Data Structures & Algorithms

Week 11

Sets and Dictionaries

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 10 (1/2)

- The number of sub-trees of a given node is called its degree. A tree consisting of only one node has a degree of 0. This one tree node is also considered as a tree by all standards.
- A node with a degree of 0 is called a leaf node.
- A ***binary tree*** is an ordered tree with the following properties: Every node has at most two children, each child node is labeled as being either a ***left child*** or a ***right child,*** and a left child precedes a right child in the order of children of a node.
- A node's depth corresponds to the level it occupies; further the depth is also the number of edges to the node from the root.
- A full binary tree is a binary tree in which each interior node contains two children.

Flashback from Lesson 10 (2/2)

- We define a balanced tree as one where "the heights of any node's left and right subtrees differ by at most one.
- Three common algorithms used for traversal: preorder and postorder traversal, breadth-first traversal, and Inorder traversal.
- A max-heap has the property, known as the heap order property, that for each non-leaf node V , the value in V is greater than the value of its two children; the min-heap has the opposite property. For each non-leaf node V , the value in V is smaller than the value of its two children.
- Other trees include BST, AVL, red black, B, and 2-3.

Content

- Dictionaries
- Sets

Dictionaries

1.1 Introduction

- Have you observed a young child between the age of about 3 and 6 or 7? What is the thing you notice the most about them? Apart from the obvious innocence of life that is…
- They are a curious lot! They are just beginning to see the world, and they want to know what everything is. They keep asking questions about everything….what is that?, why is the wheel spinning?, what do you call that (while pointing)?, who is that coming toward us?, what is your name?, …..the list is endless.
- As an adult (depending on your attitude) these questions can wear you down….literally. However, they innately expect that you have the answer to every question…regardless of whether you are a parent, uncle, aunt, teacher, grandparent, and so on; in short once they innately identify you as an older person they just know you know "everything".

1.1 Introduction

- Fast forward. They realize as they grow up that you don't know "everything". Did this happen to you? Well it happened to everyone. Honest adults will eventually tell you they don't have an answer to this or that; you realize this when they become evasive or tell you a story that doesn't make sense…yes, even at that age you can tell (ha ha).
- And so life continues, and in a few years you are the adult being asked questions…you are the parent, aunt, uncle, cousin, and so on…. in short, the one answering the questions. Life does come full circle.
- Sophophobia. Do you know the meaning of this word? I am sure with perhaps some limited knowledge of the English language you have seen the word 'phobia', and guessed this has something to do with fear. Well, that's an intelligent guess.

1.1 Introduction

- Nonetheless, supposing you are curious about what the word actually means? You have found it in a research paper on a topic of interest to you. Or perhaps it has popped up in some compulsory literature review you have been asked to summarize as part of a class assignment (where you have no option but to find out…).
- You will turn to the dictionary…the book that knows "everything" (well, apart from Google ☺). How does a dictionary operate? All words are arranged (sorted) in alphabetical order, making it easy for you to look for a particular word.
- When you are looking for (we use the phrase 'look up a word') a particular word we open a random page and move either forward or backward since everything is arranged alphabetically, until we find the word. Next to the word we shall then find an explanation of the type of word it is (noun, adjective, and so on), what it means, an example of how to use it in a sentence, synonyms, and so on.

1.1 Introduction

- This is the case even with online dictionaries; the only difference is that the search is done for you, and you only get to see the results.
- In most dictionaries there is an index, where it is even easier to find a word by simply checking there and it will show on which page to find the word. We can look up a word quickly in the case of voluminous dictionaries and encyclopedias.
- Oh, lest I forget to give the meaning, sophophobia is "the (uncontrollable/morbid) fear of learning" (*Sophophobia Definition & Meaning | YourDictionary*, n.d.).
- In Python we have a data structure called the dictionary that acts much like the dictionary we use in everyday life. As we shall see in this part, the dictionary is used to look up meanings (what we call values in Python) by accessing the word (which is what we call the key in Python). Bearing this analogy in mind will help you to understand how the dictionary in Python works and all operations that can be performed on it.

1.2 Definition

- Dictionaries were briefly mentioned in lesson 2 of this course (see slide 33) where it was stated that "the bag ADT collection can also be implemented using a list or dictionary data structure depending on the situation".
- (Dierbach, 2015) introduces the concept of the dictionary as an associative data structure: "Elements of indexed linear data structures, such as lists, are ordered—the first element (at index 0), second element (at index 1), and so forth. In contrast, the elements of an associative data structure are unordered, instead accessed by an associated key value. In Python, an associative data structure is provided by the *dictionary type*."
- However, from Python 3.7 (released 2018) dictionaries are now ordered (in Python 3.6 and before they were unordered). What this means is that "When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change. Unordered means that the items does not have a defined order, you cannot refer to an item by using an index." We also further note that dictionaries are changeable (mutable), meaning items can be changed, added or removed after the dictionary has been created. Lastly dictionaries do not allow duplicates; that is to say, no two values can have the same key. (W3schools, 2018).

