00001111

CHAPTER 15

Dictionaries

Learning Objectives

- *Explain the term key/value pair*
- *List the desirable characteristics of a key*
- *Explain the purpose of a dictionary*
- *Create and initialize dictionaries using curly braces { }*
- *Create and initialize dictionaries using the dict() constructor*
- *Extract dictionary key/value pairs with the items() method*
- *Iterate over a dictionary using a for statement*
- *Create and initialize dictionaries using dictionary comprehensions*
- *Access dictionary values by key*
- *Extract dictionary key lists*
- *Add list objects to a dictionary*
- *Convert a dictionary into a JSON string*
- *Convert a JSON string into a dictionary*

Introduction

You will often find it helpful to store and access data based on some type of mapping, where a *key* is used to calculate a location in which to store some type of *value*. Such data is said to consist of *key/value pairs*. Python provides a mapping type that does just this — the dictionary or *dict*.

In this chapter you will learn how to create and use dictionaries in your programs. You'll learn how to add keys and values to dictionaries, what types of objects make good keys, and how to formulate key names that translate well into cross-platform compatible JSON. This is a critical skill to add to your programmer's tool belt because the best way to create valid JSON is to start with a dictionary.

Also in this chapter you'll learn how to process dictionaries using `for` statements by iterating over a dictionary's key/value pairs. You'll also learn *when* to use dictionaries

The beautiful thing about dictionaries is that once you learn how to use them, you'll wonder how you ever lived without them. Once you learn to master the power dictionaries provide, you'll think of a zillion ways to use them in your programs.

1 Dictionary Fundamentals

A dictionary (`dict`) is a Python data type that lets you store and access data using *key/value* pairs. A *key* is used to index its associated data or *value*. **Keys** must be **immutable**, which limits the types of objects that can be used as keys. Values can be anything, including other dictionaries.

A dictionary is but one of a small handful of Python mapping types, but it's the one most often used. Dictionaries are also referred to as *hash tables* or *associative arrays*. Figure 15-1 gives a simplified conceptual view of how a dictionary insertion works.

```
my_dict["a"] = "valueOne"
```

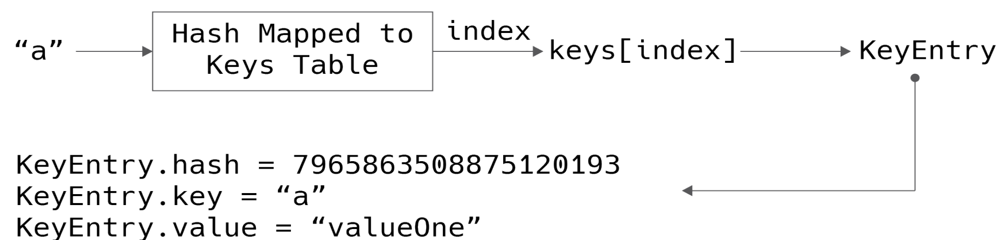

Figure 15-1: Dictionary — A Conceptual View

Referring to figure 15-1 — The incoming key `"a"` is hash mapped to an *index* value which is then used as an index into a keys table. Information about the incoming key and value is stored in a KeyEntry object which has fields for the hash, key, and value.

1.1 Why Must A Key Be Immutable?

A key must be immutable because its value cannot change while it exists. Python immutable types provide a `__hash__()` function which calculates and returns a **hash value** unique to the key's content. Key hash values are salted with a random number which changes between Python

0 0 0 0 1 1 1 1

interpreter sessions. In other words, hash values for a particular key change each time a program executes but remain constant while the program is running. Keys must also be unique within a particular dictionary instance. Subsequent values inserted with the same key will overwrite previous values. I'll talk more about this later.

Strings make great keys, as do numbers. Tuples can be used for keys but the objects within the tuple must also be immutable. I use strings as keys and have never encountered a need to do otherwise.

1.2 Creating And Populating Dictionaries

You can create and populate dictionaries in several ways as shown in example 15.1.

15.1 creating_dictionaries.py

```
1    """Demonstrate how to create dictionaries."""
2
3    def main():
4        # Print hash values of strings
5        keyOne = "a"
6        print(f'     keyOne.__hash__() == {keyOne.__hash__()}')
7        print(f'        "a".__hash__() == {"a".__hash__()}')
8        print(f'       "aa".__hash__() == {"aa".__hash__()}')
9        print(f'("a" + "a").__hash__() == {("a" + "a").__hash__()}')
10
11       # Create an empty dictionary with braces
12       my_dict = {}
13
14       # Insert a value using keyOne
15       my_dict[keyOne] = "valueOne"
16
17       # Reusing same key will overwrite stored value
18       my_dict["a"] = "valueTwo"
19
20       # Print entire dictionary
21       print(f'my_dict == {my_dict}')
22
23       # Access dictionary elements via key
24       print(f'Value at my_dict[keyOne] == {my_dict[keyOne]}')
25       print(f'Value at my_dict["a"] == {my_dict["a"]}')
26
27       # Create empty dictionary with dict() constructor
28       book_info = dict()
29       book_info['bookTitle'] = 'Computer Scripting Techniques with Python'
30       book_info['author'] = 'Rick Miller'
31       book_info['isbn13'] = '978-1-932504-13-2'
32       print(f'book_info == {book_info}')
33
34       # Create dictionary with dictionary literal
35       classrooms = {'bal-4004':{'instructor':'R. Miller'},
36                     'bal-3066':{'instructor':'K. Nesvit'}}
37       print(f'classrooms == {classrooms}')
38
39   if __name__ == '__main__':
40       main()
41
```

Referring to example 15.1 — Starting on line 5, I declare a variable named `keyOne` and initialize it to the character `"a"`. On lines 6 through 9, I print various hash values by calling the

`__hash__()` method on `keyOne`, `"a"`, `"aa"`, and `("a" + "a")`. On line 12, I create an empty dictionary using a set of empty braces `"{}"`. Then, on line 15, I use the variable `keyOne` as a key and assign the string value `"valueOne"`. Then, on line 18, I use the character `"a"` as a key and assign the string value `"valueTwo"`. Because the variable `keyOne` and `"a"` represent the same key, the value associated with that key is overwritten, the effects of which are shown when printing the entire dictionary to the console on line 21, and again when printing individual values to the console on lines 24 and 25.

