Thinking in Probabilities

Module 0 | Chapter 3 | Notebook 1

In this chapter you will get a brief crash course in probability theory. We'll start with the probabilities associated with dice. By the end of this lesson you will have learned:

- why machine learning algorithms require large amounts of data
- how different the probabilities of almost identical processes can be

---

The law of large numbers

How do statistical models such as linear regressions, support vector machines or neural networks manage to generate predictions from very large amounts of data? The data sometimes contains contradictory data points. Nevertheless, prediction models do not give up and say: "the situation represented by the data is unclear". Somehow they manage to make predictions. How do they do that?

Statistical models abstract the data. It's as if the models imagine what sort of process could have created this kind of data. The statistical models imitate this and generate data in the same way. Unfortunately, the machine learning model does not know this process except for the actual data in the data set. The same data-producing process can also lead to completely different data sets.

Let's import the `scipy` module so you can get an idea of what this is all about. We won't use `scipy` beyond this chapter. So you don't necessarily need to learn its structure. It will just help you to develop an intuition about probabilities.

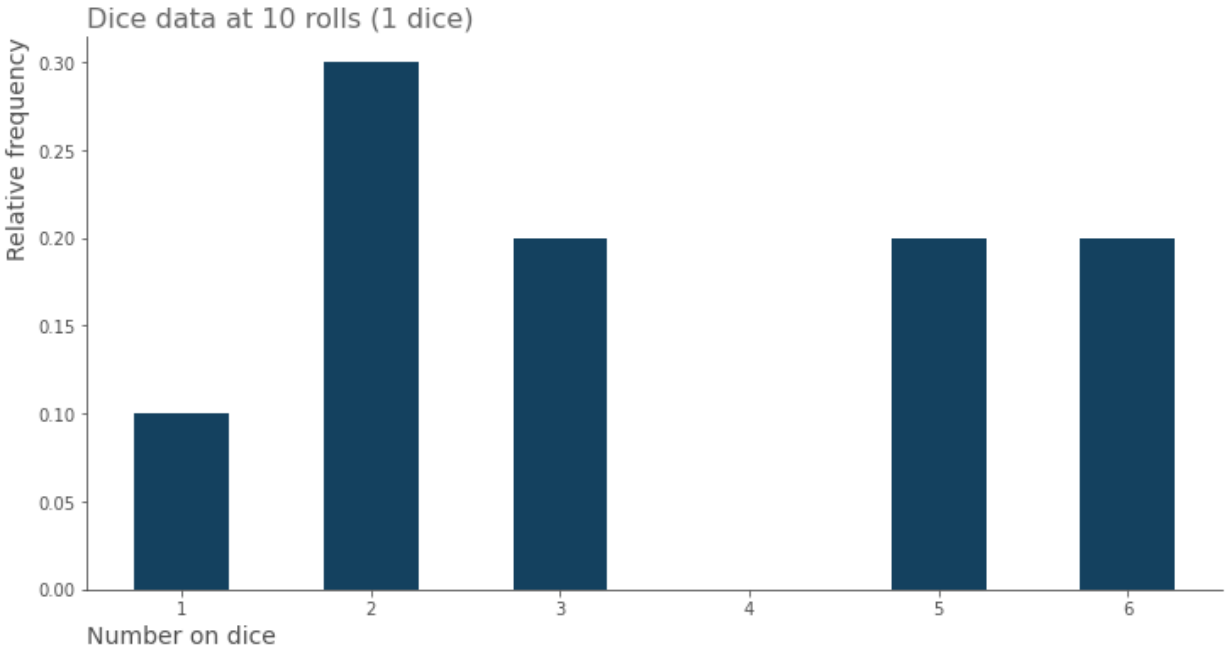
In the following we will simulate a process you are familiar with: dice. The following column chart shows the result after a fair die was rolled ten times. On the x-axis you can see the number of rolls and on the y-axis you can see how often each number number occurred in the 10 rolls. The y-axis shows the **relative** frequency for the rolls.

Tip: The code cell creates the function `dice_plot()`, which will carry out and visualize dice simulations for us. You don't have to understand it in detail. You can simply run the cell.

