

Python 入門

スクリプト言語演習

伊澤侑祐

本日の内容

- Python の基本: 値・変数・コレクション
 - 関数の定義、引数、返り値
- 動的言語としての Python
 - 関数の辞書への登録とデコレータ

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

```
f = square          # 変数に代入できる  
f(5)                # ==> 25  
square.__name__     # ==> 'square'
```

Python の書き方

Python の基本

- 動的型付け
 - 型宣言不要、実行時に型が決定 (runtime type, 動的型)
- 副作用 (side effect)
 - 変数に別の型の値も代入可能
- コレクション (標準データ構造) 操作
 - リスト (list): append, pop, etc.
 - 辞書 (dictionary): get, set, etc.
 - * キーによる辞書の参照も可能

```
n, name, flag = 42, "Alice", True  
x = 10  
x = "文字列"          # 型を持つのは値
```

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

```
d = {"a": 1}
```

```
d.get("z", 0)          # 無いキーでも落ちない
```

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

値と変数

Python の基本

- 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
```

型・文字列・演算子

Python の基本

- `/` 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
```

制御構造

Python の基本

- 条件分岐: `if` 条件 1: ... `elif` 条件 2: ... `else`: ...
 - 条件 (condition): `0 < x`, `x != True`, `x == False`, etc.

* $0 < x < 100$ のような比較の連鎖

- False と判定される空のコレクション: 0, None

```
x = 90
if 0 < x < 100:
    print("x is between 0 and 100")
else:
    print("x is out of 0--10")

x = False

if not x: # x != False
    print("x is False")
```

リストと繰り返し処理・例外

Python の基本

- リスト: 順番を持つ値の集まり
- リストを操作する便利な関数
 - enumerate: 添字と値を得る
 - zip: 2つのリストの対応する値を得る

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

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

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

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

```
print(n)
```

辞書

Python の基本

- 辞書: 名前から値を引く対応表
- 繰り返し処理などで使用

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

関数

Python の基本

- 関数の定義:

```
def 関数名(引数名 1, 引数名 2, ...):
    関数ボディ    # インデントはスペース 4 つ
    ...
    return        # 値を返す

• 名前付き引数を指定可能: def 関数名(名前 1= デフォルト値): ...

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

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

クロージャ

Python の基本

- 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
```

--

ディスパッチテーブル

動的言語としての Python

- 名前による辞書内の関数の参照
 - if の連鎖に伴う関数の肥大化を避けられる
 - 演算子追加後も 1 行の calc

```
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(f"知らない演算: {op}")
    return OPS[op](a, b)

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

実行時の機能追加

動的言語としての Python

- 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
```

@ 記法の糖衣構文”デコレータ”

動的言語としての Python

- logged: 関数を受け取り、別の関数を返す
 - add = logged(add) と同じ意味
- @deco: 定義直後の f = deco(f) を表す

```
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)
```

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

```
@logged
```

```
def add(a, b):
```

```
    return a + b
```

```
add(2, 3) # ログ出力 + 5
```

デコレータによる計算結果の再利用

動的言語としての Python

- クロージャが保持する辞書 cache
 - タプルをキーにした、引数ごとの結果の保存
 - 関数本体を変えない、呼び出しを包む処理

```
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))

```

1 行で得られる効果

動的言語としての Python

- tarai の本文は変更不要
- 同じ引数に対する再計算の削減
- 1 行の追加による、呼び出し回数の約 5 万分の 1 への削減

tarai(12, 6, 0)	呼び出し回数	時間
@memo なし	12,604,861 回	約 1 秒
@memo あり	225 回	一瞬

同じ計算の繰り返しでは、一度求めた答えを保存し再利用するのが最適化のポイント

登録デコレータ

動的言語としての Python

- command: 関数を辞書に追加して返す
- @command("shout") による COMMANDS への登録
- 定義と登録の同時実行による一覧の一貫性

```
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

文字列による命令実行

動的言語としての Python

- 先頭の語を命令、残りを引数として解釈
- 辞書にない命令へのエラー
- 文字列から処理系を動かす最小の骨格

```
COMMANDS = {}
```

```
def command(name):
```

```
    def deco(func):
```

```
        COMMANDS[name] = func
```

```
        return func
```

```
    return deco
```

```
@command("shout")
```

```
def shout(text):
```

```
    return text.upper() + "!!"
```

```
def run(line):
```

```
    name, *args = line.split()
```

```
    if name not in COMMANDS:
```

```
        raise ValueError(f"知らない命令: {name}")
```

```
    return COMMANDS[name](*args)
```

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



要点: 動的言語としての Python

- 関数は値: 変数・辞書への格納
 - ディスパッチテーブル: if の連鎖を辞書ひとつに集約
 - デコレータ: 中身を変えない機能追加と、@deco の `f = deco(f)` への展開
-

発展: 動的なメソッド生成

動的言語としての Python

- メソッドを 1 つも定義しない Echo
- 名前に応じて関数を返す `__getattr__`
- 強力さと、定義を追いにくいという難点

```
class Echo:
    def __getattr__(self, name):
        def method(*args):
            print(f"{name} が {args} で呼ばれました")
        return method
```

```
e = Echo()
```

```
e.jump(3)
```

```
e.どんな名前でも(" いける")
```

読み手には追いきにくい仕組み。課題では不要

よくある落とし穴: 新しく値を返すか、元を書き換えるか

動的言語としての Python

- 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] ーデフォルト共有の落とし穴

```

課題一覧

- 必須課題: 1～4
- 発展課題: 任意

#	課題	ねらい
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
2	ディスパッチテーブルで電卓	辞書に関数を入れる
3	@memo を書き、課題 1 に組み込む	デコレータ、クロージャ
4	@command と run() でコマンド表	字句解析、ディスパッチ

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 TODO と、直下の確認セルの OK
- 確認セルの保存
 - ー 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