06-dictionaries

September 2, 2023

1 Dictionaries (Data Structure)

Dictionaries allow us to store connected bits of information. For example, you might store a person's name and age together.

2 Contents

top

3 What are dictionaries?

Dictionaries are a way to store information that is connected in some way. Dictionaries store information in *key-value* pairs, so that any one piece of information in a dictionary is connected to at least one other piece of information.

Dictionaries do not store their information in any particular order, so you may not get your information back in the same order you entered it.

3.1 General Syntax

A general dictionary in Python looks something like this:

[ ]:
```
dictionary_name = {key_1: value_1, key_2: value_2, key_3: value_3}
```

Since the keys and values in dictionaries can be long, we often write just one key-value pair on a line. You might see dictionaries that look more like this:

[ ]:
```
dictionary_name = {key_1: value_1,
                   key_2: value_2,
                   key_3: value_3,
                   }
```

This is a bit easier to read, especially if the values are long.

3.2 Example

A simple example involves modeling an actual dictionary.

[2]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }
```

We can get individual items out of the dictionary, by giving the dictionary's name, and the key in square brackets:

[7]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

print("\nWord: %s" % 'list')
print("Meaning: %s" % python_words['list'])

print("\nWord: %s" % 'dictionary')
print("Meaning: %s" % python_words['dictionary'])

print("\nWord: %s" % 'function')
print("Meaning: %s" % python_words['function'])
```

```
Word: list
Meaning: A collection of values that are not connected, but have an order.

Word: dictionary
Meaning: A collection of key-value pairs.

Word: function
Meaning: A named set of instructions that defines a set of actions in Python.
```

This code looks pretty repetitive, and it is. Dictionaries have their own for-loop syntax, but since there are two kinds of information in dictionaries, the structure is a bit more complicated than it is for lists. Here is how to use a for loop with a dictionary:

[7]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Print out the items in the dictionary.
for word, meaning in python_words.items():
    print("\nWord: %s" % word)
    print("Meaning: %s" % meaning)
```

```
Word: function
Meaning: A named set of instructions that defines a set of actions in Python.

Word: list
Meaning: A collection of values that are not connected, but have an order.

Word: dictionary
Meaning: A collection of key-value pairs.
```

The output is identical, but we did it in 3 lines instead of 6. If we had 100 terms in our dictionary, we would still be able to print them out with just 3 lines.

The only tricky part about using for loops with dictionaries is figuring out what to call those first two variables. The general syntax for this for loop is:

[ ]:
```
for key_name, value_name in dictionary_name.items():
    print(key_name) # The key is stored in whatever you called the first␣
 ↪variable.
    print(value_name) # The value associated with that key is stored in your␣
 ↪second variable.
```

3.3 Exercises

Pet Names

- Create a dictionary to hold information about pets. Each key is an animal's name, and each value is the kind of animal.
  - For example, 'ziggy': 'canary'
- Put at least 3 key-value pairs in your dictionary.
- Use a for loop to print out a series of statements such as "Willie is a dog."

Polling Friends

- Think of a question you could ask your friends. Create a dictionary where each key is a person's name, and each value is that person's response to your question.
- Store at least three responses in your dictionary.
- Use a for loop to print out a series of statements listing each person's name, and their response.

[ ]:
```
# Ex 7.1 : Pet Names

# put your code here
```

[ ]:
```
# Ex 7.2 : Polling Friends

# put your code here
```

top

4 Common operations with dictionaries

There are a few common things you will want to do with dictionaries. These include adding new key-value pairs, modifying information in the dictionary, and removing items from dictionaries.

4.1 Adding new key-value pairs

To add a new key-value pair, you give the dictionary name followed by the new key in square brackets, and set that equal to the new value. We will show this by starting with an empty dictionary, and re-creating the dictionary from the example above.

[3]:
```
# Create an empty dictionary.
python_words = {}

# Fill the dictionary, pair by pair.
python_words['list'] ='A collection of values that are not connected, but have␣
 ↪an order.'
python_words['dictionary'] = 'A collection of key-value pairs.'
python_words['function'] = 'A named set of instructions that defines a set of␣
 ↪actions in Python.'

# Print out the items in the dictionary.
```

```
for word, meaning in python_words.items():
    print("\nWord: %s" % word)
    print("Meaning: %s" % meaning)
```

```
Word: function
Meaning: A named set of instructions that defines a set of actions in Python.

Word: list
Meaning: A collection of values that are not connected, but have an order.

