計算物理学II

第7回
数値計算プログラミング入門（3）：
リスト, タプル, 辞書

秋山 進一郎

2025年11月21日

授業日の確認

・全10回

- ・第1回：10月3日（金）
- ・第2回：10月10日（金）
- ・第3回：10月17日（金）
- ・第4回：10月24日（金）★
- ・第5回：10月31日（金）
- ・第6回：11月14日（金）★
- **・第7回：11月21日（金）**
- ・第8回：12月5日（金） ★
- ・第9回：12月12日（金）
- ・第10回：12月19日（金）★

・★の付いた授業にてレポート課題を配布予定

今日の授業の目標

- リスト, タプル, 辞書を使えるようになる
- メモリ上でのリストの表現と「参照の値渡し」について知る

毎回の授業の流れ

“compphy2”に移動する

↓

“git fetch”と“git merge”で演習資料を入手

↓

“lecture*”に移動し,
“lecture_material_*.pdf”を開いて演習開始

本日の演習内容

- リスト, タプル, 辞書を使えるようになる
- メモリ上でのリストの表現と「参照の値渡し」について知る
- 第5回の演習資料について, Collatz問題(p.55~58)とLeibnizの公式(p.63~67)のサンプルコードを/lecture5/src/の中に置きました
  - 各自で確認しておいてください

リストとは?

・Pythonでデータをまとめて保持・処理したい場合，リスト(list)というデータ構造を使う

　・Python以外の言語では，配列(array)と呼ばれることが多い

・リストは[ ](鉤括弧)で囲み，カンマで区切ることで定義する

　・リストにはどんなものでも入れられる

　　例1) [1, 2, 3] → 整数1, 2, 3からなるリスト

　　例2) [“A”, “B”, “C”] → 文字列からなるのリスト

　　例3) [“A”, 1.0, 2] → 文字列, 浮動小数点数, 整数からなるリスト

リストを作ってみよう

- ここでは，変数aに整数からなるリストを代入してみます
  - まず，Python3を対話モードで起動しましょう
  - 次に，a=[1, 2, 3]と打ってEnterを押す
  - print(a)で変数aの中身を表示してみましょう
    - 変数aに代入したリストが表示されます
  - print(a)の代わりに，aと打ってEnterを押すだけでも変数aの中身を確認できる

```
akiyama@:~/compphys2/lecture7$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = [1, 2, 3]
>>> print(a)
[1, 2, 3]
>>> a
[1, 2, 3]
```

リストの要素にアクセスする

- リストaのn番目の要素は，a[n]と書くことでアクセスできる
  - n([ ]内の数字)のことをインデックス(index)や添字と呼ぶ
  - Pythonの添字は0始まり
    - 配列の添字が1で始まる言語もあるので，他の言語を使う際には気をつけること
- a[0]と打ってEnterを押してみよう
  - リストの最初の要素である1が表示されればOK

```
>>> a[0]
1
```

リストの要素を指定して値を代入する

・インデックスでリストの要素を指定することで，特定の要素を変更することができる

・リストaの2番目の要素に文字列“A”を代入してみよう

・代入後のリストを表示し，2番目の要素が文字列に変わったことを確認しよう

```
>>> a[1] = "A"
>>> a
[1, 'A', 3]
>>>
```

リストのリスト

・リストの要素にリストを代入することもできる

・新しく変数bを使って，以下のようなリストを含んだリストを作ろう

```
>>> b = [[1,2],[3,4,5],6,7]
>>> b
[[1, 2], [3, 4, 5], 6, 7]
```

・リストに含まれたリストの要素には，添字の複数指定によってアクセスできる

・bの最初の要素(リスト)を見てみよう

・bの最初の要素(リスト)の2番目の要素を見てみよう

```
>>> b[0]
[1, 2]
>>> b[0][1]
2
>>>
```

添字の範囲外にアクセスしようとすると…

- 先ほど作ったリストaは3つの要素しかない
  - 添字の範囲は0, 1, 2である
  - 試しにa[4]と打ってEnterを押してみよう

```
>>> a[4]
Traceback (most recent call last):
  File "<python-input-13>", line 1, in <module>
    a[4]
    ~^^^
IndexError: list index out of range
```

- 添字の範囲外にアクセスしようとするとErrorになるので注意
  - 典型的なバグの原因

リストの長さの確認

・リストの長さはlen()という関数から取得することができる

・リストaの長さを取得しよう

・リストbの長さを取得しよう

・リストb[0]の長さを取得しよう

```
>>> len(a)
3
>>> len(b)
4
>>> len(b[0])
2
```

・これらのリストを改めて表示し，リストの長さが正しく得られていることを確認しよう

リストの統合

・二つのリストは+で統合できる

　・右のように，変数a, bに新しいリストをそれぞれ代入する

　・a+bを実行しよう

　・b+aを実行しよう

　・統合のされ方の違いに注意しよう

```
>>> a = [1,2]
>>> b = [3,4,5]
>>> a+b
[1, 2, 3, 4, 5]
>>> b+a
[3, 4, 5, 1, 2]
```

リストへの要素の追加

- 要素を追加するには，appendを使う
  - 例えば，リストaに整数3を追加したい場合は以下のように書く

```
>>> a.append(3)
```

  - 整数3がリストaに追加されたことを確認しよう

```
>>> a
[1, 2, 3]
```

空のリスト

・要素のないリストも作ることができる

・下のように，変数aに空のリストを代入し，aの中身を確認してみよう

```
>>> a=[]
>>> a
[]
```

リストをappendする

・空のリストaに，リストbをappendしてみよう

　・aの中身を確認し，aの最初の要素がリストbになったことを確認しよう

```
>>> a.append(b)
>>> a
[[3, 4, 5]]
```

リストをappendする時の注意

・リストaにもう一度リストbをappendしてみよう

・aの中身を確認しよう

```
>>> a.append(b)
>>> a
[[3, 4, 5], [3, 4, 5]]
```

・ここで，以下のように要素a[0][0]を更新してみよう

・aの中身を確認しよう

```
>>> a[0][0]=100
>>> a
[[100, 4, 5], [100, 4, 5]]
```

・aに含まれる二つのリストは同じものなので，一方を更新するともう一方も更新される
（a[0][0]を変更するとa[1][0]も更新される）
→ 資料後半の「メモリ上でのリストの表現と参照の値渡し」で詳しく見る

リストに特定の要素があるかどうかを調べる

・リストに特定の要素があるかどうかを調べるためには，inを使う

・今一度，aとbの中身を確認しておこう

```
>>> a
[[100, 4, 5], [100, 4, 5]]
>>> b
[100, 4, 5]
```

・inを使うとTrue/Falseが返ってくる

・以下の四つを実行してみよう

```
>>> 100 in b
True
>>> 1 in b
False
>>> 100 in a
False
>>> [100,4,5] in a
True
```

・ここまでできたら，exit()で対話モードを一旦終了しましょう

リストの要素に対して繰り返し処理を行う

- for文とinを使うことで，リストの要素を順番に取り出し，全ての要素について何らかの処理を行うことができる
  - ex_pythonという名前のディレクトリを作り，その中にex_list.pyというスクリプトを作りましょう
    - ex_list.pyには右のような三行からなるコードを書きましょう

```
home > akiyama > compphys2 > lecture7 > ex_python > ex_list.py
1   a = ["A", "B", "C"]
2   for i in a:
3       print(i)
```

  - ex_list.pyを実行しましょう
    - 下のような結果が得られればOK

```
akiyama@:~/compphys2/lecture7/ex_python$ python3 ex_list.py
A
B
C
```

リストの要素の値と添字の両方に対して繰り返し処理を行う

- リストaに対してenumerate(a)とすると，要素と添字の情報をペアで取得できる
  - 右のように，ex_list.pyに書き足そう
  - ex_list.pyを実行しよう
    - 以下の出力が得られればOK
    - コード5行目のiで添字，xで要素を取得したことが分かるだろう

home > akiyama > compphys2 > lecture7 > ex_python > ex_list.py

```
1  a = ["A", "B", "C"]
2  for i in a:
3      print(i)
4
5  for i, x in enumerate(a):
6      print(i, x)
```

```
akiyama@:~/compphys2/lecture7/ex_python$ python3 ex_list.py
A
B
C
0 A
1 B
2 C
```

タプルとは?

・Pythonでは，タプル(tuple)というデータ構造もよく使われる

　・タプルとリストは似ているが，一度作成されたタプルは修正できないという差異がある

・タプルはカンマで区切ることで定義する

　・紛らわしいときには() (丸括弧)で囲むことがある

　例1) 1, 2, 3 → 整数1, 2, 3からなるタプル

　例2) (1, 2, 3) → 整数1, 2, 3からなるタプル

タプルを作ってみよう

- 再び，Python3を対話モードで起動しましょう
- 変数aに整数1, 2, 3からなるタプルを代入します
  - 変数aの中身を確認しましょう
- リストとの共通点
  - インデックスで特定の要素を取得できる
  - lenでタプルの長さを取得できる
  - +で二つのタプルを統合できる

```
>>> a = 1, 2, 3
>>> a
(1, 2, 3)
>>> a[0]
1
>>> len(a)
3
>>> b = 4, 5
>>> a+b
(1, 2, 3, 4, 5)
```

タプルとリストの相違点

- 一度作成されたタプルは修正できない
  - 要素は更新できない
  - appendできない

```
>>> a[0]=4
Traceback (most recent call last):
  File "<python-input-20>", line 1, in <module>
    a[0]=4
    ~^^^
TypeError: 'tuple' object does not support item assignment
>>> a.append(4)
Traceback (most recent call last):
  File "<python-input-21>", line 1, in <module>
    a.append(4)
    ^^^^^^^^
AttributeError: 'tuple' object has no attribute 'append'
```

タプルによる複数の変数への代入

・以下のようにタプルを使うことで変数a, bへの代入を一度に行える

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
```

・以下のようにタプルを使うことで変数a, bの値を交換することもできる

```
>>> a, b = b, a
>>> a
2
>>> b
1
```

・ここまでできたらexit()で対話モードを終了しましょう

練習問題

- 空のリストを用意しappendを使って，1から100までの100個の整数からなるリストを作るコードを書け

- 空のリストを用意しappendを使って，1から1000までの間に存在する全ての奇数からなるリストを作るコードを書け

- Pythonの組み込み関数sum(), min(), max()を使って，これらのリストの要素の和, 最大値, 最小値を求めてみよう

  - これらの組み込み関数の引数にリストを渡せばよい

    例)リストmyListに含まれる全要素の和はsum(myList)

  - 結果が妥当かどうか確認しましょう(解答例は/lecture7/src/sol_practice.py)

辞書とは?

- Pythonのリストは, 「数字」と「要素」を結びつけるデータ構造になっていた
- Pythonの辞書は, 「数字に限らないもの」と「要素」を結びつけるデータ構造を与える
  - リスト ⇒ インデックス(数字)で要素にアクセスする
  - 辞書 ⇒ キー(key)で要素にアクセスする
    - 数字, 文字列, タプル, …などををキーにすることができる
    - リスト同様, 何でも要素にすることができる

辞書を作ってみよう 1/3

- 辞書は{}で初期化する
- 例えば，「文字列」と「整数」を結びつけてみましょう
- Pythonを対話モードで開きましょう
  - dを辞書型として初期化し，“Apple”という文字列をキー，123という整数を要素として結びつけましょう
  - 結びつけた要素(123)は，キー(Apple)を入力することで取得できます
    - print文を実行して確認しましょう

```
akiyama@:~/compphys2/lecture7/ex_python$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> d = {}
>>> d["Apple"] = 123
>>> print(d["Apple"])
123
```

辞書を作ってみよう 2/3

- 辞書の初期化時にキーと要素を指定することもできる
  - 指定方法は「キー：要素」
- ex_pythonの中に, ex1_dict.pyというファイルを作り,右のコードを書いてみよう
  - 5行目以下がプログラムのメイン部分
    - 三種類の辞書を作っている
      - 何でもキー, 要素にできる
  - 関数「key_and_elt」は渡された辞書型のキーと要素をprintする関数
    - 辞書に対してfor文を回すと, キーを取得できる

home > akiyama > compphys2 > lecture7 > ex_python > ex1_dict.py

```
1  def key_and_elt(d):
2      for key in d:
3          print(key, ":", d[key])
4
5  if __name__ == '__main__':
6
7      d1 = {"Apple":123, "Banana":456, "Cherry":789}
8      d2 = {1:"One", 2:"Two", 3:"Three"}
9      d3 = {(1,2):3, (2,3):5, (3,4):7}
10
11     key_and_elt(d1)
```

辞書を作ってみよう 3/3

```
akiyama@:~/compphys2/lecture7/ex_python$ python3 ex1_dict.py
Apple : 123
Banana : 456
Cherry : 789
```

- ex1_dict.pyを実行してみよう
  - 右のような結果が見えたらOK
- ex1_dict.pyを編集し, d2, d3のキーと要素の対応も確認しよう
  - 11行目のように, 関数「key_and_elt」にd2, d3を渡せばよい
- なお, items()というメソッドを使って, キーと要素を同時に両方取得することも可能
  - ex1_dict.pyをex2_dict.pyの名前でcpし, ex2_dict.pyを右のように編集しよう
    - 変更箇所は2, 3行目だけ
  - ex2_dict.pyも実行してみよう (出力結果は変わりません)

home > akiyama > compphys2 > lecture7 > ex_python > ex2_dict.py

```
1   def key_and_elt(d):
2       for key, elt in d.items():
3           print(key, ":", elt)
4
5   if __name__ == '__main__':
6
7       d1 = {"Apple":123, "Banana":456, "Cherry":789}
8       d2 = {1:"One", 2:"Two", 3:"Three"}
9       d3 = {(1,2):3, (2,3):5, (3,4):7}
10
11      key_and_elt(d1)
```

存在しないキーを指定するとエラーになる 1/2

- Pythonを対話モードで開き, 以下のように空の辞書を作った後, 存在しないキーを入力してみよう
  - KeyErrorとなる

```
akiyama@:~/compphys2/lecture7/ex_python$ python3
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> d = {}
>>> print(d["Apple"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Apple'
```

存在しないキーを指定するとエラーになる 2/2

- 応用上, 存在しないキーを指定した時に, デフォルトの値があると便利なことがある
  - defaultdictで辞書型を初期化すると便利
    - d = defaultdict(int)のように辞書型dを初期化しておくことで, 存在しないキーが参照された時に, 自動で0を返し, そのキーが辞書に登録される
- この機能を使うことで, 辞書型を応用した頻度カウンターを実装することができる
  - 次頁へ

辞書型を応用した頻度カウンター

- ex3_dict.pyというファイルを作り，右のようなコードを書いてみよう
  - 1行目はdefaultdictを使えるようにするためのimport文
  - 9行目で辞書(freq)を初期化
  - 11行目のリスト(data)の中にある文字の出現頻度をカウントする
  - defaultdictで初期化したことで，14行目のように書いてもKeyErrorにならず，xがキー，xの出現回数の累積がその要素として辞書に登録されていく

home > akiyama > compphys2 > lecture7 > ex_python > ex3_dict.py

