

Dictionaries,  
Or,  
How to be *really* Pythonic

October 26, 2022

# Administrative Notes

Remember that Homework 1 is out

It's due Monday, November 7 before midnight

Don't wait until the last minute to get it started

# For Project 1: Importing functions from other files

In Python, you can import any function from any registered module or python file you wrote

# Importing modules and functions in Python

Python comes with some “builtin” functions such as `len()`, `print()`, `input()`,...

There are tons of other functions that have already been written by others, and which are free to you to use in your programming career.

- There's no need to rewrite a function if you know somebody else has already written it

You get access to that code by using the `import()` function

# import()

import() tells the Python interpreter that you want access to a module that you know about, and the functions in that module

A “module” is Pythonic for a group of functions made available. Other languages might call this a “library” or a “package.”

Import random

Imports a module that contains a bunch of functions all related to the generation and management of “random” numbers

Note: the module must be present on your computer for “import” to work. If you get an error message saying the module does not exist, you’ll have to install it.

# Using a function in a module

Once you have imported a module, you can use its functions in your program

```
random.randint(1,25)
```

Generates a random integer between 1 and 25, inclusive

You can use this just like any other function:

```
for i in range(10):  
    r_num = random.randint(1,25)  
    print(r_num)
```

# How do you know what functions are in a module?


...and what parameters to use to call them?

This is where the ability to search the web is your friend. :-)

All the common modules are documented out there in Python-land, along with their Application Programming Interfaces (APIs)

- Which is a fancy way of saying “descriptions of how to call a function, what the parameters are, what the parameters mean and what the return values are.”

# What's a dictionary? *(from dictionary.com)*

**dictionary** [ dik-shuh-ner-ee ] [SHOW IPA](#) 

[SEE SYNONYMS FOR dictionary ON THESAURUS.COM](#)

*noun, plural dic-tion-ar-ies.*

- 1 a book, optical disc, mobile device, or online lexical resource (such as Dictionary.com) containing a selection of the words of a language, giving information about their meanings, pronunciations, etymologies, inflected forms, derived forms, etc., expressed in either the same or another language; lexicon; glossary. Print dictionaries of various sizes, ranging from small pocket dictionaries to multivolume books, usually sort entries alphabetically, as do typical CD or DVD dictionary applications, allowing one to browse through the terms in sequence. All electronic dictionaries, whether online or installed on a device, can provide immediate, direct access to a search term, its meanings, and ancillary information:  
*an unabridged dictionary of English; a Japanese-English dictionary.*
- 2 a book giving information on particular subjects or on a particular class of words, names, or facts, usually arranged alphabetically:  
*a biographical dictionary; a dictionary of mathematics.*
- 3 *Computers.*
  - a a list of codes, terms, keys, etc., and their meanings, used by a computer program or system.
  - b a list of words used by a word-processing program as the standard against which to check the spelling of text entered.



The definition we care about

“A list of codes, terms, keys, etc. and their meanings, used by a computer program or system”

-it's not just meanings. More generally, it's a set of *values* that we associate with the *key*.

# In Python syntax

A dictionary, or *dict* for short, is enclosed in curly braces { }

There is a set of **keys**; each key is followed by a colon : and then its **value**

Key/value pairs in the dictionary are separated by commas ,

$\text{dict} = \{ \text{key}_1:\text{value}_1, \text{key}_2:\text{value}_2, \dots, \text{key}_n:\text{value}_n \}$

## An example

```
dogs = { "doug": ["beagle","bulldog"], "lizzie": "Australian cattle dog", "penny":  
"spaniel", "bonnie":beagle, "remy": "shepherd" }
```

The **keys** in this dictionary are the strings that represent the names of the dogs: “doug”, “lizzie”, “penny”, “bonnie” and “remy”.

The **values** in this dictionary are strings that represent the breed of the dog. Doug is a beagle/bulldog mix, so his value - his breed - is a list of the strings “beagle” and “bulldog.” Lizzie is an Australian cattle dog, so her value is that string. And so on.

