04. ООР (2)

27 октомври 2022

Какво беше ООП?

- Абстракция / Abstraction
- Енкапсулация / Encapsulation
- Модулярност / Modularity

Как се пишеха класове?

Какво не е наред?

```
class Snickers:

    def __init__(size, code):
        self.size = size
        self.code = code
```

А-а-а, да!

```
class Snickers:

    def __init__(size, code):
        self.size = size
        self.code = code


snickers = Snickers(45, 'definitely_not_a_real_code') # -> ?
```

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    snickers = Snickers(45, 'definitely_not_a_real_code')
TypeError: Snickers.__init__() takes 2 positional arguments but 3 were given
```

А какво е self?

По конвенция така се кръщава първият аргумент на методите на клас.

В такъв случай...

Трябва ли всеки метод на един клас да приема `self` като първи аргумент?

- Когато методът е декориран със `@staticmethod`
- Когато методът е декориран с `@classmethod`, първият метод ще е самият клас.
  - Конвенцията при такъв случай е да кръщаваме аргумента `cls`
- Защо не може да кръстим аргумента **`class`**?
- ...**`class`** е запазена дума

Del

С ключовата дума del можем да изтриваме атрибути на обект.

```
class Statue:
    def __init__(self):
        self.left_hand = 'Generic left hand'
        self.right_hand = 'Generic right hand'


venus_de_milo = Statue()
print(f"I have {venus_de_milo.left_hand} and {venus_de_milo.right_hand}")
del venus_de_milo.left_hand
print(venus_de_milo.left_hand) # ->?

# AttributeError: 'Statue' object has no attribute 'left_hand'
```

Добре. Да надградим, но с пример.

```
class Hand:

    def __init__(self):
        self.thumb = 'Палец'
        self.index_finger = 'Показалец'
        self.middle_finger = 'Среден'
        self.ring_finger = 'Безименен'
        self.pinkie = 'Кутре'


hand = Hand()
print(hand.middle_finger) # Среден
```

Не може ли малко по-удобно?

Реално пръстите имат конкретна позиция, така че би било полезно да можем да поискаме пръст по конкретен индекс.

```
hand[0] == 'Палец'
hand[1] == 'Показалец'
hand[2] == 'Среден'
hand[3] == 'Безименен'
hand[4] == 'Кутре'
```

__getitem__

За да направим така, че класът ни да поддържа достъп от вида `object[index]`, просто трябва да дефинираме `__getitem__`.

```
class Hand:

    def __init__(self):
        ...

    def __getitem__(self, index):
        return (self.thumb, self.index_finger, self.middle_finger,
                self.ring_finger, self.pinkie)[index]


hand = Hand()
print(hand[4]) # Кутре
```

Ами присвояване?

Какво ще се случи тук?

```
hand = Hand()
hand[2] = 'кебапче'
```

```
Traceback (most recent call last):
  File "test.py", line 15, in <module>
    hand[2] = 'кебапче'
TypeError: 'Hand' object does not support item assignment
```

__setitem__

Щом има get, ще има и set.

```
class Hand:
    ...

    def __setitem__(self, index, value):
        if index == 0:
            self.thumb = value
        elif index == 1:
            self.index_finger = value
        # ...

hand = Hand()
hand[1] = 'кебапче'
print(hand.index_finger) # 'кебапче'
```

Чудесно! Ръката вече работи с индекси.

Но има още похвати, с които да я почерпим.

__getattr__

```
class Hand:
    ...

    def __getattr__(self, name):
        return f"Това е ръка v1.0. Все още няма {name} :("


hand = Hand()
print(hand.middle_finger) # Среден
print(hand.sixth_finger) # Това е ръка v1.0. Все още няма sixth_finger :(
```

__getattr__

`__getattr__` е fallback механизъм, който Python ще потърси само в случай, че няма дефиниран атрибут с търсеното име.

__setattr__

Да не се повтаряме, но - “Щом има get, ще има и set“.

