

---

---

# 07.00P (2)

— 29 октомври 2024 —

---

---

---

---

**a.k.a POOP**

— Python Object Oriented  
Programming —

---

---

# Преди това

Какво е това?

```
function_that_says_ni = lambda *args, **kwargs: "Ni!"
```

Лесни 5 точки от първото домашно. Just saying....

Със следващото няма да е толкова лесно. Пече се! Ще го получите утре.

# Как се пишеха класове?

Какво ще се случи?

```
class Snickers:
```

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

```
print(Snickers(45, 'definitely_not_a_real_code').size)
```

# A-a-a, 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

# Любимият ви спам

```
class Spam:
```

```
    def __init__(*spams):  
        setattr(*spams)
```

```
spam = Spam('spam', 'spam')  
print(spam.spam) # ?
```

```
print(spam.spam) # 'spam'
```



# Любимият ви спам.спам

```
class Spam:

    def __init__(self, spam):
        self.spam = Spam(spam-1) if spam else spam

spam = Spam(100)
print(type(spam.spam.spam.spam.spam.spam.spam.spam.spam.spam.spam)) # ?

# AttributeError: 'Spam' object has no attribute 'spma'
```



# Видове методи 1

Каква е разликата между тези методи?

```
class Spam:
```

```
    def method(self):           Public  
        pass
```

```
    def _method(self):         Protected  
        pass
```

```
    def __method(self):        Private (name mangling)  
        pass
```

# Видове методи 2

```
class Spam:  
  
    def method(self):  
        pass
```

```
spam = Spam()
```

Каква е разликата между тези обекти?

`object1 = Spam.method`      Unbound метод (очаква един аргумент)

`object2 = spam.method`      Bound метод (не очаква аргументи)

# Мистерия



# Зад това дърво има скрита имплементация на името Spam

```
type(Spam) # <class 'type'>
```

```
Spam()() # It works!
```

Какъв извод можем да направим за имплементацията на Spam?

```
class Spam:  
    def __call__(self): # Инстанциите могат да се викат като функция  
        print('It works!')
```

# Излел е *del*-ъо хайдутин

```
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) # ?
```

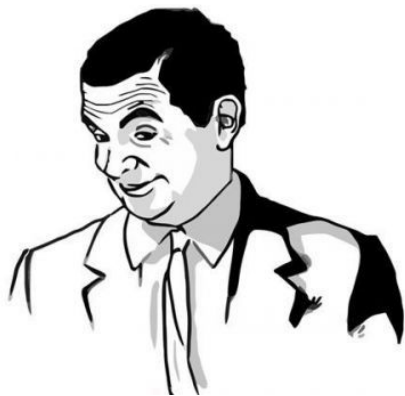
*# AttributeError: 'Statue' object has no attribute 'left\_hand'*

С ключовата дума `del` можем да изтриваме атрибути на обект.



# За какво ще си говорим днес?

- Наследяване
- Композиция
- Дъндъри
- MRO
- Fingering



# Disclaimer

- Между класовете се слагат два празни реда
- Между методите в класа се слага един празен ред
- В слайдовете може и да не е така, защото няма място



# Добре. Да надградим познания, 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]) # Kympe
```

# 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] = 'кебанче'
print(hand.index_finger) # 'кебанче'
```

**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.

# \_\_getattr\_\_

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

```
class Hand:
    ...

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

```
hand = Hand()
print(hand.pinkie)
```

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

# 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  
0x00000013E2413FA30>
```

```
print(Hand.__dict__) # {'fingers_count': 5, 'capitalize_finger': <function  
Hand.capitalize_finger at 0x000000250DD30FA30>, ...}
```

# 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'`

\*\* Картинката не включва дескриптори, но засега - толкова.