In [1]:
```
def dice_plot(rolls=10, dice=1):
    """Simulate dice rolls, return a Figure and Axes of the resulting histogram.

    Keyword arguments:
    rolls -- how many times the experiment gets repeated (default 10)
    dice  -- how many dice are used in the experiment. Their results are summed. (defa
    """
    # module import
```

```
from scipy.stats import randint
import matplotlib.pyplot as plt
import pandas as pd

# constants
low=1  # lowest possible score of one die
high=7  # highest possible score of one die +1

# data simulation
dice_data = [0]*rolls
for d in range(dice):
    dice_data = dice_data + randint.rvs(low, high, size=rolls)

# prepare data for plotting
zero_data = pd.DataFrame(data=[0] * ((dice*6)-(dice-1)),
                    index=range(dice, (dice*6)+1),
                    columns=['count'])  # all possible results with zero score
result = zero_data + pd.crosstab(index=dice_data,
                                 columns='count',
                                 normalize='columns')

# create visualisation
fig, ax = plt.subplots(figsize=[12, 6])
result.plot(kind='bar', legend=False,color='#17415f', ax=ax)

#remove top and right frame parts
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

# set left and bottom axis to #grey
ax.spines['left'].set_color('#515151')
ax.spines['bottom'].set_color('#515151')

# set ticks to grey
ax.tick_params(axis='x', colors='#515151', labelrotation=0)
ax.tick_params(axis='y', colors='#515151')

#set labels to grey
ax.yaxis.label.set_color('#515151')
ax.xaxis.label.set_color('#515151')

# align axis labels with axis ends
ax.set_xlabel(xlabel='Number on dice',
              position=[0, 0],
              horizontalalignment='left',
              color='#515151',
              size=14)
ax.set_ylabel(ylabel='Relative frequency',
              position=[0, 1],
              horizontalalignment='right',
              color='#515151',
              size=14)

# align title
ax.set_title(label='Dice data at {} rolls ({} dice)'.format(rolls, dice),
             loc='left',
             color=(0.41, 0.41, 0.41),
             size=16)

# white background color
```

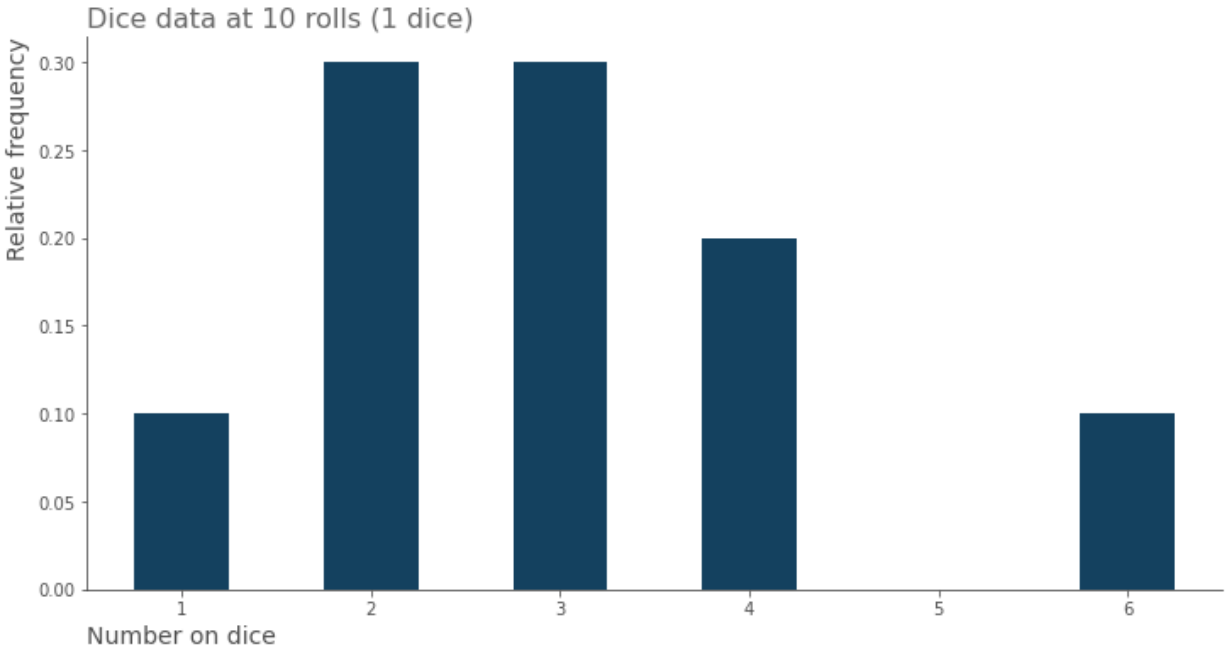
```
    fig.patch.set_facecolor('white')

    return fig, ax

dice_plot(rolls=10);
```



Now we have rolled one die ten times. With each roll there is only one of six possible results: one, two, three, four, five, or six. What share does each number take up? This is difficult to predict based on just ten rolls. If you were to repeat the ten rolls, you would get a different breakdown. Try this out by writing `dice_plot(rolls=10);` in the next code cell.

Important: Don't forget the semicolon (`;`) at the end of the line of code. In this programming environment, which supports Jupyter functionalities, this prevents the function's output from being returned. For `dice_plot()` this means that the names of the *Figure* and the *axes* are not displayed, but the visualization will be displayed.

In [2]:
```
dice_plot(rolls=10);
```



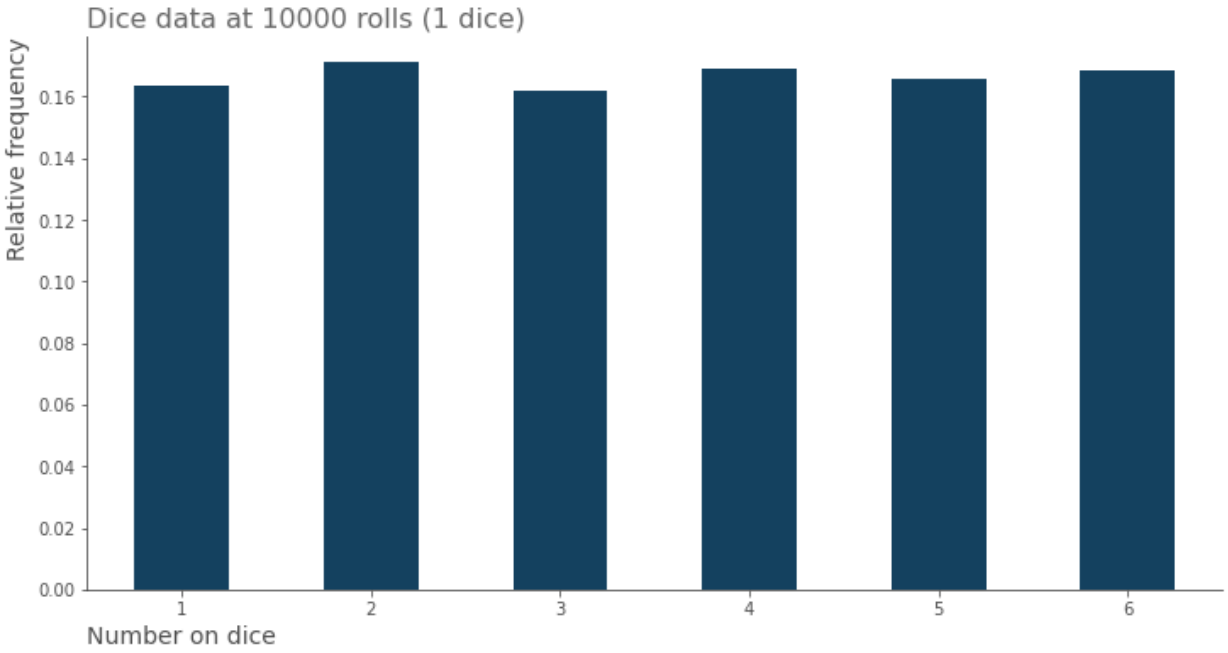
Your column charts probably look quite different. You can repeat the ten dice rolls as often as you like and you will hardly ever see the same column chart occur more than once. The same process (e.g. rolling ten times) can therefore lead to very different results. The reason for this is the influence of chance in this process.

There are of course also other data generating processes that always produce the same result. Chance doesn't play a role in them. If you multiply two numbers, this is a data producing process which a result that is 100% predictable.

However, the data-producing processes that data scientists work with behave differently. Similar to dice, you can perform them again and again, and get different results. They are random processes. How could anyone make predictions on this basis, e.g. the number on the die the next time it is rolled? How should a machine learning model imitate this kind of process to generate new data?

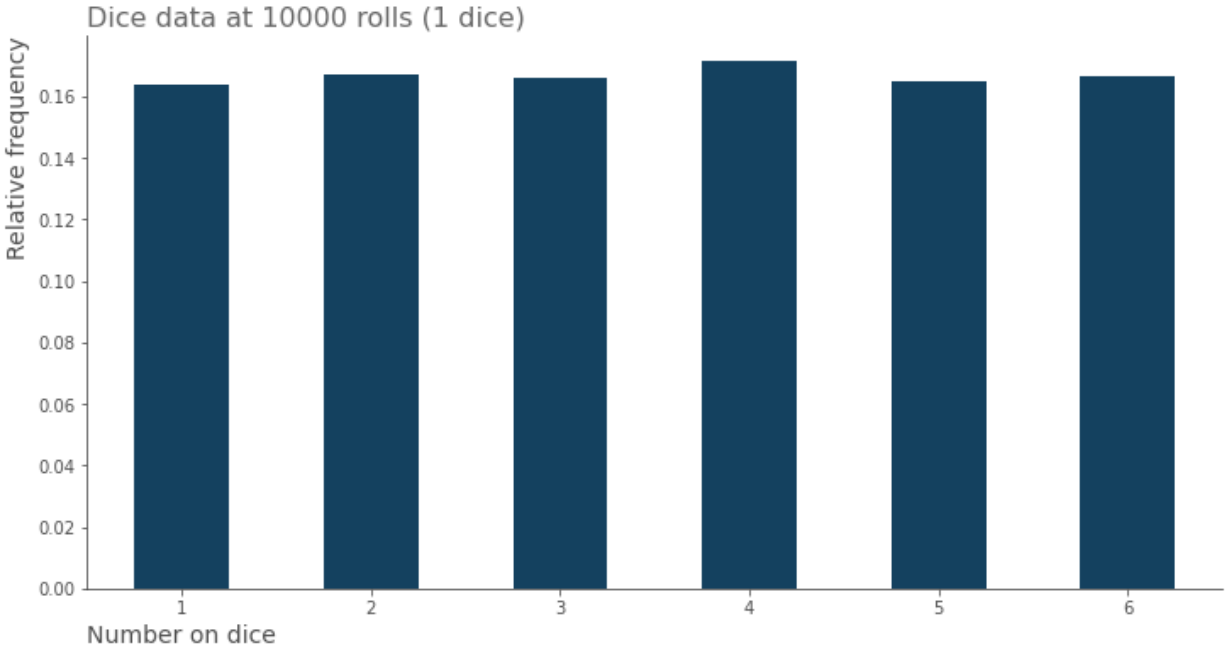
It turns out that if you roll the die a lot of times, the results are no longer so different. In the following visualization, the die was rolled 10,000 times. How often does each number occur?

In [3]:
```
dice_plot(rolls=10000);
```



Each score has a relative frequency of 16% to 18%. If we repeat this process now, the result will not be that different. Try it out by writing `dice_plot(rolls=10000);` in the following code cell.

In [4]:
```
dice_plot(rolls=10000);
```



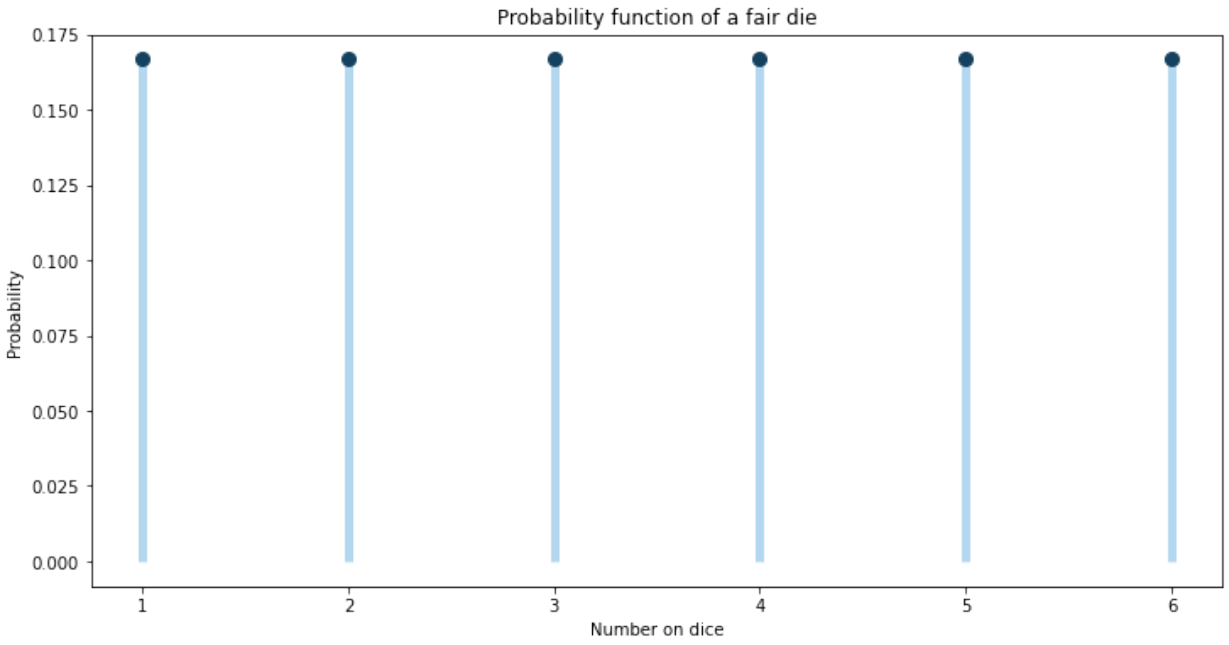
You can repeat the ten dice rolls as often as you like and you see almost exactly the same column chart. The relative frequency of the numbers on the die become increasingly similar to 1/6 (0.167 or 16.7%) as the number of rolls increases. This makes sense, because every number has the same probability, i.e. 1/6. The data-generating process therefore has the following formulations:

- with each roll, one of six possible results is achieved: 1, 2, 3...

- Each number on the die has the same probability: 1/6

You can express this statement with what's called a probability function. The following code cell draws a graph with probabilities on the y-axis and the number on the die on the x-axis.

In [5]:
```
# module import
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import randint

# initialise Figure and Axes
fig, ax = plt.subplots(figsize=[12, 6])

# initialise constants
low = 1
high = 7

# draw probability mass function
x = np.arange(low, high)  # x-axis positions
ax.plot(x, randint.pmf(x, low, high), 'o', ms=8,color='#17415f')  # blue dots
ax.vlines(x, 0, randint.pmf(x, low, high),color='#70b8e5', lw=5, alpha=0.5)  # blue li

# set Axes labels
ax.set(xlabel='Number on dice',
       ylabel='Probability',
       title='Probability function of a fair die');
```



Note the difference between the relative frequencies, which are the result of actual dice rolls, and the probabilities, which are a formulation of the data generating process. In general, the larger the data set, the closer the relative frequencies come to the probabilities. This insight is also called the law of large numbers.

So if you wanted to predict the number of the next roll, you could only estimate the theoretical probability of each number of points by using a data set. For example, you might discover that the die is loaded and that one number is more likely than another. Or you find out that it is a

fair die, as in this example. Either way, you should use a data set with a lot of rolls to train a machine learning algorithm, which will then imitate the data producing process.

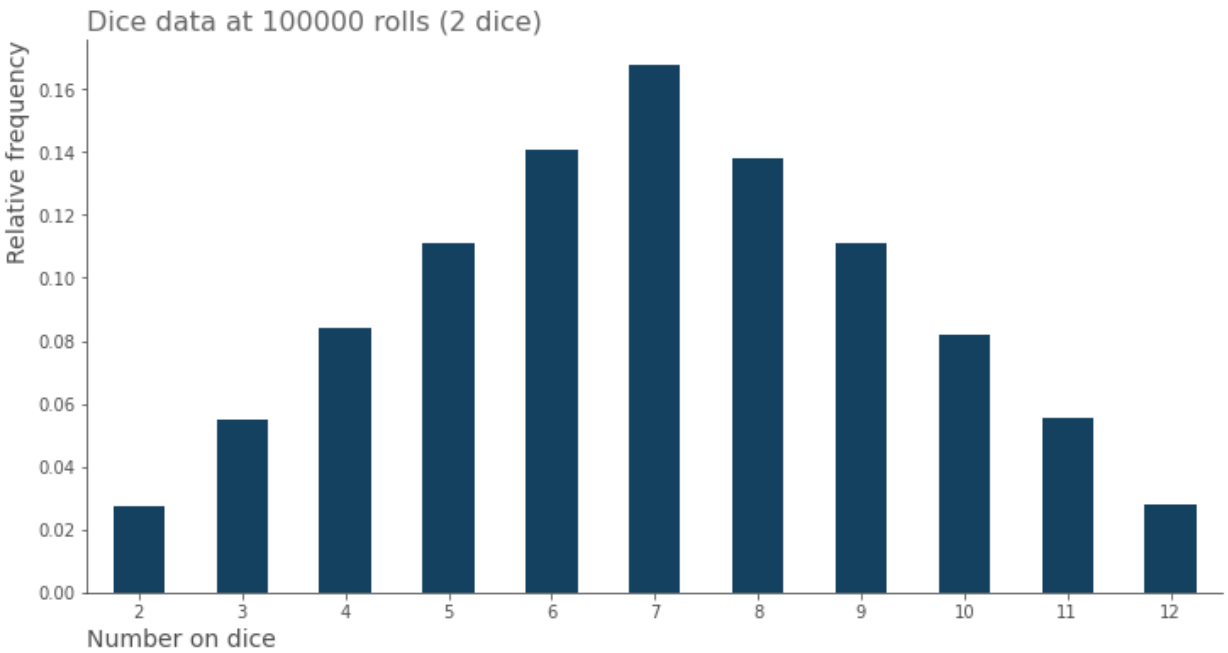
Assume that at BestBooks you have all the books of the new crime series "Dr. Macheath" (1-7) in stock and your customers are gradually buying all the books in a random order. Then you can assume, according to the law of large numbers, that as the number of total orders increases, the probability of buying one of the seven books will converge on 1/7, or 14.3%.

Congratulations: You have learned about the law of large numbers. it's one of many reasons why machine learning models require large data sets. This is the only way to estimate what kind of process created the data. In our case the process was rolling a die. There is a characteristic distribution: a uniform distribution. Each number on the die is therefore equally likely. But that is not necessarily the case with dice. We will look at an example of this next.

Rolling two dice

Rolling one die leads to a uniform distribution of the data in the long run. Each number on the die is therefore equally likely. Surprisingly, however, this all changes when you roll two dice. Let's assume we don't roll one die, but two, and add up their score. What is the relative frequency of the two numbers added together? If you have ever played the game **The Settlers of Catan**, this question will sound familiar. In this process, resource fields are assigned numbers on the dice. How many resources do you think you can with which numbers? With ten rolls the result might look like this:

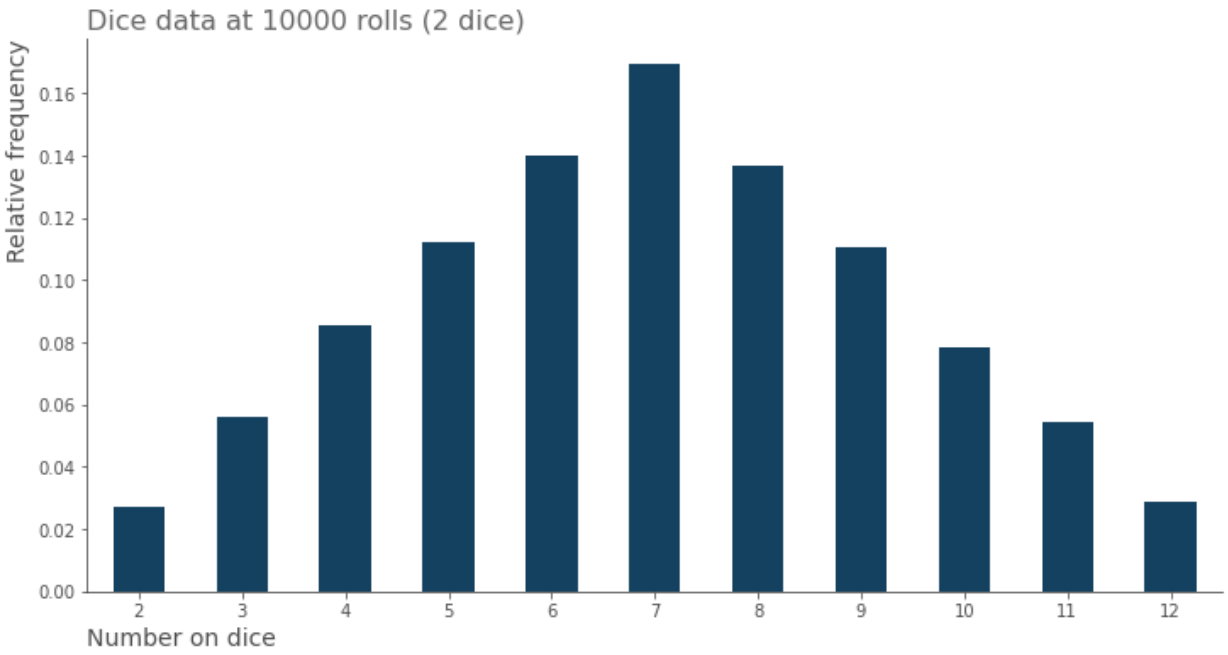
In [6]:
```
dice_plot(rolls=100000, dice=2);
```



Because we have a small data set, it is very difficult to estimate what kind of probabilities this process (sum of rolling two dice) probably operates under. Is it a uniform distribution again like last time, or something completely different? Thanks to the law of large numbers, we can

estimate the answer if we roll the two dice a lot of times. Write `dice_plot(rolls=10000, dice=2);` in the following cell.

In [7]:
```
dice_plot(rolls=10000, dice=2);
```



The relative frequency of the numbers rolled is surprisingly not equally distributed at all. We can pretty much rule out a uniform distribution here. What causes this distribution? With two dice there are a total of 36 possible combinations. The following table lists them all. The numbers of the first die are shown vertically and the numbers from the second die are shown horizonatally. Each cell shows the numbers on the two dice.

x	1	2	3	4	5	6
1	1;1	1;2	1;3	1;4	1;5	1;6
2	2;1	2;2	2;3	2;4	2;5	2;6
3	3;1	3;2	3;3	3;4	3;5	3;6
4	4;1	4;2	4;3	4;4	4;5	4;6
5	5;1	5;2	5;3	5;4	5;5	5;6
6	6;1	6;2	6;3	6;4	6;5	6;6

If you now add up the numbers for each combination, you get the following picture.

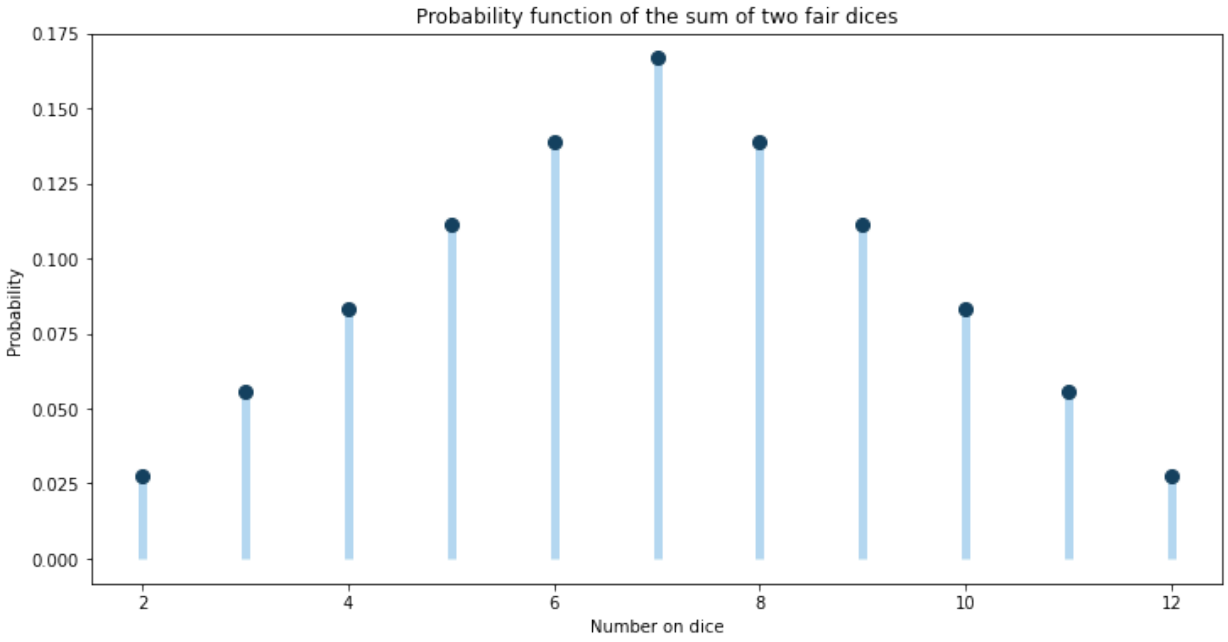
x	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10

x	1	2	3	4	5	6
5	6	7	8	9	10	11
6	7	8	9	10	11	12

If you study the added up numbers in the second table, it becomes clear why there is no uniform distribution. If you roll two dice, there are 36 possible combinations of the two dice that can occur. Only one of these 36 combinations results in a total of 2 (1;1). 2 of the 36 possible combinations result in a total of 3 (1;2 and 2;1). This continues until a you get to a total result of 7. Six of the 36 combinations produce this result (1;6, 2;5, 3;4, 4;3, 5;2, 6;1). Then more numbers are generated again by fewer and fewer combinations.

We can therefore imagine the formulations of the data generating process as follows:

In [8]:
```
# initialise Figure and Axes
fig, ax = plt.subplots(figsize=[12, 6])

# initialise constants
low = 2
high = 13

# prepare data for plotting
x = np.arange(low, high)  # x-axis positions
y1 = np.arange(low-1, 7)/36  # left half of y-axis positions
y = np.concatenate((y1, np.flip(y1[:-1])))  # all y-axis positions

# draw probability mass function
ax.plot(x, y, 'o', ms=8, color='#17415f')  # blue dots
ax.vlines(x, 0, y, color='#70b8e5', lw=5, alpha=0.5)  # blue lines

# set Axes labels
ax.set(xlabel='Number on dice',
       ylabel='Probability',
       title='Probability function of the sum of two fair dices');
```



But no matter which probability function a data-driven process follows, the probabilities of possible events always add up to 1, i.e. 100%. So if you add up the heights of the thin columns in the diagrams of the probability functions, you always end up with 1.

So if you want to win at Settlers of Catan, you should use resource fields that show numbers close to seven. If you wanted to imitate the data generation process, you should generate a total of seven the most. The probability function of the sum of two dice clearly shows which result is most likely.

Congratulations: You have learned the difference between the formulations of a data generating process and the results it produces. In this course, trained machine learning algorithms imitate data-generating processes that contain an element of chance to predict data. With these kinds of processes it is particularly difficult to deduce from a result what the formulations of the process are. We have now learned about a typical discrete probability function: uniform distribution. What other typical discrete probability functions are there? We will cover this in the next lesson.

Remember:

- You can find the relative frequencies in the data.
- Probabilities are the formulations of a data-generating process where chance plays a role.
- In the case of a uniform distribution, all results are equally likely.
