Python Basics

Data Arrays

A sequence is a collection of ordered objects. Ordered in the sense you can assign indexes to its elements - the first, the second and so on. Being able to store data in sequences is very useful for data science.

There are two ways you can have sequences in Python. You can use have tuple's or list's. The difference between the two is that tuple are frozen (you cannot change them once you make them) but you can add or remove items for list's. Let's look at tuples first.

Tuples

In Python, you can create a tuple by placing comma-separated elements within parentheses.

```python
#Populating a tuple with Integers
x = (1, 2, 3)
print(x)
```

```
(1, 2, 3)
```

You can geet the first element of your tuple by using square brackets and passing the indexing. Remember that Python has an index system where the first element index is 0.

```python
#Getting the first element
x[0]
```

```
1
```

You can populate tuples with different data types:

```python
mytuple = ("vinicius", 1, 3.1415)
print(mytuple)
```

```
('vinicius', 1, 3.1415)
```

You cannot remove, add or modify the elements of a tuple! **Tuples are immutable objects in Python**

```python
mytuple[0] = "Bjorn"
```

```
TypeError: 'tuple' object does not support item assignment
▯[1;31m---------------------------------------------------------------------------▯
[0m
▯[1;31mTypeError▯[0m                                 Traceback (most recent call
last)
Cell ▯[1;32mIn[22], line 1▯[0m
▯[1;32m---->  1▯[0m  ▯[43mmytuple▯[49m▯[43m[▯[49m▯[38;5;241;43m0▯[39;49m▯[43m]▯
[49m ▯[38;5;241m=▯[39m ▯[38;5;124m"▯[39m▯[38;5;124mBjorn▯[39m▯[38;5;124m"▯[39m

▯[1;31mTypeError▯[0m: 'tuple' object does not support item assignment
```

Lists

Lists are similar to tuples, containers of many things, but they can be modified - delete elements, add elements, change existing elements. Here is a list with different objects inside (including another list):

```python
mylist = ["Vinicius", 1.0, 3.1415, ["google.com"]]
print(mylist)
```

```
['Vinicius', 1.0, 3.1415, ['google.com']]
```

You can also instantiate an empty list!

```python
myemptylist = []
print(myemptylist)
```

```
[]
```

You can modify existing elements of a list.

```python
mylist[0] = "Bjorn"
```

```python
print(mylist)
```

```
['Bjorn', 1.0, 3.1415, ['google.com']]
```

You can count how many elements are in a list (it's the same for tuples) using the `len` function.

```python
len(mylist)
```

```
4
```

You can do many things with lists. If you want to know all the things you can do with lists (and any other Python object), you can using the `help()` command:

```
help([mylist])
```

```
Help on list object:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(self, /)
 |      Return a reverse iterator over the list.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(self, /)
 |      Return the size of the list in memory, in bytes.
 |
 |  append(self, object, /)
 |      Append object to the end of the list.
 |
 |  clear(self, /)
 |      Remove all items from list.
 |
 |  copy(self, /)
 |      Return a shallow copy of the list.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  extend(self, iterable, /)
 |      Extend list by appending elements from the iterable.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  insert(self, index, object, /)
 |      Insert object before index.
 |
 |  pop(self, index=-1, /)
 |      Remove and return item at index (default last).
 |
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(self, value, /)
 |      Remove first occurrence of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  reverse(self, /)
 |      Reverse *IN PLACE*.
 |
 |  sort(self, /, *, key=None, reverse=False)
 |      Sort the list in ascending order and return None.
 |
 |      The sort is in-place (i.e. the list itself is modified) and stable (i.e.
the
 |      order of two equal elements is maintained).
 |
 |      If a key function is given, apply it once to each list item and sort
them,
 |      ascending or descending, according to their function values.
 |
 |      The reverse flag can be set to sort in descending order.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

let's try the `pop` command to remove the first element of my list.

```python
mylist.pop(0)
```

```
'Bjorn'
```

It prints which element I removed

```python
print(mylist)
```

```
[1.0, 3.1415, ['google.com']]
```

```python
type("vene")
```

```
str
```

You can also iterate over the elements of a list. Let's say you want to find out if there's a string in your list. For small lists, you can look manually. However, if it's large, you could do a for loop!

