Programowanie Python 1

(CP1S02005)

Politechnika Białostocka - Wydział Elektryczny

Cyfryzacja przemysłu, sem. II, studia stacjonarne I stopnia

Rok akademicki 2023/2024

Wykład nr 11 (22.05.2024)

dr inż. Jarosław Forenc

Plan wykładu nr 11

- Programowanie obiektowe
  - klasa - przykład
  - dziedziczenie
  - dziedziczenie wielokrotne
  - prawa dostępu

Python - prog. obiektowe (przykład)

- klasa opisująca trójkąt

```
class Trojkat:
    def __init__(self, a, h):
        self.a = a  # podstawa
        self.h = h  # wysokość

    def pole(self):
        return (self.a * self.h) / 2

    def __str__(self):
        return f"[a = {self.a}, h = {self.h}]"

tr1 = Trojkat(6, 8)
print(tr1)
print(f"Pole trójkąta: {tr1.pole():.2f}")
```

```
[a = 6, h = 8]
Pole trójkąta: 24.00
```

Python - prog. obiektowe (przykład)

- dostęp do atrybutów klasy

```
nazwa_egzemplarza.atrybut
```

```
tr1 = Trojkat(6, 8)
tr1.a = 5
tr1.h = 9
```

- wywołanie metody

```
nazwa_egzemplarza.metoda()
```

```
tr1 = Trojkat(6, 8)
p = tr1.pole()
print(f"Pole trójkąta: {tr1.pole():.2f}")
```

Python - prog. obiektowe

- można utworzyć wiele egzemplarzy (obiektów) na podstawie jednej klasy

```
class Osoba:

    def __init__(self, imie, nazwisko, wiek):
        self.imie = imie
        self.nazw = nazwisko
        self.wiek = wiek

    def __str__(self):
        return f"{self.imie} {self.nazw}, {self.wiek} lat"

nowak = Osoba("Jan","Nowak",25)
print(nowak)

kowal = Osoba("Ela","Kowal",18)
print(kowal)
```

```
Jan Nowak, 25 lat
Ela Kowal, 18 lat
```

Python - prog. obiektowe

- nie wszystkie atrybuty muszą mieć przekazywane wartości, mogą być atrybuty, które mają wartości domyślne

```
class Osoba:
    dzieci = 0

    def __init__(self, imie, nazwisko, wiek):
        self.imie = imie
        self.nazw = nazwisko
        self.wiek = wiek

    def __str__(self):
        return f"{self.imie} {self.nazw}, {self.wiek}
                    lat, dzieci: {self.dzieci}"

nowak = Osoba("Jan","Nowak",25)
print(nowak)
```

```
Jan Nowak, 25 lat, dzieci: 0
```

Python - prog. obiektowe

- nie wszystkie atrybuty muszą mieć przekazywane wartości, mogą być atrybuty, które mają wartości domyślne

```
class Osoba:

    def __init__(self, imie, nazwisko, wiek):
        self.imie = imie
        self.nazw = nazwisko
        self.wiek = wiek
        self.dzieci = 0

    def __str__(self):
        return f"{self.imie} {self.nazw}, {self.wiek}
                    lat, dzieci: {self.dzieci}"

nowak = Osoba("Jan","Nowak",25)
print(nowak)
```

```
Jan Nowak, 25 lat, dzieci: 0
```

Python - prog. obiektowe

- wartość atrybut można zmienić bezpośrednio w egzemplarzu (obiekcie)

```
nowak = Osoba("Jan","Nowak",25)
nowak.dzieci = 2
print(nowak)
```

```
Jan Nowak, 25 lat, dzieci: 2
```

- wartość atrybut można zmienić za pomocą metody

```
class Osoba:
    ...
    def zapisz_dzieci(self, liczba_dzieci):
        self.dzieci = liczba_dzieci

nowak = Osoba("Jan","Nowak",25)
nowak.zapisz_dzieci(3)
print(nowak)
```

