GLOBAL EDITION

Intro to Python®

for Computer Science and Data Science

Paul Deitel • Harvey Deitel

Learning to Program with AI, Big Data and the Cloud

Digital Resources for Students

Your eBook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for Paul and Harvey Deitel's *Intro to Python®: for Computer Science and Data Science*, Global Edition.

1. Go to www.pearsonglobaleditions.com.
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

ISSPCD-WAHOO-DIARY-KALPA-FRACK-OOSSE

Use the login name and password you created during registration to start using the online resources that accompany your textbook.

IMPORTANT:

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable.

For technical support, go to https://support.pearson.com/getsupport.

Dictionary Views

Dictionary methods `items`, `keys` and `values` each return a view of a dictionary's data. When you iterate over a **view**, it "sees" the dictionary's current contents—it does *not* have its own copy of the data.

To show that views do *not* maintain their own copies of a dictionary's data, let's first save the view returned by `keys` into the variable `months_view`, then iterate through it:

```
In [4]: months_view = months.keys()

In [5]: for key in months_view:
   ...:     print(key, end='  ')
   ...:
January  February  March
```

Next, let's add a new key–value pair to `months` and display the updated dictionary:

```
In [6]: months['December'] = 12

In [7]: months
Out[7]: {'January': 1, 'February': 2, 'March': 3, 'December': 12}
```

Now, let's iterate through `months_view` again. The key we added above is indeed displayed:

```
In [8]: for key in months_view:
   ...:     print(key, end='  ')
   ...:
January  February  March  December
```

Do not modify a dictionary while iterating through a view. According to Section 4.10.1 of the Python Standard Library documentation,[1] either you'll get a `RuntimeError` or the loop might not process all of the view's values.

Converting Dictionary Keys, Values and Key–Value Pairs to Lists

You might occasionally need *lists* of a dictionary's keys, values or key–value pairs. To obtain such a list, pass the view returned by `keys`, `values` or `items` to the built-in `list` function. Modifying these lists does *not* modify the corresponding dictionary:

```
In [9]: list(months.keys())
Out[9]: ['January', 'February', 'March', 'December']

In [10]: list(months.values())
Out[10]: [1, 2, 3, 12]

In [11]: list(months.items())
Out[11]: [('January', 1), ('February', 2), ('March', 3), ('December', 12)]
```

Processing Keys in Sorted Order

To process keys in *sorted* order, you can use built-in function `sorted` as follows:

```
In [12]: for month_name in sorted(months.keys()):
    ...:     print(month_name, end='  ')
    ...:
February  December  January  March
```

1. `https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects`.

Self Check

1 *(Fill-In)* Dictionary method ________ returns an unordered list of the dictionary's keys.
Answer: keys.

2 *(True/False)* A view has its own copy of the corresponding data from the dictionary.
Answer: False. A view does *not* have its own copy of the corresponding data from the dictionary. As the dictionary changes, each view updates dynamically.

3 *(IPython Session)* For the following dictionary, create lists of its keys, values and items and show those lists.

```
roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5}
```

Answer:

```
In [1]: roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5}

In [2]: list(roman_numerals.keys())
Out[2]: ['I', 'II', 'III', 'V']

In [3]: list(roman_numerals.values())
Out[3]: [1, 2, 3, 5]

In [4]: list(roman_numerals.items())
Out[4]: [('I', 1), ('II', 2), ('III', 3), ('V', 5)]
```

6.2.5 Dictionary Comparisons

The comparison operators == and != can be used to determine whether two dictionaries have identical or different contents. An equals (==) comparison evaluates to True if both dictionaries have the same key–value pairs, *regardless* of the order in which those key–value pairs were added to each dictionary:

```
In [1]: country_capitals1 = {'Belgium': 'Brussels',
   ...:                      'Haiti': 'Port-au-Prince'}
   ...:

In [2]: country_capitals2 = {'Nepal': 'Kathmandu',
   ...:                      'Uruguay': 'Montevideo'}
   ...:

In [3]: country_capitals3 = {'Haiti': 'Port-au-Prince',
   ...:                      'Belgium': 'Brussels'}
   ...:

In [4]: country_capitals1 == country_capitals2
Out[4]: False

In [5]: country_capitals1 == country_capitals3
Out[5]: True

In [6]: country_capitals1 != country_capitals2
Out[6]: True
```

