ECS 32A - Dictionaries and Tuples

Aaron Kaloti

UC Davis - Summer Session #1 2020

Collections of Values

- Collections we've seen:
  - Strings are an immutable collection of characters.
  - Lists are a mutable collection of values of any type.
- *Upcoming*: dictionaries, which allow the programmer to express key-value pairs.

Review: Names and Ages

- Recall this example from the lecture on lists.

```
>>> names = ["Aaron", "Richard", "Bobby"]
>>> ages = [22, 40, 18]
>>> names[1]
'Richard'
>>> ages[1]
40
>>> names[2]
'Bobby'
>>> ages[2]
18
```

- *Issue*: deleting Richard requires two uses of `del`.

```
>>> names = ["Aaron", "Richard", "Bobby"]
>>> ages = [22, 40, 18]
>>> del names[1]
>>> del ages[1]
>>> names
['Aaron', 'Bobby']
>>> ages
[22, 18]
```

Dictionaries

- We can use dictionaries to keep the data aligned together[1]. Use dictionaries when you wish to represent key-value pairs.

Example

- Key: person's name.
- Value: person's age.

```
>>> d = {'Aaron': 22, 'Richard': 40, 'Bobby': 18}
>>> d
{'Aaron': 22, 'Richard': 40, 'Bobby': 18}
>>> d['Aaron']
22
>>> d['Bobby']
18
>>> d[2]
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    d[2]
KeyError: 2
```

- We can now delete the data with one `del` statement.

```
>>> del d['Richard']
>>> d
{'Aaron': 22, 'Bobby': 18}
```

1. Generally speaking, classes (the last topic in this course) may be a preferred way of keeping data aligned together, but when there is a key-value relationship, dictionaries are preferable.

Other Cases of Convenience: Books

- Using lists:

```
>>> titles = ["The Fault in Our Stars", "Fangirl", "Fragile Like Us"]
>>> authors = ["John Green", "Rainbow Rowell", "Sara Barnard"]
```

- Using dictionaries:

```
>>> books = {"The Fault in Our Stars": "John Green", "Fangirl": "Rainbow Rowell", "Fragile Like Us": "Sara Barnard"}
>>> books["Fangirl"]
'Rainbow Rowell'
```

Other Cases of Convenience: Homework Scores

- Using lists:

```
>>> student_id = ["9001", "9002", "9003"]
>>> hw4_scores = [50, 49, 45]
```

- Using dictionaries:

```
>>> hw4_scores = {"9001": 50, "9002": 49, "9003": 45}
>>> hw4_scores["9002"]
49
```

- Because I can't use real student IDs, I used four-digit fake ones.

"Indexing" Into Dictionaries

- You can "index" a dictionary by using a key.

```
>>> garbage = {'abc': 88, 'xyz': 5.3, 'blah': 'hello', 2: 'nah'}
>>> garbage['abc']
88
>>> garbage[2]
'nah'
```

- Note that a value cannot be treated as a key. For example, the below will crash instead of getting "blah".

```
>>> garbage['hello']
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    garbage['hello']
KeyError: 'hello'
```

"abc" -> 88

"xyz" -> 5.3

"blah" -> "hello"

2 -> "nah"

Mutability

- Dictionaries are mutable.

Example #1

```
>>> garbage
{'abc': 88, 'xyz': 5.3, 'blah': 'hello', 2: 'nah'}
>>> garbage[1] = 'hi'
>>> garbage[3] = 58000
>>> garbage['yo'] = "yo"
>>> garbage
{'abc': 88, 'xyz': 5.3, 'blah': 'hello', 2: 'nah', 1: 'hi', 3: 58000, 'yo': 'yo'}
```

"abc" -> 88
"xyz" -> 5.3
"blah" -> "hello"
2 -> "nah"
1 -> "hi"
3 -> 58000
"yo" -> "yo"

Mutability

Example #2[1]

```
>>> d = {"key1": 30, "key2": 45}
>>> d
{'key1': 30, 'key2': 45}
>>> d["key2"] = 80
>>> d
{'key1': 30, 'key2': 80}
```

1. This example was added after the corresponding lecture.

No Slicing

- Dictionaries cannot be sliced.

```
>>> garbage
{'abc': 88, 'xyz': 5.3, 'blah': 'hello', 2: 'nah', 1: 'hi', 3: 58000, 'yo': 'yo'}
>>> garbage[1:3]
Traceback (most recent call last):
  File "<pyshell#83>", line 1, in <module>
    garbage[1:3]
TypeError: unhashable type: 'slice'
```

- Regardless of whether the keys are numerical or not, the dictionary does not see the keys as ordered in any significant way, hence why splicing is unsupported.

The `in` Operator

- You can use the `in` operator to check if a dictionary has a certain key.

```
>>> staff_labels = {'Aaron': 'Lecturer',
                    'Matt': 'TA',
                    'Nikhil': 'TA'}
>>> 'Aaron' in staff_labels
True
>>> 'Bobby' in staff_labels
False
```

Traversing a Dictionary

- Ideally, you should use a `for` loop to traverse a dictionary.

```
>>> staff_labels = {'Aaron': 'Lecturer',
                    'Matt': 'TA',
                    'Nikhil': 'TA'}
>>> for name in staff_labels:
        print(name)

Aaron
Matt
Nikhil
```

"Aaron" -> "Lecturer"
"Matt" -> "TA"
"Nikhil" -> "TA"

- Notice that `name` takes on each of the *keys* in the dictionary, not the values.
- To traverse the values of the dictionary, we would merely need to index into the dictionary with each key.

```
>>> for name in staff_labels:
        print(staff_labels[name])

Lecturer
TA
TA
```

Traversing a Dictionary

- Can use the `items` method with a `for` loop to more easily traverse the keys and values.

```
>>> for (name,label) in staff_labels.items():
        print("{} is a {}.".format(name,label))

Aaron is a Lecturer.
Matt is a TA.
Nikhil is a TA.
```

Traversing a Dictionary

- Can traverse the values directly with the `values` method.

```
>>> for label in staff_labels.values():
        print(label)

Lecturer
TA
TA
```

Traversing with `while` Loop

- We previously saw that we could iterate over the values of a list with a `while` loop. This worked because the indices of a list always form a contiguous sequence of integers. In the example below, those integers are 0, 1, and 2.

```
>>> fruits = ['apple', 'banana', 'orange']
>>> i = 0
>>> while i < len(fruits):
        print(fruits[i])
        i += 1

apple
banana
orange
```

- Since there is no guarantee that a dictionary's keys form a contiguous sequence of integers, it is not generally possible to iterate over a dictionary with a `while` loop.

List Traversals vs. Dictionary Traversals

- Traversing a list's indices vs. a dictionary's keys:

```
items = [8,5,-1]
for i in range(len(items)):
    print(i)
```

list-index-traversal.py

```
0
1
2
```

```
d = {'michael': 38, 'jake': 27}
for k in d:
    print(k)
```

dict-index-traversal.py

```
michael
jake
```

- Traversing a list's values vs. a dictionary's values:

```
items = [8,5,-1]
for i in range(len(items)):
    print(items[i])
```

list-values-traversal.py

```
8
5
-1
```

```
d = {'michael': 38, 'jake': 27}
for k in d:
    print(d[k])
```

dict-values-traversal.py

```
38
27
```

Merging Two Lists Into a Dictionary

- Use `zip()`.

Example

```
>>> names = ['Bob','Frank','SpongeBob']
>>> ages = [22,33,30]
>>> d = dict(zip(names,ages))
>>> d
{'Bob': 22, 'Frank': 33, 'SpongeBob': 30}
```

Example: Reverse Lookup[1]

Prompt

- In a file called `exercises.py`, write a function called `reverse_lookup` that takes a dictionary and a value and returns the key that maps to that value. If a key cannot be found, then `None` should be returned.
    - Examples:
        - `reverse_lookup({'A': 8, 'B': 13}, 13)` returns `'B'`.
        - `reverse_lookup({'A': 8, 'B': 13}, 5)` returns `None`.
        - `reverse_lookup({}, 5)` returns `None`.
- What important assumption is this function making?

Solution

```python
def reverse_lookup(d, val):
    for key in d:
        # Check if value corresponding to @key equals @val.
        if d[key] == val:
            return key
    return None
```

1. This example is from section 11.4 of your book.

Example: Keys `==` Values

Prompt

- In `exercises.py`, write a boolean function called `keys_equal_values` that takes a dictionary and returns `True` if each key in the dictionary equals its respective value; otherwise, it returns `False`.
    - Examples:
        - `keys_equal_values({5: "abc", 58: "blah"})` should return `False`.
        - `keys_equal_values({"Aaron": 22, "Matt": 23})` should return `False`.
        - `keys_equal_values({43: 43, -5: -5, "abc": "abc"})` should return `True`.

Solution

```python
def keys_equal_values(d):
    for k in d:
        if k != d[k]:
            return False
    return True
```

Example: Build Dict of Names/Ages

Prompt

- In `exercises.py`, write a function called `get_names_ages` that takes a list of names and asks the user for the age of each person named. The function should return a dictionary mapping each name to the corresponding age. If a name is duplicated, then return "ERROR".

Solution

```
def get_names_ages(names):
    d = {}
    for name in names:
        # If @name has already been mapped/seen and given an age.
        if name in d:
            return "ERROR"
        age = int(input("Enter age of {}: ".format(name)))
        d[name] = age
    return d
```

Example: Histogram[1]

- Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:
    1. 26 variables.
        - Needless to say, this approach is horrible.
    2. List with 26 elements.
        - This approach is OK, but having to convert a letter into its corresponding index is a pain (and involves concepts that were in the Appendices of the String lecture slides).
    3. Dictionary with the letters as keys and the counts as values.

1. This example is from section 11.2 of your book.

Example: Histogram

- Suppose you are given a string and you want to count how many times each letter appears.
- Let's use a dictionary[1,2]:

```
def histogram(s):
    d = {}
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

histogram.py

```
>>> histogram('abcbcc')
{'a': 1, 'b': 2, 'c': 3}
>>> histogram('brontosaurus')
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
>>> histogram('')
{}
```

1. `if c not in d` could have also been `if not c in d`.

2. In your book, the second line is `d = dict()` instead of `d = {}`. The lines have the same effect. However, the few online sources I quickly looked at seemed to say that `d = {}` is faster (which is kind of dumb, err, I mean, unintuitive and inexplicable). Also, I personally prefer `d = {}`. You don't need to know about `dict()` for the exam.

The `setdefault` Method

- We could remove the conditional statements in the histogram example by using the `setdefault` method.

```
def histogram(s):
    d = {}
    for c in s:
        d[c] = d.setdefault(c,0) + 1
    return d
```

histogram-setdefault.py

- On interpreter:

```
>>> histogram('brontosaurus')
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

Example: Anagrams

Prompt

- In `exercises.py`, write a function called `are_anagrams` that takes two strings and returns `True` if one is an anagram of the other and `False` otherwise. Your solution should use a dictionary[1].
    - Examples:
        - `are_anagrams("abc","cba")` returns `True`.
        - `are_anagrams("abcba","bbaca")` returns `True`.
        - `are_anagrams("abcba","bbbaa")` returns `False`.
        - `are_anagrams("abcc","cba")` returns `False`.

1. If you take ECS 32B or ECS 36C, you will learn why it is faster (in terms of the speed of the program, at least for large strings) to use a dictionary in this case (as opposed to a list). For now, just take it for granted.

Example: Anagrams

Solution

```python
# Assumes have the histogram function in a file called
# histogram.py in the same folder.
from histogram import histogram

##def histogram(s):
##    d = {}
##    for c in s:
##        d[c] = d.setdefault(c,0) + 1
##    return d

def are_anagrams(s1, s2):
    hist1 = histogram(s1)
    hist2 = histogram(s2)
    print("hist1:", hist1)
    print("hist2:", hist2)
    return hist1 == hist2
    # if hist1 == hist2:
        # return True
    # else:
        # return False
```

More on Keys

- A key can be any *immutable* type.

```
>>> d = {}
>>> d[5] = 8
>>> d
{5: 8}
>>> d["abc"] = 18
>>> d
{5: 8, 'abc': 18}
```

- Since a list is *mutable*, it cannot be used as a key.

```
>>> d[[3,8,9]] = 11
Traceback (most recent call last):
  File "<pyshell#140>", line 1, in <module>
    d[[3,8,9]] = 11
TypeError: unhashable type: 'list'
```

Former Order Preservation of Keys

- In past versions of Python, a dictionary would "forget" the order in which its keys were inserted. In such past versions, I could not ask what the output of the program below was and expect a deterministic answer.

```
d = {}
d['b'] = 1
d['c'] = 2
d['d'] = -1
print(d)
```

example1.py

- Here would be two possibilities:

```
{'b': 1, 'c': 2, 'd': -1}
```

```
{'b': 1, 'd': -1, 'c': 2}
```

Former Order Preservation of Keys

- However, if you are using Python 3.7, you *can* depend on a deterministic order[1]. This was also supposedly true for Python 3.6 but not guaranteed[2].
- That said, it is bad form to depend on the order of the keys, and it is possible that a future Python version could remove the guarantee that the keys have a deterministic order.

1. Specifically, it is guaranteed that the dictionary will preserve the insertion order of the keys, as stated here: https://docs.python.org/3.7/whatsnew/3.7.html

2. Source: https://docs.python.org/3/whatsnew/3.6.html#whatsnew36-compactdict

Preserving Order vs. Sorting

- Note that there is a difference between preserving the order of insertion of the keys vs. keeping the keys sorted at all times.
- I can insert the keys in a sorted order, and the keys will consequently be sorted.

```
>>> d = {}
>>> d[5] = 81
>>> d[6] = "abc"
>>> d[7] = 5.3
>>> d
{5: 81, 6: 'abc', 7: 5.3}
```

- However, I can also insert the keys in an unsorted order, and the keys will consequently be unsorted.

```
>>> d = {}
>>> d[7] = 81
>>> d[2] = 53
>>> d[5] = "abc"
>>> d
{7: 81, 2: 53, 5: 'abc'}
```

- Note that in both of the above examples, the order in which I inserted the keys is preserved. (I use Python 3.6.8.)

Sorting Keys

- It is possible to sort the keys, which may be useful if you wish to traverse the keys in a sorted order.

```
>>> d
{7: 81, 2: 53, 5: 'abc'}
>>> for k in sorted(d):
        print("d[{}] = {}".format(k,d[k]))

d[2] = 53
d[5] = abc
d[7] = 81
>>> hist = {'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
>>> for key in sorted(hist):
        print("hist[{}] = {}".format(key,hist[key]))

hist[a] = 1
hist[b] = 1
hist[n] = 1
hist[o] = 2
hist[r] = 2
hist[s] = 2
hist[t] = 1
hist[u] = 2
```

Tuples

- Tuples are another collection (like lists or dictionaries) that has the following characteristics:
  - immutable (like strings)
  - indexable (like strings and lists)
- They're basically immutable lists.
- Syntactically, a tuple is denoted by comma-separated values[1]. Usually, parentheses are placed around the values, but it is optional.

```
>>> my_first_tuple = 5,8,"abc",3.3
>>> my_first_tuple
(5, 8, 'abc', 3.3)
>>> my_second_tuple = (88, 76, "hello")
>>> my_second_tuple[2]
'hello'
>>> my_second_tuple[1] = 90
...
TypeError: 'tuple' object does not support item assignment
>>> len(my_second_tuple)
3
>>> del my_second_tuple[0]
...
TypeError: 'tuple' object doesn't support item deletion
```

1. Some of you may have accidentally encountered a tuple while writing `ess()` in HW 4.

k-Tuple

- A k-tuple is a tuple that has k elements.
- Examples:
  - A 2-tuple (e.g. `(5,8)`) has 2 elements.
  - A 5-tuple (e.g. `(3,"abc",0.8,9,10)`) has 5 elements.

One-Element Tuples

- You need a comma to denote a 1-tuple literal.

```
>>> my_third_tuple = 3,
>>> my_third_tuple
(3,)
>>> a = 3
>>> a
3
```

Why Tuples?

