Оглавление

Неделя 3

Объектно-ориентированное программирование

Помимо всего прочего, Python является объектно-ориентированным языком программирования --- в языке реализованы классы и три основные концепции ООП: наследование, полиморфизм, инкапсуляция. Код на Python-е можно писать, используя только функции. Однако, часто код программ на Python-е (а также код стандартной библиотеки) написан именно с использованием классов. На этой неделе вы познакомитесь с классами, наследованием классов, композицией, а также научитесь обрабатывать ошибки в программах.

3.1. Классы и объекты

3.1.1. Классы и экземпляры. Часть 1

Объектно-ориентированное программирование --- это особый способ организации кода. Часто говорят, что классы используют тогда, когда нужно отобразить реальные предметы на программный код. Отчасти это так, но в общем случае классы служат для объединения функционала, связанного общей идеей и смыслом, в одну сущность, у которой может быть свое внутреннее состояние, а также методы, которые позволяют модифицировать это состояние. Реальный пример класса: обертка над соединением к базе данных (состояние --- постоянное TCP-соединение с базой, методы класса предоставляют интерфейс доступа к соединению). Тем самым TCP соединение *инкапсулируется* внутри класса, а пользователю класса предоставляем удобный интерфейс доступа к данным. Много примеров классов можно найти в реализации игр жанра RPG. Квесты, монстры, игроки, предметы инвентаря --- всё это может быть классами со своими свойствами и возможностями.

Типы данных (такие как `int`, `float` и др.) в Python являются классами, структуры данных (`dict`, `list`, ...) --- это также классы.

```
print(int)

print(dict)
```

```
<class 'int'>
<class 'dict'>
```

Для того, что узнать, принадлежит ли объект к определённому типу (т.е. классу), существует стандартная функция `isinstance`:

```
num = 13
isinstance(num, int)
```

```
True
```

```
numbers = {}
isinstance(numbers, dict)
```

```
True
```

Итак, классы есть в стандартной библиотеке Python, но также пользователь может реализовывать собственные классы. Это делается с помощью ключевого слова `class`. (Классы в Python принято называть CamelCase-ом.) После этого ставится двоеточие и дальше идет блок пространства имен класса. Посмотрим на примере простейшего класса, который ничего не делает:

```
class Human:
    pass
```

Вместо слова `pass` можем вставить docstring:

```
class Robot:
    """Данный класс позволяет создавать роботов"""

    print(Robot)
```

```
<class '__main__.Robot'>
```

Посмотрим, какие методы есть у созданного объекта (и увидим, что их достаточно много):

```
print(dir(Robot))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

Предположим, что наш класс описывает планету (причём любую, абстрактную планету). *Экземплярами* класса будут являться конкретные планеты --- Земля, Марс и т.д.

```
class Planet:
    pass
```

Для того, чтобы создать экземпляр класса, обращаются к имени класса с помощью ( ):

```
planet = Planet()
print(planet)
```

```
<__main__.Planet object at 0x10e8722b0>
```

Мы получили не просто класс, а объект этого класса. Но ничто не мешает нам оперировать с классами как с объектами, так как всё в Python есть объект.

Давайте с помощью небольшого скрипта промоделируем создание Солнечной системы:

```
solar_system = []
for i in range(8):
    planet = Planet()
    solar_system.append(planet)

print(solar_system)
```

```
[<__main__.Planet object at 0x10e872780>, <__main__.Planet object at
0x10e8722b0>, <__main__.Planet object at 0x10e8727f0>,
<__main__.Planet object at 0x10e872828>, <__main__.Planet object at
0x10e872860>, <__main__.Planet object at 0x10e872898>,
<__main__.Planet object at 0x10e8728d0>, <__main__.Planet object at
0x10e872908>]
```

Важно отметить, что экземпляры класса --- это хэшируемые объекты (могут быть ключами словаря). Например, исправим предыдущий пример так, чтобы экземпляры класса `Planet` стали ключами словаря:

```
solar_system = {}
for i in range(8):
    planet = Planet()
    solar_system[planet] = True

print(solar_system)
```

```
{<__main__.Planet object at 0x10e872978>: True, <__main__.Planet
object at 0x10e872908>: True, <__main__.Planet object at
0x10e8727f0>: True, <__main__.Planet object at 0x10e872828>: True,
<__main__.Planet object at 0x10e872860>: True, <__main__.Planet
object at 0x10e872898>: True, <__main__.Planet object at
0x10e8729e8>: True, <__main__.Planet object at 0x10e872940>: True}
```

Чтобы назвать планеты нашей Солнечной системы, мы будем использовать один из магических методов класса --- метод `__init__`. Этот метод вызывается автоматически

при создании экземпляра класса. Первым аргументом метод `__init__` принимает ссылку на только что созданный экземпляр класса, далее могут идти другие аргументы. Внутри инициализатора мы можем по ссылке `self` установить так называемые атрибуты экземпляра. В данном случае мы ставим атрибут экземпляра `name` и присваиваем ему аргумент `name` --- имя планеты:

```
class Planet:

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

Мы можем обратиться к атрибуту класса, написав его через точку после названия экземпляра:

```
earth = Planet("Earth")
print(earth.name)
print(earth)
```

```
Earth
<__main__.Planet object at 0x10e8796d8>
```

Можно сделать так, чтобы `print(earth)` печатал имя планеты. Для этого есть магический метод `__str__`, позволяющий переопределить то, как будет печататься объект:

```
class Planet:

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

    def __str__(self):
        return self.name


earth = Planet("Earth")
print(earth)
```

```
Earth
```

Давайте назовём все планеты солнечной системы:

```
solar_system = []

planet_names = [
    "Mercury", "Venus", "Earth", "Mars",
    "Jupiter", "Saturn", "Uranus", "Neptune"
]

for name in planet_names:
    planet = Planet(name)
    solar_system.append(planet)

print(solar_system)
```

```
[<__main__.Planet object at 0x10477f160>, <__main__.Planet object at
0x10477f278>, <__main__.Planet object at 0x10477f198>,
<__main__.Planet object at 0x10477f1d0>, <__main__.Planet object at
0x10477f208>, <__main__.Planet object at 0x10477f240>,
<__main__.Planet object at 0x1048637b8>, <__main__.Planet object at
0x1048637f0>]
```

Несмотря на то, что мы переопределили метод `__str__`, внутри списка мы видим объекты в старом представлении. Чтобы отображать объекты в списке, Python использует другой магический метод --- `__repr__`, который мы тоже можем переопределить:

```
class Planet:

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

    def __repr__(self):
        return f"Planet {self.name}"

solar_system = []

planet_names = [
    "Mercury", "Venus", "Earth", "Mars",
    "Jupiter", "Saturn", "Uranus", "Neptune"
]

for name in planet_names:
    planet = Planet(name)
    solar_system.append(planet)

print(solar_system)
```

```
[Planet Mercury, Planet Venus, Planet Earth, Planet Mars, Planet
Jupiter, Planet Saturn, Planet Uranus, Planet Neptune]
```

Существует огромное множество других магических методов классов, про некоторые из которых мы поговорим позже.