✓ Self Check

1 *(True/False)* The == comparison evaluates to `True` only if both dictionaries have the same key–value pairs in the same order.
Answer: False. The == comparison evaluates to `True` if both dictionaries have the same key–value pairs, regardless of their order.

6.2.6 Example: Dictionary of Student Grades

The script in Fig. 6.1 represents an instructor's grade book as a dictionary that maps each student's name (a string) to a list of integers containing that student's grades on three exams. In each iteration of the loop that displays the data (lines 13–17), we unpack a key–value pair into the variables `name` and `grades` containing one student's name and the corresponding list of three grades. Line 14 uses built-in function `sum` to total a given student's grades, then line 15 calculates and displays that student's average by dividing `total` by the number of grades for that student (`len(grades)`). Lines 16–17 keep track of the total of all four students' grades and the number of grades for all the students, respectively. Line 19 prints the class average of all the students' grades on all the exams.

```
 1  # fig06_01.py
 2  """Using a dictionary to represent an instructor's grade book."""
 3  grade_book = {
 4      'Susan': [92, 85, 100],
 5      'Eduardo': [83, 95, 79],
 6      'Azizi': [91, 89, 82],
 7      'Pantipa': [97, 91, 92]
 8  }
 9
10  all_grades_total = 0
11  all_grades_count = 0
12
13  for name, grades in grade_book.items():
14      total = sum(grades)
15      print(f'Average for {name} is {total/len(grades):.2f}')
16      all_grades_total += total
17      all_grades_count += len(grades)
18
19  print(f"Class's average is: {all_grades_total / all_grades_count:.2f}")
```

```
Average for Susan is 92.33
Average for Eduardo is 85.67
Average for Azizi is 87.33
Average for Pantipa is 93.33
Class's average is: 89.67
```

Fig. 6.1 | Using a dictionary to represent an instructor's grade book.

6.2.7 Example: Word Counts[2]

The script in Fig. 6.2 builds a dictionary to count the number of occurrences of each word in a string. Lines 4–5 create a string `text` that we'll break into words—a process known as **tokenizing a string**. Python automatically concatenates strings separated by whitespace in parentheses. Line 7 creates an empty dictionary. The dictionary's keys will be the unique words, and its values will be integer counts of how many times each word appears in `text`.

```
 1  # fig06_02.py
 2  """Tokenizing a string and counting unique words."""
 3
 4  text = ('this is sample text with several words '
 5          'this is more sample text with some different words')
 6
 7  word_counts = {}
 8
 9  # count occurrences of each unique word
10  for word in text.split():
11      if word in word_counts:
12          word_counts[word] += 1  # update existing key-value pair
13      else:
14          word_counts[word] = 1  # insert new key-value pair
15
16  print(f'{"WORD":<12}COUNT')
17
18  for word, count in sorted(word_counts.items()):
19      print(f'{word:<12}{count}')
20
21  print('\nNumber of unique words:', len(word_counts))
```

```
WORD        COUNT
different   1
is          2
more        1
sample      2
several     1
some        1
text        2
this        2
with        2
words       2
Number of unique words: 10
```

Fig. 6.2 | Tokenizing a string and counting unique words.

