

Тема 4. ООП в Python

(продолжение)

Свойства классов

- **Свойство** – это идентификатор, определенный внутри класса, через который в дальнейшем будут производиться операции получения, изменения и удаления связанного с ним атрибута.
- Свойства выглядят как обычные атрибуты (поля) класса, но при их чтении вызывается геттер (getter), при записи – сеттер (setter), а при удалении – делитер (deleter)

Свойства классов

- **Использование свойств позволяет устанавливать определять свойства доступные только для чтения, записи или для чтения и записи.**
- **Существует два способа создания свойств в Python:**
 - **с помощью функции `property()`**
 - **С помощью декоратора `@property`**

Свойства классов

- Создание свойств с помощью функции `property()`

- Формат функции `property()`

`<prop_name> = property(fget=None, fset=None, fdel=None, doc=None))`

где **fget** – имя метода для чтения атрибута,

fset – имя метода для записи атрибута,

fdel – имя метода для удаления атрибута

Свойства классов

- Создание свойств с помощью функции `property()`

```
class Horse:
```

```
def __init__(self, name, horsehair):
```

```
    self._name = name
```

```
def get_name(self): # геттер для поля self._name
```

```
    return self._name
```

```
def set_name(self, value): # сеттер для поля self._name
```

```
    self._name = value
```

```
def del_name(self): # делитер для поля self._name
```

```
    del self._name
```

```
name = property(get_name, set_name, del_name, "Name of the Horse")
```

Свойства классов

- Создание свойств с помощью функции `property()`

Пример (продолжение)

```
horse = Horse("Яша", "серый")  
print(horse.name) # здесь вызывается метод get_name  
horse.name = "Красавец" # здесь вызывается метод set_name  
print(horse.name)
```

Свойства классов

- Создание свойств с помощью функции `property()`
- *Пример 2*

```
class Horse:
    def __init__(self, name, horsehair):
        self._name = name

    def get_name(self):
        return self._name
    def del_name(self):
        del self._name
```

```
name = property(fget=get_name, fdel=del_name, doc="Name of the Horse")
```

*Свойство `name` доступно
только для чтения и
удаления.*

*Попытка записи в него
приведет к ошибке!*

Свойства классов

- Создание свойств с помощью декоратора `@property`
- При работе через декоратор `@property` преобразует следующий за ним метод в геттер атрибута
- Для задания сеттера и делитера используются декораторы свойства `setter` и `delete`:
 - `@свойство.setter` или `@свойство.deleter`
- **ВАЖНО:** названия всех методов (геттеров, сеттеров и делитеров) должно совпадать с именем свойства!

СВОЙСТВА КЛАССОВ

```
class Horse:
    def __init__(self, name, horsehair):
        self._name = name

    @property
    def name(self):
        """Name of Horse property"""
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @name.deleter
    def name(self):
        del self._name
```

*Пример создание
свойств с помощью
декоратора @property*

*Название всех
методов совпадает с
именем свойства!*

СВОЙСТВА КЛАССОВ

```
class Horse:
```

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

```
    @property
```

```
    def name(self):  
        """Name of Horse property"""
```

```
        return self._name
```

```
    @name.setter
```

```
    def name(self, value):  
        self._name = value
```

```
    @name.deleter
```

```
    def name(self):  
        del self._name
```

*Пример создание
свойств с помощью
декоратора @property*

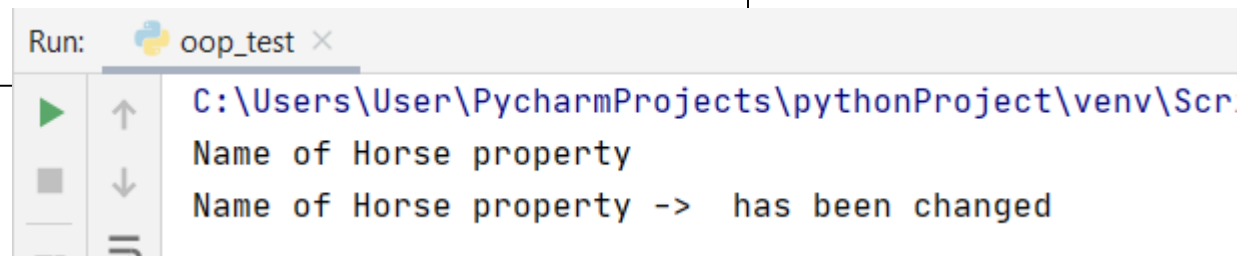
Использование свойства name:

```
horse = Horse("Яша", "серый")  
print(horse.name)  
horse.name = "Красавец"  
print(Horse.name.__doc__)
```

