**Работа с гипотезами**

Во главе исследования стоит гипотеза – предположение, требующее доказательства. Аналитик почти всегда говорит на языке гипотез, редко – на языке утверждений. Пока гипотеза не проверена на данных, она остаётся только предположением.

**Порядок формулирования гипотезы:**

1. **Определить цель.** Например, вы хотите, чтобы посетители сайта покупали больше. Вокруг этой точки и следует формировать гипотезы, именно на неё будут влиять результаты исследования;
2. **Предполагать.** В свободной форме сделайте предположения о том, что влияет на выбранное для анализа явление. Например, «Число покупок зависит от внешнего вида карточек товаров». Избегайте суждений в духе «это и так понятно». В аналитике нет ничего очевидного – всё нужно сначала доказать;
3. **Уточнить гипотезу.** Здесь два пути: можно постараться сделать гипотезу максимально точной с самого начала либо сразу приступить к проверке. В ходе работы вы обязательно сформируете новые, уточняющие предположения. Например, гипотеза «Число покупок зависит от внешнего вида карточек товаров» через некоторое время может превратиться в такую: «Возможность предварительного просмотра в карточке товара влияет на рост числа покупок».
4. **Действовать.** В хорошей гипотезе всегда есть не только цель, но и действие: «Если добавить в карточки товаров возможность предварительного просмотра, это увеличит число покупок». При формулировании гипотез один из этих элементов часто теряют. Например, вы делаете редизайн сайта. Но зачем? Есть действие, но нет цели. Или хотите увеличить продажи, но не знаете как. Цель есть, действия нет.



Подтвердится гипотеза или нет – не так важно. Оба результата полезны для бизнеса.

**Этапы классического процесса анализа данных**

1. **Постановка задачи**

Уточнение запроса и формирование образа конечного результата.

Важно всегда уточнять требования заказчика (прорабатывать постановку задачи) по следующему сценарию:

1. **Уточнить первоначальный запрос.** (как планируется применять запрашиваемую информацию) Примеры вопросов: Какую бизнес-проблему надо решить? За какой период изучить данные?
2. **Сформулировать промежуточный результат.** (и проговорить его, с постановщиком задачи для подтверждения правильности)
3. **Уточнить требования. (**определить пожелания к результату) Примеры вопросов: Каковы обязательные требования к результату? Как ты будешь использовать эту информацию?

По итогам этого этапа должны быть сформированы гипотеза (гипотезы) и образ конечного результата.

После этого выполняется декомпозиция задачи – разбивка задачи на небольшие шаги в соответствии с этапами процесса анализа данных.

1. **Получение данных**

Данные всегда где-нибудь хранятся: в базе, файле, на сайте, в онлайн-таблице. Если вы хорошо знаете, где взять данные для проекта, на этом этапе важно получить их в удобном для анализа формате. Если же вам предстоит работа с новыми источниками данных, обратитесь к разработчикам или менеджерам – они подскажут, где лежат данные и как их получить.

1. **Предобработка данных**

На практике данные часто содержат пропуски, дубликаты, ошибки. Например, при заполнении формы регистрации на сайте кто-нибудь обязательно забудет написать фамилию, а цена товара вместо ожидаемых рублей может быть выражена копейками. Ваша задача во время предобработки – найти и устранить такие проблемы. Обнаружить абсолютно все оплошности вам вряд ли удастся, но результатом этого этапа будут готовые к дальнейшему анализу данные.

1. **Исследовательский анализ данных**

Исследовательский анализ данных, или EDA, предполагает построение графиков по каждому столбцу в наборе данных. Такая визуализация позволяет увидеть некоторые особенности, закономерности и аномалии. Например, график может показать, что определённый курьер в компании доставляет заказы неправдоподобно быстро. Сделанные наблюдения помогут вам разобраться в результатах дальнейшего анализа.

1. **Подготовка прототипа решения**

На этом этапе вы готовите первый рабочий вариант решения, или прототип. Работа над аналитической задачей – цикличный процесс. Иногда прототип может стать её финальным решением, но обычно хороший результат требует неоднократных доработок.

1. **Проверка результатов**

Решение подготовлено — пора изучить результаты. Это очень ответственный момент: вы определяете, решена ли задача или нужно ещё что-нибудь доделать. Как это определить? Во-первых, соотнесите результаты с поставленной задачей и образом конечного результата. Во-вторых, оцените правдоподобность результатов, руководствуясь здравым смыслом. Могут ли быть такие значения в этом контексте? Почему значения именно такие? Результат этого этапа – завершение проекта или выявленные недоработки.

1. **Доработка решения**

Обнаружив несовершенства в прототипе решения во время проверки результатов, вы переходите к его доработке. Этот этап замыкает цикл из трёх основных шагов решения аналитической задачи: подготовка прототипа решения —> проверка результатов —> внесение изменений. Цикл повторяется до тех пор, пока результат вас не удовлетворит.

1. **Оформление результатов**

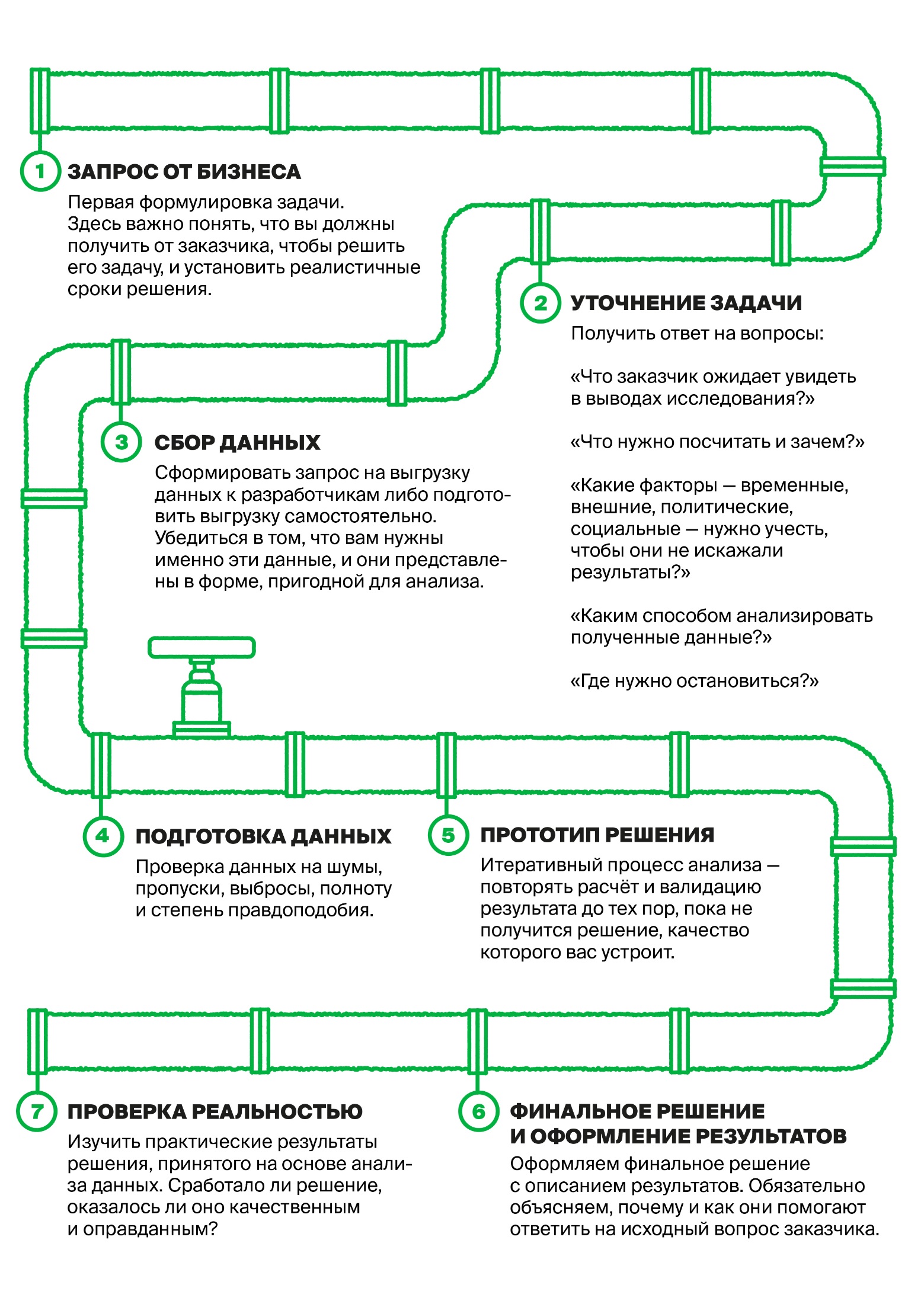
Задача решена, но заказчик об этом пока не знает. Вы можете сделать выгрузку данных или подготовить отчёт с визуализацией результатов, выводами и рекомендациями. Как узнать, в каком формате заказчик ожидает результат? Лучше всего просто спросить. Дополнительно уточните, как заказчик планирует применить полученное решение: возможно, вы предложите более подходящий вариант представления результата.

1. **Проверка реальностью**

Наиболее интересный, но и самый сложный этап. Результаты работы аналитика обычно трудно сразу же оценить. Например, вы изучили информацию о пользователях приложения и представили свои рекомендации по повышению продаж. После этого заказчик исследования, обычно менеджер из команды развития бизнеса, решает – принять ваши предложения или нет. Если ответ – да, через некоторое время можно будет оценить их эффективность. Так вы узнаете, действительно ли ваше решение было корректным.

Пайплайн (трубопровод, конвейер) – это порядок действий.

**Пайплайн аналитической задачи.**



**Модель HEART**

Модель HEART используют для оценки реакции пользователей на обновления.

Пять метрик HEART модели:

* **Happiness** — счастье. Например, высокая оценка приложения от пользователя;
* **Engagement** — вовлечённость. Например, количество действий, которые пользователь совершает за единицу времени.
* **Adoption** — принятие сервиса. Например, нового обновления.
* **Retention** — удержание. Например, сколько раз пользователь открывал приложение за единицу времени.
* **Task success** — буквально, «успех задач». Эта метрика показывает, оправдываются ли ожидания пользователя от сервиса.

**Категориальные и количественные данные**

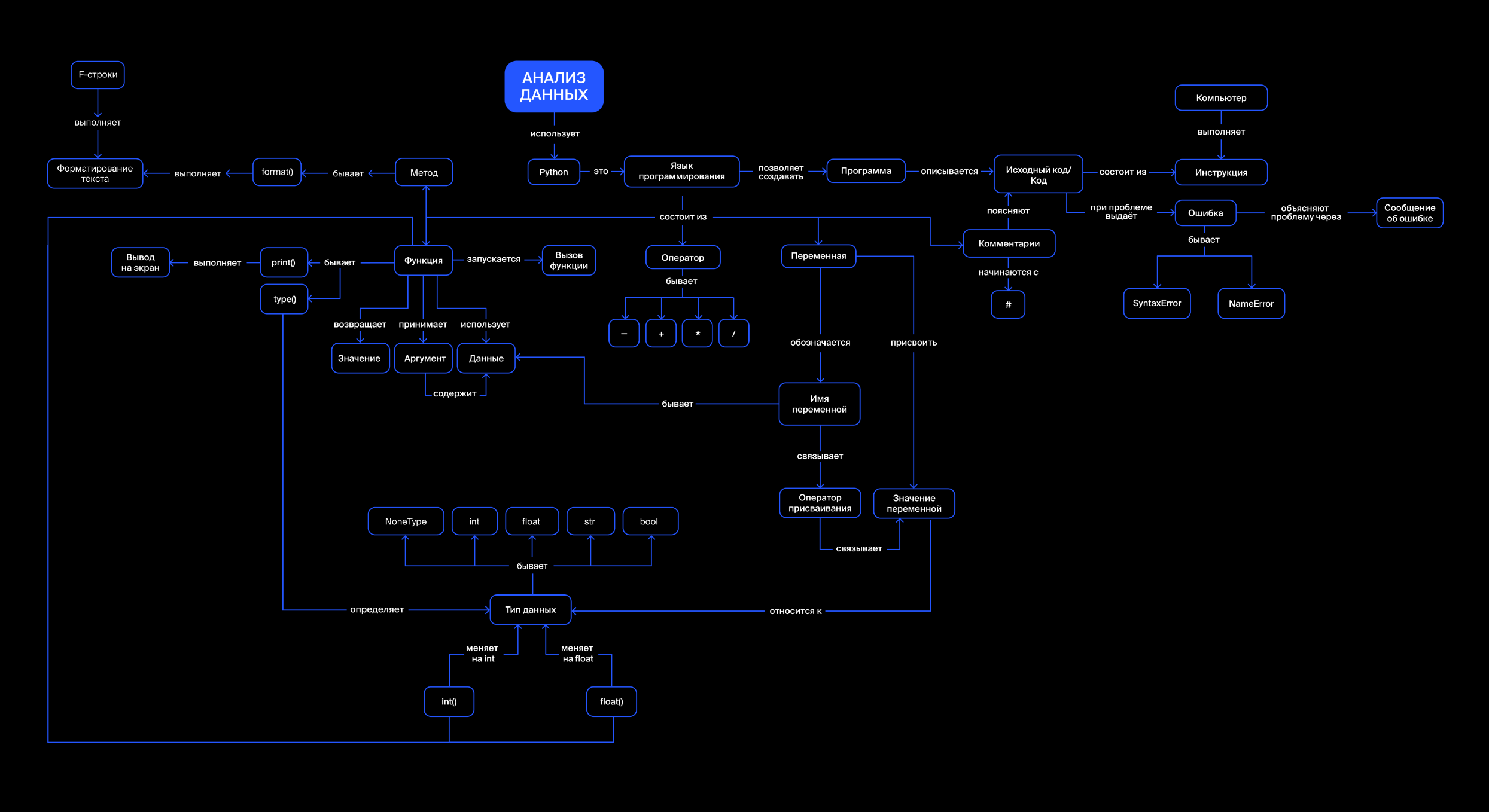
* Категориальная переменная принимает одно значение из ограниченного набора.
* Количественная принимает любое числовое значение в диапазоне.

**Стадии группировки данных**

Выражают формулой split-apply-combine:

* разделить, split — сначала данные разбивают на группы по определённому критерию;
* применить, apply — затем к каждой группе применяют методы вычисления, например: считают элементы группы методом count() или их суммы методом sum();
* объединить, combine — наконец, результаты сводят в новую структуру данных, таблицу или Series.

Концепт-карта



05.10.2022г. Первый код. (про робокотов)

import pandas

data = pandas.read\_csv('polomki.csv', index\_col='Магазин')

data['Неделя 14'] = data['Неделя 14'] \* 100

import seaborn

seaborn.heatmap(data)

Методы — это функции, которые вызываются через точку после самой строки.

Конструкция обработки исключений try … except. исходный код помещают в блок try. Если при выполнении кода из блока try возникнет ошибка, воспроизведётся код из блока except.

a = 1

b = 0

try:

print(a / b)

except:

print('Проверьте значения параметров a и b')

print('Кстати, хорошего дня')

Результат:

Проверьте значения параметров a и b

Кстати, хорошего дня

numbers = [10, 1, 0, -3, 999]

*# разделим каждое число из списка на само себя*

for number in numbers:

try:

print(f'{number} / {number} равно {number / number}')

except:

print(f'Не получилось разделить {number} на {number}')

print('Хорошего дня!')

Результат:

10 / 10 равно 1.0

1 / 1 равно 1.0

Не получилось разделить 0 на 0

-3 / -3 равно 1.0

999 / 999 равно 1.0

Хорошего дня!

Лямбда-функция – функция в одну строку.

lambda <аргумент>: <инструкция>

movies\_table\_sorted = sorted(movies\_table, key=lambda row: row[5]) *# Лямбда-функция принимает в качестве аргумента подсписок и выполняет инструкцию: получить из подсписка элемент с индексом* 5*. Этот элемент становится значением параметра* key *функции*sorted()*. В результате функция sorted() отсортирует подсписки по возрастанию элементов с индексом* 5.

Пример:

bike\_df['Age brackets'] = bike\_df['Age'].apply(lambda x: 'Less than 30' if x<=30 else('Greater than 55' if x>55 else '31 to 55'))

Функция dir(имя переменной) - подсказка, какие есть методы, аргументы для этой переменной/значения/вида данных.

? имя переменной – вызывает справку для этого вида данных

id(имя переменной) – показывает в какую ячейку памяти Python занес переменную. Небольшие значения из диапазона от -5 до 257 всегда заносятся в одну ячейку и не сбрасываются при каких-то действиях.

У функции print() есть параметр end, управляющий концом строки. По умолчанию там \n (специальный символ перевода на новую строку), но можно задать другое значение.

print(elem, end=' ') # печатается очередной элемент, в конец добавляются три пробела, без перевода на новую строку

\*\* — возведение в степень

// — целочисленное деление hours = action\_timing\_total // 60 *# находим результат целочисленного деления количества минут на 60*

% — остаток от деления minutes = action\_timing\_total % 60 *# находим остаток от деления количества минут на 60*

Для сокращения записи кода применяют операторы: += , -= , \*= , /= . С помощью них можно заменять запись вида "х = х + 1" на "х += 1".

Функция sum() — сложение. Например, всех значений в определённом столбце датасета или всех элементов списка. total\_duration = sum(movies\_duration)

Функция abs() — вычисление абсолютного значения (модуля) числа.

Функция max() — возвращает максимальное значение.

max\_duration = max(movies\_duration)

Функция min() — возвращает минимальное значение.

min\_duration = min(movies\_duration)

Функция prod() из библиотеки math — произведение.

mult\_duration = math.prod(movies\_duration)

Функция factorial() из библиотеки math — расчет факториала.

print(math.factorial(8))

Функция randint() из библиотеки random — выдает случайное число в заданном диапазоне:

import random

lucky\_number = random.randint(10, 100) *# случайное число в диапазоне от 10 до 100*

print(lucky\_number)

Типы данных:

* целые числа — int,
* вещественные числа (дробные числа) — float,
* строки — str,
* логический — bool,
* списки — list,
* дата, время — datetime,
* дата, время в формате UTC (unix time) (количество секунд, прошедших с 01.01.1970г.) — unix time,
* отсутствие значения (например, None) — NoneType.

Специальное значение NaN в отличии от None относится к вещественным числам float и может использоваться в математических операциях, не вызывая ошибку из-за несовместимых типов.

Функция type() — определение типа данных в переменной:

variable = 'Строчка'

print(type(variable))

Функции преобразования типов данных переменных в Python:

* int() — превращает объект в целое число;
* float() — превращает объект в вещественное число;
* str() — превращает объект в строку.

Функция len() — подсчет количества элементов.

**Работа со строками**

Символы внутри строки считаются с 0 (нуля). С конца строки считаются с -1 (минус единицы).

Многострочные строки заключают в три одинарные или три двойные кавычки: ''' или """.

Специальные символы экранируют, помечая знаком \:

rule = 'Строки в Python помещают в одинарные кавычки, вот так: \'текст\''

Специальные символы:

* \n — перенос строки;
* \t — табуляция.

Можно умножать строки на числа. Python вернёт новую строку, в которой исходная строка повторена заданное количество раз. Таким образом генерируют длинные строки из заданных подстрок.

print(3 \* 'ha') или print('lol\n' \* 5)

Методы для работы со строками:

* .replace() — заменяет часть строки.

some\_string = some\_string.replace('leav', 'stay') *# заменяет подстроку leav строкой stay*

* .find() — ищет подстроку в строке, выдает индекс первого вхождения, если не находит, то выдает -1.

find = word.find('hello', 15, 25) *# Три аргумента: подстрока, индекс начала поиска, индекс завершения, увеличенный на 1.*

* .upper() — заменяет все буквы в строке на прописные (верхний регистр).
* .lower() — заменяет все буквы на строчные (нижний регистр).

f-строки используют для формирования длинных строк из символов и значений переменных. Чтобы применить это форматирование, перед строкой добавляют символ f. Имена переменных внутри строки указывают в фигурных скобках {}, при этом Python сам приведёт их к типу str. Можно сразу применять к переменным методы и шаблоны.

introduction = f'Hello, this is {name}, he is {age}, his height is {height}.'

introduction = f'Hello, this is {name.upper()}, next year he will be {age + 1}.'

