Programowanie
przed 7 wykładem

Andrzej Giniewicz

19.04.2024

Dzisiejsza porcja materiału dotyczy podstaw programowania z wykorzystaniem klas. Będzie to wstęp do programowania obiektowego.

1 Programowanie obiektowe

Programowanie obiektowe nie ma jednej ogólnie akceptowalnej definicji. Istnieje wiele różnych podejść do programowania obiektowego i wiele różnych języków, które na różne sposoby realizują te same cele. Różne są nie tylko sposoby realizacji celów, ale również nazewnictwo. Postaram się przybliżyć nieco ogólne cechy języków obiektowych, współdzielonych przez wiele języków, które są powszechnie uznawane za języki obiektowe. Przy braku definicji, będzie to najbliższe próbie określenia wspólnych cech większości takich języków.

Obiekty są podstawowym elementem budulcowym w programowaniu obiektowym. Obiekt to połączenie danych i zachowania tych danych w jeden byt. Dane w obiekcie przechowywane są w **atrybutach**, a zachowania, czyli czynności, które możemy wykonać na obiekcie, opisane są tak zwanymi **metodami**. Każdy obiekt musi mieć jakąś tożsamość, najczęściej reprezentowaną przez wartość lub adres w pamięci.

Przykładem obiektu może być punkt na płaszczyźnie. Będzie on miał przynajmniej dwa atrybuty — współrzędną X oraz współrzędną Y — może on mieć również szereg metod — na przykład „narysuj", „odbij względem prostej", i temu podobne.

To, jakie atrybuty i metody zdefiniujemy, zależy od nas jako twórców programu — w szczególności możemy zdefiniować inne atrybuty, takie jak kolor czy oznaczenie, albo inne metody, takie jak przesunięcie o wektor. Wszystko zależy od tego, do czego będziemy używać naszych punktów i co będzie nam potrzebne, aby wykonać zadanie. Warto też zaznaczyć, że nie ma jedynej słusznej implementacji lub reprezentacji. Niektóre osoby mogą wybrać parę atrybutów ze współrzędnymi, kto inny wybierze krotkę z parą współrzędnych. Obie implementacje są poprawne, jeśli dają tę samą funkcjonalność.

Znaczna część obiektowych języków programowania, choć nie wszystkie, pozwala tworzyć obiekty na podstawie **klas**. Klasa zawiera definicję wszystkich obiektów danej klasy, czyli to w klasie umieszczamy opis atrybutów oraz metod. Python jest językiem obiektowym opartym na systemie klas. Obok systemu klas w innych językach istnieją inne sposoby na

tworzenie obiektów — na przykład system prototypów jest obecny w języku JavaScript, system aktorów w języku Erlang lub system cech w języku Rust.

O klasie możemy myśleć jak o szablonie, według którego powstają nowe obiekty. Klasa pełni wtedy rolę typu danych. I tak, jeśli mamy klasę `Punkt`, możemy stworzyć dwa obiekty tej klasy, przykładowo nazwane `a` oraz `b`. Możemy powiedzieć, że `a` jest typu `Punkt` lub `a` jest klasy `Punkt`. W niektórych językach nie wszystkie typy są klasami, choć wszystkie klasy są typami. Jeśli istnieją typy, które nie są klasami, mówimy, że są typami prymitywnymi. W języku Python nie ma typów prymitywnych — wszystkie typy są klasami — zatem wszystkie wartości są obiektami i mogą mieć stowarzyszone z nimi metody. Wielokrotnie widzieliśmy metody, zarazem takie w obiektach definiowanych przez klasy, jak na przykład metoda `append` dla list, jak i metody na typach liczbowych. Wcześniej takie wywołania „z kropką" nazywaliśmy już metodami, teraz wiemy, skąd pochodzi ta terminologia.

