

プログラミング初級 (Python)

関数

早稲田大学グローバルエデュケーションセンター

プログラムの中の小さなプログラム

プログラムは思ったとおりに動かない。
書いたとおりに動くのだ。

— 発言者不明

冗長なプログラム

```
1 num = int(input("Enter the range:¥t"))
2 for i in range(num):
3     for j in range((num - i) - 1):
4         print(end=" ")
5     for j in range((i + 1)):
6         print("*", end=" ")
7     print()
```

試行錯誤で完成！
あとはこれを使い
回せば…

```
1 num = int(input("Enter the range:¥t"))
2 for i in range(num):
3     for j in range((num - i) - 1):
4         print(end=" ")
5     for j in range((i + 1)):
6         print("*", end=" ")
7     print()
8 for i in range(num):
9     for j in range((num - i) - 1):
10        print(end=" ")
11    for j in range((i + 1)):
12        print("+", end=" ")
13    print()
```

同様の処理。

同様の処理。

実行結果の例

Enter the range: 5

```
  *
 * *
* * *
* * * *
* * * * *
```

Enter the range: 5

```
  *
 * *
* * *
* * * *
* * * * *
  +
 + +
+ + +
+ + + +
+ + + + +
```



Hmm...
something feels
off.

入力の二度手間

テキストエディタであれば基本的に一括でコピー＆ペーストが可能であるが、対話型インタプリタであれば個別にコピー＆ペーストするか、もう一度命令を入力し直す必要があるので大変さが身に染みる。そもそもコピー＆ペーストできない環境でのプログラミングもあり得る。また、修正が必要な場合に、変更すべき箇所が増える。

```
1 >>> num = int(input("Enter the range:¥t"))
Enter the range:      5
2 >>> for i in range(num):
...     for j in range((num - i) - 1):
...         print(end=" ")
...     for j in range((i + 1)):
...         print("*", end=" ")
...     print()
...
...     *
...     * *
...     * * *
...     * * * *
...     * * * * *
```

引き渡す記号が変わっただけなのに打ち直している。

```
3 >>> num = int(input("Enter the range:¥t"))
Enter the range:      5
4 >>> for i in range(num):
...     for j in range((num - i) - 1):
...         print(end=" ")
...     for j in range((i + 1)):
...         print("+", end=" ")
...     print()
...
...     +
...     + +
...     + + +
...     + + + +
...     + + + + +
```

💡 一連の命令を事前にまとめて作成しておき、必要に応じてまとめておいた命令を呼び出せるようにすれば良い。まとめられた一連の命令は**関数**と呼ばれる。

関数の効用

対話型インタプリタで関数を定義して、それを呼び出す例。

```
1 >>> def draw_triangle(symbol, num):
...     for i in range(num):
...         for j in range((num - i) - 1):
...             print(end=" ")
...         for j in range((i + 1)):
...             print(symbol, end=" ")
...         print()
... 
```

```
2 >>> draw_triangle('*', 5)
```

```
*
* *
* * *
* * * *
* * * * *
```

```
3 >>> draw_triangle('+', 5)
```

```
+
+ +
+ + +
+ + + +
+ + + + +
```

```
4 >>>
```

引数を使って値を動的に渡せばルーチン内で値を固定する必要はない。

引き渡す値 → 関数 → 結果

テキストエディタで関数を定義して、それを呼び出す例。

```
1 def draw_triangle(symbol, num):
2     for i in range(num):
3         for j in range((num - i) - 1):
4             print(end=" ")
5         for j in range((i + 1)):
6             print(symbol, end=" ")
7         print()
8 
```

```
9 draw_triangle('*', 5)
```

```
10 draw_triangle('+', 5)
```

```
11 #draw_triangle('-', 5)
```

```
12 #draw_triangle('/', 5)
```

予定が変わり引き渡す値が変更しても対応できる。

```
      *
     * *
    * * *
   * * * *
  * * * * *
   +
  + +
 + + +
+ + + +
+ + + + +
```

Okay. There's no waste with this!



