5.2 Advance Data Types

This section will cover the following advance topics in data types

- Collections

Collections

The collections module is a tresure trove of a built-in module that implements specialized container datatypes providing alternatives to Python’s ge

Name	Description
namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

ChainMap — Search Multiple Dictionaries

The ChainMap class manages a list of dictionaries, and can be used to searche through them in the order they are added to find values for associa

It makes a good **"context" container**, as it can be visualised as a stack for which changes happen as soon as the stack grows, with these change

Treat it as a view table in DB, where actual values are still stored in their respective table and we can still perform all the operation on them.

Accessing Values

The ChainMap supports the same API as a regular dictionary for accessing existing values.

```
import collections
# from collections import ChainMap

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)

print('Individual Values')
print('a = {}'.format(m['a']))
print('b = {}'.format(m['b']))
print('c = {}'.format(m['c']))
print("-"*20)

print(type(m.keys()))
print('Keys = {}'.format(list(m.keys())))
print('Values = {}'.format(list(m.values())))
print("-"*20)

print('Items:')
for k, v in m.items():
    print('{} = {}'.format(k, v))
print("-"*20)

print('"d" in m: {}'.format(('d' in m)))
```

```
Individual Values
a = A
b = B
c = C
--------------------
```

```
<class 'collections.abc.KeysView'>
Keys = ['b', 'a', 'c']
Values = ['B', 'A', 'C']
--------------------
Items:
b = B
a = A
c = C
--------------------
"d" in m: False
```

```
a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)

lst = []

for v in m.keys():
    lst.append(v)

for v in m.values():
    lst.append(v)

print(lst)
```

```
['b', 'a', 'c', 'B', 'A', 'C']
```

The child mappings are searched in the order they are passed to the constructor, so the value reported for the key 'c' comes from the a dictionary.

Reordering

The ChainMap stores the list of mappings over which it searches in a list in its maps attribute. This list is mutable, so it is possible to add new mapp

```
import collections

a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}

cm = collections.ChainMap(a, b)

print(cm.maps)
print('c = {}\n'.format(cm['c']))
# reverse the list
cm.maps = list(reversed(cm.maps)) # m = collections.ChainMap(b, a)

print(cm.maps)
print('c = {}'.format(cm['c']))
```

```
[{'a': '1', 'c': '3'}, {'b': '2', 'c': '33'}]
c = 3

[{'b': '2', 'c': '33'}, {'a': '1', 'c': '3'}]
c = 33
```

When the list of mappings is reversed, the value associated with 'c' changes.

Updating Values

A ChainMap does not cache the values in the child mappings. Thus, if their contents are modified, the results are reflected when the ChainMap is a

```
import collections

a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}

m = collections.ChainMap(a, b)
print('Before: {}'.format(m['c']))
a['c'] = '3.3'
print('After : {}'.format(m['c']))
```

```
Before: 3
After : 3.3
```

```
import collections

a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}

cm = collections.ChainMap(b, a)
print(cm.maps)
print('Before: {}'.format(cm['c']))
a['c'] = '3.3'
print('After : {}'.format(cm['c']))
```

```
[{'b': '2', 'c': '33'}, {'a': '1', 'c': '3'}]
Before: 33
After : 33
```

Changing the values associated with existing keys and adding new elements works the same way.

It is also possible to set values through the ChainMap directly, although only the first mapping in the chain is actually modified.

```
import collections

a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}

cm = collections.ChainMap(a, b)
print('Before: {}'.format(cm['c']))
cm['c'] = '3.3'
print('After : {}'.format(cm['c']))
print(a['c'])
print(b['c'])
```

```
Before: 3
After : 3.3
3.3
33
```

```
import collections

a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}

cm = collections.ChainMap(b, a)
print('Before: {}'.format(cm['c']))
cm['c'] = '3.3'
print('After : {}'.format(cm['c']))
print(a['c'])
print(b['c'])
```

```
Before: 33
After : 3.3
3
3.3
```

```
import collections

a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}

cm = collections.ChainMap(a, b)
print('Before: {}'.format(cm['c']))
cm['d'] = '3.3'
print('After : {}'.format(cm['c']))
print(cm.maps)
print(a)
print(b)
```

```
Before: 3
After : 3
[{'a': '1', 'c': '3', 'd': '3.3'}, {'b': '2', 'c': '33'}]
```

```
{'a': '1', 'c': '3', 'd': '3.3'}
{'b': '2', 'c': '33'}
```

When the new value is stored using m, the a mapping is updated.