Свойства классов

- Создание свойств с помощью декоратора `@property`
- Возвращаемый объект-свойство также имеет атрибуты `fget`, `fset`, `fdel`, соответствующие аргументам конструктора метода `property()`.
- Начиная с версии 3.5, строки документации у свойств доступны для записи.

```
print(Horse.name.__doc__)  
Horse.name.__doc__ += " -> has been changed"  
print(Horse.name.__doc__)
```



Наследование

Наследование

- **Базовым классом для всех объектов в Python является класс `object`**
- **Исключение составляют объекты-исключения: они порождены от `BaseException`**
- **Базовый класс должен быть определен в области видимости, в которой находится определение производного класса.**

Наследование

- Синтаксис:

Простое наследование:

```
class <ИМЯ_НОВОГО_класса>(<имя_родителя>):  
    # тело класса
```

- *Разрешение имен:* если запрошенный атрибут не найден в текущем классе, поиск продолжается в базовом классе.

Наследование

- **Функции `issubclass` и `isinstance`**
- **`isinstance()` – позволяет проверить, является ли «что-то» объектом определенного класса или нет**
- **`issubclass()` – функция проверяет, является ли один класс потомком другого или нет**

Наследование

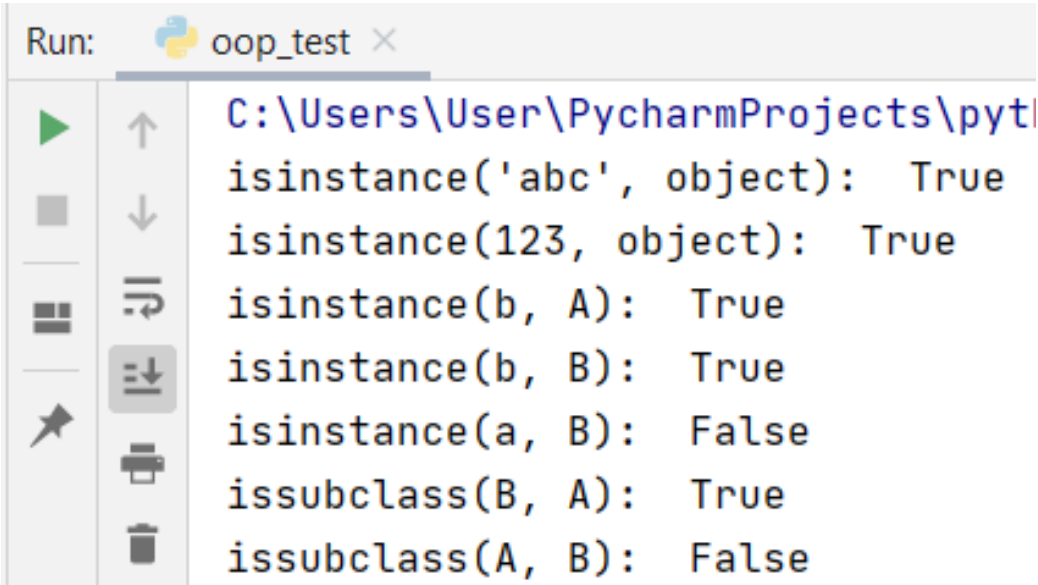
- **Функции `issubclass` и `isinstance`**

```
class A: pass  
class B(A): pass
```

```
print("isinstance('abc', object): ", isinstance('abc', object))  
print("isinstance(123, object): ", isinstance(123, object))
```

```
b = B()  
a = A()
```

```
print("isinstance(b, A): ", isinstance(b, A))  
print("isinstance(b, B): ", isinstance(b, B))  
print("isinstance(a, B): ", isinstance(a, B))  
print("issubclass(B, A): ", issubclass(B, A))  
print("issubclass(A, B): ", issubclass(A, B))
```



```
Run: oop_test x  
C:\Users\User\PycharmProjects\pytl  
isinstance('abc', object): True  
isinstance(123, object): True  
isinstance(b, A): True  
isinstance(b, B): True  
isinstance(a, B): False  
issubclass(B, A): True  
issubclass(A, B): False
```


