Проект «ИТ-класс в московской школе»

Учебное пособие

Направление «Большие данные»

подготовлен федеральным государственным бюджетным образовательным учреждением высшего образования

«Московский государственный технологический университет «СТАНКИН»

Оглавление

1. Python 8

1.1 Определение 8

1.2 Причины популярности 8

1.3 Сфера применения 12

1.4 Среды разработки 16

2. Популярные библиотеки Python 20

2.1 NumPy 20

2.1.1 Начало работы 21

2.1.2 Создание и просмотр массивов 21

2.1.3 Базовые операции над массивами 24

2.1.4 Индексы, срезы, итерации 26

2.1.4 Манипуляции с формой (размерностью) 29

2.1.5 Объединение массивов 30

2.1.6 Разбиение массива 31

2.1.7 Копии и представления 32

2.1.8 Операции из линейной алгебры 33

2.2 Scikit-learn 34

2.3.1 Начало работы 36

2.3.2 Линейные методы 36

2.3.3 Метрические методы 37

2.3.3.1 Неконтролируемые ближайшие соседи 38

2.3.3.2 Классификация ближайших соседей 39

2.3.4 Деревья решений 40

2.3.5 Метод опорных векторов (SVM) 41

2.3.5.1 Классификация 42

2.3.5.2 Регрессия 44

2.3.6 Наивный Байес 45

2.3.6.1 Гауссовский наивный Байес (GNB) 46

2.3.6.2 Мультиномиальный наивный Байес (MNB) 47

2.3.6.3 Дополнение наивного Байеса (CNB) 48

2.3.6.4 Наивный Байес Бернулли (BNB) 49

2.3.7 Стохастический градиентный спуск (SGD) 50

2.3.7.1 Классификация 51

2.3.7.2 Регрессия 55

2.3.7.3 Онлайн одноклассовая SVM 56

2.3.8 Модели нейронных сетей (с учителем) 58

2.3.8.1 Многослойный персептрон 58

2.3.8.2 Классификация 59

2.3.8.3 Регрессия 61

2.3.8.4 Регуляризация 62

2.3.9 Модели нейронных сетей (без учителя) 62

2.3.9.1 Ограниченные машины Больцмана 62

2.3 TensorFlow 67

3. Организация работы с большими данными 67

3.1 Принципы Big Data 67

3.2 Hive 68

3.3 Apache Kafka 69

3.4 Apache SPARK 72

3.5 Hadoop 73

3.6 HBase 77

4. Пример использование DataLake архитектуры для организации хранения технологических данных 80

4.1 Архитектура решения на основе DATA LAKE 80

4.2 Реализация сбора информации на базе IOT Azure 83

4.3 Архитектура решения 84

4.4. Схема базы данных 88

4.5 Итоги 89

5. Практическая часть использования и аналитики данных 90

5.1 Взаимодействие с Python в jupyter-ноутбуках 92

5.1.1 Запуск Python из командной строки 92

5.1.2. Математика в Python 94

5.1.3 Порядок операций 95

5.1.4 Комментарии 96

5.2 Программы в файле, переменные и строки 96

5.2.1 Введение 96

5.2.2 Написание программ 96

5.2.3 Переменные 98

5.2.5 Строки 99

5.3 Циклы 103

5.3.1 Введение 103

5.3.2 Цикл "While" ("пока") 103

5.3.3 Логические, или булевые, выражения 106

5.3.4 Условные выражения 107

5.3.5 else и elif - операторы "когда это не так" 109

5.3.6 Отступ 110

5.4 Функции 112

5.4.1 Введение 112

5.4.2 Использование функции 113

5.4.3 Параметры и возвращаемые значения - взаимодействие с функциями 113

5.4.4 Программа-калькулятор 115

5.4.5 Определите свои собственные функции 119

5.4.6 Передача параметров функциям 121

5.4.7 Заключительная программа 122

5.4.8 Хитрые способы передачи параметров 124

5.5 Кортежи, списки и словари 126

5.5.1 Введение 126

5.5.2 Решение - списки, кортежи и словари 126

5.5.3 Словари 130

5.5.4 Массивы 134

5.6 Цикл for 135

5.6.1 Введение 135

5.6.2 Цикл for 135

5.6.3 Создание функции "меню" 140

5.6.4 Наша первая «игра» 141

5.6.5 Улучшение игры 147

5.7 Классы 151

5.7.1 Введение 151

5.7.2 Создание класса 152

5.7.3 Использование class 153

5.7.4 Базовая терминология.. 155

5.7.5 Наследование.. 156

5.7.6 Указатели и словари классов 157

5.8 Модули 158

5.8.1 Введение.. 158

5.8.2 Модуль? Что такое модуль?.. 159

5.8.3 Пример ещё одного модуля "thingummyjigs" 160

5.9 Файловый ввод-вывод 161

5.9.1 Открытие файла.. 161

5.9.2 Курсор при вводе-выводе.. 162

5.9.3 Другие функции ввода / вывода 163

5.9.4 Объекты, сохраняемые в файл 165

5.10 Обработка исключений (ошибок).. 167

5.10.1 Введение.. 167

5.10.2 Ошибки - человеческие ошибки.. 168

5.10.3 Исключения 170

5.10.4 Бесконечные ошибки 173

5.11 Стандартные библиотеки Python.. 175

5.11.1 Интерфейс операционной системы 175

5.11.2 Поиск файлов на основе подстановочных данных (Wildcards) .. 176

5.11.3 Аргументы командной строки 177

5.11.4 Соответствие строковому шаблону (регулярные выражения).... 179

5.11.5 Математика, случайные числа и статистика 182

5.11.6 Доступ в Интернет 185

5.11.7 Даты и время 185
5.11.8 Сжатие данных 186
5.11.9 Измерение производительности 186
5.11.10 Контроль качества 188
5.11.11 Многопоточность 189
5.12 TensorFlow 192
5.12.1 Начало работы 193
5.12.2 Понятие “тензор” 193
5.12.3 Работа с тензорами в TensorFlow 195
5.12.4 Операции над тензорами 196
5.12.5 Формы тензоров 198
5.12.6 Модули, слои и модели 199
5.12.7 Пример 1. Y=(3X + 1) 206

Данный курс посвящен введению по работу с данным, в том числе рассматривается и работа с большим объемом данных, с использованием языка Python.

Курс рассчитан на уверенного пользователя ПК. Желательно иметь опыт программирования. Язык значения не имеет. В этом курсе все примеры будут рассматриваться с применением языка Python, а также различных библиотек и фреймворков. Начиная с основ (что такое python) мы плавно перейдем к понятиям данные и большие данные.

1. Python

1.1 Определение

С академической точки зрения, **Python** — высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ. Язык является полностью объектно-ориентированным — всё является объектами. Необычной особенностью языка является выделение блоков кода пробельными отступами. Python известен как интерпретируемый и используется в том числе для написания скриптов. Недостатками языка являются зачастую более низкая скорость работы и более высокое потребление памяти написанных на нём программ по сравнению с аналогичным кодом, написанным на компилируемых языках, таких как Си или C++.

Но столь исчерпывающее определение, не даёт понимания того, что такое Python, кроме скорости работы и читаемости кода.

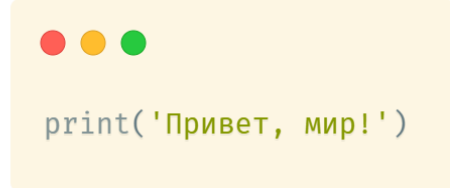
Главное, что нужно понимать, **Python** — это мощный инструмент для создания программ самого разнообразного назначения, от простейших скриптов до научных расчётов, доступный даже для детей.

1.2 Причины популярности

Почему **Python** сегодня один из самых популярных языков программирования? На самом деле, причин на то несколько, и по большой части они вытекают из сказанного ранее.

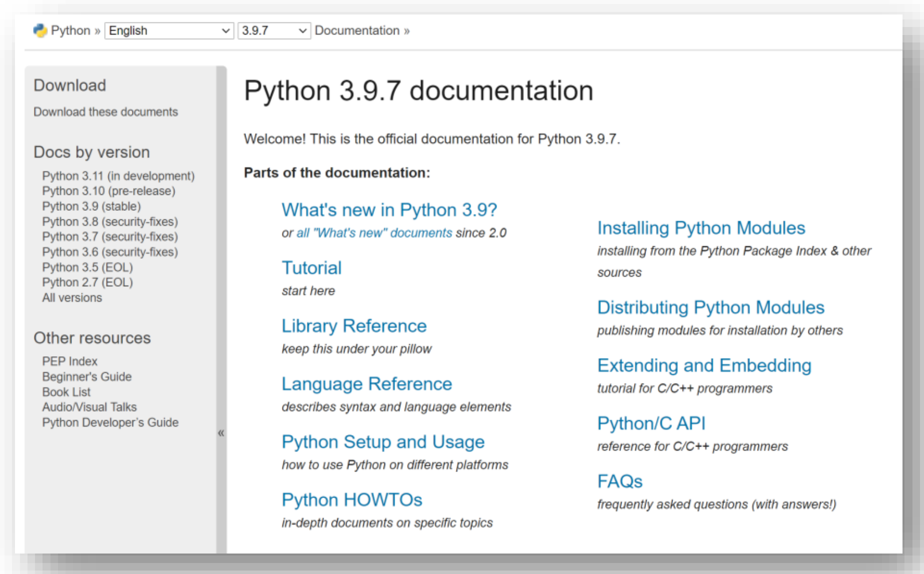
- **Он простой**

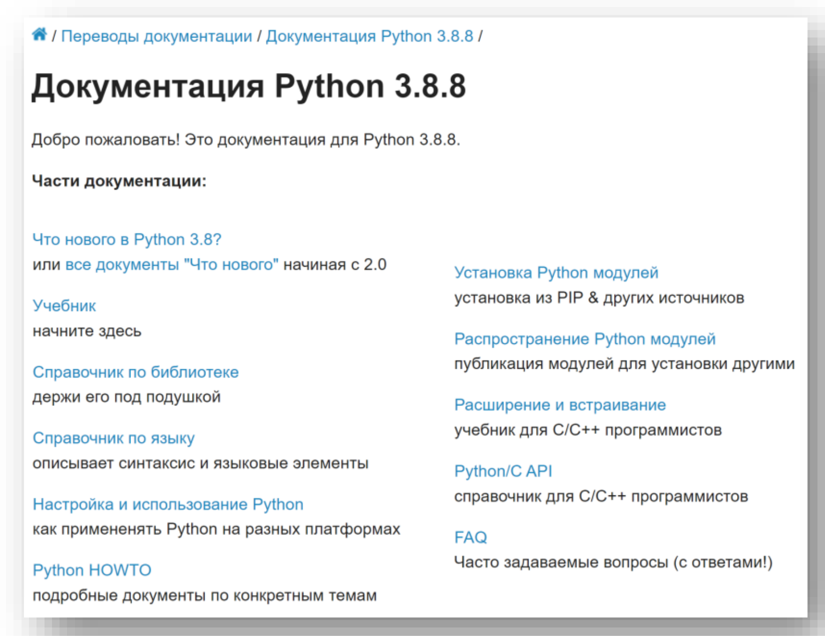
Синтаксис языка очень сильно похож на обычный английский. Например, вывод текста на экран осуществляется командой “print”, что с английского можно перевести как “печатать”. И такая простота прослеживается внутри всего языка. Даже сложные вещи можно писать почти, не пользуясь документацией, достаточно знания английского.



- **Хорошая документация**

Раз уж затронули тему документации в прошлом пункте, давайте ею и продолжим. Если вдруг у вас есть вопросы как работает какая-нибудь функция языка, какой стиль оформления кода нужен, чтобы было хорошо и красиво, как установить **Python** в конце концов, стандартная документация даст ответы на все ваши вопросы. Правда с оговоркой, что на английском. На русском тоже можно найти очень много источников, но все же лучше читать в оригинале, т.к. может много нюансов, которые могут быть понятны и без перевода.



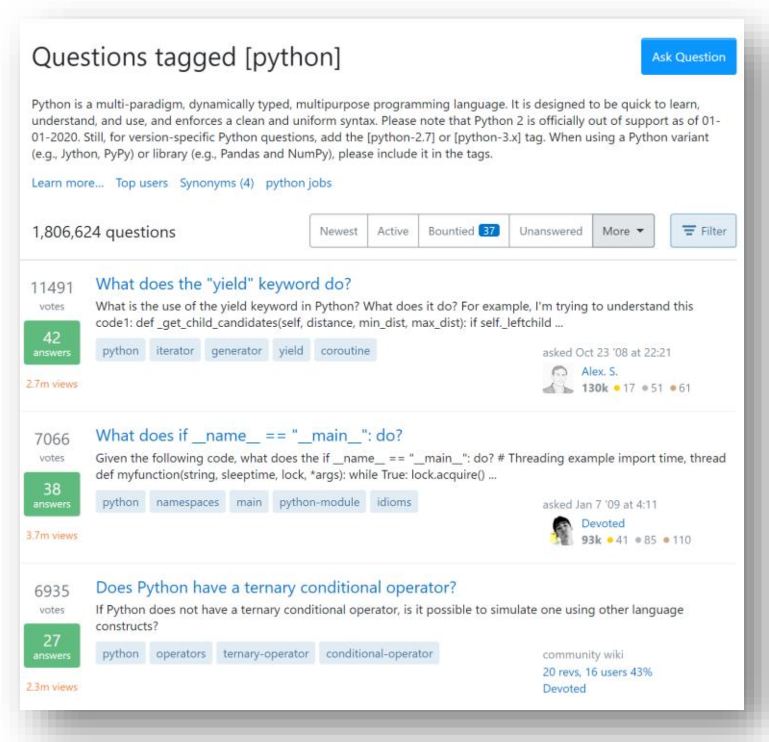


Если же ответ на Ваш вопрос не кажется понятным или вы его не нашли в ней, то всегда есть большое количество форумов по языку Python, где скорее всего люди уже всё расписали. Это же и является следующий причиной популярности.

- **Большое и крепкое сообщество**

Огромное количество людей по всему миру, объединённое любовью к этому языку, делает поистине невероятные вещи. На любой вопрос по языку либо уже есть ответ, либо вам его дадут, и будет это быстро. А если Вам хочется

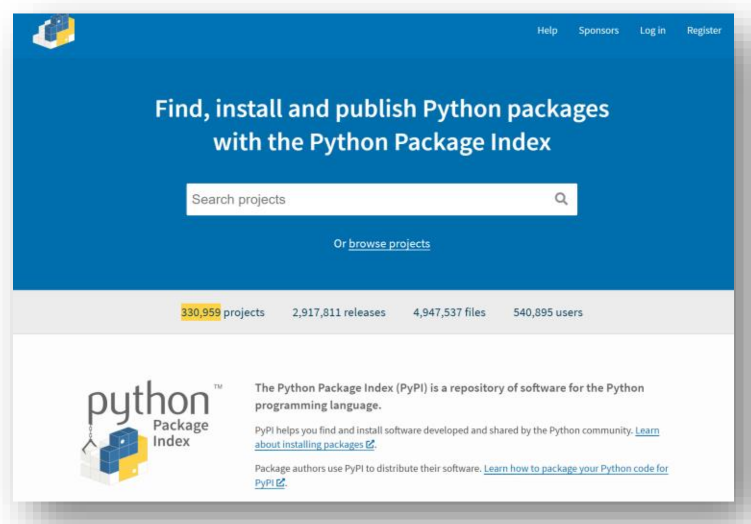
чего-то большого, то комьюнити уже разработало и может предложить огромное количество библиотек, кардинально решающих все возникающие проблемы.



К сожалению, официальная документация только на английском, но на самом деле всё не совсем так. В огромном сообществе нашлись люди, которые перевели большую часть документации на разные языки, в том числе и на русский. Большое комьюнити порождает большое количество информации, проектов и профильных ресурсов.

- **Большое количество библиотек и проектов**

Редкий язык обладает таких количеством библиотек. Почти на любую задачу уже есть готовая библиотек. Просто нужно установить и настроить нужную библиотеку, подготовить всё под себя и готово.

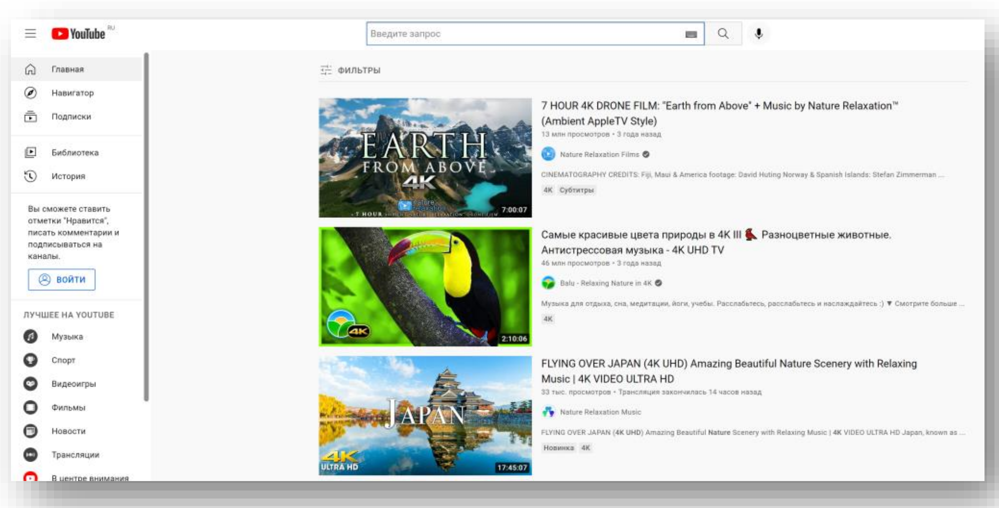


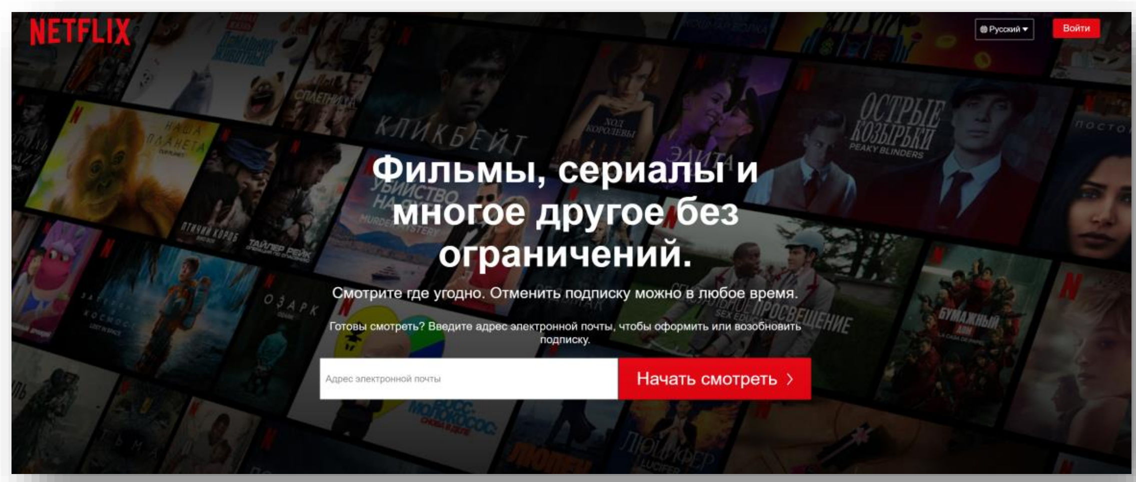
И это только верхушка айсберга. Все прелесть работы с языком программирования Python вы сможете оценить непосредственно выполняю задания на нем. Давайте перейдём к тому, где же его применяют.

1.3 Сфера применения

- **Web-разработка**

На **Python** можно делать весь backend интернет-ресурса, который будет выполняться на сервере. Делается это при помощи специальных библиотек (Flask и Django), написанных на этом языке. С их помощью упрощается процесс обработки адресов, обращение к базам данных и создание HTML, отображающихся на пользовательских страницах.

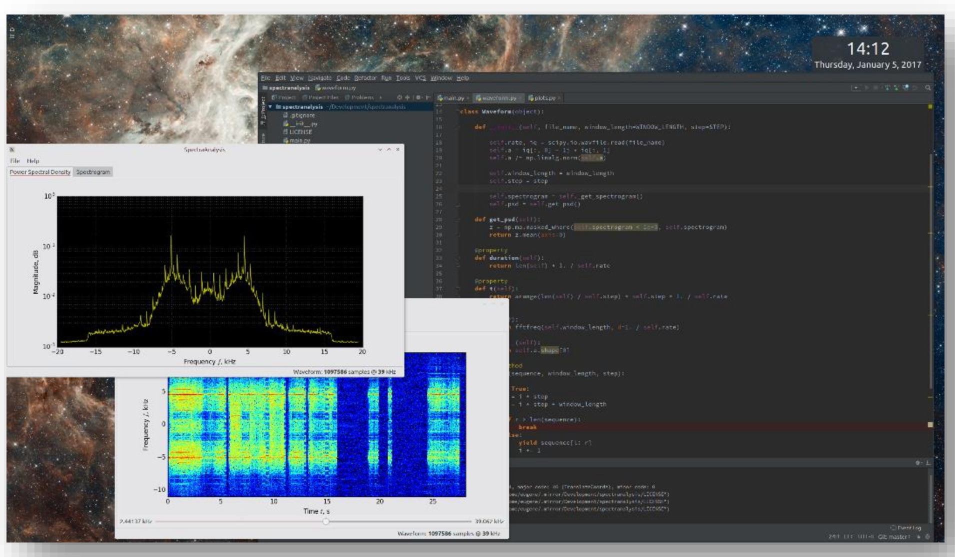




Вы будете удивлены, когда узнаете, как много всего в Интернете работает именно на **Python**. Например, это *YouTube* и *Netflix*. А ещё есть *Instagram* и *Google* – последний решает, что вам из запросов выдать как раз благодаря **Python**.

- **Системное и прикладное программирование**

Ещё одна монетка в копилку возможностей **Python** – это интерфейсы языка, которые позволяют управлять службами операционных систем Windows, MacOS, Linux и др. Благодаря этому, **Python** открывает массу возможностей для создания портативных программ. Не секрет, что этот язык применяется для написания приложений, используемых системными администраторами. Таким образом, **Python** ускоряет поиск и открытие файлов, запуск приложений, облегчает вычисления и многое другое.



- **Сложные вычисления**

Это та самая сфера, где **Python** может потягаться в своих возможностях с FORTRAN или C++. Специальная библиотека *NumPy*, написанная для математических расчётов, прекрасно функционирует с массивами, интерфейсами уравнений и другими данными.

Но *NumPy* предназначен не только для вычислений. Помимо своей основной задачи, с её помощью можно создавать анимированные элементы и прорисовывать объекты в среде 3D, производя при этом параллельные вычисления. Например, популярное дополнение *ScientificPython* может похвастаться собственными библиотеками, которые созданы для вычислительных процессов в сфере науки. При этом, применяется **Python** в абсолютно разных областях науки: от математики до химии и биологии.

Помимо расчётов, **Python** позволяет визуализировать полученные данные, что довольно удобно.

О библиотеке *NumPy* будет подробнее рассказано в следующих главах.

- **Машинное обучение**

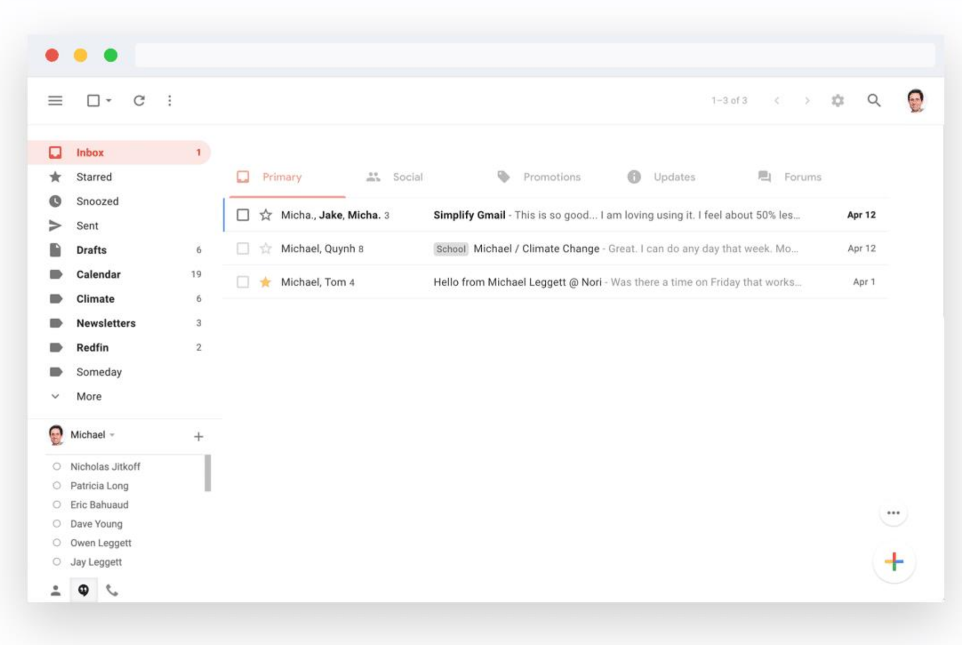
Помимо основного инструментария, у **Python** есть дополнительные библиотеки и фреймворки, позволяющие работать в области машинного обучения. Особой популярностью пользуются *scikit-learn* и *TensorFlow*. *Scikit-learn* отличается тем, что в него уже встроены самые распространенные

алгоритмы обучения. *TensorFlow*, в свою очередь – это низкоуровневая библиотека, которая открывает возможности для создания алгоритмов самому. Далее также, как и с *NumPy*, они будут рассмотрены.

Процессы машинного обучения, основанные на языке программирования **Python**, позволяют реализовывать системы распознавания лиц и голоса, создавать нейронные сети, глубокое обучение и многое другое.

- **Автоматизация процессов**

Одним из самых востребованных способов использования языка **Python** является создание мелких скриптов, автоматизирующих некоторые рабочие процессы. Например, можно написать вполне простой код, который будет «самостоятельно» работать с письмами на электронной почте. Если человеку необходимо отсортировывать письма с определенными ключевыми словами или фразами, то вручную это сделать довольно проблематично, а вот скрипт справится с этой задачей без проблем.



1.4 Среды разработки

Поговорим немного об инструментах, которыми пользуются разработчики непосредственно в процессе написания кода и его тестирования.

Среда разработки — комплекс программных средств, используемый программистами для разработки программного обеспечения.

Это значит, что большую часть времени работа будет выполняться именно в этих программах. Весь код, проверка его корректности, запуск и оценка работы готовой программы обычно происходит внутри данного программного комплекса.

Зачастую, разработчики сильно прикипают к своей среде разработки и эффективно могут работать только в ней. Понятно, работа любимыми инструментами позволяет не тратить время на поиск того, как выполнять какое-нибудь обычное действие в новой среде. К тому же, все горячие клавиши уже давно не только в голове, но и на кончиках пальцев.

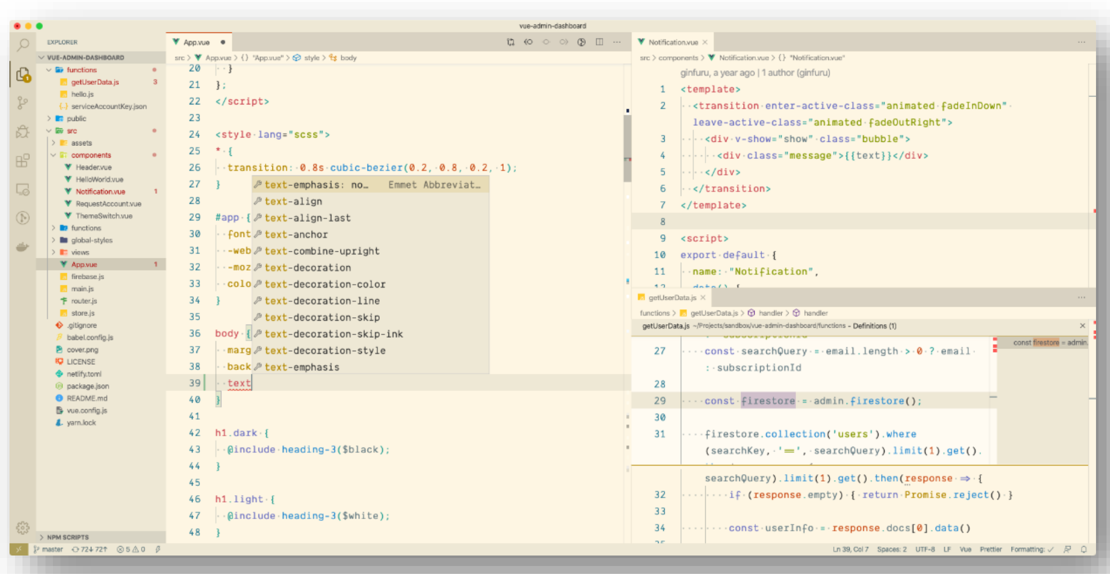
Сейчас мы рассмотрим среды разработки и текстовые редакторы, в некоторые из которых уже интегрированы основные элементы, необходимые для разработки. Наверняка, с некоторыми из них вы уже успели познакомиться.

Выбор своего инструмента позволит выполнять работу быстро и с комфортом. Поэтому не бойтесь пробовать, возможно вы что-то пропустили.

Говоря о средах разработки, нас конкретно будут интересовать решения, позволяющие вести разработку на языке **Python**. Общее число сред для прочих языков огромно и их мы опустим.

- **Visual Studio Code**

Visual Studio Code — редактор исходного кода, разработанный Microsoft для Windows, Linux и macOS. Позиционируется как «лёгкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений.



С точки зрения открытости и широты возможностей, пожалуй, самый лучший вариант. Хотя это и редактор кода, однако имеет интегрированные инструменты, достаточные для комфортного редактирования кода почти на любом языке программирования, не только на **Python**. Имеется открытая площадка, куда множество разработчиков выкладывают свои дополнения, расширяющие и без того хорошие функциональные возможности.

Также плюсом будет то, что код доступен на сайте GitHub, где каждый может предложить своё нововведение, исправить найденную ошибку или же сделать копию проекта и развивать свою версию редактора.

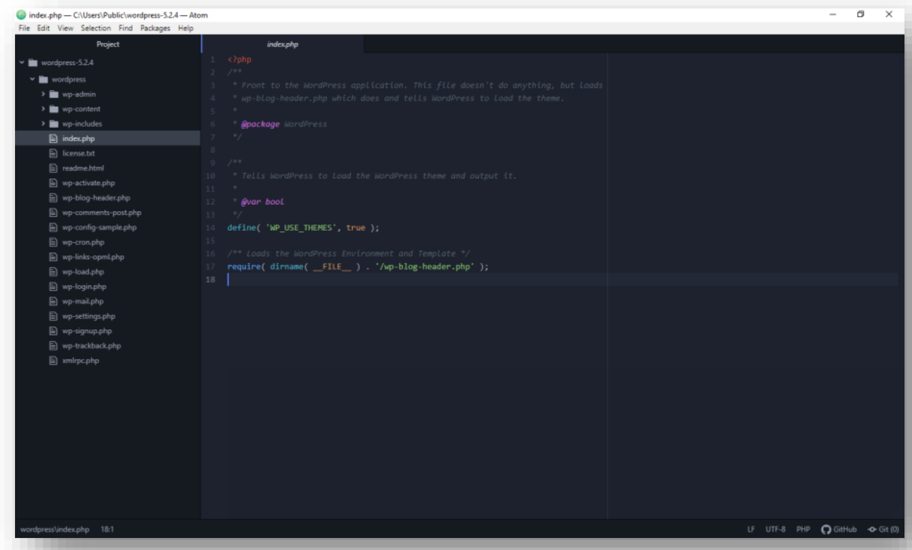
- **JetBrains PyCharm**

JetBrains PyCharm — интегрированная среда разработки для языка программирования Python. Предоставляет средства для анализа кода, графический отладчик, инструмент для запуска юнит-тестов и поддерживает веб-разработку на Django и Flask. PyCharm разработана компанией JetBrains на основе IntelliJ IDEA.

PyCharm является более удобным вариантом, нежели VS Code. Инструментов встроено гораздо больше, есть дополнения на любой вкус, пускай и в меньшем количестве по сравнению с первым вариантом. Имеет чуть меньший порог вхождения и более простой процесс управления проектами. Из коробки, что называется, есть поддержка систем контроля версий.

Распространяется в двух версиях: Community и Professional, которые между собой отличаются лишь наличием инструментов, необходимых профессиональным разработчикам. Соответственно, последняя не бесплатная и стоит порядка 200 долларов США за первый год пользования, затем с 3 года по 119. Если Вы выберете эту среды разработки, то в рамках нашей деятельности, будет достаточно и обычной версии, нам не придётся столкнуться с ограничениями инструментария.

Краткий обзор сред разработки замыкают Notepad++ и Atom. Они также являются редакторами кода, присутствуют различные дополнения, но их использования может быть оправдано скорее на небольших проектах из-за особенностей работы. Они тоже хороши, если правильно настроить, будут работать точно не хуже первых двух, но это потребует чуть больших усилий на начальном этапе.



В эту группу также можно добавить редактор Sublime Text. Если Вы их используете, в этом нет ничего плохого, главное же, чтобы именно вам было

комфортно выполнять работу. Можно делать хоть в обычном блокноте на компьютере.

2. Популярные библиотеки Python

Как уже было сказано ранее, Python обладает огромным количеством библиотек, большое число из которых предназначено для проведения различных научных вычислений и анализу крупных объёмов данных.

Самыми известными и активно используемыми библиотеками для анализа данных являются NumPy, Scikit-learn и TensorFlow.

2.1 NumPy

NumPy (от сокращения Numerical Python) — библиотека с открытым исходным кодом, позволяющая оперировать многомерными массивами (в т.ч. матрицами), а также добавляющая поддержку высокоуровневых математических функций, предназначенных для работы с многомерными массивами.

За счёт того, что внутри NumPy реализует вычислительные алгоритмы, оптимизированные под работу с многомерными массивами, скорость их выполнения сопоставима с таковой у MATLAB. Само же ядро библиотеки написано на языке Си.

NumPy можно рассматривать как альтернативу MATLAB. Язык программирования MATLAB внешне очень похож на NumPy: оба они интерпретируемые, оба позволяют выполнять операции над массивами, а не над скалярами.

2.1.1 Начало работы

Основным объектом NumPy является однородный многомерный массив (в NumPy называется numpy.ndarray). Это многомерный массив элементов одного типа.

Наиболее важные атрибуты объектов ndarray:

- ndarray.**ndim** - число измерений массива (матрицы);
- ndarray.**shape** - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждому измерению;
- ndarray.**size** - количество элементов массива, которое равно произведению всех элементов атрибута shape;
- ndarray.**dtype** - объект, описывающий тип элементов массива. NumPy имеет набор уже встроенных типов: *bool_*, *character*, *int8*, *int16*, *int32*, *int64*, *float8*, *float16*, *float32*, *float64*, *complex64*, *object_*. Есть возможность определить собственные типы данных, как и использовать стандартные для языка Python;
- ndarray.**data** - буфер, содержащий фактические элементы массива;
- ndarray.**itemsize** - размер каждого элемента массива в байтах.

2.1.2 Создание и просмотр массивов

В NumPy существует множество способов создать массив. Один из наиболее простых - создать массив из обычных списков или кортежей Python, используя функцию numpy.array():

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

Функция numpy.array() преобразует переданные ей в качестве аргумента последовательности в многомерные массивы. Тип элементов получаемого массива будет зависеть от типа элементов переданной последовательности. Однако, его можно и переопределить в момент создания массива.

Пример без переопределения типа:

```
>>> b = np.array([[1, 2, 3], [4, 5, 6]])
>>> b
array([[ 1. ,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

С переопределением:

```
>>>   b   =   np.array([[1,   2,   3],   [4,   5,   6]],
dtype=np.complex)
>>> b
array([[ 1.0+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+0.j]])
```

Функция numpy.array() не единственная функция для создания массивов. В большинстве случаев, элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с определённым исходным содержимым (по умолчанию тип создаваемого массива — float64).

Функция numpy.zeros() создает массив из нулей, а функция numpy.ones() — массив из единиц. Обе функции принимают кортеж с размерами, и аргумент dtype:

```
>>> np.zeros((3, 5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> np.ones((2, 2, 2))
array([[[ 1.,  1.],
```

```
        [ 1.,  1.]],
       [[ 1.,  1.],
        [ 1.,  1.]]])
```

Функция numpy.empty() позволяет создать массив без заполнения. Его содержимое будет случайно и зависит от состояния памяти на момент создания массива:

```
>>> np.empty((3, 3))
array([[  6.93920481e-310,   6.93920481e-310,   6.93920145e-310],
       [  6.93920058e-310,   6.93920058e-310,   6.93920018e-310],
       [  6.93920259e-310,   0.00000000e+000,   6.93920561e-310]])
>>> np.empty((3, 3))
array([[  6.93920481e-310,   6.93920481e-310,   6.93920142e-310],
       [  6.93920149e-310,   6.93920146e-310,   6.93920319e-310],
       [  6.93920259e-310,   0.00000000e+000,   3.95252567e-322]])
```

Для создания последовательностей чисел, в NumPy имеется функция numpy.arange(), аналогичная встроенной в Python range(), только вместо списков она возвращает массивы, и принимает не только целые значения:

```
>>> np.arange(10, 40, 5)
array([10, 15, 20, 25, 30, 35])
>>> np.arange(0, 0.5, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4])
```

Как известно, в вычислительной технике есть ряд определённых проблем, связанных с ограничения точности чисел с плавающей запятой. В связи с этим, сложно быть уверенным в том, сколько элементов будет получено при использовании дробных чисел при использовании функции numpy.arange(). Поэтому, в таких случаях обычно лучше использовать функцию numpy.linspace(), которая вместо шага в качестве одного из аргументов принимает число, равное количеству требуемых элементов массива:

```
>>> np.linspace(0, 2, 9)
```

```
array([  0.  ,   0.25,   0.5 ,   0.75,  1.  ,  1.25,  1.5 ,
1.75,  2.  ])
```

Говоря об отображении массивов, если массив слишком большой, чтобы его печатать, NumPy автоматически скрывает центральную часть массива и выводит только несколько первых и последних элементов:

```
>>> print(np.arange(0, 5000, 1))
[   0    1    2 ..., 4997 4998 4999]
```

Если вам действительно нужно увидеть весь массив, используйте функцию numpy.set_printoptions():

```
np.set_printoptions(threshold=np.nan)
```

Если вам требуется что-то большее, то возможно стоит посмотреть официальную документацию проекта NumPy: https://numpy.org/doc/stable/reference/

2.1.3 Базовые операции над массивами

Математические операции над массивами выполняются поэлементно. Создается новый массив, который заполняется результатами действия оператора.

