08 Prepare: Dictionaries

In this lesson, you will learn how to store data in and retrieve data from Python dictionaries. You will also learn how to write a loop that processes all the items in a dictionary.

Video

Watch this video about dictionaries in Python.

Dictionaries (6 minutes)

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson:

Dictionaries

A Python program can store many items in a **dictionary**. Each **item** in a dictionary is a key value pair. Each **key** within a dictionary must be unique. In other words, no key can appear more than once in a dictionary. Values within a dictionary do not have to be unique. Dictionaries are mutable, meaning they can be changed after they are created. Dictionaries were invented to enable computers to find items quickly.

The following table represents a dictionary that contains five items (five key value pairs) Notice that each of the keys is unique.

	items	
keys	**values**	
42-039-4736	Clint Huish	↑
61-315-0160	Amelia Davis	|
10-450-1203	Ana Soares	5 items
75-421-2310	Abdul Ali	|
07-103-5621	Amelia Davis	↓

We can create a dictionary by using curly braces ({ and }). We can add an item to a dictionary and find an item in a dictionary by using square brackets ([and]) and a key. The following code example shows how to create a dictionary, add an item, remove an item, and find an item in a dictionary.

```
1  # Example 1
2
3  def main():
4      # Create a dictionary with student IDs as
```

```
 5      # the keys and student names as the values.
 6      students = {
 7          "42-039-4736": "Clint Huish",
 8          "61-315-0160": "Amelia Davis",
 9          "10-450-1203": "Ana Soares",
10          "75-421-2310": "Abdul Ali",
11          "07-103-5621": "Amelia Davis"
12      }
13
14      # Add an item to the dictionary.
15      students["81-298-9238"] = "Sama Patel"
16
17      # Remove an item from the dictionary.
18      students.pop("61-315-0160")
19
20      # Get the number of items in the dictionary.
21      length = len(students)
22      print(f"length: {length}")
23
24      # Print the entire dictionary.
25      print(students)
26      print()
27
28      # Get a student ID from the user.
29      id = input("Enter a student ID: ")
30
31      # Check if the student ID is in the dictionary.
32      if id in students:
33
34          # Find the student ID in the dictionary and
35          # retrieve the corresponding student name.
36          name = students[id]
37
38          # Print the student's name.
39          print(name)
40
41      else:
42          print("No such student")
43
44
45  # Call main to start this program.
46  if __name__ == "__main__":
47      main()
```

```
> python example_1.py
length: 5
{'42-039-4736': 'Clint Huish', '10-450-1203': 'Ana Soares',
'75-421-2310': 'Abdul Ali', '07-103-5621': 'Amelia Davis',
'81-298-9238': 'Sama Patel'}

Enter a student ID: 10-450-1203
Ana Soares
```

Line 15 in the previous code example, adds an item to the dictionary. To add an item to an existing dictionary, write code that follows this template:

```
    dictionary_name[key] = value
```

Notice that line 15 follows this template.

Line 32 in the previous code example, uses the Python membership operator, which is the keyword `in`, to check if a key is stored in a dictionary. To check if a

key is stored in a dictionary, write code that follows this template:

```
    if key in dictionary_name:
```

Notice that line 32 follows this template.

Line 36 in the previous code example, finds a key and retrieves its corresponding value from a dictionary. To find a key and retrieve its corresponding value, write code that follows this template:

```
    value = dictionary_name[key]
```

Notice that line 36 follows this template.

Compound Values

A **simple value** is a value that doesn't contain parts, such as an integer. A **compound value** is a value that has parts, such as a list. In example 1 above, the *students* dictionary has simple keys and values. Each key is a single string, and each value is a single string. It is possible to store compound values in a dictionary. Example 2 shows a *students* dictionary where each value is a Python list. Because each list contains multiple parts, we say that the dictionary stores compound values.

```
# Example 2

def main():
    # Create a dictionary with student IDs as the keys
    # and student data stored in a list as the values.
    students = {
        # student_ID: [given_name, surname, email_address, credits]
        "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
        "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
        "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
        "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
        "07-103-5621": ["Amelia", "Davis" "dav19008@byui.edu", 0]
    }
```

Finding One Item

The reason Python dictionaries were developed is to make finding items easy and fast. As explained in example 1, to find an item in a dictionary, a programmer needs to write just one line of code that follows this template:

```
    value = dictionary_name[key]
```

That one line of code will cause the computer to search the dictionary until it finds the *key*. Then the computer will return the *value* that corresponds to the *key*. Some students forget how easy it is to find items in a dictionary, and when asked to write code to find an item, they write complex code like lines 24–28 in example 3.

```
 1  # Example 3
 2
 3  def main():
 4      # Create a dictionary with student IDs as the keys
 5      # and student data stored in a list as the values.
 6      students = {
 7          # student_ID: [given_name, surname, email_address, credits]
 8          "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
 9          "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
10          "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
11          "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
12          "07-103-5621": ["Amelia", "Davis" "dav19008@byui.edu", 0]
13      }
14
15      # Get a student ID from the user.
16      id = input("Enter a student ID: ")
17
18      # This is a difficult and slow way to find an item in a
19      # dictionary. Don't write code like this to find an item
20      # in a dictionary!
21
22      # For each item in the dictionary, check if
23      # its key is the same as the variable id.
24      student = None
25      for key, value in students.items(): # Bad example!
26          if key == id:                   # Don't use a loop like
27              student = value             # this to find an item
28              break                       # in a dictionary.
29
```

Compare the `for` loop at lines 24–28 in the previous example to this one line of code.

```
    value = students[id]
```

Clearly, writing one line of code is easier for a programmer than writing the `for` loop. Not only is the one line of code easier to write, but the computer will execute it much, much faster than the `for` loop. Therefore, when you need to write code to find an item in a dictionary, don't write a loop. Instead, write one line of code that uses the square brackets ([and]) and a key to find an item. Example 4 shows the correct way to find an item in a dictionary.

```
 1  # Example 4
 2
 3  def main():
 4      # Create a dictionary with student IDs as the keys
 5      # and student data stored in a list as the values.
 6      students = {
 7          # student_ID: [given_name, surname, email_address, credits]
 8          "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
 9          "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
10          "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
11          "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
12          "07-103-5621": ["Amelia", "Davis" "dav19008@byui.edu", 0]
13      }
14
15      # These are the indexes of the elements in the value lists.
16      GIVEN_NAME_INDEX = 0
17      SURNAME_INDEX = 1
18      EMAIL_INDEX = 2
19      CREDITS_INDEX = 3
20
```

```
    # Get a student ID from the user.
    id = input("Enter a student ID: ")

    # Check if the student ID is in the dictionary.
    if id in students:

        # Find the student ID in the dictionary and
        # retrieve the corresponding value, which is a list.
        value = students[id]

        # Retrieve the student's given name (first name) and
        # surname (last name or family name) from the list.
        given_name = value[GIVEN_NAME_INDEX]
        surname = value[SURNAME_INDEX]

        # Print the student's name.
        print(f"{given_name} {surname}")

    else:
        print("No such student")


# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_4.py
Enter a student ID: 61-315-0160
Amelia Davis

> python example_4.py
Enter a student ID: 25-143-1202
No such student
```

Processing All Items

Occasionally, you may need to write a program that processes all the items in a dictionary. Processing all the items in a dictionary is different than finding one item in a dictionary. Processing all the items is done using a `for` loop and the `dict.items()` method as shown in example 5 on line 25.

```
# Example 5

def main():
    # Create a dictionary with student IDs as the keys
    # and student data stored in a list as the values.
    students = {
        "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
        "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
        "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
        "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
        "07-103-5621": ["Amelia", "Davis", "dav19008@byui.edu", 0],
        "81-298-9238": ["Sama", "Patel", "pat21004@byui.edu", 8]
    }

    # These are the indexes of the elements in the value lists.
    GIVEN_NAME_INDEX = 0
    SURNAME_INDEX = 1
    EMAIL_INDEX = 2
    CREDITS_INDEX = 3

    total = 0
```

```
22
23      # For each item in the list add the number
24      # of credits that the student has earned.
25      for item in students.items():
26          key = item[0]
27          value = item[1]
28
29          # Retrieve the number of credits from the value list.
30          credits = value[CREDITS_INDEX]
31
32          # Add the number of credits to the total.
33          total += credits
34
35      print(f"Total credits earned by all students: {total}")
36
37
38  # Call main to start this program.
39  if __name__ == "__main__":
40      main()
```

```
> python example_5.py
Total credits earned by all students: 47
```

As with all the example code in CSE 111, example 5 contains working Python code. Even though the code works, we can combine lines 25–27 into a single line of code by using a Python shortcut called unpacking. Instead of writing lines 25–27, like this:

```
    for item in students.items():
        key = item[0]
        value = item[1]
```

We can write one line of code that combines the three lines of code and unpacks the item in the `for` statement like this:

```
    for key, value in students.items():
```

Example 6 contains the same code as example 5 except example 6 uses the Python unpacking shortcut at line 25.

```
 1  # Example 6
 2
 3  def main():
 4      # Create a dictionary with student IDs as the keys
 5      # and student data stored in a list as the values.
 6      students = {
 7          "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
 8          "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
 9          "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
10          "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
11          "07-103-5621": ["Amelia", "Davis", "dav19008@byui.edu", 0],
12          "81-298-9238": ["Sama", "Patel", "pat21004@byui.edu", 8]
13      }
14
15      # These are the indexes of the elements in the value lists.
16      GIVEN_NAME_INDEX = 0
17      SURNAME_INDEX = 1
18      EMAIL_INDEX = 2
19      CREDITS_INDEX = 3
20
21      total = 0
```

```
22
23      # For each item in the list add the number
24      # of credits that the student has earned.
25      for key, value in students.items():
26
27          # Retrieve the number of credits from the value list.
28          credits = value[CREDITS_INDEX]
29
30          # Add the number of credits to the total.
31          total += credits
32
33      print(f"Total credits earned by all students: {total}")
34
35
36  # Call main to start this program.
37  if __name__ == "__main__":
38      main()
```

```
> python example_6.py
Total credits earned by all students: 47
```

Dictionaries Are Similar to Lists

Dictionaries are similar to lists in a few ways. The following table compares lists to dictionaries and categorizes their traits as same, similar, or different.

viwo i *ea*

	Lists	Dictionaries
Similar	A list can store many *elements*.	A dictionary can store many *items*.
Different	Each element in a list does not have to be unique.	Each item in a dictionary is a key value pair. Each key must be unique within a dictionary. Each value does not have to be unique.
	Lists were designed for efficiently storing elements. Lists use less memory than dictionaries. However, finding an element in a list is relatively slow.	Dictionaries were designed for quickly finding items. Finding an item in a dictionary is fast. However, dictionaries use more memory than lists.
	A programmer uses square brackets ([and]) to create a list.	A programmer uses curly braces ({ and }) to create a dictionary.

```
    # Create a list of cities.
    cities_list = ["Delhi", "Lagos", "Dallas"]

    # Create a dictionary of people.
    people_dict = {
        "P203": "Ignacio Torres",
        "P445": "Whitney Nelson",
```

```
    "P128": "Yasmin Li"
}
```

	Lists	Dictionaries
Same	Lists are mutable, which means a program can add and remove elements after a list is created.	Dictionaries are mutable, which means a program can add and remove items after a dictionary is created.
Different	A programmer calls the `insert` and `append` methods to add an element to a list.	A programmer uses square brackets ([and]) to add an item to a dictionary.

```
# Add two cities to the cities list.
cities_list.insert(1, "Paris")
cities_list.append("Tokyo")

# Add two people to the people dictionary.
people_dict["P205"] = "Liam Myers"
people_dict["P317"] = "Davina Patel"
```

Similar	To cause the computer to check whether an *element* is in a list, a programmer uses the `in` keyword.	To cause the computer to check whether a *key* is in a dictionary, a programmer uses the `in` keyword.

```
if "Paris" in cities_list:
    print("Paris is in the list of cities.")

if "P203" in people_dict:
    print("P203 is in the dictionary of people.")
```

Different	A programmer uses the `index` method to find an element in a list.	A programmer uses square brackets ([and]) and a key to find an item in a list.

```
# Find Dallas in the cities list.
index = cities_list.index("Dallas")

# Find person P128 in the people dictionary.
person_name = people_dict["P128"]
```

Similar	A programmer uses square brackets ([and]) and an index to retrieve an element from a list.	A programmer uses square brackets ([and]) and a key to retrieve a value from a dictionary.

	Lists	Dictionaries

```
# Retrieve the element stored at
# index 2 in the cities list.
city_name = cities_list[2]

# Find person P128 in the people dictionary
# and retrieve the corresponding value.
person_name = people_dict["P128"]
```

	A programmer uses square brackets ([and]) and an index to replace an element in a list.	A programmer uses square brackets ([and]) and a key to replace a value in a dictionary.

```
# Change the city name at index 2 to London.
cities_list[2] = "London"

# Change the name of person P205 to Finn Meyers.
people_dict["P205"] = "Finn Myers"
```

Similar	A programmer can use a `for` loop to process all the elements in a list.	A programmer can use a `for` loop to process all the items in a dictionary.

```
# Process all the elements in the cities list.
for city_name in cities_list:
    print(city_name)

# Process all the items in the people dictionary.
for person_key, person_name in people_dict.items():
    print(person_name)
```

Same	A programmer uses the `pop` method to remove an element from a list.	A programmer uses the `pop` method to remove an item from a dictionary.

```
# Remove the element at index 3
# from the cities list.
cities_list.pop(3)

# Remove the key "P203" and its
# value from the people dictionary.
people_dict.pop("P203")
```

	Lists are passed by reference into a function.	Dictionaries are passed by reference into a function.

```
# Call the draw_chart function and pass
# the citites list to that function.
```

Lists **Dictionaries**

```
    draw_chart(cities_list)

    # Call the hire_people function and pass
    # the people dictionary to that function.
    hire_people(people_dict)
```

Converting between Lists and Dictionaries

It is possible to convert two lists into a dictionary by using the built-in `zip` and `dict` functions. The contents of the first list will become the keys in the dictionary, and the contents of the second list will become the values. This implies that the two lists must have the same length, and the elements in the first list must be unique because keys in a dictionary must be unique.

It is also possible to convert a dictionary into two lists by using the `keys` and `values` methods and the built-in `list` function. The following code example starts with two lists, converts them into a dictionary, and then converts the dictionary into two lists.

```
 1  # Example 7
 2
 3  def main():
 4      # Create a list that contains five student numbers.
 5      numbers = ["42-039-4736", "61-315-0160",
 6              "10-450-1203", "75-421-2310", "07-103-5621"]
 7
 8      # Create a list that contains five student names.
 9      names = ["Clint Huish", "Amelia Davis",
10              "Ana Soares", "Abdul Ali", "Amelia Davis"]
11
12      # Convert the numbers list and names list into a dictionary.
13      student_dict = dict(zip(numbers, names))
14
15      # Print the entire student dictionary.
16      print("Dictionary:", student_dict)
17      print()
18
19      # Convert the student dictionary into
20      # two lists named keys and values.
21      keys = list(student_dict.keys())
22      values = list(student_dict.values())
23
24      # Print both lists.
25      print("Keys:", keys)
26      print()
27      print("Values:", values)
28
29
30  # Call main to start this program.
31  if __name__ == "__main__":
32      main()
```

```
> python example_7.py
Dictionary: {'42-039-4736': 'Clint Huish',
'61-315-0160': 'Amelia Davis', '10-450-1203': 'Ana Soares',
'75-421-2310': 'Abdul Ali', '07-103-5621': 'Amelia Davis'}
```

```
Keys: ['42-039-4736', '61-315-0160', '10-450-1203',
'75-421-2310', '07-103-5621']

Values: ['Clint Huish', 'Amelia Davis', 'Ana Soares',
'Abdul Ali', 'Amelia Davis']
```

Tutorials

The following tutorials contain more information about dictionaries in Python.

- Official Python tutorial about dictionaries
- RealPython tutorial titled Dictionaries in Python

Summary

A dictionary in a Python program can store many pieces of data called items. An item is a key value pair. Each key that is stored in a dictionary must be unique. Values do not have to be unique. To create a dictionary, we use curly braces ({ and }). To add an item and find an item in a dictionary, we use the square brackets ([and]) and a key. To process all items in a dictionary, we write a `for` each loop. Dictionaries were invented to enable a computer to find items very quickly. Do not write a for each loop to find an item in a dictionary. To find an item in a dictionary, use square brackets ([and]) and a key.