Dictionaries

Data structures are programming constructs used to store information. `Lists` are examples of data structures. Another useful python data structure is the `dictionary`. A dictionary is an unordered set of `key: value` pairs, where each `key` is required to be unique.

We declare a dictionary as follows:

```
## Create a dictionary mapping cities to their population (in millions)
## In this example the keys are 'string's and the values are 'float's.

dict = { 'Toronto': 2.89,            ## Notice the commas, they're important
         'New York City': 8.54,
         'Mexico City': 8.91,
         'Cairo': 9.50 }
```

We can access the value associated to a `key` using the 'square bracket' notation:

```
pop1 = dict['Toronto']
print(pop1)    ## prints 2.89

pop2 = dict['New York City']
print(pop2)    ## prints 8.54
```

Square bracket notation is also used to update a `key`:`value` pairing, or to add new parings:

```
dict['Toronto'] = 2.91
print(dict['Toronto'])    ## prints 2.91

dict['Hanoi'] = 3.44
print(dict['Hanoi'])      ## prints 3.44
```

The `keys` don't have to be strings. For example, we can also use number types and `tuples` as `keys`. Here's another example:

```
## A dictionary ordering the cities from the previous example by population.
cities_by_size = { 1: 'Cairo',
                   2: 'Mexico City',
                   3: 'New York City',
                   4: 'Toronto' }

print(cities_by_size[1])    ## prints Cairo
print(cities_by_size[2])    ## prints Mexico City
```

Control Flow

If statements. So far, the computer has run our programs by starting at the top of the file and reading every line in sequence. But suppose we want certain bits of code to run only under special conditions. This can be accomplished using an `if`-statement.

In order to understand `if`-statements, let's look at a simple example. We'll break it down afterwards.

```
import random    ## Ignore this line for now

letter = random.choice(['a', 'b'])

if letter == 'a':
    output = 'Got a'    ## Notice the indentation: 1 tab
    print(output)
else:
    output = 'Got b'    ## Notice the indentation
    print('Got b')
```

The first line loads the `random` module. We don't need to worry about this for now.

In the second line the `random` module is used to make an arbitrary choice between the letters 'a' and 'b'.

The next code block is the `if`-statement. Let's break it down line by line:

- `if letter == 'a':`
  Everything between `if` and the `:` is called the *condition* of the statement. The condition can be anything that evaluates to `True` or `False`. Here we check whether `letter` is `a` using the `==` operator. If so, the program prints `Got a`.

- ```
      output = 'Got a'
      print(output)
  ```
  These two lines are a *code block*, a group of statements that get executed together. They only run if the condition evaluates to `True`. All the lines in a
  ```

block get indented by one ‘tab’; we end a block by breaking the indentation.

- `else`:
  ```
      output = 'Got b'
      print(output)
  ```
  The `else` is not indented, so it ends the first code block. We then start a new block with instructions for when the condition evaluates to `False`.

Variants of the `if`-statement. Two different slightly different versions of the `if`-statement are often useful.

The lone `if`. Suppose we don't want to do anything when the condition is `False`. We can just omit the `else` part in this case. Here's an example:

```
import random

n = random.randint(-10, 10)    ## Get a random integer n satisfying -10 <= n <=10

if n > 0:
    print('n was positive')

m = n+1
## ... program continues ...
```

The program executes the indented statements only if the condition is `True`. Unindented statements after the `if`-block are executed no matter what.

`else if` statements. These types of statements allow us to handle multiple conditions. Here's an example:

```
import random

n = random.randint(-10, 10)    ## Get a random integer n satisfying -10 <= n <=10

if n > 0:
    print('n was positive')
elif n < 0:                    ## 'elif' is an abbreviation of 'else if'
    print('n was negative')
elif n == 0:
    print('n was zero')
```

For loops

`for`-loops are used when we want to run a code block multiple times. These loops repeat a code block for each element of a sequence. The following example prints each of the integers n such that $0 \leq n < 3$:

```
for n in range(0, 3):
    print(n)

## 0
## 1
## 2
```

We often want to run the same code for each element in a list or dictionary. There is an easy syntax for these tasks. For lists:

```
grocery_list = ['apple', 'orange', 'kiwi']

for fruit in grocery_list:
    print(fruit)

## apple
## orange
## kiwi
```

For dictionaries:

```
grocery_prices = { 'apples': 2.50,
                   'oranges': 2.25,
                   'kiwis': 1.25 }
total_cost = 0.0

for key, value in grocery_list.items():
    print(key + ' cost: ' repr(value))    ## We need to write 'repr(value)'
    total_cost = total_cost + value         ## to covert 'value' to a string.

print('Total: ' + repr(total_cost))

## apples cost: 2.5
## oranges cost: 2.25
## kiwis cost: 1.25
## Total: 6.0
```