

# Python średnio zaawansowany

Dzień 1



# AGENDA KURSU

**Cel:** stworzenie aplikacji do pozyskiwania, przetwarzania, analizy i wizualizacji danych

- Blok nr 1 – wprowadzenie, powtórka i wyrównanie (3h)
- Blok nr 2 – akwizycja danych (9h)
- Blok nr 3 – przetwarzanie danych (18h)
- Blok nr 4 – analiza danych (12h)
- Blok nr 5 – aplikacja webowa (15h)
- Egzamin

# SPRAWY ORGANIZACYJNE

**Jesteście pierwsi w Polsce!**

**Czas trwania:** 20 spotkań po 3h każde, łącznie 60h. Listopad -> Styczeń

**Certyfikaty:** Po uzyskaniu 70% z egzaminu, na który składają się pytania zamknięte wielokrotnego wyboru (30-40) + zadanie praktyczne.

**Harmonogram:** wtorki i czwartki

**Kodeks** 😊

# SPRAWY ORGANIZACYJNE

**Zadania domowe** – warto, nie są obowiązkowe

**Repozytoria:**

<https://github.com/infoshareacademy/mpyws1-projekt>

<https://github.com/infoshareacademy/mpyws1-materialy>

**Slack:** <https://pythonintermediate1.slack.com/>

**Pół godziny dla kursanta** - na życzenie (Slack!)

# Powtórka i wyrównanie

# Obiekty

# Obiekty

```
1234      2.343534      'Magdalena'      [1, 3, 5, 7, 9]  
{ 'imie': 'Andrzej', 'nazwisko': 'Kowalski' }
```

Dane są instancjami klas, każdy obiekt ma:

- typ
- wewnętrzną reprezentację danych (prosta, złożona)
- zestaw procedur do interakcji z obiektem (in. interfejs)

Każda instancja jest konkretnym typem obiektu:

- 1234 jest instancją **int**
- x = 'Natalia' – x jest instancją **string**

# OOP - Object Oriented Programming

W języku Python – wszystko jest obiektem i posiada typ

- obiekty są abstrakcjami danych, które zawierają:
  - wewnętrzną reprezentację poprzez atrybuty danych
  - interfejs do interakcji z obiektem poprzez metody
- można tworzyć nowe instancje obiektów
- można niszczyć obiekty
  - wyrażnie – używając metody del
  - 'zapomnieć' – **Garbage Collector** usunie niedostępne lub zniszczone obiekty



# OOP

[1,2,3,4] ma typ list, jaka jest wewnętrzna reprezentacja?

L =

Jak można manipulować listami?

L[i], L[i:j], L[i:j:k], +

len(), min(), max(), del(L[i])

L.append(), L.extend(), L.count(), L.index(), L.insert(),  
L.pop(), L.remove(), L.reverse(), L.sort()

# OOP

Wewnętrzna reprezentacja obiektu powinna być **prywatna**

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

# Klasa vs instancja

**KLASA** – jest "idea", "schematem", "wyobrażeniem"  
właściwości (zmienne) i interfejs (metody)

**INSTANCJA** – jest "powołanym do życia" obiektem, który zawiera określone  
przez klasę właściwości.

Można mieć kilka instancji jednej klasy.

# Klasa vs instancja

Do stworzenia klasy potrzebujemy:

- nazwy klasy
- zdefiniować właściwości klasy

Używanie klasy polega na:

- utworzeniu nowej instancji
- wykonywaniu operacji na instancji

# Zalety OOP

- **tworzenie jednorodnego pakietu**, zawierającego dane oraz sposoby manipulowania nimi
- umożliwiają podejście – **divide and conquer** (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 klas

słowo kluczowe



nazwa



**class** Samochod:

→ # definicje danych  
→ # definicje metod

- **class** – podobnie jak **def**
- Samochód jest obiektem w Python

# Inicjalizacja obiektu

```
class Samochod:  
    def __init__(self, marka, model):  
        self.marka = marka  
        self.model = model
```

parametr – referencja instancji

dane inicjalizujące

specjalna metoda w Python  
ma 2 podkreślenia  
double-under-score in.  
**dunder**

atrybuty każdej instancji  
obiektu Samochod

# Definicja metod

parametr – referencja instancji

parametr metody

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



# Metody specjalne

# Metoda specjalna `__init__`

```
class Samochod(object):  
    def __init__(self, marka, model):  
        self.marka = marka  
        self.model = model
```

---

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

# Pozostałe metody specjalne

## OPERATORY:

+, -, ==, <, >, len(), print, in.