Word: dictionary
Meaning: A collection of key-value pairs.
```

[15]:
```
# Dealing with Hashing Functions
hashings = {}
hashings['function'] = hash('function')
hashings['list'] = hash('list')
hashings['dictionary'] = hash('dictionary')

for k, v in hashings.items():
    print(k,': ',v)


from collections import OrderedDict
hashings = OrderedDict()
hashings['function'] = hash('function')
hashings['list'] = hash('list')
hashings['dictionary'] = hash('dictionary')

for k, v in hashings.items():
    print(k,': ',v)
```

```
list :  507389742764312598
function :  -6418509839493444138
dictionary :  -4501548687734576673
function :  -6418509839493444138
list :  507389742764312598
dictionary :  -4501548687734576673
```

top

4.2 Modifying values in a dictionary

At some point you may want to modify one of the values in your dictionary. Modifying a value in a dictionary is pretty similar to modifying an element in a list. You give the name of the dictionary and then the key in square brackets, and set that equal to the new value.

[17]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

print('dictionary: ' + python_words['dictionary'])

# Clarify one of the meanings.
python_words['dictionary'] = 'A collection of key-value pairs. \
                              Each key can be used to access its corresponding␣
 ↪value.'

print('\ndictionary: ' + python_words['dictionary'])
```

```
dictionary: A collection of key-value pairs.

dictionary: A collection of key-value pairs. Each key can be used to access its
corresponding value.
```

top

4.3 Removing key-value pairs

You may want to remove some key-value pairs from one of your dictionaries at some point. You can do this using the same `del` command you learned to use with lists. To remove a key-value pair, you give the `del` command, followed by the name of the dictionary, with the key that you want to delete. This removes the key and the value as a pair.

[1]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Remove the word 'list' and its meaning.
_ = python_words.pop('list')

# Show the current set of words and meanings.
print("\n\nThese are the Python words I know:")
for word, meaning in python_words.items():
    print("\nWord: %s" % word)
    print("Meaning: %s" % meaning)
```

```
These are the Python words I know:

Word: dictionary
Meaning: A collection of key-value pairs.

Word: function
Meaning: A named set of instructions that defines a set of actions in Python.
```

If you were going to work with this code, you would certainly want to put the code for displaying the dictionary into a function. Let's see what this looks like:

[3]:
```
def show_words_meanings(python_words):
    # This function takes in a dictionary of python words and meanings,
    #  and prints out each word with its meaning.
    print("\n\nThese are the Python words I know:")
    for word, meaning in python_words.items():
        print("\nWord: %s" % word)
        print("Meaning: %s" % meaning)


python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

show_words_meanings(python_words)

# Remove the word 'list' and its meaning.
del python_words['list']

show_words_meanings(python_words)
```

```
These are the Python words I know:

Word: function
Meaning: A named set of instructions that defines a set of actions in Python.

Word: list
Meaning: A collection of values that are not connected, but have an order.

Word: dictionary
Meaning: A collection of key-value pairs.
```

```
These are the Python words I know:

Word: function
Meaning: A named set of instructions that defines a set of actions in Python.

Word: dictionary
Meaning: A collection of key-value pairs.
```

As long as we have a nice clean function to work with, let's clean up our output a little:

[1]:
```python
def show_words_meanings(python_words):
    # This function takes in a dictionary of python words and meanings,
    #  and prints out each word with its meaning.
    print("\n\nThese are the Python words I know:")
    for word, meaning in python_words.items():
        print("\n%s: %s" % (word, meaning))


python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

show_words_meanings(python_words)

# Remove the word 'list' and its meaning.
del python_words['list']

show_words_meanings(python_words)
```

```
These are the Python words I know:

function: A named set of instructions that defines a set of actions in Python.

dictionary: A collection of key-value pairs.

list: A collection of values that are not connected, but have an order.


These are the Python words I know:

function: A named set of instructions that defines a set of actions in Python.

dictionary: A collection of key-value pairs.
```

This is much more realistic code.

4.4 Modifying keys in a dictionary

Modifying a value in a dictionary was straightforward, because nothing else depends on the value. Modifying a key is a little harder, because each key is used to unlock a value. We can change a key in two steps:

- Make a new key, and copy the value to the new key.
- Delete the old key, which also deletes the old value.

Here's what this looks like. We will use a dictionary with just one key-value pair, to keep things simple.

[37]:
```
# We have a spelling mistake!
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.'}

# Create a new, correct key, and connect it to the old value.
#  Then delete the old key.
python_words['list'] = python_words['lisst']
del python_words['lisst']

