The last marble

The `random` module

The random module from the The Python Standard Library can be used to add randomness to your Python code. Here's a selection of the functions implemented by this module.

function	short description
`random()`	returns a random floating point number from the range $[0, 1[$
`randint(a, b)`	returns a random integer from the range $[a, b]$
`choice(s)`	returns a random element from the non-empty sequence `s`
`sample(s, k)`	returns `k` distinct elements from the sequence or set `s`
`shuffle(l)`	randomly shuffles the sequence s in place

Here are some examples.

```
>>> import random

>>> random.random()
0.954131645221452
>>> random.random()
0.3548429482674793

>>> random.randint(3, 10)
5
>>> random.randint(3, 10)
8

>>> aList = ['a', 'b', 'c']
>>> random.choice(aList)
'b'
>>> random.choice(aList)
'a'
>>> aList
['a', 'b', 'c']

>>> random.sample(aList, 2)
['a', 'c']
>>> random.sample(aList, 2)
['b', 'a']
>>> aList
['a', 'b', 'c']

>>> random.shuffle(aList)
>>> aList
['c', 'a', 'b']
```

Passing mutable objects to functions

If you pass a mutable object to a function, the function may modify the object *in place*. This might be an explicit goal of the function, but sometimes it is not desirable to modify values that are passed to a function while the function is being executed.

Say, for example, that we pass a list to a function. What we actually pass to the function is a reference to that list and not a copy of the list (*call by reference* instead of *call by value*). As a result, the parameter to which the list is assigned becomes an alias for the list, and the function is able to modify the list itself (after all, lists are mutable data structures).

```
>>> def modify(aList, element):
...     aList.append(element)
...     return aList
...
>>> aList = ['a', 'b']
>>> modified = modify(aList, 'c')
>>> modified
['a', 'b', 'c']
>>> aList
['a', 'b', 'c']
```

In the example below, we first make a copy of the list that is passed to the function. Then we modify the copy, but not the original list. Making a copy of a list can be done for example by using *slicing* (`aList[:]`) or by using the built-in function `list` (`list(aList)`).

```
>>> def modify(aList, element):
...     copy = aList[:]
...     copy.append(element)
...     return copy
...
>>> aList = ['a', 'b']
>>> modified = modify(aList, 'c')
>>> modified
['a', 'b', 'c']
>>> aList
['a', 'b']
```

The Python Tutor gives a graphical representation of the difference between the above examples:

- example without copying
- example with copying

Because we no longer need the reference to the original list that was passed to the function, we may rewrite the function `modify` from the above example in the following way.

```
def modify(aList, element):
    aList = aList[:]
    aList.append(element)
    return aList
```

In this, the reference of the variable `aList` to the original list that is passed to the function `modify`, is replaced by a reference to a copy of the list. This replacement is only visible inside the function, because the variable `aList` is a local variable of the function `modify`. You can also inspect this example using the Python Tutor.

Obscure holidays

The `datetime` module

The datetime module from the The Python Standard Library defines a couple of new data types that can be used to represent dates (`datetime.date` objects) and periods of time (`datetime.timedelta` objects) in Python code. Here are some examples.

```
>>> from datetime import date
>>> birthday = date(1990, 10, 3)
>>> birthday = date(day=3, month=10, year=1990)
>>> birthday.day              # day is a property
3
>>> birthday.month            # month is a property
10
>>> birthday.year             # year is a property
1990
>>> birthday.weekday()        # weekday is a method !!
2
>>> today = date.today()
>>> today
datetime.date(2015, 11, 10)   # executed on October 11th, 2015
>>> from datetime import timedelta
>>> tomorrow = today + timedelta(1)
>>> tomorrow
datetime.date(2015, 11, 11)
>>> difference = tomorrow - today
>>> type(difference)
datetime.timedelta
>>> difference.days
1
```

Curling

Sort based on optional parameter `key`

The list method `sort` and the built-in function `sorted` can both be used to sort the elements of a given list. They differ in the fact that the `sort` method rearranges the elements of the list *in place*, whereas the function `sorted` returns a new sorted list, while leaving the orginal list unchanged.

