Объектно-ориентированное программирование

Цель занятия

После освоения темы вы:

- узнаете понятия классов и объектов в Python и их взаимосвязь;
- узнаете особенности объектно-ориентированной модели в Python;
- сможет создавать классы и использовать методы классов;
- сможете определить и спроектировать необходимые классы и методы классов для конкретной предметной области;
- узнаете понятия и механизмы наследования;
- сможете проектировать классы с использованием механизма наследования.

План занятия

1. Введение в объектно-ориентированное программирование
2. Классы и экземпляры классов
3. Методы. Пример рефакторинга программы на ООП
4. Инкапсуляция. Полиморфизм
5. Наследование классов
6. Особенности объектной модели в Python
7. Элементы статической типизации. Абстрактные классы и протоколы
8. Множественное наследование
9. Проблемы, связанные с наследованием

Используемые термины

Инкапсуляция — разделение интерфейса объекта и его внутренней реализации.

Полиморфизм — способность создания единого интерфейса с различными реализациями.

Наследование — механизм переиспользования функциональности базового класса в классе-наследнике.

Конспект занятия

1. Введение в объектно-ориентированное программирование

Рассмотрим основные идеи, которые лежат в основе объектно-ориентированного программирования. Прежде всего зададимся вопросом, зачем нужны новые парадигмы разработки.

Гради Буч в известной книге «Объектно-ориентированный анализ и проектирование» указывает: «Сложность промышленных программ превышает интеллектуальные возможности отдельного человека. Увы, это свойство является существенной характеристикой всех крупных систем программного обеспечения».

Действительно, отдельный человек не может удержать в своем сознании всю сложность той программы, с которой он работает.

Рост сложности программ заставляет придумывать новые подходы к разработке.

Главный ключ построению сложной системы — введение абстракции.

Рассмотрим на примере устройства компьютера, как введение абстракции помогает решать задачи (рисунок 1).

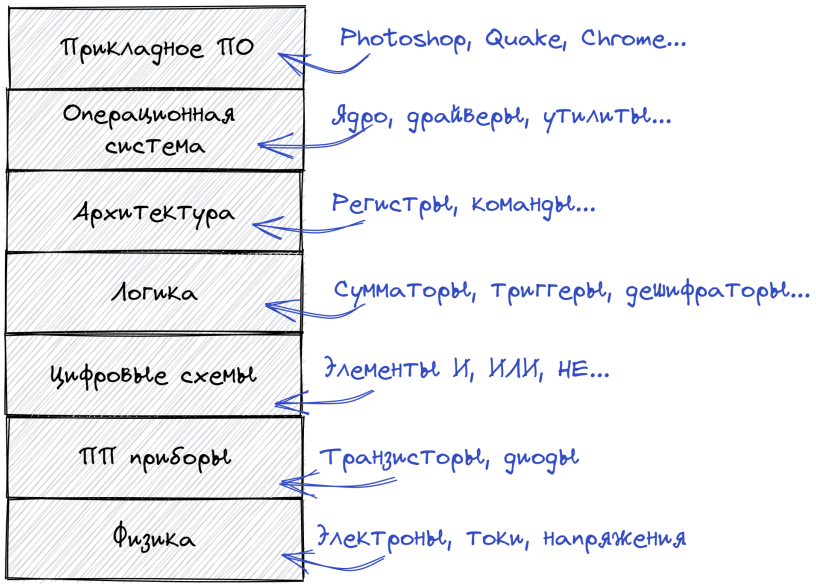

Рисунок 1. Пример сложной абстракции

На самом низком физическом уровне компьютер представляет собой электрическую схему, в которой есть электроны, напряжение, токи. Но работая на компьютере, например, делая презентацию, никто не будет думать о происходящем в этих терминах. Мы поднялись на более высокий уровень.

Но даже при проектировании компьютера мы должны абстрагироваться от уровня физики и ввести некоторые готовые приборы, которые имеют известные характеристики, и которые мы рассматриваем как готовые элементы: транзисторы, диоды.

Из этих элементов мы собираем цифровые схемы — логические элементы И, ИЛИ, НЕ. Из логических элементов мы собираем сумматоры, триггеры, дешифраторы. Далее собираем процессор с помощью какой-либо архитектуры. В процессоре будут регистры, команды. Потом на процессоре строится операционная система, и на операционной системе уже работает прикладное программное обеспечение.

Такое постепенное наращивание уровня абстракции позволяет строить очень сложные системы, которые невозможно понять, если мыслить только на физическом уровне.

Первая парадигма, которая преодолевает растущую сложность разработки — парадигма структурного программирования.

Идея парадигмы структурного программирования — декомпозиция большой задачи на меньшие подзадачи, реализуемые отдельными подпрограммами. К каждой из подзадач можно повторно применить прием декомпозиции. Так будет получена некоторая иерархическая структура.

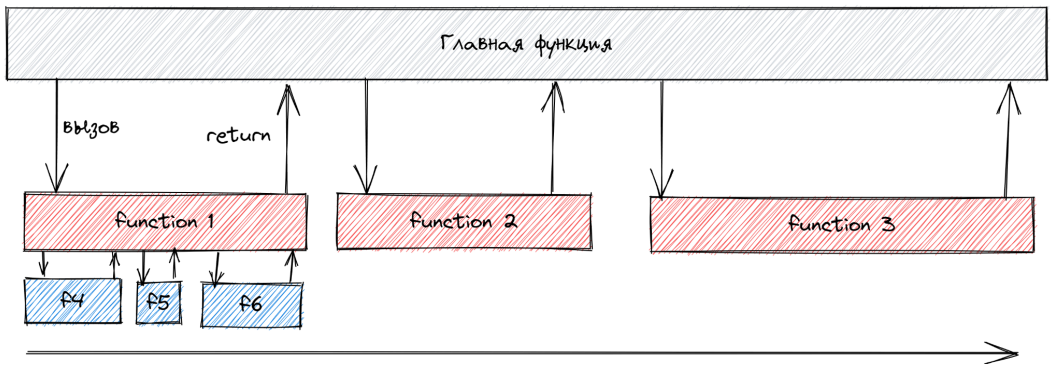

Рисунок 2. Представление программы в структурной парадигме

В структуре есть главная функция, выполняющая ту работу, которую должна выполнять программа. Главная функция вызывает более простые и маленькие функции, каждая из них вызывает более мелкие функции и так далее. Мы можем продолжать процесс, пока не дойдем до настолько простых задач, которые можем просто взять и решить.

