top

КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Основы программирования на языке Python

Урок 10

Наследование.
Исключения

Contents

Основы: наследование

Наследование — почему и как?

Прежде чем мы начнем говорить о наследовании, давайте познакомимся с новым удобным механизмом, который классы и объекты Python используют для самоидентификации.

Начнем с примера. Посмотрите на код.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

sun = Star("Sun", "Milky Way")
print(sun)
```

Программа выводит только одну строку текста, которая в нашем случае выглядит так:

```
<__main__.Star object at 0x7f1074cc7c50>
```

Если запустить этот же код на другом компьютере, получится почти то же самое, но шестнадцатеричное число (подстрока начинается с 0x) будет отличаться, так как это просто внутренний идентификатор объекта в Python, и он вряд ли останется таким же, раз код выполняется в другой среде.

Как видите, вывод данных в этом случае не очень полезен, поэтому тут нужно нечто более конкретное, или просто более красивое, или более подходящее.

К счастью, у Python есть это «нечто».

Когда Python нужно вывести класс или объект в виде строки (помещение объекта в качестве аргумента в вызов функции print() отвечает этому условию), он пытается вызвать из объекта метод с именем __str__() и использовать строку, которую он возвращает.

Метод по умолчанию __str__ () возвращает предыдущую строку. А это некрасиво и не очень информативно. Решение довольно простое — нужно указать собственный метод имени.

Мы только что сделали это — посмотрите на код ниже.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

    def __str__(self):
        return self.name + ' in ' + self.galaxy

sun = Star("Sun", "Milky Way")
print(sun)
```

Новый метод __str__() создает строку, состоящую из названий звезды и галактики — ничего особенного, но результат вывода выглядит лучше.

Вы сможете угадать, что появится на экране? Запустите код, чтобы проверить, были ли вы правы.

Вернемся к наследованию. Однокоренное слово «наследство» старше, чем программирование, и оно описывает обычную практику передачи различных вещей от одного человека другому после его смерти. А в рамках

программирования, у наследования совершенно другое значение.

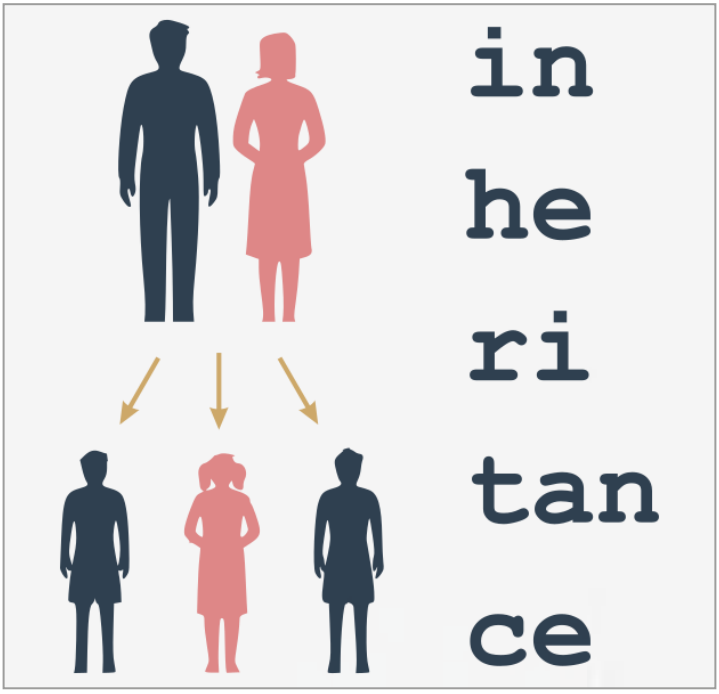

Рисунок 1

Давайте дадим ему определение: наследование является обычной практикой (в объектном программировании) передачи атрибутов и методов из суперкласса (уже определенного и существующего) в созданный класс, который называется подклассом.

Другими словами, наследование — способ создания нового класса не с нуля, а с использованием уже определенного набора характеристик. Новый класс наследует все существующие свойства, но в него можно добавлять новые, если это необходимо.

Благодаря этому можно строить более специализированные (более конкретные) классы, используя наборы общих правил и поведений.

Здесь самое важное — связь между суперклассом и всеми его подклассами.

> **Примечание:** *если B это подкласс A, а C это подкласс B, то, следовательно, и C тоже будет подклассом A, так как эта связь переносится на все нижестоящие подклассы.*

Очень простой пример — двухуровневое наследование:

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass
```

Все представленные классы пока пусты, потому что мы хотим показать вам, как работают взаимоотношения между супер- и подклассами. Мы их скоро наполним.

Можно сказать, что:

- Класс Vehicle является суперклассом для классов LandVehicle и TrackedVehicle;
- Класс LandVehicle является подклассом для Vehicle и одновременно суперклассом для TrackedVehicle;
- Класс TrackedVehicle является подклассом как для Vehicle, так и для LandVehicle.

Мы это знаем, потому что прочитали код.

А знает ли об этом Python? Можно ли спросить его об этом? Да, можно.

Inheritance: issubclass()

У Python есть функция, которая распознает отношения между двумя классами и, хотя ее принцип работы несложен, она способна проверить, является ли определенный класс подклассом другого класса.

Вот как это выглядит:

```
issubclass(ClassOne, ClassTwo)
```

Функция возвращает True (*Правда*), если ClassOne является подклассом ClassTwo, в противном случае — False (*Ложь*).

Давайте посмотрим, как она работает. Она может вас удивить. Посмотрите на код ниже. Внимательно прочитайте его.

У нас есть две вложенные петли. Их цель — проверить все возможные упорядоченные пары классов и вывести результат проверки на отношение подкласс-суперкласс в каждой паре.

Запустите код.

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass
```

```
for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

Выполнение кода даст следующий результат:

```
True False False
True True False
True True True
```

Давайте упорядочим результаты:

↓ является подклассом для →	Vehicle	LandVehicle	TrackedVehicle
Vehicle	True	False	False
LandVehicle	True	True	False
TrackedVehicle	True	True	True

Необходимо сделать одно важное замечание: каждый класс считается собственным подклассом.

Inheritance: isinstance()

Как вы уже знаете, объект является воплощением класса. То есть объект можно сравнить с тортом, который испекли по рецепту, который находится внутри класса. Из-за этого у нас могут быть некоторые проблемы.

Предположим, у вас есть торт (например, в виде аргумента, переданного в вашу функцию). Вы хотите знать, по какому рецепту его приготовили. Почему? Потому что вы должны понимать, чего от него ожидать, например,

есть ли в нем орехи, а для некоторых людей это критически важная информация. Точно так же наличие (или отсутствие) у объекта определенных характеристик может иметь решающее значение. Другими словами, принадлежит ли этот объект определенному классу или нет.

Это можно узнать с помощью функции isinstance():

```
isinstance(objectName, ClassName)
```

Функция возвращает True, если объект является экземпляром класса, в противном случае — False.

Что такое экземпляр класса? Это означает, что объект (торт) был приготовлен по рецепту, который содержится либо в классе, либо в одном из его суперклассов.

> **Примечание:** *если у подкласса такой же набор свойств, как и у любого из его суперклассов, это означает, что объекты подкласса могут делать то же, что и объекты, которые произошли от этого суперкласса, следовательно, это экземпляр его домашнего класса и любого из его суперклассов.*

Давайте проверим это. Проанализируйте код ниже.

Мы создали три объекта, по одному для каждого из классов. Далее, используя два вложенных цикла, мы проверяем все возможные пары объект-класс, чтобы выяснить, являются ли объекты экземплярами классов.

Запустите код.

```
class Vehicle:
    pass
class LandVehicle(Vehicle):
    pass
```

```
class TrackedVehicle(LandVehicle):
    pass

myVehicle = Vehicle()
myLandVehicle = LandVehicle()
myTrackedVehicle = TrackedVehicle()

for obj in [myVehicle, myLandVehicle, myTrackedVehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```

Вот что мы получим:

```
True False False
True True  False
True True  True
```

Давайте снова упорядочим результаты:

↓ является экземпляром для →	Vehicle	LandVehicle	TrackedVehicle
myVehicle	True	False	False
myLandVehicle	True	True	False
myTrackedVehicle	True	True	True

Таблица подтвердила наши ожидания?

Наследование: оператор is

Также стоит упомянуть еще один оператор Python, так как он ссылается непосредственно на объекты:

```
objectOne is objectTwo
```

Оператор is проверяет, относятся ли две переменные (тут, objectOne и objectTwo) к одному и тому же объекту.

Не забывайте, что переменные хранят не сами объекты, а только дескрипторы, указывающие на внутреннюю память Python.

Если присвоить значение переменной объекта другой переменной, то Python скопирует не объект, а только его дескриптор. Вот почему в определенных ситуациях операторы вроде is могут быть очень полезными.

Внимательно посмотрите на код ниже.

```
class SampleClass:
    def __init__(self, val):
        self.val = val

ob1 = SampleClass(0)
ob2 = SampleClass(2)
ob3 = ob1
ob3.val += 1

print(ob1 is ob2)
print(ob2 is ob3)
print(ob3 is ob1)
print(ob1.val, ob2.val, ob3.val)

str1 = "Mary had a little "
str2 = "Mary had a little lamb"
str1 += "lamb"

print(str1 == str2, str1 is str2)
```

Давайте проанализируем его:

- в коде есть очень простой класс и простой конструктор, который создает только одно свойство. Класс созда-

ет два объекта. Первый из них затем присваивается другой переменной, и его свойство val увеличивается на единицу.

- затем оператор is применяется трижды, чтобы проверить все возможные пары объектов, и выводятся все значения свойств val.
- в последней части кода проводится еще один эксперимент. После трех присваиваний обе строки содержат одинаковые тексты, но эти тексты хранятся в разных объектах.

Результат выполнения кода:

```
False
False
True
1 2 1
True False
```

Результаты доказывают, что ob1 и ob3 — на самом деле один и тот же объект, а str1 и str2 — нет, несмотря на то, что их содержимое одинаково.

Как Python находит свойства и методы

Теперь мы рассмотрим, как Python работает с наследуемыми методами.

Взгляните на этот пример кода.

```
class Super:
    def __init__(self, name):
        self.name = name
```

```
    def __str__(self):
        return "My name is " + self.name + "."

class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)

obj = Sub("Andy")

print(obj)
```

Давайте проанализируем его:

- тут есть класс Super, который определяет свой собственный конструктор, присваивая свойства объекту name.
- класс определяет метод __str__ (), который позволяет классу предоставлять информацию о себе в виде открытого текста.
- далее класс используется в качестве основы для создания подкласса с именем Sub. Класс Sub определяет свой собственный конструктор, который вызывает его из суперкласса. Обратите внимание на то, как мы это сделали: Super.__init__ (self, name).
- мы явно назвали суперкласс, указали на метод, который нужно вызвать, __init__() и предоставили все необходимые аргументы.
- мы создали экземпляр одного объекта класса Sub и вывели его результат.

Результат вывода:

```
My name is Andy.
```

Примечание: *так как в классе Sub нет метода __str__(), строка вывода должна быть выполнена внутри класса Super. Это означает, что метод __str__() был унаследован классом Sub.*

Посмотрите на код ниже.

```
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "My name is " + self.name + "."

class Sub(Super):
    def __init__(self, name):
        super().__init__(name)

obj = Sub("Andy")

print(obj)
```

Мы изменили его, чтобы показать вам другой способ получить доступ к любой сущности, которая определена внутри суперкласса.

