Дефиниция на ООП. Основни понятия в ООП. Класове и обекти. Атрибути и методи. Реализация в Python. Специални атрибути и методи.

Основни теми

- Основни понятия
- Обекти и класове
- Атрибути и методи
- Създаване и използване на класове и обекти
- Наследяване
- Модул dataclass
- Дефиниране на типове данни в Python
- Класовете като обекти

Въведение

Програмирането включва използване и съхраняване на *данни* и *обработване* на тези данни

Например, за да намерим най-малкия елемент в списък:

Имаме нужда от данните – списък с числа

И алгоритъм който намира най-малкото от тях

Методът за програмиране базиран на действия (*изпълнение на команди и функции*) за обработка и получаване на *данни* е наричан *императивно* или *процедурно* програмиране

Обектно-ориентираното програмиране обединява данните и командите за тяхната обработка в *класове* и *обекти* (представители на класовете)

Процедурно програмиране

Алгоритмите се реализират чрез програми, обработващи различни структури от данни

Програмите се организират по формата на:

- Процедури (функции, модули, блокове) включващи команди
- Структури от данни – обработвани от командите (процедурите)

В езика Python процедурите се реализират с помощта на модули, функции, команди и изрази

В езика Python структурите от данни се реализират с помощта на типове данни и обекти (елементи, константи) принадлежащи към типовете данни

Обектно-ориентирано програмиране

Алгоритмите се реализират под формата на обекти, които имат вътрешна организация (включваща както данни, така и методи за обработката на данните и обектите), и които си обменят съобщения (обращения)

Абстрактното представяне на данните и методите за група сходни обекти се нарича клас, а отделните обекти от класа – представители

В езика Python класовете се реализират под формата на тип данни

Методите, които обработват данни и реализират комуникацията, се реализират чрез функции

Елементарните данни се реализират като обекти, представители на даден клас (тип данни).

Дефиниция на ООП (Alan Kay, 1993)

- Всичко е някакъв обект
- Изчисления: обмяна на съобщения между обекти
- Всеки обект има своя памет (която включва други обекти)
- Всеки обект е екземпляр (instance) на клас
- Класът е хранилище на поведение (реализирано чрез методи, които задават действия). Всички екземпляри на даден клас могат да изпълняват едни и същи действия.
- Всички класове са организирани в йерархично дърво, което представя йерархия на наследяване на свойства между класовете
- При проектирането на класовете и обектите се спазват принципите на капсулация, полиморфизъм, абстракция и наследяване

Основни понятия

- Всеки клас включва данни (атрибути, полета) и функции (методи). Понякога говорим за атрибути данни (полета) и атрибути методи (функции).
- Всеки обект е представител (*instance)* на клас (от един клас могат да се създадат много обекти)
- Всеки вграден тип данни в Python е клас
- Всяка константа (елемент) от тип данни в Python е обект
- Класовете и типовете данни в Python на свой ред са обекти от типа данни / класа ***type***

Основни понятия

- Всеки обект в езика Python има собствена памет, която се съхранява с помощта на структура от данни от тип речник
- Всеки елемент от този речник представя отделен атрибут на обекта – всеки атрибут на свой ред е обект
- Всеки атрибут се представя в речника с двойка name : val, където name е името на атрибута, а val е неговата стойност
- Има два различни вида атрибути: за данни и за поведение (методи)
- Всеки обект има поведение, наследено от класа, което се реализира чрез методи (функции) дефинирани в класа.
- Всеки обект в езика Python има уникален идентификатор

Предимства на ООП

- **Свързва данните с операциите (функциите) върху тях** в единен пакет чрез коректно зададени интерфейси
- Програмиране чрез принципа **разделяй и владей**
  - Отделно създаване и тестване на всеки клас
  - Увеличена модулност за намаляване на сложността
- Класовете позволяват **многократно използване** на кода
  - Много модули в Python дефинират нови класове
  - Всеки клас има собствено пространство (елиминира проблема с имената на функциите)
  - Наследяването позволява чрез под-класове да се променя или разширява поведението на даден клас (суперклас)

