Data structures

- In codes, there might be different types of variables, and sometimes it is more convenient to divide them into different groups. In python, there are many ways to group multiple values together in a collection.
- The common ways are **list**, **tuple**, **set**, **range** and **dictionary**.
- Here are some examples.

```
# a list of strings
month_names = ['Jan', 'Feb', 'Mar', 'Apr']

# a list of integers
numbers = [1, 7, 4, 10, 162]

# an empty list
my_list = []

# a list of variables we defined somewhere else
things = [var1, var2, var3]
```

List

- **List**, is a list of quantities, one after another. The quantities in a list are called its elements, and they do not have to be the same type.

- For a list, r =[1, 7, 10, 15]. The individual elements are denoted r[0], r[1], r[2], and so forth. Crucially, the numbers start from ***zero***, not one.

- We can have high dimensional list with list as the elements, however, there might be some unexpected errors. BE CAREFUL!

- A special functions for lists is the function ***map***, which allows you apply ordinary functions to all the elements of a list at once.

Claim or Initialize a list

- By hand

A=[[None, None], [None, None], None]

- Use other lists

A=B; A=copy(B); A=B.copy()

A=B+C

This could give you serious mistakes. BE CAREFUL!

- Use multiplication operator (mean copy)

A=[None]*2; A=[[None]*2]*3

This could give you serious mistakes. BE CAREFUL!

- Use **for** loops

A =[None for x in range(n)]

A =[[None for x in range(n)] for y in range(m)]

A =[[[None for x in range(n)] for y in range(m)] for z in range(p)]

Example and Practice: 1D list

- T=[0,1,2,3,4]

B=T
C=T.copy()
D=T[:]

# 1. If we set T[0:5]=[1,1,1,1,1]
B=?
C=?
D=?

# 2. if we set T=[1,1,1,1,1]??
B=?
C=?
D=?

# 3. if we set B[0:5]=[1,1,1,1,1]
T=?
C=?
D=?

# 4. if we set C[0:5]=[1,1,1,1,1]
T=?
B=?
D=?

# 5. if we set D[0:5]=[1,1,1,1,1]
T=?
B=?
C=?

Why??

Example and Practice: 2D list

- T2=[0,1,2,3,4]

B2=[None,None]; B2[0]=T2; B2[1]=T2.copy(); B2[1]=T2;
B2.append(T2)
C2=B2.copy(); D2=B2[:]; E2=B2

#1. If we set T2[0:5]=[1,1,1,1,1]; B2[0][1]='*'; B2[1][0]='#'
T2=?; B2=?; C2=?; D2=?; E2=?

#2. If we set T2=[1,1,1,1,1]; B2[0][1]='*'; B2[1][0]='#'
T2=?; B2=?; C2=?; D2=?; E2=?

#3. If we set T2[0:5]=[1,1,1,1,1]; C2[0][1]='*'; C2[1][0]='#'
T2=?; B2=?; C2=?; D2=?; E2=?

Why??

Example and Practice: use multiplication

#1. A=[0, 0,0]; A[0]=1;
A=?

#2. A=[0]*3; A[0]=1;
A=?

#3. A=[[0,0,0], [0,0,0]]; A[0][0]=1;
A=?

#4. A=[[0]*3]*2; A[0][0]=1;
A=?

#5. A=[[0]*3]*2; A[0][1:2]=[1,2];
A=?

#6. A=[[0]*3]*2; A[0]=1;
A=?

Why??

