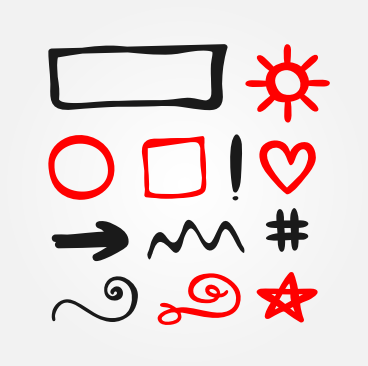

CPSC 5016—Programming I for Business

Worksheet 12

Dictionaries

Motivation

Lists were so successful in Python—a mutable collection datatype that supports:

- elements of any datatype,
- indexing, slicing, for-loop iteration, `len` built-in, `in` operator,
- literal constructions using square brackets,
- concatenation with the **+** and ***** operators, and
- modifications with append, pop, and individual assignment using index number.

Powerful, especially since they allow us to use aliasing!

One way you can think of a `list` is a *mapping function*. It maps index numbers to elements. You tell Python the index number (0, 1, 2, etc.) and it responds with the corresponding object. This is a powerful concept and one that is used extensively in computer programs. We'll see that our next built-in datatype, `dict`, abstracts this mapping feature.

Concept

The generalization of the `list` datatype allowing indexing by (almost) any datatype is the `dict` datatype. Dictionaries, like lists, are a mutable collection of objects of arbitrary datatypes. But unlike lists which only map from a zero-based integer index to the elements, dictionaries allow the indexing value, called the *key*, to be any immutable object.

As with lists where the index number uniquely specifies the "slot" in the data structure, dictionary keys each uniquely specify a certain slot for the corresponding value in the dictionary.

Note that the keys must be immutable objects, but like tuples and lists, the values have no restrictions—any object can be used. Note too that the **+** and ***** operators are unsupported.

Dictionary Literals

Think about list and tuple literals for a second. They are just comma-separated sequences. Since the position within the sequence tells Python which index number it is, there is no need to indicate that in the literal syntax. This is not the case with dictionaries which can have any key. Python needs to allow the programmer to specify both the key and the value for each element. It uses the colon to separate the key and value for each element. The whole literal is enclosed in curly brackets.

For example:

```
d = {'Bob': 12, 1: True}  # 2-element dictionary, keys 'Bob' and 1
```

The empty dictionary is just the two matching curly brackets:

```
d = {}  # empty dictionary
```

Indexing, Length, and Membership

Indexing for dictionaries works just as it does for sequenced collections like strings, tuples, and lists, but can look quite different since it uses the key instead of just an integer.

```
>>> d = {'Bob': 12, 1: True}
>>> d['Bob']
12
>>> d[1]
True
```

Also like other datatypes, indexing by a key that is not present is an error. Slicing is *not supported* by dictionaries for obvious reasons.

Another difference between the collection datatypes is their treatment of the `in` operator:

- The `str` datatype checks if the operand is a substring,
  ```
  'bc' in 'abcdefg'
  ```
- The `tuple`, `list`, and `range` datatypes check if the operand is equal to any of its *values*,
  ```
  3 in range(10)
  9.4 in [9.1, 9.2, 9.3, 9.4, 9.7]
  ```
- The `dict` datatype checks if the operand is equal to any of its *keys*,
  ```
  'Bob' in d
  1 in d
  ```

Exercise

Fill in the shell's responses:

```
>>> d = {1: 2, 3: 4, 5: 6, ('a', 'b'): [10, 9, 8]}
>>> d[5]


>>> d['a', 'b']


>>> d[len(d) - 1]
```

Modifications

Both appending an element and modifying an element are done in the same way—by using an index operation on the left-hand side of the = sign.

```
>>> d = {'Bob': 12, 1: True}
>>> d['Bob'] += 2
>>> d[10] = 'something'  # note there is no 'ordering' for new items
>>> d[1] = not d[1]
>>> d
{'Bob': 14, 10: 'something', 1: False}
```

Dictionaries elements can be deleted using the Python keyword **del**.

```
>>> big_d = {1: 2, 3: 4, 5: 6, ('a', 'b'): [10, 9, 8]}
>>> del big_d[1]
>>> big_d
{3: 4, 5: 6, ('a', 'b'): [10, 9, 8]}
```