При этом когда мы мыслим о программе в целом, то не мыслим о ней отдельными маленькими действиями, а мыслим отдельными большими блоками.

Проблема структурного программирования в том, что задача, которую мы формулируем и решаем — это задача реального мира. А в реальном мире нет ни байтов, ни алгоритмов. В реальном мире есть сущности, с которыми работаем.

Идея объектно-ориентированного программирования (ООП) — попытаться мыслить о программе в терминах реального мира. То есть смоделировать предметную область решаемой задачи в виде программных объектов, с которыми непосредственно работает программист.

ООП — особый способ представления программы. В нем программа рассматривается как набор объектов — отдельных сущностей с внутренней структурой и определенным поведением. Объекты в программе — модели объектов, с которыми мы работаем в реальной жизни.

Такой подход позволяет сформулировать задачу, решаемую программой, в почти таких же терминах, как и задача в реальном мире.

Программа в объектно-ориентированной парадигме представлена как набор взаимодействующих объектов.

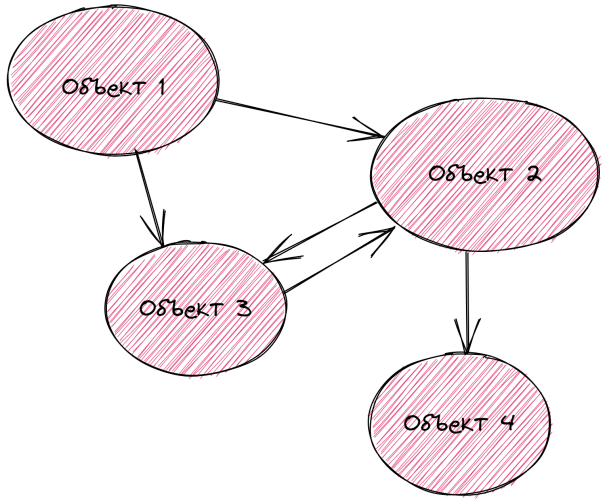

Рисунок 3. Представление программы в парадигме ООП

Сами объекты могут быть достаточно сложными, состоять из других объектов в качестве своих элементов. Объекты можно детализировать до тех пор, пока представление задачи не станет достаточно понятным и простым для ее решения.

Рассмотрим пример задачи построения системы умного дома. Необходимо регулировать интенсивность отопления в разных помещениях в зависимости от текущей температуры и времени суток.

Классический процедурный подход заставляет нас переформулировать эту задачу в терминах конкретных операций:

- считать показания с датчика температуры, и для этого отправить запрос определенного формата на определенный адрес;

- считать время с системного таймера;
- определить на основе этих данных требуемую интенсивность отопления;
- передать сигнал на контроллер отопления, а для этого опять же отправить запрос определенного формата на определенный адрес;
- повторять эту процедуру для датчиков, размещенных в разных помещениях, с определенной периодичностью.

Нетрудно заметить достаточно большой разрыв между формулировкой задачи в терминах реального мира и формулировкой алгоритма ее решения.

Объектно-ориентированный подход позволяет сократить этот разрыв:

1. Сама программа проектируется как система объектов, «копирующая» объекты реального мира. Конечно, с учетом определенного уровня абстракции.

2. В программе есть объект «комната», у которого есть атрибут «температура». Технически он точно так же считывается с того же самого датчика.

3. В программе есть метод изменения интенсивности отопления. И опять же, технически он выполняет тот же самый запрос на контроллер.

4. Программа всего лишь тривиально повторяет формулировку самой задачи, только не на русском языке, а на каком-либо языке программирования.

Использование ООП не является обязательным. Любую задачу программирования можно решить и без ООП, но использование данной парадигмы зачастую позволяет сделать это удобнее и эффективнее. Под эффективностью здесь понимается эффективность работы программиста.

ООП дает ощутимую выгоду только для достаточно крупных задач. Если задача решается несколькими строками кода, введение объектов только усложнит ее.

2. Классы и экземпляры классов

Объектно-ориентированная парадигма глубоко вшита в Python. В Python абсолютно все является объектом, все переменные и функции — это объекты.

Объект — это некоторые данные и определенный интерфейс для работы с ними. Объект в программе является моделью объекта из реального мира (с учетом определенного уровня абстракции).

В качестве объекта можно рассмотреть любые данные из языка Python. Например, когда мы работаем со строкой, мы просто работаем с текстом. И не задумываемся, как он представлен в памяти, как реализованы те или иные методы.

```
s = 'Amat victoria curam'
dir(s)  # список методов
```

Функция dir() возвращает список всех методов, которые есть у объекта.

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__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',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Методы объекта — это функции, которые имеют доступ к данным внутри объекта. Например, методы строки работают со строкой, для которой их вызывают, и возвращают некоторую информацию о строке.

Таблица 1. Примеры методов работы со строкой

Вызываемый метод	**Назначение метода**	**Вывод**
`s.isascii()`	Проверка, состоит ли строка из символов кодировки ASCII	`True`
`s.upper()`	Создает копию строки, записанную в верхнем регистре.	`'AMAT VICTORIA CURAM'`
`s.split()`	Создает из строки список слов	`['Amat', 'victoria', 'curam']`

Класс — это тип объекта. Класс содержит описание формата данных и функций для работы с ними. Объект определенного класса называют «**экземпляром класса**».