ChainMap provides a convenience method for creating a new instance with one extra mapping at the front of the maps list to make it easy to avoid

This stacking behavior is what makes it convenient to use ChainMap instances as template or application contexts. Specifically, it is easy to add or

```
import collections


a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}

m1 = collections.ChainMap(a, b)
m2 = m1.new_child()

print('m1 before:', m1)
print('m2 before:', m2)

m2['c'] = '3.3'

print('m1 after:', m1)
print('m2 after:', m2)
```

```
m1 before: ChainMap({'a': '1', 'c': '3'}, {'b': '2', 'c': '33'})
m2 before: ChainMap({}, {'a': '1', 'c': '3'}, {'b': '2', 'c': '33'})
m1 after: ChainMap({'a': '1', 'c': '3'}, {'b': '2', 'c': '33'})
m2 after: ChainMap({'c': '3.3'}, {'a': '1', 'c': '3'}, {'b': '2', 'c': '33'})
```

For situations where the new context is known or built in advance, it is also possible to pass a mapping to new_child().

```
import collections


a = {'a': '1', 'c': '3'}
b = {'b': '2', 'c': '33'}
c = {'c': '333'}

m1 = collections.ChainMap(a, b)
m2 = m1.new_child(c)

print('m1["c"] = {}'.format(m1['c']))
print('m2["c"] = {}'.format(m2['c']))
print(m2)

#This is the equivalent of
m2_1 = collections.ChainMap(c, *m1.maps)
print(m2_1)
```

```
m1["c"] = 3
m2["c"] = 333
ChainMap({'c': '333'}, {'a': '1', 'c': '3'}, {'b': '2', 'c': '33'})
ChainMap({'c': '333'}, {'a': '1', 'c': '3'}, {'b': '2', 'c': '33'})
```

Counter

Counter is a *dict* subclass which helps count the hashable objects. It stores elements as dictionary keys and the counts of the objects as value. In c

For example:

```
# Tally occurrences of words in a list
from collections import Counter

cnt = Counter()
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
    cnt[word] += 1

Counter({'blue': 3, 'red': 2, 'green': 1})

# Find the ten most common words in Hamlet
import re
```

```
words = re.findall(r'\w+', open('hamlet.txt').read().lower())
Counter(words).most_common(10)
```

```
[('the', 1150),
 ('and', 983),
 ('to', 772),
 ('of', 672),
 ('i', 638),
 ('you', 556),
 ('a', 550),
 ('my', 516),
 ('in', 450),
 ('it', 419)]
```

Where as Counter can be used:

Counter() with lists

```
l = [1 ,23 , 23, 44, 4, 44, 55, 555, 44, 32, 23, 44, 56, 64, 2, 1]

lstCounter = Counter(l)
print(lstCounter)
print(lstCounter.most_common(4))
```

```
Counter({44: 4, 23: 3, 1: 2, 4: 1, 55: 1, 555: 1, 32: 1, 56: 1, 64: 1, 2: 1})
[(44, 4), (23, 3), (1, 2), (4, 1)]
```

Counter with Strings

```
sentance = "The collections module is a tresure trove of a built-in module that implements " + \
           "specialized container datatypes providing alternatives to Python’s general purpose " + \
           "built-in containers."

wordList = sentance.split(" ")
Counter(wordList).most_common(3)
```

```
[('module', 2), ('a', 2), ('built-in', 2)]
```

Counter methods

```
# find the most common words

# Methods with Counter()
c = Counter(wordList)
print(c.most_common(4))
print(c.items())
```

```
[('module', 2), ('a', 2), ('built-in', 2), ('The', 1)]
dict_items([('The', 1), ('collections', 1), ('module', 2), ('is', 1), ('a', 2), ('tresure', 1), ('trove', 1), ('of', 1), ('built-in', 2),
```