```
class Hand:
    ...

    def __setattr__(self, name, value):
        print(f"Нова стойност за {name} - {value}")
        object.__setattr__(self, name, value)

hand = Hand()
hand.pinkie = 'Малък пръст'
```

```
Нова стойност за thumb - Палец
Нова стойност за index_finger - Показалец
Нова стойност за middle_finger - Среден
Нова стойност за ring_finger - Безименен
Нова стойност за pinkie - Кутре
Нова стойност за pinkie - Малък пръст
```

Но __setattr__ не е точно като __getattr__

- `__setattr__` се извиква **винаги** при присвояване на нова стойност
- Това значи, че ако в тялото му просто има `self.name = value`, ще се получи безкрайна рекурсия
- За да се справим с този проблем, използваме `object.__setattr__`, или с други думи, методът за сетване на атрибути от базовия Python клас.

__getattribute__

- Подобно на `__setattr__` , има метод, който се извиква при всяко записване на атрибут - `__getattribute__`
- Рядко е нужно, но е добре да се знае, че съществува

```
class Hand:
    ...

    def __getattribute__(self, name):
        print(f"Някой ми бърка по пръстите и иска {name}")
        return object.__getattribute__(self, name)


hand = Hand()
print(hand.pinkie)

# Някой ми бърка по пръстите и иска pinkie
# Кутре
```

Лирическо отклонение за подсилване на поантата 1/2

Можем да разглеждаме всеки обект като съвкупност от две неща:

- речник, съдържащ атрибутите на обекта (атрибута `__dict__` на обекта)
- връзка към класа на обекта (атрибута `__class__` на обекта)

```
class Hand:
    ...

hand = Hand()
print(hand.__dict__)
print(hand.__class__)

# {'thumb': 'Палец', 'index_finger': 'Показалец', ...}
# <class '__main__.Hand'>
```

Лирическо отклонение за подсилване на поантата 2/2

Функциите и променливите, дефинирани в тялото на класа, са атрибути на класа.

```
class Hand:

    fingers_count = 5

    def capitalize_finger(self, name):
        setattr(self, name, getattr(self, name).upper())