```
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> a + b
array([20, 31, 42, 53])
>>> a - b
array([20, 29, 38, 47])
>>> a * b
array([  0,  30,  80, 150])
>>> a / b
array([         inf,  30.        ,  20.        ,  16.66666667])
```

```
<string>:1: RuntimeWarning: divide by zero encountered in true_divide
>>> a ** b
array([     1,      30,    1600, 125000])
>>> a % b
<string>:1: RuntimeWarning: divide by zero encountered in remainder
array([0, 0, 0, 2])
```

Для того, чтобы было возможно выполнить операции, массивы должны быть одинаковых размерностей:

```
>>> c = np.array([[1, 2, 3], [4, 5, 6]])
>>> d = np.array([[1, 2], [3, 4], [5, 6]])
>>> c + d
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,3) (3,2)
```

Также можно производить математические операции между массивом и числом. В таком случае над каждым элементом массива будет выполняться выбранная операция с числом:

```
>>> a + 1
array([21, 31, 41, 51])
>>> a ** 3
array([  8000,  27000,  64000, 125000])
>>> a < 35
array([ True,  True, False, False], dtype=bool)
```

NumPy также предоставляет множество математических операций для массивов:

```
>>> np.cos(a)
array([ 0.40808206,  0.15425145, -0.66693806,  0.96496603])
>>> np.arctan(a)
```

```
array([ 1.52083793,  1.53747533,  1.54580153,  1.55079899])
```

Все математические операции, которые можно использовать, можно найти в документации:

https://numpy.org/doc/stable/reference/routines.math.html

Унарные операции, такие как, например, вычисление суммы всех элементов массива, представлены также и в виде методов класса ndarray:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.sum(a)
21
>>> a.sum()
21
>>> a.min()
1
>>> a.max()
6
```

По умолчанию, эти операции применяются ко всему массиву, как если бы он, условно, был обычным списком чисел, независимо от его размерности. Однако, если указать параметр *axis*, можно применить операцию для указанной оси массива:

```
>>> a.min(axis=0)
array([1, 2, 3])
>>> a.min(axis=1)
array([1, 4])
```

2.1.4 Индексы, срезы, итерации

Одномерные массивы осуществляют операции индексирования, срезов и итераций очень схожим образом с обычными списками и другими последовательностями Python (кроме операции удаления):

```
>>> a = np.arange(10) ** 3
>>> a
```

```
array([  0,    1,    8,   27,   64,  125,  216,  343,  512,
729])
>>> a[1]
1
>>> a[3:7]
array([ 27,  64, 125, 216])
>>> a[3:7] = 8
>>> a
array([  0,    1,    8,    8,    8,    8,    8,  343,  512,
729])
>>> a[::-1]
array([729,                                              512,
343,   8,   8,   8,   8,   8,   1,   0])
```

У многомерных массивов на каждую ось приходится один индекс. Индексы передаются в виде последовательности чисел, разделенных запятыми:

```
>>> b = np.array([[  0, 1, 2, 3],
...               [10, 11, 12, 13],
...               [20, 21, 22, 23],
...               [30, 31, 32, 33],
...               [40, 41, 42, 43]])
...
>>> b[2,3]
23
>>> b[(2,3)]
23
>>> b[2][3]
23
>>> b[:,2]
array([ 2, 12, 22, 32, 42])
>>> b[:2]
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
>>> b[1:3, : : ]
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Когда индексов меньше, чем осей, отсутствующие индексы предполагаются дополненными с помощью срезов. К примеру, следующий код выведет последнюю строку:

```
>>> b[-1]
array([40, 41, 42, 43])
```

Данная запись будет эквивалентна:

```
>>> b[-1,:]
array([40, 41, 42, 43])
```

Ещё несколько примеров:

```
>>> a = np.array(([[0, 1, 2], [10, 12, 13]], [[100,
101, 102], [110, 112, 113]]))
>>> a.shape
(2, 2, 3)
>>> a[1, ...]
array([[100, 101, 102],
       [110, 112, 113]])
>>> a[1, : , :]
array([[100, 101, 102],
       [110, 112, 113]])
>>> a[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[... ,2]
array([[  2,  13],
       [102, 113]])
>>> a[: , : ,2]
array([[  2,  13],
       [102, 113]])
```

Говоря об итерировании, в многомерных массивах оно начинается с первой оси:

```
>>> for row in a:
...     print(row)
...
[[ 0   1   2]
 [10 12 13]]
[[100 101 102]
 [110 112 113]]
```

Однако, если нужно перебрать поэлементно весь массив, как если бы он был одномерным, для этого можно использовать атрибут flat:

```
>>> for el in a.flat:
...     print(el)
...
0
1
2
10
12
13
100
101
102
110
112
113
```

2.1.4 Манипуляции с формой (размерностью)

Как уже говорилось, у массива есть форма (shape), определяемая числом элементов вдоль каждой оси:

```
>>> a
array([[[  0,   1,   2],
        [ 10,  12,  13]],

       [[100, 101, 102],
        [110, 112, 113]]])
>>> a.shape
(2, 2, 3)
```

Форма массива может быть изменена с помощью различных команд.

Получение “плоского” массива:

```
>>> a.ravel()
array([  0,   1,   2,  10,  12,  13, 100, 101, 102,
110, 112, 113])
```

Изменение формы массива:

```
>>> a.shape = (6, 2)
>>> a
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])
```

Или может быть другой вариант:

```
>>> a.reshape((3, 4))
array([[  0,   1,   2,  10],
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

Транспонирование массива:

```
>>> a.transpose()
array([[  0,   2,  12, 100, 102, 112],
       [  1,  10,  13, 101, 110, 113]])
```

2.1.5 Объединение массивов

Несколько массивов могут быть объединены вместе вдоль разных осей с помощью функций numpy.hstack() и numpy.vstack().

numpy.hstack() объединяет массивы по первым осям, numpy.vstack() — по последним:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> np.vstack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.hstack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Функция column_stack() объединяет одномерные массивы в качестве столбцов двумерного массива:

```
>>> np.column_stack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Аналогично для строк имеется функция row_stack():

```
>>> np.row_stack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

2.1.6 Разбиение массива

Используя numpy.hsplit() можно разбить массив вдоль горизонтальной оси, указав либо число возвращаемых массивов одинаковой формы, либо номера столбцов, после которых массив разрезается:

```
>>> a = np.arange(12).reshape((2, 6))
```

```
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> np.hsplit(a, 3)
[array([[0, 1], [6, 7]]),
 array([[2, 3], [8, 9]]),
 array([[ 4,  5], [10, 11]])]
>>> np.hsplit(a, (3, 4))
[array([[0, 1, 2], [6, 7, 8]]),
 array([[3], [9]]),
 array([[ 4,  5], [10, 11]])]
```

Также существует функция numpy.vsplit(), которая разбивает массив вдоль вертикальной оси, и numpy.array_split(), которая позволяет указать оси, вдоль которых произойдет разбиение.

2.1.7 Копии и представления

При работе с массивами, их данные иногда необходимо копировать в другой массив. Часто, процесс копирования вызывает сложности, поскольку возможно 3 случая:

- Простое присваивание не создает ни копии массива, ни копии его данных:

```
>>> a = np.arange(12)
>>> b = a
>>> b is a
True
>>> b.shape = (3,4)
>>> a.shape
(3, 4)
```

- Разные объекты массивов могут использовать одни и те же данные.

Метод view() создает новый объект массива, являющийся представлением тех же данных:

```
>>> c = a.view()
>>> c is a
```

```
False
>>> c.base is a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = (2,6)
>>> a.shape
(3, 4)
>>> c[0,4] = 1234
>>> a
array([[   0,  1,  2,  3],
       [1234,  5,  6,  7],
       [   8,  9, 10, 11]])
```

Вспоминая срезы массивов, они как раз будут являться представлением:

```
>>> s = a[:,1:3]
>>> s[:] = 10
>>> a
array([[   0, 10, 10,  3],
       [1234, 10, 10,  7],
       [   8, 10, 10, 11]])
```

- Метод copy() создаст настоящую копию массива и его данных:

```
>>> d = a.copy()
>>> d is a
False
>>> d.base is a
False
>>> d[0, 0] = 9999
>>> a
array([[   0, 10, 10,  3],
       [1234, 10, 10,  7],
       [   8, 10, 10, 11]])
```

2.1.8 Операции из линейной алгебры

linalg.matrix_power(M, n) - возводит матрицу в степень n;

linalg.cholesky(a) - разложение Холецкого;

linalg.qr(a[, mode]) - QR разложение;

linalg.svd(a[, full_matrices, compute_uv]) - сингулярное разложение;

linalg.eig(a) - собственные значения и собственные векторы;

linalg.norm(x[, ord, axis]) - норма вектора или оператора;

linalg.cond(x[, p]) - число обусловленности;

linalg.det(a) – определитель;

linalg.slogdet(a) - знак и логарифм определителя;

linalg.solve(a, b) - решение системы линейных уравнений Ax = b;

linalg.tensorsolve(a, b[, axes]) - решение тензорной системы линейных уравнений Ax = b;

linalg.lstsq(a, b[, rcond]) - метод наименьших квадратов;

linalg.inv(a) - обратная матрица.

Полный список операций линейной алгебры можно посмотреть на сайте официальной документации:

https://numpy.org/doc/stable/reference/routines.linalg.html

2.2 Scikit-learn

Scikit-learn – библиотека машинного обучения библиотека с открытым исходным кодом для языка программирования Python.

Scikit-learn – одна из наиболее применяемых библиотек для Data Science и Machine Learning. Также, имеет отличную документацию о своих методах и функциях, а также описание всех реализованных алгоритмов.

Библиотека спроектирована таким образом, что позволяет работать как со встроенными типами данных языка Python, так и научными библиотеками NumPy, SciPy.

Основная часть библиотеки написана на языке Python, однако в вычислениях с массивами и линейной алгеброй, для повышения производительности используется библиотеке NumPy. Также, некоторые алгоритмы написаны на Cython, для повышения получения большей скорости вычисления.

Библиотека не предназначена для загрузки, обработки и визуализации данных. Эти задачи выполняются другими библиотеками: Pandas и NumPy. Библиотека Scikit-learn, в первую очередь, специализируется на алгоритмах машинного обучения для решения задач обучения.

Библиотека имеет реализации различных методов, основные из которых:

- **Линейные**: модели, задача которых построить разделяющую (для классификации) или аппроксимирующую (для регрессии) плоскость.
- **Метрические**: модели, которые вычисляют расстояние по одной из метрик между объектами выборки, и принимают решения в зависимости от этого расстояния (K ближайших соседей).
- **Деревья решений**: обучение моделей, в основе которых лежит множество условий, которые были оптимально подобраны для решения задачи.
- **Ансамблевые методы**: методы, основанные на деревьях решений, которые берут сразу множество деревьев, и таким образом повышают их качество работы и позволяют производить отбор признаков (Gradient boosting, бэггинг, Random forest, мажоритарное голосование).
- **Нейронные сети**: комплексный нелинейный метод для задач регрессии и классификации.
- **Опорные векторы (SVM)**: нелинейный метод, который обучается определять границы принятия решений.
- **Наивный Байес**: прямое вероятностное моделирование для задач классификации.
- **PCA**: линейный метод понижения размерности и отбора признаков.
- **t-SNE**: нелинейный метод понижения размерности.

- **K-средних**: наиболее распространенный метод кластеризации, получающий число кластеров, по которым требуется распределить данные.
- **Кросс-валидация**: метод, при котором для обучения используется весь датасет, но сам процесс обучения производится много раз, и в качестве выборки для валидации на каждом шаге выступают различные части этого же датасета. Конечным результатом является усреднение полученных промежуточных результатов.
- **Grid Search**: метод для нахождения оптимальных параметров модели путем построения сетки (grid) из значений параметров и последовательного обучения моделей со всеми возможными комбинациями параметров из сетки.

Это лишь небольшая часть, она содержит в себе и другие различные виды алгоритмов классификации, регрессии и кластеризации, реализации метода опорных векторов (SVM), Random forest, Gradient boosting, метода k-средних, DBSCAN. Scikit-learn также содержит функции для расчета значений метрик, выбора моделей, препроцессинга данных и другие.

2.3.1 Начало работы

Рассмотрены будут основные методы и их главные разновидности.

Всё это можно найти в официальной документации:

https://scikit-learn.org/

2.3.2 Линейные методы

Как самый простой, рассмотрим обычный метод наименьших квадратов (OLS). Принцип основан на том, что линейная регрессия подгоняет линейную модель с коэффициентами $w=(w_1,\dots,w_p)$ к минимизации остаточной суммы квадрата между наблюдаемого целевого признака в наборе данных и предсказано целевого признака по линейной аппроксимации.

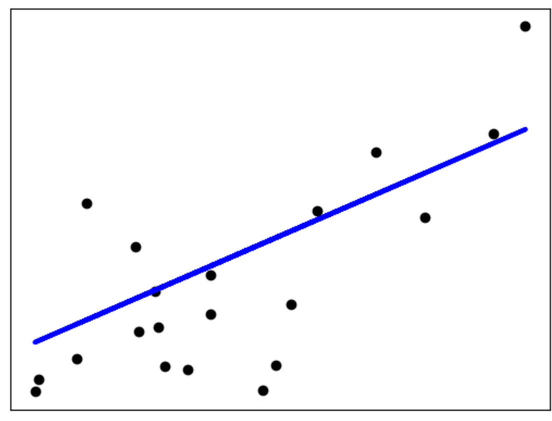


Рисунок 1. Пример линейной регрессии

Класс LinearRegression имеет метод fit() который принимает матрицу X и целевой признак y и будет хранить коэффициенты линейной модели в переменной coef_:

```
>>> from sklearn import linear_model
>>> reg = linear_model.LinearRegression()
>>> reg.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
>>> print("Коэффициенты:", regr.coef_)
```

Оценки коэффициентов для обыкновенных наименьших квадратов полагаются на независимость функций. Если столбцы матрицы плана имеют приблизительную линейную зависимость, матрица плана становится близкой к сингулярной, и в результате оценка данным методом становится очень чувствительной к случайным ошибкам в наблюдаемой цели, что приводит к большой дисперсии. Эта ситуация может возникнуть, например, когда данные собираются без экспериментального плана.

2.3.3 Метрические методы

sklearn.neighbors предоставляет функциональные возможности для методов обучения на основе ближайших соседей.

Принцип, лежащий в основе методов ближайшего соседа, состоит в том, чтобы найти определенное количество обучающих выборок, ближайших по расстоянию к выбранной точке, и предсказать метку по ним. Количество выборок может быть задано пользователем (обучение k-ближайшего соседа)

или изменяться в зависимости от локальной плотности точек (обучение соседей на основе радиуса). Расстояние может быть любой метрической мерой.

Несмотря на свою простоту, метод ближайших соседей успешно справляется с огромным количеством задач классификации и регрессии, включая рукописные цифры и кадры со спутников.

Классы sklearn.neighbors могут обрабатывать scipy.sparse в качестве входных данных либо массивы NumPy, либо матрицы. Для плотных матриц поддерживается большое количество возможных метрик расстояния.

Есть много программ обучения, которые в своей основе полагаются на ближайших соседей. Одним из примеров является оценка плотности ядра.

2.3.3.1 Неконтролируемые ближайшие соседи

NearestNeighbors реализует неконтролируемое обучение ближайших соседей. Он действует как единый интерфейс сразу для трех различных алгоритмов: BallTree, KDTree и перебор, основанный на sklearn.metrics.pairwise. Выбор алгоритма поиска соседей может быть задан через параметр ‘algorithm’, который должно иметь значение из списка: ‘auto’, ‘ball_tree’, ‘kd_tree’, ‘brute’. Когда задаётся значение ‘auto’, алгоритм пытается сам определить наилучший подход на основе данных.

```
>>> from sklearn.neighbors import NearestNeighbors
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 
1], [2, 1], [3, 2]])
>>> nbrs = NearestNeighbors(n_neighbors=2, 
algorithm='ball_tree').fit(X)
>>> distances, indices = nbrs.kneighbors(X)
>>> indices
array([[0, 1],
       [1, 0],
       [2, 1],
       [3, 4],
       [4, 3],
       [5, 4]]...)
```

```
>>> distances
array([[0.        , 1.        ],
       [0.        , 1.        ],
       [0.        , 1.41421356],
       [0.        , 1.        ],
       [0.        , 1.        ],
       [0.        , 1.41421356]])
```

2.3.3.2 Классификация ближайших соседей

Классификация на основе ближайших соседей — это тип обучения на основе экземпляров: не требуется строить общую внутреннюю модель, а можно просто сохранить экземпляры данных. Классификация выполняется обычным большинством ближайших соседей каждой точки: точке запроса назначается класс данных, который имеет наибольшее количество представителей среди ближайших соседей точки.

Scikit-learn реализует два разных классификатора ближайших соседей:

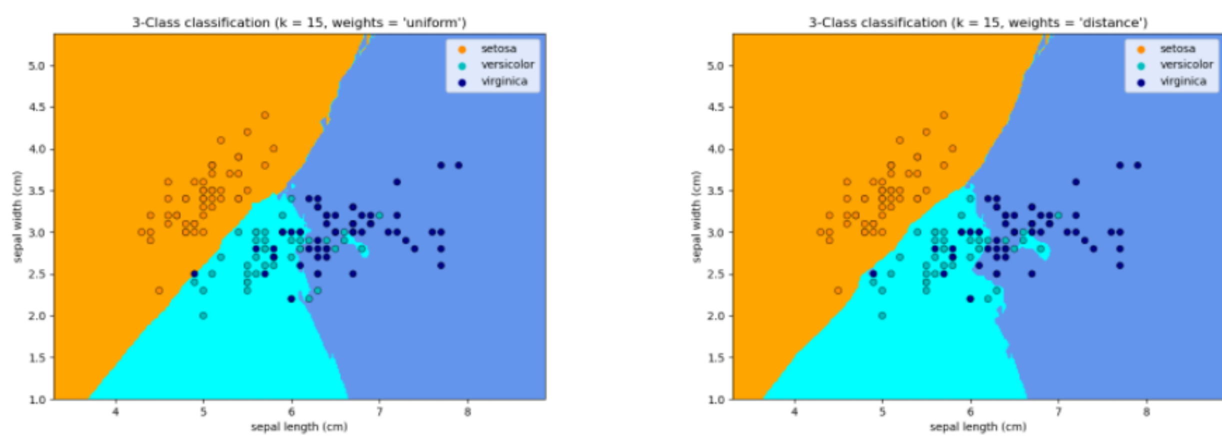
- KNeighborsClassifier реализует обучение на основе ближайших соседей каждой точки запроса, где k - целочисленное значение, указанное пользователем.
- RadiusNeighborsClassifier реализует обучение на основе количества соседей в фиксированном радиусе r каждой тренировочной точки, где r - значение с плавающей запятой, указанное пользователем.

Классификация соседей KNeighborsClassifier — наиболее часто используемый метод. Оптимальный выбор k зависит от данных: как правило, большое значение k подавляет шум, но делает границы классификации более неточными.

В случаях, когда выборка данных неравномерна, может быть лучше классификация соседей на основе радиуса. При этом указывается фиксированный радиус r, так что точки в более разреженных окрестностях используют меньшее количество ближайших соседей для классификации.

Базовая классификация ближайших соседей использует одинаковые веса: то есть значение, присвоенное точке запроса, вычисляется простым

большинством значений ближайших соседей. В некоторых случаях лучше взвесить соседей таким образом, чтобы более близкие соседи вносили больший вклад. Это можно сделать с помощью параметра weights. По умолчанию (weights = ‘uniform’) каждому соседу присваиваются одинаковые веса. Если же weights = ‘distance’, веса назначаются пропорциональные обратному расстоянию от точки запроса.



Метрические методы имеют гораздо большее разнообразие, нежели было представлено выше. Для ознакомления с полным списком, следует посетить официальную документацию.

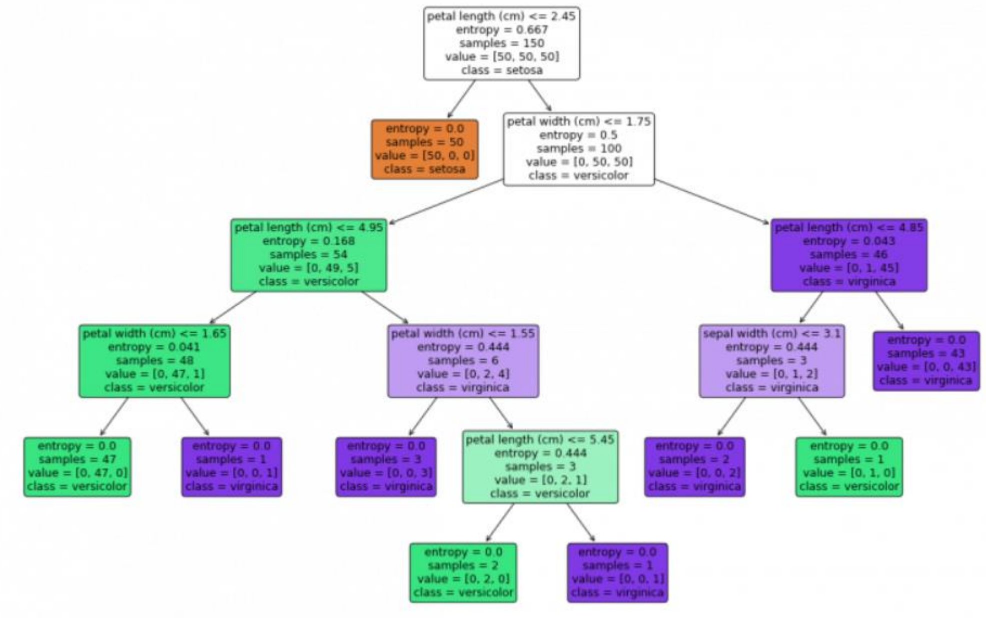
2.3.4 Деревья решений

Деревья решений в задачах классификации и регрессии достаточно просты в использовании, говоря о Scikit-Learn. После инициализации дерева решений для классификации, сразу можно проводить обучение модели через метод fit(X, Y), где X — обучающая выборка в формате массива NumPy, а Y — массив целевых значений, также в формате массива NumPy.

Scikit-Learn также позволяет визуализировать построенное дерево. Для этого есть несколько параметров, которые помогут визуализировать узлы принятия решений и разбить модель, что имеет пользу в понимании того, как она работает. В примере кода, будет использоваться обучающая выборка под названием “Ирисы Фишера”. Узлы будут раскрашены на основе имен признаков и отображать информацию о классе и объектах каждого узла.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier,
plot_tree
iris = load_iris()
plt.figure(figsize=((20,13)))
clf = DecisionTreeClassifier()
clf = clf.fit(iris.data, iris.target)
plot_tree(clf,
          filled=True,
          feature_names=iris.feature_names,
          class_names=iris.target_names,
          rounded=True)
plt.show()
```

Результат визуализации:



2.3.5 Метод опорных векторов (SVM)

Метод опорных векторов (Support Vector Machines) — это набор контролируемых методов обучения, используемых для классификации, регрессии и обнаружения выбросов.

Преимущества:

- Эффективен в пространствах больших размеров;

- По-прежнему эффективен в случаях, когда количество измерений превышает количество выборок;
- Использует подмножество обучающих точек в функции принятия решений (называемых опорными векторами), поэтому это также эффективно с точки зрения памяти;
- Универсальность: для функции принятия решения могут быть заданы различные функции ядра. Предусмотрены общие ядра, но можно задать и собственные ядра.

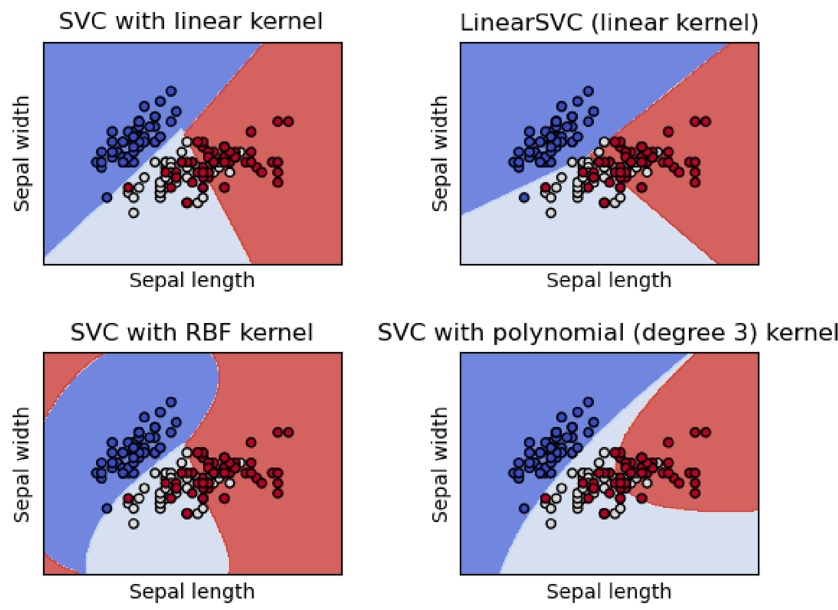
К недостаткам опорных векторных машин можно отнести:

- Если число признаков намного больше числа выборок, то при выборе функций ядра и члена регуляризации очень важно избегать чрезмерного соответствия;
- SVM не предоставляют оценки вероятности напрямую, они рассчитываются с помощью дорогостоящей пятикратной кросс-валидации.

Метод опорных векторов в scikit-learn поддерживают плотные (*numpy.ndarray*) и разреженные (*scipy.sparse*) выборочные векторы в качестве входных данных.

2.3.5.1 Классификация

SVC, NuSVC и LinearSVC являются классами, способными выполнять двоичную и мультиклассовую классификацию набора данных.



SVC и NuSVC являются аналогичными методами, но принимают несколько разные наборы параметров и имеют разные математические формулировки. С другой стороны, LinearSVC это еще одна (более быстрая) реализация классификации опорных векторов для случая линейного ядра. Обратите внимание, что LinearSVC параметр не принимает kernel, так как предполагается, что он линейный. Ему также не хватает некоторых атрибутов SVC и NuSVC, например support_.

Как и другие классификаторы SVC, NuSVC и LinearSVC в качестве входных двух массивов: массив X формы (n_samples, n_features), проведение обучающих выборок, и массив y из класса меток (строки или целые числа), в форме (n_samples):

```
>>> from sklearn import svm
>>> X = [[0, 0], [1, 1]]
>>> y = [0, 1]
>>> clf = svm.SVC()
>>> clf.fit(X, y)
SVC()
```

После обучения модель можно использовать для прогнозирования новых значений:

```
>>> clf.predict([[2., 2.]])
array([1])
```

Функция принятия решения SVM зависит от некоторого подмножества обучающих данных, называемых опорными векторами. Некоторые свойства этих опорных векторов можно найти в атрибутах support_vectors_, support_a также n_support_:

```
>>> clf.support_vectors_
array([[0., 0.],
       [1., 1.]])
>>> clf.support_
array([0, 1]...)
>>> clf.n_support_
array([1, 1]...)
```

2.3.5.2 Регрессия

Метод классификации опорных векторов может быть расширен для решения задач регрессии. Этот метод называется регрессией опорных векторов.

Модель, полученная с помощью классификации опорных векторов, зависит только от подмножества обучающих данных, потому что функция потерь для построения модели не заботится об обучающих точках, лежащих за пределами поля. Аналогично, модель, полученная с помощью регрессии опорных векторов, зависит только от подмножества обучающих данных, поскольку функция потерь игнорирует выборки, предсказание которых близко к цели.

Существует три различных реализации регрессии опорных векторов: SVR, NuSVR и LinearSVR. LinearSVR обеспечивает более быструю реализацию,

чем SVR, но учитывает только линейное ядро, а NuSVR реализует немного другую схему, нежели SVR и LinearSVR.

Как и в случае с классами классификации, метод соответствия принимает в качестве аргументов векторы X, y, только в этом случае y, как ожидается, будет иметь значения с плавающей точкой, вместо целочисленных:

```
>>> from sklearn import svm
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> regr = svm.SVR()
>>> regr.fit(X, y)
SVR()
>>> regr.predict([[1, 1]])
array([1.5])
```

2.3.6 Наивный Байес

Наивный Байес — это набор алгоритмов контролируемого обучения, основанных на применении теоремы Байеса с «наивным» предположением об условной независимости между каждой парой характеристик при заданном значении переменной класса.

Теорема Байеса утверждает следующее отношение, учитывая переменную класса y и вектор зависимых признаков x_1 до x_n:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n \mid y)}{P(x_1, \dots, x_n)}$$

Используя наивное предположение об условной независимости,

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y),$$

для всех i, это отношение упрощается до

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

$P(x_1, \dots, x_n)$ является константой с учетом входных данных, мы можем использовать следующее правило классификации:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

и мы можем использовать оценку Maximum A Posteriori (MAP) для оценки $P(y)$, а также $P(x_i \mid y)$; первое является относительной частотой встречаемости класса y в обучающем множестве.

Различные классификаторы наивного Байеса отличаются в основном предположениями, которые они делают по отношению к распределению $P(x_i \mid y)$.

Несмотря на свои, казалось бы, слишком упрощенные предположения, классификаторы наивного Байеса отлично работают во многих реальных ситуациях, например, в классификации документов и фильтрации спама. Они требуют небольшого количества обучающих данных для оценки необходимых параметров.

Методы обучения и классификаторы Наивного Байеса могут быть чрезвычайно быстрыми по сравнению с более сложными методами. Разделение условных распределений признаков по классам означает, что каждое распределение может быть независимо оценено как одномерное распределение. Это, в свою очередь, помогает облегчить решение проблем, вытекающих из проклятия размерности.

С другой стороны, хотя наивный Байес известен как хороший классификатор, он известен как плохой оценщик, поэтому вероятностные результаты predict_proba не стоит воспринимать слишком серьёзно.

2.3.6.1 Гауссовский наивный Байес (GNB)

GaussianNB реализует гауссовский наивный байесовский алгоритм для классификации. Предполагается, что вероятность появления признаков гауссова:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2})$$

Параметры σ_y и μ_y также оцениваются с использованием максимального подобия.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.naive_bayes import GaussianNB
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.5, random_state=0)
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(X_train, y_train).predict(X_test)
>>> print("Количество неправильно помеченных точек из общего
числа %d точек : %d"
...       % (X_test.shape[0], (y_test != y_pred).sum()))
Количество неправильно помеченных точек из общего числа 75
точек: 4
```

2.3.6.2 Мультиномиальный наивный Байес (MNB)

MultinomialNB реализует алгоритм наивного Байеса для мультиномиально распределенных данных и является одним из двух классических вариантов наивного Байеса, используемых в текстовой классификации (где данные обычно представлены в виде векторов количества слов, хотя известно, что TF-IDF векторы также хорошо работают). Распределение параметризуется векторами $\theta_y = (\theta_{y1}, \dots, \theta_{yn})$ для каждого класса y, где n - количество признаков (в текстовой классификации - размер словаря), а θ_{yi} - вероятность $P(x_i \mid y)$ появления признака i в выборке, принадлежащей классу y.

Параметры θ_y оцениваются с помощью сглаженной версии максимального подобия, т.е. методом подсчета относительной частоты:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

где $N_{yi} = \sum_{x \in T} x_i$ - количество раз, когда признак i встречается в выборке класса y в обучающем множестве T, а $N_y = \sum_{i=1}^{n} N_{yi}$ - общее количество всех признаков для класса y.

Сглаживающие значения приоритетов $\alpha \geq 0$ учитывают признаки, отсутствующие в обучающих выборках, и предотвращают нулевые вероятности при дальнейших вычислениях. Выбор параметра $\alpha = 1$ называется сглаживанием по Лапласу, а $\alpha < 1$ - сглаживанием по Лидстоуну.

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, y)
MultinomialNB()
>>> print(clf.predict(X[2:3]))
[3]
```

2.3.6.3 Дополнение наивного Байеса (CNB)

ComplementNB реализует алгоритм дополняющего наивного Байеса (CNB). CNB — это адаптация стандартного мультиномиального алгоритма наивного Байеса (MNB), который хорошо подходит для несбалансированных наборов данных. В частности, CNB использует статистику из дополнения каждого класса для вычисления весов модели. Изобретатели CNB эмпирически показали, что оценки параметров для CNB более стабильны, чем для MNB. Кроме того, CNB регулярно превосходит MNB (часто со значительным отрывом) в задачах классификации текстов. Процедура расчета весов выглядит следующим образом:

$$\hat{\theta}_{ci} = \frac{\alpha_i + \sum_{j:y_j \neq c} d_{ij}}{\alpha + \sum_{j:y_j \neq c} \sum_k d_{kj}}$$

$$w_{ci} = \log \hat{\theta}_{ci}$$

$$w_{ci} = \frac{w_{ci}}{\sum_j |w_{cj}|}$$

где суммируются все тексты j, не входящие в класс c, d_{ij} - количество или tf-idf значение термина i в тексте j, α_i - сглаживающий гиперпараметр, подобный тому, который используется в MNB, и $\alpha = \sum_i \alpha_i$. Вторая нормализация учитывает тенденцию преобладания более длинных текстов над оценками параметров в MNB. Правило классификации следующее:

$$\hat{c} = \arg\min_c \sum_i t_i w_{ci}$$

т.е. текст относится к тому классу, который хуже всего соответствует дополнению.

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import ComplementNB
>>> clf = ComplementNB()
>>> clf.fit(X, y)
ComplementNB()
>>> print(clf.predict(X[2:3]))
[3]
```

2.3.6.4 Наивный Байес Бернулли (BNB)

BernoulliNB реализует алгоритмы обучения и классификации наивного Байеса для данных, распределенных в соответствии с многомерным распределением Бернулли; т.е. может быть несколько признаков, но каждый из них предполагается бинарной (бернуллиевской, булевой) переменной. Поэтому данный класс требует, чтобы выборки были представлены в виде

бинарных векторов признаков; при передаче любого другого типа данных экземпляр BernoulliNB может бинаризировать свои входные данные (в зависимости от параметра binarize).

Правило принятия решения для наивного Байеса Бернулли основано на

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

которое отличается от мультиномиального правила NB тем, что оно явно наказывает за непоявление признака i, который является индикатором для класса y, в то время как мультиномиальный вариант просто игнорирует непоявление признака.

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, Y)
BernoulliNB()
>>> print(clf.predict(X[2:3]))
[3]
```

2.3.7 Стохастический градиентный спуск (SGD)

Стохастический градиентный спуск (Stochastic Gradient Descent) — это простой, но очень эффективный подход к подгонке линейных классификаторов и регрессоров под выпуклые функции потерь, такие как метод опорных векторов и логистическая регрессия. Несмотря на то, что SGD существует в сообществе машинного обучения уже давно, совсем недавно он привлек значительное внимание в контексте крупномасштабного обучения.

SGD успешно применяется для решения крупномасштабных и разреженных задач машинного обучения, часто встречающихся при классификации текста и обработке естественного языка (NLP). Учитывая, что данные немногочисленны, классификаторы в этом модуле легко масштабируются

для решения задач с более чем 10^5 обучающими примерами и более чем 10^5 признаками.

Строго говоря, SGD — это всего лишь метод оптимизации и не соответствует определенному семейству моделей машинного обучения. Это лишь способ обучения модели. Часто экземпляр SGDClassifier или SGDRegressor имеет эквивалентный оценщик в Scikit-learn API, потенциально использующий другую технику оптимизации. Например, использование SGDClassifier(loss='log') приводит к логистической регрессии, т.е. модели, эквивалентной LogisticRegression, которая подгоняется с помощью SGD вместо того, чтобы быть подогнанной одним из других решателей LogisticRegression. Аналогично, SGDRegressor(loss='squared_error', penalty='l2') и Ridge решают одну и ту же задачу оптимизации разными средствами.

Преимущества стохастического градиентного спуска:

- Эффективность;
- Простота реализации (множество возможностей для настройки кода).

К недостаткам стохастического градиентного спуска можно отнести:

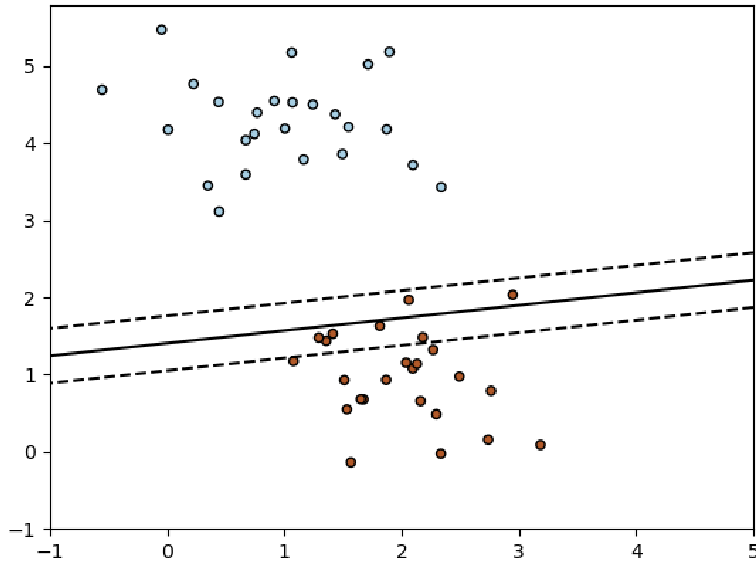
- SGD требует ряда гиперпараметров, таких как параметр регуляризации и количество итераций;
- SGD чувствителен к масштабированию функций.

Прежде чем выполнять обучение модели, убедитесь, что вы переставляете свои обучающие данные или используете *shuffle=True* для перемешивания после каждой итерации (используется по умолчанию). Кроме того, в идеале функции должны быть стандартизированы, например, с помощью make_pipeline(StandardScaler(), SGDClassifier()).

2.3.7.1 Классификация

Класс SGDClassifier реализует процедуру обучения при помощи стохастического градиентного спуска, которая поддерживает различные

функции потерь и наказания за классификацию. Ниже приведена граница принятия решения SGDClassifier обученного с функцией потери ‘hinge’, эквивалентного линейному SVM.



Как и другие классификаторы, SGD должен быть оснащен двумя массивами: массивом X формы (n_samples, n_features), содержащим обучающие образцы, и массивом y формы (n_samples,), содержащим целевые значения (метки классов) для обучающих образцов:

```
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = SGDClassifier(loss="hinge", penalty="l2",
max_iter=5)
>>> clf.fit(X, y)
SGDClassifier(max_iter=5)
```

После обучения модель может быть использована для прогнозирования новых значений:

```
>>> clf.predict([[2., 2.]])
array([1])
```

SGD подгоняет (fitting) линейную модель к обучающим данным. Атрибут coef_ содержит параметры модели:

```
>>> clf.coef_
array([[9.9..., 9.9...]])
```

Атрибут intercept_ содержит смещение (bias):

```
>>> clf.intercept_
array([-9.9...])
```

Следует ли использовать в модели смещение, а точнее смещенную гиперплоскость, контролируется параметром fit_intercept.

Знаковое расстояние до гиперплоскости (вычисляемое как произведение точек между коэффициентами и входной выборкой, плюс смещение) задается функцией SGDClassifier.decision_function:

```
>>> clf.decision_function([[2., 2.]])
array([29.6...])
```

Функция потерь может быть задана через параметр *loss*. SGDClassifier поддерживает следующие функции потерь:

- loss="**hinge**": (soft-margin) линейная Support Vector Machine;
- loss="**modified_huber**": сглаженная hinge loss;
- loss="**log**": логистическая регрессия;

и прочие регрессионные потери. В этом случае цель кодируется как -1 или 1, и задача рассматривается как задача регрессии. Тогда предсказанный класс соответствует знаку предсказанной цели.

Использование loss="log" или loss="modified_huber" позволяет использовать метод predict_proba, который дает вектор вероятностных оценок $P(y|x)$ на выборку x:

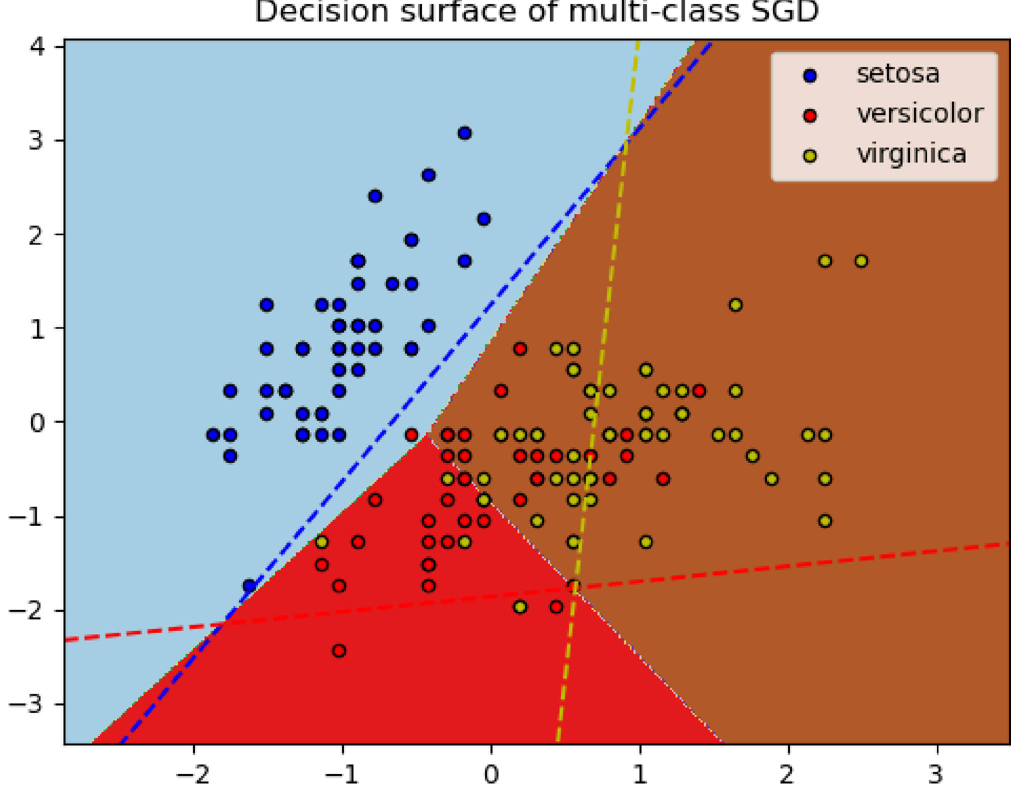
```
>>> clf = SGDClassifier(loss="log",
max_iter=5).fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([[0.00..., 0.99...]])
```

Конкретное наказание может быть задано через параметр *penalty*. SGD поддерживает следующие штрафы:

- penalty="**l2**": штраф по норме L2 на coef_;
- penalty="**l1**": штраф по норме L1 на coef_;
- penalty="**elasticnet**": Выпуклая комбинация L2 и L1; (1 - l1_ratio) * L2 + l1_ratio * L1.

По умолчанию установлено значение *penalty*="l2". Штраф по L1 приводит к разреженным решениям, сводя большинство коэффициентов к нулю. Elastic Net 11 решает некоторые недостатки штрафа L1 в присутствии сильно коррелированных атрибутов. Параметр l1_ratio управляет выпуклой комбинацией наказаний L1 и L2.

SGDClassifier поддерживает многоклассовую классификацию за счет объединения нескольких двоичных классификаторов по схеме "один против всех" (OVA). Для каждого из K классов обучается двоичный классификатор, который различает этот и все остальные $K-1$ классов. Во время тестирования мы вычисляем показатель достоверности (т.е. знаковые расстояния до гиперплоскости) для каждого классификатора и выбираем класс с наибольшей достоверностью. На рисунке ниже показан подход OVA на наборе данных “Ирисы Фишера”. Пунктирные линии представляют три классификатора OVA; цвета фона показывают поверхность принятия решения, полученную с помощью трех классификаторов.



В случае многоклассовой классификации coef_ - это двумерный массив формы (n_classes, n_features), а intercept_ - одномерный массив формы (n_classes,).

При этом, i-я строка coef_ содержит вектор весов классификатора OVA для i-го класса; классы индексируются в порядке возрастания (см. атрибут classes_). Обратите внимание, что в принципе, поскольку они позволяют создать вероятностную модель, *loss*="log" и *loss*="modified_huber" больше подходят для классификации "один против всех".

SGDClassifier поддерживает взвешенные классы и взвешенные экземпляры через параметры class_weight и sample_weight.

2.3.7.2 Регрессия

Класс SGDRegressor позволяет реализовать простую процедуру обучения методом стохастического градиентного спуска, которая поддерживает различные функции потерь и наказания для подгонки линейных регрессионных моделей. SGDRegressor хорошо подходит для задач регрессии

с большим количеством обучающих выборок (> 10.000), для других задач стоит посмотреть Ridge, Lasso или ElasticNet.

Функция потерь может быть задана с помощью параметра *loss*. SGDRegressor поддерживает следующие функции потерь:

- loss="**squared_error**": Обычные наименьшие квадраты;
- loss="**huber**": Huber loss для робастной регрессии;
- loss="**epsilon_insensitive**": линейная Support Vector Regression.

Параметр *penalty* определяет используемое наказание. Их варианты были описаны выше для классификации.