Table 1

Dictionary methods

Method	Description
dic.clear()	Remove all the elements from the dictionary
dict.copy()	Returns a copy of the dictionary
dict.get(key, default = “None”)	Returns the value of specified key
dict.items()	Returns a list containing a tuple for each key value pair
dict.keys()	Returns a list containing dictionary’s keys
dict.update(dict2)	Updates dictionary with specified key-value pairs
dict.values()	Returns a list of all the values of dictionary
pop()	Remove the element with specified key
popItem()	Removes the last inserted key-value pair
dict.setdefault(key,default= “None”)	set the key to the default value if the key is not specified in the dictionary
dict.has_key(key)	returns true if the dictionary contains the specified key.
dict.get(key, default = “None”)	used to get the value specified for the passed key.

(From Bansal, 2018)

Table 2

More dictionary operations

Operation	Results
`dict()`	Creates a new, empty dictionary

`dict(s)`	Creates a new dictionary with key values and their associated values from sequence `s`, for example,
	`fruit_prices = dict(fruit_data)`
	where `fruit_data` is (possibly read from a file):
	`[['apples', .66],…,['bananas', .49]]`

`len(d)`	Length (num of key/value pairs) of dictionary `d`.
`d[key] = value`	Sets the associated value for `key` to `value`, used to either add a new key/value pair, or replace the value of an existing key/value pair.
`del d[key]`	Remove key and associated value from dictionary `d`.
`key in d`	`True` if key value `key` exists in dictionary `d`, otherwise returns `False`.

(From Dierbach, 2015)

1.3 Operations

- Table 1 and table 2 describe the methods available to dictionaries in Python.
- The operations that can be performed on dictionaries relate to their properties, namely:
    - Mutability
    - Associative nature
    - Keys
    - Values
    - No duplication

1.3.1 Defining the dictionary

- Just like with the dictionary we are all familiar with, you define a dictionary using pairs of key:values. As earlier described you can look at the key as a word in the dictionary and value as the meaning of that word. Again, just like the ordinary dictionary words do not appear twice (that is to say, the meaning of a word is only found where the word is; for example the word 'cow' will only appear once with its meaning in the dictionary).
- Thus the dictionary is declared/defined using the syntax of curly brackets with all the key:value pairs (with key and value separated with a colon) within the curly brackets; further key:value pairs are separated using commas. Also note that dictionary keys are case sensitive.
- d={ <key>: <value>, <key>: <value>, . . . <key>: <value> } (Sturtz, n.d.)
- Where d is the name of the dictionary, followed by the described syntax.
- Let us apply this to a few examples.

1.3.1 Defining the dictionary

- daily_temps = {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8, 'thur': 73.2, 'fri': 75.6, 'sat': 74.0}
- The above example is provided by Dierbarch (2015). This is a dictionary of daily average temperatures over different days of the week; for example, the average temperature was 68.8 on Sunday, and 71.8 on Wednesday. Thus the days of the week are the keys – sun, mon, tue, wed, thur, fri, sat; while the temperatures are the values – 68.8, 70,2, 67.2, 71.8, 73.2, 75.6, 74.0. this is how the key:value pairs are arranged in the dictionary.
- The values of the temperatures can also be organized in a list; however, it won't be as descriptive as the dictionary. Further since the list is an ordered collection, we can always access the items of this list using the index as we have done in previous lessons in this course. More of this later.

1.3.1 Defining the dictionary

- Let us define a couple more dictionaries:
- dictofIntegers = { 1: data, 2: structures, 3: and, 4: algorithms} # a dictionary of integers
- Dictofmixedkeys = {'course': 'data structures', 3: [5, 6]} #a dictionary of mixed keys
- Based on the above definitions we see that a dictionary can consist of different data types of either keys, values, or both. In dictofintegers we use integers as keys, and strings as values. In the dictofmixedkeys we use both a string ('course') as well as an integer (3) as keys, while using a string ('data structures') and a list (5,6) as the respective values. How do we access the values in the list? We shall see this in the next section.
- Thus we can have a dictionary of lists, a dictionary of strings, a dictionary of integers, or a mixed dictionary; these are all accepted in Python. Let us mix them all up in one final dictionary
- Dictmixallup = {1: "yes", 'check': 1, 2: [7,8, 5], 'test': ['a', 'Python', 'mixed list', 10, 20, 30]}

1.3.1 Defining the dictionary

- We can also define an empty dictionary:
- Emptydict = {} #this creates an empty dictionary
- We can also define a dictionary using the dict() construct as shown in table 2:
- myDict = dict({1: ‘A', 2: ‘good', 3: ‘example’})
- flavorsDict = dict({‘good’: ‘vanilla’, ‘yummy’: ‘strawberry’, ‘awesome’: ‘chocolate’}) #note the brackets
- Lastly we can create a dictionary with each item as a pair:
- myspecialDict = dict([(1, ‘super'), (2, ‘boom')])
- The output of print(myspecialDict) will be:
- {1: ‘super', 2: ‘boom'}

