😄 Python 2

Python 2

ctrl + LMB на функцию - просмотр определения функции.

ctrl + d - копия текущей строчки

инфо

про числа

<https://pythonworld.ru/tipy-dannyx-v-python/chisla-int-float-complex.html>

как работают переменные и массивы в пайтон



# \*Полезные функции\*

получение данных с клавиатуры

str = input()

Получить список символов из строки без пробелов

Str = “hello world”

Chars\_list = [ x for x in str if x != “ “]

посмотреть подклассы в пайтон

b = BaseException.\_\_subclasses\_\_()

print(b)

b = object.\_\_subclasses\_\_()

print(b)

# Операторы сравнения

Операторы сравнения в Python:

* == (Равно): Проверяет, равны ли два значения. Возвращает True, если равны, иначе False.
* != (Не равно): Проверяет, не равны ли два значения. Возвращает True, если не равны, иначе False.
* > (Больше): Проверяет, больше ли первое значение второго.
* < (Меньше): Проверяет, меньше ли первое значение второго.
* >= (Больше или равно): Проверяет, больше ли первое значение второго или равно ему.
* <= (Меньше или равно): Проверяет, меньше ли первое значение второго или равно ему.

Логические операторы в Python:

* and (И): Возвращает True, если оба операнда истинны, иначе False.
* or (ИЛИ): Возвращает True, если хотя бы один операнд истинен, иначе False.
* not (НЕ): Инвертирует значение операнда. True становится False, а False становится True.

Эквиваленты операторов & и || из C++ в Python:

* & (И): В Python and выполняет ту же функцию, что и & в C++.
* || (ИЛИ): В Python or выполняет ту же функцию, что и || в C++.

# Массивы

Определения:

Изменяемость - свойство объекта, благодаря которому можно вносить изменения в объект (добавлять новые, изменять значения старых) НЕ ИЗМЕНЯЯ текущую ссылку и не создавая новых объектов.

\*для проверки можно создать две переменные ссылающиеся на один и тот же объект, изменить объект и проверить хранят ли переменные ту же ссылку что и раньше:

b = [...]

a = b

a[1] = 5

id(a) == id(b) // или же как a is b

\*

Упорядоченность - существует какой либо индекс или ключ для каждого элемента. То есть это свойство, благодаря которому при изменении порядка объектов в типе данных изменяется ссылка на объект (создается новый объект или копия).

Неупорядоченность - при изменении порядка объектов тип данных остается такой же и его ссылка не изменяется.

Поэлементное присваивание - когда можно по индексу присвоить новое значение объекту.

## Строка

инфо:

<https://pythonworld.ru/tipy-dannyx-v-python/stroki-funkcii-i-metody-strok.html>  
<https://www.w3schools.com/python/python_ref_string.asp>

<https://metanit.com/python/tutorial/5.2.php>

(неизменяемый)

str = “qwerty”

Работа со строками:

str.upper() - перевод строки в верхний регистр.

str.isdigit() - проверяет является ли строка числом.

## Списки

инфо

<https://www.w3schools.com/python/python_ref_list.asp>

<https://pythonworld.ru/tipy-dannyx-v-python/spiski-list-funkcii-i-metody-spiskov.html>

(изменяемый тип, упорядоченный по индексу, ).

list = [“a”,”b”, 3, 4.5]

Изменение списка:

### методы list:

list.append(\*объект\*) - добавление нового объекта в конец

Работа со списками, чтением и выводом списков:

преобразование строки в список

str.split(“символ разделения объектов”)

СПЛИТ СОЗДАЕТ STR объекты, а не числа, их нужно переводить в int

преобразование списка в строку, где объекты разделены символом

“символ”.join(list)

max(list), min(list) - функции возвращающие максимальное и минимальное значение списка, список должен быть однородный.

a= [‘1’,’2’,’3’]

b = a.pop(0) # b = ‘1’ a = [’2’,’3’]

### Копия

копирование списка

<https://www.youtube.com/watch?v=-ngLnCK4t10>

поверхностное (проблемы с вложенными списками - при изменении вложенного списка в списке, то каким то образом изменятся вложенные списки всех копий этого оригинального списка, даже если они по другой ссылке и в другой области памяти):

a = b[:]

a = list(b)

полное копирование (все ссылки меняются на новые и даже вложенные списки тоже в другой области памяти, ничего не зависит от изначального списка):

import copy

a = copy.deepcopy(b)

### Генераторы

Автоматическое заполнение числами типа int из клавиатурного ввода

L = input().strip().split()

L\_int = [int(item) for item in L] # выполняется действие \_ в цикле \_

Генерация списка целых чисел

L = list(range(10))

Генератор матрицы

mtrx = [[0 for i in range(3)] for i in range(3)]

0 for i in range(3) - создает список [0, 0, 0]

Генератор списка с условием

L = [i for i in range(10) if i%2 == 0] # выполняется действие \_ (создание [i]) в цикле \_ ,

при условии \_\_

Другие генераторы (словаря):

<https://www.youtube.com/watch?v=t9IpOCL-GQc&t=923s>

### Срез

срез работает со строками, списками, кортежами

Синтаксис:

list[start:stop:step]

* [необязательный] start (начало): Индекс первого элемента, который будет включен в срез. По умолчанию равен 0.
* [необязательный] stop (конец): Индекс первого элемента, который не будет включен в срез. По умолчанию равен длине последовательности.
* [необязательный] step (шаг): Определяет шаг, с которым будут выбираться элементы. По умолчанию равен 1.

Срез поддерживает отрицательные индексы (отрицательный шаг, отрицательная длина)

01234567891011

длинная\_строка

-10-9-8-7-6-5-4-3-2-1

Пример

my\_list = [1, 2, 3, 4, 5]

# Получение элементов с 1-го по 3-й (индекс 0, 1, 2)

sub\_list = my\_list[1:4] # [2, 3, 4]

# Получение элементов с 2-го по последний (индекс 1, 2, 3, 4)

sub\_list = my\_list[1:] # [2, 3, 4, 5]

# Получение элементов с начала до 3-го (индекс 0, 1, 2)

sub\_list = my\_list[:3] # [1, 2, 3]

Операции

Копия списка:

my\_list = [1, 2, 3, 4, 5]

copy\_list = my\_list[:] # [1, 2, 3, 4, 5]

Инвертированный порядок:

my\_list = [1, 2, 3, 4, 5]

reversed\_list = my\_list[::-1] # [5, 4, 3, 2, 1]

Выбор элементов с шагом:

my\_list = [1, 2, 3, 4, 5, 6]

sub\_list = my\_list[::2] # [1, 3, 5]

### Фильтр строчек

out = data[data['qwerty'] > 1]

вернёт мне data со всеми колонками, но без строчек неудовлетворяющим условию data['qwerty'] > 1]

В результате out будет содержать:

Все столбцы из data, но только строки, где значение в 'qwerty' больше 1.

## Словарь (dict)

(изменяемый(значения),

(ключем может быть ТОЛЬКО НЕИЗМЕНЯЕМЫЙ тип данных, значения изменяемые) - список, где вместо индекса - объект неизменного типа данных.

