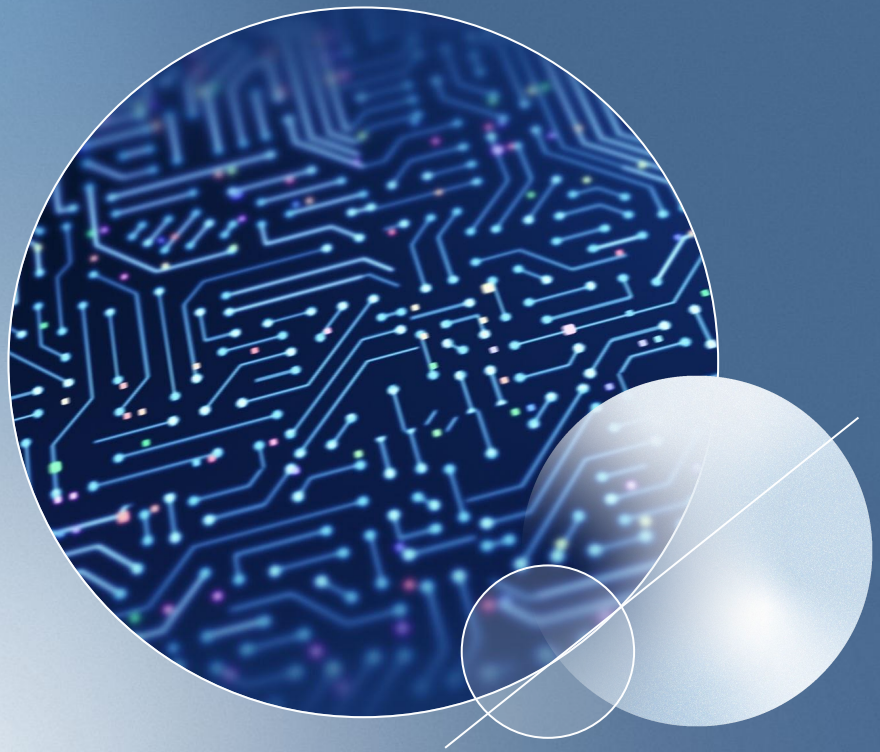
INTRODUCTION TO DATA STRUCTURES

MODULE 2

Learning Objectives



By the end of this week, students should be able to:

1. Understand the concepts and implementations of data structures in Python
2. Create plotting from Numpy array

WHAT ARE DATA STRUCTURES?

Data structures are code structures for storing and organizing data that make it easier to modify, navigate, and access information.

Data structures help to:

- Manage and utilize large datasets
- Quickly search for particular data from a database
- Build clear hierarchical or relational connections between data points
- Simplify and speed up data processing

BUILT IN DATA STRUCTURES

Text type: ***str***
Numeric types: ***int*, *float*, *complex***
Sequence type: ***list*, *tuple*, *range***
Mapping type: ***dict***
Set types: ***set*, *frozenst***
Boolean type: ***bool***
Binary types: ***bytes*, *byte array*, *memory view***

Python has a few built-in data types, and also 4 built-in data structures, lists, dictionaries, tuples, and sets. These built-in data structures come with default methods and behind the scenes optimizations that make them easy to use. You might already become familiar with some of these data types through our lab and homework, str, int, float, list, range. Each data structure has a task or situation it is most suited to solve. However some of these seem to be quite similar to each other when we do the python programming. For example, list, tuple, and range look quite similar to each other when being used as a sequence data type.

range(0,3) returns a class of immutable iterable objects that lets you iterate over them. However, it does not produce lists, and they do not store all the elements in the range in memory while **list** and **tuple** do store all the elements.

ARRAYS (LISTS)

An array is a collection of values of the same type saved under the same name.