Another way to create a dictionary, shown on line 28, is with the `dict()` constructor. I then use various string keys on lines 29 through 31 to store information about a book. On line 32, I print the entire `book_info` dictionary to the console.

Finally, on line 35, I use a dictionary literal to initialize a dictionary named `classrooms`. On line 37, I print the `classrooms` dictionary to the console. Figure 15-2 shows the results of running this program twice.

```
-bash
~/d/c/c/creating_dictionaries | main • | 5/18, 3:48 PM
Sat May 18 15:47:29 EDT 2024
~/dev/cst_with_python_1st_ed/chapter15/creating_dictionaries (main)
[517:17] swodog@macos-mojave-testbed $  python3 creating_dictionaries.py
     keyOne.__hash__() == 4117513995689881344
        "a".__hash__() == 4117513995689881344
       "aa".__hash__() == -767761817920934340
("a" + "a").__hash__() == -767761817920934340
my_dict == {'a': 'valueTwo'}
Value at my_dict[keyOne] == valueTwo
Value at my_dict["a"] == valueTwo
book_info == {'bookTitle': 'Computer Scripting Techniques with Python', 'author': 'Rick Miller', 'isbn13': '978-1-932504-13-2'}
classrooms == {'bal-4004': {'instructor': 'R. Miller'}, 'bal-3066': {'instructor': 'K. Nesvit'}}

Sat May 18 15:47:32 EDT 2024
~/dev/cst_with_python_1st_ed/chapter15/creating_dictionaries (main)
[518:18] swodog@macos-mojave-testbed $  python3 creating_dictionaries.py
     keyOne.__hash__() == 1706874858598433603
        "a".__hash__() == 1706874858598433603
       "aa".__hash__() == -4963990119651280150
("a" + "a").__hash__() == -4963990119651280150
my_dict == {'a': 'valueTwo'}
Value at my_dict[keyOne] == valueTwo
Value at my_dict["a"] == valueTwo
book_info == {'bookTitle': 'Computer Scripting Techniques with Python', 'author': 'Rick Miller', 'isbn13': '978-1-932504-13-2'}
classrooms == {'bal-4004': {'instructor': 'R. Miller'}, 'bal-3066': {'instructor': 'K. Nesvit'}}
```

Figure 15-2: Results of Running Example 15.1 Twice

Referring to figure 15-2 — Notice that each program run produces different key hash values. Again, this is due to hash values being seeded with a random number each time a program executes. This is done to prevent hacking. You can learn more about how hashing works in the Python interpreter by reading *CPython Internals: Your Guide To The Python 3 Interpreter, First Edition, by Anthony Shaw, ISBN: 9781775093343*.

Still referring to figure 15-2 — Notice how `valueTwo` has overwritten `valueOne`. This is the general behavior of dictionaries and can be summarized like so: *"The last man wins!"*.

When printing an entire dictionary to the console, it is enclosed in curly braces with keys being separated from values by a colon `':'`. In this example, all the keys and their associated values are enclosed in single quotes indicating they are strings. Note that the `classrooms` dictionary contains dictionaries as values. They are contained within curly braces because they are dictionaries. Let's look at another example with different types of objects being used as keys and values.

© 2024 Pulp Free Press

15.2 multi_object.py

```
1    """Demonstrate dictionaries with different types as keys and values."""
2
3    def main():
4        # Numbers as keys and values
5        d1 = {1:100, 2:200, 3:300}
6        print(f'd1 == {d1}')
7
8        # String keys with list and number values
9        d2 = {'names': ['Judy', 'Davis', 'Lewis'], 'count':3}
10       print(f'd2 == {d2}')
11
12
13   if __name__ == '__main__':
14       main()
15
```

Referring to example 15.2 — On line 5, I create a dictionary named `d1` comprised of numeric keys and values. On line 9, I create dictionary d2 comprised of string keys, and a list and a number as values. Figure 15-3 shows how each of these dictionaries renders to the console.

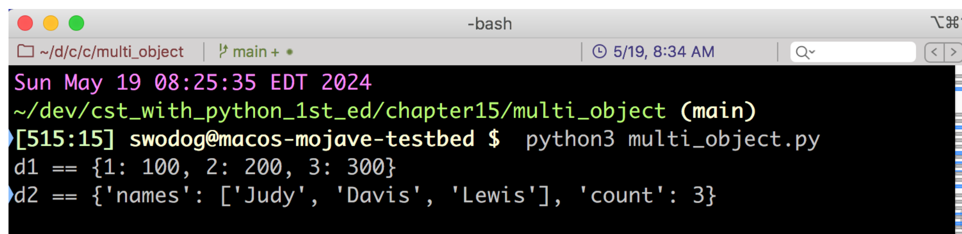

Figure 15-3: Results of Running Example 15.2

Referring to figure 15-3 — Note how the numeric values and the list are rendered within the dictionary output. Numeric values are not surrounded in quotes while lists are contained within a set of square brackets.