```python
mysecondlist = [1, "Vinicius"]

for element in mysecondlist:
    if type(element) == str:
        print("There is a string in my list")
```

```
There is a string in my list
```

Matrices

While lists and tuples are sequences and can only index their elements with a single number, we often need more complex structures, such as matrices, which require indexing with more than one number.

Lists and tuples are built-in elements of Python. To work with matrices, we need to use external numerical libraries. `Numpy`is the recommended library for it

```python
import numpy as np #Making a shorter name through "as" keyword
```

```python
# Creating a matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
```

```python
# Accessing elements
print("\nElement at (0, 0):", matrix[0, 0])
print("Element at (1, 2):", matrix[1, 2])
```

```

Element at (0, 0): 1
Element at (1, 2): 6
```

Slicing

You can “slice” matrix elements in very different ways in Python

```python
# Basic Slicing
print("\nFirst row:", matrix[0, :])
print("Second column:", matrix[:, 1])
```

```

First row: [1 2 3]
Second column: [2 5 8]
```

```python
# Slicing with step
print("\nEvery other element in the first row:")
every_other_element = matrix[0, 0::2]
print(every_other_element)
print("Explanation: This slices the first row to get every other element (step
size of 2).")
```

```

Every other element in the first row:
[1 3]
Explanation: This slices the first row to get every other element (step size
of 2).
```

```python
# Reverse slicing
print("\nReversed rows:")
```

```
reversed_rows = matrix[-1::-1] # Starting from last element up to the first with
step size 1
print(reversed_rows)
print("Explanation: This reverses the rows of the matrix.")
```

```


Reversed rows:
[[7 8 9]
 [4 5 6]
 [1 2 3]]
Explanation: This reverses the rows of the matrix.
```

Operations

Matrix addition is the simplest operation you can do with two matrices.

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])


matrix2 = np.array([[9, 8, 7],
                    [6, 5, 4],
                    [3, 2, 1]])

print(matrix + matrix2)
```

```
[[10 10 10]
 [10 10 10]
 [10 10 10]]
```

What about summing a matrix with a vector?

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

vector = np.array([[1, 2, 3]]) #My array has one element and inside this element
there are 3 other elements.

print("Matrix shape is:", matrix.shape)
print("Vector shape is:", vector.shape)
```

```
Matrix shape is: (3, 3)
Vector shape is: (1, 3)
```

This operation should not work, right?

```
matrix + vector
```

```
array([[ 2,  4,  6],
       [ 5,  7,  9],
       [ 8, 10, 12]])
```

In NumPy, broadcasting allows operations on arrays of different shapes. Broadcasting automatically expands the dimensions of the smaller array to match the larger array by repeating its elements along the mismatched dimensions, making the arrays compatible for element-wise operations.

It does the same for the * operation

```
matrix*vector
```

```
array([[ 1,  4,  9],
       [ 4, 10, 18],
       [ 7, 16, 27]])
```

If you want to make sure your operations follows what you know about linear algebra regarding matrix multiplication, use the `dot` operation instead of *.

For the + operation, there is no other way. You have to be intentional on the way you write the code.

```
np.dot(matrix, vector) ## that fails
```

```
ValueError: shapes (3,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)
□[1;31m---------------------------------------------------------------------------□
[0m
□[1;31mValueError□[0m                                Traceback (most recent call
last)
Cell □[1;32mIn[80], line 1□[0m
□[1;32m---->                 1□[0m                 □[43mnp□[49m□[38;5;241;43m.□
[39;49m□[43mdot□[49m□[43m(□[49m□[43mmatrix□[49m□[43m,□[49m□[43m
□[49m□[43mvector□[49m□[43m)□[49m □[38;5;66;03m## that fails□[39;00m

File □[1;32m<__array_function__ internals>:200□[0m, in □[0;36mdot□[1;34m(*args,
**kwargs)□[0m

□[1;31mValueError□[0m: shapes (3,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)
```

```
np.dot(vector, matrix) ## That works as we know from linear algebra!
```

```
array([[30, 36, 42]])
```

You can also apply functions to the elements of a numpy array:

```
np.tanh(vector)
```

```
array([[0.76159416, 0.96402758, 0.99505475]])
```

```
np.log(vector)
```

```
array([[0.        , 0.69314718, 1.09861229]])
```

```
np.tan(vector)
```

```
array([[ 1.55740772, -2.18503986, -0.14254654]])
```

Attention: Vector*Vector or Matrix*Matrix does not perform dot product. It squares each element of the vector or matrix!

```
vector*vector #Squares each element of the vector (it's not dot product)
```

```
array([[1, 4, 9]])
```

```
matrix*matrix #Squares each element of the matrix
```

```
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

```
np.dot(matrix, matrix) #Do matrix, matrix multiplication as in Linear Algebra
```

```
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])
```