09 Python基礎

関数の定義2

- 教科書 P.34「特殊引数」〜P.39「引数リストのアンパック」までを熟読してください。

`*` を使った引数（位置引数のタプル化）

> https://www.yoheim.net/blog.php?q=20160609#t3

```
# 必須引数と位置引数の組み合わせ
def print_more(required1, required2, *args):
    print("required1:", required1)
    print("required2:", required2)
    print("other:", args)


# 引数2個の場合
print_more(1, 2)
# required1: 1
# required2: 2
# other: ()

# 引数3個の場合
print_more(1, 2, 3)
# required1: 1
# required2: 2
# other: (3,)

# 引数5個の場合
print_more(1, 2, 3, 4, 5)
# required1: 1
# required2: 2
# other: (3, 4, 5)

# 必須引数が足りなければエラー
print_more(1)
# Traceback(most recent call last):
#     File "<stdin>", line 1, in <module >
#     print_more(1)
# TypeError: print_more() missing 1 required positional argument: 'required2'
```

- 引数の数が決まっていない関数の呼出を処理することが可能になる。

    - 例えば、print( )のような…
- 受け取った値は、タプルとなる。
    - 分解するには
        - `a1,a2 = args` のように別々の変数に格納できる。
        - `args[0]` のように使用しても良い。

確認

- もう一度、ptint関数の定義ドキュメントを確認してみよう

> https://docs.python.org/ja/3.9/library/functions.html#print

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- 最初の `*objects` が、タプル化した受け取り引数

```
print(1,"ab",[10,20,30])
```

    - このように呼び出した場合、print関数内では、すべての引数をタプルとして受け取る。

```
objects => (1,"ab",[10,20,30])
```

```
objects[0] => 1
objects[1] => "ab"
objects[2] => [10,20,30]
```

      となる。
- 引数の個数が可変の関数を作成する場合に使用する

`**` を使った引数（キーワード引数の辞書化）

- 仮引数の先頭に `**` をつけることで、キーワード引数を辞書としてまとめることが可能になる。

> https://qiita.com/hikurochan/items/e6db0feb24f754361d4f

【kwargs.py】

```
# argsとkwargsを併用する場合は、順番に注意
```

```
def say_dic(word, *args, **kwargs):
    print("word=", word)
    print("args=", args)
    print("kwargs", kwargs)
    for k, v in kwargs.items():
        print(k, v)
    print()


# 普通に呼び出す方法
say_dic('hello', 'Mike', 1, desert='banana', drink='beer')

# 辞書にしてから渡す方法
t = {'math': 15, 'science': 100}
say_dic('hi', 'Nancy', 2, **t)
```

【実行結果】

```
word= hello
args= ('Mike', 1)
kwargs {'desert': 'banana', 'drink': 'beer'}
desert banana
drink beer

word= hi
args= ('Nancy', 2)
kwargs {'math': 15, 'science': 100}
math 15
science 100
```

一般的な可変長引数の仮引数の名称

- 決まりではないが、一般的に良く使用される。
    - タプルの場合： `*args`
    - 辞書の場合： `**kwargs`
- C言語の argc,*argv と同じ感じ

```
int main(int argc, char const *argv[])
```

    - `argc` は引数の個数
    - `argv[]` はそれぞれの引数の値

練習1

- 以下のプログラムを作成してください。【func_any.py】
    - 1日の食事を記録する前処理プログラム。カンマ区切りで出力する。
    - 朝食（B)、昼食（L)、夕食（D）の記号と、メニューを入力する
        - 区切りは自分で決めてOK
        - 順序は問わない
    - 空の入力で、入力処理を終える
    - 食べてない食事は「-」を出力するものとする。
    - listで表示できればOK（単なるカンマ区切りでもOK）

【実行結果1】

```
朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：L,カレー
朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：D,ステーキ
朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：B,トースト
朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：
['トースト', 'カレー', 'ステーキ']

もしくは
トースト,カレー,ステーキ
```