1.3.2 Accessing dictionaries

- After creating the dictionary we need to know how to access it. There are two cases here: an ordinary dictionary and a nested dictionary. However, in both cases we can't access the items using an index since this is neither a list, and it is not ordered. Let us examine both cases:-
- Ordinary dictionary: we access the dictionary items using the keys rather than the index position. Let's demonstrate this using two examples of the dictionaries we've already created.
- dictofIntegers = { 1: data, 2: structures, 3: and, 4: algorithms} # a dictionary of integers
- Suppose we wish to access the value of the key 1; the Python command for this will be:
- dictofIntegers [1]; we can print this output by adding the print command: print(dictofIntegers [1]); we use the same syntax to print out any other value for a key.

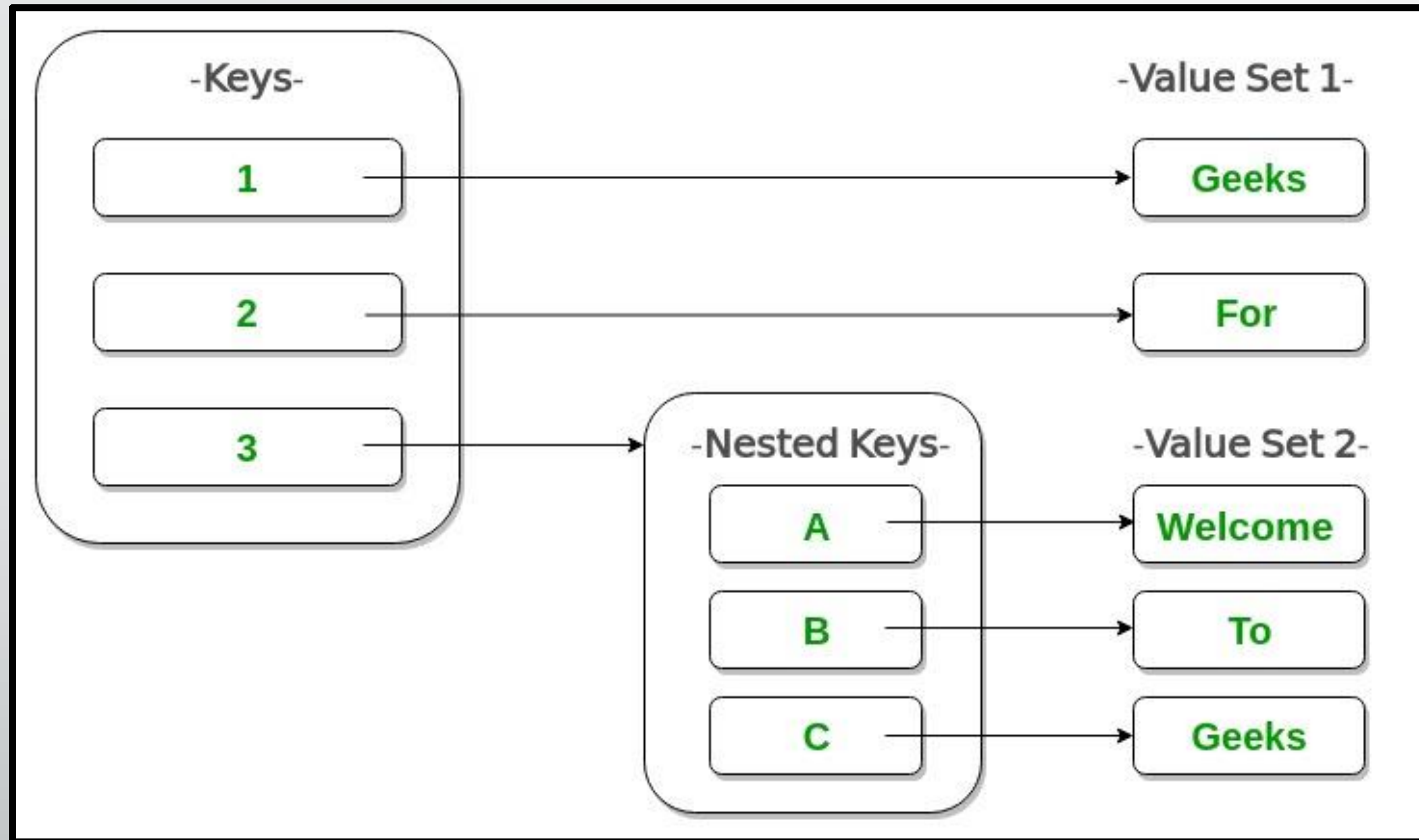
1.3.2 Accessing dictionaries

- Dictofmixedkeys = {‘course’: ‘data structures’, 3: [5, 6]}; even for dictionaries of mixed keys the same approach is used; for example, Dictofmixedkeys [‘course’] will print out ‘data structures’.
- However, is there a way of accessing the individual items of key item 3? The answer to this is, yes there is! This is the second way of accessing dictionaries, and these types of dictionaries are called nested dictionaries.
- A nested dictionary is one whereby there are keys referring to other keys, which in turn refer to values. We access list items in the same way we access nested values/keys. Figure 1 illustrates nested dictionaries.
- The dictionary in figure 1 is created as follows:

  Dict = {1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks’}}
- At this point let us also create a dictionary containing a list for access demonstration:
- myListdict = { 2: ‘this one’, 4: [5,6,7,8], ‘demo’: [10, 20, 30]}

Figure 1

Nested dictionary



(From Bansal, 2018)

1.3.2 Accessing dictionaries

- Accessing list items:
- To access a list item in a dictionary we combine our knowledge of what we have learnt in this lesson with what we learnt in lesson 9 (lists) of this course. Let us demonstrate this using the myListdict dictionary. For ease of reference let us define it here again:
- myListdict = { 2: ‘this one’, 4: [5,6,7,8], ‘demo’: [10, 20, 30]}
- The operations will be as follows:
- myListdict [2] → ‘this one’
- myListdict [‘demo’] → [10, 20, 30]
- myListdict [‘demo’][0] → 10 # this prints out the item in index 0 position of the list
- myListdict [‘demo’][2] → 30 #this prints the item in index 2 position of the list; same as using [-1] instead of [2]

1.3.2 Accessing dictionaries

- We use a similar approach with nested dictionaries; caution should be exercised, however, not to confuse index positions with keys (for dictionaries). Let us use the Dict dictionary to demonstrate this:
- Dict = {1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks’}}
- Dict[1] → ‘Geeks’
- Dict[2] → ‘For’
- Dict[3] → {'A': 'Welcome', 'B': 'To', 'C': 'Geeks’}
- Dict[3][‘A’] → ‘Welcome’ #prints the value of the nested key ‘A’
- Dict[3][‘B’] → ‘To’ #prints the value of the nested key ‘B’
- Dict[3][‘c’] → ???? #what value do we get here?
- We emphasize the difference between dictionary vs list indices next.

1.3.2 Accessing dictionaries

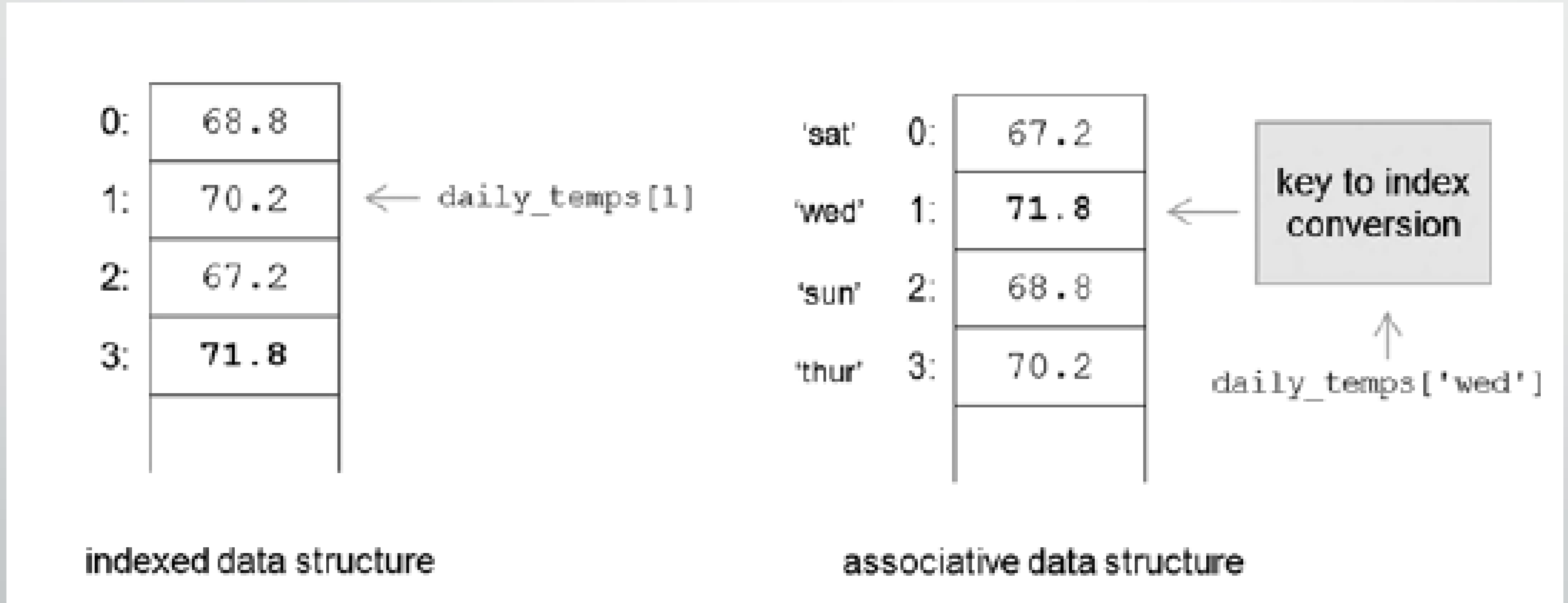
- The dictionary daily_temps (in Dierbach, 2015) stores the average temperature for each day of the week using a list. However, in this case (of the dictionary), each temperature has associated with it a unique key value ('sun', 'mon', etc.). This makes the list have some kind of meaning without having to refer to external information. The idea is that the keys should be named in a manner that they indicate some meaning to the values associated with them. The relationship between accessing items in a list and in a dictionary is demonstrated in figure 2.
- (Sturtz, n.d.) notes that "Although access to items in a dictionary does not depend on order, Python does guarantee that the order of items in a dictionary is preserved. When displayed, items will appear in the order they were defined, and iteration through the keys will occur in that order as well. Items added to a dictionary are added at the end. If items are deleted, the order of the remaining items is retained...