В последнем примере мы явно назвали суперкласс. В этом примере мы используем функцию super(), которая получает доступ к суперклассу, не зная его имени:

```
super().__init__(name)
```

Функция super() создает контекст, в котором вам не нужно (более того, вы и не должны) передавать вызываемому методу аргумент self — вот почему можно ак-

тивировать конструктор суперкласса, используя только один аргумент.

> **Примечание:** *этот механизм можно использовать не только для вызова конструктора суперкласса, но и для того, чтобы получить доступ к любому из ресурсов, которые доступны внутри суперкласса.*

Попробуем сделать что-то похожее, но со свойствами (точнее, с переменными класса).

Посмотрите на пример.

```
# Testing properties: class variables
class Super:
    supVar = 1

class Sub(Super):
    subVar = 2

obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

Как видите, класс Super определяет одну переменную класса с именем supVar, а класс Sub определяет переменную с именем subVar.

Обе эти переменные видимы внутри объекта класса Sub, поэтому код выводит:

```
2
1
```

Тот же эффект можно наблюдать у переменных экземпляра — взгляните на второй пример.

```
# Testing properties: instance variables
class Super:
    def __init__(self):
        self.supVar = 11

class Sub(Super):
    def __init__(self):
        super().__init__()
        self.subVar = 12

obj = Sub()
print(obj.subVar)
print(obj.supVar)
```

Конструктор класса Sub создает переменную экземпляра с именем subVar, а конструктор Super делает то же самое с переменной с именем supVar. Как и ранее, обе переменные доступны из объекта класса Sub.

Вывод программы следующий:

```
12
11
```

> **Примечание:** *переменная supVar существует только благодаря тому, что мы вызвали конструктор класса Super. Если этим пренебречь, в созданном объекте вообще не будет переменной (попробуйте сами).*

Теперь можно сформулировать общее правило, описывающее поведение Python.

При попытке получить доступ к объекту какого-нибудь объекта Python попытается сделать следующее (именно в этом порядке):

- найти его внутри самого объекта;
- найти его во всех классах, которые участвуют в наследовании объекта снизу вверх.

Если вышеперечисленные действия не дадут результата, будет сгенерирована ошибка (AttributeError).

Первое условие стоит разобрать подробнее. Как вы знаете, все объекты, унаследованные от определенного класса, могут иметь разные наборы атрибутов, а некоторые атрибуты могут быть добавлены объекту спустя долгое время после его создания.

Пример ниже сводит это все в трехуровневой линии наследования.

```
class Level1:
    varia1 = 100
    def __init__(self):
        self.var1 = 101

    def fun1(self):
        return 102


class Level2(Level1):
    varia2 = 200
    def __init__(self):
        super().__init__()
        self.var2 = 201

    def fun2(self):
        return 202
```

```
class Level3(Level2):
    varia3 = 300
    def __init__(self):
        super().__init__()
        self.var3 = 301

    def fun3(self):
        return 302

obj = Level3()

print(obj.varia1, obj.var1, obj.fun1())
print(obj.varia2, obj.var2, obj.fun2())
print(obj.varia3, obj.var3, obj.fun3())
```

Внимательно проанализируйте этот код.

Все, что мы до этого говорили, было связано с одиночным наследованием, когда подкласс имеет ровно один суперкласс. Это наиболее распространенная (и рекомендуемая) ситуация.

Но Python способен на большее. В следующих уроках мы покажем вам примеры множественного наследования.

Наследование становится множественным, когда у класса появляется больше одного суперкласса.

Синтаксически такое наследование представлено в виде списка суперклассов через запятую, в круглых скобках, которые следуют за новым именем класса. Как здесь:

```
class SuperA:
varA = 10
def funA(self):
return 11
```

```
class SuperB:
varB = 20
def funB(self):
return 21

class Sub(SuperA, SuperB):
pass

obj = Sub()

print(obj.varA, obj.funA())
print(obj.varB, obj.funB())
```

У класса Sub есть два суперкласса: SuperA и SuperB. Это означает, что класс Sub наследует все, что есть и у SuperA, и у SuperB.

Результат выполнения кода:

```
10 11
20 21
```

Теперь пришло время ввести новый термин — переопределение (overriding).

Как вы думаете, что произойдет, если сущность с конкретным именем будет определять не один суперкласс, а больше?

Давайте проанализируем пример ниже.

```
class SuperA:
    varA = 10
    def funA(self):
        return 11
```

```
class SuperB:
    varB = 20
    def funB(self):
        return 21

class Sub(SuperA, SuperB):
    pass

obj = Sub()

print(obj.varA, obj.funA())
print(obj.varB, obj.funB())
```

Оба класса, Level1 и Level2, определяют метод fun() и свойство var. Значит ли это, что класс объекта Level3 сможет получить доступ к двум копиям каждого объекта? Совсем нет.

Объект, определенный позже (в смысле наследования), переопределяет объект, который был определен ранее. Поэтому у нас получается такой результат:

```
200 201
```

Как видите, переменная класса var и метод fun() из класса Level2 переопределяют объекты с одинаковыми именами, полученными из класса Level1.

Это можно использовать для изменения поведения класса по умолчанию (или ранее определенного), если любой из его классов должен действовать не так, как его предок.

Python ищет сущность снизу вверх и прекращает поиск, как только находит первую сущность, которая полностью удовлетворяет условиям.

Но как это будет работать, если у класса есть два предка с одной и той же сущностью, и они лежат на одном уровне? Другими словами, чего ожидать при создании класса с множественным наследованием? Давайте рассмотрим этот вопрос.

Посмотрите на пример ниже.