Для форматирования процентов используется шаблон :.n% , где n — количество знаков после точки. Для вещественных чисел используется шаблон :.nf , где n *—* количество знаков после точки.

print(f'Доля русского языка в интернете составляет {russian\_web\_part:.2%}')

print(f'Доля русского языка в интернете составляет {russian\_web\_part:.1f}')

Можно использовать форматирование:

print(f'{elem:<45}', end='') # выравнивание каждого элемента по левому краю с фиксированной шириной строки 45

Метод .format() — формирует строки из символов и значений переменных аналогично f-строкам.

introduction = 'Hello, this is {0}, he is {1}, his height is {2}'.format(name, age, height)

introduction = 'Hello, this is {0}, next year he will be {1}'.format(name.upper(), age + 1)

print('Доля повторных покупок равна {0:.0%}'.format(repeated\_purchase\_part))

Метод .split() — разделяет строку на части по символу-разделителю и формирует из этих частей список. По умолчанию символ-разделитель – пробел, но можно задать любые разделители.

res = phrase.split() *# в res окажется список строк, а исходная строка phrase останется без изменений*

items = bands.split('---') *# передаём методу строку-разделитель*

Метод .join() — объединяет элементы списка в строку (обратно методу .split()). Вызывается через точку у строку-разделителя.

phrase = ' '.join(words) *# формируем строку из элементов списка words, соединённых пробелом*

info = '>'.join(stations) *# формирует строку с разделителем >*

Функции-предикаты — любые функции, которые возвращают True или False. Например, методы строк, которые проверяют символы в строке:

* .islower() — возвращает True, если в строке нет прописных букв;
* .isdigit() — возвращает True, если строка состоит только из числовых знаков;
* .isalpha() — возвращает True, если в строке только буквы, без пробелов и знаков препинания.

Если условие не соблюдено, эти функции вернут False.

print('hello'.islower())

print('777'.isdigit())

print('здесь есть пробелы, знаки препинания'.isalpha())

**Списки (list)**

Список — это структура данных. Чтобы создать список, его элементы перечисляют через запятую в квадратных скобках и присваивают переменной.

primer = [3, 2, 1, 0.5, 0.25, 'иголочка', 'ниточка', 'начали!']

Обращение к элементу списка происходит через его порядковый номер, или индекс. В Python нумерация индексов начинается с нуля. Индекс первого элемента списка — 0, второго — 1 и так далее.

Для обращения к определённому элементу списка пишут название этого списка и затем индекс элемента в квадратных скобках:

fruits\_and\_numbers = ['яблоко', 14, 'груша', 0.5, 'апельсин']

print(fruits\_and\_numbers[3])

Индексы могут быть отрицательными. Чтобы получить последний элемент списка, в квадратных скобках указывают -1. Индекс предпоследнего элемента — -2, третьего с конца элемента — -3 и так далее. Обратиться к одному и тому же элементу можно как по обычному, так и по отрицательному индексу:

russian\_cities = ['Москва', 'Санкт-Петербург', 'Томск', 'Екатеринбург', 'Владивосток', 'Смоленск', 'Барнаул', 'Сочи', 'Красноярск']

print(russian\_cities[6])

print(russian\_cities[-3])

Допускается складывать списки в списки:

first\_level\_dream = ['похищение', 'погоня', 'проекции']

second\_level\_dream = [528, 491]

third\_level\_dream = ['снег', 'хранилище', 'лимб']

dreams = [first\_level\_dream, second\_level\_dream, third\_level\_dream, 'волчок']

print (dreams)

Срезы позволяют получить не один, а сразу несколько идущих подряд элементов списка. Чтобы сделать срез, в квадратных скобках через знак : указывают начало и конец желаемого диапазона. Начало среза — это индекс первого входящего в желаемый диапазон элемента. Конец среза — индекс последнего элемента +1. Например, data[1:5], some\_list[7:9].

Для получения среза начала списка достаточно указать индекс элемента, следующего за концом среза, — он совпадает с количеством элементов, которое попадёт в срез. Срезы конца списка работают аналогично. Последний элемент среза совпадает с окончанием списка, поэтому достаточно указать индекс первого элемента. Удобнее делать это через отрицательный индекс.

prime\_numbers = [2, 3, 5, 7, 11, 13, 17]

print(prime\_numbers[:3])

print(prime\_numbers[-3:])

В срезах можно дополнительно указывать «шаг». Он определяет, следует ли пропускать какие-то элементы. Значение шага по умолчанию — 1, в этом случае никакие элементы не пропускаются. Его указывать необязательно. Если шаг равен 2, то в срез попадает каждый элемент со значением индекса больше текущего на 2, если 3 — на 3 и так далее. Отсчёт идёт с первого элемента — он тоже войдёт в срез.

Например, срез x[::2], где список x = [1, 2, 3, 4, 5, 6], выглядит так: [1, 3, 5].

Если указать отрицательный шаг, список развернётся и счёт пойдёт уже с последнего, а не с первого элемента списка. Например срез [::-1] будет включать все значения (иные границы не указаны), а сам список развернётся.

Результат среза — это новый список. Можно сделать «срез среза» или обратиться к определённому его элементу по индексу:

dragons = ['Валлийский зелёный', 'Венгерский хвосторог', 'Гебридский чёрный', 'Китайский огненный шар', 'Норвежский горбатый', 'Румынский длиннорог']

print(dragons[:][2:4])

print(dragons[:][3])

результат:

['Гебридский чёрный', 'Китайский огненный шар']

Китайский огненный шар

Списки можно складывать и умножать.

second\_row = ['Крёстный отец', 'США', 1972, 'драма, криминал', 175, 8.730]

extra\_info = [9.20, 'Фрэнсис Форд Коппола', 'Нино Рота']

updated\_row = second\_row + extra\_info

print(updated\_row)

Результат:

['Крёстный отец', 'США', 1972, 'драма, криминал', 175, 8.73, 9.2, 'Фрэнсис Форд Коппола', 'Нино Рота']

list = ['умножь', 'меня']

list = list \* 2

print(list)

Результат:

['умножь', 'меня', 'умножь', 'меня']

Работая с большими списками, лучше вызывать встроенные методы, так как они сработают быстрее.

Метод .append() — добавляет в конец списка один новый элемент. Этот элемент указывают в скобках. Добавляемый элемент может быть списком.

empty\_list = []

empty\_list.append(2)

print(empty\_list)

Метод .extend() — добавляет в конец списка элементы из другого списка.

second\_row.extend(['Фрэнсис Форд Коппола', 'Нино Рота']) # в конец списка second\_row добавляется имя режиссёра и имя композитора

Метод .insert() — вставляет один новый элемент в произвольное место списка. Вставляемый элемент может быть списком. В аргументах метода указывается индекс нового элемента и сам новый элемент.

dark\_knight\_movie.insert(2, 2008) # в список dark\_knight\_movie на место с индексом 2 вставляется 2008

Метод .pop() — удаляет один элемент из конца списка (первый с конца).

movies.pop()

Метод .sort() — сортирует список по возрастанию. С параметром reverse=True сортирует список по убыванию.

years.sort()

years.sort(reverse=True)

Метод .index() — возвращает индекс заданного элемента списка. Если заданного элемента в списке нет, то выдает ошибку.

ind = beatles.index('Пол Маккартни')

Функция sorted() — возвращает новый, отсортированный по возрастанию список аналогично методу .sort(). С параметром reverse=True возвращаемый список отсортирован по убыванию.

years\_sorted = sorted(years)

years\_sorted = sorted(years, reverse=True)

movies\_table\_sorted = sorted(movies\_table, key=lambda row: row[5]) *# Лямбда-функция принимает в качестве аргумента подсписок и выполняет инструкцию: получить из подсписка элемент с индексом* 5*. Этот элемент становится значением параметра* key *функции*sorted()*. В результате функция sorted() отсортирует подсписки по возрастанию элементов с индексом* 5.

Функция list() — преобразует определённый столбец датасета в список:

installs = list(data['installs'])

Конверсия — универсальный показатель, который применяют для оценки эффективности бизнеса.

Коэффициент конверсии, или просто конверсия, — это отношение целевых действий ко всем.

14.10.2022. Код про подсчет конверсии и дохода с использованием списков.

import pandas

data = pandas.read\_csv('app\_stats.csv')

installs = list(data['installs'])

payments = list(data['payments'])

print(payments[-8] / installs[-8] \* 600) # умножьте результат на 600

print(payments[-7] / installs[-7] \* 600) # умножьте результат на 600

print(payments[-6] / installs[-6] \* 600) # умножьте результат на 600

print(payments[-5] / installs[-5] \* 600) # умножьте результат на 600

print(payments[-4] / installs[-4] \* 600) # умножьте результат на 600

print(payments[-3] / installs[-3] \* 600) # умножьте результат на 600

print(payments[-2] / installs[-2] \* 600) # умножьте результат на 600

print(payments[-1] / installs[-1] \* 600) # умножьте результат на 600

**Множества (set)**

Множество — это структура данных. Чтобы создать множество, его элементы перечисляют через запятую в фигурных скобках и присваивают переменной. В отличии от других коллекций, все элементы множества должны быть уникальны, в множестве не может быть двух одинаковых элементов.

primer = {3, 2, 1, 0.5, 0.25, 'иголочка', 'ниточка', 'начали!'}

Операции с множествами:

Метод .add() — добавляет один новый элемент в существующее множество.

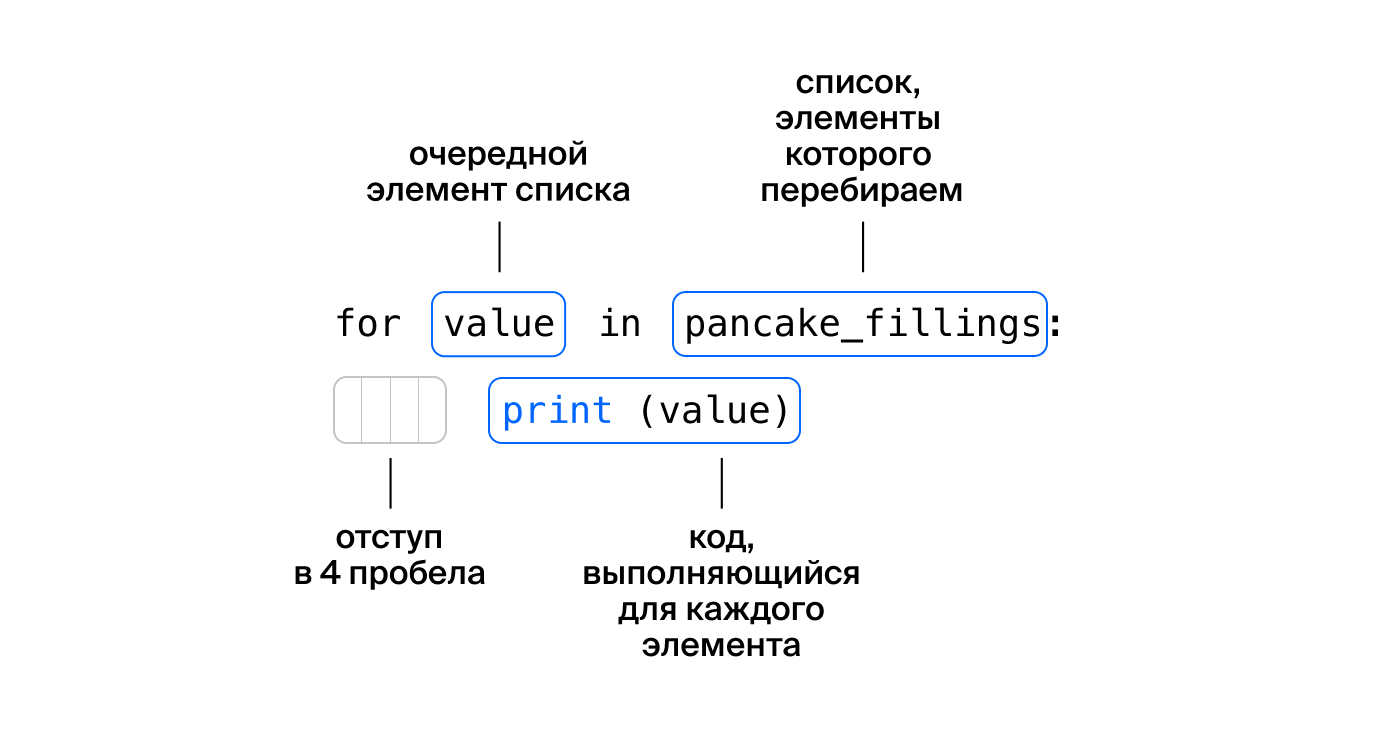
Метод .union() — объединение множеств. Создает новое множество их двух, например: playlist\_3 = playlist\_1.union(playlist\_2). В результате будет создано новое, третье множество (а оба исходных останутся такими, как были).

Метод .difference() — показывает различия двух множеств. Например: set\_1.difference(set\_2) вернёт новое множество, которое будет содержать только те элементы, которые присутствуют в set\_1, но отсутствуют в set\_2. Это похоже на «вычитание»: set\_1 - set\_2. При этом ни одно из исходных множеств не изменится.

Метод .intersection() — возвращает одинаковые элементы двух множеств.

**Циклы**

Цикл for — повторяет определённое действие для каждого элемента списка.



value — «переменная цикла» (итератор). Поочерёдно принимает значения каждого элемента списка — от первого до последнего. Имя переменной цикла может быть любым.

pancake\_fillings — название списка, по которому «проходится» цикл. Переменная value принимает значения именно из этого списка.

**Двоеточие.** Разделяет две части цикла.

print(value) — находится в «теле цикла». Тело цикла содержит команды, которые выполняются на каждой итерации (на каждом шаге повторения цикла). В нашем случае один шаг — это один элемент списка. Перед каждой командой в теле цикла ставится отступ в четыре пробела, которые означают, что идущие следом команды следует выполнить в цикле и отделяют их от остального кода.

Пример:

pancake\_fillings = ['творог', 'ягоды', 'грибы', 'капуста', 'мясо']

for value in pancake\_fillings:

print(value)

Циклы работают с любыми списками, в том числе со строками. Если запустить цикл по строке, он пройдёт по каждому символу этой строки — так же как проходит по каждому элементу списка:

client\_message = 'Робокот заболел, не хочет гулять'

for symbol in client\_message:

print(symbol)

Пример. Умножение:

numbers = [277, 389, 560, 741, 150, 40, 1598]

product = 1 # исходное значение (англ. product, «произведение»)

for x in numbers: # цикл, который поочерёдно умножит product на значения из списка

product = product \* x

print(product)

Можно написать цикл, где переменная цикла поочерёдно принимает значения из определённого списка, а можно обращаться к элементам списка по их индексам. Результаты таких кодов эквивалентны:

islands = ['Тенерифе', 'Мадейра', 'Бали', 'Бора-Бора', 'Галапагос', 'Куба']

for value in islands:

print(value)

islands = ['Тенерифе', 'Мадейра', 'Бали', 'Бора-Бора', 'Галапагос', 'Куба']

for index in [0, 1, 2, 3, 4, 5]:

print(islands[index])

Функция range() — генерирует последовательность чисел в заданном диапазоне. Существует три варианта вызова функции range():

1. Указать в её скобках одно значение. В результате формируется последовательность от 0 до этого значения, не включая его:

numbers = list(range(5))

print(numbers)

1. Указать два значения. Первое значение определяет начало последовательности, а второе — конец, но не входит в неё:

numbers = list(range(2, 9))

print(numbers)

1. Указать три значения. Первое значение определяет начало последовательности, второе — её окончание, а третье — размер шага между элементами:

numbers = list(range(3, 13, 3))

print(numbers)

Пример:

islands = ['Тенерифе', 'Мадейра', 'Бали', 'Бора-Бора', 'Галапагос', 'Куба']

for index in range(6):

print(islands[index])

Пример использования функции len() в цикле для подсчета количества элементов и повторений:

islands = ['Тенерифе', 'Мадейра', 'Бали', 'Бора-Бора', 'Галапагос', 'Куба']

for index in range(len(islands)):

print(islands[index])

\_ — нижним подчёркиванием принято называть переменные, которые не планируется далее применять в программе. Пример распечатки 100 смайлов, далее в программе переменная \_ будет не нужна:

for \_ in range(100):

print(':)')

Функция reversed() — переворачивает любую последовательность (список, диапазон) в обратный порядок, чтение последовательности начнётся с конца.

Пример:

for i in reversed(range(1, 13)):

print(i, 'бомм!')

print('C новым годом!')

В работе аналитика регулярно возникают задачи продублировать какое-нибудь значение n раз. Вот альтернативный способ создать список из ста нулей: [0] \* 100. Это работает и со строками: ['строка'] \* 100.

Пример:

Циклы работают не только со списками, но и со строками.

Объедините две строки: 'В еиоен' и 'ывлклпы'. Для этого объявите переменную final\_string с пустой строкой. Затем создайте цикл, в теле которого пропишите две инструкции: в первой поочерёдно добавляйте к значению final\_string символы из first\_string, во второй — символы из second\_string.

first\_string = 'В еиоен'

second\_string = 'ывлклпы'

final\_string = '' *# ваш код здесь*

for x in range(len(first\_string)):

final\_string = final\_string + first\_string[x]

final\_string = final\_string + second\_string[x]

print(final\_string) *# выводим результат на экран*

Результат:

Вы великолепны

Пример:

Рассчитать среднемесячный доход «Лютиков и других цветочков». Результат сохранить в переменной average\_income и вывести на экран.

payers = [75, 48, 65, 68, 74, 67, 71, 65, 90, 85, 79, 81, 87, 61, 73, 77, 76, 70, 79, 61, 63, 67, 89, 65, 73, 71, 78, 73, 61, 85, 62, 86, 80, 79, 68, 76, 87, 85, 71, 65, 75, 70, 61, 83, 65, 86, 61, 71] *# помесячное количество заплативших пользователей*

purchase\_amount = 5 *# цена приложения в долларах*

dohod = [] *# создание списка*

for x in range(len(payers)):

dohod.append(payers[x] \* purchase\_amount) *# цикл*

total = sum(dohod) *# сумма за все месяцы*

average\_income = total / len(payers) *# среднемесячный доход (деление суммы на количество элементов)*

print(average\_income) *# вывод среднего на экран*

Результат:

365.4166666666667

Функция enumerate() – похожа на range(), но возвращает индекс и значение одновременно.

a = ['100', '200', '300', '400', '500']

for idx, el in enumerate(a):

   print(f'index: {idx}, element: {el}')

Результат:

index: 0, element: 100

index: 1, element: 200

index: 2, element: 300

index: 3, element: 400

index: 4, element: 500

Функция zip() – похожа на range() и enumerate() , берет данные из нескольких списков. Списки должны быть одинаковой длинны, или возьмет только по длине самого короткого списка.

a = [1,2,3,4,5]

b = [7,8,9,0]

z = ‘asdasdas 7000 sdas’

for i\_a, i\_b, i\_z in zip(a, b, z):

   print(i\_a, i\_b, i\_z)

Результат:

1 7 a

2 8 s

3 9 d

4 0 a

Можно создавать списки определяя их наполнение через условия:

[[i\_a, i\_b] for i\_a, i\_b in zip(a, b)]

Результат:

[[1, 7], [2, 8], [3, 9], [4, 0]]

[i for i in range(11) if i % 2 != 0] *# список из нечетных чисел*

Результат:

[1, 3, 5, 7, 9]

[i if i % 2 != 0 else i \*\* 3 for i in range(11)]

Результат:

[0, 1, 8, 3, 64, 5, 216, 7, 512, 9, 1000]

[i\*\*2 for i in range(4)]

Результат:

[0, 1, 4, 9]

Цикл while

В цикле перед очередным шагом происходит проверка условия. Если это условие оказывается истинным, то выполняются инструкции из тела цикла. Цикл остановится, когда логическое выражение в условии вернёт False.

from random import randint

capacity = 400 *# грузоподъёмность лифта*

total\_weight = 0 *# переменная для суммарного веса*

while total\_weight < capacity: *# пока суммарный вес меньше грузоподъёмности лифта*

person\_weight = randint(30, 120) *# генерируется случайное целое число от 30 до 120*

total\_weight += person\_weight *# сгенерированный вес добавляется к суммарному весу*

print(f'А всё уже, всё уже! Лифт заполнен! Раньше надо было! Загруженность: {total\_weight}')

count = 0 *# количество найденных фильмов, снятых в США*

current = 0 *# счётчик, обозначающий индекс текущего списка*

filtered\_movies = [] *# в этот список будут отфильтрованы три первых фильма из США*

while count < 3: *# пока значение счётчика меньше трёх*

if movies\_table[current][1] == 'США': *# проверка первого элемента текущего списка с индексом current (страна производства текущего фильма)*

filtered\_movies.append(movies\_table[current]) *# если страна — США, то список добавляется в filtered\_movies*

count += 1 *# а значение найденных фильмов из США увеличивается на один*

current += 1 *# вне зависимости от страны индекс текущего списка увеличивается на один на каждом шаге цикла*

for movie in filtered\_movies:

for elem in movie:

print(f'{elem:<45}', end='')

print()

Пример с использованием break. Цикл будет наполнять список, пока не получит знечение 15:

a = [1, 2, 3, 4, 5]

cnt = 0

lst = []

while True:

   elem = a[cnt]\*5

   lst.append(elem)

   cnt += 1

   if elem == 15:

     break

Результат:

[5, 10, 15]

Пример с использованием continue. Пропустит значение 5.

lst = []

for i in range(11):

   if i == 5:

     continue

   lst.append(i)

Результат:

[0, 1, 2, 3, 4, 6, 7, 8, 9, 10]

**Условия и булева логика**

Операторы сравнения — сравнивают значения слева и справа от себя и возвращают результат в виде булева значения: True или False.

