CUHK Business School
The Chinese University
of Hong Kong

MSC IN FINANCE PRE-TERM COURSE WEEK2: Object-Oriented Programming, Numerical Computing with NumPy

Tutor: Yuan Lu
CUHK Department of Finance
yuan.lu@link.cuhk.edu.hk

July 25, 2023

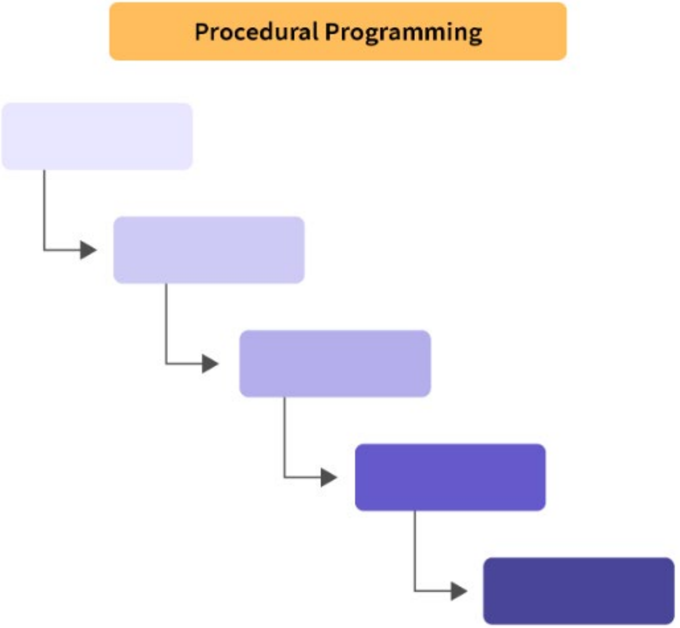
Agenda:

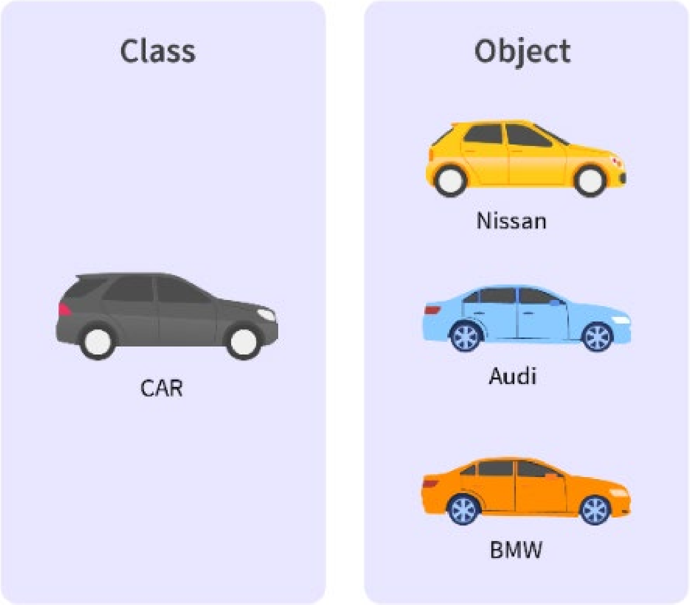
1. **Object-Oriented Programming (OOP)**
   - Class and Object
   - Class Attributes and Methods

2. **NumPy**
   - ndarray
   - Numerical Computing with NumPy

Object-Oriented Programming (OOP)

- Two basic programming paradigms:
  - Procedural
    - Organizing programs around functions or blocks of statements which manipulate data.
  - Object-Oriented
    - Structuring a program by bundling related properties and behaviors into individual objects.





Object-Oriented Programming(Cont.)

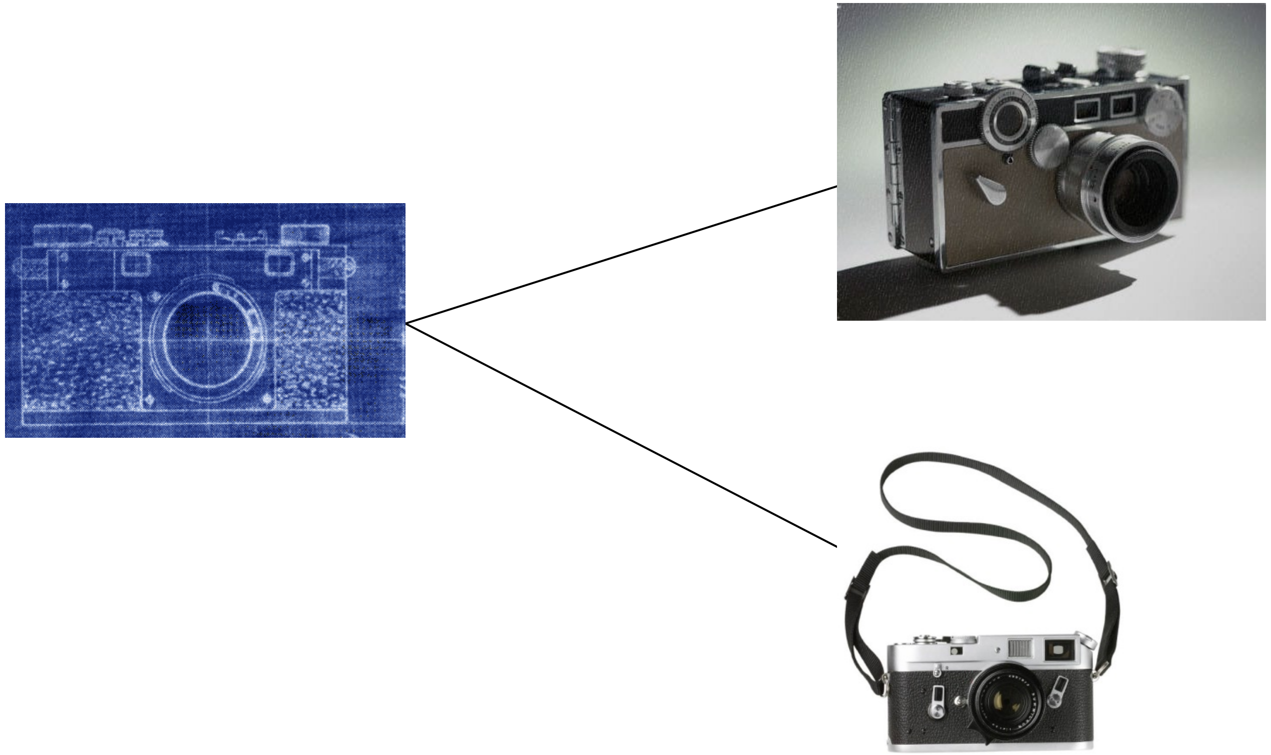
- **Classes** and **objects** are the two main aspects of object-oriented programming.
- A **class** creates a new *type*.
- An **object** is the *instance* of the class.

- Everything in Python is really an object.
- E.g.: An analogy is that we can have variables of type *int* which translates to saying that variables that store integers are variables which are instances (objects) of the *int* class.

Class and Object

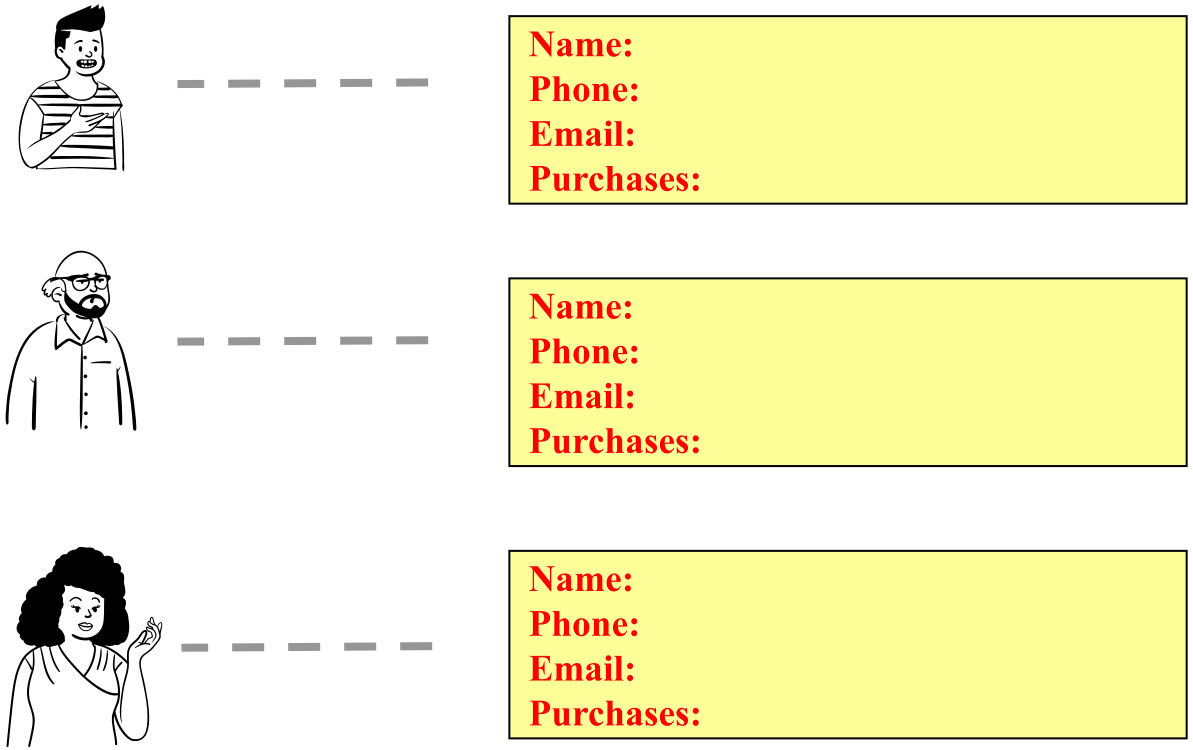
- Defining a class
  - A template that describes that class: how many fields, what type of information will be stored by each field, what default information will be stored in a field.

- Creating objects
  - Instances of that class (during instantiation) which can take on different forms.



Class: Define A Composite Type

- Classes can be used to define a generic template for a new non-homogeneous composite type.
- The class definition specifies the type of information (called "**attributes**") that each instance (example) tracks.



Name:
Phone:
Email:
Purchases:

Name:
Phone:
Email:
Purchases:

Name:
Phone:
Email:
Purchases:

Define a Class

Note the convention: The first letter is capitalized.

