# **Типы Данных**

**my\_str = ‘text’ - “text”** - **“””text”””** (многострочная строка) строки <class ‘str’>

**my\_str = f “ text {var1} and {var2} ”**  f-строки, внутрь текста добавляються значения переменных

**my\_int = 0 , 5 , -3 , 1\_000\_000** целое число <class ‘int’>

**my\_float = 3.5 , 7.2 , 8.1** числа с плавающей точкой <class ‘float’>

**my\_bool = True , False** логический <class bool>

**my\_list = [ 1, True, -5, “text” ]** список <class ‘list’>

**my\_dict = { ‘one’= 1, ‘two’= ‘text’ }**  словарь <class ‘dict’>

**my\_tuple = ( 1, -3 , True, ”text” )**  кортеж <class ‘tuple’>

**my\_set = { 1, -3, False, “text” }** набор <class ‘set’>

**my\_range = range ( 0 , 5 , 2 )** диапазон <class ‘range’>

**my\_complex = 3 + 2j ( 3 + 2j )** комплексное число <class ‘complex’>

**int ( a )** конвертация значения в тип int (не изменяет)

**float ( a )** конвертация значения в тип float (не изменяет)

**bool ( a )** конвертация значения в тип bool. **0 , [ ] , None** - False, все остальное True (не изменяет)

**str ( a )** конвертация значения в строку (не изменяет)

**list ( my\_str - my\_dict - my\_tuple - my\_set)** конвертация в список, при конвертации словаря передаються только ключи (не изменяет)

**dict ( my\_list ( ->my\_tuple) )** конвертация списка в словарь (не изменяет)

**tuple ( my\_list - my\_str - my\_dict)** конвертация в список (не изменяет)

**set ( my\_list - my\_str - my\_dict)**  конвертация в набор (не изменяет)

# **Встроенные Функции**

**print(5) print(‘text’) print (3+2)** выводит данные в консоль

**a = input ( “Enter: ” )** просит пользователя ввести данные та записывает их в переменную

**dir ( )**  возвращает глобальные функции и глобальные переменные + некоторые магические методы

**dir ( a )** возвращает все атрибуты указаного обьекта

**dir (\_\_builtins\_\_)** возвращает список из всех встроенных функций в python

**id ( a )** возвращает индентификатор указаного обьекта

**hex ( id ( a ) )** возвращает значение в памяти в формате base16

**type ( a )** возвращает тип указаного обьекта. Если передаем класс тогда возвращает < class ‘type’>

**zip ( my\_list , my\_str ) zip ( my\_list, my\_str, my\_set , my\_range )** обьединяет групу последовательностей в один набор кортежей. Получается <zip object at 0x1044bc100> . Конвертировать в **dict** можно только при обьединении двох последовательностей (не изменяет).

**globals( )** возвращает все глобальные переменные и их значения

# **Мои Функции**

**def my\_fn ( a , b ) : c = a + b return c** созданая функция

**def my\_fn ( ): pass**  проходит мимо = не выполняет никаках действий, самая короткая функция. Возвращает None

**def my\_fn () -> int :** указываем тип возвращаемого значения функции

**return** если не указывать возвращаемое значение функции, тогда возвращается None

**def my\_fn( my\_list - my\_dict - my\_set) :** при передачи изменяемых обьектов в функцию -> сами обьекты изменяються (адрес не изменяется)

**def my\_fn( my\_int - my\_bool - my\_tuple) :** при передачи неизменяемых обьектов в функцию -> создаються новые обьекты (адрес изменяется)

**my\_fn ( b = 3 , a = 7 )** ключевые аргументы, указываем значения для каждого аргумента функции, тогда порядок аругументов не важен

**def my\_fn1( my\_fn2 ) : my\_fn2( ) my\_fn1( my\_fn2 )**  callback функция

**def my\_fn ( ): “””text””” return** добавление описания функции (docstring) , описание ввидно в подсказке при наведении

**def my\_fn( a , b ) : print( dir( ) )** возвращает список из всех локальных переменных внутри функции.

↓ **Аргументы Функции** ↓

**my\_fn ( 3 , 5 )** вызов созданной функции и передача аргументов внутрь

**def my\_fn( a = 3, b = 5 ):** параметры функции по умолчанию. Аргументы по умолчанию можно указывать только слева на право