Чтобы узнать, к какому классу принадлежит объект, можно воспользоваться функцией type():

```
type(s)                                  # <class 'str'>
```

Все строки являются экземплярами одного и того же класса, поэтому:

```
type(s) == type('Hello world!')             # True
```

Далее рассмотрим, как создать собственный класс. Используется ключевое слово `class`, далее указывается имя класса и ставится двоеточие. В блоке с отступом следует описание класса. В целом даже пустого описания класса достаточно, чтобы класс заработал, и мы могли создавать экземпляры созданного класса:

```
class MyClass:
    # В блоке с отступом следует описание класса
    pass
```

Создание экземпляра класса выглядит как вызов функции:

```
x = MyClass()
```

Каждый экземпляр хранит информацию о своем классе в специальном атрибуте `'__class__'`, также его можно получить с помощью функции `type()`.

Примеры создания экземпляра класса и получения информации о его типе:

```
x = MyClass()
x.__class__           # <class __main__.MyClass>
type(x)               # <class __main__.MyClass>
```

Атрибут — это данные, размещенные «внутри» какого-либо объекта, и доступные по определенному имени. Атрибут можно воспринимать как локальную переменную. Но также объект можно воспринимать как особый вариант словаря, а атрибут — как запись в этом словаре.

Рассмотрим пример, как добавить атрибут к объекту. К созданному объекту класса `MyClass` нужно добавить точку и далее имя атрибута. После чего можно присвоить атрибуту значение. Действие выполняется аналогично созданию переменных, отличие — переменная принадлежит объекту. Синтаксически это выражается с

помощью точки. После создания атрибутов к ним можно получить доступ так же, как и к обычным переменным:

```
x.name = 'Bilbo Baggins'
x.species = 'hobbit'
print(x.name, 'is a', x.species)           # Bilbo Baggins is a
hobbit
```

Важно! Атрибут создается только у одного экземпляра, никак не задействуя другие.

Если мы создадим два экземпляра класса `MyClass` и у одного из них атрибут `name`, то у одного экземпляра атрибут появится, а у другого — нет:

```
x = MyClass()
y = MyClass()
x.name = 'Bilbo'
x.name                  # 'Bilbo'
y.name                  # Traceback (most recent call last):
                        # File "<stdin>", line 1, in <module>
                        # AttributeError: 'MyClass' object has
                        # no attribute 'name'
```

На практике почти никогда не добавляют новые атрибуты объектам уже после их создания.

3. Методы. Пример рефакторинга программы на ООП

Метод — это почти обычная функция, но с двумя особенностями:

1. Метод описывается внутри класса.
2. Методу при вызове неявно передается в качестве первого аргумента тот экземпляр класса, с которым он должен работать.

Пункт 2 технически реализуется через неявную передачу объекта в качестве первого аргумента данной функции.

В любом классе вам скорее всего понадобится хотя бы один специальный метод, который называется конструктор. **Конструктор** — особый метод, который вызывается автоматически при создании нового экземпляра класса.

Специальные методы имеют особую форму записи: они имеют два знака подчеркивания в начале и в конце. Метод конструктора должен иметь название

`__init__`. Данное название принципиально, если вы назовете конструктор иначе, он не будет вызван автоматически.

Рассмотрим пример простейшего конструктора. В примере нужно обратить внимание на отступы. Конструктор находится внутри класса, поэтому он записан с отступом. Само тело метода имеет дополнительный отступ:

```
class Box:
    def __init__(self):
        self.content = []
```

Стоит обратить внимание на аргумент `self` — это тот самый объект, с которым работает метод. Название `self` является соглашением, если вы назовете этот аргумент по-другому, все по-прежнему будет работать. На практике делать так не следует. Однако, стоит иметь в виду, что это может быть причиной ошибок. Если объявляя метод в классе, вы забыли указать первым аргументом `self`, то первый записанный аргумент на самом деле будет содержать ссылку на тот объект, с которым вы работаете.

Дальнейшее создание двух разных объектов автоматически вызывает метод `__init__`, но при этом в качестве `self` передаются два разных объекта. Следует иметь в виду, что создание объекта (выделение под него памяти и т. п.) происходит без нашего участия, конструктор получает уже созданный объект для его дальнейшей «настройки». Атрибуты объекта создаются в конструкторе.

В примере в конструкторе создается атрибут `content`, содержащий некоторый список, в который можно записывать разные значения:

```
x = Box()  # автоматически вызывается метод Box.__init__
y = Box()  # и здесь снова, но уже с другим объектом
x.content.append(1)
print(x.content)  # [1]
print(y.content)  # []
```

Важно понимать, что атрибут `content` будет разным у разных объектов данного класса.

Создание атрибутов вне конструктора возможно, но не рекомендуется. Код в примере ниже технически будет работать, но выглядит он плохо. Во-первых, такой код труднее читать и контролировать. Во-вторых, сам интерпретатор Python выполняет некоторые

оптимизации, когда видит, что атрибуты создаются внутри конструктора. Оптимизации касаются потребления оперативной памяти.

```
x = Box()
x.value = 127 # Не рекомендуется!
```

Рассмотрим создание разных атрибутов у разных значений объектов — конструктор с несколькими атрибутами:

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point2D(3, 4)  # При вызове конструктора
                   # ему передаются те аргументы,
                   # которые указаны здесь
print(p.x)  # 3
print(p.y)  # 4
```

Аргументы, которые принимает конструктор, должны быть записаны после слова `self`. При вызове метода (создании объекта) передаются только они, `self` передается автоматически. В конструктор можно передать любое количество аргументов, и при необходимости их дополнительно обработать или записать в атрибуты объекта как в примере.

В классе могут быть описаны и обычные методы, которые вызываются по их названию. В примере объявлен класс точки, содержащей метод, который вычисляет расстояние от точки до начала координат:

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_to_origin(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

p1 = Point2D(3, 4)
print(p1.distance_to_origin())  # 5.0
```

Объявленный в классе метод вызывается точно так же, как и у встроенных в Python объектов: `p1.distance_to_origin()`.