Наследование

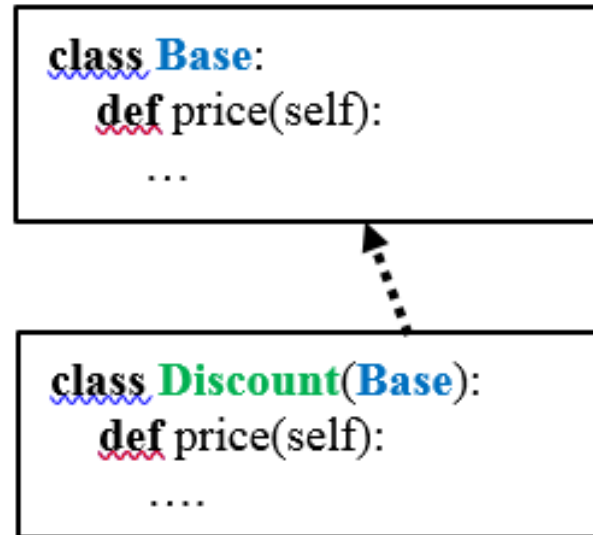
- По умолчанию, методы дочернего класса перекрывают одноименные методы базового класса.
- Для доступа к атрибутам базового класса из дочернего необходимо предварительно указать **имя базового класса** или вызвать метод **super()**.

Наследование

- Пример

```
class Base:  
    def price(self):  
        print("in Base")  
        return 10
```

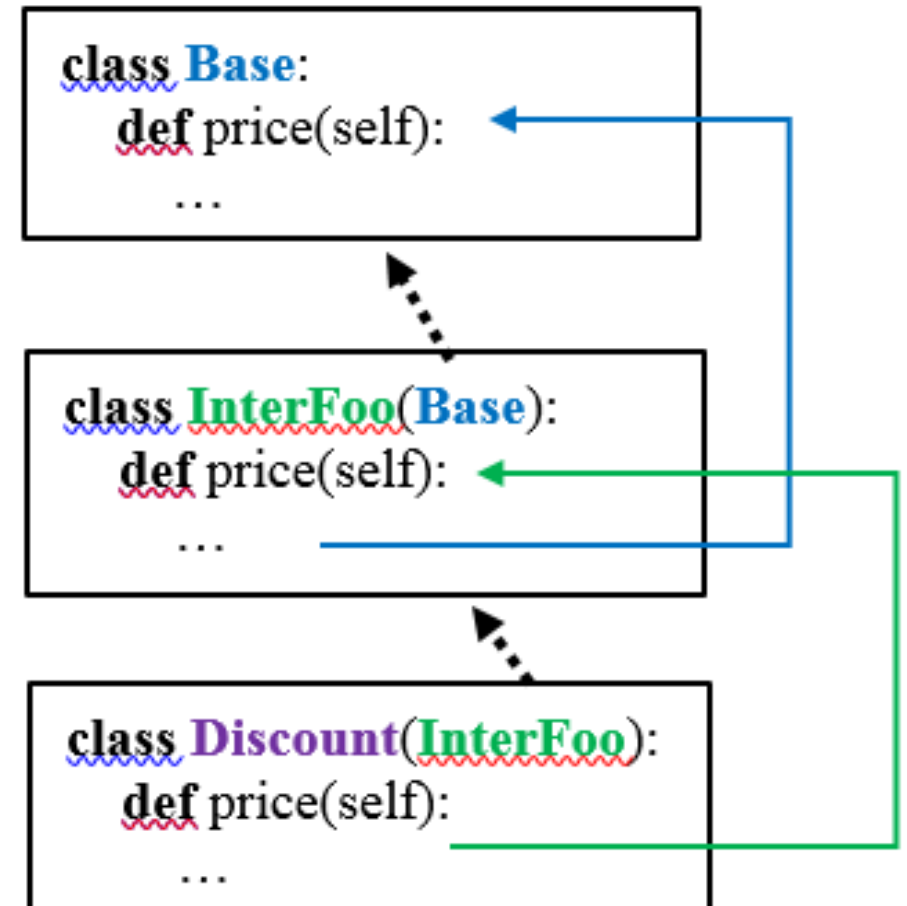
```
class Discount(Base): #создаем класс-потомок Base  
    def price(self): # переопределяем метод price  
        print("in Discount")  
        return Base.price(self)*0.8 # вызываем метод price из Base
```



Наследование

- В приведенном примере, если иерархия классов расширится, и ссылки изменятся, придется изменять код методов в классах.

