Dictionaries in Python

APRIL 6, 2015
Tuples in Python

- Tuples are closely related to lists in Python
- The one obvious syntactic difference is that round brackets are used to define tuples rather than square brackets.

**Example:** \( T = (0, -90, "hello") \)

- Elements of a tuple can be accessed just like elements of a list

**Example:** \( T[0] \)

**Example:** \( T[0:2] \)
Main difference

- Tuples are \textit{immutable} versions of lists.
- Tuples do not come with any of the methods that modify lists in place: \texttt{insert}, \texttt{extend}, \texttt{remove}, etc.

So why use tuples??

1. \textbf{Faster than lists}. Use if you know that you only have to scan, not modify. Also more memory efficient.
2. Can be used (instead of lists) in situations that require immutable objects (e.g., dictionaries).
Easy to go from lists to tuples (and back)

- Example:
  >>> L = [3, 4, "hello"]
  >>> T = tuple(L)
  >>> T
  (3, 4, 'hello')
  >>> LL = list(T)
  >>> LL
  [3, 4, 'hello']
  >>>
Dictionaries

- *Dictionary* is a Python data structure that consists of key-value pairs.

- **Example:**
  
  \[D = \{\text{“to”}: 10, \text{“be”}: 20, \text{“it”}: 31, \text{“go”}: 20\}\]

- Here the *keys* are “to”, “be”, “it”, and “go”

- The *values* are 10, 20, 31, and 20

- The dictionary \(D\) is a function that associates a value to each key.

- Keys in a dictionary have to be distinct, i.e., no duplicate keys.

- Any *immutable* object can be a key. Thus lists cannot be keys; this is where tuples becomes useful.
Accessing items in a dictionary

- **Example:**
  
  \[ D = \{ \text{“to”}: 10, \text{“be”}: 20, \text{“it”}: 31, \text{“go”}: 20 \} \]
  
  \[ D[\text{“to”}] \text{ evaluates to } 10 \]

- Typical way of accessing a dictionary is by using a key inside square brackets as a way to get to the associated value.

- A dictionary cannot be accessed using the values – only via the keys.
Modifying a dictionary

- **Example:**
  >>> D = {"to": 10, "be": 20, "it": 31, "go": 20}
  >>> D["to"] = 25
  >>> D
  {'go': 20, 'to': 25, 'it': 31, 'be': 20}
  >>> D["hello"] = 100
  >>> D
  {'go': 20, 'to': 25, 'hello': 100, 'it': 31, 'be': 20}

- The value associated with a key can be modified by an assignment.
- A new key-value pair can also be added to the dictionary by an assignment.
Deleting items from a dictionary

**Example:**

```python
>>> D = {"to": 10, "be": 20, "it": 31, "go": 20}
>>> del D["be"]
>>> D
{'go': 20, 'to': 10, 'it': 31}
```

To clear all values in a dictionary, use `D.clear()`. After a dictionary is “cleared” its value is `{}`
A few dictionary functions

1. `D.keys()` returns a list with all the keys in `D`
2. `D.values()` returns a list with all the values in `D`
3. `D.items()` returns a list of key-value pairs (as tuples)
4. `key in D`, `key not in D` evaluate to boolean values depending on whether key is in `D`
5. `D.pop(key)` removes the key-value pair corresponding to `key` and returns the value
6. `D.popitem()` removes and returns an arbitrary key-value pair from `D`
7. `D.update(uD)` updates `D` using the key-value pairs in `uD`
**Problem:** We want to generate 50,000 distinct random integers in the range \([1, 100,000]\).

**ALGORITHM**
1. Start with an empty list \(L\).
2. Pick a random integer \(r\) in \([1, 100,000]\)
3. Append to \(L\) if \(r\) is not in \(L\)
4. STOP if \(\text{len}(L)\) is 50,000; otherwise return to (2).
Using Dictionaries makes a huge difference

- Step (3) in which we are repeatedly testing if r is in L is extremely inefficient if we use the in-built Python `index` function.

- In situations such as this, we should use Python dictionaries instead.

- This is even though we only care about the keys and there is no notion of associated values.

- List based implementation: 36.4886 s
- Dictionary-based implementation: 0.1781 s
def makeRandomList():
    D = {}
    while len(D) < 50000:
        r = random.randint(1, 100000)
        if r not in D:
            D[r] = 0
    return D