関数の特徴

- プログラムの中で部分的にまとめられた命令群。別名：サブルーチン。
- プログラムを実行する際に、初めに実行する命令群はメインルーチン。
- メインルーチンからサブルーチンとして**命令を分離**し、必要に応じて**あらかじめ準備しておくことを関数の定義**という。
- メインルーチンとサブルーチンを関連づけて、実際にサブルーチンの関数を動かすことを**関数の呼び出し**という。
- サブルーチンで処理する値のことを**引数**と呼び、処理の結果を示すためにメインルーチンに返す値を**戻り値**という。
- 書式は右記。ブロックを表すインデント必須。
- 引数および戻り値の記入は必要に応じて省略可能。
- **return文**も省略可能。その場合に戻り値は特殊な型 `None` として返る。
- `function_name`（関数名）は変数名と同じ命名規則。関数名の先頭に、英字、数字、`_` 以外は使えない。

書式

```
def function_name(引数):  
    命令  
    return 戻り値  
function_name()
```

これまでに扱ってきた関数（メソッドは除いた組み込み関数）

- print関数 (初出：第1回)
- bool関数 bin関数 oct関数 hex関数 chr関数 ord関数
int関数 float関数 (初出：第3回の授業)
- str関数 len関数 (初出：第4回の授業)
- input関数 (初出：第5回の授業)
- range関数 (初出：第7回の授業)
- dir関数 type関数 id関数 tuple関数 list関数 dict関数
set関数 (初出：第8回の授業)

関数の定義と呼び出し

```
1 >>> def function():
...     print('Hello, function!')
...
2 >>> function()
Hello, function!
3 >>> function
<function function at 0x102703740>
4 >>> value = function()
Hello, function!
5 >>> print(value)
None
6 >>> def function():
...     print('Hello, function!')
...     return None
...
7 >>> value = function()
Hello, function!
8 >>> print(value)
None
9 >>> def do_nothing():
...     pass
...     return
...
10 >>> do_nothing()
```

functionは予約語ではない。

戻り値を確認。

Noneという特別な値。

戻り値を再度確認。

何も起こらない。戻り値はある。

```
11 >>> def do_nothing():
...     return
...
12 >>> do_nothing()
13 >>> def do_nothing():
...     pass
...
14 >>> print(do_nothing())
None
15 >>> def do_nothing():
...
(略)
IndentationError: expected an indented
block after function definition on line 1
16 >>> def do_nothing:
...     File "<stdin>", line 1
...     def do_nothing:
...         ^
SyntaxError: expected '('
17 >>> def greet():
...     return 'Hi'
...
18 >>> greet()
'Hi'
```

何も起こらない。戻り値はある。

None以外の戻り値が必要なければreturnは省略できる。

print関数で戻り値を直接表示することもできる。

None以外の値を戻り値としてreturnする例。





None

- NoneはPythonにおける特殊な値。
- 何も言うことがないときに使われる。
- Noneはブール値として評価すると偽。
- ブール値のFalseと同じ値ではない。

```
19 >>> type(None)
<class 'NoneType'>
20 >>> dir(None)
['__bool__', '__class__', '__delattr__',
 '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__',
 '__getattr__', '__getstate__',
 '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__',
 '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
21 >>> bool(None)
False
22 >>> None == False
False
```

```
23 >>> whatis(None)
None is None
24 >>> whatis(True); whatis(False)
True is True
False is False
25 >>> whatis(0); whatis(0.0)
0 is False
0.0 is False
26 >>> whatis(''); whatis(""); whatis("''''''")
is False
is False
is False
27 >>> whatis(()); whatis([]); whatis({})
() is False
[] is False
{} is False
28 >>> whatis(set())
set() is False
29 >>> whatis([0])
[0] is True
30 >>> whatis([''])
[''] is True
31 >>> whatis(' ')
is True
```

```
1 >>> def whatis(thing):
...     if thing is None:
...         print(thing, "is None")
...     elif thing:
...         print(thing, "is True")
...     else:
...         print(thing, "is False")
... 
```

	Non-zero value	0
		
	null	undefined
		

関数の定義と呼び出し

```
32 >>> def agree():
...     return True
...
33 >>> agree()
True
34 >>> if agree():
...     print('Agree!')
... else:
...     print('Disagree!')
...
Agree!
```

関数も式の一つであることがわかる。

```
35 >>> def echo(par):
...     return par
...
36 >>> echo('Hi')
'Hi'
37 >>> echo(True)
True
38 >>> def echo(p1, p2):
...     return p1, p2
...
39 >>> echo('Hi!', 'Taro'); echo('Howdy?', 'Hanako')
('Hi!', 'Taro')
('Howdy?', 'Hanako')
```