**def my\_fn1( a = my\_fn2 ( ) ) :** передача по умолчанию call back функции

**def my\_fn ( a : int ) :** подсказка для програмиста, какой тип данных нужно передавать

**def my\_fn( a : int = 5 ) :** и подсказка з типов данных и значение по умолчанию вместе

**def my\_fn (a = 5 ) -> int :** явно указываем тип возвращаемого значения из функици

**↓** **Большое Количество Аргументов**  **\* \*\*** **↓**

**\* \*\* операторы розпаковки**

**a , b , c = [ 1 , 2 , 3 , 4 ] -->**

**\*a, b, c = [ [ 1 , 2 ] , 3 , 4 ] a, \*b, c = [ 1 , [ 2 , 3 ] , 4 ] a, b, \*c = [ 1, 2 [ 3 , 4 ] ]**

**def my\_fn (\* my\_tuple** (\*args) **) :** можно вызвать функцию с любым количеством аргументов. Передается **кортеж** с елементами внутрь функции

**def my\_fn( \*\* my\_dict** (\*kwargs) **): def my\_fn( key1 = 1, key2 = 2 )** передается **словарь** с любым количеством пар внутрь функции. Можно использовать только ключевые аргументы.

↓ **Лямбда Функции** ↓

**a = lambda x : x + 2** создания лямбда функции, где x - ето аргумент , а x + 2 ето выражение значение которого возвращается

**def my\_fn( func = lambda x : 1)** передача лямбда функции как аргумент по умолчанию в другую функцию. Лямбда функция возвращает значение 1

↓ **Области Видимости Переменной** ↓

**global my\_var -> my\_var = 3** создание глобальной переменной внутри функции, создается только при присваивании значения. Присваивать значение можно только на другой строке кода. Можна приопределять значения глобальных переменных с таким же названием.

**nonlocal my\_var** указываем, что будем использовать переменную из внешней функции (может быть аргументом). Не создаеться новая переменая

↓ **Call Back Функции** ↓

**def my\_fn1 ( my\_fn2 ) :** когда функция передаеться внутрь другой функции, называеться call back функция

**my\_fn\_1 ( my \_fn2 (value) )** вызов етой функции и передача внутрь другой call back функции. Сначала выполняеться внутреняя функция my\_fn2 , а потом внешняя my\_fn1

↓ **Замыкания** **Функций** ↓

**def my\_fn1 (value) : def my\_fn2: print( value ) return my\_fn2** создание замыкания, my\_fn2 - ето и есть функция замыкания

**a = my\_fn1 ( value )** присвоение функции переменной a

**a ( )** вызов етой функции, значение value сохраняеться

↓ **Декораторы** **Функций** ↓

**def my\_decorator( my\_fn ): def wrapper ( value ): my\_fn( value ) return wrapper** создан декоратор для функции my\_fn, додает функционал при вызове функции

**@my\_decorator def my\_fn( ):** или **my\_fn = decorator( my\_fn )** присваивание декоратора функции

**@my\_decorator( func = lambda my\_fn :True )** передаем по умолчанию лямбда функцию в декоратор, если ее нету то по умолчанию передаем True

↓ **Функция возвращает Генератор** ↓

**def my\_fn()**

**yield 5**

ключевое слово yield , как return -> только возвращает обьект генератор. Генератор позволяет возвращает итерируемый обьект и взаемодействовать с ним. Позволяет использовать функцию next() для каждого елемента генератора

# **Работа с Файлами**

**my\_file = open (“/123.txt”, “w”)** открытие файла для добавления та удаление текста

**my\_file = open (“/123.txt”, “w+”)** открытие файла для добавления или дулаения текста, если файла нет то он создатся в указаной директории

**my\_file = open (“/123.txt”, “a”** или **“a+”)** открыте файла только для добавления текста

**my\_file = open (“/123.txt”, “r”** или **“r+”)** открытие файла для чтения

**my\_file = open (“/123.txt”, “r” или “w”, “a” , encoding = “utf-8” )** открываем файл и устанваливаем кодировку utf-8 -> можно отображать и руские и англиские символы

**my\_file.write(“text”)** записывает указаный текст внутрь файла

**my\_file.read( )** или **my\_file.read( )** возвращает весь текст или ограниченое количество символов, который написаный в файле

**with open (“/123.txt”,”r”) as my\_file** менеджер, который автоматически закривает файл после выполнения кода

**my\_file.close ( )** закрытие файла, всегда после выполнения нужно закрывать

**with менеджер контекста(open) as переменая** менеджер контекста для работы с файлами. Не нужно использовать .close(), при завершении работы автоматически закрывает файл

**with open(my\_file) as my\_var: my\_var.write(“hello”)** менеджер контекстов, создан для роботы с файлами. Переменая my\_var хранит в себе файл.

