

Содержание

Урок 1.....	6
Установка и запуск.....	6
Типы данных.....	7
Числа.....	7
Действия над числами.....	7
Сравнение чисел.....	8
Преобразования.....	8
Строки.....	9
Создание строк.....	9
Операторы строк.....	9
Сложение.....	9
Умножение.....	9
Оператор in - проверка вхождения подстроки в заданную строку.....	9
Индексация строк.....	9
Нарезка строк.....	9
Встраивания переменных внутри строки (F-функции).....	9
Изменение строк.....	10
Методы строк.....	10
2 урок.....	12
Списки.....	12
1. Объявить список.....	12
2. Обратиться к элементу.....	12
3. Добавить элемент.....	12
4. Изменить элемент.....	12
5. Удалить элемент.....	12
6. Проверить на вхождение.....	13
7. Объединить списки (метод extend).....	13
8. Копирование строк.....	13
9. Итерация по списку.....	13
10. Итерацию можно выполнить с помощью оператора цикла while.....	14
Методы списков.....	16
1. Index — позволяет узнать индекс элемента, возвращает первое совпадение.....	16
2. count (каунт) (считать) — получаем количество одинаковых элементов списка.....	17
3. clear () — очищает список.....	17
4. sort — сортирует список по указанному ключу.....	17
6. pop - позволяет получить элемент и удаляет его по номеру позиции искомого элемента в списке (целое число).....	18
Встроенные функции.....	19
7. len — получение длины списка (от слова length — длина), то есть количество элементов в списке.....	19

8. max - возвращает элемент с максимальным значением.....	19
9. min — возвращает элемент с минимальным значением.....	19
10. Приведение к типу.....	20
Урок 3 часть 1.....	21
Кортежи.....	21
Срез кортежа.....	21
Операция присваивания кортежа.....	21
Кортеж из списка.....	22
Кортеж из другого кортежа.....	22
Функция получения значения элемента по индексу.....	22
Кортеж, содержащий список.....	23
Взятие среза в кортеже.....	23
Конкатенация (сложение).....	24
Повторение *.....	25
Обход кортежа в цикле.....	25
Цикл for.....	25
Цикл while.....	25
Оператор in - проверка вхождения элемента в кортеж (есть ли заданный элемент в кортеже?).....	26
Методы работы с кортежами.....	28
Метод index(). Поиск позиции элемента в кортеже.....	28
Метод count - подсчет количества вхождений элемента в кортеж.....	29
Генератор списков.....	29
Урок 3 часть 2.....	30
Множества.....	30
Генератор множеств.....	32
Что можно делать со множествами?.....	34
(размер множества 14).....	34
setB in setA - принадлежит ли setA множеству setB.....	34
То есть все, кто записался в секцию самбо, ушли из этой секции , например, тренер заболел, и дружно записались в лыжную секцию.....	34
setA.isdisjoint(setB) - непересекающийся.....	35
setB == setA — два равенства означают, что все элементы setA принадлежат setB, все элементы setB принадлежат setA.....	35
setA.issubset(setB) или setA <= setB - все элементы setA принадлежат setB...36	36
setA.issuperset(setB).....	36
или setA >= setB — аналогично для множества setB.....	36
set.union(setB, ...) или setA setB ... - объединение нескольких множеств...37	37
setA.intersection(setB, ...) или setA & setB & ... - пересечение множеств.....38	38
setA.difference(setB, ...) или setA - setB - ... - множество из всех элементов setA, не принадлежащие ни одному из setB.....	42
setA.symmetric_difference(setB); setA ^ setB - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.....	44
setA.copy() - копия множества.....	44

Операции, которые непосредственно изменяют множество.....	45
setA.update(setB, ...); setA = setB ... - объединение.....	45
setA.intersection_update(setB, ...); setA &= setB & ... - пересечение.....	45
setA.intersection_update(setB).....	45
setA.difference_update(setB, ...); setA -= setB ... - вычитание.....	46
setA.symmetric_difference_update(setB) - симметричная разница.....	46
Методы множеств работающие с одиночными элементами данного множества.....	47
setA.add(elem) - добавляет элемент в множество.....	47
setA.remove(elem) - удаляет элемент из множества.....	47
setA.discard(elem) - удаляет элемент, если он находится в множестве.....	47
setA.clear() - очистка множества.....	48
frozenset.....	48
Урок 3 часть 3.....	49
Словари.....	49
Создание словарей.....	51
Через функцию.....	51
Через фигурные скобки.....	51
Изменение словаря.....	51
Перебор элементов словарей.....	51
Методы словарей (функции словаря как объекта).....	52
copy() - создание копии словаря.....	52
get — получение значения по ключу.....	52
clear — очистка словаря.....	52
keys() - получение всех ключей.....	53
values() - метод получения не ключей словаря, а всех их значений.....	53
items() - получение всех элементов словаря.....	53
pop() - удаляет и возвращает значение ключа.....	53
update() - обновить значение по ключу и добавляет новые ключи.....	54
Урок 4 часть 1.....	62
Функции и их аргументы.....	62
Именные функции, инструкция def.....	62
В функциях распространены следующие варианты аргументов.....	65
1. Функции с позиционными аргументами.....	65
2. Функции с именованными аргументами.....	66
3. Функции с произвольным числом аргументов.....	67
Урок 4 часть 2.....	69
Порядок вызовов смешанных аргументов.....	69
Передача именованных аргументов с помощью словаря.....	70
Анонимные функции, инструкция lambda.....	70
Файлы. Работа с файлами.....	71
Чтение из файла.....	72
Запись в файл.....	72
Урок 5.....	75

Исключения в Python.....	75
Конструкция try - except для обработки исключений.....	75
Урок 6.....	83
Регулярные выражения в Python.....	83
re.match() - поиск вхождения шаблона в начало строки.....	87
re.search().....	87
re.findall().....	88
re.compile().....	88
re.split().....	89
re.sub().....	89
re.subn().....	90
re.escape().....	90
Урок 7.....	91
Объектно-ориентированное программирование.....	91
Создание класса и экземпляра класса (или, по-другому, объекта).....	95
Методы __int__() и __del__().....	99
Переопределение методов.....	100
Экземпляры как атрибуты.....	105
Импортирование классов.....	106
Импортирование одного класса.....	106
Хранение нескольких классов в модуле.....	107
Импортирование нескольких классов из модуля.....	109
Импортирование всего модуля.....	109
Импортирование всех классов из модуля.....	110
Импортирование модуля в модуль.....	110
Использование псевдонимов.....	112
Оформление классов.....	112
Множественное наследование.....	112
Поиск идентификаторов при множественном наследовании.....	115
Миксины (примеси).....	116
MRO – Method Resolution Order (Порядок разрешения метода).....	118
Магические методы.....	118
Поддержка перегрузки операторов в пользовательских классах.....	121
Арифметические операторы.....	121
Подробнее об арифметических операторах.....	124
Методы операторов сравнения.....	125
Операторы членства.....	126
Побитовые операторы.....	128
Расширенные задания.....	129
Самоанализ объектов.....	130
Специальные методы для самоанализа.....	130
Управление доступом к атрибутам.....	131
Извлечение атрибутов.....	131
Настройка атрибутов.....	133

Удаление атрибутов.....	134
Управление атрибутами с помощью дескрипторов.....	135
Поддержка итерации с помощью итераторов и итерируемых объектов....	137
Создание итераторов.....	137
Создание итерируемых объектов.....	139
Реализация пользовательских последовательностей и сопоставлений.....	141
Настройка и демонтаж с помощью контекстных менеджеров.....	142
Декораторы функций.....	144
Использование классов в качестве декораторов.....	145
Создание синглтонов.....	148
Добавление информации о единицах измерения.....	155

Урок 1

Установка и запуск

Для linux обычно Python уже установлен, для Windows Python можно скачать по ссылке <https://www.python.org/downloads/windows/> . Загрузится архивный zip файл python-3.11.4-embed-amd64.zip для 64 разрядного компьютера. Его надо распаковать и запустить файл python.exe.

Проверить для linux наличие Python можно в терминале:

```
$ python3 --version
```

Если Python установлен, то он отобразит версию интерпретатора.

Однако, даже если Python установлен, его версия может быть не самой последней. Для установки последней доступной версии Python выполним следующую команду:

```
$ sudo apt-get update && sudo apt-get install python3
```

Загрузка интерпретатора python.

Для Windows — `python`

Для linux — `python3`

После установки необходимо перейти в терминал.

```
$ python3
```

```
5 + 4
```

Первая программа на python (pr01.py)

Набираем в любом текстовом редакторе типа блокнот, лучше gedit (в linux он может быть уже установлен. Для Windows ссылка на скачивание и установку <https://gedit.ru.uptodown.com/windows/download>.

И так, код первой программы на python (pr01.py):

```
# -*- coding : utf-8 -*-  
print("Привет, мир!")
```

Запуск программы pr01.py

```
$ python3 pr01.py
```

```
# Выдаст: Привет, мир!
```

Можно запустить, назвав только название имени файла программы.

Для этого в начале программы указывается путь интерпретатора python

```
#!/usr/bin/python3
```

```
#!/usr/local/bin/python3
```

```
#!/usr/bin/env python3 #- программа env производит поиск интерпретатора Python.
```

В Python всё является объектами. Например мы осуществляем присвоение.

```
x = "Привет, мир!"
```

```
y = 10
```

Строка "Привет, мир!" и число 10 являются объектами, то есть имеют свой адрес в оперативной памяти компьютера.

x, y — всего лишь ссылки на разные объекты.

Присвоим ссылке y значение x.

```
y = x
```

```
print(y)
```

```
# Будет выведена строка "Привет, мир!"
```

x, y — теперь ссылаются на один объект.

Другой объект (10) уничтожается, так как на него уже нет ссылок.

Типы данных

Числа

Числа (целые, вещественные, комплексные)

Целые — 10, 23, 267 (int)

Вещественные — 3.23, 67.1, 123.456 (float)

Комплексные — (35 j10)

Действия над числами

x + y - Сложение

x - y - Вычитание

x * y - Умножение

x / y - Деление

x // y - Получение целой части от деления

x % y - Остаток от деления

- x - Смена знака числа

abs(x) - Модуль числа

x * y - Умножение

x ** y - Возведение в степень

round(x, [n]) - Округление числа (до n знака)

В квадратных скобках указывается количество знаков дробной части.

Квадратные скобки указывают на необязательность этого аргумента. Его отсутствие позволит округлить переменную x только до целой части. Дробная часть отбрасывается.

Пример отбрасывания дробной части:

```
x = 1.23
```

```
round(x)
```

```
# Выдаст 1
```

Но, если дробная часть будет больше или равно 0.5, то при отбрасывании дробной части, целая часть увеличивается на 1.

```
x = 1.5
```

```
round(x)
```

```
# Выдаст 2
```

Когда мы указываем второй аргумент n происходит округление до n знаков в дробной части.

Здесь работает тоже правило, что, и для целой части в предыдущем примере. Отбрасываемая часть начинается с цифры 5 и более - к сохраняемой дробной части прибавляется 1.

Пример:

```
x = 1.235
```

```
round(x, 2)
```

```
# Выдаст 1.24
```

```
x = 1.2349
```

```
round(x, 2)
```

```
# Выдаст 1.23
```

type(x) - Узнать тип числа

Пример:

```
x = 1.49
```

```
number = round(x)
```

```
print(number)
```

```
# Выдаст 1
```

```
type(number)
```

```
<class 'int'># Тип числа 'int' - целое.
```

```
x = 1.49
```

```
n = 1
```

```
number = round(x, n)
```

```
print(number)
```

```
# Выдаст 1.5
```

```
type(number)
```

```
<class 'float'> # Тип числа 'float' - вещественное.
```

Сравнение чисел

x == y - равно

x != y - не равно

x > y - больше

x < y - меньше

x >= y — больше или равно

x <= y — меньше и равно

Преобразования

int(x) — преобразует в целое число

float(x) — преобразует в вещественное число

bin(x) — преобразует в двоичное число

oct(x) - преобразует в восьмеричное число

hex(x) - преобразует в шестнадцатеричное число
str(x) — преобразует число в строку
import random — датчик случайных чисел
print(random.random())

Строки

Создание строк

```
name = 'Алиса'  
hobby = 'плавание'
```

Операторы строк

Сложение

```
a = name + ' любит ' + hobby  
print(a)  
# Выдаст: 'Алиса любит плавание'
```

Умножение

```
b = name*3  
print(b)  
# Выдаст: 'АлисаАлисаАлиса'
```

Оператор in - проверка вхождения подстроки в заданную строку

В следующем примере надо найти подстроку b в строке a.

```
a = 'Алиса любит плавание'  
b = 'люб'  
c = b in a  
print(c)  
# Выведет True (Правда), то есть значение подстроки b находится в строке a.
```

Индексация строк

```
'Hello'  
Индексы этой строки - 0 1 2 3 4  
c = 'Hello'  
print(c[0])  
# Выведет 'H'  
print(c[5]) — Ошибка IndexError. Нет такого индекса в этой строке.
```

Нарезка строк

```
word = 'Привет'  
firstThree = word[0:3]  
print(firstThree)  
# Выведет 'При'
```

Первая буква (символ) здесь 'П', что соответствует индексу 0, указанному в начале среза. Конец среза, указанный индексом (3), не будет включаться в получаемый срез. А значит будет выведена последняя буква среза с индексом 2.

Встраивания переменных внутри строки (F-функции)

```
name = 'Алиса'  
greeting = f'Привет, {name}! Как дела?'  
print(greeting)
```

Выведет Привет, Алиса! Как дела?

Перед строкой ставится функция f. В строку могут вставляться переменные, обрамлённые фигурными скобками {}. При распечатке содержимое фигурных скобок и сами скобки заменяются значением указанной переменной.

Изменение строк

Строки неизменны. Если нужно изменить строку, то создаётся новая строка на основе старой с изменениями.

Методы строк

1. `capitalize()` - преобразует первый символ строки в верхний регистр.

```
srt1 = 'москва'
print(srt1.capitalize())
# Выведет: 'Москва'
```

2. `count()` - возвращает количество вхождений символа или подстроки в строке.

```
srt1 = 'Доброе утро.'
print(srt1.count('o'))
# Выведет: 3
```

3. `find()` - ищет первую позицию символа или подстроки в строке. Возвращает `index`.

```
srt1 = 'Доброе утро.'
print(srt1.find('o'))
# Выведет: 1 (Напомним, что индекс в строке начинается с 0)
```

4. `index()` - ищет символ или подстроку. Возвращает `index`, по которому он был первым обнаружен (похож на предыдущий метод)

```
srt1 = 'Доброе утро.'
print(srt1.index('o'))
# Выведет: 1
```

5. `join()` - соединяет объекты в одну строку.

Пусть дан список:

```
a = ['Раз', 'два', 'три', 'четыре', 'пять']
# Из него получаем строку b
b = ' | '.join(a)
print(b) # Выведет: Раз | два | три | четыре | пять
```

В качестве разделителя берётся символ `|`, обрамлённый двумя пробелами.

6. `lower()` - перевод строки к нижнему регистру

```
srt1 = 'МОСКВА'
print(srt1.lower())
# Выведет: 'москва'
```

7. `replace()` - возвращает строку с заменой подстроки

```
str1 = 'Сегодня 21 мая 2023 года'
str2 = str1.replace('Сегодня', 'Завтра')
print(str1)
# Выведет: Сегодня 21 мая 2023 года
print(str2)
```

```
# Выведет: Завтра 21 мая 2023 года
```

8. `split()` - разбивает строку по указанному символу или подстроке

```
str1 = 'Сегодня 21 мая 2023 года'
```

```
list1 = str1.split(' ')
```

```
print(list1)
```

```
# Выведет список: ['Сегодня', '21', 'мая', '2023', 'года']
```

9. `strip()` - возвращает строку с удалёнными пробелами с обеих сторон.

```
str1 = '  Сегодня 21 мая 2023 года  '
```

```
str2 = str1.strip()
```

```
str1
```

```
# Выведет: '  Сегодня 21 мая 2023 года  '
```

```
str2
```

```
# Выведет: 'Сегодня 21 мая 2023 года'
```

Было 3 пробела в начале строки `str1` и 2 пробела в конце этой строки.

После отработки метода `strip()` получаем строку `str2` уже без ведущих и конечных пробелов.

10. `upper()` - переводит символы в верхний регистр

```
str1 = 'Сегодня'
```

```
str2 = str1.upper()
```

```
print(str2)
```

```
# Выведет: СЕГОДНЯ
```

Здесь и далее символ `#` - это символ комментария на одну строчку программы.

Python комментарии не задеиствует. Они служат для пояснения действий

программы. Существуют также комментарии на несколько строк. Это `''' '''` -

тройные одинарные кавычки и `""" """` - тройные двойные кавычки.

Кроме того, тройные кавычки как одинарные, так и двойные, могут использоваться не только для много строковых комментариев, но и обрамления символьного текста при присвоении какой-либо переменной. Такое разнообразие кавычек позволяют не проводить так называемое экранирование символов, например:

```
""" Это много строковый комментарий для демонстрации программного кода с
использованием тройных двойных кавычек.
```

```
"""
```

```
mystring = '''В романе Александра Дюма "Три мушкетёра" главное действующее
лицо — д'Артаньян.'''
```

```
print(mystring)
```

```
# Выведет: В романе Александра Дюма "Три мушкетёра" главное действующее
лицо — д'Артаньян.
```

Сначала идёт много строковый комментарий в тройных двойных кавычках. Он игнорируется при исполнении кода. Выполняется только присвоение переменной `mystring` строки, обрамлённой в тройные одинарные кавычки.

Двойные и одинарные кавычки при распечатке сохраняются.

2 урок

Списки

Списки наиболее часто используемый вид сложных переменных. В его последовательности через запятую может включаться любые другие виды простых и сложных переменных.

Например:

```
mylist = ['Привет мир!', 333, {'car': 'машина', 'apple': 'яблоко'}, [1, 2, 7], [78.66, 37, 88.26, 55]]
```

Здесь в списке перечислены: 'Привет мир!' - строковая переменная, 333 - целое число, {'car': 'машина', 'apple': 'яблоко'} - словарь, [1, 2, 7] и [78.66, 37, 88.26, 55] - списки, вложенные в основной список mylist.

1. Объявить список

```
mylist = [1, 2, 3, 4] # непосредственно задают значения
mylist = list()      # функцией list(), в данном случае пустой список, тоже, что и
mylist1 = []         # пустой список (нет ни одного элемента)
```

2. Обратиться к элементу

```
mylist = [1, 2, 3, 4, 5]
print(mylist[4])
# Выведет: 5 - индекс элементов начинается с 0. (0,1,2,3,4)
```

3. Добавить элемент

```
mylist = [1, 2, 3, 4, 5]
mylist.append(6)
# Выведет: [1, 2, 3, 4, 5, 6]
либо
mylist = [1, 2, 3, 4, 5]
mylist.insert(1, 25) # вставить на позицию 1 значение 25
print(mylist)
# Выведет: [1, 25, 2, 3, 4, 5, 6]
```

4. Изменить элемент

```
mylist = [1, 2, 3, 4, 5]
mylist[1] = None
print(mylist)
# Выведет: [1, None, 3, 4, 5]
```

5. Удалить элемент

```
mylist = [1, 2, 3, 4, 5, 4, 3, 2, 1]
del mylist[2]
print(mylist)
# Выведет: [1, 2, 4, 5, 4, 3, 2, 1]
либо remove()
```

```
mylist = [1, 2, 3, 4, 5, 4, 3, 2, 1]
mylist.remove(2)
print(mylist)
# Выведет: [1, 3, 4, 5, 4, 3, 2, 1]
```

6. Проверить на вхождение

```
mylist = [1, 2, 3, 4, 5]
print(None in mylist)
# Выведет: False (Ложь)
print(3 in mylist)
# Выведет: True (Правда)
```

7. Объединить списки (метод extend)

методы это такие функции, которые привязаны к объектам и
пишутся через точку (точечная нотация).

```
mylist = [1, 2]
mylist1 = [3, 4]
mylist2 = [5]
mylist.extend(mylist1)
print(mylist)
# Выведет: [1, 2, 3, 4]
mylist.extend(mylist2)
print(mylist)
# Выведет: [1, 2, 3, 4, 5]
```

Или оператор сложения с присваиванием

```
mylist = [1, 2]
mylist1 = [3, 4]
mylist2 = [5]
mylist += mylist1 # [1, 2, 3, 4]
print(mylist)
# Выведет: [1, 2, 3, 4]
mylist += mylist2 # [1, 2, 3, 4, 5]
print(mylist)
# Выведет: [1, 2, 3, 4, 5]
# '+' - это тоже самое, что и mylist = mylist + mylist2
```

8. Копирование строк

```
mylist = [1, 2, 3, 4, 5]
mylist_copy = mylist.copy()
print(mylist)
# Выведет: [1, 2, 3, 4, 5]
print(mylist_copy)
# Выведет: [1, 2, 3, 4, 5]
```

9. Итерация по списку

Здесь появляется оператор цикла for.

```
for item in mylist:
```

Он последовательно извлекает очередное значение из списка `mylist`, присваивает переменной `item`, а затем применяет к этой переменной какие-либо действия (в данном случае распечатывает эту переменную — `print(item)`)

```
mylist = [1, 2, 3, 4, 5]
for item in mylist:
    print(item)
```

Выведет:

```
1
2
3
4
5
```

Такого же результата можно добиться с помощью функции `range`, которая генерирует последовательность чисел.

итерация по числам с нуля до 5 не включительно

```
for i in range(1, 5):
    print(i + i)
```

Выведет:

```
2
4
6
8
```

Отступы.

После символа `:` начинается блок операторов в данном случае цикла `for`, который должен иметь отступ относительно строки с `:` и по величине должен иметь сдвиг на 4 пробела или на горизонтальную табуляцию.

10. Итерацию можно выполнить с помощью оператора цикла `while`

Пояснения к циклу `while`.

Слово `while` означает пока, то есть, пока выполняется условие, цикл будет выполняться. В тот момент когда очередное значение не будет удовлетворять условию - цикл обрывается.

Предположим, у нас имеется список `list = [25, 14.3, 82, 4]`. Нужно в цикле `while` распечатать все значения в этом списке.

```
list = [25, 14.3, 82, 9]
```

```
i = 0
```

```
while i < 4:
    print(list[i])
    i = i + 1
```

Выведет:

```
25
14.3
82
9
```

Как это получилось?

Буква *i* будет у нас текущим индексом, то есть текущим номером позиции в списке *list*. Номер позиции начинается с 0, и далее 1,2,3. Индексу 3 соответствует цифра 9, а 0 - 25.

В начале мы присваиваем индексу *i* значение 0. Этому индексу соответствует число из списка 25.

В цикле, пока индекс *i* меньше (<) 4, у нас перебираются все элементы списка *list*. В начале это значение 25, так как индекс *i* имеет значение 0 (присвоен до цикла). Мы значение 25 распечатываем, и присваиваем индексу *i* следующее значение $i = i + 1$. Это означает прибавку 1, то есть получаем $0+1 = 1$. Цикл повторяется но уже с другим индексом и распечатывается значение 14.3. Далее индексу *i*, который имел текущее значение 1, прибавляется еще 1, и получается $1 + 1 = 2$. Так как *i* у нас еще меньше 4, цикл повторяется, но, уже с текущим индексом *i* равным 2.

Цикл повторяется до тех пор, пока значение *i* не станет равно 4. В этом случае условие цикла не выполняется и цикл прекращается.

Рассмотрим бесконечный цикл `while`

```
mylist = [1, 2, 3, 4, 5]
i = 0
while True:
    try:
        print(mylist[i])
    except IndexError:
        print('Done!')
        break
    i += 1
```

Выведет:

```
1
2
3
4
5
```

Done!

В этом примере образован бесконечный цикл с помощью булевского (логического) значения `True` (Тру), которое обозначает Правда. Есть еще одно булевское значение `False` (Фолс) Ложь.

В этой программе `while True:` означает Пока Правда: делаем что-то. А когда не Правда (то есть Ложь) — выходим из цикла. Проверка на Правду и Ложь осуществляется с помощью так называемых исключений `try:` (пытаться). При этом отработывает функция `print(mylist[i])`, распечатывающая значение из списка с текущим индексом. А `except IndexError:` (кроме) `IndexError` — ошибка индекса. Ошибка возникает когда индекс выходит за пределы списка. В данном случае это индекс 5. А у нас индексы списка 0,1,2,3,4. Программа перестаёт работать и распечатывает 'Done!' ('Сделано!'). И выход из цикла командой `break` (перерыв). Нужно обратить внимание на запись `i += 1`. Это укороченная запись `i = i + 1`.

Методы списков

Напомним, что любая переменная в языке Python, является объектом. А к объектам применимы функции, называемые методами. Записывается сначала объект, а затем через точку (точечная нотация) название метода, с переданными аргументами в скобках.

1. Index — позволяет узнать индекс элемента, возвращает первое совпадение

```
mylist = [1, 2, 3, 4, 5, 4, 3, 2, 1]
```

```
print(mylist.index(3))
```

```
# Выведет: 2
```

Хотя в списке mylist два значения соответствующие 3, но индекс находится только для значения первой тройки. Первая тройка находится под индексом 2 (0, 1, 2).

Если в списке такого значения нет - например, `print(mylist.index(37))` - который указывается в скобках метода, то возникает ошибка `ValueError: 37 is not in list` (Ошибка значения: 37 нет в списке). Error (Эрор) - ошибка, Value (Вэльё) - значение, переменная.

Чтобы избежать в программе возникновения подобных ошибок, можно примерить оператор условия `if` (если).

```
Mylist = [1, 2, 3, 4, 5, 4, 3, 2, 1]
```

```
value = 38
```

```
if value in Mylist:
```

```
    print(Mylist.index(value))
```

```
else:
```

```
    print(f'Значения {value} нет в списке mylist')
```

```
# Выведет: Значения 37 нет в списке mylist
```

В операторе `if` записывается условие, которое будет выполнено, если условие — Правда (True)

В противном случае, то есть условие не выполняется, то есть Ложно (False), и оно выполняется другим оператором условия `else` (иначе).

Здесь в виде условия записано `if value in Mylist`: означает - Если значение находится в (in) списке Mylist, то выполняется распечатка данного значения `print(Mylist.index(value))`, иначе (else:) распечатывается фраза `print(f'Значения {value} нет в списке mylist')`.

В таком случае, с использованием оператора `if`, не будет останавливаться программа из-за ошибки.

Очень полезная функция `f`. Она предназначена для распечатки форматированных строк.

Например:

```
name = 'Максим'
```

```
old = 12
```

```
print(f'Меня зовут {name}, мне {old} лет')
```

```
# Выведет: Меня зовут Максим', мне 12 лет
```

2. count (каунт) (считать) — получаем количество одинаковых элементов списка

```
mylist = [1, 2, 3, 4, 5, 4, 3, 2, 1]
count_for_one = mylist.count(1)
print(count_for_one)
# Выведет: 2
count_for_five = mylist.count(5)
print(count_for_five)
# Выведет: 1
```

Если в первом случае количество единиц в списке mylist равно 2, то во втором случае количество пятёрок равно 1.

3. clear () — очищает список

```
mylist = [1, 2, 3, 4, 5]
mylist.clear()
print(mylist)
# Выведет: [] - пустой список
```

4. sort — сортирует список по указанному ключу

Принимает 2 необязательных аргумента: key(функция) и булевый флаг — revers (реверс): сортирует объекты в обратном порядке. По умолчанию, в качестве ключа используют значение переменной.

```
my_list = [7, 5, 8, 2, 11, 1, 14]
my_list.sort()
print(my_list)
#Выведет: [1, 2, 5, 7, 8, 11, 14]
```

Сортировка в порядке убывания:

```
my_list = [7, 5, 8, 2, 11, 1, 14]
my_list.sort(reverse=True)
print(my_list)
# Выведет: # [14, 11, 8, 7, 5, 2, 1]
```

key(функция) может выглядеть так:

```
my_list = [7, 5, 8, 2, 11, 1, 14]
my_list.sort(key=lambda x: x%2)
print(my_list)
```

Выведет сначала чётные числа (то есть делённые на 2 без остатка - остаток 0, а затем нечётные, которые на 2 не делятся)

```
# Выведет: [8, 2, 14, 7, 5, 11, 1]
```

Лямбда-функции в Python являются анонимными, то есть без имени. Это означает, что функция безымянна. Как известно, ключевое слово def используется в Python для определения обычной функции. В свою очередь, ключевое слово lambda используется для определения анонимной функции. Как записывается Лямбда-функции:

Lambda аргументы: выражение.

Лямбда-функции могут иметь любое количество аргументов, но у каждой такой функции может быть только одно выражение. Выражение вычисляется и

возвращается. Эти функции могут быть использованы везде, где требуется объект-функция.

Например:

```
double = lambda x: x*2
```

```
print(double(5))
```

```
# Выведет: 10.
```

Рассмотрим как это работает?

В функцию `lambda` записывается только один аргумент, над которым будет производиться какое-то действие. Данный аргумент мы обозначили `x`. Над этим аргументом совершается действие: умножения на 2. И после этого полученное значение записывается в переменную `double`. Если в `double` как в функцию запишем какое-либо число (в нашем примере - 5, то в результате получим 10). В обычной функции `def`, которая называется именованной, то есть имеет имя и, однажды описав эту функцию, мы можем вызывать её в любом месте программы, любое количество раз, подставляя только разные аргументы:

```
def double(x):
```

```
    return x * 2
```

```
...
```

```
# Какой-то код программы
```

```
#...
```

```
print(5)
```

```
# Выведет: 10
```

```
#...
```

```
#Продолжение кода программы
```

```
#...
```

```
print(10)
```

```
# Выведет: 20
```

Ещё пример для Лямбда-функции, уже с 2 аргументами :

```
summa = lambda x, y: x + y*2
```

```
print(summa(4, 10))
```

```
# Выведет: 24.
```

Важно, что все элементы списка должны быть одинакового типа (строки, числа, кортежи).

6. pop - позволяет получить элемент и удаляет его по номеру позиции искомого элемента в списке (целое число)

Если такое число не указано, считается, что имеется в виду последний элемент списка. Отрицательные числа поддерживаются..

```
mylist = [1, 2, 3, 4, 5]
```

```
x = mylist.pop()
```

```
print(x)
```

```
# Выведет: 5
```

```
print(mylist)
```

```
# Выведет: [1, 2, 3, 4]
```

Здесь в скобках нет аргумента. Поэтому выбирается последнее число в списке. Это 5, и оно присваивается переменной x. А из списка оно удаляется. И список выглядит так: [1, 2, 3, 4]

Если число в скобках указывается, то удаляется элемент списка под этим индексом:

```
mylist = [1, 2, 3, 4, 5]
```

```
y = mylist.pop(2)
```

```
print(y)
```

```
# Выведет: 3
```

```
print(mylist)
```

```
# Выведет: [1, 2, 4, 5]
```

Здесь уже указывается позиция в списке, в данном случае 2. Выходное значение этой функции присваивается переменной y. Теперь список выглядит так: [1, 2, 4, 5]

Использование отрицательного числа:

```
my_list = [1, 2, 3, 4, 5]
```

```
negative_number = my_list.pop(-2)
```

```
print(negative_number)
```

```
# Выведет: 4
```

```
print(my_list)
```

```
# Выведет: [1, 2, 3, 5]
```

Здесь, если нужно получить последнее значение из этого списка (5), то ставим -1. В примере используется -2. Это указывает на предпоследний элемент списка (4).

Встроенные функции

В языке Python есть функции, аргументами которых являются объекты. Пишется встроенная функция, а в скобках объект, то есть сам список.

7. len — получение длины списка (от слова length — длина), то есть количество элементов в списке

```
mylist = [1, 2, 3, 4, 5]
```

```
list_length = len(mylist)
```

```
print(list_length)
```

```
# Выведет: 5
```

8. max - возвращает элемент с максимальным значением

```
mylist = [1, 2, 3, 4, 25, 4, 3, 2, 1]
```

```
max_value = max(mylist)
```

```
print(max_value)
```

```
# Выведет: 25
```

9. min — возвращает элемент с минимальным значением

```
mylist = [1, 2, 3, 4, 5]
```

```
min_value = min(mylist)
```

```
print(min_value)
```

```
# Выведет: 1
```

10. Приведение к типу

Из любого итерируемого объекта можно получить список. Для этого нужно передать итерируемый объект функции `list()`. Этим объектом может быть множество, кортеж, строка или генератор.

Множество, кортеж, генератор будут рассмотрены позже.

```
myset = {1, 2, 3, 4} # Множество
```

```
mytuple = (1, 2, 3, 4) # Кортеж
```

```
mystring = 'Хорошая погода' # Строка
```

```
list1 = list(myset)
```

```
list2 = list(mytuple)
```

```
list3 = list(mystring)
```

```
print(list1)
```

```
# Выведет: [1, 2, 3, 4]
```

```
print(list2)
```

```
# Выведет: [1, 2, 3, 4]
```

```
print(list3)
```

```
# Выведет: ['Х', 'о', 'р', 'о', 'ш', 'а', 'я', ' ', 'п', 'о', 'г', 'о', 'д', 'а']
```

Примечание: ни в коем случае не называйте переменную как `list`, иначе, в дальнейшем, при использовании функции `list()` для приведения к типу Список, будет выводиться ошибка `TypeError: 'list' object is not callable` (`TypeError`: объект `'list'` не может быть вызван).

Урок 3 часть 1

Кортежи

Кортеж (tuple, тупл)— список элементов, который не может изменяться.

```
d = (200, 10)
```

```
print(d[0])
```

```
# Выведет: 200
```

Формально кортеж определяется наличием запятой, поэтому, если кортеж имеет только одно значение, то после него обязательно идёт запятая:

```
my_t = (3,)
```

Срез кортежа

Срез позволяет вызывать несколько значений кортежа в диапазоне [start:stop] (start – включительно, stop – исключительно).

Например:

```
d = (200, 10, 1) # Здесь 200 имеет индекс 0, 10 — индекс 1, 1 имеет индекс 2.
```

```
b = d[0:3]
```

```
print(b)
```

```
# Выведет: (200, 10, 1)
```

Здесь срез из кортежа d с индексами от 0 до 3. 0 - включается, 3 - исключается. И так срез повторяет все индексы кортежа d (от 0 до 2) и записываются в кортеж b. Таким образом происходит полный повтор всех значений, которые имеются в первоначальном кортеже.

```
d = (200, 10, 1)
```

```
b = d[0:2]
```

```
# 0 — включаетмя, 2 — исключается. Остаются только индексы 0 и 1, что соответствует значениям 200 и 10.
```

```
print(b)
```

```
# Выведет: (200, 10)
```

Индекс 2 относится к значению 1. Так как последний индекс в срезе (2), то он исключается.

Операция присваивания кортежа

Любому объекту с именем можно присвоить кортеж. Примеры присваивания кортежей.

```
a = () # пустой кортеж
```

```
b = (5, 'Hello', True) # кортеж из 3-х элементов различных типов
```

```
c = ('Волга', (2.88, "Праздник Нового года")) # вложенный кортеж
```

```
# В этом примере 2 элемента кортежа.
```

```
# Первый элемент строка — 'Волга'
```

Второй элемент — это ещё один кортеж, вложенный в первый, в котором 2 элемента: вещественное число 2.88 и строка "Праздник Нового года".

Кортеж можно создать с помощью функции tuple(). Эта функция может в скобках принимать любой объект. Например, другой кортеж, список, строка.

Например строка:

```
d = tuple('Волга')
```

```
print(d)
```

```
# Выведет: ('В', 'о', 'л', 'г', 'а')
```

Кортеж из списка

```
lst = [2, "abc", 3.88] # Заданный список
```

```
e = tuple(lst) # Создаём кортеж
```

```
print(e)
```

```
# Выведет: (2, 'abc', 3.88)
```

Получился кортеж e со значениями из списка lst.

Кортеж из другого кортежа

```
f = tuple((3, 2, 0, -5))
```

```
print(f)
```

```
# Выведет: (3, 2, 0, -5)
```

Кортеж (3, 2, 0, -5) вставляется с помощью функции tuple() и присваивается переменной f. Теперь переменная f становится кортежем с теми же значениями, которые были переданы функции tuple().

Функция получения значения элемента по индексу

Достаточно указать номер индекса в квадратных скобках после названия кортежа.

Пример функции получения значения элемента кортежа по номеру индекса:

```
a = ('Работа', 'Новый день', 'Завтра', 'Всегда')
```

```
item = a[2]
```

```
print(item)
```

```
# Выведет: Завтра
```

Кортеж, содержащий другой кортеж и его элементы

```
a = ('Работа', 'Новый день', 'Завтра', 'Всегда')
```

```
b = (a, a[1], True)
```

```
print(b)
```

```
# Выведет: (('Работа', 'Новый день', 'Завтра', 'Всегда'), 'Новый день', True)
```

Здесь в новом кортеже b после присваивания получается 3 элемента: 1. полное повторения кортежа a, 2. один элемент из элементов кортежа a с индексом 1 и значением 'Новый день'. 3. булевская (логическая) переменная со значением True (Правда).

```
item = ('a', 'бв', 'где', 'ёжзи')
```

```
print(item[1])
```

```
# Выведет: бв
```

Здесь функция print() распечатывает один элемент из кортежа item с индексом 2, что соответствует значению бв.

Кортеж, содержащий список

```
cor = (1, [2, 3, 4], "text")
```

Кортеж cor содержит 3 элемента. Вторым элементом с индексом 1 является список [2, 3, 4]

Вынуть список из кортежа

```
item = cor[1]
```

```
print(item)
```

```
# Выведет: [2, 3, 4]
```

Внутри кортежа список можно изменять

```
cor = (1, [2, 3, 4], "text")
```

```
print(cor[1][0])
```

```
# Выведет: 2
```

Теперь присвоим этому элементу другое значение

```
cor[1][0] = 8
```

```
print(cor)
```

```
# Выведет: (1, [8, 3, 4], 'text')
```

Если в предыдущем примере мы распечатывали целиком список [2, 3, 4] из исходного кортежа, который был в кортеже под индексом 1, то чтобы добраться до элемента этого списка нам придётся указать ещё один индекс уже для этого списка [1][0].

Такой подход называется вложением (в данном случае списка в кортеж) и это второй уровень вложения.

