プログラミング初級（Python）

タプル、リスト、辞書、集合

早稲田大学グローバルエデュケーションセンター

値の分子

進む速さは問題ではない。
重要なのは途中で止まらないことだ。

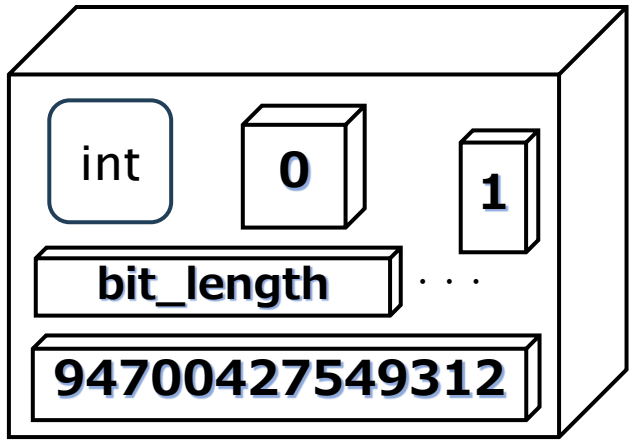
— 孔子

値の分子（比喩）

- これまでに値の種別として、ブール値、整数、浮動小数点数、文字列を紹介。
- これらの値を組み合わせて、**値の塊**を作ることができる。
- 処理内容が同じ値は散りばめておくより塊にしておいた方が命令の数を減らせたり、間違えて別の処理を実行する恐れが無くなったりする。
- 要するにプログラムが効率的なものとなり命令や値の管理が楽になる。
- Pythonでは、**値の塊の種別**として主に**タプル**、**リスト**、**辞書**、**集合**がある。
- ブール値、整数、浮動小数点数、文字列を物質でいう原子に例えると、タプル、リスト、辞書、集合は物質の分子（値の分子）に例えられる。
- 要するに、タプル、リスト、辞書、集合はブール値、整数、浮動小数点数、文字列を**寄せ集めただけの値**ともいえる。
- （復習）**Pythonではデータが全てオブジェクトとして構成**されている。
- タプル、リスト、辞書、集合もオブジェクトの一種。
- オブジェクトには様々な要素があった。**要素として関数を有する**ことも可能。

オブジェクトとは？

- Pythonにおいてオブジェクトとは、**型**、**値**、参照カウント、ID、**関数**(など)のチャンク。



- 異なるデータ型では使うことができない。
- むしろ他では使う必要がない関数であるから特定のオブジェクトで定義する設計。
- これがオブジェクト指向。想定外のデータ型で関数を使ってしまうミスを減らせる。

書式 `変数.関数名(引数)`

- オブジェクトにある関数をメソッドと呼ぶ。
- 異なるデータ型（グローバル）で使える、もしくは、使う必要がある関数は**組み込み関数**と呼ばれ、オブジェクト名を書く必要がない。例 print関数、input関数、len関数

- 値が何を意味するかを定義する型：`int`
- 型に従って解釈できる値：`0`
- **オブジェクトの持つ専用の関数**：`bit_length`
- 当該オブジェクトが何回使われているかを示す参照カウント：`1`
- 他のオブジェクトとの区別に使われるID：`94700427549312`

メソッドの実行

メソッドの内容を暗記するよりもオブジェクトやメソッドの本質を理解することが大事。

```
1  >>> number = 15
2  >>> number.bit_length()
   4
3  >>> number = 3.14
4  >>> number.is_integer()
   False
5  >>> email = "   hoge@example.com   "
6  >>> email
   '   hoge@example.com   '
7  >>> email.strip()
   'hoge@example.com'
8  >>> email.rstrip()
   '   hoge@example.com'
9  >>> email.lstrip()
   'hoge@example.com   '
10 >>> email.strip("!")
   '   hoge@example.com   '
11 >>> email = "!!!!!hoge@example.com!!!!!"
12 >>> email
   '!!!!!hoge@example.com!!!!!'
13 >>> email.strip("!")
   'hoge@example.com'
14 >>> dir(email)
```

整数型のオブジェクトが持つメソッドの例。

10進数の数値15は2進数で1111 なのでビット長が4となる。

文字列型のオブジェクトが持つメソッドの例。

オブジェクトが持つメソッドを調べる方法。

```
['__add__', '__class__', '__contains__',
'__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__',
'__getnewargs__', '__getstate__', '__gt__',
'__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition',
'removeprefix', 'removesuffix', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

タプルの操作

基本書式 `name = (value1, value2, …)` `name = ()`

