

第Ⅱ編 コレクションと関数

1 コレクション[リスト・タプル・辞書(連想配列)・集合]

数学の配列(ベクトル)は

$A = [1, 2, 3, 4]$ または $A = (1, 2, 3, 4)$
 $A[0]=1, A[1]=2$ または $A(0)=1, A(1)=2$

であった。

配列に相当するものとして、リスト・タプル・辞書(連想配列)・集合(セット)がある。

説 明	
リスト (List)	任意の値(数・文字列・オブジェクト)の要素を持つ配列である。 要素の追加や削除を行うことが 可能 である。 記号: []
タプル (Tuple)	任意の値(数・文字列・オブジェクト)の要素を持つ配列である。 要素の追加や削除を行うことが 不可能 である。 記号: ()
辞書 (Dict)	任意・唯一(unique)の Key と任意の Value で表わされる配列である。 Keyはユニーク(unique)である。要素の追加や削除を行うことが可能である。 記号: { }
集合 (Set)	任意・ 唯一の値(数・文字列・オブジェクト) の要素を持つ配列である。 要素の追加や削除を行うことが可能である。論理演算ができる。 記号: { }
例 題	
リスト	list00 = ['ab', 'ddefg', 'cdg'] # value
タプル	tuple00 = ('あい', 'かきく', 'やゆ') # value
辞書	dict00 = { 'id': 'a0012', 'name': 'Tom' } # unique key --value
集合	set00 = { '12', '88', 'a33' } # unique value

書き方

(1) リスト(List)

リストは1個の変数に数個の要素を格納することが可能である。

リストに関する説明は、オブジェクトに関して成り立つ場合がある。

そのために、リスト変数（オブジェクト）・文字列(オブジェクト)を **list_vari**、**str00** として例題で用いる。

A リストの宣言・初期化（インスタンス化=オブジェクトの生成）

書式 1	#要素を直接与える
	list_vari = [element1, element2, element3, ...]
	list_vari = [10, "str20", [30, 33]] 要素は3個 要素element1,2,3,..は数値、文字列またはリストなどのオブジェクトである。 注意：各elementは同じ型である必要はない。
書式 2	#文字列からの作成(list関数を使用)=>要素は1文字
	list_vari = list(文字列)
	list_vari = list("abcd5") # list_vari=["a","b","c","d","5"]
書式 3	#タプルからの作成(list関数を使用) タプル：（要素1,要素2,...,要素?）
	list_vari = list(タプル)
	list_vari = list((11,22,33))
	list_vari = list(("aa","bb","cc"))
	list_vari = list((11,22,[33,34],44)) list_vari = list(("11","22",["33","34"],"44"))

ここで説明する要素は文字列変数に対しても成立するものがある。

すなわち、str="abcde"という文字列は、1文字（1Byte=8bits）であるので、1Byteのリストに1文字ずつ入っているようなものである。

a	b	c	d	e
---	---	---	---	---

このように考えると、文字列はリストである。

0番地(1番目)は str[0]="a"、1番地(2番目)は str[1]="b"、...である。

なお、日本語の場合には2Bytesの大きさの配列である。

文字列変数に対して成立するものはその都度示す。

インタラクタモード(このまま続く)

```
>>> list0 = [10, "A", [30,33]]
>>> list0
```



```

C:\> コマンドプロンプト - python
>>> list0 = [10, "A", [30,33]]
>>> list0
[10, 'A', [30, 33]]

```

(A) 長さ

リストの要素数はつぎのように取得できる。

	len(リストオブジェクト)
(長さ)	len(list_vari)
	len(str00)

```
>>> len( list0 )
```

```

コマンドプロンプト - python
>>> len( list0 )
3

```

(B) 要素値の参照

リストの要素の全要素およびある番地の要素はつぎのように参照する。

	list_vari	# 全要素を参照
(参照)	list_vari[1]	# 2番目の要素を参照 (1番目は0である)
	str00[1]	# 2番目の要素を参照 (1番目は0である)

リストの要素のある部分の要素 (複数個の要素) はつぎのように参照する。

	sub_list = リストオブジェクト[スライス]
	list_vari = ["0", "1", "2", "3", "4", "5", "6"]
サブ	sub_list = list_vari[2:5] # ["2", "3", "4"]
リスト	sub_str11 = 文字列オブジェクト[スライス]
	str00 = "0123456789"
	sub_str11 = str00[::2] # "02468"

ただし、リストの要素番地は0番地、1番地…である。つまり、

```
list_vari = [ "a", "b", "c", "d", "e" ]
```

list_vari[0]="a"は0番地 (1番目の値) 、

list_vari[1]="b"は1番地(2番目)、...

list_vari[4]="e"は4番地 (5番目の値) である。

(番地=index, suffix[数字])

```
>>> list0
```

```
>>> list0[1:3]
```

```
>>> list0[1:5]
```

```

コマンドプロンプト - python
>>> list0
[10, 'A', [30, 33]]
>>> list0[1:3]
['A', [30, 33]]
>>> list0[1:5]
['A', [30, 33]]

```

スクリプトモード：

演習：(リストの生成・長さ・要素の参照・サブリスト作成)

つぎのプログラムを[C:\pythonPG\sample2411.py]に作成しなさい。

```

1 # 例題ファイル：C:\pythonPG\sample2411.py
2
3 list_vari = [ 10, "str20", [30,33], 44, "55" ]
4 print("リストの生成    =", list_vari)
5 print()
6 print("長さ            =" + str( len(list_vari) ) )
7 print("参照：1番地の値 =" + list_vari[1])
8 print("サブリスト：2番目から4番目 =" , list_vari[1:4] )
9

```

実行結果：

```

C:\PythonPG>python sample2411.py
リストの生成    = [10, 'str20', [30, 33], 44, '55']

長さ            =5
参照：1番地の値 =str20
サブリスト：2番目から4番目 = ['str20', [30, 33], 44]

```

(C) 要素値の変更

次式は1番目(0番地)の要素の値の変更（10 --> '00'）である。

(変更) list_vari[0] = "00" # 1番目の要素を"00"に変更する

```
>>> list0[2]='B'
>>> list0
```

```

C:\PythonPG>python
>>> list0[2]='B'
>>> list0
[10, 'A', 'B']

```

(D) 要素の追加

追加には

リストの最後に1個の要素を追加(append)、
 リストの最後に数個の要素を追加(extend)
 任意の番地に要素を挿入(insert)