1.3 Dictionary Comprehensions

Another way to create and initialize a dictionary is with a dictionary comprehension. Example 15.3 converts a text document into a dictionary of paragraphs indexed by paragraph number.

15.3 dictionary_comprehension.py

```
1    """Demonstrate dictionary comprehensions."""
2
3    def main():
4        document = '\tAs a kid, I dreamed of exploring the Amazon jungle. ' \
5            'My friends and I watched a TV show about Amazon explorers and ' \
6            'their adventures.\n\tThey became tangled in huge webs and fought ' \
7            'huge spiders. Every creature seemed larger than life.\n' \
8            '\tWhen we finished watching the show, we explored the woods next ' \
9            'to our neighborhood and imagined we were exploring the Amazon. ' \
10           'We fashioned machetes from aluminum window frames and chopped ' \
11           'our way through the brush as best we could. To us kids, the ' \
12           'machetes were real and our adventures were just as exciting. ' \
13           'Thankfully, the spiders were small.'
14
15       # Print document to console for reference
16       print(document)
17       print('*' * 60)
```

```
18
19      # Split the document into list of paragraphs
20      paragraph_list = document.split('\n')
21
22      # Use the enumerate() function to extract the indices and values from list
23      paragraph_dict = {(key + 1):value for key, value in enumerate(paragraph_list)}
24
25      # Print the key and value of each dictionary entry
26      for key, value in paragraph_dict.items():
27          print(f'Paragraph {key}: {value}')
28
29
30  if __name__ == '__main__':
31      main()
32
```

Referring to example 15.3 — On line 4, I initialize a variable named `document` with the text of a short story from my youth. Next, I print the document to the console for reference. On line 20, I create a list of paragraphs by splitting the document at the newline `'\n'` characters. On line 23, I use a dictionary comprehension to create a dictionary from the list of paragraphs. As you learned in the previous chapter, the `enumerate()` function returns both the indices and values from a list. The indices are numbers so they are valid keys. I add one to each `key` mainly so the first paragraph in the dictionary has a key value of `1` vs. `0`. Finally, the `for` statement on line 26 iterates through the dictionary items and prints each key/value pair to the console. Note that the `dictionary.items()` method returns an iterable set of key/value pairs. Figure 15-4 shows the results of running this program.

```
-bash
~/d/c/c/d                main + •                5/25, 7:30 AM
[528:28] swodog@macos-mojave-testbed $  python3 dictionary_comprehensions.py
        As a kid, I dreamed of exploring the Amazon jungle. My friends and I watched a TV show about Ama
zon explorers and their adventures.
        They became tangled in huge webs and fought huge spiders. Every creature seemed larger than life
.
        When we finished watching the show, we explored the woods next to our neighborhood and imagined
we were exploring the Amazon. We fashioned machetes from aluminum window frames and chopped our way thro
ugh the brush as best we could. To us kids, the machetes were real and our adventures were just as excit
ing. Thankfully, the spiders were small.
************************************************************
Paragraph 1:    As a kid, I dreamed of exploring the Amazon jungle. My friends and I watched a TV show a
bout Amazon explorers and their adventures.
Paragraph 2:    They became tangled in huge webs and fought huge spiders. Every creature seemed larger t
han life.
Paragraph 3:    When we finished watching the show, we explored the woods next to our neighborhood and i
magined we were exploring the Amazon. We fashioned machetes from aluminum window frames and chopped our
way through the brush as best we could. To us kids, the machetes were real and our adventures were just
as exciting. Thankfully, the spiders were small.
```

Figure 15-4: Results of Running Example 15.3

1.3.1 Parting Thoughts On Dictionary Comprehensions

Like their list comprehension counterparts, dictionary comprehensions can often be used to simplify and optimize dictionary initialization, but overly-complex dictionary comprehensions can lead to obfuscated code.

Pro Tip: Avoid the use of overly-complex dictionary comprehensions.

© 2024 Pulp Free Press

Quick Review

A dictionary (`dict`) is a Python mapping data type that lets you store and access data using *key/value* pairs. A *key* is used to index its associated data or *value*. **Keys** must be **immutable**, which limits the sorts of objects that can be used as keys. Values can be anything, including other dictionaries.

A key must be immutable because its value cannot change while it exists. Immutable objects provide a `__hash__()` function. The value calculated and returned by the `__hash__()` function must always be the same for a given object state.

You can create dictionaries in several different ways. The most often used approach is to first create an empty dictionary with an empty set of curly braces "`{}`" and set each key/value individually.

The `dictionary.items()` method returns an iterable set of key/value pairs.

Avoid the use of overly-complex dictionary comprehensions as they can lead to obfuscated code.

2 Processing Dictionaries

In the previous section you learned pretty much all you need to know about dictionaries to put them to effective use in your programs. In this section, I'd like to raise your awareness about a handful of unique features and operations supported by dictionaries. First, let's take a look at operations supported by dictionaries.

2.1 Operations Supported By Dictionaries

Table 15-1 lists operations supported by dictionaries. You've seen some of these in action in the previous section. Note that this is an abridged listing. To learn more about an operation please consult the reference listed at the bottom of the table.

