Лабораторные

1. Лабораторная работа №1. Виртуальное окружение (Virtualenv). Установка и запуск Jupyter Notebook.

   Виртуальное окружение (Virtualenv). Установка и запуск Jupyter Notebook Чтобы поставить сторонний пакет на операционную систему, пользуются утилитой pip. В командной строке можно набрать pip install и название сторонней библиотеки, которую вам хочется поставить. Хорошей практикой при программировании на Python является использование виртуальных окружений. Виртуальное окружение в Python --- это окружение, которое позволяет изолировать зависимости для определённого проекта. Обычно для каждого проекта создаётся своё виртуальное окружение, в которое ставятся все пакеты, используемые в данном проекте. Чтобы создать виртуальное окружение, воспользуемся модулем venv. Синтаксис: python3 −m venv и далее название директории, в которой будет создано новое виртуальное окружение. Обычно папка с виртуальным окружением создаётся рядом с папкой проекта. Виртуальное окружение активируется следующей командой: source /bin/activate После этого в виртуальное окружение можно устанавливать пакеты известной нам командой pip install. Например, поставим известное приложение Jupyter Notebook. Jupyter Notebook --- очень распространённое приложение, которое применяется повсеместно разработчиками на Python, а также учёными для презентации своей работы и интерактивного запуска кода. pip install jupyter Вместе с jupyter установится полезный пакет ipython --- расширенная версия интерактивного интерпретатора Python, которая может служить его альтернативой. Чтобы запустить Jupyter Notebook, воспользуйтесь командой jupyter-notebook. Это приложение запускается в веб-браузере. Внутри него можно создавать "ноутбуки", где в ячейках можно писать код и интерактивно его исполнять.

2. Лабораторная работа №2. Множества.

   Множество в языке Питон — это структура данных, эквивалентная множествам в математике. Множество может состоять из различных элементов, порядок элементов в множестве неопределен. В множество можно добавлять и удалять элементы, можно перебирать элементы множества, можно выполнять операции над множествами

(объединение, пересечение, разность). Можно проверять принадлежность элемента множеству.

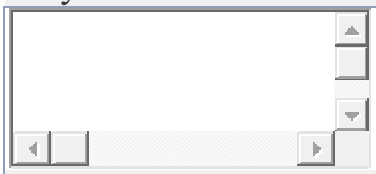
В отличие от массивов, где элементы хранятся в виде последовательного списка, в множествах порядок хранения элементов неопределен (более того, элементы множества хранятся не подряд, как в списке, а при помощи хитрых алгоритмов). Это позволяет выполнять операции типа “проверить принадлежность элемента множеству” быстрее, чем просто перебирая все элементы множества.

Элементами множества может быть любой неизменяемый тип данных: числа, строки, кортежи. Изменяемые типы данных не могут быть элементами множества, в частности, нельзя сделать элементом множества список (но можно сделать кортеж) или другое множество. Требование неизменяемости элементов множества накладывается особенностями представления множества в памяти компьютера.

Задание множеств

Множество задается перечислением всех его элементов в фигурных скобках. Исключением является пустое множество, которое можно создать при помощи функции set(). Если функции set передать в качестве параметра список, строку или кортеж, то она вернёт множество, составленное из элементов списка, строки, кортежа. Например:

запустить выполнить пошагово



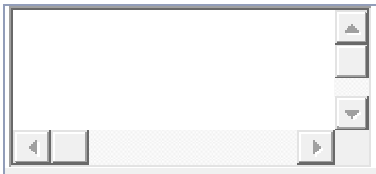
```
1
2
3
4
```

```
A = {1, 2, 3}
A = set('qwerty')
print(A)
```

выведет {'e', 'q', 'r', 't', 'w', 'y'}.

Каждый элемент может входить в множество только один раз, порядок задания элементов неважен. Например, программа:

запустить выполнить пошагово



```
A = {1, 2, 3}
B = {3, 2, 3, 1}
print(A == B)
```

выведет True, так как A и B — равные множества.

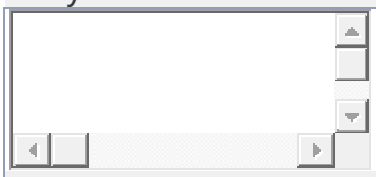
Каждый элемент может входить в множество только один раз. set('Hello') вернет множество из четырех элементов: {'H', 'e', 'l', 'o'}.

Работа с элементами множеств

Узнать число элементов в множестве можно при помощи функции len.

Перебрать все элементы множества (в неопределенном порядке!) можно при помощи цикла for:

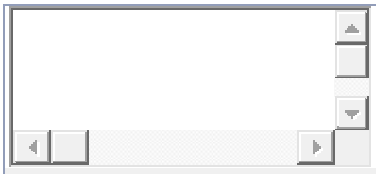
запустить выполнить пошагово



```
primes = {2, 3, 5, 7, 11}
for num in primes:
    print(num)
```

Проверить, принадлежит ли элемент множеству можно при помощи операции in, возвращающей значение типа bool. Аналогично есть противоположная операция not in. Для добавления элемента в множество есть метод add:

запустить выполнить пошагово



```
1
2
3
4
```

```
A = {1, 2, 3}
print(1 in A, 4 not in A)
A.add(4)
```

Для удаления элемента x из множества есть два метода: discard и remove. Их поведение различается только в случае, когда удаляемый элемент отсутствует в множестве. В этом случае метод discard не делает ничего, а метод remove генерирует исключение KeyError.

Наконец, метод pop удаляет из множества один случайный элемент и возвращает его значение. Если же множество пусто, то генерируется исключение KeyError.

Из множества можно сделать список при помощи функции list.

Операции с множествами

С множествами в питоне можно выполнять обычные для математики операции над множествами.

A | B **A.union(B)**	Возвращает множество, являющееся объединением множеств A и B.
A |= B **A.update(B)**	Добавляет в множество A все элементы из множества B.
A & B **A.intersection(B)**	Возвращает множество, являющееся пересечением множеств A и B.
A &= B **A.intersection_update(B)**	Оставляет в множестве A только те элементы, которые есть в множестве B.
A - B **A.difference(B)**	Возвращает разность множеств A и B (элементы, входящие в A, но не входящие в B).