Python - prog. obiektowe

- wartość atrybut można zmienić bezpośrednio w egzemplarzu (obiekcie)

```
nowak = Osoba("Jan","Nowak",25)
nowak.dzieci = 2
print(nowak)
```

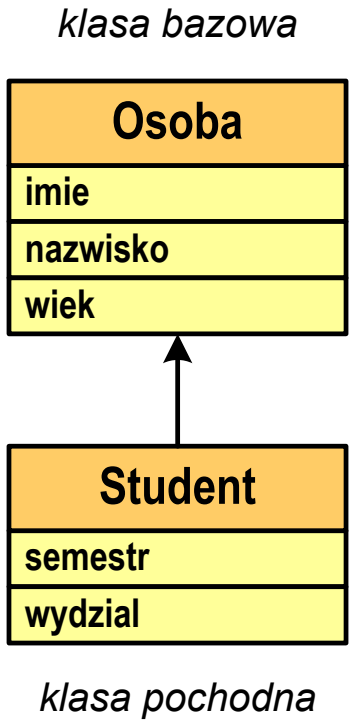
```
Jan Nowak, 25 lat, dzieci: 2
```

- wartość atrybut można zmienić za pomocą metody

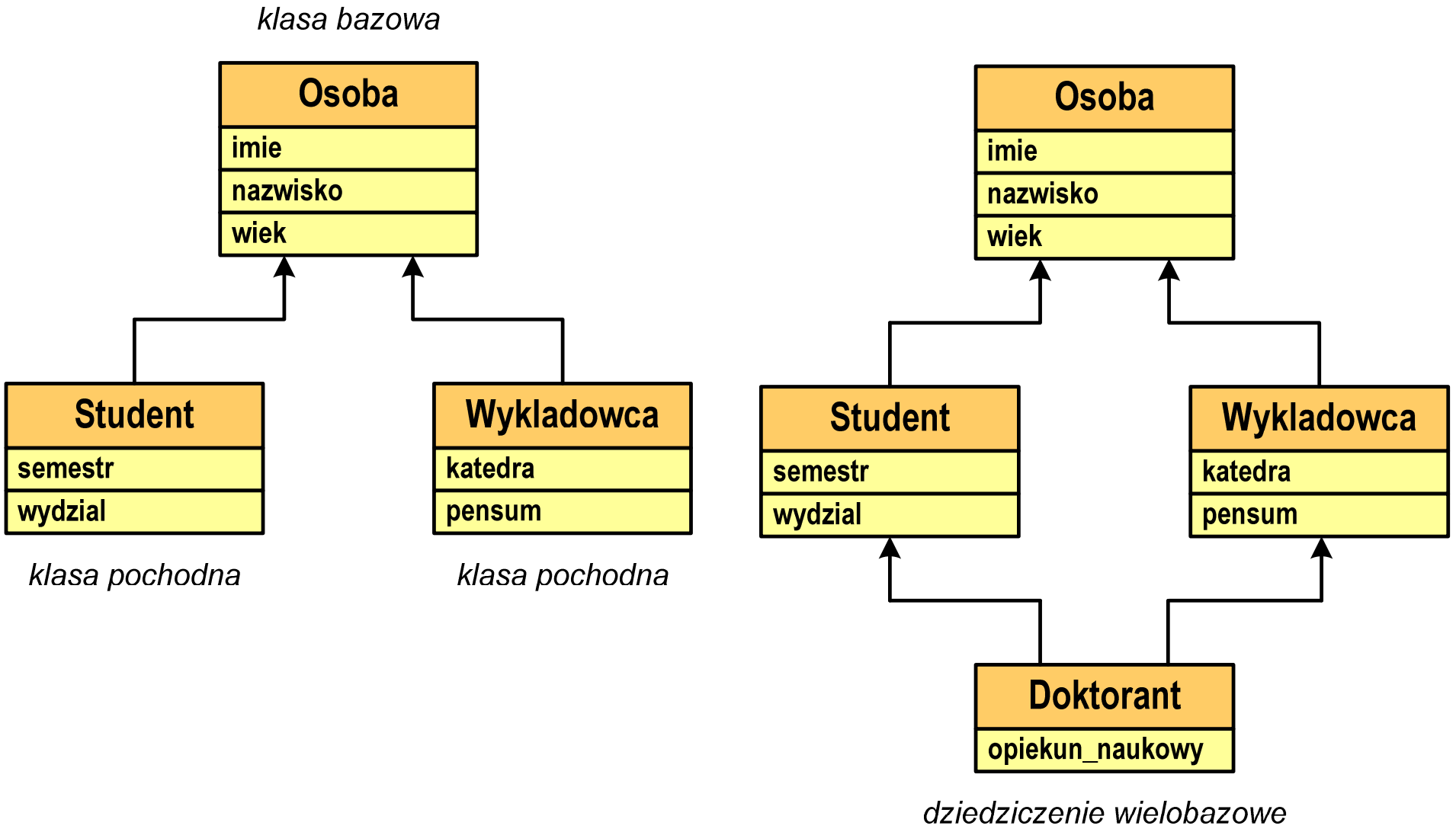
```
class Osoba:
    ...
    def dodaj_dziecko(self):
        self.dzieci += 1

nowak = Osoba("Jan","Nowak",25)
nowak.dodaj_dziecko()
print(nowak)
```

