General

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']
```

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
```

Sort based on optional parameter `key`

The method `list.sort()` and the built-in function `sorted()` can both be used to sort the elements of a given list. They differ in that the method `list.sort()` rearranges the elements of the list *in place*, whereas the function `sorted()` returns a new sorted list, while leaving the original 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 an 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 elementd (each element is 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 note that this ordering is not the same as the reverse natural ordering of the elements.

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):

    ...
```