Бывает и третий уровень вложения. Пусть в переменную cor3 будет присвоен кортеж с 3 элементами: список, целое число и строка. А в список встроен в том числе кортеж. Это может выглядеть так:

```
cor3 = ( [ 7, True, ('Длина', 'Ширина', 'Высота')], 12, "Программа")
```

```
item1 = cor3[0]
```

```
print(item1)
```

```
#[7, True, ('Длина', 'Ширина', 'Высота')]
```

#Выводим 1 уровень вложенности. Индекс [0] кортежа cor3 указывает на первый элемент кортежа. И это список

```
item2 = cor3[0][2]
```

```
print(item2)
```

```
#('Длина', 'Ширина', 'Высота')
```

Выводим 2 уровень вложенности. Составной индекс [0][2] кортежа cor3 указывает на третий элемент Списка. И это кортеж.

```
item3 = cor3[0][2][1]
```

```
print(item3)
```

```
#Ширина
```

#Выводим 3 уровень вложенности. Составной индекс [0][2][1] кортежа cor3 указывает на второй элемент в кортеже. И это строка.

Взятие среза в кортеже

С помощью операции взятия среза можно получить другой кортеж.

```
cor1 = ('Работа', 'Новый день', 'Завтра', 'Всегда')
```

```
i = 1
j = 3
cor2 = cor1[i:j]
print(cor2)
```

Выведет: ('Новый день', 'Завтра')

В новом срезе отбрасывается элемент с индексом 0 (значение 'Работа') и отбрасывается элемент с индексом 3 (значение 'Всегда').

Иногда начальные или конечные индексы срезов не указываются.

Например отсутствует начальный срез:

```
cor1 = ('Работа', 'Новый день', 'Завтра', 'Всегда')
cor2 = cor1[:3]
print(cor2)
```

Выведет: ('Работа', 'Новый день', 'Завтра')

Это означает, что срез начинается с начального индекса, то есть с 0 и заканчивается предыдущим от индекса 3 значением, то есть 2 ('Завтра').

Пример с отсутствием конечного индекса в срезе:

```
cor1 = ('Работа', 'Новый день', 'Завтра', 'Всегда')
cor2 = cor1[1:]
print(cor2)
```

Выведет: ('Новый день', 'Завтра', 'Всегда')

Срез начинается с индекса 1, что соответствует значению 'Новый день', и заканчивается конечным индексом кортежа 3 со значением 'Всегда'

Конкатенация (сложение)

Для кортежей можно выполнять операцию конкатенации (сложения), которая обозначается символом '+'. В простейшем случае для конкатенации двух кортежей общая форма операции следующая:

```
cor3 = cor1 + cor2
```

где:

cor1, cor2 – кортежи, для которых нужно выполнить операцию конкатенации.

cor1 и cor2 обязательно должны быть кортежами. При выполнении операции конкатенации для кортежей, использовать в качестве операндов любые другие типы (строки, списки) запрещено.

cor3 – кортеж, который будет результатом конкатенации (сложения).

Пример.

```
cor1 = (1, 2, 3)
cor2 = (4, 5, 6)
cor3 = cor1 + cor2
print(cor3)
```

Выведет: (1, 2, 3, 4, 5, 6)

#Конкатенация кортежей со сложными объектами

```
cor1 = (3, "abc")
cor2 = (-7.22, ['a', 5])
cor3 = cor1 + cor2
```

```
print(cor3)
# Выведет: (3, 'abc', -7.22, ['a', 5])
```

```
# Конкатенация трех кортежей
cor1 = ('a', 'aa', 'aaa')
cor2 = cor1 + (1, 2) + (True, False)
# Выведет: ('a', 'aa', 'aaa', 1, 2, True, False)
```

Повторение *

Кортеж может быть образован путем операции повторения, обозначаемой символом '*'. При использовании в выражении общая форма операции следующая:

```
cor2 = cor1 * n
```

здесь

cor2 – результирующий кортеж;

cor1 – исходный кортеж, который нужно повторить n раз;

n – количество повторений кортежа cor1.

Пример:

```
cor1 = (1, 2, 3)
cor2 = cor1 * 3
print(cor2)
# Выведет: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
# Кортеж, который содержит вложенные объекты
```

```
cor1 = ("ab", ["1", "12"])*2
print(cor1)
# Выведет: ('ab', ['1','12'], 'ab', ['1','12'])
```

Обход кортежа в цикле

Элементы кортежа можно последовательно просмотреть с помощью операторов цикла while или for.

Цикл for

```
cor1 = ('a', 'бв', 'где', 'ёжзи')
# Вывести все элементы кортежа
for item in cor1:
    print(item)
# Выведет:
a
бв
где
ёжзи
```

Цикл while

```
# Исходный кортеж - целые числа
cor1 = (-1, 3, -8, 12, -20)
# Вычислить количество положительных чисел
```

```

i = 0
k = 0 # количество положительных чисел
while i < len(cor1):
    if cor1[i]>=0:
        k = k + 1
    i = i + 1
# Вывести результат
print("k = ", k)
# Выведет: k = 3

```

В этом примере осуществляется проход по всем значениям `while` и суммируется количество положительных чисел кортежа `cor1`.

Как это работает?:

Вначале задаём кортеж `cor1` с положительными и отрицательными числами.

Затем присваиваем текущему индексу кортежа `i = 0` и `k = 0` - счётчик количества положительных чисел.

Далее запускаем цикл `while` для прохода по всем значениям кортежа - `while i < len(cor1)`:

Условие `i < len(cor1)` проверяется в каждом проходе цикла. пока индекс `i` не достигнет конца кортежа. `len(cor1)` — это функция, определяющая длину кортежа `cor1` (`len` укороченное слово `length` -длина). Проходя очередной цикл, индекс `i` увеличивается на 1 (`i = i + 1`). Достигнув максимальной длины индекс `i`, в данном случае 6. В конце выполнения последнего цикла, когда `i` будет равно 7, цикл прекратится поскольку условие `i < len(cor1)` уже не будет выполняться.

Внутри цикла `while` происходит проверка очередного числа на положенительность. Используется оператор условия `if`. И, если очередное число является положительным (`if cor1[i]>=0`: - больше или равно 0), то счётчик положительных чисел `k` квеличивается на 1 (`k = k + 1`)

Функция распечатки `print("k = ", k)` имеет два аргумента, один - символьная переменная `"k = "`, другой — целое число `k` — счётчик положительных чисел кортежа `cor1`.

Оператор `in` - проверка вхождения элемента в кортеж (есть ли заданный элемент в кортеже?)

Пусть есть кортеж, который содержит строки:

```
cor1 = ('Работа', 'Новый день', 'Завтра', 'Всегда')
```

или кортеж содержащий числа:

```
cor2 = (25, 148, 3.78, 99)
```

Для проверки вхождения элемента в кортеж применим программный ввод искомого значения. Это достигается применением функции ввода `input()`, которая позволяет пользователю самому ввести какое-либо значение, присвоив его переменной. Например, введите следующие строки и проверьте результат:

```

item = input()
print(item)

```

Когда запускается первая строка программа ожидает ввода какого-либо значения от пользователя и после ввода присваивает новое значение для переменной. Вторая строчка распечатывает значение переменной.

Надо иметь ввиду, что при вводе чисел (целых - int или вещественных — float), функция input() преобразует их в строковую переменную (введём 5 - получим '5', введём 3.62 - получим '3.62'). Вводимая строка остаётся строкой. Поэтому, полученную переменную можно не преобразовывать. Дополнительно в скобках функции input() можно указать, что именно надо вводить. Вводимая строка не содержит кавычек.

```
item = input("Введите строку: ")  
print(item)
```

Вводимое число требует преобразования из строкового вида в числовой формат. Целые числа преобразуются из строки в число функцией int(), вещественные — функцией float().

Есть два варианта такого преобразования:

1. Сначала получаем переменную (в данном случае item) и уже к ней применяет преобразование в число:

```
item1 = int(item)
```

или

```
item1 = float(item)
```

2. Преобразование идёт сразу с вводом. Для этого внутри функции int() или float() помещается функция ввода input. Сначала выполняется, которая находится внутри скобок, то есть функция ввода. Затем идет преобразование строкового вида числа в числовой.

Например:

```
item = int(input("Введите число: "))  
print(item + 1)
```

Если число не было преобразовано при распечатке (в данном случае арифметического действия прибавления 1 к введённому числу) произойдёт ошибка.

И так кортеж с содержанием строк:

```
cor1 = ('Работа', 'Новый день', 'Завтра', 'Всегда')
```

```
item = input("Введите строку: ")
```

```
if 'Новый день' in item:
```

```
    print("Переменная 'Новый день' имеется в кортеже cor1")
```

```
else:
```

```
    print("Переменной 'Новый день' нет в кортеже cor1")
```

Оператор условия if проверяет вхождение строки 'Новый день' в кортеж cor1. В данном случае в кортеже cor1 имеется строка 'Новый день'. Поэтому будет выводиться сообщение: Переменная 'Новый день' имеется в кортеже cor1. Если бы в кортеже не было бы вводимой строки, например, строки 'Добрый вечер', то отработал бы оператор else и распечаталось сообщение Переменной 'Добрый вечер' нет в кортеже cor1.

Надо отметить, что при использовании строк, в которых есть кавычки (Переменной 'Новый день' нет в кортеже cor1), всю строку обрамляют кавычками другого вида. Если внутри строки были одинарные кавычки ('), то сама строка обрамляется двойными кавычками ("). В Python существуют и другие формы кавычек — тройные одинарные кавычки (') и тройные двойные кавычки ("""), которыми удобно не только комментировать большие тексты, но и обрамлять строки, внутри которых есть простые одинарные или двойные кавычки. Теперь кортеж с числами.

Для целого числа:

```
cor2 = (25, 148, 3.78, 99)
item2 = int(input('Введите число: ')) # Вводится число 25.
if 25 in item2:
    print('Число 25 имеется в кортеже cor2')
else:
    print('Число 26 отсутствует в кортеже cor2'))
```

Для вещественного числа:

```
item2 = float(input('Введите число: ')) # Вводится число 3.78
if 25 in item2:
    print('Число 3.78 имеется в кортеже cor2')
else:
    print('Число 3.62 отсутствует в кортеже cor2'))
```

Методы работы с кортежами

Методами называются функции, воздействующие на объекты. В данном случае объектом является кортеж. Записывается метод через точку после объекта (кортежа).

Метод index(). Поиск позиции элемента в кортеже

Чтобы получить индекс (позицию) элемента в кортеже, нужно использовать метод index(). Общая форма вызова метода следующая:

```
cor2 = (25, 148, 3.78, 99)
item = 148
pos = cor2.index(item)
```

где:

cor2 - кортеж, в котором осуществляется поиск.

Item — переменная, которой присвоено значение 148 для поиска этого значения в кортеже cor2.

pos – позиция (индекс) элемента item в кортеже.

Первому элементу соответствует позиция 0. Если элемента нет в кортеже, генерируется исключительная ситуация. Поэтому, перед использованием метода index() рекомендуется делать проверку на наличие элемента (с помощью операции in).

```
cor2 = (25, 148, 3.78, 99)
item = int(input('Введите число: '))
if item in cor2:
    pos = cor2.index(item)
```

```
print(f'Число {item} имеется в кортеже cor2 и имеет позицию {pos}')
else:
    print(f'Число {item} отсутствует в кортеже cor2'))
```

В этом коде нужно обратить внимание проверку вхождения переменной `item` (if `item in cor2:`), значение которой введено нами в строчке `item = int(input('Введите число: '))`.

Далее надо обратить внимание на функцию форматирования строки `f`, которая ставится перед выводимой строкой. В фигурных скобках `{}` задается имя переменной, которая при распечатке строки будет заменено на своё значение. Это и `{item}`, и `{pos}`.

Метод `count` - подсчет количества вхождений элемента в кортеж

Задан кортеж:

```
cor = ("ab", "ac", "ab", "ab", "ca", "ad", "jklmn")
d1 = cor.count("ab")
print('Количество вхождений "ab": ', str(d1))
# Выведет: Количество вхождений "ab": 3
d2 = cor.count("jprst")
print('Количество вхождений "jprst": ', str(d2))
# Выведет: Количество вхождений "jprst": 0
d3 = cor.count("ca")
print('Количество вхождений "ca": ', str(d3))
#Количество вхождений "ca": 1
```

В этом коде функция `f` имеет 2 аргумента: строковый 'Количество вхождений {"ca"}: ' и числовой `d3` — посчитанное количество вхождений элемента "ca" в кортеж `cor`. Записываются аргументы через запятую. Так как функция распечатки `print()` работает только со строковыми аргументами, то обязательно числовой аргумент нужно преобразовать в строку `str(d3)`. Здесь же опять нужно было предусмотреть строковую переменную 'Количество вхождений "ca": ' обрaмить одинарными кавычками, а внутри этой строки использовать двойные кавычки.

Генератор списков

В языке программирования Python существует специальная синтаксическая конструкция, которая позволяет по определенным правилам создавать заполненные списки. Такие конструкции называются генераторами списков. Например, надо создать список, заполненный целыми числами до определенного числа. "Классический" способ будет выглядеть примерно так:

```
list = []
for i in range(1,15):
    list.append(i)
print(list)
# Выведет: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Сначала переменной `list` присваивается пустой список (`list = []`). Затем в цикле `for` переменной `i` с помощью функции `range(1,15)` присваиваются значения от 1 до 14 (15 не будет входить в этот список). Далее текущее значение `i` добавляется в список `list` с помощью метода `append(i)`.

На создание списка ушло три строчки кода. Генератор же сделает это за одну:

```
list = []  
list = [i for i in range(1,15)]  
print(list)  
# Выведет: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Здесь конструкция `[i for i in range(1,15)]` является генератором списка. Вся конструкция заключается в квадратные скобки, что как бы говорит, что будет создан список. Внутри квадратных скобок можно выделить три части: 1) что делаем с элементом (в данном случае с переменной 'i' ничего не делаем, просто добавляем её значение в список), 2) что берем (в данном случае элемент 'i'), 3) откуда берем (здесь из объекта `range`). Части отделены друг от друга ключевыми словами `for` и `in`.

```
list = [2, -2, 4, -4, 7, 5]  
list1 = [i**2 for i in list]  
print(list1)  
# Выведет: [4, 4, 16, 16, 49, 25]
```

Урок 3 часть 2

Знакомство со списками позволяет нам решать практически любые задачи. В некоторых случаях полезными могут оказаться и другие сложные типы данных. Это - множества и словари.

Множества

Множество в Python - "контейнер", содержащий не повторяющиеся элементы в случайном порядке. Или, по-другому, множество - это неупорядоченный набор уникальных элементов. Или грубо: множество это мешок с уникальными значениями.

Множества можно использовать для выполнения математических операций вроде объединения, пересечения, симметрической разности и так далее с использованием методов и функций.

Напомним, что метод привязан к объекту и ставится через точку после объекта: `объект.метод(аргументы)`.

Функция имеет объект в качестве аргумента, заключённого в круглые скобки: `функция(объект)`.

Создадим пустое множество :

```
a = set()  
print(a)  
# Выведет: set()
```

Для чего может создаваться пустое множество? Чаще всего оно создаётся для добавления данных в процессе выполнения программы на основе других данных.

Например:

```
b = [1,2,3,4,5,4,3]
```

```
a = set()
```

```
for elem in b:
```

```
    a.add(elem)
```

```
print(a)
```

```
# Выведет: {1, 2, 3, 4, 5}
```

Пусть дан список, в котором элементы могут повторяться `b = [1,2,3,4,5,4,3]`.

Сначала создаём пустое множество `a = set()`. Потом в цикле поэлементно из списка `b` переписываются данные в множество `a`.

```
b = [1,2,3,4,5,4,3]
```

```
for elem in b:
```

```
    a.add(elem)
```

```
print(a)
```

```
# Выведет: {1, 2, 3, 4, 5}
```

Обратите внимание на то, что после цикла идёт пустая строка перед распечаткой полученного множества `a` `print(a)`. Иначе, без этой пустой строки, когда мы скопировали весь код данной программы и вставили в интерпретатор Python, мы можем получить ошибку. Если же мы копируем в интерпретатор Python каждую строку программы и пускаем на выполнение, такой ошибки не будет даже без пустой строки после выполнения цикла. Так же не будет ошибки, если мы запускаем программу из файла командой терминала:

```
python3 имя_программы.py
```

Когда идёт перезапись данных в цикле из первоначального списка `b` в множество `a`, происходит уничтожение повторяющихся данных. В итоге, в результирующем множестве остаются только уникальные значения данных `{1, 2, 3, 4, 5}`.

Создадим непустое множество из одного слова. Как всегда символьная строка или слово должны быть обрамлены кавычками:

```
a = set('hello')
```

```
print(a)
```

```
# Выведет: {'h', 'e', 'o', 'l'}
```

Как видим символьное слово `'hello'` разложилось на символьные буквы. При чём, одновременно были уничтожены повторяющиеся буквы. Буквы, в этом случае, не упорядочены.

Добавим уже к существующему множеству `a`, которое получилось после записи `a = set('hello')` новое слово `'Восток'`.

Получим вот такую картину:

```
a = set('hello')
```

```
a.add('Восток')
```

```
print(a)
```

```
# Выведет: {'h', 'e', 'o', 'Восток', 'l'}
```

Здесь просто вклинилась целиковая строка 'Восток', а первоначальное слово 'hello' разложилось по буквам, с учётом уничтожения повторяющихся букв.

Порядок выводимых символов и строк может меняться, так как элементы множества носят неупорядоченный характер. Можно, например, выйти из интерпретатора Python, нажав Ctrl+Z, снова запустить интерпретатор и проверить действие данного программного кода.

Есть два варианта получения нужного результата.

Первое. Использовать пустое множество с поэтапным добавлением строк (в данном случае слов)

```
a = set()
a.add('hello')
a.add('Восток')
print(a)
# Выведет: {'hello', 'Восток'}
```

Второе. Не использовать функцию `set()`. А присвоить множеству непосредственно символьные строки (слова) через запятую в фигурных скобках.

```
a = {'hello', 'Восток'}
print(a)
# Выведет: {'hello', 'Восток'}
```

Генератор множеств

Пример задания множества чисел, которые идут подряд:

```
a = {0,1,2,3,4,5,6,7,8,9}
```

Но можно использовать генератор множеств, основанный на функции задания диапазона чисел `range()`.

```
a = {i for i in range(10)}
print(a)
# Выведет: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Диапазон начинается с 0 и заканчивается предпоследним числом, указанным в функции `range()`, то есть, в данном случае, 9, хотя указано число 10.

Сначала задаётся переменная `i`, которой в цикле `for` будет присваиваться очередное значение из заданного диапазона. Потом через пробел идёт сам цикл `for`, с использованием заданной переменной `i`.

К заданной переменной `i` можно применить какую-нибудь функцию, например, умножение на 3.

```
a = {i * 3 for i in range(10)}
print(a)
# Выведет: {0, 3, 6, 9, 12, 15, 18, 21, 24, 27}
```

Множества имеет тоже написание, что и словари, которые будут рассмотрены дальше. Это фигурные скобки `{}`.

Но пустое множество с помощью фигурных скобок создать нельзя.

```
a = {} # А так нельзя!
type(a)
# Выведет: <class 'dict'>
```

Как видим, получается словарь `<class 'dict'>`, а не множество.

Если мы запишем в фигурных скобках набор каких-либо элементов через запятую, присвоив его переменной, то получим множество:

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

```
type(a)
```

```
# Выведет: <class 'set'>
```

Множества удобно использовать для удаления повторяющихся элементов:

```
words = ['hello', 'daddy', 'hello', 'mum']
```

```
a = set(words)
```

```
print(a)
```

```
# Выведет: {'hello', 'daddy', 'mum'}
```

Слово 'hello' повторяется в списке дважды и одно из них исчезает, когда мы его преобразуем во множество с помощью функции set().

С множествами можно выполнять большое количество операций: находить объединение, пересечение и так далее.

Разберёмся во взаимодействии множеств между собой. Пусть существуют 2 спортивные секции, например, лыжная секция и секция борьбы самбо. Одна группа (обозначим её группа 1) записалась в лыжную секцию, другая на самбо (группа 2). Но некоторые члены из группы 1 ещё записались в секцию самбо, и несколько членов группы 2 записались ещё и в лыжную секцию. Таким образом у нас получились 3 группы - 1 группа только лыжники, 2 - только самбисты, 3 - группа, которая записана в обе секции.

Были составлены 2 списка (лыжников и самбистов).

Математически это можно записать следующим образом:

А - это у нас список, записавшихся в лыжную секцию.

```
A = ['Сидорова Маргарита Степановна', 'Нмколаев Олег Евгеньевич', 'Добода Семён Маркович', 'Сузанов Алексей Ильич', 'Тропинин Василий Альбертович', 'Крюкова Наталья Сергеевна', 'Тужилин Прокофий Сидорович', 'Настенко Сергей Павлович', 'Кондратенко Игорь Николаевич', 'Прохоров Ростислав Георгиевич', 'Стариков Михаил Владимирович', 'Сергеев Константин Петрович', 'Любомиров Николай Михайлович', 'Дразденко Илья Ильич']
```

В - список, записавшихся в секцию самбо.

```
V = ['Пронин Марк Анатольевич', 'Дразденко Илья Ильич', 'Лихачёв Эдуарт Трофимович', 'Гордеев Василий Васильевич', 'Кондратенко Игорь Николаевич', 'Свеиланов Руслан Григорьевич', 'Сергеев Константин Петрович', 'Лежебоков Георгий Алексеевич', 'Прокофьев Леонит Петрович', 'Тропинин Василий Альбертович', 'Гордеев Василий Васильевич']
```

Почему в этих списках записаны фамилии, имена и отчества участников секций?

Дело в том, что, если мы запишем только фамилии, то могут быть однофамильцы, что будет искажать поиск в этих списках. Прибавление к фамилии ещё имени может не спасти положение. Есть вероятность существования людей с одинаковыми именами и фамилиями.

А, вот, фамилия, имя и отчество, практически не повторяются.

Допустим, тот, кто составлял эти списки, мог одного и того же человека записать несколько раз.

Вот здесь и пригодится использование множеств.

```
setA = set(A) # Преобразуем список A в множество setA
```

```
setB = set(B) # Преобразуем список B в множество setB
```

```
# Проверяем количество элементов в исходных списках и в полученных множествах.
```

```
print(len(A))
```

```
# Выведет: 14
```

```
print(len(setA))
```

```
# Выведет: 14
```

```
print(len(B))
```

```
# Выведет: 11
```

```
print(len(setB))
```

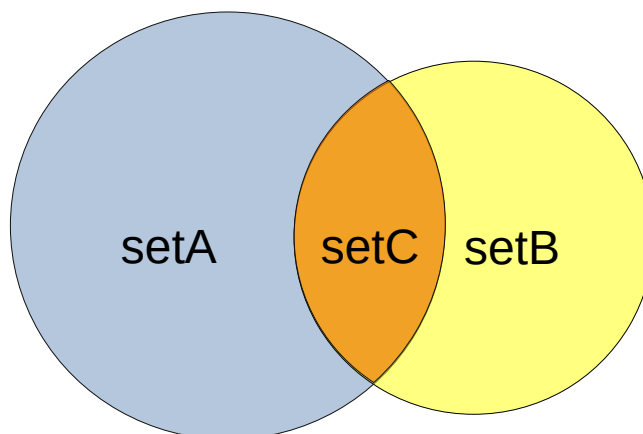
```
# Выведет: 10
```

Как видим, количество элементов в списке A и во множестве setA совпадают.

Количество элементов в списке B и во множестве setB различаются на 1.

Это означает, что тот, кто составлял список второй группы ошибся и включил кого-то дважды. И это исправил переход списка во множество (setB = set(B)).

Условно графическое изображение двух множеств можно представить как как две соприкасающиеся области:



setC - это пересечение множеств setA (лыжники) и setB(самбисты)

Что можно делать со множествами?

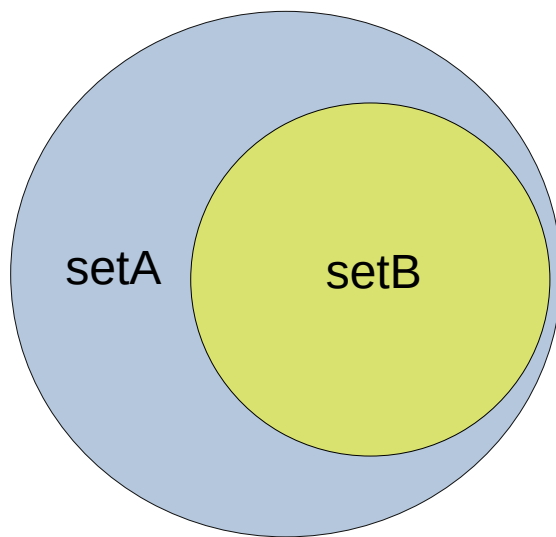
Как уже было показано, можно найти количество элементов в множестве:

len(setA) - число элементов в множестве **setA**

(размер множества 14)

setB in setA - принадлежит ли **setA** множеству **setB**

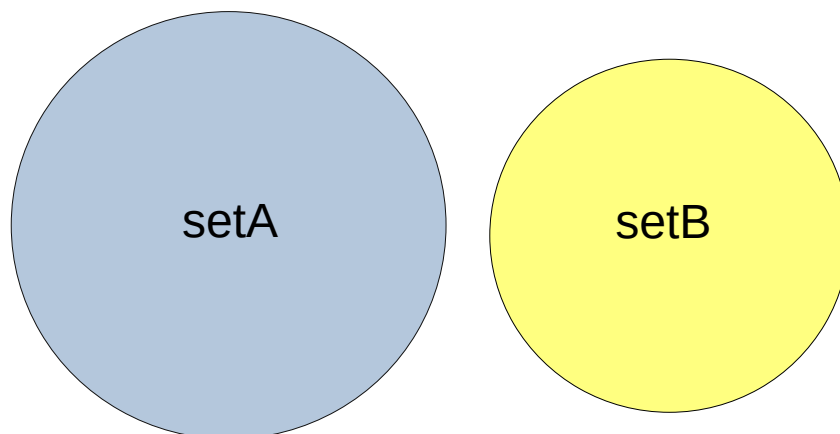
То есть все, кто записался в секцию самбо, ушли из этой секции , например, тренер заболел, и дружно записались в лыжную секцию.



Как видим, визуально множество setA увеличилось в размерах, поглотив множество setB.

setA.isdisjoint(setB) - непересекающийся

В результате получаем истина (True), если множество setA и setB не имеют общих элементов.



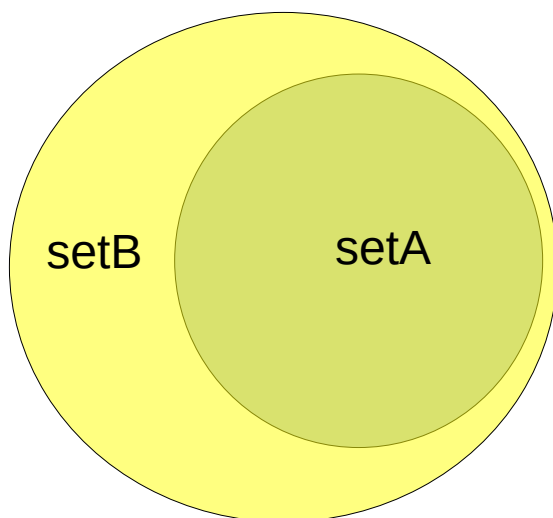
Области множеств не соприкасаются друг с другом.

setB == setA — два равенства означают, что все элементы setA принадлежат setB, все элементы setB принадлежат setA

Пусть у нас было два совершенно одинаковых списка с одинаковыми ФИО (Фамилия Имя Отчество) и каждый записался в две секции. Поэтому можно сказать, что все люди из первой секции принадлежат и второй секции, и, наоборот, люди из 2 секции принадлежат и первой секции.

setA.issubset(setB) или $\text{setA} \leq \text{setB}$ - все элементы setA принадлежат setB

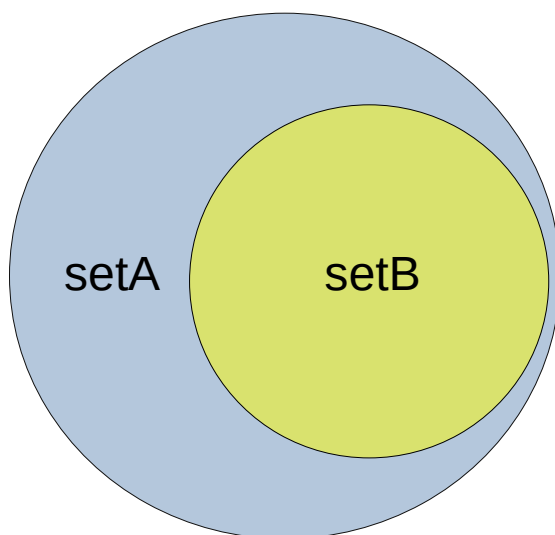
Слово subset переводится как подмножество. Поэтому метод setA.issubset(setB) означает, что множество setA является подмножеством множества setB. Запись setA.issubset(setB) можно заменить записью $\text{setA} \leq \text{setB}$. Знак $\leq$ (меньше или



равно) показывает, что множество setA не может быть больше множества setB, то есть setA входит в setB либо полностью, повторяя все элементы множества setB, либо занимает только часть множества setB.

setA.issuperset(setB)

или $\text{setA} \geq \text{setB}$ — аналогично для множества setB



Оператор $\geq$ (больше или равно) означает, что множество setA больше или равно множеству setB. На следующем примере покажем, что множество setB входит во множество setA либо полностью (отрабатывается знак $=$ из оператора $\geq$), либо занимает только часть множества setA (отрабатывается знак $>$ из оператора $\geq$).

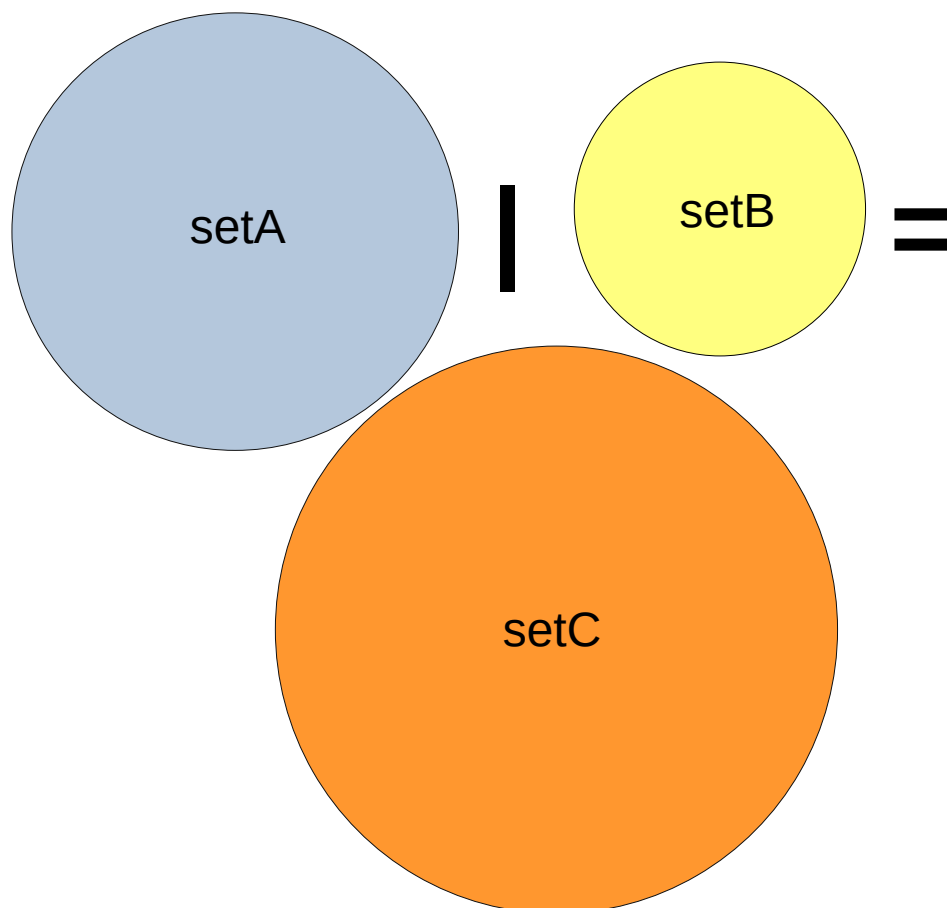
`setA = {"f", "e", "d", "c", "b", "a"}`

```

setB = {"a", "b", "c"}
boolean_value = setA.issuperset(setB)
print(boolean_value)
# Выведет: True, это означает, что setB действительно принадлежит множеству
setA.

```

set.union(setB, ...) или setA | setB | ... - объединение нескольких множеств



В данном случае это объединение двух множеств

Покажем объединение множеств на том же примере лыжников и самбистов.

Список лыжников:

```

A = ['Сидорова Маргарита Степановна', 'Николаев Олег Евгеньевич', 'Добода
Семён Маркович', 'Сузанов Алексей Ильич', 'Тропинин Василий Альбертович',
'Крюкова Наталья Сергеевна', 'Тужилин Прокофий Сидорович', 'Настенко Сергей
Павлович', 'Кондратенко Игорь Николаевич', 'Прохоров Ростислав Георгиевич',
'Стариков Михаил Владимирович', 'Сергеев Константин Петрович', 'Любомиров
Николай Михайлович', 'Дразденко Илья Ильич']

```

Список самбистов:

```

B = ['Пронин Марк Анатольевич', 'Дразденко Илья Ильич', 'Лихачёв Эдуард
Трофимович', 'Гордеев Василий Васильевич', 'Кондратенко Игорь Николаевич',
'Свеилов Руслан Григорьевич', 'Сергеев Константин Петрович', 'Лежебоков
Георгий Алексеевич', 'Прокофьев Леонид Петрович', 'Тропинин Василий
Альбертович', 'Гордеев Василий Васильевич']

```

```
setA = set(A) # Преобразуем список A в множество setA
setB = set(B) # Преобразуем список B в множество setB
setC = set.union(setB, setA)
# Распечатаем количество элементов как в исходных множествах, так и в
полученном.
```

```
print('Количество элементов в множестве setA: ', len(setA))
print('Количество элементов в множестве setB: ', len(setB))
print('Количество элементов в множестве setC: ', len(setC))
# Распечатаются следующие данные
```

Количество элементов в множестве setA: 14

Количество элементов в множестве setB: 10

Количество элементов в множестве setC: 20

Как видим, результат не совпадает с суммой исходных множеств. Почему? Потому что соблюдается всё тот же принцип уничтожения совпадающих элементов в исходных множествах (A в setB и B в setA). Это те люди, которые записались и в самбо и на лыжи.

Ещё надо отметить, что в функции print() участвуют два аргумента, перечисленные через запятую. Первый аргумент - это строка 'Количество элементов в множестве setA: ', как всегда взятая в кавычки. Второй аргумент - вычисляемое количество элементов множества setA с помощью функции len(). В этой функции print() для вывода двух и более аргументов можно использовать функцию форматирования f.

```
print(f'Количество элементов в множестве setA: {len(setA)}')
```

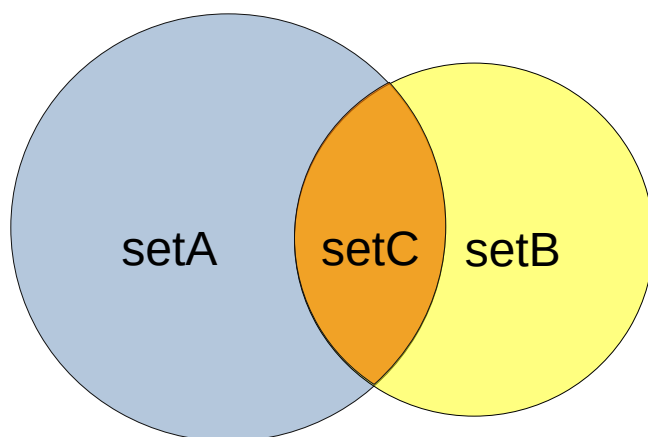
```
print(f'Количество элементов в множестве setB: {len(setB)}')
```

```
print(f'Количество элементов в множестве setC: {len(setC)}')
```

Записывается всё что будет распечатываться записывается в виде строки, обрамлённой кавычками. А перед кавычками ставится функция f. Она то и будет обрабатывать строку. При этом фигурные скобки преобразуются в результаты действия функции len().

setA.intersection(setB, ...) или **setA & setB & ...** - пересечение множеств & - амперсant.

Если сравнивать два множества, то их пересечение пересечение наглядно видно на следующем рисунке. (зона C). Эта зона содержит тех людей, которые



записались из группы лыжников (зона А) ещё и в секцию самбо, а так же самбистов, которые решили ещё заниматься лыжами (зона В).

В аргументах этой функции после запятой стоит троеточие, что означает пересечение трёх и более множеств.

На следующем рисунке показано пересечение трёх множеств:

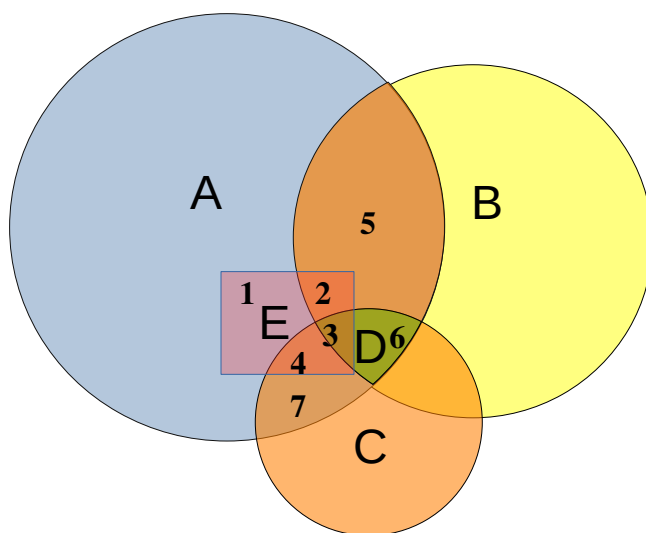
А - зона, относящаяся ко множеству setA - лыжники.

В - зона, относящаяся ко множеству setB — самбисты.

С - зона, относящаяся ко множеству setC (Здесь записаны участники секции плавания).

Д - зона, относящаяся ко множеству setD. Это и есть пересечения трёх множеств: setA - лыжники, setB — самбисты, setC — пловцы.

Е - часть зоны множества setA. Пока на неё не нужно обращать внимания. Ниже будет приведён пример с этой зоной.



Разберём такую ситуацию. У нас есть три списка лыжников, самбистов, пловцов. Пока количество участников в этих списках небольшое, можно потратить какое-то время и выяснить кто из участников каждой группы записались в другую секцию. Но на практике часто группы могут составлять около тридцати и более человек. А, если учесть, что ещё надо умножить на количество групп в каждой секции по возрастному признаку, то вручную это сделать очень трудно, практически невозможно.

Предположим информация по всем группам находится в электронном виде на компьютере. Важно знать, в каком виде была занесена эта информация в компьютер. Если списки участников групп составлены в виде электронных таблиц типа Excel, нужно после их копирования в любой текстовый редактор, создать Python-список в соответствии с правилами Python. В электронных таблицах нет так называемого ограничения целостности данных, поэтому могут быть повторы информации. Электронные таблицы и не предназначены для этого. В таком случае надо использовать перевод сформированного списка во множество, которое отбросит все повторы.