Python - prog. obiektowe (dziedziczenie)

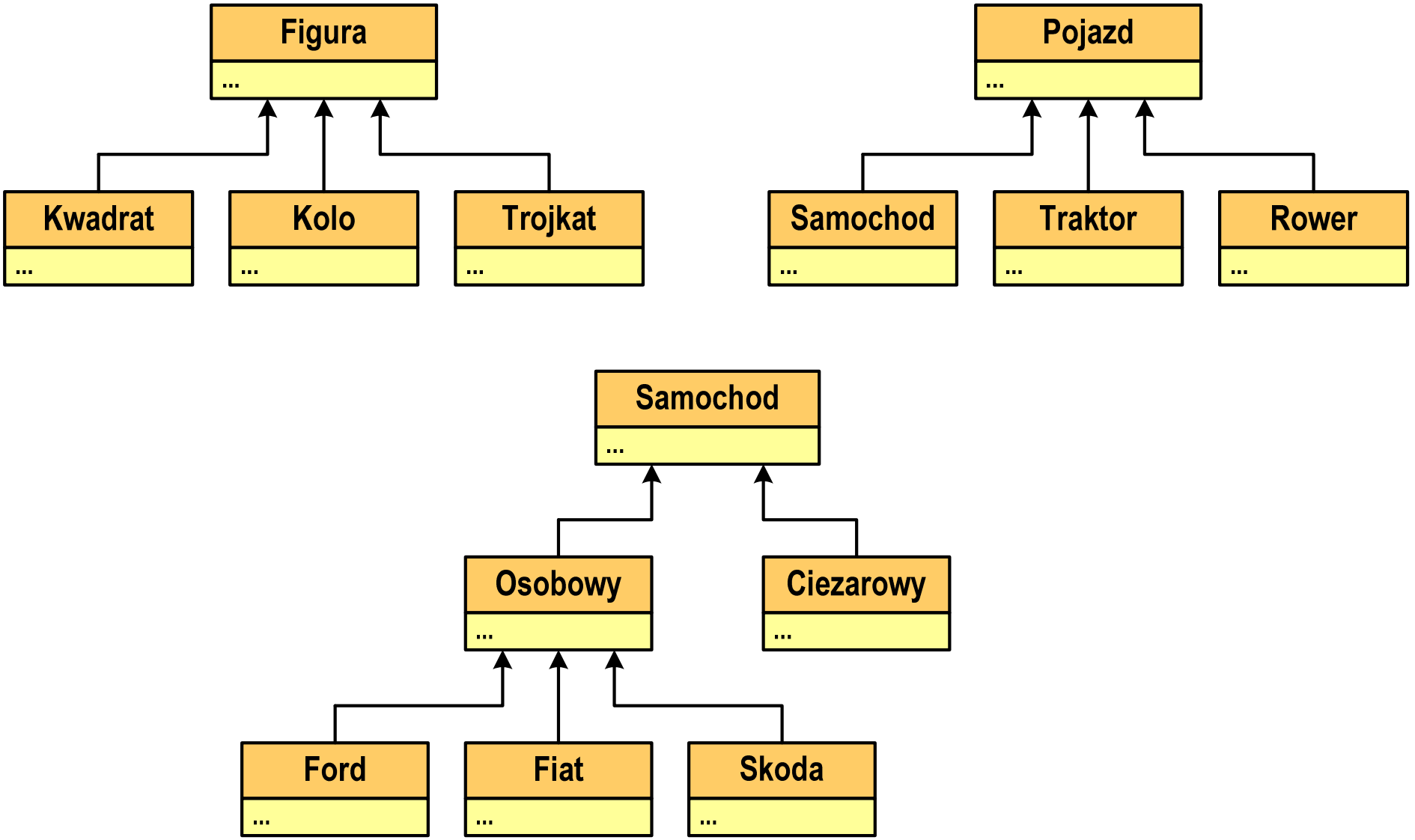
- dziedziczenie to technika, która pozwala na tworzenie nowych klas na podstawie klas już istniejących
- klasa, z której dziedziczą inne klasy, nazywana jest klasą bazową (podstawową, nadrzędną, superklasą)
- nowa klasa nazywa się klasą pochodną (potomną, subklasą)
- nowa klasa automatycznie dziedziczy atrybuty i metody klasy bazowej
- w klasie pochodnej można definiować również nowe atrybuty i metody