W Pythonie `3.14` jest obiektem a `float` jest klasą. W klasie `float` zdefiniowana jest między innymi metoda `as_integer_ratio`, której używaliśmy wcześniej, więc

```
x = 3.14
print(x.as_integer_ratio())
```

jest działającym kodem. Dodatkowo — jeśli wcześniej mieliśmy styczność z innym językiem obiektowym, który ma typy prymitywne, na przykład z C++ — zapis taki może dziwić, ponieważ w C++ liczby nie są obiektami.

Jednym z celów programowania obiektowego jest oddzielenie implementacji od interfejsu, czyli **abstrakcja**. Użytkownik klasy powinien wiedzieć, jakie są metody w klasie. Jeśli otrzyma ich dokumentację, nie powinien musieć wiedzieć nic więcej na ich temat, aby ich używać. Przykładowo nie powinien musieć wiedzieć, czy punkt został zaimplementowany jako dwie liczby w osobnych atrybutach czy krotka w jednym atrybucie. Dzięki oddzieleniu implementacji od interfejsu można w każdej chwili zmienić implementację lub wewnętrzną reprezentację a użytkownicy klasy nie muszą wprowadzać zmian w kodzie tak długo, jak nie zmieni się interfejs — czyli metoda `narysuj` nazywa się `narysuj` i ma te same argumenty co wcześniej. Niezależnie od reprezentacji punktu, my korzystamy z opisanej w dokumentacji metody. Jest to bardzo cenna właściwość, gdy tworzymy biblioteki.

2 A niech to dunder świśnie, czyli rum i magia

Python w niektórych sytuacjach oczekuje od nas konkretnych nazw metod. Metody te nazywają się metodami magicznymi, ponieważ sprawiają, że obiekt zachowuje się w sposób taki, jak standardowe typy danych, podczas gdy nim nie jest. Na przykład, kiedy Python widzi operator `+`, w dużym uproszczeniu uruchamia metodę `__add__`, w której zaimplementowane jest dodawanie. Metody magiczne nazywamy też „dunder" — dunder nie jest tu dawnym polskim określeniem pioruna lub grzmotu, ani angielskim określeniem na pofermentacyjną pianę powstałą w wyniku destylacji rumu jako produkt uboczny, który również nazywany jest „dunder", ale jest skrótem od angielskiego „double underscore" czyli „podwójne podkreślenie".

Pierwszym napotkanym dundrem, czy też metodą magiczną, jest **konstruktor**. Konstruktor jest uruchamiany, gdy tworzymy obiekt. To on jest odpowiedzialny za ustawienie atrybutów obiektu. Konstruktor nazywa się `__init__`.

3 Pierwsza klasa

Wiemy już wszystko, co jest nam potrzebne, żeby stworzyć pierwszą klasę. Spróbujemy stworzyć klasę punktów na płaszczyźnie. Zaczynamy od definicji klasy, wewnątrz której będą metody. Metody piszemy jak funkcje, których pierwszym argumentem jest tożsamość obiektu[1]. Zwyczajowo pierwszy argument metod nazywamy `self`.

```
class Punkt:
    def __init__(self):
        print(”Tworzę punkt!”)

a = Punkt()
b = Punkt()
```

W kodzie powyżej tworzymy dwa punkty: `a` i `b`. Są to dwa obiekty klasy `Punkt`. Nie są to zbyt użyteczne obiekty, ponieważ nie mają żadnych atrybutów ani metod innych niż konstruktor. Zmieńmy je nieco tak, aby przejąć oraz zapamiętać współrzędne w atrybutach.

```
class Punkt:
    def __init__(self, x, y):
        self.wsp_x = x
        self.wsp_y = y

a = Punkt(0, 1)
b = Punkt(0, 0)
```

Konstruktor oprócz tożsamości obiektu przyjmuje tu dwa argumenty, nazwane `x` oraz `y`. Dla punktu `a` przekazujemy `x=0` oraz `y=1`, natomiast dla punktu `b` obie wartości są ustawione na zero. Konstruktor jedyne co robi, to zapamiętuje te wartości w atrybutach o odpowiedniej nazwie. Zwróćmy uwagę, że atrybuty nie były wcześniej zadeklarowane — dopiero podstawienie wartości pod atrybut w metodzie powoduje ich utworzenie.

