PYTHON, RANDOM NUMBERS AND PROBABILITY

INTRODUCTION

"Every American should have above average income, and my Administration is going to see they get it."

This saying is attributed to Bill Clinton on umpteen websites. Usually, there is no context given, so it is not clear, if he might have meant it as a "joke". Whatever his intentions might have been, we quoted him to show a "real" life example of statistics. Statistics and probability calculation is all around us in real-life situations. We have to cope with it whenever we have to make a decision from various options. Can we go for a hike in the afternoon or will it rain? The weather forecast tells us, that the probability of precipitation will be 30 %. So what now? Will we go for a hike?
Another situation: Every week you play the lottery and dream of a far away island. What is the likelihood of winning the Jackpot so that you will never have to work again and live in "paradise"? Now imagine that you right on this island of your dreams. Most probably not because you won the jackpot, but rather because you booked your time as an all-inclusive holiday package. You are on holiday on a paradisal island far from home. Suddenly, you meet your neighbor, spoiling in a jiffy all dreams. Against all odds?
Uncertainty is all araound us, yet only few people understand the basics of probability theory.

The programming language Python and even the numerical modules Numpy and Scipy will not help us in understanding the everyday problems mentioned above, but Python and Numpy provide us with powerful functionalities to calculate problems from statistics and probability theory.

RANDOM NUMBERS WITH PYTHON

THE RANDOM AND THE "SECRETS" MODULES

There is an explicit warning in the documentation of the random module:

Warning:
Note that the pseudo-random generators in the random module should NOT be used for security purposes. Use secrets on Python 3.6+ and os.urandom() on Python 3.5 and earlier.

The default pseudo-random number generator of the random module was designed with the focus on modelling and simulation, not on security. So, you shouldn't generate sensitive information such as passwords, secure tokens, session keys and similar things by using random. When we say that you shouldn't use the random module, we mean the basic functionalities "randint", "random", "choise" , and the likes. There is one exception as you will learn in the next paragraph: SystemRandom

The SystemRandom class offers a suitable way to overcome this security problem. The methods of this class use an alternate random number generator, which uses tools provided by the operating system (such as /dev/urandom on Unix or CryptGenRandom on Windows.

As there has been great concern that Python developers might inadvertently make serious security errors, - even though the warning is included in the documentaiton, - Python 3.6 comes with a new module "secrets" with a CSPRNG (Cryptographically Strong Pseudo Random Number Generator).

Let's start with creating random float numbers with the random function of the random module. Please remember that it shouldn't be used to generate sensitive information:

```
import random
random_number = random.random()
print(random_number)
```

```
0.34330263184538523
```

We will show an alternative and secure approach in the following example, in which we will use the class SystemRandom of the random module. It will use a different random number generator. It uses sources which are provided by the operating system. This will be /dev/urandom on Unix and CryptGenRandom on windows. The random method of the SystemRandom class generates a float number in the range from 0.0 (included) to 1.0 (not included):

```
from random import SystemRandom
crypto = SystemRandom()
print(crypto.random())
```

```
0.8875057137654113
```

GENERATE A LIST OF RANDOM NUMBERS

Quite often you will need more than one random number. We can create a list of random numbers by repeatedly calling random().

```
import random
def random_list(n, secure=True):
    random_floats = []
    if secure:
        crypto = random.SystemRandom()
        random_float = crypto.random
    else:
        random_float = random.random
    for _ in range(n):
        random_floats.append(random_float())
    return random_floats
print(random_list(10, secure=False))
```

```
[0.9702685982962019, 0.5095131905323179,
0.9324278634720967, 0.9750405405778308,
0.9750927470224396, 0.2383439553695087,
0.03591694433088444, 0.9203791901577599,
0.07793301506800698, 0.46915245764066404]
```