== значения равны;

< левое значение меньше правого;

> левое больше правого;

<= меньше или равно правому;

>= больше или равно;

!= значения не равны.

Пример:

floor = 1

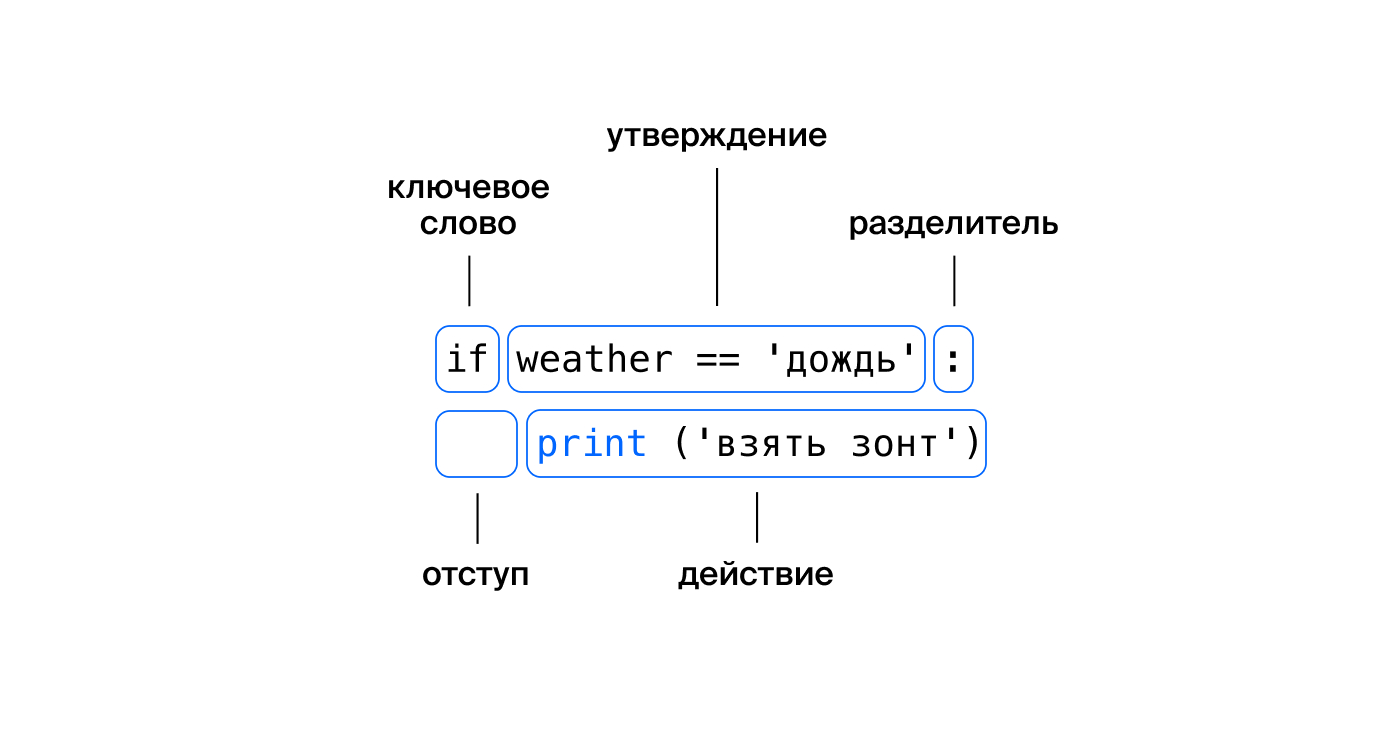
print(floor == 4)

В случае нескольких сравнений, они выполняются последовательно. Если все сравнения в цепочке истинны, то будет True. Если хоть одно ложно, будет False. Например, в результате цепочке сравнений print(1 <= 1 < 2 != 4), будет True.

True соответствует 1, а False — 0. Поэтому print(True < False) будет True. Аналогично, утверждения True == 1 и True > 0.5 будут истинными.

При сравнении строк «меньше» та строка, которая стояла бы в словаре раньше. Например, буква я «больше» буквы б, потому что больше её порядковый алфавитный номер. Сравнение происходит посимвольно, слева направо.

Условный оператор if-else «ветвит» программу — в зависимости от соответствия определённому условию выполняются разные части кода.



* **Ключевое слово if.** Предшествует утверждению.
* **Утверждение.** Проверка на истинность определённого утверждения. Для проверки используются операторы сравнения: ==, !=, >= и другие.
* **Двоеточие и отступ в четыре пробела.**
* **Действие.** Что сделает программа, если утверждение окажется истиной, то есть оператор сравнения вернёт True.
* **Ключевое слово else.** Предшествует блоку программы, который будет выполнен в ином случае. После else тоже ставится двоеточие.
* **Альтернативное действие.** Что сделает программа, если условное утверждение окажется ложным, то есть оператор сравнения вернёт False. Этот блок также начинают отступом в четыре пробела.

Пример:

if weather == 'дождь':

print('взять зонт')

else:

print('не брать зонт')

«Полноформатная» условная конструкция выглядит так: if-elif-else. В отличие от вложенных условий, оператор elif не требует дополнительных отступов, помимо стандартных четырёх пробелов.

Пример:

weather = 'дождь' *# напишите значение*

if weather == 'дождь':

item\_to\_take = 'зонт'

elif weather == 'солнце':

item\_to\_take = 'панама'

elif weather == 'снег':

item\_to\_take = 'шарф и шапка'

else:

item\_to\_take = 'ничего'

print(weather)

print(item\_to\_take)

**Логические операторы** and, or, not, in.

Оператор in можно применять и для проверки утверждений: он вернёт True, если определённый элемент есть в списке (в строке для строк). Оператор in возвращает булево значение True или False.

Пример in и not:

item = 'мармелад'

shopping\_list = ['хлеб', 'молоко', 'картофель', 'сыр', 'помидоры']

if not item in shopping\_list:

print('Купим в другой раз!')

Этот же код можно сделать более «человечным», поставив not рядом с in.

item = 'мармелад'

shopping\_list = ['хлеб', 'молоко', 'картофель', 'сыр', 'помидоры']

if item not in shopping\_list:

print('Купим в другой раз!')

Пример комбинирования логических операторов:

client\_message = 'Как включить функцию тихого мурчания? Робокот отличный, но будит по ночам.'

print(client\_message)

if ('Здравствуйте' in client\_message or

'Добрый день' in client\_message or

'Доброе утро' in client\_message or

'Добрый вечер' in client\_message or

'Привет' in client\_message):

print('Здравствуйте, с вами говорит искусственный интеллект: Агент Роберт I')

elif ('поломался' in client\_message or

'сломался' in client\_message or

'не работает' in client\_message or

'кусается' in client\_message or

'завис' in client\_message):

print('Очень жаль! С ними такое обычно не случается. Попробуйте перезагрузить.')

elif ('телефон' in client\_message or

'звонить' in client\_message):

print('Наш телефон: 8 (800) 111-11-11')

elif ('время работы' in client\_message or

'открыты' in client\_message):

print('С 9 утра до 20 вечера в любой день недели')

elif ('Спасибо' in client\_message or 'спасибо' in client\_message or

'До свидания' in client\_message or 'до свидания' in client\_message):

print('Спасибо, что обратились! До свидания!')

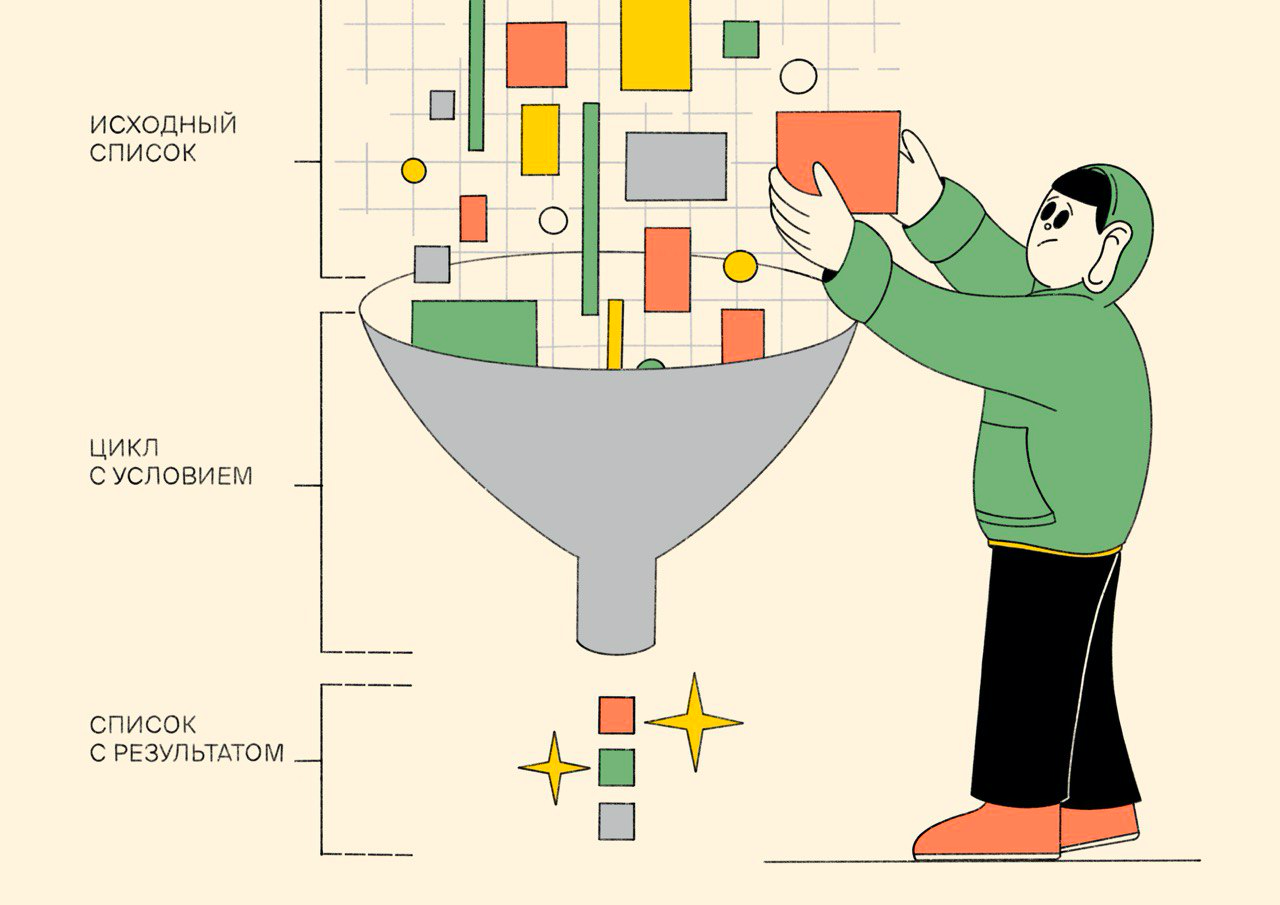
else:

print('Не могу помочь... Передам ваш вопрос специалисту.')

if 22 <= temp <= 26: заменяет if temp >= 22 and temp <= 26:

**Фильтрация списков**

Фильтрация объектов списка — это применение цикла for вместе с if. Цикл for проходит по всем объектам списка и применяет к ним различные действия в соответствии с прописанными в if условиями. Цикл сохраняет только те элементы, что при сравнении вернули True. Цикл принимает значения из одного списка, отсеивает ненужные элементы и пропускает остальные в новый список.



Пример:

*# исходный список*

raw\_numbers = [1, 10, 10000, -2, -0.5, 82, 3, 9999, 100]

*# финальный список с подходящими числами*

filtered\_numbers = []

for number in raw\_numbers: *# запускаем цикл*

if number > 1000: *# проверяем условие для элемента списка*

filtered\_numbers.append(number) *# сохраняем подходящий объект*

print('Результат фильтрации:', filtered\_numbers)

Результат:

Результат фильтрации: [10000, 9999]

Пример:

Добавьте в список mary\_julia\_songs элементы, которые есть в обоих списках.

julia\_songs = [145678, 297863, 966387, 374981, 746397, 197638]

mary\_songs = [576093, 197638, 736901, 297863, 374981, 871532]

mary\_julia\_songs = []

for song\_j in julia\_songs: # ваш код здесь

if song\_j in mary\_songs:

mary\_julia\_songs.append(song\_j)

# for song\_m in mary\_songs:

# if song\_m == song\_j:

# mary\_julia\_songs.append(song\_j)

print(mary\_julia\_songs)

Пример: С помощью фильтра найдите все объекты на первом этаже не дальше 15 км от центра города по ставке менее 1000 долларов.

prices = [1100, 999, 80, 80, 40000]

floors = [10, 1, 1, 4, 30]

distances = [10, 3, 15, 31, 37]

addresses = ['Россия, Москва, Берсеневская набережная, 6с1',

'Россия, Москва, Болотная набережная, 11с1',

'Россия, Москва, Романов переулок, 4',

'Россия, Москва, Старая Басманная улица, 20к1',

'Россия, Москва, Волгоградский проспект, 32к8']

prices\_filtered = []

floors\_filtered = []

distances\_filtered = []

addresses\_filtered = []

for index in range(len(prices)):

if floors[index] == 1 and distances[index] <= 15 and prices[index] < 1000: # напишите составное условие

prices\_filtered.append(prices[index])

floors\_filtered.append(floors[index]) # напишите ваш код здесь

distances\_filtered.append(distances[index])

addresses\_filtered.append(addresses[index])

print(prices\_filtered)

print(floors\_filtered)

print(distances\_filtered)

print(addresses\_filtered)

Результат:

[999, 80]

[1, 1]

[3, 15]

['Россия, Москва, Болотная набережная, 11с1', 'Россия, Москва, Романов переулок, 4']

Пример: Поиск в базе объектов, соответствующих заданным критериям и подсчет количества таких объектов.

import pandas

realty\_df = pandas.read\_csv('yandex\_realty\_data.csv')

filtered\_objects\_area = []

filtered\_objects\_price = []

filtered\_objects\_traffic = []

filtered\_objects\_address = []

for index in range(len(realty\_df['floor'])): # напишите цикл, который пройдёт по всем индексам датасета

if (realty\_df['floor'][index] == 1 and

realty\_df['area'][index] >= 40 and

realty\_df['price'][index] <= 190000 and

realty\_df['commercial\_type'][index] in ['FREE\_PURPOSE', 'RETAIL'] and

realty\_df['distance'][index] <= 6.7 and

realty\_df['already\_taken'][index] == 0 and

realty\_df['competitors'][index] <= 1):

filtered\_objects\_area.append(realty\_df['area'][index])

filtered\_objects\_price.append(realty\_df['price'][index])

filtered\_objects\_traffic.append(realty\_df['traffic'][index])

filtered\_objects\_address.append(realty\_df['address'][index])

print(len(filtered\_objects\_area)) # узнаем, сколько объявлений сохранил фильтр

Результат:

94

Пример: Продолжение предыдущего примера.

import pandas

realty\_df = pandas.read\_csv('yandex\_realty\_data.csv')

filtered\_objects\_area = []

filtered\_objects\_price = []

filtered\_objects\_traffic = []

filtered\_objects\_address = []

filtered\_objects\_profits = []

for index in range(len(realty\_df)):

if (realty\_df['floor'][index] == 1 and

realty\_df['area'][index] >= 40 and

realty\_df['price'][index] <= 190000 and

realty\_df['commercial\_type'][index] in ['FREE\_PURPOSE', 'RETAIL'] and

realty\_df['distance'][index] <= 6.7 and

realty\_df['already\_taken'][index] == 0 and

realty\_df['competitors'][index] <= 1):

filtered\_objects\_area.append(realty\_df['area'][index])

filtered\_objects\_price.append(realty\_df['price'][index])

filtered\_objects\_traffic.append(realty\_df['traffic'][index])

filtered\_objects\_address.append(realty\_df['address'][index])

filtered\_objects\_profits.append(realty\_df['traffic'][index] \*

18 \* 1/225 \* 0.1 \* 21000 \* 0.2 \* 30 - (realty\_df['price'][index] +

2 \* 50000 \* 1.43))

for index in range(len(filtered\_objects\_profits)):

if filtered\_objects\_profits[index] > 500000:

print(filtered\_objects\_price[index])

print(filtered\_objects\_traffic[index])

print(filtered\_objects\_address[index])

print(filtered\_objects\_profits[index])

print(filtered\_objects\_traffic[index] \* 18 \* 1/300 \* 0.05 \* 20000 \* 0.2 \* 30 - (filtered\_objects\_price[index] + 2 \* 50000 \* 1.43)) # добавьте пессимистичную оценку прибыльности

print('----------')

Результат:

173950.0

978

Россия, Москва, Берсеневская набережная, 6с1

668874.0000000001

35130.0

----------

161500.0

1097

Россия, Москва, Болотная набережная, 11с1

801276.0000000002

90420.0

----------

171000.0

820

Россия, Москва, Пятницкая улица, 41с1

512560.0

-18799.99999999994

----------

150000.0

837

Россия, Москва, Большой Кисельный переулок, 5

550696.0

8320.0

----------

150000.0

Пример: Окончательный вариант.

min\_required\_area = 40 # минимальная требуемая площадь

max\_affordable\_price = 190000 # максимально допустимая арендная ставка

third\_ring\_radius = 6.7 # максимальное расстояние от центра

open\_hours\_number = 18 # количество рабочих часов в сутки

traffic2visitors\_average\_ratio = 1 / 225 # средняя доля посетителей от числа прохожих

traffic2visitors\_pessimistic\_ratio = 1 / 300 # минимальная доля посетителей от числа прохожих

visitors2purchases\_average\_ratio = 0.1 # средняя доля покупателей от числа посетителей

visitors2purchases\_pessimistic\_ratio = 0.05 # минимальная доля покупателей от числа посетителей

average\_order\_value = 21000 # средняя стоимость покупки

average\_order\_value\_pessimistic = 20000 # минимальная стоимость покупки

trade\_margin = 0.2 # наценка

days\_in\_months = 30 # количество рабочих дней в месяц

# множитель для расчёта прибыльности в среднем сценарии

income\_multiplier\_average = (open\_hours\_number \*

traffic2visitors\_average\_ratio \*

visitors2purchases\_average\_ratio \*

average\_order\_value \*

trade\_margin \*

days\_in\_months)

# множитель для расчёта прибыльности в пессимистичном сценарии

income\_multiplier\_pessimistic = (open\_hours\_number \*

traffic2visitors\_pessimistic\_ratio \*

visitors2purchases\_pessimistic\_ratio \*

average\_order\_value\_pessimistic \*

trade\_margin \*

days\_in\_months)

number\_of\_employees = 2 # количество продавцов

employee\_salary = 50000 # зарплата продавца

tax\_multiplier = 1.43 # множитель для расчёта зарплаты с налогами

# зарплатная часть расходов

addition\_to\_expenses = number\_of\_employees \* employee\_salary \* tax\_multiplier

# минимальная ожидаемая прибыль

min\_expected\_profits = 500000

import pandas

realty\_df = pandas.read\_csv('yandex\_realty\_data.csv')

filtered\_objects\_area = []

filtered\_objects\_price = []

filtered\_objects\_traffic = []

filtered\_objects\_address = []

filtered\_objects\_profits = []

for index in range(len(realty\_df)):

if (realty\_df['floor'][index] == 1 and

realty\_df['area'][index] >= min\_required\_area and

realty\_df['price'][index] <= max\_affordable\_price and

realty\_df['commercial\_type'][index] in ['FREE\_PURPOSE', 'RETAIL'] and

realty\_df['distance'][index] <= third\_ring\_radius and

realty\_df['already\_taken'][index] == 0 and

realty\_df['competitors'][index] <= 1):

filtered\_objects\_area.append(realty\_df['area'][index])

filtered\_objects\_price.append(realty\_df['price'][index])

filtered\_objects\_traffic.append(realty\_df['traffic'][index])

filtered\_objects\_address.append(realty\_df['address'][index])

filtered\_objects\_profits.append(realty\_df['traffic'][index] \*

income\_multiplier\_average - (realty\_df['price'][index] +

addition\_to\_expenses))

for index in range(len(filtered\_objects\_profits)):

if filtered\_objects\_profits[index] > min\_expected\_profits:

print(filtered\_objects\_price[index])

print(filtered\_objects\_traffic[index])

print(filtered\_objects\_address[index])

print(filtered\_objects\_profits[index])

print(filtered\_objects\_traffic[index] \* income\_multiplier\_pessimistic -

(filtered\_objects\_price[index] + addition\_to\_expenses))

print('----------')