【実行結果2】

```
朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：B,小倉トースト
朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：D,中華飯
朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：
['小倉トースト', '-', '中華飯']

もしくは
小倉トースト,-,中華飯
```

【func_any.py】

- 入力データをキーワード引数で受け渡しする例

```
def out_csvdata(**kwargs):
    '''
    辞書型のデータをcsvで出力する
    Breakfast,Lunch,Dinnerの順に出力する
    '''
```

```
   ⋮

# main
eat = {}
while True:
    menu = input("朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：")
    if menu == '':
        break
    token, menu = menu.split(',')
    if token in ['B', 'L', 'D']:
        eat[token] = menu
    else:
        print('記号が間違っています。登録しません')
        continue


out_csvdata(**eat)
```

【func_any_2.py】

- 入力データの辞書を引数として受け渡しする例

```
def out_csvdata(eat_dict):
    '''
    辞書型のデータをcsvで出力する
    Breakfast,Lunch,Dinnerの順に出力する
    '''
   ⋮

# main
eat = {}
while True:
    menu = input("朝食(B) 昼食(L) 夕食(D)と食べたものを入力してください：")
    if menu == '':
        break
    token, menu = menu.split(',')
    if token in ['B', 'L', 'D']:
        eat[token] = menu
    else:
        print('記号が間違っています。登録しません')
        continue


out_csvdata(eat)
```

練習2

- 以下のプログラムを作成してください。【func_tuple.py】
    - 可変引数を受け取り内容を出力する関数 `out_tuple()`
        - 引数の個数と、各引数の型、引数の値を出力する
    - 動作確認するためのmain部分

【func_tuple.py】

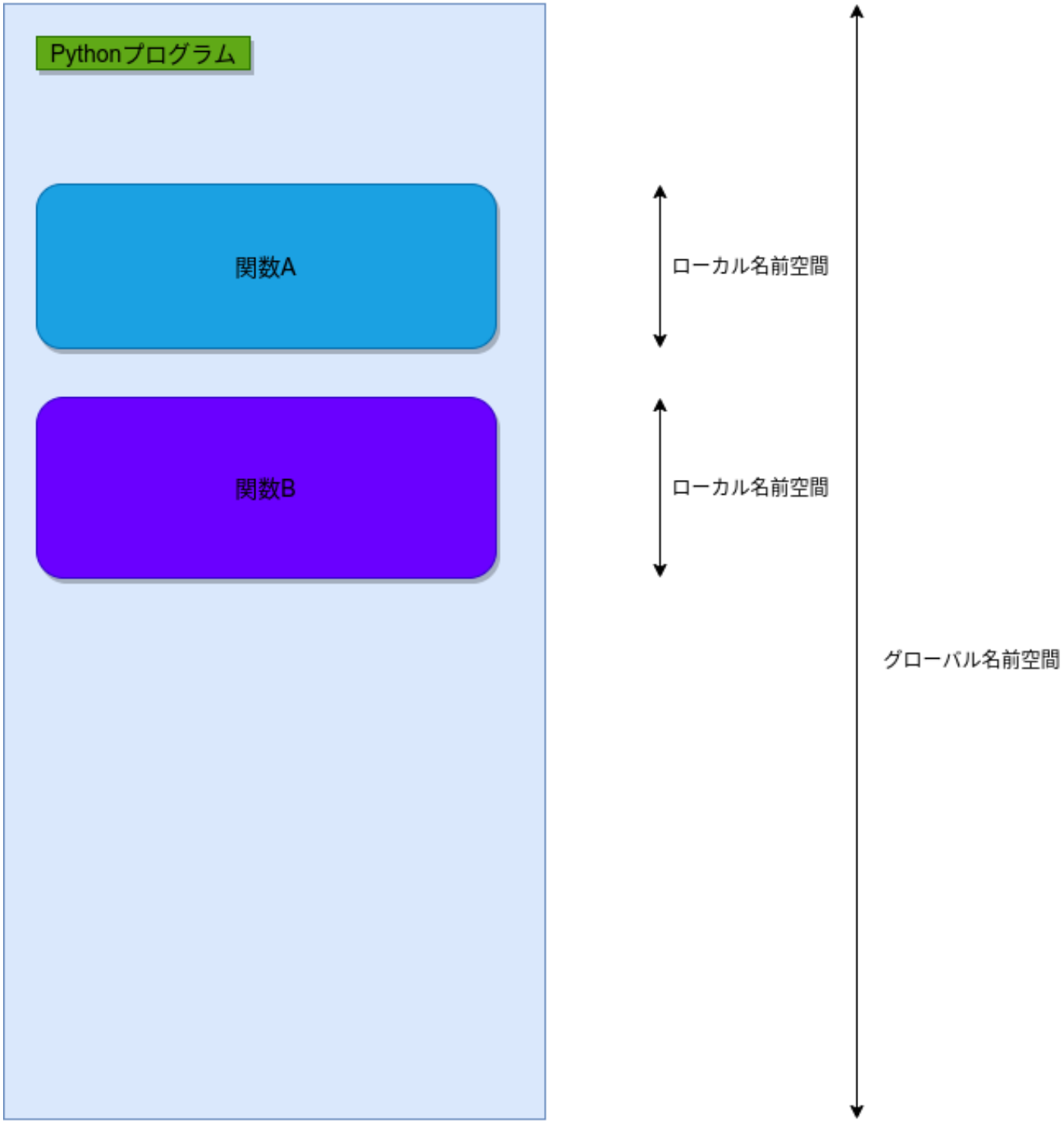
```
def out_tuple(*args):
    pass


# main
out_tuple(1, "ab", [10, 20], (5, 6))
out_tuple("xyz", 100)
out_tuple()
```