# **Основные Операторы**

**=** оператор присвоения значения

**+** (добавление) **-** (вычетание) **\*** (умножение) **/** (деление) арифметические операторы

**>** (больше) **<** (меньше) **>=** (больше или равно) **<=** (меньше или равно) **==** (равно) **!=** (не равно) операторы сравнения. Сравнивают только значения, а id не сравнивают

**+= -= \*= /= var1 + = var2 var1 = var1 + var2**  сокращеная форма записи для арифметичских операторов

**not** ( не ) **and** ( и ) **or** ( или ) логические операторы, and + or -> операторы короткого замыкания

**not** - меняет значение наоборот. True превращает в False и наоборот

**and** - возвращает первый False при нахождении.

Если все True -> тогда возвращает последний True

**or**  - возвращает первый True при нахождении.   
Если все False -> тогда возвращает последний False

**is** ( есть **->** id1 == id2) **is not** ( не есть **->** id1 != id2 ) операторы тождествености, проверяет индетификаторы в памяти , а не значения (возвращает True - False)

**int is int list is list float is not float bool is not bool** также проверяет равность типов данных. Каждый тип тоже имеет свой адрес

**in** ( в ) **not in** ( нет в ) операторы принадлежности, проверяет есть ли значение в последовательности (возвращает True - False)

**Выражение1 if Условие else Выражение 2 var = print1( ) if 5 > 0 print2( )**  тернарный оператор, можно присваивать возвращаемое значение переменной

**my\_list = [ a for a in range( ) if a%2==0 ]** List Comprehetion

**my\_set = { a for a in range( ) if a%2==0 }** Set Comprehetion

**my\_dict = { a for a in range( ) if a%2==0 }** Dict Comprehetion

# **Математические Операции**

**a % 2**  **a % 3** деления по модулю, 0 - число парное (1 или больше - число не парное)

**a \*\* 2** **a\*\*3** возведение в степень

**a // 2 a // 3** целочисленное деление

**[ 1 , 2 , 3 ] == [ 1 , 2 , 3 ]**  сравнение идет поелементно

**pow ( a, b )** функция возвращает возведеное число в степень a - число, b - степень

**round ( a )** возвращает округленое число до ближайшего целого

**abs ( a )**  возвращает абсолютное значение числа, взвращает число без минуса ( - )

**( a , j ) ( b , j ) = ab + aj + jb - число ( j2 )** умножение комплексных чисел, j в квадрате равно **-**

# **Строки**

**my\_str = “ text \n” n -** перевод на новую строку

**print ( 1 , 2 , 3 , sep = “ ” ) --> 1 sep 2 sep 3 sep**

**print ( 1 , 2 , 3, end = “ ” )** **--> 1 , 2 , 3 end**

**len ( my\_str )** возвращает длину строки

**my\_str [ 3 ] my\_str [ -1 ]** возвращает значение по индексу. В обратной последовательности начинается с **-1**

**my\_str [ 3 : 5]** возвращает все значения от 3 (включительно) по 5 (не включительно) по индексу

**my\_str = ( “text1” “text2” )** обьединяет две строки в одну (не изменяет)

**“text1” == > < “text2”** сравнение идет посимвольно

↓ **Методы Строк** ↓

**my\_str.replace ( “text1” , ”text2” )** возващает измененую строку (не изменяет)

**my\_str.count(“ ”) my\_str.count(“a”)** возвращает количество повторений указаного значения **( )** в строке

**my\_str.lower( ) my\_str.upper( )** возвращает текст в нижем и вверхнем регистре (не изменяет)

**my\_str.capitalize( )** возвращает текст с первой заглавной буквой (не изменяет)

**my\_str.split( )** розделяет указаную строку по пробелам и возвращет список из них

**my\_str.strip( )** **my\_str.strip(my\_letters)** возвращает строку без проблемов перед и после нее или без символом переданых внутрь метода

**my\_str.isdigit( )** проверяет состоит ли вся строка из цифр. Возвращает True(False)

**my\_str.rjust (my\_len , “1”) или my\_str.ljust(my\_len, “h”)**  возвращает измененую строку с указаной дленой и заполненым пустым местом указаными симолами

# **Списки**

**my\_list = [ 1, 2.3, True, [ 1 , 2 ] , “text” ]** создание списка

**my\_var1, my\_var2 = my\_list** розпаковка списка, значения присваиваються указаным переменным (не изменяет)

**my\_var1, \* my\_var2 = my\_list** розпаковка при которой последней переменой присваивается остальные елементы списка в виде нового списка

**my\_list1 = my\_list2 [ : ]** копирование именно елементов списка, а не ссылки на другой список. Ето копирование используя slice

**my\_list1 = my\_list2.copy( ) my\_list1 = list ( my\_list2 )**  создание нового списка с елементами старого

**my\_list [ var1 , var2 , var3 ]** при наполнении списка переменными -> передаються сами значения, а не ети переменные

**my\_list [ 2 ] my\_list[ -1 ]** возвращает елемент списка по указаному индексу, в обратной последовательности начинается с **-1**

**my\_list [ 2 ] = 777** изменение елемента списка по индексу

**my\_list[ : ] my\_list[ 1: ] my list[ :1 ] my\_list[ 2: -1 ]** возвращает елементы списка первое число (включительно), второе (не включительно) -> (не изменяет).

**my\_list [ 2 ] [ “key” ]** возвращает значение по ключу словаря, который лежит внутри списка

**del my\_list [ 2 ]** удаление елемента списка по индексу (изменяет)

**len ( my\_list )** возвращает длину списка

**min( my\_list ) max( my\_list )** возвращает минимальное или максимальное значение из списка (не изменяет)

**sum ( my\_list )** возвращает суму значений в списке (не изменяет)

↓ **Методы Списков** ↓

**my\_list.append ( a )** добавляет в конец списка указаный елемент (изменяет)

**my\_list.pop( 2 ) my\_list.pop( )** удаляет елемент списка по индексу или последний, если без значения (изменяет)

**my\_list.sort( ) my\_list.sort( reverse = True )** сортирует список по возрастанию или убыванию (изменяет)

**my\_list.reverse( )** просто меняет разположение елементов в списке (изменяет)

**my\_list.count( a )** возвращает количество повторений указаного значения в списке

**my\_list.insert ( 2 , ”text” )** вставляет указаное значение по индексу в список, при етом двигает все остальные значения вправо, увеличиваем список по длине на **1**

**my\_list.extend(“text”)**  или **my\_list.extend( [ 1 , 2 , 3 ] )** или **my\_list.extend( [ 1 ] \* 10 )** розделяет принятую последовательность и записывает в наш список. Также можем указывать количество повторений

**my\_list.\_\_getitem\_\_( 3 )** возвращает значение из списка по переданому индексу (не изменяет)

**my\_list.clear( )** удаляет все значения со списка (изменяет)

**my\_str.join(my\_list)** обьединяет все елементы списка и записывает в строку

# **Словари**

**my\_dict = { “key1”: 1 ,”key2” : 2 }** создание словаря

**\*my\_dict** розпаковка словаря, возвращаються только ключи

**\*\*my\_dict** розпаковка словаря внутри другого словаря или передача как аргумент в функцию, возвращаються ключи и значения

**my\_dict1 = { \*\*my\_dict2 , \*\*my\_dict3 } my\_dict1 = my\_dict2 | my\_dict3**

обьединение словарей

**my\_dict1 = my\_dict2.copy( ) my\_dict1 = dict( my\_dict2 )**  создание нового словаря с парами другого

**dict ( my\_list ) my\_list = [ [“key1”, 1 ] , [ “key2”, 2 ] ]** преобразование списка в словарь

**my\_dict [ “key” ]** возвращает значение ключа

**my\_dict [ “key” ] = 777** добавление пары с словарь

**my\_dict [ “key” ] = 777** изменение значение ключа

**var = “key” my\_dict[var] = 777**  добавление пары в словарь через ключ, который хранится в переменной

**my\_dict [ “first\_key” ] [ “second\_key” ]** обращение к ключу словаря, который лежит внутри другого словаря. Также можна добавлять ключи и изменять значения

**del my\_dict [ “key” ]** удаление пары из словаря (изменяет)

**len ( my\_dict )** возвращает количество пар в словаре

↓ **Методы Словарей** ↓

**my\_dict.get( “key” )** **my\_dict.get( “key”, 0 - True - “text” )** возвращает значение ключа словаря, если ключа нету тогда возвращает **None** или значение которое укажем вторым

**my\_dict.items( )** возвращает список, который состоит из пар ключ + значение из словаря. Ето тип dict\_items

**my\_dict.keys( )** возвращает список со всеми ключами словаря. Ето тип dict.keys

**my\_dict.values( )** возвращает список со всеми значениями. Ето тип dict.values

**my\_dict.popitem( )** удаляет последнюю добавленую пару в словаре и возвращает ее (изменяет)

**my\_dict.update( { “key1”:1 , ”key2”: 2 } )** добавление к словарю указаних пар

**my\_dict.clear( )** очищает весь словарь от пар (изменяет)

# **Кортежи**

**my\_tuple = ( 1, -3 , True, ”text” )** **my\_tuple = tuple ( my\_list - my\_str - my\_dict )** создание кортежа

**my\_tuple ( var1, var2, var3 )** создание кортежа со значениями переменных

**my\_var1, my\_var2 = my\_tuple** разпоковка кортежа, значения присваиваються переменным (не изменяет)

**my\_var1, \* my\_var2 = my\_tuple** разпаковка кортежа при которой последней переменно присваиваються остальные значения

**len ( my\_tuple )** возвращает длину кортежа

**my\_tuple [ 2 ]**  **my\_tuple[ -1 ]** возвращает значения елементов кортежа по индексу. -1 - первый елемент с конца (не изменяет)

**my\_tuple [1][“key”] = 777 my\_tuple[2][3] = 777** можно изменять словари и списки, которые лежат внутри кортежа

↓ **Методы Кортежей** ↓

**my\_tuple.count(“text” - 5 - 3.2 )** возвращает количество повторений переданого значения в кортеже

**my\_tuple.index(“text”- 5 - 3.2) my\_tuple.index(“max”, 2)** возвращает индекс переданого значения первого в кортеже. Можна начинать отсчет с второго переданого индекса (включительно)

# **Наборы / Множества**

**my\_set = {1 , -3 ,5.2 , False , “text” } my\_set = set ( my\_list ) my\_set1 = my\_set2.copy( )**  создание набора

**len ( my \_set )** возвращает длину набора

**my\_set1 == my\_set2** сравнивает только значения двух множеств, а id не сравнивает . Возращает True (False)

↓ **Методы Наборов** ↓

**my\_set.add( “text” - 5 - False )** добавление елемента в набор (изменяет)

**my\_set.discard( “text” - 5 - False )** удаляет указаный елемент из набора. При ненахождении не выдает ошибку. (изменяет)

**my\_set.remove( “text” - 5 - False )** удаляет указаный елемент из набора. При ненахождении выдает ошибку. (изменяет)

**my\_set1.union( my\_set2 - my\_list - my\_str ) my\_set1 | my\_set2** возвращает обьединение множеств (не изменяет)

**my\_set1.intersection( my\_set2 - my\_list - my\_str )** **my\_set1 & my\_set2** возвращает пересеничение множеств (не изменяет)

**my\_set1.difference( my\_set2 )** возвращает разницу первого вножества от второго (не изменяет)

**my\_set1.symetric\_difference( mt\_set2 )**  **(a | b ) - (a & b )** возвращает разницу первого множеста от второго **+** наоборот (не зименяет)

**my\_set1.issubset( my\_set2 )** проверяет включено ли первое множество внутрь второго. Возвращает True ( False )

**my\_set1.issuperset( my\_set2 )** проверяет включено ли второе множество внутрь первого. Возвращает True ( False )

**my\_set.clear( )** удаляет все елементы из набора (изменяет)

# **Диапазоны**

**my\_range = range (num) = range (num1, num2 )**  **= range( num1, num2, step )** создание диапазона от num1 (включительно) до num2 (не включительно) c указаным шагом step ( 2 -> шаг через 1 число )

**my\_range [ 1 ]** возвращает значение по идексу

↓ **Методы Диапазонов** ↓

**my\_range.start my\_range.stop my\_range.step** возвращает значения начала, конца и шага диапазона

**my\_range.index( 3 - 2 - 5 - 7 )** возвращает индекс переданого елемента в диапазоне. При ненахождении выдает ошибку

**my\_range.count( 3 - 2 - 5 - 7 )** возвращает количество повторений указаного елемента. В диапазоне елементы уникальные, потому возвращает **0** или **1**

# **Циклы**

**for Елемент in Последовательность for my\_var in my\_list: print( n )** создание цикла, **n** - возвращает текущее значение в последовательности, выполняется пока не закончатся елементы в последовательности

**for my\_pair in my\_dict.items( )** возвращает пары кортежей из словаря , вместе ключи и значения

**for my\_key, my\_value in my\_dict.items( )** возвращает отдельно ключи и значения

**for my\_var in my\_set( )** порядок вывода значений из набора изменяется при использовании цикла

**for my\_var in my\_str** возвращает строку посимвольно

**while Условие: while 5 > 1 : print( )**  простой цикл while

**break** выходит из цикла и завершает его работу

**continue** заканчивает текущую итерацию и начинает следущую

**my\_var = iter ( my\_list )** создаем итератор для последовательности

**next( my\_var )** переходим к следующему елементу итерируемого обьекта

**my\_list = [ Выражение for Елемент in Последовательность if Условие ]** сокращений for in для создания последовательности на основе другой последовательности (оригинальную последовательность не изменяет)

**my\_list = [ a for a in other\_list if a > 5 ]** сокращений for in для списков

**my\_dict = { key: value \* 2 for key , value in other\_dict.items( ) }** сокращений for in для словарей

# **Ветвления и Обработчик Ошибок**

**if ( Условие ) : else :** конструкция если - иначе, результат условия автоматически конвертируется в True или False

**if ( Условие 1 ): elif( Условие 2 ):** конструкция elif позволяет добавить еще условий

**try: except:**  обработчик ошибок. Если в блоке try будет ошибка, он не выполниться -> вместо него выполниться блок except. Ловит любые типа ошибок

**try: except TypeError:** можна указывать выполнения только при одном типе ошибок ( TypeError - ZeroDivisionError )

**try: except TypeError as my\_var:**  помещаем название ошибка в переменную, которой можно пользоватся внутри блока except

**try: except TypeError: except ZeroDivisionError:** реакция обработчика на разные указанные типы ошибок

**try: except Exception as my\_var:** Exception хранит внутри себя ошибку любого типа. Возможность пользоваться ошибкой, не указывая ее типа. Нужно прописывать после указаний типа ошибок, если ети укзания есть

**try: except: else:** если полностью выполнился блок try без ошибок, тогда + еще выполняетться else

**try: except: finally:** выполняется блок finally в любом случае. Если внутри кода есть оператор return тогда он отрабатываеть после всех других операторов (после try - except - else - finally)

**raise Exception ( “text\_error” ) raise TypeError ( “text\_error” )** создаем ошибку, тип можна указивать и не указывать

**class MyException(TypeError):** пользовательские исключения

# **Функции ООП**

**isinstance( my\_str, str )** проверяет есть ли мой обьект my\_str екзепляром класа str, подходит ко всем классам (возвращает True - False).

**isinstance ( my\_object, object )** все обьекты в python являються екземпляром класа object. Работает и с класами и с обьектами классов(екземплярами)

**issubclass( MyClass , MyMainClass )** преверять есть ли первый класс дочерним класом второго. Работает только с класами

**my\_object.\_\_sizeof\_\_()** возвращает размер обьекта в байтах

↓ **Функции для работы с атрибутами Обьекта** ↓

**setattr(MyClass,”my\_name” 1)** или **setattr(my\_object,”my\_name”, 1)** добавление атрибута в класс или обьект. Если атрибут существует, то изменяет значение

**hasattr(MyClass, “my\_name”)** или **hasattr(my\_object,”my\_name”)** проверяет есть ли указаный атрибут в классе или обьекте -> Возвращает True (False)

**getattr(MyClass,”my\_name”, True)** или **getattr(my\_object, “my\_name”, False)** возвращает значение атрибута или если его нетуо, тогда третье указаное значение

**delattr(MyClass, ”my\_name”)** или **delattr(my\_object, ”my\_name”)** удаление атрибута в указаном классе или обьекте

**del MyClass.my\_name** или **del my\_object.my\_name** тоже удаление атрибута в указаном классе или обьекте

**dir( MyClass )** или **dir( my\_object )** возвращает все методы класса или обьекта

# **Мое ООП**

**class MyClass : name = “text” , age = 10** создание класcа c свойствами

**class MyClass : age = 0 def set\_value ( ) : print()** создание класса с методами внутри.

**my\_object = MyClass ( )** создание обьекта по шаблону класса

**my\_object.name = “text” my\_object.age = 20** изменение свойств класса

**class my\_class1 ( ) class my\_class2 ( my\_class1 )** создание класса наследника, конструтор только в первом класса

**def my\_method (self):** ключевое слово self в базовом классе может ссылаться и на екзепляри свого класса и на екзепляри классов наследников

**class MyClass (MyClass1 , MyClass2)** множественное наследование

**class MyClass: class MyInnerClass: print()** класс MyInnerClass вложен в класс MyClass. Такой класс только можна создаватьс внутри внешнего класса. Не рекомендуеться работать в вложеном классе с атрибутами из внешнего класса

**my\_object = type(“MyNameClass”, (”BaseClass”) , { “my\_property”:100, “my\_method” : my\_method } )** создание свого класса явно используя метакласс type( )

**class MyClass( metaclass = my\_meta\_fn ):** создание класса из мета функции

**class MyClass( metaclass = MyMetaClass ):** создание класса из пользовательского мета класса

↓ **Колекции \_\_slots\_\_** ↓

**\_\_slots\_\_= ( “x”, ”y” )** создаеться колекция. Ограничивает количество и название локальных свойств только для екзепляра обьекта. Не воздействует на классы. Колекция создаеться в самом начале класса. Нельзя обращаться к колекции my\_object\_\_dict\_\_ потому што он не создаеться. Не работает для классов наследников, только для екзепляра класса в котором был указан.

**\_\_slots\_\_ = ‘z’ ,**  нужно прописывать пустую колекцию или с добаленией свойст в классе наследнике, чтоб работала колекция из родительского класса для текущего

**attribute** режим доступа типа public

**\_attribute** режим доступа типа protected

**\_\_attribute** режим доступа типа private

# **Декораторы ООП**

**@classmethond def my\_fn (cls)** **:**  класс метода. Можно работать только с атрибутами класса, с атрибутами обьекта нельзя работать. Позволяет обращается к свойствам и методам только через сам класс, но не через обьекты(екзепляры)

**@staticmethod( )** не может работать с свойствами и метода ни класса, ни обьекта. Позволяет обращается к свойствам и методам только через сам класс, но не через обьекты(екзепляры)

**my\_name = property (get\_value , set\_value)**  или

**my\_name = property( ) my\_name = my\_name.getter( my\_method1 ) my\_name = my\_name.setter( my\_method2 )** или -->

**@propery def my\_name (self): + @my\_name.setter def my\_name (self, a )** **+** **@my\_name.deleter def my\_name(self)** -->

**my\_object.my\_name** или **my\_object.my\_name = 5** или **del my\_object.my\_name**

Позволяет создавать динамические атрибуты при вызове который будет вызываться наша указаня функция. Можно возвращать и присваивать значения атрибутам без явного указания гетеров и сетером. Геттер всегда указывают первый. Имеет приоритет више чем у обичного свойства обьекта. Упрощает работу в гетерами и сетерами, не нужно запоминать названия.

# **Магические Свойства**

**MyClass.\_\_name\_\_** возвращает название класса

**MyClass.\_\_mro\_\_** возвращает все классы от которых унаследован текущий, ровно в том порядке в котором будет искаться методы, который вызвал пользователь

**my\_object.\_\_class\_\_** возвращае имя класса по которому был создан обьект

**MyClass.\_\_doc\_\_** или **my\_object.\_\_doc\_\_** возвращает описание класса или обьекта

**MyClass.\_\_dict\_\_** или **my\_object.\_\_dict\_\_** возвращает все атрибути класса или обьекта

**\_\_name\_\_** показывает в каком файле запущена наша програма ( \_\_main\_\_ - код запущен на прямую по умолчанию )

# **Магические Методы**

↓ **Инициализаторы , Финализаторы и Метод \_\_new\_\_** ↓

**type ( ) -> MyClass.\_\_new\_\_(cls) : return super( ).\_\_new\_\_(cls) ) -> MyClass.\_\_init\_\_(self)**

**self** ключевое слово, которое содержить ссылку на текущий обьект класса

**cls** ключевое слово, которое содержит ссылку на текущий класс

**super( ) .** содержит ссылку на родительский класс из текущего класса

**super().\_\_new\_\_(cls)** создание нашего класса , как екзепляр базового класса type()

**def \_\_new\_\_ (cls, \*args , \*\*kwargs) : return super().\_\_new\_\_(cls)** метод, который создаеть текущий класс, как екзепляр базового класса type(). Также выделяет место под етот класс в памяти

**def \_\_init\_\_ (self) :** метод вызываеться при иницыализации екзепляра класса после метода \_\_new\_\_. Возвращает сам новосозданный обьект

**def \_\_del\_\_(self) :** создание финализатора, вызываеться перед удалением текущего обьекта, когда удаляется ссылка на етот обьект

↓ **Методы, которые вызываються при работе с свойствами класса** ↓

**def \_\_getattribute\_\_(self,item): return object.\_\_getattribute(self,item)** метод вызываеться, когда обращаемся к существующему атрибуту обьекта

**def \_\_getattr\_\_(self,item): return False** методы вызывается, когда обращаемся к несуществующему атрибуту обьекта

**def \_\_setattr\_\_(self,key,value):** **object.\_\_setattr\_\_(self,key,value)** или **self.\_\_dict\_\_[key] = value** метод вызываеться, когда присваиваем значения атрибутам обьекта

**def \_\_delattr\_\_(self,item): object.\_\_delattr(self,item)** метод вызывается, когда удаляем атрибут класса

↓ **Дескрипторы** ↓

**class MyClass:** создание дескриптора, класс который вписываеться внутрь другого класса

**def \_\_set\_name\_\_( self, owner ,name ):** изменяеть названия свойств для другого класса

**self.name = “\_” + name**

**def \_\_get\_\_( self , instance , owner ):** получение данных из екзепляра второго класса

**return instance.\_\_dict\_\_[ self.name ]** или **return getattr ( instance , self.name )**

**def \_\_set\_\_ ( self , instance , value ):** запись данных в екзепляр другого класса

**instance.\_\_dict\_\_[ self.name ] = value** или **setattr ( instance , self.name , value )**

**def \_\_delete\_\_(self , instance ):** удаление свойства из второго класса

**del self.name**

**class OtherClass:** создание другого класса

**x = MyClass( ) y = MyClass( )**

**def \_\_call\_\_(self, \*args, \*\*kwargs)** методы вызываеться при обращении к классу или обьекту подобно функции ( )

**def \_\_str\_\_( self ) : return my\_str** методы вызываеться когда пытаемся получить значения свойства обьекта через print или str. Вывод информации для пользователя

**def \_\_repr\_\_( self ) : return my\_str** выводит значения свойста обьекта в режиме отладки. Обращение в интерпритаторе по названию обьекта. Вывод информации для разработчика.