# 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

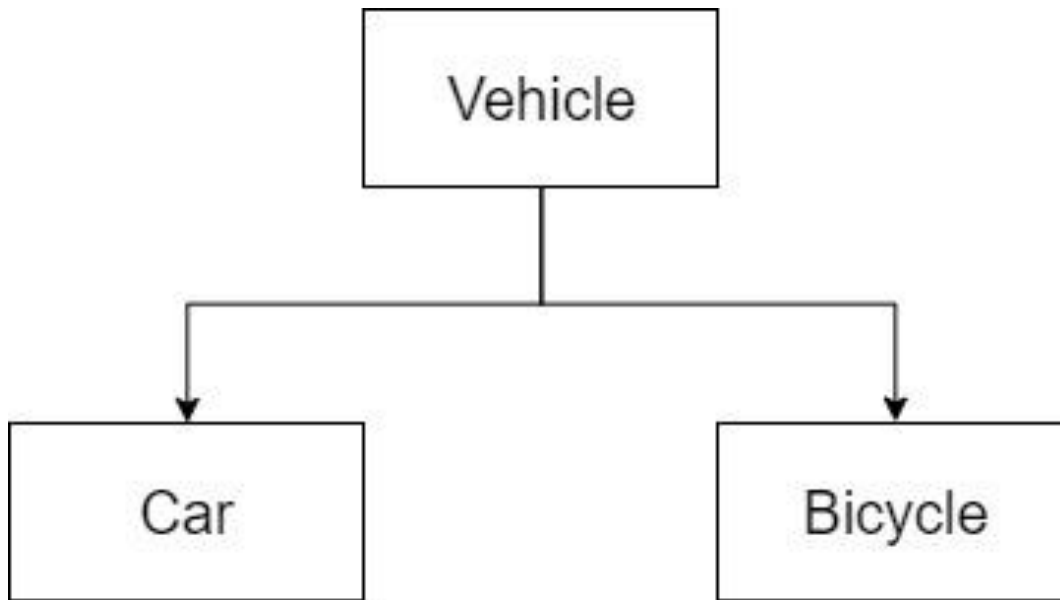


# Защо (не) искаме да наследяваме?

- 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

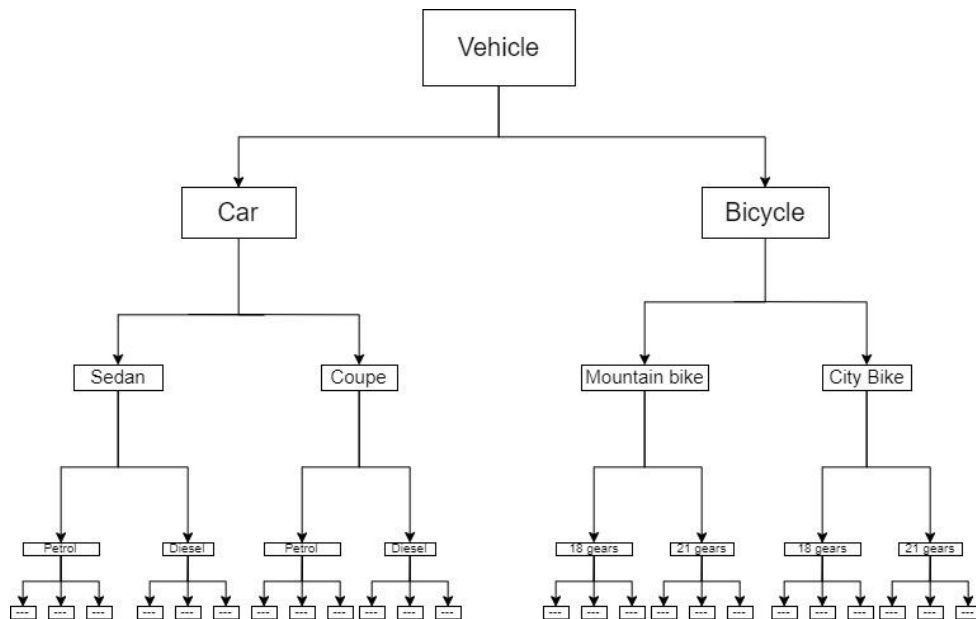
# Сложен дизайн?