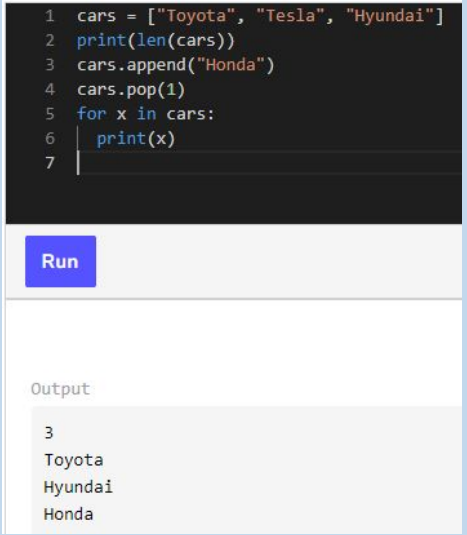
- Each value in the array is called an “element” and has an index
- value = array[index]
- The length of an array: the len() method
- Python array automatically scale up and down when elements are added/subtracted

Advantages:

- Simple to create and use data sequence
- Automatically scale to meet changing size requirement
- Used to create more complex data structures

Disadvantages:

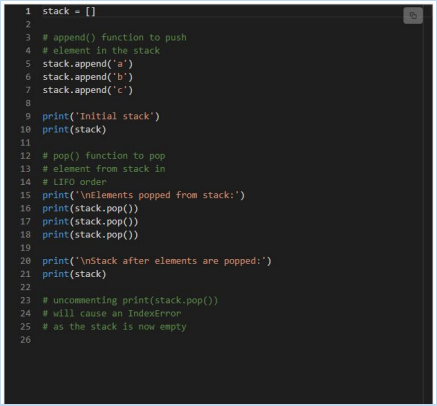
- Not optimized for scientific data (unlike Numpy array)



STACKS (LISTS)

A stack is a linear data structure that follows a particular order in which the operations are performed.

- Follows Last-in, First-out" (LIFO)(different version of queue)
- Insert new element at the top of the stack
- To access a middle element, you must first remove enough element to make it to the top
- Adding elements - push
    - Adding elements is known as a push, and removing elements is known as a pop. You can implement stacks in Python using the built-in list structure. With list implementation, push operations use the append() method, and pop operations use pop().
- Removing elements - pop



```
stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty

```

GRAPHS (DICTIONARY)

Graphs are data structure used to represent a visual of relationships between data vertices (the node of a graph). The links that connect vertices together are called edges.

- Excellent for modeling networks or web-like structures
- Used to model social network sites like Facebook

Advantages:

- Quickly convey visual information through code
- Usable for modeling a wide range of real world problems
- Simple to learn syntax

Disadvantages:

- Vertex links are difficult to understand in large graphs
- Time expensive to parse data from a graph

When written in plain text, graphs have a list of vertices and edges:

```
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}
```

In Python, graphs are best implemented using a dictionary with the name of each vertex as a key and the edges list as the values.

```python
# Create the dictionary with graph elements
graph = { "a" : ["b","c"],
          "b" : ["a", "d"],
          "c" : ["a", "d"],
          "d" : ["e"],
          "e" : ["d"]
        }

# Print the graph
print(graph)
```

QUEUES (COLLECTIONS.DEQUE)

An queue is a linear data structure that stores data in a "first in, first out" (FIFO) order.

- You can only pull the oldest element
- Great for order-sensitive tasks like online order processing and voicemail storage
- Use a Python list with append() and pop() methods to implement a queue. However, this is inefficient because lists must shift all elements by one index whenever you add a new element to the beginning.
    - Instead, it's best practice to use the deque class from Python's collections module.
- Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue").
- Deques are optimized for the append and pop operations. The deque implementation also allows you to create double-ended queues, which can access both sides of the queue through the popleft() and popright() methods.