# Print the dictionary, to show that the key has changed.
print(python_words)
```

```
{'list': 'A collection of values that are not connected, but have an order.'}
```

top

4.5 Exercises

Pet Names 2

- Make a copy of your program from Pet Names.
  - Use a for loop to print out a series of statements such as "Willie is a dog."
  - Modify one of the values in your dictionary. You could clarify to name a breed, or you could change an animal from a cat to a dog.
    - Use a for loop to print out a series of statements such as "Willie is a dog."
  - Add a new key-value pair to your dictionary.
    - Use a for loop to print out a series of statements such as "Willie is a dog."
  - Remove one of the key-value pairs from your dictionary.
    - Use a for loop to print out a series of statements such as "Willie is a dog."
- Bonus: Use a function to do all of the looping and printing in this problem.

Weight Lifting

- Make a dictionary where the keys are the names of weight lifting exercises, and the values are the number of times you did that exercise.
  - Use a for loop to print out a series of statements such as "I did 10 bench presses".
  - Modify one of the values in your dictionary, to represent doing more of that exercise.

        * Use a for loop to print out a series of statements such as "I did 10 bench presses".
    - Add a new key-value pair to your dictionary.
        * · Use a for loop to print out a series of statements such as "I did 10 bench presses".
    - Remove one of the key-value pairs from your dictionary.
        * · Use a for loop to print out a series of statements such as "I did 10 bench presses".
- Bonus: Use a function to do all of the looping and printing in this problem.

[ ]:
```
# Ex 7.3 : Pet Names 2

# put your code here
```

[ ]:
```
# Ex 7.4 : Weight Lifting

# put your code here
```

top

5 Looping through a dictionary

Since dictionaries are really about connecting bits of information, you will often use them in the ways described above, where you add key-value pairs whenever you receive some new information, and then you retrieve the key-value pairs that you care about. Sometimes, however, you will want to loop through the entire dictionary. There are several ways to do this:

- You can loop through all key-value pairs;
- You can loop through the keys, and pull out the values for any keys that you care about;
- You can loop through the values.

5.1 Looping through all key-value pairs

This is the kind of loop that was shown in the first example. Here's what this loop looks like, in a general format:

[8]:
```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for key, value in my_dict.items():
    print('\nKey: %s' % key)
    print('Value: %s' % value)
```

```
Key: key_1
Value: value_1

Key: key_3
Value: value_3
```

```
Key: key_2
Value: value_2
```

This works because the method `.items()` pulls all key-value pairs from a dictionary into a list of tuples:

[10]:
```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

print(my_dict.items())
```

```
[('key_1', 'value_1'), ('key_3', 'value_3'), ('key_2', 'value_2')]
```

The syntax `for key, value in my_dict.items():` does the work of looping through this list of tuples, and pulling the first and second item from each tuple for us.

There is nothing special about any of these variable names, so Python code that uses this syntax becomes really readable. Rather than create a new example of this loop, let's just look at the original example again to see this in a meaningful context:

[11]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

for word, meaning in python_words.items():
    print("\nWord: %s" % word)
    print("Meaning: %s" % meaning)
```

```
Word: function
Meaning: A named set of instructions that defines a set of actions in Python.

Word: list
Meaning: A collection of values that are not connected, but have an order.

Word: dictionary
Meaning: A collection of key-value pairs.
```

top

5.2 Looping through all keys in a dictionary

Python provides a clear syntax for looping through just the keys in a dictionary:

[13]:
```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for key in my_dict.keys():
    print('Key: %s' % key)
```

```
Key: key_1
Key: key_3
Key: key_2
```

This is actually the default behavior of looping through the dictionary itself. So you can leave out the `.keys()` part, and get the exact same behavior:

[14]:
```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for key in my_dict:
    print('Key: %s' % key)
```

```
Key: key_1
Key: key_3
Key: key_2
```

The only advantage of using the `.keys()` in the code is a little bit of clarity. But anyone who knows Python reasonably well is going to recognize what the second version does. In the rest of our code, we will leave out the `.keys()` when we want this behavior.

You can pull out the value of any key that you are interested in within your loop, using the standard notation for accessing a dictionary value from a key:

[17]:
```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for key in my_dict:
    print('Key: %s' % key)
    if key == 'key_2':
        print("  The value for key_2 is %s." % my_dict[key])
```

```
Key: key_1
Key: key_3
Key: key_2
  The value for key_2 is value_2.
```

Let's show how we might use this in our Python words program. This kind of loop provides a

straightforward way to show only the words in the dictionary:

[20]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Show the words that are currently in the dictionary.
print("The following Python words have been defined:")
for word in python_words:
    print("- %s" % word)
```

```
The following Python words have been defined:
- function
- list
- dictionary
```

We can extend this slightly to make a program that lets you look up words. We first let the user choose a word. When the user has chosen a word, we get the meaning for that word, and display it:

[2]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Show the words that are currently in the dictionary.
print("The following Python words have been defined:")
for word in python_words:
    print("- %s" % word)

# Allow the user to choose a word, and then display the meaning for that word.
requested_word = raw_input("\nWhat word would you like to learn about? ")
print("\n%s: %s" % (requested_word, python_words[requested_word]))
```

```
The following Python words have been defined:
- function
- list
- dictionary

What word would you like to learn about? list

list: A collection of values that are not connected, but have an order.
```

This allows the user to select one word that has been defined. If we enclose the input part of the

program in a while loop, the user can see as many definitions as they'd like:

[4]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Show the words that are currently in the dictionary.
print("The following Python words have been defined:")
for word in python_words:
    print("- %s" % word)

requested_word = ''
while requested_word != 'quit':
    # Allow the user to choose a word, and then display the meaning for that␣
 ↪word.
    requested_word = raw_input("\nWhat word would you like to learn about? (or␣
 ↪'quit') ")
    if requested_word in python_words.keys():
        print("\n  %s: %s" % (requested_word, python_words[requested_word]))
    else:
        # Handle misspellings, and words not yet stored.
        print("\n  Sorry, I don't know that word.")
```

```
The following Python words have been defined:
- function
- list
- dictionary

What word would you like to learn about? (or 'quit') list

  list: A collection of values that are not connected, but have an order.

What word would you like to learn about? (or 'quit') dictionary

  dictionary: A collection of key-value pairs.

What word would you like to learn about? (or 'quit') quit

  Sorry, I don't know that word.
```

This allows the user to ask for as many meanings as they want, but it takes the word "quit" as a requested word. Let's add an `elif` clause to clean up this behavior:

[6]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
```

```
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Show the words that are currently in the dictionary.
print("The following Python words have been defined:")
for word in python_words:
    print("- %s" % word)

requested_word = ''
while requested_word != 'quit':
    # Allow the user to choose a word, and then display the meaning for that␣
 ↪word.
    requested_word = raw_input("\nWhat word would you like to learn about? (or␣
 ↪'quit') ")
    if requested_word in python_words.keys():
        # This is a word we know, so show the meaning.
        print("\n  %s: %s" % (requested_word, python_words[requested_word]))
    elif requested_word != 'quit':
        # This is not in python_words, and it's not 'quit'.
        print("\n  Sorry, I don't know that word.")
    else:
        # The word is quit.
        print "\n  Bye!"
```

```
The following Python words have been defined:
- function
- list
- dictionary

What word would you like to learn about? (or 'quit') function

  function: A named set of instructions that defines a set of actions in Python.

What word would you like to learn about? (or 'quit') dictionary

  dictionary: A collection of key-value pairs.

What word would you like to learn about? (or 'quit') list

  list: A collection of values that are not connected, but have an order.

What word would you like to learn about? (or 'quit') class

  Sorry, I don't know that word.
```

```
What word would you like to learn about? (or 'quit') quit

  Bye!
```

top

5.3 Looping through all values in a dictionary

Python provides a straightforward syntax for looping through all the values in a dictionary, as well:

[15]:
```
my_dict = {'key_1': 'value_1',
    'key_2': 'value_2',
    'key_3': 'value_3',
    }

for value in my_dict.values():
    print('Value: %s' % value)
```

```
Value: value_1
Value: value_3
Value: value_2
```

We can use this loop syntax to have a little fun with the dictionary example, by making a little quiz program. The program will display a meaning, and ask the user to guess the word that matches that meaning. Let's start out by showing all the meanings in the dictionary:

[16]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

for meaning in python_words.values():
    print("Meaning: %s" % meaning)
```

```
Meaning: A named set of instructions that defines a set of actions in Python.
Meaning: A collection of values that are not connected, but have an order.
Meaning: A collection of key-value pairs.
```

Now we can add a prompt after each meaning, asking the user to guess the word:

[2]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Print each meaning, one at a time, and ask the user
```

```
#  what word they think it is.
for meaning in python_words.values():
    print("\nMeaning: %s" % meaning)

    guessed_word = raw_input("What word do you think this is? ")

    # The guess is correct if the guessed word's meaning matches the current␣
 ↪meaning.
    if python_words[guessed_word] == meaning:
        print("You got it!")
    else:
        print("Sorry, that's just not the right word.")
```

```
Meaning: A named set of instructions that defines a set of actions in Python.
What word do you think this is? function
You got it!

Meaning: A collection of values that are not connected, but have an order.
What word do you think this is? function
Sorry, that's just not the right word.

Meaning: A collection of key-value pairs.
What word do you think this is? dictionary
You got it!
```

This is starting to work, but we can see from the output that the user does not get the chance to take a second guess if they guess wrong for any meaning. We can use a while loop around the guessing code, to let the user guess until they get it right:

[20]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

# Print each meaning, one at a time, and ask the user
#  what word they think it is.
for meaning in python_words.values():
    print("\nMeaning: %s" % meaning)

    # Assume the guess is not correct; keep guessing until correct.
    correct = False
    while not correct:
        guessed_word = input("\nWhat word do you think this is? ")
```

```
        # The guess is correct if the guessed word's meaning matches the␣
 ↪current meaning.
        if python_words[guessed_word] == meaning:
            print("You got it!")
            correct = True
        else:
            print("Sorry, that's just not the right word.")
```

```
Meaning: A named set of instructions that defines a set of actions in Python.