```
1  from collections import defaultdict
2
3  def key_and_elt(d):
4      for key, elt in d.items():
5          print(key, ":", elt)
6
7  if __name__ == '__main__':
8
9      freq = defaultdict(int)
10
11     data = ["a", "b", "b", "b", "b", "a", "c", "b"]
12
13     for x in data:
14         freq[x] += 1
15
16     key_and_elt(freq)
17
```

- ex3_dict.pyを実行してみよう
  - 結果は右の通り

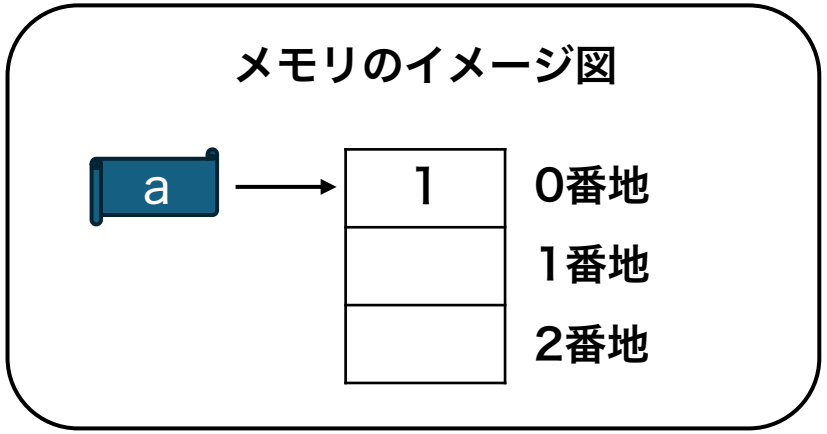
```
akiyama@:~/compphys2/lecture7/ex_python$ python3 ex3_dict.py
a : 2
b : 5
c : 1
```

メモリ上でのリストの表現と参照の値渡し

- 以下の演習資料を読み進め，途中にある三つの演習A, B, Cに取り組んでください
- 以下の内容を完全に理解する必要はありませんが，知っていないと思わぬバグに陥るので，頭の片隅に置いておくことをおすすめします

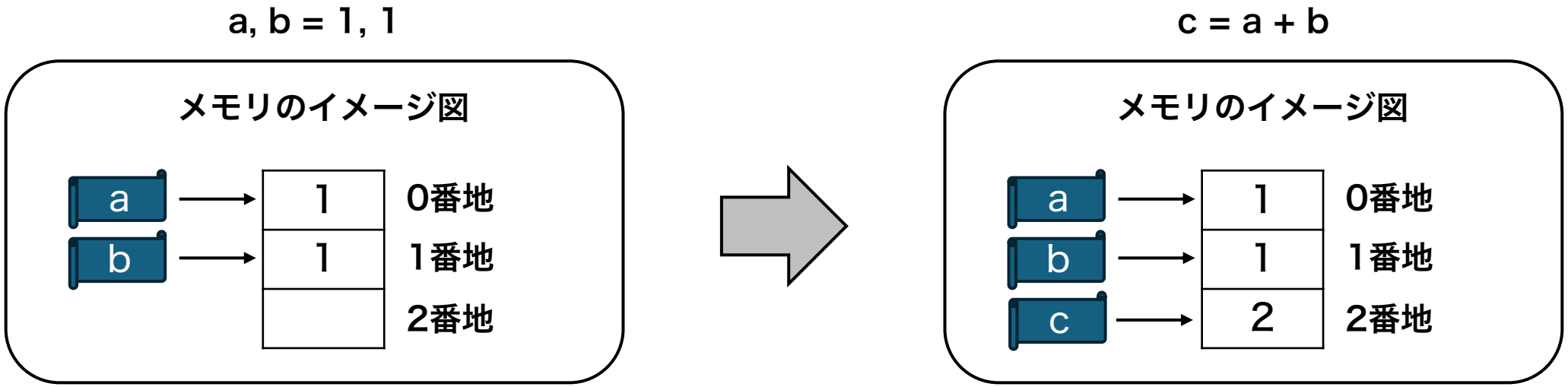
変数とはメモリ上のラベルである

- 第５講で「変数とは計算機上で再利用可能なようにラベルされた値のこと」と言いました
- 計算機とは，メモリにあるデータをCPUで処理し，その結果を再びメモリに戻す機械
  - メモリには「番地」という通し番号がついている
- a = 1というプログラムを考える
  - プログラムの意味：
    aという変数を用意し，そこに1という値を代入する
  - 実際に計算機で行われる処理：
    メモリ上にaというラベルを貼り，
    そこに1を書き込む
    （右のイメージ図参照）



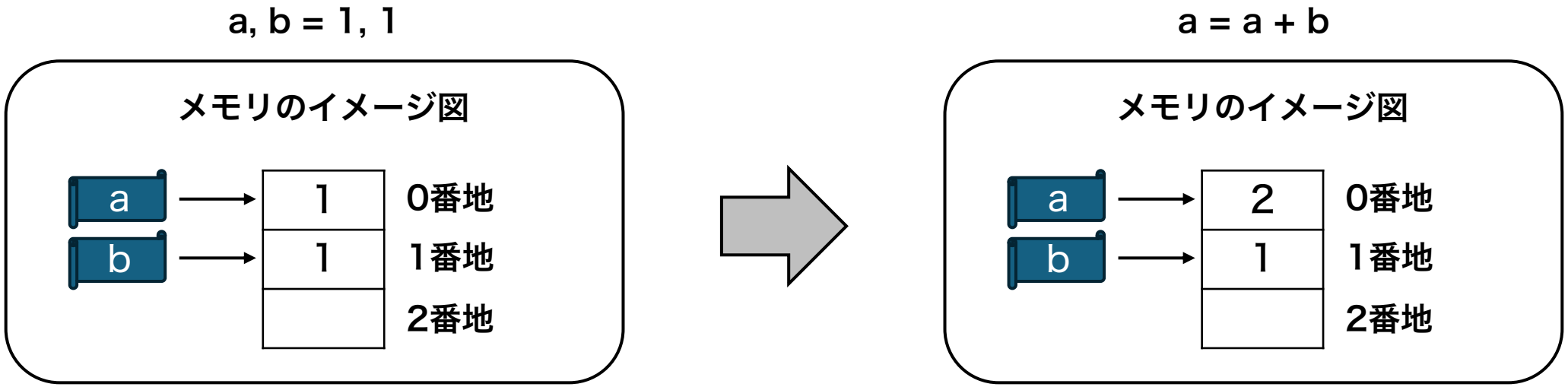
変数による足し算 1/2

- 変数a, bに1を代入し，c = a+bというプログラムを考える
- 実際に計算機で行われる処理は以下の通り