Operation	Result
`dictionary = {}`	Create empty dictionary with empty curly braces.
`dictionary = dict()`	Create empty dictionary with `dict()` constructor.
`dictionary[key] = value`	Add *value* to dictionary indexed by *key*. Key must be an immutable object. Overwrites value if key already in dictionary. (Last man wins!)
`dictionary[key]`	Return *value* indexed by *key*. Raises *KeyError* exception if key not in dictionary.
`len(dictionary)`	Returns number of items in dictionary.
`list(dictionary)`	Returns list of keys in inserted order.
`iter(dictionary)`	Returns an iterator over the keys in the dictionary.
`reversed(dictionary)`	Return a reverse iterator over the keys contained in the dictionary.

Table 15-1: Operations Supported by Dictionaries

Operation	Result
sorted(dictionary**)**	Returns list of keys in sorted order.
del dictionary[key]	Delete *value* indexed by *key*. Raises *KeyError* exception if key not in dictionary.
key **in** dictionary	Returns `True` if dictionary contains *key*.
key **not in** dictionary	Returns `True` if *key* not contained in dictionary.
dictionary.**items()**	Returns iterable set of key/value pairs. (A dynamically updated view object)
dictionary.**clear()**	Removes all items from dictionary.
dictionary.**copy()**	Returns a shallow copy of the dictionary.
dictionary.**get(**key, default=None**)**	Return value indexed by key if key is in the dictionary, else return the default value.
dictionary.**keys()**	Returns an iterable set of the dictionary's keys. (A dynamically updated view object.)
dictionary.**pop(**key [, default]**)**	Return value indexed by *key* if *key* is in the dictionary, else return *default*. If *default* value is not provided, raises KeyError exception.
dictionary.**popitem()**	Return and remove a key/value pair from the dictionary. Items are returned in Last-In-First-Out (LIFO) order. Raises KeyError exception if dictionary is empty.
dictionary.**setdefault(**key, default=None**)**	Return value indexed by *key* if *key* is in the dictionary, otherwise, insert *key* with value of *default* and return *default*. (*default* initialized by default to `None`)
dictionary.**update(**[other]**)**	Update dictionary with key/value pairs from another dictionary or an iterable of key/value pairs and overwrite existing keys. Can also supply key/value pair arguments.
dictionary.**values()**	Returns an iterable set of the dictionary's values. (Dynamically updated view object.)
dictionary **|** other	Create a new dictionary with the merged keys and values of *dictionary* and *other*.
dictionary **|=** other	Update *dictionary* with keys and values from *other*. Key/values from *other* will overwrite shared keys in *dictionary*.
Source: *https://docs.python.org/3/library/stdtypes.html#mapping-types-dict*	

Table 15-1: Operations Supported by Dictionaries (Continued)

Referring to table 15-1 — Operations fall into several broad categories:

- Operations concerned with keys
- Operations concerned with values
- Operations concerned with items (i.e., key/value pairs)

For example, and most obvious, are the methods `dictionary.keys()`, `dictionary.values()`, and `dictionary.items()`. What's not so obvious is that these methods return dictionary

0 0 0 0 1 1 1 1

view objects. Always keep in mind that the term *item* refers to a particular key/value pair. So when you call the `dictionary.items()` method, it will return a dictionary view object of key/value pair tuples. I cover dictionary view objects in more detail in the next section.

Ordinary dictionary access in the form of `dictionary[key]` will raise a `KeyError` exception if the key is not in the dictionary. All dictionary processing operations should be enclosed within a `try/except` statement. If you are unsure if the key you are trying to access is actually present in the dictionary, use the `dictionary.get(key, default=None)` method to avoid raising an exception.

Pro Tip: Place dictionary processing code in a try/except statement to properly handle KeyError exceptions.

2.1.1 Dictionary View Objects

Dictionary view objects dynamically update when changes are made to the underlying dictionary. Let's take a look at the `dictionary.items()` method in action.

15.4 dictionary_view_objects.py

```
1    """Demonstrate dictionary view objects."""
2
3    def main():
4       # Create and initialize dictionary
5       animals = {}
6       animals['zebra'] = 'Horse-like animal with black and white stripes.'
7       animals['rooster'] = 'Male chicken. Very annoying in the morning.'
8       animals['dog'] = 'Man\'s best friend.'
9       animals['cat'] = 'Internet star!'
10
11      # Extract view object
12      animal_items = animals.items()
13
14      # Iterate over view object's key/value pairs
15      for key, value in animal_items:
16         print(f'{key} : {value}')
17
18      # Add another item
19      animals['pony'] = 'Every little girl\'s dream.'
20
21      print('*' * 60)
22
23      # Iterate over view object's sorted key/value pairs
24      for key, value in sorted(animal_items):
25         print(f'{key} : {value}')
26
27   if __name__ == '__main__':
28      main()
29
```

Referring to example 15.4 — Starting on line 5, I create a dictionary named `animals` and populate it with four items: a `zebra`, a `rooster`, a `dog`, and a `cat`. On line 12, I create a variable named `view_items` and assign to it the items view object by calling the `animals.items()` method. Using the `for` statement on line 15, I iterate over the items and print the key/value pairs to the console. Next, on line 19, I add another animal to the dictionary, and again I iterate over the

items with a `for` loop. I use the `sorted()` function to obtain the keys in sorted order. Figure 15-5 shows the results of running this program.

```
-bash
~/dev/cst_...iew_objects | main + • | 5/26, 11:10 AM

Sun May 26 10:51:34 EDT 2024
~/dev/cst_with_python_1st_ed/chapter15/dictionary_view_objects.py (main)
[509:9] swodog@macos-mojave-testbed $  python3 dictionary_view_objects.py
zebra : Horse-like animal with black and white stripes.
rooster : Male chicken. Very annoying in the morning.
dog : Man's best friend.
cat : Internet star!
************************************************************
cat : Internet star!
dog : Man's best friend.
pony : Every little girl's dream.
rooster : Male chicken. Very annoying in the morning.
zebra : Horse-like animal with black and white stripes.
```

Figure 15-5: Results of Running Example 15.4

Referring to figure 15-5 — Items are normally accessed via an iterator in the order in which they were inserted, also referred to as first-in-first-out or FIFO for short. The `sorted()` function sorts the keys in ascending order. Likewise, if you wanted to obtain the keys in reverse order, you would use the `reversed()` function.

