Dziedziczenie

- Klasą bazową dla wszystkich klas Pythona jest klasa **object** (jeśli klasa jawnie nie dziedziczy po żadnej klasie, dziedziczy po klasie **object**).
- W Pythonie możliwe jest dziedziczenie wielokrotne.
- Python zapewnia kontrolę poprawności tworzonej hierarchii klas. Każda klasa może się w niej pojawić dokładnie raz, na końcu pojawia się też klasa **object**.
- Do ustalenia kolejności klas wykorzystywany jest algorytm C3.
- Kontrolę poprawności utworzonej hierarchii można wykonać korzystając z metody mro (method resolution order).

Przykład I

```
class A(object):
    pass

class B(A):
    pass

print(B.mro())
#[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

Przykład II

```
class A(object):
    def __init__(self):
        print('Init A')

class B(A):
    pass

B()
#Init A
```

Przykład III

```
class A:
    def __init__(self):
        print('Init A')

class B(A):
    def __init__(self):
        print('Init B')


B()                 #Init B
```

Przykład IV

```
class A:
    def __init__(self):
        super().__init__()
        print('Init A')

class B(A):
    def __init__(self):
        super().__init__()
        print('Init B')


B()                 #Init A
                    #Init B
```

Przykład V

```
class A:
    def __init__(self):
        super().__init__()
        print('Init A')

class B(A):
    def __init__(self):
        super().__init__()
        print('Init B')

#class C(A,B):
#TypeError: Cannot create a consistent method resolution order
#(MRO) for bases A, B

class C(B,A):
    def __init__(self):
        super().__init__()
        print('Init C')

print(C.mro())
#[<class '__main__.C'>, <class '__main__.B'>,
# <class '__main__.A'>, <class 'object'>]

C()
#Init A; Init B; Init C
```

Przykład VI

```
class A:
    def __init__(self):
        super().__init__()
        print('Init A')

class B(A):
    def __init__(self):
        super().__init__()
        print('Init B')

class C(A):
    def __init__(self):
        super().__init__()
        print('Init C')

class D(B,C):
    def __init__(self):
        super().__init__()
        print('Init D')

print(D.mro())      #[<class '__main__.D'>, <class '__main__.B'>,
                    # <class '__main__.C'>, <class '__main__.A'>,
                    # <class 'object'>]

D()                 #Init A; Init C; Init B; Init D
```

Przykład VII

```
class A:
    def __init__(self):
        super().__init__()
        print('Init A')
class B(A):
    def __init__(self):
        super().__init__()
        print('Init B')
class C:
    def __init__(self):
        super().__init__()
        print('Init C')
class D(C):
    def __init__(self):
        super().__init__()
        print('Init D')
class E(B,D):
    def __init__(self):
        super().__init__()
        print('Init E')

print(E.mro())  #[<class '__main__.E'>, <class '__main__.B'>,
                # <class '__main__.A'>, <class '__main__.D'>,
                # <class '__main__.C'>, <class 'object'>]

E()             #Init C; Init D; Init A; Init B; Init E
```

Zadanie

Zanim sprawdzicie Państwo uruchamiając program, proszę spróbować „na kartce”:

```
class F:
    pass
class E:
    pass
class D:
    pass
class C(D,F):
    pass
class B(E,D):
    pass
class A(B,C):
    pass

print(A.mro())
```

Klasa abstrakcyjna

Klasa definiuje zachowanie instancji. Metaklasa definiuje natomiast zachowanie klasy. Domyślnie metaklasą dla klas tworzonych w Pythonie jest klasa type, definiując nową klasę definiujemy nowy typ. Metaklasy wykorzystywane są przy tworzeniu klas abstrakcyjnych.

```
import abc   #abstract base class

class A(abc.ABC):
    @abc.abstractmethod
    def met(self):
        '''Metoda abstrakcyjna'''

class B(A):
    def met(self):
        '''Definicja metody abstakcyjnej'''

A().met()
#TypeError: Can't instantiate abstract class A
#with abstract methods met

B().met()
#OK
```

Metody statyczne

Oprócz metod instancji możemy definiować w klasie metody statyczne, przy czym w zależności od tego czy opatrzymy je dekoratorem czy nie, mogą one być wywoływane albo zarówno przez klasę, jak i przez instancję, albo tylko przez klasę.

```
class A:
    def met1(self):
        pass

    def met2():
        pass

    @staticmethod
    def met3():
        pass


A().met1()
#A().met2()
#TypeError: met2() takes 0 positional arguments but 1 was given
A().met3()
#A.met1()
#TypeError: met1() missing 1 required positional argument: 'self'
A.met2()
A.met3()
```