Python_review

March 4, 2025

0.1 List Comprehensions

It is an elegant way to create the lists in Python. To see how useful the list comprehensions is, consider the following example:

1. Create a list containing numbers form 1,…,5.
2. Create another list of numbers containing square of numbers form 1,…,5.
3. Repeat the above two tasks for numbers 1,…,50.
4. Create a list, say eSquare, which has only the squares of even numbers form 1,…,50. Create a new list, say PV, which has '+' for even numbers and '-' for odd numbers for numbers from 1,…,50.
5. Find the length of eSquare.
6. Find the position of 196 in eSquare.
7. Check if number 576 is inside eSquare.
8. Create two new variable, containing the max and min numbers inside eSquare.
9. Swap the values of the above two variables.
10. Remove 196 from eSquare.

[89]:
```
# 1. Create a list containing numbers form 1,...,5.
l=[1,2,3,4,5]
```

[1]:
```
# 2. Create another list of numbers containing square of numbers form 1,...,5.
l=[1,4,9,16,25]
```

[2]:
```
# 3. Repeat the above two tasks for numbers 1,...,50.
numbers=list(range(1,51))
print(numbers)


Square=[x**2 for x in numbers]
print(Square)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324,
361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089,
1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025, 2116,
2209, 2304, 2401, 2500]
```

[3]:
```
# 4. Create a list, say eSquare, which has only the squares of even numbers␣
 ↪form 1,...,50.
eSquare = [x**2 for x in numbers if x % 2 == 0]
print(eSquare)


# Create a new list, say PV, which has '+' for even numbers and '-' for odd␣
 ↪numbers for numbers from 1,...,50.
PV = ['+' if x % 2 == 0 else '-' for x in numbers]
print(PV)
```

```
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784, 900, 1024,
1156, 1296, 1444, 1600, 1764, 1936, 2116, 2304, 2500]
['-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+',
'-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+',
'-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+', '-', '+',
'-', '+']
```

[4]:
```
# 5. Find the length of eSquare.
print(len(eSquare))
```

```
25
```

[5]:
```
# 6. Find the position of 196 in eSquare.
print(eSquare.index(196))
```

```
6
```

[6]:
```
# 7. Check if number 576 is inside eSquare.
print(576 in eSquare)
```

```
True
```

[7]:
```
# 8. Create two new variable, containing the max and min numbers inside eSquare.
maxValue, minValue = max(eSquare), min(eSquare)
print(maxValue, minValue )
```

```
2500 4
```

[8]:
```
# 9. Swap the values of the above two variables.
maxValue, minValue = minValue, maxValue
print(maxValue, minValue )
```

```
4 2500
```

[9]:
```
# 10. Remove 196 from eSquare.
eSquare.remove(196)
# eSquare.pop(eSquare.index(196))
print(eSquare)
```

```
[4, 16, 36, 64, 100, 144, 256, 324, 400, 484, 576, 676, 784, 900, 1024, 1156,
1296, 1444, 1600, 1764, 1936, 2116, 2304, 2500]
```

0.2 Enumerate

It is useful, when you have a list over which you want to iterate, as well as keep track of index. To see how it works, consider the following:

1. Consider the following list:
   ['Apple', 'Banana', 'Orange', 'Watermelon', 'Plum', 'Grapes', 'Kiwi', 'Strawberry', 'Pear']
2. Sort the above list in ascending alphabetical order.
3. To the sorted list, for each element, prefix the value by its position.
   - For example, the final list should look like:
   - ['1.Apple',..., '9.Watermelon']

[10]:
```
# 1. Consider the following list:
fruits = ['Apple', 'Banana', 'Orange', 'Watermelon', 'Plum', 'Grapes', 'Kiwi',␣
 ↪'Strawberry', 'Pear']
```

[11]:
```
# 2. Sort the above list in ascending alphabetical order.
fruits.sort()
print("sorted: ", fruits)

# or
fruits2 = sorted(fruits)
print("sorted: ", fruits2)
```

```
sorted:  ['Apple', 'Banana', 'Grapes', 'Kiwi', 'Orange', 'Pear', 'Plum',
'Strawberry', 'Watermelon']
sorted:  ['Apple', 'Banana', 'Grapes', 'Kiwi', 'Orange', 'Pear', 'Plum',
'Strawberry', 'Watermelon']
```

[12]:
```
# 3. To the sorted list, for each element, prefix the value by its position.

listFruits = [str(i+1)+'.'+v for i,v in enumerate(fruits)]
print(listFruits)


# i=1
# fruits2=[]
# for x in fruits:
#     fruits2.append(str(i)+'.'+x)
#     i+=1
# print(fruits2)


# fruits2=[]
# for x in fruits:
```

```
#     fruits2.append(str(fruits.index(x)+1)+'.'+x)
# print(fruits2)


# fruits2=[]
# for i in range(len(fruits)):
#     fruits2.append(str(i)+'.'+fruits[i])

# fruits2=[]
# for i,x in enumerate(fruits):
#     fruits2.append(str(i+1)+'.'+x)
# print(fruits2)
```

```
['1.Apple', '2.Banana', '3.Grapes', '4.Kiwi', '5.Orange', '6.Pear', '7.Plum',
'8.Strawberry', '9.Watermelon']
```

0.3 Zip

It is useful, when you have to join two lists. It creates a zip object, from which a list of tuple can be obtained. Such lists are very handy in looping over two or more lists simultaneously.