Есть возможность в любой момент поменять значение атрибута класса:

```
mars = Planet("Mars")
print(mars)
```

```
Planet Mars
```

```
mars.name
```

```
'Mars'
```

```
mars.name = "Second Earth?"
mars.name
```

```
'Second Earth?'
```

Если обратиться к несуществующему атрибуту экземпляра, Python выдаст исключение AttributeError.

```
mars.mass
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
last)
<ipython-input-7-3c1085af8f48> in <module>()
----> 1 mars.mass

AttributeError: 'Planet' object has no attribute 'mass'
```

Кроме того, мы можем удалить атрибут из нашего экземпляра класса:

```
del mars.name

mars.name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
last)
<ipython-input-9-202092835a22> in <module>()
----> 1 mars.name

AttributeError: 'Planet' object has no attribute 'name'
```

3.1.2. Классы и экземпляры. Часть 2

Иногда нужно создать переменную, которая будет работать в контексте класса, но не будет связана с каждым конкретным экземпляром (т.е. будет относиться непосредственно к самому классу, а не к экземпляру). В этом примере `count` (счётчик планет) --- это атрибут класса:

```
class Planet:

    count = 0

    def __init__(self, name, population=None):
        self.name = name
        self.population = population or []
        Planet.count += 1
```

Можем напрямую обратиться к атрибуту класса через точку:

```
earth = Planet("Earth")
mars = Planet("Mars")

print(Planet.count)
```

```
2
```

Значение атрибута класса также можно получить, обращаясь к экземплярам:

```
mars.count
```

```
2
```

В этот момент Python видит, что внутри экземпляра класса такого атрибута нет, проверяет сам класс на наличие атрибута и находит его.

Когда счетчик ссылок на экземпляр класса достигает нуля (мы уже говорили про сборщик мусора в Python и то, что он использует счетчик ссылок), вызывается метод `__del__` экземпляра. Это также магический метод, который Python нам предоставляет возможность переопределить:

```
class Human:

    def __del__(self):
        print("Goodbye!")

human = Human()

del human
```

```
Goodbye!
```

Однако, на практике магический метод `__del__` рекомендуют не переопределять, так как нет гарантии, что по завершении работы интерпретатора Python он будет вызван. Лучше явно определить метод, который будет выполнять те действия, которые вам нужны (закрыть файл, разорвать сетевое соединение и т.д.).

Открыть словарь с атрибутами класса можно с помощь метода `__dict__`:

```
class Planet:
    """This class describes planets"""

    count = 1

    def __init__(self, name, population=None):
        self.name = name
        self.population = population or []


planet = Planet("Earth")

planet.__dict__
```

```
{'name': 'Earth', 'population': []}
```

Если мы добавим нашему экземпляру какой-нибудь атрибут, он появится в словаре атрибутов этого экземпляра:

```
planet.mass = 5.97e24

planet.__dict__
```

```
{'mass': 5.97e+24, 'name': 'Earth', 'population': []}
```

Словарь атрибутов есть также и у самого класса (обратите внимание на атрибуты `__doc__` и `count`):

```
Planet.__dict__
```

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Planet' objects>,
              '__doc__': 'This class describes planets',
              '__init__': <function __main__.Planet.__init__>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Planet'
objects>,
              'count': 1})
```

К элементам из *словаря атрибутов класса* можно обращаться как через имя класса, так и через имя какого-нибудь экземпляра этого класса:

```
Planet.__doc__
```

```
'This class describes planets'
```

```
planet.__doc__
```

```
'This class describes planets'
```

У экземпляра класса есть ещё много магических методов (например, метод __hash__, ведь экземпляры класса --- это хэшируемые объекты).

```
print(dir(planet))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'count', 'mass', 'name',
'population']
```

В любой момент мы можем узнать, какому классу принадлежит данный экземпляр:

```
planet.__class__
```

```
__main__.Planet
```

Конструктор экземпляра класса позволяет нам переопределить действия, которые происходят с экземпляром до его инициализации. На следующих неделях будет показан пример использования магического метода __new__, который как раз является конструктором экземпляра класса, в рамках использования метаклассов. Пока же посмотрим на простой пример класса с переопределённым методом __new__:

```
class Planet:

    def __new__(cls, *args, **kwargs):
        print("__new__ called")
        obj = super().__new__(cls)
        return obj

    def __init__(self, name):
        print("__init__ called")
        self.name = name

earth = Planet("Earth")
```

```
__new__ called
__init__ called
```

В этом примере в исходный метод `__new__` был добавлен вызов `print()`. В следующей строке метод `super()` возвращает родителя нашего класса, в данном случае это `object` --- класс, от которого наследуются все пользовательские классы в Python 3. Затем вызывается метод `__new__` класса `object`, который возвращает экземпляр класса. Этот экземпляр (который и является нашим классом) возвращается из функции.

То есть при вызове `Planet("Earth")` произошло примерно следующее:

```
planet = Planet.__new__(Planet, "Earth")

if isinstance(planet, Planet): # если __new__ вернул правильный класс
    Planet.__init__(planet, "Earth")
```

3.1.3. Методы. Часть 1

Методы --- это функции, которые действуют в контексте экземпляра класса. Таким образом, они могут менять состояние экземпляра, обращаясь к атрибутам экземпляра или делать любую другую полезную работу. В следующем примере мы создали класс `human`, у которого есть два атрибута: `name` и `age`, а также у нас есть класс планеты, у которой есть атрибут `name` и атрибут `population` (список людей, которые есть на планете):

```
class Human:

    def __init__(self, name, age=0):
        self.name = name
        self.age = age


class Planet:

    def __init__(self, name, population=None):
        self.name = name
        self.population = population or []

    def add_human(self, human):
        print(f"Welcome to {self.name}, {human.name}!")
        self.population.append(human)
```

Здесь мы объявили метод экземпляра `add human` --- просто функцию, которая принимает первым аргументом `self` (т.е. ссылку на экземпляр класса), а дальше --- любые другие аргументы (в случае выше это экземпляр класса `Human`). Обновим население планеты:

```
mars = Planet("Mars")

bob = Human("Bob")

mars.add_human(bob)
```

```
Welcome to Mars, Bob!
```

```
print(mars.population)
```

```
[<__main__.Human object at 0x10e416780>]
```

Ничто не мешает вызывать из методов другие методы. Посмотрим на примере:

```
class Human:
    def __init__(self, name, age=0):
        self._name = name
        self._age = age

    def _say(self, text):
        print(text)

    def say_name(self):
        self._say(f"Hello, I am {self._name}")

    def say_how_old(self):
        self._say(f"I am {self._age} years old")
```

Здесь объявляем класс Human, у которого названия атрибутов _name и _age начинаются с символа нижнего подчёркивания. Также у этого класса метод экземпляра _say, который также начинается с нижнего подчёркивания, а ещё два метода say_name и say_how_old, которые печатают, сколько человеку лет и какое у него имя. Символы нижнего подчёркивания показывают, что это внутренний метод, который вызывается только другими методами класса, но не должен вызываться пользователем. Такой механизм похож на private/protected атрибуты в других языках, однако в Python это всего лишь соглашение. Тем не менее, если атрибут либо метод названы с символа нижнего подчёркивания, то ими пользоваться не рекомендуется потому, что в дальнейших версиях той или иной библиотеки могут отказаться от этих атрибутов или методов, начинающихся с символа нижнего подчеркивания, либо поменять их поведение.