Например,



Наследование

- Пример

```
class Base:
```

```
    def price(self):  
        print("in Base")  
        return 10
```

```
class InterFoo(Base):
```

```
    def price(self):  
        print("in InterFoo")  
        return Base.price(self)*1.1
```

```
class Discount(InterFoo):
```

```
    def price(self):  
        print("in Discount")  
        return InterFoo.price(self)*0.8
```

Пришлось изменить код метода
price() в классе Discount()

Наследование

- **super()** – встроенная функция языка Python, возвращает прокси-объект, который делегирует вызовы методов классу родителю (или наоборот) текущего класса (или класса на выбор, если он указан как параметр).
- **Применение**: получение доступа из класса наследника к методам класса-родителя в том случае, если наследник переопределил эти методы.

Наследование

- Пример 1

```
class Base:
```

```
    def price(self):  
        print("in Base")  
        return 10
```

```
class Discount(Base):
```

```
    def price(self):  
        print("in Discount")  
        return super().price(self)*0.8
```

- Пример 2

```
class Base:
```

```
    def price(self):  
        print("in Base")  
        return 10
```

```
class InterFoo(Base):
```

```
    def price(self):  
        print("in InterFoo")  
        return super().price(self)*1.1
```

```
class Discount(InterFoo):
```

```
    def price(self):  
        print("in Discount")  
        return super().price(self)*0.8
```

Наследование

- `super()`
- При обращении к атрибутам через метод `super()` при одиночном наследовании порядок обхода родительских классов соответствует обратному порядку их определения (т.е. снизу вверх по иерархии).

Наследование

- **super()**
- Наиболее часто **super()** применяется в методе **__init__()** (как правило, первой строкой), для обеспечения целостности объектов.
- В противном случае, может оказаться, что у дочернего объекта унаследованные атрибуты не будут инициализированы!

Наследование

- `super()`

Пример 1

```
class A:
    def __init__(self):
        self.x = 10
class B(A):
    def __init__(self):
        self.y = self.x + 5

print(B().y) # ошибка!
```

Пример 2

```
class A:
    def __init__(self):
        self.x = 10
class B(A):
    def __init__(self):
        super().__init__() # инициализация
                           # атрибута x
        self.y = self.x + 5

print(B().y) # 15
```

Наследование

- `super([type[, object]])`
- **Параметры:**
 - **type** – тип, к предкам которого нужно обратиться
 - **object** – объект, к которому надо привязаться.
- **Данная форма метода применяется при использовании `super()` вне класса или при явном задании класса-предка, с которого нужно начать поиск методов.**

Наследование

- **super()**

```
class A:
    def __init__(self):
        self.x = 10

class B(A):
    def __init__(self):
        super().__init__() # инициализация атрибута x
        # или
        super(B, self).__init__() # старый формат вызова super()

print(B().y) # 15
```


Наследование

- *Множественное наследование классов*
- **Класс, получаемый при множественном наследовании, объединяет поведение своих надклассов, комбинируя стоящие за ними абстракции.**

```
class Horse:  
    def __init__(self)  
    def run(self)  
    def showName(self)  
    def showInfo(self)
```

```
class Donkey:  
    def __init__(self)  
    def run(self)  
    def showName(self)  
    def showInfo(self)
```

```
class Mule(Horse, Donkey):  
    def __init__(self)  
    def run(self)  
    def showInfo(self)
```



Наследование

- *Множественное наследование классов* применяется:
 - для получения класса с заданными общедоступными методами;
 - для добавления примесей (mixins);
 - когда объекты класса (получающегося в результате множественного наследования), нужно использовать в качестве объектов всех родительских классов.