Създаване и използване на нови типове данни в Python чрез класове

- Различно е **създаване на клас** от **използване на представител (instance)** на клас
- **създаване** на клас включва:
  - Задаване на име на класа
  - Дефиниране на неговите атрибути
  - *например, някой е написал код за работа с клас списък*
- **използване** на класа включва:
  - Създаване на нови обекти **(представители, instances)**
  - Извършване на операции над тези обекти
  - *например, L=[1,2] и len(L)*

Възникване на type annotations

- Изискване на част от професионалните организации
- Дефиниране на тип на аргументите във функциите е въведено в Python 3.0 през 2006 от Guido van Rossum
- Официално и пълно навлиза в езика с Python 3.5 (2015)
- Усъвършенствано в Python 3.6 (2016) и Python 3.7 (2018)

Какво представлява type annotation


```
# в дефиниция на функция
def my_func(a : int, b : str = "") -> bool:
    # ...

# Променливи (налично от версия 3.6)

a: int = 0
b: str
```

Налични типове за анотации – typing модул


```
from typing import *
# задава тип данни списък, с елементи цели числа
my_list_int: List[int] = [1,2,3]
# при повече възможни типове данни на елементи - Union
my_multi_type_list: List[Union[bool, int]] = [True, 33]
# редица с 1-и елем. int, 2-и елем. str, 3-и елем. float
my_tuple: Tuple[int, str, float] = (1, "abc", 3.14)
# редица с произволен брой елементи реални числа
my_float_tuple: Tuple[float, ...] = (11.14, 20.18, 0.1)
# задава тип данни списък, с елементи произволен тип
my_list_int: List[Any] = [1, 2.0, (3, "abc")]
```

Налични типове за анотации – typing модул


```
my_dict: Dict[str, int] = { "33": 17 }
# Комбиниране на контейнери
school_coords: Dict[ str, Tuple[int, int] ]
school_coords = {"Epita": (10, 20)}
# None е валиден тип данни, като Any
def f(a: None) -> int:
    ...
# None се използва в Union:
def f(a: Union[None, int]) -> int:
    ...
# Union[None, int] понякога се задава като Optional[int]
def f(a: Optional[int] = None) -> int:
    ...
```

Как Python обработва type annotations

- Анотациите са валидни изрази и се анализират от интерпретатора
- Резултатите се съхраняват в специален атрибут (__annotations__)
- След което се игнорират от Python по време на изпълнение

Type annotations се използват от външни инструментални средства : среди за програмиране, среди за валидиране на типове данни и т.н.

Къде да добавим type annotation


```
# във функциите и методите

# не на променливи на които даваме стойност
vat_rate = 20      # OK, vat_rate е от тип int

# освен ако сме сгрешили …
if reduced_vat:
    vat_rate = 5.5 # грешка от валидатор, vat_rate е int!


vat_rate: float = 20 # OK за float и int стойности
```

Къде да добавим type annotation


```
# За всеки празен контейнер

names = [] # не е ясно елементите от какъв тип данни са

names: List[str] = [] # OK

birth_dates:  Dict[str, Date]
birth_dates = {}
```

Команда за създаване на клас (тип)

- използва се ключовата дума class за създаване на дефиниция на клас:

```
class Coordinate(object):
    # в тялото се задават атрибутите
```

- подобна на def, с изместване на командите се указват тези, които са част от **дефинирането на класа (типа)**
- аргумента object означава че класа Coordinate е под-клас на класа object и **наследява** всички негови атрибути
  - Coordinate е под-клас (наследник) на object
  - object е супер-клас (родител) за Coordinate

Елементи на клас

- Това са данните и методите зададени в класа
- **Атрибути - данни**
  - Това са конкретни обекти (константи)
  - *например, ирационалните числа се задават с две реални*
- **Методи - функции**
  - Това са функции които са специфични за този клас (тип)
  - Те задават различни операции над обектите от класа
  - *например можем да получим имагинерната част на числото, което е безсмислено за цели или реални числа*