Quick Review

Dictionaries support a wide range of operations. Dictionary operations fall into three broad categories: operations concerning keys, operations concerning values, and operations concerning both keys and values (key/value pairs, otherwise referred to as items).

The methods `dictionary.`**`items()`**, `dictionary.`**`keys()`**, and `dictionary.`**`values()`** return dictionary view objects, which dynamically update when the underlying dictionary changes.

Place dictionary processing code in a `try/except` statement to properly handle cases where an attempted key access raises a KeyError exception.

3 Converting Dictionaries To JSON

A dictionary is a wonderful place to start if you want to generate valid, cross-platform compatible JSON. JSON has become the de facto standard data interchange format, surpassing even XML in that honored position. Converting data into JSON enables you to share data between systems and applications running on different hardware and software platforms. The one weird trick that makes this possible is knowing how to create valid JSON with the help of a dictionary.

I this section, you'll learn how to build a dictionary with JSON generation in mind, and how to formulate JSON-valid keys. I'll limit the conversation to building complex JSON structures from standard Python types. Later in the book when you learn about object-oriented programming and how to create classes, I'll show you how to convert user-defined types into valid JSON.

Let's start with a few simple rules.

3.1 A Few Simple Rules

You only need to keep a few simple rules in mind to build a dictionary that can be converted into valid JSON and reliably shared with other systems and programming languages. The rules are listed in table 15-2.

Rule	Discussion
Use Strings for Key Names	Only use strings for key names. Do not use numbers or tuples.
Use `camelCase` or `snake_case` for Key Names	Either case is fine and portable across platforms, but I have my preferences. For JSON I intend to share with other systems, I use `camelCase`. For JSON I intend to use internal to my program, for example, a settings or application configuration file, I use `snake_case`.
Use Consistent Key Name Case	Whether you use `camelCase` or `snake_case` for key names, be consistent. Do not mix case types for keys within the same dictionary.
Limit Values to Fundamental Data Types	For example, *strings*, *numbers*, *boolean*, *lists*, and *dictionaries*. All these types of objects can be easily reconstituted by other programming languages.
Reduce Complex User-Defined Data Types to Fundamental Data Type JSON Representations	Related to the previous rule. Later in the book, I'll show you how to convert complex user-defined types into valid JSON.
Limit Data to ASCII Characters	If your data contains non-ASCII ensure it is properly escaped. JSON data should be transmitted using UTF-8 encoding.
Express Dates in ISO 8601 Format	Takes the guesswork out of datetime strings by providing a format for exact datetime expression: `yyyy-mm-ddThh:mm:ss-00:00`
Convert *NaN* or *Infinity* values to `None`	The Python `None` value translates to `null` in JSON.
Escape Special Characters	Escape special characters such as embedded quotes, backslashes, and control characters.
Ensure Valid JSON Syntax	Test generated JSON with an online JSON validation service.
Test Across Platforms	While following the rules listed in this table will produce valid JSON, always test the hell out of everything on the target system before using in production.

Table 15-2: Simple Rules For Valid JSON

Referring to table 15-2 — This looks like a lot to keep in mind, but really it boils down to the following: Use **strings for keys** and **adopt a consistent key case**, stick with **fundamental data types for values**, use a **universal format for datetime strings**, avoid special characters if at all possible but if you can't, make sure the **data is properly represented and escaped**, and finally, **test the generated JSON to ensure validity**.

If you're new to JSON, you may want to keep a copy of these rules at hand when you're adding data to a dictionary. If you try to convert a dictionary to JSON and it contains bad data, the `json.dumps()` method will raise an exception.

3.2 The Rules In Action

Example 15.5 gives a program that builds up a complex dictionary using many of the rules listed in table 15-2, generates JSON, saves the JSON to a file, then reads the file and creates a new dictionary from the JSON string.

15.5 json_from_dictionary.py

```
1    """Demonstrate building complex dictionary structure to generate valid JSON."""
2
3    import json
4    from datetime import datetime
5
6    def main():
7       # Create dictionary following rules from table 15-2
8       classes = {}
9       classes['it566'] = {}
10      classes['it566']['campus'] = 'Ballston Center'
11      classes['it566']['semester'] = 'Fall'
12      classes['it566']['year'] = 2024
13      classes['it566']['dates'] = {}
14      classes['it566']['dates']['begin'] = datetime(2024, 8, 26).isoformat()
15      classes['it566']['dates']['end'] = datetime(2024, 12, 7).isoformat()
16      classes['it566']['classroom'] = '4004'
17      classes['it566']['students'] = []
18
19      s1 = {'firstName':'Kateryna', 'lastName':'Nesvit'}
20      s2 = {'firstName':'Sapna', 'lastName':'Surana'}
21      s3 = {'firstName':'Jose', 'lastName':'Pi'}
22
23      classes['it566']['students'].append(s1)
24      classes['it566']['students'].append(s2)
25      classes['it566']['students'].append(s3)
26
27      # Convert dictionary to JSON and write to file
28      try:
29         with open('classes.json', 'w') as f:
30            f.write(json.dumps(classes))
31      except Exception as e:
32         print(f'Problem writing JSON to file: {e}')
33
34      # Read JSON file and create new dictionary
35      new_classes_dict = None
36      try:
37         with open('classes.json', 'r') as f:
38            new_classes_dict = json.loads(f.read())
39      except Exception as e:
40         print(f'Problem writing JSON to file: {e}')
41
42      if new_classes_dict != None:
43         print(f'New Classes Dictionary = {new_classes_dict}')
44
45
46   if __name__ == '__main__':
47      main()
48
```