```
class Left:
    var = "L"
    varLeft = "LL"
    def fun(self):
        return "Left"

class Right:
    var = "R"
    varRight = "RR"
    def fun(self):
        return "Right"

class Sub(Left, Right):
    pass

obj = Sub()

print(obj.var, obj.varLeft, obj.varRight, obj.fun())
```

Класс Sub наследует от двух суперклассов: Left и Right (названия суперклассов должны отражать их назначение, т.е. для чего они используются или что делают).

Естественно, переменная класса varRight происходит от класса Right, а varLeft происходит от класса Left соответственно.

Это понятно. Но откуда тогда берется var? Можем ли мы угадать это? У нас та же проблема с методом fun() —

откуда он будет вызываться? Из Left или из Right? Запустим программу. Результат выполнения следующий:

```
L LL RR Left
```

Ответ на оба вопроса — из класса Left. Достаточно ли этого, чтобы сформулировать общее правило? Да.

Можно заключить, что Python ищет компоненты объекта в следующем порядке:

- внутри самого объекта;
- в его суперклассе, снизу вверх;
- если в конкретной линии наследования более одного класса, Python сканирует их слева направо.

Разве нужно что-то еще? Просто внесите небольшую поправку в код. Замените class Sub (Left, Right): на class Sub (Right, Left):, затем снова запустите программу и посмотрите, что произойдет.

Что вы видите сейчас? Мы видим следующее:

```
R LL RR Right
```

У вас такой же вывод или какой-то другой?

Построение иерархии классов

Построение иерархии классов — это не просто искусство ради искусства.

Можно поделить одну задачу между классами и решить, какой из них должен быть расположен вверху, а какой — внизу иерархии, а для этого необходимо тщательно проанализировать эту задачу. Но прежде чем мы покажем вам, как это делается (и как этого не надо делать), нужно

рассмотреть один интересный эффект. В нем нет ничего необычного (это просто последствие общих правил, представленных выше), но он может помочь вам понять, как работают некоторые коды, и как этот эффект можно использовать для создания гибкого набора классов.

Посмотрите на этот код.

```
class One:
    def doit(self):
        print("doit from One")
    def doanything(self):
        self.doit()

class Two(One):
    def doit(self):
        print("doit from Two")

one = One()
two = Two()

one.doanything()
two.doanything()
```

Давайте проанализируем его:

- в нем два класса, которые называются One и Two, класс Two происходит от класса One. Ничего особенного. Тем не менее, метод doit() выглядит необычно.
- метод doit() определяется дважды: сначала внутри метода One, а затем внутри метода Two. Суть примера заключается в том, что он вызывается только один раз — внутри One.

Вопрос в том, какой из этих двух методов будет вызываться в двух последних строках кода.

Первый вызов кажется простым, и это на самом деле так — вызов doanything() из объекта, который называется one, активирует первый из методов.

Второй вызов требует некоторого внимания. Он тоже простой, если вы помните, как Python находит компоненты класса. Второй вызов запустит doit(), который находится внутри класса Two, несмотря на то, что вызов происходит в пределах класса One.

По сути, результат работы кода будет следующим:

```
doit from One
doit from Two
```

> **Примечание:** *ситуация, в которой подкласс может изменять поведение своего суперкласса (как в примере), называется полиморфизмом. Слово «полиморфизм» происходит от греческого (polys — «много» и morphe — «форма») и в рамках программирования означает, что один и тот же класс может принимать различные формы в зависимости от переопределений, которые производятся любым из его подклассов.*

Метод, переопределенный в любом из суперклассов и изменяющий таким образом поведение суперкласса, называется виртуальным.

Другими словами, ни один класс не задается раз и навсегда. Поведение каждого класса может быть изменено в любое время любым из его подклассов.

Мы покажем вам как использовать полиморфизм для расширения гибкости класса.

Посмотрите на пример.

```
import time

class TrackedVehicle:
    def controltrack(left, stop):
        pass

    def turn(left):
        controltrack(left, True)
        time.sleep(0.25)
        controltrack(left, False)

class WheeledVehicle:
    def turnfrontwheels(left, on):
        pass

    def turn(left):
        turnfrontwheels(left, True)
        time.sleep(0.25)
        turnfrontwheel(left, False)
```

Ничего не напоминает? Конечно, напоминает. Он относится к примеру, который мы разбирали в начале модуля, когда говорили об общих понятиях объектно-ориентированного программирования.

Может показаться странным, но мы тогда не использовали наследование — просто чтобы показать, что это не единственный наш инструмент, и нам все равно удалось получить необходимый результат.

Мы определили два отдельных класса, способных производить два разных типа наземных транспортных средств. Основное различие между ними заключается в том, как они поворачивают. Колесный автомобиль

обычно просто поворачивает передние колеса. Транспорт на гусеничном ходу должен остановить одну из гусениц.

Следите за кодом.

- чтобы транспорт на гусеничном ходу повернулся, он останавливается и двигается дальше на одной из своих гусениц (за это отвечает метод controltrack(), который будет реализован позже)
- колесный автомобиль поворачивает тогда, когда поворачиваются его передние колеса (за это отвечает метод turnfrontwheels())
- метод turn() использует метод, подходящий для каждого конкретного транспортного средства.

Теперь вы видите, что не так с кодом?

Методы turn() выглядят слишком похожими, поэтому их нужно изменить.

Перестроим код следующим образом — введем суперкласс, где соберем все общие схемы управления транспортными средствами, и перенесем все особые случаи в подклассы.

Посмотрите на код еще раз.