A -= B **A.difference_update(B)**	Удаляет из множества `A` все элементы, входящие в `B`.
A ^ B **A.symmetric_difference(B)**	Возвращает симметрическую разность множеств `A` и `B` (элементы, входящие в `A` или в `B`, но не в оба из них одновременно).
A ^= B **A.symmetric_difference_update(B)**	Записывает в `A` симметрическую разность множеств `A` и `B`.
A <= B **A.issubset(B)**	Возвращает `true`, если `A` является подмножеством `B`.
A >= B **A.issuperset(B)**	Возвращает `true`, если `B` является подмножеством `A`.
A < B	Эквивалентно `A <= B and A != B`
A > B	Эквивалентно `A >= B and A != B`

3. Лабораторная работа №3. Декораторы.

Декоратор --- это функция, которая принимает функцию и возвращает функцию. И ничего более. Например, простейший декоратор принимает функции и возвращает её же:

```
def decorator(func):
    return func


@decorator # синтаксис декоратора
def decorated():
    print('Hello!')
```

Выражение с @ --- всего лишь синтаксический сахар. Мы можем написать то же самое без него: decorated = decorator(decorated). Вспомнив это, можно смело переходить к декораторам. **Декораторы** — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

Создадим свой декоратор "вручную":

```
>>>
```

```
>>> def my_shiny_new_decorator(function_to_decorate):
```

```
...     # Внутри себя декоратор определяет функцию-
"обёртку". Она будет обёрнута вокруг декорируемой,

...     # получая возможность исполнять произвольный
код до и после неё.

...     def the_wrapper_around_the_original_function():

...         print("Я - код, который отработает до
вызова функции")

...         function_to_decorate() # Сама функция

...         print("А я - код, срабатывающий после")

...     # Вернём эту функцию

...     return the_wrapper_around_the_original_function

...

>>> # Представим теперь, что у нас есть функция,
которую мы не планируем больше трогать.

>>> def stand_alone_function():

...     print("Я простая одинокая функция, ты ведь не
посмеешь меня изменять?")

...

>>> stand_alone_function()

Я простая одинокая функция, ты ведь не посмеешь меня
изменять?

>>> # Однако, чтобы изменить её поведение, мы можем
декорировать её, то есть просто передать декоратору,
```

```
>>> # который обернет исходную функцию в любой код,
который нам потребуется, и вернёт новую,

>>> # готовую к использованию функцию:

>>> stand_alone_function_decorated =
my_shiny_new_decorator(stand_alone_function)

>>> stand_alone_function_decorated()

Я - код, который отработает до вызова функции

Я простая одинокая функция, ты ведь не посмеешь меня
изменять?

А я - код, срабатывающий после
```

Наверное, теперь мы бы хотели, чтобы каждый раз, во время вызова stand_alone_function, вместо неё вызывалась stand_alone_function_decorated. Для этого просто перезапишем stand_alone_function:

```
>>>
```

```
>>> stand_alone_function =
my_shiny_new_decorator(stand_alone_function)

>>> stand_alone_function()

Я - код, который отработает до вызова функции

Я простая одинокая функция, ты ведь не посмеешь меня
изменять?

А я - код, срабатывающий после
```

Собственно, это и есть декораторы. Вот так можно было записать предыдущий пример, используя синтаксис декораторов:

```
>>>
```

```
>>> @my_shiny_new_decorator

... def another_stand_alone_function():

...     print("Оставь меня в покое")

...

>>> another_stand_alone_function()

Я - код, который отработает до вызова функции

Оставь меня в покое

А я - код, срабатывающий после
```

То есть, декораторы в python — это просто синтаксический сахар для конструкций вида:

```
another_stand_alone_function =
my_shiny_new_decorator(another_stand_alone_function)
```

4. Лабораторная работа №4. Классы и объекты.

Класс как шаблон для создания объектов

На самом деле классы – не модули. Они своего рода шаблоны, от которых создаются объекты-экземпляры. Такие объекты наследуют от класса его атрибуты. Вернемся к нашему классу *B* и создадим на его основе два объекта:

```
>>> class B:
...     n = 5
...     def adder(v):
...             return v + B.n
...
>>> a = B()
```

```
>>> b = B()
```

У объектов, связанных с переменными *a* и *b*, нет собственного поля *n*. Однако они наследуют его от своего класса:

```
>>> a.n
5
>>> a.n is B.n
True
```

То есть поля `a.n` и `B.n` – это одно и то же поле, к которому можно обращаться и через имя *a*, и через имя *b*, и через имя класса. Поле одно, ссылок на него три.

Однако что произойдет в момент присваивания этому полю значения через какой-нибудь объект-экземпляр?

```
>>> a.n = 10
>>> a.n
10
>>> b.n
5
>>> B.n
5
```

В этот момент у экземпляра появляется собственный атрибут *n*, который перекроет (переопределит) родительский, то есть тот, который достался от класса.

```
>>> a.n is B.n
False
>>> b.n is B.n
True
```

При этом присвоение через `B.n` отразится только на *b* и *B*, но не на *a*:

```
>>> B.n = 100
>>> B.n, b.n, a.n
(100, 100, 10)
```

Иная ситуация нас ожидает с атрибутом *adder*. При создании объекта от класса функция *adder* не наследуется как есть, а как бы превращается для объекта в одноименный метод:

```
>>> B.adder is b.adder
False
>>> type(B.adder)
<class 'function'>
>>> type(b.adder)
<class 'method'>
```

Через имя класса мы вызываем функцию *adder*:

```
>>> B.adder(33)
133
```

Через имя объекта вызываем метод *adder*:

```
>>> b.adder(33)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: adder() takes 1 positional
argument but 2 were given
```

В сообщении об ошибке говорится, что *adder* принимает только один аргумент, а было передано два. Откуда появился второй, если в скобках было указано только одно число?