`__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)`

[Dokumentacja](#)

# Paradygmaty OOP

# Paradygmaty OOP

**ABSTRAKCJA** – uproszczenie problemu

**ENKAPSULACJA (HERMETYZACJA)** – ukrycie składowych

**POLIMORFIZM** – wielopostaciowość

**DZIEDZICZENIE** – mechanizm współdzielenia funkcjonalności

# Dziedziczenie

# Klasy, klasy, klasy ;-)

słowo kluczowe



nazwa



klasa nadrzędna / rodzic

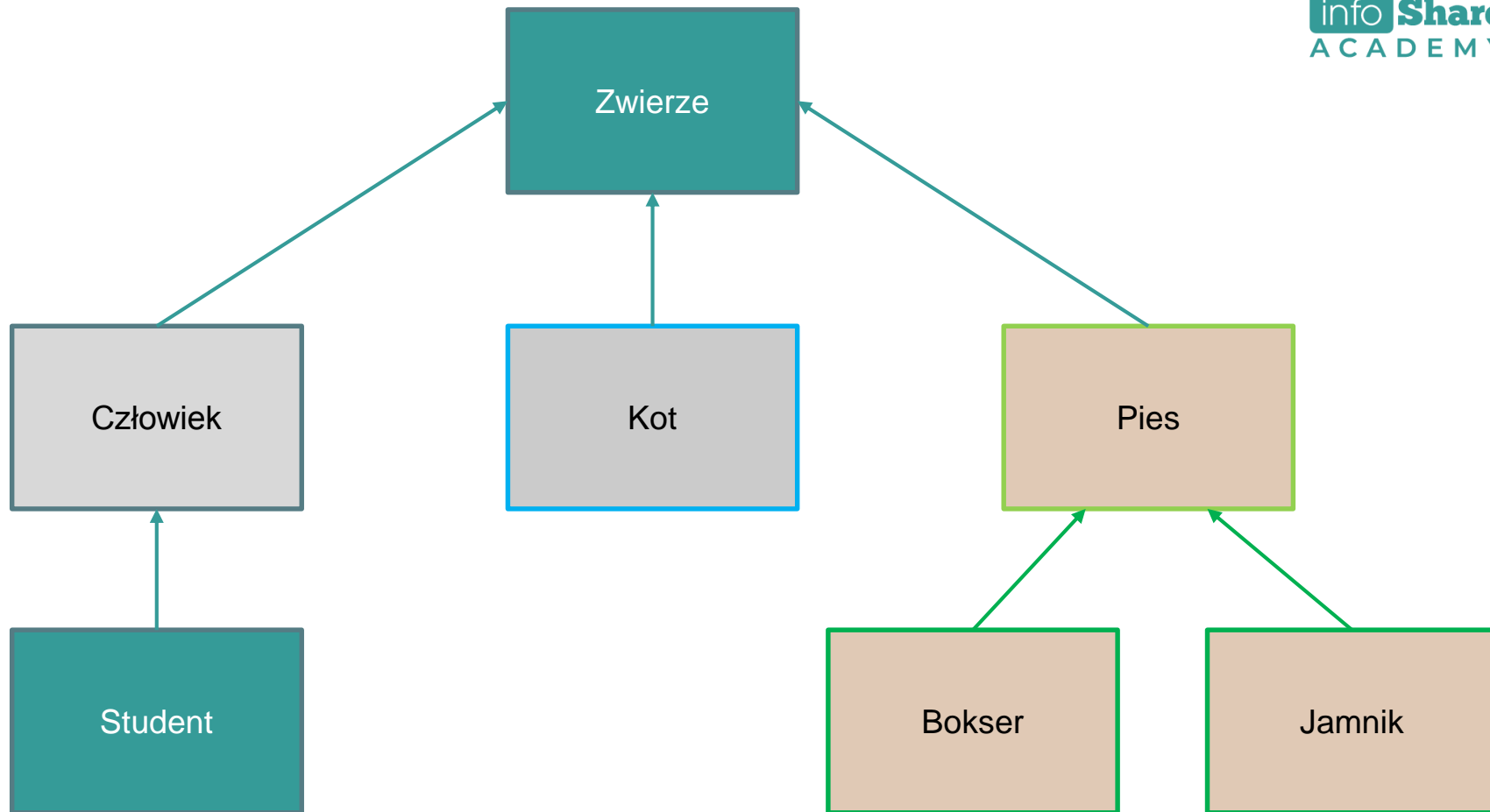


```
class Samochod(Pojazd):
```

```
# definicje danych
```

```
# definicje metod
```

- **class** – podobnie jak **def**
- słowo **Pojazd** oznacza, że Samochód jest obiektem w Python i **dziedziczy** z niego wszystkie właściwości
  - Samochod jest podklasą Pojazd
- - Pojazd jest klasą nadrzędną dla Samochod