Тоже самое, только с использованием библиотек для исключения рутины:

import pandas

realty\_df = pandas.read\_csv('yandex\_realty\_data.csv')

realty\_df['expenses'] = realty\_df['price'] + 2 \* 50000 \* 1.43

realty\_df['incomes\_normal'] = realty\_df['traffic'] \* 18 \* 1/225 \* 0.1 \* 21000 \* 0.2 \* 30

realty\_df['incomes\_pessimistic'] = realty\_df['traffic'] \* 18 \* 1/300 \* 0.05 \* 20000 \* 0.2 \* 30

realty\_df['profits\_normal'] = realty\_df['incomes\_normal'] - realty\_df['expenses']

realty\_df['profits\_pessimistic'] = realty\_df['incomes\_pessimistic'] - realty\_df['expenses']

realty\_df\_filtered = realty\_df[(realty\_df['floor'] == 1) &

(realty\_df['area'] >= 40) &

(realty\_df['price'] <= 190000) &

realty\_df['commercial\_type'].isin(['FREE\_PURPOSE', 'RETAIL']) &

(realty\_df['distance'] <= 6.7) &

(realty\_df['already\_taken'] == 0) &

(realty\_df['competitors'] <= 1) &

(realty\_df['profits\_normal'] > 500000) &

(realty\_df['profits\_pessimistic'] > 0)]

print(realty\_df\_filtered)

Пример. Вложенные циклы для суммирования данных по площадям посевов и объемам урожаев по всем штатам за каждый год. И сравнение ошибок в первой и второй моделях.

import pandas

data = pandas.read\_csv('crops\_usa.csv')

# преобразуем столбцы датасета в списки

acres = list(data['Acres'])

production = list(data['Production'])

years = list(data['Year'])

acres\_usa = [] # общая площадь посевов за каждый год

production\_usa = [] # общий объём урожая за каждый год

for year in range(1980, 2020):

acres\_one\_year = []

production\_one\_year = []

for index in range(len(data)):

if years[index] == year:

acres\_one\_year.append(acres[index])

production\_one\_year.append(production[index])

acres\_usa.append(sum(acres\_one\_year))

production\_usa.append(sum(production\_one\_year))

yield\_usa = [] # общая урожайность за каждый год

for index in range(len(production\_usa)):

yield\_usa.append(production\_usa[index] / acres\_usa[index])

years\_numbers = list(range(1980, 2020))

error\_acres = [] # ошибки первой модели, предсказана площадь посевов

for index in range(1, len(production\_usa)):

error\_acres.append(production\_usa[index] - acres\_usa[index] \* yield\_usa[index - 1])

error\_yield = [] # ошибки второй модели, предсказана урожайность

for index in range(1, len(production\_usa)):

error\_yield.append(production\_usa[index] - acres\_usa[index - 1] \* yield\_usa[index])

error\_abs\_acres = [] # модули ошибок первой модели

for value in error\_acres:

error\_abs\_acres.append(abs(value))

print(sum(error\_abs\_acres) / len(error\_abs\_acres)) # MAE первой модели

error\_abs\_yield = [] # модули ошибок второй модели

for value in error\_yield:

error\_abs\_yield.append(abs(value))

print(sum(error\_abs\_yield) / len(error\_abs\_yield)) # MAE второй модели

Результат (средняя ошибка во второй стратегии почти в два раза меньше, значит лучше предсказывать по сторой стратегии, по урожайности):

209394344.66107476

110396388.66393325

Пример. Вложенные циклы:

import pandas

import seaborn

data = pandas.read\_csv('support\_data.csv')

segment = list(data['segment'])

robocats = list(data['robocats'])

# список сегментов, чтобы цикл мог пройти по ним

names = ['Segment 0', 'Segment 1', 'Segment 2']

# список, в который будем складывать средние значения

means = []

# внешний цикл по названиям сегментов

for name in names:

# код внутри - почти то же, что было раньше

cats = 0

counter = 0

# внутренний цикл

for index in range(len(data)):

# в этой строке заменили название сегмента на переменную цикла

if segment[index] == name:

cats += robocats[index]

counter += 1

means.append(cats / counter)

# код достаточно написать один раз

# цикл повторит его столько раз, сколько нужно

seaborn.barplot(x=means, y=names)

Пример тройного вложенного цикла:

segments\_old = ['Segment 0', 'Segment 1', 'Segment 2']

segments\_new = ['Потенциальные клиенты', 'Обычные клиенты', 'VIP-клиенты']

intervals = ['До внедрения роботов', 'После внедрения роботов']

# преобразуем столбцы в списки

intervals\_column = list(data['interval'])

segments\_column = list(data['segment'])

score\_column = list(data['score'])

# список для средних оценок

mean\_scores = []

# внешний цикл по трём сегментам

for segment in segments\_old:

interval\_scores = []

# вложенный цикл по двум периодам

for interval in intervals:

score = 0

counter = 0

# вложенный цикл по строкам

for index in range(len(data)):

if (segments\_column[index] == segment and

intervals\_column[index] == interval):

score += score\_column[index]

counter += 1

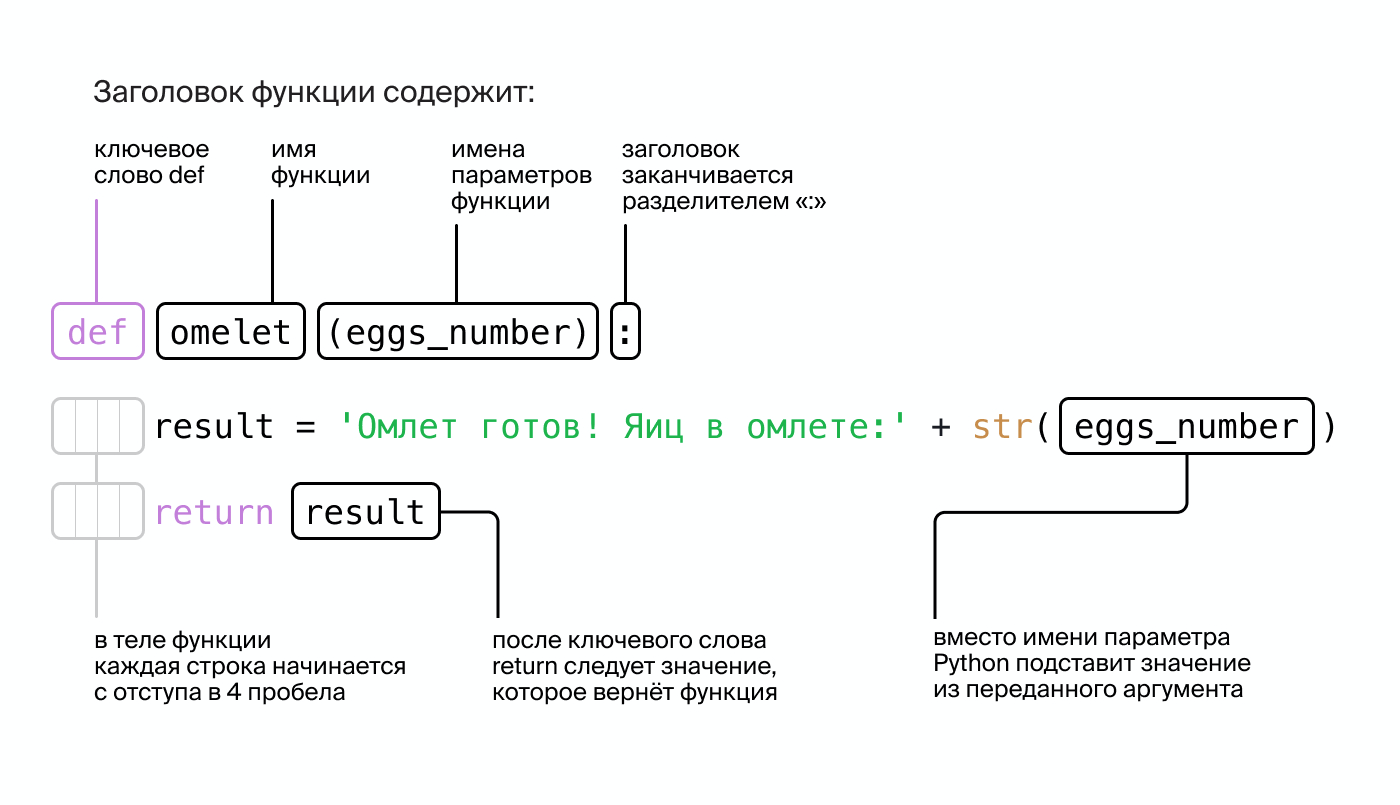
interval\_scores.append(score / counter)

mean\_scores.append(interval\_scores)

seaborn.heatmap(mean\_scores, xticklabels=intervals, yticklabels=segments\_new, annot=True, cmap='RdYlGn')

**Функции**

Функция — это набор инструкций, которому дают уникальное имя.



Хорошим стилем считается разделение двух видов кода:

* Бизнес-логики — инструкции, которые имеют отношение к задаче бизнеса (часть кода, которая непосредственно решает задачу бизнеса).
* Остальной код выполняет техническую функцию, «обслуживает» бизнес-логику.

def calculate\_total\_price(count, price):

total = count \* price

if total > 1000:

total -= total \* 0.05

return total

items\_list = [

[3, 100.0],

[2, 501.0],

[10, 40]

]

for item in items\_list:

print(calculate\_total\_price(item[0], item[1]))

При объявлении функции можно задать для любого параметра значение по умолчанию, оператором =. Например, def omelet(eggs\_number=2):

В таких случаях вокруг = не ставят пробелы: это правило стиля. Оно помогает быстро отличить значения по умолчанию от присваивания значений переменным.

Параметры со значениями по умолчанию называют необязательными. Если не указать его значение при вызове функции, он получит значение из объявления функции. Необязательные параметры в объявлении функции должны располагаться после обязательных, иначе вызов функции вызовет ошибку.

Передать аргументы функции можно двумя способами:

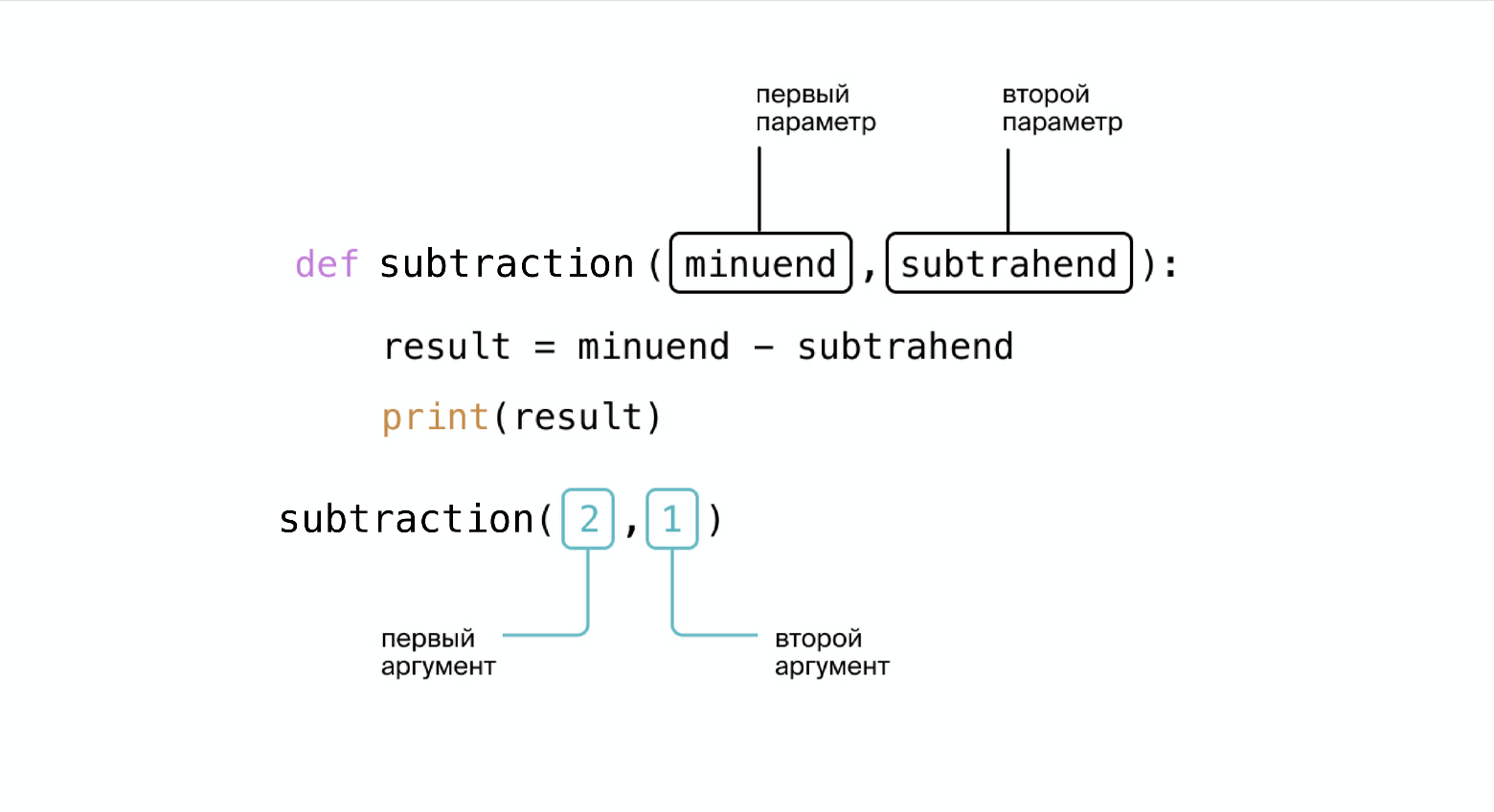
1. Позиционные аргументы. Позволяют лаконично задать обязательные параметры. Необходимо указать аргументы в определённом порядке.

*# Первый аргумент — список списков с данными,*

*# второй — продолжительность фильма*

movies\_table\_filtered = filter\_by\_timing(movies\_table, 170)

Аргументы в вызове функции subtraction(2, 1) называют позиционными. Они присваиваются параметрам согласно своей позиции, порядку следования:



Результат функции изменится, если поменять порядок параметров или аргументов.

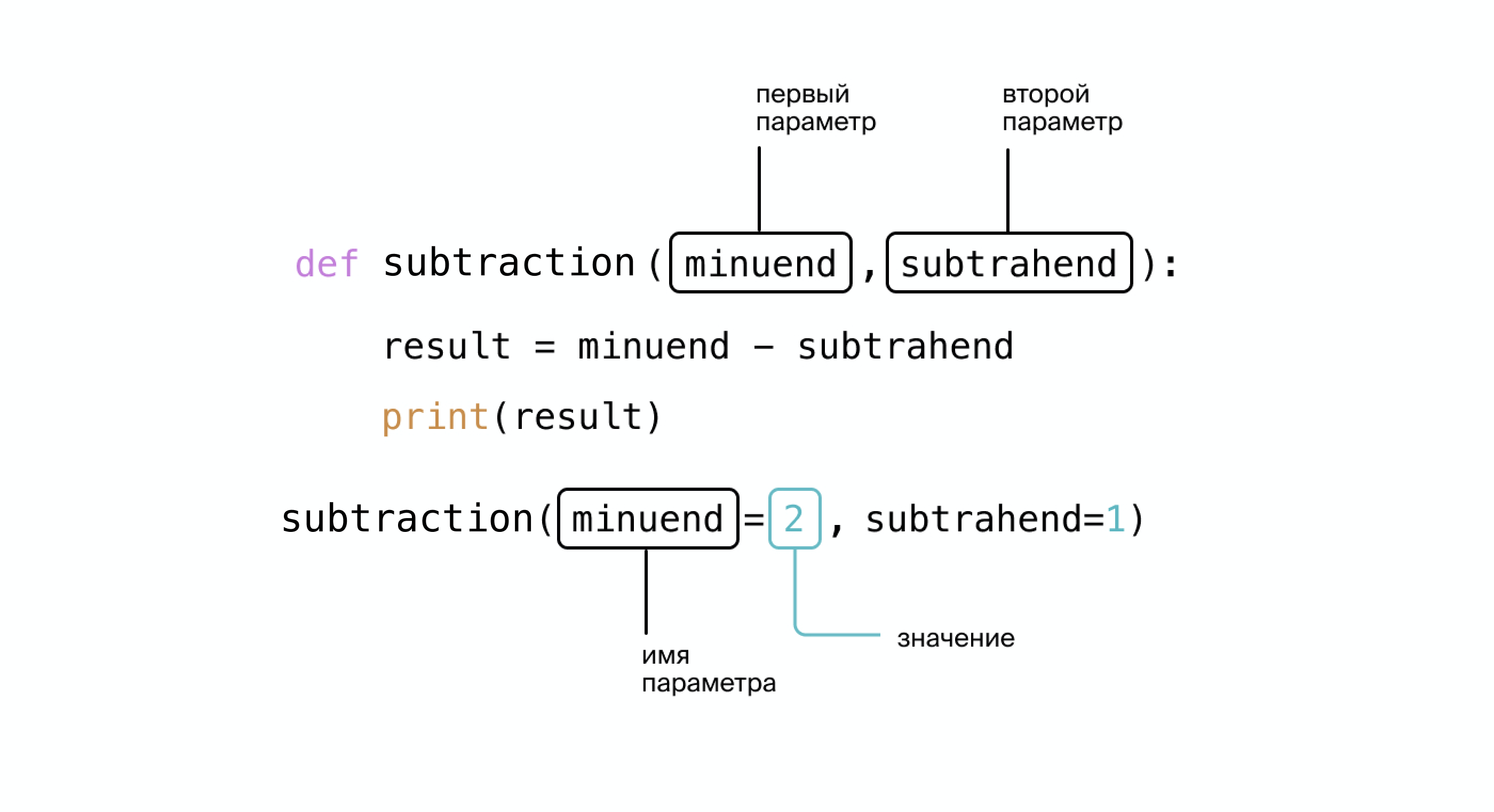
1. Именованные аргументы. Необходимо указать имя параметров, которым следует присвоить значение. Через именованные аргументы удобно изменить значение по умолчанию любого из необязательных параметров.

*# Передаём параметру data список списков movies\_table,*

*# а параметру target\_duration — значение 170*

movies\_table\_filtered = filter\_by\_timing(data=movies\_table, target\_duration=170)

По вызову функции видно, какое значение получит каждый параметр.



Передавать именованные аргументы можно в любом порядке. Поведение функции от этого не изменится.

При вызове функции можно сочетать два вида аргументов. При этом позиционные аргументы указывают первыми, за ними следуют именованные аргументы.

Переменные в теле функции называют локальными (получить их значение можно только внутри функции). Этим они отличаются от переменных глобальных (которые объявлены в коде за пределами функций). Если обратиться к локальным переменным извне функции, то произойдёт ошибка.

Технически функция всегда возвращает одно значение, но можно перечислить несколько переменных после return, а результаты работы функции присвоить нескольким глобальным переменным.

return square, perimeter

square, perimeter = find\_square\_and\_perim(7, 3)

print(f'Площадь прямоугольника равна {square}, периметр прямоугольника равен {perimeter}.')

Если в функции не будет ключевого слова return, то такая функция вернёт None в качестве результата.