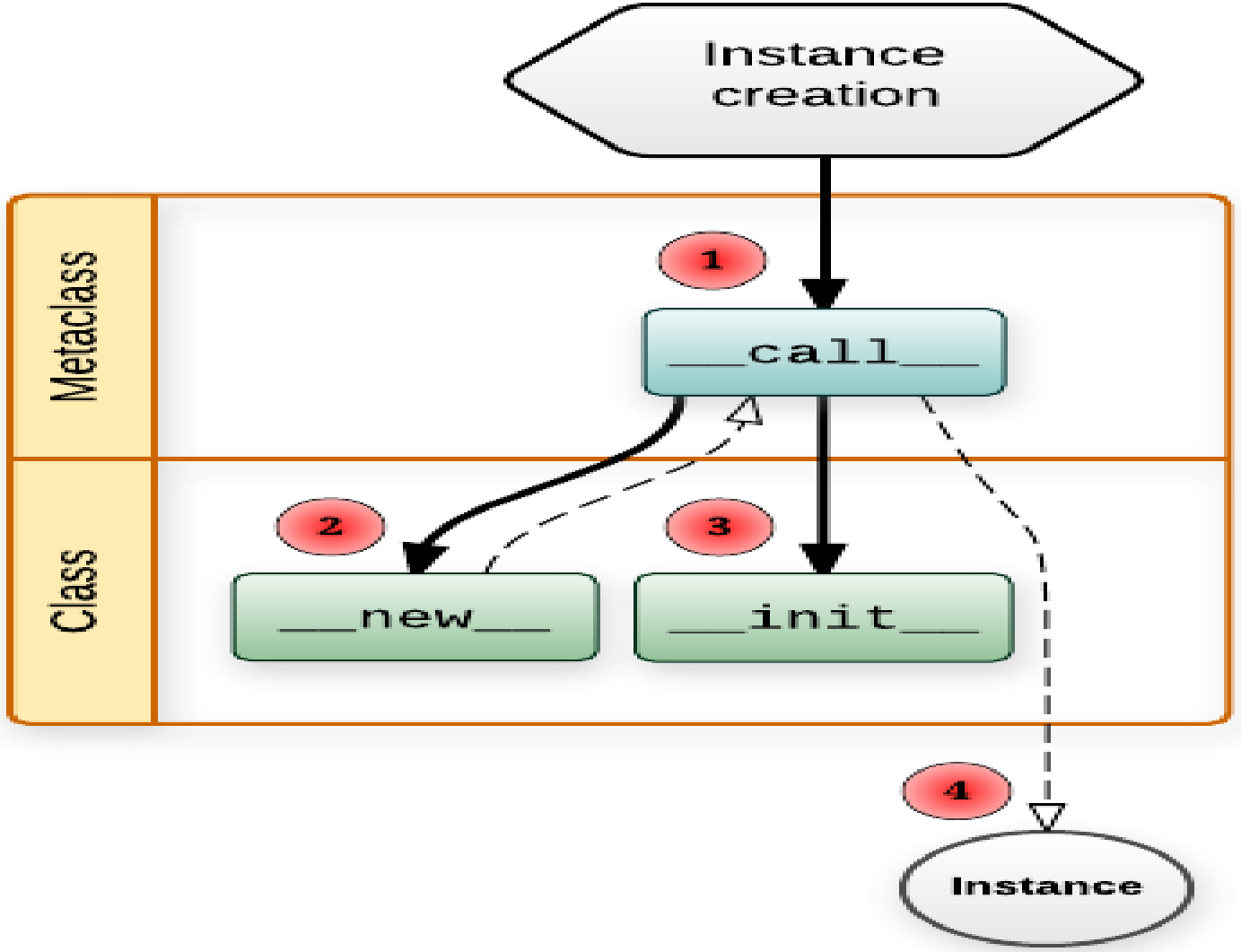
Създаване на обект (instance) от даден клас


```
c = Coordinate(3,4)          # създава обект с коорд. 3,4
origin = Coordinate(0,0)     # създава обект с коорд. 0,0
print(c.x)          # използва име c.x за достъп до атрибут
print(origin.x)
```

Атрибутите данни, описани в дефиницията на клас, се наследяват от всеки нов обект, с конкретни стойности зададени при създаването на обекта

Достъпът до тях става обикновено така: <обект>.<атрибут>

Създаване на обекти (instances) в Python



Клас, обект, метод


```
class Person:
    def sayHi(self):
        print 'Hello, how are you?'
p = Person()
p.sayHi()
# това е еквивалентно на обръщението Person().sayHi(p)
# Обръщенията към методи и атрибути: <обект>.<атрибут>
```

Какво представляват Data Classes в езика Python

Модул като част от стандартната библиотека от версия 3.7
Наличен е също и от версия 3.6 (като допълнителен модул).
Предлага автоматично добавяне в дефиницията на клас от потребител на голям брой специални методи като __init__(), __repr__() и други. За целта използва функции и декоратори.
Първоначално е описан в PEP 557.
Атрибутите се описват чрез т.нар. type annotations, описани първо в PEP 526 и налични в последните версии на езика Python.

Примери за създаване на класове


```
from dataclasses import dataclass
@dataclass
class Point:
    x: int
    y: int
p1 = Point(0, 0)
print(f'{p1}: x={p1.x}, y={p1.y}')
Point(x=0, y=0): x=0, y=0
p2 = Point(x=0, y=0)
print(f'{p1} == {p2} -> {p1 == p2} ')
Point(x=0, y=0) == Point(x=0, y=0)              ->  True
```

Пример 2


```
""" Прост минимален пример """

from dataclasses import dataclass
from datetime import datetime

# Примерен клас с име PyCon
@dataclass
class PyCon:
    location: str
    date: datetime
    year: int = 2018  # стойност по подразбиране

# Получаваме готови методите init, repr & eq
# Има и много други, налични чрез параметрите на декоратора
```

Пример 3


```
from dataclasses import dataclass

@dataclass
class Colour:
    red: int
    green: int
    blue:  int

orange = Colour(red=255, green=165, blue=0)
red = Colour(255, 0, 0)
red2 = Colour(255, 0, 0)
red ==  red2
True
orange.key = 15
```

Пример 4

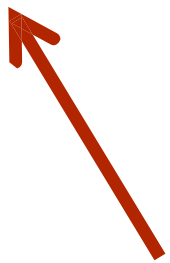

```
from dataclasses import dataclass

@dataclass
class Colour:
    red: int
    green: int
    blue: int

orange = Colour(red=255, green=165, blue=0)
red = Colour(255, 0, 0)
list(sorted([orange, red]))
Traceback (most recent call last): File "<input>",
  line 1, in <module>
TypeError:  '<' not supported between instances of 'Colour'
and 'Colour'
```

Пример 5

```
@dataclass(order=True)
Class Colour:
        red: int
        green: int
        blue: int
```



```
orange = Colour(red=255, green=165, blue=0)
red = Colour(255, 0, 0)
list(sorted([orange, red]))
```

[Colour(red=255, green=0, blue=0),
Colour(red=255, green=165, blue=0)]

Пример: създаване на immutable клас


```
@dataclass(order=True, frozen=True)
class Colour:
    red: int
    green: int
    blue: int
Red = Colour(255, 0, 0)
red.red = 240
```

Traceback (most recent call last):

File "<input>", line 1, in <module>

File "<string>", line 3, in ***setattr***

dataclasses.FrozenInstanceError: cannot assign to field 'red'

Сравнение на стандартна дефиниция на клас и създаване на клас с dataclass

Dataclass

```
@dataclass(order=True, frozen=True)
class Colour:
    red: int
    green: int
    blue: int
```

Стандартна дефиниция на клас

```
class Colour:

    def __init__(self, red, green, blue=0):
        object.__setattr__(self, "red", red)
        object.__setattr__(self, "green", green)
        object.__setattr__(self, "blue", blue)

    def __repr__(self):
        return (f"{self.__class__.__name__}("
                f"red={self.red}, green={self.green}, "
                f"blue={self.blue})")

    def __eq__(self, other):
        if other.__class__ is self.__class__:
            return (self.red, self.green, self.blue) == (
                    other.red, other.green, other.blue)
        return NotImplemented


    def __ge__(self, other):
        if other.__class__ is self.__class__:
            return (self.red, self.green, self.blue) >= (
                    other.red, other.green, other.blue)
        return NotImplemented

    def __gt__(self, other):
        if other.__class__ is self.__class__:
            return (self.red, self.green, self.blue) > (
                    other.red, other.green, other.blue)
        return NotImplemented


    def __le__(self, other):
        if other.__class__ is self.__class__:
            return (self.red, self.green, self.blue) <= (
                    other.red, other.green, other.blue)
        return NotImplemented


    def __lt__(self, other):
        if other.__class__ is self.__class__:
            return (self.red, self.green, self.blue) < (
                    other.red, other.green, other.blue)
        return NotImplemented

    def __delattr__(self, name):
        raise TypeError(f'cannot delete field {name}')

    def __setattr__(self, name, value):
        raise TypeError(f'cannot assign to field {name}')

    def __hash__(self):
        return hash((self.red, self.green, self.blue))
```

Параметри за dataclass


```
def dataclass(
        cls = None, /, *,          # име на клас, подредба на параметри
        init = True,               # дали да се генерира метод __init__()
        repr = True,               # дали да се генерира метод __repr__()
        eq = True,                 # дали да се генерира метод __eq__()
        order = False,             # дали да се генерират методи за сравнение
        unsafe_hash = False,       # дали да се генерира метод __hash__()
        frozen = False) :          # дали обектите да са mutable
```

Проблеми с някои полета


```
from typing import List

@dataclass
class Employee:
    name: str
    surname: str
    children: List[str] = []
john = Employee('John’, 'Green’)
hank = Employee('Hank’, 'Green’)
john.children.append('Henry')
```

Съобщение за грешка


```
Traceback (most recent call last):
  File "<input>", line 3, in <module>
  File "dataclasses.py", line 966, in dataclass
    return wrap(_cls)
  File "dataclasses.py", line 958, in wrap
    return _process_class(cls, init, repr, eq, order,
unsafe_hash, frozen)
  File "dataclasses.py", line 809, in _process_class
          For name, type in cls_annotations.items()] File
        "dataclasses.py", line 809, in <listcomp> for name, type
                    in cls_annotations.items()]
  File "dataclasses.py", line 702, in _get_field
    raise ValueError(f'mutable default
{type(f.default)} for field '
ValueError: mutable default <class 'list'> for field children is not
        allowed: use default_factory
```

Правилен вариант с поле (field)


```
From dataclasses import dataclass, field
From typing import List

@dataclass class Employee:
    name: str
    surname: str
    children: List[str] = field(default_factory=list)

john = Employee('John', 'Green‘)
hank = Employee('Hank', 'Green')
john.children.append('Henry')
john.children.append('Alice’)

>>> hank.children
[]
```

Вградена функция field

Тази функция се използва за задаване на начална стойност на някои атрибути, в ситуации в които може да има проблеми с директно задаване на такава стойност

```
def field(*,   # само една от default се задава
          default=MISSING, # задава стойност по подразбиране
          default_factory=MISSING, #също, но с функция
          init=True,   # да се използва като атрибут в init()
          hash=None,   # да се използва като атрибут в hash()
          repr=True,   # да се използва като атрибут в repr()
          compare=True, # да се използва за сравняване
          metadata=None): # за ползване от други програми
```

Задаване променливи на клас


```
from typing import ClassVar

@dataclass
class Colour:
    MAX:ClassVar[int]=255
    red: int
    green: int
    blue: int = 0

red = Colour(255, 0)
orange = Colour(255, 165)
>>> red.MAX
255
```