Дело в том, что в отличии от функции **в метод первым аргументом всегда передается объект, к которому применяется этот метод**. То есть выражение `b.adder(33)` как бы преобразовывается в `adder(b, 33)`. Сам же `b.adder` как объект типа **method** хранит сведения, с каким классом он связан и какому объекту-экземпляру принадлежит:

```
>>> b.adder
<bound method B.adder of
<__main__.B object at 0x7fcbf1ab9b80>>
```

В нашем случае, чтобы вызывать *adder* через объекты-экземпляры, класс можно переписать так:

```
>>> class B:
```

```
...     n = 5
...     def adder(obj, v):
...             return v + obj.n
...
>>> b = B()
>>> b.adder(33)
38
```

В коде выше при вызове метода *adder* переменной-параметру *obj* присваивается объект, связанный с переменной, к которой применяется данный метод. В данном случае это объект, связанный с *b*. Если *adder* будет вызван на другой объект, то уже он присвоится *obj*:

```
>>> a = B()
>>> a.n = 9
>>> a.adder(3)
12
```

В Python принято переменную-параметр метода, которая связывается с экземпляром своего класса, называть именем **self**. Таким образом, более корректный код будет таким:

```
>>> class B:
...     n = 5
...     def adder(self, v):
...             return v + self.n
```

Можем ли мы все также вызывать *adder* как функцию, через имя класса? Вполне. Только теперь в функцию надо передавать два аргумента:

```
>>> B.adder(B, 200)
205
>>> B.adder(a, 200)
209
```

Здесь первым аргументом в функцию передается объект, у которого есть поле *n* лишь только потому, что далее к этому полю обращаются через выражение `self.n`.

Однако если атрибут определен так, что предполагается его работа в качестве метода, а не функции, то через класс его уже не вызывают (нет смысла, логика программы этого не подразумевает).

С другой стороны, в ООП есть понятие "статический метод". По сути это функция, которая может вызываться и через класс, и через объект, и которой первым аргументом не подставляется объект, на который она вызывается. В Python статический метод можно создать посредством использования специального декоратора.

Атрибут __dict__

В Python у объектов есть встроенные специальные атрибуты. Мы их не определяем, но они есть. Одним из таких атрибутов объекта является свойство __dict__. Его значением является словарь, в котором ключи – это имена свойств экземпляра, а значения – текущие значения свойств.

```
>>> class B:
...     n = 5
...     def adder(self, v):
...             return v + self.n
...
>>> w = B()
>>> w.__dict__
{}
>>> w.n = 8
>>> w.__dict__
{'n': 8}
```

В примере у экземпляра класса B сначала нет собственных атрибутов. Свойство `n` и метод `adder` – это атрибуты объекта-класса, а не объекта-экземпляра, созданного от этого класса. Лишь когда мы выполняем присваивание новому полю `n` экземпляра, у него появляется собственное свойство, что мы наблюдаем через словарь __dict__.

В следующем уроке мы увидим, что свойства экземпляра обычно не назначаются за пределами класса. Это происходит в методах классах путем присваивание через self. Например, `self.n = 10`.

Атрибут __dict__ используется для просмотра всех текущих свойств объекта. С его помощью можно удалять, добавлять свойства, а также изменять их значения.

```
>>> w.__dict__['m'] = 100
>>> w.__dict__
```

```
{'n': 8, 'm': 100}
>>> w.m
100
```

5. Лабораторная работа №5. Отладка и тестирование.

Допустим, мы желаем проанализировать следующее приложение Python, которое использует библиотеку потоков. В своём следующем примере мы применяем класс **MyThreadClass** для создания последовательности выполнения трех потоков и управления ими. Вот код для отладки целиком:

```
import time
import os
from random import randint
from threading import Thread


class MyThreadClass (Thread):
   def __init__(self, name, duration):
      Thread.__init__(self)
      self.name = name
      self.duration = duration
   def run(self):
      print ("---> " + self.name + \
             " running, belonging to process ID "\
             + str(os.getpid()) + "\n")
      time.sleep(self.duration)
      print ("---> " + self.name + " over\n")
def main():
    start_time = time.time()
```

```
    # Thread Creation
    thread1 = MyThreadClass("Thread#1 ", randint(1,10))
    thread2 = MyThreadClass("Thread#2 ", randint(1,10))
    thread3 = MyThreadClass("Thread#3 ", randint(1,10))

    # Thread Running
    thread1.start()
    thread2.start()
    thread3.start()

    # Thread joining
    thread1.join()
    thread2.join()
    thread3.join()

    # End
    print("End")

    #Execution Time
    print("--- %s seconds ---" % (time.time() -
start_time))

if __name__ == "__main__":
    main()
```

Давайте рассмотрим следующие этапы:

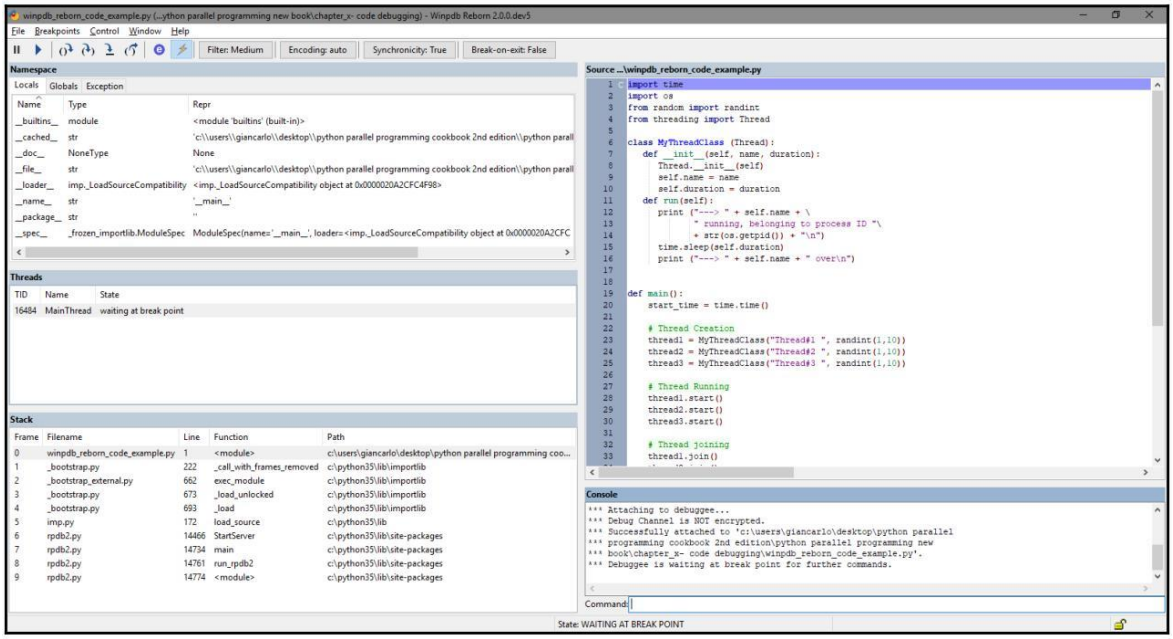
1. Откроем свою консоль и наберём необходимое название в той папке, которая содержит этот файл
примера, **winpdb_reborn_code_example.py**:

---

2. **python -m winpdb .\winpdb_reborn_code_example.py**

---

3. В случае успешной установки должен открыться GUI Winpdb Reborn:

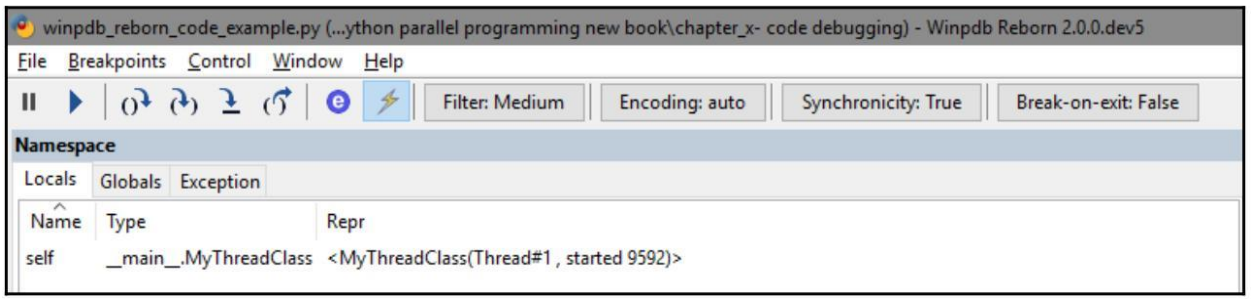


GUI Windpdb Reborn.

4. Как мы можем видеть на следующем снимке экрана, мы вставили две точки прерывания (воспользовавшись меню **Breakpoints**), в обеих отображённых красным строках **12** и **23**:

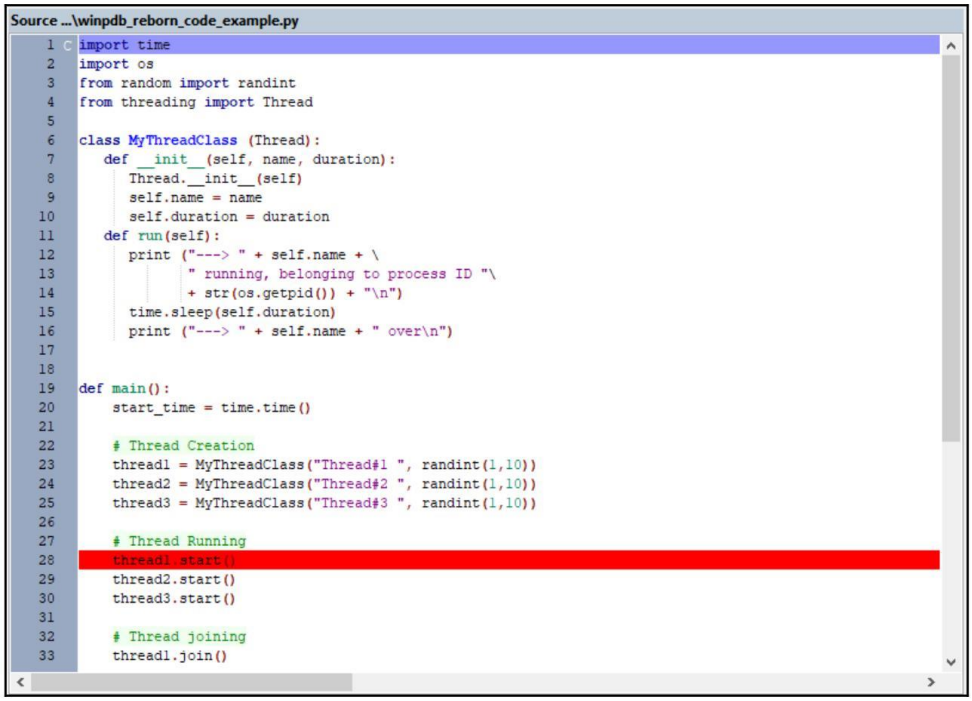
Точки прерывания кода.

5. Оставаясь в окне **Source**, мы помещаем свою мышку на строку **23**, в которой мы поместили свою вторую точку прерывания и нажимаем клавишу ***F8***, а затем на клавишу ***F5***. Наша точка прерывания позволяет исполнить весь код вплоть до выбранной строки. Как вы можете видеть, **Namespace** указывает что мы рассматриваем значение своего экземпляра класса **MyThreadClass** с **thread#1** в качестве аргумента:



Namespace

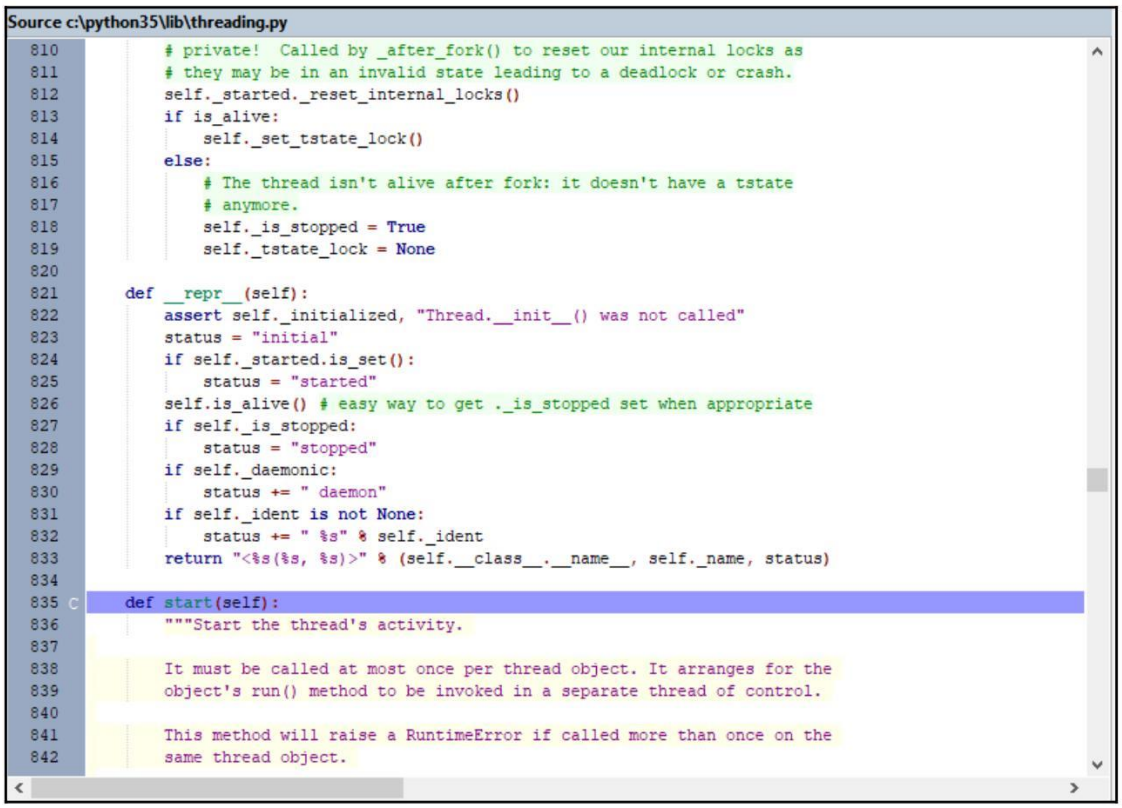
6. Другим основополагающим свойством этого отладчика является возможность **Step Into**, которая позволяет инспектировать не только отлаживаемый код, но ещё и библиотечные функции и вызываемые на исполнение подпрограммы.
7. Прежде чем вы начнёте удалять свои предыдущие точки прерывания (**Menu | Breakpoints | Clear All**), вставьте новую точку прерывания в строке **28**:



Точка прерывания в строке 28

8. Наконец, нажмите клавишу ***F5*** и ваше приложение будет выполнено вплоть до точки прерывания в строке **28**.

9. Затем нажмите ***F7***. Здесь ваше окно исходного кода больше не отображает код нашего примера, а вместо этого используемую нами библиотеку **threading** (смотрите снимок экрана внизу).
10. Тем самым, наша функциональность **Breakpoints** совместно с **Step Into** не только позволяет отлаживать сам код в запросе, но также делает возможным инспектирование всех библиотечных функций и всех применяемых подпрограмм:



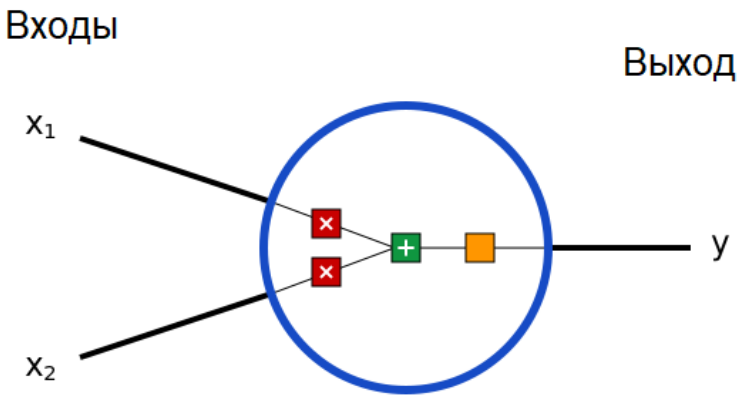
Строка 28 окна исходного кода после выполнения Step Into

6. Лабораторная работа №6. Работа с функциями библиотеки numpy для построения нейронных сетей.

Термин "нейронные сети" сейчас можно услышать из каждого утюга, и многие верят, будто это что-то очень сложное. На самом деле нейронные сети совсем **не такие сложные**, как может показаться! Мы разберемся, как они работают, реализовав одну сеть с нуля на Python.
Эта статья предназначена для **полных новичков**, не имеющих **никакого** опыта в машинном обучении. Поехали!

1. Составные элементы: нейроны

Прежде всего нам придется обсудить нейроны, базовые элементы нейронной сети. Нейрон принимает несколько **входов**, выполняет над ними кое-какие **математические операции**, а потом выдает один **выход**. Вот как выглядит нейрон с двумя входами:



Внутри нейрона происходят три операции. Сначала значения входов умножаются на **веса**:
x1⟶x1∗w1, x2⟶x2∗w2

Затем взвешенные входы складываются, и к ним прибавляется значение **порога** b:
x1∗w1+x2∗w2+b

Наконец, полученная сумма проходит через **функцию активации**:
yy=f(x1∗w1+x2∗w2+b)

Функция активации преобразует неограниченные значения входов в выход, имеющий ясную и предсказуемую форму. Одна из часто используемых функций активации – **сигмоида**:

Сигмоида

Сигмоида выдает результаты в интервале **(0, 1)**. Можно представить, что она «упаковывает» интервал от минус бесконечности до плюс бесконечности в (0, 1): большие отрицательные числа превращаются в числа, близкие к **0**, а большие положительные – к **1**.