これより引数の練習。

定義の段階の引数はまだ実行していないので**仮引数**と呼ぶ。

呼び出しの段階の引数は実行に使う値であるので**実引数**と呼ぶ。

echo関数は実引数の True とともに呼び出されている。この値は関数定義の仮引数parにコピーされる。

```
40 >>> echo('Hanako')
(略)
TypeError: echo() missing 1 required
positional argument: 'p2'
41 >>> def echo(p1):
...     return p1, p2
...
42 >>> echo('Howdy?')
(略)
NameError: name 'p2' is not defined. Did
you mean: 'p1'?
```

この命令は組み込みのsum関数。

```
43 >>> sum([1, 2, 3])
6
44 >>> def sum(p1, p2, p3):
...     return p1 + p2 + p3
...
45 >>> sum(1, 2, 3)
6
46 >>> sum([1, 2, 3])
(略)
TypeError: sum() missing 2 required
positional arguments: 'p2' and 'p3'
47 >>> del sum; sum([1, 2, 3])
6
```

ユーザが定義したsum関数。変数sumがさすオブジェクトが変更されてしまうが、命名規則には違反していないのでエラーにはならない。

この命令はユーザ定義のsum関数。

この命令はユーザ定義のsum関数。リストの足し算には対応していないのでエラーが発生。

del文で変数を削除すれば元通りになる。

docstring

PEP20でもThe Zen of Pythonが確認可能。
<https://peps.python.org/pep-0020/>

- The Zen of Pythonでは読みやすさの重要性を唱えている。
- 関数定義の先頭に三連引用符の文字列リテラルを組み込むとドキュメントを添付が可能となる。
- このドキュメントを**docstring**と呼ぶ。

```
1 >>> def echo(anything):  
...     """echo returns the given  
argument."""  
...     return anything  
... 
```

```
2 >>> help(echo)
```

```
Help on function echo in module __main__:
```

```
echo(anything)  
    echo returns the given argument.  
(END)
```

キーボードの`q`を押すと終了する。

```
3 >>> print(echo.__doc__)  
echo returns the given argument.
```

4

```
>>> import this  
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the  
rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to  
guess.  
There should be one-- and preferably only one --  
obvious way to do it.  
Although that way may not be obvious at first unless  
you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad  
idea.  
If the implementation is easy to explain, it may be a  
good idea.  
Namespaces are one honking great idea -- let's do  
more of those!
```

スタック：人間の会話

- ・ 取り留めのない会話をしているときの人間のスタックを考える。
- ・ スタックの最上位が現在の話題を表す。

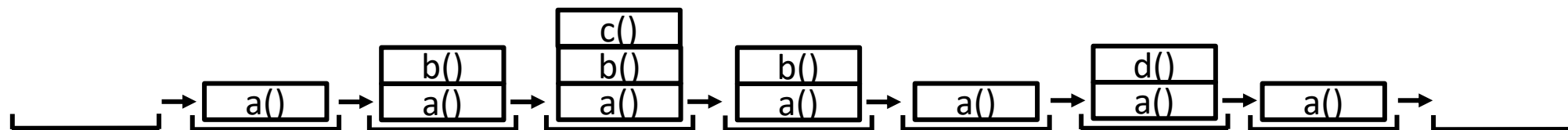


1. 話題開始。
2. Amiの話を思い起こし、Amiの話をしている。
3. Bakuの話を思い起こし、Bakuの話をしている。
4. Chikaの話を思い起こし、Chikaの話をしている。
5. Chikaの話を終えて、Bakuの話に戻り、Bakuの話をしている。
6. Bakuの話を終えて、Amiの話に戻り、Amiの話をしている。
7. Daikiのことを思い起こし、Daikiの話をしている。
8. Daikiの話を終えて、Amiの話に戻り、Amiの話をしている。
9. 話題終了。