する方法がある。

(追加)	最後に1個の要素を追加(.append)	
	リストオブジェクト.append(値)	要素として
	list_vari.append([40, 41, 42])	Listを1個追加
	最後に数個の要素(リスト)を追加(.extend)	
	リストオブジェクト.extend([追加要素 1, 追加要素 2, ...])	
	list_vari.extend([50, 60, 70])	3個の要素を追加
	最後に数個の要素(リスト)を追加(.extend)	
	リスト(len(リスト):) = 任意のリスト	
	list_vari(len(list_vari):) = [50, 60, 70]	
	任意の場所に1個の要素を追加(.insert)	
	リストオブジェクト.insert(要素番地, 値)	
	list_vari.insert(1, "Insert")	# 2番目の前に挿入
	任意の場所に1個の要素を追加(.insert)	
	リストオブジェクト[開始番地:終了番地] = リスト	

list_vari[1:1] = ["aa","bb","cc"] # 2番目の前に挿入

list_vari[1:2] = ["aa","bb","cc"] # 2番目削除して3番目の前に挿入

結果は前項の複数個の要素の追加と同じであるが、“+”演算子を用いる方法でも可能である。

(結合)	最後に追加 (+結合)
	add00 = ["plus50", "plus60", "plus70"]
	list_vari = list_vari + add00

```
>>> list0.append(['cc','dd'])
>>> list0
>>> len(list0)
>>> list0.extend(['EEE','FFF'])
>>> list0
>>> len(list0)
```

```

コマンドプロンプト - python
>>> list0
[10, 'A', 'B']
>>> list0.append(['cc', 'dd'])
>>> list0
[10, 'A', 'B', ['cc', 'dd']]
>>> len(list0)
4
>>> list0.extend(['EEE', 'FFF'])
>>> list0
[10, 'A', 'B', ['cc', 'dd'], 'EEE', 'FFF']
>>> len(list0)
6

```

(E) 要素の削除

要素の削除は、番地で削除する方法と要素の値で削除する方法がある。

[番地で削除]

リストオブジェクト.pop()を使う。
リストオブジェクトとスライスを使う。
del()関数を使う。

[要素の値で削除]

リストオブジェクト.remove()

(削除)	リストオブジェクト.pop(削除番地)	
	リストオブジェクト[開始番地 : 終了番地] = # スライスの使用	←
	del リストオブジェクト[開始番地 : 終了番地] # スライスの使用	←
	del リストオブジェクト[削除番地]	
	リストオブジェクト.remove(削除する値)	←
	list_vari.pop(2)	# 2番地の要素削除
	list_vari[2:4] = []	# (スライス)2, 3番地の要素削除
	del list_vari[2]	# 2番地の要素削除
	del list_vari[2:4]	# (スライス)2, 3番地の要素削除
	list_vari.remove(33)	# 33の値の要素を削除

```
>>> list0
>>> list0[3:5] = []
>>> list0
>>> list0.remove('B')
>>> list0
```

```

C:\ コマンドプロンプト - python
>>> list0
[10, 'A', 'B', ['cc', 'dd'], 'EEE', 'FFF']
>>> list0[3:5] = []
>>> list0
[10, 'A', 'B', 'FFF']
>>> list0.remove('B')
>>> list0
[10, 'A', 'FFF']

```

スクリプトモード：

例題：（要素変更・追加・削除）

つぎのプログラムを[C:\pythonPG\sample2412.py]に作成しなさい。

```

1 # 例題ファイル：C:\pythonPG\sample2412.py
2
3 list_vari = [ 10, "abc", 40 ]
4 print( "初期リスト=", list_vari )
5 print()
6
7 list_vari[1] = 20 # 変更
8 print( "2番目変更=", list_vari )
9 print()
10 list_vari.insert(2, 30)
11 print( "3番目に1個挿入=", list_vari )
12 print()
13 list_vari.append([51,52,53])
14 print( "最後に1個追加=", list_vari )
15 print()
16 list_vari.extend([60,70,80])
17 print( "最後に3個追加=", list_vari )
18 print()
19
20 #del list_vari[4:6]
21 list_vari[4:6] = []
22 print( "5番目、6番目を削除=", list_vari )
23

```

実行結果：

書き方

```

C:\PythonPG>python sample2412.py
初期リスト= [10, 'abc', 40]
2番目変更= [10, 20, 40]
3番目に1個挿入= [10, 20, 30, 40]
最後に1個追加= [10, 20, 30, 40, [51, 52, 53]]
最後に3個追加= [10, 20, 30, 40, [51, 52, 53], 60, 70, 80]
5番目、6番目を削除= [10, 20, 30, 40, 70, 80]

```

(F) ソート

リストの順番をアルファベット順や大きい順などのようにソートするメソッドを持っている。

(ソート)	sort(reverse= True/False) (空・True:数値なら昇順,文字列ならアルファベット順) リストオブジェクト.sort(reverse= True/False) list_vari.sort(reverse= True) # 逆順(降順)	
-------	--	--

(G) 検索 (番地と個数)

リストや文字列に「任意の文字列」が含まれているか否かやその個数を調べることができる。

(場所)	リスト・文字列内で指定した値をもつ要素の最初のインデックス (番地) を取得	
	オブジェクト.index(値)	
	str00.index("3")	# "3"が含まれる番地
(個数)	リスト・文字列内で指定した値をもつ要素の数を取得	
	オブジェクト.count(値)	
	str00.count("3")	# "3"が含まれる個数
	list_vari.count("3")	# "3"が含まれる個数

(H) 文字列からリスト

文字列から配列を作成する。

(文字列)	文字列を[任意の文字列]ごとに区切って配列を作成	
(から)	文字列オブジェクト.split("値")	
(リスト)	str00.split(",")	# ","で区切って配列を作成

(I) リストから文字列

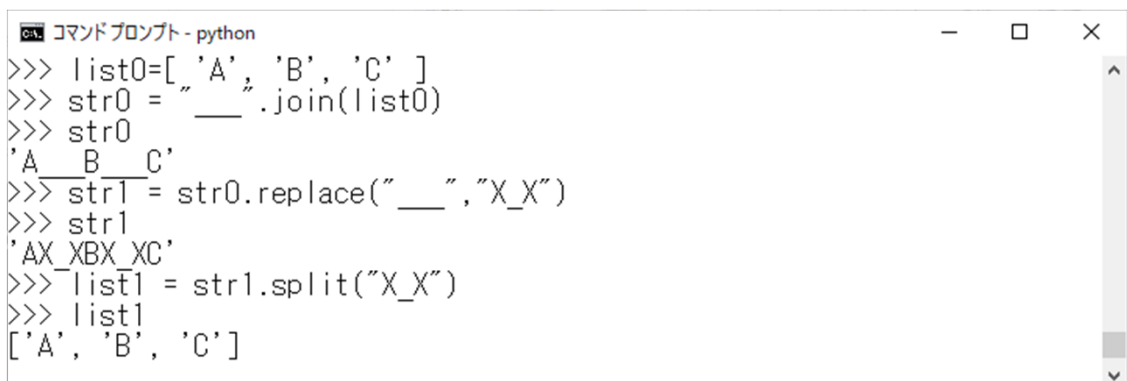
上記事項の逆を操作するものである。

(リスト)	リストを「任意の文字列」で結合した文字列を作成	
(から)	"値".join(リストオブジェクト)	# "値"で各要素を結合した文字列
(文字列)	",".join(list_vari)	# ","で各要素を結合した文字列

(J) 文字列における「任意の文字列」を「任意の文字列」で置換

(置換)	文字列における「任意の文字列」を「任意の文字列」で置換	
	文字列オブジェクト.replace(置換される文字列, 置換する文字列)	
	str00.replace("A", "-")	# "A"を "-" で置換する

```
>>> list0 = [ 'A', 'B', 'C' ]
>>> str0 = "____".join(list0)
>>> str0
'A__B__C'
>>> str1 = str0.replace("____", "X_X")
>>> str1
'AX_XBX_XC'
>>> list1 = str1.split("X_X")
>>> list1
['A', 'B', 'C']
```



```
python
>>> list0=[ 'A', 'B', 'C' ]
>>> str0 = "____".join(list0)
>>> str0
'A__B__C'
>>> str1 = str0.replace("____", "X_X")
>>> str1
'AX_XBX_XC'
>>> list1 = str1.split("X_X")
>>> list1
['A', 'B', 'C']
```


