CSUS help desk presents:

Panda & Numpy

By: Eric, Deifilia, and Robin

Quick overview of python

- In Python, you don’t declare data types. {int, float, string, etc.}

Quick overview of python

- In Python, you don't declare data types. {int, float, string, etc.}

  Example {Comparison with Java and Python}

  Java

```
int x = 2;
System.out.println(x); //It will print out 2
```

Quick overview of python

- In Python, you don't declare data types. {int, float, string, etc.}

  Example {Comparison with Java and Python}

  Java

```java
int x = 2;
System.out.println(x); //It will print out 2
```

  Python

```python
x = 2
print(x)     #It will print out 2
```

Quick overview of python

- Python don’t have explicit markers for blocks.

Quick overview of python

- Python don’t have explicit markers for blocks.

  Example {Comparison with Java and Python}

  Java

```
if(x == 2){
System.out.println("two");
}
```

Quick overview of python

- Python don’t have explicit markers for blocks.

  Example {Comparison with Java and Python}

Java

```
if(x == 2){
System.out.println("two");
}
```

Python

```
if(x == 2):
    print("two")
```

Quick overview of python

- Python don't have explicit markers for blocks.

  Example {Comparison with Java and Python}

  Java

```
if(x == 2){
System.out.println("two");
}
```

  Python

```
if(x == 2):
    print("two")
```

- **Indentation** is CRUCIAL to Python and failure to indent properly leads to ***IndentationError*** exception.

Quick overview of python

- Python don't have explicit markers for blocks.

  Example {Comparison with Java and Python}

  Java

```
if(x == 2){
System.out.println("two");
}
```

  Python

```
if(x == 2):
    print("two")
```

- **Indentation** is CRUCIAL to Python and failure to indent properly leads to ***IndentationError*** exception.
- Make sure to decide whether to indent with a space or a tab.

List in python

- Lists are mutable in Python. (List can change after the creation).

List in python

- Lists are mutable in Python. (List can change after the creation).

  Empty: `emptyList = []`

List in python

- Lists are mutable in Python. (List can change after the creation).

  Empty: `emptyList = []`

  List that contains integers: `integerList = [16,63,65]`

List in python

- Lists are mutable in Python. (List can change after the creation).

  Empty: `emptyList = []`

  List that contains integers: `integerList = [16,63,65]`

  List that contains Strings: `stringList = ["Hello","World"]`

List in python

- Lists are mutable in Python. (List can change after the creation).

  Empty: `emptyList = []`

  List that contains integers: `integerList = [16,63,65]`

  List that contains Strings: `stringList = ["Hello","World"]`

  List that contains another list: `listList = [[39,80,89],[20,21,29]]`

Quick Question

Can we create a list that contains mixed data types?

Example:

```
mixedList = ["String", 20, 3.14, True]
```

Quick Question

Can we create a list that contains mixed data types?

Example:

```
mixedList = ["String", 20, 3.14, True]
```

Answer: Yes you are allowed to create and store mixed data types in the list.

Access Elements in the List

To access the element in the list, use the index operator [i] ($i \in \mathbb{Z}$)

Example:

```
coffeeList = ["americano", "cappuccino", "mocha", "latte"]
# I want a cup of cappuccino
print("Give me a cup of " + coffeeList[1])
#prints "Give me a cup of cappuccino"
```

Access Elements in the List

To access the element in the list, use the index operator [i] ($i \in \mathbb{Z}$)

Example:

```
coffeeList = ["americano", "cappuccino", "mocha", "latte"]
# I want a cup of cappuccino
print("Give me a cup of " + coffeeList[1])
#prints "Give me a cup of cappuccino"
```

In Python, negative index value is acceptable

Example:

```
coffeeList = ["americano", "cappuccino", "latte", "mocha"]
# I want a cup of latte
print("Give me a cup of " + coffeeList[-2])
#prints "Give me a cup of latte"
```

Access Elements in the List

To get the list of elements from x to y, use the index operation [x:y] (x,y $\in \mathbb{Z}$)

Access Elements in the List

To get the list of elements from x to y, use the index operation [x:y] (x,y $\in \mathbb{Z}$)

Note: x is inclusive and y is exclusive

Access Elements in the List

To get the list of elements from x to y, use the index operation [x:y] (x,y $\in \mathbb{Z}$)

Note: x is inclusive and y is exclusive

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6]
# Exclude 0, 1, and 6
print(numberList[2:6])
#prints [2, 3, 4, 5]
```

Access Elements in the List

To get the list of elements from x to y, use the index operation [x:y] (x,y $\in \mathbb{Z}$)

Note: x is inclusive and y is exclusive

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6]
# Exclude 0, 1, and 6
print(numberList[2:6])
#prints [2, 3, 4, 5]
```

Leaving x blank is same as x = 0 and leaving y blank is same as y = length of the list -1

Access Elements in the List

To get the list of elements from x to y, use the index operation [x:y] (x,y $\in \mathbb{Z}$)

Note: x is inclusive and y is exclusive

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6]
# Exclude 0, 1, and 6
print(numberList[2:6])
#prints [2, 3, 4, 5]
```

Leaving x blank is same as x = 0 and leaving y blank is same as y = length of the list

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Greater than 4
print(numberList[5:])
#prints [5, 6, 7, 8, 9, 10]
```

Access Elements in the List

Using negative indexes also works for getting the part of the list.