Наследование

Множественное наследование:

```
class <имя_нового_класса>(<имя_родителя1>,  
    <имя_родителя2>, ..., <имя_родителяN> ):  
    # тело класса
```

Наследование

- **Пример**

```
class A:
    def a(self): return 'a'
class B:
    def b(self): return 'b'
class C:
    def c(self): return 'c'
class AB(A, B):
    pass
class BC(B, C):
    pass
class ABC(A, B, C):
    pass
```

В случае с Python наследование можно считать одним из способов собрать нужные комбинации методов в серии классов:

- Класс АВ – методы a() и b()
- Класс ВС – методы b() и c()
- Класс АВС – методы a(), b() и c()

Наследование

• Пример

```
def ma(self): return 'a'  
def mb(self): return 'b'  
def mc(self): return 'c'  
class AB:  
    a = ma  
    b = mb  
class BC:  
    b = mb  
    c = mc  
class ABC:  
    a = ma  
    b = mb  
    c = mc
```

Собрать нужные методы в классы можно и без множественного наследования:

- **Класс АВ – методы a() и b()**
- **Класс ВС – методы b() и c()**
- **Класс АВС – методы a(), b() и c()**

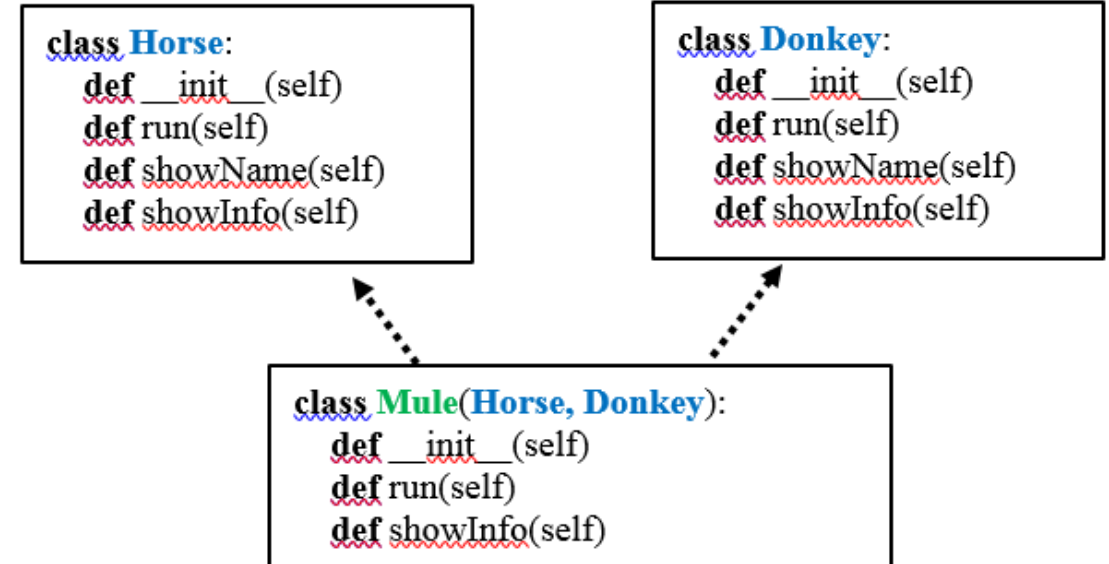
Наследование

```
class Horse:
    maxHeight = 200 # centimeter
    def __init__(self, name, horsehair):
        self.name = name
        self.horsehair = horsehair

    def run(self):
        print ("Horse run")

    def showName(self):
        print ("Name: (House's method): ", self.name)

    def showInfo(self):
        print ("Horse Info")
```



Наследование

class Donkey:

def __init__(self, name, weight):

self.name = name

self.weight = weight

def run(self):

print ("Donkey run")

def showName(self):

print ("Name: (Donkey's method): ", self.name)

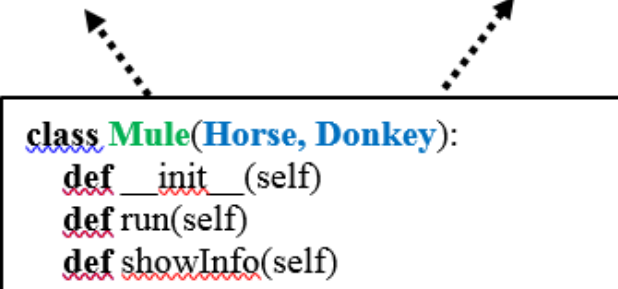
def showInfo(self):

print ("Donkey Info")

```
class Horse:
    def __init__(self)
    def run(self)
    def showName(self)
    def showInfo(self)
```

```
class Donkey:
    def __init__(self)
    def run(self)
    def showName(self)
    def showInfo(self)
```

```
class Mule(Horse, Donkey):
    def __init__(self)
    def run(self)
    def showInfo(self)
```



