9: Dictionaries & Sets

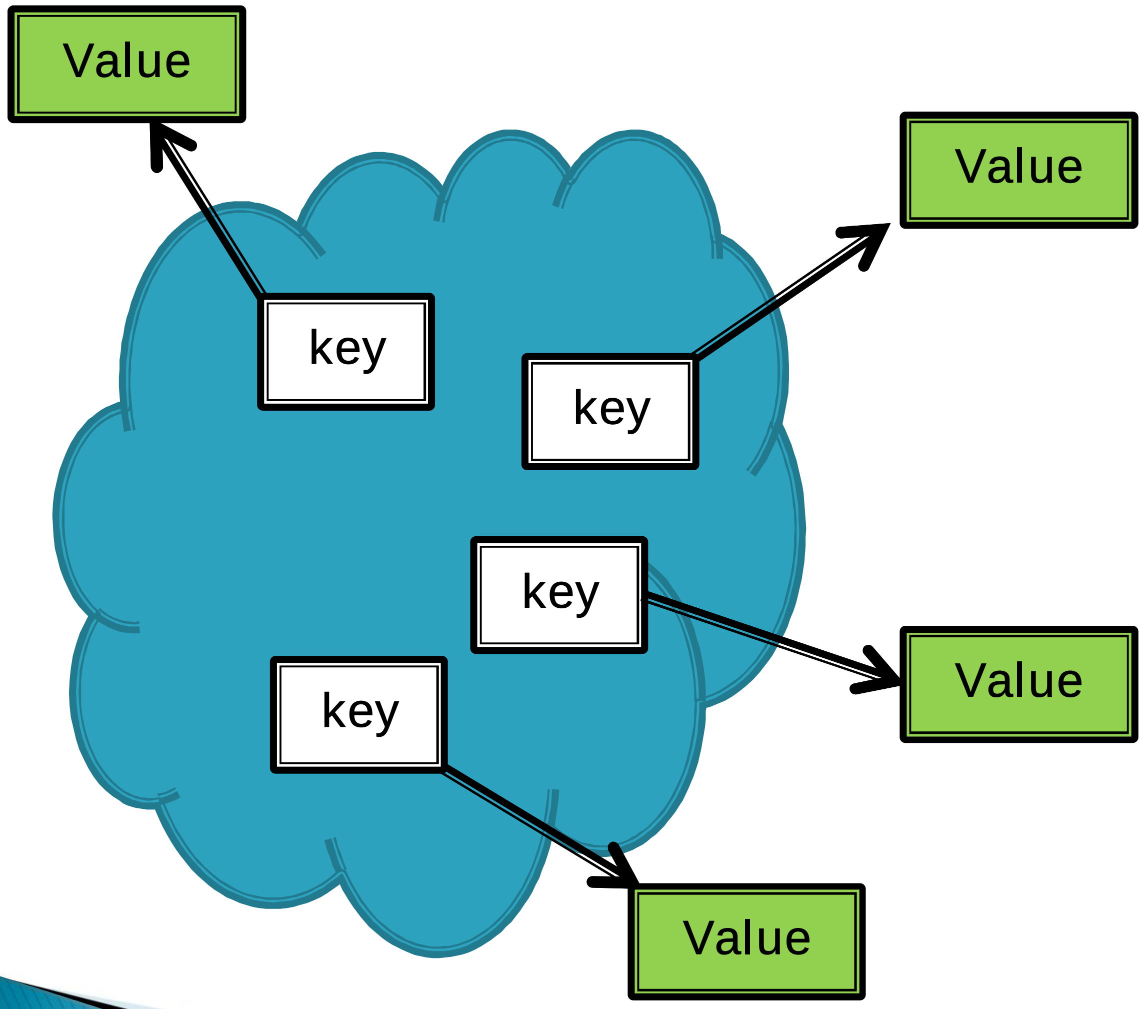
What is a dictionary?

- Dictionary in Python is a *mutable* data structure which consists of key-value pairs.
- Unlike lists and tuples, dictionary is NOT a sequence. Hence it is not ordered.
- Similar to "hash maps" or "associative arrays" in other programming languages

What is a dictionary? (cont.)