(K) リストとタプル変換

タプルは、追加・削除・ソートなどができないので、そのような要求がある場合にはそのような機能を有するリストへ変換して、その後、タプルに変換すれば、目的に合ったタプルを作ることができる。

(リスト)	「tuple」「list」関数を用いる	
(タプル)	<code>tuple00 = tuple(リストオブジェクト)</code>	# リストからタプルへ
(変換)	<code>list00 = list(タプルオブジェクト)</code>	# タプルからリストへ
	<code>list_vari = list(tuple00)</code>	
	<code>tuple00 = tuple(list_vari)</code>	

(L) 全ての要素の削除 - clear, del

リストの全ての要素を削除する方法がある。

(空化)	<code>リストオブジェクト.clear()</code>
	<code>del リストオブジェクト[:]</code>
	<code>リストオブジェク = []</code>
	<code>list_vari.clear()</code>

例題： (リストからタプル、ソート、番地、カウント、クリア)

つぎのプログラムを[C:\pythonPG\sample2413.py]に作成しなさい。

```

1 # 例題ファイル：C:\pythonPG\sample2413.py
2
3 list_vari = [ 2, 1, 5, 4, 3 ]
4 print( "初期□リスト =", list_vari )
5 print( "リスト⇒タプル=", tuple(list_vari))
6
7 print()
8 list_vari.sort(reverse=True)
9 print( "降順にソート=", list_vari )
10 print( "値5の番地=", list_vari.index(5) )
11 print( "値5の個数=", list_vari.count(5) )
12 print( "クリア      =", list_vari.clear() )
13
14 print()
15 list_vari = [ "10", "20", "30" ]
16 print( "初期□リスト =", list_vari )
17 str00 = "".join( list_vari )
18 print( "文字列化   =", str00 )
19 str11 = str00.replace("0","A")
20 print( "置換0⇒A   =", str11 )
21

```

スクリプトモード :

実行結果

書き方

```

C:\PythonPG>python sample2413.py
初期 リスト = [2, 1, 5, 4, 3]
リスト⇒タプル= (2, 1, 5, 4, 3)

降順にソート= [5, 4, 3, 2, 1]
値5の番地= 0
値5の個数= 1
クリア = None

初期 リスト = ['10', '20', '30']
文字列化 = 10_20_30
置換0⇒A = 1A_2A_3A

```

(M) 多次元リスト

3x2リスト :	list_vari=[[1,2],[3,4],[5,6]]
参照	list_vari[1][0] # 2x1の[3]を参照
2x3x2リスト :	list_vari= [[[1,2],[3,4],[5,6]], [[10,20],[30,40],[50,60]]]
参照	list_vari[1][1][0] # 2x2x1の[3 0]を参照

```

C:\PythonPG>python
>>> list0 = [ [[1,2],[3,4],[5,6]], [[10,20],[30,40],[50,60]] ]
>>> list0[1][1][0]
30

```

(N) ポインタ(アドレス)的な操作

list_vari0 = [1000,2000,3000]	
list_vari1 = list_vari0	# list_vari1 = list_vari0=[1000,2000,3000]
list_vari2 = list(list_vari0)	# list_vari2 = [1000,2000,3000]

```

>>> list0 = [1000,2000,3000]
>>> list1 = list0
>>> list1[1] = 'AAA'
>>> list0
[1000, 'AAA', 3000]
>>> list1
[1000, 'AAA', 3000]
>>>
>>> list0 = [1000,2000,3000]
>>> list2 = list0
>>> list2[1] = 'AAA'
>>> list0
[1000, 'AAA', 3000]
>>> list2
[1000, 'AAA', 3000]
>>>

```

スクリプトモード：

例題： つぎのプログラムを[C:\pythonPG\sample2414.py]に作成しなさい。

```

1 # 例題 ファイル：C:\pythonPG\sample2414.py
2
3 list_vari=[[1,2],[3,4],[5,6]]
4 print("初期3x2リスト")
5 print(list_vari)
6 print("リスト[1][0] =", list_vari[1][0])
7 list_vari= [[ [1,2],[3,4],[5,6]], [[10,20],[30,40],[50,60]] ]
8 print("初期2x3x2リスト")
9 print(list_vari)
10 print("リスト[1][1][0]=", list_vari[1][1][0])
11
12 print()
13 list_vari0 = [ 1000, 2000, 3000 ]
14 print("初期□リスト =", list_vari0)
15 list_vari1 = list_vari0
16 list_vari1[1] -= 500
17 print("500引いたリスト=", list_vari1)
18 print("初期□リスト =", list_vari0)
19
20 print()
21 list_vari0 = [ 1000, 2000, 3000 ]
22 print("初期□リスト =", list_vari0)
23 list_vari1 = list(list_vari0)
24 list_vari1[1] -= 500
25 print("500引いたリスト=", list_vari1)
26 print("初期□リスト =", list_vari0)
27

```

実行結果：

書き方

```

C:\PythonPG>python sample2414.py
初期3x2リスト
[[1, 2], [3, 4], [5, 6]]
リスト[1][0] = 3
初期2x3x2リスト
[[[1, 2], [3, 4], [5, 6]], [[10, 20], [30, 40], [50, 60]]]
リスト[1][1][0]= 30

初期 リスト = [1000, 2000, 3000]
500引いたリスト= [1000, 1500, 3000]
初期 リスト = [1000, 1500, 3000]

初期 リスト = [1000, 2000, 3000]
500引いたリスト= [1000, 1500, 3000]
初期 リスト = [1000, 2000, 3000]

```

B リスト内包表記法 : (Listの特殊な作成要領)

Listから **ある条件を満足する要素を抽出して** 新しいListを作成する場合、

for文で説明した方法 :
<pre># 例題ファイル : C:¥pythonPG¥sample2425.py print() base_list = [11, 12, 13, 14, 15, 16, 17] print("base_list=", base_list) new_list = [] for element in base_list: if element % 2 == 0: new_element = element * 2 new_list.append(new_element) print("new_list1=", new_list)</pre>
リスト内包表記法 :
<pre>print() base_list = [11, 12, 13, 14, 15, 16, 17] new_list = [element * 2 for element in base_list if element % 2 == 0] print("new_list2=", new_list)</pre>

[0]*5

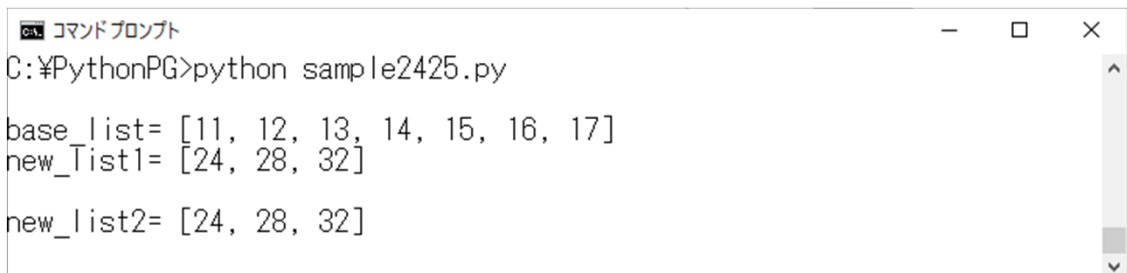
```
>>> base_list = [ 11, 12, 13, 14, 15, 16, 17 ]
>>> new_list = [element * 2 for element in base_list if element % 2 == 0]
>>> new_list
[24, 28, 32]
>>>
```

