

Модуль random

Библиотека NumPy предоставляет широкие возможности по работе со случайными числами. Для этого в ней предусмотрен модуль `random`.

Модуль `random` — это модуль, который позволяет генерировать псевдослучайные числа и таким образом имитировать случайные события.

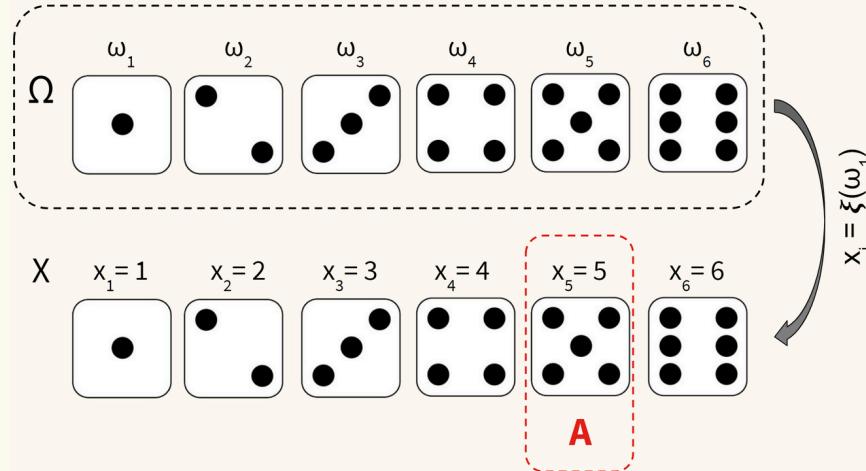
При этом прежде чем перейти к изучению кода, давайте познакомимся с некоторыми ключевыми понятиями теории вероятностей.

Случайная величина

Рассмотрим некоторое **событие** (event) A , например, выпадение пятерки на игральной кости. Это событие **случайно** (random process), так как оно может произойти, а может не произойти.

Испытанием (trial) или экспериментом будет процесс бросания игральной кости, а элементарными **исходами** (outcomes) испытания ω_i («омега» малое) — возможные результаты, то есть грани с числами от одного до шести.

Совокупность элементарных исходов называется **пространством** элементарных исходов (sample space) Ω («омега» большое).



При этом **случайная величина** (`random variable`) X — это численное значение результата испытания. Одновременно речь идет о функции ξ («кси»), которая отображает множество элементарных исходов ω_i на некоторое числовое множество $x_i = \xi(\omega_i)$.

Вероятность случайной величины

Хотя по определению случайное событие предсказать нельзя, мы можем предположить, с какой вероятностью оно произойдет. Начнем с классического или как еще говорят теоретического определения вероятности.

Классическая (теоретическая) вероятность

Пусть k — количество благоприятствующих событию A исходов (то есть исходов, которые влекут за собой наступление события A), а n — общее количество равновозможных исходов, тогда теоретической вероятностью события A будет

$$P(A) = \frac{k}{n}$$

Разберем это определение на том же примере бросания игральной кости.

В частности, выпадение двойки или тройки благоприятствует (приводит к наступлению) случайного события A . Таких исходов два (либо двойка, либо тройка) и соответственно $k = 2$. Всего исходов шесть и вероятность их наступления одинакова (они равновозможны). Значит $n = 6$. Рассчитаем теоретическую вероятность такого события.

$$P(A) = \frac{2}{6} = \frac{1}{3}$$

Аналогично, вероятность выпадения орла или решки на «честной» монете (fair coin) равна $1/2$, а вероятность вытащить червь из колоды, в которой 36 карт, равна $9/36$ или $1/4$.

Вероятность любого события **находится в диапазоне от 0 до 1**. При этом событие с нулевой вероятностью называется невозможным, а с вероятностью равной единице — достоверным. В промежутке находятся случайные события.

Сумма вероятностей событий, образующих полную группу, **всегда равна единице**.

Выпадение орла и выпадение решки образуют полную группу (других событий при подбрасывании монеты быть не может). Вероятность каждого из этих событий равна $1/2$, сумма этих вероятностей — единице.

Относительная (эмпирическая) вероятность

Эмпирическая вероятность события — это отношение частоты наступления события А к общему числу испытаний. По большому счету, мы бросаем кости несколько раз и считаем, сколько раз выпала двойка или тройка по отношению к общему количеству бросков.

Предположим, мы бросали кость 10 раз и двойка или тройка выпали 4 раза. Рассчитаем эмпирическую вероятность такого события.

$$P(A) = \frac{4}{10} = \frac{2}{5}$$

Испытаний (бросков) было проведено очень мало и эмпирическая вероятность довольно сильно отличается от теоретической. Что же будет, если кратно увеличить количество бросков?

Закон больших чисел

Закон больших чисел (Law of large numbers) утверждает, что при проведении множества испытаний эмпирическая вероятность начнет приближаться к теоретической. Другими словами, если бросать кость достаточноное количество раз, то эмпирическая вероятность выпадения двойки или тройки приблизится к 1/3.

Модуль random библиотеки Numpy

Начнем изучение модуля random с нескольких практических примеров. В частности посмотрим, как можно имитировать подбрасывание монеты и бросание игральной кости, убедимся в верности Закона больших чисел с помощью метода Монте-Карло, рассмотрим задачу о двух конвертах, а также познакомимся с методом случайных блужданий.

Подбрасывание монеты

Подбрасывание монеты можно имитировать, обозначив орел и решку через 0 и 1 и случайно генерируя соответствующие числа. Для этого подойдет **функция np.random.randint()**.

На входе функция принимает границы диапазона (верхняя граница в диапазон не входит), на выходе — генерирует целые псевдослучайные числа в его пределах.

```
In [ ]: import numpy as np  
import matplotlib.pyplot as plt
```

```
In [ ]:
```

Вероятность случайной величины

Подбрасывание монеты

```
In [ ]: # для этого подойдет функция randint()
np.random.randint(0, 2)
```

```
Out[ ]: 0
```

```
In [ ]: # можно создать массив из 10 чисел
np.random.randint(0, 2, 10)
```

```
Out[ ]: array([1, 1, 1, 1, 0, 1, 0, 0, 0, 0])
```

Бросание кости

```
In [ ]: # изменив диапазон, можно имитировать бросание кости
np.random.randint(1, 7, 10)
```

```
Out[ ]: array([5, 6, 6, 1, 2, 1, 4, 3, 2, 1])
```

Метод Монте-Карло

Теперь давайте вернемся к Закону больших чисел. Для его проверки воспользуемся методом Монте-Карло.

Метод Монте-Карло позволяет исследовать какой-либо случайный процесс, многократно имитируя его с помощью компьютера.

В нашем случае мы будем многократно бросать игральную кость и для каждой серии испытаний рассчитывать эмпирическую вероятность выпадения двойки или тройки. Если с ростом количества испытаний в серии эмпирическая вероятность приблизится к теоретической, мы сможем подтвердить Закон больших чисел.

Будем создавать нашу программу поэтапно.

Шаг 1. Метод Монте-Карло

Вначале нам потребуется список серий с количеством испытаний в каждой из них. Причем для наглядности пусть количество испытаний от серии к серии увеличивается экспоненциально.

Создать такой список, вернее массив NumPy, можно с помощью функции `np.logspace()`. Она похожа на уже изученную нами функцию `np.linspace()` с той лишь разницей, что возвращает заданное количество чисел, которые равномерно распределены по логарифмической шкале в пределах заданного диапазона.

Пусть наша последовательность будет состоять из чисел 10, введенных в степень от 1 до 6. То есть первым числом будет $10^1 = 10$, а последним $10^6 = 1000000$. Всего таких чисел будет 50 (значение функции `np.logspace()` по умолчанию).

Метод Монте-Карло

Шаг 1. Количество испытаний в каждой серии

```
In [ ]: # создадим экспоненциально растущую последовательность из 50-ти целых чисел
# со степенями десяти от 1 до 6
series = np.logspace(1, 6, dtype = 'int')
series
```

```
Out[ ]: array([    10,     12,     15,     20,     25,     32,     40,
      51,     65,     82,    104,    132,    167,    212,
     268,    339,    429,    542,    686,    868,   1098,
    1389,   1757,   2222,   2811,   3556,   4498,   5689,
    7196,   9102,  11513,  14563,  18420,  23299,  29470,
   37275,  47148,  59636,  75431,  95409, 120679, 152641,
  193069, 244205, 308884, 390693, 494171, 625055, 790604,
 1000000])
```

Промежуточные числа NumPy рассчитает самостоятельно с помощью десятичного логарифма. Например, для получения числа 790604 мы возвели 10 в степень 5.897959007291967.

```
In [ ]: # посмотрим на степень, в которую нужно возвести 10, чтобы получить 790604  
np.log10(790604)
```

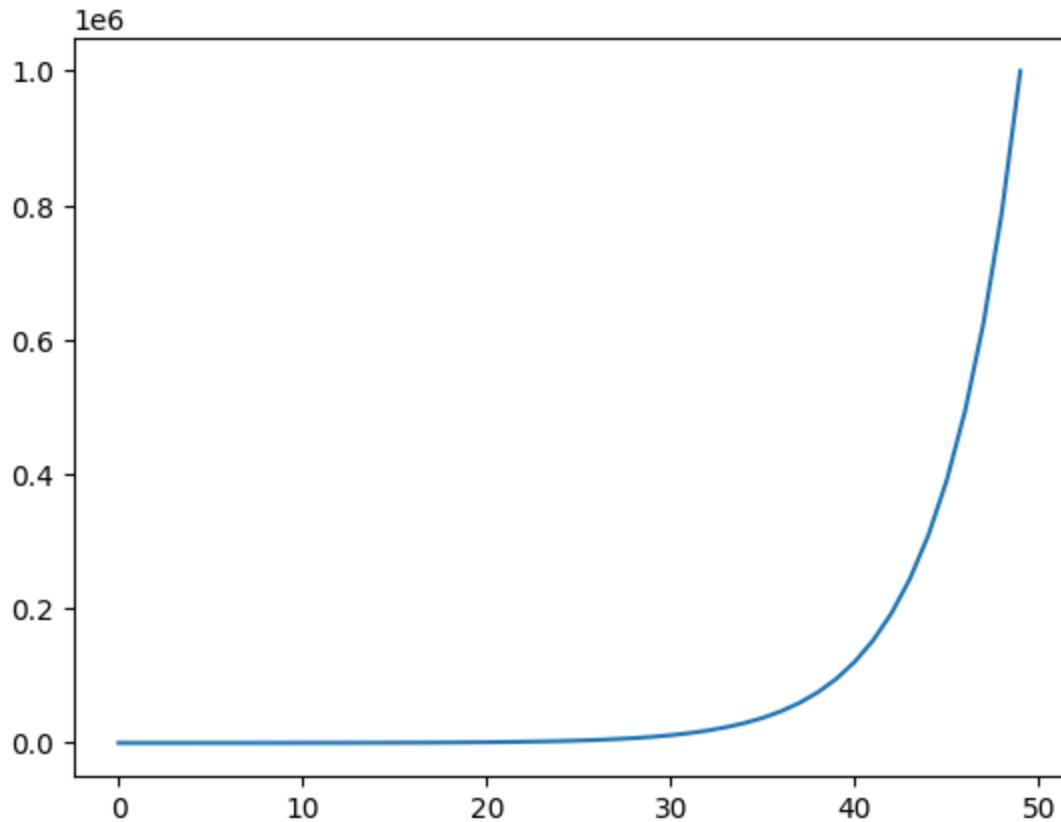
```
Out[ ]: 5.897959007291967
```

```
In [ ]: # проверим этот результат  
10 ** np.log10(790604)
```

```
Out[ ]: 790604.0000000002
```

```
In [ ]: # посмотрим на получившуюся последовательность на графике  
plt.plot(series)
```

```
Out[ ]: <matplotlib.lines.Line2D at 0x79b8195ff310>
```



Шаг 2. Напишем код для проведения испытаний

Используем два цикла:

с помощью первого (внешнего) цикла `for` мы можем имитировать серии бросков (всего 50 испытаний);
второй (внутренний) цикл отвечает за броски каждой отдельной

серии; в первый раз мы бросим кость 10 раз, во второй – 12 и так далее.

Пока ограничимся одной серией. Для этого в конце внешнего цикла поставим оператор `break`

```
In [ ]: # обозначим точку отсчета
np.random.seed(42)

# во внешнем цикле пройдемся по сериям испытаний (их будет 50)
for i, trial in enumerate(series, 1):

    # выведем серию и количество бросков внутри нее
    print(f'Серия: {i}, количество испытаний: {trial}\n')

    # во внутреннем цикле будем бросать кость
    for n in range(trial):

        # запишем результат каждого броска
        result = np.random.randint(1, 7)

        # выведем номер броска и соответствующий результат
        print(f'Испытание: {n}, выпало число: {result}')

    # прервемся после первой серии
    break
```

Серия: 1, количество испытаний: 10

```
Испытание: 0, выпало число: 4
Испытание: 1, выпало число: 5
Испытание: 2, выпало число: 3
Испытание: 3, выпало число: 5
Испытание: 4, выпало число: 5
Испытание: 5, выпало число: 2
Испытание: 6, выпало число: 3
Испытание: 7, выпало число: 3
Испытание: 8, выпало число: 3
Испытание: 9, выпало число: 5
```

Шаг 3. Рассчитаем долю успешных испытаний

Создадим счетчик, в который будем записывать выпадения двойки или тройки. После окончания очередной серии испытаний разделим количество успешных бросков на общее количество испытаний.

```
In [ ]: # обозначим точку отсчета
np.random.seed(42)

# создадим счетчик для количества успешных испытаний
success = 0

# в двух циклах пройдемся по сериям и броскам в каждой серии
for trial in series:
```

```

for n in range(trial):
    result = np.random.randint(1, 7)

    # если результат будет равен двум или трем
    if result == 2 or result == 3:

        # обновим счетчик
        success += 1

    # в конце каждой серии посчитаем долю успешных испытаний
    prob = success / trial

    print(f'Серия: {i}\n') # и выведем номер серии,
    print(f'Всего испытаний: {trial}') # общее количество испытаний,
    print(f'Успешных исходов: {success}') # количество и
    print(f'Эмпирическая вероятность: {prob}') # долю успешных испытаний

    # прервемся после окончания первой серии
    break

```

Серия: 1

Всего испытаний: 10
 Успешных исходов: 5
 Эмпирическая вероятность: 0.5

Обратите внимание, эмпирическая вероятность первой серии, в которой было только 10 бросков (1/2), далека от теоретической (1/3).

Шаг 4. Проведем полноценные испытания алгоритма

Теперь мы готовы протестировать наш алгоритм на всех 50-ти сериях.

Дополним наш код списком, в который мы будем записывать долю успешных бросков (эмпирическую вероятность) в каждой серии.

Помимо этого будем замерять время исполнения кода с помощью магической команды %time.

```

In [ ]: # замерим время исполнения ячейки
%time

# обозначим точку отсчета
np.random.seed(42)

# создадим счетчик для испытаний в каждой серии
success = 0

# а также список для записи результатов в пределах одной серии
prob_list = []

# пройдемся по сериям
for trial in series:

    # и броскам в каждой серии

```

```

for n in range(trial):
    result = np.random.randint(1, 7)

    # посчитаем количество успешных бросков
    if result == 2 or result == 3:
        success += 1

    # вычислим долю успешных бросков
    prob = success / trial

    # добавим результат серии в список
    prob_list.append(prob)

    # обнулим счетчик для записи результата следующей серии
    success = 0

# после проведения всех серий испытаний выведем их количество
print(f'Проведено серий испытаний: {len(prob_list)}')

# а также эмпирическую вероятность отдельных серий
print(f'Эмпирическая вероятность каждой пятой серии: {np.round(prob_list[::5])

```

Проведено серий испытаний: 50
 Эмпирическая вероятность каждой пятой серии: [0.5 0.22 0.34 0.37 0.34 0.33
 0.34 0.34 0.33 0.33]
 CPU times: user 11 s, sys: 17.2 ms, total: 11.1 s
 Wall time: 11.2 s

Шаг 5. Визуализация результата

```

In [ ]: # зададим размер графика
plt.figure(figsize = (10, 8))

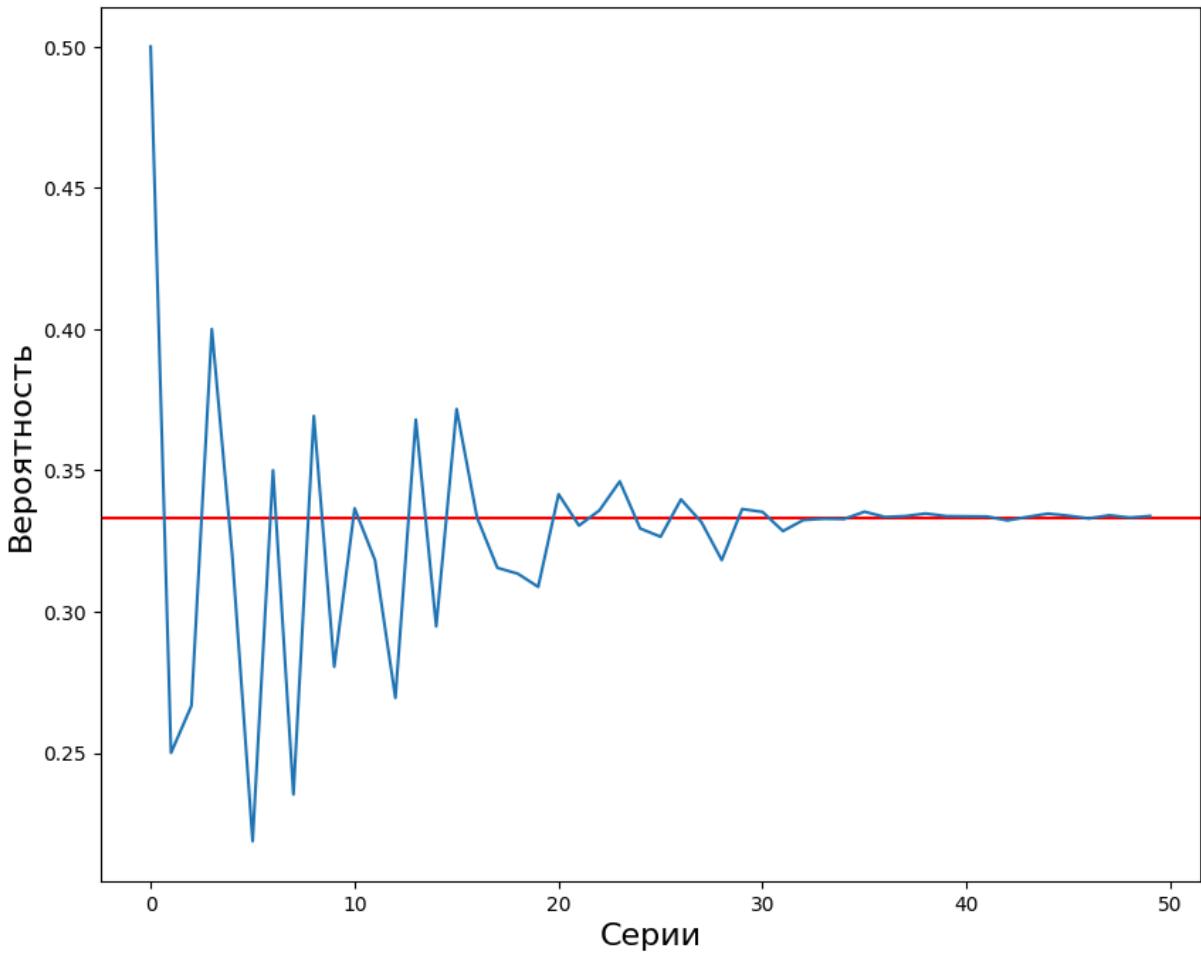
# выведем горизонтальную линию теоретической вероятности
plt.axhline(y = 1/3, c = 'r')

# а также эмпирическую вероятность по результатам 50-ти серий испытаний
plt.plot(prob_list)

# добавим подписи
plt.xlabel('Серии', fontsize = 16)
plt.ylabel('Вероятность', fontsize = 16)

plt.show()

```



Как мы видим, если в первых сериях эмпирическая вероятность довольно сильно отличалась от теоретической, с увеличением количества бросков она вплотную приблизилась к $1/3$.

Мы убедились в верности Закона больших чисел с помощью метода Монте-Карло.

Пример векторизованного кода

На прошлом занятии мы поговорили про векторизацию кода и выяснили, что векторизованный код исполняется быстрее, чем циклы `for`. Давайте посмотрим, нет ли возможности оптимизировать код с двумя циклами, написанный ранее.

```
In [ ]: %%time
np.random.seed(42)

# создадим новый список для записи результатов
prob_list_2 = []

# пройдемся по каждой из 50-ти серий
for trial in series:
```

```
# вместо второго цикла, передадим количество бросков
# непосредственно в функцию np.random.randint() через переменную trial
# и запишем полученные данные в массив result
result = np.random.randint(1, 7, trial)

# в переменную success запишем сумму выпавших двоек или троек из массива result
success = (result == 2).sum() + (result == 3).sum()

# посчитаем долю успешных исходов
prob = success / trial

# и запишем ее в список
prob_list_2.append(prob)
```

```
CPU times: user 52.2 ms, sys: 1.01 ms, total: 53.2 ms
Wall time: 53.4 ms
```

```
In [ ]: # убедимся, что при одинаковой точке отсчета, результат будет идентичным
prob_list == prob_list_2
```

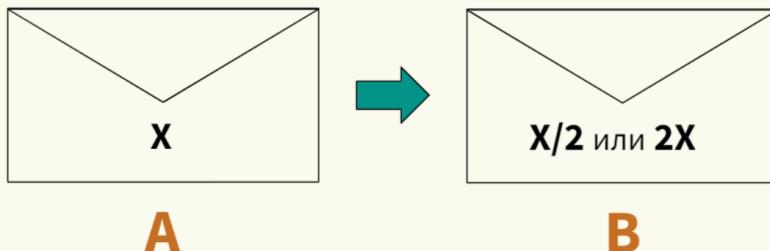
```
Out[ ]: True
```

Задача о двух конвертах

Задача о двух конвертах

Задача о двух конвертах (two envelopes problem) — популярная задача из области теории вероятностей. Сегодня мы рассмотрим один из ее вариантов и смоделируем решение с помощью модуля `random`.

Представьте, что вам дали конверт А, вы открыли его и увидели сумму X . После этого вам предлагают заменить этот конверт на другой конверт В, сумма в котором с одинаковой вероятностью либо в два раза меньше, либо в два раза больше суммы X . Стоит ли вам заменить конверт с А на В?



Аналитическое решение

Решим задачу аналитически. Пусть суммы в конверте В могут быть равны $X/2$ и $2X$, тогда в среднем результат замены конвертов составит

$$\frac{1}{2} \left(\frac{X}{2} \right) + \frac{1}{2}(2X) = \frac{5}{4}X$$

Это больше, чем изначальная сумма X в конверте А, а значит нам стоит взять конверт В.

Теперь рассмотрим конкретный пример. Предположим, что в открытом нами конверте А было 5000 рублей, тогда в среднем при замене конвертов мы получим

$$\frac{1}{2} \left(\frac{5000}{2} \right) + \frac{1}{2}(2 \cdot 5000) = \frac{5}{4} \cdot 5000 = 6250$$

Моделирование с помощью модуля `random`

Проверим это решение с помощью Питона и Закона больших чисел.

```
In [ ]: # зададим точку отсчета и
         np.random.seed(50)

# количество испытаний
trials = 100000

# создадим список для записи результатов
outcomes = []
```

```
# с помощью цикла for
for trial in range(trials):

    # будем с одинаковой вероятностью генерировать числа 0 и 1
    outcome = np.random.randint(0, 2)

    # если выпадет ноль, то мы получим 2500 рублей
    if outcome == 0:

        # запишем этот результат в список
        outcomes.append(2500)

    else:

        # в противном случае, запишем 10000
        outcomes.append(10000)

# посчитаем среднее значение
sum(outcomes)/len(outcomes)
```

Out[]: 6249.475

Как мы видим, компьютер подтвердил, что в среднем (при достаточном количестве испытаний) будет выгоднее заменить конверт А на конверт В.

Случайное блуждание по числовой прямой

Парадокс двух конвертов

Добавлю, что если изменить условие задачи и не открывать конверт А, то задача превратится в **парадокс двух конвертов** (two envelopes paradox). Ведь если в среднем выгоднее заменить конверт А на В, то будет выгодно и обратное действие (заменить В на А).

Такой обмен конвертами может длиться бесконечно, а это приводит к противоречию поскольку выше мы доказали, что изначальная замена конверта А на В более выгодна.

Рассмотрение этого парадокса выходит за рамки сегодняшнего занятия.

Метод случайных блужданий

Генератор случайных чисел также применяется в так называемом **методе случайных блужданий** (random walk models, random walkers). С его помощью мы можем конструировать путь, состоящий из последовательности случайных шагов в определенном пространстве и, таким образом, моделировать различные процессы.

Случайное блуждание по числовой прямой

В самом простом варианте мы будем подбрасывать монету и смотреть, куда мы сдвинулись относительно предыдущего шага на числовой прямой. Посмотрим на алгоритм.

```
In [ ]: # поставим точку отсчета
np.random.seed(2)

# создадим список и запишем в него исходную позицию
moves = [0]

# для наглядности запишем результат бросания монеты
coins = []

for x in range(5):

    # в цикле подбросим монету
    coin = np.random.randint(0, 2)

    # если выпадет 1, сдвинемся на одно значение от предыдущего результата
    # если выпадет 0, останемся на месте
    # запишем результат в список moves
    moves.append(moves[x] + coin)

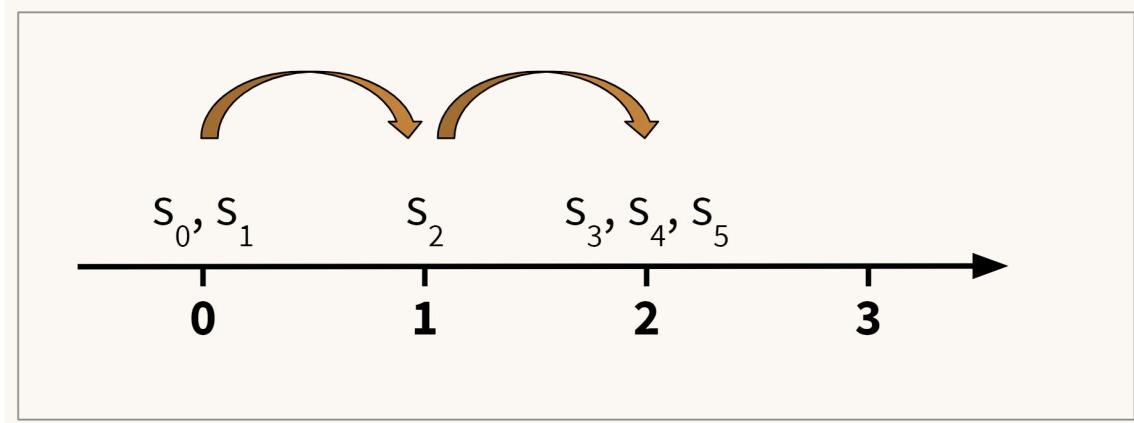
    # также запишем результат подбрасывания монеты
    coins.append(coin)
```

```
In [ ]: # посмотрим на сделанные шаги
moves
```

```
Out[ ]: [0, 0, 1, 2, 2, 2]
```

```
In [ ]: # посмотрим на результаты подбрасывания монеты
coins
```

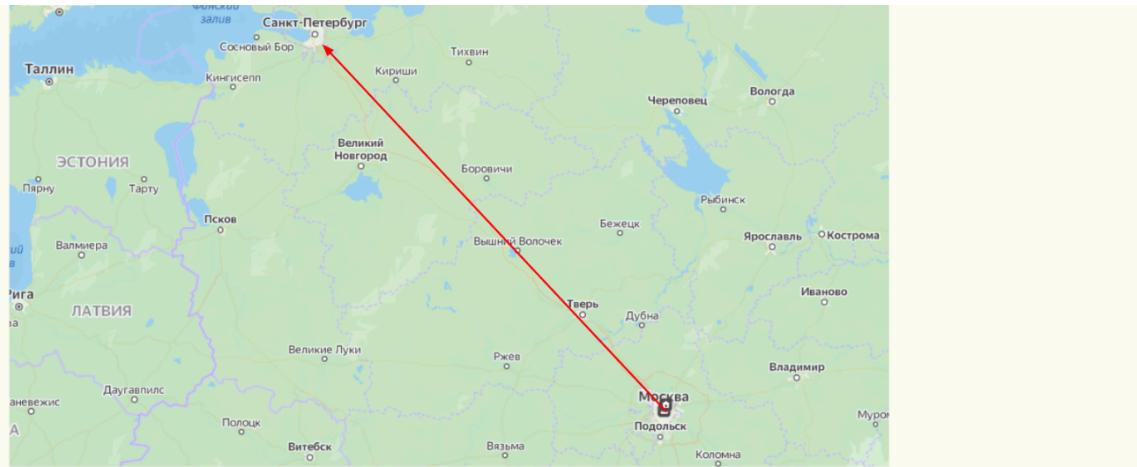
```
Out[ ]: [0, 1, 1, 0, 0]
```



Начальное положение (s_0) находится в точке ноль. При первом подбрасывании монеты выпадает 0, и на первом шаге (s_1) мы остаемся на месте. На втором шаге (s_2) выпадает единица, и мы сдвигаемся в точку один. На третьем шаге (s_3) мы перемещаемся в точку два. После этого, на шагах четыре (s_4) и пять (s_5) мы остаемся на месте.

Доеедем ли мы из Москвы до Санкт-Петербурга?

Функция случайного блуждания



Теперь усложним задачу и посмотрим, сможем ли мы добраться из Москвы до Санкт-Петербурга на довольно необычном транспортном средстве, движение которого зависит от бросания игральной кости. Условия следующие:

- нам нужно преодолеть ровно 700 километров;
- если при бросании кости выпадет единица или двойка, мы сместимся на 5 километров вперед;
- если тройка или четверка, то на 5 километров назад;
- в случае если выпадет пять или шесть, мы повторно бросим кость и сдвинемся на выпавшее число километров умноженное на пять;
- кроме того, есть вероятность внезапной поломки: она составляет менее 0,001, то есть менее 0,1%, этом случае на эвакуаторе нас отвезут обратно в Москву для ремонта.

Дополнительно договоримся, что в обратном направлении мы двигаться не можем, то есть если нам будет все время не везти, в минус мы не уйдем, число пройденных километров не должно быть отрицательным.

Теперь давайте с помощью Питона построим такой механизм.

- Как уже было сказано, для бросания игральной кости мы будем использовать функцию `np.random.randint()` с параметрами 1 и 7.
- Для того чтобы не уйти в минус, используем **функцию `max()`**. Первым параметром передадим 0, вторым — километр, на котором мы окажемся, если сдвинемся назад (то есть если выпадет твойка или четверка). В случае если второе число окажется отрицательным, то функция `max()` выберет ноль.
- Поломку мы будем имитировать с помощью **функции `np.random.rand()`**. Эта функция генерирует равномерное распределение (об этом ниже) чисел в интервале [0, 1] и, если выпавшее число меньше, чем 0,001, мы вернемся в начало пути.

```
In [ ]: # обяявим функцию random_walk()
def random_walk():

    # создадим список для записи перемещений от Москвы до Санкт-Петербурга
    random_walk = [0]

    # в цикле из 100 итераций
    for m in range(99):

        # запишем последний достигнутый километр в переменную move
        move = random_walk[-1]

        # бросим кость
        dice = np.random.randint(1, 7)

        # если выпадет 1 или 2, сместимся на пять вперед
        if dice <= 2:
            move = move + 5

        # если выпадет 3 или 4, на пять назад, но в любом случае остановимся на
        elif dice <= 4:
            move = max(0, move - 5)

        # если выпадет 5 или 6, бросим кость еще раз и умножим результат на пять
        # это и будет следующее перемещение
        else:
            move = move + 5 * np.random.randint(1, 7)

        # в случае внезапной поломки вернемся на нулевой километр
        if np.random.rand() < 0.001:
            move = 0

        # запишем каждое движение в список random_walk
        random_walk.append(move)

    # вернем этот список при вызове функции
    return random_walk
```

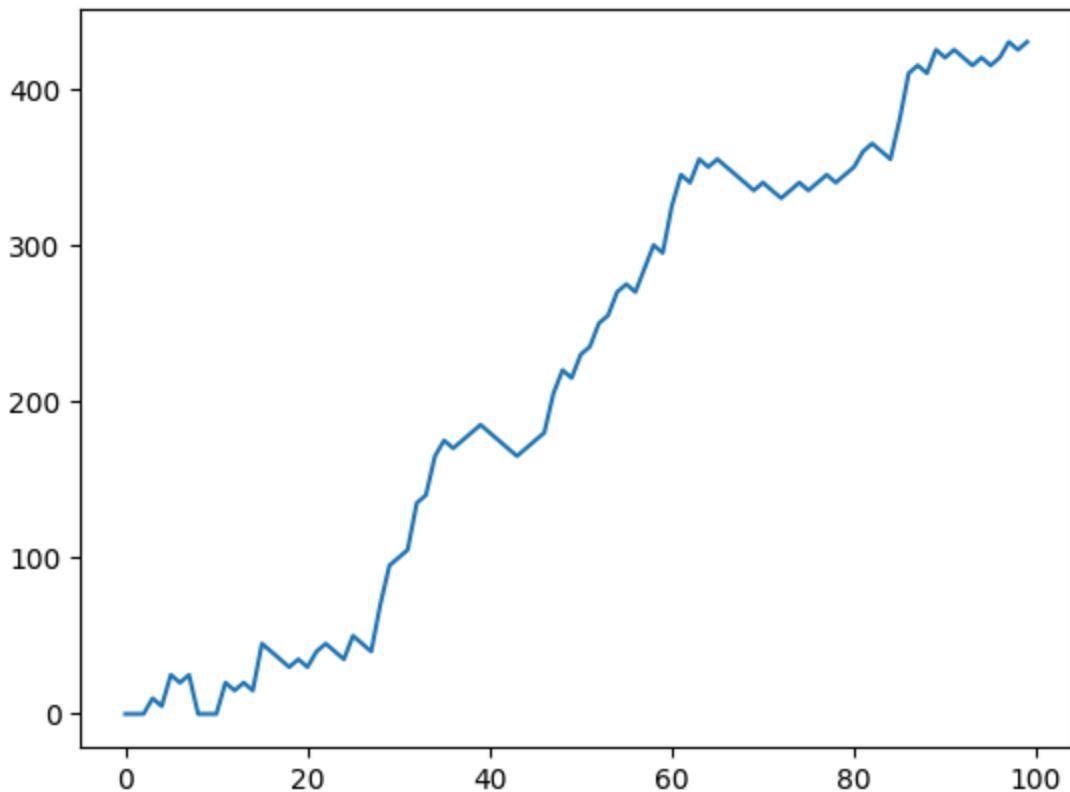
Создание одного случайного блуждания

```
In [ ]: # зададим точку отсчета
np.random.seed(42)

# вызовем функцию random_walk() и поместим результат в переменную rw
rw = random_walk()
```

```
In [ ]: # посмотрим на результат на графике
plt.plot(rw)
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x79b81968fc0>]
```



```
In [ ]: # посмотрим на расстояние, которое удалось преодолеть  
rw[-1]
```

```
Out[ ]: 430
```

Моделирование нескольких случайных блужданий

```
In [ ]: # установим точку отсчета  
np.random.seed(42)  
  
# создадим список, в который будем помещать отдельные случайные блуждания  
all_walks = []  
  
# в цикле из 1000 итераций  
for w in range(1000):  
  
    # сгенерируем случайное блуждание  
    rw = random_walk()  
  
    # и поместим его в список all_walks  
    all_walks.append(rw)
```

```
In [ ]: # превратим список списков в массив Numpy и транспонируем,  
all_walks_T = np.array(all_walks).T  
  
# чтобы отдельное случайное блуждание стало столбцом  
all_walks_T.shape
```

```
Out[ ]: (100, 1000)
```

```
In [ ]: # посмотрим на это в формате DataFrame
import pandas as pd

# для этого используем функцию pd.DataFrame()
df = pd.DataFrame(all_walks_T)

# и выведем последние пять строк
df.tail()
```

```
Out[ ]:   0   1   2   3   4   5   6   7   8   9   ...  990  991  992  993
95  415  645  785  565  355  595  545  640  665  585  ...  745  715  560  375
96  420  655  790  580  360  590  550  670  670  590  ...  740  745  555  390
97  430  665  815  600  355  585  545  675  690  610  ...  735  750  575  415
98  425  660  820  605  370  580  540  680  695  605  ...  730  745  570  420
99  430  665  815  610  365  585  535  685  700  600  ...  755  755  575  415
```

5 rows × 1000 columns

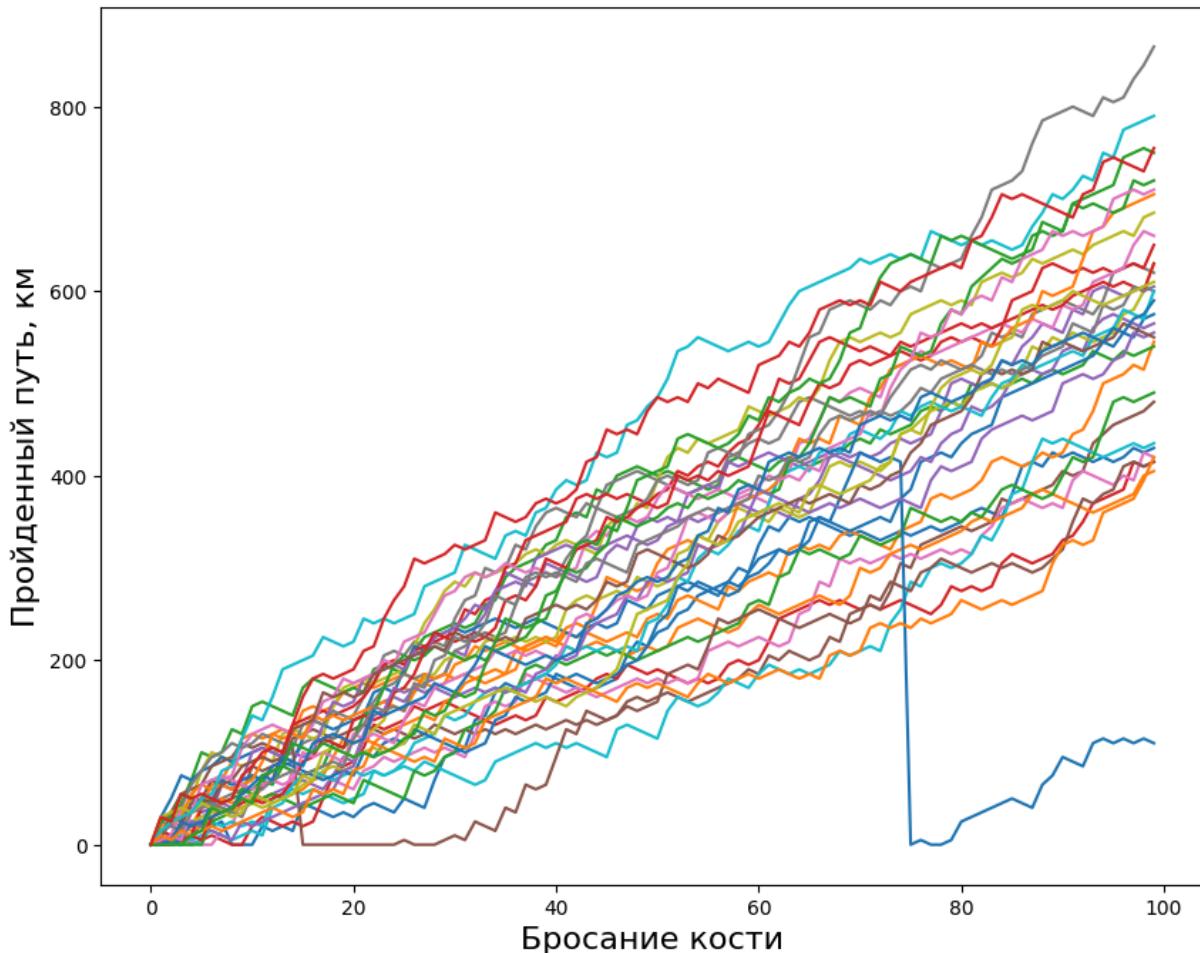
Первый столбец соответствует первому случайному блужданию и на шаге 100 (с индексом 99) мы достигли 430-ого километра.

```
In [ ]: # зададим размер графика
plt.figure(figsize = (10, 8))

# и выведем каждое 30-ое блуждание
plt.plot(all_walks_T[:, ::30])

# добавим подписи
plt.xlabel('Бросание кости', fontsize = 16)
plt.ylabel('Пройденный путь, км', fontsize = 16)

plt.show()
```



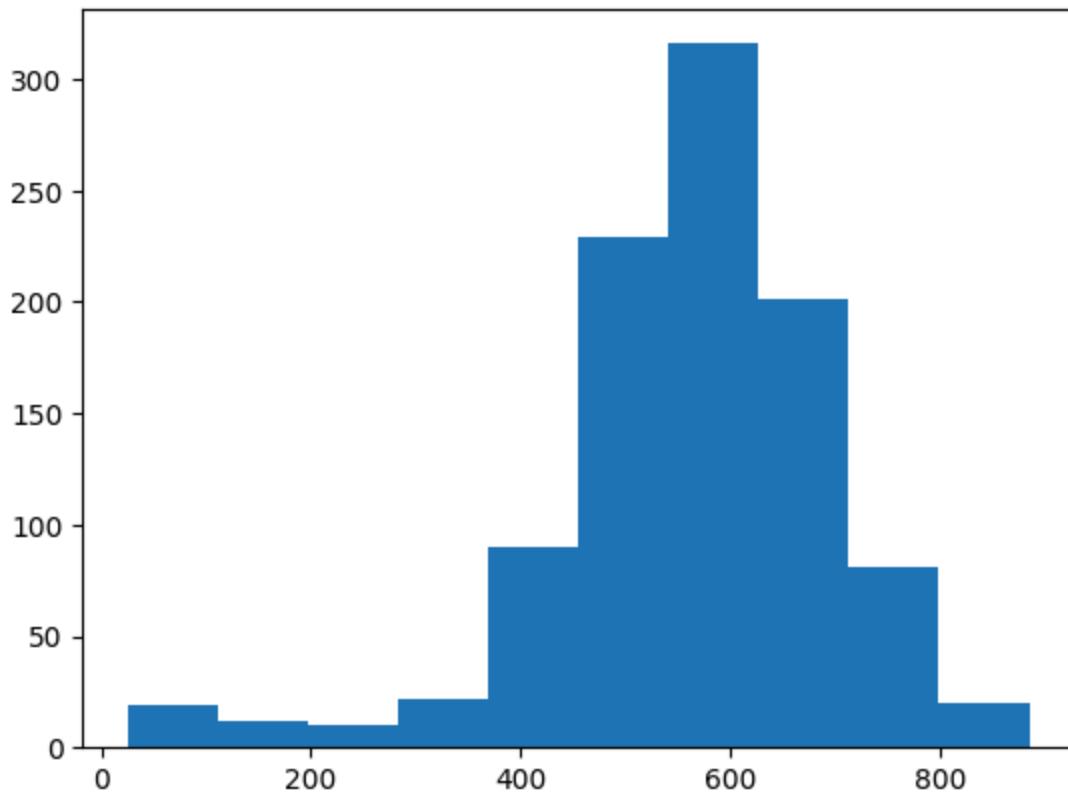
Обратите внимание на «упавшие» кривые. Это как раз те случаи, когда наше воображаемое транспортное средство ломалось, и мы возвращались в Москву на ремонт.

Вероятность преодоления пути

Теперь давайте возьмем достигнутые конечные точки каждого из 1000 испытаний и посмотрим на их распределение с помощью гистограммы.

```
In [ ]: # возьмем последнюю строку нашего массива
ends = all_walks_T[-1,:]

# и построим гистограмму с помощью plt.hist()
plt.hist(ends)
plt.show()
```



```
In [ ]: # посмотрим на средний преодоленный путь  
ends.mean()
```

```
Out[ ]: 561.415
```

Вероятностное распределение

```
In [ ]: # посмотрим на долю успешных поездок  
np.count_nonzero(ends >= 700)/len(ends)
```

```
Out[ ]: 0.128
```

Кроме того, мы можем рассчитать вероятность преодоления 700 км. Для этого с помощью функции np.count_nonzero() рассчитаем количество блужданий, преодолевших расстояние между двумя столицами, и разделим получившийся результат на общее количество испытаний.

Транспортное средство не слишком надежно, лишь в 12,8% случаев мы доехали до конечной точки.

Дискретные и непрерывные случайные величины

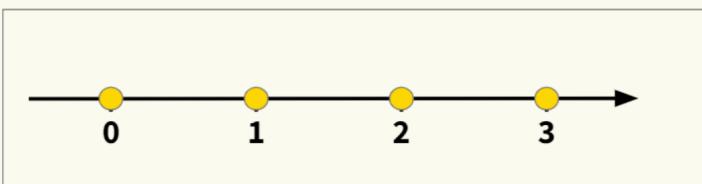
Случайные величины делятся на дискретные и непрерывные.

Дискретная случайная величина

Дискретная случайная величина — это случайная величина, значения которой конечны и счетны.

Помимо бросания игральной кости, классическими примерами дискретной случайной величины являются подбрасывания монеты или раздача колоды карт. Пример из повседневной жизни — количество автомобилей, проезжающих по улице города за сутки.

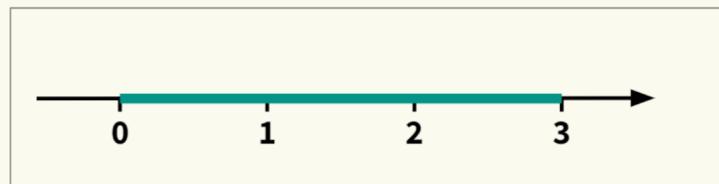
На числовой прямой такая величина может быть отмечена только на определенных точках.



Непрерывная случайная величина

Непрерывная случайная величина может быть получена в результате измерений (например, рост человека, температура воздуха, время ожидания в очереди и так далее).

На числовой прямой такая величина может принимать любое значение в пределах заданного интервала.



Вероятностное распределение

А теперь давайте попробуем провести множество подсчётов (для дискретной) и измерений (для непрерывной) случайной величины и вывести их на одном графике. Начнем с дискретных величин.

Дискретное вероятностное распределение

Мы снова будем бросать игральную кость, но на этот раз существенно увеличим количество испытаний.

Дискретное распределение

Равномерное распределение

```
In [ ]: # зададим точку отсчета
        np.random.seed(42)

        # бросим кость 100 000 раз
        dice = np.random.randint(1, 7, 100000)

        # выведем первые 10 результатов
        dice[:10]
```

```
Out[ ]: array([4, 5, 3, 5, 5, 2, 3, 3, 3, 5])
```

```
In [ ]: # функция np.unique() возвращает перечень уникальных элементов массива
        # и их количество при параметре return_counts = True
        elements, counts = np.unique(dice, return_counts = True)

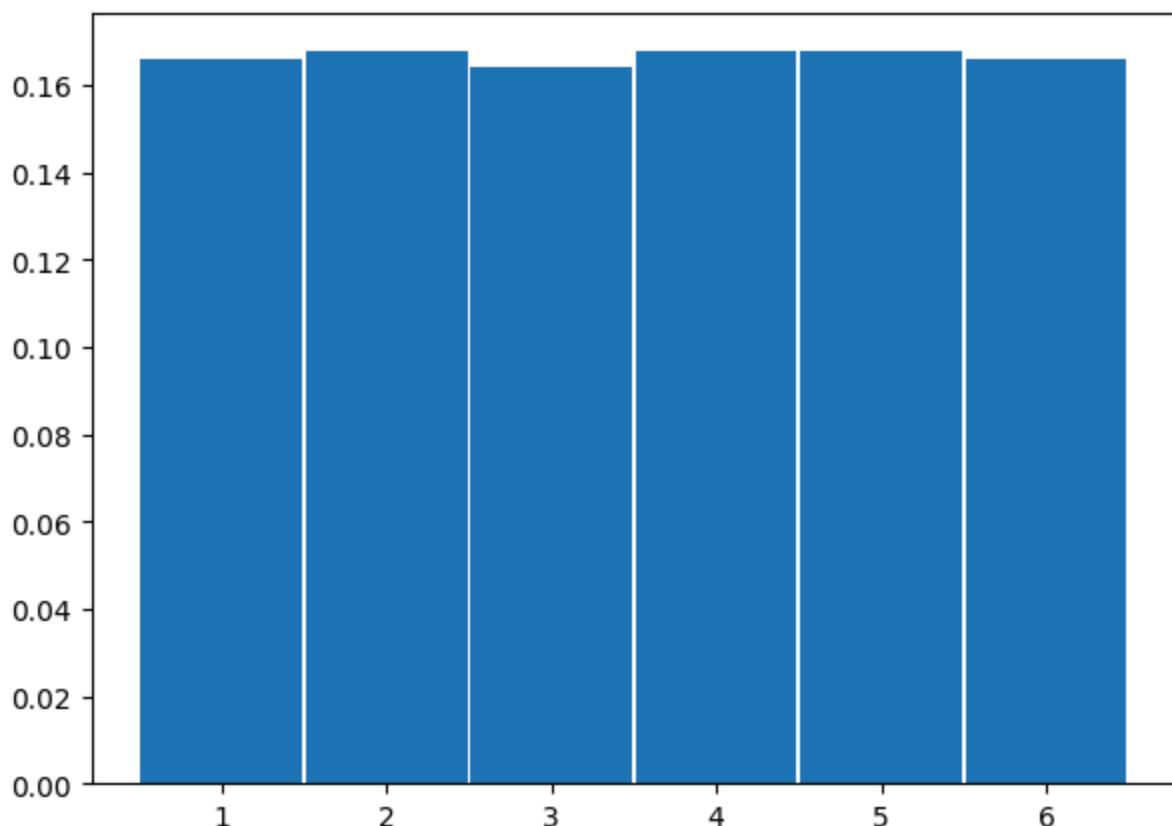
        print(elements)
        print(counts)
```

```
[1 2 3 4 5 6]  
[16592 16799 16390 16776 16810 16633]
```

```
In [ ]: # выполним поэлементное деление частоты каждого исхода на общее количество бросков  
rel_freq = (counts / len(dice)).round(3)  
rel_freq
```

```
Out[ ]: array([0.166, 0.168, 0.164, 0.168, 0.168, 0.166])
```

```
In [ ]: # зададим размер графика  
plt.figure(figsize = (7,5))  
  
# передадим категории (элементы) и их относительную частоту  
# параметр width контролирует ширину столбца  
plt.bar(elements, rel_freq, width = 0.98)  
plt.show()
```



Обратите внимание, с помощью графика мы можем оценить вероятность каждого из исходов. Именно для этого и нужно вероятностное распределение. Мы можем оценить случайный процесс в целом.

Равномерное дискретное распределение

Как вы видите, вероятность наступления каждого исхода примерно одинакова. Такое распределение называется **равномерным** (discrete uniform distribution). Именно его и использует функция `np.random.randint()` при генерации чисел.

Небольшие расхождения вероятностей каждого из шести исходов объясняются тем, что мы используем эмпирическую, а не теоретическую вероятность. Теоретически эти вероятности одинаковы.

Посмотрим на нотацию равномерного распределения

$$X \sim U(a, b)$$

В данном случае a и b — минимальное и максимальное значения случайной величины X . При бросании кости a и b равны одному и шести соответственно.

$$X \sim U(1, 6)$$

Функция вероятности

Помимо графика, дискретное вероятностное распределение удобно описывать с помощью **функции вероятности** (probability mass function, pmf). Такая функция возвращает вероятность того, что случайная величина примет определенное значение.

В случае равномерного распределения вероятность каждого исхода одинакова, а функция вероятности чрезвычайно проста.

$$pmf = \frac{1}{n}$$

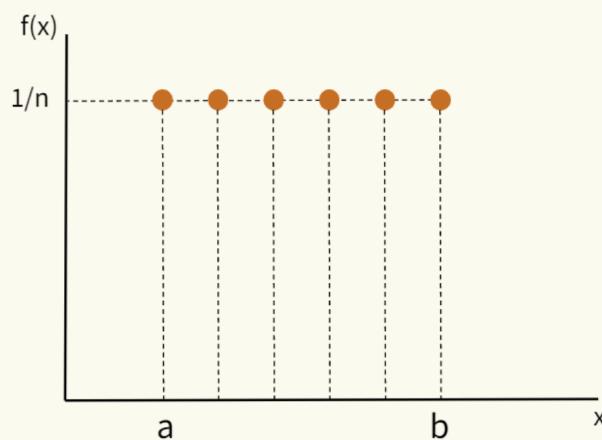
Параметр n можно рассчитать по формуле

$$n = b - a + 1$$

Вероятность каждого исхода, таким образом, будет равна $1/6$.

$$pmf = \frac{1}{6 - 1 + 1} = \frac{1}{6}$$

Рассмотрим эту функцию на графике.



Функцию вероятности рассматриваемой нами дискретной величины также можно представить в форме таблицы.

Исход	1	2	3	4	5	6
Вероятность	$1/6$	$1/6$	$1/6$	$1/6$	$1/6$	$1/6$

Еще раз обращу ваше внимание на то, что сумма вероятностей любого распределения равна единице.

```
In [ ]: # убедимся, что сумма всех вероятностей равна единице  
np.sum(rel_freq).round()
```

```
Out[ ]: 1.0
```

Функция распределения

Одновременно, вероятностное распределение бывает удобно описать с помощью (кумулятивной, накопительной) **функции распределения** (cumulative distribution function, cdf). Эта функция возвращает вероятность того, что случайная величина примет значение меньше заданного. Говоря неформально, она показывает «накопившуюся» вероятность исходов.

Для равномерного дискретного распределения функция распределения выглядит следующим образом

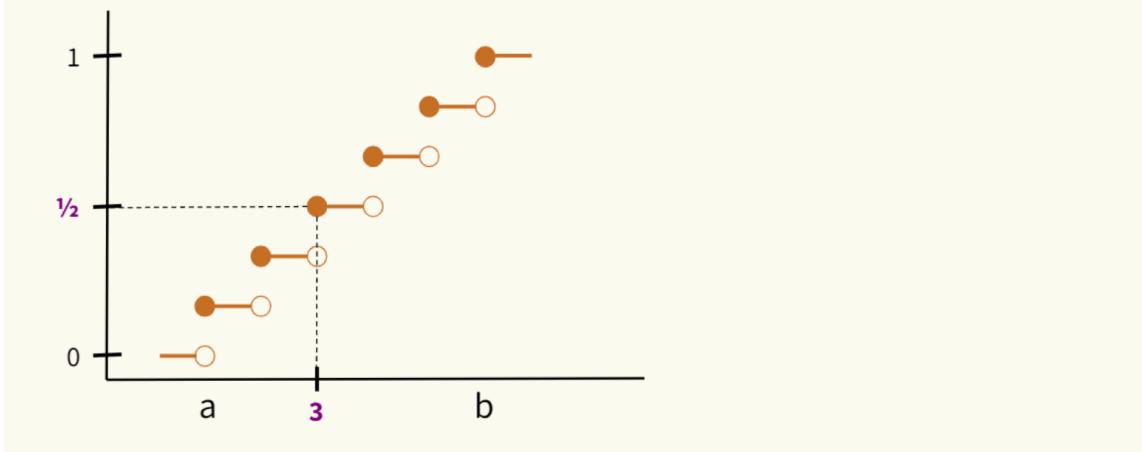
$$cdf(k; a, b) = \frac{[k] - a + 1}{b - a + 1}$$

где k — это возможный исход в границах между a и b , то есть $k \in [a, b]$.

Приведем пример. При бросании кости рассчитаем вероятность выпадения тройки или меньшего значения (то есть 1, 2 или 3).

$$cdf(3; 1, 6) = \frac{3 - 1 + 1}{6 - 1 + 1} = \frac{3}{6} = \frac{1}{2}$$

Посмотрим на эту вероятность на графике функции распределения.



Матожидание

Математическое ожидание

Кроме того, вероятностное распределение характеризуется **математическим ожиданием** (expected value). Матожидание — это среднее значение (mean value) случайной величины X , взвешенное по вероятности каждого из возможных значений.

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} x_i p_i$$

В случае бросания игральной кости расчет будет выглядеть следующим образом.

$$\mathbb{E}[X] = 1 \times \frac{1}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6} + 4 \times \frac{1}{6} + 5 \times \frac{1}{6} + 6 \times \frac{1}{6} = 3,5$$

Для равномерного распределения есть и сокращенная формула.

$$\mathbb{E}[X] = \frac{a+b}{2} = \frac{1+6}{2} = 3,5$$

На Питоне мы можем сложить результаты поэлементного умножения.

```
In [ ]: # найдем сумму элементов, умноженных на их вероятности  
np.sum(rel_freq * elements).round(2)
```

```
Out[ ]: 3.5
```

```
In [ ]: # скалярное произведение дает тот же результат  
np.dot(rel_freq, elements).round(2)
```

```
Out[ ]: 3.5
```

```
In [ ]: # посмотрим на среднее значение сгенерированного распределения  
np.mean(dice)
```

```
Out[ ]: 3.50312
```

Дисперсия

Дисперсия

Еще одной характеристикой вероятностного распределения является **дисперсия** (variance).

Для равномерного распределение она вычисляется по следующей формуле.

$$\mathbb{D}[X] = \frac{n^2 - 1}{12}$$

В нашем случае дисперсия равна

$$\mathbb{D}[X] = \frac{6^2 - 1}{12} = \frac{35}{12} \approx 2,917$$

Сравним с нашим распределением

```
In [ ]: # рассчитаем дисперсию  
((6)**2 - 1)/12
```

```
Out[ ]: 2.9166666666666665
```

```
In [ ]: # сравним с нашим распределением  
np.var(dice)
```

```
Out[ ]: 2.9156702656
```

Распределение Бернулли

Многие случайные процессы характеризуются только двумя исходами. Это и выпадение орла или решки при подбрасывании монет, и успех или неудача клинического испытания. Такие процессы можно моделировать с помощью **распределения Бернулли** (Bernulli distribution).

Давайте рассмотрим этот процесс на практике. На этот раз мы будем подбрасывать монету, но не обычную или симметричную монету (fair coin), а такую, в которой вероятность выпадения решки (обозначим ее p) равна 0,7, а вероятность орла — 0,3. Такую монету называют неправильной или несимметричной (biased, unfair coin). После каждого подбрасывания запишем получившийся результат.

В Numpy нет отдельной функции для распределения Бернулли, поэтому напишем собственную функцию.

В следующей главе я покажу, как можно использовать функцию биномиального распределения для имитации эксперимента Бернулли.

- Объявим функцию `beroulli()` с двумя параметрами, p и $iter$. Первый параметр будет отвечать за выпадение орла, второй — за количество подбрасываний.
- С помощью **функции** `np.random.rand()` мы будем генерировать значение непрерывной равномерной величины (об этом опять же чуть ниже) и если получившееся значение меньше p , мы запишем, что выпала решка (1), в противном случае, что орел (0).

Посмотрим на реализацию на Питоне.

Распределение Бернулли

```
In [ ]: # обьявим функцию bernoulli() с параметрами p, iter
def beroulli(p, iter = 1):

    # создадим пустой массив
    result = np.array([])

    # в цикле с количеством итераций iter
    for i in range(iter):

        # если значение np.random.rand() в диапазоне [0, 1) меньше или равно p:
        if np.random.rand() <= p:

            # запишем в массив result единицу
            result = np.append(result, 1)

        else:

            # в противном случае, запишем ноль
            result = np.append(result, 0)
```

```
# в результате выполнения функции вернем массив result
return result
```

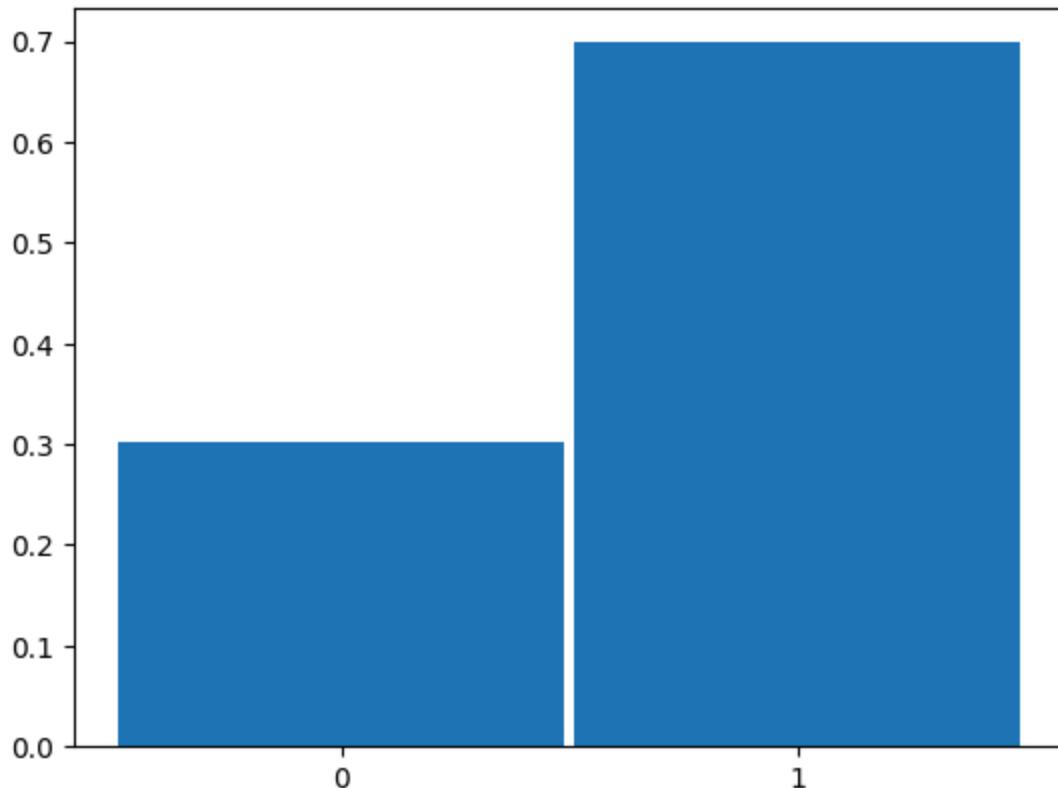
```
In [ ]: # вызовем функцию
res = bernoulli(0.7, 10000)

# выясним количество нулей и единиц в получившемся массиве
values, counts = np.unique(res, return_counts = True)

# выведем значения, частоту и относительную частоту
values, counts, counts / len(res)
```

```
Out[ ]: (array([0., 1.]), array([3015, 6985]), array([0.3015, 0.6985]))
```

```
In [ ]: # используем столбчатую диаграмму, в качестве названия столбцов передадим '0'
# в качестве высоты - относительную частоту значений
plt.bar(['0', '1'], counts / len(res), width = 0.98)
plt.show()
```



Биномиальное распределение

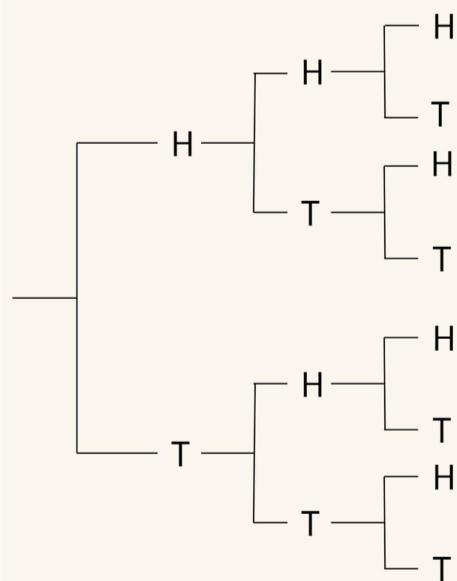
Биномиальное распределение

Пример с одинаковой вероятностью

Продолжим подбрасывать монеты, но уже не по одному разу, а по три, и каждый раз записывать результат. Всего для каждой серии испытаний (по три подбрасывания в каждой) возможны восемь исходов.

Обозначим решку через H (head), а орла через T (tail).

Броски



Вероятность каждой из комбинаций равна $1/8$, так как при одном подбрасывании вероятность выпадения орла или решки одинакова (монета симметрична).

$$P = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$$

Какова вероятность выпадения двух решек (вне зависимости от порядка, в котором они выпадали)? Вначале нам нужно посмотреть, в скольких комбинациях оказалось две решки (назовем такие испытания «успехами», successes).

Броски	Исходы	Успехи	Вероятность
H	HHH	3	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$
H	HHT	2	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$
T	HTH	2	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$
	HTT	1	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$
T	THH	2	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$
H	THT	1	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$
T	TTH	1	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$
T	TTT	0	$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = 1/8$

Таких комбинаций три (HHT, HTH, THH). Осталось количество «успехов» умножить на вероятность каждого исхода.

$$P(X = 2) = 3 \times \frac{1}{8} = \frac{3}{8}$$

Очевидно, таким образом мы можем посчитать вероятность выпадения любого количества решек. Например, у нас только один случай выпадения всех орлов, и соответственно вероятность остаться без решек составляет

$$P(X = 0) = 1 \times \frac{1}{8} = \frac{1}{8}$$

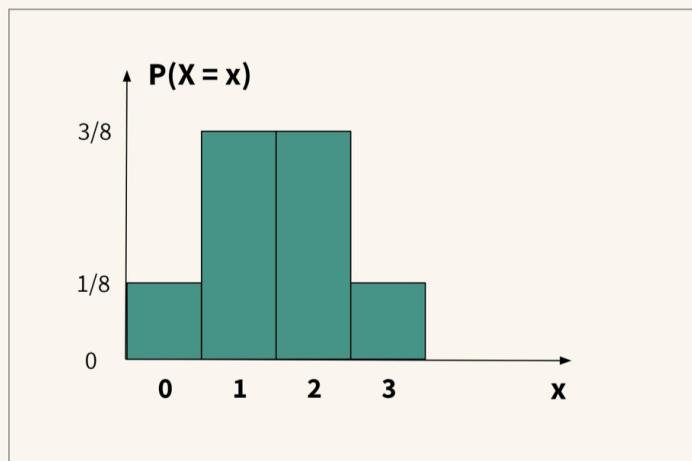
Вероятность одного и трех решек после трех испытаний равна

$$P(X = 1) = 3 \times \frac{1}{8} = \frac{3}{8}$$

$$P(X = 3) = 1 \times \frac{1}{8} = \frac{1}{8}$$

Для полноты картины замечу, что, например, $P(X = 3)$ читается как вероятность (P) того, что случайная величина (X) примет значение три.

Теперь давайте выведем эти вероятности на график. По оси x разместим возможные значения случайной величины X , а по оси y соответствующие вероятности $P(X = x)$. У нас получился график биномиального распределения.



Таким образом, **биномиальное распределение** (binomial distribution) показывает вероятность количества успехов каждой из возможных комбинаций исходов в серии одинаковых независимых испытаний Бернулли.

Пример с разной вероятностью

Давайте немного изменим условия эксперимента. На этот раз монета будет неправильной и вероятность выпадения решки составит 0,7, а орла — 0,3. Построим дерево вероятностей (tree diagram).

Испытания	Исходы	"Успехи"	Вероятность одного исхода
H	H	HHH	3
H	T	HHT	2
T	H	HTH	2
T	T	HTT	1
T	H	THH	2
T	T	THT	1
T	H	TTH	1
T	T	TTT	0

$0,7 \times 0,7 \times 0,7 = 0,343$
 $0,7 \times 0,7 \times 0,3 = 0,147$
 $0,7 \times 0,3 \times 0,7 = 0,147$
 $0,7 \times 0,3 \times 0,3 = 0,063$
 $0,3 \times 0,7 \times 0,7 = 0,147$
 $0,3 \times 0,7 \times 0,3 = 0,063$
 $0,3 \times 0,3 \times 0,7 = 0,063$
 $0,3 \times 0,3 \times 0,3 = 0,027$

Обратите внимание, несимметричность монеты повлияла на результат. Рассчитаем вероятности выпадения одного и двух решек.

$$P(X = 1) = 3 \times 0,063 = 0,189$$

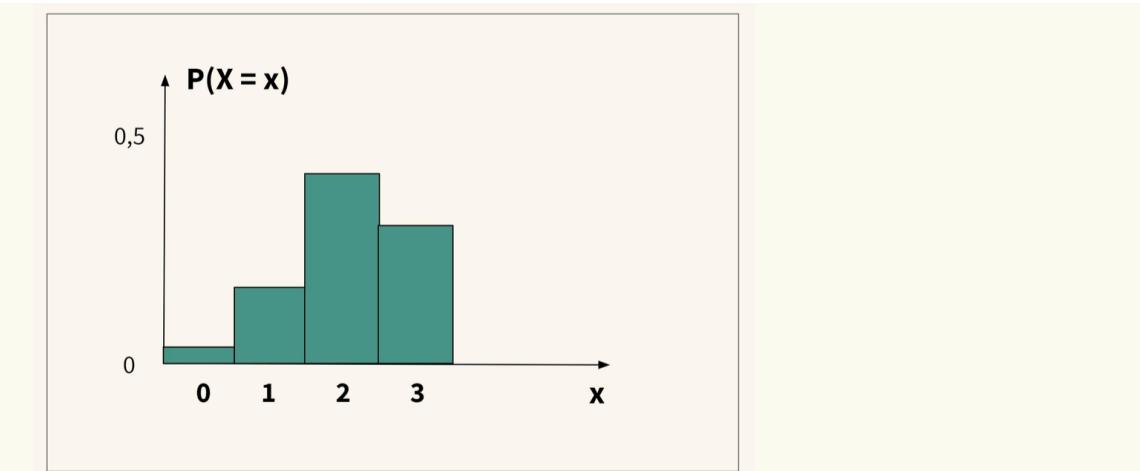
$$P(X = 2) = 3 \times 0,147 = 0,441$$

Аналогично рассчитаем вероятность того, что решка вообще не выпала и что выпали только решки.

$$P(X = 0) = 0,027$$

$$P(X = 3) = 0,343$$

Построим такое биномиальное распределение на графике.



Обратите внимание, распределение несимметрично с более пологой частью слева. Еще говорят, что распределение скошено влево (*skewed left*). Если бы вероятность выпадения решки была 0,3, а орла — 0,7, то распределение имело бы обратную форму и было скошено вправо (*skewed right*).

Формула биномиального распределения

Пока мы бросали монету по три раза, то вероятность любой комбинации исходов несложно посчитать с помощью дерева вероятностей. При этом если подбрасывать, например, 10 раз, количество вариантов достигнет $2^{10} = 1024$. Для такого распределения понадобится формула.

Возможно вы обратили внимание, что для выводения формулы нам нужно два компонента:

1. количество возможных комбинаций;
2. вероятность каждой из них.

Количество каждой из возможных комбинаций можно найти с помощью треугольника Паскаля. Например, при трех подбрасываниях ($n = 3$), мы видим, что количество возможных комбинаций будет равно {1, 3, 3, 1}, что соответствует нашему дереву вероятностей.

n = 0		1
n = 1	1	1
n = 2	1	2
n = 3	1	3
n = 4	1	4
n = 5	1	5
	3	3
	6	4
	10	10
	5	1

Можно также воспользоваться формулой биномиальных коэффициентов, в которой через n мы обозначим количество подбрасываний, а через k — количество «успехов» (выпадений орлов).

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Например, убедимся, что выпадение двух орлов ($k = 2$) при трех подбрасываниях ($n = 3$) возможно в трех комбинациях.

$$\binom{3}{2} = \frac{3!}{2!(3-2)!} = 3$$

Одновременно, нам нужно понимать **вероятность каждой комбинации**. Ее можно рассчитать по формуле

$$p^k(1-p)^{n-k}$$

В данном случае мы возводим вероятность успеха (p) в степень количества успехов (k) и вероятность неудачи ($1 - p$) в степень количества неудач ($n - k$). Например, если вероятность выпадения орла (p) равна 0,7, то вероятность выпадения двух орлов ($k = 2$) в трех бросках ($n = 3$) равна

$$0,7^2(1-0,7)^{3-2} = 0,49 \times 0,3 = 0,147$$

Остается перемножить количество комбинаций и вероятность каждой из них

$$P(X = 2) = 3 \times 0,147 = 0,441$$

Таким образом, вся формула целиком выглядит так

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Эта же формула описывает *функцию вероятности* (probability mass function, pmf) биномиального распределения.

Бином Ньютона

В целом все вероятности P случайной величины X можно представить в виде бинома Ньютона (отсюда и название распределения).

Событие 1
3 "успеха"

$$\binom{3}{0} p^{3-0} q^0 +$$

1 | HHH

$$1 \cdot (0,7 \cdot 0,7 \cdot 0,7) = \\ 1 \cdot 0,343 = 0,343$$

Событие 2
2 "успеха"

$$\binom{3}{1} p^{3-1} q^1 +$$

3 | HHT
HTH
THH

$$3 \cdot (0,7 \cdot 0,7 \cdot 0,3) = \\ 3 \cdot 0,147 = 0,441$$

Событие 3
1 "успех"

$$\binom{3}{2} p^{3-2} q^2 +$$

3 | HTT
TTH
THT

$$3 \cdot (0,7 \cdot 0,3 \cdot 0,3) = \\ 3 \cdot 0,063 = 0,189$$

Событие 4
0 "успехов"

$$\binom{3}{3} p^{3-3} q^3$$

1 | TTT

$$1 \cdot (0,3 \cdot 0,3 \cdot 0,3) = \\ 1 \cdot 0,027 = 0,027$$

Событие A

Событием A при этом будет выпадение $X = 2$ решек.

Матожидание и дисперсия

Матожидание рассчитывается по формуле

$$\mathbb{E}[X] = np$$

Например, после трех бросков мы можем ожидать, что в среднем орел выпадет $3 \times 0,7 = 2,1$ раза. Дисперсию можно рассчитать через

$$\mathbb{D}[X] = np(1 - p) \rightarrow \mathbb{D}[X] = 3 \times 0,7 \times (1 - 0,7) = 0,63$$

Биномиальное распределение на Питоне

Выполним эти же расчеты с помощью Питона. Функция `np.random.binomial()` принимает три параметра:

- `n` — количество испытаний в одном эксперименте (например, подбрасываний монеты)
- `p` — вероятность успеха (например, выпадения орла)
- `size` — количество экспериментов (серий по `n` подбрасываний)

```
In [ ]: # зададим точку отсчета
np.random.seed(42)

# и проведем 1 000 000 экспериментов (size) по три подбрасывания монеты (n)
# с вероятностью выпадения орла 0,7 (p)
res = np.random.binomial(n = 3, p = 0.7, size = 1000000)

# посмотрим на первые 10 значений
res[:10]
```

```
Out[ ]: array([2, 1, 2, 2, 3, 3, 3, 1, 2, 2])
```

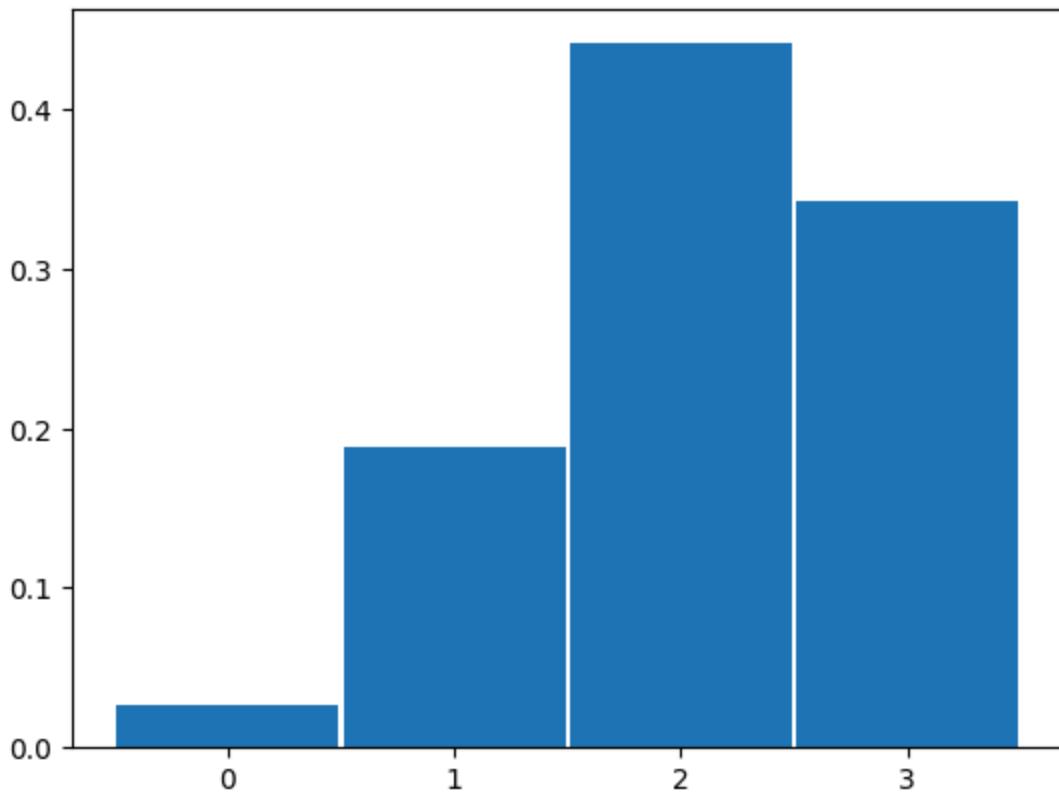
```
In [ ]: # посчитаем, сколько раз выпали 0, 1, 2 или 3 орла
_, counts = np.unique(res, return_counts = True)

# посмотрим на относительную частоту каждого из значений
counts / len(res)
```

```
Out[ ]: array([0.027142, 0.188935, 0.441131, 0.342792])
```

Как вы видите, при достаточно большом количестве экспериментов эмпирическая вероятность приблизилась к рассчитанной выше теоретической вероятности.

```
In [ ]: # построим график распределения
plt.bar(['0', '1', '2', '3'], counts / len(res), width = 0.98)
plt.show()
```



```
In [ ]: # рассчитаем среднее значение  
np.mean(res)
```

```
Out[ ]: 2.099573
```

```
In [ ]: # рассчитаем дисперсию  
np.var(res)
```

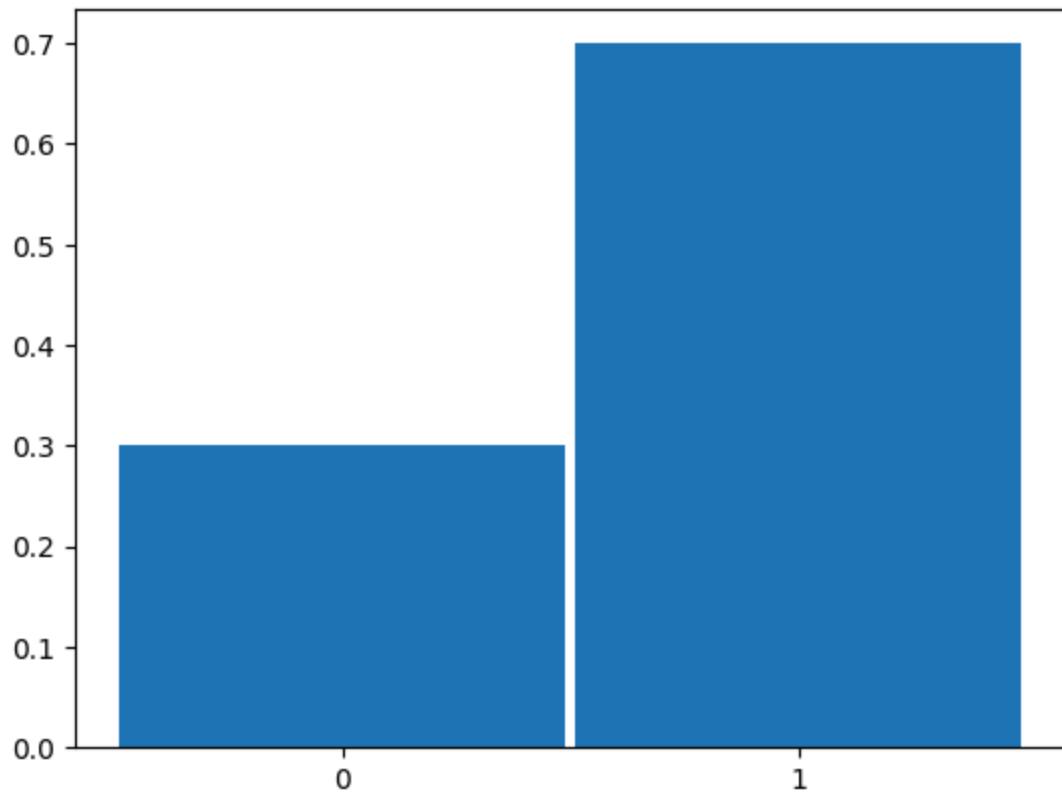
```
Out[ ]: 0.6303802176710002
```

как можно повторить эксперимент Бернулли с помощью функции `np.random.binomial()`. Для этого достаточно указать параметр `n = 1` (то есть только одно подбрасывание).

```
In [ ]: np.random.seed(42)  
  
# функция np.random.binomial() позволяет имитировать эксперимент Бернулли  
# достаточно поставить n = 1  
res = np.random.binomial(n = 1, p = 0.7, size = 1000000)  
  
# выясним количество нулей и единиц в получившемся массиве  
_, counts = np.unique(res, return_counts = True)  
  
# посмотрим на относительную частоту значений  
counts / len(res)
```

```
Out[ ]: array([0.30021, 0.69979])
```

```
In [ ]: # посмотрим на результат на графике  
plt.bar(['0', '1'], counts / len(res), width = 0.98)  
plt.show()
```



Непрерывное распределение

Непрерывное вероятностное распределение

Как уже [было сказано](#), в отличие от дискретной величины, непрерывная величина может принимать любое значение в заданном интервале.

Непрерывное равномерное распределение

Непрерывное равномерное распределение (continuous uniform distribution) описывает случайную величину, вероятность значений которой одинакова на заданном интервале от a до b .

$$X \sim U(a, b)$$

Например, если мы знаем, что автобус приходит на остановку каждые 12 минут, то время ожидания автобуса на остановке равномерно распределено между 0 и 12 минутами.

$$X \sim U(0, 12)$$

Плотность вероятности

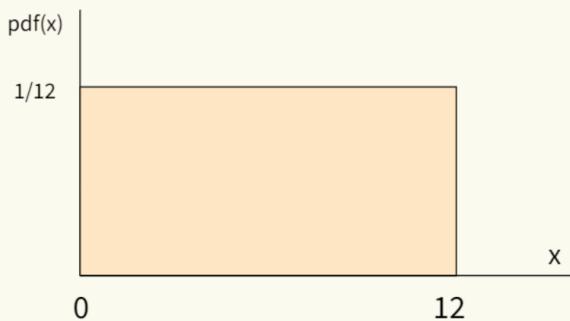
Непрерывное распределение (в отличие от дискретного) задается **плотностью вероятности** (probability density function, pdf). Для равномерного непрерывного распределения плотность вероятности определяется вот такой несложной функцией.

$$pdf(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b] \\ 0, & x \notin [a, b] \end{cases}$$

В примере с ожиданием автобуса вероятность его приезда в любой момент в пределах заданного интервала равна

$$pdf(x) = \begin{cases} \frac{1}{12-0} = \frac{1}{12}, & x \in [0, 12] \\ 0, & x \notin [0, 12] \end{cases}$$

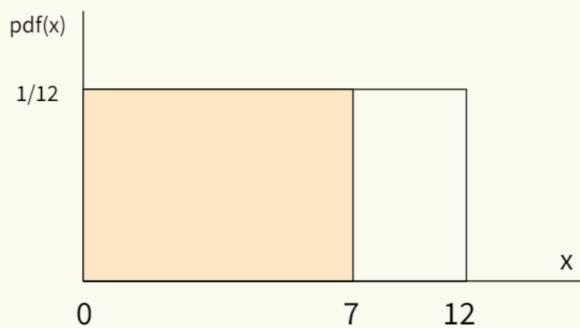
На графике равномерное распределение представляет собой прямоугольник, площадь которого всегда равна единице.



Если мы хотим посчитать вероятность приезда автобуса в пределах заданного интервала ожидания, нам, по сути, нужно вычислить отдельный участок площади прямоугольника.

Например, вероятность приезда автобуса при ожидании до 12 минут включительно составляет 1.00 или 100%, потому что такой промежуток включает всю площадь прямоугольника.

Теперь давайте рассчитаем вероятность ожидания автобуса до 7 минут включительно. Нас будет интересовать интервал от 0 до 7 минут и соответствующий участок площади прямоугольника.



Применив несложную формулу, мы без труда вычислим площадь этого участка.

$$P(7) = \frac{1}{12} \times 7 \approx 0,583$$

Матожидание и дисперсия

Остается рассчитать матожидание (среднее время ожидания автобуса) и дисперсию.

$$\mathbb{E}[X] = \frac{a + b}{2} = \frac{0 + 12}{2} = 6$$

$$\mathbb{D}[X] = \frac{(b - a)^2}{12} = \frac{(12 - 0)^2}{12} = 12$$

Реализация на Питоне

Воспользуемся функцией `np.random.uniform()` для того, чтобы создать равномерное распределение с параметрами $U(0, 12)$.

Равномерное распределение

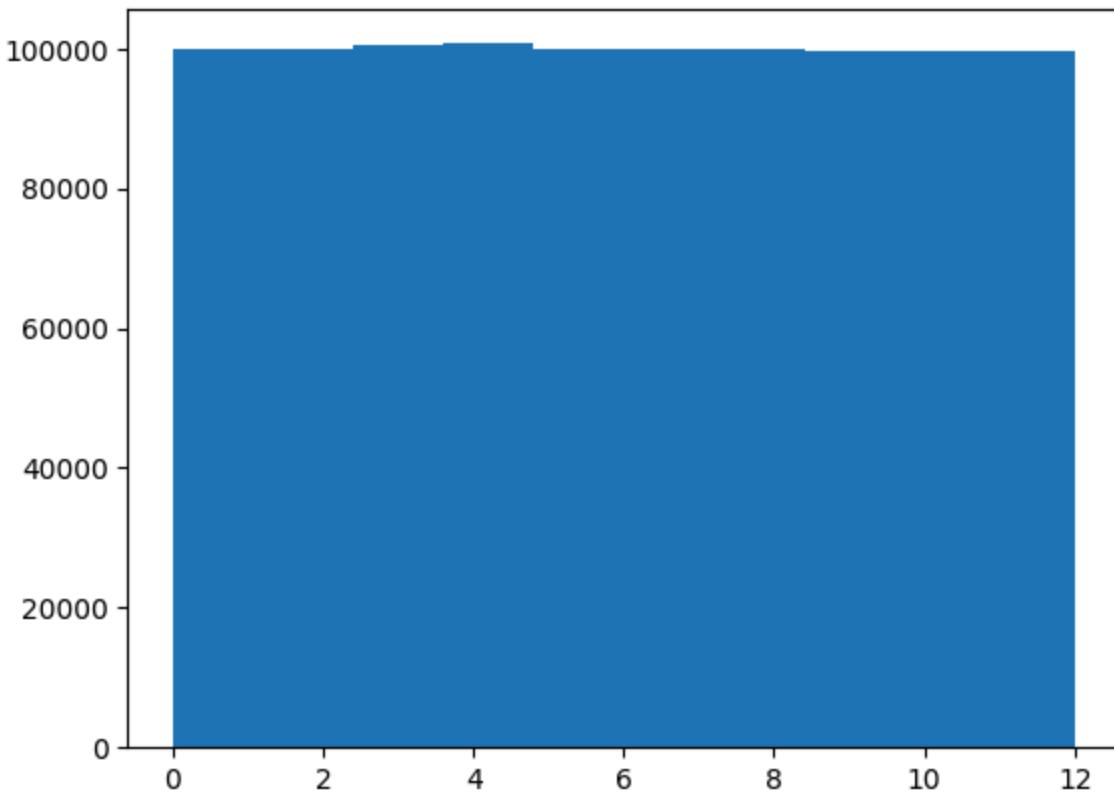
Ожидание автобуса, приходящего каждые 12 минут

```
In [ ]: # создадим распределение с параметрами a = 0 и b = 12
# повторим эксперимент 1 000 000 раз
res = np.random.uniform(0, 12, 1000000)

# посмотрим на первые десять значений
res[:10]
```

```
Out[ ]: array([ 7.14186749,  4.37660571,  0.06450744,  6.73305271, 10.75884493,
       6.38060282,  9.36585218,  1.94345171,  1.64757201, 10.72732251])
```

```
In [ ]: # выведем результат с помощью гистограммы
plt.hist(res)
plt.show()
```



```
In [ ]: # посмотрим на среднее значение  
np.mean(res)
```

```
Out[ ]: 5.993743863584736
```

```
In [ ]: # и дисперсию  
np.var(res)
```

```
Out[ ]: 11.986689948141048
```

```
In [ ]: # разделим количество значений <= 7 на общее количество значений  
len(res[res <= 7])/len(res)
```

```
Out[ ]: 0.584527
```

Разница между `np.random.random()`, `np.random.rand()` и `np.random.uniform()`

Разница между `np.random.random()`, `np.random.rand()` и `np.random.uniform()`

Как вы заметили, мы использовали три функции для генерирования равномерного распределения.

Функция `np.random.random(size = None)` создает равномерное распределение в полуоткрытом интервале $[0, 1]$. Параметр `size` задает размер этого распределения (количество экспериментов).

```
In [ ]: # создадим массив 2 x 3 в интервале от [0, 1)
np.random.seed(42)
```

```
# для этого передадим параметры в виде кортежа
np.random.random((2, 3))
```

```
Out[ ]: array([[0.37454012, 0.95071431, 0.73199394],
               [0.59865848, 0.15601864, 0.15599452]])
```

```
In [ ]: # функция np.random.rand() практически ничем не отличается от np.random.random()
# единственное отличие - размеры массива передаются отдельными параметрами,
np.random.seed(42)
np.random.rand(2, 3)
```

```
Out[ ]: array([[0.37454012, 0.95071431, 0.73199394],
               [0.59865848, 0.15601864, 0.15599452]])
```

Для функции `np.random.uniform(low = 0.0, high = 1.0, size = None)` интервал $[0, 1]$ является интервалом по умолчанию, при этом можно задать любой другой промежуток (как мы и сделали выше).

Приведем несколько примеров.

```
In [ ]: # сгенерируем одно значение из равномерного распределения в интервале [0, 9)
np.random.uniform(9)
```

```
Out[ ]: 8.535331102654403
```

```
In [ ]: # создадим двумерный массив 2 x 5 со значениями в промежутке [0, 1)
np.random.seed(42)
np.random.uniform(size = (2, 5))
```

```
Out[ ]: array([[0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864],
               [0.15599452, 0.05808361, 0.86617615, 0.60111501, 0.70807258]])
```

Напоследок замечу, что равномерное непрерывное распределение является частным случаем бета-распределения с параметрами $Beta(1, 1)$.

Нормальное распределение

Нормальное распределение

На занятии по описательной статистике мы начали изучать количественные данные с примера [роста мужчин в России](#). При этом интересно, что рост людей, как и многие другие величины (например, вес человека, артериальное давление, некоторые природные явления и так далее) имеют так называемое **нормальное распределение** (normal distribution).

Функция плотности нормального распределения

Функция плотности (pdf) нормального распределения случайной величины X определяется функцией Гаусса и поэтому нормальное распределение также называется **распределением Гаусса** (Gauss distribution).

$$pdf(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Как видно из формулы, единственными неизвестными параметрами являются μ («мю», матожидание) и σ («сигма», среднее квадратическое отклонение, СКО). Именно они и определяют нормальное распределение.

$$X \sim \mathcal{N}(\mu, \sigma)$$

Здесь важно напомнить, что среднее квадратическое отклонение (standard deviation) равно квадратному корню из дисперсии (variance).

$$\sigma = \sqrt{\sigma^2}$$

Функция np.random.normal()

В Питоне нормальное распределение создается с помощью **функции np.random.normal()**. Мы уже использовали ее, в частности, для создания данных о росте мужчин и женщин в блокноте [к десятому занятию вводного курса](#). Повторим этот код, увеличив размер массива до 100 000.

Визуализация

```
In [ ]: # зададим точку отсчета
np.random.seed(42)

# создадим 100 000 значений нормально распределенной величины с матожиданием
height_men = np.round(np.random.normal(180, 10, 100000))

# создадим еще одно распределение, но матожидание будет равно 160 см
height_women = np.round(np.random.normal(160, 10, 100000))
```

```
In [ ]: # посмотрим на первые десять значений  
print(height_men[:10])  
print(height_women[:10])
```

```
[185. 179. 186. 195. 178. 178. 196. 188. 175. 185.]  
[170. 148. 166. 154. 157. 160. 159. 143. 174. 171.]
```

Гистограмма

```
In [ ]: # выведем результат с помощью гистограммы  
plt.figure(figsize = (10,6))  
  
# зададим 18 интервалов (bin) и уровень прозрачности графиков  
plt.hist(height_men, 18, alpha = 0.5, label = 'Рост мужчин')  
plt.hist(height_women, 18, alpha = 0.5, label = 'Рост женщин')  
  
# пропишем расположение и размер шрифта легенды  
plt.legend(loc = 'upper left', prop = {'size': 14})  
  
# добавим подписи  
plt.xlabel('Рост, см', fontsize = 16)  
plt.ylabel('Количество людей', fontsize = 16)  
plt.title('Распределение роста мужчин и женщин в России', fontsize = 18)  
  
plt.show()
```

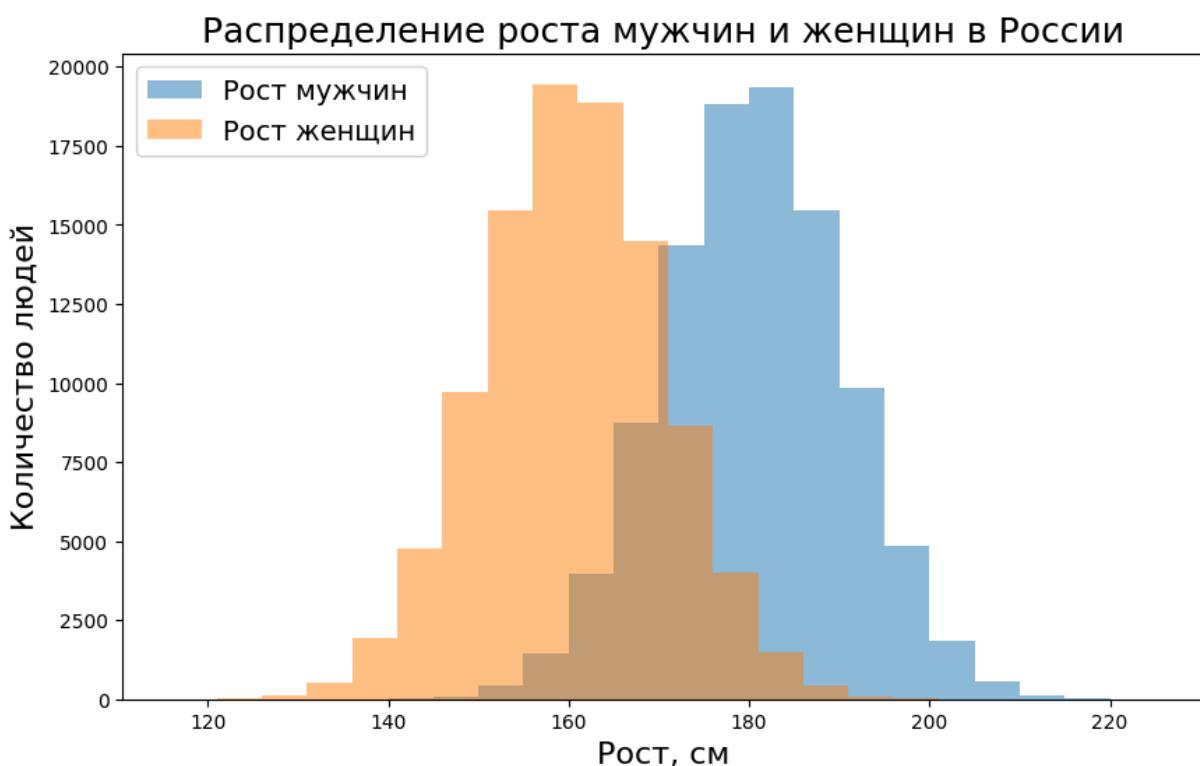


График функции плотности

```
In [ ]: # импортируем библиотеку seaborn  
import seaborn as sns
```

```

# зададим размер графиков
plt.figure(figsize = (10,6))

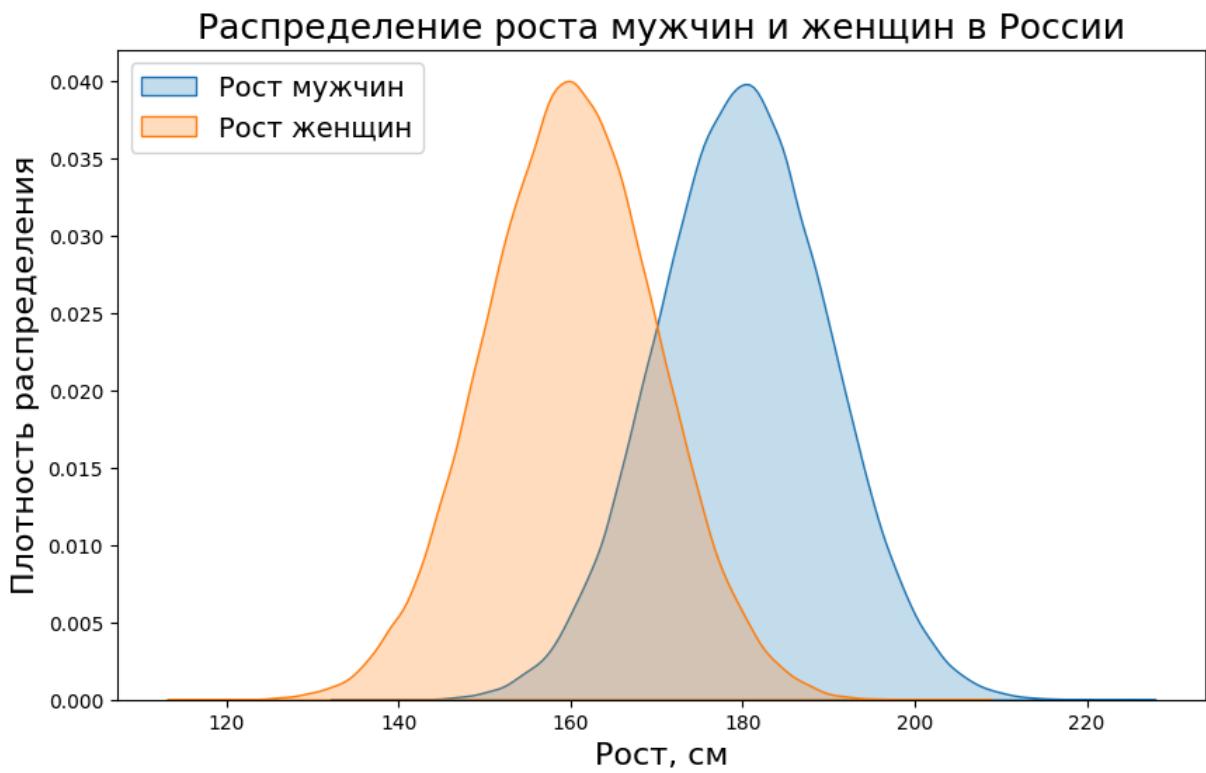
# и построим график функции плотности
sns.kdeplot(height_men, fill = True, label = 'Рост мужчин')
sns.kdeplot(height_women, fill = True, label = 'Рост женщин')

# пропишем расположение и размер шрифта легенды
plt.legend(loc = 'upper left', prop = {'size': 14})

# добавим подписи
plt.xlabel('Рост, см', fontsize = 16)
plt.ylabel('Плотность распределения', fontsize = 16)
plt.title('Распределение роста мужчин и женщин в России', fontsize = 18)

plt.show()

```



Boxplot

```

In [ ]: # boxplot, как и многие другие графики, удобно строить из библиотеки Pandas
import pandas as pd

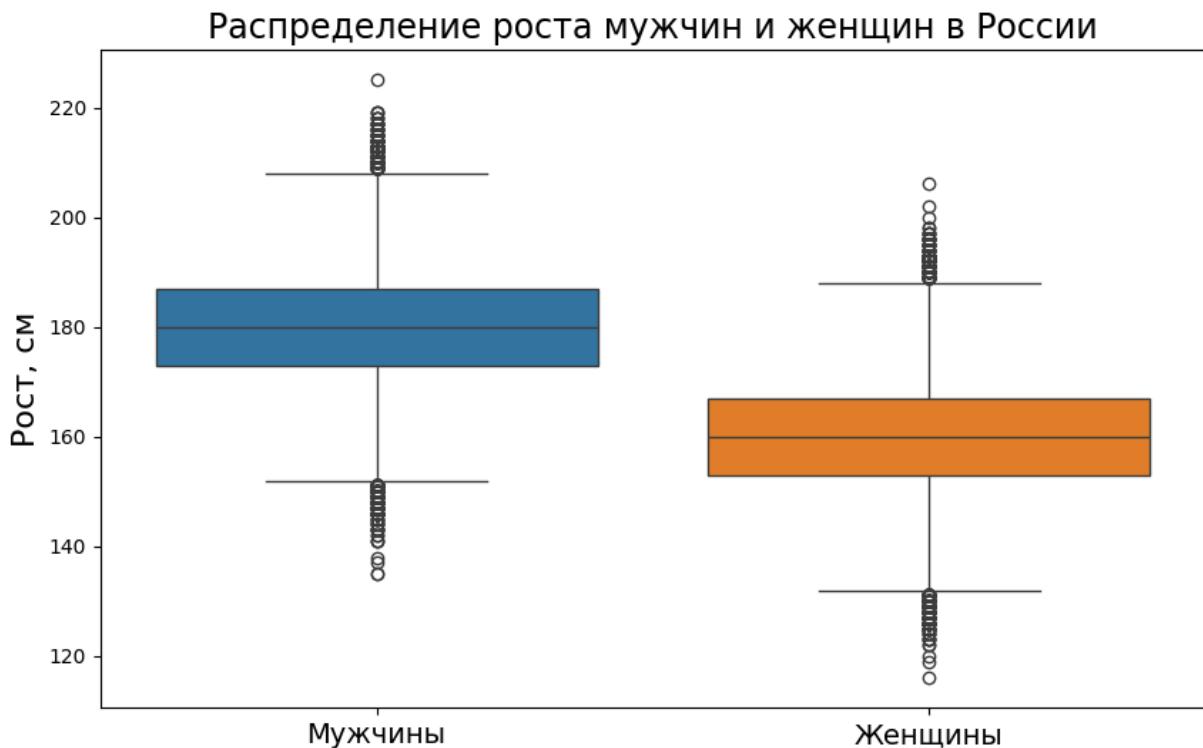
# создадим датафрейм из словаря, включающего два массива с данными о росте
data = pd.DataFrame({'Мужчины' : height_men, 'Женщины' : height_women})
data.head()

```

Out[]: Мужчины Женщины

	Мужчины	Женщины
0	185.0	170.0
1	179.0	148.0
2	186.0	166.0
3	195.0	154.0
4	178.0	157.0

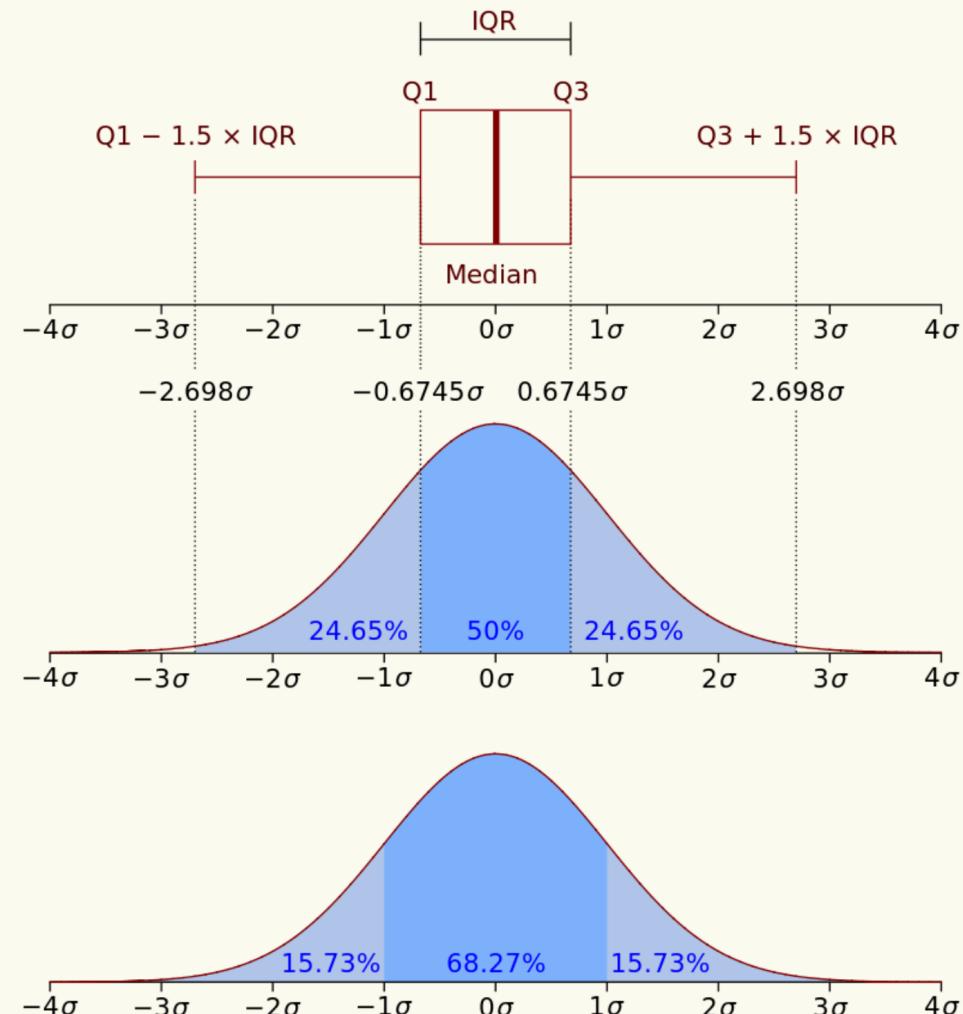
```
In [ ]: # зададим размер графика  
plt.figure(figsize = (10,6))  
  
# и построим два вертикальных boxplots, передав датафрейм с данными в параметр  
sns.boxplot(data = data)  
  
# дополнительно укажем размер подписей каждого из графиков по оси x  
plt.xticks(fontsize = 14)  
# подпись к оси y  
plt.ylabel('Рост, см', fontsize = 16)  
# и заголовок графика  
plt.title('Распределение роста мужчин и женщин в России', fontsize = 17)  
  
plt.show()
```



Boxplot позволяет увидеть медиану (median), первый и третий квартили (Quartile 1, Q1 и Quartile 3, Q3), межквартильный размах (Interquartile Range, IQR), а также, что очень важно, так называемые выбросы (outliers), то есть значения, сильно отличающиеся от среднего (на графике выше они

обозначены точками). Ни гистограмма, ни график плотности распределения этой информации не выводят.

На рисунке ниже можно увидеть связь между boxplot и графиком плотности.



Источник: [Википедия](#)

С σ («сигма»), обозначающей среднее квадратическое отклонение, мы уже знакомы, а вот что такое квартиль, межквартильный размах и о том, что измеряют $Q1 - 1.5 \times IQR$ и $Q3 + 1.5 \times IQR$ мы поговорим на следующем курсе по [анализу и обработке данных](#).

Дополнительно приведу пример того, как можно совместить boxplot с гистограммой на

Boxplot и гистограмма

```
In [ ]: # создадим два подграфика
f, (ax_box, ax_hist) = plt.subplots(nrows = 2, # из двух строк
                                      ncols = 1, # и одного столбца
                                      sharex = True, # оставим только нижние г
                                      gridspec_kw = {'height_ratios': (.15, .8
figsize = (12,8)) # зададим размер графи
```

```

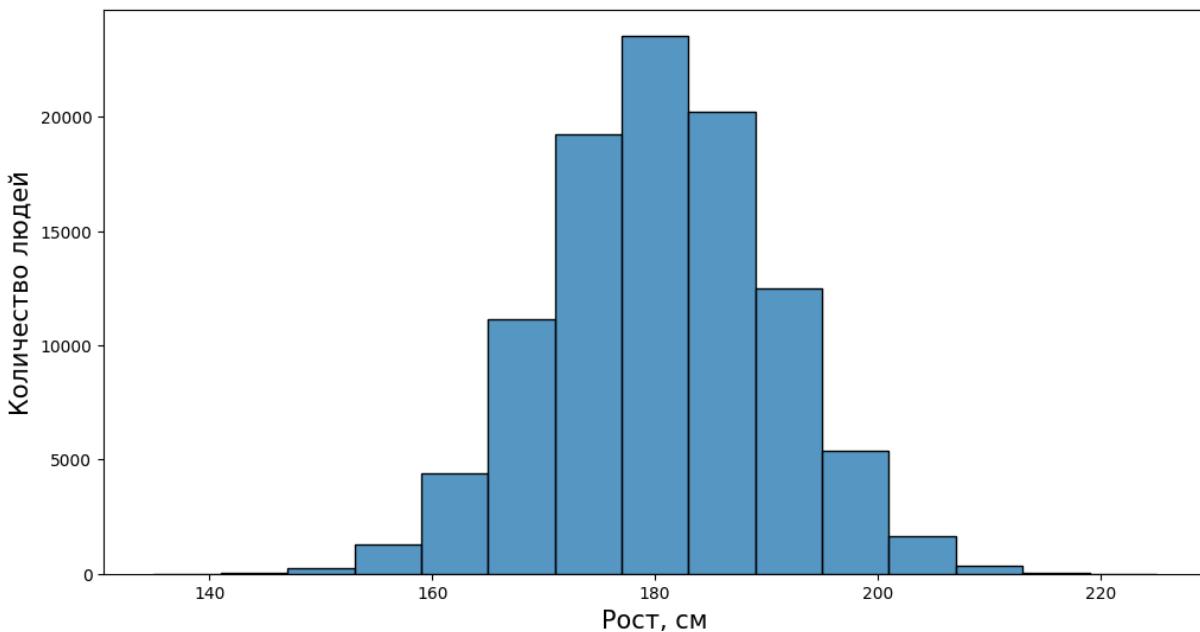
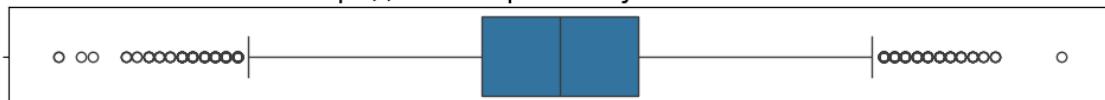
# в первом подграфике построим boxplot
sns.boxplot(x = height_men, ax = ax_box)
# во втором гистограмму
sns.histplot(data = height_men, bins = 15, ax = ax_hist)

# зададим заголовок и подписи к осям
ax_box.set_title('Распределение роста мужчин в России', fontsize = 17)
ax_hist.set_xlabel('Рост, см', fontsize = 15)
ax_hist.set_ylabel('Количество людей', fontsize = 15)

plt.show()

```

Распределение роста мужчин в России



Расчет вероятности

Теперь давайте рассчитаем вероятность того, что рост случайно встретившегося нам человека на улице составляет менее 190 см.

Разумеется, для того, чтобы это утверждать мы должны допустить, что наши данные действительно отражают генеральную совокупность, то есть рост всех людей.

Вначале рассчитаем теоретическую вероятность. Для создания «идеального» теоретического распределения воспользуемся библиотекой `scipy`.

```
In [ ]: # импортируем объект norm из модуля stats библиотеки scipy
from scipy.stats import norm
```

```

# зададим размер графика
plt.figure(figsize = (10,6))

# пропишем среднее значение и СКО
mean, std = 180, 10

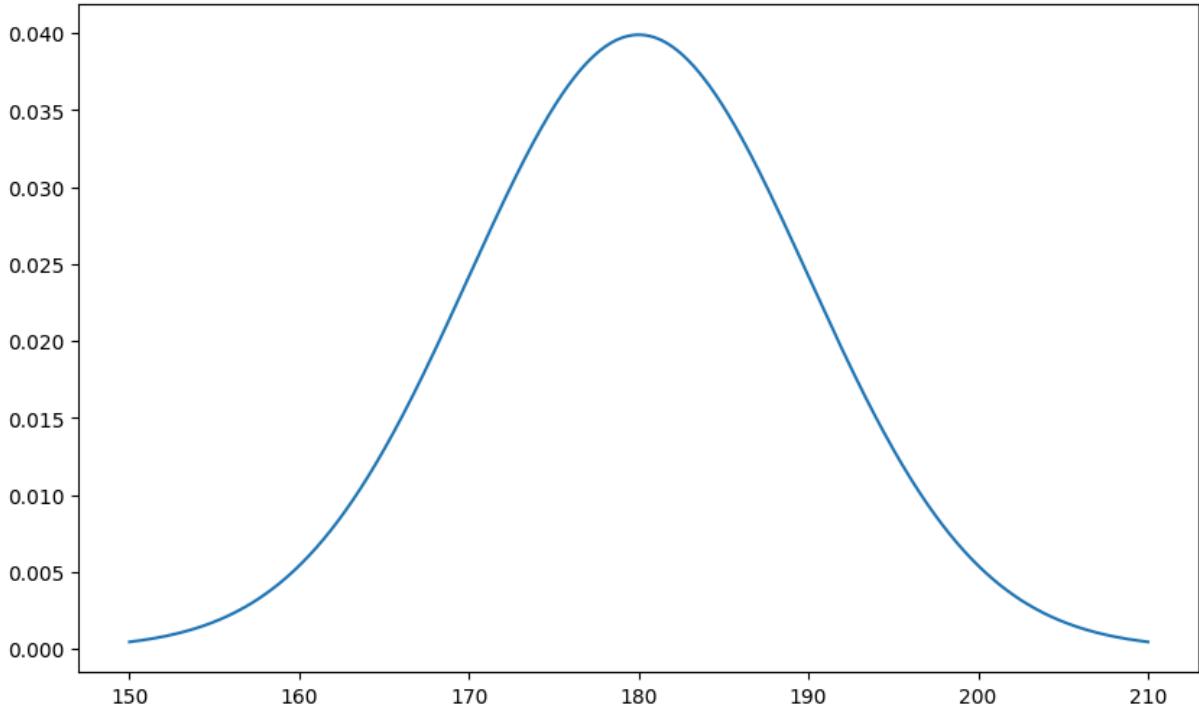
# создадим пространство из 1000 точек в диапазоне +/- трех СКО от среднего
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)

# рассчитаем значения по оси у с помощью метода .pdf()
# т.е. функции плотности распределения
f = norm.pdf(x, mean, std)

# и построим график
plt.plot(x, f)

plt.show()

```



Как и в случае с равномерным распределением задача сводится к нахождению площади под кривой от минус бесконечности до 190 см включительно.

```

In [ ]: # зададим размер графика
plt.figure(figsize = (10,6))

# пропишем среднее значение и СКО
mean, std = 180, 10

# создадим пространство из 1000 точек в диапазоне +/- трех СКО от среднего
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)

# рассчитаем значения по оси у с помощью метода .pdf()

```

```

# т.е. функции плотности распределения
f = norm.pdf(x, mean, std)

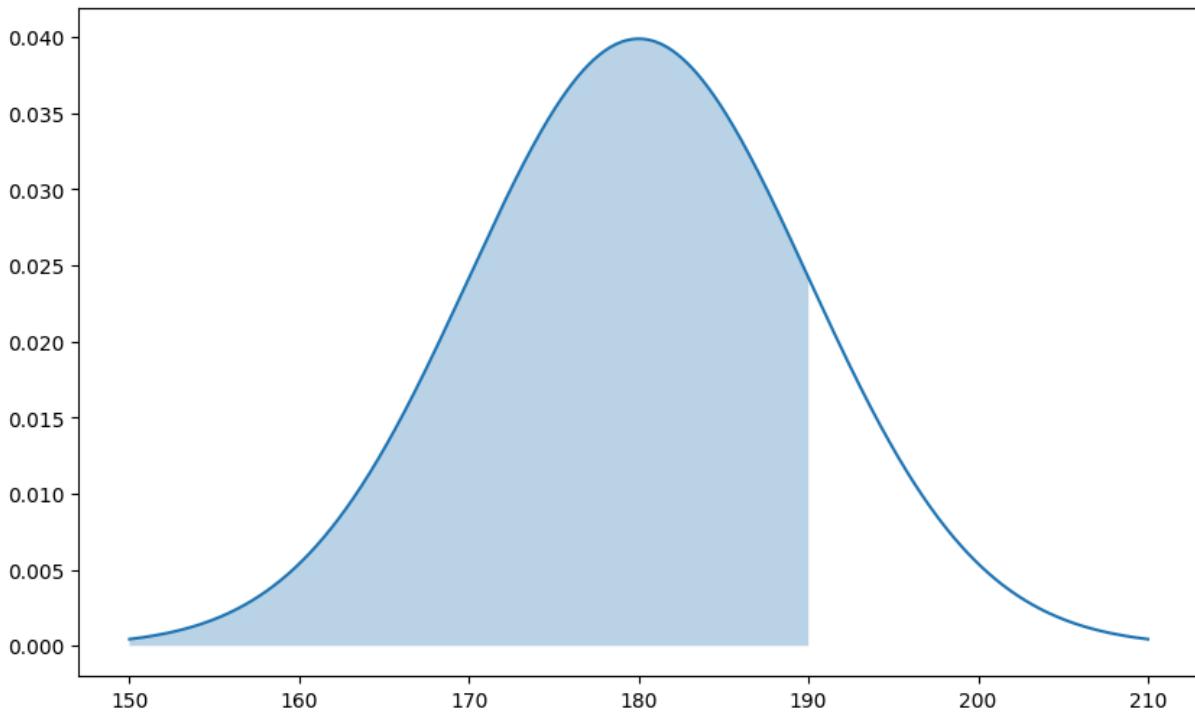
# и построим график
plt.plot(x, f)

# дополнительно создадим точки на оси x для закрашенной области
px = np.linspace(mean - 3 * std, 190, 1000)

# и заполним в пределах этих точек по оси x пространство
# от кривой нормального распределения до оси y = 0
plt.fill_between(px, norm.pdf(px, mean, std), alpha = 0.3)

# выведем оба графика
plt.show()

```



```

In [ ]: # передадим методу .cdf() границу (рост), среднее значение (loc) и СКО (scale)
area = norm.cdf(190, loc = 180, scale = 10)

# на выходе мы получим площадь под кривой
area

```

Out[]: 0.8413447460685429

```

In [ ]: # с помощью метода .ppf() можно узнать значение (рост) по площади
height = norm.ppf(area, loc = 180, scale = 10)
height

```

Out[]: 190.0

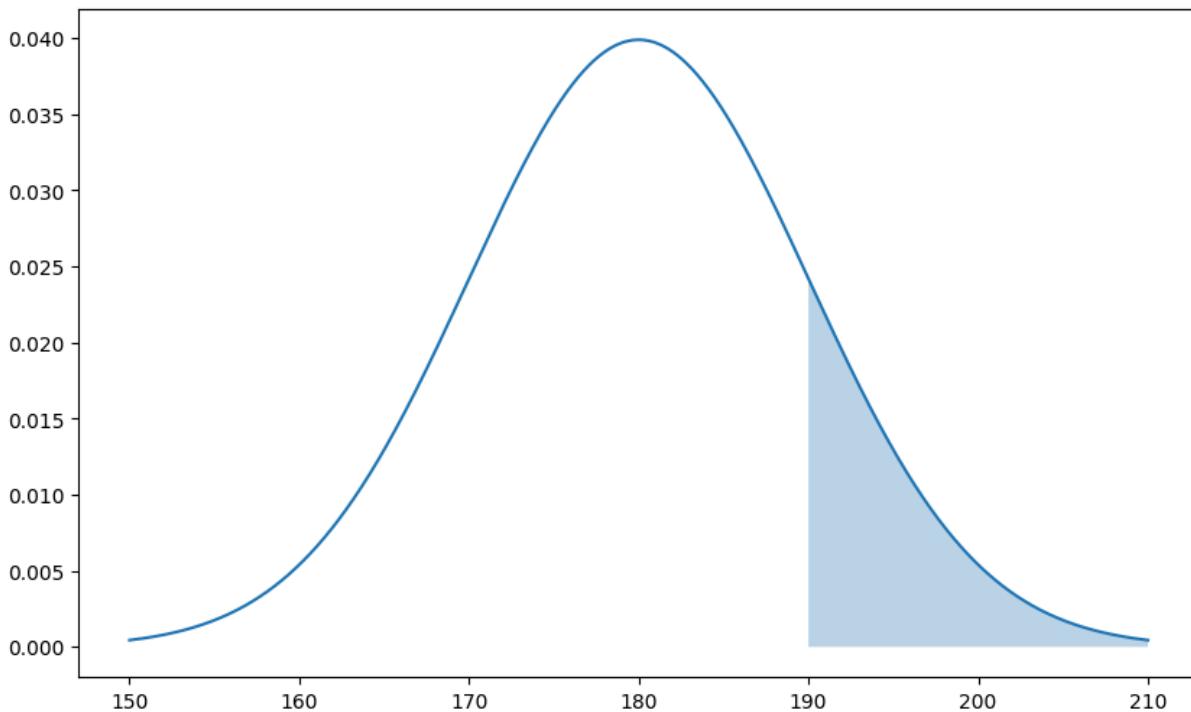
```

In [ ]: # теперь рассчитаем вероятность встретить человека выше 190 см
1 - norm.cdf(190, loc = 180, scale = 10)

```

```
Out[ ]: 0.15865525393145707
```

```
In [ ]: # зададим размер графика  
plt.figure(figsize = (10,6))  
  
# пропишем среднее значение и СКО  
mean, std = 180, 10  
  
# создадим пространство из 1000 точек в диапазоне +/- трех СКО от среднего з  
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)  
  
# рассчитаем значения по оси у с помощью метода .pdf()  
# т.е. функции плотности распределения  
f = norm.pdf(x, mean, std)  
  
# и построим график  
plt.plot(x, f)  
  
# дополнительно создадим точки на оси x для закрашенной области  
px = np.linspace(190, mean + 3 * std, 1000)  
  
# и заполним в пределах этих точек по оси x пространство  
# от кривой нормального распределения до оси у = 0  
plt.fill_between(px, norm.pdf(px, mean, std), alpha = 0.3)  
  
# выведем оба графика  
plt.show()
```

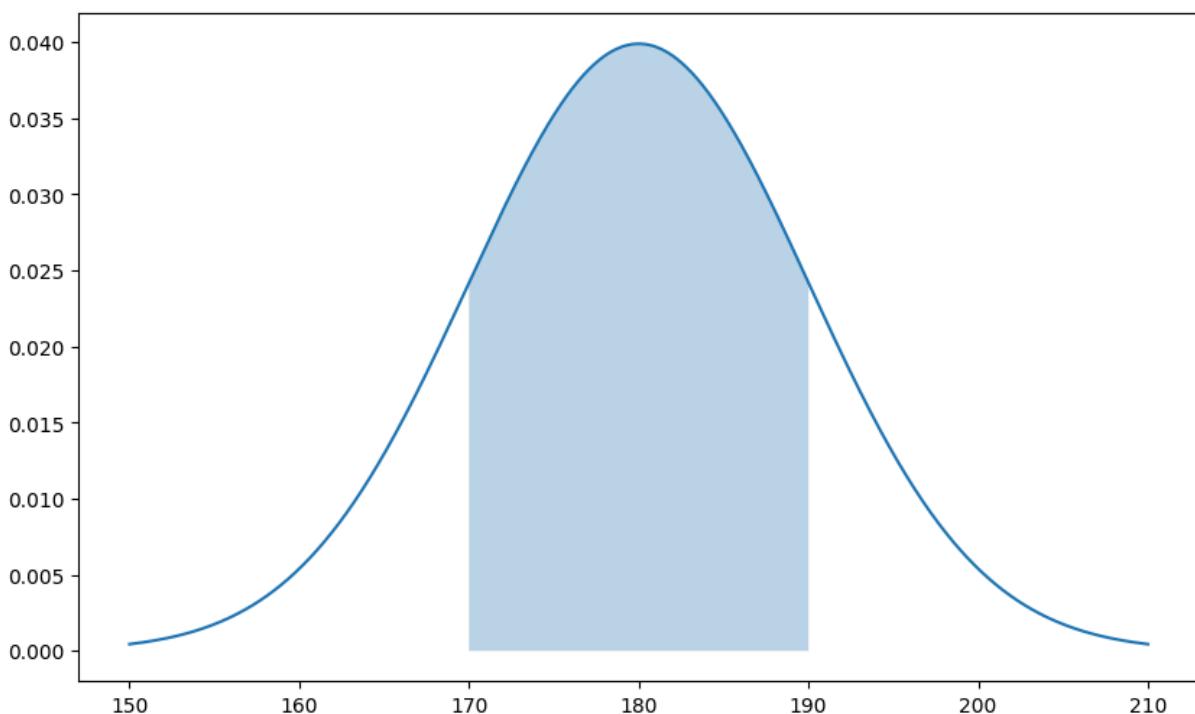


```
In [ ]: # рассчитаем меньшую площадь до нижней границы  
lowerbound = norm.cdf(170, loc = 180, scale = 10)  
  
# рассчитаем большую площадь до верхней границы  
upperbound = norm.cdf(190, loc = 180, scale = 10)
```

```
# вычтем меньшую площадь из большей  
upperbound - lowerbound
```

Out[]: 0.6826894921370859

```
In [ ]: # зададим размер графика  
plt.figure(figsize = (10,6))  
  
# пропишем среднее значение и СКО  
mean, std = 180, 10  
  
# создадим пространство из 1000 точек в диапазоне +/- трех СКО от среднего значения  
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)  
  
# рассчитаем значения по оси у с помощью метода .pdf()  
# т.е. функции плотности распределения  
f = norm.pdf(x, mean, std)  
  
# и построим график  
plt.plot(x, f)  
  
# дополнительно создадим точки на оси x для закрашенной области  
px = np.linspace(170, 190, 1000)  
  
# и заполним в пределах этих точек по оси x пространство  
# от кривой нормального распределения до оси у = 0  
plt.fill_between(px, norm.pdf(px, mean, std), alpha = 0.3)  
  
# выведем оба графика  
plt.show()
```



```
In [ ]: # разделим количество людей с ростом <= 190 на общее количество наблюдений  
len(height_men[height_men <= 190])/len(height_men)
```

```
Out[ ]: 0.85195
```

pdf и cdf

Функция плотности и функция распределения

Рассмотрим связь функции плотности, обозначим ее $P(x)$, и функции распределения $D(x)$.

Напомню, что функция плотности нормального распределения определяется по формуле

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

При этом вот что мы успели про нее узнать:

1. вероятность того, что случайная величина примет значение не более заданного в интервале $(-\infty; x]$ равна площади под кривой функции плотности на этом промежутке;
2. эта площадь вычисляется с помощью функции распределения.

Одновременно, известно, что площадь под кривой определяется как интеграл функции этой кривой на заданном промежутке.

Значит **функция распределения** $D(x)$ есть **интеграл функции плотности** $P(x)$.

Математически это выражается так.

$$D(x) = \int_{-\infty}^x P(x)dx$$

Тема интегрирования выходит за рамки сегодняшнего занятия, однако давайте попробуем на уровне интуиции понять, как связаны функция плотности и функция распределения через нахождение интеграла.

ИНСТРУКЦИЯ ПО ИНТЕГРАЛАМ <https://www.uznaychtotakoe.ru/integral/>

Видео https://www.youtube.com/watch?v=anxCPxwBrRc&ab_channel=SHIZ

Интегрирование

$$D(x) = \int_{-\infty}^x P(x)dx$$

```
In [ ]: # зададим размер графика  
plt.figure(figsize = (14,10))
```

```
# определим среднее значение и СКО
```

```

mean, std = 180, 10

# зададим последовательность точек на оси x
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)

# найдем значения функции плотности
y1 = norm.pdf(x, mean, std)

# и функции распределения
y2 = norm.cdf(x, mean, std)

# на левом графике (row 1, col 2, index 1)
plt.subplot(1, 2, 1)

# построим функцию плотности
plt.plot(x, y1)

# и заполним пространство под кривой вплоть до точки x = 180
px = np.linspace(mean - 3 * std, 180, 1000)
plt.fill_between(px, norm.pdf(px, mean, std), alpha = 0.3)

# добавим заголовок и подписи к осям
plt.title('pdf', fontsize = 16)
plt.xlabel('x', fontsize = 15)
plt.ylabel('P(x)', fontsize = 15)

# на правом графике (row 1, col 2, index 2)
plt.subplot(1, 2, 2)

# построим функцию распределения
plt.plot(x, y2)

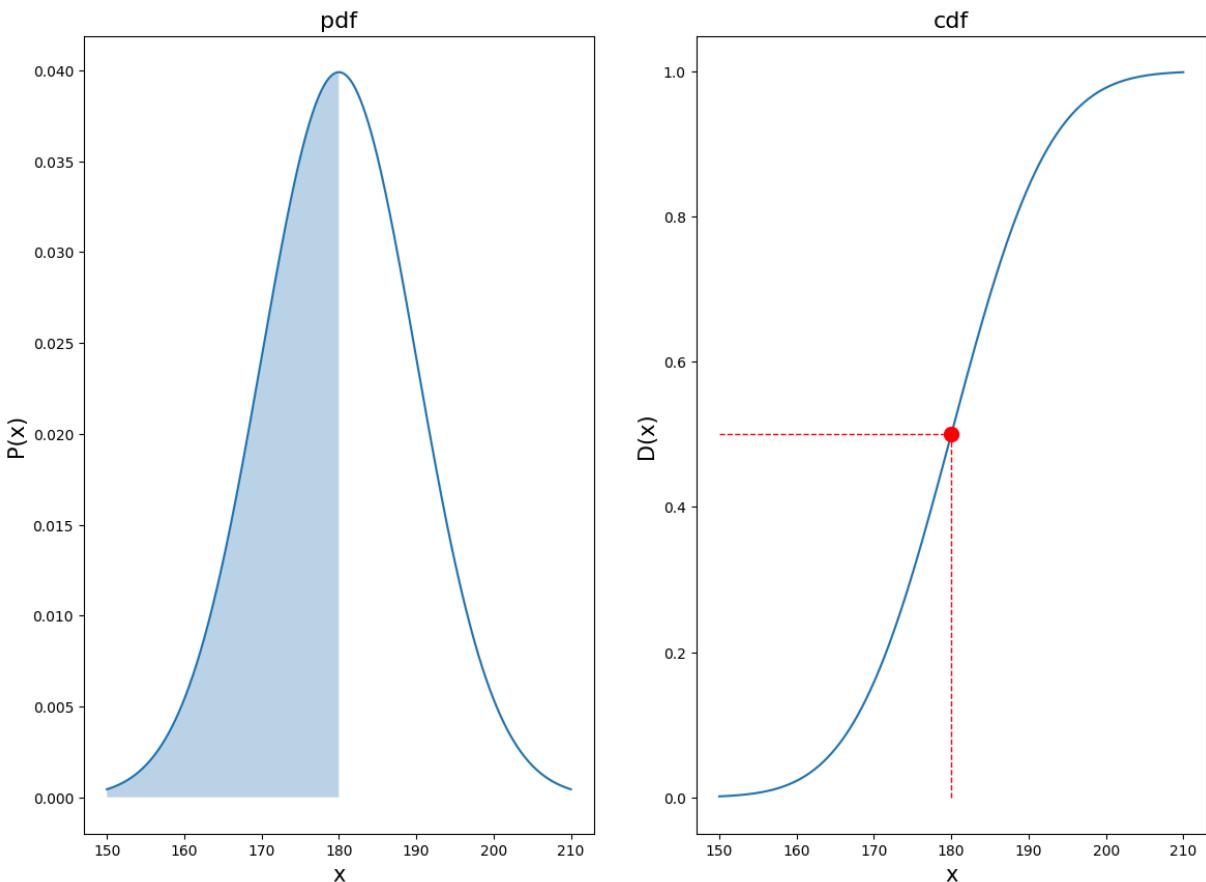
# а также горизонтальную и вертикальную пунктирные линии,
plt.hlines(y = 0.5, xmin = 150, xmax = 180, linewidth = 1, color = 'r', linestyle = 'dashed')
plt.vlines(x = 180, ymin = 0, ymax = 0.5, linewidth = 1, color = 'r', linestyle = 'dashed')

# которые сойдутся в точке (180, 0.5)
plt.plot(180, 0.5, marker = 'o', markersize = 10, markeredgecolor = 'r', markerfacecolor = 'white')

# добавим заголовок и подписи к осям
plt.title('cdf', fontsize = 16)
plt.xlabel('x', fontsize = 15)
plt.ylabel('D(x)', fontsize = 15)

plt.show()

```



Дифференцирование

$$P(x) = D'(x)$$

Вначале обратимся к графику слева. Как мы видим, вероятность встретить человека не более 180 см составляет 0,5 (закрашенный синим участок). Одновременно, проинтегрировав функцию плотности на интервале $(-\infty; 180]$, на графике справа мы видим, что «накопленная» вероятность составляет 0,5, и именно эту вероятность нам показывает график функции распределения на отметке $x = 180$.

Продолжим исследовать связь функции плотности и функции распределения.

Если функция распределения есть интеграл функции плотности, то **плотность вероятности** $P(x)$ является **производной функции распределения** $D(x)$.

Приведем формулу.

$$P(x) = D'(x)$$

```
In [ ]: # зададим функции pdf и cdf на оси x
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)
```

при этом возникает проблема: по оси у них разный масштаб

```

y1 = norm.pdf(x, mean, std)
y2 = norm.cdf(x, mean, std)

# эту проблему можно решить через функции subplots() и twinx()
# создадим сетку из одной ячейки
fig, ax_left = plt.subplots(nrows = 1, ncols = 1, figsize = (12,8))

# создадим новую ось с правой стороны
ax_right = ax_left.twinx()

# на оси x и левой оси y построим график функции плотности (pdf)
ax_left.plot(x, y1, label = 'P(x)')

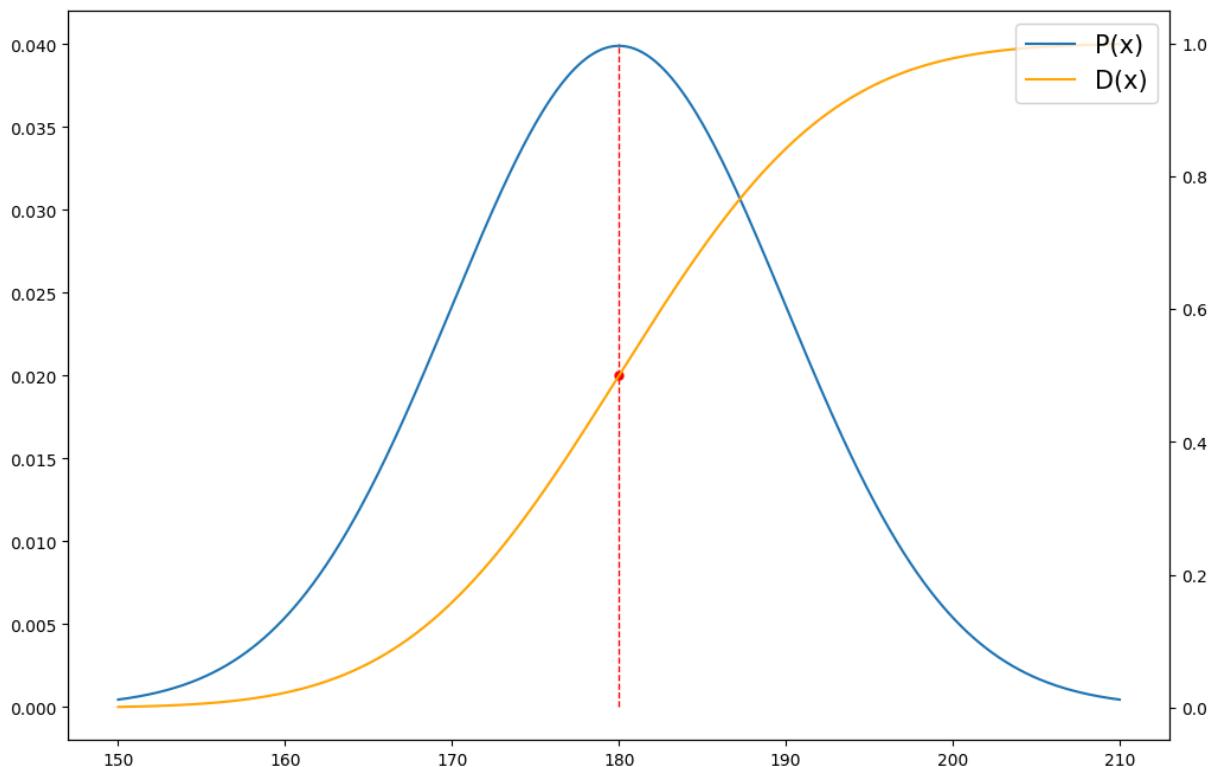
# на оси x и правой оси y построим график функции распределения (cdf)
ax_right.plot(x, y2, color = 'orange', label = 'D(x)')

# также построим вертикальную прямую и точку
ax_left.vlines(x = 180, ymin = 0, ymax = 0.040, linewidth = 1, color = 'r',
ax_left.plot(180, 0.020, marker = 'o', markersize = 5, markeredgecolor = 'r')

# из-за двух осей с легендой придется повозиться
fig.legend(loc = 'upper right',
           bbox_to_anchor = (1,1),
           bbox_transform = ax_right.transAxes,
           prop = {'size': 15})

plt.show()

```



Нахождение вероятности через интеграл

Здесь вначале посмотрим на функцию распределения $D(x)$ (оранжевый график). В промежутке от 150 до 210 см эта функция непрерывно возрастает, при этом она возрастает с разной скоростью. До точки $x = 180$ (она называется точкой перегиба, inflection point) скорость возрастания функции увеличивается, после нее убывает.

Именно это изменение описывает производная от нее функция плотности $P(x)$ (синий график). На участке от 150 до 180 она возрастает, а потом в интервале от 180 до 210 постоянно убывает.

Таким образом, плотность вероятности описывает скорость изменения функции распределения.

```
In [ ]: # воспользуемся модулем integrate библиотеки scipy
import scipy.integrate as integrate

# зададим среднее значение и СКО
mu, sigma = 180, 10

# и границы интервала от 190 до бесконечности
lowerbound = 190
upperbound = np.inf

# функция quad() в качестве первого аргумента ожидает функцию для интегрирования
# напишем функцию gauss() с одним параметром
def gauss(x):
    return norm.pdf(x, mu, sigma)

# передадим в функцию quad() функцию Гаусса, а также нижний и верхний предел
# поместим первое выводимое значение в переменную integral
integral = integrate.quad(gauss, lowerbound, upperbound)[0]

# так как мы получили вероятность от 190 см и выше (то есть площадь справа)
# для нахождения вероятности не более 190 см результат нужно вычесть из единицы
1 - integral
```

Out[]: 0.8413447460685429

Выборки

Вероятность конкретного значения

Важный и немного континтуитивный момент. Вероятность того, что случайная величина *непрерывного* распределения примет конкретное значение (то есть нам встретится человек определенного роста) равна **нулю**. Это легко продемонстрировать.

Ранее мы находили вероятность встретить человека ростом от 170 до 190 сантиметров, вычитая меньшую площадь под кривой из большей.

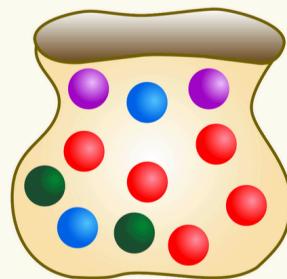
$$P(170 \leq x \leq 190) = P(x \leq 190) - P(x \leq 170) \approx 0,68$$

При этом если мы таким же способом постараемся найти вероятность встретить человека ростом ровно 190 сантиметров, то она очевидно будет равна нулю.

$$P(x = 190) = P(x \leq 190) - P(x \leq 190) = 0$$

Формирование выборки

Сделаем небольшое отступление и рассмотрим процесс **формирования выборки** (sampling). Все очень несложно. Мы берем некоторый набор элементов и из него случайным образом достаем какое-то их количество.



```
In [ ]: # возьмем вот такой мешок с разноцветными шарами  
bag = ['red', 'yellow', 'green', 'gray', 'black', 'orange', 'white', 'blue',
```

При этом формировать выборку можно двумя способами. В первом случае мы случайным образом достаем элемент, но, прежде чем взять следующий, кладем этот элемент обратно. Такой процесс называют выборкой с возвращением (sampling with replacement), и имитировать его можно с помощью функции `np.random.choice()`.

```
In [ ]: np.random.seed(42)  
  
# выберем с возвращением восемь шаров  
np.random.choice(bag, 8)
```

```
Out[ ]: array(['white', 'gray', 'blue', 'black', 'white', 'pink', 'green',  
       'white'], dtype='<U6')
```

Обратите внимание, белый шар повторяется дважды.

Кроме того, можно сформировать выборку без возвращения (sampling without replacement). В этом случае мы не кладем элемент обратно, а откладываем в сторону и только потом достаем следующий элемент. Для этого функции np.random.choice() нужно задать параметр replace = False.

```
In [ ]: np.random.seed(42)
```

```
# сделаем то же самое, только возвращать шары не будем  
np.random.choice(bag, 8, replace = False)
```

```
Out[ ]: array(['brown', 'yellow', 'orange', 'red', 'blue', 'green', 'pink',  
       'black'], dtype='<U6')
```

```
In [ ]: np.random.seed(42)
```

```
# выберем 5 чисел из массива от 0 до 9  
# "под капотом" массив из 10 чисел формируется с помощью np.arange(10)  
np.random.choice(10, 5)
```

```
Out[ ]: array([6, 3, 7, 4, 6])
```

ЦПТ

Центральная предельная теорема

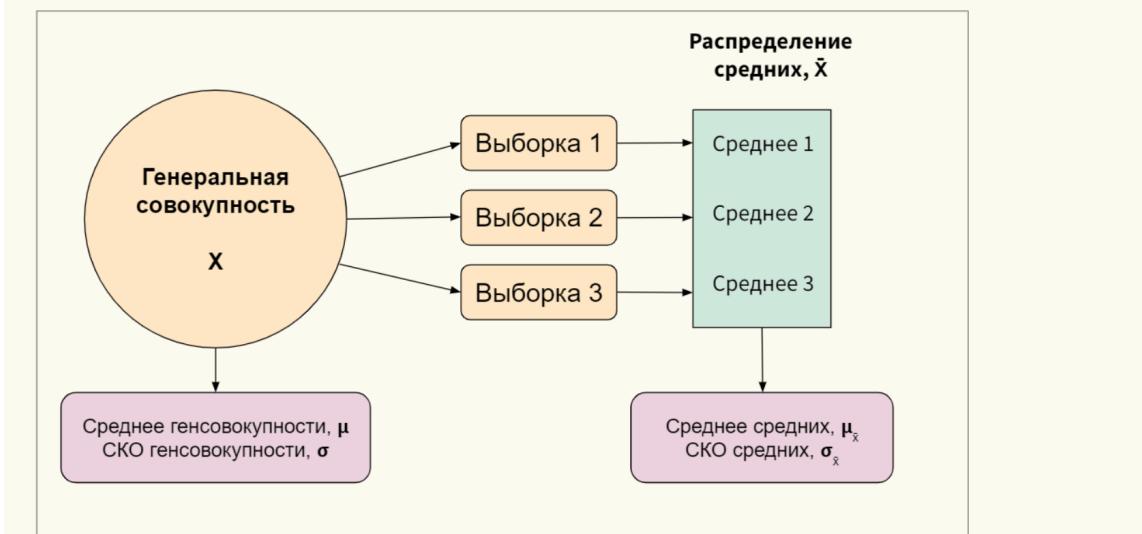
Помимо процессов в организме и природных явлений, нормальное распределение имеет большое значение для Центральной предельной теоремы (Central Limit Theorem).

Определения и нотация

Вначале вспомним несколько терминов и введем полезные обозначения.

Во-первых, напомню, что данные могут представлять собой генеральную совокупность (population) или выборку из нее (sample). Если мы берем несколько выборок из одной генеральной совокупности и измеряем для каждой из них определенный параметр (например, среднее арифметическое, sample mean), то совокупность этих средних формирует выборочное распределение (sampling distribution).

При изучении Центральной предельной теоремы нас будет интересовать поведение именно этого распределения, его среднее арифметическое и среднее квадратическое отклонение.



ЦПТ и нормальное распределение

Мы уже знаем, что среднее средних нескольких выборок (mean of sampling distribution of sample means) из одной генеральной совокупности будет стремиться к истинному среднему этой генеральной совокупности.

$$\mu_{\bar{x}} = \mu$$

Однако это не все. При соблюдении двух условий, а именно, (1) выборки сформированы случайным образом, (2) размер каждой выборки составляет не менее 30 элементов, выборочные средние будут следовать нормальному распределению.

Более того, распределение самой генеральной совокупности при этом *не обязательно* должно быть нормальным.

Одновременно, среднее квадратическое отклонение распределения средних будет стремиться к СКО генсовокупности, разделенному на квадратный корень размера одной выборки.

$$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$$

Математически эти выводы можно записать так

$$X \sim dist(\mu, \sigma) \rightarrow \bar{X} \sim \mathcal{N}\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

Проверим на Питоне

А теперь давайте проверим истинность ЦПТ с помощью Питона. Для начала создадим скошенное вправо распределение (right skewed distribution). Такое распределение может характеризовать, например, зарплаты людей.

Пример скошенного вправо распределения

```
In [ ]: # скошенное распределение мы будем строить
# с помощью объекта skewnorm модуля stats библиотеки scipy
from scipy.stats import skewnorm

# сгенерируем массив с данными о зарплате, задав искусственные параметры
salaries = skewnorm.rvs(a = 20, # скошенность (skewness)
                        loc = 20, # среднее значение без учета скошенности
                        scale = 80, # отклонение от среднего
                        size = 100000, # размер генерируемого массива
                        random_state = 42) # воспроизводимость результата
```

```
In [ ]: # зададим размер графика
plt.figure(figsize = (10,6))
```

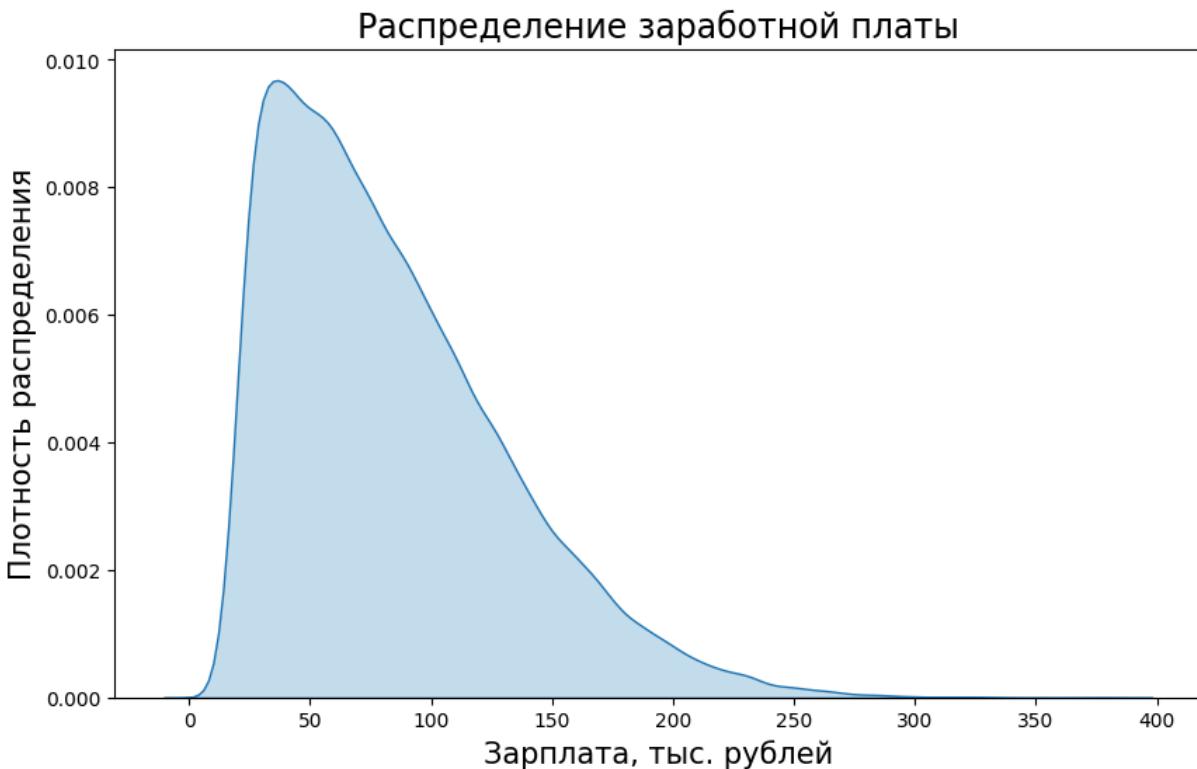
```

# и построим график функции плотности
sns.kdeplot(salaries, fill = True)

# добавим подписи
plt.xlabel('Зарплата, тыс. рублей', fontsize = 15)
plt.ylabel('Плотность распределения', fontsize = 15)
plt.title('Распределение заработной платы', fontsize = 17)

plt.show()

```



Большая часть зарплат находится в нижней границе диапазона, при этом справа есть большой хвост тех немногих, чья заработка выше среднего. Рассчитаем среднее значение и медиану.

In []: # посмотрим на среднее значение
np.mean(salaries)

Out[]: 83.84511076271755

In []: # посмотрим на медиану
np.median(salaries)

Out[]: 74.07573752156253

Как и должно быть, медианное значение меньше среднего арифметического, на которое влияют небольшое количество очень высоких зарплат. По этой причине, как мы уже говорили, для измерения средней зарплаты предпочтительнее использовать именно медиану.