Python - prog. obiektowe (dziedziczenie)



Python - prog. obiektowe (dziedziczenie)



Python - prog. obiektowe (dziedziczenie)

```
class Osoba:
    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

    def __str__(self):
        return f"Osoba: {self.imie} {self.nazwisko}"

class Student(Osoba):
    def __init__(self, imie, nazwisko, semestr, wydział):
        super().__init__(imie, nazwisko)
        self.semestr = semestr
        self.wydział = wydział

    def __str__(self):
        return (f"Student: {self.imie} {self.nazwisko}, "
                f"semestr: {self.semestr}, wydział: "
                f"{self.wydział}")
```

Python - prog. obiektowe (dziedziczenie)

- klasa bazowa musi znajdować się w tym samym pliku, w którym tworzymy klasę pochodną
- nazwa klasy bazowej musi być umieszczona w nagłówku klasy pochodnej

```
class Student(Osoba):
```

- w metodzie __init__() klasy pochodnej wywołujemy metodę __init__() z klasy bazowej

```
super().__init__(imie, nazwisko)
```

- super() jest to wbudowana funkcja, która zwraca obiekt specjalny (zwany delegatem), umożliwiający dostęp do metod i atrybutów klasy bazowej z wnętrza klasy pochodnej
- super() stosowane jest do wywołania konstruktora lub metod klasy bazowej, nazwa funkcji pochodzi od superklasy

Python - prog. obiektowe (dziedziczenie)

- utworzenie obiektów i wywołanie metod

```
osoba = Osoba("Jan", "Kowalski")
student = Student("Anna", "Nowak", 3, "WE")

print(osoba)
print(student)

print(f"Imię: {student.imie}")
print(f"Nazwisko: {student.nazwisko}")
print(f"Semestr: {student.semestr}")
print(f"Wydział: {student.wydział}")
```

```
Jestem: Jan Kowalski
Student: Anna Nowak, semestr: 3, wydział: WE
Imię: Anna
Nazwisko: Nowak
Semestr: 3
Wydział: WE
```