【実行結果】（例）

```
引数の個数： 4
 引数の型: <class 'int'> 引数の値: 1
 引数の型: <class 'str'> 引数の値: ab
 引数の型: <class 'list'> 引数の値: [10, 20]
 引数の型: <class 'tuple'> 引数の値: (5, 6)
引数の個数： 2
 引数の型: <class 'str'> 引数の値: xyz
 引数の型: <class 'int'> 引数の値: 100
引数の個数： 0
```

関数とスコープ

- 変数の利用できる範囲の区別
    - グローバル名前空間→グローバル変数
    - ローカル名前空間→ローカル変数
- 考え方は以下の図



- ローカル名前空間からは、グローバル名前空間を参照できる。逆は不可。
- ローカル名前空間内で、グローバル変数を書き換えるには、globalを使用する。

- 確認してみると良い（以下のようなプログラムを作成しテストする）

【local_global.py】

```
var_global = 100
```

```
def func():
    # global var_global
    # var_global = 200
    print(var_global)
    var_local = 'A'
    # var_global = 'A'  # はどうなる？
    print(var_local)


print(var_global)
func()
print(var_global)
print(var_local)
```

- コメントを外した場合どうなる？

```
    global var_global
    var_global = 200
    ⋮
    var_global = 'A'
```

globalの使い方

- ローカル名前空間内から、グローバル名前空間内の変数へ代入する場合は、global宣言が必要

> https://uxmilk.jp/12505

注意！

- リストをグローバル変数で使用した場合、リストの要素は変更が可能。
    - globalは使用しないでも変更可能。

```
vote_box = []
label = '票'


def vote():
    print('投票します')
    vote_box.append(label)
```

```
def reset_box():
    print('箱を空にします')
    vote_box.clear()


def check_box():
    print('票の数は{}です'.format(len(vote_box)), end=" ")
    print('vote_box:', vote_box)


# main
check_box()
for i in range(10):
    vote()

check_box()
reset_box()
vote()
vote()
check_box()
```