```
import time

class Vehicle:
    def changedirection(left, on):
        pass

    def turn(left):
        changedirection(left, True)
        time.sleep(0.25)
        changedirection(left, False)
```

```
class TrackedVehicle(Vehicle):
    def controltrack(left, stop):
        pass

    def changedirection(left, on):
        controltrack(left, on)

class WheeledVehicle(Vehicle):
    def turnfrontwheels(left, on):
        pass

    def changedirection(left, on):
        turnfrontwheels(left, on)
```

Вот что мы сделали:

- определили суперкласс с именем Vehicle, который использует метод turn() для реализации общей схемы поворота, а сам поворот выполняется методом changedirection(). Примечание: первый метод пуст, так как мы собираемся поместить все детали в подкласс (такой метод часто называют абстрактным методом, поскольку он только демонстрирует некоторую возможность, которая будет реализована позже)
- мы определили подкласс TrackedVehicle (примечание: он является производным от класса Vehicle), который инстанциировал метод changedirection(), используя конкретный метод с именем controltrack()
- соответственно, подкласс WheeledVehicle делает то же самое, но использует метод turnfrontwheel(), чтобы заставить транспортное средство повернуть.

Самое важное преимущество (кроме читабельности) состоит в том, что такой код позволяет вам реализовать

новый алгоритм поворота, просто изменив метод turn(). Это можно сделать только в одном месте, а все транспортные средства будут ему подчиняться.

Вот так полиморфизм помогает разработчику поддерживать чистоту и последовательность в коде.

Наследование — не единственный способ создания адаптируемых классов. Тех же целей можно достичь (не всегда, но очень часто), используя технику под названием композиция.

Композиция — это процесс создания объекта при помощи других объектов. Объекты, используемые в композиции, предоставляют набор желаемых характеристик (свойств и/или методов), поэтому мы можем сказать, что они действуют как блоки, которые потом используются для построения более сложной структуры.

Можно сказать, что:

- наследование расширяет возможности класса путем добавления новых компонентов и изменения существующих; другими словами, полный рецепт содержится внутри самого класса и всех его предков; объект берет и использует все, что принадлежит классу;
- композиция проецирует класс как контейнер, который может хранить и использовать другие объекты (производные от других классов), где каждый из объектов реализует часть желаемого поведения класса.

Давайте продемонстрируем разницу с помощью ранее определенных транспортных средств. Предыдущий подход привел нас к иерархии классов, в которой самый верхний класс знал об общих правилах, используемых

при развороте транспортного средства, но не знал, как управлять соответствующими компонентами (колесами или гусеницами).

Эту способность реализовали подклассы при помощи специальных механизмов. Давайте сделаем почти то же самое, но с использованием композиции. Как и в предыдущем примере, класс знает, как повернуть транспортное средство, но фактический поворот выполняется специальным объектом, который хранится в свойстве под названием controller. Controller может управлять транспортным средством, манипулируя его деталями.

Посмотрите на код — так это могло бы выглядеть.

```
import time

class Tracks:
    def changedirection(self, left, on):
        print("tracks: ", left, on)

class Wheels:
    def changedirection(self, left, on):
        print("wheels: ", left, on)

class Vehicle:
    def __init__(self, controller):
        self.controller = controller

    def turn(self, left):
        self.controller.changedirection(left, True)
        time.sleep(0.25)
        self.controller.changedirection(left, False)

wheeled = Vehicle(Wheels())
```

```
tracked = Vehicle(Tracks())
wheeled.turn(True)
tracked.turn(False)
```

Есть два класса: Tracks и Wheels. Они знают, как контролировать направление движения транспорта. У нас также есть класс Vehicle, который может использовать любой из доступных контроллеров (два уже определенных или любой другой, который будет определен позже) — сам controller передается классу во время инициализации.

Таким образом, способность транспортного средства поворачивать состоит из внешнего объекта, не реализованного внутри класса Vehicle.

Другими словами, у нас есть универсальный автомобиль, и мы можем установить на него либо гусеницы, либо колеса.

Результат работы программы будет следующим:

```
wheels: True True
wheels: True False
tracks: False True
tracks: False False
```

Единичное и множественное наследование

Как вы уже знаете, вам ничто не может помешать использовать множественное наследование в Python. Вы можете получить любой новый класс из нескольких ранее определенных классов.

Но есть одно «но». Вы можете, но далеко не всегда должны это делать.

Не забывайте, что:

- единичное наследование всегда проще, безопаснее и легче для понимания и использования;
- множественное наследование всегда рискованно, так как тут гораздо больше возможностей ошибиться при определении частей суперклассов, что обязательно повлияет на новый класс;
- переопределение при множественном наследовании может оказаться чрезвычайно сложным; более того, функция super() становится неоднозначной;
- множественное наследование нарушает принцип единственной ответственности (подробнее по ссылке: https://ru.wikipedia.org/wiki/Принцип_единственной_ответственности), поскольку новый класс создается из двух (или более) классов, которые ничего не знают друг о друге;
- мы настоятельно рекомендуем использовать множественное наследование только в качестве крайней необходимости. Если вам нужно много разных функций, принадлежащих разным классам, скорее всего композиция будет лучшим решением.

Ромбы и почему они не нужны

Вопросы, которые могут возникнуть из-за множественного наследования, иллюстрирует классическая «проблема ромба» (diamond problem) или «ромбовидное наследование». Эта проблема получила свое название благодаря очертаниям диаграммы наследования классов в этой ситуации. Посмотрите на рисунок.

У нас есть:

- самый верхний суперкласс A;
- два подкласса, полученные из A — B и C;
- и еще самый нижний подкласс D, полученный из B и C (или C и B, так как в Python эти два варианта будут отличаться).

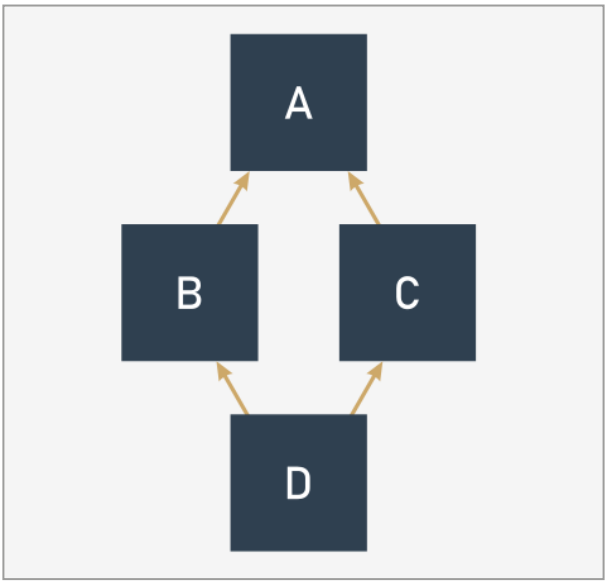

Рисунок 2

Но Python не любит ромбы и не позволит вам реализовать что-либо подобное. Если вы попытаетесь построить иерархию вроде этой:

```
class A:
pass

class B(A):
pass
```

```
class C(A):
pass

class D(A, B):
pass

d = D()
```

то получите ошибку TypeError и следующе сообщение:

```
Cannot create a consistent method resolution
order (MRO) for bases B, A
```

где MRO (*Method Resolution Order,* порядок разрешения методов) — это алгоритм, применяемый в Python для поиска необходимых методов в дереве наследования.

Ромбы красивые... но в программировании их стоит избегать.

Снова исключения

Подробнее об исключениях

В процессе обсуждения объектно-ориентированного программирования у нас появилась отличная возможность вернуться к исключениям. Объектная природа исключений в Python делает их очень гибким инструментом, отвечающим особым потребностям и даже тем, о которых вы еще не знаете.

Прежде чем мы погрузимся в объектно-ориентированные тонкости исключений, мы хотели бы показать вам некоторые синтаксические и семантические аспекты того, как Python обрабатывает блок try-except, поскольку в предыдущих модулях мы обсудили не все.

Первая особенность, о которой мы хотим сейчас поговорить, — это дополнительная ветка, которую можно поместить внутри (или, скорее, непосредственно позади) блока try-except. Это та часть кода, которая начинается с else, как в примере ниже.

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        return None
    else:
        print("Everything went fine")
        return n
```

```
print(reciprocal(2))
print(reciprocal(0))
```

Код, помеченный таким образом, выполняется, если (и только в этом случае) внутри блока try: не было сгенерировано ни одной ошибки. Можно сказать, что после блока try: может быть выполнена ровно одна ветка: либо та, которая начинается с except (не забывайте, что их может быть две и больше), либо та, которая начинается с else.

> **Примечание:** *блок else: должен идти после последнего блока except.*

Код в примере выводит следующий результат:

```
Everything went fine
0.5
Division failed
None
```

Блок try-except можно расширить еще одним способом — добавив часть с ключевым словом finally (это должна быть последняя ветка кода, который занимается обработкой исключений).

> **Примечание:** *эти два варианта (else и finally) никак не зависят друг от друга и могут сосуществовать или генерироваться независимо друг от друга.*

Блок finally всегда выполняется (он завершает выполнение блока try-except, о чем и говорит его имя) вне зависимости от того, что произошло ранее, даже при воз-

никновении исключения и независимо от того, было ли оно обработано.

Посмотрите на код ниже.

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        n = None
    else:
        print("Everything went fine")
    finally:
        print("It's time to say goodbye")
        return n

print(reciprocal(2))
print(reciprocal(0))
```

Результат вывода следующий:

```
Everything went fine
It's time to say good bye
0.5
Division failed
It's time to say good bye
None
```

Исключения — это классы

В предыдущих примерах мы вполне были довольны тем, что обнаруживали определенные типы исключений и реагировали на них соответствующим образом. А сейчас мы хотим пойти дальше и заглянуть внутрь самого исключения.

Вы, наверное, даже не удивились, когда узнали из названия выше, что исключения являются классами. Кроме того, когда возникает исключение, создается объект класса, который проходит все уровни выполнения программы в поисках ветки except, которая готова разобраться с этим исключением.

Такой объект несет некоторую полезную информацию, которая может помочь вам точно определить все нюансы рассматриваемой ситуации. Для достижения этой цели в Python есть специальный вариант описания исключения, который приведен ниже.

```
try:
    i = int("Hello!")
except Exception as e:
    print(e)
    print(e.__str__())
```

Как видите, оператор except расширен и содержит дополнительную фразу, начинающуюся с ключевого слова as, за которым следует идентификатор. Идентификатор перехватывает объект исключения, чтобы вы могли проанализировать, откуда он взялся, и сделать правильные выводы.

> **Примечание:** *область действия идентификатора распространяется только на блок except и не более.*

В примере представлен очень простой способ использования полученного объекта — вам просто нужно вывести его (как видно в примере, результат создается методом __str__() этого объекта) и содержит краткую информацию с описанием причины.

Такое же сообщение будет выведено, если в коде нет подходящего блока except, а Python будет вынужден обрабатывать его в одиночку.

Все встроенные исключения в Python образуют иерархию классов. Но вы всегда можете расширить ее, если посчитаете нужным.

Посмотрите на код ниже.

