return deco

@command("shout")
def shout(text):
    return text.upper() + "!!"
```

関数の定義位置がそのままコマンド表になる

コードの読み方

- `COMMANDS = {}`:「命令の名前 → 関数」の表。最初は空
- `@command("shout")` を関数の上に書くと、その関数が`"shout"` という名前で表に入る
- 登録しても、関数は元のまま使える
- 本編の `COMMANDS["shout"] = shout` を、定義と同時に行う書き方

クロージャ: 外側の引数を覚えた関数を返す

付録

- 関数内での別の関数の定義
- 外側の引数 n を保持する返り値の関数

```
def multiplier(n):
    def inner(x):
        return x * n      # 外側の n を覚えたまま
    return inner

double = multiplier(2)
double(10)   # 20
```

- デコレータの wrapper が外側の func を覚えているのも、この仕組み
- 名前を覚える必要はない。「関数が、作られた場所の変数を覚えている」でよい

復習: 値と変数

付録

- 値は整数・文字列・リストなどのオブジェクト
- 変数は値に付ける名前であり、型を固定しない
- `type()` による値の動的型の確認

```
x = 10
name = "Alice"
x = "文字列"        # 型を持つのは値

type(x)               # str
type(name)            # str
```

復習: 型・文字列・演算子

付録

- `/` の常に `float` という結果と、`//` の切り捨て
- 整数の桁あふれなしと、処理速度の低下
- 変更不可の文字列と、`s[0] = "X"` のエラー

```
name, score = "Alice", 92.5

7 / 2       # 3.5
7 // 2      # 3
2 ** 200    # 桁あふれしない

s = "スクリプト"
s[0], s[-1], s[0:3], s[::-1]
f"{name:>10} | {score:6.2f}"
f"{score = }"                    # print デバッグ用
```

実行時の機能追加

付録

- コンパイル済みのプログラムは機能追加時に再コンパイルする必要
- ディスパッチテーブルがあれば辞書への関数の追加だけで実現

```
def min(x, y):
    if x < y: return x
    else: return y
def add(x, y):
    return x + y

OPS = {"min": min, "add": add}
```

```
def max(x, y):
    if x > y: return x
    else: return y

OPS["max"] = max              # ここで語彙がひとつ増える

def calc(name, arg1, arg2): # 計算を行う本体
    return OPS[name](arg1, arg2)

calc("max", 3, 9)             # 9
```

よくある落とし穴: 新しく値を返すか、元を書き換えるか

付録

- ソート関数 (sort, sorted) による例
  - 新しく値を返す: sorted
  - 元を書き換える: sort() (None を返す)
- 引数のリストの変更
  - 呼び出し元にも及ぶ
  - デフォルト値の [] は定義時に 1 回だけ生成される

```
xs = [3, 1, 2]
result = xs.sort()     # None が返る — xs =
↪  xs.sort() は間違い
print(xs, result)      # [1, 2, 3] None

xs = [3, 1, 2]
xs = sorted(xs)        # 新しいリストを返す
print(xs)              # [1, 2, 3]
```

```
def add(xs, item):
    xs.append(item)    # 呼び出し元のリストが書き換わる
    return xs

def bad(item, xs=[]):
    xs.append(item)    # bad(1) -> [1],  bad(2) ->
↪  [1, 2]
    return xs

print(bad(1))  # [1]
print(bad(2))  # [1, 2] — デフォルト共有の落とし穴
```