```
>>> import numpy as np
>>> from sklearn.linear_model import SGDRegressor
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> reg = make_pipeline(StandardScaler(),
...                     SGDRegressor(max_iter=1000, tol=1e-3))
>>> reg.fit(X, y)
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('sgdregressor', SGDRegressor())])
```

2.3.7.3 Онлайн одноклассовая SVM

Класс sklearn.linear_model.SGDOneClassSVM реализует онлайновую линейную версию одноклассовой SVM с использованием стохастического градиентного спуска. В сочетании с методами kernel аппроксимации sklearn.linear_model.SGDOneClassSVM можно использовать для аппроксимации решения кернелизированной одноклассовой SVM, реализованной в sklearn.svm.OneClassSVM, с линейной сложностью по числу выборок. Обратите внимание, что сложность кернелизированной одноклассовой SVM в лучшем случае квадратична по числу выборок.

sklearn.linear_model.SGDOneClassSVM, таким образом, вполне подходит для наборов данных с большим числом обучающих выборок (> 10 000), для которых вариант SGD может быть на несколько порядков быстрее.

Реализация этого метода основана на применении стохастического градиентного спуска. В действительности, исходная задача оптимизации одноклассовой SVM имеет вид

$$\min_{w,\rho,\xi} \quad \frac{1}{2}\|w\|^2 - \rho + \frac{1}{\nu n}\sum_{i=1}^{n}\xi_i$$
$$\text{s.t.} \quad \langle w, x_i \rangle \geq \rho - \xi_i 1 \leq i \leq n$$
$$\xi_i \geq 0 1 \leq i \leq n$$

где $\nu \in (0,1]$ - заданный пользователем параметр, управляющий долей выбросов и долей опорных векторов. Избавляясь от нежелательных переменных ξ_i эта задача эквивалентна

$$\min_{w,\rho}\frac{1}{2}\|w\|^2 - \rho + \frac{1}{\nu n}\sum_{i=1}^{n} max(0, \rho - \langle w, x_i \rangle).$$

Умножив на константу ν и введя перехват (intercept) $b = 1 - \rho$, мы получим следующую эквивалентную задачу оптимизации

$$\min_{w,b}\frac{\nu}{2}\|w\|^2 + b\nu + \frac{1}{n}\sum_{i=1}^{n} max(0,1 - (\langle w, x_i \rangle + b)).$$

Как и SGDClassifier и SGDRegressor, SGDOneClassSVM поддерживает усреднение SGD. Усреднение можно включить, установив *average*=True.

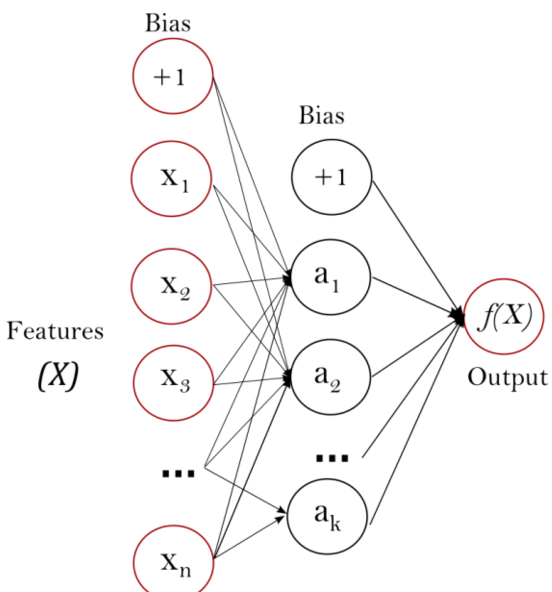
```
>>> import numpy as np
>>> from sklearn import linear_model
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> clf = linear_model.SGDOneClassSVM(random_state=42)
>>> clf.fit(X)
SGDOneClassSVM(random_state=42)
>>> print(clf.predict([[4, 4]]))
[1]
```

2.3.8 Модели нейронных сетей (с учителем)

Эта реализация не предназначена для крупномасштабных приложений. В частности, Scikit-learn не поддерживает расчёты на GPU.

2.3.8.1 Многослойный персептрон

Многослойный перцептрон (MLP) — это алгоритм обучения с учителем, который обучает функцию $f(\cdot): R^m \rightarrow R^o$ путем тренировки на наборе данных, где m - число измерений для входа, а n - число измерений для выхода. Имея набор признаков $X = x_1, x_2, \ldots, x_m$ и цель y, он может выучить аппроксиматор нелинейной функции для классификации или регрессии. Данный метод отличается от логистической регрессии тем, что между входным и выходным слоем может быть один или несколько нелинейных слоев, называемых скрытыми слоями. На рисунке ниже показан MLP с одним скрытым слоем и скалярным выходом.



Самый левый слой, известный как входной, состоит из набора нейронов $x_i | x_1, x_2, \ldots, x_m$ представляющие входные функции. Каждый нейрон в скрытом слое преобразует значения из предыдущего слоя с взвешенным линейным суммированием $w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$, за которой следует нелинейная функция активации $g(\cdot): R \rightarrow R$ — как гиперболическая функция tan. Выходной слой получает значения из последнего скрытого слоя и преобразует их в выходные значения.

Модуль содержит атрибуты coefs_ и intercepts_. coefs_ — это список матриц весов, где матрица весов по индексу представляет веса между слоем i и слоем $i + 1$. intercepts_ — это список векторов смещения (bias), где вектор по индексу i представляет значения смещения, добавленные к слою $i + 1$.

Преимущества многослойного перцептрона:

- Возможность изучать нелинейные модели;
- Возможность изучения моделей в режиме реального времени (онлайн-обучение) с использованием partial_fit.

К недостаткам многослойного персептрона (MLP) можно отнести:

- MLP со скрытыми слоями имеют невыпуклую функцию потерь, где существует более одного локального минимума. Поэтому различные случайные инициализации весов могут привести к различной точности валидации;
- MLP требует настройки ряда гиперпараметров, таких как количество скрытых нейронов, слоев и итераций;
- MLP чувствителен к масштабированию признаков.

2.3.8.2 Классификация

Класс MLPClassifier реализует алгоритм многослойного перцептрона (MLP), который обучается методом обратного распространения (Backpropagation).

MLP обучается на двух массивах: массив X размером (n_samples, n_features), который содержит обучающие выборки, представленные в виде векторов признаков с плавающей точкой; и массив y размером (n_samples,), который содержит целевые значения (метки классов) для обучающих выборок:

```
>>> from sklearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(5, 2), random_state=1)
>>> clf.fit(X, y)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(5, 2), random_state=1,
```

```
                    solver='lbfgs')
```

После обучения модель может предсказывать метки для новых образцов:

```
>>> clf.predict([[2., 2.], [-1., -2.]])
array([1, 0])
```

MLP может подгонять (fitting) нелинейную модель к обучающим данным. clf.coefs_ содержит весовые матрицы, составляющие параметры модели:

```
>>> [coef.shape for coef in clf.coefs_]
[(2, 5), (5, 2), (2, 1)]
```

В настоящее время MLPClassifier поддерживает только Cross-Entropy функцию потерь, которая позволяет оценивать вероятность путем запуска predict_proba метода.

MLP обучается с помощью обратного распространения. Точнее, он обучается с использованием некоторой формы градиентного спуска, а градиенты рассчитываются с помощью обратного распространения. Для классификации минимизируется функция потерь Cross-Entropy, дающая вектор вероятностных оценок $P(y|x)$ для выборки x:

```
>>> clf.predict_proba([[2., 2.], [1., 2.]])
array([[1.967...e-04, 9.998...-01],
       [1.967...e-04, 9.998...-01]])
```

MLPClassifier поддерживает многоклассовую классификацию, применяя Softmax в качестве выходной функции.

Кроме того, модель поддерживает многоклассовую классификацию, при которой образец может принадлежать более чем к одному классу. Для каждого класса исходный результат проходит через логистическую функцию. Значения, большие или равные 0,5, округляются до 1, в противном

случае - до 0. Для прогнозируемого выхода образца индексы, где значение равно 1, представляют назначенные классы этого образца:

```
>>> X = [[0., 0.], [1., 1.]]
>>> y = [[0, 1], [1, 1]]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(15,), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(15,), random_state=1,
              solver='lbfgs')
>>> clf.predict([[1., 2.]])
array([[1, 1]])
>>> clf.predict([[0., 0.]])
array([[0, 1]])
```

2.3.8.3 Регрессия

Класс MLPRegressor реализует многослойный перцептрон (MLP), который обучается методом обратного распространения без функции активации в выходном слое, что также можно рассматривать как использование функции тождественности в качестве функции активации. Таким образом, в качестве функции потерь используется квадратичная ошибка, а на выходе получается набор непрерывных значений.

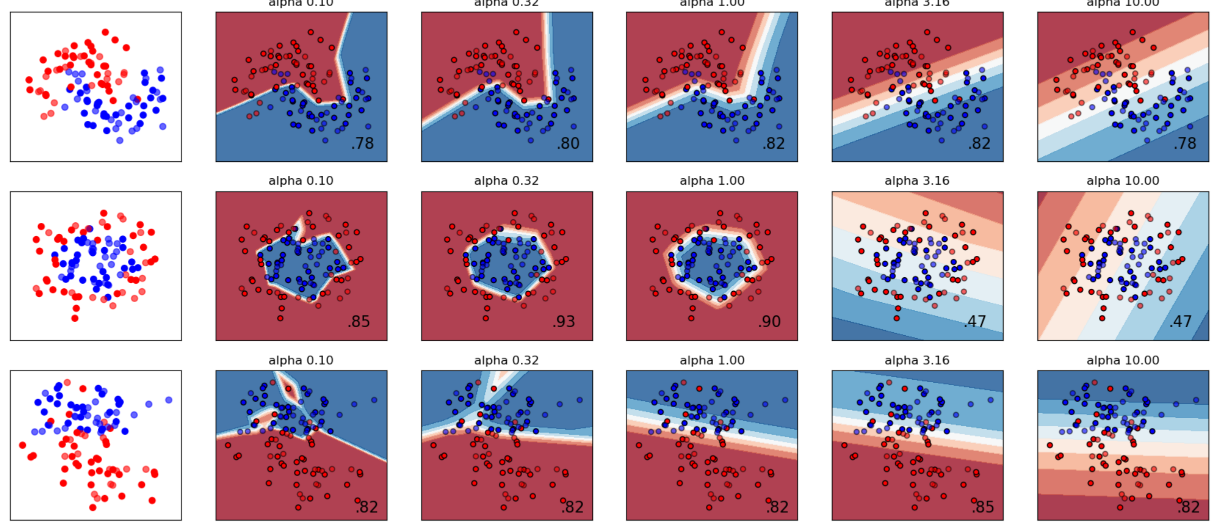
MLPRegressor также поддерживает регрессию с несколькими выходами, в которой выборка может иметь более одной цели.

```
>>> from sklearn.neural_network import MLPRegressor
>>> from sklearn.datasets import make_regression
>>> from sklearn.model_selection import
train_test_split
>>> X, y = make_regression(n_samples=200,
random_state=1)
>>> X_train, X_test, y_train, y_test =
train_test_split(X, y,
```

