Data Structures in Python

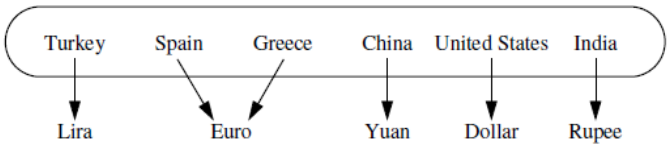
11. Maps

Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Maps and Dictionaries

- Python's **dict class** is the most significant data structure in the language.
- It represents an abstraction known as a ***dictionary*** in which unique ***keys*** are mapped to associated ***values***.
- Because of the relationship they express between keys and values, dictionaries are commonly known as ***associative arrays*** or ***maps***.
- Here, we use the term ***dictionary*** when specifically discussing Python's **dict class**, and the term ***map*** when discussing the more general notion of the abstract data type.
- As a simple example, the following figure illustrates a ***map*** from the names of ***countries*** to their associated units of ***currency***.

Maps and Dictionaries



- We note that the keys (the country names) are assumed to be unique, but the values (the currency units) are not necessarily unique. For example, we note that Spain and Greece both use the euro for currency.
- Maps use an array-like syntax for indexing, such as currency['Greece'] to access a value associated with a given key or currency['Greece'] = 'Drachma' to remap it to a new value.
- Unlike a standard array, indices for a map need not be consecutive nor even numeric.
- In this lecture we demonstrate that a map may be implemented so that a search for a key, and its associated value, can be performed very efficiently.

3

The Map ADT

- In this section, we introduce the ***map ADT***, and define its behaviors to be consistent with those of Python's built-in **dict class**.
- We begin by listing what we consider the most significant five behaviors of a ***map M*** as follows:

M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a **KeyError**. In Python, this is implemented with the special method **__getitem__.**

M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. In Python, this is implemented with the special method **__setitem__**.

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a **KeyError**. In Python, this is implemented with the special method **__delitem__**.

4

The Map ADT

len(M): Return the number of items in map M. In Python, this is implemented with the special method **__len__**.
iter(M): The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method **__iter__** , and it allows loops of the form, for k in M.

- The above five behaviors demonstrate the core functionality of a map, namely, the ability to query, add, modify, or delete a key-value pair, and the ability to report all such pairs.
- The map *M* should also support the following behaviors:

k in M: Return True if the map contains an item with key k. In Python, this is implemented with the special ***__contains__*** method.

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University 5

The Map ADT

M.get(k, d=None): Return M[k] if key k exists in the map; otherwise return default value d. This provides a form to query M[k] without risk of a KeyError.
M.setdefault(k, d): If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.
M.pop(k, d=None): Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d (or raise KeyError if parameter d is None).
M.popitem(): Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.
M.clear( ): Remove all key-value pairs from the map.

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University 6

The Map ADT

M.keys( ): Return a set-like view of all keys of M.
M.values( ): Return a set-like view of all values of M.
M.items( ): Return a set-like view of (k,v) tuples for all entries of M.
M.update(M2): Assign M[k] = v for every (k,v) pair in map M2.
M == M2: Return True if maps M and M2 have identical key-value associations.
M != M2: Return True if maps M and M2 do not have identical key-value associations.

- **Example:** The following table shows the effect of a series of operations on an initially empty map storing items with integer keys and single-character values.
  We use the literal syntax for Python's **dict class** to describe the map contents.

Science Minia University

7

The Map ADT

Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys( )	'B', 'U'	{'B': 4, 'U': 2}
M.values( )	4, 2	{'B': 4, 'U': 2}
M.items( )	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'B': 4, 'U': 2, 'A': 1}
M.popitem( )	('A': 1)	{'B': 4, 'U': 2}

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

8

Python's MutableMapping Abstract Base Class

- The Python's **collections module** provides two abstract base classes that are relevant to our current discussion: the **Mapping** and **MutableMapping** classes.
- The Mapping class includes all nonmutating methods supported by Python's **dict class**.
- The **MutableMapping** class extends that to include the mutating methods.
- These abstract base classes provide a framework to assist in creating a ***user-defined map class***.
- In particular, the **MutableMapping class** provides *concrete* implementations for all behaviors other than the first five outlined above: ***__getitem__***, ***__setitem__***, ***__delitem__***, ***__len__***, and ***__iter__***.
- As we implement the map abstraction with various data structures, as long as we provide the five core behaviors, we can inherit all other derived behaviors by simply declaring **MutableMapping** as a parent class.

9

Our MapBase Class

- The **MutableMapping abstract base class**, from Python's collections module is a valuable tool when implementing a map.
- However, in the interest of greater code reuse, we define our own **MapBase class**, which is itself a subclass of the **MutableMapping class**.
- Our **MapBase** class is defined in the following code fragment, extending the existing **MutableMapping** abstract base class so that we inherit the many useful concrete methods that class provides.
- We then define a nonpublic nested **_Item class**, whose instances are able to store both a key and value.
- It provides support for both equality tests and comparisons, both of which rely on the item's key.

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

10

Our MapBase Class

```
from collections.abc import MutableMapping
class MapBase(MutableMapping):
 """Our own abstract base class that includes a nonpublic Item class."""
 #------------------------------ nested Item class ------------------------------
 class _Item:
  """Lightweight composite to store key-value pairs as map items."""
  __slots__ = '_key' , '_value'

  def __init__ (self, k, v):
   self._key = k
   self._value = v

  def __eq__(self, other):
   return self._key == other._key    # compare items based on their keys

  def __ne__(self, other):
   return not (self._key == other._key)        # opposite of eq

  def __lt__(self, other):
   return self._key < other._key # compare items based on their keys
```


Simple Unsorted Map Implementation

- We demonstrate the use of the **MapBase class** with a very simple concrete implementation of the **map ADT**.
- The following code fragment presents an **UnsortedTableMap class** that stores ***key-value pairs*** in arbitrary order within a ***Python list***.
- An empty table is initialized as ***self._table*** within the constructor for our map.
- When a new key is entered into the map, via the ***__setitem__*** method, we create a new instance of the nested **_Item class**, which is inherited from the **MapBase class**.
- This ***list-based map implementation*** is simple, but it is not particularly efficient.
- Each of the fundamental methods, ***__getitem__***, ***__setitem__***, and ***__delitem__***, uses a for loop to scan the underlying list of items in search of a matching key.

Simple Unsorted Map Implementation

- In a best-case scenario, such a match may be found near the beginning of the list, in which case the loop terminates; in the worst case, the entire list will be examined.

```
from MapBase import MapBase
class UnsortedTableMap(MapBase):
 """Map implementation using an unordered list."""
 def __init__(self):
  """Create an empty map."""
  self._table = [ ] # list of Item's
 def __getitem__(self, k):
  """Return value associated with key k (raise KeyError if not found)."""
  for item in self. table:
   if k == item._key:
    return item._value
   raise KeyError('Key Error: ' + repr(k))
 def __setitem__(self, k, v):
  """Assign value v to key k, overwriting existing value if present."""
  for item in self._table:
   if k == item._key: # Found a match:
    item._value = v # reassign value
    return          # and quit
   # did not find match for key
   self._table.append(self._Item(k,v))
```

13

Simple Unsorted Map Implementation

```
 def __delitem__(self, k):
  """Remove item associated with key k (raise KeyError if not found)."""
  for j in range(len(self._table)):
   if k == self._table[j]._key: # Found a match:
    self._table.pop(j) # remove item
    return             # and quit
  raise KeyError('Key Error: ' + repr(k))
 def len(self):
  """Return number of items in the map."""
  return len(self. table)
 def __iter__(self):
  """Generate iteration of the map s keys."""
  for item in self._table:
   yield item._key
 def __contains__(self, k):
  try:
   self.__getitem__(k)            # access via getitem (ignore result)
   return True
  except KeyError:
   return False # attempt failed
```

Simple Unsorted Map Implementation

- **Example:** The following code exercises the Unsorted Map operations:

```
def print_map(M):
    keys = iter(M)      # get all keys
    for key in keys:
        print(key, ':', M.__getitem__(key), sep='', end=' ')
    print()
def main():
    M = UnsortedTableMap()
    print('Initial map length:', len(M))
    M.__setitem__('K', 2)
    M.__setitem__('B', 4)
    M.__setitem__('U', 2)
    M.__setitem__('V', 8)
    print('Map length after adding 4 items:', len(M))
    print('Map contents:')
    print_map(M)
    M.__setitem__('K', 9)
    print('Map contents after resetting value of key K:')
    print_map(M)
    print('item with key K exists:', M.__contains__('K'))
```


Simple Unsorted Map Implementation

```
    M.__delitem__('B')
    print('item with key B exists:', M.__contains__('B'))
    print('Map contents after deleting item with key B:')
    print_map(M)
main()
```

Output

```
Initial map length: 0
Map length after adding 4 items: 4
Map contents:
K:2 B:4 U:2 V:8
Map contents after resetting value of key K:
K:9 B:4 U:2 V:8
item with key K exists: True
item with key B exists: False
Map contents after deleting item with key B:
K:9 U:2 V:8
```

- **Example:** Find the frequencies of distinct words in a given sentence

```
freqMap = UnsortedTableMap()
S = input('Enter a sentence: ')
words = S.split()
for word in words:
    if freqMap.__contains__(word):
        freq = freqMap.__getitem__(word) + 1
        freqMap.__setitem__(word, freq)
    else:
        freqMap.__setitem__(word, 1)
keys = iter(freqMap)      # get all keys
for key in keys:
    print(key, ':', freqMap.__getitem__(key))
```

Sample Run

```
Enter a sentence: the cat sat on the mat
the : 2
cat : 1
sat : 1
on : 1
mat : 1
```

Sorted Maps

- The traditional **map ADT** allows a user to look up the value associated with a given key, but the search for that key is a form known as an ***exact search***.
- However, the **map ADT** does not provide any way to get a list of all values ordered by the key, or to search for values near to a particular key.
- In this section, we introduce an extension known as the **sorted map ADT** that includes all behaviors of the standard map, plus the following:

M.find_min(): Return the (key,value) pair with minimum key (or None, if map is empty).
M.find_max(): Return the (key,value) pair with maximum key (or None, if map is empty).
M.find_lt(k): Return the (key,value) pair with the greatest key that is strictly less than k (or None, if no such item exists).

Sorted Maps

M.find_le(k): Return the (key,value) pair with the greatest key that is less than or equal to k (or None, if no such item exists).
M.find_gt(k): Return the (key,value) pair with the least key that is strictly greater than k (or None, if no such item exists).
M.find_ge(k): Return the (key,value) pair with the least key that is greater than or equal to k (or None, if no such item).
M.find_range(start, stop): Iterate all (key,value) pairs with start <= key < stop. If start is None, iteration begins with minimum key; if stop is None, iteration concludes with maximum key.
iter(M): Iterate all keys of the map according to their natural order, from smallest to largest.
reversed(M): Iterate all keys of the map in reverse order; in Python, this is implemented with the ***__reversed__*** method.

Sorted Maps

❖ Sorted Search Tables

- Several data structures can efficiently support the sorted map ADT.
- In this section, we explore a simple implementation of a sorted map, referred to as ***sorted search table***.
- We store the map's items in an array-based sequence A so that they are in increasing order of their keys, assuming the keys have a naturally defined order.
- The figure shows a realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

- The primary advantage of this representation, is that it allows us to use the ***binary search*** algorithm for a variety of efficient operations.

19

Sorted Maps

➢ *Binary Search and Inexact Searches*

- The binary search algorithm is a means for detecting whether a given target is stored within a sorted sequence.
- A binary search function could be implemented to return ***True*** or ***False*** to designate whether the desired target was found.
- While such an approach could be used to implement the ***__contains__*** method of the map ADT, we can adapt the binary search algorithm to provide far more useful information when performing forms of ***inexact search*** in support of the sorted map ADT.
- The important realization is that while performing a binary search, we can determine the index at or near where a target might be found.
  - During a successful search, the standard implementation determines the precise index at which the target is found.

20

Sorted Maps

- During an unsuccessful search, although the target is not found, the algorithm will effectively determine *a pair of indices designating elements of the collection that are just less than or just greater than the missing target (i.e., the missing target lies in the gap between them).*

➢ ***Implementation***

- The following code shows a complete implementation of a class, **SortedTableMap**, that supports the sorted map ADT.
- The most notable feature of our design is the inclusion of a ***_find_index*** utility function.
  - This method uses the binary search algorithm, but by convention returns the *index* of the leftmost item in the search interval having key greater than or equal to *k*.
  - Therefore, if the key is present, it will return the index of the item having that key. (Recall that keys are unique in a map.)

Sorted Maps

- When the key is missing, the function returns the index of the item in the search interval that is just beyond where the key would have been located. As a technicality, the method returns index high+1 to indicate that no items of the interval had a key greater than *k*.
- This utility method is used when implementing the traditional map operations and the new sorted map operations.
- Each of the ***__getitem__***, ***__setitem__***, and ***__delitem__*** methods begins with a call to ***_find_index*** to determine a candidate index at which a matching key might be found.
  - For ***__getitem__***, we check whether that is a valid index containing the target to determine the result.
  - For ***__setitem__***, (which replaces the value of an existing item, if one with key *k* is found, otherwise inserts a new item into the map), the index returned by ***_find_index*** will be the index of the match, if one exists, or otherwise the exact index at which the new item should be inserted.

Sorted Maps

- For ***__delitem__***, we again use ***_find_index*** to determine the location of the item to be popped, if any.
- ***_find_index*** utility is also used in implementing the various inexact search methods.
- Each of the methods ***find_lt***, ***find_le***, ***find_gt***, and ***find_ge***, begins with a call to ***_find_index*** utility, which locates the first index at which there is an element with key ≥ *k*, if any.
  - This is precisely what we want for ***find_ge***, if valid, and just beyond the index we want for ***find_lt***.
  - For ***find_gt*** and ***find_le*** we need some extra case analysis to distinguish whether the indicated index has a key equal to *k*. For example, if the indicated item has a matching key, ***find_gt*** increments the index before continuing with the process.
- In all cases, we must properly handle boundary cases, reporting None when unable to find a key with the desired property.

Sorted Maps

- In implementing ***find_range***, the ***_find_index*** utility is used to locate the first item with key ≥ start (assuming start is not None).
  - With that knowledge, we use a while loop to sequentially report items until reaching one that has a key ≥ the stopping value (or until reaching the end of the table).
  - Note that the while loop may trivially iterate zero items if the first key that is ≥ start also happens to be ≥ to stop. This represents an empty range in the map.
- An implementation of a **SortedTableMap** class:

```
from MapBase import MapBase
class SortedTableMap(MapBase):
 """Map implementation using a sorted table."""
 #------------------------------ nonpublic behaviors ------------------------------
 def _find_index(self, k, low, high):
  """Return index of the leftmost item with key greater than or equal to k.
   Return high + 1 if no such item qualifies.
   That is, j will be returned such that:
     all items of slice table[low:j] have key < k
     all items of slice table[j:high+1] have key >= k
  """
```

Sorted Maps

```
  if high < low:
   return high + 1 # no element qualifies
  else:
   mid = (low + high) // 2
   if k == self._table[mid]._key:
    return mid # found exact match
   elif k < self._table[mid]._key:
    return self._find_index(k, low, mid - 1) # Note: may return mid
   else:
    return self._find_index(k, mid + 1, high) # answer is right of mid
 #------------------------------ public behaviors ------------------------------
 def __init__(self):
  """Create an empty map."""
  self._table = [ ]

 def __len__(self):
  """Return number of items in the map."""
  return len(self._table)

 def __getitem__(self, k):
  """Return value associated with key k (raise KeyError if not found)."""
  j = self._find_index(k, 0, len(self._table) - 1)
  if j == len(self._table) or self._table[j]._key != k:
   raise KeyError('Key Error:' + repr(k))
  return self._table[j]._value
```

25

Sorted Maps

```
 def __setitem__(self, k, v):
  """Assign value v to key k, overwriting existing value if present."""
  j = self._find_index(k, 0, len(self._table) - 1)
  if j < len(self._table) and self._table[j]._key == k:
   self._table[j]._value = v # reassign value
  else:
   self._table.insert(j, self._Item(k,v)) # adds new item

 def __delitem__(self, k):
  """Remove item associated with key k (raise KeyError if not found)."""
  j = self._find_index(k, 0, len(self._table) - 1)
  if j == len(self._table) or self._table[j]._key != k:
   raise KeyError('Key Error:' + repr(k))
  self._table.pop(j) # delete item

 def __iter__(self):
  """Generate keys of the map ordered from minimum to maximum."""
  for item in self._table:
   yield item._key

 def __reversed__(self):
  """Generate keys of the map ordered from maximum to minimum."""
  for item in reversed(self._table):
   yield item._key
```

26

Sorted Maps

```
def find_min(self):
 """Return (key,value) pair with minimum key (or None if empty)."""
 if len(self._table) > 0:
  return (self._table[0]._key, self._table[0]._value)
 else:
  return None

def find_max(self):
 """Return (key,value) pair with maximum key (or None if empty)."""
 if len(self._table) > 0:
  return (self._table[-1]._key, self._table[-1]._value)
 else:
  return None

def find_ge(self, k):
 """Return (key,value) pair with least key greater than or equal to k."""
 j = self._find_index(k, 0, len(self._table) - 1) # j's key >= k
 if j < len(self._table):
  return (self._table[j]. key, self._table[j]._value)
 else:
  return None
```


Sorted Maps

```
def find_lt(self, k):
 """Return (key,value) pair with greatest key strictly less than k."""
 j = self._find_index(k, 0, len(self._table) - 1) # j's key >= k
 if j > 0:
  return (self._table[j-1]._key, self._table[j-1]._value) # Note use of j-1
 else:
  return None

def find_gt(self, k):
 """Return (key,value) pair with least key strictly greater than k."""
 j = self._find_index(k, 0, len(self._table) - 1) # j's key >= k
 if j < len(self._table) and self._table[j]._key == k:
  j += 1 # advanced past match
  if j < len(self._table):
   return (self._table[j]._key, self._table[j]._value)
  else:
   return None

def find_range(self, start, stop):
 """Iterate all (key,value) pairs such that start <= key < stop.
 If start is None, iteration begins with minimum key of map.
 If stop is None, iteration continues through the maximum key of map.
 """
```

Sorted Maps

```
if start is None:
  j = 0
 else:
  j = self._find_index(start, 0, len(self._table)-1) # find first result
  while j < len(self._table) and (stop is None or self._table[j]._key < stop):
   yield (self._table[j]._key, self._table[j]._value)
   j += 1

def find_le(self, k):
 """Return (key,value) pair with greatest key less than or equal to k."""
 j = self._find_index(k, 0, len(self._table) - 1) # j's key >= k
 if j > 0:
  if self._table[j]._key != k:
   j-=1
  return (self._table[j]._key, self._table[j]._value) # Note use of j-1
 else:
  return None
```


Sorted Maps

- **Example :** The following code exercises the Sorted Map operations:

```
def print_map(M):
   keys = iter(M)      # get all keys
   for key in keys:
      print(key, ':', M.__getitem__(key), sep='', end=' ')
   print()
def reverse_print_map(M):
   rKeys = M.__reversed__()      # get all keys in reverse order
   for key in rKeys:
      print(key, ':', M.__getitem__(key), sep='', end=' ')
   print()
def main():
   M = SortedTableMap()
   M.__setitem__('K', 2)
   M.__setitem__('B', 14)
   M.__setitem__('U', 2)
   M.__setitem__('V', 8)
   print('Map length after adding 4 items:', len(M))
   print('Map contents:', end=' ')
   print_map(M)
```

Sorted Maps

```
    M.__setitem__('K', 9)
    print('Map contents after resetting value of key K:')
    print_map(M)
    M.__setitem__('F', 10)
    print('Map contents after adding an item with key F:', end=' ')
    print_map(M)
    print('Minimum  element:', M.find_min())
    print('Maximum  element:', M.find_max())
    print('Element >= F:', M.find_ge('F'))
    print('Element >= L:', M.find_ge('L'))
    print('Element > F:', M.find_gt('F'))
    print('Element < F:', M.find_lt('F'))
    print('Element <= F:', M.find_le('F'))
    print('Element <= E:', M.find_le('E'))
    range = M.find_range('F', 'U')
    print('All pairs with F <= key < U:' , end=' ')
    for elem in range:
       print(elem, end=' ')
    print()
    print('Reversed map contents:', end=' ')
    reverse_print_map(M)
main()
```


Sorted Maps

Output

```
Map length after adding 4 items: 4
Map contents: B:14 K:2 U:2 V:8
Map contents after resetting value of key K:
B:14 K:9 U:2 V:8
Map contents after adding an item with key F:
B:14 F:10 K:9 U:2 V:8
Minimum  element: ('B', 14)
Maximum  element: ('V', 8)
Element >= F: ('F', 10)
Element >= L: ('U', 2)
Element > F: ('K', 9)
Element < F: ('B', 14)
Element <= F: ('F', 10)
Element <= E: ('B', 14)
All pairs with F <= key < U: ('F', 10) ('K', 9)
Reversed map contents:
V:8 U:2 K:9 F:10 B:14
```

Sorted Maps

- **Example:** Find the frequencies of distinct words in a given sentence, using **SortedTableMap**:

```
freqMap = SortedTableMap()
S = input('Enter a sentence: ')
words = S.split()
for word in words:
  if freqMap.__contains__(word):
     freq = freqMap.__getitem__(word) + 1
     freqMap.__setitem__(word, freq)
   else:
     freqMap.__setitem__(word, 1)
keys = iter(freqMap)      # get all keys
for key in keys:
   print(key, ':', freqMap.__getitem__(key))
```

Output

```
Enter a sentence: the cat sat on the mat
cat : 1
mat : 1
on : 1
sat : 1
the : 2
```

- As can be seen the words and their frequencies are displayed in alphabetical order.