例題 :

上のプログラムは、Listの要素の中で、偶数の要素を抽出して、その値を2番した要素で作られるListを作成するプログラムである。このプログラムを[C:¥pythonPG¥sample2425.py]に作成しなさい。

実行結果 :

偶数要素(12,14,16)を抽出して、その値を2倍(24,28,32)にした要素のListが作成されていることが分かる。



```

C:¥PythonPG>python sample2425.py

base_list= [11, 12, 13, 14, 15, 16, 17]
new_list1= [24, 28, 32]
new_list2= [24, 28, 32]
```

(2) タプル(Tuple)

タプルはリストと同じようなものであるが、**要素の変更・追加や削除ができない**点が大きく異なる。使い方は、この変更できない点を有利にするプログラムの時に適しているが、具体的に思いつかない。ここでは、タプルのオブジェクトとして、**tuple00** を用いる。静的なメソッドはほとんどリストと同じである。

宣言 初期化	タプルオブジェクト = (element1, element2, element3, ...)
	tuple00 = ("A", "B", "C")
	要素オブジェクトelement1,2,3,..は数値、文字列またはリストである。 注意：各elementは同じ型である必要はない。
	# tuple()関数を用いる タプルオブジェクト = tuple(文字列) タプルオブジェクト = tuple(リスト)
	tuple00 = tuple("ABC") # ("A", "B", "C") tuple00 = tuple(["A", "B", "C"]) # ("A", "B", "C")
参照	タプルオブジェクト[番地] # 番地は0,1,2,..である。
	tuple00[1] # 1番地(=2番目)の要素の参照
サブ タプル	sub_tuple = タプルオブジェクト[スライス]
	sub_tuple = tuple00[2:5]
長さ	len(タプルオブジェクト)
	len(tuple00)
検索	タプルオブジェクト.index("値") # 「値」と同じ要素の最初の番地
	タプルオブジェクト.count("値") # 「値」と同じ要素の個数
	tuple00.index("D") # 値「D」の最初の番地
	tuple00.count("D") # 値「D」の個数

下記のプログラムを[C:\pythonPG\sample2430.py]に作成して、実行しなさい。

```

1 # 例題ファイル：C:\pythonPG\sample2430.py
2 print()
3 aaa = ("A","B","C","D","E","D")
4 print( "タプル=", aaa )
5 print( "長さ=", len(aaa) )
6 print( "2,3,4番目のサブタプル=", aaa[2:5] )
7
8 print()
9 tuple00 = tuple("ABC")
10 print( "(ABC=>Tuple)=", tuple00 )
11 tuple00 = tuple("A", "B", ["C","D"])
12 print( "(List=>Tuple)=", tuple00 )
13

```

実行結果：

```

コマンドプロンプト
C:\PythonPG>python sample2430.py

タプル= ('A', 'B', 'C', 'D', 'E', 'D')
長さ= 6
2,3,4番目のサブタプル= ('C', 'D', 'E')

(ABC=>Tuple)= ('A', 'B', 'C')
(List=>Tuple)= ('A', 'B', ['C', 'D'])

```

(3) 辞書 (連想配列: Dict)

これは他の言語では**連想配列**と呼ばれている。





すなわち

```
fruits["みかん"]=100, fruits["りんご"]=250, fruits["いちご"]=780
```

のように配列fruits[]が果物の単価を表す場合にはこのような表現が便利である。

ここでは、辞書オブジェクトとして **dict00** を用いる。

A 宣言・参照・変更

宣言	辞書オブジェクト = { キー1 : 値1, キー2 : 値2, ... }
初期化	dict00 = { "key1": "A", "key2": 20, "key3": "C" } 
	要素オブジェクト値1,2,3,...は数値、文字列である。 キーにはオブジェクトが指定できるが、 キーは唯一 (Unique) かつ変更不可である。
長さ	len(辞書オブジェクト) print(len(dict00))
クリア	辞書オブジェクト.clear() dict00.clear() 
参照	辞書オブジェクト(キー) print(dict00["key1"]) 
変更	辞書オブジェクト(キー) = 変更したい値 dict00["key1"] = "A00"  # "key1"のValueを"A00"にする

インタラクタモード :

```
>>> dic0 = {"key1": "AA"}
>>> dic0
{'key1': 'AA'}
>>> dic0["key2"] = "BB"
>>> dic0
{'key1': 'AA', 'key2': 'BB'}
>>> dic0.update( {"key3": "CC", "key4": "DD"} )
>>> dic0
{'key1': 'AA', 'key2': 'BB', 'key3': 'CC', 'key4': 'DD'}

>>> dic0["key2"] = "22"
>>> dic0
{'key1': 'AA', 'key2': '22', 'key3': 'CC', 'key4': 'DD'}

>>> dic0.keys()
dict_keys(['key1', 'key2', 'key3', 'key4'])
>>> dic0.values()
dict_values(['AA', '22', 'CC', 'DD'])
>>> dic0.items()
dict_items([('key1', 'AA'), ('key2', '22'), ('key3', 'CC'), ('key4', 'DD')])
>>>
```

スクリプトモード：

下記のプログラムを[C:\¥pythonPG¥sample2440.py]に作成して、実行しなさい。

```

1 # 例題ファイル：C:\¥pythonPG¥sample2440.py
2
3 print()
4 print("====宣言・参照・変更・クリア====")
5 print("宣言と参照")
6 dict00 = { "key1": "A", "key2": 20, "key3": "C" }
7 print( dict00 )
8 print( "1番目の値==>", dict00["key1"] )
9 print( "2番目の値==>", dict00["key2"] )
10 dict00["key1"] = "A00"
11 print( "1番目の値変更==>", dict00 )
12 dict00.clear()
13 print( "クリア==>", dict00 )
14

```

実行結果：

```

C:\¥PythonPG>python sample2440.py

====宣言・参照・変更・クリア====
宣言と参照
{'key1': 'A', 'key2': 20, 'key3': 'C'}
1番目の値==> A
2番目の値==> 20
1番目の値変更==> {'key1': 'A00', 'key2': 20, 'key3': 'C'}
クリア==> {}

```

B 要素の追加と削除

追加 (1個)	辞書オブジェクト(newキー) = 値 dict00["newkey4"] = "D"
追加 (複数)	辞書オブジェクト.update(追加する辞書オブジェクト) dict00.update({"upkey5": "E", "upkey6": "F" })
削除 (1個)	del 辞書オブジェクト[キー] del dict00["key1"]
削除 (1個)	リストオブジェクト = 辞書オブジェクト.pop(キー) # 削除した値がリストに入る、 dict00は削除された残りの辞書 リストオブジェクト = dict00.pop("key1")
削除	削除した要素のタプル = 辞書オブジェクト.popitem() # ランダム(後ろから)に1個 tuple00 = dict00.popitem()

スクリプトモード：

下記のプログラムを[C:¥pythonPG¥sample2443.py]に作成しなさい。

```

1 # 例題ファイル：C:¥pythonPG¥sample2443.py
2
3 dict00 = { "key1": "A" }
4 print( "初期 □ タプル=", dict00 ); print()
5
6 dict00.update( { "key2": "B", "key3": "C" } )
7 print( "1番目以降に2個追加=", dict00 ); print()
8 dict00["key4"] = "B"
9 print( "4番目に1個追加=", dict00 )
10
11 print()
12 del dict00["key1"]
13 print( "1番目(key1)削除", dict00 ); print()
14 print( "すべて削除かつ削除するKeyと値をタプルで出力" )
15 while dict00:
16     tuple00 = dict00.popitem()
17     print( " ", tuple00 )
18
19 print( "最終の配列=", dict00 )
20

```

[C:¥pythonPG¥sample2443.py]を実行しなさい。

```

コマンドプロンプト
C:¥PythonPG>python sample2443.py
初期   タプル= { 'key1': 'A' }

1番目以降に2個追加= { 'key1': 'A', 'key2': 'B', 'key3': 'C' }
4番目に1個追加= { 'key1': 'A', 'key2': 'B', 'key3': 'C', 'key4': 'B' }
1番目(key1)削除 { 'key2': 'B', 'key3': 'C', 'key4': 'B' }
すべて削除かつ削除するKeyと値をタプルで出力
    ('key4', 'B')
    ('key3', 'C')
    ('key2', 'B')
最終の配列= {}

```


C key(キー)とvalue(値)の一括操作

キー取得	リストオブジェクト = list(辞書オブジェクト.keys())
	key_list00 = list(dict00.keys())
値取得	リストオブジェクト = list(辞書オブジェクト.values())
	value_list = list(dict00.values())
キーと値の一覧を取得	
	キーと値のペアのタプルをリスト = list(辞書オブジェクト.items())
	tuple_list = list(dict00.items())
キーの有無	
	キー in 辞書オブジェクト # True or False(出力)
	print "key1" in dict00
繰り返しで、キー・値 の取得・表示	
	for key in dict00.keys():
	print(key)
	for value in dict00.values():
	print(value)
	for key, value in dict00.items():
	print(key, value)

(4) 集合(Set) 重複した値は保持したくない

要素は唯一であり、set変数は集合同士を比較する演算子が用意されている。

ここではセットオブジェクトを **set00** とする。

List, Tupleと異なり, **index参照はできない**、つまり、set00[1]のような参照はエラーになる。

このために、削除は値で、追加は追加関数で追加になる。

追加は最後に追加されるとは限らない。コンピュータまかせである。

何番目に追加されるかわからない。それ故に、indexは意味がないものになる。

宣言	Setオブジェクト = { element1, element2, ... }
	Setオブジェクト = set([1,2,3,4,5]) # Listからsetへ
	要素は 唯一(unique) である。
	set00 = { 0, 1, 2, 3, 4, 5 }
長さ	len (setオブジェクト)
	len(set00)
クリア	setオブジェクト.clear()
	set00.clear()
追加	setオブジェクト.add(値)
	set00.add(11)
削除	setオブジェクト.remove(値)
	set00.add(11)
要素参照	for value in set00: print(value)
含	値 in setオブジェクト # 出力 True/False
	10 in set00
	setオブジェクト.issuperset(リストオブジェクト) # 出力 True/False
	set00.issuperset([10,50])

参照に
set00[0]

インタラクタモード :

```
>>> set0 = {10,20,30,40,50}
>>> set0
{40, 10, 50, 20, 30}
>>>
```

スクリプトモード：

下記のプログラムを[C:\¥pythonPG¥sample2450.py]に作成しなさい。

```

1 # 例題ファイル：C:\¥pythonPG¥sample2450.py
2
3 print( "=====集合=====")
4 print( "宣言：set00 = { 0, 10, 20, 30, 40, 50 }" )
5 print()
6 set00 = { 0, 10, 20, 30, 40, 50 }
7 print( "表示(順番がランダムになっている)" )
8 print( set00 ); print()
9
10 set00.add(11)
11 print("値11を追加", set00); print()
12
13 set00.add(33)
14 print("値33を追加", set00); print()
15
16 set00.remove(33)
17 print("値33を削除", set00); print()
18
19 for value in set00:
20     print( value, end=" ", " ")
21
22 print(); print()
23 print( "値10が含まれるか?", 10 in set00 ); print()
24
25 print( "値[10,50]が含まれるか?", (set00.issuperset([10, 50])) );
26
27 set00.clear()
28 print("クリア", set00); print()
29

```

実行結果：

```

C:\¥PythonPG>python sample2450.py
=====集合=====
宣言：set00 = { 0, 10, 20, 30, 40, 50 }
表示(順番がランダムになっている)
{0, 40, 10, 50, 20, 30}
値11を追加 {0, 40, 10, 11, 50, 20, 30}
値33を追加 {0, 33, 40, 10, 11, 50, 20, 30}
値33を削除 {0, 40, 10, 11, 50, 20, 30}
0, 40, 10, 11, 50, 20, 30,
値10が含まれるか? True
値[10,50]が含まれるか? True
クリア set()

```

演算子	意味 (2個の集合: set00, set11 出力: set22)
和集合 union() {*, *} .update()	2個の集合set00,set11の要素で作られる集合 set22 = set00 set11 set22 = set00.union(set11) set22 = {*set00, *set11} OR set00.update(set11) # 出力はset00
積集合 & .intersection()	2個の集合の両方に含まれる要素で作成される集合 set22 = set00&set11 set22 = set00.intersection(set11) AND
差集合 - .difference()	set00のみに含まれる要素で作成される集合 set22 = set00 - set11 set22 = set00.difference(set11)
対称差 ^ .symmetric_difference()	set00、set11のみに含まれる要素で作成される集合 set22 = set00 ^ set11 set22 = set00.symmetric_difference(set11)
部分集合 <= .issubset	set00がset11 の部分集合である場合にはflg00はtrueを出力。 flg00 = set00<=set11 sflg00 = et00.issubset(set11)

インタラクタモード:

```

>>> set0 = {1,2,3}
>>> set1 = {3,4}
>>>
>>> set0|set1
{1, 2, 3, 4}
>>> set0&set1
{3}
>>>
>>> set0-set1
{1, 2}
>>> set1-set0
{4}
>>>
>>> set0^set1
{1, 2, 4}
>>>

```

2 関数

関数を使うためには、「関数定義」と「関数の呼び出し」の2通りがある。

定義および呼び出しの関係はC言語と同じであり、**構造的には「呼び出す」前に「定義」する必要がある。**

(1) def 関数名(引数1, 引数2, ..):

項 目	書 式
定 義	def 関数名(引数1, 引数2, ..): 処理 return 戻り値 # 戻り値は (数値・文字列・リスト・タプルなど)
呼び出し	戻り値Get変数 = 関数名(引数1の値, 引数2の値, ...)

	引数・戻り値の有り・無しのすべての組み合わせの場合を示す。
例題 1	引数なし・戻り値なし def aaa(): print("関数内：引数なし・戻り値なし") aaa() # 呼び出し
例題 2	引数なし・戻り値あり def bbb(): print("関数内：引数なし・戻り値あり") return ["ret1", "ret2"] ret = bbb() # 呼び出し print("Getした戻り値=", ret)
例題 3 1	引数あり・戻り値なし：：引数：数値や文字列 def ccc(x, y): # 引数 x,yはリスト、辞書、タプル、setでも可 print("引き渡された値 =", str(x) + ", " + y) ccc(10, "str_10") # 呼び出し
例題 3 2 (後述)	引数あり・戻り値なし：： def ddd(*x): # 引数： タプル(呼び出しではリスト、タプルに関わらず) print("引き渡された値 =", x) ddd(10, "str_10") # (呼び出しの引数は可変長)
例題 3 3 (後述)	引数あり・戻り値なし：： def eee(**x): # 引数：辞書 print("引き渡された値 =", x) eee(key1="str1", key2="str2") # (呼び出しの引数は可変長)
例題 3 4 (後述)	引数あり・戻り値なし：： def fff(x, *y, **z): # 引数：複合 print("引き渡された値 =", x, y, z) fff("111", "222", "333", key1="444", key2="555") # (呼び出しの引数は可変長)
例題 4	引数あり・戻り値あり def ggg(x): print("引き渡された値(半径) =", x) return x*x*3.141592 ret = ggg(10) # 呼び出し print("Getした戻り値(円の面積)=", ret)

項 目	書 式
定 義	def 関数名(引数 1, 引数2, ..): 処理 return 戻り値 # 戻り値は (数値・文字列・リスト・タプルなど)
呼び出し	戻り値Get変数 = 関数名(引数 1 の値, 引数 2 の値, ...)

インタラクタモード：

戻り値なし・引数なし

```
>>> def func_A():
...     print("AAA")
...
>>> func_A()
AAA
>>>
```

```
def func_A():
    print("AAA")
func_D("DDD") )
```

戻り値あり・引数なし

```
>>> def func_B():
...     return "ret_BBB"
...
>>> print( "戻り値=", func_B() )
戻り値= ret_BBB
>>>
```

```
def func_B():
    return "ret_BBB"

print( "戻り値=", func_B() )
```

戻り値なし・引数あり

```
>>> def func_C( str0 ):
...     print( str0 )
...
>>> func_C("CCC")
CCC
>>>
```

```
def func_C( str0 ):
    print( str0 )

func_C("CCC")
```

戻り値あり・引数あり

```
>>> def func_D( str0 ):
...     return "ret"+str0
...
>>> print( "戻り値=", func_D("DDD") )
戻り値= retDDD
>>>
```

```
def func_D( str0 ):
    return "ret"+str0

print( "戻り値=", func_D("DDD") )
```

課題2-2-010 :

プログラム :

```
def func_A( *w ):
    print( "func_A=", w )
def func_B( *w ):
    print( "func_B=", w )

func_A( 11, 22, 33 )    # func_Aの呼び出し
list_b = [ 11, 22, 33 ]
func_B( list_b )        # func_Bの呼び出し
```

実行結果 :

```
func_A= ( 11, 22, 33 )
func_B= ( [11, 22, 33], )
```


タプル: 要素に注意

上記の結果から分かる事を考察しなさい。(口答)

課題2-2-020 : 上記の結果を考慮して、下記をプログラムしなさい。

関数名 : func_A (引数あり、戻り値あり)

引数は整数(v)と3次元タプル(w)である。

まず、 func_B() を実行した後、下記を処理する。

引数 v とタプル w の要素との積をリスト変数 list_x に格納して、戻り値とする。

関数名 : func_B (引数なし、戻り値なし)

処理は print(" 関数func_A()内から関数func_B()が呼ばれた") のみである。

条件 :

a = 3, list_b = [11, 22, 33]

ret = func_A(a, list_b)

print("戻り値= ", ret)

上記を[C:\pythonPG\sample2-2-010.py]にプログラムし、実行しなさい。

```
# 例題ファイル : C:\pythonPG\sample2-2-010.py
def func_A( v, *w ):
    func_B()
    w1 = list( w[0] )
    list_x = []
    for element in w1:
        x1 = v*element
        list_x.append( x1 )
    return list_x
def func_B():
    print( " 関数func_A()内から関数func_B()が呼ばれた" )
print()
a = 3
list_b = [11,22,33]
ret = func_A( a, list_b )
print( "戻り値=", ret )
```

(2) def 関数名(引数1, 引数2, . . . ,*引数シーケンス,**引数辞書):

引数が前の定義と異なり、引数にListや辞書が含まれるばあいである。

その扱いは、C言語のポインタ（1次元配列）や2重ポインタ（2次元配列）の場合に相当する。

項 目	書 式
定 義	<pre>def 関数名(引数1, 引数2, . . . ,*引数シーケンス(タプル),**引数辞書): 処理 return 戻り値 # 戻り値は（数値・文字列・リスト・タプルなど）</pre> <p>[呼び出しの引数]と対応する[関数の引数]は</p> <pre>呼び出し引数(10, 11, "a3", 30, 40, dict1=2, dict2="3") 関数の引数 (x1, x2, *list00, **dict00)</pre> <p>ならば</p> <pre>10==>x1, 11==>x2, "a3",30,40 ==> list00 dict1=2, dict2="3" ==> dict00</pre> <p>に対応する。</p>
呼び出し	<p>戻り値Get変数 = 関数名(引数 1 の値群, 引数 2 の値群, ...)</p> <p>引数の値は、単項ごとに入力すると、システムが*のデータの一部か？ **のデータの一部か を判断して関数で処理される。</p> <p>呼び出しの引数は可変であるが、処理はそれに対応するように書いてある必要がある。</p>
例 題	<p>引数なし・戻り値なし の場合のみを説明する。</p> <p># 例題ファイル：C:\pythonPG\sample2462.py</p> <pre>print("====def 関数名)====")</pre> <pre>def func00(vari1, vari2, *list_vari, **dic_vari): print(vari1) print(vari2) print(list_vari) print(dic_vari)</pre> <pre>func00(10, 'aa' , 30, 'bb', 'cc', dic0=60, dic1='dd') # 呼び出し</pre> <pre>print("=====")</pre>

上記を [C:\pythonPG\sample2462.py] にプログラムしなさい。

実行結果：

実行

```

C:\PythonPG>python sample2462.py
====def 関数名(引数1, 引数2, . . . ,*引数シーケンス,**引数辞書)====
10
aa
(30, 'bb', 'cc')
{'dic0': 60, 'dic1': 'dd'}
=====

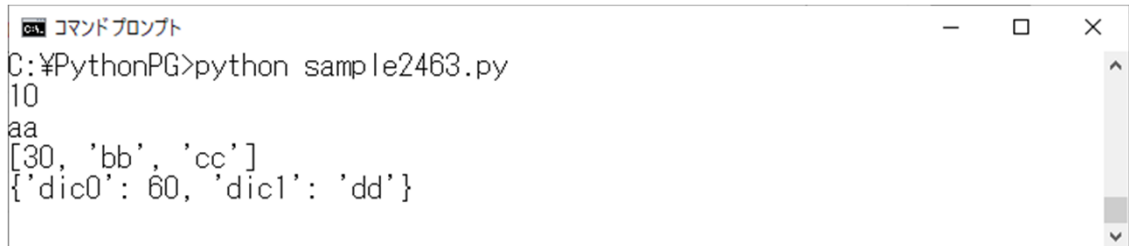
```