```
setA = set(A)
```

```
setB = set(B)
```

`setC = set(C)`

В случае, когда данные берутся из базы данных повторов может и не быть, но данные, получаемые из базы данных приходят в виде Python-списков. Их также нужно преобразовать во множества, поскольку нам придётся манипулировать полученными множествами. И лучше после создания множеств сделать их копию, чтобы не лишиться первоначальной информации в этих множествах и спокойно анализировать исходные множества, не боясь их изменить. Делается это с помощью копирования исходных множеств в страховочные копии.

Например, множество `setA`, относящееся к лыжникам, скопируем во множество `setF` с помощью метода `copy()`.

Например:

`setF = copy( setA)`

Аналогичным образом можно поступить и с другими множествами, копируя их в резервные множества.

А теперь задача.

Пусть у нас имеется большое количество членов секции лыжников в разных возрастных группах. Всем лыжникам предстоит усиленно тренироваться перед ответственными соревнованиями. Но не все участники лыжной секции ходят исправно на тренировки. Список нарушителей дисциплины известен. На рисунке он обозначен квадратом зоны Е. Тренеру надо выяснить причины отсутствия их на тренировках, чтобы оценить могут они участвовать в предстоящих соревнованиях или нет.

Для начала найдём всех участников, которые записались сразу в три секции. Отыщем их пересечение трёх множеств `setA`, `setB`, `setC` с результатом зелёной зоны D. Обозначим эту зону как множество `setD`. Понятно, что это точно будет множество, потому что получаем его из других множеств. Тренер лыжников исходит из того, что людям, записавшимся сразу в три секции не досуг посещать дополнительные тренировки по лыжам. У них просто не будет времени на дополнительные нагрузки. Но некоторые из них посещают лыжную секцию. Вот здесь и встаёт вопрос о привлечении людей из зелёной зоны D (`setD`).

Вообще-то тренеру по лыжам нужно найти множества из зон 1, 2, 3, 4, 5, 6 и 7.

Почему так?

Зона 1 (обозначим как множество `set1`) - люди записались только на лыжи, но не посещают тренировки. Тренер по лыжам однозначно не будет приглашать на дополнительные тренировки перед соревнованиями.

Зона 2 (обозначим как множество `set2`) - записавшиеся и на самбо и на лыжи не посещают лыжи. Но известно, что тренер по самбо заболел. Занятий нет и открывшееся время посещения лыж не используется. Следовательно таких людей привлекать к соревнованиям по лыжам не имеет смысла.

Зона 3 (обозначим как множество `set3`) - записались в три секции, а ещё по самбо занятий временно нет и не посещают лыжную секцию. Однозначно, связываться с ними нельзя.

Зона 4 (обозначим как множество `set4`) - записаны на лыжи и на плавание. Лыжи не посещают. Либо у них совпадает время тренировок по плаванию с

тренировками по лыжам и они предпочли плавание, либо болеют. В любом случае с ними нельзя иметь дело с дополнительными тренировками по лыжам.

Зона 5 (обозначим как множество $set5$) — самбисты, записавшиеся ещё в лыжную секцию и посещают её. Помним, что тренер по самбо заболел, так что все люди из 5 зоны посещают только лыжи. Следовательно на них можно рассчитывать в подготовке к предстоящим соревнованиям.

Зона 6 (обозначим как множество $set6$) — записаны в три секции, но на лыжи ходят. Возможно посещают плавание, а на самбо, как и все самбисты, временно не ходят в связи с болезнью тренера. С ними тренер по лыжам может договориться.

Зона 7 (обозначим как множество $set7$) — практически тоже, что и в зоне 6. Они посещают лыжи и записаны ещё на плавание. С ними договориться можно. И так, контактировать с людьми из зон с 1 по 4 нет смысла, а из зон с 5 по 7 очевидно можно.

Как найти эти зоны и составить список, кого надо обзвонить для согласования тренировок?

Запишем порядок действий нахождения зон с 1 по 7.

1. Находим пересечение множеств лыжников $setA$ и самбистов $setB$. Назовём их пересечение $setA \cap setB$. Смотрим, какие зоны входят в это пересечение. Видим, это зоны 5, 3, 6.

2. Находим пересечение множеств лыжников $setA$ и самбистов $setB$. Назовём $setA \cap setB$.

Сюда входят зоны 3, 4, 6, 7.

3. Находим пересечение трёх множеств лыжников $setA$, самбистов $setB$ и пловцов $setC$. Это зона D или множество $setD \cap setB$. Сюда входят зоны 3, 6.

4. Теперь берём множество $setA \cap setB$, полученное в результате пересечения $setA$ и $setB$, и находим пересечение с множеством $setE$. Получаем множество, которое мы обозначим $setA \cap setB \cap setE$. Сюда войдут зоны 3, 4.

5. Находим множество $set5$ (зона 5), получаемое путём вычитания из множества $setA \cap setB$ множество $setE$. Это первое нужное множество людей, с которыми тренер лыжников будет связываться

6. Повторяем пункты 4 и 5, но уже для множества $set7$.

Из множества $setA \cap setC$ вычитаем множество $setE$.

7. Получение множества $set6$. Это разность множества $setD$ и множества $setE$.

8. Находим объединение полученных множеств $set5$, $set6$, $set7$.

Найденное множество и есть тот самый список людей, с кем необходимо связаться тренеру по лыжам.

Такой пример с работой множеств полезен не только для освоения функций множеств. Во первых, нужно сказать, что такую задачу можно решить с помощью SQL (Structured Query Language - Язык Стандартных Запросов) базы данных. Но запросов будет достаточно много и требуются временные таблицы, которые уничтожаются после окончания сессии работы с базой. Если мы хотим временные таблицы сделать постоянными, мы замусорим базу данных. И в том и другом случае будет большая нагрузка на сервер с базой данных и передачу

данных по каналам связи. Использование же множеств позволит сделать только 4 запроса для получения множеств setA, setB, setC, setE. А дальше уже работа со множествами на локальной машине. Нагрузка на сервер минимальная.

setA.difference(setB, ...) или setA - setB - ... - множество из всех элементов setA, не принадлежащие ни одному из setB

Метод difference() означает разность множеств. Трехточие показывает, что разность может находится трёх и более множеств, перечисленных через запятую.

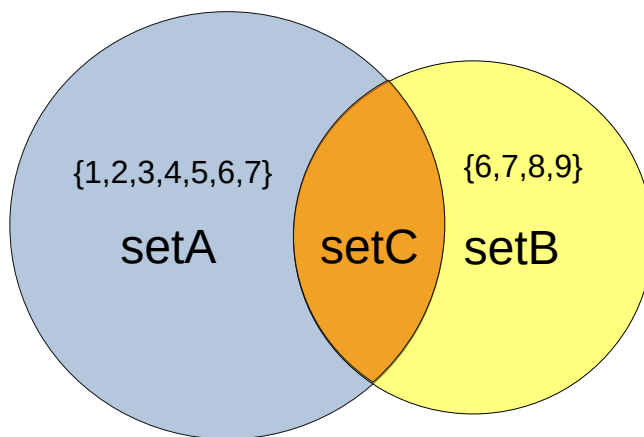
Для наглядности упростим визуально setA и setB

setA = {1,2,3,4,5,6,7}

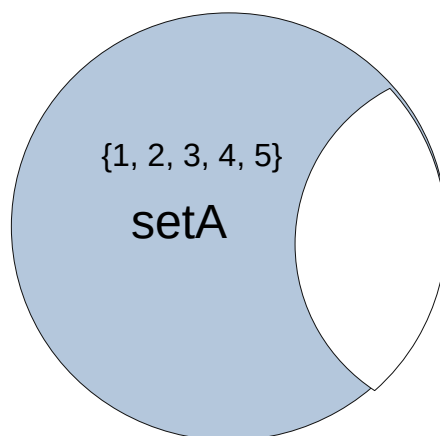
setB = {6,7,8,9}

setA.difference(setB)

Выведет: {1, 2, 3, 4, 5}



В результате множество будет выглядеть следующим образом:



Напоминает откусанный кусок круглого 'сыра'.

И, наоборот, найдём разницу между setB и setA

setB.difference(setA)

Выведет: {8, 9}

Возьмём теперь кроме множеств setA и setB ещё множество setC и найдём разность множеств setB, setC относительно множества setC.

Зададим множеству setC такие параметры:

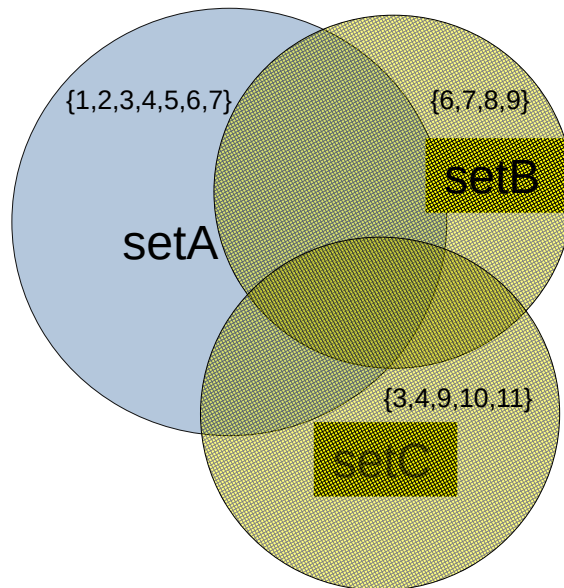
setA = {1,2,3,4,5,6,7}

setB = {6,7,8,9}

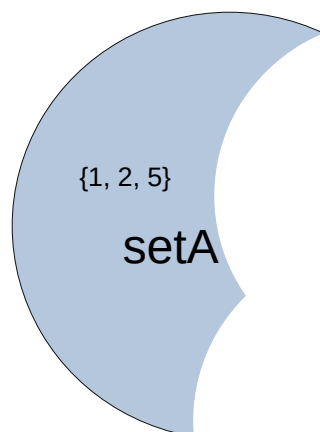
setC = {3,4,9,10,11}

setA.difference(setB, setC)

Выведет: {1, 2, 5} # Этих элементов нет во множествах setB и setC, только во множестве setA.



Получилось:



setA.symmetric_difference(setB); setA ^ setB - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих

Метод `sets.symmetric_difference()` принимает в качестве аргумента любой объект, поддерживающий итерацию по своим элементам. Это может быть список, кортеж, список ключей словаря, или другое множество.

Рассмотрим вариант, где присутствуют только множества `setA` и `setB` и их результат множество `setC`. Метод `symmetric_difference()` позволяет получить новое множество `setC`.

```
setA = {1,2,3,4,5,6,7}
```

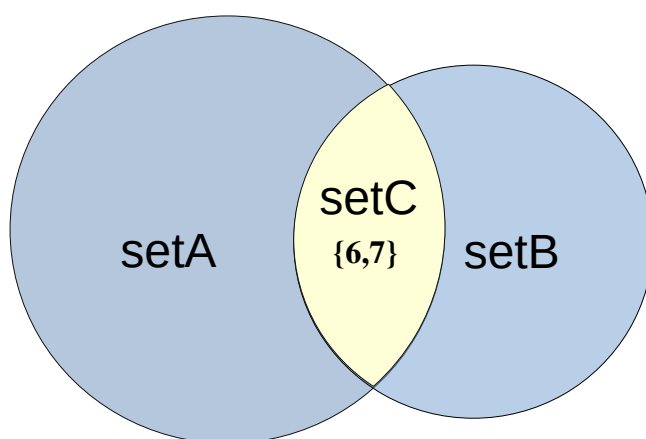
```
setB = {6,7,8,9}
```

```
setC = setA.symmetric_difference(setB)
```

```
print(setC)
```

```
# Выведет: {1, 2, 3, 4, 5, 8, 9}
```

Как видим в результате от двух множеств отбрасывается кусок пересечения этих множеств `{6,7}`.



Из результирующего множества, которое представляет собой объединение двух множеств `setA` и `setB`, как бы вырезана зона пересечения `{6,7}`.

setA.copy() - копия множества

```
setB setA = setA.copy()
```

Здесь мы получаем точную копию множества `setA`, присвоенное множеству `setB`.

Такая операция может понадобиться, если мы хотим поработать с множеством, предполагая изменения в нём, и, в тоже время сохранить исходные данные, работая только с копией.

Операции, которые непосредственно изменяют МНОЖЕСТВО

`setA.update(setB, ...); setA |= setB | ...` - объединение

`setA = {1, 2, 3, 4, 5}`

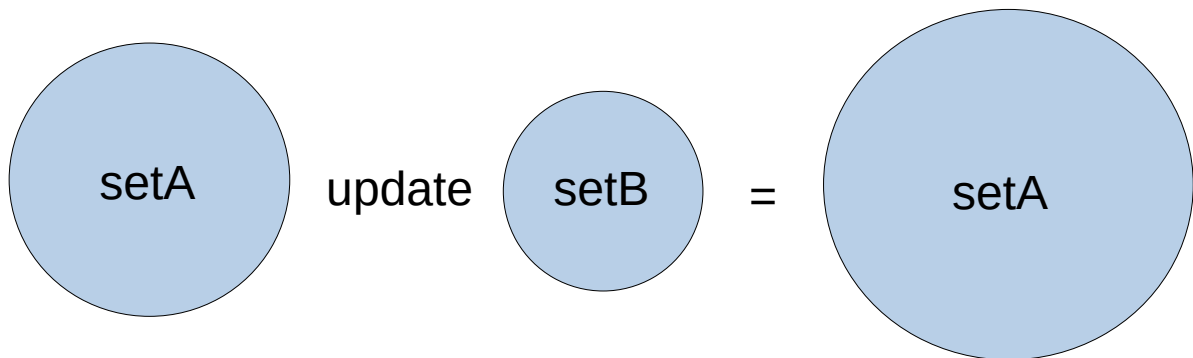
`setB = {4, 5, 6, 7, 8, 9}`

`setA.update(setB)`

`print(setA)`

Выдаст: {1, 2, 3, 4, 5, 6, 7, 8, 9}

Все элементы обоих множеств объединились и, кроме того, были удалены повторы.



Объединение может осуществляться и нескольких множеств.

`setA.intersection_update(setB, ...); setA &= setB & ...` - пересечение

При выполнении данной операции, дубликаты игнорируются.

`setA = {1, 2, 3, 4, 5}`

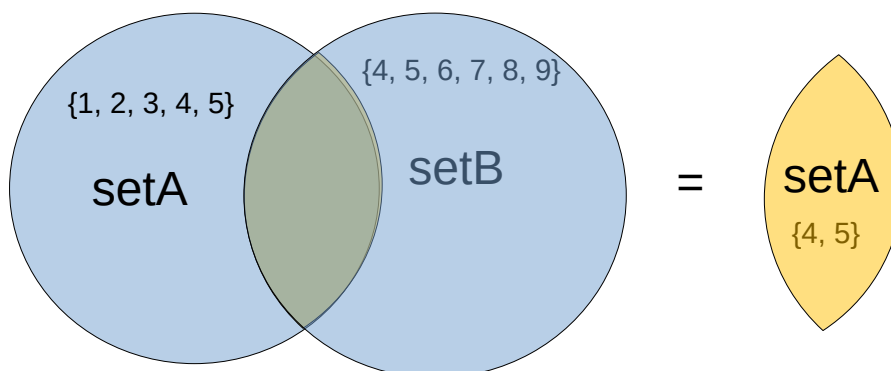
`setB = {4, 5, 6, 7, 8, 9}`

`setA.intersection_update(setB)`

`print(setA)`

#Выдаст: {4, 5}

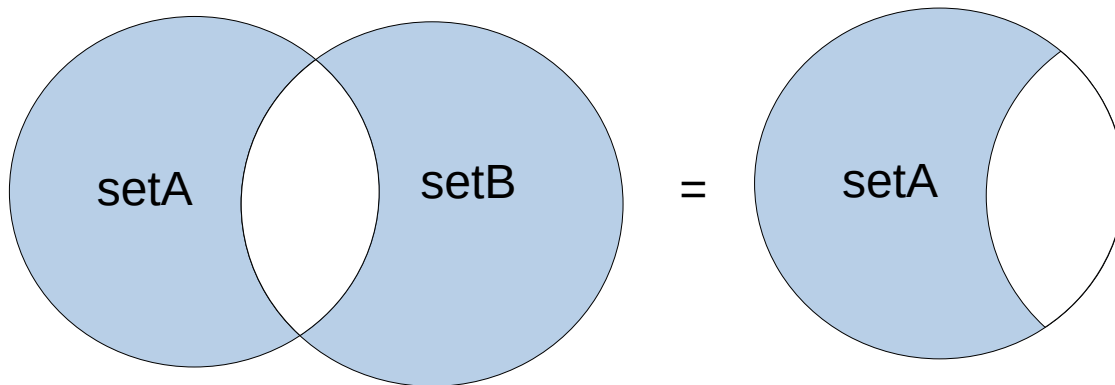
`setA.intersection_update(setB)`



setA.difference_update(setB, ...); setA -= setB | ... - вычитание

```
setA = {1, 2, 3, 4, 5}
setB = {4, 5, 6, 7, 8, 9}
setA.difference_update(setB)
print(setA)
# Выдаст: {1, 2, 3 }
```

setA.difference_update(setB)



setA.symmetric_difference_update(setB) - симметричная разница

`setA ^= setB` - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.

```
setA = {1, 2, 3, 4, 5}
setB = {4, 5, 6, 7, 8, 9}
setA.symmetric_difference_update(setB)
print(setA)
# Выведет: {1, 2, 3, 6, 7, 8, 9}
```

Как видим множество `setA` изменилось. К нему прибавились элементы из множества `setB`, кроме тех, что принадлежат сразу обоим множествам (в нашем случае это элементы 4 и 5)

А если мы используем данный метод для получения нового множества `setC`, то такого множества мы не получим. Всё дело в окончании имени метода `_update`. Оно указывает на то, что первоначальное множество может изменяться после выполнении этого метода. А, если мы захотим использовать этот метод для получения нового множества `setC`, то никакого результата мы не получим (`None`). В таком случае нужно использовать такой же метод, но без `_update` на конце `setA.symmetric_difference(setB)`.

```
setA = {1, 2, 3, 4, 5}
setB = {4, 5, 6, 7, 8, 9}
setB.symmetric_difference_update(setA)
print(setA)
# Выдаст: {1, 2, 3, 6, 7, 8, 9}
print(setB)
```

```
# Выдаст: {1, 2, 3, 6, 7, 8, 9}
setA = {1,2,3,4,5,6,7}
setB = {6,7,8,9}
setC = setA.symmetric_difference_update(setB)
print(setC)
# Выведет: None
```

Методы множеств работающие с одиночными элементами данного множества

setA.add(elem) - добавляет элемент в множество

```
setA = {9,21,3,45,17}
elem = 100
setA.add(elem)
print(setA)
# Выведет: {17, 3, 100, 21, 9, 45}
```

setA.remove(elem) - удаляет элемент из множества

Выдаёт ошибку KeyError, если такого элемента не существует.

```
elem = 21
setA = {9,21,3,45,17}
setA.remove(elem)
print(setA)
# Выведет: {17, 3, 9, 45}
```

Во множестве элементы не упорядочены, поэтому после распечатки элементы множества могут поменять порядок. Элемент множества 21 был удалён.

Присвоим переменной значение 100, которого нет в исходном множестве. В этом случае будет ошибка.

```
elem = 100
setA = {9,21,3,45,17}
print(setA)
# Выведет: {17, 3, 21, 9, 45}
setA.remove(elem)
# Выведет: Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# KeyError: 100
```

setA.discard(elem) - удаляет элемент, если он находится в множестве

Если элемент в данном множестве нет, Ошибки не возникает и множество остаётся прежним.

```
setA = {9,21,3,45,17}
setA.discard(17)
print(setA)
# Выведет: {3, 21, 9, 45}
setA.discard(100)
print(setA)
# Выведет: {3, 21, 9, 45}
```

setA.pop() - удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.

```
setA= {9,21,3,45,17}
```

```
print(setA)
```

```
# Выведет: {17, 3, 21, 9, 45} Это показывает, что множество — объект неупорядоченный.
```

```
a = setA.pop()
```

```
print(a)
```

```
# Выведет: 17
```

```
print(setA)
```

```
# Выведет: {3, 21, 9, 45}
```

```
b = setA.pop()
```

```
print(b)
```

```
# Выведет: 3
```

```
print(setA)
```

```
# Выведет: {21, 9, 45}
```

setA.clear() - очистка множества

```
setA= {1,2,3,4,5}
```

```
print(setA)
```

```
# Выведет: {1, 2, 3, 4, 5}
```

```
setA.clear()
```

```
print(setA)
```

```
# Выведет: set()
```

Как видим после работы метода `clear()` множество очищается и выдаётся пустое множество `set()`.

frozenset

Единственное отличие `set` от `frozenset` заключается в том, что `set` - изменяемый тип данных, а `frozenset` - нет. Примерно похожая ситуация со списками и кортежами (списки могут изменяться, а кортежи нет).

```
a = set('qwerty')
```

```
print(a)
```

```
# Выведет: {'q', 'w', 'r', 'e', 't', 'y'}
```

```
b = frozenset('qwerty')
```

```
print(b)
```

```
#Выдаст: frozenset({'q', 'w', 'r', 'e', 't', 'y'})
```

```
print(a == b)
```

```
# Выдаст: True
```

```
print(a - b) # разность множеств
```

```
#Выведет: set() - пустое множество, так как множества a и b равны
```

```
type(a - b)
```

```
#Выдаст: <class 'set'>
```

```
print(a | b) # объединение множеств.
```

```
#Выведет: {'q', 'w', 'r', 'e', 't', 'y'}
```

```
type(a | b)
```

```
#Выведет: <class 'set'>
a.add(1)
print(a)
#Выведет: # {1, 'e', 'w', 'y', 'r', 't', 'q'}
b.add(1)
"""Выдаст: Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'"""
```

Важнейшим аспектом рассмотренных методов и функций, с помощью которых взаимодействуют множества между собой, является практический материал, который лежит в основе реляционных баз данных. Использование систем управления базами данных (СУБД) вместе с языком SQL (Structured Query Language - Язык Структурированных Запросов) для хранения и манипулирования данными, носит такой частый характер, что является неотъемлемой частью любого языка программирования, в том числе и Python. Взаимодействие Python с СУБД PostgreSQL и сама установка и использование её, будет рассмотрена на уроке по базам данных. Надо отметить, что язык SQL, не является алгоритмическим, то есть для получения нужного результата не нужно описывать процесс его получения (алгоритм), а всего лишь описать только цель, что нужно получить как в алгоритмическом языке, такой, например, как Python.

Урок 3 часть 3

Словари

Словарь ассоциативный массив. Состоит из ключей и связанных с ними объектами.

Что это значит?

Например, в списках нахождение элемента (или объекта) производится по индексу, то есть номеру позиции, на котором он располагается. В некоторых случаях это не совсем удобно. Нужно обязательно знать номер позиции объекта в списке, А это целое число, которое ничего не говорит о предмете, к которому относится данный объект. А нам нужно создать некую ассоциацию, то есть дать имя по которому мы можем найти тот объект, который нас интересует. Для этого и существуют словари.

Например, расписание самолётов. Каждый борт самолёта имеет в своём имени как буквы, так и цифры. И это отличается от целых чисел, используемые в списках для нахождения объектов списка. В рассматриваемом примере номер борта может служить ключом для нахождения объекта. Этот объект есть значение этого ключа. Таким образом словарь это неупорядоченная последовательность пар ключ:значение.

Рейс DP 6861 Москва – Саратов авиакомпании Победа

Ключ: 'DP 6861'

Значение: 'Москва, Шереметьево SVO, Саратов, Гагарин (Саратов) GSV, Вылет 14:20, Прилёт 16:30'

Рейс SU 1132 Москва – Сочи авиакомпании Аэрофлот

Ключ: 'SU 1132'

Значение: 'Москва, Шереметьево SVO, Сочи, Сочи AER, Вылет 14:36, Прилёт 17:53'

Ещё пример словаря.

Создадим пустой словарь Capitals - столицы

Capitals = dict()

Заполним его несколькими значениями. В квадратных скобках пишем ключ и присваиваем ключу новое значение:

Capitals['Россия'] = 'Москва'

Capitals['Белоруссия'] = 'Минск'

Capitals['США'] = 'Вашингтон'

print(Capitals)

Выведет: {'Россия': 'Москва', 'Белоруссия': 'Минск', 'США': 'Вашингтон'}

Как видим выводится словарь в виде последовательности ключ:значение через запятую, обрамлённой фигурными скобками.

Теперь создадим список стран:

Countries = ['Россия', 'Франция', 'США', 'Белоруссия']

Здесь используются квадратные скобки

Для каждой страны из списка проверим, есть ли она в словаре Capitals:

for country in Countries:

 if country in Capitals:

 print('Столица страны ' + country + ': ' + Capitals[country])

 else:

 print('В базе нет страны с названием ' + country)

Выведет следующие данные:

#Столица страны Россия: Москва

#В базе нет страны с названием Франция

#Столица страны США: Вашингтон

#Столица страны Белоруссия: Минск

В нашем примере ключом является название страны, взятое из списка Countries (страны), название столицы его значением. Значения ключей — уникальны, двух одинаковых ключей в словаре быть не может.

В жизни широко распространены словари, например, привычные бумажные словари. Например, иностранные словари. В них ключом является слово или выражение, а значением — перевод этого слова или выражения на другой язык. Для того, чтобы получить доступ к переводу, необходимо указать ключ - слово. Телефонный справочник также относится к словарям. В нем ключом является имя, а значение — номер телефона.

Создание словарей

Через функцию

```
dict1 = dict()
```

Функция dict() создаёт словарь, в данном случае пустой.

Через фигурные скобки

```
dict2 = {}
```

Определяем тип, созданных переменных

```
print(type(dict1))
```

```
# Выведет: <class 'dict'>
```

```
print(type(dict2))
```

```
# Выведет: <class 'dict'>
```

Как видим обе созданные переменные относятся к классу словарей (<class 'dict'>)

Создадим уже заполненный словарь через функцию и фигурные скобки

Через функцию:

```
dict1 = dict({'car': 'машина', 'apple': 'яблоко'})
```

Через фигурные скобки:

```
dict2 = {  
    'car': 'машина',  
    'apple': 'яблоко'  
}
```

Изменение словаря

Пусть дан словарь dict с двумя ключами и привязанными к ним значениями.

```
dict = {'car': 'машина', 'apple': 'яблоко'}
```

Изменим словарь, добавив к нему новый ключ со своим значением:

```
dict['orange'] = 'апельсин'
```

В квадратных скобках указывается имя ключа, а значение после знака равенства =.

Для удаления ключа и его объекта используют имя ключа в квадратных скобках

```
dict = {'car': 'машина', 'apple': 'яблоко', 'orange': 'апельсин'}
```

```
print(dict)
```

```
# Выведет: {'car': 'машина', 'apple': 'яблоко', 'orange': 'апельсин'}
```

Теперь удалим один из ключей:

```
del dict['orange']
```

```
print(dict)
```

```
# Выведет: {'car': 'машина', 'apple': 'яблоко'}
```

Перебор элементов словарей

```
dict = {'car': 'машина', 'apple': 'яблоко', 'orange': 'апельсин'}
```

```
for key in dict:
```

```
    print(key, dict[key])
```

```
# Выдаст:
```

```
"""
```

```
car машина
```

```
apple яблоко
```

```
orange апельсин
```

```
"""
```

Для вывода значения по ключу используются имя словаря, а в квадратных скобках имена нужного ключа:

```
dict = {
    'car': 'машина',
    'apple': 'яблоко',
    'orange': 'апельсин'
}
print(dict['car'])
# Выведет: машина
```

Методы словарей (функции словаря как объекта)

copy() - создание копии словаря

```
dict1 = {
    'car': 'машина',
    'apple': 'яблоко',
    'orange': 'апельсин'
}
dict2 = dict1.copy()
print(dict2)
# Выдаст: {'car': 'машина', 'apple': 'яблоко', 'orange': 'апельсин'}
```

get — получение значения по ключу

```
dict = {
    'car': 'машина',
    'apple': 'яблоко',
    'orange': 'апельсин'
}
print(dict.get('car'))
# Выдаст: 'машина'
```

Тоже самое, когда мы обращаемся непосредственно к ключу словаря:

```
print(dict['car'])
# Выдаст: 'машина'
```

clear — очистка словаря

```
dict = {
    'car': 'машина',
    'apple': 'яблоко',
    'orange': 'апельсин'
}
dict.clear()
print(dict)
# Выведет: {}
```

Видно, что словарь опустел. Применяется, когда нам нужна данная переменная для дальнейшего использования, заполняя пустой словарь нужными элементами (ключ:значение).

keys() - получение всех ключей

```
dict = {  
    'car': 'машина',  
    'apple': 'яблоко',  
    'orange': 'апельсин'  
}  
print(dict.keys())  
# Выдаст: dict_keys(['car', 'apple', 'orange'])
```

А теперь мы хотим поработать с этими ключами, превратив их в список с помощью функции list()

```
list1 = list(dict.keys())  
print(list1)  
# Выдаст список ключей: ['car', 'apple', 'orange']
```

values() - метод получения не ключей словаря, а всех их значений

```
dict = {  
    'car': 'машина',  
    'apple': 'яблоко',  
    'orange': 'апельсин'  
}  
print(dict.values())  
# Выдаст список значений: dict_values(['машина', 'яблоко', 'апельсин'])
```

items() - получение всех элементов словаря

```
dict = {  
    'car': 'машина',  
    'apple': 'яблоко',  
    'orange': 'апельсин'  
}  
print(dict.items())  
# Выведет : dict_items([('car', 'машина'), ('apple', 'яблоко'), ('orange', 'апельсин')])
```

Похоже на список : квадратные скобки указывают, что это список, а круглые скобки, обрамляющие каждый элемент списка — кортеж из двух значений.

Превратим полученный результат в список с помощью функции list():

```
list1 = list(dict.items())  
print(list1)  
# Выведет : [('car', 'машина'), ('apple', 'яблоко'), ('orange', 'апельсин')]
```

pop() - удаляет и возвращает значение ключа

То есть выдаст значение по указанному ключу и удалит указанный ключ со значением из словаря.

```
dict = {  
    'car': 'машина',  
    'apple': 'яблоко',  
    'orange': 'апельсин'  
}  
val = dict.pop('car')
```

```
print(val)
# Выдаст : машина
print(dict)
# Выдаст : {'apple': 'яблоко', 'orange': 'апельсин'}
```

Как видим указанный ключ удалился вместе со значением из словаря, присвоив переменной val значение обозначенного ключа в методе pop()

update() - обновить значение по ключу и добавляет новые ключи

```
dict = {
    'car': 'машина',
    'apple': 'яблоко',
    'orange': 'апельсин'
}
dict.update({'car': 'автомобиль', 'home': 'дом'})
print(dict)
# Выдаст : dict = {'car': 'автомобиль', 'apple': 'яблоко', 'orange': 'апельсин', 'home': 'дом'}
```

Здесь надо обратить внимание на то, что при использовании метода update(), если ключ существует в словаре, то его значение поменяется на новое, указанное в этом методе. Если же ключа не было, то в словарь добавится указанный ключ со своим значением.

А теперь вернёмся к нашему примеру со спортивными секциями.

Мы помним, что получили список, тех членов секции лыж, которые не посещают лыжную секцию, но потенциально могут участвовать в предстоящих соревнованиях и дополнительно тренироваться. Список то мы получили, но нужно их ещё обзвонить и уточнить будут ли они участвовать в соревнованиях или нет. Для этого необходимо знать телефон каждого участника этого списка. Пойдём с самого начала.

Пусть вся информация об участниках спортивных секций находится в базе данных. Там имеются фио (фамилия, имя, отчество), телефон, адрес, год рождения, секция и другая информация. Нам необходимо вытащить из базы хотя бы телефон каждого участника с привязкой к фио (фамилия, имя отчество). В Python существует уже предустановленная, начиная с 3 версии, легковесная однофайловая система управления базами данных (СУБД) SQLite. На ней можно опробовать свою программу взаимодействия Python с базой данных. Лучше, конечно, установить полноценную, промышленную СУБД типа PostgreSQL. Установка и работа с СУБД PostgreSQL описана на уроке 9.

Рассмотрим маленький пример:

```
import sqlite3
cars = [
    ('Audi', 52642),
    ('Mercedes', 57127),
    ('Skoda', 9000),
    ('Volvo', 29000),
    ('Bentley', 350000)]
```

```
con = sqlite3.connect("cars.db")
cur = con.cursor()
```

```
cur.executescript("""CREATE TABLE IF NOT EXISTS cars (
    car_id INTEGER PRIMARY KEY AUTOINCREMENT,
    model TEXT,
    price INTEGER)
""")
```

```
cur.executemany("INSERT INTO cars VALUES(NULL,?, ?)", cars)
```

```
cur.execute("SELECT model, price FROM cars")
```

```
rows = cur.fetchall()
```

```
print(rows)
```

```
# [('Audi', 52642), ('Mercedes', 57127), ('Skoda', 9000), ('Volvo', 29000), ('Bentley',
350000)]
```

Сначала импортируем модуль база данных sqlite3 третьей версии `import sqlite3`
Затем формируем список, содержащий кортежи из двух элементов — название автомобиля и цена.

```
cars = [
    ('Audi', 52642),
    ('Mercedes', 57127),
    ('Skoda', 9000),
    ('Volvo', 29000),
    ('Bentley', 350000)
```

```
]
```

Коннектимся (подсоединяемся) к базе cars.db.

Если базы данных cars.db не существовало, то она будет создана и произойдёт подключение к этой базе.

```
con = sqlite3.connect("cars.db")
```

Присваиваем переменной car значение текущего курсора данного соединения:

```
cur = con.cursor()
```

Далее создаётся таблица

```
cur.executescript("""CREATE TABLE IF NOT EXISTS cars (
    car_id INTEGER PRIMARY KEY AUTOINCREMENT,
    model TEXT,
    price INTEGER)
""")
```

Метод курсора `executescript` позволяет выполнить в одной строке (обрамлённая `""")` сразу нескольких команд SQL, Эти команды отделяются символом `;`. В данном случае выполняется только одна команда. Можно было бы использовать обычный метод курсора `execute`.

Команда `CREATE TABLE` создаёт таблицу с именем cars, если такая таблица ещё не существовала. Существование таблицы с таким именем проверяется записью `IF NOT EXISTS` (если не существует). Положительный ответ, то есть `True` позволяет создать эту таблицу. В ней описаны первичный ключ `car_id`. Он имеет `INTEGER`,

то есть целое число. **PRIMARY KEY** указывает, что это первичный ключ. Другими словами - это имя поля таблицы, которое однозначно определяет каждую строку в этой таблице. В первичном ключе не будет никогда повторов. Если, даже, какая-нибудь строка удаляется, то новая строка, которая будет вставлена, будет иметь значение больше на 1, чем значение ключа, последней существующей записи. За это отвечает аргумент первичного ключа **AUTOINCREMENT** (автоинкремент: автоматическое увеличение значения первичного ключа на 1). Далее через запятую перечислены имена полей (или столбцов создаваемой таблицы).

model TEXT - марка модели автомобиля, имеющий текстовый тип большой длины TEXT.

price INTEGER - цена автомобиля. Тип INTEGER, то есть целое число. Значит цена не должна иметь дробной части.

Команда `cur.executemany("INSERT INTO cars VALUES(NULL,?, ?)", cars)` позволяет нам одним махом вставить содержимое списка cars в таблицу cars. Используя метод `execute`, нам бы пришлось каждую строку списка cars вставлять отдельным оператором **INSERT** (вставить). Или составить цикл `for` по вставке каждого элемента списка в таблицу.

Вместо знаков вопроса (?) будут подставлены текущие значения из списка cars в столбцы под именем model и price таблицы cars. То, что список cars и таблица cars, путаницы не будет.

Первый атрибут запроса **CREATE TABLE** указывается как **NULL**, означает что не присвоено ему никакого значения, а **AUTOINCREMENT** присвоит текущей вставляемой строке целое значение, каждый раз увеличенное на 1.

`cur.execute("SELECT model, price FROM cars")` оператор SQL **SELECT** делает выборку нужных столбцов из (FROM) нашей таблицы cars.

`rows = cur.fetchall()` - метод курсора `fetch` (принести), а `fetchall` (принести всё), присваивает переменной rows все значения из таблицы cars.

`print(rows)` - выполняет распечатку всех строк таблицы cars по тем столбцам, что перечислены в запросе **SELECT**.

Для нашего примера со спортсменами создадим базу данных sport.db и таблицу sports_section, в которой собраны спортсмены по всем секциям.

Полный список типов полей при создании таблицы в SQLite:

- **NULL** – значение NULL;
- **INTEGER** – целочисленный тип;
- **REAL** – вещественный тип;
- **TEXT** – строковый тип;
- **BLOB** (двоичные данные, хранятся «как есть», например, для небольших изображений).

lesson3_part_3_1.ru (Тексты программ и данных располагаются в zip файле по адресу: pageforweb.ru/lessons_py.zip)

```
import sqlite3
```

```
con = sqlite3.connect("sport.db")
```

```

cur = con.cursor()
cur.execute("""DROP TABLE sports_section;""")
cur.execute("""CREATE TABLE IF NOT EXISTS sports_section (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    fio TEXT,
    telephon TEXT,
    adress TEXT,
    year_of_birth INTEGER,
    section INTEGER,
    absent boolean);""")
cur.execute("""INSERT INTO sports_section (fio, telephon, adress, year_of_birth,
section, absent) VALUES ('Сидорова Маргарита Степановна', '+7 (495) 517-41-80', '',
2012, 1, True), ('Николаев Олег Евгеньевич', '+7 (495) 620-30-21', '', 2009, 1, False),
('Добода Семён Маркович', '+7 (495) 620-30-37', '', 2013, 1, False), ('Сузанов Алексей
Ильич', '+7 (495) 725-70-00', '', 2011, 1, False), ('Тропинин Василий Альбертович', '+7
(495) 725-71-52', '', 2005, 1, True), ('Крюкова Наталья Сергеевна', '+7 (495) 926-40-23',
'', 2005, 1, False), ('Тужилин Прокофий Сидорович', '+7 (495) 926-44-58', '', 2007, 1,
False), ('Настенко Сергей Павлович', '+7 (495) 926-44-80', '', 2006, 1, False),
('Кондратенко Игорь Николаевич', '+7 (495) 926-44-97', '', 2009, 1, False), ('Прохоров
Ростислав Георгиевич', '+7 (495) 941-27-71', '', 2010, 1, False), ('Стариков Михаил
Владимирович', '+7 (495) 941-35-53', '', 2005, 1, False), ('Сергеев Константин
Петрович', '+7 (499) 130-81-25', '', 2010, 1, False), ('Любомиров Николай
Михайлович', '+7 (499) 143-12-88', '', 2005, 1, False), ('Дразденко Илья Ильич', '+7
(916) 157-50-86', '', 2007, 1, False), ('Пронин Марк Анатольевич', '+7 (916) 157-50-86',
'', 2010, 2, False), ('Дразденко Илья Ильич', '+7-906-095-19-35', '', 2008, 2, False),
('Лихачёв Эдуарт Трофимович', '+7 (495) 607-12-22', '', 2009, 2, True), ('Гордеев
Василий Васильевич', '+7 (495) 268-06-08', '', 2011, 3, False), ('Кондратенко Игорь
Николаевич', '+7 (495) 926-44-97', '', 2012, 3, False), ('Светланов Руслан Григорьевич',
'+7 (499) 550-15-08', '', 2005, 3, False), ('Сергеев Константин Петрович', '+7 (499) 130-
81-25', '', 2009, 3, False), ('Лежебоков Георгий Алексеевич', '+7 (925) 416-90-06', '',
2009, 3, False), ('Прокофьев Леонид Петрович', '+7 (901) 546-42-22', '', 2006, 3, False),
('Тропинин Василий Альбертович', '+7 (495) 725-71-52', '', 2010, 3, True);""")
cur.execute("SELECT * FROM sports_section;")
result = cur.fetchall()
for row in result:
    print(row)

```

Можно сделать так, что данные будут загружаться не из программы, а из текстового файла. Создадим для этого новую таблицу с суффиксом _1 (sports_section_1). Текстовый файл с данными для загрузки в таблицу

[sports_section_1.txt](#).

[sports_section_1.py](#)

```
import sqlite3
```

```
con = sqlite3.connect("sport.db")
```

```
cur = con.cursor()
```

```

cur.execute("""DROP TABLE sports_section_1;""")
cur.execute("""CREATE TABLE IF NOT EXISTS sports_section_1 (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    fio TEXT,
    telephon TEXT,
    adress TEXT,
    year_of_birth INTEGER,
    section INTEGER,
    absent boolean);""")
# Проверим наличие данных:
cur.execute("SELECT * FROM sports_section_1;")
result = cur.fetchall()
for row in result:
    print(row)
# Удалим данные из таблицы, если они есть, удаляем
if result != []:
    cur.execute("delete from sports_section_1;")
#Открываем текстовый файл sports_section_1.txt на чтение с присвоением
переменной file_sports_section_1
Это текстовая переменная, которая имеет на конце символ перевода строки,
который записывается так '\n'. Этот символ нужно удалить. Текстовую
переменную изменять нельзя. Поэтому получаем новую строку без этого
символа путём метода строк line_value.replace('\n', ''), то есть заменяем этот
символ на пустой символ '' (одинарные кавычки).
file_sports_section_1 = open('sports_section_1.txt', 'r')
line_value = file_sports_section_1.read()
line_value_1 = line_value.replace('\n', '')
# Пишем загрузку данных из переменной line_value_1 в таблицу sports_section_1,
используя функцию f.
cur.execute(f""""INSERT INTO sports_section_1 (fio, telephon, adress, year_of_birth,
section, absent) VALUES {line_value_1};""")
# И снова проверяем
cur.execute("SELECT * FROM sports_section_1;")
result = cur.fetchall()
for row in result:
    print(row)

```

Данные загрузились в базу данных в таблицу sports_section_1.

Текстовый файл должен находится в той же директории, что база данных и запускать python3 необходимо из этой директории. Для перехода можно использовать оператор cd <рабочая директория>.

Нужно сделать 4 запроса к базе и получить списки А, В, С, Е.

Запросы выборки данных из базы описывают результат, который нужно получить. select (выбрать) такие-то столбцы from (из) такой-то таблицы, а where (где) описывает условия выборки строк.

1. В результате данного запроса получим всех, кто записан в секцию лыж в виде списка.

```
cur.execute("""select fio ,telephon from sports_section_1 where section = 1;""")
A = cur.fetchall()
print(A)
```

Проверим, точно ли результат запроса имеет тип списка:

```
print(type(A))
# Выдаст: <class 'list'> (значит это список)
```

Видим, что это действительно список (обрамлено []). А, вот, элементы списка составляют кортежи из двух значений — ФИО и телефон. Кортежи обрамлены круглыми скобками.

2. Запрос на всех самбостов

```
cur.execute("""select fio, telephon from sports_section_1 where section = 2;""")
B = cur.fetchall()
print(B)
```

3. Здесь запрашиваются все пловцы

```
cur.execute("""select fio, telephon from sports_section_1 where section = 3;""")
C = cur.fetchall()
print(C)
```

4. В этом запросе запрашиваются все члены лыжной секции, которые не посещают занятия.

```
cur.execute("""select fio, telephon from sports_section_1 where absent = True;""")
E = cur.fetchall()
print(E)
```

(absent — отсутствие)

Все результаты запросов представлены в виде списков.

Полученные списки можно далее использовать для распечатки, перевода в словарь и других действий.

(А - лыжи, В - самбо, С - плавание) и список не посещающих секцию лыж (Е). Создадим функцию получения списка только с ФИО.

```
def fio(list_in):
    list_out = []
    for row in list_in:
        elem = row[0]
        list_out.append(elem)
    return list_out
```

Подставим списки с результатами запросов, используя эту функцию.

```
A1 = fio(A)
B1 = fio(B)
C1 = fio(C)
E1 = fio(E)
```

Как можно воспользоваться таким результатом?

Переведём списки во множества.

```
setA1 = set(A1)
```

```
print(setA1)
setB1 = set(B1)
print(setB1)
setC1 = set(C1)
print(setC1)
setE1 = set(E1)
print(setE1)
```

Видим, что квадратные скобки, характерные для списка, заменились на фигурные, применяемые во множествах.

Далее, основываясь на действия над множествами, описанные в уроке python3_2, получим объединённое множество областей 5, 6, 7. Это и будут фамилии, имена и отчества тех спортсменов лыжников не посещающих тренировки, с которыми будет связываться тренер по лыжам, сообщая им о предстоящих соревнованиях. Так как они скорее всего смогут приходить на дополнительные занятия.

Но как же тренер сможет связаться с указанными членами лыжной секции, если у него в наличии только фамилия, имена и отчества?

Поможет перевод множества всех лыжников в словарь.

```
dictA1 = dict(setA1)
# Проверяем:
print(type(dictA1))
# Выдаст : <class 'dict'>
```

Теперь мы можем легко получить нужный нам номер телефона по фамилии, имени и отчеству.

Например:

```
print(dictA1['Кондратенко Игорь Николаевич'])
# Выведет : +7 (495) 926-44-97
```

Во всех результатах мы увидим множества, обрамлённые фигурными скобками. Посмотрим, входят ли одни множества в другие.

```
setB1 in setA1
setC1 in setA1
setE1 in setA1
setE1 in setB1
setE in setC1
```

Все результаты дают False. Это значит, что все множества либо могут пересекаться, либо вообще не пересекаются, но нет ни одного множества, которые полностью принадлежат другому множеству.

Определим пересечение множеств относительно множества лыжников (множество setA):

```
setA1.isdisjoint(setB1)
# Выведет: True
setA1.isdisjoint(setC1)
# Выдаст: False
setA1.isdisjoint(setE1)
```

```
# Выведет: True
```

```
setA1.intersection(setC1)
```

```
# Выведет: {'Сергеев Константин Петрович', '+7 (499) 130-81-25'}, ('Кондратенко  
Игорь Николаевич', '+7 (495) 926-44-97'), ('Тропинин Василий Альбертович', '+7  
(495) 725-71-52')}
```

```
setA1.intersection(setB1)
```

```
setA1.intersection(setC1)
```

```
setA1.intersection(setE1)
```

```
setA1.intersection(setC1)
```

Относительно пересечения множеств setA и setE другого и быть не может, так как множество setE относится к множеству setA — это те люди, которые записались в лыжную секцию, но не посещают.

Найти телефон по ФИО на локальном компьютере с помощью множества достаточно сложно. Можно запросом из базы данных, но в реальности база данных находится на сервере, а не на локальной машине. Остаётся только перевести наше множество в словарь.

```
dictE = dict(setE)
```

```
print(dictE)
```

```
telephon = dictE['Кондратенко Игорь Николаевич']
```

```
print(telephon)
```

1. Находим пересечение множеств лыжников setA и самбистов setB. Назовём их пересечение setAminussetB. Смотрим, какие зоны входят в это пересечение.

Видим, это зоны 5, 3, 6.

```
setAminussetB = setA.difference(setB)
```

2. Находим пересечение множеств лыжников setA и пловцов setC. Назовёт setAminussetC.

Сюда входят зоны 3, 4, 6, 7.

```
setAminussetC = setA.difference(setC)
```

3. Находим пересечение трёх множеств лыжников setA, самбистов setB и пловцов setC. Это зона D или множество setAminussetBminussetC. Сюда входят зоны 3, 6.

```
setD.intersection_update(setA, setB, setC) # пересечение
```

4. Теперь берём множество setAminussetB, полученное в результате пересечения setA и setB, и находим пересечение с множеством setE. Получаем множество, которое мы обозначим setAminussetBminussetE. Сюда войдут зоны 3, 4.

```
setAminussetBminussetE = setA.difference(setB, setE)
```

5. Находим множество set5 (зона 5), получаемое путём вычитания из множества setAminussetB множества setD и множества setE. Это первое нужное множество людей, с которыми тренер лыжников будет связываться

```
set5 = setA.difference(setB) - setD - setE
```

6. Повторяем пункты 4 и 5, но уже для множества set7.

Из множества setAminussetC вычитаем множество setE.

```
set6 = setAminussetC - setE
```

7. Получение множества set6. Это разность множества setD и множества setE.

```
set6 = setD - setE
```

8. Находим объединение полученных множеств set5, set6, set7.

```
setReult = set()
```

```
setReult.update(set5, set6, set7) #- объединение.
```

Найденное множество и есть тот самый список людей, с кем необходимо связаться тренеру по лыжам.

Урок 4 часть 1

Функции и их аргументы

Именные функции, инструкция def

Функция в Python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции def.

Определим простейшую функцию:

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

x, y — аргументы, поступающие на вход функции add (сложить).

Всё тело функции (в данном случае только одна строка return x + y) пишется с отступом нажатием клавиши горизонтальной табуляции или кратное количество пробелов (2, 4, 8).

Инструкция return говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать, подставляя конкретные аргументы:

```
add(1, 10)  
# Возвратит: 11
```

Могут быть сложены не только числа, но и строки

```
add('Сейчас ', 'светло')  
# Возвратит: 'Сейчас светло'
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

```
def newfunc(n):  
    def myfunc(x, y):  
        c = x + y  
        return c * n  
    return myfunc
```

```
new = newfunc(100) # new - это функция  
new(2, 10)  
# Вернёт: 1200
```

Складываются два числа x и y (2 + 10 = 12)

Затем этот результат умножается на n (100).

В описании функции newfunc(n) присутствует другая функция def myfunc(x,y). Нужно обратить внимание, что при любом описании функции, где бы она не находилась, внутри или снаружи, после названия функции, тело этой функции идёт с отступом, относительно её названия.

Запускаем внешнюю функцию newfunc(100) с аргументом 100 (это n в описании внешней функции). На выходе должна получиться функция. Даём название этой функции new = newfunc(100). Функция new нам заменяет, описанную внутреннюю myfunc(x, y) .

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

```
def func():  
    pass  
print(func())  
# Вернёт: None
```

Оператор pass часто используют, когда много взаимно связанных функций и нужно, чтобы не забыть мы даём только название функций. При этом хорошо бы после названия функций расписать в тройных кавычках, что же будет делать данная функция. Это полезно проделывать и когда пишется не пустая функция. Примеры функций.

Функция получения конечной цены продукта с учётом скидок:

Формула для расчета цены со скидкой проста:

Цена со скидкой = Цена * (100 - Скидка) / 100

Напишем функцию:

```
def discounted_price(price, discount):  
    """  
    Функция получения конечной цены с учётом скидки, которая задаётся в  
    процентах.  
    """  
    dp = price*(100 - discount)/100  
    return dp  
print(discounted_price(100, 25))  
# Выдаст: 75.0
```

Пример вставки пробелов в числа (Программа находится в файле spaces.py).

Часто бывает необходимо выводить числа, отделяя триады числа пробелами при выводе для большей наглядности. Например в бухгалтерии, где суммы могут иметь большое количество цифр. Число 76549098.78 плохо воспринимается, где миллиарды, где миллионы, где тысячи. А, если отделить их пробелами, картина радикально поменяется в лучшую сторону читабельности этого числа (76 549 098.78). Надо понимать, что входной аргумент 76549098.78 имеет тип числа float (вещественный) или int (целый), а выходной параметр этой функции будет уже иметь строковый тип — str.

```
spaces_in_numbers.py  
def spaces_in_numbers():  
    """
```

Функция преобразования числа в строку с разделением триад пробелами.

```
'''
num_input = input('Введите число: ')
# Функция input() обеспечивает ввод пользователем как числа так и
строки. Но вводя числа она преобразует их в строковый тип. Это надо
учитывать.
num_array = num_input.split('.')
''' Разложим полученное число на составляющие до точки (целая часть) и
после точки (дробная часть) с помощью метода split('.'), указав в качестве
разделителя точку. У нас получится список из двух элементов: целой части и
дробной. '''
num_str = num_array[0]
# Обозначим целую часть (в этом списке она под индексом 0) num_str.
if len(num_array) == 2:
    num_ost = num_array[1]
''' Делаем проверку — существует ли дробная часть, то есть количество
элементов = 2. И, если так, присваиваем значение второго элемента списка (под
индексом 1) переменной num_ost. '''
num_len = len(num_str)
''' Определим теперь размер (количество цифр) в целой части числа.
Обозначим его num_len. '''
itog = []
# Создадим теперь пустой список itog
if num_len % 3 == 1:
    itog.append(num_str[0:1])
    itog.append(' ')
    srez = num_str[1:num_len]
''' Если остаток от деления на 3 равно 1, то есть цифры в числе
укладываются в триады за исключением одной цифры. Добавляем в пустой
список itog срез num_str[0:1], означающее цифру, которая имеет индекс 0. Затем
добавляем символ пробела и получаем срез srez со второго индекса (он у нас 1)
до num_len, означающий количество цифр в символьной переменной целой
части введенного числа. '''
if num_len % 3 == 2:
    itog.append(num_str[0:2])
    itog.append(' ')
    srez = num_str[2:num_len]
''' Такая же ситуация, но теперь для остатка 2. Берём уже больший срез в
две цифры, добавляем в itog, и записываем пробел. И уже от третьего индекса
(2) до конца символьного отображения числа получаем срез srez. '''
if num_len % 3 == 0:
    srez = num_str
# Если ни какого остатка нет(0), то srez делаем равным num_str.
for i in range(len(srez)):
```

```

        itog.append(srez[i])
        if (i + 1) % 3 == 0:
            itog.append(' ')
        """ Далее идёт цикл for по всему срезу srez. В itog добавляется очередной
        символ цифры, и, если остаток (индекса + 1) на 3 будет 0, это означает, что мы
        получили очередную триаду и можем вставить в itog пробел. Пробел поставится
        и после последней триады. Поэтому его нужно уничтожить. """
        del itog[len(itog) - 1]
        # Уничтожение лишнего пробела после последней триады.
        itog_str = "".join(itog)
        # Список из символов цифр itog соединяем в строку, используя пробел как
        разделитель.
        if len(num_array) == 2:
            itog_str += '.'
        # Если была дробная часть числа (список num_array состоит из двух
        частей), добавляем точку
        itog_str += num_ost
        # и num_ost дробную часть.
        return itog_str
Вызываем функцию spaces_in_numbers():
spaces_in_numbers()

```

В функциях распространены следующие варианты аргументов

1. Функции с позиционными аргументами

У функций, которые мы изучали в самом начале, были в основном позиционные аргументы. Когда они вызываются, аргументы в них подставляются согласно их позиции в определении функции. Например:

```

def add(x, y):
    return x + y

```

В этом случае с аргументами (10, 20) аргумент x получит значение 10, а y - 20.

Правда, здесь, от перемены мест слагаемых сумма не меняется и можно запросто записать аргументы в последовательности (20, 10). Но, если напишем функцию возведения в степень, то при перемене мест аргументов результат будет разным.

```

def exponentiation(x, y):
    return x**y # возведение в степень
exponentiation(5, 2)
# Выдаст: 25
exponentiation(2, 5)
# Выдаст: 32

```

2. Функции с именованными аргументами

При вызове функции с позиционными аргументами важную роль играет порядок размещения передаваемых аргументов. Но если мы хотим изменить порядок, то нужно использовать именованные аргументы.

Они передаются не просто как значения, а как пара «имя=значение». Поэтому их можно передавать в любом порядке:

```
def truncate(text='MyText', length=3): # Усекать
    srez = text[0:length]
    return srez
```

```
print(truncate(text='MyText', length=3)) # Вызов функции truncate со значениями
по умолчанию. Аргументы переданы в одном порядке
```

```
# Выдаст: MyT
```

```
# Изменим значение одного из аргументов
```

```
print(truncate(text='MyText', length=5)) #
```

```
# Выдаст: MyTex
```

```
# Изменим значение другого аргумента
```

```
print(truncate(text='Программа', length=5))
```

```
# Выдаст: Progr
```

А теперь поменяем местами аргументы

```
print(truncate(length=5, text='Программа'))
```

```
# Выдаст такой же результат, что и в предыдущем случае: Progr
```

Есть две причины использовать именованные аргументы:

- Они повышают читаемость, так как сразу видны имена
- Можно не указывать все промежуточные параметры, которые нам сейчас не нужны

Последнее полезно, если у функции много необязательных параметров.

Посмотрим на примере:

```
def print_params(a=1, b=2, c=None, d=4):
    print(a, b, c, d)
```

Именованные аргументы позволяют передавать не все значения аргументов, а только, например, d. Для остальных аргументов используются значения по умолчанию.

```
print_params(d=8)
```

```
# Выдаст: 1 2 None 8
```

Именованные аргументы можно передавать одновременно с позиционными.

Тогда позиционные должны идти в самом начале.

Передадим позиционно только a и b. c не передадим. d передадим как именованный:

```
print_params(3, 4, d=3)
```

```
# Выдаст: 3 4 None 3
```

Хотя c мы не передавали, однако его значение передалось по умолчанию.

3. Функции с произвольным числом аргументов

Не всегда заранее известно, сколько аргументов будет передано функции. В таких случаях в Python имеются `*args` и `**kwargs`, позволяющие передавать переменное количество аргументов.

`*args` подразумевает один или несколько неименованных аргументов. `*` перед именем аргумента показывает нам, что число аргументов, передаваемых как кортеж, может быть любым и доступны внутри функции под тем же именем, что и имя параметра, только без `*`. Сначала запишем функцию сложения 3 значений.

```
def adder(x, y, z):  
    print("sum: ", x + y + z)
```

Выполним данную функцию:

```
adder(5, 7, 10)
```

```
# Выведет: sum: 22
```

Если же мы передадим этой функции не 3 аргумента, а больше, то получим ошибку:

```
adder(5, 10, 15, 20, 25)
```

```
#File "<stdin>", line 1, in <module>
```

```
#TypeError: adder() takes 3 positional arguments but 5 were given
```

Видно, что вместо 3 позиций аргументов передано 5.

А теперь воспользуемся `*`.

```
def adder(*nums):  
    sum = 0  
    for n in nums:  
        sum += n
```

```
    print("sum: ", sum)
```

Используем ту же функцию сложения чисел с разным количеством аргументов:

```
adder(3, 5)
```

```
adder(4, 5, 6, 7)
```

```
adder(1, 2, 3, 5, 6)
```

Ошибок не будет и будет выведено соответственно:

```
sum: 8
```

```
sum: 22
```

```
sum: 17
```

Это были не именованные аргументы (`*args`) с произвольным количеством и передаются они в виде кортежа.

А теперь, именованные аргументы (`**kwargs`) с произвольным количеством. Они передаются в качестве словаря. В словаре, как известно, имеются ключи, используемые как имена к значениям переменных.

```
def intro(**data):  
    for key in data:  
        print(key, data[key])
```

```
intro(Firstname="Прохор", Lastname="Пупкин", Age=22, Phone=1234567890)
```

```
# Выведет:
```

```
Firstname Прохор
Lastname Пупкин
Age 22
Phone 1234567890
```

```
intro(Firstname="Иван", Lastname="Лансберг", Email="ivanlansberg@mail.ru",
Country="Россия", Age=25, Phone=9876543210)
```

```
# Выведет:
```

```
Firstname Иван
Lastname Лансберг
Email ivanlansberg@mail.ru
Country Россия
Age 25
```

Функции с обязательными аргументами - это, как правило, позиционные аргументы. Их обязательно вводят когда запускается функция.

Необязательными аргументами можно пренебречь при вызове функции, так как в описании функции они присутствуют уже с присвоенными значениями аргументов. И если они не указаны в вызове функции, ничего страшного. Им будет присвоено значение по умолчанию, которое присвоено при описании функции.

Разберем более сложную передачу аргументов функции, основанную на смешении рассмотренных нами вариантов.

Произвольное количество позиционных аргументов можно получить в виде `*args`. На выходе получим кортеж.

```
def tuple1(*args):
    return args
```

```
a = tuple1(1, 2, 3)
print(a)
# Выведет: (1, 2, 3)
print(type(a))
# Выведет: <class 'tuple'>
```

Произвольное количество именованных аргументов `**kwargs`.

На выходе они получаются в виде словаря, что позволяет сохранить имена аргументов в ключах:

```
def dict1(**kwargs):
    return kwargs
b = dict1(x=1, y=2, z=None)
print(b)
# Выведет: {'x': 1, 'y': 2, 'z': None}
print(type(b))
# Выведет: <class 'dict'>
```

По соглашению аргумент, который получает подобный словарь, принято называть `kwargs` — от словосочетания `keyword arguments` (аргументы ключевого слова).

Урок 4 часть 2

Порядок вызовов смешанных аргументов

Рассмотрим разные комбинации аргументов:

```
def func(*args, **kwargs):  
    return (args, kwargs)  
# Вызываем функцию с комбинированными аргументами.  
func(1, 2, 3, 4, kx='a', ky='b', kz='c')  
# Выведет: ((1, 2, 3, 4), {'kx': 'a', 'ky': 'b', 'kz': 'c'})
```

И так, если в описании функции стоит `**kwargs` (вторая часть аргументов), а при вызове функции имеются именованные аргументы, на выходе получается словарь. Не именованные аргументы будут относиться к `*args`, и сколько бы значений мы не ввели они будут представлять собой кортеж всех этих значений (первая часть аргументов).

Заметим, что `*args` всегда указывается перед `**kwargs`, иначе будет ошибка:

```
def func(**kwargs, *args):  
    return (kwargs, args)  
func(1, kx='a') # Выведет: SyntaxError: invalid syntax
```

Синтаксическая ошибка возникает даже на этапе описания функции с неправильным порядком следования аргументов `*args` и `**kwargs`.

Можно так же комбинировать позиционные аргументы, значения по умолчанию, `*args` и `**kwargs` одновременно. При использовании обычных позиционных аргументов их следует добавлять в начало перед аргументом `*args`:

```
def func(x, *args, **kwargs):  
    return (x, args, kwargs)  
func(1, 2, 3, kx='a', ky='b')  
# Выведет: (1, (2, 3), {'kx': 'a', 'ky': 'b'})
```

Аргументы со значением по умолчанию следует добавлять после аргумента `*args`, но перед аргументом `**kwargs`:

```
def func(*args, kk=42, **kwargs):  
    return (args, kk, kwargs)
```

Видим, что в описании функции идут сначала аргументы с произвольным количеством не именованных `*args` (превратятся при вызове функции в кортеж), затем в описании идёт именованный параметр со своим значением `kk=42`, замыкает аргументы с произвольным количеством именованных аргументов `**kwargs` (При вызове функции превратится в словарь).

```
func(1, 2, 3, 4, kx='Длина', ky='Ширина', kz='Высота')  
# Выведет: ((1, 2, 3, 4), 42, {'kx': 'Длина', 'ky': 'Ширина', 'kz': 'Высота'})
```

Здесь появился аргумент со значением 42, хотя мы его не указывали, когда вызывали функцию. Но так как в описании функции он присутствовал в виде именованного аргумента со своим значением, это значение и подставилось при вызове функции.

```
func(1, 2, 3, 4, kk='Ящик', kx='Длина', ky='Ширина', kz='Высота')  
# Выведет: ((1, 2, 3, 4), 'Ящик', {'kx': 'Длина', 'ky': 'Ширина', 'kz': 'Высота'})
```

Именованному аргументу `kk` мы подставили другое значение. Вместо 42 записали 'Ящик'. Это значение и вышло при вызове функции.

Аргумент `*args` в определении функции пишется после всех обычных позиционных аргументов, но перед первым аргументом со значением по умолчанию. А `**kwargs` пишется в самом конце после последнего аргумента со значением по умолчанию.

Согласно этому правилу у нас идет следующий порядок расстановки аргументов:

1. Обычные позиционные аргументы
2. Аргумент `*args`
3. Аргументы со значением по умолчанию
4. Аргумент `**kwargs`

```
func(1, 2, 3, 4, kk='Ящик', kx='Длина', ky='Ширина', kz='Высота')  
# Выведет: ((1, 2, 3, 4), 'Ящик', {'kx': 'Длина', 'ky': 'Ширина', 'kz': 'Высота'})  
def func(x, y, *args, kx=None, ky=42, **kwargs):  
    return (x, y, args, kx, ky, kwargs)
```

```
func(1, 2, 3, 4, kx='a', ky='b', kz='Почта')  
# Выведет: (1, 2, (3, 4), 'a', 'b', {'kz': 'Почта'})
```

1, 2 - относятся к позиционным аргументам `x`, `y`.

(3, 4) - получившийся кортеж, относится к аргументам, с произвольным количеством - `*args`.

'a', 'b', - именованные аргументы `kx`, `ky`.

{'kz': 'Почта'} — словарь, полученный с произвольным количеством именованных аргументов `**kwargs`. В данном случае один элемент словаря. Одновременно все 4 пункта на практике практически не используются. Но важно, то какие формы можно сочетать.

Передача именованных аргументов с помощью словаря

Как и в случае позиционных аргументов, именованные можно передавать в функцию в виде словаря. Для этого перед словарем нужно поставить две звездочки. Пример:

```
def coords(x, y):  
    return (x, y)  
coords(x=1, **{'y': 2})  
# Выведет: (1, 2)
```

Здесь указан обычный именованный аргумент, а другой завернут в словарь.

Анонимные функции, инструкция `lambda`

Анонимные функции могут содержать лишь одно выражение, но, при всём при этом, выполняются они быстрее. Анонимные функции создаются с помощью

инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как с описанием функции def func():

```
func = lambda x, y: x + y  
func(1, 2)
```

Возвратит: 3

После ключевого слова lambda идут аргументы, а после :, сама функция.

```
func('a', 'b')
```

Возвратит: 'ab'

Если мы подставим символьные строки (буквы) к нашим аргументам x, y, то получим результирующую строку как сумму двух исходных.

```
(lambda x, y: x + y)(1, 2)
```

Возвратит: 3

Можно и не присваивать описание функции lambda, а сразу задать значение в скобках как кортеж. И также для строковых переменных:

```
(lambda x, y: x + y)('a', 'b')
```

Возвратит: 'ab'

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

```
func = lambda *args: args  
func(1, 2, 3, 4)
```

Возвратит: (1, 2, 3, 4)

```
func = lambda *args: args * 3
```

```
func(1, 2, 3, 4)
```

Возвратит: (1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)

Файлы. Работа с файлами.

Python может обращаться к файлам на дисках. Например, может открывать существующий файл.

```
f = open('text.txt', 'r')
```

Здесь open() - функция, которая позволяет открыть файл 'text.txt', а параметр 'r' означает что будет произведено чтение из файла. 'r' от слова read (читать).

У функции open много параметров. Путь к файлу может быть относительным или абсолютным. Второй аргумент, это режим, в котором мы будем открывать файл.

Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
'x'	открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме. По умолчанию режим равен 'rt'.

И последний аргумент, encoding, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

Чтение из файла

Открыли файл, а теперь хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них.

Первый - метод read, читающий весь файл целиком, если был вызван без аргументов, и второй, если был вызван с аргументом (целым числом n).

Пусть у нас есть текстовый файл text.txt, в котором записана строка:

```
Hello world!\nThe end.\n\n\n
```

Откроем его.

```
f = open('test.txt')
```

```
f.read(1)
```

```
# Возвратит: 'H'
```

```
f.read()
```

```
# Возвратит: 'ello world!\nThe end.\n\n\n'
```

В начале мы читаем один символ из файла и после прочтения курсор как бы передвигается на одну позицию. И, если мы хотим прочитать весь файл, то не указываем аргументы в функции open(). Ясно видно, что оставшаяся часть файла будет распечатываться без первого символа.

Также видно, что выводя полное содержимое текстового файла, в самом конце появляется ещё один знак перевода строки \n (а в файле было только два символа перевода строки). Это надо учитывать, и, если нужно, удалить этот символ, используя функцию replace('\n\n\n', '\n\n'):

```
z = 'ello world!\nThe end.\n\n\n'
```

```
print(z.replace('\n\n\n', '\n\n'))
```

```
# Возвратит: 'ello world!\nThe end.\n\n'
```

Ещё один способ сделать это - прочитать файл построчно, воспользовавшись циклом for:

```
f = open('test.txt')
```

```
for line in f:
```

```
    line
```

```
# Возвратит: 'Hello world!\n'
```

```
'\n'
```

```
'The end.\n'
```

```
'\n'
```

```
'\n'
```

```
f.close()
```

После работы с файлом указатель файла нужно закрыть, используя метод close().

Запись в файл

Теперь рассмотрим запись в файл.

Сначала рассмотрим создание списка из одного элемента:

```
i = 0
```

```
list = [str(i)+str(i-1)]
```

```
print(list)
```

```
# Получим: ['0-1']
```

Здесь функция `str()` преобразует целочисленный тип переменной в строковый и складывает эти строковые символы '0' и '-1'. Так как, если от 0 вычесть 1, то получится -1. И сложение этих величин уже в строковом виде и будет '0-1'. А обрамление в квадратные скобки нам дает тип списка из одного символьного элемента.

Теперь получим список от 0 до 19 в символьном виде, то есть преобразуем целочисленный вид в строковый, как мы уже делали, с помощью функции `str()`.

При этом используем цикл `for`:

```
list = [str(i) for i in range(20)]
```

```
print(list)
```

```
# Получим: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19']
```

Попробуем записать в файл вот такой вот список:

```
list = [str(i)+str(i-1) for i in range(20)]
```

```
print(list)
```

```
# Возвратит: ['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98', '109', '1110', '1211', '1312', '1413', '1514', '1615', '1716', '1817', '1918']
```

```
# И запишем его в файл 'text.txt' в текущем каталоге. Для этого откроем файл на запись:
```

```
f = open('text.txt', 'w')
```

```
# Запись в файл осуществляется с помощью метода write:
```

```
for index in list:
```

```
    f.write(index + '\n')
```

```
# Возвратит:
```

```
4
```

```
3
```

```
3
```

```
3
```

```
3
```

```
3
```

```
3
```

```
3
```

```
3
```

```
4
```

```
5
```

```
5
```

```
5
```

```
5
```

```
5
```

```
5
```

```
5
```

```
5
```

5

5

После окончания работы с файлом его **обязательно нужно закрыть** с помощью метода `close`:

```
f.close()
```

Возникает вопрос: что это за цифры в распечатке? Поясним, метод `write()` возвращает число записанных символов.

Открыв файл `text.txt` мы в этом можем убедиться. Но почему, например, первая строка показывает нам 0-1. Вроде бы 3 символа, а метод `write()` даёт 4 символа, а вторая строка 10 даёт 3 символа. В чём дело? А дело в том, что мы не учли символ перевода строки `'\n'`, который прибавляется к текущему элементу. Этот символ управляющий. Он не виден в содержании файла, но он воздействует на представление информации в файле. Поэтому-то и записывается каждый новый элемент с новой строки.

Теперь попробуем воссоздать этот список из получившегося файла. Откроем файл на чтение и прочитаем строки.

```
f = open('text.txt', 'r')
```

```
list = [line.strip() for line in f]
```

```
print(list)
```

```
# Возвратит: ['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98', '109', '1110', '1211', '1312', '1413', '1514', '1615', '1716', '1817', '1918']
```

```
f.close()
```

Мы получили тот же список, что и был. В более сложных случаях (словарях, вложенных кортежей и т. д.) алгоритм записи придумать сложнее. Но это и не нужно. В Python уже давно придумали средства, такие как `pickle` или `json`, позволяющие сохранять в файле сложные структуры.

Метод `strip()` используется для удаления начальных и конечных пробелов каждого элемента, а так же управляющих символов.

Если не использовать этот метод, то получим следующую картину?

```
f = open('text.txt', 'r')
```

```
list = [line for line in f]
```

```
print(list)
```

```
f.close()
```

```
['0-1\n', '10\n', '21\n', '32\n', '43\n', '54\n', '65\n', '76\n', '87\n', '98\n', '109\n', '1110\n', '1211\n', '1312\n', '1413\n', '1514\n', '1615\n', '1716\n', '1817\n', '1918\n']
```

Урок 5

Исключения в Python

Конструкция try - except для обработки исключений

Исключения необходимы для того, чтобы сообщать программисту об ошибках. Самый простейший пример исключения - деление на ноль. Загрузим Python и напишем:

```
100 / 0
```

```
# Выведет:
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
```

Интерпретатор Python нам сообщает о том, что он нашёл ошибку (поймал исключение) и напечатал информацию Traceback (most recent call last) - отслеживание самого последнего вызова. ZeroDivisionError: division by zero — ошибка деления на ноль.

Можно привести и сообщения о других ошибках (исключениях):

```
2 + '1'
```

```
# Выведет:
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Нельзя складывать различные типы данных. В данном случае целочисленный тип 2 и символьный тип '1'.

Пройдут без ошибок варианты:

```
2 + 1
```

```
# Возвратит: 3
```

```
'2' + '1'
```

```
# Возвратит: '21'
```

```
2 + 1.5
```

```
# Возвратит: 3.5
```

Будет возникать ошибка в попытке перевести символьную переменную, отличную от цифр:

```
int('Восход')
```

```
# Выведет:
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

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

Если же мы напишем символьную переменную только из цифр, ошибки не будет:

```
int('12345')
# Возвратит: 12345
int('12345.89')
# Выведет:
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: invalid literal for int() with base 10: '12345.89'

Для перевода символьной строки, обозначающую вещественное число, снова появится ошибка, так как в символьной строке присутствует символ точки, отличной от символов цифр.

Чтобы не было ошибки надо такую символьную строку преобразовывать с помощью функции `float()`, то есть преобразование в вещественное число:

```
float('12345.89')
```

```
# Возвратит: 12345.89
```

В этих примерах генерируются исключения `TypeError` и `ValueError`. Подсказки дают полную информацию о том, где порождено исключение, и с чем оно связано.

Рассмотрим список встроенных в Python исключений. Надо сказать, что список встроенных в Python исключений имеет вложенную структуру, то есть имеются базовые исключения, которые содержат в себе более конкретные исключения. Такой список приведён ниже:

- **BaseException** - базовое исключение, от которого берут начало все остальные.
 - **SystemExit** - исключение, порождается функцией `sys.exit` при выходе из программы.
 - **KeyboardInterrupt** - порождается при прерывании программы пользователем (обычно сочетанием клавиш `Ctrl+C`).
 - **GeneratorExit** - порождается при вызове метода `close` объекта `generator`.
 - **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
 - **StopIteration** - порождается встроенной функцией `next`, если в итераторе больше нет элементов.
 - **ArithmeticError** - арифметическая ошибка.
 - **FloatingPointError** - порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
 - **OverflowError** - возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как Python поддерживает длинные числа), но может возникать в некоторых других случаях.
 - **ZeroDivisionError** - деление на ноль.

- **AssertionError** - выражение в функции assert ложно.
- **AttributeError** - объект не имеет данного атрибута (значения или метода).
- **BufferError** - операция, связанная с буфером, не может быть выполнена.
- **EOFError** - функция наткнулась на конец файла и не смогла прочитать то, что хотела.
- **ImportError** - не удалось импортирование модуля или его атрибута.
- **LookupError** - некорректный индекс или ключ.
 - **IndexError** - индекс не входит в диапазон элементов.
 - **KeyError** - несуществующий ключ (в словаре, множестве или другом объекте).
- **MemoryError** - недостаточно памяти.
- **NameError** - не найдено переменной с таким именем.
 - **UnboundLocalError** - сделана ссылка на локальную переменную в функции, но переменная не определена ранее.
- **OSError** - ошибка, связанная с системой.
 - **BlockingIOError** - может возникать в ситуациях при выполнении операций ввода-вывода. Например, операция выполняется в неблокирующем режиме, операция ввода-вывода блокируется из-за нехватки ресурсов.
 - **ChildProcessError** - неудача при операции с дочерним процессом.
 - **ConnectionError** - базовый класс для исключений, связанных с подключениями.
 - BrokenPipeError
 - ConnectionAbortedError
 - ConnectionRefusedError
 - ConnectionResetError
 - **FileExistsError** - попытка создания файла или директории, которые уже существуют.
 - **FileNotFoundError** - файл или директория не существует.
 - **InterruptedError** - системный вызов прерван входящим сигналом.
 - **IsADirectoryError** - ожидался файл, но это директория.
 - **NotADirectoryError** - ожидалась директория, но это файл.
 - **PermissionError** - не хватает прав доступа.
 - **ProcessLookupError** - указанного процесса не существует.
 - **TimeoutError** - закончилось время ожидания.

- **ReferenceError** - попытка доступа к атрибуту со слабой ссылкой (Модуль weakref).
- **RuntimeError** - возникает, когда исключение не попадает ни под одну из других категорий.
- **NotImplementedError** - возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
- **SyntaxError** - синтаксическая ошибка.
 - **IndentationError** - неправильные отступы.
 - **TabError** - смешивание в отступах табуляции и пробелов.
- **SystemError** - внутренняя ошибка.
- **TypeError** - операция применена к объекту несоответствующего типа.
- **ValueError** - функция получает аргумент правильного типа, но некорректного значения.
- **UnicodeError** - ошибка, связанная с кодированием / раскодированием unicode в строках.
 - **UnicodeEncodeError** - исключение, связанное с кодированием unicode.
 - **UnicodeDecodeError** - исключение, связанное с декодированием unicode.
 - **UnicodeTranslateError** - исключение, связанное с переводом unicode.
- **Warning** - предупреждение.

В этом списке показана вложенность исключений путём сдвига вправо, относительно базового исключения.

Для обработки исключений используется конструкция try - except.

Как же работает инструкция try/except? Сначала будет выполнено выражение, которое расположено в блоке try. Если внутри блока try не возникло никаких исключений, то блок except пропускается и выполняется дальнейший код. Если все же внутри блока try возникло исключение, то оставшийся код в этом блоке пропускается и выполняется код в блоке except и управление передается следующей инструкции после этого блока.

Инструкция try может содержать более одного except, чтобы указать обработчики для разных исключений try оператора.

Пример применения этой конструкции:

try:

```
a = input("Введите первое число: ")
b = input("Введите второе число: ")
print("Результат: ", int(a)/int(b))
```

except ValueError:

```
print("Пожалуйста, вводите только числа")
```

except ZeroDivisionError:

```
print("На ноль делить нельзя")
```

Текст этого кода запишите в файл, например, try.py, выйдете из интерпретатора и запустите команду:

```
python3 try.py
```

В коде этой программы с помощью try отлавливаются ошибки, во-первых при введении чисел. А они как мы знаем вводятся символами. И, чтобы произвести над ними математические действия, нужно их перевести в целочисленный вид с помощью функции int(). А вот это уже может содержать ошибку, если пользователь ввёл вместо строк с символами только цифр, чревато напороться на исключение ValueError. Если мы введём вещественное число, содержащее точку перед дробной частью, то будет отловлено исключение при переводе этой символьной строки, содержащей вещественное число, в целое, так как точка это символ, отличный от символов цифр. А дальше, если второе введённое число равно нулю, произойдёт уже второе исключение ZeroDivisionError со своим обработчиком.

Ещё примеры, позволяющие перехватывать как само исключение, так и его потомки. Например, перехватывая ArithmeticError, мы также перехватываем FloatingPointError, OverflowError и ZeroDivisionError, так как ArithmeticError является предком для этих исключений. Тем самым мы увеличиваем диапазон обработки возможных исключений. Правда здесь мы не сможем определить какую же конкретную ошибку отследила система.

Поясним: если FloatingPointError — ошибка плавающей точки (будет рассмотрено более подробно).

OverflowError - ошибка переполнения. Самая частая причина этой ошибки - попросить компьютер посчитать что-то слишком большое.

Чтобы исправить ошибку OverflowError есть два пути - использовать специальные библиотеки для вычисления больших чисел или уменьшить сами вычисления.

ZeroDivisionError - уже известное ошибка деления на 0.

```
try:
```

```
    k = 1 / 0
```

```
except ArithmeticError:
```

```
    print("Перехваченное исключение ArithmeticError, а конкретно исключение  
ZeroDivisionError, которое в него входит, но система это не конкретизирует.
```

```
Потому, как говорится, догадайся сам.")
```

```
    k = 0
```

```
print(k)
```

```
0
```

Для рассмотрения исключения FloatingPointError надо пояснить представление чисел с плавающей точкой практически во всех языках программирования, в том числе и в Python.

Компьютер обманывает нас.

Попробуйте и увидите такой факт, что $0.1 + 0.2$ не равно 0.3 , а равно 0.30000000000000004 .

Проблема касается и сравнения:

```
0.1 + 0.2 <= 0.3
# Выдаст: False
10.4 + 20.8 > 31.2
# Выдаст: True
0.8 - 0.1 > 0.7
# Выдаст: True
```

Что происходит? Когда вы вводите в интерпретатор Python число 0.1, оно сохраняется в памяти как число с плавающей точкой и происходит преобразование. 0.1 — это десятичное число с основанием 10, но числа с плавающей точкой хранятся в двоичной записи. То есть основание 0.1 преобразуется из десятичной в двоичную систему счисления..

Рассмотрим эти системы счисления.

Десятичная система счисления.

Она характеризуется наличием 10 цифр от 0 до 9.

Сначала мы записываем 0, затем, чтобы продолжить этот ряд цифр на увеличение, мы увеличиваем 0 на 1 ($0 + 1 = 1$). далее к результату снова прибавляем 1. Получаем $1 + 1 = 2$. К 2 прибавляем 1 получаем 3. И таким образом доходим до последней наибольшей цифры в десятичной системе счисления, то есть к 9.

Дальнейшее увеличение этой цифры (9), сталкивает нас с ситуацией, когда необходимо вводить какое-либо обозначение уже числа, большего цифры 9 на 1. Такого обозначения нет. Поэтому вводят понятие зависимости числа от позиции.

Следующее число после цифры 9 записывается как двузначное. Мы записываем вместо цифры 9 цифру 0, а слева от 0 вторую цифру из ряда цифр от 0 до 9, то есть цифру 1. Получилось 10. Прибавляя к этому результату 1, мы вместо 0 записываем 1, а 1 в более старшем разряде, что находится слева, мы не трогаем. В записи получилось 11. В следующем шаге мы также прибавляем 1, то есть вместо 1 младшего разряда (справа) мы вместо 1 записываем 2. Получилось 12. Как мы видим старший разряд мы не трогаем.

Так, прибавляя по 1, мы доходим до числа 19. Следующее прибавление 1 к этому числу изменит значение как младшего, так и старшего разряда и получится число 20.

Это касается того момента, когда у нас закончатся двузначные числа, ограниченные числом 99. Дальше пойдут трёхзначные числа. Сначала мы записываем 0 там, где было 9 младшего разряда, и должны прибавить 1 к следующему слева на право разряду. А там у нас стояла ещё одна девятка. Поэтому мы ставим здесь 0 и сдвигаемся ещё влево и ставим там 1. Получилось 100.

Продолжая такие действия (от 0 до бесконечности), используя цифры от 0 до 9 с позиционным написанием, мы получим ряд из положительных целочисленных рациональных чисел.

А что, если нам рассмотреть другую систему счисления, не десятичную, а, например восьмеричную. В такой системе счисления не 10 цифр, а 8. От 0 до 7.

Когда мы прибавляем 1 в этой системе счисления, мы начинаем также от 0 и доходим до 7. Дальше в этой системе счисления у нас больше цифр нет. Поступаем также, как в десятичной системе счисления. Сдвигаемся на более старший разряд слева и ставим 1, а вместо 7 в младшем разряде вместо 7 ставим 0. Получилось 10. Эта композиция в восьмеричной системе счисления означает 8. Поэтому можно записать, что $8_{10} = 10_8$. А, например, $10_{10} = 12_8$, $12_{10} = 14_8$. В компьютерах используется двоичная система счисления. Физически удобно хранить информацию в таком виде. Каждая ячейка памяти или регистра процессора, может находиться в одном из состояний, либо 0, то есть нет потенциала, либо 1, то есть есть потенциал. Из трёх разрядов в двоичной системе исчисления можно составить 8 комбинаций чисел, как в 8-ричной, так и в 10-тичной системах счисления. Проверим:

$000_2 = 0_8$ или 0_{10} или 0_{16}

$001_2 = 1_8$ или 1_{10} или 1_{16}

$010_2 = 2_8$ или 2_{10} или 2_{16}

$011_2 = 3_8$ или 3_{10} или 3_{16}

$100_2 = 4_8$ или 4_{10} или 4_{16}

$101_2 = 5_8$ или 5_{10} или 5_{16}

$110_2 = 6_8$ или 6_{10} или 6_{16}

$111_2 = 7_8$ или 7_{10} или 7_{16}

$1000_2 = 10_8$ или 8_{10} или 8_{16}

$1001_2 = 11_8$ или 9_{10} или 9_{16}

$1010_2 = 12_8$ или 10_{10} или A_{16}

$1011_2 = 13_8$ или 11_{10} или B_{16}

$1100_2 = 14_8$ или 12_{10} или C_{16}

$1101_2 = 15_8$ или 13_{10} или D_{16}

$1110_2 = 16_8$ или 14_{10} или E_{16}

$1111_2 = 17_8$ или 15_{10} или F_{16}

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F — шестнадцатеричная система счисления.

Перевод числа из двоичной системы счисления в десятичную может понадобиться при программировании, криптографии, а также в различных математических и инженерных расчетах. В языке программирования Python есть несколько способов выполнения такого перевода.

Один из самых простых и понятных способов перевода числа из двоичной системы в десятичную — использовать встроенную функцию `int()` с указанием основания системы счисления. Например, чтобы перевести число 101011 из двоичной системы в десятичную, можно использовать следующий код:

```
int('1111011', 2)
```

```
# Выдаст: 123
```

Преобразовать десятичное число в двоичный формат можно при помощи функции `bin()`: Пример

```
a = bin(20)
```

```
print(a)
```

```
# Выдаст: 0b10100
```

Аналогично можно преобразовать в шестнадцатеричный формат при помощи функции `hex()`, например:

```
a = hex(20)
```

```
print(a)
```

```
# Выдаст: 0x14
```

Обратите внимание, функции `bin()` и `hex()` возвращают результат в виде строки.

```
number = 123
```

```
result = bin(number)
```

```
print(result)
```

```
# Выдаст: 0b1111011
```

У результата также будет префикс `0b`, указывающий на основание системы счисления. В - бинарное, то есть двоичное.

Получающееся двоичное число может недостаточно точно представлять исходное число с основанием 10. 0.1 - один из примеров. Двоичным представлением будет 0.0(0011). Читается как ноль, точка, ноль и 0011 в периоде, то есть бесконечно повторяющаяся последовательность. То есть 0.1 — это бесконечно повторяющееся десятичное число, записанное с основанием 2. То же происходит, когда в виде десятичного числа с основанием 10 записывается дробь $\frac{1}{3}$. Получается бесконечно повторяющееся десятичное число 0.3(3).

Память компьютера конечна, поэтому бесконечно повторяющееся представление двоичной дроби 0.1 округляется до конечной дроби. Её значение зависит от архитектуры компьютера (32- или 64-разрядная).

Эта ошибка представления чисел с плавающей точкой встречается достаточно часто.

Ошибка представления числа очень типична

Существуют три причины, почему число округляется при его представлении в виде числа с плавающей точкой:

1. В числе больше значащих разрядов, чем позволяет плавающая точка.
2. Это иррациональное число, то есть число с бесконечно повторяющейся последовательностью, таких как число π . Название числа π происходит от начальных букв греческих слов "окружность" и "периметр". $\pi = 3,141592653589793238462643...$ и так до бесконечности.
3. Число рациональное, то есть число с бесконечно повторяющейся последовательностью, таких как $\frac{1}{3} = 0.3(3)$.

Множество рациональных чисел содержит как дробные числа (обыкновенные и десятичные дроби, смешанные числа), так и целые числа.

64-разрядные числа с плавающей точкой имеют 16 или 17 значащих разрядов.

Любое число, у которого значащих разрядов больше, округляется.

Конечные дроби здесь только те, знаменатель которых равен степени числа 2. В результате дроби $\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{6}$, $\frac{1}{7}$, $\frac{1}{9}$ и $\frac{1}{10}$ - бесконечные, когда представлены в двоичной записи.

Вернёмся к проблеме исключений.

Существуют ещё две инструкции, это `finally` и `else`. `finally` выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда

нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция `else` выполняется в том случае, если исключения не было.

Допустим у нас есть файл `1.txt`. Он содержит следующую информацию:

```
1
2
3
'string'
4
5
```

И напишем программу, иллюстрирующую эти инструкции `finally` и `else`.

```
f = open('1.txt')
ints = []
try:
    for line in f:
        ints.append(int(line))
except ValueError:
    print('Это не число. Выходим.')
else:
    print('Всё хорошо.')
finally:
    f.close()
    print('Я закрыл файл.')
    # Именно в таком порядке: try, группа except, затем else, и только потом
    finally.
```

```
# Выдаст
```

```
Это не число. Выходим.
```

```
Я закрыл файл.
```

```
Если мы уберём символьную строчку 'string' из файла 1.txt, то будет выдано
сообщение:
```

```
Всё хорошо.
```

```
Я закрыл файл.
```

```
Как видим инструкция finally выполняется всегда.
```

Урок 6

Регулярные выражения в Python

Регулярные выражения представляют собой последовательности символов, составляющие шаблоны. По этим шаблонам в исследуемой строке или тексте ищутся соответствия. Для работы с регулярными выражениями (сокращённо `regex`) в Python имеется встроенный модуль `re`.

Обычное использование регулярного выражения:

1. Поиск подстроки в строке (search and find)
2. Поиск всех подходящих строк (findall)
3. Разделение строки на подстроки (split)
4. Замена части строки (sub)

Рассмотрим пару примеров нахождения подстроки в строке по шаблону.

```
import re
pattern = '\d\d\.\d\d\.\d{4}'
string = 'Я родился 18.01.1952 года.'
reg = re.search(pattern, string)
print(reg)
```

Выдаст: <re.Match object; span object; span=(10, 20), match='18.01.1952'>
span (интервал) находится между 10 и 20 символом исследуемой строки.
Начинается после 10 символа и заканчивая 20 символом.

match (совпадение) - это результат поиска функции search() нашей строки

```
import re
print(re.findall(r'\d\d\.\d\d\.\d{4}', r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'))
```

Выдаст: ['19.01.2018', '01.09.2017']

Метод findall модуля re находит все совпадения шаблону и помещает результат в список, который в данном случае, выводится функцией print().

Что же означают эти загадочные символы в шаблоне '\d\d\.\d\d\.\d{4}'?

\d специальное значение означающее одну цифру от 0 до 9.

А вот точка здесь служит собственно символом поиска, то есть в исследуемом тексте или строке после двух любых цифр идёт символ точки, и она экранируется обратной косой чертой. Это делается для того, чтобы не перепутать точку как символ и точку как метасимвол, означающий любой символ, кроме символа новой строки. Потом идут следующие любые две цифры и снова точка.

Последняя запись в этом шаблоне \d{4} означает, что цифр должно быть 4, что и записано в фигурных скобках.

Попробуем немного поэкспериментировать с последним примером.

Для начала добавим ещё одну дату в исследуемое предложение. Дата будет в том же формате, что и предыдущие.

```
import re
print(re.findall(r'\d\d\.\d\d\.\d{4}', r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017 или 18.07.2019'))
```

Выдаст: ['19.01.2018', '01.09.2017', '18.07.2019']

Теперь в последней дате сделаем ошибку. Вместо традиционных разделителей между числом, месяцем и годом поставим какую-нибудь букву. Предыдущий шаблон проигнорирует последнюю дату.

```
import re
print(re.findall(r'\d\d\.\d\d\.\d{4}', r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017 или 18ю07ю2019'))
```

Выдаст: ['19.01.2018', '01.09.2017']

А если мы вместо \. поставим \D, что означает любой не числовой символ, то программа найдёт все 3 числа

```
import re
print(re.findall(r'\d\d\D\d\d\D\d{4}', r'Эта строка написана 19.01.2018, а могла бы и 01.09.2018 или 18ю07ю2019'))
# Выдаст: ['19.01.2018', '01.09.2018', '18ю07ю2019']
```

Заменим в последней дате в качестве разделителя тире (-).

```
import re
print(re.findall(r'\d\d\d\d\d\d\d{4}', r'Эта строка написана 19.01.2018, а могла бы и 01.09.2018 или 18-07-2019'))
# Выдаст: ['18-07-2019']
```

Видим, что найдено только последняя дата, так как только в ней присутствует в качестве разделителя тире.

Нужно пояснить, что метасимвол точка (.) означает любой символ, кроме символа новой строки. Сюда может входить буква, цифра, любой другой знак. Вот почему, когда мы использовали метасимвол точка (.) и написали в исследуемой строке в качестве разделителя числа, месяца, года не точку (как символ), а цифру, пробел, букву, мы получили все три даты. Правда с нетрадиционными разделителями.

```
import re
print(re.findall(r'\d\d\d\d\d\d\d{4}', r'Эта строка написана 19a01a2018, а могла бы и 01109ю2018 или 18-07-2019'))
# Выдаст: ['119a01a2018 ', '01109ю2018', '18-07-2019']
```

Какие же метасимволы используются в регулярных выражениях?

- \ используется для игнорирования специального значения символа
- [] указывает на целый класс символов. Например: [a-z] — все буквы латинского алфавита в нижнем регистре, [a-zA-Z0-9] — все буквы латинского алфавита в обоих регистрах плюс цифры, [a-яA-Я0-9] — все буквы русского алфавита в обоих регистрах плюс цифры. Порядок ряда символов в наборе неважен. Так, например, регулярное выражение [Tt]he означает: T или t, за которым следует буква h, за которой следует буква e. [Tt]he => **The car parked in the garage.**

- ^ соответствует началу текста (этот метасимвол указывает, что стоящий за ним текст должен соответствовать шаблону). Обычно символ ^ представляет начало строки, но когда он внутри квадратных скобок, все символы, которые находятся после него, исключаются из шаблона. Например, выражение [^c]ar поможет отыскать все символы кроме c, за которыми следуют a и r.

```
import re
print(re.findall(r'[^c]ar', r'The car parked in the garage.))
# Выдаст: ['par', 'gar'], а 'car' не выдаст
```

- \$ обозначает конец текста (этот метасимвол указывает, что стоящий перед ним текст должен соответствовать шаблону).

- . соответствует любому символу, кроме символа новой строки (символ новой строки обозначается \n)
- ? обозначает одно или ноль вхождений (то есть может быть один какой-либо символ, а может и не быть ни одного. Скажем, номер мобильного телефона может быть записан как 89260113682 или 8-926-011-36-82. И тогда использование в шаблоне -? означает, что знак — может присутствовать или отсутствовать в номере телефона).

```
import re
print(re.findall(r'\d-?\d{3}-?\d{3}-?\d{2}', r'Мой телефон имеет номер
89260113682, а номер телефона моего друга 8-926-111-53-25'))
# Выдаст: ['89260113682', '8-926-111-53-25']
```

- | Перечисления (операция «ИЛИ»)

Запись A | B означает A или B

Например, отдельные овощи в тексте можно искать при помощи шаблона
морковк|св[её]кл|картошк|редиск

Таким шаблоном ищутся или морковк (окончание отсутствует, потому что неизвестно в каком падеже это слово встречается в тексте морковка, морковке и т.д.), или св[её]кл (свекла, свёкле и т.д., [её] показывает диапазон либо е, либо ё), или картошк (тоже без окончания), редиск (без окончания).

Давайте испробуем этот шаблон.

Напишем такой текст:

'Для приготовления борща в нашем рецепте две большие морковки, одна крупная свекла, картошка (можно около килограмма), лук, кусочек не жирного мяса. Сначала отвариваем мясо, режем лук кольцами, очищаем морковку и натираем её на крупной тёрке. Затем морковку и лук пассивируем на сковороде. Свеклу отвариваем, натираем на тёрке, и все ингредиенты добавляем в мясной бульон. Варим 40 минут.'

```
import re
print(re.findall(r'морковк|св[её]кл|картошк|редиск', r'Для приготовления борща в
нашем рецепте две большие морковки, одна крупная свекла, картошка (можно
около килограмма), лук, кусочек не жирного мяса. Сначала отвариваем мясо,
режем лук кольцами, очищаем морковку и натираем её на крупной тёрке. Затем
морковку и лук пассивируем на сковороде. Свеклу отвариваем, натираем на
тёрке, и все ингредиенты добавляем в мясной бульон. Варим 40 минут.))
# Выдаст: ['морковк', 'свекл', 'картошк', 'морковк', 'морковк'] # Ни мяса, ни лука не
будет выдано, так как в шаблоне их не было.
```

- * любое количество вхождений (включая 0 вхождений, эквивалентно {0,})
Следующие мета-символы +, * или ? используются для того, чтобы обозначить допустимое количество повторения подшаблона.
a* - говорит нам о том, что символ a может повторяться сколько угодно раз или повторений не будет.
- + одно и более вхождений
- {} указывает на несколько совпадений предыдущего RE.
- () отделяет группу в регулярном выражении

И ещё в регулярных выражениях используются специальные значения. Они как правило начинаются с обратной косой черты (backslash). Нам уже знакомы \d и \D, а из метасимволов точка (.). Использование точки не как специального значения, управляющего шаблоном, а как просто символа (смотри самый первый пример для функции findall()), мы должны этот символ 'экранировать', то есть поставить перед точкой обратную косую черту.

Вот список специальных значений в регулярных выражениях:

- \d соответствует любой десятичной цифре, это то же самое, что и [0-9]
- \D соответствует любому не числовому символу
- \s соответствует любому пробельному символу
- \S соответствует любому не пробельному символу
- \w соответствует любому буквенно-числовому символу; это то же самое, что и [a-zA-Z0-9_].
- \W соответствует любому не буквенно-числовому символу.

Если же необходимо использовать backslash просто как символ, без учета специального значения, его нужно «экранировать» еще одной обратной косой чертой – \\.

Теперь давайте посмотрим, какие методы доступны в модуле re.

re.match() - поиск вхождения шаблона в начало строки

В качестве аргументов требуется передать сначала шаблон, а потом строку для проверки:

```
import re
str = "Банкиров ребрендили-ребрендили-ребрендили, да не выребрендировали"
print(re.match(r'pe', str))
```

Если запустить этот код, то Python выведет None, несмотря на то что «ре» в строке встречается восемь раз. Все из-за того, что re.match() ищет вхождение шаблона именно в начале строки, а не в любом месте. Можно исправить r'pe' на r'Ба', тогда получим нужный результат.

re.search()

Этот метод возвращает совпадающую часть строки и останавливается сразу же, как находит первое совпадение. Таким образом, его можно использовать для проверки выражения, а не для извлечения данных.

Синтаксис: re.search(шаблон, строка)

Возвращаемое значение может быть либо подстрокой, соответствующей шаблону, либо None, если такой подстроки не окажется.

Давайте разберем пример: поищем в строке число и месяц.

```
# -*- coding: utf-8 -*-
```

```
import re
regex = r"((\d+) [a-яA-Я]+)"
match = re.search(regex, "Мой сын родился 20 июля")
if match != None:
    print("Совпадение по индексу %s, %s" % (match.start(), match.end())) #Это
обеспечивает индекс совпадающей строки
    print("Полное соответствие: %s" % (match.group(0)))
```

else:

```
print("Данный шаблон регулярного выражения не соответствует")
```

Выведет:

Совпадение по индексу 16, 23

Полное соответствие: 20 июля

re.findall()

Этот метод возвращает все совпадения с шаблоном, которые встречаются в строке. При этом строка проверяется от начала до конца. Совпадения возвращаются в том порядке, в котором они идут в исходной строке.

Синтаксис: `re.findall(шаблон, строка)`

Возвращаемое значение может быть либо списком строк, совпавших с шаблоном, либо пустым списком, если совпадений не нашлось.

Рассмотрим пример. Используем регулярное выражение для поиска почтовых индексов в исходной строке.

```
import re
```

```
string = '119048 10-летия Октября Улица 11, 13, 13 стр. 1, 15 стр. 4, 2 стр. 1, 2 стр. 2, 2 стр. 3; 121170 1812 года Улица 1, 10 к. 1, 10 к. 1 стр. 3, 10 к. 2, 12, 14, 14 стр. 1А; 123100 1905 года Улица 1, 3, 5, 5 стр. 2, 7 стр. 1, 7 стр. 3, 7 стр. 4; 111395 1 Маевки Аллея 1, 11 к. 1, 11 к. 2, 13А стр. 1, 13А стр. 2, 13А стр. 3'
```

```
regex = '\d{6}'
```

```
match = re.findall(regex, string)
```

```
print(match)
```

Выведет: ['119048', '121170', '123100', '111395']

А найти и выдать вообще все числа в этой строке:

```
import re
```

```
string = '119048 10-летия Октября Улица 11, 13, 13 стр. 1, 15 стр. 4, 2 стр. 1, 2 стр. 2, 2 стр. 3; 121170 1812 года Улица 1, 10 к. 1, 10 к. 1 стр. 3, 10 к. 2, 12, 14, 14 стр. 1А; 123100 1905 года Улица 1, 3, 5, 5 стр. 2, 7 стр. 1, 7 стр. 3, 7 стр. 4; 111395 1 Маевки Аллея 1, 11 к. 1, 11 к. 2, 13А стр. 1, 13А стр. 2, 13А стр. 3'
```

```
regex = '\d+'
```

```
match = re.findall(regex, string)
```

```
print(match)
```

Выведет: ['119048', '10', '11', '13', '13', '1', '15', '4', '2', '1', '2', '2', '2', '3', '121170', '1812', '1', '10', '1', '10', '1', '3', '10', '2', '12', '14', '14', '1', '123100', '1905', '1', '3', '5', '5', '2', '7', '1', '7', '3', '7', '4', '111395', '1', '1', '11', '1', '11', '2', '13', '1', '13', '2', '13', '3']

re.compile()

С помощью этого метода регулярные выражения компилируются в объекты шаблона и могут использоваться в других методах. Рассмотрим это на примере поиска совпадений с шаблоном.

```
import re
```

```
e = re.compile('[a-e]') # Откомпилированный шаблон присваивается переменной e. И потом к этой переменной можно применить любой метод регулярных выражений, например, findall().
```

```
print(e.findall("I born at 11 A.M. on 20th July 1989"))
```

```
# Результат:
# ['b', 'a']
e = re.compile('\d') # \d - эквивалент [0-9].
print(e.findall("I born at 11 A.M. on 20th July 1989"))
# Результат:
# ['1', '1', '2', '0', '1', '9', '8', '9']
p = re.compile('\d+') # группа из одной или более цифр
print(p.findall("I born at 11 A.M. on 20th July 1989"))
# Результат:
# ['11', '20', '1989']
```

re.split()

Данный метод разделяет строку по заданному шаблону. Если шаблон найден, оставшиеся символы из строки возвращаются в виде результирующего списка. Более того, мы можем указать максимальное количество разделений для нашей строки.

Синтаксис: `re.split(шаблон, строка, maxsplit = 0)`

Возвращаемое значение может быть либо списком строк, на которые была разделена исходная строка, либо пустым списком, если совпадений с шаблоном не нашлось.

Рассмотрим, как работает данный метод, на примере.

```
import re
string = "шел дождь и два студента"
result = re.split(r' ', string, 4)
print(result)
# Выведет: ['шел', 'дождь', 'и', 'два', 'студента']
import re
string = "шел дождь и два студента"
result = re.split(r' ', string, 1)
print(result)
# Выведет: ['шел', 'дождь и два студента']
```

re.sub()

Здесь значение «sub» - это сокращение от substring, т.е. подстрока. В данном методе исходный шаблон сопоставляется с заданной строкой и, если подстрока найдена, она заменяется параметром `repl`.

Кроме того, у метода есть дополнительные аргументы. Это `count`, счетчик, в нем указывается, сколько раз заменяется регулярное выражение. А также `flag`, в котором мы можем указать флаг регулярного выражения (например, `re.IGNORECASE`)

Синтаксис: `re.sub(шаблон, repl, строка, count = 0, flags = 0)`

В результате работы кода возвращается либо измененная строка, либо исходная.

```
import re
string = "шел дождь и два студента"
result = re.sub(r'два', 'три', string)
print(result)
```

Выведет: шел дождь и три студента

re.subn()

Функциональность subn() во всех отношениях такая же, как и sub().

Единственная разница – это формат вывода. subn() возвращает кортеж, содержащий общее количество замен и новую строку.

Синтаксис: re.subn(шаблон, repl, строка, count = 0, flags = 0)

Допустим мы совершили ошибку, которая сразу не видна. Вместо русских букв ор мы поставили латинские буквы с похожим написанием op. Трудно определить на глаз, где стоят русские или латинские буквы. Для замены всех таких ошибочных написаний используем заменяющий метод subn().

Пример:

```
import re
print(re.subn('op', 'ор', 'Автомобиль попал в затор. Совершил манёвр и выехал на простор.'))
t = re.subn('op', 'ор', 'Автомобиль попал в затор. Совершил манёвр и выехал на простор.', flags=re.IGNORECASE)
print(t)
print(len(t))
```

re.escape()

Этот метод возвращает строку с обратной косой чертой \ перед каждым не буквенно-числовым символом. Это полезно, если мы хотим сопоставить произвольную буквенную строку, которая может содержать метасимволы регулярного выражения.

Синтаксис: re.escape(строка)

Чтобы лучше понять принцип работы метода, рассмотрим следующий пример.

```
import re
print(re.escape('http://www.python.org'))
# http://www\python\.org
```

```
import re
```

```
# Здесь из не буквенно-цифровых символов есть только пробелы.
```

```
print(re.escape("doctor appointment booked successfully at 1PM"))
```

```
# doctor\ appointment\ booked\ successfully\ at\ 1PM
```

```
# Здесь есть , ' ', '^', '-', '[]', '\ ' - все эти символы не относятся к буквенно-цифровым
```

```
print(re.escape("He asked what is this [0-9], I said \t ^Numeric class"))
```

```
# He\ asked\ what\ is\ this\ \[0\-9\]\ \ said\ \ \ ^Numeric\ class
```

Урок 7

Объектно-ориентированное программирование

Прежде чем рассматривать объектно-ориентированное программирование стоит понять об организации программ на Python на наиболее высоком уровне. Самой крупной организационной программной единицей является модуль, который вмещает в себя программный код и данные, готовые для многократного использования.

Каждый файл с программой - это отдельный модуль. Модуль может импортировать другие модули для доступа к именам (в том числе функций), которые в них определены. Надо сказать, что модули, файлы программ, могут иметь расширения, написанные на других языках программирования, таких как C, Java или C#.

Существуют две инструкции получения доступа к функциям модуля:

1. `import <модуль>`
2. `from <модуль> import <блок или функция>`

В первом случае загружается модуль целиком.

Во втором - только определённые имена из модуля.

Модули играют как минимум три роли:

1. Повторное использование программного кода.

Код программы всегда остаётся в файле, поэтому после отработки программы и выхода из оболочки Python, используемый здесь модуль не исчезает. Им можно воспользоваться в любой момент и в другой программе.

2. Разделение системы пространств имен.

Модули позволяют изолировать имена в замкнутые пакеты. И это позволяет избежать конфликтов имен.

3. Реализация служб или данных для совместного пользования

Построение сложной системы требует использования готовых компонентов, копии которые могут импортироваться множеством клиентов, участвующих в разработке системы, и не только им.

Обычно программы состоят более, чем из одного файла. Даже, если ухитриться засунуть весь код программы в один файл, всё рано не избежать использования уже готовых модулей.

Программа организована как один главный файл, к которому могут подключаться дополнительные файлы, как модули.

Инструкция `import`, во-первых, идентифицирует внешний файл и, во-вторых, становится именем переменной, которая будет представлять загруженный

модуль. Любой файл может импортировать функциональные возможности из любого другого файла. Оформив программные компоненты в виде файлов модулей, вы сможете использовать их не только в текущей программе, но и в любых других ваших программах.

Иногда `import` сравнивают с инструкцией `#include` в других языках, например, в С. Это в корне не верно. `#include` отличается от `import` тем, что `#include` просто включает часть текста кода.

Интерпретатор Python, при импортировании модуля, делает следующие действия:

1. Отыскивают файл модуля.
2. Компилируют в байт-код, если он ещё не был создан (У него расширение имени файла не `.py`, а `.pyc` или `.pyo`).
3. Запускают программный код модуля, чтобы создать объекты, которые он определяет.

Python имеет обширную стандартную библиотеку модулей, поставляемую по умолчанию при установке интерпретатора. Все каталоги и файлы этой библиотеки, если брать, например, linux, находятся обычно в файле `/usr/lib64/Python64/site-packages/`.

Найти место положение библиотеки модулей Python можно войдя в интерпретатор из консоли и набрав команду:

```
import sys
sys.path
# Выдаст:
['/usr/lib64/Python64/site-packages/', '/usr/lib64/python3.8.zip', '/usr/lib64/python3.8',
'/usr/lib64/python3.8/lib-dynload', '/home/zhur/.local/lib/python3.8/site-packages',
'/usr/local/lib64/python3.8/site-packages', '/usr/local/lib/python3.8/site-packages',
'/usr/lib64/python3.8/site-packages', '/usr/lib/python3.8/site-packages']
```

По этим каталогам Python будет искать импортируемый модуль. Написав свой собственный модуль, состоявший из одного файла с функциями внутри, или каталога с несколькими программами (файлами), созданные для многократного использования, можно их поместить, например, в каталог `/usr/lib64/Python64/site-packages/` со стандартными модулями. Или, как видно из нулевой позиции списка каталогов (а это пустой каталог `''`). Это означает, что мы можем расположить наш модуль и в текущей папке, где находится и основная программа.

В языке программирования Python модулями являются все файлы с расширением `*.py` (`*` обозначает, что на этом месте может стоять любой символ или любое их количество). Исключением является служебный файл `__init__.py` с двумя подчёркиваниями спереди и сзади.

Чтобы воспользоваться модулем в своей программе нужно его импортировать. Допустим, у нас имеется модуль `own.py`. В нём прописаны некоторые функции и переменные, в том числе функция сложения двух чисел `add(x, y)` и переменная `val = 10`:

```
val = 10
```

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

И мы хотим воспользоваться модулем `own.py` и написать небольшую программу.

```
import own  
print(own.add(7, 14))  
# Выдаст 21  
print(own.add(7, own.val))  
# Выдаст 17
```

Можно использовать `from` для импорта конкретных функций и переменных из модуля, чтобы сократить запись кода.

В этом варианте предыдущая программа будет выглядеть следующим образом:

```
from own import add, val  
print(add(7, 14))  
# Выдаст 21  
print(add(7, val))  
# Выдаст 17
```

А вот пакетами будут все директории с модулями, содержащие пустой файл `__init__.py`. Если такая директория имеет только файл `__init__.py`, то и она будет представлять пакет Python.

Импорт модулей из пакетов записывается через точечную нотацию, то есть сначала идёт имя пакета, а затем через точку имя модуля.

```
import math  
math.factorial(10)  
# Выдаст 3628800
```

Здесь импортируется стандартный математический пакет `math` и вызывается его модуль нахождения факториала. В качестве аргумента берётся 10.

Факториал обозначается $n!$, произносится эн факториал — произведение всех натуральных чисел от 1 до n включительно.

$2! = 1 * 2 = 2$

$3! = 1 * 2 * 3 = 6$

$4! = 1 * 2 * 3 * 4 = 24$

...

$10! = 1 * 2 * 3 * 4 * \dots * 10 = 3628800$

Если имя импортируемой функции конфликтует с именем в вашей функции, то есть они имеют одно и то же имя, или такой случай, что имя импортированной функции слишком велика, на помощь приходят псевдонимы.

Они записываются так:

```
from math import factorial as fc  
fc(10)  
# Выдаст 3628800
```

Из математического модуля `math` стандартной библиотеки Python импортируется функция `factorial()` и, с помощью специального слова `as` (как)

этой функции присваивается новое имя `fs`. И уже оно может быть использовано в программе.

А теперь непосредственно перейдём к объектно-ориентированному программированию (ООП).

Объектно-ориентированное программирование (ООП) — это способ организации программы, позволяющий использовать один и тот же код многократно. В отличие от функций и модулей, ООП позволяет не только разделить программу на фрагменты, но и на основании классов, описывающих как бы прототип (каркас) для объектов реального мира, также организовать связи между этими объектами.

Основа ООП - класс. Класс это сложный тип данных, включающий набор переменных и функций для управления значениями, хранящимися в этих переменных. В ООП переменные называют атрибутами, а функции - методами. Такую терминологию нужно чётко понимать и запомнить, чтобы в дальнейшем не путаться.

И так, класс это фабрика объектов, что позволяет создать неограниченное количество экземпляров, основанных на этом классе. Его также можно сравнить с чертежами и сопутствующей конструкторской документацией по которым можно создавать, например, дома, если, конечно, данная конструкторская документация относится к строительству домов. По этой конструкторской документации можно возвести бесконечное множество домов. И здесь мы подходим к понятию объекта, потому что любой дом возведённый по этой конструкторской документации (то есть это класс), будет являться объектом. Значит по одному классу можно создать множество объектов.

Конечно, эти объекты, если мы конкретно говорим о домах, построенных по заданной конструкторской документации, что мы отнесли к понятию класса, могут отличаться друг от друга. Например, локацией, то есть местоположением, используемыми материалами, не ухудшающими тактико-технических характеристик зданий. Такое часто бывает, когда строится целый микрорайон. Есть такой старый советский анекдот. Американцы выкрали у Советского Союза конструкторскую документацию по изготовлению баллистической ракеты. Стали по ней делать ракету, а у них получалось корыто. Раз делали, два. Всё равно получается корыто. Пошли на поклон к русским. Спрашивают: написано ракета, а получается корыто. Почему? Русские отвечают: «А Вы самое главное не выкрали - лист регистрации изменений».

Конечно, это утрированное сравнение, но на самом деле она, как раз, и характеризует понятия класса и объектов, основанных на них. Получается, что классы могут изменяться. Прирастать новыми свойствами в плане каких-либо показателей и в плане их поведения. Вот здесь мы подходим к основному свойству классов: наследованию. Есть первоначальный класс, называемый родительским. И от него, дополняя и заменяя свойства и действия над ними, мы получаем новый класс с дополнительными свойствами. Он будет потомком первоначальному, родительскому классу. Этот новый класс тоже может иметь своих потомков с дополненными характеристиками.

А, теперь, переходя к объектам, мы можем на основе любого из этих классов, создавать сколько угодно объектов реального мира. Они могут взаимодействовать между собой. На примере домов, это могут быть инженерные сооружения, такие, как теплоцентрали, подача и разводка электричества в дома и так далее.

ООП основывается не только на наследовании. Ещё есть такое понятие как инкапсуляция.

Опять же анекдот.

Крестьянин увидел первый раз паровоз и спросил что это. Ему ответили, что это паровоз. Он сам ездит. У него есть паровой котёл, который подогревается углём через топку. Вода вскипает и образуется пар. Пар под давлением толкает поршень. Поршень через шатун крутит колёса, и те, по железным рельсам катят паровоз вперёд. Крестьянин говорит: «Это всё понятно, только куда впрягать кобылу спереди или сзади?».

В этом анекдоте внутреннее устройство паровоза крестьянина не интересует. Ему подавай главное, как может двигаться паровоз только с его точки зрения. Вот и инкапсуляция скрывает внутреннее устройство класса. Наружу выставляются только главные его характеристики, которые и определяют сущность класса. Например, нам не нужно знать как устроен автомобиль при его покупке. Нам важны марка автомобиля, его пробег, если он не новый, цвет, дата выпуска, стоимость. А как работает двигатель, как крутятся колёса нас не интересуют.

Третье понятие, на чём основывается ООП, полиморфизм.

Снова анекдот.

Покупатель приходит в хозяйственный магазин и спрашивает имеются в наличии зарубежные унитазы. Ему отвечают, что зарубежные унитазы закончились, есть только наши, сибирские. Покупатель сказал: «Жалко, ну что делать, покажите наши, сибирские». Продавец ушёл на склад и приносит две палки. Очень удивился покупатель: «А как же этим пользоваться?». «Очень просто. Одной палкой на морозе замёрзший кал отбиваете, а второй отгоняете волков» - ответил продавец.

Разбирая этот шуточный пример, мы видим, что палки совсем одинаковые, из одного и того же дерева, одной и той же длины, и одного и того же диаметра, А вот действия или функции они выполняют разные.

Перейдём непосредственно к программированию.

Создание класса и экземпляра класса (или, по-другому, объекта)

Класс тоже объект и создаётся с помощью ключевого слова **class** по следующей схеме:

```
class <Название класса>[(<Класс1>[, ..., <КлассN>]]):  
    [""" Строка документирования """]  
    «Описание атрибутов и методов»
```

После названия класса в круглых скобках можно указать один или несколько базовых классов через запятую от которых идёт наследование атрибутов и методов и которые могут добавляться своими атрибутами и методами.

Пример создания класса, внутри которого просто выводится сообщение:

```
# -*- coding: utf-8 -*-
```

```
class MyClass:
```

```
    """ Это строка документирования """
```

```
    print("Инструкции выполняются сразу")
```

```
input()
```

Этот пример содержит лишь определение класса `MyClass` и не создает экземпляр класса. Поэтому когда мы запускаем программу, описывающую только класс, выполнение достигнет инструкции `class` и сообщение, указанное в функции `print()`, будет сразу выведено.

Создание атрибута класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, - с помощью инструкции `def`. Задание аргументов в методе класса в отличие от функции вне тела класса, обязательно должен стоять общепринятый в Python параметр `self`. Именно с помощью этого параметра `self` осуществляется доступ к атрибутам и методам класса. Это значит, что обратиться, например, к атрибуту `x` из метода класса можно таким образом: `self.x`.

Определив класс, можно создавать на его основе множество объектов (экземпляров класса) с помощью значка равно (`=`).

Чтобы добраться до метода класса в экземпляре класса, нужно просто записать имя экземпляра класса, и через точку имя нужного метода. Атрибут класса через экземпляр класса будет доступен также через точку. Сначала записывается имя экземпляра класса и через точку имя нужного атрибута.

Рассмотрим пример класса `MyClass` с атрибутом `x` и методом `print_x()`, выводящим значение этого атрибута, а затем создадим экземпляр класса и вызовем метод

```
class MyClass:
```

```
    def __init__(self): # Конструктор
```

```
        self.x = 10    # Атрибут экземпляра класса
```

```
    def print_x(self):  # self - это ссылка на экземпляр класса
```

```
        print(self.x)  # Выводим значение атрибута
```

```
ic = MyClass()          # Создание экземпляра класса
```

```
                        # Вызываем метод* print_x()
```

```
ic.print_x()           # self не указывается при вызове метода. Выдаст 10
```

```
print(ic.x - 2)        # К атрибуту можно обратиться непосредственно.
```

```
Выдаст 8
```

Попытаемся рассмотреть данный пример в интегрированной среде PyCharm. Она реализована на всех платформах и считается наиболее продвинутой для разработки на Python. Можно свободно скачать и установить её бесплатную версию. Зададим в ней ещё один экземпляр этого класса, например, `ac = MyClass()`, и откроем `PythonConsole`. Что мы увидим? Мы увидим, что экземпляры

этого класса `is` и `as` имеют разные адреса в оперативной памяти. Именно на эти адреса и указывает переменная `self`. Поэтому экземпляры различны.

Существует правило, что название класса пишется с заглавной буквы. Атрибуты в классе это существительные (`point`, `count`, `x`, `y` ...), а методы это действия. Их так и надо называть (`get_count` - получить счёт, `set_point` - установить точку, `empty_value` - очистить переменную ...)

Организуем ещё один класс `Point()` и зададим её характеристики, например, цвет и круг с маленьким диаметром:

```
class Point():
    color = 'red'
    circle = 2
```

Класс имеет собственное пространство имён (атрибутов и методов, в данном классе только атрибуты цвета и диаметра точки) и мы можем обратиться к элементам этого пространства имён и даже изменить их значения.

Выведем первоначальное значение атрибута `color` класса `Point`:

```
print(Point.color) # Выведет red
```

Присвоим этому атрибуту класса `Point()` новое значение:

```
Point.color = 'black'
```

Снова посмотрим значение атрибута `color` класса `Point`:

```
print(Point.color) # Выведет black
```

Значение другого атрибута этого класса можно посмотреть, используя такую же схему написания:

```
print(Point.circle) # Выведет 2
```

Чтобы посмотреть все атрибуты класса нужно записать следующий код:

```
Point.__dict__
```

Выведет:

```
mappingproxy({'__module__': '__main__', 'color': 'black', 'circle': 2, '__dict__': <attribute  
'__dict__' of 'Point' objects>, '__weakref__': <attribute '__weakref__' of 'Point' objects>,  
'__doc__': None})
```

Как видим, присутствуют наши атрибуты `'color': 'black'`, `'circle': 2`.

Создадим два экземпляра этого класса.

```
a = Point()
```

```
b = Point()
```

В этих экземплярах нет ни одного собственного атрибута, но если мы обратимся к атрибутам любого из этих экземпляров класса `Point`, то получим:

```
print(a.color)
# Выведет: black
```

```
print(b.color)
# Выведет: black
```

Атрибут `color` в классе `Point`, даёт нам результат `black` в обоих экземплярах `a` и `b` этого класса.

А теперь проверим наличие собственных атрибутов в экземплярах `a` и `b`, используя специальный метод `__dict__`:

```
print(a.__dict__)
```

```
# Выведет: {}  
print(b.__dict__)  
# Выведет: {}
```

Нет ни одного собственного атрибута в экземплярах a и b.

Как мы видим, словари атрибутов экземпляров класса пусты, но, при обращении к ним, они берутся непосредственно из пространства имён класса Point, из которого они получены. Если же мы присвоим в экземпляре класса a другое значение, например:

```
a.color = 'green'
```

И посмотрим наличие атрибутов в экземпляре a, то получим следующую картину:

```
print(a.__dict__)  
# Выведет не пустой словарь: {'color': 'green'}
```

Появилось в пространстве имён экземпляра класса значение атрибута color, отличное от значения этого атрибута пространства имён самого класса, то есть в экземпляре класса {'color': 'green'}, а в самом классе Point атрибут color остался с прежним значением:

```
print(Point.__dict__)  
{'__module__': '__main__', 'color': 'black', 'circle': 2, '__dict__': <attribute '__dict__' of  
'Point' objects>, '__weakref__': <attribute '__weakref__' of 'Point' objects>, '__doc__':  
None}
```

В Python существуют функции, которые позволяют обращаться к внутренним атрибутам и методам экземпляров класса.

Чтобы прочесть значение какого-либо атрибута, можно воспользоваться функцией `getattr()`. Она отрабатывает по названию этого атрибута. Если имя атрибута отсутствует в экземпляре класса, то возникает ошибка. Чтобы избежать вывода сообщения об ошибке, можно в третьем параметре указать значение, которое будет возвращаться, если атрибут не существует.

Также можно задать значение атрибута, используя функцию `setattr()`. Отсутствие имени атрибута в экземпляре класса не приводит к ошибке, а возникает новый атрибут со своим значением.

Можно проверить наличие атрибута в экземпляре класса. Для этого есть такая функция как `hasattr()`. Она возвращает `True`, если атрибут существует, в противном случае - `False`.

Удаляется атрибут функцией `delattr()`.

Пример с функциями `getattr()`, `setattr()`, `hasattr()`, `delattr()`:

```
class MyClass:  
    def __init__(self):  
        self.x = 10  
    def get_x(self):  
        return self.x + 2
```

```
c = MyClass()
```

```
# Создаем экземпляр класса
```

<code>print(getattr(c, "x"))</code>	<code># Выведет: 10, это значение</code>
атрибута x	
<code>print(c.get_x())</code>	<code># Выведет: 12, это результат</code>
работы метода <code>get_x()</code>	
<code>print(getattr(c, "y", "Атрибут не найден"))</code>	<code># Выведет что: Атрибут не</code>
найден	
<code>setattr(c, "y", 20)</code>	<code># Создаем атрибут y</code>
<code>print(getattr(c, "y", "Атрибут не найден"))</code>	<code># Выведет: 20</code>
<code>delattr(c, "y")</code>	<code># Удаляем атрибут y</code>
<code>print(getattr(c, "y", "Атрибут не найден"))</code>	<code># Выведет что: Атрибут не</code>
найден	
<code>print(hasattr(c, "x"))</code>	<code># Выведет: True</code>
<code>print(hasattr(c, "y"))</code>	<code># Выведет: False</code>

Все атрибуты класса являются открытыми (public) и доступны для изменения как из самого класса, так и из других классов и из основного кода программы.

Методы `__init__()` и `__del__()`

С помощью метода `__init__()` можно присвоить начальные значения атрибутам класса. Этот метод ещё называют Конструктор.

Пример:

```
class MyClass:
    def __init__(self, value1, value2): # Конструктор
        self.x = value1
        self.y = value2
```

```
mc = MyClass(100, 300) # Создаем экземпляр класса
```

```
mc.__dict__
```

```
# Выведет: {'x': 100, 'y': 300}
```

Если конструктор вызывается при создании объекта, то перед уничтожением объекта автоматически вызывается метод Деструктор `__del__()`. Следует заметить, что метод не будет вызван, если на экземпляр класса существует хотя бы одна ссылка. В языке Python не имеет особого смысла. Интерпретатор сам заботится об удалении объектов на которые нет ни одной ссылки.

Рассмотрим подробнее наследование - главное понятие ООП.

При помощи наследования мы можем создать новый класс, в котором будет реализован доступ ко всем атрибутам и методам базового класса.

```
class ClassOwn: # Базовый класс
    def func1(self):
        print ("Метод func1() класса ClassOwn")
    def func2(self):
        print("Метод func2() класса ClassOwn")
```

```
class ClassTwo(ClassOwn): # Класс ClassTwo является наследником от класса ClassOwn
```

```

def func3(self):
    print("Метод func3() класса ClassTwo")
two = ClassTwo() # Создаем экземпляр класса ClassTwo
two.func1() # Выведет: Метод func1() класса ClassOwn
two.func2() # Выведет: Метод func2() класса ClassOwn
two.func3() # Выведет: Метод func3() класса ClassTwo

```

Первые два метода, вызываемые из экземпляра класса two берутся из базового, родительского класса Own. А третий метод экземпляра класса two забирает из собственного класса Two, который является потомком класса Own.

Переопределение методов

Пусть у нас имеется в базовом методе функция (метод) func1(), и мы хотим в производном классе, наследуемым от базового, создать свою функцию (метод) с точно таким же названием как в базовом классе. И чтобы они, эти функции (методы) работали независимо друг от друга и не вызывали ошибку.

```

class ClassOwn:                                # Базовый класс
    def __init__(self):
        print("Конструктор базового класса")
    def func1(self):
        print("Метод func1() класса ClassOwn")

class ClassTwo(ClassOwn):                      # Класс ClassTwo наследуется от класса
    def __init__(self):
        print("Конструктор производного класса")
        ClassOwn.__init__(self) # Вызываем конструктор базового класса
    def func1(self):
        print("Метод func1() класса ClassTwo")
        ClassOwn.func1(self)    # Вызываем метод базового класса
two = ClassTwo()                # Создаем экземпляр класса ClassTwo.
                                # При этом будет выводиться:
                                # Конструктор производного класса
                                # Конструктор базового класса
print(two. func1())             # Выведет:
                                # Метод func1() класса ClassTwo
                                # Метод func1() класса ClassOwn

```

Как видим, при создании экземпляра класса, который был основан на другом классе, сначала запускается конструктор производного класса, а затем конструктор базового класса.

В пространстве имён каждого из этих классов имеются методы с одинаковым названием func1(). И при вызове этого метода экземпляра второго класса, отработает метод из второго класса, а уже потом метод из первого класса. Запишем наследование одного класса (потомка) от другого (родительского) с использованием атрибутов из родительского класса:

```

class A:
    def __init__(self):

```

```
self.x = 10
```

```
class B(A):  
    def __init__(self):  
        self.y = self.x + 5
```

```
print(B().y)  
# Ошибка!
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in __init__

AttributeError: 'B' object has no attribute 'x'

Как видим, получаем ошибку, что в классе потомке не существует объект с атрибутом x.

Воспользовавшись функцией super(), мы можем избежать ошибки.

```
class B(A):  
    def __init__(self):  
        super().__init__() # <- не забудь!  
        self.y = self.x + 5
```

```
print(B().y)  
# Введет: 15
```

В Python метод super() позволяет явно сослаться на родительский класс. Это полезно когда мы хотим вызывать методы непосредственно из родительского класса.

В методе super() не нужно явно передавать указатель self в вызываемый метод.

```
super().__init__() # Вызываем конструктор базового класса
```

Пример. Пусть имеется класс Car (Автомобиль)

```
class Car():  
    #Простая модель автомобиля.  
    def __init__(self, manufacturer, model, year):  
        self.manufacturer = manufacturer  
        self.model = model  
        self.year = year  
        self.odometer_reading = 0  
    def get_descriptive_name(self):  
        long_name = f"{self.year} {self.manufacturer} {self.model}"  
        return long_name.title()  
    def read_odometer(self):  
        #Выводит пробег автомобиля в милях.  
        print(f"У этого автомобиля пробег {self.odometer_reading} миль.")  
    def update_odometer(self, mileage):  
        if mileage >= self.odometer_reading:
```

```

        self.odometer_reading = mileage
    else:
        print("Вы не можете скручивать одометр!")
    def increment_odometer(self, miles):
        self.odometer_reading += miles

```

Поясним этот пример. У класса Car (Автомобиль) в конструкторе класса заложены три атрибута `manufacturer` - фирма производитель, `model` - модель, `year` - год создания. Все переданные значения через конструктор присваиваются переменным с префиксом `self`. Но существует ещё один атрибут, который не был передан конструктору при создании класса. Это - считывание одометра (Прибор, который показывает пробег транспортного средства). Обычно, если это новый автомобиль, то считается, что он без пробега. И условно можно этой величине присвоить значение 0). Метод `get_descriptive_name(self)` - получения полного описания названия автомобиля. С помощью функции `f` будет распечатываться полное названия автомобиля - год создания, производитель и модель.

Метод `read_odometer(self)` - метод чтения показания одометра. Первоначально, как мы видим, показания одометра, заданные при инициализации в конструкторе `__init__()` в классе Car, равно 0. Но, если нам потребуется изменить показания одометра, что ближе к действительности (даже у нового автомобиля может быть пробег, например, за счёт тестовой проверки), то мы можем это сделать через экземпляр класса, используя дополнительный метод `my_new_car(self, miles)`.

```

my_new_car = Car('audi', 'a4', '2019') # Создаём новый экземпляр класса Car.
print(my_new_car.read_odometer()) # Выводит первоначальное значение пробега,
равное 0.

```

```

my_new_car.update_odometer(32) # Изменяем показания одометра
print(my_new_car.read_odometer()) # Выводим новое значение одометра

```

Часто, заботясь о безопасности программы, программисты не позволяют напрямую изменять значение атрибута, типа `my_new_car.odometer_reading = 32`. Пишут новый метод, который и изменяет показания пробега - `update_odometer(self, mileage)`. Кроме изменения показателя пробега, этот метод проверяет вводимое значение пробега на предмет сброса существующего значения. И, в случае не нарушения изменения этого показателя в меньшую сторону, он записывается, в атрибут `self.odometer_reading`.

И ещё в этом классе Car() добавлен метод `increment_odometer(self, miles)`. Он прибавляет к существующему пробегу новое значение. Этот метод может пригодиться в том случае, когда вы купили подержанный автомобиль с пробегом и проехали на нём 200 миль. И этот показатель прибавиться к уже имеющемуся значению пробега. При желании в этот метод может быть добавлен анализ ввода отрицательного значения, чтобы предотвратить скручивание пробега (как в методе `update_odometer()`).

Теперь на основании класса Car(), описывающего автомобиль вообще, создадим новый класс со специфическими свойствами, которых нет в классе Car(). Пусть

это будет класс электрического автомобиля. И назовём этот класс EelecnicCar и он будет строиться на основании класса Car().

Car() - класс-родитель. EelecnicCar(Car) - класс-потомок.

При создании класса-потомка, класс-родитель должен находиться в том же файле, что и класс-потомок, и предшествовать классу-потомку. То есть описывается сначала класс-родитель, а затем класс-потомок.

Содержание нового файла **car1.py** (Тексты программ и данных располагаются в zip файле по адресу: pageforweb.ru/lessons_py.zip):

```
class Car():
    #Простая модель автомобиля.
    def __init__(self, manufacturer, model, year):
        self.manufacturer = manufacturer
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()
    def read_odometer(self):
        #Выводит пробег автомобиля в милях.
        print(f"У этого автомобиля пробег {self.odometer_reading} миль.")
    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("Вы не можете скручивать одометр!")
    def increment_odometer(self, miles):
        self.odometer_reading += miles
```

class EelecnicCar(Car): # В этом классе появляются новые аспекты, характеризующие специфику электроавтомобиля.

```
    def __init__(self, manufacturer, model, year):
        super().__init__(manufacturer, model, year) # Инициализация атрибутов
        класса-родителя.
```

```
my_tesla = EelecnicCar('tesla', 'model s', 2019)
```

```
print(my_tesla.get_descriptive_name())
```

```
# Выведет: 2019 Tesla Model S
```

Экземпляр my_tesla класса EelecnicCar работает так же как и экземпляр класса-родителя Car. Теперь мы можем приступить к добавлению атрибутов и методов, специфичных для электроавтомобиля. Для начала добавим атрибут мощности батареи аккумулятора и метода вывода его показателя. Наш файл с описанием класса-родителя Car и класса-потомка EelecnicCar будет теперь иметь вид:

car2.py

```

class Car():
    #Простая модель автомобиля.
    def __init__(self, manufacturer, model, year):
        self.manufacturer = manufacturer
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()
    def read_odometer(self):
        #Выводит пробег автомобиля в милях.
        print(f"У этого автомобиля пробег {self.odometer_reading} миль.")
    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("Вы не можете скручивать одометр!")
    def increment_odometer(self, miles):
        self.odometer_reading += miles

class ElectricCar(Car): # В этом классе появляются новые аспекты,
    характеризующие специфику электроавтомобиля.
    def __init__(self, manufacturer, model, year): # Инициализируем атрибуты
        класса-родителя. Затем инициализируем атрибуты класса-потомка,
        специфические для электромобиля
        super().__init__(manufacturer, model, year)
        self.battery_size = 80
    def describe_battery(self): # Выводит информацию о мощности
        аккумулятора.
        print(f'У этого автомобиля мощность аккумулятора {self.battery_size}
— кВт.')
    def fill_gas_tank(self):
        # У электромобилей нет бензобака!
        print("У электромобилей нет бензобака!")

```

```
my_tesla = ElectricCar('tesla', 'model s', 2019)
```

```
print(my_tesla.get_descriptive_name())
```

```
# Выведет: 2019 Tesla Model S
```

```
my_tesla.describe_battery()
```

```
# Выведет: У этого автомобиля мощность аккумулятора 80 - кВт.
```

Программный код класса ElectricCar описывает специфику, относящуюся только к электромобилям.

Как видим в классе-родителе Car имеется метод вывода ёмкости бензинового бака `fill_gas_tank()`. Такой показатель для электромобилей не имеет смысла. Поэтому этот метод в классе `ElectricCar` переопределён так, чтобы выводилось сообщение, что такого показателя у электромобиля нет.

Любой метод родительского класса, который в наследуемом классе делает не то, что нужно, можно переопределить. Для этого в классе-потомке определяется метод с тем же именем, что и у метода класса-родителя. Python игнорирует метод родителя и обращает внимание только на метод, определенный в потомке. В классе-потомке добавляются новые атрибуты и методы, необходимых для того, чтобы потомок отличался от родителя. Так были добавлены новый атрибут, показывающий величину мощности аккумулятора и метод для вывода информации об этом атрибуте.

Экземпляры как атрибуты

Чтобы классы не дополнялись большим количеством атрибутов и методов, часть одного класса можно записать в виде отдельного независимого класса.

Например, в классе `ElectricCar` слишком много атрибутов и методов, относящихся к аккумулятору. В таком случае можно переместить все эти атрибуты и методы в отдельный класс с именем `Battery`. Затем экземпляр `Battery` становится атрибутом класса `ElectricCar`:

`car3.py`

```
class Car():
    #Простая модель автомобиля.
    def __init__(self, manufacturer, model, year):
        self.manufacturer = manufacturer
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()
    def read_odometer(self):
        #Выводит пробег автомобиля в милях.
        print(f"У этого автомобиля пробег {self.odometer_reading} миль.")
    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("Вы не можете скручивать одометр!")
    def increment_odometer(self, miles):
        self.odometer_reading += miles

class Battery():
    #Простая модель аккумулятора электромобиля.
    def __init__(self, battery_size=75):
```

```

        #Инициализирует атрибуты аккумулятора.
        self.battery_size = battery_size
    def describe_battery(self):
        #Выводит информацию о мощности аккумулятора.
        print(f"У этого автомобиля аккумулятор {self.battery_size} кВт..")

```

```

class ElectricCar(Car):
    #Представляет аспекты машины, специфические для электромобилей.
    def __init__(self, manufacturer, model, year):
        #Инициализирует атрибуты класса-родителя.
        #Затем инициализирует атрибуты, специфические для
электромобиля.
        super().__init__(manufacturer, model, year)
        self.battery = Battery()

```

```

my_tesla = ElectricCar('tesla', 'model', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()

```

Импортирование классов

Python позволяет хранить классы в модулях и импортировать нужные классы в основную программу.

Импортирование одного класса

Пусть создаётся модуль с одним только классом Car.

Создадим файл **car.py**.

```

class Car():
    #Простая модель автомобиля.
    def __init__(self, manufacturer, model, year):
        #Инициализирует атрибуты описания автомобиля.
        self.manufacturer = manufacturer
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        #Возвращает отформатированное описание.
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()
    def read_odometer(self):
        #Выводит пробег машины в милях.
        print(f"У этого автомобиля пробег {self.odometer_reading} миль.")
    def update_odometer(self, mileage):
        #Устанавливает на одометре заданное значение.
        #При попытке обратной подкрутки изменение отклоняется.
        if mileage >= self.odometer_reading:

```

```

        self.odometer_reading = mileage
    else:
        print("Вы не можете скручивать одометр!")
def increment_odometer(self, miles):
    #Увеличивает показания одометра с заданным приращением.
    self.odometer_reading += miles

```

Теперь мы создадим отдельный файл с именем `my_car.py`. Из этого файла импортируем класс `Car` и создадим экземпляр этого класса:

```

my_car.py
from car import Car
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
# Выведет: 2019 Audi A4
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
# Выведет: У этого автомобиля пробег 23 миль.

```

У нас получилось 2 файла `car.py` и `my_car.py`. Выйдем из интерпретатора Python и запустим на исполнение файл `my_car.py`:

```

python3 my_car.py

```

При запуске этого файла импортируется родительский класс `Car` из файла `car.py`, который теперь является модулем `car` и записывается без расширения `.py`. После отработки программного кода будет выведена следующая информация:

```

2019 Audi A4
У этого автомобиля пробег 23 миль.

```

Хранение нескольких классов в модуле

В предыдущем примере в одном файле (модуле) у нас хранился только один класс `Car`, который был использован в другой программе с именем `my_car.py`. Но в одном модуле можно хранить сколько угодно классов, хотя все эти классы должны быть каким-то образом связаны друг с другом. Оба класса, `Battery` и `ElectricCar`, используются для представления автомобилей, поэтому мы добавим их в модуль `car_com.py`:

```

car_com.py
#Классы для представления машин с бензиновым и электродвигателем.
class Car():
    #Простая модель автомобиля.
    def __init__(self, manufacturer, model, year):
        #Инициализирует атрибуты описания автомобиля.
        self.manufacturer = manufacturer
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        #Возвращает отформатированное описание.
        long_name = f"{self.year} {self.manufacturer} {self.model}"

```

```

        return long_name.title()
    def read_odometer(self):
        #Выводит пробег машины в милях.
        print(f"У этого автомобиля пробег {self.odometer_reading} миль.")
    def update_odometer(self, mileage):
        #Устанавливает на одометре заданное значение.
        #При попытке обратной подкрутки изменение отклоняется.
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("Вы не можете скручивать одометр!")
    def increment_odometer(self, miles):
        #Увеличивает показания одометра с заданным приращением.
        self.odometer_reading += miles

```

```

class Battery():
    #Простая модель аккумулятора электромобиля.
    def __init__(self, battery_size=70):
        #Инициализация атрибутов аккумулятора.
        self.battery_size = battery_size
    def describe_battery(self):
        #Выводит информацию о мощности аккумулятора.
        print(f"This car has a {self.battery_size}-kWh battery.")
    def get_range(self):
        #Выводит приблизительный запас хода для аккумулятора.
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315
        print(f"Этот автомобиль может проехать около {range} миль на
полной зарядке.")

```

```

class ElectricCar(Car):
    #Представляет аспекты машины, специфические для электромобилей.
    def __init__(self, manufacturer, model, year):
        #Инициализирует атрибуты класса-родителя.
        #Затем инициализирует атрибуты, специфические для
электромобиля.
        super().__init__(manufacturer, model, year)
        self.battery = Battery()

```

В одном файле car.py содержатся 3 класса. Родительский класс Car(), класс-потомок ElectricCar(Car) и класс Battery(), входящий атрибутом в класс ElectricCar(Car).

Можно импортировать класс ElectricCar и создать новый экземпляр электромобиля:

```
from car_com import ElectricCar
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

Выведет:

Tesla Model S 2019

Этот автомобиль имеет аккумулятор емкостью 75 кВт*ч.

Этот автомобиль может проехать около 260 миль на полной зарядке.

Большая часть логики программы скрыта в модуле.

Импортирование нескольких классов из модуля

В файл программы можно импортировать столько классов, сколько потребуется. Если вы захотите создать обычный автомобиль и электромобиль в одном файле, потребуется импортировать оба класса, Car и ElectricCar:

[electric_car.py](#)

```
from car_com import Car, ElectricCar
my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())
my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

Выведет:

2019 Volkswagen Beetle

2019 Tesla Roadster

Чтобы импортировать несколько классов из модуля, нужно разделить их имена запятыми. После того как необходимые классы будут импортированы, можно создать столько экземпляров каждого класса, сколько потребуется.

В этом примере создается обычный автомобиль Volkswagen Beetle и электромобиль Tesla Roadster.

Импортирование всего модуля

Также возможно импортировать весь модуль, а потом обращаться к нужным классам с использованием точечной записи. Этот способ прост, а полученный код легко читается. Так как каждый вызов, создающий экземпляр класса, включает имя модуля, в программе не будет конфликтов с именами, используемыми в текущем файле.

[electric_car_2.py](#)

```
import car_com
my_beetle = car_com.Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())
my_tesla = car_com.ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

Выведет:

2019 Volkswagen Beetle

2019 Tesla Roadster

Импортируется весь модуль `car_com`, после чего программа обращается к нужным классам с использованием синтаксиса `имя_модуля.имя_класса`. Затем создается экземпляр Volkswagen Beetle, а потом — экземпляр Tesla Roadster.

Импортирование всех классов из модуля

Для импортирования всех классов из модуля используется следующий синтаксис:

```
from имя_модуля import *
```

Использовать этот способ не рекомендуется, так как возможны конфликты с именами Ваших классов и именами импортируемых классов, про которые ничего не известно. Что стоит под `*` мы не знаем.

Итак, если вам нужно импортировать большое количество классов из модуля, лучше импортировать весь модуль и воспользоваться синтаксисом `имя_модуля.имя_класса`. Хотя вы не видите перечень всех используемых классов в начале файла, по крайней мере ясно видно, где модуль используется в программе. Также предотвращаются потенциальные конфликты имен, которые могут возникнуть при импортировании каждого класса в модуле.

Импортирование модуля в модуль

Чтобы избежать чрезмерного разрастания одного файла и хранения несвязанных классов в одном модуле, классы приходится распределять по нескольким модулям. Допустим, что один класс из одного модуля зависит от класса из другого модуля. В этом поможет импортирование необходимого класса одного модуля в другой модуль.

Рассмотрим пример.

Классу `ElectricCar` необходим доступ к классу-родителю `Car`, поэтому класс `Car` импортируется прямо в модуль. Также необходимо обновить модуль `car`, чтобы он содержал только класс `Car`:

```
car_mod.py
```

```
#Простая модель автомобиля.
```

```
class Car():
```

```
    #Простая модель автомобиля.
```

```
    def __init__(self, manufacturer, model, year):
```

```
        #Инициализирует атрибуты описания автомобиля.
```

```
        self.manufacturer = manufacturer
```

```
        self.model = model
```

```
        self.year = year
```

```
        self.odometer_reading = 0
```

```
    def get_descriptive_name(self):
```

```
        #Возвращает отформатированное описание.
```

```
        long_name = f"{self.year} {self.manufacturer} {self.model}"
```

```
        return long_name.title()
```

```
    def read_odometer(self):
```

```
        #Выводит пробег машины в милях.
```

```
        print(f"У этого автомобиля пробег {self.odometer_reading} миль.")
```

```

def update_odometer(self, mileage):
    #Устанавливает на одометре заданное значение.
    #При попытке обратной подкрутки изменение отклоняется.
    if mileage >= self.odometer_reading:
        self.odometer_reading = mileage
    else:
        print("Вы не можете скручивать одометр!")
def increment_odometer(self, miles):
    #Увеличивает показания одометра с заданным приращением.
    self.odometer_reading += miles

```

electric_car_mod.py

```

class Battery():
    #Простая модель аккумулятора электромобиля.
    def __init__(self, battery_size=75):
        #Инициализирует атрибуты аккумулятора.
        self.battery_size = battery_size
    def describe_battery(self):
        #Выводит информацию о мощности аккумулятора.
        print(f"У этого автомобиля аккумулятор {self.battery_size} кВт..")

class ElectricCar(Car):
    #Представляет аспекты машины, специфические для электромобилей.
    def __init__(self, manufacturer, model, year):
        #Инициализирует атрибуты класса-родителя.
        #Затем инициализирует атрибуты, специфические для
электромобиля.
        super().__init__(manufacturer, model, year)
        self.battery = Battery()

```

Теперь можно импортировать классы из каждого модуля по отдельности и создать ту разновидность машины, которая вам нужна:

my_cars_mod.py

```

from car_mod import Car
from electric_car_mod import ElectricCar
my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())
my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
# Выведет:

```

2019 Volkswagen Beetle

2019 Tesla Roadster

Класс Car импортируется из своего модуля, а класс ElectricCar - из своего. После этого создаются экземпляры обеих разновидностей.

Использование псевдонимов

Псевдонимы уменьшают написание кода. Например, имя класса ElectricCar повторять много раз утомительно. В команде import можно назначить псевдоним этому имени с более коротким написанием:

```
from electric_car import ElectricCar as EC
```

Теперь можно использовать этот псевдоним каждый раз вместо имени ElectricCar:

```
my_tesla = EC('tesla', 'roadster', 2019)
```

Оформление классов

Имена классов должны записываться в «верблюжьей» нотации. Первая буква каждого слова записывается в верхнем регистре, слова не разделяются пробелами.

Имена экземпляров и модулей записываются в нижнем регистре с разделением слов символами подчеркивания.

Урок 7 часть 2

Множественное наследование

Python, в отличие от некоторых других языков программирования, поддерживает множественное наследование. В этом случае один класс-потомок может наследовать атрибуты и методы от нескольких классов-родителей. Записывается такое наследование в описании создаваемого класса как и в случае наследования от одного класса, только в скобках указываются имена базовых классов через запятую.

```
class ИмяКласса(БазовыйКласс_1, БазовыйКласс_2, ..., БазовыйКласс_N)
```

Пример:

```
class ClassOwn: # Базовый класс
```

```
    def func1(self):
```

```
        print("Метод func1() класса ClassOwn")
```

```
    # В базовом классе ClassOwn определён только один метод func1().
```

```
class ClassThu(ClassOwn): # Класс ClassThu наследует класс ClassOwn
```

```
    def func2(self):
```

```
        print("Метод func2() класса ClassThu")
```

```
class ClassThree(ClassOwn): # Класс ClassThree наследует класс ClassOwn
```

```
    def func1(self):
```

```
        print("Метод func1() класса ClassThree")
```

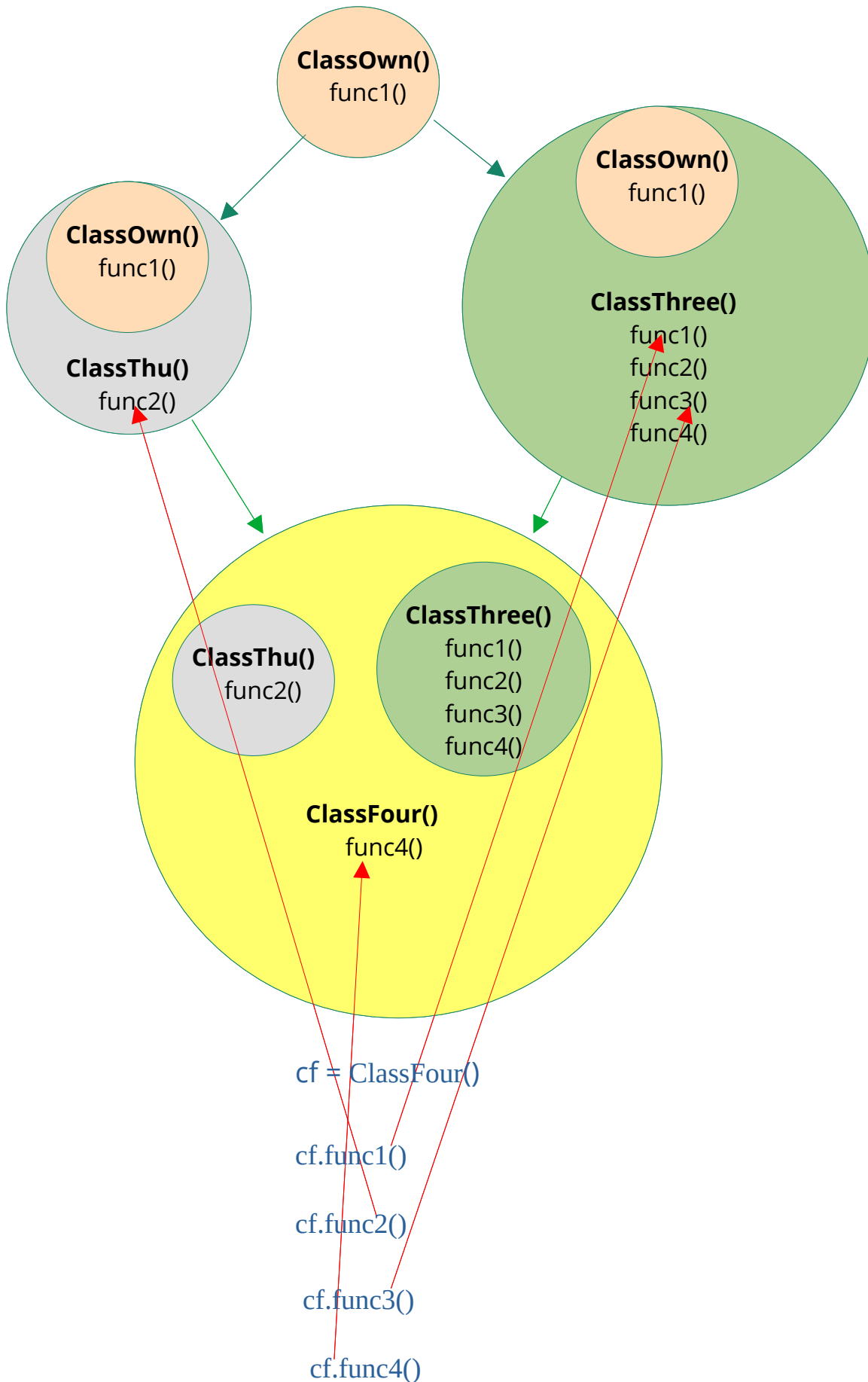
```
    def func2(self):
```

```
        print("Метод func2() класса ClassThree")
```

```
    def func3(self):
```

```
print("Метод func3() класса ClassThree")
def func4(self):
    print("Метод func4() класса ClassThree")
```

```
class ClassFour(ClassThu, ClassThree): # Множественное наследование
```



Когда мы создаём экземпляр класса `Class.Four()` и называем его `cf`, мы можем обращаться к различным методам, как внутри данного класса, так и к методам, находящимся в других классах, которые являются базовыми для класса `ClassFour()`.

Как видим в базовых классах есть методы с одинаковыми названиями. Существует правило по которому вызов метода, находящегося в базовых классах будет осуществляться только от того базового класса, который находится в непосредственной близости от класса-потомка.

Так метод `func1()`, определенный в классах `ClassOwn()` и `ClassThree()`, будет найден в классе `ClassThree()`, так как он ближайший к классу `Class.Four()`, метод `func1()` класса `ClassOwn()` не находится непосредственно вблизи к рассматриваемому классу.

Второе правило: вызов метода из двух и более базовых классов, определённых как классы от которых идёт множественное наследование класса-потомка, зависит от порядка записи классов-родителей. Какой класс-родитель записан первым в скобках, тот и будет выбран первым, при совпадении имён методов, к которым обращается класс-потомок.

Вот почему метод `func2()`, определенный в двух классах `ClassThu()` и `ClassThree()`, будет выбран из класса `ClassThu()`.

Чтобы наследовать метод `func2()` из класса `ClassThree()`, следует указать это явным образом:

```
class ClassFour(ClassThu, ClassThree): # Множественное наследование
    # Наследуем func2() из класса ClassThree, а не из класса ClassThu
    func2 = ClassThree.func2
    def func4(self):
        print("Метод func4() класса ClassFour")
```

В предыдущем примере метод `func3()` определен только в классе `ClassThree`, поэтому метод наследуется от этого класса.

Метод `func4()`, определенный в классе `ClassThree`, переопределяется в классе-потомке `ClassFour()` и вся иерархия наследования от классов-родителей просматриваться не будет.

Для получения полной картины перечня базовых классов можно воспользоваться атрибутом `__bases__`. Он возвращает кортеж. Если наследование идёт только от одного класса-родителя, то кортеж будет состоять только из одного элемента.

Надо не забывать, что даже самый первый класс-родитель, является тоже наследуемым классом от абстрактного класса `object()`.

Значение атрибута `__bases__` для всех классов из предыдущего примера:

```
print(ClassOwn.__bases__)
# Выведет: (<class 'object'>,)
print(ClassThu.__bases__)
# Выведет: (<class '__main__.ClassOwn'>,)
print(ClassThree.__bases__)
# Выведет: (<class '__main__.ClassOwn'>,,)
```

```
print(ClassFour.__bases__)  
# Выведет: (<class '__main__.ClassThu '>, <class '__main__.ClassThree '>)
```

Поиск идентификаторов при множественном наследовании

```
class ClassOwn:
```

```
    x = 10
```

```
class ClassThu(ClassOwn):  
    pass
```

```
class ClassThree(ClassThu):  
    pass
```

```
class ClassFour(ClassThree):  
    pass
```

```
class ClassFive(ClassThu):  
    pass
```

```
class ClassSix(ClassFive):  
    pass
```

```
class ClassSeven(ClassFour, ClassSix):  
    pass
```

```
f7 = ClassSeven()
```

```
print(f7.x)
```

```
# Выведет: 10
```

Примечание: запускать предыдущий программный код в терминальном режиме рекомендуется со вставкой пустой строки после описания каждого класса. Иначе, при копировании всего кода в терминальный режим, будут возникать ошибки.

Атрибута `x` в классе `ClassSeven` нет. Создав экземпляр этого класса `f7` и обратившись к атрибуту `x`, Python будет искать его в классах-родителях.

Последовательность поиска атрибута `x` будет такой:

`ClassSeven -> ClassFour -> ClassThree -> ClassSix -> ClassFive -> ClassThu -> ClassOwn`

Почему класс `ClassSeven` обращается для поиска атрибута `x` к классу `ClassFour`, а не к классу `ClassSix`? Потому что класс `ClassFour` стоит первым в перечне классов-родителей класса `ClassSeven`.

Получить всю цепочку наследования позволяет атрибут `__mro__`:

```
print(ClassSeven.__mro__)
```

Результат выполнения:

```
(<class '__main__.ClassSeven '>, class '__main__.ClassFour '>,  
<class '__main__.ClassThree '>, class '__main__.ClassSix '>,  
<class '__main__.ClassFive '>, class '__main__.ClassThu '>,
```

```
<class '__main__.ClassOwn '>, class 'object'>)
```

Миксины (примеси)

Миксины - это особый вид множественного наследования в Python.

Допустим необходимо расширить функциональность уже существующего класса.

Классический подход в этом случае - создание подкласса и добавление новой функциональности в него. Однако, в некоторых случаях такой подход может быть неэффективным, например, когда необходимо добавить одну и ту же функциональность к нескольким классам. В такой ситуации миксины могут оказаться наиболее подходящим решением. Для самостоятельного использования миксины не предназначены. Методы, определенные в миксине, можно использовать в любом другом классе, который наследуется от миксина.

```
class Mixin:
```

```
    def mixin_method(self):  
        return "Вызван миксин метод"
```

```
class MyClass(Mixin):  
    pass
```

```
my_ex = MyClass()  
print(my_ex.mixin_method())  
# Выведет: Вызван миксин метод
```

Класс MyClass наследуется от класса Mixin. Класс MyClass не имеет своих атрибутов и методов, но экземпляр my_ex класса MyClass благополучно отработает mixin_method метод из класса Mixin.

В случае с миксинами, необходимую функциональность можно просто «подмешать» в наследуемый класс вместе с основным базовым классом. Это позволяет избежать дублирования кода и позволяет его повторное использование.

Миксины позволяют избежать проблем, связанных с множественным наследованием, так как они не имеют базовых классов.

Пример использования миксинов.

Пусть имеется интернет-магазин, торгующий электроникой по продаже ноутбуков, дисков, процессоров и т.п. Каждый товар определяться своим классом и общим (базовым) для всех классов: товары - класс Goods. Обязательные поля для всех классов:

name – наименование;

weight – вес;

price – цена .

Сначала запишем базовый класс Goods:

```
class Goods:  
    def __init__(self, name, weight, price):  
        print("init MixinLog")  
        self.name = name  
        self.weight = weight
```

```
        self.price = price
    def print_info(self):
        print(f"{self.name}, {self.weight}, {self.price}")
```

И один дочерний класс, например, для ноутбуков:

```
class Notebook(Goods):
    pass
```

Затем, мы можем создать объект для ноутбука:

```
n = Notebook("Acer", 1.5, 30000)
```

и распечатать информацию о нем:

```
n.print_info()
```

```
# Выведет: Acer, 1.5, 30000
```

В классе Notebook пока нет ни атрибутов, ни методов, поэтому экземпляр класса Notebook() будет брать атрибуты и методы из класса-родителя Goods.

Рассмотрим такую ситуацию. Вышел указ программистам добавить возможность изменения цен на товары. Допустим, применить скидки на товары, в частности, на ноутбуки.

Прописать логику изменение цен можно непосредственно в базовом классе Goods(). Но это плохой сценарий, так как нужно изменять всё время базовый класс Goods(). Лучше, в этом случае использовать миксины.

Создадим ещё один класс:

```
class Goods:
    def __init__(self, name, weight, price):
        print("init MixinLog")
        self.name = name
        self.weight = weight
        self.price = price
    def print_info(self):
        print(f"{self.name}, {self.weight}, {self.price}")
```

```
class MixinLog:
    def __init__(self, discount):
        print("init MixinLog")
        self.discount = discount
    def save_sum(self):
        self.discount = 20
        print(f"Товар уценён на {self.discount}%, окончательная стоимость {self.price - (self.price/100*self.discount)}")
```

```
class Notebook(Goods, MixinLog):
    pass
```

```
m = Notebook("Acer", 1.5, 30000)
```

```
m.save_sum()
```

```
# Выведет: Товар уценён на 20%, окончательная стоимость 24000.0
```

Этот класс работает совершенно независимо от классов Goods и Notebook и лишь добавляет функционал изменения цен товаров. Это и есть так называемые классы примеси — миксины.

В этом примере при изменении скидок можно менять только класс миксин.

MRO – Method Resolution Order (Порядок разрешения метода)

В Python существует специальный алгоритм обхода базовых классов при множественном наследовании. Сокращенно, он называется MRO.

Этот метод определяет в каком порядке обходить базовые классы:

Мы можем увидеть эту цепочку обхода базовых классов, если распечатать специальную коллекцию `__mro__` любого класса:

```
print(NoteBook.__mro__)
```

В консоли появится следующая последовательность:

```
(<class '__main__.NoteBook'>, <class '__main__.Goods'>, <class '__main__.MixinLog'>, <class 'object'>)
```

Если поменять местами базовые классы класса Notebook:

```
class NoteBook(MixinLog, Goods):  
    pass
```

То получим ошибку, так как в метод `__init__` передаются четыре аргумента, а он принимает только один от класса MixinLog. Так что порядок следования базовых классов при множественном наследовании имеет важное значение. Первым должен идти «основной» класс и у него, как правило, инициализатор имеет несколько параметров. А далее, записываются классы, у которых, опять же, как правило, инициализаторы имеют только параметр `self`.

Примеси активно применяются в различных дополнительных библиотеках - в частности, в популярной библиотеке Web-программирования Django.

Магические методы

Их ещё называют волшебные, специальные или дандер (dunder) методы.

Магический метод - это метод, название которого начинается и заканчивается двойным подчеркиванием. С некоторыми магическими методами мы уже сталкивались:

`__new__()` Используется для создания объектов классов

`__init__()` Проводит инициализацию объектов экземпляров классов

Ещё можно рассмотреть магические методы представления строк:

`__repr__()` Выводит строковое представление объектов для разработчика

`__str__()` Выводит строковое представление объектов для пользователя

Пара других магических методов:

`__len__()` Поддерживает функцию `len()` - подсчёта количества элементов в строке, в списках, кортежах, словарях, множествах и т. д.

`__abs__()` Поддерживает функцию `abs()` - нахождения абсолютного значения переменной.

Это маленькая часть всех магических методов.

Можно, конечно, напрямую вызывать магические методы, например,

`(10).__add__(8)` — получим 18 (`__add__()` магический метод сложения целых чисел), но надо знать, что они не предназначены для прямого использования. Python

сам вызывает их автоматически в ответ на операцию сложения. В данном случае $10 + 8$, и ответ будет такой же: 18.

Заметим, что хотя методы и называются магическими методами, но в сообществе Python часто можно встретить другое название. В официальной документации вместо этого называет их специальными методами.

Не желательно использовать двойные подчёркивания с двух сторон в именовании своих переменных. Ничего хорошего из этого не получится. И выполняться они не будут, и другие программисты не поймут такую конструкцию кода.

Рассмотрим примеры этих специальных функций.

Методы `__new__()` и `__init__()` уже рассматривались при создании классов и объектов классом.

Специальный метод `__str__()` отработывает, когда вызывается метод `str()` для отображения экземпляров класса. Ещё этот специальный метод может вызываться при использовании методов `print()` и `format()`. Пример:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f"Меня зовут {self.name}, и мне {self.age} лет."
```

В том классе метод `__str__()` возвращает удобное для пользователя строковое представление данных о человеке.

Посмотрим, как работает данный класс:

```
maxim = Person("Максим", 12)
str(maxim)
# Выведет: 'Меня зовут Максим, и мне 12 лет.'
print(maxim)
# Выведет: Меня зовут Максим, и мне 12 лет.
```

Такое представление специально создано для пользователей вашего кода.

Для программистов используется специальный метод `__repr__()`.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __repr__(self):
        return f"{type(self).__name__}(name='{self.name}',
age={self.age})"
```

Как видно, в `__repr__()` используется f-строка для представления объекта.

Сначала представляется имя класса. Затем в круглых скобках записываются аргументы конструктора и их значения. Получим экземпляр класса:

```
maxim = Person("Максим", 12)
repr(maxim)
#Выведет : "Person(name='Максим', age=12)"
```

Python вызывает специальные методы автоматически в ответ на заданную операцию. Все доступные специальные методы поддерживают встроенные функции, например, встроенные типы, такие как списки, строки и словари, реализуют большую часть своей основной функциональности с помощью специальных методов. В своих пользовательских классах можно использовать специальные методы при создании объектов, их способов сравнения, настройки объектов и многого другого. Но следует избегать именования пользовательских методов с использованием двойного подчеркивания в начале и конце. Когда мы создаём свой пользовательский класс, мы сталкиваемся первым делом с наиболее распространенным методом, которым является `__init__()`. Этот метод позволяет вам предоставлять начальные значения любым атрибутам экземпляра, которые должны быть определены в этих классах. Метод `__init__()` называют конструктором класса. Цель метода `__init__()` - инициализировать любой атрибут экземпляра вашего класса. Рассмотрим пример:

Class Point:

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

`point = Point(21, 42)` # Создание экземпляра класса Point

`point.x`

Выведет: 21

`point.y`

Выведет: 42

Python автоматически вызывает `__init__()` при создании экземпляра класса Point, который принимает те же значения, которые были переданы при создании экземпляра класса.

Ещё существует специальный метод `__new__()`.

Этот метод обрабатывает когда нужно создать новый экземпляр самого класса.

Метод `__new__()` реже реализуется на практике. Но в некоторых случаях он незаменим. Например, есть неизменяемые типы, такие как `int`, `float`, `tuple`, `str`.

Рассмотрим следующий пример класса, который наследует встроенный тип данных `float`:

class Storage(float):

```
def __new__(cls, value, unit):
    instance = super().__new__(cls, value)
    instance.unit = unit
    return instance
```

Заметим, что при использовании `__new__()`, который является методом класса, а не конструктором экземпляра класса. Потому что он получает в качестве аргумента текущий класс (`cls`), а не текущий экземпляр (`self`). Сначала создаётся новый экземпляр текущего класса, `cls` вызывая `__new__()` этот `float` класс через встроенную `super()` функцию. Этот вызов создает новый экземпляр `float` и инициализирует его, используя `value` в качестве аргумента. Затем настраивается новый экземпляр, динамически прикрепляя к нему `.unit` атрибут. Далее

возвращается новый экземпляр класса, по умолчанию соответствующий поведению `__new__()`.

На практике это выглядит так:

```
disk = Storage(1024, "GB")
```

disk # Будет соответствовать поведению по умолчанию

Выдаст: 1024.0

```
disk.unit # Новое поведение
```

Выдаст: 'GB'

Числовое значение `disk` неизменяемо, как и его родительский тип, `float`.

Атрибут `.unit` является изменяемым и его можно изменять в любое время.

Наряду с `__init__()`, методы `__str__()` и `__repr__()`, возможно, являются наиболее часто используемыми специальными методами в пользовательских классах.

Поддержка перегрузки операторов в пользовательских классах

В Python можно переопределить операторы, то есть изменить операции, которые он выполняет. Это называется перегрузкой операторов. Один и тот же оператор в Python по-разному ведет себя с разными типами. Например, оператор `+` в зависимости от типа операндов может суммировать 2 числа, сливать 2 списка или объединять 2 строки.

Перегрузка операторов означает предоставление операторам дополнительной функциональности.

Рассмотрим конкретные специальные методы, поддерживающие операторы Python такие как арифметические действия, операции сравнения, наличия в различных структурах Python, побитовые операторы и другие.

Арифметические операторы

Список арифметических операторов в Python и поддерживающих их специальных методов:

Оператор	Вспомогательный метод
----------	-----------------------

+	<code>__add__(self, other)</code>
---	-----------------------------------

-	<code>__sub__(self, other)</code>
---	-----------------------------------

*	<code>__mul__(self, other)</code>
---	-----------------------------------

/	<code>__truediv__(self, other)</code>
---	---------------------------------------

//	<code>__floordiv__(self, other)</code>
----	--

%	<code>__mod__(self, other)</code>
---	-----------------------------------

**	<code>__pow__(self, other[, modulo])</code>
----	---

Второй аргумент `other`. В большинстве случаев этот аргумент должен быть того же или совместимого типа, что и `self`. Иначе будет ошибка. Как можно перегрузить некоторые из этих операторов? Рассмотрим на примере класса `Storage`. Допустим, мы хотим убедиться, что при добавлении двух экземпляров этого класса оба операнда имеют одинаковую единицу измерения.

storage.py

```
class Storage(float):
```

```
    def __new__(cls, value, unit):
```

```

        instance = super().__new__(cls, value)
        instance.unit = unit
        return instance
    def __add__(self, other):
        if not isinstance(other, type(self)):
            raise TypeError(
                "неподдерживаемый операнд для +: "
                f"'{type(self).__name__}' and '{type(other).__name__}'"
            )
        if not self.unit == other.unit:
            raise TypeError(
                f"incompatible units: '{self.unit}' and '{other.unit}'"
            )
        return type(self)(super().__add__(other), self.unit)

```

В описание класса Storage включён специальный метод `__add__()`. В этом методе сначала проверяется `other` на предмет является ли он также экземпляром `Storage`. Если нет, то создаётся `TypeError` исключение с соответствующим сообщением об ошибке. Затем проверяется, имеют ли оба объекта одинаковую единицу измерения. Если нет, то снова создаётся `TypeError`. Если проверки прошли успешно, то создаётся и возвращается новый класс `Storage`, в котором добавляются два значения и присоединяется единица измерения.

Загрузим снова интерпретатор Python и проверим работу следующего кода:

```

from storage import Storage
disk_1 = Storage(500, "GB")
disk_2 = Storage(1000, "GB")
disk_3 = Storage(1, "TB")
disk_1 + disk_2
# Выведет: 1500.0

```

```

disk_2 + disk_3

```

Выведет:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 13, in __add__

TypeError: incompatible units: 'GB' and 'TB'

```

disk_1 + 100

```

Выведет:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 8, in __add__

TypeError: неподдерживаемый операнд для +: 'Storage' and 'int'

Как видно, при добавлении двух объектов с одинаковой единицей измерения, получаем правильное значение. Иначе ошибка `TypeError` (единицы измерения)

несовместимы). И действие `disk_1 + 100` тоже влечёт за собой ошибку, так как попытка добавить `Storage` экземпляру `Storage` встроенный в Python числовой тип `'int'` не являющийся экземпляром класса `Storage`.

В приведенном выше примере мы перегрузили оператор сложения (+) для встроенного `float` типа. Можно использовать `.__add__()` метод и любой другой арифметический метод для поддержки соответствующего оператора в пользовательском классе.

Рассмотрим следующий пример `Distance` класса и запишем его в файл.

`distance.py`

```
class Distance:
```

```
    _multiples = {
        "mm": 0.001,
        "cm": 0.01,
        "m": 1,
        "km": 1000,
    }
    def __init__(self, value, unit="m"):
        self.value = value
        self.unit = unit.lower()
    def to_meter(self):
        return self.value * type(self)._multiples[self.unit]
    def __add__(self, other):
        return self._compute(other, "+")
    def __sub__(self, other):
        return self._compute(other, "-")
    def _compute(self, other, operator):
        operation = eval(f"{self.to_meter()} {operator} {other.to_meter()}")
        cls = type(self)
        return cls(operation / cls._multiples[self.unit], self.unit)
```

В `Distance` есть непубличный атрибут класса с именем `._multiples`. Этот атрибут содержит словарь единиц длины и соответствующие им коэффициенты преобразования. В `.__init__()` методе инициализируются атрибуты `.value` и `.unit` в соответствии с определением пользователя. Метод `str.lower()` нужен для нормализации строки `unit` и принудительного использования строчных букв. В `.to_meter()` вы используете `._multiples` для выражения текущего расстояния в метрах. Метод `.to_meter()` можно использовать в качестве вспомогательного средства в `.__add__()` и `.__sub__()` методах. Таким образом, такой класс сможет правильно суммировать или вычитать расстояния в разных единицах измерения.

Методы `.__add__()` и `.__sub__()` поддерживают операторы сложения и вычитания соответственно. В обоих методах вы используете `._compute()` вспомогательный метод для выполнения вычислений.

В `._compute()` мы принимаем `other` и `operator` в качестве аргументов. Потом используем встроенную `eval()` функцию для вычисления выражения и получения

предполагаемого результата для текущей операции. Функция `eval()` используется для вычисления выражений, записанных в виде строк. Например, `eval('2 + 3')` вернет результат 5. Однако, `eval()` не может выполнять сложные конструкции, такие как циклы, условные операторы и т.д. Затем создаём экземпляр `Distance` с помощью встроенной `type()` функции. Наконец, возвращаем новый экземпляр класса с вычисленным значением, преобразованным в единицу текущего экземпляра.

Вот как класс `Distance` работает на практике:

```
from distance import Distance
distance_1 = Distance(200, "m")
distance_2 = Distance(1, "km")
total = distance_1 + distance_2
print(total.value)
# Выведет: 1200.0
print(total.unit)
# Выведет: m
displacement = distance_2 - distance_1
print(displacement.value)
# Выведет: 0.8
print(displacement.unit)
# Выведет: km
```

Как видим класс `Distance` поддерживает операторы сложения и вычитания.

Подробнее об арифметических операторах

На специальные методы, поддерживающие операторы, влияет относительное положение каждого объекта в содержащем их выражении. Вот почему `displacement` в приведенном выше разделе указано в километрах, а `total` - в метрах.

Python здесь вызывает `__add__()` метод для левого операнда. Если этот операнд не реализует метод, то операция завершится неудачей. Если поменять операнды местами, то реализация этого метода будет вести себя не так, как бы хотелось. К счастью, в Python есть правосторонняя версия операторных методов (`__r*__()`) для решения этих проблем. Пример:

```
class A():
    def __init__(self):
        self.value = 10
    def __add__(self, other):
        return "Выполняется метод __add__."
    def __radd__(self, other):
        return "Выполняется метод __radd__."
    def __iadd__(self, other):
        self.value += other
        return self
    def __str__(self):
        return f"value: {self.value}."
```

```

a = A()
print(a + 1)
# Выведет: Выполняется метод __add__.
print(1 + a)
# Выведет: Выполняется метод __radd__.
a += 1
print(a)
# Выведет: value: 11.
a += 1
print(a)
# Выведет: value: 12.

```

Вот краткое описание `.__r*__()` методов:

Оператор Метод правой руки

```

+      .__radd__(self, other)
-      .__rsub__(self, other)
*      .__rmul__(self, other)
/      .__rtruediv__(self, other)
//     .__rfloordiv__(self, other)
%      .__rmod__(self, other)
**     .__rpow__(self, other[, modulo])

```

Python вызывает эти методы, когда левый операнд не поддерживает соответствующую операцию, а операнды имеют разные типы.

В Python также есть несколько унарных операторов. Он называет их унарными, потому что они работают с одним операндом:

Оператор Вспомогательный метод Описание

```

-      .__neg__(self)      Возвращает целевое значение с обратным знаком
+      .__pos__(self)      Обеспечивает дополнение к отрицанию без выполнения
каких-либо преобразований

```

Можно перегружать эти операторы во встроенные числовые типы Python и поддерживать их в своих собственных классах.

Методы операторов сравнения

За операторами сравнения стоят специальные методы. Например, когда мы сравниваем `5 < 2`, Python вызывает `.__lt__()`. Краткое описание всех операторов сравнения и поддерживающих их методов:

Оператор Вспомогательный метод

```

<      .__lt__(self, other)
<=     .__le__(self, other)
==     .__eq__(self, other)
!=     .__ne__(self, other)
>=     .__ge__(self, other)
>      .__gt__(self, other)

```

Чтобы продемонстрировать, как можно использовать некоторые из этих методов в пользовательских классах, рассмотрим класс `Rectangle` для создания

нескольких разных прямоугольников, чтобы иметь возможность сравнивать эти прямоугольники.

Вот Rectangle класс, который поддерживает операторы equality (==), less than (<) и greater than (>):

rectangle.py

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width
    def area(self):
        return self.height * self.width

    def __eq__(self, other):
        return self.area() == other.area()

    def __lt__(self, other):
        return self.area() < other.area()

    def __gt__(self, other):
        return self.area() > other.area()
```

В этом классе есть .area() метод, который вычисляет площадь прямоугольника, используя его высоту и ширину. Затем у вас есть три необходимых специальных метода для поддержки предполагаемых операторов сравнения. Все они используют площадь прямоугольника для определения конечного результата. Вот как этот класс работает на практике. Сравним площадь баскетбольной площадки (basketball_court) с площадью футбольного поля (soccer_field):

```
from rectangle import Rectangle
basketball_court = Rectangle(15, 28)
soccer_field = Rectangle(75, 110)
basketball_court < soccer_field
# Выведет: True
basketball_court > soccer_field
# Выведет: False
basketball_court == soccer_field
# Выведет: False
```

Операторы членства

В Python есть два оператора, которые позволяют определить, находится ли данное значение в коллекции значений. Операторами являются in и not in. Они поддерживают проверку, известную как тест на членство.

Например, предположим, нужно определить, находится ли число в списке чисел:

```
2 in [2, 3, 5, 9, 7]
# Выведет: True
10 in [2, 3, 5, 9, 7]
# Выведет: False
```

В первом примере число 2 находится в списке чисел, поэтому получаем True. Во втором примере число 10 отсутствует в списке, и будет выведено False. Оператор `not in` работает аналогично `in`, но как отрицание. С помощью этого оператора можно узнать, нет ли данного значения в коллекции.

Специальный метод, который поддерживает операторы членства `in` и `not in` является `__contains__()`. Реализуя этот метод в пользовательском классе, можно контролировать, как его экземпляры будут вести себя при тестировании членства. Чтобы проиллюстрировать, как поддерживать тесты членства в пользовательском классе, рассмотрим пример. Предположим, нужно реализовать базовую структуру данных стека, которая поддерживает обычные операции `push` и `pop`, а также операторы `in` и `not in`.

`stack.py`

```
class Stack:
    def __init__(self, items=None):
        self.items = list(items) if items is not None else []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def __contains__(self, item):
        for current_item in self.items:
            if item == current_item:
                return True
        return False
```

В этом примере описывается класс `Stack` с методами `.push()` и `.pop()`. Первый метод добавляет новый элемент в верхнюю часть стека, в то время как второй удаляет и возвращает элемент в верхней части стека.

Затем `__contains__()` метод принимает целевой элемент в качестве аргумента и использует цикл `for`, чтобы определить, есть ли элемент в списке, в котором хранятся данные стека. Вместо цикла можно использовать оператор `in`.

```
from stack import Stack
stack = Stack([2, 3, 5, 9, 7])
stack.__contains__(10)
# Выведет: False
stack.push(10)
stack.__contains__(10)
# Выведет: True
stack.pop()
# Выведет: 10 и удалит его из списка, так как это число добавилось последним в
список
stack.__contains__(10)
# Выведет: False
```

Побитовые операторы

Побитовые операторы Python позволяют манипулировать отдельными битами данных на самом низком уровне. С помощью этих операторов можно выполнять побитовые операции AND, OR, XOR, NOT и битового сдвига. Эти операторы также реализуются с помощью специальных методов.

Побитовые операторы Python и поддерживающие их специальные методы:

Оператор Вспомогательный метод

& .__and__(self, other)
| .__or__(self, other)
^ .__xor__(self, other)
<< .__lshift__(self, other)
>> .__rshift__(self, other)
~ .__invert__()

С помощью этих специальных методов можно заставить пользовательские классы поддерживать побитовые операторы.

[bitwise_number.py](#)

```
class BitwiseNumber:
    def __init__(self, value):
        self.value = value
    def __and__(self, other):
        return type(self)(self.value & other.value)
    def __or__(self, other):
        return type(self)(self.value | other.value)
    def __xor__(self, other):
        return type(self)(self.value ^ other.value)
    def __invert__(self):
        return type(self)(~self.value)
    def __lshift__(self, places):
        return type(self)(self.value << places)
    def __rshift__(self, places):
        return type(self)(self.value >> places)
    def __repr__(self):
        return bin(self.value)
```

Чтобы реализовать требуемые методы в пользовательском классе

BitwiseNumber можно использовать побитовые операторы, соответствующие данному методу. Методы .__and__(), .__or__() и .__xor__() работают со значением текущего BitwiseNumber экземпляра и далее other. Этот второй операнд должен быть другим экземпляром BitwiseNumber.

Метод .__invert__() поддерживает побитовый оператор NOT, который является унарным оператором, поскольку он действует с одним операндом.

Следовательно, этот метод не принимает второй аргумент. Наконец, два метода, которые поддерживают операторы побитового сдвига, принимают аргумент с именем places . Этот аргумент представляет количество мест, на которые нужно сдвинуть бит в любом направлении.

В BitwiseNumber находится `__repr__()` метод предоставляет информацию, как работает класс.

```
from bitwise_number import BitwiseNumber
five = BitwiseNumber(5)
print(five)
# Выведет: 0b101
ten = BitwiseNumber(10)
print(ten)
# Выведет: 0b1010
```

```
five & ten
# Выведет: 0b0
```

```
five | ten
# Выведет: 0b1111
```

```
five ^ ten
# Выведет: 0b1111
```

```
~five
# Выведет: -0b110
```

```
five << 2
# Выведет: 0b10100
```

```
ten >> 1
# Выведет: 0b101
```

Расширенные задания

Если использовать арифметические операции для обновления самой переменной. Классический пример этой операции, позволяющий обновить счетчик, записывается следующим образом:

```
counter = 0
counter = counter + 1
counter = counter + 1
counter
# Выведет: 2
```

Вторая и третья строки в этом фрагменте кода обновляют значение счетчика, используя предыдущее значение. Этот тип операций настолько распространен в программировании, что в Python для него есть сокращение, известное как расширенные операторы присваивания. Сокращённая запись выглядит так:

```
counter = 0
counter += 1
counter += 1
print(counter)
```

Выведет: 2

Оператор (+=) не единственный, который поддерживает Python. Вот краткое описание поддерживаемых операторов в математическом контексте:

Оператор Вспомогательный метод

```
+=    .__iadd__(self, other)
-=    .__isub__(self, other)
*=    .__imul__(self, other)
/=    .__itruediv__(self, other)
//=   .__ifloordiv__(self, other)
%=    .__imod__(self, other)
**=   .__ipow__(self, other[, modulo])
```

Побитовые операторы также имеют расширенные вариации:

Оператор Вспомогательный метод

```
&=    .__iand__(self, other)
|=    .__ior__(self, other)
^=    .__ixor__(self, other)
<<=   .__ilshift__(self, other)
>>=   .__irshift__(self, other)
```

Опять же, можно использовать эти методы для поддержки соответствующих расширенных побитовых операций в собственных классах.

Самоанализ объектов

Можно использовать некоторые специальные методы для поддержки самоанализа в пользовательских классах. Например, можно управлять поведением объекта при его проверке с помощью встроенных функций, таких как `dir()`, `isinstance()` и других.

Специальные методы для самоанализа

Метод	Описание
-------	----------

<code>__dir__()</code>	Возвращает список атрибутов и методов объекта.
<code>__hasattr__()</code>	Проверяет, имеет ли объект определенный атрибут.
<code>__instancecheck__()</code>	Проверяет, является ли объект экземпляром определенного класса.
<code>__subclasscheck__()</code>	Проверяет, является ли класс подклассом определенного класса.

Реализуя эти специальные методы в своих классах, можно обеспечивать пользовательское поведение и контролировать процесс самоанализа объектов. В качестве примера можно добавить `__dir__()` метод в `Rectangle` класс из предыдущего:

`rectangle1.py`

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width
    def area(self):
        return self.height * self.width
```

```

def __eq__(self, other):
    return self.area() == other.area()

def __lt__(self, other):
    return self.area() < other.area()

def __gt__(self, other):
    return self.area() > other.area()

# ...

def __dir__(self):
    print("__dir__ called")
    return sorted(self.__dict__.keys())

```

В этой реализации `__dir__()` мы возвращаем только имена текущих атрибутов экземпляра `Rectangle` класса. Вот как ведет себя этот класс, когда мы используем один из его экземпляров в качестве аргумента встроенной `dir()` функции:

```

from rectangle1 import Rectangle
dir(Rectangle(12, 24))
# Выведет: __dir__ called
# ['height', 'width']

```

Как видно, когда передаётся экземпляр `Rectangle` функции `dir()`, получается список, содержащий имена всех атрибутов экземпляра этого класса. Это поведение отличается от поведения по умолчанию, которое возвращает все атрибуты и методы.

Управление доступом к атрибутам

За каждым доступом к атрибутам, их назначением и удалением найдутся специальные методы, которые поддерживают конкретную операцию. Вот эти методы и их краткое описание:

`__getattr__(self, name)` запускается при обращении к атрибуту, называемому `name`.

`__getattribute__(self, name)` запускается при обращении к атрибуту, которого нет в текущем объекте.

`__setattr__(self, name, value)` выполняется, когда идёт присвоение `value` атрибуту `name`.

`__delattr__(self, name)` запускается при удалении атрибута `name`.

Их можно использовать для проверки и настройки способов доступа, установки и удаления атрибутов в пользовательских классах.

Извлечение атрибутов

В Python есть два метода для обработки доступа к атрибутам. `__getattribute__()` Метод выполняется при каждом доступе к атрибуту, в то время как `__getattr__()` метод выполняется только в том случае, если целевой атрибут не существует в текущем объекте.

[circle.py](#)

```

import math
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def __getattribute__(self, name):
        print(f"__getattribute__ called for {name}")
        return super().__getattribute__(name)

    def __getattr__(self, name):
        print(f"__getattr__ called for {name}")
        if name == "diameter":
            return self.radius * 2
        return super().__getattr__(name)

```

В `__getattribute__()` методе распечатывается сообщение только для определения того, когда Python вызвал метод. Затем делегируется доступ к атрибуту родительского класса с помощью встроенной `super()` функции. Python вызывает этот метод в ответ на каждый доступ к атрибуту.

Далее идёт `__getattr__()` метод. Там распечатывается сообщение для идентификации вызова метода. Условно можно сказать, что проверяется вызван ли атрибут, к которому осуществляется доступ, "diameter". Если это так, то вычисляется диаметр и возвращается его значение. В противном случае делегируется доступ к атрибуту родительского класса.

Примечание: Класс `object` является родительским или базовым классом всех классов Python, включая пользовательские. Этот класс предоставляет реализации нескольких специальных методов по умолчанию.

Проверим работу класса `Circle`:

```

from circle import Circle
circle = Circle(10)
print(circle.radius)
# Выведет: __getattribute__ called for 'radius'
# 10
print(circle.diameter)
# Выведет: __getattribute__ called for 'diameter'
# __getattr__ called for 'diameter'
# __getattribute__ called for 'radius'
# 20

```

Когда обращаемся к `.radius` используя точечную нотацию, Python неявно вызывает `__getattribute__()`. Следовательно, получаем соответствующее сообщение и значение атрибута. Python вызывает этот метод при каждом доступе к атрибуту.

Затем происходит доступ к `.diameter`. Этому атрибута не существует в `Circle`. Сначала вызывается Python `__getattribute__()`. Поскольку этот метод не находит `.diameter` атрибут, Python продолжает вызывать `__getattr__()`. Чтобы

вычислить значение диаметра, снова обращается к `.radius`. Вот почему появляется последнее сообщение.

Настройка атрибутов

Обычно оператор присваивания используется для присвоения нового значения данному атрибуту. При этом вызывается специальный метод `__setattr__()`. С помощью этого метода можно настроить некоторые аспекты процесса присвоения. Предположим, что необходимо подтвердить, что радиус целое положительное число.

`circle1.py`

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def __getattribute__(self, name):
        print(f"__getattribute__ called for {name}")
        return super().__getattribute__(name)

    def __getattr__(self, name):
        print(f"__getattr__ called for {name}")
        if name == "diameter":
            return self.radius * 2
        return super().__getattr__(name)
    # ...
    def __setattr__(self, name, value):
        if name == "radius":
            if not isinstance(value, int):
                raise TypeError("Радиус должен быть числом")
            if value <= 0:
                raise ValueError(" Радиус должен иметь положительное значение")
        super().__setattr__(name, value)
```

Метод `__setattr__()` позволяет настроить поведение пользовательского класса в контексте задания присвоения нового значения атрибута.

В этом примере мы проверяем, назван ли целевой атрибут "radius". Если это так, то проверяется является ли предоставленное значение целым числом. Затем проверяется, что это число больше 0. В обоих случаях создается соответствующее исключение и распечатывается сообщение об ошибке. И всё это назначение делегируется родительскому классу.

```
from circle1 import Circle
circle = Circle(10)
print(circle.radius)
# Выведет: __getattribute__ called for radius
# 10
```

```
print(circle.radius)
# Выведет: __getattr__ called for radius
# 20
circle.radius = "42"
# Выведет: Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# ...
# in __setattr__
# raise TypeError("Радиус должен быть числом")
# TypeError: Радиус должен быть числом
```

```
circle.diameter = 42
print(circle.diameter)
# Выведет: __getattr__ called for diameter
# 42
```

Когда вызывается конструктор класса Circle() мы получаем сообщение, указывающее, что `__setattr__()` был вызван. Это потому, что `__init__()` метод выполняет присваивание.

Затем мы присваиваем новое значение `.radius`, и Python вызывает `__setattr__()` снова. Когда мы пытаемся присвоить строку для `.radius`, мы получаем `TypeError`, потому что входное значение не является допустимым числом.

Удаление атрибутов

Иногда нужно удалить данный атрибут из объекта после его использования.

Если мы хотим иметь точный контроль над тем, как наши классы реагируют на удаление атрибута с помощью `del` инструкции, то мы можем использовать специальный метод `__delattr__()`.

Например, мы хотим создать класс, запрещающий удаление атрибутов:

`non_deletable.py`

```
class NonDeletable:
    def __init__(self, value):
        self.value = value
    def __delattr__(self, name):
        raise AttributeError(f"{type(self).__name__} не поддерживает удаление
атрибутов")
```

У этого маленького класса есть атрибут с именем `.value`. Мы переопределяем метод `__delattr__()` и создаём исключение всякий раз, когда кто-то пытается удалить атрибут в этом классе.

```
from non_deletable import NonDeletable
one = NonDeletable(1)
print(one.value)
# Выведет: 1
del one.value
# Выведет:
#Traceback (most recent call last):
```

```
# ...
```

#AttributeError: NonDeletable не поддерживает удаление атрибутов

Когда кто-то пытается использовать `del` оператор для удаления `.value` атрибута, получается исключение `AttributeError`. Метод `__delattr__()` будет перехватывать все попытки удалить атрибуты экземпляра в этом классе.

Управление атрибутами с помощью дескрипторов

Дескрипторы Python - это классы, которые позволяют иметь контроль над атрибутами пользовательского класса. Для этого можно использовать дескрипторы.

Дескрипторы состоит из следующих методов:

Метод	Описание
-------	----------

<code>__get__(self, instance, type=None)</code>	Метод Getter позволяет извлекать текущее значение атрибута
---	--

<code>__set__(self, instance, value)</code>	Метод Setter позволяет установить новое значение атрибута
---	---

<code>__delete__(self, instance)</code>	Метод Deleter, позволяет удалить атрибут, содержащийся в этом классе
---	--

<code>__set_name__(self, owner, name)</code>	Метод установки имени, который позволяет определить имя для управляемого атрибута
--	---

Чтобы проиллюстрировать, для чего хороши дескрипторы, предположим, что у нас есть `shapes.py` модуль. Мы определили классы `Circle`, `Square` и `Rectangle`. Нам нужно проверять такие параметры, как радиус круга, сторона квадрата и так далее. Логика проверки является общей для всех этих параметров.

В этой ситуации мы можем использовать дескриптор для управления логикой проверки, которая заключается в проверке того, является ли предоставленное значение целым положительным числом. Вот возможная реализация этого дескриптора:

`shapes.py`

```
class PositiveNumber:
```

```
    def __set_name__(self, owner, name):
```

```
        self._name = name
```

```
    def __get__(self, instance, owner):
```

```
        return instance.__dict__[self._name]
```

```
    def __set__(self, instance, value):
```

```
        if not isinstance(value, int) or value <= 0:
```

```
            raise ValueError("ожидается целое положительное число")
```

```
        instance.__dict__[self._name] = value
```

В этом классе мы сначала определяем специальный метод `__set_name__()`.

Аргумент `owner` представляет содержащий класс, а `name` аргумент представляет имя атрибута.

Затем мы определяем метод `__get__()`. Этот метод принимает экземпляр и ссылку на содержащий его класс. Внутри этого метода мы используем специальный атрибут `__dict__` для доступа к пространству имен экземпляра и извлекаем значение управляемого атрибута.

Наконец, мы определяем метод `__set__()`, который принимает экземпляр содержащего его класса и новое значение для управляемого атрибута. В этом методе мы проверяем, что предоставленное значение не является целым положительным числом, и в этом случае мы создаём `ValueError` исключение. В противном случае мы снова присваиваем входное значение управляемому атрибуту, используя `__dict__` пространство имен.

Теперь можно использовать этот дескриптор для определения параметров наших фигур:

`shapes1.py`

```
import math
```

```
class PositiveNumber:
```

```
    def __set_name__(self, owner, name):
        self._name = name
```

```
    def __get__(self, instance, owner):
        return instance.__dict__[self._name]
```

```
    def __set__(self, instance, value):
        if not isinstance(value, int | float) or value <= 0:
            raise ValueError("ождается целое положительное число")
        instance.__dict__[self._name] = value
```

```
class Circle:
```

```
    radius = PositiveNumber()
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return round(math.pi * self.radius**2, 2)
```

```
class Square:
```

```
    side = PositiveNumber()
    def __init__(self, side):
        self.side = side
    def area(self):
        return round(self.side**2, 2)
```

```
class Rectangle:
```

```
    height = PositiveNumber()
    width = PositiveNumber()
    def __init__(self, height, width):
        self.height = height
        self.width = width
    def area(self):
        return self.height * self.width
```

В этом коде выделенные строки показывают, как вы создаете управляемые атрибуты с помощью `PositiveNumber` дескриптора. Обратите внимание, что

управляемые атрибуты должны быть атрибутами класса с аналогом атрибута экземпляра.

Вот как работают наши фигуры:

```
from shapes1 import Circle, Square, Rectangle
circle = Circle(-10)
# Выведет: Traceback (most recent call last):
#...
# raise ValueError("ожидается целое положительное число")
#ValueError: ожидаемое положительное число
square = Square(-20)
# Выведет: Traceback (most recent call last):
#...
# raise ValueError("ожидается целое положительное число")
#ValueError: ожидаемое положительное число
```

```
rectangle = Rectangle(-20, 30)
# Выведет: Traceback (most recent call last):
#...
# raise ValueError("ожидается целое положительное число")
#ValueError: ожидаемое положительное число
```

Все наши формы проверяют входной параметр при создании экземпляра. Это потому, что все они используют наш PositiveNumber класс для управления своими параметрами. Таким образом, мы повторно используем код проверки во всех своих классах.

Поддержка итерации с помощью итераторов и итерируемых объектов

В Python итераторы и итерируемые объекты - это два разных, но связанных инструмента, которые пригодятся, когда нужно выполнить итерацию по потоку данных или контейнеру. Итераторы обеспечивают управление процессом итерации, в то время как итерируемые объекты обычно содержат данные, которые можно перебирать.

Итераторы и итерируемые объекты являются фундаментальными компонентами в программировании на Python. Они используются прямо или косвенно практически во всех программах.

Создание итераторов

Для создания итератора в Python существуют два специальных метода. Реализуя эти методы, можно получить контроль над процессом итерации. С помощью них мы можем определять, как именно Python извлекает элементы, когда мы используем класс в for цикле или другой итерационной конструкции.

Методы, составляющие итератор:

Метод	Описание
__iter__()	Вызывается для инициализации итератора. Он должен возвращать объект итератора.
__next__()	Вызывается для выполнения итерации. Он должен возвращать следующее значение в потоке данных.

В `__iter__()` обычно возвращает `self` текущий объект. В `__next__()` необходимо последовательно возвращать следующее значение из потока данных. Этот метод должен вызывать `StopIteration` когда поток данных исчерпан. Таким образом, Python знает, что итерация подошла к концу.

Реализуя эти два метода в пользовательских классах, мы превращаем их в итераторы. Например, создадим класс, который предоставляет итератор чисел в последовательности Фибоначчи.

`fibonacci.py`

```
class FibonacciIterator:
```

```
    def __init__(self, stop=10):
        self._stop = stop
        self._index = 0
        self._current = 0
        self._next = 1
    def __iter__(self):
        return self
    def __next__(self):
        if self._index < self._stop:
            self._index += 1
            fib_number = self._current
            self._current, self._next = (
                self._next,
                self._current + self._next,
            )
            return fib_number
        else:
            raise StopIteration
```

В этом классе `FibonacciIterator` сначала инициализируется несколько атрибутов. Атрибут `._stop` определяет количество значений, которые будет возвращать наш класс. По умолчанию оно равно 10. Атрибут `._index` содержит индекс текущего значения Фибоначчи в последовательности. Атрибуты `._current` и `._next` представляют соответственно текущее и следующее значения в последовательности Фибоначчи.

В `__iter__()` мы просто возвращаем текущий объект, `self`. Эта практика распространена при создании итераторов в Python.

Метод `__next__()` немного более сложный. В этом методе у нас есть условие, которое проверяет, не достиг ли текущий индекс последовательности значения `._stop`, и в этом случае увеличивается текущий индекс. Затем вычисляем число Фибоначчи, соответствующее текущему индексу, возвращая результат вызывающей стороне `__next__()`.

Когда `._index` достигает значения `._stop`, создаём исключение `StopIteration`, которое завершает процесс итерации.

Теперь наш класс готов к повторению.

```
from fibonacci import FibonacciIterator
```

```

for fib_number in Fibonacciliterator():
    print(fib_number)
0
1
1
2
3
5
8
13
21
34

```

Класс `Fibonacciliterator` вычисляет новые значения в режиме реального времени, выдавая значения по запросу, когда мы используем класс в цикле. Когда количество значений Фибоначчи вырастает до 10, класс создает `StopIteration` исключение, и Python завершает цикл.

Создание итерируемых объектов

Итерируемые объекты немного отличаются от итераторов. Обычно коллекция или контейнер являются итерируемыми, когда они реализуют специальный метод `__iter__()`.

Например, встроенные в Python типы контейнеров - такие как списки, кортежи, словари и множества - являются итеративными объектами. Они предоставляют поток данных, который позволяет выполнять итерации. Однако они не предоставляют `__next__()` метод, который управляет процессом итерации. Для выполнения итерации по итерируемому объекту с помощью `for` цикла Python неявно создает итератор.

`stack_iter.py`

```

class Stack:
    def __init__(self, items=None):
        self.items = list(items) if items is not None else []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def __contains__(self, item):
        for current_item in self.items:
            if item == current_item:
                return True
        return False
    def __iter__(self):
        return iter(self.items[::-1])

```

Эта реализация `__iter__()` не возвращает текущий объект, `self`. Вместо этого метод возвращает итератор по элементам в стеке, используя встроенную `iter()` функцию. Функция `iter()` предоставляет возможность итерации входных данных с

помощью метода по умолчанию `__next__()`, что позволяет ей быть полноценным итератором.

Перед итерацией мы меняем список элементов в стеке на противоположный, чтобы соответствовать порядку извлечения элементов из стека.

```
from stack_iter import Stack
```

```
stack = Stack([1, 2, 3, 4])
```

```
for value in stack:
```

```
    print(value)
```

```
4
```

```
3
```

```
2
```

```
1
```

В этом фрагменте сначала создаётся новый экземпляр `Stack` с четырьмя числами. Затем запускается цикл `for` над экземпляром. Цикл выводит текущее значение на каждой итерации. Это показывает, что элементы `order` будут выталкиваться при повторном вызове `.pop()`. Класс `Stack` теперь поддерживает итерацию.

В Python вызываемый объект - это любой объект, который можно вызвать, используя пару круглых скобок и, необязательно с аргументами. Например, функции, классы и методы являются вызываемыми в Python. Вызываемые функции являются фундаментальными во многих языках программирования, потому что они позволяют выполнять действия.

Python позволяет создавать собственные вызываемые функции. Для этого можно добавить специальный метод `__call__()` в свой класс. Любой экземпляр класса с `__call__()` методом ведет себя как функция.

Обычный вариант использования вызываемых экземпляров - это когда нужен вызываемый объект с сохранением состояния, который кэширует вычисленные данные между вызовами. Это будет удобно, когда необходимо оптимизировать некоторые алгоритмы. Например, можно создать класс `Factorial`, который кэширует уже вычисленные значения для повышения эффективности.

`factorial.py`

```
class Factorial:
```

```
    def __init__(self):
```

```
        self._cache = {0: 1, 1: 1}
```

```
    def __call__(self, number):
```

```
        if number not in self._cache:
```

```
            self._cache[number] = number * self(number - 1)
```

```
        return self._cache[number]
```

В этом примере определяется `Factorial` с помощью `__call__()` метода. В `__init__()` методе мы создаём и инициализируем `_cache` атрибут для хранения уже вычисленных значений.

Далее мы определяем `__call__()`. Python автоматически вызывает этот метод, когда мы используем экземпляр содержащего его класса в качестве функции. В этом примере метод вычисляет факториал заданного числа, кэшируя вычисленные значения в `_cache`.

Как работает этот класс покажем на примере:

```
from factorial import Factorial
factorial_of = Factorial()
factorial_of(4)
# Выведет: 24
factorial_of(5)
# Выведет: 120
factorial_of(6)
# Выведет: 720
```

В этом примере кода создаётся экземпляр Factorial. Затем этот экземпляр используется как функция для вычисления факториала различных чисел. Реализуя `__call__()` специальный метод в этом классе, мы заставляем их вести себя как вызываемые объекты, которые ведут себя как функции, когда мы вызываем их с помощью пары круглых скобок. Это позволяет добавить гибкости и функциональности нашему объектно-ориентированному коду.

Реализация пользовательских последовательностей и сопоставлений

Последовательности и сопоставления в Python являются фундаментальными встроенными типами данных. Списки, кортежи и строки являются примерами последовательностей, в то время как словари являются примерами типов отображения.

Можно создавать свои собственные классы, подобные последовательностям и отображениям, реализуя необходимые специальные методы. С этой целью Python определяет протоколы последовательности и отображения. Эти протоколы представляют собой наборы специальных методов.

Например, протокол sequence состоит из следующих методов:

Метод	Описание
<code>__getitem__()</code>	Вызывается, когда мы обращаемся к элементу с помощью индексации, как в <code>sequence[index]</code>
<code>__len__()</code>	Вызывается при вызове встроенной <code>len()</code> функции для получения количества элементов в базовой последовательности
<code>__contains__()</code>	Вызывается, когда мы используем последовательность в тесте членства с помощью оператора <code>in</code> or <code>not in</code>
<code>__reversed__()</code>	Вызывается, когда мы используем встроенную функцию <code>reversed()</code> для получения обратной последовательности.

Реализуя эти специальные методы, мы можем создавать пользовательские классы в Python, похожие на последовательности.

`stack1.py`

```
class Stack:
    def __init__(self, items=None):
        self.items = list(items) if items is not None else []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
```

```

def __contains__(self, item):
    for current_item in self.items:
        if item == current_item:
            return True
    return False

# ...
def __getitem__(self, index):
    return self.items[index]
def __len__(self):
    return len(self.items)
def __reversed__(self):
    return type(self)(reversed(self.items))

```

В `__getitem__()` методе мы возвращаем целевой элемент из внутреннего контейнера данных, который является обычным `list`. С помощью этого метода поддерживается операция индексации.

Затем определяется `__len__()` для поддержки встроенной `len()` функции. Этот метод возвращает количество элементов во внутреннем списке данных.

Наконец, определяется `__reversed__()`. Этот метод позволяет получить обратную версию исходных данных, используя встроенную функцию `reversed()`.

```

from stack1 import Stack
stack = Stack([1, 2, 3, 4])
stack[1]
# Выведет: 2
stack[-1]
# Выведет: 4
len(stack)
# Выведет: 4
stack1 = reversed(stack)
stack1[0:4]
# Выведет: [4, 3, 2, 1]

```

Теперь `stack` объект поддерживает индексирование с помощью `[index]` оператора. Он также поддерживает встроенные функции `len()` и `reversed()`. Обратите внимание, что при вызове `reversed()` с `Stack` экземпляром в качестве аргумента получается новый `Stack` экземпляр вместо списка - `stack1`. Это поведение отличается от поведения по умолчанию `reversed()`, которое обычно возвращает обратный итератор.

Настройка и демонтаж с помощью контекстных менеджеров

Инструкция Python `with` весьма полезна для управления внешними ресурсами в пользовательских программах. Это позволяет воспользоваться контекстными менеджерами для автоматического выполнения этапов настройки и демонтажа при использовании внешних ресурсов.

Например, встроенная `open()` функция возвращает файловый объект, который также является контекстным менеджером. Обычно эту функцию используют в `with` инструкции для передачи управления базовому контекстному менеджеру,

который представляет собой контекст времени выполнения от открытия файла до момента закрытия его по завершении работы. Эти процессы включают в себя настройку и демонтаж соответственно.

Вот пример использования `open()` для работы с текстовым файлом:

```
with open("hello.txt", mode="w", encoding="utf-8") as file:
```

```
    file.write("Hello, World!")
```

```
# Выведет: 13
```

```
with open("hello.txt", mode="r", encoding="utf-8") as file:
```

```
    print(file.read())
```

```
# Выведет: Hello, World!
```

Первая `with` инструкция открывает файл `hello.txt` для записи. В блоке инструкции `with` в файл записывается текст `"Hello, World!"`. После записи блока Python закрывает файл `hello.txt`, чтобы освободить полученный ресурс.

Примечание: Число 13 в выходных данных соответствует возвращаемому значению `.write()` и представляет количество символов, которые было записано в файл. Это число видно только когда код запускался в интерактивном режиме. Вторая инструкция открывает файл для чтения и выводит его содержимое в окно терминала.

Если нужно создать `context manager` или добавить функциональность `context manager` в существующий класс, то нужно закодировать два специальных метода:

Метод	Описание
-------	----------

<code>__enter__()</code>	Настраивает контекст среды выполнения, получает ресурсы и может возвращать объект, который можно привязать к переменной с помощью <code>as</code> спецификатора в <code>with</code> заголовке
--------------------------	---

<code>__exit__()</code>	Очищает контекст среды выполнения, освобождает ресурсы, обрабатывает исключения и возвращает логическое значение, указывающее, следует ли распространять какие-либо исключения, которые могут возникнуть в контексте.
-------------------------	---

`reader.py`

```
class TextFileReader:
```

```
    def __init__(self, file_path, encoding="utf-8"):
```

```
        self.file_path = file_path
```

```
        self.encoding = encoding
```

```
    def __enter__(self):
```

```
        self.file_obj = open(self.file_path, mode="r", encoding=self.encoding)
```

```
        return self.file_obj
```

```
    def __exit__(self, exc_type, exc_val, exc_tb):
```

```
        self.file_obj.close()
```

Этот класс принимает путь к файлу в качестве аргумента. Метод `__enter__()` открывает файл для чтения и возвращает результирующий файловый объект. `__exit__()` метод принимает три обязательных аргумента, представляющих тип исключения, значение и обратную трассировку. Затем файловый объект закрывается, вызывая `.close()` метод для этого объекта.

Вот как `TextFileReader` класс работает на практике:

```
from reader import TextFileReader
with TextFileReader("hello.txt") as file:
    print(file.read())
```

Выведет: Hello, World!

Класс работает точно так же, как и при использовании `open()` для открытия файла в режиме чтения.

Декораторы функций

Часто бывает, что нужно изменить функцию, добавив в неё новый функционал. А этот новый функционал может применяться не только к данной исходной функции, но и к другим функциям без их изменения. Для этого и служат декораторы функций. Так, что такое декораторы функций? Это своего рода обёртка над исходной функцией.

Чтобы показать, что мы обёртываем какую-либо функцию, нам просто надо на предыдущей строке, после которой идёт описание данной функции, поставить символ `@` и название обёртки, по другому — декоратора.

Сама функция декоратора должна быть заранее описана.

Разберём конкретный пример. Когда мы пишем код на Python и у нас критично выполнение этого кода по времени, нам нужно понять какая функция забирает основную часть времени на своё выполнение. И потом постараться время выполнения этой функции уменьшить.

Но, сначала нам нужно протестировать каждую функцию кода на время её выполнения. Если мы будем изменять код каждой функции, добавляя инструменты, позволяющие определить время их выполнения, то такая практика вредна и противоречит принципу «не повторяйся!».

Если же мы напишем общий код для определения времени выполнения любой функции, и обернём ею исследуемую функцию, то мы не будем повторяться и не будем изменять в код в исследуемых функциях.

Код декоратора:

```
import time
def my_decor(func):
    def my_wr():
        start_time = time.time()
        func()
        print(time.time() - start_time)
    return my_wr
```

Здесь, внутри функции-декоратора `my_decor(func)` аргументом является функция `func`.

Во внутренней функции `def my_wr()` функции -декоратора `my_decor(func)` сначала определяется текущее на данный момент время (то есть до вызова и отработки функции `func`). Далее вызывается сама функция `func` и после её выполнения распечатывается разность между текущим временем на момент уже

отработанной функции func и первоначальным временем start_time, до отработки функции func.

Затем, функция -декоратор my_decor(func) возвращает с помощью оператора return внутреннюю функцию my_wr с выполнением переданной функции и определения времени её выполнения.

```
@my_decor
```

```
def sp():
```

```
    list = [i ** 2 for i in range(100000) if i % 2 == 0]
```

```
    print(list)
```

Мы записали после вызовы функции -декоратора my_decor нашу исследуемую функцию def sp(). В ней записано создание списка, в котором переменная i в цикле от 0 до 100000, возводится в степень 2, и по условию if i % 2 == 0, то есть только чётные числа.

Запускаем функцию sp:

```
sp()
```

Получим список квадратов чётных чисел в диапазоне от 0 до 100000 и время выполнения исследуемой функции. То есть последнее выводимое число, показывающее время выполнения исследуемой функции, что-то вроде 0.030806541442871094, то есть наша функция исполнялась приблизительно 30 миллисекунд. На каждом компьютере результат будет иной, в зависимости от быстродействия.

Такой декоратор определения времени выполнения кода можно применять для бесконечного числа функций.

Использование классов в качестве декораторов

В Python использование классов приветствуется, так как в Python все элементы языка являются объектами. Вот и декораторы могут быть классами, в инициализаторе которых будет не атрибут, а функция.

__init__() метод производит инициализацию.

Чтобы можно было вызывать экземпляр класса, применяется

специальный __call__() метод:

```
class Counter:
```

```
    def __init__(self, start=0):
```

```
        self.count = start
```

```
    def __call__(self):
```

```
        self.count += 1
```

```
        print(f"Текущий счетчик равен {self.count}")
```

```
counter = Counter() # Инициализируем экземпляр класса с помощью __init__()
```

```
for i in range(0,10):
```

```
    counter()
```

```
# Выведет:
```

```
Текущий счетчик равен 1
```

Текущий счетчик равен 2
Текущий счетчик равен 3
Текущий счетчик равен 4
Текущий счетчик равен 5
Текущий счетчик равен 6
Текущий счетчик равен 7
Текущий счетчик равен 8
Текущий счетчик равен 9
Текущий счетчик равен 10

В этом цикле каждый раз, когда вызывается counter(), состояние счётчика увеличивается на 1.

Типичная реализация класса decorator должна реализовываться через `__init__()` и `__call__()`:

decorators.py

```
import functools
```

```
class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0
    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Вызов {self.num_calls} из {self.func.__name__}")
        return self.func(*args, **kwargs)
```

Здесь импортируется `functools`. Эта библиотека предназначена для работы с функциями высшего порядка. Такие функции могут принимать другие функции, а также возвращать переработанные функции.

Понятно, что с помощью `functools` можно писать переиспользуемый код, а это избавит от избыточного кода и увеличит возможности кода в Python.

Метод `__init__()` должен хранить ссылку на функцию и может выполнять любую другую необходимую инициализацию. Вместо оформленной функции будет вызван метод `__call__()`.

Декоратор `@CountCalls` работает точно так же:

```
from decorators import CountCalls
```

```
@CountCalls
def say_whee():
    print("Ух ты!")
```

```
say_whee()
# Выведет:
Вызов 1 из say_whee()
```

Ух ты!

```
say_whee()  
# Выведет:  
Вызов 2 из say_whee()  
Ух ты!
```

```
say_whee.num_calls  
# Выведет:  
2  
Каждый вызов say_whee() подсчитывается и запоминается.
```

Ещё пример. Допустим декоратор @slow_down затормаживает исполнение функции на одну секунду. Увеличить задержку исполнения функции можно, используя необязательный rate аргумент:

decorators_new.py

```
import functools  
import time
```

```
# ...
```

```
def slow_down(_func=None, *, rate=1):  
    """Перевод функции в режим ожидания на заданное количество секунд  
    перед её вызовом"""  
    def decorator_slow_down(func):  
        @functools.wraps(func)  
        def wrapper_slow_down(*args, **kwargs):  
            time.sleep(rate)  
            return func(*args, **kwargs)  
        return wrapper_slow_down  
    if _func is None:  
        return decorator_slow_down  
    else:  
        return decorator_slow_down(_func)
```

Пример рекурсивной функции countdown(), которая теперь будет ждать две секунды между каждым отсчетом:

```
from decorators_new import slow_down
```

```
@slow_down(rate=2)  
def countdown(from_number):  
    if from_number < 1:  
        print("От винта!")
```

```
else:
    print(from_number)
    countdown(from_number - 1)
```

Чтобы увидеть эффект от декоратора напишем:

```
countdown(3)
```

Выведет:

3

2

1

От винта!

Между каждой цифрой обратного отсчета будет двухсекундная пауза.

Создание синглтонов

Синглтон - это класс, имеющий только один экземпляр. В Python есть несколько часто используемых синглтонов, в том числе None, True и False. Тот факт, что None является синглтоном, позволяет нам сравнивать значение None с помощью ключевого слова is, как при создании декораторов с необязательными аргументами:

```
if _func is None:
    return decorator_name
else:
    return decorator_name(_func)
```

Использование is возвращает значение True только для объектов, которые являются одним и тем же экземпляром. Следующий декоратор @singleton превращает класс в синглтон, сохраняя первый экземпляр класса в качестве атрибута. Последующие попытки создания экземпляра просто возвращают сохраненный экземпляр:

[decorators_singleton.py](#)

```
import functools
```

```
# ...
```

```
def singleton(cls):
    """Делает класс одноэлементным классом singleton (только один
    экземпляр)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if wrapper_singleton.instance is None:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
```

```
return wrapper_singleton
```

Этот class decorator следует тому же шаблону, что и function decorators. Единственное отличие заключается в том, что используется cls вместо func в качестве имени параметра, чтобы указать, что это должен быть декоратор класса.

Проверяем:

```
from decorators import decorators_singleton
```

```
@singleton
class TheOne:
    pass
```

```
first_one = TheOne()
another_one = TheOne()
```

```
id(first_one)
# Выведет: 140481222233584
```

```
id(another_one)
# Выведет: 140481222233584
```

```
first_one is another_one
True
```

Проверка идентификаторов объектов с помощью ключевого слова is, мы видим, что first_one действительно является тем же экземпляром, что и another_one. Эффект одноэлементности лучше реализовать в виде глобальной переменной внутри модуля. Декораторы классов используются реже, чем декораторы функций.

Декораторы могут предоставить удобный механизм для кэширования и запоминания.

В качестве примера можно рассмотреть обычное решение последовательности Фибоначчи. Чтобы вычислить десятое число Фибоначчи таким образом, нужно будет вычислить только предыдущие числа Фибоначчи, но эта реализация требует 109 вычислений:

```
from decorators import CountCalls
```

```
@CountCalls
def fibonacci(n):
    if n in (1, 2):
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Хотя эта реализация проста, но ее производительность желает лучшего:

fibonacci(10)

Выведет:

Вызов 1 из fibonacci()

Вызов 2 из fibonacci()

Вызов 3 из fibonacci()

Вызов 4 из fibonacci()

Вызов 5 из fibonacci()

Вызов 6 из fibonacci()

Вызов 7 из fibonacci()

Вызов 8 из fibonacci()

Вызов 9 из fibonacci()

Вызов 10 из fibonacci()

Вызов 11 из fibonacci()

Вызов 12 из fibonacci()

Вызов 13 из fibonacci()

Вызов 14 из fibonacci()

Вызов 15 из fibonacci()

Вызов 16 из fibonacci()

Вызов 17 из fibonacci()

Вызов 18 из fibonacci()

Вызов 19 из fibonacci()

Вызов 20 из fibonacci()

Вызов 21 из fibonacci()

Вызов 22 из fibonacci()

Вызов 23 из fibonacci()

Вызов 24 из fibonacci()

Вызов 25 из fibonacci()

Вызов 26 из fibonacci()

Вызов 27 из fibonacci()

Вызов 28 из fibonacci()

Вызов 29 из fibonacci()

Вызов 30 из fibonacci()

Вызов 31 из fibonacci()

Вызов 32 из fibonacci()

Вызов 33 из fibonacci()

Вызов 34 из fibonacci()

Вызов 35 из fibonacci()

Вызов 36 из fibonacci()

Вызов 37 из fibonacci()

Вызов 38 из fibonacci()

Вызов 39 из fibonacci()

Вызов 40 из fibonacci()

Вызов 41 из fibonacci()

Вызов 42 из fibonacci()
Вызов 43 из fibonacci()
Вызов 44 из fibonacci()
Вызов 45 из fibonacci()
Вызов 46 из fibonacci()
Вызов 47 из fibonacci()
Вызов 48 из fibonacci()
Вызов 49 из fibonacci()
Вызов 50 из fibonacci()
Вызов 51 из fibonacci()
Вызов 52 из fibonacci()
Вызов 53 из fibonacci()
Вызов 54 из fibonacci()
Вызов 55 из fibonacci()
Вызов 56 из fibonacci()
Вызов 57 из fibonacci()
Вызов 58 из fibonacci()
Вызов 59 из fibonacci()
Вызов 60 из fibonacci()
Вызов 61 из fibonacci()
Вызов 62 из fibonacci()
Вызов 63 из fibonacci()
Вызов 64 из fibonacci()
Вызов 65 из fibonacci()
Вызов 66 из fibonacci()
Вызов 67 из fibonacci()
Вызов 68 из fibonacci()
Вызов 69 из fibonacci()
Вызов 70 из fibonacci()
Вызов 71 из fibonacci()
Вызов 72 из fibonacci()
Вызов 73 из fibonacci()
Вызов 74 из fibonacci()
Вызов 75 из fibonacci()
Вызов 76 из fibonacci()
Вызов 77 из fibonacci()
Вызов 78 из fibonacci()
Вызов 79 из fibonacci()
Вызов 80 из fibonacci()
Вызов 81 из fibonacci()
Вызов 82 из fibonacci()
Вызов 83 из fibonacci()
Вызов 84 из fibonacci()
Вызов 85 из fibonacci()

Вызов 86 из fibonacci()
Вызов 87 из fibonacci()
Вызов 88 из fibonacci()
Вызов 89 из fibonacci()
Вызов 90 из fibonacci()
Вызов 91 из fibonacci()
Вызов 92 из fibonacci()
Вызов 93 из fibonacci()
Вызов 94 из fibonacci()
Вызов 95 из fibonacci()
Вызов 96 из fibonacci()
Вызов 97 из fibonacci()
Вызов 98 из fibonacci()
Вызов 99 из fibonacci()
Вызов 100 из fibonacci()
Вызов 101 из fibonacci()
Вызов 102 из fibonacci()
Вызов 103 из fibonacci()
Вызов 104 из fibonacci()
Вызов 105 из fibonacci()
Вызов 106 из fibonacci()
Вызов 107 из fibonacci()
Вызов 108 из fibonacci()
Вызов 109 из fibonacci()

55

Ситуация быстро ухудшается при вычислении двадцатого числа Фибоначчи, которое будет равно 6765 и потребует 13638 вычислений, а для тридцатого числа с результатом 832040 потребует 1677717 вычислений и расчёт может занять более минуты. Это потому, что код продолжает вычислять числа Фибоначчи, которые уже известны. Однако может помочь кэширование вычислений. Для этого добавляется в свой модуль декоратор @cache:

`decorators_cache.py`

`import functools`

`# ...`

`def cache(func):`

`"""Сохранение кэш предыдущих вызовов функций"""`

`@functools.wraps(func)`

`def wrapper_cache(*args, **kwargs):`

`cache_key = args + tuple(kwargs.items())`

`if cache_key not in wrapper_cache.cache:`

`wrapper_cache.cache[cache_key] = func(*args, **kwargs)`

```
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = {}
    return wrapper_cache
```

Кэш работает как таблица подстановки, поскольку хранит вычисления в словаре. Можно добавить его в функцию fibonacci():

```
from decorators import CountCalls
from decorators_cache import cache
```

```
@cache
@CountCalls
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

В который раз мы используем декоратор класса @CountCalls для контроля производительности наших вычислений. А с декоратором @cache в функции fibonacci() необходимые вычисления выполняются только один раз:

```
fibonacci(10)
# Выведет:
Вызов 1 из fibonacci()
Вызов 2 из fibonacci()
Вызов 3 из fibonacci()
Вызов 4 из fibonacci()
Вызов 5 из fibonacci()
Вызов 6 из fibonacci()
Вызов 7 из fibonacci()
Вызов 8 из fibonacci()
Вызов 9 из fibonacci()
Вызов 10 из fibonacci()
Вызов 11 из fibonacci()
55
```

```
fibonacci(8)
# Выведет: 21
```

При вызове функции fibonacci(8) никаких новых вычислений не потребовалось, поскольку восьмое число Фибоначчи уже было вычислено для функции fibonacci(10).

В стандартной библиотеке кэш в последнее время используется @functools.lru_cache. Кроме того, можно использовать обычный кэш с помощью @functools.cache.

У этих декораторов больше возможностей, чем у того, который мы использовали. Лучше воспользоваться декораторами `@functools.lru_cache` или `@functools.cache` вместо того, чтобы писать свой собственный декоратор кэша.

В следующем примере мы не возвращаем результат сразу. Вместо этого мы добавляем вызов функции `print()`, чтобы видеть, когда результат вычисляется, а не просто извлекается из кэша:

```
import functools

@functools.lru_cache(maxsize=4)
def fibonacci(num):
    if num < 2:
        value = num
    else:
        value = fibonacci(num - 1) + fibonacci(num - 2)
    print(f"Вычисленный по методу Фибоначчи({num}) = {value}")
    return value
```

Параметр `maxsize` указывает, сколько последних вызовов кэшируется. Значение по умолчанию равно 128, но вы можете указать значение `maxsize=None` для кэширования всех вызовов функций. Использование `@functools.cache` дает тот же эффект, что и значение `maxsize=None`. Однако это может вызвать проблемы с памятью, если мы кэшируем много больших объектов.

Мы можем использовать метод `.cache_info()`, чтобы увидеть, как работает кэш, и при необходимости настроить его. В нашем примере мы использовали специально небольшую величину максимального размера, чтобы увидеть эффект удаления элементов из кэша:

```
fibonacci(10)
Вычисленный по методу Фибоначчи(1) = 1
Вычисленный по методу Фибоначчи(0) = 0
Вычисленный по методу Фибоначчи(2) = 1
Вычисленный по методу Фибоначчи(3) = 2
Вычисленный по методу Фибоначчи(4) = 3
Вычисленный по методу Фибоначчи(5) = 5
Вычисленный по методу Фибоначчи(6) = 8
Вычисленный по методу Фибоначчи(7) = 13
Вычисленный по методу Фибоначчи(8) = 21
Вычисленный по методу Фибоначчи(9) = 34
Вычисленный по методу Фибоначчи(10) = 55
55
```

```
fibonacci(8)
21
```

```
fibonacci(5)
Вычисленный по методу Фибоначчи(1) = 1
Вычисленный по методу Фибоначчи(0) = 0
Вычисленный по методу Фибоначчи(2) = 1
Вычисленный по методу Фибоначчи(3) = 2
Вычисленный по методу Фибоначчи(4) = 3
Вычисленный по методу Фибоначчи(5) = 5
5
```

```
fibonacci(8)
Вычисленный по методу Фибоначчи(6) = 8
Вычисленный по методу Фибоначчи(7) = 13
Вычисленный по методу Фибоначчи(8) = 21
21
```

```
fibonacci(5)
5
```

```
fibonacci.cache_info()
CacheInfo(hits=17, misses=20, maxsize=4, currsize=4)
```

В этих примерах мы вычисляем несколько чисел Фибоначчи. Наш кэш одновременно выполняет только четыре вычисления. Например, после вычисления `fibonacci(10)` в нем остаются седьмое, восьмое, девятое и десятое числа.

Таким образом, мы можем найти значение `fibonacci(8)` без каких-либо пересчётов. Затем мы запрашиваем значение `фибоначчи(5)`, но это пятое число было удалено из кэша. Поэтому его нужно вычислять с нуля.

В большинстве приложений нам не нужно ограничивать свой кэш, и мы можем напрямую использовать `@functools.cache`.

Добавление информации о единицах измерения

Следующий пример в некоторой степени похож на предыдущий, но на самом деле он не изменяет поведение оформленной функции. Он просто добавляет единицу измерения в расчётах как атрибут функции:

```
decorator_set_unit.py
# ...
```

```
def set_unit(unit):
```

```

"""Зарегистрируйте единицу измерения в функции"""
def decorator_set_unit(func):
    func.unit = unit
    return func
return decorator_set_unit

```

Например, вычисляем объем цилиндра на основе его радиуса и высоты в сантиметрах:

```

import math
from decorator_set_unit import set_unit

@set_unit("cm^3")
def volume(radius, height):
    return math.pi * radius**2 * height

```

Из математической библиотеки `math` берётся число `Пи`. С помощью декоратора `@set_unit("cm^3")` мы указываем функции `volume()`, что результат будет рассчитываться в кубических сантиметрах. Мы также можем, если необходимо, определить в каких единицах измерение был произведён расчёт. Для этого можно обратиться к атрибуту функции `.unit`:

```

volume(3, 5)
#Выведет: 141.3716694115407

```

```

volume.unit
#Выведет: 'cm^3'

```

Можно установить ещё один интересный библиотечный модуль, который может конвертировать расчётные показатели в других единицах измерения. Такой библиотекой является `pint`. Установка `pint` осуществляется командой терминала:

```
python -m pip install Pint
```

Используя `pint` можно, например, преобразовать объем в кубические дюймы или галлоны:

```

import pint
ureg = pint.UnitRegistry()
vol = volume(3, 5) * ureg(volume.unit)

vol
#Выведет:
<Quantity(141.3716694115407, 'centimeter ** 3')>

vol.to("cubic inches")
#Выведет:

```

```
<Quantity(8.627028576414954, 'inch ** 3')>
```

```
vol.to("gallons").m # Magnitude
```

```
#Выведет:
```

```
0.0373464440537444
```

Мы используем `pint` для создания переменной, которая имеет как величину, так и единицу измерения. Вызывая `.to()`, мы преобразуем переменные в другие единицы измерения. Например, объем цилиндра составляет около 141 кубического сантиметра, что в переводе составляет примерно 8,63 кубических дюйма и 0,0373 галлона.

Мы также можем изменить декоратор так, чтобы он возвращал `pint Quantity` напрямую. Такой `Quantity` создается путем умножения значения на единицу измерения.

```
decorator_use_unit_pint.py
```

```
import functools
```

```
import pint
```

```
# ...
```

```
def use_unit(unit):
```

```
    """Пусть функция возвращает величину с заданной единицей измерения"""
```

```
    use_unit.ureg = pint.UnitRegistry()
```

```
    def decorator_use_unit(func):
```

```
        @functools.wraps(func)
```

```
        def wrapper_use_unit(*args, **kwargs):
```

```
            value = func(*args, **kwargs)
```

```
            return value * use_unit.ureg(unit)
```

```
        return wrapper_use_unit
```

```
    return decorator_use_unit
```

С `@use_unit` decorator преобразовать единицы измерения достаточно легко:

```
from decorator_use_unit import use_unit
```

```
@use_unit("meters per second")
```

```
def average_speed(distance, duration):
```

```
    return distance / duration
```

```
bolt = average_speed(100, 9.58)
```

```
print(bolt)
```

```
# Выведет: 10.438413361169102 meter / second
```

```
print(bolt.to("km per hour"))
```

Выведет: 37.578288100208766 kilometer / hour

bolt.to("mph").m # Распечатывается только величина без единицы измерения

Выведет: 23.350065679064745

Когда Усэйн Болт пробежал 100 метров за 9,58 секунды на чемпионате мира 2009 года, его средняя скорость составляла 10,4 метра в секунду. Это примерно 37,6 километра в час или 23,4 мили в час.