  - まずa+bの計算が行われる
  - 変数cでラベルされるメモリ上の番地が準備される
  - a+bの計算結果が変数cでラベルされたメモリ上の番地へ書き込まれる



変数による足し算 2/2

- 変数a, bに1を代入し，a = a+bというプログラムを考える
- 実際に計算機で行われる処理は以下の通り
  - まずa+bの計算が行われる
  - 変数aはすでに0番地をラベルしているので，0番地の値がa+bの計算結果で更新される



型とはメモリ上の数値に対する解釈ルールである 1/2

- 計算機のメモリには整数しか保存できない
- 「メモリに保存された整数をどのように解釈するか」を指定するのが型
- a = “test”というプログラムを考える
  - 変数aには“test”というstr型の文字が代入されている
  - ASCIIコードにより，tは0x74, eは0x65, sは0x73という数値としてメモリ上で表現される
    - 0xは16進数という意味
  - str型により，aが指す先のメモリ上の数値は文字列として解釈される
    - aが指す74, 65, 74, 73という数字の列はtestという文字列として解釈される

型とはメモリ上の数値に対する解釈ルールである 2/2

- 浮動小数点数も同様
- a = 1.0というプログラムを考える
  - 変数aには1.0というfloat型の文字が代入されている
  - IEEE754という規格に従い，1.0という浮動小数点数は8バイトの数字としてメモリ上で表現される
  - float型により，aが指す先のメモリ上の数値は浮動小数点数として解釈される
    - aが指す8バイトの数字は1.0という浮動小数点数として解釈される

メモリ上でのリストの表現

- リストは複数の要素から構成される
  - リストを表すラベルは「リストの先頭を指す場所」を指すという仕様になっている
  - リストのラベルは「リストの先頭そのもの」を指しているのではない
- a = [1, 2, 3]というプログラムを例に，実際に計算機で行われる処理は以下の通り
  - まず，メモリ上に[1, 2, 3]というリストを作成する（図1）
  - 次に，その先頭の番地を指す場所を作成し，その場所にaというラベルを貼る（図2）

図1

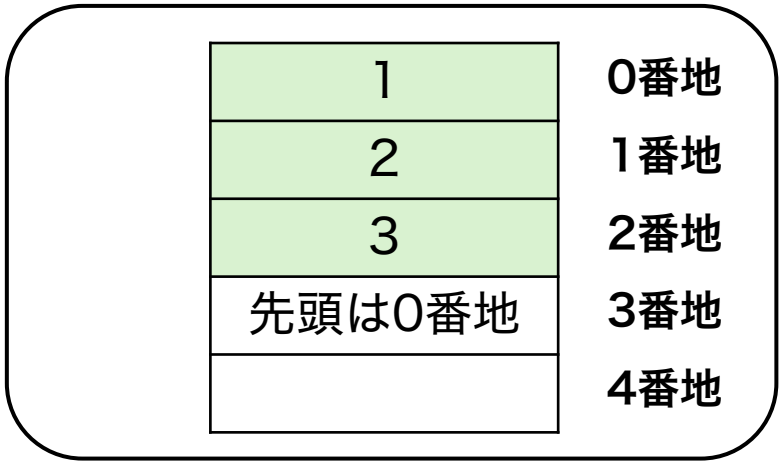
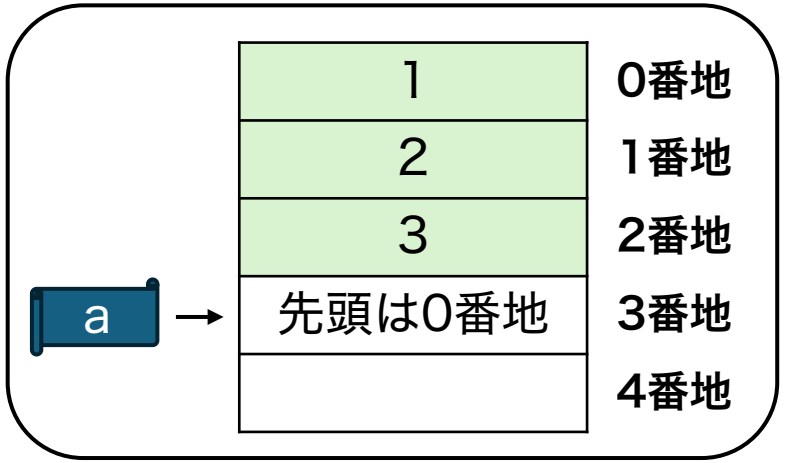


図2

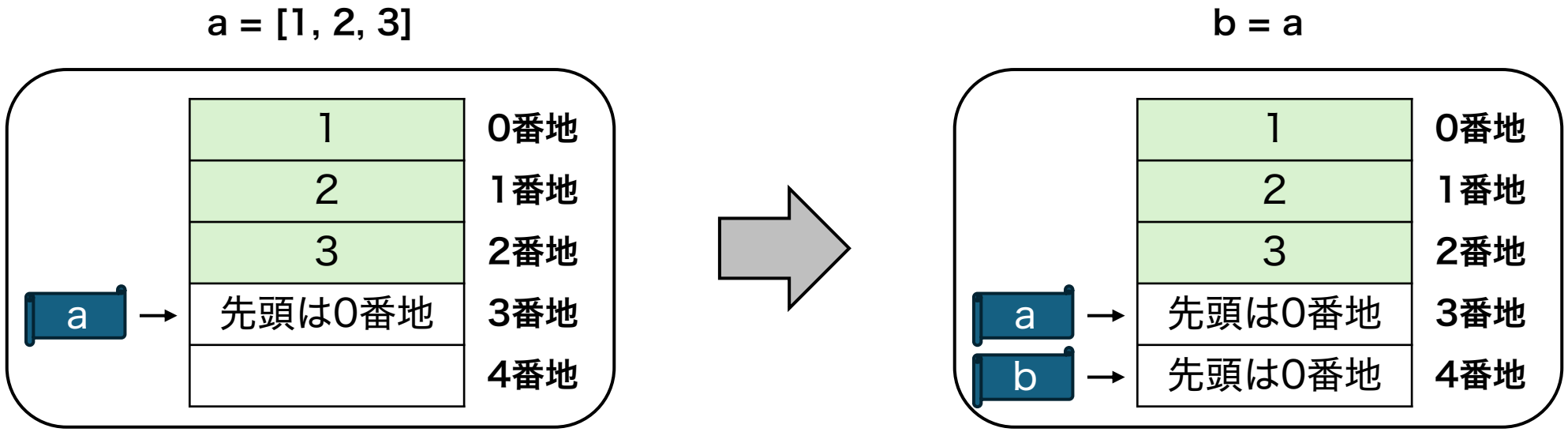


リストのコピー 1/4

・[1, 2, 3]というリストをラベルする変数aを新しい変数bにコピーすることを考える

・aの指す内容はリストそのものではなく「リストの先頭を指す場所」

・そのため，b=aというプログラムは以下のような処理になる



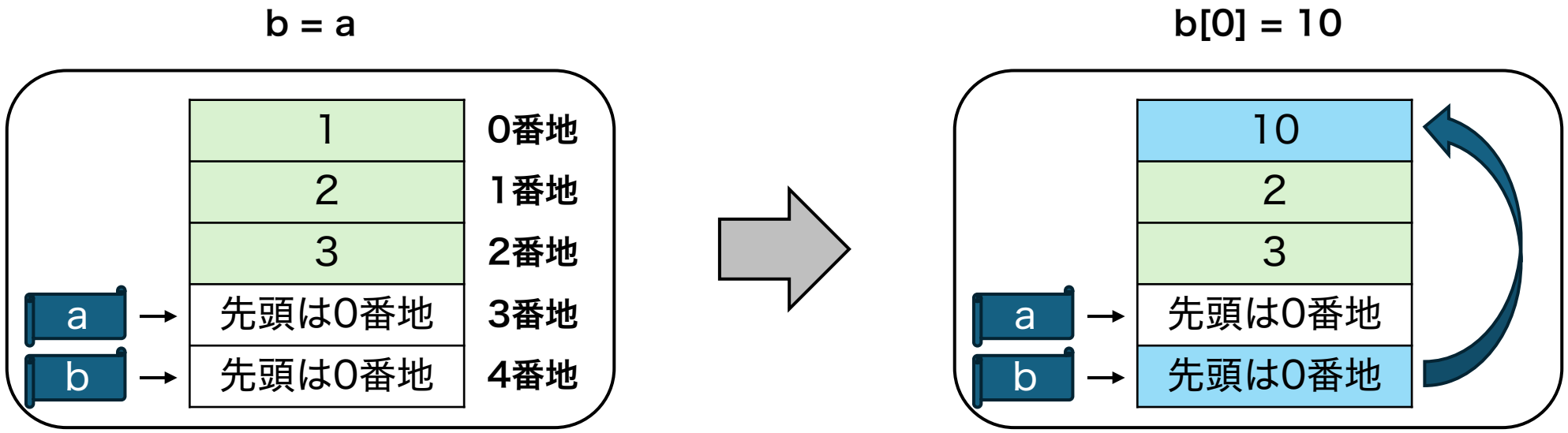
リストのコピー 2/4

・この状態でb[0]=10のようにリストを修正することを考える

・bの指す内容はリストそのものではなく「リストの先頭を指す場所」

・したがって，コピー元のaが指すリストも修正されることになる

・これがp.18のような挙動を生む理由



リストのコピー 3/4

- ここまでの処理で，a = [10, 2, 3]，b = [10, 2, 3]となっている
  - a, bは同じリストを指すラベルになっている
- この状態で，bに異なるリスト[4, 5, 6]を代入したとする
  - b = [4, 5, 6]
    - まず，メモリ上に[4, 5, 6]が作られる（次項の図1）
    - 次に，bが新しく作られたリストの先頭の場所を指すようになる（次項の図2）
- この代入以後は，aとbは異なるリストを指すラベルになる
- bを修正してもaは修正されない（aとbは無関係になる）

リストのコピー 4/4

・b = [4, 5, 6]による処理のイメージ図は以下の通り

図1

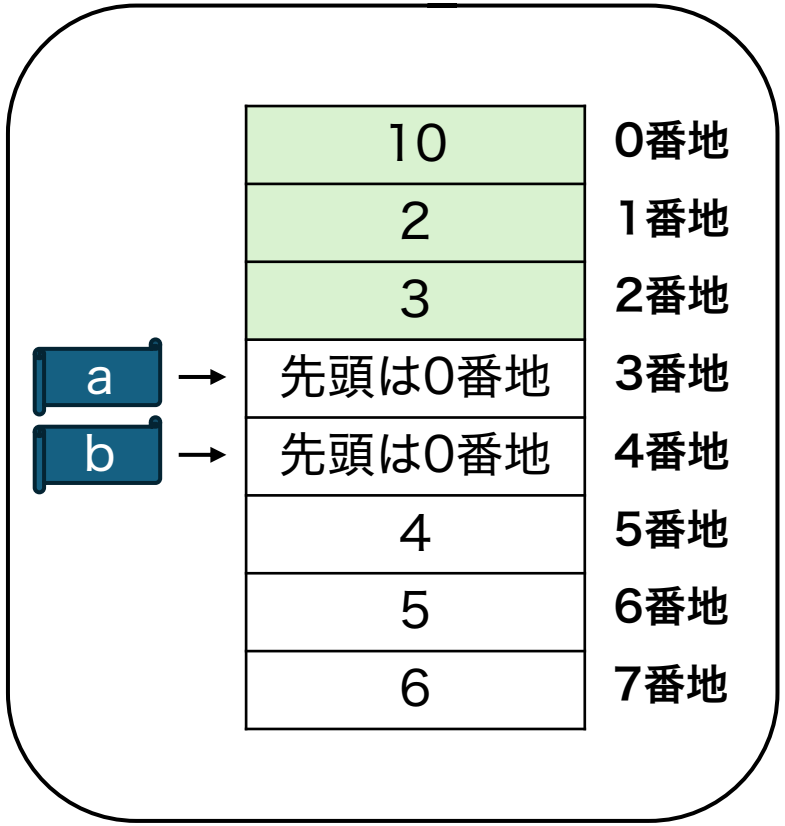
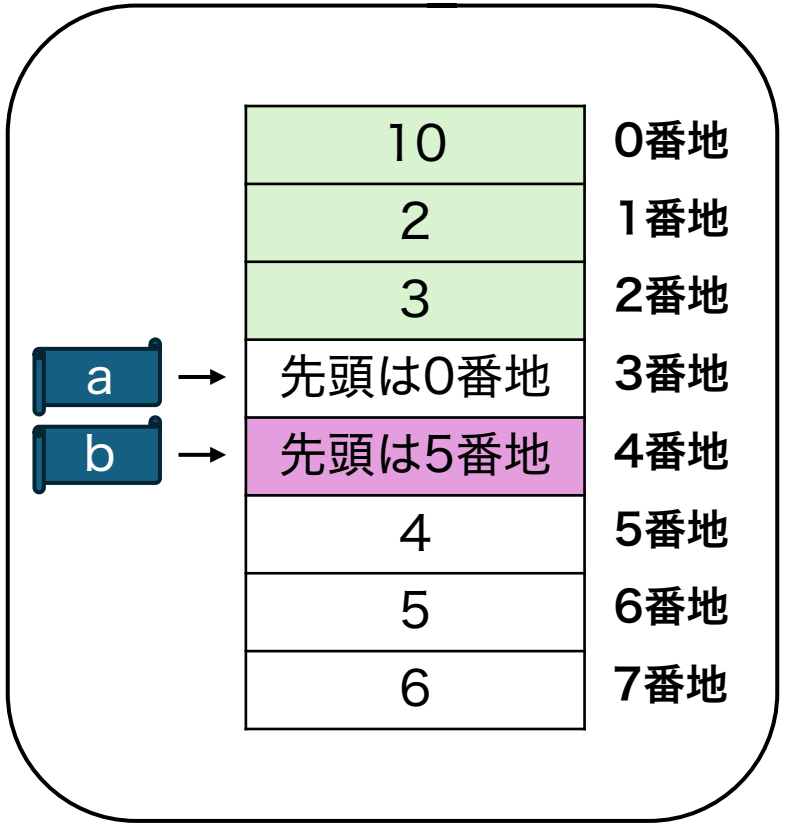


図2



参照

- リストを指す変数は，リストそのものではなく「リストの先頭の場所」を指している
- このように，値そのものではなく，「この場所を見よ」というような情報のことを参照(reference)と呼ぶ

関数の引数としての変数 1/2

- 第5講で見たように，関数内で宣言された変数はローカル変数と呼ばれる
  - ローカル変数はローカルスコープ(関数が作るコードブロック)内でのみ有効である
- 関数の引数もローカル変数である
- 例として，右のプログラムを考える
  - 6行目の出力結果は1になる

```
1  def func(x):
2      x = 2
3
4  x = 1
5  func(x)
6  print(x)
```

演習A

- 同じプログラムを書いて結果を確認しておきましょう

関数の引数としての変数 2/2

- このプログラムを実行した時のメモリ上での処理の流れは以下の通り
  - まず，グローバル変数xに1が代入される(図1)
  - 次に，関数内のローカル変数xが作られ，funcの引数の値がコピーされ，ローカル変数xに渡される(図2)
  - 続いて，ローカル変数xに2が代入される(図3)

```
1    def func(x):
2        x = 2
3
4    x = 1
5    func(x)
6    print(x)
```

図1

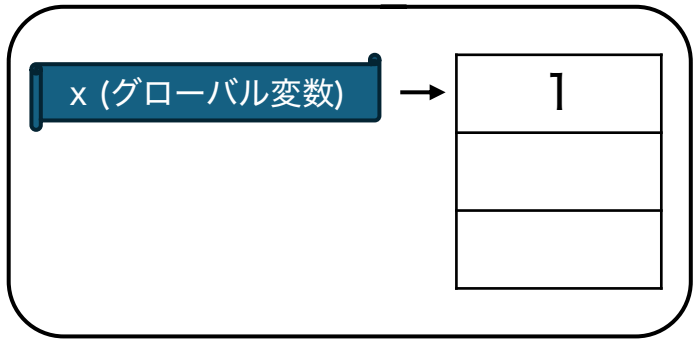


図2

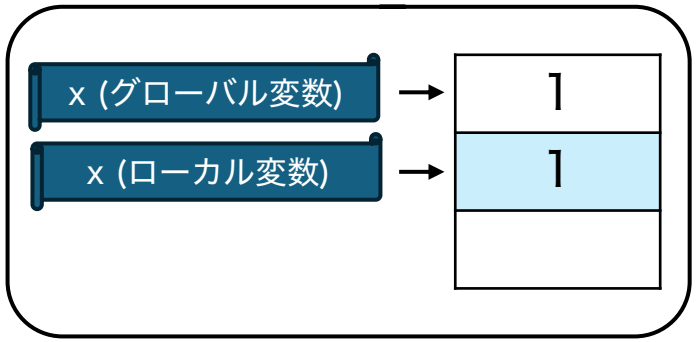
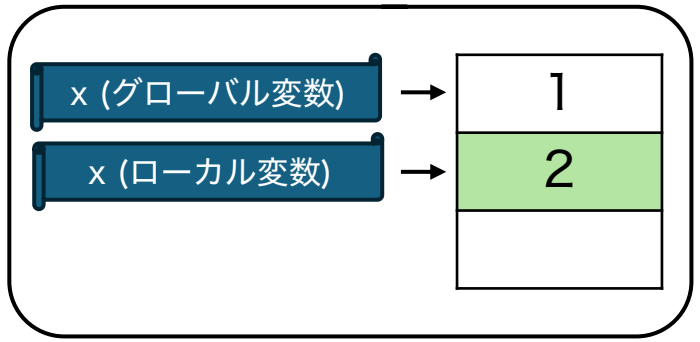


図3



- このように，関数の引数に値をコピーして渡す方法を値渡し(call by value)と呼ぶ

コーディング時の注意

- 先の例のように，関数の引数としてグローバル変数と同じ名前を使うのは避ける
  - バグの原因になります
- 関数の引数，あるいは関数の内部でグローバル変数と同じ名前のローカル変数を宣言しても，グローバル変数とは全く異なる変数として扱われる
  - このことを知らずにコードを書くとバグの原因になります
- 関数の引数は値がコピーされてから関数内のローカル変数に渡される(値渡し)
  - このことを踏まえて，次頁から関数の引数としてリストを渡すことを考えてみよう

関数の引数としてリストを渡す