```
>>> empty_tuple = ()
>>> empty_tuple
()
>>> one_tuple = ('Python',)
>>> one_tuple
('Python',)
>>> one_tuple = 'Python',
>>> one_tuple
('Python',)
>>> one_tuple = ('Python')
>>> one_tuple
'Python'
>>> one_tuple = 'Python'
>>> one_tuple
'Python'
>>> lang_tuple = ('Python', 'C', 'Java')
>>> lang_tuple
('Python', 'C', 'Java')
>>> lang_tuple = 'Python', 'C', 'Java'
>>> lang_tuple
('Python', 'C', 'Java')
>>> type(lang_tuple)
<class 'tuple'>
>>> type('Python',)
<class 'str'>
>>> type(('Python',))
<class 'tuple'>
>>> a, b, c = lang_tuple
>>> a
'Python'
>>> b
'C'
>>> c
'Java'
>>> ('Python',) + ('C', 'Java')
('Python', 'C', 'Java')
>>> ('Go!',) * 3
('Go!', 'Go!', 'Go!')
>>> a = (3, 1); b = (3, 1, 4)
>>> a == b
False
>>> a <= b
True
>>> a >= b
False
```

タプルには丸括弧を用いる。

要素がひとつでもカンマが必要。

要素がひとつでカンマをつけないと文字列リテラルになる。

type関数を用いるとデータ型が分かる。

関数に入れる場合は引数のカンマと区別するためにカッコが必要。

タプルがあるなら要素を一度に代入可能。

タプルの操作

```
>>> words = ('UNIX', 'BSD', 'Linux')
>>> for word in words:
...     print(word)
...
UNIX
BSD
Linux
>>> t1 = ('hoge', 'foo')
>>> t2 = ('bar',)
>>> t1 + t2
('hoge', 'foo', 'bar')
>>> t1 += t2
>>> t1
('hoge', 'foo', 'bar')
>>> t1 = ('hoge', 'foo')
>>> t2 = ('bar',)
>>> id(t1)
2550724736448
>>> t1 += t2
>>> id(t1)
2550724798592
>>> scores = (100, 80, 80)
>>> sum(scores)
250
```

イテラブルなシークエンスなのでin演算子と併用可能。結果、for文でも活用可能。

このt1が指すオブジェクトは次のt1が指すオブジェクトとは異なる。

このオブジェクトはt1+t2が指す新しいオブジェクト。IDは異なる。

```
>>> len(scores)
3
>>> scores.count(100)
1
>>> hoge = 'foo'; tuple(hoge)
('f', 'o', 'o')
>>> words[1]
'BSD'
>>> words[1] = 'FreeBSD'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
>>> 'Free' + words[1]
'FreeBSD'
>>> words_unix = ('UNIX', 'Free' + words[1],
'Linux')
>>> words_unix
('UNIX', 'FreeBSD', 'Linux')
>>> words[0:2:]
('UNIX', 'BSD')
>>> words[::-1]
('Linux', 'BSD', 'UNIX')
```

タプルのメソッドの例。

タプルは**イミュータブル**なので値が書き換えられない。

タプルは、焼き固めるかのように値の書き換えができないことを保証する。誤って書き換える危険がないため、管理上の利点となることもある。リストよりも消費スペースが小さい。

読み込みは可能なので、新しいオブジェクトを作ればタプルの値を使うことはできる。

文字列同様、タプルにもスライスが適用できる。

リストの操作

基本書式 `name = [value1, value2, …]` `name = []`

```
>>> empty_list = []
>>> empty_list
[]
>>> one_list = ['Python']
>>> one_list
['Python']
>>> one_list = ['Python',]
>>> one_list
['Python']
>>> another_empty_list = list()
>>> another_empty_list
[]
>>> list('Linux')
['L', 'i', 'n', 'u', 'x']
>>> t_linux = ('Debian', 'RHEL',
'Slackware')
>>> list(t_linux)
['Debian', 'RHEL', 'Slackware']
>>> linux = list(t_linux); linux
['Debian', 'RHEL', 'Slackware']
>>> linux[0]
'Debian'
>>> linux[-1]
'Slackware'
```

リストには角括弧を用いる。

リストに単一の要素を作るときカンマは必須でない。

list関数はシーケンスの塊ごとに要素を作成する。

これよりオフセットによる要素の取り出し。