What word do you think this is? function
You got it!

Meaning: A collection of values that are not connected, but have an order.

What word do you think this is? dictionary
Sorry, that's just not the right word.

What word do you think this is? list
You got it!

Meaning: A collection of key-value pairs.

What word do you think this is? dictionary
You got it!
```

This is better. Now, if the guess is incorrect, the user is caught in a loop that they can only exit by guessing correctly. The final revision to this code is to show the user a list of words to choose from when they are asked to guess:

[8]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

def show_words(python_words):
    # A simple function to show the words in the dictionary.
    display_message = ""
    for word in python_words.keys():
        display_message += word + '  '
    print display_message

# Print each meaning, one at a time, and ask the user
#  what word they think it is.
for meaning in python_words.values():
```

```
    print("\n%s" % meaning)

    # Assume the guess is not correct; keep guessing until correct.
    correct = False
    while not correct:

        print("\nWhat word do you think this is?")
        show_words(python_words)
        guessed_word = raw_input("- ")

        # The guess is correct if the guessed word's meaning matches the␣
↪current meaning.
        if python_words[guessed_word] == meaning:
            print("You got it!")
            correct = True
        else:
            print("Sorry, that's just not the right word.")
```

```
A named set of instructions that defines a set of actions in Python.

What word do you think this is?
function  list  dictionary
- function
You got it!

A collection of values that are not connected, but have an order.

What word do you think this is?
function  list  dictionary
- dictionary
Sorry, that's just not the right word.

What word do you think this is?
function  list  dictionary
- list
You got it!

A collection of key-value pairs.

What word do you think this is?
function  list  dictionary
- dictionary
You got it!
```

top

6 Looping through a dictionary in order

Dictionaries are quite useful because they allow bits of information to be connected. One of the problems with dictionaries, however, is that they are not stored in any particular order. When you retrieve all of the keys or values in your dictionary, you can't be sure what order you will get them back. There is a quick and easy way to do this, however, when you want them in a particular order.

Let's take a look at the order that results from a simple call to *dictionary.keys()*:

[2]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

for word in python_words.keys():
    print(word)
```

```
function
list
dictionary
```

The resulting list is not in order. The list of keys can be put in order by passing the list into the *sorted()* function, in the line that initiates the for loop:

[3]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }

for word in sorted(python_words.keys()):
    print(word)
```

```
dictionary
function
list
```

This approach can be used to work with the keys and values in order. For example, the words and meanings can be printed in alphabetical order by word:

[8]:
```
python_words = {'list': 'A collection of values that are not connected, but␣
 ↪have an order.',
                'dictionary': 'A collection of key-value pairs.',
                'function': 'A named set of instructions that defines a set of␣
 ↪actions in Python.',
                }
```

```
for word in sorted(python_words.keys()):
    print("%s: %s" % (word.title(), python_words[word]))
```

```
Dictionary: A collection of key-value pairs.
Function: A named set of instructions that defines a set of actions in Python.
List: A collection of values that are not connected, but have an order.
```

In this example, the keys have been put into alphabetical order in the for loop only; Python has not changed the way the dictionary is stored at all. So the next time the dictionary is accessed, the keys could be returned in any order. There is no way to permanently specify an order for the items in an ordinary dictionary, but if you want to do this you can use the OrderedDict structure.

top

6.1 Exercises

Mountain Heights

- Wikipedia has a list of the tallest mountains in the world, with each mountain's elevation. Pick five mountains from this list.
  - Create a dictionary with the mountain names as keys, and the elevations as values.
  - Print out just the mountains' names, by looping through the keys of your dictionary.
  - Print out just the mountains' elevations, by looping through the values of your dictionary.
  - Print out a series of statements telling how tall each mountain is: "Everest is 8848 meters tall."
- Revise your output, if necessary.
  - Make sure there is an introductory sentence describing the output for each loop you write.
  - Make sure there is a blank line between each group of statements.

Mountain Heights 2

- Revise your final output from Mountain Heights, so that the information is listed in alphabetical order by each mountain's name.
  - That is, print out a series of statements telling how tall each mountain is: "Everest is 8848 meters tall."
  - Make sure your output is in alphabetical order.

[ ]:
```
# Ex 7.5 : Mountain Heights

# put your code here
```

[ ]:
```
# Ex 7.6 : Mountain Heights 2

# put your code here
```

top

7 Nesting

Nesting is one of the most powerful concepts we have come to so far. Nesting involves putting a list or dictionary inside another list or dictionary. We will look at two examples here, lists inside of a dictionary and dictionaries inside of a dictionary. With nesting, the kind of information we can model in our programs is expanded greatly.

7.1 Lists in a dictionary

A dictionary connects two pieces of information. Those two pieces of information can be any kind of data structure in Python. Let's keep using strings for our keys, but let's try giving a list as a value.

