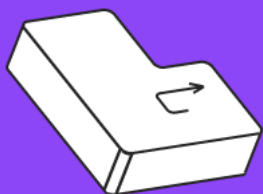




Знакомство с Языком Программирования Python

Курс



Оглавление

Введение	2
Термины, используемые в лекции	3
Начало работы с Python	3
Операторы ввода и вывода данных	11
Логические операции	14
Управляющие конструкции: if, if-else	15
Управляющие конструкции: while и вариация while-else	19
Цикл for, range	21
Срезы	24

Введение

Доброго времени суток, уважаемые студенты! Перед тем как начать изучать Python, я бы хотел с Вами окунуться в историю создания данного языка. Оглянемся назад, а именно в декабрь 1989 года голландец Гвидо (Guido van Rossum) — будущий создатель одного из самых популярных языков программирования — искал хобби-проект, которому можно было бы посвятить рождественские каникулы... Сам Гвидо вспоминает это время так: ***“Каждое приложение, которое мы должны были писать для Атоева, представляло собой либо shell-скрипт, либо программу на С. И я обнаружил, что у обоих вариантов были недостатки. Мне захотелось, чтобы существовал третий язык, который был бы посередине: ощущался как настоящий язык программирования (возможно, интерпретируемый), был проще в использовании, кратким и выразительным как shell-скрипты, но чтобы с читаемостью всё было не так ужасно, как у этих скриптов.”***



В качестве названия **Guido van Rossum** выбрал **Python** в честь комедийных серий ВВС "Летающий цирк **Монти-Пайтона**", а вовсе не по названию змеи.

Требовался второй язык программирования на С или С++ для решения задач, для которых написание программы на С было просто неэффективным.

Первая «официальная» версия языка увидела свет в 1994 году, когда Гвидо еще работал в CWI. Среди прочего в первой версии появились инструменты функционального программирования и поддержка комплексных чисел. Но самое главное, что следующий шаг сделал не только проект, но и его сообщество.

В том же 1994 году состоялась первая рабочая встреча пользователей Python. Встреча прошла в государственном бюро стандартов США (NBS, сегодня это государственный институт стандартов и технологий — NIST).

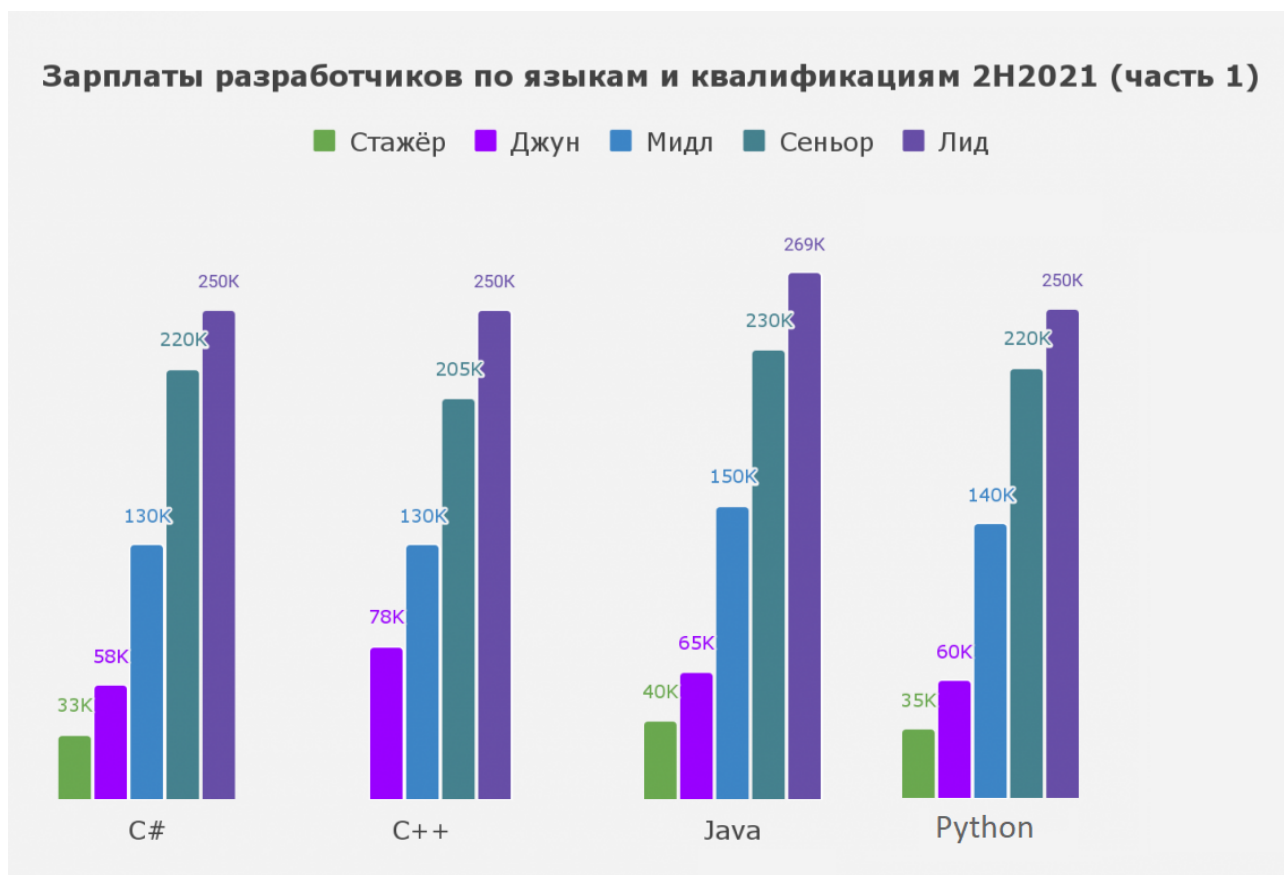
Начиная с 1994 года по настоящее время мы можем пользоваться таким удобным и простым языком Python. Почему люди после появления Python все равно пользуются такими языками как: С++, Java, С, ведь на Python можно создавать графические интерфейсы, игры, веб-приложения. Давайте разберемся.

Начнем мы с того что, Python - это скриптовый язык программирования с динамической типизацией данных. Это означает, что при создании переменных не нужно будет указывать ее тип. Python самостоятельно определяет ее тип согласно присвоенному значению. А вот Java и С++ — статические типизированные языки. Все типы переменных здесь должны быть объявлены. Если допустить ошибку, то программа работать не будет или будет, но с проблемами.

У статически типизированных языков есть недостатки, часть которых была описана выше. Но у них есть и достоинства, которых тоже немало. Так, например, Java обеспечивает безопасность типов, которая улавливает все потенциальные ошибки во время компиляции, а не в процессе выполнения, как Python. Таким образом, вероятность появления ошибок уменьшается. В конечном итоге все это упрощает управление большими приложениями. Ошибки во время выполнения (которые появляются при разработке веб-приложений, например, на Python) сложнее идентифицировать и исправлять, чем ошибки во время компиляции. Кроме того, анализировать Java-код гораздо легче, чем код Python, что полезно в ситуациях, когда над одним проектом работает команда программистов. Java-программисты быстро поймут код друг друга, поскольку все объявлено явно, а вот Python-программисты могут столкнуться с несколькими проблемами при чтении кода веб-приложения. Дело в том, что все определяется или отображается в ходе выполнения приложения, когда становятся известны переменные или сигнатуры. Собственно, ни Java, ни Python не являются лучшим вариантом для создания высоконагруженных приложений, но у первого языка есть солидные преимущества по сравнению со вторым. Все это благодаря JIT (Just-in-Time Compiler), преобразующему обычный код в машинный язык. В итоге производительность Java-приложений примерно равна производительности того, что написано на С/С++.

После слов, сказанных ранее, не нужно думать, что Python плохой язык программирования и совершенно не востребованный, ниже привожу его статистику.

Уровень профессиональных качеств разработчика является ключевым коэффициентом формирования его заработной платы. В IT сформировались 4 основные градации специалистов: Junior, Middle, Senior и Lead. Узнаем, сколько зарабатывают Python-разработчики в разрезе этих характеристик на графике:



Как мы видим Python-разработчики получают почти наравне с C++ и Java разработчиками. Но на деле зарплата C++ и Java разработчиков больше в среднем на 15-20%.

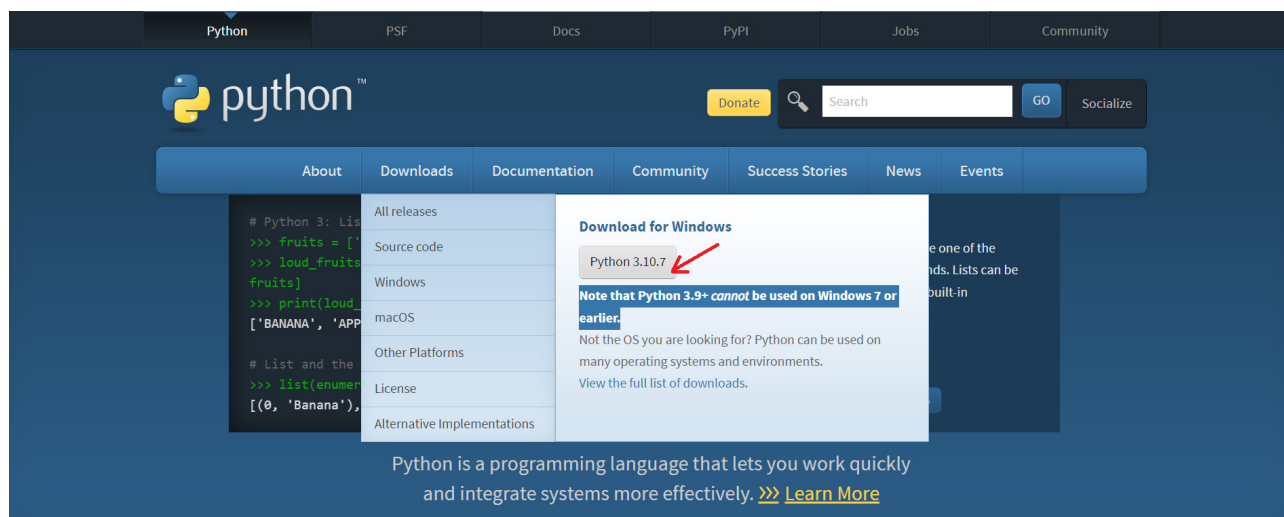
Давайте теперь перейдем к изучению одного из самых популярных языков программирования, Python.

Начало работы с Python

Перед тем как начать учить синтаксис Python, необходимо установить программу(интерпретатор), чтобы мы смогли запустить программный код. Интерпретация - это построчный анализ, обработка и выполнение исходного кода программы или запроса, в отличие от компиляции, где весь текст программы, перед

запуском анализируется и транслируется в машинный или байт-код без её выполнения.

1. Необходимо установить интерпретатор: [по ссылке](#)
2. С Windows, начиная с версии 8 и выше, можно смело устанавливать последнюю версию интерпретатора, но с Windows 7 или ниже необходимо установить версию 3.8 или ниже