Referring to example 15.5 — Note first that all key names are strings and in `camelCase`. The only two-syllable key not in `camelCase` is `classroom` on line 16. Note that `camelCase` is normally formed from multiple words, where the first word starts with a lower-case letter and each

subsequent word starts with an upper-case letter. (i.e., IT 566 becomes it566, First Name becomes firstName, Last Name becomes lastName, etc.)

Note also that on each highlighted line the required data structure is created before it can be populated. For example, on line 8, I create the topmost dictionary named `classes`. On line 9, the first class I add to the dictionary is `it566`, which is also a dictionary.

The next dictionary is created on line 13, `classes['it566']['dates'] = {}`. I create datetime objects then call the `datetime.isoformat()` method to render the datetime strings in ISO 8601 format. The default output string format is `'yyyy-mm-ddThh:mm:ss'`.

On line 17, I create an empty list of `students`. On lines 19 through 21, I create several student dictionaries, then append them to the `students` list on lines 23 through 25.

On lines 29 and 30, I use the built-in `open()` function to open a file for writing, then convert the classes dictionary to JSON using the `json.dumps()` method and write the JSON to the file. In this case, the name of the output file is `classes.json`.

Finally, to prove this is not a hoax, I create a new dictionary named `new_classes_dict` and then open the `classes.json` file and convert the JSON string back into a Python dictionary with the `json.loads()` method. Figure 15-6 shows the results of running this program.

```
-bash
~/d/c/c/json_from_dictionary   main   10/04, 4:00 PM
Fri Oct  4 15:59:30 EDT 2024
~/dev/cst_with_python_1st_ed/chapter15/json_from_dictionary (main)
[509:10] swodog@macos-mojave-testbed $  python3 json_from_dictionary.py
New Classes Dictionary = {'it566': {'campus': 'Ballston Center', 'semester': 'Fall', 'year': 2024, 'dates': {'begin': '2024-08-26T00:00:00', 'end': '2024-12-07T00:00:00'}, 'classroom': '4004', 'students': [{'firstName': 'Kateryna', 'lastName': 'Nesvit'}, {'firstName': 'Sapna', 'lastName': 'Surana'}, {'firstName': 'Jose', 'lastName': 'Pi'}]}}
```

Figure 15-6: Results of Running Example 15.5

Example 15.6 lists the contents of the classes.json file.

15.6 classes.json File Contents

```
1    {
2        "it566": {
3            "campus": "Ballston Center",
4            "classroom": "4004",
5            "dates": {
6                "begin": "2024-08-26T00:00:00",
7                "end": "2024-12-07T00:00:00"
8            },
9            "semester": "Fall",
10           "students": [
11               {
12                   "firstName": "Kateryna",
13                   "lastName": "Nesvit"
14               },
15               {
16                   "firstName": "Sapna",
17                   "lastName": "Surana"
18               },
19               {
20                   "firstName": "Jose",
21                   "lastName": "Pi"
22               }
23           ],
24           "year": 2024
25       }
26   }
```


© 2024 Pulp Free Press

Referring to example 15.6 — This JSON has been sorted and formatted for readability using the Visual Studio JSON: Sort Document feature. Note how the datetime strings are rendered in ISO 8601 format.

3.3 So, Tell Me Again Why I Need JSON?

There are other ways (i.e., Pickle, etc) to serialize Python objects for either data transmission or saving to disk, but they are not portable across systems, hardware, and programming languages. Valid, properly formatted JSON is platform agnostic.

Quick Review

JSON is the de facto standard for information interchange between systems, platforms, and programming languages. By following a few simple rules, you can build complex Python dictionary structures that can then be converted into valid, platform agnostic JSON.

4 Dictionary Use Cases

Dictionaries come in handy in many situations, some of which might surprise you. In this section, I'd like to introduce you to several dictionary use cases and provide a few examples. Table 15-3 lists a heaping handful of dictionary use cases.

Use Case	Discussion
General Mapping	Mapping keys to values. Provides a way to retrieve data based on a unique identifier. This is the basic dictionary use case.
Caching	Repeatedly accessed data or time-consuming calculations can be stored as values and accessed via a key
Application Configuration Settings	Application settings subject to change can be stored in a dictionary. The application accesses settings via a setting name key.
Application State	Similar to application configuration settings, a dictionary can be used to record last window location coordinates, theme settings, last used directory, etc.
JSON Data Generation	As discussed in the previous section, dictionaries make a great starting point for the formulation of valid, cross-platform compatible JSON.
Counting and Frequency Analysis	Counting occurrences of elements within a collection or the frequency distribution of items.
Feature Flags	Applications can use a dictionary to check if a particular feature is enabled or disabled. This is related to the Application Configuration Settings use case above.
Error Handling	Map error codes or messages to their corresponding descriptions or actions.

