Data Structures in Python

and their properties

Sukrit Gupta and Nitin Auluck

February 19, 2024

Outline

1 Some definitions

2 A scenario

3 Lists in Python

4 Other structured data types in Python
- Dictionaries
- Tuples
- Sets

Acknowledgement and disclaimer

All mistakes (if any) are mine.
I have used several other sources which I have referred to in the appropriate places.

Section 1

Some definitions

An example in architecture

- Data structures and algorithms are central to the development of good quality computer programs. Can be understood by the following diagram from *Aho, Hopcroft, and Ullman* (1983).
- Mathematical Model $\Longrightarrow$ Abstract Data Type (ADT) $\Longrightarrow$ Data Structures
- Informal Algorithm $\Longrightarrow$ Pseudo Language Program $\Longrightarrow$ Program in C, Python, et cetera.
- Example: A "Queue" is an ADT which can be defined as a sequence of elements with operations such as null(Q), empty(Q), enqueue(x, Q), and dequeue(Q). This can be implemented using data structures such as an array, singly linked list, et cetera.

What is an Abstract Data Type?

An Abstract Data Type (ADT) is a class of objects whose logical behavior is defined by a set of values and a set of operations.

- The operations can take as operands instances and non-instances of the ADT.
- Similarly results need not be instances of the ADT
- At least one operand or the result is of the ADT type in question.

For example, integers are an ADT, defined as the values {..., -2, -1, 0, 1, 2, ...} and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc., which behave according to familiar mathematics (with care for integer division), independently of how the integers are represented by the computer.

Data Structures

Data Structures are an implementation of an ADT and a translation into statements of a programming language.

- the **declarations** that define a variable to be of that ADT type; and
- the **operations** defined on the ADT (using procedures of the programming language)

An ADT implementation chooses a data structure to represent the ADT. Each data structure is built up from the basic data types of the underlying programming language using the available data structuring facilities, such as arrays, records (structures in C), pointers, files, sets, etc.

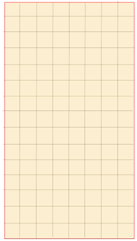
Section 2

A scenario

An example in architecture

- You're a student who has just started university.
- This is your first day at the university. You reach a building that contains your lecture halls.
- You see this building:

- A model to represent the building?



- What does the above look like?

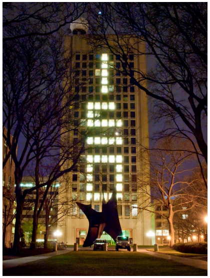


Figure: A matrix with 0s and 1s.

Go beyond 0s and 1s?

Actually, have a look at this link

Matrices

In mathematics, a matrix (plural matrices) is a rectangular array or table of numbers, symbols, or expressions, arranged in rows and columns, which is used to represent a mathematical object or a property of such an object. A matrix containing m rows and n columns is represented as:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = (a_{ij}) \in \mathbb{R}^{m \times n}.$$

Matrices

Denote a matrix by an uppercase bold letter.
BTW, what is a matrix with a single dimension called?

Matrix Operations: Addition/subtraction

The sum $\mathbf{A} + \mathbf{B}$ of two $m \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$ is calculated entrywise:

$$(\mathbf{A} + \mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$$

where $1 \leq i \leq m$ and $1 \leq j \leq n$.

$$\begin{bmatrix} 1 & 3 & 1 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 5 \\ 7 & 5 & 0 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 & 1+5 \\ 1+7 & 0+5 & 0+0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 6 \\ 8 & 5 & 0 \end{bmatrix}$$

Matrix Operations: Scalar multiplication

The product $c \cdot \mathbf{A}$ of a number c (also called a scalar) and a matrix $\mathbf{A}$ is computed by multiplying every entry of $\mathbf{A}$ by c:

$$(cA)_{i,j} = c \cdot A_{i,j}$$

Matrix Operations: Transposition

The transpose of an $m \times n$ matrix A is the $n \times m$ matrix A^T formed by turning rows into columns and vice versa:

$$(A^T)_{i,j} = A_{j,i}$$

Example

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & 7 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 \\ 2 & -6 \\ 3 & 7 \end{bmatrix}$$

Matrix Operations: Matrix multiplication

Multiplication of two matrices is defined if and only if **the number of columns of the left matrix** is the same as the **number of *r*ows of the *r*ight matrix**.