Note: x is still inclusive, y is still exclusive

Access Elements in the List

Using negative indexes also works for getting the part of the list.

Note: x is still inclusive, y is still exclusive

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Range between 3 and 7
print(numberList[-8:-3])
#prints [3, 4, 5, 6, 7]
```

Access Elements in the List

Using negative indexes also works for getting the part of the list.

Note: x is still inclusive, y is still exclusive

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Range between 3 and 7
print(numberList[-8:-3])
#prints [3, 4, 5, 6, 7]
```

Leaving x blank and a negative y is allowed (Vice versa)

Access Elements in the List

Using negative indexes also works for getting the part of the list.

Note: x is still inclusive, y is still exclusive

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Range between 3 and 7
print(numberList[-8:-3])
#prints [3, 4, 5, 6, 7]
```

Leaving x blank and a negative y is allowed (Vice versa)

Example:

```
numberList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Greater than 6
print(numberList[-4:])
#prints [7, 8, 9, 10]
```

Quick Question

What is the output of the following code?

```
numberList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numberList[-8:9])
```

Quick Question

What is the output of the following code?

```
numberList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numberList[-8:9])
```

Answer: [3, 4, 5, 6, 7, 8]

List Operations

- To get the size of the list, use **len(*list*)**.

Example:

```
gradesList = ['A', 'A-', 'B+', 'B', 'B-', 'C+', 'C', 'D', 'F']
print(len(gradesList)) #prints 9
```

List Operations

- To get the size of the list, use **len(*list*)**.

Example:

```
gradesList = ['A', 'A-', 'B+', 'B', 'B-', 'C+', 'C', 'D', 'F']
print(len(gradesList)) #prints 9
```

- To change the element in the specific index, code ***list[i] = element***.

Example:

```
petList = ['cat', 'dog', 'hamster', 'rabbit', 'parrot']
#replace hamster with guinea pig
petList[2] = "guinea pig"
print(petList) #prints ['cat', 'dog', 'guinea pig', 'rabbit', 'parrot']
```

List Operations

- To get the size of the list, use **len(*list*)**.

Example:

```
gradesList = ['A', 'A-', 'B+', 'B', 'B-', 'C+', 'C', 'D', 'F']
print(len(gradesList)) #prints 9
```

- To change the element in the specific index, code ***list[i] = element***.

Example:

```
petList = ['cat', 'dog', 'hamster', 'rabbit', 'parrot']
#replace hamster with guinea pig
petList[2] = "guinea pig"
print(petList) #prints ['cat', 'dog', 'guinea pig', 'rabbit', 'parrot']
```

- To join two lists together, code **list3 = *list1 + list2***.

Example:

```
FruitList = ['Apple', 'Banana', 'Peach']
VegetableList = ['Carrot', 'Lettuce', 'Broccoli']
HealthyList = FruitList + VegetableList
print(HealthyList)
#prints ['Apple', 'Banana', 'Peach', 'Carrot', 'Lettuce', 'Broccoli']
```

List Operations

- To add an element into the list, use **append(*element*)**.

Example:

```
cityList = ['Montreal', 'Vancouver', 'Toronto', 'Calgary']
cityList.append('Winnipeg')
print(cityList)
#prints ['Montreal', 'Vancouver', 'Toronto', 'Calgary', 'Winnipeg']
```

List Operations

- To add an element into the list, use **append(*element*)**.

Example:

```
cityList = ['Montreal', 'Vancouver', 'Toronto', 'Calgary']
cityList.append('Winnipeg')
print(cityList)
#prints ['Montreal', 'Vancouver', 'Toronto', 'Calgary', 'Winnipeg']
```

- To remove an element from the list, use **remove(*element*)**.

Example:

```
cartList = ['notebook', 'headphone', 'kitchen knife', 'T-shirts', 'towel']
cartList.remove('T-shirts')
print(cartList)
#prints ['notebook', 'headphone', 'kitchen knife', 'towel']
```

List Operations

- To add an element into the list, use **append(*element*)**.

Example:

```
cityList = ['Montreal', 'Vancouver', 'Toronto', 'Calgary']
cityList.append('Winnipeg')
print(cityList)
#prints ['Montreal', 'Vancouver', 'Toronto', 'Calgary', 'Winnipeg']
```

- To remove an element from the list, use **remove(*element*)**.

Example:

```
cartList = ['notebook', 'headphone', 'kitchen knife', 'T-shirts', 'towel']
cartList.remove('T-shirts')
print(cartList)
#prints ['notebook', 'headphone', 'kitchen knife', 'towel']
```

- To empty the list, use **clear()** .

Example:

```
garbageList = ['used tissue', 'food wrappers', 'expired food']
garbageList.clear()
print(garbageList) #prints []
```

Check the Element

- To check whether the element exists in the list,

Example:

```
travelList = ["baggage","ticket","hand sanitizer"]
if "ticket" in travelList:
    print("yes")
```

Check the Element

- To check whether the element exists in the list,

  Example:

```
travelList = ["baggage","ticket","hand sanitizer"]
if "ticket" in travelList:
    print("yes")
```

- You can also use **if *list*.count(*element*) > 0** syntax to test the element's existence.

Check the Element

- To check whether the element exists in the list,

Example:

```
travelList = ["baggage","ticket","hand sanitizer"]
if "ticket" in travelList:
    print("yes")
```

- You can also use **if *list*.count(*element*) > 0** syntax to test the element's existence.
- List comprehension is another way to check for the element in the list.

Multi-Dimensional Lists in Python

- Creating a single list is considered as 1-dimensional list.

Multi-Dimensional Lists in Python

- Creating a single list is considered as 1-dimensional list.
- N-number of lists in the list is considered as N-dimensional list.

Example: {2-dimensional list}

```
teamList = [['Justin', 'Sarah', 'Sam'],['Gord', 'Alex', 'Amanda']]
print(teamList[1][0]) #Find Gord
print(teamList[0][2]) #Find Sam
```

Multi-Dimensional Lists in Python

- Creating a single list is considered as 1-dimensional list.
- N-number of lists in the list is considered as N-dimensional list.

Example: {2-dimensional list}

```
teamList = [['Justin', 'Sarah', 'Sam'],['Gord', 'Alex', 'Amanda']]
print(teamList[1][0]) #Find Gord
print(teamList[0][2]) #Find Sam
```

Explanation: To find "Gord", we start off with exploring the outside list.(**[[x],[y]]**)

"Gord" can be found in the **y** list. So we explore **teamList**[1] = ['Gord', 'Alex', 'Amanda']

Let **teamList**[1] = **blueTeamList**, then **blueTeamList**[0] = 'Gord'

Replace **blueTeamList** with **teamList[1]**. Then we have **teamList[1][0]** = 'Gord'

Quick Question

Consider a single list containing 2-D list and 3-D list.

Example:

```
jaggedList = [[1,2,3,4,5],[[6, 7, 8],[9,10]]]
```

Quick Question

Consider a single list containing 2-D list and 3-D list.

Example:

```
jaggedList = [[1,2,3,4,5],[[6, 7, 8],[9,10]]]
```

1. It is allowed?
2. If so, how to get #4 and #8? If not, explain.

Quick Question

Consider a single list containing 2-D list and 3-D list.

Example:

```
jaggedList = [[1,2,3,4,5],[[6, 7, 8],[9,10]]]
```

1. It is allowed?
2. If so, how to get #4 and #8? If not, explain.

Answer #1: Yes! This list structure is known as “jagged list”

Quick Question

Consider a single list containing 2-D list and 3-D list.

Example:

```
jaggedList = [[1,2,3,4,5],[[6, 7, 8],[9,10]]]
```

1. It is allowed?
2. If so, how to get #4 and #8? If not, explain.

Answer #1: Yes! This list structure is known as “jagged list”

Answer#2: #4 = jaggedList[0][3], #8 = jaggedList[1][0][2]

Bonus: List Comprehension in Python

- List comprehension (LC) is a best way to improve performance and makes it easier to read under right circumstances.

Bonus: List Comprehension in Python

- List comprehension (LC) is a best way to improve performance and makes it easier to read under right circumstances.

  Example: {Using for loops}

  For-loop:

```
charList = []
for i in 'python':
    charList.append(i)
print(charList) #prints out ['p','y','t','h','o','n']
```

Bonus: List Comprehension in Python

- List comprehension (LC) is a best way to improve performance and makes it easier to read under right circumstances.

Example: {Using for loops}

For-loop:

```
charList = []
for i in 'python':
    charList.append(i)
print(charList) #prints out ['p','y','t','h','o','n']
```

LC:

```
charList = [character for character in 'python']
print(charList) #prints out ['p', 'y', 't', 'h', 'o', 'n']
```

Bonus: List Comprehension in Python

- List comprehension (LC) is a best way to improve performance and makes it easier to read under right circumstances.

  Example: {Using for loops}

  For-loop:

```
charList = []
for i in 'python':
    charList.append(i)
print(charList) #prints out ['p','y','t','h','o','n']
```

  LC:

```
charList = [character for character in 'python']
print(charList) #prints out ['p', 'y', 't', 'h', 'o', 'n']
```

- If you are interested, there are many sources about list comprehension. Feel free to explore.

Why using a library like pandas and numpy?

1. Using the library correctly can increase the performance and optimization.

Why using a library like pandas and numpy?

1. Using the library correctly can increase the performance and optimization.
2. Using the library correctly can reduce the memory usage.

Why using a library like pandas and numpy?

1. Using the library correctly can increase the performance and optimization.
2. Using the library correctly can reduce the memory usage.
3. Libraries contain functions that are easy to use and some algorithms are computationally optimal.

Why using a library like pandas and numpy?

1. Using the library correctly can increase the performance and optimization.
2. Using the library correctly can reduce the memory usage.
3. Libraries contain functions that are easy to use and some algorithms are computationally optimal.
4. Most of the libraries are specialized.

Why using a library like pandas and numpy?

1. Using the library correctly can increase the performance and optimization.
2. Using the library correctly can reduce the memory usage.
3. Libraries contain functions that are easy to use and some algorithms are computationally optimal.
4. Most of the libraries are specialized.
5. Some libraries allow visual representation of the output.

Why using a library like pandas and numpy?

1. Using the library correctly can increase the performance and optimization.
2. Using the library correctly can reduce the memory usage.
3. Libraries contain functions that are easy to use and some algorithms are computationally optimal.
4. Most of the libraries are specialized.
5. Some libraries allow visual representation of the output.

Let’s Focus on Numpy and Pandas Today!

NumPy

What is NumPy anyways?

- **Array-based** data manipulation
  - Fixed-size container
  - Each item has the same shape and size
  - Multi-dimensional possible

Benefits of NumPy

\+ Much faster than Python lists
\+ Uses less memory
\+ Can specify the data types (better automatic optimization)

Important definitions

- **np.dtype:** data type (int64, int, float)
- **np.shape:** Tuple that gives the dimensions of the array
- **np.axes:** dimensions are called axes
- **np.rank**: Number of singular values of the array that are greater than zero

Initializing NumPy arrays

np.array(): **Specify inputs**

```
np.array([[1, 2, 3],[ 4, 5, 6]])
```

$$\begin{bmatrix}[1 & 2 & 3], \\ [4 & 5 & 6]\end{bmatrix}$$

np.zeros(): **Initialize with 0 values**

```
np.zeros(4)
```

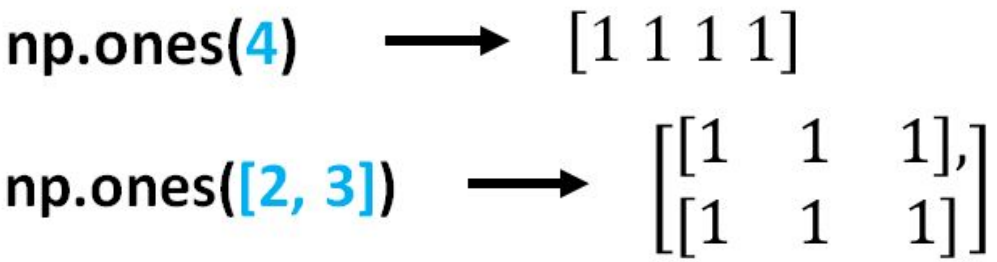
$$[0\ 0\ 0\ 0]$$

```
np.zeros([2, 3])
```

$$\begin{bmatrix}[0 & 0 & 0], \\ [0 & 0 & 0]\end{bmatrix}$$

Initializing NumPy arrays

np.ones(): **Initialize with ones**



np.empty(): **Initialize empty array**

- What is the difference between this and np.zeros()?

Initializing NumPy arrays

np.arange(): **Initialize with values over a range**

np.arange(4) ⟶ [0 1 2 3]

np.arange(first, last, step size)

np.arange(2, 10, 3) ⟶ [2 5 8]

np.linspace(): **Initialize with values over a specified range, linearly spaced apart**

np.linspace(first, last, number of terms)

np.linspace(0, 10, 5) ⟶ [0, 2.5, 5, 7.5, 10]

Shape: Getting to know the dimensions of the array

- **ndarray.ndim:** number of dimensions
- **ndarray.size:** number of elements
- **ndarray.shape:** tuple

```
In [8]: arr = np.array([[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]], [[12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23]]])
        print(arr)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

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

Shape: Getting to know the dimensions of the array

- **ndarray.ndim:** number of dimensions
- **ndarray.size:** number of elements
- **ndarray.shape:** tuple

```
In [8]: arr = np.array([[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]], [[12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23]]])
print(arr)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

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

```
In [11]: print("Array has", arr.ndim, "dimensions.")
         print("Array has", arr.size, "elements.")
         print("Array has shape", arr.shape)

         Array has 3 dimensions.
         Array has 24 elements.
         Array has shape (2, 3, 4)
```

Reshape

- Changes the shape of the array
- ! The shape of the new array must have the same number of elements as the first
- **ndarray.reshape(new, shape)**

```
In [16]: arr.reshape(12, 1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-16-7514b0b68abe> in <module>
----> 1 arr.reshape(12, 1)

ValueError: cannot reshape array of size 24 into shape (12,1)
```

Caution: Reshape

Why does this occur?

```
In [17]: arr.reshape(12, 2)
```

```
Out[17]: array([[ 0,  1],
                [ 2,  3],
                [ 4,  5],
                [ 6,  7],
                [ 8,  9],
                [10, 11],
                [12, 13],
                [14, 15],
                [16, 17],
                [18, 19],
                [20, 21],
                [22, 23]])
```

