Лабораторная работа №13
Объектно-ориентированное программирование

1 Цель работы

Изучить объектно-ориентированное программирование и научиться применять полученные знания на практике.

2 Краткая теория

2.1 Основы объектно-ориентированного подхода

Начнем с примера. Предположим, что существует набор строковых переменных для описания адреса проживания некоторого человека:

```
addr_name = 'Ivan Ivanov'  # имя человека
addr_line1 = '1122 Main Street'
addr_line2 = ''
addr_city = 'Panama City Beach'
addr_state = 'FL'
addr_zip = '32407'  # индекс
```

Напишем функцию, которая выводит на экран всю информацию о человеке:

```
def printAddress(name, line1, line2, city, state, zip):
    print(name)
    if len(line1) > 0:
        print(line1)
    if len(line2) > 0:
        print(line2)
    print(city + ", " + state + " " + zip)


# Вызов функции, передача аргументов:
printAddress(addr_name, addr_line1, addr_line2, addr_city, addr_state,
addr_zip)
```

В результате работы программы:

Ivan Ivanov

1122 Main Street

Panama City Beach, FL 32407

Предположим, что изменились начальные условия и у человека в адресе появился второй индекс. Почему бы и нет? Создадим новую переменную:

```
# добавим переменную, содержащую индекс
addr_zip2 = "678900"
```

Изменим функцию printAddress() с учетом новых сведений:

```
def printAddress(name, line1, line2, city, state, zip, zip2):
    # добавили параметр zip2
    print(name)
    if len(line1) > 0:
        print(line1)
    if len(line2) > 0:
        print(line2)
    # добавили вывод на экран переменной zip2
    print(city + ", " + state + " " + zip + zip2)
```

```
# добавили новый аргумент addr_zip2:
printAddress(addr_name, addr_line1, addr_line2, addr_city, addr_state,
             addr_zip, addr_zip2)
```

Пришлось несколько раз добавить новый индекс, чтобы функция printAddress() корректно отработала при новых условиях. Какой недостаток у рассмотренного подхода? Огромное количество переменных! Чем больше сведений о человеке хотим обработать, тем больше переменных мы должны создать. Конечно, можно поместить всё в список (элементами списка тогда будут строки), но в Python есть более универсальный подход для работы с наборами разнородных данных, ориентированный на объекты.

Создадим структуру данных (класс) с именем Address, которая будет содержать все сведения об адресе человека:

```
class Address:  # имя класса выбирает программист
    name = ""  # поля класса
    line1 = ""
    line2 = ""
    city = ""
    state = ""
    zip = ""
```

Класс задает шаблон для хранения адреса. Превратить шаблон в конкретный адрес можно через создание объекта (экземпляра) класса Address:

```
homeAddress = Address()
```

Теперь можем заполнить поля объекта конкретными значениями:

```
# заполняем поле name объекта homeAddress:
homeAddress.name = "Ivan Ivanov"
homeAddress.line1 = "701 N. C Street"
homeAddress.line2 = "Carver Science Building"
homeAddress.city = "Indianola"
homeAddress.state = "IA"
homeAddress.zip = "50125"
```

Создадим еще один объект класса Address, который содержит информацию о загородном доме того же человека:

```
# переменная содержит адрес объекта класса Address:
vacationHomeAddress = Address()
```

Зададим поля объекта, адрес которого находится в переменной vacationHomeAddress:

```
vacationHomeAddress.name = "Ivan Ivanov"
vacationHomeAddress.line1 = "1122 Main Street"
vacationHomeAddress.line2 = ""
vacationHomeAddress.city = "Panama City Beach"
vacationHomeAddress.state = "FL"
vacationHomeAddress.zip = "32407"
```

Выведем на экран информацию о городе для основного и загородного адресов проживания (через указание имен объектов):

```
print("Основной адрес проживания "+homeAddress.city)
print("Адрес загородного дома "+vacationHomeAddress.city)
```

Изменим исходный текст функции printAddress() с учетом полученных знаний об объектах:

```
def printAddress(address):  # передаем в функцию объект
    print(address.name)  # выводим на экран поле объекта
    if len(address.line1) > 0:
        print(address.line1)
    if len(address.line2) > 0:
        print(address.line2)
    print(address.city + ", " + address.state + " " + address.zip)
```

Если объекты homeAddress и vacationHomeAddress ранее были созданы, то можем вывести информацию о них, передав в качестве аргумента функции printAddress():

```
printAddress(homeAddress)
printAddress(vacationHomeAddress)
```

В результате выполнения программы получим:

Ivan Ivanov
701 N. C Street
Carver Science Building
Indianola, IA 50125
Ivan Ivanov
1122 Main Street
Panama City Beach, FL 32407

Возможности классов и объектов не ограничиваются лишь объединением переменных под одним именем, т.е. хранением состояния объекта. Классы также позволяют задавать функции внутри себя (методы) для работы с полями класса, т.е. влиять на поведение объекта.

Создадим класс Dog:

```
class Dog:
    age = 0  # возраст собаки
    name = ""  # имя собаки
    weight = 0  # вес собаки

    # первым аргументом любого метода всегда является self, т.е. сам объект
    def bark(self):  # функция внутри класса называется методом
        # self.name - обращение к имени текущего объекта-собаки
        print(self.name, "говорит гав")
        # создадим объект myDog класса Dog:


myDog = Dog()
# присвоим значения полям объекта myDog:
myDog.name = "Spot"  # придумываем имя созданной собаке
myDog.weight = 20  # указываем вес собаки
myDog.age = 1  # возраст собаки
# вызовем метод bark() объекта myDog, т.е. попросим собаку подать голос:
myDog.bark()
# полная форма для вызова метода myDog.bark() будет: Dog.bark(myDog),
# т.е. полная форма требует в качестве первого аргумента сам объект - self
```

Результат работы программы:

Spot говорит гав

Данный пример демонстрирует объектно-ориентированный подход в программировании, когда создаются объекты, приближенные к реальной жизни. Между объектами происходит взаимодействие по средствам вызова методов. Поля объекта (переменные) фиксируют его состояние, а вызов метода приводит к реакции объекта и/или изменению его состояния (изменению переменных внутри объекта).

```
class Dog:
    name = ""

    # конструктор вызывается в момент создания объекта этого типа;
    # специальный метод Python, поэтому два нижних подчеркивания
    def __init__(self):
        print("Родилась новая собака!")


# создаем собаку (объект myDog класса Dog)
myDog = Dog()
```

В предыдущем примере между созданием объекта myDog класса Dog и присвоению ему имени (myDog.name="Spot") прошло некоторое время. Может случиться так, что программист забудет указать имя и тогда собака будет безымянная – такого допустить мы не можем! Избежать подобной ошибки позволяет специальный метод (конструктор), который вызывается сразу в момент создания объекта заданного класса.

Сначала рассмотрим работу конструктора в общем виде:

Родилась новая собака!

Рассмотрим пример присвоения имени собаки через вызов конструктора класса:

```
class Dog():
    name=""
    # Конструктор
    # Вызывается на момент создания объекта этого типа
    def __init__ (self, newName):
        self.name = newName

# Создаем собаку и устанавливаем ее имя:
myDog = Dog ("Spot")

# Вывести имя собаки, убедиться, что оно было установлено
print (myDog.name)

# Следующая команда выдаст ошибку, потому что
# конструктору не было передано имя
# herDog = Dog()
```

Результат работы программы:

Spot

Теперь имя собаки присваивается в момент ее создания. В конструкторе указали self.name, т.к. момент вызова конструктора вместо self подставится конкретный объект, т.е. myDog.

В предыдущем примере для обращения к имени собаки мы выводили на экран поле myDog.name, т.е., переводя на язык реального мира, мы залезали во внутренности объекта и доставали оттуда информацию. Звучит жутковато, поэтому обеспечим «гуманные» методы для работы с именем объекта-собаки (setName и getName):

```
class Dog:
    name = ""

    # конструктор вызывается в момент создания объекта этого класса
    def __init__(self, newName):
        self.name = newName

    # можем в любой момент вызвать метод и изменить имя собаки
    def setName(self, newName):
        self.name = newName

    # можем в любой момент вызвать метод и узнать имя собаки
    def getName(self):
        return self.name  # возвращаем текущее имя объекта


# создаем собаку с начальным именем:
myDog = Dog("Spot")

# выводим имя собаки:
print(myDog.getName())

# установим новое имя собаки:
myDog.setName("Sharik")

# посмотрим изменения имени:
print(myDog.getName())
```

Проверим, что все работает:

Spot

Sharik

Для справки. Как скрыть атрибуты объектов.

В Python все атрибуты и методы являются публичными (открытыми), поэтому использование собственных методов get() и set() не защищает от непосредственного доступа к атрибутам. Для закрытия атрибутов в Python используется несколько способов, например функция property().

Напишем программу.

```
class Ddd:
    def __init__(self, input_name):
        self.hidden_name = input_name  # атрибут hidden_name

    def get_name(self):
        print('внутри getter')
        return self.hidden_name

    def set_name(self, input_name):
        print('внутри setter')
        self.hidden_name = input_name
```

```
    # методы get_name и set_name - свойства атрибута name
    # через name обращаемся к hidden_name
    name = property(get_name, set_name)


# создание объекта класса Ddd
f = Ddd('H')
# обращение к атрибуту name приведет к вызову метода get_name()
print(f.name)
print(f.get_name())
# установка значения атрибута name приведет к вызову метода set_name()
f.name = 'D'
print(f.name)
f.set_name('K')
print(f.name)
```

Результат работы программы:

внутри getter

H

внутри getter

H

внутри setter

внутри getter

D

внутри setter

внутри getter

K

Напишем программу с использованием декораторов без явного указания методов set и get:

```
class Ddd:
    def __init__(self, input_name):
        self.hidden_name = input_name

    @property
    def name(self):
        print('внутри getter')
        return self.hidden_name

    @name.setter
    def name(self, input_name):
        print('внутри setter')
        self.hidden_name = input_name


f = Ddd('H')
print(f.name)
f.name = 'D'
print(f.name)
```

Результат работы программы:

внутри getter

H

внутри setter
внутри getter
В Python существует связь свойств с вычисляемым значением:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def diameter(self):
        return 2 * self.radius


c = Circle(5)
print(c.radius)

# обращение к свойству diameter как к атрибуту
print(c.diameter)
c.radius = 7

# изменение атрибута radius привело к вычислению свойства diameter
print(c.diameter)

# приведет к ошибке, т.к. не указали метод set():
c.diameter = 20
```

Результат:
5
10
14
Traceback (most recent call last):
File "C:/Users/Admin/PycharmProjects/students/program.py", line 19, in <module>
c.diameter = 20
AttributeError: can't set attribute

Исправим, сделав атрибут __name «закрытым» (на самом деле он просто получит более длинное название _Ddd__name) с помощью двух нижних подчеркиваний:

```
class Ddd:
    def __init__(self, input_name):
        self.__name = input_name

    @property
    def name(self):
        print('внутри getter')
        return self.__name

    @name.setter
    def name(self, input_name):
        print('внутри setter')
        self.__name = input_name
```

```
f = Ddd('H')
print(f.name)
f.name = 'D'
print(f.name)
# print(f.__name)
print(f._Ddd__name)
```

Результат запуска программы:

внутри getter

H

внутри setter

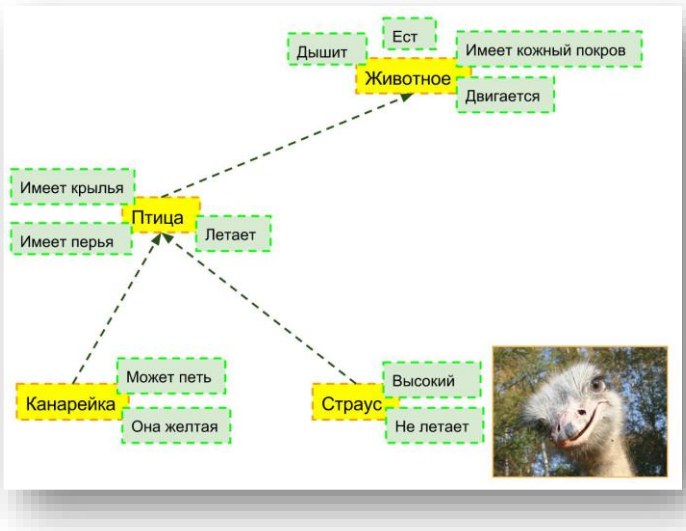
внутри getter

D

D

2.2 Наследование

Объектно-ориентированный подход в программировании тесно связан с мышлением человека, с работой его памяти. Для того чтобы нам лучше понять свойства ООП, рассмотрим модель хранения и извлечения информации из памяти человека (модель предложена учеными Коллинзом и Квиллианом). В своем эксперименте они использовали семантическую сеть, в которой были представлены знания о канарейке:

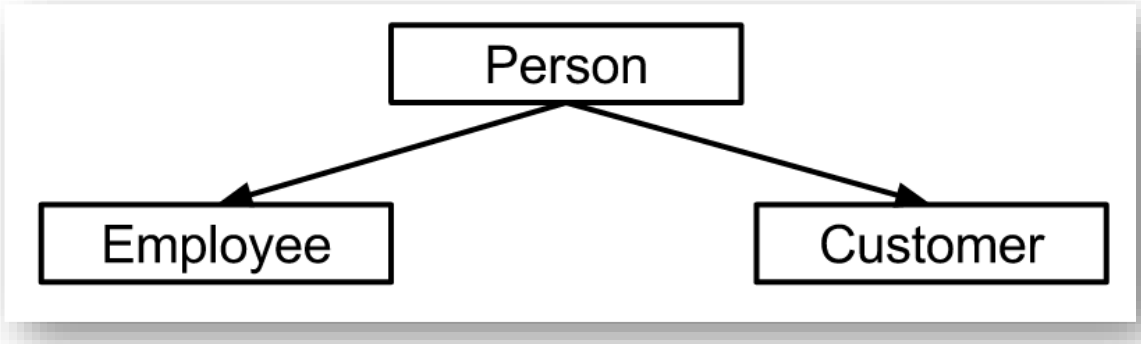


Например, «канарейка – это желтая птица, которая умеет петь», «птицы имеют перья и крылья, умеют летать» и т. п. Знания в этой сети представлены на различных уровнях: на нижнем уровне располагаются более частные знания, а на верхних – более общие. При таком подходе для понимания высказывания «Канарейка может летать» необходимо воспроизвести информацию о том, что канарейка относится к множеству птиц, и у птиц есть общее свойство «летать», которое распространяется (наследуется) и на канареек. Лабораторные эксперименты показали, что реакции людей на простые вопросы типа «Канарейка – это птица?», «Канарейка может летать?» или «Канарейка может петь?» различаются по времени. Ответ на вопрос «Может ли канарейка летать?» требует большего времени, чем на вопрос «Может ли канарейка петь». По мнению Коллинза и Квиллиана, это связано с

тем, что информация запоминается человеком на наиболее абстрактном уровне. Вместо того чтобы запоминать все свойства каждой птицы, люди запоминают только отличительные особенности, например, желтый цвет и умение петь у канареек, а все остальные свойства переносятся на более абстрактные уровни: канарейка как птица умеет летать и покрыта перьями; птицы, будучи животными, дышат и питаются и т. д. Действительно, ответ на вопрос «Может ли канарейка дышать?» требует большего времени, т. к. человеку необходимо проследовать по иерархии понятий в своей памяти. С другой стороны, конкретные свойства могут перекрывать более общие, что также требует меньшего времени на обработку информации. Например, вопрос «Может ли страус летать» требует меньшего времени для ответа, чем вопросы «Имеет ли страус крылья?» или «Может ли страус дышать?».

Упомянутое выше свойство наследования нашло свое отражение в объектно-ориентированном программировании.

К примеру, 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, представлен ниже:

```
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:

```
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"
```

В объектах классов Employee и Customer появилось поле name, унаследованное от класса Person.