3. После установки откройте командную строку(cmd), введите слово “python”, если все успешно установилось, у Вас выведется сообщение: “Python 3.10.7...”, какая именно версия была установлена

```
Командная строка - python
Microsoft Windows [Version 10.0.19044.2006]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\79190>python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep  5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

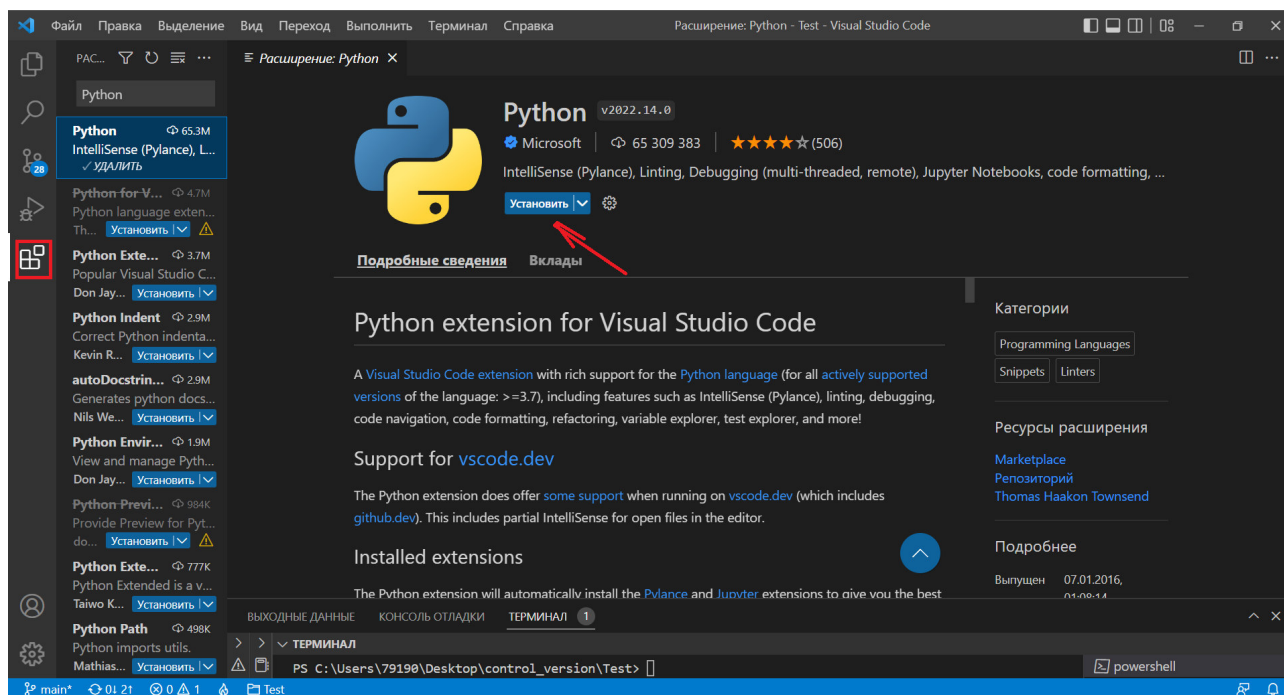
Интерпретатор успешно установлен!

Мы будем работать в **Visual Studio Code**

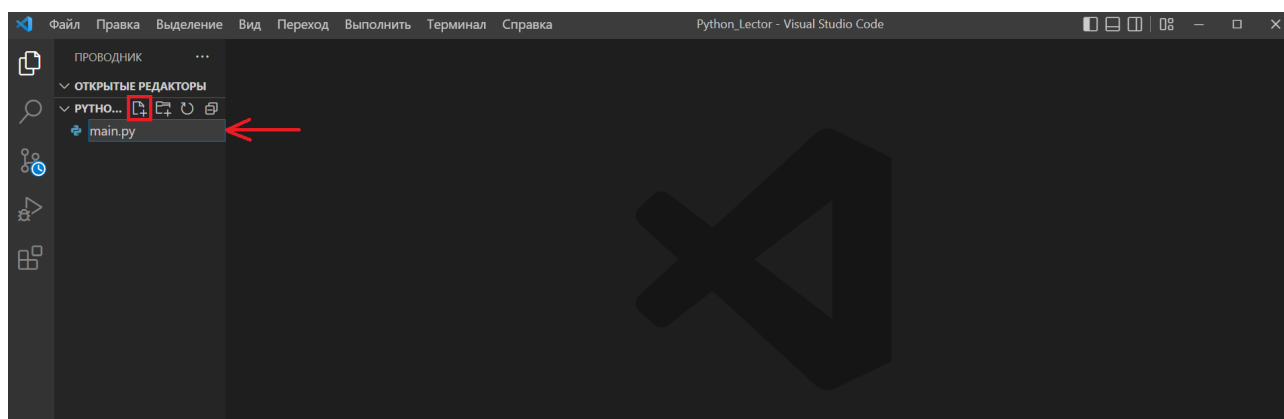
Если Вас не устраивает данная среда, в дополнительных материалах будет ссылка и текст для установки среды разработки от компании JetBrains(PyCharm)

💡 Чтобы работать было удобнее, установите расширение, которое будет подсвечивать синтаксис Python.

- Расширения(Extension) → введите в поиске «python» → Установить(install).



- В рабочем пространстве (Explorer) создайте папку с файлом для будущей программы с расширением .py (Указываем, что это файл Python).



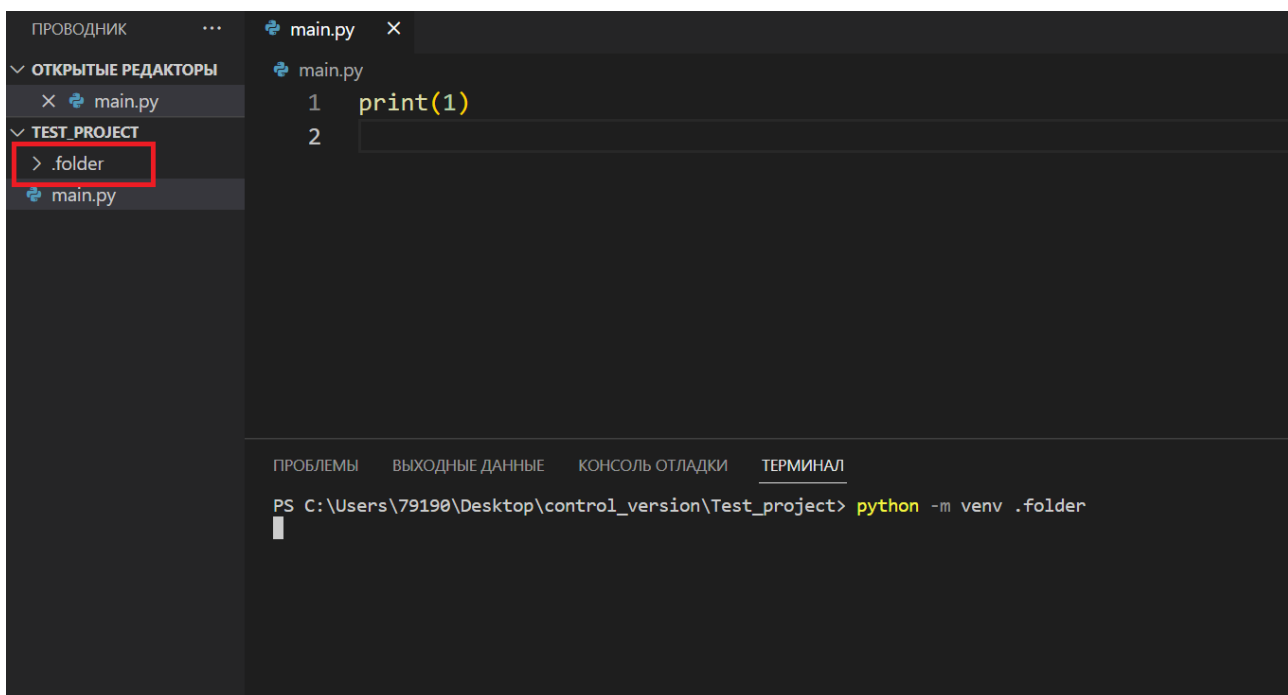
Виртуальное окружение

Мы с Вами установили интерпретатор, который позволит нам запускать пусть наши скрипты на Python. Представьте себе такую ситуацию: допустим у нас есть два проекта: **"Project A"** и **"Project B"**. Оба проекта зависят от библиотеки **Simplejson**. Проблема возникает, когда для **"Project A"** нужна версия **Simplejson 3.0.0**, а для проекта **"Project B"** — **3.17.0**. Python не может различить версии в глобальном каталоге **site-packages** — в нем останется только та версия пакета, которая была установлена последней.

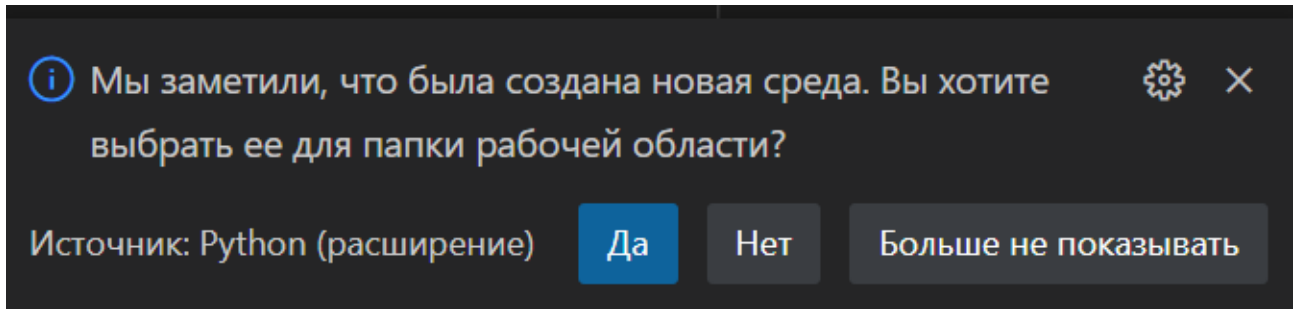
Решение данной проблемы — создание виртуального окружения (**virtual environment**).

Как добавить виртуальное окружение в свое приложение:

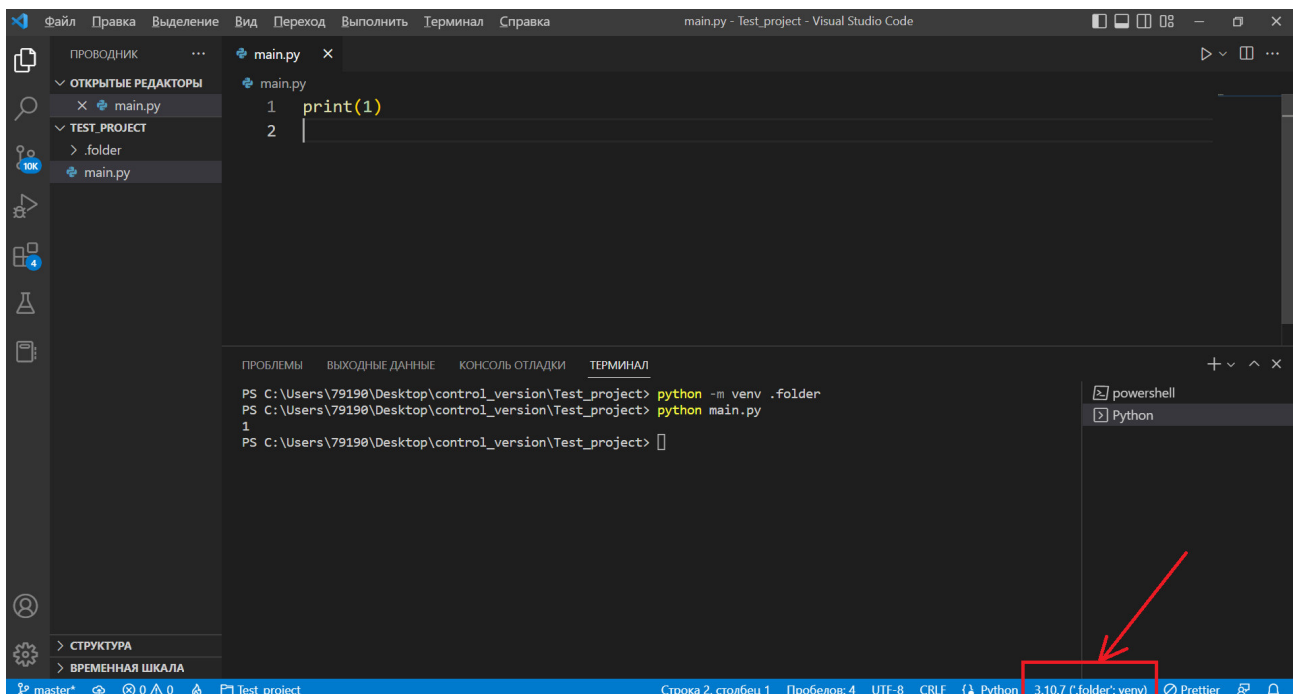
1. Создать новый проект в пустой папке (рабочего пространства).
2. В терминале прописать **python3 -m venv .folder** (.folder — название папки, в которой будет работа).



3. Выбираем “Yes” для работы с новым виртуальным окружением.



4. Справа внизу у Вас должно появиться виртуальное окружение, которое мы установили



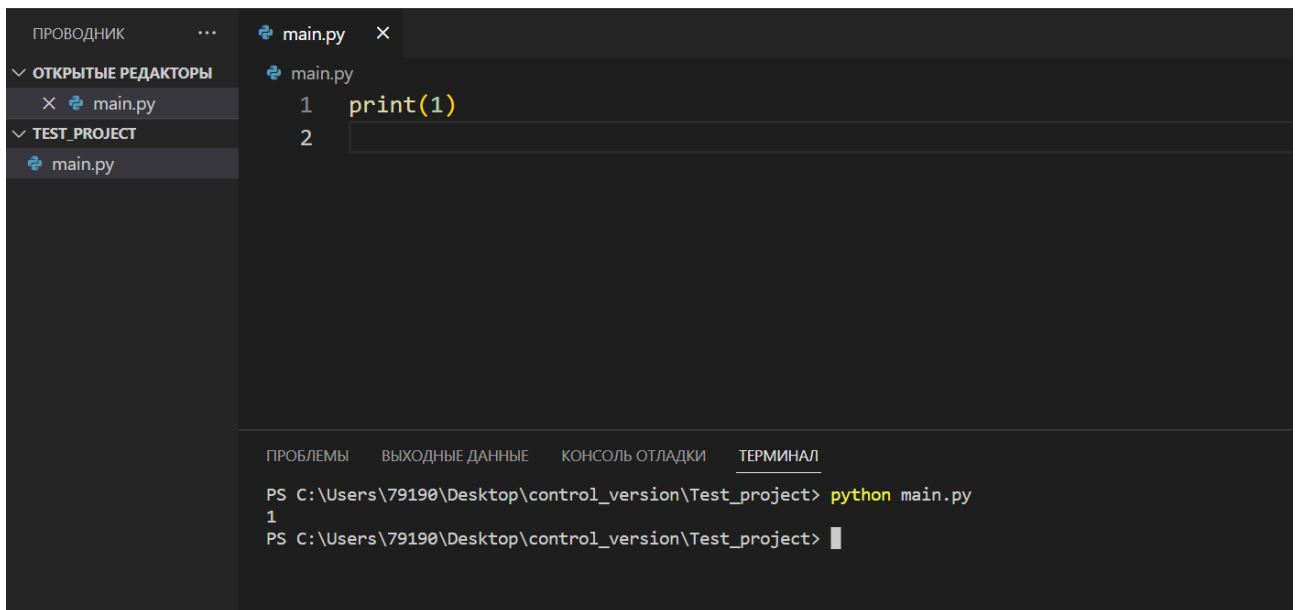
Как запустить скрипт:

💡 Если после выполнения команды в терминале остаются стрелочки >>>, нажмите ctrl + Z, чтобы выйти непосредственно в терминал.

Чтобы запустить программный код, используйте следующую команду в терминале

```
python name_file.py
```

Где name_file - имя вашего файла



```
main.py
1 print(1)
2

PROБЛЕМЫ Выходные данные КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ
PS C:\Users\79190\Desktop\control_version\Test_project> python main.py
1
PS C:\Users\79190\Desktop\control_version\Test_project>
```

Здесь мы написали программный код для проверки правильности установки интерпретатора Python.

Оператор вывода данных

print(var1, var2, var3) - функция, которая выводит данных на экран, где var1, var2, var3 - переменные или значения.

Синтаксис Python

Синтаксис Python очень простой и примитивный, приведу пример с языком программирования C#, который Вы проходили ранее.

Console.WriteLine(1);	print(1)
int n = 1;	n = 1
int n = Convert.ToInt32(Console.ReadLine());	n = int(input())

Как Вы видите синтаксис Python очень простой. Давайте познакомимся с базовыми типами данных.

Базовые типы данных Python:

int	Целые числа
float	Дробные числа
bool	Логический тип данных (True/False)
str	Строка

Объявление переменной

- название переменной = значение переменной (один знак равенства обозначает присвоение значения к переменной)

Программный код:

```
a = 123
b = 1.23
print(a)
print(b)
```

Вывод:

```
123
1.23
```

💡 Нельзя указать переменную, не присвоив ей какое-либо значение. Но можно присвоить значение None и использовать переменную дальше по коду.

Программный код:

```
value = None
a = 123
b = 1.23
print(a)
print(b)
value = 1234
print(value)
```

Вывод:

```
123
1.23
1234
```

Как узнать, какой тип данных содержится в переменной value?

Возникают такие ситуации, когда мы хотим узнать тип данных у переменной, для того, чтобы это выполнить необходимо применить функции `type(varName)`.

```
print(type(name)) # функция, которая указывает на тип данных
```

Как объявить строку?

Чтобы создать строку и сохранить ее в переменную необходимо написать следующим образом:

```
s = 'hello,' # создание 1-ой строки
s = "world" # создание 2-ой строки
print(s, w)
```

Как мы видим, строку можно создавать как одинарными кавычками, так и двойными.

Как сделать комментарий?

Если Вы хотите закомментировать 1 строку достаточно применить специальный символ “#”, если Вам нужно закомментировать сразу несколько строк выделите их и нажмите **ctrl + /** или же используйте тройные кавычки `'''`

```
# print(1)
# -----
'''print(1)
print(1)
print(1)
print(1)
print(1)'''
```

Использование одинарных или двойных кавычек внутри строки

Можно ли писать кавычки в виде текста внутри строки? Пример: **my mom shouted: “good luck!”**. Но для того чтобы создать строку мы должны использовать еще одни кавычки, как это сделать?

Используйте разные кавычки для объявления переменной и внутри строки:

```
s = 'hello "world"'\ns = "hello 'world'" \ns = 'hello \'world'
```

Иногда возникают такие ситуации, когда нужно вывести в одном предложении и числа и текст, но как это сделать более рационально и красиво, обратимся к такому понятию, как **интерполяция**



Интерполяция — способ получить сложную строку из нескольких простых с использованием специальных шаблонов.

```
a = 3\nb = 11\ns = 2022\nprint(a, b, s)\nprint(a, '-b, '-s)\nprint('{} - {} - {}'.format(a,b,s))\nprint(f'first - {a}  second - {b}  third  - {s}')
```

Логическая переменная

Ранее мы с Вами проговорили основные типы данных, такие как int, str, float и **bool**.

Bool - это логический тип данных, которые используются для представления двух значений истинности логики и булевой алгебры. Он назван в честь Джорджа Буля, который впервые определил алгебраическую систему логики в середине 19 века.

Но где именно можно и нужно применять этот тип данных? На самом деле мы с Вами поговорим об этом чуть-чуть позже, когда поговорим о циклах.

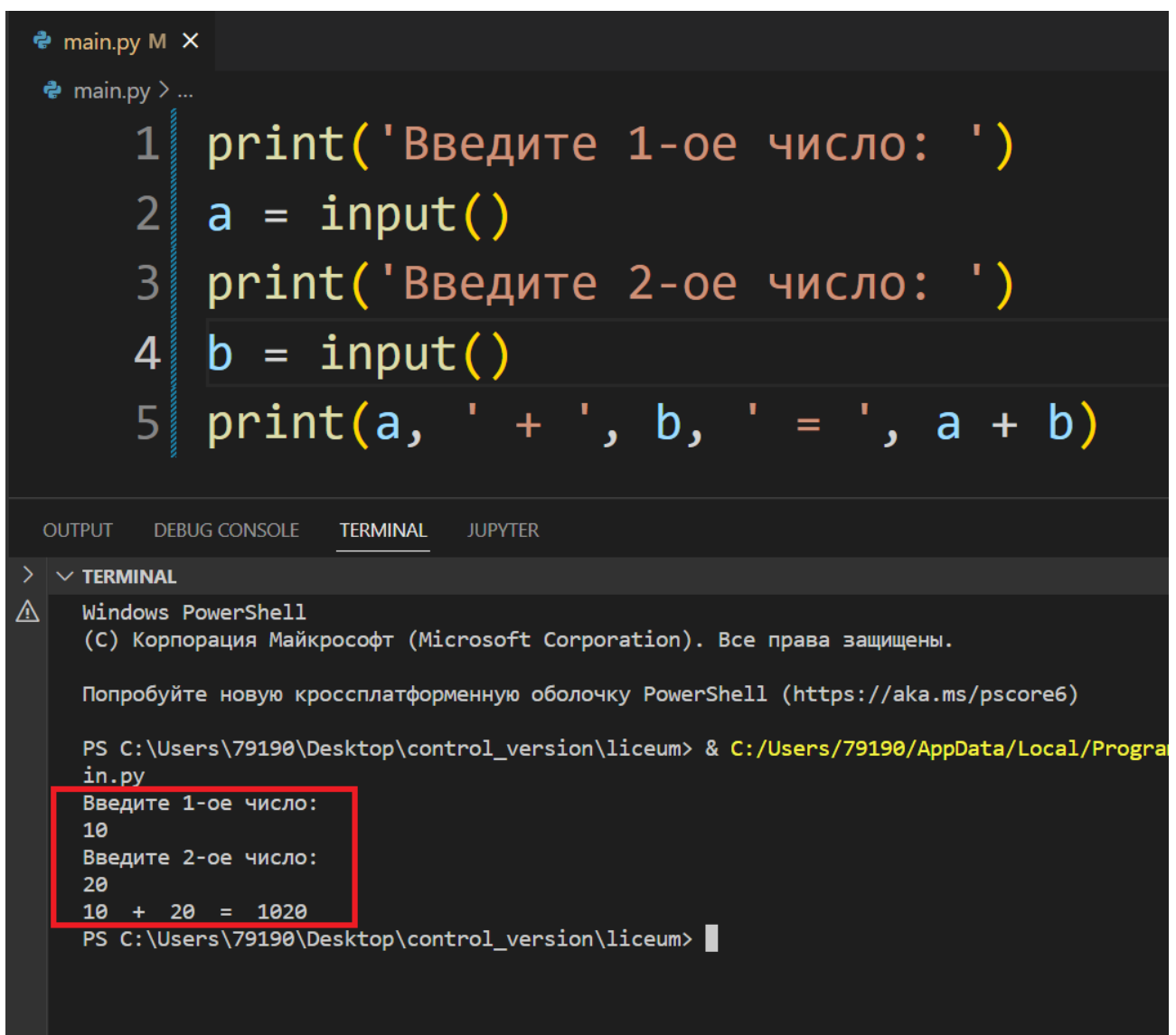
Оператор ввода данных

Как и в любом языке программирования, у Python есть операторы ввода данных. Не все так просто, как может показаться :)

- `input()` — ввод данных(строка)

Функция **`input()`** вводит строку, но тогда как ввести число? Давайте разбираться.

Как показать сумму двух чисел?



The screenshot shows a Jupyter Notebook interface with a code editor and a terminal output pane. The code editor contains the following Python code:

```
1 print('Введите 1-ое число: ')\n2 a = input()\n3 print('Введите 2-ое число: ')\n4 b = input()\n5 print(a, ' + ', b, ' = ', a + b)
```

The terminal output pane shows the execution of the code in a Windows PowerShell environment. The output is as follows:

```
Windows PowerShell\n(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.\n\nПопробуйте новую кроссплатформенную оболочку PowerShell (https://aka.ms/pscore6)\n\nPS C:\\Users\\79190\\Desktop\\control_version\\liceum> & C:/Users/79190/AppData/Local/Programs/Python/Python39-32/python.exe in.py\nВведите 1-ое число:\n10\nВведите 2-ое число:\n20\n10 + 20 = 1020\nPS C:\\Users\\79190\\Desktop\\control_version\\liceum>
```

The input and output of the program are highlighted with a red box in the terminal output.

Когда мы ввели 2 числа(a, b). В переменных находились на самом деле не числа, а строки: `a = '10'` `b = '20'`

Поэтому при сложении получился такой результат: строки соединились.

Так как по умолчанию с помощью функции `input()` вводится строка, необходимо воспользоваться вспомогательными функциями, которые позволят вводить числа и работать с ними.

Встроенные типы

- `int()` - функция, которая позволяет перевести из любого типа данных в число(если это возможно)

Программный код:

```
n = 1.345
print(int(n)) # Отбрасывается дробная часть вне зависимости больше 0.5 или
меньше

m = '345'
print(m * 2) # При умножении строки на число, она повторяется столько раз на
какое была умножена
print(int(m) * 2)
```

Вывод:

```
1
345345
690
```

- `str()` - функция, которая позволяет перевести из любого типа данных в строку(если это возможно)

Программный код:

```
n = 1.345
print(str(n) * 2)
```

Вывод:

```
1.3451.345
```

- `float()` - функция, которая позволяет перевести из любого типа данных в вещественный(если это возможно)

Программный код:

```
n = '1.345'
print(float(n) * 2)

m = 2
print(float(m))
```

Вывод:

```
2.69
2.0
```

Примечание:

Иногда все-таки нельзя перевести один тип данных в другой.

Программный код:

```
n = '123Hello'
print(int(n))
print(float(n))
```

Вывод:

```
ValueError: invalid literal for int() with base 10: '123Hello'
```

Это ошибка типа данных. Невозможно сделать число из строки.

Познакомившись, с функциями **`int()`**, **`float()`**, **`str()`**, пора бы поговорить нам о том, как все-таки ввести число в Python?

```
n = int(input()) # 5
print(n * 2) # 10
```

Арифметические операции

Давайте посмотрим какой синтаксис в Python у базовых арифметических операций



Без них Вы не напишете ни одной программы.

Знак операции	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление (по умолчанию в вещественных числах)
%	Остаток от деления
//	Целочисленное деление
**	Возведение в степень

Приоритет арифметических операций

1. Возведение в степень (**)
2. Умножение (*)
3. Деление (/)
4. Целочисленное деление (//)
5. Остаток от деления (%)
6. Сложение (+)
7. Вычитание (-)



В Python нет лимита по хранению данных (нет ограничения по битам для хранения числа из-за динамической типизации данных)

Округление числа

Можно указать количество знаков после запятой

```
a = 1.43425
b = 2.2983
c = round(a * b, 5) # 3,29633
```

Сокращенные операции присваивания

Помните в C# внутри цикла for мы писали `i++`. Это было сокращение от `i = i + 1`.
Посмотри как можно сокращать операторы присваивания в Python

```
iter = 2
iter += 3 # iter = iter + 3
iter -= 4 # iter = iter - 4
iter *= 5 # iter = iter * 5
iter /= 5 # iter = iter / 5
iter //= 5 # iter = iter // 5
iter %= 5 # iter = iter % 5
iter **= 5 # iter = iter ** 5
```

Логические операции

Знак операции	Операция
>	Больше
>=	Больше или равно
<	Меньше
<=	Меньше или равно
==	Равно (проверяет, равны ли числа)
!=	Не равно (проверяет, не равны ли значения)
not	Не (отрицание)
and	И (конъюнкция)
or	Или (дизъюнкция)

Кое-что ещё: is, is not, in, not in

В Python мы можем выполнять следующие сравнения. Результатом чего будет либо True, либо False

```
a = 1 > 4
print(a) # False
```

```
a = 1 < 4 and 5 > 2
print(a) # True
```

```
a = 1 == 2
print(a) # False
```

```
a = 1 != 2  
  
print(a) # True
```

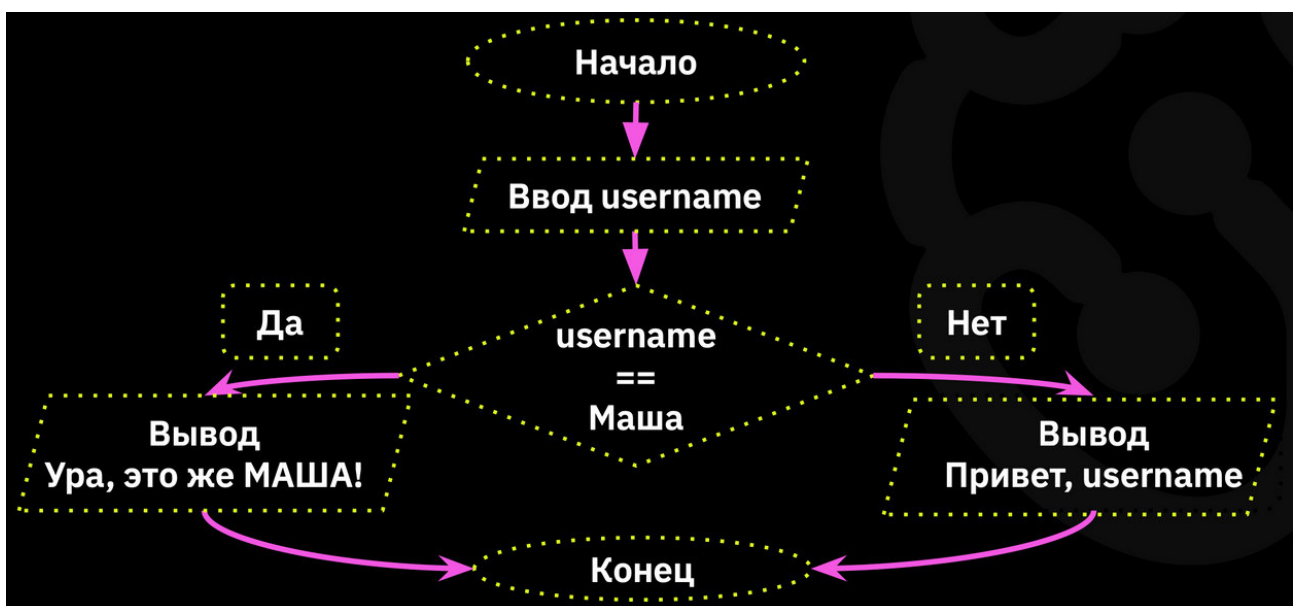
Можно сравнивать не только числовые значения, но и строки:

```
a = 'qwe'  
b = 'qwe'  
  
print(a == b) # True
```

В Python можно использовать тройные и даже четверные неравенства:

```
a = 1 < 3 < 5 < 10  
  
print (a) # True
```

Управляющие конструкции: if, if-else



Отступы в Python

Отступы в Python играют огромную роль, стоит Вам поставить на 1 пробел меньше, чем нужно, Ваша программа будет не рабочая.

Отступом отделяется блок кода, который находится внутри операторов ветвления, циклов, функций и тд. Обычно внутри VSC отступы ставятся автоматически, но Вы должны знать чему равны отступы:

- Кнопка TAB
- или
- 4 пробела

Но необязательно это кнопка **TAB** или **4** пробела, можно настроить чему равен отступ, как Вам больше нравится, мы же будем использовать вариант, который описан Выше.

Пример оформления программного кода с операторами ветвления:

```
if condition:
    # operator 1
    # operator 2
    # ...
    # operator n
else:
    # operator n + 1
    # operator n + 2
    # ...
    # operator n + m
```

```
a = int(input("a = "))
b = int(input("b = "))
if a > b:
    print(a)
else:
    print(b)
```

Ещё один вариант использования операторов else-if → в связке с elif (else if)

Проверяем первое условие, если оно не выполняется, проверяем второе и так далее. Как только будет найдено верное условие, все остальные будут игнорироваться.

```
if condition1:
    # operator
elif condition2:
    # operator
elif condition3:
    # operator
else:
    # operator
```

Пример программного кода:

```
username = input('Введите имя: ')
if username == 'Маша':
    print('Ура, это же МАША!')
elif username == 'Марина':
    print('Я так ждала Вас, Марина!')
elif username == 'Ильнар':
    print('Ильнар - топ')
else:
    print('Привет, ', username)
```

Сложные условия

Сложные условия создаются с помощью логических операторов, таких как: **and**, **or**, **not**

```
if condition1 and condition2: # выполнится, когда оба условия окажутся верными
    # operator

if condition3 or condition4: # выполнится, когда хотя бы одно из условий
    окажется верным
    # operator
```

```
n = int(input())

if n % 2 == 0 and n % 3 == 0:
    print('Число кратно 6')

if n % 5 == 0 and n % 3 == 0:
    print('Число кратно 15')
```

Управляющие конструкции: while и вариация while-else



Цикл позволяет выполнить блок кода, пока условие является верным.

Пример программного кода:

```
while condition:
    # operator 1
    # operator 2
    # ...
    # operator n
```

```
n = 423
summa = 0
while n > 0:
    x = n % 10
    summa = summa + x
    n = n // 10
print(summa) # 9
```

Управляющие конструкции: while-else

```
while condition:
    # operator 1
    # operator 2
    # ...
    # operator n
else:
    # operator n + 1
    # operator n + 2
    # ...
    # operator n + m
```

Блок `else` выполняется, когда основное тело цикла перестает работать **самостоятельно**. А разве кто-то может прекратить работу цикла? Если мы вспомним С#, то да и это конструкция `break`. Внутри Python она также существует и используется точно также. Пример:

```
i = 0
while i < 5:
    if i == 3:
        break
    i = i + 1
else:
    print('Пожалуй')
    print('хватит ')
print(i)
```

После выполнения данного кода в консоль выведется только цифра 3, то что находится внутри `else` будет игнорироваться, так как цикл завершился не самостоятельно.

Пример программного кода без использования break:

```
n = 423
summa = 0
while n > 0:
    x = n % 10
    summa = summa + x
    n = n // 10
else:
    print('Пожалуй')
    print('хватит ')
print(summa)
# Пожалуй
# хватит )
# 9
```

После того, как мы с Вами обговорили оператор **break** и цикл **while**, стоит рассказать почему не стоит использовать **break** и как в этом случае нам поможет Булевский тип данных? Давайте разбираться. **break** отличная конструкция, которую нельзя **не** использовать в некоторых алгоритмах, но break не функциональный стиль программирования. В ООП нет ничего, что предполагает **break** внутри метода-плохая идея, так как может произойти путаница. На замену **break** отлично подходит метод флажка.

Задача: Пользователь вводит число, необходимо найти минимальный делитель данного числа

```
n = int(input())

flag = True

i = 2
while flag:

    if n % i == 0: # если остаток при делении числа n на i равен 0

        flag = False

        print(i)

    elif i > n // 2: # делить числа не может превышать введенное число, деленное
на 2

        print(n)

        flag = False

    i += 1
```

Данный алгоритм будет работать до тех пор, пока не найдется минимальный делитель введенного числа. Когда будет найден первый делитель цикл остановит свою работу, так как условие, которое находится внутри станет ложным(False)

Цикл for, range

В Python цикл for в основном используется для перебора значений

Пример использования цикла for:

```
for i in enumeration:
    # operator 1
    # operator 2
    # ...
    # operator n

for i in 1, -2, 3, 14, 5:
    print(i)
# 1 -2 3 15 5
```

Range

- Range выдает значения из диапазона с шагом 1.
- Если указано только одно число — от 0 до заданного числа.
- Если нужен другой шаг, третьим аргументов можно задать приращение.

```
r = range(5) # 0 1 2 3 4
r = range(2, 5) # 2 3 4
r = range(-5, 0) # ----
r = range(1, 10, 2) # 1 3 5 7
r = range(100, 0, -20) # 100 80 60 40 20
r = range(100, 0, -20) # range(100, 0, -20)

for i in r:
    print(i)
# 100 80 60 40 20
```

```
for i in range(5):  
  
    print(i)  
  
# 0 1 2 3 4
```

Можно использовать цикл **for()** и со строками, так как у строк есть нумерация, такая же как и у массивов, начинается с 0:

```
for i in 'qwerty':  
  
    print(i)  
  
# q  
  
# w  
  
# e  
  
# r  
  
# t  
  
# y
```

Можно использовать вложенные циклы:

```
line = ""  
  
for i in range(5):  
  
    line = ""  
  
    for j in range(5):  
  
        line += "*"   
  
    print(line)
```

Программный код выведет 5 строк "*****". Сначала запускается внешний цикл с i(счетчик цикла). После этого запускается внутренний цикл с j(счетчик цикла). После того как внутренний цикл завершил свою работу, переменная line = "*****" и выводится на экран, далее опять повторяется внешний цикл и так 5 раз.

Немного о строках

Возникают ситуации, когда в некоторых задачах необходимо поработать со строкой, которую ввел пользователь. Например: необходимо сделать все буквы маленькими, или поменять все буквы “ё” на “е”.

Команды для работы со строками:

```
text = 'СъЕШЬ ещё этих МяГкИх французских булок'
```

1. Определить количество символов в строке:

```
print(len(text)) # 39
```

2. Проверить наличие символа в строке (in):

```
print('ещё' in text) # True
```

3. Функция, которая делает все буквы строки маленькими:

```
print(text.lower()) # съешь ещё этих мягких французских булок
```

4. Функция, которая делает все буквы строки большими:

```
print(text.upper()) # СЪЕШЬ ЕЩЁ ЭТИХ МЯГКИХ ФРАНЦУЗСКИХ БУЛОК
```

5. Заменить слово на другое :

```
print(text.replace('ещё', 'ЕЩЁ')) # СЪЕШЬ ЕЩЁ этих МяГкИх французских булок
```



`help(название функции)` — встроенная справка о функции.

Срезы

- Мы представляем строку в виде массива символов. Значит мы можем обращаться к элементам по индексам.
- Отрицательное число в индексе — счёт с конца строки

```
text = 'съешь ещё этих мягких французских булок'

print(text[0])           # с
print(text[1])           # ъ
print(text[len(text)-1]) # к
print(text[-5])          # б
print(text[:])           # съешь ещё этих мягких французских булок
print(text[:2])          # съ
print(text[len(text)-2:]) # ок
print(text[2:9])         # ешь ещё
print(text[6:-18])       # ещё этих мягких
print(text[0:len(text):6]) # сеикакл
print(text[::6])         # сеикакл
text = text[2:9] + text[-5] + text[:2] # ...
```

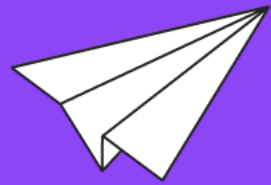
Итоги:

1. Изучили историю создания языка программирования Python и проанализировали востребованность на рынке
2. Познакомились с функциями ввода и вывод данных
3. Изучили операторы ветвления
4. Изучили циклы внутри Python и проговорили отличия с цикла на C#

На следующей лекции мы пройдем коллекции данных в Python, а также пройдем очень важную тему “Профилирование и отладка”, именно эта тема позволить Вам анализировать программный код.

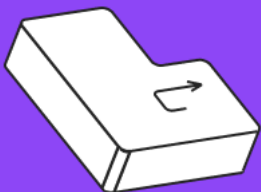
Дополнительный материалы:

1. Как установить Pycharm? - <https://it-robionek.ru/pt/pycharm-ustanovka.html>
2. Грокаем Алгоритмы - https://en.ecomed.dgmu.ru/wp-content/uploads/2020/01/Грокаем_алгоритмы.pdf



Коллекции данных. Профилирование и отладка.

Курс



Оглавление

Термины, используемые в лекции	2
Списки	2
List Comprehension	10
Профилирование и отладка	12

Термины, используемые в лекции

Кортеж — это неизменяемый список.

Словари — неупорядоченные коллекции произвольных объектов с доступом по ключу.

Коллекции данных, что это такое и для чего они нужны?

Коллекция — структура данных, которая создана, чтобы содержать в себе некоторое количество объектов (они могут быть одного типа или могут быть разных типов).

Рано или поздно переменную становятся не такими удобными, как могло показаться на первый взгляд. Например: **Необходимо записать имя каждого студента на потоке 1111**. И так, как мы будем решать эту задачу? На потоке около 5000 студентов и заводить для каждого человека отдельную переменную не очень разумное решение, в этом случае нам помогает такой тип данных, как List(список)

Списки

Список - это упорядоченный конечный набор элементов. Давайте разбираться, по сути список - это тот же самый массив, в котором можно хранить элементы любых типов данных.

Как работать со списками?

```
list_1 = [] # Создание пустого списка  
list_2 = list() # Создание пустого списка  
list_1 = [7, 9, 11, 13, 15, 17]
```

В списках существует нумерация, которая начинается с 0, чтобы вывести первый элемент списка воспользуемся следующей конструкцией:

```
list_1 = [7, 9, 11, 13, 15, 17]  
print(list_1[0]) # 7
```

Чтобы узнать количество элементов в списке необходимо использовать функцию **len(имя_списка)**:

```
list_1 = [7, 9, 11, 13, 15, 17]  
print(len(list_1)) # 6
```

Можно список заполнять не только при его создание, но и во время работы программы:

```
list_1 = list() # создание пустого списка  
for i in range(5): # цикл выполнится 5 раз  
    n = int(input()) # пользователь вводит целое число  
    list_1.append(n) # сохранение элемента в конец списка  
# 1-я итерация цикла(повторение 1): n = 12, list_1 = [12]  
# 2-я итерация цикла(повторение 2): n = 7, list_1 = [12, 7]  
# 3-я итерация цикла(повторение 3): n = -1, list_1 = [12, 7, -1]  
# 4-я итерация цикла(повторение 4): n = 21, list_1 = [12, 7, -1, 21]
```

```
# 5-я итерация цикла (повторение 5): n = 0, list_1 = [12, 7, -1, 21, 0]
print(list_1) # [12, 7, -1, 21, 0]
```

Мы обговорили с Вами создание списка и поняли, что мы можем пользоваться нумерацией, для того чтобы узнать какой элемент стоит на той или иной позиции. Но это не всегда удобно, особенно, когда список будет состоять из 1000, 1000000... элементов. В этом случае необходимо использовать цикл **for**.

Взаимодействие цикла for со списком:

```
list_1 = [12, 7, -1, 21, 0]
for i in list_1:
    print(i) # вывод каждого элемента списка

# 1-я итерация цикла (повторение 1): i = 12
# 2-я итерация цикла (повторение 2): i = 7
# 3-я итерация цикла (повторение 3): i = -1
# 4-я итерация цикла (повторение 4): i = 21
# 5-я итерация цикла (повторение 5): i = 0
```

Не забываем, что у списка есть нумерация:

```
list_1 = [12, 7, -1, 21, 0]
for i in range(len(list_1)):
    print(list_1[i]) # вывод каждого элемента списка

# 1-я итерация цикла (повторение 1): list_1[0] = 12
# 2-я итерация цикла (повторение 2): list_1[1] = 7
# 3-я итерация цикла (повторение 3): list_1[2] = -1
# 4-я итерация цикла (повторение 4): list_1[3] = 21
# 5-я итерация цикла (повторение 5): list_1[4] = 0
```

Основные действия со списками:

1. Удаление последнего элемента списка.

Метод pop удаляет последний элемент из списка:

```
list_1 = [12, 7, -1, 21, 0]

print(list_1.pop()) # 0

print(list_1) # [12, 7, -1, 21]

print(list_1.pop()) # 21

print(list_1) # [12, 7, -1]

print(list_1.pop()) # -1

print(list_1) # [12, 7]
```

2. Удаление конкретного элемента из списка.

Надо указать значение индекса в качестве аргумента функции pop:

```
list_1 = [12, 7, -1, 21, 0]

print(list_1.pop(0)) # 12

print(list_1) # [7, -1, 21, 0]
```

3. Добавление элемента на нужную позицию.

Функция insert — указание индекса (позиции) и значения.

```
list_1 = [12, 7, -1, 21, 0]

print(list_1.insert(2, 11))

print(list_1) # [12, 7, 11, -1, 21, 0]
```

Срез списка

Помните в конце первой лекции Вы прошли срезы строк? Также существует срез списка, давайте научимся изменять наш список

- Отрицательное число в индексе — счёт с конца списка

```
list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(list_1[0])           # 1
print(list_1[1])           # 2
print(list_1[len(list_1)-1]) # 10
print(list_1[-5])          # 6
print(list_1[:])           # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(list_1[:2])          # [1, 2]
print(list_1[len(list_1)-2:]) # [9, 10]
print(list_1[2:9])         # [3, 4, 5, 6, 7, 8, 9]
print(list_1[6:-18])       # []
print(list_1[0:len(list_1):6]) # [1, 7]
print(list_1[::6])         # [1, 7]
```

Кортежи



Кортеж — это неизменяемый список.

Тогда для чего нужны кортежи, если их нельзя изменить? В случае защиты каких-либо данных от изменений (намеренных или случайных). Кортеж занимает **меньше места в памяти и работают быстрее**, по сравнению со списками

```
t = () # создание пустого кортежа

print(type(t)) # class <'tuple'>

t = (1,)

print(type(t))

t = (1)

print(type(t))

t = (28, 9, 1990)

print(type(t))

colors = ['red', 'green', 'blue']

print(colors)          # ['red', 'green', 'blue']

t = tuple(colors)

print(t)                # ('red', 'green', 'blue')

t = tuple(['red', 'green', 'blue'])

print(t[0]) # red

print(t[2]) # blue

for e in t:

    print(e) # red green blue

t[0] = 'black' # TypeError: 'tuple' object does not support (нельзя изменять кортеж)
```


Можно распаковать кортеж в независимые переменные:

```
t = tuple(['red', 'green', 'blue'])

red, green, blue = t

print('r:{} g:{} b:{}'.format(red, green, blue)) # r:red g:green b:blue
```

Словари

 Словари — неупорядоченные коллекции произвольных объектов с доступом по ключу.

В списках в качестве ключа используется индекс элемента. В словаре для определения элемента используется значение ключа (строка, число).

```
dictionary = {}

dictionary = {'up': '↑', 'left': '←', 'down': '↓', 'right': '→'}

print(dictionary) # {'up':'↑', 'left':'←', 'down':'↓', 'right':'→'}

print(dictionary['left']) # ←

# типы ключей могут отличаться

print(dictionary['up']) # ↑

# типы ключей могут отличаться

dictionary['left'] = '⇐'

print(dictionary['left']) # ⇐

print(dictionary['type']) # KeyError: 'type'

del dictionary['left'] # удаление элемента

for item in dictionary: # for (k,v) in dictionary.items():

    print('{}: {}'.format(item, dictionary[item]))

# up: ↑

# down: ↓

# right: →
```

Множества



Множества содержат в себе уникальные элементы, не обязательно упорядоченные.

Одно множество может содержать значения любых типов. Если у Вас есть два множества, Вы можете совершать над ними любые стандартные операции, например, объединение, пересечение и разность. Давайте разберем их.

```
colors = {'red', 'green', 'blue'}

print(colors) # {'red', 'green', 'blue'}

colors.add('red')

print(colors) # {'red', 'green', 'blue'}

colors.add('gray')

print(colors) # {'red', 'green', 'blue', 'gray'}

colors.remove('red')

print(colors) # {'green', 'blue', 'gray'}

colors.remove('red') # KeyError: 'red'

colors.discard('red') # ok

print(colors) # {'green', 'blue', 'gray'}

colors.clear() # { }

print(colors) # set()
```

Операции со множествами в Python

```
a = {1, 2, 3, 5, 8}

b = {2, 5, 8, 13, 21}

c = a.copy() # c = {1, 2, 3, 5, 8}

u = a.union(b) # u = {1, 2, 3, 5, 8, 13, 21}

i = a.intersection(b) # i = {8, 2, 5}

dl = a.difference(b) # dl = {1, 3}

dr = b.difference(a) # dr = {13, 21}

q=a.union(b).difference(a.intersection(b)) # {1, 21, 3, 13}
```


Неизменяемое или замороженное множество(frozenset) — множество, с которым не будут работать методы удаления и добавления.

```
a = {1, 2, 3, 5, 8}
b = frozenset(a)
print(b) # frozenset({1, 2, 3, 5, 8})
```

Обобщение свойств встроенных коллекций в сводной таблице:

Тип коллекции	Изменяемость	Индексированность	Уникальность	Как создаём
Список (list)	+	+	-	<code>[]</code> <code>list()</code>
Кортеж (tuple)	-	+	-	<code>()</code> , <code>tuple()</code>
Строка (string)	-	+	-	<code>"</code> <code>" "</code>
Множество (set)	+	-	+	<code>{elm1, elm2}</code> <code>set()</code>
Неизменяемое множество (frozenset)	-	-	+	<code>frozenset()</code>
Словарь (dict)	+ элементы - ключи + значения	-	+ элементы + ключи - значения	<code>{}</code> <code>{key: value,}</code> <code>dict()</code>

List Comprehension

У каждого языка программирования есть свои особенности и преимущества. Одна из культовых фишек Python — list comprehension (редко переводится на русский, но можно использовать определение «генератора списка»). Comprehension легко читать, и их используют как начинающие, так и опытные разработчики.

List comprehension — это упрощенный подход к созданию списка, который задействует цикл for, а также инструкции **if-else** для определения того, что в итоге окажется в финальном списке.

1. Простая ситуация — список:

```
list_1 = [exp for item in iterable]
```

2. Выборка по заданному условию:

```
list_1 = [exp for item in iterable (if conditional)]
```

Задача: Создать список, состоящий из четных чисел в диапазоне от 1 до 100.

Решение:

1. Создать список чисел от 1 до 100

```
list_1 = []  
for i in range(1, 101):  
    list_1.append(i)  
print(list_1) # [1, 2, 3, ..., 100]
```

Эту же функцию можно записать так:

```
list_1 = [i for i in range(1, 101)] # [1, 2, 3, ..., 100]
```

2. Добавить условие (только чётные числа)

```
list_1 = [i for i in range(1, 101) if i % 2 == 0] # [2, 4, 6, ..., 100]
```

Допустим, вы решили создать пары каждому из чисел (кортежи)

```
list_1 = [(i, i) for i in range(1, 101) if i % 2 == 0] # [(2, 2), (4, 4), ..., (100, 100)]
```

Также можно умножать, делить, прибавлять, вычитать. Например, умножить значение на 2.

```
list_1 = [i * 2 for i in range(10) if i % 2 == 0]
print(list_1) # [0, 4, 8, 12, 16]
```

Профилирование и отладка

Мы с вами люди, а людям суждено ошибаться, даже при написании программного кода 😊

Давайте разберем с Вами самые частые ошибки в написании кода на Python.



Самые распространенные ошибки:

- `SyntaxError`(Синтаксическая ошибка)

```
number_first = 5
number_second = 7
if number_first > number_second # !!!!!
    print(number_first)

# Отсутствие :
```

- IndentationError(Ошибка отступов)

```
number_first = 5
number_second = 7
if number_first > number_second:
print(number_first) # !!!!!

# Отсутствие отступов
```

- TypeError(Типовая ошибка)

```
text = 'Python'
number = 5
print(text + number)

# Нельзя складывать строки и числа
```

- ZeroDivisionError(Деление на 0)

```
number_first = 5
number_second = 0
print(number_first // number_second)

# Делить на 0 нельзя
```

- KeyError(Ошибка ключа)

```
dictionary = {1: 'Monday', 2: 'Tuesday'}
print(dictionary[3])

# Такого ключа не существует
```

- `NameError`(Ошибка имени переменной)

```
name = 'Ivan'

print(names)

# Переменной names не существует
```

- `ValueError`(Ошибка значения)

```
text = 'Python'

print(int(text))

# Нельзя символы перевести в целые значения
```

Итоги:

1. Мы изучили коллекции данных:

- *списки*
- *кортежи*
- *словари*
- *множества*

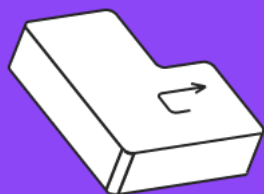
2. List Comprehension

3. Профилирование и отладка



Функции, рекурсия, алгоритмы.

Курс



Оглавление

Термины, используемые в лекции	2
Функции и модули	2
Алгоритмы	5
Что можно почитать ещё	9

Термины, используемые в лекции

Функция — это фрагмент программы, используемый многократно:

Alias (псевдоним) — альтернативное имя, которое даётся функции при её импорте из файла.

Рекурсия — это функция, вызывающая сама себя.

Функции и модули



Функция — это фрагмент программы, используемый многократно

Мы знакомы уже с функциями с C#, давайте теперь посмотрим, как создаются и используются функции внутри Python.

```
def function_name(x):  
    # body line 1  
    # ...  
    # body line n  
    # optional return
```

Задание: Необходимо создать функцию **sumNumbers(n)**, которая будет считать сумму всех элементов от 1 до n.

Решение:

1. Необходимо создать функцию:

```
def sumNumbers(n):
```

Очень важно понимать одну вещь, сколько аргументов мы передаем, столько и принимаем. Или наоборот сколько аргументов мы принимаем, столько и передаем. В нашем случае функция **sumNumbers** принимает 1 аргумент(n).

2. Реализовать решение задачи внутри функции

```
def sumNumbers(n):  
    summa = 0  
    for i in range(1, n + 1):  
        summa += i  
    print(summa)
```

Так как нам нужны все значения из промежутка [1, n], мы вызываем функцию **range**, от 1 до n + 1, так как range **не** включает последний элемент. **НО** возникает вопрос, почему мы здесь используем **print()** ? Где же всем известный **return**, которым мы пользовались на C#?

На самом деле, нам не нужно теперь писать **void**, для того чтобы выводить данные, а для того чтобы возвращать значения ставить их тип, все намного проще. В Python нет такого понятия как процедура(void). Здесь существует только **def**. Вернемся к задаче

3. Спросим у пользователя число

```
def sumNumbers(n):  
    summa = 0  
    for i in range(1, n + 1):  
        summa += i  
    print(summa)  
  
n = int(input()) # 5  
sumNumbers(n) # 15
```

Программный код, который мы написали прекрасно справляется с поставленной задачей. Давайте изменим наш код и добавим в него **return**. **НО** перед этим давайте вспомним, что делает **return**:

1. Завершает работу функции
2. Возвращает значение

```
def sumNumbers(n):  
    summa = 0  
    for i in range(1, n + 1):  
        summa += i  
    return summa  
  
n = int(input()) # 5  
print(sumNumbers(n)) # 15
```

Модульность

Вы когда-нибудь задавались вопросом, как например работает функция **.append** Это же точно такая же функция, как и **sumNumbers(n)**, но мы ее нигде не создаем, все дело в том что, это функция автоматически срабатывает и чтобы ей пользоваться ничего дополнительно писать не надо.

Представьте себе такую ситуацию, что Вы создаете огромный проект и у Вас имеется большое количество функций, к примеру 5 функций работают со словарями, 18 со списками и тд. и у каждой функции свой алгоритм, но их объединяет работа с одной коллекцией данных. Согласитесь неудобно работать в таком большом файле, где около 80 функций, очень легко потеряться и на перемутку кода Вы будете терять драгоценное время. Решение данной проблемы есть. Давай будем создавать отдельные файлы, где будут находиться только функции, и эти функции при необходимости вызывать из главного файла.

1. **function_file.py** (Новый Python файл, в котором находятся функция f(x))

```
def f(x):  
    if x == 1:  
        return 'Целое'  
    elif x == 2.3:  
        return 23  
    return # выход из функции
```

2. **working_file.py**

Чтобы начать взаимодействовать с функцией в файле **function_file.py** необходимо добавить эту возможность к себе в программный код. Сначала мы обращаемся к файлу(без расширения)

С помощью **import** мы можем вызвать эту функцию в другом скрипте и дальше использовать её в новом файле. Можно сократить название функции в рабочем файле с помощью команды:



Alias (псевдоним) — альтернативное имя, которое даётся функции при ет импорте из файла.

```
import function_file
```

```
print(function_file.f(1)) # Целое
print(function_file.f(2.3)) # 23
print(function_file.f(28)) # None
```

Значения по умолчанию для функции



В Python можно перемножать строку на число.

В данной функции есть два аргумента: ***symbol*** (символ или число) и ***count*** (число, на которое умножается первый аргумент).

Если введены оба аргумента, функция работает без ошибок. Если только символ — функция выдает ошибку.

```
def new_string(symbol, count):
    return symbol * count

print(new_string('!', 5)) # !!!!!
print(new_string('!')) # TypeError missing 1 required ...
```

Можно указать значение переменной `count` по умолчанию. Например, если значение явно не указано (нет второго аргумента), по умолчанию значение переменной `count` равно трем.

```
def new_string(symbol, count=3):
    return symbol * count

print(new_string('!', 5)) # !!!!!
print(new_string('!')) # !!!
print(new_string(4)) # 12
```

Возможность передачи неограниченного количества аргументов

- Можно указать любое количество значений аргумента функции.
- Перед аргументом надо поставить *.

В примере ниже функция работает со строкой, поэтому при введении чисел программа выдаёт ошибку:

```
def concatenatio(*params):  
    res = ""  
    for item in params:  
        res += item  
    return res  
  
print(concatenatio('a', 's', 'd', 'w')) # asdw  
print(concatenatio('a', '1')) # a1  
# print(concatenatio(1, 2, 3, 4)) # TypeError: ...
```

Рекурсия



Рекурсия — это функция, вызывающая сама себя.

С рекурсией Вы знакомы с С#, в Python она ничем не отличается, давай рассмотрим следующую **задачу**: Пользователь вводит число n . Необходимо вывести n - первых членов последовательности Фибоначчи.

Напоминание: Последовательно Фибоначчи, это такая последовательность, в которой каждое последующее число равно сумме 2-ух предыдущих



При описании рекурсии важно указать, когда функции надо остановиться и перестать вызывать саму себя. По-другому говоря, необходимо указать базис рекурсии

Решение :

```
def fib(n):  
    if n in [1, 2]:  
        return 1  
    return fib(n - 1) + fib(n - 2)  
  
list_1 = []  
for i in range(1, 10):  
    list_1.append(fib(i - 2))  
  
print(list_1) # [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Внутри функции `fib(n)`, мы сначала задаем базис, если число n равно 1 или 2, это означает, что первое число и второе число последовательности равны 1. Мы так и делаем, возвращаем 1. Как мы ранее проговорили: “Последовательно Фибоначчи, это такая последовательность, в которой каждое последующее число равно сумме 2-ух предыдущих”. Так и делаем, складываем на 2 предыдущих числа друг с другом и получаем 3.

Алгоритмы

Алгоритмом называется набор инструкций для выполнения некоторой задачи. В принципе, любой фрагмент программного кода можно назвать алгоритмом, но мы с Вами рассмотрим 2 самых интересных алгоритмы сортировок:

- Быстрая сортировка
- Сортировка слиянием

Быстрая сортировка

“Программирование это разбиение чего-то большого и невозможного на что-то маленькое и вполне реальное”

Быстрая сортировка принадлежит такой стратегии, как “разделяй и властвуй”. Сначала рассмотрим пример, затем напишем программный код

Два друга решили поиграть в игру: один загадывает число от 1 до 100, другой должен отгадать. Согласитесь, что мы можем перебирать эти значения в случайном порядке, например: 32, 27, 60, 73... Да, мы можем угадать в какой-то момент, но что если мы обратимся к стратегии “разделяй и властвуй” Обозначим друзей, друг_1 это Иван, который загадал число, друг_2 это Петр, который отгадывает. Итак начнем:

Иван загадал число **77**.

Петр: Число больше 50? Иван: Да.

Петр: Число больше 75? Иван: Да.

Петр: Число больше 87? Иван: Нет.

Петр: Число больше 81? Иван: Нет.

Петр: Число больше 78? Иван: Нет.

Петр: Число больше 76? Иван: Да

Число оказалось в диапазоне $76 < x < 78$, значит это число 77. Задача решена. На самом деле мы сейчас познакомились с алгоритмом бинарного поиска, который также принадлежит стратегии “разделяй и властвуй”. Давайте перейдем к обсуждению программного кода быстрой сортировки.

```
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10, 5, 2, 3]))
```

- **1-е** повторение рекурсии:
 - array = [10, 5, 2, 3]
 - pivot = 10
 - less = [5, 2, 3]
 - greater = []
 - return quicksort([5, 2, 3]) + [10] + quicksort([])
- **2-е** повторение рекурсии:
 - array = [5, 2, 3]
 - pivot = 5
 - less = [2, 3]
 - greater = []
 - return quicksort([2, 3]) + [5] + quicksort([]) # Важно! Не забывайте, что здесь помимо вызова рекурсии добавляется список [10]
- **3-е** повторение рекурсии:
 - array = [2, 3]
 - return [2, 3] # Сработал базовый случай рекурсии

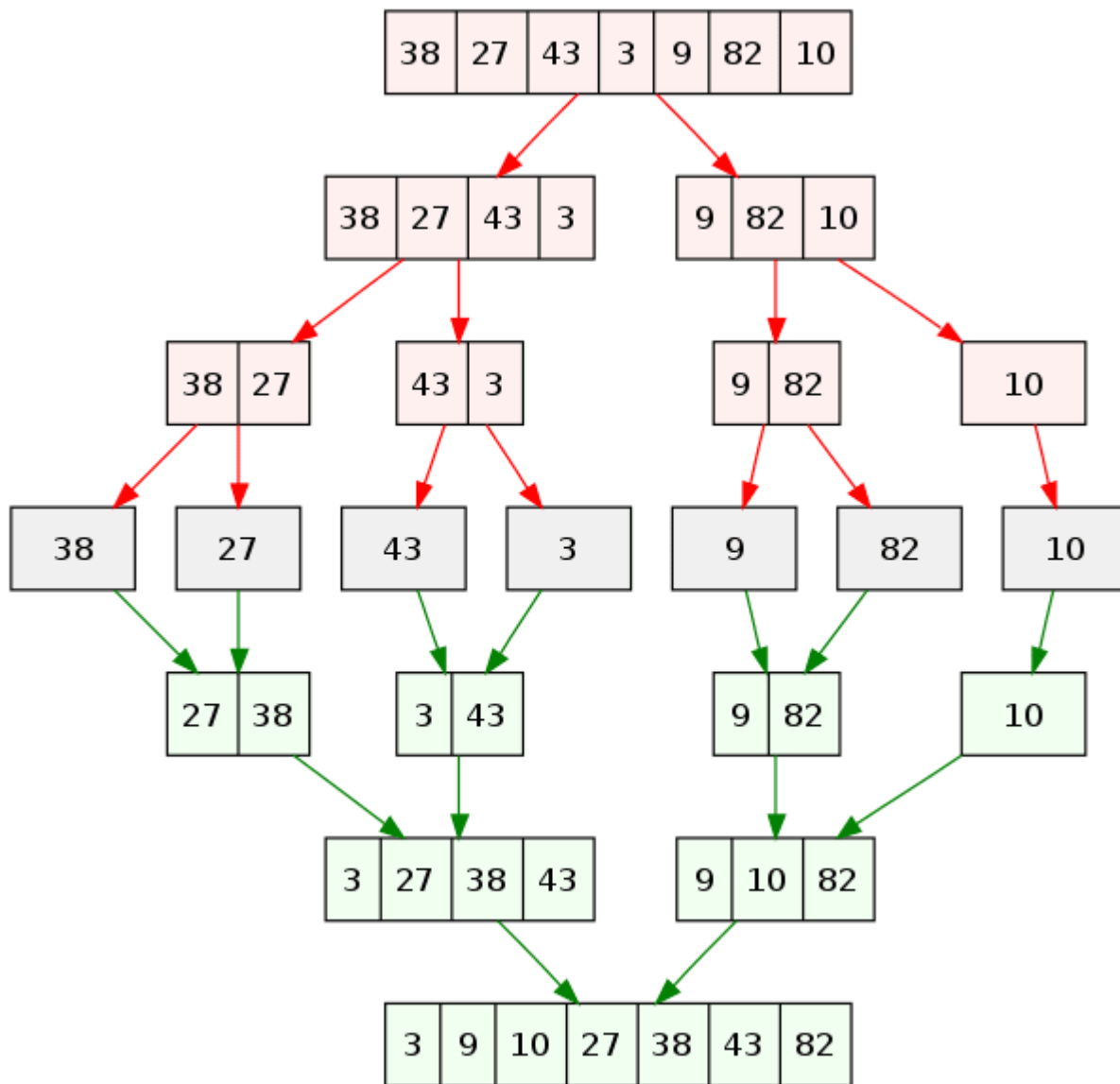
На этом работа рекурсии завершилась и итоговый список будет выглядеть таким образом: [2, 3] + [5] + [10] = [2, 3, 5, 10]

Сортировка слиянием

```
def merge_sort(nums):  
    if len(nums) > 1:  
        mid = len(nums) // 2  
        left = nums[:mid]  
        right = nums[mid:]  
        merge_sort(left)  
        merge_sort(right)  
        i = j = k = 0  
        while i < len(left) and j < len(right):  
            if left[i] < right[j]:  
                nums[k] = left[i]  
                i += 1  
            else:  
                nums[k] = right[j]  
                j += 1  
            k += 1  
        while i < len(left):  
            nums[k] = left[i]  
            i += 1  
            k += 1  
        while j < len(right):  
            nums[k] = right[j]  
            j += 1  
            k += 1
```

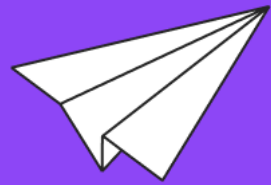


```
nums = [38, 27, 43, 3, 9, 82, 10]
merge_sort(nums)
print(nums)
```



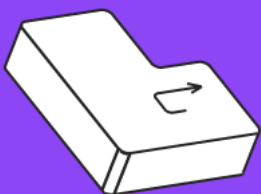
Итоги:

1. *Создание и использование функций*
2. *Рекурсию*
3. *Алгоритмы сортировок*



Функции высшего порядка, работа с файлами.

Курс



Оглавление

Анонимные, lambda-функции	2
Задача для самостоятельного решения	5
Функция map	6
Функция filter	7
Функция zip	8
Функция enumerate	9
Файлы	9
Модуль os	12
Модуль shutil	13

Анонимные, lambda-функции

На предыдущей лекции мы с Вами обговорили создание и использование функций в Python, но некоторые функции могут понадобиться всего раз за всю работу приложения. Как можно обойтись без их явного описания? Как сократить подобный код?

```
def f(x)
    return x ** 2

print(f(2))
```



Функция в примере занимает всего две строчки кода, но в дальнейшем размеры описания функций будут увеличиваться. И тогда сокращение кода будет актуальным.

Какой тип данных у функции? → <class “function”>

У функции есть тип, значит мы можем создать переменную типа функции и положить в эту переменную какую-то другую функцию.

```
def f(x)
    return x ** 2

g = f
```

Теперь в контексте этого приложения **g** может использоваться точно так же, как и **f**.

g — это переменная, которая хранит в себе ссылку на функцию.

```
def f(x)
    return x ** 2

g = f

print(f(4)) # 16

print(g(4)) # 16
```

Зачем это может потребоваться?

Есть некая функция **calc**, которая принимает в качестве аргумента число, а в качестве результата возвращает это число + 10:

```
def calc1(x)
    return x + 10

print(calc1(10)) # 20
```

Если мы добавим в код не только сложение, но и умножение, деление и вычитание, внутри одного кода будем плодить одинаковую логику.



Достаточно взять функцию calc, которая будет в качестве аргумента принимать операцию и что-то выдавать.

```
def calc2(x)
    return x * 10

def math(op, x)
    print(op, x)

math(calc2, 10) # 100
```

Попробуем описать ту же логику для функции с двумя переменными.

op — операция, воспринимаем её как отдельную функцию. В примере это либо сумма (**sum**), либо перемножение(mylt):

```
def sum(x, y):
    return x + y

def mylt(x, y):
    return x * y
```

```
def calc(op, a, b):  
    print(op(a, b))  
  
calc(mylt, 4, 5) # 20
```

Можно создать псевдоним для функции сложения (f).

```
def sum(x, y):  
    return x + y  
  
f = sum  
  
calc(f, 4, 5) # 9
```

В Python есть механизм, который позволяет превратить подобный вызов во что-то более красивое — **lambda**.

```
def sum(x, y):  
    return x + y  
  
# ⇔ (равносильно)  
  
sum = lambda x, y: x + y
```

Теперь, чтобы вызвать функцию суммы, достаточно просто вызвать **sum**.

Также можно пропустить шаг создания переменной sum и сразу вызвать **lambda**:

```
calc(lambda x, y: x + y, 4, 5) # 9
```

Итак:

1. Сначала мы избавились от классического описания функций.
2. Затем научились описывать лямбды, присваивая результат какой-то переменной.
3. После избавились от этой переменной, пробрасывая всю лямбду в качестве функции.

Задача для самостоятельного решения

1. В списке хранятся числа. Нужно выбрать только чётные числа и составить список пар (число; квадрат числа).

Пример: 1 2 3 5 8 15 23 38

Получить: [(2, 4), (8, 64), (38, 1444)]

Решение:

```
data = [1, 2, 3, 5, 8, 15, 23, 38]
out = []
for i in data :
    if i % 2 == 0:
        out.append((i, i ** 2))
print(out)
```

Как можно сделать этот код лучше, используя lambda?

```
def select(f, col):
    return [f(x) for x in col]

def where(f, col):
    return [x for x in col if f(x)]

data = [1, 2, 3, 5, 8, 15, 23, 38]
res = select(int, data)
```



```
res = where(lambda x: x % 2 == 0, res)

print(res) # [2, 8, 38]

res = list(select(lambda x: (x, x ** 2), res))
```

Функция map

💡 Функция `map()` применяет указанную функцию к каждому элементу итерируемого объекта и возвращает итератор с новыми объектами.

Есть набор данных. Функция `map` позволяет увеличить каждый объект на 10.

$f(x) \Rightarrow x + 10$

`map(f, [1, 2, 3, 4, 5])`

↓ ↓ ↓ ↓ ↓

`[11, 12, 13, 14, 15]`

```
list_1 = [x for x in range (1,20)]

list_1 = list(map(lambda x: x + 10, list_1 ))

print(list_1)
```

Задача: С клавиатуры вводится некий набор чисел, в качестве разделителя используется пробел. Этот набор чисел будет считан в качестве строки. Как превратить list строк в list чисел?

1. Маленькое отступление, функция **строка.split()** - убирает все пробелы и создает список из значений строки, пример:

```
data = '1 2 3 5 8 15 23 38'.split()
print(data) # ['1', '2', '3', '5', '8', '15', '23', '38']
```

2. Теперь вернемся к задаче. С помощью функции **map()**:

```
data = list(map(int, input().split()))
```

Результат работы `map()` — это итератор. По итератору можно пробежаться только один раз. Чтобы работать несколько раз с одними данными, нужно сохранить данные (например, в виде списка).

Как можно сделать этот код лучше, используя `map()`?



`map()` позволит избавиться от функции `select`.

```
def where(f, col):
    return [x for x in col if f(x)]

data = '1 2 3 5 8 15 23 38'.split()

res = map(int, data)

res = where(lambda x: x % 2 == 0, res)
res = list(map(lambda x: (x, x ** 2), res))
print(res)
```

Функция filter



Функция `filter()` применяет указанную функцию к каждому элементу итерируемого объекта и возвращает итератор с теми объектами, для которых функция вернула `True`.

```
data = [x for x in range(10)]
res = list(filter(lambda x: x % 2 == 0, data))
print(res) # [0, 2, 4, 6, 8]
```

Как в данном случае работает функция **filter()**? Все данные, которые находятся внутри проходят через функцию, которая указана следующим образом:

```
lambda x: x % 2 == 0
```

То есть делает проверку на те числа, которые при делении на 2 дают в остатке 0. Тем самым мы ищем только четные числа. Действительно преобразовав наши итоговые данные в список, с помощью функции **list()**, мы с Вами можем наблюдать такой красивый результат 😊

Как можно сделать этот код лучше, используя **filter()**?



`filter()` позволит избавиться от функции `where`, которую мы писали ранее

```
data = '1 2 3 5 8 15 23 38'.split()
res = map(int, data)
res = filter(lambda x: x % 2 == 0, res)
res = list(map(lambda x: (x, x ** 2), res))
print(res)
```

Функция zip



Функция `zip()` применяется к набору итерируемых объектов и возвращает итератор с кортежами из элементов входных данных

```
zip ([1, 2, 3], [ 'о', 'д', 'т'], [ 'f', 's', 't'])  
      ↓  
[(1, 'о', 'f'), (2, 'д', 's'), (3, 'т', 't')]
```

На выходе получаем набор данных, состоящий из элементов соответствующих исходному набору.

Пример:

```
users = ['user1', 'user2', 'user3', 'user4', 'user5']  
ids = [4, 5, 9, 14, 7]  
data = list(zip(users, ids))  
print(data) # [('user1', 4), ('user2', 5), ('user3', 9), ('user4', 14),  
('user5', 7)]
```

Функция `zip ()` пробегает по минимальному входящему набору:

```
users = ['user1', 'user2', 'user3', 'user4', 'user5']  
ids = [4, 5, 9, 14, 7]  
salary = [111, 222, 333]  
data = list(zip(users, ids, salary))  
print(data) # [('user1', 4, 111), ('user2', 5, 222), ('user3', 333)]
```

Функция enumerate



Функция `enumerate()` применяется к итерируемому объекту и возвращает новый итератор с кортежами из индекса и элементов входных данных.

```
enumerate(['Казань', 'Смоленск', 'Рыбки', 'Чикаго'])  
↓  
[(0, 'Казань'), (1, 'Смоленск'), (2, 'Рыбки'), (3, 'Чикаго')]
```

Функция `enumerate()` позволяет пронумеровать набор данных.

```
users = ['user1', 'user2', 'user3']  
  
data = list(enumerate(users))  
  
print(data) # [(0, 'user1'), (1, 'user2'), (2, 'user3')]
```

Файлы

Файлы в текстовом формате используются для:

- Хранения данных
- Передачи данных в клиент-серверных проектах
- Хранения конфигов
- Логирования действий

Что нужно для работы с файлами:

1. Завести переменную, которая будет связана с этим текстовым файлом.
2. Указать путь к файлу.
3. Указать, в каком режиме мы будем работать с файлом.

Варианты режима (мод):

1. **a** – открытие для добавления данных.
 - Позволяет дописывать что-то в имеющийся файл.
 - Если вы попробуете дописать что-то в несуществующий файл, то файл будет создан и в него начнётся запись.
2. **r** – открытие для чтения данных.
 - Позволяет читать данные из файла.
 - Если вы попробуете считать данные из файла, которого не существует, программа выдаст ошибку.
3. **w** – открытие для записи данных.
 - Позволяет записывать данные и создавать файл, если его не существует.

Миксованные режимы:

4. **w+**
 - Позволяет открывать файл для записи и читать из него.
 - Если файла не существует, он будет создан.
5. **r+**
 - Позволяет открывать файл для чтения и дописывать в него.
 - Если файла не существует, программа выдаст ошибку.

Примеры использования различных режимов в коде:

1. Режим a

```
colors = ['red', 'green', 'blue']

data = open('file.txt', 'a') # здесь указываем режим, в котором будем работать
data.writelines(colors) # разделителей не будет

data.close()
```

- `data.close()` — используется для закрытия файла, чтобы разорвать подключение файловой переменной с файлом на диске.
- `exit()` — позволяет не выполнять код, прописанный после этой команды в скрипте.
- В итоге создаётся текстовый файл с текстом внутри: `redbluedreen`.
- При повторном выполнении скрипта `redbluedreenredbluedreen` — добавление в существующий файл, а не перезапись файлов.

Ещё один способ записи данных в файл:

```
with open('file.txt', 'w') as data:

    data.write('line 1\n')

    data.write('line 2\n')
```

2. Режим r

- Чтение данных из файла:

```
path = 'file.txt'

data = open(path, 'r')

for line in data:

    print(line)

data.close()
```

3. Режим w

```
colors = ['red', 'green', 'blue']

data = open('file.txt', 'w')

data.writelines(colors) # разделителей не будет

data.close()
```

- В итоге создаётся текстовый файл с текстом внутри: **‘redbluedreen’**.
- В случае перезаписи новые данные записываются, а старые удаляются.

Модуль os

Модуль os предоставляет множество функций для работы с операционной системой, причем их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми.

Для того, чтобы начать работать с данным модулем необходимо его импортировать в свою программу:

```
import os
```

Познакомимся с базовыми функциями данного модуля:

- **os.chdir(path)** - смена текущей директории.

```
import os

os.chdir('C:/Users/79190/PycharmProjects/GB')
```

- **os.getcwd()** - текущая рабочая директория

```
import os

print(os.getcwd()) # 'C:\Users\79190\PycharmProjects\webproject'
```


- **os.path** - является вложенным модулем в модуль os и реализует некоторые полезные функции для работы с путями, такие как:
 - **os.path.basename(path)** - базовое имя пути

```
import os

print(os.path.basename('C:/Users/79190/PycharmProjects/webproject/main.py')) #
'main.py'
```

- **os.path.abspath(path)** - возвращает нормализованный абсолютный путь.

```
import os

print(os.path.abspath('main.py'))

# 'C:/Users/79190/PycharmProjects/webproject/main.py'
```

Это лишь малая часть возможностей модуля **os**.

Модуль shutil

Модуль **shutil** содержит набор функций высокого уровня для обработки файлов, групп файлов, и папок. В частности, доступные здесь функции позволяют копировать, перемещать и удалять файлы и папки. Часто используется вместе с модулем os.

Для того, чтобы начать работать с данным модулем необходимо его импортировать в свою программу:

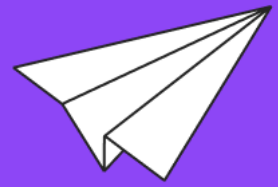
```
import shutil
```

Познакомимся с базовыми функциями данного модуля:

- **shutil.copyfile(src, dst)** - копирует содержимое (но не метаданные) файла src в файл dst.
- **shutil.copy(src, dst)** - копирует содержимое файла src в файл **или папку** dst.
- **shutil.rmtree(path)** - Удаляет текущую директорию и все поддиректории; path должен указывать на директорию, а не на символическую ссылку.

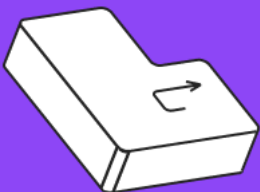
Итоги:

1. Изучали функции высшего порядка, такие как:
 - **map**
 - **filter**
 - **zip**
 - **lambda**
 - **enumerate**
2. Научили работать с файлами
3. Изучили библиотеки для работы с операционной системой и файлами



Лекция 5. Google Colab(Jupyter). Знакомство с аналитикой.

Курс



Оглавление

Введение	2
Термины, используемые в лекции	3
Чтение и предварительный просмотр данных	3
Выбор данных	5
Простая статистика	7
Изображаем статистические отношения	8
Линейные графики	10
Гистограмма	11
Выводы	13

Введение

Знакомство с аналитикой. Мы будем пользоваться таким инструментом как Google colab

На лекции мы познакомимся с инструментом для работы с табличными данными(**pandas**) и способами визуализации данных с помощью библиотек **matplotlib** и **seaborn**. Прежде, чем приступить непосредственно к машинному обучению, важно произвести **EDA**(Exploratory Data Analysis) - Разведочный анализ данных.

Он состоит в анализе основных свойств данных, нахождения в них общих закономерностей, распределений и аномалий, построение начальных моделей, зачастую с использованием инструментов визуализации.

Понятие введено математиком **Джоном Тьюки**, который сформулировал цели такого анализа следующим образом:

1. Максимальное «проникновение» в данные
2. Выявление основных структур
3. Выбор наиболее важных переменных
4. Обнаружение отклонений и аномалий
5. Проверка основных гипотез

Термины, используемые в лекции

Перцентиль - это показатель, используемый в статистике, показывающий значение, ниже которого падает определенный процент наблюдений в группе наблюдений

Scatterplot (Точечный график) - Математическая диаграмма, изображающая значения двух переменных в виде точек на декартовой плоскости.

Медиана набора чисел — число, которое находится в середине этого набора, если его упорядочить по возрастанию, то есть такое число, что половина из элементов набора не меньше него, а другая половина не больше.

Базовые функции для работы с данными

Библиотека **pandas** может читать многие форматы, включая: **.csv**, **.xlsx**, **.xls**, **.txt**, **sql** и многие другие. Полный список по [ссылке](#)

Чтобы подключить библиотеку к Вашей программе необходимо написать следующее:

```
import pandas as pd
```

Напоминание: as(alias) - псевдоним. Мы можем сократить название все библиотеки до 2-х букв.

Прочтем файл **.csv**(он находится в Google Colab в папке **sample_data**) с помощью библиотеки **pandas**

```
df = pd.read_csv('sample_data/california_housing_train.csv')
```

Для того чтобы прочитать первые n строк таблицы, необходимо воспользоваться следующей функцией:

```
DataFrame.head(n=5)
```

Где DataFrame - это таблица с данными, которая предварительно была открыта. Мы ее открыли и записали в переменную **df**. Необязательно указывать n=5, вместо 5 мы можем указать любое число(число не должно превосходить количество строк в таблице). Если Вы ничего не укажете в круглых скобках, то ошибка не вылезет, по умолчанию будут выведены первые 5 строк таблицы.

Пример:

```
df.head()
```

Результат:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	-114.31	34.19	15.0	5612.0	1283.0	1015.0	472.0	1.4936	66900.0
1	-114.47	34.40	19.0	7650.0	1901.0	1129.0	463.0	1.8200	80100.0
2	-114.56	33.69	17.0	720.0	174.0	333.0	117.0	1.6509	85700.0
3	-114.57	33.64	14.0	1501.0	337.0	515.0	226.0	3.1917	73400.0
4	-114.57	33.57	20.0	1454.0	326.0	624.0	262.0	1.9250	65500.0

Как мы знаем, в нашем мире почти все симметрично, есть отрицательные числа, а есть положительные и тд. Значит, если есть функция, которая показывает первые 5 строк таблицы, то и есть функция, которая показывает последние 5 строк таблицы. Да, действительно, это так. Давайте с ней познакомимся

Пример:

```
df.tail()
```

Результат:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
16995	-124.26	40.58	52.0	2217.0	394.0	907.0	369.0	2.3571	111400.0
16996	-124.27	40.69	36.0	2349.0	528.0	1194.0	465.0	2.5179	79000.0
16997	-124.30	41.84	17.0	2677.0	531.0	1244.0	456.0	3.0313	103600.0
16998	-124.30	41.80	19.0	2672.0	552.0	1298.0	478.0	1.9797	85800.0
16999	-124.35	40.54	52.0	1820.0	300.0	806.0	270.0	3.0147	94600.0

Данная функция работает аналогично с head(). Необязательно выводить последние 5 строчек, можно указать сколько угодно.

Иногда заранее неизвестно сколько строк и столбцов находится внутри таблицы, чтобы это сделать необходимо воспользоваться специальной функцией.

Пример:

```
df.shape
```

Результат:

```
(17000, 9)
```

Функция shape возвращает размеры таблицы: кортеж из 2 значений, 1 - количество строк, 2 - количество столбцов.

Согласитесь, что все не раз делали заказ на каком-нибудь маркетплейсе. И когда мы заполняли поле **“Email”**, то могли его пропустить, потому что указано, что оно необязательное и не хотели видеть лишнего спама. Вы когда-нибудь задумывались, как в этом случае эти данные будут выглядеть внутри базы данных(таблице)? Пропуск? Пустая ячейка? Нет. Когда нужно указать, что в данной ячейке таблицы ничего нет указывается значение **null**.

Чтобы обнаружить пустые значения в таблице данных необходимо воспользоваться функцией `.isnull()`.

Пример:

```
df.isnull()
```

Результат:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False
...
16995	False	False	False	False	False	False	False	False	False
16996	False	False	False	False	False	False	False	False	False
16997	False	False	False	False	False	False	False	False	False
16998	False	False	False	False	False	False	False	False	False
16999	False	False	False	False	False	False	False	False	False

Функция привела нашу таблицу к следующему виду **True-False**, где **True** - это пустая ячейка, **False** - это заполненная ячейка. Но это неудобно, то есть нам надо просматривать $17\ 000 * 9 = 153\ 000$ ячеек. Вау... Это займет слишком много времени. Однако, мы можем воспользоваться еще одной функцией `.sum()`. Данная функция выведет количество null-значений в каждой ячейке по столбцам.

Пример:

```
df.isnull().sum()
```

Результат:

```
longitude      0
latitude       0
housing_median_age  0
total_rooms    0
total_bedrooms 0
population     0
households     0
median_income  0
median_house_value 0
```


Можно сделать следующий вывод: пустые значения в нашей таблицы отсутствуют.

Еще при работе с C#, мы поняли, что у каждой переменной есть свой тип данных. Также и здесь, у каждого столбца есть свой тип данных, чтобы это узнать, нужно применить функцию **.dtypes**.

Пример:

```
df.dtypes
```

Результат:

```
longitude          float64
latitude           float64
housing_median_age  float64
total_rooms         float64
total_bedrooms      float64
population          float64
households          float64
median_income       float64
median_house_value  float64
```

Делаем вывод: у всей таблицы данных один тип **float(дробное число)**

Чтобы узнать название всех столбцов в таблице, воспользуйтесь функцией **.columns**.

Пример:

```
df.columns
```

Результат:

```
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
      'total_bedrooms', 'population', 'households', 'median_income',
      'median_house_value'],
      dtype='object')
```

Выборка данных



Медиана набора чисел — число, которое находится в середине этого набора, если его упорядочить по возрастанию, то есть такое число, что половина из элементов набора не меньше него, а другая половина не больше.

Если Вы хотите вывести 1 столбец на экран, то можно указать следующее выражение, которое позволит это сделать.

Пример:

```
df['latitude']
```

Результат:

```
0      34.19
1      34.40
2      33.69
3      33.64
4      33.57
...
16995   40.58
16996   40.69
16997   41.84
16998   41.80
16999   40.54
Name: latitude, Length: 17000, dtype: float64
```

Что мы будем делать, если нам потребуется вывести на экран сразу несколько столбцов? Не очень удобно будет это прописывать вот таким образом:

```
print(df['latitude'])
print(df['population'])
```

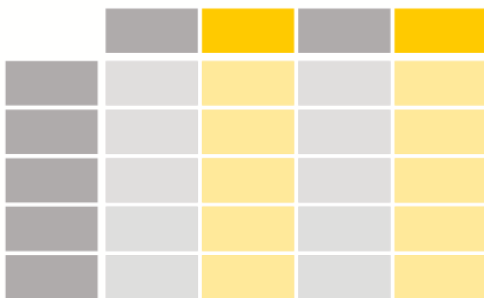
Конечно есть решение данного вопроса

Пример:

```
df[['latitude', 'population']]
```

Результат:

	latitude	population
0	34.19	1015.0
1	34.40	1129.0
2	33.69	333.0
3	33.64	515.0
4	33.57	624.0
...
16995	40.58	907.0
16996	40.69	1194.0
16997	41.84	1244.0
16998	41.80	1298.0
16999	40.54	806.0



Задание: Необходимо вывести столбец **total_rooms**, у которого медианный возраст здания(**housing_median_age**) меньше **20**.

Для того чтобы решить это задание, давайте познакомимся с синтаксисом выборки данных. На самом деле, это ничем не отличается от операторов ветвления.

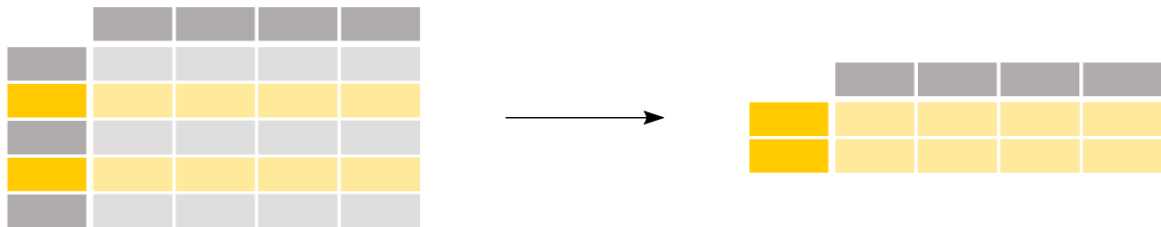
Решение:

```
df[df['housing_median_age'] < 20]
```

Результат:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	-114.31	34.19	15.0	5612.0	1283.0	1015.0	472.0	1.4936	66900.0
1	-114.47	34.40	19.0	7650.0	1901.0	1129.0	463.0	1.8200	80100.0
2	-114.56	33.69	17.0	720.0	174.0	333.0	117.0	1.6509	85700.0
3	-114.57	33.64	14.0	1501.0	337.0	515.0	226.0	3.1917	73400.0
10	-114.60	33.62	16.0	3741.0	801.0	2434.0	824.0	2.6797	86500.0
...
16983	-124.19	41.78	15.0	3140.0	714.0	1645.0	640.0	1.6654	74600.0
16987	-124.21	41.77	17.0	3461.0	722.0	1947.0	647.0	2.5795	68400.0
16991	-124.23	41.75	11.0	3159.0	616.0	1343.0	479.0	2.4805	73200.0
16997	-124.30	41.84	17.0	2677.0	531.0	1244.0	456.0	3.0313	103600.0
16998	-124.30	41.80	19.0	2672.0	552.0	1298.0	478.0	1.9797	85800.0

4826 rows × 9 columns



Если Вам нужно поставить другое условие, то аналогично.

Мы помним с С#, что иногда приходится проверять несколько условий сразу. Чтобы проверить несколько условий внутри Google Colab, указывается так:

```
df[(df['housing_median_age'] > 20) & (df['total_rooms'] > 2000)]
```

& - выполнение одновременно **всех** условий.

| - выполнение **хотя бы одного** из условия.

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
6	-114.58	33.61	25.0	2907.0	680.0	1841.0	633.0	2.6768	82400.0
8	-114.59	33.61	34.0	4789.0	1175.0	3134.0	1056.0	2.1782	58400.0
13	-114.61	34.83	31.0	2478.0	464.0	1346.0	479.0	3.2120	70400.0
42	-115.49	32.67	25.0	2322.0	573.0	2185.0	602.0	1.3750	70100.0
45	-115.50	32.67	35.0	2159.0	492.0	1694.0	475.0	2.1776	75500.0
...
16986	-124.19	40.73	21.0	5694.0	1056.0	2907.0	972.0	3.5363	90100.0
16990	-124.22	41.73	28.0	3003.0	699.0	1530.0	653.0	1.7038	78300.0
16993	-124.23	40.54	52.0	2694.0	453.0	1152.0	435.0	3.0806	106700.0
16995	-124.26	40.58	52.0	2217.0	394.0	907.0	369.0	2.3571	111400.0
16996	-124.27	40.69	36.0	2349.0	528.0	1194.0	465.0	2.5179	79000.0

5624 rows × 9 columns

Первую часть задания мы успешно выполняли! Только загвоздка... Нам не нужна вся таблица, а лишь один столбец, как это сделать?

Решение:

```
df[df['housing_median_age'] < 20, 'total_rooms']  
  
# или (если необходимо вывести 2 и более столбцов)  
df[df['housing_median_age'] < 20, ['total_bedrooms', 'total_rooms']]
```

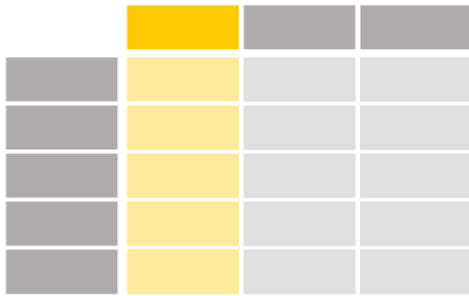
Результат:

```
17      44.0  
19      97.0  
113     96.0  
116    208.0  
120    186.0  
...  
16643   255.0  
16733   411.0  
16743    89.0  
16801    98.0  
16851   133.0  
Name: total_rooms, Length: 178, dtype: float64
```



Простая статистика

Pandas позволяет получить основные простые данные для описательной статистики. Такие как минимальное значение в столбце, максимальное значение, сумма всех значений, среднее значение



Максимальное значение:

```
print(df['population'].max())
```

Результат:

35682.0

Минимальное значение:

```
print(df['population'].min())
```

Результат:

3.0

Среднее значение:

```
print(df['population'].mean())
```

Результат:

1429.5739411764705

Сумма:

```
print(df['population'].sum())
```

Результат:

24302757.0

Эту же статистику можно рассчитывать сразу для нескольких столбцов



Медианное значение:

```
df[['population', 'total_rooms']].median()
```

Результат:

```
population    1155.0
total_rooms   2106.0
dtype: float64
```



Перцентиль - это показатель, используемый в статистике, показывающий значение, ниже которого падает определенный процент наблюдений в группе наблюдений

Получить общую картину можно простой командой **describe**

```
df.describe()
```

Результат:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	3000.000000	3000.000000	3000.000000	3000.000000	3000.000000	3000.000000	3000.000000	3000.000000	3000.000000
mean	-119.589200	35.63539	28.845333	2599.578667	529.950667	1402.798667	489.91200	3.807272	205846.27500
std	1.994936	2.12967	12.555396	2155.593332	415.654368	1030.543012	365.42271	1.854512	113119.68747
min	-124.180000	32.56000	1.000000	6.000000	2.000000	5.000000	2.00000	0.499900	22500.00000
25%	-121.810000	33.93000	18.000000	1401.000000	291.000000	780.000000	273.00000	2.544000	121200.00000
50%	-118.485000	34.27000	29.000000	2106.000000	437.000000	1155.000000	409.50000	3.487150	177650.00000
75%	-118.020000	37.69000	37.000000	3129.000000	636.000000	1742.750000	597.25000	4.656475	263975.00000
max	-114.490000	41.92000	52.000000	30450.000000	5419.000000	11935.000000	4930.00000	15.000100	500001.00000

count - Общее кол-во не пустых строк

mean - среднее значение в столбце

std - стандартное отклонение от среднего значения

min - минимальное значение

max - максимальное значение

Числа **25%, 50%, 75%** - перцентили

Изображаем статистические отношения

Scatterplot (Точечный график)

Математическая диаграмма, изображающая значения двух переменных в виде точек на декартовой плоскости. Библиотека **seaborn** без труда принимает **pandas DataFrame**(таблицу). Чтобы изобразить отношения между двумя столбцами достаточно указать, какой столбец отобразить по оси **x**, а какой по оси **y**.

Для того чтобы начать работу с библиотекой **seaborn**, ее необходимо импортировать к себе в программу:

```
import seaborn as sns
```

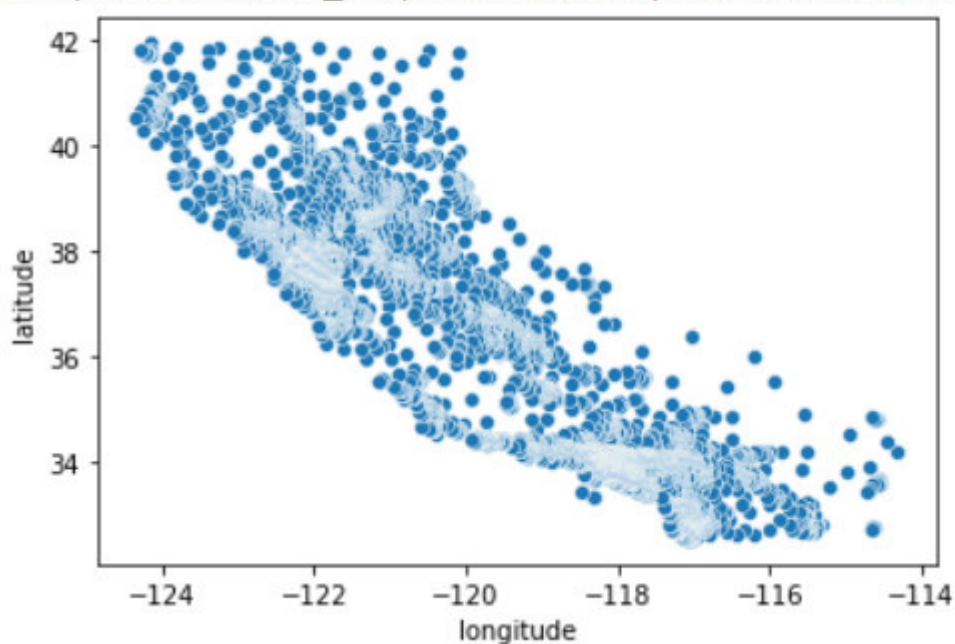
Вернемся к нашей таблице. Можно заметить, что дома расположены в определенной "полосе" долготы и широты.

Изображение точек долготы по отношению к широте:

```
sns.scatterplot(data=df, x="longitude", y="latitude")
```

Результат:

<matplotlib.axes._subplots.AxesSubplot at 0x7fa155b53050>

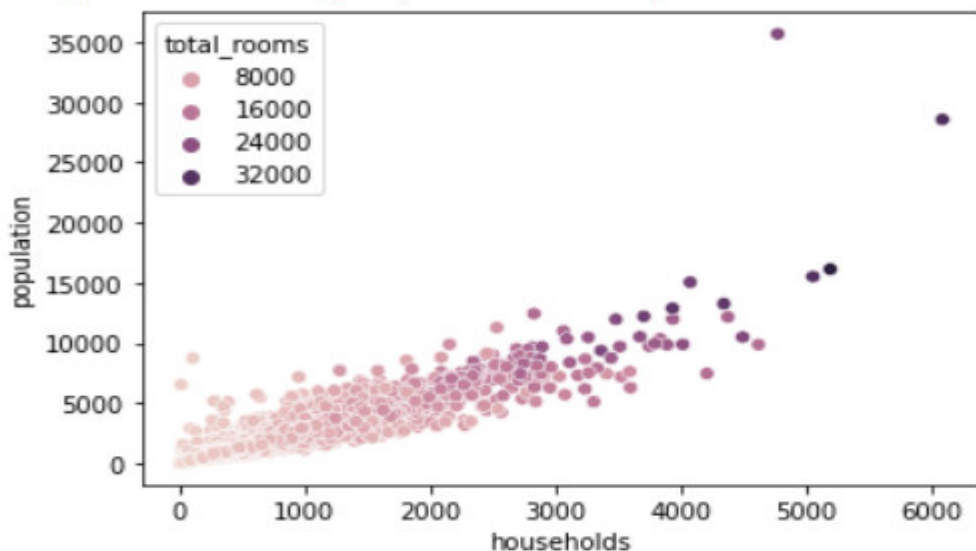


Помимо двумерных отношений, мы можем добавить "дополнительное измерение" с помощью цвета. В данном случае опять же достаточно очевидное отношение, чем выше кол-во семей, тем выше кол-во людей и соответственно комнат.

```
sns.scatterplot(data=df, x="households", y="population", hue="total_rooms")
```

Результат:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa14df5bcd0>
```

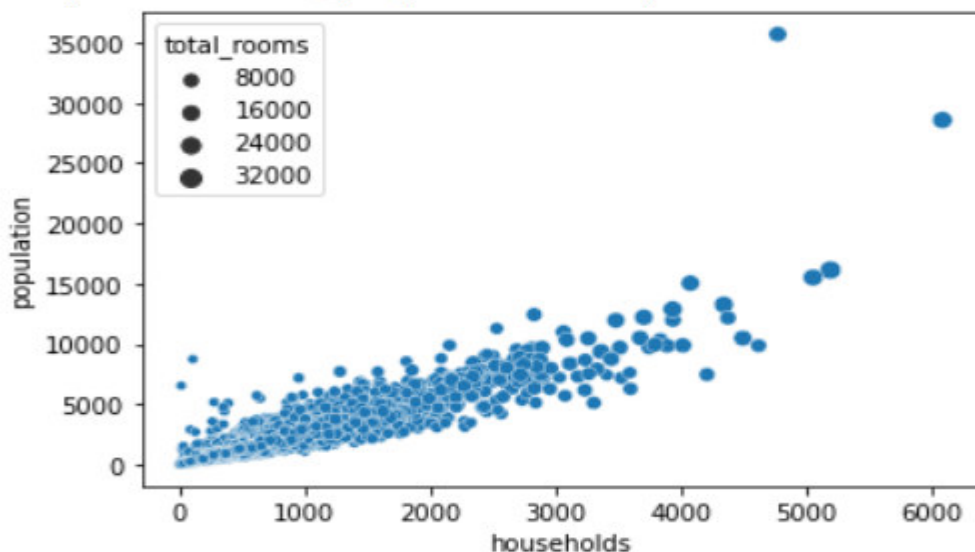


Помимо обозначения дополнительного измерения цветом мы можем использовать **size**.

```
sns.scatterplot(data=df, x="households", y="population", hue="total_rooms")
```

Результат:

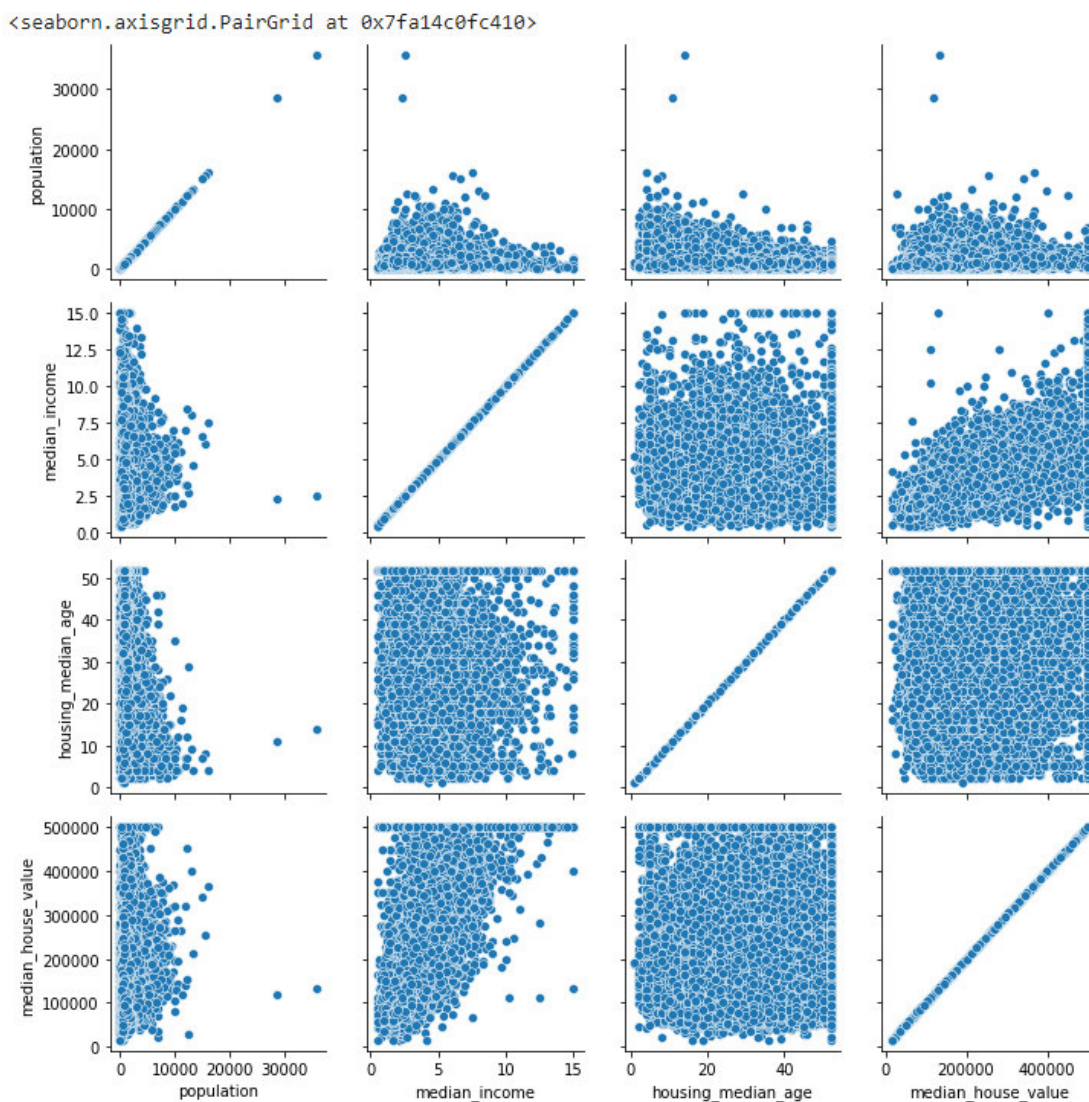
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa14c1a3490>
```



Мы можем визуализировать сразу несколько отношений, используя класс **PairGrid** внутри **seaborn**. **PairGrid** принимает как аргумент pandas **DataFrame** и визуализирует все возможные отношения между ними, в соответствии с выбранным типом графика.

```
cols = ['population', 'median_income', 'housing_median_age', 'median_house_value']
g = sns.PairGrid(df[cols])
g.map(sns.scatterplot)
```

Результат:



Как Вы думаете, чем вызвана линейная зависимость по диагонали?