The "simple" random function of the random module is a lot faster as we can see in the following:

```
%%timeit
random_list(100)
```

```
10000 loops, best of 3: 158 Âµs per loop
```

```
%%timeit
random_list(100, secure=False)
```

```
100000 loops, best of 3: 8.64 Âµs per loop
```

```
crypto = random.SystemRandom()
[crypto.random() for _ in range(10)]
```

The code above returned the following:

```
[0.5832874631978111,
 0.7494815897496974,
 0.6982338101218046,
 0.5164288598133177,
 0.15423895558995826,
 0.9447842390510461,
 0.08095707071826808,
 0.5407159221282145,
 0.6124979567571185,
 0.15764744205801628]
```

Alternatively, you can use a list comprehension to create a list of random float numbers:

```
%%timeit
[crypto.random() for _ in range(100)]
```

```
10000 loops, best of 3: 157 Âµs per loop
```

The fastest and most efficient way will be using the random package of the numpy module:

```
import numpy as np
np.random.random(10)
```

The above Python code returned the following:

```
array([ 0.0422172 ,  0.98285327,  0.40386413,
0.34629582,  0.25666744,
        0.69242112,  0.9231164 ,  0.47445382,
0.63654389,  0.06781786])
```

```
%%timeit
np.random.random(100)
```

```
The slowest run took 16.56 times longer than the
fastest. This could mean that an intermediate
result is being cached.
100000 loops, best of 3: 2.1 Âµs per loop
```

Warning:
The random package of the Numpy module apparantly - even though it doesn't say so in the documentation - is completely deterministic, using also the Mersenne twister sequence!

RANDOM NUMBERS SATISFYING SUM-TO-ONE CONDITION

It's very easy to create a list of random numbers satisfying the condition that they sum up to one. This way, we turn them into values, which could be used as probalities. We can use any of the methods explained above to normalize a list of random values. All we have to do is divide every value by the sum of the values. The easiest way will be using numpy again of course:

```
import numpy as np
list_of_random_floats = np.random.random(100)
sum_of_values = list_of_random_floats.sum()
print(sum_of_values)
normalized_values = list_of_random_floats /
sum_of_values
print(normalized_values.sum())
```

```
52.3509839137
1.0
```

GENERATING RANDOM STRINGS OR PASSWORDS WITH PYTHON

We assume that you don't use and don't like weak passwords like "123456", "password", "qwerty" and the likes. Believe it or not, these passwords are always ranking to 10. So you looking for a safe password? You want to create passwords with Python? But don't use some of the functions ranking top 10 in the search results, because you may use a functions using the random function of the random module.

We will define a strong random password generator, which uses the SystemRandom class. This class uses, as we have alreay mentioned, a cryptographically strong pseudo random number generator:

```
from random import SystemRandom
sr = SystemRandom() # create an instance of the
SystemRandom class

def generate_password(length,
                      valid_chars=None):
    """ generate_password(length, check_char) ->
password
        length: the length of the created password
        check_char: a Boolean function used to
check the validity of a char
    """
    if valid_chars==None:
        valid_chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        valid_chars += valid_chars.lower() +
"0123456789"

    password = ""
    counter = 0
    while counter < length:
        rnum = sr.randint(0, 128)
        char = chr(rnum)
        if char in valid_chars:
            password += chr(rnum)
            counter += 1
    return password
print("Automatically generated password by Python:
" + generate_password(15))
```

```
Automatically generated password by Python:
1l6Zki280gfUqMD
```

RANDOM INTEGER NUMBERS

Everybody is familar with creating random integer numbers without computers. If you roll a die, you create a random number between 1 and 6. In terms of probability theory, we would call "the rolling of the die" an experiment with a result from the set of possible outcomes {1, 2, 3, 4, 5, 6}. It is also called the sample space of the experiment.