1.3.2 Accessing dictionaries

- further, ..." You can only count on this preservation of order very recently. It was added as a part of the Python language specification in version 3.7. However, it was true as of version 3.6 as well—by happenstance as a result of the implementation but not guaranteed by the language specification.
- *In Python 3.6, this ordering was just a nice consequence of that implementation of dict. In Python 3.7, however, dictionaries preserving their insert order is part of the language specification. As such, it may now be relied on in projects that support only Python >= 3.7 (or CPython >= 3.6).*"
- There are other ways we can access the values in the dictionary and these are explored in another part of this lesson.

Figure 2

List and dictionary access



(From Dierbach, 2015)

1.3.3 Building incrementally

- The dictionaries we have defined so far are based on the premise that we know all the keys and associated values. Suppose we don’t have the information and wish to build our dictionary? Sturtz (n.d.) demonstrates how you can start off with an empty dictionary and build up on it:

```
person = {} #an empty dictionary
>>> person['fname'] = 'Joe’
>>> person['lname'] = 'Fonebone’
>>> person['age'] = 51
>>> person['spouse'] = 'Edna’
 >>> person['children'] = ['Ralph', 'Betty', 'Joey’]
>>> person['pets'] = {'dog': 'Fido', 'cat': 'Sox’}
```

- The dictionary person now takes this form: { ‘fname’: ‘Joe’, ‘lname’: 'Fonebone’, 'age’: 51 , 'spouse’: 'Edna’, 'children’: ['Ralph', 'Betty', 'Joey’], ‘pets’: {'dog': 'Fido', 'cat': 'Sox’} }
- We can then access the dictionary items as demonstrated in section 1.3.2

1.3.4 Restrictions

- There are literally no restrictions on data values in a dictionary (of course as long as syntax rules are followed); however, there are restrictions on dictionary keys:
- We have seen so far that the keys of a dictionary can be of any type – int, float, string, or even boolean; even floats can be used as keys. However, there are some restrictions that all keys must abide by; these are described by Sturtz (n.d) –
- Rule #1 - a given key can appear in a dictionary only once. Duplicate keys are not allowed. A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once. Therefore, when you assign a value to an already existing dictionary key, it does not add the key a second time, but replaces the existing value. Let us demonstrate this using a simple example:
- Foods = {'breakfast': 'bacon', 'lunch': 'chicken', 'supper': 'pork'} #let's add something
- Foods['breakfast'] = 'eggs' #let's print the dictionary
- Print(Foods) → {'breakfast': 'eggs', 'lunch': 'chicken', 'supper': 'pork'} # the new value overrides the first

1.3.4 Restrictions

- Further, if you specify a key a second time during the initial creation of a dictionary, the second occurrence will override the first. Let's demonstrate this with our food dictionary:
- Foods = {'breakfast': 'bacon', 'lunch': 'chicken', 'supper': 'pork', 'lunch': 'mutton'} #which value will be picked?
- Print (Foods) → {'breakfast': 'bacon', 'supper': 'pork', 'lunch': 'mutton'} #the second occurrence overrides the first.
- Rule #2 - a dictionary key must be of a type that is immutable. Hitherto , we have already seen examples where two immutable types—integer and string—have served as dictionary keys. The same is true for float and Boolean type. Consequently tuples can also serve as keys since they are immutable. Further, it follows that lists can't serve as keys since they are mutable (we saw this in lesson 9, remember?). Let us demonstrate this using examples.