1. Consider the following list:
   [[‘Apple’, ‘Banana’, ‘Orange’], [‘Watermelon’, ‘Plum’, ‘Grapes’, ‘Kiwi’], [‘Strawberry’, ‘Pear’, ‘Mango’]]
2. Create a new list containing all the ‘fruits’.
3. Create a new list, say zipZap, containing tuple of fruits name followed by length of the name.
   - For example, the final list should look like:
   - [(‘Apple’, 5), ..., (‘Watermelon’, 10)]
4. Iterate over zipZap and print all the elements.
5. Iterate over zipZap and print the index and all the corresponding elements.

[13]:
```
# 1. Consider the following list:
#     [['Apple', 'Banana', 'Orange'], ['Watermelon', 'Plum', 'Grapes', 'Kiwi'],␣
 ↪['Strawberry', 'Pear', 'Mango']]
# 2. Create a new list containing all the 'fruits'.

fruits = [['Apple', 'Banana', 'Orange'], ['Watermelon', 'Plum', 'Grapes',␣
 ↪'Kiwi'], ['Strawberry', 'Pear', 'Mango']]

allFruits = [y for innerList in fruits for y in innerList]
print(allFruits)

# ## or the following style
# allFruits = []
# for innerList in fruits:
#     for y in innerList:
#         allFruits.append(y)
```

```
# print(allFruits)
```

```
['Apple', 'Banana', 'Orange', 'Watermelon', 'Plum', 'Grapes', 'Kiwi',
'Strawberry', 'Pear', 'Mango']
```

[14]:
```
lenFruits = [len(v) for v in allFruits]
print(lenFruits)
```

```
[5, 6, 6, 10, 4, 6, 4, 10, 4, 5]
```

[15]:
```
# 3. Create a new list, say zipZap, containing tuple of fruits name followed by␣
 ↪length of the name.

zipZap = list(zip(allFruits,lenFruits))
print(zipZap)
```

```
[('Apple', 5), ('Banana', 6), ('Orange', 6), ('Watermelon', 10), ('Plum', 4),
('Grapes', 6), ('Kiwi', 4), ('Strawberry', 10), ('Pear', 4), ('Mango', 5)]
```

[16]:
```
# 4. Iterate over zipZap and print all the elements.

for k in zipZap:
    print(k[0],k[1])

## Another unpacking approach that gives similar results
# for k0,k1 in zipZap:
#     print(k0,k1)
```

```
Apple 5
Banana 6
Orange 6
Watermelon 10
Plum 4
Grapes 6
Kiwi 4
Strawberry 10
Pear 4
Mango 5
```

[17]:
```
# 5. Iterate over zipZap and print the index and all the corresponding elements.

for i,k in enumerate(zipZap):
    print(i,k[0],k[1])

## Another unpacking approach that gives similar results
# for i,(k0,k1) in enumerate(zipZap):
#     print(i,k0,k1)
```

```
0 Apple 5
1 Banana 6
```

```
2 Orange 6
3 Watermelon 10
4 Plum 4
5 Grapes 6
6 Kiwi 4
7 Strawberry 10
8 Pear 4
9 Mango 5
```

0.4 Dictionaries

Dictionaries are used to store data values in key:value pairs. For example, consider the following lists: - List of fruits:
['Apple', 'Banana', 'Orange', 'Watermelon', 'Plum', 'Grapes', 'Kiwi', 'Strawberry', 'Pear', 'Mango']
- List of corresponding prices per KG:
[47, 27, 35, 13, 28, 10, 30, 56, 15, 25]

Do the following: 1. Create a dictionary, say book1, which has fruits as keys, and prices as values. 2. Loop over all the key & value pairs of book1. 3. Create a dictionary, say book2, which has fruits as values, and unique numbers as keys. 4. Display book2 and ask the user to pick a number. Then display the price of the fruit using book1.