Looping

Iterating through a dictionary, like the **in** membership operator, works on the keys, not the values. This makes sense since the dictionary can easily supply the corresponding value through indexing. For example,

```
>>> d = {'Bob': 12, 1: True}
>>> for x in d:
        print(x)
Bob
1
```

and,

```
>>> for x in d:
        print(d[x])  # get the value easily by indexing!
12
True
```

Important: There is no guaranteed order when iterating (or printing) a dictionary. The only guarantee is that the for-loop will get to every key in the dictionary once in any order it chooses. Do not rely on any expectation of order. For small dictionaries they appear to go in the order of insertion, but for larger ones that's not true and Python reserves the right to do any ordering it wants for any dictionary. It can even change from one for-loop to the next if it wants. To apply a guaranteed ordering, you can use the **sorted** built-in if all the keys in your dictionary are comparable.

```
for key in sorted(my_dictionary):
```

Exercise

Write a function that takes a dictionary as an argument and prints out the key and value for each element, one key and value per line in sorted order by key.

Lookup Pattern

Dictionaries are used in countless ways. One common pattern is to use a dictionary to look up a value based on the key. Here's an example:

```
catalog = {1220: 'Data-Driven Programming',
           5061: 'Programming I for Business',
           4600: 'Parallel Programming',
           2430: 'Data Structures and Algorithms'}
course = int(input('CPSC course number? '))
if course in catalog:
    print("That's CPSC", course, '--', catalog[course])
else
    print('Unknown course CPSC', course)
```

Exercise

Write a function, **grade_range**, that takes a string grade as an argument and returns the corresponding grade range. Use the following code inside your function:

```
grade_table = {'A': (0.9, 1.0), 'B': (0.8, 0.9), 'C': (0.7, 0.8),
               'D': (0.6, 0.7), 'F': (0.0, 0.6)}
```

Keyed Accumulator Pattern

Another common pattern is to have a collection of accumulators. Here's a function that counts the number of occurrences of each letter in a string:

```
def letter_frequency(text):
    """Return a dictionary keyed by the alphabetic letters in given
    text, counting the occurences of each letter.
    """
    letter_counts = {}
    for letter in text:
        if letter.isalpha():  # only count letters
            if letter not in letter_counts:
                letter_counts[letter] = 0  # add key to dictionary
            letter_counts[letter] += 1  # increment count for letter
    return letter_counts
```

```
>>> letter_frequency('sally sold seashells')
{'s': 5, 'a': 2, 'l': 5, 'y': 1, 'o': 1, 'd': 1, 'e': 2, 'h': 1}
```

Exercise

Modify the above function to put capital and lower-case letters in the same bucket.
For example, with your improved function we'd get this:

```
>>> letter_frequency('Sally sold seashells!')
{'S': 5, 'A': 2, 'L': 5, 'Y': 1, 'O': 1, 'D': 1, 'E': 2, 'H': 1}
```

Python Knowledge Chart:

We've learned a whole bunch of new datatypes in the last few weeks. Now is a good time to recap the new datatypes we have encountered and put them in our Python knowledge chart of datatypes.

DATATYPES				**Version 3**
Datatype	**Collection***	**Mutable**	**Literal Examples/Notes**	**Some Operators**
1. `str`	yes		`'hello'`, `"hello"`	`+ * [i] [:] len() in`
2. `int`			`1`, `0`, `193932`, `-3`, `1_000`	`+ - * // % **`
3. `float`			`-12.62`, `1.234e-6`	`+ - * / **`
4. `bool`			just two: `True False`	`not and or`
5. **NoneType**			just one: `None`	
6. `range`	yes		created with constructor	
7. file†			created with `open` built-in	
8. `function`			defined with `def` keyword	can be called
9. `tuple`	yes		`()`, `(12.6,)`, `(1, 3, 2)`	`+ * [i] [:] len() in`
10.`list`	yes	yes	`[]`, `[12.6]`, `[1, 3, 2]`	`+ * [i] [:] len() in`
11.`dict`	yes	yes	`{'Bob': 12, 1: True}`	`[key] len() in`

*Collection datatypes often support indexing, slicing, `in`, `len()`, and for-loops.
†Technically, the file datatype we are using is called a **`TextIOWrapper`**.

Remember it:

Cover up the knowledge chart above and reproduce it here from your memory:

DATATYPES				**Version 3**
Datatype	**Collection**	**Mutable**	**Literal Examples/Notes**	**Some Operators**
1.				
2.				
3.				
4.				
5.				
6.				
7.				
8.				
9.				
10.				
11.				