A good tool/function id()

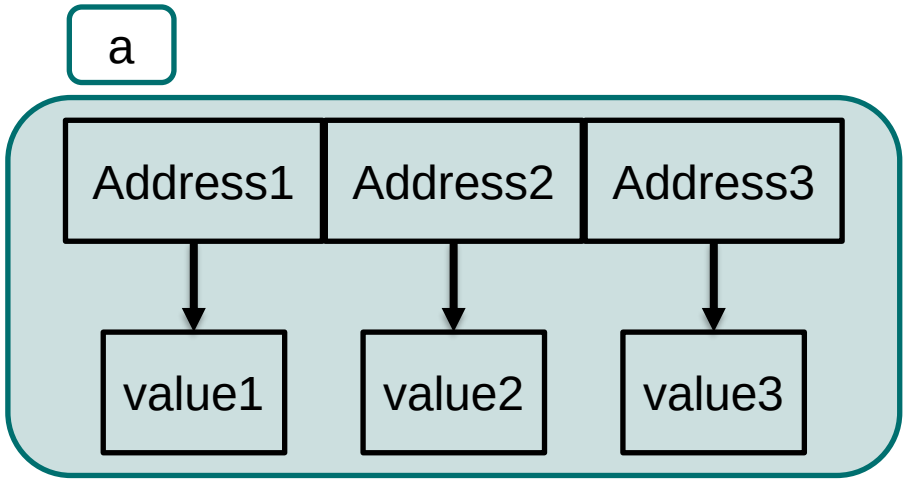
- id(object): return the address of the object in the memory. The object could be anything. Is it a fixed number?

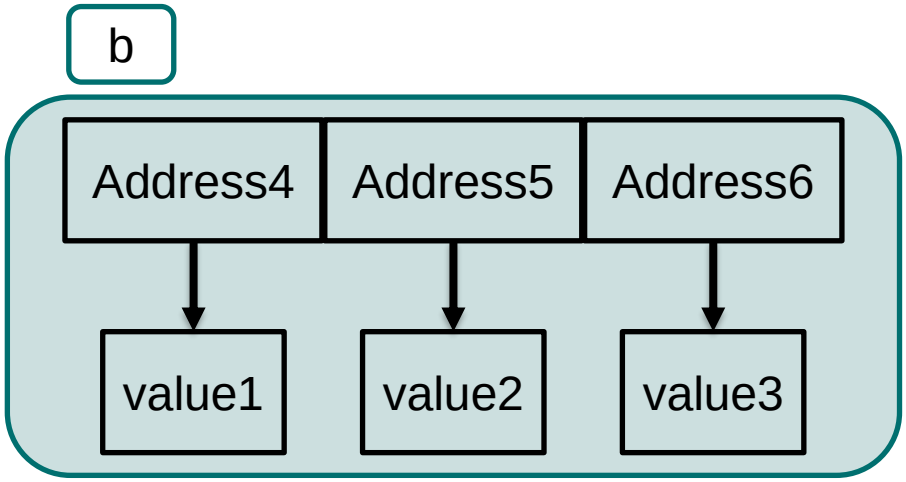
a=1
b=1
Q1: id(a)=?
Q2: id(b)=?
Q3: id(1)=?

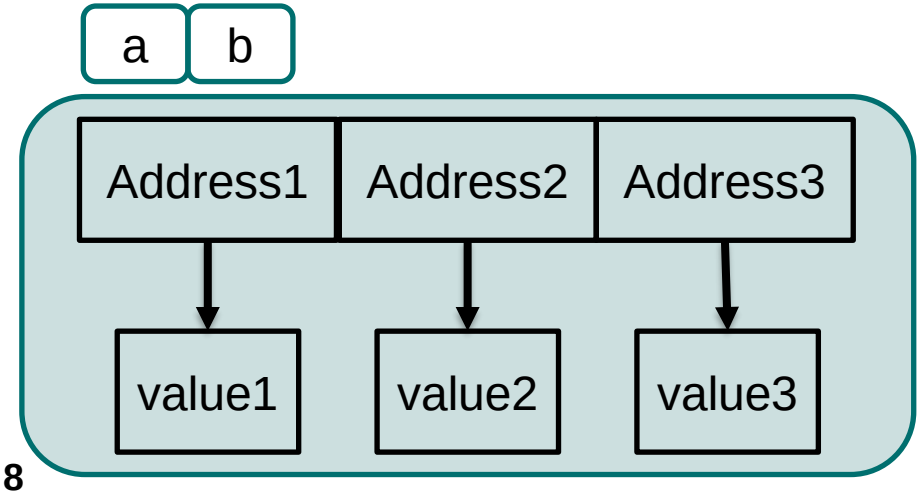
a=[1]
b=[1]
Q4: id(a)=?
Q5: id(b)=?
Q6: id(a[0])
Q7: id(b[0])

Shallow copy and deep copy

- All the confusions come from the differences between shallow copy and deep copy. They might have different names like value copy(值复制) and address copy(位复制).