print(Hand.fingers_count) # 5
print(Hand.capitalize_finger) # <function Hand.capitalize_finger at
0x0000013E2413FA30>
print(Hand.__dict__) # {'fingers_count': 5, 'capitalize_finger': <function
Hand.capitalize_finger at 0x00000250DD30FA30>, ...}
```

Поанта!

Когато Python срещне кода `some.thing`, прави следните неща:

- Връща стойността на `some.__dict__['thing']`
- Ако не намери, търси `some.__class__.thing`
  - ако не е функция, се връща директно
  - ако е функция, се връща bound method
- Ако го няма и там, се вика `some.__getattr__('thing')`
- В краен случай - `AttributeError: 'Some' object has no attribute 'thing'`

Наследяване

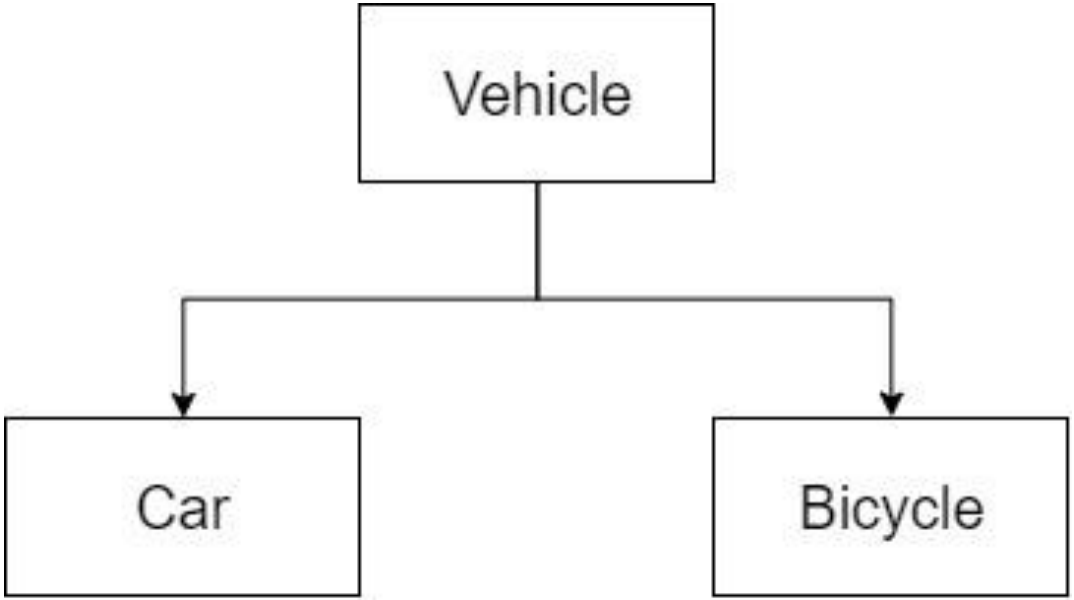
- Да, в Python има наследяване
- Както вече видяхме, всички класове в Python наследяват `object`
  - * В Python 2 това не е така - просто го имайте предвид
- Можете да напишете клас, който наследява друг клас, както и такъв, който наследява няколко

Защо (не) искаме да наследяваме?

- Клас X, който наследява клас Y, “взима” всички атрибути и методи на своя родител (класа Y)
  - Това позволява преизползване на код
  - Както и по-добра модуларност
- Можете да постигнете логическа свързаност между класовете си
