Классы и объектно-ориентированное программирование

Перегрузка операций

```
class C2: ...                   # Создание объектов суперклассов
class C3: ...

class C1(C2, C3):
   def __init__(self, who):     # Установка name при конструировании
      self.name = who           # self является либо I1, либо I2

I1 = C1('bob')                  # Установка I1.name в 'bob'
I2 = C1('sue')                  # Установка I2.name в 'sue'
print(I1.name)                  # Выводит 'bob'
```

- Метод __init__ известен как конструктор из за момента своего запуска. Он является самым часто используемым представителем крупной группы методов, называемых методами перегрузки операций. Такие методы обычным образом наследуются в деревьях классов и содержат в начале и конце своих имен по два символа подчеркивания, чтобы акцентировать внимание на их особенности. Скажем, для реализации пересечения множеств класс может либо предоставить метод по имени intersect, либо перегрузить операцию выражения & и обеспечить требующуюся логику за счет реализации метода по имени__ and__ . Поскольку схема с операциями делает экземпляры больше похожими на встроенные типы, она позволяет ряду классов предоставлять согласованный и естественный интерфейс, а также быть совместимыми с кодом, который ожидает встроенного типа. Однако за исключением конструктора__init__,которыйприсутствуетвбольшинствереалистичныхклассов, во многих программах лучше использовать более просто именованные методы, если только их объекты не подобны объектам встроенных типов.

Полиморфизм и классы

```
class Employee:                          # Универсальный суперкласс
    def computeSalary(self): ...         # Общие или стандартные линии поведения
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...

class Engineer(Employee):                # Специализированный подкласс
    def computeSalary(self): ...         # Что-то специальное

bob = Employee()     # Стандартное поведение
sue = Employee()     # Стандартное поведение
tom = Engineer()     # Специальный расчет заработной платы

company = [bob, sue, tom]                # Составной объект
for emp in company:
    print(emp.computeSalary())           # Выполнить версию для данного объекта:
                                         #   стандартную или специальную
```

Полиморфизм и классы

В других приложениях полиморфизм также может использоваться для сокрытия (т.е. инкапсуляции} отличий в интерфейсах. Скажем, программа обработки потоков данных может быть реализована так, чтобы ожидать объекты с методами ввода и вы вода, не заботясь о том, что в действительности делают эти методы:

```
def processor(reader, converter, writer):
    while True:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Полиморфизм и классы

```
class Reader:
    def read(self): ...          # Стандартное поведение и инструменты
    def other(self): ...
class FileReader(Reader):
    def read(self): ...          # Читать из локального файла
class SocketReader(Reader):
    def read(self): ...          # Читать из сетевого сокета
...
processor(FileReader(...), Converter, FileWriter(...))
processor(SocketReader(...), Converter, TapeWriter(...))
processor(FtpReader(...), Converter, XmlWriter(...))
```

пример

- В Python принято до вольно строгое соглашение начинать имена модулей с буквы нижнего регистра, а име на классов — с буквы верхнего регистра. Как и имя аргументов self в методах, язык этого не требует, но соглашение получило настолько широкое распространение, что отклонение от него может сбить с толку тех, кто впоследствии будет читать ваш код.

```
# Файл person.py (начало)
class Person:    # Начало класса
```

Написание кода конструкторов

- Первое, что мы хотим делать с помощью класса Person, связано с регистраци ей основных сведений о людях — заполнением полей записей, если так понятнее. Разумеется, в терминологии Python они известны как атрибуты объекта экземпляра и обычно создаются путем присваивания значений атрибутам self в функциях мето дов класса. Нормальный способ предоставления атрибутам экземпляра первоначаль ных значений предусматривает их присваивание через self в методе конструктора __init__,которыйсодержиткод,автоматическивыполняемыйPythonкаждыйраз, когда создается экземпляр. Давайте добавим к классу метод конструктора:

```
# Добавление инициализации полей записи

class Person:
    def __init__(self, name, job, pay): # Конструктор принимает три аргумента
        self.name = name                # Заполнить поля при создании
        self.job = job                  # self - новый объект экземпляра
        self.pay = pay
```

-

Написание кода конструкторов

- Первое, что мы хотим делать с помощью класса Person, связано с регистраци ей основных сведений о людях — заполнением полей записей, если так понятнее. Разумеется, в терминологии Python они известны как атрибуты объекта экземпляра и обычно создаются путем присваивания значений атрибутам self в функциях мето дов класса. Нормальный способ предоставления атрибутам экземпляра первоначаль ных значений предусматривает их присваивание через self в методе конструктора __init__,которыйсодержиткод,автоматическивыполняемыйPythonкаждыйраз, когда создается экземпляр. Давайте добавим к классу метод конструктора:

```
# Добавление инициализации полей записи

class Person:
    def __init__(self, name, job, pay): # Конструктор принимает три аргумента
        self.name = name                # Заполнить поля при создании
        self.job = job                  # self - новый объект экземпляра
        self.pay = pay
```

-

Написание кода конструкторов

```
# Добавление стандартных значений для аргументов конструктора
class Person:
    def __init__(self, name, job=None, pay=0):    # Нормальные аргументы функции
        self.name = name
        self.job = job
        self.pay = pay
```

- Такой код

добавление методов, реализующих поведение

```
# Обработка встроенных типов: строки, изменяемость

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])    # Извлечение фамилии из объекта
    sue.pay *= 1.10                # Предоставление этому объекту повышения
    print('%.2f' % sue.pay)
```

Написание кода методов

- На самом деле мы здесь хотим задействовать концепцию проектирования программного обеспечения, известную как инкапсуляция — помещение операционной логики в оболочку интерфейсов, чтобы код каждой операции был написан только один раз в программе. Тогда если в будущем возникнет необходимость в изменении, то изменять нужно будет только одну копию. Более того, мы можем практически произвольно из менять внутренности одиночной копии, не нарушая работу кода, который ее потребляет.

```
# Добавление методов для инкапсуляции операций с целью повышения удобства
сопровождения

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):                          # Методы реализации поведения
        return self.name.split()[-1]             # self - подразумеваемый объект
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))    # Потребуется изменять
                                                    #   только здесь

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

```
print(bob.lastName(), sue.lastName())      # Использовать новые методы
sue.giveRaise(.10)                          # вместо жесткого кодирования
print(sue.pay)
```

Написание кода методов

- На самом деле мы здесь хотим задействовать концепцию проектирования программного обеспечения, известную как инкапсуляция — помещение операционной логики в оболочку интерфейсов, чтобы код каждой операции был написан только один раз в программе. Тогда если в будущем возникнет необходимость в изменении, то изменять нужно будет только одну копию. Более того, мы можем практически произвольно из менять внутренности одиночной копии, не нарушая работу кода, который ее потребляет.

```
# Добавление методов для инкапсуляции операций с целью повышения удобства
сопровождения

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):                          # Методы реализации поведения
        return self.name.split()[-1]             # self - подразумеваемый объект
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))    # Потребуется изменять
                                                    #   только здесь

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

```
print(bob.lastName(), sue.lastName())      # Использовать новые методы
sue.giveRaise(.10)                          # вместо жесткого кодирования
print(sue.pay)
```

перегрузка операций

- Реализация отображения

```
Bob Smith 0
Sue Jones 100000
Smith Jones
<__main__.Person object at 0x00000000029A0668>
```

- мы можем использовать вторые по частоте применения в Python методы перегрузки операций после __init__ : метод __repr__ , который мы реализуем здесь, и его двойник__ str__

```
# Добавление метода перегрузки операции __repr__ для вывода объектов
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):                                # Добавленный метод
        return '[Person: %s, %s]' % (self.name, self.pay) # Строка для вывода
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

перегрузка операций

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
```