```
In [18]: print(arr)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

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

Caution: Reshape

Why does this occur?

```
In [17]: arr.reshape(12, 2)

Out[17]: array([[ 0,  1],
                [ 2,  3],
                [ 4,  5],
                [ 6,  7],
                [ 8,  9],
                [10, 11],
                [12, 13],
                [14, 15],
                [16, 17],
                [18, 19],
                [20, 21],
                [22, 23]])
```

```
In [18]: print(arr)

[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

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

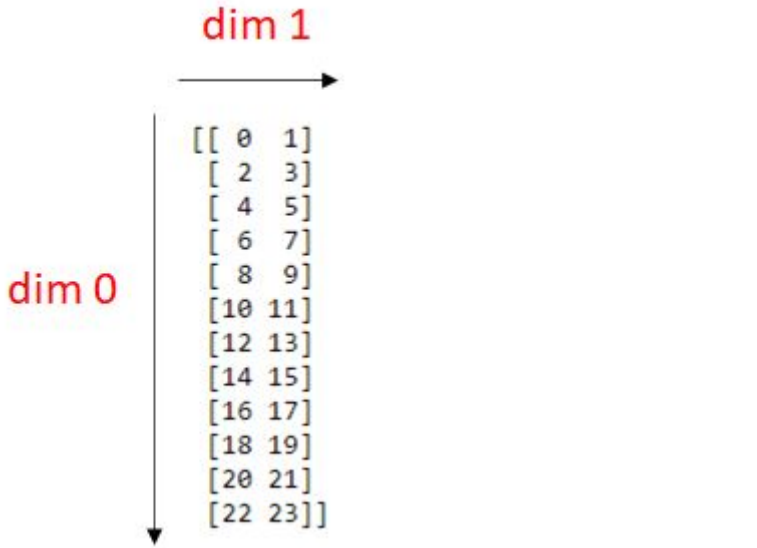
```
In [22]: arr = arr.reshape(12, 2)

In [23]: print(arr)

[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]
 [12 13]
 [14 15]
 [16 17]
 [18 19]
 [20 21]
 [22 23]]
```

Reshape: Easy initialization

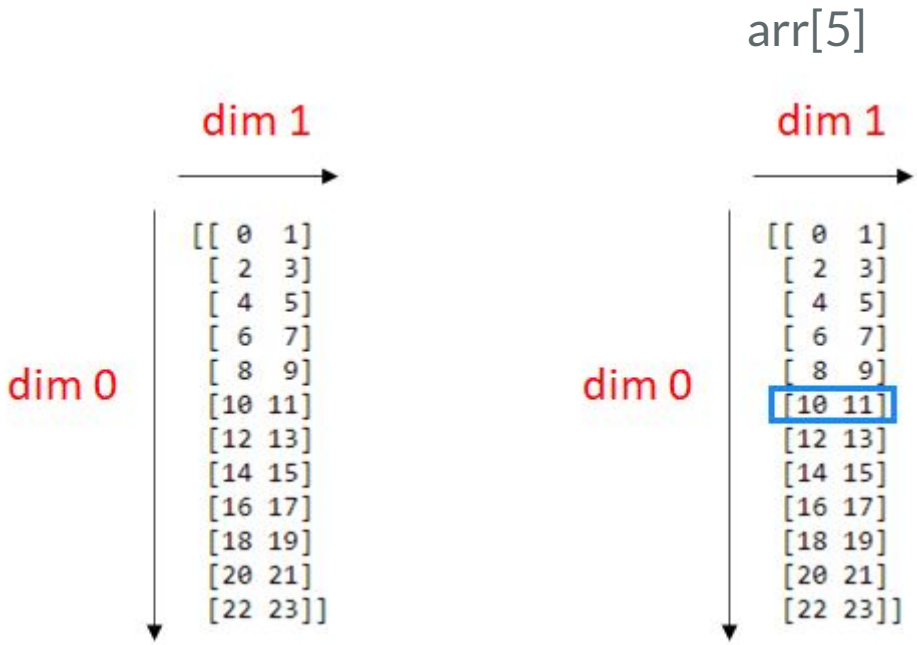
In [22]:
```
threeByFour = np.arange(12).reshape(3, 4)
print(threeByFour)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