Изменение объектов словаря в цикле недопускается

создание

1) вручную: countries = { “russia” : “moscow”, “russia” : “kazan”}

2) с помощью метода fromkeys: dict.fromkeys([“a”, “b”], 100) ⇒ {“a” : 100, “b” : 100}

3) с помощью цикла:

словарь = {}

ключи = ["имя", "возраст", "город"]

значения = ["Иван", 30, "Москва"]

for i, ключ in enumerate(ключи):

словарь[ключ] = значения[i]

print(словарь)

Перевод из упорядоченного списка в словарь: dict(a)

### Методы dict

a.get():

dict = {‘a’ : 1, ‘b’ : 2}

value = dict.get(‘a’) # value = 1

### Полезно

можно проверить есть ли ключ в словаре:

if 'left' in t:

in\_order\_traversal(t['left'])

print(t['key'])

if 'right' in t:

in\_order\_traversal(t['right'])

## Множество (set)

(изменяемый, неупорядоченный, разные типы данных)

- тип данных, который хранит только неповторяющиеся элементы.

К объектам множества нельзя обращаться по индексу,

a = set(“usa”, “canada”, 1, 4, 5.6)

a = {“a”, “b”, 1, 2}

a = {“hello!”} (объекты: “h”, “e”, “l”, “l”, “o”, “!”)

работа с множествами

добавление - a.add(\*\*\*)

### Функции для множеств в Python:

Создание множеств:

* set(iterable): Создает новое множество из итерируемого объекта (например, списка, кортежа, строки).
* {}: Создает пустое множество (не пустой словарь!).

Основные операции:

* add(element): Добавляет элемент в множество.
* remove(element): Удаляет элемент из множества. Вызывает ошибку, если элемент не найден.
* discard(element): Удаляет элемент из множества, если он есть. Не вызывает ошибку, если элемент не найден.
* pop(): Удаляет и возвращает произвольный элемент из множества.
* clear(): Удаляет все элементы из множества.

Проверка принадлежности:

* in: Проверяет, присутствует ли элемент в множестве.
* not in: Проверяет, отсутствует ли элемент в множестве.

Операции с множествами:

* union(other\_set): Возвращает новое множество, содержащее все элементы из обоих множеств.
* intersection(other\_set): Возвращает новое множество, содержащее только элементы, присутствующие в обоих множествах.
* difference(other\_set): Возвращает новое множество, содержащее элементы, присутствующие только в первом множестве, но не во втором.
* symmetric\_difference(other\_set): Возвращает новое множество, содержащее элементы, присутствующие только в одном из множеств, но не в обоих.

Пример

a = set(data.columns)

b = set(['DateoFdeath', 'name'])

c = a.intersection(b)

print(c)

Другие функции:

* len(set): Возвращает количество элементов в множестве.
* copy(): Создает и возвращает копию множества.
* update(iterable): Добавляет все элементы из итерируемого объекта в множество.
* intersection\_update(other\_set): Изменяет исходное множество, удаляя элементы, не присутствующие в other\_set.
* difference\_update(other\_set): Изменяет исходное множество, удаляя элементы, присутствующие в other\_set.
* symmetric\_difference\_update(other\_set): Изменяет исходное множество, удаляя элементы, присутствующие в обоих множествах, и добавляя элементы, присутствующие только в одном из них.

Магические методы:

* |: Union (объединение).
* &: Intersection (пересечение).
* -: Difference (разность).
* ^: Symmetric difference (симметричная разность).
* <=: Подмножество.
* >=: Надмножество.
* ==: Равенство.
* !=: Неравенство.

Примеры:

*# Создание множества*

my\_set = {1, 2, 3, 4, 5}

*# Добавление элемента*

my\_set.add(6)

*# Удаление элемента*

my\_set.remove(2)

*# Проверка принадлежности*

print(5 in my\_set) *# True*

print(7 in my\_set) *# False*

*# Объединение множеств*

set1 = {1, 2, 3}

set2 = {3, 4, 5}

union\_set = set1.union(set2)

print(union\_set) *# {1, 2, 3, 4, 5}*

*# Пересечение множеств*

intersection\_set = set1.intersection(set2)

print(intersection\_set) *# {3}*

## Замороженное множество (frozenset)

(неизменяемый, неупорядоченный, разные типы данных)

## Кортеж (tuple)

(упорядоченный, НЕизменяемый, разная природа)

Тоже самое что список, но неизменяемый.

Кортеж не поддерживает поэлементное присваивание.

a = tuple(“a”, “b”, …)

a = (“a”, “b”, …)

## Итого

изменяемые: множества, списки, словарь.

неизменяемые: число, строка, кортеж

## Функции работы со строками и массивами

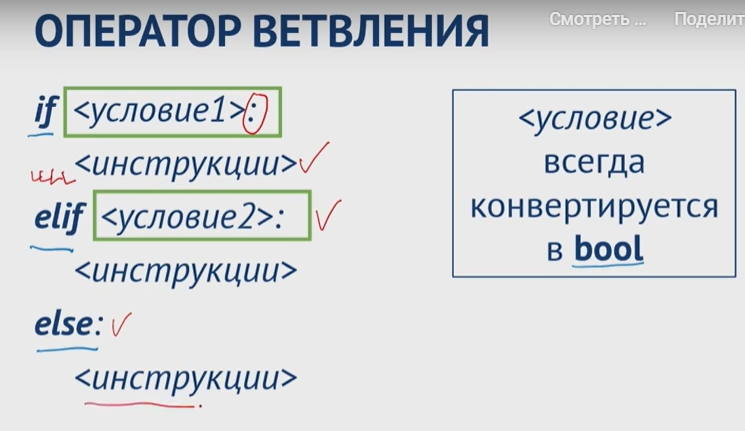
input().strip().split()

strip() - удаляет пробельные символы в начале и в конце строки.

split() - создает список из строк разделенных символом \* или по умолчанию пробелом