Dodajmy teraz metodę `przesuń_o_wektor`.

```
class Punkt:
    def __init__(self, x, y):
        self.wsp_x = x
```

[1]Jest to w przybliżeniu prawda, na kolejnym wykładzie poznamy też metody klasy i metody statyczne, które mają inne zasady dotyczące pierwszego argumentu, niemniej jednak nie będą nam one potrzebne na tym etapie, zatem przeżyjemy z takim przybliżeniem rzeczywistości.

```
        self.wsp_y = y

    def przesuń_o_wektor(self, x, y):
        self.wsp_x += x
        self.wsp_y += y

a = Punkt(0, 1)
a.przesuń_o_wektor(3, 4)
```

Gdy wywołujemy kod `a.przesuń_o_wektor(3, 4)`, Python sprawdza, jakiej klasy jest obiekt `a` i znajduje metodę `przesuń_o_wektor` w klasie `Punkt`. Kod

```
a.przesuń_o_wektor(3, 4)
```

jest wobec tego równoważny do

```
Punkt.przesuń_o_wektor(a, 3, 4)
```

Po ustaleniu, z której klasy metodę należy wywołać, Python uruchamia kod metody z ustawionym obiektem `a` jako `self` i pozostałymi parametrami `x` i `y` ustawionymi na `3` i `4`. Wobec tego wykonywany kod to

```
a.wsp_x += 3
a.wsp_y += 4
```

Kod ten modyfikuje atrybuty obiektu `a`. Po ich wykonaniu obiekt `a` ma zmienioną wartość współrzędnych, jest to jednak wciąż ten sam obiekt. Oznacza to, że implementacja obiektu jest mutowalna. Niestety trudno zobaczyć, co znajduje się w obiekcie. Możemy co prawda wypisywać wartości

```
print((a.wsp_x, a.wsp_y))
```

ale wygodniej byłoby użyć po prostu `print(a)`. Aby temu zaradzić, dodajmy drugą magiczną metodę — metodę `__str__`. Metoda ta jest używana przy zamianie obiektu na napis, na przykład przez funkcję `print`. Metoda ta powinna zwracać napis a przyjmować jedynie tożsamość obiektu.

```
class Punkt:
    def __init__(self, x, y):
        self.wsp_x = x
        self.wsp_y = y
```

```
    def przesuń_o_wektor(self, x, y):
        self.wsp_x += x
        self.wsp_y += y

    def __str__(self):
        return f"({self.wsp_x}, {self.wsp_y})"

a = Punkt(0, 1)
print(a)
a.przesuń_o_wektor(3, 4)
print(a)
```

Lista metod magicznych jest długa i obszerna. Niemal każda operacja dziejąca się w Pythonie może zostać rozbudowana. Pełna lista metod magicznych, wraz z ich opisem i wymaganiami (co przyjmują, co zwracają, kiedy używa ich Python), znajduje się w dokumentacji — `https://docs.python.org/3/reference/datamodel.html#special-method-names`.

4 Dziedziczenie jednokrotne

Do przykładów zdefiniujmy klasę `Student`.

```
class Student:
    def __init__(self, indeks, imię, nazwisko):
        self.indeks = indeks
        self.imię = imię
        self.nazwisko = nazwisko
        self.email = f"{indeks}@student.pwr.edu.pl"

    def kto_to(self):
        return f"{self.imię} {self.nazwisko} <{self.email}>"
```

Obiekty klasy `Student` tworzymy podając nazwę klasy

```
student1 = Student(123456, "Jan", "Niekowalski")
student2 = Student(654321, "Janina", "Teżnienowak")
```

W kodzie powyżej stworzyliśmy dwa obiekty klasy `Student`. Każdy z obiektów ma swoje własne wartości atrybutów (są cztery atrybuty: `indeks`, `imię`, `nazwisko` oraz `email`). Na metodach tych można wywołać metodę `kto_to`.

```
print(student1.kto_to())
print(student2.kto_to())
```

Wywołania te wypiszą odpowiednie dane osobowe.