Функции-предикаты — это функции, которые возвращают True или False.

def is\_age\_appropriate\_two(age, threshold):

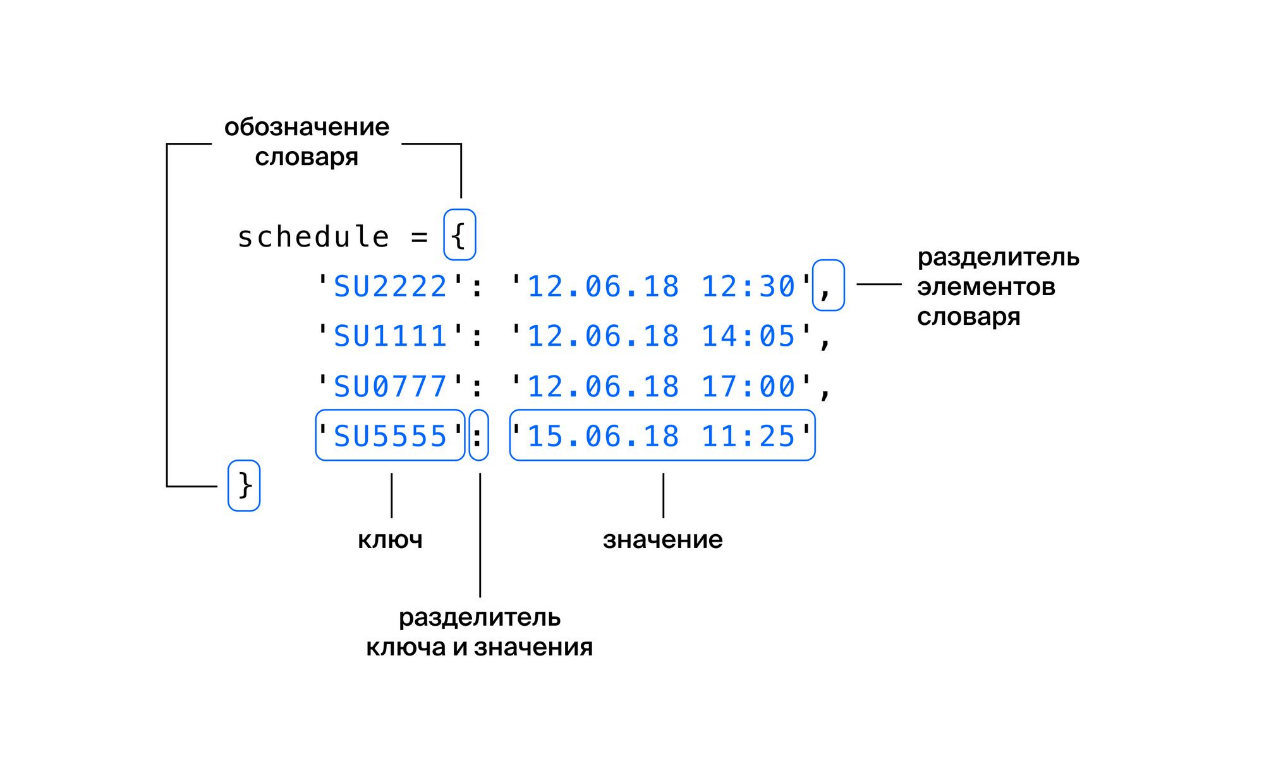
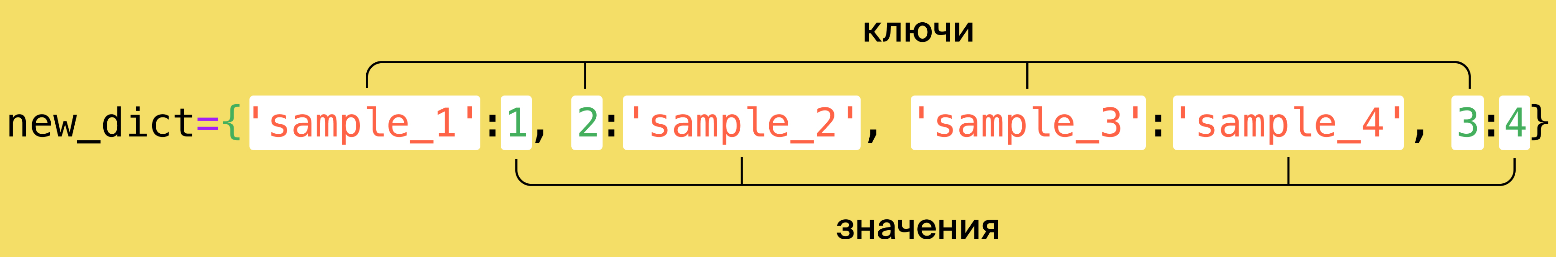
return age >= threshold *# вернёт True или False*

print(is\_age\_appropriate\_two(17, 12))

**Словари**

Словарь — набор пар «ключ:значение»:

* ключами могут быть строки (str) и числа (int и float);
* значениями — любые типы данных.

Значения из словаря получают по ключу двумя способами:

1. Небезопасный способ. Конкретный ключ указывается в квадратных скобках после имени словаря.

value = dictionary\_name['key']

В этом случае если ключа в словаре нет, ошибка прервёт работу программы.

1. Безопасный способ.
   1. Чтобы избежать ошибки, небезопасный способ сочетают с конструкцией try...except:

try:

value = dictionary\_name['key'] *# ищем ключ key в словаре*

print(value) *# если ключ найден — выводим значение*

except:

print('Ключ не найден!') *# иначе выводим сообщение*

* 1. Метод .get()

Конкретный ключ передается в скобках. Если метод не найдёт ключ в словаре, он вернёт значение по умолчанию — None:

value = dictionary\_name.get('key')

Значение по умолчанию (None) можно изменить. Для этого передается новое значение вторым аргументом:

value = dictionary\_name.get('key', 'Ключ не найден!')

Безопасный способ применяется для больших объёмах данных, когда сложно проверить, что нужный ключ есть в словаре.

Чтобы добавить новую пару в словарь, указывается новый ключ в квадратных скобках после словаря и присваивается новое значение:

dictionary['new\_key'] = 'новое значение'

Из словарей и списков создают сложные структуры данных. К значениям в списке словаря обращаются по двум «координатам»: индексу списка и имени ключа. Список словарей напоминает таблицы pandas, где к данным обращаются по номеру строки и имени колонки. Циклы могут обойти список словарей как обычный список. Например, так можно отфильтровать данные или посчитать сумму значений любой «колонки»:

movies\_table = [

{'movie\_name':'Побег из Шоушенка', 'country':'США', 'genre':'драма', 'year':1994, 'duration':142, 'rating':9.111},

{'movie\_name':'Крёстный отец', 'country':'США', 'genre':'драма, криминал', 'year':1972, 'duration':175, 'rating':8.730},

{'movie\_name':'Тёмный рыцарь', 'country':'США', 'genre':'фантастика, боевик, триллер', 'year':2008, 'duration':152, 'rating':8.499}

]

total\_duration = 0 *# присваиваем переменной с общей длительностью стартовое значение*

for movie in movies\_table: *# перебираем каждый словарь в списке*

total\_duration += movie['duration'] *# добавляем к переменной длительность фильма*

print(total\_duration)

Метод .values(). Извлечение значений из словаря.

Метод .keys(). Извлечение ключей из словаря.

Метод .items(). Извлечение ключей и значений из словаря.

Превратит словарь в специальный набор пар из ключей и значений. Этот набор отличается тем, что в нём цикл for пройдёт как по ключам, так и по значениям, соответственно, у цикла будет два итератора:

* person — для ключей;
* points — для значений.

game\_scores = {

'Иванов': [23, 35, 70, 45],

'Петров': [38, 72, 65, 80],

'Сидоров': [30, 35, 90, 73],

'Антонов': [45, 20, 95, 80]}

*# Цикл переберёт пары в словаре —*

*# в person попадает ключ (строка с фамилией игрока),*

*# в points — значение (список с набранными очками):*

for person, points in game\_scores.items():

total\_points = sum(points) *# функция sum() вернёт сумму элементов списка points*

print(f'{person} - {total\_points}') *# выводим сумму игрока person*

Если при объявлении цикла после названия словаря не указать метод, то цикл будет перебирать ключи словаря (аналогично методу .keys()):

for keys in game\_scores:

Метод .update(). Объединяет два словаря (добавляет в один словарь элементы другого).

Функция dumps() из библиотеки json. Готовит список словаря к печати, добавляя отступы, которые разделяют две «оси координат» в списке словарей: индексы списка и ключи словаря.

from json import dumps *# подключение dumps() для красивого вывода словаря*

movies\_table = [

{'movie\_name':'Побег из Шоушенка', 'country':'США', 'genre':'драма', 'year':1994, 'duration':142, 'rating':9.111},

{'movie\_name':'Крёстный отец', 'country':'США', 'genre':'драма, криминал', 'year':1972, 'duration':175, 'rating':8.730},

{'movie\_name':'Тёмный рыцарь', 'country':'США', 'genre':'фантастика, боевик, триллер', 'year':2008, 'duration':152, 'rating':8.499}

]

print(dumps(movies\_table, indent=4, ensure\_ascii=False))

Функцией dumps() управляют два параметра:

* indent=4 — настроит отступ;
* ensure\_ascii=False — обработает кириллические символы.

Функция set(). Преобразует в множество. Если аргумент – словарь, то в множество преобразуются ключи словаря.

Функция list(). Преобразует в список. Если аргумент – словарь, то в список преобразуются ключи словаря.

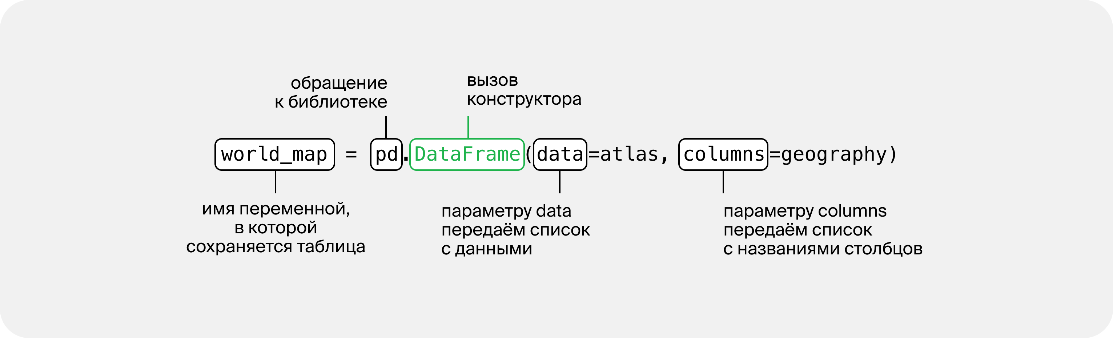
Списки, множества и словари в Python называются **коллекциями**. Строки – это тоже коллекции.

**Работа с датафреймами**

import pandas as pd – подключение библиотеки pandas и обозначение её как pd.

Датафрейм (DataFrame) — это двумерная структура данных pandas, где у каждого элемента две координаты: строка и столбец.

Метод .DataFrame() – конструктор из библиотеки pandas для создания нового объекта (датафрейм, таблица). Принимает два аргумента: список с данными, названия столбцов таблицы.



import pandas as pd

*# подготавливаем данные и названия столбцов*

atlas = [

['Франция','Париж'],

['Россия','Москва'],

['Китай','Пекин'],

['Мексика','Мехико'],

['Египет','Каир'],

]

geography = ['country', 'capital']

*# создаём таблицу*

world\_map = pd.DataFrame(data=atlas , columns=geography) *# создаём таблицу и сохраняем её в переменную world\_map*

print(world\_map) *# выводим таблицу на экран*

Результат:

country capital

0 Франция Париж

1 Россия Москва

2 Китай Пекин

3 Мексика Мехико

4 Египет Каир

Метод .Series() –создание нового объекта Series. Принимает два аргумента: data – значения строки и index – названия столбцов.

row\_values = [24, 1] *#значения*

row\_columns = ['age', 'unemployed'] *#названия столбцов*

row = pd.Series(data=row\_values, index=row\_columns)

Метод .read\_csv() – загружает данные из файла .csv в таблицу/датасет. Есть аргументы sep , отвечающий за разделитель колонок, и decimal , отвечающий за разделитель дробей. Если разделитель \, то указывают \\, чтобы не совпадало со служебным символом Python.

df = pd.read\_csv('/datasets/music.csv') *# аргумент — путь к файлу*

file = pd.read\_csv('file.csv', sep=';', decimal=',')

Метод .read\_excel() – загружает данные из файла Excel в датасет. Первый параметр – путь и имя файла. Второй параметр sheet\_name – название листа.

df = pd.read\_excel('/datasets/Экселевский файл.xlsx', sheet\_name='Самый первый лист')

Со столбцами датасета можно выполнять арифметические действия, при этом в датасете появляется новый столбец:

purchases['total'] = purchases['first'] + purchases['repeated'] *# после сложения в датасете появляется новый столбец total*

purchases['repeated\_share'] = purchases['repeated'] / purchases['total'] *# после деления в датасете появляется новый столбец repeated\_share*

С датасетами можно выполнять арифметические действия, в результате появляется новый датасет:

visits = logs.groupby('source')['user\_id'].count() *# количество визитов*

purchase = logs.groupby('source')['purchase'].sum() *# количество покупок*

conversion = purchase / visits *# конверсия – результат деления – в новом датасете*

Метод .count() – подсчет количества значений.

Метод .sum() – подсчет суммы значений.

Метод .head() – вывод первых строк таблицы. По умолчанию – 5. Можно указать количество строк.

print(data\_df.head(10))

Метод .tail() – вывод последних строк таблицы. По умолчанию – 5. Можно указать количество строк.

print(data\_df.tail(10))

Датафрейм хранит не только сами данные, но и общую информацию об этих данных, которую можно посмотреть через вызов атрибутов датасета. Синтаксис атрибутов напоминает похож на синтаксис методов, но при вызове атрибута, в отличии от метода, скобки после имени атрибута не ставятся.

Атрибуты датасета:

* dtypes — показывает тип данных значений в столбцах датасета print(df.dtypes);

Типы данных библиотеки pandas и их соответствие типам данных Python:

| **\_** | **Тип в Python** | **Тип в Pandas** |
| --- | --- | --- |
| Строка | str | object |
| Целые числа | int | int64 |
| Вещественные числа | float | float64 |
| Логический тип данных | bool | bool |

Тип object в pandas может соответствовать не только строке, но и данным, которые не подходят ни под один другой тип.

* columns — показывает список названий столбцов датасета print(df.columns);
* shape — показывает данные о размерах таблицы. Представляет из себя кортеж – особый тип списков. Первое значение в кортеже — это количество строк, второе — столбцов. print(df.shape)

Кортежи похожи на списки, но:

* списки пишут в квадратных скобках, а кортежи – в круглых;
* элементы списка можно менять, добавлять новые или удалять, а кортеж – это неизменяемая структура.

Получить отдельное значение из кортежа можно так же, как из списка:

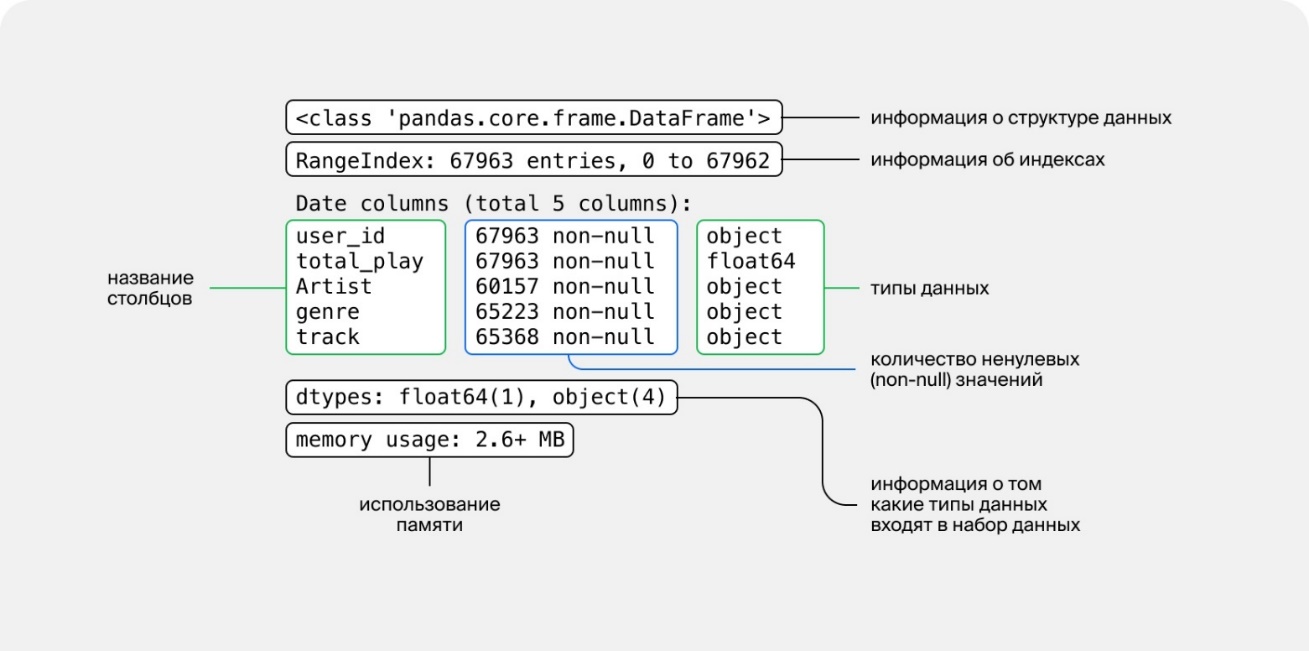
*# получаем количество строк из кортежа*

rows\_number = df.shape[0]

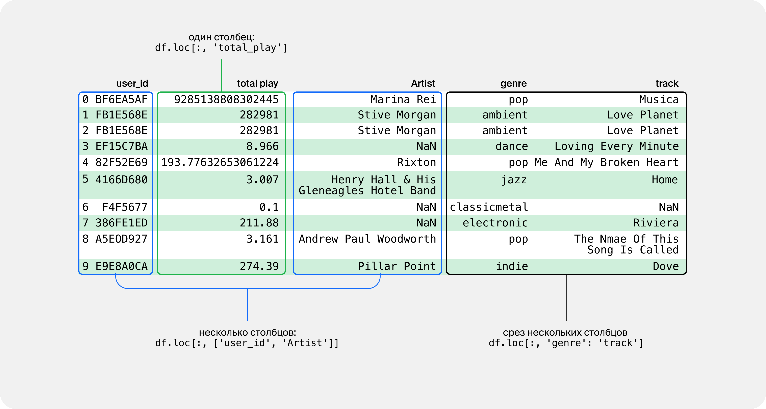
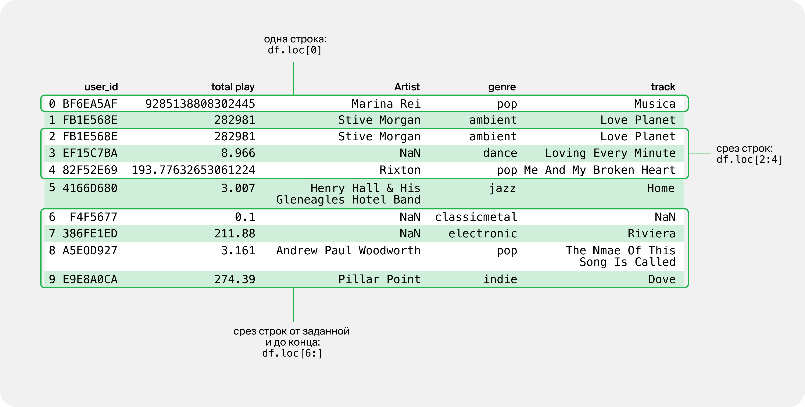
*# получаем количество столбцов из кортежа*

columns\_number = df.shape[1]

Метод .info() – получение всех атрибутов датасета сразу. df.info()



Атрибут .loc[строка, столбец] – обращение к ячейкам датасета по их координатам (индексация). result = df.loc[4, 'genre']



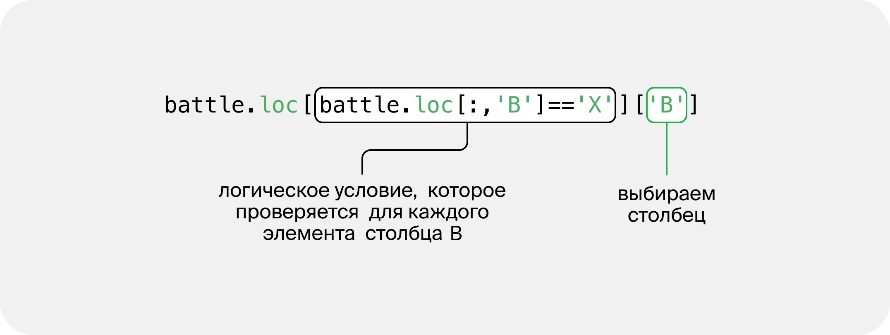
На практике применяют сокращённую форму записи для индексации. В ней не вызывают атрибут .loc, а сразу пишут индексы в квадратных скобках:

| **\_** | **Полная запись** | **Сокращённая запись** |
| --- | --- | --- |
| Один столбец | .loc[:, 'genre'] | df['genre'] |
| Несколько столбцов | .loc[:, ['genre', 'Artist']] | df[['genre', 'Artist']] |
| Все строки, начиная с заданной | .loc[1:] | df[1:] |
| Все строки до заданной | .loc[:3] (включая 3) | df[:3] (не включая 3) |
| Несколько строк подряд (срез) | .loc[2:5] (включая 5 | df[2:5] (не включая 5) |
| Одна ячейка | .loc[7, 'genre'] | - |
| Одна строка | .loc[1] | - |
| Несколько столбцов подряд (срез) | .loc[:, 'total\_play': 'genre'] | - |

Логическая индексация. Использует условия, чтобы отфильтровать данные в датасете.

battle.loc[battle.loc[:,'В'] == 'X'] *# отфильтровывает из датасета battle все строки, где в столбце B встречается ‘X’*

Отфильтровывание ячеек в B со значением ‘X’:



Подсчет количества ячеек в B со значением ‘X’:

print(battle.loc[battle.loc[:,'В'] == 'X']['В'].count()) *# полная запись*

print(battle[battle['В'] == 'X']['В'].count()) *# сокращенная запись*

Пример логической индексации\_1:

import pandas as pd

df = pd.read\_csv('game\_board.csv')

print(board\_df[board\_df['H'] == 1]['H']) *# Выведите на экран корабль 4. В решении примените сокращённую запись логической индексации. Для строк задайте условие: значение в колонке H равно 1. Затем выберите столбец H.*

Пример логической индексации\_2:

import pandas as pd

​df = pd.read\_csv('music\_log.csv')

*# строки датафрейма, в которых жанр — джаз (jazz)*

jazz\_df = df[df['genre'] == 'jazz']

print(jazz\_df, '\n')

*# строки датафрейма, в которых время прослушивания total\_play больше 90*

high\_total\_play\_df = df[df['total play'] > 90]

print(high\_total\_play\_df, '\n')

*# строки датафрейма, в которых время прослушивания total\_play меньше или равно 10*

low\_total\_play\_df = df[df['total play'] <= 10]

print(low\_total\_play\_df)

Пример логической индексации\_3:

import pandas as pd

df = pd.read\_csv('music\_log.csv')

*# выбираем строки, в которых жанр — джаз и время воспроизведения находится в диапазоне от 80 до 130*

df = df[df['total play'] >= 80]

df = df[df['total play'] <= 130]

df = df[df['genre'] == 'jazz']

print(df)

Series — это отдельный тип объекта - каждая колонка таблицы. Из таблицы можно извлечь отдельные Series, а из Series — собрать новую таблицу.