Помимо полей базового класса происходит наследование методов:

```
class Person:
    name = ""

    def __init__(self):  # конструктор базового класса
        print("Создан человек")


class Employee(Person):
    job_title = ""


class Customer(Person):
    email = ""


johnSmith = Person()
janeEmployee = Employee()
bobCustomer = Customer()
```

Результат работы программы:

Создан человек

Создан человек

Создан человек

Таким образом, при создании объектов вызывается конструктор, унаследованный от базового класса. Если дочерние классы содержат собственные методы, то выполняться будут они:

```
class Person:
    name = ""

    def __init__(self):  # конструктор базового класса
        print("Создан человек")


class Employee(Person):
    job_title = ""

    def __init__(self):  # конструктор дочернего класса
        print("Создан работник")


class Customer(Person):
    email = ""

    def __init__(self):  # конструктор дочернего класса
        print("Создан покупатель")


johnSmith = Person()
```

```
janeEmployee = Employee()
bobCustomer = Customer()
```

Результат работы программы:

Создан человек

Создан работник

Создан покупатель

Видим, что в момент создания объекта вызывается конструктор, содержащийся в дочернем классе, т.е. конструктор дочернего класса переопределил конструктор базового класса.

Порой требуется вызвать конструктор базового класса из конструктора дочернего класса:

```
class Person:
    name = ""

    def __init__(self):
        print("Создан человек")


class Employee(Person):
    job_title = ""

    def __init__(self):
        Person.__init__(self)  # вызываем конструктор базового класса
        print("Создан работник")


class Customer(Person):
    email = ""

    def __init__(self):
        Person.__init__(self)  # вызываем конструктор базового класса
        print("Создан покупатель")


johnSmith = Person()
janeEmployee = Employee()
bobCustomer = Customer()
```

Результат работы программы:

Создан человек

Создан человек

Создан работник

Создан человек

Создан покупатель

3 Порядок выполнения работы

Получить задание для выполнения лабораторной работы (раздел 4) согласно своему варианту (по журналу). Разработать программу.

4 Задания для выполнения работы

Разработайте UML-диаграмму и программу.

Напишите иерархию геометрических фигур с родительским классом ***Shape*** и 3 дочерними классами С1, С2 и С3. Базовый класс содержит координаты одной точки для вывода фигур на экран и методы **square()** и **perimeter()**, которые подсчитывают площадь и периметр фигуры соответственно, **move()**, **fill()** – для сдвига и заливки фигуры цветом, **compare(x, y)** – для сравнения фигур x и y по площади, **is_intersect(x, y)** – для определения факта пересечения фигур x и y, **is_include(x, y)** – определяет факт включения y в x. Дочерние классы (отрезок, треугольник, четырехугольник, параллелограмм, трапеция, прямоугольник, ромб, квадрат, правильный пятиугольник) должны содержать параметры фигуры, по которым их можно нарисовать и рассчитать площадь и периметр. Создайте список из 15 объектов дочерних классов. Здесь допускаются одинаковые объекты. Выведите на экран величины площади, периметра, цвета заливки для всех объектов, попарно сравните фигуры по площади, определите факты пересечения фигур и включения. ***В список вносите программно только уникальные объекты.***

Задание	С1	С2	С3
1	ромб	квадрат	правильный пятиугольник
2	параллелограмм	прямоугольник	ромб
3	параллелограмм	квадрат	правильный пятиугольник
4	трапеция	квадрат	правильный пятиугольник
5	трапеция	прямоугольник	квадрат
6	трапеция	прямоугольник	ромб
7	четырехугольник	трапеция	правильный пятиугольник
8	отрезок	параллелограмм	трапеция
9	прямоугольник	квадрат	правильный пятиугольник
10	четырехугольник	прямоугольник	ромб
11	прямоугольник	ромб	квадрат
12	параллелограмм	прямоугольник	квадрат
13	четырехугольник	ромб	правильный пятиугольник
14	четырехугольник	прямоугольник	правильный пятиугольник
15	треугольник	четырехугольник	ромб
16	прямоугольник	ромб	правильный пятиугольник

17	треугольник	квадрат	правильный пятиугольник
18	параллелограмм	трапеция	правильный пятиугольник
19	параллелограмм	ромб	правильный пятиугольник
20	параллелограмм	ромб	квадрат
21	треугольник	трапеция	прямоугольник
22	трапеция	ромб	квадрат
23	четырехугольник	квадрат	правильный пятиугольник
24	отрезок	параллелограмм	ромб
25	трапеция	прямоугольник	правильный пятиугольник