The first example will involve storing a number of people's favorite numbers. The keys consist of people's names, and the values are lists of each person's favorite numbers. In this first example, we will access each person's list one at a time.

[20]:
```
# This program stores people's favorite numbers, and displays them.
favorite_numbers = {'eric': [3, 11, 19, 23, 42],
                    'ever': [2, 4, 5],
                    'willie': [5, 35, 120],
                    }

# Display each person's favorite numbers.
print("Eric's favorite numbers are:")
print(favorite_numbers['eric'])

print("\nEver's favorite numbers are:")
print(favorite_numbers['ever'])

print("\nWillie's favorite numbers are:")
print(favorite_numbers['willie'])
```

```
Eric's favorite numbers are:
[3, 11, 19, 23, 42]

Ever's favorite numbers are:
[2, 4, 5]

Willie's favorite numbers are:
[5, 35, 120]
```

We are really just working our way through each key in the dictionary, so let's use a for loop to go through the keys in the dictionary:

[21]:
```
# Example for a Sparse Matrix Implementation

sparse_matrix = {}
sparse_matrix[0] = {1: 12.3, 23: 25.5}
```

```
sparse_matrix[1] = {3: 12.0, 15: 25.5}

sparse_matrix[1][15]

full_matrix = [[1, 3, 4],
               [2, 5, 3]]

full_matrix[1][2]
```

[21]: 3

[7]:
```
# This program stores people's favorite numbers, and displays them.
favorite_numbers = {'eric': [3, 11, 19, 23, 42],
                    'ever': [2, 4, 5],
                    'willie': [5, 35, 120],
                    }

# Display each person's favorite numbers.
for name in favorite_numbers:
    print("\n%s's favorite numbers are:" % name.title())
    print(favorite_numbers[name])
```

```
Willie's favorite numbers are:
[5, 35, 120]

Ever's favorite numbers are:
[2, 4, 5]

Eric's favorite numbers are:
[3, 11, 19, 23, 42]
```

This structure is fairly complex, so don't worry if it takes a while for things to sink in. The dictionary itself probably makes sense; each person is connected to a list of their favorite numbers.

This works, but we'd rather not print raw Python in our output. Let's use a for loop to print the favorite numbers individually, rather than in a Python list.

[13]:
```
# This program stores people's favorite numbers, and displays them.
favorite_numbers = {'eric': [3, 11, 19, 23, 42],
                    'ever': [2, 4, 5],
                    'willie': [5, 35, 120],
                    }

# Display each person's favorite numbers.
for name in favorite_numbers:
    print("\n%s's favorite numbers are:" % name.title())
    # Each value is itself a list, so we need another for loop
```

```
    #  to work with the list.
    for favorite_number in favorite_numbers[name]:
        print(favorite_number)
```

```
Willie's favorite numbers are:
5
35
120

Ever's favorite numbers are:
2
4
5

Eric's favorite numbers are:
3
11
19
23
42
```

Things get a little more complicated inside the for loop. The value is a list of favorite numbers, so the for loop pulls each *favorite_number* out of the list one at a time. If it makes more sense to you, you are free to store the list in a new variable, and use that to define your for loop:

[15]:
```
# This program stores people's favorite numbers, and displays them.
favorite_numbers = {'eric': [3, 11, 19, 23, 42],
                    'ever': [2, 4, 5],
                    'willie': [5, 35, 120],
                    }

# Display each person's favorite numbers.
for name in favorite_numbers:
    print("\n%s's favorite numbers are:" % name.title())

    # Each value is itself a list, so let's put that list in a variable.
    current_favorite_numbers = favorite_numbers[name]
    for favorite_number in current_favorite_numbers:
        print(favorite_number)
```

```
Willie's favorite numbers are:
5
35
120

Ever's favorite numbers are:
```

```
2
4
5

Eric's favorite numbers are:
3
11
19
23
42
```

top

7.2 Dictionaries in a dictionary

The most powerful nesting concept we will cover right now is nesting a dictionary inside of a dictionary.

To demonstrate this, let's make a dictionary of pets, with some information about each pet. The keys for this dictionary will consist of the pet's name. The values will include information such as the kind of animal, the owner, and whether the pet has been vaccinated.