Advantages:

- Automatically orders data chronologically
- Scales to meet size requirements
- Time efficient with deque class

Disadvantages:

- Can only access data on the ends

```python
from collections import deque

# Initializing a queue
q = deque()

# Adding elements to a queue
q.append('a')
q.append('b')
q.append('c')

print("Initial queue")
print(q)

# Removing elements from a queue
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())

print("\nQueue after removing elements")
print(q)

# Uncommenting q.popleft()
# will raise an IndexError
# as queue is now empty
```

```
Output                                                    0.95s

Initial queue
deque(['a', 'b', 'c'])

Elements dequeued from the queue
a
b
c

Queue after removing elements
```

A linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

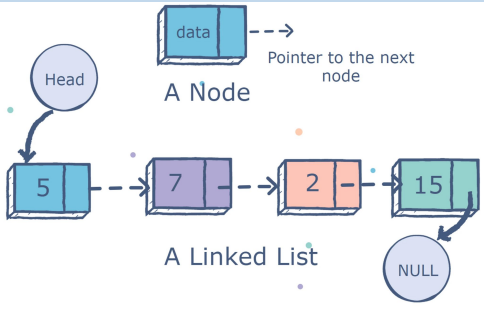
- Sequential collection of data - use relational pointers on each data node to link to the next node in the list
- Individual links only have a connection to their immediate neighbors
- All the links together form a larger structure
- Unlike arrays, linked lists do not have objective positions in the list. Instead, they have relational positions based on their surrounding nodes.
- The first node in a linked list is called the head node, and the final is called the tail node, which has a null pointer.
- Arrays allow random access and require less memory per element (do not need space for pointers) while lacking efficiency for insertion/deletion operations and memory allocation. On the contrary, linked lists are dynamic and have faster insertion/deletion time complexities.

Advantages:

- Efficient insertion and deletion
- Simpler to reorganize than arrays
- Useful as a starting point for advanced data structures like graphs

Disadvantages:

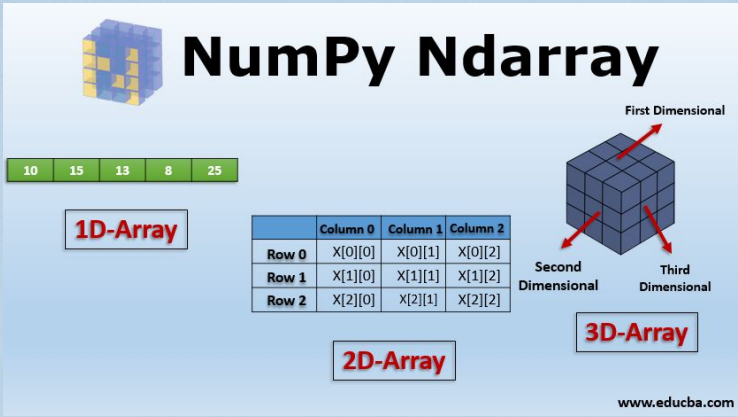
- Storage of pointers with each data point increases memory usage
- Must always traverse the linked list from head node to find a specific element



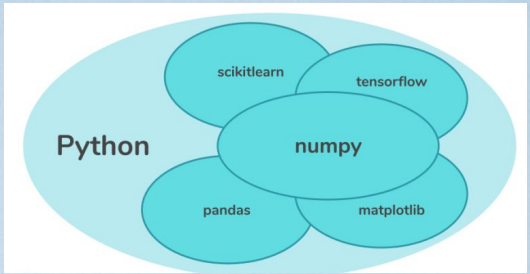
LINKED LISTS (CLASS)

INTRO: NUMPY ARRAYS

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.



At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.



CREATE A NUMPY ARRAY FROM PYTHON LIST

```
>>> import numpy as np
>>> a = np.array([2, 3, 4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

Numpy Arrays versus Python Lists

What is the difference?

1. Size - Numpy data structures take up less space
2. Performance - Numpy array runs faster
3. Functionality - Numpy has optimized functions such as built-in linear algebra operations

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> b = a[:2]
>>> b += 1
>>> print('a =', a, '; b =', b)
a = [2 3 3 4 5 6] ; b = [2 3]
```

```
>>> a = np.array([1, 2, 3, 4])
>>> b = a[:2].copy()
>>> b += 1
>>> print('a = ', a, 'b = ', b)
a =  [1 2 3 4] b =  [2 3]
```