ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

Л.А. Надейкина

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие
по выполнению лабораторных работ № 5–7

для студентов
направления 09.03.01
очной формы обучения

Москва
ИД Академии Жуковского
2024

УДК 004.42
ББК 6Ф7.3
Н17

Рецензент:
Черкасова Н.И. – канд. физ.-мат. наук

Надейкина Л.А.
Н17 Современные технологии программирования [Текст] : учебно-методическое пособие по выполнению лабораторных работ № 5–7 / Л.А. Надейкина. – М.: ИД Академии Жуковского, 2024. – 48 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Современные технологии программирования» по учебному плану направления подготовки 09.03.01 для студентов очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 18.07.2024 г. и методического совета 18.07.2024 г.

УДК 004.42
ББК 6Ф7.3

В авторской редакции

Подписано в печать 25.11.2024 г.
Формат 60x84/16 Печ. л. 3 Усл. печ. л. 2,79
Заказ № 1042/0909-УМП02 Тираж 30 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (499) 755-55-43
E-mail: zakaz@itsbook.ru

1 ЛАБОРАТОРНАЯ РАБОТА № 5

Разработка функций и модулей пользователя

1.1 Цель лабораторной работы

Целью лабораторной работы является рассмотрение особенностей разработки функций и модулей пользователя.

1.2 Теоретические сведения

Создание функций пользователя

Кроме использования встроенных функций языка Python и методов его классов, можно создавать и использовать свои собственные функции.

Описание любой функции начинается со служебного слова *def* и имеет следующий вид:

```
def <имя функции> ([<параметры>]):
    <блок>.
```

Функция может не иметь формальных параметров, иметь один параметр или несколько. Если функция имеет несколько параметров, то они перечисляются через запятую.

Первой строкой блока функции может быть строка ее документирования (docstring), которая берется в тройные кавычки и появляется на экране в качестве подсказки при вызове функции во время задания аргументов.

Для возвращения значения функции используется оператор *return*. Если возвращается несколько значений, то они должны быть указаны в виде списка. Если оператор *return* не задан в блоке функции или указан без операнда, возвращается значение *None*.

Функцию можно определять в модуле, внутри другой функции или в классе. Функция, определенная в классе, называется методом.

Функции в языке Python являются объектами. С ними можно работать, как и с другими объектами языка. В этом состоит их отличие от функций в таких языках, как Java, C++ и C#.

При вызове фукнции указывается ее имя и в скобках – аргументы (если функция их имеет). Число аргументов и их типы должны соответствовать числу и типам параметров функции. При выполнении функции значения аргументов присваиваются соответствующим параметрам.

В качестве примера создания функции, рассмотрим задачу генерации чисел Фибоначчи. Решение этой задачи реализуем в виде функции *fibo(k)*, которая определяет первые *k* чисел Фибоначчи и возвращает их в виде списка:

```
>>> def fibo (k):
        '''Формирование чисел Фибоначчи (k – их количество)'''
        a, b, i, f=0,1,0, []
```

```
    while i<k:
        f+=[b]
        a, b=b, a+b
        i+=1
    return f
```

Отметим, что созданная функция является объектом:

```
>>> print (insistence (fibo, object))
True,
```

который можно вызвать по имени, потому что встроенная функция *callable()* возвращает значение True:

```
>>> print ( callable (fibo))
True,
```

и получить 10 чисел Фибоначчи:

```
>>> print (fibo(10))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Функция *fibo()* имеет один параметр *k*, с помощью которого пользователь задает число необходимых ему чисел Фибоначчи.

В строке документирования указано назначение функции и ее параметра.

Имеется несколько способов передачи значений параметрам функции:

- *с помощью позиционных аргументов;*
- *с помощью именнованных аргументов;*
- *с помощью аргументов, имеющих значения по умолчанию;*
- *с помощью аргументов, заданных списком;*
- *с помощью аргументов, заданных словарем.*

Использование позиционных аргументов

Позиционными являются такие аргументы функции, порядок которых при вызове функции строго определен порядком параметров функции, то есть, их позицией. Нарушение порядка следования аргументов (кроме отдельных случаев, когда порядок параметров неважен) приведет к нарушению правильной работы функции.

Рассмотрим пример создания функции, имеющей несколько позиционных параметров:

```
>>> def val (x, y, op):
        if op== '+':
            return x+y
        elif op== '-':
            return x-y
        elif op== '*':
            return x*y
        elif op== '/':
```

```
            return x/y
        else:
            return 'Неверная операция'
```

Параметры *x* и *y* задают операнды, а *op* – одну из арифметических операций. Функция возвращает результат выполненной операции:

```
>>> val(3, 2, '+')
5
```

Если будет указана недопустимая операция, например,

```
>>> val(3, 2, '//'),
```

функция в качестве значения возвращает сообщение:

```
'Неверная операция'
```

Использование именованных аргументов

В случае использования позиционных аргументов существует опасность нарушения порядка следования аргументов при вызове функции, например:

```
>>> val ('+', 3, 2)
'Неверная операция'
```

Чтобы этого не произошло, а также для более ясного текста программы, в языке Python разрешено использовать именованные аргументы, то есть. аргументы, в которых кроме значения указывается также имя параметра. В таком случае при вызове функции аргументы могут быть указаны в любом порядке:

```
>>> val(op= '+', x=3, y=2)
5
```

Отметим, что можно одновременно использовать именованные аргументы и позиционные. Надо только следить за тем, чтобы справа от именованного аргумента были указаны только именованные аргументы:

```
>>> val(3, op= '+', y=2)
5
```

Использование аргументов, заданных по умолчанию

В языке Python разрешено также использование аргументов с заданными по умолчанию значениями. Это полезно делать в случаях, когда значения параметров изменяются достаточно редко. Например, если функция *val()* используется в основном для выполнения операции сложения, то ее параметры можно описать следующим образом (остальная часть функции не изменилась):

```
>>> def val (x, y, op= '+'):
```

Тогда для выполнения операции сложения при вызове функции третий аргумент можно не указывать:

```
>>> val(3, 2)
5
```

Если же функцию *val()* необходимо использовать для умножения чисел, то тогда явно нужно указать третий параметр, значение которого переопределяет значение, заданное по умолчанию:

```
>>> val(3, 2, '*')
6
```

Использование аргументов, заданных списком

Python позволяет создавать функции с переменным числом значений аргументов. Для этого перед именем параметра необходимо указать символ *.

При вызове функции переданные значения для данного параметра представляются в виде списка, в виде заключенной в квадратные скобки последовательности данных, разделенных запятыми. В качестве примера приведем описание функции, выполняющей сложение произвольного числа слагаемых:

```
>>> def sum1 (*args):
        s=0
        for el in args:
                s+=el
        return s

>>> sum1(3,5,7,2)
17
```

Можно также, задав список значений:

```
>>> a_list=[7,9,3,11]
```

вызвать функцию, указав в качестве аргумента имя списка с впереди стоящей звездочкой:

```
>>> sum1(*a_list)
30
```

Использование аргументов, заданных словарем

Другим вариантом создания функции с переменным числом аргументов является использование перед именем параметра двух символов "звездочка" (**). В этом случае при вызове функции переданные значения аргумента представляются в виде словаря.

В качестве примера приведем описание функции *list_of_keys()*, имеющей два аргумента – первый (позиционный) *color* указывает цвет, второй ***args* указывает на аргумент в виде словаря.

Функция просматривает элементы словаря и сравнивает их значения с значением первого аргумента. При совпадении значений ключ элемента словаря заносится в формируемый список ключей, который возвращает функция:

```
>>> def list_of_keys (color, **args):
        l_color=[]
```

```
        for k in args:
                if args[k]==color:
                        l_color+=[k]
        return l_color
```

```
>>> a_dict={ '1' : 'blue' , '2' : 'blue' , '3' : 'red' , '4' : 'blue' , '5' : 'green' , '6' : 'green' ,
'7' : 'blue' }
>>> list_of_keys( 'blue', **a_dict)
['1', '2', '7', '4']
```

lambda функции

lambda функции широко используются в языках функционального программирования. В языке Python также осуществлена их поддержка:

```
>>> def f1 (n): return lambda x:x**n
>>> f2=f1(3)
>>> f2(5)
125
```

Можно просто вызвать функцию *f1*:

```
>>> f1(3)(5)
125
```

Использование *lambda* функций показано также ниже

Замыкание

Замыкание – это особый вид функции, которая определена в теле другой функции и создается каждый раз при выполнении внешней функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции.

Ссылки на переменные внешней функции в случае замыкания действительны внутри вложенной функции даже если внешняя функция завершила работу и переменные вышли из области видимости.

Приведем пример простого замыкания:

```
>>> def make_add (x):
        def add (y):
                return x+y
        return add
>>> make_add(2)(3)
5
```

Отметим, что в языке Python изменять значения можно только тех переменных замыкания, которые относятся к изменяемым типам. Поэтому при выполнении примера, в котором подчитывается число вызовов функции замыкания, заданного целочисленным типом:

```
>>> def create_counter (count_calls=0):
        def change_counter ():
            count_calls+=1
            return count_calls
        return change_counter
```

возникает ошибка:

nboundLocalError: local variable 'count_calls' referenced before assignment.

Исправить положение можно, задав число вызовов в виде элемента списка:

```
>>> def create_counter (count_calls=[0]):
        def change_counter ():
                count_calls[0]+=1
                return count_calls[0]
        return change_counter
>>> create_counter()()
1
>>> create_counter()()
2
>>> create_counter()()
3
```

Использование функции при сортировке

Списки поддерживают метод *sort()*, который осуществляет сортировку элементов списка. При использовании этого метода элементами списка могут быть только данные, которые можно сравнивать между собой – числа или строки, причем список не должен содержать сразу и числа, и строки.

Необходимо отметить, что метод *sort()* выполнив сортировку элементов списка, не возвращает результаты своей работы, точнее, всегда возвращает значение *None*. Метод не имеет обязательных параметров – если при его вызове аргументы не заданы, то элементы списка будут отсортированы по возрастанию их значений (строки сортируются по значению кодов их символов). Например:

```
>>> a_list=[5,2.0,44,-3,27.5]
>>> a_list.sort()
>>> a_list
[-3, 2.0, 5, 27.5, 44],
>>> b_list=[ 'Python', 'cat' , 'zz' , 'A' ]
>>> b_list.sort()
>>> b_list
['A', 'Python', 'cat', 'zz'].
```

Два необязательных параметра предназначены для изменения порядка сортировки:

- *reverse – когда получает значение 1, изменяет направление сортировки: элементы списка сортируются от большего значения к меньшему (по умолчанию reverse = 0):*

```
>>> a_list.sort(reverse=1)
>>> a_list
[44, 27.5, 5, 2.0, -3];
```

- *key – указывает имя функции, которая в качестве аргумента принимает значения элементов списка и возвращает значения, используемые при сортировке списка вместо значений его элементов.*

Для использования параметра *key* необходимо разработать функцию, которая преобразовывала бы элементы списка для сравнения. Например, для сортировкм слов не по алфавиту, а по их длине, необходимо, чтобы функция возвращала их длину:

```
>>> def sort1 (item):
        return len (item)
```

Теперь выполним сортировку слов списка, расположив их в порядке уменьшения их длин:

```
>>> b_list.sort(key=sort1, reverse=1)
>>> b_list
['Python', 'cat', 'zz', 'A'],
```

Использование параметра *key* позволяет осуществить сортировку списков, имеющих *элементы разных типов*.

Пусть, например, список *c_list* содержит числа, списки чисел и строки:

```
>>> c_list=[6, 'Питон', [3,7,6], 'web', [1.6,20], 0.67e1]
```

Использование метода *sort()* без аргументов

```
>>> c_list.sort()
```

вызовет исключение:

```
TypeError: unorderable types: int() < list(),
```

Для устранения этого ограничения разработаем функцию *sort2*, которая анализирует тип элемента списка и возвращает следующие значения:

- *для списка – среднее арифметическое элементов списка;*
- *для строки – ASCII код первого символа строки;*
- *для числа – само число:*

```
>>> def sort2 (item):
        if isinstance (item, list ):
            s=0
```

```
            for el in item:
                    s+=el
            return s/len (item)
    elif isinstance (item, str ):
            return ord (item[0])
    else:
          return item
```

После вызова метода сортировки

```
>>> c_list.sort(key=sort2)
```

получаем отсортированный список

```
>>> c_list
[[3, 7, 6], 6, 6.7, [1.6, 20], 'web', 'Питон'].
```

Чтобы сделать копию результата сортировки (без изменения самого списка), необходимо вместо использования метода *sort()* применить встроенную функцию *sorted()*, которая имеет те же необязательные параметры:

```
>>> d_list=sorted(c_list, key=sort2, reverse=1)
>>> d_list
['Питон', 'web', [1.6, 20], 6.7, 6, [3, 7, 6]]
>>> c_list
[[3, 7, 6], 6, 6.7, [1.6, 20], 'web', 'Питон']
```

Использование функций при фильтрации и формировании данных

Использование функции filter()

Для фильтрации данных итерабельного типа может быть использована встроенная функция *filter(function, iterable)*, которая возвращает итератор, формируемый из тех элементов аргумента iterable, для которых первый аргумент – функция function имеет значение True.

В качестве примера рассмотрим задачу формирования простых чисел. Сначала разработаем функцию *simple()*, которая возвращает значение True, если ее аргументом является простое числом, и False – в противном случае:

```
>>> def simple (n):
        for _ in range (2, n):
            if n% _==0:
                return False
        return True
```

Функция *simple(n)* имеет параметр *n*, являющийся целым числом, и проверяет остатки от деления этого числа на числа от 2 до *n-1*. Если все остатки от деления не равны 0, т.е. число *n* не делится нацело ни на одно из этих чисел и, следовательно, является простым, то возвращается значение True. Иначе возвращается значение False.

Теперь используем функции *filter()* и *simple()* для формирования простых чисел в диапазоне от 2 до 100:

```
>>> list (filter (simple, range (2,100)))
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

Функция *filter()* возвращает значения в виде итератора, поэтому, так же, как и для объектов типа *range*, необходимо конкретизовать с помощью конструктора тип данных, например, в виде списка.

Использование функции map()

Для формирования данных может быть использована встроенная функция *map(function,iterable)*, которая имеет те же аргументы, что и функция *filter()*, но выполняет не фильтрацию, а формирование с помощью функции *function* элементов итератора из элементов iterable.

В качестве примера рассмотрим формирование чисел вида *x*x-1*, заданного *lambda* функцией, в диапазоне от 1 до 10, представленных в виде кортежа:

```
>>> tuple (map (lambda x:x*x-1, range (1,10)))
(0, 3, 8, 15, 24, 35, 48, 63, 80)
```

Использование оператора yield

Оператор *yield*, используемый в теле функции, как и оператор *return* завершает ее работу, но в отличие от оператора *return* возвращает генератор. Рассмотрим задачу генерации кубов чисел, с использованием оператора *yield*:

```
>>> def create_gen ():
        l= range (1,11)
        for x in l:
            yield x**3

>>> my_gen=create_gen()
>>> my_gen
<generator object create_gen at 0x01E2B030>
>>> for x in my_gen:
        print (x, end= ' ')
1 8 27 64 125 216 343 512 729 1000.
```

Начиная с версии 3.3 языка Python, появилась возможности делегировать часть операций генератора другому генератору с помощью оператора *yield from*. В качестве примера создадим функцию, которая формирует генератор, состоящий из двух других генераторов, которые передают ему часть своих операций:

```
>>> def gen (x):
        yield from range (x,0,-1)
```

```
    yield from range (x)
>>> list (gen(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4].
```

Пространства имен и области видимости

Пространство имен. Пространство имен представляет собой отображение имен (идентификаторов) в объекты. По функциональности пространства имен эквивалентны словарям и реализуются в виде словарей.

В языке Python всегда присутствуют три пространства имен:

- *глобальное пространство имен модуля;*
- *локальное пространство имен, указанных в определении функции или класса;*
- *пространство имен встроенных функций, исключений и других объектов, хранящихся в модуле builtins.*

Важно понимать, что между именами в разных пространствах имен нет связи. Например, два модуля могут определить функции с именем “max_item”, не создавая при этом путаницы – пользователь должен ссылаться на них с использованием имени модуля в качестве префикса.

Под словом атрибут (англ. attribute) будет подразумеваться любое имя, следующее после оператора точки. Имена в модулях являются атрибутами модуля: в выражении *mod_name.func_name*, *mod_name* является объектом - модулем и *func_name* является его атрибутом.

В этом случае имеет место прямое соответствие между атрибутами модуля и глобальными именами, определенными в модуле: они совместно используют одно и то же пространство имен.

Пространства имен создаются в разные моменты времени и имеют разную продолжительность жизни:

- *пространство имен, содержащее встроенные имена, создается при запуске интерпретатора и существует все время его работы;*
- *глобальное пространство имен модуля создается при его считывании и, обычно, также существует до завершения работы интерпретатора;*
- *операторы, выполняемые на верхнем уровне, то есть в интерактивном режиме или путем считывания из файла в сценарном режиме, рассматриваются как часть модуля __main__, который автоматически подключается при запуске интерпретатора и имеет собственное глобальное пространство имен.*

Отметим, что этот объект-модуль имеет скрытый атрибут __dict__, содержащий словарь, используемый для реализации пространства имен модуля, который не является глобальным именем.

- *локальное пространство имен функции создается при вызове функции и удаляется при выходе из нее (при этом возвращается значение функции или генерируется исключение, которое не обрабатывается внутри функции).*

При рекурсивном вызове функции создается собственное локальное пространство имен для каждого вызова.

Область видимости

Каждому пространству имен соответствует область видимости (англ. scope) – фрагмент программы, в котором пространство имен доступно непосредственно, где имена указываются без использования оператора точки.

Необходимо подчеркнуть, что область видимости определяется по тексту: глобальная область видимости функции, определенной в модуле соответствует пространству имен этого модуля, независимо от того, откуда или под каким псевдонимом функция была вызвана.

Является ли имя локальным или глобальным определяется в момент компиляции, т.е. статически: в отсутствии оператора global имя, добавляемое в блоке кода, является локальным во всем блоке; все остальные имена считаются глобальными. Оператор global заставляет интерпретатор считать указанные имена глобальными.

Несмотря на статическое определение, области видимости используются динамически. В любой момент времени выполнения программы имеется ровно три вложенных области видимости (три непосредственно доступных пространства имен). Сначала поиск имени производится во внутренней области видимости, содержащей локальные имена. Далее – в средней, содержащей глобальные имена модуля. И, наконец, во внешней, содержащей встроенные имена.

Обычно локальная область видимости соответствует локальному пространству имен текущей функции (класса, метода). За пределами функции (класса, метода) локальная область видимости соответствует тому же пространству имен, что и глобальная: пространству имен текущего модуля.

Пространства имен и области видимости при описании функций

Все переменные, созданные внутри функций (в том числе и параметры функций) являются локальными переменными – их область видимости ограничивается описанием функции.

Переменные, созданные в программе вне функций, являются глобальными – они видимы везде в программе, включая области описания функций, т.е. они имеют глобальную область видимости. Глобальные переменные доступны для чтения внутри функций:

```
>>> def f1 ():
        var1=1
        print (var1,var)
>>> var=10
>>> f1()
1 10
```

Однако попытка присвоить переменной *var* в теле функции новое значение приведет к тому, что глобальная переменная с этим именем будет воспринята

интерпретатором как локальная переменная, которая будет иметь в своей локальной области видимости это значение:

```
>>> def f1 ():
        var1=1
        var=25
        print (var1,var)
>>> var=10
>>> f1()
1 25
```

Значение же глобальной переменной *var* осталось неизменным:

```
>>> print (var)
10
```

Чтобы функция получила возможность изменять значение глобальной переменной, необходимо использовать оператор *global*:

```
>>> def f1 ():
        global var
        var1=1
        var=25
        print (var1,var)
>>> var=10
>>> f1()
1 25
>>> print (var)
25
```

Определение пространства имен в модулях и функциях

Для получения пространства имен необходимо вызвать встроенную функцию *dir([object])*, которая возвращает список имен, отсортированный по алфавиту:

- *если аргумент функции dir() не указан, она возвращает список имен текущей области видимости;*
- *если объект object (аргумент функции) имеет метод __dir__(),*

то вызывается этот метод, который должен возвратить список атрибутов объекта;

- *если для объекта object метод __ dir__() не определен, функция*

старается собрать как можно больше информации, используя тип объекта и его атрибут __dict__() (если он указан).

По умолчанию поведение функции *dir()* зависит от типа объектов:

- *если объект является модулем, то возвращаемый список содержит имена атрибутов модуля;*
- *если объект является типом или классом, то возвращаемый список содержит имена его атрибутов и рекурсивно имена атрибутов его базовых классов;*

- *в других случаях возвращаемый список содержит имена атрибутов объекта, имена атрибутов его класса и рекусивно имена атрибутов базовых классов его класса.*

При выполнении программы верхнего уровня текущей областью видимости будет текст модуля *__main__*, загружаемый в память при вызове интерпретатора.

Поэтому функция *dir()*, вызванная без аргумента, возвратит список атрибутов этого модуля:

```
>>> dir ()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
```

Поскольку в программе не были объявлены имена ни переменных, ни функций, ни классов, то содержащиеся в списке атрибуты представляют собой пространство имен модуля *__main_*, изначально для него заданное.

Для получения дополнительной информации можно вызвать встроенные функции *globals()* и *locals()*, которые возвращают соответственно глобальное и локальное пространства имен модуля в виде словарей, где ключи представлены именами (те же имена, что возвращает функция *dir()*), а значения – значениями соответствующих объектов:

```
>>> globals ()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__', '__doc__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__package__': None,
'__spec__': None}
>>> locals ()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__', '__doc__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__package__': None,
'__spec__': None}
```

Результаты работы функций показывают, что:

- *текущий модуль имеет имя (__name__) – '__main__';*
- *строка документации (__doc__) – не указана;*
- *загрузчик модуля (__loader__) – '_frozen_importlib.BuiltinImporter' помещен в память;*
- *пакет (__package__) – не указан;*
- *спецификация для загрузки модуля (__spec__) – не указана;*
- *для модуля пространства глобальных имен и локальных имен, а также значение атрибута __dict__ совпадают.*

Добавим в программу оператор присваивания, задающий значение 25 глобальной переменной *x*:

```
>>> x=25
25
```

И определим теперь пространство имен модуля:

```
>>> dir ()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', 'builtins', 'x'].
```

Пространств имен модуля изменилось – к нему добавилось имя переменной *x*. Имя этой переменной и ее значение добавились, естественно, и в словарь глобальных имен модуля:

```
>>> globals ()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__', '__doc__': None,
'x': 25, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__package__': None,
'__spec__': None}
```

Добавим теперь в программу описание и вызов функции *func()*, содержащей глобальную переменную *y*, локальную переменную *loc_x* и вызовы функций *dir()*, *globals()* и *locals()* с выводом возвращаемых ими значений:

```
>>> x=25
>>> def func ():
        loc_x=100
        global y
        y=7
        print ( dir ())
        print ( globals ())
        print ( locals ())
>>> func()
['loc_x']
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__', 'func': <function
func at 0x023A0588>, '__doc__': None, 'x': 25, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__package__': None, '__spec__': None, 'y': 7}
{'loc_x': 100}
```

Результаты работы функции *func()* показывают, что для локальной области видимости (блока функции):

- *пространство имен состоит из имени 'loc_x';*
- *к глобальному пространству имен добавились имя функции func() и имя переменной y, объявленное в теле функции как глобальное;*
- *локальное пространство имен состоит из имени 'loc_x'.*

Имена могут быть добавлены (только в локальное пространство имен) следующими способами:

- *передача формальных аргументов функции;*
- *использование инструкции import;*
- *определение класса или функции (добавляет в локальное пространство имен имя класса или функции);*

- *использование оператора присваивания;*
- *использование цикла for (в заголовке указывается новое имя);*
- *указывая имя во второй позиции после ключевого слова except.*

Если глобальное имя не найдено в глобальном пространстве имен, его поиск производится в пространстве встроенных имен, которое является пространством имен модуля *builtin*. Этот модуль (или словарь определенных в нем имен) доступен под глобальным именем текущего блока *__builtins__*. Если же имя не найдено в пространстве встроенных имен, генерируется исключение NameError.

Создание модулей пользователя

По мере возрастания сложности программ появляется необходимость разбить их на несколько файлов для облегчения поддержки. Также может возникнуть необходимость в многократном использовании написанных функций в нескольких программах, не копируя их определения в каждую из программ.

Во всех этих случаях оправдана разработка модулей. Модули выполняют как минимум три важных функции:

- *повторное использование кода: такой код может быть загружен много раз во многих местах;*
- *управление адресным пространством: модуль – это высокоуровневая организация программ, это пакет имен, который избавляет вас от конфликтов. Поэтому модуль – это средство для группировки системных компонентов;*
- *глобализация сервисов и данных: для реализации объекта, который используется во многих местах, достаточно написать один модуль, который будет импортирован.*

Модуль на языке Python – это файл с расширением *.py*, содержащий описания функций, классов, переменные и другие объекты, как правило, общей области применения. Программы на языке Python тоже представляют собой файлы с расширением *.py*, но в отличие от модулей, которые используются для подключения к другим программам, предназначены для непосредственного выполнения.

Для создания пользовательского модуля необходимо сформировать файл с расширением *.py*, содержащий необходимые объекты. В качестве примера создадим модуль с именем *my_module* в виде файла *my_module.py*, куда запишем описание функции *fibo()*.

Для подключения созданного модуля к программе, как и любого другого используется оператор *import*:

```
>>> import my_module
```

Оператор *import* позволяет также подключать сразу несколько модулей, перечисляя их через запятую.

При вызове функции *fibo()* ее необходимо указать как метод модуля *my_module*:

```
>>> my_module.fibo(12)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

Непосредственный вызов функции

```
>>> fibo(12)
```

вызовет исключение:

```
NameError: name 'fibo' is not defined
```

Это происходит потому, что пространство имен модуля не входит в пространство имен программы. Для того, чтобы вызвать функцию *fibo()* непосредственно, так, как если бы она была описана в самой программе, необходимо использовать оператор *from*

```
>>> from my_module import fibo
>>> fibo(14)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

В операторе *from* можно задавать одновременно сразу несколько объектов модуля, перечисляя их через запятую.

Компиляция и выполнение фрагментов кода

Python позволяет во время работы программы динамически осуществлять компиляцию и выполнение отдельных фрагментом кода. Для этого используются следующие встроенные функции *compile(), exec() и eval().*

Функция compile()

compile (source, filename, mode, flags=0, dont_inherit=False, optimize=-1*)* – компилирует исходный код, указанный аргументом *source*, в код или объект AST.

Объекты кода могут быть выполнены с помощью функций *exec()* или *eval()*. Исходный код source может быть строкой, типом bytes или AST объектом (для получения информации о том, как работать с AST объектами необходимо обратиться к документации модуля *ast*).

Аргумент *filename* должен указывать имя файла, откуда будет прочитан исходный код (если код находится в интерактивной среде разработки – указывается значение *'<string>'*).

Аргумент *mode* указывает вид кода, который должен быть компилирован. Это может быть *'exec'*, если код состоит из последовательности операторов (используется в большинстве случаев), *'eval'* – если код состоит из единственного выражения (если указать оператор, то возникнет исключение *"invalid syntax"*), или *'single'*, если код состоит из единственного оператора, используемого в интерактивном режиме (если операторов больше – они не будут выполняться).

Необязательные аргументы flags и dont_*inherit* определяют, какие операторы *future_statement* (см. PEP 236 http://legacy.python.org/dev/peps/pep-0236/) влияют на компиляцию исходного кода.

Необязательный аргумент *optimize* указывает уровень оптимизации компилятора. Значение, принимаемое по умолчанию (-1), выбирает уровень оптимизации интерпретатора. Явно задавать можно следующие уровни:

- *0 – нет оптимизации, значение __debug__ равно True;*
- *1 – функции assert() удалены, значение __debug__ равно False;*
- *2 – строки документации удалены также.*

Функция *compile()* вызывает исключение *SyntaxError*, если компилируемый код неверен, и исключение *TypeError*, если исходный код содержит байты null.

Приведем пример компиляции небольшого фрагмента кода:

```
>>> my_code= compile ('''
x=5
y=x**3+1
print(y) ''', '<string>', 'exec')
```

Переменная *my_code* указывает на скомпилированный фрагмент кода. Определим ее тип:

```
>>> type (my_code)
<class 'code'>
```

Таким образом скомпилированный код имеет тип (класс) *code*.

Функция exec()

Встроенная функция *exec(object[, globals[, locals]])* поддерживает динамическое выполнение кода языка Python. Аргумент *object* должен быть строкой или объектом типа *code*. Если это строка, то производится ее грамматический разбор, как это делается с операторами языка. После чего происходит ее выполнение (до завершения или до появления ошибки). Если это объект типа *code*, то он просто выполняется.

Приведем два примера использования функции *exec()*: один с аргументом в виде строки:

```
>>> x=1.25
>>> exec ( 'y=x*x+x+1' )
>>> y
3.8125
```

другой – с аргументом в виде скомпилированного кода. Для этого воспользуемся переменной *my_code*, полученной выше:

```
>>> exec (my_code)
126
```

Функция eval()

Аргументами встроенной функции *eval(expression, globals=None, locals=None)* являются: строка *expression*, задающая выражение, и необязательные

словари *globals* и *locals*, указывающие пространства имен соответственно глобальных и локальных переменных. При выполнении функции осуществляется грамматический разбор выражения и вычисление его значения:

```
>>> x=5
>>> eval ( 'x'+1, { 'x' :1})
2
```

Если аргумент *locals* опущен, значения берутся из словаря *globals*. Если опущены оба необязательных аргумента, выражение вычисляется в том окружении, в котором функция *eval()* была вызвана:

```
>>> x=5
>>> eval ( 'x+1' )
6.
```

Ранее была рассмотрена функция *val(x, y, op)*, выполняющая заданные арифметические операции (*op*) над двумя числами (*x* и *y*). Использование функции *eval()* в блоке функции *val()* (новый вариант функции – *new_val()*) не только делает описание функции намного компактнее:

```
>>> def new_val (x, y, op):
        return eval ( str (x)+op+ str (y)),
```

но и существенно расширяет функциональные возможности функции *val()*, поскольку функция *new_val()* может выполнять не только те операции, которые были предусмотрены для функции *val()*:

```
>>> new_val(3, 2, '/')
1.5,
```

но и любые двуместные операции для операндов *x* и *y*:

```
>>> new_val(3,2,'**')
9.
```

Причем даже такие, которых нет среди стандартных операций языка Python:

```
>>> new_val(3, 2, '**2+')
11.
```

В заключение отметим, что *eval(repr(object))* возвращает *object*. Например:

```
>>> eval ( repr ( "cat" ))
'cat'.
```

1.3 Задания на выполнение лабораторной работы

Разработать программу на языке Python, в которой:

- *1 Определена и выполнена функция func1() с аргументами в виде списка чисел (целых и с плавающей точкой), которая выполняет операцию, заданную колонкой "Операция" табл. №1):*
  - *1 – определение суммы квадратов элементов списка;*
  - *2 – определение максимального числа среди элементов списка;*

- *3 – определение квадратных корней элементов списка;*
- *4 – определение минимального числа среди элементов списка;*
- *5 – определение среднего значения элементов списка;*
- *6 – определение разности между суммой четных чисел и суммой нечетных чисел (если число нецелое – привести к целому);*
- *7 – определение разности между суммой кубов и суммой квадратов элементов списка.*

- *2.1 Создан словарь a_dict (числом элементов не меньше 8), ключи которого именуются произвольно, а значения заданы в виде, указанном колонкой "Вид значений" табл. №1):*
  - *1 – латинские буквы;*
  - *2 – цифры;*
  - *3 – буквы кириллицы;*
  - *4 – наименования встроенных функций;*
  - *5 – наименования операторов.*

При этом отдельные ключи (числом не меньше трех) должны иметь одинаковое значение.

- *2.2 Определена функция func2(), которая имеет два аргумента, первый – в виде словаря, второй – указывает значение ключа словаря. Функция func2() возвращает список ключей словаря, значения которых совпадают со значениями второго аргумента.*
- *2.3 Проверена работа функции func2(), при вызове которой в качестве первого аргумента задан словарь a_dict, а в качестве второго аргумента – значение, которое имеют несколько ключей словаря.*
- *3.1 Создан список a_list, элементы которого имеют тип, указанный колонкой "Тип" табл. №1):*
  - *1 – числа;*
  - *2 – логические значения;*
  - *3 – строки;*
  - *4 – списки с элементами в виде чисел;*
  - *5 – списки с элементами в виде строк.*
- *3.2 Определена функция func3(), которая преобразует каждый элемент заданного списка a_list в целое число (механизм преобразования – на усмотрение студента).*
- *3.3 Выполнена с использованием функции func3() и метода sort() сортировка элементов списка a_list:*
  - *для четных номеров индивидуального задания – по возрастанию;*
  - *для нечетных номеров индивидуального задания – по убыванию.*

- *4.1 Задана строка str_code, содержащая небольшой фрагмент кода на языке Python и получен скомпилированный с помощью встроенной функции compile() код – comp_code.*
- *4.2 С помощью встроенной функции exec() код comp_code исполнен.*
- *5 Для программы и одной из функций программы определено множество глобальных имен и множество локальных имен.*

Таблица 1 – Перечень индивидуальных заданий

Номер п/п	**Операция**	**Вид значений**	**Тип**
1	1	5	1,5
2	2	4	2,3
3	3	3	2,4
4	4	2	2,5
5	5	1	3,4
6	6	5	3,5
7	7	4	1,5
8	1	3	2,3
9	2	2	2,4
10	3	1	2,5
11	4	5	3,4
12	5	4	3,5
13	6	3	1,5
14	7	2	2,3
15	1	1	2,4
16	2	5	2,5
17	3	4	3,4
18	4	3	3,5
19	5	2	1,5
20	6	1	2,3

2 ЛАБОРАТОРНАЯ РАБОТА № 6

Объектно-ориентированное программирование на языке Python. Создание пользовательских классов

2.1 Цель лабораторной работы

Целью лабораторной работы является получение навыков ООП на языке Python. Рассмотрение способов создания классов, объектов и их использования.

2.2 Теоретические сведения

Классы в языке Python – это объекты первого рода, т.е. их можно создавать в процессе выполнения программы, сохранять в переменной, передавать в качестве аргументов функций, возвращать как результат из функций и др.

Объявление класса

Объявление класса осуществляется с помощью оператора *class*, в первой строке которого (заголовке), указывается служебное слово *class*, за которым следует имя класса. По соглашению имена классов должны начинаться с заглавной буквы. В скобках указывается базовый класс (его называют также родительским классом или надклассом (англ. superclass)) – это может быть имя одного из ранее созданных классов или встроенный тип самого верхнего уровня – *object.* Отметим, что, если базовым классом является *object*, его можно не указывать. В этом случае скобки не нужны. Заголовок заканчивается двоеточием. Например,

```
>>>class My_class (object):
```

или

```
>>> class My_class:
```

Класс может состоять только из заголовка. Но поскольку синтаксис требует, чтобы после заголовка класса следовало описание класса, необходимо указать вместо него оператор *pass*, который ничего не делает, но показывает наличие описания:

```
>>> class Min_class:
        pass
```

или

```
>>> class My_class: pass
```

При описании класса второй строкой может быть строка, взятая в тройные кавычки, которая документирует класс (Обычно в этой строке указывается назначение класса, например, так:

```
'''Пример простого класса'''
```

Классы могут иметь атрибуты, которые делятся на атрибуты-переменные и атрибуты-методы. Атрибуты-переменные указываются с помощью оператора присваивания и являются ссылками на объекты, значения которых им было присвоено.

Форма описания атрибутов-методов класса совпадает с описанием функций, но первым аргументом каждого метода (за исключением статических методов - см. ниже) должно быть слово *self*, которое возвращает ссылку на объект, вызывающий данный метод. Можно отметить, что *self* в языке Python не является ключевым словом и используется по соглашению.

Добавив в класс My_class описания атрибута-переменной a и атрибута-метода *f(),* получим следующее объявление класса:

```
>>> class My_class:
        '''Пример простого класса'''
```

```
    a=25
    def f (self):
        print ( "Привет всем!" )
```

При создании класса *формируется его пространство имен*, которое состоит из имен его атрибутов. В языке Python класс является объектом класса *type*:

```
>>> type (My_class)
<class 'type'>,
```

который поддерживает два вида операций:

- *доступ к своим атрибутам;*
- *инстанцирование, то есть. создание экземпляров класса (англ. instances).*

Для доступа к атрибутам класса используется обычный для объекта синтаксис с использованием оператора точки – *Obj.name*, где: *Obj* – имя объекта, *name* – имя атрибута объекта.

Например, получить значение атрибута-переменной *a* класса *My_class* можно следующим образом:

```
>>> My_class.a
25
```

Если попытаться вызвать атрибут-метод *f()* класса *My_class*, то это вызовет ошибку:

```
>>> My_class.f()
TypeError: f() missing 1 required positional argument: 'self',
```

потому, что так вызывать (с именем класса) можно только статические методы классов, а методы, в которых указан параметр self, могут вызывать только объекты – экземпляры класса.

Каждый класс имеет преопределенные атрибуты (табл. 1)

Приведем некоторые преопределенные атрибуты класса *My_class*:

```
>>> My_class._ _name_ _
'My_class'

>>> My_class._ _bases_ _
(<class 'object'>,)

>>> My_class._ _doc_ _
'Пример простого класса'

>>> My_class._ _module_ _
'_ _main_ _'.
```

Хотя базовый класс явно задан не был, по умолчанию таковым является класс object, указанный в виде кортежа.

Python позволяет создавать классы динамически. Для этого используется встроенная функция *type()*, которая, если задан один аргумент – *type(object)*, возвращает тип (класс) объекта *object*. Если при вызове функции указано три

аргумента – *type(name, bases, dict)*, то она возвращает класс с именем *name*, который наследует классы, заданные кортежем *bases*, и имеет атрибуты, заданные словарем *dict*:

```
>>> A= type ("A", (object,), dict (a=25)).
```

Этот класс эквивалентен следующему классу, созданному статически:

```
>>> class A:
        a=25.
```

Таблица 1 - Преопределенные атрибуты класса

Атрибут	**Тип**	**Доступ**[(1)]	**Описание**
_ _dict_ _	словарь	R/W	пространство имен класса
_ _name_ _	строка	R	имя класса
_ _qualname_ _	строка	R	полное имя класса или функции
_ _bases_ _	кортеж классов	R	классы, от которых наследуется данный класс
_ _doc_ _	строка или None	R/W	строка документации класса
_ _module_ _	строка	R/W	имя модуля, в котором данный класс был определен

Примечание (1): R/W – разрешены запись и чтение, R – разрешено только чтение.

Создание объектов – экземпляров класса

После объявления класса можно выполнить операцию инстанцирования класса. Создадим один экземпляр этого класса – объект *m1*:

```
>>> m1=My_class()
```

Экземпляры имеют следующие преопределенные атрибуты (табл. 2).

Таблица 2 – Преопределенные атрибуты экземпляра класса

Атрибут	**Тип**	**Доступ**	**Описание**
_ _dict_ _	словарь	R/W	пространство имен экземпляра
_ _class_ _	строка	R/W	класс этого экземпляра

Определим класс объекта *m1*:

```
>>> m1._ _class_ _
<class '_ _main_ _.My_class'>.
```

Важно понимать, что при создании каждого экземпляра класса создается новое пространство имен, принадлежащее этому экземпляру, которое не

совпадает с пространством имен самого класса. Поэтому даже если атрибуты класса и экземпляра класса имеют одинаковое имя, они ссылаются на разные объекты и могут иметь разные значения. Например, класс *My_class* и экземпляр этого класса *m1* имеют следующие пространства имен:

- ***My_class:***

```
>>> My_class._ _dict_ _
mappingproxy({'f': <function My_class.f at 0x023C0588>, '_ _doc_ _': 'Пример
простого класса', '_ _module_ _': '_ _main_ _', 'a': 25, '__dict__': <attribute '_ _dict_ ' of
'My_class' objects>, '_ _weakref_ _': <attribute '__weakref__' of 'My_class' objects>});
```

- ***m1:***

```
>>> m1._ _dict_ _
{}
```

Пространство имен класса ***My_class*** содержит:

- ***имена преопределенных атрибутов класса:***
  '_ _doc_ _';
  '_ _module_ _';
  '_ _main_ _',
  '_ _dict_ _';
  '_ _weakref_ _' (weakref – слабая ссылка). Эти ссылки отличаются от обычных тем, что, если на объект указывают только слабые ссылки, то системе сборки мусора разрешается при необходимости удалять этот объект и освобождать память;

- ***имена атрибутов класса:***
  'a';
  'f'.

Пространство имен экземпляра класса *m1* пусто. Тем не менее при обращении к атрибуту *a* этого объекта

```
>>> m1.a
25
```

получаем значение 25, т.е. значение атрибута класса. Это происходит потому, что интерпретатор сначала ищет искомое имя в адресном пространстве экземпляра класса. Если его там нет, то – в адресном пространстве самого класса.

После присвоения объекту нового значения:

```
>>> m1.a=100
```

его адресное пространство изменилось:

```
>>> m1._ _dict_ _
{'a': 100}
```

и значения атрибута *a* класса и экземпляра класса стали отличаться:

```
>>> m1.a
100
```

и

```
>>> My_class.a
25
```

Вызов метода *f()* как метода экземпляра класса не вызывает ошибки:

```
>>> m1.f()
Привет всем!
```

Описание класса в языке Python является динамическим, то есть в него в последствии можно добавить новые атрибуты, например, атрибут *new_a*:

```
>>> My_class.new_a= 'Новый'
```

или удалить старые (*f*):

```
>>> del My_class.f
```

После создания объекта *m2*, экземпляра измененного класса *My_class*:

```
>>> m2=My_class ()
```

проверим его атрибуты:

```
>>> m2.new_a
'Новый'
>>> m2.f()
AttributeError: 'My_class' object has no attribute 'f'
```

То есть, атрибут *new_a* в класс добавлен, а атрибут *f* из класса удален.

Отметим, что преопределенный атрибут *__qualname__* был введен в язык Python, начиная с версии 3.3. Он указывает полное имя (англ. qualified name) класса или функции.

Для классов и функций верхнего уровня атрибут *__qualname__* совпадает с *__name__*.

Для проверки создадим класс *A*, содержащий класс *B*:

```
>>> class A:
        class B: pass
```

и определим для классов *A* и *B* значения атрибутов *__name__* и *__qualname__*:

```
>>> A._ _name_ _
'A'
>>> A._ _qualname_ _
'A'
>>> A.B._ _name_ _
'B'
>>> A.B._ _qualname_ _
'A.B'
```

То есть для класса верхнего уровня *A* значения атрибутов _ *_name_* _ и _ *_qualname_* _ совпадают, а для встроенного класса *B* – нет.

Использование атрибутов и методов при объявлении класса.

Использование специальных методов.

В языке Python имеются специальные методы, предназначенные для использования во время объявления класса:

- *__new__(cls[, . .]) – статический метод класса, вызывается первым при создании экземпляра класса, создает новый объект, возвращая на него ссылку. Первый параметр cls указывает на класс, остальные параметры являются необязательными и, если указаны, передаются методу __init_ ();*

- *__init__(self[, . .]) – вызывается при создании экземпляра класса. Первый параметр self (хотя и не является ключевым словом языка, но менять его на другое имя не следует) указывает на созданный объект, остальные аргументы получает от метода __new__(). Устанавливает начальные значения атрибутов объекта. Никогда не возвращает значения. В литературе метод __init__ иногда называют конструктором класса, но, строго говоря, это не так – он реализует только часть функций, выполняемых конструкторами таких языков, как C++ или Java – осуществляет инициализацию атрибутов объекта, но не создает сам объект, поскольку эту функцию, как указывалось выше, выполняет метод __new__. Поэтому называя в языке Python метод __init__ "конструктором класса", необходимо об этом помнить;*

- *__str__() – вызывается функциями str(), print() и format(). Возвращает строковое представление объекта;*

- *__del__() – вызывается при уничтожении экземпляра класса, то есть является деструктором класса. Отметим, что метод __del__ не влияет на*

работу оператора del, но влияет на работу системы сборки мусора.

Метод *__new__()* при создании пользовательских классов используется нечасто. Можно привести две области его применения.

Во-первых, когда нужно выполнить какие-то действия до выполнения метода *__init__()*. Это бывает необходимо, например, при наследовании классов неизменяемых типов.

Во-вторых, когда необходимо контролировать процесс создания экземпляров класса. Примером может служить класс, который инстанцирует только один объект:

```
>>> class One:
        obj= None
        def __new__ (cls,*p,**kw):
                if cls.obj is None:
```

```
                cls.obj= object.__new__(cls,*p,**kw)
            return cls.obj
```

При создании каждого экземпляра класса *One* первым вызывается метод *__new__* класса, который проверяет значение атрибута *obj* класса One.

Если оно равно None, то есть создается первый экземпляр класса, вызывается метод *__new__* базового класса *object*. Он создает объект – экземпляр класса, ссылка на который присваивается атрибуту *obj* класса *One*. Это же значение возвращается методом *__new__* класса *One*.

При создании последующих экземпляров класса, поскольку значение атрибута *obj* уже не равно None, метод *__new__* базового класса не вызывается и вновь созданный объект получает ссылку с тем же значением, которое хранится в атрибуте *obj*.

Таким образом все последующие экземпляры класса *One* являются одним и тем же объектом. Это можно проверить. Создадим два экземпляра класса *One*:

```
>>> one1=One()
>>> one2=One()
```

и проверим их идентичность:

```
>>> id (one1)
36932784
>>> id (one2)
36932784
```

Можно показать, что ссылки *one1* и *one2* – это ссылки на один и тот же объект и по-другому:

```
>>> one1 is one2
True
```

Использование специальных методов *__init__()*, *__str__()* и *__del__()* показано в объявлении класса *Virt_zoo*, который описывает поведение виртуальных зверьков:

```
>>> class Virt_zoo ( object ):
        '''Виртуальные зверьки '''
        def __init__ (self, name):
                print ( 'Появился новый зверёк.' )
                self.name=name
        def  __str__(self):
                return 'Класс – Virt_zoo, имя зверька – ' +self.name
        def __del__ (self):
                print ( 'Зверек исчез.' )
```

С помощью метода*__ init__ ()*, который вызывается при каждом создании экземпляра класса), на экран выводится сообщение об этом и выполняется инициализация атрибута *name*, указывающего имя этого зверька.

Отметим, что первым параметром каждого метода в объявлении класса, за исключением статических методов, является слово *self*, которое при вызовах этих методов не указывается.

Метод *__str__*вызывается при использовании функции *print(obj)* и возвращает данные об объекте *obj*, в данном случае это имя класса и имя созданного зверька.

С помощью деструктора класса (метода *__del__*) осуществляется вывод на экран сообщения "Зверек исчез." при удалении объекта класса *Virt_zoo*.

После объявления класса можно выполнять операции его инстанцирования, то есть создания объектов класса.

Создаются два экземпляра класса *Virt_zoo* – объекты *v1* и *v2*:

```
>>> v1=Virt_zoo( 'Колобок' )
Появился новый зверек.
>>> v2=Virt_zoo( 'Серый волк' )
Появился новый зверек.
```

При создании каждого объекта на экран выводится, предусмотренное конструктором, сообщение "Появился новый зверек.".

По умолчанию все методы и атрибуты класса являются открытыми (public) и клиентский код может обращаться к этим методам и получать значения атрибутов.

Поэтому получим для каждого созданного объекта значение его атрибута *name*:

```
>>> v1.name
'Колобок'
>>> v2.name
'Серый волк'
```

Указав имя класса в функции *print()*, получим данные, определяемые *__str__()*, заданные по умолчанию для класса *Virt_zoo*:

```
>>> print (Virt_zoo)
<class ' __main__. Virt_zoo'>,
```

то есть класс Virt_zoo объявлен в модуле __main__.

Указав имя объекта в функции *print()*, получим данные об объекте, определяемые пользовательским специальным методом *__str__()*:

```
>>> print (v1)
Класс – Virt_zoo, имя зверька – Колобок
```

После удаления созданного объекта *v1*

```
>>> del v1
```

в соответствии с работой пользовательского специального метода *__del__()* появляется сообщение:

```
Зверек исчез.
```

И последующее обращение к атрибуту объекта вызовет исключение:

```
>>> v1.name
NameError: name 'v1' is not defined
```

Как уже отмечалось, после объявления класса в него можно добавлять новые атрибуты и методы.

Поэтому создадим метод *talk()*:

```
>>> def talk (self):
        print ( 'Привет! Меня зовут ' +self.name+ '.' ),
```

который при вызове выводит на экран сообщение, сделанное зверьком, и добавим этот метод в класс *Virt_zoo*:

```
>>> Virt_zoo.talk=talk
```

После этого вызовем для каждого созданного объекта класса *Virt_zoo* метод *talk()*:

```
>>> v1.talk()
Привет! Меня зовут Колобок.
>>> v2.talk()
Привет! Меня зовут Серый волк.
```

Применение атрибутов класса и статических методов

Атрибуты позволяют присваивать разные значения разным объектам одного и того же класса. Но бывает и такая информация, которая относиться не к индивидуальным объектам, а ко всему классу. Например, необходимо следить за количеством созданных зверьков. Можно было для этого присвоить каждому объекту атрибут *total*, но тогда при создании каждого нового объекта необходимо будет изменять значение этого атрибута у всех уже созданных объектов. А это сделать непросто.

Язык Python позволяет создавать значения, связанные с классом, которые называются атрибутами класса, и создавать методы, связанные с классом, называемые статическими.

Внесем в вышеописанный класс *Virt_zoo* следующие изменения:

- *не будем указывать, что создаваемый класс является наследником класса object (это ни на что не влияет);*
- *добавим атрибут класса:*

total=0 # Число созданных зверьков;

- *добавим описание статического метода status(), который выводит на экран текущее количество зверьков:*

```
@ staticmethod
def status ():
    print ('Всего зверьков – ', Virt_zoo.total)
```

(для указания того, что метод является статическим используется декоратор staticmethod);

- *добавим в конструктор увеличение значения атрибута total на 1:*

Virt_zoo.total+=1;

- *уберем методы __str__ и __del__ и в дальнейших примерах использоваться не будут.*

После внесенных изменений описание класса выглядит следующим образом:

```
>>> class Virt_zoo:
        ''' Виртуальные зверьки '''
        #переменная класса
        total=0 # Число созданных зверьков
        def __init__ (self, name):
            print ('Появился новый зверек.')
            self.name=name
            Virt_zoo.total+=1
        @ staticmethod
        def status ():
            print ('Всего зверьков – ', Virt_zoo.total)
        def talk (self):
            print ( 'Привет! Меня зовут ' +self.name, '.')
```

Обращаться к атрибутам класса и статическим методам можно как через созданные объекты, так и непосредственно из класса:

```
>>> v1=Virt_zoo( 'Колобок' )
Появился новый зверек.
>>> v1.total
1
>>> v2=Virt_zoo ( 'Змей Горыныч' )
Появился новый зверек.
>>> Virt_zoo.status()
Всего зверьков – 2
```

Использование закрытых атрибутов и методов

В языке Python используется три вида атрибутов:

- *открытые – предназначенные для использования из клиентского кода;*

• *закрытые – предназначенные для использования внутри описания класса и запрещенные для доступа из клиентского кода (перед именем таких атрибутов ставятся два символа подчеркивания);*

• *атрибуты, предназначенные для использования внутри описания класса, но разрешенные для доступа из клиентского кода (перед именем таких атрибутов ставится один символ подчеркивания).*

С целью использования закрытого атрибута внесем в вышеописанный класс *Vitt_zoo* следующие изменения:

• *добавим в описание конструктора класса Virt_zoo() закрытый атрибут __mood, отражающий настроение созданного зверька;*

• *укажем в методе talk() вывод на экран не только имени зверька, но и его настроения.*

А также удалим из него:

• *описание статического атрибута total вместе с изменением его значения в конструкторе класса;*

• *описание статического метода status().*

После внесенных изменений описание класса выглядит следующим образом:

```
>>> class Virt_zoo:
        '"Виртуальные зверьки"'
        def __init__ (self,name, mood):
            print ( 'Появился новый зверек.' )
            self.name=name
            self.__mood=mood
        def talk (self):
            print ( 'Привет! Меня зовут ' +self.name+ '. Чуствую себя '
                +self.__mood)
```

Проверим использование закрытого аргумента:

```
>>> v1=Virt_zoo( 'Колобок', 'отлично.' )
Появился новый зверек.
>>> v1.talk()
Привет! Меня зовут Колобок. Чувствую себя отлично.
>>> v2=Virt_zoo ('Серый волк', 'голодным.')
Появился новый зверек.
>>> v2.talk()
Привет! Меня зовут Серый волк. Чуствую себя голодным.
```

Попытка непосредственного обращения к атрибуту с именем *mood* или *__mood* ведет к ошибке:

```
>>> v1.mood
AttributeError: 'Virt_zoo' object has no attribute 'mood'
>>> v1.__mood
AttributeError: 'Virt_zoo' object has no attribute '__mood'
```

То есть объект класса *Virt_zoo* не имеет ни атрибута *mood*, ни атрибута *__mood*.

Тем не менее в языке Python имеется возможность обратиться к закрытому аргументу из клиентского кода и получить его значение. Для этого необходимо использовать следующую форму: *объект._ИмяКласса__ИмяАргумента*:

```
>>> print (v1._Virt_zoo__mood)
отлично.
```

Аналогично атрибутам в языке Python используется три вида методов:

- *открытые – предназначенные для использования из клиентского кода;*
- *закрытые – предназначенные для использования внутри описания класса и запрещенные для доступа из клиентского кода (перед именем таких методов ставятся два символа подчеркивания);*
- *методы, предназначенные для использования внутри описания класса, но разрешенные для доступа из клиентского кода (перед именем таких методов ставится один символ подчеркивания).*

Рассмотрим небольшой пример создания и использования закрытого метода:

```
>>> class C:
        def __private_method (self):
                print ( 'Это закрытый метод' )
        def public_method (self):
                print ( 'Это открытый метод' )
                self.__private_method()
>>> c1=C ()
>>> c1.public_method()
Это открытый метод
Это закрытый метод
```

А непосредственное обращение к закрытому методу приводит к ошибке:

```
>>> c1.__private_method ()
AttributeError: 'C' object has no attribute 'private_method'
```

Так же, как и для атрибутов в языке Python имеется возможность обратиться к закрытому методу из клиентского кода. Для этого необходимо использовать следующую форму: *объект._ИмяКласса__ИмяМетода*:

```
>>> c1._C__private_method()
Это закрытый метод
```

Но применять ее (также, как и форму доступа к закрытому атрибуту) конечно не следует.

Управление атрибутами

Python поддерживает возможность управления доступом к атрибутам класса.

Создадим класс D с одним закрытым атрибутом *__x*:

```
>>> class D:
        def __init__ (self):
            self.__x= None
```

и создадим объект *d1:*

```
>>> d1=D ()
```

При попытке обратиться к свойству *x:*

```
>>> d1.x
```

вырабатывается исключение:

```
AttributeError: 'D' object has no attribute 'x'.
```

Аналогичное сообщение получим при попытке обратиться к атрибуту *__x*.

Для предоставления возможности получить доступ к атрибуту *x* из пользовательского кода, представим этот атрибут в виде свойства класса с разрешением на чтение.

Для этого создается метод *x()* с декоратором *@property*:

```
>>> class D:
        def __init__ (self):
            self.__x= None
        @ property
        def x (self):
            return self.__x
```

Снова попытаемся обратиться к свойству *x*:

```
>>> d1=D()
>>> d1.x
```

На этот раз ошибки не возникло. Но на экран ничего не выведено, так как атрибуту *x* изначально было присвоено значение *None*. Попробуем задать этому свойству некоторое другое значение:

```
>>> d1=D()
>>> d1.x=7
AttributeError: can't set attribute
```

Ошибка о невозможности установки атрибута произошла потому, что пользователю не было разрешено изменять значения свойства. Чтобы дать ему такую возможность добавим в класс *D* декоратор *@x.setter* с методом *x()*:

```
>>> class D:
        def __init__ (self):
            self.__x= None
        @ property
        def x (self):
            return self.__x
        @ x.setter
        def x (self,value):
            self.__x=value
```

Теперь попытаемся изменить свойство *x* и определить его новое значение:

```
>>> d1=D()
>>> d1.x=77
>>> d1.x
77
```

Задание нового значения свойству *x* произошло успешно, а попытка удалить это свойство привела к ошибке:

```
>>> del d1.x
AttributeError: can't delete attribute
```

Чтобы дать возможность пользователю удалить свойство (если это необходимо), добавим в описание класса еще один декоратор – *@x.deleter* и еще один метод *x()*:

```
>>> class D:
        def __init__ (self):
            self.__x= None
        @ property
        def x (self):
            return self.__x
        @ x.setter
        def x (self,value):
            self.__x=value
        @ x.deleter
        def x (self):
            del self.__x
```

Зададим новое значение свойству *x* и попытаемся удалить это свойство:

```
>>> d1=D()
>>> d1.x=125
>>> d1.x
125
>>> del d1.x
>>> d1.x
AttributeError: 'D' object has no attribute '_D__x'
```

То есть свойство *x* удалено.

2.3 Задания на выполнение лабораторной работы

Разработать программу на языке Python, в которой:

- *создается класс, описывающий поведение объектов, представляющих (см. колонку "Поведение" табл. 3) таких персонажей:*

  1 – пользователей компьютера;

  2 – литературных персонажей;

  3 – студентов;

  4 – героев мультипликации;

  5 – исторических персонажей;

  6 – персонажей художественных фильмов;

- *класс должен иметь следующие специальные методы: __init__(), __str__() и __del__();*
- *класс должен иметь такие атрибуты и/или методы (см. колонку "Атрибуты/методы"табл. 3):*

  1 – статический метод;

  2 – атрибут класса;

  3 – метод экземпляра класса;

  4 – закрытый атрибут

  5 – закрытый метод

- *осуществляется управление двумя атрибутами класса, для первого устанавливается режим "только чтение", для второго – согласно колонке, "Управление" табл. 3):*

  1 выполняется чтение атрибута и запись в него;

  2 выполняется чтение и удаление атрибута;

  3 выполняется чтение, запись и удаление атрибута;

- *создаются объекты класса и проверяется их работа.*

Таблица 3 – Перечень индивидуальных заданий

Номер п/п	Поведение	Атрибуты/ методы	Управление
1	1	1,3,4	1
2	2	2,3,5	2
3	3	1,3,5	3
4	4	2,3,4	1
5	5	1,3,4	2
6	6	2,3,5	3
7	1	1,3,5	1

8	2	2,3,4	2
9	3	1,3,4	3
10	4	2,3,5	1
11	5	1,3,5	2
12	6	2,3,4	3
13	1	1,3,4	1
14	2	2,3,5	2
15	3	1,3,5	3
16	4	2,3,4	1
17	5	1,3,4	2
18	6	2,3,5	3
19	1	1,3,5	1
20	2	2,3,4	2

3 ЛАБОРАТОРНАЯ РАБОТА № 7

Наследование классов

3.1 Цель лабораторной работы

Целью лабораторной работы является получение навыков ООП на языке Python. Рассмотрение способов создания новых классов путем их наследования

3.2 Теоретические сведения

Консервация классов

Над объектами – экземплярами классов, как и над другими объектами языка Python, могут быть осуществлены операции сериализации (представление объектов в строковом виде) и консервации (запись строкового представления объекта на диск), а также обратные им операции десериализации и деконсервации.

Для выполнения этих операций создается класс *My_pickle*, который имеет два метода:

- *put(self,obj) – осуществляет сериализацию и запись на диск объекта obj в файл, имя которого указывается при создании объекта;*
- *get(self) – осуществляет чтение из указанного файла и десериализацию объекта.*

```
>>> class My_pickle:
        import pickle
        def __init__ (self,file):
            self.file=file
        def put (self,obj):
            f= open (self.file, 'wb')
            My_pickle.pickle.dump(obj,f)
```

```
            f.close()
    def get (self):
            with open (self.file, 'rb' ) as f:
                    return My_pickle.pickle.load(f)
```

Модуль *pickle* описан как атрибут класса. Метод *dump()* модуля *pickle* сериализует данные в открытый файл. Поэтому при доступе к его методам необходимо указывать имя класса: *My_pickle.pickle.dump(obj,f)*.

Для проверки класса *My_pickle* и его методов создадим объект *p1* класса *My_pickle*, для которого в конструкторе класса укажем имя файла *"p1.dat"* текущего каталога:

```
>>> p1=My_pickle("p1.dat"),
```

и объект *v1* класса *Virt_zoo*:

```
>>> v1=Virt_zoo( 'Колобок', 'превосходно.' )
Появился новый зверек.
```

С помощью метода *put()* объекта *p1* осуществим консервацию объекта *v1* на диске в файле *"p1.dat"*:

```
>>> p1.put(v1)
```

С помощью метода *get()* объекта *p1* осуществим деконсервацию данных объекта, сохраненных на диске, в объект *v_new*:

```
>>> v_new=p1.get()
```

И в заключение для объекта *v_new* выполним метод *talk()*:

```
>>> v_new.talk()
Привет! Меня зовут Колобок. Чуствую себя превосходно.
```

Т.е. операции консервации и деконсервации объекта *v1* осуществлены успешно.

Наследование классов. Наследование класса Virt_zoo.

Для проверки наследования классов создается класс *Smart_zoo()* на базе выше рассмотренного класса *Virt_zoo()* (лаб. раб. №6). При этом осуществляется:

- *расширение конструктора нового класса путем:*
  - *добавления параметра where, указывающего место событий;*
  - *вывода значения параметра where;*
  - *вызова конструктора базового класса (Virt_zoo);*
- *перегрузка метода talk();*
- *добавление метода tell().*

```
>>> class Smart_zoo (Virt_zoo):
        def __init__ (self, name, mood, where):
                print (where, end= "")
                super ().__init__(name, mood)
```

```
    def talk (self):
        print ("Я – " +self.name+ ".")
    def tell (self,obj):
        print ("А ты кто?")
        obj.talk()
        print (obj.name, "," obj.name, ".", sep= "", end= "")
        if self.name== "Серый волк":
            print ( " А я тебя съем!" )
        else:
            print (" А давай дружить!")
```

Отметим, что при вызове конструктора класса *Smart_zoo* – метода *__init__* сначала происходит вывод значения нового атрибута *where*, указывающего местонахождение зверюшки.

Затем с помощью встроенной функции *super()* осуществляется вызов метода *__init__* родительского класса *Virt_zoo*, который выполняет все действия, предусмотренные конструктором этого класса.

Функция *super([type[,object_or_type]])* возвращает объект, который делегирует вызов метода родительскому классу, указанному параметром *type*. Это полезно для доступа к наследуемым методам, которые переопределяются в классе.

Если класс является единственным наследником, то параметры функции *super()* можно опустить, то есть для класса *Smart_zoo* выражение:
super (Smart_zoo,self).__init__(name,mood) эквивалентно
super ().__init__(name,mood).

Выполним инстанцирование двух объектов нового класса:

```
>>> s1=Smart_zoo ('Колобок', 'превосходно.', 'Волшебный лес. ')
Волшебный лес. Появился зверек.
>>> s2=Smart_zoo ('Серый волк', 'очень голодным.', 'Там же. ')
Там же. Появился зверек.
```

С помощью специального атрибута *__class__* можно указать для объекта любой класс в иерархии его наследования:

```
>>> s1.__class__=Virt_zoo
```

С помощью методов *talk()* и *tell()* класса *Smart_zoo* реализуем два варианта диалога между двумя созданными зверьками: Колобком (объект s1) и Серым волком (объект s2). В первом варианте разговор начинает Серый волк:

```
>>> s2.talk()
Я – Серый волк.
>>> s2.tell(s1)
А ты кто?
Привет! Меня зовут Колобок. Чувствую себя превосходно.
```

Колобок, Колобок. А я тебя съем!

Для объекта *s1* был указан класс *Virt_zoo* и *s1* выполнил метод *talk()* в "стиле" этого класса, то есть вывел на экран строку "Привет! Меня зовут Колобок. Чуствую себя превосходно.", вместо "Я – Колобок.".

Во втором варианте разговор начинает Колобок:

```
>>> s1.talk()
Я – Колобок.
>>> s2.tell(s1)
А ты кто?
Я – Серый волк.
Серый волк, Серый волк. А давай дружить!
```

Множественное наследование

Python поддерживает также множественное наследование. Рассмотрим создание класса *New_zoo()*, который имеет одновременно два базовых класса: *Smart_zoo* и *My_pickle*. Этот класс:

- *наследует:*
  - *из класса Smart_zoo – метод talk();*
  - *из класса My_pickle – методы put() и get();*
- *перегружает метод tell() класса Smart_zoo.*

```
>>> class New_zoo (Smart_zoo, My_pickle):
        def __init__ (self, name, mood, where, file):
            print (where, 'Появился зверек.' )
            self.name=name
            self.file=file
        def tell (self,obj):
            print ('А ты кто? ')
            obj.talk()
            print (obj.name+ ", я сохраню твои данные." )
            self.put(obj)
```

Для проверки создадим один объект класса Smart_zoo:

```
>>> s1=Smart_zoo('Золотая рыбка', 'чудесно', 'Море.')
Море. Появился зверек.
```

и один объект класса New_zoo:

```
>>> new1=New_zoo(' 'Колобок', 'хорошо.', 'Берег моря.', 'obj.dat' )
Берег моря. Появился зверек.
>>> new1.talk()
Я – Колобок.
>>> new1.tell(s1)
```

```
А ты кто?

Я – Золотая рыбка.
Золотая рыбка, я сохраню твои данные.

>>> s2=new1.get()
>>> s2.name
Золотая рыбка
```

Наследование встроенных классов

Сначала рассмотрим наследование встроенных неизменяемых классов. Для этого создадим класс, который наследует класс *float* и осуществляет вычисление обратного значения заданной величины, используя магический метод __rtruediv__(self, other):

```
>>> class Rev (float):
        '''Получение обратной величины'''
        def __init__ (self,arg):
            arg=arg.__rtruediv__(1)
>>> Rev(7)
7.0
```

Результат показывает, что было возвращено исходное значение заданного аргумента (7.0) без вычисления его обратного значения. Это произошло потому, что при инстанцировании класса *Rev()* был получен экземпляр объекта неизменяемого типа и поэтому его значение не могло быть изменено при инициализации.

Решением является перенос процесса вычисления обратной величины туда, где новый объект еще не создан. А это может быть только блок (тело) специального метода *__new__()*, который вызывается самым первым при инстанцировании класса:

```
>>> class Rev (float):
        '''Получение обратной величины'''
        def __new__ (cls,arg):
            arg=arg.__rtruediv__(1)
            return float .__new__(cls,arg)
>>> Rev(7)
0.14285714285714285
```

Рассмотрим еще один пример наследования неизменяемого класса. Создадим класс *W*, который наследует встроенный неизменяемый класс *str* и переопределяет в нем операции сравнения с помощью специальных методов.

Вместо того, чтобы сравнивать слова между собой по величине кода их символов, как это делается в стандартных строках, объекты класса *W*, которыми являются слова, будут сравниваться по их длине, т.е. по числу символов:

```
>>> class W ( str ):
```

```
'''Сравнение слов по их длине'''
def __new__ (cls,word):
        if ' ' in word:
            word=word[:word.index( ' ' )]
        return str __new__(cls,word)
def __gt__ (self,other):
        return len (self)> len (other)
def __ge__ (self,other):
        return len (self)>= len (other)
def __le__ (self,other):
        return len (self)<= len (other)
def __lt__ (self,other):
        return len (self)< len (other)
```

Создадим путем инстанцирования объекты *a*, *b* класса *str* и объекты aw, bw класса *W*:

```
>>> a= 'cat'
>>> aw=W( 'cat' )
>>> b= 'z'
>>> bw=W( 'z' )
```

и проверим операции сравнения объекта *a* с объектом *b* и объекта *aw* с объектом *bw*:

```
>>> a>b                                                    b
False
>>> aw>bw
True
>>> a<b
True
>>> aw<bw
False
>>> a==b
False
>>> aw==bw
False
```

Результаты показывают, что сравнение строк *a* и *b* осуществляется по величине кода их первых символов, а сравнение строк *aw* и *bw* – по их длине.

Отметим, что сравнение строк *aw* и *bw* на равенство (операция «==») осуществлялось с использованием метода *__eq__ (self,other)* базового класса *str*, поскольку для класса *W* этот метод не был определен.

Создание пользовательских последовательностей

В качестве примера создания пользовательской последовательности рассмотрим класс *My_list*, функционально близкий к списку, но имеющий ряд методов, которых нет в классе *list*, и в то же время не поддерживающий некоторые методы списка:

```
>>> class My_list :
        '''Пользовательский класс, близкий к списку '''
        def __init__ (self,values= None ) :
                if values is None :
                        self.values=[]
                else :
                        self.values=values
        def __getitem__ (self,key):
                ''' Получить значение по ключу '''
                return self.values[key]
        def __setitem__ (self,key,value):
                ''' Установить значение по ключу '''
                self.values[key]=value
        def first (self):
                ''' Получить значение первого элемента '''
                return self.values[0]
        def last (self):
                ''' Получить значение последнего элемента '''
                return self.values[-1]
        def __iter__ (self):
                ''' Сделать последовательность итерабельной '''
                return iter (self.values)
        def __str__ (self):
                ''' Возвратить строковое представление объекта '''
                return 'My_list: ' str (self.values)
        def __len__ (self):
                ''' Возвратить длину последовательности '''
                return len (self.values)
        def __delitem__ (self,key):
                ''' Удалить элемент '''
                del self.values[key]
        def push (self, value):
                ''' Добавить элемент в конец последовательности'''
                self.values[len (self): len (self)] = value
```

Класс *My_list* обладает следующими функциональными возможностями, реализуемых его методами:

- *осуществлять инстанцирование объектов и их инициализацию с помощью методов __new__() базового класса object и __init__():*

```
>>> m1=My_list()
>>> m2=My_list([1,2,3,4,5,6,7]);
```

- *выводить на экран с помощью метода __str__() строковое представление созданных "экземпляров класса:*

```
>>> print (m1)
My_list: []
>>> print (m2)
My_list: [1, 2, 3, 4, 5, 6, 7];
```

- *получать значение элемента последовательности с помощью метода __getitem__() по его ключу (индексу):*

```
>>> m2[3]
4;
```

- *изменять с помощью метода __setitem__() значение элемента последовательности:*

```
>>> m2[3]=77
>>> m2[3]
77;
```

- *получать с помощью метода first() значение первого элемента последовательности:*

```
>>> m2.first()
1;
```

- *получать с помощью метода last() значение последнего элемента последовательности:*

```
>>> m2.last()
7;
```

- *добавить с помощью метода push() элемент в конец последовательности:*

```
>>> m2.push( 'A' )
>>> print (m2)
My_list: [1, 2, 3, 77, 5, 6, 7, 'A'];
```

- *определить с помощью метода __len__() число элементов последовательности:*

```
>>> len (m2)
8;
```

- *сделать с помощью метода __iter__() последовательность итерабельной, что позволяет использовать оператор for in:*

```
>>> for i in m2:
print (i, end= ' ' )
1, 2, 3, 77, 5, 6, 7, A;
```

- *удалить с помощью метода __del__() элемент последовательности:*

```
>>> del m2[5]
>>> print (m2)
My_list: [1, 2, 3, 77, 5, 7, 'A'];
```

- *вернуть с помощью метода __pop__() значение последнего элемента последовательности и удалить этот элемент:*

```
>>> m2.pop()
'A';
>>> print (m2)
My_list: [1, 2, 3, 77, 5, 7];
```

3.3 Задания на выполнение лабораторной работы

Разработать программу на языке Python, в которой необходимо:

1. создать класс, наследующий родительский (базовый) класс, в качестве которого использовать класс, разработанный в лаб. раб. №6 (этот класс описывает, например, поведение персонажей). При этом наследуемый класс должен иметь:
   - расширенный (по отношению к родительскому классу) конструктор (метод __init__());
   - перегружаемый метод;
   - новый метод (или методы);
2. создать второй класс (по усмотрению студента);
3. создать третий класс путем множественного наследования первого и второго классов (этот класс должен содержать методы как первого, так и второго классов);
4. создать класс, который наследует неизменяемый класс float и с помощью специального метода __new__() преобразует значения длины и веса, заданные в одной системе единиц измерения, в другую согласно колонке "Преобразовать" табл. 1:
   - 1 – из дюймов в миллиметры;
   - 2 – из сантиметров в дюймы;
   - 3 – из футов в метры;
   - 4 – из метров в футы;
   - 5 – из фунтов в килограммы;
   - 6 – из граммов в караты;
   - 7 – из килограммов в унции;
5. создать класс в виде последовательности, который должен иметь методы, номера которых указаны в колонке "Методы" табл. 1:
   - 1 – __init__();
   - 2 – __getitem__();

- 3 – __setitem__();
- 4 – first();
- 5 – last();
- 6 – __iter__();
- 7 – __str__();
- 8 – __len__();
- 9 – push();
- 10 – pop ().

Таблица 1 – Перечень индивидуальных заданий

Номер п/п	**Преобразование**	**Методы**
1	1	1,2,3,4,6,7
2	2	1,2,3,5,7,8
3	3	1,2,3,6,8,9
4	4	1,2,3,8,9,10
5	5	1,2,4,5,7,10
6	6	1,2,3,4,6,7
7	7	1,2,3,5,7,8
8	1	1,2,3,6,8,9
9	2	1,2,3,8,9,10
10	3	1,2,4,5,7,10
11	4	1,2,3,4,6,7
12	5	1,2,3,5,7,8
13	6	1,2,3,6,8,9
14	7	1,2,3,8,9,10
15	1	1,2,4,5,7,10
16	2	1,2,3,4,6,7
17	3	1,2,3,5,7,8
18	4	1,2,3,6,8,9
19	5	1,2,3,8,9,10
20	6	1,2,4,5,7,10

4. СПИСОК ЛИТЕРАТУРЫ

1. Лутц М. Изучаем Python. Том 1 – Москва: "Диалектика", 2020.
2. Лутц М. Изучаем Python. Том 2 – Москва: "Диалектика", 2020.

СОДЕРЖАНИЕ

1 Лабораторная работа № 5
Разработка функций и модулей пользователя. 3
1.1 Цель лабораторной работы 3
1.2 Теоретические сведения 3
1.3 Задания на выполнение лабораторной работы 20
2 Лабораторная работа № 6
Объектно-ориентированное программирование на языке Python.
Создание пользовательских классов. 22
2.1 Цель лабораторной работы 22
2.2 Теоретические сведения 23
2.3 Задания на выполнение лабораторной работы 37
3 Лабораторная работа № 7
Наследование классов. 38
3.1. Цель лабораторной работы 38
3.2. Теоретические сведения 38
3.3. Задание на выполнение лабораторной работы 46
4 СПИСОК ЛИТЕРАТУРЫ 47