[18]:
```
# given lists
fruits=['Apple', 'Banana', 'Orange', 'Watermelon', 'Plum', 'Grapes', 'Kiwi',␣
 ↪'Strawberry', 'Pear', 'Mango']
prices=[47, 27, 35, 13, 28, 10, 30, 56, 15, 25]
```

[19]:
```
# 1. Create a dictionary, say book1, which has fruits as keys, and prices as␣
 ↪values.

book1 = dict(zip(fruits,prices))
print(book1)
```

```
{'Apple': 47, 'Banana': 27, 'Orange': 35, 'Watermelon': 13, 'Plum': 28,
'Grapes': 10, 'Kiwi': 30, 'Strawberry': 56, 'Pear': 15, 'Mango': 25}
```

[20]:
```
# 2. Loop over all the key & value pairs of book1.

for k,v in book1.items():
    print(f'The price of {k} is {v} SAR.')
```

```
The price of Apple is 47 SAR.
The price of Banana is 27 SAR.
The price of Orange is 35 SAR.
The price of Watermelon is 13 SAR.
The price of Plum is 28 SAR.
The price of Grapes is 10 SAR.
The price of Kiwi is 30 SAR.
The price of Strawberry is 56 SAR.
```

```
The price of Pear is 15 SAR.
The price of Mango is 25 SAR.
```

[21]:
```python
# 3. Create a dictionary, say book2, which has fruits as values, and unique␣
 ↪numbers as keys.

book2 = dict(zip(list(range(len(fruits))),fruits))
print(book2)
```

```
{0: 'Apple', 1: 'Banana', 2: 'Orange', 3: 'Watermelon', 4: 'Plum', 5: 'Grapes',
6: 'Kiwi', 7: 'Strawberry', 8: 'Pear', 9: 'Mango'}
```

[22]:
```python
# 4. Display book2 and ask the user to pick a number. Then display the price of␣
 ↪the fruit using book1.
for k,v in book2.items():
    print(f'{k}. {v}')
itemSelected = input('Enter the number corresponding to a fruit to know its␣
 ↪price.')
itemSelected  = int(itemSelected)

# priceSelected = book1[book2[itemSelected]]  # compact and better approach

fruitSelected = book2[itemSelected]
priceSelected = book1[fruitSelected]

width = len(f'*The price per KG of {fruitSelected} is {priceSelected}.*')

print('\n'*4)
print('*'*width)
print(f'{f"*You have selected {fruitSelected}.":<{width-1}}*')  #f strings can␣
 ↪be nested
print(f'*The price per KG of {fruitSelected} is {priceSelected}.*')
print('*'*width)


####
# f'{string: <{width}}' is used for formatting (align, width) while printing␣
 ↪strings
#to align the string to the right use >, and to the left use <
#example: f'{s:>5}' implies right align string with space of 5 chars
```

```
0. Apple
1. Banana
2. Orange
3. Watermelon
4. Plum
5. Grapes
6. Kiwi
```

```
7. Strawberry
8. Pear
9. Mango
```

```
**********************************
*You have selected Apple.        *
*The price per KG of Apple is 47.*
**********************************
```

Things to remember about keys: - Duplicate keys are not allowed. That is, a given key can appear in a dictionary only once. - Dictionary keys can be of any type, but they must be immutable. - Re-assigning a new value to an existing key, will override the first/previous value.

0.5 String Methods

Out of the box, Python has inbuilt functions that are very handy in text processing. For example, consider the following text:

we are reviewing python programming topics in ise 291. it is the most popular language for data scientists. also, it is a good general purpose programming language.

Do the following: 1. Breakdown the given text into list of sentences. 2. Capitalize the first word in each sentence. 3. Replace 'ise' with 'ISE" everywhere. 4. Join all the above sentences into a new text. 5. Print the new text.

[23]:
```
text = """we are reviewing python programming topics in ics 574.
it is the most popular language for data scientists.
also, it is a good general purpose programming language."""
```

[24]:
```
sentences = text.split('.')
sentences.pop(-1)   # to get rid to empty last element
print(sentences)
```

```
['we are reviewing python programming topics in ics 574', '\nit is the most
popular language for data scientists', '\nalso, it is a good general purpose
programming language']
```

[25]:
```
sentences1=[s.strip().capitalize().replace('ics','ICS')+'.'
           for s in sentences]
print(sentences1)

## other variations
# sentences2=[]
# for s in sentences:
#     sentences2.append(s.strip().capitalize().replace('ise','ISE')+'.')
# print(sentences2)
```

```
## or
# sentences3=[]
# for s in sentences:
#     temp=s.strip()
#     temp=temp.capitalize()
#     temp=temp.replace('ise','ISE')
#     temp=temp+'.'
#     sentences3.append(temp)
# print(sentences3)
```

```
['We are reviewing python programming topICS in ICS 574.', 'It is the most
popular language for data scientists.', 'Also, it is a good general purpose
programming language.']
```

[26]:
```
newText="\n".join(sentences1)
print(newText)
```

```
We are reviewing python programming topICS in ICS 574.
It is the most popular language for data scientists.
Also, it is a good general purpose programming language.
```

1 Lambda & Map

Beyond basic python

The map and lambda functions extends the ability of Python to perform complex operations using a compact & simple style. 1. Python's **lambda()** function is a small **anonymous function**, which can take any number of arguments, but can only have one expression. 2. Python's **map()** function apply one function to each element of an **iterable** like list, tuple in Python.