【実行結果】

```
> python vote.py
票の数は0です vote_box: []
投票します
投票します
投票します
投票します
投票します
投票します
投票します
投票します
投票します
投票します
票の数は10です vote_box: ['票', '票', '票', '票', '票', '票', '票', '票', '票', '票']
箱を空にします
投票します
投票します
票の数は2です vote_box: ['票', '票']
```

関数オブジェクト

- 関数自体がオブジェクトであるため、他の数値、文字列等と同じように使うことができる。
    - 変数に関数を代入することができる。
    - 戻り値に関数を返すこともできる。

- どんな結果になるだろうか？

【func_obj.py】

```
functions = [sum, min, max]
number_list = range(1, 11)

for func in functions:
    print("Function: {}, Result: {}".format(func.__name__, func(number_list)))
```

練習3

- 以下のプログラムを作成してください。【select_func.py】
    - キーボードから、「a」または「b」を選択する
        - 空文字の時は終了する
        - 選択肢以外の場合は、再入力
    - 選択に応じて、実行するプログラムを切り替える。
        - ただし、ifを使わずに切り替えたい。（関数を辞書に格納して利用する）
    - 「c」を入力した場合、`func3()`を実行するように、関数を追加してください。
        - 関数動作は、func1,func2とは異なるメッセージを表示するようにしてください。

【実行例】

```
a=>Hello,b=>Goodbye
どちらを実行しますか？:a
Hello
a=>Hello,b=>Goodbye
どちらを実行しますか？:b
Goodbye
a=>Hello,b=>Goodbye
どちらを実行しますか？:c
どちらかを選択してください。
a=>Hello,b=>Goodbye
どちらを実行しますか？:
```

【select_func.py】

```python
# 関数を辞書で管理し、実行する
def func1():
    print("Hello\n")


def func2():
    print("Goodbye\n")


# 一般的なmainの記述方法
if __name__ == "__main__":
    # execute only if run as a script
    run_list = {'a': func1, 'b': func2}

  # 以下作成
```

ラムダ式

- 最近の言語には、ほぼ用意されている。無名関数とも呼ばれる。
- 関数を作らずに、関数同様の処理を行なう
    - 関数名を考えなくて良い。
    - 使う場所に直接記述するので、わかりやすい
    - シンプルに書ける
    - （欠点）複雑な処理は適さない

関数定義（再掲）

```
def 関数名(引数,引数…):
  処理
    return 戻り値
```

ラムダ式の書き方

```
lambda 引数：処理
```

- 単独で使用する場合もあるが、関数を引数として使用するような関数で用いられることが多い。
    - map
    - filter
    - sorted
- それぞれの使い方を確認してください。

練習4

- 以下のプログラムを作成してください。【lambda_str.py】
    - リスト「my_list」内の要素で、文字列の要素リストを作成する。
    - ラムダ式を使用して作成してください。

【実行結果】

```
> python lambda_str.py
[1, 2, 'ab', 'xyz', 5, 6, 7, 'zz']  ← my_listの出力
['ab', 'xyz', 'zz']          ← new_listの出力
```

【lambda_str.py】

```
my_list = [1, 2, 'ab', 'xyz', 5, 6, 7, 'zz']

new_list =

print(my_list)
print(new_list)
```

関数内関数

- 関数内で、関数を定義することが可能。
    - グローバル名前空間での関数名を増やさず処理ができる。
    - ローカル変数と同じ扱い（グローバル名前空間からの直接呼出はできない）

```
def func(a):
    """関数内関数のテスト"""
    def func_inner(b):
        print(b)
        return
    return func_inner(a)

# main
x = func(10)
x = func(5)
```

- 関数内関数を関数外から呼び出そうとすると、**not defined**となる

- 関数外への関数内関数の持ち出し方法1

```
def func():
    print("a")
    def func_inner():
        print("b")
    return func_inner

x=func()
x()
```