Метод всегда неявно принимает первым аргументом `self` — тот объект, у которого этот метод вызван. Но может принимать и другие аргументы. Например, можно описать метод, вычисляющей расстояние от одной точки до другой:

```
...

    def distance_to(self, other_point):
        return ((self.x - other_point.x) ** 2 + (self.y -
other_point.y) ** 2) ** 0.5

p1 = Point2D(1, 1)
p2 = Point2D(1, 8)
print(p1.distance_to(p2))  # 7.0
```

Пример рефакторинга программы на ООП. Рассмотрим практический пример. У нас есть программа, написанная в процедурном стиле, мы сделаем ее рефакторинг, чтобы переписать в объектно-ориентированном стиле.

Рефакторинг — изменение исходного кода программы, которое не меняет ее поведение, но меняет ее архитектуру и принципы построения программного кода. Обычно рефакторинг бывает нужен, когда мы столкнулись с невозможностью дальнейшего усложнения программного кода из-за того, что были приняты не самые удачные решения при его написании.

Программа представляет собой систему управления умным домом. В программе есть функция-заглушка `send_message`, которая имитирует отправку данных по сети. Функция возвращает сообщение в разном формате в зависимости от того, какое было исходное сообщение.

В программе есть списки сенсоров температуры, которые содержат пары «адрес—производитель»:

```
room_sensors = [
    # список содержит пары (адрес, производитель)
    ('192.168.1.45', 'KineticSensor'),  # Гостиная
    ('192.168.1.46', 'SmartApp'),       # Спальня
    ('192.168.1.47', 'SmartApp'),       # Кухня
    ('192.168.1.48', 'KineticSensor'),  # Ванная
]
```

Сложность задачи состоит в том, что есть сенсоры разных производителей, с которыми нужно общаться по разным протоколам.

Имеется также аналогичный список контроллеров отопления:

```
room_controller = [
    # содержит пары (адрес контроллера, номер пина)
    ('192.168.1.60', 1),  # Гостиная
    ('192.168.1.60', 2),  # Спальня
    ('192.168.1.61', 1),  # Кухня
    ('192.168.1.61', 2),  # Ванная
]
```

И существует список, который содержит данные о требуемой температуре в различных помещениях:

```
target_temperature = [
    # содержит пары (требуемая температура, допустимая дельта)
    (22.5, 0.3),
    (23.0, 0.2),
    (23.0, 0.5),
    (24.0, 0.1)
]
```

Логика работы программы: есть функция `get_temperatures`, которая получает список текущих температур в разных помещениях. Функция перебирает все сенсоры в списке и отправляет им запрос в зависимости от вида производителя сенсора. Способ чтения также зависит от вида сенсора. Полученная температура записывается в список, и этот список функция возвращает.

```
def get_temperatures():
    result = []
    for address, manufacturer in room_sensors:
        if manufacturer == 'KineticSensor':
            response = send_message(address, 'SENSOR READ 0')
            t = float(response)
        elif manufacturer == 'SmartApp':
            response = send_message(address, 'GET TEMP')
            t = float(response[5:])
        result.append(t)
    return result
```

В программе есть вспомогательная функция `set_heating`, которая отправляет запрос на контроллер. Функции `set_heating` нужно передать адрес контроллера, номер пина — то есть конкретного помещения, в котором регулируется температура, и действие — включить (1) или выключить (0).

```
def set_heating(address, pin, action):
    send_message(address, f'RELAY-SET-255-{pin}-{action}')
```

Алгоритм работы программы: считываем все температуры, потом собираем всю информацию воедино, с помощью условного оператора проверяем соответствие фактической температуры требуемой и зависимости от результата проверки включаем или выключаем отопление:

```
while True:
    temps = get_temperatures()
    for (addr, pin), (target, dt), fact in zip(room_controller,
target_temperature, temps):
        if fact > target + dt:
            set_heating(addr, pin, 0)
        elif fact < target - dt:
            set_heating(addr, pin, 1)
    time.sleep(5)
```

Рассмотрим причины рефакторинга программы. Во-первых, есть несколько видов сенсоров, требующих разной логики работы. Если встанет задача изменить сенсор, то придется глубоко перерабатывать код. Во-вторых, разрозненность данных, касающаяся одних и тех же объектов: есть два списка `room_sensors` и `room_controller`, которые описывают одни и те же помещения. Внесение изменений в список помещений или изменение порядка помещений будет сложной задачей.

Рассмотрим эту же программу, но уже в объектно-ориентированной парадигме. Некоторые фрагменты программы останутся такими же — например, останется функция `send_message`.

Создадим классы, описывающие сенсоры разных производителей. В каждом классе понадобится конструктор, который должен получить адрес для отправки сообщений. Также понадобится метод `read` для чтения информации с датчика. Метод `read` у датчиков будет отличаться в зависимости от их производителя.

```
class KineticSensor:
    def __init__(self, address):
        self.address = address

    def read(self):
        response = send_message(self.address, 'SENSOR READ 0')
        return float(response)


class SmartApp:
    def __init__(self, address):
```

```
        self.address = address

    def read(self):
        response = send_message(self.address, 'GET TEMP')
        return float(response[5:])
```

Опишем в программе класс `HeatingController`, который представляет контроллер отопления. Конструктор класса должен получить данные об адресе этого контроллера и о номере пина, которым мы будем управлять, и сохранить эти данные в соответствующем объекте.

