Cheatsheet

Dictionaries

Preliminaries

- Like a list, a dictionary is a type of data that stores zero or more other pieces of data. Each *value* that a dictionary stores is identified by a *key*. For example, in a dictionary that stores the sounds that animals make, the key `'cat'` identifies the value `'meow'` and the key `'pig'` identifies the value `'oink'`.
- To access or save a particular dictionary value, you need to know its key. Unlike lists, dictionaries have no notion of order between values that they store.
- Although Python permits you to use many different types as keys, we strongly recommend that you only use strings as keys.
- Within a dictionary, keys are unique. A key can only appear in a dictionary once.
- Dictionary values can be of any type. It is acceptable to use values of different types for different keys. For example, the key `'is_cat'` could have a boolean value (`True` or `False`) while the key `'name'` could have a string value (`'Pixie'` or `'Judge Posner'`).
- If two dictionaries are two different examples of the same type of data (i.e., two different Supreme Court cases), they should have the same schema.
- Dictionaries are reference types like lists.

Creating Dictionaries

A dictionary is created by enclosing zero or more key-value pairs (`'cat' : 'meow'` or `'pig' : 'oink'`) in curly brackets ({}) and separating them with commas. For example, the dictionary containing animal sounds would be written as

```
{'cat' : 'meow', 'pig' : 'oink'}
```

with keys `'cat'` and `'pig'` and their values `'meow'` and `'oink'` respectively.

Since dictionaries have no notion of order, the order in which the key-value pairs are written is of no consequence. The empty dictionary (a dictionary containing no items) is written as an empty pair of curly brackets {}.

Accessing Dictionary Elements

A value can be accessed from a dictionary by placing the corresponding key in square brackets following the name of the dictionary. This expression evaluates to the key's value.

```
>>> sounds = {'cat' :  'meow', 'pig' :  'oink' }
>>> sounds['cat']
'meow'
>>> sounds['pig']
```

```
'oink'
```

Modifying Dictionaries

A dictionary key can be set or modified to store a particular value by using the assignment (=) operator.

```
>>> sounds['dog'] = 'woof'
>>> sounds
{'cat' : 'meow', 'pig' : 'oink', 'dog' : 'woof'}
>>> sounds['cat'] = 'purr'
>>> sounds
{'cat' : 'purr','pig' : 'oink', 'dog' : 'woof'}
```

To the left of the = operator, write the name of the dictionary followed by square brackets containing the key. To the right of the = operator, write the corresponding value. If the key is already present, this new value will overwrite whichever value the dictionary currently stores.

To delete a key and its value from the dictionary, write the `del` keyword followed by a space and the name of the dictionary with the key to be deleted in square brackets.

```
>>> del sounds['cat']
>>> sounds
{'pig' : 'oink', 'dog' : 'woof'}
```

Working with Dictionaries

Length. The `len` function will evaluate to the number of values stored in a dictionary.

Testing for membership. The boolean `in` and `not in` operators test whether the key to the left of the operator is present in the dictionary to the right. If it is, the `in` operator evaluates to `True`; if it isn't, it evaluates to `False`. The `not in` operator has the opposite behavior.

```
>>> sounds
{'cat' : 'purr','pig' : 'oink', 'dog' : 'woof'}
>>> 'pig' in sounds
True
>>> 'cow' in sounds
False
```

Note that these operators check whether a *key* is present in the dictionary; it does not check for the presence of values.

```
>>> 'woof' in sounds
False
```

The `in` operator is far more efficient on dictionaries than on lists. If a list contains n values, then each use of the `in` operator will cause Python to examine $\frac{n}{2}$ elements of the list (on average). No matter how many keys a dic-

tionary contains, the `in` operator will cause Python to examine no more than a couple of keys, meaning it will complete almost instantaneously.

Default values. The `setdefault` method sets the value of a key if the key isn't already present in the dictionary; if it is, then the existing value is not modified. The method operates on a dictionary object and takes two arguments: a key and a value.

```
>>> sounds
{'cat' : 'purr', 'dog' : 'woof'}
>>> sounds.setdefault('cow', 'moo')
>>> sounds
{'cat' : 'purr', 'dog' : 'woof', 'cow' : 'moo'}
>>> sounds.setdefault('dog', 'bow wow')
>>> sounds
{'cat' : 'purr', 'dog' : 'woof', 'cow' : 'moo'}
```

The `get` method attempts to access the value corresponding to a particular key. If the key is found, the method call evaluates to the corresponding value. Otherwise, it evaluates to a default value provided to the function. The method operates on a dictionary object and takes two arguments: the key to be accessed and the default value.

```
>>> sounds
{'cat' : 'purr', 'pig' : 'oink', 'dog' : 'woof'}
>>> sounds.get('dog', 'bow wow')
'woof'
>>> sounds.get('cow', 'moo')
'moo'
```

The `get` method does not modify its dictionary object.