Наследование

Класс Mule унаследован от Horse и Donkey.

```
class Mule(Horse, Donkey):
```

```
    def __init__(self, name, hair, weight):
```

```
        Horse.__init__(self, name, hair) # вызываем __init__ класса Horse
```

```
        Donkey.__init__(self, name, weight)
```

```
    def run(self):
```

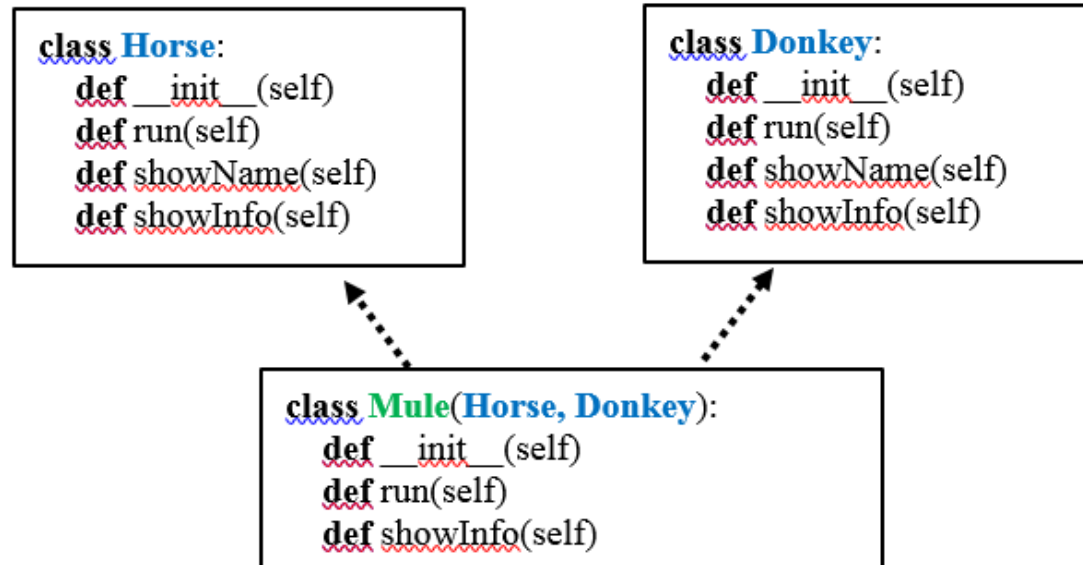
```
        print ("Mule run")
```

```
    def showInfo(self):
```

```
        print ("-- Call Mule.showInfo: --")
```

```
        Horse.showInfo(self)
```

```
        Donkey.showInfo(self)
```



Наследование

- *Множественное наследование классов.*
- *Замечание:*
 - Родительские классы могут иметь одинаковые атрибуты (attribute) или методы
 - Подкласс будет приоритетно наследовать атрибуты, методы, ... первого класса в списке наследования.