# Циклы и ветви

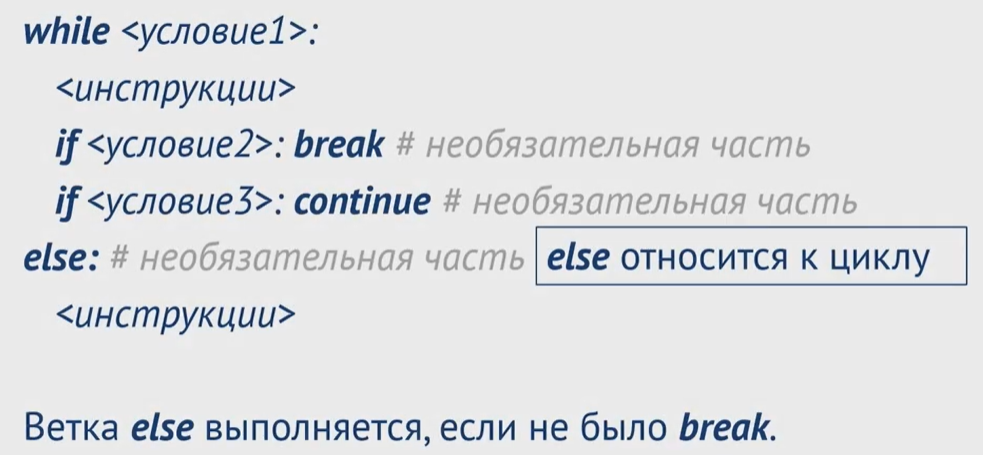
## if



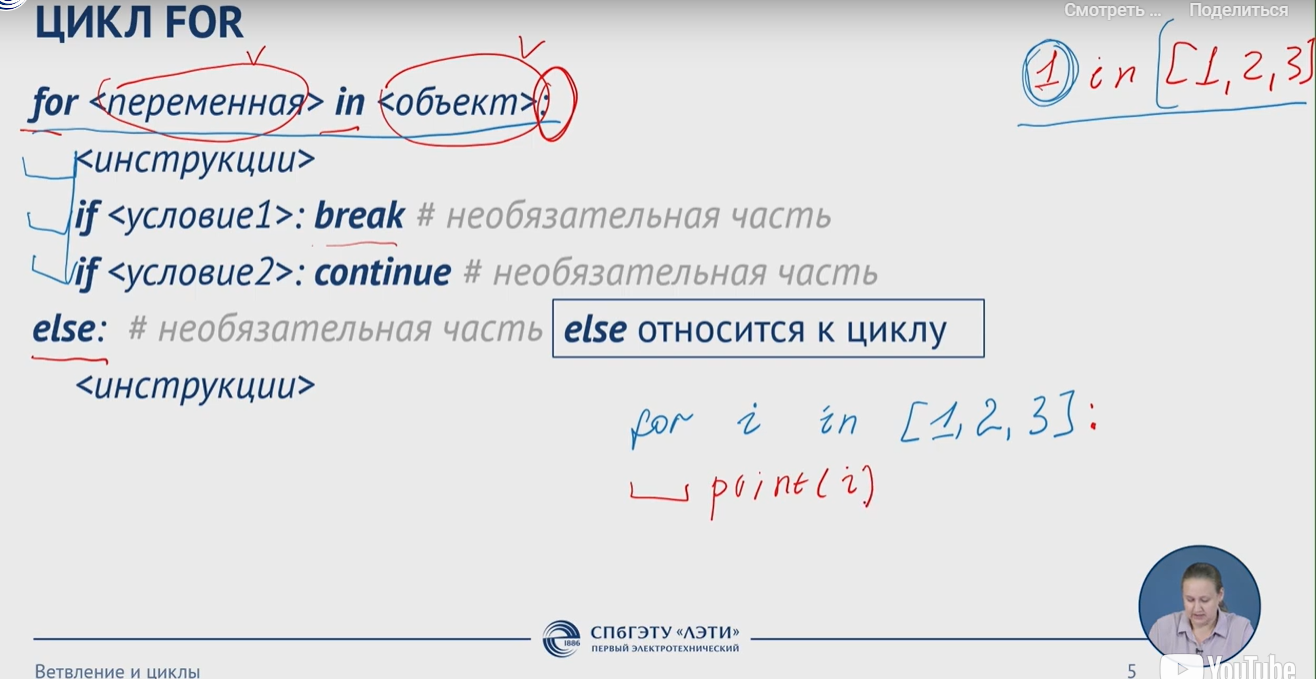
break - выход из цикла

continue - переход на следующую итерацию

## while



## for



## Функции для циклов

range() - функция возвращающая список



range не создает объект, чтобы создать объект нужна вызывающая функция:

list(range(10))

tuple(range(10))

enumerate(list/tuple/set) - создает индекс для каждого объекта множества

используется в цикле for

два способа:

list = [“a”, “b”, … ]

1)

for indx, object in enumerate(list): #кол-во итераций соответствует кол-ву элементов в list

print (“{indx+1} \t {object})

2)

for object in enumerate(list):

print(“индекс: {**object[0]**} \t объект: {**object[1]**}”) # где object[0] создается при

каждой итерации и означает

индекс, а [1] объект

# Собственные функции

def nameoffunc (arg2, arg1 = 0): # \_\_ - именованный аргумент., который имеет

значение по умолчанию, его необязательно

вводить. Именованный значит чтобы его

изменить нужно ввести его имя.

a = arg1 +0

b = arg2 + 1

return a, b

print(nameoffunc(1))

print(type(nameoffunc(1)))

вывод:

(0, 2)

<class 'tuple'> - функция возвращает кортеж

## Возвращаемый тип данных

-> int - тип данных который должна вернуть функция, если функция возвращает тип данных отличающийся от заданного, то ошибка.

**def find(self, index) -> int: *# Предположим, что данные - целые числа***

**if 0 <= index < len(self.data):**

**return self.data[index]**

**else:**

**return None**

## Функции для функций

проверка на правильность аргумента

if not isinstance(a, list):

raise TypeError("Аргумент должен быть списком")

if not all(isinstance(x, int) for x in a):

raise TypeError("Все элементы списка должны быть целыми числами")

if len(a) == 0:

raise ValueError("Список не может быть пустым")

# Import

инфо

модуль math:

<https://pythonworld.ru/moduli/modul-math.html>

import math

math.factorial(x)

import math as m

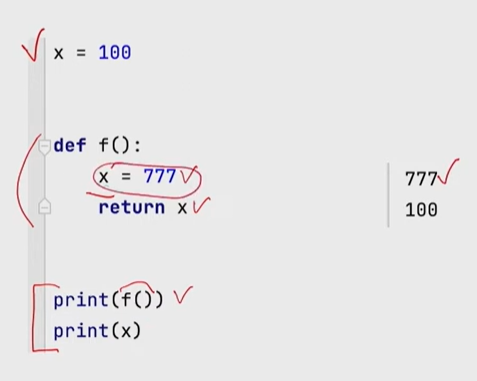
m.factorial(x)

from math import \*

factorial(x)

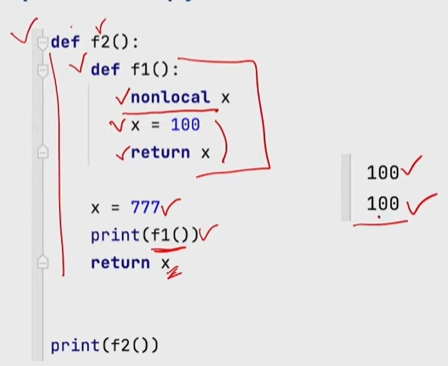
# Области видимости

как работает область видимости

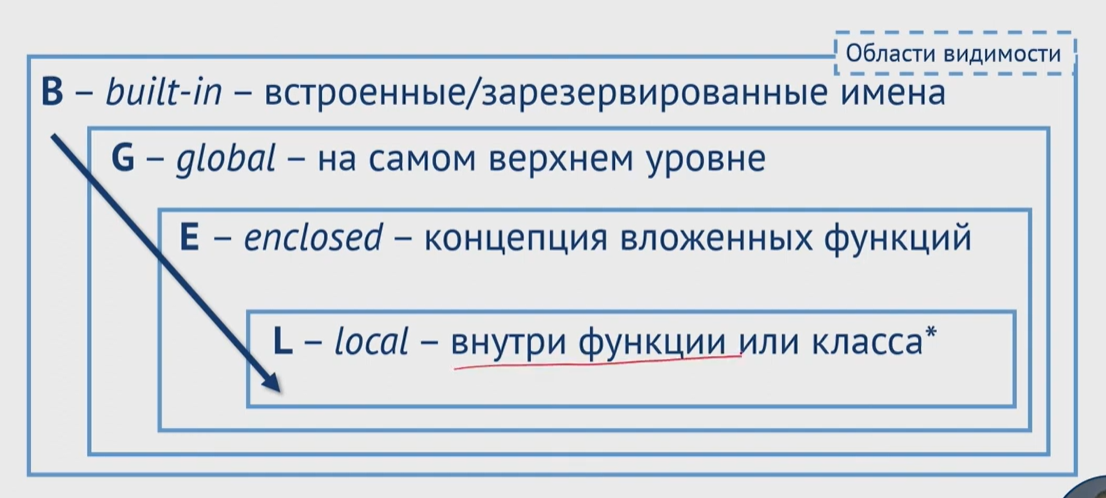


операторы nonlocal и global позволяют изменить значение переменной в ограниченной области видимости

nonlocal позволяет изменить значение перемнной в зоне +1 по отношению к своей, но не имеет силы на зонах +2 и глобальной



global позволяет изменить глобальное значение переменной



# Cсылки в Python

визуализатор ссылок

<https://pythontutor.com/visualize.html#mode=edit>

# ООП

инфо

<https://www.w3schools.com/python/python_inheritance.asp>

class Class: # название с большой буквы

def \_\_name\_\_(self): # инициализация конструктора класса

классы также можно оставлять незаполненными

class Class:

pass

Создание объекта собственного класса:

name = MyClass(...)

## Self

Аргумент self в Python - это специальный аргумент, который используется в методах классов. Он ссылается на сам объект класса, к которому применяется метод.

То есть если этот метод меняет символы в строке, то self отвечает за то, чтобы поменять символы в текущей строке str.

## Конструктор класса

КК- метод вызывающийся при создании объекта автоматически

инфо конструктор класса

<https://younglinux.info/oopython/init>

class Dog:

def \_\_init\_\_(self, name, breed):

self.name = name

self.breed = breed

def bark(self):

print(f"Гав-гав, я {self.name}, собака породы {self.breed}!")

my\_dog = Dog("Рекс", "Овчарка") # Создаем объект Dog

my\_dog.bark()

значения по умолчанию (если не указывать их, то будут заданы по умолчанию)::

class Point:

def \_\_init\_\_(self,x=0, y=0):

self.x = x

self.y = y

## Инкапсуляция

инкапсуляция - это принцип ООП, который позволяет создавать поля и методы, недоступные к вызову для пользователя - это помогает не нарушать работу класса. и таким образом пользователь работает только с интерфейсем класса.

приватные поля и методы создаются с помощью \_\_ : \_\_a = 50; def \_\_func():

пример

class Phone:

username = "Kate" # public variable

\_\_how\_many\_times\_turned\_on = 0 # private variable

def call(self): # public method

print( "Ring-ring!" )

def \_\_turn\_on(self): # private method

self.\_\_how\_many\_times\_turned\_on += 1

print( "Times was turned on: ", self.\_\_how\_many\_times\_turned\_on )

my\_phone = Phone()

my\_phone.call() # ошибки не вызовет

print( "The username is ", my\_phone.username ) # ошибки не вызовет

my\_phone.turn\_on() # ошибка компиляции

my\_phone.\_\_turn\_on() # ошибка компиляции

print( “Turned on: “, my\_phone.\_\_how\_many\_times\_turned\_on) # ошибка компиляции

print( “Turned on: “, my\_phone.how\_many\_times\_turned\_on) # ошибка компиляции

Но можно обойти это ограничение в пайтон, если прописать метод немного по другому:

my\_phone = Phone()

my\_phone.\_Phone\_\_turn\_on()

my\_phone.\_Phone\_\_serial\_number = "44.55.66"

print( "New serial number is ", my\_phone.\_Phone\_\_serial\_number )

поэтому в пайтон нет полной инкапсуляции.

## Наследование

Наследование позволяет перенести все поля и методы одного класса другому так, чтобы другой мог дополнить и расширить функционал базового класса (добавить некоторые методы и поля)

В дочернем (который наследует) классе будут доступны все поля и методы родителя.

class Phone:

net = ‘3g’

def call():

print(‘ring ring’)

class Iphone(Phone):

wifi = ‘wlan’

net2 = ‘5g’

def internet()

## Полиморфизм

Можно переопределять все методы, включая магические методы родительских классов.

class Shape:

def \_\_init\_\_(self, color):

self.color = color

def area(self):

raise NotImplementedError("Area must be implemented by subclass")

def \_\_str\_\_(self):

return f"Shape (color: {self.color})"

class Rectangle(Shape):

def \_\_init\_\_(self, width, height, color):

super().\_\_init\_\_(color) # Вызов конструктора базового класса

self.width = width

self.height = height

def area(self):

return self.width \* self.height

def \_\_str\_\_(self):

return f"Rectangle (width: {self.width}, height: {self.height}, color: {self.color})"

class Circle(Shape):

def \_\_init\_\_(self, radius, color):

super().\_\_init\_\_(color) # Вызов конструктора базового класса

self.radius = radius

def area(self):

return 3.14159 \* self.radius\*\*2

def \_\_str\_\_(self):

return f"Circle (radius: {self.radius}, color: {self.color})"

# Полиморфизм в действии

shapes = [Rectangle(5, 10, "red"), Circle(5, "blue"), Rectangle(3, 7, "green")]

for shape in shapes:

print(shape) # Вывод: Rectangle, Circle, Rectangle

print(f"Area: {shape.area()}") # Вычисление площади для каждого объекта

Важно, что можно модифицировать такие функции как append() для списков и многие другие фундаментальные функции.

Например этот код позволяет добавлять в список только нечетные числа:

class MyClass(Class):

def append(a):

if a%2 != 0:

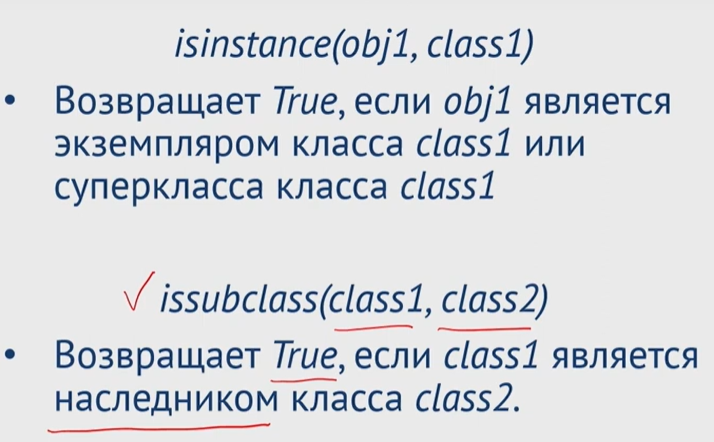
super().append(a)

else:

print(“error”)

## Функции для классов

### проверка sub (isinstance)



### super()

функция super() позволяющая обратиться к методу родителя

<https://www.programiz.com/python-programming/methods/built-in/super>

Пример вызова метода родителя в качестве его конструктора.

class A:

parent pole = ‘remember parents’

def \_\_init\_\_(self, name = ‘name’):

self.name = name

class B(A):

def \_\_init\_\_(self, name, age):

super().\_\_init\_\_(name)

self.age = age

В Python 2, super() принимал два аргумента: super(X, self). В Python 3 этот синтаксис все еще работает, но рекомендуется использовать super() без аргументов.

Где super(X, self)

X - метод наследник, к родителю которого нужно обратится.

self - self

### Магические методы!!!!

def \_\_init\_\_(a,b): # конструктор класса, выполняет операцию при создании объекта

pass

def \_\_str\_\_(a,b): # метод, срабатывающий, когда print(obj), где obj - объект класса

return “{} {}”.format(a,b)

def \_\_len\_\_(): # скрытый магический метод определенный для каждого

встроенного типа данных (класса). При этом явная функция len()

обращается к \_\_len\_\_() для каждого встроенного класса и

поэтому каждый встроенный класс по разному может

реализовать счёт элементов

Функция len() сначала проверяет существует ли метод \_\_len\_\_() у обращаемого класса, если есть, то вызывает его.

### Перегрузка операторов

<https://pythonworld.ru/osnovy/peregruzka-operatorov.html>

Инфо

Во многих встроенных функциях Пайтон:

метод \_\_str\_\_() вызывается при использовании print()

## Детали

### Осторожно со списками

class Counter:

count = 0

Когда мы работаем с неизменяемыми типами полей, то изменяя это поле у созданного объекта класса мы действуем независимо от самого класса

a = Counter()

a.count =10

print(a.count, Counter.count)

результат будет : 10 0

в случае с изменяемым типом данных:

class Counter:

count = [1]

a = Counter()

a.count.append(5)

print(a.count, Counter.count)

результат будет: [1, 5] [1, 5]

что значит меняя поле изменяемого типа данных мы изменяем это поли и у самого класса, что опасно

Решение этого: нужно перед изменением поля с изменяемым типом данных создать новый объект:

class Counter:

count = [1]

a = Counter()

a.count = []

a.count.append(5)

print(a.count, Counter.count)

результат будет: [5] [1]

### Переопределение методов родителя

class MyClass(Class): # нужно обращение к родительскому

def append(a): классу через super(), чтобы избежать рекурсии.

if a%2 != 0:

super().append(a)

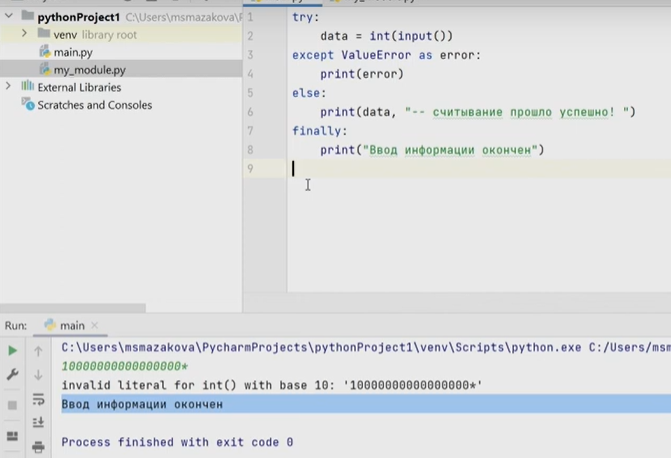
else:

print(“error”)

# Работа с ошибками и исключениями

## try-except

Эта конструкция позволяет сделать так, чтобы вместо ошибки (если они совпадает с указанной нами) появлялся определенный текст или что то ещё другое.



try:

код с возможной ошибкой

except \*тип ошибки\* as error:

print(error)

else:

\*код если ошибки нет например\*

finally:

\*выполнение кода даже после получения ошибки\*

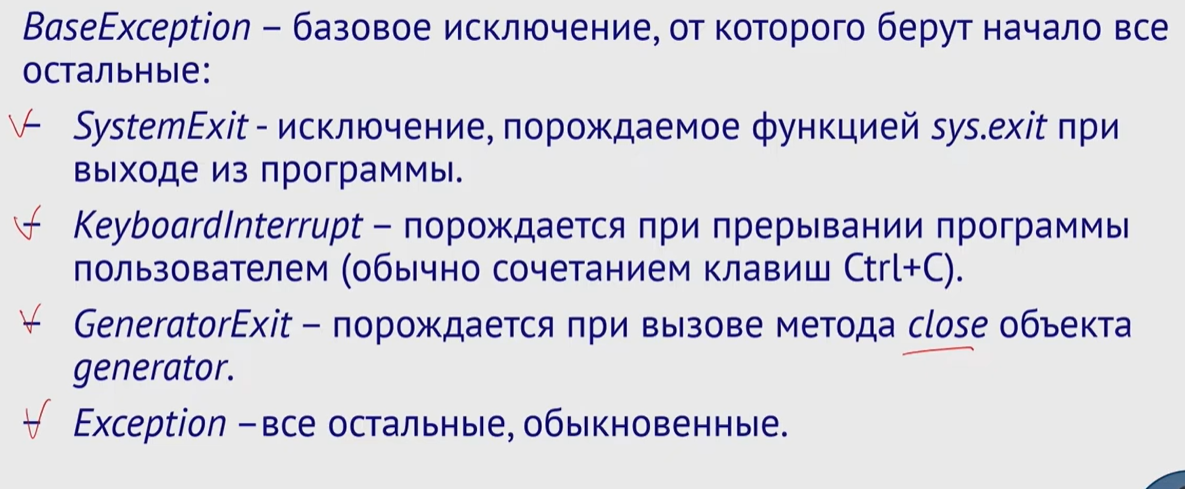
## raise

Эта конструкция позволяет выдать конкретную ошибку с текстом

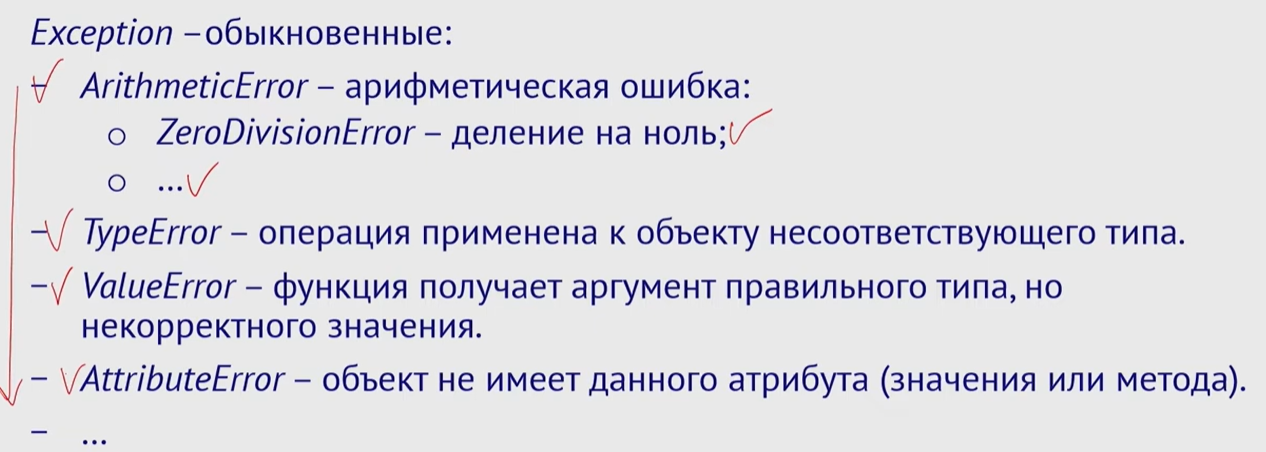
if a is not int:

raise TypeError(‘поступило не число’)

Иерархия ошибок



для Exception



можно посмотреть список подклассов через

b = baseException.\_\_subclasses\_\_()

print(b)

# Функциональное программирование

аксиомы функционального программирования

* нет циклов
* данные неизменяемые
* программа - это совокупность чистых функций
* можно работать с функциями высшего порядка

чистая функция

* не изменяет переменных
* не имеет побочных эффектов

## Итератор и итерируемый объект

Для того, чтобы сохранить выполнение аксиом функционального программирования вводятся новые функции итераторов

Итерируемый объект (iterable) - список/ множество/ кортеж/ словарь

L = [1, 2, 3]

Итератор (iterator) - функция, которая копирует элементы из итерируемого объекта начиная с первого.

iter\_L = iter(L)

работа с итератором:

### Next()

Для работы с перечислением элементов массива в функциональном программировании используется функция next(), которая удаляет предыдущий элемент из итератора и копирует следующий элемент. Next() работает только с итератором iter()

Next(L) # ошибка

Next(iter\_L) # в итератор передаётся следующий элемент

### Перевод

для перевода итератора в список и наоборот:

iter\_L = iter(L)

L = list(iter\_L)

## Создание iter с yield

yield — это ключевое слово в Python, которое используется в функциях для создания **генераторов**. Генераторы позволяют функции возвращать значения по одному за раз, вместо того чтобы возвращать весь результат сразу, как это делает обычная функция с return. В отличие от return, который завершает выполнение функции, yield приостанавливает её выполнение и сохраняет текущее состояние, чтобы при следующем вызове можно было продолжить с того же места.

def simple\_generator():

yield 1

yield 2

yield 3

gen = simple\_generator()

# Получаем значения по одному:

print(next(gen)) # 1

print(next(gen)) # 2

print(next(gen)) # 3

## Функции в ФП

## 1) filter

filter(<function>, <iterable>)

Аргументы:

Function - какая то функция, которая получает себе в аргумент элемент из iterable. function должна возвращать только true или false.

Iterable - список, множество, кортеж итд.

Возвращает:

Итератор, содержащий новый список, состоящий из элементов из iterable, для

которых function вернула значение true.

## 2) map

map(<function>, <iterable\_1> [, <iterable\_2>, …])

Аргументы:

применяет function ко всем элементам itarable\_1 [, iterable\_2, …]

* function должна возвращать значение

в аргументе должно быть столько списков (itarable\_1 [, iterable\_2, …]), сколько принимает аргументов function

map возвращает…

Применение:

можно модифицировать элементы списков с помощью функций

Сделать все элементы списка строками: map(str, list)

## 3) zip

zip(<iterable\_1>,<iterable\_2>[, iterable\_3, …])

Аргументы:

Zip принимает несколько списков и склеивает элементы по индексам в один

кортеж.

Пример:

iterable\_1 = [1, 2, 3]

iterable\_2 = [Маша, Ваня, Ира]

iterable\_3 = [девочка, мальчик, девочка]

zip(iterable\_1, iterable\_2, iterable\_3) →

((1, “Маша”, “девочка”), (2, “Ваня”, “мальчик”), (3, “Ира”, “девочка”))

Возвращает:

Zip возвращает ИТЕРАТОР, позволяющий перечислять пары элементов.

Детали:

Zip останавливается в склеивании, когда доходит до конца самого короткого

списка, остальные элементы отбрасываются.

Применение:

1. это позволяет в цикле легко перечислять объекты:

names = ["Иван", "Мария", "Петр"]

ages = [25, 30, 28]

for name, age in zip(names, ages):

print(f"{name}: {age}")

1. Создание словарей

## 4) Lambda выражения

func = Lambda arg1, arg2, arg3, … : expression

Аргументы:

argi - любые объекты

expression - любое выражение с этими аргументами (arg1 + arg2 \* arg3)

Применение:

Вместо того, чтобы создавать функции (простые, без условий и циклов) можно

воспользоваться лямбда выражениями.

Даже простые выражения не работают в аргументах функций ФП:

def do\_filter(a):

return list(filter(a%2 != 0, a)) // ошибка

нужно

def do\_filter(a):

return list(filter(lambda x: x%2 != 0, a))

# Numpy

инфо

<https://numpy.org/doc/stable/#>

гайд для начинающих

<https://numpy.org/doc/stable/user/absolute_beginners.html>

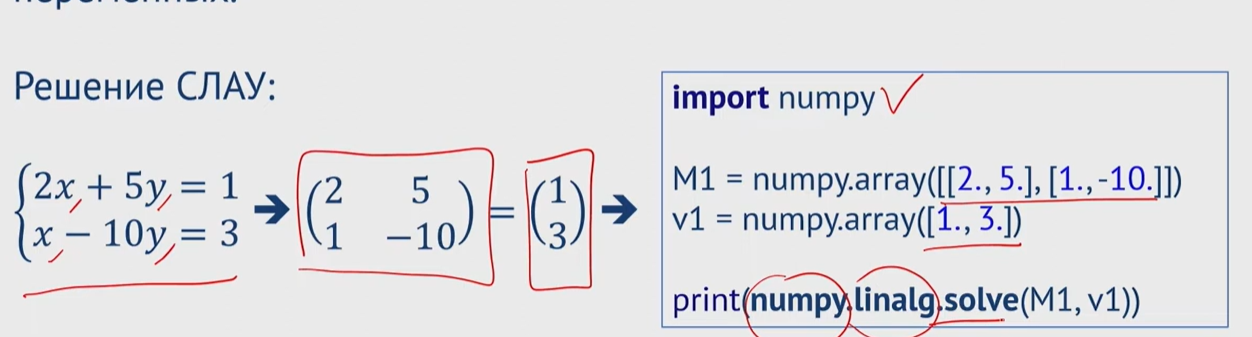
при изменении массива или матрицы всегда будет возвращаться новая матрица, тк numpy работает с неизменяемыми массивами и матрицами

## Создание ndarray

Значения матрицы всегда передаются списком или списком из списков, где каждый внутренний список это СТРОКА.

import numpy as np

data = np.array([[1,2],[3,4]])



Обращение к элементам массива:

a = np.array[[...], [...]]

a[i,j] = 2

## Отличие matrix() от array()

В NumPy numpy.matrix() и numpy.array() - это два разных класса, которые представляют собой многомерные массивы, но с некоторыми ключевыми отличиями:

1. Тип данных:

* numpy.array(): Создает объект ndarray (n-мерный массив). Это наиболее универсальный тип данных для работы с массивами в NumPy.
* numpy.matrix(): Создает объект matrix (матрица). Матрица - это специальный случай двумерного массива, оптимизированный для операций с матрицами, таких как умножение матриц.

2. Операции:

* numpy.array(): Поддерживает широкий спектр операций с массивами, включая математические операции, индексирование, срезы и многое другое.
* numpy.matrix(): Оптимизирован для матричных операций, таких как умножение матриц (\*), транспонирование (T), инверсия (I) и т.д. При использовании стандартных операторов +, -, / numpy.matrix будет вести себя как двумерный массив.

3. Размерность:

* numpy.array(): Может быть n-мерным массивом (любого числа измерений).
* numpy.matrix(): Всегда двумерным массивом.

## Методы

### np.sum(a, axis =1/0)

axis 1 - строка

axis 0 - столбец

Линейная алгебра

# Pandas

инфо

<https://pandas.pydata.org/docs/>

## series

(одномерный массив - столбец)

создание series:

pd.Series(a,b) # a - одномерный массив, b - индексы для этого массива

Пример:

my\_series = pd.Series(

[random.randint(0, 10) for \_ in range(5)], index=['a', 'b', 'c', 'd', 'e'])

print(my\_series.head()) # вывод на экран первых 5 строк

print(my\_series[['a', 'c', 'e']]) # обращение по индексам

print(my\_series.max())

print(my\_series.min())

print(my\_series.mean())

## DataFrame

(двумерный массив - таблица данных)

создание:

df = pd.DataFrame(a, b) # а - двумерный массив (данные), b - названия к столбцам

Пример:

from sklearn.datasets import load\_iris

import numpy as np

import pandas as pd

iris = load\_iris()

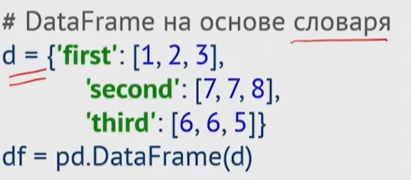
data=np.c\_[iris['data'], iris['target']]

columns=iris['feature\_names'] + ['target']

df = pd.DataFrame(data=np.c\_[iris['data'] +iris['target']], columns=iris['feature\_names'] + ['target'])

print(df.head())

### Создание с помощью словоря



## Навигация по dataframe

Метод df.loc[] в Pandas используется для доступа к данным в DataFrame по меткам строк и столбцов. Он позволяет вам выбирать данные, используя строковые метки (имена строк и столбцов) вместо числовых индексов.

Синтаксис:

df.loc[строковые\_метки\_строк, строковые\_метки\_столбцов]

Пример:

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, 30, 28, 22],

'City': ['New York', 'London', 'Paris', 'Tokyo']}

df = pd.DataFrame(data)

print(df)

*# Выбор строки по метке 'Alice'*

print(df.loc['Alice'])

*# Выбор столбца по метке 'Age'*

print(df.loc[:, 'Age'])

*# Выбор строк по меткам 'Alice' и 'Charlie', и столбца 'Age'*

print(df.loc[['Alice', 'Charlie'], 'Age'])

*# Выбор всех строк и столбцов 'Name' и 'City'*

print(df.loc[:, ['Name', 'City']])

Ключевые особенности:

* Строковые метки: df.loc[] использует строковые метки для строк и столбцов, что делает код более читаемым, особенно когда у вас много данных.
* Срезы: df.loc[] поддерживает срезы строк, например, df.loc['Alice':'Charlie'], чтобы выбрать все строки от ‘Alice’ до ‘Charlie’.
* Изменение данных: df.loc[] может использоваться не только для выборки, но и для изменения значений в DataFrame.

## Извлечение данных из DataFrame

print(df)

print(df[“название столбца”]) / print(df[“столбец 1”, “столбец 2”])

print(df[“название столбца”].unique()) - выводит только неповторяющиеся элементы

print(df.loc[[54, 55, 56], 'target']) # извлечение данных столбца target по номеру строки

print(df.iloc[0:10]) # извлечение всех данных по диапазону строчек (не включая 10)

setosa = df[df['target'] == 0.0] # извлечение с условием

## Работа с df

### Индексы

### Тип данных столбца

data.info()

ИЛИ  
  
column\_types = data.dtypes

print(column\_types)

### Изменить тип столбца

data[num\_cols[1]] = data[num\_cols[1]].astype(float)

for col in data.columns:

if data[col].dtype == 'bool':

data[col] = data[col].astype(int)

data.info()

### Группировка данных df

df.groupby("target") - позволяет сгруппировать элементы с одинаковым значением столбца target

df.groupby(‘target’)[high].mean() - позволяет получить среднее значение значений столбца [high] элементов с одинаковым значением в ‘target’

### Конкатенация фреймов

конкатенация возможна только если у фреймов одинаковые названия столбцов и одинаковое их количество

df = pd.DataFrame({'first': [1, 2, 3],

'second': [7, 7, 8],

'third': [6, 6, 5]})

df1 = pd.DataFrame({'first': [-7, 0, 1],

'second': [2, -8, 4],

'third': [5, 1, 9]})

df\_res = df.\_append(df1, ignore\_index=True) # игнор индекс для безопасности лучше

print(df\_res) писать его

### Все строки/столбцы

data.columns

data.index

### Добавление столбцов в df

df1 = pd.DataFrame() пустой фрейм

df1['имя столбца'] = pd.Series([1,2,3]) - добавление нового столбца и

передача нескольких данных в этот столбец

примеры:

df1['random'] = pd.Series(random.randint(-100,100) for i in range(10))

df1['ones'] = pd.Series([1] \* 10)

Можно добавлять столбец с данными, основанными на данных в других столбцах (математические операции)

df1['new\_column'] = (df1['random'] + 3) \* df1['ones']

### Удаление столбцов

import pandas as pd

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})

1. df = df.drop('B', axis=1)
2. del df['B']
3. df = df[['A', 'C']]
4. df.pop('B') # Этот метод удаляет столбец и возвращает его как Series.

print(df)

### Добавление строк в df

# df.index - атрибут объекта df хранящий список индексов всех строк

df.loc[len(df.index)] = [random.randint(-10,10) for i in range(len(df.columns))]

### Замещение символов в строке

Replace or Remove Empty Strings:

Replace with NaN:

df['column\_name'] = pd.to\_numeric(df['column\_name'], errors='coerce')

This will convert non-numeric values (including empty strings) to NaN (Not a Number).