```
>>> linux[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> linux[::2]
['Debian', 'Slackware']
>>> linux[::-1]
['Slackware', 'RHEL', 'Debian']
>>> linux[4:]
[]
>>> others = ['CentOS', 'Ubuntu']
>>> linux + others
['Debian', 'RHEL', 'Slackware', 'CentOS',
'Ubuntu']
>>> linux * 2
['Debian', 'RHEL', 'Slackware', 'Debian',
'RHEL', 'Slackware', 'Debian', 'RHEL',
'Slackware', 'Debian', 'RHEL', 'Slackware']
>>> linux * others
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-
int of type 'list'
```

これよりスライスによる要素の取り出し。

リストが逆順になるスライス。

文字列同様、スライスは無効なインデックスを指定しても例外が発生しない。

リストの操作

```
>>> linux = list(t_linux); linux
['Debian', 'RHEL', 'Slackware']
>>> linux.append('Puppy')
>>> linux
['Debian', 'RHEL', 'Slackware', 'Puppy']
>>> linux.append(others)
>>> linux
['Debian', 'RHEL', 'Slackware', 'Puppy',
['CentOS', 'Ubuntu']]
>>> linux = list(t_linux); linux
['Debian', 'RHEL', 'Slackware']
>>> linux[2] = 'Puppy'
>>> linux
['Debian', 'RHEL', 'Puppy']
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [8, 9]
>>> numbers
[1, 8, 9, 4]
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = []
>>> numbers
[1, 4]
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = (98, 99, 100)
>>> numbers
[1, 98, 99, 100, 4]
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = 'Python'
>>> numbers
[1, 'P', 'y', 't', 'h', 'o', 'n', 4]
>>> linux
['Debian', 'RHEL', 'Puppy']
>>> del linux[1]
>>> linux
['Debian', 'Puppy']
>>> linux.insert(1, 'Puppy')
>>> linux
['Debian', 'Puppy', 'Puppy']
>>> linux.remove('Puppy')
>>> linux
['Debian', 'Puppy']
>>> linux.pop()
'Puppy'
>>> linux
['Debian']
>>> linux.extend(['Slackware', 'Puppy'])
>>> linux
['Debian', 'RHEL', 'Slackware', 'Puppy']
```

これよりリストの書き換え操作。

delは文でありリストのメソッドではない。
戻り値はない。

最初に見つかった要素を削除する。

デフォルトのオフセットは-1。

リストの操作

```
54 >>> linux.index('Debian')
   0
55 >>> linux.append('Debian')
56 >>> linux
   ['Debian', 'RHEL', 'Slackware', 'Debian']
57 >>> linux.index('Debian')
   0
58 >>> 'Debian' in linux; 'Puppy' in linux
   True
   False
59 >>> linux.count('Debian'); linux.count('Puppy')
   2
   0
60 >>> ', '.join(linux)
   'Debian, RHEL, Slackware, Debian'
61 >>> joined = ', '.join(linux)
62 >>> joined.split(',')
   ['Debian', ' RHEL', ' Slackware', '
   Debian']
63 >>> len(linux)
   4
```

これよりリストに対する他の処理。

joinはあくまで文字列のメソッド。

splitも文字列のメソッド。joinメソッドで得た文字列をsplitメソッドで元のリストに戻している。つまり、これらは逆の関係にある。

```
64 >>> a = [3, 1]; b = [3, 1, 4]
65 >>> a == b
   False
66 >>> a <= b
   True
67 >>> a > b
   False
68 >>> linux.clear()
   []
```

辞書の操作

基本書式 `name = {key1: value1, key2: value2, …}` `name = {}`

```
>>> empty_dict = {}
>>> empty_dict
{}
>>> languages = {
...     'Python': 'disc_py',
...     'C': 'disc_c',
...     'Java': 'disc_java',
... }
>>> languages
{'Python': 'disc_py', 'C': 'disc_c',
'Java': 'disc_java'}
>>> languages = dict(Python='disc_py',
C='disc_c', Java='disc_java')
>>> languages
{'Python': 'disc_py', 'C': 'disc_c',
'Java': 'disc_java'}
>>> two_seq = ['Python', 'disc_py'], ['C',
'disc_c'], ['Java', 'disc_java']
>>> dict(two_seq)
{'Python': 'disc_py', 'C': 'disc_c',
'Java': 'disc_java'}
>>> languages['Python']
'disc_py'
```

辞書には波括弧を用いる。

オフセットはない。代わりに個々のvalueに**一意**のkeyを与えて管理する。

オフセットはないので、スライスはできない。