[24]:
```
# This program stores information about pets. For each pet,
#   we store the kind of animal, the owner's name, and
#   the breed.
pets = {'willie': {'kind': 'dog', 'owner': 'eric', 'vaccinated': True},
        'walter': {'kind': 'cockroach', 'owner': 'eric', 'vaccinated': False},
        'peso': {'kind': 'dog', 'owner': 'chloe', 'vaccinated': True},
        }

# Let's show all the information for each pet.
print("Here is what I know about Willie:")
print("kind: " + pets['willie']['kind'])
print("owner: " + pets['willie']['owner'])
print("vaccinated: " + str(pets['willie']['vaccinated']))

print("\nHere is what I know about Walter:")
print("kind: " + pets['walter']['kind'])
print("owner: " + pets['walter']['owner'])
print("vaccinated: " + str(pets['walter']['vaccinated']))

print("\nHere is what I know about Peso:")
print("kind: " + pets['peso']['kind'])
print("owner: " + pets['peso']['owner'])
print("vaccinated: " + str(pets['peso']['vaccinated']))
```

```
Here is what I know about Willie:
kind: dog
owner: eric
```

```
vaccinated: True

Here is what I know about Walter:
kind: cockroach
owner: eric
vaccinated: False

Here is what I know about Peso:
kind: dog
owner: chloe
vaccinated: True
```

Clearly this is some repetitive code, but it shows exactly how we access information in a nested dictionary. In the first set of `print` statements, we use the name 'willie' to unlock the 'kind' of animal he is, the 'owner' he has, and whether or not he is 'vaccinated'. We have to wrap the vaccination value in the `str` function so that Python knows we want the words 'True' and 'False', not the values `True` and `False`. We then do the same thing for each animal.

Let's rewrite this program, using a for loop to go through the dictionary's keys:

[12]:
```
# This program stores information about pets. For each pet,
#   we store the kind of animal, the owner's name, and
#   the breed.
pets = {'willie': {'kind': 'dog', 'owner': 'eric', 'vaccinated': True},
        'walter': {'kind': 'cockroach', 'owner': 'eric', 'vaccinated': False},
        'peso': {'kind': 'dog', 'owner': 'chloe', 'vaccinated': True},
        }

# Let's show all the information for each pet.
for pet_name, pet_information in pets.items():
    print("\nHere is what I know about %s:" % pet_name.title())
    print("kind: " + pet_information['kind'])
    print("owner: " + pet_information['owner'])
    print("vaccinated: " + str(pet_information['vaccinated']))
```

```
Here is what I know about Peso:
kind: dog
owner: chloe
vaccinated: True

Here is what I know about Willie:
kind: dog
owner: eric
vaccinated: True

Here is what I know about Walter:
kind: cockroach
owner: eric
```

```
vaccinated: False
```

This code is much shorter and easier to maintain. But even this code will not keep up with our dictionary. If we add more information to the dictionary later, we will have to update our print statements. Let's put a second for loop inside the first loop in order to run through all the information about each pet:

[13]:
```
# This program stores information about pets. For each pet,
#   we store the kind of animal, the owner's name, and
#   the breed.
pets = {'willie': {'kind': 'dog', 'owner': 'eric', 'vaccinated': True},
        'walter': {'kind': 'cockroach', 'owner': 'eric', 'vaccinated': False},
        'peso': {'kind': 'dog', 'owner': 'chloe', 'vaccinated': True},
        }

# Let's show all the information for each pet.
for pet_name, pet_information in pets.items():
    print("\nHere is what I know about %s:" % pet_name.title())
    # Each animal's dictionary is in 'information'
    for key in pet_information:
        print(key + ": " + str(pet_information[key]))
```

```
Here is what I know about Peso:
owner: chloe
kind: dog
vaccinated: True

Here is what I know about Willie:
owner: eric
kind: dog
vaccinated: True

Here is what I know about Walter:
owner: eric
kind: cockroach
vaccinated: False
```

This nested loop can look pretty complicated, so again, don't worry if it doesn't make sense for a while.

- The first loop gives us all the keys in the main dictionary, which consist of the name of each pet.
- Each of these names can be used to 'unlock' the dictionary of each pet.
- The inner loop goes through the dictionary for that individual pet, and pulls out all of the keys in that individual pet's dictionary.
- We print the key, which tells us the kind of information we are about to see, and the value for that key.
- You can see that we could improve the formatting in the output.

- We could capitalize the owner's name.
- We could print 'yes' or 'no', instead of True and False.

Let's show one last version that uses some if statements to clean up our data for printing:

[3]:
```
# This program stores information about pets. For each pet,
#   we store the kind of animal, the owner's name, and
#   the breed.
pets = {'willie': {'kind': 'dog', 'owner': 'eric', 'vaccinated': True},
        'walter': {'kind': 'cockroach', 'owner': 'eric', 'vaccinated': False},
        'peso': {'kind': 'dog', 'owner': 'chloe', 'vaccinated': True},
        }

# Let's show all the information for each pet.
for pet_name, pet_information in pets.items():
    print("\nHere is what I know about %s:" % pet_name.title())
    # Each animal's dictionary is in pet_information
    for key in pet_information:
        if key == 'owner':
            # Capitalize the owner's name.
            print(key + ": " + pet_information[key].title())
        elif key == 'vaccinated':
            # Print 'yes' for True, and 'no' for False.
            vaccinated = pet_information['vaccinated']
            if vaccinated:
                print('vaccinated: yes')
            else:
                print('vaccinated: no')
        else:
            # No special formatting needed for this key.
            print(key + ": " + pet_information[key])
```

```
Here is what I know about Peso:
kind: dog
vaccinated: yes
owner: Chloe

Here is what I know about Walter:
kind: cockroach
vaccinated: no
owner: Eric

Here is what I know about Willie:
kind: dog
vaccinated: yes
owner: Eric
```

This code is a lot longer, and now we have nested if statements as well as nested for loops. But

keep in mind, this structure would work if there were 1000 pets in our dictionary, and it would work if we were storing 1000 pieces of information about each pet. One level of nesting lets us model an incredible array of information.