Простой пример

Допустим, наш двухвходовой нейрон использует сигмоидную функцию активации и имеет следующие параметры:

w=[0,1] b=4

w=[0, 1] – это всего лишь запись w_1=0, w_2=1 в векторном виде. Теперь зададим нашему нейрону входные данные: **x=[2, 3]**. Мы используем скалярное произведение векторов, чтобы записать формулу в сжатом виде:
(w·x)+b=((w1∗x1)+(w2∗x2))+b=0∗2+1∗3+4=7

$y=f(w\cdot x+b)=f(7)=0.999$

Наш нейрон выдал 0.999 при входах x=[2, 3]. Вот и все! Процесс передачи значений входов дальше, чтобы получить выход, называется **прямой связью (feed forward)**.

Пишем код для нейрона

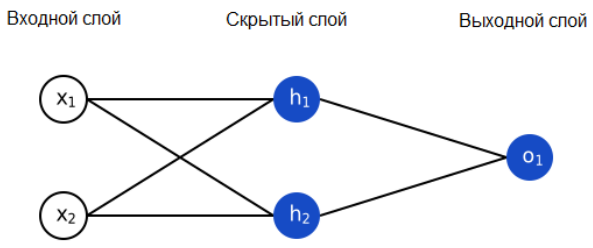
Настало время написать свой нейрон! Мы используем NumPy, популярную и мощную расчетную библиотеку для Python, которая поможет нам с вычислениями:

```
import numpy as np
def sigmoid(x):
  # Наша функция активации: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))
class Neuron:
  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias
  def feedforward(self, inputs):
    # Умножаем входы на веса, прибавляем порог, затем используем функцию активации
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)
weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4                   # b = 4
n = Neuron(weights, bias)
x = np.array([2, 3])       # x1 = 2, x2 = 3
print(n.feedforward(x))    # 0.9990889488055994
```

Узнаете эти числа? Это тот самый пример, который мы только что рассчитали! И мы получили тот же результат – **0.999**.

2. Собираем нейронную сеть из нейронов

Нейронная сеть – это всего лишь несколько нейронов, соединенных вместе. Вот как может выглядеть простая нейронная сеть:



У этой сети два входа, скрытый слой с двумя нейронами (h_1 и h_2) и выходной слой с одним нейроном (o_1). Обратите внимание, что входы для o_1 – это выходы из h_1 и h_2. Именно это создает из нейронов сеть.
Замечание

Скрытый слой – это любой слой между входным (первым) слоем сети и выходным (последним). Скрытых слоев может быть много!

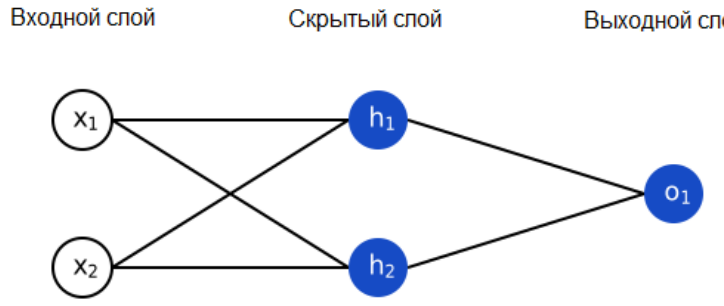
Пример: прямая связь

Давайте используем сеть, изображенную выше, и будем считать, что все нейроны имеют одинаковые веса w=[0, 1], одинаковые пороговые значения b=0, и одинаковую функцию активации – сигмоиду. Пусть h_1, h_2 и o_1 обозначают выходные значения соответствующих нейронов.
Что получится, если мы подадим на вход x=[2, 3]?
h1=h2=f(w·x+b)=f((0∗2)+(1∗3)+0)=f(3)=0.9526o1=f(w·[h1,h2]+b)=f((0∗h1)+(1∗h2)+0)=f(0.9526)=0.7216

Если подать на вход нашей нейронной сети **x=[2, 3]**, на выходе получится **0.7216**. Достаточно просто, не правда ли?
Нейронная сеть может иметь **любое количество слоев**, и в этих слоях может быть **любое количество нейронов**. Основная идея остается той же: передавайте входные данные по нейронам сети, пока не получите выходные значения. Для простоты мы будем использовать сеть, показанную выше, до конца статьи.

Пишем код нейронной сети

Давайте реализуем прямую связь для нашей нейронной сети. Напомним, как она выглядит:




```
import numpy as np
# ... вставьте сюда код из предыдущего раздела
class OurNeuralNetwork:
  '''
  Нейронная сеть с:
    - 2 входами
```

```
    - скрытым слоем с 2 нейронами (h1, h2)
    - выходным слоем с 1 нейроном (o1)
  Все нейроны имеют одинаковые веса и пороги:
    - w = [0, 1]
    - b = 0
  '''
  def __init__(self):
    weights = np.array([0, 1])
    bias = 0
    # Используем класс Neuron из предыдущего раздела
    self.h1 = Neuron(weights, bias)
    self.h2 = Neuron(weights, bias)
    self.o1 = Neuron(weights, bias)
  def feedforward(self, x):
    out_h1 = self.h1.feedforward(x)
    out_h2 = self.h2.feedforward(x)
    # Входы для o1 - это выходы h1 и h2
    out_o1 = self.o1.feedforward(np.array([out_h1,
out_h2]))
    return out_o1
network = OurNeuralNetwork()
x = np.array([2, 3])
print(network.feedforward(x)) # 0.7216325609518421
```

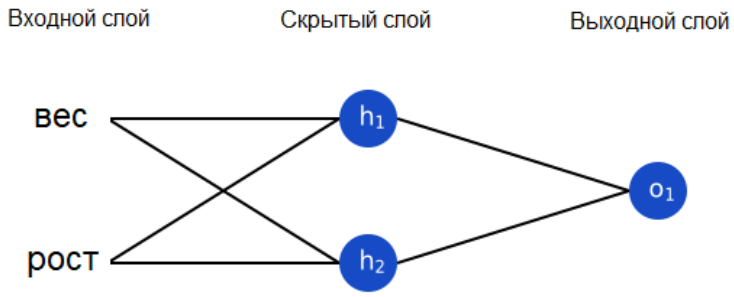
Мы снова получили 0.7216! Похоже, наша сеть работает.