```
>>> languages['JavaScript']
(略)
KeyError: 'JavaScript'
>>> 'JavaScript' in languages
False
>>> languages.get('C')
'disc_c'
>>> languages.get('JavaScript', 'No existing')
'No existing'
>>> languages.get('JavaScript')
>>> languages['JavaScript'] = 'disc_js'
>>> print(languages)
{'Python': 'disc_py', 'C': 'disc_c',
'Java': 'disc_js', 'JavaScript': 'disc_js'}
>>> languages['Java'] = 'disc_JAVA'
>>> languages
{'Python': 'disc_py', 'C': 'disc_c',
'Java': 'disc_JAVA', 'JavaScript':
'disc_js'}
>>> others_lang = {'PHP': 'disc_php'}
>>> {**languages, **others_lang}
{'Python': 'disc_py', 'C': 'disc_c',
'Java': 'disc_JAVA', 'JavaScript':
'disc_js', 'PHP': 'disc_php'}
```

これより主に辞書の書き換え操作。

要素の**新規作成**。

あえてprint関数で出力。

要素の**更新**。

辞書の操作

```
>>> del languages['JavaScript']
>>> languages
{'Python': 'disc_py', 'C': 'disc_c',
'Java': 'disc_JAVA'}
>>> len(languages)
3
>>> languages.pop('Java')
'disc_JAVA'
>>> languages
{'Python': 'disc_py', 'C': 'disc_c'}
>>> languages['Java'] = 'disc_JAVA'
>>> for lang in languages.items():
...     print(lang)
...
('Python', 'disc_py')
('C', 'disc_c')
('Java', 'disc_JAVA')
```

辞書に数のインデックスがないためかデフォルトの値はない。

popはオブジェクトから要素を削除するだけでなく、要素を出して知らせてくれる。まさに取り出し。

```
>>> for lang in languages:
...     print(lang)
...
Python
C
Java
>>> for lang in languages.keys():
...     print(lang)
...
Python
C
Java
>>> for lang in languages.values():
...     print(lang)
...
disc_py
disc_c
disc_JAVA
>>> languages.clear()
>>> languages
{}
```

何も指定しないとキーが取り出せる。

明示的に指定してもキーが取り出せる。

バリューのみ取り出す方法もある。

集合の操作

基本書式 `name = {value1, value2, …}` `name = set()`

```
>>> empty_set = set()
>>> empty_set
set()
>>> even_numbers = {0, 2, 4, 6, 8}
>>> even_numbers
{0, 2, 4, 6, 8}
>>> set('letters')
{'t', 'e', 's', 'l', 'r'}
>>> linux = set(['Debian', 'RHEL',
'Slackware', 'Puppy'])
>>> len(linux)
4
>>> linux.add('Vine')
>>> linux
{'Slackware', 'RHEL', 'Vine', 'Debian', 'Puppy'}
>>> linux.remove('RHEL')
>>> linux.remove('Vine'); linux
{'Puppy', 'Debian', 'Slackware'}
>>> for dist in linux:
...     print(dist)
...
Puppy
Slackware
Debian
```

括弧だけの表記で空集合を作れない。関数を用いる。

個々の値は一意で重複した値を持てない。

集合は何かがあるかだけが分かればよく、他のことを知らなくても良い時に使う。

オフセットはないので、スライスはできない。

要素の追加はappendではなく、addメソッドが用意されている。

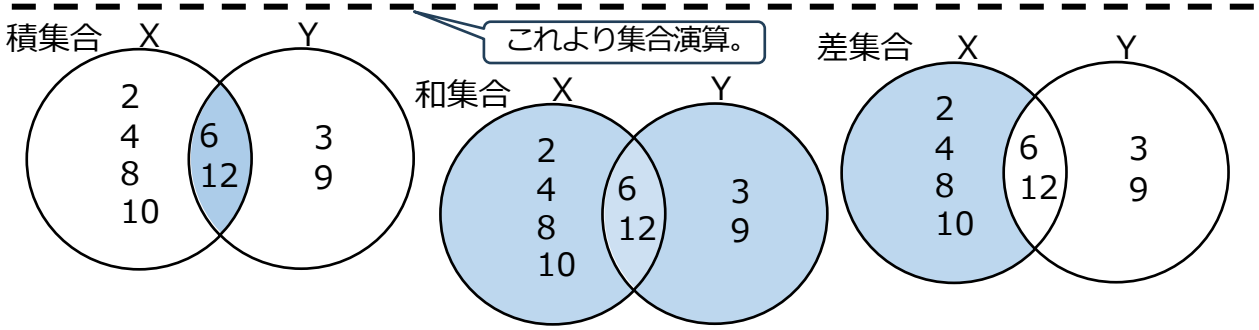
removeメソッドはリストと同様にある。