Линейные графики

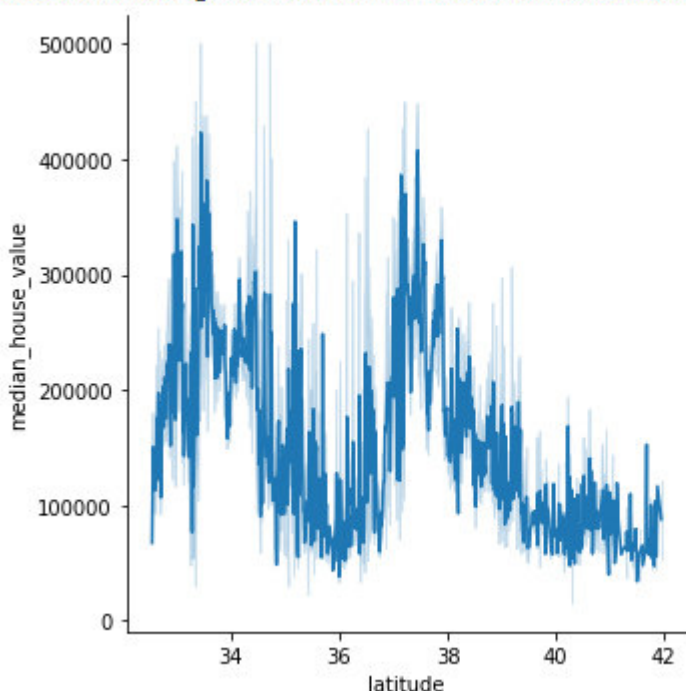
Хорошо подойдут, если есть временная или какая-либо иная последовательность и значения, которые могут меняться в зависимости от нее. Для генерации линейных графиков в **seaborn** используется **relplot** функцию. Она также принимает **DataFrame**, **x**, **y** - столбцы.

Для визуализации выбирается тип **line**:

```
sns.relplot(x="latitude", y="median_house_value", kind="line", data=df)
```

Результат:

```
<seaborn.axisgrid.FacetGrid at 0x7fa14c03eb90>
```



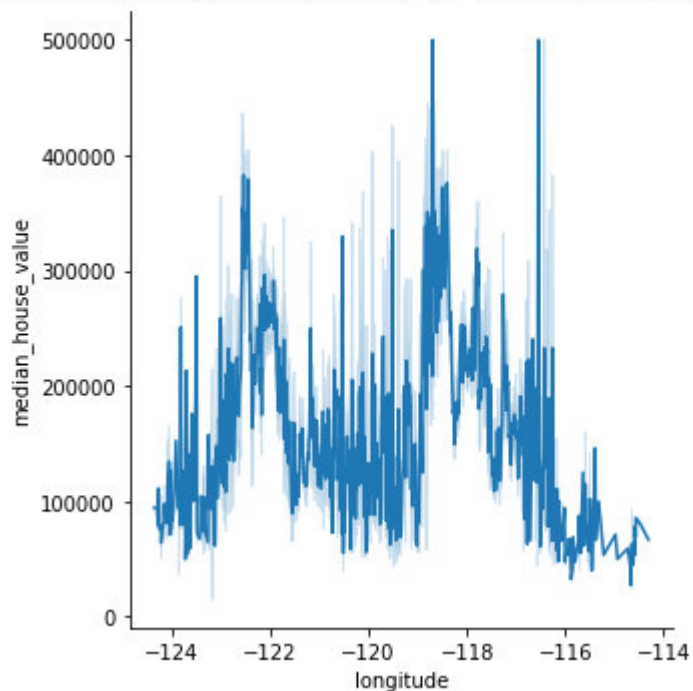
Можно видеть, что в определенных местах долготы цена за дома резко подскакивает.

Попробуем визуализировать longitude по отношению к median_house_value и поймем в чем же дело, почему цена так резко подскакивает.

```
sns.relplot(x = 'longitude', y = 'median_house_value', kind = 'line', data = df)
```

Результат:

```
<seaborn.axisgrid.FacetGrid at 0x7fa14c180b10>
```



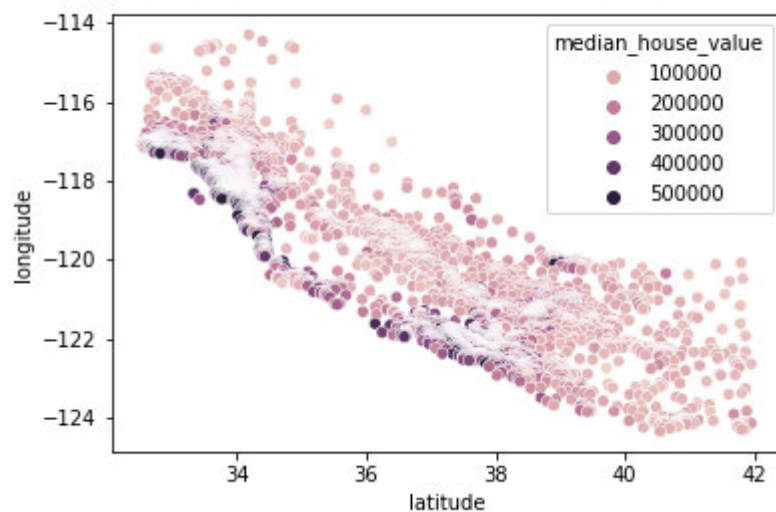
Можно видеть, что в определенных местах широты цена за дома также очень высока.

Используя точечный график можно визуализировать эти отношения с большей четкостью. Скорее всего резкий рост цен связан с близостью к ценному объекту, повышающему качество жизни, скорее всего побережью океана или реки.

```
sns.scatterplot(data=df, x="latitude", y="longitude", hue="median_house_value")
```

Результат:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa149121fd0>
```



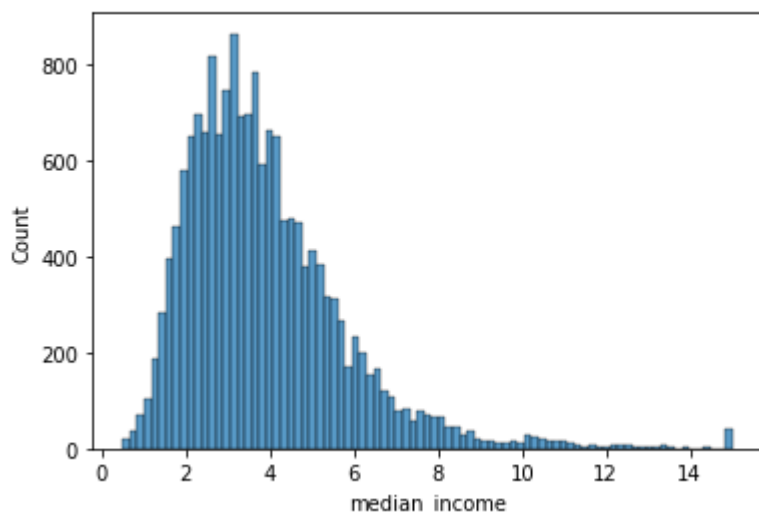
Гистограмма

Способ представления табличных данных в графическом виде — в виде столбчатой диаграммы. По оси **x** обычно указывают значение, а по оси **y** - встречаемость(кол-во таких значений в выборке)

```
sns.histplot(data=df, x="median_income")
```

Результат:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa14915eb10>
```



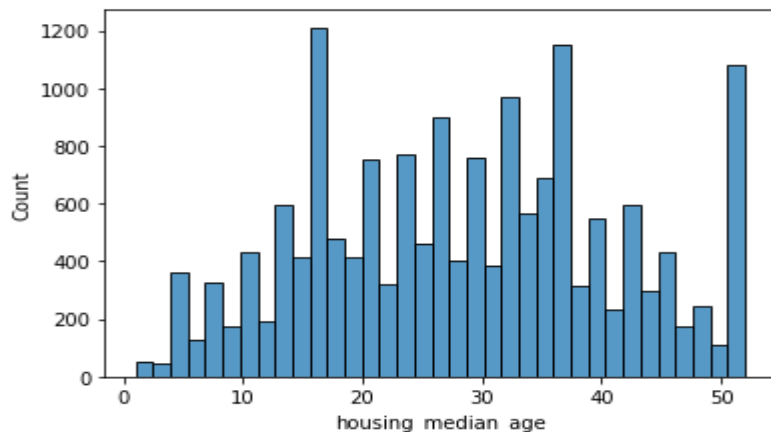
Можно видеть что у большинства семей доход находится между значениями 2 и 6. И только очень небольшое количество людей обладают доходом > 10.

Изобразим гистограмму по **housing_median_age**.

```
sns.histplot(data = df, x = 'housing_median_age')
```

Результат:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa148fc6c50>
```



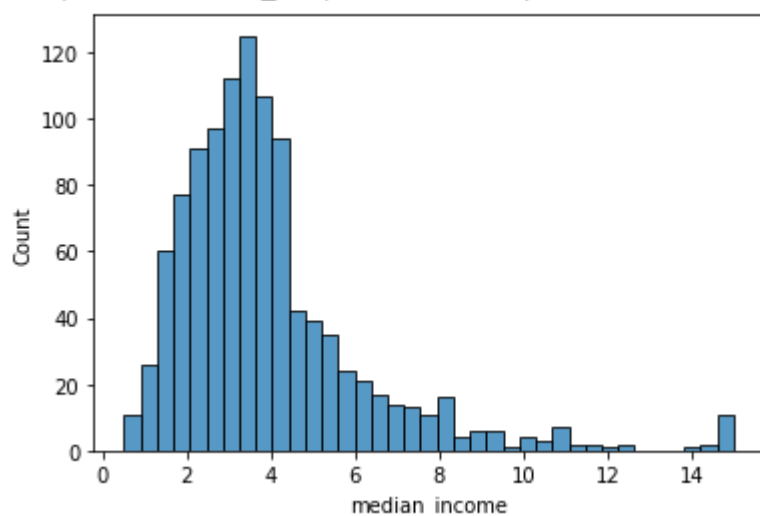
Распределение по возрасту более равномерное. Большую часть жителей составляют люди в возрасте от 20 до 40 лет. Но и молодежи не мало. Также очень много пожилых людей > 50 лет медианный возраст.

Давайте посмотрим медианный доход у пожилых жителей.

```
sns.histplot(data=df[df['housing_median_age']>50], x="median_income")
```

Результат:

<matplotlib.axes._subplots.AxesSubplot at 0x7fa1490f3350>



Большого отличия от популяции в целом не наблюдается. Скорее всего это местные жители.

Давайте разобьем возрастные группы на 3 категории те кто моложе 20 лет, от 20 до 50 и от 50, чтобы посмотреть влияет ли это на доход.

```
df.loc[df['housing_median_age'] <= 20, 'age_group'] = 'Молодые'
df.loc[(df['housing_median_age'] > 20) & (df['housing_median_age'] <= 50),
'age_group'] = 'Ср. возраст'
df.loc[df['housing_median_age'] > 50, 'age_group'] = 'Пожилые'
```

Что в этом случае происходит внутри таблицы? Добавился новый столбец **age_group**, в котором будет указана соответствующая категория.

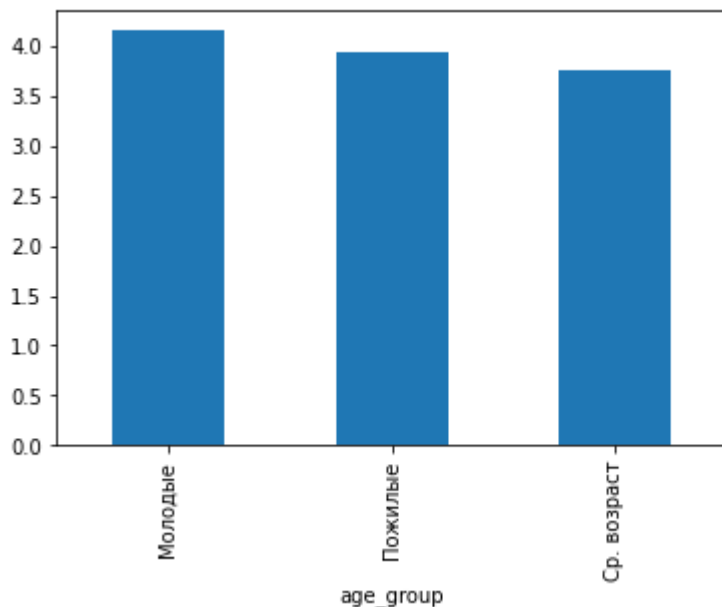
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	age_group
0	-114.31	34.19	15.0	5612.0	1283.0	1015.0	472.0	1.4936	66900.0	Молодые

Применим **group_by**, чтобы получить среднее значение.

```
df.groupby('age_group')['median_income'].mean().plot(kind='bar')
```

Результат:

<matplotlib.axes._subplots.AxesSubplot at 0x7fa14833aa90>



Молодые оказываются самой богатой группой населения. Но отличие в доходе не значительное.

Seaborn так же позволяет нам смотреть распределение по многим параметрам. Давайте поделим группы по доходам на 2. Те у кого медианный доход выше 6 и те у кого меньше. Изобразим дополнительное измерение с помощью оттенка в виде возрастных групп и групп по доходам.

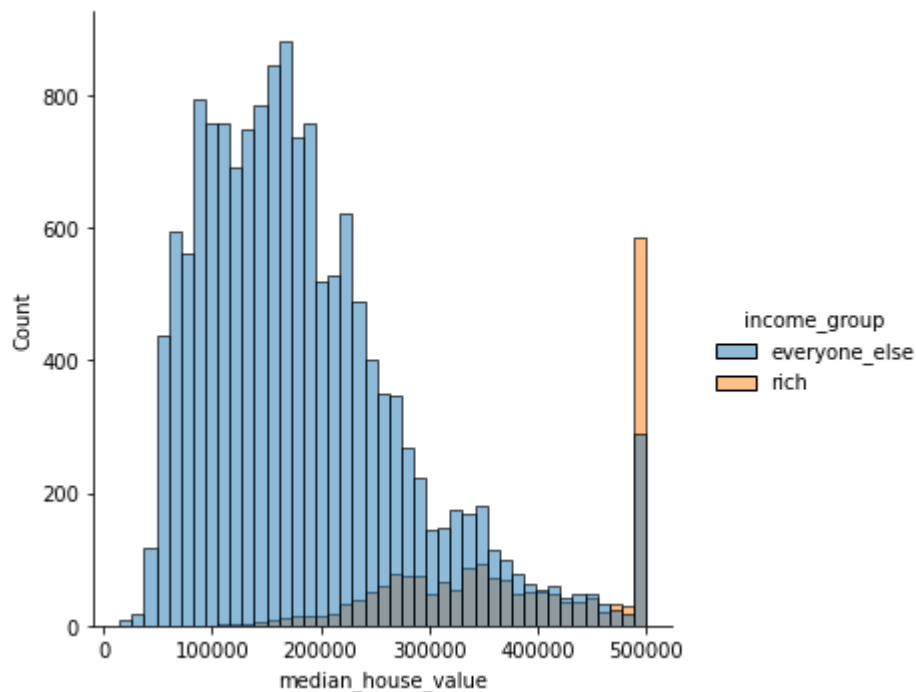
```
df.loc[df['median_income'] > 6, 'income_group'] = 'rich'  
df.loc[df['median_income'] < 6, 'income_group'] = 'everyone_else'
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	age_group	income_group
0	-114.31	34.19	15.0	5612.0	1283.0	1015.0	472.0	1.4936	66900.0	Молодые	everyone_else

```
sns.displot(df, x="median_house_value", hue="income_group")
```

Результат:

<seaborn.axisgrid.FacetGrid at 0x7fa148267f50>



Итоги:

Анализ данных должен предоставлять информацию и инсайт, которые не видны невооруженным взглядом. В этом и есть красота аналитики. В данном случае можно сделать следующий вывод. Стоимость домов напрямую зависит от их расположения, в определенной полосе(скорее всего побережье) цена на дома высокая. Чем выше доход, тем больше шанс, что человек проживает в богатом районе. Распределение по возрастам примерно одинаковое во всех группах доходов. Ну и очевидно чем больше людей, тем больше семей, и соответственно комнат и спален.