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



# Сложен дизайн?

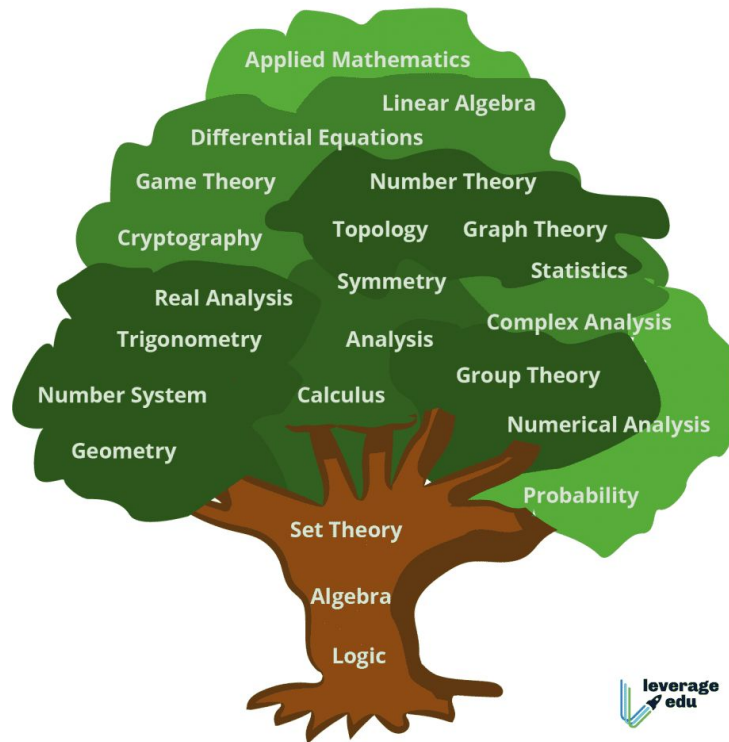
- ...но ако в продължение в същия дух, ще стигнете до нещо такова, което трудно се менажира
- всяка промяна ще пропагира в останалите слоеве и ще получите доста чуплив дизайн



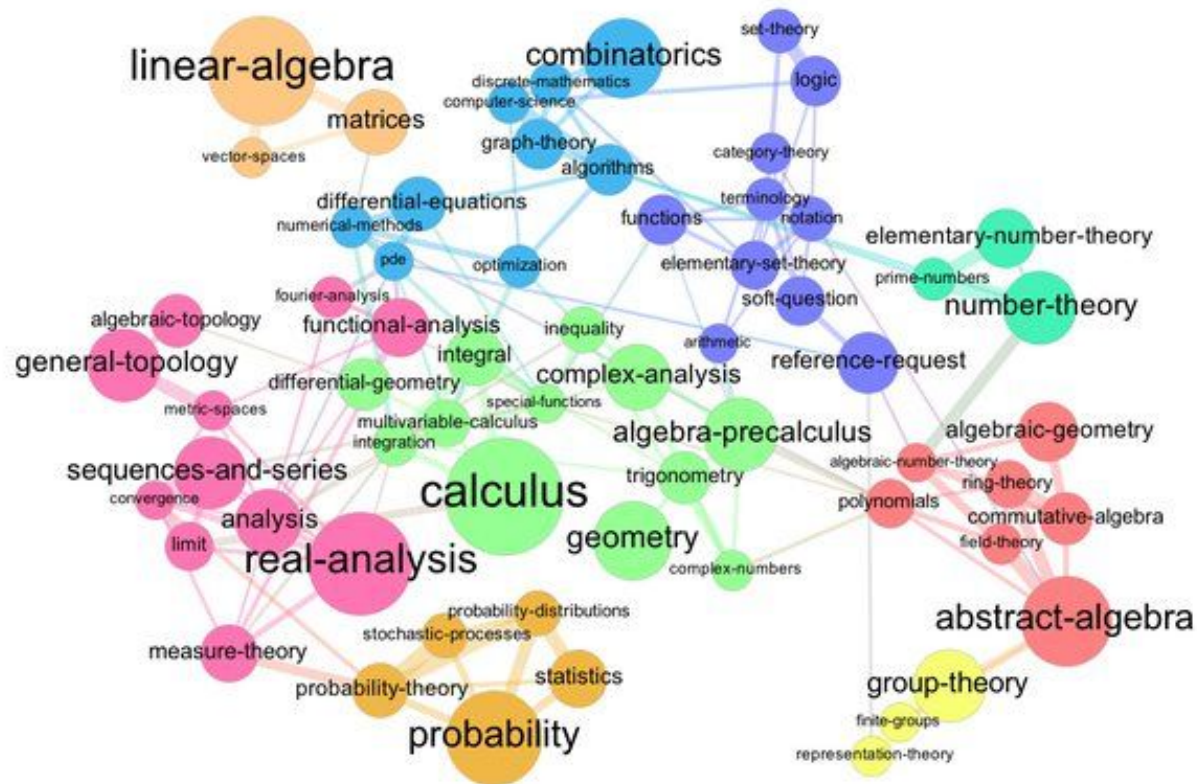
# Да се поучим от другите науки

- 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 елементи в логически обособени части, за да е по-лесно
  - Звучи и изглежда обещаващо, но...