Python - prog. obiektowe (dziedziczenie)

- istnieje możliwość nadpisania dowolnej metody klasy bazowej

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

    def opis(self):
        return f"Pojazd marki {self.marka}, model {self.model}"

class Samochod(Pojazd):
    def __init__(self, marka, model, rok):
        super().__init__(marka, model)
        self.rok = rok

    def opis(self):
        podstawowy_opis = super().opis()
        return f"{podstawowy_opis}, rok produkcji: {self.rok}"
```

Python - prog. obiektowe (dziedziczenie)

- metoda z klasy pochodnej przesłania widoczność metody z klasy bazowej

```
pojazd = Pojazd("Toyota", "Corolla")
samochod = Samochod("Ford", "Mustang", 2022)

print(pojazd.opis())
print(samochod.opis())
```

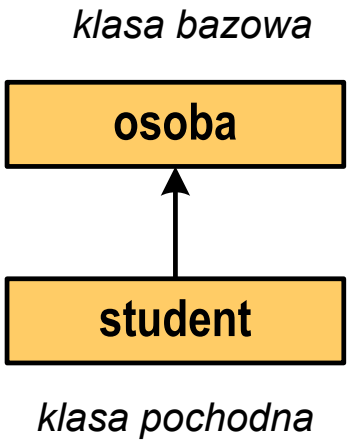
```
Pojazd marki Toyota, model Corolla
Pojazd marki Ford, model Mustang, rok produkcji: 2022
```

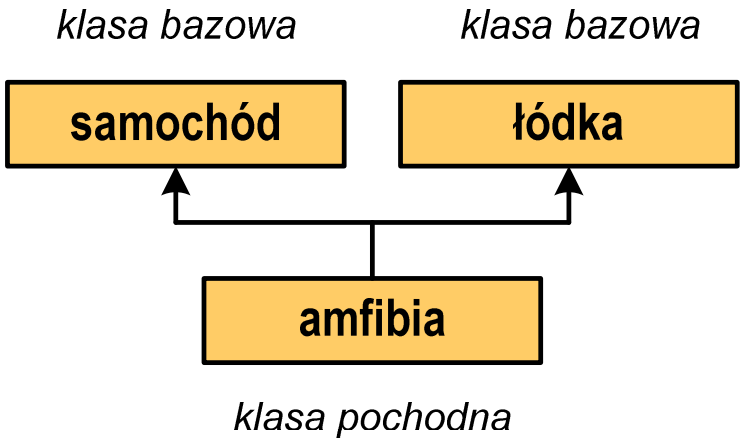
- korzystając z funkcji super() można odwołać się do metody z klasy bazowej

```
def opis(self):
    podstawowy_opis = super().opis()
    return f"{podstawowy_opis}, rok produkcji: {self.rok}"
```

Python - prog. obiektowe (dziedziczenie)

- dziedziczenie jednokrotne (single inheritance) - klasa pochodna dziedziczy po jednej klasie bazowej
- dziedziczenie wielokrotne (multiple inheritance) - klasa pochodna dziedziczy po więcej niż jednej klasie bazowej





Python - prog. obiektowe (dziedziczenie wielokrotne)

```
class A:
    def __init__(self):
        self.cecha_a = "Cecha z klasy A"

    def metoda_a(self):
        return "Metoda z klasy A"

class B:
    def __init__(self):
        self.cecha_b = "Cecha z klasy B"

    def metoda_b(self):
        return "Metoda z klasy B"

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        self.cecha_c = "Cecha z klasy C"

    def metoda_c(self):
        return "Metoda z klasy C"
```

Python - prog. obiektowe (dziedziczenie wielokrotne)