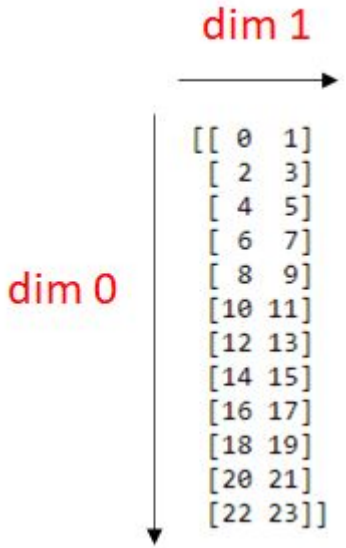
Selection of specific dimensions



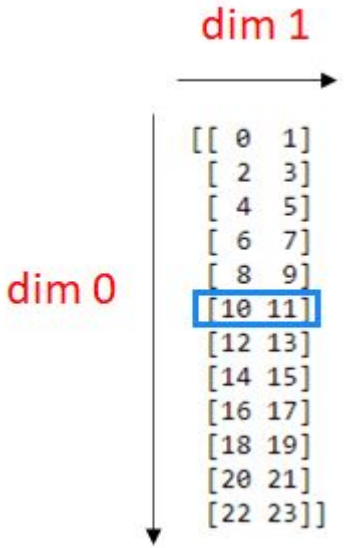
Selection of specific dimensions



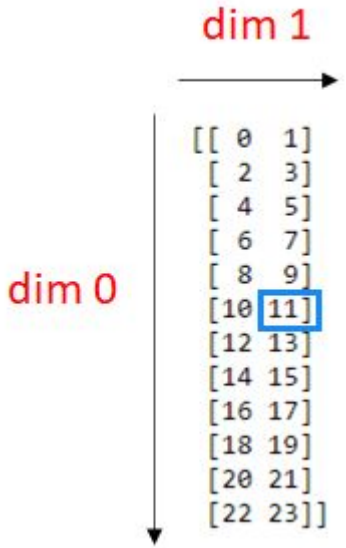
Selection of specific dimensions



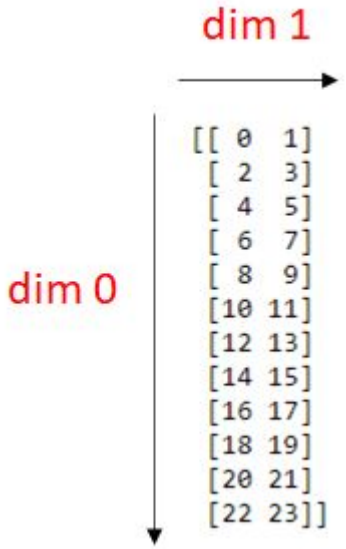
arr[5]



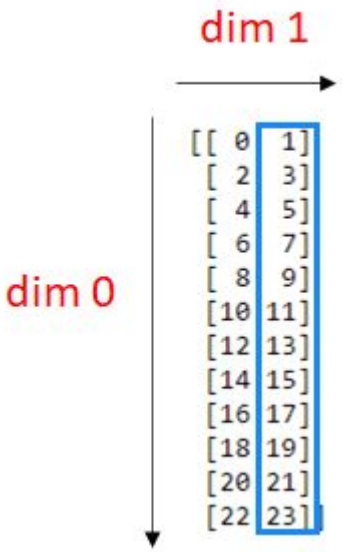
arr[5, 1]



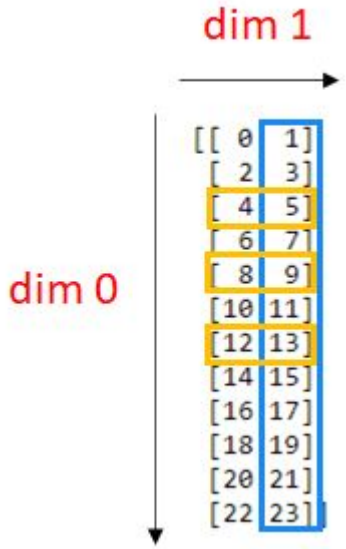
Selection of specific dimensions



arr[:,1]



arr[2:8:2, 1]



Using np.delete()

np.delete(arr, object, (optional) axis)

```
threeByFour = np.arange(12).reshape(3, 4)
print(threeByFour)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Using np.delete()

np.delete(arr, object, (optional) axis)

```
|: threeByFour = np.arange(12).reshape(3, 4)
print(threeByFour)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
threeByFour = np.arange(12).reshape(3, 4)
print(np.delete(threeByFour, 0, axis=0))
```

```
[[ 4  5  6  7]
 [ 8  9 10 11]]
```

** Caution! np.delete returns a **copy** of the array. You need to say a = np.delete(a…) if you want to save the new array into the same variable

Using np.delete()

np.delete(arr, object, (optional) axis)

```
threeByFour = np.arange(12).reshape(3, 4)
print(np.delete(threeByFour, 0, axis=0))
```

```
[[ 4  5  6  7]
 [ 8  9 10 11]]
```

```
threeByFour = np.arange(12).reshape(3, 4)
print(np.delete(threeByFour, 0))
```

```
[ 1  2  3  4  5  6  7  8  9 10 11]
```

Using np.delete()

np.delete(arr, object, (optional) axis)

```
threeByFour = np.arange(12).reshape(3, 4)
print(threeByFour)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
threeByFour = np.arange(12).reshape(3, 4)
print(np.delete(threeByFour, [1, 3],axis=1))
```

```
[[ 0  2]
 [ 4  6]
 [ 8 10]]
```

Using np.delete()

np.delete(arr, object, (optional) axis)

```
threeByFour = np.arange(12).reshape(3, 4)
print(threeByFour)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
threeByFour = np.arange(12).reshape(3, 4)
print(np.delete(threeByFour, slice(0, 2, 1),axis=1))
```

```
[[ 2  3]
 [ 6  7]
 [10 11]]
```

Using np.delete()

np.delete(arr, object, (optional) axis)

```
threeByFour = np.arange(12).reshape(3, 4)
print(threeByFour)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
threeByFour = np.arange(12).reshape(3, 4)
print(np.delete(threeByFour, np.s_[0:2:1],axis=1))
```

```
[[ 2  3]
 [ 6  7]
 [10 11]]
```

Saving data

```
np.savetxt(filename, data, delimiter)
```

-

Saving data

np.savetxt(path-to-file, data, delimiter)