- **Format:**

```
class <Name of the class>:
     name of first field = <default value>
     name of second field = <default value>
```

- **Example:**

```
class Client:
    name = "default"
    phone = "(123)456-7890"
    email = "foo@bar.com"
    purchases = 0
```

Describes what information that would be tracked by a “Client” but doesn’t actually create a client variable

Contrast this with a list definition of a client

```
client = [“xxxxxxxxxxxxxxx",
          “0000000000",
          “xxxxxxxxx",
          0]
```

Create an Instance (Object) of a Class

- Creating an actual instance (instance = object) is referred to as

- **Format:**

```
<reference name> = <name of class>()      # Create a new object
<reference name>.<field name>             # Accessing value
<reference name>.<field name> = <value>  # Changing value
```

- **Example:**

```
aClient = Client()
aClient.name = “Lily"
aClient.email = “python@cuhk.edu.hk"
```

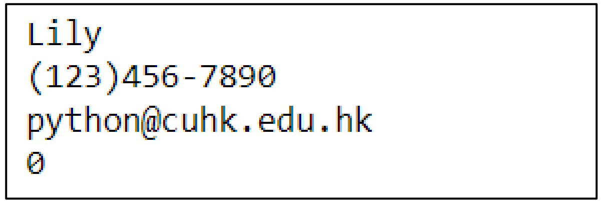
Example: The Client Class

```
class Client:
    name = "default"
    phone = "(123)456-7890"
    email = "foo@bar.com"
    purchases = 0
```

```
firstClient = Client()
firstClient.name = “Lily”
firstClient.email = “python@cuhk.edu.hk"
print(firstClient.name)
print(firstClient.phone)
print(firstClient.email)
print(firstClient.purchases)
```

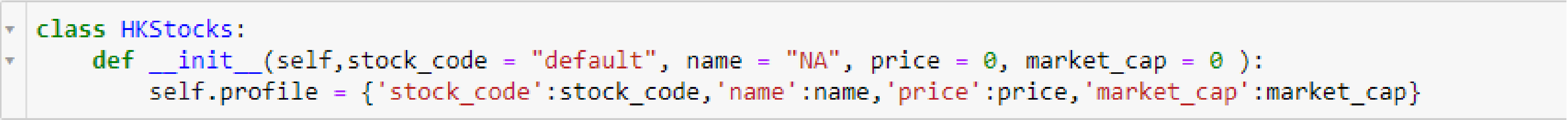
```
name = "default"
phone = "(123)456-7890"
email = "foo@bar.com"
purchases = 0
```

```
name = “Lily"
email = “python@cuhk.edu.hk"
```

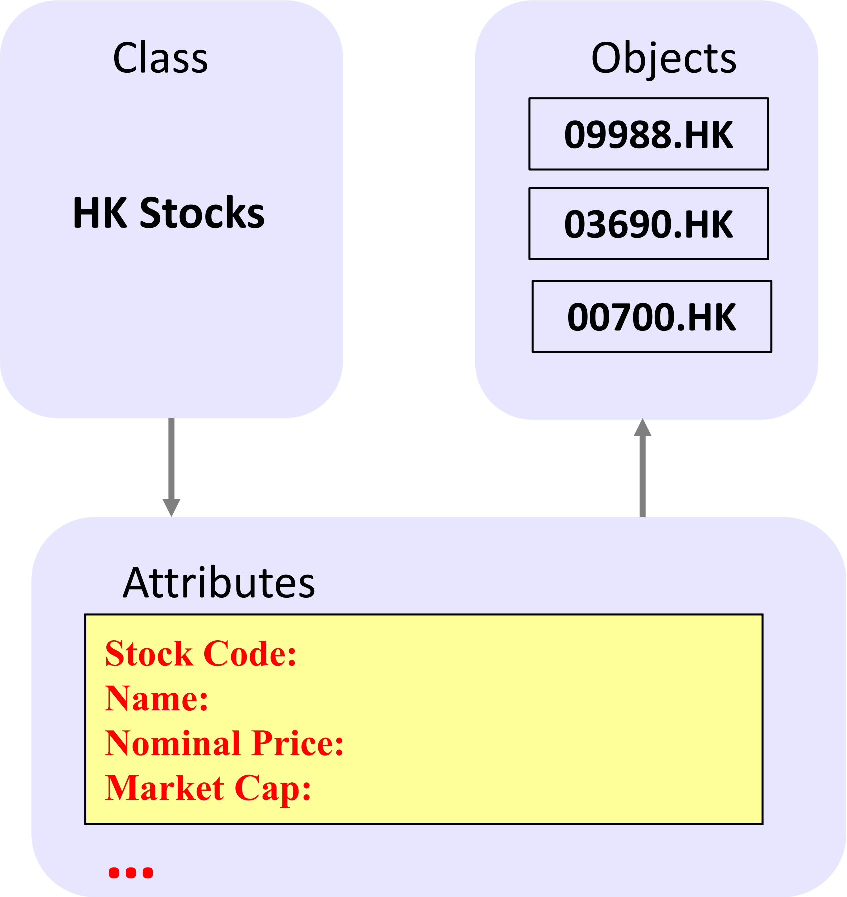


```
Lily
(123)456-7890
python@cuhk.edu.hk
0
```

Example: HK Stocks



```
class HKStocks:
    def __init__(self,stock_code = "default", name = "NA", price = 0, market_cap = 0 ):
        self.profile = {'stock_code':stock_code,'name':name,'price':price,'market_cap':market_cap}
```



Class

HK Stocks

Objects

09988.HK

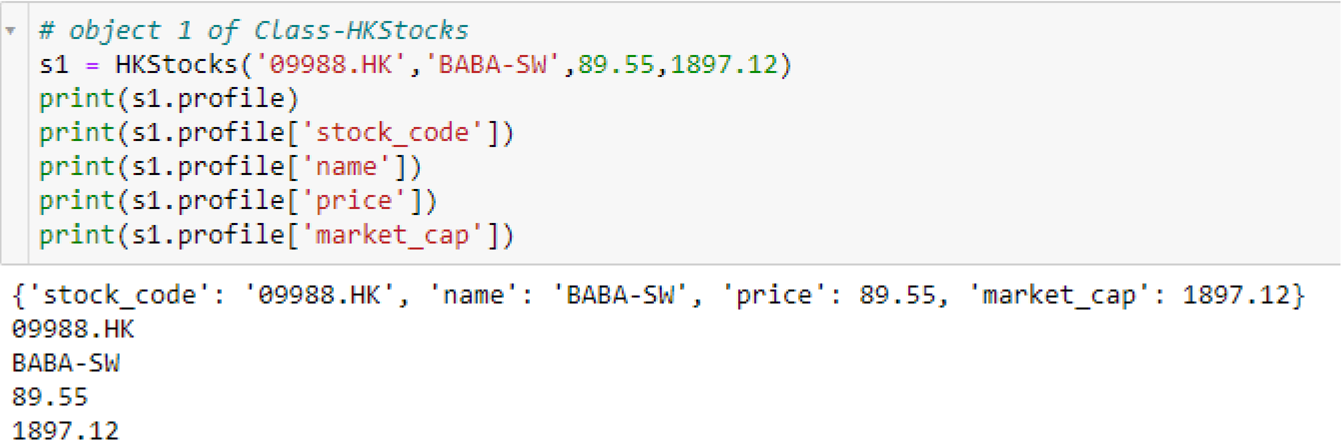
03690.HK

00700.HK

Attributes

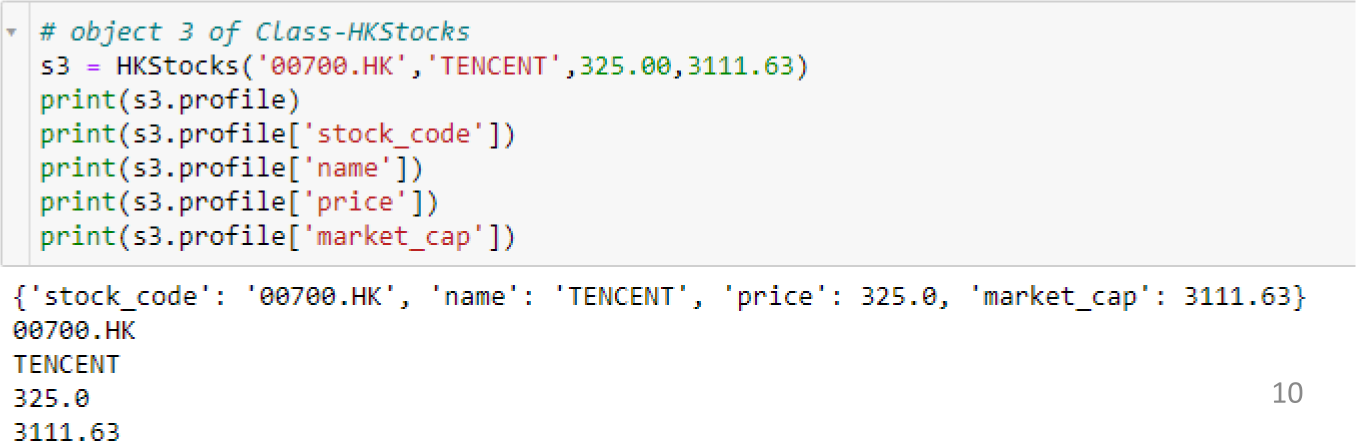
Stock Code:
Name:
Nominal Price:
Market Cap:

…



```
# object 1 of Class-HKStocks
s1 = HKStocks('09988.HK','BABA-SW',89.55,1897.12)
print(s1.profile)
print(s1.profile['stock_code'])
print(s1.profile['name'])
print(s1.profile['price'])
print(s1.profile['market_cap'])
```

```
{'stock_code': '09988.HK', 'name': 'BABA-SW', 'price': 89.55, 'market_cap': 1897.12}
09988.HK
BABA-SW
89.55
1897.12
```



```
# object 3 of Class-HKStocks
s3 = HKStocks('00700.HK','TENCENT',325.00,3111.63)
print(s3.profile)
print(s3.profile['stock_code'])
print(s3.profile['name'])
print(s3.profile['price'])
print(s3.profile['market_cap'])
```

```
{'stock_code': '00700.HK', 'name': 'TENCENT', 'price': 325.0, 'market_cap': 3111.63}
00700.HK
TENCENT
325.0
3111.63
```

The Benefits of Defining a Class

- It allows new types of variables to be declared. The new type can model information about most any arbitrary entity:
  - Car, Movie, A bacteria or virus in a medical simulation, An ‘object’ (e.g., sword, ray gun, food, treasure) in a video game.
  - A member of a website (e.g., a social network user could have attributes to specify the person’s: images, videos, links, comments and other posts associated with the ‘profile’ object).
  - Firms (e.g. firms with different characteristics and also different investment choices)
  - Snapshots of order books (e.g. at each timestamp, the order book for each stock records information of price and volume and they will be changed by the incoming orders)

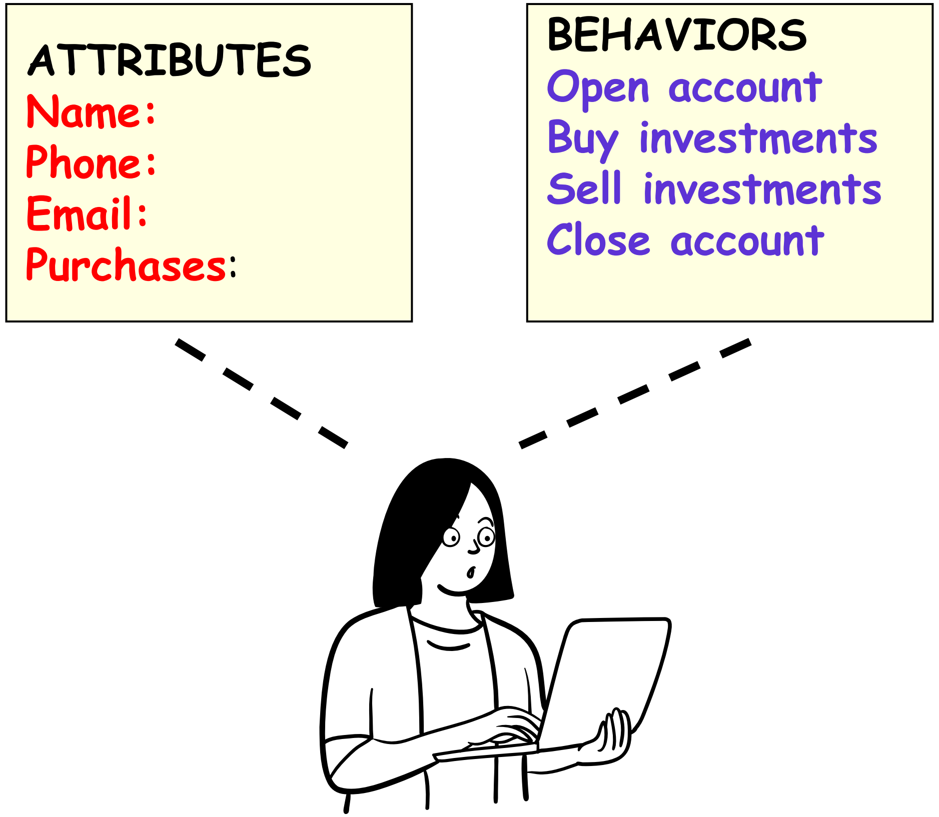
- Unlike creating a composite type by using a list, a predetermined number of fields can be specified, and those fields can be named.

```
class Client:
   name = "default"
   phone = "(123)456-7890"
   email = "foo@bar.com"
   purchases = 0
```

```
firstClient = Client ()
print(firstClient.middleName)  # Error: no such field defined
```

Class Methods (“Behaviors”)

Classes Have **Attributes,** But Also **Behaviors**



ATTRIBUTES
Name:
Phone:
Email:
Purchases:

BEHAVIORS
Open account
Buy investments
Sell investments
Close account

Functions: not tied to a composite type or object.