Default dict

The standard dictionary includes the method `setdefault()` for retrieving a value and establishing a default if the value does not exist. By contrast,

```
d = {"a": 1, "b": 2}
print(d)
print(d['a'])
print(d['d'])
```

```
{'a': 1, 'b': 2}
1


---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-48-f78105e82f88> in <module>()
```

```
      2 print(d)
      3 print(d['a'])
----> 4 print(d['d'])


KeyError: 'd'
```

```
from collections import defaultdict

dd  = defaultdict(object)
print(dd)
print(dd['one'])
print(dd)
dd['Two'] = 2
print(dd)
for d in dd:
    print(d)
    print(dd[d])
```

```
<class 'str'>
defaultdict(<class 'object'>, {})
<object object at 0x000002DB01A77310>
defaultdict(<class 'object'>, {'one': <object object at 0x000002DB01A77310>})
defaultdict(<class 'object'>, {'one': <object object at 0x000002DB01A77310>, 'Two': 2})
one
<object object at 0x000002DB01A77310>
Two
2
```

```
help(defaultdict)
```

```
Help on class defaultdict in module collections:

class defaultdict(builtins.dict)
 |  defaultdict(default_factory[, ...]) --> dict with default factory
 |  
 |  The default factory is called without arguments to produce
 |  a new value when a key is not present, in __getitem__ only.
 |  A defaultdict compares equal to a dict with the same items.
 |  All remaining arguments are treated the same as if they were
 |  passed to the dict constructor, including keyword arguments.
 |  
 |  Method resolution order:
 |      defaultdict
 |      builtins.dict
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __copy__(...)
 |      D.copy() -> a shallow copy of D.
 |  
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |  
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  __missing__(...)
 |      __missing__(key) # Called by __getitem__ for missing key; pseudo-code:
 |      if self.default_factory is None: raise KeyError((key,))
 |      self[key] = value = self.default_factory()
 |      return value
 |  
 |  __reduce__(...)
 |      Return state information for pickling.
 |  
 |  __repr__(self, /)
 |      Return repr(self).
 |  
 |  copy(...)
 |      D.copy() -> a shallow copy of D.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  default_factory
|      Factory for default value called by __missing__().
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.dict:
|
|  __contains__(self, key, /)
|      True if D has a key k, else False.
|
|  __delitem__(self, key, /)
|      Delete self[key].
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getitem__(...)
|      x.__getitem__(y) <==> x[y]
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(...)
|      D.__sizeof__() -> size of D in memory, in bytes
|
|  clear(...)
|      D.clear() -> None.  Remove all items from D.
|
|  fromkeys(iterable, value=None, /) from builtins.type
|      Returns a new dict with keys from iterable and values equal to value.
|
|  get(...)
|      D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.
|
|  items(...)
|      D.items() -> a set-like object providing a view on D's items
|
|  keys(...)
|      D.keys() -> a set-like object providing a view on D's keys
|
|  pop(...)
|      D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
|      If key is not found, d is returned if given, otherwise KeyError is raised
|
|  popitem(...)
|      D.popitem() -> (k, v), remove and return some (key, value) pair as a
|      2-tuple; but raise KeyError if D is empty.
|
|  setdefault(...)
|      D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D
|
|  update(...)
|      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
|      If E is present and has a .keys() method, then does:  for k in E: D[k] = E[k]
|      If E is present and lacks a .keys() method, then does:  for k, v in E: D[k] = v
|      In either case, this is followed by: for k in F:  D[k] = F[k]
|
|  values(...)
|      D.values() -> an object providing a view on D's values
|
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from builtins.dict:
 |  
 |  __hash__ = None
```

```
# Initializing with default value

dd = defaultdict(1)
print(dd)
print(dd['one'])
print(dd)
dd['Two'] = 2
print(dd)

for d in dd:
    print(d)
    print(dd[d])
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-58-845758bcfd69> in <module>()
      1 # Initializing with default value
      2 
----> 3 dd = defaultdict(1)
      4 print(dd)
      5 print(dd['one'])


TypeError: first argument must be callable or None
```

```
# Using factory function
import collections

def default_factory():
    return 'default value'

d = collections.defaultdict(default_factory, india='new delhi')
print('d:', d)
print('india =>', d['india'])
print('bar =>', d['bar'])
print(d)
```

```
d: defaultdict(<function default_factory at 0x000002DB032A4F28>, {'india': 'new delhi'})
india => new delhi
bar => default value
defaultdict(<function default_factory at 0x000002DB032A4F28>, {'india': 'new delhi', 'bar': 'default value'})
```

```
# Using factory function
import collections

def default_factory():
    return 'Bhopal'

d = collections.defaultdict(default_factory,
                            {"india": 'new delhi',
                             "karnataka":"Bangaluru"})
print('d:', d)
print('india =>', d['india'])
print('MP =>', d['MP'])
print(d)
```

```
d: defaultdict(<function default_factory at 0x000002DB030A1620>, {'india': 'new delhi', 'karnataka': 'Bangaluru'})
india => new delhi
MP => Bhopal
defaultdict(<function default_factory at 0x000002DB030A1620>, {'india': 'new delhi', 'karnataka': 'Bangaluru', 'MP': 'Bhopal'})
```

```
# Using factory function

# ---------------------------------------------------
# TODO:  How can i pass value to the default function
# ---------------------------------------------------
```

```
import collections

def default_factory():
    return 'default value'

d = collections.defaultdict(default_factory, foo='bar')
print('d:', d)
print('foo =>', d['foo'])
print('bar =>', d['bar'])
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-61-488d8cb20dbf> in <module>()
      5     return 'default value'
      6 
----> 7 d = collections.defaultdict(default_factory(), foo='bar')
      8 print('d:', d)
      9 print('foo =>', d['foo'])


TypeError: first argument must be callable or None
```

```
# Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:

from collections import defaultdict

countryList = [("India", "New Delhi"), ("Iceland", "Reykjavik"),
               ("Indonesia", "Jakarta"), ("Ireland", "Dublin"),
               ("Israel", "Jerusalem"), ("Italy", "Rome")]
d = defaultdict(list)
for country, capital in countryList:
    d[country].append(capital)

print(d.items())
```

```
# Setting the default_factory to int makes the defaultdict useful for counting
quote = 'Vande Mataram'
dd = defaultdict(int)
print(dd)
for chars in quote:
    dd[chars] += 1

print(dd.items())
print(dd['T'])
```

```
defaultdict(<class 'int'>, {})
dict_items([('V', 1), ('a', 4), ('n', 1), ('d', 1), ('e', 1), (' ', 1), ('M', 1), ('t', 1), ('r', 1), ('m', 1)])
0
```