# Rules for keys and values

**Keys** must be *unique* and *immutable*. - ints, floats, strings, booleans - NOT lists or dicts\*

- Keys can only appear once in a dictionary. In the dog example, it would be illegal to list key “doug” with two different values. {“doug”:”beagle”, “doug”:”bulldog”} is not permitted. That’s why we put the two values in a list
- Keys have to be ints, floats, booleans, or strings. Keys cannot be lists or other dictionaries

**Values** can be anything. There are no practical restrictions.

- Values can be mutable or immutable. Values can be lists, ints, floats, booleans, strings... Values can even be dictionaries. Yes, you can nest a dictionary inside another dictionary.
- Values can be repeated. I used to have two beagles, Baron and Cleo. So they would be entered in a dictionary like this:

```
old_dogs = {“Baron”:”beagle”, “Cleo”:”beagle”}
```

***\*yes, dictionaries, like lists, are mutable. You can change their value and keep them in the same location in memory***

# How do you create a dictionary?

Create an empty dictionary:

```
cats = { }
```

Create a dictionary with content (the dogs dictionary is one example of this)

```
governors = {"Maryland": "Hogan", "Virginia": "Youngkin", "Louisiana": "Edwards"}
```

`governors[0]` - does not exist - "the first entry in governors" - don't know what that is

# Dictionaries are ***unordered***

**Ordered** means that there is a specific relationship between the elements in a structure. Each element has a location relative to other elements in the structure

**Lists** are **ordered**. Each list element has an index. We can specify a value in a list by giving its index. `student[0]`, `student[1]`, `student[len(student)-1]`

Dictionaries are not ordered. There is no “first element” in a dictionary. No “first key”, no “first value.” You can’t reference an element in a dictionary by its location.

`dogs{0}` is undefined. You’ll get an error if you try to reference it.

# So how do you access elements of a dictionary?

By key value only. Put the key in square brackets:

`dogs['doug']` gets you the list `['beagle', 'bulldog']`

`dogs['remy']` gets the string `'shepherd'`

What if that key's not present?

`dogs['pepper']`

***Operations on dictionaries are easy ways to create bugs that crash your program. “Defensive programming” is in order - use methods that are resilient to bugs.***

# The ***get*** method

Another way to access a dictionary element that's more resilient to errors

```
dogs.get('pepper')
```

If that key doesn't exist, the method will return `None` (that special value we talked about) instead of erroring out. So your program can continue rather than crashing.

You can make it return something other than `None`, if you want

```
dogs.get('pepper', 'Key not found')
```

```
dogs.get('pepper', -1)
```



# Adding to a dictionary

To add a new key/value pair, just use an assignment statement: `dict[key] = value`

```
dogs['cinder'] = 'doberman'
```

Since the dictionary has no order, there's no 'append' or 'insert.' You can't control where in the dictionary this goes

Also: if that key already exists, this statement overwrites the value associated with that key

```
dogs['penny'] = 'cocker spaniel'
```

doesn't add a new element to the dictionary, it just changes the value of the existing key.

# Removing an element from a dictionary

```
del dict[key]
```

In the dogs example:

```
del dogs['cinder']
```

Removes the entire key/value pair associated with 'cinder'

But again, if that key doesn't exist in the dictionary, you'll error out. Your program may crash.

```
del dogs['lady']
```

```
if 'lady' in dogs.keys():
```

```
    del dogs['lady'] # gets rid of key and value pair
```

# A more resilient way to delete - the pop method

```
dogs.pop('lady')
```

Still causes an error if 'lady' is not a key in the dictionary

But you can specify a default value to return if the key isn't there

```
dogs.pop('lady', None)
```

OR

```
dogs.pop('lady', -1)    etc.
```

# What's the difference between a dictionary and a list?

1. A list establishes a binding between a value and its location in the list; a dictionary establishes a binding between a key and a value
2. A list is ordered; a dictionary is unordered

# When would you use a dictionary?

Use a list if you have an ordered collection of items - a list of student IDs; a list of grades; etc.