```
dim1 = dim2 = np.arange(6)
np.savetxt('output.csv', (dim1, dim2), delimiter=",")
```

1D arrays, equal sized

	A	B	C	D	E	F
1	0.00E+00	1.00E+00	2.00E+00	3.00E+00	4.00E+00	5.00E+00
2	0.00E+00	1.00E+00	2.00E+00	3.00E+00	4.00E+00	5.00E+00

dim1

Saving data

np.savetxt(path-to-file, data, delimiter)

```
dim1 = np.arange(0, 3*4, 2).reshape(2, 3)
np.savetxt('output.txt', dim1, delimiter=" ")
```

1- or 2-D array

File Edit Format View Help

```
0.000000000000000000e+00 2.000000000000000000e+00 4.000000000000000000e+00
6.000000000000000000e+00 8.000000000000000000e+00 1.000000000000000000e+01
```

Print all data

input - N

File Edit Fo

```
1, 1
2, 4
3, 9
4, 16
5, 25
6, 36
7, 49
8, 64
```

```
readIn = np.loadtxt('input.txt', delimiter=",")
print(readIn)
```

```
[[ 1.  1.]
 [ 2.  4.]
 [ 3.  9.]
 [ 4. 16.]
 [ 5. 25.]
 [ 6. 36.]
 [ 7. 49.]
 [ 8. 64.]]
```

Print all data

weather.txt:

```
September, 24, 14
October, 15, 8
November, 9, 2
December, 3, -3
```

```
readWeather = np.loadtxt('weather.txt',
                         dtype={'names': ('month', 'high', 'low'),
                                'formats':('S10', 'i4', 'i4')},
                         delimiter=",")
print(readWeather)
```

```
[(b'September', 24, 14) (b'October', 15,  8) (b'November',  9,  2)
 (b'December',  3, -3)]
```

Broadcasting

- Performing operations with on inputs with different dimensions

Broadcasting

- Performing operations with on inputs with different dimensions
- Operations are done element-wise

```
arr = np.arange(10)
b   = 3
add = b + arr
print(add)
```

```
[ 3  4  5  6  7  8  9 10 11 12]
```

```
a = np.array([4, 2, 3, 1])
b = np.array([6, 4, 0, 1])
print(a*b)
```

```
[24  8  0  1]
```

Broadcasting

- A dimension is compatible if
  - Same size in the dimension, OR one of the arrays has size 1

Broadcasting

- A dimension is compatible if
  - Same size in the dimension, OR one of the arrays has size 1
- Broadcastable **iff** all dimensions are compatible

Broadcasting

- A dimension is compatible if
  - Same size in the dimension, OR one of the arrays has size 1
- Broadcastable **iff** all dimensions are compatible
- Resulting array: **element-wise maximum** of shapes of input arrays

Broadcasting

- A dimension is compatible if
  - Same size in the dimension, OR one of the arrays has size 1
- Broadcastable **iff** all dimensions are compatible
- Resulting array: **element-wise maximum** of shapes of input arrays
- If an array has size 1, and the other array has size > 1, the first array is copied

Broadcasting

- Revisiting the previous example

```
arr = np.arange(10)
b   = 3
add = b + arr
print(add)
```

```
[ 3  4  5  6  7  8  9 10 11 12]
```

```
a = np.array([4, 2, 3, 1])
b = np.array([6, 4, 0, 1])
print(a*b)
```

```
[24  8  0  1]
```

Broadcasting

```
a = np.arange(6).reshape(6, 1)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

Broadcasting

```
a = np.arange(6).reshape(6, 1)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

```
print(np.shape(c))
```

```
(3, 6, 2)
```

Broadcasting

```
a = np.arange(6).reshape(6, 1)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

```
print(np.shape(c))
```

```
(3, 6, 2)
```

```
print(c)
```

```
[[[ 0  1]
  [ 1  2]
  [ 2  3]
  [ 3  4]
  [ 4  5]
  [ 5  6]]

 [[ 2  3]
  [ 3  4]
  [ 4  5]
  [ 5  6]
  [ 6  7]
  [ 7  8]]

 [[ 4  5]
  [ 5  6]
  [ 6  7]
  [ 7  8]
  [ 8  9]
  [ 9 10]]]
```

Which examples can be broadcasted?

-

```
a = np.arange(6).reshape(1, 6, 1)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

```
a = np.arange(6).reshape(1, 6)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

```
a = np.arange(6).reshape(1, 6, 1)
b = np.arange(6).reshape(3, 2, 2)
c = a + b
```

```
a = np.arange(6).reshape(6, 1)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

Which examples can be broadcasted?