1.3.4 Restrictions

- Let us consider a dictionary of tuples as key values:
- x = {(0, 1): ‘z', (1, 0): ‘y', (2, 1): ‘x', (2, 2): ‘w’}
- Print (x[(1,0)]) → ‘y’ # Python reads the key exactly and in order
- x = {[0, 1]: ‘z’, [1, 0]: ‘y’, [2, 1]: ‘x’, [2, 2]: ‘w’} #let’s use a list instead of tuple
- Once you hit enter Python immediately detects a type error and returns “unhashable type: 'list’ “
- Dictionary keys must be hashable (to use the more technical reference). “an object must be hashable, which means it can be passed to a hash function. A hash function takes data of arbitrary size and maps it to a relatively simpler fixed-size value called a **hash value** (or simply hash), which is used for table lookup and comparison. (Stuart, n.d.).
- Immutable data types (structures or containers) are hashable, while mutable ones are not. In Python all built –in data types are immutable.

1.3.5 Operators and functions

- Whereas there are several operators that can be used with dictionaries, we describe examples of common operators; these are found in both table 1 and table 2. However, the examples are not exhaustive; nonetheless, the approach and syntax is the same in most cases.
- Foods = {‘breakfast’: ‘bacon’, ‘lunch’: ‘chicken’, ‘supper’: ‘pork’}
- ‘breakfast’ in Foods → true # in is Boolean and checks whether key is present
- len (Foods) → 3 #returns number of key: value pairs in dictionary
- del Foods[‘lunch’] #deletes the key and its associated value
- print(Food) → {‘breakfast’: ‘bacon’, ‘supper’: ‘pork’} #lunch has been deleted.

1.3.6 Methods

- A few examples on the use of the methods described in table 1 (we continue using the Foods dictionary unless otherwise indicated):
- Foods.clear() #empties the dictionary of all key:value pairs
- Foods ={}
- Foods.get('breakfast') → 'bacon' #returns the value associated with the key
- Foods.get('breakfast', 'none') #none is default value should key not be found
- Foods.get('brunch', 'not in list') → 'not in list'
- myDict = {'z': 10, 'y': 20, 'x': 30}
- list(myDict.items()) → [('z', 10), ('y', 20), ('x', 30)] # returns a list of tuples containing the key-value pairs in myDict.

1.3.6 Methods

- To find values in specific positions in myDict –
- (myDict.items())[1][0] → 'y' #find the item in index position 1 of the list;
- [('z', 10), ('y', 20), ('x', 30)] #view it as a list of tuples; first tuple at index 0 is ('z', 10), second tuple at index 1 is ('y', 20), and third tuple at index 2 is ('x', 30); then using similar notation for the tuple items as in list. Remember the only difference between a tuple and a list is that the former is immutable.
- list(myDict.keys()) → ['z', 'y', 'x'] #returns a list of the keys
- List(myDict.values()) #similarly will return a list of values; this is true if the values are duplicated; remember it is only keys that can't have duplicates, while values can take any value given to it as long as it is associated with a unique key.
- myDict.pop('y') → 20 # if key is present it will be popped (removed) and its associated value returned; if not present an error will be raised.
- Print(myDict) → {'z': 10, 'x': 30}

1.3.6 Methods

- A default argument can also be used with pop in order to prevent the error from arising:
- myDict.pop('t', 'absent') → 'absent' #t is not a key in the dictionary
- myDict.popitem() → ('x', 30) # removes the last key-value pair added from myDict and returns it as a tuple
- myDict = {'z': 10, 'y': 20, 'x': 30}
- myDict2 = { 'y': 60, 'x': 100}
- myDict.update(myDict2) → {'z': 10, 'y': 60, 'x': 100} #myDict is updated with new values from myDict2 for all the affected values; those not affected remain the same.
- Let us wind up by considering one consolidated example from Bansal (2018).

1.3.6 Methods

```
# demo for all dictionary methods
dict1 = {1: "Python", 2: "Java", 3: "Ruby", 4: "Scala"}

# copy() method
dict2 = dict1.copy()
print(dict2)

# clear() method
dict1.clear()
print(dict1)
```

```
# get() method
print(dict2.get(1))
# items() method
print(dict2.items())
```

1.3.6 Methods

```
# keys() method
print(dict2.keys())

# pop() method
dict2.pop(4)
print(dict2)

# popitem() method
dict2.popitem()
print(dict2)
```

```
# update() method
dict2.update({3: "Scala"})
print(dict2)

# values() method
print(dict2.values())
```

1.3.6 Methods

- Output:
- {1: 'Python', 2: 'Java', 3: 'Ruby', 4: 'Scala’}
- {}
- Python dict_items([(1, 'Python'), (2, 'Java'), (3, 'Ruby'), (4, 'Scala')])
- dict_keys([1, 2, 3, 4])
- {1: 'Python', 2: 'Java', 3: 'Ruby’}
- {1: 'Python', 2: 'Java’}
- {1: 'Python', 2: 'Java', 3: 'Scala’}
- dict_values(['Python', 'Java', 'Scala'])

