Dictionaries

All of the compound data types we have studied in detail so far — strings, lists, and tuples — are sequential collections. This means that the items in the collection are ordered from left to right and they use integers as indices to access the values they contain.

Dictionaries are a different kind of collection. They are Python's built-in **mapping type**. A map is an unordered, associative collection. The association, or mapping, is from a **key**, which can be any immutable type, to a **value**, which can be any Python data object.

As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings and the values will also be strings.

One way to create a dictionary is to start with the empty dictionary and add **key-value pairs**. The empty dictionary is denoted `{}`

```
1  eng2sp = {}
2  eng2sp['one'] = 'uno'
3  eng2sp['two'] = 'dos'
4  eng2sp['three'] = 'tres'
```

<< First | < Back | Step 1 of 4 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

CodeLens: 1 (chp12_dict1)

The first assignment creates an empty dictionary named `eng2sp` . The other assignments add new key-value pairs to the dictionary. The left hand side gives the dictionary and the key being associated. The right hand side gives the value being associated with that key. We can print the current value of the dictionary in the usual way. The key-value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

The order of the pairs may not be what you expected. Python uses complex algorithms, designed for very fast access, to determine where the key-value pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredictable.

Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output.

```
1 eng2sp = {'three': 'tres', 'one': 'uno', 'two': 'dos
2 print(eng2sp)
```

<< First | < Back | Step 1 of 2 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

CodeLens: 2 (chp12_dict2)

It doesn't matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so there is no need to care about ordering.

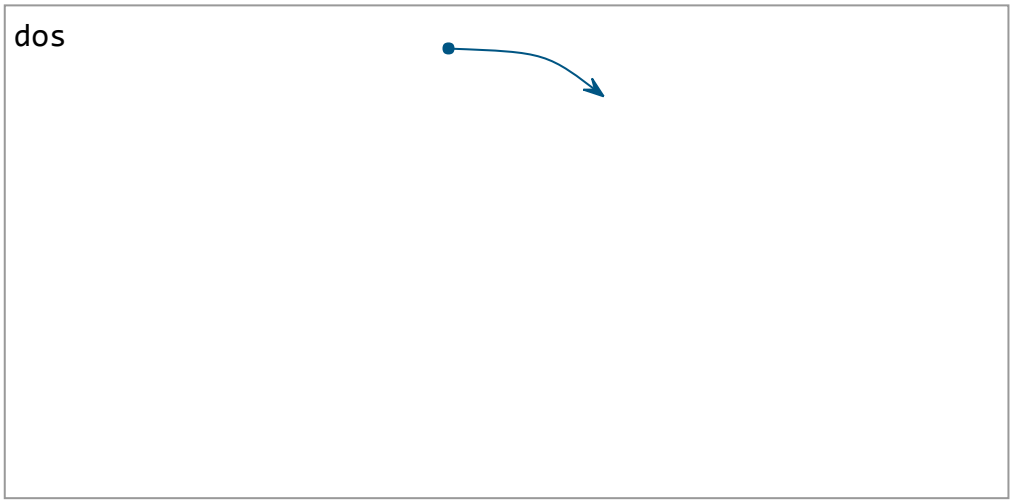
Here is how we use a key to look up the corresponding value.

```
1 eng2sp = {'three': 'tres', 'one': 'uno', 'two': 'dos
2
3 value = eng2sp['two']
4 print(value)
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed
next line to execute

Program output:



CodeLens: 3 (chp12_dict3)

The key `'two'` yields the value `'dos'`.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```
1
2
```

ActiveCode: 1 (scratch_11_01)

Run

Check your understanding

dict-1: A dictionary is an unordered collection of key-value pairs.

a) False

b) True

Check Me | Compare Me

Correct!! Yes, dictionaries are associative collections meaning that they store key-value pairs.

dict-2: What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23}
print(mydict["dog"])
```

a) 12

b) 6

c) 23

d) Error, you cannot use the index operator with a dictionary.

Check Me | Compare Me

Correct!! Yes, 6 is associated with the key dog.

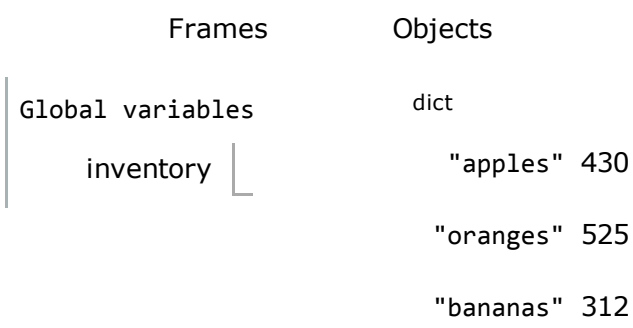
Dictionary operations

The `del` statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock. If someone buys all of the pears, we can remove the entry from the dictionary.

```
1 inventory = {'apples': 430, 'bananas': 312, 'orange
2
3 del inventory['pears']
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed
next line to execute



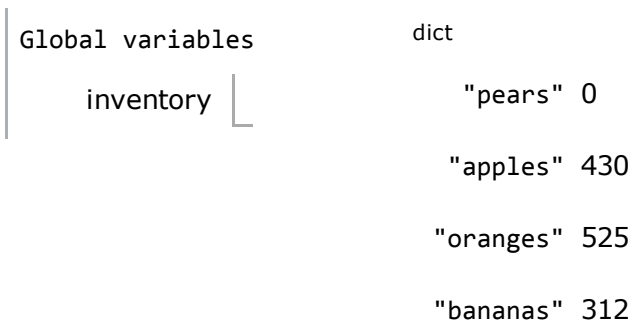
CodeLens: 4 (ch12_dict4)

Dictionaries are also mutable. As we've seen before with lists, this means that the dictionary can be modified by referencing an association on the left hand side of the assignment statement. In the previous example, instead of deleting the entry for `pears`, we could have set the inventory to `0`.

```
1 inventory = {'apples': 430, 'bananas': 312, 'orange
2
3 inventory['pears'] = 0
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed
next line to execute



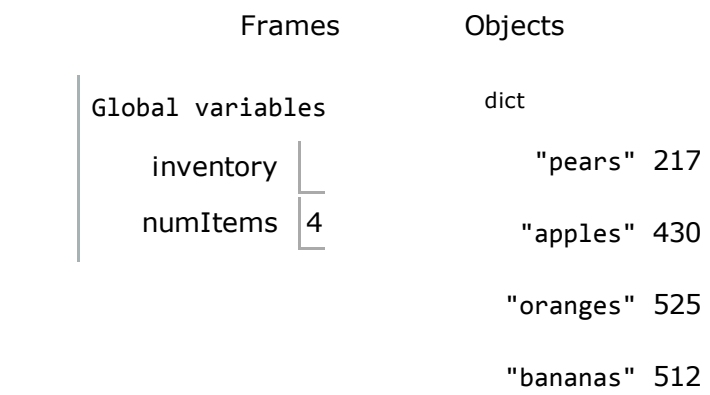
CodeLens: 5 (ch12_dict4a)

Similarily, a new shipment of 200 bananas arriving could be handled like this.

```
1  inventory = {'apples': 430, 'bananas': 312, 'orange
2  inventory['bananas'] = inventory['bananas'] + 200
3
4
5  numItems = len(inventory)
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed
next line to execute



CodeLens: 6 (ch12_dict5)

Notice that there are now 512 bananas—the dictionary has been modified. Note also that the `len` function also works on dictionaries. It returns the number of key-value pairs:

Check your understanding

dict-3: What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23}
mydict["mouse"] = mydict["cat"] + mydict["dog"]
print(mydict["mouse"])
```

- a) 12
- b) 0
- c) 18
- d) Error, there is no entry with mouse as the key.

Check Me | Compare Me

Correct!! Yes, add the value for cat and the value for dog (12 + 6) and create a new entry for mouse.

Dictionary methods

Dictionaries have a number of useful built-in methods. The following table provides a summary and more details can be found in the Python Documentation (http://docs.python.org/py3k/library/stdtypes.html#mapping-types-dict).

Method	Parameters	Description
keys	none	Returns a view of the keys in the dictionary
values	none	Returns a view of the values in the dictionary
items	none	Returns a view of the key-value pairs in the dictionary
get	key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise

The `keys` method returns what Python 3 calls a **view** of its underlying keys. We can iterate over the view or turn the view into a list by using the `list` conversion function.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

for akey in inventory.keys():     # the order in which we get the keys is not defined
   print("Got key", akey, "which maps to value", inventory[akey])

ks = list(inventory.keys())
print(ks)

```

ActiveCode: 2 (chp12_dict6)

Run

```
Got key apples which maps to value 430
Got key bananas which maps to value 312
Got key oranges which maps to value 525
Got key pears which maps to value 217
['apples', 'bananas', 'oranges', 'pears']
```

It is so common to iterate over the keys in a dictionary that you can omit the `keys` method call in the `for` loop — iterating over a dictionary implicitly iterates over its keys.

```
1 inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
2
3 for k in inventory:
4    print("Got key", k)
5
```

ActiveCode: 3 (chp12_dict7)

Run

```
Got key apples
Got key bananas
Got key oranges
Got key pears
```

As we saw earlier with strings and lists, dictionary methods use dot notation, which specifies the name of the method to the right of the dot and the name of the object on which to apply the method immediately to the left of the dot. The empty parentheses in the case of `keys` indicate that this method takes no parameters.

The `values` and `items` methods are similar to `keys` . They return view objects which can be turned into lists or iterated over directly. Note that the items are shown as tuples containing the key and the associated value.

```
1  inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
2
3  print(list(inventory.values()))
4  print(list(inventory.items()))
5
6  for (k,v) in inventory.items():
7      print("Got",k,"that maps to",v)
8
9  for k in inventory:
10     print("Got",k,"that maps to",inventory[k])
11
```

ActiveCode: 4 (chp12_dict8)

Run

```
[430, 312, 525, 217]
[('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
Got apples that maps to 430
Got bananas that maps to 312
Got oranges that maps to 525
Got pears that maps to 217
Got apples that maps to 430
Got bananas that maps to 312
Got oranges that maps to 525
Got pears that maps to 217
```

Note that tuples are often useful for getting both the key and the value at the same time while you are looping. The two loops do the same thing.

The `in` and `not in` operators can test if a key is in the dictionary:

```
1 inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
2 print('apples' in inventory)
3 print('cherries' in inventory)
4
5 if 'bananas' in inventory:
6     print(inventory['bananas'])
7 else:
8     print("We have no bananas")
9
```

ActiveCode: 5 (chp12_dict9)

Run

```
True
False
312
```

This operator can be very useful since looking up a non-existent key in a dictionary causes a runtime error.

The `get` method allows us to access the value associated with a key, similar to the `[ ]` operator. The important difference is that `get` will not cause a runtime error if the key is not present. It will instead return None. There exists a variation of `get` that allows a second parameter that serves as an alternative return value in the case where the key

is not present. This can be seen in the final example below. In this case, since “cherries” is not a key, return 0 (instead of None).

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

print(inventory.get("apples"))
print(inventory.get("cherries"))

print(inventory.get("cherries",0))

```

ActiveCode: 6 (chp12_dict10)

Run

```
430
None
0
```

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```

```

ActiveCode: 7 (scratch_11_02)

Run

Check your understanding

dict-4: What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
keylist = list(mydict.keys())
keylist.sort()
print(keylist[3])
```

- a) cat
- b) dog
- c) elephant
- d) bear

Check Me | Compare Me

Correct!! Yes, the list of keys is sorted and the item at index 3 is printed.

dict-5: What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
answer = mydict.get("cat")//mydict.get("dog")
print(answer)
```

- a) 2
- b) 0.5
- c) bear
- d) Error, divide is not a valid operation on dictionaries.