```
...
 random_state=1)
>>> regr = MLPRegressor(random_state=1,
max_iter=500).fit(X_train, y_train)
>>> regr.predict(X_test[:2])
array([-0.9..., -7.1...])
>>> regr.score(X_test, y_test)
0.4...
```

2.3.8.4 Регуляризация

MLPRegressor и MLPClassifier используют параметр alpha для регуляризации (L2-регуляризация), который помогает избежать переобучения, “наказывая” веса с большими величинами. Следующий график показывает изменение функции принятия решения в зависимости от значения параметра *alpha*.



2.3.9 Модели нейронных сетей (без учителя)

Данная группа методов не столь многочисленна, как с учителем, однако в ней тоже есть в чём разобраться.

2.3.9.1 Ограниченные машины Больцмана

Машины Больцмана с ограничениями (RBM) — это нелинейные обучающие функции без учителя, основанные на вероятностной модели. Признаки,

извлеченные с помощью RBM или иерархии RBM, часто дают хорошие результаты при подаче в линейный классификатор, такой как линейный SVM или перцептрон.

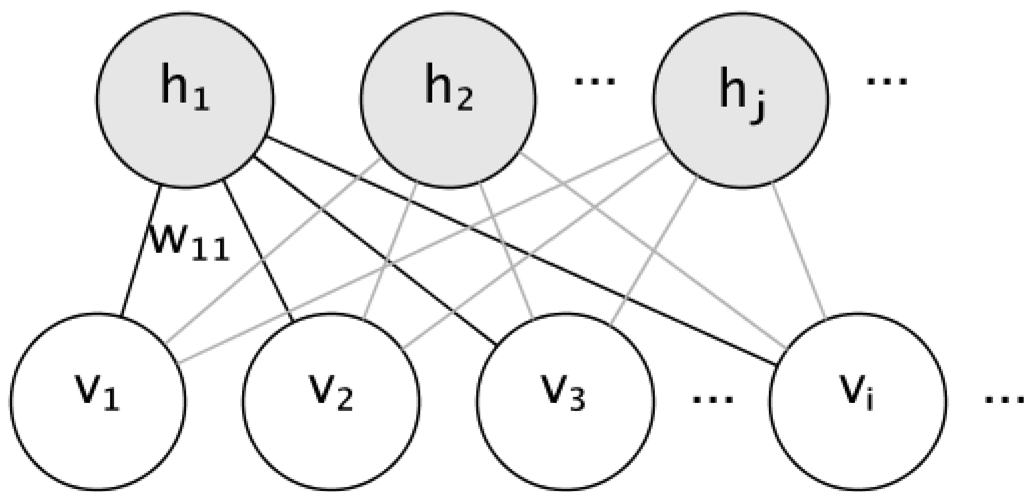
Модель делает предположения относительно распределения входов. На данный момент scikit-learn предоставляет только данные BernoulliRBM, предполагающие, что входными данными являются либо двоичные значения, либо значения от 0 до 1, каждое из которых кодирует вероятность того, что конкретная функция будет включена.

RBM пытается максимизировать вероятность получения данных с помощью конкретной графической модели. Используемый алгоритм обучения параметрам (стохастический максимум правдоподобия) предотвращает отклонение представлений от входных данных, что заставляет их фиксировать интересные закономерности, но делает модель менее полезной для небольших наборов данных и, как правило, бесполезной для оценки плотности.

Метод стал популярным для инициализации глубоких нейронных сетей с весами независимых RBM. Этот метод известен как предварительное обучение без учителя.

Графическая модель и параметризация

Графическая модель RBM — это полносвязный двудольный граф.



Узлы являются случайными переменными, состояние которых зависит от состояния других узлов, с которыми они соединены. Поэтому модель параметризуется весами связей, а также смещением для каждого видимого и скрытого узла, которое для простоты скрыто на изображении.

Энергетическая функция измеряет качество совместного задания:

$$E(\mathbf{v},\mathbf{h}) = -\sum_i \sum_j w_{ij} v_i h_j - \sum_i b_i v_i - \sum_j c_j h_j$$

В приведенной выше формуле $\mathbf{b}$ и $\mathbf{c}$ - векторы пересечения для видимого и скрытого слоев, соответственно. Совместная вероятность модели определяется в терминах энергии:

$$P(\mathbf{v},\mathbf{h}) = \frac{e^{-E(\mathbf{v},\mathbf{h})}}{Z}$$

Слово «ограниченный» относится к двусоставной структуре модели, которая запрещает прямое взаимодействие между скрытыми элементами или между видимыми элементами. Это означает, что предполагаются следующие условные зависимости:

$$h_i \perp h_j | \mathbf{v}$$
$$v_i \perp v_j | \mathbf{h}$$

Двудольная структура позволяет использовать для вывода эффективную блочную выборку Гиббса.

Ограниченные машины Больцмана Бернулли

В BernoulliRBM все единицы являются двоичными стохастическими единицами. Это означает, что входные данные должны быть либо двоичными, либо вещественными значениями от 0 до 1, означающими вероятность того, что видимая единица включится или выключится. Это хорошая модель для распознавания символов, где интерес представляет то, какие пиксели активны, а какие нет. Для изображений естественных сцен она уже не подходит из-за фона, глубины и тенденции соседних пикселей принимать одинаковые значения.

Условное распределение вероятности каждого блока задается логистической сигмоидной функцией активации входного сигнала, который он получает:

$$P(v_i = 1|\mathbf{h}) = \sigma(\sum_j w_{ij}h_j + b_i)$$

$$P(h_i = 1|\mathbf{v}) = \sigma(\sum_i w_{ij}v_i + c_j)$$

где σ - логистическая сигмовидная функция:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Стохастическое обучение с максимальным правдоподобием

Реализованный алгоритм обучения BernoulliRBM известен как стохастическая максимальная вероятность (SML) или стойкая контрастная дивергенция (PCD). Прямая оптимизация максимальной вероятности невозможна из-за формы вероятности данных:

$$\log P(v) = \log \sum_h e^{-E(v,h)} - \log \sum_{x,y} e^{-E(x,y)}$$

Для простоты уравнение выше написано для одного обучающего примера. Градиент по отношению к весам складывается из двух членов, соответствующих вышеприведенным. Их обычно называют положительным и отрицательным градиентами из-за их соответствующих знаков. В данной реализации градиенты оцениваются по мини-сериям (mini-batches) образцов.

При максимизации логарифмического правдоподобия положительный градиент заставляет модель предпочитать скрытые состояния, которые совместимы с наблюдаемыми данными обучения. Благодаря двухсторонней структуре RBM он может быть вычислен эффективно. Отрицательный градиент, напротив, является трудновычислимым. Его цель - снизить энергию совместных состояний, которые предпочитает модель, тем самым заставляя ее оставаться соответствующей данным. Он может быть аппроксимирован цепью Маркова Монте-Карло с использованием блочной выборки Гиббса путем итеративной выборки каждого из v и h с учетом остальных, пока цепь не перемешается. Образцы, сгенерированные таким образом, иногда называют фантазийными частицами. Это неэффективно, и трудно определить, перемешалась ли цепь Маркова.

Метод контрастной дивергенции (PCD) предполагает остановку цепочки после небольшого числа итераций, k, обычно даже 1. Этот метод быстрый и имеет низкую дисперсию, но выборки далеки от распределения модели.

Устойчивая контрастная дивергенция решает эту проблему. Вместо того чтобы начинать новую цепочку каждый раз, когда требуется градиент, и выполнять только один шаг выборки Гиббса, в PCD мы храним ряд цепочек (частиц фантазии), которые обновляются на k шагов Гиббса после каждого обновления веса. Это позволяет частицам более тщательно исследовать пространство.

2.3 TensorFlow

3. Организация работы с большими данными

Рассмотрим организацию работы с большими данными. Стоит отметить, данные отличаются от области к области. Например, работа с большими данными в области маркетинга и области медицины будет отличаться как по набору, так и применяемым подходам. Большие данные в промышленности – это еще одно отдельно направление. Всего несколько датчиков могут передавать абсолютно разную информацию об одном и том же технологическом процессе. Поэтому работа с такими данными будет осложнена на всех работы – от этапа сборки и хранения до проведения аналитики над этими данными.

3.1 Принципы Big Data

Кажется очевидным, что на протяжении большого количества времени поток такого числа источников данных требуется применения подхода Big Data, т.е. это серия подходов, инструментов и методов обработки структурированных и неструктурированных данных огромных объёмов и значительного многообразия для получения воспринимаемых человеком результатов, эффективных в условиях непрерывного прироста, распределения по многочисленным узлам вычислительной сети, сформировавшихся в конце 2000-х годов, альтернативных традиционным системам управления базами данных и решениям класса Business Intelligence. Важно, что под Big Data будем понимать не какой-то конкретный объём данных и даже не сами данные, а методы их обработки, которые позволяют распределённо обрабатывать информацию.

Исходя из определения **Big Data**, можно сформулировать основные принципы работы с такими данными:

- **масштабируемость**. Поскольку данных может быть сколь угодно много – любая система, которая подразумевает обработку больших данных, должна быть расширяемой. В 2 раза вырос объём данных – в 2 раза увеличили количество железа в кластере и всё продолжило работать.
- **Отказоустойчивость**. Принцип горизонтальной масштабируемости подразумевает, что машин в кластере может быть много. Например, Hadoop-кластер Yahoo имеет более 42000 машин. Это означает, что часть этих машин будет гарантированно выходить из строя. Методы работы с большими данными должны учитывать возможность таких сбоев и переживать их без каких-либо значимых последствий.
- **Локальность данных.** В больших распределённых системах данные распределены по большому количеству машин. Если данные физически находятся на одном сервере, а обрабатываются на другом – расходы на передачу данных могут превысить расходы на саму обработку. Поэтому одним из важнейших принципов проектирования BigData-решений является принцип локальности данных – по возможности обрабатываем данные на той же машине, на которой их храним.

3.2 Hive

Apache Hive - программное обеспечение, используемое хранилищами данных. Облегчает использование запросов и управление большими объемами данных, находящихся в распределенных хранилищах. Hive предоставляет механизм проектирования структур для этих данных и позволяет создавать запросы с использованием SQL -подобного языка, называемым HiveQL.

Фактически Hive превращает SQL-запросы в цепочки map-reduce задач. Движок включает в себя такие компоненты, как Parser(разбирает

входящие SQL-запрсоы), Optimimer(оптимизирует запрос для достижения большей эффективности), Planner (планирует задачи на выполнение) Executor(запускает задачи на фреймворке MapReduce. Для работы hive также необходимо хранилище метаданных. Дело в том что SQL предполагает работу с такими объектами как база данных, таблица, колонки, строчки, ячейки и тд. Поскольку сами данные, которые использует hive хранятся просто в виде файлов на HDFS - необходимо где-то хранить соответствие между объектами hive и реальными файлами.

В качестве metastorage используется обычная реляционная СУБД, такая как MySQL, PostgreSQL или Oracle. База данных представляет аналог базы данных в реляционных СУБД. База данных представляет собой пространство имён, содержащее таблицы.

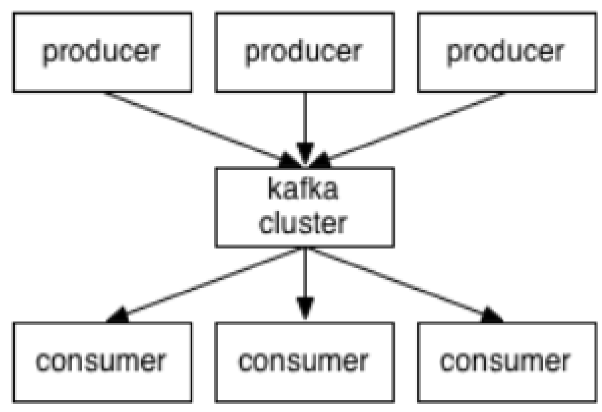
Таблица в hive представляет из себя аналог таблицы в классической реляционной БД. Основное отличие — что данные hive'овских таблиц хранятся прост в виде обычных файлов на hdfs. Это могут быть обычные текстовые csv-файлы, бинарные sequence-файлы, более сложные колоночные paruqet-файлы и другие форматы. Но в любом случае данные, над которыми настроена hive-таблица очень легко прочитать и не из hive.

3.3 Apache Kafka

Kafka был разработан в компании LinkedIn в 2011 году и с тех пор значительно усовершенствовался. Сегодня Kafka – это целая платформа, обеспечивающая избыточность, достаточную для хранения абсурдно огромных объемов данных. Здесь предоставляется шина сообщений с колоссальной пропускной способностью, на которой можно в реальном времени обрабатывать абсолютно все проходящие через нее данные.

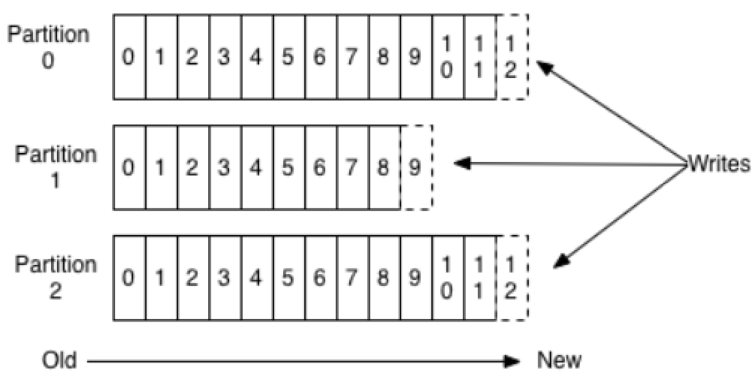
Приложения (генераторы) посылают сообщения (записи) на узел Kafka (брокер), и указанные сообщения обрабатываются другими приложениями,

так называемыми потребителями. Указанные сообщения сохраняются в теме, а потребители подписываются на тему для получения новых сообщений.



Консьюмер-продюсер

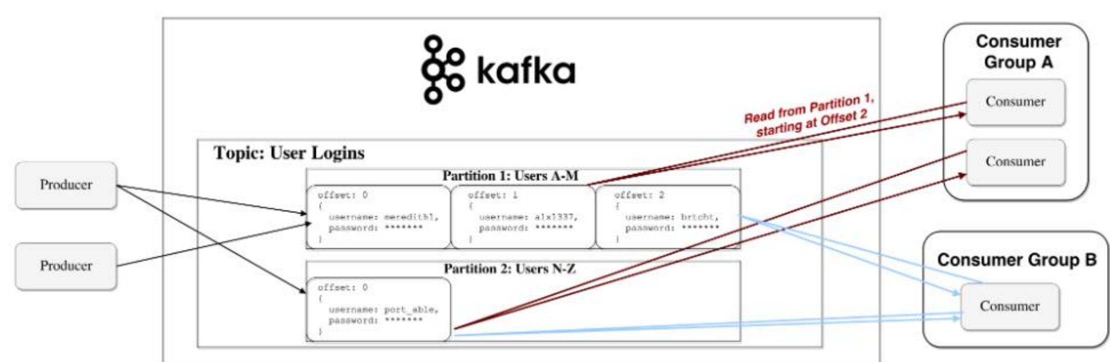
Темы могут разрастаться, поэтому крупные темы подразделяются на более мелкие секции для улучшения производительности и масштабируемости. Kafka гарантирует, что все сообщения в пределах секции будут упорядочены именно в той последовательности, в которой поступили. Конкретное сообщение можно найти по его смещению, которое можно считать обычным индексом в массиве, порядковым номером, который увеличивается на единицу для каждого нового сообщения в данной секции.



Журнал коммитов

В Kafka соблюдается принцип «тупой брокер – умный потребитель». Таким образом, Kafka не отслеживает, какие записи считываются потребителем и после этого удаляются, а просто хранит их в течение заданного периода времени (например, суток), либо до тех пор, пока не будет

достигнут некоторый порог. Потребители сами опрашивают Kafka, не появилось ли у него новых сообщений, и указывают, какие записи им нужно прочесть. Таким образом, они могут увеличивать или уменьшать смещение, переходя к нужной записи; при этом события могут переигрываться или повторно обрабатываться.



Организация тем и подписок в kafka

Zookeeper – это распределенное хранилище ключей и значений. Оно сильно оптимизировано для считывания, но записи в нем происходят медленнее. Чаще всего Zookeeper применяется для хранения метаданных и обработки механизмов кластеризации (пульс, распределенные операции обновления/конфигурации, т.д.).
Таким образом, клиенты этого сервиса (брокеры Kafka) могут на него подписываться – и будут получать информацию о любых изменениях, которые могут произойти. Именно так брокеры узнают, когда ведущий в секции меняется. Zookeeper исключительно отказоустойчив (как и должно быть), поскольку Kafka сильно от него зависит.

Он используется для хранения всевозможных метаданных, в частности:

- Смещение групп потребителей в рамках секции (хотя, современные клиенты хранят смещения в отдельной теме Kafka)
- ACL (списки контроля доступа)—используются для ограничения доступа /авторизации

- Квоты генераторов и потребителей — максимальные предельные количества сообщений в секунду
- Ведущие секций и уровень их работоспособности

3.4 Apache SPARK

Apache Spark - фреймворк с открытым исходным кодом для реализации распределённой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Apache Hadoop. В отличие от классического обработчика из ядра Apache Hadoop, реализующего двухуровневую концепцию Map Reduce с дисковым хранилищем, Spark использует специализированные примитивы для рекуррентной обработки в оперативной памяти, благодаря чему позволяет получать значительный выигрыш в скорости работы для некоторых классов задач, в частности, возможность многократного доступа к загруженным в память пользовательским данным делает библиотеку привлекательной для алгоритмов машинного обучения. Состоит из ядра и нескольких расширений, таких как Spark SQL (позволяет выполнять SQL-запросы над данными), Spark Streaming (надстройка для обработки потоковых данных), Spark MLlib (набор библиотек машинного обучения), GraphX (предназначено для распределённой обработки графов). Может работать как в среде кластера Apache Hadoop под управлением YARN, так и без компонентов ядра Apache Hadoop, поддерживает несколько распределённых систем хранения - HDFS, NoSQL –СУБД Cassandra, Amazon S3.

SparkSQL – это компонент Spark, поддерживающий запрашивание данных либо при помощи SQL, либо посредством Hive Query Language. Библиотека возникла как порт Apache Hive для работы поверх Spark (вместо MapReduce), а сейчас уже интегрирована со стеком Spark. Она не только обеспечивает поддержку различных источников данных, но и позволяет переплетать SQL-запросы с трансформациями кода; получается очень мощный инструмент.

Spark Streaming поддерживает обработку потоковых данных в реальном времени; такими данными могут быть файлы логов рабочего веб-сервера, информация из соцсетей, например, Twitter, а также различные очереди сообщений вроде Kafka. «Под капотом» Spark Streaming получает входные потоки данных и разбивает данные на пакеты. Далее они обрабатываются движком Spark, после чего генерируется конечный поток данных (также в пакетной форме).

MLlib - это библиотека для машинного обучения, предоставляющая различные алгоритмы, разработанные для горизонтального масштабирования на кластере в целях классификации, регрессии, кластеризации, совместной фильтрации и т.д. Некоторые из этих алгоритмов работают и с потоковыми данными — например, линейная регрессия с использованием обычного метода наименьших квадратов или кластеризация по методу k-средних (список вскоре расширится).

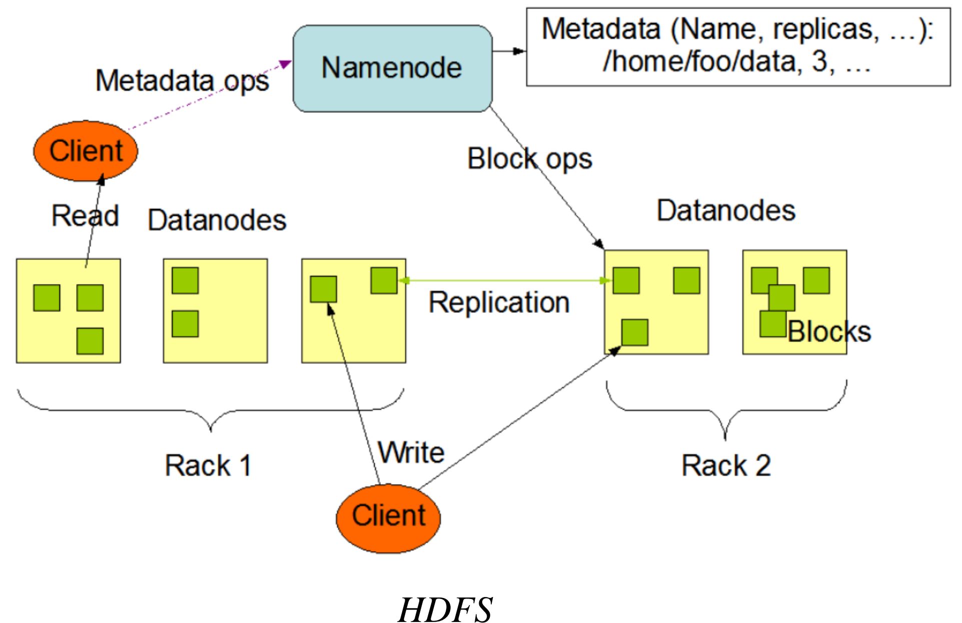
GraphX - это библиотека для манипуляций над графами и выполнения с ними параллельных операций. Библиотека предоставляет универсальный инструмент для ETL, исследовательского анализа и итерационных вычислений на основе графов.

3.5 Hadoop

Apache Hadoop - проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах из сотен и тысяч узлов. Проект состоит из четырёх модулей – Hadoop Common (связующее программное обеспечение — набор инфраструктурных программных библиотек и утилит, используемых для других модулей и родственных проектов), HDFS (распределённая файловая система), YARN (система для планирования заданий и управления кластером) и Hadoop MapReduce (платформа программирования и выполнения распределённых MapReduce-вычислений).

Распределенная файловая система, используемая в проекте Apache Hadoop предназначенная для хранения файлов больших размеров, поблочно распределённых между узлами вычислительного кластера. Все блоки в HDFS (кроме последнего блока файла) имеют одинаковый размер, и каждый блок может быть размещён на нескольких узлах, размер блока и коэффициент репликации (количество узлов, на которых должен быть размещён каждый блок) определяются в настройках на уровне файла. Благодаря репликации обеспечивается устойчивость распределённой системы к отказам отдельных узлов. Файлы в HDFS могут быть записаны лишь однажды (модификация не поддерживается), а запись в файл в одно время может вести только один процесс. Организация файлов в пространстве имён — традиционная иерархическая: есть корневой каталог, поддерживается вложение каталогов, в одном каталоге могут располагаться и файлы, и другие каталоги.

HDFS имеет архитектуру master/slave. Кластер HDFS состоит из одного NameNode, главного сервера, который управляет пространством имен файловой системы и регулирует доступ к файлам клиентами. Кроме того, существует несколько DataNodes, обычно по одному на узел в кластере, которые управляют хранилищем, прикрепленные к узлам, на которых они запускаются. HDFS предоставляет пространство имен файловой системы и позволяет сохранять пользовательские данные в файлах. Внутри файл разбивается на один или несколько блоков, и эти блоки хранятся в наборе DataNodes. NameNode выполняет операции с пространством имен файловой системы, такие как открытие, закрытие и переименование файлов и каталогов. Он также определяет отображение блоков в DataNodes. DataNodes отвечают за обслуживание запросов на чтение и запись от клиентов файловой системы. DataNodes также выполняют создание, удаление и репликацию блока по команде из NameNode.



HDFS

Одной из основных целей Hadoop изначально было обеспечение горизонтальной масштабируемости кластера посредством добавления недорогих узлов, без использования к мощных серверов и дорогих сетей хранения данных. Функционирующие кластеры размером в тысячи узлов подтверждают осуществимость и экономическую эффективность таких систем. Тем не менее, считается, что горизонтальная масштабируемость в Hadoop-системах ограничена, для Hadoop до версии 2.0 максимально возможно оценивалась в 4 тыс. узлов при использовании 10 MapReduce-заданий на узел. Во многом этому ограничению способствовала концентрация в модуле MapReduce функций по контролю за жизненным циклом заданий, считается, что с выносом её в модуль YARN в Hadoop 2.0 и децентрализацией — распределением части функций по мониторингу на узлы обработки — горизонтальная масштабируемость повысилась.

Ещё одним ограничением Hadoop-систем является размер оперативной памяти на узле имён, хранящем всё пространство имён кластера для распределения обработки, притом общее количество файлов, которое способен обрабатывать узел имён — 100 млн. Для преодоления этого ограничения ведутся работы по распределению узла имён, единого в текущей архитектуре на весь кластер, на несколько независимых узлов.

Hadoop MapReduce — программный каркас для программирования распределённых вычислений в рамках парадигмы MapReduce. Разработчику приложения для Hadoop MapReduce необходимо реализовать базовый обработчик, который на каждом вычислительном узле кластера обеспечит преобразование исходных пар «ключ — значение» в промежуточный набор пар «ключ — значение» (класс, реализующий интерфейс Mapper, назван по функции высшего порядка Map), и обработчик, сводящий промежуточный набор пар в окончательный, сокращённый набор (свёртку, класс, реализующий интерфейс Reducer). Каркас передаёт на вход свёртки отсортированные выводы от базовых обработчиков, сведение состоит из трёх фаз:

- shuffle (тасовка, выделение нужной секции вывода)
- sort (сортировка, группировка по ключам выводов от распределителей — досортировка, требующаяся в случае, когда разные атомарные обработчики возвращают наборы с одинаковыми ключами, при этом, правила сортировки на этой фазе могут быть заданы программно и использовать какие-либо особенности внутренней структуры ключей)
- reduce (свёртка списка) — получения результирующего набора. Для некоторых видов обработки свёртка не требуется, и каркас возвращает в этом случае набор отсортированных пар, полученных базовыми обработчиками.

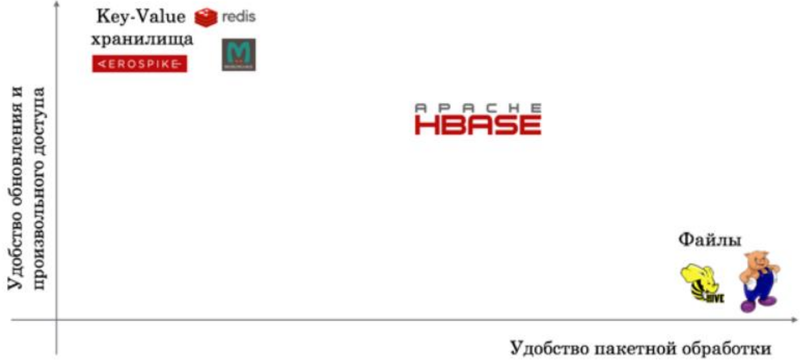
Hadoop MapReduce позволяет создавать задания как с базовыми обработчиками, так и со свёртками, написанными без использования Java: утилиты Hadoop streaming позволяют использовать в качестве базовых обработчиков и свёрток любой исполняемый файл, работающий со стандартным вводом-выводом операционной системы (например, утилиты командной оболочки UNIX), есть также SWIG-совместимый прикладной интерфейс программирования Hadoop pipes на C++. Также, в состав дистрибутивов Hadoop входят реализации различных конкретных базовых

обработчиков и свёрток, наиболее типично используемых в распределённой обработке.

3.6 HBase

HBase — СУБД класса NoSQL с открытым исходным кодом, проект экосистемы Hadoop. Написана на Java; относится к категории «семейство столбцов», многие технические решения переняты из Google BigTable. Работает поверх распределенной файловой системы HDFS и обеспечивает BigTable-подобные возможности для Hadoop, то есть обеспечивает отказоустойчивый способ хранения больших объёмов, разреженных данных.

Обычные файлы довольно неплохо подходят для пакетной обработки данных, с использованием парадигмы MapReduce. С другой стороны, информацию хранящуюся в файлах довольно неудобно обновлять; Файлы также лишены возможности произвольного доступа. Для быстрой и удобной работы с произвольным доступом есть класс nosql-систем типа key-value storage, таких как Aerospike, Redis, Couchbase, Memcached. Однако в обычно в этих системах очень неудобна пакетная обработка данных. Hbase представляет из себя попытку объединения удобства пакетной обработки и удобства обновления и произвольного доступа.



Сравнения подхода HBASE

Apache HBase обеспечивает случайный доступ в реальном времени к данным в Hadoop. Он был создан для размещения очень

больших таблиц, что делает его отличным выбором для хранения многостраничных или разреженных данных. Пользователи могут запрашивать HBase в определенный момент времени, делая запросы «воспоминания» возможными. Эти следующие характеристики делают HBase отличным выбором для хранения полуструктурированных данных, таких как данные журнала, а затем очень быстро предоставляют эти данные пользователям или приложениям, интегрированным с HBase.

Таблица 1. Характеристика HBase

Характеристика	Выгода
Отказоустойчивость	• репликация через центр обработки данных; • атомные и согласованные операции на уровне строк; • высокая доступность благодаря автоматическому отказоустойчивости; • автоматическая балансировка и балансировка таблиц.
Быстрота	• поиск в реальном времени; • кэширование в памяти через кеш-память блока и фильтры; • обработка на стороне сервера через фильтры и сопроцессоры.
Годность к потреблению	• модель данных поддерживает широкий спектр вариантов использования; • экспорт показателей через плагины File и Ganglia; • легкий API Java, а также API-интерфейс шлюза и REST-шлюза.

Предприятия используют хранилище с низкой задержкой Apache HBase для сценариев, которые требуют анализа в реальном времени и табличных данных для приложений пользователя. Apache HBase обеспечивает сверхбыстрый доступ к огромному быстро изменяющемуся хранилищу данных.

Приложения хранят данные в таблицах, состоящих из строк и столбцов. Для ячеек таблицы (пересечения строк и столбцов) действует контроль версии. По умолчанию в качестве версии используется временная метка, автоматически назначаемая HBase на момент вставки. Содержимое ячейки представляет собой неинтерпретируемый массив байтов.

Данные организованы в таблицы, проиндексированные первичным ключом, который в Hbase называется RowKey. Для каждого RowKey ключа может храниться неограниченны набор атрибутов (или колонок). Колонки организованны в группы колонок, называемые Column Family. Как правило в одну Column Family объединяют колонки, для которых одинаковы паттерн использования и хранения.

Для каждого атрибута может храниться несколько различных версий. Разные версии имеют разный timestamp. Записи физически хранятся в отсортированном по RowKey порядке. При этом данные соответствующие разным Column Family хранятся отдельно, что позволяет при необходимости читать данные только из нужного семейства колонок.

При удалении определённого атрибута физически он сразу не удаляется, а лишь маркируется специальным флажком tombstone. Физическое удаление данных произойдет позже, при выполнении операции Major Compaction. Атрибуты, принадлежащие одной группе колонок и соответствующие одному ключу физически хранятся как отсортированный список. Любой атрибут может отсутствовать или присутствовать для каждого ключа, при этом если атрибут отсутствует — это не вызывает накладных расходов на хранение пустых значений.

Список и названия групп колонок фиксирован и имеет четкую схему. На уровне группы колонок задаются такие параметры как time to live (TTL) и максимальное количество хранимых версий. Если разница между timestamp для определенно версии и текущим временем больше TTL — запись помечается к удалению. Если количество версий для определённого атрибута

превысило максимальное количество версий — запись также помечается к удалению.

4. Пример использование DataLake архитектуры для организации хранения технологических данных

Любое предприятие содержит большое количество разнообразного оборудования. Каждая отдельная единица оборудования состоит из большого количества зависимых электромеханических, логических устройств и датчиков. Каждая система или подсистема несет в себе информацию о ходе процесса, диагностическую информацию и возможно данные о будущих проблемах и поломках. Для оперативного анализа и выявление закономерностей между протекающими процессами внутри отдельного оборудования, цеха, предприятия нужно хранить всю информацию для последующего анализа, выполнения диагностических действия в реальном времени.

Каждое отдельное устройство внутри любого станка или оборудования, имеющая возможность отправки данных на веб-сервис должна логироваться с соответствующим уровнем. Таким образом мы можем получать как низкоуровневый лог, предоставляемый как витрина данных для рабочего отдельно этого оборудования (Оценка возможных будущих неисправностей оборудования, диагностика износа отдельных модулей и т.п.). Так и предоставление верхнеуровнего лога работы отдельных станков внутри цеха, предприятия. Выделение отдельных уровней отправки логов на веб-сервисы позволяет построить иерархическую структуру функционирования предприятия.

4.1 Архитектура решения на основе DATA LAKE

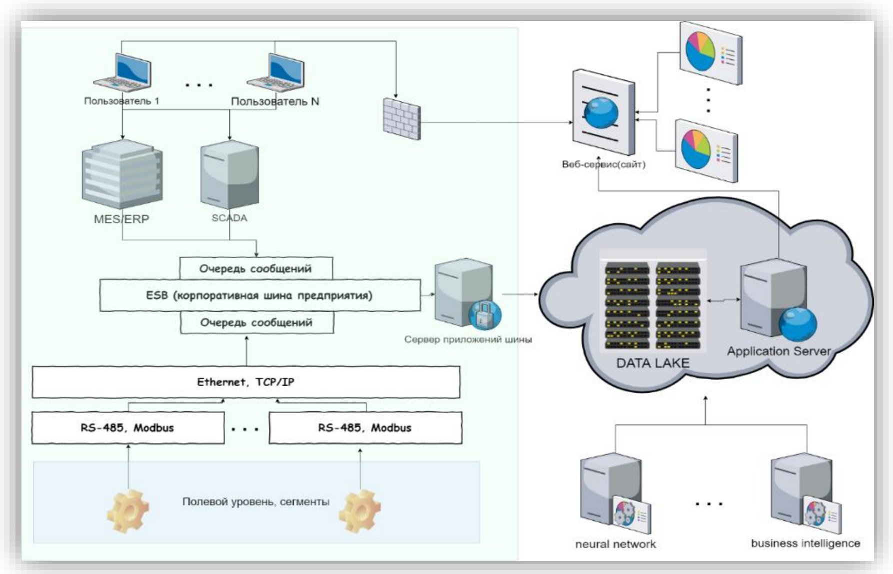
Иерархические организационные структуры в управлении предприятиям условно подразделяются: на производственные единицы, цеха, подразделения и т.п. В каждом организационной единицы имеется свой

набор оборудования, протоколы обмена информации. Информация может быть дискретной, может быть поточной, может со временем изменяться. Технология предусматривающая хранение разряженной информации, а также управление сложностью является – DATA LAKE.

В основе методологии построения архитектуры DATA LAKE является возможность хранить большие объемы данных доступных только на запись, организация очередей обработки, а также предоставления доступа к сохраненным данных на SQL-подобном языке.

Предлагаемая архитектура включает имеет иерархическое подразделение на полевом уровне, каждое устройство/система/датчик, подключенное в сеть должно иметь возможность передавать информацию в общую Ethernet-сеть. На уровне Ethernet формируется очередь сообщений перед отправкой в облако. В зависимости от масштаба предприятия или отправляемой информации может применяться «Корпоративная шина предприятия» (ESB), очереди сообщений (Kafka, для парадоксально больших объемов данных). В каждом из выбранных подходов имеется возможность синхронной (для важных данных в реальном времени), так и асинхронной отправки (для будущей аналитики). Данные с очереди отправляются в зависимости от приоритета в облако в котором развернут кластер Hadoop.

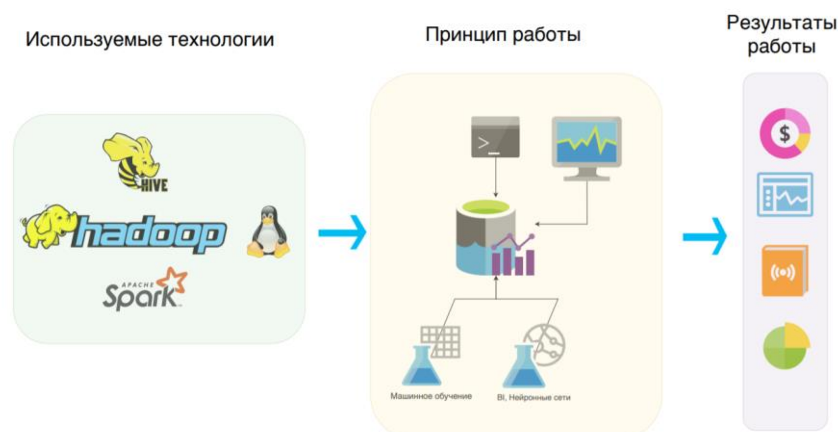
Общая схема взаимодействия:



Общая схема взаимодействия

Де-факто стандарт хранения и обработки информации в архитектурном паттерне DATA LAKE является кластер Hadoop (распределенное, отказоустойчивое хранилище), предоставляющие возможности обработки информации с применением map-reduce, spark (рекуррентный map-reduce), SQL-подобный синтаксис извлечения данных и т.п.

Фактически Hadoop - инфраструктура, построенная вокруг необходимости хранения больших данных, так и плохо индексируемых небольших файлов. Одной из основных проблем реляционных баз данных, является высокая вероятность отказа единой точки входа при достижении определенных пределов хранения (в зависимости от базы данных), отсутствие индексируемости BLOB, CLOB и т.п. файлов



Рабочий процесс

Открываются возможности применения технологий машинного обучения, нейронных сетей, построение аналитики на BI-средствах. Отправной точкой, которая является при анализе данных предприятия является регрессионный анализ зависимости параметров, позволяющих перебор различных параметров выявить закономерности между протекающими процессами.

4.2 Реализация сбора информации на базе IOT Azure

Microsoft Azure - название облачной платформы Microsoft. Предоставляет возможность разработки и выполнения приложений и хранения данных на серверах, расположенных в распределённых дата-центрах. Azure реализует две облачные модели — платформы как сервиса (PaaS) и инфраструктуры как сервиса (IaaS). Работоспособность платформы Microsoft Azure обеспечивает сеть глобальных дата-центров Microsoft.

Основные особенности данной модели:

- оплата только потреблённых ресурсов;
- общая, многопоточная структура вычислений;
- абстракция от инфраструктуры.

В основе работы Microsoft Azure лежит запуск виртуальной машины для каждого экземпляра приложения. Разработчик определяет необходимый объём для хранения данных и требуемые вычислительные мощности (количество виртуальных машин), после чего платформа предоставляет соответствующие ресурсы. Когда первоначальные потребности в ресурсах изменяются, в соответствии с новым запросом заказчика платформа выделяет под приложение дополнительные или сокращает неиспользуемые ресурсы дата-центра.

Microsoft Azure состоит из:

- *Compute* — компонент, реализующий вычисления на платформе Windows Azure.
- *Storage* — компонент хранилища предоставляет масштабируемое хранилище. Хранилище не имеет возможности использовать реляционную модель и является альтернативной, «облачной» версией SQL Server.
- *Fabric* — Microsoft Azure Fabric по своему назначению является «контролёром» и ядром платформы, выполняя функции мониторинга в

реальном времени, обеспечения отказоустойчивости, выделении мощностей, развертывания серверов, виртуальных машин и приложений, балансировки нагрузки и управления оборудованием.

Практически все сервисы Microsoft Azure имеют интерфейс взаимодействия API, построенный на основе ограничений для распределённых гипер-систем REST, что позволяет разработчикам использовать «облачные» сервисы с любой операционной системы, устройства и платформы.

4.3 Архитектура решения

Архитектура облака Azure предоставляет стандартные средства для работы, конфигурирования и обеспечения безопасности работы с неограниченным количеством подключаемых устройств, имеющих доступ в интернет. В эту категорию относятся отдельные датчики, одноплатные компьютеры, системы ЧПУ, SCADA-системы и т.п.

Выбранное решение включает основывается на таких технологиях AZURE как: «Центр интернет вещей»(IotHubCenter), «Служба аналитики в реальном времени по требованию» (Stream analytics data), кластер базы данных MSSQL.

Центр Интернета вещей - размещенная в облаке управляемая служба, которая действует в качестве центра сообщений для двусторонней связи между приложением Интернета вещей и устройствами, которыми оно управляет. Центр Интернета вещей Azure можно использовать для создания решения Интернета вещей с надежными и безопасными связями между миллионами устройств Интернета вещей и серверной частью решения, размещенного в облаке. Подключаться к Центру Интернета вещей можно практически со всех устройств.

Центр Интернета вещей поддерживает обмен данными с устройства в облако и из облака на устройство. Центр Интернета вещей также поддерживает несколько шаблонов обмена сообщениями, таких как передача

телеметрии с устройства в облако, передача файлов с устройств и метод "запрос — ответ" для управления устройствами из облака. Мониторинг Центра Интернета вещей позволяет поддерживать работоспособность решения путем отслеживания событий, таких как создание устройства, сбои в работе устройств и подключение устройств.

Azure Stream Analytics — это модуль обработки событий, который позволяет проверять большие потоки данных из устройств. Данные могут поступать из устройств, датчиков, веб-сайтов, каналов социальных сетей, приложений и других источников. Эта служба также поддерживает извлечение информации из потоков данных, определение шаблонов и связей. Эти шаблоны можно использовать для активации других подчиненных действий, таких как создание оповещений, отправка информации в средства создания отчетов или хранение данных для последующего использования.

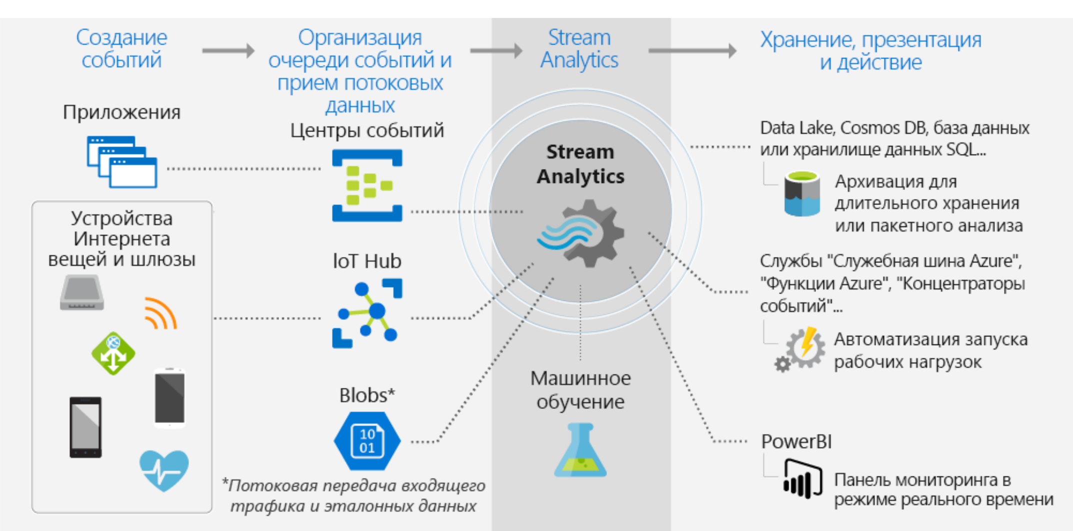
Azure Stream Analytics запускается с источником данных потоковой передачи, которые принимает концентратор событий Azure или Центр Интернета вещей и которые могут передаваться из хранилища данных, например хранилища BLOB-объектов Azure. Чтобы изучить потоки, создайте задание Stream Analytics, указывающее источник входных данных, выполняющий потоковую передачу. Задание также указывает запрос преобразования, который определяет, как выполнять поиск данных, шаблонов или связей. Запрос преобразования использует язык SQL-запросов для простого выполнения фильтрации, сортировки, вычисления и объединения потоковых данных за определенный период времени. Для задания можно настроить параметры упорядочения событий и продолжительность временных окон при выполнении операций вычисления.

После завершения анализа входящих данных укажите место вывода для преобразованных данных. Вы можете контролировать, что делать в ответ на

информацию, которую вы проанализировали. Например, можно выполнять такие действия:

- отправлять данные в контролируемую очередь для запуска оповещений или подчиненных пользовательских рабочих потоков;
- отправлять данные на панель мониторинга Power BI для визуализации в режиме реального времени;
- хранить данные в других службах хранилища Azure, чтобы обучать модель машинного обучения на основе данных журнала или выполнять пакетную аналитику.

На следующем рисунке показан конвейер Stream Analytics. Задание Stream Analytics может использовать все входные и выходные данные или выбранный набор. На рисунке показано, как данные отправляются в Stream Analytics, анализируются и направляются для выполнения других действий, например хранения или отображения:



Структура Stream analytics

Предлагаемое решение на текущем этапе имеет возможность подключения различных датчиков, данные с которых считываются через raspberry pi, omega, системы ЧПУ axiOMA и/или любой другой системы

ЧПУ, данные отправляются на IotHubCenter. На уровне IotHubCenter выдаются политики безопасности и регистрирует отдельные устройства.

Датчики подключенные к raspberry pi имеют одну политику безопасности в IotHubCenter, это сделано для исходя из выбора реляционной базы данных и необходимости явного преобразования json формата в реляционный вид. Преобразование json в реляционный вид происходит на уровне stream analytics data на SQL-подобном языке.

Входные данные (3)
- inputOmega
- inputRPI
- inputRPI1

Выходные данные (3)
- outputDeviceValue
- outputDeviceValueRPI1
- outputOmega

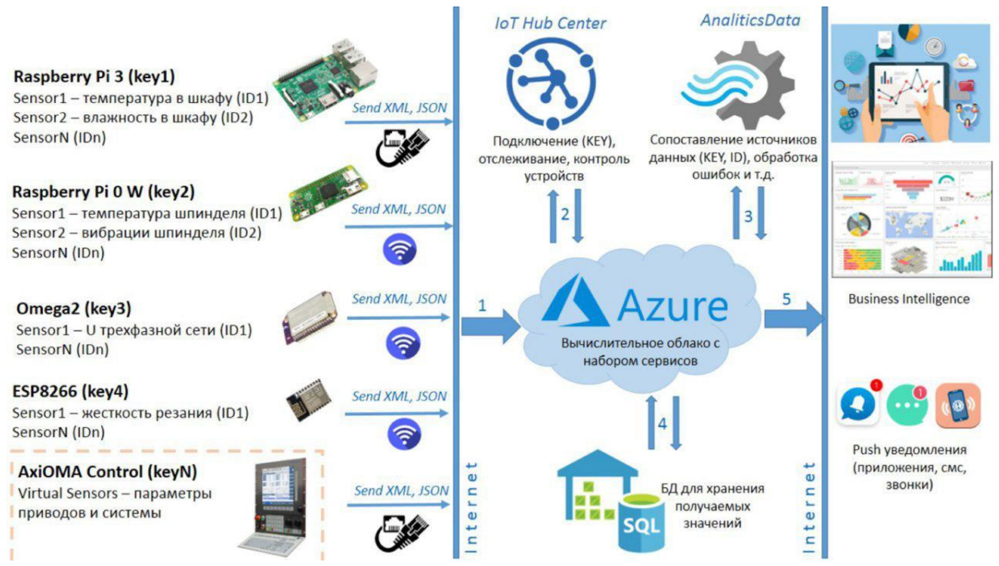
Нужна помощь с запросом? Некоторые распространенные шаблоны запросов Stream Analytics можно посмотреть з

```
1  SELECT
2      attribute2, device_id, attribute1, time, 1 as context_id
3  INTO
4      outputDeviceValue
5  FROM
6      inputRPI
7
8  SELECT
9      attribute2, device_id, attribute1, time, 2 as context_id
10 INTO
11     outputDeviceValueRPI1
12 FROM inputRPI1
13
14 SELECT
15     attribute1, device_id, time, 3 as context_id
16 INTO
```

Сопоставление поток данных

Политики безопасности явно задают какое устройство может приниматься в созданном потоке «Входные данные», такой подход позволяет управлять сложностью при разнообразии датчиков и устройств. Очевидно деление входных потоков на тип устройств и запись «как есть» в реляционный схему данных говорит о необходимости в отчетности или приложении хранить описание и сопоставление параметров отдельного в БД (отражено на схеме базы данных) или приложения.

Общая схема работы с технологиями AZURE:



Общая схема работы с технологиями AZURE

4.4. Схема базы данных

Структура база данных отражает ожидаемые потоки от различных устройств: raspberry_devices, machine_devices. Атрибуты ожидаемые в этих таблицах имеют связи через contex_id с таблицей description_flex, в которое хранится точное определение принимаемого параметра для конкретного устройства с определенным контекстом. Таблица devices хранит описание параметров устройств.

Организация хранения данных в такой структуре позволяет создавать новые таблицы для заранее неизвестных источников данных.

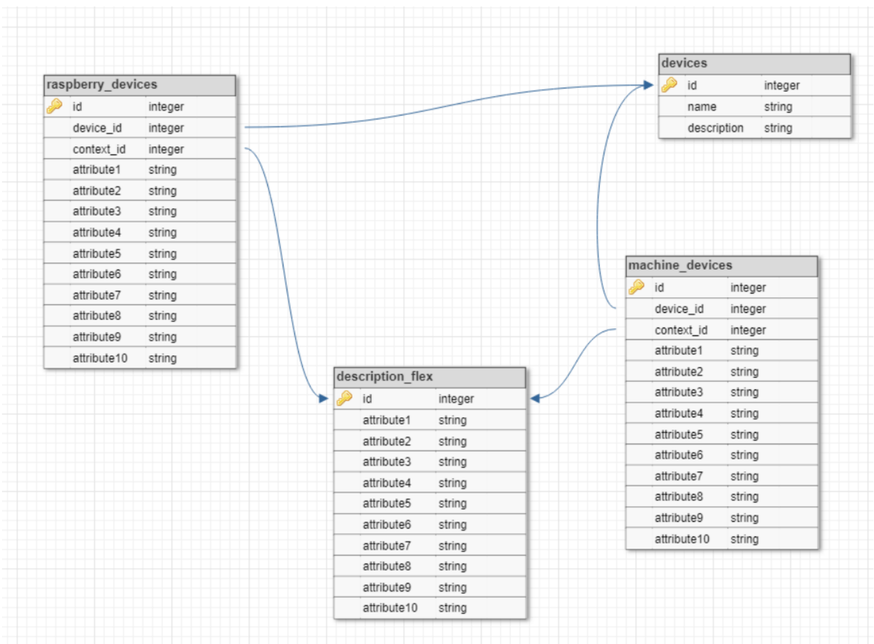


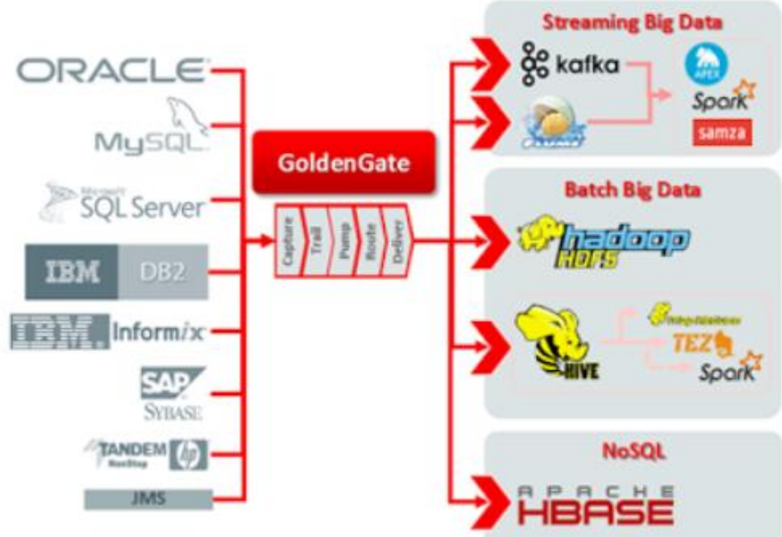
Схема базы данных

4.5 Итоги

Реализация хранения информации различных датчиков на основе облачных технологий AZURE можно рассматривать как один из потоков корпоративной/производственной DATA LAKE. Azure stream analytics позволяет передавать данные в реальном времени (с учетом сетевых взаимодействий). Архитектура выделение ресурсов по требованию позволяет снизить стоимостные затраты на информационные мощности на первоначальном этапе, а также позволяет масштабировать решение горизонтально без остановки получаемых данных.

Архитектура DATA LAKE подразумевает хранение на чтение любых данных от заранее неопределенных источников, но не изменение и удаление данных. MSSQL используемое для реализации решения на базе интернета вещей является одним из стандартных источников для любых из выбираемых технологий построения DATA LAKE.

Например, одна из рекомендуемых архитектур при построении DATA LAKE от ORACLE имеет вид:



Коннектов Golden Gate

Фактически они предоставляет коннектов с названием Golden Gate по преобразованию данных из наиболее известных баз данных, а так же плоских файлов в стандартный для архитектуры DATA LAKE последовательность обработки Kafka/шина предприятия -> SPARK -> Hadoop.

5. Практическая часть использования и аналитики данных

Jupyter-ноутбук — это среда разработки, где сразу можно видеть результат выполнения кода и его отдельных фрагментов. Отличие от традиционной среды разработки в том, что код можно разбить на куски и выполнять их в произвольном порядке. Представьте, что вы можете написать кусочек кода на салфетке и сказать салфетке: «Выполнись».

В такой среде разработки можно, например, написать функцию и сразу проверить её работу, без запуска программы целиком. А ещё можно поменять порядок выполнения кода. Можно отдельно загрузить файл в память, отдельно проверить его содержимое, отдельно обработать содержимое.

А ещё в jupyter-ноутбуках есть вывод результата сразу после фрагмента кода. Например, можно прямо в середине кода увидеть построенный график, получить предварительные цифры или любую другую визуализацию.

Чтобы протестировать фрагменты кода, вы можете запустить Python из командной оболочки вашей операционной системы в интерактивном режиме. (т.е. в режиме выполнения кода без создания файла с этим кодом).

```
user1@user1-VirtualBox: ~
Файл Правка Вид Поиск Терминал Справка
user1@user1-VirtualBox:~$ python3
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information
.
>>> print("This is my beginning of machine learning!")
This is my beginning of machine learning!
>>> 2+2
4
>>> print("Результат 2+2: ", 2+2)
Результат 2+2:  4
>>> quit()
```

Пример вывода результата программы Python в консоли Ubuntu

```
Администратор: C:\Windows\system32\cmd.exe - python

C:\Users\drumm>python
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> print("Результат 2+3: ", 2+3)
Результат 2+3:  5
>>>
```

Пример вывода результата программы Python в консоли Windows

Но это неудобно, когда вам нужно постоянно использовать один и тот же код. При этом, просто создавать большую программу для аналитики данных с расширением *.py* вам тоже будет не удобно, потому что вы в аналитике данных вам нужно оставлять много текстового описания к вашему кода, потому что суть вашей работы не в программировании, а в аналитике при помощи программирования.

Поэтому Jupyter блокноты являются наиболее полезным инструментом для аналитики данных. Весь код из ноутбуков можно экспортировать через меню File, если вы хотите использовать его как готовую программу. А если вам необходима отчетность, то вы можете распечатать ваш ноутбук через это же меню, потому что ноутбук и должен представлять собой готовый отчет по аналитике.

Вы можете вводить команды в поля и выполнять их.

5.1 Взаимодействие с Python в jupyter-ноутбуках

Jupyter-ноутбук — это среда разработки, где сразу можно видеть результат выполнения кода и его отдельных фрагментов. Отличие от традиционной среды разработки в том, что код можно разбить на куски и выполнять их в произвольном порядке. Представьте, что вы можете написать кусочек кода на салфетке и сказать салфетке: «Выполнись».

В такой среде разработки можно, например, написать функцию и сразу проверить её работу, без запуска программы целиком. А ещё можно поменять порядок выполнения кода. Можно отдельно загрузить файл в память, отдельно проверить его содержимое, отдельно обработать содержимое.

А ещё в jupyter-ноутбуках есть вывод результата сразу после фрагмента кода. Например, можно прямо в середине кода увидеть построенный график, получить предварительные цифры или любую другую визуализацию.

5.1.1 Запуск Python из командной строки

Чтобы протестировать фрагменты кода, вы можете запустить Python из командной оболочки вашей операционной системы в интерактивном режиме. (т.е. в режиме выполнения кода без создания файла с этим кодом)

```
user1@user1-VirtualBox: ~
Файл Правка Вид Поиск Терминал Справка
user1@user1-VirtualBox:~$ python3
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information
.
>>> print("This is my beginning of machine learning!")
This is my beginning of machine learning!
>>> 2+2
4
>>> print("Результат 2+2: ", 2+2)
Результат 2+2:  4
>>> quit()
```

Но это неудобно, когда вам нужно постоянно использовать один и тот же код. При этом, просто создавать большую программу для аналитики данных с расширением .py вам тоже будет не удобно, потому что вы в аналитике данных вам нужно оставлять много текстового описания к вашему кода, потому что суть вашей работы не в программировании, а в аналитике при помощи программирования.

Поэтому Jupyter блокноты являются наиболее полезным инструментом для аналитики данных. Весь код из ноутбуков можно экспортировать через меню File, если вы хотите использовать его как готовую программу. А если вам необходима отчетность, то вы можете распечатать ваш ноутбук через это же меню, потому что ноутбук и должен представлять собой готовый отчет по аналитике.

Вы можете вводить команды в поля и выполнять их.

В поле введите:

```
print('Hello, World')
```

Затем нажмите shift + return или кнопку Run в верхнем меню для выполнения команды.

```
print('Hello, World')
```

Hello, World

Что произошло?

Вы только что создали программу, которая печатает слова «Hello, World». Среда Python, в которой вы находитесь, немедленно компилирует все, что вы набрали. Это полезно для тестирования, например, когда проверяете будет ли работать определенная строка.

Функция print() в Python выводит заданные объекты на стандартное устройство вывода (экран) или отправляет их текстовым потоком в файл.

Полный синтаксис функции print():

- print(*objects, sep=' ', end='n', file=sys.stdout, flush=False)
- objects – объект, который нужно вывести * обозначает, что объектов может быть несколько;
- sep – разделяет объекты. Значение по умолчанию: ‘ ‘;
- end – ставится после всех объектов;
- file – ожидается объект с методом write (string). Если значение не задано, для вывода объектов используется файл sys.stdout;
- flush – если задано значение True, поток принудительно сбрасывается в файл. Значение по умолчанию: False. Примечание: sep, end, file и flush — это аргументы-ключевые слова. Если хотите воспользоваться аргументом sep, используйте: print(*objects, sep = 'separator'), а не print(*objects, 'separator').

Можно печатать несколько значений сразу, если передать их функции print() с разделением через запятые, например, выполните:

```
print(1,2,3)
```

1 2 3

5.1.2. Математика в Python

Напечатайте

```
1 + 1
```

и выполните код.

```
print (20+30)
```

50

Теперь напечатайте

```
20 + 80
```

и выполните код.

```
20+80
```

Это была арифметическая операция "сложение". В Python есть представленные ниже математические операции. Так же при помощи использования функций из большого числа библиотек, можно пользоваться и другими, обычно более сложными операциями - такими, как перемножение матриц и др.

Оператор/ функция	Описание
X + Y	Сложение
X - Y	Вычитание
X * Y	Умножение
X / Y	Деление
X // Y	Деление с округлением вниз (до ближайшего меньшего)
X % Y	Остаток от деления
divmod(X,Y)	Пара (X // Y, X % Y)
- X	Смена знака числа
abs(X)	Модуль числа
X ** Y	Возведение в степень
pow(X, Y[, Z])	X в степени Y, [X в степени Y по модулю Z: вычисление остатка от деления числа X в степени Y на число Z (модуль)]
X.as_integer_ratio()	Возвращает пару целых чисел, чье отношение равно вещественному X
X.is_integer()	Проверка - является ли значение вещественного X целым числом
int([строка-число], [основание системы счисления])	Преобразование к целому числу в десятичной системе счисления числа в системе от 2 до 36 (по умолчанию в десятичной)
bin(X)	Преобразование целого числа в двоичную строку
hex(X)	Преобразование целого числа в шестнадцатеричную строку
oct(X)	Преобразование целого числа в восьмеричную строку

Попробуйте выполнить некоторые операции из представленных.

Например, нахождение остатка от деления при помощи оператора %:

```
23 % 3
```

5.1.3 Порядок операций

Порядок операций в Python можно задавать при помощи круглых скобок ().

Вот несколько примеров, которые вы, возможно, захотите попробовать

```
1 + 2 * 3
(1 + 2) * 3
```

```
print(df)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-2a851eda2e88> in <module>
----> 1 print(df)

NameError: name 'df' is not defined
```

5.1.4 Комментарии

Коммента́рии — пояснения к исходному тексту программы, находящиеся непосредственно внутри комментируемого кода. Синтаксис комментариев определяется языком программирования. С точки зрения компилятора или интерпретатора, комментарии — часть текста программы, не влияющая на её семантику. Комментарии не оказывают никакого влияния на результат компиляции программы или её интерпретацию.

Выполните следующие три строчки:

```
# Строчка ниже закомментирована и не выполнится
# print("Привет")
print("Мир")
```

и вы увидите, что выполнится только print("Мир"), а print("Привет") не выполнится.

Type Markdown and LaTeX: $\alpha 2$

5.2 Программы в файле, переменные и строки

5.2.1 Введение

Что ж, мы можем делать однострочные программы. Ну и что? Вы хотите отправлять программы другим людям, чтобы они могли их использовать, не зная, как их писать.

5.2.2 Написание программ

Программы Python - это текстовые документы - вы можете открыть их в блокноте и посмотреть на них. Если вы разрабатываете сложные программы, то лучше всего использовать специальные среды, или, так называемые - Integrated Development Environment (IDE)

(https://en.wikipedia.org/wiki/Integrated_development_environment) для разработки своих программ. Например, одна из популярных IDE для программирования на языке Python - это PyCharm. В среде IDE всегда есть экран, на котором вы можете ввести код с подсветкой синтаксиса (как здесь), окно, в котором вы можете увидеть результат, и кнопку для запуска скрипта.

Посмотрим на программу (Есенин.ру), исходный код которой представлен ниже:

```
#Простая программа
print("Пой же, пой. На проклятой гитаре")
print("Пальцы пляшут твои вполукруг.")
print("Захлебнуться бы в этом угаре,", end = "\n")
print("Мой последний, единственный друг.")
```

Пой же, пой. На проклятой гитаре

Пальцы пляшут твои вполукруг.

Захлебнуться бы в этом угаре,

Мой последний, единственный друг.

Если вы запустите программу (<shift> + <Return>), интерпретатор выполнит строки с 1 по 5 друг за другом.

В строке 4 параметр end ="\n", передающийся фукнции print, настраивает её таким образом, чтобы она вставляла спец символ перевода строки \n, который отвечает за действие в конце выполнения этой функции. Хотя она и так по умолчанию переводит строку на новую, но для наглядности это сообщено вам. Вы можете, например, вставить пробел вместо перевода строки.

```
print("Захлебнуться бы в этом угаре,", end = " ")
print("Мой последний, единственный друг.")
```

Захлебнуться бы в этом угаре, Мой последний, единственный друг.

5.2.3 Переменные

Теперь давайте начнем вводить переменные. Переменные хранят значение, которое можно просмотреть или изменить позже. Создадим программу, использующую переменные:

```
#демонстрация переменных
print("Эта программа представляет собой демонстрацию переменных")
v = 1
print("Сейчас значение v: ", v)
v = v + 1
print("v теперь равно самому себе плюс один, что делает его равным ", v)
v = 51
print("v может хранить любое числовое значение для использования в другом месте.
print("например, в предложении. v теперь равно ", v)
print("v умножить на 5 равно", v * 5)
print("но само v все еще остается равно", v)
print("чтобы увеличить v в пять раз, нужно набрать v = v * 5")
v = v * 5
print("вот так, теперь v равно", v, "а не ", v / 5)
```

Эта программа представляет собой демонстрацию переменных

Сейчас значение v: 1

v теперь равно самому себе плюс один, что делает его равным 2

v может хранить любое числовое значение для использования в другом месте.

например, в предложении. v теперь равно 51

v умножить на 5 равно 255

но само v все еще остается равно 51

чтобы увеличить v в пять раз, нужно набрать v = v * 5

вот так, теперь v равно 255 а не 51.0

Запустите программу и попробуйте разобраться в результатах.

Обратите внимание, что мы также можем записать v = v + 1 какv + = 1. Это можно использовать для всех операторов (например, - =, * =, / =). Попробуйте это в приведенном выше коде.

Для переменных рекомендуется использовать строчные буквы или "верблюжий регистр". Не используйте специальные символы и не начинайте с цифры!

CamelCase (с англ. — «ВерблюжийРегистр», также «ГорбатыйРегистр», «СтильВерблюда») — стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы пишется с прописной буквы. Стиль получил название CamelCase, поскольку прописные буквы внутри слова напоминают горбы верблюда (англ. Camel).

5.2.5 Строки

Как видите, переменные хранят значения для использования в более позднее время. Вы можете изменить их в любой момент. Однако вы можете ввести больше, чем просто цифры. Переменные могут содержать, например текст. Переменная, содержащая текст, называется строкой. Попробуйте эту программу:

```
#обьявление и определение строковых переменных, сложение строк
word1 = "Добрый"
word2 = "вечер"
word3 = "вам!"
print(word1, word2)
sentence = word1 + " " + word2 + " " + word3
print(sentence)
```

Добрый вечер

Добрый вечер вам!

Как видите, указанные выше переменные содержали текст. Имена переменных также могут быть длиннее одной буквы - здесь у нас были слова

word1, word2 и word3. Как видите, строки можно складывать вместе, чтобы получились более длинные слова или предложения. Тем не менее, он не добавляет пробелов между словами - поэтому я вставляю вещи " " (между ними есть один пробел).

Часто нам нужно манипулировать строками. Например, если мы хотим редактировать имена файлов или делать выборки из текста. Строки похожи на списки, которые вы узнаете позже. Таким образом, аналогичные операции (называемые * разрезанием списка *) применяются к строкам.

Попробуйте использовать следующий код и объясните, что он делает:

```
text = "abcdefghij"
len(text)
```

10

Да, он показывает нам количество символов в строке.

А теперь попробуйте это:

```
print(text[4])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-f5ab9aa42160> in <module>
----> 1 print(text[4])

NameError: name 'text' is not defined
```

Здесь мы хотим напечатать символ в позиции 4. Обратите внимание, что первый символ «а» находится в позиции 0! Итак, позиция 4 возвращает нам «е».

А теперь попробуйте:

```
print(text[:4])
print(text[4:])
```

abcd

efghij

Здесь вы видите, что [: 4] выбирает символы с индексами 0,1,2,3, то есть «abcd». С [4:] мы начинаем с позиции 4 (считая от 0!) До конца строки, в результате получается «efghij».

Мы также можем указать диапазон. Попробуйте это:

```
print(text[4:8])
```

efgh

Для диапазонов максимальное значение не включается в выборку.

У нас все еще есть переменная «предложение» (sentence). Напишите код ниже, чтобы выбрать второе слово предложения, используя правильный диапазон.

Введите код

Сперва попробуем вывести второй символ справа. А затем попробуем выбрать четвертый символ и правее. Обратите внимание, что здесь мы начинаем отсчет с -1 для первого символа справа. На человеческом языке: начните с 4-го символа справа и дайте мне все символы от этой позиции до конца строки.

```
#text = "abcdefghij"
print(text[-2])
print(text[-4:])
```

i

ghij

Попробуйте и узнайте что делает этот код:

```
print(text[::2])
```

acegi

Мы также можем добавить результаты нарезки списка к переменным и вставить их в строки, используя операцию форматирования:

```
nrOfCharacters = len(sentence)
lastWord = sentence[-4:-1]
print("В предложении есть {} символов, а последнее слово: {}".format(nrOfCharact
```

В предложении есть 17 символов, а последнее слово: вам

Таким образом, вы можете легко вставлять переменные в места, отмеченные оператором ассоциации ({}). Они будут подставлены в том же порядке, что и переменные в .format () Это также можно записать как:

```
print(f"В предложении {nrOfCharacters} букв и последнее слово: {lastWord}")
```

В предложении 17 букв и последнее слово: вам

Помимо нарезки списка есть также другие операции, которые мы можем применить к строкам. Мы можем подсчитать количество вхождений определенного символа в строку:

```
print(sentence.count('o'))
```

0

Мы также можем найти положение символа:

```
print(text.find('e'))
```

4

```
sometext = "Привет, как дела как?"
print(sometext.rfind("как"))
```

17

Метод rfind возвращает последнее появление строки. Итак, в sometext мы дважды употребляем слово «как». rfind возвращает17, что означает, что последний раз, когда он нашел «как», он начинался с позиции 17 (считая с 0).

Есть еще несколько полезных строковых операций. Запустите приведенный ниже код и будет очевидно, что он делает:

```
# Переводим строку в верхний регистр
print(sometext.upper())

# Разбиваем строку на символ и возвращаем ее как элементы списка. Вы узнаете о с
print(sometext.split(","))

# Заменяем строки
print(sometext.replace("?","!"))
```

ПРИВЕТ, КАК ДЕЛА КАК?

['Привет', ' как дела как?']

Привет, как дела как!

Также есть некоторые специальные символы:

\n переходит на новую строку

\ - escape-символ. Вы можете поместить его перед другим символом, который имеет значение в коде и не считается строкой. Это часто используется, чтобы иметь строки с обратной косой чертой для имен файлов в Windows (например, " C: \\ folder \\ filename.txt "). Поскольку \ уже является escape-символом, нам нужно использовать его дважды, чтобы избежать escape-символа!

Примеры:

```
print("Это очень длинное предложение, и я хочу разбить его на две строки.")
print("Это очень длинное предложение \n, и я хочу разбить его на две строки.")
print("Это предложение содержит цитату, и я не хочу, чтобы строка заканчивалась
```

Это очень длинное предложение, и я хочу разбить его на две строки.

Это очень длинное предложение

, и я хочу разбить его на две строки.

Это предложение содержит цитату, и я не хочу, чтобы строка заканчивалась (пока) "

5.3 Циклы

5.3.1 Введение

Представьте, что вам нужна программа, которая сделает что-то 20 раз.Вы можете скопировать и вставить код 20 раз и получить практически нечитаемую программу, не говоря уже о медленной и бессмысленной. Или вы можете приказать компьютеру повторять фрагмент кода между точками A и B, пока не придет время, когда вам нужно будет остановиться. Это называется циклом.

5.3.2 Цикл "While" ("пока")

Ниже приведены примеры цикла, называемого циклом while:

```
a = 0
while a < 10:
    a = a + 1
    print(a)
```

1

2

3

4

5

6

7

8

9

10

Как работает эта программа? Давайте опишем простым языком:

'а' теперь равно 0

Пока «а» меньше 10, сделайте следующее:

Сделайте единицу больше, чем она есть.

Напечатайте на экране, сколько сейчас стоит «а».

Что это значит? Давайте рассмотрим, какую последовательность действий компьютер будет выполнять, при выполнении цикла «while»:

ИТЕРАЦИЯ 0

«А» меньше 10? ДА (его 0)

Сделать 'а' на один больше (теперь 1)

напечатать на экране значение 'а' (1)

ИТЕРАЦИЯ 1

«А» меньше 10? ДА (его 1)

Сделайте "а" на один больше (теперь 2)

напечатать на экране, что такое 'а' (2)

ИТЕРАЦИЯ 9

«А» меньше 10? НЕТ (значение «А» = 10, следовательно, не оно меньше 10)

Не делайттть цикл

Кода большене осталось, поэтому программа завершается

while имеет следующий синтаксис:

while {условие продолжения цикла}:

    {код тела цикла (что делать)}

    {тело всегда отделяется tabом или 4мя пробелами}

код здесь не зациклен

потому что он без отступа

Now try to understand this example and run to see if it is what you expected.

Теперь попытайтесь понять, предсказать и сравнить с фактическим результат выполнения:

```
x = 10
while x != 0:
    print(x)
    x = x - 1
    print("вау, мы отсчитали x, и теперь он равен", x)
print("И вот цикл закончен.")
```

10

вау, мы отсчитали x, и теперь он равен 9

9

вау, мы отсчитали x, и теперь он равен 8

8

вау, мы отсчитали x, и теперь он равен 7

7

вау, мы отсчитали x, и теперь он равен 6

6

вау, мы отсчитали x, и теперь он равен 5

5

вау, мы отсчитали x, и теперь он равен 4

4

вау, мы отсчитали x, и теперь он равен 3

3

вау, мы отсчитали x, и теперь он равен 2

2

вау, мы отсчитали x, и теперь он равен 1

1

вау, мы отсчитали x, и теперь он равен 0

И вот цикл закончен.

5.3.3 Логические, или булевые, выражения

Что вы набираете в области, отмеченной {условия продолжения цикла}? Ответ - логическое выражение.

Логическое выражение просто означает вопрос, на который можно ответить ИСТИНА или ЛОЖЬ. Например, если вы хотите сказать, что ваш возраст такой же, как и у человека рядом с вами, вы должны ввести:

Мой возраст == возраст человека рядом со мной

И утверждение было бы ИСТИННЫМ. Если бы вы были моложе человека напротив, вы бы сказали: Мой возраст <возраст человека напротив меня

И утверждение было бы ИСТИННЫМ. Однако если бы вы сказали следующее, а человек напротив был моложе вас:

Мой возраст <возраст человека напротив меня

Утверждение было бы ЛОЖНЫМ - правда в том, что это наоборот. Так думает цикл - если выражение истинно, продолжайте цикл. Если это ложь, не зацикливаться. Имея это в виду, давайте посмотрим на операторы (символы, представляющие действие), которые используются в логических выражениях:

Выражение	Функция
<	Менее чем
<=	Меньше или равно
>	Больше чем
> =	Больше или равно
! =	Не равно
<>	Не равно (альтернативно)
==	Равно

Не путайте = и == - оператор присваивания = делает то, что находится слева, равным тому, что находится справа. Оператор сравнения == говорит, совпадает ли элемент слева с тем, что находится справа, и возвращает True или False.

5.3.4 Условные выражения

Условные выражения - это когда часть кода запускается только при соблюдении определенных условий. Это похоже на цикл while, который вы только что написали, который выполняется только тогда, когда x не равно 0. Однако условные выражения выполняются только один раз. Самым распространенным условным условием в любом языке программирования является оператор if. Вот как это работает:

```
if {условия, которые должны быть выполнены}:
    {сделай это}
    {и это}
    {и это}
{но это происходит независимо}
{потому что это без отступа}
```

Теперь несколько примеров на Python:

```
#Пример 1
y = 1
if y == 1:
    print("y по-прежнему равно 1, я просто проверял")
```

y по-прежнему равно 1, я просто проверял

```
#Пример 2
print("Мы покажем четные числа до 20")
n = 1
while n <= 20:
    if n % 2 == 0:
        print(n)
    n = n + 1
print("готово")
```

Мы покажем четные числа до 20

2

4

6

8

10

12

14

16

18

20

готово

Пример 2 выглядит непростым. Но все, что мы сделали, это запускали оператор if каждый раз, когда запускается цикл while. Помните, что % просто означает остаток от деления - просто проверка того, что ничего не осталось, если число делится на два - показывая, что оно четное. Если он четный, он печатает, что такое n.

5.3.5 else и elif - операторы "когда это не так"

Есть много способов использования оператора if для ситуаций, когда ваше логическое выражение оказывается ЛОЖНЫМ. Это else и elif.

else просто сообщает компьютеру, что делать, если условия if не выполняются. Например, прочтите следующее:

```
a = 1
if a > 5:
    print("Это не должно выполниться.")
else:
    print("Это должно выполниться.")
```

Это должно выполниться.

а не больше пяти, поэтому то, что указано вelse, делается.

elif - это просто сокращенный способ сказатьelse if. Когда оператор if не может быть истинным, elif будет делать то, что под ним, ЕСЛИ выполняются условия. Например:

```
z = 4
if z > 70:
    print("Что-то не так")
elif z < 7:
    print("Все ок")
```

Все ок

Оператор if вместе с else и elif следуют этой форме:

```
if {#условия}:
    {#выполнить этот код}
elif {#условия}:
    {run this code}
elif {#условия}:
    {#выполнить этот код}
else:
    {#выполнить этот код}
# У вас может быть столько или меньше операторов elif, сколько вам нужно
#anywhere от нуля до неба.
# У вас может быть не более одного оператора else
# и только после всех остальных if и elifs.
```

Один из наиболее важных моментов, который следует помнить, - это то, что вы ДОЛЖНЫ иметь двоеточие : в конце каждой строки с if,elif, else илиwhile в нем.

5.3.6 Отступ

Еще один важнейший момент заключается в том, что код, который должен быть выполнен, если условия выполняются, ДОЛЖЕН БЫТЬ УКАЗАН. Это означает, что если вы хотите зациклить следующие пять строк с помощью цикла while, вы должны поставить заданное количество пробелов в начале каждой из следующих пяти строк. Это хорошая практика программирования на любом языке, но Python требует, чтобы вы это делали. Вот пример обоих вышеперечисленных пунктов:

```
a = 10
while a > 0:
    print(a)
    if a > 5:
        print("Большое число!")
    elif a % 2 != 0:
        print("Это нечетное число")
        print("И не больше пяти.")
    else:
        print("это число не больше 5")
        print("и не является нечетным")
        print("чувствовуете себя особенным?")
    a = a - 1
    print("мы просто сделали на единицу меньше, чем было!")
    print ("и если а не больше 0, мы выполним цикл снова.")
print ("ну, похоже, что 'a' теперь не больше 0!")
print ("цикл окончен, и без дальнейших действий, и эта программа тоже!")
```

10

Большое число!

мы просто сделали на единицу меньше, чем было!

и если а не больше 0, мы выполним цикл снова.

9

Большое число!

мы просто сделали на единицу меньше, чем было!

и если а не больше 0, мы выполним цикл снова.

8

Большое число!

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

7

Большое число!

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

6

Большое число!

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

5

Это нечетное число

И не больше пяти.

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

4

это число не больше 5

и не является нечетным

чувствовуете себя особенным?

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

3

Это нечетное число

И не больше пяти.

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

2

это число не больше 5

и не является нечетным

чувствовуете себя особенным?

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

1

Это нечетное число

И не больше пяти.

мы просто сделали на единицу меньше, чем было!

и если a не больше 0, мы выполним цикл снова.

ну, похоже, что 'a' теперь не больше 0!

цикл окончен, и без дальнейших действий, и эта программа тоже!

Обратите внимание на три уровня отступов:

1. Каждая строка первого уровня начинается без пробелов. Это основная программа, и она всегда будет выполняться.
2. Каждая строка на втором уровне начинается с четырех пробелов. Когда есть if или цикл на первом уровне, все на втором уровне после этого будет зациклено / ifed, пока новая строка снова не начнется на первом уровне.
3. Каждая строка третьего уровня начинается с восьми пробелов. Когда есть if или цикл на втором уровне, все на третьем уровне после этого будет зациклено / ifed, пока новая строка снова не начнется на втором уровне.
4. Это продолжается бесконечно, пока у человека, пишущего программу, не произойдет взрыв в мозгу, и он не сможет понять ничего, что он написал.

5.4 Функции

5.4.1 Введение

Углубимся в целенаправленное программирование. Это связано с пользовательским вводом, а для пользовательского ввода требуется вещь, называемая функциями.

Какие есть функции? По сути, функции - это небольшие автономные программы, которые выполняют определенную задачу, которую вы можете включить в свои собственные, более крупные программы. После того, как вы создали функцию, вы можете использовать ее в любое время и в любом месте. Это экономит ваше время и усилия, связанные с необходимостью пересказывать компьютеру, что делать каждый раз, когда он выполняет обычную задачу, например, заставляет пользователя что-то вводить.

5.4.2 Использование функции

Python имеет множество готовых функций, которые вы можете использовать прямо сейчас, просто «вызывая» их. «Вызов» функции подразумевает, что вы вводите функцию, и она возвращает значение (как переменная) в качестве вывода. Не понимаю? Вот общая форма вызова функции:

имя_функции (параметры)

- Function_name определяет, какую функцию вы хотите использовать (вы бы поняли ...). Например, функция raw_input, которая будет первой функцией, которую мы будем использовать.
- Параметры - это значения, которые вы передаете функции, чтобы сообщить ей, что она должна делать и как это делать ... например, если функция умножила любое заданное число на пять, то в параметрах указывается, какое число его следует умножить на пять. Введите число 70 в параметры, и функция выполнит 70 x 5.

5.4.3 Параметры и возвращаемые значения - взаимодействие с функциями

Что ж, это все хорошо, что программа может умножать число на пять, но что она для этого должна показать? Ваша программа должна видеть результаты того, что произошло, видеть, что такое 70 x 5, или видеть, есть ли где-то проблема (например, вы дали ей букву вместо числа). Итак, как функция показывает, что делает?

По сути, когда компьютер выполняет функцию, он видит не имя функции, а результат того, что функция сделала. Переменные делают то же самое - компьютер не видит имени переменной, он видит значение, которое содержит переменная. Назовем эту программу, которая умножает любое число на пять, multiply (). Вы помещаете в скобки число, которое хотите умножить. Итак, если вы набрали это:

a = умножить (70)

Компьютер действительно увидит это:

a = 350

Примечание: не вводите этот код - multiply () не является стандартной функцией, если вы не создадите ее.

Функция запустилась сама, а затем вернула номер в основную программу в зависимости от того, какие параметры ей были заданы.

Теперь давайте попробуем это с реальной функцией и посмотрим, что она делает. Функция называется input и просит пользователя что-то ввести. Затем он превращает его в строку текста. Попробуйте следующий код:

```
# эта строка делает 'a' равным тому, что вы вводите
a = input("Введите что-нибудь, и это будет повторяться на экране:")
# эта строка печатает, сколько значение 'a'
print(a)
```

Введите что-нибудь, и это будет повторяться на экране: 1

1

Скажем, в приведенной выше программе вы набрали hello, когда она попросила вас ввести что-то. На компьютере эта программа будет выглядеть так:

```
a = "hello"
print("hello")
```

Помните, что переменная - это просто сохраненное значение. Для компьютера переменная *a* не выглядит как *a* - она выглядит как значение, которое хранится внутри нее. Функции аналогичны - основной программе (то есть программе, в которой выполняется функция) они выглядят как значение того, что они дают в ответ на выполнение.

5.4.4 Программа-калькулятор

Напишем другую программу, которая будет выступать в роли калькулятора. На этот раз она будет выполнять что-то более масштабное, чем то, что мы делали раньше. Появится меню, в котором вас спросят, хотите ли вы перемножить два числа, сложить два числа, разделить одно число на другое или вычесть одно число из другого. Единственная проблема - функция input возвращает то, что вы вводите в виде строки - нам нужна цифра 1, а не буква 1 (и да, в Python есть разница).

К счастью, кто-то написал функцию eval, которая возвращает в основную программу то, что вы набрали, но на этот раз она вводит ее как число. Если вы вводите целое число (целое число), то на выходе получается целое число. И если вы поместите это целое число в переменную, переменная будет переменной целого типа, что означает, что вы можете складывать и вычитать и т. д.

```
# в этой строке «a» становится равным введенному вами значению. Он не принимает
a = eval(input("Введите что-нибудь, и это будет повторено на экране: "))
# эта строка печатает, чему равно 'a'
print(a)
```

Введите что-нибудь, и это будет повторено на экране: 1

1

А теперь давайте спроектируем этот калькулятор как следует. Мы хотим, чтобы меню возвращалось каждый раз, когда вы заканчиваете сложение, вычитание и т. Д. Другими словами, для цикла, вы говорите, что программа все еще должна работать. Мы хотим, чтобы в меню была опция, если вы введете это число. Это включает в себя ввод числа (также известного как ввод) и цикла if.

Давайте сначала напишем это на понятном английском языке (псевдокод):

```
НАЧАТЬ ПРОГРАММУ
вывести приветсвенное сообщение

 # Распечатайте, какие у вас есть варианты
  распечатать Вариант 1 - сложить
  печать Вариант 2 - вычесть
  печать Вариант 3 - умножить
  печать Вариант 4 - разделить
  вариант печати 5 - выйти из программы

  спросить, какой вариант ты хочешь
  если это вариант 1:
      спроси первое число
      спроси второе число
      сложите их вместе
      распечатать результат на экране
  если это вариант 2:
      спроси первое число
      спроси второе число
      вычесть одно из другого
```

распечатать результат на экране

если это вариант 3:

спроси первое число

спроси второе число

умножить!

распечатать результат на экране

если это вариант 4:

спроси первое число

спроси второе число

разделить одно на другое

распечатать результат на экране

если это вариант 5:

скажите циклу прекратить цикл

Распечатать на экране прощальное сообщение

КОНЕЦ ПРОГРАММЫ

Реализуем данный псевдокод при помощи Python:

```python
# программа-калькулятор

# Здесь мы определим наши функции
# это печатает главное меню и предлагает выбор
def menu():
    #печать списка доступных функций
    print("Добро пожаловать в программу calculator.py")
    print("Доступны следующие функции:")
    print(" ")
    print("1. Сложение")
    print("2. Вычитание")
    print("3. Умножение")
    print("4. Деление")
    print("5. Выход")
    print(" ")
    return eval(input("Осуществите выбор функции: "))

# функция сложения
def add(a,b):
    print(a, "+", b, "=", a + b)

#функция вычитания
def sub(a,b):
    print(b, "-", a, "=", b - a)

#функция умножения
def mul(a,b):
    print(a, "*", b, "=", a * b)

# функция деления
def div(a,b):
    print(a, "/", b, "=", a / b)

loop = 1
choice = 0
while loop == 1:
    choice = menu()
    if choice == 1:
        add(eval(input("Прибавить: ")),eval(input("к: ")))
    elif choice == 2:
        sub(eval(input("Вычесть: ")),eval(input("из: ")))
    elif choice == 3:
        mul(eval(input("Умножить: ")),eval(input("на: ")))
    elif choice == 4:
        div(eval(input("Разделить: ")),eval(input("на: ")))
    elif choice == 5:
        loop = 0

print("Спасибо за использование программы calculator.py!")
```

Добро пожаловать в программу calculator.py

Доступны следующие функции:

1. Сложение

2. Вычитание

3. Умножение

4. Деление

5. Выход

Осуществите выбор функции: 1

Прибавить: 1

к: 1

1 + 1 = 2

Добро пожаловать в программу calculator.py

Доступны следующие функции:

1. Сложение
2. Вычитание
3. Умножение
4. Деление
5. Выход

Осуществите выбор функции: 5

Спасибо вам за использование программы calculator.py!

Поэкспериментируйте с этим - попробуйте все варианты, вводя целые числа (числа без десятичной точки) и числа с добавлением после десятичной точки (известные в программировании как числа с плавающей запятой). Попробуйте ввести текст и посмотрите, как программа устраняет незначительное совпадение и перестает работать (с этим можно справиться, используя обработку ошибок, о которой мы поговорим позже).

5.4.5 Определите свои собственные функции

Хорошо, что вы можете использовать чужие функции, но что, если вы хотите написать свои собственные функции, чтобы сэкономить время и, возможно, использовать их в других программах? Здесь на помощь приходит оператор def. (Оператор - это просто то, что говорит Python, что делать, например, оператор + говорит Python добавить что-то, оператор if говорит Python что-то делать, если условия выполнены. .)

Вот как работает оператор def:

```
def function_name(параметр_1,параметр_2):
    {код функции}
    {больше кода}
    {больше кода}
    return {возвращаемое значение функции}
{этого кода нет в функции}
{потому что это без отступа}
# не забудьте поставить двоеточие ":" в конце
# строки, начинающейся с def
```

function_name это имя функции. Вы пишете код, который находится в функции под этой строкой, с отступом. (Мы поговорим о параметр_1 ипараметр_2 позже, а пока представьте, что между скобками нет ничего.

Функции выполняются полностью независимо от основной программы. Помните, я сказал, что когда компьютер переходит к функции, он видит не функцию, а значение, которое функция возвращает? Вот цитата:

Для компьютера переменная 'a' не похожа на 'a' - она выглядит как значение, которое хранится внутри нее. Функции аналогичны - основной программе (то есть программе, в которой выполняется функция) они выглядят как значение того, что они дают в ответ на выполнение.

Функция похожа на миниатюрную программу, которой передаются некоторые параметры - затем она запускается сама, а затем возвращает значение. Ваша основная программа видит только возвращаемое значение. Если бы эта функция прилетела на Луну и обратно, а затем в конце имела бы:

```
return "hello"
```

тогда вся ваша программа увидит строку «hello», где было имя функции. Он понятия не имел, что еще делает программа.

Поскольку это отдельная программа, функция не видит никаких переменных, которые есть в вашей основной программе, а ваша основная программа не видит никаких переменных, которые находятся в функции. Например, вот функция, которая выводит на экран слова «привет», а затем возвращает число «1234» в основную программу:

```
# Ниже приведено определение пользовательской функции
def hello():
    print("Привет")
    return 1234

# А вот вызов (использование) данной пользовательской функции
print(hello())
```

Привет

1234

1. При запуске def hello () была создана (объявлена) функция с именем hello
2. Когда была запущена строка print (hello ()), была выполнена функция hello (код внутри нее был запущен)
3. Функция hello напечатала на экране «hello», а затем вернула число «1234» обратно в основную программу.
4. Основная программа теперь видит строку как print (" 1234 ") и в результате выводит 1234.

Этим объясняется все, что произошло. Помните, что у основной программы НЕ было представления о том, что на экране были напечатаны слова hello. Все, что она увидела, было «1234», и распечатал его на экране.

5.4.6 Передача параметров функциям

Вспомните, как мы определяли функции:

```
def имя_функции (параметр_1, параметр_2):
    {это код функции}
    {дополнительный код}
    {дополнительный код}
    return {значение (например, текст или число), чтобы вернуться в осн
овную программу}
```

Где параметр_1 ипараметр_2 (в скобках), вы помещаете имена переменных, в которые вы хотите поместить параметры. Введите столько, сколько вам нужно, просто разделите их запятыми. Когда вы запускаете функцию, первое значение, которое вы помещаете в круглые скобки, переходит в переменную, где находится параметр_1. Второй (после первой

запятой) перейдет к переменной, в которой находится параметр_2. Это происходит для любого количества параметров функции (от нуля до неба).

Например:

```
def funnyfunction(first_word,second_word,third_word):
    print("Слово: " + first_word + second_word + third_word)
    return first_word + second_word + third_word
```

```
Когда вы запускаете (вызываете) указанную выше функцию, вы должны ввести что-то
вроде этого: `funnyfunction (" мясо "," людоед "," человек ")`. Первое значение
(то есть «мясо») будет помещено в переменную с именем first_word. Второе
значение в скобках (то есть «eater») будет помещено в переменную с именем
second_word и так далее. Вот как значения передаются из основной программы в
функции - в скобках после имени функции.

Добавьте новую строку в приведенный выше сценарий, вызывающий функцию.
```

5.4.7 Заключительная программа

Вспомните программу-калькулятор. Давайте перепишем её с использованием функций, чтобы она была более читаемой.

Сначала мы определим все функции, которые мы собираемся использовать, с помощью оператора def. Затем у нас будет основная программа, в которой весь этот беспорядочный код будет заменен красивыми, аккуратными функциями.

```
# программа-калькулятор

# Здесь мы определим наши функции
# это печатает главное меню и предлагает выбор
def menu():
    #печать списка доступных функций
    print("Добро пожаловать в программу calculator.py")
    print("Доступны следующие функции:")
    print(" ")
    print("1. Сложение")
    print("2. Вычитание")
    print("3. Умножение")
    print("4. Деление")
    print("5. Выход")
    print(" ")
    return eval(input("Осуществите выбор функции: "))

# функция сложения
def add(a,b):
    print(a, "+", b, "=", a + b)

#функция вычитания
def sub(a,b):
    print(b, "-", a, "=", b - a)

#функция умножения
def mul(a,b):
    print(a, "*", b, "=", a * b)

# функция деления
def div(a,b):
    print(a, "/", b, "=", a / b)

loop = 1
choice = 0
while loop == 1:
    choice = menu()
    if choice == 1:
        add(eval(input("Прибавить: ")),eval(input("к: ")))
    elif choice == 2:
        sub(eval(input("Вычесть: ")),eval(input("из: ")))
    elif choice == 3:
        mul(eval(input("Умножить: ")),eval(input("на: ")))
    elif choice == 4:
        div(eval(input("Разделить: ")),eval(input("на: ")))
    elif choice == 5:
        loop = 0

print("Спасибо за использование программы calculator.py!")
```

Добро пожаловать в программу calculator.py

Доступны следующие функции:

1. Сложение
2. Вычитание
3. Умножение
4. Деление
5. Выход

Осуществите выбор функции: 1

Прибавить: 2

к: 2

2 + 2 = 4

Добро пожаловать в программу calculator.py

Доступны следующие функции:

1. Сложение
2. Вычитание
3. Умножение
4. Деление
5. Выход

Осуществите выбор функции: 5

Спапсибо за использование программы calculator.py!

В исходной программе было 34 строки кода. В новом фактически было 35 строк кода! Он немного длиннее, но если вы посмотрите на него правильно, на самом деле он проще.

Вы определили все свои функции вверху. На самом деле это не часть вашей основной программы - это просто множество маленьких программ, которые вы вызовете позже. Вы даже можете повторно использовать их в другой программе, если они вам нужны, и вы не хотите указывать компьютеру, как снова складывать и вычитать.

Если вы посмотрите на основную часть программы (между строками loop = 1 иprint («Спасибо за ...»)), то это всего 15 строк кода. Это означает, что если бы вы захотели написать эту программу по-другому, вам нужно было бы написать всего около 15 строк, в отличие от 34 строк, которые вам обычно приходилось бы писать без функций.

5.4.8 Хитрые способы передачи параметров

Рассмотрим значение строкт add (eval (input (" Сложить: ")), eval (input (" с: "))).

Так выглядит вызов пользовательской функции add и передача ей значений параметров:

add (2,30)

После вызова функции, запустится её код,т.е. добавит 2 и 30, а затем распечатает результат. В программе добавления нет оператора return - он ничего не возвращает в основную программу. Он просто складывает два числа и выводит их на экран, а основная программа ничего из этого не видит.

Вместо `(eval (input ("Прибавь: ")), eval (input (" к: "))) в качестве параметров для программы сложения вы можете использовать переменные. Например.

число1 = 45
число2 = 7
сложиьть (число1, число2)

Что касается вышеизложенного, помните, что функция, в которую вы передаете переменные, не может изменять сами переменные - они просто используются как значения. Вы даже можете поместить значения прямо в функцию:

добавить (45,7)

Это потому, что единственное, что видит функция, - это значения, которые передаются в качестве параметров. Эти значения помещаются в переменные, которые упоминаются при определении add (строкаdef add (a, b)). Затем функция использует эти параметры для выполнения своей работы.

Коротко:

- Единственное, что функции основной программы видят - это параметры, которые ей передаются.
- Единственное, что основная программа видит среди функций, - это возвращаемое значение, которое она передает обратно.

Примечание: То есть, чтобы функции были доступны ВСЕ переменные программы, нужно внутри функции разрешить к ним доступ через global. Иначе будут созданы новые локальные переменные с аналогичным именем внутри функции. Подробнее про область видимости в Python читайте самостоятельно.

5.5 Кортежи, списки и словари

5.5.1 Введение

Вернемся к чему-то простому - переменным, но немного более сложным. Переменные хранят одну единицу информации. Но что, если вам нужно хранить длинный список информации? Например, названия месяцев в году. Или, может быть, длинный набор больших данных? Как бы Вы это сделали?

5.5.2 Решение - списки, кортежи и словари

Для этих трех проблем Python использует четыреразных решения - массивы, кортежи, списки и словари:

- Массивы - хранят набор данных одного и того же типа. Каждое из них нумеруется, начиная с нуля. Первый - имеет нулевой индекс, второй - 1, третий - 2 и т.д. Пример: строковые имена ваших контактов.
- Списки - тоже самое, что и массивы, но могут хранить значения не одного типа, а разных: строковых, числовых и других - и при этом хранить их одновременно. Вы можете удалять значения из списка и добавлять новые значения в конец. Пример: имена ваших контактов,

где имя задано не только строкой, но и числом или любым типом вперемешку...

- Кортежи похожи на списки, но вы не можете изменить их значения. Значения, которые вы даете ему первыми, - это значения, которых вы придерживаетесь для остальной части программы. Опять же, каждое значение пронумеровано, начиная с нуля, для удобства. Пример: названия месяцев в году.
- Словари похожи на то, что предполагает их название - словарь. В словаре у вас есть «указатель» слов и для каждого из них определение. В Python это слово называется «ключом», а определение - «значением». Значения в словаре не нумеруются - они также не в каком-то определенном порядке - ключ делает то же самое. Вы можете добавлять, удалять и изменять значения в словарях. Пример: телефонная книга.

5.5.2.1 Кортежи

Кортежи создаются следующим образом: вы даете кортежу имя, а затем список значений, которые он будет хранить. Например, месяцы года:

```
months = ('Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', \
'Июль', 'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь')
```

- Обратите внимание, что \ в конце первой строки переносит эту строку кода на следующую строку. Это полезный способ сделать большие строки более читабельными.
- Технически вам не нужно помещать эти круглые скобки (символы (и)), но лучше их ставить.
- У вас могут быть пробелы после запятых, если вы считаете это необходимым - на самом деле это не имеет значения

Затем Python организует эти значения в удобный пронумерованный индекс - начиная с нуля, в том порядке, в котором вы их вводили. Он будет организован следующим образом:

Индекс	Значение
0	Январь
1	Февраль
2	Март
3	Апрель
4	Май
5	Июнь
6	Июль
7	Август
8	Сентябрь
9	Октябрь
10	Ноябрь
11	Декабрь

Это и есть кортеж. Он прост для понимания.

5.5.2.2 Списки

Списки очень похожи на кортежи. Списки изменяемы (или 'mutable'- мутабельны, как может сказать программист), т.е. их значения могут быть изменены. В большинстве случаев мы используем списки, а не кортежи, потому что мы хотим легко изменить значения элементов, если нам нужно.

Списки определяются очень похоже на кортежи. Допустим, у вас есть ПЯТЬ кошек, которых зовут Том, Снэппи, Китти, Джесси и Честер. Чтобы поместить их в список, сделайте следующее:

```
cats = ['Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester']
```

Как видите, код в точности такой же, как у кортежа, ЗА ИСКЛЮЧЕНИЕМ того, что все значения заключены в квадратные скобки, а не в скобки. Опять же, вам не обязательно ставить пробелы после запятой.

Вы вызываете значения из списков точно так же, как и с кортежами. Например, чтобы напечатать имя вашего 3-го кота, вы должны сделать это:

```
print(cats[0:2])
```

['Tom', 'Snappy']

Вы также можете вспомнить ряд операторов работы со строками, например: - cats [0: 2] выведет именя котов из диапазона (0:2], т.е. имена ваших 1-го и 2-го котов. Попробуйте это в поле выше.

Списки сами по себе могут быть изменены. Чтобы добавить значение в список, вы используете функцию append () (append с англ. - добавить). Допустим, у вас появилась новая кошка по имени Кэтрин. Чтобы добавить ее в список, сделайте следующее:

```
cats.append('Catherine')
```

Используйте поле ниже, чтобы проверить, добавлена ли кошка в список.

В Python есть 2 индекса, которые указывают на последний элемент в списке.

- cats[len(cats) - 1] - По определению указывает на последний элемент. Функция len() возвращает длинну объекта, то есть в данном случае размер списка.
- cats[-1] - В python отрицательная индексация начинается с конца

```
print (cats[-1]) # вывод последнего элемента
print (cats[ len(cats) - 1 ]) # вывод последнего элемента
print (cats) # вывод всех элементов
```

Catherine

Catherine

['Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester', 'Catherine']

Синтаксис cats.append('Catherine') может показатся странным. Эта функция находится после точки после имени списка, в программировании это называется метод. Вы еще больше узнаете об этом на позже. А пока рассмотрим синтаксис функции, которая добавляет новое значение в список:

добавить новое значение в конец списка:

list_name.append(значение, которое вы хотите добавить)

#например. добавить число 5038 в список 'числа'

numbers.append(5038)

Для добавления множества элементов используйте метод extend().

Теперь к печальной ситуации - кот Снэппи был случайно застрелен соседом. Вам необходимо удалить его из списка. К счастью, удалить этого кота - простая задача:

```
#Удаление кота с индексом 1, то есть второго кота.
del cats[1]
```

Еще раз проверьте, какие кошки есть в списке:

```
print (cats)
```

['Tom', 'Kitty', 'Jessie', 'Chester', 'Catherine']

Кот Снэппи действительно удален.

5.5.3 Словари

Рассмотрим другой пример. Вам нужно позвонить своей сестре, маме, сыну и всем, кому нужно знать, что их любимый кот мертв. Для этого вам понадобится телефонная книга.

Итак, списки, которые мы использовали выше, не совсем подходят для телефонной книги. Вам нужно знать число (представляющее собой телефонный номер), основанное на чьем-то имени, а не наоборот, как мы сделали с кошками. В примерах месяцев и кошек мы дали компьютеру номер, а он дал нам имя. На этот раз мы хотим дать компьютеру имя, и он дает нам номер. Для этого нам понадобятся * Словари *.

У словарей есть пары ключ-значение. В телефонной книге есть имена людей, а затем их номера. Видите сходство?

Когда вы изначально создаете словарь, это очень похоже на создание кортежа или списка. Кортежи содержат элементы (и), списки содержат элементы [и]. В словаряхэлементы содержатся в { и }, т.е. в фигурных скобках. Ниже представлен пример, показывающий словарь с четырьмя телефонными номерами в нем:

```
#Создать телефонную книгу, представляющую собой словарь Python
phonebook = {'Andrew Parson':8806336, \
'Emily Everett':6784346, 'Peter Power':7658344, \
'Lewis Lame':1122345}
print('Телефон контакта \"Lewis Lame\": ',phonebook['Lewis Lame'])
```

Телефон контакта "Lewis Lame": 1122345

Номер Льюиса Ламе напечатан на экране. Обратите внимание, как вместо определения значения числовым индексом, как в примерах с кошками и месяцами, мы определяем значение, используя другое значение в качестве индекса, или ключа - в данном случае имя человека.

Теперь добавим в книгу новые телефонные номера:

```
## Добавьте человека 'Gingerbread Man' (по русски - "Колобок") в телефонную книг

phonebook['Gingerbread Man'] = 1234567
```

Все, что говорит эта строка, это то, что в телефонной книге есть человек по имени Колобок, и его номер - 1234567. Другими словами, ключ - Gingerbread Man, а значение -1234567.

Проверьте, добавлен ли он, используя поле ниже.