```
bob = Human("Bob", age=29)

bob.say_name()
bob.say_how_old()
```

```
Hello, I am Bob
I am 29 years old
```

```
# не рекомендуется!
print(bob._name)

# не рекомендуется!
bob._say("Whatever we want")
```

```
Bob
Whatever we want
```

Бывает, что вам нужно объявить метод, который не привязан к конкретному экземпляру, но в тоже время вовлекает в свою работу сам класс. Для этого существует стандартный декоратор `@classmethod` (метод класса). Например, создадим класс, который отображает какое-нибудь событие:

```
class Event:

    def __init__(self, description, event_date):
        self.description = description
        self.date = event_date

    def __str__(self):
        return f"Event \"{self.description}\" at {self.date}"

from datetime import date

event_description = "Рассказать, что такое @classmethod"
event_date = date.today()

event = Event(event_description, event_date)
print(event)
```

```
Event "Рассказать, что такое @classmethod" at 2017-07-09
```

Дополним этот класс методом класса. Метод `from_string` извлекает из пользовательского ввода информацию о некотором событии (например, такой метод может быть полезен при написании бота для мессенджера, который заносит события в календарь). Этот метод принимает на вход первым атрибутом сам класс `cls`, а затем ввод пользователя. Он возвращает экземпляр класса, таким образом, это альтернативный способ создания класса.

```
class Event:

    def __init__(self, description, event_date):
        self.description = description
        self.date = event_date

    def __str__(self):
        return f"Event \"{self.description}\" at {self.date}"

    @classmethod
    def from_string(cls, user_input):
        description = extract_description(user_input)
        date = extract_date(user_input)
        return cls(description, date)
```

Вообще, извлечение данных из сообщений --- это сложная задача. В данном примере мы используем заглушки:

```
def extract_description(user_string):
    return "открытие чемпионата мира по футболу"


def extract_date(user_string):
    return date(2018, 6, 14)
```

Протестируем:

```
event = Event.from_string("добавить в мой календарь открытие
чемпионата мира по футболу на 14 июня 2018 года")
print(event)
```

```
Event "открытие чемпионата мира по футболу" at 2018-06-14
```

Внутри стандартной библиотеки класс-методы тоже активно используются. Например, тип `dict` --- это класс, у которого есть метод `fromkeys`. `fromkeys` --- как раз метод класса, который принимает итерабельный объект и возвращает проинициализированный словарь:

```
dict.fromkeys("12345")
```

```
{'1': None, '2': None, '3': None, '4': None, '5': None}
```

3.1.4. Методы. Часть 2

Иногда нужно объявить метод в контексте класса, но этот метод не оперирует ни ссылкой на конкретный экземпляр класса, ни самим классом непосредственно (как в случае

@classmethod). В таком случае используют статический метод (@staticmethod). Пример:

```
class Human:

    def __init__(self, name, age=0):
        self.name = name
        self.age = age

    @staticmethod
    def is_age_valid(age):
        return 0 < age < 150
```

У статического метода нет аргументов self или class. К статическому методу можно обращаться по-разному:

```
# можно обращаться от имени класса
Human.is_age_valid(35)
```

```
True
```

```
# или от экземпляра:
human = Human("Old Bobby")
human.is_age_valid(234)
```

```
False
```

Функцию is_age_valid можно было объявить вне пространства имён класса. Где её объявлять --- это вопрос организации кода.

Ещё один важный концепт --- вычисляемые свойства класса (property). Property позволяют изменять поведение и выполнять какую-либо работу при обращении к атрибуту экземпляра, либо при изменении атрибута, либо при его удалении. Начнём немного издалека и определим класс Robot с атрибутом power:

```
class Robot:

    def __init__(self, power):
        self.power = power
```

Этим можно пользоваться так:

```
wall_e = Robot(100)
wall_e.power = 200
print(wall_e.power)
```

```
200
```

Предположим, вы заметили, что другие программисты, которые пользуются вашим классом `Robot`, иногда ставят ему отрицательную мощность (`power`):

```
wall_e.power = -20
```

Вам хотелось бы, чтобы в таком случае мощность на самом деле ставилась бы в ноль. Для этого можно отрефакторить класс и добавить метод экземпляра `set_power`, в котором и будет реализован такой функционал:

```
class Robot:

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

    def set_power(self, power):
        if power < 0:
            self.power = 0
        else:
            self.power = power

wall_e = Robot(100)
wall_e.set_power(-20)
print(wall_e.power)
```

```
0
```

Но в таком случае не только вам, но и всем программистам, использующим ваш класс, придётся менять код. Есть способ проще --- сделать `power` объектом `property()`. Далее объявим три метода и обернём их декораторами: `power.setter` (будет выполняться при изменении атрибута `power`) `power.getter` (выполнится при чтении атрибута `power`) и `power.deleter` (будет выполняться при удалении атрибута):

```
class Robot:

    def __init__(self, power):
        self._power = power

    power = property()

    @power.setter
    def power(self, value):
        # повторяет функционал старого метода set_power
        if value < 0:
            self._power = 0
        else:
            self._power = value

    @power.getter
    def power(self):
        return self._power

    @power.deleter
    def power(self):
        print("make robot useless")
        del self._power
```

```
wall_e = Robot(100)
wall_e.power = -20
print(wall_e.power)
```

```
0
```

```
del wall_e.power
make robot useless
```

Иногда единственное, что вам требуется --- это модифицировать чтение атрибута. Вам не нужно менять поведение при изменении значения атрибута/его удалении. В таком случае есть более короткая запись. Тогда можно обернуть метод декоратором `@property` и обращаться к нему просто с помощью `.power`:

```
class Robot:
    def __init__(self, power):
        self._power = power

    @property
    def power(self):
        # здесь могут быть любые полезные вычисления
        return self._power

wall_e = Robot(200)
wall_e.power
```

```
200
```

3.1.5. Пример на классы

Класс будет принимать на вход название города и иметь метод, который будет возвращать прогноз погоды в этом городе. (Этот класс можно будет расширять бесконечно, например, добавляя ему методы для получения грядущих событий в городе либо новостей, относящихся к нему.)

Создадим директорию для работы, перейдем в неё, создадим virtual env --- вы уже умеете это делать. После того, как виртуальное окружение создастся, мы должны его активировать. Нам понадобятся две сторонние библиотеки для работы нашей программы: `requests` и `python-dateutil`. `requests` нужна, чтобы делать http-запросы, а `python-dateutil` позволит преобразовывать даты, которые представлены в виде строки, в Python-овское представление (то есть в объекты модуля `datetime`).

Будем программировать в среде программирования Visual Studio Code (VSCode). Внутри VSCode мы откроем директорию, в которой только что создали виртуальное окружение. Это делается с помощью File -> Open. Следующим шагом мы должны указать, где находится интерпретатор Python для нашего проекта.

Чтобы открыть поиск команд, нажмём Shift+Cmd+P (*Примечание: или другое сочетание клавиш, подходящее для вашей ОС*), затем найдём комаду Select wordspace interpreter. Выбираем интерпретатор Python из нашего виртуального окружения. Внутри нашего проекта создадим файл `city.py`. Как только мы создали файл, Visual Studio Code предлагает нам установить linter-ы --- специальные утилиты, которые помогают нам программировать. В данном случае мы установим linter pep8, чтобы он следил за стилем нашего кода, а также linter flake8, чтобы помогать находить очевидные ошибки в коде.

Напишем `if __name__ == "__main__"` , чтобы запускать программу только тогда, когда она вызывается напрямую, а не импортируется. Внутри этой конструкции вызовем функцию `_main`. Назовем её с символа нижнего подчеркивания, указывая таким образом на то, что она является приватной и не должна использоватся из стороннего кода. Внутри функции `_main` напишем скелет будущей программы:

```
import pprint

def _main():
    city = CityInfo("Moscow") # будем смотреть погоду в Москве
    forecast = city.weather_forecast() # метод, который возвращает
                                       # прогноз погоды
    pprint.pprint(forecast) # напечатаем прогноз красиво
                            # с помощью PrettyPrinter


if __name__ == "__main__":
    _main()
```

flake8 должен подчеркнуть `CityInfo`, т.к. мы ещё его не объявили. Сделаем это:

```
class CityInfo:

    def __init__(self, city, forecast_provider=None):
        self.city = city

    def weather_forecast(self):
        pass # пока не знаем, откуда получать прогноз
```

Чтобы получить прогноз, воспользуемся Yahoo Weather API. На этой странице можно экспериментировать с http-запросами к API. Там уже составлен практически правильный запрос, нужно лишь поменять его окончание на `where text="moscow") and u="c"` (город Москва, температура в градусах Цельсия). После нажатия кнопки Test в окошке снизу появится ссылка на JSON с нужной нам информацией. Прогноз погоды на сегодняшнюю и следующие даты скрыт под тегом forecast.

Не будем писать код, делающий http-запросы, внутри метода `weather_forecast`. Мы создадим специальный класс `YahooWeatherForecast` с методом `get`, который будет принимать город. Это даёт нам возможность в будущем подменить этот класс другим. Если вдруг с API Yahoo что-то случится, мы сможем поменять только один компонент нашей программы, а весь остальной код продолжит работать.

Напишем новый класс, в URL-запрос подставим аргумент с названием города, а затем получим из JSON-а список словарей, содержащих дату и самую высокую температуру на этот день. Чтобы преобразовать дату из строки в питоновский формат, будем использовать функцию `parse()` из `dateutil.parser`:

```
import requests
from dateutil.parser import parse

class YahooWeatherForecast:

    def get(self, city):
        url = f"https://query.yahooapis.com/v1/public/yql?q="
        + f"select%20*%20from%20weather.forecast%20where%20w"
        + f"oeid%20in%20(select%20woeid%20from%20geo.places("
        + f"1)%20where%20text%3D%22{city}%22)%20and%20u%3D%2"
        + f"7c%27&format=json&env=store%3A%2F%2Fdatatables.o"
        + f"rg%2Falltableswithkeys"
        data = requests.get(url).json()
        forecast = []
        forecast_data =
data["query"]["results"]["channel"]["item"]["forecast"]
        for day_data in forecast_data:
            forecast.append({
                "date": parse(day_data["date"]),
                "high_temp": int(day_data["high"])
            })
        return forecast
```

Теперь вернемся к методу экземпляра `weather_forecast`, где мы обратимся к классу `YahooWeatherForecast` и вызовем его метод `get` для нужного города. Также расширим метод `__init__` --- он будет принимать необязательный параметр `weather_forecast` и ставить приватную переменную `_weather_forecast` экземпляру класса. Если этот параметр не был передан, по умолчанию будет использоваться `YahooWeatherForecast`:

```
class CityInfo:

    def __init__(self, city, forecast_provider=None):
        self.city = city.lower()
        self._forecast_provider = forecast_provider or
                                  YahooWeatherForecast()

    def weather_forecast(self):
        return self._forecast_provider.get(self.city)
```

Запустим нашу программу из консоли и получим прогноз погоды в Москве:

```
[{'date': datetime.datetime(2017, 9, 5, 0, 0), 'high_temp': 18},
 {'date': datetime.datetime(2017, 9, 6, 0, 0), 'high_temp': 28},
 {'date': datetime.datetime(2017, 9, 7, 0, 0), 'high_temp': 33},
 {'date': datetime.datetime(2017, 9, 8, 0, 0), 'high_temp': 33},
 {'date': datetime.datetime(2017, 9, 9, 0, 0), 'high_temp': 34},
```

```
{'date': datetime.datetime(2017, 9, 10, 0, 0), 'high_temp': 30},
{'date': datetime.datetime(2017, 9, 11, 0, 0), 'high_temp': 22},
{'date': datetime.datetime(2017, 9, 12, 0, 0), 'high_temp': 18},
{'date': datetime.datetime(2017, 9, 13, 0, 0), 'high_temp': 23},
{'date': datetime.datetime(2017, 9, 14, 0, 0), 'high_temp': 26}]
```

Можно сделать ещё лучше и смоделировать ситуацию, в которой нам будет приходить много http-запросов (которые иногда занимают значительное время), но мы справимся с этим с помощью кэширования данных. Также имя города будем передавать как аргумент скрипта:

```
import sys
import pprint
import requests
from dateutil.parser import parse


class YahooWeatherForecast:

    def __init__(self):
        self._city_cache = {}

    def get(self, city):
        if city in self._cached_data:
            return self._cached_data[city]
        url = f"https://query.yahooapis.com/v1/public/yql?q="
        + f"select%20*%20from%20weather.forecast%20where%20w"
        + f"oeid%20in%20(select%20woeid%20from%20geo.places("
        + f"1)%20where%20text%3D%22{city}%22)%20and%20u%3D%2"
        + f"7c%27&format=json&env=store%3A%2F%2Fdatatables.o"
        + f"rg%2Falltableswithkeys"
        print("sending HTTP request")
        data = requests.get(url).json()
        forecast = []
        forecast_data =
data["query"]["results"]["channel"]["item"]["forecast"]
        for day_data in forecast_data:
            forecast.append({
                "date": parse(day_data["date"]),
                "high_temp": int(day_data["high"])
            })
        self._cached_data[city] = forecast
        return forecast


class CityInfo:
```

```
    def __init__(self, city, forecast_provider=None):
        self.city = city.lower()
        self._forecast_provider = forecast_provider or
YahooWeatherForecast()

    def weather_forecast(self):
        return self._forecast_provider.get(self.city)


def _main():
    weather_forecast = YahooWeatherForecast()
    for i in range(5):
        city = CityInfo(sys.argv[1])
        forecast = city.weather_forecast()
        pprint.pprint(forecast)


if __name__ == "__main__":
    _main()
```

В данном примере мы применили композицию классов --- это очень хороший подход к написанию кода.

В данном примере мы не обрабатывали исключения, которые могли возникнуть. Во-первых, сайт Yahoo мог быть недоступен, во-вторых, каждый раз, когда мы обращались к ключу словаря, мог произойти `KeyError` (потому что Yahoo вернул нам невалидные данные). Также exception мог произойти, когда мы преобразовывали дату из строки в объект `datetime`. В скором будущем вы научитесь обрабатывать такие exception-ы.

3.2. Наследование

3.2.1. Наследование в Python

Наследование классов нужно для изменения поведения конкретного класса, а также расширения его функционала. Допустим, у нас есть готовый класс для домашнего питомца:

```
class Pet:
    def __init__(self, name=None):
        self.name = name
```

Давайте представим, что нам необходимо промоделировать процесс заселения планеты Земля домашними питомцами. Но нам неинтересно населять планету Земля непонятными питомцами, мы хотим населить её конкретно собаками, при этом не меняя класса

Pet. Поэтому давайте расширим этот класс.

Чтобы *унаследовать* класс "питомец", мы объявляем класс Dog, и в скобках указываем родительский класс Pet. Новый класс, созданный при помощи наследования, наследует все атрибуты и методы родительского класса. В данном случае класс "питомец" является родительским классом, также его называют базовым классом или суперклассом. А класс "собака" называется дочерним классом или классом-наследником.

Чтобы изменить поведение класса Dog, переопределим метод __init__ и добавим новый атрибут breed, в котором будем хранить породу собаки. В новом методе __init__ мы также вызовем инициализатор родительского класса, используя функцию super(). Кроме того, мы добавим новый метод say():

```
class Dog(Pet):
    def __init__(self, name, breed=None):
        super().__init__(name)
        self.breed = breed

    def say(self):
        return "{0}: waw".format(self.name)

dog = Dog("Шарик", "Доберман")
print(dog.name)
```

```
Шарик
```

```
print(dog.say())
```

```
Шарик: waw!
```

В Python разрешено наследование от нескольких классов предков, или как это ещё называется, множественное наследование. Очень часто этот приём используется для реализации классов-примесей. Предположим, что нам необходимо экспортировать данные о наших объектах (собачках) в формате json, чтобы хранить эти данные на жестком диске, либо передавать по сети. Мы можем решить подобную задачу при помощи классов-примесей и множественного наследования.

Объявим класс ExportJSON, реализуем метод, который экспортирует данные в формате json, и создадим новый класс, который называется ExDog --- он будет наследоваться от класса "собака" и нашего нового класса-примеси
ExportJSON:

```
import json

class ExportJSON:
    def to_json(self):
        return json.dumps({
            "name": self.name,
            "breed": self.breed
        })

class ExDog(Dog, ExportJSON):
    pass
```

```
{"name": "\u0411\u0435\u043b\u043a\u0430", "breed":
"\u0414\u0432\u043e\u0440\u043d\u044f\u0436\u043a\u0430"}
```

С одной стороны, это кажется удобным и гибким, однако множественное наследование и использование большого количества примесей ухудшает читаемость кода. Поэтому не стоит сильно увлекаться и создавать большое количество классов-примесей.

Любой класс в Python является потомком класса `object`. Мы можем легко убедиться в этом, если попробуем использовать функцию issubclass:

```
>>> issubclass(int, object)
True
>>> issubclass(Dog, object)
True
>>> issubclass(Dog, Pet)
True
>>> issubclass(Dog, int)
False
```

Также при помощи функции isinstance мы можем проверять, является ли конкретный объект экземпляром какого-то класса:

```
>>> isinstance(dog, Dog)
True
>>> isinstance(dog, Pet)
True
>>> isinstance(dog, object)
True
```

При помощи наследования Python позволяет выстраивать достаточно сложные иерархии классов. Мы построили довольно сложную иерархию: есть класс `ExDog`, который мы создали при помощи множественного наследования от класса `Dog` и класса-примеси `ExportJSON`. В свою очередь, класс `Dog` наследуется от класса "питомец", и все остальные классы наследуются от класса `object`. Если мы попробуем создать экземпляр класса `ExDog` и обратиться к атрибуту `name`, то как же Python будет искать этот атрибут в существующей иерархии классов?

Для этого в Python существует так называемый Method Resolution Order, или порядок разрешения методов, и это отдельная тема для изучения. Однако все, что вам нужно знать --- это порядок, в котором Python ищет нужный атрибут или метод. Этот порядок можно получить при помощи атрибута `__mro__`. Он говорит о том, что если мы попробуем обратиться к атрибуту `name`, Python будет искать сначала в классе `ExDog`, затем `Dog`, и после того, как он обратится к классу `Pet`, нужный атрибут `name` будет найден. Данный список ещё называется линеаризацией класса, то есть Python последовательно ищет любые атрибуты и методы в этом списке. Если он пройдется по всему списку и не найдет нужный атрибут или метод, то будет сгенерировано исключение `AttributeError`.

```
#     object
#     /   \
#    /     \
#  Pet    ExportJSON
#   |       /
#  Dog     /
#    \   /
#    ExDog

# Method Resolution Order
>>> ExDog.__mro__
(<class '__main__.ExDog'>, <class '__main__.Dog'>,
 <class '__main__.Pet'>, <class '__main__.ExportJSON'>,
 <class 'object'>)
```

В самом начале, когда мы создавали класс `Dog`, мы рассматривали вызов инициализатора базового класса с помощью функции `super()` без параметров. Однако в Python можно обратиться не только к базовому классу, но и к любому методу в существующей иерархии. Вызов функции `super()` без параметров равносилен тому, что мы указали сам класс и передали туда объект `self`. Однако если необходимо вызвать метод конкретного класса, то в функцию `super()` надо передать его *родителя*.

Итак, если мы создадим новый класс `WoolenDog` и захотим обратиться к инициализатору класса "питомец", то нам необходимо в функции `super()` указать класс-родитель. Далее попробуем создать объект класса `WoolenDog` и обратиться к атрибуту `breed`:

```
class ExDog(Dog, ExportJSON):
    def __init__(self, name, breed=None):
        # вызов метода по MRO
        super().__init__(name, breed)
        # то же самое, что
        # super(ExDog, self).__init__(name)

class WoolenDog(Dog, ExportJSON):
    def __init__(self, name, breed=None):
        # явное указание метода конкретного класса
        super(Dog, self).__init__(name)
        self.breed = "Шерстяная собака породы {0}".format(breed)

dog = WoolenDog("Жучка", breed="Такса")
print(dog.breed)
```

```
Шерстяная собака породы Такса
```

Также в Python существуют приватные атрибуты. Для того чтобы создать приватный атрибут, необходимо его имя записать через два символа нижнего подчёркивания. Предположим, что атрибут `breed` мы решили сделать приватным, тогда в самом классе к нему можно обращаться так же, а вот для классов-наследников этот атрибут будет уже недоступен:

```
class Dog(Pet):
    def __init__(self, name, breed=None):
        super().__init__(name)
        self.__breed = breed

    def say(self):
        return "{0}: waw!".format(self.name)

    def get_breed(self):
        return self.__breed

class ExDog(Dog, ExportJSON):
    def get_breed(self):
        return "порода: {0} - {1}".format(self.name, self.__breed)
```

Можете проверить, что такой вызов приведёт к `AttributeError`:

```
>>> dog = ExDog("Фокс", "Мопс")
>>> dog.get_breed()
```

Можно распечатать внутренний атрибут `__dict__`, который нам покажет все атрибуты нашего созданного объекта. Мы видим, что Python автоматически изменил имя приватного агрумента.

```
>>> dog.__dict__
{'name': 'Фокс', '_Dog__breed': 'Мопс'}
```

Мы можем обратиться к аргументу по новому имени `_Dog__breed` Таким образом, Python всё же позволяет обращаться к приватным атрибутам класса вне самого класса, однако не стоит этим увлекаться.

3.2.2. Композиция классов, пример

В Python существует альтернативный подход наследованию --- это *композиция*. Вспомним пример из предыдущего видео. У нас был класс "питомец", мы от него унаследовали класс `Dog`. Затем мы захотели, чтобы наши объекты классов "собака" могли выполнять экспорт данных, и мы ввели класс-примесь `ExportJSON`. После этого наш финальный класс `ExDog` использовал множественное наследование и наследовался от класса "собачка" и `ExportJSON`. Если бы нам пришлось экспортировать данные не только в формате json, но в другом формате (например, XML), нам бы понадобился ещё один класс-примесь:

```
class ExportJSON:
    def to_json(self):
        pass

class ExportXML:
    def to_xml(self):
        pass

class ExDog(Dog, ExportJSON, ExportXML):
      pass

dog = ExDog("Фокс", "мопс")
dog.to_xml()
dog.to_json()
```

Давайте представим, что нам нужно будет добавлять еще несколько методов для экспорта данных. В таком случае нам постоянно придется изменять код нашего класса `ExDog`, дописывая туда новые классы-примеси, что может слишком усложнить наш код. Именно для того, чтобы этого избежать, используют композицию классов.

Создадим новый класс `PetExport`:

```
class PetExport:
    def export(self, dog):
        # не будем создавать экземпляров класса,
        # он нужен только для наследования
        raise NotImplementedError


class ExportXML(PetExport):
    def export(self, dog):
        pass


class ExportJSON(PetExport):
    def export(self, dog):
        pass
```

Вспомним про классы, которые у нас уже были:

```
class Pet:
    def __init__(self, name):
        self,name = name


class Dog(Pet):
    def __init__(self, name, breed=None):
        super().__init__(name)
        self.breed = breed
```

Теперь снова создадим класс ExDog, уже без использования множественного наследования:

```
class ExDog(Dog):
    def __init__(self, name, breed=None, exporter=None):
        super().__init__(name, breed)
        self._exporter = exporter

    def export(self):
        return self._exporter.export(self)
```

Давайте попробуем создать экземпляр нашего класса ExDog. Предположим, мы хотим, чтобы объект этого класса умел экспортировать свои данные в xml. Давайте передадим нужный exporter. Обратите внимание, что при использовании композиции нужный объект создается именно в момент выполнения конкретной программы:

```
dog = ("Шарик", "Дворняга", exporter=ExportXML())
dog.export()
```

Осталось реализовать только методы для экспорта в начальной иерархии классов. С json все просто --- используем модуль `json` и метод `dumps`:

```
class ExportJSON(PetExport):
    def export(self, dog):
        return json.dumps({
            "name": dog.name,
            "breed": dog.breed,
        })
```

Давайте реализуем теперь метод export в классе ExportXML:

```
class ExportXML(PetExport):
    def export(self, dog):
        return """<?xml version="1.0" encoding="utf-8"?>
<dog>
  <name>{0}</name>
  <breed>{1}</breed>
</dog>""".format(dog.name, dog.breed)
```

Однако, неудобно каждый раз задавать метод для экспорта. Давайте немного изменим наш класс `ExDog` и зададим метод для экспорта по умолчанию. Также сделаем проверку на то, является ли переданный объект экземпляром класса `PetExport` и может ли он вообще выполнять экспорт данных. Для этого мы можем воспользоваться проверкой `isinstance`.

Что делать, если нам передали объект, который не может выполнять экспорт? Давайте сгенерируем исключение (в данном случае, `ValueError`) --- это будет означать, что программа дальше не сможет продолжить свою работу и будет остановлена.

```
class ExDog(Dog):
    def __init__(self, name, breed=None, exporter=None):
        super().__init__(name, breed)

        self._exporter = exporter or ExportJSON()

        if not isinstance(self._exporter, PetExport):
            raise ValueEror("bad export instance value", exporter)

    def export(self):
        return self._exporter.export(self)
```

Теперь, если нам нужно будет добавить в этот код новый метод для экспорта, мы с легкостью сможем сделать это. Просто объявим новый класс, добавим его в существующую иерархию для экспорта, а класс `ExDog` менять не будем. А экспортировать в различные форматы мы сможем легко и удобно в итоговой программе, подставив нужный exporter

или создав его.

3.3. Работа с ошибками

3.3.1. Классы исключений и их обработка

В предыдущих видео мы несколько раз упоминали про исключения в Python. Для начала давайте попробуем вызвать исключение и посмотрим, что при этом произойдет:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

При делении на 0 возникло исключение, и в этом случае на стандартный вывод печатается информация о его типе (в нашем случае это `ZeroDivisionError`), дополнительная информация об исключении, а также стек вызовов.

В этом примере стек вызовов очень небольшой. Когда исключение возникает в реальной программе, в стеке вызовов мы можем увидеть всю последовательность вызовов функций, которая привела к исключению.

В Python есть два больших типа исключений. Первый --- это исключения из стандартной библиотеки в Python, второй тип исключений --- это пользовательские исключения. Они могут быть сгенерированы и обработаны самим программистом при написании программ на Python. Давайте посмотрим на иерархию исключений в стандартной библиотеке Python. Все исключения наследуются от базового класса BaseException:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- AssertionError
      +-- AttributeError
      +-- LookupError
           +-- IndexError
           +-- KeyError
      +-- OSError
      +-- SystemError
      +-- TypeError
      +-- ValueError
```

Существуют несколько системных исключений, например, `SystemExit` (генерируется, если мы вызвали функцию `OSExit`), `KeyboardInterrupt` (генерируется, если мы нажали сочетание клавиш Ctrl + C) и так далее. Все остальные исключения генерируется от базового класса `Exception`. Именно от этого класса нужно порождать свои исключения.

Давайте посмотрим и обсудим некоторые исключения из стандартной библиотеки, такие как, например, `AttributeError`, `IndexError`, `KeyError`, `TypeError`, и попробуем их вызвать.

Обращаемся к несуществующему атрибуту класса, чтобы получить `AttributeError`:

```
>>> class MyClass():
...     pass
...
>>> obj = MyClass()
>>> obj.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute 'foo'
```

Ищем значение по несуществующему ключу в словаре, получаем `KeyError`:

```
>>> d = {"foo": 1}
>>> d["bar"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'bar'
```

Обратимся к несуществующему индексу в списке, получим `IndexError`:

```
>>> l = [1,2]
>>> l[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Попробуем преобразовать строку в целое число, видим `ValueError`:

```
>>> int("asdf")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'asdf'
```

Наконец, получим `TypeError` при попытке сложить целое число со строкой:

```
>>> 1 + "10"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Если исключение сгенерировано, Python-интерпретатор остановит свою работу и на экран будет выведен стек вызовов и информация о типе исключений. Чтобы программа не останавливала работу, можно обработать исключение при помощи блока `try except`. То есть код, который потенциально может генерировать исключения, мы обрамляем в блок `try except`, и при генерации исключений управление будет передано в блок `except`. Таким образом можно отловить все исключения, которые генерируются в блоке `try except`:

```
try:
    1 / 0
except Exception:
    print("Ошибка")
```

В блоке `except` можно указать тип исключения (в данном случае `Exception`), чтобы отлавливать исключения всех типов, у которых класс этого типа является родителем. В целом неправильно ждать любые исключения, и это может привести к непредвиденным сюрпризам работы вашей программы. Покажем на примере:

```
while True:
    try:
        raw = input("введите число: ")
        number = int(raw)
        break
    except:
        # не указали тип исключения, значит, обрабатываем все
        print("некорректное значение")
```

Если мы запустим эту программу в командной строке и попробуем завершить её сочетанием Ctrl-C, мы получим ответ `некорректное значение`, что отличается от желаемого поведения программы. Поэтому всегда нужно обрабатывать конкретные исключения. Перепишем нашу программу в соотвествии с этим правилом:

```
while True:
    try:
        raw = input("введите число: ")
        number = int(raw)
        break
    except ValueError:
        print("некорректное значение")
```

Также у блока `try except` может быть блок `else`. Блок `else` вызывается в том случае, если никакого исключения не произошло:

```
while True:
    try:
        raw = input("введите число: ")
        number = int(raw)
    except ValueError:
        print("некорректное значение!")
    else:
        break
```

Если нам нужно обработать несколько исключений, мы можем использовать несколько блоков except и указать разные классы для обработки исключения. Причем в каждом блоке except может быть свой собственный обработчик:

```
while True:
    try:
        raw = input("введите число: ")
        number = int(raw)
        break
    except ValueError:
        print("некорректное значение!")
    except KeyboardInterrupt:
        print("выход")
        break
```

Если обработчик исключений выглядит одинаково, то несколько исключений можно передать в виде списка в блок except:

```
total_count = 100_000
while True:
    try:
        raw = input("введите число: ")
        number = int(raw)
        total_count = total_count / number
        break
    except (ValueError, ZeroDivisionError):
        print("некорректное значение!")
```

Бывает удобно пользоваться иерархией классов исключений. Посмотрим на два класса исключений IndexError и KeyError и их родителя LookupError:

```
# +-- LookupError
#      +-- IndexError
#      +-- KeyError

>>> issubclass(KeyError, LookupError)
True
```

```
>>> issubclass(IndexError, LookupError)
True
```

Благодаря наследованию мы можем обрабатывать сразу обе ошибки в следующей программе, которая получает надписи на футболках из базы данных, где они отсортированы по цвету:

```
database = {
    "red": ["fox", "flower"],
    "green": ["peace", "M", "python"]
}

try:
    color = input("введите цвет: ")
    number = input("введите номер по порядку: ")

    label = database[color][int(number)]
    print("вы выбрали:", label)
# except (IndexError, KeyError):
except LookupError:
   print("Объект не найден")
```

Также у исключений есть дополнительный блок `finally`. Рассмотрим проблему. Например, мы открываем файл, читаем строки, обрабатываем эти строки, и в процессе работы нашей программы возникает исключение, которое мы не ждем. В таком случае файл закрыт не будет. Открытые файловые дескрипторы могут накапливаться, чего не следует допускать. Таким же образом могут накапливаться открытые сокеты или не освобождаться память. Для контроля таких ситуаций существуют, во-первых, контекстные менеджеры, а во-вторых, можно использовать блок `finally` в исключениях.