Атрибути на обекти (представители)


```
from dataclasses import dataclass, field, InitVar
# С InitVar се задават атрибути на обекти

@dataclass
class Address:
    protocol: InitVar[str]
    host: InitVar[str]
    url: str = field(init=False)

# Това е метод за допълнителна инициализация на атрибути
    def __post_init__(self, protocol, host):
        self.url = f'{protocol}://{host}/'

>>> addr = Address('http', 'google.com')
>>> addr
Address(url='http://google.com/')
```

Наследяване на класове


```
@dataclass
class  Point1d:
    x:  float
@dataclass
class  Point2d(Point1d):
    y:  float
@dataclass
class  Point3d(Point2d):
    z:  float

>>> p1 = Point1d(11.1)
>>> p2 = Point2d(22.2, 33.3)
>>> p3 = Point3d(44.4, 55.5, 66.6)
>>> p1,  p2,  p3
(Point1d(x=11.1),  Point2d(x=22.2,  y=33.3),  Point3d(x=44.4,  y=55.5, z=66.6))
```

Неизменяеми обекти със __slots__


```
@dataclass
class Colour:
    __slots__=('red', 'green', 'blue')
    red: int
    green: int
    blue: int

>>> red.check = True
Traceback (most recent call last):
File "<input>", line 1,    in <module>
AttributeError: 'Colour’ object has no attribute 'check'
```

Налични методи и атрибути в dataclass

Методи:

- asdict – преобразува обект в речник (от двойки атрибут:стойност)
- astuple – преобразува обект в редица от стойности на атрибути
- make_dataclass – за създаване на dataclass (на практика: dataclass())
- is_dataclass – проверява дали обекта е от клас dataclass (isinstance)
- replace – създава нов обект с посочени стойности на атрибути
- и много други …

Атрибути:

- __annotations__ # атрибут съдържащ всички анотации на променливи
- _DataclassParams # какви стойности са използвани за параметри
- fields # редица от всички атрибути (fields) за даден обект от клас dataclass

Примери


```
>>> dataclasses.asdict(red)
{'red': 255, 'green': 0, 'blue': 0}
>>> dataclasses.astuple(red)
(255, 0, 0)
>>> red2 = dataclasses.replace(red, blue=12)
>>> red2
Colour(red=255, green=0, blue=12)
>>>
```

Заключение

- Модулът dataclasses може съществено да подпомогне работата с класове и обекти е Python
- Наличен е като част от системната библиотека (не изисква инсталация)
- Използва много от съществените подобрения в езика
- Повишава разбирането и производителността

Пакет attrs


```
pip install attrs  # https://www.attrs.org

import attr

@attr.s
class Colour:
    red: int = attr.ib()
    green: int = attr.ib()
    blue: int = attr.ib()

>>> red = Colour(255, 0, 0)
>>> red
Colour(red=255, green=0, blue=0)
```

Класовете като обекти

Класовете (като функциите) са обекти от най-висок ред:

- могат да бъдат създавани динамично по време на изпълнение на програмата
- могат да бъдат подавани като аргументи
- могат да бъдат връщани като резултат от изпълнението на функции и други изпълними обекти (callable)
- Могат да бъдат присвоявани на променливи.

Клас, който се използва за създаване на обекти-класове, се нарича мета-клас

Унищожаване на класове и обекти

- del(<име_обект>) за премахване на връзката на променлива с обект
- Деструктор __del__ който се извиква преди унищожаване на обекта (с автоматичната процедура или след края на програмата)
- del(<име_клас>) за явно унищожаване на клас
- Автоматично изтриване на обект когато няма свързана с него променлива

Достъп до атрибути в клас / обект

Нормален достъп до атрибут <attr> от клас / обект <obj>:

<obj>.<attr> # стандартен достъп

Аналогичен достъп с вградена функция:

getattr(<obj>, '<attr>')

директен достъп с вградена функция

Пример за достъп до атрибути


```
>>> f = student("Bob Smith", 23)

>>> f.full_name # достъп до атрибут данни
"Bob Smith"

>>> f.get_age() # достъп до метод
23
```

Елементи – методи и атрибути

- Методът е процедурен атрибут, който е **специална функция достъпна само в класа**
- Python винаги предава обектът като първи аргумент на всеки метод
  - За целта в аргументите на метода се използва **self** като име на обекта (първи аргумент)
- Специалният **оператор** “**.**” се използва за достъп до всеки елемент от обект:
  - Атрибутите за данни
  - Методите като процедурни елементи

Използване на метод


```
def distance(self, other):
    # някакви команди
```

- Стандартен начин

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

- Еквивалентен е на

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

Достъп до неизвестен атрибут

- Ако не знаем името на атрибут за данни или метод в клас, тогава можем да използваме функцията dir:
  dir(<class_name>), която връща имената на всички атрибути в класа
- Присвояваме на променлива string името на някой атрибут (получено по време на изпълнение на програма чрез dir)
- След това с функцията getattr(object_instance, string) виждаме стойността на атрибута, т.е. е еквивалентна на object_instance.string

getattr(object_instance, string)


```
>>> f = student('Bob Smith', 23)
>>> getattr(f, 'full_name')
'Bob Smith'
>>> getattr(f, 'get_age')
 <method get_age of class studentClass
 at 010B3C2>
>>> getattr(f, 'get_age')() # вика го
23
>>> getattr(f, 'get_birthday')
# AttributeError – Няма такъв метод!
```

hasattr(object_instance,string)


```
>>> f = student('Bob Smith', 23)
>>> hasattr(f, 'full_name')
True
>>> hasattr(f, 'get_age')
True
>>> hasattr(f, 'get_birthday')
False
```

setattr(object_instance, string, val)

```
>>> f = student('Bob Smith', 23)
>>> setattr(f, 'full_name', 'Rob Jhon')
>>> getattr(f, 'full_name')
'Rob Jhon'
>>> setattr(f, 'birthday', '12_05_1986')
>>> getattr(f, 'birthday')
'12_05_1986'
```

Два вида атрибути (за данни)

- Атрибути на обект (instance attributes)
  - Обекти налични в *екземпляр (instance)* на клас
  - Всеки екземпляр има обект с различна стойност
  - По правило се инициализират с дефиницията на dataclass или с метода __init__ (при нормално създаване на клас)
- Атрибути на клас (class attributes)
  - Налични само в пространството на класа
  - *Общи са за всички екземпляри*
  - В някои езици за програмирани се наричат още статични (“static”) променливи
  - Полезни за общи константи за клас или за броячи (например колко пъти е извикан клас или метод)

Атрибути за клас

- Всички обекти споделят едно копие на атрибут.
  Ако *някой* обект промени стойността му, е за *всички*
- Class attributes се задават *вътре* в дефиницията на клас и *извън* дефинициите на всички методи
- Тъй като всеки такъв атрибут *за клас* не зависи от конкретен *обект*, за безопасен достъп до тях използваме нотацията **self.__class__.name** – което е най-безопасният начин на достъп (атрибута __class__ задава името на класа за обекта self)

```
class sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Алтернативен достъп до клас атрибут

- Вместо описания начин:

```
class sample:
   x = 23
   def increment(self):
      self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Можем да изпуснем името self защото метода не е свързан с конкретен обект (специален вид метод: Class method)

```
class sample:
   x = 23
   def increment():
      __class__.x += 1
```

```
>>> a = sample()
>>> sample.increment()
>>> a.x
24
```

Класове и типове данни

- Тип на обект – неговият клас

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> type(c)
<class _main_.Coordinate>
```

- Тип на клас – тип данни