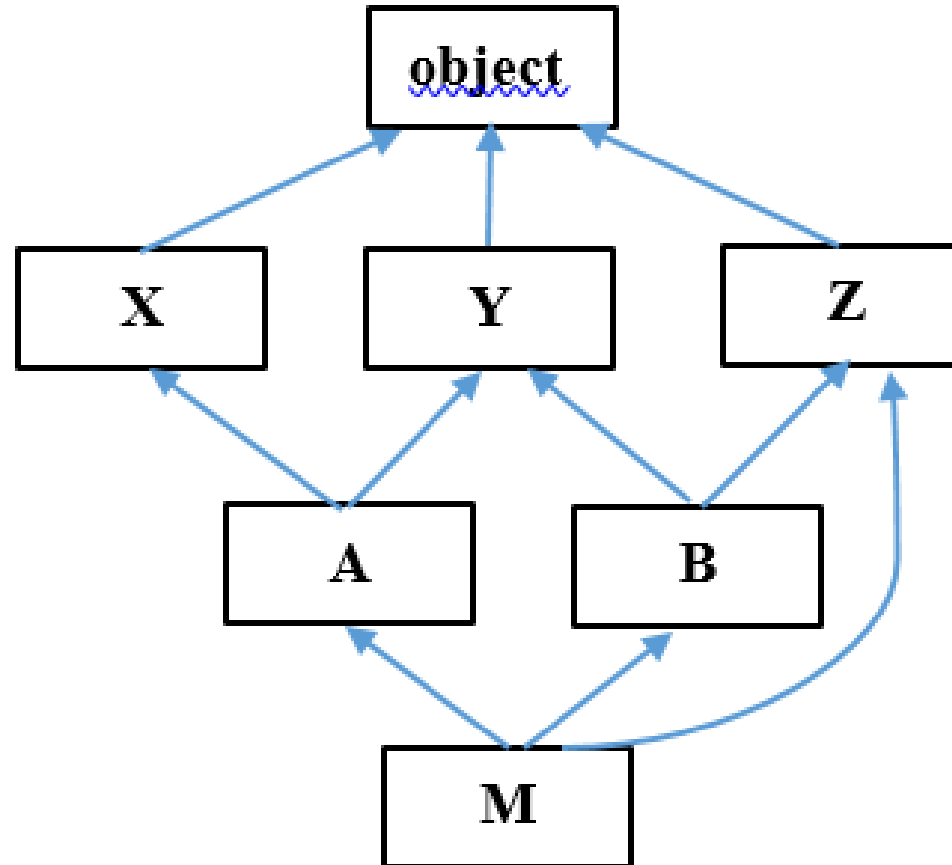
Наследование

- *Множественное наследование классов.*
- *Метод **super()**:*
 - При множественном наследовании **super()** необязательно указывает на родителя текущего класса, а может указывать и на собрата.
 - Все зависит от структуры наследования и начальной точки вызова метода!

Наследование

- *Множественное наследование классов.*

```
class X:  
    pass  
class Y:  
    pass  
class Z:  
    pass  
class A(X, Y):  
    pass  
class B(Y, Z):  
    pass  
class M(B, A, Z):  
    pass
```



В каком
порядке будут
просмотрены
классы
иерархии?

Порядок обхода
определяется
списком **MRO**.

Наследование

- *Множественное наследование классов.*
- ***MRO*** (*method resolution order*) – порядок разрешения методов (любых атрибутов класса)
- *Метод **mro()*** – возвращает список классов в том порядке, в котором Python будет искать методы в иерархии классов пока не найдет нужный или не выдаст ошибку.
- *Метод **mro()*** возвращает список MRO по алгоритму C3-линеаризации.

