python

# Dictionaries

The compound types —strings, lists, and tuples—use integers as indices. If you try to use any other type as an index, you get an error.

**Dictionaries** are similar to other compound types except that they can use any immutable type as an index.

One way to create a dictionary is to start with the empty dictionary and add elements. The empty dictionary is denoted {}:

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
```

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
```

The first assignment creates a dictionary named `eng2sp`; the other assignments add new elements to the dictionary. We can print the current value of the dictionary in the usual way:

```
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

The elements of a dictionary appear in a comma-separated list. Each entry contains an index and a value separated by a colon. In a dictionary, the indices are called **keys**, so the elements are called **key-value pairs**.

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

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

If we print the value of `eng2sp` again, we get a surprise:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The key-value pairs are not in order! Fortunately, there is no reason to care about the order, since the elements of a dictionary are never indexed with integer indices. Instead, we use the keys to look up the corresponding values:

```
>>> print eng2sp['two']
'dos'
```

The key `'two'` yields the value `'dos'` even though it appears in the third key-value pair.

# 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:

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

If someone buys all of the pears, we can remove the entry from the dictionary:

```
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

Or if we're expecting more pears soon, we might just change the value associated with pears:

```
>>> inventory['pears'] = 0
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
```

The `len` function also works on dictionaries; it returns the number of key-value pairs:

```
>>> len(inventory)
4
```

# Dictionary methods

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the `keys` method takes a dictionary and returns a list of the keys that appear, but instead of the function syntax `keys(eng2sp)`, we use the method syntax `eng2sp.keys()`.

```
>>> eng2sp.keys()
['one', 'three', 'two']
```

This form of dot notation specifies the name of the function, `keys`, and the name of the object to apply the function to, `eng2sp`. The parentheses indicate that this method has no parameters.

A method call is called an **invocation**; in this case, we would say that we are invoking `keys` on the object `eng2sp`.

The `values` method is similar; it returns a list of the values in the dictionary:

```
>>> eng2sp.values()
['uno', 'tres', 'dos']
```

The `items` method returns both, in the form of a list of tuples—one for each key-value pair:

```
>>> eng2sp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]
```

The syntax provides useful type information. The square brackets indicate that this is a list. The parentheses indicate that the elements of the list are tuples.

The syntax provides useful type information. The square brackets indicate that it's a list. The parenthesis indicate that the elements in the list are tuples.

# Aliasing and copying

Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

If you want to modify a dictionary and keep a copy of the original, use the `copy` method. For example, `opposites` is a dictionary that contains pairs of opposites:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

`alias` and `opposites` refer to the same object; `copy` refers to a fresh copy of the same dictionary. If we modify `alias`, `opposites` is also changed:

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

If we modify `copy`, `opposites` is unchanged:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

# Sparse matrices

consider a sparse matrix like this

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

The list representation contains a lot of zeroes:

```
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] ]
```

An alternative 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]
1
```

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:

```
>>> matrix[1,3]
KeyError: (1, 3)
```

The `get` method solves this problem:

```
>>> matrix.get((0,3), 0)
1
```

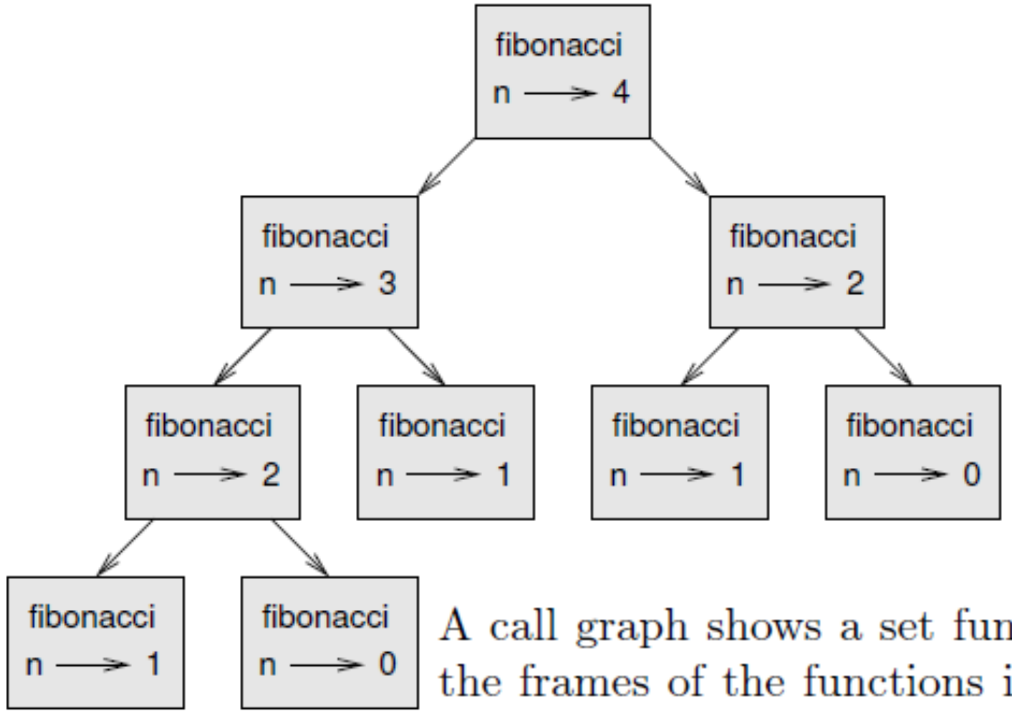
The first argument is the key; the second argument is the value `get` should return if the key is not in the dictionary:

```
>>> matrix.get((1,3), 0)
0
```

`get` definitely improves the semantics of accessing a sparse matrix. Shame about the syntax.

the `fibonacci` function, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly.

To understand why, consider this **call graph** for `fibonacci` with `n=4`:




A call graph shows a set function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.

A good solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **hint**. Here is an implementation of `fibonacci` using hints:

```
previous = {0:1, 1:1}

def fibonacci(n):
  if previous.has_key(n):
    return previous[n]
  else:
    newValue = fibonacci(n-1) + fibonacci(n-2)
    previous[n] = newValue
    return newValue
```

Using this version of `fibonacci`, our machines can compute `fibonacci(40)` in an eyeblink. But when we try to compute `fibonacci(50)`, we see the following:

```
>>> fibonacci(50)
20365011074L
```

The `L` at the end of the result indicates that the answer +(20,365,011,074) is too big to fit into a Python integer. Python has automatically converted the result to a long integer.

# Counting letters

Dictionaries provide an elegant way to generate a histogram:

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...    letterCounts[letter] = letterCounts.get (letter, 0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

We start with an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.

It might be more appealing to display the histogram in alphabetical order. We can do that with the `items` and `sort` methods:

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

**Iterate over Python dictionaries using for loops**

```
d={'red':1,'green':2,'blue':3}
for color_key, value in d.items():
        print(color_key,'corresponds to', d[color_key])
```

OUTPUT:

```
blue corresponds to 3
green corresponds to 2
red corresponds to 1
```

**Remove a key from a Python dictionary**

```
myDict = {'a':1,'b':2,'c':3,'d':4}
print(myDict)
if 'a' in myDict:
    del myDict['a']
print(myDict)
```

OUTPUT:

{'d': 4, 'a': 1, 'b': 2, 'c': 3}
 {'d': 4, 'b': 2, 'c': 3}

**Sort a Python dictionary by key**

```
color_dict = {'red':'#FF0000',
          'green':'#008000',
          'black':'#000000',
          'white':'#FFFFFF'}

for key in sorted(color_dict):
    print(key," :", color_dict[key]))
```

OUTPUT:

black: #000000
green: #008000
red: #FF0000
white: #FFFFFF

**Find the maximum and minimum value of a Python dictionary**

```
my_dict = {'x':500, 'y':5874, 'z': 560}

key_max = max(my_dict.keys(), key=(lambda k: my_dict[k]))
key_min = min(my_dict.keys(), key=(lambda k: my_dict[k]))

print('Maximum Value: ',my_dict[key_max])
print('Minimum Value: ',my_dict[key_min])
```

OUTPUT:

```
Maximum Value: 5874
Minimum Value: 500
```

**Concatenate two Python dictionaries into a new one**

```
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
dic4 = {}
for d in (dic1, dic2, dic3): dic4.update(d)
print(dic4)
```

OUTPUT:

{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}

**Test whether a Python dictionary contains a specific key**

```
fruits = {}
fruits["apple"] = 1
fruits["mango"] = 2
fruits["banana"] = 4
```

```
if "mango" in fruits:
    print("Has mango")
else:
    print("No mango")

if "orange" in fruits:
    print("Has orange")
else:
    print("No orange")
```

OUTPUT:

Has mango
No orange

# QUESTIONS

Q1:Write a Python script to add a key to a dictionary.

Sample Dictionary : {0: 10, 1: 20}
Expected Result : {0: 10, 1: 20, 2: 30}

Q2: Write a Python script to concatenate following dictionaries to create a new one.
Sample Dictionary :
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
Expected Result : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}

Q3: Write a Python script to check if a given key already exists in a dictionary.

Q4: Write a Python script to print a dictionary where the keys are numbers between 1 and 15 (both included) and the values are square of keys.

Sample Dictionary
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225}

Q6: Write a Python script to merge two Python dictionaries.

Q7: Write a Python program to sum all the items in a dictionary.

Q8:  Write a Python program to multiply all the items in a dictionary.

Q9: Write a Python program to remove a key from a dictionary.

Q10: Write a Python program to remove duplicates from Dictionary.

Q11: Write a Python program to create and display all combinations of letters, selecting each letter from a different key in a dictionary.

Sample data : {'1':['a','b'], '2':['c','d']}
Expected Output:
ac
ad
bc
bd