得てして人間同士の会話ではこのようにならないことも往々にあるがそれは気にしない。

コンピュータにおけるスタックの状態遷移

- 関数を実行しているときのコンピュータのスタックを考える。
- スタックの最上位が実行している関数を表す。



1. プログラム起動。
2. 関数aを呼び出し、関数aを実行中。
3. 関数aを実行中に関数bを呼び出し、関数bを実行中。
4. 関数bを実行中に関数cを呼び出し、関数cを実行中。
5. 関数cの実行を終えて関数bに戻り、関数bを実行中。
6. 関数bの実行を終えて関数aに戻り、関数aを実行中。
7. 関数aの実行中に関数dを呼び出し、関数dを実行中。
8. 関数dの実行を終えて関数aに戻り、関数aを実行中。
9. プログラム終了。

コンピュータが関数の呼び出し元を記憶しており、**関数の実行が終わると元の場所に必ず戻ってくる。**

スタックの状態遷移をプログラムで確認

- コールスタック：呼び出し元の記憶場所。変数ではなく、システム側で処理される。
- フレームオブジェクト：呼び出し元の行番号や呼び出し先の変数情報などが格納されているオブジェクト。呼び出しがあるとフレームオブジェクトがコールスタックの最上位に積まれる。Pythonはそれを使用して呼び出し元に戻る。その際に使用されたフレームオブジェクトは削除されるのでなくなる。

```
1 >>> def a():  
...     print('a() starts.')...     b()  
...     d()  
...     print('a() ends.')...  
2 >>> def b():  
...     print('b() starts.')...     c()  
...     print('b() ends.')...  
3 >>> def c():  
...     print('c() starts.')...     print('c() ends.')...  
4 >>> def d():  
...     print('d() starts.')...     print('d() ends.')...
```

```
>>> a()  
a() starts.  
b() starts.  
c() starts.  
c() ends.  
b() ends.  
d() starts.  
d() ends.  
a() ends.
```

- スタックに関する知識はプログラムの書き方を覚えるという点では必須ではない。
- 実行対象は関数を実行し終えた後に、その関数を呼び出した元へ戻ることを知っておけば良い。
- しかし、**コールスタックやフレームオブジェクトのことを知っておく**と、次に説明する**グローバルスコープとローカルスコープ**について理解しやすくなる。

グローバルスコープとローカルスコープ

```
1 >>> def hi():
...     name = 'Ami'
...
2 >>> hi()
3 >>> print(name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
4 >>> def hi():
...     name = 'Ami'
...     bye()
...     print(name)
...
5 >>> def bye():
...     name = 'Chika'
...
6 >>> hi()
Ami
7 >>> def hi():
...     print(name)
...
8 >>> name = 'Ami'
9 >>> hi()
Ami
```

ローカルスコープはもう消滅している。
よって、ローカル変数も既に無い。

ローカルスコープはグローバルスコープから使えない。

複数のローカルスコープが同時に存在。

ローカルスコープは他のローカルスコープの変数を使えない。ある関数のローカル変数は、他の関数のローカル変数とは例え名前が同じでも全く別物ということ。

グローバル変数はローカルスコープから読むことができる。

```
10 >>> print(name)
Ami
11 >>> def hi():
...     name = 'Ami in local hi()'
...     print(name)
...
12 >>> def bye():
...     name = 'Ami in local bye()'
...     print(name)
...     hi()
...     print(name)
...
13 >>> name = 'Ami in global'
14 >>> bye()
Ami in local bye()
Ami in local hi()
Ami in local bye()
15 >>> print(name)
Ami in global
```

3つの独立した変数に同じ名前がついている。技術的には全く問題ないが、即座にはどれがどのスコープの変数かが分かりづらくなるので異なるスコープに同じ名前を付けるのは推奨されていない。

- 変数をグローバルかローカルかに分割する理由は、関数の中で変数を更新してもプログラムの他の部分に影響を与えないようにするため。この工夫により、もしバグが発生した際にその原因箇所を特定しやすくなる。
- もしプログラムがグローバル変数しか持っておらず、しかも何百、何千行もあるソースコードであったら、確認する箇所が膨大に増えてしまう。
- もしローカル変数の値がおかしい場合、その関数をつつ調べるだけで済む。

グローバル文

```
16 >>> def hi():
...     global name
...     name = 'Ami'
...
17 >>> name = 'Chika'
18 >>> hi()
19 >>> print(name)
```

関数の中からグローバル変数を変更したい時、global文を使う。

この関数の中で、global文を用いた変数名のローカル変数を作らないとの意思表示にもなる。

```
Ami
-----
20 >>> def hi():
...     global name
...     name = 'Ami'
...
21 >>> def bye():
...     name = 'Baku'
...
22 >>> def howdy():
...     print(name)
...
23 >>> name = 'Chika'
24 >>> hi()
25 >>> print(name)
Ami
```

グローバル変数かローカル変数かもう一度確認。

グローバル変数。

ローカル変数。

グローバル変数。代入文がないから。

```
26 >>> def hi():
...     print(name)
...     name = 'Ami in local'
...
27 >>> name = 'Ami in global'
28 >>> hi()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in hi
UnboundLocalError: cannot access local
variable 'name' where it is not associated
with a value
```

グローバル変数かローカル変数か分からなくなる状況を作ると...

- 変数は、常に**グローバル**か**ローカル**かの、どちらかのスコープにある。
- 基本はスタックにフレームオブジェクトがあるか無いかで変数のスコープがグローバルかローカルであるかの判断が行える。
- global文が関数で使用されている場合はその限りではないので例外的に考えれば良い。

enumerate関数

- enumerate関数は、イテラブルなオブジェクトの各要素とそのインデックスをペアで提供する。
- 以下のソースコードは、果物リスト内の各果物の名前とそれに対応するインデックスを表示する例。

```
1 fruits = ["apple", "banana", "cherry"]  
2 for index, fruit in enumerate(fruits):  
3     print(f"Index {index}: {fruit}")
```

enumerate.py

実行結果の例

```
Index 0: apple  
Index 1: banana  
Index 2: cherry
```

```
1 fruits = ["apple", "banana", "cherry"]  
2 for index, fruit in enumerate(fruits, start=1):  
3     print(f"Index {index}: {fruit}")
```

enumerate1.py

実行結果の例

```
Index 1: apple  
Index 2: banana  
Index 3: cherry
```

filter関数

- filter関数は、与えられた条件に合致する要素だけをイテラブルなオブジェクトから選択する。
- 以下のソースコードは、リストから偶数のみを抽出し、その要素からなる新しいリストとして保存し直して表示する例。

```
1 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 def even_num(n):
3     if n % 2 == 0:
4         return True
5     else:
6         return False
7
8 filtered_num = []
9 for number in filter(even_num, numbers):
10     filtered_num.append(number)
11 print(filtered_num)
```

filter.py

実行結果の例

[2, 4, 6, 8, 10]

map関数

- map関数は、与えられた関数をイテラブルなオブジェクトの各要素に適用し、変換された要素からなる新しいイテラブルなオブジェクトを生成する。
- 以下のソースコードは、各数値を自乗して新しいタプルを作成する例。
- map関数が返す戻り値だけでもイテラブルなオブジェクトであるが、そのままでは表示できるものではないので、中身を見る場合は例えば基本のデータ型に変換する必要がある。例 tuple関数の併用。
- そのルールはenumerate関数とfilter関数も同様。

```
1 def square(x):  
2     return x**2  
3  
4 numbers = (1, 2, 3, 4, 5)  
5  
6 squared_numbers = tuple(map(square, numbers))  
7 print(squared_numbers)
```

実行結果の例

(1, 4, 9, 16, 25)

sorted関数

- sorted関数は、イテラブルなオブジェクトの各要素を指定した条件に応じて並べ替える。

```
1 fruits = ["cherry", "apple", "banana"]  
2 sorted_fruits = sorted(fruits)  
3 print(sorted_fruits)
```

sorted.py

実行結果の例

```
['apple', 'banana', 'cherry']
```

```
1 fruits = ["cherry", "apple", "banana"]  
2 sorted_fruits = sorted(fruits, reverse=True)  
3 print(sorted_fruits)
```

SortedR.py

実行結果の例

```
['cherry', 'banana', 'apple']
```

```
1 fruits = ["cherry", "apple", "banana"]  
2 sorted_fruits = sorted(fruits, key=len)  
3 print(sorted_fruits)
```

sortedL.py

実行結果の例

```
['apple', 'cherry', 'banana']
```

```
1 numbers = [-10, 5, -3, 8, -1]  
2 sorted_numbers = sorted(numbers, key=abs)  
3 print(sorted_numbers)
```

sortedA.py

実行結果の例

```
[-1, -3, 5, 8, -10]
```

zip関数

- zip関数は、複数のイテラブルなオブジェクトをペアにして新しいイテラブルなオブジェクトを作成する。
- 以下のソースコードは、名前とスコアの要素をペアにして、新しいイテラブルなオブジェクトを作成する例。
- zip関数が返す戻り値だけでもイテラブルなオブジェクトであるが、そのままでは表示できるものではないので、中身を見る場合は例えば基本のデータ型に変換する必要がある。例 list関数やdict関数の併用。

```
1 names = ["Alice", "Bob", "Charlie"]
2 scores = [85, 92, 78]
3 zipped_data = list(zip(names, scores))
4 print(zipped_data)
```

zip.py

実行結果の例

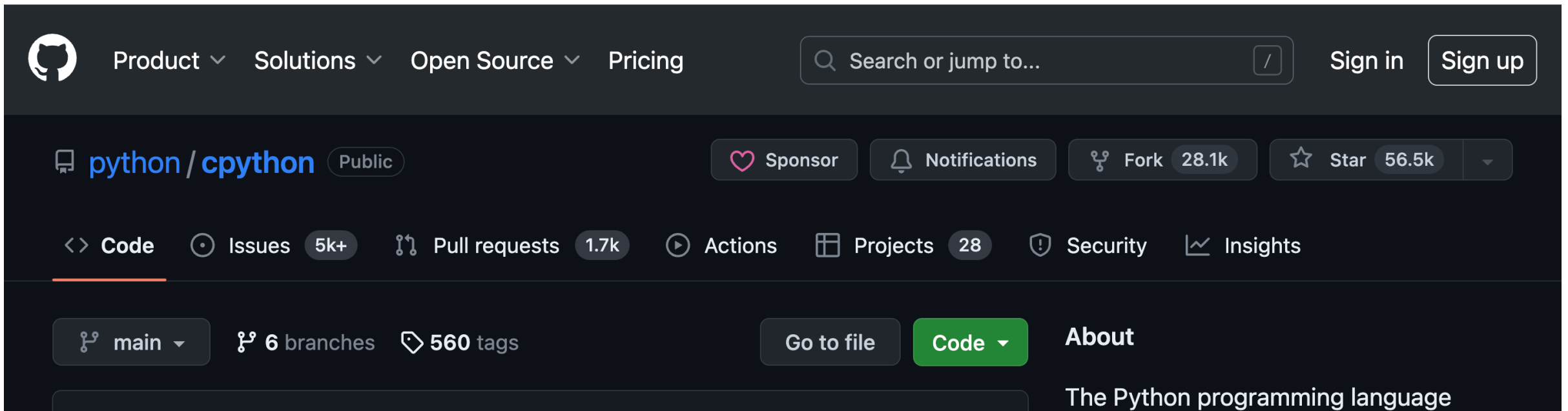
```
[('Alice', 85), ('Bob', 92), ('Charlie', 78)]
```

```
1 names = ["Alice", "Bob", "Charlie"]
2 scores = [85, 92, 78]
3 zipped_data = dict(zip(names, scores))
4 print(zipped_data)
```

zipDict.py

実行結果の例

```
{'Alice': 85, 'Bob': 92, 'Charlie': 78}
```



- 例えば「bltinmodule.c」に組み込み関数に関するソースコードがあり。
- このように、興味があればソースコードを確認して中身から理解することが可能。
- しかし、単に関数を使うだけなら、細かい仕掛けを知る必然性はない。
- 関数については、その入力（引数）と出力（戻り値）を知っていれば基本的には問題がない。

Python » Japanese » 3.12.0 » 3.12.0 Documentation »

Theme Auto | クイック検索 | 検索 | モジュール | 索引

ダウンロード

これらのドキュメントのダウンロード

バージョンごとのドキュメント

Python 3.13 (in development)
Python 3.12 (stable)
Python 3.11 (stable)
Python 3.10 (security-fixes)
Python 3.9 (security-fixes)
Python 3.8 (security-fixes)
Python 3.7 (EOL)
Python 3.6 (EOL)
Python 3.5 (EOL)
Python 3.4 (EOL)
Python 3.3 (EOL)
Python 3.2 (EOL)
Python 3.1 (EOL)
Python 3.0 (EOL)
Python 2.7 (EOL)
Python 2.6 (EOL)
全てのバージョン

その他のリソース

PEP 一覧
初心者向けガイド
本の一覧
発表の音声 / 映像
Python開発者ガイド

Python 3.12.0 ドキュメント

ようこそ! Python 3.12.0 公式ドキュメントへ。

ドキュメント一覧

What's new in Python 3.12?

あるいは2.0からのすべての "What's new" ドキュメント

チュートリアル

ここから始めましょう

ライブラリーリファレンス

枕の下に置きましょう

言語リファレンス

構文と言語要素の解説

Python のセットアップと利用

各プラットフォームでの Python の使い方

Python HOWTO

特定のトピックに関する、より深いドキュメント

Python モジュールのインストール

Python Package Index などからのインストール

Python モジュールの配布

他人がインストールできるようにモジュールを配布する

拡張と埋め込み

C/C++ プログラマ向けチュートリアル

Python/C API

C/C++ プログラマ向けリファレンス

FAQ

よくある質問 (解答つき!)

索引と目次

全モジュール索引

すべてのモジュールに素早くアクセス

総索引

全ての関数、クラス、用語

用語集

重要な用語の説明

メタ情報

バグを報告する

ドキュメントへの貢献

このドキュメントについて

検索

このドキュメントから検索する

全体の目次

全てのセクションとサブセクションの一覧

Python の歴史とライセンス

Copyright

Download the documentation

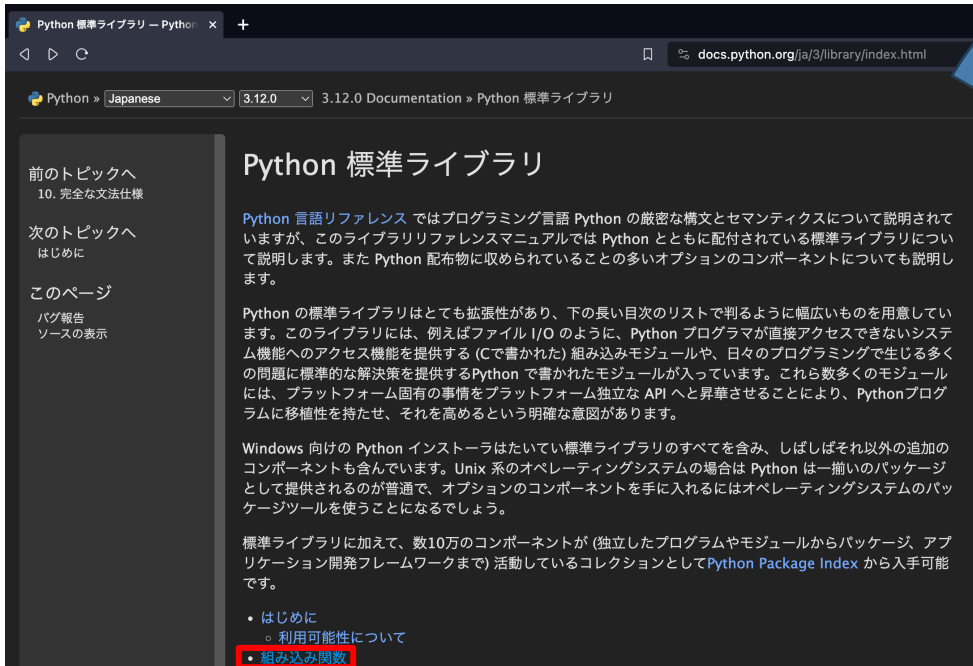
Python » Japanese » 3.12.0 » 3.12.0 Documentation »