The call is ‘stand alone’, just name of function

E.g.,

print(), **input()**

Methods: must be called through an instance of a composite.

E.g.,

```
filename = "foo.txt"
name, suffix = filename.split('.’)
```

Functions that are associated with classes are referred to as ***methods***.

Define Class Methods

- **Format**:

```
class <classname>:
     def <method name> (self, <other parameters>):
          <method body>
```

- **Example**:

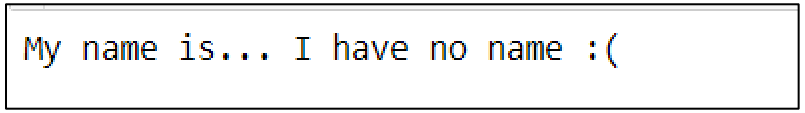
```
class Person:
   name = "I have no name :("
   def sayName (self):
      print ("My name is...", self.name)
```

Unlike functions, every method of a class must have the ‘`self`’ parameter (more on this later)

When the attributes are accessed inside the methods of a class they MUST be preceded by the prefix “`self.`”

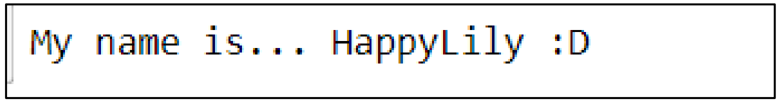
Example: Defining Class Methods

```
class Person:
    name = "I have no name :("
    def sayName(self):
        print("My name is...", self.name)

aPerson = Person()
aPerson.sayName()
```



```
My name is... I have no name :(
```

```
aPerson.name = “HappyLily :D"
aPerson.sayName()
```



```
My name is... HappyLily :D
```

Recap: Accessing Attributes & Methods

- **Inside the class definition** (inside the body of the class methods)
  - Prefix the attribute or method using the ‘**self**’ reference

```
class Person:
    name = " I have no name :( "
    def sayName(self):
        print("My name is...", self.name)
```

- **Outside the class definition**
  - Prefix the attribute or method using the **name of the reference** used when creating the object.

```
lisa = Person()
lisa.name = "Lisa, Nice to meet you.“
lisa.sayName()
```

Constructor: A Special Method

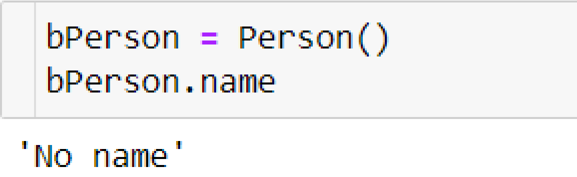
- Classes have a special method '__init__()' that can be used to initialize the starting values of a class to some specific values.
- This method is automatically called whenever an object is created.
- **Format**:

```
class <Class name>:
def __init__(self, <other parameters>):
     <body of the method>
```

- **Example**:

```
class Person:
   name = ""
   def __init__(self):
       self.name = "No name"
```

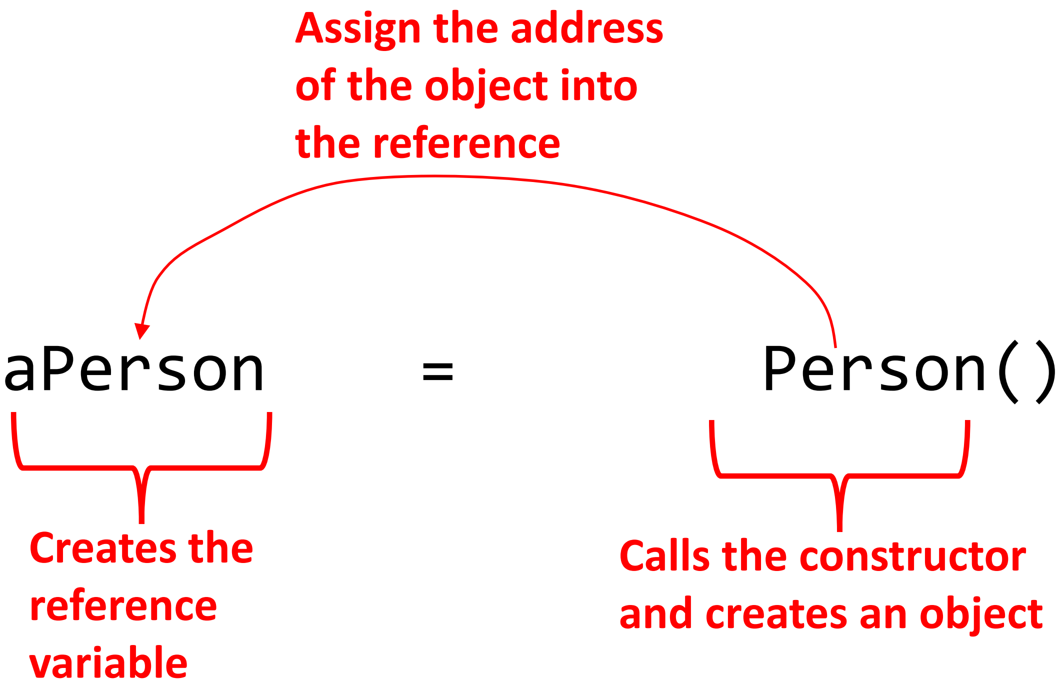
This design approach is consistent with many languages



```
bPerson = Person()
bPerson.name
```

```
'No name'
```

Example: Using The "__Init__()" Method



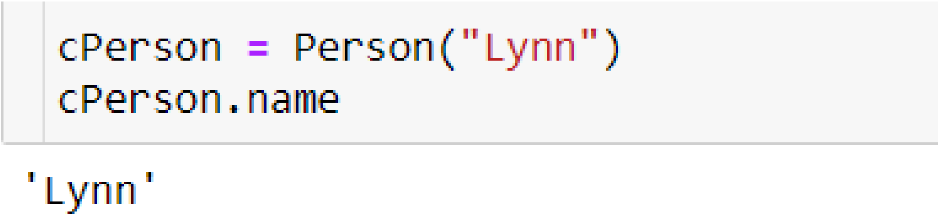
Assign the address of the object into the reference

aPerson = Person()

Creates the reference variable

Calls the constructor and creates an object

```
class Person:
    name = ""

    def __init__(self, aName):
        self.name = aName
```



```
cPerson = Person("Lynn")
cPerson.name
```

```
'Lynn'
```

Example: Using The "__Init__()" Method (Contd.)

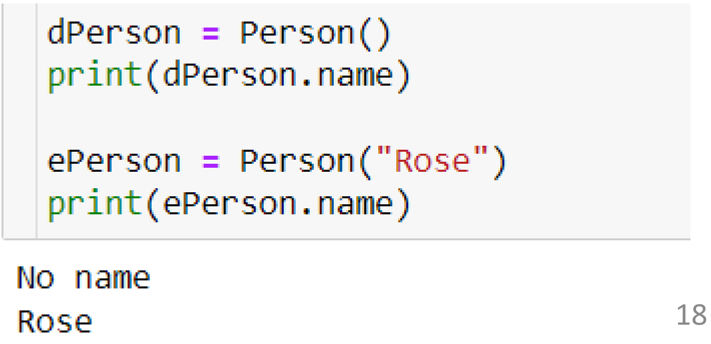
- Similar to other methods, 'init' can be defined so that if parameters aren't passed into them then default values can be assigned.
- **Example**:

```
def __init__ (self, aName="No name"):
    self.name = aName
```

This method can be called either when a personalized name is given or if the name is left out.

- Method calls (to 'init'), both will work

```
dPerson = Person()
ePerson = Person("Rose")
```



```
dPerson = Person()
print(dPerson.name)

ePerson = Person("Rose")
print(ePerson.name)
```

```
No name
Rose
```

Complete Example: Class Person with Birthday

```
class Person:
    name = “No Name"
    age = 0

    def __init__(self,newName,newAge):
        self.name = newName
        self.age = newAge

   def haveBirthday(self):
       print("Happy Birthday!")
       self.mature()

    def mature(self):
        self.age = self.age + 1
```

```
def __init__(self,newName,newAge
    self.name = newName
    self.age = newAge
```

```
aPerson = Person("Cartman",8)
```

```
Cartman is 8.
```

```
print("%s is %d." %(aPerson.name,aPerson.age))
aPerson.haveBirthday()
```

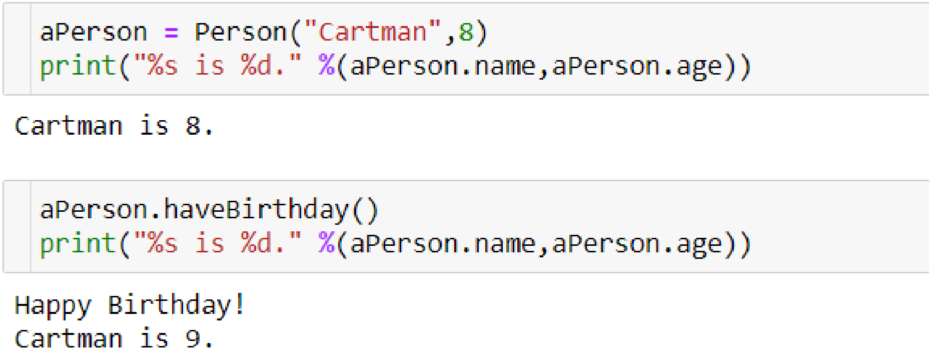
```
Happy Birthday!
```

```
print("%s is %d." %(aPerson.name,aPerson.age))
```

```
Cartman is 9.
```

```
def haveBirthday(self)
    print("Happy Birthday!")
    self.mature()
```

```
def mature(self):
    self.age = self.age + 1
```



```
aPerson = Person("Cartman",8)
print("%s is %d." %(aPerson.name,aPerson.age))
```

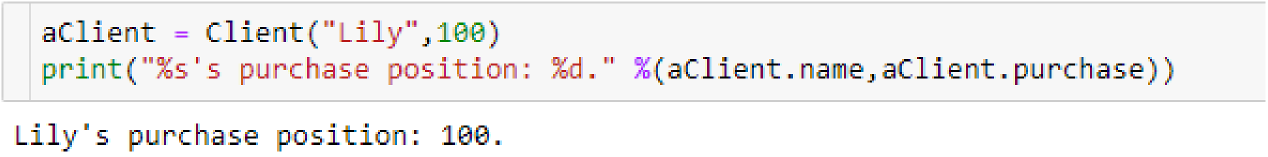
```
Cartman is 8.
```

```
aPerson.haveBirthday()
print("%s is %d." %(aPerson.name,aPerson.age))
```

```
Happy Birthday!
Cartman is 9.
```