Мы пишем блок `finally` и вызываем метод `close()` для нашего объекта `f`. Возникло исключение или не возникло --- блок `finally` будет выполнен и файл закроется:

```
f = open("/etc/hosts")
try:
    for line in f:
        print(line.rstrip("\n"))
        1 / 0
except OSError:
    print("ошибка")
finally:
    f.close()
```

3.3.2. Генерация исключений

Для получения доступа к объекту исключений, нам необходимо воспользоваться конструкцией `except ... as err`. В следующем примере, если будет сгенерировано исключение `OSError`, то сам объект исключений будет связан с переменной `err` и эта переменная `err` будет доступна в блоке `except`. У каждого объекта типа исключений есть свои свойства, например, `errno` и `srterror` --- это строковое описание ошибки и код ошибки. При помощи этих атрибутов можно получать доступ и обрабатывать исключения нужным вам образом.

```
try:
    with open("/file/not/found") as f:
        content = f.read()
except OSError as err:
    print(err.errno, err.strerror)
```

При вызове исключения можно передать ему любые аргументы, которые потом будут доступны как атрибут `args`. Предположим, продолжая предыдущий пример, что в момент генерации исключения `ValueError` мы передали туда строку и имя файла. Теперь в блоке `except` обратиться к атрибуту `args` объекта исключения --- это и будет список наших параметров:

```
import os.path

filename = "/file/not/found"
try:
    if not os.path.exists(filename):
        raise ValueError("файл не существует", filename)
except ValueError as err:
    message, filename = err.args[0], err.args[1]
    print(message, code)
```

Иногда нам может потребоваться стек вызовов при обработке исключения. Стек вызовов можно получить при помощи модуля `traceback`, вызвав метод `print_exc`:

```
import traceback

try:
    with open("/file/not/found") as f:
        content = f.read()
except OSError as err:
    trace = traceback.print_exc()
    print(trace)
```

Как вы, возможно, уже догадались, исключение генерируется инструкцией `raise`. Для генерации исключения мы должны написать `raise` и указать класс исключения. Также можно указывать не только класс, но и объект исключения, указав ему дополнительные

свойства. Как уже было сказано, к этим свойствам потом можно будет обратится через объект исключения при помощи атрибута `args`. В следующем примере мы проверяем, что пользователь ввёл число, в противном случае генерируем исключение и затем в блоке `except` обрабатываем его:

```
try:
    raw = input("введите число: ")
    if not raw.isdigit():
        raise ValueError
except ValueError:
    print("некорректное значение!")
```

```
try:
    raw = input("введите число: ")
    if not raw.isdigit():
        raise ValueError("плохое число", raw)
except ValueError as err:
    print("некорректное значение!", err)
```

Иногда, поймав исключение, мы хотим делегировать обработку этого исключения другим функциям --- тем, который вызвали данную функцию. Для этого нужно использовать инструкцию `raise` без параметров. В следующем примере в случае некорректного числа на экран выводится надпись: "Некорректное значение", и при помощи инструкции `raise` мы делегируем исключение на уровень выше. Если мы попробуем исполнить данный код, интерпретатор покажет на стандартный вывод информации об этом исключении и прекратит работу программы.

```
try:
    raw = input("введите число: ")
    if not raw.isdigit():
        raise ValueError("плохое число", raw)
except ValueError as err:
    print("некорректное значение!", err)
    # делегирование обработки исключения
    raise
```

Говоря об исключениях, нельзя не затронуть инструкцию `assert`. По умолчанию, если выполнить инструкцию `assert` с логическим выражением, результат которого равен `True`, ничего не произойдет. Но если попробовать выполнить инструкцию `assert` с логическим выражением, которое равно `False`, то будет сгенерировано исключение `AssertionError`. Также мы можем передать дополнительную строку в сам объект `AssertionError`.

```
>>> assert True
>>> assert 1 == 0
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AssertionError
>>>
>>> assert 1 == 0, "1 не равен 0"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: 1 не равен 0
```

Предположим, у нас есть функция `get_user_by_id`, которая ищет пользователя по численному идентификатору.Убедиться в том, что нам действительно передали число, можно при помощи `assert` и функции `isinstance`. Если `isinstance` вернёт `False`, то будет сгенерирована `AssertionError`:

```
def get_user_by_id(id):
    assert isinstance(id, int), "id должен быть целым числом"

    print("выполняем поиск")

if __name__ == "__main__":
    get_user_by_id("foo")
```

Исключения `AssertionError` предназначены скорее для программистов. При написании наших программ на этапе разработки мы должны видеть, что делаем что-то не так (например, передали в функцию некорректное значение). Не нужно, например, обрабатывать пользовательский ввод и пытаться обработать исключение `AssertionError` блоком `try except`. Если таких мест будет очень много, то это затронет и производительность нашей программы.

Однако, есть возможность отключить все инструкции `assert` при помощи флага `-0`. Тогда `AssertionError` не будет сгенерирована. Этим и отличаются исключения `AssertionError` от обычных пользовательских исключений и исключений стандартной библиотеки.

Несмотря на то, что механизм обработки исключений очень удобен, он не "бесплатен". Рассмотрим пример: в цикле мы обращаемся к несуществующему ключу словаря и каждый раз при этом обращении генерируется исключение `KeyError`. Попробуем при помощи `timeit` замерить, сколько это займет:

```
%%timeit
my_dict = {"foo": 1}
for _ in range(1000):
    try:
        my_dict["bar"]
    except KeyError:
        pass
```

```
1000 loops, best of 3: 511 µs per loop
```

Если мы попробуем реализовать ту же самую задачу, но без использования исключений (например, при помощи конструкции `in`), мы увидим, что результат отличается на порядок (а иногда он может отличаться и на несколько порядков):

```
%%timeit
my_dict = {"foo": 1}
for _ in range(1000):
    if "bar" in my_dict:
        _ = my_dict["bar"]
```

```
10000 loops, best of 3: 78.3 µs per loop
```

Вывод: если исключения генерируются в программе очень часто, возможно, следует изменить код вашей программы, чтобы не потерять в производительности.

3.3.3. Исключения в requests, пример

Рассмотрим теперь, как устроены *пользовательские исключения* в библиотеке `requests`.

Напишем для примера программу, которая на вход принимает адрес в интернете, выполняет скачивание содержимого страницы и выводит его на стандартный вывод. Эта программа будет обрабатывать исключения, которые могут возникнуть.

Можно посмотреть, какие исключения вообще есть в библиотеке `requests`. Для этого импортируем её и посмотрим, где находятся её файлы:

```
>>> import requests
>>> requests.__file__
'/usr/lib/python3/dist-packages/requests/__init__.py'
```

В этой же директории можем найти файл `exceptions.py`. Открыв его, увидим, что все исключения наследуют базовый класс `RequestException`.

Напишем каркас программы:

```
import sys
import requests

url = sys.argv[1]

response = requests.get(url)

print(response.content)
```

Давайте зададим аргумент `timeout` для метода `requests.get()` и напишем обработку исключения `Timeout` (мы получим его, если не получим ответ в течение указанного времени). Также давайте обрабатывать исключение `HTTPError` (неправильный адрес) и все остальные исключения из библиотеки `requests`

```
try:
    response = requests.get("https://github.com/not_found",
timeout=30)
    response.raise_for_status()
except requests.Timeout:
    print("ошибка timeout, url:", url)
except requests.HTTPError as err:
    code = err.response.status_code
    print("ошибка url: {0}, code: {1}".format(url, code))
except requests.RequestException:
    print("ошибка скачивания url: ", url)
else:
    print(response.content)
```

Итак, мы написали программу, которая скачивает URL и обрабатывает все нужные исключения, которые могут возникнуть при работе с библиотекой `requests`.