Computer Science I CSCI-141
Python Dictionary Introduction Lecture

10/09/2018

dictionary:

1. a reference book containing words arranged along with information about their forms, ... meanings and idiomatic uses
2. a reference book listing alphabetically terms or names important to a subject...
3. a general comprehensive list, collection or repository...

1 Problem

Python lists are great for storing ordered sequences of data and accessing individual elements based on their position in the sequence. For example, lists work well for storing a sorted sequence of distinct numbers. If the numbers are stored in ascending order, then the list ordering specifies that the number stored at index k is smaller than all numbers stored at indices greater than k, and is greater than all numbers stored at indices less than k. We can also easily access the median value, or the value at any particular index by exploiting this order and directly accessing a specific index of the list.

As another example, suppose students sign up for a course having limited capacity. Preference is given based on the order students sign up. A data structure must be used to maintain a record of those students who have signed up. Again, a list works great for this scenario. Students are stored in the list in the order that they sign up for the course. That ordering can be used to determine who should be enrolled in the course, as well as who has priority in case a seat becomes available.

This positional representation is not ideal, however, for all data collections.

Suppose we are interested in maintaining a collection of names along with their most common nickname. We'd like to be able to take a given name, and efficiently look up the associated nickname.

We could store this information in a list. Each list element could be a (name, nickname) pair. And then we could search through the list to locate the name of interest.

But in this case, the positional ordering of the data provided by the list is not particularly meaningful for our objective. We'd prefer a data structure that organizes the data by *association*. In our case, name associated with most common nickname.

2 The Python Dictionary

The type of data structure we want is known as an *associative array*, or *map*, which stores a collection of (key, value) pairs. For the example above, the (key, value) pairs are (name, nickname) pairs.

The Python *dictionary* is an implementation of a *map*.

2.1 Construction

We can create an empty dictionary two different ways, similar to two different ways to create an empty list:

```
nicknames = dict()      # creates a dictionary with no entries
nicknames = {}          # also creates a dictionary with no entries
```

We can create and populate a dictionary with initial values:

```
nicknames = {'Elizabeth' : 'Beth', 'Robert' : 'Bob'}
```

2.2 Inserts, Updates, Fetches and Queries

We can add, update and access entries in a dictionary using indexing syntax. The key fills the place of the integer index used in list access. Note that it is not possible to have duplicate keys in a dictionary. Each key is unique.

The syntax for inserting a new (key, value) pair is identical to the syntax for updating an existing (key, value) pair.

```
nicknames[ 'Jonathan' ] = 'Jon'     # Inserts entry ('Jonathan', 'Jon')
nicknames[ 'Rebecca' ] = 'Becky'    # Inserts entry ('Rebecca', 'Becky')
nicknames[ 'Rebecca' ] = 'Becca'    # Updates ('Rebecca', 'Becky')
                                    # to ('Rebecca', 'Becca')
nicknames[ 'Rebecca' ]              # Access value for key 'Rebecca';
                                    # returns 'Becca'
```

Accessing a non-existent key raises an exception. We must guard against such exceptions by using the Python keyword `in` to verify the existence of a key before trying to access its associated value.

```
nicknames[ 'James' ]                # Access value for key 'James';
                                    # raises exception
'James' in nicknames                # returns False
```

2.3 Deleting Entries

Dictionary entries can be deleted by specifying the key. Again, trying to delete an entry corresponding to a non-existent key causes an exception to be raised.

```
del nicknames[ 'Jonathan' ]       # Deletes the ('Jonathan', 'Jon') entry
del nicknames[ 'James' ]          # Raises exception
```

2.4 Iterating through Dictionaries

In the context of a `for` loop, we use the `in` keyword to iterate through the keys or values contained in the dictionary.

Note, however, that entries in a dictionary are not stored in a 'normal' order; when iterating through a dictionary using `for`, the order in which the entries emerge appears random.

```
for name in nicknames:              # Iterates over the keys
for name in nicknames.keys():       # Iterates over the keys
for nick in nicknames.values():     # Iterates over the values
```

2.5 Restrictions on Key Type

Just as Python lists can contain data of different types, Python dictionaries can contain keys of different types and values of different types. There are no restrictions on the type of data used for the value in the (key, value) pair. However, **a key must be hashable**.

To be hashable, an element must be encodable into a constant value, typically an integer. The consequence of hashability is that the element must be *immutable*; otherwise, its encoding will not be constant.

Strings and integers are common data types used as key types in a Python dictionary. A list is an example of a data type that is not allowed as a key in a Python dictionary.

```
mixMatch = {}                          # Creates empty dictionary
mixMatch[ 'Hello' ] = [ 2, 3, 7 ]  # An entry with key of type string
                                       # and value of type list
mixMatch[ 3.14 ] = False               # An entry with key of type float
                                       # and value of type boolean
mixmatch[ [1, 2, 3] ] = 42         # TypeError: unhashable type: 'list'
```

2.6 Demonstration Code

See the module, `dictionary_demo.py`, for more examples of dictionary use.

3 Related Data Structure: The Python Set

The dictionary's requirement for a unique key value is also a requirement for values in a *mathematical set*. The Python set data structure works like a dictionary that has no associated values; logically there are only keys.

```
nicknames = set()  # creates an empty set
nicknames = {}     # dictionary or set? Oh No! ambiguous syntax!
```

We have to call the function `set()` to make an empty set. Otherwise, the syntax of constructing non-empty sets is the same as a dictionary, other than having no value part:

```
s = { 1, 2, 3, 5, 8, 13 }
```

Here are some example set operations:

```
>>> s = { 25, "or", 6, "to", 4 }    # Keys can be heterogeneous.
>>> s
{4, 6, 'to', 25, 'or'}
>>> 6 in s
True
>>> 5 in s
False
>>> 3.6 in s
False
>>> for thing in s:
...   print( "-->", thing, "<--" )
...
--> 4 <--
--> 6 <--
--> to <--
--> 25 <--
--> or <--
>>> list( s )
[4, 6, 'to', 25, 'or']
>>> tuple( s )
(4, 6, 'to', 25, 'or')
```

Ordering is not determined by insertion; output order is unpredictable, and it can change on each run of a program. The run below, with the same set definition as the above, prints in a different order.

```
$ python3
>>> s = { 25, "or", 6, "to", 4 }
>>> s
{'to', 4, 6, 'or', 25}
>>> s.add( 9)
>>> s
{'to', 4, 6, 9, 'or', 25}
>>> s.remove(25)
>>> s.remove( 'fred')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fred'
```

We can use the `set.add()` function to add to a set, and the `set.remove()` function to remove an element, but the element must be present, or it is a `KeyError`.