```
if phonebook.get('Gingerbread Man') != None:
    print ('Да, Gingerbread Man добавлен')
```

Да, Gingerbread Man добавлен

Вы удаляете записи в словаре, как в списке. Допустим, Эндрю Парсон - ваш сосед и застрелил вашу кошку. Вы больше никогда не захотите с ним разговаривать, а значит, вам не нужен его номер. Как и в списке, вы должны сделать это:

```
del phonebook['Andrew Parson']
```

Оператор del удаляет любую функцию, переменную или запись в списке или словаре (запись в словаре - это просто переменная с числом или текстовой строкой в качестве имени. Это пригодится позже).

Проверим, пропал ли номер, используя первое поле ниже. Если phonebook.get() возвращает None (можно перевести как "ничего/никто/нет/пустой, эквивалент нуля), значит такой записи нет.

```
print (phonebook.get('Andrew Parson'))
```

None

```
#а это вызовет ошибку, ведь мы пытаемся обратится к несуществующему ключу
print (phonebook['Andrew Parson'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-13-5fab7f37904b> in <module>
      1 #а это вызовет ошибку, ведь мы пытаемся обратится к несуществующему клю
чу
----> 2 print (phonebook['Andrew Parson'])

KeyError: 'Andrew Parson'
```

Помните ту функцию добавления (append()), которую мы использовали со списком? Подобных функций и методов довольно много, и их тоже можно использовать со словарями. Ниже я напишу вам программу, и в нее будут включены некоторые из этих функций. В коде есть комментарии, объясняющие, что он делает.

Проведите несколько экспериментов на свой вкус.

```
# Несколько примеров работы со словарём

# Сначала определяем словарь
# на этот раз в нем ничего не будет
возраст = {}
ages = {}

#Добавляем несколько имен в словарь
ages['Sue'] = 23
ages['Peter'] = 19
ages['Andrew'] = 78
ages['Karren'] = 45

# Используем оператор if, чтобы найти ключ в списке.
# Помните как работают операторы if -
# они выполняют код внутри своего тела, если что-то ИСТИНО
# и не выполняют, когда что-то ЛОЖНО.
if 'Sue' in ages:
    print("Sue есть в словаре. Ей", \
ages['Sue'], " лет")

else:
    print("Sue нет в словаре.")

#Используйте функцию keys() -
#Эта функция возвращает список
# всех названий ключей. Например:
print("В словаре есть следующие люди: ")
print(ages.keys())

# Вы можете использовать эту функцию, чтобы
# поместить все имена ключей в список:
keys = ages.keys()

# Вы также можете получить список
# всех значений в словарепри помощи метода values ():
print("Возраст людей следующий: ", \
ages.values())

## Поместите возраст в список:
values = ages.values()

# Вы можете сортировать списки с помощью функции sorted ()
# Отсортируем все значения в списке
# алфавитно, численно и т. д.
# Словари нельзя сортировать -
# они в произвольном порядке
print(keys)
sortedkeys = sorted(keys)
print(sortedkeys)

print(values)
sortedvalues = sorted(values)
print(sortedvalues)

# Вы можете узнать количество записей
# при помощи функции len ():
print("В словаре ", \
len(ages), " записей")
```

Sue есть в словаре. Ей 23 лет

В словаре есть следующие люди:

dict_keys(['Sue', 'Peter', 'Andrew', 'Karren'])

Возраст людей следующий: dict_values([23, 19, 78, 45])

dict_keys(['Sue', 'Peter', 'Andrew', 'Karren'])

['Andrew', 'Karren', 'Peter', 'Sue']

dict_values([23, 19, 78, 45])

[19, 23, 45, 78]

В словаре 4 записей

5.5.4 Массивы

Отличия массивов от словарей представлены ниже. Основное отличие, что массивы хранят данные только в одном конкретном типе. Обычно они используются только при системном программировании на Python, поскольку там могут повысить скорость выполнения кода. Списки более удобны и работают с хорошей скоростью, поэтому лучше пользоваться ими.

Список	Массив
Может состоять из элементов, принадлежащих к разным типам данных	Состоит только из элементов, принадлежащих к одному типу данных
Нет необходимости явно импортировать модуль для объявления	Необходимо явно импортировать модуль (import array) для объявления
Невозможно напрямую обрабатывать арифметические операции	Может напрямую обрабатывать арифметические операции
Могут быть вложенными для размещения элементов различного типа	Должны содержать либо все вложенные элементы одинакового размера
Предпочтительно для более короткой последовательности элементов данных	Предпочтительно для более длинной последовательности элементов данных
Большая гибкость позволяет легко изменять (добавлять, удалять) данные	Меньшая гибкость после добавления, удаление должно выполняться поэлементно
Весь список можно распечатать без явного зацикливания	Цикл должен быть сформирован для печати или доступа к компонентам массива
Увеличивает объем памяти для удобного добавления элементов	Сравнительно компактнее по объему памяти

Код типа	С тип	Python тип	Минимальный размер (байт)
b	signed char	int	1
B	unsigned char	int	1
u	Py_UNICODE	Unicode символ; устарело с Python 3.3	2
h	signed short	int	2
H	unsigned short	int	2
i	signed int	int	2
I	unsigned int	int	2
l	signed long	int	4
L	unsigned long	int	4
q	signed long long	int	8
Q	unsigned long long	int	8
f	float	float	4
d	double	float	8

Сравните два поля ниже. В первом - работа с массивом, во втором - работа со списком.

```
#Пример - массив
import array

arr=array.array('i',) #объявление массива
arr=[1,3,4] #определение массива
print (arr)
```

[1, 3, 4]

```
#Пример - список
myList=list() #объявление списка
myList = [1,3,4] #определение списка
print (myList)
```

[1, 3, 4]

Да, как можно заметить, сперва можно объявлять (создавать) что-либо можно, а потом определять (т.е. заполнять значениями). Это полезно, когда код большой, очень гибкий и легко масштабируемый.

5.6 Цикл for

5.6.1 Введение

Ранее был рассмотрен цикл while. На текущий момент вы получили базовые понятия о списках, поэтому теперь можно рассмотреть цикл for.

5.6.2 Цикл for

Цикл for часто используется, чтобы что-то делать с каждым значением в списке, или повторить какой-либо участок кода несколько раз.

Рассмотрим пример:

```
# Пример цикла for
# Сначала создадим список:
newList = [45, 'eat me', 90210, "The day has come, the walrus said, \
to speak of many things", -67]

# создаем цикл:
# цикл проходит по каждому элементу списка newList и печатаем его
for value in newList:
    print(value)
```

45

eat me

90210

The day has come, the walrus said, to speak of many things

-67

Когда цикл выполняется, происходит последовательный перебор все значений в списке, указанном после in. На каждом новом шаге (итерации) цикла значение "активного" элемента списка (т.е. элемента с индексом, соответсвующим номеру текущей итерации выполнения данного цикла) в записывается переменную value и печатается.

Рассмотрим ещё один аналогичный пример.

```
word = 'Привет'
index = 0
for letter in word:
    print("итерация",index,letter)
    index += 1 #прибавить к индексу 1
```

итерация 0 П

итерация 1 р

итерация 2 и

итерация 3 в

итерация 4 е

итерация 5 т

Обратим внимание:

- Как видите, строки - это просто списки с большим количеством символов.
- Программа последовательно перебрала каждую букву (или значение) в слове и распечатала их на экране.

Однако, пройти весь список целиком можно было и при помощи цикла while, например так:

```
word = 'Привет'
index = 0
while index < len(word): #не определенные ранее переменные по умолчанию получают
    print("итерация",index, word[index])
    index += 1 #прибавить к индексу 1
```

итерация 0 П

итерация 1 р

итерация 2 и

итерация 3 в

итерация 4 е

итерация 5 т

Как нам быть, если нужно перебирать что-то в цикле, но в каком-то конкретном заданном диапазоне?

5.6.2.1 Функция range()

Range переводится как "диапазон". Данная функция может принимать один, два или три аргумента.

- Если задан только один, то генерируются числа от 0 до указанного числа, не включая его.
- Если заданы два, то числа генерируются от первого до второго, не включая его.
- Если заданы три, то третье число – это шаг.

Функция range() генерирует последовательность целых чисел в указанном диапазоне. Так, range(5, 11) сгенерирует последовательность 5, 6, 7, 8, 9, 10. Однако это будет не структура данных типа "список". Функция range() производит объекты своего класса – диапазоны:

```
a = range(-10, 10)
print (a)
print (type(a))
```

range(-10, 10)

<class 'range'>

Несмотря на то, что мы не видим последовательности чисел, она есть, и мы можем обращаться к ее элементам:

```
print ('первый элемент: ',a[0])
print ('последний элемент:',a[-1])
```

первый элемент: -10

последний элемент: 9

Обратите внимание, что последний элемент последовательности равен 9, а не 10. Это потому что кодом range(-10, 10) числа генерируются от первого включительно и до второго не включительно параметра.

Итак, зачем нам понадобилась функций range() в теме про цикл for? Дело в том, что вместе они образуют неплохой тандем. For как цикл перебора элементов, в отличие от while, позволяет не следить за тем, достигнут ли конец структуры. Не надо вводить счетчик для этого, изменять его и проверять условие в заголовке. С другой стороны, range() дает последовательность целых чисел, которые можно использовать как индексы для элементов того же списка.

```
#получим диапазон индексов букв в слове
#функция len() возвращает длинну объекта, т.е. в данном случае- длинну слова
range(len(word))
```

range(0, 6)

Здесь с помощью функции len() измеряется длина списка. В данном случае она равна 6. После этого число 6 передается в функцию range(), и она генерирует последовательность чисел от 0 до 5 включительно. Это как раз индексы элементов нашего списка. Теперь "соединим" for и range():

```
index = 0
for index in range(len(word)):
    print("итерация",index, word[index])
```

итерация 0 П

итерация 1 р

итерация 2 и

итерация 3 в

итерация 4 е

итерация 5 т

В заголовке цикла for берутся элементы вовсе не списка, а объекта range. Список, элементы которого планируется перезаписывать, тут по-сути не фигурирует. Если заранее знать длину списка, то заголовок может выглядеть так: for index in range(6). То, как используется index в теле цикла, вопрос второй. Примечание. Вместо названия идентификатора index может быть любой другой, или даже несколько.

Ещё есть весьма полезная функция zip(), которая берёт на вход несколько списков и создаёт из них список (в Python 3 создаётся не list, а специальный zip-объект) кортежей, такой, что первый элемент полученного списка содержит кортеж из первых элементов всех списков-аргументов. Таким образом, если ей передать три списка, то она отработает следующим образом:

```
list1 = 'abc'
list2 = [10, 20, 30]
print (list(zip(list1,list2)))
```

[('a', 10), ('b', 20), ('c', 30)]

При помощи функции zip() мы можем сделать циклом for более непонятным (попробуйте понять как это работает):

```
my_list = [['Петя', 'Коля','1'], ['Троечник', 'Отличник', '2345']]

new_list=list(zip(my_list[0],my_list[1]))
print (new_list)

print ('----\n')

for i, j in new_list:
    print(i,j)

print ('----\nТоже самое, обратите внимание на индексы В КОДЕ:')
print (new_list[0][0],new_list[0][1])
print (new_list[1][0],new_list[1][1])
print (new_list[2][0],new_list[2][1])
```

[('Петя', 'Троечник'), ('Коля', 'Отличник'), ('1', '2345')]

Петя Троечник

Коля Отличник

1 2345

Тоже самое, обратите внимание на индексы В КОДЕ:

Петя Троечник

Коля Отличник

1 2345

Т.е. написав после for две переменных, первая будет итератором верхнего уровня списка, а вторая - итератором более низкого уровня списка: new_list[i][j].

5.6.3 Создание функции "меню"

Приступим к написанию более сложных программ. До сих пор мы изучили переменные, списки, циклы и функции. Это почти все, что нам нужно для базового программирования.

Итак, давайте поставим перед собой задачу:

- Программа запрашивает строку со всеми пунктами меню в ней, и текстовую строку с вопросом.
- Необходимо наличие проверки, что каждый пункт меню уникален.

```
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print (") " + entry)
    return eval(input(question)) - 1
```

def menu (list, question): функция требует два параметра для работы:

- список всех пунктов меню,
- вопрос, который она задаст, когда все параметры будут напечатаны

Для каждой записи в списке выполните следующие действия:

- print(1 + list.index(entry),end="")) использует функцию .index () для поиска где в списке находится запись. Функция print затем распечатает ее номер увеличенный на 1, чтобы нумерация начиналась не с 0, а с 1 и была более понятной
- print ")" + entry печатает скобку, а затем имя записи
- после завершения цикла for eval(input(question) - 1) задает вопрос, и возвращает значение индекса пункта меню в основную программу

Фактически, программа заняла всего пять строк.

Было бы хорошо, если все описанные вне кода комментарии мы бы оставили в самом коде, используя символ #. Помните, что если вы собираетесь публиковать свой код с открытым исходным кодом, многие люди будут проверять написанный вами код, и чтобы его можно было быстрее понять, лучше оставлять комментарии внутри кода.

5.6.4 Наша первая «игра»

В качестве примера рассмотрим текстовую приключенческую игру? Он будет происходить в одной комнате дома и будет очень простой. Будет пять вещей и дверь. В одной из пяти вещей спрятан ключ от двери. Игроку нужно

найти ключ, затем с его помощью открыть дверь. Сначала опишем программу на русском языке, а затем реализуем её на Python:

Вывести доступные функции меню программы

Вывести приветственное сообщение с описанием комнаты.
Выведем список из шести доступных вещей: горшечные растения, живопись, \ ваза, абажур, туфля и дверь

Компьютер считает, что дверь заперта и знает где ключ

Ввести меню, показывающее, чем вы можете «управлять»:
вывести 6 предметов, один из которых пользователь может выбрать, чтобы посмотреть

если пользователь захотел посмотретьна предмет:
    Горшечное растение:
        Если ключ здесь, отдайте ключ игроку
        в противном случае скажите им, что его здесь нет
    картина:
        то же, что и выше
    и т.п.
    дверь:
        Если у игрока есть ключ, пусть откроет дверь
        В противном случае попросите их присмотреться

Сообщить игроку о завершении игры.

Исходя из этих исходных данных, мы можем написать программу.

```python
# Текстовая приключенческая игра

#функция, реализующая функциональность меню
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print (") " + entry)

    return eval(input(question)) - 1

# Сообщим компьютеру основную информацию о комнате:
items = ["Цветок в горшке","Картина","Ваза","Абажур","Обувь","Дверь"]

# Ключ находится в вазе (или запись номер 2 в списке выше):
keylocation = 2

# Переменная, в значении которой хранится нашли вы ключ или нет. 0 - не нашли
keyfound = 0

loop = 1 #переменная для зацикливания

# Дайдим вводный текст:
print("Прошлой ночью вы легли спать, не выходя из собственного дома.")

print("Теперь вы заперты в комнате. Вы не знаете как")
print("вы туда попали или сколько сейчас времени. В комнате вы видите")
print(len(items), "вещи:")
for x in items:
    print(x)
print("")
print("Дверь заперта. Может быть где-нибудь можно найти ключ?")

#Пусть ваше меню выводится циклически, пока вы не найдете ключ:
while loop == 1:
    choice = menu(items,"Что вы хотите проверить? ")
    if choice == 0:
        if choice == keylocation:
            print("Вы нашли ключик в горшке.")

            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в горшке.")
            print("")
    elif choice == 1:
        if choice == keylocation:
            print("Вы нашли маленький ключик за картиной.")
            print("")

            keyfound = 1
        else:
            print("Вы ничего не нашли за картиной.")
            print("")
    elif choice == 2:
```

```
        if choice == keylocation:
            print("Вы нашли в вазе маленький ключик.")
            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в вазе.")

            print("")
    elif choice == 3:
        if choice == keylocation:
            print("Вы нашли маленький ключик в абажуре.")
            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в абажуре.")
            print("")

    elif choice == 4:
        if choice == keylocation:
            print("Вы нашли ключик в туфле.")
            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в туфле.")
            print("")
    elif choice == 5:
        if keyfound == 1:
            loop = 0
            print("Вы вставляете ключ, поворачиваете его и слышите щелчок")

            print("")
        else:
            print("Дверь заперта, нужно найти ключ.")
            print("")

# Помните, что обратная косая черта продолжает
# код в следующей строке
print("Свет заливает комнату так, как будто \
вы открываете дверь к своей свободе.")
```

Прошлой ночью вы легли спать, не выходя из собственного дома.

Теперь вы заперты в комнате. Вы не знаете как

вы туда попали или сколько сейчас времени. В комнате вы видите

6 вещи:

Цветок в горшке

Картина

Ваза

Абажур

Обувь

Дверь

Дверь заперта. Может быть где-нибудь можно найти ключ?

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 3

Вы нашли в вазе маленький ключик.

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 2

Вы ничего не нашли за картиной.

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 5

Вы ничего не нашли в туфле.

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 2

Вы ничего не нашли за картиной.

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 3

Вы нашли в вазе маленький ключик.

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 6

Вы вставляете ключ, поворачиваете его и слышите щелчок

Свет заливает комнату так, как будто вы открываете дверь к своей свободе.

Очень простая, но веселая игра. Пусть вас не пугает объем кода, 53 строки - это просто операторы if, которые легче всего читать (как только вы поймете все отступы).

5.6.5 Улучшение игры

Функция menu () сокращает объем кода. Цикл while, который у нас есть, плохо читаем - четыре уровня отступов для простой программы. Давайте немного перепишем код, чтобы он стал более читаемым.

Код сильно улучшится, когда мы введем классы. Но это будет рассмотрено позже. А пока давайте создадим функцию, которая улучшить читабильность кода. Мы передадим её две вещи - сделанный нами выбор меню и расположение клавиши. Он вернет одно - был ли ключ найден.

```
def inspect(choice,location):
    if choice == location:
        print("\nВы нашли ключ!\n")
        return 1
    else:
        print("\nНичего интересного тут...\n")
        return 0
```

Теперь основная программа cnfytn немного проще.

В поле ниже:

- Вставьте функцию inspect
- Замените цикл while следующим кодом:

```
while loop == 1:
    keyfound = inspect(menu(items,"Что вы хотите проверить? "),keylocati
on)
    if keyfound == 1:
        print("Вы вставляете ключ в замок двери, поворачиваете его и слы
шите щелчок!")
        loop = 0
```

```
# Текстовая приключенческая игра

#функция, реализующая функциональность меню
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print (") " + entry)

    return eval(input(question)) - 1

# Сообщим компьютеру основную информацию о комнате:
items = ["Цветок в горшке","Картина","Ваза","Абажур","Обувь","Дверь"]

# Ключ находится в вазе (или запись номер 2 в списке выше):
keylocation = 2

# Переменная, в значении которой хранится нашли вы ключ или нет. 0 - не нашли
keyfound = 0

loop = 1 #переменная для зацикливания

# Дайдим вводный текст:
print("Прошлой ночью вы легли спать, не выходя из собственного дома.")

print("Теперь вы заперты в комнате. Вы не знаете как")
print("вы туда попали или сколько сейчас времени. В комнате вы видите")
print(len(items), "вещи:")
for x in items:
    print(x)
print("")
print("Дверь заперта. Может быть где-нибудь можно найти ключ?")
#Пусть ваше меню выводится циклически, пока вы не найдете ключ:
while loop == 1:
    choice = menu(items,"Что вы хотите проверить? ")
    if choice == 0:
        if choice == keylocation:
            print("Вы нашли ключик в горшке.")

            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в горшке.")
            print("")
    elif choice == 1:
        if choice == keylocation:
            print("Вы нашли маленький ключик за картиной.")
            print("")

            keyfound = 1
        else:
            print("Вы ничего не нашли за картиной.")
            print("")
    elif choice == 2:
        if choice == keylocation:
            print("Вы нашли в вазе маленький ключик.")
            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в вазе.")

            print("")
    elif choice == 3:
        if choice == keylocation:
            print("Вы нашли маленький ключик в абажуре.")
            print("")
```

```
            keyfound = 1
        else:
            print("Вы ничего не нашли в абажуре.")
            print("")

    elif choice == 4:
        if choice == keylocation:
            print("Вы нашли ключик в туфле.")
            print("")
            keyfound = 1
        else:
            print("Вы ничего не нашли в туфле.")
            print("")
    elif choice == 5:
        if keyfound == 1:
            loop = 0
            print("Вы вставляете ключ, поворачиваете его и слышите щелчок")

            print("")
        else:
            print("Дверь заперта, нужно найти ключ.")
            print("")

# Помните, что обратная косая черта продолжает
# код в следующей строке
print("Свет заливает комнату так, как будто \
вы открываете дверь к своей свободе.")
```

Прошлой ночью вы легли спать, не выходя из собственного дома.

Теперь вы заперты в комнате. Вы не знаете как

вы туда попали или сколько сейчас времени. В комнате вы видите

6 вещи:

Цветок в горшке

Картина

Ваза

Абажур

Обувь

Дверь

Дверь заперта. Может быть где-нибудь можно найти ключ?

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 5

Вы ничего не нашли в туфле.

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 3

Вы нашли в вазе маленький ключик.

1) Цветок в горшке

2) Картина

3) Ваза

4) Абажур

5) Обувь

6) Дверь

Что вы хотите проверить? 6

Вы вставляете ключ, поворачиваете его и слышите щелчок

Свет заливает комнату так, как будто вы открываете дверь к своей свободе.

Теперь программа стала намного короче и читабельнее - её размер уменьшился с 83 строк до 50! Конечно, вы теряете немного универсальности - все предметы в комнате теперь реагируют одинаково. Вы автоматически открываете дверь, когда находите ключ. Игра становится чуть менее интересной.

5.7 Классы

5.7.1 Введение

Использование функций сильно сокращает объем кода, повышает его читаемость и масштабриуемость (т.е. код становится более гибким). Например, вы можете много раз в разных частях вашей программы выполнять одну и ту же функцию, а потом легко её изменить, если потребуются доработки и вам при этом не придется переделывать весь код.

У функций есть свои ограничения. Функции не хранят никакой информации, как это делают переменные - каждый раз, когда функция запускается, она запускается заново. Однако некоторые функции и переменные очень тесно связаны друг с другом. Например, представьте, что у вас есть класс "клюшка для гольфа". Он содержит информацию о нем (то есть о переменных), например о длине вала, материале рукоятки и материале головки. У него также есть функции, связанные с ним, такие как функция раскачивания вашей клюшки или функция разбивания клюшки при чистом разочаровании. Для этих функций вам необходимо знать все эти переменные клюшки.

Когда клюшка выполняет какую-либо свою функцию, она меняет свои переменные, например переменная "усталость клюшки" увеличивается при каждом вызове функции "воспользоватся клюшкой". Необходим, способ сгруппировать тесно связанные функции и переменные в одном месте, чтобы они могли взаимодействовать друг с другом.

Если у вас есть несколько клюшек для гольфа, вы можете сделать базовый класс "клюшка", от которого будут использоваться стандартные вещи для клюшек, а более тонкие особенности будут реализовываться в отдельном классе, созданном вами для каждой конкретной клюшки. Таким образом создание программного гольф-клуба сильно упростится.

Для решения подобных задач используется такой подход, как объектно-ориентированное программирование. Он объединяет функции и переменные таким образом, чтобы они могли видеть друг друга и работать вместе, тиражироваться и изменяться по мере необходимости. И для этого мы используем так называемые классы.

5.7.2 Создание класса

Что такое класс? Думайте о классе как о проекте. Это не что-то само по себе, это просто описывает, как что-то сделать. Вы можете создать множество объектов из этого чертежа, известного в ООП как * экземпляр * класса.

Классы создаются при помощи оператора class:

```
# Объявление класса
class class_name:
    [выражение 1]
    [выражение 2]
    [и т.д.]
```

Добавим конкретики - посмотрим на пример класса Shape (с англ. фигура):

```
#Пример класса
class Shape:
    def __init__(self,x,y):
        self.x = x #размеры по оси x и y
        self.y = y
    description = "Эта форма еще не описана"
    author = "У этой формы ещё нет автора"
    def area(self):  #функция вычисления площади
        return self.x * self.y
    def perimeter(self):  #функция вычисления периметра
        return 2 * self.x + 2 * self.y
    def describe(self,text):  #функция установки описания
        self.description = text
    def authorName(self,text): #функция установки авторства
        self.author = text
    def scaleSize(self,scale): #функция масштабирования
        self.x = self.x * scale
        self.y = self.y * scale
```

Вы создали описание формы (то есть её переменных) и то, какие операции вы можете делать с этой формой (то есть функции). Это очень важно - вы создали не настоящую форму, а просто описание того, что такое форма. Форма имеет ширину (x), высоту (y), а также площадь и периметр (area (self) и perimeter (self)). Когда вы определяете класс, код не запускается - вы просто создаете функции и переменные.

Функция под названием __init__ запускается, когда мы создаем экземпляр Shape, то есть, когда мы создаем фактическую форму, в отличие от имеющегося у нас `` чертежа '', запускается __init__. Как это работает, вы поймете позже.

self - это отношение элементов класса к классу и другим элементам этого класса. self - это первый параметр в любой функции, определенной внутри класса. Любая функция или переменная, созданная на первом уровне отступа (то есть строки кода, начинающиеся на одну вкладку справа от того места, где мы помещаем класс Shape, автоматически помещается в self. Чтобы получить доступ к этим функциям и переменным в другом месте внутри класса, их имени должны перед ними написать self и точку (например, self.variable_name). Без self вы можете использовать переменные только внутри функции, в которой они определены, а не в других функциях того же класса `.

5.7.3 Использование class

Ниже представлен пример того, что называется созданием экземпляра класса. Учтите, что приведенный должен был бы быть запущен.

```
rectangle = Shape(100,45)
```

Сперва действует функция инициализации __init__. Мы создаем экземпляр класса, сначала определяя его имя (в данном случае Shape), а затем в скобках значения, передаваемые в функцию__init__. Функция init запускается (используя параметры, которые вы указали в скобках), а затем

выдает экземпляр этого класса, которому в данном случае присваивается имя «rectangle».

Экземпляр класса rectangle можно считать некоторой замкнутой коллекцией переменных и функций. Поскольку мы присвоили экземпляру класса имя rectangle(прямоугольник), для доступа к функциям и переменным данного экземпляра класса извне мы теперь можем писать, например rectangle.perimeter().

Посмотрим как работает показанный ранее код:

```
class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    description = "Эта форма еще не описана"
    author = "У этой формы ещё нет автора"
    def area(self):
        return self.x * self.y
    def perimeter(self):
        return 2 * self.x + 2 * self.y
    def describe(self,text):
        self.description = text
    def authorName(self,text):
        self.author = text
    def scaleSize(self,scale):
        self.x = self.x * scale
        self.y = self.y * scale

rectangle = Shape(100,45)

#вычисление площади прямоугольника
print(rectangle.area())

#вычисление периметра прямоугольника
print(rectangle.perimeter())

#установка описания прямоугольника прямоугольника
rectangle.describe("Ширина прямоугольник более чем в два раза больше его высоты"

#уменьшение прямоугольника на 50%
rectangle.scaleSize(0.5)

#вывод новой площади прямоугольника
print(rectangle.area())
```

4500

290

1125.0

Как видите, если self будет использоваться внутри экземпляра класса, назначенное ему имя используется вне класса. Мы делаем это для просмотра и изменения переменных внутри класса, а также для доступа к функциям, которые там есть.

Мы не ограничены одним экземпляром класса - у нас может быть столько экземпляров, сколько захотим. Например:

longrectangle = Shape(120,10) #длинный прямоугольник

fatrectangle = Shape(130,120) #жирный прямоугольник

и как longrectangle, так и fatrectangle имеют свои собственные функции и переменные, содержащиеся внутри них - они полностью независимы друг от друга. Нет ограничений на количество экземпляров, которые мы можем создать

Поэкспериментируйте с несколькими разными экземплярами в поле выше.

5.7.4 Базовая терминология

Рассмотрим основные положения объектно-ориентированного программирования:

- Когда мы впервые описываем класс, мы * определяем * его (так же как и с функциями)
- Возможность группировать похожие функции и переменные вместе называется * инкапсуляцией *
- Переменная внутри класса называется * атрибутом * класса
- Функция внутри класса называется * методом * класса
- Класс находится в той же категории вещей, что и переменные, списки, словари и т. Д. То есть все они * объекты *
- Класс известен как «структура данных» - он содержит данные и методы их обработки.

5.7.5 Наследование

Давайте еще раз взглянем на введение. Мы знаем, как классы группируют вместе переменные и функции, известные как атрибуты и методы, так что и данные, и код для их обработки находятся в одном месте. Мы можем создать любое количество экземпляров этого класса, поэтому нам не нужно писать новый код для каждого нового объекта, который мы создаем. Но как насчет добавления дополнительных функций в элементы нашего гольф-клуба? Здесь в игру вступает * наследование *.

Мы определяем новый класс на основе другого, «родительского» класса. Наш новый класс переносит все от родителя, и мы также можем добавить к нему другие элементы. Если какие-либо новые атрибуты или методы имеют то же имя, что и атрибут или метод в нашем родительском классе, они используются вместо родительского. Рассмотрим вновь класс Shape:

```
class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    description = "Эта форма еще не описана"
    author = "У этой формы ещё нет автора"
    def area(self):
        return self.x * self.y
......................
```

Если бы мы хотели определить новый класс, скажем, квадрат, на основе нашего предыдущего класса Shape, мы бы сделали это (обратите внимания, что у квадрата стороны равны):

```
class Square(Shape):
    def __init__(self,x):
        self.x = x
        self.y = x
```

Это похоже на обычное определение класса, но на этот раз мы указали в скобках после имени родительский класс, от которого мы унаследовали все атрибуты, методы и т.п. Как видите, из-за этого мы очень * быстро * смогли описать квадрат. Это потому, что мы унаследовали все от класса shape и

изменили только то, что нужно было изменить. В данном случае мы переопределили только функцию __init__ в Shape так, чтобы значения X и Y были одинаковыми.

Создадим еще один новый класс, на этот раз унаследованный от Square. Это будут два квадрата, один сразу слева от другого:

```
# По форме это будет выглядть примерно так:
# _____________
#|     |       |
#|     |       |
#|_____|_______|

class DoubleSquare(Square): #класс "двойной квадрат"
    def __init__(self,y):
        self.x = 2 * y
        self.y = y
    def perimeter(self):
        return 2 * self.x + 3 * self.y
```

На этот раз нам также пришлось переопределить функцию периметр, так как есть линия, идущая вниз по середине фигуры. Попробуйте создать экземпляр этого класса в поле ниже и поэкспериментируйте с разными значениями. Поскольку класс Shape уже был запущен, вы можете просто добавить сюда только новые классы и добавить определение экземпляров.

```
class Square(Shape):
    def __init__(self,x):
        self.x = x
        self.y = x

class DoubleSquare(Square):
    def __init__(self,y):
        self.x = 2 * y
        self.y = y
    def perimeter(self):
        return 2 * self.x + 3 * self.y
testsquare = Square(5)
testdouble = DoubleSquare(6)
```

5.7.6 Указатели и словари классов

Когда вы указываете, что одна переменная равна другой, например variable2 = variable1, переменная слева от знака равенства принимает значение переменной справа. С экземплярами класса это происходит немного иначе - имя слева становится экземпляром класса справа. Таким образом, в

instance2 = instance1, instance2 указывает на instance1 - одному экземпляру класса дано два имени, и вы можете получить доступ к экземпляру класса через любое имя.

В других языках подобные действия выполняются с помощью * указателей *, однако в Python все это происходит за кулисами.

Последнее, что мы рассмотрим, - это словари классов. Мы можем назначить экземпляр класса записи в списке или словаре. Это позволяет существовать практически любому количеству экземпляров класса при запуске нашей программы. Давайте посмотрим на пример ниже:

```
# Сначала создайте словарь:
dictionary = {}

# Затем создайте несколько экземпляров классов в словаре:
dictionary["DoubleSquare 1"] = DoubleSquare(5)
dictionary["long rectangle"] = Shape(600,45)

# Теперь вы можете использовать их как обычный класс:
print(dictionary["long rectangle"].area())

dictionary["DoubleSquare 1"].authorName("Есенин")
print(dictionary["DoubleSquare 1"].author)
```

27000

Есенин

Используя словари работать с экземплярами классов удобнее. Особенно если вам потребуется обрабатывать их в циклах.

5.8 Модули

5.8.1 Введение

Если вы зададитесь вопросом «Как мне использовать мои классы в различных программах, просто постоянно копировать их туда?». Ответ - нет, необходимо поместить классы в модуль, чтобы его можно было использовать для импорта в другие программы. Часто модули называются библиотеками. Они так же позволяют скрывать свое внутреннее устройство и предоставлять только то, что доступно через свой интерфейс. Например, вы можете откомпилировать свой модуль и продавать его не как файл с кодом, а как .dll или .so библиотеку.

5.8.2 Модуль? Что такое модуль?

Модуль - это файл Python, который (как правило) содержит только определения переменных, функций и классов. Например, так может выглядеть модуль, который мы храним в файле moduletest.py:

```
### ПРИМЕР МОДУЛЯ PYTHON
# Определите некоторые переменные:
numberone = 1
ageofqueen = 78

# объявление некоторых функций
def printhello():
    print("привет")

def timesfour(input):
    print(eval(input) * 4)

# объявление класса
class Piano:
    def __init__(self):
        self.type = input("Какое пианино? ")
        self.height = input("Какая высота в метрах? ")
        self.price = input("Сколько оно стоит ")
        self.age = input("Сколько ему лет? ")

    def printdetails(self):
        print("Это пианино имеет высоту " + self.height + " метров", en
d=" ")
        print(self.type, "ему, " + self.age, "лет и его стоимость\
         " + self.price + " долларов.")
```

Как видите, модуль очень похож на вашу обычную программу Python.

Итак, что нам делать с модулем? Мы делаем import его частей или его полностью в другие программы.

Чтобы импортировать все переменные, функции и классы из moduletest.py в другую программу, которую вы пишете, мы используем оператор import. Например, чтобы импортировать moduletest.py в вашу основную программу (mainprogram.py), у вас будет следующее:

```
### mainprogam.py
### импорт другого модуля
import moduletest
```

Это предполагает, что модуль находится в том же каталоге, что и mainprogram.py, или является модулем по умолчанию, который поставляется

с Python. Указывать расширение .py при импорте не нужно. Обычно все операторы import помещают в начало файла Python для удобства, но технически они могут быть где угодно. Чтобы использовать элементы модуля в основной программе, вы используете следующее:

```
### ИСПОЛЬЗОВАНИЕ ИМПОРТИРОВАННОГО МОДУЛЯ
# Используйте код вида modulename.itemname
print(moduletest.ageofqueen)
cfcpiano = moduletest.Piano()
cfcpiano.printdetails()
```

Как видите, модули, которые вы импортируете, очень похожи на классы, которые мы рассматривали в прошлом уроке - все, что находится внутри них, должно начинаться с имени этого модуля, чтобы оно работало.

5.8.3 Пример ещё одного модуля "thingummyjigs"

Чтобы каждый раз при использовании элементов модуля не писать его имя можно использовать оператор from. Синтаксис следующий: from modulename import itemname. Вот пример:

```
###импорт элементов прямо в вашу программу

#их импорт
from moduletest import ageofqueen
from moduletest import printhello

#теперь используем их
print(ageofqueen)
printhello()
```

Это уберет множество вложенностей в вашем коде и сделает его более читабельным.

Если вы хотите, вы можете импортировать все из модуля, используя from modulename import *. Конечно, это может вызвать проблемы, если в вашей программе есть объекты с такими же именами, как у некоторых элементов в модуле. С большими модулями это может легко произойти и может вызвать много головной боли. Лучший способ сделать это - импортировать модуль обычным способом (без оператора from), а затем назначить элементам локальное имена:

```
### Присвоение элементам модуля локального имени
timesfour = moduletest.timesfour

# Использование локального имени
print(timesfour(565))
```

Последний удобный способ импорта модулей - использование псевдонима, для этого вы можете использовать оператор as (с англ.-как). Это выглядит так:

```
### ИМПОРТ МОДУЛЯ С псевдонимом
import moduletest as mt

# использование модуля
print(mt.age)
cfcpiano = mt.Piano()
cfcpiano.printdetails()
```

5.9 Файловый ввод-вывод

5.9.1 Открытие файла

Чтобы открыть текстовый файл, можно использовать функцию open (). Вы передаете функции open () определенные параметры, чтобы указать, каким образом следует открывать файл - r только для чтения, w только для записи (если есть старый файл, он будет перезаписан), а для добавления (добавления чего-либо в конец файла) и r +как для чтения, так и для записи.

Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
'x'	открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

Давайте откроем файл для чтения. Откройте обычный текстовый файл. Затем мы распечатаем то, что читаем внутри файла:

```
filename = 'readme.txt' # путь к файлу
fl = open(filename, 'r') # имя файла
for line in fl:
    print(line)
fl.close() #Закрыть файл
```

Лучше написать код следующим образом:

```
filename = 'readme.txt'
with open(filename, 'r') as fl:
    for line in fl:
        print(line)
```

При втором способе вам не нужно добавлять fl.close, он автоматически закрывается. Конструкция with ... as используется для оборачивания выполнения блока инструкций менеджером контекста. Иногда это более удобная конструкция, чем try...except...finally для обработки ошибок. Основная причина использования ContextManager во время открытия файла - заключается в том, что этот файл будет открыт в любом случае, независимо от того, все ли в порядке или возникает какое-либо исключение и гарантированно будет закрыт, а место, которое он занимал при открытии файла будет очищено.

5.9.2 Курсор при вводе-выводе

Если вы хотите распечатать весь файл, вместо того, чтобы перебирать строки, вы можете использовать это:

```
filename = './readme.txt'
fl = open(filename, 'r')
print(fl.read())
```

Если вы попытаетесь выполнить print (fl.read ()) второй раз, то произойдет ошибка, которая заключается в том, что «курсор» сменил свое место. Невидимый курсор сообщает функции чтения (и многим другим функциям ввода-вывода), с чего начать. Чтобы установить, где находится курсор, вы используете функцию seek (). Он используется в форме seek(offset, whence) (с англ. искать (смещение, откуда)).

whence (с англ. откуда) не является обязательным параметром и определяет, откуда искать. Если whence равен 0, байты / буквы считаются с начала. Если он равен 1, байты отсчитываются от текущей позиции курсора. Если это 2, то байты отсчитываются с конца файла. Если туда ничего не

помещено, предполагается, что 0, т.е. чтение будет производится сначала файла

«offset» (с англ. смещение) определяет, как далеко от «откуда» перемещается курсор. Например:

- fl.seek (45,0) переместит курсор на 45 байт / букв после начала файла.
- fl.seek (10,1) переместит курсор на 10 байт / букв после текущей позиции курсора.
- fl.seek (-77,2) переместит курсор на 77 байт / букв перед концом файла (обратите внимание на - перед 77)

Мы можем использовать fl.seek (), чтобы перейти к любому месту в файле, а затем попробовать ввести print (fl.read ()). Он будет печататься с того места, куда вы искали. Но помните, что fl.read () перемещает курсор в конец файла - вам придется выполнить поиск снова.

5.9.3 Другие функции ввода / вывода

Есть много других функций, которые помогут вам работать с файлами. Давайте посмотрим на некоторые из них: tell (), readline (), readlines (), write () и close ().

- tell () - возвращает курсор в файле. У него нет параметров, просто введите его (как в примере ниже). Это полезно для понимания того, где находится курсор. Чтобы использовать эту функцию, введите fileobjectname.tell () - где fileobjectname - это имя файлового объекта, созданного при открытии файла (в openfile = open ('pathtofile', 'r') имя файлового объекта openfile).
- readline () - читает с того места, где находится курсор, до конца строки. Помните, что конец строки - это не край экрана - строка заканчивается, когда вы нажимаете клавишу Enter, чтобы создать новую строку. Эта функция полезна, например, при чтении журнала событий. Единственный параметр, которые вы можете передаьь в readline () при

необходимости - это максимальное количество байтов / букв для чтения, поместив соответсвующее число в скобки.

- readlines () - считывает все строки, начиная с курсора и до конца, и возвращает список, в котором каждый элемент списка содержит строку кода. Используйте его с fileobjectname.readlines (). Например, если у вас есть текстовый файл:

Линия 1

Строка 3

Строка 4

Строка 6

тогда список, возвращенный из readlines (), будет выглядеть следующим образом:

Index	Value
0	'Line 1'
1	''
2	'Line 3'
3	'Line 4'
4	''
5	'Line 6'

- write () - функция записи в файл. Она пишет с того места, где находится курсор, и перезаписывает текст перед ним - как в MS Word, где вы нажимаете «вставить», и он записывает поверх старого текста. Синтаксис следующий fileobjectname.write ('Строка, которую вы хотите записать').
- close - закрывает файл.

Попробуйте поэкспериментировать с созданием и чтением файла через Python. Создать файл вы можете в стороннем редакторе, или сразу начать работать с функцией записи.