7.3 An important note about nesting

While one level of nesting is really useful, nesting much deeper than that gets really complicated, really quickly. There are other structures such as classes which can be even more useful for modeling information. In addition to this, we can use Python to store information in a database, which is the proper tool for storing deeply nested information.

Often times when you are storing information in a database you will pull a small set of that information out and put it into a dictionary, or a slightly nested structure, and then work with it. But you will rarely, if ever, work with Python data structures nested more than one level deep.

7.4 Exercises

Mountain Heights 3

- This is an extension of Mountain Heights. Make sure you save this program under a different filename, such as *mountain_heights_3.py*, so that you can go back to your original program if you need to.
  - The list of tallest mountains in the world provided all elevations in meters. Convert each of these elevations to feet, given that a meter is approximately 3.28 feet. You can do these calculations by hand at this point.
  - Create a new dictionary, where the keys of the dictionary are still the mountains' names. This time however, the values of the dictionary should be a list of each mountain's elevation in meters, and then in feet: {'everest': [8848, 29029]}
  - Print out just the mountains' names, by looping through the keys of your dictionary.
  - Print out just the mountains' elevations in meters, by looping through the values of your dictionary and pulling out the first number from each list.
  - Print out just the mountains' elevations in feet, by looping through the values of your dictionary and pulling out the second number from each list.
  - Print out a series of statements telling how tall each mountain is: "Everest is 8848 meters tall, or 29029 feet."
- Bonus:
  - Start with your original program from Mountain Heights. Write a function that reads through the elevations in meters, and returns a list of elevations in feet. Use this list to create the nested dictionary described above.

Mountain Heights 4

- This is one more extension of Mountain Heights.
  - Create a new dictionary, where the keys of the dictionary are once again the mountains' names. This time, the values of the dictionary are another dictionary. This dictionary should contain the elevation in either meters or feet, and the range that contains the mountain. For example: {'everest': {'elevation': 8848, 'range': 'himalaya'}}.
  - Print out just the mountains' names.
  - Print out just the mountains' elevations.
  - Print out just the range for each mountain.

- Print out a series of statements that say everything you know about each mountain: "Everest is an 8848-meter tall mountain in the Himalaya range."

[ ]:
```
# Ex 7.7 : Mountain Heights 3

# put your code here
```

[ ]:
```
# Ex 7.8 : Mountain Heights 4

# put your code here
```

top

8 Overall Challenges

Word Wall

- A word wall is a place on your wall where you keep track of the new words and meanings you are learning. Write a terminal app that lets you enter new words, and a meaning for each word.
  - Your app should have a title bar that says the name of your program.
  - Your program should give users the option to see all words and meanings that have been entered so far.
  - Your program should give users the option to enter a new word and meaning.
    - Your program must not allow duplicate entries.
  - Your program should store existing words and meanings, even after the program closes.
  - Your program should give users the option to modify an existing meaning.
- Bonus Features
  - Allow users to modify the spelling of words.
  - Allow users to categorize words.
  - Turn the program into a game that quizzes users on words and meanings.
  - (later on) Turn your program into a website that only you can use.
  - (later on) Turn your program into a website that anyone can register for, and use.
  - Add a visualization feature that reports on some statistics about the words and meanings that have been entered.

Periodic Table App

- The Periodic Table of the Elements was developed to organize information about the elements that make up the Universe. Write a terminal app that lets you enter information about each element in the periodic table.
  - Make sure you include the following information:
    - symbol, name, atomic number, row, and column
  - Choose at least one other piece of information to include in your app.
  - Provide a menu of options for users to:
    - See all the information that is stored about any element, by entering that element's symbol.
    - Choose a property, and see that property for each element in the table.

- Bonus Features
  - Provide an option to view the symbols arranged like the periodic table. (hint)

[ ]:
```
# Challenge: Word Wall

# put your code here
```

[ ]:
```
# Challenge: Periodic Table App

# put your code here
```

top

9 Hints

Periodic Table App

- You can use a for loop to loop through each element. Pick out the elements' row numbers and column numbers.
- Use two nested for loops to print either an element's symbol or a series of spaces, depending on how full that row is.