Series — это одномерная структура. Для обращения к данным в Series достаточно только индекса. Также у Series есть имя, которое становится названием колонки, когда из нескольких колонок собирают таблицу. Кроме имени, у Series есть длина — количество ячеек.

Индексация в Series аналогична индексации в датасетах, но без второй оси координат:

*# Получаем Series из датафрейма*

artist = df['Artist']

*# Получаем ячейку из Series по единственной координате*

print(artist[0])

Полная и сокращённая запись в Series:

| **\_** | **Полная запись** | **Сокращённая запись** |
| --- | --- | --- |
| Один элемент | total\_play.loc[7] | total\_play[7] |
| Несколько элементов | total\_play.loc[[5, 7, 10]] | total\_play[[5, 7, 10]] |
| Несколько элементов подряд (срез) | total\_play.loc[5:10] (включая 10) | total\_play[5:10] (не включая 10) |
| Все элементы, начиная с заданного | total\_play.loc[1:] | total\_play[1:] |
| Все элементы до заданного | total\_play.loc[:3] (включая 3) | total\_play[:3] (не включая 3) |

Итоговый пример:

import pandas as pd

df = pd.read\_csv('music\_log.csv')

rock = df[df['genre'] == 'rock'] *# из df получить таблицу только с жанром rock и сохраните её в переменной rock*

rock\_time = rock['total play'] *# выделите время прослушивания роковых композиций в особую структуру данных – сохраните колонку total play таблицы rock в переменной rock\_time*

rock\_haters = rock\_time[rock\_time <= 5].count() *# посчитайте треки в rock\_time, которые слушали пять секунд или менее, результат сохраните в переменной rock\_haters*

print("Количество пропущенных треков жанра рок равно", rock\_haters)

pop = df[df['genre'] == 'pop'] *# выберите в таблице df только строки с жанром 'pop' и сохраните эту новую таблицу в переменной pop*

pop\_time = pop['total play'] *# Из таблицы pop извлеките столбец total play и сохраните его в переменную pop\_time*

pop\_haters = pop\_time[pop\_time <= 5].count() *# посчитайте треки в pop\_time, которые слушали пять секунд или менее, результат сохраните в переменной pop\_haters*

print("Количество пропущенных треков жанра поп равно", pop\_haters)

rock\_skip = rock\_haters / rock\_time.count() *# rock\_haters разделите на общее число треков в таблице rock, результат сохраните в переменную rock\_skip*

pop\_skip = pop\_haters / pop\_time.count() *# pop\_haters разделите на число треков в pop, результат сохраните в pop\_skip*

print('Доля пропущенных композиций жанра рок равна:', rock\_skip)

print('Доля пропущенных композиций жанра поп равна:', pop\_skip)

Результат:

Количество пропущенных треков жанра рок равно 2272

Количество пропущенных треков жанра поп равно 2974

Доля пропущенных композиций жанра рок равна: 0.33357803553075904

Доля пропущенных композиций жанра поп равна: 0.345854169089429

В атрибут .loc можно передавать список:

rows = [False, True, False, False, False, False, False, False]

print(data.loc[rows])

Результат:

Имя Роль Новая функция Эксперимент Статья

1 Петя аналитик + - -

Также в атрибут .loc можно передавать Series:

rows = data['Роль'] == 'разработчик'

print(rows)

print(data.loc[rows])

Результат:

0 True

1 False

2 True

3 False

4 False

5 False

6 True

7 False

Name: Роль, dtype: bool

Имя Роль Новая функция Эксперимент Статья

0 Маша разработчик + NaN NaN

2 Вася разработчик NaN + +

6 Настя разработчик + + -

Можно одновременно с фильтрацией по одному столбцу, извлекать значения другого столбца:

print(data.loc[data['Новая функция'] == '+', 'Имя'])

Результат:

0 Маша

1 Петя

3 Катя

6 Настя

Name: Имя, dtype: object

С помощью можно осуществлять замену данных в датасете:

data.loc[data['Эксперимент'] == '+', 'Роль'] = 'экспериментатор' *# всем у кого «+» в поле «Эксперимент», «Роль» изменена на «эксперементатор»*

print(data)

Результат:

Имя Роль Новая функция Эксперимент Статья

0 Маша разработчик + NaN NaN

1 Петя аналитик + - -

2 Вася экспериментатор NaN + +

3 Катя менеджер + - +

4 Ваня экспериментатор NaN + NaN

5 Дима экспериментатор NaN + NaN

6 Настя экспериментатор + + -

7 Лена копирайтер NaN NaN +

Логические операторы and, or, not в Pandas заменяют на знаки & , | и ~ соответственно. И каждое сравнение окружают скобками, например: (x > 0) & (x < 10).

print(data.loc[(data['Новая функция'] == '+') & (data['Роль'] == 'разработчик')]) *# выборка всех разработчиков, занимающихся Новой функцией*

data.loc[(data['Роль'] == 'разработчик') & (data['Новая функция'] == '+'), 'Новая функция'] = '-' *# все разработчики, участвующие в проекте Новая функция снимаются с проекта*

Ещё примеры с применением логических операторов:

(df['Is\_Direct']) & (df['Price'] < 21000)

(df['Has\_luggage']) | (df['Price'] < 20000)

~((df['Is\_Direct']) | (df['Has\_luggage']))

~(df['Has\_luggage'])

Применение сокращенной записи атрибута .loc ограничено следующими правилами:

* При передаче только условия можно использовать как полную, так и сокращенную запись. Логические операторы при этом также используются.

*# полная запись:*

print(data.loc[data['Новая функция'] == '+'])

*# сокращённая запись:*

print(data[data['Новая функция'] == '+'])

* Если необходимо указать условие и столбец или изменить значения в таблице, то можно использовать только полную запись с .loc.

*# указание условия и столбца:*

print(data.loc[data['Эксперимент'] == '+', 'Имя'])

*# внесение изменений в таблицу:*

rows = (data['Новая функция'] == '+') & (data['Роль'] == 'разработчик')

data.loc[rows, "Роль"] = "улучшатель"

Метод .iloc() – тоже самое, что .loc(), но обращается не по названию индекса, а по его порядковому номеру. Также есть аналогичный метод .at().

Метод .isin() – логически проверяет на соответствие указанным значениям, применяется для фильтрации.

df[df['Date\_From'].isin(['04.07.2019', '05.07.2019'])] *# находим элементы столбца Date\_From, равные 4 или 5 июля*

Метод .query() – применяется для формирования срезов из датафрейма, в скобках передается одно или насколько условий с учетом следующих правил:

* Применяются операции сравнения: !=, >, >=, <, <=.
* Проверяют, входят ли конкретные значения в список, конструкцией in: Date\_To in ["07.07.19", "09.07.2019"]. Отсутствие в списке значений проверяют конструкцией not in: Date\_To not in ["07.07.19", "09.07.2019"].
* Логические операторы указываются в виде or, and, not. Без скобок операции выполняются в следующем порядке: сначала not, затем and, затем or.
* Используют двойные кавычки, чтобы различать одинарные кавычки, оформляющие строку, и кавычки для элемента.
* Используют математические операции (+, -, \*, /), вызывают методы (например, .max(), .min()), а также внешние переменные, которые помечают знаком @.

print(df.query('To == "Barcelona"'))

print(df.query('Is\_Direct == True or Has\_luggage == True'))

df.query('Travel\_time\_from < 2 \* Travel\_time\_to ')

df.query('Travel\_time\_from < Travel\_time\_to.mean()')

maximum\_price = 20000

df.query('Price <= @maximum\_price')

Метод .rename() – переименование столбцов датафрейма. Имеет параметр columns – это словарь, в котором ключи — старые названия столбцов, а значения — новые. Переименовать можно как все столбцы, так и только отдельные.

df = df.rename(columns={'Track ID': 'track\_id', 'Release Year': 'release\_year'})

Обычно встречаются два вида отсутствующих значений в ячейках датасета:

* NaN — общепринятый способ заполнять ячейку, для которых не хватает данных;
* плейсхолдеры — произвольно выбранные значения (разработчики могут выбрать любое обозначение для пропусков, например: 0, '?', 'NN', 'n/a').

Метод .index.names – присвоение имени столбцу с индексами датасета.

df.index.names = ['Название']

Метод .isna() – поиск пропусков значений в таблице (метод isnull() ему эквивалентен, но по правилу хорошего стиля используют метод .isna()). Ищет в таблице NaN значения. Если значение отсутствует, он вернёт True, иначе вернет False.

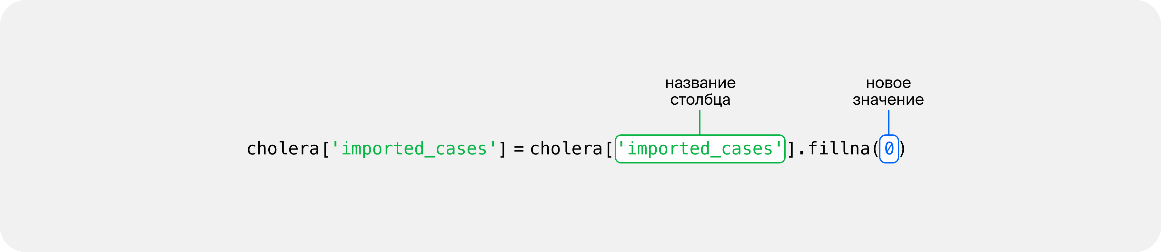
Часто .isna() вызывают вместе с методом .sum(), чтобы посчитать количество NaN в каждой колонке:

print(cholera.isna().sum())

print(logs[logs['email'].isna()].head()) *# выражение logs['email'].isna() проверяет все строки в столбце email датасета logs и оставляет только те, в которых есть NaN. Метод .head() выводит первые пять строк датасета logs со значениями NaN в столбце email.*

print(len(logs[logs['email'].isna()])) *# подсчет количества строк без email-адресов*

Метод .fillna() – заполняет пропуски другими значениями (возвращает копию исходного столбца, заменяя все NaN  на значение из аргумента).



*# В столбце imported\_cases заменим все NaN на 0*

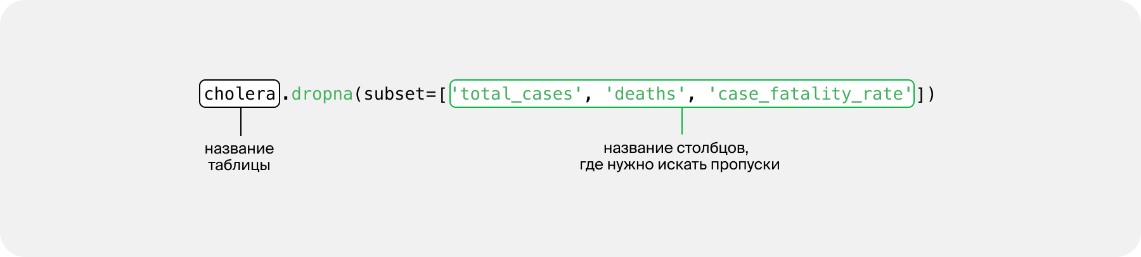
cholera['imported\_cases'] = cholera['imported\_cases'].fillna(0)

logs = logs.fillna('example@example.com') *# заменяет*

data.loc[:, 'Новая функция'] = data.loc[:, 'Новая функция'].fillna('+') *# замена всех значений в столбце «Новая функция» на «+».*

Метод .dropna() – удаляет строку или колонку с пропущенными значениями (в которой встречается NaN ). Имеет два параметра:

* axis — ось, по которой будет ориентироваться метод. Если передать параметру значение 'columns', метод удалит столбцы, в которых встречается NaN. Если не указывать этот аргумент, метод удалит строки, в которых встречается NaN.
* subset — список с названиями столбцов, в которых метод должен искать NaN.



*# удаление строк, в которых в столбцах total\_cases, deaths или case\_fatality\_rate встречается NaN*

cholera = cholera.dropna(subset=['total\_cases', 'deaths', 'case\_fatality\_rate'])

*# удаление столбцов, в которых в столбцах total\_cases, deaths или case\_fatality\_rate встречается NaN*

cholera = cholera.dropna(subset=['total\_cases', 'deaths', 'case\_fatality\_rate'], axis='columns')

Метод .lower() – приводит все символы датасета в строках к нижнему регистру.

print(giraffe\_height.lower())

Метод .str.lower() – приводит все строки в Series к нижнему регистру.

df['новый\_столбец'] = df['старый\_столбец'].str.lower()

Метод .duplicated() – находит строки-дубликаты, у которых полностью совпадают значения в ячейках, возвращает Series с значением True для таких строк.

import pandas as pd

df = pd.DataFrame({

'item': ['shirt', 'shirt', 'shirt', 'shirt', 'hat'],

'style': ['classic', 'classic', 'casual', 'classic', 'casual'],

'rating': [4, 4, 3.5, 4, 5]

})

print(df, '\n')

print(df.duplicated())

Результат:

item style rating

0 shirt classic 4.0

1 shirt classic 4.0

2 shirt casual 3.5

3 shirt classic 4.0

4 hat casual 5.0

0 False

1 True

2 False

3 True

4 False

dtype: bool

Получить таблицу со всеми дубликатами датафрейма df c помощью логической индексации и метода .duplicated() можно так:

duplicated\_df = df[df.duplicated()]

Для подсчета количества строк-дубликатов, результат метода .duplicated() передают методу .sum() :

print(df.duplicated().sum())

или

df['название столбца'].duplicated().sum()

Метод .drop\_duplicates() – удаляет строки-дубликаты. Нужно иметь в виду, что метод удаляет строки вместе с их индексами, в результате в нумерации строк появляются пропуски.

df = df.drop\_duplicates()

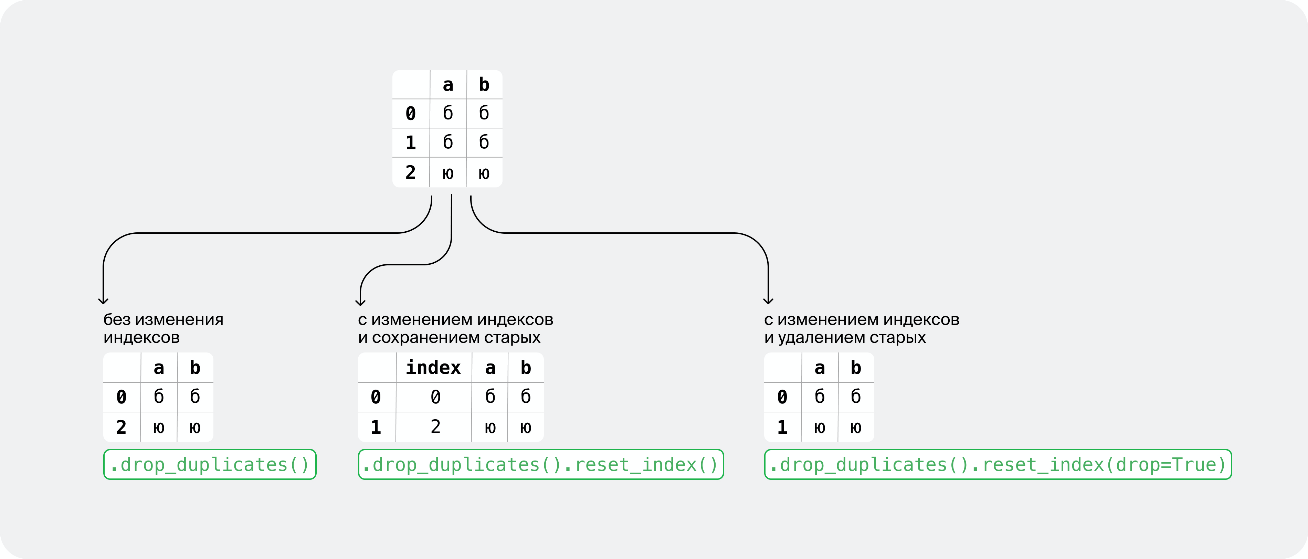
У метода .drop\_duplicates() есть параметры:

* subset - передает список с названиями столбцов в которых нужно искать дубликаты (будет искать дубликаты только в указанных столбцах);
* keep – определяет, какая строка из дубликатов будет оставаться (keep='first' - первая, keep='last' - последняя).

df = df.drop\_duplicates(subset=['name'], keep='first')

Метод .reset\_index() – упорядочивает индексы в датафрейме (создает новый датафрейм, в котором индексы идут без пропусков).

df = df.drop\_duplicates().reset\_index(drop=True)



Метод .unique() – возвращает перечень уникальных значений в столбце.

print(tennis['name'].unique())

print('Уникальных email:', len(logs['email'].unique())) *# подсчет количества уникальных email*

Метод .value\_counts() – возвращает перечень уникальных значений в столбце и количество их упоминаний.

print(logs['source'].value\_counts()) *# показывает какие уникальные значения есть в столбце source и их количество*

Метод .replace() – заменяет значения на другие. Первый аргумент – старое значение, второй аргумент – новое значение.

tennis['name'] = tennis['name'].replace('Roger Federer', 'Роджер Федерер')

*# пример1 с использованием списка:*

duplicates = ['Roger Fderer', 'Roger Fdrer', 'Roger Federer'] *# список неправильных имён*

name = 'Роджер Федерер' *# правильное имя*

tennis['name'] = tennis['name'].replace(duplicates, name) *# замена всех значений из duplicates на name*

*# пример2 с использованием собственной функции:*

def replace\_wrong\_values(wrong\_values, correct\_value): *# на вход функции подаются список неправильных значений и строка с правильным значением*

for wrong\_value in wrong\_values: *# перебираем неправильные имена*

tennis['name'] = tennis['name'].replace(wrong\_value, correct\_value) *# и для каждого неправильного имени вызываем метод replace()*

duplicates = ['Roger Fderer', 'Roger Fdrer', 'Roger Federer'] *# список неправильных имён*

name = 'Роджер Федерер' *# правильное имя*

replace\_wrong\_values(duplicates, name) *# вызов функции, replace() внутри будет вызван 3 раза*

print(tennis) *# датафрейм изменился, неявные дубликаты устранены*

Метод .groupby() – группировка данных. Принимает в качестве аргумента название столбца, по которому нужно группироваться. Возвращает объект особого типа – DataFrameGroupBy – это сгруппированные данные. Если применить к ним какой-нибудь метод pandas, то они станут новой структурой данных типа DataFrame или Series.

print(exoplanet.groupby('discovered').count()) *# подсчет сгруппированных по годам экзопланет. Результат – DataFrame.*

exo\_number = exoplanet.groupby('discovered')['radius'].count() *# подсчет количества планет по годам, радиус которых известен. Результат – Series.*

exo\_radius\_sum = exoplanet.groupby('discovered')['radius'].sum() *# подсчет сумм радиусов планет по годам, радиус которых известен. Результат – Series.*

exo\_radius\_mean = exo\_radius\_sum/exo\_number *# можно делить Series на*

print(hogwarts\_points.groupby('faculty\_name')['points'].sum()) *# группирует датасет hogwarts\_points по столбцу faculty\_name и считает для каждого значения faculty\_name сумму значений в столбце points*