```
>>> print(Coordinate)
<class _main_.Coordinate>
>>> type(Coordinate)
<class 'type'>
```

- проверка с isinstance() дали обект е от клас Coordinate

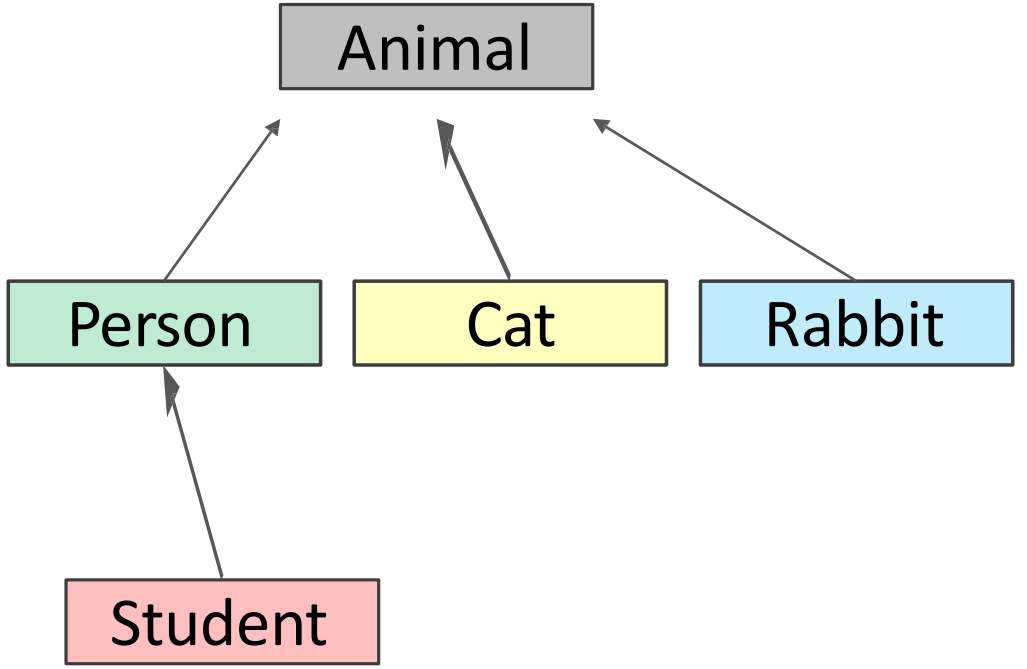
```
>>> isinstance(c, Coordinate)
True
```

Наследяване

- **В Python обектите наследяват класовете, а класа наследява родителските класове (суперкласове)**
- **За да наследи един клас други, трябва тези други да се укажат в скоби при неговата дефиниция**
- **Обектите наследяват атрибутите и методите както от своя клас, така и от всички негови родители**
- Всеки атрибут не дефиниран в обект се търси от долу нагоре, от ляво на дясно в йерархията от всички класове
- Промяна в логиката се прави в под-класовете, а не в родителските класове

Йерархии от класове

- Клас родител
- (superclass)
- Клас наследник
- (под-клас, subclass)
  - **наследява** атрибути и методи от родителя
  - **добавя** повече **данни**
  - **добавя** повече **методи**
  - **променя** методи



Наследяване

- Често има нужда от класове, които споделят някои (много, или всички) атрибути на даден клас, но са различни.
- Пример 1: Геометрия
  - Точка (в равнина) има координати *x* и *y*
  - Окръжност се представя като точка (център) и радиус
  - Цилиндър се представя с окръжност и височина
- Пример 2: Университетска система (СУСИ)
  - Всеки човек има ЕГН и адрес
  - Студент е човек и изучава предмети
  - Преподавател е човек и преподава предмети, и др.
- В тези случаи имаме *основен клас* и дефинираме другите класове като негово разширение (допълнение)
- Това се нарича *наследяване*.

Наследяване (2)

- Ако клас A наследява клас B задаваме:
  class A(B):
- Можем да извикваме конструктора на базовия клас явно (като Circle се вика от Point.__init__(...))
- Класовете наследници могат да ползват атрибутите и методите от базовия клас (например обектите cylinder и circle наследяват атрибутите **x** и **y** от клас point).

Заключение

- Основни понятия
- Обекти и класове
- Атрибути и методи
- Създаване и използване на класове и обекти
- Специални методи и атрибути
- Промяна в действие на оператори
- Наследяване