Table 15-3: Dictionary Uses Cases

 © 2024 Pulp Free Press

Referring to table 15-3 — This is not an exhaustive list by no means. For examples of how dictionaries are used within the Python language itself see: *https://github.com/python/cpython/blob/main/Objects/dictnotes.txt*

You'll encounter many of the dictionary use cases listed above as you progress through the book. Also, there are many places within the Python API where dictionaries are used to store and retrieve vital information. Example 15.7 briefly demonstrates the use of the `environ` dictionary found in the `os` module.

15.7 os_environ.py

```
1     """Demonstrates the use of the os.environ dictionary."""
2
3     import os
4
5     def main():
6         for key, value in os.environ.items():
7             print(f'{key} : {value}')
8
9
10    if __name__ == '__main__':
11        main()
12
```

Referring to example 15.7 — First, I import the `os` package. Next, on line 6, I iterate over the `os.environ` dictionary items and print each key and value. Figure 15-4 shows the results of running this program.

```
-bash
~/d/c/chapter15/os_environ    main +    5/29, 4:11 PM
Wed May 29 16:08:15 EDT 2024
~/dev/cst_with_python_1st_ed/chapter15/os_environ (main)
[530:30] swodog@macos-mojave-testbed $  python3 os_environ.py
TERM_PROGRAM : iTerm.app
TERM : xterm-256color
SHELL : /bin/bash
PIPENV_VENV_IN_PROJECT : true
TMPDIR : /var/folders/2q/vn6w1s5j5gxflzdjdttp2zxm0000gn/T/
TERM_PROGRAM_VERSION : 3.5.0
OLDPWD : /Users/swodog/dev/cst_with_python_1st_ed/chapter15
TERM_SESSION_ID : w0t0p0:CF5A9476-226A-44D3-B33C-8595278590DC
USER : swodog
SSH_AUTH_SOCK : /private/tmp/com.apple.launchd.5KdruxTwZp/Listeners
__CF_USER_TEXT_ENCODING : 0x1F5:0x0:0x0
TERM_FEATURES : T3LrMSc7UUw9Ts3BFGsSyHNoSxF
TERMINFO_DIRS : /Users/swodog/Applications/iTerm.app/Contents/Resources/terminfo:/usr/share/terminfo
PATH : ~/Applications/Visual Studio Code.app/Contents/Resources/app/bin:~/Applications/Sublime Text.app/Contents/SharedSupport/bin:/us
r/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin:/Users/swodog/Applications/iTerm.app/Contents/Resources/utilities
PWD : /Users/swodog/dev/cst_with_python_1st_ed/chapter15/os_environ
LANG : en_US.UTF-8
ITERM_PROFILE : Default
XPC_FLAGS : 0x0
ITERM_ORIG_PS1 : \n\[\033[35m\]$(/bin/date)\n\[\033[32m\]\w \[\033[1;33m\]$(__git_ps1 '(%s)')\n\[
\]\[\033[1;32m\][\!:\#]\[\033[1;33m\] \u@\h $ \[\e[0m\]
PS1 : \n\[\033[35m\]$(/bin/date)\n\[\033[32m\]\w \[\033[1;33m\]$(__git_ps1 '(%s)')\n\[
\]\[\033[1;32m\][\!:\#]\[\033[1;33m\] \u@\h $ \[\e[0m\]
XPC_SERVICE_NAME : 0
SHLVL : 1
HOME : /Users/swodog
COLORFGBG : 15;0
LC_TERMINAL_VERSION : 3.5.0
ITERM_PREV_PS1 : \[
\]\n\[\033[35m\]$(/bin/date)\n\[\033[32m\]\w \[\033[1;33m\]$(__git_ps1 '(%s)')\n\[
\]\[\033[1;32m\][\!:\#]\[\033[1;33m\] \u@\h $ \[\e[0m\] \[\]
ITERM_SESSION_ID : w0t0p0:CF5A9476-226A-44D3-B33C-8595278590DC
LOGNAME : swodog
LC_TERMINAL : iTerm2
COLORTERM : truecolor
_ : /usr/local/bin/python3
```

Figure 15-7: Results of Running Example 15.7

Referring to figure 15-7 — Notice how each key is in capital letters and `snake_case`. You'll find the `os` module quite useful as you expand your knowledge of Python programming.

Quick Review

The use cases for dictionaries are many and varied and range from general mapping applications where data can be indexed with unique keys, to application settings, caching, counting, and JSON data generation.

5 Parting Thoughts On Dictionaries

Python dictionaries are optimized for speed. They automatically resize to accommodate large sets of keys and values. They utilize a key/value structure where each key is unique and must be an immutable type, and values can be just about any type, including lists and dictionaries.

Dictionaries are mutable and unordered. The order of keys returned by the `dictionary.keys()` method is first-in-first-out (FIFO). To sort the keys upon retrieval use the `sorted()` built-in function.

Internally, dictionary entries are hashed for efficient retrieval of values. Dictionary elements are accessed via key. A dictionary *item* includes both the *key/value* pair. You can retrieve and iterate over dictionary items via the `dictionary.items()` method.

Summary

A dictionary (`dict`) is a Python mapping data type that lets you store and access data using *key/value* pairs. A *key* is used to index its associated data or *value*. **Keys** must be **immutable**, which limits the types of objects that can be used as keys. Values can be anything, including other dictionaries.

A key must be immutable because its value cannot change while it exists. Immutable objects provide a `__hash__()` function. The value calculated and returned by the `__hash__()` function must always be the same for a given object state.

