Lists, Tuples, Dictionaries

Now that we're familiar with the scalar types, let's move on to various collections.

In this section we're going to cover a few new types:

1. List
2. Dict
3. Tuple
4. Set

As well as the concept of mutability.

Lists

Many languages only have one built-in complex type, commonly referred to as an array. You are probably familiar with syntax like [1, 2, 3, 4].

In Python, [1,2,3,4] defines what we call a list. A list can also be declared as list(1, 2, 3, 4).

The simplest list is the empty list [].

(It is worth noting that the empty list evaluates to False when used in a boolean context.)

There are a few things we typically want to do with lists:

retrieve (or set) an item at a specific position via indexing

check for membership (e.g. does b_list contain the number 2?)

concatenate two lists together

get the length of the list

add an item to the list

remove an item from the list

Indexing

Like most languages we can also use ourlist[idx] to get and set an item at a specific index:

```
>>> words = ['green', 'blue', 'red']
>>> words[0]
'green'
>>> words[2] = 'orange'
>>> words
['green', 'blue', 'orange']
```

Unlike a lot of other languages, the indexing syntax in Python is quite powerful.

You can get a range of elements using what is called a slice:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[1:4]
['dog', 'snake', 'fish']
```

This gets the elements starting at 1 and ending at 3. Note that the range is inclusive on the lower end only. (This seems odd but is quite helpful when doing math on indices.)

You can also use negative indices, which count from the back of the list:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[-1]
['elephant']
>>> words[1:-1]    # trims off first and last
['dog', 'snake', 'fish']
```

You can specify a third part of the slice called the stride. By default the stride is one, meaning it steps through from start to end one at a time.

Let's try changing that:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[0:5:2]
['cat', 'snake', 'elephant']
```

That took every other element.

You can omit any part of the slice and it'll default to start, end and 1. So let's look at one last example:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[::-1]
['elephant', 'fish', 'snake', 'dog', 'cat']
```

Checking For Membership with in

The first thing we're going to do is to check if an item is a member of a list:

```
>>> a_list = [1, 2, 3, 4]
>>> 4 in a_list
```

```
True
>>> 'five' in a_list
False
```

This is actually a new operator, in checks for containment in a collection. It can also be used on strings:

```
>>> 'hello' in 'hello world'
True
```

Concatenating Lists

Let's try +:

```
>>> a_list = [1, 2, 3, 4]
>>> b_list = ['cinco', 'six', 'VII']
>>> a_list + b_list
[1, 2, 3, 4, 'cinco', 'six', 'VII']
```

len

There's a special builtin function to get the length of a list:

```
>>> len([1, 2, 3, 4])
4
```

Appending & Removing Items

Sometimes you just want to append a single item. There isn't an operator for this but we'll use a method of the list:

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

And we'll use .pop() to remove the first item:

```
>>> a_list.pop()
1
>>> a_list
```

```
[2, 3, 4, 5]
```

Pop by default removes the first item and returns it. You can also pass an argument to pop to remove a different item.

You can also remove items by value using .remove():

```
>>> a_list = [1, 2, 3, 4]
>>> a_list.remove(2)
>>> a_list
[1, 3, 4]
>>> a_list = [2, 2, 2, 2]
>>> a_list.remove(2)       # will only remove the first occurence
>>> a_list
[2, 2, 2]
```

There are plenty of other methods on list to explore.

See the documentation on common sequence operations and mutable sequence types to see the other methods and builtins that operate on lists.

