# Dictionaries

## Definition and Use of Dictionaries

In common usage, a dictionary is a collection of words matched with their definitions. Given a word, you can look up its definition. Python has a built in dictionary type called dict which you can use to create dictionaries with arbitrary definitions for character strings. It can be used for the common usage, as in a simple English-Spanish dictionary.

Look at the example program spanish1.py and run it.

*"""A tiny English to Spanish dictionary is created,*

*using the Python dictionary type dict.*

*Then the dictionary is used, briefly.*

*"""*

spanish = dict()

spanish['hello'] = 'hola'

spanish['yes'] = 'si'

spanish['one'] = 'uno'

spanish['two'] = 'dos'

spanish['three'] = 'tres'

spanish['red'] = 'rojo'

spanish['black'] = 'negro'

spanish['green'] = 'verde'

spanish['blue'] = 'azul'

print(spanish['two'])

print(spanish['red'])

First an empty dictionary is created using dict(), and it is assigned the descriptive name spanish.

To refer to the definition for a word, you use the dictionary name, follow it by the word inside square brackets. This notation can either be used on the left-hand side of an assignment to make (or remake) a definition, or it can be used in an expression (as in the print functions), where its earlier definition is retrieved. For example,

spanish['hello'] = 'hola'

makes an entry in our spanish dictionary for 'hello' , with definition 'hola'.

print(spanish['red'])

retrieves the definition for 'red', which is 'rojo'.

Since the Spanish dictionary is defined at the top-level, the variable name spanish is still defined after the program runs: After running the program, usespanish in the Shell to check out the translations of some more words, other than 'two' and 'red'.

Creating the dictionary is a well defined and quite different activity from the use of the dictionary at the end of the code, so we can use a functions to encapsulate the task of creating the dictionary, as in the example program spanish2.py, which gives the same result:

*"""A tiny English to Spanish dictionary is created,*

*using the Python dictionary type dict.*

*Then the dictionary is used, briefly.*

*"""*

**def** createDictionary():

*'''Returns a tiny Spanish dictionary'''*

spanish = dict()

spanish['hello'] = 'hola'

spanish['yes'] = 'si'

spanish['one'] = 'uno'

spanish['two'] = 'dos'

spanish['three'] = 'tres'

spanish['red'] = 'rojo'

spanish['black'] = 'negro'

spanish['green'] = 'verde'

spanish['blue'] = 'azul'

**return** spanish

**def** main():

dictionary = createDictionary()

print(dictionary['two'])

print(dictionary['red'])

main()

This code illustrates several things about functions.

* First, like whole files, functions can have a documentation string immediately after the definition heading. It is a good idea to document the return value!
* The dictionary that is created is returned, but the local variable name in the function, spanish, is lost when the function terminates.
* To remember the dictionary returned to main, it needs a name. The name does not have to match the name used in createDictionary. The namedictionary is descriptive.

We could also use the dictionary more extensively. The example program spanish2a.py is the same as above except it has the following main method:

**def** main():

dictionary = createDictionary()

print('Count in Spanish: ' + dictionary['one'] + ', ' +

dictionary['two'] + ', ' +dictionary['three'] + ', ...')

print('Spanish colors: ' + dictionary['red'] + ', ' +

dictionary['blue'] + ', ' +dictionary['green'] + ', ...')

Try it, and check that it makes sense.

Python dictionaries are actually more general than the common use of dictionaries. They do not have to associate words and their string definitions. They can associate many types of objects with some arbitrary object. The more general Python terminology for word and definition are key and value. Given a key, you can look up the corresponding value. The only restriction on the key is that it be an immutable type. This means that a value of the key’s type cannot be changed internally after it is initially created. Strings and numbers are immutable. A dictionary is mutable: its value can be changed internally. (You can add new definitions to it!) We will see more mutable and immutable types later and explore more of the internal workings of data types.

### Number Dictionary Exercise

Write a tiny Python program numDict.py that makes a dictionary whose keys are the words ‘one’, ‘two’, ‘three’, and ‘four’, and whose corresponding values are the numerical equivalents, 1, 2, 3, and 4 (ints, not strings). Include code to test the resulting dictionary by referencing several of the definitions and printing the results.

## Dictionaries and String Formatting

At the end of the main function in spanish2a.py from the last section, two strings are constructed and printed. The expressions for the two strings include a sequence of literal strings concatenated with interspersed values from a dictionary. There is a much neater, more readable way to generate these strings. We will develop this in several steps. The first string could be constructed and printed as follows:

numberFormat = 'Count in Spanish: {one}, {two}, {three}, ...'

withSubstitutions = numberFormat.format(one='uno', two='dos', three='tres')

print(withSubstitutions)

There are several new ideas here!. We are using an alternate form of format string and format method parameters from those described in [String Format Operation](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/io.html#format-strings).

Note the form of the string assigned the name numberFormat: It has the English words for numbers in braces where we want the Spanish definitions substituted. (This is unlike in [String Format Operation](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/io.html#format-strings), where we had empty braces or a numerical index inside.)

As in [String Format Operation](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/io.html#format-strings), the second line uses method calling syntax. A reminder of the syntax for methods:

object.methodname(parameters)

has the object followed by a period followed by the method name, and further parameters in parentheses.

In the example above, the object is the string called numberFormat. The method is named format. The parameters in this case are all keyword parameters. You have already seen keyword parameters sep and end used in print function calls. In this particular application, the keywords are chosen to include all the words that appear enclosed in braces in the numberFormat string.

When the string numberFormat has the format method applied to it with the given keyword parameters, a new string is created with substitutions into the places enclosed in braces. The substitutions are just the values given by the keyword parameters. Hence the printed result is

Count in Spanish: uno, dos, tres, ...

Now we go one step further: The keyword parameters associate the keyword names with the values after the equal signs. The dictionary fromspanish2a.py includes exactly the same associations. There is a special notation allowing such a dictionary to supply keyword parameters. Assumingdictionary is the Spanish dictionary from spanish2a.py, the method call

numberFormat.format(one='uno', two='dos', three='tres')

returns the same string as

numberFormat.format(\*\*dictionary)

The special syntax \*\* before the dictionary indicates that the dictionary is not to be treated as a single actual parameter. Instead keyword arguments forall the entries in the dictionary effectively appear in its place.

Below is a substitute for the main method in spanish2a.py. The whole revised program is in example program spanish3.py:

**def** main():

dictionary = createDictionary()

numberFormat = "Count in Spanish: {one}, {two}, {three}, ..."

withSubstitutions = numberFormat.format(\*\*dictionary)

print(withSubstitutions)

print("Spanish colors: {red}, {blue}, {green}, ...".format(\*\*dictionary))

In this main function the string with the numbers is constructed in steps as discussed above. The printing of the string with the Spanish colors is coded more concisely. There are not named variables for the format string or the resulting formatted string. You are free to use either coding approach.

In general, use this syntax for the string format method with a dictionary, returning a new formatted string:

formatString.format(\*\*aDictionary)

where the format string contains dictionary keys in braces where you want the dictionary values substituted. The dictionary key names must follow the rules for legal identifiers.

At this point we have discussed in some detail everything that went into the first sample program, madlib.py, in [A Sample Program, Explained](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/idle.html#madlib-explained-i)! This is certainly the most substantial program so far.

Look at madlib.py again, see how we have used most of the ideas so far. If you want more description, you might look at section [A Sample Program, Explained](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/idle.html#madlib-explained-i) again (or for the first time): it should make much more sense now.

We will use madlib.py as a basis for more substantial modifications in structure in [The Revised Mad Lib Program](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/madlib2.html#the-revised-mad).

## Dictionaries and Python Variables

Dictionaries are central to the implementation of Python. Each variable identifier is associated with a particular value. These relationships are stored in dictionaries in Python, and these dictionaries are accessible to the user: You can use the function call locals() to return a dictionary containing all the current local variables names as keys and all their values as the corresponding dictionary values. This dictionary can be used with the string format method, so you can embed local variable names in a format string and use them very easily!

For example, run the example program arithDict.py

*'''Fancier format string example, with locals().'''*

x = 20

y = 30

sum = x + y

prod = x \* y

formatStr = '{x} + {y} = {sum}; {x} \* {y} = {prod}.'

equations = formatStr.format(\*\*locals())

print(equations)

Note the variable names inside braces in formatStr, and the dictionary reference used as the format parameter is \*\*locals().

A string like formatStr is probably the most readable way to code the creation of a string from a collection of literal strings and program values. The ending part of the syntax, .format(\*\*locals()), may appear a bit strange, but it is very useful! We will use this notation extensively. It clearly indicate how values are embedded into the format string, and avoids having a long list of parameters to format.

The example program hello\_you4.py does the same thing as the earlier hello\_you versions, but with a dictionary reference:

*'''Hello to you! Illustrates locals() for formating in print.*

*'''*

person = input('Enter your name: ')

greeting = 'Hello, {person}!'.format(\*\*locals())

print(greeting)