Результат:

faculty\_name

Гриффиндор 27

Когтевран 10

Пуффендуй 17

Слизерин 28

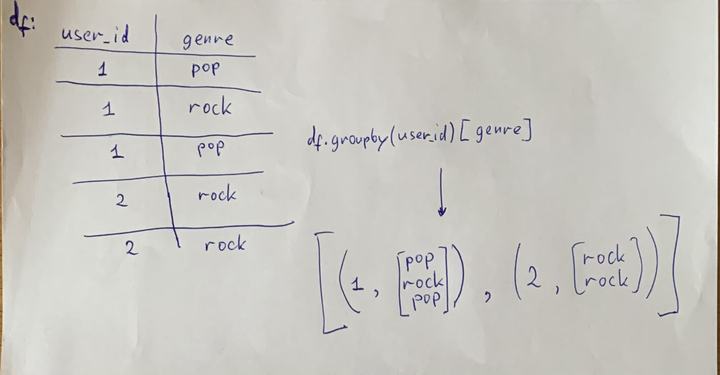
Name: points, dtype: int64

print(hogwarts\_points.groupby('faculty\_name')['points'].sum().sum()) *# в продолжение предыдущего примера, суммирует суммы по полю points*

Результат:

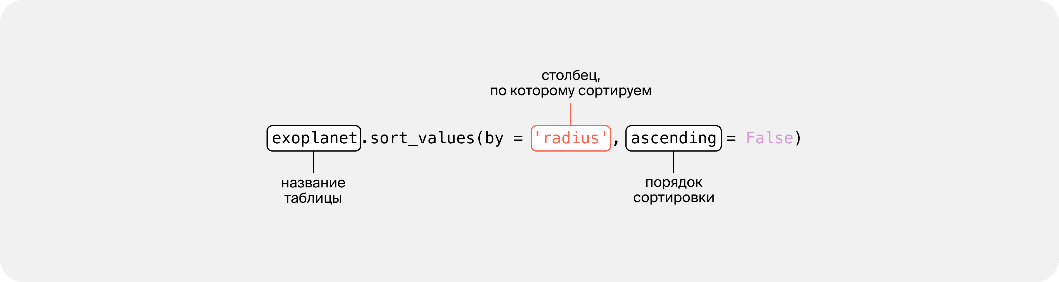
82

Структура группировки данных:



Для показа структуры можно использовать list(). print(list(df.groupby(…)))

Метод .sort\_values() – сортировка данных.



print(exoplanet.sort\_values(by='radius').head(30))

*# экзопланеты меньше Земли и ещё открытые в 2014 году*

exo\_small\_14 = exoplanet[exoplanet['radius'] < 1]

exo\_small\_14 = exo\_small\_14[exo\_small\_14['discovered'] == 2014]

print(exo\_small\_14)

exo\_small\_14 = exo\_small\_14.sort\_values(by='radius', ascending=False) *# сортировка в порядке убывания радиуса планеты*

*Эти две записи ниже эквивалентны:*

print(support['type\_id'].value\_counts())

print(support.groupby('type\_id')['type\_id'].count().sort\_values(ascending=False))

Метод .max() – выбор максимального значения.

print(df['total\_play\_seconds'].max())

print(df[df['total\_play\_seconds'] == df['total\_play\_seconds'].max()])*# запрос из df строки с самым длинным треком*

Метод .idxmax() – возвращает индекс элемента с наибольшим численным значением.

print('Кубок получает', hogwarts\_points.groupby('faculty\_name')['points'].sum().idxmax())

Результат:

>>> Кубок получает Слизерин

Метод .mix() – выбор минимального значения.

print(df\_drop\_null['total\_play\_seconds'].min())

print(df\_drop\_null[df\_drop\_null['total\_play\_seconds'] == df\_drop\_null['total\_play\_seconds'].min()])()])*# запрос из df строк с самыми короткими прослушиваниями*

Метод .median() – определение медианы набора значений. Применяют ко всей таблице, к отдельному столбцу или к сгруппированным данным. Медиана – это такое число в выборке, что ровно половина элементов выборки больше этого числа, а другая половина меньше. Например, в выборке [0, 2, 3, 4, 9] медиана равна 3. Если количество значений в выборке четное, то за медиану берут среднее двух значений, оказавшихся посередине, например, в выборке [0, 2, 3, 4, 9, 22] медиана 3,5 (среднее двух центральных значений 3 и 4).

print(df\_stat['total\_play\_seconds'].median())

Метод .mean() – определение среднего арифметического. Применяют ко всей таблице, к отдельному столбцу или к сгруппированным данным.

print(df\_drop\_null['total\_play\_seconds'].mean())

Метод .agg() – указывает, какие методы нужно применить к столбцу датасета, при этом название столбца датасета и названия методов записываются в формате словаря. Например: {'purchase': ['count','sum']}, где 'purchase' – название столбца является ключом словаря, 'count','sum' – названия применяемых методов, являются значениями словаря.

logs\_grouped = logs.groupby('source').agg({'purchase': ['count', 'sum']})

Результат:

purchase

source count sum

None 1674 108

context 52032 3029

email 12279 913

other 133834 8041

undef 181 12

Пример. Заполнение пропусков в столбце time средним временем отдельно для каждой категории в столбце device\_type.

import pandas as pd

*# создаём датафрейм metrica из CSV-файла*

metrica = pd.read\_csv('/datasets/metrica\_data.csv')

*# перебираем каждый тип девайса в наборе уникальных значений столба device\_type*

for d in metrica['device\_type'].unique():

*# на каждом шаге цикла с помощью атрибута loc выбираем строки,*

*# в которых в device\_type текущий тип девайса (d) и есть пропуски в time*

metrica.loc[(metrica['device\_type'] == d) & (metrica['time'].isna()), 'time'] = \

metrica.loc[(metrica['device\_type'] == d), 'time'].mean()

*# и записываем в них среднее значение time среди строк с текущим типом девайса (d)*

*# проверяем, что все пропуски заполнены*

print(metrica['time'].isna().sum())

Результат:

0

Метод .to\_numeric(errors= ) – превращает значения столбца в числовой тип float64 (вещественное число). От значения параметра errors зависит поведение при встрече с некорректным значением:

* `errors='raise' – дефолтное поведение: при встрече с некорректным значением выдаётся ошибка, операция перевода в числа прерывается;
* `errors='coerce' – некорректные значения принудительно заменяются на NaN;
* `errors='ignore' – некорректные значения игнорируются, но остаются.

transactions['amount'] = pd.to\_numeric(transactions['amount'], errors='coerce')

Метод .astype() – изменение типа значения.

df['cjlumn'] = df['cjlumn'].astype('int')

Метод .to\_datetime() – библиотеки pandas превращает значения столбца в формат дата-время. В качестве аргумента получает столбец значений и аргумент format со следующими значениями:

* %d — день месяца (от 01 до 31)
* %m — номер месяца (от 01 до 12)
* %Y — четырёхзначный номер года (например, 2019)
* %y — двузначный номер года (например, 19);
* Z или T — стандартный разделитель даты и времени
* %H — номер часа в 24-часовом формате
* %I — номер часа в 12-часовом формате
* %M — минуты (от 00 до 59)
* %S — секунды (от 00 до 59)

arrivals['date\_datetime'] = pd.to\_datetime(arrivals['date'], format='%d.%m.%YZ%H:%M:%S')

arrivals['target\_datetime'] = pd.to\_datetime(arrivals['target\_time'], format='%Y-%m-%dZ%H:%M:%S')

data['date\_time'] = pd.to\_datetime(data['date\_time'], format='%Y%m%dT%H%M%S')

arrivals['target\_datetime'] = pd.to\_datetime(arrivals['unix\_time'], unit='s') *# перевод времени из формата unix time в обычный формат с точностью до секунды*

Метод .dt.round() – округление времени для значений в формате datetime. В качестве параметра передают строку с шагом округления в днях, часах, минутах или секундах: D — дни; H — часы; min или T — минуты; S — секунды.

import pandas as pd

df = pd.DataFrame({'time': ['2011-03-01 17:34']})

df['time'] = pd.to\_datetime(df['time'], format='%Y-%m-%d %H:%M')

df['time\_rounded'] = df['time'].dt.round('1H') *# округляем до ближайшего значения с шагом в один час*

print(df['time\_rounded'])

Результат:

0 2011-03-01 18:00:00

Метод .dt.ceil() – округление времени к большему значению, параметры аналогичны методу .dt.round().

Метод .dt.floor() – округление времени к большему значению, параметры аналогичны методу .dt.round().

df['ceil'] = df['time'].dt.ceil('1H') *# округляем вверх*

df['floor'] = df['time'].dt.floor('1H') *# округляем вниз*

Метод .dt.weekday – возвращает номер дня в неделе для значения в формате datetime (понедельник – 0, вторник – 1, …, воскресение – 6).

df['weekday'] = df['time'].dt.weekday

Метод (возможно, класс) .pd.Timedelta() – библиотеки pandas переводит время в другой часовой пояс, в аргументе hours передается положительная или отрицательная разница во времени.

df['petropavlovsk-kamchatsky\_time'] = df['moscow\_time'] + pd.Timedelta(hours=9) *# создаем столбец с временем, сдвинутым на 9 часов для Петропавловска-Камчатского относительно Москвы*

Класс DatetimeIndex .

Атрибут .month .

arrivals['month'] = pd.DatetimeIndex(arrivals['date\_datetime']).month *# добавляет в таблицу столбец month с номером месяца, высчитывая его из столбца date\_datetime*

position['month'] = pd.DatetimeIndex(position['timestamp']).month *# аналогично*

Метод .merge() – объединение (присоединение) таблиц. Применяют к таблице к которой присоединяют. Аргументы:

* right — имя DataFrame или Series, присоединяемого к исходной таблице;
* on — название общего списка (столбца) в двух соединяемых таблицах (по нему происходит слияние);
* how — чьи id включать в итоговую таблицу. Если left , то в итоговую таблицу будут включены id из левой (исходной) таблицы. Если right , то в итоговую таблицу будут включены id из правой (присоединяемой) таблицы.

data\_subcategory = data.merge(subcategory\_dict, on='subcategory\_id', how='left')

Метод .pivot\_table() – формирование сводной таблицы. Формирует «широкий» вид таблицы. Аргументы:

* index — столбец или столбцы, по которым группируют данные (например, название товара)
* columns — столбец, по значениям которого происходит группировка (например, даты)
* values — значения, по которым мы хотим увидеть сводную таблицу (например, количество проданного товара)
* aggfunc — функция, применяемая к значениям (например, сумма товаров)

data\_pivot = data\_final.pivot\_table(index=['category\_name', 'subcategory\_name'], columns='source', values='visits', aggfunc='sum')

Результат:

source direct organic

category\_name subcategory\_name

Авто Автоакустика 5915 15433

Автомагнитолы 7783 18690

Автомобильные инверторы 145 150

Пример применения .pivot\_table():

data\_pivot = data.pivot\_table(values='debt', index='total\_income\_category', aggfunc=['count','sum', 'mean'], margins=True)

data\_pivot['mean'] = data\_pivot['mean'].round(4) \* 100

data\_pivot['Отклонение от общего процента должников, %'] = data\_pivot['mean'] - data\_pivot.loc['All', 'mean']

data\_pivot.insert(0, 'Уровень дохода, руб.', ['1000001 и выше', '200001–1000000', '50001–200000', '30001–50000', '0–30000', ''])

data\_pivot.rename(columns={'count': 'Количество заемщиков', 'sum': 'Количество должников', 'mean': 'Средний процент должников, %', 'debt': ''}, inplace=True)

data\_pivot.index.names = ['Категория']

display(data\_pivot)

Пример с использованием принципа «Бритва Оккама» - ‘Entia non sunt multiplicanda praeter necessitatem.’ – «Сущности не должны множиться сверх необходимого» – совершенство в простоте:

data.query('id == "3c1e4c52"').pivot\_table(values='time\_spent', index='date\_hour', aggfunc='count').plot(grid=True, figsize=(12, 5)) *# вариант "бритвы Охама" - в одну строчку, без введения дополнительных не нужных далее переменных*

Формирование сводной таблицы методами groupby() и agg(). Формирование «длинного» вида сводной таблицы. Метод groupby() принимает один аргумент — столбец (или список столбцов), по которым группируют данные. В метод agg() передают словарь. Его ключ — это названия столбцов, а значение — функции, которые будут к этим столбцам применены (например, .sum или .count). Такие функции называются агрегирующие.

data\_grouped = data\_final.groupby(['category\_name','subcategory\_name','source']).agg({'visits':'sum'})

print(data\_grouped.head(10))

Результат:

visits

category\_name subcategory\_name source

Авто Автоакустика direct 5915

organic 15433

Автомагнитолы direct 7783

organic 18690

Автомобильные инверторы direct 145

organic 150

Метод .apply() – применяет к значениям столбца или всего датафрейма функцию, указанную в его аргументе. Позволяет применить функцию к каждой строке или столбцу датасета без использования цикла.

название\_датафрейма['название\_столбца'].apply(название\_функции)

Когда метод .apply() оперирует данными из нескольких столбцов, его вызывают ко всему датафрейму. В таком случае он принимает не только название функции, но и параметр axis: со значением 1, чтобы применить метод ко всем строкам датафрейма, и со значением 0 — ко всем столбцам (0 – по умолчанию).

clients['full\_name'] = clients.apply(make\_full\_name, axis=1)

Сброс ограничений на количество выводимых рядов:

pd.set\_option('display.max\_rows', None)

pd.set\_option('display.max\_rows', 7) # установка 7 строк для вывода

Сброс ограничений на число столбцов:

pd.set\_option('display.max\_columns', None)

pd.reset\_option('display.max\_columns') # возврат к значению по умолчанию

Сброс ограничений на количество символов в записи:

pd.set\_option('display.max\_colwidth', None)

pd.set\_option('display.max\_colwidth, 10) # будет выводить до 10 символов в столбце

**Визуализация**

1. Метод display из библиотеки IPython.display.

Для более красивого вывода таблиц вместо print().

from IPython.display import display

display(df)

1. Тепловая карта (хитмэп) для датасета seaborn.heatmap(data)

Пример. Построить хитмэп по таблице с данными об установках и покупках «Книжного грызуна».

import pandas *# импорт библиотеки pandas*

data = pandas.read\_csv('app\_stats.csv') *# чтение файла и сохранение его в переменной data*

import seaborn *# импорт библиотеки seaborn*

seaborn.heatmap(data) *# строительство хитмэп*

Расширенный хитмэп:

seaborn.heatmap(mean\_scores, xticklabels=intervals, yticklabels=segments\_new, annot=True, cmap='RdYlGn')

где:

xticklabels=intervals — для подписей на оси X взять значения из intervals;

yticklabels=segments\_new — для подписей на оси Y взять значения из segments\_new;

annot=True — добавить на график аннотации (отображать не только цвета «ячеек», но и их значения);

cmap='RdYlGn' — цветовая схема для числовой шкалы, в варианте 'RdYlGn' красный соответствует минимальным значениям, а зелёный — максимальным.

Пример с кодом:

import pandas

import seaborn

data = pandas.read\_csv('support\_data.csv')

# списки со старыми и новыми названиями сегментов, а также периодами

segments\_old = ['Segment 0', 'Segment 1', 'Segment 2']

segments\_new = ['Потенциальные клиенты', 'Обычные клиенты', 'VIP-клиенты']

intervals = ['До внедрения роботов', 'После внедрения роботов']

# вымышленные значения

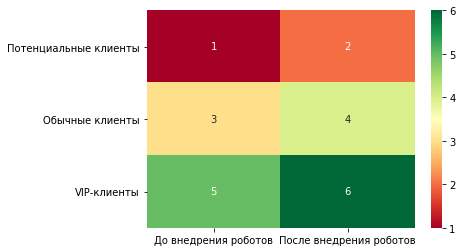
mean\_scores = [[1, 2],

[3, 4],

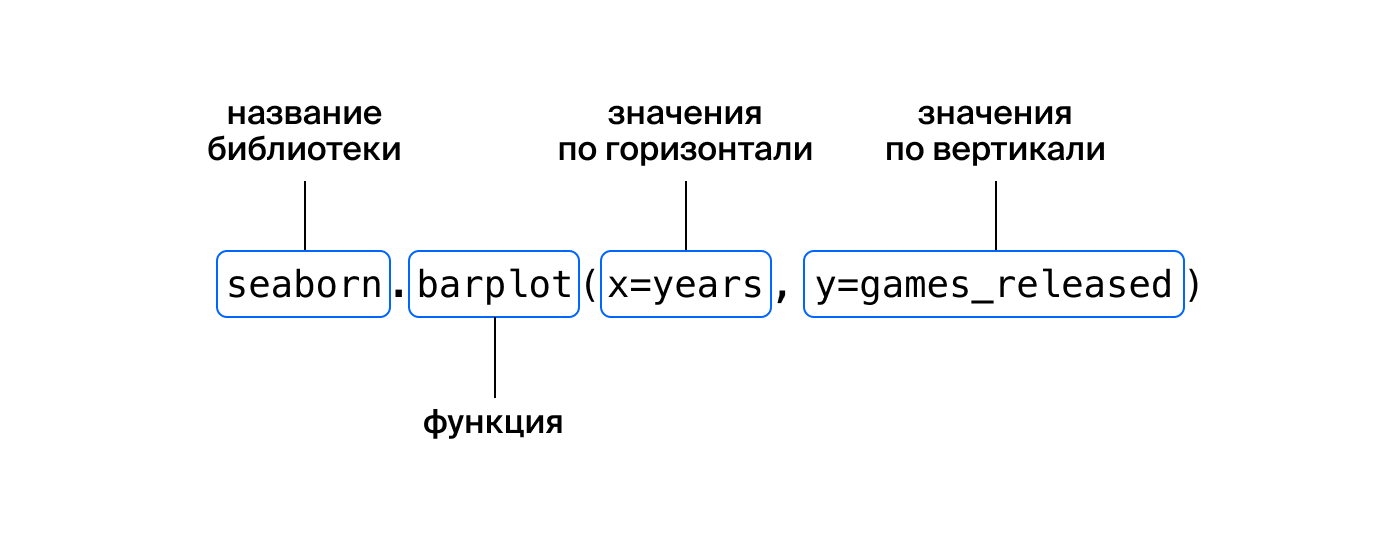
[5, 6]]

# настраиваем и строим хитмэп

seaborn.heatmap(mean\_scores, xticklabels=intervals, yticklabels=segments\_new, annot=True, cmap='RdYlGn')



1. Столбчатая диаграмма seaborn.barplot(x=years, y=games\_released)



import pandas

data = pandas.read\_csv('app\_stats.csv')

import seaborn

seaborn.barplot(x=data['week\_number'], y=data['ad\_campaign']) *# построение столбчатой диаграммы по двум столбцам датасета* week\_number *и* ad\_campaign*.*

Пример. Построение столбчатой диаграммы, отражающей значения конверсии из списка conversions по неделям (столбец 'week\_number').

import pandas

data = pandas.read\_csv('app\_stats.csv')

payments = list(data['payments']) *# преобразование столбца датасета в список*

installs = list(data['installs']) *# преобразование столбца датасета в список*

conversions = [] *# создание нового пустого списка*

for index in range(len(payments)): *# в цикле расчет конверсии по неделям с занесением результатов в список 'conversion'*

   conversions.append(payments[index] / installs[index])

import seaborn

seaborn.barplot(x=data['week\_number'], y=conversions) *# построение столбчатой диаграммы*

Пример. Расчет конверсии пользователей, привлеченных рекламной компанией, и построение столбчатой диаграммы окупаемости рекламных компаний.

import pandas

data = pandas.read\_csv('app\_stats.csv')

payments = list(data['payments']) *# список с числом платежей*

installs = list(data['installs']) *# список с числом установок*

campaign\_weeks = [7, 9, 13, 15, 17, 19, 29, 31, 33, 45]

installs\_from\_ads = [] *# список с числом «рекламных» установок*

payments\_from\_ads = [] *# список с числом «рекламных» платежей*

for week\_number in campaign\_weeks:

   installs\_from\_ads.append(installs[week\_number] - installs[week\_number - 1])

   payments\_from\_ads.append(payments[week\_number] - payments[week\_number - 1])

conversions\_from\_ads = [] *# значения конверсии для «рекламных» установок*

for index in range(len(installs\_from\_ads)):

   conversions\_from\_ads.append(payments\_from\_ads[index] / installs\_from\_ads[index])

ads\_install\_average\_profit = [] *# доходы от одной «рекламной» установки*

for conversion in conversions\_from\_ads:

   ads\_install\_average\_profit.append(conversion \* 600)

import seaborn *# построение столбчатой диаграммы*

seaborn.barplot(x=campaign\_weeks, y=ads\_install\_average\_profit)

Альтернативный вариант кода:

import pandas

data = pandas.read\_csv('app\_stats.csv')

payments = list(data['payments']) *# список с числом платежей*

installs = list(data['installs']) *# список с числом установок*

campaign\_weeks = [7, 9, 13, 15, 17, 19, 29, 31, 33, 45]

installs\_from\_ads = [] *# список с числом «рекламных» установок*

payments\_from\_ads = [] *# список с числом «рекламных» платежей*

conversions\_from\_ads = [] *# значения конверсии для «рекламных» установок*

ads\_install\_average\_profit = [] *# доходы от одной «рекламной» установки*

for week\_number in campaign\_weeks:

   installs\_from\_ads.append(installs[week\_number] - installs[week\_number - 1])

   payments\_from\_ads.append(payments[week\_number] - payments[week\_number - 1])