Python List Methods:

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value

extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Example:

```
l1 = [1,4,5,6,6,4]
l1.append(7) # append 7 at last of the list
print(l1)
l2=l1.copy() #l2 is copied with l1 elements
print(l2)
print(l1.count(6)) # counting number of 6 in list
l1.extend([3,4]) #extending the l1 with 3,4 elements
print(l1)
print(l1.index(7)) # finding index of 7 in l1
```

```
l1.insert(2,6) #inserting 6 at index position 2
print(l1)
l1.pop() # removing last element
print(l1)
l1.remove(6) # removing first existing 6 in l1
print(l1)
l1.reverse() # reversing the elements in l1
print(l1)
l1.sort() # sorting elements in l1
print(l1)
```

Output:

```
[1, 4, 5, 6, 6, 4, 7]
[1, 4, 5, 6, 6, 4, 7]
2
[1, 4, 5, 6, 6, 4, 7, 3, 4]
6
[1, 4, 6, 5, 6, 6, 4, 7, 3, 4]
[1, 4, 6, 5, 6, 6, 4, 7, 3]
[1, 4, 5, 6, 6, 4, 7, 3]
[3, 7, 4, 6, 6, 5, 4, 1]
[1, 3, 4, 4, 5, 6, 6, 7]
```

Strings:

A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn about Unicode from Python Unicode

Python String Operations

There are many operations that can be performed with strings which makes it one of the most used data types in Python

Concatenation of Two or More Strings

```
# Python String Operations

str1 = 'Hello'

str2 ='World!'

# using +

print('str1 + str2 = ', str1 + str2)
```

```
# using *

print('str1 * 3 =', str1 * 3)
```

When we run the above program, we get the following output:

```
str1 + str2 =  HelloWorld!
str1 * 3 = HelloHelloHello
```

Iterating Through a string

We can iterate through a string using a for loop. Here is an example to count the number of 'l's in a string.

```
# Iterating through a string
count = 0
for letter in 'Hello World':
    if(letter == 'l'):
        count += 1
print(count,'letters found')
```

When we run the above program, we get the following output:

```
3 letters found
```

Tuples

We saw earlier that a list is an ordered mutable collection. There's also an ordered immutable collection.

In Python these are called tuples and look very similar to lists, but typically written with () instead of []:

```
a_list = [1, 'two', 3.0]
a_tuple = (1, 'two', 3.0)
```

Similar to how we used list before, you can also create a tuple via tuple(1,2,3).

The difference being that tuples are immutable. This means no assignment, append, insert, pop, etc. Everything else works as it did with lists: indexing, getting the length, checking membership, etc.

Like lists, all of the common sequence operations are available.

Another thing to note is that strictly speaking, the comma is what makes the tuple, not the parentheses. In practice it is a good idea to include the parentheses for clarity and because they are needed in some situations to make operator precedence clear.

Let's look at a quick example:

```
>>> a_tuple = (1, 2, 3)
>>> another_tuple = 1, 2, 3
>>> a_tuple == b_tuple
True
```

This is also important if you need to make a single element tuple:

```
>>> x = ('one')
>>> y = ('one',)
>>> type(x)
str
>>> type(y)
tuple
```

Methods in Tuples:

Method	Description
count()	**Returns the number of times a specified value occurs in a tuple**
index()	**Searches the tuple for a specified value and returns the position of where it was found**

Examples:

t = (1,2,3,4,5,2,4,5,2)

print(t.count(2))

print(t.index(5))

Output:

3

4

Dicts

Dictionaries are a common feature of modern languages (often known as maps, associative arrays, or hashmaps) which let you associate pairs of values together. The term dictionary is a nod to this, as you can think of them as being terms/descriptions in a sense.

In Python dict is perhaps the most important type there is, when we learn about modules and classes we'll see why.

A dictionary declaration might look like this:

```
birth_year = {'Hamilton': 1755, 'Jefferson': 1743, 'Franklin': 1706, 'Adams': 1735}
```

Like list there is a second way to declare a dict:

```
sound = dict(dog='bark', cat='meow', snake='hiss')
```

And an empty dict can be declared as either {} or dict()

In Python we refer to 'dog', 'cat', and 'snake' as the keys and 'bark', 'meow', and 'hiss' as the values.

A few things we already saw on list work the same for dict:

> Similarly to how we can index into lists we use d[key] to access specific elements in the dict. There are also a number of methods available for manipulating & using data from dict.
>
> len(d) gets the number of item in the dictionary.
>
> k in d checks if k is a key in the dictionary.

d.pop(k) pops an item out of the dictionary and returns it, similarly to how list's pop method worked.

Other commonly used dict methods:

keys() - returns an iterable of all keys in the dictionary.

values() - returns an iterable of all values in the dictionary.

items() - returns an iterable list of (key, value) tuples.

Methods in Dictionary

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair

update()	**Updates the dictionary with the specified key-value pairs**
values()	**Returns a list of all the values in the dictionary**

Examples for dictionary methods:

```
d1 = {1:1,2:4,3:9,4:16,5:25}
d2=d1.copy()
print(d2)
print(d1.get(3))
print(d1.items())
print(d1.keys())
print(d1.values())
d1.pop(3)
print(d1)
d1.popitem()
print(d1)
d1.update({6:36})
print(d1)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
9
dict_items([(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)])
dict_keys([1, 2, 3, 4, 5])
dict_values([1, 4, 9, 16, 25])
{1: 1, 2: 4, 4: 16, 5: 25}
{2: 4, 4: 16, 5: 25}
{2: 4, 4: 16, 5: 25, 6: 36}
```

Sets

You may have noticed in the definition of lists above we called them ordered in addition to mutable.

Items in a list have a definite order, which is what allows us to index into them or sort them. Sometimes you don't care what the order of items is, simply whether the collection contains it or not. In this case Python has a special type that is probably not familiar to you from other languages: set.

A set is an unordered, mutable collection. In math they're typically denoted with {}, you can use the same in Python. The difference from dict syntax is that there aren't key-value pairs separated by :.

Many of the same operations you have on lists and tuples are available on set, but anything dealing with order isn't. This means you can't index into a set, or pop a specific element. (You can call s.pop() which will pop an arbitrary element out of s.)

So now we've seen an ordered mutable collection (list), an ordered immutable collection (tuple), and an unordered mutable collection (set), so you may be wondering if there is an unordered immutable. The answer is that there is, and it is called frozenset.

mutable?	Ordered	Unordered
Mutable	list	set
Immutable	tuple	frozenset

Constructing Sets:

```
a = {1,2,3}                 # set literal
b = {}                      # not a set!, ``dict`` literal
c = set()                   # empty set
d = frozenset({1,2,3})      # frozenset
e = frozenset()             # empty frozenset
f = frozenset([1,1,1,1,1])  # == frozenset({1}), remember- sets cannot contain duplicates
```

The sequence operations available on list aren't available, though some basics like len and in still work.

Instead of append, set uses add:

```
>>> a = {1, 2, 3}
```

```
>>> a.add(4)
>>> a.add(2)
>>> a
{1, 2, 3, 4}
```

Common set arithmetic operators are available as methods and operators:

method	operator
a.issubset(b)	a <= b
a.issuperset(b)	a >= b
a.union(b)	a | b
a.intersection(b)	a & b
a.difference(b)	a - b
a.symmetric_difference(b)	a ^ b
a.isdisjoint(b)	

Difference Between List and Tuple in Python:

SR.NO.	LIST	TUPLE
1	Lists are mutable	Tuples are immutable
2	Implication of iterations is Time-consuming	The implication of iterations is comparatively Faster
3	The list is better for performing operations, such as insertion and deletion.	Tuple data type is appropriate for accessing the elements
4	Lists consume more memory	Tuple consume less memory as compared to the list
5	Lists have several built-in methods	Tuple does not have many built-in methods.
6	The unexpected changes and errors are more likely to occur	In tuple, it is hard to take place.

Difference between Sets and dictionaries:

Sets	Dictionary
Set data structure is also non-homogeneous data structure but stores in single row	**Dictionary is also a non-homogeneous data structure which stores key value pairs**

Set can be represented by { }	Dictionary can be represented by { }
Set can be created using set() function	Dictionary can be created using dict() function.
Set is unordered	Dictionary is ordered