Задание #1 (лаб.3). Создайте файл **Lab 3_1.py**. В файле задайте класс Country, у которого должны быть следующие атрибуты:

1. Атрибут city со значением ‘Novosibirsk’;
2. Атрибут street со значением ‘Russian_street’;
3. Атрибут building со значением 55;

К классу Country создайте два ЭК и сохраните их в переменные msc, spb. Измените:

1. В ЭК msc:
   - значение city на ‘Moscow’;
   - значение street на ‘Nikolskaya’;
   - значение building на 5.
2. В ЭК spb:
   - значение city на ‘St_Petersburg’;
   - значение street на ‘Horse_lane’;
   - значение building на 12.

Программа должна вывести на экран (разделенные запятой):

1. Атрибуты класса Country;
2. Атрибуты ЭК msc;
3. Атрибуты ЭК spb.

Вывод программы (пример):

Novosibirsk, Russian_street, 35
Moscow, Nikolskaya, 1
St_Petersburg, Horse_lane, 12

Задание #2 (лаб.3). Скачайте файл **Lab 3_2.py**. Доработайте программу по следующим критериям:

1. Добавьте в класс Students (при помощи функции setattr) атрибут student_ticket со значением 1298;
2. Создайте экземпляр student_one класса Students;
3. Создайте экземпляр student_two класса Students;
4. Измените атрибут name ЭК student_two на “Michail”;
5. Измените атрибут student_ticket ЭК student_two на 1301
6. Программа должна вывести на экран:
   a. Атрибуты класса Students;
   b. Атрибуты ЭК student_one;
   c. Атрибуты ЭК student_two;
   d. Фразу из функции questionnaire класса Students;
   e. Фразу по атрибутам ЭК student_one: “Студент: (ссылка на имя). Группа: (ссылка на группу). Курс: (ссылка на курс). Студенческий билет: (ссылка на № билета)”;
   f. Фразу по атрибутам ЭК student_two: “Студент: (ссылка на имя). Группа: (ссылка на группу). Курс: (ссылка на курс). Студенческий билет: (ссылка на № билета)”.

Задание #3 (лаб.3). Создайте файл **Lab 3_3.py**.В файле создайте класс с именем User. Создайте два атрибута класса first_name и last_name, а затем еще несколько атрибутов(минимум 2), которые обычно хранятся в профиле пользователя. Напишите функцию greet_user() для вывода приветствия. Создайте четыре ЭК User. Измените атрибуты каждого из ЭК. С использованием приветствия (функция greet_user) вызовите каждый ЭК по атрибутам в одну отдельную строку(для каждого ЭК).

Пример:

Добро пожаловать: Jackie Lewis
Добро пожаловать: Freddy Smith

Задание #4 (лаб.3). Пройдите тест-форму для лабораторной работы №3.

Введение в классы

Ключевым понятием ООП является понятие «**Объект**». И ЯП Python этим понятием активно пользуется на всех уровнях, потому что любое значение, с которым вы работаете когда программируете на Python, является объектом.

С точки зрения ООП **объект** - это контейнер состоящий из:

1. Данных (обозначающих текущее состояния объекта);
2. Поведения.

```
my_list = [1, 2, 3]
print(my_list)
my_list.append(5)
print(my_list)
```

```
[1, 2, 3]

[1, 2, 3, 5]
```

```
my_list.reverse()
print(my_list)
```

```
[5, 3, 2, 1]
```

В этом примере мы создали объект списка с *данными*, в нем хранятся значения [1, 2, 3]. У нашего объекта списка есть *поведение*, например добавление в конец наших данных нового значения. Это поведение реализуется через метод **.append**. После вызова метода у нас меняются текущее состояние нашего объекта, а именно поменялись данные, которые в нем содержатся. Также у списков есть поведение **.reverse**, разворачивающее данные в другой последовать. После вызова метода также меняются данные объекта списка.

Класс

Следующим ключевым понятие ООП является понятие «**Класс**»

Класс объекта - это коллекция характеристик объединяющая атрибуты имеющиеся у всех экземпляров (объектов) подобного класса. Похожая классификация существует в биологии, где вся структура живого мира объединяется в классы.

К примеру, в класс "Птицы" объединены колибри, синицы, попугаи, совы и другие птицы, которые уже разбиваются на подклассы. Безусловно, объекты могут иметь разное поведение. Здесь уместно снова провести условное сравнение с животным миром:

- класс птицы умеют летать, но не умеют бегать
- класс кошка может бегать, но при этом не может летать.

И *для каждого типа данных свойственно своё поведение*, например для типа данных строка имеются методы определяющие поведение строчного типа данных. Мы не можем использовать поведение из одних типов данных с поведением других типов данных.

Связь объекта с классом

Как мы уже сказали, каждое значение в Python является объектом. Каждый объект принадлежит к определенному классу. Определить к какому классу относится конкретный объект, позволяет функция **type()**, например:

```
hello = 'Hello World!'
print(type(hello))

my_list = [2, 4, 43]
print(type(my_list))
print(type(True))
print(type(3))
print(type(6/3))
```

```
<class 'str'>

<class 'list'>
<class 'bool'>
<class 'int'>
<class 'float'>
```

Проверить принадлежность объекта к определенному классу поможет функция **isinstance()**

```
hello = 'Hello World!'
print(hello, 'Это строка?', isinstance(hello, str))
print(hello, 'Это bool?', isinstance(hello, bool))

my_list = [2, 4, 43]
print(my_list, 'Это список?', isinstance(my_list, list))
print(25, 'Это список?', isinstance(25, list))
print(25, 'Это целое число?', isinstance(25, int))
print(25, 'Это объект?', isinstance(25, object))
print(hello, 'Это объект?', isinstance(hello, object))
print(my_list, 'Это объект?', isinstance(my_list, object))
```

```
('Hello World!', 'Это строка?', True)

('Hello World!', 'Это bool?', False)

([2, 4, 43], 'Это список?', True)

(25, 'Это список?', False)

(25, 'Это целое число?', True)

(25, 'Это объект?', True)

('Hello World!', 'Это объект?', True)

([2, 4, 43], 'Это объект?', True)
```

В последних строчках мы видим, что любое значение принадлежит классу **object**, это обозначает, *что все является объектом*.

Создание своего класса

Объекты принадлежат классам и создаются на основании классов. Для создания своего класса используйте ключевое слово **class** и через пробел нужно указать **имя класса**

```
class Car:
    pass
```

По стандарту pep8 имя класса обязательно пишется с большой буквы, если имя содержит несколько слов необходимо оформлять по типу написания CamelCase.

К классу можно добавить строку документации для понимания поведения класса:

```
class Car:
    "Класс для определения характеристик машин"
    pass

print(Car.__doc__)
```

Итак, определив базовый класс мы можем создавать на его основе **экземпляры класса**. Экземпляр класса (далее ЭК) - это объект созданный на основании класса. Для того чтобы создать ЭК необходимо вызвать класс. Результатом вызова класса является ЭК и его можно сохранить в переменную. Вот пример ниже:

```
class Car:
    pass

a = Car()
b = Car()

print(type(a))
print(type(b))

print('Этот объект принадлежит классу Car?',
isinstance(a, Car))
print('Этот объект принадлежит классу Car?',
isinstance(b, Car))
print('Этот объект принадлежит классу Car?',
isinstance('hello', Car))
```

```
<class '__main__.Car'>
<class '__main__.Car'>

('Этот объект принадлежит классу Car?', True)

('Этот объект принадлежит классу Car?', True)

('Этот объект принадлежит классу Car?', False)
```

В переменных a и b лежат экземпляры класса Car.

Мы можем также проверить, что переменные имеют разные идентификаторы и занимают разные участки памяти, что говорит о том, что это разные объекты:

```
class Car:
    pass

a = Car()
b = Car()

print(id(a), a)
print(id(b), b)
```

```
(2, <__main__.Car object>)
(3, <__main__.Car object>)
```

Пока наш класс Car не содержит данных (атрибутов класса) и не имеет поведения. Добавим к классу Car атрибуты model и engine, которые будут присваиваться к объектам (ЭК) при создании:

```
class Car:
    model = "BMW"
    engine = 1.6

a = Car()
b = Car()

print(a.model)
print(b.model)
```

```
BMW
BMW
```

Теперь при создании у каждого ЭК есть свои данные, к которым можно обращаться.

Нужно понимать, что на данном этапе атрибуты model и engine являются общими и одинаковыми для всех ЭК класса Car. Мы можем изменить значение атрибута в ЭК применив следующую конструкцию:

```
b.model = "VAZ"  # Изменяем значение атрибута model в ЭК
b.color = 'white'  # Добавляем новый атрибут в ЭК
print(b.model, b.color)  # Вывод: VAZ white
```

Обращение к атрибутам класса

Обращение к атрибутам класса производится по следующей схеме:

class_name.attribute_name

где class_name - имя класса, а attribute_name - имя атрибута

К примеру создадим класс Person с атрибутами name и age и посмотрим как можно обратится к этим атрибутам класса

class Person:
  name = 'Jared'
  age = 30
 	
print(Person.name)
print(Person.age)
	Jared
	30

Также доступ к атрибутам класса Person можно получить, используя функцию getattr:

getattr(...)

   getattr(object, name[, default]) -> value

   Get a named attribute from an object; getattr(x, 'y') is equivalent to x.y.
   When a default argument is given, it is returned when the attribute doesn't
   exist; without it, an exception is raised in that case.

которая принимает:

- object - это объект, в котором будет осуществлен поиск атрибута
- name - название атрибута поиска
- необязательный аргумент default - значение, получаемое в случае отсутствия искомого атрибута. Если искомого атрибута нет в классе и параметр default не задан, происходит обработка исключения AttributeError

class Person:
  name = 'Jared'
  age = 30
 	
print(getattr(Person, 'money', 'Нет такого атрибута'))
print(getattr(Person, 'money', 100))
	Нет такого атрибута
	
	100

Для получения всех атрибутов содержащихся в классе или в экземпляре класса применяется магический атрибут __dict__.

class Person:
  """Класс Person"""
  name = 'Bob'
  age = 35
 	
print(Person.__dict__)
	{'__module__': '__main__', '__doc__': 'Класс Person', 'name': 'Bob', 'age': 35, '__dict__': <attribute '__dict__' of 'Person' objects>, '__weakref__': <attribute '__weakref__' of 'Person' objects>}

Вы увидите словарь, в ключах которого будут магические атрибуты и наши два созданных атрибута name и age

Магические атрибуты и методы - это некоторые встроенные атрибуты и методы Python, которые имеют особое значение. Добавьте два символа подчеркивания до и после названия, и оно будет вызываться автоматически при выполнении системных операций. Более подробно о магических атрибутах и методах поговорим в дальнейшем

Если имеется необходимость проверить наличие конкретного атрибута в классе, то можете воспользоваться функцией **hasattr**:

hasattr(obj, name, /)

   Return whether the object has an attribute with the given name.

This is done by calling getattr(obj, name) and catching AttributeError.
где:

- object - это объект, который будет подвержен проверке;
- attribute_name - это название искомого атрибута в виде строки.

```
class Person:
 name = 'Jared'
 age = 30

print(hasattr(Person, 'name'))
print(hasattr(Person, 'money'))
```

```
True
False
```

Изменение атрибута класса

Для изменения значения существующего атрибута класса следует применять следующую конструкцию:

class_name.attribute_name = value
где class_name - имя класса, attribute_name - имя атрибута, а value - новое значение атрибута

```
class Person:
 name = 'Jared'
 age = 30

print(Person.name)
Person.name = 'Michail'
print(Person.name)
```

```
Jared

Michail
```

Создание атрибута класса

При помощи присвоение значения атрибуту, как мы делали в примере выше, можно создавать новые атрибуты. Если атрибут класса с указываемым именем отсутствует в классе, то автоматически создаётся новый атрибут с указанным именем:

```
class Person:
   name = 'Ivan'
   age = 30

Person.money = 100
Person.phone = '+1 202 777 xxx'
print(Person.__dict__)
print(Person.money)
print(Person.phone)
```

```
{'__module__': '__main__', 'name': 'Ivan', 'age': 30,
'__dict__': <attribute '__dict__' of 'Person' objects>,
'__weakref__': <attribute '__weakref__' of 'Person'
objects>, '__doc__': None, 'money': 100, 'phone': '+1 202
777 xxx'}
100
+1 202 777 xxx
```

Также для создания нового атрибута или изменения существующего применяется функция **setattr**:

setattr(obj, name, value, /)
Sets the named attribute on the given object to the specified value.

setattr(x, 'y', v) is equivalent to x.y = v

```
class Person:
   name = 'Ivan'
   age = 30

setattr(Person, 'money', 200)
setattr(Person, 'phone', '+1 202 777 xxx')
print(Person.__dict__)
print()
setattr(Person, 'name', 'Vasya')
setattr(Person, 'age', '43')
print(Person.__dict__)
```

```
{'__module__': '__main__', 'name': 'Ivan', 'age': 30,
'__dict__': <attribute '__dict__' of 'Person' objects>,
'__weakref__': <attribute '__weakref__' of 'Person'
objects>, '__doc__': None, 'money': 200, 'phone': '+1 202
777 xxx'}

{'__module__': '__main__', 'name': 'Vasya', 'age': '43',
'__dict__': <attribute '__dict__' of 'Person' objects>,
'__weakref__': <attribute '__weakref__' of 'Person'
objects>, '__doc__': None, 'money': 200, 'phone': '+1 202
777 xxx'}
```

В результате мы получили новые атрибуты класса или изменили имеющиеся.

Удаление атрибутов класса

Для удаления атрибута следует применять оператор del или функцию delattr:

```
class Person:
    name = 'Ivan'
    age = 30
    money = 200

print(Person.__dict__)
print()
delattr(Person, 'age')
print(Person.__dict__)
print()
del Person.money
print(Person.__dict__)
```

```
{'__module__': '__main__', 'name': 'Ivan', 'age': 30,
'money': 200, '__dict__': <attribute '__dict__' of 'Person'
objects>, '__weakref__': <attribute '__weakref__' of
'Person' objects>, '__doc__': None}

{'__module__': '__main__', 'name': 'Ivan', 'money': 200,
'__dict__': <attribute '__dict__' of 'Person' objects>,
'__weakref__': <attribute '__weakref__' of 'Person'
objects>, '__doc__': None}

{'__module__': '__main__', 'name': 'Ivan', '__dict__':
<attribute '__dict__' of 'Person' objects>, '__weakref__':
<attribute '__weakref__' of 'Person' objects>, '__doc__':
None}
```

Атрибуты экземпляра класса

Все операции, которые применимы в отношении базового класса имеют такое же применение и по отношению к экземплярам классов (далее ЭК). При работе с ЭК изменения синтаксиса незначительны, так нам достаточно заменить имя класса на имя переменной на которую ссылается ЭК, пример:

instance_name.attribute_name

где instance_name - экземпляра класса к которому обращаемся, а attribute_name - имя атрибута. Вот пример:

```
class Person:
    name = 'Ivan'
    age = 30


man = Person()  # Создаём ЭК и связываем его с
переменной man
print(man.age)  # Получаем текущее значение
атрибута age в ЭК man
man.money = 100  # Создаём в ЭК атрибут money со
значением 100
print(man.money)  # Получаем текущее значение
атрибута money в ЭК man
man.money = 250  # Изменяем текущее значение
атрибута money в ЭК man на 250
print(man.money)  # Получаем текущее значение
атрибута money в ЭК man
delattr(man, 'money')  # Удаляем атрибут money из ЭК
print(hasattr(man, 'money'))  # Проверяем наличие
атрибута money в ЭК man: False
```

```
30
100
250
False
```

Помните, что атрибуты класса относятся только к самому классу и при создании ЭК эти атрибуты не создаются в самом ЭК, а получают ссылку на атрибут класса. Пример:

```
class Person:
    name = 'Ivan'
    age = 30


man = Person()
print(man.__dict__)
print('-------')
print(Person.__dict__)
```

```
{}
-------
{'__module__': '__main__', 'name': 'Ivan', 'age': 30,
'__dict__': <attribute '__dict__' of 'Person' objects>,
'__weakref__': <attribute '__weakref__' of 'Person'
objects>, '__doc__': None}
```

Но в случае изменения атрибута из под ЭК, данный атрибут со своим значением отразится в словаре атрибутов, пример:

```
class Person:
    name = 'Ivan'
    age = 30


man = Person()
print(man.__dict__)# Печатает пустой словарь: {}
man.name = 'Alex'
print(man.__dict__)# Печатает словарь: {'name':
'Alex'}
```

```
{}
{'name': 'Alex'}
```