- 関数外への関数内関数の持ち出し方法2

```
def func():
    global y
    print("a")
    def func_inner():
        print("b")
    y = func_inner

func()
y()
```

> https://teratail.com/questions/187315

クロージャ,エンクロージャ、nonlocalについて

- 関数内関数を用いることで、クロージャを作ることができる。
    - 関数内関数は、内側で定義した関数を使って値を返す。
    - クロージャは、内側で定義した関数を返す。

```
def func(a):
    def func_inner():
        print(a)
        return
    return func_inner

# main
x = func(10)
x()
x = func(5)
x()
```

- `x = func( 10 )` は、xに関数オブジェクトが代入されるだけ
    - `x( )` で初めて実行される。

> クロージャ自体を理解するのは、サンプルをいくつも見て、読んで欲しい。
>
> 内容は分かりにくい（スコープ、グローバル変数の振る舞い、関数オブジェクト、関数内関数などを理解していないと分からない）部分もあるが、利用されることが増えているのでマスターして欲しい！

- python特有の動き（グローバル変数は、参照できるが変更できない）があるため、これまでリストを用いることが多かったが、python3になってnonlocalキーワードが用意された。

- nonlocalとは、ローカル変数ではなく、1つ外側のスコープの変数であることを宣言する。

> https://www.lifewithpython.com/2014/02/python-scope-namespace-rule-02.html

良く読んで理解しよう

> https://qiita.com/kangetsu121/items/2fd0c6ec2729fc711340
>
> https://www.lifewithpython.com/2014/09/python-use-closures.html
>
> https://codor.co.jp/python/closure

- クロージャを含む関数を、エンクロージャという。

```
def func():
    x = 3
    def add3(y):
        return y+x
    return add3

f = func()
print(f(4)) # 7
```

- `func()` →エンクロージャ
- `add3()` →クロージャ

> https://qiita.com/naomi7325/items/57d141f2e56d644bdf5f

デコレータ

- 既存の関数の中身を変更せず、機能追加するための方法
    - オブジェクトとして関数を渡す・受け取る
    - 関数内関数を利用する
    - `*args`, `**kwargs` を利用する
- 既存のライブラリなどを利用する場合などに有効

【例:deco_sample.py】

```
def startend(func):
    '''startとendを前後に出力するデコレータ'''
    def funcname(*args, **kwargs):
        print('[start]--')
        reslut = func(*args, **kwargs)
        print('--[end]\n')
        return reslut
    return funcname


def test(n):
    print('test->{}'.format(n))


# main
test(10)    # 通常の呼出


# デコレータ使用
deco_func = startend(test)
deco_func(20)


# 同一の関数名でデコレータ使用(testを再定義)
test = startend(test)
test(30)


# @記法を使用する場合以下のように記述する
@startend
def test2(n):
    print('test2->{}'.format(n))
```

```
test2(100)
```

【実行結果】

```
> python deco_sample.py
test->10
[start]--
test->20
--[end]

[start]--
test->30
--[end]

[start]--
test2->100
--[end]
```

練習5

- 以下のプログラムを作成してください。【deco_time.py】
    - 関数の実行時間を実行後に表示する run_time( )デコレータを作成する。
        - 呼び出した関数の実行時間を表示する
    - テスト用の関数を作成し、実行時間を表するmain部を作成する。

【実行結果】

```
> python deco_time.py
実行関数:test 実行時間：3.002683162689209
実行関数:test 実行時間：10.008330821990967
```

【deco_time.py】

```
import time


def run_time(func):
    '''実行時間を計測・出力するデコレータ'''
```

```
# main

@run_time
def test(n):
    for i in range(n):
        time.sleep(i)   # i秒ウェイトする


test(3)

test(5)
```

ヒント

- 現在時刻の取得は、timeモジュールの `time()` を使用する