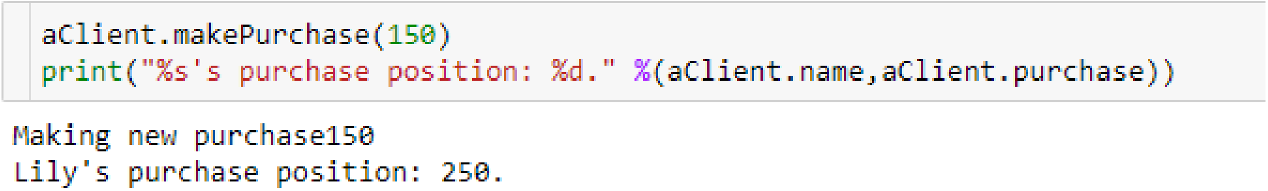
Complete Example: Class Client with Purchases

```
class Client:
    name = “No Name"
    purchase = 0

    def __init__(self,newName,newPurchase):
        self.name = newName
        self.purchase = newPurchase

    def makePurchase(self, addPurchase):
        print(“Making new purchase “+ str(addPurchase))
        self.purchase = self. purchase + addPurchase
```



```
aClient = Client("Lily",100)
print("%s's purchase position: %d." %(aClient.name,aClient.purchase))
```

```
Lily's purchase position: 100.
```



```
aClient.makePurchase(150)
print("%s's purchase position: %d." %(aClient.name,aClient.purchase))
```

```
Making new purchase150
Lily's purchase position: 250.
```

Numpy

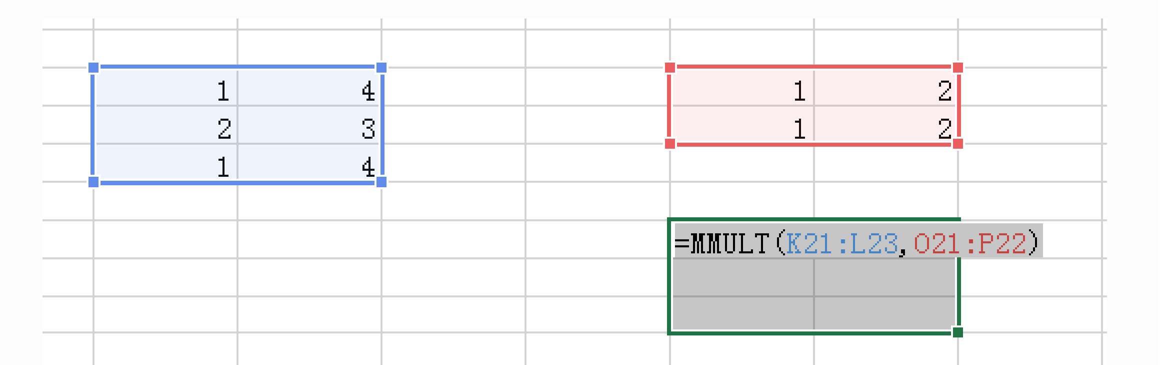
1. Why NumPy?
2. ndarray
3. Copies vs Views
4. Broadcasting
5. Iterations

Why NumPy?

- ndarray stands for N-dimensional array.
- A NumPy array is a grid of values, all of the same data type.
- This simplifies the data storage process and makes numerical operations faster.
- NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, such as
  - Pandas,
  - Scikit Learn
  - and more …

Why NumPy: Compared with Excel in Matrix Operations

Matrices multiplication in Excel



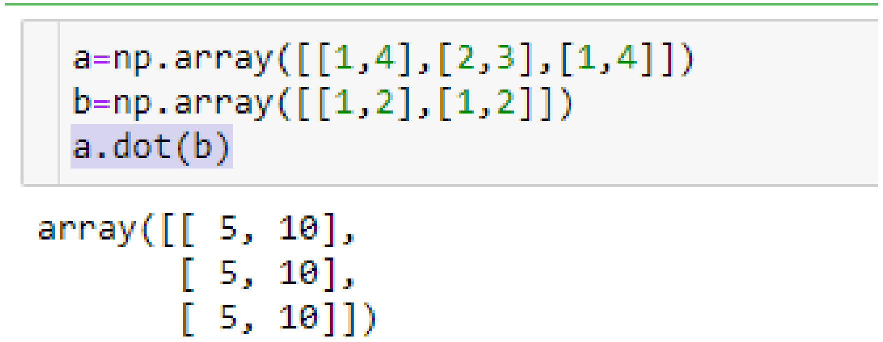
1	4
2	3
1	4

1	2
1	2

=MMULT(K21:L23,O21:P22)

- Three steps:
- Select the input and output areas
- array formula =MMULT(XX:XX,XX:XX)
- ctrl+shift+enter

Matrices multiplication in NumPy



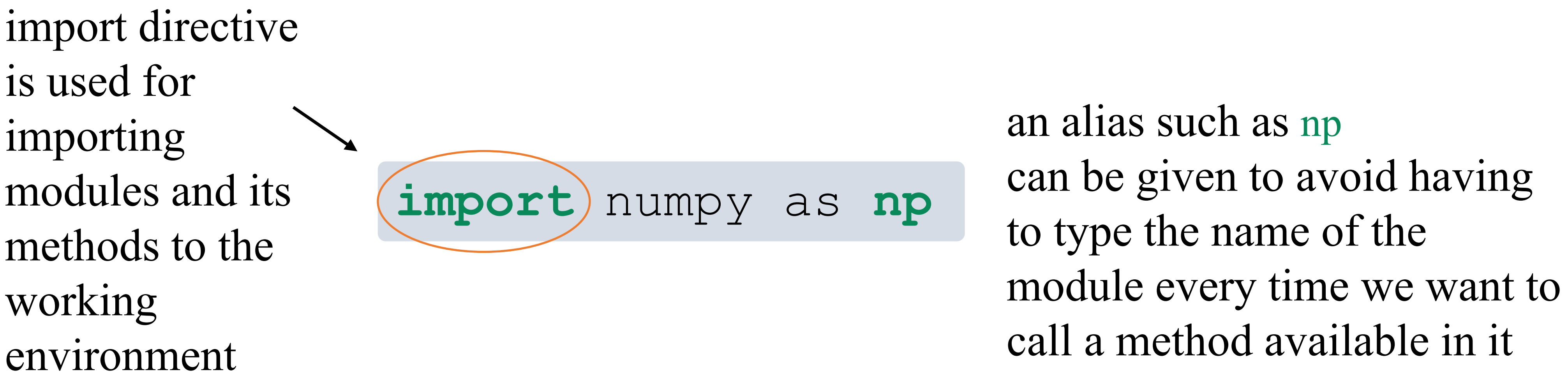
```
a=np.array([[1,4],[2,3],[1,4]])
b=np.array([[1,2],[1,2]])
a.dot(b)
```

```
array([[ 5, 10],
       [ 5, 10],
       [ 5, 10]])
```

- Only need one line:
- a.dot(b)

ndarray: import

- We begin using modules by importing them



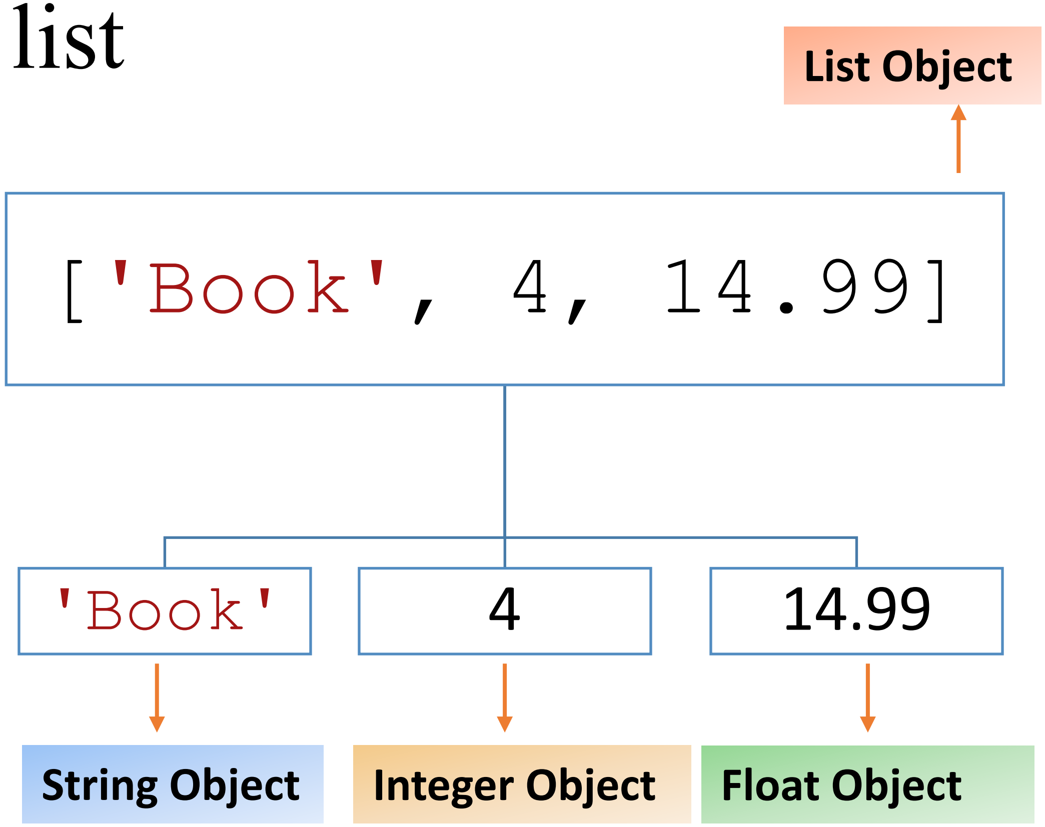
import directive is used for importing modules and its methods to the working environment

```
import numpy as np
```

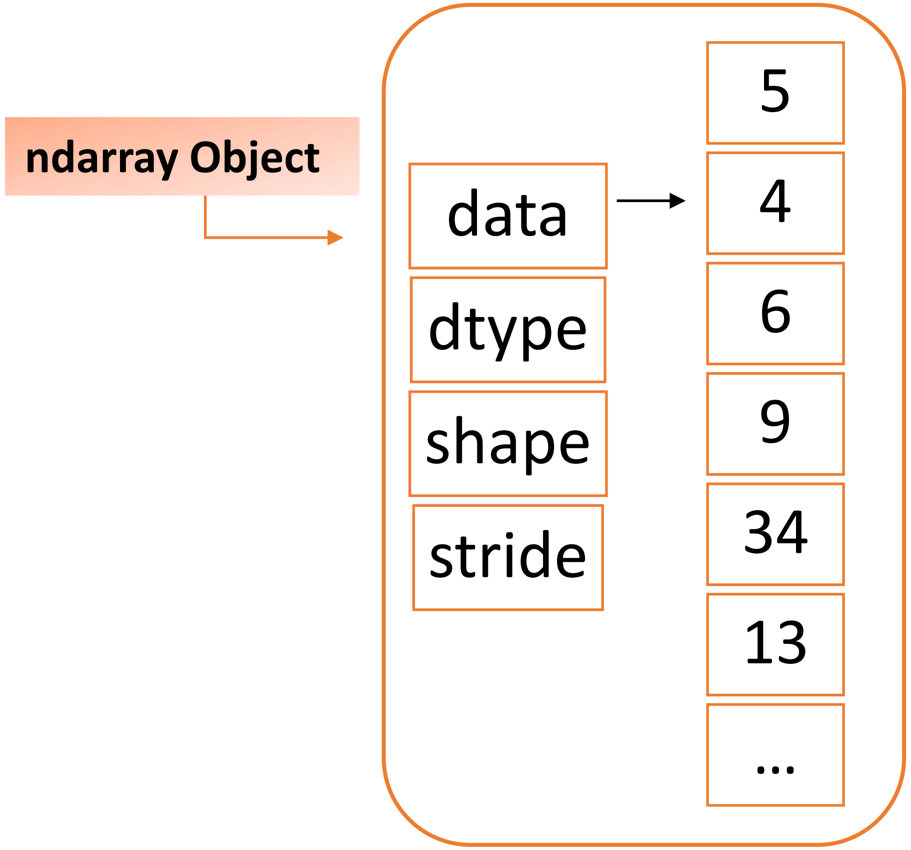
an alias such as np can be given to avoid having to type the name of the module every time we want to call a method available in it