If **A** is an $m \times n$ matrix and **B** is an $n \times p$ matrix, then their matrix product **AB** is the $m \times p$ matrix whose entries are given by dot product of the corresponding row of **A** and the corresponding column of **B**:

$$[\mathbf{AB}]_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \cdots + a_{i,n}b_{n,j} = \sum_{r=1}^{n} a_{i,r}b_{r,j}$$

where $1 \leq i \leq m$ and $1 \leq j \leq p$.

Section 3

Lists in Python

Lists

```
A = ["a", "b", 1, 2]
```

ADT: List and what is the data structure behind it? Contiguous array of references. We'll see.

- Lists are ordered.
- Lists are mutable.
- Lists allow duplicates.

Lists in memory

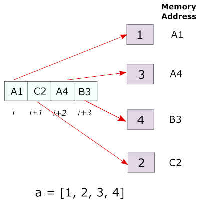


Figure: List Allocation

- Python list can contain elements of multiple types.
- Python list implementation uses two structures:
  - For storage of list elements: a *contiguous array of references to other objects*. Why?
  - Internal data structure for the list: Pointer to the above array and the array's length is stored in a list head structure.
- So when you access an element from a Python list, you perform two operations, get the address for array element and then read the value at the address.

```
>>> lst = [1, 2, 3, 257]
>>> hex(id(lst))
'0x236330edf88'
>>> hex(id(lst[0])) #any random location in memory
'0x7ffdf176a190'
>>> hex(id(lst[3])) #memory location where element at idx 3 is stored
'0x7ffdf176a1b0'
```

The example clearly shows that the memory address of the list object is different from its items.
It makes sense since a list is a collection of objects, each of its items has its own identity and is a separate object with a different memory address.

Mutability again ...

What happens when we add a new item to a list. Does the memory address change? Let's test it.

```
>>> lst = [1, 2, 3]
>>> hex(id(lst))
'0x23633104888'
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
>>> hex(id(lst))
'0x23633104888'
>>> print(hex(id(lst + [1])) #returns a new list
```

Interesting, the memory address for the list remains the same. The reason is that a list is a mutable object, and if you add items to it, the object is still the same object with one more item.

Aliasing

If you instantiate different variables from a mutable object, all of them will change if you make any change to the object. Let me show it with a simple code.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst1.append(4)
>>> lst2
[1, 2, 3, 4]
>>> lst2.append(5)
>>> lst1
[1, 2, 3, 4, 5]
```

One way to get a separate copy of a mutable object is to use copy method.

Some common ops done with lists

- append
- insert
- extend
- remove
- del
- pop
- reverse
- sort
- loop through a list
- loop through the list indices

Side note: Example with Lists

Problem: Based on a list of fruits, you want a new list, containing only the fruits with the letter “a” in the name.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

Side note: List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

```
newlist = [expression for item in iterable if condition]
```

List Comprehension

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

Use cases for lists

- (Do not quote me on this) Python lists can store data of the same type (other types too) and can be therefore used to perform operations on it. Matrix ops vs. scalar ops have a huge difference in time efficiency.
- Use lists when you care about order: Lists are used as the sequence in the for loop.
- Lists are not preferred when you want to search something in them.

Section 4

Other structured data types in Python

Back to the university scenario

You're now a student in the university.

The university wants to store your information as unique roll number, name and hand phone number.

If you had to search for a student's hand phone number from their roll number using lists how would you do it?

Searching in a list

You could do something like:

```
    roll_nos = ['1', '2', '3']
    names = ['Sukrit', 'Swapnil', 'Sudarshan']
    phones = ['754', '755', '756']
```

If you had to look for Sukrit's contact number, you would have to first search for 'Sukrit' from the roll_nos list. Then use the corresponding index to get the number from the third list.
Let's say that there are n people and it takes you k ms to go through one element in the list. Average time will be $0.5nk$ (with linear search).

What if we want to do better?

Two probable options:

- Better search algorithm; or
- A different data structure

4.1 Dictionaries

Let's discuss the latter: dict()

A dictionary is a collection which stores data in key: value pairs.
A dictionary is:

- *ordered* (Python 3.7 onwards): the items have a defined order, and that order will not change;
- mutable: We can change, add or remove items after the dictionary has been created; and
- does not allow duplicates: cannot have two items with the same key.

ADT: Dictionary. What is the data structure in Python? Hash Tables.

The same example with dict()

```
names_phones_dict = {'Sukrit': '754',
                     'Swapnil': '755',
                     'Sudarshan': '756'}
search_for = 'Swapnil'
print(names_phones_dict[search_for])
```

With dict(), average search time is a constant i.e. it is independent of the number of keys.
How is this happening?

Behind the scenes: dict()

Python dictionaries are implemented using hash tables. The hashes are usually used to index a fixed-size table called a hash table.

At the backend, you basically have an array whose indices are obtained using a hash function on the keys. So basically:

$$array\_index = f(dict\_key)$$

Hash Tables: Hash functions

A hash function is any function that can be used to map data of arbitrary size to fixed-size values.

For an input x, the hash function f returns the hash value (or simply hash) given by y:

$$y = f(x)$$

The average search time for dict() is a constant i.e. it is independent of the number of keys. Can someone tell me now how this is possible/how Python dictionaries are implemented?

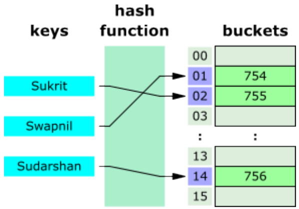


Figure: Hash Table

Obtaining the *array_index* from the f takes constant time. Accessing the array element at index *array_index* also takes constant time. Therefore, the whole operation takes constant time.

Hash collisions

Let's say we add 'Sudeepta' to our dict(): names_phones_dict = {'Sukrit': '754', 'Swapnil': '755', 'Sudarshan': '756', 'Sudeepta': '757'}.

Let's assume that Sukrit and Sudeepta have the same hash, i.e. f('Sukrit') = f('Sudeepta').

These are referred to as *collisions*, i.e. different keys have the same hash.

Hash collisions

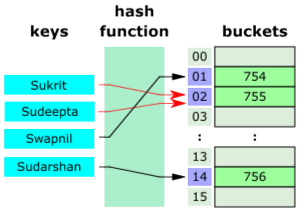


Figure: Hash Collisions

No matter what kind of a hash function you design, there will always be the possibility of collisions. Why?
Either 'Sukrit' or 'Sudeepta' will have to be stored at another location.
Python performs collision resolution by a scheme known as *open addressing*. You can read more about it here.

4.2 Tuples

Tuples

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and immutable.
- Tuples are written with round brackets ().
- Tuples are implemented in the memory like Python Lists, except that the mutable operations like `append` and `__setitem__` are not implemented for tuples.

Tuples:

At first sight, it might seem that lists can always replace tuples.

Using a tuple instead of a list can give the programmer and the interpreter a hint that the data should not be changed.

Tuples are commonly used as the equivalent of a dictionary without keys to store data.

4.3 Sets

Sets

- A set is a collection which is unordered, mutable (set items are immutable, but you can remove items and add new items), unindexed and do not allow duplicate values.
- Sets are written with curly brackets {}.
- How are sets implemented in Python? One of the primary operations on a set is checking if an element is in the set.
- The original set implementation actually *was* a dict with dummy values. Now, dict and set implementations have diverged significantly, but they still use hash tables primarily.

Sets

Allow you to perform set operations in Python.

```
A = {0, 2, 4, 6, 8};
B = {1, 2, 3, 4, 5};

# union
print("Union :", A | B)

# intersection
print("Intersection :", A & B)

# difference
print("Difference :", A - B)
```

What did we learn today?

1. Some definitions
2. A scenario
3. Lists in Python
4. Other structured data types in Python
   - Dictionaries
   - Tuples
   - Sets

Thank you!