Consider the following examples:

[27]:
```
# Lambda function with one argument

print((lambda x : x**2)(2))
```

```
4
```

[28]:
```
# Lambda function with multiple argument
print((lambda x, y, z : x + y * z)(5, 6, 2))
```

```
17
```

[29]:
```
# Lambda function with conditional statement
# check even or odd
print((lambda x : f"{x} is an even number." if x%2==0 else f"{x} is an odd␣
 ↪number.")(7))
```

```
7 is an odd number.
```

[30]:
```
# Map function
def squareFunction(x):
    return x**2

numbers = [1, 2, 3, 4, 5]

squares = list(map(squareFunction, numbers))

print(squares)
```

```
[1, 4, 9, 16, 25]
```

[31]:
```
# Lambda & Map functions with single lists
numbers = [1, 2, 3, 4, 5]

squares = list(map((lambda x:x**2), numbers))

print(squares)
```

```
[1, 4, 9, 16, 25]
```

[32]:
```
# Lambda & Map functions with multiple lists

list1 = [1, 2, 3, 4, 5]
list2 = [5, 5, 3, 5, 5]

listSub = list(map((lambda x,y:x-y), list1,list2))
print(listSub)


listSub = list(map((lambda x,y:x-y), list2,list1))
print(listSub)
```

```
[-4, -3, 0, -1, 0]
[4, 3, 0, 1, 0]
```

[33]:
```
# Lambda & Map functions with conditions
numbers = list(range(11))

squares1 = list(map((lambda x:x**2 if x%2==0 else None), numbers))
print(squares1)


squares2 = list(map((lambda x:x**2), [x for x in numbers if x%2==0]))
print(squares2)
```

```
[0, None, 4, None, 16, None, 36, None, 64, None, 100]
[0, 4, 16, 36, 64, 100]
```

2 Numpy Library

A must for scientific work

[34]:
```
import numpy as np  #to import the libary
```

[35]:
```
# !pip instal numpy
```

2.1 Creating Numpy Arrays

Do the following tasks: 1. Create an integer array of 10 elements filled with all zeros. 2. Create a 3x5 floating-point array filled with ones. 3. Create a 3x5 array filled with pi values. 4. Create a 3x3 array of uniformly distributed random values between 0 and 1. 5. Create a 3x3 array of random integers in the interval [0, 10). 6. Create a 3x3 identity matrix. 7. Create an array of 10 elements uniformly dividing the interval [0,1], i.e, 0, 0.1,..., 0.9, 1.

[36]:
```
#1. Create an integer array of 10 elements filled with all zeros.
np.zeros(10, dtype="int")

## another way
# list1 = [0 for x in range(10)]
# print(list1)
# print(type(list1))

# print('-'*8)
# array1 = np.array(list1)
# print(array1)
# print(type(array1))
```

[36]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[37]:
```
#2. Create a 3x5 floating-point array filled with ones.
np.ones((3,5), dtype="int")

# or the default
# np.ones((3,5))
```

[37]:
```
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

[38]:
```
#3. Create a 3x5 array filled with pi values.
print(np.full((3,5), np.round(np.pi,3)))

## or the following
# print(np.full((3,5), np.round(np.pi,3)))  # numpy functions are Universal␣
↪Functions (ufuncs) more on this in Sec 3.10.
```

```
[[3.142 3.142 3.142 3.142 3.142]
 [3.142 3.142 3.142 3.142 3.142]
 [3.142 3.142 3.142 3.142 3.142]]
```

[39]:
```
#4. Create a 3x3 array of uniformly distributed random values between 0 and 1.
np.random.seed(0) # seed for reproducibility
print(np.random.random((3,3)))

# print(np.random.rand(3,3))

# or the following with specific decimal places
# np.random.seed(0) # seed for reproducibility
# print(np.round(np.random.random((3,3)),2)) #
```

```
[[0.5488135  0.71518937 0.60276338]
 [0.54488318 0.4236548  0.64589411]
 [0.43758721 0.891773   0.96366276]]
```

[40]:
```
#5. Create a 3x3 array of random integers in the interval [0, 10).
np.random.randint(0,10,(3,3))
```

[40]:
```
array([[1, 6, 7],
       [7, 8, 1],
       [5, 9, 8]])
```

[41]:
```
#6. Create a 3x3 identity matrix.
np.eye(3)   # default data type is float
```

[41]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[42]:
```
# 7. Create an array of 11 elements uniformly dividing the interval [0,1], i.e,␣
↪0. , 0.1,..., 0.9, 1.
np.linspace(0,1,11)
```

[42]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

2.2 Numpy Array Attributes

Numpy array have the following attributes: 1. ndim (the number of dimensions), 2. shape (the size of each dimension), and 3. size (the total size of the array).

For example, do the following: 1. Create a random one-dimensional array (x1), and check all the above attributes. 2. Create a random two-dimensional array (x2), and check all the above attributes.

[43]:
```
# 1. Create a random one-dimensional array (x1), and check all the above␣
↪attributes
```

```
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10,size=6) #it's same ((np.random.randint((0,10),␣
 ↪size=6))) # One-dimensional array
print(x1)
print(f"x1 ndim: {x1.ndim}")
print(f"x1 shape: {x1.shape}")
print(f"x1 size: {x1.size}") #totaly,6 elements
```