## Branches of Mathematics



# 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 THE {self.name}"

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

# Ако детето няма *конструктор*, използва се родителския

```
class Limb:

    name = 'Limb'

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

# Наследяване от няколко класа

```
class Инженер:  
    def introduce(self):  
        return "инж."
```

```
class Академик:  
    def introduce(self):  
        return "акад."
```

```
class Професор:  
    def introduce(self):  
        return "проф."
```

```
class Сульо(Професор, Академик, Инженер):  
    pass
```

```
сульо = Сульо()  
сульо.introduce() # ?
```



Days since somebody at FMI asked for the diamond problem.

O

O

# За любознателните - MRO

Method Resolution Order

`class.__mro__`



# 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 пръст 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.

# Пример (част 1)

- Имате следния клас за пращане на съобщения

```
class Messenger:  
  
    def __init__(self):  
        self._socket = 'Some magic'  
  
    def send_message(self, message):  
        self._socket.write(message)
```

## Пример (част 2)

- Искате да добавите spell check. Бързо си правите един клас за това.

```
class SpellChecker:  
  
    def __init__(self):  
        self._checker = 'Some magic'  
  
    def check(self, text):  
        return self._checker(text)
```

## Пример (част 3)

- Наследявате и получавате резултат, готов за използване.

```
class SpellCheckedMessenger(Messenger, SpellChecker):  
  
    def send_message(self, message):  
        super().send_message(self.check(message))
```

## Пример (част 4)

- Супер. Но след 2 месеца искате да направите curse check. Лесно си правите клас.

```
class CurseChecker:  
  
    def __init__(self):  
        self._checker = 'Some magic'  
  
    def check(self, text):  
        return self._checker(text)
```

## Пример (част 5)

- Лесно можете да направите същото наследяване.

```
class CurseCheckedMessenger(Messenger, CurseChecker):  
  
    def send_message(self, message):  
        super().send_message(self.check(message))
```



## Пример (част 6)

- А как можете да направите едновременно curse check и spell check?
- А какво правим, ако ни трябват още 20 чекъра?

```
class CurseAndSpellCheckedMessenger(Messenger,  
                                     CurseChecker, SpellChecker):  
    # ???
```

## Пример (част последна)

- А можеше просто да използвате композиция.
- Класовете `SpellChecker` и `CurseChecker` са същите и техни инстанции се подават при инициализация чрез `checkers` параметъра.

```
class Messenger:
```

```
    def __init__(self, checkers):  
        self._socket = 'Some magic'  
        self._checkers = checkers  
  
    def send_message(self, message):  
        for checker in self._checkers:  
            message = checker.check(message)  
        self._socket.write(message)
```

# 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 от много други класове

# Нагледно

- Дефинираните 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 Scratcher:
    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

- Наследяват вече подготвените Mixins, където е нужно

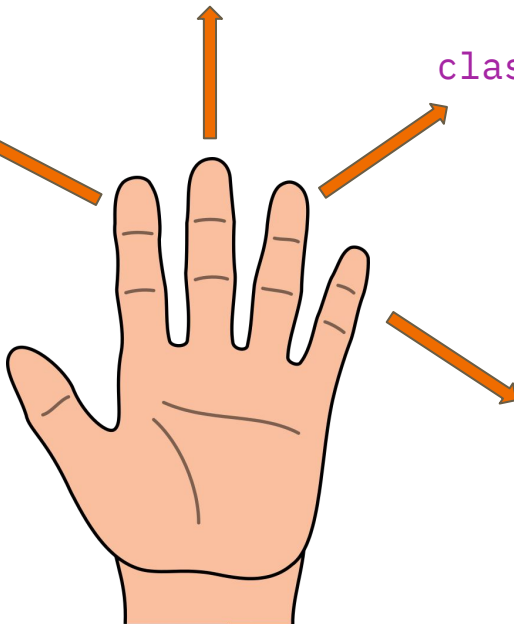
```
class MiddleFinger(DriverAssistant,  
                  ForePlayer,  
                  Scrather)
```

```
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; прави възможно дефиниране на еднакви имена в два наследяващи се класа)
- Интерпретатора променя имената от тип `__name` до `_classname__name`. Нарича се *name mangling*.

# 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
```

## Излъгах. Още едно. *type* с три аргумента.

```
class Hand(Limb):  
    name = 'hand'  
    def say_hi(self):  
        print(f"Hi. I am a {self.name}.")
```

# Е същото като

```
Hand = type('Hand',  
            (Limb, ),  
            {  
                'name': 'hand',  
                'say_hi': lambda self: print(f"Hi. I am a {self.name}.")  
            }  
            )
```



**Въпроси?**