### NaN

1. Удаление строк с NaN:

dropna(): Этот метод удаляет все строки, содержащие хотя бы одно NaN.

df.dropna(inplace=True)

inplace=True модифицирует исходный DataFrame. Если вы хотите сохранить исходный DataFrame, удалите inplace=True.

dropna(subset=['column\_name']): Удаляет строки, содержащие NaN только в указанном столбце.

df.dropna(subset=['column\_name'], inplace=True)

2. Удаление столбцов с NaN:

dropna(axis=1): Удаляет все столбцы, содержащие хотя бы одно NaN.

df.dropna(axis=1, inplace=True)

dropna(axis=1, subset=['column\_name']): Удаляет столбцы, содержащие NaN только в указанном столбце.

df.dropna(axis=1, subset=['column\_name'], inplace=True)

3. Заполнение NaN:

fillna(): Заполняет NaN указанным значением.

df.fillna(0, inplace=True)

Можно использовать method='ffill' для заполнения NaN предыдущим значением в столбце, method='bfill' - следующим.

interpolate(): Заполняет NaN с помощью интерполяции.

df.interpolate(method='linear', inplace=True)

заполнение медианным значением:

data['dateOfBirth'].fillna(data['dateOfBirth'].median())

# MatPlotLib



import matplotlib.pyplot as plt

Построить график

plt.plot(x,y)

plt.show()

Изменить размер окна

plt.figure(figsize=(6, 4))

plt.plot(x,y) - график с линиями

plt.scatter(x,y) - график с отдельными точками

построить на отдельных графиках

plt.figure()

plt.plot(x,y)

plt.figure()

plt.plot(x,y)

## Subplot

import matplotlib.pyplot as plt

import numpy as np

# Plot 1:

x1 = np.array([1, 2, 3, 4])

y1 = np.array([10, 20, 25, 30])

plt.subplot(1, 2, 1)

plt.plot(x1, y1)

# Plot 2:

x2 = np.array([1, 2, 3, 4])

y2 = np.array([30, 25, 20, 10])

plt.subplot(1, 2, 2)

plt.plot(x2, y2)

plt.show()

ИЛИ

import matplotlib.pyplot as plt

import numpy as np

# Создаем фигуру с размером (9, 4)

fig1 = plt.figure(figsize=(9, 4))

# Plot 1:

x1 = np.array([1, 2, 3, 4])

y1 = np.array([10, 20, 25, 30])

# Добавляем первый подграфик (1 строка, 2 столбца, позиция 1)

ax1 = fig1.add\_subplot(1, 2, 1)

ax1.plot(x1, y1)

ax1.set\_title("Plot 1") # Опционально: добавляем заголовок

# Plot 2:

x2 = np.array([1, 2, 3, 4])

y2 = np.array([30, 25, 20, 10])

# Добавляем второй подграфик (1 строка, 2 столбца, позиция 2)

ax2 = fig1.add\_subplot(1, 2, 2)

ax2.plot(x2, y2)

ax2.set\_title("Plot 2") # Опционально: добавляем заголовок

# Показываем фигуру

plt.tight\_layout() # Чтобы подграфики не перекрывались

plt.show()

ИЛИ

fig = plt.figure(figsize = (14,4))

a = 1

b=2

if a == 1:

ax2 = fig.add\_subplot(131, projection='3d')

if b==2:

ax2 = fig.add\_subplot(132, projection='3d')

### subplot с помощью ax[0,0]

extent = [xmin, xmax, ymin, ymax]

fig, axs = plt.subplots(2, 2, figsize=(10, 8))

# Отображение функций

axs[0, 0].imshow(F\_smooth, extent=[-5, 5, -5, 5], origin='lower', cmap='viridis')

axs[0, 0].set\_title("Гладкая функция")

axs[0, 1].imshow(F\_noisy, extent=[-5, 5, -5, 5], origin='lower', cmap='viridis')

axs[0, 1].set\_title("Шумная функция")

# Отображение нормы градиента

axs[1, 0].imshow(grad\_smooth, extent=[-5, 5, -5, 5], origin='lower', cmap='magma')

axs[1, 0].set\_title("Градиент гладкой функции")

axs[1, 1].imshow(grad\_noisy, extent=[-5, 5, -5, 5], origin='lower', cmap='magma')

axs[1, 1].set\_title("Градиент шумной функции")

## Создание кластеров

from sklearn.datasets import make\_blobs

X, y = make\_blobs(n\_samples=1000, centers=[[-2,0.5],[3,-0.5]], cluster\_std=1, random\_state=42)

#y = y.reshape(-1, 1)

colors = ("red", "green")

colored\_y = np.zeros(y.size, dtype=str)

for i, cl in enumerate([0,1]):

colored\_y[y.ravel() == cl] = str(colors[i])

plt.figure(figsize=(15,10))

plt.scatter(X[:, 0], X[:, 1], c=colored\_y)

plt.show()

визуализация предсказания

clf = MyElasticLogisticRegression(0.1, 0.1)

clf.fit(X, y, epochs=1000)

w = clf.get\_weights()

from matplotlib.colors import ListedColormap

plt.figure(figsize=(15,8))

eps = 0.1

xx, yy = np.meshgrid(np.linspace(np.min(X[:,0]) - eps, np.max(X[:,0]) + eps, 200),

np.linspace(np.min(X[:,1]) - eps, np.max(X[:,1]) + eps, 200))

Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

cmap\_light = ListedColormap(['#FFAAAA', '#AAFFAA'])

plt.pcolormesh(xx, yy, Z, cmap=cmap\_light)

plt.scatter(X[:, 0], X[:, 1], c=colored\_y)

# Plotly

import plotly.graph\_objects as go

fig = go.Figure() # Создаем фигуру

if modeldata == 1:

# Добавляем точки предсказаний модели

fig.add\_trace(go.Scatter3d(

x=x\_meta\_pca[:, 0],

y=x\_meta\_pca[:, 1],

z=predictions\_meta,

mode='markers',

marker=dict(

size=5,

color=predictions\_meta, # Цвет точек соответствует предсказаниям

colorscale='Plasma', # Цветовая карта Plasma

colorbar=dict(title='Network Output') # Добавляем цветовую шкалу

),

name='Predictions' # Имя для легенды

))

if truedata == 1:

# Добавляем точки реальных значений

fig.add\_trace(go.Scatter3d(

x=x\_meta\_pca[:, 0],

y=x\_meta\_pca[:, 1],

z=val\_loader.dataset.y.detach().numpy(),

mode='markers',

marker=dict(

size=5,

color='orange', # Оранжевый цвет для реальных значений

),

name='True Data' # Имя для легенды

))

# Настройка графика

fig.update\_layout(

title='Network Output in PCA Space (Interactive 3D)',

scene=dict(

xaxis\_title='PCA Component 1',

yaxis\_title='PCA Component 2',

zaxis\_title='Values'

)

)

# Показываем график

fig.show()

Next

# Декораторы