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 function that are 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(s)`	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']
```

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
```

Buzz-phrases

General information

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 type 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
```

Proizvolovs identity

General information

Remarks

If you want to traverse the elements of two or more *iterable objects* (objects of compound data types that have an associated iterator) simultaneously, you do this using the built-in function `zip`. This function returns an iterator that initially returns a tuple containing the first elements of all iterable objects passed to the function `zip`, then a tuple containing all second elements of those iterable objects, and so on.

Say, for example, that you can to add two lists element-wise, thereby creating a new list whose i-th element is the sum of the i-th elements of the two original lists. This can be done in the following way.

```
>>> first = [1, 2, 3]
>>> second = [4, 5, 6]
>>> added = []
>>> for term1, term2 in zip(first, second):
...     added.append(term1 + term2)
...
>>> added
[5, 7, 9]
```

This can also be written a bit shorter by making use of a *list comprehension*.

```
>>> first = [1, 2, 3]
>>> second = [4, 5, 6]
>>> added = [term1 + term2 for term1, term2 in zip(first, second)]
>>> added
[5, 7, 9]
```

The iterator stops (raises a `StopIteration` exception) as soon as one of the iterable objects is exhausted. If you want to traverse two or more iterable objects simultaneously until the last of those objects is exhausted, you may do this using the function `zip_longest` from the `itertools` module.