Line 10 tokenizes `text` by calling string method **`split`**, which separates the words using the method's delimiter string argument. If you do not provide an argument, `split` uses a space. The method returns a list of tokens (that is, the words in `text`). Lines 10–14

2. Techniques like word frequency counting are often used to analyze published works. For example, some people believe that the works of William Shakespeare actually might have been written by Sir Francis Bacon, Christopher Marlowe or others. Comparing the word frequencies of their works with those of Shakespeare can reveal writing-style similarities. We'll look at other document-analysis techniques in the "Natural Language Processing (NLP)" chapter.

iterate through the list of words. For each word, line 11 determines whether that word (the key) is already in the dictionary. If so, line 12 increments that word's count; otherwise, line 14 inserts a new key–value pair for that word with an initial count of 1.

Lines 16–21 summarize the results in a two-column table containing each word and its corresponding count. The for statement in lines 18 and 19 iterates through the dictionary's key–value pairs. It unpacks each key and value into the variables word and count, then displays them in two columns. Line 21 displays the number of unique words.

Python Standard Library Module collections

The Python Standard Library already contains the counting functionality that we implemented using the dictionary and the loop in lines 10–14. The module **collections** contains the type **Counter**, which receives an iterable and summarizes its elements. Let's reimplement the preceding script in fewer lines of code with Counter:

```
In [1]: from collections import Counter

In [2]: text = ('this is sample text with several words '
   ...:         'this is more sample text with some different words')
   ...:

In [3]: counter = Counter(text.split())

In [4]: for word, count in sorted(counter.items()):
   ...:     print(f'{word:<12}{count}')
   ...:
different   1
is          2
more        1
sample      2
several     1
some        1
text        2
this        2
with        2
words       2

In [5]: print('Number of unique keys:', len(counter.keys()))
Number of unique keys: 10
```

Snippet [3] creates the Counter, which summarizes the list of strings returned by text.split(). In snippet [4], Counter method **items** returns each string and its associated count as a tuple. We use built-in function sorted to get a list of these tuples in ascending order. By default sorted orders the tuples by their first elements. If those are identical, then it looks at the second element, and so on. The for statement iterates over the resulting sorted list, displaying each word and count in two columns.

✓ Self Check

1 *(Fill-In)* String method ________ tokenizes a string using the delimiter provided in the method's string argument.
Answer: split.

2 *(IPython Session)* Use a comprehension to create a list of 50 random integers in the range 1–5. Summarize them with a Counter. Display the results in two-column format.

Answer:

```
In [1]: import random

In [2]: numbers = [random.randrange(1, 6) for i in range(50)]

In [3]: from collections import Counter

In [4]: counter = Counter(numbers)

In [5]: for value, count in sorted(counter.items()):
   ...:     print(f'{value:<4}{count}')
   ...:
1   9
2   6
3   13
4   10
5   12
```

6.2.8 Dictionary Method update

You may insert and update key–value pairs using dictionary method **update**. First, let's create an empty country_codes dictionary:

```
In [1]: country_codes = {}
```

The following update call receives a dictionary of key–value pairs to insert or update:

```
In [2]: country_codes.update({'South Africa': 'za'})

In [3]: country_codes
Out[3]: {'South Africa': 'za'}
```

Method update can convert keyword arguments into key–value pairs to insert. The following call automatically converts the parameter name Australia into the string key 'Australia' and associates the value 'ar' with that key:

```
In [4]: country_codes.update(Australia='ar')

In [5]: country_codes
Out[5]: {'South Africa': 'za', 'Australia': 'ar'}
```

Snippet [4] provided an incorrect country code for Australia. Let's correct this by using another keyword argument to update the value associated with 'Australia':

```
In [6]: country_codes.update(Australia='au')

In [7]: country_codes
Out[7]: {'South Africa': 'za', 'Australia': 'au'}
```

Method update also can receive an iterable object containing key–value pairs, such as a list of two-element tuples.

6.2.9 Dictionary Comprehensions

Dictionary comprehensions provide a convenient notation for quickly generating dictionaries, often by mapping one dictionary to another. For example, in a dictionary with *unique* values, you can reverse the key–value pairs: