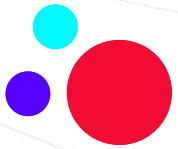
infoShare ACADEMY

Python Podstawy – Intel

infoShare Academy

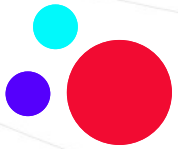
infoShareAcademy.com



OOP – Object Oriented Programming

w Python – **wszystko jest obiektem** i posiada typ

- obiekty są abstrakcjami danych, które zawierają:
  - wewnętrzną reprezentację poprzez **właściwości**
  - interfejs do interakcji z obiektem poprzez **metody**
- można tworzyć nowe instancje klas (obiekty)
- można niszczyć obiekty
  - wyraźnie – używając metody del
  - 'zapomnieć' – Garbage Collector usunie niedostępne obiekty



OOP – Object Oriented Programming

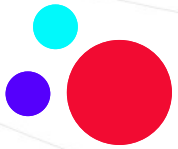
2.343 'Magdalena ' [1, 2, {'imie': Andrzej', 'nazwisko': 'kowalski'}]

Dane ww. są instancjami klasy, każdy obiekt ma:

- typ
- dane - informacje proste (int, float...), złożone (inne obiekty)
- interfejs - procedury do interakcji z obiektem <metody>

1234 jest instancją klasy **int**

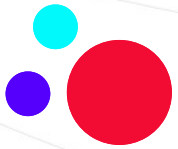
x = 'Natalia' – x jest instancją klasy **string**



Klasa vs Instancja

- **KLASA**
  - jest "ideą", "schematem", "wyobrażeniem"
  - określa właściwości i metody

- **INSTANCJA**
  - jest "powołanym do życia" **obiektem**, który zawiera określone przez klasę atrybuty (właściwości i metody).
  - można mieć kilka instancji jednej klasy



Klasa vs Instancja

- **Do stworzenia klasy potrzebujemy:**
  - nazwy klasy
  - zdefiniować właściwości i metody klasy

- **Używanie klasy polega na:**
  - utworzeniu nowego obiektu
  - wykonywaniu operacji na obiekcie

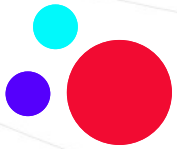
Paradygmaty OOP

abstrakcja

enkapsulacja / hermetyzacja

polimorfizm

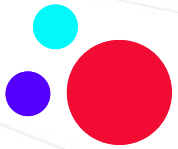
dziedziczenie



Abstrakcja

Uproszczenie rozpatrywanego problemu, polegające na ograniczeniu cech obiektów wyłącznie do kluczowych dla algorytmu, a jednocześnie niezależnych od implementacji.