How can we simulate the rolling of a die in Python? We don't need Numpy for this aim. "Pure" Python and its random module is enough.

```
import random
outcome = random.randint(1,6)
print(outcome)
```

```
4
```

Let's roll our virtual die 10 times:

```
import random
[ random.randint(1, 6) for _ in range(10) ]
```

The above code returned the following:

```
[2, 1, 5, 5, 6, 5, 4, 4, 1, 1]
```

We can accomplish this easier with the NumPy package random:

```
import numpy as np
outcome = np.random.randint(1, 7, size=10)
print(outcome)
```

```
[6 6 6 1 3 6 2 5 3 3]
```

You may have noticed, that we used 7 instead of 6 as the second parameter. randint from numpy.random uses a "half-open" interval unlike randint from the Python random module, which uses a closed interval!

The formal definition:

numpy.random.randint(low, high=None, size=None)

This function returns random integers from 'low' (inclusive) to 'high' (exclusive). In other words: randint returns random integers from the "discrete uniform" distribution in the "half-open" interval ['low', 'high'). If 'high' is None or not given in the call, the results will range from [0, 'low'). The parameter 'size' defines the shape of the output. If 'size' is None, a single int will be the output. Otherwise the result will be an array. The parameter 'size' defines the shape of this array. So size should be a tuple. If size is defined as an integer n, this is considered to be the tuple (n,).

The following examples will clarify the behavior of the parameters:

```
import numpy as np
print(np.random.randint(1, 7))
print(np.random.randint(1, 7, size=1))
print(np.random.randint(1, 7, size=10))
print(np.random.randint(1, 7, size=(10,))) # the
same as the previous one
print(np.random.randint(1, 7, size=(5, 4)))
```

```
5
[3]
[1 4 3 5 5 1 5 4 5 6]
[2 1 4 3 2 1 6 5 3 3]
[[4 1 3 1]
 [6 4 5 6]
 [2 5 5 1]
 [4 3 2 3]
 [6 2 6 5]]
```

Simulating the rolling of a die is usually not a security-relevant issue, but if you want to create cryptographically strong pseudo random numbers you should use the SystemRandom class again:

```
import random
crypto = random.SystemRandom()
[ crypto.randint(1, 6) for _ in range(10) ]
```

This gets us the following result:

```
[2, 1, 6, 4, 5, 6, 2, 5, 2, 1]
```

We have learned how to simulate the rolling of a die with Python. We assumed that our die is fair, i.e. the probability for each face is equal to 1/6. How can we simulate throwing a crooked or loaded die? The randint methods of both modules are not suitable for this purpose. We will write some functions in the following text to solve this problem.

First we want to have a look at other useful functions of the random module.

RANDOM CHOICES WITH PYTHON

"Having a choice" or "having choices" in real life is better than not having a choice. Even though some people might complain, if they have too much of a choice. Life means making decisions. There are simple choices like "Do I want a boiled egg?", "Soft or Hard boiled?" "Do I go to the cinema, theater or museum? Other choices may have further reaching consequences like choosing the right job, study or what is the best programming language to learn.

Let's do it with Python. The random module contains the right function for this purpose.This function can be used to choose a random element from a non-empty sequence.

This means that we are capable of picking a random character from a string or a random element from a list or a tuple, as we can see in the following examples. You want to have a city trip within Europe and you can't decide where to go? Let Python help you:

```
from random import choice
possible_destinations = ["Berlin", "Hamburg", 
"Munich",
                         "Amsterdam", "London", 
"Paris",
                         "Zurich", "Heidelberg", 
"Strasbourg",
                         "Augsburg", "Milan", 
"Rome"]
print(choice(possible_destinations))
```

```
Strasbourg
```

The choice function of the random package of the numpy module is more convenient, because it provides further possibilities. The default call, i.e. no further parameters are used, behaves like choice of the random module:

```
from numpy.random import choice
print(choice(possible_destinations))
```

```
Augsburg
```