```
#просто создадим файл и запишем в него пару строк
filename = './readme.txt'
fl = open(filename, 'w')
fl.write('Привет\n')
fl.write('123')
fl.close() #Закрыть файл

#прочитаем содержимое файла
with open(filename, 'r') as fl:
    for line in fl:
        print(line)
```

Привет

123

5.9.4 Объекты, сохраняемые в файл

Вы можете сохранять объекты напрямую в файл. Объект в этом случае может быть переменной, экземпляром класса или списком, словарем или кортежем. Можно делать dump (с англ. - сброс, т.е. сброс данных) и другие вещи. Другими словами, вы «сохраняете» свои объекты.

Для сохранения объектов вы можете воспользоватся функцией dump (), которая находится внутри модуля pickle (с англ. мариновать) - поэтому в начале вашей программы вам нужно будет написать import pickle.

После импорта откроем пустой файл и используем функцию pickle.dump (), чтобы поместить объект в этот файл:

```
### pickletest.py

import pickle

# давайте создадим список для "маринования"

picklelist = ['один',2,'три','четыре',5,'Можете сосчитать?']

# теперь создаем файл
# вы можете заменить имя файла на любое
# wb означает, что файл записан в двоичном виде (т.е. не .txt и не .docx)
file = open('filename', 'wb')

#"замаринуем рассол"
pickle.dump(picklelist,file)

#закроем файл
file.close()
```

Обновите список файлов и убедитесь. что файл с именем filename был создан.

- object_to_pickle - это объект, который вы хотите сохранить в файл
- file_object - это файловый объект, в который вы хотите осуществить запись (в данном случае файловым объектом является file)

Теперь, чтобы повторно открыть или разобрать ваш файл, мы будем использовать pickle.load ():

```
import pickle

# теперь открываем файл для чтения
unpicklefile = open('filename', 'rb')

# теперь загружаем список из файла
unpickledlist = pickle.load(unpicklefile)

#закроем файл
unpicklefile.close()

# Попробуем напечатать список и проверить, что все ок
for item in unpickledlist:
    print(item)
```

один

2

три

четыре

5

Можете сосчитать?

Есть важное ограничение: таким образом мы можем поместить в файл только один объект. Мы могли бы обойти это, поместив множество выбираемых объектов в список или словарь, а затем при чтении обработав этот список или словарь. Это самый быстрый и простой способ, более сложные способы вы можете изучить самостоятельно.

5.10 Обработка исключений (ошибок)

5.10.1 Введение

При помощи программы ниже продемонстрируем возникновение исключения в программе:

```
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print(") " + entry)

    return input(question) - 1

# вызов функции меню
answer = menu(['A','B','C','D','E','F','H','I'],\
'Какая буква ваша любимая ')

print('Вы выбрали ответ ' + str(answer + 1))
```

1) A

2) B

3) C

4) D

5) E

6) F

7) H

8) I

Какая буква ваша любимая 1

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-82d612b55785> in <module>
      7
      8 # вызов функции меню
----> 9 answer = menu(['A','B','C','D','E','F','H','I'],\
     10 'Какая буква ваша любимая ')
     11

<ipython-input-36-82d612b55785> in menu(list, question)
      4         print(") " + entry)
      5
----> 6     return input(question) - 1
      7
      8 # вызов функции меню

TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

5.10.2 Ошибки - человеческие ошибки

Наиболее частые проблемы с вашим кодом возникают по вашей собственной вине. Печально, но факт. Что мы видим, когда пытаемся запустить нашу урезанную программу?

```
-----------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-2-1502b4586513> in <module>()
      8 # running the function
      9 # remember what the backslash does
---> 10 answer = menu(['A','B','C','D','E','F','H','I'],'Какая буква ва
ша любимая? ')
     11
     12 print('Вы выбрали ответ ' + str(answer + 1))

<ipython-input-2-1502b4586513> in menu(list, question)
      4         print(") " + entry)
      5
----> 6     return input(question) - 1
      7
      8 # running the function

TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Python пытается сказать вам, что вы не можете объединить строку букв и число в одну строку текста. Давайте рассмотрим сообщение об ошибке и посмотрим, как оно сообщает нам, что:

- ---> показывает строки, в которых обнаружена ошибка. Сначала он указывает на строку 10 (строка), а затем на строку 6 (вычисление, при котором мы вычитаем целое число из строки). Обратите внимание, что строка 6 была в функции.
- TypeError: неподдерживаемые типы операндов для -: 'str' и 'int' ' - ошибка типов, когда вы попытались вычесть друг из друга несовместимые переменные.

Существует несколько списков файлов (в данном случае- функций) и кодов для одной ошибки, потому что ошибка возникла при взаимодействии двух строк кода (например, при использовании функции ошибка возникла в строке, где функция была вызвана, И строке в функции, где все пошло не так).

Теперь, когда мы знаем, в чем проблема, и как ее исправить. В сообщении об ошибке указано, где находится проблема, поэтому мы сосредоточимся только на этом фрагменте кода.

```
answer = menu (['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I'], \
'Какая буква ваша любимая? ')
```

Это вызов функции. Ошибка произошла в функции в следующей строке:

```
    return input(question) - 1
```

input всегда возвращает строку, отсюда и наша проблема. Давайте заменим input () на функцию eval (input ()).

Функция eval позволяет программе Python запускать код Python внутри себя.

Функция "меню" по завершению своего выполнения возвращает выбранное пользователем число минус 1, чтобы индексы были с нуля, а не как человек привык.

```
return eval(input(question)) - 1
```

Ошибка в программе исправлена!

```
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print(") " + entry)

    return eval(input(question)) - 1

# вызов функции меню
answer = menu(['A','B','C','D','E','F','H','I'],\
'Какая буква ваша любимая ')

print('Вы выбрали ответ ' + str(answer + 1))
```

1) A

2) B

3) C

4) D

5) E

6) F

7) H

8) I

Какая буква ваша любимая 1

Вы выбрали ответ 1

5.10.3 Исключения

Все работает правильно, если вводить число. А если пользователь случайно введет букву вместо числа появится возникнет ошибка

Если не знаете, как это предотвратить возникновение ошибки, то один из лучших и простых способов - использовать операторы try и except для обработки исключений.

Пример использования try в программе:

```
try:
    function(world,parameters)
except:
    print(world.errormsg)
```

Сначала запускается код под try:. Если возникает ошибка, интерпретатор переходит в раздел except и печатает код ошибки world.errormsg. Программа не останавливается и не дает сбой, т.е. при

возникновении ошибки она просто запускает код под except:, а затем продолжает работу.

Давайте попробуем обработать подобные ошибки. Попробуйте ввести букву, когда вас просят ввести число, и посмотрите, что произойдет. Мы исправили одну проблему, но теперь она вызвала другую проблему:

```
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print(") " + entry)
    try:
        return eval(input(question)) - 1
    except NameError:
        print("Введите правильное число")

# running the function
# remember what the backslash does
answer = menu(['A','B','C','D','E','F','H','I'],\
'Какая буква ваша любимая ')

print('Вы выбрали ответ ' + str(answer + 1))
```

1) A

2) B

3) C

4) D

5) E

6) F

7) H

8) I

Какая буква ваша любимая A

Введите правильное число

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-9f17d20db70c> in <module>
     13 'Какая буква ваша любимая ')
     14
---> 15 print('Вы выбрали ответ ' + str(answer + 1))

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

На этот раз произошло то, что функция меню не вернула никакого значения - она только напечатала сообщение об ошибке. Когда в конце

программы мы пытаемся напечатать возвращаемое значение плюс 1, какое значение будет возвращено? Нет возвращаемого значения? Итак, что такое 1 + ... -эти некорректное выражение!

Мы могли просто вернуть любой старый номер, но это было бы не правильно. Что нам действительно нужно, так это переписать программу, чтобы справиться с этим исключением.

```
def menu(list, question):
    for entry in list:
        print(1 + list.index(entry),end="")
        print(") " + entry)
    try:
        return eval(input(question)) - 1
    except NameError:
        print("Введите правильное число")

answer = menu(['A','B','C','D','E','F','H','I'],\
'Какая буква ваша любимая? ')
try:
    print("Вы выбрали ответ ", (answer + 1))
    # вы можете ставить что-либо после запятой в операторе 'print',
     # и это будет продолжаться, как если бы вы снова набрали 'print'
except:
    print("\nНеправильный ответ")
    # '\ n' используется для форматирования, а именно - это сибол переноса строк
```

1) A

2) B

3) C

4) D

5) E

6) F

7) H

8) I

Какая буква ваша любимая? M

Введите правильное число

Неправильный ответ

Проблема снова решена.

5.10.4 Бесконечные ошибки

Подход, который мы использовали выше, не рекомендуется, потому что кроме ошибки, которая, как мы знаем, может произойти, except: также перехватывает все остальные ошибки. Что, если это означает, что мы никогда не увидим ошибку, которая могла бы вызвать проблемы в будущем? Если except: отлавливает каждую скрытую ошибку, у нас нет никакого способа контролировать, с какими ошибками мы имеем дело, а также другие, которые мы хотим видеть, потому что до сих пор мы не работали с ними. У нас также мало шансов на устранение более чем одного типа ошибок в одном и том же блоке кода. Что делать, когда все безнадежно? Вот пример кода с такой ситуацией:

```
print("Программа вычитания, v0.0.1 (бета)")
a = eval(input('Введите 1ое число для вычитания > '))
b = eval(input('Введите 2ое число > '))
print(a - b)
```

Программа вычитания, v0.0.1 (бета)

Введите 1ое число для вычитания > M

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-41-95bfdd9a742a> in <module>
      1 print("Программа вычитания, v0.0.1 (бета)")
----> 2 a = eval(input('Введите 1ое число для вычитания > '))
      3 b = eval(input('Введите 2ое число > '))
      4 print(a - b)

<string> in <module>

NameError: name 'M' is not defined
```

Вы вводите свои два числа, и все работает. Введите букву, и она выдаст вам NameError. Давайте перепишем код, чтобы иметь дело только с ошибкой NameError. Мы поместим программу в цикл, чтобы она перезапускалась в случае возникновения ошибки (с помощью оператора continue (с англ. продолжить), который снова запускает цикл сверху, и break, который выходит из цикла):

```
print("Программа вычитания, v0.0.2 (бета)")
loop = 1
while loop == 1:
    try:
        a = eval(input('Введите 1ое число для вычитания > '))
        b = eval(input('Введите 2ое число > '))
    except NameError:
        print("\nВы не можете вычитать буквы")
        continue
    print(a - b)
    try:
        loop = eval(input('Нажмите 1, чтобы попробовать снова> '))
    except NameError:
        loop = 0
```

Программа вычитания, v0.0.2 (бета)

Введите 1ое число для вычитания > 2

Введите 2ое число > 2

0

Нажмите 1, чтобы попробовать снова> !

```
Traceback (most recent call last):

  File "/home/pmelikov/.local/lib/python3.8/site-packages/IPython/core/interact
iveshell.py", line 3441, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-42-207f142451a1>", line 12, in <module>
    loop = eval(input('Нажмите 1, чтобы попробовать снова> '))

  File "<string>", line 1
    !
    ^
SyntaxError: unexpected EOF while parsing
```

Мы перезапустили цикл, если пользователь ввел что-то не так. В строке 12 мы предположили, что вы хотите выйти из программы, если не нажали 1, поэтому выходим из программы.

Но проблемы остались. Если мы оставим что-то пустым или введем специальный символ, например ! или ;, программа выдает нам SyntaxError. Давайте с этим разберемся. Когда мы просим вычесть числа, мы выдаем другое сообщение об ошибке. Когда мы просим нажать 1, мы снова будем предполагать, что пользователь хочет выйти.

```
print("Subtraction program, v0.0.2 (beta)")
loop = 1
while loop == 1:
    try:
        a = eval(input('Введите 1ое число для вычитания > '))
        b = eval(input('Введите 2ое число > '))
    except NameError:
        print("\nYou cannot subtract a letter")
        continue
    except SyntaxError:
        print("\nВы не можете вычитать буквы")
        continue
    print(a - b)
    try:
        loop = eval(input('Нажмите 1, чтобы попробовать снова> '))
    except (NameError,SyntaxError):
        loop = 0
```

Subtraction program, v0.0.2 (beta)

Введите 1ое число для вычитания > !

Вы не можете вычитать буквы

Введите 1ое число для вычитания > 2

Введите 2ое число > 43

-41

Нажмите 1, чтобы попробовать снова> 0

Введите код

5.11 Стандартные библиотеки Python

5.11.1 Интерфейс операционной системы

Модуль os предоставляет большое количество функций для взаимодействия с операционной системой:

```
import os

print (os.getcwd())       # Вернуть (получить) текущий рабочий каталог
os.system('mkdir today')  # Запускаем команду mkdir в системной оболочке
os.chdir('./today')   # Изменить текущий рабочий каталог (перейти в созданную па
print (os.getcwd())

os.chdir('../')
os.rmdir('today') #удалить созданную папку
print (os.getcwd())
```

/home/pmelikov/Python для анализа данных. Семинар 3.

/home/pmelikov/Python для анализа данных. Семинар 3./today
/home/pmelikov/Python для анализа данных. Семинар 3.

Обязательно используйте import os вместо from os import *. Это не позволит функции os.open () заменить встроенную функцию open (), которая работает по-другому.

Вы можете воспользоваться функциями dir () и help (), передав os в качестве параметра, чтобы получить список доступных методов этой библиотеки и справку по ним. Например:

```
import os

dir(os)
help(os)
```

Для повседневных задач управления файлами и каталогами модуль shutil предоставляет более простой в использовании интерфейс более высокого уровня:

```
import shutil

#копирование файла
shutil.copyfile('Python для анализа данных. Семинар 1..ipynb', 'copy_of_
lesson.ipynb')
#перемещение файла
shutil.move('/build/executables', 'installdir')
```

5.11.2 Поиск файлов на основе подстановочных данных (Wildcards)

Модуль glob находит все пути, совпадающие с заданным шаблоном в соответствии с правилами, используемыми системной оболочкой операционной системы.

Обрабатываются символы:

- "*" (произвольное количество символов),
- "?" (один символ), и диапазоны символов с помощью [].

Для использования тильды "~" и переменных окружения необходимо использовать os.path.expanduser() и os.path.expandvars().

Для поиска спецсимволов, заключайте их в квадратные скобки. Например, [?] соответствует символу "?".

- glob.glob(pathname) возвращение список (возможно, пустой) путей, соответствующих шаблону pathname. Путь может быть как абсолютным (например, /usr/src/Python-1.5/Makefile) или относительный (как ../../Tools//.gif).
- glob.escape(pathname) - экранирует все специальные символы для glob ("?", "*" и "["). Специальные символы в имени диска не экранируются (так как они там не учитываются), то есть в Windows escape("//?/c:/Quo vadis?.txt") возвращает "//?/c:/Quo vadis[?].txt".

Указанный ниже код выведет имена всех файлов в текущей папке, содержащие *.ipynb. Звездочка означает подстановку любого количества любых знаков.

```
import glob

glob.glob('*.ipynb')
#или так, если нам нужно именно занятие от 0 до 9.
# ? - соответствует одному любому символу, в данном случае - первой точке
glob.glob('*[0-9]?.ipynb')
```

['Python для анализа данных. Семинар 3..ipynb']

5.11.3 Аргументы командной строки

Когда вы делаете какие-либо сценарии, когда при запуске кода вам нужно передать ему какие то параметры, то возникает необходимость в обработке аргументов командной строки. Эти аргументы хранятся в атрибуте argv модуля sys в виде списка.

Рассмотрим пример. Здесь код запущен через вызов системной команды вызова интерпретатора, о чем говорит восклицательный знак. Т.е. здесь не Jupyter выполняет python код, а системный интерпретатор. При этом код передаем просто как аргумент.

Если бы наш код хранился в каком то файле, то мы бы написали python demo.py one two three, например.

Вы могли, например использовать один и тот же скрипт, но в разных целях. Например, если передан параметр one - делать одно, а если другой -то другое действие, или все сразу. Или просто передавать какое-либо значение, например путь к датасету.

```
!python3 -c 'import sys; print(sys.argv)' one two three
```

['-c', 'one', 'two', 'three']

Модуль argparse предоставляет более сложный механизм для обработки аргументов командной строки. Следующий код извлекает одно или несколько имен файлов и необязательное количество отображаемых строк:

```
command= """import argparse
parser = argparse.ArgumentParser(prog = "top_lines", description = "Показать вер
parser.add_argument("filenames", nargs="+") #добавление аргументов к программе,
parser.add_argument("-l", "--lines", type=int, default=10) #добавление аргумента
args = parser.parse_args() #запись прочитанных аргументов в массив
print(args) """

!python3 -c '{command}' "Python для анализа данных. Семинар 1..ipynb" #запуск пр
!echo "----"
!python3 -c '{command}' -h
```

Namespace(filenames=['Python для анализа данных. Семинар 1..ipynb'], lines=10)

----

usage: top_lines [-h] [-l LINES] filenames [filenames ...]

Показать верхние строки из каждого файла

positional arguments:

  filenames

optional arguments:

  -h, --help   show this help message and exit

  -l LINES, --lines LINES

Ниже представлены три основных аргумента для argparse.ArgumentParser

- prog - Название программы (по умолчанию: sys.argv [0])
- description - Текст, отображаемый перед справкой по аргументу (по умолчанию: None)
- add_help - Добавить -h/--help параметр парсера (по умолчанию: True)

Если бы вы создали данный код отдельным python файлом, то из системной оболочки (командной строки или терминала вашей ОС), например командой python3 top.py --lines = 5 alpha.txt beta.txt сценарий устанавливает args.lines равным 5, а args.filenames - ['alpha.txt', 'beta.txt'].

5.11.4 Соответствие строковому шаблону (регулярные выражения)

Модуль re предоставляет инструменты регулярных выражений для расширенной обработки строк. Для сложных сопоставлений и манипуляций регулярные выражения предлагают краткие, оптимизированные решения:

```
import re
print (re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')) #найдем любы
```

['foot', 'fell', 'fastest']

Когда вам нужно найти символ в строке, в большей части случаев вы можете просто использовать этот символ или строку. Так что, когда нам нужно проверить наличие слова «dog», то мы будем использовать буквы в dog. Конечно, существуют определенные символы, которые заняты регулярными выражениями. Они так же известны как метасимволы. Внизу изложен полный список метасимволов, которые поддерживают регулярные выражения Python:

. ^ $ * + ? { } [ ] | ( )

Давайте взглянем как они работают. Основная связка метасимволов, с которой вы будете сталкиваться, это квадратные скобки: [и]. Они используются для создания «класса символов», который является набором символов, которые вы можете сопоставить. Вы можете отсортировать символы индивидуально, например, так: [xyz]. Это сопоставит любой внесенный в скобки символ. Вы также можете использовать тире для выражения ряда символов, соответственно: [a-g]. В этом примере мы сопоставим одну из букв в ряде между a и g. Фактически для выполнения поиска нам нужно добавить начальный искомый символ и конечный. Чтобы упростить это, мы можем использовать звездочку. Вместо сопоставления *, данный символ указывает регулярному выражению, что предыдущий символ может быть сопоставлен 0 или более раз. Давайте посмотрим на пример, чтобы лучше понять о чем речь:

a[b-f]*f

Этот шаблон регулярного выражения показывает, что мы ищем букву a, ноль или несколько букв из нашего класса, [b-f] и поиск должен закончиться на f. Давайте используем это выражение в Python:

```
import re
text = 'abcdfghijk'

parser = re.search('a[b-f]*f', text)
print(parser.group()) # 'abcdf'
```

abcdf

Это выражение просмотрит всю переданную ей строку, в данном случае это abcdfghijk. Выражение найдет нашу букву a в начале поиска. Затем, в связи с тем, что она имеет класс символа со звездочкой в конце, выражение прочитает остальную часть строки, что бы посмотреть, сопоставима ли она. Если нет, то выражение будет пропускать по одному

символу, пытаясь найти совпадения. Вся магия начинается, когда мы вызываем поисковую функцию модуля re. Если мы не найдем совпадение, тогда мы получим None. В противном случае, мы получим объект Match. Чтобы увидеть, как выглядит совпадение, вам нужно вызывать метод group().

Метод / Атрибут	Цель
group()	Вернуть строку, совпадающую с RE
start ()	Вернуть начальную позицию матча
end ()	Вернуть конечную позицию матча
span ()	Вернуть кортеж, содержащий (начало, конец) позиции матча (span - промежуток с англ.)
match()	Определите, совпадает ли RE в начале строки.
search ()	Просматривайте строку в поисках любого место, где соответствует этот RE.
findall ()	Найдите все подстроки, которым соответствует RE, и возвращает их в виде списка.
finditer ()	Найдите все подстроки, которым соответствует RE, и возвращает их как итератор.

Существует еще один повторяемый метасимвол, аналогичный *. Этот символ +, который будет сопоставлять один или более раз. Разница с *, который сопоставляет от нуля до более раз незначительна, на первый взгляд.

Символу + необходимо как минимум одно вхождение искомого символа. Последние два повторяемых метасимвола работают несколько иначе. Рассмотрим знак вопроса ?, применение которого выглядит так: со-?ор. Он будет сопоставлять и соор и со-ор. Последний повторяемый метасимвол это {a,b}, где a и b являются десятичными целыми числами. Это значит, что должно быть не менее a повторений, но и не более b. Ниже представлен пример:

```
import re
text = 'xbbbbz  xz xz xbbz'

parser = re.findall('xb{1,4}z', text)
print(parser) # 'abcdf'
```

['xbbbbz', 'xbbz']

Это очень примитивный пример, но в нем говорится, что мы сопоставим следующие комбинации: xz, xbbz, xbbbz и xbbbbz, но не xz, так как он не содержит «b».

Следующий метасимвол это ^. Этот символ позволяет нам сопоставить символы которые не находятся в списке нашего класса. Другими словами, он будет дополнять наш класс. Это сработает только в том случае, если мы разместим ^ внутри нашего класса. Если этот символ находится вне класса, тогда мы попытаемся найти совпадения с данным символом. Наглядным примером будет следующий: [ˆa]. Это выражение будет искать совпадения с любой буквой, кроме «а». Символ ^ также используется как якорь, который обычно используется для совпадений в начале строки.

Подробнее про регулярные выражения и их другие метасимволы можно почитать тут: https://python-scripts.com/import-re-regular-expression

```
import re
text = 'anton oleg aanton'

parser = re.findall('[^a]* ', text)
print(parser) # 'abcdf'
```

['nton oleg ']

Когда вам необходимы только простые операции, предпочтительны строковые методы, потому что их легче читать и отлаживать. Например замена слов при помощи строкового метода .replace():

```
'tea for too'.replace('too', 'two')
```

'tea for two'

5.11.5 Математика, случайные числа и статистика

Математический модуль предоставляет доступ к базовым функциям библиотеки С для математики с плавающей запятой, реализованным в откомпилированном при помощи языка С модуле Python. (Откомпилированные модули Python обладают наибольшей доступной производительностью. Лучше только асемблерный код или машинный.)

Функция	Описание
ceil (x)	Возвращает наименьшее целое число, большее или равное x.
copysign (x, y)	Возвращает x со знаком y
fabs (x)	Возвращает абсолютное значение x
factorial (x)	Возвращает факториал x
floor (x)	Возвращает наибольшее целое число, меньшее или равное x
fmod (x, y)	Возвращает остаток от деления x на y
frexp (x)	Возвращает мантиссу и показатель степени x как пару (m, e)
fsum (итерация)	Возвращает точную сумму значений с плавающей запятой в итерируемом
isfinite (x)	Возвращает True, если x не является ни бесконечностью, ни NaN (не числом)
isinf (x)	Возвращает True, если x - положительная или отрицательная бесконечность
isnan (x)	Возвращает True, если x - NaN
ldexp (x, i)	Возвращает x * (2 ** i)
modf (x)	Возвращает дробную и целую части x
trunc (x)	Возвращает усеченное целочисленное значение x
exp (x)	Возврат e ** x
expm1 (x)	Возвращает e ** x - 1
log (x [, b])	Возвращает логарифм x по основанию b (по умолчанию e)
log1p (x)	Возвращает натуральный логарифм 1 + x
log2 (x)	Возвращает логарифм x по основанию 2
log10 (x)	Возвращает десятичный логарифм числа x
pow (x, y)	Возвращает x в степени y
sqrt (x)	Возвращает квадратный корень из x
acos (x)	Возвращает арккосинус x
asin (x)	Возвращает арксинус x
атан (x)	Возвращает арктангенс x
atan2 (y, x)	Возвращает атан (г / x)
cos (x)	Возвращает косинус x
hypot (x, y)	Возвращает евклидову норму sqrt (x * x + y * y)
sin (x)	Возвращает синус x
tan (x)	Возвращает тангенс x
degrees (x)	Преобразует угол x из радианов в градусы
radians (x)	Преобразует угол x из градусов в радианы
acosh (x)	Возвращает обратный гиперболический косинус x
asinh (x)	Возвращает гиперболический синус, обратный x
atanh (x)	Возвращает обратный гиперболический тангенс x
cosh (x)	Возвращает гиперболический косинус x
sinh (x)	Возвращает гиперболический косинус x
tanh (x)	Возвращает гиперболический тангенс x
erf (x)	Возвращает функцию ошибок в x
erfc (x)	Возвращает дополнительную функцию ошибок в x
gamma (x)	Возвращает гамма-функцию в точке x
lgamma (x)	Возвращает натуральный логарифм абсолютного значения гамма-функции в точке x
pi Математическая константа, отношение длины окружности к ее диаметру (3,14159 ...)	
e	математическая константа e (2,71828 ...)

```
import math
print (math.cos(math.pi / 4))
print(math.log(1024, 2))
```

0.7071067811865476

10.0

Модуль random предоставляет инструменты для случайного выбора и генерации случайных чисел:

```
import random

print(random.choice(['apple', 'pear', 'banana'])) #случайный выбор из списка
print (random.sample(range(100), 10)) # генерация случайного набора значений
print (random.random())    # случайное с плавающей точкой float
print (random.randrange(6))    # случайное значение integer выбранное из диапазо
```

apple

[65, 79, 15, 97, 3, 5, 49, 98, 46, 25]

0.46503625848606844

0

Модуль статистики вычисляет основные статистические свойства (среднее значение, медиана, дисперсия и т. д.) числовых данных. В проекте SciPy https://scipy.org есть много других модулей для численных вычислений.

```
import statistics

data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
print (statistics.mean(data)) #среднее
print (statistics.median(data)) # медиана
print (statistics.variance(data)) #дисперсия
```

1.6071428571428572

1.25

1.3720238095238095

5.11.6 Доступ в Интернет

Существует ряд модулей для доступа в Интернет и обработки интернет-протоколов. Два из самых простых - urllib.request для получения данных из URL-адресов и smtplib для отправки почты.

Например, в коде ниже скачивается код страницы 'https://stankin.ru/sys/img/logo_ru.svg'и отображается в блокноте.

```
from urllib.request import urlopen
with urlopen('https://stankin.ru/sys/img/logo_ru.svg') as response:
    html = response.read()
    #for line in response:
    #    print(line)


from IPython.display import HTML, display
#отрисовка html кода
display (HTML(html.decode("utf-8")))
#показать кусочек скачанного html кода
display (html[:60])
```

b'<svg width="567" height="318" viewBox="0 0 567 318" fill="no'

5.11.7 Даты и время

Модуль datetime предоставляет классы для управления датами и временем как простыми, так и сложными способами. Хотя арифметика даты и времени поддерживается, основное внимание в реализации уделяется эффективному извлечению элементов для форматирования и обработки вывода. Модуль также поддерживает объекты с учетом часового пояса.

```
# даты легко конструируются и форматируются
from datetime import date

now = date.today()
print (now)
print (date(2003, 12, 2))

print (now.strftime("%m-%d-%y. %d %b %Y это %A %d день %B."
'12-02-03. 02 Dec 2003 это Вторник или 02 день декабря.'))

# даты поддерживают календарную арифметику
birthday = date(1964, 7, 31)
age = now - birthday
print ("Вам ", age.days," дней или", int(age.days / 365.25), " лет")
```

2021-07-28

2003-12-02

07-28-21. 28 Jul 2021 это Wednesday 28 день July.12-02-03. 02 Dec 2003 это Вторник или 02 день декабря.

Вам 20816 дней или 56 лет

5.11.8 Сжатие данных

Стандартные форматы архивирования и сжатия данных напрямую поддерживаются модулями, включая: zlib, gzip, bz2, lzma, zipfile и tarfile.

```
import zlib

s = bytes("Пой же пой на проклятой гитаре", 'utf-8')
print("Размер до сжатия: ",len(s))
crc1= zlib.crc32(s)
print ("Контрольная сумма до сжатия: ", crc1)

t = zlib.compress(s)
print ("Размер после сжатия: ",len(t))

t=zlib.decompress(t)
crc2= zlib.crc32(t)
print ("Контрольная сумма после распаковки: ", crc2)
if crc1==crc2:
    print ('\033[92m'+"Контрольная суммы равны!"+'\033[0m')
```

Размер до сжатия: 55

Контрольная сумма до сжатия: 1530206883

Размер после сжатия: 53

Контрольная сумма после распаковки: 1530206883

Контрольная суммы равны!

5.11.9 Измерение производительности

Некоторые пользователи Python проявляют глубокий интерес к знанию относительной производительности различных подходов к одной и той же проблеме. Python предоставляет инструмент измерения, который сразу же отвечает на эти вопросы.

Например, может возникнуть соблазн использовать функцию упаковки и распаковки кортежей вместо традиционного подхода к обмену аргументами. Модуль timeit быстро демонстрирует скромное преимущество в производительности:

- первый параметр - функция, которую вы хотите протестировать
- второй параметр - начальная инициализация значений для теста функции
- сколько раз тестировать

```
#1 вариант измерения производительности
import timeit
iterations=5 # сколько раз тестировать быстродействие
usec=timeit.timeit('a=a**b; print ("Я посчитал:",a)', setup='a=5; b=2', number=
print("суммарно за", usec, "наносекунд")
```

Я посчитал: 25

Я посчитал: 625

Я посчитал: 390625

Я посчитал: 152587890625

Я посчитал: 23283064365386962890625

суммарно за 0.00030600797617807984 наносекунд

Второй подход - измерение времени через таймер:

```
#2 вариант измерения производительности
import time

start_time = time.time()

for i in range(5):
    a=5; b=2
    a=a**b; print ("Я посчитал:",a)

print("суммарно за %s наносекунд ---" % (time.time() - start_time))
```

Я посчитал: 25

Я посчитал: 25

Я посчитал: 25

Я посчитал: 25

Я посчитал: 25

суммарно за 0.0016069412231445312 наносекунд ---

5.11.10 Контроль качества

Один из подходов к разработке высококачественного программного обеспечения - это писать тесты для каждой функции по мере ее разработки и часто запускать эти тесты в процессе разработки.

Модуль doctest предоставляет инструмент для сканирования модуля и проверки тестов, встроенных в строки документации программы. Создание теста так же просто, как вырезание и вставка типичного вызова вместе с его результатами в строку документации. Это улучшает документацию, предоставляя пользователю пример и позволяет модулю doctest убедиться, что код соответствует документации:

```
def average(values):
    """Вычисляет среднее арифметическое списка чисел.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # автоматически проверять встроенные тесты
```

TestResults(failed=0, attempted=1)

Первая строчка в """ - это комментарий, который будет отображатся, когда вы наводите мышкой на функцию average() в среде разработки, например PyCharm.

Модуль unittest не так прост, как модуль doctest, но он позволяет хранить более полный набор тестов в отдельном файле:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self): #тест функции upper() (преобразует регистр в верхний)
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper()) #проверка что все в верхнем регистре
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # проверяем, что s.split не работает, если разделитель не является строк
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

5.11.11 Многопоточность

Многопоточность - это метод разделения задач, которые не зависят друг от друга. Потоки можно использовать для улучшения реакции приложений, которые принимают ввод пользователя, в то время как другие задачи выполняются в фоновом режиме. Связанный вариант использования - запуск ввода-вывода параллельно с вычислениями в другом потоке.

В следующем коде показано, как модуль потоковой передачи более высокого уровня может выполнять задачи в фоновом режиме, пока основная программа продолжает выполняться:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Готова фоновая zip запаковка файла:', self.infile)

f = open("mydata.txt", "a")
f.write("Теперь в файле больше содержимого!")
f.close()

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('Основная программа продолжает работать на переднем плане.')

background.join()    # Дождитесь завершения фоновой задачи
print('Основная программа дождалась завершения фоновой обработки.')
```

Основная программа продолжает работать на переднем плане. Готова фоновая zip запаковка файла: mydata.txt

Основная программа дождалась завершения фоновой обработки.

Основная проблема многопоточных приложений - это координация потоков, которые совместно используют данные или другие ресурсы. С этой целью модуль потоковой передачи предоставляет ряд примитивов синхронизации, включая блокировки, события, переменные состояния и семафоры.

Несмотря на то, что эти инструменты являются мощными, незначительные ошибки проектирования могут привести к проблемам, которые трудно воспроизвести. Таким образом, предпочтительный подход к координации задач состоит в том, чтобы сконцентрировать весь доступ к ресурсу в одном потоке, а затем использовать модуль очереди для подачи в этот поток запросов из других потоков. Приложения, использующие объекты Queue для межпоточного взаимодействия и координации, проще разрабатывать, они более читабельны и надежны.

Использование библиотеки multiprocessing более предпочтительно, т.к. с ней интерпретатор и программа смогут занимать больше чем 1 логическое ядро процессора. Да, threading тоже дает многопоточность, но в том и заключается отличие, что мы не можем загружать больше одного логического ядра процессора. Пример: умножение чисел в массиве на 2. Каждый экземпляр функции выполняется отдельным процессом с уникальным ID. Т.е. за счет многопоточности мы в итоге могли выполнить данную программу ровно в 5 раз быстрее, потому что элементов в массиве 5 и было бы выполнено 5 параллельных умножений, если на компьютере доступно 5 ядер процессора (игнорируя тот факт, что современные процессоры имеют умный кеш и гипервизор задач, и на самом деле умножат все одним ядром).

```
import time
start_time = time.time()

import os
from multiprocessing import Process

def doubler(number):
    """
    Функция умножитель на два
    """
    result = number * 2
    proc = os.getpid()
    print('{0} doubled to {1} by process id: {2}\n'.format(
        number, result, proc))

if __name__ == '__main__':
    numbers = [5, 10, 15, 20, 25]
    procs = []

    for index, number in enumerate(numbers):
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    for proc in procs:
        proc.join()

print("--- %s наносекунд заняло наше умножение.---" % (time.time() - start_time)
print ("Так долго - потому что накладные расходы большие. Умножение работает быс
```

5 doubled to 10 by process id: 71169

10 doubled to 20 by process id: 71170

20 doubled to 40 by process id: 71176

15 doubled to 30 by process id: 71175

25 doubled to 50 by process id: 71181

--- 0.09059286117553711 наносекунд заняло наше умножение.---

Так долго - потому что накладные расходы большие. Умножение работает быстро, а вот создание потоков и процессов-нет.

5.12 TensorFlow

TensorFlow —библиотека для машинного обучения с открытым исходным кодом.

Была создана компанией Google для решения задач, связанных с проектированием, построением, и тренировкой нейронных сетей с целью автоматического детектирования и классификации образов. Активно используется как для исследований, так и для разработки внутренних продуктов самой компании.

Библиотека написана на языках Python, C++ и CUDA. Последний делает возможным выполнение вычислений общего назначения не только на CPU, но и на GPU, опираясь на программно-аппаратную архитектуру CUDA графических процессоров компании Nvidia.

Основной API для работы с библиотекой реализован для Python, однако существуют реализации для C++, Java, Go и нескольких других языков программирования.

Вычисления TensorFlow производятся с помощью data-flow графов. Это означает, что в таких графах вершинами являются математические операции, а рёбрами являются данные, которые обычно представляются в виде многомерных массивов или *тензоров*, которые же и сообщаются между

этими рёбрами. Поэтому данная библиотека и получило название TensorFlow, что дословно — поток тензоров.

5.12.1 Начало работы

TensorFlow – огромная библиотека. Далее будут представлены основные понятия и главные функциональные возможности.

Больше подробностей можно найти в официальной документации:

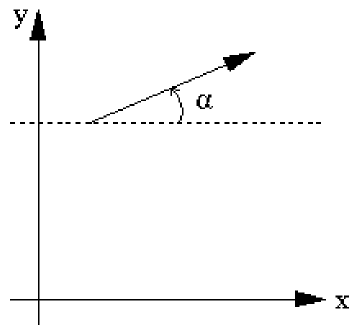
5.12.2 Понятие “тензор”

Чтобы хорошо понять, что такое *тензор*, потребуется иметь хорошие знания в линейной алгебре и свободно производить вычисления с векторами. Говоря о прикладной части, тензоры реализованы в TensorFlow просто как многомерные массивы данных. Чтобы лучше понять их природу, а также оценить влияние на машинное обучение в целом, следует вникнуть чуть глубже.