You can create dictionaries in several different ways. The most often used approach is to first create an empty dictionary with an empty set of curly braces `"{}"` and set each key/value individually.

The `dictionary.items()` method returns an iterable set of key/value pairs.

Avoid the use of overly-complex dictionary comprehensions as they can lead to obfuscated code.

Dictionaries support a wide range of operations. Dictionary operations fall into three broad categories: operations concerning keys, operations concerning values, and operations concerning both keys and values (key/value pairs, otherwise referred to as items).

The methods `dictionary.`**`items()`**, `dictionary.`**`keys()`**, and `dictionary.`**`values()`** return dictionary view objects, which dynamically update when the underlying dictionary changes.

Place dictionary processing code in a `try/except` statement to properly handle cases where an attempted key access raises a KeyError exception.

© 2024 Pulp Free Press

JSON is the de facto standard for information interchange between systems, platforms, and programming languages. By following a few simple rules, you can build complex Python dictionary structures that can then be converted into valid, platform agnostic JSON.

The use cases for dictionaries are many and varied and range from general mapping applications where data can be indexed with unique keys, to application settings, caching, counting, and JSON data generation.

Skill-Building Exercises

1. **Deeper Exploration:** Explore the source code for the CPython `dictobject` implementation. This may be quite challenging if you are not familiar with the C programming language. Follow the code and figure out how the hash values are calculated and stored. The source code for the Python dictionary is located here: *https://github.com/python/cpython/blob/main/Objects/dictobject.c*

2. **Deeper Exploration:** Continuing with the study of the CPython `dictobject.c` source code file, draw the internal structure of a Python dictionary.

3. **Dictionary Use Cases:** Search online for examples of how Python dictionaries are used. Did you find any that were not mentioned in this chapter?

4. **Documentation Deep Dive:** Study the Python documentation at *https://docs.python.org/3/reference/index.html*. Focus on dictionaries and mapping types in general. Note any topics not covered in this chapter.

5. **Dictionary Operations:** Study table 15-1 and try out any dictionary operations not specifically demonstrated in this chapter.

6. **Dictionary Merging:** Explore different ways to merge two or more dictionaries, handling conflicts and duplicate keys appropriately.

7. **Dictionary Iteration:** Practice iterating over the keys, values, and key-value pairs of dictionaries using for loops and dictionary comprehensions.

8. **Dictionary Comprehensions:** Practice creating dictionaries with dictionary comprehensions. Search the Internet for examples of dictionary comprehensions.

9. **JSON Deep Dive:** Study the JSON Specification located at *https://www.json.org/json-en.html*

10. **Nested Dictionaries:** Practice creating and accessing dictionaries that contain nested lists and dictionaries. See example 15.5.

© 2024 Pulp Free Press

Suggested Projects

1. **Word Frequency Counter:** Create a program that reads a text file and generates a dictionary where the keys are words and the values are the frequencies of those words in the text.

2. **Contact Manager:** Build an interactive contact manager application that uses a dictionary to store contact information (e.g., name, phone number, email). Implement features such as adding contacts, searching for contacts, and deleting contacts.

3. **Inventory Management System:** Develop a system for managing inventory using dictionaries to store product information (e.g., name, price, quantity). Include features for adding new products, updating quantities, and generating reports.

4. **Language Translator:** Build a simple language translator using dictionaries to map words or phrases from one language to another. Users can input text, and the program will translate it based on the dictionary mappings.

5. **Dictionary Quiz Game:** Create a quiz game where users are presented with definitions or descriptions, and they have to guess the corresponding word. Use a dictionary to store the questions and answers, and track the user's score.

6. **Student Grade Tracker:** Develop a program for teachers to track student grades using dictionaries to store student names as keys and their corresponding grades as values. Implement features such as adding new grades, calculating averages, and generating reports.

7. **Weather Data Analysis:** Build a tool for analyzing historical weather data using dictionaries to store information such as temperature, precipitation, and date. Allow users to query the data for specific time periods or locations.

8. **Dictionary-based Password Generator:** Create a password generator that uses a dictionary of common words and phrases to generate strong, memorable passwords. Users can specify the length and complexity of the passwords.

9. **File Metadata Extractor:** Develop a utility that extracts metadata from files (e.g., images, documents) and stores it in a dictionary. Metadata could include file type, size, creation date, and author information.

10. **Anagram Finder:** Write a program that finds anagrams of a given word using a dictionary of words. Users can input a word, and the program will search the dictionary for all possible anagrams.

Self-Test Questions

1. What is a dictionary?

2. How do you create an empty dictionary?

3. How can you initialize a dictionary with key/value pairs?

4. How do you access the value associated with a specific key?

5. What happens if you try to access a key that does not exist in the dictionary?

6. What happens if you insert a value using a pre-existing key?

7. Can a dictionary have duplicate keys? Explain why or why not.

8. What does the dictionary.items() method do?

9. In what natural order are the keys retrieved via the dictionary.keys() method?

10. What two built-in functions can you use to change the ordering of keys retrieved from a dictionary?

References

Python Documentation, Dictionaries: *https://docs.python.org/3/tutorial/datastructures.html#dictionaries*

Python C API Documentation: *https://docs.python.org/3/c-api/dict.html*

ISO 8601-1:2029, iso.org: *https://www.iso.org/standard/70907.html*

JSON.org: *https://www.json.org/json-en.html*

CPython Repository, github.com: *https://github.com/python/cpython*

Notes