Dictionaries

Ben Bolker

20 October 2019

Reference; reference

Dictionaries

- An ordinary dictionary can be viewed as a **map** from the set of words in the language to their definitions.
- Some words have multiple definitions and so the value of this map for some words is a list of definitions.
- A Python dictionary (`dict`) object is a map that associates **keys** to **values**.
- A **key** of a dictionary can be any *immutable* Python object, such as a string (`str`) (like a word in a regular dictionary), a number, or a tuple.
- the **value** associated with a given key can be any Python object.
- A dictionary consists of a set of key:value pairs
- dictionaries are created using braces (`{` and `}`) or the `dict()` function
- the values associated with a given key can be accessed (looked up) using square brackets `[` and `]`.

basic dictionary setup

```
d = {"A":1,"B":2,"C":3}
empty = {}  ## empty dictionary
print(d["A"])
## d[1] won't work; no indices!
```

dictionary operations

```
## 'in' operator: does a given KEY exist in a dictionary's set of keys?
print("A" in d)
print(1 in d)           ## 1 is a value, not a key
print(d.values())       ## print all of the values
print(d.keys())         ## print all of the keys
## convert a tuple to a dictionary:
x = (("A",1),("B",2))
dict(x)
```

other dictionary operations

- dictionaries are *mutable*

- add and remove entries

```
d = {"A":1,"B":2,"C":3}
d["D"]=5     ## add an entry
del d["A"]   ## remove an entry
d.pop("C")   ## remove an entry *and return its value*
```

updating dictionaries

- **updating** adds the entries from one dictionary to another

```
d2 = {"F":5, "G":7, "H":10}
d.update(d2)
print(d)
```

the `dict()` function

- Can also create a dictionary directly via `dict()`
- **only** if keys can be represented as a Python symbol

```
dict(A=1,B=2,C=3)
```

processing a dictionary

- loop over *keys* in a dictionary:

```
d = dict(A=1,B=2,C=3)
for i in d:  ## loop over keys
    ## do something
    print(i)
```

dictionary surprises

- dictionaries occur in **arbitrary order**
  (this is completely unlike real dictionaries!)
- arbitrary order allows dictionaries to be highly efficient
  (searching, adding, subtracting)
- dictionaries are *mutable*
  (like lists and sets, unlike tuples and strings)

other dictionary machinery

- extract keys with `d.keys()`
  (a set-like object)
- `for k in d:` works about the same as `for k in d.keys():`
- extract items with `d.items()`
  a set-like object containing `(key, value)` tuples

```
for i, v in d.items():  ## unpack tuples as we go along
    ## do something
    print(i," maps to", v)
```

testing for a key/value pair

Two equivalent tests:

```
print(("A",2) in d.items())
print("A" in d and d["A"]==2)
```

dictionary inversion

- sometimes you might want to **invert** a dictionary.
- the dictionary provides a map from a set of keys to a set of values
- in the inverted version of this dictionary, the **keys** will be the values from the old dictionary and the **values** are the keys.
- if we have a simple dictionary with a one-to-one match between keys and values:

```
inv = {}         ## initialize an empty dictionary
for k in d:      ## loop over keys
   inv[d[k]] = k ## add d[k] as a key with k as its value
```

or

```
inv = {}
for k,v in d.items():
   inv[v] = k
```

more complex inversion

- suppose that `number_to_grades` is a dictionary with keys consisting of student numbers and values the (letter) grade for each student in a course
- the inverted version of this dictionary could be called `grades_to_numbers` and would have the set of (letter) grades as its keys and student numbers as its values
- in the original dictionary, each student number has a single grade associated with it
- in the inverted dictionary, there may be several students having the same grade.
- so, the *values* for the inverted dictionary would naturally be a list or a set

inverting example

- The file `grade_file.txt` contains a list of student numbers and a letter grade for each student.
- Create a dictionary called `numbers_to_grades` from this file that has the student numbers as keys and the grades as values.
- Then, invert it to create a dictionary called `grades_to_numbers`.

inversion

```
grades_file = open('grade_file.txt')
number_to_grades = {}    ## initialize the dict
for line in grades_file: ## for each line, add the pair
    number, grade = tuple(line.split())
    number_to_grades[number] = grade
grades_file.close()
## now invert
grades_to_numbers = {}   ## intialize the inverted dict
for number, grade in number_to_grades.items():
    if grade in grades_to_numbers: # old key
        grades_to_numbers[grade].append(number)
    else: # new key, so add it (as a one-element list) to the dict
        grades_to_numbers[grade] = [number]
```

revisiting Benford's Law

- use a `dict` rather than a `list` to keep track of the number of leading digits found. Remember:

```
def ben_count(file_name):
   digits_count = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
   fn = open(file_name, 'r')
   for line in fn:
       last_word = get_last_word(line)
       leading_digit = get_leading_digit(last_word)
       if leading_digit > 0:
           digits_count[leading_digit] += 1
   fn.close()
   return tuple(digits_count)I
```

replace list with a dictionary

replace the list `digits_count` with a dictionary

```
ben_dict = {} # initialize the dict.
fn = open(file_name, 'r')
```

```
for line in fn:
   last_word = get_last_word(line)
   l_d = get_leading_digit(last_word)
   if l_d > 0:
      if l_d in ben_dict: # l_d is already a key.
         ben_dict[l_d] += 1
      else: #l_d isn’t yet a key.
         ben_dict[l_d] = 1
   fn.close()
```