ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

(Python Beginning)

«Лучший способ объяснить – это самому сделать!»
(Из сказки Льюиса Кэрролла «Приключения Алисы в стране чудес»)

Урок 8. (Lesson 8) Основы программирования

ОБЪЕКТНО_ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Реализация концепции ООП, выполненная в *Python*, лишь поверхностно напоминает ООП в *C++*. В большей мере это связано механизмами создания объектов *PVM*. Лишь при учитывания факта влияния *PVM* на время жизни объектов, можно понять те «***навороты***», привнесенные в этот язык. Итак, милостивые государи и государыни, прошу чуть-чуть отойти привычного миропонимания и погрузиться в «***пучину вод***» *Python* – ***немножечко сойти с ума***, чтобы ***перейти на другой уровень понимания*** вещей…

«-На что мне безумцы? – сказала Алиса.
- Ничего не поделаешь, - возразил Кот. – Все мы не в своем уме – и ты, и я!
-Откуда вы знаете, что я не в своем уме? – спросила Алиса.
- Конечно, не в своем, - ответил Кот. – Иначе как бы ты здесь оказалась?»
(Из сказки Льюиса Кэрролла «Приключения Алисы в стране чудес»)

Определение класса

Напомню, что, равно как в *UNIX*, все является файлом, также и в *Python* ***все является объектом*** - экземпляром класса, производного от суперкласса «*object*». Типы данных, файлы, числовые данные и все остальное реализованы через механизмы ООП, функции – суть методы классов. Атомарных типов данных, не связанных с ООП в Python нет! Даже ваша программа – это тоже экземпляр класса. А *PVM* реализует в своем окружении объект – экземпляр определенного вами класса.

Класс объявляется и определяется по следующей схеме:

```
class <Имя_класса> [(<Имя_суперкласса1>[, … ,<Имя_суперклассаN>])]:
    ["""Строка встроенной документации"""]
    <Объявление атрибутов и методов класса>
    [def <Имя_метода>(self,[<Параметры_метода>]):]
        ["""Встроенная документация на атрибуты/методы класса"""]
```

Ключевое слово *class* определяет для интерпретатора начало блока определения. *<Имя класса>* задаётся по правилам именования идентификаторов ссылок. *<Имя класса>* задает ссылку на ***объект класса***. После *<Имени класса>* в круглых скобках указываются имена базовых класса, от которых производится

данный класс. Порядок указания суперклассов имеет значение при реализации множественного наследования в плане выбора метода, определенного в нескольких суперклассах. Имена методов суперклассов могут совпадать, а ключевого слова *virtual*, как в *C++*, в *Python* нет…

Обратите внимание на возможность внедрения в объявляемый класс сопроводительной документации. Тип данных (класс) имеет обязательный атрибут *__doc__*, он позволяет определять встроенную документацию ко всем классам и всем атрибута классов, включая методы класса. Если программист не определит значение данного атрибута, то он связан с объектом *None*.

Рассмотрим следующий пример объявления класса.

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_01.py
Abstract: Пример экземпляра объекта типа класс
"""
class AnyClass:
    """Простой класс для демонстрации основ ООП в Python 3"""
    print("Мы объявили класс AnyClass")
# END of AnyClass
print("Прикольно, да? А ведь мы просто объявили класс.")
# END of test_class_01.py
```

Рассмотрим результат работы данного кода.

```
Мы объявили класс AnyClass
Прикольно, да? А ведь мы просто объявили класс.
>>>
```

Выражения были выполнены при создании объекта класса, а не при создании объекта экземпляра класса. Данный факт можно использовать для выполнения инструкций, связанных с подготовкой системного окружения, необходимого для работы класса, выполняемых до вызова конструктора, то есть до создания экземпляров класса.

```
class AnyClass:
    """Класс с 'изюминкой'."""
    # Данный блок операций связан с созданием экземпляра класса
    # и определением атрибут класса.
    # Здесь могут выполняться действия, связанные с подготовкой
    # ресурсов, настройкой системного окружения и много другое.
# END of AnyClass
```

Объявление класса в *C++* связано с работой компилятора, направленной на формирование «***дескриптора***» класса. Язык программирования *Python* при объявлении класса позволяет делать гораздо больше. Этот набор действий позволяет заменить использование директив условной компиляции, которые были

доступны в *C/C++*, связанный с настройкой приложения под особенности программно-аппаратной платформы.

В блоке, связанном с «настройкой класса» могут быть выполнены любые необходимые вам действия, например:

```
# import subprocess  обеспечение возможности вызова подпрограмм
class AnyClass1:
    """Класс с 'изюминкой'."""
    import subprocess
    cmd = "program.exe -server"
    subprocess.Popen(cmd, shell=True)
    # здесь может быть все, что угодно. И оно будет выполнено
    # на "фоне" работы вышего приложения
# END of AnyClass

class AnyClass2:
    from tkinter import Tk, Label
    root = Tk()
    label = Label(root, text = "Hello", font = ("arial", 20))
    label.pack()
    root.mainloop()
# END of AnyClass
```

Создание переменной внутри класса выполняется так же, как и создание обычной переменной с той лишь разницей, что оно выполняется внутри тела класса (извините за использование терминологии *C++*). Но, в отличие от *C++*, в *Python* выделяются переменные класса и переменные экземпляра класса. *Python* обеспечивает возможность динамически, по ходу выполнения программы, изменять свойства класса: создавать новые, удалять старые атрибуты. Это приводит к тому, что внутри вашей программы может быть несколько экземпляров одного класса, которым доступны совершенно различные функциональные интерфейсы и разные наборы атрибут.

Такова лирика программирования на языке Python, у которого «свой подход» к определению времени жизни программных объектов…

Создание объекта класса происходит при определении класса, а создание экземпляра класса выполняется посредством вызова конструктора. Имя конструктора, используемое при вызове, не совпадает с именем класса – внутри тела класса конструктор имеет имя *__init__*. Кто помнит имена процессов в *UNIX/Linux* – системах, сразу поймет, «***откуда ноги растут***».

Если вы еще «***не забыли великий и могучий***» *C++*, то поймете, почему разработчики *Python* отказались от всего того многообразия конструкторов: конструктор по умолчанию, перегруженный конструктор по умолчанию, перегруженный конструктор, конструктор копирования, конструктор преобразования, да ещё возможность управления запрета/разрешения явного и

неявного вызова конструктора посредством использования ключевого слова explicit. Внешне программирование на *Python* проще, чем на *C++*. *Чем проще инструмент, тем меньше вероятность допустить ошибку при его использовании...* Но это лишь видимая простота, за которую приходится расплачиваться потерей возможности контроля над создаваемым кодом.

Рассмотрим синтаксис создания объекта – экземпляра класса.

```
<Имя_ссылки_на_экземпляра_класса> = <Имя_класса>([Параметры])
```

Рассмотрим простой класс (SimpleClass) и использующий его код.

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_02.py
Abstract: Демонстрация возможносте ООП
"""
class SimpleClass:
    """Простой класс для демонстрации возможностей ООП"""
    count = 0  # переменная класса, счетчик экземпляров класса
    def __init__(self, x=0):
        """Конструктор класса SimpleCalss"""
        self.x = x
        SimpleClass.count += 1 # увеличить счетчик экземпляров класса
    def print_x(self):
        print(self.x)
    def get_x(self):
        return self.x
    def set_x(self, new_x):
        self.x = new_x
    def get_count(self):
        return SimpleClass.count
    def __del__(self):
        # уменьшить значение счетчика экземпляров класса
        SimpleClass.count -= 1
    # END of class
# --------------------------------------------------------------------
obj1 = SimpleClass()
print("Количество экземпляров класса =", obj1.get_count())
obj2 = SimpleClass(21)
print("Количество экземпляров класса =", obj2.get_count())
print("obj1.x =",obj1.get_x())
print("obj2.x =",obj2.get_x())
obj1.set_x(12)  # изменить атрибут через вызов метода
obj2.x = 4      # изменить атрибут объекта напрямую, по имени
print("Внесены изменения")
print("obj1.x =",obj1.get_x())
print("obj2.x =",obj2.get_x())
del obj1  # удалить ссылку на объект
print("Количество экземпляров класса =", obj2.get_count())
del obj2  # удалить ссылку на объект
```

В рассмотренном примере есть некоторый недостаток, связанный с подсчетом количества экземпляров класса, но подробно о нём мы поговорим чуть позже.

Рассмотрим ещё один пример, связанный с использованием атрибут класса. Он должен помочь вам по другому взглянуть на вопросы информационной безопасности: используя в своем проекте чужой код, даже проверенный при помощи антивируса, которому вы доверяете, вы запускаете «чужой код» со своими «личными правами». В данном примере всего лишь имитируется «фоновая активность» чужой библиотеки.

```
class EasyClass:
    """Такой вот тихоня =)"""
    import sys
    from tkinter import Toplevel, Button, Label
    for i in range(1, 11):
        frm = Toplevel()
        Button(frm, text="Hello", font=("arial", 20),
               command=sys.exit).pack()
    Label(text="Hello World! Hello Dummy!",
          font=("arial", 32)).pack()
    # END конец блока атрибут класса...
# END of class EasyClass
```

Помните, используя «чужие инструменты», такие как библиотеки, полученные из «свободных источников», вы целиком и полностью берете на себя ответственность за возможное причинение ущерба оными в создаваемых вами программных системах!

Это сейчас вы можете выявить часть кода реализации класса, отвечающего за «***паразитный теневой процесс***», в реальности, модули и библиотеки поставляются в виде предварительно откомпилированного кода, то есть в виде бинарных файлов, и получить доступ к исходному коду нет никакой возможности. Данное утверждение касается не только *Python*-модулей, но и *Java*, *C#*, *C++* модулей и библиотек.

Однако, имеется ряд ограничений на включение информации из других модулей в состав атрибут класса – вы должны четко указывать перечень всех необходимых компонент, импортируемых из модуля. Инструкции, вроде:

```
from <Имя_модуля> import *
```

в пространстве объявления атрибут класса, приведут к ошибке трансляции.

Существует в *Python* и возможность для объявления класса без определения его структуры. Для этого, как и для определения «пустых» методов, используется ключевое слово *pass*.

```
>>> class AnyClass:
     pass
```

```
>>> A = AnyClass()
>>> type(A)
<class '__main__.AnyClass'>
>>>
```

Ранее мы уже разбирали примеры, связанные с использованием библиотеки/модуля *tkinter*, и встречали обязательный атрибут *__main__*, связанный с именем главного модуля проекта.

```
if __name__ == '__main__':
```

Данный атрибут не используется для хранения информации об имени класса, но через него мы получаем доступ к классу, связанным с нашим модулем, который, как уже было сказано ранее, тоже является объектом класса.

Мы получили уведомление, что объект *A* является атрибутом главного модуля и является экземпляром класса *AnyClass*.

Методы класса определяются внутри класса так же, как и обычные функции, с использованием ключевого слова *def*. Методам класса передается ссылка на экземпляр класса – *self*. Роль данной ссылки совпадает с ролью указателя *this* в *C++*. Использование *self* в качестве первого обязательного параметра метода класса считается признаком «***хорошего тона***», если его опустить, то доступ к атрибутам класса внутри метода становится невозможен.

Доступ к атрибутам и методам класса осуществляется через точечную нотацию. Формат вызова метода:

```
<Ссылка_на_экземпляр_класса>.<Имя_метода_класса>([Параметры])
```

Формат обращения к атрибутам класса:

```
<Ссылка_на_экземпляр_класса>.<Имя_атрибута_класса>
```

По умолчанию атрибуты класса являются открытыми, Надежных механизмов сокрытия данных в *Python* практически нет.

Рассмотрим пример создания класса для представления точек на плоскости в Декартовой системе координат. Атрибуты класса должны позволять хранить значения координаты точки и, возможно, её имя.

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_02.py
Abstract: Демонстрация возможносте ООП
"""
class Point2D:
    """Простой класс для демонтсрации возможностей ООП"""
    count = 0  # счетчик экземпляров класса
    def __init__(self, name="", x=0, y=0):
        """Конструктор класса AnyCalss"""
        self.name = name
```

```
        self.x = x
        self.y = y
        count += 1  # увеличить счетчик экземпляров класса
    def print_point(self):
        full_name = self.name + "(" + str(self.x) + ","
                    + str(self.y) + ")"
        print(full_name)
    def get_x(self):
        return selx.x
    def get_y(self):
        return self.y
    def get_xy(self):
        return tuple(x,y)
    def get_name(self):
        return self.name
    def set_x(self, new_x):
        self.x = new_x
    def set_y(self, new_y):
        self.y = new_y
    # END of class
```

Атрибуты объекта класса и экземпляра класса

При объявлении класса интерпретатор *Python* создает объект класса, а при вызове конструктора – экземпляр класса. Возможно, данный факт и послужил базой того, что выделяются атрибуты объекта класса и атрибуты экземпляра класса. Атрибуты класса «***внешне похожи***» на статические атрибуты класса в *C++*, но, практически, ни как с ними не связаны. Это обусловлено тем, что в ряде случаев атрибуты объекта класса могут становиться атрибутами объекта класса. Сама структура класса в процессе работы программы может претерпевать кардинальные изменения: создаваться новые и удаляться старые атрибуты, переопределяться существующие методы и, конечно же, добавляться новые методы. *Python* –язык динамической типизации!

Рассмотрим следующий пример кода, использующего как атрибуты класса, так и атрибуты экземпляра класса.

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_04.py
Abstract: Демонстрация возможносте ООП
"""
class AnyClass:
    """Простой класс для демонстрации возможностей ООП"""
    count = 0  # атрибут класса
    def __init__(self, x=0):
        """Конструктор класса SimpleCalss"""
        self.x = x  # атрибут экземпляра класса
    def print_x(self):
```

```
        print(self.x)
    def get_x(self):
        return self.x
    def set_x(self, new_x):
        self.x = new_x
    # END of class
# ------------------------------------------------------------------
obj_A = AnyClass()
print("obj_A.x =", obj_A.x)
print("AnyClass.count =", AnyClass.count)
AnyClass.y = 10  # определение нового атрибута класса
obj_B = AnyClass(13)

# определим внешнюю функцию. func - это ссылка на функцию
def func():
    print("Hello Dummy")

AnyClass.meth = func  # определили в интерфейсе класса новый атрибут.
# Данный атрибут является ссылкой на функцию. Это новый метод класса.
# Данный метод можно вызвать только через имя класса, попытка вызова
# через ссылку на экземпляр класса приведет к выбросу исключения.
# Попробуйте, раскомментируйте следующую строку кода...
# obj_B.meth()

obj_C = AnyClass(4)
print(type(obj_C.meth))
AnyClass.meth()  # вызов метода через объект класса
```

Возможно, данный пример для вас слишком «громоздкий», поэтому рассмотрим более простой пример, который можно выполнить в режиме пошаговой отладки.

```
>>> class AnyClass:
     """Пустое определение класса"""
     pass

>>> AnyClass.var1 = 1   # определяем отрибут объекта класса
>>> obj_1, obj_2 = AnyClass(), AnyClass()
>>> obj_1.var2 = 2
>>> obj_2.var2 = 3
>>> print(obj_1.var1, obj_1.var2)
1 2
>>> print(obj_2.var1, obj_2.var2)
1 3
>>>
```

Нет ничего обидного в том, что выполнен некоторый «откат» по уровню сложности кода примеров. Это сделано с учетом того, что концепция ООП в Python

существенно отличается от концепции ООП, реализованной в C++. С C++ мы уже хорошо знакомы, а Python ещё только-только изучаем ☺.

Как видно из приведенного примера, можно задавать новые атрибуты не только для класса, но и для экземпляра класса. Посредство операции *is* можно проверить механизмы использования памяти и показать, что даже в случае создания в разных экземплярах класса атрибут, носящие одинаковое имя – имена ссылок совпадают, они будут связаны с разными объектами в памяти.

```
>>> obj_1.var1 is obj_2.var1  # проверка атрибута объекта класса
True
>>> obj_1.var2 is obj_2.var2  # проверка атрибута экземпляра класса
False
>>>
```

Атрибуты класса можно удалять, равно как и обычные программные объекты, через ссылки на них. Но атрибут класса нельзя удалить через ссылку на экземпляр класса.

```
>>> del obj_1.var1
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    del obj_1.var1
AttributeError: var1
>>> del obj_1.var2
>>>
```

Для выполнения манипуляций с атрибутами класса в Python определены следующие методы: *getattr()*, *setattr()*, *delattr()*, *hasattr()*.

Опять же воспользуемся простым по структуре классом и на его примере изучим правила работы с данными методами.

```
class AnyClass:
    """Простой класс для демонстрации возможностей ООП в Python"""
    var1 = 10
    var2 = "AnyClass"
    def __init__(self, var=0):
        self.var2=var
    def func(self):
        print("Var2 = %s" % self.var2)
# END of AnyClass
```

- Метод *getattr()* возвращает значение атрибута по его имени-ссылке, заданному в виде строки. Формат вызова метода:

getattr(<Объект>, <Атрибут> [, <Значение_по_умолчанию>])

Если указанный атрибут является методом класса, то возвращается ссылка на метод. В случае, если указанный атрибут не найден, – в данном классе его нет,

выбрасывается объект исключения *AttributeError*. Чтобы избежать возбуждения исключения, необходимо задать значение атрибута по умолчанию.

```
obj = AnyClass(17)
print("AnyClass.var1 =", getattr(AnyClass, "var1"))
print("obj.var1 =", getattr(obj, "var1"))
print("obj.var1 =", getattr(obj, "var2"))
```

Обратите внимание, что атрибут передается методу в виде строки. Попытка передачи атрибута не в виде строки приведет к тому, что будет сгенерировано исключение.

```
print("obj.var1 =", getattr(obj, var2))
```

```
Traceback (most recent call last):
  File "C:/Python35/Examples_29_06/test_class_06.py", line 14, in
<module>
    print("obj.var1 =", getattr(obj, var2))
NameError: name 'var2' is not defined
>>>
```

Аналогичным образом можно получать доступ к ссылкам на методы класса, а затем и вызывать их.

```
method = getattr(obj, "func")
method()
```

- Метод *setattr()* задает значение указанного атрибута. Формат вызова метода:

setattr(<Объект>, <Атрибут>, <Значение>])

```
obj.var3 = "Hello"
print(getattr(obj, "var3"))
setattr(obj, "var3", "New String!")
obj.var4 = list()
getattr(obj,"var4").append("A")
print(getattr(obj, "var4")[0])
# setattr(obj, "var4", "B") атрибут var4 ссылочного типа!
```

Установить новое значение для атрибута, связанного с объектом ссылочного типа через вызов метода *setattr()* можно, но это приведет к потере объекта ссылочного типа, связанного с данным атрибутом! Для изменения самого объекта ссылочного типа, связанного с атрибутом, нужно использовать доступ к ссылке на атрибут, который мы получаем посредством вызова *getattr()*.

```
getattr(obj, "var4")[0] = "B"
print(getattr(obj, "var4")[0])
```

Метод *delattr()* удаляет указанный атрибут. Формат вызова метода:

delattr(<Объект>, <Атрибут>)

```
delattr(obj, "var4")
# Атрибут var4 удален.
# Выполним обращение к не существующему атрибуту
print(getattr(obj, "var4")[0])

Traceback (most recent call last):
  File "C:/Python35/Examples_29_06/test_class_06.py", line 27, in
<module>
    print(getattr(obj, "var4")[0])
AttributeError: 'AnyClass' object has no attribute 'var4'
>>>
```

Метод *hasattr()* выполняет проверку наличия указанного атрибута. Если указанный атрибут существует, возвращается значение *True*. Данный метод непосредственно обращается к дескриптору класса и дескриптору экземпляра класса.

```
print(hasattr(obj, "var4"))
print(hasattr(obj, "var3"))
print(hasattr(AnyClass, "var3"))
print(hasattr(obj, "var2"))
print(hasattr(AnyClass, "var2"))
print(AnyClass.var2 is obj.var2)
```

Python предоставляет программисту богатые возможности для изменения класса и его экземпляров уже во время работы приложения. Фактически становится возможной ситуация, когда в памяти программы могут находиться несколько экземпляров класса, реализующие совершенно различные интерфейсы. Данный механизм обеспечивает возможность «***горячей замены***» кода класса.

Определение конструктора и деструктора класса

Как вы уже знаете из курса «Программирование», при создании объекта вызывается специальный метод класса, называемый конструктором, а при его удалении – деструктор. Конструктор класса имеет имя *__init__*. Формат определения метода-конструктора:

```
def __init__(self [,<Параметр_1=Значение_по_умолчанию_1> [, … ,
            <Параметр_N=Значение_по_умолчанию_N>]]):
    ["""<Документация_метода>"""]
    <Инструкции_метода>
```

Формат вызова конструктора:

```
<Экземпляр_класса> = <Имя_класса> ([<Значение1> [, … <ЗначениеN>]])
```

На деструктор возлагаются обязанности по освобождению ресурсов, использованных объектом – экземпляром класса. Деструктор имеет имя *__del__*. Деструктор может принимать единственный параметр – ссылку на экземпляр

класса *self*. Это связано с тем, что *Python* не предоставляет возможности для явного вызова деструктора класса. Деструктор вызывается в случае, когда объект выходит за пределы области видимости или, если мы вызываем встроенный метод *del*. Формат определения деструктора:

```
def __del__(self):
    ["""<Документация_метода>"""]
    <Инструкции_метода>
```

Учтите, что «***освобождение ресурсов***», связанных с объектами – это ваша «святая обязанность» перед пользователями кода. Объекты, с которыми приходится работать, могут быть связаны с файловыми потоками, кортами ввода-вывода, с сокетами и многими другими сложными ресурсами, требующими выполнения процедуры «***разъединения связи***»… Конечно, если вы не определите конструктор и деструктор класса, за вас это сделает транслятор языка *Python*, но он сделает ли он это так, как надо?

Кроме того, необходимо помнить, что деструктор класса вызывается только в том случае, когда на объект не указывает ни одной ссылки, в встроенный метод *del* удаляет именно ссылку!

Рассмотрим следующий пример.

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_07.py
Abstract: Определение констркторов и деструкторов класса
"""
class AnyClass:
    """Простой класс для демонстрации основ ООП в Python 3"""
    print("Создан объект класса AnyClass")
    def __init__(self):
        """Констркор класса AnyClass"""
        print("Call: AnyClass.__init__()")
    def __del__(self):
        """Деструктор класса AnyClass"""
        print("Call: AnyClass.__del__()")
# END of AnyClass
obj1 = AnyClass()
arr = list()
arr.append(obj1)  # в список скопирована ссылка на объект
del obj1  # удаляется ссылка, а не объект
del arr   # удаляется контейнер, содержащий ссылку на объект
obj2 = AnyClass()  # создать ссылку на объект в памяти
obj3 = obj2
del obj2  # удалить ссылку
del obj3  # удалить ссылку и вызвать деструктор
# END of test_class_07.py
```

Обратите внимание на то, что ссылка связана с объектом, а не наоборот. Можно создать объект, не связанный ни с одной ссылкой, но сразу после создания для него будет вызван деструктор.

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_08.py
Abstract: Определение констркторов и деструкторов класса
"""
class AnyClass:
    """Простой класс для демонстрации основ ООП в Python 3"""
    print("Создан объект класса AnyClass")
    def __init__(self):
        """Констрктор класса AnyClass"""
        print("Call: AnyClass.__init__()")
    def __del__(self):
        """Деструктор класса AnyClass"""
        print("Call: AnyClass.__del__()")
# END of AnyClass
AnyClass()  # Создан анонимный объект
# END of test_class_08.py
```

Рассмотрим следующие примеры.

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_09.py
"""
class Point2D:
    count = 0  # счетчик экземпляров класса Person
    """Конструктор класса Person"""
    def __init__(self, x=0, y=0):
        """Констрктор класса Point2D"""
        self.x = x
        self.y = y
        Point2D.count += 1  # увеличиваем счетчик
    def __del__(self):
        """Деструктор класса Person"""
        Point2D.count -= 1  # уменьшаем счетчик
    def __str__(self):
        str1 = '('+ str(self.x) + ',' + str(self.y) + ')'
        return str1
# END of AnyClass
if __name__ == '__main__':
    A = Point2D(12,18)
    B = Point2D(-4,-10)
    print("A",A)
    print("B",B)
# END of test_class_09.py
```

```
# -*- coding: utf-8 -*-
"""
Filename: test_class_10.py
Abstract: Класс для представления информации о сотрудниках
"""
class Person:
    count = 0  # счетчик экземпляров класса Person
    """Конструктор класса Person"""
    def __init__(self, name, age, pay, job=None):
        """Констрктор класса Person"""
        self.name = name
        self.age = age
        self.pay = pay
        self.job = job
        Person.count += 1  # увеличиваем счетчик
    def __del__(self):
        """Деструктор класса Person"""
        Person.count -= 1  # уменьшаем счетчик
    def lastName(self):
        return self.name.split()[-1]
    def __str__(self):
        str1 = str(self.name) + ' : ' + str(self.job)
        return str1
# END of AnyClass
if __name__ == '__main__':
    alex = Person('Alex Trofomov', 46, 45000, 'top manager')
    ivan = Person('Ivan Petrov', 42, 30000, 'softvare')
    oleg = Person('Oleg Kochnev', 45, 40000, 'hardvare')
    print("Person.count =", Person.count)
    print("Boss is", alex.lastName())
    print(alex)
# END of test_class_10.py
```

Наследование

Механизмы наследования, реализованный в *Python*, значительно проще механизмов наследования, реализованных в *C++*. Не удивительно, ведь разработчики языка *Python* преследовали цель упрощения языка по сравнению с *C++*, который был использован в качестве «***прототипа***».

Как вы уже заметили, в Python нет модификаторов доступа *public*, *private* и *protected*, нет и модификатора *explicit*. Фактически, в *Python* реализованы только публичные интерфейсы: атрибуты и методы классов, и публичное наследование.

Напоминаю, наследование – это механизм ООП, позволяющий определять новые классы, производные от некоторых базовых классов. Производный класс «***расширяет***» интерфейс родительского класса, хотя и имеет право быть для него

всего лишь «***внешней оболочной***» (*волком в овечьей шкуре*). Родительский класс принято называть суперклассом.

Формат определения наследования при объявлении производного класса:

```
def <Имя_класса> (<Имя_суперкласса_1> [, … , <Имя_суперкласса_N>]):
    <Определение_внутренней_структуры_класса>
```

Рассмотрим следующие примеры.

```
# -*- coding: utf-8 -*-
"""Filename: test_class_11.py"""
class A:
    def func1(self):
        print("Method of class A: A.func1()")
# END of class A
class B(A):
    def func2(self):
        print("Method of class B: B.func2()")
# END of class B
if __name__ == '__main__':
    obj_1 = A()
    obj_2 = B()
    obj_1.func1()
    obj_2.func1()
    obj_2.func2()
# END of test_class_11.py
```

Производный класс расширяет интерфейс суперкласса.

```
# -*- coding: utf-8 -*-
"""Filename: test_class_12.py"""
class A:
    def func(self):
        print("Method of class A: A.func()")
# END of class A
class B(A):
    def func(self):
        print("Method of class B: B.func()")
# END of class B
if __name__ == '__main__':
    obj_1 = A()
    obj_2 = B()
    obj_1.func()
    obj_2.func()
# END of test_class_12.py
```

```
# -*- coding: utf-8 -*-
"""Filename: test_class_13.py"""
class A:
    def func(self):
        print("Method of class A: A.func()")
```

```
# END of class A
class B(A):
    def func(self):
        # Переопределение метода базового класса
        print("Method of class B: B.func()")
        A.func(self)  # вызов метода базового класса
# END of class B
if __name__ == '__main__':
    obj_1 = A()
    obj_2 = B()
    obj_1.func()
    obj_2.func()
# END of test_class_13.py
```

На данном примере явно прослеживается механизм переопределения метода в производном классе, совпадающем с именем родительского класса.

```
# -*- coding: utf-8 -*-
"""Filename: test_class_14.py"""
class A:
    def __init__(self, var1=0):
        """Констрктор класса A"""
        self.var1 = var1
        print("Call A.__init__()")
    def __str__(self):
        str1 = str(self.var1)
        return str1
    def func_A(self):
        print("var1 =", self.var1)
# END of class A
class B(A):
    def __init__(self, var1=0, var2=0):
        A.__init__(self, var1)
        self.var2 = var2
        print("Call B.__init__()")
    def __str__(self):
        str1 = str(self.var1) + ',' +str(self.var2)
        return str1
    def func_B(self):
        self.func_A()
        print("var2 =", self.var2)
# END of class B
if __name__ == '__main__':
    obj_1 = A(2)
    print("obj_1 :", obj_1)
    obj_1.func_A()
    obj_2 = B(3,4)
    print("obj_2 :", obj_2)
    obj_2.func_A()
    obj_2.func_B()
```

```
# END of test_class_14.py
```

В *Python*, как и в *C++*, равно как и в *Java*, конструктор базового класса автоматически не вызывается, если он переопределен в производном классе.

Для доступа к атрибутам родительского класса из производного класса необходимо использовать встроенную функцию *super()*.

```
# -*- coding: utf-8 -*-
"""Filename: test_class_15.py"""
class A:
    def __init__(self, var=0):
        print("Call constructor of class A")
        self.var = var
    def func1(self):
        print("Method of class A: A.func1()")
        print("A.var =", self.var, "\n==================")
# END of class A
class B(A):
    def __init__(self, var):
        A.__init__(self, var)
        print("Call constructor of class B")
    def func1(self):
        print("Method of class B: B.func1()")
        A.func1(self)  # вызов метода базового класса
# END of class B
if __name__ == '__main__':
    obj_1 = A()
    obj_2 = B(1)
    obj_1.func1()
    obj_2.func1()
# END of test_class_15.py

# -*- coding: utf-8 -*-
"""Filename: test_class_16.py"""
class A:
    def __init__(self, var=0):
        print("Call constructor of class A")
        self.var = var
    def func1(self):
        print("Method of class A: A.func1()")
        print("A.var =", self.var, "\n==================")
# END of class A
class B(A):
    def __init__(self, var=0):
        super().__init__(var)
        # полный формат вызова конструктора базового класса
        # super(A, self).__init__()
        print("Call constructor of class B")
    def func1(self):
```

```
        print("Method of class B: B.func1()")
        super().func1()  # вызов метода базового класса
        # полный формат вызова метода родительского класса
        # super(A, self).func()
# END of class B
if __name__ == '__main__':
    obj_1 = A()
    obj_2 = B(1)
    obj_1.func1()
    obj_2.func1()
# END of test_class_16.py
```

Функция *super()* возвращает ссылку на объект родительского класса (суперкласс), ближайший в «цепочке наследования». Формат функции *super()*:

super([<Имя_производного_класса>, <Ссылка self>])

В выше приведенных примерах использовались обе формы вызова функции *super()*: полная – *super(A, self).func()* и сокращенна – *super().func1()*.

Язык программирования Python поддерживает множественное наследование, но через функцию super() можно получить ссылку только на объект «ближайшего в цепочке наследования» суперкласса. В тоже время, используя механизмы точечной нотации, вы можете получить доступ к интерфейсу уже известного суперкласса, например: `A.__init__(self, var1)`.

Пример использования механизмов наследования.

```
"""Filename: test_class_17.py"""
from tkinter import *
from tkinter.messagebox import askokcancel

class Window(Frame):  # создание класса, производного от Frame
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        lbl = Label(self, text="Simple Window", font=('arial', 20))
        lbl.pack()
        btn = Button(self, text='Quit', command=self.quit)
        btn.pack()
    # END of __init__() method
    def quit(self):
        ans = askokcancel('Very exit', 'Really quit?')
        if ans: Frame.quit(self)
    # END of quit() method
# END of Window class
if __name__ == '__main__': Window().mainloop()
```

Был определен класс *Window*, производный от встроенного класса *Frame*, и использованы методы родительского класса.

Механизмы множественного наследования

В предыдущем разделе мы уже рассмотрели формат объявления класса:

```
def <Имя_класса> (<Имя_суперкласса_1> [, … , <Имя_суперкласса_N>]):
    <Определение_внутренней_структуры_класса>
```

В круглых скобках указывается список суперклассов, от которых производится новый класс. Рассмотрим абстрактный пример, основанный на детском афоризме:

```
class <'Нечто'> (<'Бульдог'>, <'Носорог'>):
   <Опроделяем_новые_атрибуты>
```

Фактически, мы сообщаем интерпретатору, что определяется новый класс, полностью наследующий свойства двух родительских класса и потомок получает полный доступ к свойствам, унаследованным от родителей. В этом ключевое отличие механизмов наследования в *Python* от механизмов наследования в *C++*.

Рассмотрим следующий пример.

```
# -*- coding: utf-8 -*-
"""Filename: test_class_18.py"""
class Parent:  # базовый класс
    def func1(self):
        print("Method of class Parent: Parent.func1()")
# END of class Parent
# ------------------------------------------------------
class Derived1(Parent):
    def func2(self):
        print("Method of class Derived1: Derived1.func2()")
# END of class Derived1
# ------------------------------------------------------
class Derived2(Parent):
    def func1(self):
        print("Method of class Derived2: Derived2.func1()")
    def func2(self):
        print("Method of class Derived2: Derived2.func2()")
    def func3(self):
        print("Method of class Derived2: Derived2.func3()")
    def func4(self):
        print("Method of class Derived2: Derived2.func4()")
# END of class Derived2
# ------------------------------------------------------
class SuperDerived(Derived1, Derived2):
    def func4(self):
        print("Method of class SuperDerived: Derived1.func4()")
# END of class Derived1
# ------------------------------------------------------
if __name__ == '__main__':
    obj = SuperDerived()
    obj.func1()
    obj.func2()
```

```
    obj.func3()
    obj.func4()
# END of test_class_18.py
```

Обратите внимание, каким образом ищется метод при множественном наследовании – выполняется просмотр цепочки наследования и определяется первый встретившийся метод, совпадающий по сигнатуре. Порядок цепочки наследования определяется списком суперкласса, заданный в круглых скобках при определении класса.

Механизмы перегрузки методов и операций

- Метод __call__() используется для обработки вызова экземпляра класса. Если использовать терминологию C++, то речь идет о перегрузке оператора вызова функции operator(), механизме определения "функтора". Данный метод используется при проектировании классов, реализующих обобщенные атогитмы.
- Методе __setitem__(self, <Ключ>, <Значение>) предназначен для присвоения значения объекту, поддерживающему итерационный протокол.
- Метод __getitem__(self, <Ключ>) предназначен для чтения данных объекта, поддерживающему итерационный протокол
- Метод __delitem__(self, <Ключ>) предназначен для удаления элемента в объекте, поддерживающем итерационный протокол.

В языке C++ используется перегружаемая операция operator[], возвращающая значение по ссылке. Она одна предназначена для выполнения той работы, которую выполняют методы __setitem__(), __getitem__(), __delitem__().

- Метод __getattr__(self, <Атрибут>) вызывается при попытке обращения к несуществующему атрибуту класса.
- Метод __getattribute__(self, <Атрибут>) вызывается при обращении к любому атрибуту класса.
- Метод __delattr__(self, <Атрибут>) вызывается при удалении атрибута класса и атрибута экземпляра класса при помощи вызова инструкции del.
- Метод __iter__(self) предназначен для организации поддержки классом итерационного протокола. Объявление в классе данного метода требует также определения в классе метода __next__(), обеспечивающего доступ к элементам при каждой итерации.

Задачи

Для закрепления материала решите несколько простых задач. Рассмотрите варианты решения, основанные на использовании встроенных методов и основанные на использовании процедуры перебора элементов списков. Программа должна обеспечивать процедуру ручного ввода данных списка непосредственно

пользователем программы. По возможности, напишите вариант программы с графическим интерфейсом ☺☺☺. Программный интерфейс должен содержать одно поле для ввода значения или последовательности значений

1. Разработать класс `PiConst`, предназначенный для представления значения константы Пи. Вызов `PiConst()` должен возвращать значение константы. Например: `S1 = 2 * PiClass() * R1` – вычисление значения длины окружности, заданной величиной радиуса `R1`. Обоснуйте выбранное проектное решение, оцените его трудоёмкость.
2. Разработать класс Shape (фигура), предназначенный для организации иерархии наследования – базовый абстрактный класс. Данный класс должен содержать базовый набор методов, позволяющих работать с фигурами, представленными экземплярами производных от Shape классов. В классе Shape должны быть реализованы средства, запрещающие создавать его экземпляры.
3. Разработать класс Point2D для представления точек в декартовой системе координат.
4. Разработать класс Point3D для представления точек в 3-х мерном ортогональном базисе.
5. Разработать класс Point3C для представления точек в сферической системе координат.
6. Разработать класс Polygon (многоугольник), описываемый набором точек в декартовой системе координат.
7. Разработать класс, аналог встроенного класса complex, предназначенный для работы с комплексно сопряженными числами.
8. Разработать класс для реализации чисел с фиксированной точностью.
9. Разработать класс, представляющий интерфейс для взаимодействия с портами ввода/вывода.
10. Реализация обобщенных алгоритмов в виде Python-классов. Алгоритмы: sum(), max(), sort(), inverce().
11. Разработать класс, предоставляющий интерфейс работы с shelve-хранилищем. Данные в хранилище должны быть зашифрованы…
12. Реализовать свою версию встроенного класса str – строка.
13. Использование метода XOR для простого обратимого шифрования данных.
14. Использование стандартных методов шифрования.
15. Разработать класс для ГПСЧ. Предложить алгоритмы.

```
class Prod:
     def __init__(self, value):
         self.value = value
     def __call__(self, other):
         return self.value * other

x = Prod(2)
print x(3)

print x(4)
# ============================================================================
class Life:
     def __init__(self, name='unknown'):
         print 'Hello', name
         self.name = name
     def __del__(self):
         print 'Goodbye', self.name

brian = Life('Brian')

brian = 'loretta'

print "Before delete"

print brian

del brian

print "after delete"
# ============================================================================
class adder:
     def __init__(self, value=0):
         self.data = value                    # initialize data
     def __add__(self, other):
         self.data += other                   # add other in-place

class addrepr(adder):                        # inherit __init__,
__add__
     def __repr__(self):                      # add string
representation
         return 'addrepr(%s)' % self.data    # convert to string as
code

class addstr(adder):
     def __str__(self):                       # __str__ but no __repr__
         return '[Value: %s]' % self.data    # convert to nice string

x = addstr(3)
x + 1
```

```
print x                                         # runs __str__

print str(x), repr(x)
# =====================================================================
class Employee:
    def __init__(self, lastname, firstname=None):
        self.lastname = lastname
        self.firstname = firstname

    def __str__(self):
        if self.firstname:
            return "%s %s" % (self.firstname, self.lastname)
        else:
            return self.lastname
# =====================================================================
class MyMeta:
    def __str__(cls):
       return "Beautiful class "

x = MyMeta()

print x
# =====================================================================
class MyList:
    def __init__(self, start):
        #self.wrapped = start[:]                    # Copy start: no
side effects
        self.wrapped = []                           # Make sure it's a
list here.
        for x in start: self.wrapped.append(x)
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
    def __getitem__(self, offset):
        return self.wrapped[offset]
    def __len__(self):
        return len(self.wrapped)
    def __getslice__(self, low, high):
        return MyList(self.wrapped[low:high])
    def append(self, node):
        self.wrapped.append(node)
    def __getattr__(self, name):                    # Other members:
sort/reverse/etc
        return getattr(self.wrapped, name)
    def __repr__(self):
        return self.wrapped

if __name__ == '__main__':
```

```
x = MyList('spam')
print x
print x[0]
print x[1:]
print x + ['eggs']
print x * 3
x.append('a')
x.sort( )
for c in x: print c,
```

Литература и источники в Интернет

1. Numeric and Mathematical Modules [электронный ресурс]: https://docs.python.org/3/library/numeric.html (дата обращения – 07.08.2017).
2. Дистанционная подготовка по информатике [электронный ресурс]: https://informatics.mccme.ru/ (дата обращения – 17.08.2017).
3. Прохоренок Н.А. Python 3 и PyQt 5. Разработка приложений [Текст] / Н.А. Прохоренок, В.А. Дронов. – СПб.: БХВ-Петербург, 2016. – 832.: ил.
4. Python: коллекции, часть 4/4: Все о выражениях-генераторах, генераторах списков, множеств и словарей [электронный ресурс]: https://habrahabr.ru/post/320288/ (дата обращения – 2.09.2017).
5. Doug Hellmann. The Python 3 Standard Library by Example [Текст / Электронный ресурс] Addison-Wesley ISBN-13: 978-0-13-429105-5, ISBN-10:0-13-429105-0 https://www.amazon.com/Python-Standard-Library-Example-Developers/dp/0134291050 [Дата обращения - 2.09.2017].
6. Doug Hellmann. Python 3 Module of the Week [электронный ресурс]: https://pymotw.com/3/ (дата обращения – 2.09.2017)
7. Jackson Cooper. Python`s range() Function Explained [электронный ресурс]: http://pythoncentral.io/pythons-range-function-explained/ Python Central (дата обращения – 4.09.2017).
8. Смоленский А., Как работает yield [электронный ресурс]: https://habrahabr.ru/post/132554/ Хабрахабр (дата обращения – 4.09.2017).