Язык программирования Питон (Python)

Объектно-ориентированное программирование

Объектно-ориентированный стиль программирования основан на понятии ***объекта***,
который заключает в себе ***данные*** и ***процедуры***

Методология программирования, предполагает представление программы в виде совокупности объектов и организации взаимодействия между ними.
(*в качестве основных логических конструктивных элементов выступают объекты, а не алгоритмы***).**

Каждый ***объект*** интегрирует в себе некоторую структуру ***данных*** и
доступные только ему ***процедуры*** обработки этих данных, называемые ***методами***.

Объединение данных и процедур в одном объекте называется ***инкапсуляцией***.

Для описания объектов служат ***классы***. Любой объект – экземпляр класса.
Классы образуют иерархию ***наследования*** (*при создании новых объектов их атрибуты могут добавляться или наследоваться от объектов-предков*).
Класс определяет свойства и методы объекта, принадлежащего этому классу.

Переменные для хранения данных называют ***полями***.

Вместе поля и методы называют ***атрибутами***. Они могут:

- принадлежать отдельному экземпляру объекта;
- принадлежать всему классу.

В процессе работы с объектами допускается ***полиморфизм*** – возможность использования методов с одинаковыми именами для обработки данных разных типов.

Для справки.

Процедурно-ориентированный (***императивный***) стиль программирования – программы состоят из последовательности операторов (инструкций), которые манипулируют данными и задают процедуру решения задачи.

Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, т.е. значений исходных данных, в заключительное, т.е. в результаты.

Последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка

Функциональное (***аппликативное***) программирование – способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции, а единственным правилом композиции — оператор суперпозиции функций.

Выполнение программы заключается в вычислении результатов функций (в математическом их понимании) от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы.

Программа описывает связи между входными и выходными параметрами, некое тело взаимодействий – зависимости, отношения, преобразования и композиции функций, т.е. описывает "что нужно сделать" не опускаясь до конкретных случаев,

а вот как это делать, решает транслятор.

Замечания. История развития методологий программирования
движима борьбой со сложностью разработки программного обеспечения.

Сложность больших программных систем, в создании которых участвует сразу большое количество разработчиков, уменьшается, если на верхнем уровне не видно деталей реализации нижних уровней.

Один из подходов к реализации данного принципа – использование ***инкапсуляции*** (***incapsulation***) – сокрытия информации о внутреннем устройстве объекта, при котором работа с объектом может вестись только через его общедоступный (public) интерфейс.

(*Т.о., другие объекты не должны вмешиваться в "дела" объекта, кроме как используя вызовы методов*.)

В языке **Python** инкапсуляции не придается принципиального значения:
ее соблюдение зависит от дисциплинированности программиста.

– можно получить доступ к некоторому атрибуту (не методу) напрямую, используя ссылку, если этот атрибут описан в документации как часть интерфейса класса.

– такие атрибуты называются ***свойствами*** (***properties***).

(*В других языках программирования принято для доступа к свойствам создавать специальные методы*).

Подчеркивание ("_") в начале имени атрибута указывает на то, что он не входит в общедоступный интерфейс.

– обычно применяется ***одиночное подчеркивание***, которое в языке не играет особой роли, но говорит программисту: "этот метод только ***для внутреннего использования***".

– ***двойное подчеркивание*** работает как указание на то, что ***атрибут - приватный***. При этом атрибут все же доступен, но уже под другим именем (см. пример ниже)

Создание класса

Класс ***создаётся*** ключевым словом **`class`**.

– поля и методы класса записываются в блоке кода с отступом.

– ***переменные (поля) класса***
- доступ к ним могут получать все экземпляры этого класса.
- когда любой из объектов изменяет переменную класса, это изменение отразится и во всех остальных экземплярах того же класса.
- при обращении перед именем переменной добавляют имя класса через точку.

– ***переменные (поля) объекта***
- принадлежат каждому отдельному экземпляру класса. (*у каждого объекта есть своя собственная копия поля не связанная с другими такими же полями в других экземплярах*).
- при обращении перед именем переменной добавляют **`self`** через точку.
- переменная объекта с тем же именем, что и переменная класса, сделает недоступной («спрячет») переменную класса

– ***методы объекта***
- должны иметь дополнительный параметр – **`self`** , добавляемый к началу списка параметров и указывающий на сам объект экземпляра класса.
- при вызове метода этому параметру значение присваивать не нужно – его укажет Python.
- если метод не принимает аргументов, у него будет один аргумент – **`self`**.

– ***методы класса***
- могут определяться как **`classmethod`** или **`staticmethod`**, в зависимости от того, нужно ли нам знать, в каком классе мы находимся

– ***метод*** **`__init__`** запускается при создании объекта класса:
- используется для осуществления необходимой инициализации;
- явным образом не вызывается;
- аргументы передаются в скобках после имени класса.

– ***метод*** **`__del__`** вызывается при уничтожении объекта. При этом занимаемая им память возвращается операционной системе.

Пример: (*создание класса, объектов класса, использование подчёркиваний в именах атрибутов*)

```
class Person:

    a = 1
    _b = 2
    __c = 3

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

    def sayHi(self):
        print('Привет! Меня зовут', self.name)

p1 = Person('Иван')                         # создание объекта класса
p2 = Person('Мария')

dir(Person)    ->  ['_Person__c', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
                    '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
                    '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
                    '__subclasshook__', '__weakref__', '_b', 'a', 'name', 'sayHi']

p1.sayHi()                                  # вызов метода
p2.sayHi()
Person('Илья').sayHi()                      # тот же вызов короче
               ->  Привет! Меня зовут Иван
                   Привет! Меня зовут Мария
                   Привет! Меня зовут Илья

p1.a           ->  1                        # обращение к свойству a
p1._b          ->  2                        # обращение к свойству _b
p1._Person__c  ->  3                        # обращение к свойству __c
```

Пример: (*Использование конструкторов, деструкторов, методов объекта, методов класса*)

```
class Robot:
    '''Представляет робота с именем.'''

    # Переменная класса, содержащая количество роботов
    population = 0

    def __init__(self, name):
        '''Инициализация данных. (Конструктор класса)'''
        self.name = name
        print('(Инициализация {0})'.format(self.name))
        # При создании робота увеличивается значение 'population'
        Robot.population += 1

    def __del__(self):
        '''Уничтожение робота. (Деструктор класса)'''
        print('{0} уничтожается!'.format(self.name))
        Robot.population -= 1
        if Robot.population == 0:
            print('{0} был последним.'.format(self.name))
        else:
            print('Осталось {0:d} работающих роботов.'.format(Robot.population))

    def sayHi(self):
        '''Приветствие робота. (Метод объекта)  '''
        print('Приветствую! Мои хозяева называют меня {0}.'.format(self.name))

    def howMany():
        '''Выводит численность роботов. (Метод класса)  '''
        print('У нас {0:d} роботов.'.format(Robot.population))

    howMany = staticmethod(howMany)
```

Продолжение примера

```
droid1 = Robot('R2-D2') ->  (Инициализация R2-D2)
droid1.sayHi()          ->  Приветствую! Мои хозяева называют меня R2-D2.
Robot.howMany()         ->  У нас 1 роботов.

droid2 = Robot('C-3PO') ->  (Инициализация C-3PO)
droid2.sayHi()          ->  Приветствую! Мои хозяева называют меня C-3PO.
Robot.howMany()         ->  У нас 2 роботов.

print("\nЗдесь роботы могут проделать какую-то работу.\n")
print("Роботы закончили свою работу. Давайте уничтожим их.")

del droid1              ->  R2-D2 уничтожается!
                            Осталось 1 работающих роботов.
del droid2              ->  C-3PO уничтожается!
                            C-3PO был последним.
```

Замечание *В языке Python объект уничтожается в случае удаления последней ссылки на него либо в результате сборки мусора, если объект оказался в неиспользуемом цикле ссылок.*

Так как Python сам управляет распределением памяти, ***деструкторы*** *в нем нужны очень редко.*

Обычно в том случае, когда объект управляет ресурсом, который нужно корректно вернуть в определенное состояние.

Пример: (*Использование специальных методов для доступа к свойствам объекта*)

```
class Dog():
    name=""
    def __init__ (self, newName):     # конструктор - вызывается при созд.объекта
        self.name = newName
    def setName (self, newName):      # метод – изменяет имя
        self.name = newName
    def getName (self):
        return self.name              # метод - возвращает текущее имя объекта

dog1 =Dog('Шарик')
print(dog1.getName())                 ->  Шарик
dog1.setName('Бобик')
print(dog1.getName())                 ->  Бобик
```

Замечания.

Если имя переменной начинается с двойного подчёркивания – она является ***приватной*** (private).

(*принято имя любой переменной, которая должна использоваться только внутри класса или объекта, начинать с подчёркивания*)

Все остальные атрибуты класса являются ***публичными*** (public) и могут использоваться в других классах/объектах, а все методы – ***виртуальными*** (virtual) – т.е. могут быть переопределены в классах - наследниках

Наследование

– возможность создавать специализированные классы на основе ***базовых*** – способ обеспечения многократного использования одного и того же кода в объектно-ориентированном программировании.

– при определении ***производного*** класса (***потомка***) указываем имена его базовых классов в виде кортежа, следующего за сразу за его именем.

(*если при наследовании перечислено более одного класса, это называется* ***множественным наследованием***)

– из конструктора производного класса метод `__init__` базового класса вызывается явно, чтобы инициализировать часть объекта, относящуюся к базовому классу.

– если производный класс не имеет своего конструктора, то при создании объекта производного класса будет вызван метод `__init__` базового класса.

– можно вызывать методы базового класса, одноимённые методам производного класса, предваряя запись имени метода именем базового класса, а затем передавая переменную `self` вместе с другими аргументами ().

Замечание *Python всегда начинает поиск методов в самом классе. Если же он не находит метода, он начинает искать методы, принадлежащие базовым классам по очереди, в порядке, в котором они перечислены в кортеже при определении класса.*

– класс называется ***абстрактным***, если он предназначен только для наследования. Экземпляры абстрактного класса обычно не имеют большого смысла. Классы с рабочими экземплярами называются ***конкретными***.

Пример:

Пусть необходимо создать программу, содержащую описание классов Работника (Employee) и Клиента (Customer). Эти классы имеют общие свойства, присущие всем людям, поэтому создадим базовый класс Человек (Person) и наследуем от него дочерние классы Employee и Customer

```
# Код, описывающий иерархию классов:

class Person():
    name=""                                 # имя у любого человека

class Employee (Person):
    job_title=""                            # наименование должности работника

class Customer (Person):
    email=""                                # почта клиента

# Создадим объекты на основе классов и заполним их поля (наследование полей):

johnSmith = Person()
johnSmith.name = "John Smith"

janeEmployee = Employee()
janeEmployee.name = "Jane Employee"         # поле наследуется от класса Person
janeEmployee.job_title = "Web Developer"

bobCustomer = Customer()
bobCustomer.name = "Bob Customer"           # поле наследуется от класса Person
bobCustomer.email = "send_me@spam.com"

print(johnSmith.name)                         ->  John Smith
print(bobCustomer.name, bobCustomer.email)    ->  Bob Customer send_me@spam.com
```

Пример: (*Наследование методов*)

```
# Код, описывающий иерархию классов:

class Person():
    name=""                          # имя у любого человека
    def __init__ (self):             # конструктор базового класса
        print("Создан человек")

class Employee (Person):
    job_title=""                     # наименование должности работника

class Customer (Person):
    email=""                         # почта клиента

# Создадим объекты на основе классов (наследование конструктора):

johnSmith = Person()            ->  Создан человек
janeEmployee = Employee()       ->  Создан человек
bobCustomer = Customer()        ->  Создан человек
```

Пример: (*Если дочерние классы содержат собственные методы, то выполняться будут они*)

```
# Код, описывающий иерархию классов:

class Person():
    name=""                          # имя у любого человека
    def __init__ (self):             # конструктор базового класса
        print("Создан человек")

class Employee (Person):
    job_title=""                     # наименование должности работника
    def __init__(self):              # конструктор дочернего класса
        print ("Создан работник")

class Customer (Person):
    email=""                         # почта клиента
    def __init__(self):              # конструктор дочернего класса
        print ("Создан покупатель")

# Создадим объекты на основе классов и заполним их поля:

johnSmith = Person()                 ->  Создан человек
janeEmployee = Employee()            ->  Создан работник
bobCustomer = Customer()             ->  Создан покупатель
```

Пример: (*Вызов конструктора базового класса из конструктора дочернего класса*)

```
# Код, описывающий иерархию классов:

class Person():
    name=""                             # имя у любого человека
    def __init__ (self):                # конструктор базового класса
        print("Создан человек")

class Employee (Person):
    job_title=""                        # наименование должности работника
    def __init__ (self):
        Person.__init__ (self)          # вызываем конструктор базового класса
        print ("Создан работник")

class Customer (Person):
    email=""                            # почта клиента
    def __init__ (self):
        Person.__init__ (self)          # вызываем конструктор базового класса
        print ("Создан покупатель")

# Создадим объекты на основе классов и заполним их поля:

johnSmith = Person()                    ->  Создан человек
janeEmployee = Employee()               ->  Создан человек
                                            Создан работник
bobCustomer = Customer()                ->  Создан человек
                                            Создан покупатель
```

Пример:

Программа, которая отслеживает информацию о преподавателях и студентах в колледже.
Вначале создаём общий класс с именем SchoolMember, а затем классы преподавателя и студента, которые наследуют этот класс, т.е. становятся подтипами этого типа (класса).
После этого добавляем любые специфические характеристики к этим подтипам.

```
class SchoolMember:
    '''Представляет любого человека в школе (базовый класс).'''

    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Создан SchoolMember: {0})'.format(self.name))

    def tell(self):
        '''Вывести информацию.'''
        print('Имя:"{0}" Возраст:"{1}"'.format(self.name, self.age), end=" ")

            #  Параметр end используется в методе tell() для того,
            #  чтобы новая строка начиналась через пробел после вызова print().
```

Продолжение примера

```
class Teacher(SchoolMember):
    '''Представляет преподавателя (производный класс).'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Создан Teacher: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Зарплата: "{0:d}"'.format(self.salary))


class Student(SchoolMember):
    '''Представляет студента (производный класс).'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Создан Student: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Оценки: "{0:d}"'.format(self.marks))
```

Продолжение примера

```
t = Teacher('Доц. С.П. Иванов', 40, 50000)
                                        ->  (Создан SchoolMember: Доц. С.П. Иванов)
                                            (Создан Teacher: Доц. С.П. Иванов)

s = Student('Петров И.А.', 25, 7)
                                        ->  (Создан SchoolMember: Петров И.А.)
                                            (Создан Student: Петров И.А.)

members = [t, s]
for member in members:
    member.tell()           # работает как для преподавателя, так и для студента

                        ->  Имя:"Доц. С.П. Иванов" Возраст:"40" Зарплата: "50000"
                            Имя:"Петров И.А." Возраст:"25" Оценки: "7"
```