**def \_\_len\_\_( self ) : return len( my\_value )** метод вызываеться когда вызываем функцию len

**def \_\_bool\_\_(self) :** **return bool( self.x )** метод вызываеться, когда вызываем функцию bool. Если bool не определен в классе, тогда вызываеться метод len. Можно использовать как выражение в if - elif

**def \_\_abs\_\_( self ) : return abs( my\_value )** методы вызываеться когда вызываем функцию abs

↓ **Математические Магические Методы** ↓

**def \_\_add\_\_ (self, my\_value):** -> **my\_object + 10**  метод вызываеться при добавлении к обьекту

**def \_\_radd\_\_(self, my\_value):** -> **10 + my\_object**  паралельно добавление к значению обьекта

**def \_\_iadd\_\_(self,my\_value):** -> **my\_object += 10** метод вызываеться при давблении обьекта укороченой записью

**def \_\_sub\_\_(self,other): -> my\_object - 10**  метод вызываеться при вычетании обьекта

**def \_\_mul\_\_ (self,other):** -> **my\_object \* 10** метод вызываеться при умножении обьекта

**def \_\_truediv\_\_(self,other):** -> **my\_object / 10** метод вызываеться при делении обьекта

**def \_\_floordiv\_\_ (self,other):** -> **my\_object // 10** методы вызываеться при целочисленом делении обьекта

**def \_\_mod\_\_(self,other):** -> **my\_object % 10** метод вызываеться при делении обьекта по модулю

↓ **Магические Методы Сравнения** ↓

**def \_\_eq\_\_(self,other):**  -> **my\_object1 == my\_object2** вызов оператора сравнения ==

**def \_\_ne\_\_(self,other):** -> **my\_object1 != my\_object2** вызов оператора не равенства !=

**def \_\_lt\_\_(self,other):** -> **my\_object1 < my\_object2** вызов оператора меньше <

**def \_\_le\_\_(self,other):** -> **my\_object1 <= my\_object2** вызов поератора меньше или равно <=

**def \_\_gt\_\_(self,other):** -> **my\_object1 > my\_object2** вызов оператора больше >

**def \_\_ge\_\_(self,other): -> my\_object >= my\_object2** вызов оператора больше или ранов >=

↓ **Методы для работы с Списками через [ ]** ↓

**def \_\_getitem\_\_(self, item):** метод вызываеться при обращении к последовательности через [ ]

**def \_\_setitem\_\_(self, key ,value ):** методы вызываеться при присвоении значения последовательности my\_list [ key ] = value

**def \_\_delitem\_\_(self,key):** метод вызываеться при удалении елемента из списка по ключу del my\_list[ key ]

↓ **Методы для Работы с Циклами** ↓

**def \_\_iter\_\_(self):** метод вызываеться, когда используем цикл for my\_var in my\_list. Само создание цикла

**def \_\_next\_\_(self):** метод вызываеться когда вызываем функцию next( ) - получаем следующий елемент в итерации

↓ **Методы для Работы с Менеджером Контекста** ↓

**def \_\_enter\_\_( self ) :** методы вызываеться в момент созздания менеджера контекста with

**def \_\_exit\_\_( self , exc\_type , exc\_val , ex\_tb ):** метод вызываеться в момент завершения работы менеджера контекста with. Внутрь exc\_type передаеться None если исключений не возникло

↓ **Точка входа в Программу** ↓

**if \_\_name\_\_ = “\_\_main\_\_”:** условие истенное, если текущий файл исполняемый. Служит чтоб не запускать ненужные функции из подключених модулей к исполняемому файлу. Также служит чтоб не перепутать модулю с исполняемым файлов

**def main( ) : my\_fn( ) if \_\_name\_\_ = “\_\_main\_\_” : main( )** функция my\_fn только если текущий файл будет исполняемый. Функцией main можна указывать точку входа в програму (обезательно прописывать в главном файле)

Если **\_\_name\_\_**  возвращает **\_\_main\_\_** тогда ето выполняемый файл

Если **\_\_name\_\_** возвращает название текущего файла (test.py), тогда ето будет модуль который подключаеться к выполняемому файлу

↓ **Хеширование** ↓

**hash ( my\_object )** производит хеширование обьекта, возвращает набор символов фиксированой длины

**def \_\_eq\_\_( self, other ) : return self.x == other.x and self.y == other.y** функция hash опеределяет равенство обьектов про def \_\_eq\_\_

**def \_\_hash\_\_( self ) : return hash ( (self.x, self.y) )** методы вызываеться, когда вызываем функцию hash. Также нужно переопределять оператора равенства == def \_\_eq\_\_(self,other)

# **Другое**

# добавление коментариев