Sets

2.1 Introduction

- Python has an in-built type called *set.* The characteristics of set are (Sturtz, 2018):
- Sets are unordered.
- Set elements are unique. Duplicate elements are not allowed.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.
- Dierbach (2015) defines it as "… a mutable data type with nonduplicate, unordered values, providing the usual mathematical set operations."
- We all have encountered sets from our high school mathematics and beyond, and so they are not a new concept *per se.*

2.1 Introduction

- Lambert (2014) highlights the typical operations of a set based on mathematics:
- 1 Return the number of items in the set.
- 2 Test for the empty set (a set that contains no items).
- 3 Add an item to the set.
- 4 Remove an item from the set.
- 5 Test for set membership (whether or not a given item is in the set).
- 6 Obtain the union of two sets. The union of two sets A and B is a set that contains all of the items in A and all of the items in B.

2.1 Introduction

- 7 Obtain the intersection of two sets. The intersection of two sets A and B is the set of items in A that are also items in B.
- 8 Obtain the difference of two sets. The difference of two sets A and B is the set of items in A that are not also items in B.
- 9 Test a set to determine whether or not another set is its subset. The set B is a subset of set A if and only if B is an empty set or all of the items in B are also in A.
- Lambert (2014) further adds that "the difference and subset operations (8 and 9) are not symmetric. For example, the difference of sets A and B is not always the same as the difference of sets B and A."
- We shall demonstrate these operations using suitable examples in this part. As we shall see the behavior of a set is very similar to other collections in Python; indeed, the function of many of these operations on the class is the same across the board for other collection items.

2.2 Set behavior

- (Goodrich et al., 2018) describes what they believe are the five most fundamental behaviors of a set; they further add that these five behaviors are the basis (or rather all other behaviors of a set can be derived from) of a set: "
- S.add(e): Add element e to the set. This has no effect if the set already contains e.
- S.discard(e): Remove element e from the set, if present. This has no effect if the set does not contain e.
- e in S: Return True if the set contains element e. In Python, this is implemented with the special contains method.
- len(S): Return the number of elements in set S. In Python, this is implemented with the special method len .
- iter(S): Generate an iteration of all elements of the set. In Python, this is implemented with the special method __iter__ ."

2.2 Set behavior

- The following three operations are used for removing elements from a set (Goodrich, 2018):
- S.remove(e)**:** Remove element e from the set. If the set does not contain e, raise a KeyError.
- S.pop()**:** Remove and return an arbitrary element from the set. If the set is empty, raise a KeyError.
- S.clear( )**:** Remove all elements from the set.
- The next group of behaviors perform Boolean comparisons between two sets.

2.2 Set behavior

- These are operations for performing comparison between sets (Goodrich, 2018):
- S == T**:** Return True if sets S and T have identical contents.
- S != T**:** Return True if sets S and T are not equivalent.
- S <= T**:** Return True if set S is a subset of set T.
- S < T**:** Return True if set S is a *proper* subset of set T.
- S >= T**:** Return True if set S is a superset of set T.
- S > T**:** Return True if set S is a *proper* superset of set T.
- S.isdisjoint(T)**:** Return True if sets S and T have no common elements.

2.2 Set behavior

- Thirdly, Goodrich (2018) adds, “there exists a variety of behaviors that either update an existing set, or compute a new set instance, based on classical set theory operations:
- S | T: Return a new set representing the union of sets S and T.
- S |= T: Update set S to be the union of S and set T.
- S & T: Return a new set representing the intersection of sets S and T.
- S &= T: Update set S to be the intersection of S and set T.
- S ˆ T: Return a new set representing the symmetric difference of sets
- S and T, that is, a set of elements that are in precisely one of S or T.
- S ˆ= T: Update set S to become the symmetric difference of itself and set T.
- S − T: Return a new set containing elements in S but not T.
- S −= T: Update set S to remove all common elements with set T.

Table 3

Set operators

Set operator	Set A = {1,2,3}	Set B = {3,4,5,6}	
membership	`1 in A`	True	*True if 1 is a member of set*
add	`A.add(4)`	{1,2,3,4}	*Adds new member to set*
remove	`A.remove(2)`	{1,3}	*Removes member from set*
union	`A | B`	{1,2,3,4,5,6}	*Set of elements in either set A or set B*
intersection	`A & B`	{3}	*Set of elements in both set A and set B*
difference	`A - B`	{1,2}	*Set of elements in set A, but not set B*
symmetric difference	`A ^ B`	{1,2,4,5,6}	*Set of elements in set A or set B, but not both*
size	`len(A)`	3	*Number of elements in set (general sequence operation)*

(From Diebarch, 2015)

2.3 Set operations

- Dierbach (2015) used two sets A and B to demonstrate the meaning and application of mathematical operations on sets. In this section we demonstrate the use of sets in Python via some common operations.
- (*Sets in Python*, 2016) state that “The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table. Since sets are unordered, we cannot access items using indexes as we do in lists.” Let us examine a few examples from the same source:
- mySet = {‘my’, ‘first’, ‘set’} #see the curly brackets? Create a set
- # typecasting list to set
- myset = set(["a", "b", "c"])
- print(myset) → {'c', 'b', 'a'}