  - В случай, че самите класове **наистина** имат връзка в смисъла на “клас X е вид клас Y“, т. е. е негов подклас
- Но…
- Наследяването води до прекалено много code coupling, което може да доведе до пълна каша
- Лесно е да приложите наследяване в случай, който по-скоро изисква друг подход - трябва да се внимава
- Прекомерната употреба води до прекалено сложен дизайн

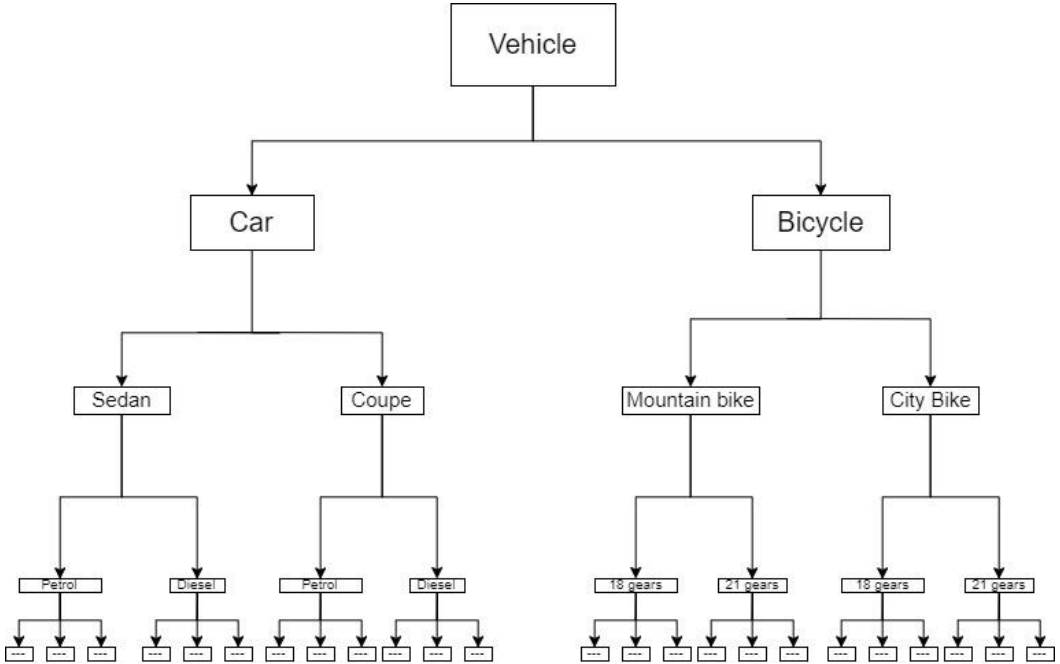
Сложен дизайн?

- Често срещана грешка е да приложите наследяване, за да получите йерархия от този тип, което за начало звучи логично…



Сложен дизайн?

- …но ако в продължите в същия дух, ще стигнете до нещо такова, което трудно се менажира
- всяка промяна ще пропагира в останалите слоеве и ще получите доста чуплив дизайн

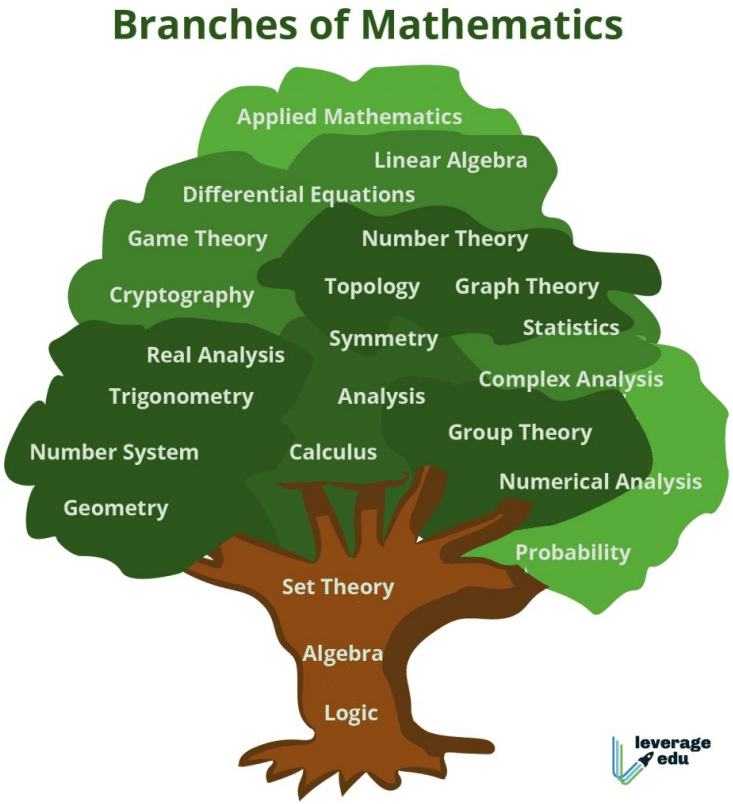


Да се поучим от другите науки

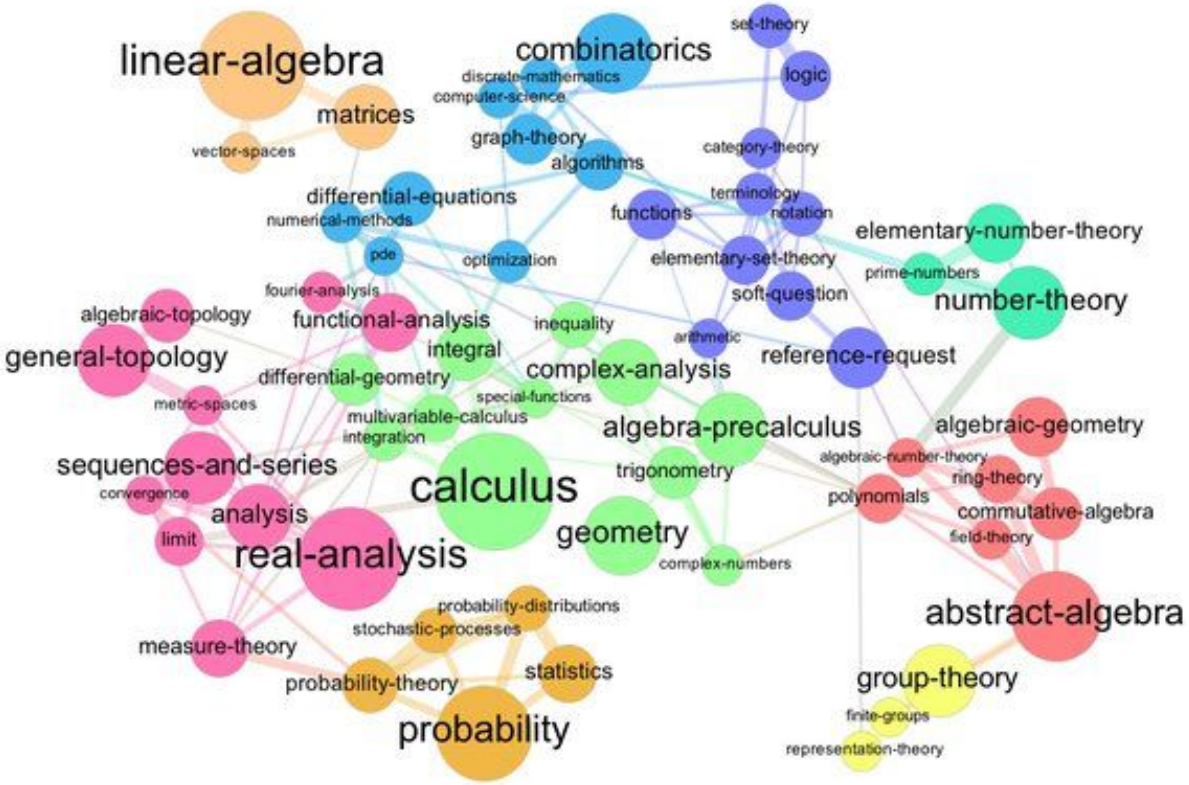
- И във физиката хората са се опитали да разглеждат света като комбинации от малки елементи (атоми), които го изграждат - очевидно не е точно така

- И в математиката сме се опитали да класифицираме различни елемнти в логически обособени части, за да е по-лесно

    - Звучи и изглежда обещаващо, но…



Реалната картинка е по-скоро това



Добре де, ще внимавам, но как да наследя в Python?