```
obiekt = C()
print(obiekt.cecha_a)
print(obiekt.cecha_b)
print(obiekt.cecha_c)

print(obiekt.metoda_a())
print(obiekt.metoda_b())
print(obiekt.metoda_c())
```

```
Cecha z klasy A
Cecha z klasy B
Cecha z klasy C
Metoda z klasy A
Metoda z klasy B
Metoda z klasy C
```

Python - prog. obiektowe (prawa dostępu)

- w Pythonie nie ma formalnych modyfikatorów dostępu, takich jak np. w języku C++ (public, protected, private)
- zamiast tego stosowane są konwencje nazewnicze i pewne mechanizmy do oznaczania poziomów dostępu do atrybutów i metod w klasach
- domyślnie wszystkie atrybuty i metody są publiczne - oznacza to, że są one dostępne z zewnątrz klasy; nie ma specjalnych prefiksów w nazwach

```
class Klasa:
    def __init__(self):
        self.publiczny_atrybut = "Dostęp publiczny"

    def publiczna_metoda(self):
        return "To jest metoda publiczna"

obiekt = Klasa()
print(obiekt.publiczny_atrybut)  # Dostęp publiczny
print(obiekt.publiczna_metoda())  # To jest metoda publiczna
```

Python - prog. obiektowe (prawa dostępu)

- atrybuty i metody chronione (protected) są oznaczane przez jeden znak podkreślenia _ na początku nazwy
- jest to konwencja wskazująca, że atrybut lub metoda nie powinny być używane na zewnątrz klasy lub jej podklas, choć technicznie jest to możliwe

```
class Klasa:
    def __init__(self):
        self._chroniony_atrybut = "Dostęp chroniony"

    def _chroniona_metoda(self):
        return "To jest metoda chroniona"

obiekt = Klasa()
print(obiekt._chroniony_atrybut)  # Dostęp możliwy, niezalecany
print(obiekt._chroniona_metoda())  # Dostęp możliwy, niezalecany
```

Python - prog. obiektowe (prawa dostępu)

- atrybuty i metody prywatne (private) są oznaczane przez dwa znaki podkreślenia __ na początku nazwy
- Python stosuje name mangling, aby utrudnić dostęp do tych elementów z zewnątrz klasy; Python zmienia nazwę atrybutu, dodając przedrostek _NazwaKlasy

```
class Klasa:
    def __init__(self):
        self.__prywatny_atrybut = "Dostęp prywatny"

    def __prywatna_metoda(self):
        return "To jest metoda prywatna"

obiekt = Klasa()
print(obiekt.__prywatny_atrybut)
print(obiekt.__prywatna_metoda())
```

Python - prog. obiektowe (prawa dostępu)

- atrybuty i metody prywatne (private) są oznaczane przez dwa znaki podkreślenia __ na początku nazwy
- Python stosuje name mangling, aby utrudnić dostęp do tych elementów z zewnątrz klasy; Python zmienia nazwę atrybutu, dodając przedrostek _NazwaKlasy

```
class Klasa:
    def __init__(self):
        self.__prywatny_atrybut = "Dostęp prywatny"

    def __prywatna_metoda(self):

obiekt
print
print
```

```
Traceback (most recent call last):
  File "d:\tempCodeRunnerFile.py", line 9, in <module>
    print(obiekt.__prywatny_atrybut)
          ^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'Klasa' object has no attribute
'__prywatny_atrybut'. Did you mean:
'_Klasa__prywatny_atrybut'?
```

Python - prog. obiektowe (prawa dostępu)

- można uzyskać dostęp do prywatnych atrybutów i metod przez name mangling, ale nie jest to zalecane

```
class Klasa:
    def __init__(self):
        self.__prywatny_atrybut = "Dostęp prywatny"

    def __prywatna_metoda(self):
        return "To jest metoda prywatna"

obiekt = Klasa()

# Name mangling:
print(obiekt._Klasa__prywatny_atrybut)
print(obiekt._Klasa__prywatna_metoda())
```

```
Dostęp prywatny
To jest metoda prywatna
```

Koniec wykładu nr 11

Dziękuję za uwagę!