Python Data Structures

DA2303 – Python Programming

Data Structures

- Data structures are basically just that - they are *structures* which can hold some *data* together.

- In other words, they are used to store a collection of related data.

- There are four built-in data structures in Python:
  - *List*
  - *Tuple*
  - *Dictionary*
  - *Set*.

- We will see how to use each of them and how they make life easier for us.

Table of contents

01

Sequence

Overview and Example of
List, Tuple and Dictionary
(Data Structure)

List

- A list is a data structure that holds an ordered collection of items i.e. you can store a *sequence* of items in a list.

- This is easy to imagine if you can think of a shopping list where you have a list of items to buy, except that you probably have each item on a separate line in your shopping list whereas in Python you put commas in between them.

- The list of items should be enclosed in square brackets so that Python understands that you are specifying a list.

List (cont.)

- Once you have created a list, you can add, remove or search for items in the list.

- Since we can add and remove items, we say that a list is a *mutable* data type i.e. this type can be altered.

Quick Introduction To Objects and Classes

- A list is an example of usage of objects and classes.
- When we use a variable i and assign a value to it, say integer 5 to it, you can think of it as creating an *object* (i.e. instance) i of *class* (i.e. type) int.
- A class can also have *methods* i.e. functions defined for use with respect to that class only.

Quick Introduction To Objects and Classes (cont.)

- You can use these pieces of functionality only when you have an object of that class.
- For example, Python provides an append method for the list class which allows you to add an item to the end of the list.
- For example, mylist.append('an item') will add that string to the list mylist.
- Note the use of dotted notation for accessing methods of the objects.

Quick Introduction To Objects and Classes (cont.)

- A class can also have *fields* which are nothing but variables defined for use with respect to that class only.
- You can use these variables/names only when you have an object of that class.
- Fields are also accessed by the dotted notation, for example, mylist.field.

Example: List

- save as ds_using_list.py

```python
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print('I have', len(shoplist), 'items to purchase.')

print('These items are:', shoplist)

print('\nI also have to buy rice.')
shoplist.append('rice')
print('My shopping list is now', shoplist)

print('I will sort my list now')
shoplist.sort()
print('Sorted shopping list is', shoplist)

print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

Example: List (cont.)

- Run the script:

```
$ python ds_using_list.py
I have 4 items to purchase.
These items are: ['apple', 'mango', 'carrot', 'banana']

I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

Example: List - How it works

- The variable shoplist is a shopping list for someone who is going to the market.

- In shoplist, we only store strings of the names of the items to buy but you can add *any kind of object* to a list including numbers and even other lists.

- We print the content of the shoplist

- Next, we add an item to the list using the append method of the list object, as already discussed before.

Example: List - How it works (cont.)

- Then, we check that the item has been indeed added to the list by printing the contents of the list by simply passing the list to the print function which prints it neatly.

- Then, we sort the list by using the sort method of the list. It is important to understand that this method affects the list itself and does not return a modified list - this is different from the way strings work. This is what we mean by saying that lists are *mutable* and that strings are *immutable*.

- Next, when we finish buying an item in the market, we want to remove it from the list. We achieve this by using the del statement. Here, we mention which item of the list we want to remove and the del statement removes it from the list for us.

Example: List - How it works (cont.)

- We specify that we want to remove the first item from the list and hence we use del shoplist[0] (remember that Python starts counting from 0).
- If you want to know all the methods defined by the list object, see help(list) for details.

https://docs.python.org/3.9/tutorial/introduction.html#lists

https://docs.python.org/3.9/tutorial/datastructures.html#more-on-lists

Tuple

- Tuples are used to hold together multiple objects. Think of them as similar to lists, but without the extensive functionality that the list class gives you.

- One major feature of tuples is that they are *immutable* like strings i.e. you cannot modify tuples.

- Tuples are defined by specifying items separated by commas within an optional pair of parentheses.

- Tuples are usually used in cases where a statement or a user-defined function can safely assume that the collection of values (i.e. the tuple of values used) will not change.

Example: Tuple

- save as ds_using_tuple.py

```
# I would recommend always using parentheses
# to indicate start and end of tuple
# even though parentheses are optional.
# Explicit is better than implicit.
zoo = ('python', 'elephant', 'penguin')
print('Number of animals in the zoo is', len(zoo))

new_zoo = 'monkey', 'camel', zoo # parentheses not required but are a good idea
print('Number of cages in the new zoo is', len(new_zoo))
print('All animals in new zoo are', new_zoo)
print('Animals brought from old zoo are', new_zoo[2])
print('Last animal brought from old zoo is', new_zoo[2][2])
print('Number of animals in the new zoo is', len(new_zoo)-1+len(new_zoo[2]))
```

Example: Tuple (cont.)

- Run the script

```
$ python ds_using_tuple.py
Number of animals in the zoo is 3
Number of cages in the new zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python', 'elephant', 'penguin'))
Animals brought from old zoo are ('python', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

Example: Tuple - How it works

- The variable zoo refers to a tuple of items.
- We see that the len function can be used to get the length of the tuple.
- We are now shifting these animals to a new zoo since the old zoo is being closed.
- Therefore, the new_zoo tuple contains some animals which are already there along with the animals brought over from the old zoo.
- Back to reality, note that a tuple within a tuple does not lose its identity.

Example: Tuple - How it works (cont.)

- We can access the items in the tuple by specifying the item's position within a pair of square brackets just like we did for lists.
- This is called the *indexing* operator.
- We access the third item in new_zoo by specifying new_zoo[2] and we access the third item within the third item in the new_zoo tuple by specifying new_zoo[2][2].
- This is pretty simple once you've understood the idiom.

Dictionary

- A dictionary is like an address-book where you can find the address or contact details of a person by knowing only his/her name i.e. we associate *keys* (name) with *values* (details).

- Note that the key must be unique just like you cannot find out the correct information if you have two persons with the exact same name.

- Note that you can use only immutable objects (like strings) for the keys of a dictionary but you can use either immutable or mutable objects for the values of the dictionary.

- This basically translates to say that you should use only simple objects for keys.

Dictionary (cont.)

- Pairs of keys and values are specified in a dictionary by using the notation d = {key1 : value1, key2 : value2 }.

- Notice that the key-value pairs are separated by a colon and the pairs are separated themselves by commas and all this is enclosed in a pair of curly braces.

- Remember that key-value pairs in a dictionary are not ordered in any manner. If you want a particular order, then you will have to sort them yourself before using it.

- The dictionaries that you will be using are instances/objects of the dict class.

Example: Dictionary

- save as ds_using_dict.py

```python
# 'ab' is short for 'a'ddress'b'ook
ab = {
    'Swaroop': 'swaroop@swaroopch.com',
    'Larry': 'larry@wall.org',
    'Matsumoto': 'matz@ruby-lang.org',
    'Spammer': 'spammer@hotmail.com'
}

print("Swaroop's address is", ab['Swaroop'])

# Deleting a key-value pair
del ab['Spammer']

print('\nThere are {} contacts \
in the address-book\n'.format(len(ab)))

print(ab)

# Adding a key-value pair
ab['Guido'] = 'guido@python.org'
print(ab)
```

Example: Dictionary (cont.)

- Run the script

```
$ python ds_using_dict.py
Swaroop's address is swaroop@swaroopch.com
There are 3 contacts in the address-book

{'Swaroop': 'swaroop@swaroopch.com', 'Larry': 'larry@wall.org', 'Matsumoto':
'matz@ruby-lang.org’}
{'Swaroop': 'swaroop@swaroopch.com', 'Larry': 'larry@wall.org', 'Matsumoto':
'matz@ruby-lang.org', 'Guido': 'guido@python.org'}
```

Example: Dictionary – How it works

- We create the dictionary ab using the notation already discussed.
- We then access key-value pairs by specifying the key using the indexing operator as discussed in the context of lists and tuples.
- Observe the simple syntax.
- We can delete key-value pairs using our old friend - the del statement.
- We simply specify the dictionary and the indexing operator for the key to be removed and pass it to the del statement.

Example: Dictionary – How it works (cont.)

- There is no need to know the value corresponding to the key for this operation.
- Next, we print the content of ab.
- We can add new key-value pairs by simply using the indexing operator to access a key and assign that value, as we have done for Guido in the above case.
- We can check if a key-value pair exists using the in operator.
- For the list of methods of the dict class, see help(dict).

https://docs.python.org/3.9/tutorial/datastructures.html#dictionaries

https://docs.python.org/3.9/library/stdtypes.html#typesmapping

Sequence

- Lists, tuples and strings are examples of sequences, but what are sequences and what is so special about them?

- The major features are *membership tests*, (i.e. the in and not in expressions) and *indexing operations*, which allow us to fetch a particular item in the sequence directly.

- The three types of sequences mentioned above - lists, tuples and strings, also have a *slicing* operation which allows us to retrieve a slice of the sequence i.e. a part of the sequence.

Example: Sequence

- save as ds_seq.py

```
shoplist = ['apple', 'mango', 'carrot',
'banana']
name = 'swaroop'

# Indexing or 'Subscription' operation #
print('Item 0 is', shoplist[0])
print('Item 1 is', shoplist[1])
print('Item 2 is', shoplist[2])
print('Item 3 is', shoplist[3])
print('Item -1 is', shoplist[-1])
print('Item -2 is', shoplist[-2])
print('Character 0 is', name[0])
```

```
# Slicing on a list #
print('Item 1 to 3 is', shoplist[1:3])
print('Item 2 to end is', shoplist[2:])
print('Item 1 to -1 is', shoplist[1:-1])
print('Item start to end is', shoplist[:])

# Slicing on a string #
print('characters 1 to 3 is', name[1:3])
print('characters 2 to end is', name[2:])
print('characters 1 to -1 is', name[1:-1])
print('characters start to end is', name[:])
```

Example: Sequence (cont.)

- Run the script.