List vs ndarray data structure

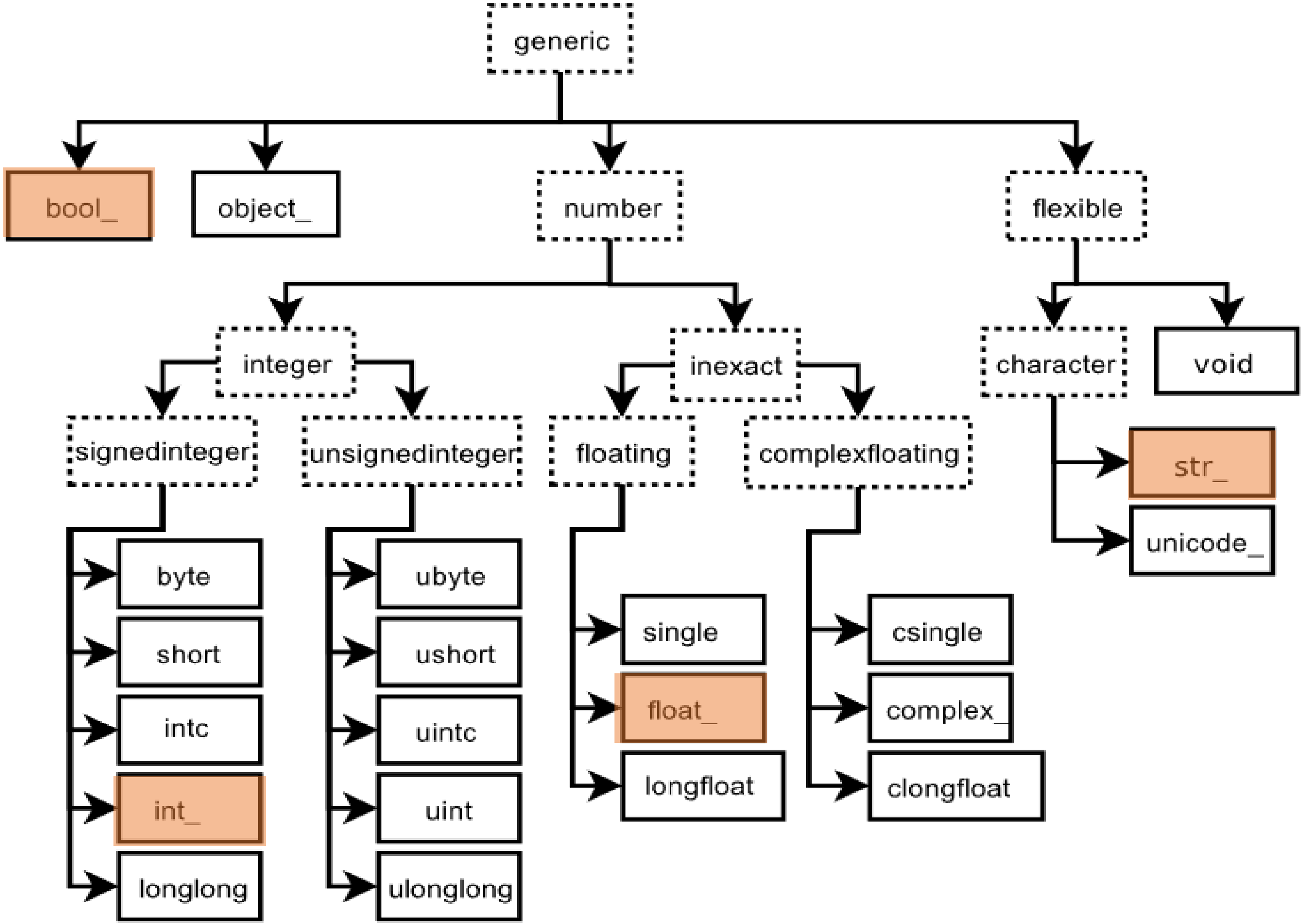
list



ndarray



ndarray: dtype



and there are more data types supported by NumPy ….

such as datetime objects

ndarray vs list operation

add 2 to each item of -

➢ a list

```
numlist = [1, 2, 3]
```

```
doubled_list = []
for i in numlist:
    doubled_list.append(i+2)
```

➢ an array

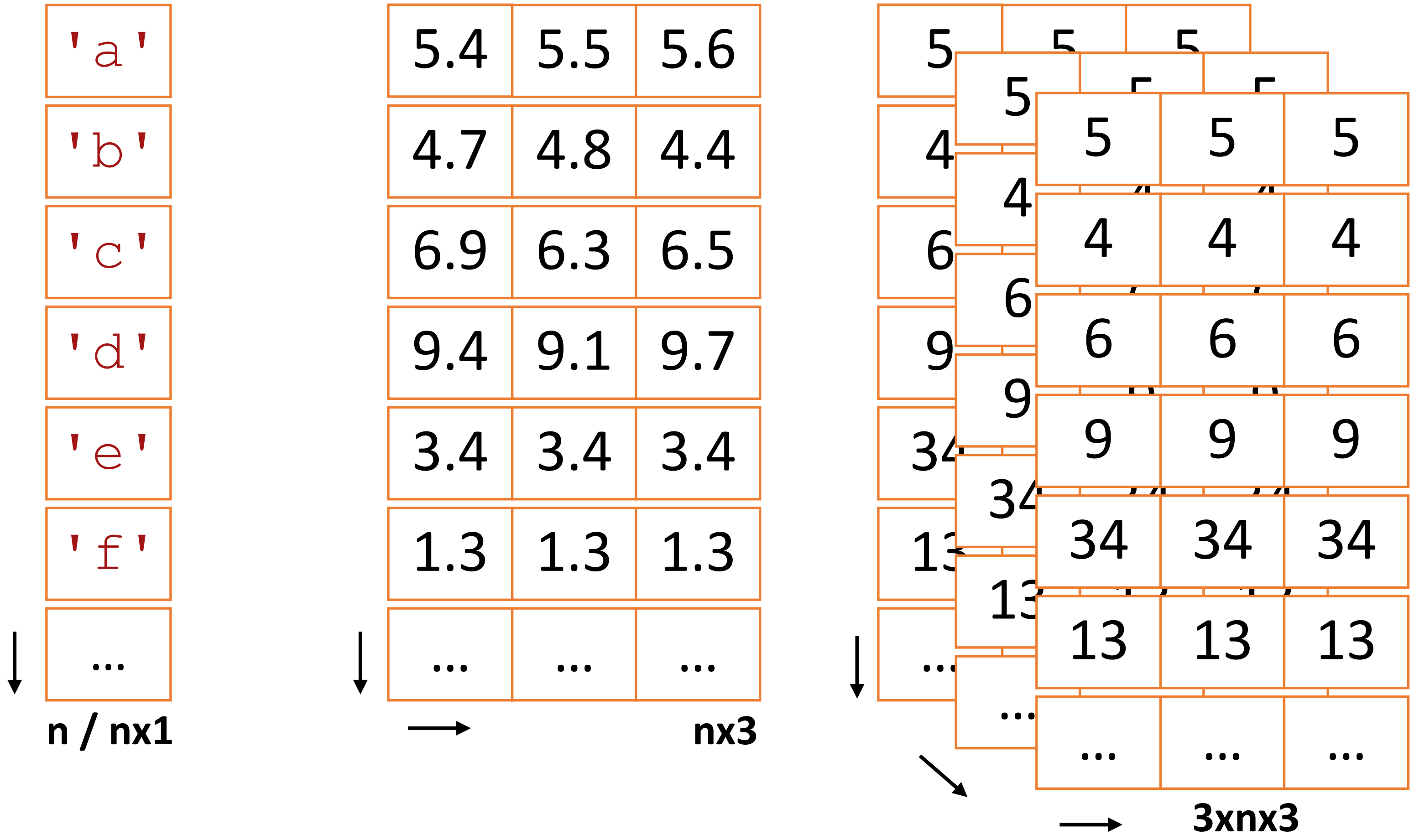
```
A = np.array([1, 2, 3])
```

```
doubleA = np.add(A,2)
```

numerical operations are made easier with NumPy

click here to see a list of them

ndarray: shape



'a'
'b'
'c'
'd'
'e'
'f'
…

n / nx1

5.4	5.5	5.6
4.7	4.8	4.4
6.9	6.3	6.5
9.4	9.1	9.7
3.4	3.4	3.4
1.3	1.3	1.3
…	…	…

nx3

5	5	5
4	4	4
6	6	6
9	9	9
34	34	34
13	13	13
…	…	…

3xnx3

can be even more dimensions!

ndarray: numerical operations

x	y
0	5
1	4
2	3
3	2
4	1
5	0

Arithmetic

x/3	x+y
0	5
0.33	5
0.67	5
1.0	5
1.33	5
1.67	5

Conditional

x>=3	x<y
False	True
False	True
False	True
True	False
True	False
True	False

ndarray: numerical operations (Cont.)

What if you want to perform arithmetic operation on arrays with different shapes?

`x`

0
1
2
3
4
5

`z`

0	0
0	0
0	0

`x+z`

not possible! ✗

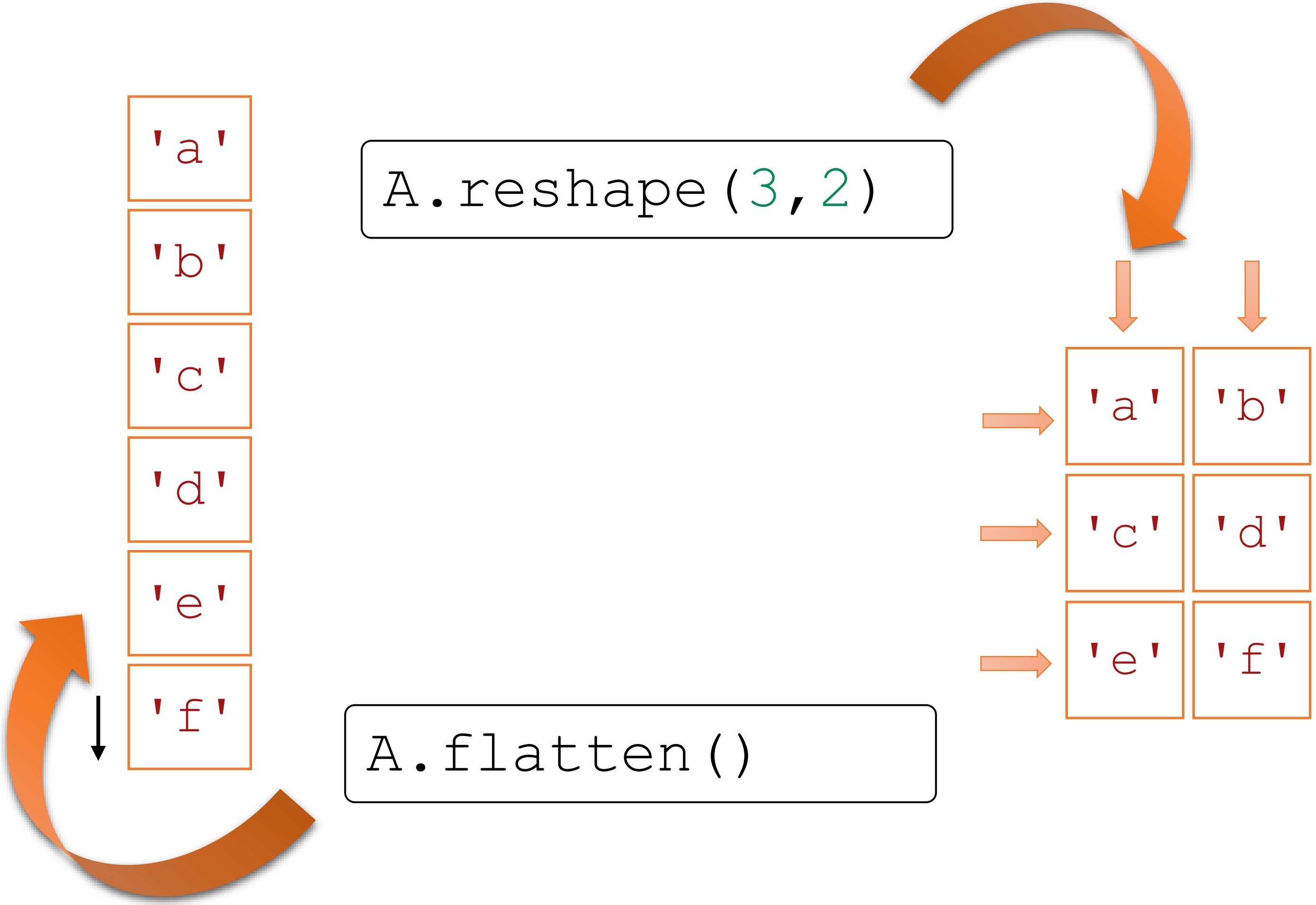
operations are done element wise

so the dimensions must match

we must reshape the array to perform this operation

ndarray: reshape

```
A = np.array(['a','b','c','d','e','f'])
```



```
A.reshape(3,2)
```

```
A.flatten()
```