deque — Double-Ended Queue

A double-ended queue, or deque, supports adding and removing elements from either end of the queue. The more commonly used stacks and que

```
import collections

d = collections.deque('Vande Mataram')
print('Deque:', d)
print('Length:', len(d))
print('Left end:', d[0])
print('Right end:', d[-1])

d.remove('e')
print('remove(e):', d)
```

```
Deque: deque(['V', 'a', 'n', 'd', 'e', ' ', 'M', 'a', 't', 'a', 'r', 'a', 'm'])
Length: 13
Left end: V
Right end: m
remove(e): deque(['V', 'a', 'n', 'd', ' ', 'M', 'a', 't', 'a', 'r', 'a', 'm'])
```

Adding

```
import collections

# Add to the right
d1 = collections.deque()
d1.extend('Vande')
print('extend    :', d1)

for a in " Mataram":
    d1.append(a)

d1.extend(" !!!")
print('append    :', d1)
d1.extendleft(" #!* ")
print('append    :', d1)

# Add to the left
d2 = collections.deque()
d2.extendleft(range(6))
print('extendleft:', d2)
d2.appendleft(6)
print('appendleft:', d2)
```

```
extend    : deque(['V', 'a', 'n', 'd', 'e'])
append    : deque(['V', 'a', 'n', 'd', 'e', ' ', 'M', 'a', 't', 'a', 'r', 'a', 'm', ' ', '!', '!', '!'])
append    : deque([' ', '*', '!', '#', ' ', 'V', 'a', 'n', 'd', 'e', ' ', 'M', 'a', 't', 'a', 'r', 'a', 'm', ' ', '!', '!', '!'])
extendleft: deque([5, 4, 3, 2, 1, 0])
appendleft: deque([6, 5, 4, 3, 2, 1, 0])
```

Consuming

OrderedDict

It is a dictionary subclass that remembers the order in which its contents are added.

Lets start with a normal dictionary:

```
fruitsCount = {}
fruitsCount["apple"] = 10
fruitsCount["grapes"] = 120
fruitsCount["mango"] = 200
fruitsCount["kiwi"] = 2000
fruitsCount["leeche"] = 20
print(fruitsCount)
for fruit in fruitsCount:
    print(fruit)
```

```
{'apple': 10, 'grapes': 120, 'mango': 200, 'kiwi': 2000, 'leeche': 20}
apple
grapes
mango
kiwi
leeche
```

```
# Now lets try this with OrderedDict

from collections import OrderedDict as OD

fruitsCount = OD()
fruitsCount["apple"] = 10
fruitsCount["grapes"] = 120
fruitsCount["mango"] = 200
fruitsCount["kiwi"] = 2000
fruitsCount["leeche"] = 20
print(fruitsCount)
for fruit in fruitsCount:
    print(fruit)
```

```
OrderedDict([('apple', 10), ('grapes', 120), ('mango', 200), ('kiwi', 2000), ('leeche', 20)])
apple
grapes
mango
kiwi
leeche
```

namedtuple

Named tuples helps to have meaning of each position in a tuple and allow us to code with better readability and self-documenting code. You can us

```
from collections import namedtuple

Point = namedtuple("India", ['x', 'y', "z"])  # Defining the namedtuple
p = Point(10, y=20, z = 30)  # Creating an object
print(p)
print(p.x + p.y + p.z)
p[0] + p[1]  # Accessing the values in normal way
x, y, z  = p     # Unpacking the tuple
print(x)

print(y)
```

```
India(x=10, y=20, z=30)
60
10
20
```

More Details:

- **https://docs.python.org/3/library/collections.html <https://docs.python.org/3/library/collections.html>**,
- **http://alexmarandon.com/articles/python_collections_tips/ <http://alexmarandon.com/articles/python_collections_tips/>**,
- **http://pymbook.readthedocs.io/en/latest/collections.html <http://pymbook.readthedocs.io/en/latest/collections.html>**

Practice Questions

- Write a function lensort to sort a list of strings based on length.
- Write a program to count frequency of characters in a given file. Can you use character frequency to tell whether the given file is a Python prog
- Write a program similar to 'tail'
- write a program similar to "wc"