5.12.2.1 Математические векторы

Начать стоит с того, что вообще такое вектор. *Вектор* — в простейшем случае математический объект, характеризующийся величиной и направлением. Другими словами, *вектор* — скалярная величина, которая имеет направление.

Длина математического вектора — это абсолютная величина, в то время как направление — относительная. Длина измеряется относительно направления, а в качестве единиц выступают градусы или радианы. Принято, что направление положительно и отсчитывается против часовой стрелки относительно начальной точки отсчета.



5.12.2.2 Плоские векторы

Плоский вектор — это простейший тензор. Они очень похожи на обычные векторы, такие как вы видели выше, однако они могут сами определять себя в векторном пространстве.

Иметь данные вектора и их представление на координатной плоскости полезно, но, в целом, важно лишь то, какие можно производить операции над ними. В этом поможет выражение данных векторов через базисные или единичные вектора.

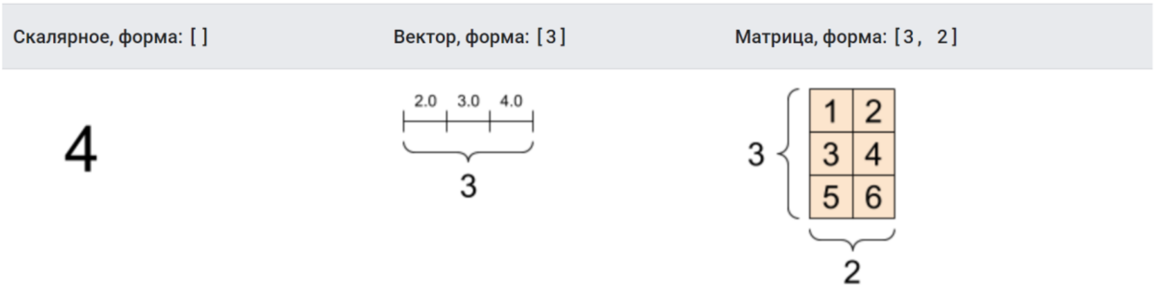
Единичные векторы — векторы длиной в единицу. Двумерные или трехмерные вектора хорошо раскладываются в сумму ортогональных единичных векторов, таких как оси координат.

5.12.2.3 Тензоры

Плоский вектор — это лишь частный случай тензора. Вектор, как было сказано выше, определяется как скаляр, которому задали направление. Тензор же — это математическое представление физической сущности, которая может быть задана величиной и ***несколькими*** направлениями.

И так же, как можно представить скаляр одним числом, а трехмерный вектор как тройку чисел, тензор представляется в виде массива 3R чисел в трехмерном пространстве.

R в этой записи означает ранг тензора: в трехмерном пространстве тензор ранга 2 может быть представлен девятью числами. В N-мерном пространстве скаляр требует только одного числа, вектор требует N чисел, а тензор уже требует $\mathbf{N^R}$ чисел. Этим объясняется, почему скаляры часто называют тензорами ранга 0: у них нет направления, и они могут быть представлены только одним числом.



5.12.3 Работа с тензорами в TensorFlow

Создадим несколько тензоров.

Тензор ранга 0

```
>>> rank_0_tensor = tf.constant(4)
>>> print(rank_0_tensor)
tf.Tensor(4, shape=(), dtype=int32)
```

Тензор ранга 1

```
>>> rank_1_tensor = tf.constant([2.0, 3.0, 4.0])
>>> print(rank_1_tensor)
tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
```

Тензор ранга 2

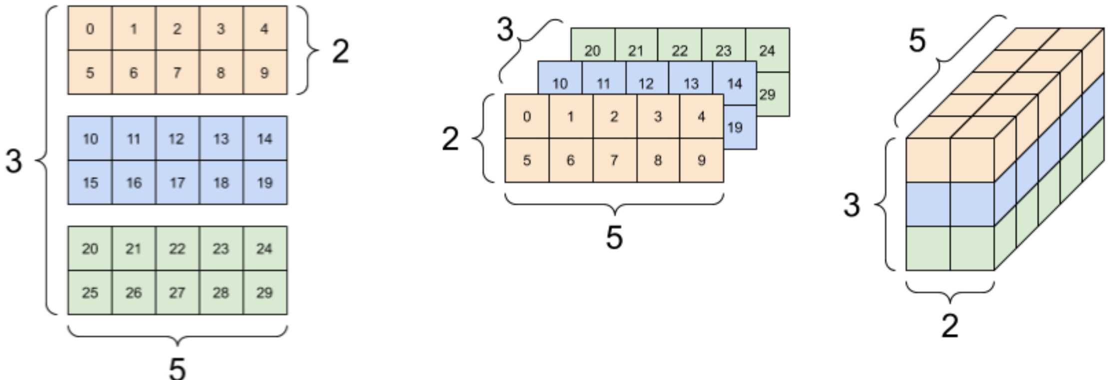
```
>>> rank_2_tensor = tf.constant([[1, 2],
                                 [3, 4],
                                 [5, 6]],
dtype=tf.float16)
>>> print(rank_2_tensor)
tf.Tensor(
[[1. 2.]
 [3. 4.]
 [5. 6.]], shape=(3, 2), dtype=float16)
```

У тензоров может быть больше осей; вот тензор с тремя осями:

```
>>> rank_3_tensor = tf.constant([
  [[0, 1, 2, 3, 4],
   [5, 6, 7, 8, 9]],
  [[10, 11, 12, 13, 14],
   [15, 16, 17, 18, 19]],
  [[20, 21, 22, 23, 24],
   [25, 26, 27, 28, 29]],])
>>> print(rank_3_tensor)
tf.Tensor(
```

```
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

Графически он может быть представлен разными способами:



5.12.4 Операции над тензорами

Как и с обычными многомерными матрицами, рассмотренными ранее, над тензорами можно проводить стандартные математические операции:

```
a = tf.constant([[1, 2],
                 [3, 4]])
b = tf.constant([[1, 1],
                 [1, 1]])
print(tf.add(a, b), "\n")
print(tf.multiply(a, b), "\n")
print(tf.matmul(a, b), "\n")

tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tf.Tensor(
```

```
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)
```

Либо же, используя более лаконичные аналогичные записи:

```
print(a + b, "\n")
print(a * b, "\n")
print(a @ b, "\n")

tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)
```

Кроме того, можно использовать и другие операции:

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])

# Поиск максимального значения
print(tf.reduce_max(c))
# Поиск индекса максимального значения
print(tf.argmax(c))
# Подсчёт softmax
print(tf.nn.softmax(c))

tf.Tensor(10.0, shape=(), dtype=float32)
tf.Tensor([1 0], shape=(2,), dtype=int64)
tf.Tensor(
[[2.6894143e-01 7.3105854e-01]
 [9.9987662e-01 1.2339458e-04]], shape=(2, 2),
dtype=float32)
```

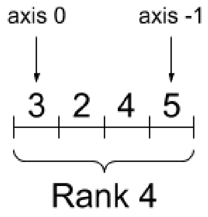
5.12.5 Формы тензоров

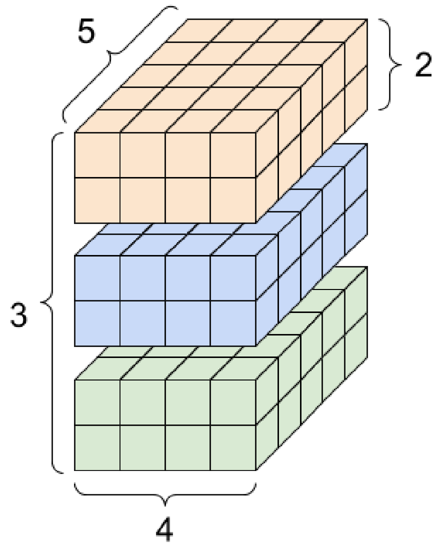
Тензоры имеют форму:

- *Форма*: Длина (количество элементов) каждой из осей тензора.
- *Ранг*: Число осей тензора. Скаляр имеет ранг 0, вектор - ранг 1, матрица – ранг 2.
- *Ось* или *измерение*: Определенная размерность тензора.
- *Размер*: Общее количество элементов в тензоре.

Например:

```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```






```
print("Тип каждого элемента:", rank_4_tensor.dtype)
print("Количество осей:", rank_4_tensor.ndim)
print("Форма тензора:", rank_4_tensor.shape)
print("Элементы вдоль оси 0 тензора:",
rank_4_tensor.shape[0])
print("Элементы вдоль последней оси тензора:",
rank_4_tensor.shape[-1])
print("Общее количество элементов (3*2*4*5): ",
tf.size(rank_4_tensor).numpy())

Тип каждого элемента: <dtype: 'float32'>
```

```
Количество осей: 4
Форма тензора: (3, 2, 4, 5)
Элементы вдоль оси 0 тензора: 3
Элементы вдоль последней оси тензора: 5
Общее количество элементов (3*2*4*5): 120
```

Во многом, взаимодействие с данными схоже по своей сути с таковой в представленных выше библиотеках.

Как и с библиотекой NumPy, в TensorFlow для тензоров поддерживается синтаксис стандартных для Python срезов, в том числе и многоосевых. Манипуляции с изменением формы также доступны. При необходимости, можно перевести тензор в формат массива библиотеки NumPy через соответствующую команду:

```
>>> a = tf.constant([[1, 2], [3, 4]])
>>> b = tf.add(a, 1)

>>> a.numpy()
array([[1, 2],
       [3, 4]], dtype=int32)

>>> b.numpy()
array([[2, 3],
       [4, 5]], dtype=int32)

>>> tf.multiply(a, b).numpy()
array([[ 2,  6],
       [12, 20]], dtype=int32)
```

5.12.6 Модули, слои и модели

tf.Module - это класс для управления объектами tf.Variable и объектами tf.function, которые работают с ними. Класс tf.Module необходим для поддержки двух важных функций:

- Вы можете сохранять и восстанавливать значения ваших переменных с помощью tf.train.Checkpoint. Это полезно во время обучения, так как позволяет быстро сохранять и восстанавливать состояние модели.
- Вы можете импортировать и экспортировать значения tf.Variable и графики tf.function с помощью tf.saved_model. Это позволяет запускать модель независимо от программы Python, которая ее создала.

Вот полный пример экспорта простого объекта tf.Module:

```
class MyModule(tf.Module):
  def __init__(self, value):
    self.weight = tf.Variable(value)

  @tf.function
  def multiply(self, x):
    return x * self.weight

mod = MyModule(3)
mod.multiply(tf.constant([1, 2, 3]))
```

```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([3,
6, 9], dtype=int32)>
```

Сохранение модуля (*Module*):

```
save_path = './saved'
tf.saved_model.save(mod, save_path)
```

```
INFO:tensorflow:Assets written to: ./saved/assets
2021-12-14 18:18:18.020215: W
tensorflow/python/util/util.cc:368] Sets are not
currently considered sequences, but this may change
in the future, so consider avoiding using them.
```

Результат SavedModel не зависит от языка, на котором он был создал. Вы можете загрузить SavedModel из Python, либо других языков. Например, для работы с TensorFlow Lite или TensorFlow JS.

Сама загрузка модуля выглядит следующим образом:

```
reloaded = tf.saved_model.load(save_path)
reloaded.multiply(tf.constant([1, 2, 3]))
```

```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([3,
6, 9], dtype=int32)>
```

Тренировочные циклы

Поскольку библиотека TensorFlow, в первую очередь, создана для машинного обучения, то в ней уже реализованы возможности для создания моделей и циклов их тренировки.

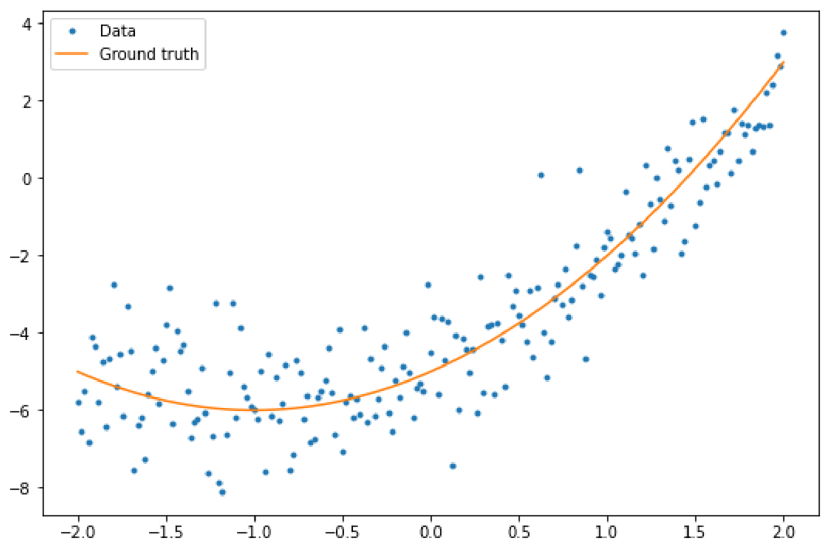
Для этого соберём все вместе, чтобы построить базовую модель и обучить ее с нуля.

Сначала создаются данные, а точнее, создается облако точек, которое в общих чертах повторяет квадратичную кривую:

```
import matplotlib
from matplotlib import pyplot as plt

matplotlib.rcParams['figure.figsize'] = [9, 6]

x = tf.linspace(-2, 2, 201)
x = tf.cast(x, tf.float32)

def f(x):
  y = x**2 + 2*x - 5
  return y

y = f(x) + tf.random.normal(shape=[201])

plt.plot(x.numpy(), y.numpy(), '.', label='Data')
plt.plot(x, f(x),  label='Ground truth')
```

```
plt.legend();
```

Для отображения используется библиотека Matplotlib:

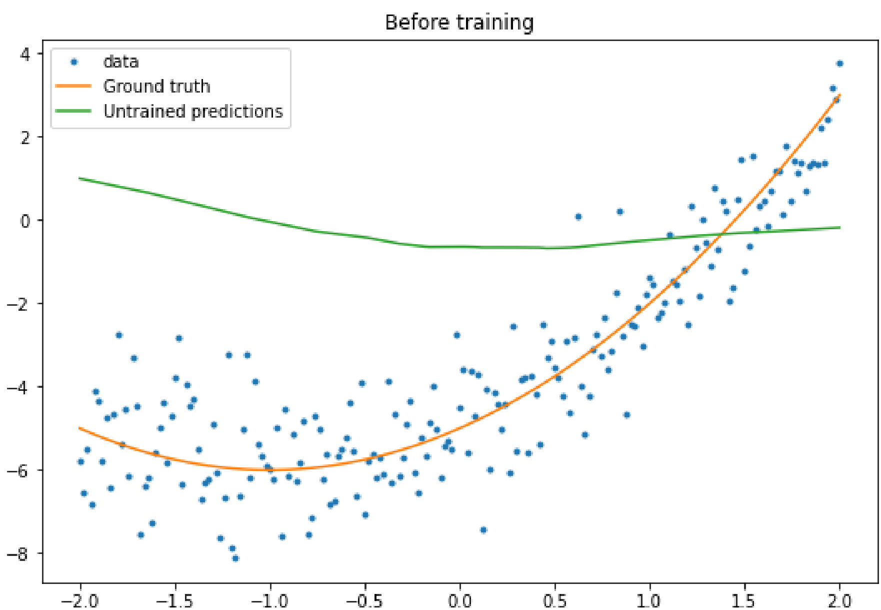


Процесс создания модели будет выглядеть следующим образом:

```
class Model(tf.keras.Model):
  def __init__(self, units):
    super().__init__()
    self.dense1 = tf.keras.layers.Dense(units=units,
                                        activation=tf
.nn.relu,
                                        kernel_initia
lizer=tf.random.normal,
                                        bias_initiali
zer=tf.random.normal)
    self.dense2 = tf.keras.layers.Dense(1)

  def call(self, x, training=True):
    x = x[:, tf.newaxis]
    x = self.dense1(x)
    x = self.dense2(x)
    return tf.squeeze(x, axis=1)

model = Model(64)
```

```
plt.plot(x.numpy(), y.numpy(), '.', label='data')
plt.plot(x, f(x), label='Ground truth')
plt.plot(x, model(x), label='Untrained predictions')
plt.title('Before training')
plt.legend();
```

Результатом кода будет зелёная линия, отображающая предсказание нейронной сети без проведённого обучения:



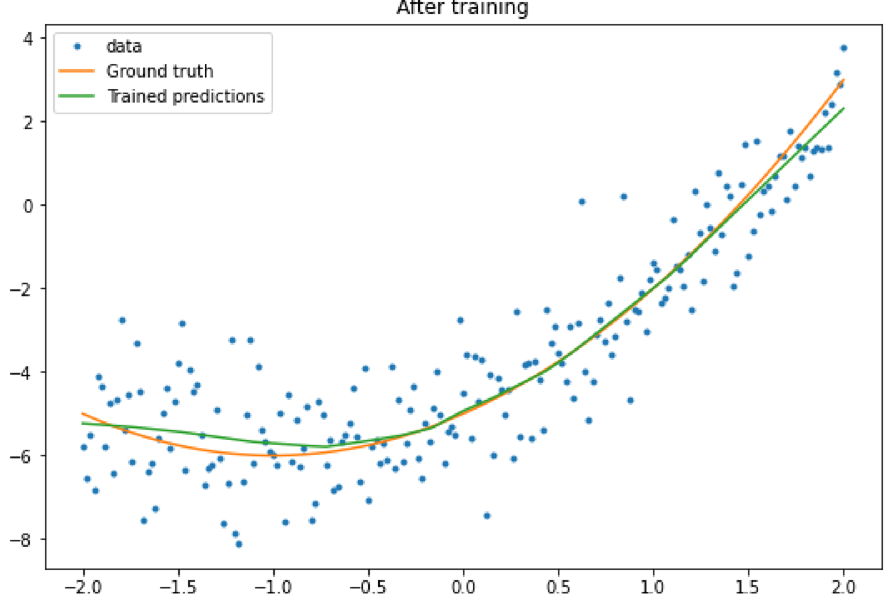
Для повышения точности прогноза, требуется провести обучение:

```
variables = model.variables

optimizer = tf.optimizers.SGD(learning_rate=0.01)

for step in range(1000):
  with tf.GradientTape() as tape:
    prediction = model(x)
    error = (y-prediction)**2
    mean_error = tf.reduce_mean(error)
  gradient = tape.gradient(mean_error, variables)
  optimizer.apply_gradients(zip(gradient, variables))

  if step % 100 == 0:
    print(f'Mean squared error:
{mean_error.numpy():0.3f}')
```

```
Mean squared error: 19.473
Mean squared error: 1.140
Mean squared error: 1.130
Mean squared error: 1.124
Mean squared error: 1.121
Mean squared error: 1.120
Mean squared error: 1.119
Mean squared error: 1.118
Mean squared error: 1.118
Mean squared error: 1.117
```

Проверка результата цикла обучения:

```
plt.plot(x.numpy(),y.numpy(), '.', label="data")
plt.plot(x, f(x),  label='Ground truth')
plt.plot(x, model(x), label='Trained predictions')
plt.title('After training')
plt.legend();
```



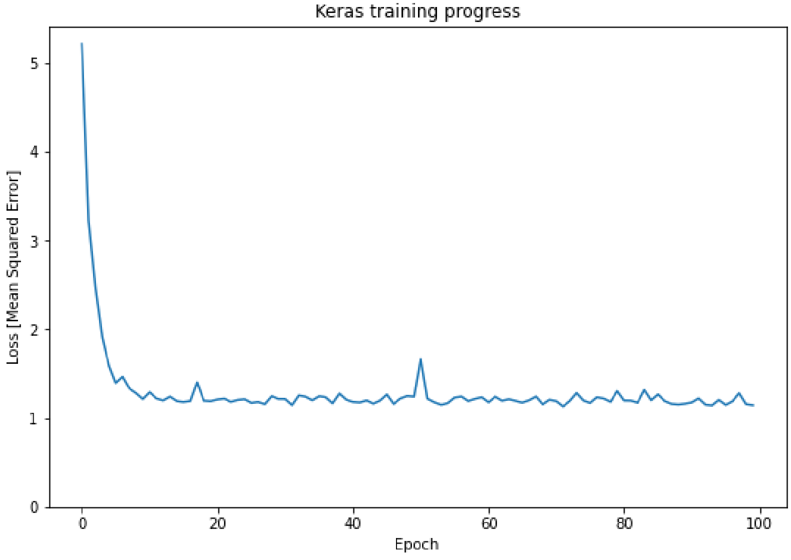
Обучение работает, но следует помнить, что реализации наиболее распространенных утилит обучения доступны в модуле tf.keras. Поэтому, прежде чем писать свои собственные, воспользуйтесь ими. Для начала методы Model.compile и Model.fit реализуют для вас цикл обучения:

```
new_model = Model(64)

new_model.compile(
    loss=tf.keras.losses.MSE,
    optimizer=tf.optimizers.SGD(learning_rate=0.01))

history = new_model.fit(x, y,
                        epochs=100,
                        batch_size=32,
                        verbose=0)

model.save('./my_model')
```

```
INFO:tensorflow:Assets written to: ./my_model/assets
```

Результат обучения с помощью встроенных методов:

```
plt.plot(history.history['loss'])
plt.xlabel('Epoch')
plt.ylim([0, max(plt.ylim())])
plt.ylabel('Loss [Mean Squared Error]')
plt.title('Keras training progress');
```



5.12.7 Пример 1. Y=(3X + 1)

Теперь приступим к реализации простейшей нейронной сети TensorFlow Keras для решения простейшей математической задачи. Рассмотрим следующие наборы чисел, представленные в таблице:

X:	-1	0	1	2	3	4
Y:	-2	1	4	7	10	13

Данные пары значений соответствуют результату уравнения Y = 3X + 1 и они необходимы нам для обучения нейронной сети, которую нам предстоит написать.

Начнем с подключения необходимых библиотек. Здесь мы импортируем TensorFlow и называем его «tf» для простоты использования. Далее мы импортируем библиотеку с именем «numpy», которая помогает нам легко и быстро представлять наши данные в виде списков. Структура для определения нейронной сети как набора последовательных слоев называется «keras», поэтому мы импортируем её тоже.

```
import tensorflow as tf  #библиотека для машинного обучения
import numpy as np  #математическая библиотека (поддержка многомерных массивов и
                    #высокоуровневых математических функций над ними
from tensorflow import keras #библиотека высокоуровневого API для tensorflow и н
from tensorflow.keras.utils import plot_model #утилита для визуализации полученн

print(tf.__version__)
print(keras.__version__)
print(np.__version__)
```

2.5.0

2.5.0

1.19.5

Далее мы создадим простейшую из возможных нейронных сетей. У неё будет 1 слой («keras.layers.Dense()» – добавление слоя), в котором – 1 нейрон (units=1), и входная в него величина имеет только размерность 1, т.е. на вход данного слоя приходит только одно значение (input_shape=[1] или input_dim=1 («dim»–dimension, размерность). Так же можно указать и функцию активации этого слоя (таблица 2, их описание доступно в

официальной документации: https://keras.io/activations/) «activation='linear'», но линейная функция активации является значением по умолчанию, поэтому не будем это отдельно указывать.

Название функции	Область значений	Синтаксис	Вид/формула
Экспоненциальная линейная функция (англ. Exponential linear unit, ELU). Синтаксис:	(0, 1)	keras.activations.elu(x, alpha=1.0)	$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & x < 0 \\ x & x \geq 0 \end{cases}$
Softmax – это обобщение логистической функции для многомерного случая. Функция преобразует вектор Z размерности K в вектор той же размерности, где каждая координата σ (сигма) полученного вектора представлена вещественным числом в интервале [0,1] и сумма координат равна 1.	(0, 1)	keras.activations.softmax(x, axis=-1)	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$
Масштабированная экспоненциальная линейная функция (англ. Scaled exponential linear unit, SELU)., где λ =1,0507 и α =1,67326	(-αλ, ∞)	keras.activations.selu(x)	$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & x < 0 \\ x & x \geq 0 \end{cases}$
Softplus.	(0, ∞)	keras.activations.softplus(x)	$f(x) = \ln(1 + e^x)$
Softsign	(-1, 1)	keras.activations.softsign(x)	$f(x) = \frac{x}{1 + \lvert x \rvert}$
Линейный выпрямитель с «утечкой» (англ. Leaky rectified linear unit, Leaky ReLU)	(-∞, ∞)	keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0.0)	$f(x) = \begin{cases} 0{,}01x & x < 0 \\ x & x \geq 0 \end{cases}$
Гиперболический тангенс	(-1, 1)	keras.activations.tanh(x)	$f(x) = \operatorname{th}(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
Логистическая (сигмоида или Гладкая ступенька)	(0, 1)	keras.activations.sigmoid(x)	$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$
Hard sigmoid (резкий сигмоид). Эта функция является кусочно-линейной аппроксимацией сигмоидальной функции. Равна 0 в диапазоне [-∞; -2,5), затем линейно увеличивается от 0 до 1 в диапазоне [-2,5; +2,5] и остается равной 1 в диапазоне (+2,5; +∞].	[0, 1]	keras.activations.hard_sigmoid(x)	
Экспоненциальная функция активации. F(x)=ex=exp(x)	(0, ∞)	keras.activations.exponential(x)	(0,1) (1,e)
Линейная функция активации (или тождественная).	(-∞, ∞)	keras.activations.linear(x)	$f(x) = x$

Финальный вид кода для создания описанной модели нейронной сети, выглядит так:

```
#use_bias=True - использовать нейрон смещения (хотя он и так по умолчанию исполь
model = tf.keras.Sequential(
    [keras.layers.Dense(units=1, input_shape=[1], use_bias=True)])
```

Напишем код для компиляции нашей нейронной сети. Когда мы делаем это, мы должны указать 2 функции: «потери» и «оптимизатор». Функция «потери» считает разность угаданных ответов с известными правильными ответами и измеряет, насколько точно произведено решение. Затем модель использует функцию оптимизатора, чтобы сделать еще одно предположение. Основываясь на результатах функции потерь, она попытается минимизировать потери. Модель повторит это для заданного количества эпох.

Укажем программе использовать «среднеквадратичную ошибку» («mean_squared_error») для потерь и «стохастический градиентный спуск» («sgd») для оптимизатора (список алгоритмов оптимизации доступен в официальной документации: https://keras.io/optimizers/).

```
model.compile(optimizer='sgd', loss='mean_squared_error')
```

Далее мы введем обучающую выборку для нашего уравнения. Библиотека «numpy» предоставляет множество структур данных в типе данных «массив», для этого используем функцию «np.array»:

```
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-2.0, 1.0, 4.0, 7.0, 10.0, 13.0], dtype=float)
```

Код, необходимый для определения нейронной сети, написан. Теперь обучим нашу нейросеть. Процесс обучения нейронной сети (рис. 13), где она «изучает» отношения между X и Y, находится в процессе «model.fit». Здесь программа пройдет цикл, о котором было написано ранее: получит догадку, измерит, насколько она точна (иначе говоря, потерю), использует оптимизатор, чтобы сделать еще одну догадку и так далее. Программа сделает это за количество эпох, которое вы укажете. Когда вы запустите этот код, вы увидите, что по мере прогрессирования обучения нейронной сети

потери будут уменьшаться с каждой следующей эпохой и станут очень маленькими. Не факт, что для достижения заданной точности нужны 40 эпох, поэтому вы можете поэкспериментировать с их количеством.

```
history = model.fit(xs, ys, epochs=40)
```

Epoch 1/40

1/1 [==============================] - 0s 168ms/step - loss: 112.8983

Epoch 2/40

1/1 [==============================] - 0s 2ms/step - loss: 88.8381

Epoch 3/40

1/1 [==============================] - 0s 2ms/step - loss: 69.9084

Epoch 4/40

1/1 [==============================] - 0s 2ms/step - loss: 55.0151

Epoch 5/40

1/1 [==============================] - 0s 2ms/step - loss: 43.2975

Epoch 6/40

1/1 [==============================] - 0s 2ms/step - loss: 34.0783

Epoch 7/40

1/1 [==============================] - 0s 2ms/step - loss: 26.8249

Epoch 8/40

1/1 [==============================] - 0s 2ms/step - loss: 21.1179

Epoch 9/40

1/1 [==============================] - 0s 2ms/step - loss: 16.6277

Epoch 10/40

1/1 [==============================] - 0s 2ms/step - loss: 13.0947

Epoch 11/40

1/1 [==============================] - 0s 2ms/step - loss: 10.3149

Epoch 12/40

1/1 [==============================] - 0s 3ms/step - loss: 8.1276

Epoch 13/40

1/1 [==============================] - 0s 2ms/step - loss: 6.4064

Epoch 14/40

1/1 [==============================] - 0s 2ms/step - loss: 5.0521

Epoch 15/40

1/1 [==============================] - 0s 2ms/step - loss: 3.9863

Epoch 16/40

1/1 [==============================] - 0s 2ms/step - loss: 3.1476

Epoch 17/40

1/1 [==============================] - 0s 2ms/step - loss: 2.4875

Epoch 18/40

1/1 [==============================] - 0s 2ms/step - loss: 1.9679

Epoch 19/40

1/1 [==============================] - 0s 2ms/step - loss: 1.5589

Epoch 20/40

1/1 [==============================] - 0s 2ms/step - loss: 1.2369

Epoch 21/40

1/1 [==============================] - 0s 2ms/step - loss: 0.9834

Epoch 22/40

1/1 [==============================] - 0s 2ms/step - loss: 0.7837

Epoch 23/40

1/1 [==============================] - 0s 2ms/step - loss: 0.6264

Epoch 24/40

1/1 [==============================] - 0s 2ms/step - loss: 0.5024

Epoch 25/40

1/1 [==============================] - 0s 2ms/step - loss: 0.4047

Epoch 26/40

1/1 [==============================] - 0s 2ms/step - loss: 0.3276

Epoch 27/40

1/1 [==============================] - 0s 2ms/step - loss: 0.2668

Epoch 28/40

1/1 [==============================] - 0s 2ms/step - loss: 0.2187

Epoch 29/40

1/1 [==============================] - 0s 2ms/step - loss: 0.1808

Epoch 30/40

1/1 [==============================] - 0s 2ms/step - loss: 0.1507

Epoch 31/40

1/1 [==============================] - 0s 2ms/step - loss: 0.1269

Epoch 32/40

1/1 [==============================] - 0s 2ms/step - loss: 0.1080

Epoch 33/40

1/1 [==============================] - 0s 2ms/step - loss: 0.0929

Epoch 34/40

1/1 [==============================] - 0s 2ms/step - loss: 0.0809

Epoch 35/40

1/1 [==============================] - 0s 2ms/step - loss: 0.0713

Epoch 36/40

1/1 [==============================] - 0s 2ms/step - loss: 0.0636

Epoch 37/40

1/1 [==============================] - 0s 2ms/step - loss: 0.0574

Epoch 38/40

1/1 [==============================] - 0s 2ms/step - loss: 0.0523

Epoch 39/40

1/1 [==============================] - 0s 2ms/step - loss: 0.0482

Epoch 40/40

1/1 [==============================] - 0s 1ms/step - loss: 0.0448

Метод «model.evaluate()» возвращает значения потерь при следующих аргументах: входные данные, выходные данные. В данном случае этот метод вернет минимальное значение потери обученной модели, которое достигнуто при последней эпохе обучения:

```
print(model.evaluate(xs, ys))
```

1/1 [==============================] - 0s 75ms/step - loss: 0.0420
0.042010411620140076

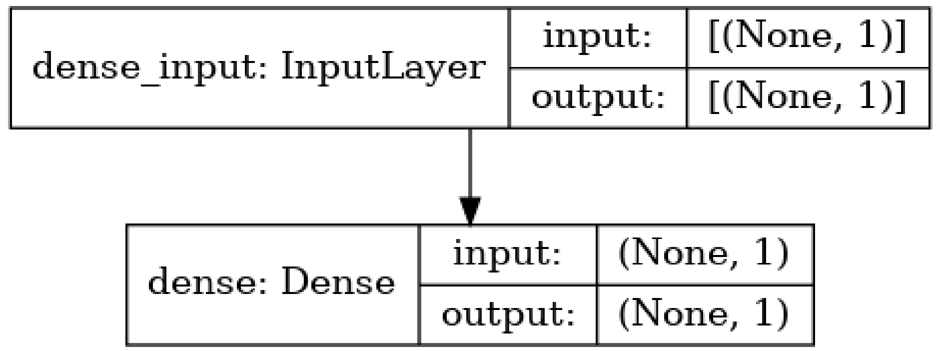
После того, как вы обучили модель для решения уравнения Y = 3X + 1, Вы можете использовать метод «model.predict()», чтобы он вычислил Y для указанного и ранее неизвестного X:

```
print(model.predict([10.0]))
```

[[30.091446]]

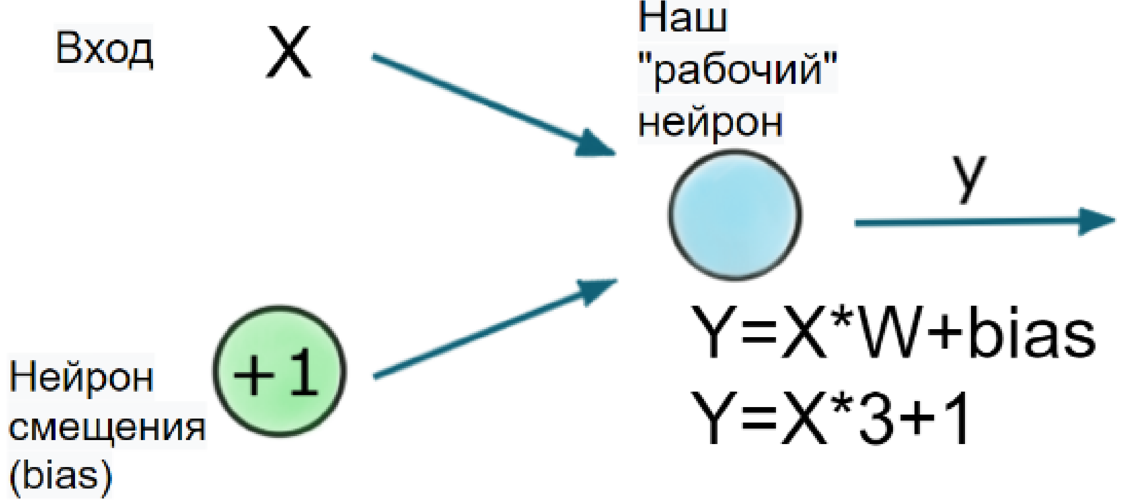
Чтобы сохранить структуру нейронной сети в виде картинки, воспользуемся следующим кодом, и программа сохранит модель в файл model_example1.png:

```
plot_model(model,
           to_file='model_example1.png',
           show_shapes=True,
           show_layer_names=True,
           rankdir='TB',
           expand_nested=False,
           dpi=96)
```



```python
for layer in model.layers:
    weights = layer.get_weights()  # list of numpy arrays
print(weights)
print(f'Weight = {round(weights[0][0][0], 2)}')
print(f'Bias = {round(weights[1][0],2)}')
```

```
[array([[2.8857362]], dtype=float32), array([1.2340839], dtype=float32)]
Weight = 2.890000104904175
Bias = 1.2300000190734863
```




```python
import matplotlib.pyplot as plt
%matplotlib inline

print(history.history.keys())

plt.title(
    'Изменение метрики обучения "ошибка" в зависимости от итерации обучения\n')
plt.xlabel('Итерация обучения, или эпоха (Epoch)')
plt.ylabel('Средне-квадратичная ошибка\n(mean_squared_error)')
plt.plot(history.history['loss'])

model.get_config()['layers'][1]
```

dict_keys(['loss'])

{'class_name': 'Dense',

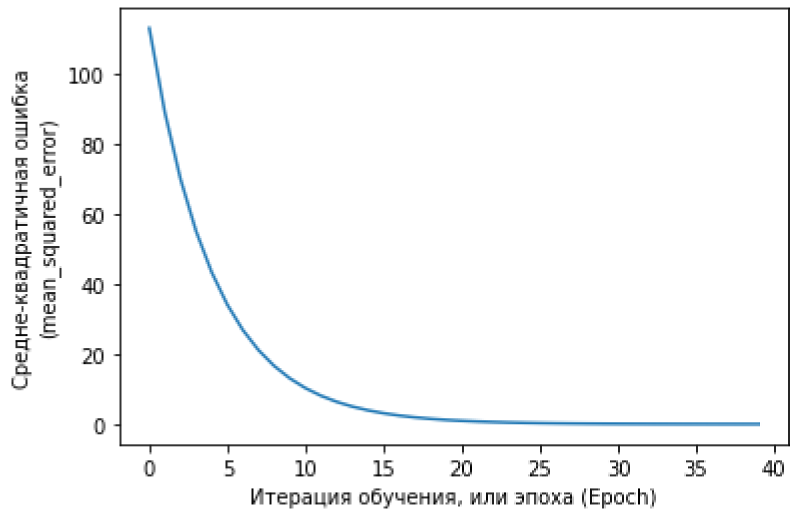
'config': {'name': 'dense',

'trainable': True,

'batch_input_shape': (None, 1),

'dtype': 'float32',

```
'units': 1,
'activation': 'linear',
'use_bias': True,
'kernel_initializer': {'class_name': 'GlorotUniform',
 'config': {'seed': None}},
'bias_initializer': {'class_name': 'Zeros', 'config': {}},
'kernel_regularizer': None,
'bias_regularizer': None,
'activity_regularizer': None,
'kernel_constraint': None,
'bias_constraint': None}}
```



Для вычисления среднеквадратической ошибки (MSE) все отдельные остатки регрессии возводятся в квадрат, суммируются, сумма делится на общее число ошибок:

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Computes the mean of squares of errors between labels and predictions. loss = square(y_true - y_pred)

```
y_true = [[0., 1.], [0., 0.]]
y_pred = [[1., 1.], [1., 0.]]
# Using 'auto'/'sum_over_batch_size' reduction type.
mse = tf.keras.losses.MeanSquaredError()
mse(y_true, y_pred).numpy()
```

0.5

((1-0)^2+(1-1)^2+(1-0)^2+(0-0)^2)/4=0.5

```
x_pred = []
y_pred = []
for var in range(-1, 5):
    y_pred.append(model.predict([var])[0][0])
    x_pred.append(var)

import plotly.offline as pyoff
import plotly.graph_objs as go

plot_data = [
    go.Scatter(
        x=x_pred,
        y=y_pred,
        name='Предсказание',
    ),
    go.Scatter(
        x=xs,
        y=ys,
        name='Реальные данные',
    )
]

plot_layout = go.Layout(
    xaxis={"type": "linear"},
    title=
    'График по реальным параметрам и график предсказаний (сильно приблизьте, что
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-2a5fe84180e3> in <module>
      2 y_pred = []
      3 for var in range(-1, 5):
----> 4     y_pred.append(model.predict([var])[0][0])
      5     x_pred.append(var)
      6 

NameError: name 'model' is not defined
```

В конечном итоге, написан код на языке Python при помощи фреймворка TensorFlow с надстройкой Keras, являющийся реализацией нейросети для решения уравнения Y = 3X + 1.