通常は、上記のようなことをしないで、つぎのようにプログラムする。

```
def func00( vari1, vari2, list_vari, dic_vari):
    print(vari1)
    print(vari2)
    print(list_vari)
    print(dic_vari)
list0 = [ 30, 'bb', 'cc' ]
dic00 = { "dic0":60, "dic1":'dd' }
func00(10, 'aa' , list0, dic00 )  # 呼び出し
```

実行結果：

実行



```
C:\PythonPG>python sample2463.py
10
aa
[30, 'bb', 'cc']
{'dic0': 60, 'dic1': 'dd'}
```

まとめて書くと、つぎのようになる。

関数	def func00(vari1, vari2, *list_vari, **dic_vari):
呼び出し	func00(10, 'aa' , 30, 'bb', 'cc', dic0=60, dic1='dd') # 呼び出し
関数	def func00(vari1, vari2, list_vari, dic_vari):
呼び出し	list0 = [30, 'bb', 'cc'] dic00 = { "dic0":60, "dic1":'dd' } func00(10, 'aa' , list0, dic00) # 呼び出し

(3) 引数に初期値のある引数

引数に初期値がある場合の書式はつぎのようになる。

項 目	書 式
定 義	def 関数名(引数 1 =初期値 1, 引数2 = 初期値 2, ..): 処理 return 戻り値 #この初期値は、いわゆるデフォルト値である。
呼び出し	戻り値Get変数 = 関数名(引数 1 の値, 引数 2 の値, ...) 戻り値Get変数 = 関数名() # この場合デフォルト値が使われる。

インタラクタモード：

```
def func_A( carname = "TOYOTA" ):
    print(carname)

func_A( )
func_A( "NISSAN" )
func_A( carname="HONDA" )
```

```
コマンドプロンプト - python
>>> def func_A( carname = "TOYOTA" ):
...     print(carname)
...
>>> func_A( )
TOYOTA
>>> func_A( "NISSAN" )
NISSAN
>>> func_A( carname="HONDA" )
HONDA
>>>
```

スクリプトモード：

下記のプログラムを[C:\pythonPG\sample2463.py]に作成し、実行しなさい。

デフォルトで **"TOYOTA"** に設定する。引数で値を与えない場合と **'NISSAN'** を与える場合を考える。

```
1 # 例題ファイル：C:\pythonPG\sample2464.py
2
3 def func_ini(carname="TOYOTA"):
4     CarName = carname
5     print(CarName)
6
7 print()
8 print("引数に初期値を与えない場合")
9 func_ini()
10
11 print()
12 print("引数に値を与える場合")
13 func_ini(carname="NISSAN")
14
```

実行結果

```
コマンドプロンプト
C:\PythonPG>python sample2464.py

引数に初期値を与えない場合
TOYOTA

引数に値を与える場合
NISSAN
```

(4) 関数オブジェクト 後述の関数デコレータとも関連する

pythonでは、関数（プログラムの形式言語）そのものを変数（オブジェクト）として扱うことができる。これを**関数オブジェクト**という。この使用例は関数デコレータを参照しなさい。例題で示す。

例題1	<pre>def func_obj1(): print("関数中表示") # 関数オブジェクトの生成 obj1 = func_obj1 # 注意;()は不要である。 obj1() # func_obj1()と同じこと</pre>	<pre>def AA1(): print("ObjectAA1") def AA2(): print("ObjectAA2") def BBB(x): x() aa=[1,2] aa[0] = AA1 aa[1] = AA2 for n in [0,1]: BBB(aa[n])</pre>
例題2	<pre>def func_obj2(vari): print("関数中表示", vari) return "戻り値だよ" # 関数オブジェクトの生成 obj2 = func_obj2 # 注意;()は不要である。 obj2("引数・戻り値あり") # func_obj2("引数・戻り値あり")と同じこと</pre>	

下記のプログラムを[C:¥pythonPG¥sample2465.py]に作成し、実行しなさい。

```

1 # 例題ファイル : C:¥pythonPG¥sample2465. py
2
3 def func_obj1():
4     print("関数オブジェクト使用の関数中で表示")
5
6 print()
7 obj1 = func_obj1 # 関数オブジェクト
8 obj1() # ==>func_obj1()
9 print()
10

```

実行結果：

```

C:\PythonPG>python sample2465.py

関数オブジェクト使用の関数中で表示

```

(5) 内部関数

for文やif文でネストなる概念があったが、関数にも同様の概念がある。

すなわち、関数(外部関数: outer function)の中に関数(内部関数: inner function)が存在する。
通常の使い方と関数オブジェクトを用いた場合の2通りを例題で示す。

A 定義	例題1	<pre>def outer_function(): print("外部関数での表示") def inner_function(): print("内部関数での表示") inner_function() outer_function()</pre>
	例題2	<pre>def outer_function(): print("外部関数での表示") def inner_function(): print("内部関数での表示") inner_obj=inner_function #関数オブジェクトの生成 return inner_obj inner_function_obj = outer_function() inner_function_obj()</pre>

B nonlocal宣言

後述するglobal変数と同じように、外部関数の変数を内部関数内で参照することが可能である。
これが、nonlocal宣言である。(下記の例ではnonlocal ==> global NG)

例題	<pre>def outer_function(): vari = 200 print(vari) def inner_function(): nonlocal vari print(vari + 22) # 外部関数の変数に内部関数で 2 2 を加算 inner_function() outer_function()</pre>
----	--

(6) 無名関数(演算子 : lambda)

関数名がなく、関数オブジェクトを用いる関数で、短い処理などの場合に有効と思われる。

ラムダ演算子 : lambda

書式	lambda 引数: 戻り値(=処理)
例題	<pre>func = lambda x, y: x+y twice_value = func(3,5) print(twice_value) #=> 8</pre>
	<pre>>>> func = lambda x, y: x+y >>> print(func(3,5)) 8 >>></pre>

3 関数デコーダ・ドキュメンテーション文字列・関数アノテーション

(1) ドキュメンテーション文字列(ドキュメントストリング)

モジュール(ファイル)の冒頭、クラスの冒頭、関数の先頭に**三重クォート** `"""..(複数行可).."""` で記述した**コメント(説明文)**である。

つまり、**ファイル・クラス・関数などの説明文**である。

モジュール(ファイル名)オブジェクト、クラスオブジェクト、

関数オブジェクトの `__doc__` アトリビュートで参照する。例題を下記に示す。

アクセス書式	ファイル名. <code>__doc__</code>	# ファイルの説明
	ファイル名.クラス名. <code>__doc__</code>	# クラスの説明
	ファイル名.クラス名.関数名. <code>__doc__</code>	# 関数の説明
	オブジェクト名.(関数名). <code>__doc__</code>	# クラス(関数)の説明


```

# doc_base.py
"""
ファイルの説明だよ。
これには、クラスMyClassとメソッドmyfunc()がある。
"""

class MyClass:
    """マイクラスの説明だよ。"""
    def myfunc( ):
        """関数(メソッド)の説明だよ"""