Check Me | Compare Me

Correct!! get returns the value associated with a given key so this divides 12 by 6.

dict-6: What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
print("dog" in mydict)
```

- a) True
- b) False

Check Me | Compare Me

Correct!! Yes, dog is a key in the dictionary.

dict-7: What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
print(23 in mydict)
```

- a) True
- b) False

Check Me | Compare Me

Correct!! Yes, the in operator returns True if a key is in the dictionary, False otherwise.

dict-8: What is printed by the following statements?

```
total = 0
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
for akey in mydict:
   if len(akey) > 3:
      total = total + mydict[akey]
print(total)
```

- a) 18
- b) 43
- c) 0
- d) 61

Check Me | Compare Me

Correct!! Yes, the for statement iterates over the keys. It adds the values of the keys that have length greater than 3.

Aliasing and copying

Because dictionaries are mutable, you need to be aware of aliasing (as we saw with lists). Whenever two variables refer to the same dictionary object, changes to one affect the other. For example, `opposites` is a dictionary that contains pairs of opposites.

```
opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
alias = opposites

print(alias is opposites)

alias['right'] = 'left'
print(opposites['right'])

```

ActiveCode: 8 (ch12_dict11)

Run

```
True
left
```

As you can see from the `is` operator, `alias` and `opposites` refer to the same object.

If you want to modify a dictionary and keep a copy of the original, use the dictionary `copy` method. Since *acopy* is a copy of the dictionary, changes to it will not effect the original.

```
acopy = opposites.copy()
acopy['right'] = 'left'    # does not change opposites
```

Check your understanding

dict-9: What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
yourdict = mydict
yourdict["elephant"] = 999
print(mydict["elephant"])
```

- a) 23
- b) None
- c) 999

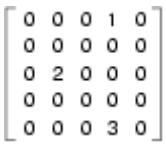
d) Error, there are two different keys named elephant.

Check Me | Compare Me

Correct!! Yes, since yourdict is an alias for mydict, the value for the key elephant has been changed.

Sparse matrices

A matrix is a two dimensional collection, typically thought of as having rows and columns of data. One of the easiest ways to create a matrix is to use a list of lists. For example, consider the matrix shown below.



We can represent this collection as five rows, each row having five columns. Using a list of lists representation, we will have a list of five items, each of which is a list of five items. The outer items represent the rows and the items in the nested lists represent the data in each column.

```
matrix = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
```

One thing that you might note about this example matrix is that there are many items that are zero. In fact, only three of the data values are nonzero. This type of matrix has a special name. It is called a sparse matrix (http://en.wikipedia.org/wiki/Sparse_matrix).

Since there is really no need to store all of the zeros, the list of lists representation is considered to be inefficient. An alternative representation is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix.

```
matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

We only need three key-value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.

To access an element of the matrix, we could use the `[]` operator:

```
matrix[(0, 3)]
```

Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.

There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key. The alternative version of the `get` method solves this problem. The first argument will be the key. The second argument is the value `get` should return if the key is not in the dictionary (which would be 0 since it is sparse).

```
matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
print(matrix.get((0,3)))

print(matrix.get((1, 3), 0))

```

ActiveCode: 9 (chp12_sparse)

Run

```
1
0
```

Lab

- Counting Letters (http://dcs.asu.edu/faculty/abansal/CST100/Labs/Dictionaries-CountingLettersLab.html) In this guided lab exercise we will work through a problem solving exercise that will use dictionaries to generalize the solution to counting the occurrences of all letters in a string.

Lab

- Letter Count Histogram (http://dcs.asu.edu/faculty/abansal/CST100/Labs/Dictionaries-LetterCountHistogramLab.html) Combine the previous lab with the histogram example.