```
1    def func(myList):
2        myList[1] = 100
3
4    myList = [1, 2, 3]
5    func(myList)
6    print(myList)
```

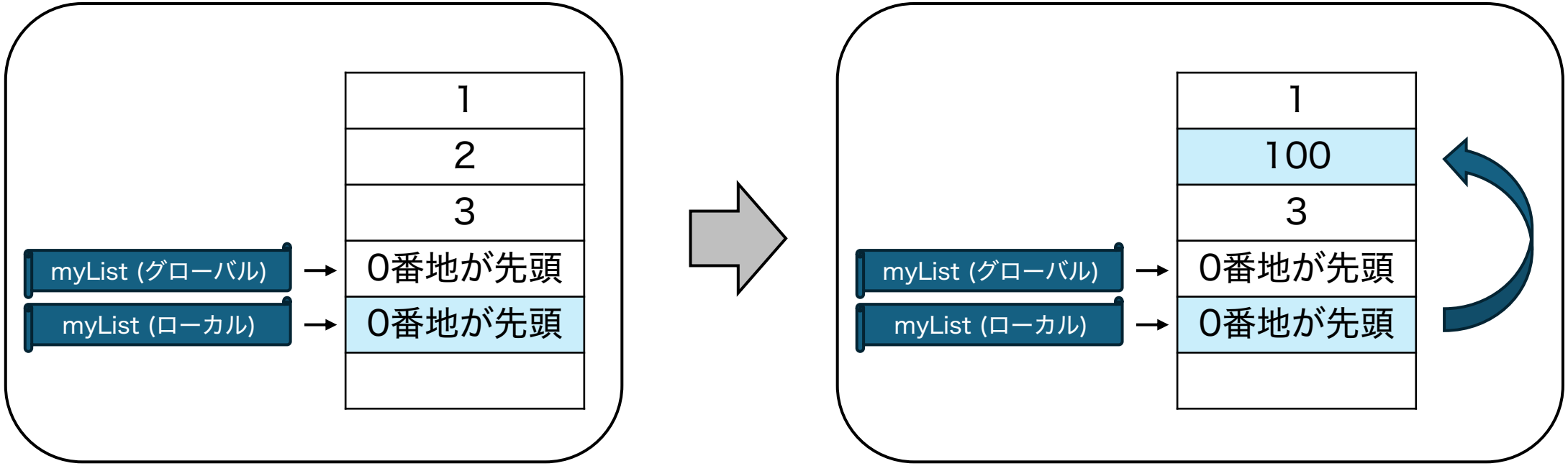
・右のようなリストを引数にする関数funcを書いた
  ・6行目の出力結果は?
・コードを書く前に結果を予想してみよう
  ・ポイントは以下の二点
    ・リストを指す変数は，リストそのものではなく「リストの先頭の場所」を指している
    ・関数の引数は値がコピーされてから関数内のローカル変数に渡される

演習B

・結果を予想したら，同じプログラムを書いて出力結果を確認してみよう

出力結果は次のように理解できる

- リストを指す変数は，リストそのものではなく「リストの先頭の場所」を指している
- 関数の引数は値がコピーされてから関数内のローカル変数に渡される(値渡し)



- リストを指す変数を関数に渡す場合，「リストの先頭の場所」という参照が値としてコピーされて関数内部のローカル変数に渡されるので参照の値渡し(call by sharing)と呼ばれる

クイズ

・次のコードを実行した場合，6行目の出力結果は?

```
1    def func(myList):
2        myList = ["A", "B", "C"]
3
4    myList = [1, 2, 3]
5    func(myList)
6    print(myList)
```

演習C

・6行目の出力結果を予想した上で，同じプログラムを書いて結果を確認してみましょう