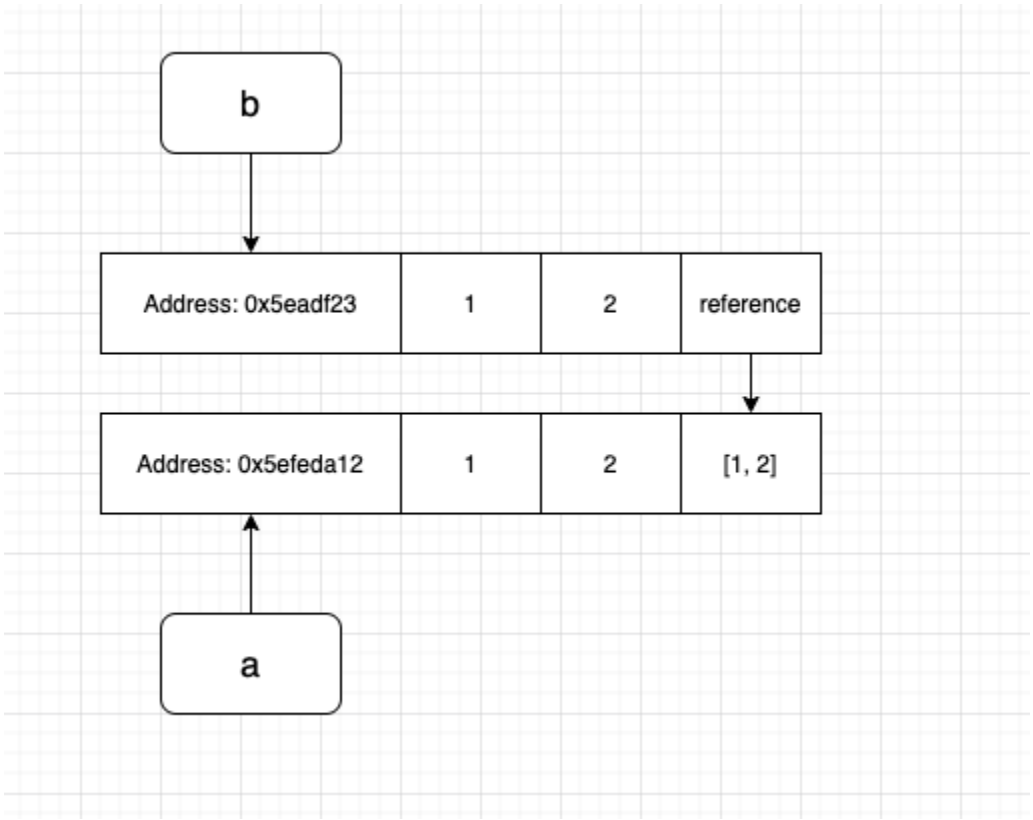
- copy.copy() and copy.deepcopy()
- Be careful and do you own test before you use it.