3. Обучаем нейронную сеть (часть 1)

Допустим, у нас есть следующие измерения:

Имя	Вес (в фунтах)	Рост (в дюймах)	Пол
Алиса	133 (54.4 кг)	65 (165,1 см)	Ж
Боб	160 (65,44 кг)	72 (183 см)	М
Чарли	152 (62.2 кг)	70 (178 см)	М
Диана	120 (49 кг)	60 (152 см)	Ж

Давайте обучим нашу нейронную сеть предсказывать пол человека по его росту и весу.



Мы будем представлять мужской пол как 0, женский – как 1, а также сдвинем данные, чтобы их было проще использовать:

Имя	Вес (минус 135)	Рост (минус 66)	Пол
Алиса	-2	-1	1
Боб	25	6	0
Чарли	17	4	0
Диана	-15	-6	1

Замечание

Я выбрал величину сдвигов (135 и 66), чтобы числа выглядели попроще. Обычно сдвигают на среднее значение.

Потери

Прежде чем обучать нашу нейронную сеть, нам нужно как-то измерить, насколько "хорошо" она работает, чтобы она смогла работать "лучше". Это измерение и есть потери (loss).
Мы используем для расчета потерь среднюю квадратичную ошибку (mean squared error, MSE):
$MSE=1n\sum i=1n(ytrue-ypred)2$

Давайте рассмотрим все используемые переменные:

- n – это количество измерений, в нашем случае 4 (Алиса, Боб, Чарли и Диана).
- y представляет предсказываемое значение, Пол.
- y_{true} – истинное значение переменной ("правильный ответ"). Например, для Алисы y_{true} будет равна 1 (женский пол).
- y_{pred} – предсказанное значение переменной. Это то, что выдаст наша нейронная сеть.

$(y_{true}-y_{pred})^2$ называется квадратичной ошибкой. Наша функция потерь просто берет среднее значение всех квадратичных ошибок – поэтому она и называется средней квадратичной ошибкой. Чем лучшими будут наши предсказания, тем меньшими будут наши потери!

Лучшие предсказания = меньшие потери.

Обучение нейронной сети = минимизация ее потерь.

Пример расчета потерь

Предположим, что наша сеть всегда возвращает 0 – иными словами, она уверена, что все люди мужчины. Насколько велики будут наши потери?

Имя	y_{true}	y_{pred}	$(y_{true}-y_{pred})2$
Алиса	1	0	1
Боб	0	0	0
Чарли	0	0	0
Диана	1	0	1

MSE=14(1+0+0+1)=0.5

Пишем функцию средней квадратичной ошибки

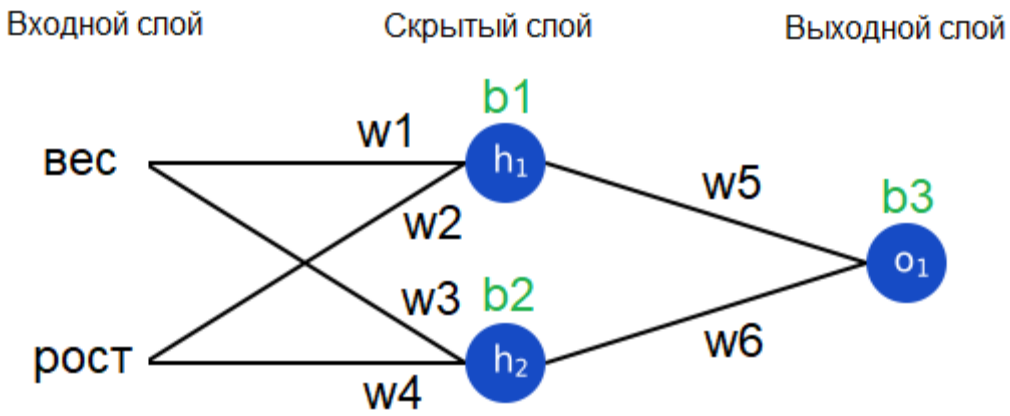
Вот небольшой кусок кода, который рассчитает наши потери. Если вы не понимаете, почему он работает, прочитайте в руководстве NumPy про операции с массивами.

```
import numpy as np
def mse_loss(y_true, y_pred):
  # y_true и y_pred - массивы numpy одинаковой длины.
  return ((y_true - y_pred) ** 2).mean()
y_true = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])
print(mse_loss(y_true, y_pred)) # 0.5
```

Пишем код всей нейронной сети

Наконец настало время реализовать всю нейронную сеть.

Имя	Вес (минус 135)	Рост (минус 66)	Пол
Алиса	-2	-1	1
Боб	25	6	0
Чарли	17	4	0
Диана	-15	-6	