Theme Auto | クイック検索 | 検索 | モジュール | 索引

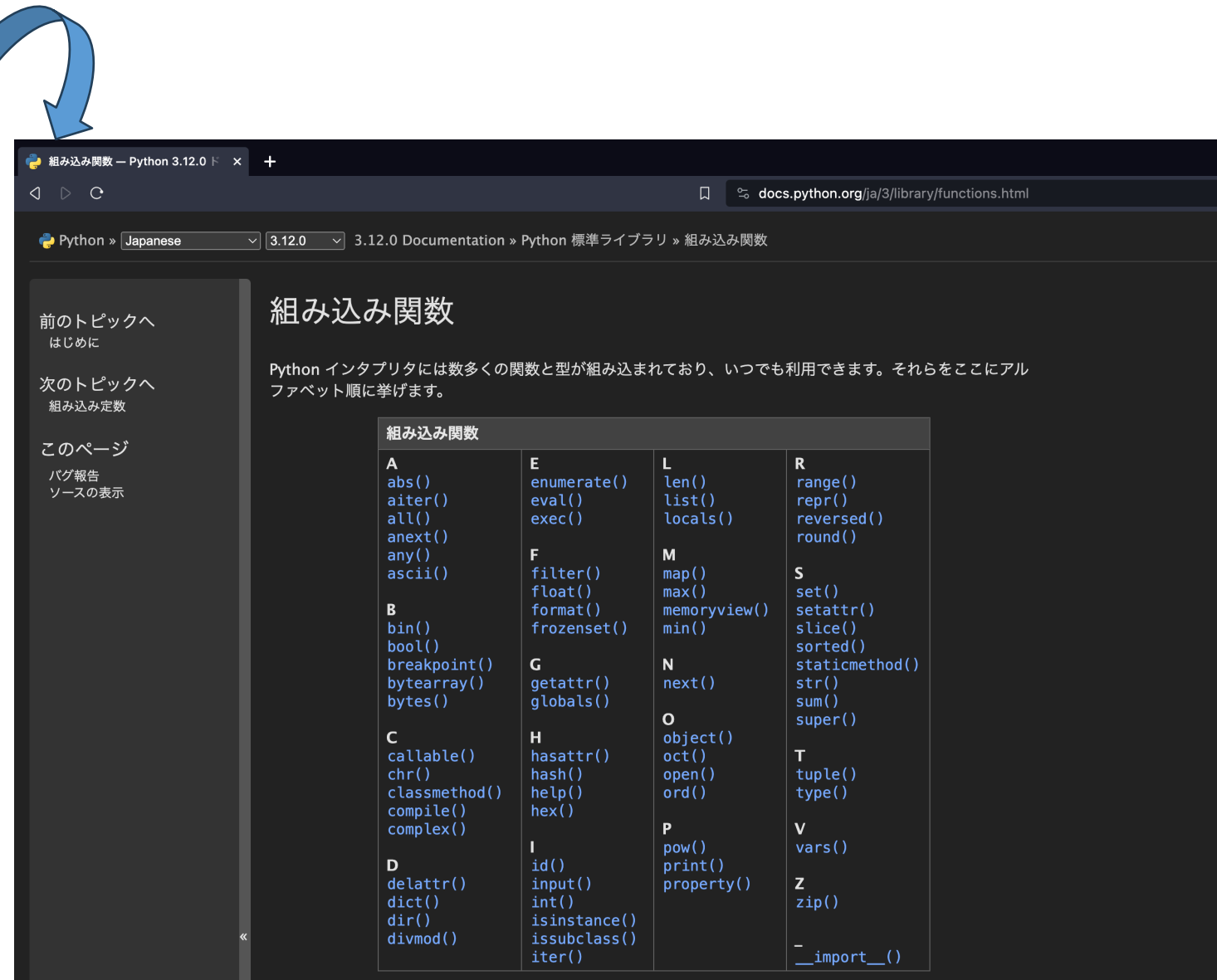
© Copyright 2001–2023, Python Software Foundation.
This page is licensed under the Python Software Foundation License Version 2.
Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.
See [History and License](#) for more information.

23

Python ドキュメント（組み込み関数）



<https://docs.python.org/ja/3/library/index.html>



<https://docs.python.org/ja/3/library/functions.html>

まとめ

- 関数を用いると**プログラムの冗長性が減り、効率的な作業**となる。
- 関数は**プログラムの中で部分的にまとまった命令群**。別名サブルーチン。
- プログラムを実行して初めに処理される命令群はメインルーチンという。
- メインルーチンから関数が見えるようにすることを**関数の定義**という。
- メインルーチンから関数を実行することを**関数の呼び出し**という。
- 関数に入力する値のことを**引数**と呼び、関数での処理の結果を示すためにメインルーチンに返す値を**戻り値**と呼ぶ。
- 関数を定義する際は**docstring**を付加すると関数の中身が分かりやすくなる。**The Zen of Python**の精神にも沿う。
- 変数には、**グローバルかローカルかのスコープ**があるので使い分ける。
- **global文**を用いると関数からグローバル変数を更新することができる。
- **グローバル変数ばかりに頼るプログラムは良くない。**