Рассчитаем СКО.

```
In [ ]: # рассчитаем СКО  
np.std(salaries)
```

```
Out[ ]: 48.366511941114176
```

Выборки с возвращением

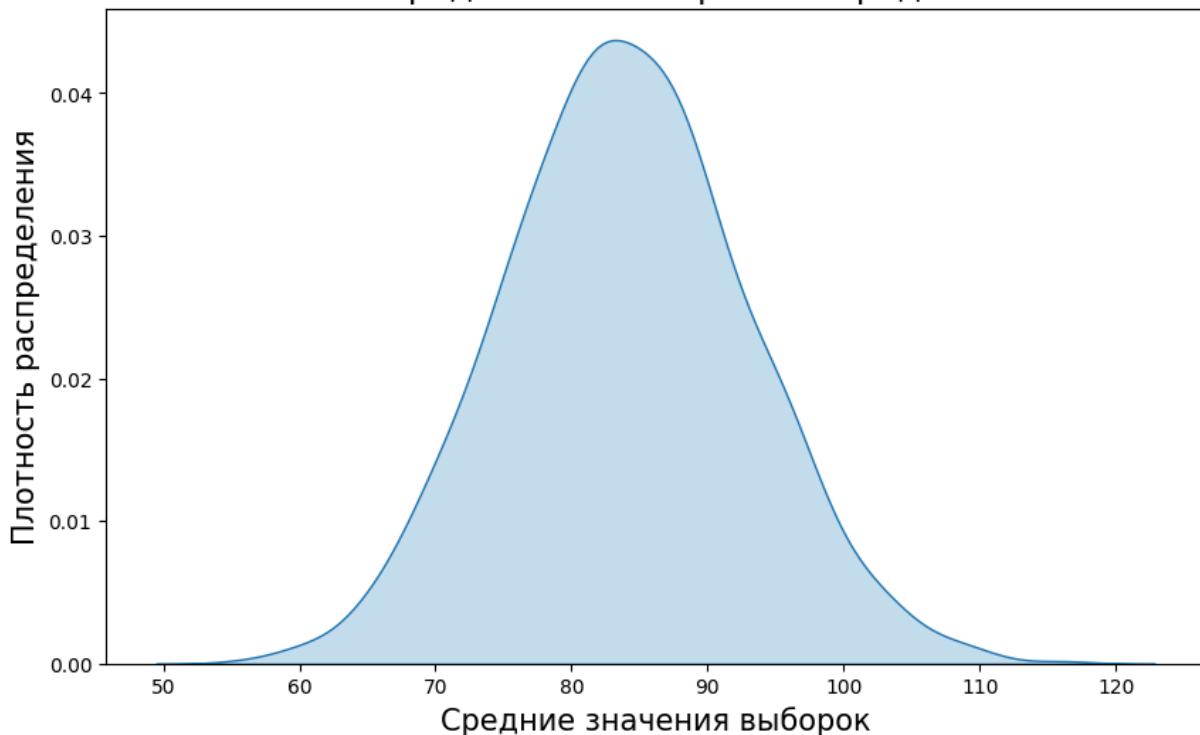
Выборки с возвращением. Теперь давайте брать выборки из нашей скошенной генеральной совокупности salaries и смотреть, что произойдет с распределением выборочных средних. Вначале создадим необходимое количество выборок.

```
In [ ]: # зададим точку отсчета  
np.random.seed(42)  
  
# создадим список, в который будем записывать выборочные средние  
sample_means = []  
  
# зададим количество и размер выборок  
n_samples = 1000  
sample_size = 30  
  
# в цикле будем формировать выборки  
for i in range(n_samples):  
  
    # путем отбора нужного количества элементов из генеральной совокупности  
    sample = np.random.choice(salaries, sample_size, replace = False)  
  
    # для каждой выборки рассчитаем среднее значение и поместим в список выборок  
    sample_means.append(np.mean(sample))  
  
# убедимся, что сформировали нужное количество выборок  
len(sample_means)
```

```
Out[ ]: 1000
```

```
In [ ]: # зададим размер графика  
plt.figure(figsize = (10, 6))  
  
# и построим график функции плотности  
sns.kdeplot(sample_means, fill = True)  
  
# добавим подписи  
plt.xlabel('Средние значения выборок', fontsize = 15)  
plt.ylabel('Плотность распределения', fontsize = 15)  
plt.title('Распределение выборочных средних', fontsize = 17)  
  
plt.show()
```

Распределение выборочных средних



Обратите внимание, распределение выборочных средних гораздо ближе к нормальному распределению, нежели исходное распределение salaries.

Остается рассчитать значения, к которым, согласно ЦПТ, должны стремиться среднее значение и СКО выборочных средних.

```
In [ ]: # посмотрим к чему, согласно ЦПТ, должны стремиться среднее и СКО выборочного
        np.mean(salaries), np.std(salaries)/np.sqrt(sample_size)
```

```
Out[ ]: (83.84511076271755, 8.830476539330403)
```

```
In [ ]: # и сравним с фактическими данными
        np.mean(sample_means), np.std(sample_means)
```

```
Out[ ]: (83.93586714203595, 8.94762726282006)
```

Выборки без возвращения

Как мы видим, ЦПТ подтверждается.

Выборки без возвращения. Теперь в качестве упражнения, давайте понаблюдаем, как будет меняться распределение выборочных средних при различных значениях количества выборок и их размера.

Кроме того, на этот раз будем делать выборку без возвращения, потому что если вы обратите внимание, несмотря на указанный параметр `replace = False`, на каждой итерации приведенного выше алгоритма мы формировали выборку из одной и той же генеральной совокупности, а значит элементы могли повторяться.

Важно, что при формировании выборки без возвращения ЦПТ будет выполняться, если размер одной выборки составляет не более 5% от размера генеральной совокупности.

Для формирования распределения выборок без возвращения напишем собственную функцию `sample_means()`. На входе она будет принимать следующие параметры:

- `data` — набор данных (генеральная совокупность);
- `n_samples` — количество выборок;
- `sample_size` — размер одной выборки;
- `replace = True` — с возвращением делать выборки или без;
- `random_state = None` — воспроизводимость результата.

```
In [ ]: # объявим собственную функцию для формирования распределения выборочных средних
def sample_means(data, n_samples, sample_size, replace = True, random_state = None):
    # создадим список, в который будем записывать выборочные средние
    sample_means = []

    # пропишем воспроизводимость
    np.random.seed(random_state)

    # в цикле будем формировать выборки
    for i in range(n_samples):

        # путем отбора нужного количества элементов из генеральной совокупности
        sample = np.random.choice(data, sample_size, replace = False)

        # для каждой выборки рассчитаем среднее значение и поместим в список выборок
        sample_means.append(np.mean(sample))

    # если указано, что выборки делаются без возвращения
    if replace == False:

        # удалим эту выборку из данных о зарплатах
        data = np.array(list(set(data) - set(sample)))

    # вернем список с выборочными средними
    return sample_means
```

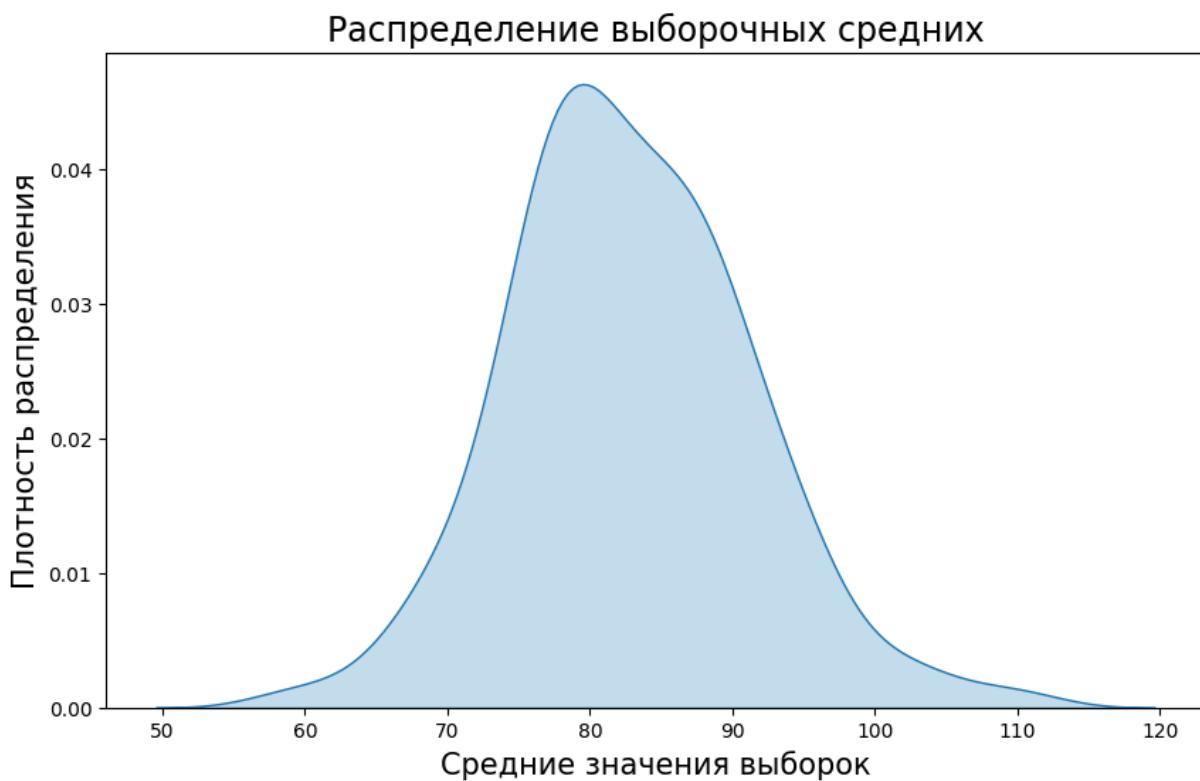
```
In [ ]: # сформируем 100 выборок без возврата по 30 элементов в каждой
res = sample_means(salaries, 100, 30, replace = False, random_state = 42)
```

```
In [ ]: # зададим размер графика
plt.figure(figsize = (10,6))

# и построим график функции плотности
sns.kdeplot(res, fill = True)

# добавим подписи
plt.xlabel('Средние значения выборок', fontsize = 15)
plt.ylabel('Плотность распределения', fontsize = 15)
plt.title('Распределение выборочных средних', fontsize = 17)

plt.show()
```



Распределение выборочных средних при разном количестве и размере выборок

```
In [ ]: # создадим сетку подграфиков 3 x 3
fig, ax = plt.subplots(nrows = 3, ncols = 3, figsize = (16,12))

# списки с разным количеством
number_of_samples_list = [20, 100, 500]
# и размером выборок
sample_size_list = [2, 10, 30]

# кроме этого, создадим списки для учета получившихся
# средних значений
mean_list = []
```

```
# и СКО
std_list = []

# по строкам сетки 3 x 3 разместим разное количество выборок
for i, n_samples in enumerate(number_of_samples_list):
    # по столбцам разный размер выборки
    for j, sample_size in enumerate(sample_size_list):

        # на каждой итерации будем генерировать "свежую" генеральную совокупность
        salaries = skewnorm.rvs(a = 20, loc = 20, scale = 80, size = 100000, random_state = np.random.randint(1, 100))

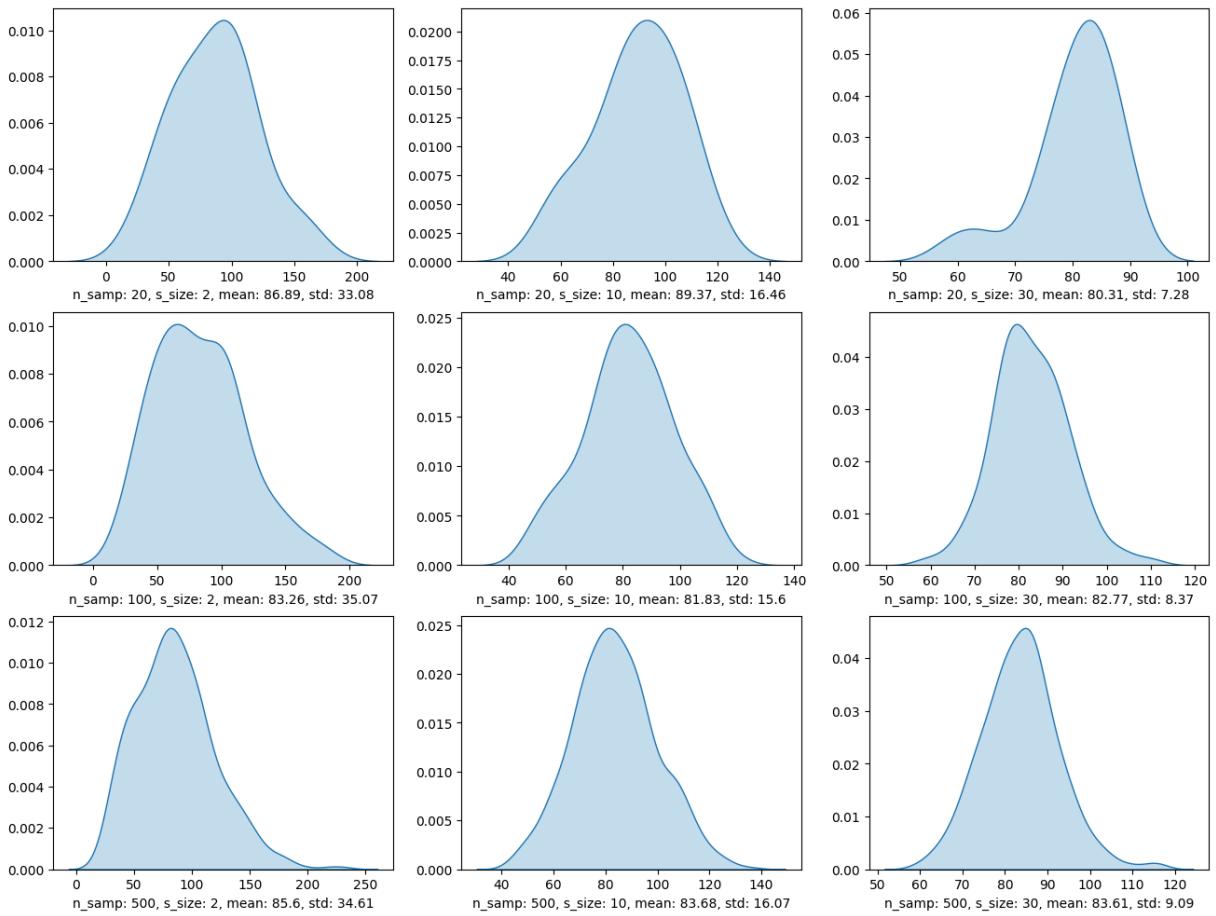
        # и создавать распределение с заданными параметрами
        res = sample_means(salaries, n_samples, sample_size, replace = False, random_state = np.random.randint(1, 100))

        # кроме того, мы вычисляем среднее значение и СКО
        mean = np.mean(res).round(2)
        std = np.std(res).round(2)

        # и записываем их в соответствующие списки
        mean_list.append(mean)
        std_list.append(std)

        # помещаем график плотности распределения в "ячейку" с координатами [i, j]
        sns.kdeplot(res, fill = True, ax = ax[i, j])
        # а под графиком выводим количество выборок, размер выборки, среднее значение и СКО
        ax[i, j].set_xlabel('n_samp: {}, s_size: {}, mean: {}, std: {}'.format(n_samples, sample_size, mean, std))
        # подпись к оси у оставляем пустой
        ax[i, j].set_ylabel('')

# выведем результат
plt.show()
```



```
In [ ]: # вновь создадим генеральную совокупность
salaries = skewnorm.rvs(a = 20, loc = 20, scale = 80, size = 100000, random_
# в словарь sampling_distributions поместим
sampling_distributions = {
    # количество выборок в каждом из девяти распределений
    'Number_of_samples' : np.repeat(number_of_samples_list, 3),
    # размер каждой выборки
    'Sample_size' : sample_size_list * 3,
    # среднее значение генеральной совокупности
    'Pop_mean' : [np.mean(salaries)] * 9,
    # фактическое среднее значение каждого распределения
    'Actual_mean' : mean_list,
    # расчетное СКО (в соответствии с ЦПТ)
    'Expected_std' : [np.std(salaries) / np.sqrt(n) for n in sample_size_list],
    # фактическое СКО каждого распределения
    'Actual_std' : std_list
}

# превратим словарь в датафрейм и округлим значения
pd.DataFrame(sampling_distributions).round(2).astype(str)
```

Out[]:	Number_of_samples	Sample_size	Pop_mean	Actual_mean	Expected_std	/
0	20	2	83.85	86.89	34.2	
1	20	10	83.85	89.37	15.29	
2	20	30	83.85	80.31	8.83	
3	100	2	83.85	83.26	34.2	
4	100	10	83.85	81.83	15.29	
5	100	30	83.85	82.77	8.83	
6	500	2	83.85	85.6	34.2	
7	500	10	83.85	83.68	15.29	
8	500	30	83.85	83.61	8.83	

Стандартное нормальное распределение

Стандартное нормальное распределение

Любое нормальное распределение со средним значением μ и СКО σ можно привести к стандартному нормальному распределению (standard normal distribution) со средним значением ноль и СКО равным единице.

$$Z \sim \mathcal{N}(0, 1)$$

Для этого воспользуемся следующей формулой.

$$z = \frac{x - \mu}{\sigma}$$

Таким образом мы приводим каждое значение x к соответствующей z-оценке (z-score), вычитая среднее μ и деля результат на СКО σ . Например, приведем данные о росте мужчин к стандартному виду (для этого воспользуемся [векторизацией и трансляцией кода](#)).

```
In [ ]: # воспользуемся векторизацией и трансляцией кода
# для приведения распределения к стандартному виду
height_men_standard = (height_men - np.mean(height_men))/np.std(height_men)
height_men_standard[:10]
```

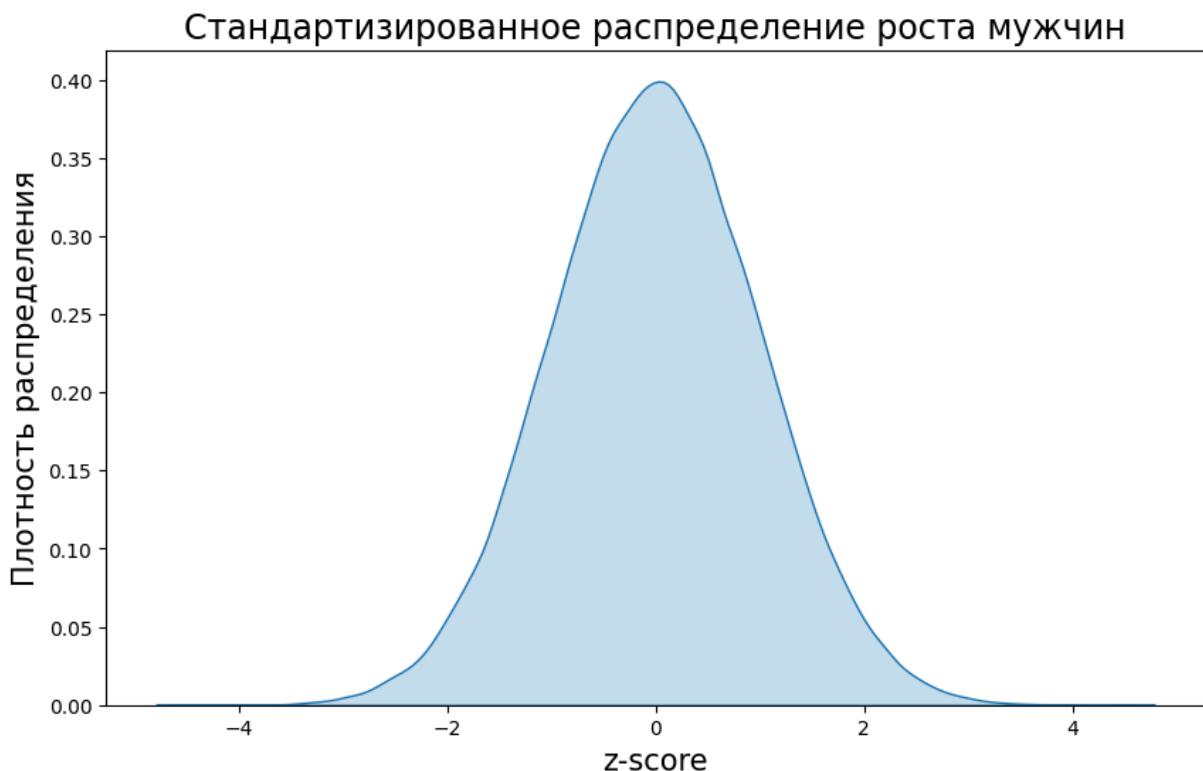
```
Out[ ]: array([ 0.49836188, -0.10076518,  0.59821639,  1.49690697, -0.20061969,
   -0.20061969,  1.59676148,  0.7979254 , -0.50018322,  0.49836188])
```

```
In [ ]: # зададим размер графика
plt.figure(figsize = (10,6))

# построим график плотности
sns.kdeplot(height_men_standard, fill = True)

# добавим подписи
plt.xlabel('z-score', fontsize = 15)
plt.ylabel('Плотность распределения', fontsize = 15)
```

```
plt.title('Стандартизированное распределение роста мужчин', fontsize = 17)  
plt.show()
```



Стоит сказать, что ровно такого же результата можно добиться, применив метод `.fit_transform()` класса `StandardScaler` модуля `preprocessing` библиотеки `sklearn`.

```
In [ ]: # импортируем класс StandardScaler  
from sklearn.preprocessing import StandardScaler  
  
# создаем объект этого класса  
scaler = StandardScaler()  
  
# применяем метод .fit_transform() к данным о росте, предварительно превратив  
scaled_height_men = scaler.fit_transform(height_men.reshape(-1, 1))  
  
# убираем второе измерение  
scaled_height_men = scaled_height_men.flatten()  
  
# и сравниваем получившийся результат с ранее стандартизованными данными  
np.array_equiv(height_men_standard, scaled_height_men)
```

Out[]: True

```
In [ ]: # сразу создадим стандартное нормальное распределение  
st_norm = np.random.standard_normal(10000)  
st_norm[:10]
```

```
Out[ ]: array([-0.76366945, -1.11864174, -0.77088274,  0.96929238, -0.69869956,
   -1.34853749, -0.94994848,  0.99902749,  0.30575659, -0.45619504])
```

```
In [ ]: # зададим размер графика
plt.figure(figsize = (10,6))

# построим график плотности
sns.kdeplot(st_norm, fill = True)

# добавим подписи
plt.xlabel('z-score', fontsize = 15)
plt.ylabel('Плотность распределения', fontsize = 15)
plt.title('Стандартное нормальное распределение', fontsize = 17)

plt.show()
```



```
In [ ]: # создаваемый массив может быть многомерным
np.random.standard_normal((2, 3))
```

```
Out[ ]: array([[-0.40187015, -0.0797878 , -0.21066058],
   [-0.70034594,  0.10488661,  0.85464581]])
```

```
In [ ]: # вычислим z-score, соответствующий 95 процентам площади под кривой
# loc = 0, scale = 1 можно не указывать, это параметры по умолчанию
zscores = norm.ppf(0.95)
zscores
```

```
Out[ ]: 1.6448536269514722
```

```
In [ ]: # убедиться в верности результата можно с помощью функции распределения
norm.cdf(zscores)
```

Out[]: 0.95

Критерии нормальности

Критерии нормальности распределения

Давайте еще раз посмотрим на распределение выборочных средних. Мы сказали, что при определенных условиях оно стремится к нормальному.

Одновременно пока что мы определяли нормальность распределения «на глаз» (с помощью гистограммы, графика плотности или boxplot), хотя в некоторых случаях полезно иметь более надежный критерий нормальности. Это важно, например, при оценке:

- распределения остатков модели линейной регрессии;
- распределения наблюдений в классах модели линейного дискриминантного анализа (Linear Discriminant Analysis, LDA).

Рассмотрим два способа оценки нормальности распределения.

Способ 1. График нормальной вероятности

График нормальной вероятности (normal probability plot) показывает соотношение упорядоченных по возрастанию данных и соответствующих им квантилей нормального распределения.

Если данные распределены нормально, все точки будут лежать на одной прямой, если нет — мы будем наблюдать отклонения.

Алгоритм создания графика нормальной вероятности следующий:

1. сортируем исходные данные по возрастанию;
2. находим накопленную вероятность (cumulative probability) каждого значения;
3. с помощью квантиль-функции выясняем, какому квантилю соответствовала бы эта вероятность, если бы распределение было нормальным;
4. по оси x отмечаем квантили, по оси y — отсортированные данные.

Накопленную вероятность будем вычислять по формуле

$$P = \frac{i - 0.375}{n + 0.25},$$

где i — это индекс (начиная с единицы) значения в перечне отсортированных данных, а n — количество наблюдений.

С помощью Питона построим график нормальной вероятности для данных о росте.

Probability plot

```
In [ ]: # вначале отсортируем данные о росте
height_men_srt = sorted(height_men)

# рассчитаем длину массива
n = len(height_men_srt)

# вычислим накопленную вероятность
cum_probability = [(i - 0.375)/(n + 0.25) for i in range(1, n + 1)]

# рассчитаем квантили, как если бы данные были нормально распределены
quantiles = norm.ppf(cum_probability)
```

```
In [ ]: # как и должно быть, накопленная вероятность - это значения от 0 до 1
cum_probability[0], cum_probability[-1]
```

```
Out[ ]: (6.249984375039063e-06, 0.999993750015625)
```

```
In [ ]: # также посмотрим на соответствующие им квантили
quantiles[0], quantiles[-1]
```

```
Out[ ]: (-4.368680139037586, 4.368680139037566)
```

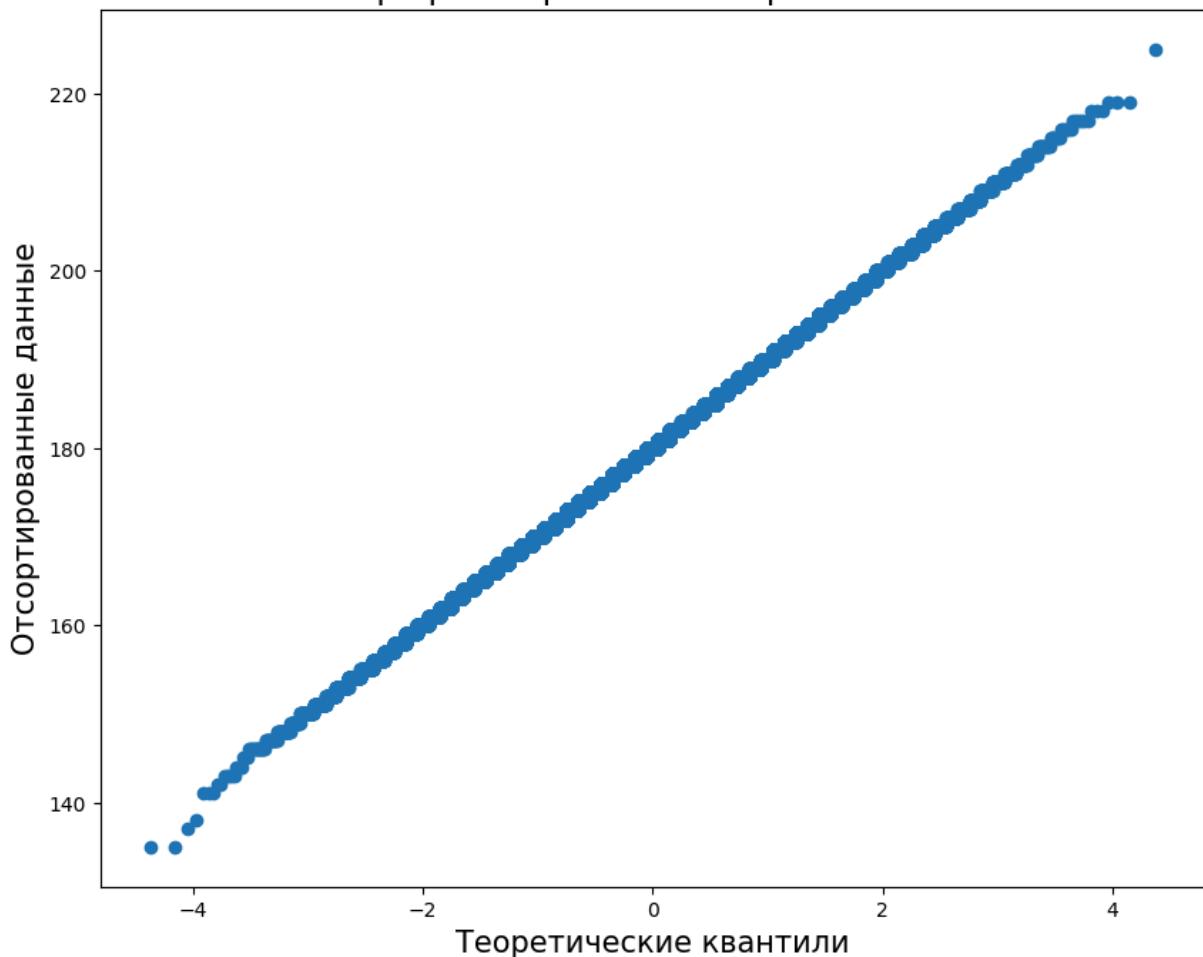
```
In [ ]: # зададим размер графика
plt.figure(figsize = (10,8))

# на точечной диаграмме по оси x разместим теоретические квантили
# по оси у - отсортированные по возрастанию данные
plt.scatter(quantiles, height_men_srt)

# добавим подписи
plt.xlabel('Теоретические квантили', fontsize = 15)
plt.ylabel('Отсортированные данные', fontsize = 15)
plt.title('График нормальной вероятности', fontsize = 17)

plt.show()
```

График нормальной вероятности

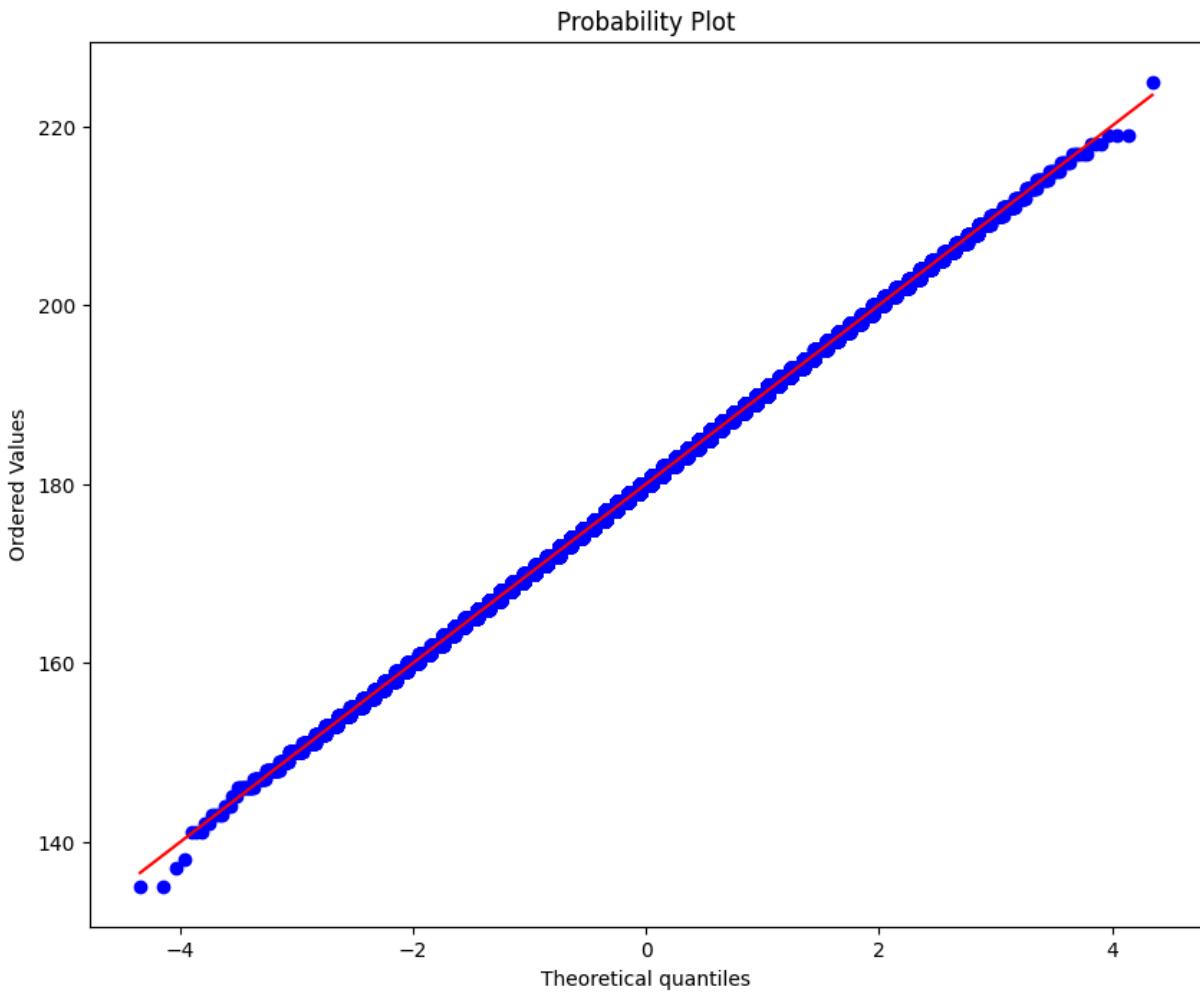


```
In [ ]: # можно также воспользоваться функцией probplot() модуля stats библиотеки scipy.stats import probplot
```

```
In [ ]: plt.figure(figsize = (10,8))

# параметр dist = 'norm' указывает на сравнение данных с нормальным распределением
# plot = plt строит график с помощью matplotlib.pyplot
probplot(height_men, dist = 'norm', plot = plt)

plt.show()
```

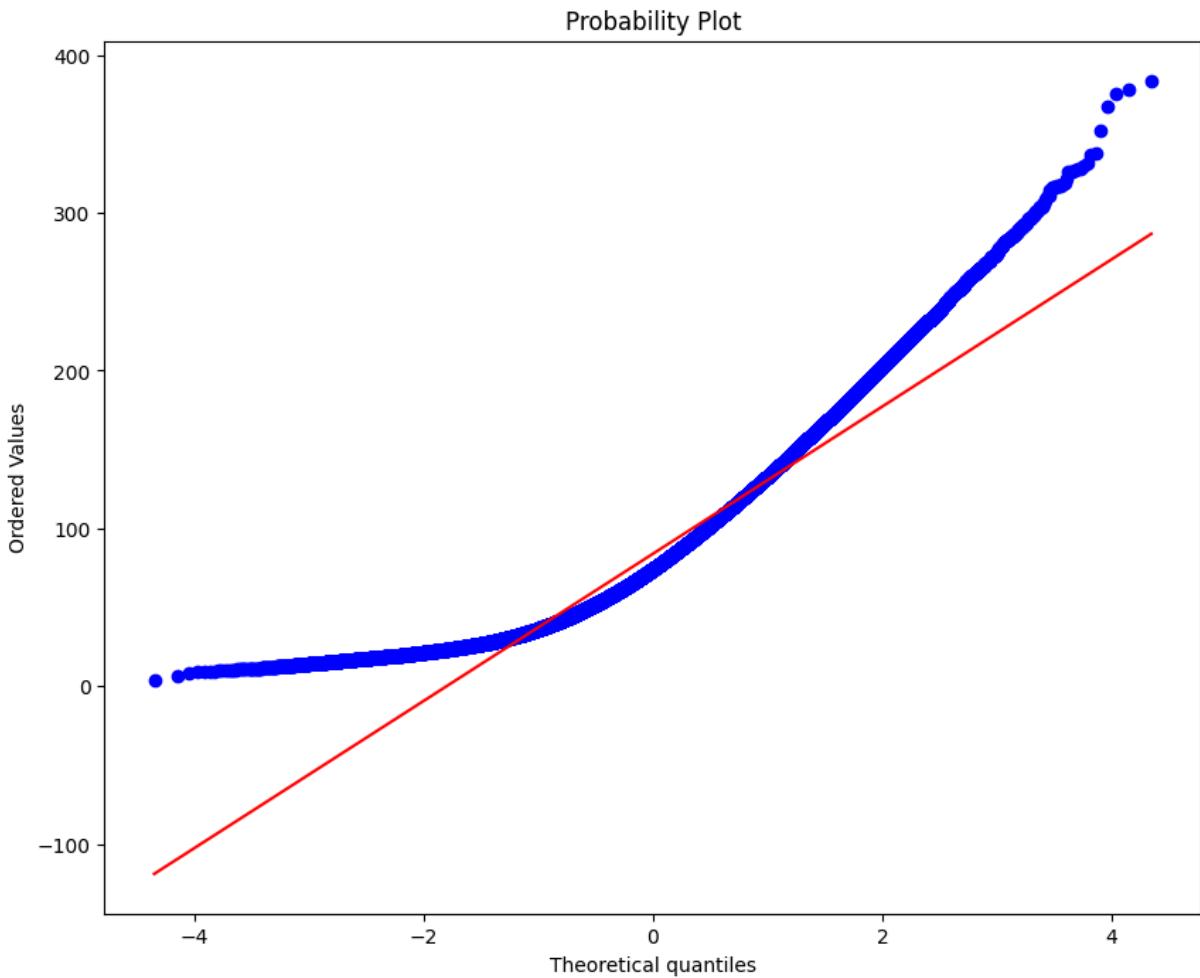


```
In [ ]: # функция probplot() при параметрах plot = None, fit = False возвращает два
# квантили и отсортированные данные, возьмем первый массив [0]
quantiles = probplot(height_men, dist = 'norm', plot = None, fit = False)[0]

# и посмотрим на первое и последнее значения
quantiles[0], quantiles[-1]
```

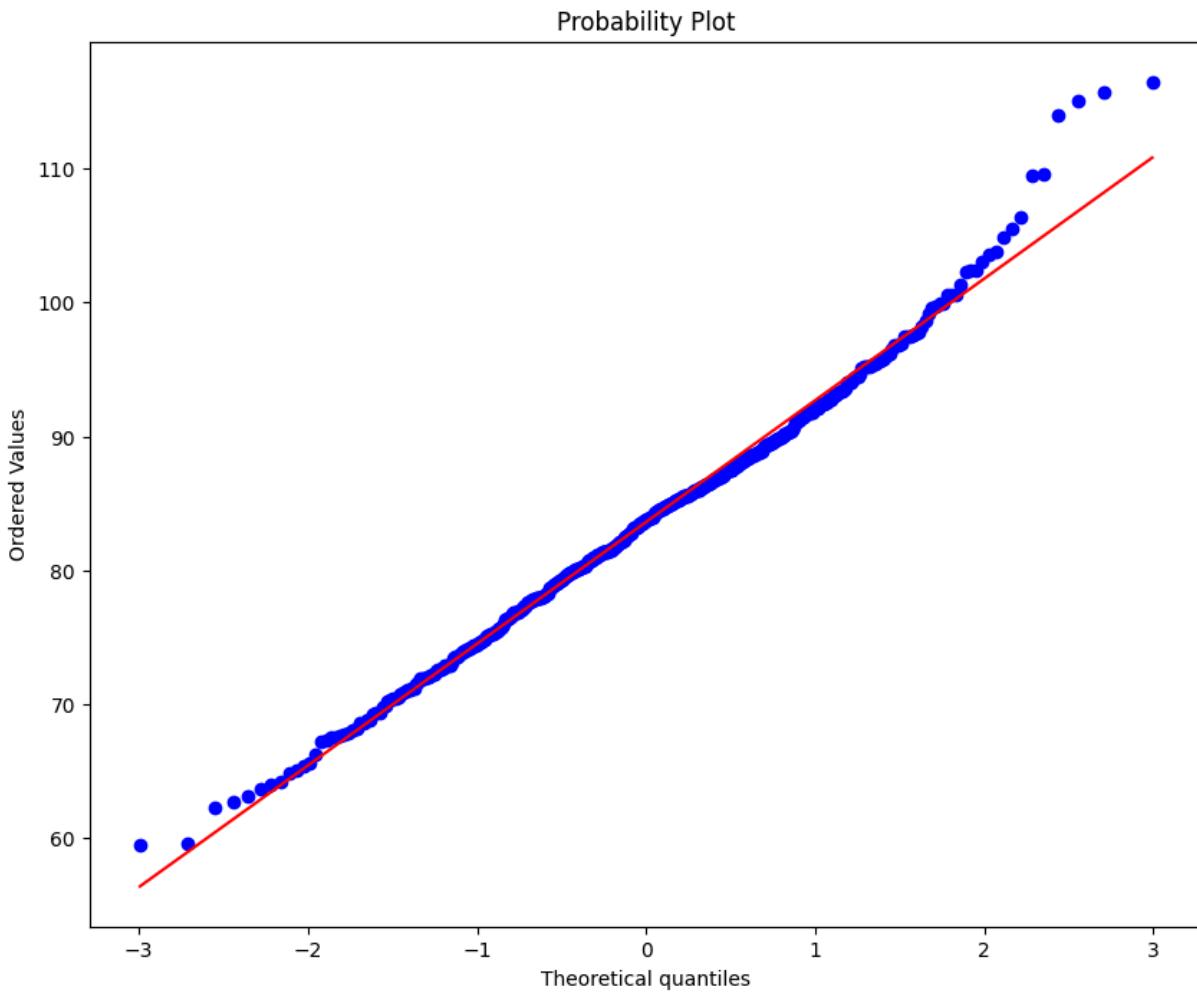
```
Out[ ]: (-4.346021549886044, 4.346021549886044)
```

```
In [ ]: # построим график нормальной вероятности для распределения зарплат
plt.figure(figsize = (10,8))
probplot(salaries, dist = 'norm', plot = plt)
plt.show()
```



```
In [ ]: # снова воспользуемся функцией sample_means() с параметрами n_samples = 500,
sampling_dist = sample_means(salaries, 500, 30, replace = False, random_stat

# построим график нормальной вероятности
plt.figure(figsize = (10,8))
probplot(sampling_dist, dist = 'norm', plot = plt)
plt.show()
```



Тест Шапиро-Уилка (Shapiro-Wilk test)

Способ 2. Тест Шапиро-Уилка

Тест Шапиро-Уилка (Shapiro-Wilk test) позволяет сделать статистически значимый вывод о нормальности распределения:

- нулевая гипотеза предполагает, что распределение нормально;
- альтернативная гипотеза утверждает обратное.

Тест Шапиро-Уилка чувствителен к количеству элементов (N) в наборе данных и теряет точность при $N > 5000$.

Проведем тест для распределений роста и распределения зарплат при пороговом значении 0,05, однако вначале создадим распределения с меньшим количеством элементов.

```
In [ ]: # вновь создадим данные о росте и зарплатах с меньшим количеством элементов
np.random.seed(42)
```

```
height_men = np.round(np.random.normal(180, 10, 1000))

salaries = skewnorm.rvs(a = 20, loc = 20, scale = 80, size = 1000, random_st
```

```
In [ ]: # импортируем функцию shapiro()
from scipy.stats import shapiro

# проведем тест на нормальность для распределения данных о росте
_, p_value = shapiro(height_men)
p_value
```

```
Out[ ]: 0.17031297087669373
```

```
In [ ]: # проведем тест на нормальность для распределения данных о зарплате
_, p_value = shapiro(salaries)
p_value
```

```
Out[ ]: 1.3727537853780034e-21
```

Нормальное приближение

Нормальное приближение биномиального распределения

Теорема Муавра-Лапласа

Теорема Муавра-Лапласа (de Moivre-Laplace theorem), частный случай Центральной предельной теоремы, утверждает, что при определенных условиях нормальное распределение может быть использовано в качестве приближения биномиального распределения (Normal Approximation to Binomial Distribution).

Напомню формулу биномиального распределения

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

где n — количество испытаний, k — количество успехов, а p — вероятность успеха.

Так вот если $np \geq 5$ и $n(1 - p) \geq 5$, то выполняется следующее

$$B(n, p) \sim \mathcal{N}(np, \sqrt{np(1 - p)})$$

Другими словами, если взять вероятность успеха p близкое к 0,5, либо достаточно большое количество испытаний n , то биномиальное распределение по мере увеличения n будет приближаться нормальному распределению.

Проиллюстрируем теорему с помощью Питона. Будем подбрасывать несимметричную монету ($p = 0,8$) по 3, 5, 10, 15, 25 и 50 раз и сравнивать получившиеся распределения с нормальным.

Теорема Муавра-Лапласа

$$B(n, p) \sim N(np, \sqrt{np(1-p)})$$

```
In [ ]: # создадим список с количеством испытаний (подбрасываний монеты)
n_list = [3, 5, 10, 15, 25, 50]

# подграфики будут расположены в одном столбце с количеством строк,
# равном длине списка n_list
fig, ax = plt.subplots(nrows = len(n_list),
                      ncols = 1,
                      # размер графика также будет определяться длиной списка
                      figsize = (8, 4 * len(n_list)),
                      # constrained_layout регулирует пространство между по-
                      constrained_layout = True)

# пройдемся по списку n_list
for i, n in enumerate(n_list):

    # зададим вероятность успеха
    p = 0.8
    # и проверим выполняются ли условия при заданных p и n
    cond1, cond2 = n * p, n * (1 - p)

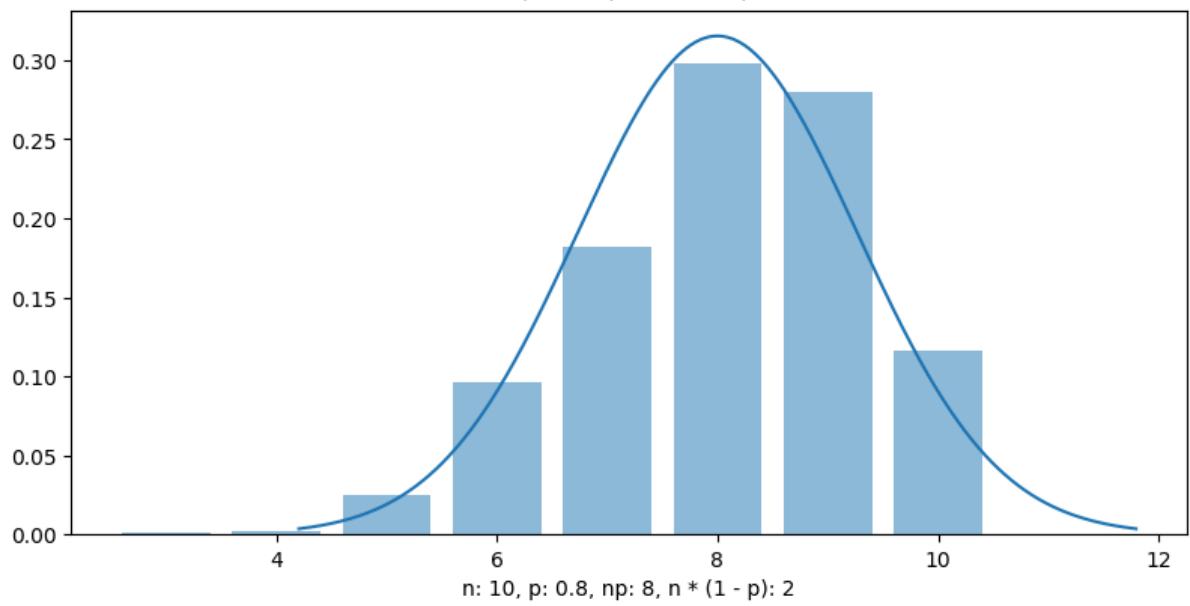
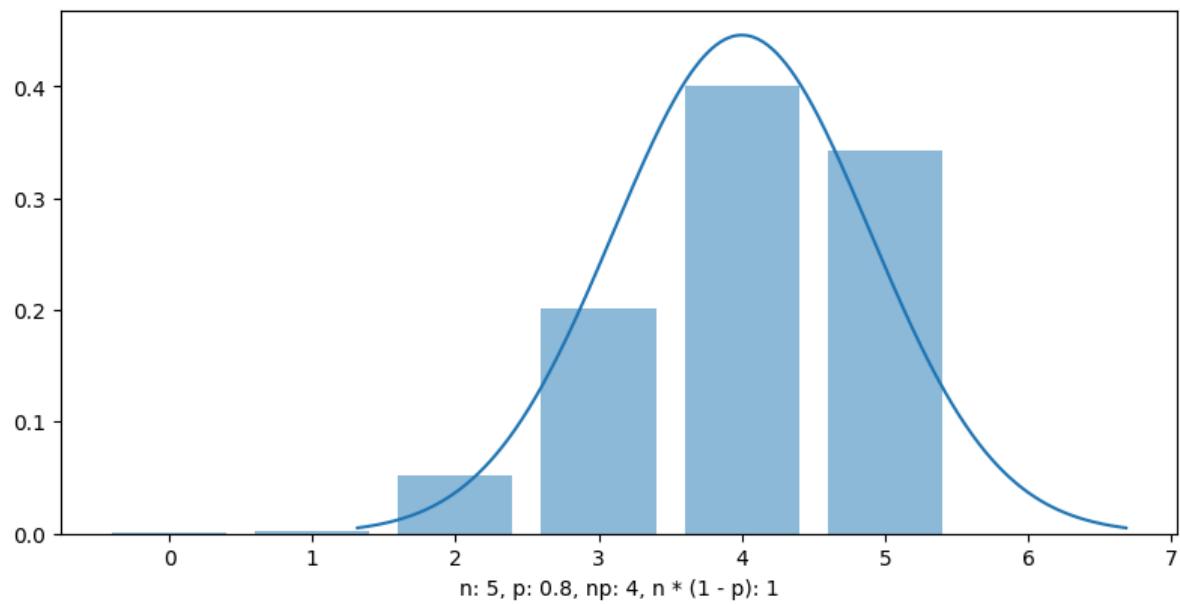
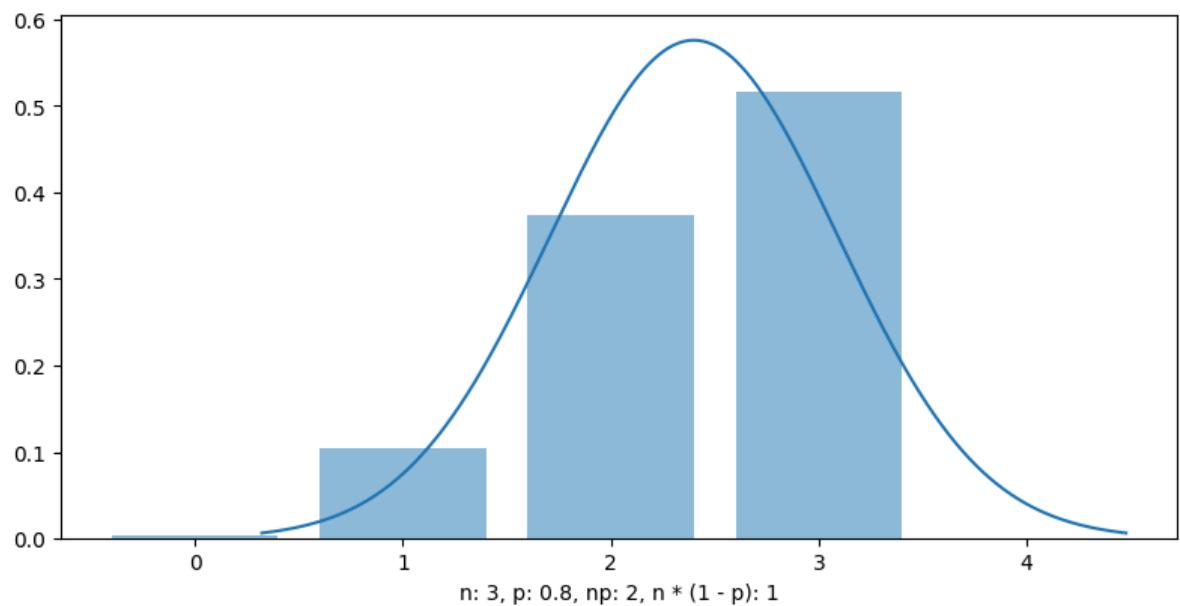
    # сгенерируем биномиальное распределение с заданными p и n
    np.random.seed(42)
    res = np.random.binomial(n = n, p = p, size = 1000)
    # запишем успешные серии подбрасываний и их количество
    successes, counts = np.unique(res, return_counts = True)

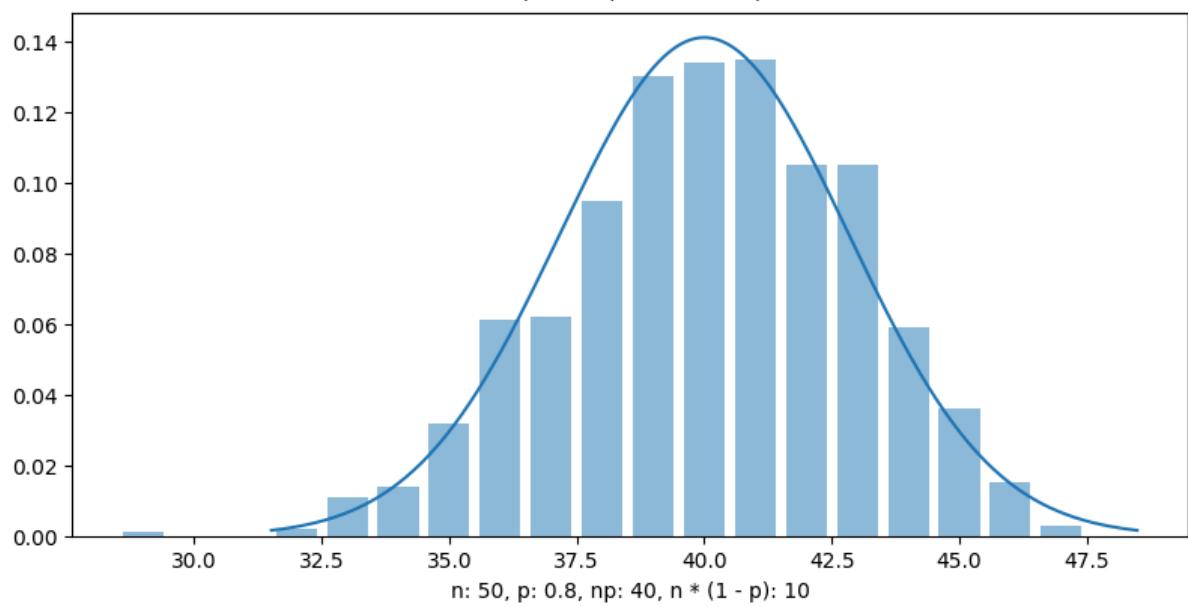
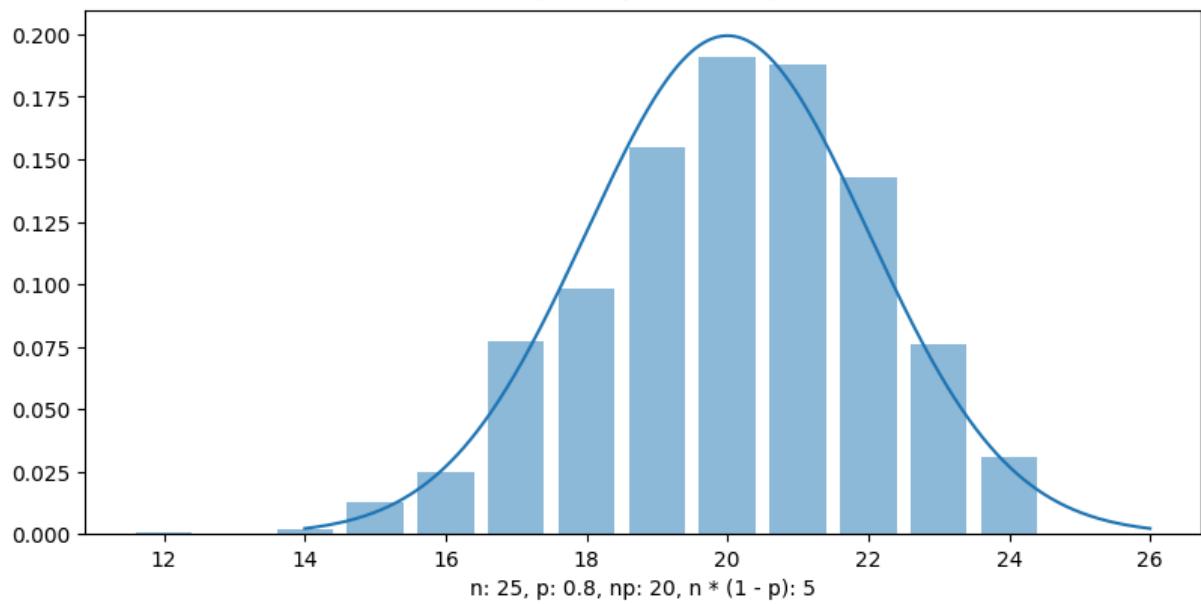
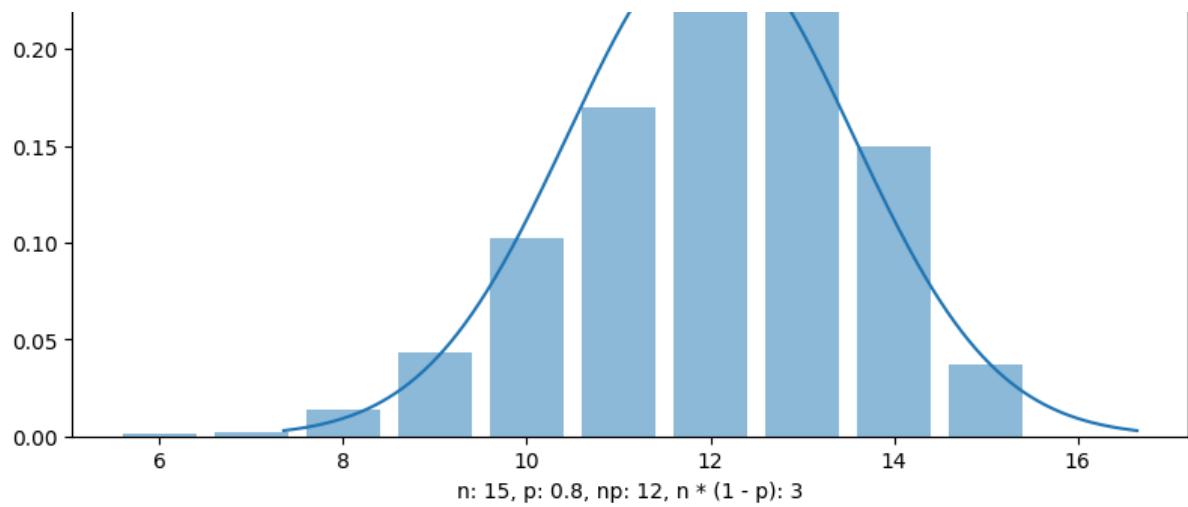
    # с помощью столбчатой диаграммы выведем относительную частоту каждого значе-
    ax[i].bar(successes, counts / len(res), alpha = 0.5)
    # в подписи к графикам выведем n, p, а также результат проверки условий
    ax[i].set_xlabel('n: {}, p: {}, np: {}, n * (1 - p): {}'.format(n, p, np, cond1))

    # рассчитаем среднее значение и СКО согласно теореме Муавра-Лапласа
    mean, std = n * p, np.sqrt(n * p * (1 - p))
    # создадим 1000 точек в +/- 3 СКО от среднего значения
    x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)
    # найдем точки на оси y по формуле Гаусса
    f_norm = norm.pdf(x, mean, std)
    # построим график плотности нормального распределения
    ax[i].plot(x, f_norm)

    # дополнительно можно вывести "идеальное" биномиальное распределение с помо-
    # from scipy.stats import binom
    # x2 = np.arange(n)
    # binom_p = binom.pmf(x2, n, p)
    # ax[i].stem(x2, binom_p, use_line_collection = True)

plt.show()
```





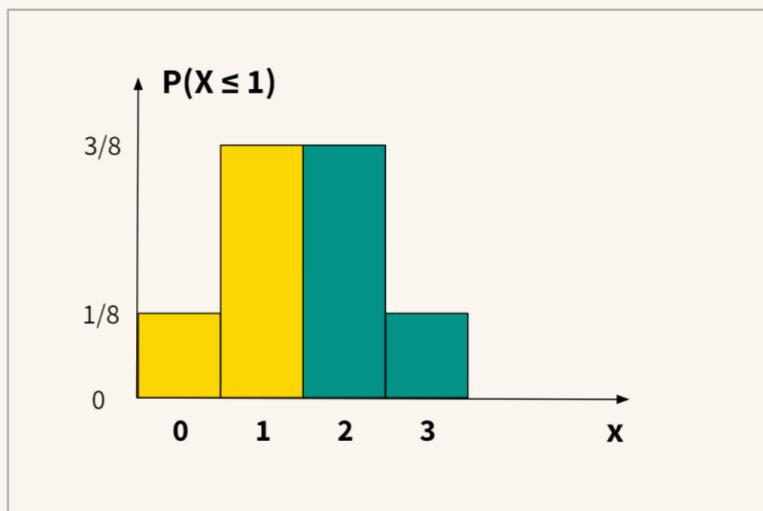
Поправка на непрерывность распределения

Как мы видим, вначале (например, при $n = 3$), распределение было ожидаемо скошенным, при этом по мере увеличения количества бросков оно стало все больше «вписываться» в график плотности нормального распределения.

Поправка на непрерывность распределения

Нормальное приближение биномиального распределения удобно использовать, когда нужно посчитать вероятность большого числа исходов. Однако прежде внесем одно уточнение.

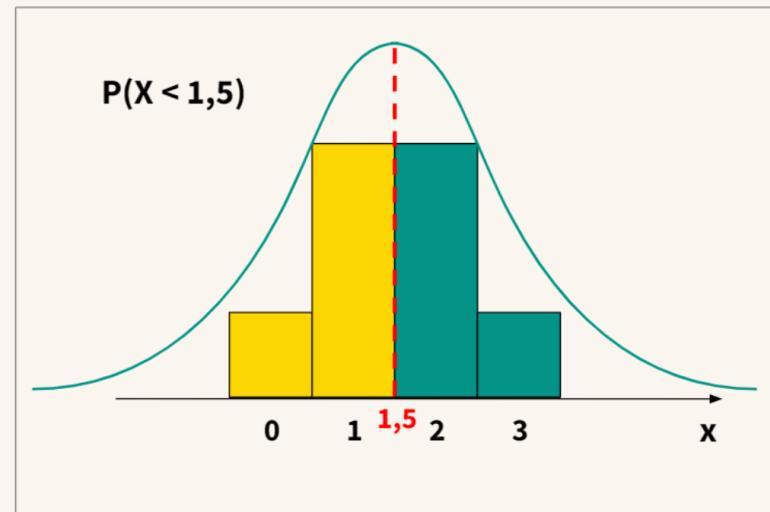
На графике ниже видно, что для расчета биномиальной вероятности нам нужно сложить площади столбцов гистограммы. Например, чтобы найти вероятность выпадения не более одного орла при трех последовательных подбрасываниях монеты, нужно сложить первый и второй столбцы.



Получаем

$$P_B(X \leq 1) = P_B(X = 0) + P_B(X = 1) = \frac{1}{8} + \frac{3}{8} = \frac{4}{8} = \frac{1}{2}$$

При этом, если мы хотим рассчитать эту же площадь с помощью кривой нормального распределения, нам нужно сместить границу таким образом, чтобы захватить все интересующие нас столбцы. В данном случае прибавить 0,5. То есть $P_N(X < 1,5)$.



Воспользуемся теоремой Муавра-Лапласа для расчета среднего и СКО (напомню $n = 3, p = 0,5$).

$$\mu = np = 3 \times 0,5 = 1,5$$

$$\sigma = \sqrt{np(1-p)} = \sqrt{3 \times 0,5 \times 0,5} = 0,75$$

Остается воспользоваться Питоном для нахождения площади под кривой нормального распределения.

$$B(3; 0,5) \sim N(1,5; 0,75)$$

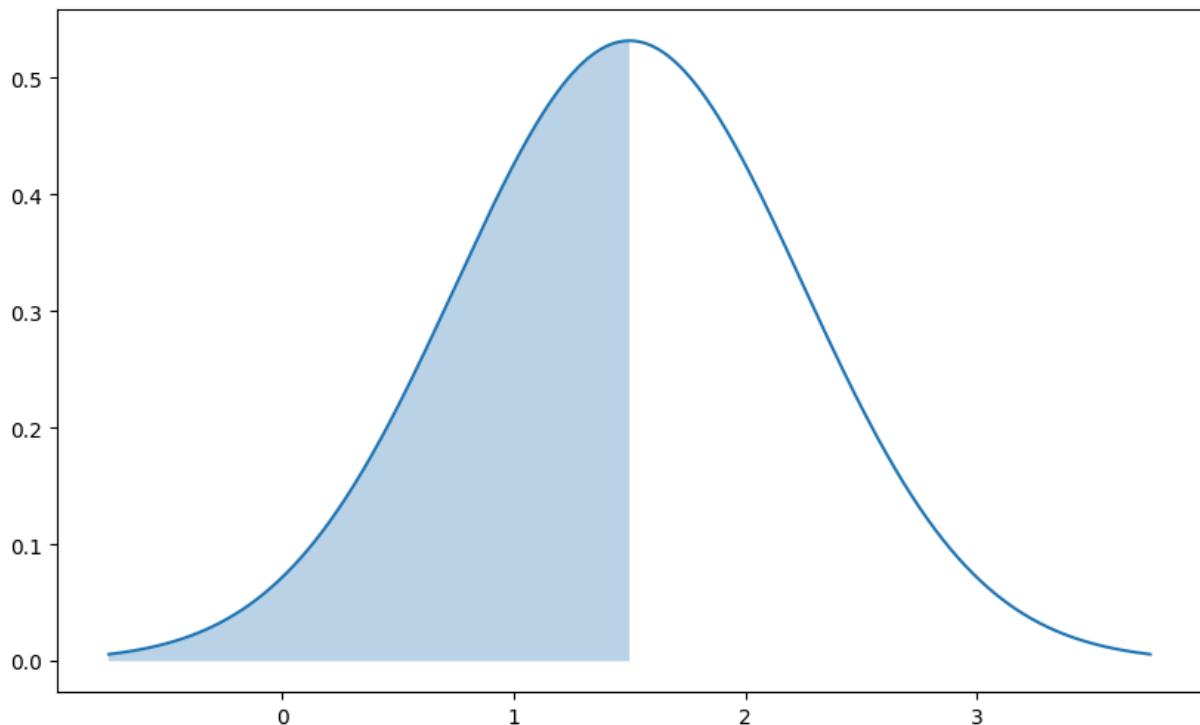
$$P_B(X \leq 1) \sim P_N(X < 1,5)$$

```
In [ ]: # передадим методу .cdf() границу, среднее значение (loc) и СКО (scale)
area = norm.cdf(1.5, loc = 1.5, scale = 0.75)

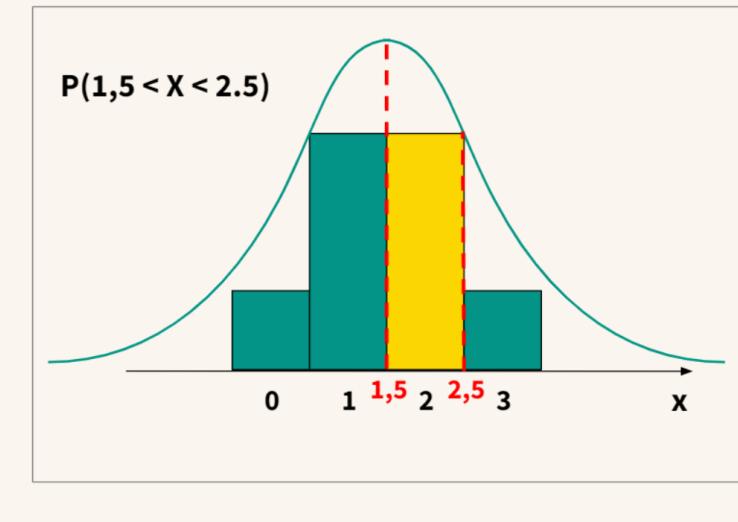
# на выходе мы получим площадь под кривой
area
```

Out[]: 0.5

```
In [ ]: # зададим размер графика  
plt.figure(figsize = (10,6))  
  
# пропишем среднее значение и СКО  
mean, std = 1.5, 0.75  
  
# создадим пространство из 1000 точек в диапазоне +/- трех СКО от среднего э  
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)  
  
# рассчитаем значения по оси у с помощью метода .pdf()  
# т.е. функции плотности вероятности  
f = norm.pdf(x, mean, std)  
  
# и построим график  
plt.plot(x, f)  
  
# дополнительно создадим точки на оси x для закрашенной области  
px = np.linspace(mean - 3 * std, mean, 1000)  
  
# и заполним в пределах этих точек по оси x пространство  
# от кривой нормального распределения до оси у = 0  
plt.fill_between(px, norm.pdf(px, mean, std), alpha = 0.3)  
  
# выведем оба графика на экран  
plt.show()
```



Если же нас попросят найти вероятность конкретного значения, например, выпадения двух орлов $P_B(X = 2)$ (то есть площадь одного столбца гистограммы), то при расчете площади под кривой нормального распределения нужен интервал $P_N(1,5 < X < 2,5)$. Другими словами мы прибавили по 0,5 с обеих сторон.



$$P_B(X = 2) \sim P_N(1,5 < X < 2,5)$$

```
In [ ]: # передадим методу .cdf() границу, среднее значение (loc) и СКО (scale)
area = norm.cdf(2.5, loc = 1.5, scale = 0.75) - norm.cdf(1.5, loc = 1.5, sca
# на выходе мы получим площадь под кривой
area
```

Out[]: 0.4087887802741321

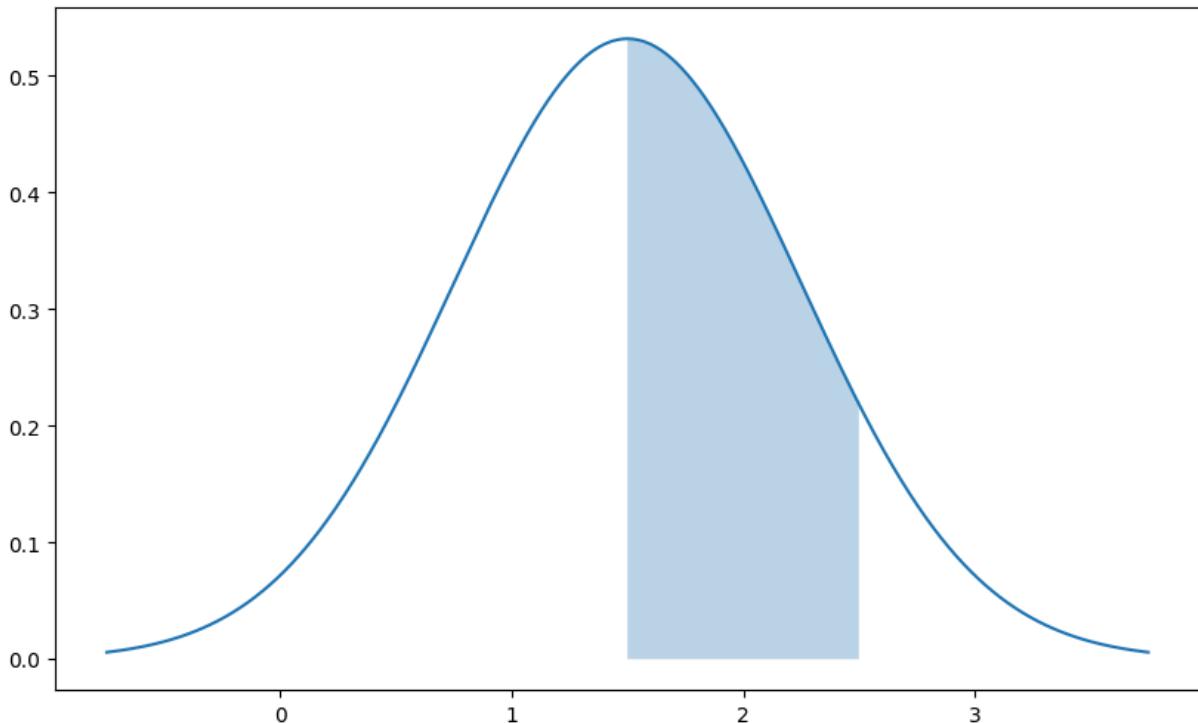
Напомню, площадь столбца составляет $P_B(X = 2) = \frac{3}{8} = 0,375$, что довольно существенно отличается от площади под кривой. Это связано с тем, что мы взяли слишком маленькое n и условия $np \geq 5$ и $n(1 - p) \geq 5$ в данном случае не выполняются.

По мере увеличения n поправка на непрерывность распределения становится все менее значимой.

```
In [ ]: # рассчитаем биномиальную вероятность P(X = 2)
3/8
```

Out[]: 0.375

```
In [ ]: # зададим размер графика  
plt.figure(figsize = (10,6))  
  
# пропишем среднее значение и СКО  
mean, std = 1.5, 0.75  
  
# создадим пространство из 1000 точек в диапазоне +/- трех СКО от среднего э  
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)  
  
# рассчитаем значения по оси у с помощью метода .pdf()  
# т.е. функции плотности вероятности  
f = norm.pdf(x, mean, std)  
  
# и построим график  
plt.plot(x, f)  
  
# дополнительно создадим точки на оси x для закрашенной области  
px = x = np.linspace(1.5, 2.5, 1000)  
  
# и заполним в пределах этих точек по оси x пространство  
# от кривой нормального распределения до оси у = 0  
plt.fill_between(px, norm.pdf(px, mean, std), alpha = 0.3)  
  
# выведем оба графика на экран  
plt.show()
```



Пример приближения

Пример приближения

Для закрепления пройденного материала рассмотрим более жизненный пример. Предположим, вам поставили партию из 500 единиц оборудования. При этом вам известно, что в среднем 2% (то есть 0,02) оборудования имеют различные дефекты. Какова вероятность того, что в партии не менее 15 бракованных единиц оборудования?

Речь идет о биномиальном распределении, так как мы последовательно достаем из партии одну единицу оборудования, и в ней либо будет дефект, либо нет.

$$B(n, p) = B(500; 0,02) \rightarrow P_B(X \geq 15)$$

$$B(n, p) = B(500; 0,02) \rightarrow P_B(X \geq 15)$$

```
In [ ]: # импортируем объект binom из модуля stats библиотеки scipy
from scipy.stats import binom

# зададим параметры биномиального распределения
n, p = 500, 0.02

# метод .cdf() рассчитывает вероятность до значения включительно (!),
# таким образом, чтобы найти X >= 15,
# мы берем значения вплоть до 14 и вычитаем результат из единицы
1 - binom.cdf(14, n, p)
```

Out[]: 0.08135689951689384

Теперь решим эту задачу с помощью нормального приближения. По теореме Муавра-Лапласа выводим следующее.

$$B(500; 0,02) \sim \mathcal{N}(10; \sqrt{9,8}) \rightarrow P_N(X > 14,5)$$

Таким образом, задача сводится к нахождению площади под кривой с учетом внесенной поправки на непрерывность распределения ($15 - 0,5 = 14,5$).

$$B(500; 0,02) \sim \mathcal{N}(10; \sqrt{9,8}) \rightarrow P_N(X > 14,5)$$

```
In [ ]: # рассчитаем среднее значение и СКО
mean, std = n * p, np.sqrt(n * p * (1 - p))

# рассчитаем площадь под кривой слева от границы и вычтем результат из единицы
1 - norm.cdf(14.5, loc = mean, scale = std)
```

```
Out[ ]: 0.07529192313308641
```

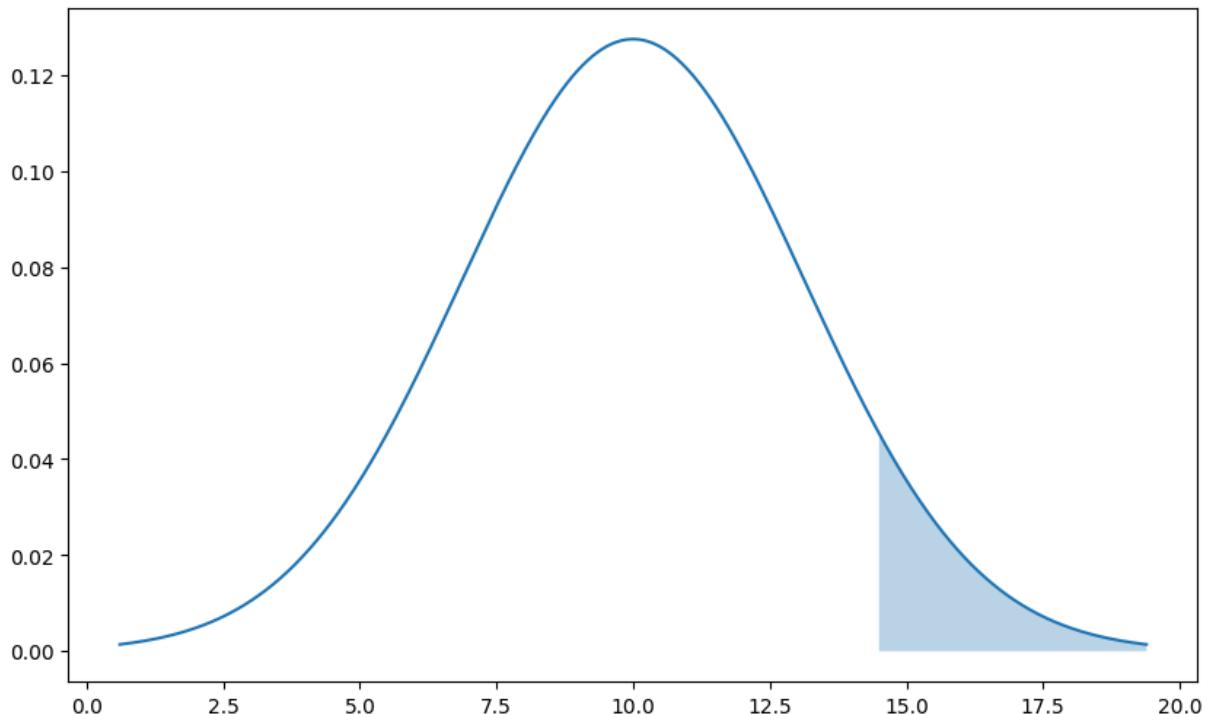
```
In [ ]: # зададим размер графика
plt.figure(figsize = (10,6))

# создадим 1000 точек в +/- 3 СКО от среднего значения
x = np.linspace(mean - 3 * std, mean + 3 * std, 1000)
# найдем точки на оси у по формуле Гаусса
f_norm = norm.pdf(x, mean, std)
# построим график плотности нормального распределения
plt.plot(x, f_norm)

# дополнительно создадим точки на оси x для закрашенной области
px = np.linspace(14.5, mean + 3 * std, 1000)

# и заполним в пределах этих точек по оси x пространство
# от кривой нормального распределения до оси у = 0
plt.fill_between(px, norm.pdf(px, mean, std), alpha = 0.3)

plt.show()
```



Количественная случайная величина	
Дискретная величина	Непрерывная величина
Распределение величины задается функцией вероятности (probability mass function, pmf)	Распределение величины задается плотностью вероятности (probability density function, pdf)
Функция распределения ((cumulative) distribution function, cdf) показывает, с какой вероятностью случайная величина примет значение, меньшее или равное данному	
Примеры распределений: <ul style="list-style-type: none"> - равномерное дискретное распределение <code>np.random.randint()</code> - распределение Бернулли <code>np.random.binomial(n = 1)</code>¹ - биномиальное распределение <code>np.random.binomial()</code> - ... 	Примеры распределений: <ul style="list-style-type: none"> - равномерное непрерывное распределение <code>np.random.uniform()</code>, <code>np.random.random()</code>, <code>np.random.rand()</code> - нормальное распределение <code>np.random.normal()</code>² - ...

¹ Напомню, что распределение Бернулли можно получить через функцию биномиального распределения, если указать, что будет проводиться только $n = 1$ испытаний.

² Модуль `random` генерирует псевдослучайные числа (как если бы мы собирали данные в реальной жизни). Для создания теоретического, "идеального" распределения используйте модуль `stats` библиотеки `SciPy`.

Комбинаторика в Питоне

`np.random.shuffle()` и `np.random.permutation()`

```
In [ ]: # создадим массив и передадим его в функцию np.random.shuffle()
arr = np.array([1, 2, 3, 4, 5])
```

```
# сама функция выдала None, исходный массив при этом изменился
print(np.random.shuffle(arr), arr)
```

None [3 5 4 1 2]

```
In [ ]: # еще раз создадим массив
arr = np.array([1, 2, 3, 4, 5])

# передав его в np.random.permutation(),
# мы получим перемешанную копию и исходный массив без изменений
np.random.permutation(arr), arr
```

Out[]: (array([1, 2, 3, 4]), array([1, 2, 3, 4, 5]))

Функция для разделения на обучающую и тестовую выборки

```
In [ ]: # объявим функцию, которая принимает признаки X и целевую переменную y в формате массива NumPy или датафрейма
# дополнительно пропишем размер тестовой выборки с параметром по умолчанию 6
# и возможностью задать точку отсчета
def split_data(X, y, test_size = 0.3, random_state = None):
```

```

# проверим, является ли X массивом NumPy с помощью функции isinstance()
if isinstance(X, np.ndarray):
    # если да, превратим в датафрейм
    X = pd.DataFrame(X)
# сделаем то же самое для у
if isinstance(y, np.ndarray):
    y = pd.DataFrame(y)

# еще один способ выполнить такую проверку
# if type(X). __module__ == np. __name__:
#     X = pd.DataFrame(X)
# if type(y). __module__ == np. __name__:
#     y = pd.DataFrame(y)

# установим точку отсчета
np.random.seed(random_state)

# перемешаем индексы строк датасета X, не изменяя исходный массив
indices = np.random.permutation(len(X))

# определим количество строк, которые войдут в тестовую выборку
# для этого умножим количество строк в X на долю тестовой выборки
data_test_size = int(X.shape[0] * test_size)

# начиная с этого количества (границы), будет обучающая выборка
train_indices = indices[data_test_size:]
# перед ним, тестовая
test_indices = indices[:data_test_size]

# с помощью метода .iloc() найдем в X все строки,
# соответствующие индексам в переменных train_indices и test_indices
X_train = X.iloc[train_indices]
X_test = X.iloc[test_indices]

# сделаем то же самое для у
y_train = y.iloc[train_indices]
y_test = y.iloc[test_indices]

# выведем выборки в том порядке, в котором они выводятся в sklearn
return X_train, X_test, y_train, y_test

```

```

In [ ]: import pandas as pd

# импортируем данные о пациентах с диабетом
from sklearn.datasets import load_diabetes

# помещаем их в переменную data
data = load_diabetes()

# создаем два датафрейма
X = pd.DataFrame(data.data, columns = data.feature_names)
y = pd.DataFrame(data.target, columns = ['target'])

# для проверки работы функции можно также создать массивы NumPy

```

```
# X = data.data  
# y = data.target
```

```
In [ ]: # вызовем функцию split_data()  
X_train, X_test, y_train, y_test = split_data(X, y, random_state = 42)
```

```
In [ ]: # посмотрим на индексы строк в переменной X_train  
X_train.head(3)
```

```
Out[ ]:      age      sex      bmi      bp      s1      s2      s3  
108  0.019913  0.050680  0.045529  0.029894 -0.062111 -0.055802 -0.072854  
225  0.030811  0.050680  0.032595  0.049415 -0.040096 -0.043589 -0.069172  
412  0.074401 -0.044642  0.085408  0.063187  0.014942  0.013091  0.015505
```

```
In [ ]: # теперь посмотрим на y_train  
y_train.head(3)
```

```
Out[ ]:      target  
108    232.0  
225    208.0  
412    261.0
```

Модуль itertools

Перестановки

Перестановки без замены

1. Перестановки без повторений

```
In [ ]: # импортируем модуль math  
import math  
  
# передадим функции factorial() число 3  
math.factorial(3)
```

```
Out[ ]: 6
```

```
In [ ]: # импортируем модуль itertools  
import itertools  
  
# создадим строку из букв A, B, C  
x = 'ABC'  
# помимо строки можно использовать и список  
# x = ['A', 'B', 'C']
```

```
# и передадим ее в функцию permutations()
# так как функция возвращает объект itertools.permutations,
# для вывода результата используем функцию list()
list(itertools.permutations(x))
```

```
Out[ ]: [('A', 'B', 'C'),
          ('A', 'C', 'B'),
          ('B', 'A', 'C'),
          ('B', 'C', 'A'),
          ('C', 'A', 'B'),
          ('C', 'B', 'A')]
```

```
In [ ]: # чтобы узнать количество перестановок, можно использовать функцию len()
len(list(itertools.permutations(x)))
```

```
Out[ ]: 6
```

```
In [ ]: # теперь элементов исходного множества шесть
x = 'ABCDEF'

# чтобы узнать, сколькими способами их можно разместить на трех местах,
# передадим параметр r = 3 и выведем первые пять элементов
list(itertools.permutations(x, r = 3))[:5]
```

```
Out[ ]: [('A', 'B', 'C'),
          ('A', 'B', 'D'),
          ('A', 'B', 'E'),
          ('A', 'B', 'F'),
          ('A', 'C', 'B')]
```

```
In [ ]: # посмотрим на общее количество таких перестановок
len(list(itertools.permutations(x, r = 3)))
```

```
Out[ ]: 120
```

2. Перестановки с повторениями

```
In [ ]: # импортируем необходимые библиотеки
import itertools
import numpy as np
import math

# объявим функцию permutations_w_repetition(), которая будет принимать два г
# x - строка, список или массив Numpy
# r - количество мест в перестановке, по умолчанию равно количеству элементов
def permutations_w_repetition(x, r = len(x)):

    # если передается строка,
    if isinstance(x, str):
        # превращаем ее в список
        x = list(x)

    # в числителе рассчитаем количество перестановок без повторений
    numerator = len(list(itertools.permutations(x, r = r)))
```

```
# для того чтобы рассчитать знаменатель найдем,
# сколько раз повторяется каждый из элементов
_, counts = np.unique(x, return_counts = True)

# объявим переменную для знаменателя
denominator = 1

# и в цикле будем помещать туда произведение факториалов
# повторяющихся элементов
for c in counts:

    # для этого проверим повторяется ли элемент
    if c > 1:

        # если да, умножим знаменатель на факториал повторяющегося элемента
        denominator *= math.factorial(c)

    # разделим числитель на знаменатель
    # деление дает тип float, поэтому используем функцию int(),
    # чтобы результат был целым числом
return int(numerator/denominator)
```

```
In [ ]: # создадим строку со словом "молоко"
x = 'МОЛОКО'

# вызовем функцию
permutations_w_repetition(x)
```

```
Out[ ]: 120
```

Перестановки с заменой

```
In [ ]: # посмотрим, сколькими способами можно выбрать два сорта мороженого
list(itertools.product(['Ваниль', 'Клубника'], repeat = 2))
```

```
Out[ ]: [('Ваниль', 'Ваниль'),
          ('Ваниль', 'Клубника'),
          ('Клубника', 'Ваниль'),
          ('Клубника', 'Клубника')]
```

```
In [ ]: # посмотрим на способы переставить с заменой два элемента из четырех
list(itertools.product('ABCD', repeat = 2))
```

```
Out[ ]: [('A', 'A'),  
         ('A', 'B'),  
         ('A', 'C'),  
         ('A', 'D'),  
         ('B', 'A'),  
         ('B', 'B'),  
         ('B', 'C'),  
         ('B', 'D'),  
         ('C', 'A'),  
         ('C', 'B'),  
         ('C', 'C'),  
         ('C', 'D'),  
         ('D', 'A'),  
         ('D', 'B'),  
         ('D', 'C'),  
         ('D', 'D')]
```

```
In [ ]: # убедимся, что таких способов 16  
len(list(itertools.product('ABCD', repeat = 2)))
```

```
Out[ ]: 16
```

Сочетания

```
In [ ]: # возьмем пять элементов  
x = 'ABCDE'  
  
# и найдем способ переставить два элемента из этих пяти  
list(itertools.permutations(x, r = 2))
```

```
Out[ ]: [('A', 'B'),  
         ('A', 'C'),  
         ('A', 'D'),  
         ('A', 'E'),  
         ('B', 'A'),  
         ('B', 'C'),  
         ('B', 'D'),  
         ('B', 'E'),  
         ('C', 'A'),  
         ('C', 'B'),  
         ('C', 'D'),  
         ('C', 'E'),  
         ('D', 'A'),  
         ('D', 'B'),  
         ('D', 'C'),  
         ('D', 'E'),  
         ('E', 'A'),  
         ('E', 'B'),  
         ('E', 'C'),  
         ('E', 'D')]
```

```
In [ ]: # уменьшим на количество перестановок каждого типа r!  
int(len(list(itertools.permutations(x, r = 2)))/math.factorial(2))
```

```
Out[ ]: 10
```

```
In [ ]: # то же самое можно рассчитать с помощью функции combinations()
list(itertools.combinations(x, 2))
```

```
Out[ ]: [('A', 'B'),
          ('A', 'C'),
          ('A', 'D'),
          ('A', 'E'),
          ('B', 'C'),
          ('B', 'D'),
          ('B', 'E'),
          ('C', 'D'),
          ('C', 'E'),
          ('D', 'E')]
```

```
In [ ]: # посмотрим на количество сочетаний
len(list(itertools.combinations(x, 2)))
```

```
Out[ ]: 10
```

Сочетания с заменой

```
In [ ]: # сколькими способами с заменой можно выбрать два элемента из двух
list(itertools.combinations_with_replacement('AB', 2))
```

```
Out[ ]: [('A', 'A'), ('A', 'B'), ('B', 'B')]
```

```
In [ ]: # очевидно, что без замены есть только один такой способ
list(itertools.combinations('AB', 2))
```

```
Out[ ]: [('A', 'B')]
```

Биномиальные коэффициенты

```
In [ ]: # дерево вероятностей можно построить с помощью декартовой степени
list(itertools.product('HT', repeat = 3))
```

```
Out[ ]: [('H', 'H', 'H'),
          ('H', 'H', 'T'),
          ('H', 'T', 'H'),
          ('H', 'T', 'T'),
          ('T', 'H', 'H'),
          ('T', 'H', 'T'),
          ('T', 'T', 'H'),
          ('T', 'T', 'T')]
```

```
In [ ]: # посмотрим, в скольких комбинациях выпадет две решки при трех бросках
comb = len(list(itertools.combinations('ABC', 2)))
comb
```

```
Out[ ]: 3
```

```
In [ ]: # вычислим вероятность выпадения двух орлов в трех бросках
# при вероятности выпадения орла 0,7
```

```
round(comb * (0.7 ** 2 * (1 - 0.7)** (3 - 2)), 3)
```

```
Out[ ]: 0.441
```

```
In [ ]: from scipy.stats import binom
```

```
# то же самое можно вычислить с помощью функции вероятности (probability mass function)
# биномиального распределения библиотеки scipy
binom.pmf(k = 2, n = 3, p = 0.7).round(3)
```

```
Out[ ]: 0.441
```

Дополнительные примеры

Матожадание и среднее значение

Математическое ожидание

```
In [ ]: # с помощью библиотеки scipy
from scipy.stats import binom

# для биномиального распределения со следующими параметрами
n, p = 3, 0.7

# рассчитаем матожидание и ожидаемую дисперсию
expected_value, variance = binom.stats(n, p, moments='mv')
expected_value, variance
```

```
Out[ ]: (array(2.1), array(0.63))
```

Фактическое среднее значение

```
In [ ]: # теперь с помощью модуля random
import numpy as np

# проведем миллион биномиальных экспериментов
res = np.random.binomial(n = 3, p = 0.7, size = 1000000)

# и посмотрим на фактическое среднее значение и фактическую дисперсию
np.mean(res), np.var(res)
```

```
Out[ ]: (2.100117, 0.6288295863109998)
```

```
In [ ]:
```