Use a dictionary if you have a set of unique keys that each map to a value.

The dog example - we have a set of dog names ('keys') that each map to one or more breeds of dogs ('values'). It would be awkward to use lists to store dog names and breeds and maintain the relationship.

- You could do it, but it would be awkward

# Another example: a recipe

Here's a recipe for chocolate chip cookies (courtesy AltonBrown.com)

Let's make a dictionary to hold this recipe. The keys will be the ingredients: butter, eggs, flour. The values will be the quantity I need of each ingredient

```
recipe = { 'butter':[2,'sticks'], 'flour':[1.5, 'cups'], 'salt':[1, 'teaspoon'],  
          'Soda':[1,'teaspoon'], 'Granulated sugar':[0.25, 'cup'], 'brown  
sugar':[1,'cup'], 'egg':2, 'milk':[2,'tablespoons'], 'vanilla':[1.5, 'teaspoon'],  
          'Chocolate chips':[12,'ounces']}
```

Or a simpler version:

```
recipe = {'butter':2, 'flour':1.5, 'salt':1, 'Soda':1, 'Granulated sugar':0.25, 'brown sugar':1,  
          'egg':2, 'milk':2, 'vanilla':1.5, 'Chocolate chips':12}
```

- 8 ounces unsalted butter
- 12 ounces bread flour
- 1 teaspoon kosher salt
- 1 teaspoon baking soda
- 2 ounces granulated sugar
- 8 ounces light brown sugar
- 1 large egg
- 1 large egg yolk
- 2 tablespoons whole milk
- 1 1/2 teaspoons vanilla extract
- 12 ounces semisweet chocolate chips

# How do we use this dictionary in a program?

```
eggs = int(input("how many eggs do you have?"))
if eggs < recipe["egg"]:
    print("Add a dozen eggs to your grocery list")

chips = int(input("how many ounces of chocolate chips do you have?"))
if chips < recipe["Chocolate chips"]:
    print("Get a bag of chocolate chips at the store")
```

# Creating a dictionary from two lists

## Python “zip” function and “dict” constructor method

tl;dr:

```
a = [1,2,3]
b = [4,5,6]
d = dict(zip(a,b))
print(d)
[1:4, 2:5, 3:6]
```

The longer story: “zip” is a function that takes multiple iterable values (lists) and interweaves them, one element at a time. One element from list a, then one from list b, then the next one from a, then from b... until the shortest list is done. The return value from zip() is an iterator that can be used to build other structures. The return value from zip() is not meaningful to the average human.

“dict” is a constructor method that operates on an iterator and returns a dictionary. It’s analogous to the “list” method that creates lists.

Here, “zip(a,b)” weaves together the values 1,4,2,5,3,6 and returns an iterator built from them. Then “dict(zip(a,b))” takes that iterator and turns it into the dictionary you see. The first item becomes a key; the second its associated value. The third item becomes the second key; the fourth is its associated value. And so on.



# Changing the value associated with a key

The easiest way is to just use another assignment statement

- Oops, a DNA test shows that Doug is a beagle-boxer mix, not beagle-bulldog

```
dogs[ 'doug' ] = [ 'beagle', 'boxer' ]
```

But depending on the type of the value, you can use any valid Python function or method for that type

```
dogs[ 'doug' ]
```

- the value is a list; we can remove a value from the list and then append a new one

```
dogs[ 'doug' ].remove( 'bulldog' )
```

```
dogs[ 'doug' ].append( 'boxer' )
```

# Can you have a key with no value?

No, you can't

- But, you CAN have a key with the value None
- Our old friend None, of type NoneType

# ‘Keys’ and ‘values’

Pre-defined methods that operate on dictionaries and return “view objects” containing, respectively, the list of keys in the dictionary and the list of values in the dictionary

```
dogs.keys()
```

```
dogs.values()
```

You can use these methods in your programming - to do some error checking, for example