Shallow copy

- The problem with shallow copy is it does not copy a nested object, in this case, it is [1, 2] which is nested in the original list.

```
import copy
a = [1, 2, [1, 2]]
b = copy.copy(a)
b[2][0] = 11

print("list a:", a)
print("list b:", b)
```



Check the size of memory used

```
###### test list size
import sys
import matplotlib.pyplot as plt

a=1
print(a.__sizeof__())
print(sys.getsizeof(1))

a=[1]
print(a.__sizeof__())
print(sys.getsizeof(a))

a=['sjdfosajfsaodfsaojdfsjaodfjasdfjso']
print(sys.getsizeof(a))
# sys.getsizeof only take account of the list itself, not items it contains.

list_size=[]
size_l=[]
mem_l=[]
size_l.append(len(list_size))
mem_l.append(sys.getsizeof(list_size))
print(sys.getsizeof(list_size))
for n in range(100):
    list_size.append(n)
    print(list_size)
    print(id(list_size),sys.getsizeof(list_size))
    size_l.append(len(list_size))
    mem_l.append(sys.getsizeof(list_size))

plt.plot(size_l,mem_l,'*-')
```

More List methods and functions

- **len(list)** # the length of a list
- **sum(numbers)** # the sum of a list of numbers
- **any([1,0,1,0,1])** # are any of these values true?
- **all([1,0,1,0,1])** # are all of these values true?
- **list.append(5)** #add an element to the end:
- **list.count(5)** #count how many times a value appears:
- **list.extend([56, 2, 12])** # append several values at once
- **list.index(3)** # find the index of a value, if the value appears more than once, we will get the index of the first one; if the value is not in the list, we will get a ValueError!
- **list.insert(0, 45)** # insert 45 at the beginning of the list
- **my_number = numbers.pop(0)** # remove an element by its index and assign it to a variable
- **numbers.remove(12)** # remove an element by its value, if the value appears more than once, only the first one will be removed

Tuple

- Python has another sequence type which is called tuple. Tuples are similar to lists in many ways, but they are immutable. We define a tuple literal by putting a comma-separated list of values inside round brackets ().
- We can use tuples in much the same way as we use lists, except that **we can't modify them**. Tuple is used to create a sequence of values that we don't want to modify.

```
WEEKDAYS = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
```

Set

- Python uses ***set*** to collect **unique** elements. If we add multiple copies of the same element to a set, the duplicates will be eliminated.
- To define a set, we need to put a comma-separated list of values inside curly brackets {}, and we can perform various operations on the sets.
- It is important to note that unlike lists and tuples sets are **not ordered**. When we print a set, the order of the elements will be random. If we want to process the contents of a set in a particular order, we will first need to convert it to a list or tuple and sort it.

```
even_numbers = {2, 4, 6, 8, 10}
big_numbers = {6, 7, 8, 9, 10}

# subtraction: big numbers which are not even
print(big_numbers - even_numbers)

# union: numbers which are big or even
print(big_numbers | even_numbers)

# intersection: numbers which are big and even
print(big_numbers & even_numbers)

# numbers which are big or even but not both
print(big_numbers ^ even_numbers)
```

range

- In Python, range is another kind of **immutable** sequence type. It is very specialized – we use it to create ranges of integers. It is mainly used in for loop.
- Ranges are generators. The numbers in a range are generated one at a time as they are needed, and not all at once.

```
# print the integers from 0 to 9
print(list(range(10)))

# print the integers from 1 to 10
print(list(range(1, 11)))

# print the odd integers from 1 to 10
print(list(range(1, 11, 2)))
```

```
#use range() in the for loop
for i in range(10):
    print("i=",i**2)
```

dictionary

- In Python, the dictionary type is called **dict**. We can use a dictionary to store key-value pairs.
- To define a dictionary literal, we put a comma-separated list of **key-value pairs** between curly brackets {}. We can use a colon to separate each key from its value.
- We can access values in the dictionary in the same way as list or tuple elements, but we use keys instead of indices.
- The keys of a dictionary don’t have to be strings – they can be any immutable type. **Keys must be unique**, but when we store a value in a dictionary, the key doesn’t have to exist – it will be created automatically.
- Like sets, dictionaries are not ordered – if we print a dictionary, the order will be random.

Example of dictionary

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }

# Get a value by its key, or None if it doesn't exist
marbles["red"]
marbles.get("orange")
# We can specify a different default
marbles.get("orange", 0)

# Add several items to the dictionary at once
marbles.update({"orange": 34, "blue": 23, "purple": 36})

# All the keys in the dictionary
marbles.keys()
# All the values in the dictionary
marbles.values()
# All the items in the dictionary
marbles.items()
```

Containers: Arrays

- **Array**, is also an ordered set of values, but there are some important differences between lists and arrays. 1) The number of elements in an array is fixed; 2) The element of an array must all be of the same type.

- Advantages of **arrays**: 1) Arrays can be two-dimensional, like matrices in algebra; 2) Arrays behave roughly like vectors or matrices: you can do arithmetic with them; 3) Arrays work faster than lists.

- *a = numpy.zeros([m,n], complex),* create a 2D complex array with *m* rows and *n* columns

- *a = numpy.array(r, float)*, convert a list *r* to be an array *a*

- *a[m,n,k]* or *a[m][n][k]* the element in a 3D array

Some examples

```
>>> A=[[1, 2, 3, 'r', 'tet'],[5,4]]
>>> A
[[1, 2, 3, 'r', 'tet'], [5, 4]]
>>> A[0]
[1, 2, 3, 'r', 'tet']
>>> A[1]
[5, 4]
>>> A[0][2]
3
```

```
>>> C=np.array(A[1])
>>> C
array([5, 4])
>>> type(C[1])
<class 'numpy.int64'>
```

```
>>> A
[[1, 2, 3, 'r', 'tet'], [5, 4]]
>>> B=np.array(A)
>>> B
array([list([1, 2, 3, 'r', 'tet']), list([5, 4])], dtype=object)
>>> type(B[1])
<class 'list'>
>>> type(B[1][1])
<class 'int'>
>>> B[1][1]
4
>>> B[0][1]
```

Some examples

```
>>> import numpy as np
>>> A=np.zeros([3,3],int)
>>> A
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> A[0,0]=1.5
>>> A
array([[1, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Slicing

- Here is a useful trick, called slicing, which works with arrays, lists and many other containers. We pass slice like this: ***[start:end]***, and we can also define the step like this ***[start:end:step]***. If we don't pass start its considered 0; If we don't pass end its considered length of array in that dimension; If we don't pass step its considered 1. The slicing could be negative.

- Suppose we have a list ***r***, then ***r[m:n]*** is another list composed of a subset of the elements of r, starting with element ***m*** and going up to but ***NOT*** including element ***n***. ***r[m:]*** means all the elements staring with element ***m***.

- Slicing works with two-dimensional arrays as well. For example, ***a[2:4,3:6]*** gives you a 2D array of size 2*3 with values drawn from the appropriate sub-block of ***a*** starting at ***a[2,3]***.

Shape and Reshape()

- The shape of an array is the number of elements in each dimension. ***a.shape***

- We can **reshape** the array by adding or removing dimensions or changing number of elements in each dimension.
- We can reshape an array into any shape as long as the elements required for reshaping are equal in both shapes.
- You are allowed to have one "unknown" dimension. Pass -1 as the value, and NumPy will calculate this number for you.

e.g.
A=np.array([[1, 2, 3, 4], [5, 6, 7, 8]]); B=A.reshape(2,2,-1)
C=A.reshape(-1)

Q: Is B or C an independent new array??

Array Iterating

- Iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy, we can do this using basic **for** loop of python.
- If we iterate on a 1-D array it will go through each element one by one. In a 2-D array it will go through all the rows. To return the actual values, the scalars, we have to iterate the arrays in each dimension. If we iterate on a n-D array it will go through n-1th dimension one by one.
- The function **nditer()** is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.
- Enumeration means mentioning sequence number of somethings one by one. Sometimes we require corresponding index of the element while iterating, the **ndenumerate()** method can be used for those cases.

```python
import numpy as np

n1=2;n2=3;n3=4;
A1 =np.array([x for x in range(n1)])
A2 =np.array([[x+y*10 for x in range(n1)] for y in range(n2)])
A3 =np.array([[[x+y*10+z*100 for x in range(n1)]
               for y in range(n2)]
              for z in range(n2)])
```

```python
for x in A1:
    print(x)

print()
for x in A2:
    print(x)

print()
for x in A2:
    print(x)
    for y in x:
        print(y)
```

```python
print()
for x in A3:
    print(x)

print()
for x in np.nditer(A3):
    print(x)

for x in np.nditer(A3[:, ::2]):
    print(x)

for idx, x in np.ndenumerate(A2):
    print(idx, x)
```

Joining Array

- Joining means putting contents of two or more arrays in a single array. In NumPy we join arrays ***by axes***.
- ***concatenate()***, join arrays along with the axis. If axis is not explicitly passed, it is taken as 0.
- **stack()**, join array along a new axis; **hstack(),** stack along rows; **vstack()**, stack along columns; **dstack()**, stack along heights/depths.

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr_j0 = np.concatenate((arr1, arr2), axis=0)
arr_j1 = np.concatenate((arr1, arr2), axis=1)

print(arr_j0)
print(arr_j1)
```

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr_s0 = np.stack((arr1, arr2), axis=0)
arr_s1 = np.stack((arr1, arr2), axis=0)
arr_hs = np.hstack((arr1, arr2))
arr_vs = np.vstack((arr1, arr2))
arr_ds = np.dstack((arr1, arr2))
print(arr_s0)
print(arr_s1)
print(arr_hs)
print(arr_vs)
print(arr_ds)
```

Splitting Array

- We use ***array_split()*** for splitting one array into multiple.
- If the array has less elements than required, it will adjust from the end accordingly.
- For high dimension array, you can specify which ***axis*** you want to do the split around.
- Similarly, we can have ***hsplit()***, ***vsplit()*** and ***dsplit()***.

```
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 4)


arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])
newarr2d_a0 = np.array_split(arr2d, 3)
newarr2d_a1 = np.array_split(arr2d, 3, axis=1)

print(newarr2d_a0)
print(newarr2d_a1)
```

Searching Arrays

- To search an array, use the ***where()*** method.
- ***searchsorted()*** can performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order. By default the left most index is returned, but we can give ***side='right'*** to return the right most index instead.
- To search for more than one value, use an array with the specified values.

```
arr = np.array([1, 2, 3, 4, 2, 6])
x = np.where(arr%2 == 0)
print(x)

#The method starts the search from the left and returns the first index
# where the number 2.5 is no longer larger than the next value.
x_s = np.searchsorted(arr, 2.5)
x_ms = np.searchsorted(arr, [2.5,4.5,6.5])
```

Sorting Arrays

- The NumPy ndarray object has a function called ***sort()***, that will sort a specified array. This method returns a copy of the array, leaving the original array unchanged. This method can also use axis to specify which axis.
- ***argsort()***, returns an array of indices of the same shape as a that index data along the given axis in sorted order.
- Sort a numpy 2d array by a certain row with columns maintained, on need to use NumPy indexing.

```
arr = np.array([[3,2,4], [5,0,1],[7,9,3]])
print(np.sort(arr,axis=0)) ;print(np.sort(arr,axis=1))

ind=np.argsort(arr,axis=0);print(np.take_along_axis(arr,ind,axis=0))

#sort based on the first row
print(arr[:, arr[0].argsort()])
```

Filter Array

- You filter an array using a boolean index list. If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.
- There are many different ways to create the filter array. For example, one can use ***for*** loop to do it.

```
arr = np.array([41,42,43,44])
x = [True,False,True,False]
newarr = arr[x]
print(newarr)

filter_arr = arr > 42
print(arr[filter_arr])

filter_arr = arr%2==0
print(arr[filter_arr])
```

```
filter_arr = []

for element in arr:
if element % 2 == 0:
    filter_arr.append(True)
  else:
    filter_arr.append(False)
```