Lecture 6 - Data Structures in Python

Adnan Ferdous Ashrafi

Stamford University Bangladesh

Table of Contents

Data Structures in Python

Data Structure

Data structures are the fundamental constructs around which you build your programs. Each data structure provides a particular way of organizing data so it can be accessed efficiently, depending on your use case. Python ships with an extensive set of data structures in its standard library.

Some of the most common data structures are as follows:

1. Lists or Arrays
2. Tuples
3. Dictionaries

Lists

Definition

Like a string, a list is a **sequence of values**. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes items.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```python
[10, 20, 30, 40]
['CSI 115', 'Computer and Programming Concept', 'CSE-S-74-A']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list:

```python
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

Lists

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>>> burgers = ['Chicken', 'Tandoori', 'BBQ']
>>> numbers = [42, 123]
>>> empty = []
>>> print(burgers, numbers, empty)
['Chicken', 'Tandoori', 'BBQ'] [42, 123] []
```

Lists- Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> burgers[0]
'Chicken'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

Lists- Lists are mutable

Lists are represented by boxes with the word “list” outside and the elements of the list inside. *burgers* refers to a list with three elements indexed 0, 1 and 2. numbers contains two elements; the previous example shows that the value of the second element has been reassigned from 123 to 5. *empty* refers to a list with no elements.
List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an Index-Error.
- If an index has a negative value, it counts backward from the end of the list.

The *in* operator also works on lists.

```
>>> burgers = ['Chicken', 'Tandoori', 'BBQ']
>>> 'Chicken' in burgers
True
>>> 'Vegetable' in burgers
False
```

Lists- Traversing a list

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```python
for burger in burgers:
    print(burger)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions *range* and *len*:

```python
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses (i.e. walks through) the list and updates each number with double of its original value.

Lists- Traversing a list

A for loop over an empty list never runs the body:

```python
for x in []:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```python
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Lists- List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

The * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

Lists- List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Lists- List slices

Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.
A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

List methods- Appending to Lists

Python provides methods that operate on lists. For example, *append* adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

List methods- Extending Lists

extend takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

This example leaves *t*2 unmodified.

List methods- Sorting Lists

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

Map, filter and reduce- Reduce

To add up all the numbers in a list, you can use a loop like this:

```python
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

total is initialized to 0. Each time through the loop, x gets one element from the list.
The += operator provides a short way to update a variable and is an *augmentedassignmentstatement*.
As the loop runs, total accumulates the sum of the elements; a variable used this way is sometimes called an *accumulator*.

Map, filter and reduce- Reduce

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, *sum*:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Map, filter and reduce- Map

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

res is initialized with an empty list; each time through the loop, we append the next element. So res is another kind of accumulator.
An operation like *capitalize_all* is sometimes called a map because it **"maps"** a function (in this case the method capitalize) onto each of the elements in a sequence.

Map, filter and reduce- Filter

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```python
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

isupper is a string method that returns *True* if the string contains only upper case letters. An operation like *only_upper* is called a filter because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of map, filter and reduce.

Tuples

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable. A tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Tuples

Another way to create a tuple is the built-in function tuple.

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

And the slice operator selects a range of elements.

```
>>> t[1:3]
('b', 'c')
```

Tuples

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
"TypeError: object doesn't support item assignment"
```

Because tuples are immutable, you can’t modify the elements. But you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

This statement makes a new tuple and then makes t refer to it.

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 4, 2000000) < (0, 3, 4)
False
```

Tuples- Tuple Assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; tuple assignment is more elegant:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Tuples- Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute $x//y$ and then $x\%y$. It is better to compute them both at the same time.

The built-in function *divmod* takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

Dictionaries

Definition

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.
A dictionary contains a collection of indices, which are called keys, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a key-value pair or sometimes an item.

In mathematical language, a dictionary represents a mapping from keys to values, so you can also say that each key “maps to” a value. As an example, we’ll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

Dictionaries

The function *dict* creates a new dictionary with no items. Because *dict* is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

The curly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key ’one’ to the value ’uno’. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> eng2sp
{'one': 'uno'}
```

Dictionaries

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print eng2sp, you might be surprised:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs might not be the same. If you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.
But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> eng2sp['two']
'dos'
```

Dictionaries- Looping and dictionaries

Suppose you are given a string and you want to count how many times each letter appears.There are several ways you could do it. One good way is:

You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary.After that you would increment the value of an existing item. This can be implemented as such:

```python
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Here's how it works:

```python
h = histogram('brontosaurus')
# output
h{'a':1,'b':1,'o':2,'n':1,'s':2,'r':2,'u':2,'t':1}
```

Dictionaries- Looping and dictionaries

If you use a dictionary in a *for* statement, it traverses the keys of the dictionary. For example, *print_hist prints* prints each key and the corresponding value:

```python
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Here's what the output looks like:

```python
h = histogram('parrot')
print_hist(h)
# output
a 1
p 1
r 2
t 1
o 1
```

NumPy Arrays

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

NumPy Arrays- Creating NumPy Arrays

NumPy is used to work with arrays. The array object in NumPy is called *ndarray* (n-dimensional array). We can create a NumPy ndarray object by using the *array*() function.

```python
import numpy as np
# Let's create a 1x5 dimensional array
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

```python
import numpy as np

arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

NumPy Arrays- Dimensions

A dimension in arrays is one level of array depth (nested arrays).

0-D Array: 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```python
import numpy as np
arr = np.array(42)
print(arr)
```

1-D Array: An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

```python
import numpy as np
arr = np.array([1, 2, 3, 4, 5]) # 1x5 matrix
print(arr)
```

NumPy Arrays- Dimensions

2-D Array: An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

```python
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]]) # 2x3 matrix
print(arr)
```

3-D array: An array that has 2-D arrays (matrices) as its elements is called 3-D array. These are often used to represent a 3rd order tensor.

```python
import numpy as np
# 2x2x3 matrix
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

NumPy Arrays- Dimensions

Check Number of Dimensions: NumPy Arrays provides the *ndim* attribute that returns an integer that tells us how many dimensions the array have.

```python
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Higher Dimensional Arrays: An array can have any number of dimensions. When the array is created, you can define the number of dimensions by using the *ndmin* argument.

```python
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

Further Reading

- Chapter 10, 11 and 12 of Think Python(2nd Edition) - Allen B. Downey
- Chapter 3, 4 and 6 of Python Crash Course - Eric Matthes
- Python Official Website
- Lecture 6 Jupyter Notebook

Thank you.
Any Questions?