```

赤丸は「”」でも可である。

```

# doc_call.py
import doc_base

print (doc_base.__doc__)           #=> ファイルの説明
print (doc_base.MyClass.__doc__)   #=> マイクラスの説明
print (doc_base.MyClass.myfunc.__doc__) #=> 関数(メソッド)の説明

myclass = doc_base.MyClass()
print(myclass.__doc__)             # オブジェクトからのアクセス

```

上記の2個のプログラムを[C:¥pythonPG¥doc_base.py], [C:¥pythonPG¥doc_call.py]に作成し、doc_call.pyを実行しなさい。

```

コマンドプロンプト
C:¥PythonPG>python doc_call.py

ファイルの説明だよ。
これには、クラスMyClassとメソッドmyfunc()がある。

マイクラスの説明だよ。
関数(メソッド)の説明だよ)
マイクラスの説明だよ。

```

#で始まるコメントとの違いは、上記例題で分かるように画面に表示可能なことである。

アクセスは後述するクラスメソッド・クラス変数のようにアクセスする。

しかしながら、プログラマーに必要なコメントを何故表示させる必要があるのかという疑問は残る。

(2) 関数デコレータ

関数デコレータとは、「関数に対して新機能を追加する」ための機能です。クラスの継承に似た機能である。

- (A) 内部関数の考え方で機能を追加する方法：
- (B) 関数デコレータで機能を追加する方法：

(A) 内部関数の考え方で機能を追加する方法：

```
# func_deco.py
def deco_func(func00):
    def new_func():
        print('付加機能 1 ')
        ret_str = func00()
        print(ret_str)
        print('付加機能 2 ')
    return new_func
def base_func():
    return "基礎の機能"
func11 = base_func # 関数オブジェクト
new_func2 = deco_func(func11) # 新たに機能を追加した関数オブジェクトを作成
new_func2()           # 新たに作成した関数の実行
```

新機能(関数)の追加

追加する関数

上記のプログラムを[C:¥pythonPG¥func_deco.py]に作成し、実行しなさい。

```
コマンドプロンプト
C:¥PythonPG>python func_deco.py
付加機能 1
基礎の機能
付加機能 2
```

4 ジェネレータ関数

pythonには、特殊な関数としてジェネレータがある。
これは、Listを関数で実現したようなものである。

定義	<pre>def 関数名(): yield 値1 yield 値2 yield 値3 ...</pre>	<div>同じ変数名であるが 値が異なることに</div>
取得	<pre>obj = 関数名() # ジェネレータオブジェクトを生成する value = obj.__next__() print(value)</pre>	取得
例題	<pre># 例題ファイル : C:¥pythonPG¥sample2480.py print("=====ジェネレータ===== ") def lions(): yield '秋山選手' yield '源田選手' yield '浅村選手' yield '山川選手' obj = lions() # ジェネレータオブジェクトの生成 for person in obj: print(person) print("=====")</pre>	

上記のプログラムを[C:¥pythonPG¥sample2480.py]に作成し、実行しなさい。

```

コマンドプロンプト
C:¥PythonPG>python sample2480.py
=====ジェネレータ=====
秋山選手
源田選手
浅村選手
山川選手
=====

```

5 変数のスコープ（ローカル・グローバル変数の有効範囲）

変数にはその有効範囲で

ローカル(local)変数

グローバル(global)変数

の呼び方がある。

```
y=100 # global(モジュール)領域
```

```
def func():
```

```
    #ブロックA
```

```
    x=200
```

```
    Y=0
```

```
    y=y+11
```

```
    #ブロックA
```

```
for element in [ "みかん", "なし", "りんご"]:
```

```
    #ブロックB
```

```
    x=300
```

```
    print( y )
```

```
    #ブロックB
```

```
else:
```

```
    #ブロックC
```

```
    x=400
```

```
    print( y+33 )
```

```
    #ブロックC
```

```
#ブロックD
```

```
#ブロックD
```

変数xはすべて別々の**ローカル変数**でその色のブロックのみが有効範囲である。

変数y=100は、青色(参照のみ)・オレンジ色(参照のみ)・灰色ブロックは有効範囲である。

ただし、黄色ブロックの変数yは異なるもの(local変数)である。

なお、上記はfor文の例であるが、if文でも同じ結果です。

変数yは**グローバル変数**という。ただし、下記のように関数内で定義すると関数内で参照・更新が可能ある。

z	グローバル領域で定義・宣言	関数内では 参照のみ 可能である 関数内の式で使う場合はlocal変数である。 それ故、初期化する必要がある。 global 領域： z = 1000 関数内： z1 = z+100 #OK 関数内： z = z + 100 # NG 初期化必要
global y	グローバル領域で定義と宣言 または 関数内で定義と宣言。	関数内のyを関数外でも参照・変更可能 global 領域： y1 = 1000 関数内： global y1 y1 = y1 + 100 関数内： global y2 y2 = 2000 y2 = y2 + 100 global 領域： print(y2)

global と nonlocal の関係を最後に例題で示す ←

下記のプログラムを[C:\pythonPG\sample2470.py]に作成し、実行しなさい。

```

1 # 例題ファイル : C:\pythonPG\sample2470.py
2
3 y1 = 1000
4 y3 = 3000
5 def func_A():
6     print( "Global 参照のみ==", y3 + 333 )
7     global y1
8     y1 = y1 + 111
9     global y2
10    y2 = 2000
11    y2 = y2 + 222
12
13 func_A()
14 print ( "Global 変数==", y1 )
15 print ( "Global 変数==", y2 )
16

```

実行結果：

実行

```

コマンドプロンプト
C:\PythonPG>python sample2470.py
Global 参照のみ== 3333
Global 変数== 1111
Global 変数== 2222

```

6 pass(何もしない)

設計の当初段階で、その関数名だけを決めて、全体的な構造を決定する場合がある。

つまり、処理事項がない関数名だけのプログラムを作成する必要がある。

書式	def 関数名(): pass
----	--------------------

Global / Nonlocal 変数の説明

例題ファイル: C:\pythonPG\Sample2475.py

aaa = 111

bbb = 222

def func10(): # ++++++

print("Global参照= ", aaa)

global bbb

bbb = 333 # global変数の更新

ddd = 444

def func20(): # -----

nonlocal ddd

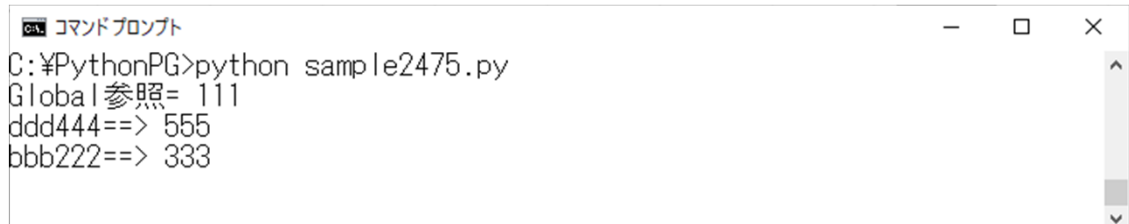
ddd = 555 # 外部関数の変数の更新

func20() # 内部関数の実行

print("ddd444==>", ddd)

func10() # ++++++

print("bbb222==>", bbb)



```

C:\PythonPG>python sample2475.py
Global参照= 111
ddd444==> 555
bbb222==> 333
  
```