Converting to a list. The `keys` method produces a list-like type containing the dictionary's keys in an arbitrary order. It must be passed to the `list` function to convert it into a list.

```
>>> list(sounds.keys())
['dog', 'pig', 'cat']
```

The `values` method produces a list-like type containing the dictionary's values in the same order as the `keys` method.

```
>>> list(sounds.values())
['woof', 'oink', 'purr']
```

The `items` method produces a list-like type containing key-value tuples of the dictionary's elements.

```
>>> list(sounds.items())
[('dog', 'woof'), ('pig', 'oink'), ('cat', 'purr')]
```

You can iterate over these lists using for-loops.

Deduplication. You can use a dictionary to remove the duplicates from a list of elements. Add the elements to a dictionary as the keys. Since a dictionary cannot contain duplicate keys, the dictionary's keys will contain the deduplicated collection of elements. You can turn this back into a list using the `keys` method.

JSON

JSON is a way of representing in string form any datastructure constructed from nested lists, nested dictionaries, and the five fundamental types we introduced in Chapter 3 (integers, floats, strings, booleans, and the none type).

Python comes with a library called `json` that provides a function for converting a Python datastructure into a JSON string (`json.dumps`) and a JSON string into a Python datastructure (`json.loads`). Before you can use these functions, you need to insert the statement

```
import json
```

Saving a Python datastructure to a JSON file. Assume that the datastructure is stored in the variable `datastructure` and we want to save the datastructure to the file `datastructure.json`.

1. Convert the datastructure into a JSON string.

```
>>> datastructure_json = json.dumps(datastructure)
```

2. Write the string `datastructure_json` to a text file called `datastructure.json`.

```
>>> fh = open('datastructure.json', 'w'
>>> fh.write(datastructure_json)
>>> fh.close()
```

Loading a Python datastructure from a JSON file. Assume that the JSON version datastructure is stored in the file `datastructure.json` and we want to restore the Python version to the variable `datastructure`.

1. Open the file in read-mode, read the contents of the JSON string, and close the file.

```
>>> fh = open('datastructure.json')
>>> datastructure_json = fh.read()
>>> fh.close()
```

2. Reconstitute the Python datastructure.

```
>>> datastructure = json.loads(datastructure_json)
```

Datastructures

A *datastructure* is a combination of nested lists, dictionaries, and other types used to represent some dataset (like a Supreme Court case or the United States Code). To design a datastructure, start at the very highest level and break your datastructure down into smaller pieces. Determine whether each piece should be represented as a list, a dictionary, or another type. If each of these pieces have smaller subcomponents, repeat the process for these smaller elements using nested lists, dictionaries, or other types. As you choose the appropriate representation for your dataset, consider how you intend to use it in your program and aim to make doing so as easy as possible.

Keywords

Accessing—The act of obtaining a value from a dictionary using the requisite key.

Datastructure—A combination of nested lists, dictionaries, and other types used to represent a dataset.

Data cleaning—The process of refining data from an outside source to remove inconsistencies and put it in a format such that it can be processed easily by a program.

Deduplication—The process of removing all duplicates from a collection of items.

Deserialize—To reconstitute a Python datastructure from a string. The inverse of serializing. Example: `json.loads` deserializes JSON strings into Python datastructures.

Eagerly—Performing an operation proactively.

Efficiency—The resources necessary for a program to complete its task. Efficiency is generally measured in the number of operations your program has to perform (which equates with the time necessary for the program to run) and the amount of space necessary to perform those operations.

Indexing—In the context of dictionaries, see: accessing.

Key—In the context of a dictionary, the name used to refer to a specific value that a dictionary stores.

Lazily—Performing an operation reactively at the last possible moment.

Library—A bundle of related Python functions that your program can import and call.

List-like type—A special Python type that is different from a list but still contains an ordered collection of elements and supports some of the same operations as lists. It can be converted into a list by passing it to the `list` function.

Schema—A dictionary's keys and the types of the corresponding values. This information is sufficient to write a program that operates on any dictionary configured in this fashion.

Serialize—To convert a Python datastructure of any kind into a string (or any other format where it can be saved to a file). Example: `json.dumps` serializes Python datastructures into JSON strings.

Set—A collection of items that contains no duplicates. You can mimic a set using a dictionary's keys.

Value—In the context of a dictionary, the element that a dictionary stores. Each value is referred to by a unique key.