```
class Limb:

    def __init__(self):
        ...


class Hand(Limb):

    def __init__(self):
        ...


class Leg(Limb):

    def __init__(self):
        ...
```

Можем да използваме атрибути и методи на родителя

```
class Limb:

    name = 'Limb'

    def introduce(self):
        return f"I am a {self.name}"


class Hand(Limb):
    pass

hand = Hand()
print(hand.introduce()) # I am a Limb
```

Можем да ”презаписваме” методи на родителя

```
class Limb:

    name = 'Limb'

    def introduce(self):
        return f"I am a {self.name}"


class Hand(Limb):

    name = 'Hand'

    def introduce(self):
        return f"I am a {self.name}"

hand = Hand()
print(hand.introduce()) # I am a Hand
```

Детето има достъп и търси за атрибути и в родителя

```
class Limb:

    name = 'Limb'

    def introduce(self):
        return f"I am a {self.name}"


class Hand(Limb):

    def introduce(self):
        return f"I am a {self.name}"

hand = Hand()
print(hand.introduce()) # I am a Limb
```

В случай, че детето няма конструктор, използва се родителския

```
class Limb:

    name = 'Limb'

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

    def introduce(self):
        return f"I am a {self.name}"


class Hand(Limb):

    def introduce(self):
        return f"I am a {self.name}"

hand = Hand('Герасим')
print(hand.introduce()) # I am a Герасим
```

В случай, че има, използва се неговият

```
class Limb:

    name = 'Limb'

    def __init__(self, name):
        print('Executing parent constructor.')
        self.name = name


class Hand(Limb):

    def __init__(self, name):
        print('Executing child constructor.')
        self.name = name

hand = Hand('Герасим')
# Executing child constructor.
```

Можете да използвате и родителския конструктор

```
class Limb:

    name = 'Limb'

    def __init__(self, name):
        print('Executing parent constructor.')
        self.name = name


class Hand(Limb):

    def __init__(self, name):
        super().__init__(name)
        print('Executing child constructor.')
        self.name = name

hand = Hand('Герасим')
# Executing parent constructor.
# Executing child constructor.
```

Един практически пример

```
class Limb:

    def move(self):
        return f"{self.action} your {self.name}s"

class Hand(Limb):

    name = 'hand'
    action = 'clap'

    def sing(self):
        return f"If you're happy and you know it {super().move()}"

class Leg(Limb):

    name = 'feet'
    action = 'stomp'

    def sing(self):
        return ("If you're happy and you know it and "
                f"you really want to show it {super().move()}")
hand = Hand()
leg = Leg()
print(hand.sing()) # If you're happy and you know it clap your hands
print(leg.sing()) # If you're happy and you know it and you really want to show it stomp your feets
```

Къде е проблемът тук?

```
class Finger:

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

class Hand(Finger):

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

Наследяване vs Композиция

- Наследяване използваме, когато връзката между класовете е IS (X е просто вид Y)
  - ръката Е крайник
- Композиция използваме, когато връзката между класовете е HAS (X има компоненти от тип Y)
  - ръката ИМА пръсти

```
class Limb:

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

class Hand(Limb):

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

```
class Finger:

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

class Hand:

    def __init__(self):
        self.thumb = Finger('thumb')
```

Композиция > наследяване

- Композицията не води до толкова силен code coupling, но…
  - …ако се използва разумно

- Ръката не трябва да казва на всяка фаланга от пръста да мърда по определен начин. Тя трябва да каже на пръста - “свий се“ и той да знае какво да направи.
- Ръката не трябва да проверява отделно дали даден пръсти има нужда от маникюр, лепенка при порязване, или друг вид поддръжка - тя трябва просто да пита дали има проблеми за отстраняване.
- Пръстът не трябва да се интересува от факта, че е част от ръка. Той е просто пръст и не знае какво се случва в останалия свят.

Mixins

- Миксините са добра причина (една от малкото) за използване на множествено наследяване
- Миксин класовете не се използват сами по себе си. Те са написани, за да се наследяват.
- Можете да гледате на Mixin като интерфейс с имплементирани методи
- Има два главни случая, в които е добра идея да използвате Миксини
  - Когато искате да "забъркате" множество атрибути и методи в един клас
  - Когато искате клас, който предлага само едно поведение, което поведение искате да ползвате само като част от много други класове

Нагледно

- Дефинирате класове, които просто изпълняват функционалността

```
class NosePicker:
    def pick_nose(self):
        print('Извършвам действия, нужни за почистване на носа.')

class DriverAssistant:
    def assist_while_driving(self):
        print('Конфигурирам се за сигнализиране на останалите шофьори.')

class Scrather:
    def scratch(self):
        print('Почесвам всичко, което кажеш.')

class RingHolder:
    def wear_a_ring(self):
        print('Слагам си пръстен.')

class ForePlayer:
    def help_please_a_woman(self):
        print('Привеждам партньора си в готовност за забавления.')
```

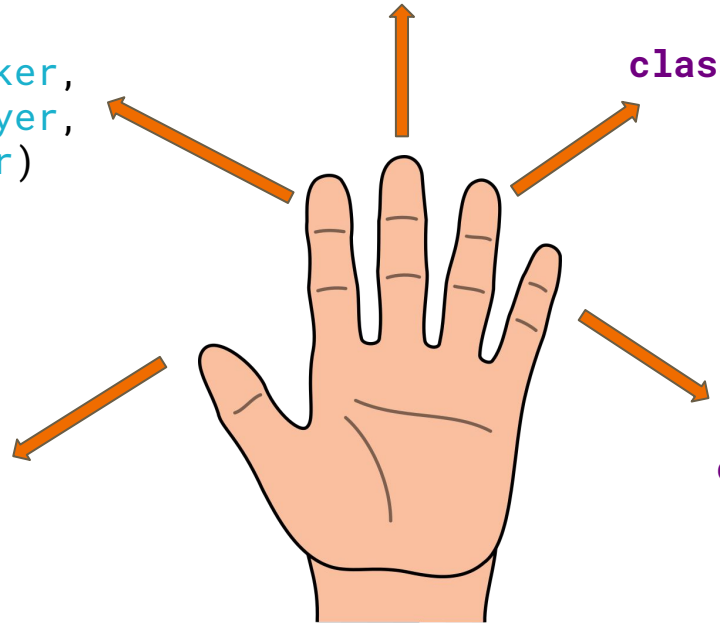
Нагледно

- Наследявате вече подготвените Mixins, където е нужно

```
class MiddleFinger(DriverAssistant,
                   ForePlayer,
                   Scrather)
```

```
class IndexFinger(NosePicker,
                  ForePlayer,
                  Scrather)
```

```
class RingFinger(ForePlayer,
                 Scrather,
                 RingHolder)
```



```
class Thumb(RingHolder)
```

```
class Pinkie(NosePicker)
```

И последно. Да напомним. И допълним.

- В Python енкапсулацията е въпрос на добро възпитание
- Имена от типа `_name` са *protected* (не го пипай - не ти трябва)
- Имена от типа `__name` са *private* (не, не, наистина не го пипай - ще счупиш нещо)
- Интерпретатора променя имената от тип `__name` до `_classname__name`. Нарича се *name mangling* и създава ефект, подобен на този в C++/Java.

Бонус. Последно, обещавам!

```
class Limb:
    pass


class Hand(Limb):
    pass


hand = Hand()
print(type(hand) is Hand) # True
print(type(hand) is Limb) # False
print(type(hand)) # <class '__main__.Hand'>
print(isinstance(hand, Hand)) # True
print(isinstance(hand, Limb)) # True
print(issubclass(Hand, Limb)) # True
print(issubclass(Limb, Hand)) # False
```

Въпроси?