- In dictionary, **keys** are used as references to **values**.
- For each key in dictionary, a single value can be found.
- Dictionary is *indexed by keys*, not by index numbers! To find a value in dictionary, you need to know the key.

Keys and values



Keys in dictionary

- Keys in dictionary are:
  - **Unique –** two keys with equal value cannot exist in the same dictionary
  - **Immutable objects** – hence e.g. a list can not be used as a key; a tuple can be used as a key, if all of the items in tuple are also immutable (for example strings, integers or other tuples).

Values in dictionary

- Values in dictionary can consist of any objects in Python, including e.g. strings, numbers, lists, tuples, other dictionaries...

- Value types can also be mixed; each value (and key) in dictionary can be of different type.

Dictionary semantically

- Again, semantically the keys should share a common type in a dictionary
- Same applies to values

Defining a dictionary

- An empty dictionary can be constructed by using the curly brackets:

```
myDict = {}
```

Defining a dictionary (cont.)

- Alternatively, it is possible to give key-value pairs (separated with a colon) in the definition:

```
myDict = {"num1" : 100, "num2" : 250 }
```

...defines a new dictionary with two items in it.

Assigning values to dictionary

- Assigning values to dictionary is done by using the key as a reference:

```
dictionary[key] = value
```

- For example

```
myDict["num1"] = 300
```

Assigning values (cont.)

- If the key exists in the dictionary, the value is changed.

```
d = {1 : "hello", 2: "hey" }
d[1] = "hi" # replaces the value in key 1
```

- If the key doesn't exist, a **new item is added**:

```
d[3] = "ho!"
print d # outputs {1 : 'hi', 2: 'hey', 3: 'ho!'}
```

Retrieving values

- Values can be retrieved by using the key as a reference:

```
d = { 1 : "first", 2 : "second", 4 : "fourth" }
print d[1] #outputs first
print d[4] # outputs fourth
```

- If the key is not found in dictionary, an error is thrown.

Retrieving values (cont.)

- What does the following print?

```
d = { 1 : "first", "1" : "second", '1' : "fourth" }
print d[1]
print d["1"]
```

Checking if a key exists

- Again, to check if a key exists in dictionary, the **in** operator can be used:

```
d = { "name" : "James Python", "age" : 27 }
print "name" in d # outputs True
print "age" in d # outputs True
print "height" in d # outputs False
```

Number of items in dictionary

- The **len** function also works for dictionaries. It returns the number of items (key-value pairs) in dictionary.

```
test = { 1 : 10, 2: 20, 3: 30, 4: 45 }
print len (test)  # outputs 4
```

Deleting items

- An item can be deleted by using the **del** statement:

```
del dictionary[key]
```

- For example:

```
myDict = { 1 : "apple", 2 : "orange", 3 "banana" }
del myDict[2]
print myDict # outputs { 1 : 'apple', 3 : 'banana' }
```

The del statement

- Note, that the `del` statement also works with lists:

```
lst = range(1,5)
del(lst[0])
print lst # Output [2, 3, 4]
```

Retrieving all keys in dictionary

- To retrieve a list of all keys in a dictionary, we can use the **keys()** method:

```
d = {1 : "p", 2 : "y", 3 : "t", 6 : "h", 4 : "o"}
myKeys = d.keys()
print myKeys # outputs [1, 2, 3, 6, 4]
```

- Note, that since a dictionary is not ordered, the list of keys **is not sorted in any way**.

...and all values in dictionary

- Or, to retrieve a list with all values in a dictionary, we can use the **values()** method:

```
d = {1 : "p", 2 : "y", 3 : "t", 6 : "h", 4 : "o"}
myValues= d.values()
print myValues # outputs ['p', 'y', 't', 'h', 'o']
```

Iterating though a dictionary

- The **for** statement can be used to iterate through all **keys** in a dictionary:

```
d = {"A": 55, "T": 11, "G": 14, "C": 20}
for key in d:
   print key, ":", d[key], "%"
```

Hence..

- These two are hence equivalent:

```python
d = {"A": 55, "T": 11, "G": 14, "C": 20}

# version 1
for key in d:
   print key, ":", d[key], "%"

# version 2
for key in d.keys():
   print key, ":", d[key], "%"
```

Iterating…(cont.)

- The `iteritems()` method returns a **generator** for retreiving all items in dictionary as tuples:

```
d = {1: 10, 2: 20, 3: 30}
for item in d.iteritems():
    print item
```

…will output

```
(1, 10)
(2, 20)
(3, 30)
```

Iterating... (cont.)

- To iterate through keys *and* values, we can use the **iteritems()** method and for loop and assign the key and value into two variables:

```
d = {"A": 55, "T": 11, "G": 14, "C": 20}
for key, value in d.iteritems():
   print key, value
```

Example 1

- Find the name of the oldest person in a dictionary where key represents the name and value the age:

```
def getOldest(d):
    oldest = 0
    oldestPerson = ""
    for name, age in d.iteritems():
        if age > oldest:
            oldest = age
            oldestPerson = name
    return oldestPerson
```

Example 2

- Query the user for a string and save the amount of each letter in a dictionary. Finally output all occurrences.

```
s = raw_input("Give a string: ")
occurrences = {} # empty dictionary
for c in s:
   # key is unique, no need to worry about duplicates
   occurrences[c] = s.count(c)

# output
for char, amount in occurrences.iteritems():
   print char,"appeared",amount,"times."
```

Example 3

- Function that gets a dictionary and the value as a parameter, and finds and returns a key that points to that value:

```python
def getKey(dict, value):
    for key in dict:
        if dict[key] == value:
            return key
    return None # return an "empty" value
```

Sets

- Set is an **unordered** collection of **unique** items
- Hence, it shares some properties of dictionaries and lists
- Sets are mutable. Python also has a **frozenset** for immutable sets.

Why sets?

- Used to store items when no duplicates are allowed
- Sets also support basic mathematic operations for sets, such as unions or intersections

Set in Python

- To create a set, use the **set** function
- The function gets an iterable sequence (such as a list / tuple / string etc.) as an argument

Example

- Create a set from a list

```
lst = [1,2,3,4,5]
mySet = set(lst)
```

- ...or from a string:

```
s = "abcde"
mySecondSet = set(s)
```

Set contains unique items

- All items in set are unique, no duplicates are allowed:

```
myList = [1,1,2,2,3,3]
mySet = set(myList)
print mySet # outputs ([1, 2, 3])
```

Set contains unique items (2)

- Hence, set is an excellent tool for removing duplicates in a sequence:

```
def removeDuplicates(lst):
    s = set(lst) # Convert to set
    return list(s) # convert back to list

l = [1,2,3,1,2,4]
print removeDuplicates(l) # [1, 2, 3, 4]
```

Set is a collection of items

- Set, as an entity, must be seen as a holistic collection of items

- Hence, individual items in a set **cannot be assigned or referenced**

- **Remember:** the set is not ordered, there is no first or last item in a set.

Set operations

- To find out if an item is included in the set, use the **in** operator

```
s = set(range(1,10))
print 1 in s # True
print 10 in s # False, items 1…9 only
```

- Again, the number of items can be returned with **len** function:

```
print len(s) # Outputs 9
```

Set operations (2)

- The **union** method joins the items from set(s) given as parameter into current set

```
set1 = set([1,2,3])
set2 = set(range(4,6))
set1 = set1.union(set2)
print set1 # ([1,2,3,4,5])
```

Set operations (3)

- The **intersection** method returns common items from given sets

```
s1 = set([1,2,3,4])
s2 = set([3,4,5,6])
s3 = s1.intersection(s2)
print s3 # ([3, 4])
```

Set operations (4)

- The **difference** method returns the items that are present in current set, but not in the set given as argument

```
s1 = set([1,2,3,4])
s2 = set([3,4,5,6])
s3 = s1.difference(s2)
print s3 # ([1, 2])
```

Other useful set operations

- See Python documentation for details

Operation	Explanation
`s1.issubset(s2)`	Returns **true**, if all items in s1 can be found in s2
`s1.issuperset(s2)`	Returns **true**, if all items in s2 can be found in s1
`s1.isdisjoint(s2)`	Returns **true**, if the sets share no common items. i.e. if `len(s1.intersection(s2)) == 0`

Iterating through set

- Again, a set can be iterated by using a for loop:

```
s = set([1,2,3,4])
for item in s:
   print s
```