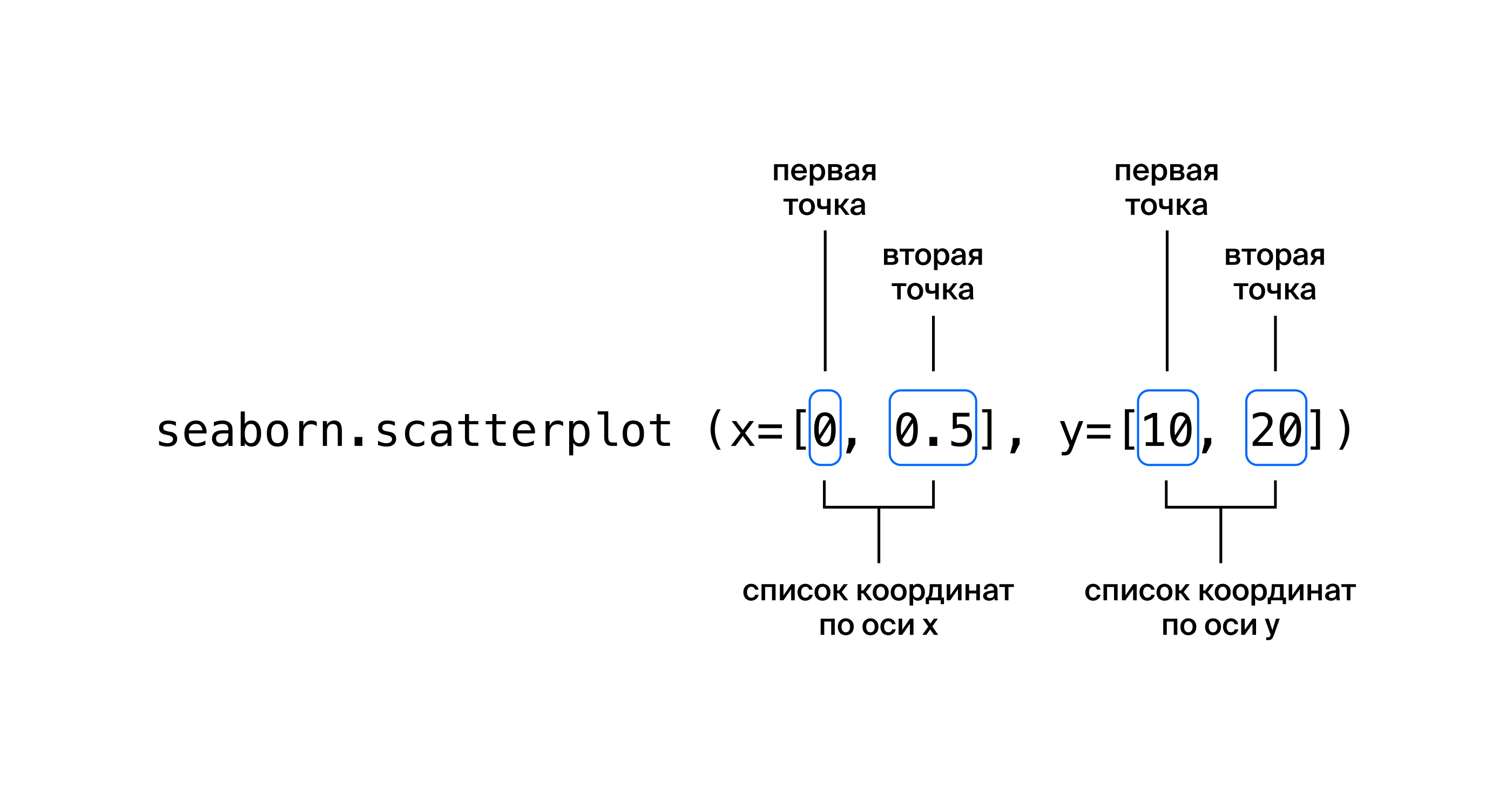
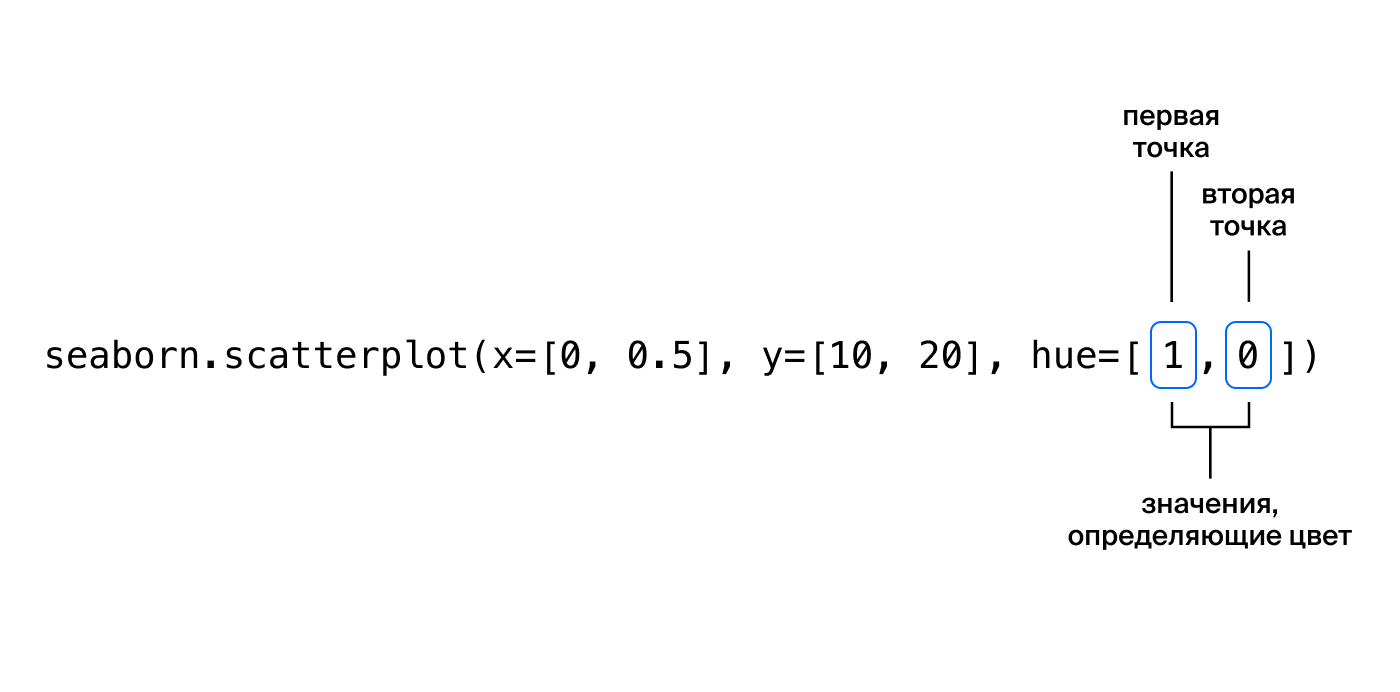
   conversions\_from\_ads.append(payments\_from\_ads[-1] / installs\_from\_ads[-1])

   ads\_install\_average\_profit.append(conversions\_from\_ads[-1] \* 600)

import seaborn *# построение столбчатой диаграммы*

seaborn.barplot(x=campaign\_weeks, y=ads\_install\_average\_profit)

1. Диаграмма рассеяния

Пример.

import seaborn

# список координат по оси X

x\_coordinates = [-0.4, 0.4, -0.2, -0.7, -0.8, -1.8, 0.5, 0.9, -0.1, 0.3]

# список координат по оси Y

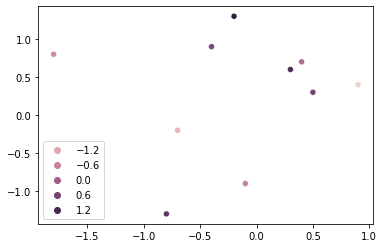
y\_coordinates = [0.9, 0.7, 1.3, -0.2, -1.3, 0.8, 0.3, 0.4, -0.9, 0.6]

# список значений для окрашивания точек

dots\_features = [0.5, -0.2, 1.5, -1.4, 0.9, -0.8, 0.6, -1.8, -0.6, 1.1]

seaborn.scatterplot(x=x\_coordinates, y=y\_coordinates, hue=dots\_features)

Результат:



1. Гистограмма

Метод .hist() – применяется к списку или столбцу датафрейма. Находит в наборе чисел минимальное и максимальное значения, полученный диапазон делит на области (корзины), затем считает, сколько значений попало в каждую область (корзину) и отображает это на графике. Параметр bins определяет, на сколько областей делить диапазон данных (по умолчанию 10). Параметр range указывает границы интересующего интервала – range=(min\_value, max\_value).

import matplotlib.pyplot as plt *# вызов модуля pyplot библиотеки matplotlib*

pd.Series([6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]).hist(bins=11)

pd.Series([6, 7, 8, 9, 10, 11, 12, 13, 14, 100]).hist(range = (6, 14))

data['time\_spent'].hist(bins=100, range=(0, 1500))

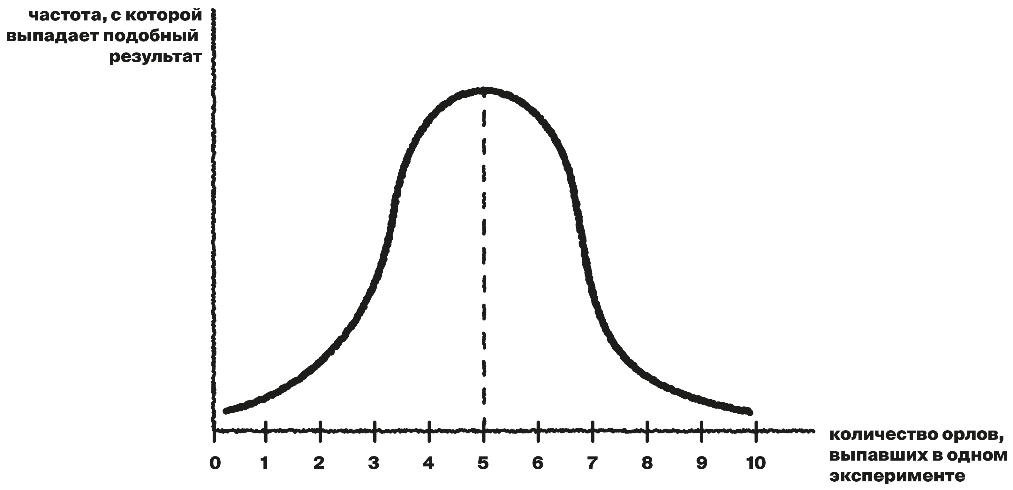
Метод .show() – из модуля pyplot библиотеки matplotlib отображение графика на экране.

plt.show()

1. Распределения

Нормальное распределение

Наиболее распространены в жизни. Если есть отклонение распределения от нормального, то обычно это признак наличия аномалии.



В общем случае любое отклонение от ожидаемого распределения — это сигнал, что данные не в порядке.

Распределение Пуассона

Описывает число событий в единицу времени. Например, в среднем в магазин заходит два человека в минуту. Но это не значит, что каждую минуту заходит ровно два человека. Может, в эту минуту не зайдёт ни одного, а через минуту зайдёт пять. Если каждый посетитель заходит случайным образом, независимо от других, то это и есть распределение Пуассона.

Форма распределения Пуассона зависит от числа событий. При небольших значениях распределение смещено влево, а с увеличением числа событий становится симметричным, уходит вправо и стремится к нормальному распределению.

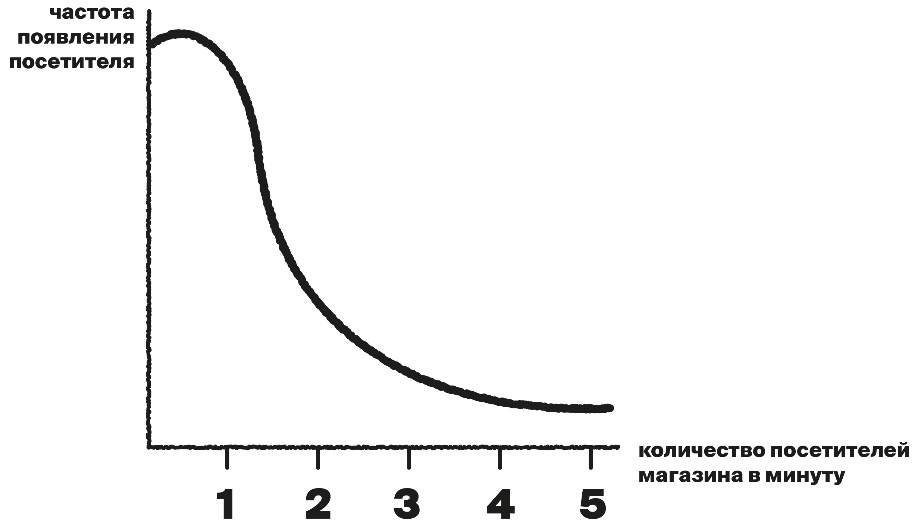
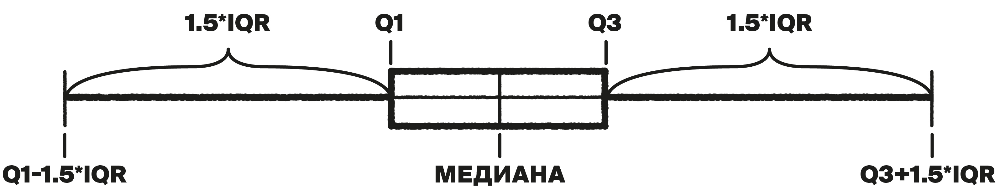
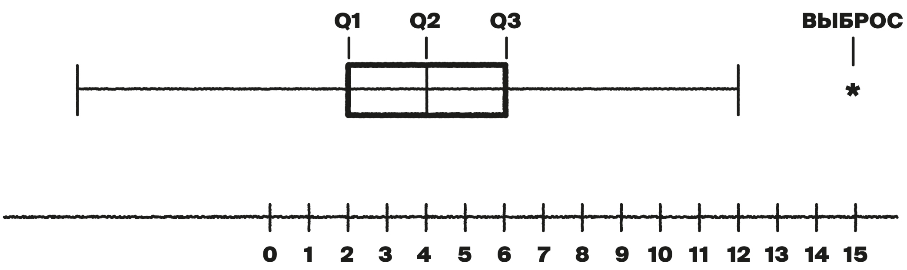
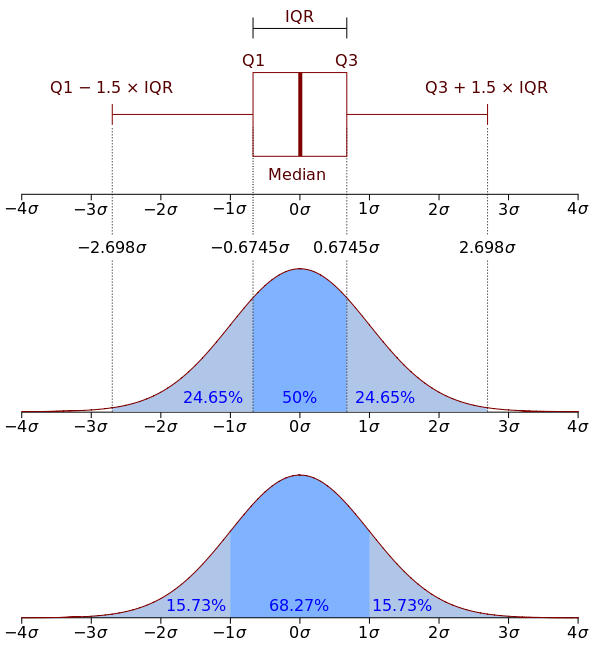


Диаграмма размаха (ящик с усами)



«Ящик» ограничен первым и третьим квартилями. Внутри ящика обозначают медиану. «Усы» простираются влево и вправо от границ ящика на расстояние, равное 1,5 межквартильного размаха (сокращённо IQR, от англ. interquartile range, «межквартильный размах»). В размах «усов» попадают нормальные значения, а за пределами находятся выбросы, изображённые точками. Если правый «ус» длиннее максимума, то он заканчивается максимумом. То же — для минимума и левого «уса».





Справочно:

*Квартили* (от лат. quartus — «четвёртый») разбивают упорядоченный набор данных на четыре части. Первый квартиль Q1 — число, отделяющее первую четверть выборки: 25% элементов меньше, а 75% — больше него. Медиана — второй квартиль Q2, половина элементов больше и половина меньше неё. Третий квартиль Q3 — это отсечка трёх четвертей: 75% элементов меньше и 25% элементов больше него. *Межквартильный размах* — это расстояние между Q1 и Q3.

Метод .boxplot() – построение диаграммы размаха.

Метод .ylim(y\_min, y\_max) из библитотеки matplotlib – изменение масштаба графика по вертикальной оси ординат.

Метод .xlim(x\_min, x\_max) из библитотеки matplotlib – изменение масштаба графика по горизонтальной оси абсцисс.

import matplotlib.pyplot as plt *# вызов модуля pyplot библиотеки matplotlib*

plt.ylim(-50, 500)

plt.xlim(0, 200)

Метод .describe() – числовое описание данных (характеристики, дающие представление о выборке).

print(data.describe())

результат:

count 3.0 *# количество наблюдений в наборе данных*

mean 2.0 *# среднее арифметическое*

std 1.0 *# стандартное отклонение*

min 1.0 *# минимальное значение*

25% 1.5 *# первый квартиль*

50% 2.0 *# медиана, или второй квартиль*

75% 2.5 *# третий квартиль*

max 3.0 *# максимальное значение*

dtype: float64 *# тип данных*

1. Графики

Метод .plot() – построение графика. Агрументы:

title – заголовок, название графика;

x, y – значения по осям координат;

style – вид точек на графике (варианты: o, x, o-);

xlim, ylim – задают минимальные и максимальные значения для осей координат;

grid – показывать сетку на графике (grid=True);

figsize – задает размеры графика на экране по ширине и высоте.

df.plot(title='A и B', x='b', y='a', style='o-', xlim=(0, 30), grid=True, figsize=(10, 3))

**Теория вероятностей**

Термины и определения

Эксперимент — это повторяемый опыт, который может окончиться элементарными исходами.

Элементарный исход — это исход, который либо случился, либо нет.

Равновероятные исходы — это исходы, вероятность каждого из которых одинакова.

Вероятностное пространство — множество всех элементарных исходов эксперимента.

Событие — это подмножество с некоторым количеством элементарных исходов на вероятностном пространстве.

Событие, состоящее из пустого подмножества вероятностного пространства, вероятность возникновения которого равна нулю, называется невозможным.

Событие, состоящее из подмножества, содержащего в себе все возможные исходы вероятностного пространства (равное всему вероятностному пространству), вероятность возникновения которого равна единице, называется достоверным.

Вероятность события — количество исходов, входящих в это событие, делённое на общее количество исходов (на размер вероятностного пространства). При обязательном условия равновероятности всех элементарных исходов.

Закон больших чисел — чем больше раз повторяется эксперимент, тем ближе частота заданного на этом эксперименте события будет к его вероятности.

Взаимоисключающие события — события, которые при проведении эксперимента не могут произойти одновременно. Вероятность пересечения взаимоисключающих событий равна нулю. На диаграмме Эйлера-Венна они не пересекаются.

События называют независимыми, если наступление одного из них не влияет на вероятность другого.

Вероятность пересечения независимых событий равна произведению их вероятностей.

Случайная величина — это переменная, которая принимает случайные значения (значения, которые нельзя предсказать).

Математическое ожидание случайной величины — численное значение, к которому будет в среднем стремиться случайная величина при многократном повторе эксперимента. Если эксперимент состоит из равновероятных элементарных исходов, заданных численно, математическое ожидание будет равно среднему возможных значений.

Математическое ожидание случайной величины — сумма всех значений случайной величины, помноженных на их вероятности.

Биномиальные эксперименты — это эксперименты всего с двумя исходами. Обычно один из результатов биномиального эксперимента называют успехом, а второй, соответственно, неудачей.

Центральная предельная теорема — множество независимых случайных величин, сложенных вместе, дают близкое к нормальному распределение.

Правило трех сигм — для нормальных распределений практически все значения (около 99%) находятся в промежутке: (𝜇−3𝜎, 𝜇+3𝜎).

Значения вне этого интервала называют выбросами.

Дисперсия совокупности:

Дисперсия выборки:

Стандартное отклонение совокупности:

Стандартное отклонение выборки:

Число сочетаний: