МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО» (СГУ)

УТВЕРЖДАЮ
Зав.кафедрой,
к. ф.-м. н.
_____ С. В. Миронов

ОТЧЕТ О ПРАКТИКЕ

Студентки 3 курса 351 группы факультета КНиИТ
Ахмановой Элины Дамировны

вид практики: учебная
кафедра: математической кибернетики и компьютерных наук
курс: 3
семестр: 5
продолжительность: 2 нед., с 11.01.2019 г. по 24.01.2019 г.

Руководитель практики от университета,

к. ф.-м. н., доцент _____ А. С. Иванова

Руководитель практики от организации (учреждения, предприятия),

к. ф.-м. н., доцент _____ А. С. Иванова

Тема практики:«ООП в языке Питон»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методология объектно-ориентированного программирования (далее - ООП) в языке Python изучается во время прохождения учебных курсов курсов, например: языки программирования, методология программирования, программная инженерия. Данная практическая работа направлена на разработку методических рекомендаций по теме «ООП в языке Python» в рамках изучения дисциплины языки программирования. Работа предназначена для студентов, уже имеющих представление о базовых конструкциях Python, и желающих научиться применять принципы ООП. Кроме того, разработка методических рекомендаций полезна и для самого автора, потому что позволяет дополнить и уточнить уже сложившуюся систему знаний. Цель ознакомительной практики - приобретение навыков методической работы. Задачи:

— описать структуру классов, используемых в языке Python;
— описать основные операции по работе с классами, привести примеры;
— разработать по теме задачи и тест, содержащий не менее 10 вопросов.

1 Теоретический материал

1.1 Структура классов в языке Python

Объектно-ориентированное программирование (далее - ООП) - один из наиболее эффективных подходов к написанию программ [1]. Когда вы создаёте класс, вы описываете общее поведение, которое будет иметь определённая категория объектов [2]. В Python при работе с ООП используется два типа сущностей:

— класс, атрибуты которого определяют поведение;

— экземпляр, атрибуты которого хранят данные [3].

В классах по умолчанию:

1. Выражение class создаёт объект типа класс и даёт ему имя.
2. Атрибуты внутри класса наследуют состояние и поведение класса.
3. Вложенные методы класса со специальным аргументом на первом месте устанавливают предполагаемое состояние атрибутов и классов, в том числе [4].

Экземпляры, созданные из классов:

1. Вызывают класс так же, как функция создаёт свой новый экземпляр.
2. Каждый экземпляр наследует от класса атрибуты и своё собственное пространство имён [4].

В следующем пункте будут рассмотрены все основные операции для работы с классами. Для того, чтобы разобрать пример использования ООП в Python наглядно, забежим вперёд и изучим метод $\_\_init\_\_$. Он нужен для инициализации экземпляра класса и получает на вход необходимые атрибуты.

Давайте реализуем простой класс:

```
class Lobster():
    # Попытка создать модель омара

    def __init__(self, name, age):
        # Инициализируем атрибуты имени и возраста
        self.name = name
        self.age = age

    def click_tick(self):
```

```
        # Сэмулируем щелканье клешнями
        print(self.name.title() + "is now clicking.")

    def backslide(self):
        # Теперь сэмулируем то, как омар пятится
        print(self.name.title() + "backs away!")
```

В нашем классе есть два атрибута: имя и возраст. Значения для них передаются в виде аргументов функции $\_\_init\_\_$. Также у нас есть ещё два метода. Каждый из них выводит строку с именем и действием, которое соответствует вызванному методу.

Добавим, что с точки зрения программирования, классы в языке программирования Python [5] являются компонентами, такими же, как функции и модули: все они упаковывают какие-то данные для последующего удобного использования.

Класс определяет новое пространство имён, в точности как модуль [2]. Однако по сравнению с другими программными единицами, классы имеют три главных отличия, которые делают их крайне полезными при создании новых объектов:

1. Множество экземпляров. Классы, по сути, являются заводами по производству объектов. Каждый раз, когда мы вызываем какой-либо класс, мы создаём новый объект с определённым пространством имён.
2. Изменение через наследование [2]. Мы можем переопределить один класс, расширив его атрибуты в другом, подклассе, и получим иерархическую систему классов. Это понятие схоже с биологической теорией видов, проводя грань с которой мы можем вспомнить, что у всех позвоночных есть позвоночник, но от вида к виду он различается. Или все млекопитающие кормят своих детёнышей молоком, но кенгуру и собака делают это разными способами.
3. Перегрузка операторов позволяет переопределить уже существующие методы для нового класса (конкатенация, получение элемента по индексу и др.).
4. Наследование классов и полиморфизм (примеры указаны в Приложении 1).

Таким образом, объектно-ориентированных подход позволяет создавать

более сложные приложения по-сравнению с процедурным.

1.2 Специальные методы

Объекты классов и специальные методы

Объект-класс создается с помощью определения класса. Объекты-классы имеют следующие атрибуты:

$\_\_name\_\_$ — имя класса;

$\_\_module\_\_$ — имя модуля;

$\_\_dict\_\_$ — словарь атрибутов класса, можно изменять этот словарь напрямую;

$\_\_bases\_\_$ — кортеж базовых классов в порядке их следования;

$\_\_doc\_\_$ — строка документации класса.

Экземпляры классов и специальные методы

Экземпляр (инстанс) класса возвращается при вызове объекта-класса. Объект у класса может быть один, экземпляров (или инстансов) — несколько. Экземпляры имеют следующие атрибуты:

$\_\_dict\_\_$ — словарь атрибутов класса, можно изменять этот словарь напрямую;

$\_\_class\_\_$ — объект-класс, экземпляром которого является данный инстанс;

$\_\_init\_\_$ — конструктор. Если в базовом классе есть конструктор, конструктор производного класса должен вызвать его;

$\_\_del\_\_$ — деструктор. Если в базовом классе есть деструкор, деструктор производного класса должен вызвать его;

$\_\_cmp\_\_$ — вызывается для всех операций сравнения;

$\_\_hash\_\_$ — возвращает хеш-значение объекта, равное 32-битному числу;

$\_\_getattr\_\_$ — возвращает атрибут, недоступный обычным способом;

$\_\_setattr\_\_$ — присваивает значение атрибуту;

$\_\_delattr\_\_$ — удаляет атрибут;

$\_\_call\_\_$ — срабатывает при вызове экземпляра класса.

Экземпляры классов в качестве последовательностей

Экземпляры классов можно использовать для эмуляции последовательностей. Для такой реализации есть встроенные методы:

$\_\_len\_\_$ — возвращает длину последовательности;

__getitem__ — получение элемента по индексу или ключу;

__setitem__ — присваивание элемента с данным ключом или индексом;

__delitem__ — удаление элемента с данным ключом или индексом;

__getslice__ — возвращает вложенную последовательность;

__setslice__ — заменяет вложенную последовательность;

__delslice__ — удаляет вложенную последовательность;

__contains__ — реализует оператор in.

Приведение объектов к базовым типам

Объекты классов можно привести к строковому или числовому типу.

__repr__ — возвращает формальное строковое представление объекта;

__str__ — возвращает строковое представление объекта;

__oct__ , *__hex__* , *__complex__* , *__int__* , *__long__* , *__float__* — возвращают строковое представление в соответствующей системе счисления.

Bound и unbound методы

Рассмотрим конкретный пример. Есть базовый класс Cat, и есть производный от него класс Barsik:

```
class Cat:
  def __init__(self):
      self.hungry = True
  def eat(self):
      if self.hungry:
          print 'I am hungry...'
          self.hungry = False
      else:
          print 'No, thanks!'

class Barsik(Cat):
  def __init__(self):
      self.sound = 'Aaaammm!'
      print self.sound
```

Создаем экземпляр производного класса:

```
>>> brs = Barsik()
Aaaammm!
>>> brs.eat()
AttributeError: Barsik instance has no attribute 'hungry'
```

На первый взгляд — странная ошибка, поскольку атрибут hungry есть в базовом классе. На самом деле, конструктор производного класса — перегруженный, при этом конструктор базового класса не вызывается, и его нужно явно вызвать. Это можно сделать двумя путями. Первый вариант считается устаревшим:

```
class Barsik(Cat):
  def __init__(self):
      Cat.__init__(self)
      self.sound = 'Aaaammm!'
      print self.sound
```

Здесь мы напрямую вызываем конструктор базового класса, не создавая инстанс базового класса Cat — поэтому такой базовый конструктор относится к категории unbound-методов, в пику методам, которые вызываются для инстансов классов и называются bound-методами. Для вызова bound-метода в качестве первого параметра методу нужно передать инстанс класса.

Метод super

Второй вариант: в начале программы нужно определить метакласс, который указывает на то, что класс реализован в так называемом новом стиле — new-style. Затем нужно вызвать стандартный метод super для базового конструктора:

```
__metaclass__ = type
...
class Barsik(Cat):
    def __init__(self):
      super(Barsik, self).__init__()
      self.sound = 'Aaaammm!'
      print self.sound

>>> brs = Barsik()
>>> brs.eat()
Aaaammm!
I am hangry...
```

Статические методы

Статический метод — функция, определенная вне класса и не имеющая атрибута self:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

def printNumInstances( ):
    print "Number of instances created: ", Spam.numInstances
```

```
>>> a=Spam()
>>> b=Spam()
>>> printNumInstances()
Number of instances created:  2
```

Статический метод может быть определен и внутри класса — для этого используется ключевое слово staticmethod, причем метод может быть вызван как статически, так и через инстанс:

```
class Multi:
    def imeth(self, x):
        print self, x
    def smeth(x):
        print x
    def cmeth(cls, x):
        print cls, x
    smeth = staticmethod(smeth)
    cmeth = classmethod(cmeth)

>>> Multi.smeth(3)
3
>>> obj=Multi()
>>> obj.smeth(5)
5
```

Методы класса определяются с помощью ключевого слова classmethod — здесь автоматически питон передает в качестве первого параметра сам класс (cls):

```
>>> Multi.cmeth(7)
__main__.Multi 7
>>> obj.cmeth(10)
__main__.Multi 10
```

Итератор

Итераторы хороши там, где списки не подходят в силу того, что занимают много памяти, а итератор возвращает его конкретное значение. В

классе нужно определить два стандартных метода — $__iter__$ и next. Метод $__iter__$ будет возвращать объект через метод next:

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('12345'):
>>>   print char
```

Итератор можно сконвертировать в список:

```
>>> rvr = list(Reverse('12345'))
>>> rvr
['5', '4', '3', '2', '1']
```

Property

Property — атрибут класса, возвращаемый через стандартную функцию property, которая в качестве аргументов принимает другие функции класса:

```
class DateOffset:
    def __init__(self):
        self.start = 0

    def _get_offset(self):
        self.start +=  5
        return self.start

    offset = property(_get_offset)
```

```
>>> d = DateOffset()
>>> d.offset
5
>>> d.offset
10
```

Singleton

Данный паттерн позволяет создать всего один инстанс для класса. Используется метод $__new__$:

```
class Singleton(object):
  def __new__(cls, *args, **kw):
      if not hasattr(cls, '_instance'):
        orig = super(Singleton, cls)
        cls._instance = orig.__new__(cls, *args, **kw)
      return cls._instance


>>> one = Singleton()
>>> two = Singleton()
>>> id(one)
3082687532
>>> id(two)
3082687532
```

Слоты

Слоты — это список атрибутов, задаваемый в заголовке класса с помощью $__slots__$. В инстансе необходимо назначить атрибут, прежде чем пользоваться им:

```
class limiter(object):
    __slots__ = ['age', 'name', 'job']

>>> x=limiter()
>>> x.age = 20
```

Функтор

Функтор — это класс, имеющий метод $__call__$ — при этом объект можно вызвать как функцию.

Пусть у нас имеется класс Person, имеется коллекция объектов этого класса- people, нужно отсортировать эту коллекцию по фамилиям. Для этого можно использовать функтор Sortkey:

```
class SortKey:
  def __init__(self, *attribute_names):
      self.attribute_names = attribute_names

  def __call__(self, instance):
      values = []
      for attribute_name in self.attribute_names:
          values.append(getattr(instance, attribute_name))
      return values

class Person:
  def __init__(self, forename, surname, email):
      self.forename = forename
      self.surname = surname
      self.email = email

>>> people=[]
>>> p=Person('Petrov','','')
>>> people.append(p)
>>> p=Person('Sidorov','','')
>>> people.append(p)
>>> p=Person(u'Ivanov','','')
>>> people.append(p)
>>> for p in people:
...   print p.forename
Petrov
Sidorov
Ivanov
```

```
>>> people.sort(key=SortKey("forename"))
>>> for p in people:
...   print p.forename
Ivanov
Petrov
Sidorov
```

Дескриптор

Дескриптор — это класс, который хранит и контролирует атрибуты других классов. Вообще любой класс, который имплементирует один из специальных методов — $\_\_get\_\_$, $\_\_set\_\_$, $\_\_delete\_\_$, является дескриптором.

Пример:

```
class ExternalStorage:
  __slots__ = ("attribute_name",)
  __storage = {}

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

  def __set__(self, instance, value):
      self.__storage[id(instance), self.attribute_name] = value

  def __get__(self, instance, owner=None):
      if instance is None:
          return self
      return self.__storage[id(instance), self.attribute_name]

class Point:
  __slots__ = ()
  x = ExternalStorage("x")
  y = ExternalStorage("y")
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

```
>>> p1=Point(1,2)
>>> p2=Point(3,4)
```

В данном случае класс Point не имеет собственных атрибутов x, y, хотя вызывает их так, как будто они есть — на самом деле они хранятся в дескрипторе ExternalStorage.

Sequence

Последовательность реализуется с помощью таких методов: $\_\_getitem\_\_$, $\_\_setitem\_\_$. В данном примере класс MySequence возвращает по индексу элемент последовательности неопределенной длины, представляющей собой арифметическую прогрессию вида: 1 3 5 7 ... Здесь нельзя применить стандартные методы $\_\_del\_\_$, $\_\_len\_\_$:

```
class MySequence:
  def __init__(self, start=0, step=1):
      self.start = start
      self.step = step
      self.changed = {}
  def __getitem__(self, key):
      return self.start + key*self.step
  def __setitem__(self, key, value):
      self.changed[key] = value