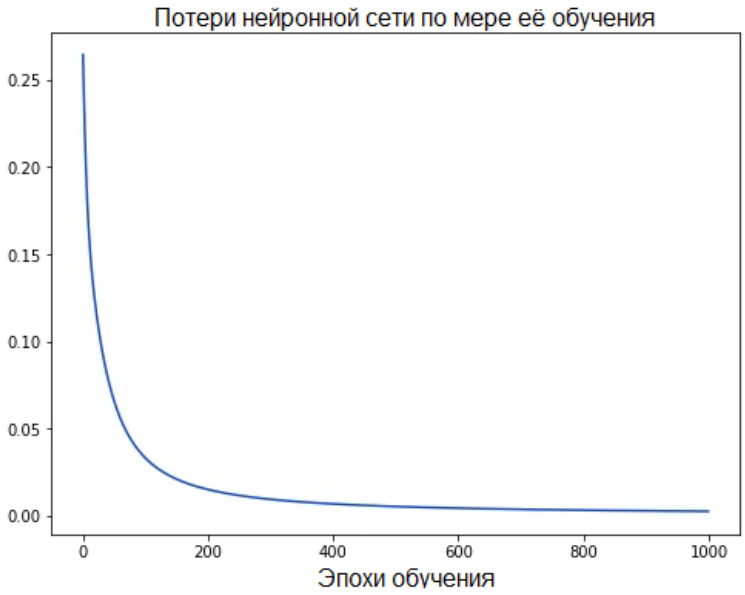



```
import numpy as np
def sigmoid(x):
  # Сигмоидная функция активации: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))
def deriv_sigmoid(x):
  # Производная сигмоиды: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)
def mse_loss(y_true, y_pred):
  # y_true и y_pred - массивы numpy одинаковой длины.
  return ((y_true - y_pred) ** 2).mean()
class OurNeuralNetwork:
  '''
  Нейронная сеть с:
    - 2 входами
    - скрытым слоем с 2 нейронами (h1, h2)
    - выходной слой с 1 нейроном (o1)
```

```
  *** DISCLAIMER ***:
  Следующий код простой и обучающий, но НЕ оптимальный.
  Код реальных нейронных сетей совсем на него не похож. НЕ
копируйте его!
  Изучайте и запускайте его, чтобы понять, как работает эта
нейронная сеть.
  '''
  def __init__(self):
    # Веса
    self.w1 = np.random.normal()
    self.w2 = np.random.normal()
    self.w3 = np.random.normal()
    self.w4 = np.random.normal()
    self.w5 = np.random.normal()
    self.w6 = np.random.normal()
    # Пороги
    self.b1 = np.random.normal()
    self.b2 = np.random.normal()
    self.b3 = np.random.normal()
  def feedforward(self, x):
    # x is a numpy array with 2 elements.
    h1 = sigmoid(self.w1 * x[0] + self.w2 * x[1] + self.b1)
    h2 = sigmoid(self.w3 * x[0] + self.w4 * x[1] + self.b2)
    o1 = sigmoid(self.w5 * h1 + self.w6 * h2 + self.b3)
    return o1
  def train(self, data, all_y_trues):
    '''
    - data - массив numpy (n x 2) numpy, n = к-во
наблюдений в наборе.
    - all_y_trues - массив numpy с n элементами.
      Элементы all_y_trues соответствуют наблюдениям в
data.
    '''
    learn_rate = 0.1
    epochs = 1000 # сколько раз пройти по всему набору
данных
    for epoch in range(epochs):
      for x, y_true in zip(data, all_y_trues):
        # --- Прямой проход (эти значения нам понадобятся
позже)
        sum_h1 = self.w1 * x[0] + self.w2 * x[1] + self.b1
        h1 = sigmoid(sum_h1)
        sum_h2 = self.w3 * x[0] + self.w4 * x[1] + self.b2
        h2 = sigmoid(sum_h2)
```

```
        sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
        o1 = sigmoid(sum_o1)
        y_pred = o1
        # --- Считаем частные производные.
        # --- Имена: d_L_d_w1 = "частная производная L по
w1"
        d_L_d_ypred = -2 * (y_true - y_pred)
        # Нейрон o1
        d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
        d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
        d_ypred_d_b3 = deriv_sigmoid(sum_o1)
        d_ypred_d_h1 = self.w5 * deriv_sigmoid(sum_o1)
        d_ypred_d_h2 = self.w6 * deriv_sigmoid(sum_o1)
        # Нейрон h1
        d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
        d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
        d_h1_d_b1 = deriv_sigmoid(sum_h1)
        # Нейрон h2
        d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
        d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
        d_h2_d_b2 = deriv_sigmoid(sum_h2)
        # --- Обновляем веса и пороги
        # Нейрон h1
        self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1
* d_h1_d_w1
        self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1
* d_h1_d_w2
        self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1
* d_h1_d_b1
        # Нейрон h2
        self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2
* d_h2_d_w3
        self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2
* d_h2_d_w4
        self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2
* d_h2_d_b2
        # Нейрон o1
        self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
        self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
        self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3
      # --- Считаем полные потери в конце каждой эпохи
      if epoch % 10 == 0:
        y_preds = np.apply_along_axis(self.feedforward, 1,
data)
```

```
        loss = mse_loss(all_y_trues, y_preds)
        print("Epoch %d loss: %.3f" % (epoch, loss))
# Определим набор данных
data = np.array([
  [-2, -1],  # Алиса
  [25, 6],   # Боб
  [17, 4],   # Чарли
  [-15, -6], # Диана
])
all_y_trues = np.array([
  1, # Алиса
  0, # Боб
  0, # Чарли
  1, # Диана
])
# Обучаем нашу нейронную сеть!
network = OurNeuralNetwork()
network.train(data, all_y_trues)
```



Теперь мы можем использовать нашу сеть для предсказания пола:

```
    # Делаем пару предсказаний
emily = np.array([-7, -3]) # 128 фунтов (52.35 кг), 63
дюйма (160 см)
frank = np.array([20, 2])  # 155 pounds (63.4 кг), 68
inches (173 см)
print("Эмили: %.3f" % network.feedforward(emily)) # 0.951 -
Ж
print("Фрэнк: %.3f" % network.feedforward(frank)) # 0.039 -
М
```

Вы сделали это! Давайте перечислим все, что мы с вами сделали:

- Определили **нейроны**, составные элементы нейронных сетей.
- Использовали **сигмоидную функцию активации** для наших нейронов.
- Увидели, что нейронные сети – это всего лишь несколько нейронов, соединенных друг с другом.
- Создали набор данных, в котором Вес и Рост были входными данными (или **признаками**), а Пол – выходным (или **меткой**).
- Узнали о **функции потерь** и **средней квадратичной ошибке (MSE)**.
- Поняли, что обучение нейронной сети – это всего лишь минимизация ее потерь.
- Использовали **метод обратного распространения (backpropagation)** для расчета частных производных.
- Использовали **стохастический градиентный спуск (SGD)** для обучения нашей сети.

Перед вами – множество путей, на которых вас ждет масса нового и интересного:

- Экспериментируйте с большими и лучшими нейронными сетями, используя подходящие библиотеки
  вроде Tensorflow, Keras и PyTorch.