```
[5 0 3 3 7 9]
x1 ndim: 1
x1 shape: (6,)
x1 size: 6

x1 ndim: 1
x1 shape: (6,)
x1 size: 6
```

[44]:
```
# 2. Create a random two-dimensional array (x2), and check all the above␣
 ↪attributes.
np.random.seed(0) # seed for reproducibility
x2 = np.random.randint(10,size=(3,4)) # Two-dimensional array
print(x2)
print(f"x2 ndim: {x2.ndim}")
print(f"x2 shape: {x2.shape}")
print(f"x2 size: {x2.size}") #totaly,12 elements
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
x2 ndim: 2
x2 shape: (3, 4)
x2 size: 12
```

2.3 Indexing

Numpy array indexing for 1d array is same as Python's lists. However, Numpy nd-array indexing is slightly different from Python's nd-lists. For example: 1. 2d-list elements are accessed as:[index1][index2] 2. 2d-array elements are accessed as:[index1,index2]

where index1 and index2 can be indices or slices. See the following examples:

[45]:
```
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) # One-dimensional array
print(x1)
print(f'indices:{list(range(6))}')
print(f'array  :{x1.tolist()}')   #.tolist() converts numpy array into python's␣
 ↪list. The conversion is not permanant.

print('-'*10);
```

```
print(x1[0],x1[4],x1[-1],x1[-2])
```

```
[5 0 3 3 7 9]
indices:[0, 1, 2, 3, 4, 5]
array  :[5, 0, 3, 3, 7, 9]
----------
5 7 9 7
```

[46]:
```
np.random.seed(0) # seed for reproducibility
x2 = np.random.randint(10, size=(3,4)) # Two-dimensional array

print(x2)

print('-'*10);


print(x2[2,1],x2[2,0],x2[2,-4],x2[-2,-3])
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
----------
4 2 2 9
```

[47]:
```
# Updating values
x1[0] = 3.14159 # this will be truncated!
print(x1)
```

```
[3 0 3 3 7 9]
```

[48]:
```
# Updating values
x2[0,0]=12
print(x2)
```

```
[[12  0  3  3]
 [ 7  9  3  5]
 [ 2  4  7  6]]
```

2.4 Slicing

Slicing refer to selecting/extracting a sub-array from a given array. As an example for 1d arrays, do the following tasks: 1. Create a random 1d numpy array of size 10. 2. Display the first five elements. 3. Display elements from index 5 to end. 4. Display elements from index 4 to 6. 5. Display alternate elements from the first element. 6. Display alternate elements starting from the second element. 7. Display elements starting from fourth element from the end and ending at last but one element. 8. Display all element in the reverse order of indices.

[49]:
```
# 1. Create a random 1d numpy array of size 10.
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10,size=10) # One-dimensional array
```

```
print(f'indices:{list(range(10))}')
print(f'array  :{x1.tolist()}')
```

```
indices:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
array  :[5, 0, 3, 3, 7, 9, 3, 5, 2, 4]
```

[50]:
```
# 2. Display the first five elements.
x1[0:5]
```

[50]:
```
array([5, 0, 3, 3, 7])
```

[51]:
```
# 3. Display elements from index 5 to end.
x1[5:]
```

[51]:
```
array([9, 3, 5, 2, 4])
```

[52]:
```
# 4. Display elements from index 4 to 6.
x1[4:7]
```

[52]:
```
array([7, 9, 3])
```

[53]:
```
# 5. Display alternate elements starting from the first element.
x1[::2]
```

[53]:
```
array([5, 3, 7, 3, 2])
```

[54]:
```
# 6. Display alternate elements starting from the second element.
x1[1::2]
```

[54]:
```
array([0, 3, 9, 5, 4])
```

[55]:
```
# 7. Display elements starting from fourth element from the end and ending at␣
 ↪last but one element.
x1[-4:-1]
```

[55]:
```
array([3, 5, 2])
```

[56]:
```
# 8. Display all element in the reverse order of indices.
x1[::-1]
```

[56]:
```
array([4, 2, 5, 3, 9, 7, 3, 3, 0, 5])
```

As an example for 2d arrays, do the following tasks: 1. Create a random 2d numpy array of size 3x4. 2. Display first two rows, first three columns 3. Display all rows, and every other column 4. Display all rows and columns in the reverse order 5. Display first column 6. Display first row 7. Display third row elements in order of column 3, then column 1 then column 2 8. Display second and following row elements in order of column 3, then column 1 then column 2

[57]:
```
# 1. Create a random 2d numpy array of size 3x4.
np.random.seed(0) # seed for reproducibility
x2 = np.random.randint(10, size=(3,4)) # Two-dimensional array
print(x2)
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

[58]:
```
# 2. Display first two rows, first three columns
x2[:2, :3]
```

[58]:
```
array([[5, 0, 3],
       [7, 9, 3]])
```

[59]:
```
# 3. Display all rows, and every other column
x2[:3,::2]
```

[59]:
```
array([[5, 3],
       [7, 3],
       [2, 7]])
```

[60]:
```
# 4. Display all rows and columns in the reverse order
x2[::-1,::-1]
```

[60]:
```
array([[6, 7, 4, 2],
       [5, 3, 9, 7],
       [3, 3, 0, 5]])
```

[61]:
```
# 5. Display first column
print(x2[:, 0])
```

```
[5 7 2]
```

[62]:
```
# 6. Display first row
print(x2[0,:])

# print(x2[0]) # equivalent to x2[0, :]
```

```
[5 0 3 3]
```

[63]:
```
# 7. Display third row elements in order of column 3, then column 1 then column␣
 ↪2
x2[2,[2,0,1]]
```

[63]:
```
array([7, 2, 4])
```

[64]:
```
# 8. Display second and following row elements in order of column 3, then␣
 ↪column 1 then column 2
```

```
x2[1:, [2, 0, 1]]
```

[64]:
```
array([[3, 7, 9],
       [7, 2, 4]])
```

2.5 Sorting

Numpy *.sort()* method sorts an array in-place. The sorting is done row/column wise, and order/preference can be provided too. As an example, do the following: 1. Create a 1d random array of size 6, and sort it. Print the corresponding original indices in the sorted array. 2. Create a random 2d numpy array of size 3x4. Sort row wise. 3. Create a random 2d numpy array of size 3x4. Sort column wise.

[65]:
```
# 1. Create a 1d random array of size 6, and sort it. Print the corresponding␣
 ↪original indices in the sorted array.
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10,size=6) # One-dimensional array

print(f'original indices:{list(range(6))}')
print(f'given    array  :{x1.tolist()}')
print(f'sorted-array    :{np.sort(x1).tolist()}') # Ascending
print(f'sorted indices  :{np.argsort(x1).tolist()}')
print(f'sorted-array    :{np.sort(x1)[::-1].tolist()}') #Descending
```

```
original indices:[0, 1, 2, 3, 4, 5]
given    array  :[5, 0, 3, 3, 7, 9]
sorted-array    :[0, 3, 3, 5, 7, 9]
sorted indices  :[1, 2, 3, 0, 4, 5]
sorted-array    :[9, 7, 5, 3, 3, 0]
```

[66]:
```
# 2. Create a random 2d numpy array of size 3x4. Sort row wise.
np.random.seed(0) # seed for reproducibility
x2 = np.random.randint(10, size=(3,4)) # Two-dimensional array
print(x2)
print('-'*10)
x2.sort(axis=0)
print(x2)
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
----------
[[2 0 3 3]
 [5 4 3 5]
 [7 9 7 6]]
```

[67]:
```
#3. Create a random 2d numpy array of size 3x4. Sort column wise.
np.random.seed(0) # seed for reproducibility
x2 = np.random.randint(10, size=(3,4)) # Two-dimensional array
```

```
print(x2)
print('-'*10)
x2.sort(axis=1)
print(x2)
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
----------
[[0 3 3 5]
 [3 5 7 9]
 [2 4 6 7]]
```

2.6 Copy of Array

Arrays are mutable too. Thus, the assignment "=" is by reference. Use ".copy()" to copy the entire numpy array. For example, see the code in the following cells:

[68]:
```
np.random.seed(0) # seed for reproducibility
x2 = np.random.randint(10, size=(2,4)) # Two-dimensional array
print(f'x2 is:\n{x2}')
print('-'*10)

x2_copy = x2  # new reference not actual copy
print(f'x2_copy is:\n{x2_copy}')
print('-'*10)

x2_copy[0,1:3]=[4,4]
print(f'x2_copy modified is:\n{x2_copy}')
print('-'*10)

print(f'x2 is:\n{x2}')
## x2_copy = x2[:,:]
```

```
x2 is:
[[5 0 3 3]
 [7 9 3 5]]
----------
x2_copy is:
[[5 0 3 3]
 [7 9 3 5]]
----------
x2_copy modified is:
[[5 4 4 3]
 [7 9 3 5]]
----------
x2 is:
[[5 4 4 3]
 [7 9 3 5]]
```

[69]:
```
np.random.seed(0) # seed for reproducibility
x2 = np.random.randint(10, size=(2,4)) # Two-dimensional array
print(f'x2 is:\n{x2}')
print('-'*10)

x2_copy = x2.copy()  # it will be true copy
print(f'x2_copy is:\n{x2_copy}')
print('-'*10)

x2_copy[0,1:3]=[4,4]
print(f'x2_copy modified is:\n{x2_copy}')
print('-'*10)

print(f'x2 is:\n{x2}')
```

```
x2 is:
[[5 0 3 3]
 [7 9 3 5]]
----------
x2_copy is:
[[5 0 3 3]
 [7 9 3 5]]
----------
x2_copy modified is:
[[5 4 4 3]
 [7 9 3 5]]
----------
x2 is:
[[5 0 3 3]
 [7 9 3 5]]
```

2.7 Reshaping

Numpy *.reshape* method gives a new shape to an existing array without changing its data. As an exmaple, do the following task: 1. Create a 1d random array of size 10. 2. Reshape the 1d array into size 5x2. 3. Reshape the 1d array into size 2x5.

[70]:
```
# 1. Create a 1d random array of size 10.
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10,size=10) # One-dimensional array
print(x1)
```

```
[5 0 3 3 7 9 3 5 2 4]
```

[71]:
```
# 2. Reshape the 1d array into size 5x2.
print(x1.reshape(5,2))
```

```
[[5 0]
 [3 3]
```

```
 [7 9]
 [3 5]
 [2 4]]
```

[72]:
```
# 3. Reshape the 1d array into size 2x5.
print(x1.reshape(5,-1))

## when you want numpy to estimate the other dimension
# print(x1.reshape(2,-1))
# print(x1.reshape(-1,5))
```

```
[[5 0]
 [3 3]
 [7 9]
 [3 5]
 [2 4]]
```

2.8 Concatenate/Split

Concatenate means joining, and Numpy's *.concatenate()* function is used to join two or more arrays of the same shape along a specified axis. Numpy's *.vstack()* (*.hstack()*) can be used for row wise (column wise) joining. Numpy's split/vsplit/hsplit methods are for breaking/splitting the array, opposite to the concatenation.

For example, see the code in the following cells:

[73]:
```
x = np.array([1,2,3])
y = np.array([3,2,1])
z = np.array([9,9,9])

print(np.concatenate((x,y,z)))   # to join all 1d arrays along axis 0
```

```
[1 2 3 3 2 1 9 9 9]
```

[74]:
```
x = np.array([1,2,3])
y = np.array([3,2,1])
z = np.array([9,9,9])

print(np.vstack((x,y,z)))  # to vertically stack all 1d arrays
```

```
[[1 2 3]
 [3 2 1]
 [9 9 9]]
```

[75]:
```
x = np.array( [1,2,3])
y = np.array([[9,8,7],
              [6,5,4]])

np.vstack([x,y])  # vertically stack the arrays
```

[75]:
```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

[76]:
```
y = np.array([[9,8,7],
              [6,5,4]])

z = np.array([[99],
              [99]])

np.hstack([z,y])  # horizontally stack the arrays
```

[76]:
```
array([[99,  9,  8,  7],
       [99,  6,  5,  4]])
```

[77]:
```
# An example to split array into three sub-arrays using the given break points

x = [1,2,3,99,99,3,2,1]
print(x)

x1, x2, x3 = np.split(x, [3,5])
print(x1, x2, x3)
```

```
[1, 2, 3, 99, 99, 3, 2, 1]
[1 2 3] [99 99] [3 2 1]
```

[78]:
```
# An example to split 2d-array into three sub-arrays using the given break␣
 ↪points

x = np.array([[1,2,3,99],[99,3,2,1]])
print(x)
print('-'*16)

#  Split along the rows
x1, x2 = np.split(x, 2, axis=0)
print(x1, x2)
print('-'*16)

#  Split along the columns
x1, x2 = np.split(x, 2, axis=1)
print(x1, x2)
```

```
[[ 1  2  3 99]
 [99  3  2  1]]
----------------
[[ 1  2  3 99]] [[99  3  2  1]]
----------------
[[ 1  2]
```

```
[99  3]] [[ 3 99]
 [ 2  1]]
```

2.9 Vectorized Operations

Vectorized operations are perhaps the most crucial factor to the wide usage of Numpy library. Vectorized operations simply put are those **operations that can be done on arrays without using loops**.

For example, do the following: 1. Create a random id array, and add, subtract,multiply and divide all the elements with 5. 2. Take the square of all elements. 3. Find remainder w.r.t 2 for all elements. 4. Transform the array as: $-(\frac{x}{2}+1)^2$, where x is the random 1d array.

[79]:
```
# 1. Create a random id array, and add, subtract,multiply and divide all the␣
 ↪elements with 5.

np.random.seed(0) # seed for reproducibility
x = np.random.randint(10,size=10) # One-dimensional array
print(f"x = {x}")

print(f"x + 5 = {x + 5}") # adding 5 to each element
print(f"x - 5 = {x - 5}") # subtracting 5 to each element
print(f"x * 5 = {x * 5}") # multiply each element by 5
print(f"x / 5 = {x / 5}") # divide each element by 5
print(f"x // 5 = {x // 5}") # floor division each element by 5
```

```
x = [5 0 3 3 7 9 3 5 2 4]
x + 5 = [10  5  8  8 12 14  8 10  7  9]
x - 5 = [ 0 -5 -2 -2  2  4 -2  0 -3 -1]
x * 5 = [25  0 15 15 35 45 15 25 10 20]
x / 5 = [1.  0.  0.6 0.6 1.4 1.8 0.6 1.  0.4 0.8]
x // 5 = [1 0 0 0 1 1 0 1 0 0]
```

[80]:
```
# 2. Take the square of all elements.
# 3. Find remainder w.r.t 2 for all elements.
print(f"x ** 2 =  {x ** 2}")
print(f"x % 2 =  {x % 2}")
```

```
x ** 2 =  [25  0  9  9 49 81  9 25  4 16]
x % 2 =  [1 0 1 1 1 1 1 1 0 0]
```

[81]:
```
# 4. Transform the array as: $-(\frac{x}{2}+1)^2$, where $x$ is the random 1d␣
 ↪array.
newArray=-(0.5*x+1) ** 2

print(f"Given array      : {x}")
print(f"Transformed array: {newArray}")
```

```
Given array      : [5 0 3 3 7 9 3 5 2 4]
Transformed array: [-12.25  -1.    -6.25  -6.25 -20.25 -30.25  -6.25 -12.25  -4.
```

```
-9.  ]
```

2.10 Ufuncs

A universal function (or ufunc for short) is a **vectorized** wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

For example, see the following cell:

[82]:
```
x = np.array([-2,-1,0,1,2])

print(x) # the original array

print('-'*10)
print(np.abs(x))  # convert all elements to +ve numbers
print(np.sqrt(np.abs(x)))  # get square-roots of all abs(elements)
print(np.min(x))  # get the min value
print(np.max(x))  # get the max value
print(np.sum(x))  # get the sum of all value
```

```
[-2 -1  0  1  2]
----------
[2 1 0 1 2]
[1.41421356 1.         0.         1.         1.41421356]
-2
2
0
```

2.11 Broadcasting

Vectorization involving multiple arrays can be done. Vectorization follows **broadcasting** rules. The key broadcasting rule is: **In order to broadcast, the size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one**. In rough sense, what it means is: 1. If two arrays of same dimensions have the same shape, then corresponding elements can be broadcasted. 2. If the arrays do not have the same dimensions, prepend the shape of the lower dimension array with 1s until both arrays have the same dimensions. 3. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

For example, see the following cells:

[83]:
```
# 1. Add/multiply corresponding elements of two similar sized 1d arrays.
x = np.array([0,1,2])
y = np.array([5,5,3])

print(f"The dimensions for the arrays are: {x.shape} & {y.shape}.\n\n")

print(f'  array 1: {x}')
print(f'  array 2: {y}')
print('-'*20)
```

```
print(f'array sum: {x+y}')
print(f'array dot: {x*y}')

## list1 + list2, where list1 and list2 are Python 1d lists results in␣
 ↪concatenation!
## list1 * list2, where list1 and list2 are Python 1d lists results in error!
```

```
The dimensions for the arrays are: (3,) & (3,).


  array 1: [0 1 2]
  array 2: [5 5 3]
--------------------
array sum: [5 6 5]
array dot: [0 5 6]
```

[84]:
```
# 2. Add/multiply corresponding elements of two different sized arrays.
M = np.ones((2,3), dtype='int')
a = np.arange(3,dtype='int')
print(f"The dimensions for the arrays are: {M.shape} & {a.shape}.\n\n")

print(f'  Matrix 1:\n {M}')
print(f'   Array 1:\n {a}')
print('-'*10)
print(f'   The sum:\n {M+a}')
print(f'   The dot:\n {M*a}')
```

```
The dimensions for the arrays are: (2, 3) & (3,).


  Matrix 1:
 [[1 1 1]
 [1 1 1]]
   Array 1:
 [0 1 2]
----------
   The sum:
 [[1 2 3]
 [1 2 3]]
   The dot:
 [[0 1 2]
 [0 1 2]]
```

2.12 Masking

Masked array is an **array of booleans** that determines for each element of the associated array whether the value is valid or not. The masked arrays are obtained using different conditions. Numpy array comparison are vectorized too.

For example, see the following cells:

[85]:
```
x = np.array([1,2,3,4,5,6])

print(x<3)  # less than
print(x>3)  # greater than
print(x<=3) #less than or equal
print(x>=3) #greater than or equal
print(x!=3) #not equal
print(x==3) #equal
```

```
[ True  True False False False False]
[False False False  True  True  True]
[ True  True  True False False False]
[False False  True  True  True  True]
[ True  True False  True  True  True]
[False False  True False False False]
```

[86]:
```
mask=x<3  # masked array that shows which elements of x are strictly less than 3
x[mask]
```

[86]:
```
array([1, 2])
```

[87]:
```
mask=(x > 2) & (x < 5)  # shows which elements of x are strictly between than 2␣
↪and 5
x[mask]
```

[87]:
```
array([3, 4])
```

[88]:
```
mask=(x < 2) | (x >= 5)  # shows which elements of x are not in between than 2␣
↪and 4
x[mask]
```

[88]:
```
array([1, 5, 6])
```

3 References

3.1 Online:

Main ref. Materalis from ISE 291 1. https://docs.python.org/3/ 2. https://numpy.org/