>>> s = MySequence(1,2)
>>> s[0]
1
>>> s[1]
3
>>> s[100]
201
```

3 Практические задания

1. Создайте класс, который называется Thing, не имеющий содержимого, и выве-дите его на экран. Затем создайте объект example этого класса и также выведите его. Совпадают ли выведенные значения?
2. Создайте новый класс с именем Thing2 и присвойте его атрибуту letters значе-ние 'abc'. Выведите на экран значение атрибута letters.
3. Создайте еще один класс, который, конечно же, называется Thing3. В этот раз присвойте значение 'xyz' атрибуту объекта, который называется letters. Выведите на экран значение атрибута letters. Понадобилось ли вам создавать объект класса, чтобы сделать это?
4. Создайте класс, который называется Element, имеющий атрибуты объекта name, symbol и number. Создайте объект этого класса со значениями 'Hydrogen', 'H' и 1.
5. Создайте словарь со следующими ключами и значениями: 'name': 'Hydrogen', 'symbol': 'H', 'number': 1. Далее создайте объект с именем hydrogen класса Element с помощью этого словаря.
6. Для класса Element определите метод с именем dump(), который выводит на экран значения атрибутов объекта (name, symbol и number). Создайте объект hydrogen из этого нового определения и используйте метод dump(), чтобы вывести на экран его атрибуты.
7. Вызовите функцию print(hydrogen). В определении класса Element измените имя метода dump на $\_\_str\_\_$, создайте новый объект hydrogen и затем снова вызовите метод print(hydrogen).
8. Модифицируйте класс Element, сделав атрибуты name, symbol и number закрытыми. Определите для каждого атрибута свойство получателя, возвращающее значение соответствующего атрибута.
9. Определите три класса: Bear, Rabbit и Octothorpe. Для каждого из них определи-те всего один метод — eats(). Он должен возвращать значения 'berries' (для Bear), 'clover' (для Rabbit) или 'campers' (для Octothorpe). Создайте по одному объекту каждого класса и выведите на экран то, что ест указанное животное.
10. Определите три класса: Laser, Claw и SmartPhone. Каждый из них имеет только один метод — does(). Он возвращает значения 'disintegrate' (для Laser), 'crush' (для Claw) или 'ring' (для SmartPhone). Далее опре-

делите класс Robot, который содержит по одному объекту каждого из этих классов. Определите метод does() для класса Robot, который выводит на экран все, что делают его компоненты [6].

4 Проверочный тест

1. Какие базовые типы сущностей ООП вы знаете?

   а) класс

   б) инкапсуляция

   в) инкапсуляция и экземпляр

   г) класс и экземпляр

   Ответ: г.

2. Какой метод инициализирует экземпляр класса без параметров для атрибутов?

   а) $\_\_init\_\_(initial)$

   б) $\_\_class\_\_()$

   в) $initial(class)$

   г) $\_\_init\_\_(self)$

   Ответ: г.

3. Дополните: “Атрибуты внутри класса ... состояние и поведение класса.”

   а) скрывают

   б) наследуют

   в) инициализируют

   Ответ: в.

4. Мы создаём новый объект с определённым пространством имён, когда ... .

   а) вызываем родительский класс

   б) создаём любой класс

   в) создаём экземпляр

   Ответ: б.

5. Может ли использоваться родительский метод в дочернем классе?

   а) да

   б) нет

   в) только если он определён и в дочернем классе

   Ответ: а.

6. Как классы связаны с модулями?

   Ответ:

Классы всегда находятся внутри модулей – они являются атрибутами объекта модуля. Классы и модули являются пространствами имен, но классы соответствуют инструкциям (а не целым файлам) и поддерживают такие понятия ООП, как экземпляры класса, наследование и перегрузка операторов. В некотором смысле модули напоминают классы с единственным экземпляром, без наследования, которые соответствуют целым файлам [7].

7. С помощью какой конструкции создаются родительские классы?

   а) class
   б) class < parent
   в) parent class
   г) init class

   Ответ: а.

8. Где и как создаются атрибуты классов?

   Ответ:

   Атрибуты класса создаются присваиванием атрибутам объекта класса. Обычно они создаются инструкциями верхнего уровня в инструкции class – каждое имя, которому будет присвоено значение внутри инструкции class, становится атрибутом объекта класса (с технической точки зрения область видимости инструкции class преобразуется в пространство имен атрибутов объекта класса). Атрибуты класса могут также создаваться через присваивание атрибутам класса в любом месте, где доступна ссылка на объект класса, то есть даже за пределами инструкции class [8].

9. Где и как создаются атрибуты экземпляров классов?

   Ответ:

   Атрибуты экземпляра создаются присваиванием атрибутам объекта экземпляра. Обычно они создаются внутри методов класса, в инструкции class – присваиванием значений атрибутам аргумента self (который всегда является подразумеваемым экземпляром). Повторюсь: возможно создавать атрибуты с помощью операции присваивания в любом месте программы, где доступна ссылка на экземпляр, даже за пределами инструкции class [9].

10. Согласно идее наследования:

а) Лучше вызывать метод суперкласса для выполнения действий по умолчанию, чем копировать и изменять его программный код в подклассе.

б) Лучше создать метод дочернего класса для выполнения действий, которые в нём не определены.

в) Лучше определить метод вне классов.

Ответ: а.

ЗАКЛЮЧЕНИЕ

В рамках выполнения данной практической работы были созданы материалы для изучения ООП на примере языка Python. По итогам практики были приобретены навыки методической работы.

Особое внимание было уделено практическим задачам и примерам концепций ООП. Была описана структура классов, используемых в Python, и их основные операции. Также были разработаны практические задания и тест для проверки результатов освоения темы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Shaw, Z. A.* Learn Python the hard way : a very simple introduction to the terrifyingly beautiful world of computers and code / Z. A. Shaw. — Crawfordsville, Indiana: RR Donnelley, 2013. — 288 pp.

2 *Lutz, M.* Learning Python / M. Lutz. — Beijing: O'Reilly Media, 2009. — 1214 pp.

3 Mark Lutz's Books, Software, Etc. [Электронный ресурс]. — URL: https://learning-python.com/ (Дата обращения 20.01.2019). Загл. с экр. Яз. англ.

4 *Lutz, M.* Python Pocket Reference: Python In Your Pocket / M. Lutz. — Beijing: O'Reilly Media, 2014. — 266 pp.

5 Python Documentation [Электронный ресурс]. — URL: https://www.python.org/ (Дата обращения 20.01.2019). Загл. с экр. Яз. англ.

6 *Любанович, Б.* Простой Python. Современный стиль программирования / Б. Любанович. — СПб.: Питер, 2016. — 480 с.

7 *Lubanovic, B.* Introducing Python: Modern Computing in Simple Packages / B. Lubanovic. — O'Reilly Media, 2014. — 460 pp.

8 Основные понятия объектно-ориентированного программирования [Электронный ресурс]. — URL: https://devpractice.ru/python-lesson-14-classes-and-objects/ (Дата обращения 20.01.2019). Загл. с экр. Яз. рус.

9 *Лутц, М.* Изучаем Python / М. Лутц. — Символ-Плюс, 2011. — 1280 с.

10 *Lee, M.* Programming language pragmatics / M. Lee. — Morgan Kaufmann, 2006. — 481 pp.

11 Course (Using Databases with Python) [Электронный ресурс]. — URL: https://www.coursera.org/learn/python-databases?specialization=python (Дата обращения 20.01.2019). Загл. с экр. Яз. англ.

ПРИЛОЖЕНИЕ А
Примеры задач

Наследование

Под наследованием понимается возможность создания нового класса на базе существующего. Наследование предполагает наличие отношения “является” между классом наследником и классом родителем. При этом класс потомок будет содержать те же атрибуты и методы, что и базовый класс, но при этом его можно (и нужно) расширять через добавление новых методов и атрибутов [10].

Примером базового класса, демонстрирующего наследование, можно определить класс “автомобиль”, имеющий атрибуты: масса, мощность двигателя, объем топливного бака и методы: завести и заглушить. У такого класса может быть потомок – “грузовой автомобиль”, он будет содержать те же атрибуты и методы, что и класс “автомобиль”, и дополнительные свойства: количество осей, мощность компрессора и т.п..

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка есть несколько родителей. Не все языки программирования поддерживают множественное наследование, но в Python можно его использовать.

Синтаксически создание класса с указанием его родителя/ей выглядит так:

```
class class_name(parent_1, [parent_2,.., parent_n])
```

Доработаем наш пример так, чтобы в нем присутствовало наследование.

```
class Figure:
    def __init__(self, color):
        self.color = color

    def get_color(self):
        return self.color


class Rectangle(Figure):
```

```
    def __init__(self, color, width=100, height=100):
        super().__init__(color)
        self.width = width
        self.height = height

    def square(self):
        return self.width*self.height


rect1 = Rectangle("blue")
print(rect1.get_color())
print(rect1.square())
rect2 = Rectangle("red", 25, 70)
print(rect2.get_color())
print(rect2.square())
```

Полиморфизм

Полиморфизм позволяет одинаково обращаться с объектами, имеющими однотипный интерфейс, независимо от внутренней реализации объекта. Например с объектом класса “грузовой автомобиль” можно производить те же операции, что и с объектом класса “автомобиль”, т.к. первый является наследником второго, при этом обратное утверждение неверно (во всяком случае не всегда). Другими словами полиморфизм предполагает разную реализацию методов с одинаковыми именами. Это очень полезно при наследовании, когда в классе наследнике можно переопределить методы класса родителя [7].

Как уже было сказано, полиморфизм, как правило, используется с позиции переопределения методов базового класса в классе наследнике. Проще всего это рассмотреть на примере. Добавим в наш базовый класс метод info(), который печатает сводную информацию по объекту класса Figure и переопределим этот метод в классе Rectangle, где добавим дополнительные данные и вывод [11].

```
class Figure:
    def __init__(self, color):
        self.color = color
```

```
    def get_color(self):
        return self.color

    def info(self):
        print("Figure")
        print("Color: " + self.color)


class Rectangle(Figure):
    def __init__(self, color, width=100, height=100):
        super().__init__(color)
        self.width = width
        self.height = height

    def square(self):
        return self.width * self.height

    def info(self):
        print("Rectangle")
        print("Color: " + self.color)
        print("Width: " + str(self.width))
        print("Height: " + str(self.height))
        print("Square: " + str(self.square()))

fig1 = Figure("green")
print(fig1.info())
rect1 = Rectangle("red", 24, 45)
print(rect1.info())
```

Таким образом наследник класса может расширять и модифицировать функционал класса родителя [8].