other reshape methods are

- np.concatenate()
- np.hstack()
- np.vstack()
- np.transpose()

ndarray: initialization

```
x = np.arange(6)
```

0
1
2
3
4
5

```
x = np.zeros(3)
```

0
0
0

```
x = np.zeros((3,2))
```

0	0
0	0
0	0

and there are more ways such as

- np.full
- np.random.rand
- np.ones
- np.eye …

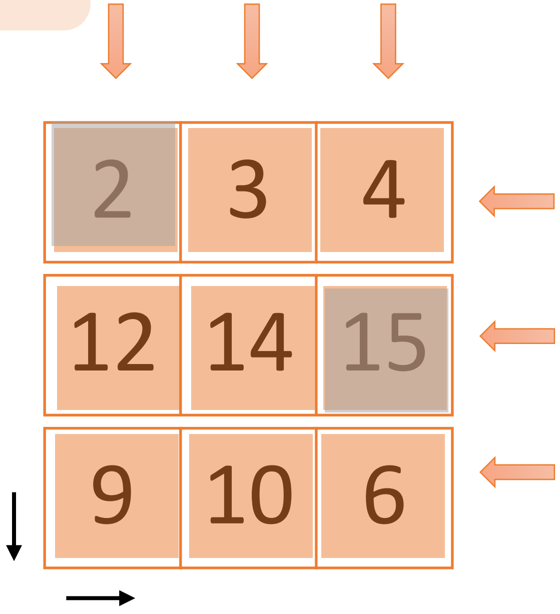
ndarray: methods

```
B = np.array([(2,3,4),
              (12,14,15),
              (9,10,6)])
```

```
print(B.ndim)
→ 2
```

```
print(B.shape)
→ (3,3)
```

```
print(B.size)
→ 9
```



2	3	4
12	14	15
9	10	6

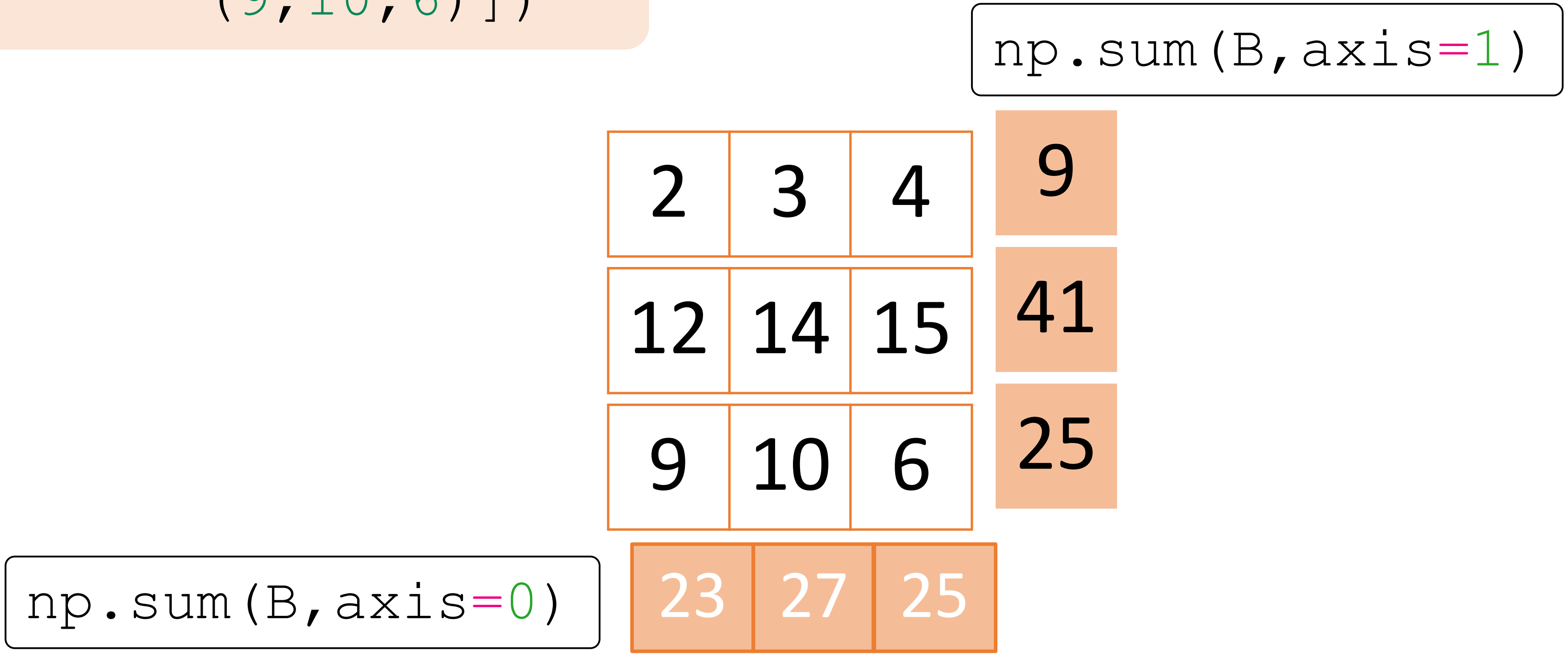
```
print(B.dtype)
→ int64 or int32
```

```
print(B.max())
→ 15
```

```
print(B.min())
→ 2
```

ndarray: methods (Cont.)

```
B = np.array([(2,3,4),
              (12,14,15),
              (9,10,6)])
```



```
np.sum(B,axis=1)
```

2	3	4	9
12	14	15	41
9	10	6	25
23	27	25	

```
np.sum(B,axis=0)
```

ndarray: indexing, slicing, stepping

```
A = np.array(['a','b','c','d','e','f'])
```

Indexing

```
A[0]
'a'
```

Slicing

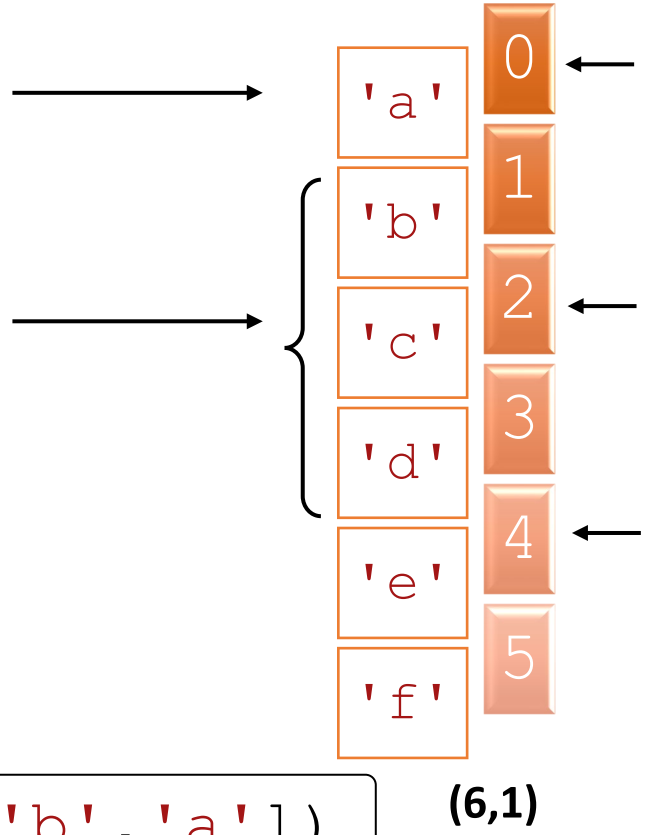
```
A[1:4]
array(['b','c','d'])
```

Stepping

```
A[::2]
array(['a','c','e'])
```

Negative Indexing

```
A[::-1]
nd.array(['f','e','d','c','b','a'])
```



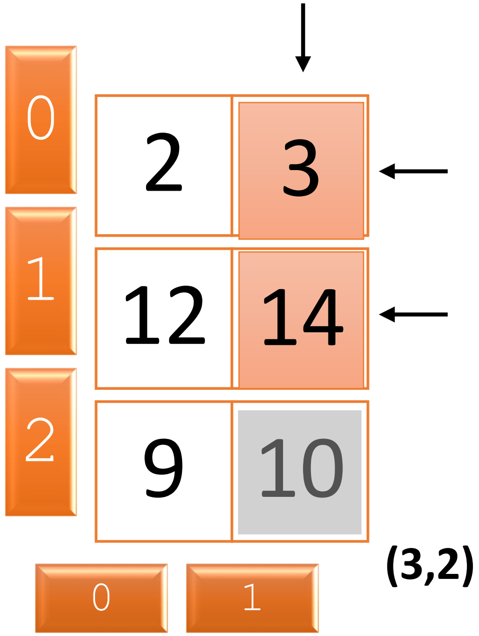
ndarray: indexing, slicing, stepping (Cont.)

```
B = np.array([(2,3),
              (12,14),
              (9,10)])
```

- ➢ comma separates the each dimension
- ➢ colon (:) selects everything in that dimension

```
print(B[2,1])
```

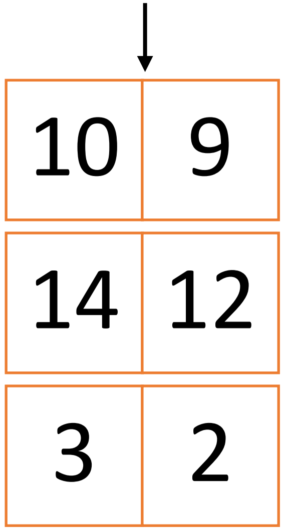
```
print(B[:2,1])
```



	0	1
0	2	3
1	12	14
2	9	10

(3,2)

```
print(B[::-1,::-1])
```



10	9
14	12
3	2

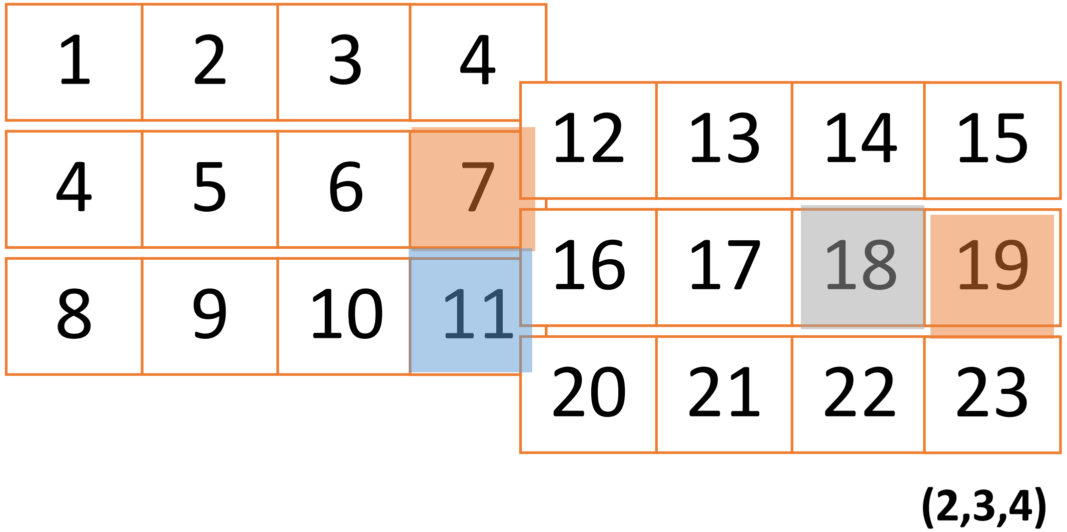
ndarray: indexing, slicing, stepping (Cont.)

```
C = np.array([
      [[1,2,3,4],
       [4,5,6,7],
       [8,9,10,11]],

      [[12,13,14,15],
       [16,17,18,19],
       [20,21,22,23]]
    ])
```

```
print(C[0,2,-1])
```

```
print(C[1,1,2])
```

```
print(C[:,1:2,3:])
```



1	2	3	4
4	5	6	7
8	9	10	11

12	13	14	15
16	17	18	19
20	21	22	23

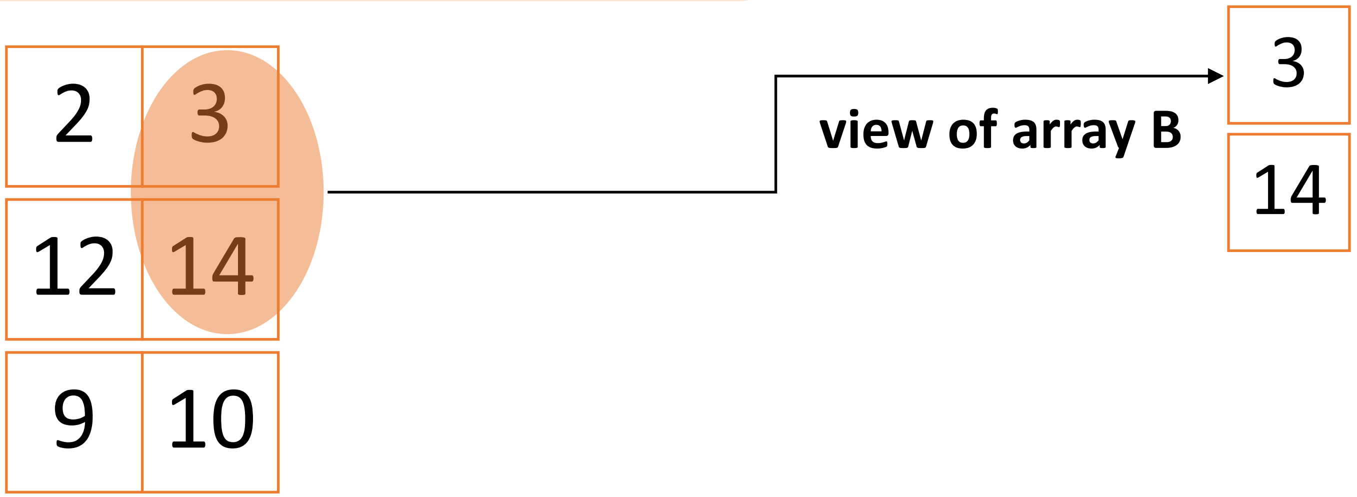
(2,3,4)

ndarray: copies vs views

- Slicing creates a view on the original array, not a new array.

```
B = np.array([(2,3),
              (12,14),
              (9,10)])
```

```
c = B[:2,1]
```



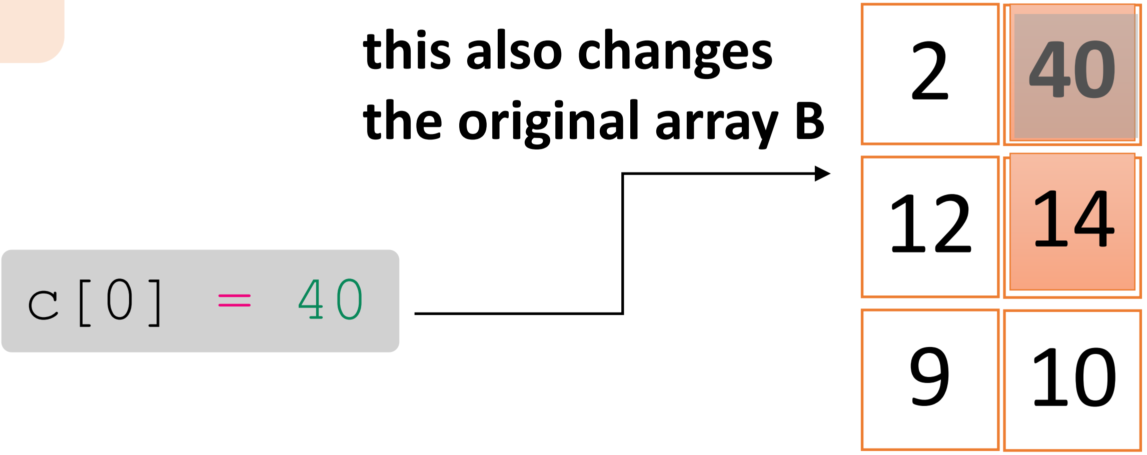
ndarray: copies vs views (Cont.)

- Changing a view changes the original array.

```
B = np.array([(2,3),
              (12,14),
              (9,10)])
```

```
c = B[:2,1]
```

2	**3**
12	14
9	10



```
c[0] = 40
```

this also changes the original array B

2	**40**
12	14
9	10

ndarray: copies vs views (Cont.)

- Use .copy() method to not change the original array.

```
B = np.array([(2,3),
              (12,14),
              (9,10)])
```

2	3
12	14
9	10

```
c = B[:2,1].copy()
```

3
14

```
c[0] = 40
```

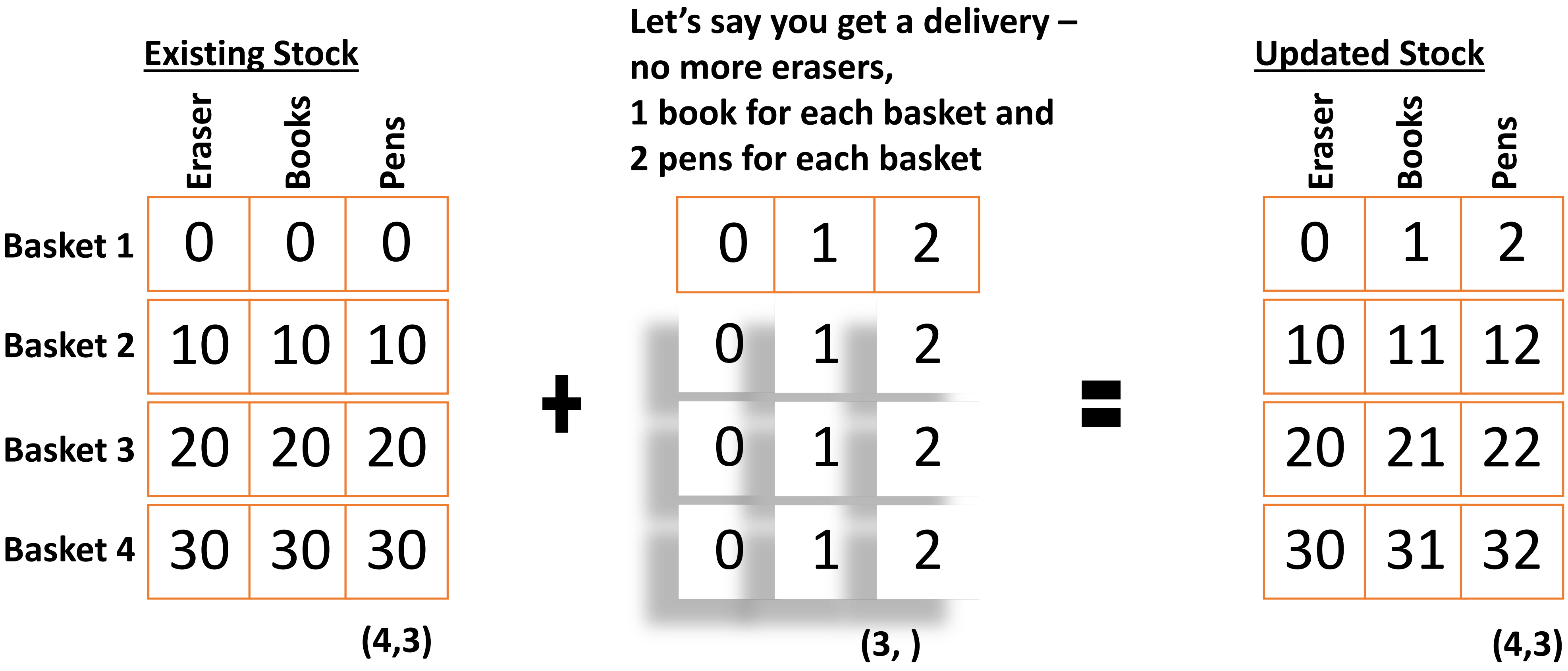
B remains unchanged because c is a copy of B, not a view of B

Broadcasting

- It is possible to do operations on arrays of different dimensions if NumPy can transform these arrays so that they all have same size.
- This conversion is called broadcasting.

Broadcasting (Cont.)

- RULE 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading side.



Existing Stock

	Eraser	Books	Pens
Basket 1	0	0	0
Basket 2	10	10	10
Basket 3	20	20	20
Basket 4	30	30	30

(4,3)

+

Let’s say you get a delivery – no more erasers,
1 book for each basket and
2 pens for each basket

0	1	2
0	1	2
0	1	2
0	1	2

(3,)

=

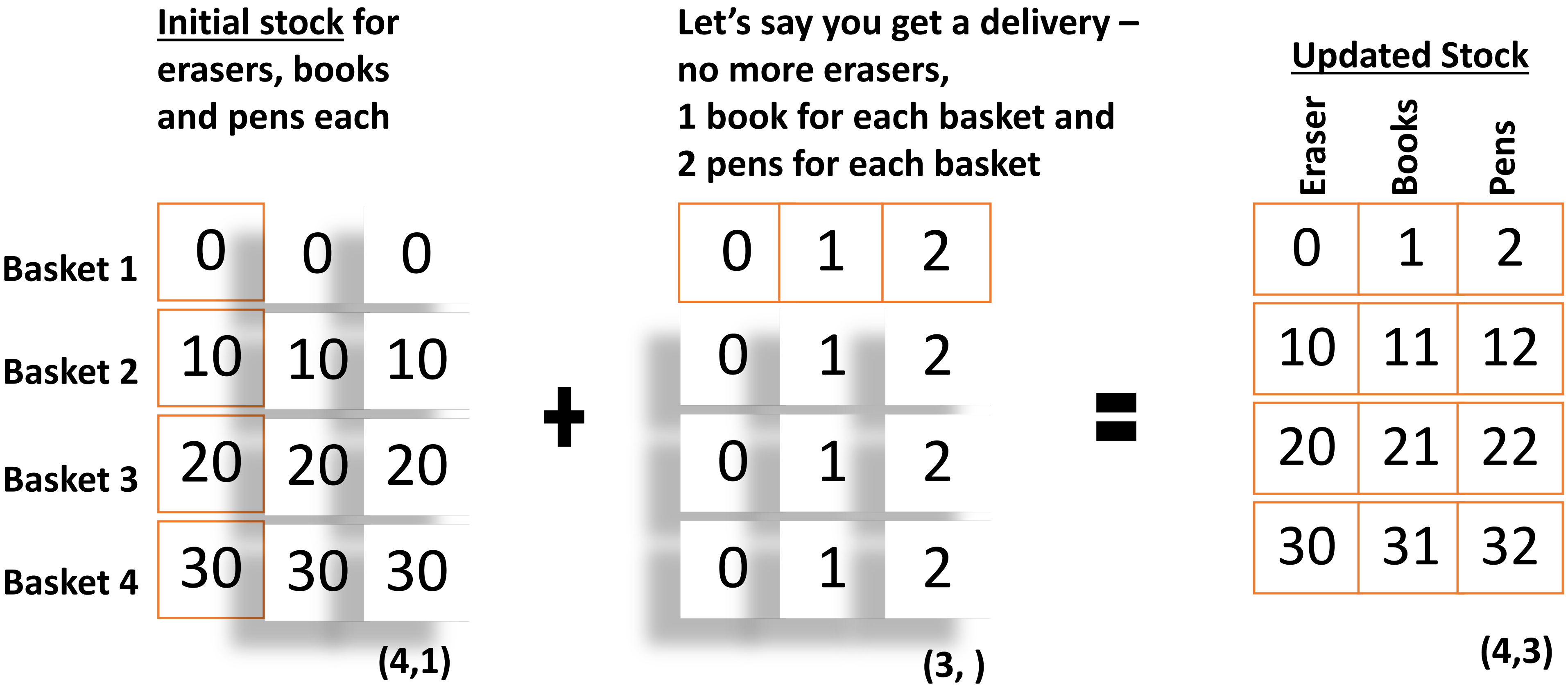
Updated Stock

Eraser	Books	Pens
0	1	2
10	11	12
20	21	22
30	31	32

(4,3)

Broadcasting (Cont.)

- RULE 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.



Initial stock for erasers, books and pens each

Basket 1	0	0	0
Basket 2	10	10	10
Basket 3	20	20	20
Basket 4	30	30	30

(4,1)

+

Let’s say you get a delivery – no more erasers,
1 book for each basket and
2 pens for each basket

0	1	2
0	1	2
0	1	2
0	1	2

(3,)

=

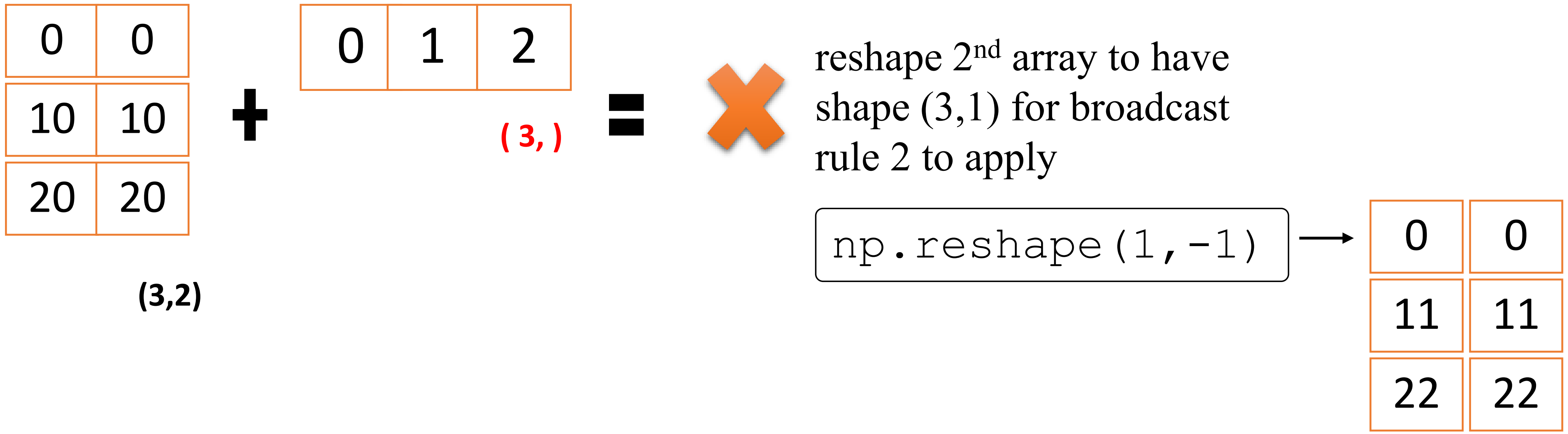
Updated Stock

Eraser	Books	Pens
0	1	2
10	11	12
20	21	22
30	31	32

(4,3)

Broadcasting (Cont.)

- RULE 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.



0	0
10	10
20	20

(3,2)

+

0	1	2

(3,)

=

✗

reshape 2nd array to have
shape (3,1) for broadcast
rule 2 to apply

```
np.reshape(1,-1)
```

→

0	0
11	11
22	22

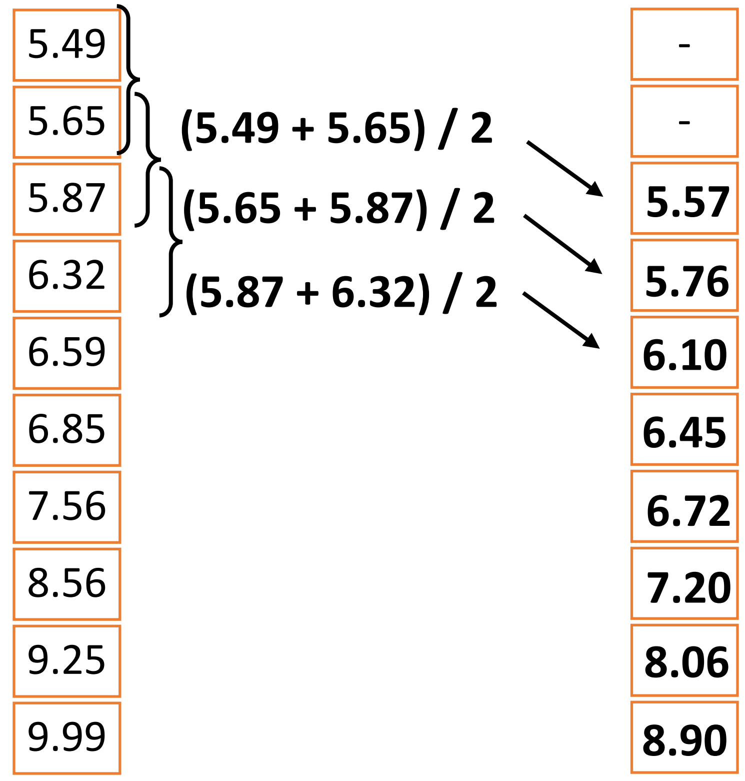
Broadcasting (Cont.)

- In fact, many examples in previous sections are all using broadcasting technique.
- Being able to write short codes is not the only reason for broadcasting in NumPy.
- Broadcasting often makes array operation faster.

Iteration

daily price
of a stock

Simple model that predict today's price as
the average of the price of the last 2 days



**N-
days**

**N-2
days**

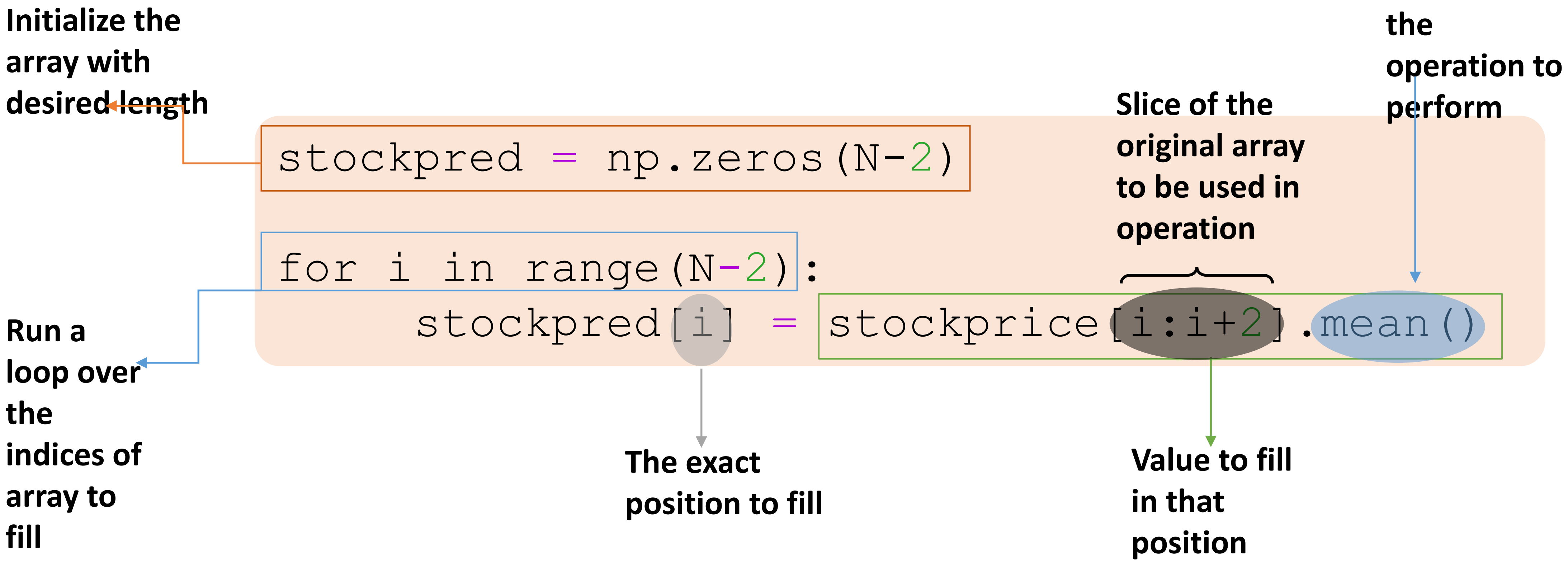
How do we do this in NumPy?

```
stockpred[0] = stockprice[0:2].mean()
stockpred[1] = stockprice[1:3].mean()
stockpred[2] = stockprice[2:4].mean()
stockpred[3] = stockprice[3:5].mean()
```

```
stockpred[i] = stockprice[i:i+2].mean()
```

Iteration(Cont.)

Simple model that predicts today’s price as the average of the price of the last 2 days



Initialize the array with desired length

```
stockpred = np.zeros(N-2)

for i in range(N-2):
       stockpred[i] = stockprice[i:i+2].mean()
```

Slice of the original array to be used in operation

the operation to perform

Run a loop over the indices of array to fill

The exact position to fill

Value to fill in that position

This Week

1. **Object-Oriented Programming (OOP)**
   - Class and Object
   - Class Attributes and Methods
2. **NumPy**
   - ndarray
   - Numerical Computing with NumPy

Additional sources for beginner Python programmers:
- NumPy official webpage https://numpy.org/
- NumPy official documentation https://docs.scipy.org/doc/numpy/reference/
- SciPy Lectures https://scipy-lectures.org/
- Python for data Analysis http://shop.oreilly.com/product/0636920023784.do
- Guide to NumPy https://web.mit.edu/dvp/Public/numpybook.pdf
- Python Data Science Handbook https://jakevdp.github.io/PythonDataScienceHandbook/

Next Weeks

1. **Pandas**

2. **Matplotlib**

3. **Linear Algebra in Python**

4. **…**