# Definiowanie klas

```
class Zwierze:  
    # 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.

# Sprawdzanie zależności

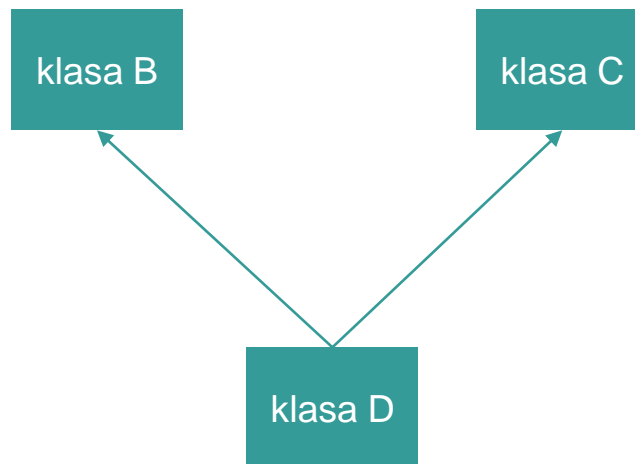
**isinstance(obiekt, klasa)** – sprawdza czy dany obiekt jest instancją klasy

**issubclass(klasaA, klasaB)** – sprawdza czy klasaA jest podklasą klasy B

# Dziedziczenie diamentowe

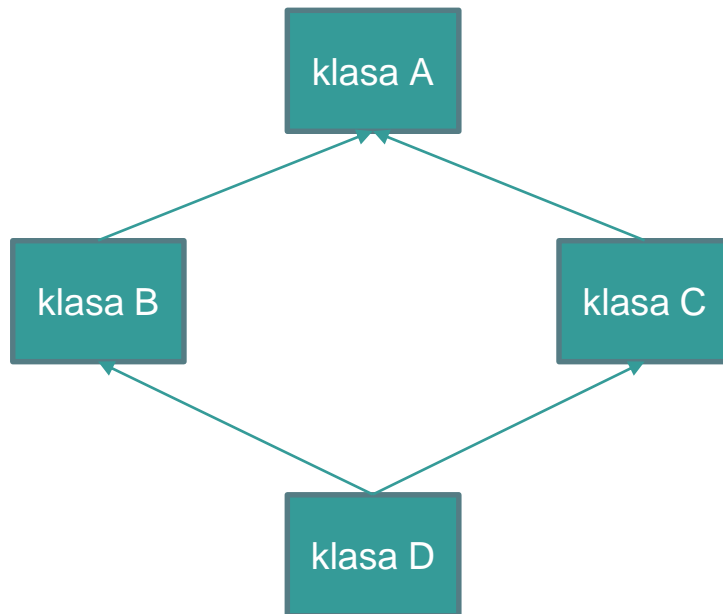
# Dziedziczenie od wielu rodziców

Klasa może dziedziczyć z wielu klas

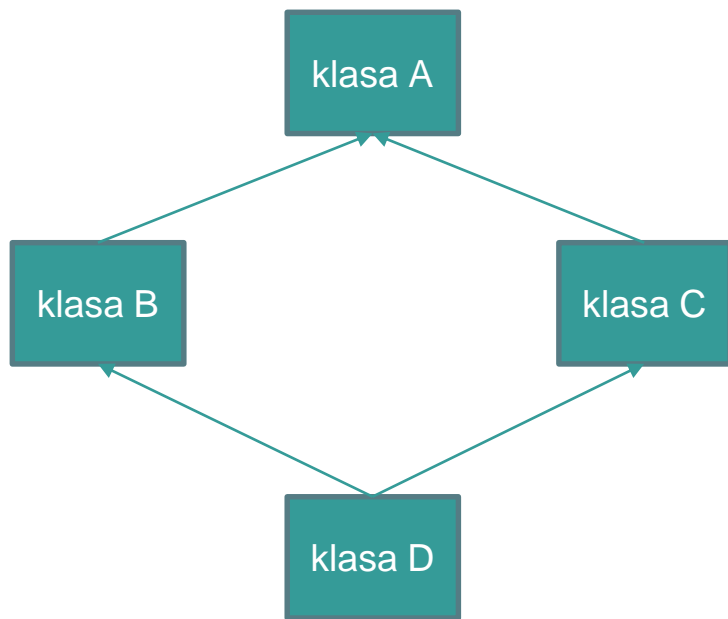


# Dziedziczenie diamentowe

Ale co w przypadku dziedziczenia diamentowego?



# Dziedziczenie diamentowe



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

W poniższym przykładzie, najpierw poszuka w klasie Horse, a następnie w Donkey

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

# Funkcja super

Musimy uważać jeśli dziedziczymy używając **super()** jako odwołanie do klasy nadrzędnej.

# Pola klasy, metody klasy i metody statyczne



# Pola klasy

Zmienne definiowane na poziomie klasy.

**Nie używamy** słówka **self**

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

# Metody 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 my_class_method(cls):
    pass
```

# Metody 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 my_static_method():  
    pass
```

```
MyClass.my_static_method()
```