Apart from this difference, both functions have many things in common. They both have an optional parameter `reverse` that takes a Boolean value. The value indicates whether the elements have to be sorted in increasing (value `False`, the default value) or decreasing (value `True`) order. Both functions also have a second optional parameter `key` that can be used to determine the order of the elements. This ordering of the elements will be used when sorting the list.

The parameter `key` takes a function as its argument. This function must take a single argument. In case a function `f` is passed to the parameter `key`, the order of the elements is not determined by the elements themselves, as is the default behaviour, but is based on the values returned by the function `f` for each of the element (each element is this passed individually as an argument to the function `f`).

Say, for example, that you have defined a function `f` and that you pass this function to the parameter `key`. Before the actual sorting takes place, a function call `f(element)` is done for each `element` in the list that needs to be sorted. Afterwards, the elements of the list are sorted based on the values returned by the

function `f` for each of the elements in the list. At the first position in the sorted list you will find the `element` that results in the smallest value for `f(element)` (or the largest value in case `reverse=True`), and at the last position in the sorted list you will find the `element` that results in the largest value for `f(element)` (or the smallest value in case `reverse=True`).

The natural order in which tuples are sorted is to sort the tuples first based on their first elements, and in case these elements have equal values sort them further based on successive elements in the tuple. Say, for example, that we have a list of tuples, where each tuple contains two integers. The natural ordering of these tuples results in the following outcome.

```
>>> aList = [(2, 7), (0, 10), (4, 0), (1, 6), (2, 5), (2, 6)]
>>> sorted(aList)
[(0, 10), (1, 6), (2, 5), (2, 6), (2, 7), (4, 0)]
```

If we wanted to sort the tuples first on their second element, and then on their first element, we could do this in the following way.

```
>>> def sortkey(pair):
...     return pair[1], pair[0]
...
>>> aList = [(2, 7), (0, 10), (4, 0), (1, 6), (2, 5), (2, 6)]
>>> sorted(aList, key=sortkey)
[(4, 0), (2, 5), (1, 6), (2, 6), (2, 7), (0, 10)]
```

Please not that this ordering is not the same as the reverse natural ordering of the elements.

Paarsgewijs vergelijken van elementen in een lijst

Whenever you want to compare all elements of a list in a pairwise manner, you can use a double for loop that iterates over the indexes of the elements in the series.

```
>>> list = ['a', 'b', 'c', 'd']
>>> for i in range(len(list)):
...     for j in range(i + 1, len(list)):
...         compare(list[i], list[j])
```

This will compare the elements in the following order:

- elements on indexes 0 and 1 (‘a’ with ‘b’)
- elements on indexes 0 and 2 (‘a’ with ‘b’)
- elements on indexes 0 and 3 (‘a’ with ‘b’)
- elements on indexes 1 and 2 (‘a’ with ‘b’)
- elements on indexes 1 and 3 (‘a’ with ‘b’)
- elements on indexes 2 and 3 (‘a’ with ‘b’)

Distance between two points

To compute the distance between two points with coordinates (x_1, y_1) and (x_2, y_2), the following formula can be used.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Specific information

To implement the function `score` it can be useful to first sort the stones in function of their distance to the dolly. In that way the stone closest to the dolly is at the first position of the list, the second closest on the second position, etc.

Five up

The string method `join`

The string method `join` can be used to concatenate all string in an iterable object (e.g. a list) into a single string. This is done by concatenating all strings in the iterable object using a separator, which is the string on which the string method `join` is called.

```
>>> aList = ['a', 'b', 'c']
>>> ' '.join(aList)
'a b c'
>>> ''.join(aList)
'abc'
>>> '---'.join(aList)
'a---b---c'
>>> ' - '.join(aList)
'a - b - c'
```