General

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.

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

Assigning functions to variables

In Python functions are themselves object of the data type `function`, so they can be assigned to variables just like any other object. This is handy if you have to code fragments that are exactly the same, except for the fact that at some point you need to call another function.

Say, for example, that we want to write a program that first needs to print all words containing the letter `a` from a given list of words, and then also needs to print all words containing the letter `b` from the same list of words. We could do this in the following way

```
>>> words = ['apple', 'banana', 'berry']
>>>
>>> def contains_a(word):
...     return 'a' in word
...
>>> def contains_b(word):
...     return 'b' in word
...
>>> for word in words:
...     if contains_a(word):
...         print(word)
...
apple
banana
>>> for word in words:
...     if contains_b(word):
...         print(word)
banana
berry
```

We could slightly rewrite the two `for` loops in the above code fragments

```
>>> func = contains_a
>>> for word in words:
...     if func(woord):
...         print(word)
...
apple
banana
>>> func = contains_b
>>> for word in words:
...     if func(word):
...         print(word)
banana
berry
```

so that we twice get exactly the same `for` loop. Because we want to avoid code duplication (we never want to program the exact same code twice), we can rewrite this by introducing another `for` loop that iterates over both functions

```
>>> functions = [contains_a, contains_b]
>>> for func in functions:
...     for word in words:
...         if func(word):
...             print(word)
apple           # generated during first iteration (func == contains_a)
banana
banana          # generated during second iteration (func == contains_b)
berry
```

ISBN

None as default value

If you want to define a function that has an optional parameter, you always need to assign a default value to that parameter. However, it may happen that this default value is not known when the function is defined, and can only be determined at run-time (when the function is called). This is, for example, the case when the default value itself depends on argument that are passed to other parameters. In this case, it's a good idea to assign the value `None` as the default value when defining the function, and to assign a computed default value in the body of the function in case the value `None` was assigned to the optional parameter (meaning: no explicit value was passed to this parameter).

```
def func(first, second=None):
    if second is None:
        # assign the same value to the second parameter that was passed as an
        # argument to the first (mandatory) parameter, in case no explicit value
        # was passed to the second argument when calling the function
        second = first

    ...
```

This could not be solved in the following way

```
def func(first, second=first):

    ...
```

because at the time the function is defined, the parameter `first` has not been assigned a specific value. This is the same thing as using a variable that has not yet been defined.

You should also use `None` as a default value, if the actual default value that you want to assign has a mutable data type. It is recommended not to write

```
def func(first, second=[]):

    # NOTE: in this case a single empty list is created at the time the function
    #       is defined; because lists are mutable, the list can be modifed in
    #       place each time the function is called; usually, this is not the
    #       expected behavior

    ...
```

but to implement this in the following way

```
def func(first, second=None):

    # assign empty list as a default value
    # NOTE: now a new empty list is created each time the function is called, so
    #       this list is specific for that function call
    if second is None:
        second = []

    ...
```

If you want to assign a default value to a parameter that is fixed at the time the function is defined, and that has an immutable data type (`int`, `bool`, `string`, `tuple`, ...), you may safely assign the default value in the traditional way.

```
def func(first, second=42):

    ...
```

Buzz-phrases

Functions that take an arbitrary number of arguments

In defining a function, you fix the number of arguments that needs to be passed when calling the function. This number corresponds to the number of parameters that is given with the definition of the function. For example, the following code snippet defines a function `sum` that takes exactly two arguments, and will return the sum of adding the two objects that are passed to these parameters.

```
>>> def sum(term1, term2):
...     return term1 + term2
...
>>> sum(1, 2)
3
```

Say, however, that we wanted to write the function in such a way that it takes an arbitrary number of arguments and still returns the result of adding all the objects that are passed when calling the function. This can be done by preceding a parameter with an asterisk (`*`). This parameter will be assigned a typle containing all positional arguments that are passed to the function that are not assigned to other parameters.

```
>>> def sum(*terms):
...     total = 0
...     for term in *terms:
...         total += getal
...     return total
...
>>> sum(1, 2)
3
>>> sum(1, 2, 3)
6
>>> sum(1, 2, 3, 4)
10
```

In this case, all arguments passed to the function `sum` will be bundled in a tuple that is assigned to the local variable `terms`. It is also possible to name other parameters when defining the function, as long as the parameter carrying the asterisk closes the list of parameters. In addition, there can only be a single parameter that carries an asterisk.

For example, in the following code snippet we define a function `sum` that takes at least two arguments. The function still return the sum of all arguments passed to the function.

```
>>> def sum(term1, term2, *terms):
...     total = term1 + term2
...     for term in terms:
...         total += term
...     return totaal
...
>>> sum(1, 2)
3
>>> sum(1, 2, 3)
6
>>> sum(1, 2, 3, 4)
```

```
10
```

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

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

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

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

Unpredictable birthdays

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
```

Five up

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(some_list, element):
...     some_list.append(element)
...     return some_list
...
>>> some_list = ['a', 'b']
>>> modified = modify(some_list, 'c')
>>> modified
['a', 'b', 'c']
>>> some_list
['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* (`some_list[:]`) or by using the built-in function `list` (`list(some_list)`).

```
>>> def modify(some_list, element):
...     copy = some_list[:]
```

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

In this, the reference of the variable `some_list` 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 `some_list` is a local variable of the function `modify`. You can also inspect this example using the Python Tutor.