- **"Reason" #1**: The Python "culture" prefers tuples for heterogeneous (i.e. different) types and lists for homogenous (i.e. same) types.
  - Example: Would store the values 5.5, "hi", and -3 in a tuple rather than a list.

1. This is similar to the C++ "culture" preferring that structs be used for data types with no methods while classes be used for data types that *do* have methods. Again, there is nothing binding about this, and you could use structs the same way you use classes. I once wrote a Checkers game using gigantic structs (the code was pretty awful), and I did not get struck by lightning or anything.

But Really, Why Tuples?

- **Real Reason #1**: Returning multiple values from a function. This is the only time I have personally seen tuples used.

```
>>> def goo():
                important_value1 = "blah"
                important_value2 = "blahblah"
                return (important_value1, important_value2)

>>> a,b = goo()
>>> a
'blah'
>>> b
'blahblah'
>>> c = goo()
>>> c
('blah', 'blahblah')
```

- The `a,b =` syntax can be used without functions as well.

```
>>> a,b = 5,3
>>> a
5
>>> b
3
```

But Really, Why Tuples?

- **Real Reason #2**: Keys in dictionaries. We previously saw that lists cannot be used as keys for dictionaries, because lists are mutable. However, tuples *can* be used as keys, because tuples are immutable.

```
>>> d = {}
>>> d[5,8] = "abc"
>>> d
{(5, 8): 'abc'}
>>> d[5]
...
KeyError: 5
>>> d[5,8]
'abc'
```

Comparing Collections

Collection	String	List	Dictionary	Tuple
Index	Integer	Integer	Key	Integer
Empty	`""`	`[]`	`{}`	`()`
Immutable?	Yes	No	No	Yes
Contains	Characters	Values of any/mixed types	Key-value pairs of any/mixed types	Values of any/mixed types
Maps	Index to Character	Index to Value	Key to Value	Index to Value

Appendix: Sets

- As stated on W3Schools, a set is a collection which is unordered and unindexed.

```
>>> staff_names = {"aaron", "matt", "jiarui", "nikhil", "sanjat"}
>>> staff_names[1]    # unindexed
...
TypeError: 'set' object does not support indexing
>>> "matt" in staff_names
True
>>> for name in staff_names:
        print(name)

matt     # unordered
nikhil
sanjat
jiarui
aaron
>>> staff_names.add("tracy")
>>> staff_names
{'matt', 'nikhil', 'sanjat', 'tracy', 'jiarui', 'aaron'}
>>> len(staff_names)
6
>>> staff_names.remove('sanjat')
>>> staff_names
{'matt', 'nikhil', 'tracy', 'jiarui', 'aaron'}
```

Appendix: Sets

- You may wonder what the point of using sets is, given that sets are unordered and unindexed and have unchangeable elements[1]. As you will (or should, at least) learn in ECS 32B, sets (and dictionaries) are implemented using hash tables. This means that for *large* amounts of data, sets and dictionaries are *much* faster at taking in that data (and checking the presence of it) than are lists and tuples. For example, checking if an element is in a set of 10000 elements is much faster than checking if it is in a list of 10000 elements.
- For those of you with a C++ background, do not be fooled by `std::set` in C++. Sets and dictionaries in Python are akin to `std::unordered_set` and `std::unordered_map` in C++. `std::set` is slower but maintains the lexicographic (sorting) order of the elements (which is useful in certain cases).

1. This is not the same as immutability, as you can mutate a set by using the `add` method, as shown on the previous slide. See frozenset for a completely immutable version of set.

Appendix: Sets

- For more on sets, see the [W3Schools page: https://www.w3schools.com/python/python_sets.asp.

Collection	String	List	Dictionary	Tuple	Sets
Index	Integer	Integer	Key	Integer	None
Empty	`""`	`[]`	`{}`	`()`	`set()`
Immutable?	Yes	No	No	Yes	Sort of
Contains	Characters	Values of any/mixed types	Key-value pairs of any/mixed types	Values of any/mixed types	Values of any/mixed types
Maps	Index to Character	Index to Value	Key to Value	Index to Value	No mapping