```
Item 0 is apple
Item 1 is mango
Item 2 is carrot
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Character 0 is s
Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']
characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

Example: Sequence – How it works

- First, we see how to use indexes to get individual items of a sequence.
- This is also referred to as the *subscription operation*.
- Whenever you specify a number to a sequence within square brackets as shown above, Python will fetch you the item corresponding to that position in the sequence.
- Remember that Python starts counting numbers from 0.
- Hence, shoplist[0] fetches the first item and shoplist[3] fetches the fourth item in the shoplist sequence.

Example: Sequence – How it works (cont.)

- The index can also be a negative number, in which case, the position is calculated from the end of the sequence.
- Therefore, shoplist[-1] refers to the last item in the sequence and shoplist[-2] fetches the second last item in the sequence.
- The slicing operation is used by specifying the name of the sequence followed by an optional pair of numbers separated by a colon within square brackets.
- Note that this is very similar to the indexing operation you have been using till now.
- Remember the numbers are optional but the colon isn't.

Example: Sequence – How it works (cont.)

- The first number (before the colon) in the slicing operation refers to the position from where the slice starts and the second number (after the colon) indicates where the slice will stop at.

- If the first number is not specified, Python will start at the beginning of the sequence.

- If the second number is left out, Python will stop at the end of the sequence.

- Note that the slice returned *starts* at the start position and will end just before the *end* position i.e. the start position is included but the end position is excluded from the sequence slice.

Example: Sequence – How it works (cont.)

- Thus, shoplist[1:3] returns a slice of the sequence starting at position 1, includes position 2 but stops at position 3 and therefore a *slice* of two items is returned.

- Similarly, shoplist[:] returns a copy of the whole sequence.

- You can also do slicing with negative positions.

- Negative numbers are used for positions from the end of the sequence.

- For example, shoplist[:-1] will return a slice of the sequence which excludes the last item of the sequence but contains everything else.

Example: Sequence – How it works (cont.)

- You can also provide a third argument for the slice, which is the *step* for the slicing (by default, the step size is 1):

```python
shoplist = ['apple', 'mango', 'carrot', 'banana']

print(shoplist[::1])
print(shoplist[::2])
print(shoplist[::3])
print(shoplist[::-1])
```

output

```
['apple', 'mango', 'carrot', 'banana']
['apple', 'carrot']
['apple', 'banana']
['banana', 'carrot', 'mango', 'apple']
```

Example: Sequence – How it works (cont.)

- Notice that when the step is 2, we get the items with position 0, 2,... When the step size is 3, we get the items with position 0, 3, etc.

- Try various combinations of such slice specifications using the Python interpreter interactively i.e. the prompt so that you can see the results immediately.

- The great thing about sequences is that you can access tuples, lists and strings all in the same way!

02

Set

Overview and Example of
Set (Data Structure)

Set

- Sets are *unordered* collections of simple objects.
- These are used when the existence of an object in a collection is more important than the order or how many times it occurs.
- Using sets, you can test for membership, whether it is a subset of another set, find the intersection between two sets, and so on.

Example: Set

```python
bri = set(['brazil', 'russia', 'india'])

print('india' in bri)
print('usa' in bri)

bric = bri.copy()
bric.add('china')
print(bric.issuperset(bri))

bri.remove('russia')
print(bri & bric) # OR bri.intersection(bric)
```

output

```
True
False
True
{'brazil', 'india'}
```

Example: Set – How it works

- If you remember basic set theory mathematics from school, then this example is fairly self-explanatory.
- But if not, you can google "set theory" and "Venn diagram" to better understand our use of sets in Python.

03

References

Explanation how python handle objects

References

- When you create an object and assign it to a variable, the variable only *refers* to the object and does not represent the object itself!
- That is, the variable name points to that part of your computer's memory where the object is stored.
- This is called *binding* the name to the object.
- Generally, you don't need to be worried about this, but there is a subtle effect due to references which you need to be aware of.

Example: References

- save as ds_reference.py

```
print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
# mylist is just another name pointing to the same object!
mylist = shoplist

# I purchased the first item, so I remove it from the list
del shoplist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that both shoplist and mylist both print
# the same list without the 'apple' confirming that
# they point to the same object

print('Copy by making a full slice')
# Make a copy by doing a full slice
mylist = shoplist[:]
# Remove first item
del mylist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that now the two lists are different
```

Example: References (cont.)

- Run the script

```
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

Example: References – How it works

- Most of the explanation is available in the comments.
- Remember that if you want to make a copy of a list or such kinds of sequences or complex objects (not simple *objects* such as integers), then you have to use the slicing operation to make a copy.
- If you just assign the variable name to another name, both of them will ''refer'' to the same object and this could be trouble if you are not careful.

Thanks

Do you have any question?

jailani.rahman@pb.edu.bn
(+673) 223 4466 ext 241
Unit 6.01, Ong Sum Ping Campus,
Politeknik Brunei.