infoShare ACADEMY

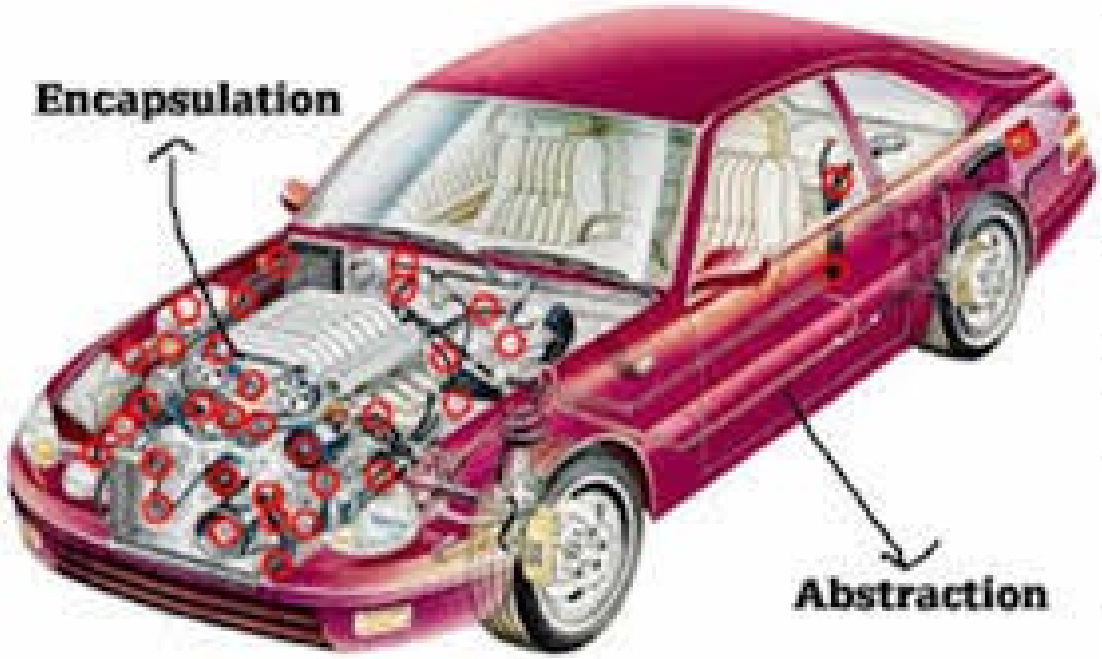


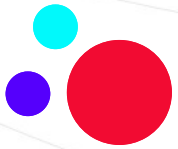
Enkapsulacja

Polega na „ukrywaniu” pewnych danych składowych lub metod danej klasy tak, aby były one dostępne tylko metodom wewnętrznym danej klasy

Właściwe zachowanie obiektu, może być zagrożone, jeśli będziemy manipulować bezpośrednio na wnętrzu obiektu – należy używać zdefiniowanych interfejsów (pól i metod)

Abstrakcja i enkapsulacja

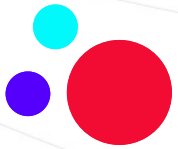




Polimorfizm

Kilka implementacji jednego polecenia pod wspólną nazwą tworzy wygodny abstrakcyjny interfejs niezależny od typu wartości na której operujemy.

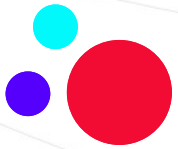
Niezależnie czy należy wydrukować **liczbę** czy **napis**, czytelniej jest gdy operacja taka nazywa się po prostu **drukuj**, a nie drukuj_liczbę i drukuj_napis. Jednak napis musi być drukowany inaczej niż liczba, dlatego będą istniały dwie implementacje polecenia drukuj, ale nazwanie ich wspólną nazwą tworzy wygodny abstrakcyjny interfejs niezależny od typu drukowanej wartości.



Dziedziczenie

Porządkuje i wspomaga polimorfizm i enkapsulację dzięki umożliwieniu definiowania i **tworzenia specjalizowanych obiektów na podstawie bardziej ogólnych**.

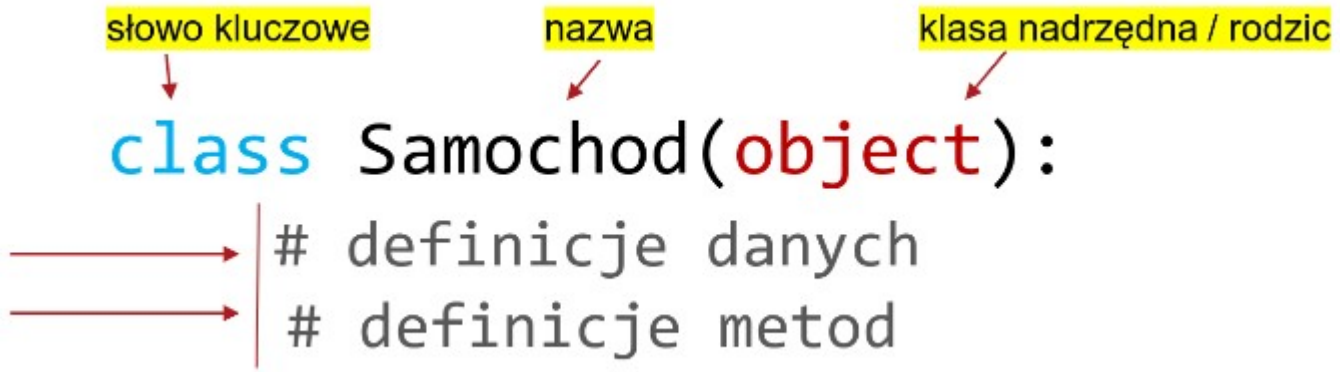
Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy.



Zalety OOP

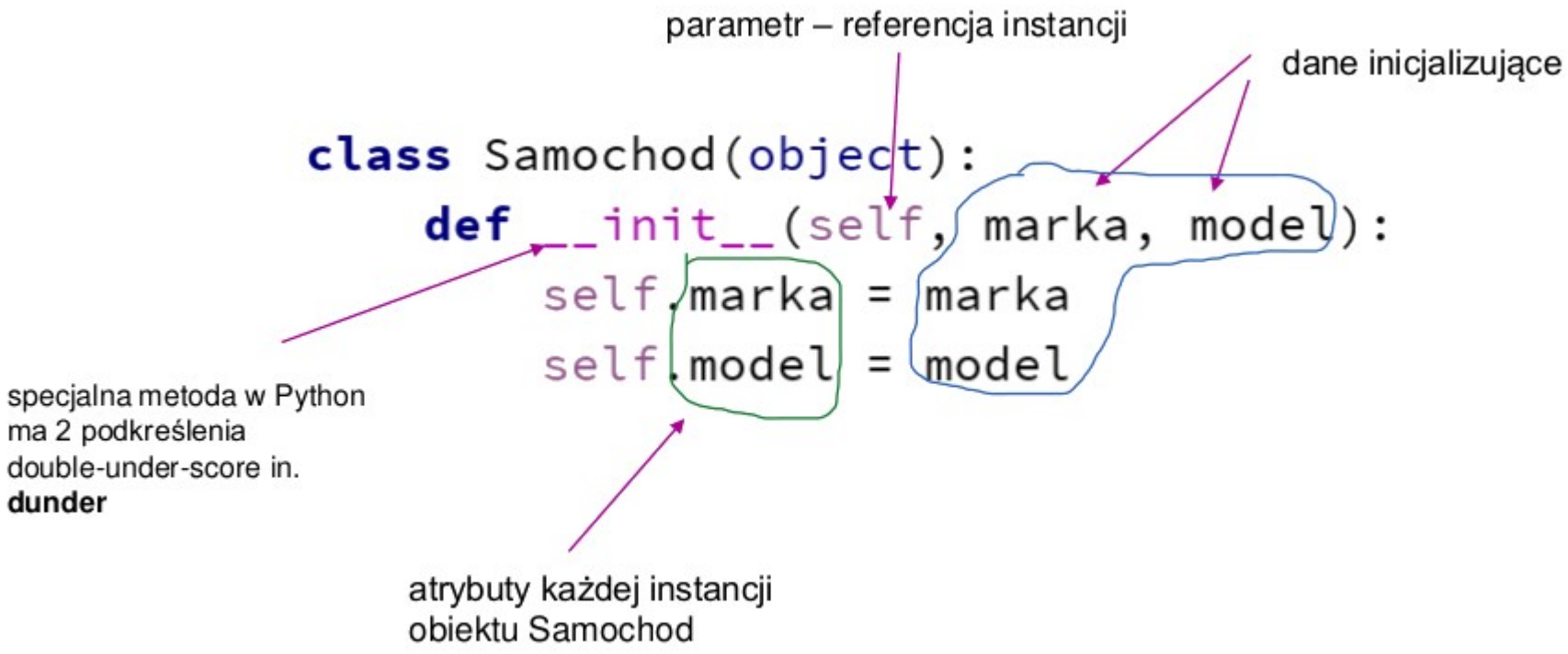
- **tworzenie jednorodnego pakietu**, zawierającego dane oraz sposoby manipulowania nimi
- umożliwiają podejście – **divide and concquer** (dziel i zwyciężaj)
  - można testować zachowanie każdej z klas oddzielnie
  - zwiększa modularność, zmniejsza kompleksowość
- **klasy ułatwiają ponowne użycie kodu**
  - każda z klas tworzy oddzielne "środowisko" – różne klasy mogą mieć takie same nazwy funkcji
  - dziedziczenie pozwala aby podklasa, zredefiniowała lub rozszerzyła wybrane właściwości klasy nadrzędnej

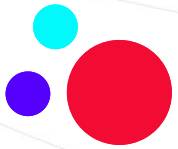
Definiowanie klasy



- **class** słowo kluczowe podobnie jak def przy definiowaniu funkcji
- **object** oznacza że Samochod jest obiektem w Python i dziedziczy z niego wszystkie jego właściwości:
  - Samochod jest podklasą object
  - Object jest klasą nadrzędną dla Samochod

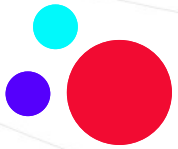
Definiowanie konstruktora





Definiowanie metod

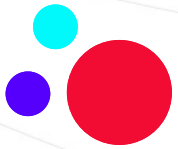
```
def accelerate(self, value):
    self.speed += value
```



Metody specjalne – dunder

Określenie ich w klasie umożliwia zdefiniowanie własnych zachowań dla operatorów i metod specjalnych

`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__eq__(self, other)`	`self == other`
`__lt__(self, other)`	`self < other`
`__len__(self)`	`len(self)`
`__str__(self)`	`print(self)`



Definiowanie dziedziczenia

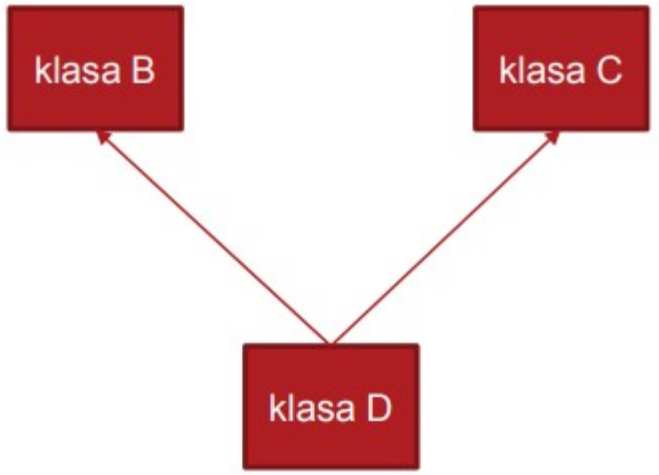
```
class Zwierze(object):
    # definicje danych
    # definicje metod

class Czlowiek(Zwierze):
    # definicje danych
    # definicje metod

class Student(Czlowiek):
    # definicje danych
    # definicje metod
```

Dziedziczenie umożliwia tworzenie klas, które korzystają z atrybutów klas nadrzędnych (superklasa / rodzic).

Klasy dziedziczące (podklasy / dzieci) mogą część atrybutów mieć zdefiniowanych według własnych potrzeb.

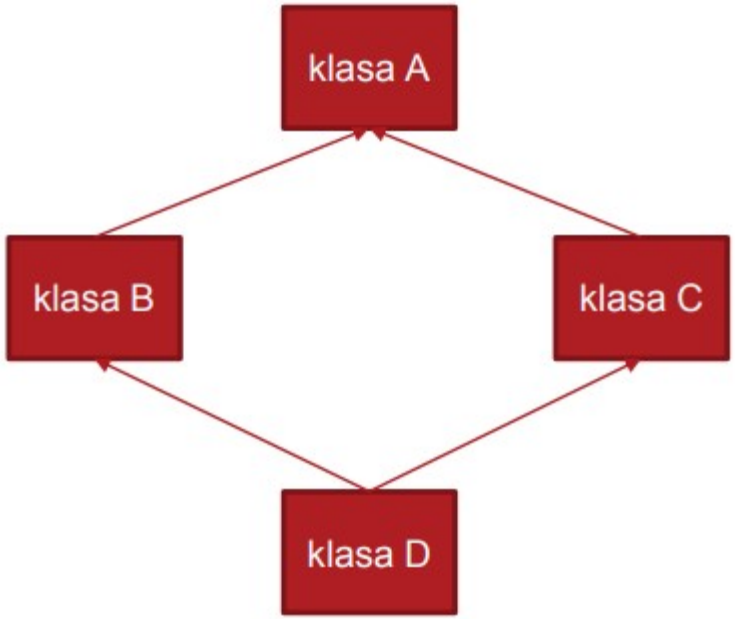
Dziedziczenie od wielu rodziców

Klasa może dziedziczyć z wielu klas




```
class B (object):
    pass

class C (object):
    pass

class D (B, C):
    pass
```

