ТЕМА 7: КЛАССЫ И ОБЪЕКТЫ В PYTHON

PLEȘCA NATALIA

СОДЕРЖАНИЕ

- Классы и объекты – синтаксис их определения в Python
- Принципы ООП в Python
- Setter-ы и getter-ы в Python
- Декораторы
  - https://m.habr.com/ru/post/560572/

PYTHON И ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

- Python - это мультипарадигмальный язык программирования. Это означает, что он поддерживает разные подходы к программированию
- Одним из популярных подходов к решению проблемы программирования является создание объектов или **объектно-ориентированное программирование** (ООП)
- Концепция ООП в Python направлена на создание кода многократного использования - можно определить компоненты программы в виде классов
- Классы предоставляют возможности объединения данных и функциональностей вместе. Создание нового класса создает новый тип объекта, позволяя создавать новые экземпляры этого типа

КЛАССЫ И ОБЪЕКТЫ

- **Класс** является шаблоном или формальным описанием объекта
- **Объект** представляет экземпляр этого класса, его реальное воплощение
- С точки зрения кода класс объединяет набор функций и переменных, которые выполняют определенную задачу. Объект (например *бабочка*) имеет:
  - **Свойства** или переменные (для бабочки «название», «цвет» являются свойствами). Переменные класса называют и **атрибутами-** они хранят состояние класса
  - **Поведение,** определенное функциями (для бабочки «полет» **-** это поведение). Функции класса еще называют **методами**

ОБЪЕКТЫ В PYTHON

- Всё в Python **является объектами**
- Это означает, что каждый объект в Python имеет метод и значение по той причине, что все объекты базируются на классе. **Класс** – это проект объекта
- Посмотрим на примере, что это значит - ключевое слово **dir**, выдает **список всех методов**, которые можно присвоить строке или числу:

```
name = "Katy"
print("Members of string: ", dir(name))
nmbr = 77
print("Members of number: ", dir(nmbr))
```

```
Members of string:  ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattri
bute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str_
_', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index
', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupp
er', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'sp
lit', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
Members of number:  ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '
__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index
__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__
new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__r
lshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__
setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denomina
tor', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

КЛАСС В PYTHON

- Класс «**butterfly**» можно представить как набросок (абстрактное представление) с разными описаниями. Он содержит все подробности об названии, цвете, размере и т. д. На основании этих описаний можно создавать и изучать бабочек
- Чтобы создать класс в Python, используется ключевое слово **class**
- **Синтаксис:**

```
class MyClass:
    '''This is a docstring. I have created a new class''‘
    pass
```

- Примером для создания класса «**butterfly**» в Python может быть:

```
class Butterfly:
    '''I have created a class Butterfly'''
    name = "Skipper"
```

  - здесь мы используем ключевое слово **class**, чтобы определить класс Butterfly. На основе класса можно строить экземпляры
- **Экземпляр** - это определенный объект, созданный из определенного класса

КЛАСС В PYTHON. 2

- Класс создает новое локальное пространство имен, в котором определены все его атрибуты. В Python ***атрибуты*** могут быть **данными** или **функциями**
- Есть также специальные атрибуты, которые начинаются с двойного знака подчеркивания (__). Например, __**doc**__ дает нам строку документации этого класса
- Как только мы определим класс, создается новый объект класса, с тем же именем. Этот объект класса позволяет нам получать доступ к различным атрибутам, а также создавать новые объекты этого класса
- Пример:

```
class Butterfly:
    '''My first class'''
    name = "Skipper"
print(Butterfly.__doc__)
```

```
My first class
```

КЛЮЧЕВОЕ СЛОВО PASS

- В Python мы используем ключевое слово / оператор «pass», чтобы указать, что ничего не происходит - оператор **pass** используется для обозначения пустых функций, классов и циклов
- С помощью **pass** мы указываем «нулевой» блок. Его можно разместить на одной линии или на отдельной строке
- Пример создания пустого класса:

```
class Batterfly:
    pass
```

МОДИФИКАТОРЫ ДОСТУПА В PYTHON

- Классические объектно-ориентированные языки, такие как C ++ и Java, контролируют доступ к ресурсам класса по используемыми ключевыми словами: **public**, **private**, **protected**

Вспоминаем!

- К **приватным** членам класса, запрещен доступ из среды вне данного класса. Они могут быть обработаны только изнутри класса
- **Публичные** члены (обычно методы, объявленные в классе) доступны извне класса. Требуется объект того же класса для вызова данного открытого метода
- !!!Такое определение - приватных переменных и публичных методов экземпляра, обеспечивает принцип *инкапсуляции данных*
- **Защищенные** члены класса доступны внутри класса, но и для всех своих дочерних классов. В остальном – ниоткуда не позволен доступ к защищенным членам класса. Это и позволяет принцип наследования, примененный наследникам – позволен доступ к ресурсам класса-родитель

ПРЕФИКСЫ ИМЁН В PYTHON

- В Python нет механизма, который эффективно ограничивал бы доступ к любой переменной или методу экземпляра
- Python-у предписывается соглашение о префиксе имени переменной или метода с одинарным или двойным знаком подчеркивания, для эмуляции поведения спецификаторов доступа «protected» и «private»
- Все члены в классе Python **являются публичными по умолчанию**. Любой член может быть доступен из внешней среды данного класса
- Пример:

```
class employee:
    def __init__(self, name, sal):
        self.name=name # public atributte
        self.salary=sal
empl = employee("Nelu", 8000)
print(empl.name + " have " + str(empl.salary))
```

```
Nelu have 8000
```

ЗАЩИЩЕННЫЕ ЧЛЕНЫ КЛАССА

- Соглашение Python о создании **защищенной** переменной экземпляра заключается в добавлении к ней префикса _ (один знак подчеркивания). Это предотвращает доступ к нему, но не в случае если обращение происходит внутри подкласса - это не мешает обращаться к экземпляру или изменять переменные экземпляра из дочернего класса
- Пример:

```
class employee:
    def __init__(self, name, sal):
        self._name=name # protected attribute
        self._salary=sal # protected attribute
empl = employee("Nelu", 8000)
print(empl.name + " have " + str(empl.salary))
```

Следовательно, ответственный программист не будет производить доступ и изменения переменных экземпляра, с префиксом _ вне класса ☺

```
Traceback (most recent call last):
  File "D:\ND\dezvoltarea_aplicatiilor_web_Python\example\empl.py", line 6, in <module>
    print(empl.name + " have " + str(empl.salary))
AttributeError: 'employee' object has no attribute 'name'
```

ПРИВАТНЫЕ ЧЛЕНЫ КЛАССА

- Префикс двойное подчеркивание __, добавленный переменной, делает ее закрытой. Это обеспечивает отсутствие доступа извне класса, где была определена данная переменная. Любая попытка произведения доступа к ней приведет к ошибке *AttributeError*:

```
class employee:
    def __init__(self, name, sal):
        self.__name=name  # private attribute
        self.__salary=sal # private attribute
empl = employee("Nelu", 8000)
print(empl.name + " have " + str(empl.salary))
```

```
Traceback (most recent call last):
  File "D:\ND\dezvoltarea_aplicatiilor_web_Python\example\empl.py", line 6, in <module>
    print(empl.name + " have " + str(empl.salary))
AttributeError: 'employee' object has no attribute 'name'
```

СОЗДАНИЕ ОБЪЕКТОВ В PYTHON

- **Объект** - это экземпляр класса, и представляет собой набор данных (переменных) и методов (функций), которые воздействуют на эти данные. При создании объекта ему выделяется память
- Когда класс определен, определяется только описание объектов. Следовательно, память ему не выделяется
- Пример создания объекта класса Butterfly может быть:

```
object1 = Butterfly()
print(object1.name)
```

```
>>> %Run objects1.py
  Skipper
>>>
```

СВОЙСТВА КЛАССА

- **Свойства являются характеристиками объекта**
- Пример определения и доступа к свойствам объекта:

```
class Butterfly:
    # atribut
    species = "insect"   # атрибуты одинаковы для всех экземпляров класса
    #methods
    def __init__(self, name, color):
        self.name = name      # атрибуты экземпляра, различны для каждого экземпляра класса
        self.color = color
    def fly(self):
        pass
obj = Butterfly("Skipper", "yellow")
print("Butterfly is a {}".format(obj.__class__.species))
print(obj.name + " is " + obj.color)
```

```
Butterfly is a insect
Skipper is yellow
```

МЕТОДЫ ОБЪЕКТОВ

- Объекты, помимо свойств также могут содержать **методы.** Методы определяются в классах
- Методы - это функции, которые принадлежат объекту и которые определяют его поведение
- Пример:

```
class Butterfly:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def fly(self):
        pass
obj = Butterfly("Skipper", "yellow")
obj.fly()
print("Done")
print(obj.name, obj.color, sep="-->")
```

```
Done
Skipper-->yellow
```

КОНСТРУКТОРЫ В PYTHON

- Функции класса, которые начинаются с двойного подчеркивания (__), называются специальными функциями, поскольку они имеют особое значение
- Особый интерес представляет функция **__init __**(). У классов есть особый метод, под названием **__init__**. Эта специальная функция вызывается всякий раз, когда создается новый объект этого класса
- Этот тип функции также называют **конструктором** в объектно-ориентированном программировании (ООП) - этот термин редко встречается в Python. Обычно конструктор используется для инициализации всех переменных

ФУНКЦИЯ (МЕТОД) __INIT__()

- **__init __()** это встроенная функция - все классы имеют функцию с именем **__init __**(), которая всегда выполняется при инициализации класса
- Метод **__init__** вызывается единожды, и не может быть вызван снова внутри программы
- Функция **__init __**() вызывается автоматически каждый раз, когда класс используется для создания нового объекта
- !!! Используйте функцию **__init __()** для присвоения значений свойствам объекта или другим операциям, которые необходимо выполнить при создании объекта

ПРИМЕР

```
class Butterfly:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    name = "Skipper"
object1 = Butterfly("Skipper", "yellow")
print(object1.name, object1.color, sep="-->")
```

```
Skipper-->yellow
```

МЕТОД СОДЕРЖАЩИЙ «PASS». ПРИМЕР

```
class Butterfly:
    def __init__(self, name, color):
        pass
obj = Butterfly("Skipper", "yellow")
print("Done")
```

```
Done
>>>
```

ПАРАМЕТР SELF

- Классам нужен способ, чтобы **ссылаться на самих себя**. Это способ сообщения между экземплярами
- Слово **self** это способ описания любого объекта, буквально
- Параметр **self** является ссылкой на текущий экземпляр класса и используется для доступа к переменным, принадлежащим классу
- Его не обязательно называть **self** - можно называть его как угодно, но это должен быть первый параметр любой функции в классе

ПРИМЕР

```
class Butterfly:
    # atribut
    species = "insect"
    #methods
    def __init__(this, name, color):
        this.name = name
        this.color = color
    def fly(qwerty):
        pass
obj = Butterfly("Skipper", "yellow")
print("Butterfly is a {}".format(obj.species))
print("Butterfly is a {}".format(obj.__class__.species))
obj.fly()
print("Done")
print(obj.name, obj.color, sep="-->")
```

```
def __init__(self, name, color):
    self.name = name
    self.color = color
```

```
Butterfly is a insect
Butterfly is a insect
Done
Skipper-->yellow
```

УДАЛЕНИЕ АТРИБУТОВ И ОБЪЕКТОВ

- Любой атрибут объекта может быть удален в любое время с помощью оператора **del**. Так же при помощи данного оператора можно удалить и объект. Пример:

```
class Butterfly:
    '''My first class'''
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def fly(self):
        pass
obj = Butterfly("Skipper", "yellow")
print(obj.name)
del obj.name
del obj
print(obj.name) # выдаст ошибку
```

ДЕСТРУКТОРЫ

- Все знают что в ООП конструктор вызывается когда создается определенный объект, но тогда когда объект уничтожается, будет вызван специальный метод, называемый **деструктором**
- В Python деструктор внедряется при помощи специального метода __del__()
- Необходимо уточнить что метод не будет вызван, если на объект есть хотя бы одна ссылка
- Но, поскольку интерпретатор сам занимается уничтожением объектов по окончанию жизненного цикла объекта, очень часто использование деструктора не имеет смысла

ПРИНЦИПЫ ООП В PYTHON

Наследование	Процесс использования деталей из нового класса (дочернего) без изменения существующего класса (родительского)
Инкапсуляция	Скрытие личных (приватных) деталей класса от других объектов
Полиморфизм	Концепция использования общих операций по-разному, с целью ввода различных дополнительных данных

GETTER-Ы И SETTER-Ы В PYTHON

- Эти методы в ООП, являются средством получения данных и средством изменения данных
- Согласно принципу инкапсуляции, атрибуты класса делаются приватными, чтобы скрыть и защитить их
- Если бы мы писали программу по рекомендациям Java или PHPOO, мы бы сделали атрибуты: name и color – недоступными. И код бы был:

```
class Butterfly:
    def __init__(self, name, color):
        self.set_name_color(name, color)
    def get_name(self):
        return self.__name
    def get_color(self):
        return self.__color
    def set_name_color(self, name, color):
       self.__name = name
       self.__color = color
    def fly(qwerty):
        pass
try:
    obj = Butterfly("Skipper", "yellow")
    print(obj.get_name(), obj.get_color())
except:
    print("It has no values...")
```

```
Skipper yellow
```

КАК РЕШАЕТ ИНКАПСУЛЯЦИЮ PYTHON?

- Python предлагает другое решение этой проблемы. Решение называется **properties -** значительно упрощается процесс объектно-ориентированного программирования
- В Python **property() -** это встроенная функция, которая создает и возвращает объект свойства
- Синтаксис этой функции:

property(fget=None, fset=None, fdel=None, doc=None), где

fget - функция для получения значения атрибута,

fset - функция для установки значения атрибута,

fdel - функция для удаления атрибута, а

doc - строка (как комментарий)

Как видно из реализации, эти аргументы функции являются необязательными

СОЗДАНИЕ ОБЪЕКТА PROPERTY()

- Объект **property()** может быть создан следующим образом: property()
- У объекта **property()** есть три метода:
  - getter (),
  - setter () и
  - deleter ()
- Пример:

```
class Butterfly:
    def __init__(self, name="a butterfly", color="no color"):
        self.name = name
        self.color = color
    def get_name(self):
        return self.__name
    def get_color(self):
        return self.__color
    def set_name(self, name):
        self.__name = name
    def set_color(self, color):
        self.__color = color
    def fly(qwerty):
        pass
    name = property(get_name, set_name)
    color = property(get_color, set_color)
try:
    obj = Butterfly()
     print(obj.name, obj.color)
except:
    print("It has no values...")
```

ОБЪЕКТ PROPERTY() МОЖНО БЫЛО СОЗДАТЬ И ТАК

```
name = property()
name = name.getter(get_name)
name = name.setter(set_name)
color = property()
color = color.getter(get_color)
color = color.setter(set_color)
```

- … результат тот же


```
name = property(get_name, set_name)
color = property(get_color, set_color)
```

ДЕКОРАТОРЫ В PYTHON

- Перед тем как рассматривать декораторы в Python… вспомним….

РАССМОТРИМ БОЛЕЕ ДЕТАЛЬНО ФУНКЦИИ ИЗ PYTHON

- Разберем два аспекта связанные с функциями в *Python*
  - Во-первых: функция – это объект специального вида, поэтому ее можно передавать в качестве аргумента другим функциям
  - Во-вторых: внутри функций можно создавать другие функции, вызывать их и возвращать как результат через *return*
- Остановимся на этих моментах более подробно…

ФУНКЦИЯ КАК ОБЪЕКТ

- В *Python* передача одной функции в качестве аргумента другой функции – это нормальная практика
- Например, если есть список целых чисел, и необходимо на его основе получить другой список, элементами которого будут квадраты первого, то такую задачу можно решить в одну строчку, использовав lambda-функцию
- `print(list(map(lambda x: x**2, [1, 2, 3, 4, 5, 7])))`

```
[1, 4, 9, 16, 25, 49]
```

- !!! В *Python* функция – это специальный объект, который имеет метод *__call__()*

ФУНКЦИЯ ВНУТРИ ФУНКЦИИ

- Вторым важным свойством функции, для понимания темы декораторов, является то, что их можно создавать, вызывать и возвращать из других функций. На этом построена идея замыкания (*closures*) – вспоминаем “область видимости переменных”
- Рассмотрим такой пример - внутри функции ***summ()*** создается еще одна функция, которая называется ***into()***. Функция **summ()** возвращает функцию **into()** как результат работы:

```
def summ(nmbr):
    def into(nm):
        return nmbr + nm
    return into
# функция summ() вызывается следующим образом
print(summ(5)(7))  # 12
```

ТЕПЕРЬ МОЖНО ПЕРСОНАЛИЗИРОВАТЬ НАШУ ФУНКЦИЮ…

```
def summ(nmbr):
    def into(nm):
        return nmbr + nm
    return into

new_summ = summ(1870)
print(new_summ(77))
print(new_summ(888))
```

```
1947
2758
```

ЧТО ТАКОЕ ДЕКОРАТОР ФУНКЦИИ В *PYTHON*?

- Конструктивно декоратор в *Python* представляет собой некоторую функцию, аргументом которой является другая функция
- Декоратор предназначен для добавления дополнительного функционала к данной функции без изменения содержимого последней
- По сути, декоратор принимает функцию, добавляет некоторую функциональность и возвращает ее
- Таким образом, **декоратор** - это вызываемое, которое возвращает вызываемое (функция в которой определена другая функция)

СОЗДАНИЕ ПРОСТОГО ДЕКОРАТОРА. I

- Пусть будут следующие 2 функции:

```
def subst(a,b):
    print("a-b = ", a-b)
def summ(x,y):
    print("x+y = ", x+y)
subst(88, 77)
summ(88, 77)
```

- Дополним их так, чтобы перед вызовом основного кода функции печаталась строка *“Run function”*, а по окончании – *“Stop function”*
- Сделать это можно двумя способами
  - Первый – это добавить указанные строки в начало в конец каждой функции, но это не очень удобно, т.к. если мы захотим убрать это, нам придется снова модифицировать тело функции. А если они написаны не нами, либо являются частью общей кодовой базы проекта, сделать это будет уже не так просто
  - Второй вариант – возможности функций в Python… ☺

СОЗДАНИЕ ПРОСТОГО ДЕКОРАТОРА. 2

- Создадим «обёртку» для этих функций

```
def decore(func):
    def wrapper(t,z):
        print("Run function")
        func(t,z)
        print("Stop function")
    return wrapper
```

Полученный результат:

```
def subst(a,b):
    print("a-b = ", a-b)
def summ(x,y):
    print("x+y = ", x+y)

def decore(func):
    def wrapper(t,z):
        print("Run function")
        func(t,z)
        print("Stop function")
    return wrapper
subst_wrapped = decore(subst)
summ_wrapped = decore(summ)
subst(88,77)
summ(88, 77)
subst_wrapped(88, 77)
summ_wrapped(88, 77)
```

```
Run function
a-b =  11
Stop function
Run function
x+y =  165
Stop function
```

ИДЕЯ «ДЕКОРАТОРА»

- Почему «декоратор»?
- Можно увидеть, что функция декоратора добавила некоторый дополнительный функционал в исходную функцию. Это похоже на упаковку подарка. Декоратор действует как обертка
- Характер объекта, который был украшен (фактический подарок внутри), не меняется. Но теперь, он выглядит симпатично (так как он был украшен)
- Это обычная конструкция для Python, и по этой причине есть синтаксис, упрощающий процесс декорации

ДЕКОРАТОРЫ МЕТОДОВ В PYTHON

- Методы классов также можно объявлять с **декоратором**
- Функции и методы называются **вызываемыми**, так как они могут быть вызваны
- Фактически, любой объект, который реализует специальный метод **__call __()**, называется вызываемым
- В Python есть и предопределенные декораторы
- Используется символ @ вместе с именем функции декоратора, и это все вместе помещается над определением функции, которая будет декорирована

ПРЕДЫДУЩИЙ ПРИМЕР МОЖНО ПЕРЕПИСАТЬ…

```
def decore(func):
    def wrapper(t,z):
        print("Run function")
        func(t,z)
        print("Stop function")
    return wrapper
@decore
def subst(a,b):
    print("a-b = ", a-b)
@decore
def summ(x,y):
    print("x+y = ", x+y)

subst(88,77)
summ(88, 77)
```

```
Run function
a-b =  11
Stop function
Run function
x+y =  165
Stop function
```

И ЕСЛИ ВЕРНУТСЯ К PROPERTY() И SETTER()-АМ

- **…и использовать декораторы, можно переписать код**

```
class Butterfly:
    def __init__(self, name="a butterfly", color="no color"):
        self.name = name
        self.color = color
    def get_name(self):
        return self.__name
    def get_color(self):
        return self.__color
    def set_name(self, name):
        self.__name = name
    def set_color(self, color):
        self.__color = color
    def fly(qwerty):
        pass
    name = property(get_name, set_name)
    color = property(get_color, set_color)
```

```
try:
obj = Butterfly()
    print(obj.name, obj.color)
    obj.name = "Skipper"
    obj.color = "Blue"
    print(obj.name, obj.color)
except:
    print("It has no values...")
```

```
a butterfly no color
Skipper Blue
```

```
class Batterfly:
    def __init__(self, name="a butterfly", color="no color"):
        self.name = name
        self.color = color
    @property
    def name(self):
        return self.__name
    @property
    def color(self):
        return self.__color
    @name.setter
    def name(self, name):
        self.__name = name
    @color.setter
    def color(self, color):
        self.__color = color
    def fly(qwerty):
        pass
try:
    obj = Batterfly("Sulphurus", "yellow")
    print(obj.name + " is " + obj.color)
    obj.color = "Red"
    print(obj.name + " is " + obj.color)
except:
    print("It has no values...")
```

```
Sulphurus is yellow
Sulphurus is Red
```

ОБЪЯСНЕНИЯ

- Метод типа **getter**, украшен **«@property»**, то есть помещаем эту строку непосредственно перед заголовком метода
- Метод, который должен функционировать как **setter**, украшен **"@name.setter"** или **"@color.setter**
- Следует отметить что:
  - мы просто помещаем строку кода «**self.name = name**» и «**self.color = color**» в метод **__init__**
  - мы написали «два» метода с одинаковым именем и разным количеством параметров «**def name(self)**» и «**def name(self, name)**». Но мы уже знаем, что теоретически это невозможно. Здесь это возможно - из-за декоратора

ПОВТОРИМ

Какой будет результат интерпретирования?

1)

```
class Foo:
    def printLine(self, line='Python'):
        print(line)
o1 = Foo()
o1.printLine('Java')
```

2)

```
class Point:
    def __init__(self, x = 0, y = 0):
      self.x = x+1
      self.y = y+1

p1 = Point()
print(p1.x, p1.y)
```

РЕЗУЛЬТАТЫ

1) Java
2) 1 1