В класс `HeatingController` поместим вспомогательную функцию `set_heating`, поскольку эта функция явно работает с теми данными, которые хранятся в объекте класса `HeatingController`. В функции больше не нужны аргументы `address` и `pin`, вместо этого мы получаем из объекта `self`. Метод `set_heating` мы скорее всего не будем извлекать извне — будет удобнее создать два конкретных метода, которые будут включать и выключать отопление. Такие методы в Python принято именовать начиная с символа нижнего подчеркивания «_», это говорит другим программистам, что метод является внутренним для данного класса.

Для использования извне мы должны создать публичный интерфейс — методы `increase` и `decrease`:

```
class HeatingController:
    def __init__(self, address, pin):
        self.address = address
        self.pin = pin

    def _set_heating(self, action):
        send_message(self.address,
f'RELAY-SET-255-{self.pin}-{action}')

    def increase(self):
        self._set_heating(1)

    def decrease(self):
        self._set_heating(0)
```

Напишем класс помещения — `Room`. Помещение должно быть связано с сенсором, который находится в помещении, и с контроллером, который управляет

температурой. В класс будем передавать готовые объекты: `sensor` — экземпляр класса `SmartAppSensor` или `KineticSensor`, `controller` — экземпляр класса `HeatingController`. Температуру будем передавать как два числа: значение температуры `target_temp` и допустимый предел `dt`.

Класс `Room` будет содержать один метод `check_temp`, проверяющий температуру и управляющий контроллером. Атрибут `name` содержит имя комнаты.

```
class Room:
    def __init__(self, name, sensor, controller, target_temp, dt):
        self.name = name
        self.dt = dt
        self.target_temp = target_temp
        self.controller = controller
        self.sensor = sensor

    def check_temp(self):
        fact = self.sensor.read()
        if fact > self.target_temp + self.dt:
            self.controller.decrease()
        elif fact < self.target_temp - self.dt:
            self.controller.increase()
```

Теперь нужно создать конкретные объекты, с которыми будет работать программа:

```
rooms = [
    Room('Гостиная', KineticSensor('192.168.1.45'),
         HeatingController('192.168.1.60', 1),
         22.5, 0.3),
    Room('Спальня', SmartApp('192.168.1.46'),
         HeatingController('192.168.1.60', 2),
         23.0, 0.2),
    Room('Кухня', SmartApp('192.168.1.47'),
         HeatingController('192.168.1.61', 1),
         23.0, 0.5),
    Room('Ванная', KineticSensor('192.168.1.48'),
         HeatingController('192.168.1.61', 2),
         24.0, 0.1)
]
```

Основная логика программы — пишем бесконечный цикл, в котором проходим по списку комнат и у каждой комнаты вызываем метод `check_temp`, после этого делаем паузу:

```
while True:
```

```
    for room in rooms:
        room.check_temp()
    time.sleep(5)
```

4. Инкапсуляция. Полиморфизм

Инкапсуляция (буквально — размещение в капсуле) — разделение интерфейса объекта и его внутренней реализации.

Чтобы лучше понять определение, обратимся к ранее рассмотренному примеру класса:

```
class HeatingController:
    def __init__(self, address, pin):
        ...

    def increase(self):
        ...

    def decrease(self):
        ...
```

В примере намеренно скрыто все лишнее. То, что представлено в коде — это интерфейс данного класса. То есть то, что должен знать человек, который будет этим классом пользоваться:

- конструктор класса принимает на вход адрес и номер пина;
- класс реализует методы увеличения и уменьшения отопления.

То, что находится внутри методов — не очень интересно тому, кто использует данный класс. Реализация классов интересна тому программисту, который этот класс пишет.

Таким образом, такой подход позволяет снизить когнитивную нагрузку. Инкапсуляция дает возможность абстрагироваться от малозначительных технических подробностей, которые могут быть достаточно сложными, и позволяет рассуждать об объектах в терминах их поведения, а не их внутреннего устройства.

Еще одно преимущество, которое дает инкапсуляция — простота внесения правок в код. Если при внесении изменений во внутреннюю логику работы классов, не меняется интерфейс, то весь код, которые опирается на интерфейс менять не нужно.

Понятие инкапсуляции в разных языках программирования отличается:

- в Python инкапсуляция — только механизм, позволяющий связать данные с методами, которые их обрабатывают;
- в других языках (C++, C#, Java) к этому добавляется специальный механизм, запрещающий доступ к некоторым внутренним данным объекта извне, который реализуется с помощью модификаторов доступа `public/private;`
- в Python есть соглашение, что имена атрибутов и методов, начинающиеся с символа '_' следует считать «закрытыми».

«Закрытые» атрибуты и методы нужны, так как в классе может потребоваться описание сложной внутренней логики работы, которую мы не хотим публиковать во вне, но она нужна для внутренних задач.

Основная идея инкапсуляции, которой следует руководствоваться на практике при проектировании объектно-ориентированного кода: вся логика, опирающаяся на внутреннее состояние объекта, должна быть инкапсулирована, а любые функции вне класса не должны непосредственно работать с внутренними свойствами объекта, а опираться только на его публичный интерфейс. Даже простое чтение атрибута объекта считается не очень хорошим тоном. Зачастую стараются сделать публичную функцию для чтения атрибута. Это делается в расчете на то, что в будущем потребуется поменять структуру атрибута.

Полиморфизм — одна из идей, которые делают парадигму ООП сильной и мощной.

Полиморфизм — единый интерфейс с различными реализациями.