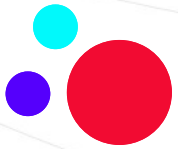
Dziedziczenie diamentowe




```
class A (object):
        pass
class B (A):
        pass
class C (A):
        pass
class D (B, C):
        pass
```

Klasa dziecka będzie szukać atrybutu od lewej do prawej, z dołu w górę.

W tym przykładzie, najpierw poszuka w klasie B, a następnie w klasie C



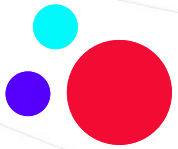
Pola klasy

Zmienne definiowane na poziomie klasy.

Nie używamy słówka **self**

Służą do przechowywania danych niezależnych od instancji (wspólne dla wszystkich instancji)

```
class Produkt:

    vat_rate = 0.23
```



(pseudo) prywatność

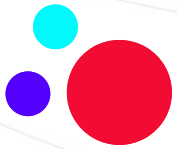
Python daje możliwość stworzenia pseudo-prywatnych pól i metod.

Do nazwy (pola, metody) dodajemy **podkreślniki tylko z przodu**.

Można je użyć wewnątrz klasy, ale poza nią są niewidoczne.

Można i tak ich użyć!!!

```
class MojaKlasa:

        __moje_pole_prywatne

        def __metoda_prywatna(self, arg1):

                self.__moje_pole_prywatne = arg1
```

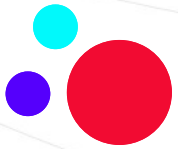


namespace

Namespace jest obszarem nazw, które są dostępne dla klasy.

```
print(MojaKlasa.__dict__)

print(obiekt.__dict__)

print(dir(MojaKlasa))

print(dir(obiekt))
```

W ten sposób możemy znaleźć pseudo-prywatny atrybut.



Metoda klasy

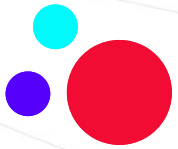
Metody, które jako pierwszy argument przyjmują klasę zamiast instancji.

Używamy **dekoratora @classmethod** nad definicją metody.

Pierwszy argument to słowo kluczowe **cls**

Możemy używać jako alternatywne konstruktory

```
@classmethod
def nazwa_metody(cls):
        pass
```



Metoda statyczne

Metody, które nie przyjmują ani instancji ani klasy jako argument.

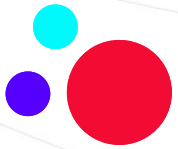
Wyglądają jak normalne metody

Używamy **dekoratora @staticmethod** nad definicją metody.

Używamy je gdy przekazanie jakiejś informacji nie wymaga

tworzenia instancji klasy. (matematyczne)

```
@staticmethod
def nazwa_metody():
        pass
```



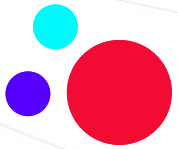
Properties

Metody które mogą nam przysłonić dostęp do właściwości pod inną i w kontrolwoany sposób nimi manipulować. Często używane do pól prywantych. Używane za pomocą dekoratorów

@property

@imie.setter

@imie.deleter



Properties

```
class Czlowiek(object):

        @property

        def imie(self):

                return str(self.__imie).capitalize()

        @imie.setter

        def imie(self, nowa_wartosc):

                self.__imie = nowa_wartosc

        @imie.deleter

        def imie(self):

                self.__imie = None
```

infoShare ACADEMY

DZIĘKUJĘ NA DZIŚ

Python Podstawy - Intel

infoShareAcademy.com