Наследование

Цель занятия

После освоения темы:

- вы узнаете понятия и механизмы наследования;
- сможете проектировать классы с использованием механизма наследования.

План занятия

1. Наследование классов
2. Особенности объектной модели в Python
3. Элементы статистической типизации. Абстрактные классы и протоколы
4. Множественное наследование
5. Проблемы, связанные с наследованием
6. Композиция классов
7. Практические рекомендации

Используемые термины

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

Инкапсуляция — разделение элементов внутри класса и интерфейса, который они представляют.

Конспект занятия

1. 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.

2. 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.

3. Элементы статической типизации. Абстрактные классы и протоколы

3.1. 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.

3.2. 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).

3.3. 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
```

3.4. 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)
```

3.5. 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.

3.6. 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!
```

4. 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».

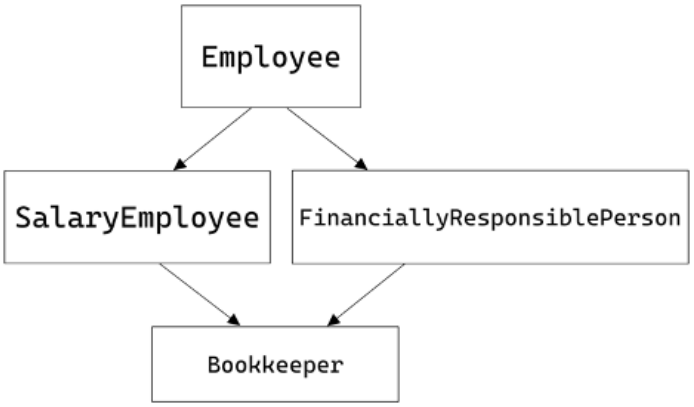

Рисунок 1. Проблема «ромба»

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.

5. 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.

6. 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:

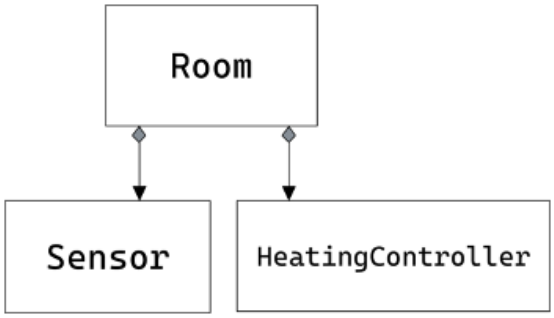

Рисунок 2. UML-диаграмма

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

7. 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».