Рассмотрим конкретный пример. Есть два класса `SmartAppSensor` и `KineticSensor`, которые описывают разные виды сенсоров. Их интерфейс выглядит абсолютно одинаково, поэтому эти классы мы можем считать полиморфными классами, а основной метод классов `read` — полиморфным методом:

```
class SmartAppSensor:
    def __init__(self, address):
        ...

    def read(self):
        ...


class KineticSensor:
    def __init__(self, address):
```

```
        ...

    def read(self):
        ...
```

Полиморфизм можно считать следствием инкапсуляции, поскольку мы разделили внутреннюю логику и интерфейс, и ничего не мешает создать несколько классов с одним и тем же интерфейсом, но разной внутренней логикой. При этом код, который использует интерфейс, может даже не знать, с каким конкретным типом он работает в данный момент. Мы воспользовались этим свойством при написании классам Room. В методе `check_heating` мы считываем данные о температуре с какого-то сенсора, при этом этот сенсор может быть или одним, или другим. Класс Room об этом не знает:

```
class Room:
    def __init__(self, name, sensor, controller, target_temp,
delta):
        ...

    def check_heating(self):
        fact_temp = self.sensor.read()
        ...
```

Стоит отметить, что только в Python с его «утиной» типизацией это можно сделать насколько просто. Другие языки программирования потребовали бы дополнительных действий для реализации полиморфного поведения, и технически бы это выглядело чуть сложнее.

Идея полиморфизма — конкретный тип, так же как и конкретная реализация метода, не имеет значения. То есть мы можем менять одну реализацию метода на другую даже непосредственно во время работы программы, при этом никак не меняя тот код, который непосредственно с этими данными работает.

Реализация полиморфного поведения глубоко внедрена в структуру самого языка Python. Есть множество функций, которые мы можем вызывать с самыми разными объектами — например, функции нахождения максимума или минимума, сортировка. Данные функции требуют лишь того, чтобы объекты, которые подаются в функцию, можно было сравнивать. То есть сказать, какой объект больше, а какой меньше. А конкретный тип используемого объекта не имеет никакого значения.

5. Наследование классов

Наследование позволяет создавать «похожие» классы более компактно.

На языке Python наследование может выглядеть следующим образом:

```
class Parent:  #Базовый класс
      def do_work (self):
            …
class Child(Parent):  #Класс-наследник
      def do_more_work (self):
            …
```

Наследник получает всю функциональность базового класса. Он может расширять его или добавлять новые детали, которых у базового не было.

Базовый класс называют родительским, а класс наследника — дочерним.

Важно! Экземпляр класса-наследника — «представитель» сразу двух классов: собственного и базового класса. Проверить это можно с помощью функции `isinstance()`:

```
x = Child()
isinstance(x, Child), isinstance(x, Parent)
```

Вывод функции будет содержать:

```
(True, True)
```

Однако экземпляр базового класса не может представлять класс наследника, поэтому вывод функции покажет ЛОЖЬ:

```
y = Parent()
isinstance(y, Child), isinstance(y, Parent)
(False, True)
```

Давайте рассмотрим ситуацию, где мы можем воспользоваться возможностями наследования. Возьмем пример с прошлого урока:

```
class KineticSensor:
    def __init__(self, address):
        self.address = address

    def read(self):
        response = send_message(self.address, 'SENSOR READ 0')
```

```
        return float(response)


class SmartApp:
    def __init__(self, address):
        self.address = address

    def read(self):
        response = send_message(self.address, 'GET TEMP')
        return float(response[5:])
```

Описанные классы очень похожи и содержат одинаковый код. Если мы захотим описать еще один сенсор, нам нужно повторить код. Это может спровоцировать возникновение ошибок.
Решить проблему и избавиться от повторяющегося кода можно, написав общий базовый класс Sensor:

```
class Sensor:
message = ' '  #Атрибут класса

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

    def read(self):
        response = send_message(self.address, self.message)
        return self.parse.response(response)

     def parse_response(self, response):
     """Этот метод должен быть определен в наследниках"""
     …
```

Важно! У класса как у объекта есть свои атрибуты. Они принадлежат всему классу, а не только отдельному экземпляру.

6. Особенности объектной модели в Python

Разновидности типизации данных в программировании:

- статическая — типы всех данных нам известны заранее;
- динамическая — типы данных вычисляются во время работы программы.

Есть еще строгая и нестрогая, которые отличаются тем, что строгая типизация не позволяет смешивать различные типы данных. Языки с нестрогой типизацией могут автоматически преобразовать данные для выполнения операции.

В Python все типы (как встроенные, так и определенные пользователем) являются наследниками одного супер-класса `object`.

Целое число является объектом:

```
isinstanse(5, object)
True
```

Строка является объектом:

```
isinstanse('Hello', object)
True
```

Экземпляр определенного нами класса тоже является объектом:

```
x = MyClass()
isinstanse(x, object)
True
```

Наследование всех классов от `object` дает возможность в полной мере использовать динамическую типизацию, поскольку в основе всех объектов один и тот же тип данных. Но зачем нам использовать динамическую типизацию?

Ее главное достоинство — простота обобщенного программирования. Это значит, что нам не обязательно заранее знать, с каким типом данных будет работать функция. Например, функция `add` может работать с объектами любых типов, если только для них определена операция сложения (числа, строки, кортежи, списки…):

```
def add(x, y):
    return x + y
```

В Python любая функция благодаря динамической типизации является обобщенной.

Утиный тест (неявная типизация, латентная типизация) — тест, который позволяет определить сущность объекта по его внешним признакам. «*Если нечто выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка*». Нам неважно, чем является объект по существу. Нам важно, какими свойствами он обладает и что умеет делать.

В результате теста важно выяснить, какой интерфейс реализует данный объект. Если он реализует подходящий интерфейс, то мы можем его использовать.

Близкие понятия, которые можно встретить наряду с динамической типизацией:

- структурная типизация — предполагает номинальную типизацию;
- типизация на основе поведения.

7. Элементы статической типизации. Абстрактные классы и протоколы

7.1. Недостатки динамической типизации

Динамическая типизация имеет свои недостатки. Главный минус заключается в том, что сложно заранее определить, будет ли функция правильно работать с тем или иным типом. Это сложно определить и программисту, и интерпретатору. По этой причине если вызов функции сопровождается ошибкой, то мы узнаем о ней только по факту.

При анализе многостраничного кода можно увидеть, что есть функция, которая принимает несколько аргументов, но непонятно, какие именно типы она принимает. Поэтому нам остается методом проб и ошибок пытаться вызвать функцию, либо вчитываться в ее код, и процесс может слишком затянуться.

7.2. Аннотации типов

Далеко не всегда нам нужны обобщенные функции, чаще мы уже знаем, с какими типами будем работать. Описать эти типы нам поможет синтаксис:

```
def typed_function(arg1: int, arg2: str) -> str:
...
```

После аргумента ставим двоеточие и указываем тип, который эта функция возвращает.

Аннотации типов никак не влияют на работу программы. Это подсказки для программиста, которые нужны по двум причинам:

1. В качестве средства документации.
2. Для инструментов статического анализа кода — например, встроенный в PyCharm.

Существуют более сложные аннотации. Они используются для описания переменных, которые хранят списки. Недостаточно написать просто `list`, потому что скорее всего это будет список конкретных объектов, которые хранят одинаковый тип. Для этого существует особый синтаксис:

```
user_ids: list[int]
user: tuple[int, str, str, list[int]]
```

Синтаксис для словарей:

```
user_by_id: dict[int, tuple[int, str, str, list[int]]]
```

Важно! Если аннотации типов получаются очень сложными — не стоит их писать. Лучше описать это в документации обычным текстом.

Аннотации — это удобный инструмент для подключения некоторых модулей (подробнее об этом в следующих уроках).

7.3. Инструкция `import`

Язык Python содержит обширную стандартную библиотеку, которая разбита на модули. Эти модули нужно подключать отдельно с помощью инструкции `import`. Это происходит потому, что их очень много, и их загрузка по умолчанию займет много времени и памяти. Их подключение происходит по требованию:

```
import math
math.sin(math.pi / 6)
```

```
0.49999999999994
```

Чтобы увидеть, какие атрибуты содержатся в модуле, можно использовать функцию `dir`:

```
dir(math)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan',
```

```
'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Если из модуля нужны только 1–2 функции, можно использовать другую форму импорта:

```
from math import sin, cos, pi
cos(pi / 3)
0.5000000000000001
```

7.4. Декораторы

Декоратор — это функция, которая «оборачивает» другую функцию для расширения или изменения ее поведения без непосредственного изменения ее кода.

Синтаксис выглядит таким образом:

```
@decorator
def function(*args):
    …
# Данная запись эквивалентна следующей:
def function(*args):
    …
function = decorator(function)
```

7.5. Абстрактные классы и реализация подклассов

Вернемся к нашему примеру с базовым классом Sensor. Этот класс может использоваться исключительно для создания классов-наследников. Чтобы это проверить существует специальный механизм — **абстрактный класс.**

```
import abc # abstract base class
class Sensor(abc.ABC):

def _init_(self, address):
    self.adress = address
```

```
def read(self):
      response = send_message(self.address, self.message)
      return self.parse_response(response)
```

Чтобы создать абстрактный метод, используем декоратор `@abc.abstractmethod`:

```
@abc.abstractmethod
def parse_response(self, response):
“”” Этот метод должен быть определен в наследниках “””
```

Чтобы создать абстрактное свойство, используем декоратор `@abc.abstractproperty`:

```
@abc.abstractproperty
def message(self):
“”” Должен выдать строку запроса “””
```

Пока мы не переопределим методы, объявленные как абстрактные, мы не можем создавать классы. Наследники обязаны переопределить все абстрактные методы:

```
class SmartAppSensor(Sensor):
      def parse_response(self, response):
      return float(responsee[5:])

@property #property - возможность создания вычисляемого атрибута
def message(self):
      return ‘GET TEMP’
```

В этом случае декоратор `@property` позволяет создать некий виртуальный атрибут.

7.6. Протоколы

Еще один инструмент статической типизации — **протокол**.

Протокол не предполагает историю наследования, и от этого становится только интереснее. Для описания протокола мы используем наследование, объявляем класс, но не пишем в нем никакой реализации. Класс используется только для аннотации типов. Выглядеть он будет так:

```
from typing import Protocol, runtime_checkable
@runtime_checkable
class Readable(Protocol)”
      def read(self) -> float: …

sensor = SmartAppSensor(‘192.168.1.45’)
print(isinstanse(sensor, Readable))    # True
```

Использование протоколов позволяет найти баланс между абсолютно динамической типизации и полной статической типизации. Он вводит некие ограничения на объект.

Модуль `collection.abc` содержит определения стандартных протоколов. Например, рассмотрим модуль `Sequence`:

```
from collection.abc import Sequence
# Sequence — какая-нибудь последовательность, объект, который
имеет размер и который можно использовать в цикле

def increment_all(numbers: Sequence[int]) -> None:
      for number in numbers:
            print (number + 1)

increment_all([1, 2, 3]) # OK
increment_all(range(1_000_000)) # OK
increment_all([‘hello’, ‘good bye’]) # NO!
```

8. Множественное наследование

Множественное наследование — возможность построить производный класс на основе нескольких базовых классов.

Необходимость в множественном наследовании может возникнуть в случае, если один класс должен реализовывать сразу несколько интерфейсов и использоваться в разных контекстах в качестве представителя разных базовых классов.

Например, класс «бухгалтер» может быть наследником одновременно двух классов – класса «сотрудник, получающий зарплату» и класса «материально ответственное лицо»:

```
class SalaryEmployee(Employee):
      """ Сотрудник, получающий зарплату """

class FinanciallyResponsiblePerson(Employee):
      """" Материально ответственное лицо """"

class Bookkeeper(SalaryEmployee, FinanciallyResponsiblePerson):
      """" Бухгалтер """"
```

В одну общую иерархию наследования этот класс не вписывается. Также оба класса «сотрудник, получающий зарплату» и «материально ответственное лицо» — наследниками другого базового класса «сотрудник». Отсюда мы получаем классическую проблему «ромба».

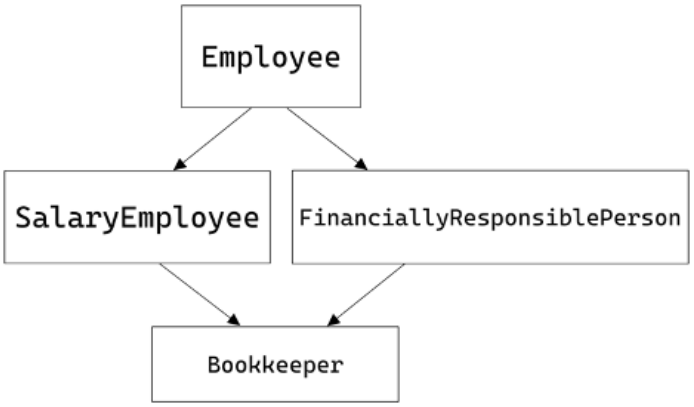

Рисунок 1. Проблема «ромба»

На рисунке представлены два наследника одного базового класса и один наследник этих двух наследников базового класса. Достаточно запутанно.

Важно! Стоит избегать множественного наследования, потому что оно может приводить к «странностям» в поведении программы из-за достаточно запутанной иерархии наследников.

Множественное наследование можно считать безопасным, если речь идет о наследовании только абстрактных интерфейсов.

Благодаря утиной типизации в Python для этого совсем не нужно использовать наследование. Для статической надежности можно использовать протоколы.

9. Проблемы, связанные с наследованием

Проблема «хрупкого базового класса». Наследование нарушает инкапсуляцию. Идея инкапсуляции состоит в том, чтобы различать интерфейс объекта и его реализацию. Любой код вне описания класса должен опираться только на интерфейс. При этом используя наследования, мы опираемся и на устройство базового класса. Это приводит к проблеме «хрупкого базового класса».

Наследование делает будущие изменения родительского класса крайне проблематичными, поскольку любое изменение в базовом классе затронет всех его наследников и может испортить логику их работы. То есть внести любые изменения в базовый класс становится почти невозможным, поэтому его проектирование — это очень ответственно. Неудачные решения, заложенные изначально в иерархию базового класса, могут испортить код, и потребуется его переписывать.

Проблема «Йо-йо». Еще одна проблема, с которой сталкиваются в наследовании, в шутку называется «Йо-йо» — как известная детская игрушка на нитке. Она заключается в длинной иерархии наследников, каждый из которых дополняет другого. Чтобы определить метод базового класса, нужно пройти всю эту цепочку. Это усложняет процесс анализа сложного кода.

Проблема «круга-эллипса». Это когда наследование в программировании не соответствует нашему бытовому представлению о соотношении объектов.

Например, с математической точки зрения круг — это частный случай эллипса, но в программировании все не так. Если взять эллипс за базовый класс, то можно сказать, что все присущие ему свойства должны по умолчанию так же относиться и к

кругу, как к классу наследник. Однако, у круга нет такого свойства базового класса эллипс, как растягиваться. Если растянуть круг, то он перестанет быть кругом — интерфейс разрушится.

Обратное соотношение — эллипса как наследника класса круг — тоже не сработает. Так как у круга есть метод, возвращающий размер радиуса, но его нельзя применить к эллипсу.

Проблема «взрыва классов». Иерархии классов могут становиться большими и сложными. Но если эта сложность будет превышать сложность задачи, которую мы решаем, то это сделает работу над кодом только запутаннее.

10. Композиция классов

Наследование определяется отношением **«является»**	**Композиция** определяется отношением **«имеет»**
Автомобиль является транспортом	*Автомобиль имеет двигатель*

Важно! Составной класс не наследует интерфейс класса компонента, но может его использовать.

На UML-диаграмме композиция обозначается стрелкой с ромбом:

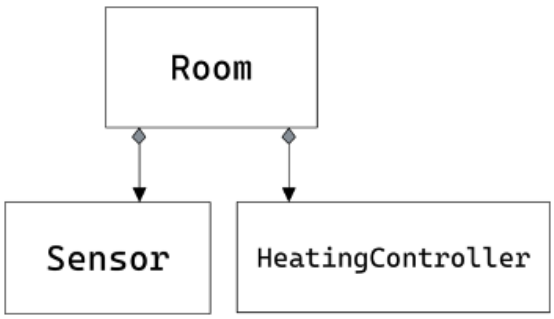

Рисунок 2. UML-диаграмма

Композиция по сравнению с наследованием создает меньше зависимостей между классами и не ломает инкапсуляцию.

11. Практические рекомендации

1. Если возможно, используйте композицию вместо наследования.
2. Если речь идет о наследовании только интерфейса без конкретной реализации, используйте Protocol.
3. Старайтесь избегать больших и сложных иерархий классов.
4. Не нарушайте принцип инкапсуляции.
5. Не используйте множественное наследование.
6. Неукоснительно следуйте «принципу подстановки» Б. Лисков (LSP):

   «Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом» (формулировка Р. Мартина).
7. «Если это выглядит как утка, плавает как утка и крякает как утка, но нуждается в батарейках, то, вероятно, вы пользуетесь неверными абстракциями».

Дополнительные материалы для самостоятельного изучения

1. Classes