При этом стоит отметить, что изменения не затронут сам класс, так как изменение значения проводилось из под ЭК, проверим это:

```
class Person:
    name = 'Ivan'
    age = 30


man = Person()
print(man.__dict__)
man.name = 'Alex'
print('Атрибуты ЭК:', man.__dict__)
print('Атрибуты класса:', Person.__dict__)
```

```
{}
Атрибуты ЭК: {'name': 'Alex'}
Атрибуты класса: {'__module__': '__main__', 'name':
'Ivan', 'age': 30, '__dict__': <attribute '__dict__' of
'Person' objects>, '__weakref__': <attribute
'__weakref__' of 'Person' objects>, '__doc__': None}
```

Из этого примера мы видим, что атрибут с именем name стал принадлежать ЭК, а значит операции с атрибутами самого класса уже не затронут атрибуты принадлежащие экземпляру класса. Теперь проверим, как будут вести себя атрибуты двух разных ЭК:

```
class Person:
    name = 'Ivan'
    age = 30


man = Person()
print('Атрибуты ЭК man:', man.__dict__)
man.name = 'Alex'
print('Атрибуты ЭК man:', man.__dict__)
dude = Person()
print('Атрибуты ЭК people:', dude.__dict__)
dude.name = 'Sergey'
print('Атрибуты ЭК people:', dude.__dict__)
```

```
Атрибуты ЭК man: {}
Атрибуты ЭК man: {'name': 'Alex'}
Атрибуты ЭК people: {}
Атрибуты ЭК people: {'name': 'Sergey'}
```

Как видно из работы программы, атрибуты двух ЭК независимы друг от друга и все изменения касаются непосредственно тех ЭК, где они проводятся. В то же время операции с атрибутами класса отражаются по всей цепочке от класса до экземпляра класса, кроме тех атрибутов, которые уже принадлежат ЭК, что указывает нам на различие пространства имён этих объектов.

Если в классе и ЭК находятся аргументы с одинаковыми именами и разными значениями, то в случае удаления одноимённого аргумента из ЭК, то при обращении к аргументу ЭК будет выводится аргумент из самого класса:

```
class Person:
    name = 'Ivan'
    age = 30

man = Person()
man.name = 'Alex'
print(Person.name, man.name) # Печатает «Ivan Alex»
```

```
Ivan Alex
Ivan Ivan
```

```
del man.name
print(Person.name, man.name) # Печатает «Ivan Ivan»
```

АТРИБУТЫ - ФУНКЦИИ

Кроме атрибутов в виде переменных класса, в качестве атрибутов класса также применяются функции, например:

```
class Car:
    model = "BMW"
    engine = 1.6

    def drive():
        print("Let's go")


print(Car.__dict__)
```

```
{'__module__': '__main__', 'model': 'BMW', 'engine': 1.6, 'drive': <function Car.drive at 0x7f12e5370f70>, '__dict__': <attribute '__dict__' of 'Car' objects>, '__weakref__': <attribute '__weakref__' of 'Car' objects>, '__doc__': None}
```

Как показано в выводе словаря - в нём имеются атрибуты класса и представлен ключ drive, значение которого является объектом-функцией.

Для вызова данного атрибута-функции можно использовать следующую конструкцию:

class_name.attribute_function_name()
где class_name - имя класса, а attribute_function_name - имя атрибута

Или используя функцию getattr() с последующим оператором вызова:

getattr(class_name, 'attribute_name')()

Пример:

```
class Car:
  model = "BMW"
  engine = 1.6

  def drive():
    print("Let's go")

Car.drive()
getattr(Car, 'drive')()
```

```
Let's go
Let's go
```

Не забывайте про оператор вызова (круглые скобочки в самом конце).

Обратиться напрямую к атрибуту-функции(методу) класса через экземпляр класса невозможно, возникнет ошибка отсутствия обязательного аргумента.

Если же вы хотите сейчас вызвать атрибут-функцию через экземпляр класса, то можете воспользоваться декоратором @staticmethod (о нем мы тоже поговорим далее). Код:

```
class Car:
  model = "BMW"
  engine = 1.6

  @staticmethod
  def drive():
    print("Let's go")

a = Car()
b = Car()
Car.drive()
getattr(Car, 'drive')()
a.drive()
b.drive()
```

```
Let's go
Let's go
Let's go
Let's go
```