```
a = np.arange(6).reshape(1, 6, 1)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

```
a = np.arange(6).reshape(1, 6)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

```
a = np.arange(6).reshape(1, 6, 1)
b = np.arange(6).reshape(3, 2, 2)
c = a + b
```

```
a = np.arange(6).reshape(6, 1)
b = np.arange(6).reshape(3, 1, 2)
c = a + b
```

Other useful functions:

- **Minimum:** np.min()
  - np.min(array) yields global minimum of array
  - np.min(array, axis=0) gives minimum along axis = 0
- **Maximum:** np.max()
- **Mean**: np.mean()
- **Median**: np.median()
- **Standard deviation**: np.std()

Example: Masking & counting

- Make T/F masks based on criteria
- From a list [0, 5, …, 95], count the number of elements that:
  - Are odd
  - Are in between 10 and 80, exclusive

```
a = np.linspace(0, 95, 20).reshape(5, 4)
mask1 = a % 2
mask2 = a > 10
mask3 = a < 80
```

Example: Masking & counting

- Make T/F masks based on criteria
- From a list [0, 5, …, 95], count the number of elements

```
a = np.linspace(0, 95, 20).reshape(5, 4)
mask1 = a % 2
mask2 = a > 10
mask3 = a < 80
```

```
print(mask1)
```

```
[[0. 1. 0. 1.]
 [0. 1. 0. 1.]
 [0. 1. 0. 1.]
 [0. 1. 0. 1.]
 [0. 1. 0. 1.]]
```

```
print(mask2)
```

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

```
print(mask3)
```

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

Example: Masking & counting

- Make T/F masks based on criteria
- From a list [0, 5, …, 95], count the number of elements

```
result = a * mask1 * mask2 * mask3
print(result)
```

```
[[ 0.  0.  0. 15.]
 [ 0. 25.  0. 35.]
 [ 0. 45.  0. 55.]
 [ 0. 65.  0. 75.]
 [ 0.  0.  0.  0.]]
```

```
values = np.nonzero(result)
std = np.std(result)
avg = np.mean(result)
print("There are ", np.count_nonzero(result), "elements.")
print("The average is", avg, "and the stdev is", std)
```

```
There are  7 elements.
The average is 15.75 and the stdev is 24.508926945094924
```

Example: Masking & counting

- Make T/F masks based on criteria
- From a list [0, 5, …, 95], count the number of elements

```
result = a * mask1 * mask2 * mask3
print(result)
```

```
[[ 0.  0.  0. 15.]
 [ 0. 25.  0. 35.]
 [ 0. 45.  0. 55.]
 [ 0. 65.  0. 75.]
 [ 0.  0.  0.  0.]]
```

```
values = np.nonzero(result)
std = np.std(result)
avg = np.mean(result)
print("There are ", np.count_nonzero(result), "elements.")
print("The average is", avg, "and the stdev is", std)
```

```
There are  7 elements.
The average is 15.75 and the stdev is 24.508926945094924
```

Example 2: Replace missing grades with average

89	84	68	87	71	96	79	83	60	68
89	86	83	71	62	99	72	61	63	82
99	0	74	92	76	90	96	98	62	91
72	90	99	64	74	87	83	63	66	97
63	88	77	0	97	78	96	93	64	74
84	99	75	88	85	75	92	0	93	60
91	61	98	71	71	71	0	0	89	72
78	73	83	94	90	88	67	99	65	72
88	69	72	70	75	87	63	0	61	83
70	75	81	77	63	77	62	70	67	69

Example 2: Replace missing grades with average

```
num_nonzero  = np.count_nonzero(grades, axis=1)       # count number of non-zero elements per row
sum_line     = np.sum(grades, axis=1)                 # find sum over all rows
inds         = np.where(grades==0)                    # get indices where values are 0
grades[inds] = np.take(sum_line/num_nonzero, inds[1]) # replaces the values at the index by the new average
```

Example 2: Replace missing grades with average

```
num_nonzero  = np.count_nonzero(grades, axis=1)       # count number of non-zero elements per row
sum_line     = np.sum(grades, axis=1)                 # find sum over all rows
inds         = np.where(grades==0)                    # get indices where values are 0
grades[inds] = np.take(sum_line/num_nonzero, inds[1]) # replaces the values at the index by the new average
```

```
print(grades)
```

```
[[89.  84.  68.  87.  71.  96.  79.  83.  60.  68. ]
 [89.  86.  83.  71.  62.  99.  72.  61.  63.  82. ]
 [99.  76.8 74.  92.  76.  90.  96.  98.  62.  91. ]
 [72.  90.  99.  64.  74.  87.  83.  63.  66.  97. ]
 [63.  88.  77.  79.5 97.  78.  96.  93.  64.  74. ]
 [84.  99.  75.  88.  85.  75.  92.  80.9 93.  60. ]
 [91.  61.  98.  71.  71.  71.  78.  80.9 89.  72. ]
 [78.  73.  83.  94.  90.  88.  67.  99.  65.  72. ]
 [88.  69.  72.  70.  75.  87.  63.  80.9 61.  83. ]
 [70.  75.  81.  77.  63.  77.  62.  70.  67.  69. ]]
```

PANDAS

Introduction to Pandas

In this section of the tutorial we will learn how to use pandas for data analysis. You can think of pandas as an extremely powerful version of Excel, with a lot more features. We will be going over the following topics:

- Introduction to Pandas
- Series
- DataFrames
- Missing Data
- GroupBy
- Merging,Joining,and Concatenating
- Operations
- Data Input and Output

Series in Pandas

- Series is one of the main data types in the Pandas library. It is a 1D labeled array capable of holding different data types (integer, string, float, python objects, etc.)
- The axis labels are collectively called *index*. Pandas Series is nothing but a column in an excel sheet.
- A Series is really similar to a numpy array (in fact it's built on top of it).

Let's see some examples!