Użytkowanie klas w ten sposób zwykle skraca kod w miejscu jego używania, kosztem bardziej złożonej definicji (w porównaniu do podobnego rozwiązania wykorzystującego funkcje i krotki). Przejdziemy teraz do drugiego ważnego zastosowania programowania obiektowego — ponownego wykorzystania kodu za pomocą dziedziczenia.

Zajmiemy się najpierw najprostszą formą dziedziczenia, tak zwanym dziedziczeniem jednokrotnym (z jednej klasy). Wyobraźmy sobie, że oprócz klasy `Student` chcemy zaprogramować klasę `CzłonekSamorządu`. Możemy to zrobić bez dziedziczenia

```
class CzłonekSamorządu:
    def __init__(self, indeks, imię, nazwisko, funkcja):
        self.indeks = indeks
        self.imię = imię
        self.nazwisko = nazwisko
        self.email = f"{indeks}@student.pwr.edu.pl"
        self.funkcja = funkcja

    def kto_to(self):
        return f"{self.imię} {self.nazwisko} <{self.email}>"

    def zwołaj_zebranie(self):
        if self.funkcja == "Przewodniczący":
            print("Zwołuję zebranie")
```

Zauważmy, że sporo kodu jest identyczna z klasą `Student`, ponieważ każdy członek samorządu studenckiego jest studentem. Jeśli pomiędzy klasami istnieje relacja, którą możemy wypowiedzieć za pomocą słowa „jest" w sensie „bycia jednocześnie kimś/czymś innym", dziedziczenie jest przydatną techniką. Napiszemy

```
class CzłonekSamorządu(Student):
    def __init__(self, indeks, imię, nazwisko, funkcja):
        super().__init__(indeks, imię, nazwisko)
        self.funkcja = funkcja

    def zwołaj_zebranie(self):
        if self.funkcja == "Przewodniczący":
            print("Zwołuję zebranie")
```

Aby skorzystać z dziedziczenia, musimy w nawiasie napisać nazwę klasy, z której dziedziczymy. Klasa w nawiasie nazywa się rodzicem klasy. Jeśli skorzystamy z dziedziczenia, w nowej klasie będą znajdować się metody i atrybuty z rodzica. Niektóre metody możemy przeciążyć, czyli zdefiniować na nowo w klasie dziedziczącej. W przykładzie powyżej taką metodą jest `__init__`, który został odziedziczony z rodzica, ale również jest zdefiniowany w klasie dziedziczącej. W klasie dziedziczącej możemy użyć funkcji `super()`, aby odwołać się do metody zdefiniowanej w rodzicu. Dzięki temu konstruktor klasy `CzłonekSamorządu` nie musi

powtarzać kodu znajdującego się w klasie `Student`. Wywołuje on konstruktor rodzica, który ustawia atrybuty wykorzystywane przez wszystkich studentów, a następnie ustawia dodatkowy atrybut, w którym ustawia funkcję. Nie ma potrzeby definiowania metody `kto_to`, która jest dostępna w nowej klasie.

Sposób ten ma sporo zalet w porównaniu z konkurencyjnymi rozwiązaniami, takimi jak pisanie wszystkiego od początku:

- Tę samą funkcjonalność zaimplementowaliśmy w znacznie mniejszej liczbie linii, czyli pisząc mniej. Tym samym mamy mniej potencjalnych miejsc, gdzie możemy się pomylić.
- Nie mamy duplikatu kodu, więc gdy znajdziemy błąd w metodzie `kto_to`, musimy poprawić ją tylko w jednym miejscu w definicji klasy `Student`, nie w wielu.
- Jeśli rozbudowujemy funkcjonalność dla wszystkich studentów, nie musimy dodawać jej również we wszystkich pozostałych klasach, wystarczy dopisać ją w jednym miejscu.

Kolejną przydatną funkcjonalnością jest to, że możemy bardziej precyzyjnie sprawdzać typy obiektów. Wcześniej poznaliśmy jedynie funkcję `type`.

```
student1 = Student(123456, "Jan", "Niekowalski")
student2 = CzłonekSamorządu(654321, "Janina", "Teżnienowak", "Przewodniczący")

# Wyrażenie zwracające True
type(student1) is Student
type(student2) is CzłonekSamorządu


# Wyrażenie zwracające False
type(student1) is CzłonekSamorządu
type(student2) is Student
```

Zachowanie nie jest zgodne z rzeczywistością, ponieważ członek samorządu też jest studentem, niestety sprawdzanie za pomocą funkcji `type` nie daje takiej możliwości. Problem rozwiązuje funkcja `isinstance`, która uwzględnia dziedziczenie.

```
student1 = Student(123456, "Jan", "Niekowalski")
student2 = CzłonekSamorządu(654321, "Janina", "Teżnienowak", "Przewodniczący")

# Wyrażenie zwracające True
isinstance(student1, Student)
isinstance(student2, Student)
isinstance(student2, CzłonekSamorządu)

# Wyrażenie zwracające False
isinstance(student1, CzłonekSamorządu)
```

Oprócz funkcji `isinstance`, która służy do sprawdzania, czy obiekt jest instancją danej klasy, istnnieje funkcja `issubclass` sprawdzająca, czy klasa jest dzieckiem (inaczej podklasą) innej klasy (rodzica, inaczej nazywanego nadklasą).

```
student1 = Student(123456, ”Jan”, ”Niekowalski”)
student2 = CzłonekSamorządu(654321, ”Janina”, ”Teżnienowak”, ”Przewodniczący”)

# Wyrażenie zwracające True
issubclass(CzłonekSamorządu, Student)

# Wyrażenie zwracające False
issubclass(Student, CzłonekSamorządu)
```

5 Skąd się biorą domyślne metody

Można zadać sobie pytanie, skąd się biorą domyślne metody, takie jak implementacja metody `__str__`, gdy nie zdefiniujemy własnej ich implementacji? W języku Python wykorzystano rozwiązanie polegające na tym, że każda klasa, której nie podamy, z czego dziedziczy, dziedziczy automatycznie z klasy `object`[2]. Oznacza to, że nasza implementacja studenta z wcześniejszego przykładu

```
class Student:
    def __init__(self, indeks, imię, nazwisko):
        self.indeks = indeks
        self.imię = imię
        self.nazwisko = nazwisko
        self.email = f”{indeks}@student.pwr.edu.pl”

    def kto_to(self):
        return f”{self.imię} {self.nazwisko} <{self.email}>”
```

jest równoważna wersji z dziedziczeniem

```
class Student(object):
    def __init__(self, indeks, imię, nazwisko):
        self.indeks = indeks
        self.imię = imię
        self.nazwisko = nazwisko
        self.email = f”{indeks}@student.pwr.edu.pl”

    def kto_to(self):
        return f”{self.imię} {self.nazwisko} <{self.email}>”
```

To w klasie `object` zdefiniowane są wszystkie domyślne implementacje metod, których nie musimy ręcznie załączać lub implementować, dzięki opisanemu zachowaniu.

[2] `https://docs.python.org/3/library/functions.html#object`.

6 Dziedziczenie wielokrotne

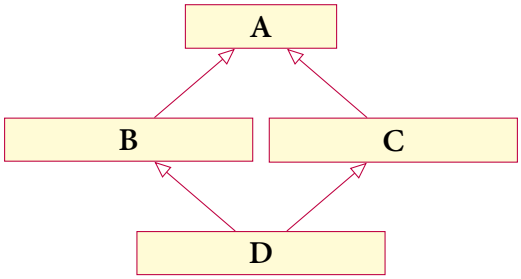
Python pozwala nie tylko na dziedziczenie z jednej klasy, ale z wielu jednocześnie. Gdyby wcześniej zdefiniować odpowiednie klasy figur geometrycznych, moglibyśmy określić kwadrat jako

```
class Kwadrat(Prostokąt, WielokątForemny):
    ...
```

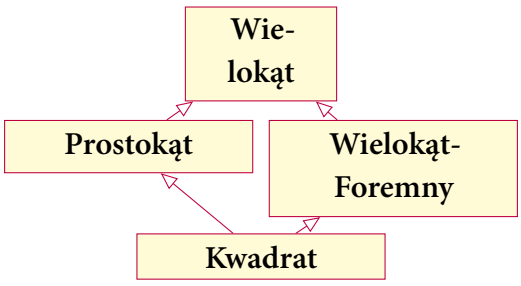
ponieważ kwadrat **jest** prostokątem oraz kwadrat **jest** wielokątem foremnym. Dzięki temu odziedziczymy wszystkie metody z obu klas, zawierające zapewne szereg przydatnych metod do wykonywania operacji na odpowiednich obiektach.

Języki programowania, które dopuszczają dziedziczenie z więcej niż jednej klasy, muszą opracować sposób radzenia sobie z trudną sytuacją, nazywaną w programowaniu „diamentem śmierci". Diament śmierci występuje, gdy mamy następującą sytuację:

- Mamy jakąś klasę `A`.
- Z klasy tej dziedziczą dwie klasy `B` i `C`.
- Klasa `D` dziedziczy z `B` i `C`.



Tego typu hierarchie klas są częstsze, niż mogłoby się wydawać. Na przykład



Co się dzieje, jeśli ta sama metoda jest zdefiniowana we wszystkich klasach? To, który jej wariant (z klasy `Prostokąt` czy `WielokątForemny`) zostanie wywołany jako nadrzędny dla klasy `Kwadrat`, nie jest zawsze jednoznaczne.

Różne języki mają różne strategie radzenia sobie z tym problemem. Spora część z nich zabrania wielokrotnego dziedziczenia. W Pythonie do radzenia sobie z wielokrotnym dziedziczeniem mamy MRO (ang. *Method Resolution Order*), czyli metodę określającą kolejność wywoływania metod.

7 Kolejność wywoływania metod

Kolejność wywoływania metod (MRO) to algorytm, który pozwala wyznaczyć linearyzację skomplikowanej hierarchii klas. Linearyzacja polega na ułożeniu klas, w których poszukiwane są metody, w liniowej kolejności. Dla klasy C jej linearyzacją będziemy nazywać listę klas zaczynającą się od C oraz jej uporządkowanych nadklas. Operator linearyzacji oznaczać będziemy $\mathcal{L}$. Operator ten jako argument przyjmuje klasę, a jako wynik zwracał będzie listę klas stanowiących jej linearyzację. Linearyzacja klasy `object` (była o niej mowa w sekcji 5) jest najprostsza, ponieważ nie ma ona żadnych nadklas.

$$\mathcal{L}(\text{object}) = [\text{object}].$$

Do opisu linearyzacji bardziej skomplikowanych klas, potrzebujemy więcej operacji i terminologii. Zacznijmy od nazewnictwa elementów listy.

Dla listy klas $[C_1, C_2, \ldots, C_n]$

- C_1 nazywamy głową listy,
- $[C_2, \ldots, C_n]$ nazywamy ogonem listy.

Dodanie nowej głowy do listy będziemy zapisywać za pomocą operatora +, czyli

$$C_1 + [C_2, \ldots, C_n] = [C_1, C_2, \ldots, C_n].$$

Główną częścią algorytmu jest funkcja $\text{merge}(L_1, \ldots, L_n)$ scalająca kilka linearyzacji w jedną. Działa ona następująco:

1. Wybierz głowę pierwszej linearyzacji.
2. Jeśli głowa wybranej linearyzacji znajduje się w ogonie dowolnej innej linearyzacji, wybierz kolejną linearyzację o ile to możliwe.
3. Jeśli nie ma takiej linearyzacji, dla której głowa nie występuje w ogonie żadnej innej linearyzacji, dziedziczenie nie jest możliwe i klasa nie zostanie zdefiniowana (algorytm zakończy się błędem).
4. Po wybraniu głowy z linearyzacji, należy dodać ją do wyniku, po czym wykreślić ze wszystkich linearyzacji $L_1, \ldots, L_n$.
5. Jeśli w linearyzacjach $L_1, \ldots, L_n$ zostały jakieś klasy, należy wrócić do kroku 1.

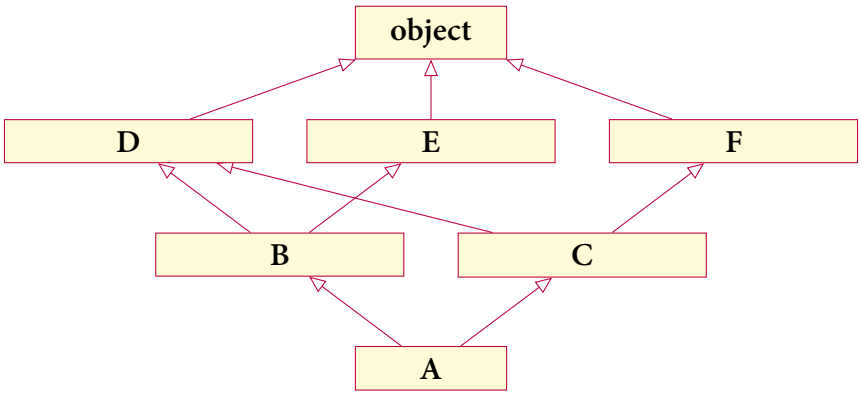
Wykorzystując wprowadzone oznaczenia, linearyzacja klasy C dziedziczącej z klas $B_1, \ldots, B_n$ może być zapisana jako

$$\mathcal{L}(C) = C + \text{merge}(\mathcal{L}(B_1), \ldots, \mathcal{L}(B_n), [B_1], \ldots, [B_n]).$$

Algorytm stanie się bardziej jasny po obserwacji przykładu. Kod

```
class F: pass

class E: pass

class D: pass

class C(D, F): pass

class B(D, E): pass

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

można zobrazować na rysunku jako



Na rysunku widać, że w hierarchii klas występuje więcej niż jeden diament śmierci. Rozpiszmy linearyzację poszczególnych klas:

object — jest to najprostsza linearyzacja, korzystamy jedynie z faktu, że klasa ta nie ma rodzica, zatem

$$\mathcal{L}(\text{object}) = [\text{object}].$$

D — klasa dziedziczy z `object`, zatem

$$\begin{aligned}\mathcal{L}(D) &= D + \text{merge}(\mathcal{L}(\text{object}), [\text{object}]) \\ &= D + \text{merge}([\text{object}], [\text{object}]).\end{aligned}$$

Musimy połączyć dwie listy w jedną. Wybieramy głowę pierwszej, czyli `object`. Nie ma jej w ogonach pozostałych, zatem wyciągamy ją z istniejących linearyzacji.

$$\mathcal{L}(D) = D + \text{object} + \text{merge}([], []).$$

Na liście pozostały jedynie puste listy, zatem kończymy scalanie i gotowy wynik, to

$$\mathcal{L}(D) = D + [\text{object}] = [D, \text{object}].$$

E, F — klasy te mają linearyzacje analogiczne do klasy D, czyli

$$\mathcal{L}(E) = [E, \text{object}]$$

oraz

$$\mathcal{L}(F) = [F, \text{object}]$$

C — korzystamy ze wzoru na linearyzację

$$\begin{aligned}\mathcal{L}(B) &= B + \text{merge}(\mathcal{L}(\text{D}), \mathcal{L}(\text{E}), [D], [E]) \\ &= B + \text{merge}([D, \text{object}], [E, \text{object}], [D], [E]).\end{aligned}$$

Musimy wykonać scalenie czterech list z linearyzacją w jedną. Głowa pierwszej listy to D. Nie ma jej w ogonie żadnej innej listy, zatem wyciągamy ją do wyniku

$$\mathcal{L}(B) = B + D + \text{merge}([\text{object}], [E, \text{object}], [], [E]).$$

Głowa pierwszej listy to `object`. Jest on w ogonie drugiej listy, zatem nie możemy wyciągnąć `object` jako kolejny element listy. Patrzymy zatem na głowę drugiej listy. Jest to E, którego nie ma w ogonie innych list. Wobec tego

$$\mathcal{L}(B) = B + D + E + \text{merge}([\text{object}], [\text{object}], [], []).$$

Głowa pierwszej listy to `object`. Tym razem nie ma tej klasy w ogonie żadnej listy.

$$\mathcal{L}(B) = B + D + E + \text{object} + \text{merge}([], [], [], []).$$

Wszystkie pozostałe listy są puste, zatem możemy zapisać wynik

$$\mathcal{L}(B) = B + [D, E, \text{object}] = [B, D, E, \text{object}].$$

C — podobnie jak dla B, mamy

$$\mathcal{L}(C) = [C, D, F, \text{object}].$$

A — w końcu możemy zapisać linearyzację klasy A.

$$\begin{aligned}\mathcal{L}(A) &= A + \text{merge}(\mathcal{L}(\text{B}), \mathcal{L}(\text{C}), [B], [C]) \\ &= A + \text{merge}([B, D, E, \text{object}], [C, D, F, \text{object}], [B], [C]) \\ &= A + B + \text{merge}([D, E, \text{object}], [C, D, F, \text{object}], [], [C]) \\ &= A + B + C + \text{merge}([D, E, \text{object}], [D, F, \text{object}], [], []) \\ &= A + B + C + D + \text{merge}([E, \text{object}], [F, \text{object}], [], []) \\ &= A + B + C + D + E + \text{merge}([\text{object}], [F, \text{object}], [], []) \\ &= A + B + C + D + E + F + \text{merge}([\text{object}], [\text{object}], [], []) \\ &= A + B + C + D + E + F + \text{object} + \text{merge}([], [], [], []) \\ &= A + [B, C, D, E, F, \text{object}] = [A, B, C, D, E, F, \text{object}].\end{aligned}$$

Oznacza to, że linearyzacja klasy A jest możliwa, czyli kod jest prawidłowy. Możemy sprawdzić linearyzację prawidłowego kodu w Pythonie za pomocą atrybutu `__mro__`. Uruchomienie kodu

```
class F: pass

class E: pass
```

```
class D: pass

class C(D, F): pass

class B(D, E): pass

class A(B, C): pass

print(A.__mro__)
```

daje w wyniku

```
(<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__mai
    n__.D'>, <class '__main__.E'>, <class '__main__.F'>, <class 'object'>)
```

8 Super `super()`

Wywołanie `super().metoda(...)` wewnątrz definicji klasy powoduje, że metoda o wskazanej nazwie będzie poszukiwana w klasach według MRO, od lewej do prawej. Wywołana zostanie pierwsza napotkana w tej kolejności metoda. W przypadku, gdy metody nie będzie w żadnej z klas, zwrócony zostanie błąd. Zastosowanie MRO pozwala na rozwiązanie problemu z diamentem śmierci, ponieważ określamy jednoznaczną kolejność wywołań metod, co pozwala uniknąć niezdefiniowanego zachowania kodu.