With the help of the parameter "size" we can create a numpy.ndarray with choice values:

```
x1 = choice(possible_destinations, size=3)
print(x1)
x2 = choice(possible_destinations, size=(3, 4))
print(x2)
```

```
['London' 'Augsburg' 'London']
[['Strasbourg' 'London' 'Rome' 'Berlin']
 ['Berlin' 'Paris' 'Munich' 'Augsburg']
 ['Heidelberg' 'Paris' 'Berlin' 'Rome']]
```

You might have noticed that the city names can have multiple occurrences. We can prevent this by setting the optional parameter "replace" to "False":

```
print(choice(possible_destinations, size=(3, 4),
replace=False))
```

```
[['Heidelberg' 'London' 'Milan' 'Munich']
 ['Hamburg' 'Augsburg' 'Paris' 'Rome']
 ['Berlin' 'Strasbourg' 'Zurich' 'Amsterdam']]
```

Setting the "size" parameter to a non None value, leads us to the sample function.

RANDOM SAMPLES WITH PYTHON

A sample can be understood as a representative part from a larger group, usually called a "population".

The module numpy.random contains a function random_sample, which returns random floats in the half open interval [0.0, 1.0). The results are from the "continuous uniform" distribution over the stated interval. This function takes just one parameter "size", which defines the output shape. If we set size to (3, 4) e.g., we will get an array with the shape (3, 4) filled with random elements:

```
import numpy as np
x = np.random.random_sample((3, 4))
print(x)
```

```
[[ 0.99824096  0.30837203  0.85396161  0.84814744]
 [ 0.45516418  0.64925709  0.19576679  0.8124502 ]
 [ 0.45498107  0.20100427  0.42826199
0.57355053]]
```

If we call random_sample with an integer, we get a one-dimensional array. An integer has the same effect as if we use a one-tuple as an argument:

```
x = np.random.random_sample(7)
print(x)
y = np.random.random_sample((7,))
print(y)
```

```
[ 0.07729483  0.07947532  0.27405822  0.34425005
0.2968612   0.27234156
  0.41580785]
[ 0.19791769  0.64537929  0.02809775  0.2947372
0.5873195   0.55059448
  0.98943354]
```

You can also generate arrays with values from an arbitrary interval [a, b), where a has to be less than b. It can be done like this:

`(b - a) * random_sample() + a`

Example:

```
a = -3.4
b = 5.9
A = (b - a) * np.random.random_sample((3, 4)) + a
print(A)
```

```
[[ 5.87026891 -0.13166798  5.56074144  3.48789786]
 [-2.2764547   4.84050253  0.71734827 -0.7357672 ]
 [ 5.8468095   4.56323308  0.05313938
-1.99266987]]
```

The standard module random of Python has a more general function "sample", which produces samples from a population. The population can be a sequence or a set.

The syntax of sample:

`sample(population, k)`

The function creates a list, which contains "k" elements from the "population". The result list contains no multiple occurrences, if the the population contains no multiple occurrences.

If you want to choose a sample within a range of integers, you can - or better you should - use range as the argument for the population.

In the following example we produce six numbers out of the range from 1 to 49 (inclusive). This corresponds to a drawing of the German lottery:

```
import random
print(random.sample(range(1, 50), 6))
```

```
[27, 36, 29, 7, 18, 45]
```

TRUE RANDOM NUMBERS

Have you ever played a game of dice and asked yourself, if something is wrong with the die? You rolled the die for so many times and you still haven't got a certain value like 6 for example.

You may also have asked yourself, if the random modules of Python can create "real" or "true" random numbers, which are e.g. equivalent to an ideal die. The truth is that most random numbers used in computer programs are pseudo-random. The numbers are generated in a predictable way, because the algorithm is deterministic. Pseudo-random numbers are good enough for many purposes, but it may not be "true" random rolling dice or lottery drawings.

The website RANDOM.ORG claims to offer true random numbers. They use the randomness which comes from atmospheric noise. The numbers created this way are for many purposes better than the pseudo-random number algorithms typically used in computer programs.