2.3 Set operations

- # Adding element to the set
- myset.add("d")
- print(myset) → {'d', 'c', 'b', 'a'}
- myset = {"Geeks", "for", "Geeks"}
- print(myset) → {"Geeks", "for"} #set can't have duplicate values
- myset[1] = "Hello"
- print(myset) → TypeError: 'set' object does not support item assignment # set items are can't be changed; can only add or delete items from a set.
- myset = {"Geeks", "for", 10, 52.7, True} #can store different data types like dictionary

2.3 Set operations

```
# A Python program to
# demonstrate adding elements
# in a set

# Creating a Set
teachers = {"Jay", “Simba", “Newton"}

print(“teachers:", end = " ")
print(teachers)
# This will add Susan
# in the set
teachers.add(“Susan")
```

```
# Adding elements to the
# set using iterator
for x in range(1, 6):
        teachers.add(x)

print("\nSet after adding new teacher:", end = " ")
print(teachers)
```

2.3 Set operations

```
# Python Program to
# demonstrate union of
# two sets

teachers = {"Jay", “Simba", “Newton"}
vampires = {“Elena", “Damon"}
dracula = {“Lee", “Stefan"}

# Union using union()
# function
population = teachers.union(vampires)
```

```
print("Union using union() function")
print(population)

# Union using "|“ operator
population = teachers|dracula

print("\nUnion using '|' operator")
print(population)
```

Output:

```
Union using union() function
{"Jay", “Simba", “Newton“, “Elena", “Damon" }
Union using '|' operator
{"Jay", “Simba", “Newton", “Lee", “Stefan" }
```

2.3 Set operations

- Other operations on sets are performed much like the ones we have described; for intersections, differences and so on. However, let us make a special mention of the type called *frozenset*.
- Python provides another built-in type called a **frozenset**, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozenset. (Sturtz, 2018). Consider the following example, also from Sturtz (2018):
- x = frozenset(['foo', 'bar', 'baz']) #declaration of a frozen set
- x & {'baz', 'qux', 'quux'} →frozenset({'baz'}) #intersection of the two; this is permitted as long as there's no attempt to modify x
- Any operation that attempts to modify x will result in an error and will not be allowed; for example, to pop, add, or clear operation.

2.3 Set operations

- Consider the following example:
- x = frozenset(['foo', 'bar', 'baz’])
- X.add(‘don’) → AttributeError: 'frozenset' object has no attribute 'add’
- X.pop() → AttributeError: 'frozenset' object has no attribute ‘pop’
- X.clear() → AttributeError: 'frozenset' object has no attribute ‘pop’
- Thus any attempt to modify the frozenset results in an error in Python.
- Frozensets are useful in situations where you want to use a set, but you need an immutable object. For example, you can’t define a set whose elements are also sets, because set elements must be immutable. (Sturtz, 2018)

Summary

- Elements of indexed linear data structures, such as lists, are ordered—the first element (at index 0), second element (at index 1), and so forth. In contrast, the elements of an associative data structure are unordered, instead accessed by an associated key value. In Python, an associative data structure is provided by the *dictionary type.*
- The operations that can be performed on dictionaries relate to their properties, namely: mutability, associative nature, keys, values, and no duplication.
- A set is a mutable data type with nonduplicate, unordered values, providing the usual mathematical set operations.
- The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table.
- Python provides another built-in type called a **frozenset**, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozenset

References

- Bansal, A. (2018, January 25). *Python Dictionary*. GeeksforGeeks. https://www.geeksforgeeks.org/python-dictionary/
- Dierbach, C. (2015). *Introduction to Computer Science Using Python: A computational problem-solving focus*. Wiley.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2018). *Data structures and algorithms in Python*. Wiley.
- Lambert, K. (2014). *Fundamentals of Python*. Cengage Learning Ptr.
- *Sets in Python*. (2016, May 16). GeeksforGeeks. https://www.geeksforgeeks.org/sets-in-python/?ref=lbp
- *Sophophobia Definition & Meaning | YourDictionary*. (n.d.). Www.yourdictionary.com; Lovetoknow. Retrieved November 11, 2023, from https://www.yourdictionary.com/sophophobia

References

- Sturtz, J. (n.d.). *Dictionaries in Python – Real Python*. Realpython.com. Retrieved November 11, 2023, from https://realpython.com/python-dicts/#defining-a-dictionary
- Sturtz, J. (2018, August 20). *Sets in Python – Real Python*. Realpython.com. https://realpython.com/python-sets/#defining-a-set
- W3schools. (2018). *Python Dictionaries*. W3schools.com. https://www.w3schools.com/python/python_dictionaries.asp