```
def printExcTree(thisclass, nest = 0):
    if nest > 1:
        print("    |" * (nest - 1), end="")
    if nest > 0:
        print("    +---", end="")

    print(thisclass.__name__)

    for subclass in thisclass.__subclasses__():
        printExcTree(subclass, nest + 1)

printExcTree(BaseException)
```

Эта программа выводит все готовые классы исключений в виде дерева.

Потому что дерево — прекрасный пример рекурсивной структуры данных, а рекурсия кажется лучшим инструментом для обхода дерева. Функция printExcTree() принимает два аргумента:

- точка внутри дерева, с которой мы начинаем обходить дерево;
- уровень вложенности (мы будем использовать его для построения упрощенного представления ветвей дерева).

Давайте начнем с корня дерева — корнем классов исключений является класс BaseException (это суперкласс для всех других исключений).

Для каждого из обнаруженных классов выполните один и тот же набор операций:

- выведите его имя, которое нужно взять из свойства __name__;
- выполните обход в списке подклассов, который находится в методе __subclasses__(), и рекурсивно вызовите функцию printExcTree(), увеличивая, таким образом, уровень вложенности.

Обратите *внимание на то, как мы показали ветки и вилки. Результат вывода никак не сортируется, так что вы можете попытаться отсортировать его самостоятельно, если вам интересно проверить свои силы. Кроме того, некоторые ветки представлены с небольшими неточностями. Это тоже можете исправить при желании.*

Вот как это выглядит:

```
BaseException
+---Exception
| +---TypeError
| +---StopAsyncIteration
| +---StopIteration
| +---ImportError
| | +---ModuleNotFoundError
| | +---ZipImportError
| +---OSError
| | +---ConnectionError
| | | +---BrokenPipeError
```

```
| | | +---ConnectionAbortedError
| | | +---ConnectionRefusedError
| | | +---ConnectionResetError
| | +---BlockingIOError
| | +---ChildProcessError
| | +---FileExistsError
| | +---FileNotFoundError
| | +---IsADirectoryError
| | +---NotADirectoryError
| | +---InterruptedError
| | +---PermissionError
| | +---ProcessLookupError
| | +---TimeoutError
| | +---UnsupportedOperation
| | +---herror
| | +---gaierror
| | +---timeout
| | +---Error
| | | +---SameFileError
| | +---SpecialFileError
| | +---ExecError
| | +---ReadError
| +---EOFError
| +---RuntimeError
| | +---RecursionError
| | +---NotImplementedError
| | +---_DeadlockError
| | +---BrokenBarrierError
| +---NameError
| | +---UnboundLocalError
| +---AttributeError
| +---SyntaxError
| | +---IndentationError
| | | +---TabError
| +---LookupError
| | +---IndexError
```

```
| | +---KeyError
| | +---CodecRegistryError
| +---ValueError
| | +---UnicodeError
| | | +---UnicodeEncodeError
| | | +---UnicodeDecodeError
| | | +---UnicodeTranslateError
| | +---UnsupportedOperation
| +---AssertionError
| +---ArithmeticError
| | +---FloatingPointError
| | +---OverflowError
| | +---ZeroDivisionError
| +---SystemError
| | +---CodecRegistryError
| +---ReferenceError
| +---BufferError
| +---MemoryError
| +---Warning
| | +---UserWarning
| | +---DeprecationWarning
| | +---PendingDeprecationWarning
| | +---SyntaxWarning
| | +---RuntimeWarning
| | +---FutureWarning
| | +---ImportWarning
| | +---UnicodeWarning
| | +---BytesWarning
| | +---ResourceWarning
| +---error
| +---Verbose
| +---Error
| +---TokenError
| +---StopTokenizing
| +---Empty
| +---Full
```

```
| +---_OptionError
| +---TclError
| +---SubprocessError
| | +---CalledProcessError
| | +---TimeoutExpired
| +---Error
| | +---NoSectionError
| | +---DuplicateSectionError
| | +---DuplicateOptionError
| | +---NoOptionError
| | +---InterpolationError
| | | +---InterpolationMissingOptionError
| | | +---InterpolationSyntaxError
| | | +---InterpolationDepthError
| | +---ParsingError
| | | +---MissingSectionHeaderError
| +---InvalidConfigType
| +---InvalidConfigSet
| +---InvalidFgBg
| +---InvalidTheme
| +---EndOfBlock
| +---BdbQuit
| +---error
| +---_Stop
| +---PickleError
| | +---PicklingError
| | +---UnpicklingError
| +---_GiveupOnSendfile
| +---error
| +---LZMAError
| +---RegistryError
| +---ErrorDuringImport
+---GeneratorExit
+---SystemExit
+---KeyboardInterrupt
```

Подробная анатомия исключений

Давайте подробнее рассмотрим объект исключения, поскольку здесь есть действительно интересные тонкости (мы вернемся к этой проблеме, когда будем рассмотрим базовые методы ввода-вывода в Python, поскольку их подсистема исключений немного расширяет эти объекты).

Класс BaseException вводит свойство args. Это кортеж, предназначенный для сбора всех аргументов, переданных конструктору класса. Если конструкция была вызвана без каких-либо аргументов, то этот конструктор пустой. Или содержит только один элемент, если конструктор получил один аргумент (аргумент self здесь не учитывается) и так далее.

Мы подготовили простую функцию, которая элегантно выведет свойство args. Посмотрите на функцию ниже.

```
def printargs(args):
    lng = len(args)
    if lng == 0:
        print("")
    elif lng == 1:
        print(args[0])
    else:
        print(str(args))

try:
    raise Exception
except Exception as e:
    print(e, e.__str__(), sep=' : ' ,end=' : ')
    printargs(e.args)

try:
    raise Exception("my exception")
```

```
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    printargs(e.args)

try:
    raise Exception("my", "exception")

except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    printargs(e.args)
```

Мы использовали функцию для печати содержимого свойства args в трех разных случаях, где исключение класса Exception вызывается тремя разными способами. Мы также вывели сам объект и результат вызова __str__().

Первый случай выглядит совершенно обычно — есть только имя Exception, которое следует за ключевым словом raise. Это означает, что объект этого класса был создан самым обычным способом.

Второй и третий случаи могут показаться немного странными на первый взгляд, но на самом деле здесь нет ничего необычного — это всего лишь вызовы конструктора. Во втором выражении raise конструктор вызывается с одним аргументом, а в третьем — с двумя.

Как видите, выходные данные программы демонстрируют это через соответствующее содержимое свойства args:

```
: :
my exception : my exception : my exception
('my', 'exception') : ('my', 'exception') :
                      ('my', 'exception')
```

Как создать собственное исключение

Иерархия исключений не является ни закрытой, ни законченной, и вы всегда можете расширить ее, если необходимо создать свой мир, заполненный вашими исключениями.

Это может пригодиться при создании сложного модуля, который обнаруживает ошибки и вызывает исключения, и вам нужно, чтобы эти исключения отличались от стандартных исключений в Python.

Для этого нужно задать новые исключения в качестве подклассов, наследующих стандартные классы.

> **Примечание:** *если нужно создать исключение, которое будет использоваться как специальный случай любого встроенного исключения, то его нужно наследовать только из этого исключения. Если вам нужно построить собственную иерархию, и вы не хотите, чтобы она была тесно связана с деревом исключений Python, выведите его из любого верхнего класса исключений, например,* *Exception*.

Представьте, что вы создали совершенно новую арифметику, которая подчиняется вашим собственным законам и теоремам. Понятно, что деление тоже изменится и должно будет вести себя не так, как обычное деление. Также ясно, что это новое деление должно вызвать свое собственное исключение, отличное от встроенного ZeroDivisionError. Но разумно также предположить, что в некоторых обстоятельствах вы (или пользователь) можете рассматривать все деления на ноль одинаково.

Подобные требования можно выполнить представленным ниже способом. Посмотрите на код ниже.

```
class MyZeroDivisionError(ZeroDivisionError):
    pass

def doTheDivision(mine):
    if mine:
        raise MyZeroDivisionError("some worse news")
    else:
        raise ZeroDivisionError("some bad news")

for mode in [False, True]:
    try:
        doTheDivision(mode)
    except ZeroDivisionError:
        print('Division by zero')

for mode in [False, True]:
    try:
        doTheDivision(mode)
    except MyZeroDivisionError:
        print('My division by zero')
    except ZeroDivisionError:
        print('Original division by zero')
```

И давайте проанализируем его:

- Мы определили наше собственное исключение MyZeroDivisionError, полученное из встроенного ZeroDivisionError. Как видите, мы решили не добавлять новые компоненты в класс.
  В зависимости от того, что вам нужно, исключение этого класса можно рассматривать как обычное ZeroDivisionError или как особый случай.

- Функция doTheDivision() генерирует либо MyZeroDivisionError, либо ZeroDivisionError, в зависимости от значения аргумента.
  Функция вызывается в общей сложности четыре раза, в то время как первые два вызова обрабатываются с использованием только одного (более общего) блока except, а два последние с двумя разными блоками, которые способны различать исключения (не забывайте: порядок блоков имеет принципиальное значение!).

Если вы строите совершенно новую вселенную, наполненную совершенно новыми существами, которые не имеют ничего общего с привычными нам, то тут не обойтись без собственной структуры исключений.

Например, если вы работаете с большой системой моделирования, которая предназначена для моделирования деятельности пиццерии, скорее всего, придется сформировать отдельную иерархию исключений.

Вы можете начать с определения общего исключения в качестве нового базового класса для любого другого специального исключения. Например, вот так:

```
class PizzaError(Exception):
def __init__(self, pizza, message):
Exception.__init__(message)
self.pizza = pizza
```

Примечание: *мы будем собирать более конкретную информацию, чем обычное исключение Exception, поэтому наш конструктор будет принимать два аргумента:*

- первый устанавливает пиццу в качестве субъекта процесса,
- а второй содержит более или менее точное описание проблемы.

Как видите, мы передаем второй параметр конструктору суперкласса и сохраняем первый в нашем собственном свойстве.

Более конкретная проблема (например, избыток сыра) может потребовать и более конкретного исключения. Можно вывести новый класс из готового класса PizzaError, как мы сделали здесь:

```
class TooMuchCheeseError(PizzaError):
def __init__(self, pizza, cheese, message):
PizzaError._init__(self, pizza, message)
self.cheese = cheese
```

Исключению TooMuchCheeseError нужно больше информации, чем обычному PizzaError, поэтому мы добавляем его в конструктор. Затем сохраняем имя cheese для дальнейшей обработки.

Посмотрите на код ниже.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese
```

```
def makePizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa',
                     'calzone']:
        raise PizzaError(pizza,
                         "no such pizza on the menu")

    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese,
                                 "too much cheese")

    print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110),
                 ('mafia', 20)]:
    try:
        makePizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

Мы объединили два ранее созданных исключения и использовали их для работы с небольшим фрагментом кода.

Один из них генерируется внутри функции makePizza(), если обнаруживается любая из этих двух ошибок: неправильный запрос на пиццу или запрос на слишком большое количество сыра.

Примечание:

- *если удалить ветку, которая начинается с* ***except*** *TooMuchCheeseError, это приведет к тому, что все исключения будут классифицированы как PizzaError;*
- *если удалить ветку, которая начинается с* except *PizzaError, то ошибки TooMuchCheeseError оста-*

нутся необработанными и приведут к завершению программы.

Предыдущее решение элегантное и эффективное, но имеет один важный недостаток. Из-за несколько ленивого способа объявления конструкторов, новые исключения нельзя использовать как есть, без полного списка необходимых аргументов.

Мы устраним этот недостаток, установив значения по умолчанию для всех параметров конструктора. Посмотрите:

```
class PizzaError(Exception):
    def __init__(self, pizza='uknown', message=''):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='uknown', cheese='>100',
                 message=''):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese

def makePizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa',
                     'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110),
                 ('mafia', 20)]:
    try:
        makePizza(pz, ch)
```

```
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

Теперь, если условия позволяют, можно использовать только имена классов.