```
>>> cocktails = {
...     'martini': {'vodka', 'vermouth'},
...     'manhattan': {'rye', 'vermouth'},
...     'screwdriver': {'vodka', 'orange
juice'},
...     'bloody mary': {'vodka', 'tomato
juice'},
...     'gimlet': {'gin', 'lime juice',
'sugar'},
...     'xyz': {'rum', 'curacao', 'lemon
juice'},
... }
>>> for cocktail, ingredients in
cocktails.items():
...     if 'vodka' in ingredients:
...         print(cocktail)
...
martini
screwdriver
bloody mary
```

辞書の中に集合を定義。

集合の操作

```
>>> for cocktail, ingredients in cocktails.items():
...     if 'vodka' in ingredients and not('vermouth' in ingredients or 'tomato juice' in ingredients):
...         print(cocktail)
...
screwdriver
```




```
>>> cocktails['martini'] & {'vermouth'}
{'vermouth'}
>>> for cocktail, ingredients in cocktails.items():
...     if 'vodka' in ingredients and not ingredients & {'vermouth', 'tomato juice'}:
...         print(cocktail)
...
screwdriver
```

```
>>> cocktails['martini'] & cocktails['xyz']
set()
>>> new_cocktail = cocktails['martini'] | cocktails['xyz']
>>> new_cocktail
{'rum', 'lemon juice', 'curacao', 'vermouth', 'vodka'}
>>> new_cocktail.intersection(cocktails['bloody mary'])
{'vodka'}
>>> original_cocktail = new_cocktail - cocktails['bloody mary']
>>> original_cocktail
{'rum', 'vermouth', 'lemon juice', 'curacao'}
>>> cocktails['martini'].union(cocktails['xyz'])
{'rum', 'lemon juice', 'curacao', 'vermouth', 'vodka'}
>>> new_cocktail.difference(cocktails['bloody mary'])
{'rum', 'vermouth', 'lemon juice', 'curacao'}
```

リスト・辞書・集合の内包表記

```
>>> seq = range(1, 6)
>>> list(seq)
[1, 2, 3, 4, 5]
>>> [num for num in range(1,6)]
[1, 2, 3, 4, 5]
>>> [num for num in range(1,6) if num % 2
== 1]
[1, 3, 5]
>>> [num - 1 for num in range(1,6) if num %
2 == 1]
[0, 2, 4]
```

リストの内包表記。

```
>>> word = 'letters'
>>> {letter: word.count(letter) for letter
in word}
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
>>> {letter: word.count(letter) for letter
in set(word)}
{'t': 2, 'l': 1, 'r': 1, 'e': 2, 's': 1}
```

辞書の内包表記。

集合を併用。**How Pythonic!**

```
>>> {num for num in range(1, 6) if num % 3
== 1}
{1, 4}
```

集合の内包表記。

```
>>> (num for num in range(1, 6))
<generator object <genexpr> at
0x000001D60FF17580>
```

タプルの内包表記はない。**内包表記があるのはリスト、辞書、集合**である。

まとめ

- Pythonではデータが全てオブジェクトとして構成されている。
- 基本のデータもしくはその型として、ブール値、整数、浮動小数点数、文字列、**タプル**、**リスト**、**辞書**、**集合**がある。
- オブジェクトには様々な**関数が同梱**されている。それを**メソッド**と呼ぶ。
- タプルは**イミュータブル**。作成後に値は直接的に書き換えられない。
- リストは**ミュータブル**。　作成後に値は直接的に書き換えられる。
- 辞書は、**ミュータブル**。集合は、**ミュータブル**。
- リストまたは辞書の要素を削除するのに使う**del文はメソッドではない**。
- 基本的な集合演算として、**積集合**、**和集合**、**差集合**がある。
- **辞書と集合を併用した簡易的な検索プログラム**を作成できる。その際、集合演算を用いるとプログラムの簡略化につながることもある。
- タプルに内包表記はない。**リスト、辞書、集合には内包表記**がある。

次回予告　**関数**