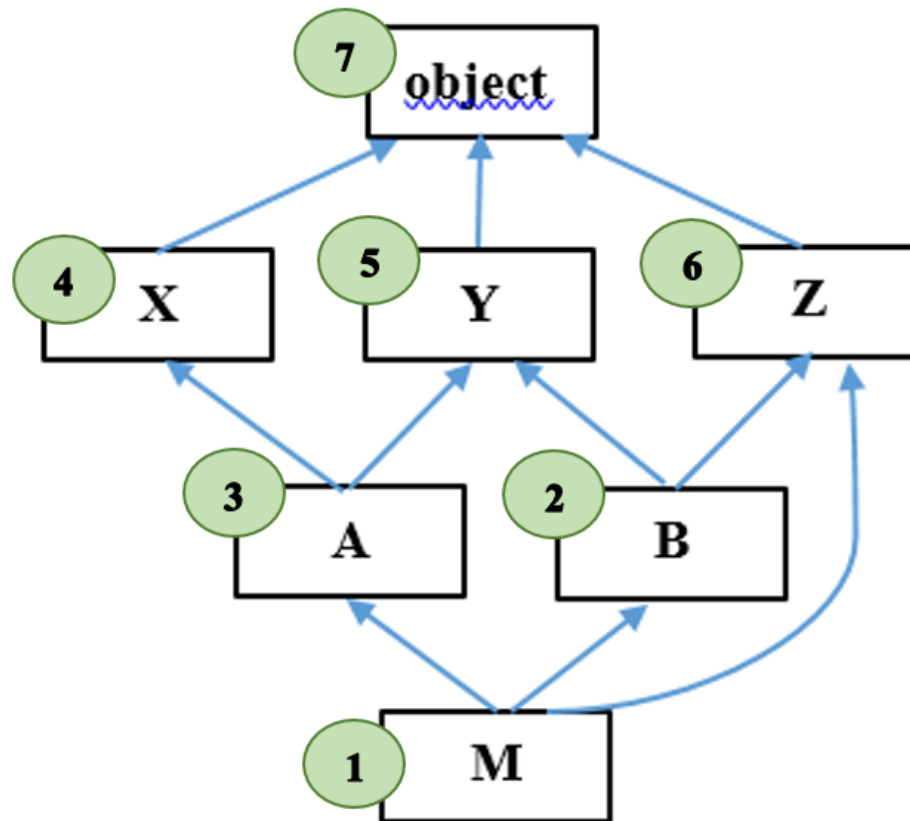
Наследование

- *Множественное наследование классов.*
- Алгоритм С3-линеаризации.
 1. Каждый класс должен входить в список ровно 1 раз.
 2. Порядок обхода классов должен соответствовать порядку наследования.
 - Например: для класса `class D(A, B, C):` порядок появления в списке MRO: `D -> ... -> A -> ... -> B -> ... -> C -> ...`
 - Между ними могут оказаться и другие классы, но исходный порядок должен быть соблюден.
 3. Родители данного класса должны появляться по порядку старшинства.
 - Сначала идут непосредственные родители, потом дедушки и бабушки, но не наоборот.

Наследование

- Множественное наследование классов.*

```
class X: pass
class Y: pass
class Z: pass
class A(X, Y):
    pass
class B(Y, Z):
    pass
class M(B, A, Z):
    pass
print(M.mro())
```



```
Run: oop_test x
<class '__main__.M'>
<class '__main__.B'>
<class '__main__.A'>
<class '__main__.X'>
<class '__main__.Y'>
<class '__main__.Z'>
<class 'object'>
```

Наследование

- *Множественное наследование классов.*
- Таким образом, вызов `super()` автоматически определяет к какому классу (родителю или брату) обращаться в соответствии со списком MRO.
- Все зависит от иерархии класса и начальной точки вызова.

Наследование

- *Множественное наследование классов.*
- Бывают ситуации, когда список **MRO** составить невозможно.
- В этом случае, буде выдано сообщение об ошибке!

Пример 1

```
class X: ...  
class Y(X): ...  
class A(X, Y): ...
```

Здесь класс X наследуется дважды, поэтому будет нарушено либо правило порядка наследования, или правило старшинства.

Наследование

- *Абстрактные классы и методы*
- Абстрактный класс – класс, содержащий хотя бы один абстрактный метод
- Абстрактный метод – метод, который объявлен, но не реализован.
- Нельзя создать экземпляры абстрактного класса.
- Для объявления абстрактного метода используют встроенный декоратор Python **@abstractmethod**.

Наследование

- *Абстрактные классы и методы*
- В Python нет синтаксической поддержки абстрактных классов, но есть встроенный модуль abc (abstract base classes), который помогает проектировать абстрактные сущности.
- Абстрактный класс наследуют от класс abc.ABC (Python 3.4+)

Наследование

- *Абстрактные классы и методы*
- В Python нет синтаксической поддержки абстрактных классов, но есть встроенный модуль abc (abstract base classes), который помогает проектировать абстрактные сущности.
- Абстрактный класс наследуют от класс abc.ABC (Python 3.4+)

Наследование

- *Абстрактные классы и методы*

```
from abc import ABC
class Hero(ABC):
    pass

hero = Hero()
print(f"hero: {hero}")
```

- Следует заметить, пока в класс не добавлены абстрактные методы, его экземпляры можно создавать!

Наследование

- *Абстрактные классы и методы*
- Для объявления абстрактного метода используют встроенный декоратор Python из модуля abc `@abc.abstractmethod`.

```
class Hero(abc.ABC):  
    @abc.abstractmethod  
    def attack(self):  
        pass
```

```
hero = Hero()  
print(f"hero: {hero}")
```



TypeError: Can't instantiate abstract class Hero with abstract methods attack

Наследование

- *Абстрактные классы и методы*

```
class Hero(abc.ABC):  
    @abc.abstractmethod  
    def attack(self):  
        pass  
  
class Archer(Hero):  
    def attack(self):  
        print('выстрел из лука')  
  
Archer().attack()
```

- **Экземпляры
потомков можно
создавать и
использовать
реализацию
метода attack**

Наследование

- *Абстрактные классы и методы*
- Кроме обычных методов, абстрактными можно обозначить и статические, классовые методы, а также свойства

```
class Hero(abc.ABC):  
    @staticmethod  
    @abc.abstractmethod  
    def attack(self): pass  
  
    @property  
    @abc.abstractmethod  
    def name(self):  
        """Hero name property"""  
        pass
```